第 十九 章

進階的Shell程式設計——包括Korn與bash

在這一章節中，我們將看到shell的進階功能。我們將分析shell和其設計方法所建立的環境處理流程。一些shell的指令執行在次要殼(sub-shells)中，通常無法做到我們想做的事。因此我們將學習將外部的環境變數傳遞到sub-shell和執行命令時不用再產生sub-shell。我們也將討論以前屬於外部命令的部分，但現在被內建在最新的shell中之功能。

本章中，Korn shell與bash有一些重要的功能，它們是取材自現代化的程式語言和處理運算、字串及陣列的能力。Korn的*ksh93*版本提升了shell的地位甚至增加不少驚豔的功能，它讓我們在shell中擁有了許多awk與perl的特色。

在本章中討論到的其他功能在Bourne shell中也同樣有效，在順利讀完本章中所有的範例之後，我們強烈建議您應該轉換成Korn shell和bash。我們將討論Korn與bash的共通功能，並在每一章節的標題中指出其每一個功能。

目　標

- 使用set及shift從單一命令列的輸出來取出區塊。 (19.1)
- 在相同的手稿程式中利用此地文件(here document)特色，放置指令的資料。 (19.2)
- 在Korn shell及bash中使用let來計算。 (19.3)
- 在迴圈與條件句的關鍵字中提供轉向(redirection)。 (19.4.1)
- 合併標準輸出及標準錯誤輸出，用符號1>&2及2>&1表示。（19.4.2和19.4.3）
- 用export讓 sub-shell看見被定義的變數。 (19.6.1)
- 了解()及{}運算符號在程序中使用的意義。 (19.6.2)
- 使用Korn shell及bash所支援的陣列。 (19.7)
- Korn shell及bash內建處理字串的能力。 (19.8)
- 依據變數設定與否，以計算變數值的不同用法。 (19.9)
- 了解shell函數(functions)為何優於別名(aliases)。（19.10和19.11）
- 用eval評估計算指令行兩次並產生一般性的提示符號及變數。（19.12和19.13）
- 用exec置換目前的程式成另外一個及處理多重的資料串流 (multiple streams)。

(19.14)

- 用set -x來做指令手稿的除錯。 (19.15)
- 用trap接收訊號來決定手稿要採取的反應動作。 (19.16)

19.1 set：指定變數值到位置參數

一些UNIX的指令，像date會產生單一列的輸出。其他的命令通常經由過濾器，像grep及head來產生單一列的輸出。有時您需要在這行之中，取出其一個或多個區塊。我們在之前的章節(18.10.1)碰到相同的狀況，當時我們用cut來取出date的區塊做為輸出：

```
case `date | cut -d"" -f1` in
```

呼叫外部命令來取出單一區塊確實有點誇張，而使用shell本身的set命令敘述即可完成此事。這命令執行一個非常簡單的函數：它將位置參數**$1**, **$2**指定給它本身的引數（argument），這個功能在取出一個程式輸出內容的單一區塊時特別有用。使用date與指令代換，您可以將輸出的值指定給位置參數：

```
$ set `date`
$ echo $*
Thu Sep 30 08:27:50 EST 1999
$ echo "The date today is $2 $3, $6"
The date today is Sep 30, 1999
$ echo $#
6
```

當使用set時， **$***及**$#**通常也用一般方式被設定；您不再需要用cut取出任何的區塊；配合set即可輕易的做到。set預設的語法是用空白字串當分隔來搜尋字串值，並將找到的值指定到對應的位置參數中。Bourne shell允許直接存取至九個參數，但Korn及bash卻能存取任何參數。 如果您用set來設定本章全部的內容，您即可存取每一章節中的每一個字。您可以用**${}**符號來取得第十個參數以後的值：

```
$ set `cat ux3rd19`
$ echo $#
20202                                   這樣多的文字出現在本章中
$ echo $4 $5
SHELL PROGRAMMING
$ echo ${12}                            只能在bash及ksh中動作
BASH
```

可能的輸出結果實在太多了。在任何命令或是管線的輸出都可使用set來解析（parse）出個別的字組，並不需用到任何的外部命令。事實上，set並沒有限制要使用空白當作分隔符號，也可以使用shell的變數**IFS**來指定分隔符號。


Caution

如果手稿接受命令列參數配合使用set敘述，不要忘了在使用set前，分別儲存命令列的參數，因為set在它自己的參數上使用相同的標記法，因此會覆寫這些參數。


KORN Shell


BASH Shell

不像在Bourne shell中，他們沒有限制參數的數目，您可以直接存取，舉例來說，您可以使用**${40}**符號來存取第四十個參數：

```
echo ${40}
```

19.1.1　IFS變數：set預設的分隔符號

IFS包含具有字元的字串，它被使用在命令列中當成文字之間的分隔。而此字串通常包含有空白、定位及新增一行的字元，這些字元無法被顯示，因此用echo $IFS會輸出空白行：

```
$ echo $IFS
空白列
```

雖然沒有任何東西顯示出來，但仍有一些東西在那裡，您可以藉由八進位的轉換來確定這些變數的內容：

```
$ echo "$IFS" | od -bc
0000000 040 011 012 012
              \t  \n  \n
0000004
```

間隔、定位及新增一列組成的IFS

空白是由ASCII的八進位值040表示；您可以用echo來看到\t及\n。set使用IFS的值來決定它們的分隔符號。這變數通常不會造成您的困擾，但如果一行的資料必須用其他分隔符號才能解析出來，您就需要暫時改變IFS值。舉例，考慮/etc/passwd檔中的這一行：

```
$ grep henry /etc/passwd
henry:x:501:100:henry blofeld:/home/henry:/bin/ksh
```

如果當您用read指令讀取一個檔案並發現有這樣的一行時，可以在使用set之前先改變IFS值，這樣就可以簡單的取出第六個區塊：

```
$ IFS=:
$ set `grep "^henry" /etc/passwd`
$ echo $6
/home/henry
```

在本章的一些例子中，我們需要改變**IFS**值。set及**IFS**在撰寫 shell script時是個很有用的附屬工具，但他們要變得更好用時，仍要搭配shift。

19.1.2 shift：參數往左邊移動

在很多手稿中，使用第一個參數作為一個特別項目，例如檔案名稱。其他的參數可能表示一組連續的字串，也許是檔案中要被選取的不同樣式。我們知道在一個串列的資料中，如何排除最後的參數(18.17)；這次我們將用shift敘述來排除第一個參數。

shift會立即對降低的位置參數重新命名，每當呼叫一次，**$2**變成**$1**，**$3**變成**$2**，諸如此類。 試試這個位置參數放到date命令列中：

```
$ echo $*
Thu Sep 30 08:27:50 EST 1999
$ echo $1 $2 $3
Thu Sep 30
$ shift                                                    $1被移走了
$ echo $1 $2 $3
Sep 30 08:27:50
$ shift 2                                                  移動2個位置
$ echo $1 $2 $3
08:27:50 EST 1999
```

注意最左邊的參數 **$1**，每次shift執行就會不見了。因此，如果手稿使用十二個參數，您可以移三次，最後只剩下**$1**到**$9**能用。這可能可以設計一個手稿並接受檔案名稱和樣式當作參數：

```
$ cat emp7.sh
# 在手稿使用移動（shift）的功能
case $# in
    0|1) echo "Usage: $0 file pattern(s)" ; exit 2 ;;
      *) flname=$1                                         # 在不見之前先儲存$1變數
         shift
         for pattern in "$@" ; do                          # 我們將使用 "$@" 代替$*
             grep "$pattern" $flname || echo "Pattern $pattern not found"
         done ;;
esac
```

您現在可以使用含有不定數目參數的手稿（不要少於兩個）：

```
$ emp7.sh emp.lst
Usage: emp7.sh file pattern(s)
$ emp7.sh emp.lst wilcocks 1006 9877
3212|bill wilcocks   |d.g.m.   |accounts  |12/12/55| 85000
1006|gordon lightfoot|director |sales     |09/03/38|140000
Pattern 9877 not found
```

注意到此處**flname** 儲存字串**emp.lst**， for迴圈用三個字串**wilcocks,1006**及**9877**來重複。


Tip

每次當您使用shift**時，最左邊的變數就會不見；因此在使用**shift**前，變數應該被儲存起來。如果您必須從第四個參數開始重複，先儲存前三個參數，然後再使用** shift 3。

19.1.3 set --：代換命令的幫手

您經常需要將set與命令代換合用。特別是當命令輸出是由-符號開始時，會有一些小小的問題出現：

```
$ set `ls -l unit01`
-rw-r--r--: bad option(s)
```

