# 第十八章

# 殼(Shell)程式設計

殼(shell)的行為並不侷限於單獨解譯指令而已，還包含了本身整組的內部指令，因為它有自己的變數(variable)、條件子句(conditional)及迴圈(loop)，所以可將其視為一種語言(language)。它大部分的概念源自於C語言，但更為簡潔，而其易於結合外部UNIX指令的特性，使得shell程式變得非常強而有力。

為了能夠好好使用shell程式所帶來的便利性，我們需要兩個章節來更詳細地討論shell程式設計的特色，而本章著重於討論Bourne shell，它是所有shell的基礎，而在下一章則將討論另外兩個較進階的shel，包括Korn及bash shell，當然，本章所討論到的概念也適用於這兩個shell。另外，由於C shell使用完全不一樣的程式設計結構，我們將放在附錄A個別討論。

shell程式執行於解譯模式(interpretive mode)，也就是一次解譯一個敘述句(statement)，所以執行速度比使用高階語言(high-level language)撰寫而成的程式慢，但速度在大多數的工作裡面並不是最重要的因素，使用shell還是有其優點存在，尤其是系統管理這項苦差事，因此，UNIX系統管理者必須是熟練的shell程式設計者。

## 目　標

- 學習執行指令手稿(shell script)的各種方式。(18.2)
- 使用read接收從鍵盤及檔案之輸入。(18.3)
- 了解命令列參數如何轉送至指令手稿，並在指令手稿中當成位置參數(positional parameter)來解譯。(18.4)
- 學習指令之離開狀態(exit status)及參數**$?**的意義。(18.5)
- 利用**&&**及**||**運算子(operator)來執行單一行的條件子句。(18.6)
- 使用if敘述句及作為其控制指令(control command)的UNIX指令來執行多向分支(multiway branching)。(18.8)
- 使用test來比較字串、整數，以及測試檔案屬性。(18.9)
- 使用case的樣式比對(pattern matching)功能，再配合萬用字元來比對字串。(18.10)

- 使用expr來計算數字及處理字串。(18.11)
- 學習如何設計指令手稿，以執行時手稿名稱的不同來執行不一樣的任務。(18.12)
- 使用while及until迴圈控制來重覆執行指令。(18.14)
- 使用for迴圈控制來針對每一元素執行一組敘述句。(18.16)
- 使用basename來更改檔案副檔名(extension)。(18.16.4)


Note

**本章著重於Bourne shell的討論與範例，但實際上亦可應用於Korn及bash shell，但幾乎均不應用於C shell。使用bash時有兩件事要注意，首先，當使用跳脫符號 \c及\n時，必須加上echo -e，其次，$0這個特殊的變數，與Bourne及Korn中的對應部分之意義是不同的，在本章接下來的討論中，要特別注意這些事項。**

## 18.1 shell變數

shell變數在第八章已介紹過，是一種方便儲存數值的方式，在本章中，將使用於指令手稿。讓我們回顧一下，一個變數名稱必須由英文字母開頭，且只能由英文字母、數字字元及底線所組成，用等號(=)指定變數值時，不需加上$，但計算(evaluate)其值時則必須在變數前加上$：

```
$ fname=profile
$ echo $fname
profile
```

unset可將一變數從shell中移除，而只要將變數一個個放在一起就可將其值連接起來；此連接動作和其他程式語言不同的地方在於，它不需要任何運算符號(operator)：

```
$ x=foo ; y=.doc                          一行中可指定多個變數值
$ z=$x$y                                  由其他變數值之組點來指定變數值
$ echo $z
foo.doc
```

shell使用另一種表示法來計算其值，即以一對大括號將變數括起來，例如以下為計算**fname**變數值的另一種方式：

```
$ echo ${fname}
profile
```

shell通常需要以這種形式來計算變數值以利各方面之使用，然而，變數與字串的連結不能使用兩變數連結的方式，所以，要將**x**置於一變數後面的話，須使用下列兩方式之一：

```
$ echo ${fname}x
profilex
$ echo $fname"x"                          單引號亦可
profilex
```

這種方式至少有個好處，使用者可直接修改一檔名即可得到另一組檔名，尤其在更改副檔名時更為好用。

指令代換(command substitution)及相關shell變數之使用，通常可幫助產生簡潔俐落的指令手稿，而大括號的使用，可將3.7節之多重指令順序重寫為如下之敘述句：

```
boldon=`tput smso` ; boldoff=`tput rmso`
echo ${boldon}Come to the Web$boldoff
```

這指令將文字“**Come to the Web**”以粗體顯示於螢幕上，但要注意此處第一個變數需加上大括號，而第二個則不用。當使用這種方式將tput之相關指令設定於變數後，接下來就可以直接在指令手稿中使用這些變數。

## 18.2　指令手稿

理論上，所有shell敘述句及UNIX指令均可在命令行執行，但是，當必須時常重覆執行一群(group)指令時，將之儲存於檔案是較好的選擇，所有這種檔案稱為（指令手稿**shell scripts**）或shell程式，它們沒有強制一定要加何種副檔名，但通常都使用**.sh**，這樣在使用萬用字元來比對手稿名稱時會較方便。

以下之script.sh檔案包含如下之指令，其中有四個為echo：

```
$ cat script.sh
# Sample shell script
echo “The date today is `date`”              #使用指令替換(command substitution)
echo Your shell is $SHELL
echo Your home directory is $HOME
echo The processes running on your system are shown below:
ps
```

使用者可使用vi或emacs來建立這個指令手稿，注意此處之註解符號(**#**)，它可以置於一行中之任意處，而其後所有文字均會被忽略。

有兩種方式可以用來執行這個檔案，而最常用的方式則是，在執行前先用chmod將此檔案設為可執行檔：

```
$ chmod +x script.sh ; ls -l script.sh
-rwxr-xr-x   1 romeo    dialout       154 Feb 25 22:33 script.sh
```

此時這個指令手稿已擁有執行之權限，直接在命令行輸入檔名就可執行：

```
$ script.sh
The date today is Fri Feb 25 22:35:49 IST 2000
Your shell is /bin/sh
Your home directory is /home/romeo
The processes running on your system are shown below:
```

```
PID TTY STAT TIME COMMAND
  192   1 S    0:00 sh
  588   1 R    0:00 ps
```

所有敘述句已依序執行，這個指令手稿實在是太簡單了，所以不需解釋就看得懂，它沒有輸入、沒有命令行參數，也沒有控制結構(control structure)，接下來，我們會逐步將上述功能加入未來介紹之指令手稿中。

指令手稿會執行於一個別的shell，即次要殼(sub-shells)，當一指令手稿正在執行時，使用ps指令會顯示至少兩個sh程序正在執行（在其他shell可能是ksh或bash），其中，登入的shell是正在執行此指令手稿之sub-shells的父殼(parent shell)。如果在指令手稿中執行像是while這種迴圈，則會製造出更多的sub-shells，但只要記住一件事，當一指令手稿正在執行時，至少會有兩個shell process在執行。

現在考慮在執行指令手稿時要使用哪一個shell？使用者自然而然會想要使用現在登入的shell，但假如登入的是 Korn shell，而想執行的指令手稿是撰寫於Bourne shell呢？使用者可能會嘗試以如下方式直接執行：

```
script.sh                                          或是 ./script.sh
```

這方式會呼叫Korn的sub-shells來執行，並且很可能產生錯誤。但既然在各系統通常都會有Bourne shell，可以使用如下方式來呼叫：

```
sh script.sh                                       明白地指出所引用的sub-shells
```

這裡使用指令手稿檔名當作sh指令的參數，sh會開啟此檔案，並依序執行裡面所有的敘述句，此時，這個檔案不再需要可執行權限，而既然是呼叫一個sub-shells來執行，當然使用者仍處於原來的shell之中。


Note

**上述引用檔名來執行指令手稿的方式，只有當現在目錄是定義於PATH中才能執行，假若沒有的話，有一種不必加上絕對路徑的方式也可以，即使用./foo。**

**假如使用者使用vi編輯器來編輯shell及perl之指令手稿，那麼不用離開編輯器也可以執行指令手稿，只要以如下方式在$HOME/.exrc檔案中設定好*[F1]*功能鍵即可：**


Tip

```
:map #1 ^[:w^M:!%^M
```

**這個設定可在執行檔案(:!%)前先將緩衝區的資料儲存起來，^[及^M分別代表*[Esc]*及*[Enter]*鍵(4.3.5)，此時可按下*[F1]*鍵來執行目前的指令手稿（但必須先使用chmod來設定此指令手稿檔案為可執行）。如果這個指令手稿必須輸入參數的話，可用同樣的對應方式來設定*[F2]*鍵，只需移除最後的^M，並在按下*[F2]*鍵時，在最後一行輸入參數。**

指定解譯器　在指令手稿中也可指定欲使用的shell，所以，除了執行sh *filename*之外，最好的方式就是在scrip最前面一行放入以下的敘述句：

```
#!/bin/sh                                          假設 sh是在 /bin
```

shell將#視為註解符號，但假若其後緊接著!的話，shell會將其後的字串解讀為解譯器之指定(interpreter specification)，現在就可以執行指令手稿而不用明確的呼叫sh了。所以，假若一指令手稿只能在特定的shell下才能正確執行，則最好的方式就是在指令手稿最前面直接指定解譯器（在輸入任何敘述前)。


Note

**有些UNIX系統中，C shell在選擇shell來執行指令手稿的方式有點不同，即使登入的shell為csh，但仍須在指令手稿最前面加入下面的敘述句：**

```
#!/bin/csh
```

**所以為了安全考量，每一指令手稿前面最好還是加上這一行指定解譯器的敘述句。**

## 18.3 read：使指令手稿能雙向溝通(interactive)

read敘述句是shell的內部工具程式，它能接受使用者的輸入，也就是說，它使指令手稿能雙向溝通。它使用了一個或多個變數，而從標準輸入(standard input)讀取的輸入值則儲存於這些變數中。當使用者使用了如下的敘述句時：

```
read name
```

指令手稿執行到此敘述句就會暫停，以等待從鍵盤的輸入，而使用者不論輸入了哪些值，都會儲存於變數name中，因為這是指定變數值的形式，所以不需加上$。這裡所使用的第一個指令手稿檔案emp1.sh，會使用read來讀取搜尋字串及檔名：

```
$ cat emp1.sh
#!/bin/sh
# Script: emp1.sh - Interactive version
# The pattern and filename to be supplied by the user
echo "Enter the pattern to be searched: \c"                    # 換行可用\n
read pname
echo "Enter the file to be used: \c"
read flname
echo "Searching for $pname from file $flname"
grep "$pname" $flname
echo "Selected lines shown above"
```

在指令手稿中，#可放在任何地方，而先前也介紹過\c的意義(8.5)，執行時此指令手稿會暫停兩次以等待輸入：

```
$ emp1.sh
Enter the pattern to be searched: director
Enter the file to be used: emp2.lst
Searching for director from file emp2.lst
```

```
9876|bill johnson     |director |production|03/12/50|130000
2365|john woodcock    |director |personnel |05/11/47|120000
Selected lines shown above
```

首先，指令手稿會要求輸入一樣式(pattern)，此時輸入**director**，shell會將它指派給變數**pname**。接著，它要求輸入檔名，此時輸入**emp2.lst**，**shell**會將它指派給變數**flname**。接收完畢後，會執行grep及echo指令。

在按下*[Enter]*前，一行中可以輸入兩個參數，不過在使用read時，就要加上兩個變數：

```
echo "Enter the pattern and filename: \c"
read pname flname
```

