Linux 隨身指南

40. Shell Scripts 程式設計

先前談到 shell（bash）的時候，我們曾說過它內建了一套程式語言。事實上，你可寫出「程式」（或曰「shell scripts」）來完成單一命令所不能達成的工作。如同任何好的程式語言，shell 也有「變數」、「條件判斷」（if-then-else）、迴圈、輸入、輸出 ... 等等元素。單就 shell script 這個議題，就足夠寫出一整本書了，所以這裡只提供剛好足以入門的基本參考資訊。至於完整的說明訊息，請參考 `info bash`，或是《bash shell 入門》（http://www.oreilly.com.tw/chinese/linux/learn_bashshell.html）。

40.1. 空格與斷行

bash shell scripts 對於空格與換列非常敏感，因為這套程式語言中的「關鍵字」（keywords），其實是由 shell 估算的命令，而命令的引數之間必須以空格隔開。類似的情況，出現於命令中間的換列（`\n`）字元會使得 shell 誤認為命令不完整。所以，學 shell script 的第一課就是養成正確的語法慣例，以免造成日後不必要的除錯麻煩。

40.2. 變數

「變數」（variable）讓你一個名稱來代表某種意義的數值或字串：

```
$ MYNAME="smith "
```

```
$ MYAGE=35
$ echo $MYNAME $MYAGE
smith 35
```

儲存於變數中的值，本質上其實都是字串，即使它們全部都是數字。不過，在必要時，shell 也能將這種「純數字字串」當成數值來處理：

```
$ NUMBER="10"
$ expr $NUMBER + 5        (+ 符號的左右側，至少要有一個空格)
15
```

在 shell script 裡表示變數值時，最好以雙引號包裝，以免造成執行期的錯誤。如果沒有雙引號，當 shell 遇到沒定義的變數（通常是因為打錯變數名稱造成的），或是變數值裡含有空格，就可能引發意料不到的後果，造成 script 行為錯亂。

```
$ FILENAME="My Document"              含有空格的檔名
$ ls $FILENAME                        列出來看看
ls: My: No such file or directory     糗了！ls 見到兩個引數
ls: Document: No such file or directory
$ ls -l "$FILENAME"                   這樣才對
My Document                           ls 只見到一個引數
```

如果變數名稱與另一個字串緊接在一起，則必須以一對曲括號包裝之，以免發生意料外的情況：

```
$ HAT="fedora"
$ echo "The plural of $HAT is $HATs"
The plural of fedora is           糗了！沒「HATs」這個變數
$ echo "The plural of $HAT is ${HAT}s"
The plural of fedora is fedoras   這才是我們要的結果
```

40.3. 輸入與輸出

Script 的輸出，主要是由 echo 與 printf 命令提供。174 頁《34. 螢幕輸出》已介紹過這兩個命令。

```
$ echo "Hello world"
Hello world
$ printf "I am %d years old\n" `expr 20 + 20`
I am 40 years old
```

Shell script 的輸入主要是靠 read 命令來取得，它每次從 stdin 讀入一列資料，並存入一個變數：

```
$ read name
Sandy Smith Enter
$ echo "I read the name $name"
I read the name Sandy Smith
```

40.4. 邏輯值與傳回值

程式的精華在於「條件判斷」與「迴圈」，而這其中的關鍵就在於「邏輯測試」（Boolean test），也就是分辨「真」（true）與「假」（false）。對於 shell，數值 0 代表「真」或「成功」，除此之外的其它數值一律視為「假」或「失敗」。

此外，任何 Linux 命令結束時，都會傳回一個代表執行結果的整數值給 shell，此值稱為「傳回值」（return value）或「結束狀態」（exit status）。你可用特殊變數 $? 來表示傳回值：

```
$ cat myfile
My name is Sandy Smith and
I really like Fedora Linux
$ grep Smith myfile
My name is Sandy Smith and    找到一處相符
$ echo $?
0                             所以結束狀態為「成功」
$ grep aardvark myfile
$ echo $?                     沒找到
1                             所以結束狀態為「失敗」。
```

許多 Linux 命令的傳回值具有特殊含意，各命令的 manpage 裡通常會說明傳回值的真正意義。唯一可以確定的是，傳回值 0 一定是代表成功，因為這是所有 Linux 命令的共識，同時也是 POSIX 標準的規定。

test 和「 [」

對於只涉及數值和字串的邏輯算式，可用 bash shell 內建的 test 命令來估算其邏輯值。如果估算結果為「真」，則 test 傳回 0，否則傳回 1：

```
$ test 10 -lt 5          (10 < 5 ?)
$ echo $?
1                        (當然不)
$ test -n "hello"  ("hello" 字串的長度不為 0 嗎?)
$ echo $?
0                        (沒錯,長度不是 0)
```

