第十三章 Shell Script 程式設計

13-1 Shell Script 程式簡介

『命令稿』(Script)與批次(Batch)檔案非常類似，都是集合多個命令(或稱公用程式)而成為另一個命令，然而命令稿的功能比批次檔案強了許多。一般『命令行』(Command line)每次僅能輸入一行命令，利用 Enter 鍵表示結束並傳送給系統執行。雖然『命令行』也利用其他方法(譬如管道，|)，將多個命令整合且一併傳送給系統執行，但沒有將它編制成檔案，這是『批次』與『命令稿』之間的差別。批次檔案僅將多個命令編寫成一個檔案，要求系統依照檔案內『順序』執行命令，命令稿則不然。命令稿可以在命令檔案內中加入一些控制程序，如判斷敘述(if/else)、迴圈敘述(for/while/until)等等，並非僅順序執行，而是會依照狀況改變執行程序。所以命令稿是屬於程式化的敘述 Shell 命令，因此稱之為『Shell script』(外殼命令稿)。從 Shell 的最基礎架構開始，Shell 命令就是為了整合許多系統函數編寫而成，利用它可以輕易的操作系統，是最基本的使用者介面。然而，當基本工具不能滿足使用者所需時，就得利用 Shell script 整合多個 Shell 命令，並於其中加入適當的敘述語句，如此便可製作出更符合使用者所需的工具。簡單的說，Shell script 最主要的功能就是整合多個 Shell 命令，並建立出新的 Shell 命令。

命令稿是一種直譯程式，可以直接執行；而一般程式語言(如 C 語言)都必須經過編譯程式，將原始程式翻譯成目的碼(Object file)，再連結多個目的碼成為一個可執行檔。然而，一般命令稿大多不需要經過編譯成目的碼，也沒有連結多個程式的功能，命令稿的語句都可以直接執行，無需經過編譯。

除了建立新命令之外，我們也可以發現大部分 Unix/Linux 系統的運作程序，都是利用 Shell Script 所描述而成。由此可見，想要了解或管理 Unix/Linux 系統，認識 Shell script 是不容或缺的，這也是本章介紹 Shell script 最主要的目的。從另一個角度來看，目前在 Unix/Linux 系統上雖有許多不同的 Shell，但在 Unix/Linux 系統上，除了 Shell 之間大致上可以相容之外，還可以在每一個 Shell script 程式指定自己所屬的 Shell 環境，通常系統也都會安裝多個 Shell 環境以供各種程式呼叫，亦即每一個命令稿都可以宣告自己的執行環

境，如 sh、csh、bash、perl 等等。本章採用相容性最高的 bash 來介紹 Shell script 的各種語法，相信這些敘述語句在其他 Shell 應該都可以相容。

13-1-1 Shell Script 執行

先不考慮相關控制敘述，僅就簡單的命令敘述所編寫的一個批次檔案，來觀察 Shell script 的執行程序。以下是利用 vi 所編輯的程式範例 **($vi ex7_1)**：

```
# A simple shell script
# file name:ex7_1
cal
date
who
```

上述 Script 中，井號（#）後面的任何文字都是『註解』，作為程式說明使用，並不是程式內容，接下來是連續三個 Shell 命令 cal、date 與 who。編輯完後，利用 ls –l 命令觀察它的屬性如何，操作如下：

```
$ ls -l ex7_1
-rw-rw-r--    1 tsnien tsnien 61 Jul 21 09:50 ex7_1
```

我們可以發現，編輯後的 ex7_1 僅是一般文書檔案，而且使用者沒有執行該程式的權利，但可以利用下述兩種方法執行該程式（其餘 Shell script 也是如此）：

【A. 執行 Shell script】

呼叫某一個 Shell 來執行該 Shell script，譬如，sh（Bourne shell）、bash（Bash shell）或 csh（C Shell），以 bash 為例如下：**($bash shell.scr)**

```
$ bash ex7_1                                    【執行 bash 程式】

      July 2005                                 【執行 cal 命令結果】
Su Mo Tu We Th Fr Sa
                1  2
3  4 5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
Thu Jul 21 10:00:10 CST 2005                    【執行 date 命令結果】
```

```
root        :0              Jul 20 08:54          【執行 who 命令結果】
tsnien      pts/9           Jul 21 09:35 (140.127.138.31)
```

【B. 變更執行權利】

另一種方法是將 Shell script 設定成可執行檔，並且讓使用者可以直接執行，設定方法如下（**$ chmod u+x ex7_1**）：

```
$ chmod u+x ex7_1r
$ ls -l shell.scr
-rwxrw-r--    1 tsnien tsnien 61 Jul 21 09:50 ex7_1
```

既然 ex7_1 已成為可執行檔（對 User 本身而言），執行方法如下：**（$ ./ex7_1）**

```
$ ./ex7_1
        July 2005
Su Mo Tu We Th Fr Sa
                   1  2
3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
Thu Jul 21 10:23:06 CST 2005
root        :0              Jul 20 08:54
tsnien      pts/9           Jul 21 09:35 (140.127.138.31)
```

上述命令是因為 PATH 變數沒有指定到『目前目錄』（Current directory，.），而必須採用相對路徑來表示命令的所在位置。我們藉由設定 PATH 變數，讓它可以到目前目錄下搜尋命令，設定方法如下：（請編寫到 .bash_profile 檔案內）

```
$ PATH=$PATH:.          【PATH 變數內增加 ":." 為目前目錄】
$ export PATH
$ echo $PATH
/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/tsnien/bin:.
```

【C. 子 shell 執行】

上述範例是將命令稿設定成可執行檔，並直接執行該命令稿，如果沒有指定在哪一個 Shell 執行該命令稿，就表示在目前 Shell 下執行該命令稿。其實，我們可以在命令稿內指定（或呼叫）一個『子 shell』（Sub-shell）來執行，指定方法是加在命令稿的第一行（必須

第一行才行)，寫下指定的 Sub-shell 名稱，格式為 #! Sub-shell。範例如下：**($vi ex7_1)**

```
#! /bin/bash
# A simple shell script
# file name:shell.scr
cal
date
who
```

讀者可以試試其他的 Shell (如 /bin/sh、/bin/csh)，可發現執行的結果都會相同，但對其他的命令稿就不盡然。

13-1-2 Shell script 變數

之前第六章介紹過，Shell 環境有環境變數、自訂變數與位置變數等三種類型的變數，Shell script 也都可以直接存取它們，其中又以環境變數與 Shell 環境設定最有關係，因已在第六章介紹過，這裡不再重複。然而對程式設計而言，位置變數與自訂變數則較有關聯，以下分別介紹這兩種變數的特性。