Note

**如果輸入的參數個數少於變數個數，則剩下的變數仍然未接受任何值，反之，如果輸入的參數個數大於變數個數，則多餘的參數會指定給最後的變數。**

### 18.3.1 將讀取輸入轉向(redirecting)

上述為互動式之指令手稿，使用read從標準輸入讀取參數值，有一個問題是，指令手稿不能被導向而直接讀取檔案作為其輸入嗎？要回答之前，先看看以下檔案，它將先前輸入的兩個參數儲存於檔案中：

```
$ cat list
director
emp2.lst
```

現在，執行指令手稿檔案emp1.sh，但將其導向並直接從這個檔案讀取輸入參數：

```
$ emp1.sh < list
Enter the pattern to be searched: Enter the file to be used:
Searching for director from file emp2.lst
9876|bill johnson     |director |production|03/12/50|130000
2365|john woodcock    |director |personnel |05/11/47|120000
Selected lines shown above
```

這次一樣顯示兩個提示子句，但並不暫停以等待輸入，並且已用非互動式地完成了一次雙向互動的指令手稿。這項技術對於執行擁有固定回應集(fixed set of responses)之表單驅動(menu-driven)程式是很有幫助的，使用者可讓程式讀取儲存於檔案之回應，而不必直接從鍵盤讀取。

既然如此，那麼vi可以直接從檔案讀取其所有指令嗎？畢竟vi也是從標準輸入讀取指令啊！

## 18.4 位置參數

指令手稿接受其他方式輸入的參，亦即數直接從本身的命令列，C及perl語言也使用這種方式，事實上，只要是用C語言寫的UNIX工具程式都是用這種方式。

這種非互動式指定命令列參數的方法，構成了工具程式發展的基礎，並且可被使用於轉向及管線。

指令手稿指定參數後，會將其分派給某些特殊的變數(variable)，這些稱為位置變數**(positional parameter)**。shell所讀取的第一個參數會分派給**$1**變數，第二個給**$2**...等等。在計算一變數值時，變數名稱前面要加上**$**，雖然這些位置變數前面也有個**$**，但實際上，以**$1**為例，卻沒有一個變數名稱叫做**1**，所以技術上這些不能稱為shell變數。

現在我們要重寫先前之指令手稿以接受命令列參數，在指令手稿中使用這些特殊參數的方式如下：

**$1**——第一個參數。

**$2**——第一個參數。

**$0**——指令手稿之名稱。

**$#** ——參數數目。

**$*** ——一個字串，儲存所有位置參數的集合。

上述有三個特殊的參數，包括**$0**, **$#**及**$***，在本章其他地方尚會用到，但在指令手稿檔案emp2.sh中，首先使用下列這種比較被動的方式：

```
$ cat emp2.sh
#!/bin/sh
echo " Program: $0 "                                    # $0 包含程式名稱
echo " The number of arguments specified is $# "
echo " The arguments are $* "                           # 所有參數均儲存於$*
grep " $1 " $2
echo " \nJob Over "
```

在討論links的時候(7.13.1)，有介紹使用不同名稱來呼叫一程式的方法，這裡的**$0**就是所使用之指令手稿的名稱。shell利用**$#**來計算參數的數目，所以在指令手稿中就可以判斷所輸入之參數數目是否正確。使用下述方式執行此指令手稿，並且加上兩參數，即**director**為樣式、emp1.lst為檔名，其結果如下：

```
$ emp2.sh director emp1.lst
Program: emp2.sh
The number of arguments specified is 2
The arguments are director emp1.lst
1006|gordon lightfoot|director |sales      |09/03/38|140000
6521|derryk o 'brien  |director |marketing |09/26/45|125000

Job Over
```

以這種方式來指定參數，則第一個字(word)會指定給**$0**（指令本身），第二個字會

指定給**$1**（第一個參數），第三個字則指定給**$2**（第二個參數）。這種方式可持續指定位置參數到**$9**（加上shift敘述句的話，還可以更多）。

因為指令手稿接受兩個參數，如果要搜尋的樣式包含多個字的話，例如**robert dylan**，該怎麼做？方法就是用引號括起來，shell把它們當成一個參數看待：

```
$ emp2.sh "robert dylan" emp1.lst
Program: emp2.sh
The number of arguments specified is 2
The arguments are robert dylan emp1.lst
5678|robert dylan    |d.g.m.    |marketing |04/19/43| 85000

Job 0ver
```

**$#**的值一樣為**2**，而如果不使用引號的話，參數的數目會變成**3**，grep就會將**dylan**視為檔名，這個指令手稿就會因產生錯誤而終止執行。


Note

**使用$0可讓一指令手稿知道自己的名稱，但如果一指令手稿檔名含有**links**，則可依據所引用的名稱來做出不同的反應，在學習如何使用**case**敘述句後，就可知道如何利用這項功能了。**


BASH Shell

在bash中，**$0**的定義稍有不同，例如在上述例子中，它會顯示**./emp2.sh**而非**emp2.sh**，所以此指令手稿在接下來使用**$0**來比對時，必須先將**./**移除。我們稍後會學到該如何處理。

## 18.5 指令的離開狀態(exit status)

當grep指令無法比對成功時(15.2.2)，我們稱這個指令執行失敗。每一指令都可能執行失敗，使用者應該要知道執行失敗所產生的影響。在先前章節中，已使用了很多UNIX的指令，有些依照使用者所希望的方式來執行，但有些卻沒有。例如以下的指令：

```
$ cat foo
cat: can't open foo
```

這個指令執行失敗，並且回應一錯誤訊息(error message)，也許是因為沒有這個檔，或是它無法讀取。另外，grep如果無法比對成功的話，它只是回到提示符號下，而沒有產生任何訊息。

每一指令執行完畢時，都會傳回一個值，這個值稱為離開狀態**(exit status)**，或是指令的傳回值**(return value)**，如果指令執行成功，則這個值為真(true)，如果執行失敗，則這個值為假(false)。先前的cat指令因為沒有執行成功，所以稱為傳回一false的離開狀態。

這個傳回值對於程式設計者而言是相當重要的，因為可利用它來設計程式的邏輯，當一指令成功或失敗時，會跳到不同的路徑來執行不同的反應動作。例如，在一個script的執行過程中，如果一重要檔案不存在，或是無法讀取的話，此指令手稿就無法繼續執行，所以shell就提供了test敘述句來測試傳回值。

### 18.5.1 $?參數

shell所使用的另外一個特殊參數為**$?**，它儲存了最後一個指令的離開狀態，如果指令執行成功的話，它的值為0，而失敗的話則為一個非0的值。例如，如果grep比對失敗的話，傳回值為1，但假如是檔案無法讀取的話，則傳回值為2。任何狀況下，只要傳回值大於0的話，就解譯為指令執行失敗。現在讓我們使用另外一種方式來執行：

```
$ grep director emp.lst >/dev/null; echo $?
0                                          比對director成功
$ grep manager emp.lst >/dev/null; echo $?
1                                          比對manager失敗
$ grep manager emp3.lst >/dev/null; echo $?
grep: can't open emp3.lst
2
```

這裡要注意的是，所有位置變數及特殊變數其值的設定都是由shell自動完成的，使用者不能自行修改，除非使用間接的方式，但由於它們在很多方面有極大的用處，並一再的使用於指令手稿中，所以了解它們的用法是有其必要性。表18.1整理並列出了這些變數，其中有兩個先前已介紹過，另有一個待會兒會介紹到。


Tip

**要找出一指令是否執行成功，只要在指令後面使用**echo $?**，0代表成功，而其他值則代表失敗。**

## 18.6 邏輯運算子(logical operator) &&及||——有條件的執行(conditional execution)

emp1.sh這個指令手稿檔案在比對失敗時，仍然會顯示“Selected lines shown above”的訊息，那是因為沒有使用grep的離開狀態來控制程式執行的流程。shell提供了兩個運算子來允許有條件的執行，即&&及||，其語法如下：

```
cmd1 && cmd2
cmd1 || cmd2
```

**表 18.1 shell所使用的特殊參數**

| shell之參數 | 意 義 |
|---|---|
| $1, $2, etc. | 位置參數 |
| $* | 所有位置參數的集合，並以一字串來表示 |
| $# | 指令列參數的數目 |
| $0 | 所執行指令的指令名稱 |
| $@ | 和$*相同，除了必須以雙引號括起來 |
| $? | 最後指令的離開狀態 |
| $$ | 目前shell的PID(10.3) |
| $! | 最後放入背景執行工作的PID(10.10.2) |

&&運算子分隔了兩指令，*cmd1*執行成功的話，才會接著執行*cmd2*，它可使用下列方式來搭配grep指令之執行：

```
$ grep ‘director’ emp1.lst && echo “pattern found in file”
1006|gordon lightfoot|director |sales      |09/03/38|140000
6521|derryk o’brien  |director |marketing |09/26/45|125000
pattern found in file
```

||運算子則相反，只有當*cmd1*執行失敗的話，才會接著執行*cmd2*。如果使用者從一個檔案“grep”一樣式，而想在執行失敗時顯示這項失敗訊息的話，可以這樣做：

```
$ grep ‘manager’ emp2.lst || echo “Pattern not found”
Pattern not found
```

在本章後半段，使用者在設計指令手稿時，會常常用到這些簡潔的運算子(operator)，awk(16.4)及perl(20.7)也會用到它們。

## 18.7 exit：指令手稿之終止(termination)

有時候使用者會測試程式是否有執行失敗的狀況，一旦失敗的話，就希望直接結束程式，不要再繼續執行，因為可能某個重要的資源（例如想要搜尋的檔案）已經找不到了，exit敘述句就是用來提前結束程式。當一指令手稿執行到這個敘述句時就會停止，並將控制權交回呼叫此指令手稿的程式，而大部分狀況下都是shell。在第一章中，就使用了這個敘述句來登出系統。

在指令手稿最後並不需要加上exit指令，因為shell會知道一指令手稿何時執行完畢。而當一指令有機會執行失敗時，就會常常使用這個指令。現在就修改先前的emp2.sh程式，在比對成功時會顯示結果，而當比對失敗時，則終止程式：

```
$ cat emp2a.sh
#!/bin/sh
echo "Program: $0"                                    # $0 為程式名稱
echo "The number of arguments specified is $#"
echo "The arguments are $*"                           # 所有參數儲存於$*
grep "$1" $2 >patlist 2>/dev/null  || exit 2          # 結束程式離開時，傳回一參數
echo "Pattern found -- Contents shown below"
cat patlist
```

當grep指令執行失敗時，程式會終止，但執行成功時則顯示內容。錯誤訊息會轉向至**/dev/null**，所以即使**$2**所代表的檔案不存在的話，grep的錯誤訊息也不會出現於螢幕上。待會兒會討論exit的參數，但先來執行這個scrip，並讓它搜尋一個實際上不存在的檔案：

```
$ emp2a.sh manager emp3.lst
Program: emp2a.sh
The number of arguments specified is 2
The arguments are manager emp3.lst
```

最後的echo敘述句並未執行，而**patlist**檔案的內容也未顯示，這表示程式提前終止了。但它的傳回值是什麼呢？只要檢查**$?**的值：

```
$ echo $?
2                                                     因為執行了 exit 2
```

這是在指令手稿中exit之參數所指定的結果，這個參數不是必要的，但假如在指令手稿中指定它的話，一旦此指令手稿由於執行這一敘述句而終止了，就會傳回這個指定的值。如果沒要傳回任何值的話，**$?**會顯示0，即真值(true value)，所以即使程式執行失敗，使用者還是可以自訂傳回一真值! UNIX就是這麼富有彈性，它可以讓使用者控制任何傳回值。

以這些方式來使用這些**&&**及**||**邏輯運算子，已經進入shell之條件語句(conditional)的領域了，這些運算子有其使用上的限制，所以一般只用來做一些簡單的判斷，如果需要做較複雜的判斷時，就要使用if敘述句。

