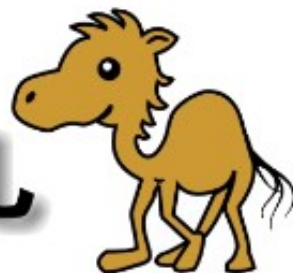




# Perl 學習手札



There is more than one way to do it.

目錄：

## [0. 關於本書](#)

### [1. 關於Perl](#)

- 1.1 Perl的歷史
- 1.2 Perl的概念
- 1.3 特色
- 1.4 使用Perl的環境
- 1.5 開始使用 Perl
- 1.6 你的第一支Perl程式

### [2. 純量變數\(Scalar\)](#)

- 2.1 關於純量
  - 2.1.1 數值
  - 2.1.2 字串
  - 2.1.3 數字與字串轉換
- 2.2 使用你自己的變數
- 2.3 賦值
  - 2.3.1 直接設定
  - 2.3.2 還可以這樣
- 2.4 運算
- 2.5 變數的輸出/輸入
- 2.6 Perl預設變數
- 2.7 defined 與 undef

### [3. 串列與陣列](#)

- 3.1 何謂陣列
- 3.2 Perl 的陣列結構
- 3.3 push/pop
- 3.4 shift/unshift
- 3.5 切片
- 3.6 陣列還是純量？
- 3.7 一些常用的陣列運算
  - 3.7.1 sort
  - 3.7.2 join
  - 3.7.3 map
  - 3.7.4 grep

### [4. 基本的控制結構](#)

- 4.1 概念
  - 4.1.1 關於程式的流程
  - 4.1.2 真，偽的判斷
  - 4.1.3 區塊
  - 4.1.4 變數的生命週期

- 4.2 簡單判斷
  - 4.2.1 if
  - 4.2.2 unless
  - 4.2.3 一行的判斷
  - 4.3.4 else/elsif
- 4.3 重複執行
  - 4.3.1 while
  - 4.3.2 until
- 4.4 for
  - 4.4.1 像 C 的寫法
  - 4.4.2 其實可以用 ...
  - 4.4.3 有趣的遞增/遞減算符
  - 4.4.4 對於陣列內的元素

## [5. 雜湊\(Hash\)](#)

- 5.1 日常生活的雜湊
- 5.2 雜湊的表達
- 5.3 雜湊賦值
- 5.4 each
- 5.5 keys跟values
- 5.6 雜湊的操作
  - 5.6.1 exists
  - 5.6.2 delete
- 5.7 怎麼讓雜湊上手

## [6. 副常式](#)

- 6.1 關於Perl的副常式
- 6.2 參數
- 6.3 傳回值
- 6.4 再談參數
- 6.5 副常式中的變數使用

## [7. 正規表示式](#)

- 7.1 Perl 的第二把利劍
- 7.2 甚麼是正規表示式
- 7.3 樣式比對
- 7.4 Perl 怎麼比對
- 7.5 怎麼開始使用正規表示式

## [8. 更多關於正規表示式](#)

- 8.1 只取一瓢飲
- 8.2 比對的字符集合
- 8.3 正規表示式的特別字元
- 8.4 一些修飾字元
- 8.5 取得比對的結果
- 8.6 定位點
- 8.7 比對與替換
- 8.8 有趣的字串內交換
- 8.9 不貪多比對
- 8.10 如果你有疊字
- 8.11 比對樣式群組
- 8.12 比對樣式的控制

## [9. 再談控制結構](#)

- 9.1 迴圈操作

- 9.1.1 last
- 9.1.2 redo
- 9.1.3 next
- 9.1.4 標籤
- 9.2 switch
- 9.2.1 如果你有複雜的 if 敘述
- 9.2.2 利用模組來進行
- 9.3 三元運算符
- 9.4 另一個小訣竅

## [10. Perl的檔案存取](#)

- 10.1 檔案代號 (FileHandle)
- 10.2 預設的檔案代號
- 10.3 檔案的基本操作
  - 10.3.1 開檔/關檔
  - 10.3.2 意外處理
  - 10.3.3 讀出與寫入

## [11. 檔案系統](#)

- 11.1 檔案測試
- 11.2 重要的檔案相關內建函式
- 11.3 localtime

## [12. 字串處理](#)

- 12.1 簡單的字串形式
- 12.2 uc 與 lc
- 12.3 sprintf
- 12.4 排序
- 12.5 多字鍵排序

## [13. 模組與套件](#)

- 13.1 關於程式的重用
- 13.2 你該知道的 CPAN
- 13.3 使用CPAN與CPANPLUS
- 13.4 使用模組
- 13.5 開始寫出你的套件

## [14. 參照 \(Reference\)](#)

- 14.1 何謂參照
- 14.2 取得參照
- 14.3 參照的內容
- 14.4 利用參照進行二維陣列

## [15. 關於資料庫的基本操作](#)

- 15.1 DBM
  - 15.1.1 與DBM連繫
  - 15.1.2 DBM檔案的操作
  - 15.1.3 多重資料
- 15.2 DB\_File
- 15.3 DBI
- 15.4 DBIx::Password

## [16. 用Perl實作網站程式](#)

- 16.1 CGI
- 16.2 Template

## 16.3 Mason

## [17. Perl與系統管理](#)

17.1 Perl在系統管理上的優勢

17.2 Perl的單行執行模式

17.3 管理檔案

17.4 郵件管理

17.4.1 Mail::Audit + Mail::SpamAssassin

17.4.2 Mail::Sendmail 與 Mail::Bulkmail

17.4.3 POP3Client 及 IMAPClient

17.5 日誌檔

17.6 報表

## [附錄A. 習題解答](#)



# Perl 學習手札



There is more than one way to do it.

## 關於作者：

簡信昌

「傲爾網」專案經理

博仲法律事務所資訊部門

台北Perl 推廣組 (Taipei.pm) 召集人

Newzilla 召集人

## 目前專案：

Open Source Foundry (OSSF)

Newzilla 線上雜誌

## 著作：

「獨當一面」 (共同創作)

「Blog 部落格線上出版，網路日誌實作」 (共同創作)

「Perl 學習手札」

## 參與翻譯：

Learning Perl 3rd

Network programming with Perl

Effective Perl Programming

Absolute BSD

## 主辦：

「2003 Perl, PHP, Python Party」

「Perl Training」

「[YAPC: 2004](#)」

「OSX Workshop」

## 公開文章：

2003. 10 科技月刊(HopeNet) - Blog

2004. 5 資訊傳真(PC Magazine) - Perl 簡介

## 其他：

宜蘭縣網中心研習講師

臺北市立興德國小研習講師

## 目前使用：

FreeBSD 4.x

Mac OS 10.3

Perl 5.8.x

Subversion 1.0.x

## 相關工作內容：

資訊系統分析

程式設計

相關文章/專欄/書籍撰寫

系統規劃，架設

教學，演講

## 聯絡：

MSN: hcc129@hotmail.com

E-Mail: hcchi en@hcchi en.org

## 關於本書：

[Perl 學習手札全文](#)



[相關討論](#)

紙板發行：[上奇科技](#)

網路版：所有內容採用 [Creative Commons](#) 授權

網路版所有捐款，贊助將全數捐給[台北 Perl 推廣組](#)

贊助方式：

ATM: 合作金庫 (006) 0969872815598

PayPal : [hcchi en@hcchi en. org](mailto:hcchi en@hcchi en. org)



# Perl 學習手札



There is more than one way to do it.

## 1. 關於Perl

當你翻開這本書的時候，你也就進入了一個奇幻的世界。Perl確實是一種非常吸引人的程式語言，而之所以這麼引人入勝的原因不單單在於他的功能，也在於他寫作的方式，或說成為一種程式寫作的藝術。即使你只是每天埋首於程式寫作的程式設計師，也不再讓生活過份單調，至少你可以嘗試在程式碼中多一些變化。而且許多Perl的程式設計師已經這麼作了，這也是Perl的理念-「There is more than one way to do it」。

常常遇到有人問我：「Perl到底可以拿來作甚麼呢？」，不過後來我慢慢的發現，這個問題的答案卻是非常的多樣化。因為在不同的領域幾乎都有人在使用Perl，所以他們會給你的答案就會有很大的差異了。有人會覺得Perl拿來用在生物資訊上真是非常方便，有人也來進行語料的處理，資料庫，網頁程式設計更是有著廣泛的運用。當然，還有許多人把Perl拿來當成系統管理的利器，更是處理系統日誌的好幫手。

### 1.1 Perl的歷史

由Larry Wall創造出來的Perl在1987年時最早出現在usenet的新聞群組comp.source。從當時所釋出的1.0版本，到3.0版為止，幾乎維持著一年有一次大版本的更新，也就是說在1989年時，Perl已經有了3.0版。而1991年，Perl開發團隊發展出相當關鍵的4.0版。因為隨著4.0版的釋出，Perl發表了新的版權聲明，也就是Perl Artistic Licence(藝術家授權)。Perl4跟Perl5之間相隔了有三年之久，漸漸的，Perl的架構已經日趨穩定。一直到最近，Perl釋出了新的Perl 5.8版，並且同時進行新一代版本的開發。

### 1.2 Perl的概念

Perl是非常容易使用的程式語言，或者我們應該說他是方便的程式語言，你可以隨手就寫完一支Perl的程式，就像你在命令列中打一個指令一樣(註一)。因為Perl的誕生幾乎就在於讓使用者能夠以更好方便的方式去撰寫程式碼，卻不必像寫C一樣的考慮很多細節。

另外，Perl的黏性非常的強(註二)，你可以用Perl把不同的東西輕易的連接起來。而且你可以用Perl解決你大部份的問題，雖然有些時候你並不想這麼作，但這並不表示Perl作不到。

### 1.3 特色

很多人對Perl的印象就是一種寫CGI(註三)的程式語言，或者直覺的認為Perl只是拿來處理文字的工具。不過就像我們所說的，Perl幾乎可以完成大部份你希望達成的工作。但是不可否認的，正規表示式顯然是Perl足以傲人的部份，這也就是Perl大量被拿來使用作為文字處理的原因之一。

而且Perl對於你希望快速的完成某些工作確實可以提供非常大的幫助。甚至在Unix-like的環境下，還可以直接使用Perl為基礎的Shell，讓你用Perl當指令。而不必像許多程式語言，在還沒正式工作之前，你必須先準備一堆事情，包括你的變數定義，你的資料結構等等。也因此，許多Unix-like系統的管理員都喜歡拿Perl來進行系統管理。畢竟沒人希望要處理一個郵件紀錄檔還要先花一堆時間搞清楚該怎麼把紀錄檔內的東西轉成合適的資料結構。

### 1.4 使用Perl的環境

雖然大多數的Unix-like系統管理員選擇Perl來幫助他們管理他們的伺服器，可是這絕對不表示Perl只能在這些系統上執行。相反的，Perl可以在絕大多數的作業系統上執行。而目前Windows上的Perl則是以Active Perl(<http://www.activeperl.com>)這家公司所提供的直譯器為主。Perl的使用在不同的作業系統下會略有不同，本書則以unix-like為執行環境。

### 1.5 開始使用 Perl



在開始使用Perl之前，必須先確定你的機器上是否已經安裝Perl。在許多unix-like的作業系統中，都預設會安裝Perl，你也可以先執行下面的指令來確定目前系統內的Perl版本。

```
[hcchi en@Appl e]% perl -v

This is perl, v5.8.2 built for darwin-2level

Copyright 1987-2003, Larry Wall

Perl may be copied only under the terms of either the Artistic License or the
GNU General Public License, which may be found in the Perl 5 source kit.

Complete documentation for Perl, including FAQ lists, should be found on
this system using `man perl' or `perldoc perl'. If you have access to the
Internet, point your browser at http://www.perl.com/, the Perl Home Page.
```

我們看到在版本的部份，這裡使用的是Perl 5.8.2的版本，然後有著作者Larry Wall的名字，也就是版權擁有者。接下來是Perl的版權說明。另外，你應該要注意`perldoc perl' 這個部份：直接在你的終端機下打這行指令，就可以看到Perl內附的文件，而且內容非常詳細。

在這裡，我們建議使用Perl 5.8以上的版本，如果你的版本過於老舊，或是系統中還沒有安裝Perl，可以從<http://www.perl.com/>下載，並且安裝Perl。

如果你的系統已經有Perl，並且正常運作，那麼你可以開始使用你的Perl。你可以試著使用所有程式語言都會使用的範例來作為使用Perl的開端：

```
[hcchi en@Apple]% perl -e 'print "hello world!\n"'
hello world!
```

不過在Windows上，因為命令列不能使用單引號，所以得這樣寫：

```
[hcchi en@Apple]% perl -e "print \"Hello world\n\""
hello world!
```

## 1.6 你的第一支Perl程式

事實上，你剛剛已經有了你的第一支Perl程式。當然，你可以不承認那是一支Perl程式。不過讓我們真正來寫一支程式吧。如果你習慣於使用許多整合性程式開發工具，你大概會希望知道要安裝甚麼樣的工具來寫Perl。不過你可能要失望了，因為我們全部所需要的就只是一個文字編輯器。你在unix上，可以選擇vi (vim)，joe或任何你習慣的編輯器，在Windows上可以使用記事本，或下載UltraEdit(<http://www.ultraedit.com/>)。不過請不要使用類似Word的這樣的文書處理工具，因為這樣子你只是讓事情更複雜了。當然，即使你在Windows上，你還是可以選擇Vim或是Emacs這些在Unix世界獲得高度評價的文字編輯器，而且他們還是自由軟體。

現在，我們可以打下第一支程式了：

```
#!/usr/bin/perl

print "hello world\n";
```

相信大家很快就打完了這支程式。先別管裡面到底說了甚麼（雖然妳們應該都看懂了），我們先來執行他



吧！

```
[hcchi en@Appl e]% perl ch1.pl  
hello world
```

好極了，結果就像我們直接用命令列執行的樣子。不過至少我們知道了，只要用Perl去執行我們寫出來的程式就可以了，當然，你還可以有更簡單的辦法。你可以讓你的檔案變成可以執行，在Unix下，你只需要利用chmod來達成這樣的目的。當然，我們假設你已經可以操作你的系統，至少能夠了解檔案權限。修改完權限之後，你只需要在檔案的所在目錄打：

```
[hcchi en@Appl e]% ./ch1.pl  
hello world
```

那麼第一行又是甚麼意思呢？其實這是Unix系統中，表明這支程式該以甚麼方式執行的表達方式。在這裡，我們希望使用"/usr/bin/perl"這個程式來執行。所以請依照你系統內的實際狀況適時改寫。否則當你在執行的時候，很可能會看到"Command not found"之類的錯誤訊息。

不過在真正開始寫Perl之前，我們還要提醒幾件事情，這些事情對於你要開始寫Perl的程式是非常的重要的。

1. Perl的敘述句是用分號(;)隔開的，因此只要你的敘述句還沒出現分號，Perl就不會把他當成一個完整的結束，除非你的這個敘述句是在一個區塊的最後一句。我們可以在perl doc裡面找到這樣的範例；

```
print  
"hello world\n"  
;
```

而且這樣的寫法對 Perl來說並沒有什麼不同，只是對於需要維護你的程式的人來說顯然並不會特別高興。適時的空白確實可以提高程式的可讀性，不過記得不要濫用，造成自己遭受埋怨。

2. Perl是以井字號(#)作為程式的註解標示，也就是只要以井字號開始，到敘述句結束前的內容都會被當成程式註解，Perl並不會嘗試去執行他，或編譯他。對於有些習慣於C程式寫作的程式員而言，能夠使用(/\* ... \*/)來進行程式的註解確實是相當方便的。Perl並沒有正式的定義方式來進行這樣整個區塊的註解，不過卻可以利用其他方式來達到同樣的目的。例如使用pod(plain-old documentation format，簡明文件格式)：

```
#!/usr/bin/perl  
  
print "hello world\n";  
  
=head1  
這裡其實是註解，所以也是很方便的  
主要是可以一次放很多行註解  
=cut
```

如果你還想找出其他可能的替代方案，可以直接看perlfaq這份文件，而方法就是直接執行perl doc perlfaq (註四)就可以了。

接下來，我們便要真正進入Perl的環境中了。

習題：

1. 試著找出你電腦上的Perl版本為何。

2. 利用perldoc perl找出所有的perl文件內容
3. 利用Perl寫出第一個程式，印出你的名字

註一：事實上，Perl有所謂的單行模式，你就只需要在命令列中執行Perl的敘述句。

註二：因此也有人戲稱Perl是「膠水程式語言」。

註三：就是所謂的「Common Gateway Interface」，動態網站程式的設計界面。

註四：perldoc裡有著許多非常有用的文件，你可以考慮試著看看perldoc perldoc。



# Perl 學習手札



There is more than one way to do it.

## 2. 純量變數(Scalar)

在Perl的世界裡，變數其實是以非常簡單的形式存在。至少比起你必須記憶一大堆int，char等等的資料形態算是方便許多了。對於所有只需要儲存單一變數值的資料結構，在Perl裡面都是使用純量數值來進行。所以你在寫Perl的時候不需要去考慮你的某個變數是要儲存數字或字串，大多數的時候你也不需要煩惱著要進行數字與字串間的轉換(註一)。

### 2.1 關於純量

一般的資料型態中，大多就是數值與字串兩種型態。當然，其中的數值也還可以分成整數型態跟浮點數型態，字串則是由一個或多個字元組合而成。在許多程式語言中，對這方面的定義非常的嚴格，你不但要考慮程式的流程，還必須隨時注意是否該對你的變數進行資料型態的轉換。不過在Perl中，對於所有這一類型的變數一律一視同仁，因此不論你所要儲存的是數值或字串，在Perl中都只需要使用純量變數。

其實就和我們現實生活中相當接近，我們在使用自然語言時，並不會特別去聲明接下來要使用甚麼樣的描述或解釋，而大多數是取決於當時的語境。而Larry Wall既然是一位自然語言學家，顯然對於這一方面特別在行，而這也造就了Perl能夠以非常接近口語的方式表達電腦程式語言的重要原因。

#### 2.1.1 數值

數值對於Perl的意義在於「整數值」跟「浮點數值」，Perl其實對於數字的看待方式是以倍精度的方式去運算，對於目前大多數的系統而言，這樣的精確度顯然可以應付大部份的需求。否則許多券商都使用Perl進行系統開發，甚至太空科學或DNA運算也都大量的使用Perl，難道他們會想拿石頭砸自己的腳？何況，如果需要的話，Perl也可以支援精確度無限的BigNum(「大數」)運算。

你可以很容易的在Perl中使用數值，不論是整數或浮點數。例如下面都是非常典型的數值表達方式：

```
1
1.2
1.0
1.2e3
-1
-1.2
-1e3
```

其中的1.2e3是表示1.2乘以10的三次方，這也許在科學或數學上的使用比較多，不過你多知道一些也是有幫助的。

另外，也許你希望使用類似1,300,000來清楚的顯示一個數值的長度，很可惜，這在Perl中會造成誤解，當然也不會達到你的要求。如果你真的期待以區隔的方式來表現數字的長度，你可以嘗試使用1\_300\_300的形式。不過我個人並不經常使用，也許我處理數字長度都還不足以使用這樣的表達方式。你可以看看在Perl裡面數值的表達方式：

```
#!/usr/local/bin/perl

$a = 1_300_300;
$b = 1.3e6;

print "$a\n";
print "$b\n";
```

執行結果會像這樣

```
1300300
1300000
```

你也許會希望使用非十進位的表示方式，例如十六進位的數字就可以在你撰寫網路相關程式的時候提供相當的幫助。這時候，Perl會有一些特殊的方式來幫助你完成這樣的工作。例如：

```
#!/usr/local/bin/perl

$a = 0266;
$b = 0xff;

print "$a, $b\n";
printf "%lo, %lx\n", $a, $b;
```

我們可以看到結果就會像是：

```
182, 255
266, ff
```

其中，printf 是以格式化的形式列印，我們稍後會提到。在這裡，我們只要知道這樣的方式可以印出八進位跟十六進位的數字就可以了。

### 2.1.2 字串

在純量的資料形態中，除了數字，另一個重要的部份則是字串。事實上，Perl的純量值就不外乎這兩種資料型態。在Perl中使用字串也非常的容易，你只需要在所表達的字串前後加上一對的單引號或雙引號就可以了。

```
#!/usr/local/bin/perl

$a = "perl";
$b = 'perl';
$c = "在\tPerl\t中使用字串非常容易";
$d = '在\tPerl\t中使用字串非常容易';
$e = "23";
$f = "";

print "$a\n";
print "$b\n";
print "$c\n";
print "$d\n";
print "$e\n";
print "$f\n";
```

於是可以印出結果：

```
perl
perl
在    Perl    中使用字串非常容易
在\tPerl\t中使用字串非常容易
23
```

在這裡，我們看到一些比較有趣的東西。也是在Perl裡面對字串處理的一些特性：

1. Perl雖然可以使用單引號或雙引號來表示字串，但是兩者的使用卻有些許不同。像我們在\$*c*跟\$*d*兩個變數中所看到的一樣，兩個字串表面上看起來一樣，但是卻因為使用了單引號跟雙引號的差別，使得兩個出來的結果就產生了差異。因為在Perl的單引號中，是無法使用像\n（換行字元），\t（跳格字元）這些特殊字元的，因此在單引號中的這些字元都會被忠實的呈現。就是我們在變數\$*d*的結果所看到的。而這些特殊字元還包括了：

```
\a：會發出嗶的警鈴聲
\d：代表一個數字的字元
\D：代表一個非數字的字元
\e：跳脫符號（escape）
\f：換頁
\n：換行
\s：一個空白字元（包括空行，換頁，跳格鍵也都屬於空白字元）
\S：非空白字元
\t：跳格字元（Tab）
\w：一個字母，包括了a-z，A-Z，底線跟數字
\W：非字母
```

除此之外，我們可以在雙引號中內插變數，可是卻無法在單引號中使用這樣的方式。比如在程式中，我們使用了print這個函數，後面接著字串，字串裡面包含了變數名稱跟換行字元。恰巧這兩者都是無法以單引號呈現的，因此我們可以看看下面的程式表現出他們的差異：

```
#!/usr/local/bin/perl

$c = "在Perl中使用字串非常容易";

print '$c\n';
print "$c\n";

我們會得到這樣的結果

[hcchi en@Apple]% perl ch2.pl
$c\n在Perl中使用字串非常容易
```

很明顯的看到第一行的輸出結果就單純的把單引號的內容表現出來，這樣的差異對之後程式的寫作其實有著相當的影響。

2. 利用引號將數字括住，雖然可以清楚的表達你希望將這個數值以字串的方式運算，但是其實這樣的方式有點囉唆，而且Perl的程式設計師並不會希望自己在寫程式時要弄的這麼複雜。因此只要在變數的使用時根據語境的不同而有不同的運算方式。也許可以看看以下的例子：

```
#!/usr/local/bin/perl

$a = "number";
$b = 3674;
$c = "4358";
$d = $a. $b;
$e = $b+$c;

print "$d\n";
```

```
print "$e\n";
```

可以得到：

```
number3674  
8032
```

所以你可以看到變數\$c，我們利用引號把數字括起來，但是當我們把變數\$b跟變數\$c相加時，Perl就會根據語意，自動把\$c轉成數字後再進行加法的運算。也因此，如果你的程式把變數寫成像上例的變數\$c一樣，雖然語法上不會有錯誤，不過對於有經驗的Perl程式設計師而言，反而會顯得很不習慣。如果他們看到這樣的程式寫法，也許還會發出會心的一笑。

3. 我們可以在定義變數時給定初值，就像上述的例子所使用的方式。而在Perl中，如果你只是定義了變數，而沒有給定初值，那麼這個變數會被Perl視為undef，也就是「尚未定義」的意思。

### 2.1.3 數字與字串轉換

毫無疑問，大部份我們在寫Perl的時候是不需要特定對數字與字串進行轉換，因為Perl通常會幫我們處理這一類的事情。例如你對兩個包含數字的純量變數進行加法運算以及連接運算則會產生不同的結果，我們可以作個實驗：

```
$a = 1357;  
$b = 2468;  
  
print $a+$b, "\n";  
print $a. $b, "\n";
```

就可以看到印出：

```
3825  
13572468
```

不過有時候你必須強制要求Perl使用某種資料型態，那麼就可以使用轉換函式，你可以利用int()函數把變數強制使用整數的型態。不過這樣的機會並不多，因此你只需要注意使用這些變數時用的算符，避免讓Perl產生誤會就可以了。

## 2.2 使用你自己的變數

在一般的情況下，Perl並不需要事先定義變數後才能使用，因此就像我們的範例中所看到的，你可以直接指定一個數值到某個變數名稱，而Perl大多也會欣然接受這樣的方式。可是在大部份的狀況，程式設計師所出現錯誤的機會遠大於Perl出現錯誤的可能。比如你可能會犯下所有程式員都會犯的錯誤：

```
$foo = 3;  
$f00 = 6;  
print $foo;
```

程式執行之後會得到3這樣的結果，這也許是你想要的，也許不是。不過如果你在編譯訊息裡面要求Perl送出編譯訊息給你(註二)，你也許會得到這樣的訊息：

```
Name "main::f00" used only once: possible typo at ch2.pl line 4.
```

這並不是錯誤，你也許因為打錯字而造成的結果還不足以影響Perl的執行，但是卻會影響你希望產生的結果。不過，使用編譯的警告參數還有其他用法。你可以在你的程式中加上"use warnings"，所不同的是，你在程式的一開始就使用"-w"這個參數，是要求Perl對你程式中每一行都進行檢查。可是有時候會因為使用不



同的版本，讓原來一切正常的程式在換為其他版本時產生出警告訊息。因此有些時候你可以單單對於某個區塊進行檢查的動作，這樣的強況下，你就可以使用"use warnings"這樣的方式來要求Perl幫忙。相對於此，我們還有其他的方式來跳過某個區塊的警告訊息。就像這樣：

```
#!/usr/local/bin/perl

use warnings;

{
  no warnings;
  $foo = 3;
  $f00 = 6;
}

print $foo;
```

我們在程式的一開始就使用了"use warnings"這個選項來對我們的程式進行編譯的檢查。不過卻在某個區塊中定義了"no warnings"，讓Perl跳過這個區塊進行檢查。這樣一來，我們在執行程式時，Perl就不會再發出上面的那些警告訊息。當然，除非你真的知道自己在做甚麼，否則還是盡量不要省略這樣的檢查才是正途。

另外一個良好的書寫習慣，就是在程式的前面加上use strict的描述，告訴Perl你希望使用比較嚴謹的方式來對程式進行編譯。而一旦使用use strict來對你自己的Perl程式進行嚴謹的規則時，你就需要用my這個關鍵字來定義你自己所需要的區域變數。因此假設我們剛剛的程式會寫成這樣：

```
use strict;
my $foo = 3;
$f00 = 6;
print $foo;
```

那麼一旦我們想要執行這支程式，Perl就會發生錯誤：

```
Global symbol "$f00" requires explicit package name at ch2.pl line 5.
Execution of ch2.pl aborted due to compilation errors.
```

因為我們沒有定義\$f00這個變數，Perl不知道你是忘了，或者只是打錯字，這是相當有用的，尤其在你的程式長度已經超過一個螢幕的長度時。因為我們打錯字的機會確實還不少，而且這樣的程式錯誤是非常難以除錯的。因此能夠在程式的一開始就強制使用嚴謹的定義是比較正確的作法。

### 2.2.1 變數的命名

其實我們已經看了好多例子，裡面都包含了Perl的純量變數，因此相信大家應該都不算陌生了。Perl的變數是以字母，底線，數字為基本元素，你可以用字母或底線作為變數的開始，然後接著其他的字母，底線以及數字。可是在Perl的規定中，是不能以數字開始一個變數名稱的。

而在Perl中，純量變數則是以\$符號作為辨識，因此像之前看到的都是屬於純量變數的範圍。

當然，怎麼幫你的變數名稱命名也是必須注意的，因為在Perl中，大小寫的字母是會被視為不同的。因此你如果用了\$foo跟\$f00，這在Perl中是屬於兩個不同的變數，只是我們十分不建議這樣的命名方式。否則將來可能維護你的程式碼的人也許會默默的咒罵你，那可別怪我們沒事先警告了。

另外，你應該讓其他看你的程式設計師看到某個變數都大概可以猜出這個變數的作用，你自己想想，如果你看到某支程式裡面的變數名稱是像這樣子：\$11，\$22，\$33，或者像這樣：\$100000001，\$100000002....。

你會不會想要殺了這程式的作者呢？

至於大小寫也是在定義變數時可以運用的另一項特點，例如有人就習慣利用像這樣的方式來定義變數名稱：\$ChangeMe。這樣可以避免某些因為連字時造成的混淆，不過一般而言，只要能清楚的表現變數的特性，大小寫與否則視個人的習慣。不過這一切都是為了將來程式維護上的方便，對Perl來說，這些變數的命名對他



並沒有特別的意義。不過如果將來維護程式的可能是你自己，還是別找自己的麻煩吧！

### 2.3 賦值

既然變數就像一個容器，是拿來存放變數，能夠賦予變數他的內容就是非常重要，而且非常基本的一件事了。在程式中，我們經常會對於變數進行賦值運算，否則我們只需要一個定數就好，何必大費周章的使用變數呢？一般來說，要指定一個值給某一個變數的動作並不複雜，而且大概有這兩種主要的方式。

#### 2.3.1 直接設定

這是直接用等號(=)來將某個數值或運算結果指定給變數。例如我們也許會這麼寫：

```
$foo = 3;
$foo = 2 + 1;
$bar = "bar";
$foo = $foo + 2;
$bar = $bar. " or foo";
```

就可以看到印出的結果

```
8
bar or foo
```

在後面的兩個式子中我們看到左，右兩邊分別出現了兩次相同的變數名稱。這樣的方式是非常常見的，我們在右邊先取變數原來的數值，經過運算之後，把得到的結果指定給左邊的變數。因為Perl的自由度非常高，也許會有人想要把許多賦值的工作一次完成，那麼他可能把程式寫成這樣：

```
use strict;
my $foo = 3;
my $bar;
$foo = 3 + $bar = 2;
print "$foo\n";
```

那麼應該會看到這樣的結果：

```
Can't modify addition (+) in scalar assignment at ch2.pl line 6, near "2;"
Execution of ch2.pl aborted due to compilation errors.
```

沒錯，我們雖然說過Perl的語法其實相當自由，但是這樣的寫法確實會讓Perl搞混了。當然，我們可以把程式改寫成：

```
use strict;
my $foo = 3;
$foo = 3 + (my $bar = 2);
print "$foo\n";
```

Perl也會輸出結果為5，可是這樣雖然Perl可以清楚的了解你要表達的意思，只怕其他看程式的人還是會一頭霧水，而且這樣並不會讓你的程式執行得更快，當然無法靠這種方式讓你贏得Perl Golf(註三)，所以除非你有很好的理由，否則還是少用吧！

#### 2.3.2 還可以這樣

就像我們剛剛看到的，我們可以在賦值前先在等號右邊取出變數的值進行運算，就像這樣：`$foo=$foo+3`。可是其實某些二元算符可以有更方便的賦值方式，我們可以寫成這樣：

```

use strict;
my $foo = 3;
print "$foo\n";
$foo+=3; # 其實就是 $foo = $foo + 3
print "$foo\n";
$foo*=3; # 乘號也可以這樣使用
print "$foo\n";
$foo/=3; # 其實所有的二元運算符都可以這麼用
print "$foo\n";

```

可以發現這樣的形式確實非常方便：

```

3
6
18
6

```

## 2.4 運算

其實變數的運算就跟一般的數值是一樣的，我們可以利用大部份我們所熟知的運算符號來對變數進行運算。例如我們當然可以把兩個變數相加，然後賦值給另一個變數，就像這樣：`$third=$first+$second`。或者更複雜的運算，這從過去我們學到的數學中都可以看到。當然，在Perl裡面也的運算式也符合現實生活中的規則，Perl會先乘號，除號進行運算，然後在把結果作相加或相減（如果你的運算式內有這些算符的話）。所以`$foo=3*8+2*4`就應該是32，而不是106。

不過Perl裡面並沒有數學中的中括號或大括號，而所有你希望先行計算的部份，都是由小括號將他括起來，例如你可以改寫剛剛的算式成：`$foo=3*(8+2)*4`，那麼結果顯然就變成120了。而運算符的優先順序正是你需要進行運算時非常最要的部份，雖然你已經知道乘號與除號的優先順序高於加號跟減號。而在這個時候，你還可能需要知道的某些算符的優先順序依照他們的優先性大概有下列幾種：

```

++, --
**
*, /, %, x
+, -, .
&
&&
||
+=, -=, *=, /=...

```

當然，Perl的算符並不只有這些，不過我們後面陸續會提到。如果你現在想要知道更多關於Perl算符的說明，可以看一下perldoc perlop這份文件。

## 2.5 變數的輸出/輸入

當我們寫了一堆程式之後，我們當然希望程式運算的結果可以被看到，否則即使程式運作的結果讓人非常滿意，你也無從得知。當然，換個角度想，如果你的程式錯的一踏糊塗，也不會有人知道。不過如果如此，那何必還花了大量的時間寫這支程式呢？如果你無法從程式得到任何結果。

最簡單的輸出方式，其實我們已經看了很多了，那就是利用print這個Perl的內建函數。而且用法非常直覺，你只需要把你想要的結果透過print送出到標準輸出(STDOUT)，當然，通常標準輸出指的就是螢幕，除非你自己動了甚麼手腳。我們可以來看看下面的例子會有甚麼結果：

```

use strict;
my $foo = 3;
print $foo;
print $foo*3;
print "列印字串\n";

```

```
print $foo, $foo+3, $foo*3;
```

很簡單的，我們就可以看到：

```
39列印字串
369
```

從範例中我們很清楚的就發現，我們可以單純的列印一個變數，一個運算式，一個字串，或者一堆用逗點(,) 分隔開來的運算式。因為我們可以在print後面連接一個運算式，所以我們當然也可以寫成這樣：

```
print "foo = ", $foo;
```

或者你希望最後的輸出結果還可以換行，那麼你可以這麼寫：

```
print "foo = ", $foo, "\n";
```

可是如果你有三個變數，那麼你寫起來也許會像這樣：

```
print "first = ", $first, "second = ", $second, "third = ", $third, "\n";
```

好吧，雖然吃力，而且可能容易產生錯誤，不過畢竟你做到了。只是如果現在又多了一倍，那困難度可又增加了不少。

### 2.5.1 變數內插

我想你大概可以慢慢感受到，以Perl的程式設計師的個性，他們絕對不希望這樣的事情發生，因為這些程式設計師總是不希望自己的時間浪費在打字這件事情上，因此當然要有方法能夠少打一些字，又容易維持程式的正確性。而變數的內插就提供了這樣的福音。

我們之前提過對於字串的表示中，單引號與雙引號之間的差異。其實這兩者之間還有一個重要的差異，就是雙引號中可以進行內插變數，而單引號依然很真實的呈現引號內的字串內容。我們可以看看其中的差異：

```
my $foo = 3;
print "foo = $foo\n";
print 'foo = $foo\n';
```

很明顯的，輸出後就有了極大的不同：

```
foo = 3
foo = $foo\n
```

在雙引號中，不但特殊字元\n會被轉換為換行字元，變數名稱 \$foo 也會被取代為變數的值後輸出。反觀利用單引號的時候，不論變數名稱或特殊字元都會被完整而原始的表示。不過如果你希望輸出這樣的字串呢？  
print "\$ 表示錢字符號\n"

在雙引號中，如果你希望正確的表達某些符號，例如用來提示特殊字元的倒斜線，表示變數的符號時，你必須用一個倒斜線來讓原來符號的特殊意義消失，看看下面的寫法：

```
print "\$ 用來提示純量變數， \@ 則是陣列";
```

那麼你就可以正確的顯示你要的結果。除此之外，我們也許還有一些好玩的技巧，記得Perl的名言嗎？「辦法不只一種」。

讓我們來看下面的程式：

```
print "\$ 用來提示純量變數， \@ 則是陣列，還有 \"\n";
print qq/\$ 用來提示純量變數， \@ 則是陣列，還有 \"\n/;
print qq|\$ 用來提示純量變數， \@ 則是陣列，還有 \"\n|;
```

結果看來都是一樣的

\$ 用來提示純量變數， @ 則是陣列，還有 "

這確實非常神奇，首先我們提示一下，為了避免Perl誤以為你要結束某個字串，因此如果你要印出雙引號時，記得先讓Perl知道，於是就是利用\"的方式來解除雙引號原來的作用。可是一旦如此，你的字串內也許會變得難以判讀，尤其常常雙引號又是成雙的出現時。這時候，你可以利用qq來描述字串，而在範例中，我們用了qq//，qq||，其實qq後面可以接任何成對的符號。如果有興趣也可以自己動手試試。

接下來，我們可以來談談怎麼接受使用者的輸入，也就是讓程式可以根據使用者的需求而有不同的反應。經常被使用的方式應該是程式在進行時，停下來等使用者輸入，當接收到換行字元時，程式就繼續往下執行。這時候我們就是大多就是依賴的方式。很顯然的就是STDOUT的對應，也就是所謂的標準輸入，而我們常用的應該大多就是鍵盤。因此，Perl在遇到時便會等待輸入，我們可以用這個簡單的例子來試試：

```
print "please enter your name: ";
my $name = ;
print "\n";
print "hello, $name\n";
```

當我們執行時，就會有這樣的結果：

```
please enter your name: hcchi en

hello, hcchi en
```

看起來，王子跟公主似乎過著幸福，快樂的日子。可是唯一小小的缺憾，卻是Perl連我們在結束字串輸入的換行字元也一併當成字串的一部份了。這樣的動作有時候會有很大的影響，因此我們也許要考慮把這樣的錯誤彌補過來。這時候，chomp函數就派上用場了。這個函數生下來似乎就只為了進行這項工作，至少我們再也不用擔心使用者的換行字元該怎麼辦。他的用法顯然也不特別困難，只要把你需要修正的字串當成傳入值就可以了。

```
chomp($name);
```

所以我們只要把這一行加到剛剛的程式裡面，我想你應該就會發現一些變化。沒錯，原來我們的輸出最後還多了一行空行，那是因為我們輸入時打了最後一個換行字元，不過經過chomp的修正，那個字元果然就沒有了。那麼chomp有沒有甚麼資訊可以讓我們參考呢？既然他是一個函數，他就會有一個回傳值，而chomp的回傳值就是被移除的換行字元個數。例如我們如果有一個字串：

```
my $name = "hcchien\n";
```

那麼一但我執行了

```
chomp($name);
```

理論上就會傳回1的值。實際上也確實如此，那麼我們可以再來試試，如果變數\$name的值變成

```
$name = "hcchien\n\n";
```

然後我們發現回傳值還是1，也就是說，chomp只對字串結尾的那個換行字元有效。因此如果我們只執行了一次chomp，並不是把字串後面的換行字元全部取消，而只是移除了一個。

## 2.6 Perl預設變數

很多時候，剛學Perl的程式設計師似乎常常會遇到一些問題，也就是不容易看懂其他的Perl程式。這其中的原因當然很多，例如Perl的寫作形式非常自由，同樣的需求可以利用各種方式達成，有些程式設計師常常會用非常簡略的語法而讓易讀性降低。另外，許多Perl的預設變數對於初學者也是一個問題。你常常會看到一堆符號在程式裡飛來飛去，卻完全不知道他們在說甚麼，你當然可以利用Perl的線上文件perlvar去找到你要的答案，不過我們會適時的在不同的章節提到一些Perl常用的預設變數。

## 2.7 defined 與 undef

你也許有過經驗，當你在寫一支程式的時候，你定義了一個變數，就像我們平常作的：

```
my $foo;
```

或者你可能寫成這樣：

```
my $foo = "";
```

於是在你的程式過程中，這樣兩種方式所定義出來的變數在你的程式並沒有產生不同。於是程式平靜的結束，你開始想像你多打了好幾個字，只為了告訴Perl \$foo這個變數是個空字串。可是所有的事情一如你所預測的一般，你還是沒想想透，到底宣告變數是空字串到底有沒有意義呢？其實大部份的時候，你是不需要在

定義變數時宣告為空字串，因為當這個變數被定義時，Perl會指定他為undef。而當這個變數被作為數值時，他瞬間就被當成零，同樣的，在被當成字串運算時，他則會被作為空字串。

不過，如果你的程式開啟了warnings參數，而打算列印一個undef的變數，可是會遭到警告的，因為Perl顯然很難理解你為什麼需要列印一個沒有被定義的變數。而這很可能是你的程式有某部份發生了問題，所以千萬別隨便忽略Perl的警告，再仔細檢查你的程式吧！

所以你也許要確認你的程式在某些敘述是否正如你所期待的正確的進行了某些運算，這時候有一個函數就可以派上用場了，那就是defined()。你可以用defined來確定某個變數是否是經過定義，很簡單的就像這樣：

```
defined($name);
```

而許多程式的寫作中，也經常使用這個函數來進行判斷。例如你可以這麼寫：

```
my $name;
if (defined($name)) {
    print $name;
} else {
    print "it's undefined";
}
```

我們可以很清楚的看到這樣的宣告一個變數會被設為undef。

而且，undef在perl中也是個關鍵字，你可以直接指定某個變數是undef，就像你在賦值給任何變數一樣。所以你可以很簡單的寫成：

```
$name=undef;
```

習題：

1. 使用換行字元，將你的名字以每個字一行的方式印出。
2. 印出'\n', '\t'字串。
3. 讓使用者輸入姓名，然後印出包含使用者姓名的招呼語(例如：hello xxx)。

註一：有些時候是必須強制進行轉換，Perl才知道你真正需要的是甚麼。

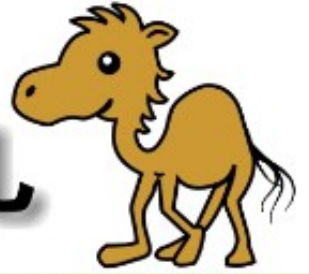
註二：你可以在程式的最前面寫成這樣「#!/usr/bin/perl -w」，告訴Perl你希望啟動編譯警告。

註三：一種長期並不定時舉行的Perl程式設計遊戲，以程式碼最短路者獲勝，就像高爾夫球，最少桿者獲勝，詳細可以參考<http://perlgolf.sourceforge.net/>。





# Perl 學習手札



There is more than one way to do it.