因為檔案的存取權限是以- 符號起頭（一般的檔案），set會把它當成選項，並且發現這是「錯誤」的。 set評估一個空字串的參數會產生另一個問題，考慮一下這下面這個命令：

```
set `grep PPP /etc/passwd`
```

如果字串**PPP**不在檔案中，set將沒有任何參數可運作，而且在終端機上會顯示所有變數來（這是他的預設輸出），這會困擾使用者！要解決這些問題，可以在set後面立即使用- -（兩個連字符號）：

```
set -- `ls -l unit01`                    最前面的-已經被處理
set -- `grep PPP /etc/passwd`            空的輸出沒有問題了
```

set現在知道在 - - 之後的參數不會被誤認為選項了。如果參數被計算成空字串時，兩個連字號也會指示 set 抑制它的預設行為。

19.1.4 使用set找出磁碟剩餘空間

注意在6.17節中的df命令輸出，它可以顯示每一個檔案系統中所使用及剩餘的空間。典型的輸出看起來像這樣：

```
/home              (/dev/dsk/c0t0d0s3 ): 107128 blocks   495356 files
```

我們可以設計一個手稿，它可以接受檔案系統裝載目錄(mounting directory)（第一個區塊）當成參數來比對出符合的行及取出其剩餘空間（兩個數字中前面的那個)。當剩餘空間低於某個臨界的數字時，它甚至可以郵寄一個這樣的訊息。df.sh手稿就是做這樣的工作：

```
$ cat df.sh
# df.sh -- 程式用來監控剩餘的磁碟空間
# 使用檔案系統的名稱當作參數
```

```
set -- `df | grep "^$1"`
if [ $4 -lt 200000 ] ; then
      echo "Free space in $1 has dropped to $4 blocks" | mail root
fi
```

在此UNIX系統中，檔案系統被顯示在第一個欄位中，這就是為何grep要使用^來定位樣式。數字在第四個區塊中比較，當檔案系統的剩餘空間低於200,000區塊(blocks)時會送一封信給系統管理者。這手稿可以用裝載目錄的名稱當參數：

```
df.sh /home                                    找 /home檔案系統剩餘空間
```

系統管理者會想重複的執行這個命令手稿，因此最好放置在管理者的crontab檔中，以便由cron來執行。下面是設定在每個工作天的上午九點至下午五點間，每一個小時做一次這個工作：

```
0 09-17 * * 1-5  /home/admin/scripts/df.sh
```

19.2 此地文件(<<)

有時候當程式中所讀取的資料是固定及相當受限制時。shell使用<<符號從包含手稿的相同檔案上讀取資料，這個特性被稱為此地文件**(here document)**，表示資料就在此地不在別的檔案。任何使用標準輸入的命令也可以從此地文件中得到輸入。

這個功能在指令無法接受檔案名稱當參數時非常有用的 （例如像 mail命令)。如果是短訊息（任何正常被預期的郵件訊息)，在同一個手稿中，您能同時將命令和訊息放在同一個script：

```
mail juliet << MARK
Your program for printing the invoices has been executed
on `date`. Check the print queue                    允許命令替換
The updated file is known as $flname                也允許變數估算
MARK
```

此地文件符號(<<)接著三行的資料及分隔符號（字串**MARK**)。手稿將命令之後由**MARK**隔開的每一行當成命令的輸入。在另一端主機上的使用者juliet只會看到這三行的文字訊息； **MARK**這個字本身並不會顯示，因為mail沒有讀取外部檔案，亦即資料全在此地，所以執行較快。


Note

此地文件的內容，被命令當成輸入資料前由shell來詮釋及處理。這意味著您能夠使用命令替換和用變數來輸入，但不能由標準輸入來進行輸入。

19.2.1 在交談程式中使用此地文件

很多命令需要使用者的輸入，通常由相同的輸入裝置來鍵入命令所提問題的回

答。例如，當命令暫停等待輸入時，您可能需要鍵入y兩次或三次，但問題不可能快速的接受到回答而有結果。與其等待提示符號出現，我們可以指示命令手稿從此地文件讀取輸入。

考慮Linux的fdisk命令(21.9.1)，按下內部命令p會顯示檔案的分割區及按下q離開fdisk。身為一個系統管理者，當您做這工作時，會需要確定是否使用正確設備名稱下的磁碟分割區。藉此手稿，您可以做快速的檢查：

```
# cat showpart.sh                                    使用系統管理者的提示符號—#
fdisk << END
p
q
END
```

當您在系統管理者的提示符號下(#)執行showpart.sh時，您會立刻得到分割區的訊息。我們在之前(18.3.1)已做過一次，我們只是再做一次而已；我們讓一個交談程式以非交談方式來執行。典型的fdisk輸出顯示在21.9.1節中。


Tip

如果您寫了一個手稿並使用一個或多個read指令敘述，而且此手稿都用預先定義的回答來執行，您就可以在執行手稿時運用此地文件以非交談方式來執行。這對自動化工作也有幫助。

19.3　let：計算工作——第二次檢視（ksh及bash）

Korn及bash內建有整數的處理機制，所以完全不需要用expr。您可以用let敘述來計算：

```
$ let sum=256+128                                    變數後沒有空白
$ echo $sum
384
```

如果您為了得到更佳的可讀性而加上空白，只要用引號將表示式括起來：

```
$ let sum="3 * 6 + 4 / 2" ; echo $sum
20
```

現在看一下let如何處理變數。首先定義三個變數；單一let做法如下：

```
$ let x=12 y=18 z=5
$ let z=x+y+$z                                       let不需要用到$
$ echo $z
35
```

在一個指定式中let允許您除去全部的$。因為這個計算功能是內建的，所以比在手稿中使用expr執行的更快，之後，我們將於我們的手稿中用let來代替 expr。

用let仍被限定在處理整數運算，然而，*ksh93*版本的Korn shell也可以做浮點數的運算。printf現在已被內建在這兩個shell中，有許多工作我們甚至不需要使用

awk就能做到！

用（(和)）的第二種運算的形式 Korn shell及bash使用(( ))算數符號來取代let 的敘述：

```
$ x=22 y=28 z=5
$ z=$((x+y + z))                                    空白是不重要的
$ echo $z
55
$ z=$((z+1))                                        也能使用z=$((z+=1))
$ echo $z
56
```

POSIX規格中，建議使用（(及)）而不是let，這種形式似乎已變成shell的標準功能，因為變數之前不需要用**$**而變得更容易使用。然而，所有運算的動作之前必須要加**$**。

19.4 轉向——第二次檢視

到目前為止，我們還沒用到轉向。我們之前已經將轉向符號使用於簡單的命令處理，亦即在所有需要它們的指令中使用提供這些符號。我們也將每一個資料流(streams)分開的對待而非把這些資料流結合在一起。在之後的章節中，我們將學習應用轉向技術在更實際的地方，並使用一些新的符號來結合兩個不同的資料流。

19.4.1 在關鍵字處轉向

參考手稿emp5.sh(18.14)，在那裡我們用>>符號於一行中加上新增一列的字元：

```
echo “ $code|$description ” >> newlist
```

當每次echo被呼叫時，將echo的敘述轉向會造成**newlist**檔案的不斷地被開啟，但這樣的工作方式不是很有效率的方法。shell避免類似開啟及關閉多個檔案的動作而在done 關鍵字上提供轉向機制：

```
done > newlist
```

在這裡，檔案的開或關只有一次，但也導致一個嚴重的結果：所有在迴圈內的命令如果使用標準輸出，也會被轉向至**newlist**檔案中。有些命令的輸出仍然需要送到終端機上，因此您應該明確地將他們轉向至**/dev/tty**中，這有一個修訂過的emp5.sh 程式版本：

```
$ cat emp5a.sh
answer=y                                        # 必須設定它為y在第一次進入迴圈前
while [ “ $answer ” = “ y ” ]                    # 控制命令
do
    echo “ Enter the code and description: \c ”  >/dev/tty
    read code description                       # 一起讀取這兩個變數
```

```
    echo "$code|$description"                          # 這裡沒有轉向
    echo "Enter any more (y/n)? \c"   >/dev/tty
    read anymore
    case $anymore in
         y*|Y*) answer=y ;;                              # 任何用y或Y開始的
         n*|N*) answer=n ;;                              # 任何用n或N開始的
             *) answer=y ;;                           # 任何其他的回答方式表示y
    esac
done > newlist                                      # 只有一個echo敘述轉到檔案中
```

在這個script的兩個敘述被轉向至/dev/tty中。即使如果轉向在done關鍵字上，在迴圈中的敘述會全部轉向至標準輸出，而這兩個敘述將會被排除在外，因為我們明白地把它們轉向至終端機上。