【A. 使用者自訂變數】

任何程式語言都需要宣告變數，程式運作也是針對變數做一些處理，並得到最後的結果。Shell script 也不例外，可依照使用者的需求產生各種變數，來輔助程式的執行，此類型變數大多使用完即拋棄，而與系統環境無關，通常稱為『自訂變數』。一般 Shell (如 csh 或 sh) 的自訂變數大多無法指定變數型態，也不需要事先宣告直接採用即可。我們用一個簡單的範例，來說明自訂變數的產生與使用方法，程式如下：**($vi ex7_2)**

```
#! /bin/bash
# User defind variables
int=5
float=5.5
char=character
string="Good luck to you"
echo "integer = $int"
echo "float = $float"
echo "char = $char"
echo "string = $string"
```

其中 int=5 (注意，等號前後不可以空格)，表示產生一個名稱為 int 的變數，並且直

接將 5 填入該變數內。另外，變數 float、chat 與 string 都是以相同的方法產生。其實填入變數的內容是不分整數、浮點數、字元或字串，而是將所欲填入的內容以 ASCII 碼方式填入，至於該變數為何種型態，待爾後使用時再去判斷處理。當變數做某些處理時，必須在變數的前面增加一個錢號($)，代表變數而非字串。然而所謂的「處理」可能是輸出(echo)、運算 (expr)、條件判斷 (test) 等等 (容後介紹)。上述範例執行結果如下：

```
$ ex7_2                          【執行 ex7_2 程式】
integer = 5
float = 5.5
char = character
string = Good luck to you
$ echo $string                   【觀察 string 變數內容】

$ echo $char                     【觀察 char 變數內容】
```

最後兩道的命令，表示執行完 variable1 程式後，觀察是否有自訂變數遺留下來，其中 string 與 char 為 variable1 程式內的自訂變數，測試後發現這兩個變數都不存在。也就是說，當系統產生一個 process (行程，第 12 章介紹) 來執行 Shell script 程式時，執行完後該 process 便不存在，它所產生的自訂變數也不會遺留下來。

【B. 位置變數】

位置變數 (Positional variable) 非常類似 C 語言的 argv 與 args 變數。簡單的說，位置變數是命令行所輸入的字元變數，以 $0 (如 C 語言的 argv[0]) 表示命令本身、$1 為第一個攜帶參數、$2 為第二個參數，依此類推。而且以 $* 表示所有參數 ($1 以後的字元變數)，$# (如 C 語言的 args) 表示所攜帶參數的個數。範例如下：

命令行為：$ cp /etc/passwd file_1

相關位置變數：

$0 = cp
$1 = /etc/passwd
$2 = file_1
$* = /etc/passwd file_1
$# = 2

由上述可以看出，位置變數就是一般程式語言所說的『位置參數』(Position parameter)，

也就是執行某一程式時，給予相當的參數，而這些參數放置於命令行內，再利用它放置的次序來分辨。接下來，我們用一個簡單的外殼程式來介紹位置變數的使用方法，外殼程式如下：

（ex7_3，利用 vi 編輯）

```
#! /bin/sh
# Illustrate the use of position parameter

echo "first parameter = $1"
echo "second parameter = $2"
echo "third parameter = $3"
echo "your typed $# argnment: $*"
```

執行結果如下：

```
$ chmod u+x ex7_3
$ ./ex7_3 Good Luck To You                    【執行 ex7_3 程式】
first parameter = Good
second parameter = Luck
third parameter = To
your typed 4 argument: Good Luck To You
```

由上述範例可以看出，$1 為第一個位置參數(Good)，$2 為第二個位置參數(Luck)；另外，$# 為參數的數目（4），$* 為所有參數。

13-2 基本敘述句

基本上，所有 Shell 命令都是 Shell script 的敘述句，譬如 ls、cp、cat、date 等等，或是由 Shell script 所新建立的命令皆是。為了提高 Shell script 程式的變化性，除了 Shell 命令外，其他都必須增加一些基本敘述句來輔助，以下介紹一些較常用的敘述句。

13-2-1 輸出敘述 – echo

echo 敘述是將資料輸出到終端機上，其資料可以是變數內容或文字訊息，範例如下**($vi ex7_4)**：

```
#! /bin/bash
# file_name:ex7_4
echo $USER
echo "Good luck to you"
echo Good luck to you
```

```
echo "Date: `date`"
```

上述範例中，第一個敘述是將變數（$USER）內容輸出；第二與第三個敘述是直接將訊息輸出，其中雙引號可有可無；第四個敘述是將命令（date）的結果一併輸出，注意必須使用反向單引號（`）。執行該命令稿之後的結果如下：

```
$ ex7_4                                    【執行 ex7_4 程式】
tsnien
Good luck to you
Good luck to you
Date: Thu Jul 21 14:10:25 CST 2005
```

13-2-2 輸入敘述 - Read

Read 敘述是由鍵盤讀入資料，並存入變數內，範例如下：**（$vi ex7_5）**

```
#! /bin/bash
# file_name: ex7_5
echo –n “What is your name =>”
read name
echo “Welcome, $name”
```

其中 echo –n 表示沒有換行（Enter 輸入）的功能，當系統執行到 read name 時，會暫停並等待輸入，直到有 Enter 輸入，再將鍵盤輸入文字放入 name 變數內。另外，變數並不需要事先宣告，使用時就直接採用已包含宣告的功能。執行結果如下：

```
$ ex7_5                                    【執行 ex7_5 程式】
What is your name =>Tien-shou
Welcome, Tien-shou
```

利用 read –p 語法，也可以達到相同結果，如下**（$vi ex7_5_1）**：

```
#! /bin/bash
# file name: ex7_5-1
read -p "What is your name =>" name
echo "Welcome, $name"
```

13-2-3 設定位置變數 – Set

執行某些命令會出現類似欄位格式的多筆資訊，可利用 set 命令將訊息指定到位置變數內。以執行 date 命令為例，它會出現星期、月、日、時間等訊息，如利用 set 命令指定位置變數，其關係如圖 13-1 所示，程式範例如下（**$vi ex7_6**）：

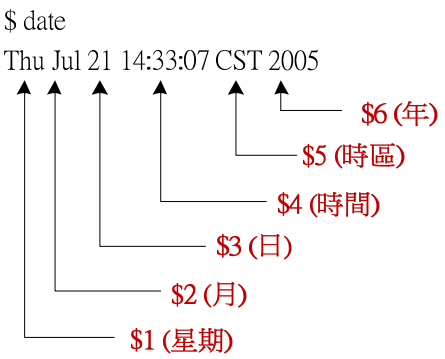

圖 13-1 設定位置變數範例

```
#! /bin/bash
# file_name: ex7_6
set `date`                          【注意必須使用反向單引號】
echo "Time: $4 $5"
echo "Day: $1"
echo "Date: $3 $2 $6"
```

執行結果如下：

```
$ chmod u+x ex7_6                   【設定成可執行檔】
$ ex7_6                             【執行 set_scr 程式】
Time: 15:07:17 CST
Day: Thu
Date: 21 Jul 2005
```