## 3. 串列與陣列

在我們已經知道怎麼使用純量變數之後，我們就可以處理非常多的工作。可是有些時候，當我們要使用純量變數來儲存許多性質相近的變數時，卻很容易遭遇瓶頸。例如我希望儲存某個班級四十位學生的數學期末成績，這時候如果每個學生的成績都需要用單獨的一個變數來儲存的話，那會讓資料難以處理，也許你從此再也不想寫程式了，而且你的程式大概會長的像這樣子：

```
my $first = '40';  
my $second = '80';  
my $third = '82';  
...  
...
```

沒錯，這樣的寫法雖然可能可以讓我們比過去使用紙張的方式正確率高一些，可是卻未必會省事。另外，如果我希望從資料庫找出今天總共有多少人在我的網路留言板留言，那這時候的留言個數是未知的，要怎麼批次處理這些資料就很花腦筋了，所以要有適當的資料結構可以作這樣的處理。

很顯然的，陣列的運用非常的廣泛，幾乎大部分撰寫程式的時候都會使用陣列來進行資料的存取，在許多程式語言中，陣列的結構相當的複雜，這確實是必要的。因為陣列的使用要必須足夠靈活，才能夠發揮它的功能，可是如果太過複雜卻也是造成入門者的進入門檻。Perl對於這方面卻有一些不同的做法，它提供的陣列結構非常簡單，如果你用最入門的方式去看它，很多第一次接觸的人甚至也可以輕易上手。可是Perl的陣列卻也可以利用非常強大的方式擴展開來，讓許多第一次看到Perl陣列結構卻非常失望的人也能重拾對Perl的信心。當然，使用這些技巧來進行Perl陣列的擴充，不但可以像其他程式語言一般，可以進行多維陣列之外，還可以能精準的結合某些資料結構，當然，這部份我們不會在一開始介紹陣列時就把大家嚇走，不過如果你已經對陣列的方式有些熟悉，可以在後面的章節慢慢看出Perl在這方面設計的巧妙。

### 3.1 何謂陣列

對於我們剛剛提出來的資料結構需求，希望能把相同的東西簡單的存取，並且讓它們能被歸納在一起。陣列正是解決這個問題的方案，也就是把一堆性質接近的變數放在同一個資料結構裡，這樣可以很方便的處理跟存取。就像一疊盤子一樣，他們都是性質相接近的東西，於是我們就把盤子碟子一疊，而屬於不同性質的東西就分放在其他地方，比如我們就不太應該把碗跟盤子放在同一疊裡面。在Perl裡面，你可以定義一個陣列，而陣列裡面存放的就是純量，當然存放的個數可以由零個到許多個，至於實際可以儲存的個數則依據每部電腦不同而有所差別，因為Perl依然依循它自己的個性，並不對程式設計師進行太多的限制，因此它可以允許你使用系統上所有的資源，換句話說，你可能會因為一個陣列過大而佔用系統的所有資源。

### 3.2 Perl 的陣列結構

我們先來看看怎麼在Perl裡面定義一個陣列。在Perl中，陣列變數是以@符號開頭，例如你可以定義一個變數名稱叫做@array。然後利用\$array[0]，\$array[1]...的方式來存取陣列裡的元素。也就是說，你在定義了陣列@array之後，你可以指定陣列裡面的值，就像這樣的方式：

```
my @array;  
$array[0] = 'first';  
$array[1] = 'second';  
$array[2] = 'third';  
....
```

這樣比起剛剛我們一個一個變數慢慢的指定雖然方便了不少，至少我們可以很清楚的了解這些數值都是屬於同一個群組的，因為它們被放在同一個陣列中(註一)。不過這樣的寫法實在太辛苦了，尤其當你已經知道你陣列中的元素個數，以及他們個別的值，你就可以用簡單一點的方式來把陣列的值指定給你的陣列，就像這樣：

```
my ($array[0], $array[1], $array[2]) = qw/first second third.../;
```

其中，qw/first second third.../這一串東西就被稱為串列，例如：

```
my ($one, $two, $three) = (1, 2, 3);
```

也就是把一個串列一次指定給三個變數。利用qw也是同樣的方式，因此剛剛那一行程式其實也可以寫成：

```
my ($first, $second, $third) = qw/first second third/;
```

這樣的方式，就是我們把串列的值指定給變數，所以當然這些變數也可以是陣列的元素。不過既然我們確定要把串列的值指定給某個陣列，我們顯然可以更簡單的這麼作：

```
my @array = qw/first second third/;
```

這樣的方式就是直接利用串列賦值給陣列的方式，而類似的方式還可以寫成這樣：

```
my @array = (1...10);  
my @array = (0, 1, 2, 4...8, 10);  
my @array2 = (3, -1, @array, 13);  
my @array2 = qw/3, -1, @array, 13/; # 這應該不是你想要的東西
```

當然，如果你定義了一個陣列，但是卻沒有賦值給他，那麼這個陣列就會是一個空陣列。相同的狀況，你也可以指定任意的陣列大小給Perl，當然前提是你的電腦有足夠的能力承受。這當然也是Perl的傳統之一。Perl從來就不是一個嚴謹的程式語言，因此對於陣列的部份也採取同樣的規定。你不需要在程式的一開始就規定你的陣列長度，因此你可以在程式裡面隨時新增元素到你的陣列中。例如你的程式也許會寫的像這樣子：

```
my @array = qw/第零 第一 第二/;  
$array[3] = '第三';  
$array[4] = '第四';
```

沒錯，你可以使用串列形式來指定陣列的值，也可以直接把值指定給陣列的某個索引值，就像我們剛剛所使用的方式。另外，你也會發現，如果你這麼寫的話，Perl也不會阻止你：

```
$array[15] = '一下子就到 15 了';
```

那麼Perl會直接幫你的陣列程度擴充到15，也就是陣列的索引值會變成從0-14，而陣列大小變為16。至於陣列中間沒有被指定的值，Perl都會自動幫你設為undef，所以你的陣列中，有許多還沒定義的值。好吧，很多人也許對於這樣的設計不以為然，不過有時候這樣還是很方便的，不是嗎？想像你已經預測你的陣列會有20個元素，可是你現在只知道最後一個元素的值，你總不希望必須先把前面十九個元素值填滿之後才能開始使用你期待已久的那個元素值吧？

當然，對於那些認為應該嚴謹的定義程式語言語法，不能讓程式設計師為所欲為的人來說，Perl顯然不是他們會選擇的工具。而且這樣的戰爭已經持續了很長的一段時間，也不是我們可以在這裡解決的。讓我們暫且跳開風格爭議，繼續回來看Perl在陣列中的用法吧。

有時候我們需要知道陣列中的元素個數，比如我們希望在陣列中依序取出陣列中的元素並且進行運算，那麼我們就可以利用下面的方式來進行：



```
my @array = qw{first second third};
# 記得利用qw賦值給字串的作法嗎？用qw賦值給陣列也是類似
$array[4] = 'fifth';           # 我們跳過索引值3
print $array;                  # 這裡取得的是最後一個索引值
print $array[3];               # 這裡應該不會有任何結果
```

既然\$array是陣列中最後一個索引值，所以我們可以利用(\$array + 1)得到目前陣列中的元素個數(註二)。不過如果你打算利用這個索引值來確定目前陣列的長度，並且加入新的元素，就像這樣：

```
my @array = qw{first second third};
$array[$array+1] = 'forth';     # 把新的值放到現在最大索引值的下一個
```

當然，如果你這樣寫也是可以接受的：

```
my @array = qw{first second third};   # 一開始，你還是有三個元素值
$array[$array+1] = 'forth';           # 這時候的 $array 其實是 2
$array[$array+1] = 'fifth';           # 可是這時候 $array 已經變成 3 了

print @array;
```

### 3.3 push/pop

沒錯，我是說那樣的寫法可以被接受，可是好像非常辛苦，尤其當你已經被一大堆程式搞到焦頭爛耳，卻還要隨時注意現在的陣列到底發展到多大，接下來你應該把最新的值放到那裡，這樣顯然非常辛苦。你一定也猜到了，Perl不會讓這種事情發生的。所以Perl提供了push這個指令把你想要新增的值「推」入陣列中，同樣的，你也可以利用pop從陣列中取出最後一個元素。不過為甚麼要使用push/pop這樣的指令，這當然和整個陣列的資料結構是具有相關性的，如果你弄清楚了陣列的形式也許就很容易理解了。我們可以把陣列的儲存看成是一疊盤子，因此如果你要放新的盤子，或者是拿盤子，都必須從最上面動作。這也就是為甚麼我們可以利用push/pop來對陣列新增，或是取出元素的最重要原因。我們可以從下面的例子看到 push跟pop的運作：

```
my @array = qw{first second third};
push @array, 'fourth';
print $array;           # 這裡印出來的是3，表示'fourth' 已經被放入陣列
pop @array;
print $array;           # 至於pop，則是把元素從陣列中取出
```

而且利用pop取出元素一律是從陣列的最後一個元素取出，也就是「後進先出 (last in, first out)」的原則。當然，pop的回傳值也就是被取出的陣列元素，上面的例子來看，取出的就是'fourth'這個元素。另外，在使用push時，也不限定只能放入一個元素，你可以放入一整個陣列。那麼就像這樣的寫法：

```
my @array = qw{first second third};
my @array2 = qw{fourth fifth};
push @array, @array2;
print @array;           # 現在你有五個元素了
```

### 3.4 shift/unshift

沒錯，push/pop確實非常方便，他讓我們完全不需要考慮目前陣列的大小，只需要把東西堆到陣列的最後面，或者把陣列裡的最後一個元素拿掉。不過我們也發現了，這樣的操作只能針對陣列的最後一個元素，實在有點小小的遺憾。其實我們想想，如果我把陣列中非結尾的某個元素去掉，那會發生甚麼事呢？比如我現在有一個陣列，他目前總共有三個元素，因此索引值就是0..2。如果我想要把索引值為1的那個元素取消，那麼索引值是不是也就需要作大幅更動。尤其當陣列的元素相當多的時候，其實也會有一些困擾。

不過Perl還是允許我們從「頭」對陣列進行運算，也就是利用shift/unshift的指令。如果我們已經知道push/pop的運作，那麼我們可以從範例中輕鬆的了解shift/unshift對陣列的影響：

```
my @array = (1..10);  
shift @array;          # 我把1拿掉了  
unshift @array, 0;      # 現在補上0  
print @array;          # 現在陣列的值變成了(0, 2..10)
```

現在你的陣列進行了大幅度的改變，我們應該來檢查一下，當我們在進行shift運算過程中，陣列元素的變化。

我們還是用剛剛的陣列來看看完整的陣列內容：

```
my @array = (1..10);          # 我們還是使用這個陣列  
shift @array;                 # 我把1拿掉了  
print "$_\t$array[$_]" for (0..9); # 現在陣列的值變成了(0, 2..10)
```

好極了，我們看到了輸出的結果：

```
0      2  
1      3  
2      4  
3      5  
4      6  
5      7  
6      8  
7      9  
8      10  
Use of uninitialized value in concatenation (.) or string at ch3.pl line 7.  
9
```

沒錯，我們看到了錯誤訊息。因為我們的陣列個數少了一個，因此索引值9目前並不存在，Perl也警告了我們。所以我們發現了，Perl在進行shift的時候，會把索引也重新排列過。不過你能不能從中間插入一個值，並且改變陣列的索引排列，或是攔腰砍斷，取走某些元素，然後希望Perl完全不介意這件事呢？目前看來似乎沒有辦法可以這麼作的。不過有些方式可以讓你單讀取出陣列中某些連續性的元素，也就是使用切片的方式。

### 3.5 切片

就如我們之前提到，我們總是把一堆串列放入陣列中，雖然放入的方式不盡相同，但是至少我們可以在陣列中找出0個以上的元素所組成的陣列。沒錯，如果我們知道一個陣列中的元素，而且我希望取出這個陣列中的某些連續性元素是不是可行呢？例如有一個陣列的元素是(2003...2008)，那麼如果我希望取得的是這個陣列中2004-2006這三個元素，並且把這三個元素拿來進行其他運算或運用，我是不是應該這樣寫：

```
my @year = (2003...2008);
```

```
my ($range[0], $range[1], $range[2]) = ($year[1], $year[2], $year[3]);
```

其實如果你真的這麼寫了，也不會有人說你的程式有錯誤，雖然這樣的寫法總是很容易讓人產生錯誤。即使不是語法上的錯誤，也容易因為打字的原因而產生可能的邏輯錯誤。既然如此，我們顯然應該找出容易的方法來作這件事。我們用一個很容易看清楚的例子來說明吧：

```
my @array = (0...10);  
my @array2 = @array[2...4];  
print @array2;                                # 沒錯，你拿到了(2, 3, 4) 三個元素
```

這個方法，我們就稱為切片，就像我們把生魚片取出其中的一片。可是如果我要的範圍並不屬於連續性的話，還能切片嗎？其實就像你一個一個取出陣列中的元素，只是有些部份是連續的，你不希望把每個元素都打一次。所以如果你希望多切幾片，可以考慮這麼作：

```
my @array = (0...10);  
my @array2 = @array[2...4, 6];
```

這時候，你拿到的不但是(2, 3, 4)三個元素，也包含了6這一個元素。這樣是不是非常方便呢？

### 3.6 陣列還是純量？

如果你已經開始自己試著寫一些Perl程式，不知道你有沒有遇到這個問題，你有一個陣列@array，你想新增一個陣列，元素跟原來的陣列@array相同，於是你想寫了這樣一個式子：

```
my @array2 = @array;
```

沒想到一時手誤，把這個式子打成這樣：

```
my $array2 = @array;
```

這時候，Perl卻沒有傳回錯誤給你，可是程式會傳回什麼結果呢？我們可以來實驗看看，只要打這幾行：

```
my @array = (0...10);  
my $array2 = @array;  
print $array2;                                # 程式傳回 11
```

這個值恰好就是陣列@array的元素個數，所以我們似乎發現好方法來找到陣列的元數個數了。不過也許應該來研究一下，為什麼Perl對於資料型態能夠進行這樣的處理。這其實是非常重要的一个部份，也就是語境的轉換。這很像我們在之前曾經遇過的例子，當我有兩個變數，分別是：

```
my $a = 4;  
my $b = 6;
```

可是當我使用 \$a.\$b 跟 \$a+\$b 兩個不同的運算子時，Perl也會自動去決定這時候該把兩個變數使用字串，或是變數進行處理。因為語境的不同，讓運算的方式也有所不同，這在Perl當中是非常重要的觀念。不過這個觀念絕非由Perl所獨創，相反的，這樣的用法在現實生活中是屢見不鮮。比如有人問你平常用甚麼寫程式，你也會依照當時聊天的情況回答你是用甚麼編輯器，或者是用甚麼程式語言。因此在語言的使用中，如何選對適當的語境確實相當重要，而既然Larry Wall就是研究語言的專家，把這種方法運用在Perl裡面也是再自然不過了。

我們再來看看剛剛的例子，我們指定一個陣列，並且指定這個陣列的元素包括一個從0到10的串列，而當我們把這個陣列賦值給一個純量變數時，Perl便會把串列元素個數指定為這個純量變數的值。這也就表示Perl正以純量變數的語境在處理你的運算，而對一個陣列以純量變數的語境進行運算時，Perl就如我們所看到的，以陣列中串列元素的個數表示。所以你可以寫出這樣的運算式：

```
my @array = (1..10);           # 利用串列賦值給陣列
my $scalar = @array + 4;       # 在純量語境中進行
my @scalar_array = @array + 4;
# 先以純量語境進行運算，然後以串列方式賦值給陣列
```

這樣看起來會不會有一點眼花繚亂？程式第一行的中，就像我們所熟知的狀況，我們把一個串列賦值給陣列。接下來，我們利用純量語境把陣列內串列元素的個數取出，並進行運算，然後把結果放到一個純量變數裡，這裡全部都是以純量變數的方式在進行。第三行就比較複雜一點了，我們先用純量語境，取出陣列的串列元素個數，以純量方式進行運算，接下來把這個得到的結果以串列的方式指定給陣列@scalar\_array。所以最後一行其實也像是這樣：

```
my @array = (1..10);
my $scale = @array + 4;           # 這裡是純量語境
my @scalar_array = ($scale);     # 把得到的結果放進串列中，並且賦值給陣列 @scalar_array
```

其實就像這裡所看到的，如果你的需求是一個串列，而你卻只能得到一個純量，那麼Perl就會給你一個只有一個元素的串列。其實要訣就是仔細看看你希望得到甚麼樣的東西，而Perl可以給你甚麼東西。而有時候，當理想與現實有些落差的時候，也許就會有些undef產生。假如我們把剛剛的例子改成這樣：

```
my @array = (0..10);
my ($scalar1, $scalar2) = (@array + 4);
```

當我們要求的串列無法獲得滿足時，Perl就會幫忙補上undef。

### 3.7 一些常用的陣列運算

既然我們總是喜歡把性質類似的變數放在一起變成陣列，那麼很多時候我們就會希望對這一整個陣列進行某些運算。例如排序，過濾，一起帶入某個公式中進行運算等等。這時候我們經常利用迴圈來幫我們處理這一類的事情，不過有些常用的運算，Perl已經幫我們設想好了，我們只需要輕鬆的一個式子就可以進行一些繁複的工作。

#### 3.7.1 sort

排序總是非常必要的，我們在舉陣列的時候有提到，如果我們要把某個班級學生的數學成績放入陣列，那麼我們也許會希望利用這些成績來排序。這時候，sort就非常有用。我們可以這樣作：

```
my @array = qw/45 33 75 21 38 69 46/;
@array = sort { $a <=> $b } @array;
這樣Perl 就會幫我們把陣列重新排列成為
21      33      38      45      46      69      75
```

其實，如果你這樣寫也是有相同的效果：

```
@array = sort @array;
```

當然，如果你需要比較複雜的排序方式，就要把包含排序的區塊加入，所以你也可以寫成：

```
@array = sort { $b <=> $a } @array;
```

其中\$a跟\$b是Perl的預設變數，在排序時被拿來作為兩兩取出的兩個數字。而<=>則是表示數字的比較，如果陣列中的元素是字串，則必須以cmp來進行排序。

我們可以用接下來的例子來說明怎麼樣進行更複雜的排序工作。

```
my @array = qw/-4 45 -33 8 75 21 -15 38 -69 46/;  
@array = sort { ($a**2) <=> ($b**2) } @array; # 這次我們以平方進行排序
```

所以得到的結果會是：

```
-4      8      -15     21      -33     38      45      46      -69     75
```

### 3.7.2 join

有時候，你也許會希望把串列裡面的元素值用某種方式連接成一個字串。比如也許你想要把串列中的元素全部以','來隔開，然後連接成一個字串，那麼join就可以幫上忙了。你可以在串列中這麼用：

```
print join ' ', qw/-4 45 -33 8 75 21 -15 38 -69 46/;
```

這一行顯然也可以寫成：

```
my @array = qw/-4 45 -33 8 75 21 -15 38 -69 46/;  
print join ' ', @array;
```

和join函數相對應的則是split，他可以幫忙你把一個字串進行分隔，並且放進陣列中。

### 3.7.3 map

很多人會使用Excel的公式，而公式的作用就是針對某一行/列進行統一的運算。比如小時候在學校考試的時候，老師常常會因為全班成績普遍太差，而進行所謂「開平方乘以十」的計算。這時候，如果可以用map就顯得很方便了。

```
my @array = map { sqrt($_)*10 } qw/45 33 8 75 21 15 38 69 46/;
```

我們可以看到，串列裡面是學生的成績，所謂map就是把陣串列裡的元素一一提出，並進行運算，然後得到另外一個串列，我們就把所得到的串列放到陣列中。於是就可以得到這樣的一個陣列：

```
67.0820393249937  
57.4456264653803  
28.2842712474619  
86.6025403784439  
45.8257569495584  
38.7298334620742  
61.6441400296898  
83.0662386291807  
67.8232998312527
```

當然，map還有許多有趣的使用範例，而且如果能適時運用，確實能大幅降低你寫程式的時間，也可以讓你的程式更加乾淨俐落。

### 3.7.4 grep

我們既然可以針對串列中的每一個元素進行運算，並且傳回另一個串列，那麼是否可以針對串列進行篩選呢？例如我希望選出串列中大於零的元素，或者以字母開始的字串元素，那麼我可以怎麼作呢？這時候，



grep就會是我們的好幫手。如果各位是Unix系統的使用者，應該大多用過系統的grep指令，而Perl的grep函數雖然不盡相同，不過精神卻是相近的。我們可以利用grep把串列中符合我們需求的元素保留下來。就像這樣：

```
my @array = qw/6 -4 8 12 -22 19 -8 42/;          # 指定一個串列給陣列 @array
my @positive = grep {$_ > 0} @array;             # 把@array裡大於零的數字取出
print "$_\n" for @positive;                       # 印出新的陣列 @positive
```

而且答案就正如我們所想像的，Perl能夠正確的找出這個陣列中大於零的數字。  
也許你會有一些不錯的想法，如果我們想要把剛剛的陣列中所找出大於零的數字取得平方值之後印出，那麼我們應該怎麼做比較容易呢？當然，一般的情況下，我們就會想到迴圈，而這也正是我們接下來要說的部份。

習題：

1. 試著把串列 (24, 33, 65, 42, 58, 24, 87) 放入陣列中，並讓使用者輸入索引值 (0...6)，然後印出陣列中相對應的值。
2. 把剛剛的陣列進行排序，並且印出排序後的結果。
3. 取出陣列中大於40的所有值。
4. 將所有陣列中的值除以 10 後印出。

註一：當然，你也可以把程式中的所有純量變數全部放在一個陣列中，不過很快的，你會發現連你自己都不想再看到這支程式了。

註二：別忘了，Perl的索引值是由零開始。



# Perl 學習手札



There is more than one way to do it.

## 4. 基本的控制結構

### 4.1 概念

大部份的時候，程式總不會跟著你寫程式的順序，一行一行乖乖的往下走。尤其是當你的程式由平鋪直敘漸漸變成有些起伏，這時候，怎麼確定你的程式到底應該往那裡走，或者他們現在到底到了那裡。如果你無法掌握程式的流程，只怕他們很快就會離你而去。你從此再也無法想像你的程式會怎麼運作，當然也很有可能你就寫出了會產生無窮迴圈的程式了。

#### 4.1.1 關於程式的流程

在程式的進行當中，你經常會因為過程發生了不同的事件，因為結果的不同而必須進行不同的運算，這個時候，你就必須進行程式中的流程控制。或者，你會需要對某些工作進行重複性的運算，這時候，重複性的流程控制就可以大大的幫助你減輕工作負擔。因此，你常常會發現，流程的控制在你的程式之中確實是非常重要的而且經常被使用的。雖然Perl的流程控制跟其他程式語言並沒有太大的差異，不過我們還是假設大家並沒有這方面的基礎。所以還是從頭來看看最基本的流程控制應該怎麼作呢！

#### 4.1.2 真，偽的判斷

流程的基本控制主要在於判斷某個敘述句是否成立，並藉以判斷在不同情況下該怎麼進行程式的流程。比如我們可以用簡單的例子來認識一下流程控制的進行：

```
my $num = <STDIN>;
chomp($num);

if ($num<5) {
    print "small";
} else {
    print "big";
}
```

這樣看起來是不是非常簡單呢？

不過Perl比較特殊的部份在於他並不存在一種獨立的布林資料型態，而是有他獨特對於真，偽值的判定方式。所以我們應該先要知道，在Perl中，那些值屬於真，那些值屬於偽，這樣一來，我們才能知道判斷句是否成立。

- \* 0 屬於偽值
- \* 空字串屬於偽值
- \* 如果一個字串的內容是"0"，也會被視為偽值。
- \* 一個undef的值也屬於偽值。

當然，有些運算式也是透過這些方式來判斷，我們可以輕易的找到例子來觀察Perl的處理方式。例如你可以看看Perl對這樣的判斷式怎麼處理：

```
my $true = (1 < 2);
print $true;
```



沒錯，回傳值是1，表示這是個真值，因此如果你在流程控制中用這樣的判斷式，很清楚可以知道流程的方向。

#### 4.1.3 區塊

在開始進入正式的判斷式之前，我們應該先來說說Perl程式中的區塊。在Perl中，你可以用一對大括號{}來區分一個Perl區塊，這樣的方式在程式的流程控制中其實非常常見。

在Perl的語法中，區塊中的最後一個敘述不必然要加上分號的，比如你可以這麼寫：

```
my $num = 3;
{
    my $max = $num;
    print $num
}
```

不過如果你未必覺得自己足夠細心的話，也許你該考慮留下這個分號，因為一但你的程式略有修改，你也許會忘了加上該有的分號。那你花在加這些分號的時間可能會讓你覺得應該隨時記得替你的敘述句加上分號才是。另外，區塊本身是不需要以分號作為結束的，不過你可別把區塊所使用的大括號跟雜湊所使用的混在一起。

#### 4.1.4 變數的生命週期

既然提到區塊，我們似乎應該在這裡稍微提起Perl裡面關於變數的生命週期。一般來說，Perl的生命週期都是以區塊來作為區別的。這和有些程式語言的定義方式似乎有些差距，當然，Perl的區分方式應該是屬於比較簡單的一種，所以一般而言，你只需要找到相對應的位置，就很容易可以知道某個變數現在是否還存在他的生命週期中。我們可以看個範例：

```
my $num = 3;
{
    my $max = $num;
    print $max;
}
print $max;
```

在程式裡，我們在區塊中宣告了變數\$max的存在，並且把變數\$num的值給了他。就在這一切的運算結束之後，我們進行了兩次的列印動作。而兩次列印分別在大括號的結束符號前後，表示一個列印是在區塊中進行，另一個則是在區塊結束後才列印。不過當我們試著執行這支程式時，發生了一個錯誤：

```
Global symbol "$max" requires explicit package name at ch2.pl line 12.
Execution of ch2.pl aborted due to compilation errors.
```

沒錯，我們在區塊內定義了變數\$max，也因此，變數\$max的生命週期也就僅止於區塊內，一但區塊結束之後，變數\$max也就隨之消失了。另外，我們也可以看看這個類似的例子：

```
my $num = 3;
{
    my $max = $num;
    print "$max\n";
}
{
```

```

my $max = $num*3;
print "$max\n";
}

```

這個例子中，我們看到了變數\$max被定義了兩次，可是這兩次卻因為分屬於不同的區塊，因此Perl會把他們視為是完全獨立的個體。也不會警告我們有個叫做\$max的變數被重複定義了。這看起來非常簡單吧？！這讓你可以你在需要的區塊裡，定義屬於那個區塊自己的同名變數，可是有時候其實你會把自己搞的頭暈，不信的話，你可以看看接下來的寫法：

```

my $a = 3;
my $b = 9;
{
    print "$a\n"; # 屬於外層的區塊，所以你會看到 3
    my $b = 6;    # 定義了這區塊內自己的變數
    print "$b\n"; # 於是你看到的這個$b的值其實是6
}
{
    print "$a\n"; # 這個區塊沒有自己的$a
    print "$b\n"; # 也沒有自己的$b
                  # 所以你在這裡看到的值其實是上一層的變數值
}
print "$a\n";    # 這裡似乎毫無疑問
print "$b\n";    # Perl 還是印出期待中的3跟9

```

## 4.2 簡單判斷

好極了，現在我們已經知道甚麼是真值，甚麼是偽值。這樣就可以運用在程式的流程判斷了。

### 4.2.1 if

if的判斷非常的直覺，也就是說，只要判斷式傳回真值，程式就會執行條件狀況下的內容。這是一個非常簡單的例子：

```

my $num = 3;
if ($num < 5) {
    print "這是真的";
}

```

沒錯，這個程式雖然簡單，但卻很清楚的表達出if判斷式的精神。在(\$num < 5)裡，Perl傳回一個真值，於是我們就可以執行接下來的區塊，也就是列印出字串"這是真的"。

提示：

由於這些判斷式會用到大量的二元運算符，為了避免執行上產生難以除錯的問題，我們在這裡提醒各位一些容易忽略的部份。

"<", ">", ">=", "<=", "==", "!="：這些算符都是在針對數字時用到的比較算符。

"eq", "lt", "gt", "le", "ge", "ne"：如果你是對字串進行比對，請記得使用這些比較算符。

### 4.2.2 unless

和if相對應的，就是unless了。其實在其他程式語言，很少使用unless的方式來進行判斷。因為我們可以使用if的否定來進行同樣的工作例如你可以用

```
if (!$a < 3)
```

這樣的方式來描述一個否定的判斷句。可是利用否定的運算符"!"來進行判斷顯然不夠直覺，也因此比較容易

出錯。這個時候unless就顯得方便多了。從口語來看，if敘述就是我們所說的「假如...就...」，而unless就變成了「除非...就...」。這樣在運算式看起來，就顯得清楚，也清爽多了。所以你可以寫成：

```
unless ($a < 3)
```

這樣的寫法跟上面的那個例子是一樣的效果，不過在易讀性上明顯好了許多。尤其當你的判斷式稍微複雜一些，你就更可以感受到unless的好用之處了。

#### 4.2.3 一行的判斷

在許多時候，我們會用非常簡單的判斷來決定程式的走向。這時候，我們便希望能以最簡單的方式來處理這個敘述句。尤其當我們進行了判斷之後，只需要根據判斷的結果來執行一行敘述時，使用區塊的方式就顯得有點冗長了，比方你有一個像這樣的需求：

```
if ($num < 5) {  
    $num++;  
}
```

這樣的寫法確實非常工整，可是對於惜字如金的Perl程式員來說，這樣的寫法似乎非常不經濟。於是一種簡單的模式被大量使用：

```
$num++ if ($num < 5);
```

你沒看錯，確實就是如此，把判斷句跟後續的運算句合併為一個運算式。而且這種用法不僅止於if/unless判斷式，而是被大量使用在許多Perl的運算式中。我們以後還會有機會遇到。不過先讓我們繼續往下看。

#### 4.3.4 else/elsif

你總是有很多機會使用到if/unless判斷式，而且常常必須搭配著其他的判斷才能完整的讓你的程式知道他該做甚麼事。這個時候，比如你也許會想這麼寫：

```
if ($num == 1) { ... }  
if ($num != 1) { ... }
```

這樣的程式雖然也對，不過總覺得那裡不太對勁，畢竟這兩個判斷式顯然正在對同一件事進行判斷，不過卻必須分好幾個敘述句。當然，如果你的判斷還是簡單（像我們的例子所寫的）也就還能手工進行。可是如果你的判斷式長的像這樣呢？

```
if ($num == 1) { ... }  
if ($num == 2) { ... }  
if ($num == 3) { ... }  
.....  
if (($num != 1) && ($num != 2) && ($num != 3) && ...) { ... }
```

沒錯，你現在可以想像人工進行這件事情的複雜度了吧！所以如果有簡單的方式來進行，同時還能增加程式的易讀性，似乎是非常好的主意，而else/elsif就是這個問題的解答。

如果我們在一個，或一大堆if判斷式的最後希望能有一個總結，表示除了這些條件之外，其他所有狀況下，我們都要用某個方式來處理，那麼else就是非常好的助手。最簡單的形式大概就會像這樣：

```
if ($num == 1) {  
    ...  
} else { # 其實這裡就是 if ($num != 1) 的意思了
```

```
    ...  
}
```

不過如果我們有超過一個判斷式的時候，就像之前的例子，我希望\$num在1..3的時候，能有不同的處理方式，甚至我如果進行一個禮拜七天的工作，我希望每天都能有不同的狀況，那只有else顯然不夠。我總是不希望每次都來個if，到然後還要判斷使用者打錯的情況。這時候，elsif就派上用場了，你每次在其他條件下，如果還要訂下其他的條件，那麼你就可以寫成像這樣：

```
if ($date eq '星期一') {  
    ....  
} elsif ($date eq '星期二') {  
    ....  
} elsif ($date eq '星期三') {  
    ....  
    ....  
} else {  
    print "你怎麼會有$date\n";  
}
```

其實，利用if/else/elsif已經可以處理相當多的問題，可是在許多程式語言中還可以利用switch/case來進行類似的工作。在Perl中，也有類似的方式，這是由Damian Conway寫的一個模組，目前已經放進Perl的預設套件裡了。不過我們並不打算在這裡增加所有人的負擔。

#### 4.3 重複執行

我們剛剛所提到的只是對於某個條件進行判斷，並藉由判斷的結果來決定程式的流程，因此條件的不同會讓程式往不同的地方繼續前進。不過很多時候我們需要在某些條件成立的時候進行某些重複的運算，比如我們希望算出10!，也就是10的階乘。這就表示只要我們指定的數字不超過10，就讓這個運算持續進行，這時候，我們顯然需要進行重複的運算。

##### 4.3.1 while

while就是一個很好的例子，讓我們來看看怎麼利用while來完成階乘的例子：

```
my $num = 1;  
my $result = 1;          # 小心，這裡一定要指定$result為1  
while ($num <= 10) {     # 確定你是否超過範圍  
    $result*=$num;  
    $num = $num + 1;  
}
```

看起來不難吧！你只要掌握幾個原則，理論上就可以很容易讓while迴圈輕鬆上手。

首先，你總得讓你的迴圈有正常運作的機會。當然你如果不希望這個迴圈有任何機會執行，Perl也不會在你的耳邊大叫，不過維護你的程式的人大概會很難理解這一段不可能執行到的程式碼有甚麼功用吧。

其次，別忘了讓你的程式有機會離開他的迴圈，除非你知道自己在作甚麼否則你的程式會不斷的持續進行，當然，那就是所謂的無限迴圈。例如你寫了一個非常簡單的程式：

```
while (1) {              # 在這裡，程式會得到永遠的真值  
    print "這是無限迴圈";  
}
```

第三，在這裡，你還是可以讓只有單一敘述的while迴圈利用倒裝句達成：

我們假設你完全知道剛剛的程式會發生甚麼事，而那正是你所希望達成的，那麼我們就可以來改寫一下，讓他變得更簡潔：

```
print "這是無限迴圈" while (1);
```

別怪我們太囉唆，不過這樣的寫法確實讓程式乾淨許多，而且許多Perl程式設計師（也包括我自己在內）非常喜歡這樣的用法，如果你有機會讀到別人的程式，還是先在這裡熟悉一下吧。

#### 4.3.2 until

類似if/unless的相對性，你也可以用until來取代while的反面意義，例如你可以用until來作剛剛階乘的同樣程式。語法其實跟while一樣：

```
until (判斷式) {  
    ....  
}
```

雖然語法看起來完全一樣，不過如果是剛剛的階乘，判斷式就會變成這樣：

```
my $num = 1;  
my $result = 1;  
until ($num > 10) {  
    $result*=$num;  
    $num = $num + 1;  
}
```

#### 4.4 for

for迴圈也是非常有用的迴圈，尤其在你使用陣列時，你可以很方便的取出所有陣列中的元素。而且你幾乎不需要知道現在陣列中有多少元素，聽起來非常神奇不是嗎？不過先讓我們來看看for到底是怎麼用的呢？

##### 4.4.1 像 C 的寫法

如果你寫過C語言，你應該對這樣的寫法非常熟悉，所以你應該可以直接跳過這一小段。當然，我們假設大部份的人都不熟C，那麼我以為，如果你覺得太累，也可以晚一點再回來看這一段。因為作者個人的偏見，以為這雖然是Perl的基本語法，可是在實際程式寫作時用的機會卻比其他方式少了一些。不過我想大家都是好學生，還是讓我們來看看基本的for迴圈應該怎麼寫呢？還是維持我們的傳統，來看看這個例子吧：

```
my $result = 1;  
for (my $num = 1; $num <= 10; $num = $num + 1) {  
    $result *= $num;  
}
```

跟剛剛的while/until終於有些不太一樣，而且主要的差別似乎在於這一行：

```
for (my $num = 1; $num <= 10; $num = $num + 1)
```

沒錯，這一行確實就是for迴圈的奧秘所在。首先我們看到小括號裡面的三個敘述，這三個分別代表迴圈的初值，迴圈的條件，以及每次迴圈進行後所作的改變。從這個例子來看，我們先定義了一個變數\$num，初值是1。接下來要求迴圈執行，只要\$num在不超過10的狀況下就不斷的執行，最後一個則表示，當迴圈每做一次，\$num的值就被加1。

當然，這三個部份都是獨立的，所以如果你有比較特殊的判斷方式，也可以隨意修改。例如你可以用總和超過/不超過多少來作為判斷的依據。或者每次在執行完一次迴圈就把\$num的值加3，這樣的寫法對於應付比



較複雜的狀況顯然非常好用。

不過如果我們大部份的時候都只是非常有規律的遞增，或遞減，並且以此為判斷迴圈是否應該結束的依據。既然如此，也許我們還有更清楚，而且簡單的方式嗎？

#### 4.4.2 其實可以用 ...

大家對於Perl程式設計師喜歡簡單的體會應該已經非常深刻了，因此我們就來看看怎麼樣可以讓非常具有規則性的for迴圈可以用更簡單的方式來表達吧！例如像我們前面提到的例子：

```
for (my $num = 1; $num <= 10; $num = $num + 1)
```

這麼簡單的for迴圈還要寫這麼長一串真是太累人了，記得有一種非常簡單的語法嗎？

```
for $num = 1 to 10
```

沒錯，如果Perl也可以這樣寫那就非常口語化了。而且再也不用那麼長的敘述句只為了告訴Perl：「給我一個1到10的迴圈吧！」不過很顯然，Perl的程式設計師找到了更簡便的方法，你只要用...就可以表示你需要的迴圈範圍了。

```
for my $num (1...10) {    # 這就是表示$num從1到10
    print $num;
}
```

這其實可以寫成：

```
for (1...10) {            # $_ 經常被拿來作為迴圈的預設變數
    print $_;
}
```

更簡化的寫法：

```
print for (1...10);
```

覺得很神奇嗎？其實一點也不會，因為這正是Perl程式設計師經常在使用的方式。而你需要更熟悉的也許就是要習慣於這些人對於習慣寫作的方式。

#### 4.4.3 有趣的遞增/遞減算符

如果你寫過C語言程式，或是你看過其他人寫Perl的迴圈，應該會常常看到這樣的敘述句：

```
print for ($i = 1; $i <= 10; $i++);    # 印出 1...10
```

可是對於遞增(++)與遞減(--)運算子卻又是似懂非懂。那麼這兩個運算子到底在說些甚麼呢？不過顧名思義，他們主要的工作就是對數字變數進行遞增或遞減的運算。例如你也許會這麼用：

```
my $i = 1;
while ($i <= 10) {
    print $i;
    $i++;                # 把 $i 加上 1
}
```

沒錯，於是在 while 迴圈中，\$i就會由1到10，靠的正是這個遞增運算子。當然，在這裡你也可以把這個式子替換為：

```
$i = $i + 1; # 或是 $i += 1
```

不過看起來總是沒有\$i++簡潔吧！同樣的，遞減運算子(--)也是進行類似的工作，也就是每次把你的數值減1。不過遞增或遞減運算子總有時候會讓人感覺困擾，讓我們來看看以下的例子：

```

my $i = 1;
while ($i <= 10) {
    print ++$i. "\n";           # 印出 2...11
}

my $j = 1;
while ($j <= 10) {
    print $j++. "\n";          # 印出 1...10
}

```

從這個例子來看，應該比較清楚，在第一個迴圈中，Perl會先幫*\$i*加1之後印出，也就是根據遞增（遞減）運算子的位置來決定如何運算。當然，我們就可以確定在第二個迴圈中，遞增運算子是怎麼運作的。當遞增運算子在前時，Perl會先對運算元進行運算，就像第一個迴圈的狀況，反之，則是在進行完原來的運算式之後，才進行遞增（或遞減）的運算，也就是像第二個迴圈中所看到的結果。

#### 4.4.4 對於陣列內的元素

我們當然可以用 `for (;;)` 這樣的方式來取出陣列中的所有元素來運算。不過這時候你只是依照陣列的索引順序，因此你還需要根據索引的值來取得陣列中的元素值。就像這樣子：

```

my @array = qw/1 2 3 4 5/;
for (my $i = 0; $i < $#array; $i++) {
    print $array[$i];
}

```

顯然這樣的寫法太過繁瑣，我們其實可以利用`foreach`來進行更簡單的取值動作。那麼剛剛的迴圈部份可以寫成：

```
print foreach (@array);
```

不要懷疑，就是這麼簡單，不過讓我們先來解釋一下整個迴圈的運作。

當你對一個陣列使用`foreach`迴圈時，Perl就會自動取出這個迴圈每一個值。接下來，你可以指定Perl把取得的值放到某個變數中，例如你可以寫成：

```
foreach my $element (@array)
```

不過這時候我們就用到了Perl最常用到的預設變數`$_`，當我們在迴圈中沒有指定任何變數時，Perl就會把取出來的值放入預設變數`$_`中。緊接著我們希望把迴圈取得的值列印出來，也就是執行

```
print $_;
```

這樣的式子，當然，這樣的式子在Perl出現的機率真是太少，因為大部份的時候，如果你只要列印單一的`$_`，Perl程式員也就會省略`$_`這個變數。而因為我們在迴圈中只打算執行`print`這個指令，所以倒裝句也就順勢產生。看起來應該顯得非常簡潔吧！現在你應該還不習慣這樣的寫法，不過如果你有機會接觸其他Perl程式員寫的程式，那千萬要慢慢接受這樣的寫法。

那麼還有一個問題，那就是`foreach`是否只能用在迴圈的取值，或是`foreach`跟`for`該怎麼區別他們的用法呢？這個問題倒很容易，因為在Perl之中，`foreach`跟`for`所進行的工作基本上是一模一樣，或說他們之中的任何一個都只是另一個的別名。因此只要你可以使用`for`的地方，即使你將他替換為`foreach`也完全可以被Perl接



受，不過有時候也許你會希望以foreach來表達你的算式能夠讓可能維護你的程式的人比較容易接受，當然，很多時候Perl程式員確實不願意多打四個字母，所以你被需按照當時的語境確定實際執行的是for或foreach的語意了。

還記得我們在上一章提出的問題嗎？如果我們有一個含有整數串列的陣列，而我們想取出其中大於零的整數然後取他們的平方值，那麼我們應該怎麼作呢？現在我們可以嘗試來玩玩這個問題。我們先用迴圈的方式來解決這個問題：

```
my @array = qw/6 -4 8 12 -22 19 -8 42/;
my @positive;
for (@array) {
    push @positive, ($_**2) if ($_ > 0); # 針對陣列的每個元素檢察
}                                     # 如果大於零就取平方值

print for (@positive);
```

那如果使用我們在上一章介紹的函數呢？

```
my @array = qw/6 -4 8 12 -22 19 -8 42/;
my @positive = map { $_**2 }
                grep { $_ > 0 } @array;
# 倒裝，把@array的元素先放進 grep 檢查，再把通過檢查的結果利用map取得平方值放進新的陣列

print for (@positive);
```

相當有趣吧，這也就是Perl的名言：「解決事情的方法不只一種。」

習題：

1. 算出1+3+5+...+99的值
2. 如果我們從1加到n，那麼在累加結果不超過100，n的最大值應該是多少？
3. 讓使用者輸入一個數字，如果輸入的數字小於50，則算出他的階乘，否則就印出數字太大的警告。



# Perl 學習手札



There is more than one way to do it.

## 5. 雜湊(Hash)

雜湊對一般使用者大概都非常不熟悉，尤其是沒接觸過Perl的人來說，雜湊對他們來說都是全新名詞。但是在現實生活中，雜湊卻是不斷出現在一般人的生活之中。因此只要搞懂雜湊到底在講甚麼，你就會覺得這個東西用起來真是自然極了，而且沒有了雜湊還可能讓很多事情顯得不知所措，因為你要花大量的時間跟精力才能利用其他資料結構做出雜湊所達到的結果。

聽起來雜湊確實非常吸引人，那我們先來了解一下甚麼是雜湊。所謂的雜湊，其實用最簡單的話來說，也就是一對鍵值(key-value)，沒錯，就是這麼簡單，一個鍵搭配著一個值的對應方式。當然，你可以搭配Perl所提供複雜的方式來建立多層的雜湊來符合程式的需要，不過那不是基本的雜湊，而且所謂複雜的結構，還是依循著最簡單的原理，也就是鍵跟值的相對應關係。

### 5.1 日常生活的雜湊

沒錯，如果我們只說雜湊是一對鍵值的組合，那要讓人真正理解顯然並不容易。所以如果我們可以使用一般人常用的詞彙來解釋雜湊這個東西，顯然應該會容易許多。既然如此，我們就來看看大家每天接觸的資料中，有甚麼是能夠以雜湊精確的表現出來的。

最簡單的例子應該算是身份證字號了吧，我們可以很容易的用身份證字號知道一個人的姓名，其中的身份證字號就是雜湊的鍵(key)，而利用這個鍵所得到的值(value)就是姓名。而且鍵是這個雜湊中唯一的值，也就是一個雜湊中，不能有重複的鍵。這也應該很明顯，如果有兩個一模一樣的身份證字號，那麼我們要怎麼確認使用者希望找到的是那一個呢？所以這也是雜湊中的限制，我們必須要求雜湊中的鍵值必須是不重複的，很顯然，這樣的限制是非常合理的。另外，我們每個人的行動電話中也藏著使用雜湊的好素材。如果你曾經使用行動電話的電話簿功能，那麼你也許每天都會接觸這種非常雜湊式的結構，因為電話簿功能也是雜湊足以發揮功用的地方。電話簿就是一整個的雜湊，他裡面的鍵是以姓名為主，值則是這個人的電話。所以你必須為每一個人鍵入一個獨特的鍵，大多也就是名字，以及這個鍵所對應的值，當然就是電話了。因為我們只要找到鍵（姓名）就可以查到依附在這個鍵的值（電話）。如此一來，我們應該很容易可以理解雜湊的代表意義了。

### 5.2 雜湊的表達

雜湊在Perl中是以百分比符號(%)作為表示，變數的命名方式則維持一貫原則，也就是可以包含字母，數字及底線的字串，但是不能以數字作為開頭。所以你可以像這樣的方式定義一個雜湊變數：

```
my %hash;           # 基本的命名方式
my %ID_Hash;        # 包含底線的變數
my %i d_hash;       # 大小寫還是被認為是不同的字串
my %_underline;     # 以底線開始的變數名稱
my %2hash;          # 程式會產生錯誤，因為這不是合法的變數
```

雜湊的存取，我們可以利用大括號來進行，因此我們把所想要取得的雜湊鍵放入大括號中，就可以藉此找到相對應的值。同樣的方式，我們也可以利用這樣的形式指定某一對鍵值，這樣的作法非常接近我們存取陣列的形式：

```
my %hash;
$hash{key} = 'value'; # 最簡單的賦值形式
print $hash{key};
```

就如我們說的，我們是使用大括號({})來標示所要存取的雜湊鍵，這和使用陣列是不同的。不過更重要的是千萬別把你的程式寫得像這個樣子：

```
my $var = 1;
my @var = (1, 2, 3, 4, 5, 6);
my %var;
$var{1} = 2;
$var{3} = 4;
$var{5} = 6;

print $var[2];
```

這樣的形式對於Perl來說當然是合法的，不過我們顯然不希望你用這樣的形式來寫程式，否則即使Perl可以很容易的分辨出來，只怕寫程式或維護的人自己還先搞混了。有時候，我們會忽略一些小地方，那就會讓自己找不到雜湊中的值，其中有一個非常重要的部份，也就是雜湊鍵的資料型態。Perl會把雜湊鍵全部轉為字串，這樣的轉換其實是有些道理的。我們來研究一下這樣的程式會發生甚麼狀況呢：

```
my %hash;
$hash{2} = 'two';           # 指定雜湊的一對鍵值
$hash{'4/2'} = '這是字串 4/2'; # 注意引號的使用
print $hash{4/2};          # 先運算後轉為字串的鍵
```

你認為Perl會輸出甚麼樣的結果呢？答案是'two'。沒錯，很有趣吧，所以你可以在雜湊鍵的地方放上一個運算式，那麼Perl會先進行運算，然後把運算結果轉為字串，所以上面的例子，我們所要求Perl輸出的其實是\$hash{2}，否則你可以利用引號來指定字串，就像\$hash{'4/2'}這樣的方式。我們再看看另一個例子：

```
my %hash;
for (1..5) {
    $hash{$_*2} = $_**2;
}
```

那我們可以得到的雜湊就是像是這個樣子：

```
$hash{2} = 1;
$hash{4} = 4;
$hash{6} = 9;
$hash{8} = 16;
$hash{10} = 25;
```

沒錯，正如我們所預料的，Perl會把運算出來的結果轉為字串後當成雜湊的鍵。還記得我們可以利用字串的內插方式來插入變數到字串嗎？你可以猜測以下的程式會產生出甚麼不同的結果：

```
my %hash;
for (1..5) {
    $hash{"$_*2"} = $_**2;
}
```

如果你可以想辦法看到雜湊的內容，你會發現你得到的雜湊鍵變成了 "1\*2", "2\*2".....。沒錯，因為他們被視為一個字串了。所以如果你以為你可以利用\$hash{2}或\$hash{4}來得到雜湊內的值，恐怕會失望了。所以當你要開始使用雜湊時，可就要小心別搞混了。

### 5.3 雜湊賦值

我們剛剛學到了利用 \$hash{2} = 4; 這樣的方式來指定一對鍵值給雜湊，沒錯，這是賦值給雜湊的最基本方式，不過就跟我們使用陣列一樣，我們經常需要一次指定大量的雜湊鍵值，想必Perl的開發者一定也會遇到相同的問題，而且應該有一些合理的解決方案。既然如此，我們應該有其他方式可以一次指定超過一組的鍵值。利用串列的方式賦值給雜湊就是其中之一，而且當你在定義某個雜湊時就預先知道他的一些鍵值時特別有用，看看下面的例子：

```
my %hash = qw/1 one 2 two 3 three/;
```

這樣的賦值方式看起來跟處理陣列時候的方式非常接近，我們利用qw//來指定一個串列，並且將這個串列賦值給雜湊。這時候，Perl會按照串列的順序，分別為【鍵】，【值】，並且賦予雜湊。所以在這個例子中，所得到的結果就跟我們這麼寫是一樣的：

```
$hash{1} = 'one';  
$hash{2} = 'two';  
$hash{3} = 'three';
```

或許你會想到某個狀況，也就是鍵值的個數不一的時候。這時候，Perl會把最後一個鍵所對應的值設為undef (註二)，你可以利用這個程式來確認：

```
my %hash = (1, 2, 3, 4, 5);  
print 'false' unless defined($hash{5});
```

當然利用串列賦值的方式是方便了一些，可是就像我們剛剛遇到的問題，有時候會發現利用串列賦值的情況似乎比較容易發生錯誤。尤其當一個串列的元素足夠多的時候，你要怎麼確認某個串列中的元素應該是鍵，還是值呢？最簡單的方式大概就是進行人工比對，所以你或許可以考慮用另外的方式來賦值給雜湊，就像這樣的寫法：

```
my %hash = (  
    1 => 'one',  
    2 => 'two',  
    3 => 'three',  
);
```

在這裡，我們利用箭號(=>)來表示雜湊中鍵跟值的相對關係，而且在一對鍵值的後面加上逗號作為區隔。這樣的方式就顯得方便、也直覺了許多。不過當你在使用箭號進行指定時，你可能會發現一些不同。因為箭號左邊的雜湊鍵已經完全被視為一個字串，所以你如果使用這樣的方式：

```
my %hash = (  
    4/2 => 3,
```



```
);
print $hash{'4/2'};
print $hash{2};
```

別忘了，跟之前的狀況一樣，Perl還是會幫你先把箭號左邊的運算式算出結果，然後轉成字串，作為雜湊的鍵。所以當你在取值時使用了引號確保你要找雜湊鍵等於'4/2'的值時，你就沒辦法找到任何結果，因為目前雜湊中只有一個雜湊鍵為'2'的值。

要從雜湊中取出現有的值以目前的方式應該足夠方便，你只需要知道雜湊中的鍵，就可以取得他的內容值。不過這樣顯然還不夠，因為雜湊跟陣列還是有著相當的差異。在陣列中，你可以很清楚的知道陣列的索引值是從0到最後一個陣列的大小減1，可是在雜湊中卻並不是這麼一回事。如果你沒辦法知道雜湊的鍵，又怎麼取出他的值呢？那麼這個時候，你應該考慮先把整個雜湊讀過一次。

#### 5.4 each

就像在陣列當中，你可以使用foreach這樣的迴圈來找到陣列中的每一個值，當然我們也經常需要在雜湊中進行類似的工作，我們希望可以在雜湊中能一次取出所有的鍵，值。所以你必須仰賴類似foreach的工具來幫助你，那就是each函數。例如你可以利用下面的寫法讀出剛剛我們所建立起來的雜湊：

```
while (my ($key, $value) = each (%hash)) {
    # 取出雜湊中的每一對鍵值，並且分別放入$key, $value
    print "$key => $value\n";
}
```

很明顯的，每次each函數都會送回了一個包含兩個值的串列，其中這兩個值分別是一個雜湊鍵跟相對應的值。因此我們把取回的串列指定給\$key和\$value兩個變數，接著印出結果，就可以看到一對一對的鍵值了。而當傳回空陣列時，while判斷就會變成偽值，while迴圈也就結束了。利用這樣的函式對我們有很大的幫助，如果我們想要整理一個雜湊的內容，我們可以在完全不知道雜湊中有什麼內容的狀況下開始進行處理。使用each函數在處理雜湊時是讓事情顯得容易許多，可是有時候還是有點不方便的地方，舉例來說：如果我有一個包含著主機ip跟主機名稱的雜湊，雖然我不知道雜湊裡面到底有多少資料，可是我卻希望能找出所有的雜湊鍵值，然後取出以192開始的ip位址。這時候如果使用each來作，那就必須先把所有的鍵值取出，然後再一一進行比對，所以也許程式就像這樣：

```
my %hash = (
    '168.1.2.1' => 'verdi',
    '192.1.2.2' => 'wagner',
    '168.1.2.3' => 'beethoven',
);
my @hostname;
while (my ($key, $value) = each (%hash)) {
    if ($key =~ /^192/) {
        push @hostname, $value;
    }
}

print @hostname;
```

很顯然，這樣的寫法確實可以讓程式正確的找出我們要的結果，不過我們總是還會繼續思考可以有更乾淨俐落的寫法，畢竟使用Perl的程式設計師都不太喜歡拉拉雜雜的程式。所以有甚麼方法可以讓過濾出需要的鍵值可以顯得方便些呢？

#### 5.5 keys跟values

如果我們可以用簡單的方式一次取得雜湊的所有鍵(keys)，那麼要進行過去的過程就非常容易，而我們所需



要的就是過濾後留下來的鍵，跟他們的相對值。當然，有某些時候，你可能只想要拿到雜湊中的所有值，這時候你就不需要擔心他們是屬於什麼鍵的相關。為了因應這樣的需求，有兩個函數可以滿足我們，他們分別是keys跟values。很顯然的，這兩個函數所作的工作就是取出雜湊的鍵跟值。和使用 each相當不同的是：你可以只單讀取出所有的鍵，或所有的值，而不需要一次全部取出。

例如我們可以用這樣來把雜湊鍵放在同一個陣列中：

```
my @keys = keys(%hash);
```

如果你希望取出所有的值，那麼不妨這樣寫：

```
my @values = values(%hash);
```

當然，你可以用他來完成each的工作，就像這樣：

```
my @keys = keys(%hash);
for (@keys) {
    print "$_ => $hash{$_}\n";
}
```

其實跟這麼寫是一樣的效果：

```
while (my ($key, $value) = each(%hash)) {
    print "$key => $value\n";
}
```

不過你顯然會發現，有時候用keys/values比較簡單，有時候用each比較方便，當然，至於要使用何者是完全取決於你所想要得出的結果，或者你認為最省力，簡潔，或是效率比較好的寫法。

在雜湊中使用keys/values這兩個函數都傳回串列，因此我們可以把我們所得到的串列輕易的放入陣列，接下來再以陣列的方式進行運算。這樣的方便之處在於我們可以有許多可供利用的陣列函數，所以我們可以把剛剛的那個例子改寫成這樣：

```
my %hash = (
    '168.1.2.1' => 'verdi',
    '192.1.2.2' => 'wagner',
    '168.1.2.3' => 'beethoven',
);

my @keys = map { $hash{$_} }
            grep { (m/^192/) } keys(%hash);

print @keys;
```