轉向也用在fi及esac 的關鍵字上，包含輸入轉向及管線：

```
fi > foo                                    所有的敘述在if和fi之間
esac > foo                                  所有的敘述在case和esac之間
done < param.lst                            整個迴圈從param.lst取得輸入
done | while true                           管線輸出到一個while迴圈
```

我們已私下的在之前的章節中(18.14.2)使用最後一種型式。我們現在要正式使用它。

19.4.2　合併資料流(1>&2)

轉向至/dev/tty 也會有些許的問題，串流（stream）轉向過一次就不能再被轉向。這意味著每次您要輸出到終端機時只要鍵入>/dev/tty這九個字元。shell支援合併串流的功能，在做轉向時會比較容易得多。

這是個簡單的構想，因為標準錯誤輸出串流會輸出在終端機上，原本輸出在終端機上的敘述句可以指示將它們的標準輸出與此串流合併，然後您只需要處理標準錯誤輸出串流以控制這些敘述指令（statement）即可。合併的方式是將&運算子與轉向符號一起使用。當您在手稿中以下面的方法使用echo 敘述指令時：

```
echo "None of the patterns found" 1>&2          >&2一樣也將會執行
```

您實際上是說：「合併兩個串流及傳送合併的串流至標準錯誤輸出，此處預設是終端機」。這說明了如果您轉向迴圈或甚至全部於手稿到一個別檔案去，而此敘述的輸出一定會被顯示在終端機上。因為1是預設標準輸出的檔案描述符號（file descriptor），您也能用>&2來表示。


Note

您可以用一般的方法來轉向合併的串流。當您使用find_number.sh > foo 2> bar，所有的手稿敘述只要有接1>&2符號的，事實上是寫到bar檔案中，其他手稿的輸出則被存到foo檔案中。

19.4.3 儲存錯誤訊息及輸出至同一個檔案中 (2>&1)

您能將標準輸出及標準錯誤輸出串流的角色互換。以下的指令照常合併兩個串流，但這次標準錯誤輸出串流會轉向到標準輸出要去的位置：

```
cat foo1 foo2 foo3 > bar 2>&1
```

當我們使用1>&2或2>&1兩者是做法是否有所不同？是的，這兩者是不一樣的。在這個例子中，如果foo3不能被開啟，cat 命令的錯誤訊息將被儲存到含有foo1 及foo2內容的bar檔案中。我們現在也有儲存錯誤訊息與正常輸出在同一個檔案的能力，如果您了解這方法，您會同意上面的敘述也能被表示成下面的方法：

```
cat foo1 foo2 foo3 2> bar 1>&2
```

對於您是手稿開發者及系統管理員時，這功能蘊涵著很大的意義。假如您不在現場而想執行一個程式時，您就可以轉向手稿讓它的輸出和錯誤訊息存到同一檔案中：

```
content.sh 2>&1 | sort > bar                    將合併的輸出作排序
content.sh 2> bar 1>&2
```

bar現在含有兩個串流的輸出。第一種方式甚至讓您用管線把串流轉向到另一個命令中，如果您使用cron來執行手稿，這也是唯一讓您知道程式的錯誤的方法。您必須了解及記住這點。


Note

當您把符號1>&2或2>&1接在一個使用標準輸出的命令後面時，您只能合併標準輸出及標準錯誤輸出串流，但不能轉向它們。如果您要轉向合併串流，必須提供額外的轉向符號。

19.5 user_details.sh：列出使用者詳細資料的手稿

現在考慮一個迴圈轉向及合併串流的有趣應用，從/etc/passwd及/etc/group檔中列出使用者的詳細資料。以下的Korn shell手稿接受一個範圍的UID當作兩個參數並且從/etc/passwd檔中對應於每一UID讀取其相關的區塊，它同時查詢/etc/group的GUID數字及取出群組名稱。

這個手稿也使用在大部分UNIX系統都有支持的printf命令，它現在內建在bash及 *ksh93*中。printf 沒有新的功能；您已經在awk中使用它來格式化輸出，這手稿 user_details.sh表示在圖19.1中。

您必須記住此處printf 沒有使用到逗號。這兩個參數值被指定給變數uidmin及uidmax。在while迴圈中依序的讀取/etc/passwd的每一行，在read敘述表示式中每一行被分為七個區塊。 用set敘述來搜尋檔案/etc/group中的第四個區塊（群組識別編號：gid)。變數kcount 計算從檔案/etc/passwd中已經讀取的行數。

圖 19.1　user_details.sh**手稿**

```
#!/bin/ksh
case $# in
  2) ;;                                                    # 需要兩個參數
  *) echo "Usage: $0 min_guid max_guid" 1>&2 ; exit
esac

IFS=:
uidmin=$1 ; uidmax=$2              # 在set 取代 $1 和 $2之前，先將它們儲存
echo "\
Username    UID  GUID  Gname    GCOS         Home Directory    Login Shell
------------------------------------------------------------------------"
kount=0
while read username password uid gid gcos homedir shell
do
    if [ $uid -ge $uidmin -a $uid -le $uidmax ] ; then
         set -- `grep ":$gid:" /etc/group`    #  置換先前的$1和 $2
         gname=$1
         printf "%-10s %4d %4d   %-8s %-14s %-14s    %-12s\n" \
                 $username $uid $gid $gname $gcos $homedir $shell
         kount=`expr $kount + 1`
    fi
done < /etc/passwd
case $kount in
   0) echo "No lines found" 1>&2    ;;
   *) echo "$kount lines found" 1>&2
esac
```

此迴圈在done的關鍵字上讀取輸入，注意到使用合併標準輸出至標準錯誤輸出串流（用**1>&2**）的這三個敘述，這表示了您現在能安全地將全部的手稿轉向到一個檔案中及顯示任何訊息至終端機上。我們將用三種方法來執行手稿：

```
$ user_details.sh  > newlist                                    沒有參數
Usage: user_details.sh min_guid max_guid
$ user_details.sh 701 703 > newlist                             無效的參數
No lines found
$ user_details.sh 601 603 > newlist                             有效的參數
3 lines found
```

串流的合併工作看起來似乎很正常，現在來看**newlist**的內容：

```
$ cat newlist
Username    UID   GUID   Gname   GCOS          Home Directory    Login Shell
-------------------------------------------------------------------------------
romeo       601   100    users   Romeo N.      /home/romeo       /bin/ksh
juliet      602   100    users                 /home/juliet      /bin/bash
henry       603   100    users   henry blofeld /home/kaust       /bin/bash
```

注意另外兩個敘述（echo及printf）的標準輸出如何以它們自己的方法產生此檔。您搜尋這兩個檔案，並把搜尋的內容用printf產生格式化的輸出，而手稿中只使用到一個外部命令(grep)！


Note

如果您從Bourne shell**中執行此手稿，您應該能使用兩個外部命令（**grep**及**printf**）。即使您在檔案newlist中找到它們，您依然會得到訊息"No lines found"。在**Bourne**的**sub-shell**中執行**while**迴圈時，迴圈中的kount變數在外部是無效的。**exec**敘述**(19.14)**能夠解決這個問題。**

19.6 Sub-shell的問題

當shell建立一個程序時，它把某些自身環境的特色傳遞給子程序(child)。被建立的程序（表示命令）也能夠繼承這些參數並使用它們。這些參數包含：

- 雙親程序的PID。
- 程序的UID（擁有者）及GUID（群組擁有者）。
- 目前工作的目錄。
- 三個標準（輸出／入）檔案。
- 由雙親程序開啟來使用的其他檔案。
- 在雙親程序中，一些有效的環境變數。

我們已經在指令手稿中（在sub-shell中執行）使用變數，像是**HOME**及**SHELL**，而未曾定義過它們。但我們定義在 **.profile**與其他手稿中的這些變數又發生了什麼事？他們也可以傳遞至sub-shell中嗎？有許多其他相關於shell 環境的議題也值得我們注意，下面章節中我們將一一討論。

19.6.1 export：輸出shell的變數

預設存在shell的變數中的值，是不會傳至孩子的shell中。但shell也可以遞迴地輸出（用export敘述）這些變數到所有的子程序中，因此它們在全區域中皆為有效。您之前已使用過這樣敘述，現在您應該了解為什麼要這樣做了吧！

考慮一個簡單的手稿，它把變數**x**的值echo出來，變數**x**定義在手稿未被執行前的shell環境中：

```
$ cat var.sh
echo The value of x is $x
x=20                                                    # 現在更改x的值
echo The new value of x is $x
```

首先在提示符號下指定**x**的值為10，然後執行手稿：

```
$ x=10 ; var.sh
The value of x is                                       x值沒有在次sub-shell中顯示
The new value of x is 20
$ echo $x                                               內部手稿設定的值沒有
10                                                      影響到外部的手稿
```