## 18.8 if條件語句(conditional)

if敘述句可以依據某些狀況來執行兩方向(two-way)的判斷，在shell中，這個敘述句使用下面的格式，就像使用於其他語言一般：

```
if command is successful
then
   execute commands
else
   execute commands
fi
```

Form 1

```
if command is successful
then
   execute commands
fi
```

Form 2

```
if  command is successful
then
     execute commands
elif command is successful
then...
else...
fi
```

Form 3

在BASIC中，if會跟隨著一個then。它會依據條件句之command執行的結果，如果command成功的話，則執行接下來的指令，反之如果command失敗的話，則執行else敘述句（如果有的話），但這個敘述並不一定要有，就如同上述的Form 2。每一個if會以一個fi來代表結束，如果沒有的話，就會產生錯誤。

使shell程式設計之功能變得如此強大的原因，是因為在UNIX程式中，每一指令執行成功與否取決於它的傳回值。所有指令都會傳回一值，就像使用cat及grep時所看見的一樣，所以使用者可以想像shell程式設計可以帶領我們到何種境界！

在下個例子中，grep首先被執行，接著if會使用其傳回值來控制程式流程：

```
$ cat emp3.sh
if grep " ^$1 " /etc/passwd 2>/dev/null                    # 在每行前面搜尋使用者名稱
   then
       echo " Pattern found - Job Over "
   else
       echo " Pattern not found "
fi
```

這是簡單的*if-else*結構，if測試grep的傳回值，而其他部分的意義就相當明顯了。這次，我們要搜尋**/etc/passwd**來看看firewall這個使用者是否存在：

```
$ emp3a.sh firewall
firewall:x:41:31:firewall account:/tmp:/bin/false
Pattern found - Job Over
```

這看起來不像是一個尋常的帳號，因為它的家目錄在**/tmp**，而其shell則為**/bin/false**，但現在先不用煩惱這件事。現在可以先測試else敘述句的部分，試著輸入一不存在於檔案中之樣式：

```
$ emp3.sh mail
Pattern not found
```

為什麼我們要使用grep來搜尋樣式呢？為什麼不使用awk？在何種情況下才能比對特定欄位的樣式？現在先試著將指令手稿中的grep指令置換為awk：

```
$ cat emp3a.sh
#!/bin/sh
if awk -F: ' $1 ~ ' /$1/ ' { print } ' /etc/passwd 2>/dev/null ; then
       echo " Pattern found - Job Over "
   else
       echo " Pattern not found "
fi
```

注意這裡有兩個**$1**，第一個是awk的欄位識別符號(field identifier)，第二個則為指令手稿的第一個參數。先前(16.9)曾討論過，指令手稿參數需要以單引號括起來，如此awk才能正確解譯。then的位置也改變了，它可以與if位於同一行，但必須以分

號(;)隔開。現在就以一個存在的使用者帳號來測試這個指令手稿：

```
$ emp3.sh firewall
firewall:x:41:31:firewall account:/tmp:/bin/false
Pattern found - Job Over
```

程式執行沒有產生任何問題，其結果也在預料之中，但如果使用下列方式的話，可能就會大吃一驚：

```
$ emp3a.sh mail
Pattern found - Job Over
```

**mail**這個字串明明就不存在，awk應該找不到它，可是為什麼沒有傳回false的離開狀態呢？if偵測不到錯誤狀態(error condition)，所以產生了錯誤的判斷。那麼如何才能產生錯誤狀態呢？是可以計算awk輸出的字元數，並用wc來檢查awk製造了什麼輸出嗎？

```
if awk -F: ‘ $1 ~‘ /$1/ ’ { print } ’ /etc/passwd | wc ; then
```

壞消息是即使wc無法計算出任何值，它也會回值一真(true)值，然而，也有個好消息，我們可以檢查wc所計算的字元數來看看是否真的為零。我們先學習如何處理數字(number)後，再來討論這個問題。

### 18.8.1　if-elif：多向分支跳躍(multiway branching)

我們已看過if條件句的兩個形式，包括*if-then-fi* 及 *if-then-else-fi*。現在介紹第三個形式，即*if-then-elif-then-else-fi*。在這個形式中可以有很多個elif，而else仍是可選擇的(optional)。現在就來使用這個形式從henry, romeo及juliet的crontab檔案中搜尋是否有安排執行某一指令：

```
$ cat cronfind.sh
#!/bin/sh
crondir=/var/spool/cron/crontabs
message=” has scheduled the $1 command ”
if grep “ $1 ” $crondir/henry ; then
   echo “ henry $message ”
elif grep “ $1 ” $crondir/romeo ; then
   echo “ romeo $message ”
elif grep “ $1 ” $crondir/juliet ; then
   echo “ juliet $message ”
else
   echo “ None of the users is using the $1 command ”
fi
```

這裡我們正確且有效地使用了shell的變數，例如使用**$crondir**來記錄crontab目錄，縮短了程式碼的長度，因為這項基本訊息在每一搜尋動作都會被重複呼叫，所以將

這種共同訊息轉換儲存於變數中，是很有道理的。

如果發現磁碟有不尋常及特別長的動作，就可使用這個指令手稿來找出這三人之中，到底是誰安排在crontab檔案中執行一find /（從根目錄開始搜尋）指令：

```
$ cronfind.sh "find */"
59 16 * * * find      / -name "*.html" -print |mail romeo
romeo has scheduled the find */ command
```

romeo故意在find及/間放置了很多空格(whitespace)，想要設法不讓管理者察覺，但管理者很聰明的使用正規表示法，在**find**及**/**間加入了*來比對空格。需要再次提醒空格後加上*****代表**0**或多個空格嗎？

所有這些指令手稿都有一嚴重的缺點，就是它們都不指出為何一樣式無法比對成功。即使檔案不存在，還是會出現**not found**及**None of the users**訊息，而使用**2>**將診斷訊息轉向，也確保grep及awk出現之錯誤訊息不會出現於螢幕。理想的指令手稿應該在開始搜尋字串前，就要檢查此檔案是否存在，待會兒就會討論這個問題。

在if敘述句後面的條件句可稱為控制指令(**control command**)，任何UNIX的指令都可成為if、while或until結構的控制指令，確實是很驚人的能力！


Note

**每一if都必須伴隨著then及fi，而else則是選擇性的。else子句(clause)可能包含其他巢狀的if 敘述句(else if)，這裡的每一個if仍須要以fi做為結尾，而elif子句則不需要，所以它產生的程式碼較簡潔。**

## 18.9 test及[ ]：if的同伴(companion)

當使用if來計算表示式時，通常會使用test敘述句作為控制指令。test使用一些運算子來計算其右邊的條件句，並傳回真或假(false)的離開狀態，if會使用這個值來做判斷。test的用途如下：

- 比較兩數字。
- 比較兩字串，或檢查一字串是否為空字串(null)。
- 檢查檔案的屬性。

test也可結合shell其他的敘述句來做測試，但目前我們只討論它如何和if相互運用。test並不顯示任何輸出，只傳回一值給**$?**參數，在接下來的章節裡，會討論如何檢查這個傳回值。

### 18.9.1 數字的比較

test所使用的數字比較運算子（表18.2）與其他程式所使用的形式有所不同，它們都是由連字號(-)開頭，其後接著兩個字元，前後以空白字元做為分隔，下面是

這些運算子的典型例子：

-ne　　*不等於*

**表** 18.2　test**所使用的數字比較運算子**

| 運算子 | 意　義 |
|---|---|
| -eq | 等於 |
| -ne | 不等於 |
| -gt | 大於 |
| -ge | 大於或等於 |
| -lt | 小於 |
| -le | 小於或等於 |

這些運算子相當好記，**-eq**即是equal to（等於），**-gt**即是greater than（大於）……等等，這些意思一看就知道。要注意的是，shell中的數字比較只限制於整數，小數部分會忽略不計。

下面例子指定了三個變數，並且用來檢查相不相等。最後一個測試證明了數字比較只限於整數：

```
$ x=5; y=7; z=7.2
$ test $x -eq $y ; echo $?
1                                                    不相等
$ test $x -lt $y ; echo $?
0                                                    真(True)
$ test $z -gt $y ; echo $?
1                                                    7.2不大於 7!
$ test $z -eq $y ; echo $?
0                                                    7.2等於 7!
```

在單獨使用過test後，現在使用它來做為if的控制指令。接下來的指令手稿使用test及特殊的shell變數**$#**來測試一條件句，這個指令手稿只檢查是否輸入正確的參數數目：

```
$ cat arg_number_check.sh
if test $# -ne 3 ; then
    echo "You didn't enter three arguments"
else
    echo "You entered the right number"
fi
```

現在，試著使用一個及三個參數來執行這個指令手稿：

```
$ arg_number_check.sh 1024
You didn't enter three arguments
$ arg_number_check.sh /home list.tar 1024
You entered the right number
```

現在已經知道如何使用test來比較數字，另外，再來看看之前使用awk比對樣式的問題。下面使用test的敘述句可以行得通嗎？

```
if test awk -F: '$1 ~ '/$1/' { print }' /etc/passwd | wc -c -ne 0
```

此處由於test的參數太多了，所以會產生錯誤，而wc也產生錯誤，因為它將**-ne**解譯為不合法的選項(option)。這裡，我們要檢查的是這個管線(pipeline)的輸出（一個數字），而非它的傳回值，所以必須使用指令代換來輸出wc所計算的字元數，接著使用test來檢查這個值。下面這一行才是正確的：

```
if test `awk -F: '$1 ~ '/$1/' { print }' /etc/passwd | wc -c` -ne 0
```

這裡使用反引號來包住管線，但它並不在螢幕上顯示所選取的行，如果想要看得到的話，可以使用下面的指令手稿來解決：

```
$ cat emp3b.sh
#!/bin/sh
if test $# -ne 1; then                                  # 如果輸入的參數不為1的話
    echo "Usage: $0 pattern"; exit 3
else
    if test `awk -F: '$1 ~ '/$1/'' /etc/passwd \
    | tee /dev/tty | wc -c` -ne 0                       # 也顯示在螢幕上
    then
        echo "Pattern found - Job 0ver"
    else
        echo "Pattern not found"; exit 2
    fi
fi
```

這裡使用了兩個if的巢狀結構，其中一個包含於另一個之中，並且有各自的fi。這裡使用awk的方式又有點不同，{ print }敘述句不見了，因為awk在預設上如果有指定選取條件的話，就會印出結果，而使用tee指令一樣可以在螢幕上顯示結果。注意此處\將這個有三個指令的管線分成兩行。wc -c會檢查awk的輸出（經tee處理後），計算結果為零的話，表示找不到樣式。這個指令手稿現在就可正常執行了：

```
$ emp3b.sh
Usage: emp3b.sh pattern
$ emp3b.sh image
image:x:502:100:The PPP server account:/home/image:/bin/ksh
```

```
Pattern found - Job 0ver
$ emp3b.sh mail
Pattern not found
$ echo $?
2
```

現在，我們已經可以使用if加上一指令及指令代換（需使用test），來處理使用grep及awk時可以做搜尋的工作。

test的速記法**(shorthand)**　由於test很廣泛地被使用，它也有一個執行的速記方法，就是使用一對方括號來包住敘述句，因此，以下兩個形式是相等的：

```
test $x -eq $y
[ $x -eq $y ]
```

這兩個方括號[及]在其內側必須加上空格。第二個形式比較容易處理，接下來都會這樣來使用，但別忘記要加上空格！


Note

**大部分的程式語言都有一特色，就是可使用像是**if x**這種條件句，這裡的x是一個變數。如果x大於**0**，這個敘述句稱為真**(true)**。我們也可應用相同的邏輯，並使用** if [ $x ] **做為**if [ $x -gt 0 ] **的速記法。**