13-2-4 數學運算 – expr

Shell script 同樣具有加（+）、減（-）、乘（*）、除（/）與求餘數（%）的運算功能，但執行這些運算時，都必須利用 expr 敘述句來完成（大部分的 Script 皆是如此），可先在提示符號（$）底下測試運算功能，如下：

```
$ 3 + 4                             【系統判斷 3 是輸入命令】
-bash: 3: command not found
$ echo 3 + 4                        【輸出 3 + 4 訊息】
3 + 4
$ echo expr 3 + 4
expr 3 + 4
$ echo `expr 3 + 4 `                【expr 執行運算功能，反向單引號】
7
```

第一個命令系統判斷 3 為輸入命令；第二與第三個命令僅輸出訊息(3 + 4 或 expr 3 + 4)；第四個命令才是真正執行運算功能，其中 expr 運算敘述必須利用反向單引號(`)包括起來。以下用一個較複雜的範例，來實作各種運算子，範例如下：**($vi ex7_7)**

```
#! /bin/bash
# file_name: ex7_7
echo "Test operator"
echo -n "Please enter first number =>"
read number1
echo -n "Please enter second number =>"
read number2
echo "Calculated results:"
sum=`expr $number1 + $number2`                    【等號前後不可以有空格】
echo "      Sum = $sum"
dele=`expr $number1 - $number2`
echo "      Dele = $dele"
echo "      Mult = `expr $number1 \* $number2`"      【\ 抑制 * 特殊功能】
echo "      Div = `expr $number1 / $number2`"
echo "      Rem = `expr $number1 % $number2`"
```

上述範例比較特殊的地方，星號 (*) 在 Shell 環境裡是代表 0 個或多個任何字元 (請參考 4-6-2 節說明) 的意思，因此必須在它的前面加上反斜線 (\ )，表示抑制其特殊功能，並恢復原來字元的表示法 (* 代表乘法)。上述範例執行結果如下：

```
[tsnien@localhost shell]$ ex7_7
Test operator
Please enter first number =>320
Please enter second number =>200
Calculated results:
   Sum = 520
   Dele = 120
   Mult = 64000
   Div = 1
   Rem = 120
```

13-3 條件敘述句

無論選擇性或重複性流程控制，大多需要條件判斷敘述，一般程式語言大多如此。在 Script 語言裡，條件敘述大多利用 test 命令，或一對中括號 ([ ]) 包起來表示，有下列三種主要類型：

- 數值比較
- 字串比較（或測試）
- 測試檔案系統

以下分別介紹上述三種條件判斷敘述方法。

13-3-1 數值比較敘述

顧名思義，數值比較為兩個數值之間的比較結果，來作為條件判斷的依據。如果以變數表示的話，則判斷變數內存放數值（Shell 變數沒有指定變數型態，如 int、string 等等）。數值比較的格式如下：

```
test $integer1 –eq $integer2
```

其中 –eq 表示『相等』的比較旗號。比較的結果會傳回『真』(1) 或『假』(0)，test 運算子如表 13-1。

表 13-1 數值比較的運算子

比較旗號	說 明
-eq	等於（=，若兩數相等則傳回 “真（1）”，否則為 0）
-ne	不定於（!=）
-lt	小於（<）
-le	小於或等於（<=）
-gt	大於（>）
-ge	大於或等於（>=）

我們用一個簡單範例來驗證 test 命令語句的使用方法。下列程式為輸入兩個數值之後，再判斷兩數是否相同，並輸出其結果，程式範例如下：**（$vi ex7_8）**

```
#! /bin/bash
# file_name: ex7_8
echo -n "Enter first number =>"
read n1
echo -n "Enter second number =>"
read n2
if test $n1 -eq $n2
```

```
   then
          echo "Two numbers are eqaul"
   else
          echo "Not equal"
fi
```

執行結果如下：

```
$ ex7_8
Enter first number =>500
Enter second number =>500
Two numbers are eqaul
```

一般 Shell script 還提供另一種不需要 test 的語法，而用中括號包括起來的方法。值得注意的是，左括號的右邊與右括號的左邊都必須留空格，否則系統會將中括號當作變數名稱的一部分，程式範例如下：**($vi ex7_9)**

```
#! /bin/bash
# file_name: equal_scr
echo -n "Enter first number =>"
read n1
echo -n "Enter second number =>"
read n2
if [ $n1 -eq $n2 ]        #[ 之後 與 ] 之前需空格
   then
          echo "Two numbers are eqaul"
   else
          echo "Not equal"
fi
```

讀者可以自行練習看看，執行的結果應該與上個範例相同。

13-3-2 字串比較敘述

字串比較就是比較兩個字串作為判斷條件的依據，只是比較字串與比較數值之間有很大的不同，一般比較字串也有比較字串是否相等、或字串是否為空字串、以及利用 ASCII 碼比較兩字串的大小，如表 13-2 所示。

表 13-2 字串比較的運算子

比較運算子	說　　明

=	比較兩字串是否相符合，如 str1 = str2。
!=	不符合（即是兩字串內容不相同），如 str1 != str2。
<	小於（依 ASCII 碼比較大小），如 str1 < str2。
>	大於（依 ASCII 碼比較大小），如 str1 > str2。
-n	不是空字串（字串長度大於 0），如 –n str1。
-z	空字串（字串長度為 0），如 –z str1。

接下來，我們用 string_scr 程式範例來驗證字串比較，程式中希望輸入 Good 字串，再比較是否輸入正確。這裡可看到採用變數與字串之間的比較關係，範例如下：**（ $vi ex7_10 ）**

```
#! /bin/bash
# file_name: string_scr
echo "Test string is: Good"
echo -n "Please enter test string =>"
read n1
if [ “$n1” = "Good" ]        #[ 之後 與 ] 之前需空格
   then
        echo "Corrected input"
   else
        echo "Not correct input"
fi
```

執行結果如下：

```
$ ex7_10
Test string is: Good
Please enter test string =>Good
Corrected input
$ ex7_10
Test string is: Good
Please enter test string =>lucky
Not correct input
```

13-3-3 檔案屬性測試敘述

另一個重要的條件判斷，即是檔案屬性的檢查，這在管理系統上非常有幫助，基本命令格式如下：

```
test –d file_1
```

其中 –d 為測試選項之一，如果 file_1 是目錄的話，則傳回『真』。表 13-3 是所有能測試的屬性。

表 13-3 檔案測試選項

選　　項	說　　明
-l	該檔案是否屬於鏈結檔案，如是則傳回『真』。
-d	如果是目錄則傳回真。
-e	如果檔案存在則傳回真。
-f	如果檔案存在並且是一般檔案則傳回真。
-g	如果檔案存在並且是特定群組可執行的，則傳回真。
-r	如果檔案存在並且可讀的，則傳回真。
-s	如果檔案存在且存有資料，則傳回真。
-w	如果檔案存在且可寫入資料，則傳回真。
-x	如果檔案存在且可執行，則傳回真。
-nt	比較兩檔案是否較新（修改時間），如 file1 –nt file2。
-ot	比較兩檔是否較舊，如 file1 –ot file2。