因為**x**是一個在登入shell的本地變數（local variable），它的值不能被sub-shell中所執行手稿的echo來存取。要讓**x**變數全區域有效，您需要在手稿被執行前使用export敘述：

```
$ x=10 ; export x
$ var.sh
The value of x is 10                                    外部手稿設定的值現在可以被看見
The new value of x is 20
$ echo $x
10                                                      手稿中重置的變數值，在外部仍然無效
```

當**x**被輸出時，它指定的值(10)在手稿中也有效。但是當您輸出一個變數時，會有另外的重要結果；在手稿中(sub-shell) 再次指定(**x=20**)的值不能在執行此手稿的雙親shell中被看到。

您必須輸出已定義的變數，除非您有堅強的理由不讓sub-shell繼承這些變數值。要知道您是否已經輸出了變數，只要使用沒有參數的export命令，它可以列出所有輸出的環境變數（已經被輸出）及使用者定義輸出的變數（像**x**）。env 指令也會將所有輸出的變數顯示出來。


Note

一個變數對於定義它的程序來說只是本地的(local)**。但是，當被輸出時，在它所有的子程序中都是有效的。然而，當孩子改變變數值時，其雙親無法看到改變的值。大多數的人常犯的錯誤是每次改變變數值就使用**export**命令。事實上是不需要的。**

19.6.2　群組命令(Command Grouping)

除了在(2.5.1)的圓括號 "**()** " 外，shell也使用**{}** 來將命令集合成一組。這兩者之間的不同點在於前者將這一組的命令執行在sub-shell中，而其他的指令只會使用目前shell。當我們使用內建的命令cd及pwd時可以看出明顯的不同：

```
$ pwd
/home/romeo
$ ( cd progs ; pwd )
/home/romeo/progs
$ pwd
/home/romeo                                             回到最初的目錄
```

在sub-shell中，執行cd來改變工作目錄（一個環境參數）到/home/romeo/progs中。雙親（登入shell）不會接受這樣的改變，因此會回到最初的目錄中。相同的一組命令，而這次使用{}來運作以說明不同的情況：

```
$ pwd
/home/romeo
$ { cd progs ; pwd
> }
/home/romeo/progs
$ pwd
/home/romeo/progs                                    現在永遠的改變目錄
```

這兩個命令在沒有複製新shell的情況下被執行，現在永遠的改變目錄。


Tip

結束的大括號 }本身被放在另外一行，然而，當您需要兩個大括號（{}符號）放在同一行時，只要在最後一個命令後面加上分號：

```
{ cd progs ; pwd ; }
```

但是什麼時候會用到命令群組的功能呢？回顧第十八章所設計的手稿，您會注意到，許多手稿的某些段落可以用適當的命令群組來取代，舉下面的例子來說：

```
[ $# -ne 1 ] && { echo "Usage: $0 pattern" ; exit 3 ; }
```

能簡單的取代手稿emp3b.sh (18.9.1)中的開頭段落，如果使用者沒有指定一個參數就會造成程式執行的失敗。


Note

()運算符號在sub-shell中執行被括起來的一組指令，而{}運算符號同樣地把括起來的一組指令執行在目前的shell中。也就是說，如果在那組指令中有exit 指令，你必須使用{}，否則您將不能離開所有的shell！

19.7 陣列（ksh及bash）

Korn及bash支援簡單的一維陣列，其第一個索引的元素(elements)是0。*ksh93*版本限制元素個數為4096，bash則無此限定。在此，您可設定及計算陣列prompt的第三個元素值：

```
$ prompt[2]="Enter your name"
$ echo ${prompt[2]}
Enter your name
```

請注意計算工作必須放在大括號中，並把prompt[2]當作一個變數。而且，它與您定義在同一shell的變數prompt是沒有衝突的。當一群元素需要被指定時，您可以在()中使用以空白來分隔的資料列：

```
month_arr=(0 31 29 31 30 31 30 31 31 30 31 30 31)
```

這是陣列**month_arr**的定義。這種語法可用在bash及*ksh93*中。如果您使用Korn舊的版本，您可以用set -A 敘述：

```
set -A month_arr 0 31 29 31 30 31 30 31 31 30 31 30 31
```

在上述例子中，陣列儲存十二個月份中每個月可用的日期數字。perl的陣列也使用第一種形式來指定元素。第一個元素很明顯的必須指定0，可以用如下的數字簡單的找出六月的天數：

```
$ echo ${month_arr[6]}
30
```

使用**@**或***** 當作陣列索引指標（subscript）中，您不但可以顯示陣列中所有的元素，而且可顯示元素的數目。除了其中之一用到#，形式幾乎相同：

```
$ echo ${month_arr[@]}
0 31 29 31 30 31 30 31 31 30 31 30 31
$ echo ${#month_arr[@]}
13
```

陣列的長度

我們能使用陣列來確認輸入的日期嗎？顯示於表19.2的下一個手稿dateval.sh就能做到，它也可以將閏年的二月的改變納入考慮（除了這樣之外，每四百年也要閏年一次)。

在外部case結構的第一個選項檢查是否有輸入(null response)。第二個選項使用表示式**$n/$n/$n**來做**mm/dd/yy**八位元字串的檢查。使用改變**IFS**的值，日期的組件被設定在三個位置參數中並被用來檢查合法的月份。第二個case構成閏年的檢查及使用陣列來驗證日期。continue敘述讓您做驗證檢查測試失敗時，迴圈會從頭開始。

現在，讓我們來試驗這個手稿：

```
$ dateval.sh
Enter a date: [Enter]
No value entered
Enter a date: 13/28/00
Illegal month
Enter a date: 04/31/00
Illegal day
Enter a date: 02/29/01
2001 is not a leap year
Enter a date: 02/29/00
02/29/00 is a valid date
[Ctrl-c]
```

圖 19.2 dateval.sh: **一個使用陣列的日期確認之手稿**

```
#!/bin/ksh
IFS="/"
n="[0-9][0-9]"
set -A month_arr 0 31 29 31 30 31 30 31 31 30 31 30 31

while echo "Enter a date: \c" ; do
  read value
  case "$value" in
        "") echo "No value entered" ; continue ;;
  $n/$n/$n) set $value
            let rem="$3 % 4"                          # 檢查是否閏年
            if [ $1 -gt 12 -o $1 -eq 0 ] ; then
                echo "Illegal month" ; continue
            else
                case "$value" in
              02/29/??) [ $rem -gt 0 ] &&             # 二月是29或28天？
                        { echo "20$3 is not a leap year" ; continue ; } ;;
                     *) [ $2 -gt ${month_arr[$1]} -o $2 -eq 0 ] &&
                        { echo "Illegal day" ; continue ; } ;;
                esac
            fi;;
         *) echo "Invalid date" ; continue ;;
  esac
  echo "$1/$2/$3" is a valid date
done
```

要手稿完全的離開，我們必須用中斷鍵來中斷執行。我們稍後也將討論如何處理離開的方法。

19.8 字串處理（ksh 及 bash）

Korn及bash不需要用expr，它們自己有適合的字串處理功能。不像expr，它們用萬用字元卻不能用正規表示式。所有形式的用法需要以{}符號來圍住變數名字及特殊符號。變數在它們的形式中變得很難去記住，有時也不太好使用。

字串長度　字串的長度可以很容易的於變數名稱前加上#來找出，考量下面的例子：

```
$ name=”vinton cerf”
$ echo ${#name}
11
```

我們於(18.11.2)中使用expr來計算字串長度，但內建的功能相當容易使用：

```
if [ `expr "$name" : '.*'` -gt 20 ] ; then          使用expr
if [ ${#name} -gt 20 ] ; then                       Korn及bash
```

第二種形式看起來更容易閱讀，也更快速因為不需要呼叫外部命令。它應該很容易被記住，因為perl使用類似的形式來估算陣列的長度(20.8)。

取出子字串　*ksh93*及bash提供簡單的技術來取出子字串就像高階語言的處理方法一樣。使用同樣的變數name的值，以下是您如何實現在awk及perl中的substr()函數：

```
$ echo ${name:3:3}                                  第一個位置是0
ton
$ echo ${name:7}                                    取出字串剩餘的部分
cerf
```

這功能比採用正規表示式相關技術的expr還要容易使用許多。

19.8.1　取出符合樣式的字串

您能使用特別的樣式符合功能來取出子字串。這些函數會用到#及%這兩個字元，而選擇的方式似乎是基於記憶的考量上。當在大括號{}中計算變數，#是使用在符合開頭部分，而%是用在結尾部分。