Tip

**雖然在處理數字測試時，沒有使用引號來括住以**$**為字首的變數名稱，但在使用字串時就要特別注意。如果這個變數只包含單一個字**(word)**，則沒有使用引號也可以，但如果包含多個字時，沒有使用引號就會導致指令手稿執行失敗。此外，如果是空字串的話，不使用引號也會導致錯誤的發生。**

**總之，養成使用引號的習慣，就不會有機會發生類似的錯誤事件。相同的，將字串用引號括起來，也會方便變數的比對，例如“**$file**”=“**j**”。**

### 18.9.2　字串比對

test可以用來比對字串，但所使用的是另一組運算子。相等是以=來表示，而不相等是以C的!=來表示。就像其他的test運算子一樣，這些運算子在兩邊都要加上空白。表18.3列出了test用於比對字串的運算子。

下一個指令手稿對於C及Java程式設計者是很有用的，依據所使用選項之不同，它將最後修改的C或Java程式儲存於變數**file**中，然後再編譯這個程式。執行這個指令手稿時，需要加上一個參數，即檔案的型態，可以是**c**（C檔案）或**j**（Java檔案）。

```
$ cat compile.sh
#!/bin/sh
if [ $# -eq 1 ] ; then
     if [ $1 = "j" ] ; then
          file=`ls -t *.java | head -1`
          javac $file
     elif [ $1 = "c" ] ; then
          file=`ls -t *.c | head -1`
          cc $file && a.out
     else
          echo "Invalid file type"
     fi
else
     echo "Usage: $0 file_type\nValid file types are c and j"
fi
```

javac及cc分別是Java及C程式的編譯器(compiler)。這程式一開始時，先檢查**$1**來看看是否只有一個參數，不是的話，就會顯示訊息並且結束程式。這裡甚至使用cc的離開狀態來執行**a.out**，它是C的編譯器預設上會產生的可執行檔。現在就來執行這個指令手稿：

**表** 18.3 test**所使用的字串測試**

| 測試 | 測試為真的狀況 |
|---|---|
| *s1* = *s2* | 字串*s1* = *s2* |
| *s1* != *s2* | 字串*s1*不等於*s2* |
| *stg* | 字串*stg*有設定且不為空值(null) |
| **-n** *stg* | 字串*stg*不為空字串 |
| **-z** *stg* | 字串*stg*為空字串 |
| *s1* == *s2* | 字串*s1* = *s2*（只在Korn及bash中使用） |

```
$ compile.sh
Usage: compile.sh file_type
Valid file types are c and j
$ compile.sh c
hello world
```

最後修改的C程式實際上包含一printf敘述句，並且回應最有名的一句話，即**hello world**。那麼，如果讓指令手稿本身自己來判斷何者是最後修改的程式，並且選用最合適的編譯器，使用者就不需要提供任何參數，這樣不是好多了嗎？答案是肯定的，但是要等到我們已學習如何使用case敘述句以後才行。

現在來看看如何檢查輸入的是空值，可以使用下面的指令手稿來進行字串比對，它會檢查使用者實際上是輸入了一字串，或是直接按下*[Enter]*鍵：

```
$ cat emp4.sh
#!/bin/sh
echo "Enter the string to be searched: \c"
read pname
if [ -z "$pname" ] ; then                                    # -z 檢查是否為空字串
    echo "You have not entered the string" ; exit 1
else
    echo "Enter the file to be used: \c"
    read flname
    if [ ! -n "$flname" ] ; then                              # ! -n與 -z一樣
        echo "You have not entered the filename" ; exit 2
    else
        grep "$pname" "$flname" || echo "Pattern not found"
    fi
fi
```

test可使用!運算子來加以否定。這個指令手稿在兩個點暫停，第一在讀取樣式時，其次在讀取檔名時。注意可使用兩種方式來檢查空字串：

```
[ -z "$x" ]
[ ! -n "$x" ]
```

這的確是兩種不同的方法，但說的都是相同的事。如果其中有一個輸入是空字串的話，則這個指令手稿就會中斷執行：

```
$ emp4.sh
Enter the string to be searched: director
Enter the file to be used: [Enter]
You have not entered the filename
$ emp4.sh                                                    再執行一次
Enter the string to be searched: director
Enter the file to be used: emp1.lst
1006|gordon lightfoot|director |sales     |09/03/38|140000
6521|derryk o'brien  |director |marketing |09/26/45|125000
```

test也允許在同一行執行多個條件句判斷，只需使用-a(AND)及-o(OR)運算子。現在就可以使用這個功能來簡化先前的指令手稿，移除第一個if結構，並在已接收這兩個字串後，使用如下的敘述：

```
if [ -n "$pname" -a -n "$flname" ] ; then
     grep "$pname" "$flname" || echo "Pattern not found"
else
     echo "At least one input was a null string" ; exit 1
fi
```

只有當兩個變數均為非空字串時，test之輸出才為真，也就是說，使用者在指令手稿兩次等待時，輸入了一些非空白字串。

### 18.9.3 test：檔案測試

test可用來測試各種檔案特徵，例如，使用者可以測試一檔案是否擁有讀取、覆寫或執行的權限。除了perl之外，在其他程式語言是不會找到如此精巧的測試語法（表18.4），不是沒有這項功能，就是使用了大量的句子才能達到相同的功能。

test所使用之檔案測試語法是很精簡的，現在就在提示符號下測試檔案**emp.lst**的一些屬性：

```
$ ls -l emp.lst
-rw-rw-rw-   1 romeo    group        870 Jun  8 15:52 emp.lst
$ [ -f emp.lst ] ; echo $?
0                                                          普通檔
$ [ -x emp.lst ] ; echo $?
1                                                          非執行檔
$ [ ! -w emp.lst ] || echo "False that file is not writable"
False that file is not writable
```

**表** 18.4 test**之檔案相關的測試**

| 測試 | 測試為真的狀況 |
|---|---|
| **-f** *fname* | *fname*存在，並為一普通檔案 |
| **-r** *fname* | *fname*存在，並為可讀檔 |
| **-w** *fname* | *fname*存在，並為可寫入的檔案 |
| **-x** *fname* | *fname*存在，並為可執行檔 |
| **-d** *fname* | *fname*存在，並為一目錄 |
| **-s** *fname* | *fname*存在，且檔案大小為大於零 |
| **-e** *fname* | *fname*存在（只使用於Korn及bash） |
| **-u** *fname* | *fname*存在，並設定了SUID位元 |
| **-k** *fname* | *fname*存在，並設定了**sticky**位元 |
| **-L** *fname* | *fname*存在，並為符號鏈結 (symbolic link)（只使用於Korn及bash） |
| *f1* -nt *f2* | *f1*比*f2*還新（只使用於Korn及bash） |
| *f1* -ot *f2* | *f1*比*f2*還舊（只使用於Korn及bash） |
| *f1* -ef *f2* | *f1*鏈結至*f2*（只使用於Korn及bash） |

**!**將測試結果否定，所以[ **! -w file** ]否定了[ **-w file** ]。使用這項功能可以設計一指令手稿來接受檔名做為參數，並且執行一些測試：

```
$ cat filetest.sh
#!/bin/sh
if [ ! -f $1 ] ; then
    echo "File does not exist"
elif [ ! -r $1 ] ; then
    echo "File is not readable"
```

```
elif [ ! -w $1 ] ; then
    echo "File is not writable"
else
    echo "File is both readable and writable"
fi
```

使用兩個檔名來測試這個指令手稿，一個不存在，另一個則存在：

```
$ filetest.sh emp3.lst
File does not exist
$ filetest.sh emp.lst
File is both readable and writable
```


Tip

**有些人使用test時，誤用了shell的萬用字元，例如，不能使用[ -w inde*.html ]來測試一檔案是否可寫入。**

## 18.10 case條件語句

case敘述句是shell所提供的第二個條件語句，在大多數語言（包括perl）中都沒有與它相似的功能。這個敘述句實際上將一表示式與多個樣式比對，亦即使用簡潔的結構來執行多向分支跳躍。它使用萬用字元來比對字串，使得這指令成為不可或缺的字串比對工具。以下是其語法：

```
case expression in
     pattern1) commands1 ;;
     pattern2) commands2 ;;
     pattern3) commands3 ;;
               . . . . .
esac
```

case首先比對*expression*與*pattern1*，如果比對成功，就執行*commands1*，它可以是一或多個指令的組合。如果比對失敗，則繼續比對*pattern2*，…諸如此類。每一指令串列以一對分號(;)為結束符號，而整個結構以esac（case的反向）為結束。

現在，就使用一簡單的指令手稿來顯示檔案系統一些重要的資訊，它有四個選項，並使用多行的echo敘述句來顯示這些資訊：

```
$ cat filesys.sh
#!/bin/sh
tput clear
echo "\n  1. Find files modified in last 24 hours\n  2. The free disk space
  3. Space consumed by this user\n  4. Exit\n\n SELECTION: \c"
read choice
case $choice in
   1) find $HOME -mtime -1 -print ;;
   2) df ;;
   3) du -s $HOME ;;
   4) exit ;;
   *) echo "Invalid option"
esac
```

case比對了**$choice**的值及**1**、**2**、**3**及**4**這四個字串，前三個選項分別執行find、df及du指令，它們的意義先前已介紹過，而選項**4**則會跳離程式。這四個如果都不對不成功，則最後的選項(*****)就會比對成功。接下來我們就會好好利用這項功能。

如果想要找出使用者所有檔案佔據了多少磁碟空間，可以執行這個指令手稿，並選擇選項**3**：

```
$ filesys.sh
  1. Find files modified in last 24 hours
  2. The free disk space
  3. Space consumed by this user
  4. Exit
SELECTION: 3
269440  /home/sumit
```

在使用者的家目錄下，sumit使用了這麼多區塊（每一區塊為512個位元組）的磁碟空間。相同的邏輯可應用於if敘述句，但使用case明顯地精簡多了。現在就來看看其他的一些功能。

### 18.10.1 比對多個樣式

case使用像是egrep之方式可比對多個樣式，如果有一檔案備份計劃表，它指定在每週三或週五做完整的備份工作，而其他日子則只做新增部分之檔案備份，那麼，case就可提供一相當精簡的結構來執行這個邏輯。因為date指令之輸出可被“cut”縮短為三個字元之星期名稱表示式，所以在下面的指令手稿中，就可使用指令代換來提供這個字串給case使用：

```
$ cat back.sh
case `date | cut -d"“  -f1` in                          # 輸出三個字的日期表示式
    Wed|Fri) tar -cvf /dev/fd0 * ;;
          *) find . -newer .last_full_backup_time -print > tarilist
             tar -I tarilist -cvf /dev/fd0 ;;              # -I 選項使用於 Solaris
esac
```

date輸出之第一個欄位顯示了星期名稱，使用cut可將之取出為三個字的日期表示式，來提供作為case的輸入。第一個選項顯示兩個樣式，就像egrep及awk一樣，case在比對多個樣式時，也使用**|**作為樣式之分隔符號。這個選項將星期名稱與**Wed**或**Fri**做比對，並執行tar指令，我們並不需要煩惱其他的日期，因為*****會比對其他的日期，亦即除了先前這兩個日期之外，其他的日期都會與*****比對成功。

這個指令手稿在任何日期都可執行，而正確的指令會自動被執行。在第22.10.2節會討論tar如何來完成完整及新增部分檔案的備份工作。

現在考慮另一個例子，程式設計者常常會遇到詢問使用者是或否（**y**及**Y**，或**n**

及N）的問題，要使用if來執行這項邏輯，就必須使用如下的複合條件句：

```
if [ "$choice" = "y" -o "$choice" = "Y" ]
```

case使用y|Y這麼精簡的表示式來處理這種情況，它可比對大寫或小寫：

```
echo "Do you wish to continue? (y/n): \c"
read answer
case "$answer" in
    y|Y) ;;                                    空的敘述句，沒有執行任何動作
    n|N) exit ;;
      *) echo "Invalid option" ;;
esac
```