我們再利用一個範例來說明檔案測試的功能，ex7_11 程式可測試所輸入的程式是目錄或一般檔案，以及使用者對該檔案的權限如何，程式範例如下：**（$vi ex7_11）**

```
#! /bin/bash
# file_name: test_file
if [ ! -e $1 ]; then
   echo "file $1 does not exist."
   exit 1
fi
if [ -d $1 ]; then
   echo -n "$1 is a directory that you may "
   if [ ! -x $1 ]; then
        echo -n "not "
   fi
   echo "search."
   elif [ -f $1 ]; then
        echo "$1 is a requlal file."
   else
        echo "$1 is a special file."
```

```
fi
if [ -O $1 ]; then
   echo "you own the file"
else
   echo "you do not own the file"
fi
if [ -r $1 ]; then
   echo "you have read permission on the file"
fi
if [ -w $1 ]; then
   echo "you have write permission on the file"
fi
if [ -x $1 ]; then
   echo "you have execute permission on the file"
fi
```

上述範例中可以發現 if/then 編寫格式有稍微改變一些，標準 then 語句是另開始一行，如果與 if 同一行的話，則需利用分號（;）隔開。也就是說，在 Shell script 程式設計中，分號（;）表示另開始一行的意思；讀者可自行測試看看，是否屬實。其執行結果如下：（假設目錄底下有 set_scr 檔案，並測試該檔案的屬性如何）

```
$ ex7_11 set_scr
set_scr is a regular file.
you own the file
you have read permission on the file
you have write permission on the file
you have execute permission on the file
$
```

13-3-4 條件組合敘述

一般程式語言大多可以利用『且』（and，&&）與『或』（or，||）兩個邏輯運算子，來整合多個條件組合。其中 && 表示兩個條件都成立則為『真』（True）；|| 表示兩條件中任一條件成立則為『真』。Shell script 也不例外，同樣有這兩個運算子，程式範例如下：**（$vi ex7_12）**

```
#! /bin/bash
# file_name: or_scr
#if (test -r $1) && (test -x $1)
if [ -r $1 ] && [ -x $1 ]; then
   echo "You have read/write to $1 Permission."
else
   echo "You donot permission for $1."
```

```
fi
```

執行結果如下：(假設 args 檔案存在並測試它)

```
$ ex7_12 args
You have read/write to args Permission.
```

13-4 選擇性敘述

一般 Shell script 大多提供有下列選擇性敘述：

- if/then/elif/else/fi
- case/esac

以下分別介紹之。

13-4-1 if 選擇結構

if 程式結構為最簡單且最常用的選擇性敘述，若是『條件』(Condition) 成立的話，則執行某一程式區塊 (一些敘述句的組合)，但它也有幾種變形，以下分別介紹之。

【A. if/then/fi 敘述】

Shell script 的程式區塊與一般程式語言之間有很大的不同。以 C 語言為例，程式區塊是利用大括號包括起來 (如 {.....})，Shell script 並沒有使用到大括號的功能，然而 if 敘述是利用 then 語句表示程式區塊的開始，fi 表示程式區塊的結束。至於其他敘述 (case/for/while 等等) 亦大同小異，多半利用敘述句 (如 case) 的反向 (如 esac) 為程式區塊的結束。if 簡單格式如下：

```
if 條件判斷
then
        命令敘述區段
fi
```

上述 if 程式意思是條件成立的話，則執行 then 與 fi 所涵蓋的程式區塊。基本上，Shell script 並不理會空白格與空白行的數量，也就是說，空一格或空兩格以上是沒有差別的，空一行與空多行亦同。但某些關鍵字 (如 then 或 if 等等) 就必須獨立一行，如果需要與其它關鍵字結合成一行的話，必須使用分號 (;) 分隔。也就是說，分號 (;) 代表分行 (或另

開一行）的意思。下列是將 if 敘述重寫的格式：

```
if 條件判斷; then
        命令敘述區段
fi
```

更簡潔的格式如下：

```
if 條件判斷; then; 命令敘述區段; fi
```

【B. 範例 ex7_13】

此範例是測試某一檔案是否為一般檔案，並顯示其測試結果。**($vi ex7_13)**

```
#!/bin/sh

if test -f $1
then
    echo "$1 is a requlare file"
fi
```

執行結果：

```
$ ex7_13 add.scr
add.scr is a requlare file
```

【C. if/else 敘述】

if/else 敘述表示條件成立的話，執行 then 程式區塊，否則執行 else 程式區塊，格式如下：

```
if 條件判斷
then
        命令敘述區段
else
        命令敘述區段
fi
```

【D. if/elif 敘述】

if/elif 敘述表示條件判斷不成立的話，再判斷其他條件（else if）是否成立，同樣的情

況可以一直延伸下去，如此一來便可以整合多個條件判斷。每一次條件判斷的結果（成立或不成立），都可以建立它相對應的程式區塊，最後不管 if/elif 延伸多長，還是 fi 作為結束的語句。基本格式如下：

```
if 條件判斷
then
        命令敘述區段
        .......
elif 條件判斷
then
        命令敘述區段
        ......
.....
else
         命令敘述區段
         ........
fi
```

【E. 範例 ex7_14】

此範例一樣測試某一檔案是否為一般檔案，但此程式先檢測是否有指定被測試檔案名稱，接著再測試是否為一般檔案，程式如下：**（$ vi ex7_14）**

```
#!/bin/sh
if test $# -ne 1    # 測試輸入格式是否正確
then
    echo "Usage:$0 test_file"
elif test -f $1
then
    echo "$1 is a regular file"
else
    echo "$1 is not regular file"
fi
```

13-4-2 case 選擇結構

case 敘述與 C 語言的 switch 很相似，都是依照某一變數的內容，再執行該內容的相對應程式區塊，但兩者之間還是有一些關鍵性的不同點。C 語言大多依照某一變數 (或表示式) 的數值當作判斷的依據，但 Shell script 的 case 不僅可依照數值之外，也可依照『圖樣』(Pattern) 作為判斷依據。圖樣可利用『正規式』(，如 *、?、[] 等等，第 4-6-2 節介紹) 表現形形色色的結果，它的變化性會比數值多。基本格式如下：

```
case variable in
pattern_1)  命令敘述區塊  ;;
pattern_2)
            statement 1;
            statement 2;
            statement 3;;
pattern_3)  命令敘述區塊  ;;
.....
esac
```

由上述中可以看出，由 case 到 esac 關鍵字之間為 case 的敘述區塊，其中每一 pattern 程式區塊都是利用兩個分號 (;;) 做結束記號，而敘述之間則是以一個分號 (;) 分隔。