在移除檔案名稱的副檔名時，先前您是使用外部命令，即basename (18.16.4)。這次，您能使用變數的*${variable%pattern}*格式來執行取出動作，下面是您該如何使用：

```
$ filename=quotation.txt
$ echo ${filename%txt}
quotation.                                          刪除txt
```

在變數名稱之後的%符號刪除了符合變數內容尾端最短的的字串。如果使用兩個%，會取符合最長的那一個。當從FQDN取出主機名稱時，%要與萬用字元合用：

```
$ fqdn=java.sun.com
$ echo ${fqdn%%.*}
java
```

您會想起basename也能夠從路徑名稱中取出基礎的檔案名稱。您需要刪除在開頭部分符合樣式*/之最長樣式的變數值：

```
$ filename=" /var/spool/mail/henry "
$ echo ${filename##*/}
henry
```

以上會刪除/var/spool/mail的這段，其在開頭部分符合樣式*/的長樣式。現在您知道可以試著用#來做符合的規則。Korn及bash樣式符合形式在表19.1中。

19.9 有條件的替換參數

繼續變數計算的主題，您可以計算一個變數依據它是否有一空值或是有定義的值。您在這裡也要用到大括號{}，但這次您必須在變數名稱後面加上，接著是任何的+,-,=或?符號，而符號後緊接著字串。這種功能被稱為參數替換(**parameter substitution**)，它也能用在Bourne shell中。

+選項　使用*${variable:+string}*格式。在此，*variable*如果包含非空的值會被計算成*string*。假如不是空目錄，最好的方式是回應一些訊息：

```
found=`ls`
echo ${found:+"This directory is not empty" }
```

ls如果沒有找到檔案就不會顯示，在這樣的例子，變數 found 會被設定成空字串，不過，如果ls至少找到一個檔案時，會有訊息出現。

-選項　這是把+函數相反。在手稿中用於提示使用者輸入檔案名稱，當使用者只是按下*[Enter]*時會指定的預設值：

```
echo " Enter the filename : \c "
read flname
fname=${flname:-emp.lst}                    替換使用[-z $flname]的條件
```

如果在提示符號下沒有任何的輸入，flname會計算成emp.lst。但fname事實上就是包含這個值。此精簡的指定敘述替代了if的條件式。

表 19.1　bash及 ksh中使用樣式配對

格　式	刪除後所剩下的部分
${*var#pat*}	在開頭的$*var*符合短測試段 *pat*
${*var##pat*}	在開頭的$*var*符合長測試段 *pat*
${*var%pat*}	在結尾的$*var*符合短測試段 *pat*
${*var%%pat*}	在結尾的$*var*符合長測試段 *pat*

=選項　它的作法也相當類似，除了它更進一步的用指定的方式給要被計算的變數。要取代以分開的敘述句來設定起始及測試迴圈的變數，像是：

```
x=1 ; while [ $x -le 10 ]
```

您可以將它們結合成單一的敘述句：

```
while [ ${x:=1} -le 10 ]
```

？選項　它的工作類似-選項，除了會中斷及殺掉shell，如果其值是空（null）的話。如果使用者沒有回應，您能夠中斷手稿：

```
echo "Enter the filename : \c"
read flname
grep $pattern ${flname:?"No filename entered .... quitting"}
```

除了=外，所有的運算符號也能與位置參數合用。在本章的最後，您將使用這個知識來精簡之前設計的手稿序列。shell的參數替換函數列在表19.2中。


Note

在這兩節中，字串的處理技術只是一部分計算變數的方法。除了用=選項之外，它們並沒有用任何的方法改變變數值。但您能把這些值設給其他的變數，舉例來設：bname=${filename##*/}或fname=${flname:-emp.lst}。

19.10　shell函數

shell函數是由一群敘述所組成的同時被執行之指令串流，其在Bourne shell中也有這樣功能。函數比shell的alias(17.4)作法更進一步，可寫成一連串命令的捷徑，它會傳回一個值（但alias不能）：

```
function_name(){
        statements
        return value                                    可加可不加
}
```

表 19.2　**參數的替換運算符號**

形　式	計算
${*var*:+*pat*}	*pat*如果*var*有設定；否則為空(null)
${*var*:-*pat*}	$*var*如果*var*有設定否則為*pal*
${*var*:=*pat*}	如上，但設定*var*到*pat*
${*var*:?*pat*}	$*var*如果*var*有設定；否則印出*pat*及中斷

函數定義接在()之後，其主體(body)是在{}中。當函數被呼叫時，它會執行主體中所有的敘述。如果有return敘述，則會傳回函數成功或失敗的一個值（不是字串值）。因為shell敘述被執行在直譯模式中，shell函數必須放在呼叫它的敘述句之前。

讓我們先考慮一個簡單的應用。當要檢視一個有很多檔案的目錄時，我們經常會被強迫使用ls -l | more。有時，您可能也喜歡用ls加上選擇的檔名，對shell函數來說這些指令的序列就是理想的候選者，我們將此函數叫做ll：

```
$ ll () {
> ls -l $* | more
> }
```

雖然在定義中需要用()，當呼叫函數時您卻不需使用它們。現在您可以呼叫這函數加或不加參數：

```
ll                                          執行 ls -l | more
ll ux3rd??                                  執行 ls -l ux3rd?? | more
```

C shell的alias在此也能運作良好，但alias有其限制。像在指令手稿中，shell函數也使用命令列參數（像$1, $2等等）。$*及$#在函數中也保留他們一般的意義，然而alias不能認出他們。

哪裡要用到shell函數的定義？函數可以在很多地方被定義：

- 在每一個使用他們的手稿開頭。
- 在.profile（或是不同shell之特定啟始檔案），所以在本節中會用到。
- 是分別的“ library ”檔，所以其他的應用程式也可以使用它們。

我們已使用過很像程序的ll函數，但是shell函數也能用return敘述傳回一個成功或失敗的值來存於$?中。

在指令手稿中，由命序列參數設定的位置參數直接傳給shell函數是無效的。它們必須被儲存在分開的變數或傳到函數中，當成它本身的參數：

```
ll $2
```

Note

此處，指令手稿的第二個參數($2)可被讀進函數ll內部當成第一個參數($1)看待。$1在函數內外表示不同的實體。

19.11 設計shell函數

到目前為止，您已經設計了許多shell 程式，肯定大部分您會用到重複序列。序列會問使用者是否要繼續或它會驗證使用者的輸入。在即將到來的小節裡，我們將精巧地製作一些有用的shell函數，而您會想用在您的手稿上。我們將為它們發展一個程式庫(library)，所以它們會容易地被取用。

19.11.1　從系統日期產生一個檔名

身為一個系統管理者，您必須經常維護依據每日的特殊活動所區分出來的不同檔案。如果這些檔案是取自系統日期，您可以簡單的依相關日期來識別檔案。這裡有一個很好的主意，是利用shell函數dated_fname() 依序由date輸出中計算檔案名稱。讓我們在提示符號下來定義這函數：

```
$ dated_fname () {
> set -- `date`
> year=`expr $6 : ' ..\(..\) '`                    從年份中選出最後的兩個字元
> flname=" $2$3_$year "
> }
```

我們在檔案**flname**中儲存需要的值，因為shell函數只能傳回真值或假值，但非字串。此外，在函數內部設定的位置參數對目前的shell而言是無效的。當執行這函數之後，在目前的shell 中 **flname**的值應該是有效的：

```
$ echo $flname
Sep22_00
```

這字串可以被用來組成單一檔名，且每天一個。Korn及bash的使用者應該比較喜歡用**year=${6##??}**來取代使用expr。Oracle（譯註：一種資料庫軟體名稱）的使用者能簡單的使用這個函數來做出系統產生的dump檔名，以提供給exp(export) 命令：

```
exp scott/tiger file=$flname
```

這將產生輸出的 dump檔案**Sep22_00.dmp**。建立一個程式庫檔mainfunc.sh並且將此函數的定義放進來。我們稍後將會用這檔案，在我們加入其他兩個函數後。

19.11.2　繼續或不繼續

我們現在要考慮一下shell函數傳回的值。回想一下在手稿dentry1.sh(18.15.2)最後使用的序列，它提示使用者按下**y**來繼續，或是按**n**來中止最外面的迴圈。這個例行程序常用於指令手稿中，所以相當好的主意是轉換它成為函數anymore()：

```
anymore () {
   echo " \n$1 ?(y/n) : \c "  1>&2                  提示符號的輸入當成參數
   read response
   case " $response " in
     y|Y) echo 1>&2 ; return 0 ;;
       *) return 1 ;;
   esac
}
```