這樣的寫法比起之前的方式看起來是不是乾淨許多了呢？我們來看看最關鍵的一行，結果到底怎麼產生的：我們先用keys函數取出雜湊中的所有鍵，就如我們所說的，這個函數傳回一個串列。然後我們對所得到的串列進行過濾，利用grep取出串列中以192開頭的ip子串列，最後利用map一一比對得出雜湊中以對應這些ip的主機名稱。

## 5.6 雜湊的操作

毫無疑問，雜湊這樣的資料結構對於程式的寫作有著莫大的幫助，但是我們必須能熟悉對雜湊的操作才能夠讓我們更容易發揮雜湊的功能。其中最重要的大概就是exists跟delete兩個函數了，這兩個函式能讓我們有效的掌握雜湊的元素，同時它們也是perl內建相關於雜湊函數的最後兩個(註一)。

### 5.6.1 exists

我們就繼續用ip跟主機的雜湊當例子吧。假如我有一個ip，我不確定我是否有這部主機的資料，如果我們只用剛剛的方法，那我們就必須取得所有的ip，然後把手上的ip跟取得的ip串列一一比對，以便確定自己有沒有這個ip的主機資料。所以我們的程式也許長的像這樣：

```
my %hash = (
    '168.1.2.1' => 'verdi',
    '192.1.2.2' => 'wagner',
    '168.1.2.3' => 'beethoven',
);
my $ip = '192.1.2.2';
print "bingo" if ($hash{$ip});
```

在這裡，我們有一個雜湊，其中三個鍵分別是'168.1.2.1'，'192.1.2.2'，'168.1.2.3'，而我們希望判定目前手上的一組ip'192.1.2.2'是不是我們主機所擁有的ip。於是我們利用這個ip作為雜湊鍵，並判斷如果取得的值為真，那麼我們就說這個ip屬於雜湊的其中一個鍵，這樣的想法似乎暫時解決了我們的需求。不過我們來看看下面的例子：

```
my %hash = (
    'cd' => 2,
    'book' => 10,
    'video' => 0,
);
my $media = 'video';
print "bingo" if ($hash{$media});
```

我們假設這是某個社區圖書館目前外借的東西數量，其中的鍵就是代表則可以外借的圖書館資產，其中包含了CD，書跟錄影帶。而所對應到的值則是他們目前被借出的數量。我們看到，CD被借走了兩套，書被借走了十本，而錄影帶則是原封不動，一卷也沒被借走。是的，大家都不喜歡錄影帶了。

這時候，我們希望知道圖書館是否提供錄影帶外借，也就是要檢查video這個鍵是否存在。於是我們利用剛剛的方式，看看\$hash{\$media}是否傳回真值。很遺憾，因為錄影帶這個鍵目前的值是0，因此當我們利用錄影帶當成鍵來取的相對應的值時，Perl會傳回0給我們。而我們知道0其實是個偽值。於是我們以為'video'這個鍵並不存在於這個雜湊中，也就是說這個圖書館並沒有錄影帶出借，但是這樣的結果跟我們的認知有所不同，因為取得的值為0只是代表目前沒人借出。所以我們發現這個方法並不正確，至少我們已經知道他會產生錯誤的結果。所以我們必須嘗試其他方法，例如利用keys找到包含所有索引鍵的串列，然後進行一一的比對。就像這樣：

```
print "exist" if (grep { $_ eq 'video' } keys (%hash));
```

這樣就可以確定某個鍵是否存在於這個雜湊，可是程式還是有點長，而且我們也許必須經常去判斷某個值是否為雜湊的鍵。所幸Perl提供了簡潔的函式可以使用，所以利用exists這個函式讓我們有了極佳的判斷方式。有了exist之後，對於剛剛那一行程式，我們只需要這麼改寫：

```
print "exists" if (exists $hash{video});
```

這樣的寫法顯然輕鬆了許多。

### 5.6.2 delete

有些時候，我們也會遇到某些鍵值我們不再需要的狀況，這時候如果可以把這些沒有必要的鍵值移除似乎是非常必要的。所以Perl也提供了移除雜湊鍵值的函式，也就是delete。這個函式的使用其實非常容易，你只需要指定想要刪除的某一個雜湊鍵，就像這樣：

```
delete $hash{video};
```

當然，所謂的移除是指這個鍵將不再存在於這個雜湊，而不是指讓這個鍵對應的雜湊值消失。所以並不是把需要被delete的這對鍵值設為undef。也就是說，即使有一個鍵所對應的雜湊值為undef，那這個鍵依然被視為存在(exists)的，這在剛剛解釋exists這個函數的例子中就可以了解了。

### 5.7 怎麼讓雜湊上手

在Perl中要使用雜湊，有一些重點也許還是應該提醒大家的。首先，Perl對於雜湊的大小限制依然採取了「放任」的態度，也就是以最沒有限制的方式。只要電腦可以容量的大小，Perl都可以接受。因此程式設計師可以有很大的揮灑空間，只是也必須注意避免讓系統因為被Perl佔用太多資源而導致無法正常運作。另外，使用者可以利用任何的純量值來表示雜湊中的鍵與值。可是在雜湊鍵的部份，Perl會把所有的鍵轉換為字串。所以如果你在不注意的情況下把運算式當成雜湊的鍵，Perl會幫你先進行運算，然後利用運算所得的結果作為雜湊鍵，這樣的情況可能會有出乎意料的結果。當然，如果你使用運算式來作為雜湊的鍵值，那就應該有些準備，因此應該會更小心的注意，而我們也在前面提到了不少例子。

另外，你還會希望知道自己甚麼時候該用雜湊，這就必須依賴你對於雜湊的感覺，最基本的原則還是以雜湊的特性來看，如果你有一個可以辨識的鍵，而且希望藉由這個鍵找到相關連的值，這時候你幾乎就可以放心的使用雜湊了，只不過這裡所謂的值當然不限定單指特定的值，而可能是任何一種純量值，也就是因為這個特性，可以讓我們搭建出複雜的雜湊結構，不過這個部份則是屬於進階的內容，我們就不在這裡解釋。就像我們所舉的例子，你可以利用ip作為每一部主機的辨識，那麼你可以藉由ip找到那部機器的相關資料。還有一個常常被搞混的問題，也就是雜湊的順序。許多人想當然爾，以為雜湊的順序是依照新增的順序來決定的。其實事實並非如此，雜湊的排列方式並非按照使用者加入的順序，而是Perl會依照內部的演算法找出最佳化的排列。

習題：

1. 將下列資料建立一個雜湊：

John => 1982.1.5

Paul => 1978.11.3

Lee => 1976.3.2

Mary => 1980.6.23

2. 印出1980年以後出生的人跟他們的生日。

3. 新增兩筆資料到雜湊中：

Kayle => 1984.6.12

Ray => 1978.5.29

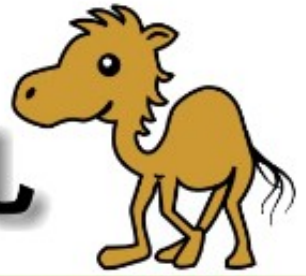
4. 檢查在不修改程式碼的情況下，能否達成第二題的題目需求

註一：可以利用perldoc perlfunc來查看perl所提供的函數。

註二：其實，如果你在程式裡打開了警告訊息的選項，這樣的指定會讓Perl產生警告訊息："Odd number of elements in hash assignment"。



# Perl 學習手札



There is more than one way to do it.

## 6. 副常式

當你的程式在很多時候總是不斷在進行類似的工作時，而且你總是為了這些工作在寫相同的重複程式碼時，你應該考慮替你自己把這些段落用更簡潔的方式來萃取出一個可以重複使用的區塊。這時候，你就可以考慮使用副常式，不過如果要更清楚的了解副常式的意義，也許我們應該先來看看這樣的例子：

```
my @array = qw/6 8 -4 18 -9 -22 36 48/;
my @new_array = map { ($_**2) }
                  grep { $_ > 0 } @array;
print @new_array;
print "\n";
my @array2 = qw/16 8 -24 8 -12 20 16 28/;
my @new_array2 = map { ($_**2) }
                  grep { $_ > 0 } @array2;
print @new_array2;
```

從這幾行程式，你應該很容易看出一些端倪，因為我們發現在這個小小的程式中，幾乎除了陣列中的元素不同之外，其他的幾行程式顯然都在進行相同的工作。也許你覺得只有兩個陣列需要運算還算容易，不過當你需要進行相同運算的陣列達到十個，或二十個的時候，你總不希望又把相同的程式碼進行大量的複製吧？也許你會認為複製，貼上這樣的動作總比弄懂副常式來得簡單許多，不過你也許需要考慮，當你的程式需要進行修改怎麼辦？讓我們來舉一個例子吧，如果你的程式在取平方值的地方現在希望可以取立方值，或甚至進行更複雜的運算，而你現在總共複製了一百份的程式碼在做同樣的運算。沒錯，恭喜你，你現在就因為這樣小小的改變而必須修改這一百個地方，而且這對程式的除錯會變成非常大的負擔。除非你認為你的工作時間就應該花在這樣的工人智慧上，否則你確實要考慮來使用副常式了。當然，如果你還是堅持不用副常式，千萬別找別人去維護你的程式。

使用副常式不但可以增加程式的可重用性，降低維護成本，當然還可以提昇程式的可讀性。就像剛剛的例子，你對於每次需要進行某個運算時，只需要使用不同的參數，而如果運算部份的程式需要修改時，也只需要修改一次。對於想要看程式碼的人來說，他也可以清楚的看懂這一部份到底進行甚麼樣的工作。因此適當的在自己的程式中使用副常式確實是非常必要而且提高效率的方式。

### 6.1 關於Perl的副常式

在Perl中，使用副常式並不困難，尤其當你曾經使用其他程式語言寫過副常式，那麼Perl的副常式對你來說更是容易上手。不過我們假設你從來沒寫過任何程式語言，那麼我們準備從Perl來學習副常式了。

在Perl中，我們可以用&來表明副常式，而識別字的命名方式也是和其他變數的命名方式相同。也就是可以用數字，底線和字母組成，但是不能以數字開始。關於副常式，大概有兩個部份你必須特別注意的，也就是副常式的定義跟叫用。也就是副常式本身的區塊以及使用副常式。就像我們剛剛說到的，副常式的本身是以&符號作為辨識，所以如果你有一個叫做DoSub的副常式，那麼你就可以利用&DoSub的方式來叫用。當然，利用&符號叫用副常式本身，並不是絕對必要的，除非你的副常式名稱和Perl內建的函式名稱有所重疊，否則你其實可以省略&，也就是說，你可以直接使用DoSub來叫用你自己寫的DoSub副常式。當然，有時候我們會建議你在叫用副常式時盡量加上&符號，除非你能夠非常確定你使用的副常式名稱和Perl並不重複。但是當你看到許多程式設計師都省略&符號時，可別以為他們寫錯了，他們也許都是經驗老練的高手，已經能輕易確定自己所使用的函式名稱不會發生衝突。如果你還是不太了解我們解釋的意思，那還是讓我們來看看這樣的寫法吧：



```
my $num = 12;
print hex($num), "\n";    # 這是Perl 提供的hex函式
print &hex($num), "\n";    # 我們自己寫的hex副常式

sub hex {
    my $param = shift;
    $num*2;
}
```

這個例子應該就可以非常簡單的看出&帶來的不同，我們第一次使用了hex來呼叫函式，因為Perl內建了hex這個函式，所以Perl會直接使用內建的hex函式，而第二次我們使用&hex呼叫時，才真正叫用了我們自己定義的副常式hex。

不過說了這麼多，我們還是必須在開始叫用副常式之前，先嘗試寫出你自己的副常式。也就是主要運作的那一個部份，你可以使用sub這個Perl的關鍵字來定義一個副常式，而且既然我們已經使用了sub這個關鍵字，你總不會還以為我們會喜歡多打一個&字元吧，所以最典型的副常式大多會長的像這樣子：

```
sub subroutine {
    ...
    ...
}
```

當然，副常式內的縮排並非絕對必要的，不過為了保持程式的可讀性跟維持好的程式寫作習慣，我們還是極力建議各位在進行程式寫作時，能夠養成區塊內的縮排。在Perl中，一般的使用情況下(註一)，你可以把副常式放在程式中的任何位置，只要你叫用的時候，能夠讓程式本身不至於找不到副常式而發生錯誤就可以。雖然筆者自己習慣把副常式放在最後，不過對於已經有其他程式語言寫作習慣的人來說，也許有規定副常式的位置，而養成在程式的一開始就定義出程式中用了那些副常式的習慣。不過不管如何，Perl對於這些情況都是允許的，所以你可以試著找到自己習慣的方式。

另外，在Perl中使用副常式還有一個特點，也就是對於程式中全域變數的存取。由於副常式也是屬於程式的一部份(對Perl來說，那就是另一個程式中的區塊)，因此在Perl的設計中，你可以任意的存取程式中的全域變數，就像我們之前使用的那些變數。對很多使用其他程式語言的人來說，這實在非常不可想像，當然，也因此持反對意見的人應該也不在少數。不過保留這樣的功能卻未必就是鼓勵使用者以這樣的方式來寫程式，而只是保留某種彈性的空間。筆者還是建議各位能盡量使用參數的方式，並且使用副常式中的私有變數，這樣的建議當然是有一些理由的，因為你很可能在程式的發展過程中寫了一些副常式，並且把他們放在程式之中，等到程式慢慢成熟之後，你也許就可以把這些副常式放進模組裡，以方便建立程式的可重用性(註二)，以及屬於自己的函式庫。這時候，如果你的副常式足夠獨立的話，那麼搬移的工作就可以輕鬆許多，也不容易產生一些難以除錯的狀況。相反的，如果你在開始使用副常式的时候就大量使用全域變數時，你可能會發現要把這些副常式放入模組中就顯得特別困難。不過有時候能夠在副常式中使用全域變數也是非常方便的，例如有些模組中，你可能會有一些不提供給外部使用的副常式，這時候你也許會直接叫用程式的全域變數。現在，我們來寫我們的第一個副常式：

```
sub hello {
    print "hello\n";
}
```

沒錯，這個副常式雖然簡單，但是卻能夠讓我們一窺副常式的奧秘，所以當你叫用這個副常式時，他只會印出"hello"這個字串。那我們就來試試：

```
&hello;    # 印出 hello
```



```
&hello;          # 再印一次

sub hello {
    print "hello\n";
}
```

你應該發現了，副常式就是這麼簡單。

## 6.2 參數

沒錯，副常式的用法其實並不太困難，不過要能發揮副常式更重要的功能，可就還要下些功夫了，也就是讓副常式能根據我們的需求進行不同的回應。所以我們應該想辦法讓副常式能根據我們的需求來進行一些調整，進行不同的運算。首先，我們需要的是參數，所謂的參數也就是需求不同的那一個部份，利用參數來告訴副常式我們所需要的調整。使用參數，當然會有傳入跟接收的部份。發送端也就是叫用的部份，也就是我們要告訴副常式，我們需要進行調整的內容，我們只要直接把所要傳送的值放進小括號內，就像這樣：

```
&hello('world');
```

不過只有傳送當然是不夠的，我們的副常式也需要知道外面的世界發生了什麼事，它需要接收一些資訊。那我們來看看傳送的資訊去哪裡了，我們先來做個實驗：

```
&hello("world");          # 我們傳了參數 "world"

sub hello {
    print @_;              # 原來參數傳到這裡了
}
```

我們可以看到，當我們呼叫副常式，並且把參數傳給副常式時，參數會被放到預設的陣列變數@\_裡。這樣我們就可以叫用參數來進行操作了。既然如此，我們來改寫hello這個副常式吧！

```
&hello("world");          # 傳參數"world"

sub hello {
    my $name = shift @_;   # 把參數從預設陣列拿出來
    print "hello $name\n"; # 根據參數不同印出不同的招呼
}
```

## 6.3 傳回值

大多數的時候，我們除了參數，還會希望副常式可以有回傳值，也就是讓副常式利用我們的參數運算之後，也能夠傳回運算結果給我們，比如我們想要寫一個找階乘(註四)的副常式，因此我們告訴副常式，我們希望找到某個數的階乘，而當然也期望從副常式得到運算的結果，也就是我們需要副常式的回傳值。最簡單的方式，就是在副常式中使用return這個指令來要求副常式回傳某個值。我們可以試著把階乘的副常式寫出來：

```
my $return = &times(4);          # 把回傳值放到變數$return
print $return;

sub times {
    my $max = shift;             # 把參數指定為變數$max
    my $total = 1;               # 如果不指定，預設會是0，那乘法會產生錯誤
    for (1..$max) {              # 從 1 到 $max
        $total *= $_;            # 進行階乘的動作
    }
}
```

```
    return $total          # 傳回總數
}
```

在這裡，又有一些簡便的使用方式來處理Perl的傳回值，因為Perl會把副常式中最後一個運算的值當成預設的回傳值，所以你可以省略在進行運算後還必須再進行一次return的動作。就像這樣的寫法：

```
my $return = &square(4);
print $return;

sub square {
    my $base = shift;
    $base**2;
}
```

這時候，我們看到副常式的最後一次運算是把參數進行了一次取平方的動作，而這個運算結果就會直接被Perl當為回傳值，所以你就不需要再另外進行回傳的動作。這樣確實可以簡化寫副常式時的手續，繼續維持了Perl的簡樸風格。當然，如果你還是不太熟悉這種回傳的方式，你還是可以加上return的敘述，不過當你在看其他Perl程式設計師的程式時，可別被這樣的寫法搞混了。

#### 6.4 再談參數

我們已經知道了在副常式中怎麼使用參數及回傳值，而且我們還看到了Perl在處理參數時所使用的預設陣列。聰明的讀者應該早就猜到，當我們使用超過一個的參數時，應該就是依照陣列的規則一個一個被填入預設的陣列中，因此我們也可以按照這樣的原則來取出使用。我們可以用剛剛的概念，很容易的理解多個參數時的運用：

```
my $return = &div(4, 2);      # 這時候有兩個參數
print $return;

sub div {
    $_[0]/$_[1];              # 只是進行除法
}
```

這樣的寫法其實非常粗糙，不過我們只是舉例來說明副常式的參數運用。這次我們直接取出預設陣列中的元素來進行預算，因為只有一個運算式，所以運算結果也自然的被當成回傳值了。這樣的運用方式非常的簡明，所以當你在寫副常式的時候，你便可以使用許多組的參數。不過如果我們在叫用副常式的時候傳了三個參數，就像：

&div(4, 2, 6);

那會產生什麼結果呢？其實回傳值就跟原來的一樣，因為Perl並不會去在意參數的個數問題。不過如果你的程式有需要，應該去確認參數的個數，避免參數個數無法應付需要，以確保程式能正常而順利的進行。既然Perl的參數是以陣列的方式儲存，而我們也知道，Perl的陣列並沒有大小的限制，也就是以系統的限制為準。那麼我們很容易的可以傳入多個參數，而且還可以正確的運算並且回傳運算的結果。就像這樣：

```
my $return = &adv(4, 2, 6, 4, 9); # 我們一次傳入五個參數
print $return;

sub adv {
    my $total;
    for (@_) {
        # 針對預設陣列進行運算
    }
}
```

```

$total += $_;          # 加總
}
$total / ($#_+1);      # 除以總數 (取平均)
}

```

這時候，不論你的參數個數多少，Perl都可以輕鬆的應付，然後算出所有參數的平均值而且這時候。而且我們所需要的就是不管使用者有多少參數，都可以正確的算出他們的平均值。不過使用不定個數的時機或許不像固定參數個數來得頻繁，很多時候，我們都會使用固定的參數個數，然後確定每個參數的用途。當然這樣的用法有時候會讓人產生一些困擾，尤其是在你的程式會被大量重用時(註五)，不過要考慮這個問題還需要對Perl有更深入的了解，所以暫時我們就先不討論這種深入的用法。

## 6.5 副常式中的變數使用

就像大部分人所想的，副常式也是一個區塊，所以有屬於這個區塊自己的變數，也就是副常式的私有變數。不過就如我們所說的，副常式是可以使用程式中的全域變數，就像程式中的其他區塊一般。因此我們只需要在副常式中宣告my變數，也就是定義了副常式的私有變數。那麼就像我們知道的，變數將會維持到這個區塊的結束，也就是你無法在程式的其他地方存取這個變數。

另外，在副常式中，還有一種相當特殊的變數，也就是利用local來定義變數。不過這個部份目前用的人已經非常的少，所以你可以記著副常式裡面有這樣的用法，然後跳過一個部份。而我們打算在這裡提出來的原因是因為各位也許會有機會在某些程式裡面看到這樣的用法，為了避免大家看到這種用法卻又不知道它的作用，我們就在這裡簡單的介紹local的用法，讓大家未來有機會看到時可以能有一些參考的資料。

其實local的用途在於確認某些變數是在副常式中私用的，可是因為副常式會有機會被其他程式引用，所以你無法預期在某個引用的程式之中是否也有名稱相同的變數。因此使用local來確立這是副常式中的私有變數，而如果原來的程式中有相同的變數名稱時，就把主程式的變數放入堆疊，也就是先暫時儲存了主程式的這個變數，然後把相同的變數名稱清空以提供副常式使用。一但離開了副常式之後，Perl就會復原原來被儲存，並且清空的變數了。這樣子看起來，local和my的用法看起來似乎非常接近。

當然，你會發現這跟my之間會有什麼差異呢？我們先來看看這個程式：

```

$var1 = "global";
&sub1;          # 印出 sub1
print "$var1\n"; # 印出 global
&sub2;          # 現在變成 sub2
print "$var1\n"; # 又回到 global

sub sub1 {
    my $var1 = "sub1";
    print "$var1\n";
}

sub sub2 {
    local $var1 = "sub2";
    print "$var1\n";
}

```

看起來沒什麼不同，好像兩者之間沒有太大的差別，可是如果我們改寫一下程式：

```

$var1 = "global";
$var2 = "for local";
&sub1;          # 印出 local, for local
&sub2;          # 印出 global, for local

sub sub1 {
    local $var1 = "local";
    my $var2 = "my";
}

```

```

    &sub2;
}

sub sub2 {
    print "var1=$var1\tvar2=$var2\n";
}

```

從這裡，我們好像可以發現一些不同。差別就在於當我們先呼叫sub1的時候，sub1會把原來的變數\$var1放進堆疊，清空後把新的值"local"放入。而在呼叫sub2的時候，因為還在sub1的區塊內，因此local還佔用著\$var1這個變數。所以印出"local"的值，可是使用my就有所不同。雖然我們在sub1使用了my來定義區域變數\$var2，可是my卻不會把佔用原來\$var2變數的空間。所以當我們呼叫sub2時，會使用sub2裡的\$var2變數。而在sub2裡面因為沒有定義\$var2，所以Perl直接叫用全域變數，也就印出了"for local"的字串。

習題：

1. 下面有一段程式，包含了一個陣列，以及一個副常式diff。其中diff這個副常式的功能在於算出陣列中最大與最小數值之間的差距。請試著將這個副常式補上。

```

#!/usr/bin/perl -w

use strict;

my @array = (23, 54, 12, 64, 23);
my $ret = diff(@array);
print "$ret\n";          # 印出 52 (64 - 12)
my @array2 = (42, 33, 71, 19, 52, 3);
my $ret2 = diff(@array2);
print "$ret2\n";        # 印出 68 (71 - 3)

```

2. 把第四章計算階乘的程式改寫為副常式型態，利用參數傳入所要求得的階乘數。

註一：如果你想要了也更複雜的副常式使用方式，可以參考perldoc perlsub。

註二：就像你弄出了小螺絲釘，你總不希望每次遇到一樣的需要就重作一次螺絲釘。

註三：筆著第一次學Perl的時候就是被預設變數\$\_打敗的。(XXX 正文中沒有出現)

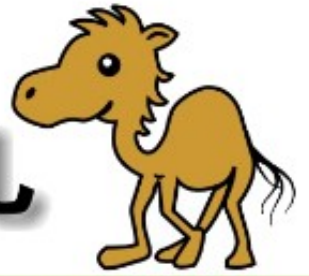
註四：階乘就是從一乘到某個數，比如4的階乘就是1x2x3x4。

註五：也就是你的副常式被放入模組中，而會不斷被重用時。那麼你固定的參數的個數及順序，一但將來副常式要改寫時，很容易影響過去使用的程式碼，而產生無法正確執行的問題。不過這屬於進階的問題，我們並不在這裡討論。





# Perl 學習手札



There is more than one way to do it.

## 7. 正規表示式

正規表示式其實並不是Perl的專利，相反的，在很多Unix系統中都一直有不少人使用正規表示式在處理他們日常生活的工作。尤其在許多Unix系統中的log更是發揮正規表示式的最好歷練，系統把所有發生過的狀況都存在log檔之中，可是你應該怎麼找出你要的資訊，並且統計成有用的資料。當然，大部份的Unix管理員可以求助許多工具，不過很大多數的狀況下，這些工具也是利用正規表示式在進行，所以如果說一個足夠深入管理Unix的系統管理員都曾經直接，或是間接的使用過正規表示式，我想應該很少人會反對吧。不過這顯然也充分的表達出正規表示式的重要性。

### 7.1 Perl 的第二把利劍

沒錯，正規表示式並不是Perl所獨有，或由Perl首創。可是在Perl之中卻被充分發揮，還有人說如果Perl少掉了雜湊跟正規表示式，那可就甚麼都不是了。情況也許沒有這麼誇張，可是卻可以從這裡明顯感覺出來正規表示式在Perl世界中所佔有的地位。對於許多人而言，聽到Perl的時候總不免聽其他人介紹Perl的文字處理能力，而這當然也大多是拜正規表示式所賜。

### 7.2 甚麼是正規表示式

講了那麼多，那到底甚麼是正規表示式呢？簡單的說，就是樣式比對。大部份的人用過各種文字處理器，文書編輯器，應該或多或少都用過編輯器裡面的搜尋功能，或是比對的功能吧！我彷彿聽到有人回答：那是基本功能啊。是啊，而且那也是最基本的樣式表示。就像我要在一大堆的文字中找到某個字串，這確實是非常需要的功能。不過如果你寫過其他程式語言，那麼你不妨回想一下，這樣的需求你應該怎麼表達呢？或者假設你現在是公司的網路管理員，如果你拿到一個郵件伺服器的log檔案，你希望找到所有寄給某個同事的所有郵件寄送資料，而你現在手上也許正在使用C或Java，或其他程式語言，你要怎麼完成你的工作呢？這樣說好像太過抽象，也許我們應該來舉個文件中搜尋關鍵字例子。

例如我希望在perlfunc這份Perl文件中找sort這個字串，這樣的需求很簡單，大部份的時候你也都可以完成這樣的程式。可是我如果希望找到sort或者delete呢？好吧，雖然麻煩，不過多花點時間還是沒問題的。不過實際去找了之後，我發現找出來的結果真是非常的多。於是我看到某些找到的結果是這樣的：

```
sort SUBNAME LIST
sort BLOCK LIST
sort LIST
```

沒錯，這些正是我想要找的結果，可是如果一個一個找也實在太辛苦了。所以如果我可以把這些東西寫成一個樣式，讓程式去辨別這樣樣式，符合樣式條件的才傳回來，這樣一來，應該比較符合我們的期待了。而所謂符合的條件，也就是我們所希望的「樣式」，於是我們開始想像這個樣式會是甚麼樣子，在這個例子中，我們開始設計我們需要的樣式：以sort開始，中間可能有一些其他的字，可能沒有，最後接著一個LIST，於是符合這樣的樣式都是我們所要搜尋的結果。相反的，如果在文章中其他地方出現的sort，可是並沒有符合我們的樣式，那麼也不能算是成功的比對。

就這樣，當我們再度拿起其他程式語言時，好像忽然覺得很難下手，因為要完成這樣的工作，顯然是非常的艱辛。不過在Perl的正規表示式中，這才是剛開始。因為你也許會希望在浩瀚的網路中找到你想要的某些資料，你也許知道某個網站有你所需要的資訊，比如每天的股票收盤價格，而你希望程式每天自動收集這些資訊之後自動去分析股票的走勢。當然，也許你已經可以每天派出機器人去各大新聞網站收集最新的消息，可是你也許需要利用正規表示式去萃取對你有幫助的新聞內容。或者你根本就想模仿google，去進行新聞的比對，然後過濾掉相同的新聞，利用機器人完成一份足夠動人的報紙。當然，並不是用了正規表示式就可以輕易完成這些工作，不過相較於其他開發工具，Perl在這方面顯然佔有相當大的優勢。

### 7.3 樣式比對



在Perl中，你要進行比對前，應該先產生出一個你所需要的「樣式(pattern)」，也就是說，你必須告訴Perl：在尋找的目標裡，如果發現存在著我所指定的樣式，就回傳給我。也就是說，你必須告訴Perl，我需要的東西大概長的像這個樣子，如果你有任何發現，就回傳給我。

所以樣式的寫法與精準與否就會影響比對的結果，通常而言，如果你發現比對出來的結果跟你的想像有所差距，那麼你顯然應該從比對的樣式著手，看看樣式上到底出了甚麼差錯。因為當你把結果反過來跟原來所寫的樣式比對，就會發現這些回傳結果確實還是符合比對的樣式。當然，要寫出正確的樣式是必須很花精神的，或者應該說要非常小心的。

如果我們要以簡單的方式來描述樣式的模型，那麼我們可以說樣式其實是由一個個單一位元所組成出來的一個比對字串。例如最簡單的一個單字是一個樣式，就像你寫了一個"Perl"，他就是一個樣式。可是在樣式中也可能有一些特殊符號，他們雖然沒辦法用一般的字元來表示，可是使用了特殊符號之後，在Perl的比對中，他們還是逐字元的進行比對。很常見的就是我們在列印程式結果也會用到的"\t"或是"\n"等等。所以如果你寫了這樣的一個字串，他也算是一個比對的樣式：

```
"Perl\tPython\tPHP"
```

另外，你還可能會用到一些量詞，也就是用來表達數量。量詞的使用對於Perl的正規表示式中是佔有重要地位的，因為使用了比對量詞，你就可以讓你的比對樣式開始具有彈性。例如你想在你的比對字串內找到一個字，這個字可能是：

```
WOW
WOOW
WOOOW
```

不過你又不想要把每一個字都放到你的比對樣式中，所謂的每一個字就也許包含

了'wow', 'woow', 'woow'...，而且也許他們有可能會變成"woooooow"，甚至中間夾雜了更多的"o"，甚至在你寫程式的時候也都還無法預測中間會出現多少次的'o'，這時候就是你需要使用量詞的時候了。另外還有許多技巧跟參數，例如你希望進行忽略大小寫的比對，或是你希望這個樣式只出現在句首或句尾等等，而這種種的東西都是拿來描述比對的樣式，讓Perl能更精準的比對出你所需要的字串。而在Perl之中使用正規表示式其實有許多的技巧，我們接下來就是要來討論該怎麼學習這些技巧。

#### 7.4 Perl 怎麼比對

我們之前提過，Perl所使用的是逐字元比對，也就是說，Perl根據你的樣式去目標內容一個字元一個字元進行比對。例如你的目標內容是字串 "I am a perl monger"，而你的樣式是字串"monger"。那麼Perl會根據樣式中的第一個字元"m"去字串中比對，當他瀏覽過"I"，空白鍵，"a"之後，他遇到了句子中的第一個"m"字元。於是Perl拿出樣式字串中的第二個字元"o"，可是目標字串的下一個字元卻是另一個空白鍵，於是Perl退回到比對字串的第一個字元"m"繼續比對。

就這樣繼續前進，一直到Perl找到下一個"m"。於是又拿出比對樣式的第二個字元"o"，發現也符合目標字串的下一個字元。然後繼續往前進，等到Perl把整個比對字串都完成，並且在目標字串對應到相同的字串，整個比對的結果就傳回1，也就是進行了成功的比對。

也許我們可以用圖示的方式來表達Perl在正規表示式中的比對方式。

[圖]

#### 7.5 怎麼開始使用正規表示式

如果你對Perl進行比對的方式有點理解，那麼要怎麼開始寫自己的正規表示式呢？

首先，我們要先知道，Perl使用了一個比對的運算子(=~)，也就是利用這個運算子來讓Perl知道接下來是要進行比對。接下來，就要告訴Perl你所要使用的樣式，在Perl中，你可以用m//來括住你的樣式。而就像其他的括號表達，//也可以替換為其他成對出現的符號，例如你可以用m{}，m||，或是m!!來表達你的樣式。不過對於習慣使用傳統的m//作為樣式表達的程式設計師來說，Perl倒是允許他們可以省略"m"這個代表比對(match)的字元。所以下面的方式都可以用來進行正規表示式：

```
$string =~ m/$pattern/
```

```
$string =~ m{$patten}
$string =~ m|$patten|
$string =~ m!$patten!
$string =~ /$patten/
```

Perl在完成比對之後，會傳回成功與否的數值，所以你可以將正規表示式放到判斷式中，作為程式流程控制的決定因素。不過也僅止於此，也就是說，當比對成功時，正規表示式就會結束，而且傳回比對成功的結果。當然，如果Perl比對到字串結束還是沒有找到符合比對樣式的字串，那麼比對依然會結束，然後Perl會傳回比對失敗的結果。例如下面的例子就是一個利用正規表示式來控制程式的例子：

```
my $answer = "monger";
until ((my $patten = <STDIN>) =~ /$answer/) {
    # 持續進行，直到使用者輸入含有 monger 的字串
    print "wrong\n";          # 在這裡，表示比對失敗
};
```

我們首先定義了一個字串"monger"，並且把這個字串作為我們的比對樣式，其實我們也可以直接把這個樣式放到正規表示式裝，不過我們在這裡只是讓大家可以比叫清楚的分辨出樣式的內容。。接下來，我們從標準輸入裝置（一般就是鍵盤）讀取使用者輸入的字串，並且把讀進來的字串放到變數\$patten中，接下來再去判斷使用者是否輸入含有"monger"的字串，如果沒有，就一直持續等候輸入，然後繼續進行比對，一直到比對成功才結束這個程式。

當然，如果正規表示式只能作這麼簡單的比對，那就真的太無趣了。而且如果他的功能這麼陽春，也實在稱不上是Perl的強力工具。還記得我們提過的量詞嗎？他可以讓我們的比對樣式變得更有彈性，現在我們可以用最簡單的量詞來重新描述我們的樣式。我們繼續使用剛剛的例子來看看：

```
my $answer = "mo*r";          # 使用量詞
while (1) {                   # 所以其實是無限迴圈
    if ((my $patten = <STDIN>) =~ /$answer/) { # 判斷是否比對成功
        print "*match*\n";
    } else {
        print "*not match*\n";
    }
};
```

我們試著來執行看看

```
[hcchi en@Apple]% perl ch3.pl
mor
*match*
mooor
*match*
moor
*match*
mar
*not match*
mur
*not match*
muur
*not match*
```

在這裡，我們用了這一次的量詞來進行比對。也就是"\*"這個比對的量詞，它代表零次以上的任何次數，在這裡因為他接在字母"o"的後面，也就表示了"o"這個字元出現零次以上次數都符合我們所想要的樣式。所以我

們看到前面幾次的比對都是比對成功即使我們只有輸入"mr"這個字串，但是因為這個字串中，"m"跟"r"之間，"o"總共出現了零次，因此對Perl而言，這也算是比對成功的。不過至少我們可以開始更有彈性的使用比對的樣式了，可是該怎麼要求Perl能夠最少比對一個"o"呢？在正規表示式中，'+'就表示至少出現一次，所以這時候我們就可以把"\*"換成"+"符號。也就是說，我們如果以剛剛的例子來看，當我們把比對樣式改成"mo+r"，原來可以成功比對的"mr"就不再成立了。

既然可以要求某個字元出現0次或1次，那麼如果我希望"o"至少出現二次，或其他更多的次數，有沒有辦法可以做到呢？答案也是肯定的，我們可以使用另一種方式來表示所需要的量詞數目，也就是說可以讓你限定次數的量詞，而它的表示方式會像這個樣子：

```
{min, max}
```

讓我們還是繼續以剛剛的例子來看，如果你希望掌握"o"出現的次數在某個區間內，那你就可以用這樣的方式。讓我們來改寫一下剛剛的程式變成這樣：

```
my $answer = "mo{2,4}r"; # 新的比對樣式
while (1) {
    if ((my $pattern = <STDIN>) =~ /$answer/) {
        print "*match*\n";
    } else {
        print "*not match*\n";
    }
};
```

我們試著執行看看：

```
[hcchi en@Apple]% perl ch3.pl
mor
*not match*
mooor
*match*
mr
*not match*
moor
*match*
```

很顯然的，比對樣式和剛剛有了明顯的變化。我們利用o{2,4}來限制了"o"只能出現兩次至四次，所以只要"o"出現的次數少於兩次或大於四次，我們都無法接受。而從執行的結果來看，Perl也符合我們的期待，因為當我們輸入"mor"或"mooor"時，Perl都傳回比對失敗的訊息。不過如果"m"跟"r"中間能夠比對到二到四次的"o"，那也就成功的比對了我們的樣式。

我們當然可能只需要設定某一邊的限制，例如我也許只要求某個字元出現三次以上，至於最多可能出現多少次我並不在意。這時候我們可以用這樣的樣式：mo{3,}r。很顯然，我們也可以這麼寫：mo{,8}r，這也就是表示我們並不限制"o"出現的最少次數，即使沒出現也可以，可是最多卻不能出現超過八次。

另外，我們剛剛都一直在討論某個位元使用量詞的比對，可是我們還希望能同時對某個字串使用量詞進行比對。就像這樣的字串"wowwow"，他也可能是"wow"或是"wowwowwow"。那麼我們應該怎麼來使用量詞呢？這時候，我們就需要定義某個群組了，而在正規表示式中，我們可以利用小括號()來把我們想要進行一次比對的字串全部拉進來，成為一個群組。所以如果我們希望比對出現一次以上的"wow"字串，那麼我們應該這麼寫：

```
my $answer = "(wow)+"; # 新的比對樣式
while (1) {
    if ((my $patten = <STDIN>) =~ /$answer/) {
        print "*match*\n";
    } else {
        print "*not match*\n";
    }
};
```

沒錯，當我們定義了群組(wow)之後，接下來Perl的比對每次都會以(wow)這個字串為主，也就是必須這個字串同時出現才算是比對成功。當然，你還是可以利用群組比對作限定量詞的方式，只要把剛剛的比對樣式改成(wow){2,4}，那麼跟比對單一字元是一樣的方式。Perl還是會比對"wow"這個字串是不是出現二到四次之間，就像我們比對單一字元的狀況一樣。

我們好像講了不少關於Perl正規表示式的技巧，不過這只是一小部份，其實關於正規表示式中還有許多技巧可以善加利用的。不過我們把這些留在下一章再來討論，這時候也許是該喝杯茶休息一下了。

習題：

1. 讓使用者輸入字串，並且比對是否有Perl字樣，然後印出比對結果。
2. 比對當使用者輸入的字串包含foo兩次以上時(foofoo 或是 foofoofoo 或是 ...)，印出比對成功字樣。





# Perl 學習手札



There is more than one way to do it.

## 8. 更多關於正規表示式

正規表示式確實能夠完成很多字串比對的工作，可是當然也需要花更多的時間去熟悉這個高深的學問。如果你從來沒有用過正規表示式，你可以在學Perl時學會用Perl，然後在很多其他Unix環境下的應用程式裡面使用。當然，如果你曾經用過正規表示式，那麼可以在這裡看到一些更有趣的用法。我們在上一章已經介紹了正規表示式的一些基本概念，千萬別忘記，那些只是正規表示式最基本的部份，因為Perl能夠妥善的處理字串幾乎就是仰賴正規表示式的強大功能。所以我們要來介紹更多關於正規表示式的用法。

### 8.1 只取一瓢飲

當你真正使用了正規表示式去進行字串比對的時候，你會發現，有時候會有可選擇性的比對。比如我希望找「電腦」或「資訊」這兩個詞是否在一篇文章裡，也就是只要「電腦」或「資訊」中任何一個詞出現在文章裡都算是比對成功，那麼我們就應該使用管線符號`|`來表示。所以我們的樣式應該試寫成這樣：`/電腦|資訊/`。

還有可能，你會想要找某個字串中部份相等的比對，就像這樣：

```
/f(oo|ee)t/           # 找 foot 或 feet
/it (is|was) a good choice/ # 在句子中用不同的字
/on (March|April|May)/ # 顯然也可以多個選擇
```

### 8.2 比對的字符集合

在Perl中的所有的命名規則都必須以字母或底線作為第一個字元，那麼我們如果要以正規表示式來描述這樣的規則應該怎麼作呢？你總不希望你的樣式表達寫成這個樣子吧？

```
(/a|b|c|d|.....|z|A|B|C|D|.....|_|)
```

這樣的寫法也確實是太過壯觀了一些。那麼我們應該怎麼減少自己跟其他可能看到這支程式的程式設計師在維護時的負擔呢？Perl提供了一種不錯的方式，也就是以「集合」的方式來表達上面的那個概念。因此剛剛的寫法以集合的方式來表達就可以寫成這樣：

```
[a-zA-Z_]
```

很顯然的，有些時候我們希望比對的字元是屬於數字，那麼就可以用`[0-9]`的方式。如果有需要，你也可以這麼寫`[13579]`來表示希望比對的是小於10的奇數。

有時候你會遇到一個問題，你希望比對的字元也許是各種標點，也就是你在鍵盤上看到，躲在數字上緣的那一堆字符，所以你想要寫成這樣的集合：

```
[!@#$%^&*()_+]=]
```

可是這時候問題就出現了，我們剛剛使用了連字號`-`來取得`a-z`，`A-Z`的各個字符，可是這裡有一個`[+=]`會變成甚麼樣子呢？這恐怕會產生出讓人意想不到的結果。所以我們為了避免這種狀況，必須跳脫這個特殊字元，所以如果你真的希望把連字號放進你的字符集合的話，就必須使用`(\)`的方式，所以剛剛的字符集合應該要寫成：

```
[!@#$%^&*()_+\-=]
```

另外，在字符集合還有一個特殊字元`^`，這被稱為排除字元。不過他的效用只在集合的開始，例如像是這樣：

```
[^24680]
```

這就表示比對24680以外的字元才算符合。

### 8.3 正規表示式的特別字元

就像我們在介紹正規表示式的概念的時候所說的，Perl是逐字元在處理樣式比對的。可是對於某一些字元，



我們卻很難使用一般鍵盤上的按鍵去表達這些字元。所以我們就需要一些特殊字元的符號。這些就是Perl在處理正規表示式時常用的一些特殊字元：

\s：很多時候，我們回看到要比對的字串中有一些空白，可是很難分辨他們到底是空格，跳格符號或甚至是換行符號（註一），這時候我們可以用\s來對這些字元進行比對。而且\s對於空白符號的比對掌握非常的高，以處理(\n\t\f\r)這五種字元。除了原來的空白鍵，以及我們所提過的跳格字元(\t)，換行字元(\n)外，\s還會驚??藉以表示回行首的\r跟換頁字元\f。

\S：在大部份的時候，正規表示式特殊字元的大小寫總是表示相反的意思，例如我們使用\s來表示上面所說的五種空白字元，那麼\S也就是排除以上五種字元。

\w：這個特殊字元就等同於[a-zA-Z]的字符集合，例如你可以比對長度為3到10的英文單字，那就要寫成：\w{3,10}，同樣的，你就可以比對英文字母或英文單字了。\W：同樣的，如果你不希望看到任何在英文字母範圍裡的字符，不妨就用這個方式避開。

\d：這個特殊的字元就是字符集合[0-9]的縮寫。

\D：其實你也可以寫成[^0-9]，如果你不覺得麻煩的話。

這些縮寫符號也可以放在中括號括住的集合內，例如你可以寫成這樣：[\d\w\_]，這就表示字母，數字或底線都可以被接受。而且看起來顯然比起[a-zA-Z0-9\_]舒服多了。

另外，你也可以這麼寫[\d\D]，這表示數字或不是數字，所以就是所有字元，不過既然要全部字元，那就不如用"."來表示了。

#### 8.4 一些修飾字元

現在是不是越來越進入狀況了呢？我們已經可以使用一般的比對樣式來對需要的字串進行比較了。於是我們拿到了一篇文章，就像這樣：

```
I use perl and I like perl. I am a Perl Monger.
```

我們現在希望找出裡面關於Perl的字串，這樣該相當簡單，所以我們把這串文字定義為字串\$content。然後只要用這樣的樣式來比對：

```
$content =~ /perl/;
```

不過好像不太對勁，或許我們應該改寫成這樣：

```
$content =~ /Perl/;
```

可是萬一我們打算從檔案裡面取出一篇文章，然後去比對某個字串，這時候我們不知道自己會遇到的是Perl或perl。既然如此，我們可以用字符集合來表示，就像我們之前說過的樣子：

```
$content =~ /[pP]erl/;
```

可是我要怎麼確定不會寫成PERL呢？其實你可以考慮忽略大小寫的比對方式，所以你只要這樣表示：

```
$content =~ /perl/i;
```

其中的修飾字元i就是告訴Perl，你希望這次的比對可以忽略大小寫，也就是不管大小寫都算是比對成功。所以你可能比對到Perl，perl，PERL。當然也可能有pErL這種奇怪的字串，不過有時候你會相信沒人會寫出這樣的東西在自己的文章裡。

Perl在進行比對的修飾字元，除了/i之外，我們還有/s可用。我們剛剛稍微提到了可以使用萬用字元點號(.)來進行比對，可是使用萬用字元卻有一個問題，也就是如果我們拿到的字串不在同一行內，萬用字元是沒辦法自動幫我們跨行比對，就像這樣：

```
my $content = "I like perl. \n I am a perl monger. \n";
if ($content =~ /like.*monger/) {
    print "$1*\n";
}
```

我們想要找到like到monger中間的所有字元，可是因為中間多了換行符號(\n)，所以Perl並不會找到我們真正需要的東西。這時候我們就可以動用/s來要求Perl進行跨行的比對。因此我們只要改寫原來的樣式為：

```
$content =~ /like.*monger/s
```

那麼就可以成功的進行比對了。可是如果有人還是喜歡用Perl Monger或是PERL MONGER來表達呢？我們當然還是可以同時利用忽略大小寫的修飾字元，因此我們再度重寫整個比對樣式：

```
$content =~ /like.*monger/is
```

這兩個修飾字元對於比對確實非常有用。

### 8.5 取得比對的結果

雖然樣式比對的成功與否對我們非常有用，可是很多時候我們並無法滿足於這樣的用法。尤其當我們使用了一些量詞，或修飾字元之後，我們還會希望知道自己到底得到了甚麼樣的字串。就以剛剛的例子來看，我的比對樣式是表示從like開始，到monger結束，中間可以有隨便任何字元。可是我要怎麼知道我到底拿到了甚麼呢？這時候我就需要取得比對的結果了。

Perl有預設變數來讓你取得比對的結果，就是以錢號跟數字的結合來表示，就像這樣：(\$1, \$2, \$3....)。而用法也相當簡單，你只要把需要放入預設變數的比對結果以小括號刮起括就可以了，就以我們剛剛的例子來看，你只要改寫比對樣式，就像這樣：

```
my $content = "I like perl. \n I am a perl monger. \n";
if ($content =~ /(like.*monger)/s) {
    print "$1\n";
}
```

這裡的\$1就是表示第一個括號括住的比對結果。所以Perl會送出這樣的結果：

```
[hcchien@Apple]% perl ch3.pl
like perl.
I am a perl monger
```

當然，預設的比對變數也是可以一次擷取多個比對結果，就像下面的例子：

```
my $content = "I like perl. \n I am a perl monger. \n";
if ($content =~ /(perl)\s(monger)/s) {
    print "$1\n";
} # $1 = "perl", $2 = "monger" # 印出 perl
```

不過我們如果再把這個小程序改寫成這樣呢？

```
my $content = "I like perl. \n I am a perl monger. \n";
if ($content =~ /((perl)\s(monger))/s) {
    print "$1\n$2\n$3\n";
}
```

```
}
```

結果非常有趣：

```
[hcchi en@Appl e]% perl ch3.pl
perl monger
perl
monger
```

看出來了嗎？我們用括號拿到三個比對變數，而Perl分配變數的方式則是根據左括號的位置來進行。因此最左邊的括號是整個比對結果，也就是"perl monger"，接下來是"perl"，最後才是"monger"。相當有趣，也相當實用。

不過在使用這些暫存變數有一些必須注意的部份，那就是這些變數的生命週期。因為這些變數回被放在記憶體中，直到下次比對成功，要注意，是比對成功。所以如果你的程式是這麼寫的話：

```
my $content = "Taipei Perl Monger";
$content =~ /(Monger$)/;      # $1 現在是 Monger
print $1;
$content = /(perl)/;          # 比對失敗
print $1;                     # 所以還是印出 Monger
```