【A. 範例 ex7_15】

此範例以數值作為選擇的依照 (如同 C 語言功能)，其功能是將輸入數值轉換成英文數字。程式範例如下：**($ vi ex7_15)**

```
#! /bin/bash
# file_name: ex7_15
if [ $# -ne 1 ]; then
   echo "Usage: $0 digital"
   exit 1
fi
case $1 in
   1) echo -n "One";
        echo "一";;
   2) echo -n "Two    "; echo "二";;
   3) echo -n "Three    "; echo "三";;
```

```
   4) echo -n "Four    "; echo "四";;
   5) echo -n "Five    "; echo "五";;
   6) echo -n "Six    "; echo "六";;
   7) echo -n "Seven    "; echo "七";;
   8) echo -n "Eight    "; echo "八";;
   9) echo -n "Nine    "; echo "九";;
esac
```

執行結果如下：

```
$ ex7_15
Usage: ./ex7_15 digital
$ ex7_15 6
Six    六
```

【B. 範例 ex7_16】

範例二是以正規式作為 case 判斷的依據，在正規式中的星號（*）代表零個或一個以上的任何字元。下述範例由判斷輸入姓名（FirtName_SecondName）中，尋找符合的『姓』（SecondName），並判斷該姓名為哪一家族的成員，程式範例如下：**（$vi ex7_16）**

```
#! /bin/bash
# file_name: ex7_16
echo "Test Nien, Lin, Chung, Lie, Chou or Wu family"
echo " "
if [ $# -lt 1 ]; then
    echo "Usage: $0 FirstName_SecondName"
    exit 1
fi
case $1 in
    *Nien) echo "Your Nien family";;
    *Lin) echo "Your Lin family";;
    *Chung) echo "Your Chung family";;
    *Lie) echo "Your Lie family";;
    *Chou) echo "Your Chou family";;
    *Wu) echo "Your Wu family";;
    *) echo "Not test family";;
esac
```

我們執行該程式，以 TienShou_Nien 為引數。Case 比對 *Nien，執行結果如下：

```
$ ex7_16 TienShou_Nien
Test Nien, Lin, Chung, Lie, Chou or Wu family

Your Nien family
$ ex7_16 ShunJie_Wu
Test Nien, Lin, Chung, Lie, Chou or Wu family

Your Wu family
```

13-5 重複性敘述

所謂重複性程式結構大多指迴圈敘述，Shell script 的迴圈敘述有 for、while、until 與 select 等四種類型，其中 while 和 until 敘述與 C 語言比較類似，但 for 和 select 幾乎與 C 語言的 for 迴圈完全不同，以下分別介紹之。

13-5-1 for 迴圈結構

一般程式語言的 for 迴圈大多有一個計數器（counter），計算迴圈內執行的次數。然而 Shell script 的 for 迴圈則完全不同，它是依照某一序列內的元件數量，再依據每一個元件的數值執行迴圈一次，元件的數量就代表迴圈的執行次數。for 迴圈的基本結構如下：

```
for var in List
do
        commands
        commands
done
```

其中，List 表示一連串的數值（或序列），變數 var 連續取代執行 List 串列中的數值（或其他格式的內容），每取代一個數值則執行迴圈一次，至於程式的實體則包含在關鍵字 do 與 done 之間。

【A. 範例 ex7_17】

此範例是列出目前目錄底下的一般檔案名稱，程式範例如下：**（ $vi ex7_17 ）**

```
#! /bin/bash
# file_name: ex7_17
for var in `ls`
do
   if [ -f $var ]; then
```

```
        echo "$var is a requlal file."
    fi
done
```

在 for var in `ls` 敘述中，ls 是利用兩個『反向單引號』(`) 包起來，表示執行 ls 命令的意思 (第 4-3-3 節介紹); 整個敘述的功能是執行 ls 命令後產生一個串列的結果，而變數 var 取用串列中元件，每取一個元件則執行 do 與 done 之間的程式區塊一次。執行結果如下：

```
$ ex7_17
and_scr is a requlal file.
args is a requlal file.
……
```

13-5-2 select 迴圈結構

select 敘述不但具有迴圈的功能，也具有選擇性功能，運作情況與 for 迴圈有點相同。我們先觀察程式結構，再來探討其功能為何，其程式結構如下：

```
select var in List
do
        commands
        commands
        …….
done
```

select 程式結構與 for 非常相似，但兩者之間的運作還是有很大的差異。相同的 List 表示一串列的元件，但 select 敘述被執行時，系統會將這一串列元件當作選項，並以環境變數 PS3 作為選項輸入的提示。當使用者在 PS3 提示下輸入某一選項時，選項的元件即被存入 var 變數內，並且執行 do 與 done 之間的程式區塊；執行完畢以後，再回到選項提示下，所以 select 不但具有選擇性功能，也具有迴圈性的功能。

【A. 範例 ex7_18】

本範例功能是選擇測試目錄下檔案是否為一般檔案，程式範例如下：**($vi ex7_18)**

```
#! /bin/bash
# file_name: ex7_18
PS3="Selection =>"
select var in `ls`
```

```
do
   if [ $var ]; then
         if [ -f $var ]; then
               echo "$var is a requlat file."
         else
               echo "$var not a requlat file."
         fi
   else
         echo "Bad input"
   fi
done
```

程式一開始即設定環境變數 PS3(Selection =>)，再利用 ls 顯示目前目錄下所有檔案。當使用者選擇一選項後，立即進入 do 與 done 之間的程式區塊，其執行結果如下：

```
$ ex7_18
1) args          3) dir_2          5) file_2
2) dir_1         4) file_1         6) select_scr
Selection =>1
args is a requlat file.
Selection =>3
dir_2 not a requlat file.
Selection =>                                    【Ctrl+D 結束】
$
```

13-5-3 while 迴圈結構

Shell script 的 while 迴圈功能幾乎與 C 語言的 while 相同，唯一差別在於條件判斷的格式。 while 敘述在條件成立時，會執行 do 與 done 之間的程式區塊，執行完後再回來測試條件判斷，如條件再成立就必須再執行程式區塊，如此反覆，直到條件不成立再跳出迴圈。基本格式如下：

```
while 條件判斷
do
      命令區塊
done
```

【A. 範例 ex7_19】

此範例是以 1 加到 100 來驗證 Shell script 的 while 敘述功能，程式範例如下：($ vi

ex7_19)

```
#! /bin/bash
# print 1+2+3, ...+100
count=1
sum=0
while [ $count -le 100 ]; do
    sum=`expr $sum + $count`
    count=`expr $count + 1`
done
echo "1+2+3+, ...+100 = $sum"
```

執行結果如下：

```
$ ex7_19
1+2+3+, ...+100 = 5050
```

13-5-4 until 迴圈結構

until 敘述正好與 while 迴圈相反，也就是說，當條件不成立時，會執行迴圈區塊，直到條件成立才跳出迴圈。其命令格式如下：

```
until 條件判斷
do
        命令區塊
done
```