這函數使用**$1**參數來決定整個提示字串看起來的樣子。當函數用字串**Wish to continue** 被呼叫時，您被提示並要求回應：

```
$ anymore "Wish to continue"
Wish to continue ?(y/n) : n
$ echo $?
1                                          這個值被指定在return敘述句
```

在本章稍後的章節中，我們將使用函數的傳回值。這函數也定義在mainfunc.sh手稿中。

19.11.3 驗證資料輸入

再一次的回憶**dentry1.sh**手稿(18.15.2)。注意while迴圈中一再的提示使用者輸入合法的項目。這序列也值得轉換成shell函數。我們預期函數valid_string()會去檢查兩個東西：首先是所有被輸入的東西，接下來確認是否超出長度：

```
valid_string () {
    while echo "$1 \c"  1>&2 ; do
        read name
        case $name in
            "") echo "Nothing entered"  1>&2 ; continue ;;
             *) if [ `expr "$name" : '.*'` -gt $2 ] ; then
                  echo "Maximum $2 characters permitted" 1>&2
                else
                  break
                fi ;;
        esac
    done
    echo $name
}
```

這函數讀取兩個參數，包括提示的字串及回應字串最大的長度。Korn及bash使用者會喜歡用**[${#name} -gt $2]**來取代使用expr命令。 我們也將置放這函數於我們的程式庫檔案mainfunc.sh中。現在這個檔案擁有三個shell函數，您可以讓他們在任何shell script中生效，在script中藉由點命令來執行程式庫檔案。我們將在手稿user_passwd.sh 中這樣做：

```
$ cat user_passwd.sh
# 手稿用來驗證使用者輸入 -- 使用一個shell函數兩次
. mainfunc.sh                                    # 讓函數 valid_string 生效
user=`valid_string "Enter your user-id :" 16`
stty -echo                                       # 密碼不會被顯示
```

```
password=`valid_string " Enter your password: " 9`
stty echo                                                    # 打開回應功能
echo " \nYour user-id is $user and your password is $password "
```

這小小的手稿接受使用者的名字及密碼，分別確定他們未超出十六個及九個字元的長度。一個簡單的片段顯示shell函數可以減少手稿的大小：

```
$ user_passwd.sh
Enter your user-id : robert louis stevenson
Maximum 16 characters permitted
Enter your user-id : scott
Enter your password:
Nothing entered
Enter your password: *****                          不要顯示在螢幕上
Your user-id is scott and your password is tiger
```

因為他們接受所有shell的結構，包含位置參數，shell函數通常可取代指令手稿。呼叫shell函數能減少磁碟輸出／入(I/O)因為函數常駐在記憶體中。此外，因為shell函數被執行在目前的shell中，變數定義在shell中也可在函數中看見，反之亦然（但不是位置參數）。不過，確定在您登入期間盡可能使用您所需求的函數。


Tip

如果許多shell函數被使用在多個程式中，您應該把他們全部置放在單一「程式庫」檔中，並將這檔案儲存在一個合適的地方。在每一個手稿的開頭需要用到這些函數時，插入一用點（.）命令來執行程式庫檔案的敘述。

19.12　eval：計算兩次

您是否曾經試著在管線中設定變數然後執行它嗎？試著這樣執行：

```
cmd="ls | more"
$cmd                                        對於ls而言，|及more是參數!
```

這樣不能產生您期望的分頁輸出。現在定義一個 「數字提示」及試著計算它：

```
$ prompt1="User Name: " ; x=1
$ echo $prompt$x                                        $prompt 未被定義
1
```

在第一個例子中，shell把|及more當作是ls的參數之後才計算變數。結果，ls把它們看作兩個參數，而且產生不可預期的輸出。在第二個例子中， shell首先計算$prompt；它未被定義。然後它計算$x的值為1。

要讓這些命令序列被正確的執行，我們必須能夠延遲計算部分的命令列。此時

可使用eval 敘述（在Bourne一樣有效）來計算命令列兩次以達到上述的功能。在第一輪時它先抑制一些計算動作，然後只在第二輪才做計算。

我們能用eval 來讓上面的第一個序列工作，像是：

```
eval $cmd
```

在第一輪時，eval 找出三個參數—**ls**, **|** 及**more**。然後再重新計算命令列並用**|**來區分它們為兩個命令。現在命令應該能正確的執行。

第二個序列能由隱藏在第一個**$**前的****來讓它動作，然後使用eval ：

```
$ x=1 ; eval echo \$prompt$x
User Name:
```

第一輪用跳脫符號****將**$**忽略；這計算的結果變成**\$prompt1**。第二輪將****符號忽略，並把**$prompt1**當變數來計算，而此正好是我們要的。現在我們要看看一個有用的應用例子，即使用許多編號變數。

如果一個手稿必須從終端機輸入十次，您必須去定義及使用十個變數去裝這些值。有時，您甚至不知道在執行當時變數到底有多少個。我們比較喜歡設計一個更通用的手稿能自己動態的產生變數名稱。如此就能儲存提示符號當作變數及讀取輸入放進這些「編號變數」。

我們已經使用過編號提示；我們現在需要使用像是**value1**, **value2**, **value3**等等可以裝輸入值的變數：

```
$ { x=1
> eval echo \$prompt$x ' \\c '
> read value$x                                          OK—沒問題
> eval echo \$value$x ; }
User Name: kleinrock
Kleinrock
```

read value$x敘述讀取回應並設到變數**value1**中；eval會經由兩輪的處理來顯示這個值。到目前為止你已經建立一個意義重大的結果，我們會利用在下一個手稿中。

我們能用**$1**或其他的來取出位置參數，但我們能直接取出最後的參數嗎？因為我們已經有**$#**的值，所以透過eval的服務：

```
$ tail -1 /etc/passwd
martha:x:605:100:martha mitchell:/home/martha:/bin/ksh
$ IFS=:
$ set `tail -1 /etc/passwd`                          set --在這不一定需要
$ eval echo \$$#
/bin/ksh
```

注意，我們甚至不需要知道在/etc/passwd檔案裡其中一行的區塊數。因為我們能簡單用上述的方法分離出script 的最後一個參數，在手稿cpback2.sh(18.17)中，我們還需要用到head及tail來排除表列中的目錄嗎？在本章的練習中有這個題目。

19.13　createuser.sh：使用eval來建立使用者

我們現在要來開發一個手稿並使用useradd命令(22.3.2)來建立使用者。在本章中只有這手稿需要管理者的權限（在19.2.1節中討論的fdisk也要用管理者來執行)。useradd所有需要的參數將透過交談方式來供給，這手稿要有下列的功能：

- 接受使用編號提示和變數來輸入六個區塊（field)。
- 使用放在檔案mainfunc.sh中的anymore()函數。
- 依據shell是bash或其他的來決定echo是否使用-e選項。

因為我們必須提示使用者六次，相當吸引我們使用eval來產生精簡的手稿。createuser.sh手稿顯示在圖 19.3中。這次mainfunc.sh手稿被執行的方式有點不同，通常目前的目錄不會出現在管理者的指令路徑，點(.)命令（除了bash之外的所有shell）不會在目前的目錄中尋找手稿，這就是為什麼要使用到相對路徑名稱(17.9.1-Note)。

圖 19.3　createuser.sh：**使用**eval**的手稿來建立使用者**

```
# Program to create user -- uses eval to create prompts and variables
# Will run in the Korn and bash shells without modification

option=
[ $SHELL = "/bin/bash" ] && option=-e
. ./mainfunc.sh                          # 讓anymore()函數生效

prompt1="User Name:"     ; prompt2="User-id:"    ; prompt3="Group-id:"
prompt4="Home Directory:"; prompt5="Login Shell:"; prompt6="GCOS Details:"

while true ; do
    x=1
    while [ $x -le 6 ] ; do
        eval echo $option \$prompt$x '\\c' 1>&2      # 產生六個提示
        read value$x
        x=`expr $x + 1`
    done
    useradd -u $value2 -g $value3 -d $value4 -s $value5 -c "$value6" -m $value1
    anymore "More users to create" 1>&2 || break
done
```

在手稿的開始定義六個提示並當作編號變數，在內圈的while迴圈依序將它們顯示出來。這些值被讀到**value1**, **value2**等等的變數上。現在，我們建立一個使用者ppp：

```
# ./createuser.sh                                    當不位在管理者的路徑下時需要使用
User Name: ppp
User-id: 520
Group-id: 100
Home Directory: /home/ppp
Login Shell: /etc/ppp/ppplogin
GCOS Details: PPP Server Account
More users to create ?(y/n) : n
```

但願這應該能建立新的使用者帳號，可以從**/etc/passwd**檔案觀察其最後一行來確定：

```
# tail -1 /etc/passwd
ppp:x:520:100:PPP Server Account:/home/ppp:/etc/ppp/login
```