當你第一次成功比對之後，Perl會把你所需要的結果放如暫存變數\$1中，所以你第一次列印\$1時就會看到Perl印出Monger，於是我們繼續進行下一次的比對，這次我們希望比對perl這個字串，並且把比對要的字串同樣的放入\$1之中。可惜我們的字串中，並沒有perl這個字串，而且我們也沒有加上修飾符號去進行忽略大小寫的比對，因此這次的比對是失敗的，可是Perl並不會先清空暫存變數\$1，因此變數的內容還是我們之前所比對成功的結果，也就是Monger，這從最後印出來的時候就可以看出來了。

比較容易的解決方式就是利用判斷式去根據比對的成功與否決定是否列印，就像這樣：

```
my $content = "Taipei Perl Monger";
print $1 if ($content =~ /(Monger$)/);      # 因為比對成功，所以會印出Monger
print $1 if ($content = /(perl)/);          # 這裡就不會印出任何結果了
```

## 8.6 定位點

要能夠精確的描述正規表示式，還有一項非常重要的工具，就是定位點。其中你可以指定某個樣式必須要被放在句首或是句尾，比如你希望比對某個字串一開始就是"Perl"這個字串。那麼你可以把你的樣式這樣表示：

```
/^Perl/
```

其中的^就是表示字串開始的位置，也就是只有在開始的位置比對到這個字串才算成功。當然，你可以使用\$來表示字串結束的位置。以這個例子來看：

```
my $content = "Taipei Perl Monger";
if ($content =~ /Monger$/s) {                # 以定位字元進行比對
    print "*Match*";                          # 在這裡可以成功比對
}
```

## 8.7 比對與替換

就像很多編輯器的功能，我們不只希望可以找到某個字串，還希望可以進行替換的功能。當然正規表示式也有提供類似的功能，甚至更為強大。不過其實整個基礎還是基於比對的原則。也就是必須先比對成功之後才能開始進行替換，所以只要你能了解整個Perl正規表示式的比對原理，接下來要置換就顯得容易多了。現在我們先來看一下在Perl的正規表示式中該怎麼描述正規表示式中的替換。

我們可以使用s///來表示替換，其中第一個部份表示比對的字串，第二個部份則是要進行替換的部份。還是舉個例子來看會清楚一些：

```
my $content = "I love Java";  
print $content if ($content =~ s/Java/Perl/); # 假如置換成功，則印出替換過的字串
```

當然，就像我們所說的，置換工作的先決條件是必須完成比對的動作之後才能進行，因此如果我們把剛剛的程式改寫成

```
my $content = "I love Java";  
print $content if ($content =~ s/java/perl/);
```

那就甚麼事情也不會發生了。當你重新檢查字串\$content時，就會發現正如我們所預料的，Perl並沒有對字串進行任何更動。

不過有時候我們會有一些問題，就像這個例子：

```
my $content = "水果對我們很有幫助，所以應該多吃水果";  
print $content if ($content =~ s/水果/零食/); # 把水果用零食置換
```

看起來好像很容易，我們把零食取代水果，可是當結果出來時，我們發現了一個問題。Perl的輸出是：「零食對我們很有幫助，所以應該多吃水果」。當然，這跟我們的期待是不同的，因為我們實在想吃零食啊。可是Perl只說了零食對我們有幫助，我們還是得吃水果。

沒錯，我們注意到了，Perl只替換了一次，因為當第一次比對成功之後，Perl就接收到比對成功的訊息，於是就把字串依照我們的想法置換過，接著....收工。好吧，那我們要怎麼讓Perl把整個字串的所有的「水果」都換成「零食」呢？我們可以加上/g這個修飾字元，這是表示全部置換的意思。所以現在應該會是這個樣子：

```
my $content = "水果對我們很有幫助，所以應該多吃水果";  
print $content if ($content =~ s/水果/零食/g); # 把水果全部換成零食吧
```

就像我們在比對時用的修飾字元，我們在這裡也可以把那些修飾字元再拿出來使用。就像這樣：

```
my $content = "I love Perl. I am a perl monger";  
print $content if ($content =~ s/perl/Perl/gi);
```

我們希望不管大小寫，所有字串中的Perl一律改為Perl，所以就可以在樣式的最後面加上/gi兩個修飾字元。而且使用的方式和在進行比對時是相同的方式。

## 8.8 有趣的字串內交換

這是個有趣的運用，而且使用的機會也相當的多，那就是字串內的交換。這樣聽起來非常難以理解，舉個例子來看看。

我們有一個字串，就像這樣：

```
$string = "門是開著的，燈是關著的"
```

看起來真是平淡無奇的一個句子。可是如果我們希望讓門關起來，並且打開燈，我們應該怎麼作呢？根據我們剛剛學到的替換，這件事情好像很簡單，我們只要把門跟燈互相對調就好，可是應該怎麼作呢？如果我們這麼寫：

```
$string =~ s/門/燈/;
```

那整個字串就變成了「燈是開著的，燈是關著的」，那接下來我們要怎麼讓原來「燈」的位置變成「門」呢？所以這種作法似乎行不通，不過既然要交換這兩個字，我們是不是有容易的方法呢？利用暫存變數似乎是個可行的方法，就像這樣：

```
my $string = "門是開著的，燈是關著的";  
  
print $string if ($string =~ s/(門)(.)*(,)(燈)(.+)/$3$2$1$4/);
```

看起來好像有點複雜，不過卻非常單純，我們只要注意正規表示式裡面的內容就可以了。在樣式表示裡面，非常簡單，我們要找門，然後接著是「門」和「燈」中間的那一串文字，緊接在後面的就是「燈」，最後的就全部歸在一起。按照這樣分好之後，我們希望如果Perl比對成功，就把每一個部份放在一個暫存變數中。接下來就是進行替換的動作，我們把代表「門」跟「燈」的暫存變數\$1及\$3進行交換，其餘的部份則維持不變。我們可以看到執行之後的結果就像我們所期待的一樣。

當然，這樣只是最簡單的交換，如果沒有正規表示式，那真的會非常的複雜，不過現在我們還可以作更複雜的交換動作。

## 8.9 不貪多比對

其實在很多狀況下，我們常常不能預期會比對甚麼樣的內容，就像我們常常會從網路上抓一些資料回來進行比對，這時候我們也許有一些關鍵的比對樣式，但是大多數的內容卻是未知的。因此比對的萬用字元(.)會經常被使用，可是一旦使用了萬用字元，就要小心Perl會一路比對下去，一直到不合乎要求為止，就像這樣：

```
<table>  
  <tr><td>first</td></tr>  
  <tr><td>second</td></tr>  
  <tr><td>third</td></tr>  
</table>
```

這是非常常見的HTML語法，假設我們希望找到其中的三個元素，所以就必須過濾掉那些HTML標籤。如果你沒注意，也許會寫成：

```
my $string = "<table><tr><td>first</td></tr><tr><td>second</td></tr><tr><td>  
third </td></tr></table>";  
  
if ($string =~ m|<tr><td>(.+)</td></tr>|) {  
  print "$1";  
}
```



可是當你看到執行結果時可能會發現那並不是你要的結果，因為程式印出的\$1居然是：

```
first</td></tr><tr><td>second</td></tr><tr><td>third
```

讓我們來檢查一下程式出了甚麼問題。我們的比對樣式中告訴Perl，從<tr><td>開始比對，然後比對所有字元，一直到遇到</td></tr>時結束。而且Perl也很符合我們的期望，他找到了符合我們需求的最大集合。這就是重點了，Perl預設會去找到符合需求的最大集合。因此在這裡他就取得了比對結果"first</td></tr><tr><td>second</td></tr><tr><td>third"。可是我們要的卻是「從<tr><td>開始，遇到</td></tr>就結束」，而不是「找出字串中<tr><td>到</td></tr>的最大字串」。

可是我們剛剛的比對樣式中並沒有告訴Perl：「遇到</td></tr>就停下來」，所以他會一直比對到字串結束，然後找出符合樣式的最大字串，這就是所謂的貪多比對。相對於此，我們就應該告訴Perl，請他以不貪多的方式進行比對。所以我們就在比對的量詞後面加上問號(?)來表示不貪多，並且改寫剛剛的比對樣式：

```
$string =~ m|<table><tr><td>(.*?)</td></tr>|
```

如此一來，就符合我們的要求了。

### 8.10 如果你有疊字

在正規表示式中，有一種比對的技巧稱為回溯參照 (backreference)。我們如果可以用個好玩的例子來玩玩回溯參照也是不錯的，比如我們有個常見的句子：「庭院深深深幾許」。如果我希望比對中間三個深，我可以怎麼作呢？當然，直接把「深深深」當作比對的樣式是個方法，不過顯而易見的，這絕對不是個好方法。至少你總不希望看到有人把程式寫成這樣吧：

```
my $string = "庭院深深深幾許";
print $string if ($string =~ /深深深/);    # 這樣寫程式好像真的很糟
```

這時候回溯參照就是一個很好玩的東西，我們先把剛剛的程式改成這樣試試：

```
my $string = "庭院深深深幾許";
print $string if ($string =~ /(深)\1\1/);
```

你應該發現了，我們把「深」這個字先放到暫存變數中，然後告訴Perl：如果有東西長的跟我們要比對的那個變數裡的東西一樣的話，那麼就用來繼續比對吧。可是這時候你卻不能使用暫存變數\$1，因為暫存變數是在比對完成之後才會被指定的，而回溯參照則是在比對的期間發生的狀況。剛剛那個例子雖然可以看出回溯參照的用途，可是要了解他的有用之處，我們似乎該來看看其他的例子：

```
my $string = "/Chinese/中文/";
if ($string =~ m|([\w\|'"])(.*?)\1(.*?)\1|) {    # 這時候我們有一堆字符集合
    print "我們希望用 $3 來替換 $2 \n";
}
```

看出有趣的地方了嗎？我們在字符集合裡面用了一堆符號，因為不論在字符集合裡的那一個符號都可以算是正確比對。但是我們在後面卻不能照舊的使用[\w\|'"']來進行比對，為甚麼呢？你不妨實驗一下這個例子：

```
my $string = "/Chinese|中文' ";
if ($string =~ m|([\w\|'"])(.*?)[\w\|'"](.*?)[\w\|'"]|) {
```

```
print "我們希望用 $3 來替換 $2 \n";  
}
```

很幸運的，我們在這裡還是比對成功。為甚麼呢？我們來檢查一下這次的比對過程：首先我們有一個字符集合，其中包括了/|"四種字符，而這次我們的字串中一開始就出現了/字符，正好符合我們的需求。接下來我們要拿下其他所有的字元，一直到另一個相同的字符集合，不過我們這次拿到了卻是|字符，最後我們拿到了一個單引號(')。顯然不單是方便，因為沒有使用回溯參照的狀況下，我們拿到了錯誤的結果。那我們回過頭來檢查上一個例子就會清楚許多了，我們一開始還是一個字符集合，而且我們也比對到了/字符。接下來我們需要找到跟剛剛比對到相同的內容（也就是要找到下一個/），然後還要再找最後一次完全相同的比對內容。我們經常會遇到單引號(')或雙引號(")必須成對出現，而利用回溯參照就可以很容易的達成這樣的要求。

## 8.11 比對樣式群組

我們剛剛說了關於回溯參照的用法，不過如果我們的比對並沒有那麼複雜，是不是也有簡單的方式來進行呢？我們都知道很多人喜歡用blahblah來進行沒有什麼意義的留言，於是我們想把這些東西刪除，可是他們可能是寫"blahblah"或是"blahblahblah"等等。這時候使用回溯參照可能會寫成這樣：

```
my $string = "blahbl ahblah means nothing";  
  
if ($string =~ s/(blah)\1*//) {  
    print "$string";  
}
```

當然這樣的寫法並沒有錯，只是好像看起來比較礙眼罷了，因為我們其實可以用更簡單的方法來表達我們想要的東西，那就是比對樣式群組。這是小括號()的另外一個用途，所以我們只要把剛剛的比對樣式改成這樣：/(blah)+/就可以了。這樣一來，Perl就會每次比對(blah)這個群組，然後找尋合乎要求的群組，而不是單一字元(除非你想把某一個字元當群組，只是我們並不覺得這樣的方式會有特殊的需求)。而當我們設定好某個群組之後，他的操作方式就跟平常在寫比對樣式沒什麼兩樣了，我們就可以利用/(blah)+/找出(blah)這個群組出現超過一次的字串。如果你覺得不過癮，/(blah){4,6}/來確定只有blah出現四到六次才算比對成功也是可以的。

## 8.12 比對樣式的控制

一開始使用正規表示式的人總有一個疑問，為甚麼要寫出正確比對的樣式這麼不容易。而比對錯誤的主要原因通常在於得到不必要的資料，也就是比對樣式符合了過多的文字，當然，還有可能是比對了不被我們期待的文字。就像我們有這樣的一堆字串：

```
I am a perl monger  
I am a perl killer  
it is so popular.
```

如果你的比對樣式是/p.\*r/，那麼你會比對成功：

```
perl monger  
perl killer  
popul ar
```

可是這跟我們的需求好像差距太大，於是你希望用這樣的樣式來進行比對：/p\w+\s\w+r/，那你也還會得到

```
perl monger  
perl killer
```

這兩種結果。所以怎麼在所取得的資訊中寫出最能夠精確比對的樣式確實是非常重要的，也需要一些經驗的。

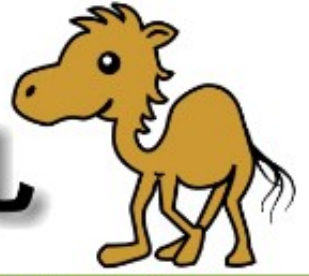
習題：

1. 延續第七章的第一題，比對出perl在字串結尾的成功結果。
2. 繼續比對使用者輸入的字串，並且確定是否有輸入數字。
3. 利用回溯參照，找出使用者輸入中，引號內(雙引號或單引號)的字串。
4. 找出使用者輸入的第一個由p開頭，l結尾的英文字。

註一：有時候因為作業系統的不同，換行符號並不會被忠實的呈現。



# Perl 學習手札



There is more than one way to do it.

## 9. 再談控制結構

程式中的控制結構是用來控制程式進行方向的重要依據，所以有足夠靈活的程式控制結構能節省程式設計師的大量時間。可是也必須謹慎的使用，否則如果一個程式裡面到處充滿了迴圈控制，然後中斷，轉向，反而讓程式變得非常沒有結構，在程式的結構遭到破壞之後，一旦程式出了問題而需要開始追蹤整個程式的行進就變成要花大量的時間。很顯然的，如果沒有保持程式良好的結構性，對於日後的維護將會是一大負擔。不過如果能夠小心使用，這些控制程式流程的工具會是程式設計師重要的工具。所以我們就來看看除了之前提過的for, while, until, if等等各式各樣的流程控制之外，我們還能有什麼其他的方式可以方便的操控Perl吧。

### 9.1 迴圈操作

既然是流程控制，我們有時候會因為程式的狀況而希望離開迴圈或某些條件敘述的判斷區塊，也可能需要省略迴圈中的某次運算，或進行其他的跳躍。聽起來這些方式好像很容易讓人搞的眼花繚亂，現在我們就要來看看這些功能到底有什麼幫助。

#### 9.1.1 last

顧名思義，這也就是最後的意思，因此Perl看到這個關鍵字就會相當高興，表示他距離休息又靠近了一步。不過這樣可以提前結束迴圈的函數到底扮演著甚麼樣的角色呢？讓我們來試試：

```
for (1..10) {  
    last if ($_ == 8);  
    print;                # 這樣會印出 1...7  
}
```

好玩吧，當你在迴圈內加上了另外一個判斷式，並且允許迴圈在某個條件下跳出迴圈的執行，這時候，Perl就提早下班，回家休息了。當然，last的使用是有限制的，也就是他只允許在可以執行結束的區塊內，其中最常被應用的是在迴圈內，而這裡的迴圈指的是for, foreach, while, until, 另外也可以在單獨的區塊中使用，就像這樣：

```
print "start\n";  
{  
    print "last 前執行\n";    # 會順利印出  
    last;  
    print "這裡就不執行了\n";    # 所以這一行永遠不會被執行  
}  
print "然後就結束了\n";
```

沒錯，這個例子的程式確實非常無趣，因為我們寫了一行永遠不會執行的程式，不過卻讓我們透過這個小程序清楚的看到last的執行過程。所以我們可以輕易的讓Perl跳出某個區塊，而既然可以提前結束區塊的執行範圍，我們有時候也需要Perl可以重複執行迴圈內的某些條件這時候redo這個函式就派上用場了。

#### 9.1.2 redo

雖然我們可以用last讓Perl提早下班，同樣的，我們也可以要求Perl加班，也就是利用redo這個函式讓Perl重

新執行迴圈中的某些條件。例如我們在迴圈中可以利用另一個判斷敘述來決定目前的狀況，並且利用這個額外的敘述判斷來決定是否要讓迴圈中的某個條件重複執行。例如我們有一個迴圈，我們可以很容易的要求perl在迴圈中的某個階段重複執行一次，就像下面的例子：

```
for (1..10) {  
    $_++;                                # redo 其實會來這裡  
    redo if ($_ == 8);                  # 我們希望 redo 的條件  
    print;                             # 會印出 2, 3, 4, 5, 6, 7, 9, 9, 10, 11  
}
```

我們可以研究一下redo的過程，在上面那個程式的第三行，我們要求Perl在迴圈的變數等於8的時候就執行redo。所以當我們在迴圈內的條件符合redo的要求時，Perl就會跳到迴圈的第一行，也就是說當迴圈的值進行到7的時候，經過\$\_++的運算就會讓\$\_變成8，這時候就符合了redo的條件，因此還來不及印出來變數\$\_，Perl就被要求回到迴圈中的第一行，於是\$\_變成了9，這就是第一次9的出現。接下來迴圈恢復正常，就接連印出9...11。也就是我們看到的結果了。

不過這裡要注意的是我們迴圈的使用方式，我們使用了for(1..10)，而不是使用for (\$\_ = 1; \$\_ <= 10); \$\_++)這樣的敘述，而這兩者有著相當的差異。如果各位使用了後者的迴圈表示，結果就會有所不同。

我們還是實際上來看看使用for (;;)來檢查redo的效果時，也可以藉此看看兩者的差異了：

```
for ($_ = 1; $_ <= 10; $_++) {  
    $_++;                                # 我們還是先把得到的元素進行累加  
    redo if ($_ == 8);                  # 遇到8的時候就重複一次  
    print $_;                          # 印出目前的 $_，我們得到2, 4, 6, 9, 11  
}
```

很有趣吧，我們來看看這兩者有什麼不同，首先我們看到第一個例子中，Perl是拿出串列1...10的元素，並且把得到的元素放進變數\$\_中。接下來就像我們在迴圈中所看到的樣子了，所以迴圈並不是以\$\_作為計數的依據。這樣的方式就像這樣的寫法：

```
for (my @array = (1..10))
```

可是當我們看到第二個例子的時候，我們卻是指定了\$\_作為迴圈的計數標準。所以我們在迴圈中對\$\_進行累加，就完全影響到迴圈的執行。因此我們一開始拿到\$\_等於1，可是一進迴圈就馬上又被累加了一次，我們就印出了2，接著Perl又執行迴圈的遞增，讓我們取得3，我們自己又累加了一次，也就印出4，等我們累加到8的時候，迴圈被要求執行redo，因此我們又累加一次，\$\_變成9，緊接著最後一次的迴圈，經過累加之後，我們印出了11。看起來好像非常複雜，不過你只要實際跟著迴圈跑一次應該就可以看出其中的變化了。不過在使用redo的時候必須非常小心，因為你很可能因為設定了redo的條件而產生無窮迴圈。就像剛剛的例子，如果我們改寫成：

```
for (1..10) {  
    redo if ($_ == 8);  
    print $_;  
}
```

在這個迴圈裡，我們希望迴圈的控制變數在8的時候可以進行redo，於是它就一直卡在8而跳不出來了。就像我們說的，這裡變成了無窮迴圈，你的程式也就有加不完的了。



### 9.1.3 next

我們剛剛使用了last來結束某個區塊，也透過redo來重複執行迴圈中的某個條件敘述，那麼既然可以在迴圈內重複執行某個條件的敘述，那麼略過某個條件下的敘述也應該不是太難的問題。是的，其實只要利用next，那麼我們可以在某些情況下直接結束這次的執行，也就是省略迴圈中某一些狀況的執行。當然，描述還是不如直接看看實例，我們還是利用簡單的例子來了解next的作用：

```
for (1..10) {  
    next if ($_%2);           # 以串列值除以2的餘數判斷  
    print $_;  
}
```

這個例子裡面，我們會印出1...10之間的所有偶數。首先，這是一個從1到10的迴圈，主要的工作在於印出目前迴圈進行到的值，不過就在列印之前，我們使用了一個next函式，而決定是否執行next的判斷是以串列值除以2的餘數來作為條件

。如果餘數為真(在這裡的解釋就是：如果餘數為1)，就直接結束這次的執行，當然，如果餘數為0(在這個程式中，我們可以解釋為「遇到偶數時」)，就會印出串列的值，所以程式會印從1到10的所有偶數值。

雖然有些時候，你會發現next的好用之處，可是如果你會因為next而造成追蹤程式的困擾時，那就可能要修改一下你的使用方式了。例如改變迴圈的判斷條件或是索引的遞增方式等等，就像上面的例子，我們也許可以改用while來判斷，或者使用for(;;)，而不是使用foreach加上next來增加程式的複雜性，不過這些都必須依賴經驗來達成。

### 9.1.4 標籤

標籤的作用主要就是讓Perl知道他該跳到哪裡去，這樣的寫法並不太常被使用，主要是因為對於程式的結構會有一定程度的破壞，因為你可以任意的設置一個標籤位置，然後要求Perl跳到標籤的位置，當然，他確實有一些使用上的要求，而不是完全漫不限制的隨便下一個標籤就讓Perl轉換執行的位置，至少這並不是goto在做的工作。不過撇開這個暫且不談，我們先來看看怎麼使用標籤。下面的例子應該可以讓大家都能夠看出輪廓：

```
LABEL: for my $outter (1..5) {  
    for (1..10) {  
        if ($_ > 2) { next LABEL; } else { print "inner $_ \n"; }  
    }  
    next LABEL if ($outter%2);  
    print $_;  
}
```

當我們有時候單單利用next或last無法逃離迴圈到正確的地方時時，使用標籤就能夠幫助我們找到出路。就像我們的例子中，我們一共有兩個for迴圈，兩個if判斷，我們要怎麼讓Perl不會在裡面迷路呢？這時候標籤的使用就很方便了，就像我們在內部的for迴圈中根據得到的值來決定是否要跳出上一層的for迴圈。

可是使用標籤時有一個特別需要注意的部份，就是標籤的使用並非針對程式中的某一個點，而必須是一個迴圈或是區塊。否則整個標籤的使用就會太過混亂，你會發現要檢查程式的錯誤變成了「不可能的任務」。當然，如果你在迴圈中插了大量的標籤也會讓其他人非常困擾，因為就算是Perl可以處理這樣的標籤，只怕你自己也會搞的頭暈。這又是寫程式時的風格問題了。

標籤可以配合我們之前所提的幾種控制指令來運用，因此你可以要求使用next，redo，last加上標籤來標明迴圈的方向。就像上面的例子，我們先在第一行的地方加上標籤'LABEL'，表明接下來如果需要，要求Perl直接來這裡。接下來我們用了一個foreach迴圈，其中的值是從1到10。可是在這個迴圈中，我們又使用了next，要求如果變數\$\_大於2就執行next，而且是跳到標籤LABEL的位置。也就是說，他除了跳過裡面的迴圈之外，也會跳出外層迴圈的其他敘述。所以當內層迴圈的\$\_變數大於2的時候，程式中最後面的兩行敘述都不會被執行。當然，大家應該還是想要知道這樣的程式會產生出甚麼樣的結果：

```
i n n e r 1
i n n e r 2
i n n e r 1
i n n e r 2
i n n e r 1
i n n e r 2
i n n e r 1
i n n e r 2
i n n e r 1
i n n e r 2
```

你應該發現了，程式一直都只執行了一部份，因為當內圈的變數\$\_大於2的時候，Perl就急著要回去LABEL的地方，所以就連裡面的迴圈都沒辦法完整執行，外面的迴圈更是被直接略過，這樣應該就很容易理解了。

## 9.2 switch

如果你用過其他程式語言，例如C或Java，你現在也許會很好奇，為甚麼我們到目前為止還沒有提到Switch這個重要的流程控制函式，主要是因為Perl在最初的設計是沒有放入Switch的。其實很多人對於Perl沒有提供switch都覺得非常不可思議，不過Larry Wall顯然有他的理由，至於這些歷史原因，我們也沒必要在這裡討論。

好吧，我聽到一陣嘩然，為甚麼Perl沒有這個可以為程式畫上彩妝的工具呢？其實我個人也覺得Switch用來進行各種條件判斷的流程控制確實是非常方便，而且會讓程式看起來相當整齊，不過大部份的時候，你有甚麼流程控制非得需要Switch才能完成呢？因為我們在進行Switch的時候，其實也就是希望表達出許多層的if {} elsif {} elsif {} .....。也就是說，if敘述其實已經可以滿足我們的需求了，那麼Switch就真的是幫助我們取得比較整齊，易讀的程式碼。不過在大部份的情況下，你想要用漂亮的程式碼來吸引Perl的黑客(註一)們協助完成一項工作，倒不如告訴他們怎麼樣可以少打一些字。

### 9.2.1 如果你有複雜的 if 敘述

Switch之所以受到歡迎，當然有過人之處，雖然我們也可以用其他方式達到同樣的目的，可是至少對我來說，程式的易讀性似乎還是以Switch來得好些，不過這部份可就是見仁見智了。就像我們說的，如果你有一大堆if {} elsif {} elsif {} .... 的敘述時，你的程式看起來也許看起來會像這樣：

```
my $day = <STDIN>;
chomp($day);
if ($day eq 'mon') {
    ...
} elsif ($day eq 'tue') {
    ...
} elsif ($day eq 'wed') {
    ...
} elsif ($day eq 'thu') {
    ...
} elsif ($day eq 'fri') {
    ...
}
```

其實這樣的程式碼也沒甚麼不妥，可是你也許會覺得這樣的寫法有點麻煩。當然，對這些人來說，如果可以把上面這段程式碼利用Switch寫成這樣，那好像看起來更讓人感覺神清氣爽：

```
my $day = <STDIN>;
chomp($day);
swi ch ($day) {
    case ('mon') { ... }
    case ('tue') { ... }
```

```

    case ('wed') { ... }
    case ('thu') { ... }
    case ('fri') { ... }
}

```

以可讀性來講，使用Switch確實比用了一大堆的 `if {...} elsif {...} elsif {...}` 要好的多，那麼我們要怎麼樣可以使用Switch來寫我們的程式呢？

### 9.2.2 利用模組來進行

很顯然的，還是有許多Perl的程式設計師對於switch的乾淨俐落難以忘懷。因此有人寫了perl模組，我們就可以利用這個模組來讓我們的程式認識switch。

利用Switch模組，我們就可以寫出像上面一樣的語法，讓你的程式看起來更簡潔有力。而且switch的使用上，不單可以比對某個數字或字串，你還可以使用正規表示式進行複雜的比對來決定程式的進行方向。我們在這裡只是告訴大家一些目前已經存在的解決方案，而不應該在這裡講太多關於模組的使用，以免造成大家的負擔。

另外，還有部份程式設計師不太喜歡目前Switch的運作，認為破壞了原來Perl在流程控制的結構而也會因此而破壞原來Perl程式的穩定性。因為不管如何，這些意見都是僅供參考。不過既然「辦法不只一種」，那麼就看個人的接受度如何了。

### 9.3 三元運算符

另外也有一種非常類似 `if {...} else {...}` 的運算符，我們稱為三元運算符。他的寫法也就是像這樣：

```

my ($a, $b) = (42, 22);
my $max = ($a > $b) ? $a : $b;
print "$max\n";

```

首先我們把串列 (42, 22) 指定給變數 \$a 跟 \$b，接著我們找到兩個值中較大的一個，於是利用判斷式 (`$a > $b`) 來檢查兩個數字之間的關係。如果 `$a > $b` 成立，那麼 \$max 就是 \$a，否則就是 \$b。所以很明顯的，上面的三元運算符也可以改寫成這樣：

```

my ($a, $b) = (44, 22);
if ($a > $b) { $max = $a } else { $max = $b }
print "$max\n";

```

以上面兩個例子來看，相較之下，三元運算符的方式應該簡單許多，只是這樣的方式並不夠直覺，對於剛開始寫Perl的人而言可能會有點障礙。不過我們還是必須提醒，這樣的寫法很可能常常出現在其他的程式裡，所以即使你只想依賴 `if {...} else {...}` 來完成同樣的工作，至少你也要知道別人的程式碼中表達的是甚麼。而且，其實利用三元運算符也可以完成不少複雜的工作。例如你可以在判斷式的地方用一個副常式，並且根據回傳的結果來決定你要的值等等。因此一但有機會，也許你也可以試試。在這裡，我們可以再舉一個怎麼增加便利性的寫法的例子：

```

my $return = cal(5);
print "$return\n";

sub cal {
    my $param = shift;
    ($param > 4) ? $param*2 : $param**2; # 利用參數來判斷回傳值的運算方式
}

```

#### 9.4 另一個小訣竅

接下來我們來點飯後甜點，也就是 `||` 算符。其實不只 `||` 算符，其他的邏輯算符也可以拿來作流程控制的小小螺絲釘。不過首先我自己偏愛使用 `||` (也是使用機會比較高的)，而且我們只打算來個甜點，這時候顯然不適合大餐了。

我們有時候會希望某些變數可以有預設值，例如副常式的參數，或是希望使用者輸入的變數等等。所以你當然可以這樣寫：

```
sub input {
    my $key = shift;
    $key = "預設值" unless ($key);
    print "$key\n";
}
```

這個副常式甚麼也沒作，就只拿了使用者傳來的參數，然後印出來。可是我們還可以讓他更簡單一些，我們把他改成這樣：

```
sub input {
    my $key = shift || "預設值";
    print "$key\n";
}
```

這時候，`||` 算符被我們拿來當一個判斷的工具。我們先確定使用者有沒有傳入參數，也就是平常我們所使用的 `shift`，如果 `$_` 中是空陣列，那麼 `$key = shift` 就會得到偽值，這時候 `||` 就會啟動，讓我們的預設值產生效果。因此我們就得到 `$key = "預設值"`。

另外，`||` 還常常被用來進行意外處理。因為我們必須知道，如果某個運算式失敗，那麼我們就可以讓程式傳回錯誤訊息。就像這樣：

```
output() || die "沒有回傳值";

sub output {
    return 0;
}
```

我們在程式裡面呼叫 `output` 這個副常式，不過因為回傳值是 0，於是 `||` 也發生效用，就讓程式中斷在這裡，並且印出錯誤訊息。

習題：

1. 陸續算出 (1...1) 的總和，(1...2) 的總和，...到 (1...10) 的總和。但是當得到總和大於50時就結束。
2. 把下面的程式轉為三元運算符形式：

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

use strict;

chomp(my $input = <STDIN>);
if ($input < 60) {
    print "不及格";
} else {
```

```
    print "及格";  
}
```

註一：其實我們指的就是hacker，不過現今大多數人都誤用cracker(指潛入或破壞其他人系統者)為hacker(指對某些領域有特別研究的人)





# Perl 學習手札



There is more than one way to do it.

## 10. Perl的檔案存取

檔案系統在寫程式時是非常重要的一个部份，尤其對於Perl的使用者來說，因為Perl能夠處理大量而且複雜的資料，所以經常被拿來作為Unix作業系統的管理工具，尤其對於Unix-like系統管理員而言，在進行系統日誌的管理時，存取檔案，讀取檔案內容並加以分析就是最基本的部份。當然，你還可能進行目錄的修改，檔案權限的維護等等跟系統有密切關係的操作。

### 10.1 檔案代號 (FileHandle)

當你的Perl想要透過作業系統進行檔案存取時，可以利用檔案代號取得和檔案間的繫結，接下來的操作就是透過這個檔案代號和實體的檔案間進行溝通。也就是說，我們要進行檔案操作時，可以先定義相對應實體檔案的代號，以便我們用更簡便的方式對檔案進行存取。

而所謂的檔案代號其實就是由使用者自行命名，並且用來跟實體檔案進行連結的名稱，他的命名規則還是依循Perl的命名規則，大家對於這個規則應該相當熟悉了，不過我們還是再次提醒一下：可以數字，字母及底線組成，但是不能以數字作為開始。而且一般來說，我們幾乎都習慣以全部大寫來作為檔案代號，因為檔案代號並不像其他變數，會使用某些符號作為識別，所以幾乎約定成俗的全部大寫習慣也是有存在的道理。當然，你也可以依照自己的習慣來為檔案代號命名(註一)，這表示所謂的全部大寫絕對不是一種鐵律，就像Perl程式語言本身，希望以最少的限制來進行程式設計的工作。

### 10.2 預設的檔案代號

對於檔案的輸出，輸入而言，其實就跟平常時候，你利用Perl在進行其他的操作非常接近，有時候只是輸出到不同的媒介上。所以Perl其實已經預定了幾種檔案代號，讓你不需要每次寫Perl的程式就必須去重新定義這些代號，很顯然的，幾乎大部份的程式都會需要這些檔案代號。

這六個預設的檔案代號分別是：STDIN，STDOUT，STDERR，DATA，ARGV，ARGOUT，看起來相當熟悉吧？沒錯，因為很多時候，我們其實就是靠這些預設的檔案代號在進行程式的輸出，輸入。只是我們還沒有了解這些其實就是檔案代號。換個角度來看，其實即使我們都不知道他們是預設的檔案代號，我們就能運作自如，那麼對於檔案代號的使用顯然就不是太難。不過，我們還是要再來看看這六個Perl預設的檔案代號。其中有些我們已經使用過了，我們就先對其中幾個預設的檔案代號來進行介紹：

STDIN：這也就是我們常看到的「標準輸入裝置」，當Perl開始執行時，它預設接受的外部資訊就是從這裡而來。就像我們之前曾經看過的寫法：

```
my $input = <STDIN>;          # 從標準輸入裝置取得資料
print $input;
```

這時候，當我們從鍵盤輸入時，Perl就可以正確的取得資訊，並且透過STDIN取得使用者用鍵盤打入的一行字串。因此他的運作方式就是以一個檔案代號來進行。當然，你可以透過系統函式庫的配合，讓你的標準輸入轉為其他設備之後你就進行其他運用，不過這顯然不是這裡的主題，還是讓我們言歸正傳。對於Perl來說，他從檔案系統讀入資料是以行為單位，因此即使是利用STDIN，Perl還是會等到使用者鍵入換行鍵時才會有所動作。

STDOUT：相對於標準輸入，這就是所謂的標準輸出，也就是在正常狀況下，你希望Perl輸出的結果就是透過STDOUT來進行輸出的。而一般來說，我們所使用的就是螢幕輸出。你可以看看這個程式裡的寫法：

```
my $output = "標準輸出";
```

```
print "Soutput\n";  
print STDOUT "Soutput\n";
```

沒錯，就像我們所預期的，Perl透過螢幕印出了兩行一模一樣的結果，也就是印了兩行「標準輸出」。原因非常簡單，因為當我們使用print的指令時，Perl會使用STDOUT當作預設的檔案代號，所以一般狀況下，如果我們沒有指定檔案代號時，Perl就會自動輸出到STDOUT。所以事實上，我們早就開始使用檔案代號了，只是我們自己並沒有發覺。或說，Perl原來的期望就是希望使用者都可以在最沒有負擔的狀況下任意輸出到螢幕，或從鍵盤輸入，畢竟Perl程式設計師那麼的怕麻煩，一般的鍵盤輸入，螢幕輸出又是使用的那麼頻繁，當然要讓程式設計師以最簡單的方式達成。而且非常顯然，這個目的也算達到了。

STDERR：標準的錯誤串流，也就是程式錯誤的標準輸出。正常而言，當程式發生錯誤時，程式可以發出錯誤訊息來通知使用者，這時候這些錯誤訊息也能透過檔案代號處理，把這些訊息丟進錯誤訊息串流。不過這樣說實在不太容易理解，那我們來玩個遊戲吧：

```
my $output = "標準輸出";  
  
print "Soutput\n";  
print STDERR "Soutput\n";
```

我們一開始定義了一個字串\$output，一開始我們先直接從標準輸出印出這個字串，接下來我們便要求Perl把這個字串送出到錯誤串流中。這樣會發生甚麼有趣的事呢？讓我們來看看：

```
[hcchi en@Apple]% perl stderr.pl  
標準輸出  
標準輸出  
[hcchi en@Apple]% perl stderr.pl > error.txt  
標準輸出
```

第一次，我們直接執行了stderr.pl這支程式，而結果顯然有點平淡無奇。於是我們第二次執行時，就在後面加上了">error.txt"，對於熟悉Unix操作的人大概知道，這樣的方式其實是把程式執行時的錯誤訊息導向檔案"error.txt"了。所以STDOUT只輸出了第一行的print結果，而系統也產生了另外的error.txt的檔案，因為我們把標準錯誤串流送到了這個檔案裡，所以我們可以發現檔案裡正好有我們輸出到標準錯誤串流的字串。這樣的作法對於可能把Perl拿來進行系統管理的腳本程式時，就可以發揮很大的功能。因為我們也許希望某個程式可以幫我們進行一些日常的瑣事，而在處理這些瑣事的同時，如果發生甚麼異常狀況，可以把錯誤訊息存在某個檔案中，這樣一來我們就可以只檢查這個日誌檔案。

ARGV：我們可以直接利用參數來讀取某些檔案的內容，使用者只需要在執行程式時，在程式後加上檔案名稱作為參數，然後在程式中我們就可以直接讀到檔案的內容了。還是用個例子比較容易理解：

```
my $input = <ARGV>;  
print "$input\n";  
  
於是我們試著執行它，並且加上參數"error.txt"  
  
[hcchi en@Apple]% perl argv.pl > error.txt  
標準輸出
```

沒錯，當我們用了剛剛得到的error.txt當參數時，程式裡面直接使用預設檔案代號ARGV來讀取檔案內容，所以當我們印出來時，就可以看到剛剛寫入檔案的內容了。不過由於Perl讀檔案的性質，其實我們只印出了檔

案內的第一行，不過這部份我們稍後會再提到，這裡暫且略過不談。

不過Perl的ARGV其實非常好用，讓我們來看看使用陣列形式的@ARGV。也就是程式的參數，跟我們曾經提過的副常式參數有幾分相似。它也是把取得的參數放入陣列中，然後在程式裡，就可以直接叫用陣列，取出參數，就像這樣：

```
my $input = shift @ARGV;
print "$input\n";
```

我們用同樣的方式執行，可以看到這樣的結果：

```
[hcchi en@Apple]% perl argv.pl error.txt
error.txt
```

另外，我們也可以對ARGV進行一般檔案代號的操作方式，不過這些將在稍後提到檔案操作時再來討論。

### 10.3 檔案的基本操作

我們剛剛提到了一些Perl預設的檔案代號，這些檔案代號都是由Perl自動產生的。因此當我們開始執行Perl的程式時，就可以直接使用這些檔案代號。可是除此之外，當我們希望自己來對某些檔案進行存取時，就必須手動控制某些程序。所以現在應該來關心一下，當我們要手動進行這些檔案的控管時，應該怎麼做呢？

#### 10.3.1 開檔/關檔

最基本的，我們要先開啟一個檔案，也就是我們必須將檔案代號和我們想要存取的檔案接上線。首先，我們可以使用open這個指令來開啟檔案代號，並且指定這個檔案代號所對應的檔案名稱，所以我們使用的指令應該會這樣：

```
open FILE, "file.txt";
open OUTPUT, "<output.txt"; # 從檔案輸出
open INPUT, ">input.txt"; # 輸入到檔案
open append, ">>append.txt"; # 附加在現有檔案結尾
```

其實要開起一個檔案代號非常的容易，至少從上面的例子來看，應該還算是非常的平易近人。那麼我們只需要稍微的解釋一些特殊的部份，大部份的人應該就可以輕鬆的開始使用檔案代號了。

首先，最基本的語法也就是利用open這個指令來結合檔案代號跟系統上實際的檔案。所以我們看到了所有的敘述都是以open接下檔案代號，接著是檔案的名稱。這樣一來，我們就把檔案代號跟檔案名稱連接起來，當然，前提是沒有錯誤發生。不過不管如何，這看起來應該非常容易了。接下來，看看在檔案名稱前面有一些大，小於符號，這些又是甚麼意思呢？這些符號主要在於對於檔案操作需求不同而產生不同的形式。首先我們看到的是一個小於(<)符號，這個符號代表我們會從這個檔案輸出資料，其實如果你對Unix系統有一點熟悉，你會發現這些表示方式跟在一般使用轉向的方式接近。所以當你使用小於符號時，就像把檔案的資料轉向到檔案代號中。如果你可以想像小於符號的方向性，那麼大於符號也就是同樣道理了。大於符號也就是把資料從檔案代號中轉入實際的檔案系統裡，也就是寫入到某個檔案中。而如果系統中沒有這個檔案，Perl會細心的幫你建立這個檔案，然後你透過檔案代號送出的資料就會由Perl幫你寫入檔案中。不過有一個部份必須要特別注意的地方，也就是如果你透過大於符號建立的檔案繫結，Perl會把你指定的檔案視為全新的檔案，就如我們所說的，如果你的系統中沒有這個檔案，Perl會先幫你建立一個新的檔案。不過如果你的系統本來就已經存在同樣的檔名，那麼Perl會把原來的檔名清空，然後再把資料寫入。當然，這樣就遇到問題了，因為如果你的程式正在監視網站伺服器，而你希望只要伺服器有狀況發生就把發生的狀況寫入日誌檔。這時候你大多會希望保留舊的日誌，那麼如果Perl每次都清空舊的日誌內容就會讓我們造成困擾。這時候我們總會希望Perl能把新的狀況附加在原來的檔案最後面的位置，那麼我們就應該使用兩個大於(>>)的符號，這也就是">>"跟">"的不同之處。

既然你開啟了一個檔案代號，最好的方式就是在你使用完後要歸回原處(從小媽媽就這麼告誡我們)。因此如果你不再使用某個檔案代號時，你最好養成關閉這些檔案代號的習慣，對了，應該還要提醒的是「適時」關閉不需要的檔案代號。雖然Perl會在程式結束時自動幫你關閉所有還開著的檔案代號，不過有些時候，你如



果沒有在檔案處理完之後就儘快處理的話，恐怕會有讓系統資源的負擔增加。

至於關閉檔案代號的方式也是非常簡單，你只要使用close這個關鍵字，然後告訴Perl你所要關閉的檔案代號，這樣就沒問題了。因此你如果需要關閉檔案代號，你只需要這麼做：

```
close FILE;
```

沒錯，就是這麼容易。不過卻也相當重要，至少你應該考慮好你自己的系統資源管理。否則等到等到持續拖累系統資源時才要怪罪Perl時可就有失公允了。另外，Perl也會在你關閉檔案代號時檢查緩衝區是否還存有資料，如果有的話，Perl也會先把資料寫入檔案，然後關閉檔案。另外，檔案也可能因為你的開啟而導致其他人無法對它正常的操作，因此盡可能在完成檔案操作後馬上關閉檔案代號是重要的習慣。

### 10.3.2 意外處理

有些時候，當我們想要開啟檔案時卻會發現一些狀況。例如我們想要從某個已經存在的檔案中讀入某些資料，可是卻發生檔案不存在，或是權限不足，而無法讀入的狀況。我們先看看以下的例子：

```
#!/usr/local/bin/perl

use strict;

open FILE, "<foo.txt";
while (<FILE>) {
    print $_;
}
```

在這裡，我們希望開啟一個檔案"foo.txt"，並且從檔案中讀取資料，接著再把檔案內容逐行印出。不過非常可惜，我們的系統中並沒有這個檔案。不過Perl預設並不會提醒你這樣的狀況，而且如果你沒有使用任何的警告或中斷，Perl也能安穩的執行完這個程式，當然結果是「沒有結果」。可是當我們在寫程式，或是使用者在跟程式進行互動時，實在難保這些時候都不會甚麼錯誤會發生，也許只要把檔案名稱打錯，可是Perl卻不會自動的警告你。於是我們應該考慮發出一些警告，讓發生錯誤的人可以即時修正錯誤。當然，你可以使用warnings來讓Perl對於人為的錯誤發生一些警告，不過我們還有另外一種方法可以讓你更輕易的掌握錯誤發生的狀況，也就是讓程式「死去(die)」。

die函式就像他的字面意思，他可以讓程式停止執行，也就是讓程式「死去」。因此當我們希望程式在某些狀況下應該停止執行時，我們就可以使用die函式來達成。而檔案發生問題的狀況則是die函式經常被使用的地方。因為很多時候我們一旦開啟了某個檔案，大多就會把操作內容圍繞著這個被開啟的檔案，可是如果檔案其實沒有被正確的開啟，就很容易產生一些難以預料的問題，因此我們可以在檔案開啟失敗時就讓程式停止執行。以剛剛的程式作為例子，我們就可以把開啟檔案的部份寫成：

```
open File, "foo.txt" or die "開啟檔案失敗: $!";
```

在這裡，有幾個地方需要解釋的，首先自然就是die的用法。我們先嘗試開啟foo.txt這個檔案，接著用了一個邏輯運算元'or'，後面接著使用die這個敘述。根據我們對or運算符的了解，程式會先嘗試開啟檔案"foo.txt"，如果成功開啟，就會傳回1，因此or後面的敘述就會被省略。相反的，如果開啟檔案失敗，open敘述會傳回0。如此一來，Perl就會去執行or後面的敘述，因此他就會die了，也就是只執行到這裡為止。

利用die結束程式的執行時，我們會希望知道程式為甚麼進入die的狀況，因此我們便利用die印出目前的情況。這聽起來就像程式說完遺言之後就不動了。而die的列印就跟我們一般使用print沒甚麼不同，因此我們可以加上可以提醒程式寫作者或使用者的字串。不過在剛剛的例子，我們看到了一個不尋常的變數："\$!"。這是Perl預設的一個變數，他會儲存系統產生出來的錯誤訊息。因為當我們透過Perl要進行檔案的存取時，其實只是透過Perl和作業系統進行溝通，因此一旦Perl對作業系統的要求產生失敗的狀況，他便會從作業系統得到相關的錯誤訊息，而這個訊息也會被存入\$!這個變數中。

所以如果我們執行剛剛改過的那個程式，就可以得到像這樣的結果：

```
[hcchi en@Apple]% perl ch3.pl
開啟檔案失敗: No such file or directory at ch3.pl line 5.
```

因為檔案不存在的原因，導致這一支Perl程式無法繼續執行而在執行完die之後就停止了。而且die這個指令也在我們的要求下，傳達了系統的錯誤訊息給我們，問題發生在你要開啟檔案時卻沒有發現這個檔案或資料夾。所以利用die這個指令，你就可以在程式無法正確開啟檔案時，就馬上中斷程式，以避免不可預知的問題產生。

既然提到die，我們就順便來談一下die的親戚，"warn"吧！當你發生一些狀況，可能導致程式發生無法正常運作時，你會希望使用die來強制中斷程式的執行。可是有些時候，錯誤也許並沒有這麼嚴重，那麼你就只需要發出一些警告，讓執行者知道程式出了一點問題，讓他們決定是否應該中斷程式吧！我們把剛剛的程式改成這樣：

```
#!/usr/local/bin/perl

use strict;

open FILE, "<foo.txt" or warn "open failed: $!";
while (<FILE>) {
    print $_;
}

print "程式在這裡結束了\n";
```

你應該發現了，我們把die改成了warn，然後最後加了一行列印的指令，告訴我們程式的結尾在那裡。接下來我們來試著執行這支修改過的程式，你會看到這樣的結果：

```
[hcchi en@Apple]% perl ch3.pl
open failed: No such file or directory at ch3.pl line 5.
the end of the script
```

### 10.3.3 讀出與寫入

在我們可以正確的開啟檔案代號之後，接下來我們就可以開始存取檔案中的資料，當然最主要的就是讀取，以及寫入檔案。

透過檔案代號來讀取檔案內容倒是不太有甚麼困難。我們大多使用鑽石符號(<>)來進行檔案內容的讀取。所以我們可以像這樣進行檔案操作：

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

use strict;

open LOG, "/var/log/messages";           # 打開這個日誌檔
while (<LOG>) {                           # 利用鑽石符號讀入資料
    print if (/sudo/);                   # 符合比對的資料就列印出來
}
```



看起來非常容易，不是嗎？

我們先用剛剛了解的方式開啟了一個檔案代號，並且利用這個檔案代號聯繫到檔案"/var/log/messages"。在一些Unix系統中也許會看到這個檔案，它會紀錄一些使用者登入或是使用root權限的消息。而在這個檔案中，如果有使用者利用sudo這個指令進行某些操作時也會被記錄下來。因此我們就可以透過這個檔案知道伺服器上有些甚麼狀況正在發生。

接下來我們透過鑽石符號開始逐行讀取日誌檔案中的資料，透過迴圈while讀取檔案中的資料時，while會把所讀到的資料內容放進Perl的預設變數\$\_中，一直到檔案結束，傳回EOF時，迴圈便會結束。因此我們就將所讀取的資料進行比對，以sudo這個關鍵字作為比對樣式，把符合的結果印出來。

這樣一來，只要系統中有人使用sudo進行系統操作時，我們就可以檢查出來，而且印出來的結果會像是這樣：

```
TTY=ttyp0 ; PWD=/var/log ; USER=root ; COMMAND=/bin/rm -rf httpd-error.log
TTY=ttyp0 ; PWD=/var/log ; USER=root ; COMMAND=/bin/rm -rf httpd-access.log
TTY=ttyp0 ; PWD=/var/log ; USER=root ; COMMAND=/bin/rm -rf 192.168.1.1_access_log
192.168.1.1_error_log
TTY=ttyp0 ; PWD=/usr/home ; USER=root ; COMMAND=/bin/rm -rf interchange/
TTY=ttyp0 ; PWD=/usr/home ; USER=root ; COMMAND=/bin/rm -rf gugod/
TTY=ttyp0 ; PWD=/usr/home ; USER=root ; COMMAND=/bin/rm -rf mysql /
TTY=ttyp0 ; PWD=/ ; USER=root ; COMMAND=/bin/rm kernel.old
TTY=ttyp0 ; PWD=/ ; USER=root ; COMMAND=/bin/rm -rf modules.old/
TTY=ttyp0 ; PWD=/ ; USER=root ; COMMAND=/bin/rm -rf opt/
```

如果你是負責管理一些Unix的伺服器，利用這樣簡單的方式，確實可以幫忙你完成不少工作。很顯然，利用檔案的操作，你還可以進行更多對日誌檔案的分析。例如你可以分析網站伺服器的各項資料，雖然其實已經有很多人用Perl幫你完成這樣的工作了。(註二)

基本上，從檔案內讀取內容的方式就是這麼容易，因此你可以簡單的運用檔案的內容進行所需要的工作。還記得我們在介紹open時的說明嗎？我們有幾個開啟檔案的方式包括了幾種描述子，例如大於(>)，小於(<)，以及兩個大於(>>)。而且我們也都簡單的描述過他們的差異，現在也許就是測試這些描述子的好時機，我們先來看看小於符號用於開檔的時候，會有甚麼影響。

我們之前也提過小於符號用在開檔作為描述的話，是用來表示從檔案內讀取資料。那我們是不是就只能允許使用者讀取資料呢？先來看看這個小小的程式吧：

```
open LOG, "<log.txt" or die $!;
while (<LOG>) {
    print $_;
}
print LOG "write to log" or die $!;
```

假設我們已經有了"log.txt"這個檔案，否則程式就會掛在中間，沒辦法繼續執行。那麼我們來看看執行結果吧：

```
file for log
Bad file descriptor at ch3.pl line 9, <LOG> line 1.
```

第一行就是原來log.txt裡面的內容，我們可以很輕鬆的讀出其中的資料，並且印出來，可是當我們要將資料寫入時，卻出現了錯誤訊息。沒錯，當初我們在開啟這個檔案時，只要求Perl給我們一個可以讀出資料的檔案，如今要求寫入，果然就遭到拒絕。

看來一但我們使用了小於符號作為開啟檔案代號的描述子，那麼我們就不能輕易的把資料寫入所開啟的檔案中。想當然爾，Perl應該也不會讓我們在開啟一個利用大於符號指定為寫入的檔案中把資料讀出吧？要想測試這樣的結論，我們只需要把剛剛的程式修改一個字元，也就是把小於符號改成大於，那麼就讓我們來看看

執行後的結果吧：

我們嘗試著執行被我們修改了一個字元的程式，結果發生了甚麼事呢？檔案沒有輸出任何結果。好像很出乎意料？其實一點也不，而且正如Perl所要求我們的，我們使用了大於符號表明我們想要把資料寫入檔案log.txt，因此當我們想要從檔案讀取資料並且逐行印出結果時就無法成真。不過我們接下來去看看log.txt的內容。正如我們所預料的，程式已經正確的把字串"write to log"寫到檔案log.txt裡面了。

既然使用大於符號跟小於符號都符合我們的期待，那麼如果我們甚麼描述子都沒有使用，會是甚麼樣的情況呢？我們只需要使用剛剛的測試程式，並且把描述子全部取消，再來試試結果如何吧！

結果我們發現，Perl還是可以讀出檔案的內容，可是卻無法寫入。也就是跟我們使用小於符號時是一樣的狀況，這點其實對於經常必須使用檔案的人來說其實是非常重要的。所以如果你有機會使用檔案的存取時，可別忘了這一點。

另外，大於符號與兩個大於的差別我們也曾經提過，這部份對於可能使用Perl來進行日常管理工作的人更是必須牢記。我們之前提過，一樣是開啟一個可以寫入的檔案，使用一個大於符號(>)的時候，Perl會判斷你是否已經有存在這個檔名的檔案，如果檔案已經存在，那麼Perl將會清空檔案內容，把他視作一個新的檔案來進行操作。如果在系統中檔案並不存在，那麼Perl就會跟系統要求開啟一個新的檔案。當然，在你使用兩個大於符號的時候，Perl會把你寫入檔案的內容以附加的方式存入。當然，如果你的系統中並沒有這個檔案，那麼Perl也會先開啟一個新檔，並且把你所要求的內容寫入檔案中。這對於想要建立類似日誌檔的需求有著絕對的幫助，例如你可能需要Perl來作為監控網路的狀況，這時候你會需要每次有新狀況時就把它記錄下來，而且需要保留原來的紀錄。那麼如果你還是使用大於符號的話，你可就要小心原來的資料內容遺失了。

當然，我們知道開啟檔案時可以利用三種描述子去指定所要開啟檔案代號的狀態，不過如果你甚麼都沒加的狀況下，Perl又會作怎麼樣的處理呢？我們繼續用剛剛的例子來進行實驗吧。我們把開啟檔案的描述子拿掉，其他的部份一切照舊。所以你的程式就像這樣：

```
open LOG, "log.txt" or die $!;  
print LOG "write to log\n" or die $!;
```

接著我們發現，這樣的結果就跟我們使用小於符號的效果是相同的，也就是Perl只會從檔案中讀出資料，卻無法寫入。

有了基本讀寫檔案的能力之後，我們還必須了解該怎麼樣透過Perl去控制系統的檔案以及資料夾。這樣才能確實掌握系統的檔案管理，尤其當你希望使用Perl來進行系統管理時，也就會更需要這樣的能力，所以我們接下來就要討論利用Perl對檔案系統的操作。

習題：

1. 試著將下面的資料利用perl寫入檔案中：

```
Paul, 26933211  
Mary, 21334566  
John, 23456789
```

2. 在檔案中新增下列資料：

```
Peter, 27216543  
Ruby, 27820022
```

3. 從剛剛已經存入資料的檔案讀出檔案內容，並且印出結果。

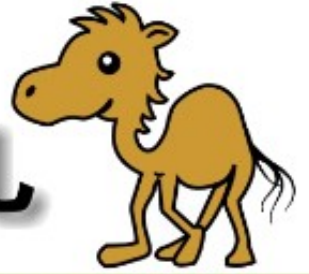
註一：不過當你打算這麼作的時候，也許要考慮這支程式未來只有你在維護，否則你這樣的動作很可能會因為接下來維護的人需要花更多的時間來看懂程式而提高不少維護成本。

註二：其實跟這章主題不太有關，不過例如awstats就是這類型的工具。





# Perl 學習手札



There is more than one way to do it.