### 18.10.2　萬用字元：case也使用它們

case使用萬用字元可擁有極佳的字串比對功能，它使用檔案比對中介字元*、?及字元類別(character class)(8.2)，但只用於字串比對，而非現在目錄的檔案。先前之例子在經過修改case結構後，可讓使用者使用多個方式來回答問題：

```
case "$answer" in
       [yY][eE]*) ;;                           比對YES, yes, Yes, 等等
        [nN][oO]) exit ;;                      比對NO, no, nO 及 No
              *) echo "Invalid response"       當所有比對均不成功時
esac
```

前兩個選項之萬用字元的使用是相當精簡了吧！注意*出現於兩個選項中，但其意義有點不同。在第一個選項中，是一個正常的萬用字元，但在最後的選項中，它提供了所有其他未比對成功選項之收容所。注意最後的case選項不需要;，但想要的話也可以加上去。

使用????可以比對一包含四個字元的字串，如果它必須只能包含數字的話，則要使用[0-9][0-9][0-9][0-9]才對。以下是比對六個數字字元的方法：

```
n="[0-9][0-9][0-9][0-9][0-9][0-9]"
echo "Enter a date string\c "
read dstring
case $dstring in
  ???????*) echo "String exceeds six characters" ;;
        $n) echo "A six character numeric field" ;;
         *) echo "The string is either non-numeric"
            echo "or less than six characters long" ;;
esac
```

現在我們已擁有功能強大的字串比對功能，就可以修改compile.sh這個指令手稿(18.9.2)讓它的功能更強一些。經過修改的程式可以編譯最後修改的C或Java程式，並自動選擇編譯器：

```
$cat compile2.sh
file=`ls -t *.java *.c 2>/dev/null | head -1`
case $file in
       *.c) cc $file && a.out ;;
    *.java) javac $file ;;
         *) echo "There's no Java or C program in the current directory "
esac
```

第一個敘述句就已完成了大部分的工作，它選擇了最後修改的**.c**或**.java**檔案，將它儲存於變數**file**中。接下來是一簡單的case敘述句，它比對了副檔名，並呼叫相對的編譯器。這程式只有短短六行程式碼，難怪使用者實在脫離不了UNIX的魅力！


Note

**case使用*的方式有兩種，每次使用時都很像萬用字元。其中，嵌在一樣式中的*可比對任何數目的字元，但case最後選項之單一個*，只要是前面選項比對不成功的部分，永遠會在此比對成功。**

## 18.11 expr：執行計算及字串處理

Bourne shell可檢查一整數是否大於其他整數，但卻不能執行計算功能，它必須依靠外部指令expr。這指令同時包含兩個功能：

- 執行整數的算術運算。
- 處理字串。

我們將使用expr來執行這兩個功能，但在字串處理部分之程式碼，其可讀性並不高。如果使用Korn shell或是bash，就會有更好的方法來處理這些事情(19.8)，但必須了解的是，在Bourne中還是必須要面對這些無奈的事情，因為很可能使用者必須要找出別人程式的錯誤地方，而這些程式使用了一些expr的敘述句。

### 18.11.1 算術運算功能

expr可以執行基本的四個算術運算，以及計算餘數的功能：

```
$ x=3 ; y=5                                          多個指定敘述句
$ expr 3 + 5
8
$ expr $x - $y
-2
$ expr 3 \* 5                                        星號必須加以跳脫
15
$ expr $y / $x
1                                                    忽略了小數部分
$ expr 13 % 5
3
```

這些運算元之中的+、-、*...等等，其兩邊必須加上空格，並注意相乘符號(*)必須加上跳脫字元，以防止shell將之解譯為檔案中介字元。因為expr只能處理整數，所以使用除法時，只能得到整數部分。

expr通常會加上指令代換功能來設定一變數，例如，可將z的值設為其他兩數字的和：

```
$ x=6 ; y=2 ; z=`expr $x + $y`
$ echo $z
8
```

或許expr最常用來增加一變數的值，所有程式語言都有一速記法來處理這一件事，而UNIX自然也擁有自己的方法：

```
$ x=5
$ x=`expr $x + 1`                                    和 C的 x++相同
$ echo $x
6
```

如果使用Bourne shell的話，在很多指令手稿中就必須採用expr。

### 18.11.2　字串處理

雖然 expr的字串處理功能實在不怎麼精緻，但Bourne shell的使用者實在是沒什麼其他的選擇。處理字串時，expr使用兩個以冒號分隔的表示式，要處理的字串放在冒號(:)的左邊，而右邊則放置一正規表示式。依據正規表示式的性質，expr可以執行下列三個重要的字串處理功能：

- 計算字串長度。
- 取出一子字串(substring)。
- 找出一字串中，某個字元所在的位置。

字串長度　計算字串長度相對之下較為簡單，正規表示式 .*告訴expr它必須列印出符合樣式的字元個數，它有效地簡化為整個字串的長度：

```
$ expr " robert_kahn " : ' .* '                      注意：兩邊的空白
11
```

這裡的expr已計算出所有出現字元的數目(.*)，有些地方與使用grep及sed之輸出相當不同。這項功能在確認輸入資料時相當有用，例如，如果使用者想要確認一使用者從鍵盤輸入的名字，使它不超過二十個字元，則下面的expr敘述句就相當有用：

```
echo " Enter your name: \c "
read name
if [ `expr " $name " : ' .* ' `  -gt 20 ] ; then
   echo " Name too long "
fi
```

取出子字串　expr可用來取出用跳脫字元 \( 及 \) 包住的子字串，例如希望從擁有四個數字字元的字串中，取出代表年份的兩個數字字元，則必須使用下面的方式來設計一樣式組(pattern group)，以取出此子字串：

```
$ stg=2001
$ expr "$stg" : '..\(..\)'                                    取出最後兩個字元
01
```

注意這個樣式組 \(..\)，它就是在sed(15.12.2)中所使用的標籤化正規表示式(tagged regular expression，TRE)，但其意義卻有點不同。它表示在**$stg**之值當中，最前面兩個字元必須忽略，而從第三個位置（這裡不使用\1及\2）開始的兩個字元，則必須被取出（不只是記下來而已）。

找出字元所在的位置　expr傳回在某一字串中，一字元第一次出現的位置。例如，想要找出**$stg**之值中字元**b**的位置，就必須先找出在字元**b**出現前，已出現了多少個其他非**b([^b]*)**的字元：

```
$ stg="paul_baran"
$ expr "$stg" : '[^b]*b'
6
```

expr也使用了test敘述句的一些功能，即運用相同方式來使用數字比較的運算子。但此處並不討論這些功能，因為test是shell內建的功能，所以執行速度快多了。Korn shell及bash也有內建的計算及字串處理功能，它們並不使用expr，在下一章中將會討論。

## 18.12 $0：使用不同名稱來呼叫指令手稿

在討論links時(7.13.1)，曾介紹使用不同名稱來呼叫一檔案，並依據名稱之不同來執行不同任務的可能性，實際上，很多UNIX指令就可以這樣子做。現在我們已知如何使用case，以及使用expr來取出一字串，就是可以設計單一指令手稿來編譯、編組或是執行最後的Java程式的時候了。這個指令手稿檔案有三個名稱，但在設計之前，先來了解Java程式設計的一些要點。

Java程式的副檔名為**.java**，當使用**javac** *filename*來編譯時，會產生一個**.class**的檔案，但是在使用java指令來執行程式時，不需要加上副檔名。例如使用者可編譯一程式**hello.java**來產生**hello.class**，並且使用java hello來執行程式。我們必須要在拿掉副檔名時，能夠取出「基礎的」的檔名，而使用expr來處理時，整個流程就變的相當簡單。

在本章一開始時曾談到有關bash在使用指令求**$0**之值時，在指令前面會加上**./**，所以，這個指令手稿也可修改成獨立於shell的程式，只需偵測是為bash時就

將./移除。這個可執行所有功能的指令手稿檔comj如下：

```
$ cat comj
# 使用不同名稱來呼叫的指令手稿
lastfile=`ls -t *.java |head -1`
case $SHELL in
*/ksh|*/sh) command=$0 ;;
     *bash) command=`expr $0 : ' .*/\([^/]*\)' `  ;;                    # 移除 ./
esac
case $command in
        runj) lastfile=`expr $lastfile : '\(.*\).java' `              #移除.java
              java $lastfile ;;
         vij) vi $lastfile ;;
        comj) javac $lastfile && echo "$lastfile compiled successfully" ;;
esac
```

這裡有兩個case結構，第一個從**$0**取出指令手稿的名稱。在Bourne及Korn shell中，除了將**$0**指定給變數**command**外，不做其他任何事。而在bash中，則使用expr指令來移除**./**，expr忽略了**.*/**，它代表所有在/前面的字元，然後取出後面所有不以/為開始的字元，這樣能有效的取出儲存於變數**command**中之指令名稱。

第二個case條件句檢查所執行的程式名稱，注意此處第一個選項(**runj**)取出了基礎檔名(base filename)，並已移除了**.java**副檔名。由於最後修改的檔名儲存於變數**lastfile**中，剩下的工作就變得相當簡單了，唯一要做的就是使用ln指令來建立comj的鏈結(links)：

```
ln comj runj
ln comj vij
```

接著就可執行vij來編輯程式、comj來編譯以及runj來執行編譯好的程式碼。現在我們只執行編譯動作：

```
$ comj
hello.java compiled successfully
```

所有的動作是不是都很簡單呢？在使用vi時，利用兩個功能鍵來對應到這些工作，就會更簡單了，只要在**.exrc**檔案中設定以下兩個項目：

```
:map #1 :w^M:!comj^M                                    先使用 :w^M來儲存檔案
:map #2 :!runj^M
```

當編輯一程式時，可以按下*[F1]*鍵來編譯程式，以及按下*[F2]*鍵來執行。還有什麼事是想要UNIX來幫你完成的呢？

## 18.13 sleep及wait

有時使用者會想要在指令手稿中增加一些延遲時間，以便在指令手稿繼續執行其他工作前，可看見一些顯示於螢幕上的訊息。另外，使用者可能也會想要有規律地（例如每分鐘）檢查一事件的發生（例如一檔案被建立）。sleep是UNIX用來製定延遲時間的指令，使用時會加上一參數，用來描述shell在繼續執行前必須暫停或沈睡(sleep)的秒數：

```
$ sleep 100 ; echo "100 seconds have elapsed"
100 seconds have elapsed
```

在指令執行前，這個訊息會顯示100秒。這指令的特殊功能是，當它在沈睡時，並不會增加系統額外的負擔。

wait是shell內建的功能，它用來檢查是否所有背景執行的程序都已執行完畢，所以當使用者正在背景執行一工作，並想要確定這指令是否已完成，以便能繼續執行其他程式時，這個指令就顯得相當有用。下列wait指令的執行方式，可使用或不使用一程序的PID來等待最後一個背景工作的完成：

```
wait                    等待所有背景程序的完成
wait 138                等待 PID為138的程序之完成
```

## 18.14 while及until：迴圈

到目前為止，我們所發展之樣式掃描 (pattern scanning)的指令手稿，都不提供訂正一個錯誤回應的機會，而迴圈則可讓使用者反覆執行一組命令。shell提供了三個型態的loop功能：while、until及for，這三者使用某些關鍵字來包住想要重複執行的命令，重複的次數則取決於這些控制指令所設定的條件。

while敘述句對於大多數的程式設計者是相當熟悉的，它重複地執行一組命令，直到控制指令傳回一真(true)值的離開狀態。這個指令的一般語法如下：

```
while condition is true
do                          注意 do 的關鍵字
        commands            迴圈的本體
done                        注意 done 這個關鍵字
```