【A. 範例 ex7_20】

此範例是當使用者輸入任何文字，程式會回應相同文字，直到出現 quit 字元為止，程式如下：**($vi ex7_20)**

```
#! /bin/bash
# echo input character until "quit"
echo -n "Input your character (quit) =>"
read string1
until test $string1 = "quit"; do
        echo "Your input is $string1"
        echo -n "Input again (quit) =>"
        read string1
done
echo "You're Welcome"
```

執行結果如下：

```
$ ex7_20
Input your character (quit) =>Good
Your input is Good
Input again (quit) =>luck
Your input is luck
Input again (quit) =>quit
You're Welcome
```

13-5-5 break 與 continue 命令

一般程式語言的 for、while 與 until 迴圈，大多會利用 break 與 continue 敘述來中斷或延續迴圈的執行。同樣的，Shell script 迴圈也擁有這兩個命令的功能，如果是巢狀迴圈的話，也可在 break 或 continue 敘述後增加一個數字引數 (如 break 2 或 continue 2)，指定中斷或延續第幾個迴圈。

13-6 字串處理

字串處理在 Shell Script 裡算是比較複雜且重要的工具，有了這些工具才能顯現出它與一般程式語言不同的地方。這裡僅介紹一些入門觀念，如欲更深入地研究，可能須參考有關 Shell 或 Perl 語言工具書籍。

13-6-1 字串串接處理

基本上，Shell 變數內都是以字串格式儲存，只不過當它被取出來使用時，才再決定要以何種資料型態處理。而字串的串接方法就顯得非常容易，只要將兩個變數排列一起，就是串接處理，操作範例如下：

```
$ name=Data                 => 設定 name 變數內容為字串 Data
$ number=5                  => 設定 number 變數內容為字串 5
$ fileName=$name$number     => fileName 內容為兩字串串接
$ echo $fileName            => 顯示 fileName 變數內容
Data5
```

(A) 範例 ex7_21

許多情況下，檔案名稱需要加入日期，才能顯現出該檔案所記錄資料的產生時間。命令 date 為顯示日期功能，可利用 %Y、%m 與 %d 表示所欲顯示的年、月與日的格式，因此我們可利用 `date +%Y_%m_%d`(為反向單引號)來擷取當天的年、月、日，操作範例如下：

```
[tsnien@Secure-1 ~]$ today=`date +%Y_%m_%d`
[tsnien@Secure-1 ~]$ echo $today
2009_07_21
```

假設我們需要每天記錄某些事件，檔案名稱最好與當天日期相吻合，則產生該檔案的程式如下 **($vi ex7_21)**：

```
#! /bin/bash
# file name: ex7_21
mainName=secure                          => 設定 mainName 變數內容
today=`date +%Y_%m_%d`                   => 設定 today 變數內容
fileName=$mainName$today                 => 兩字串串接
touch $fileName                          => 利用 touch 產生空白檔案
```

執行結果如下：

```
[tsnien@Secure-1 ~]$ ex7_21
[tsnien@Secure-1 ~]$ ls |grep secure
secure2009_07_21                         => 已產生名稱加日期的檔案名稱
```

(B) 範例 ex7_22

某些紀錄檔案並不需要日期，而是以週期性的序號標示，譬如由 secure0、secure1..... 到 secure 5，依序產生。當紀錄檔產生到 secure5 之後，下一個再由 secure0 開始。我們需利用一個環境變數 (Count)，來紀錄目前所產生的序號。每次要產生記錄檔案之前，先測試該變數是否已超過 5，如到達 5，則由 0 再開始，否則累加 1。程式範例如下 **($vi ex7_22)**

```
# file name: ex7_22
mainName=secure
max=5
if test $Count -gt $max
    then
        Count=0
    else
        Count=`expr $Count + 1`
```

```
fi
fileName=$mainName$Count
if test -f $fileName
    then
        `rm $fileName`
fi
touch $fileName
```

執行此範例之前須產生一個環境變數 Count，並必須將它 export 成整體變數，才會有效，如下：

```
[tsnien@Secure-1 ~]$ export Count=0     => 產生並 export 變數 Count
[tsnien@Secure-1 ~]$ ex7_22             => 執行範例
[tsnien@Secure-1 ~]$ ls -1 |grep secure  => 檢視是否產生 secure1 檔案
-rwxrw-r-- 1 tsnien tsnien 244  7 月  21 11:34 make_secure
-rw-rw-r-- 1 tsnien tsnien   0  7 月  22 09:46 secure1
```

13-6-2 字串替換處理 –{:}

到目前為止，我們對環境變數應該有一點初步的認識，為何它是設定使用者環境的主要因素？這是因為規劃環境程式係利用 Shell Script 編寫而成(如第六章所介紹)，當執行設定程式時，則須按照某些環境變數的內容來決定執行結果如何。這樣雖然方便，但所需的環境變數是否已產生，在執行程式之前我們可能一無所悉，如此可能會造成錯誤。因此，我們可能需要某些字串處理工具，基本上它包含下列功能：

- ✓ 判斷變數是否存在(變數已被定義與空值並不相同)
- ✓ 設定變數的預設值
- ✓ 移除變數中某些式樣與相符部份

當然上述功能是相互混合的。譬如我們欲變更某一變數的內容時，須先測試該變數是否存在，存在與否處理的方式當然不同。字串處理的符號是兩左右大括號({...})包著變數名稱，外邊再用一個錢記號表示變數，如 ${....}，括號內決定處理哪些事物。表 13-1 為字串替換處理符號。

表 13-1 字串替換處理符號

字串處理符號	功　　能

${varName:-word}	若 varName 存在而且不是空值，則傳回該值，否則傳回 word。功能是測試某個變數是否被定義，若無則回傳預設值。
${varName:=word}	若 varName 存在而且不是空值，則傳回該值，否則將它設定為 word，並回傳。功能是測試某個變數是否被定義，若無則產生並設定初值 word。
${varName:+word}	若 varName 存在而且不是空值，則傳回 word，否則傳回空值（null）。功能是測試某個變數是否存在。
${varName:?message}	若 varName 存在而且不是空值，則回傳該值，否則印出 varName 與 message 訊息，並結束當時的 Shell 程式。功能是測試某個變數是否被定義，以免造成執行上錯誤。
${varName:offset} ${varName:offset:length}	傳回 $varName 變數內由 offset 位置開始（由 0 開始計算），共計 length 長度的字串。如無指定 length，則傳回 offset 位置以後的字元。功能是切割字串。譬如 count=1234567，則 ${count:3}，則傳回 4567。如 ${count:3:2}，則傳回 45。

在表 13-1 中，每一個敘述句都包含冒號（如 ${varName:-word}），其實除了最後一個敘述句之外，冒號是可以省略的。如沒有冒號，則功能是將『若 varName 存在而且不是空值』改為『若 varName 存在』。譬如 ${varName=word}，則表示『若 varName 存在，則傳回該值，否則設定成 word，並回傳』。