## 11. 檔案系統

上一章我們提到了一些關於在Perl當中使用檔案代號來進行檔案存取的工作，不過要能靈活運用這些操作，你應該要有對於系統本身的檔案架構有一些認識。因為運用檔案代號，實際上你也是在操控整個系統的目錄跟檔案。所以我們接下來就要簡單提醒大家一些基本的事項，並且告訴大家應該怎麼利用Perl去進行檔案的操作。

### 11.1 檔案測試

我們在上一章曾經嘗試打開一個檔案，並且從檔案內讀出其中的內容。不過我們也遇到了一些問題，也就是檔案可能會因為不存在而使資料讀取發生問題。因此我們利用die的方式來判斷，假如程式無法打開這個檔案代號，那麼就中止程式繼續進行。當然，找不到檔案是我們設法開啟檔案代號時可能發生的錯誤之一。我們也許還可能發生其他問題，比如沒有權限打開指定的檔案等等。不過對於這當中的某些狀況，我們其實在準備開啟檔案時可以先進行測試，也就是所謂的檔案測試，在說明可以進行測試的項目之前，我們可以先來看看這個例子：

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

use strict;

my $logfile = "/var/log/messages"; # 先指定檔案到變數 $logfile
if (-e $logfile) {                 # 判斷檔案是否存在
    open LOG, $logfile or die "$!"; # 開啟檔案代號
    my $line_num = 1;
    while (<LOG>) {
        print "$line_num\t$_\n";
        $line_num++;
    }
} else {
    print "檔案不存在\n";
}
```

這個程式的主要工作在於讀出系統日誌檔的內容，並且幫忙加上行號印出。當然，我們先指定要開啟的檔案名稱是"/var/log/messages"這個檔案，接下來便利用檔案判斷的參數"-e"來確定檔案是否存在。如果這個檔案確實存在，我們就打開檔案代號"LOG"，用來聯繫"\$logfile"這個檔案，也就是"/var/log/messages"。當然，這時候我們雖然確定檔案存在，可是因為還是可能存在其他導致無法正常開啟檔案的狀況，因此我們還是決定一但開啟失敗就利用die印出錯誤訊息，然後中斷程式。如果檔案開啟沒有問題，我們就可以開始一行一行把資料讀進來，然後加上行號後輸出了。

這樣看來，"-e"的判斷似乎功用不大，因為我們判斷如果檔案不存在，好像也沒有特殊的動作。所以我們來讓"-e"看起來能有些幫助：

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

use strict;

while (-e (my $logfile = shift)) { # 判斷檔案是否存在
    open LOG, $logfile or die "$!"; # 開啟檔案代號
```



```

my $line_num = 1;
while (<LOG>) {
    print "$line_num\t$_\n";
    $line_num++;
}
}

```

這樣看起來好像有趣了一些，我們來看看到底改了甚麼。首先，我們把原來指定給變數\$logfile的檔案取消，讓\$logfile變成是使用者由執行時輸入的參數。接著我們依然檢查了這個檔案是否存在，如果存在則打開並加上行數印出。

其實並不困難，我們只需要以指定的參數就可以用來檢查檔案的屬性。所以我們來看看到底有那些參數可以使用：

- A 檔案上次存取至今的時間
- B 檔案被判斷為二進位檔
- C 檔案的 inode 被更改至今的時間
- M 檔案上次修改至今的時間
- O 目前實際使用者是否為該檔案或目錄的擁有者
- R 目前實際的使用者具有讀的權限
- S 檔案代號是否為 socket
- T 檔案判斷為文字檔
- W 目前實際的使用者具有寫的權限
- X 目前實際的使用者具有執行的權限
- c 字元型檔案
- e 檢查檔案或目錄是否存在
- f 判斷檔案是否為文字檔
- g 檔案或目錄具有 setgid 屬性
- k 檔案或目錄設定了 sticky 位元
- l 檔案代號是一個符號連結
- o 目前的使用者是否為該檔案或目錄的擁有者
- r 目前的使用者具有讀的權限
- s 檔案或目錄存在而且有內容
- t 檔案代號是 TTY 裝置
- u 檔案或目錄具有 setuid 屬性
- w 目前的使用者具有寫的權限
- x 目前的使用者具有執行的權限
- z 檔案或目錄存在而且沒有內容

其中有許多是關於系統本身的相關知識，例如使用者id，群組id等等。這部份建議各位應該能夠針對自己所使用的作業系統，去找到相關的參考書籍。其他例如在Unix系統上使用，大多則採用相類似的權限判斷方式。當然，其中有些部份是僅供參考，例如檔案是否為文字檔，或是二進位檔。對於big5檔案來說，Perl就可能誤判成二進位檔。

當然，很多時候我們還是需要在對檔案進行存取之前，先確定他們相關的狀況。例如是否能夠有足夠的權限，或是我們可以得到檔案最後被修改的時間等等。大部份的時候，這些判斷可以給我們當作很好的參考。例如我們可以設定時間清除過久沒有更新的檔案等等。這些工具對於使用Perl來管理日常工作的管理者來說更是能夠提供非常好的幫助。

## 11.2 重要的檔案相關內建函式

對於系統中的檔案系統，Perl大多數的時候總是透過底層的作業系統去進行操作，因此你會發現很多的函式和作業系統提供的函式大多非常接近(註一)。這樣其實也非常能夠幫助使用者用簡單的方式記憶，而不需要多背另一套指令函式。例如我們剛剛提到的檔案測試，也就是Perl所提供第一個屬於檔案操作的函式。因此如果你想要獲得更精準的說明，你可以考慮使用"perldoc -f -X"來查看所有的測試符號。

接下來，我們來看看還有那些函式是我們可以善加利用的部份。Perl在處理檔案代號或其他檔案相關的函式多達十幾個，其實已經足以應付大多數的使用。接下來我們將挑出幾個經常被使用的內建函式，讓讀者可以



開始熟悉該怎麼在Perl中控制檔案系統。

chdir：就像你在大多數作業系統下所使用的指令一樣，你可以利用chdir來切換目前工作的目錄。因此我們可以使用下面的方式來指定我們想要操作的目錄：

```
chdir "/tmp" or die $!;  
open LOG, ">log.txt" or die $!;  
print LOG "write to log\n" or die $!;
```

沒錯，我們只是小小的修改了剛剛的程式，把原來沒有指定目錄的狀況，改成在目錄"/tmp"下了一個檔案log.txt，並且寫入字串。就像你在大多數Unix作業系統中的狀況，你也可以單獨使用chdir而沒有附帶任何的參數，這時候系統會根據你的環境變數\$ENV{HOME}來決定應該切換到哪一個目錄。

chmod：對於熟悉Unix的系統對於此應該也是非常的熟悉，這個函式就是呼叫系統中chmod的操作，來修改檔案或是目錄的權限。如果你對於系統的權限結構還不太熟悉，建議你先看一些相關的文件，可以了解Unix系統下對於權限的限制跟實作的方式。當然，Perl並不太願意改變大家的使用習慣，所以如果你經常使用Unix下的chmod指令，那麼你可以繼續你的使用習慣，就像這樣：

```
chmod 0444, 'log.txt';
```

不過也有比較具有彈性的用法，例如你可以這樣使用：

```
$mode = 0644;  
chmod $mode, 'log.txt';
```

有些部份通常會讓你搞錯，因此你必須特別注意。如果你剛剛把\$mode這個變數寫成下面的形式，那麼可能執行之後，可能會發生一些讓你意想不到的狀況。

```
$mode = "0644";  
chmod $mode, 'log.txt';
```

我們直接來看看實際的狀況吧！它的權限目前是0444。如果我們想要把它利用剛剛的權限來修改它，那麼會發生甚麼事呢？

```
Inappropriate file type or format at ch3.pl line 6.
```

Perl毫不留情的給了我們一個錯誤訊息，告訴我們這樣指定檔案權限是不被允許的。很多人可能已經一頭霧水了，我們加了引號之後到底有甚麼差別呢？你還記得我們剛剛指定權限的作法嗎？

```
$mode = 0644;
```

其實當你在使用這樣的純量賦值時，Perl會把你所指定的數字設定為八進位。可是當你幫它加上引號之後，

也就是使用了`$mode = "0644"`後，它就變為一個字串了。可是`chmod`所需要的可不是字串，而是一個八進位的數字，所以如果你使用了引號來定義權限的值，別忘了把他轉為八進位制，所以我們可以改寫成這樣：

```
$mode = '0644'; chmod oct($mode), 'log.txt';
```

當然，最省力還是前一種的方式，不過既然方法不只一種，使用者可以選擇自己最容易接受的方式。至於直接使用八進位的變數定義，應該是最被推薦的使用方式。不需要繁雜的轉換手續，也減少打字跟錯誤的機會。

`chown`：修改檔案或是資料夾的擁有者也是你在管理Unix系統會遇到的狀況。其中這包括了使用者id(uid)跟群組id(gid)，使用的方式則是將使用者id跟群組id利用串列的方式來描述，配合上想要修改的檔案，所以指令的形式應該是：

```
chown LIST;
```

用實際的例子來看，我們則可以寫成這樣：

```
chown $uid, $gid, 'log.txt';
```

至於後面的檔案，則可以利用串列的方式表示，或直接以陣列方式。也就是說，你當然可以用這樣的方式來表達`chown`的形式：

```
chown $uid, $gid, @array;
```

直接使用陣列確實是有相當的好處，我們可以利用樣式比對找出我們的所有檔案，然後一次進行相關的修改。例如在系統的使用上，我們常常使用星號(\*)作為萬用符號，比如你可以利用這樣的方式找出所有Perl的檔案：

```
ls *.pl
```

而在Perl中，也有相關的用法，也就是`glob`。因為這個功能非常重要，所以我們接下來就來看看`glob`的用法。

`glob`：他的語法其實相當簡單，也就是利用`glob`接上一個樣式，作為比對的標準。所以你可能這麼使用：

```
@filelist = glob "*.pl";
```

這樣的方式就跟你在系統下尋找符合某些條件的檔案用法一樣，所以你可以把利用`glob`所傳回來的檔案串列放入一個陣列之中。然後再針對這個陣列進行`chown`或是`chmod`相關的操作。也許你會考慮，這樣的作法跟你在shell底下的運作有甚麼差別嗎？其實很多時候，Perl可以利用這些方式把你日常必須重複進行的工作處理掉。

不過其實有時候你也許會看到某種寫法，就像這樣：

```
@filelist = <*.pl>;
```

這樣得出的結果其實跟你使用glob有著異曲同工之妙，也就是取得目前目錄下的檔案，並且依據你所描述的樣式傳回你需要的檔案。因此你可以輕易的取得你想要的檔案，例如你想要印出目錄下的所有附屬檔名是txt的檔案，那麼你只需要這麼作：

```
for $file (<*.txt>) {  
    open FH, $file;  
    print $_ while (<FH>);  
}
```

看出這其中有一些奧妙了嗎？我們利用角符號代替了glob的工作，可是同時角符號也被我們拿來作為讀取檔案內容的運算。確實是如此，那麼Perl會如何分辨其中的差別呢？其實由於檔案代號必須符合Perl的命名原則，因此Perl可以藉此判斷你目前的語境下是裡是用角括號來處理檔案代號或是進行glob的處理。當然，其中會有一些例外，比如你用這樣的方式來表達檔案代號：

```
open FH, $file;  
$filehandle = "FH";  
print $_ while (<$filehandle>);
```

這時候角括號裡面放的其實是一個Perl的純量變數，不過這個純量變數卻是被指定到另外一個檔案代號，所以Perl還是會以對待檔案代號的方式來對待它。這應該一點都不讓人意外，不過你現在應該可以應付大多數的狀況了。

link：你有時候會需要把檔案建立起鏈結，在系統底下，你可以直接使用"ln"這個指令來達成你需要的目的。而透過Perl，則可以利用link來達到一樣的工作。他的語法就像這樣：

```
my $res = link "/home/foo", "/home/bar";
```

這樣的意思就是把"/home/foo"這個檔案連結到"/home/bar"，或者你可以說"/home/bar"是"/home/foo"的一個連結。至於link這個指令則會有回傳值，如果連結成功，則回傳值為真值，相反的，如果連結失敗，則會傳回偽值。我們來試試這個例子：

```
#!/usr/local/bin/perl  
  
use strict;  
  
my $ret = link "log.txt", "log.bak";  
open FH, "log.bak" or die $! if ($ret);  
print $_ while (<FH>);
```

執行之後，我們就可以看到資料夾中多了一個叫做"log.bak"的檔案，不過如果你需要真正了解他的運作，我們還是建議你去看看關於Unix下關於檔案及資料夾的解釋，其中inode這個觀念可以幫助你確實了解這樣的連結所產生的意義。不過在這裡，我們就暫且先不深入的探討Unix下的相關部份。

mkdir：接下來，我們應該來告訴大家，該怎麼開啟自己的一個資料夾。這個指令跟你在Unix底下的使用非常接近，你只需要使用這樣的方式就可以了：

```
mkdir PATH;
```

這看起來跟你在命令列下的用法一模一樣，而且就是這麼簡單。所以你幾乎不需要學習新的東西就可以很輕鬆的在Perl底下新增一個資料夾。另外，你還可以透過umask來指定這個新資料夾的權限。而用法也是跟剛剛類似，唯一的差別只是把你希望指定的umask放在敘述的最後。所以看起來應該就像這樣：

```
mkdir PATH, umask;
```

所以你可以把新增加的這個資料夾指定某個特殊的權限，例如你希望開一個所有人都可以任意存取的資料夾，那麼就可以這樣寫：

```
mkdir foo, 0777;
```

rename：接下來我們來看看如何使用Perl來幫你的檔案改名字，其實當你開始利用Perl來對檔案進行操作時，修改檔名是非常有用的一項工具。我們可以先來看看一個實際的範例：

```
my $file = "messages.log";
if (-e $file) {
    rename $file, "$file.old";
}
open FH, ">$file";
print FH, "接下來就可以寫入資料";
```

在實際運用時，如果我們可以適時的搭配檔案的測試運算，那就可以產生出很不錯的效果。就這個例子，我們先利用"-e"來判斷檔案是否存在。如果檔案存在，我們就把檔案更名，也就是再檔案結尾加上".old"，在這裡，我們就看到了rename 的用法，也就是：

```
rename $oldfile, $newfile;
```

當我們正確的把檔案改了名字之後，就可以安心的把新的資料寫入檔案了，你應該注意到了，我悶在這裡因為是使用了大於(>)符號來進行開啟檔案代號的動作，所以如果之前沒有先把檔案更名，那麼舊有的資料就會被取代了。

rmdir：就像你在作業系統下的作法一樣，你可以利用rmdir來刪除一個資料夾。不過也跟你在終端機前使用rmdir一樣，如果資料夾裡面還有存在其他檔案，rmdir就會產生失敗，而且會傳回偽值，很顯然，這是相對於刪除成功所傳回的真值。所以如果你是Unix系統的慣用者，也許你應該非常熟悉這個函式，你只需要這麼指定：

```
rmdir FILENAME;
```

stat：其實如果你想要更靈活的使用我們介紹的這些函式來對檔案系統進行控制時，你應該要先了解stat這個重要的函式。為甚麼stat這個函式這麼重要，也許我們來看看下面的範例就能夠很快的了解了：

```
my @ret = stat "log.txt";
print "$_\n" for (@ret);
```

於是我們試著執行這個程式，會看到這樣的結果：

```
234881034      # 裝置編號
1183005        # inode 編號
33060          # 檔案模式(類型及權限)
1              # 檔案的連結數目
501            # uid
501            # gid
0              # 裝置辨識
17             # 檔案大小
1078894964     # 最後存取時間
1078894638     # 最後修改時間
1078939576     # inode 修改的時間
4096           # 檔案存取時的區塊大小
8              # 區塊的數目
```

毫無疑問，我們確實可以利用stat個函式得到相當多的檔案相關資訊，因此如果你想要對檔案進行操作之前，也許可以先利用stat來得到相關的訊息。

我們剛剛利用一個陣列來儲存stat的回傳值，這樣也許不容易分辨各個值所代表的意義，所以你當然可以改用這樣的方式來取得相關的資料：

```
($dev, $ino, $mode, $nlink, $uid, $gid, $rdev, $size, $atime, $mtime, $ctime, $blksize,
$blocks) = stat("log.txt");
```

另外，有些時候你也許會看到有人使用lstat來取得檔案的相關資訊，不過基本上這兩個函式所進行的工作應該是一樣的，所以除非你想要多打一些字，否則還是可以直接使用stat就好了。

unlink：就像你使用的rm一樣，unlink也可以讓你刪除系統中的某些檔案。而且unlink的用法十分簡單，基本上就是傳進你想要刪除的檔案串列。意思就是說，如果你搭配著glob或是角括號(<>)使用，那麼你就可以過濾出某些特殊的檔案，並且加以刪除。相信大家經過上面幾個函式的訓練，應該可以很輕易的使用這個函式，就像這樣：

```
my @files = <*.txt>;
unlink @files or die $!;
```

當然，別忘了要刪除檔案千萬要非常的小心，可別因為一時大意就把資料全部的毀了(註二)。當然，我們剛剛說了，在unlink後面所連接的參數是一個串列，所以你可以使用任何表達串列的方式，其中當然包括一一列出你所要刪除的檔案。所以如果有一個程式寫的像這樣：

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



```
unlink @_ or die $!;
```

那麼他看起來像不像陽春的rm指令呢？其實有時候玩玩也是還滿有趣的。

utime：Perl另外也提供了一個讓你修改檔案時間的函式，也就是utime。utime的用法也是傳入一個串列，所以基本上會是：

```
utime LIST;
```

不過不太一樣的地方在於你必須指定你所要修改的時間參數，所以其實比較常看到的用法也許會比較接近這樣的形式：

```
utime $atime, $mtime, @files;
```

其中第一個參數就是檔案存取時間，第二個參數就是檔案最後一次修改的時間。

### 11.3 localtime

這個函式看起來跟檔案操作並沒有甚麼直接的關係，不過我們剛剛看到了一些不太友善的數字，也就是對於檔案相關時間的描述。例如我們利用stat傳回來的日期都是這樣子的表示方式：

```
1078894964      # 最後存取時間
1078894638      # 最後修改時間
1078939576      # inode 修改的時間
```

這時候，我們就可以使用localtime來轉換成一般人可以接受而且使用的資訊。localtime會傳回一個串列，分別代表用來表示時間的各個欄位，所以你可以利用這樣的方式取得你需要的欄位：

```
@realtime = localtime($timestamp);
```

只是如果你使用這樣的方式，恐怕自己也很難很快的使用，所以也許可以換一個方式：

```
($sec, $min, $hour, $day, $mon, $year, $yday, $isdat) =  
localtime ($timestamp);
```

所以如果你想要取得檔案最後修改的正確時間，你可以利用下面的方式達成：

```
my ($dev, $ino, $mode, $nlink, $uid, $gid, $rdev, $size, $atime, $mtime, $ctime, $blksize,  
$blocks) = stat("log.txt");  
($sec, $min, $hour, $day, $mon, $year, $yday, $isdat) =  
localtime ($mtime);
```

呼，確實有一點冗長。不過確實可以讓你正確的取得大部份的資訊。

習題：

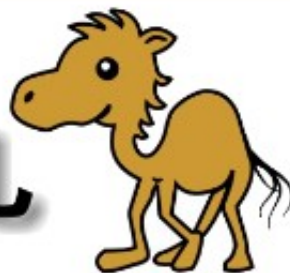
1. 列出目前所在位置的所有檔案/資料夾名稱。
2. 承一，只列出資料夾名稱。
3. 利用Perl，把目錄下所有附檔名為.pl的檔案修改權限為可執行。

註一：當然所謂的接近，指的是和Unix系統的接近。

註二：上面的程式就讓本書內容差點付之一炬，幸好作者使用了版本控制系統來進行備份。



# Perl 學習手札



There is more than one way to do it.

## 第十二章 字串處理

我們前面兩章提到了許多關於檔案的操作，現在我們應該可以很輕鬆的從檔案中取得我們需要的資訊了。不過如果只是空有一大堆的資訊，卻沒有辦法處理的話，只怕也沒有甚麼幫助。不過既然對於Perl來說，大多數的東西都是由數字跟字串組合而成，那麼一旦我們可以用簡單的方式來整理字串的話，那麼應該就可以讓這些資訊變得相當有用。

### 12.1 簡單的字串形式

我們在講解變數的時候已經提過關於Perl是如何對待字串的，雖然對於不少其他程式語言來說，字串其實只是字元的串列。可是Perl卻簡化了這樣的觀念，因此反而讓字元變成長度為一的字串。就像對待數字一樣，Perl並不會要求程式設計師去強制規定某些變數只能放整數，某些變數只能放浮點數。

這樣寬鬆的規定確實讓程式設計師省了許多麻煩，不過當你的分類越粗略時，要怎麼有效的對這些資料進行處理就顯得更加重要。而就像許多人對於Perl的印象，它在處理字串時非常的具有威力。這當中的原因除了正規表示式之外，Perl對於字串的控制顯然也有一些有趣的部份。

對於字串最基本的操作應該就是長度了，我們經常會要求知道字串的長度，這時候，只需要使用length這個函式就可以取得你所指定字串的長度了。

```
my $string = "string";  
print length($string);          # 長度是6
```

當然，有時候你會被某些控制字元所欺騙，因為他們也是佔有長度的，就像這樣：

```
my $string = "string\n";  
print length($string);          # 這時候長度變成7了
```

取得字串長度只是第一步，接下來我們可能需要找出字串的相關性。例如我們會想要取得某個字串的其中一段。不過我們可能會先需要知道這個子字串在原來字串的位置這時候就可以使用index來取得相關的訊息了。用實際的例子我們可以很容易看到index的用法：

```
my $mainstring = "Perl Mongers";  
my $substring = "Mongers";  
print index($mainstring, $substring); # 印出 5
```

看來相當容易對吧，Perl會告訴你子字串第一個字母所在的位置，只是字串是由0開始算起。也就是說，如果你在字串的一開始就找到符合的子字串，那麼Perl就會傳回0。不過如果Perl發現你所指定的子字串不在原來的字串中，那麼就會傳回-1。

當然，有些人會關心中文字串的處理，我們先來試試下面的例子：

```
my $mainstring = "台北Perl推廣組"; # Big5 碼的中文
my $substring = "推廣組";
print index($mainstring, $substring); # 印出8
```

其實index傳回的是位元，所以如果你要利用index來找到某些中文字在字串中是位於第幾個位元那麼就沒有問題。當然，如果你要的是以中文的角度來看，那麼「字」的觀念在這裡顯然並不存在。

另外，就像正規表示式一樣，index在進行比對時，也會確定找到你所需要的字串就停止了。所以index傳回的就永遠會是第一次找到子字串的位置。實際的結果會像是這樣：

```
my $mainstring = "perl training by Taipei perl mongers";
my $substring = "perl";
print index($mainstring, $substring); # 結果是0
```

因為Perl在一開始就找到了比對成功的字串"perl"，因此它馬上傳回0，然後就停止比對了。可是這樣有時候是不是會有些不方便呢？所以我們來看看Perl對於index這個函式的描述。

```
index STR, SUBSTR, POSITION
index STR, SUBSTR
```

我們好像發現一些曙光，沒錯，根據上面的語法，其實我們還可以使用index的第三個參數，也就是位置。所以你可以要求index從第幾個位元開始找起，例如：

```
my $mainstring = "perl training by Taipei perl mongers";
my $substring = "perl";
my $first = index($mainstring, $substring); # 先找到第一次出現perl的地方
print index($mainstring, $substring, $first+1); # 接下去找
```

這樣的用法就可以讓你找到下一個出現子字串的地方。當然，如果你沒有加第三個參數的話，那麼index會把它預設為0。也就是我們一開始一直使用的方式。

不過如果你不知道子字串會出現多少次，可是你又想找到最後一次出現的位置，那麼你會想要怎麼作呢？用個迴圈好像是我們目前可以想到的作法，所以我們就來試試吧：

```
my $mainstring = "perl training by Taipei perl mongers";
my $substring = "perl";
my ($pos, $current);
my $pos;
my $current = -1;
until ($pos == -1) { # 到找不到正確字串為止
    $pos = index($mainstring, $substring, $current + 1); # 從上次找到的位置往下找
    $current = $pos unless ($pos == -1);
}

print $current; # 印出 24
```

看起來好點小小的複雜，因為我們必須用一個迴圈去搜尋所有的子字串，一直到它找到最後一個。不過有沒有可能從字串的尾端去找，那麼我們就只需要找到第一個符合的字串，因為對於從字串開頭而言，那就會是

最後一次比對成功的字串了。

看來這樣的需求不少，因此Perl的開發者也就提供了另外一個函式，也就是rindex，基本上rindex的使用方式跟index幾乎一模一樣，只不過它是從字串尾端開始找起。既然如此，我們就改用rindex來完成剛剛的工作：

```
my $mainstring = "perl training by Taipei perl mongers";
my $substring = "perl";
print rindex($mainstring, $substring);          # 同樣印出 24
```

這樣顯然方便了許多，不過對於rindex來說，如果我們指定了第三個參數，那其實是用來表示搜尋的上限。也就是我們要求rindex在某個位置之前的就不找了。這樣描述似乎太過籠統，我們不如來看看實際的運作情形吧：

```
my $mainstring = "Taipei perl mongers";
my $substring = "perl";
print rindex($mainstring, $substring, 4);      # 結果傳回 -1
```

其實參數的意義也就是「以這裡為開始搜尋的起點」，所以如果我們把參數設定為4的話，Perl就只會從第四個位元往回進行比對，所以當然不會比對成功。

利用index找出字串的位置之後，我們還可以利用substr來取出某個字串內的字串。我們先看看substr的標準語法：

```
substr EXPR, OFFSET, LENGTH
substr EXPR, OFFSET
```

最簡單的方式就是只有指定要處理的字串跟另一個我們想取得字串的起始點，所以你可以讓它看起來像這樣：

```
my $string = "substring";
print substr($string, 3);    # 果然印出 string 了
```

如果你沒有傳入長度這個參數，那麼Perl會預設幫你取到字串結束。所以我們剛剛取得的字串就是"string"，如果你想要的只是"str"三個字母，你就可以指定長度，也就是像這樣：

```
my $string = "substring";
print substr($string, 3, 3); # 這樣就只會印出 str
```

有時候如果字串太長，也許從字串結尾開始算起會比較容易，就像index搜尋字串的位置，可以利用rindex來要求Perl從字串尾端找起，那麼substr要如何使用類似的方式呢？答案就是利用負數的起始點，這樣說好像不如直接看個範例：

```
my $string = "Taipei Perl Mongers";
```



```
print substr($string, -12, 4); #印出 Perl
```

另外，我們之前使用過正規表示式來進行取代的工作，例如下面的字串，我們想把"London"以"Taipei"取代，所以可以利用正規表示式，作這樣的處理：

```
my $string = "London Perl Mongers";  
$string =~ s/London/Taipei/;
```

當然，有些時候使用正規表示式未必比較方便。或是我們可以取得的資料有限，這樣的情況下，也許可以利用substr來進行字串替換。substr也可以進行替換，別擔心，你沒看錯，我們就來實驗看看，利用substr來把"London"換成"Taipei"。

```
my $string = "London Perl Mongers";  
substr($string, 0, 6) = "Taipei";  
print $string; # 就會印出 "Taipei Perl Monger"
```

這樣看起來好像沒甚麼，顯然不夠絢麗，我們來把它改寫一下吧！

```
my $string = "London Perl Mongers";  
print substr($string, 0, 6) = "New York";  
print $string; # 你完全不需要考慮字串長度
```

字串長度對Perl來說並不是個問題，所以我們可以很安心的使用長度不相等的字串來進行替換，Perl可以自動的幫你處理長度的問題。其實這種需求顯然相當的高，所以這也是substr的另一種標準語法，也就是說，我們可以把剛剛的語法用這種方式來取代：

```
my $string = "London Perl Mongers";  
substr($string, 0, 6, "New York"); # 使用第四個參數  
print $string; # 也是會替換為 New York Perl Mongers
```

## 12.2 uc 與 lc

字串中，偶而會有一些惱人的狀況，也就是字串的大小寫問題。例如你弄了一個會員帳號系統，因此這個系統必須讓管理者可以開帳號，使用者可以登入等等。有許多牽涉到帳號的輸入，比對問題，這時候如果還有字母大小寫的問題，也許會更讓人氣餒，尤其目前的大多數使用者幾乎都習慣了大小寫不分的使用狀況。所以有時候也許需要藉由系統自動轉換的方式來避開這一類瑣碎的事。

uc也就是upper case的意思，所以很清楚的，它會幫你將字串中的英文字母轉換成大寫，然後回傳，就像這樣：

```
my $string = "I want to get the uppercased string";  
print uc $string; # 結果就變成了 "I WANT TO GET THE UPPERCASED STRING"
```

怎麼樣，一點都不意外吧！而且依此類推，lc 則是轉成小寫之後回傳，這應該不需要重新舉例了。

這樣一來，我們雖然可以取得全部大小或全部小寫的字串，可是在更多時候，我們其實只要字首的大小就好了，那麼可以怎麼作呢？也許可以考慮使用ucfirst，看這個函式名稱就覺得它是我們想要的東西。，既然如此，那我們就直接來試一下吧：

```
my $string = "upper case";  
print ucfirst $string;          # 印出 Upper case
```

就像我們所預期的一樣，我們讓Perl把第一個字母印出了大寫，不過這完全是意料之中？相對應於ucfirst，Perl也提供了lcfirst這個函式，而且正如大家所猜想的一樣，它會把字串的第一個字母轉為小寫。

### 12.3 sprintf

我們已經非常習慣使用print來印出我們執行程式所得到的結果了，可是很多時候print印出的結果卻未必讓人滿意，不滿意的原因有很多時候是因為它的輸出格式無法依照我們的要求，或者說我們需要花更多的力氣才能達到我們所期待的樣子。所以這時候，sprintf就可以派上用場了。sprintf主要是可以幫助我們作格式化的列印指令。例如你總是希望印出兩位數的小數點，那麼這時候，你應該就會非常需要sprintf來幫助你。我們來看看我們可以怎麼作呢？

```
my $num = 21.3;  
my $formatted = sprintf "%.2f", $num;    # 先設定好格式  
print $formatted;
```

當然，sprintf的功能相當的豐富，如果你打算使用的話，應該先來看看sprintf提供甚麼樣的強大功能：

%%	百分比符號
%c	字元
%s	字串
%d	包含正負號的十進位整數
%u	不包含正負號的十進位整數
%o	不包含正負號的八進位整數
%x	不包含正負號的十六進位整數
%e	以科學符號表示的浮點數
%f	固定長度的十進位浮點數
%X	使用大寫表示的%x
%E	使用大寫表示的%E

其他還有一些不同的格式指定方式，當你開始使用的時候，你可以參考printf的說明文件。

### 12.4 排序

對於字串的另一個重頭戲，也就是排序了。因為當我們有了資料之後，要怎麼讓資料可以更容易的讓人可以進行檢索，或如何進行有效的整理就是非常重要的議題了，而排序正是這些議題的第一門課程。所謂的排序其實主要在進行的也就是「比較」，「交換」的工作，因此我們可以先從Perl如何交換兩個變數的值來看起。

在傳統的方式，或其他程式語言目前的實作方式還是如此，也就是使用另一個變數來作為暫存的變數。例如我們如果想要把\$a跟\$b兩個變數裡面的值進行交換，那麼可能的作法也許會是這樣：

```
$tmp = $a;    # 先把$a的值放進暫存變數  
$a = $b;      # 把$b的值指定給$a  
$b = $tmp;
```

```
$b = $tmp;      # 從$tmp中取得$a原來的值，並指定給 $b
```

可是在Perl當中，我們就可以輕鬆一些了。我們如果要交換兩個變數的值，只需要使用這樣的方式就可以了：

```
($a, $b) = ($b, $a);
```

這樣看起來好像有點差距，可是又相差不大，部過一但變數夠多，你利用其他方式可能只會讓自己變得頭昏腦脹，不然你試著自己弄一個四個變數的狀況，然後用原來的方式寫寫看，我想總還是很難比這樣看起來更方便了吧：

```
($a, $b, $c, $d) = ($b, $c, $d, $a);
```

能夠輕鬆的交換變數內的值之後，我們如果利用排序的結果來決定是否要把兩個正在進行比較的變數值交換，那麼最後就可以完成整個串列的排序。如果你學過某些相關的內容，應該會覺得非常熟悉，這似乎是某種被稱為「泡沫排序法」的方式。當然，你可以使用其他在資料結構那堂課中所學的其他排序，好吧，不過暫時先忘了這些課本上的東西。我們先來看看最基本的排序方式：

```
sub my_sort {  
    my ($a, $b) = @_;  
    ($a, $b) = ($b, $a) if ($a > $b);  
    .....          # 繼續其他運算  
}
```

利用比較，交換的方式，我們似乎完成了一個簡單，可以用來排序的副常式。不過既然每次排序我們都需要這樣的東西，那麼Perl很顯然的，應該會有更簡易的方式。於是我們發現了一個新的運算符：<=>。有人稱這個符號為太空船符號，確實是有幾分像，那麼它有甚麼便利性呢？我們實際利用這個符號來進行排序吧。這裡還有一個很大的特點，當我們在進行比較時，通常會定義兩個變數來表示正在進行比較的值，很多時候我們都用\$a跟\$b來代表這兩個值。只不過如果每次我們都需要這兩個變數，那不是很累人嗎？Perl也非常體諒我們打字的辛苦，所以\$a跟\$b已經被設為Perl排序時的內建變數。意思也就是說，以後如果你在Perl中要進行排序，你不需要自己另外定義這兩個變數。

```
my @array = (6, 8, 24, 7, 12, 14);  
my @ordered = sort { $a <=> $b } @array;  
print @ordered;  
# 結果變成 6, 7, 8, 12, 14, 24
```

你可能會很好奇，這樣的方式難道不能直接用sort來作嗎？我們之前學過，直接使用sort這個函式來對陣列進行排序。所以現在的狀況應該可以使用同樣的方式來進行排序。那麼何不來試試呢？

好啊，這已經讓我快要一頭霧水了。因為上面的例子實在讓人很想改寫成這樣：

```
my @array = (6, 8, 24, 7, 12, 14);  
my @ordered = sort @array;  
print @ordered;
```

```
# 這次輸出 12, 14, 24, 6, 7, 8
```

聰明的你可能已經看出排列出來的結果了，沒錯，sort預設會使用字串排列的方式，這時候，我們應該先提示一下sort的語法：

```
sort SUBNAME LIST      # 你可以使用副常式
sort BLOCK LIST         # 或使用一個區塊
sort LIST               # 這是我們一開始說的方式
```

因此，如果你沒有指定區塊或是副常式，Perl預設會使用字串的方式去進行排序，也就是我們第二次看到的結果了。那麼如果我要強制Perl使用字串比對，或是針對字串進行比對，那應該怎麼寫呢？你可以參考另一個和<=>相對應的運算符，也就是'cmp'，這也就是比較的意思。讓我們直接來試試這樣的比較方式吧：

```
my @array = (6, 8, 24, 7, 12, 14);
my @ordered = sort { $a cmp $b } @array;
print @ordered;
# 這次還是輸出 12, 14, 24, 6, 7, 8
```

沒錯吧，果然和我們第二次只使用sort的結果是一樣的。特別要注意的就是'cmp'這個東西，如果你要進行字串的排序，可不能使用太空船符號。另外，我們還可以直接進行遞減的排序，而且非常簡單，我們直接利用第一個例子來試試吧：

```
my @array = (6, 8, 24, 7, 12, 14);
my @ordered = sort { $b <=> $a } @array;
print @ordered;
# 遞減排序： 24, 14, 12, 8, 7, 6
```

其實一但可以利用區塊或副常式來進行獨特的排序方式，我們可以玩出不少其他的花樣。例如你可以對雜湊進行排序，或是比對多個值來進行排序。其中雜湊的排序是非常常用的。尤其我們知道，雜湊的安排是依據系統計算出存取的最佳化方式，因此大多數的時候，我們拿到一個雜湊通常是沒有甚麼順序性。要能夠對於其中的鍵或值排序都是非常重要的，而透過sort的方式，我們就很容易做到了。

```
my %hash = (john, 24, mary, 28, david, 22);
my @order = sort { $hash{$a} <=> $hash{$b} } keys %hash;
print @order;                # 依序是 david john mary
```

雖然只有三行程式，不過我們還是應該來解釋一下其中到底發生了甚麼事，否則看起來實在讓人有點頭暈。第一行的問題應該不大，或者說如果你第一行看起來有點吃力，那你可能要先翻回去看看雜湊那一章，至少你應該要懂得怎麼定義一個雜湊，然後指定雜湊的鍵跟值。這裡所用的方式一點也不特別，我們只是用串列來賦值給一個雜湊。最複雜的應該是第二行（除非你覺得最後一行要印出一個陣列對你而言太過困難），我們先看等號左邊，那裡定義了一個陣列，因為我們希望可以得到一個依照雜湊值排序過的雜湊鍵陣列。這聽來好像不難，讓我們先想像一下，我們該怎麼取得這樣的陣列呢？

首先我們應該先拿到包含所有雜湊鍵的陣列，也就是利用keys這個函式取得的一個陣列。拿到這個陣列之後，我們就可以來進行排序了。排序的重點在於區塊內的那一小段程式。我們還是使用了Perl預設的兩個變數，也就是\$a跟\$b，分別代表從陣列(keys %hash)拿出來準備比較的兩個數值。部過我們並不是直接對變數\$a，\$b進行比較，而是以他們為鍵，而取的雜湊值來進行排序。

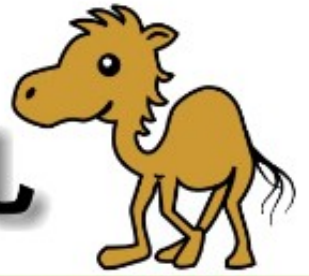




1. 讓使用者輸入字串，取得字串後算出該字串的長度，然後印出。
2. 利用printf做出貨幣輸出的表示法，例如：136700以\$136,700，26400以\$26,400表示。
3. 利用雜湊%hash = (john, 24, mary, 28, david, 22, paul, 28)進行排序，先依照雜湊的值排序，如果兩個元素的值相等，則依照鍵值進行字串排序。



# Perl 學習手札



There is more than one way to do it.