只要*condition*持續為真(true)，則包含於do及done的*commands*就會重覆被執行，使用者可使用先前介紹過之任何UNIX指令或test作為控制指令（此處為條件句）。

我們將由傳統的while迴圈應用來作為開始。**emp5.sh**這個指令手稿在同一行接收一個編碼(code)及敘述(description)，並將它寫入一**newlist**檔案，接著再詢問使用者是否還有項目要輸入：

```
$ cat emp5.sh
#!/bin/sh
# 程式名稱: emp5.sh -- 示範while迴圈的用法
answer=y                                          # 必須設為y以進入迴圈
while [ "$answer" = "y" ]                         # 控制指令
do
    echo "Enter the code and description: \c"
    read code description                         # 兩者一起讀取
    echo "$code|$description" >> newlist          # 將這一行附加於檔案後
echo "Enter any more (y/n)? \c"
    read anymore
    case $anymore in
         y*|Y*) answer=y ;;                       # 以y或Y為開頭
         n*|N*) answer=n ;;                       #以n或N為開頭
             *) answer=y ;;                       # 任何其他輸入則視為y
    esac
done
```

在上述程式中，只要適當地加入一些註解，就不需再解釋太多即可看得懂。現在先來執行它：

```
$ emp5.sh
Enter the code and description: 03 analgesics
Enter any more (y/n)? y
Enter the code and description: 04 antibiotics
Enter any more (y/n)? [Enter]                     不輸入任何字的話，則視為 y
Enter the code and description: 05 OTC drugs      此處有三個字(word)
Enter any more (y/n)? n
```

這是相當簡單的指令手稿，它沒有任何確認功能，不論輸入什麼都會寫入**newlist**檔案中：

```
$ cat newlist
03|analgesics
04|antibiotics
05|OTC drugs
```

### 18.14.1　使用while來等待一檔案

現在來看看一很有趣的while迴圈之應用。假如在某一狀況下，程式*b*需要讀取程式*a*所建立的檔案，只有在此檔案被建立後，*b*才能執行，在接下來的例子中就來設計這個邏輯。

monitfile.sh這個指令手稿週期性地監測磁碟，檢查一檔案（這裡為**invoice.lst**）是否存在，只要這檔案找得到的話，就接著執行alloc.pl這個指令

手稿：

```
$ cat monitfile.sh
while [ ! -r invoice.lst ]                          # 當檔案invoice.lst無法被讀取時
do
   sleep 60                                                    # 每60秒尋找一次
done
alloc.pl                                                  # 離開迴圈後執行這個程式
```

只要讀取不到**invoice.lst**檔案，這個迴圈就會一直執行（**! -r**代表不可讀取），如果這個檔案變成可讀取，則迴圈就會結束，而程式alloc.pl就會被執行。這個指令手稿在理想狀態下可於背景執行：

```
monitfile.sh &
```

我們使用sleep指令每60秒來檢查這個檔案是否已存在。

### 18.14.2 找出使用者之空間使用量

參考22.10.1節可看到du -s /home/*指令的輸出，這指令使用**/home/***做為參數來顯示每一使用者磁碟使用狀況的摘要，以下是幾行範例：

```
166     /home/enquiry
4054    /home/henry
647     /home/image
64308   /home/sumit
```

這裡假設家目錄均放置於**/home**下面。我們可使用while迴圈來讀取這個輸出的每一行，並且將超出一特殊限制（預設值為4000區塊）的使用者，將其訊息寄給root，也就是說，du.sh這個指令手稿不需參數，或可接受一參數：

```
$ cat du.sh
# du.sh - - 用來監測磁碟空間的程式
case $# in
     0) size=4000 ;;                                # 使用者不指定時，所預設的大小
     1) size=$1 ;;                                  # 設定為輸入的值
     *) echo "Usage: $0 [blocks]" ; exit ;;
esac
du -s /home/* | while read blocks user
do
  [ $blocks -gt $size ] && echo "$user has consumed $blocks blocks" \
                          | mail root                          #將列表寄給root
done
```

此處while迴圈從du的標準輸出取得其所要的輸入，並且讀取每一行，將其值設定給**blocks**及**user**這兩個變數。接著它會比較由du而來的計算，不是4000（如果不輸

入任何參數的話)，就是由參數而來的值。在每一行中，每一個別的訊息都會寄給root，執行這個指令的方法有二：

```
du.sh
du.sh 8000
```

***選擇那些超過8000個區塊的使用者***

當root打開他的信箱時，應該會看見所有超出使用限制之每一使用者的訊息，這訊息看起來如下：

```
/home/sumit has consumed 64308 blocks
```

這個指令手稿對於系統管理者是很有用的，因為他必須經常監測磁碟空間，並且找出是誰超出了空間使用的限制。最佳的執行方式就是在crontab工作中設定以下的一行，並在一般上班時間早上十點到下午七點每三小時執行一次：

```
0 10,13,16,19 * * 1-5 /home/admin/scripts/du.sh
```

*crontab**中的一個項目***

這個指令手稿有個缺點，尤其當使用者數目眾多時會更嚴重。雖然它完成了工作，並針對每一使用者傳送了相關的訊息給root，但傳送單一包含所有訊息的檔案會是較理想的。雖然這件事實際上不容易察覺到，但我們仍應該在done這個關鍵字後，將echo的輸出以管線方式傳送：

```
done |mail root
```

在下一章中，會再次看到轉向，但現在這個指令手稿已可滿足我們的需求。

### 18.14.3　設定一無窮迴圈(infinite loop)

如果身為一系統管理者，想要每五分鐘就監測磁碟可用的空間，可以使用無窮迴圈，並且使用一替代指令(dummy command)作為while的控制指令。這個指令並不做任何事情，除了傳回一真值之外。事實上，這個指令的名稱為true，放置於**/bin**中，所以使用者就可以在背景執行以下的迴圈：

```
while true ; do
   df -t
   sleep 300
done &
```

一旦使用者在背景執行這個程式後，就可繼續執行其他的工作，但每五分鐘就可能在螢幕上看見一些由df產生的輸出(6.17)。想要殺掉這個程序的話，可以使用kill $!，這個指令會殺掉最後一個被放入背景執行的程式(10.10.2)。

像true一樣，也有一指令false，它永遠傳回一假值。這兩者都都是UNIX上最

簡單的指令：

```
$ cat /bin/true
$ _                                          這個檔案沒有包含任何東西，但它會傳回真值
$ cat /bin/false
exit 255
```

Linux版本的這兩個指令有一些較不同的功能，但在此並不介紹。只要記得一件事，除非使用者在背景執行這些無窮迴圈，不然就必須按下中斷鍵來結束這些程式。然而，在迴圈中可使用break關鍵字來中斷程式，break及continue這兩個使用於迴圈中的關鍵字在下一節會介紹。

### 18.14.4 until：while的補數(complement)

until敘述句與while結構正好是互補的，只有在條件持續為false的狀態下，它的迴圈本體才會重複執行，這只是使用另一種觀點來看待整個邏輯。這兩種形式可以交替使用，但從其中一個切換到另一個時，其條件表示式必須被加以否定。
有些人喜歡將先前的while控制指令以下面方式來表達：

```
until [ -r invoice.lst ]                     直到 invoice.lst 可以被讀取
```

他們或許是對的，這一行被轉換成這樣的意思：「直到invoice.lst檔案可被讀取…」，這個形式就比較容易理解了。

## 18.15 兩個指令手稿的範例

現在，我們要運用到目前為止已獲得的知識，來發展兩個有趣的指令手稿，一個是cp指令的加強版，當目的檔案(destination)已存在時，會將它加上一數字的副檔名。另一個為輸入資料項目的指令手稿，但它有確認的功能。

在開始從事這些指令手稿前，應該要知道使用於shell迴圈的兩個關鍵字，break及continue，在C及Java語言中也有這兩個相同名字的關鍵字。

continue敘述句會暫停執行其後的所有敘述句，並將控制流程切換到下一次重複迴圈的開始。break敘述句會跳出迴圈，兩者都可使用參數。

### 18.15.1 cpback.sh：使用數字之副檔名來備份檔案

曾經因為不小心，結果使用其他檔案將一重要檔案覆蓋掉嗎？圖18.1所看見的指令手稿檔案cpback.sh可防止一檔案不小心被cp指令覆蓋掉。這個程式的第一個版本接受要複製的檔案名稱，以及一目錄名稱作為它的兩個參數，它會檢查檔案foo是否已存在於目的地，如果有的話，會加上一數字的副檔名，並且從foo.1開

始，而這個檔案也有可能已存在，所以數字會持續增加，直到最後複製時不會覆蓋任何檔案為止。

**圖**18.1　cpback.sh：**用來複製一檔案而不會覆蓋此檔案的指令手稿**

```
# Program cpback.sh -- Copies a file to a directory
# Makes a backup instead of overwriting the destination file
# Copies foo to foo.1 if foo exists or foo.2 if foo.1 exists .......

if [ $# -ne 2 ] ; then                          # 需要兩個參數
    echo "Usage: $0 source destination" ; exit
elif [ ! -d $2 ] ; then
    echo "Directory $2 doesn't exist"
else
    file=$1
    if [ ! -f $2/$file ] ; then                 # 目的地檔案不存在
       cp $1 $2
    else
       copies=1                                 # 開始檢查是否已存在
       while true ; do                          # 數字之副檔名
          if [ ! -f $2/$file.$copies ] ; then
             cp $1 $2/$1.$copies                # 提供一數字之副檔名、接著複製檔案
             echo "File $1 copied to $1.$copies"
             break                              # 完成工作-- 離開迴圈
          else                                  # 如果這個已有數字副檔名的檔案
             copies=`expr $copies + 1`          # 也存在的話，就找下一個
          fi
       done
    fi
fi
```

當檔案被複製時，break敘述句會中斷迴圈的反複，現在我們先建立一目錄，名為**safe**，並且將**vvi.sh**複製到這個目錄，先來試試下面的指令：

```
$ cpback.sh
Usage: cpback.sh source destination
$ cpback.sh vvi.sh safe1
Directory safe1 doesn't exist
$ cpback.sh vvi.sh safe
$ _
```

***檔案已複製為* vvi.sh**

希望這樣已複製了檔案，但為了再確定一下，我們再重複幾次這個複製動作：

```
$ cpback.sh vvi.sh safe
File vvi.sh copied to vvi.sh.1
$ cpback.sh vvi.sh safe
File vvi.sh copied to vvi.sh.2
```

這個指令手稿已正確執行，現在就可使用這個指令手稿在一目錄下儲存所有版本的程式。待會再來修改這個指令手稿，以便它可以處理多個檔案。

### 18.15.2 dentry1.sh：資料項目輸入的指令手稿

接下來，在18.2中的指令手稿檔dentry1.sh，它會從終端機接受一編碼及相對的敘述，執行一些基本的確認檢查，然後再加入一項目至**desig.lst**檔案中，並且會確認所輸入的編碼是否已存在於檔案中。這個指令手稿會重複等待使用者的輸入，直到輸入適當的反應。

這個指令手稿會等待輸入兩個欄位，指定的編碼及其敘述。它使用了兩個迴圈，其中一個包含於另一個之中。如果所輸入的編碼已存在於檔案中，或是不為兩個數字字元的結構，就必須重新輸入。同樣的，如果所輸入的敘述句包含了除了空格以外之非英文字母字元(***[!\ a-zA-Z]***)，也必須重新輸入。continue敘述句可讓您重新輸入資料，或是開始一新的迴圈，而迴圈內部的break敘述句則在加入一行後，會離開迴圈。