(A) 範例 ex7_23

在 13-3-1 節範例中，我們依據環境變數 Count 內容，來決定所欲產生的檔案名稱如何，如果 Count 變數不存在的話，可能會造成執行的錯誤。下一個範例是利用 ${Count:=1} 測試 Count 變數是否存在，如存在的話，則回傳該變數的內容，如不存在的話，則回傳空值。程式範例如下：**($vi ex7_23)**

```
# file name: ex7_23
mainName=secure
max=5                                   => 最高數值為 5
```

```
echo "Start Count = $Count"                  => 顯示目前 Count 變數的內容
now=${Count:=1}                              => 將測試 Count 變數的結果存入 now
if [ -n "$now" ]                             => 測試 now 變數內是否空值
    then                                     => 如不是空值，則執行下列語句
        if [ $now -ge $max ]                 => now 是否超過或等於 5
            then
                now=0
            else
                now=`expr $now + 1`          => now 變數累加一
        fi
        fileName=$mainName$now               => 製作檔案名稱，如 secure3
        if [ -f $fileName ]                  => 該檔案存在則刪除
            then
                rm $fileName
        fi
        touch $fileName                      => 產生新的檔案
    else
        echo "The Count variable is not Exist, no success"
fi
export Count=$now                            => 將新的 Count 內容輸出成環境變數
echo "Last Count = $Count"                   => 顯示最後的 Count 內容
```

上述範例中比較特殊的地方。是測試某變數的內容是否是空值，它的語法是 [-n “$now”]，利用雙引號包起來表示字串的意思，字串才可以判斷他的長度是否為 0，如果是 0 的話，則表示內容是空值。編輯完後，我們的執行結果如下：

```
[tsnien@Secure-1 ~]$ ex7_23                  => 還未產生 Count 之前執行
Start Count =
The Count variable is not Exist, no success
Last Count =
[tsnien@Secure-1 ~]$ export Count=3          => 產生環境變數 Count
[tsnien@Secure-1 ~]$ ex7_23
Start Count = 3
Last Count = 4
[tsnien@Secure-1 ~]$ ls secure*
ecure4
```

13-6-3 字串比對處理 –{#}

將某變數（varName）的內容以字串格式觀察，與某一字串樣式（pattern）做比對，如變數內容的起頭（#）或結尾（%）相符，則刪除取回最短（#、%）或最長部分（##、%%）。另一種處理方式，是變數內容與樣式比對，變數內相同部份由另一個樣式替換（/）。表 13-2 為字串替換處理符號。

字串比對符號	功 能
${varName#pattern}	若變數（varName）內容的開頭與 pattern 相符，則刪除相符部分（取最短者），並傳回變數剩餘部份。
${varName##pattern}	若變數（varName）內容的開頭與 pattern 相符，則刪除相符部分（取最長者），並傳回變數剩餘部份。
${varName%pattern}	若變數（varName）內容的結尾與 pattern 相符，則刪除相符部分（取最短者），並傳回變數剩餘部份。
${varName%%pattern}	若變數（varName）內容的結尾與 pattern 相符，則刪除相符部分（取最長者），並傳回變數剩餘部份。
${varName/pattern/string} ${varName//pattern/string}	若變數（varName）內容的開頭與 pattern 相符（varName/），則由 string 替換該相符部分。另一種是比對結尾部分(varName//)，如果與 pattern 相符，則由 string 字串替換。

(A) 範例 ex7_24：測試樣式比對

由表 13-2 中也很難看出樣式比對的用法，我們利用一個很簡單的範例來驗證它。假設某一變數 Name=/first/second/third/full.file.name，我們製作一程式，分別擷取出：

- ✓ full.file.name（##）：由起頭開始比對與樣式相同最長者。
- ✓ second/third/full.file.name（#）：由起頭開始比對與樣式相同最短者。
- ✓ /first/second/third/full.file（%）：由結尾開始比對與樣式相同最短者。
- ✓ /first/second/third/full（%%）：由結尾開始比對與樣式相同最長者。

程式範例如下（**$vi ex7_24**）

```
#! /bin/bash
# file name: ex7_24
Name="/first/second/third/full.file.name"
echo "Original name => $Name"
name1=${Name##/*/}
echo "From head and match all => $name1"
name2=${Name#/*/}
echo "From head and match one => $name2"
name3=${Name%.*}
echo "From tail and match one => $name3"
name4=${Name%%.*}
echo "From tail and match all => $name4"
```

執行結果如下：

```
[tsnien@Secure-1 ~]$ ex7_24
Original name => /first/second/third/full.file.name
From head and match all => full.file.name
From head and match one => second/third/full.file.name
From tail and match one => /first/second/third/full.file
From tail and match all => /first/second/third/full
```

(B) 範例 ex7_25：變更 Windows 路徑表示

在許多情況下，我們常會利用網路將 Windows 系統下某一目錄掛於 Linux 系統，但 Windows 路徑表示的斜線與 Linux 相反，我們可以編寫一程式將 Windows 的反斜線變更為正斜線，並加入所掛載於系統目錄位置，程式範例如下（**$ vi ex7_25**）：

```
#! /bin/bash
# file name: ex7_25
win="c:\windows\data\dir\file.type"
echo "Windows file name = $win"
name1=${win##[cC]:}                    => C 或 c 磁碟機皆可
echo "Delete device name => $name1"
name2=${name1//'\'/'/'}
echo "Change slash => $name2"
net="/network"
name3=${net}$name2
echo "Mount to /netowrk => $name3"
```

執行結果如下：

```
[tsnien@Secure-1 ~]$ ex7_25
```

```
Windows file name = c:\windows\data\dir\file.type
Delete device name => \windows\data\dir\file.type
Change slash => /windows/data/dir/file.type
Mount to /netowrk => /network/windows/data/dir/file.type
```

(C) 範例 ex7_26：變更 URL 成為檔案目錄位置

許多情況，我們必須將 URL 所表示轉換成檔案目錄表示法，程式範例如下（**$vi ex7_26**）：

```
#! /bin/bash
# file name: ex7_26
url="Linux.mis.csu.edu.tw"
echo "URL => $url"
name1=${url//'.'/'/'}
echo "Change to file name => $name1"
```

執行結果如下：

```
[tsnien@Secure-1 ~]$    ex7_26
URL => Linux.mis.csu.edu.tw
Change to file name => Linux/mis/csu/edu/tw
```

13-7 函數

Shell script 也具有『函數』(Function) 的功能，但它的呼叫和宣告與一般程式語言 (如 C 語言) 稍有不同。第一個不同點是區域與整體變數之間的界限如何；第二個不同點是函數呼叫時，如何攜帶引數的問題。如能克服這兩個不同點，使用 Shell script 的函數呼叫就沒有什麼困難。函數有兩種定義方法，如下所示：

```
Function function_name
{
        命令或程式區塊
        ……
}
或
function_name()
{
        命令或程式區塊
```

```
        ……
}
```