## 13 模組與套件

Perl之所以可以這麼受到歡迎，除了本身有許多專為懶人設計的語法以及相對於其他程式語言，更接近自然語言的用法之外，豐富的模組資源更是讓Perl能持續維持高人氣的主要因素。而數以千計的模組不但能吸引住眾多的Perl開發者，更能讓這些開發者貢獻出其他的模組，如此一來，便會造成「網絡效應」，而持續讓更多人願意投入Perl的懷抱。

對於Perl的使用者來說，如果你不會使用各式各樣的模組，那麼你對Perl的使用率可能不到十分之一。因此能夠寫Perl的人，可能也因此對於程式碼重用的部份對於其他程式語言的程式設計師有更深的感受。當然，大多數的Perl程式設計師總是需要學會如何開始使用模組，緊接著便會了解如果善用模組，找到自己所需要的資源。再下一階段就是如何寫出自己的模組。

可惜很多程式設計師，或是專案管理員對於這方面並不重視，他們總是只看著手邊的東西。而不肯多花時間把手邊的程式碼整理成模組，很多人不相信自己還會重新用到這些程式碼，或者不認為同樣的這些程式碼如果整理成模組，可以讓許多人節省更多的時間。當然，對於這些程式設計師或管理專案的人而言，更不用提怎麼進行好一個專案，版本控制，分支，合併了。（註一）

不論如何，一但你開始使用Perl，你應該就必須有足夠的能力去使用各式各樣的模組。而且還必須了解模組與套件的結構，因為你可能會需要對於你使用的Perl模組進行除錯的工作，雖然這些事情未必經常發生。不過你從這裡開始，就會開始慢慢的學會如何寫好自己的模組。所以現在就開始來進入的世界吧！

### 13.1 關於程式的重用

我們之前提到過可以節省程式碼寫作的時間，大幅提昇程式可重用性的方式就是副常式。可是如果你沒有好好管理你的程式碼，等到下一次你需要同樣的函式時，你還是必須重寫一次。當然，很多人這時候就會利用複製，貼上的方法。把原來的副常式複製到新的程式之中，這樣一來，就可以再度使用同樣的副常式了。

可是利用這樣的方式還是會有一些問題存在，就像我們在描述副常式時所說的，當你一再使用複製，貼上這樣的方式時，很容易就會造成管理上的問題。因為你還是沒有辦法統一的管理一個套件，讓你以後只要修改模組，接著就可以一次修正使用相同模組的所有程式。

而不會像你使用拷貝，貼上的方式，你一但找到副常式的一個錯誤，就必須同時修正所有使用這個副常式的程式，當然還可以因為你忘了某個程式中忘了修改而讓自己踩到地雷。所以既然你都已經使用了副常式，除非你確定某些副常式只會在目前的程式使用，否則他們都有機會成為模組。

另外，當你開始使用獨立的模組之後，你更需要做好檔案的管理，因為你可能會公開給所有的人使用，就像CPAN (Comprehensive Perl Archives Network, Perl綜合典藏網) 上面所有的模組一樣，或是開放給公司內部使用。不管如何，你的程式一但公開釋出之後，就應該考慮使用者的使用性以及如何更新版本，修正錯誤的問題。這其實是非常嚴肅的問題，因為一但沒有辦法做好程式碼的管理，很容易會加重負擔，反而增加管理成本。這個部份對於部份許多公司或個人來說，都還需要更深的著墨，我們應該在附錄用一些篇幅，介紹這個部份，雖然它們並不屬於Perl的範圍之內。

### 13.2 你該知道的 CPAN

也許你不太同意我們剛剛所說的部份，不過如果你確定你要開始使用Perl來解決生活，或工作上的問題時，你大概很難不先知道，而且學會如何使用CPAN。

剛剛說了，CPAN就是Perl典藏網，可是葫蘆裡到底賣的是甚麼藥呢？其實CPAN上主要的就是上面的許多模組，目前已經有好幾千個模組在CPAN上。所以你可以在上面取得這些模組的原始碼，文件，有些模組可能還提供其他的二進位檔案，讓你在沒有辦法編譯時也可以使用。

目前的CPAN，上面已經充滿了各式各樣的模組，大部份的需求幾乎都可以有現成的模組來解決你的問題，或七成以上的問題。當然，如果你想要在CPAN上找到符合需求的模組確實需要花點功夫，因為浩瀚CPAN大海，既然有各種各樣的模組，雖然你可以使用搜尋的功能，可是如果你完全沒有頭緒，恐怕是需要一點時間來適應CPAN這個圖書館了。不過接下來的章節，我們會在各個部份介紹相關使用的模組，這些模組很大部份的時候幾乎都是你在寫相關程式時一定會用到的模組。另外，我們也會在附錄整理五十個在CPAN上非常有用的模組，這個清單將會包括很多領域的部份模組，相信可以作為一個參考清單及介紹。

接下來，也許你終於花了一些時間找到了一個符合你需求的模組，那麼你在開始使用之前，必須先安裝這個

模組。最傳統的方式，你可以下載這個模組的原始碼，然後試著自己編譯。這時候，你可以先到CPAN網站上搜尋你需要的模組。(圖一)，然後下載原始檔，接著就開始編譯，像這樣。(圖二)

不過這樣確實有點辛苦，尤其Perl的使用上，我們會經常大量的安裝模組，如果每次都要這樣子來一步一步來就會顯得相當吃力。因此我們必然需要更方便的工作來幫助我們完成安裝模組的工具，而在你安裝完Perl之後，其實Perl就會給你一個叫做cpan的工具程式，而他正是幫助你完成大量安裝Perl模組的好幫手。很多時候，你可以直接進入cpan的命令列中。(圖三)

不過如果你在Win32的作業平台上，CPAN對你的幫助顯然就小的多了，尤其當大多數的Windows使用者並沒有安裝相關可以提供編譯這些模組的編譯器，那麼他們所需要的，應該是能夠提供Windows平台上的二進位檔安裝了。當然，這時候你應該使用由ActivePerl所提供的ppm程式，以便讓你可以容易的安裝Perl模組。cpan是目前的Perl版本內附的CPAN模組安裝程式工具，不過下一個階段的取代性程式也正在發展當中，而且目前也已經相當穩定，不久之後將會取代cpan，成為Perl內附的工具程式，這個新的工具就是cpanplus。就像這個模組的名稱，他正是cpan的加強版，例如他可以幫你確認目前機器上安裝的模組版本，以及該模組的最新釋出版本，提醒你該升級。你可以輕鬆的解除某個模組的安裝或是下載某個模組，解除安裝等等。(圖四)

很可惜，截至目前為止，cpanplus還是只能在Unix的環境下執行，主要的問題還是因為編譯器的問題，就如我們剛剛說的，絕大多數的Windows系統中並沒有編譯器，所以如果你的Windows環境下能夠裝起合適的編譯器，當然還是可以使用這些方便的工具。另外，Mac OS X的系統預設也並沒有安裝編譯器，所以如果你希望使用cpan/cpanplus的話，就必須安裝相關的套件。

### 13.3 使用CPAN與CPANPLUS

如果你只是想要學習Perl的語法，那麼也許你不需要使用CPAN，不過當你要開始利用Perl來完成某些工作，或作業。而且完全不想自己重新開始，那麼學會使用CPAN/CPANPLUS就是一件非常重要的工作了。正如我們所說的，如果你在Linux/\*BSD的作業系統中，一般而言，你一但裝好了perl，核心安裝也會自動把CPAN這麼模組安裝進去。所以也就有命令列的執行程式"cpan"，如果你是第一次執行cpan，會需要作一些設定，以確定你電腦內的環境以及各種程式的位置。完成設定之後，你會看到提示符號，就像這樣：

```
cpan>
```

這就表示你可以開始使用cpan了。

當然，你可以利用help取得完整的cpan使用說明，不過我們還是就一些常用的功能進行介紹。最常用的大概就是install了，幾乎還無疑問，你可以利用install來安裝需要的模組。所以如果你想安裝CPANPLUS這個模組，就只需要這麼作：

```
cpan>install CPANPLUS
```

利用cpan安裝模組，它會幫你進行完整的步驟，也就是一般我們手動從原始碼安裝時會進行的步驟：

```
perl Makefile.PL
make
make test
make install
```

所以有時候你會在使用cpan安裝的過程中遇到測試不過或其他狀況，這時候你可以使用強迫安裝的方式來要求cpan進行強制安裝。使用的方式也非常簡單，就只要在install時加上force的選項：

```
cpan>force install CPANPLUS
```

另外，如果你只想下載某個模組，而不想進行編譯或安裝，那麼就使用get指令。

```
cpan>get CPANPLUS
```

類似的指令則有make, test, install, clean等等，這些都是針對某個模組進行安裝相關的動作。而如果你想查詢某個模組的相關資料，你可以使用i這個指令，就像這樣：

```
cpan> i CPANPLUS
Strange distribution name [CPANPLUS]
Module id = CPANPLUS
  CPAN_USERID  AUTRIJUS (Autrijus Tang <autrijus@autrijus.org>)
  CPAN_VERSION 0.049
  CPAN_FILE     A/AU/AUTRIJUS/CPANPLUS-0.049.tar.gz
  MANPAGE      CPANPLUS - Command-line access to the CPAN interface
  INST_FILE    /usr/local/lib/perl5/site_perl/5.8.4/CPANPLUS.pm
  INST_VERSION 0.049
```

我們可以知道模組的名稱，版本，作者等等各種資訊。另外，i也可以使用正規表示式，所以如果你使用

```
cpan>i /CPANPLUS/
```

就會傳回一串內容有關CPANPLUS的模組。不過因為使用i這個指令會傳回所有關於模組，作者，散佈或集結而成的模組(例如Bundle::CPAN)等等資訊。而如果你想單純的搜尋其中一個部份，就可以使用作者(a)，模組(m)，散佈(m)跟集結(b)。我們繼續用CPANPLUS作例子：

```
cpan> d /CPANPLUS/
Distribution  A/AU/AUTRIJUS/CPANPLUS-0.049.tar.gz
Distribution  B/BD/BDULFER/CPANPLUS-Shell-Tk-0.02.tar.gz
Distribution  K/KA/KANE/CPANPLUS-0.042.tar.gz
Distribution  M/MA/MARCUS/CPANPLUS-Shell-Curses-0.06.tar.gz
4 items found
```

這樣應該清楚多了，這表示我們只要搜尋內容有CPANPLUS的相關散佈檔案，而不想要包山包海的把所有相關資訊都收集起來，當然你也可以只使用m或a來取得相關的資訊。

接下來，還有一個你也許會常用到的功能，也就是"reload index"。CPAN上的模組幾乎無時無刻不在更新，所以你的電腦裡面的各種資料其實會需要經常更新，這時候你只需要利用這樣的方式，CPAN會自動取網路上找到最新的內容索引：

```
cpan> reload index
Fetching with LWP:
  ftp://ftp.perl.org/pub/CPAN/authors/01mailrc.txt.gz
Going to read /Users/hcchi/en/.cpan/sources/authors/01mailrc.txt.gz
Fetching with LWP:
  ftp://ftp.perl.org/pub/CPAN/modules/02packages.details.txt.gz
Going to read /Users/hcchi/en/.cpan/sources/modules/02packages.details.txt.gz
```



```
Database was generated on Tue, 11 May 2004 09:34:08 GMT
Fetching with LWP:
ftp://ftp.perl.org/pub/CPAN/modules/03modlist.data.gz
Going to read /Users/hcchi en/.cpan/sources/modules/03modlist.data.gz
Going to write /Users/hcchi en/.cpan/Metadata
```

最後，如果你要離開，就請使用quit，CPAN會移除某些暫存檔，讓你平安的回到地面。CPAN在perl的使用上確實是非常方便的，不過有些地方還是讓人有點感覺不夠，例如你如果順手想要安裝CPANPLUS，那麼你可能需要進入cpan的命令列下，或是利用Perl的單行模式執行這樣的指令：

```
>perl -MCPAN -e"install CPANPLUS"
```

這倒還好，雖然指令長了一點，不過總是一次可以解決。只是我們都知道，因為在Perl中使用其他各式各樣的模組是縮短開發時程跟降低成本的好方法，所以大部份模組其實都還是用了其他模組，我們就說這個要被安裝的模組必須依賴其他某些模組，可是在CPAN裡卻沒辦法幫我們完成這些相關性的安裝工作。所以如果你安裝某一個模組卻發現它必須依賴其他模組時，CPAN會發出錯誤訊息給你，然後就停擺了。當然在我們的期望中，如果它可以「順便」幫我們把其他需要的模組也安裝進去，那顯然會減少許多手動的工作。因此在這樣的需求下，CPANPLUS也就因應而生了。

CPANPLUS在使用上有一些不同於CPAN的地方，例如你可以直接在shell下面執行CPANPLUS的安裝手續，就只要這麼打：

```
>cpanp -i SVK
```

接著，神奇的事情就要發生了，當我們安裝一個模組，而它所依賴的其他模組並不存在時，系統就會自動詢問使用者是不是要同時安裝相關的模組，就像這樣：

```
[root@Apple]# cpanp -i IO::All
CPANPLUS: :Shell::Default -- CPAN exploration and modules installation (v0.03)
*** Please report bugs to <cpanplus-bugs@lists.sourceforge.net>.
*** Using CPANPLUS: :Backend v0.049.  ReadLine support suppressed in batch mode.

Installing: IO::All
Warning: prerequisite Spiffy 0.16 not found. We have 0.15.
Checking if your kit is complete...
Looks good
Writing Makefile for IO::All

Spiffy is a required module for this install.
Would you like me to install it? [Y/n]:
```

接下來，CPANPLUS也有自己的終端機，你只需要用"cpanp"就可以進入：

```
[root@Apple]# cpanp
CPANPLUS: :Shell::Default -- CPAN exploration and modules installation (v0.03)
*** Please report bugs to <cpanplus-bugs@lists.sourceforge.net>.
*** Using CPANPLUS: :Backend v0.049.
*** ReadLine support available (try 'i Term::ReadLine::Perl').
```



CPANPLUS還有一個非常好用的功能，就是列出目前系統中還沒更新的模組清單，所以你只要進入cpanp，然後使用"o"就可以得到系統中需要更新的模組，不過這通常需要花費一段時間：

```
CPAN Terminal > o
```

1	3.04	3.05	CGI	LDS
2	1.06	1.08	Digest	GAAS
3		1.05	Encode: : CN: : HZ	DANKOGAI
4	0.56	0.59	ExtUtils: : AutoInstall	AUTRIJUS
.....				
.....				

另外，還有跟CPAN比較不同的部份在於你使用CPANPLUS可以解安裝某個模組，也就是使用"u"這個選項，也就是代表uninstall的意思。你還可以利用cpanp進行本地端的perl模組管理，例如使用e來新增某些目錄到你的@INC中。至於在cpan中使用reload index的工作，在cpanp中只需要按下x就可以了。

其實在不久之後，CPANPLUS將取代CPAN在Perl核心中的地位，因此現在開始熟悉CPANPLUS似乎也不是甚麼壞事。

### 13.4 使用模組

看了一堆長篇大論，我們終於要開始寫程式了，或是你根本直接跳過前面的敘述來到這裡。不管如何，我們假設你已經學會怎麼裝模組了，而現在該來學習怎麼使用這些已經存在你硬碟中的模組了吧！

先來用一個簡單的模組吧，這個模組對於將來你在寫程式的除錯時會有相當的助益。就像下面這一段程式碼所寫的樣子：

```
use strict;
use Data::Dumper;          # 說明我們要使用的模組名稱

my %hash = ("john", 24, "mary", 28, "paul", 22, "alice", 19);
print Dumper(%hash);        # 這就是模組裡面提供的函式
```

當我們決定要使用某個模組時，我們就用關鍵字"use"，也就是「使用某某模組」的意思，還真是口語化。接下來，你就可以使用模組內提供的函式了。所以我們在接下來的地方，定義了一個雜湊，是包含了名字，以及他們的年紀。程式最後，我們用了Dumper這個函式來印出雜湊hash裡的所有內容，而Dumper其實就是Data::Dumper這個模組所提供的函式。

最基本，對於模組的使用大概就是這個樣子。當然，這也是最簡單的方式。在你使用use來指定你所要使用的模組時，Perl會載入模組，並且把模組匯入。而當你在使用use這個指令時，其實還可以指定匯入模組中的某些函式。例如我們找到一個模組，叫做Cwd，它主要的功能是可以幫助我們找到目前的路徑，如果你是一個Unix的使用者，那麼他就非常接近pwd這個Unix指令。這個模組提供了不少函式，不過我們有時候並不想全部用到，所以你雖然可以像原來的方式，這麼使用它：

```
use Cwd;

my $dir = cwd();

print $dir;          # 印出目前的路徑
```

另外一種方式則是在use後多加一個參數，用來表示要匯入的函式。就像這樣：

```
use Cwd qw(abs_path);          # 我只想用abs_path

my $dir = Cwd:abs_path;        # 這時候，要加上完整的模組名稱

print $dir;
```

在你使用模組的時候，你也許還要注意另外一件事，也就是Perl能不能正確的載入你的模組。大多數的時候，你會從CPAN安裝模組，這些狀況下其實並不會有太大的問題，因為系統都會幫你安排好你的模組所應該擺放的位置。可是如果你利用其他方式取模組，或準備安插自己的模組時，有時候卻會因為Perl找不到你指定的模組而無法進行載入。因為對於Perl來說，它有一個模組載入的路徑，而這些路徑其實是被紀錄在@INC底下的，所以如果你沒有安裝某個模組，或是你的模組並不在Perl的載入路徑內的話，那麼它就會告訴你無法找到這個模組。

因為@INC也是Perl內建的陣列變數，所以如果你要知道系統中的@INC，你也可以直接用這樣的方式印出來：

```
print "@INC\n";                # 別懷疑，只需要這樣
```

或者你直接在命令列底下使用perl -V也可以看到相關的內容。不過這樣會輸出非常多的內容，只怕在這裡列出來會佔去一大頁，所以還是由各位自己試試看吧。

當然，如果你的模組地處邊緣，沒有辦法被@INC含蓋進去的話，也不用擔心，你可以自己指定程式內使用的模組路徑。其中一種方式是直接修改@INC變數，只是這個部份牽扯到關於模組載入的時機，也就是如果Perl在編譯的時候無法找到指定的模組，它就會開始不高興，然後大聲哭鬧。所以你在程式碼執行的部份修改了@INC這個變數對於停止Perl的問題並沒有太大的幫助（註二）。既然解決方法不只一種（There is more than one way to do it），我們顯然可以試試其他辦法。有另外一個方式，也就是直接使用use指令，就像這樣：

```
use lib "/home/hcchi en/pm/";

use Personal;
```

其實你還是有其他各式各樣的方式，不過在這裡我想還是這樣就足夠了。至少這也是目前我遇過最常用的方式，如果某一天你覺得這樣的方式已經不敷使用，相信你已經有能力找到更多方式來幫助你解決問題了。

### 13.5 開始寫出你的套件

經過一番努力，你應該要慢慢熟悉怎麼使用cpan/cpanplus從CPAN安裝各式各樣的模組了（註三）。接下來，你應該蓋上這本書，然後開始寫Perl程式了。或是你已經可以自己寫一些程式，然後拿來解決一些日常生活中的問題，或工作上的需要。接下來，你已經準備把手上已經寫好的程式碼集結起來，先把它們集結出一個套件吧。

我們剛剛一直在討論怎麼使用CPAN上豐富的模組資源，不過現在我們應該回過頭來看看模組的組成元素。其中非常重要的一個部份，也就是套件。套件其實就是你在寫Perl程式時的零件箱，也就是你可以放進一堆可以重新使用的小零件，那麼在程式裡面，你就可以直接拿出來，兜起來，很快就可以寫好自己的程式。我們先來用個簡單的例子吧，雖然大多數的工作已經有了模組可以讓我們使用，可是要讓大家簡單明瞭的看懂套件的寫法，我們還是來看看下個這個程式吧：

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

```

use strict;

my @grades = (67, 73, 57, 44, 82, 79, 67, 88, 95, 70);

my $adv = adv(@grades);          # 叫用 adv 這個副常式
print $adv;

sub adv {
    my @input = @_;
    my $total;
    $total += $_ for (@input);    # 算總和
    $adv = $total / scalar(@input); # 求平均
}

```

這程式相當簡單，我們在主要的程式部份看到兩個重點，第一個就是定義一個陣列，在這裡我們定義了一個關於成績的陣列，裡面放滿了一堆學生的成績。第二個重點則是在程式裡面叫用了adv這個副常式。至於在副常式adv裡面，我們取得了主程式傳來的成績陣列，緊接著計算總和，然後算出平均。接下來，我們希望開放這個方便的副常式給其他程式使用，所以我們必須把它放進套件中，就像這樣：

```

package Personal;  # 套件的開始

sub adv {
    my @input = @_;
    my $total;
    $total += $_ for (@input);
    $adv = $total / scalar(@input);
}

1;                # 回傳一個真值

```

於是我把它儲存為另一個檔案，叫做Personal.pm，接下來我們就可以開始使用這個套件了。也就像我們之前所說的方式，我們還是使用use這個指令。所以原來的程式就會變成這樣：

```

use strict;
use Personal;          # 現在我們直接使用這個套件了

my @grades = (67, 73, 57, 44, 82, 79, 67, 88, 95, 70);

my $adv = Personal::adv(@grades);    # 然後呼叫套件的adv
print $adv;

```

乍看之下，好像不過就是把原來的程式切成兩半，實在一點也沒甚麼特殊的。不過事實當然絕非如此，現在我們假設又要寫另一個程式了，這次要算的是一大群人的平均年齡。沒錯，我們又需要用到算平均的函式了。所以我們寫了這樣的程式：

```

#!/usr/bin/perl -w

use strict;
use Personal;

my %member = ( 'john' => 22,

```

```
'mary' => 42,  
'paul' => 27,  
'alice' => 19,  
'joshua' => 37 );
```

```
my @age = values %member; # 取出所有人的年齡，放入陣列  
  
my $adv = Personal::adv(@age); # 還是使用了adv這個函式  
print $adv;
```

這樣的用法應該不難理解，我們現在有了自己的套件檔案，也就是Personal.pm。接下來，我們只要再度需要使用adv的部份，就只需要載入Personal.pm就可以。當然，使用者也可以自己不斷加入新的函式，來讓自己的函式庫越來越豐富。

一般來說，我們都以.pm來作為分辨一般Perl程式與套件的方式。而我們在套件的一開始，則是以關鍵字package來表明這個套件的名稱，就像我們剛剛寫的方式：

```
package Personal;
```

當我們開始使用套件的時候，其實套件內部就像是另一個獨立的程式一樣，你在使用一個套件時，並沒有辦法提供一個全域變數來供所有人使用。而且這當然是完全合理的邏輯，否則不是會讓人一團混亂嗎？不過在套件內部確實也有一些機制來保持一個專屬的空間，避免套件與套件之間，套件與程式之間，所有的變數名稱，副常式變成像一盤義大利麵一樣，全部打結混在一起。因此，我們在將自己的套件完成到某個程度之後，使用套件名稱作為檔名儲存起來，就可以開始使用了。以剛剛的套件為例，我們就把他儲存為Personal.pm。

Perl使用的方式就是所謂的「符號表 (symbol table)」。一般沒有被定義套件名稱的部份，其實都是被委以main這個套件，當然你也可以隨時使用package來定義套件名稱。套件的有效範圍是從你使用package宣告開始，一直到這個區塊的結束，或是另一個package的宣告。所以這也就是某個套件的有效命名空間，也就是說在這有效範圍內的變數命名都會是屬於這個套件的命名空間下。用簡單的表示方式就會是：

```
$PACKAGE: : $VARIABLE
```

這樣的方式其實就讓人比較可以理解為甚麼我們會使用類似Cwd::abs\_path這樣的方式來呼叫某個副常式。而且你也可能偶而會發現這樣的程式錯誤：

```
Undefined subroutine &main::adv called at ch3.pl line 14.
```

也就是Perl在main這個命名空間下並沒有找到adv這個副常式。

而就像很多情況看到的，我們可以使用包含的關係來使用套件的命名空間，所以你當然也可能看到類似這樣的方式：

```
$PACKAGE1: : $PACKAGE2: : $VARIABLE
```

套件與模組的使用對於大多數的程式寫作都是非常重要的議題，而對於Perl來說更是如此，因為他可以讓你大量減少程式寫作的時間。至少在看完這一章之後，你應該可以開始學習使用CPAN上廣大的資源。

習題：

1. 試著在你的Unix-like上的機器裝起CPANPLUS這個模組。
2. 還記得我們寫過階乘的副常式嗎？試著把它放入套件My.pm中，並且寫出一個程式呼叫，然後使用這個副常式。

註一：許多專案管理的方式就是採用拷貝，複製的方式，和我們所提的方式顯然大異其趣。

註二：我們確實可以在程式碼中要求Perl在編譯期間就修改變數@INC，不過我們不打算在這裡把這件事情搞的這麼複雜。

註三：千萬別小看這部份，如果沒有CPAN，你寫Perl會感覺太辛苦了。





# Perl 學習手札



There is more than one way to do it.

## 14 參照 (Reference)

對於一個剛開始使用Perl的使用者來說，要深切的了解參照確實是非常困難的一件事。可是如果不使用參照，你就會發現有很多時候，事情會變得相當複雜。因此大略的了解Perl參照的使用實在是相當重要。不過對於許多從事Perl教學的人來說，Perl的參照是既複雜又困難的事，因此在大約20小時的教學中，並不容易包含參照的使用。所以有些給Perl入門使用者的書籍也會避開這個題目，而在更進階的書籍再專文介紹。可是我以為如果失去了參照的使用，很多相當方便的使用方式都無法被實作，讓人無法領略Perl的方便性。可是要如何以簡單的方式概略的介紹Perl的參照也是另外一項複雜的議題。不過雖然如此，我們還是來嘗試以比較淺顯易懂的方式來介紹參照的常用方式。

### 14.1 何謂參照

就像C程式語言造成許多程式設計師的困擾一般，參照之於Perl也有類似的效果。當然造成這個狀況的原因大概也是因為參照的抽象觀念也足夠跟C的指標媲美。其實這也許只是危言聳聽，我們應該想辦法讓指標變得更容易使用，而且根據Perl的80/20定律，我們只需要學會其中的百分之二十，就可以應付百分之八十的狀況。

「參照」，其實跟所謂的指標在意義上是非常接近的，也就是某個變數指向另外一個變數。比如我們有一個陣列變數像是這樣：

```
my @array = (1...10);
```

那麼我希望使用一個純量變數\$ref來表示@array這個陣列時跟怎麼辦呢？這時候，我們只要表明它是一個陣列，而且它被儲存的位置在那裡。我們就可以順著這個線索找到@array這個陣列，而且得到他的內容。所以我們可以用這樣子來表示\$ref變數：

```
ARRAY(0x80a448)
```

這樣看起來就非常清楚，我們有一個陣列，位址在0x80a448。所以我們可以藉由這樣的資料取得@array這個陣列的詳細資料。這就跟我們在Unix系統下使用檔案連結的方式有點接近，我們透過某個連結資訊來找到被連結的檔案。

而在Perl裡面，所謂的參照，其實確實是建立了另外一個符號，就像儲存一般的變數一樣，不過這次我們得到的是一個純量變數。Perl確實可以使用純量變數來指向任何其他的資料結構，也就是另一個純量變數，陣列或雜湊，就像我們剛剛看到的樣子。

我想我們可以用一個比較簡單的方式來解釋參照，例如妳們家的成員可以組成一個資料結構的型態(你可以使用陣列或雜湊，看你想要儲存的資料內容而定)，於是如果想要用一個純量的資料型態來取得家裡成員的內容，那麼我們也許可以用門牌號碼來代表(當然，如果是有戶政單位的作業疏失造成門牌號碼重複等各種錯誤可不在我們討論的範圍內)。

### 14.2 取得參照

當然，你其實大可不必擔心怎麼找出參照的位址，因為這個部份可以由Perl代勞。我們來看看怎麼取得參照：

```
my @array = (1..10);
my $ref = \@array; # 取得陣列@array的參照

print $ref;
```

當然，這個程式的執行結果就會看起來像是我們剛剛寫的樣子，只是位址會有所差異。接下來，我們也可以來看看怎麼利用一個參照來取得被參照的資料結構內容。就以剛剛的例子來看，我們定義了一個陣列@array，然後利用反斜線(\)來取得@array的參照。那麼我們要怎麼取得\$ref所參照的@array內容呢？其實我們只需要這麼作就可以了：

```
print @$ref; # 利用參照找回陣列
```

很顯然的，我們可以利用純量變數來取得另一個純量變數，或是陣列，或是雜湊的參照。當然，取得的方式都是類似的，所以我們只要這樣：

```
$scalar = "1..10";
@array = (1..10);
%hash = (1..10);
$scalar_ref = \$scalar;
$array_ref = \@array;
$hash_ref = \%hash;
```

這樣看起來應該就清楚多了，不過很多人可能還是沒辦法想像這樣的參照能有甚麼很大的用途。其實參照很重要的用途之一，就是在增加資料結構的彈性，或者你也可以說它是增加資料結構的複雜性。我們先來看看一個實例：

```
@john = (86, 77, 82, 90);
@paul = (88, 70, 92, 65);
@may = (71, 64, 68, 78);
```

我們現在假設有一個考試，總共考了四個科目，上面的陣列是這三個學生每一科的成績。可是我們如果要針對某個科目，一次取得每個學生的成績就會顯得很麻煩，因為我們可能需要\$john[0], \$paul[0], \$mat[0]這樣的方式。因此在其他程式語言的實作方式則是使用所謂的「多維陣列」，例如你可以定義一個陣列，那麼它的結構會看起來這樣子：

```
grades[0][0] = 86;
grades[0][1] = 77;
grades[0][2] = 82;
.....
grades[2][2] = 68;
grades[2][3] = 78;
```

很可惜，Perl的陣列並不接受這種方式的定義，也就是並沒有提供所謂的多維陣列的概念。可是卻不能依此

推論Perl的資料結構太過簡陋，因為透過參照的方式可以讓Perl的三種資料結構都獲得最充分的運用。所以我們來看看該怎麼使用Perl的參照方式來實作多維陣列。  
首先我們還是有三個陣列，分別代表三個學生的各科成績，就像剛剛的例子一般。接下來的重點就是把這三個陣列的參照放進另外一個陣列中，就像這樣：

```
@grades = (\@j ohn, \@paul, \@may);
```

所以現在看來，@grades這個陣列中其實已經包含了@john，@paul，@may三個陣列了。這樣就可以實作出一個多維的陣列。各位應該可以想像整個陣列中夾雜著陣列參照的狀況，其實我們可以用圖來觀察，也許會比較容易理解。

<<圖一>>

這個圖裡面，我們其實並不是忠實的表達出資料在電腦記憶體中儲存的形式，不過在概念上卻可以清楚的看出利用參照來表達複雜資料結構的方法。也就是達到過去我們曾經在其他個種程式語言中所使用的多維陣列的方式。當然，在實際的使用上，如果我們想要達到過去利用其他語言做出來的多維陣列的，還是需要一點點小小的轉換，因為我們必須不只一次的在陣列中使用陣列參照，如此一來，如果要取出最後一層的陣列值就會讓人有點頭痛。而不像過去我們使用多維陣列的方式，如果有多維陣列，我們幾乎就只要這麼寫：

```
array[3][2][4];
```

既然如此，那麼為甚麼Perl不直接給我們多維陣列就好了呢？首先，在實際運用上，我們最常用的還是以一維跟二維陣列為主，所以利用參照的方式就可以容易的達成這個需求。其次，一但利用參照的方式，我們就可以使用更有彈性的資料結構，而不單只是一個多維陣列。聽起來好像非常神秘，不過其實仔細想想，確實很有道理。還記得嗎？我們說過陣列的元素其實就是一堆的純量所組成的，而且參照本身就是一個純量值，只是利用這個純量值，我們可以取得被參照的資料儲存在記憶體的位置。然後還有一個提示，也就是參照可以用來取得各種在Perl中原來就有的資料結構型式(註一)。說到這裡，也許你已經看出一些端倪，也就是利用參照的方式，你可以把大多數的資料都以純量的方式來表示，因此就有各式各樣的運用方式。下面就是一些我們可能會運用到的方式：

```
my @array = ('j ohn', ' paul', ' ken');  
my %info = ( 'date' => '3/27',  
            'people' => \@array,  
            'place' => '台北車站' );
```

這也許是某一次活動的資料，我們先取得了一個陣列，其中是參加活動的人員。接下來，我們會有或動的其他資料，例如活動的日期，地點。然後我們還希望把參加的人員也一起放入活動資料中，所以我們就使用了雜湊來儲存這些資料，可是雜湊中關於人員這個部份，我們是以陣列參照來表示。這就是另外一個非常典型使用參照來活化資料結構的例子。當然，如果你還有力氣，可以看看更複雜的例子，就像這樣：

```
my @j ohn_grades = (65, 87, 92, 77, 53);  
my %j ohn = ( id => '7821434',  
             birth => '1983/11/12',  
             grades => \@j ohn_grades );  
.....  
.....  
my %students = ( j ohn => \%j ohn,  
                .....
```

```
..... );
```

這個例子顯然可以好好來解釋一下，就像魔術一般，我們用了雜湊跟陣列的多次排列，把學生的資料全部堆在一起了。首先我們先拿到學生的成績，這是一個陣列，而且是最簡單的陣列，所有的成績依序排列在陣列中。接下來，我們要取得某個學生的資料，其中包括了他的個人成績。因此我們使用了一個雜湊來儲存學生的個人資料，而在成績的部份，則是使用了陣列參照。這樣子，我們可以完整的描述一個學生的資料，接下來我們只需要另外一個雜湊來收集所有學生的資料就可以，而在這裡，我們這個雜湊的每個鍵是學生的名字（在假設學生不會同名的狀況下），然後把剛剛取得的單一學生資料取參照，作為這個整合參照的值。這樣的寫法看起來確實複雜了許多，不過這是因為我們使用了逐步講解的方式，所以把整個過程的詳細的列出來。不過很多時候我們其實會使用匿名陣列或匿名雜湊的方式來表現。那麼就可以讓整個架構看起來容易，也清楚一些。我們把這種匿名陣列/雜湊寫法放在這裡，也許可以讓大家參考一下：

```
#!/usr/bin/perl

use strict;

my %students = ( john => { id => 'foo',           # 這是雜湊的第一對鍵值
                        tel => '11223344',
                        grades => [34, 56, 78]},
                paul => { id => 'bar',           # 第二對鍵值從這裡開始
                        tel => '223344',
                        grades => [44, 55, 66]},
                );
```

這樣的寫法顯然乾淨許多，雖然你可能還有點不習慣，不過可以確定的是，至少你不會再看到一大堆的變數名稱，而且每一個部份的相互關係也清楚了許多。當然，這時候你完全可以先忽略這個部份，不過為了讓你之後回過頭來看這一段程式碼時可以了解其中的奧妙之處，我們還是先說明一下這種方式的解讀方式。首先，我們定義了一個學生的雜湊，這也就是我們最終想要的資料處理方式。接下來，我們看到雜湊%students中包含了兩對的鍵值。其中第一對的鍵就是john，而其相對應的值則是一個雜湊參照。這時候出現了第一個匿名的雜湊，他包含了三對的鍵值，其中的鍵分別是id, tel, grades。而grades對應的值則是一個匿名陣列的參照，這個陣列一共有三個值，分別代表三個科目的成績。因此在雜湊%students的第一對鍵值中，我們包含了兩個參照，分別為一個雜湊參照與另一個陣列參照。接下來的第二對鍵值則是以paul為對應的鍵，並且包含著一個結構相同的值。好吧，你可以暫時先忘了這麼複雜的部份，至少你暫時應該可以使用最簡單的參照結構來實作一個二維陣列。

#### 14.3 參照的內容

我們剛剛已經學到利用各種方式得到某個資料型態的參照，並且可以把取得的參照值放入其他的資料型態內，組成其他比較複雜的資料儲存形式。可是接下來我們總會在程式當中取出這些值，因此該怎麼解開參照，讓他們指向原來所代表的那一個資料內容呢？我們先看看這樣一個簡單的參照：

```
my @array = qw/John Paul May/;           # 一個陣列
my $array_ref = \@array;                 # 取得這個陣列的參照
```

接下來，我們用大括號將參照包起來，並且恢復他應該有的資料型態代表符號，在這個例子中就是@號。所以看起來應該像是這樣子：



```
print @{$array_ref}; # 印出JohnPaul Mary
```

當然，我們也可以利用陣列的方式來取得某個索引的值，也就是這樣：

```
print ${array_ref}[1]; # 這樣就跟 $array[1] 一樣
```

看起來好像不太困難，那我們來依樣畫葫蘆，試試看雜湊參照的解法。當然，還是先建立一個雜湊吧，並且取得他的參照吧：

```
my %hash = qw/John 24 Paul 30 May 26/;
my $hash_ref = \%hash;
```

接下來，好像並不困難，我們只要把{\$hash\_ref}視為一個雜湊變數的名稱，所以要取得雜湊中，雜湊鍵為"John"的值就只需要這麼作：

```
print ${$hash_ref}{John}; # 果然印出 24
print ${$hash_ref}{Paul}; # 結果是 30
print ${$hash_ref}{may}; # 正如我們的期待，就是 26
```

當然，你也可以把%{\$hash\_ref}當成一個一般的雜湊來運作，所以你幾乎可以毫無疑問的這麼使用：

```
for (keys %{$hash_ref}) {
    print ${$hash_ref}{$_}. "\n"; # 印出 24, 26, 30
}
```

你是一個很簡單的例子，我們可以直接把%{\$hash\_ref}當成一般的雜湊來操作。所以一般使用於雜湊的函數也可以直接用於%{\$hash\_ref}上，相同的狀況，我們也可以在解開陣列參照之後，用相同的方式來操作。所以如果用剛剛的例子，我們也可以這麼寫：

```
my @array = qw/John Paul May/;
my $array_ref = \@array;
for (@{$array_ref}) {
    print "姓名：$_\n";
}
```

#### 14.4 利用參照進行二維陣列

我們在前面已經提過了利用參照來實作二維陣列的方式，可是為甚麼這一小節還要再重新解釋一次呢？主要是因為我們剛剛可以利用參照建立一個簡單的二維陣列，可是我們卻還不知道怎麼能靈活的操作這個陣列。而且利用參照來營造一個二維陣列是非常常見的參照使用方式，所以我們必須再詳細的逐步解釋二維陣列的建構，以及解構。最後並且引申出利用雜湊的值包含陣列參照的運作與利用。

如果你還不熟悉，我們先來建立一個二維陣列。我們假設這是一個日期與氣溫的對照，每天定時量測當地氣溫三次，分別紀錄於陣列中。所以我們以比較繁雜的手續建立起這樣的一個二維陣列：



```

my @d1 = (24.2, 26.3, 23.4);          # 每天的溫度
my @d2 = (23.5, 27.5, 22.6);
my @d3 = (25.2, 28.7, 24.8);
.....
.....
my @d30 = (19.8, 22.1, 19.2);

my @daily = (\@d1, \@d2, \@d3, ....., \@d30); # 當月每天的溫度

```

雖然複雜，不過終於把整個陣列建立起來了。目前我們已經有了30個陣列，各代表了第一天到第三十天中每天的溫度紀錄，接下來就是定義一個陣列包含了這三十個陣列的參照值，而這個陣列也就包含了這個月每天三次溫度的紀錄。於是我們可以利用參照的方式取得某一天的溫度，例如`$$daily[4][0]`就代表了第五天的第一次測量。這次你一定受不了了，這麼複雜的結構並沒有為程式設計師帶來比較舒適的環境，反正讓人徒增困擾。因為我們必須為每一天先建立一個陣列，然後再將陣列參照放入另一個陣列中，接著解參照，取出第二層陣列中的值。

很顯然，如果我們只有這種方式可以使用，那麼負責Perl設計與維護的那些黑客們一定自己先受不了。所以我們的另一個方式就是「匿名陣列」，「匿名雜湊」，而且這個作法我們剛剛已經稍微看過了。現在我們再來了解一下它們的用法。首先在賦值上，陣列所使用的是中括號`[]`，也就是當你在對陣列取值時的符號。而對於匿名雜湊，則是使用大括號`{}`，同樣的，也是利用你對雜湊取值時所用的方式類似。所以剛剛的例子如果重新定義陣列`@daily`就應該要寫成：

```
@daily = ([24.2, 26.3, 23.4], [23.5, 27.5, 22.6], [25.2, 28.7, 24.8]...);
```

看起來好像跟其他程式語言的方式比較接近了，可以取值應該怎麼作呢？還是要先解參照，然後取出陣列的某個值，然後再來解參照嗎？很慶幸的，這種複雜的工作實在不適合用來放在這種可能在日常生活中會大量使用的二維陣列中，因此我們也可以用很方便的方式來取得其中的值。所以要取值的方法就像這樣：

```
print $daily[2][1];
```

這樣真的清爽多了，如果你用過其他程式語言的二維陣列，其實大概也都是這樣的寫法。當然，你可以作的絕對不只二維陣列，你可以用同樣的方式來實作多維陣列，就像你可以很容易的造出一個三維陣列。

```
@demo = ([2, 4, 5], [3, 2], [2, 6, 7]), [4, 7, 2], [[1, 3, 5], [2, 4, 6]]);
```

現在回想起來，如果這個陣列要一個一個把名字定義出來，然後取它們的參照，放入其他陣列中.....，這實在太辛苦了。於是匿名陣列節省了我們不少的時間，當然，想必也降低了很多錯誤的機會。

## 11.5 陣列中的參照，參照中的陣列，陣列中的陣列

這個標題實在太繞口令了，雖然我們應該直接取標題為：「匿名雜湊與匿名陣列」，不過這樣的標題好像非常不容易平易近人，所以還是維持這個冗長的標題吧。

在上一節其實已經利用匿名陣列了，也就是我們用來實作二維陣列的輕鬆愉快版本。另外，我們也嘗試過在雜湊裡面放入陣列，可是既然我們可以方便的利用匿名陣列來進行多維陣列的實作，那麼利用類似的方式，把匿名雜湊，匿名陣列的交互使用，顯然可以讓整個資料結構更具有彈性。

還記得我們怎麼整理學生的資料嗎？那時候我們已經用了這樣結構的處理方式。學生的個人資料項目是一個匿名雜湊，而每個學生的成績則是由匿名陣列來組成的。因此我們就可以用簡單的方式來取出需要的值，所

以我們就可以這麼用：

```
print $students{john}{grades}[2];
```

這樣應該非常方便，你並不需要手動去解參照，或者進行甚麼繁雜的手續。而就像一般的陣列或參照的用法一樣，用中括號來取得陣列的值，或是用大括號才使用雜湊。而匿名雜湊也是常用的方式，它們可能被隱藏在陣列或雜湊中，就像我們剛剛看到學生資料的例子，就是一個「雜湊中的雜湊」實作的例子。

另外很常用的一種匿名雜湊方式則是陣列中的雜湊，很好的一個例子就是從資料庫擷取出來的資料，這時候我們常常會把每一筆資料依據欄位的名稱，跟所得值存放在雜湊中，然後將每筆這樣的雜湊存入陣列中所以一個陣列看起來會像是這個樣子：

```
@data = ( { 'column1' => 'data1',  
            'column2' => 'data2' },  
          { 'column1' => 'data3',  
            'column2' => 'data4' } );
```

如果你的資料儲存形式像是這個樣子，在陣列中放入匿名雜湊，那麼你如果要取出某個值，就只需要這麼寫：

```
print $data[1]{column2}; # 這樣你就可以得到data4
```

其實你也許不太習慣，為甚麼在使用匿名陣列，或匿名雜湊時，總會有不同於正常指定陣列或雜湊的方式呢？不過我們可以來看看這樣的狀況：

```
my @array = ((3, 5, 7, 9), (1, 4, 8, 6), (2, 5, 4, 2));
```

這時候，我們知道最外面一層是一個陣列，我們利用串列的方式指定了三個元素給這個陣列，而這三個元素卻都是串列，也就是說，我們希望把這三個串列放入陣列中。可是這時候問題就出現了，因為很明顯的，我們必須在最外層的陣列裡面定義三個變數，才能利用參照的方式把串列放入陣列裡，可是在一般使用的時候，不管陣列或參照，我們都可以使用串列的方式來賦值。像這樣的兩種形式其實都是可能的：

```
@temp = (3, 5, 7, 9);  
%temp = (3, 5, 7, 9);
```

所以如果我們利用剛剛的方式，希望把三個串列利用匿名陣列或匿名雜湊放進陣列@array的話，就會造成Perl的錯亂，因為它無法清楚的明白你所需要的是匿名的陣列或是雜湊。這也就是你必須清楚的表示你的需求，因此你如果希望使用匿名陣列或雜湊，就必須適當的分別清楚，所以依據你自己的需求，你就必須作不同的定義，就像這樣：

```
my @array = ([3, 5, 7, 9], [1, 4, 8, 6], [2, 5, 4, 2]);  
my @array = ({3, 5, 7, 9}, {1, 4, 8, 6}, {2, 5, 4, 2});
```

因為在Perl當中，你都是利用最簡單的陣列，雜湊的資料結構，配合上參照(當然還包括匿名陣列與雜湊)的方式，來組成更複雜的資料結構，例如多維陣列，或是陣列中的雜湊，雜湊中的陣列等等。也就因此，你可以有更大的彈性來玩弄各種結構的組成。比如你可以在陣列中的各個不同的元素裡，擺放不同資料結構的參照，所以你當然可以這麼作：

```
my @array = ({3, 5, 7, 9}, [1, 4, 8, 6], {2, 5, 4, 2});
```

所以，你對於這個變數的取值就有可能有：

```
print $array[0][2]; # 得到的結果是7
print $array[1][8]; # 這裡會印出6
```

其實參照的用法並不僅只於這些資料結構上的變化，你還可以取得副常式的參照，當然也可以使用匿名副常式的方式。就像你在使用陣列或雜湊的參照一般。參照的用法非常的靈活，而且運用非常的廣泛，Perl的物件導向寫法也是參照的運用。不過我們不希望剛入門的使用者被大量的參照困擾，所以等各位寫過一陣子的Perl之後可以再去參考其他的Perl文件，了解更多關於Perl參照的用法。

習題：

1. 下面程式中，%hash是一個雜湊變數，\$hash\_ref則是這個雜湊變數的參照。試著利用\$hash\_ref找出參照的所有鍵值。

```
%hash = ( name => 'John',
          age  => 24,
          cell phone => '0911111111' );
$hash_ref = \%hash;
```

2. 以下有一個雜湊，試著將第一題中的雜湊跟這個雜湊放入同一陣列@array\_hash中。

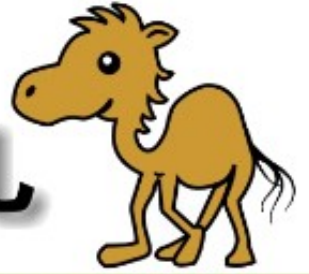
```
%hash1 = ( name => 'Paul',
           age  => 21,
           cell phone => '0922222222',
           birthday => '1982/3/21' );
```

3. 承上一題，印出陣列\$array\_hash中每個雜湊鍵為'birthday'的值，如果雜湊鍵不存在，就印出「不存在」來提醒使用者。

註一：其實不單只是這些資料可以取得參照，還有其他部份也可以使用參照來操作，不過我們並不在此討論。



# Perl 學習手札



There is more than one way to do it.