在此指令手稿中加入對話後，這個邏輯變得很有說服力：

```
$ dentry1.sh
Designation code: 01
Code exists
Designation code: 07
Description     : security officer
Wish to continue? (y/n): Y
Designation code: 8
Invalid code                                    必須有兩個數字字元
Designation code: 08
Description     : vice president 1
Can contain only alphabets and spaces
Description     : vice president

Wish to continue? (y/n): n
```

使用“ cat ”指令顯示**newlist**檔案內容時，會看見兩個接靠(appended)在一起的項目：

```
$ cat newlist
07|security officer
08|vice president
```

我們已經用了很多方式來使用while迴圈，例如等待一檔案之產生、重複增加一數字及持續提示使用者輸入一有效的項目。這是一個非常重要的結構，而且任何一位熟練的shell程式設計師都得學會去駕馭它。

**圖**18.2　dentry1.sh **之指令手稿檔案**

```
#!/bin/sh

while echo "Designation code: \c" ; do                          # 在Linux中使用-e
    read desig
    case "$desig" in
     [0-9][0-9]) if grep "^$desig" desig.lst >/dev/null ; then  # 如果編碼已存在
                    echo "Code exists" ; continue              # 回到迴圈最前面
                fi ;;
              *) echo "Invalid code"  ; continue ;;            # 不是兩個數字字元的編碼
    esac

    while echo "Description      : \c"   ; do
         read desc
         case "$desc" in
      *[!\ a-zA-Z]*) echo "Can contain only alphabets and spaces" ; continue ;;
                 "") echo "Description not entered" ; continue ;;
                  *) echo "$desig|$desc" >> newlist ; break   # 終止迴圈
         esac
    done

    echo "\nWish to continue? (y/n): \c"
    read answer
    case "$answer" in
         [yY]*) continue ;;                                    # 回到迴圈最前面
             *) break ;;                                       # 終止迴圈
    esac
done
```

## 18.16　for：使用一列表來循環迴圈

for迴圈與其他程式語言在結構上是不同的，對BASIC的使用者而言是一個全新的經驗，它沒有next敘述句，也沒有step的指令規格。和while及until不同之處在於，for並不測試一條件句，但使用了一列表(list)，它的語法結構非常簡單：

```
for variable in list ; do
   commands                                          迴圈本體
done
```

迴圈本體是相同的（關鍵字do及done也相同），但有額外的參數*variable*及*list*。在list中有多少項目，迴圈本體就會被執行多少次。看看下面這個簡單的例子就比較清楚：

```
$ for file in chap20 chap21 chap22 chap23 ; do
>     cp $file ${file}.bak
>     echo $file copied to $file.bak
> done
```

```
chap20 copied to chap20.bak
chap21 copied to chap21.bak
chap22 copied to chap22.bak
chap23 copied to chap23.bak
```

這裡的*list*包含了一些字串（**chap20**等等），它們使用空格來分隔。列表中每一項目都會被指定給一變數**file**，**file**首先會設定為**chap20**，接著**chap21**，諸如此類。每一**file**會複製為加上**.bak**副檔名的檔案，當每一檔案複製完成時，就會顯示完成的訊息。

在命令列也可使用這樣一系列的變數，在執行迴圈前，shell會先指定其值：

```
$ for var in $PATH $HOME $MAIL ; do echo "$var" ; done
/bin:/usr/bin:/home/local/bin:/usr/bin/X11:.:/oracle/bin
/home/romeo
/var/mail/romeo
```

在單一行中，如果想要輸入整個迴圈的話，必須要在適當位置加上分號(;)。上面三行代表了這三個相對之環境變數的值。

這裡也可以使用指令代換來產生這個串列，下面這個for的指令行會從檔案**clist**中取出這個串列：

```
for file in `cat clist`
```

當這個串列很大時，一個個去指定它是很不實際的，所以上面這個方法就很適合，而如果想要改變這個串列時，就不必更改程式。

這個串列還能由其他項目所組成嗎？當然，只要shell能看得懂，且能夠處理的話，它幾乎可以由任何表示式來組成。我們已看見它可由變數及指令代換來產生，在下面的章節中，要使用萬用字元及位置參數來產生這個串列。for在UNIX系統中或許是最常用的迴圈，所以最重要的是使用者能夠完全了解它。

### 18.16.1 從檔名及萬用字元而來的串列

和case一樣，for也可以使用萬用字元，但它實際上是拿來比對現在目錄中的所有檔案。在第一個例子中，我們使用for來複製一群檔案，可以簡單使用下面萬用字元的樣式：

```
for file in chap2[0-3] ; do
```

shell會產生一以字元順序排列的檔案列表，接著使用程式本體中的指令依序來對每個檔案執行動作。以下兩個例子將會示範如何使用for來對一群檔案產生動作。

編譯一群C的程式　因為*可比對任何數目的字元，所以可使用下面的方式來編譯所有的C程式：

```
$ for file in *.c ; do
>     cc -o $file{x} $file                    x 附加於每一物件檔(object filename)之後
> done
```

cc的-o選項可用來選擇輸出的可執行檔檔名，在這裡使用原來的檔名，其後面加上一個x。這個迴圈選取了每一個C程式，例如stringfind.c，接著建立stringfind.cx為其可執行檔。

一群檔案的代換動作　在使用sed來代換一群檔案時，for迴圈是不可或缺的，因為標準輸出被轉向至一個別的檔案，所以可在for迴圈內可執行整個工作，並且把輸出寫回相同檔案中。例如，以下的指令手稿作用於現在目錄中的每一HTML檔案：

```
$ cat lower2upper.sh
for file in *.htm *.html ; do
    sed ' s/strong/STRONG/g
    s/img src/IMG SRC/g ' $file > $$
    mv $$ $file
    compress $file
done
```

這裡的for會取出每一HTML檔案，使用sed執行一些代換動作，將輸出移至一暫存檔$$（目前shell的PID），並且將結果寫回原來的檔案，最後將這些檔案壓縮：

### 18.16.2 從位置參數而來的串列

for最重要的應用之一，就是處理經由位置參數提供之外來資料的能力，下一個指令手稿檔案emp6.sh會針對每一參數來重複掃瞄檔案：

```
$ cat emp6.sh
for pattern in $* ; do
     grep " $pattern " emp.lst || echo " Pattern $pattern not found "
done
```

$*記錄的是所有參數的完整串列，並以空格分隔(18.4)，shell將這些串列解譯為$1、$2…等等。這裡我們並不需要個別運用這些位置參數，迴圈會反複地將每一參數指定給變數。現在就來執行這個指令手稿，並輸入四個參數：

```
$ emp6.sh 2345 1265 4379 367
2345|james wilcox     |g.m.      |marketing |03/12/45|110000
1265|p.j. woodhouse   |manager   |sales     |09/12/63| 90000
Pattern 4379 not found
Pattern 367 not found
```

因為for最常使用**$***（或“ **$@** ”）來存取指令列的參數，所以預設上這個參數為一空白的串列。以下這兩個敘述句意義是相同的：

```
for pattern in $*
for pattern
```

***實際上是***“ $@ ”，***詳見***18.16.3***節意指***“ $@ ”


Tip

**為了可讀性起見，在指令手稿中要使用**for *variable* in **$*而不要使用**for *variable*，**有時過於簡潔反而會引起混淆。下一節會介紹使用“ $@ ”來代換**$*。

### 18.16.3 $@：另一個特殊的參數

emp6.sh這個指令手稿檔案原本使用四個字元之員工代號(empid)來掃瞄**emp.lst**檔案，但如果使用的是名字的話，for迴圈就會產生錯誤。現在使用兩個名字來執行同一指令手稿：

```
$ emp6.sh “robert dylan” “ringo lennon”
5678|robert dylan    |d.g.m.    |marketing |04/19/43| 85000
5678|robert dylan    |d.g.m.    |marketing |04/19/43| 85000
Pattern ringo not found
6213|michael lennon  |g.m.      |accounts  |06/05/62|105000
```

它產生了一些有趣，但是很令人疑惑的輸出，因為for將**robert**視為一個參數，**dylan**為另一個；而**ringo**亦為一參數，**lennon**則為另一個。雖然找不到**ringo**，但卻找到了**lennon**，但它是**michael**的姓。注意有一行出現了兩次。

將**$***用引號括起來（就像for pattern in“ $* ”中的一樣）只會使問題更糟：

```
$ emp6.sh “robert dylan” “ringo lennon”
Pattern robert dylan ringo lennon not found
```

這兩個參數在這裡被“ **$*** ”視為單一參數，而shell最常被忽略的特殊參數—**$@**，即可用來拯救這個緊急的狀況。它的用法和**$***不使用引號時是一樣的，但當它被加上引號時，會將每一用引號括起來的參數視為單一參數，就是我們現在所需要的。將for敘述句更改為這樣：

```
for file in “$@”
```

接著使用先前方式來執行這個指令手稿：

```
$ emp6.sh “robert dylan” “ringo lennon”
5678|robert dylan    |d.g.m.    |marketing |04/19/43| 85000
Pattern ringo lennon not found
```

這個輸出有著重要的涵義，當一指令手稿使用多個字的字串來作為參數時，就必須使用“ **$@** ”，以讓指令手稿之相關程式結構能正確擷取每一參數。這裡會極度建議使用“ **$@** ”，而不要使用**$***，因為“ **$@** ”可以處理**$***所能處理的任何字串。


Tip

**如果在指令手稿中必須使用多個字的字串當作參數的話，就要使用“ $@ ”（括在引號中）而非$*，甚至除此以外，在處理單一字之字串時，“ $@ ”的作用也和 $*一樣。實際上，**for pattern**敘述句實際上指的是**for pattern in “ $@ ”。

### 18.16.4 basename：更改副檔名

這裡我們將要討論另一外部指令basename，因為它用於for迴圈內部時，是相當有效率的，尤其兩者合併使用時，對於更改一群檔案的副檔名是相當有用的。Windows的使用者可能對於UNIX無法接受像下面這樣的敘述句，而感到沮喪：

```
mv *.txt *.doc                          試著將所有副檔名為 .txt 之檔案更改為 .doc
```

在UNIX中需使用指令手稿來達成在Windows下只需一行RENAME指令就能做到的事。我們可使用expr(18.12)來取出一檔案的基礎檔名，但basename指令就能做到相同的事，而且不必使用正規表示式(regular expression)：

```
$ basename /home/henry/project3/dec2bin.pl
dec2bin.pl
```

當basename加上第二個參數時，它會從第一個參數中，刪除第二個參數所代表的字串：

```
$ basename hello.java .java
hello                                                    .java 被刪除了
```

現在就在迴圈中使用這項功能，將副檔名.txt更改為.doc：

```
for file in *.txt ; do
  leftname=`basename $file .txt`                         儲存檔名之左半部
  mv $file ${leftname}.doc
done
```

如果for處理的第一個檔名為seconds.txt，則leftname儲存的是seconds（沒有句點），mv只是將取出的字串(seconds)加上一.doc於其後，這件處理工作並不需要使用expr。

## 18.17 cpback2.sh：最後一個指令手稿

現在知道while及for的用法後，就來設計本章最後一個指令手稿，它用來加強先前之指令手稿檔案cpback.sh(18.15.1)之功能，使之可以接受多個檔名。因為這個指令手稿現在使用了最少兩個參數，最後一個參數必須為目錄名稱。圖18.3顯示的就是這個指令手稿。