現在，真是太神奇了，感謝有eval指令，我們可以在一個小小的手稿中管理讀入六個由使用者輸入的回應並送到六個變數中！ 如果使用者登入的shell是bash，手稿會自動使用echo的**-e**選項，但不用修改就能在其他所有的shell中執行。我們必須用./createuser.sh來執行手稿，因為目前的目錄不在管理者的路徑下。


Note

當建立ppp使用者時我們指定的登入shell，並不是一個shell程式，正確來說，它是可執行的任何命令（甚至是指令手稿）且不需要限定是shell程式。事實上，當您建立一個PPP伺服器時，您不須給使用者有執行shell的機會。登入的過程將執行ppplogin的手稿，然後將使用者登出，通常就是這樣做的。

19.14 exec敘述

在您研究程序建立的機制時(10.2)，曾介紹使用exec的系統呼叫，即可以覆寫fork程序的那個。這個屬性對指令手稿設計者有些重要性，這些人有時候需要用其他的程式碼來覆寫目前的shell。這也是目前我們未曾做過的事，但如果您用exec來執行任何的UNIX命令，命令就會覆寫目前的shell。在您完成命令後會有登出的效果：

```
$ exec date
Sun Apr  9 14:12:03 EST 2000
login:
```

有時，您需要讓使用者在登入時自動執行一個單獨的程式並拒絕他有跳脫進入shell的機會，您可以在**.profile**檔案中適當的放置exec來執行命令。當使用者登入時會

開始執行命令，但是當命令執行完畢後就會登出。原本就沒有shell的存在。

用exec來置換目前的shell至另一shell時是很有用的，當exec ksh時，會由新的shell來置換目前的shell環境。這功能也能讓您使用exec login *userid*來登入到不同的使用者帳號中，當您用telnet在遠端的電腦工作時是很有用的。

19.14.1　在目前的shell中產生轉向

exec還有另外的重要功能；它能轉向整個手稿的標準串流(standard stream)。如果手稿的一些命令要用標準輸出送到一個檔案時，與其分別在每一個命令上使用轉向符號，您可以使用exec來重新指定他們預設的目的位置，像這樣：

```
exec > found.lst                                    也可以使用 >>
```

您可能會說這沒有了不起；有些手稿就能自己轉向，但是exec能夠建立一些串流，除了那三個標準（0, 1 和 2）之外，即每一個使用自己的描述符號(descriptor)。舉例來說，您能建立一個檔案描述符號3將所有輸出轉向到一實體檔案foundfile：

```
exec 3>foundfile
```

您現在能藉由合併標準輸出串流及檔案描述符號3來寫到檔案中：

```
echo "This goes to foundfile" 1>&3
```

在程式語言中，這意味著您能使用exec來提供一個檔案的邏輯名稱（在perl中的*filehandle*）。在此強大的I/O處理器的協助下，您現在應該能用更簡潔有力的方法處理檔案。我們現在來設計一個 script，從一個檔案中讀取empid（員工編號），然後用它來搜尋emp.lst檔案並依據下面三種狀況儲存在三個分別的檔案：

- 找到行。
- empids沒有找到。
- 不正確的empid格式。

首先，這是包含empid的檔案，它還包含二個三位數之empid（其他都是四位數），它們應該可以被手稿找出來：

```
$ cat empid.lst
2233
9765
2476
789
1265
9877
5678
245
2954
```

圖 19.4 countpat.sh：**使用**exec**來建立多個串流**

```
exec  > $2                    # 開啟檔案1來儲存選擇到的行
exec 3> $3                    # 開啟檔案3來儲存未找到的樣式
exec 4> $4                    # 開啟檔案4來儲存無效的樣式

[ $# -ne 4 ] && { echo "4 arguments required" ; exit 2 ; }

exec < $1                                          #  轉向輸入
while read pattern ; do                            #  現在從$1讀取
     case "$pattern" in
        ????) grep $pattern emp.lst ||
              echo $pattern not found in file 1>&3 ;;
           *) echo $pattern not a four-character string 1>&4 ;;
     esac
done
exec 0<&- ; exec >&- ; exec 3>&- ; exec 4>&-              # 關閉檔案
exec >/dev/tty                 # 轉向標準輸出重回終端機
echo Job Over
```

在圖19.4中的countpat.sh手稿，劃分標準輸出到三個串流及轉向它們至三個分開的檔案中。這需要四個參數，包括檔案含有樣式及三個串流的檔案。

標準輸出串流被合併到檔案描述符號**1**,**3**及**4**中。注意到，我們也已經設定了所有標準輸入的來源（source）為**$1**，所以在迴圈中的read敘述會從**$1**來輸入含有樣式的檔案。當所有的檔案已寫入完畢後，標準輸出串流必須重新指定到終端機上(exec >/dev/tty)，否則結束工作訊息（**Job Over**）將也會經由**$2**被儲到檔案中。

這手稿相當的乾淨俐落而且有兩個敘述使用到合併符號。grep使用標準輸出的檔案描述符號，因此沒有合併的必要。這手稿取得四個參數及轉向輸出至它們之中的三個：

```
$ countpat.sh empid.lst foundfile notfoundfile invalidfile
Job Over
```

這個訊息出現在終端機上而非轉到這些檔案中，現在您自己去觀察這三個檔案看實際上發生了什麼：

```
$ cat foundfile
2233|charles harris  |g.m.      |sales     |12/12/52| 90000
2476|jackie wodehouse|manager   |sales     |05/01/59|110000
1265|p.j. woodhouse  |manager   |sales     |09/12/63| 90000
5678|robert dylan    |d.g.m.    |marketing |04/19/43| 85000
```

```
$ cat notfoundfile
9765 not found in file
9877 not found in file
2954 not found in file
$ cat invalidfile
789 not a four-character string
245 not a four-character string
```

這是exec強大的地方，它同時開啟一些檔案並且分開取用每一個檔，這和perl使用其 filehandle的方法是一樣的。通常比較好的作法是以檔案描述符號代替使用檔案名稱，因為指令手稿就會完全獨立於所使用的檔案。

19.15　set -x：指令手稿的除錯

shell通常也支援手稿的除錯功能，亦即它的 **-x**選項。當使用在手稿的內部（甚至在$的提示符號下），它會在終端機上回應(echo)每一個敘述，如它被執行時的內容前面再加上+的符號。修改cpback2.sh手稿(18.17)，在它的最前面放置如下的敘述來開啟set的選項：

```
set -x
```

您可以在手稿的結尾用set +x來關閉set -x功能。我們繼續從之前停止的位置執行手稿，但這次只使用一個檔案，即**index**（**index.2** 已經被拷貝到 **safe**），附上註解的輸出如下：

```
$ cpback2.sh index safe
+ [ 2 -lt 2 ]                                          檢查參數編號
+ tr ' ' ' \012'
+ echo index safe                                      這是echo $*
+ 1> 707
+ tail -1 707                                          識別目錄名稱
+ destination=safe                                     及設定它的變數
+ [ ! -d safe ]
+ expr 2 - 1
+ count=1
+ head -1 707                                          拷貝唯一的檔案
+ [ ! -f safe/index ]
+ copies=1
+ true
+ [ ! -f safe/index.1 ]
+ expr 1 + 1
+ copies=2                                             index.1存在
+ true
+ [ ! -f safe/index.2 ]
+ expr 2 + 1
+ copies=3                                             index.2也存在
```

```
+ true
+ [ ! -f safe/index.3 ]
+ cp index safe/index.3                                    檔案拷貝至index.3
+ echo ‘File index copied to index.3’
File index copied to index.3
+ break
+ rm 707                                                   移除暫存檔案
```

如果您要找出為什麼手稿不能如預期般的工作，可以用這個理想的工具。注意到shell如何列印每一個被執行的敘述加上+符號，它會告訴您head,tail及cp命令的命令列，它甚至顯示expr在每一個重覆的迴圈中如何增加變數 copies的值！

19.16 trap：中斷程式

每回中斷鍵被按下時，指令手稿預設會結束執行。這不是一個結束指令手稿的好方法，因為可能會留下很多在磁碟中的暫存檔案。trap敘述可以讓您的手稿如果收到一個訊號時做任何您想做的工作，這敘述通常被放在指令手稿的開頭及使用兩個列表（list）：

```
trap ‘command_list’ signal_list
```

當手稿被送進任何放在*signal_list*中的訊號，trap就會執行在*command_list*中的命令。訊號列表包含整數值、一個或是更多訊號的名稱，即您在使用kill 指令時所使用過的。除了使用**215**來表示訊號列表，您也可以使用**INT**, **TERM**。