## 15. 關於資料庫的基本操作

如果讀者練習足夠認真的話，也許你已經對Perl可以慢慢的上手了。當然，你可能也準備用Perl來寫公司的一些小系統，或是個人日常生活使用的一些小程序。而大部份的時候，你都會希望能把存下一些資料，至少總不會每次要執行程式時，又要重新輸入這些相關的資訊。當然，很多資料也總是隨著程式不斷進行時被記錄下來的。這時候你可以利用不少不同的方式來儲存資料，例如你可以寫入檔案中。最簡單的方式應該也是如此，我們總是可以舉出這樣的例子。

```
sub savePhone {  
    my ($name, $phone) = shift;  
    open PHONE, ">>phone";  
    print PHONE "$name\t$phone";  
    close PHONE;  
}
```

這是一個很小，也很簡單，作為通訊錄的副常式。或者它根本算不上是一個通訊錄，只是紀錄姓名跟電話的相對應資料。而我們的作法也不過就是把得到的資料寫入檔案中，而這個檔案的名稱叫做"phone"。於是我們把姓名跟電話新增到檔案的最後，而且就只作這個動作，然後就把檔案關閉。如果我們持續新增朋友的電話，那麼檔案也就會不斷增長，也就長的像一個通訊錄了。

這樣的通訊錄有甚麼特色呢？特色就是它會長長的一串，如果你交友廣闊，也許這樣的檔案可以讓你印出好幾頁。當然，如果你要搜尋也不是太容易，要排序也是要自己處理。所以歸納出一個特色，這樣的通訊錄純粹只是作為「紀錄」，實在無法拿來運用。不相信的話，你可以想像一下，如果你現在想紀錄的不只姓名跟電話，你還想把眾多朋友的地址，Email全部記下來，那麼你會發現用檔案真是非常辛苦。何況如果你想在用檔案儲存這些資料的同時還可以方便的搜尋，刪除，排序等等，那麼你大概很快就會放棄這樣的想法。當然，如果你只是想要簡單的記下一些資料，那麼單純的使用檔案也會讓你自已輕鬆一點，不過如果你打算開始利用Perl幫你處理一些稍微複雜一點的資料時，那麼你應該好好考慮使用資料庫的形式。

### 15.1 DBM

一種非常簡單的資料庫形式，而且在你安裝Perl之後，你也就同時擁有這樣的資料庫系統，也就是「DBM檔案」。不過雖然同樣都是「DBM檔案」，運作的方式卻根據所在的環境而不盡然完全相同。所以如果對於DBM的運作方式有興趣的人也許就得自己去翻翻其他資料了。

不過就像本章大多數的部份，我們會盡量讓大家在比較沒有負擔的狀況下使用資料庫。畢竟在這個時候，很多技巧與程式的語法似乎都是使用比自己動手開始寫要重要許多。

#### 15.1.1 與DBM連繫

「DBM檔案」的資料庫非常有趣，它是利用特殊的雜湊來存取資料庫，或說達成資料庫的形式。所以它會利用雜湊跟所謂的DBM進行緊密的結合。這從我們開始操作DBM就可以看出來，所以我們先來開啟一個DBM檔案，試試怎麼使用DBM。

```
dbmopen (%HASH, "dbmfile", 0666) or  
die "檔案打不開!";
```



看起來是不是有點面熟？其實跟開檔案的方式確實有點接近。不過我們所指定的並不是檔案代號，而是一個雜湊，用來聯繫DBM檔案，這樣的方式可以讓你在操作雜湊時就等於在操作DBM檔案一般，也就是讓你用簡單的方式對雜湊進行改變時，Perl會直接幫你反應到檔案中。

其實當我們使用了dbmopen時，Perl也會自動幫我們建立起相對的資料庫檔案，可是我們在寫程式所對應的還是只有雜湊，這樣想像起來好像非常的輕鬆。即使Perl已經幫我們在系統中建立了兩個實體檔案，我們卻並不需要理會，繼續使用方便的雜湊吧。

當然，這個雜湊就像我們平常所看到的一樣，所以你也要遵守雜湊的命名規則。不過很多時候，程式設計師總會有一些慣用法，就像檔案代號大多使用全部大寫的方式，我們也習慣使用全部大寫的雜湊代號來代表繫結DBM檔案的雜湊。而且請各位務必記住，所謂的繫結是指在目前的程式當中，一但我們利用dbmopen這個指令時，Perl會讓我們以雜湊的方式來對DBM檔案運作，可是一但我們關閉這個繫結或是離開程式後，這樣的關係就不存在了。所以檔案內當然也不會有相關的雜湊名稱被記錄下來。

而關閉的方式則是只要使用像這樣的指令：

```
dbmclose(%HASH);
```

沒錯，還是跟你在關閉一個檔案一樣的方式。而且如果你沒有手動關閉這個DBM檔案，Perl也會在程式結束時自動將它關閉，不過你應該知道，這不會是一個好習慣的。

### 15.1.2 DBM檔案的操作

在開啟一個DBM檔案資料庫之後，我們就可以對它進行存取，也就是進行一般的操作。例如你可以修改資料庫的內容，新增資料，搜尋你要的資料等等。至少值得慶幸的是，所有的事情都可以利用雜湊來完成，那是大家都還算熟悉的東西。其實我們如果看個例子，應該很容易就可以上手了。

```
dbmopen (%HASH, "dbmfile", 0666) or
    die "檔案打不開!";
print $HASH{'John'} if (exists $HASH{'Hohn'});
$HASH{'Mary'} = '0227331122';

@sort_keys = sort {$a cmp $b} (keys %HASH);
for (@sort_keys) {
    print "$_: $HASH{$_}\n";
}

delete $HASH{'Paul'};
dbmclose(%HASH);
```

沒錯吧，除了第一行需要讓你的雜湊跟DBM檔案建立繫結以及最後一行關閉DBM檔案之外，你幾乎看不出來你正在對一個DBM資料庫進行操作。因此如果你打算要用DBM資料庫，那麼幾乎是可以非常容易的上手。不過其實使用DBM資料庫還是有一些比較深入的技巧，例如你可能要鎖定資料庫，以避免因為超過一個以上的行程在存取資料庫而產生問題。不過一開始使用，你暫時還不需要讓自己煩惱這麼多（否則只怕光擔心這些問題就讓人不敢使用了）。而這相關的問題，你可以在某些進階的Perl相關書籍中找到合適的作法與解答。

### 15.1.3 多重資料

如果你還記得雜湊的特性（希望你會記得，不然該怎麼使用DBM資料庫呢），那麼你應該記得雜湊其實是由一對的鍵值所構成的。也就是說，雜湊中每個獨特的鍵都會被對應到一個值上。而我們剛剛也利用了這個特性，紀錄了姓名與電話的對應關係。但是這時候卻有問題了，如果我希望記下的不只電話，或說除了家裡電話之外，我還希望記下好友的行動電話號碼，那麼在雜湊中好像就不適用了。

可是如果一次只能紀錄一個鍵跟一個值，那麼DBM資料庫還真是不實用，畢竟大多數的時候，我們總會需要使用一大堆的資料欄位。因此最好可以讓DBM資料庫可以讓一個鍵對應到多個欄位，這樣子我們就可以紀錄



行動電話，或是生日，地址等等其他資料了。

可是既然雜湊的特性沒辦法改變，那麼要怎麼樣可以把多個欄位擠在一個雜湊值中呢？很直覺的，我們當然可以直接把所有的資料「連」成一個超長的字串。所以你可能得到這樣的資料：

```
$personal_data = "02-27631122\t0931213987\t1974.12.3"
```

接著你利用split來把字串切成一個小陣列，讓你可以獨立使用每一個部份，就像這樣的方式：

```
@data = split /\t/, @personal_data;
```

這樣好像很容易，你可以利用簡單的方式來儲存多個欄位的資料。可是如果有其他更有效率的方式也許會更好，畢竟把所有東西全部胡亂的擠成一團似乎不是甚麼太好的方式。所以pack跟unpack這樣的函式似乎可以幫我們解決這類的問題，而且它們也正是為了這個理由而存在的。當然，有一部份的理由是在於把資料打包成為系統或網路的傳輸時使用。

pack的運作方式主要在於格式化我們的資料，也就是當我們在進行打包的時候，我們必須先定義出打包的方式。例如你要打包成位元組，整數等等不同的資料格式。所以Perl會根據我們所定義出來的格式，把資料打包成整齊的字串，以方便我們進行儲存或傳輸的工作。至於我們常用的格式字元包括c(字元)，s(整數)，l(長整數)，a(字串)，h(以低位元組開始的十六進位字串)，H(以高位元組開始的十六進位字串)，f(浮點數)，x(一個null的位元組)等等。當然，各式各樣的格式字元種類繁雜，各位如果有興趣可以利用perldoc 來檢查手上版本所有可用的格式字元集。我們先用簡單的例子來看看pack的用法。

```
my $data1 = length(pack("s l", 23, 6874192));  
my $data2 = length(pack("s l", 463, 66250921));  
print "$data1\t$data2";
```

結果我們發現兩個長度都是一樣的，沒錯，因為Perl依照我們的格式把資料打包起來了。也就是整數以兩個位元組，長整數以四個位元組來儲存。當我們取得被打包過的資料後，我們可以用unpack把已經打包起來的資料解開，以便取得原來的資料格式，簡單的寫法就像這樣：

```
my $pack = pack("s l", 23, 6874192);  
my ($short, $long) = unpack("s l", $pack);  
print "$short\t$long";
```

另外，你可以在格式字元後使用重複次數的方式，例如使用"cccc"來表示重複使用c四次，不過一般來說，我們都不會這麼用，因為使用c4可以達到同樣的效果，而且當然更簡潔易讀。只是當你使用數字來代表重複的次數時，有一個要注意的部份，因為有些格式字元的數字並不是用來表示重複的用法，例如你使用a6其實是表示長度為6的字串。當然，如果你的字串長度不足，Perl是會幫你補進NULL的。

其實使用pack/unpack這一組打包資料的函式的另一項好處就是你可以利用固定的長度來儲存某些特定的資料欄位。例如你訂好某些資料的欄位，接下來當你把打包過的資料存到檔案裡面，那麼你就可以保證每一比資料的長度都是相等的。以後當你要查某一筆資料的時候，你可以直接用seek的方式，迅速的找到資料所在的位置。

回頭來看我們的通訊錄吧！我們現在希望儲存不只一種欄位，所以使用pack來把各種需要的欄位打包在一起，當成雜湊的值存入DBM資料庫中。所以我們先使用pack來儲存電話跟行動電話兩個欄位吧，首先要先訂出需要的長度。假設他們各是長度為12的字串，所以我們應該這麼作：

```
my $packed = pack("a12 a12", $tel, $cellphone);
$HASH{'John'} = $packed;

my $data = $HASH{'Mary'};
my ($mary_tel, $mary_cell) = unpack("a12 a12", $data);
```

接下來，你可以試著用DBM作簡單的資料管理，而且可以使用多個欄位。

## 15.2 DB\_File

另外，使用Perl，你也可以簡單的使用雜湊的方式來存取Berkeley DB，而所需的模組DB\_File目前已經內附在Perl的核心當中，因此你裝完Perl之後，如果你的系統中已經有Berkeley DB的話，你就可以在程式中直接使用DB\_File來存取Berkeley DB。

使用DB\_File其實非常容易，和使用dbmopen非常類似，你只需要把資料庫檔案和雜湊建立起繫結，那麼你就可以直接使用雜湊來控制資料庫的檔案，不過實際的使用上，還是有著很大的差別。最簡單的使用方式，其實只需要這麼作：

```
$filename = "test";
tie %hash, "DB_File", $filename;
```

接下來，你就可以把建立起繫結(tie)的雜湊直接使用，就像一般雜湊一般。於是我們就直接使用%hash來存取，就像這樣：

```
$hash{'John'} = '27365124';
$hash{'Mary'} = '26421382';
```

沒錯，完全就只是雜湊的操作，感覺被騙了嗎？其實在操作上，最簡單的方式也就是這樣，跟你直接使用dbmopen沒有相差太遠。不過就使用上而言，DB\_File還是有些比較方便的地方，例如你可以在建立繫結的時候就指定檔案的權限。另外，你也可以控制相同雜湊鍵的處理方式。就像這樣的寫法：

```
$DB_BTREE->{'flags'} = R_DUP;    # 允許鍵值重複
$tie = tie %h, "DB_File", "test", O_RDWR, 0644, $DB_BTREE or die $!;
```

接下來，你就可以利用 \$tie 來進行一些操作。你可以使用

```
$tie->del($key);
$tie->put($key, $value);
$tie->get($key, $value);
```

等等方式來對資料庫進行方便的存取。

可是你慢慢會發現，這樣的資料庫雖然方便，可是有時候卻未必能夠滿足我們的需求。於是我們也許可以轉向求助於關聯式資料庫，雖然它的資料庫形式複雜不少，但是以功能來說，卻能夠讓你在處理大量資料時還能夠隨心所欲。

## 15.3 DBI

接下來，我們假設你已經知道的需求，你對於簡單的DB\_File，也就是Berkeley DB所提供的簡易資料庫再也無法滿足。你需要更強大的資料庫功能，最好是能應付各種複雜狀況的「關聯式資料庫」。你的資料需要大量的表格來儲存，表格和表格間也許還存在著交纏的連結與相關性。總之，至少在這時候，你應該要有「關聯式資料庫」的概念，當然也要可以自己獨立掌握資料庫的操作，因為我們並沒有打算在這裡教導大家怎麼使用資料庫。當然，目前在開放源碼社群經常使用的MySQL也是屬於「關聯式資料庫」，你也可以在網路上找到非常完善的相關文件與使用手冊等等。所以如果你對於這部份還不太熟悉，那麼可以先閱讀相關的資料後再回頭來看這一節，利用Perl連結到各種的關聯式資料庫。

不過在正式上路之前，我們還是必須來看看在Perl的使用上，整個DBI的概念。所謂的DBI，其實是Database Interface，所以其實對於DBI來說，它只是一個讓使用者可以降低成本的方式去控制各種資料庫，也就因此，它的使用必須搭配所謂的DBD，也就是Database driver。我們可以利用簡單的圖來表達DBI，DBD跟實際資料庫之間的關係。

<<圖一>>

從圖上可以看得出來，其實使用者所接觸到的部份幾乎只有DBI的部份，只不過在使用前必須根據使用者實際搭配的資料庫安裝DBD。當然，我們在這裡也就只針對DBI的部份介紹。可是因為DBI的使用其實是利用Perl物件導向的程式寫作方式來進行，而我們並沒有介紹物件導向的程式寫法，所以對於大多數的人來說也許有些困擾，不過一開始就讓初學者學習怎麼使用Perl的物件導向寫法似乎太過躁進。不過即使還沒開始自己動手寫物件導向程式前，當然也可以安心的使用已經現成的各式各樣物件導向模組。而且目前在CPAN上有不少模組都已經是以物件導向的方式開發，因此使用者也可以利用學習DBI的機會，順便也學學物件導向模組的使用。

一般來說，我們在開始使用一個物件時，都會先利用建構元來產生一個物件。有些程式語言也許是使用new這個「方法(method)」來達到進行這樣的工作，在Perl的物件導向語法中，雖然也有很多模組使用new來達成建構一個物件的目的，不過Perl本身的物件導向語法並不強制規定建構方式的關鍵字。所以你可以使用

```
$dbh = DBI->connect(...);
```

來建構出一個DBI物件，然後使用\$dbh來進行各式各樣該物件所提供的方法。

一開始，你顯然要先在你的程式使用DBI這麼模組，接著就是連接上你的資料庫，就像這樣：

```
use DBI;
```

```
my $dbh = DBI->connect("dbi:Pg:dbname=foo", "user", "passwd");
```

其中的Pg其實代表的是資料庫的驅動程式部份(Pg指的是PostgreSQL的驅動程式)，其實比較精準的用法應該是這樣：

```
$dbh = DBI->connect($data_source, $username, $auth, \%attr);
```

其中的\$data\_source也就是描寫相關的資料庫驅動以及所要連結的資料庫等資訊，當然，如果需要，你還必須把資料庫所在的主機位置也寫在這裡，就像這樣：

```
$dbh = DBI->connect("dbi:Pg:dbname=foo;host=db.host", "user", "password");
```

另外，由於DBI可以處理相關失敗時的錯誤狀況，如果連接資料庫失敗，它會傳回錯誤訊息：`$DBI::errstr`。所以我們在進行資料庫連結時，可以使用這樣的方式：

```
$dbh = DBI->connect("dbi:Pg:dbname=foo", $user, $passwd) or die $DBI::errstr;
```

有些時候，你可能不知道你的資料庫是不是已經安裝了專門給DBI使用的驅動程式，這時候你可以利用這樣的方式來取得目前機器上可以使用的所有驅動程式的陣列：

```
@available = DBI->available_drivers;
```

連接上資料庫之後，最簡單的方式就是直接執行你的SQL命令，所以我們要用最簡單的方式應該是這麼寫的：

```
my $sql = "DELETE FROM foo";
my $return = $dbh->do($sql);      # 傳回受到改變的資料筆數
```

這樣的方式是讓你執行SQL語法的最簡單方式，不過它的主要限制在於並沒有辦法傳回你從資料庫中選取的資料。因此大多數的時候，我們都會使用其他的方式來進行資料庫的select動作。一般的方式大概都會是這個樣子：

```
my $sql = "SELECT * FROM foo";
my $sth = $dbh->prepare($sql);
$sth->execute;
while (my @result = $sth->fetchrow_array) {
    print $result[0];
    # @result 回依序取得每筆資料的各欄位值
}
```

這樣是非常常見的寫法，主要就是用來從資料庫取出某些特定的資料。這是因為我們並不能直接使用do這個方法來執行select的語法，因為那並不會得到我們真正想要的內容。所以我們還是先寫好我們要取值的SQL語法，也就是"select \* from foo"這個敘述。接下來，我們要告訴\$dbh，處理我們所要的這個SQL敘述，並且建構出一個敘述的控制器(statement handler)，因為我們在進行select的時候，大多是透過執行某個select語法，接著一筆一筆取回執行的結果。而敘述控制器剛好就是為了這樣的需求而存在的。因此在利用prepare產生出控制器之後，我們就可以要求控制器執行我們的SQL敘述。緊接著利用while這個迴圈逐筆的把取得的資料當到其他的變數供我們使用。

而取回的方式最簡單的就是利用陣列的方式，也就是利用fetchrow\_array這個方法來一筆一筆，所以每當fetchrow\_array執行後有取得結果，就會讓while敘述成真，因此我們就可以從陣列@result中取得該筆的值。而陣列的元素則是依照資料庫select出來的結果來排序。

所以很多人並不喜歡這樣的用法，因為當你的資料表格欄位足夠多時，你會發現這種用法真的讓人很頭痛。你看看下面的例子吧：

```
my $sql = "SELECT a, b, c, d, e, f, g, h, i, j FROM data_table";
my $sth = $dbh->prepare($sql);
$sth->execute;
```



```
while (my @result = $sth->fetchrow_array) {
    ....
}
```

然後你希望取出欄位e跟i來作運算，於是你就要從0開始算，算出e是索引為4，i則是索引為8的欄位。這種事情真的是太辛苦了，當然，你的欄位數目也許還會更多，那時候你可能會想要找一片堅固一點的牆了。所以我們還是寧願使用雜湊的方式來取得資料庫送出來的值，也就是DBI提供的fetchrow\_hashref。至少使用雜湊參照的方式能夠讓使用者在取值時更為直覺。如果我們改寫剛剛的取值方式，應該可以這麼使用：

```
while (my $hash_ref = $sth->fetchrow_hashref) {
    print $hash_ref->{'e'};      # 你可以直接叫用欄位名稱
    print $hash_ref->{'i'};      # 這樣顯然愉快了
}
```

另外，有時候你還會需要更方便的形式來取出資料，其中常用的包括了fetchall\_arrayref跟fetchall\_hashref。其中的用法會是類似這樣的形式：

於是，你在也不需要手動計算某個欄位應該位於陣列的第幾個元素了，希望這樣能夠讓你心情愉快的寫程式。可是你又發現，很多時候，你只是打算先把全部的資料從資料庫擷取出來之後，再一次進行運算，還是乾脆包成一個陣列，丟給副常式去處理，那麼你可能會寫成這樣的方式：

```
$arrayref = $sth->fetchall_arrayref;
或是
$hashref = $sth->fetchall_hashref($key);
```

這兩種方法都會一次傳回使用者所選定的條件，只是傳回值儲存的方式有所差異。其中特別有趣的是fetchall\_hashref這個方法，使用者可以選定資料庫鍵值欄位，並依此來得到相關的其他欄位值。假如我們剛剛的眾多欄位中，欄位'a'是資料表的Primary Key，那麼我們就可以使用這種方式，取出特殊鍵值的那一筆資料：

```
$hashref = $sth->fetchall_hashref('a');
print $hashref->{'foo'}->{'c'};
```

這一行很顯然的，我們使用了fetchall\_hashref這個物件的方法，重要的是我們指定了主要鍵的欄位'a'。於是敘述控制器到資料庫選出了我們的所有資料，並且以主要鍵作為雜湊的鍵，而它的值則是其他個欄位鍵，值所形成的雜湊參照。因此我們要取得某個主要鍵值為'foo'這筆資料中，欄位'c'的值，就可是使用第二行的方式取得。

以上我們提到的大多是一般在進行一次的DBI操作時會使用到的方法，可是整個DBI的操作卻是相當複雜，提供的功能也非常強大，要詳細討論的話，都可以寫出一本書來。不過即使你沒打算買回DBI的書仔細鑽研，在你可以使用簡單的DBI操作之後，我們還是建議你看看DBI的官方文件，你可以直接使用perldoc DBI來閱讀。

你以為我忘了另外一個重要的部份了嗎？當然沒有。接下來我們要來看看在你對於資料庫的操作結束時，你應該要作的工作就是把它釋放，也就是利用disconnect的方法來釋出它所佔用的資源。雖然Perl會在程式結束時自動釋出還沒佔用的相關資源，可是我們還是強烈建議你，你應該在使用完DBI的資源後，手動將它關閉。因為你的程式也許會在系統運作一段時間，而且你卻只有某一段程式使用了這些資源，更有甚者，你也許一次開啟了多個資料庫連結，那麼適時的釋放這些資源顯然是程式寫作的好習慣。



```
$dbh->di sconnect;
```

## 15.4 DBIx::Password

相對於DBI的複雜程度，DBIx::Password只能算是協助使用DBI的一個小小工具。它也沒有提供類似DBI那樣魔術般的強大功能，簡單的說，它只是讓你用比較偷懶的方式來建構出一個資料庫控制物件，也就是所謂的database handler。別忘了，資料庫是相當重要的，我們經常要幫資料庫設定出繁雜的密碼，以避免遭人破解。可是這些密碼到底有多複雜呢？很多時候，連管理人員也總是搞不清楚這些密碼，再加上如果你的伺服器上還有不只一個資料庫，或是你的程式要連接多部資料庫，而且它們還分屬於不同的種類，那麼要寫個程式倒也相當辛苦。

針對這個需要考驗程式設計師記憶力的問題，DBIx::Password提出了解決的方式。也就是只要設定一次，那麼設定會被寫入Password.pm這個檔案之中。當你第一次安裝DBIx::Password這個模組時，它會問你一堆問題，其實也就是問你目前使用中的資料庫設定。而所有的資料將會像這樣的存在資料庫中：

```
my $virtual1 = {
  'dbix' => {
    'database' => 'db_name',    # 資料庫名稱
    'password' => 'passwd',      # 使用者密碼
    'attributes' => {},          # 連接資料庫的其他參數
    'port' => '',
    'username' => 'user',       # 使用者名稱
    'host' => 'local host',      # 主機
    'driver' => 'mysql',        # 資料庫類型
    'connect' => 'DBI:mysql:database=db_name;host=local host'
  },
};
```

那麼以後當你要連接資料庫時，就只需要使用虛擬的名稱了，原來建構資料庫控制器的方式也由DBI轉移倒DBIx。所以使用者再也不需要記一長串的資料庫連接需要的參數，因為所有的東西現在都由DBIx::Password負責了。讓生活快樂一點的寫法就變成了：

```
my $dbh = DBIx::Password->connect($user);
```

剛剛我們舉的例子中，就可以把\$user設為"dbix"，那麼DBIx::Password就會幫忙我們使用DBI進行資料庫的連結。而且在DBIx::Password中，你當然可以設定多組的參數，因為它只是利用雜湊存起你所有資料庫的相關資料。

雖然DBIx::Password還提供其他的一些操作方式，不過其實我們幾乎有超過百分之九十的機會都是使用connect這個物件方法，所以我們應該暫時可以先下課了。

### 習題：

1. 利用自己熟悉的資料庫系統(例如 MySQL 或 Postgres)，建立一個資料庫，並且利用DBI連上資料庫，取得Database Handler。
2. 試著建立以下的一個資料表格，並且利用Perl輸入資料如下：

資料表格：

```
name: varchar(24)
cell phone: varchar(12)
```

company: varchar(24)

title: varchar(12)

資料內容

[ name: 王小明

cell phone: 0911111111

company: 甲上資訊

title: 專案經理 ]

[ name: 李小華

cell phone: 0922222222

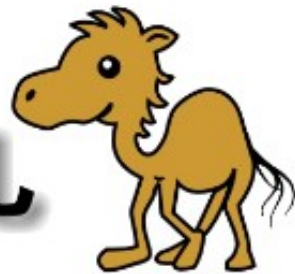
company: 乙下軟體

title: 業務經理 ]

3. 從資料庫中取出所有資料，並且利用fetchrow\_array的方式逐筆印出資料。
4. 呈上題，改利用fetchrow\_hashref進行同樣的工作。



# Perl 學習手札



There is more than one way to do it.

## 16. 用Perl實作網站程式

很多人開始接觸Perl都是因為把他拿來作為寫CGI(Common Gateway Interface)程式的工具，當然，也因此不少人都把Perl定位在「寫網站程式」。雖然事實絕非如此，不過用Perl來寫CGI也確實是非常方便的。尤其在不少前輩的努力之下，讓我們現在得以更方便的建立網路相關的程式。雖然目前已經有其他非常方便的腳本語言(scripting language)也可以讓人非常輕鬆的寫出CGI程式，例如像PHP或ASP一開始就是以最方便的嵌入HTML來進行互動式網頁作為主要目的(目前PHP已經把觸角延伸到其他方面，例如PHP-GTK)，可是Perl所依賴的不單單只是方便的CGI寫成模式，更重要的是程式語言本身所能達到的效果。

很多人在準備開始寫CGI的時候都會遇到類似的問題：「我應該學PHP或是Perl？」其實如果你大多數的時候只希望把Perl拿來寫網站，而且你手上的東西又非常的急迫，那麼PHP也許可以很快的讓你達到目的。雖然很多人可能不以為然，不過我倒以為，可以把學Perl當成純粹的學習一種程式語言，而CGI只是一種實作Perl GUI(圖形使用界面，Graphical User Interface)的方式。何況越來越多人把瀏覽器當成是GUI以及Client/Server架構最容易的達成方式，其中當然也因為使用瀏覽器作為用戶端程式可以降低使用平台的困擾，而Perl正是驗證了這種詮釋。

利用Perl來寫CGI程式的另外一個最大的疑慮大概是Perl的效率問題，這也許要從網站結構跟原理說起。一般來說，整個網站的原理並不算太困難，也就是用戶端發出一個需求，伺服器端收到之後，根據用戶的需求發出回應，然後關閉兩者的連線。而如果想要達到動態網站的目的，也就是根據使用者的需求，由伺服器端在收到需求之後，根據伺服器的設定把資料傳給後端的程式，接著程式依照需求產生出結果之後再傳回給網站伺服器。接著，網站伺服器就根據正常的流程把資料傳回給用戶端。我們可以從圖一看到比較清楚的流程。在正常的狀況下，Perl每次收到由網站伺服器傳來的需求時，就會重新開啟一個程序(process)，然後開始根據需求來產生結果。可是問題在於Perl在初始化的過程必須耗費相當的時間跟記憶體，因此當Perl完成初始化，然後產生出適當的結果並且回傳給網站伺服器。這樣的過程其實是相當漫長的，尤其當你的伺服器負載過大，或是硬體本身的效率不佳的時候，就會讓人感到非常的不耐。而這也經常是Perl作為CGI程式最讓人疑慮的部份。

當然，這部份也已經有解決的方案了。現在使用者可以利用FastCGI，或是在Apache中搭配mod\_perl使用，這樣一來就可以讓原來的程式被保留在記憶體中，而不必因為每次有使用者發出瀏覽的需求就必須重新啟動Perl，造成因為Perl初始化的延遲問題。不過如果想要有更好的效能來使用mod\_perl，那麼足夠的記憶體就變成非常重要的。所幸硬體的價格不斷下降，讓這樣的資源使用不至於發生太大的問題。

如果要使用Perl來增進網站的速度，是需要進行一些設定上的改變，當然有時候能夠跟程式配合是更理想的。不過這對於剛開始準備使用Perl來作為網站程式的工具而言顯然是有些困難的，所以我們並沒有要在這一章中介紹更多相關於mod\_perl的使用。待各位對於使用Perl來建構出網站這樣的工作熟悉之後，可以繼續選擇相關書籍或文件進行研究。

### 16.1 CGI

現在使用Perl作為網路應用程式的主要程式語言時，幾乎所有人第一個會遇到的就是CGI這個模組。CGI模組提供的功能非常的強大，不管是輸出至網頁上或是透過CGI取得使用者輸入的參數，另外也可以利用CGI動態產生HTML的各種表單選項。對於動態網頁的支援，CGI目前已經可以算是非常完善。甚至有時候會讓人感到相當意外，因為有些時候你會忽然發現，原來這些東西CGI.pm也可以達到。

現在我們就來看一下，如果要使用Perl開始寫CGI程式，那麼應該怎麼下手呢？毫無疑問，你總得先載入CGI這麼模組，所以就像我們所熟知的方式，先在你的程式加入這一行吧：

```
use CGI;
```

接下來，我們應該建立一個新的CGI物件，方法也不太難，也就是利用new這個物件的操作方法。所以只要這樣子就可以建立起CGI物件：

```
my $q = CGI->new;
```

如果你想要把任何內容輸出到網頁上，你大概都要先送出HTML的標頭檔到網頁上，也就是所謂的header。最簡單的方式當然就只要這麼寫：

```
print $q->header;
```

不過其實header有很多的參數，例如你應該要可以設定內容的編碼或輸出的內容型態等等。所以header這個函式也允許你使用相關的參數來改變header的屬性，例如你可以設定網頁輸出的內容編碼為UTF-8：

```
print $q->header(-charset => 'utf-8');
```

而且你可以一次指定多種屬性，就像這樣的方式：

```
print $q->header(-charset => 'utf-8',  
                -type => 'text/html');
```

當然，HTML的標頭檔還有各式各樣的屬性，使用者也都可以利用header來設定，而且這樣子的方式似乎也比起手動輸入各種header的設定值要簡潔不少，因此這個函式對於經常使用CGI的人來說還是非常方便的。而另一個重要的功能則是CGI的param，也就是利用CGI傳回來的參數。很多時候，我們都會利用這樣的方式來取得由HTML表單中傳回來的欄位值：

如果你的HTML裡面寫的是這個樣子：

```
<input type="text" name="user">
```

那麼你的Perl 就可以這麼使用：

```
$q->param('name');
```

也就是藉由HTML裡面的表單欄位名稱作為param的參數來取得使用者輸入的值，這樣就可以讓程式設計師很方便的取得使用輸入的結果。舉個簡單的例子吧！如果我們想製作一個使用者登入的程式，那麼畫面上大多不外乎就是使用者名稱、密碼兩個欄位。所以我們通常的作法就是利用取得的使用者名稱去資料庫搜尋相對應的密碼資料，如果沒有相關的資料就表示沒有這個使用者，或是使用者輸入的使用者的名稱有誤。如果可以找到相對應的使用者，那麼我們就比對資料庫中擷取出來的密碼與使用者輸入的是否一樣。而程式的寫法大概就像是這樣：

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

```

use strict;
use CGI;
use DBI;

my $q = CGI->new;
print $q->header;
my $user = $q->param('user');
my $dbh = ('DBD:mysql:database=foo', 'user', 'passwd');
my $sql = "SELECT password FROM table WHERE user = '$user'";
my $sth = $dbh->prepare($sql);
$sth->execute;
unless (my @pwd_check = $sth->fetchrow_array) {
    print "沒有這位使用者";
} else {
    unless ($pwd_check[0] eq $q->param('passwd')) {
        print "密碼錯誤";
    } else {
        .....
    }
}
}

```

有些時候，你會希望可以一次取得所有由HTML表單傳過來的欄位名稱，這時候CGI的另一個函式就可以派上用場了。也就是可以使用params這個函式，它會傳回一個陣列，這個陣列就包含了所有由HTML傳來的表單欄位。因此你的程式中可以這樣寫：

```
my @params = $q->params();
```

我們一開始就提到了關於利用CGI這麼模組送出動態header的方式，其實它不只能夠送出檔頭，你還可以動態的產生其他的網頁元素，例如你可以使用這樣的方式來印出字級為h1的文字內容：

```
print h1('這是h1級的字');
```

或是使用

```
print start_h1, "這是h1級的字", end_h1;
```

結果都可以產生

```
<h1>這是h1級的字</h1>
```

這樣的HTML內容

類似的用法還有包括下面幾種：

```

start_table() # 送出<table>這個標籤
end_table()   # 送出</table>這個標籤
start_ul()    # 送出<ul>
end_ul()      # 送出</ul>


```

當然，你還可能不直接送出header，因為你想要在處理完某些狀況之後，把網頁的位址送到其他地方。這時



候你就應該使用redirect這個函式。它的用法也是非常的簡單，你只需要告訴它你想轉往的其他網址就可以了。

```
$q->redirect('http://url.you.want/');
```

至於HTML的元素，也有很多時候你可以利用CGI這麼模組動態產生，例如你可以使用img來產生出這樣的HTML標籤。其他諸如ul，comment等等也都能夠輕易的被產生。不過另外一組非常完整的則是HTML表單的產生方式，也就是你可以利用CGI來產生大多數的表單欄位。一個非常基本的例子，就是利用CGI模組來產生一個text的表單欄位。

```
print $q->textfield( -name=>'field',  
                    -default=>'default value');
```

另外的一些欄位也都可以用類似的方式產生，就像接下來的例子：

```
print $query->textarea( -name => 'textarea',  
                      -default => '會放在欄位的預設值',  
                      -rows => 5,  
                      -columns => 10);  
print $query->password_field( -name => 'password',  
                             -value => '這裡也是預設值',  
print $query->checkbox_group( -name=>'checkbox',  
                          -values=>['value1', 'value2', 'value3', 'value4'],  
                          -default=>['value1', 'value3'],  
                          -linebreak=>'true',  
                          -labels=>\%labels);
```

當然，還有各式各樣的表單欄位，使用的方式也都大同小異，事實上，CGI模組的文件中有詳細的描述。不過有一個非常重要的卻是不可忽略的，也就是錯誤訊息。當使用者使用CGI進行一些運作時，有時候會有一些錯誤，這時候CGI模組會透過cgi\_error來傳回錯誤的訊息，所以你可以在程式中加上這個函式，讓CGI發生錯誤時能送出錯誤訊息。就像這樣的方式：

```
if ($q->cgi_error) {  
    print "無法處理CGI 程式";  
    print $q->cgi_error;  
}
```

接下來，我們再來講一個CGI模組的重要功能，也就是cookie的存取。許多時候，網站設計者為了減少使用者重複輸入資料的困擾，或是取得使用者瀏覽的紀錄，常常會藉由用戶端瀏覽器的cookie功能來紀錄一些相關性的資料。而CGI也可以針對這些cookie進行存取。如果我們想要取得已經存在用戶端瀏覽器上的cookie資料，我們只需要這麼作：

```
use CGI;  
my $q = CGI->new;  
my $cookie = $q->cookie('cookie_name');
```

接下來，我們當然也可能需要寫入cookie到使用者端，那麼我們可以利用CGI的檔頭來完成這項工作，也就是HTML的header。而完整的用法就是先把你所要寫入的cookie值，屬性都先設定好，然後直接用header來把這些內容送給使用者的瀏覽器中。

```
$cookie = $q->cookie( -name=>'cookie_name',  
                    -value=>'value',  
                    -expires=>'1h',  
                    -domain=>'.my.domain',  
                    -secure=>1);  
  
print $q->header( -cookie=>$cookie);
```

很顯然的，CGI的功能還不止於此，雖然我們已經介紹了大部份使用CGI時常用的功能。不過如果你還有進一步的需求，應該務必使用perldoc CGI來詳細閱讀CGI的相關文件。

## 16.2 Template

Template雖然不單單用於網路應用程式中，可是卻有許多人在寫網站相關的程式時總會大量的使用，因為對於能夠讓程式設計師單獨的處理程式而不需要擔心網頁的設計對於許多視設計為畏途的程式設計師而言，實在是非常重要。

Template其實完整的名稱是Template Toolkit，因為目前的Template Toolkit的版本是2.13，所以一般人又習慣稱呼目前的為TT2。至於正接受Perl基金會發展的則是Template::Toolkit的第三版，也就是俗稱的TT3。當然，既然可以接受Perl基金會的贊助開發新版Template::Toolkit，可見這個模組對於Perl社群的重要性了。

首先我們先來談談template系統的概念，讓大家能更深刻的感受使用template系統對程式設計師在網站程式的重要性。我們在剛剛可以看到CGI這個模組的運作，所以我們可以透過CHI的使用，輸出絕大多數的HTML標籤，也就是完成一個HTML頁面的輸出。當然，還有一種可能就是直接使用print指令，一個一個的把HTML標籤手動印出。可是不管是上面兩種方式選擇哪一種都會遇到相同的問題，也就是要怎麼跟網頁設計者一起合作來完成一個美觀，功能又強的網站呢？有一種可能的方式也許是由設計者做好一般的HTML頁面，然後你將這個頁面當作一般的文字檔案將它逐行讀入。然後自行置換掉要動態產生的部份。如此一來，只要網頁設計者能在網頁中加上讓你的程式可以認得的關鍵字，那麼你就可以不理會畫面的改變，而且還能夠讓程式正確的執行。這樣的概念正是孕育出模板系統的主要想法。

Perl的模板系統其實不只一種，可是Template Toolkit卻是深受許多人喜愛的，因為它不但可以讓使用者把模板與程式碼分隔，也可以讓程式設計師在模板中加上各式各樣的簡單控制。雖然說簡單，可是功能卻也一點也不含糊。因為使用者可以在裡面使用迴圈，可以取得由程式傳來的各種變數，當然也可以在模板中自己設定變數。當然，最後如果所有的方法都用盡，還不能達成你的要求，你也可以在模板裡面加上Perl的程式碼。

基本的使用TT2，你應該要有一個Perl的程式碼，跟相關的模板。而如果你只需要單純的變數替換，那麼使用的方式非常的簡單。你可以這麼寫：

```
use Template;  
  
my $config = {  
    INCLUDE_PATH => '/template/path',  
    EVAL_PERL    => 1,  
};  
  
my $template = Template->new($config);  
  
my $replace = "要放入模板的變數";  
my $vars = {  
    var => $replace,  
};
```

```
my $temp_file = 'template.html';
my $output;
$template->process($temp_file, $vars, $output)
    || die $template->error();

print $output;
```

這時候，你就必須有一個符合Perl程式碼中指定的模板檔案，也就是template.html。而在這個模板中，大多數都是一般的HTML標籤。而需要被置換的變數則被定義在\$vars中。我們可以先來看看這個template可能的形式。

```
<html>
  <head>
    <title>這是TT2示範</title>
  </head>
  <body>
    這裡可以替換變數 [% var %]
  </body>
</html>
```

好了，現在你可以把這個HTML拿去給網頁設計的人，讓他負責美化頁面，只要他在設計完頁面後，能把關鍵的標籤[% var %]留在適當的位置就可以了。不過你可不能輕鬆，讓我們來研究一下Template是怎麼運作的。首先，在我們新建立一個Template的物件時，我們必須設定好相關的內容，在這裡的例子中，我們只有設定了兩個參數，一個是INCLUDE\_PATH，也就是你的template檔案所放置的位置，這裡的內容可以是一個串列。也就是說，你可以指定不只一個路徑。另一個我們設定的是EVAL\_PERL這個選項是設定是否讓你的Template執行Perl的區塊。當然，選項還有好幾項，例如你可以設定POST\_CHOMP，這個選項跟chomp函式有一些類似，它可以幫你去除使用者參數的空白字元。另外，還有PRE\_PROCESS的選項則是設定所有的模板在被載入之後，都必須預先執行某個程序，例如先把檔頭輸出到模板中等等。另外，你還可以修改Template預設的標籤設定，例如我們剛剛看到的[% var %]，就是利用Template的預設標籤[% %]把它表現出來。而Template允許你在建立Template物件時使用START\_TAG跟END\_TAG來改變這樣的預設標籤。如果你用了這樣的方式：

```
my $template = Template->new({
    START_TAG => quotemeta('<?'),
    END_TAG   => quotemeta('>?'),
});
```

那麼剛剛在模板裡的變數就應該改寫成

```
<? var ?>
```

這樣似乎跟PHP有一點像了。

不過Template的作者也知道很多人大概很習慣PHP或是ASP的標籤，所以在Template也提供了另一個設定選項，也就是TAG\_STYLE。你可以設定成php(<? ... ?>)或是asp(<% ... %>)，不過其實我個人以為[% ... %]的原創形式還算順手，所以倒是沒換過任何其他標籤風格。Template的設定選項非常多樣化，不過只要了解以上的這些項目就大概都能應付百分之八十的情況了。如果還需要其他的資訊，則可以參閱Template::Manual::Config。

接下來，我們來看看使用上有甚麼需要注意的。在程式中，基本上你如果有甚麼特別需要注意的部份，那大概就是變數的傳遞了。如果你要傳一個純量變數，跟我們剛剛的範例一樣，那麼就只是把變數指定為雜湊變數的一個值。就像我們剛剛的用法一樣，你就只要指定：

```
my $vars = {  
    var => $replace  
};
```

可是很多時候，我們也許會傳送一整個陣列或是雜湊，那麼這時候你最方便的方式就是傳送這些變數的參照，寫法也許就像是這樣：

```
my @grades = (86, 54, 78, 66, 53, 92, 81);  
my $vars = {  
    $var => $replace,  
    $var2 => \@grades,  
};
```

這時候，你的模板內容顯然也需要改寫，把印出陣列的這部份加入你的模板中，一般來說，我們可以使用Template提供的FOREACH迴圈。所以你可以在你的template加上類似的一塊：

```
[% FOREACH grade = grades %]  
成績：[% grade %]  
[% END %]
```

當然，除了陣列，我們還可以使用雜湊。使用的方式卻也不太一樣，雖然你還是傳遞雜湊參照，可是在模板中的使用卻是直接利用雜湊鍵。所以你的Perl程式碼跟模板中分別是這麼寫的：

```
my %hash = ( height => 178,  
            weight => 67,  
            age => 28 );  
my $vars = { var => \%hash };
```

於是你必須在模板中做出相對應的修改：

```
[% var.height %]  
[% var.weight %]  
[% var.age %]
```

我們剛剛看到在模板中放了FOREACH這個Template提供的迴圈，其實模板中還有許多可供利用的特殊功能，例如一個非常方便的就是[% INCLUDE %]。很多時候，人們都喜歡在網頁的某個部份加上一些制式的內容，最常見的當然就是版權說明了。所以我們可以把這些版權說明的內容放到某一個檔案中，例如就叫做copyright.tt2吧！所以我們有了一個template檔案，內容其實就是一些文字敘述：

```
copyright.tt2:  
Copyright Hsin-Chang Chien 2004 - 2005
```

好了，現在我們有不少其他的模板檔案，希望每一頁都能加上這一段內容。那麼我們只需要在這些檔案的適當位置加上這樣的一行()所謂的適當位置就是你希望看到這些內容的位置：

```
[% INCLUDE copyright.tt2 %]
```

這樣的寫法可以讓你一次省去相當多的麻煩，尤其當你有可能更動這些檔案的內容時。例如我現在想把版權的內容進行調整，那麼你只需要修改copyright.tt2一個檔案，而不需要把所有的檔案一個一個叫出來修改。不過其實INCLUDE可以應付比較複雜的模板系統，而像版權聲明這種純文字的檔案，還有更簡潔的載入方法，也就是INSERT，如果你要被載入的檔案是一個純文字檔，不需要Template幫你進行任何的處理，那麼就考慮使用[% INSERT %]吧！

另外，Template還支援另一種常用的重複敘述，也就是WHILE。它的語法相當簡單，也就是使用這樣的區塊把要執行的內容標示出來：

```
[% WHILE condition %]  
....  
[% END %]
```

而IF敘述的基本用法也是和Perl語法幾乎一樣只是他是使用大寫字母，而且用Template的標籤區隔出來，所以你可以輕鬆的使用：

```
[% IF condition %]  
....  
[% END %]
```

或是

```
[% IF condition %]  
....  
[% ELSE %]  
....  
[% END %]
```

而ELSIF也是被允許的，用法也是類似：

```
[% IF condition %]  
....  
[% ELSIF condition2 %]  
....  
[% ELSE %]  
....  
[% END %]
```

當然，你還有UNLESS可以使用，用法也是相同：

```
[% UNLESS condition %]  
....  
[% END %]
```

至於如果你真的想在模板裡面寫Perl程式，也只需要這麼作：

```
[% PERL %]  
# perl 程式碼
```



```
....  
[% END %]
```

只是我個人並不建議你常常需要這麼使用，否則你有可能其實是挑錯工具了。因為Perl有其他模板系統也許比較符合你的習慣跟需求。而我們接下來就要介紹另一種在Perl社群中最近非常風行的另一套網路應用程式的搭配系統，也就是Mason。至於Template，它還有很多奧妙，你可以試著參考相關的官方文件。

### 16.3 Mason

網頁設計跟程式碼怎麼切割，這個想法會根據寫程式的人的習慣而有很大的差距。很多人喜歡利用像Template::Toolkit這樣的工具來讓網頁的版面跟程式碼分離的越乾淨越好。當然也有人認為像php形式的內嵌式作法可以讓網站的雛型在很短的時間就產生出來，因此Mason也就以類似的作法誕生了。因此在這一兩年來，Mason已經成為非常重要的模組，尤其在作為網站的工具時。根據job.perl.org(一個專門張貼Perl相關工作機會的網站)上的資訊，Mason已經是許多國外企業在徵求網站相關程式開發人員時需求度很高的技術了。而且許多大型網站現在也都使用了Mason來產生他們的網頁內容，例如亞馬遜書店(<http://www.amazon.com>)就是一例。

Mason基本的操作原理是在你的Apache中加上一個控制器(Handler)，讓使用者的要求全部送給Mason處理，這樣一來，Mason就可以把各式各樣的使用者需求都預先處理好，然後送出合適的內容。使用Mason，你除了裝上HTML::Mason這個模組之外，你的Apache還必須支援mod\_perl，在一切準備就緒之後，你可以在你的Apache中像這樣的進行設定：

```
PerlModule HTML::Mason::ApacheHandler  
  
<Location />  
    SetHandler perl-script  
    PerlHandler HTML::Mason::ApacheHandler  
</Location>
```

在Mason中，你可以使用百分比(%)符號作為Perl程式碼的引導符號，或是<% ... \$>。如果是一行的開始是由%引導，那麼表示這一行是Perl程式碼。或是利用<% ... %>來設定一個區塊的perl程式碼。所以你的網頁可以像這樣子：

```
<%perl>  
my $num = 1;  
my $sum = 0;  
while ($num <= 10) {  
    $sum+=$num;  
}  
</%perl>  
總和是： <% $sum %>
```

那麼結果就會在網頁上呈現出計算後的總和，因此他已經把網頁的HTML跟perl程式碼作了緊密的結合。尤其如果我們使用%作為程式行的起始，就更能表達出其中不可切割的關係了：

```
% $foo = 70;  
% if ($foo >= 60) {  
你的成績及格了  
% } else {  
你被當了  
% }
```

我們常使用這樣的方式來把條件判斷穿插在HTML裡面，所以你的內容已經是夾雜各種語法的一個綜合體了。而且不像Template使用自己定義的特殊語法，你在Mason中使用的大多是標準的HTML跟Perl語法(雖然他們總是夾雜在一起)。所以你當然可以這樣寫：

```
% my @array = (67, 43, 98, 72, 87);
% for my $grade (@array) {
  你的成績是： <% $grade %>
% }
```

接下來比較特殊的是一些Mason專用的元件，利用這些元件，你可以很容易的處理一些資料。例如使用者傳來的需求就是其中一個很好的例子。在Mason中最基本的兩個全域變數(每次使用者發出各種要求時，就會產生的兩個變數)分別是\$r跟\$m。其中\$r是Apache傳來的需求內容，至於\$m則是負責Mason自己的API。所以你可以藉由\$r來取得由Apache傳來的資料，例如：

```
$r->uri
$r->content_type
```

不過我們暫時先不管\$m這個負責處理Mason API的變數，因為我們還有更有趣的東西要玩。也就是在Mason頁面中常常會被使用的<%args>...</%args>區塊。這個區塊可以用來取得由使用者藉由POST/GET傳來的參數，一個很簡單的例子當然就是像這樣：

```
http://my.site/mason.html?arg1=value1&arg2=value2
```

於是我們就在mason.html裡面加上args區塊，利用ARGS取得這些變數之後，我們就可以在頁面中自由使用了。

```
<%args>
$arg1
$arg2
</%args>
```

所以剛剛累加的程式，我們可以由使用者輸入想要累加的數字，這時候，我們只要把while的結束條件利用使用者輸入的變數來替換就可以了。

```
<%perl>
my $sum = 0;
while ($end > 0) {
  $sum+=$end;
}
</%perl>
總和： <% $sum %>

<%args>
$end
</%args>
```

看來應該不是太困難，只是有點雜亂。如果我們每次都想要在頁面開始之前，就先用perl進行一堆運算，判

斷的時候，可以把<%perl> ... </%perl>這一大段的區塊搬離開應該屬於HTML的位置嗎？其實在Mason也替你想到了這個問題，所以在Mason中你可以使用<%init> ... </%init>這個區塊，也就是進行初始化的工作。我們把剛剛的那一段頁面的程式碼重新排列組合一下。

```
總和： <% $sum %>

<%init>
my $sum = 0;
while ($send > 0) {
    $sum+=$send;
}
</%init>

<%args>
$send
</%args>
```

這樣看起來顯然乾淨多了，不過記得我們在使用Template::Toolkit的時候有一個很不錯的概念，也就是[% INSERT %]/[% INCLUDE %]的方式，在Mason中也有類似，也就是利用<& ... &>的方式來載入你的自訂元件。用個簡單的例子來看：

```
<HTML>
<HEAD><TITLE>標題</TITLE></HEAD>
<BODY>
<TABLE>
% for my $grade (@grades) {
<TR><TD><% $grade %></TD></TR>
% }
</TABLE>
<HR>
copyright 2004-2005 Hsin-chan Chien
</BODY>
</HTML>

<%args>
@grades
</%args>
```

這時候，我們可以把前、後的HTML分別放到header跟footer兩個地方，然後利用<& ... &>來載入，所以這個內容就會被改寫為：

```
<& header &>
<TABLE>
% for my $grade (@grades) {
<TR><TD><% $grade %></TD></TR>
% }
</TABLE>
<& footer &>

<%args>
@grades
</%args>
```

這對於一整個網站維持所有網頁中部份元素的統一是一種非常方便而且有用的方式。而且任何在獨立的元素中被修改的部份也會在所有的頁面一次更新，這絕對比起你一個一個檔案修改要來得經濟實惠許多。尤其當你所進行的是一個非常龐大的網站時，更能了解這種用法的重要性。

不可否認，我們花了這麼多的頁面來講這三個目前在Perl社群中最被常用來進行網站程式的工具模組，卻只能對每一個部份做非常入門的介紹。畢竟這三個模組都是非常複雜而且功能強大的。另外的特點就是他們都可以詳細到各自出版一本完整的使用手冊。不過對於一開始想要嘗試使用這幾個模組的人來說，事實上也能用陽春的功能幫你進行許多繁複的工作了。別忘了，大多數的Perl模組或語法，你只要了解他們的百分之二十，就可以處理百分之八十的日常工作了。

習題：

1. 以下是一個HTML頁面的原始碼，試著寫出action中指定的print.pl，並且印出所有欄位中，使用者填入的值。

```
<HTML>
  <HEAD>
    <TITLE>習題</TITLE>
  </HEAD>
  <BODY>
    <FORM ACTION="print.pl" METHOD="POST">
      姓名：<INPUT TYPE="text" NAME="name"><BR/>
      地址：<INPUT TYPE="text" NAME="address"><BR/>
      電話：<INPUT TYPE="text" NAME="tel"><BR/>
      <INPUT TYPE="submit">
    </FORM>
  </BODY>
</HTML>
```

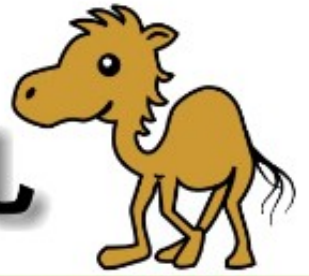
2. 承上題，試著修改剛剛的print.pl，並且利用Template模組搭配以下的模板來進行輸出。

```
<TABLE>
  <TR><TD>姓名：</TD><TD>[% name %]</TD></TR>
  <TR><TD>地址：</TD><TD>[% address %]</TD></TR>
  <TR><TD>電話：</TD><TD>[% tel %]</TD></TR>
</TABLE>
```

3. 承上題，將利用Template輸出的部份改為HTML::Mason。



# Perl 學習手札



There is more than one way to do it.