Shell script 並沒有像 C 語言有函數宣告的敘述，因 Shell script 是直譯程式，它會直接編譯執行，不像一般程式語言需經過兩階段的編譯動作，才會產生可執行檔。因此，Shell script 的函數宣告必須在函數呼叫之前完成，否則便會找不到函數程式的位置，下列範例應可窺視出函數的宣告與呼叫方法。

【A. 範例 ex7_27】

此範例會先宣告一個函數名為 disp，它的功能是畫出一條直線，接著主程式再呼叫該函數；程式如下：**($vi ex7_27)**

```
#! /bin/bash
# 1+2! + 3! +, ..., +k! = total
function disp
{
    echo "----------------------"
}

echo -n "A number of display line =>"
read k
while [ $k -gt 0 ]; do
    echo -n "$k    "
    disp
    k=`expr $k - 1 `
done
```

執行結果如下：

```
$ ex7_27
A number of display line =>4
4    ----------------------
3    ----------------------
2    ----------------------
1    ----------------------
```

13-7-1 區域變數與整體變數

一般的程式語言，其函數的程式實體都是獨立的記憶體空間，所宣告的變數一般稱之為區域變數 (Local variable)。在巢狀函數 (函數內再宣告函數) 的情況下，父函數與子函數的

區域變數是不會衝突的，除非使用者特意將某些父函數的變數宣告成整體變數（Global variable），父函數與子函數才可以共用這些變數，否則函數之間的變數是無關聯的。以上這些觀念與 Shell script 有點相反，Shell script 是除非有特殊聲明，否則所有的變數都是整體變數，且與 Shell 環境的特性有關。

Shell script 變數有環境變數、位置變數與自訂變數等三種類型，其中環境變數是系統產生並適用於任何函數，因此它必定是『整體』變數，至於位置變數會依照函數（無論子函數或命令程式）被呼叫時，攜帶引數的所在位置而定，並自動以 $1、$2、$3、...、$n 表示之；函數若再呼叫其他函數，其位置也都使用相同的方式表示，因此位置變數一定是『區域』變數。至於自訂變數，除非特別指定為區域變數（local 命令），否則一律為『整體』變數。以下幾個範例用來說明函數變數的特性。

【A. 範例 ex7_28】

本範例說明函數之間的自訂變數與位置變數為整體性或區域性關係，程式如下：**（$vi ex7_28）**

```
#! /bin/bash
# 位置變數為區域性, 自訂變數為整體性
funct()
{
    echo " "
    echo "Inside function "
    echo "       funct: $# agrs: $1 $2 $3"     # $# $1 $2 $3 為區域變數
    var1="in function"                          # var1 為整體變數
    echo "       var1 = $var1"
}

echo "Outside function"
echo "       $0: $# args: $1 $2 $3"
var1="Out function"
echo "       var1 = $var1"

funct arg1 arg2 arg3
echo " "
echo "Outside function"
echo "       var1 = $var1"
```

主程式內產生一個整體變數 var1，它可以被主程式與 funct 函數共用，執行結果如下：

```
$ ex7_28 par1 par2
Outside function
    ./function2: 2 args: par1 par2
    var1 = Out function

Inside function
    funct: 3 agrs: arg1 arg2 arg3
    var1 = in function

Outside function
    var1 = in function
```

由執行結果可以看出，主程式與函數的位置變數都是以 $1、$2、$3 與 $# 表示，但他們之間的內容並不相同，因為他們都是區域變數的關係；另外 $0 為整體變數，僅表示主程式的命令（ex7_28）。至於主程式所產生的 var1 亦為整體變數，函數內將 var1 設定為其他內容之後，主程式內的 var1 也會跟著改變。

【B. 範例 ex7_29】

如果需要區域變數的話，可以在產生變數的標頭加上 local 命令即可，範例如下：**（$vi ex7_29）**

```
#! /bin/bash
# local 產生區域變數
funct()
{
    echo " "
    echo "Inside function "
    local var1="in function"
    echo "       var1 = $var1"
}

echo "Outside function"
var1="Out function"
echo "       var1 = $var1"

funct
echo " "
echo "Outside function"
echo "       var1 = $var1"
```

上述程式中，函數 funct() 內產生一個 var1 的區域變數，雖然它與主程式的 var1 變數相同名稱，但所佔的記憶體空間不同。執行結果如下：

```
$ ex7_29
Outside function
    var1 = Out function

Inside function
    var1 = in function

Outside function
    var1 = Out function
```

主程式的 var1 內容與函數的 var1 並不相互衝突，這兩個變數是獨立的，他們只不過名稱相同罷了。

13-7-2 引數傳遞與傳回

在一般程式語言裡，呼叫函數常需要攜帶某些引數，來作為執行函數的輸入，而在函數執行完畢之後，也常會要求傳回某些執行的結果。簡單的說，這就是函數的輸入與輸出，當然 Shell script 函數也需要這些功能，但它的製作方式與其他語言略有不同。基本上，Shell script 並無法指定引數的資料型態，也無法利用 return 傳回數值，它可能會利用位置變數作為引數的輸入，或利用整體變數作為函數與主程式之間的資料交換；以下用兩個範例來說明傳遞方法。

【A. 範例 ex7_30】

此範例以位置變數($1)作為引數輸入，呼叫函數時，會在函數的後面加入一個引數(disp $count)，亦即將該變數的內容傳遞給函數引用，程式如下：**($vi ex7_30)**

```
#! /bin/bash
# argument($1) an input parameter
function disp
{
    t=$1
    while [ $t -gt 0 ]; do
        echo -n "*"
        t=`expr $t - 1`
    done
    echo " "
```

```
}

echo -n "A number of display line =>"
read k
count=0
while [ $count -le $k ]; do
    disp $count
    count=`expr $count + 1 `
done
```

執行結果如下：

```
$ ex7_30
A number of display line =>5

*
**
***
****
*****
```

【B. 範例 ex7_31】

針對函數回傳部份（即是函數執行後輸出），Shell script 大多將所欲回傳的數值存入整體變數內，主程式再從整體變數內取出，程式範例如下：**($vi ex7_31)**

```
#! /bin/bash
# 1+2! + 3! +, ..., +k! = total
fun1()
{
    t=$1; sum=1
    while [ $t -gt 1 ]; do
        sum=`expr $sum \* $t`
        t=`expr $t - 1`
    done
    return $sum
}

echo -n "Please enter a number =>"
read k
sum=1; total=0; i=1
while [ $k -ge $i ]; do
    fun1 $i
    total=`expr $total + $sum `
```

```
    i=`expr $i + 1`
done
echo "1 + 2! +3! +, ...+ $k ! = $total"
```

執行結果如下：

```
$ ex7_31
Please enter a number =>7
1 + 2! +3! +, ...+ 7 ! = 5913
```