如果您習慣用shell的PID編號當作建立暫存檔案的檔名，您應該使用一些trap服務，不論何時發生中斷就會移除這些暫存檔：

```
trap ‘rm $$* ; echo “Program interrupted” ; exit’ 1 2 15
```

現在，當您送訊號**1**,**2**或**15**，trap攔截（「抓取」）訊號，移除所有以**$$***展開的檔案，回應訊息及終止手稿。當中斷鍵按下時，它送出訊號編號**2**，把它包含在所有的進階手稿中也不失為一個好主意。

您可能喜歡忽略訊號及繼續進行手稿。在這例子中，您應該在程式中使用空的（null）命令列表來排除這類的訊號：

```
trap ‘’ 1 2 15                                             這手稿不能被殺除
```

在您的指令手稿中，沒有強制要用trap敘述。然而，如果有的話，不要忘了加上exit敘述在命令列表的結尾，除非您要手稿忽略特殊的訊號。Korn及Bourne shell不能在登出後再執行一個檔案，但使用trap，您就可以讓它們這樣做。您將需要使用**0**來表示訊號編號，這些shell也使用敘述trap -來重新設定它們預設的訊號值。您也

能在手稿中使用多個trap命令，每一個覆寫先前的設定。

我們將結束與shell的旅程。與awk及perl一樣，shell也應該被善加利用如果想讓UNIX系統的功能發揮到極致。雖然shell程式執行比C程式慢，但對大部分的管理工作而言，shell的速度和解決方案是完全的被接受。

摘　要

set將值放進位置參數， shift則從左邊來移除參數。最左邊的變數會被移走，最好在使用shift前先將它們儲存到其他變數。Set所使用的區塊分隔符號可由IFS變數來決定。set -- 如果輸出是空的或以連字號起頭時必須使用。

此地文件(<<)從手稿本身提供手稿的輸入，它能與命令替換及變數合用。它通常用在不以檔名當參數的命令或執行交談程式卻無互動行為的狀況。

let在Korn shell及bash中執行內建的計算功能，當指定一個值時，它不需要在字首前加$。Korn shell及bash使用（(及)）運算符號，被當成新增的POSIX相容的計算工具。

一個條件式或是迴圈能在fi及done的關鍵字上被轉向或是輸出到管線。在轉向的結構中所有以終端機為預設的標準輸出必須分別用>/dev/tty來做轉向。shell也能使用1>&2及2>&1來合併標準輸出及標準錯誤輸出串流。

當雙親定義的變數只有被輸出之後才可能被孩子看到。然而，當孩子更改變數的值時，改變的變數不能被雙親看到。可以用()運算符號在sub-shell中執行群組命令，但{}不會產生sub-shell。

Korn及bash提供簡單的一維陣列，陣列可用來驗證日期。

字串處理的功能在Korn及bash中發展的很好。變數的內容能夠使用萬用符號來比對，使用字元#及%，您可以配合樣式在開頭及結尾的字串，取出不符合的部分。

shell的變數可以依照是不是被指定為非空值而在條件式中被計算。=運算符號指定變數的變數值，而?列出錯誤訊息及離開shell。

shell函數讓您壓縮重要及重複的序列，這比別名還更好用。他們接受位置參數，但不是傳遞到手稿中的參數，除非將它們以參數方式再傳進函數。函數只能傳回真或假值。

eval處理命令列兩次，可用來模擬陣列及執行變數。使用eval，您能建立通用的編號提示及變數而明顯簡化程式碼。

當在命令前加上exec會覆蓋目前的shell，它能建立多個串流及連結每一個串流和其本身的檔案描述符號。舉例來說，它使用1>&7寫到非標準的檔案描述符號7。

指令手稿的除錯，在手稿開始的地方使用set -x，因此每一個命令列會被回應（echo）到螢幕上，此命令會將一個反覆迴圈的執行過程全部顯示。

如果您要讓您的手稿回應一個中斷的特別方法就是使用trap。利用它來處理當手稿接收到一個訊號時，移除暫存檔案的情況是很有用的。您也能讓您的手稿忽略中斷訊號。

自我測試

19.1 請觀察這個命令對您來說有任何的意義嗎？

```
set `set`
```

19.2 如果手稿使用十二個參數，您如何存取最後一個呢？

19.3 這個命令有什麼問題嗎？

```
set `grep -c "A HREF" catalog.html`
```

19.4 如果一個更改目錄的動作被執行在指令手稿中，為什麼完成手稿後會回存至家目錄中？您如何克服這個問題呢？

19.5 如果在命令提示符號下定義變數，您如何讓它也能在指令手稿中被使用？

19.6 執行script命令及在提示符號下定義變數。現在使用exit離開script，然後顯示這個變數值。您看到了什麼，為什麼呢？

19.7 命令echo ${#x}產生輸出**x: Undefined Variable**。何時會發生，而這敘述句被用來做什麼呢？

19.8 這個敘述做什麼？

```
flname=${1:-emp.lst}
```

19.9 何時您需要在命令中使用此地文件？

19.10 寫一個shell函數來移除執行這個函數當時的目前目錄。

19.11 在手稿var.sh(19.6.1)中，您如何在不使用export的情況下，讓變數 **x** 與它在外部設定的值相同？

19.12 您如何確定在登入時立即執行一個特別的程式，當程式完成後使用者會被登出？

練　習

19.1 如果命令set `cat foo`產生錯誤 **unknown option**，會是什麼原因呢？假設**foo**是一個很小的可讀檔案。

19.2 撰寫一個能接受檔名當作參數的手稿，如果檔案存在就顯示它的最後修改時

間；如果它不存在時，也能顯示合適的訊息。

19.3 撰寫一個手稿能接受單一或是多個字組（word）樣式當參數，從一組檔案（**foo***）中以它來搜尋並且當這些檔案包含該式樣時開啟vi編輯器。您如何在每一個檔案中找出該樣式？

19.4 撰寫一個手稿能接受匿名的ftp站（像是**ftp.planets.com**）及任何數目的ftp命令（像“**cd pub**”，“**get cp32.tar.gz**”）當參數。它應該能連結到此網際網路站台，自動登入及執行ftp的命令。

19.5 重覆練習19.3，請遞迴地在目前的目錄中搜尋該式樣。

19.6 此exit命令，以下面的方式置放手稿中，為何不會終止手稿？您將如何恢復它？

```
( statements; exit )
```

19.7 呼叫su命令（如果您知道系統管理者的密碼），然後執行ps -t 加上終端機名稱。您得到什麼結論？

19.8 您有一個很小的手稿cman，它包含了這兩行：

```
#!/bin/ksh
x=`find $HOME -name $1 -print`
cd $x
```

當您執行cman man1，您發現目前的目錄沒有改變，即使**man1**目錄存在於您家目錄第三層中。為什麼會發生這樣的狀況，以及您如何改變到這個目錄中？您能在Bourne shell中執行這序列嗎？

19.9 這個敘述有什麼錯誤嗎？您如何修改它，使它執行正確呢？

```
[ $# -ne 2 ] && echo "Usage: $0 min_guid max_guid" ; exit
```

19.10 您需要在夜晚執行一個工作，並且將輸出和錯誤訊息放在同一個檔案中。您要如何執行這個手稿？

19.11 您如何使用exec 來儲存手稿輸出於一個檔案及錯誤訊息在另一個檔案中？

19.12 使用陣列，您如何取出手稿的最後一個命令列參數？

19.13 手稿包含了敘述while [${count:=1} -lt 50]，卻不能執行迴圈。可能的原因是什麼？

19.14 為rm寫一個shell函數，每當您使用超過三個檔案名稱時，就進入做交談模式。

19.15 撰寫一個shell函數 lstot()，能列出參數中所有檔案大小（所有檔案不用參數）的總和？

19.16 修改在19.13節中的範例使用陣列來代替eval。

19.17 為什麼您不能在函數中使用exit敘述將控制權交還給呼叫它的程式呢？

19.18 修改手稿cpback2.sh (18.17)來複製檔案至一個目錄中，只有當檔案不存在於目錄時才能被複製，您只使用一個外部命令(cp)及利用Korn及bash shells的功能，手稿的最後一個參數是這目錄（提示：使用eval敘述來驗證目錄名稱)。

19.19 修改在練習19.18發展出的程式，所以只有較舊的檔案才會被覆寫。

19.20 您必須在手稿中使用迴圈而且想觀察在迴圈中每一次重覆時這些變數的值。您如何不使用任何echo 敘述來做到呢？

19.21 您如何確定在中斷離開前手稿會提示您呢？

19.22 您如何保證所有在Bourn或Korn shell登出時，所有以數字起頭的檔案會被移除呢？