## 17. Perl與系統管理

Perl之所以能夠一直在腳本程式中佔有一席之地，有一些部份其實也是因為在系統管理中，Perl還能發揮著不錯的效用，而且在使用上也是非常方便。它能夠像shell script一樣，拿了就直接用，而不需要定義一堆變數，物件，物件的方法之後才開始寫程式碼。你可以找到相關的模組，然後非常迅速的完成你想達到的目的。所以它保留了shell script的方便性，卻又比shell script擁有更多的資源。當然優點也在某些程度上被當為缺點，例如有些人認為Perl程式碼非常的不夠嚴謹，因為相對於Java或Python，它顯然太過自由了。其實Perl對於Linux/\*BSD的方便性幾乎是不言而喻，很多時候在這些系統中都會預設安裝了Perl，因為Perl能夠相當程度的處理系統中的雜事。但是不只如此，如果你是一個系統管理員，Perl能幫你作的事情也許比你想像中的還要多。尤其當你需要對很多系統的資料進行搜尋，比對的時候，Perl就更能顯示它的重要性。何況很多時候，我們可以透過日誌檔(log)的分析來進行系統的監控跟效率的評比，而這也正是Perl的其中一項專長。這也就對於很多系統管理的統計部份(例如MRTG跟awstats)是藉由perl來達成能做出很好的解釋了。這一章的大綱是根據Autrijus Tang在對一群準Linux系統管理員上課前，我們一起討論出來的結果，而內容則有許多是直接使用他上課中使用的範例。這些範例在原作中以開放文本的方式釋出，各位可以直接使用在測試或其他正式的產品中。

### 17.1 Perl在系統管理上的優勢

其實就像我們前面所提的，Perl在系統管理上有著非常重要的應用跟地位，對於許多Unix-like之類的作業系統管理來說，Perl經常是他們的好幫手。可是Perl到底有甚麼特別的優勢呢？除了我們剛剛提到的文字比對，處理的優勢外，Perl其實還有不少能夠吸引人的地方。其中最特殊幾點大概包括：

1. Perl的黏性特強：Perl被稱為是一種黏膠程式語言，他能夠用最省力的方式把各種東西綁在一起。作為一個系統管理員總是會大量接觸各式各樣的工具，因此需要整合這些工具的機會就非常的大，所以perl的優勢在這個時候就能夠容易的顯現出來。
2. Perl資源豐富：我們之前多次提到的CPAN就是最豐富的資料庫，不單單是網路程式，資料庫處理，其實就像系統管理相關的部份，也有為數不少的模組。這些模組不但是許多系統管理員實際用來管理的工具，當然也是他們的經驗。所以你看這些模組，不單單是找尋可供應用的工具，也可以藉此挖掘這些人管理系統的方式。
3. Perl的作業環境：除了Unix-like的各種作業系統能夠輕易的安裝，執行Perl之外，微軟的Windows或是OS/2，以及蘋果公司的Mac OS也都可以讓Perl正常的運作。因此很多時候，當系統管理者同時必須管理一種以上的作業平台時，也能夠輕易的使用相同的工具。
4. Perl幾乎是一種標準：這裡所謂的標準，其實是因為目前已經很多的系統管理工具都是使用Perl所開發出來的，所以如果系統管理人員如果可以擁有Perl的能力，那麼自己維護或修改這些系統的可能性就可以大大增加。

### 17.2 Perl的單行執行模式

很多系統管理員使用Perl當然是因為Perl的順手跟方便，就像我們說的，你總不會希望找一個檔案，或置換檔案的某些字串前還要先定義一大堆變數，或是先弄一個物件，然後拿來繼承，再利用被繼承的物件寫出置換字串的物件方法。然後又沒有好的正規表示式，然後你可能得用substr一個一個去找出來。最慘的可能是你寫完這樣的程式已經是下班時間，所以你還得加班把需要的結果搞定。而且你知道，程式寫出來不一定可以那麼順利，尤其你寫了那麼長的程式，有bugs也在所難免，然後……。於是，下場應該是可以預料的。雖然很多人批評Perl非常不嚴謹，可是換個角度看，應該是說Perl允許你「隨手」寫出可以解決手邊工作的工具。而且Perl的「隨手」還不是普通的隨手。因為Perl提供單行執行模式，所以你可能可以看到這樣的一行程式：



```
perl -MEncode -pi -e '$_ = encode_utf8(decode(big5 => $_))' *
```

執行完這一行之後，你該目錄下的Big5檔案就會全部都變成UTF8了。這實在非常神奇，不是嗎？其實不只這樣，你還可以利用一行的程式把檔案中的某些字串一次換掉。就像這樣：

```
perl -pi.bak -e "s,foo,bar," *
```

這樣一來，所有檔案中的foo就全部被換成bar了。希望你沒有繼續想像如果你手動替換或是正在用其他結構嚴謹的程式語言在進行中。當然，如果你現在還在這裡，我們就利用那些堅持一定要有足夠完整的程式架構才能執行的傢伙還在奮鬥的時間，來看看怎麼執行單行的Perl吧。

首先，最基本的就是"-e"這個參數了，你可以使用perl -e來執行一行程式。不過請注意，我說的是一行程式，而不是一個敘述句，所以你當然可以這麼寫：

```
perl -e "$foo = 2; print $foo"
```

接下來，雖然只是一行的Perl程式，可是你還是可以使用Perl的其他各式各樣模組，如果沒有辦法使用模組，那單行的執行模式大概也沒有人願意使用了吧！所以我們使用了"-M"的選項來指定所要使用的模組，就像前面的第一個例子中，我們用了"-MEncode"來指定使用Encode這個模組。

接下來，我們要讓這一行程式能對所有我們指定的檔案工作，所以我們使用了"-p"這個選項這個選項允許使用者以迴圈的方式來執行這一行程式，在這裡也就讓我們可以使用\*這個萬用字元來指定所有的檔案。接下來，我們看到另一個選項是"-i"，這是讓perl會自動幫你進行備份，免得你對檔案進行操作之後，結果非常不滿意，卻還得手動改回來。所以當我們指定了"-i.bak"就是讓所有被修改的檔案在修改前就先備份一個副檔名為.bak的檔案。不過這是小寫i，如果你用了大寫的I，那意義可就大不相同了，"-I"讓你可以指定@INC的內容。

### 17.3 管理檔案

系統管理一開始大概都要面對一大堆的檔案吧，而基本的檔案管理其實還沒有太過複雜，只要你的shell夠熟，幾乎也可以應付很多狀況。例如你可以輕鬆的利用find找到你要的檔案，甚至讓他們排序，就像這樣：

```
find /home/hcchien/svn/ *.txt -print | sort
```

如果你用File::Rule來寫，可能會像這樣：

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

```
use File::Find::Rule;
```

```
my $rule = File::Find::Rule->new;
```

```
my @files = $rule->file->name( '*.txt' )->in('/home/hcchien/');
```

```
print "$_/" for @files;
```

Perl的寫法不但比較長，比較複雜，而且速度比起你在shell底下真是慢了好幾倍。暫時看起來，Perl在這樣簡單的狀況下實在不特別好用。可是別忘了，像這樣簡單的狀況，每個人都習慣隨手就用shell解決，可是如果情況稍微複雜一點呢？比如我想找出檔案狀態是可執行的.txt檔案，那麼你應該怎麼做呢？接下來，我想把這些檔案的可執行模式取消，然後也許再修改某些內容……。如果只是單一內容，shell確實非常輕巧，快速，可是一但我們要把一堆操作集合在一起時，你就會發現perl有甚麼過人之處了。

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

use File::Find::Rule;

my $rule = File::Find::Rule->new;
$rule->file;
$rule->executable;
my @files = $rule->name( '*.txt' )->in('/home/hcchi en/');

for my $file (@files) {
    open READ, $file;
    s/foo/bar/g while (<READ>);
}
```

當然，你還可以對這些檔案做其他的操作，例如每個檔案都插入一列新的資料等等。這時候，有另外一個模組就顯得非常有用，這麼模組就是IO::All。還記得我們之前怎麼讀入一個檔案的內容吧？IO::All現在可以讓你非常簡單的控制檔案，我們來比較看看：

用傳統的作法，我們可以這麼寫：

```
open FILE, "<foo.txt";
$buf.=$_ while (<FILE>);
```

倒也相當簡潔，不過現在使用IO::All，就只要這麼做：

```
my $buf < io('foo.txt');
```

不過我們並不打算在這裡深切的介紹IO::All這個模組，因為我們還有其他更重要的事情要做。

## 17.4 郵件管理

接下來讓我們來看看作為伺服器的一個重要工作，也就是關於Mail的管理。也因為對於郵件的管理需求其實非常的大，所以其實相關的管理工具也不在少數。例如可以過濾廣告信件，發送大量信件或是寄發一般的通知信件等等。不過由於某些工作的特殊性質，使得Perl成為這些工作中非常能夠勝任重要工具，這些工作中當然有不少是字串內容的分析，而最具代表性的也就是廣告信的過濾了。

### 17.4.1 Mail::Audit + Mail::SpamAssassin

這是一個非常具有殺手級實力的一個Perl模組，如果你已經是一個現任的網路管理人員，整天聽到你所管理的郵件伺服器中不斷傳來廣告郵件，而且你還不知道這個模組，那麼應該先去CPAN上搜尋Mail::SpamAssassin這個模組。然後你還可以搭配Mail::Audit這個模組。這裡的例子是許多人使用Mail::Audit跟Mail::SpamAssassin時經常會作為過濾信件第一關的檢查工具：

```
use Mail::Audit;
use Mail::SpamAssassin;
my $m = Mail::Audit->new(
    emergency => "~/emergency_mbox",
    nomime    => 1,
);
my $sa = Mail::SpamAssassin->new;
$m->pipe("listgate cle") if $m->from =~ /svk-devel/;    # 送到pipe
$m->accept("~/perl")    if $m->from =~ /perl/;           # 接進特定信箱
```

```

$sm->reject("no rbl")      if $sm->rbl check;      # 拒絕黑名單
$sm->ignore                 if $sm->subject =~ /sex/i;  # 忽略信件
my $status = $sa->check($m);      # 檢查垃圾信
if ($status->is_spam) {
    $status->rewrite_mail;      # 加上檔頭
    $sm->accept("~/Mail/spam");  # 收進垃圾桶
}
$sm->noexit(1); $sm->accept("~/Mail/%Y%m"); $sm->noexit(0); # 按月彙整
$sm->accept;                  # 其餘接收

```

其實如果利用Mail::Audit，還可以直接套用Mail::Audit::MAPS。因為這樣就可以直接把一些已經被列為黑名單的寄件者先排除在外了，而且使用上也沒有甚麼太大的差別。其實你只需要多做一個判斷：

```

if ($mail->rbl check) {
    .....
}

```

另外，Mail::Audit還有一個常用的外掛程式則是Mail::Audit::KillDups。他可以幫你刪除一些ID重複的信件，其實這也是可能的廣告信來源之一，所以你可以又事先過濾掉一些垃圾郵件。其實不只這些，Mail::Audit的外掛程式種類之多，實在讓人覺得有趣，我自己另一個常用的是Mail::Audit::PGP，不過這就未必適合所有人了。

至於Mail::SpamAssassin，除了搭配Mail::Audit之外，其實也有命令列程式。透過命令列程式，你可以設定自己的黑名單(blacklist)跟白名單(whitelist)，也可以透過手動訓練的方式，來增進SpamAssassin的準確率。

#### 17.4.2 Mail::Sendmail 與 Mail::Bulkmail

另外，perl也可以用非常方便的方式來傳送mail。雖然你在系統管理時，如果需要寄發mail，可以方便的使用sendmail來傳送。可是其實有些時候你會在系統中定時執行一些程式，或許是進行系統的意外檢查，或許是做定時的工作。那麼你可能會希望這些工作如果發生意外，你可以很快的收到電子郵件通知。這時候Mail::Sendmail就變得很有用了。

```

use Mail::Sendmail;

%mail = ( To      => 'yourmail@hostname.com',
          From     => 'mail@server.com',
          Message  => "救命啊！Apache不動了！！"
        );

sendmail(%mail) or die $Mail::Sendmail::error;

```

各位大概都聽過「自相矛盾」的成語故事吧？而且千萬別以為那是書上拿來騙小孩的故事，因為真實的就發生在perl的模組中。我們之前介紹了目前幾乎被公認最好阻擋廣告信的模組：SpamAssassin，可是現在我們也要介紹另一個被認為發廣告信的強力模組，也就是Mail::Bulkmail。

```

use Mail::Bulkmail;
my $bulk = Mail::Bulkmail->new(
    LIST      => "~/listfile",      # 地址清單
    From      => 'admin@bulkmail.com', # 寄件人
    Subject   => "System Information", # 標題
    Message   => '~/announcement.txt' # 內文檔名
);

```

```
message_from_file => 1 # 從檔案讀取內文
);
$bulk->bulkmail() or die Mail::Bulkmail->error;
```

在這裡，你只需要把要一次傳送的一大堆郵件地址逐行放進一個純文字檔。Mail::Bulkmail就會幫你讀出這些地址，而且發送郵件。至於內文，你雖然可以直接寫在程式中，可是更有彈性的方式也是可以利用文字檔來讀入郵件內容。這樣一來，如果你的郵件是每週定時發送，那麼你只需要修改傳送名單跟郵件內容的檔案就可以輕鬆的讓程式替你完成其他工作了。

### 17.4.3 POP3Client 及 IMAPClient

有些時候，你可能有一些帳號是專門用來處理一些特定的工作，那麼其實這些送到該帳號的信件未必需要由一個特定的人去收，而可以利用程式去進行處理。這個時候，你可以使用Mail::POP3Client這個模組來讀取這些郵件。我們先用一個簡單的例子來看看這個模組的用法：

```
use Mail::POP3Client;
my $pop = Mail::POP3Client->new(
    USER      => "me",
    PASSWORD   => "mypassword",
    HOST       => "pop3.example.com",
);
foreach ( 1 .. $pop->Count-1 ) {
    $pop->Head( $_ ) =~ /^From: \s+somebody\@example.com/ or next;
    open my $fh, '>', "mail-$_.txt" or die $!;
    $pop->RetrieveToFile($fh, $_);
}
```

這個程式可以讓某個人寄來的信都被備份到一個特定的檔案中，其實主要的就是透過郵件檔頭中的寄件者進行比對。所以如果你可以針對主題進行比對，就可以對郵件進行分類。當然，如果你想出其他更好用的郵件過濾演算法，也許可以從這裡開始進行實做(雖然我們並不建議你這麼做)。其實一但可以直接取出郵件中的每一封郵件，我們可以做的應用就非常的廣泛了。而有了POP3Client，好像也少不了IMAPClient。我們還是先來看另一個例子吧：

```
use Mail::IMAPClient;

my $imap = Mail::IMAPClient->new;
$imap = Mail::IMAPClient->new(
    Server => $host,
    User   => $id,
    Password=> $pass,
) or die "無法連上主機：$host as $id: $@";
```

使用IMAPClient，也因為郵件伺服器種類的特性差異，讓它提供更多的操作方式給使用者。最基本的比如IMAPClient的Connected跟Unconnected，另外可以透過Range來選定某一個特定範圍的郵件。例如：

```
$imap->Range($imap->messages);
```

其中\$imap->messages會取得目前所在資料夾的所有郵件，然後透過\$imap->Range()來把這些郵件設定成要操作的郵件範圍。不過你也可以在\$imap->Range()中使用逗點分隔來指定某幾封特定的信件。另外，如果你

考慮搬移信件，你可以使用move來進行，其實非常方便，就像這樣：

```
my $newUid = $imap->move($newFolder, $oldUid)
               or die "Could not move: $@\n";
$imap->expunge;
```

另外，還有一些常用的方法例如delete\_messages和restore\_messages就分別是用來刪除郵件跟回覆被刪除的郵件。或是你可以用search來搜尋郵件。其實IMAPClient非常的強大，確實需要根據自己的需求去研究相關的文件才能確實掌握。

## 17.5 日誌檔

作為一個系統管理，能確實掌握每日的紀錄檔確實是非常重要的工作。不過如果你進入/etc/log資料夾，就會發現裡面的檔案其實是相當多的。也就是說，如果你身為一個系統管理員，每天要注意這些檔案中有沒有異常，如果你還在使用傳統的工人智慧，那麼每天進行這樣的工作就要浪費許多時間。因此要希望能夠在這部份節省更多的時間來從事其他的管理工作，我們可以使用一些工作來簡化這些日常的程序，Parse::Syslog就是其中之一。

我們先來看看一個簡單的例子，來了解Parse::Syslog怎麼幫我們處理這些日誌檔案。

```
use Parse::Syslog;
my $syslog = Parse::Syslog->new('/var/log/syslog');

while (my $entry = $syslog->next) {
    $entry->{program} =~ /sudo$/ or next;
    print localtime($entry->{timestamp}). "\n",
          "$entry->{text}\n\n";
}
```

其實程式並不困難，首先我們設定要處理的檔案，在這裡我們針對"/var/log/syslog"這個檔案來檢查。接著Parse::Syslog傳回一個物件，也就是\$syslog。接下來我們使用next這個物件操作方法逐行處理這個檔案，在檔案結束之前，\$syslog->next都會傳回真值，因此讓我們可以一行一行來進行比對的工作。我們試圖找出日誌檔中關於有使用者使用sudo這個指令，所以我們使用了正規表示式。

接下來，Parse::Syslog可以幫你取得事件發生的時間，所以當我們比對成功之後，就可以印出使用者使用sudo這個指令的時間。

所以如果我們一次把所有要監視的檔案內容利用Parse::Syslog設定好，那麼其實以後的日子就會輕鬆愉快許多了。當然，如果你擷取出這魔一堆檔案，那麼要把這些讓人需要特別注意的紀錄進行處理，才能方便管理者閱讀或檢查，這時候可以使用另一個模組來進行。

Log::Dispatch是不錯的選擇。你只要新建立一個相關的物件，並且指定要寫入的檔名，那麼可以依照紀錄的等級來將相關的訊息寫入檔案中。

```
use Log::Dispatch;
my $log = Log::Dispatch->new;           # 建立紀錄物件
# 新增紀錄檔
$log->add( Log::Dispatch::File->new(
    name => 'file',                      # 物件名稱
    min_level => 'debug',                # 紀錄門檻
    filename => '/var/log/test.log',      # 紀錄檔名
) );
```

接下來，我們把想要寫入的訊息像這樣的加入檔案中：



```
$log->log( level => 'alert', message => 'Strange data in incoming request' );
```

這裡有一個有趣的部份，也就是紀錄門檻。他可以讓使用者依照不同的等級來分門別類，而門檻的類別分別為：

```
除錯 (debug)
消息 (info)
提示 (notice)
警告 (warning)
錯誤 (error)
關鍵 (critical)
警鈴 (alert)
緊急事件 (emergency)
```

既然有了這些分別，你就可以進行更多的動作來確保系統正常的運作。例如你可以在系統遇到異常現象時發出信件給自己，就像這樣：

```
$log->add( Log::Dispatch::Email::MailSend->new(
    name => 'email',                # 物件名稱
    min_level => 'emergency',        # 紀錄門檻
    to => [ 'admin@example.com' ],   # 收件地址
    subject => 'HELP!!!!',           # 郵件標題
) );
```

於是如果你的日誌得到了一些緊急事件的訊息時，就會自動發出電子郵件給你。如此一來你可以在第一時間就知道系統異常，並且進行檢修。當然，如果你是超級負責的系統管理員，你也許希望在你沒有網路的時候也能得到相關的緊急警告，這時候手機簡訊也許是另一種通知的好方法。

```
use Net::SMS;
my $df = `df -h`;                # 取得硬碟配置資訊
$df =~ /\- \d/ or exit;           # 若沒有數字是負的就直接離開
my $sms = Net::SMS->new;           # 簡訊物件
$sms->accountId("123-456-789-12345"); # 帳號
$sms->accountPassword("mypassword"); # 密碼
$sms->sourceAddr("0928-000-000");    # 來源
$sms->destAddr("0928-999-999");     # 目的
$sms->msgData("HARD DISK FULL: \n$df"); # 訊息
$sms->submit;                       # 送出簡訊
$sms->success or die $sms->errorDesc; # 偵測錯誤
```

我們使用Net::SMS可以直接發送簡訊，在這個例子中，我們使用shell指令去檢查磁碟空間。並且當發現磁碟空間不足時就發出手機簡訊來警告系統管理人員。雖然你寫了這些程式之後，未必就可以獲得老闆的賞識而加薪，不過我想讓老闆少點機會找你麻煩應該還算是一項大利多吧！

## 17.6 報表

身為系統管理人員，尤其當你是在企業內部進行系統管理時，最容易遭到忽略。因為當大家系統非常順暢時，幾乎沒人會歸功於系統管理員的認真。於是要怎麼拿出實際的內容來說服其他的人，這也算是系統管理人員的重大業務之一了吧！很多人使用MRTG(註一)來監測網路流量，使用awstats(註二)來看網站的各式數

據。  
可惜的是大多數的人對這些報表並不太有興趣，除非你管理的是一個(或一堆)網站，而且公司的業務也就是經營這些網站。既然如此，那麼一個系統管理員有些時候其實還是要呈現出自己優良的管理績效，在沒有現成的套裝工具下，要怎麼樣快速的建立出漂亮的報表其實也是不小的學問。  
如果你需要的是文字的報表，那麼我們前一章使用的Template就非常適合，你可以利用Template::Toolkit畫一個HTML的報表。

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

use strict;
use Template;
use IO::All;
use Mail::MboxParser;

my $dir = io('/var/mail');
my %all;
while (my $io = $dir->next) {
    if ($io->is_file) {
        eval {
            my $mb = Mail::MboxParser->new("$io", newline => '#DELIMITER');
            %all{$io} = $mb->nmsgs;
        }
    }
}

my $config = {
    INCLUDE_PATH => '/search/path',
    POST_CHOMP   => 1,
};

my $template = Template->new($config);
my $vars = { messages => \%all };
my $input = 'report.html';
my $output;

$template->process($input, $vars, $output) # 處理模板內容
|| die $template->error();
print $output;
```

這時候，你可以取得一個含有所有信箱郵件個數的一個雜湊變數(我們使用了Mail::MboxParser)。如此一來，你只要弄一個漂亮的模板，就可以讓系統動態把資料填入，隨時可以監控目前大家信箱內的郵件數目了。不過這樣其實還沒結束，因為很多老闆或主管對於文字的接受度總是比較低，所以如果你有漂亮的報表，那麼給他們的印象應該也會隨之提高。這時候，你也許可以認真考慮使用GD::Graph這個模組。這個模組可以让你輕鬆的畫出漂亮的統計圖表，你可以根據資料的屬性以及需求的差異，畫出例如圓餅圖，曲線圖，柱狀圖等等，如果你還想更絢麗，GD::Graph還可以畫出立體的3D圖形。  
我們用另一個例子來看看怎麼使用GD::Graph吧：

```
use GD::Graph::bars3d;
my $graph = GD::Graph::bars3d->new(800, 600); # 新增柱狀圖
my @files = </var/log/maillog.*.bz2>;

my $image = $graph->plot([
    [map /(\d+)\./g, @files],
    [map -s, @files],
]) or die $graph->error;
```

```
open my $fh, '>', '3.png' or die $!;
print $fh $image->png;      # 儲存影像
```

我們取得了每次的電子郵件日誌檔的，然後根據這些檔案的大小進行統計。這時候，我們只需要訂出橫座標跟縱座標的內容，交給GD去畫就好了，你當然也可以定時的要求程式幫你畫出某些統計圖表。很有用吧！你可以定時交出漂亮的工作報表，而且還是由電腦自動產生。

利用Perl來協助系統管理其實還算是非常方便的，何況已經有許多的系統管理員早就在做這些工作，也因此我們有很多方便的工具可以使用。這完全讓我們省下許多時間，尤其當更多的系統管理員每天都花許多時間在這些繁瑣而且又沒有變化的工作上。

習題：

1. 找出maillog中被reject(退信)的資料，也就是找到日誌檔中以reject標明的內容。例如：

```
Jun  3 00:00:46 dns2 postfix/smtpd[71431]: D988D6A: reject: RCPT
from smtp2.wanadoo.fr[193.252.22.29]: 450 <
fnatterdobkl@hcchien.org>: User unknown in local recipient
table; from=<> to=<fnatterdobkl@hcchien.org>
proto=ESMTP helo=<mwinf0203.wanadoo.fr>
```

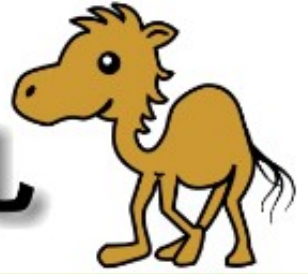
2. 承上題，統計當月每天的退信數字，並且畫成長條圖。

註一：Multi Router Traffic Grapher (<http://people.ee.ethz.ch/~oetiker/webtools/mrtg/>)

註二：<http://awstats.sourceforge.net/>



# Perl 學習手札



There is more than one way to do it.

## 附錄A. 習題解答

### 第一章：

#### 1. 試著找出你電腦上的Perl版本為何。

解答：當然，你得先確定你的電腦上確實裝了Perl。如果你在Unix/Linux/\*BSD或是Mac OS X上，打開你的終端機(terminal)，進入shell，接著打入perl -v，其中第一行中就可以看到你電腦上的Perl版本了。詳細的內容可以參閱第一章內容。

#### 2. 利用perldoc perl找出所有的perl文件內容

解答：當你能看到perl -v的內容之後，你的電腦應該已經安裝Perl。接下來，你可以在shell中打入perldoc perl。於是你可以看到所有的文件，像這樣：

#### Overview

perl	Perl overview (this section)
perlintro	Perl introduction for beginners
perltoc	Perl documentation table of contents

.....

perluts	Perl notes for UTS
perl vmesa	Perl notes for VM/ESA
perl vms	Perl notes for VMS
perl vos	Perl notes for Stratus VOS
perl win32	Perl notes for Windows

至於如果你想看其中的任何一份文件，只要使用perldoc這個指令即可，例如可以使用perldoc perlsyn來看關於Perl語法的相關文件。

#### 3. 利用Perl寫出第一個程式，印出你的名字

解答：你只需要使用print就可以解決這個問題，所以像是這樣：

```
print "簡信昌";  
當然，你還可以用單引號，至少在這裡的用法是一樣的：  
print '簡信昌';
```

### 第二章：

#### 1. 使用換行字元，將你的名字以每個字一行的方式印出。

解答：最簡單的方式，你可以這麼寫

```
print "簡\n信\n昌\n";
另外，你當然可以逐行印出：
print "簡\n";
print "信\n";
print "昌\n";
```

2. 印出'\n', '\t'字串。

解答：你可以單純的使用單引號

```
print '\n \t';
或是使用雙引號，然後加上跳脫字元：
print "\\n \\t";
```

3. 讓使用者輸入姓名，然後印出包含使用者姓名的招呼語(例如：hello xxx)。

解答：這裡主要是要能夠讓使用者輸入，所以我們應該使用<STDIN>

```
#!/usr/bin/perl

use strict;

chomp(my $input = <STDIN>);
print "Hello $input \n";
```

第三章：

1. 試著把串列 (24, 33, 65, 42, 58, 24, 87) 放入陣列中，並讓使用者輸入索引值 (0...6)，然後印出陣列中相對應的值。

解答：在這裡，我們並不先對輸入值做判斷，也就是假設使用者都會乖乖的輸入0...6的數字。

```
#!/usr/bin/perl

use strict;

my @array = (24, 33, 65, 42, 58, 24, 87);
chomp(my $input = <STDIN>);      # 使用者輸入
print $array[$input];
```

2. 把剛剛的陣列進行排序，並且印出排序後的結果。

解答：這部份其實只需要使用一個排序的函式sort。

```
print sort @array;
```



3. 取出陣列中大於40的所有值。

解答：至於這一個部份，我們則是可以使用grep這個函式直接完成：

```
print grep {$_ > 40} @array;  
你當然還可以把過濾出來的值再進行排序，就像這樣：  
print sort grep {$_>40} @array;
```

4. 將所有陣列中的值除以 10 後印出。

解答：至於要把一個陣列中的所有值同時進行某種轉換，對應，就可以使用map

```
print map {$_/10} @array;  
同樣的，你還是可以試著將結果排序
```

第四章：

1. 算出1+3+5+...+99的值。

解答：我們可以使用for迴圈或是while迴圈來進行。

```
#!/usr/bin/perl  
  
use strict;  
my $sum = 0;  
for (my $i = 0; $i < 100; $i+2) {  
    $sum+=$i;  
}  
  
print $sum;
```

如果使用while，那麼程式碼應該像是這樣

```
#!/usr/bin/perl  
  
use strict;  
  
my ($sum, $i);  
while ($i < 100) {  
    $sum+=$i;  
    $i++;  
}  
print $sum;
```

2. 如果我們從1加到n，那麼在累加結果不超過100，n的最大值應該是多少？

解答：這時候，我們用while迴圈似乎就比較方便了

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

```

my ($sum, $i);
while ($sum <= 100) {
    $sum+=$i;
    $i++;
}

print $i;

```

3. 讓使用者輸入一個數字，如果輸入的數字小於50，則算出他的階乘，否則就印出數字太大的警告。

解答：這裡有兩個重點，一個是if判斷式，另一個則是計算階乘的迴圈。

```

#!/usr/bin/perl

use strict;

chomp(my $input = <STDIN>);
if ($input < 50) {
    my $total = 1;      # 這跟算總和不同
    for (my $i = 1; $i <= $input; $i++) {
        $total*=$i;    # 進行階乘
    }
    print $total;
} else {
    print "數字太大了";
}

```

## 第五章：

1. 將下列資料建立一個雜湊：

John => 1982.1.5

Paul => 1978.11.3

Lee => 1976.3.2

Mary => 1980.6.23

解答：我們可以很簡單的使用串列，或是=>來建立雜湊：

```

my %hash = ( John => "1982.1.5",
             Paul => "1978.11.3",
             Lee => "1976.3.2",
             Mary => "1980.6.23" );

```

至於如果使用串列，則非常單純的只要：

```

my %hash = qw/John 1982.1.5 Paul 1978.11.3 Lee 1976.3.2 Mary 1980.6.23/;

```

2. 印出1980年以後出生的人跟他們的生日。

解答：我們逐個取出雜湊的鍵值，然後比較資料。

```

my %hash = qw/John 1982.1.5 Paul 1978.11.3 Lee 1976.3.2 Mary 1980.6.23/;
while ( ($key, $value) = each %hash) {
    print "$key, $value" if ($value gt "1980");
}

```

3. 新增兩筆資料到雜湊中：

Kayle => 1984.6.12

Ray => 1978.5.29

解答：要新增雜湊中的內容很簡單，只需要單純的指定鍵跟對應的值就可以了。

```
$hash{Kayle} = '1984.6.12';  
$hash{Ray} = '1978.5.29';
```

4. 檢查在不修改程式碼的情況下，能否達成第二題的題目需求

解答：由於我們使用while迴圈，它會自動檢查雜湊中所有的內容，因此即使我們新增了兩筆資料，對於迴圈的運作並不會有所影響。

第六章：

1. 下面有一段程式，包含了一個陣列，以及一個副常式diff。其中diff這個副常式的功能在於算出陣列中最大與最小數值之間的差距。請試著將這個副常式補上。

```
#!/usr/bin/perl -w  
  
use strict;  
  
my @array = (23, 54, 12, 64, 23);  
my $ret = diff(@array);  
print "$ret\n";          # 印出 52 (64 - 12)  
my @array2 = (42, 33, 71, 19, 52, 3);  
my $ret2 = diff(@array2);  
print "$ret2\n";        # 印出 68 (71 - 3)
```

解答：我們需要進行的工作包括讀取透過副常式傳來的陣列內容，並且取得陣列中的最大值與最小值，再進行兩個值的計算。

```
sub diff {  
    my @param = @_;  
    my ($max, $min) = ($param[0], $param[0]);  
    for (@param) {  
        $max = $_ if $_ > $max;          # 求最大值  
        $min = $_ if $_ < $min;          # 求最小值  
    }  
    $max - $min;  
}
```

2. 把第四章計算階乘的程式改寫為副常式型態，利用參數傳入所要求得的階乘數。

解答：我們先來看看第四章中關於階乘的這段程式碼。

```
my $total = 1;
```

```
for (my $i = 1; $i <= $input; $i++) {
    $total*=$i;
}
print $total;
```

接下來我們將它改為副常式型態：

```
sub times {
    my $input = shift;                # 取得使用者傳入的參數
    my $total = 1;
    for (my $i = 1; $i <= $input; $i++) { # 計算階乘
        $total*=$i;
    }
    return $total;
}
```

## 第七章：

1. 讓使用者輸入字串，並且比對是否有Perl字樣，然後印出比對結果。

解答：這裡只需要使用最單純的樣式比對來判斷比對的結果。

```
#!/usr/bin/perl

use strict;

chomp(my $input = <STDIN>);
if ($input =~ /Perl/) {
    print "比對成功\n";
} else {
    print "比對失敗\n";
}
```

2. 比對當使用者輸入的字串包含foo兩次以上時(foofoo 或是 foofoofoo 或是 foofoofoofoo...), 印出比對成功字樣。

解答：使用群組比對的方式似乎可以簡單的達到這個要求，所以設定樣式為foo。請注意，你不能將樣式設定為foofoo，否則如果foo的出現次數是單次(例如三次)的話，那就無法正確比對了。所以我們的寫法可以像這樣：

```
#!/usr/bin/perl

use strict;

chomp(my $input = <STDIN>);
if ($input =~ /(foo){2,}/) {    # 必須出現兩次以上
    print "比對成功\n";
} else {
    print "比對失敗\n";
}
```

## 第八章：

1. 延續第七章的第一題，比對出perl在字串結尾的成功結果。

解答：和第七章的第一個問題不同的是在於我們必須使用定位點的概念。所以我們只寫出需要進行字串結尾

的樣式。

```
$i nput =~ /perl $/;
```

2. 繼續比對使用者輸入的字串，並且確定是否有輸入數字。

解答：這個問題主要的部份在於可以使用字符集或是字符集的簡寫。最簡單的當然是直接使用\d的簡寫形式。

```
$i nput =~ /\d/;  
而其實也就是可以使用  
$i nput =~ /[0-9]/;
```

3. 利用回溯參照，找出使用者輸入中，引號內(包括雙引號或單引號)的字串。

解答：在這裡，我們特別要求不管使用者使用單引號或雙引號時都可以找出引號中的字串。因此在比對時，就必須使用字符集，也就是["'"]必須同時被納入。但是一但使用字符集來進行比對，為了避免產生錯誤的對稱，例如"單引號"，我們就必須使用回溯參照，以確定我們比對的是對稱的引號。另外，則是要注意使用記憶變數來取得我們比對出來的內容。所以比對的樣式應該可以這麼寫：

```
$i nput =~ /['"](.+?)\1/;  
print $1; # 比對出來的內容
```

我們還要注意括號裡的內容，首先是一個萬用符號.，接下來是重複符號+，這是指至少出現一次的重複符號。接下來是為了避免比對超過第一次對稱的引號範圍，所以我們用了不貪多的修飾符號?。而這裡面正是我們所要取得的全部內容，所以就用了記憶變數的符號。

4. 找出使用者輸入的第一個由p開頭，l結尾的英文字。

解答：這裡我們要確定的有幾個部份，也就是我們要比對的是一個「字(word)」，因此p跟l分別是這個字的兩個端點，我們也就可以利用\b來畫出這個字的界線。當然，記憶變數還是需要的，因為我們不但要確定是否比對成功，因為我們還想取得比對成功的字串內容。所以我們就把比對樣式寫成這樣：

```
$i nput =~ /\b(p\w*l)\b/;  
print $1;
```

第九章：

1. 陸續算出 (1...1) 的總和，(1...2) 的總和，...到 (1...10) 的總和。但是當得到總和大於50時就結束。

解答：這個題目主要有兩個部份，第一個是關於計算加總的部份，一般我們也許常用for迴圈來進行加總的部份，當然你也可以使用while迴圈或其他方式。接下來，你要考慮計算出來的總和，讓他不超過50。這個情況下，可以使用last來做迴圈的餓外控制。

```
#!/usr/bin/perl -w  
  
use strict;
```



```

my ($base, $sum) = (0, 0);
for $base (1...10) {
    $sum = sum($base);
    last if ($sum > 50);
    print "$base => $sum\n";
}

sub sum {
    my $index = shift;
    my $summary;
    for (1...$index) {
        $summary += $_;
    }
    return $summary;
}

```

2. 把下面的程式轉為三元運算符形式：

```

#!/usr/bin/perl -w

use strict;

chomp(my $input = <STDIN>);
if ($input < 60) {
    print "不及格";
} else {
    print "及格";
}

```

解答：這部份其實只要考慮if敘述句內的部份。所以我們先找出關鍵的部份，也就是if條件跟else的內容。接下來就只需要一一對照轉換就可以了。

```

原來寫法：      if ($input < 60) { print "及格" } else { print "不及格" }
三元算符寫法：  ($input < 60) ? print "及格" : print "不及格";

```

第十章：

1. 試著將下面的資料利用perl寫入檔案中：

```

Paul, 26933211
Mary, 21334566
John, 23456789

```

解答：這裡主要的重點就是開啟檔案，寫入內容，至於資料，我們可以使用雜湊來處理。

```

#!/usr/bin/perl -w

use strict;

my %tel = ("Paul", 26933211, "Mary", 21334566, "John", 23456789);

```

```
open FILE, ">tel ephone";
for (keys %tel) {
    print FILE "$_ => $tel{$_}\n";
}
close FILE;
```

然後你可以去看檔案"tel ephone"的內容，也就是：

```
John => 23456789
Mary => 21334566
Paul => 26933211
```

## 2. 在檔案中新增下列資料：

```
Peter, 27216543
Ruby, 27820022
```

解答：剛剛我們在開啟檔案時使用了">"來表示寫入一個新檔。接下來，我們只是要在現有的檔案中加入新的內容，因此我們應該改用">>"的方式，以避免原來的檔案被清空。

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

use strict;

my %tel = ("Peter", 27216543, "Ruby", 27820022);
open FILE, ">>tel ephone";
for (keys %tel) {
    print FILE "$_ => $tel{$_}\n";
}
close FILE;
```

這樣我們就可以很容易的看出其中的不同了。

## 3. 從剛剛已經存入資料的檔案讀出檔案內容，並且印出結果。

解答：不同於剛剛寫入檔案，我們現在需要的是把檔案內容讀出。

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

use strict;

open FILE, "tel ephone";
while (<FILE>) {
    print $_;
}
close FILE;
```

## 第十一章：

### 1. 列出目前所在位置的所有檔案/資料夾名稱。

解答：我們可以用簡單的角括號方式來取得目前目錄下的所有內容。

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

use strict;

my @files = <*>;
print "$_\n" for @files;
```

2. 承一，只列出資料夾名稱。

解答：在這裡，我們只需要修改剛剛的程式，在列印前判斷我們取得的是檔案或資料夾。

```
my @files = <*>;
for (@files) {
    print "$_\n" if (-d $_);
}
```

3. 利用perl，把目錄下所有附檔名為.pl的檔案修改權限為可執行。

解答：首先我們還是使用角括號，但是我們這次要取出的只有所有附檔名為.pl的檔案。接下來，再以chmod來修改權限。

```
my @files = <*.pl>;
chmod 0755, @files;
```

第十二章：

1. 讓使用者輸入字串，取得字串後算出該字串的長度，然後印出。

解答：這裡主要還是要使用length這個函式，來取得字串長度。

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

use strict;

chomp(my $str = <STDIN>);
print length($str);
```

2. 利用sprintf做出貨幣輸出的表示法，例如：136700以\$136,700，26400以\$26,400表示。

3. 利用雜湊%hash = (john, 24, mary, 28, david, 22, paul, 28)進行排序，先依照雜湊的值排序，如果兩個元素的值相等，則依照鍵值進行字串排序。

第十三章：

1. 試著在你的Unix-like上的機器裝起CPANPLUS這個模組。

解答：你可以直接透過CPAN來安裝CPANPLUS，或是到<http://search.cpan.org/>下載CPANPLUS的原始碼，解開之後直接安裝。成功安裝之後，你可以在shell底下使用CPANPLUS，就像這樣：

```
[hcchi en@Apple]% cpanp
```

```
CPANPLUS: : Shell::Default -- CPAN exploration and modules installation (v0.03)
*** Please report bugs to <cpanplus-bugs@lists.sourceforge.net>.
*** Using CPANPLUS: : Backend v0.049.
*** ReadLine support available (try 'i Term::ReadLine::Perl').

CPAN Terminal >
```

2. 還記得我們寫過階乘的副常式嗎？試著把它放入套件My.pm中，並且寫出一個程式呼叫，然後使用這個副常式。

解答：其實如果知道Package的包裝方式跟使用方式，這個問題可以很容易的解決。

```
sub times {
    my $input = shift;                # 取得使用者傳入的參數
    my $total = 1;
    for (my $i = 1; $i <= $input; $i++) { # 計算階乘
        $total*=$i;
    }
    return $total;
}
```

第十四章：

1. 下面程式中，%hash是一個雜湊變數，\$hash\_ref則是這個雜湊變數的參照。試著利用\$hash\_ref找出參照的所有鍵值。

```
%hash = ( name => 'John',
           age  => 24,
           cellphone => '0911111111' );
$hash_ref = \%hash;
```

解答：其實你只要解開雜湊參照，就可以簡單的使用keys函式來取得參照的所有鍵。

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

use strict;

my %hash = ( name => 'John',
             age  => 24,
             cellphone => '0911111111' );
my $hash_ref = \%hash;

my @keys = keys %{$hash_ref};
print $_ for @keys;
```

2. 以下有一個雜湊，試著將第一題中的雜湊跟這個雜湊(@hash\_array)放入同一陣列中。

```
%hash1 = ( name => 'Paul',
           age  => 21,
           cellphone => '0922222222',
```

```
birthday => '1982/3/21' );
```

解答：由於陣列中的元素都是純量，所以我們需要的是把兩個雜湊的參照放進陣列@hash\_array中。

```
my @hash_array = ( { name => 'John',  
                    age => 24,  
                    cell phone => '0911111111' },  
                  { name => 'Paul',  
                    age => 21,  
                    cell phone => '0922222222',  
                    birthday => '1982/3/21' } );
```

3. 承上一題，印出陣列\$hash\_array中每個雜湊鍵為'birthday'的值，如果雜湊鍵不存在，就印出「不存在」來提醒使用者。

解答：在這裡，我們應該先從陣列中依序取出雜湊的參照，然後解開參照，判斷參照鍵'birthday'是否存在。如果存在就可以取出其中的雜湊值。

```
#!/usr/bin/perl -w  
  
use strict;  
  
my @array = ( { name => 'John',  
               age => 24,  
               cell phone => '0911111111' },  
             { name => 'Paul',  
               age => 21,  
               cell phone => '0922222222',  
               birthday => '1982/3/21' } );  
  
for (@array) {  
    if (exists ${$_}{birthday}) { # 解開參照，並且判斷雜湊鍵是否存在  
        print ${$_}{birthday};  
    } else {  
        print "the key doesn't exist";  
    }  
}
```

其實你可以用更簡潔的方式來解開參照，也就是\$\_->{birthday}

## 第十五章：

1. 利用自己熟悉的資料庫系統(例如 MySQL 或 Postgres)，建立一個資料庫，並且利用DBI連上資料庫，取得 Database Handler。

解答：假設我們在MySQL建了一個資料庫叫做'perlbook'所以我們要連接上資料庫，就只要使用DBI。

```
my $dbh = DBI->connect('dbi:mysql:dababase=perlbook', 'user', 'password');
```

2. 試著建立以下的一個資料表格，並且利用Perl輸入資料如下：



資料表格：

```
name: varchar(24)
cellphone: varchar(12)
company: varchar(24)
title: varchar(12)
```

資料內容

```
[ name: 王小明
  cellphone: 0911111111
  company: 甲上資訊
  title: 專案經理 ]
[ name: 李小華
  cellphone: 0922222222
  company: 乙下軟體
  title: 業務經理 ]
```

解答：建立資料表格，我們可以透過各種方式，例如MySQL的用戶端程式，或現成的管理程式。當然也可以利用DBI的方式來建立新的資料表格。

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

use strict;
use DBI;

my $dbh = DBI->connect('dbi:mysql:database=perlbook', 'user', 'password');

my $create = <<"END";
CREATE TABLE address (
    name varchar(24),
    cellphone varchar(12),
    company varchar(24),
    title varchar(12)
);
END

$dbh->do($create) or die "can't create";    # 先把資料表格建起來

my $sql;
$sql = "INSERT INTO address VALUES ('王小明', '0911111111', '甲上資訊', '專案經理')";
$dbh->do($sql);
$sql = "INSERT INTO address VALUES ('李小華', '0922222222', '乙下軟體', '業務經理')";
$dbh->do($sql);
```

3. 從資料庫中取出所有資料，並且利用fetchrow\_array的方式逐筆印出資料。

解答：和新增資料不同，一般我們要從資料庫抓資料出來，都會先使用prepare，然後execute之後才取得資料內容。所以寫法和剛剛會有不少的差別。

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

use strict;
use DBI;

my $dbh = DBI->connect('dbi:mysql:database=perlbook', 'user', 'password');
```

```

my $sql = "select * from address";
my $sth = $dbh->prepare($sql);
$sth->execute;                                # 先取得所有的內容
while (my @result = $sth->fetchrow_array) {    # 逐筆取出
    print "姓名：$result[0]\n";
    print "電話：$result[1]\n";
    print "公司：$result[2]\n";
    print "職稱：$result[3]\n";
}

$dbh->disconnect;

```

4. 呈上題，改利用fetchrow\_hashref進行同樣的工作。

解答：在這裡，我們只需要修改while迴圈內的程式碼。將原來使用fetchrow\_array的部份改成使用fetchrow\_hashref就可以了。當然，因為fetchrow\_hashref拿到的是一個雜湊參照，所以我們得先解開參照，然後取得其中的值。

```

while (my $result = $sth->fetchrow_hashref) { # 逐筆取出
    print "姓名：$result->{name}\n";
    print "電話：$result->{cell phone}\n";
    print "公司：$result->{company}\n";
    print "職稱：$result->{title}\n";
}

```

第十六章：

1. 以下是一個HTML頁面的原始碼，試著寫出action中指定的print.pl，並且印出所有欄位中，使用者填入的值。

```

<HTML>
  <HEAD>
    <TITLE>習題</TITLE>
  </HEAD>
  <BODY>
    <FORM ACTION="print.pl" METHOD="POST">
      姓名：<INPUT TYPE="text" NAME="name"><BR/>
      地址：<INPUT TYPE="text" NAME="address"><BR/>
      電話：<INPUT TYPE="text" NAME="tel"><BR/>
      <INPUT TYPE="submit">
    </FORM>
  </BODY>
</HTML>

```

解答：基本上，這個題目我們想要的就是取得使用者輸入的內容，所以利用CGI模組就可以簡單的做到這件事。

```

#!/usr/bin/perl -w

use strict;
use CGI;

```

```
my $q = CGI->new;
print "姓名：". $q->param(' name' ). "\n";
print "地址：". $q->param(' adress' ). "\n";
print "電話：". $q->param(' tel ' ). "\n";
```

2. 承上題，試著修改剛剛的print.pl，並且利用Template模組搭配以下的模板來進行輸出。

```
<TABLE>
  <TR><TD>姓名：</TD><TD>[% name %]</TD></TR>
  <TR><TD>地址：</TD><TD>[% address %]</TD></TR>
  <TR><TD>電話：</TD><TD>[% tel %]</TD></TR>
</TABLE>
```

解答：這裡的主要工作就是把Template的物件建起來，這樣一來，我們就可以使用Template::Toolkit來建立漂亮的模板。我們假設把上面的模板存成template.html。

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

use strict;
use Template;
use CGI;

my $q = CGI->new;
my $config = {
    INCLUDE_PATH => './',
    EVAL_PERL    => 1,
};

my $template = Template->new($config);
my $vars = {
    name    => $q->param(' name' ),
    address => $q->param(' address' ),
    tel     => $q->param(' tel ' )
};

my $temp_file = 'template.html';
my $output;
$template->process($temp_file, $vars, $output)
    || die $template->error();

print $output;
```

3. 承上題，將利用Template輸出的部份改為HTML::Mason。

解答：我們假定各位的HTML::Mason都設定完成，也就是其實目前都可以執行HTML::Mason的相關程式。因此我們接下來需要的只是處理這一頁的Mason程式。

```
<TABLE>
  <TR><TD>姓名：</TD><TD><% $name %></TD></TR>
  <TR><TD>地址：</TD><TD><% $address %></TD></TR>
  <TR><TD>電話：</TD><TD><% $tel %></TD></TR>
</TABLE>
```

```
<%args>
$name
$address
$tel
</%args>
```

## 第十七章：

1. 找出maillog中被reject(退信)的資料，也就是找到日誌檔中以reject標明的內容。例如：

```
Jun  3 00:00:46 dns2 postfix/smtpd[71431]: D988D6A: reject: RCPT from
smtp2.wanadoo.fr[193.252.22.29]: 450 <fnatterdobkl@hcchien.org>:
User unknown in local recipient table; from=<>
to=<fnatterdobkl@hcchien.org> proto=ESMTP helo=
<mwinf0203.wanadoo.fr>
```

解答：對於系統的日誌檔而言，其實最有利的大多還是格式的固定(規則)化。所以我們可以比較容易的處理這些日誌檔，進而用比較輕鬆的方式取得我們需要的資料。在這裡，我們發現郵件伺服器的日誌檔格式是以':'來作為區隔。所以如果我們把每一筆資料(一行)視為一個字串，利用split來將字串切開為包含各欄位的陣列的話，我們就發現陣列的第三個元素就可以用來判斷是否為退信的資料，因此這樣就顯得容易多了。讓我們來試試看：

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

use strict;

my $file = "/var/log/mail.log";
open LOG, $file;
while (<LOG>) {
    my @columns = split /:/, $_;
    print $_ if ($columns[2] eq 'reject');
}
close LOG;
```

2. 承上題，統計當月每天的退信數字，並且畫成長條圖。