從參數串列中，我們必須能夠製造出第二個不含目錄名稱的串列，也就是說，我們必須要能夠重複指令手稿所有的參數，除了最後一個之外。UNIX的shell並沒

有任何一個變數代表最後一個參數，所以我們必須自己使用已知的功能來製造一個過濾器。

**圖18.3 cpback2.sh：用來複製多個檔案而不會覆蓋的指令手稿**

```
#!/bin/sh
# Program cpback2.sh -- Copies multiple files to a directory
# Makes backups instead of overwriting the destination files
# Copies foo to foo.1 if foo exists or foo.2 if foo.1 exists .......

if [ $# -lt 2 ] ; then
    echo "Usage: $0 source(s) destination" ; exit
fi

echo $* | tr ' ' '\012' > $$                  # 將每一參數置於一行
destination=`tail -1 $$`                              # 最後一行為目錄
if [ ! -d $destination ] ; then
    echo "Directory $destination doesn't exist"
else
    count=`expr $# - 1`                     # 將每一參數置於一行
    for file in `head -$count $$` ; do
       if [ ! -f $destination/$file ] ; then
           cp $file $destination                                  #重複次數比參數數目少一
       else
           copies=1
           while true ; do
               if [ ! -f $destination/$file.$copies ] ; then
                   cp $file $destination/$file.$copies
                   echo "File $file copied to $file.$copies"
                   break                                  # 這個動作不會覆蓋檔案
               else
                   copies=`expr $copies + 1`                    #繼續下一數字
               fi
           done
       fi
    done
fi
rm $$                                                     #移除暫存檔
```

為了製造第二個串列，我們使用tr來將參數串列中的空格轉換成換行(newline)字元，它確保每一個手稿參數都放置於獨立的一行並儲存在由**$$**所表示的檔案中，而最後一行就是目錄名稱(tail -1 $$)，所以將它儲存在變數**destination**中。現在將參數數目減1，(**count=`expr $# - 1`**)，然後使用head來處理除了最後一行以外之所有的行(head -$count $$)，並將之提供給for使用。

在工作結束時，別忘了要移除暫存檔(rm $$)，其他的每一件事依然很類似，所

以就來繼續先前的複製練習，並將檔案複製到**safe**目錄中，但這次使用更多的檔名。現在假設一檔案**toc.pl**（但並非**index**這個檔）已存在於**safe**目錄中，接著執行數次下面的指令：

```
$ cpback2.sh vvi.sh toc.pl index safe
File vvi.sh copied to vvi.sh.3
File toc.pl copied to toc.pl.1
$ cpback2.sh vvi.sh toc.pl index safe                    再次執行
File vvi.sh copied to vvi.sh.4
File toc.pl copied to toc.pl.2
File index copied to index.1
$ cpback2.sh vvi.sh toc.pl index safe                    又再一次
File vvi.sh copied to vvi.sh.5
File toc.pl copied to toc.pl.3
File index copied to index.2
```

即使有多個檔名，這個指令手稿照樣能正常執行，現在終於可以複製一或多個檔案，而不必煩惱會覆蓋掉目的地的檔案了！一旦修改了某個程式後，就可以使用cpback2.sh這個指令手稿來將它複製到一目錄中，而數字的副檔名會自動加上去！如果仍不明瞭這些事是如何發生的，請進一步參考19.15節。


Note

**現在已經知道如何應用**while**迴圈來撰寫程式了，例如，可在背景使用它來發出提醒語句，如果使用**vi**時，常忘記要使用**:w**來儲存檔案，就可以使用**while**來在螢幕上發出警告。本章最後有一習題就是寫出這樣的一個程式。**

現在已完成了先前預留作為介紹shell程式設計之兩章其中的一章，現在已可感受到shell程式設計卓越的能力。在繼續介紹下一個更進階的shell功能之前，最好確定已了解本章的每一個細節，包括Korn及bash shell全部的功能。我們將會學習更新的技術，發展更好、更精鍊的程式碼，並完成一些不可能的任務。

## 摘　要

shell也是一程式語言，它使用解譯模式(interpretive mode)來執行指令手稿，每一次執行一行。指令手稿執行的速度比像是C的編譯語言慢，但對於大多數的工作而言，速度並不是那麼重要。

殼變數可使用由大括號包住變數名稱的方式來取出其值，這些括號亦可用來連接變數與字串。

執行一指令手稿前，必須先設定這個指令手稿檔為可執行，也可使用sh指令來執行。在指令手稿檔中，可在第一行加上**#!/bin/sh**敘述句來指定想要引用的shell，如果想使用Korn及bash shell來執行的話，sh也可用**ksh**及**bash**來取代。

read敘述句可讓指令手稿接受從鍵盤而來之輸入，這些輸入會讀至一或多個變數中。包含read敘述句的指令手稿可被轉向而讀取從檔案來的輸入。

在命令列指定好參數後，可非互動式地執行一指令手稿，這些參數會被讀入至$1、$2、…等位置參數中。shell的參數$#儲存了參數的數目，$*包含了所有的參數，而$0則包含了所執行之指令手稿的名稱。

每一指令或指令手稿在結束時，都會傳回一離開狀態（或傳回值），這個值儲存於參數$?中，0代表一真(true)值，而任何非零值則代表執行失敗。

&&及||這兩個運算子扮演了單一方向的條件句，當&&分隔了兩指令時，只有當第一個指令執行成功，才會接著執行第二個指令，而||運算子則正好相反。

exit敘述句會中止一指令手稿，它可加上一數字來代表執行成功或失敗，這個數字會儲存於$?中。

if敘述句有三種形式，並且以fi為結束。它會根據控制指令所傳回的值，來執行一組指令。else及elif可選擇性地用來指定另一組指令，以利先前之控制指令執行失敗時使用，這些控制指令也可以是 UNIX的指令。

test敘述句同等於[]，它可和很多運算子合併使用來比較數字及字串，就和測試檔案屬性的方式一樣。test並不製造任何輸出，它只將結果儲存於$?之中。

case是一精簡的字串比對結構，並以esac為結束。它可使用shell的萬用字元來比對多個egrep類型的樣式，而*使用於最後一個選項時，可比對任何先前未比對成功的樣式，這裡的萬用字元是用來比對字串，而非是檔案。

case特別適合用來比對$0的檔名，它可用來設計一指令手稿，針對所執行之檔名來完成不同的工作。

expr用於整數計算及字串操作，在Bourne shell中也可用來增加一變數的值。它使用正規表示式來取出子字串，尋找一字元的位置，以及計算字串的長度。在Korn及bash中並不需要expr。

while迴圈只有當其控制指令傳回一真值時，才會執行其程式本體。在指令手稿中，它用來反覆增加一變數的值，或是提供使用者多重的選擇機會。使用true作為控制指令的話，也可建立一無窮迴圈。until則是while的互補敘述句。

for一次使用一個串列中的元素，這個串列可以是經由變數、萬用字元、位置參數以及指令代換所產生。您可以在迴圈中使用sed指令對一群檔案進行大量的取代工作，或可用來編譯一群C程式。

所有的迴圈都使用do及done這兩個關鍵字，break敘述句可用來終止一迴圈，而continue則可用來開始一新的循環。

在for迴圈中，當使用多個字之字串作為參數時，應該使用“ $@ ”。而在for迴圈中使用basename則可更改檔案的副檔名。

## 自我測驗

18.1　如果x的值為10，則x$x$及$x$x之值為何？

18.2　在執行一foo.sh的指令手稿時，它的執行結果和在此指令手稿中所描述的不太一樣，為什麼會這樣？需做什麼相關設定才能使它正常執行？

18.3　如果在Korn shell下設計了一指令手稿，如何確保即使登入的shell不同，而這個指令手稿在執行時仍能引用正確的shell？

18.4　如果一個指令手稿呼叫它自己，那將會發生什麼事？

18.5　如果一個指令手稿以foo -l -t bar[1-3]的方式來執行則$#及$*的值為何？如果這些選項結合在一起的話，會發生什麼事？

18.6　什麼是一個指令的離開狀態？它的值正常狀況下為何？而這個值會儲存於何處？

18.7　如果一個檔案沒有包含任何執行動作的話，那傳回值會是什麼？

18.8　使用grep、sed及awk來搜尋一樣式，如果搜尋失敗時請測試其個別的傳回值。請為得到的結果下結論。

18.9　本章所介紹之UNIX的外部指令為何？為何這裡必須介紹這些指令？

18.10　在執行一個指令手稿時，可使用sh < foo.sh來代替sh foo.sh嗎？

18.11　您如何以非互動式地執行emp4.sh這個指令手稿(18.9.2)，並且不需等待輸入就能將其最後結果顯示出來？

18.12　寫出一個可接受包含一串數學運算式的檔案作為參數之指令手稿，然後使用bc以*expression*=*value*的形式來印出這個運算式及其值。

18.13　這個結構式有何功用？為什麼？

```
while [ 5 ]
```

18.14　如何使用for迴圈重覆執行一個指令共十次？

18.15　同上，這個程式的獨特功能為何？

## 練　習

18.1　如何防止一使用者執行一指令手稿？需要移除這個指令手稿檔案的可執行權限嗎？

18.2　如果x的值為5，則以x=" expr $x + 10 " 來重新設定其值的話，x的新值為何？在這個敘述式中，如果使用單引號的話，x的值又為何？這種方式會產生錯誤嗎？

18.3　有一指令手稿名為test，它包含df及du指令，但執行時卻沒有顯示任何訊息，為什麼？敘述兩種可讓這個指令手稿正確執行的方法。

18.4 設計一指令手稿在家目錄下，找出參數所指的檔案，其所有的硬體鏈結 (hard links)檔案，而所提供作為參數的檔案必須是存在於目前目錄下的檔案。

18.5 設計一指令手稿，它可以加上lm參數根據修改時間來列出檔案，並且可加上la參數以存取時間來列出檔案。在執行這個指令手稿前，尚須執行哪些額外動作？

18.6 設計一指令手稿，它可以接受一樣式及一檔名作為參數，並且計算在這個檔案中，此樣式出現的次數。（這個樣式只包含字母、數字符號及底線，且在一行中可以出現好幾次）

18.7 設計一指令手稿依據修改及存取時間邊靠邊(side-by-side)一起列出檔案，且必須列出其權限、檔案大小及檔名。這個指令手稿可接受任何數目的參數來選擇性地顯示檔案，而當所指定的檔案找不到時，必須中斷執行，並須為每一欄位取個標題。

18.8 設計一指令手稿，它可接受一輸入字串，如果此字串少於十個字元，請用case來回應相對的訊息。

18.9 如何使用expr來設計和上題相同的動作？

18.10 使用expr從一檔案的絕對路徑名稱中取出其父目錄名稱。

18.11 以下這個程式至少有六個語法上的錯誤，請找出來。（左邊為行號）

```
1   ppprunning = yes
2   while $ppprunning = yes ; do
3      echo "    INTERNET MENU\n
4      1. Dial out
5      2. Exit
6      Choice:
7      read choice
8      case choice in
9        1) if [ -z "$ppprunning" ]
10            echo "Enter your username and password"
11         else
12           chat.sh
13         endif ;
14       *) ppprunning=no
15     endcase
16   done
```

18.12 使用(i) while 迴圈 (ii) for迴圈每三十秒列出系統中所有執行的程序五次。

18.13 將檔案**msg.lst**的內容傳送給每一個登入的使用者，重覆登入的相同使用者只能接受一個訊息。

18.14 設計一指令手稿，它可以head指令形式來顯示現在目錄每一檔案的最後三行，其前面須加上此檔案的檔名。

18.15 設計一指令手稿，它可以接受一或多個檔名作為參數，並將這些檔名轉為

大寫。

18.16　設計一指令手稿，它可以接受兩個目錄名稱**bar1**及**bar2**，並將**bar2**中與**bar1**中相同檔名及內容的檔案刪除。

18.17　設計一指令手稿，它只允許使用者romeo及henry從螢幕型態為**tty05**及**tty06**之螢幕中執行它。

18.18　從一指令手稿中執行vi，每三分鐘就可聽到一嗶聲，並且可在最後一行看見一訊息，這個訊息以反白顯示，並提醒要儲存緩衝區中的資料。（提示：使用tput來指定游標位置，並在背景設計一迴圈，而在vi執行結束時殺掉這個迴圈。）