《表十二》列出常見的 test 算符，它們可用於檢查整數、字串、檔案的本質。

test 有一個不尋常的別名：「 [」(左方括號)，以便用於條件判斷與迴圈。當你使用這種寫法時，必須在測試敘述的末端補一個「] 」符號(右方括號)。下列各項測試與前例是完全等效的：

```
$ [ 10 -lt 5 ]
$ echo $?
1
$ [ -n "hello" ]
$ echo $?
0
```

切記，「 [」是一個命令，它和任何其它命令一樣，命令名稱與各引數之間至少要保持一個空格，如果你疏忽了，將發生奇怪的事：

```
$ [ 5 -lt 4]     在「4」和「]」之間沒有空格
bash: [: missing ']'
```

在此例中，test 認為它的最後一個引數是「4]」字串，所以向你抱怨它找不到結尾的「]」符號。

表十二：test 命令的常用算符

檔案測試	
-d *name*	測試 *name* 是否為一個目錄。
-f *name*	測試 *name* 是否為普通檔案。
-L *name*	測試 *name* 是否為象徵連結。
-r *name*	測試 *name* 檔案是否存在，且為可讀。
-w *name*	測試 *name* 檔案是否存在，且為可寫。
-x *name*	測試 *name* 檔案是否存在，且為可執行。

-s *name*	測試 *name* 檔案是否存在，且其長度不為 0。
f1 -nt *f2*	測試 *f1* 是否比 *f2* 更新。
f1 -ot *f2*	測試 *f1* 是否比 *f2* 更舊。
字串測試	
s1 = *s2*	測試兩字串的內容是否完全一樣。
s1 != *s2*	測試兩字串的內容是否有差異。
-z *s1*	測試 *s1* 字串的長度是否為 0。
-n *s1*	測試 *s1* 字串的長度是否不為 0。
整數測試	
a -eq *b*	測試 *a* 與 *b* 是否相等。
a -ne *b*	測試 *a* 與 *b* 是否不相等。
a -gt *b*	測試 *a* 是否大於 *b*。
a -ge *b*	測試 *a* 是否大於等於 *b*。
a -lt *b*	測試 *a* 是否小於 *b*。
a -le *b*	測試 *a* 是否小於等於 *b*。
組合與否定	
t1 -a *t2*	AND（交集）：當 *t1* 與 *t2* 條件都成立時，才算成立。
t1 -o *t2*	OR（聯集）：只要 *t1* 或 *t2* 任一條件成立，就算成立。
!*your_test*	否定測試，當 *your_test* 失敗時，則條件成立。
\(*your_test* \)	改變運算順序（與代數一樣）。

true 與 false

bash 內建兩個與邏輯值有關的命令：true 與 false，它們唯一的作用，是分別傳回 0 與 1 結束狀態：

```
$ true
$ echo $?
0
$ false
$ echo $?
1
```

這兩個命令主要是用於條件測試與迴圈。

40.5. 條件判斷

if 敘述依據條件測試的結果選擇執行路徑。「條件」可能是簡單或複雜的命令。最簡單的 if 敘述形式是 if-then：

```
if command        若 command 的結束狀態為 0
then
  body            條件成立時
fi
```

範例：

```
if [ `whoami` = "root" ]
then
  echo "You are the superuser"
fi
```

另一種形式是 if-then-else 敘述：

```
if command
then
  body1           條件成立時
else
  body2           條件失敗時
fi
```

範例：

```
if [ `whoami` = "root" ]
then
  echo "You are the superuser"
else
  echo "You are an ordinary dude"
fi
```

最複雜的形式是 if-then-elif-else，這可讓你測試許多條件：

```
if command1
then
  body1           當 command1 成立時
elif command2
then
  body2           當 command2 成立時
elif ...
...
else
```

```
  bodyN          當所有條件都不成立時
fi
```

範例：

```
if [ `whoami` = "root" ]
then
  echo "You are the superuser"
elif [ "$USER" = "root" ]
then
  echo "You might be the superuser"
elif [ "$bribe" -gt 10000 ]
then
  echo "You can pay to be the superuser"
else
  echo "You are still an ordinary dude"
fi
```

當需要測試的條件太多，但是受測對象都一樣時，`if-then-elif-else` 就顯得囉唆，這時候 `case` 敘述是比較好的選擇，它依據單一測試條件的結果，選擇最適當的執行途徑：

```
echo 'What would you like to do?'
read answer
case "$answer" in       (受測對象為 answer 變數)
eat)
  echo "OK, have a hamburger"
  ;;
sleep)
  echo "Good night then"
  ;;
*)
  echo "I'm not sure what you want to do"
  echo "I guess I'll see you tomorrow"
  ;;
esac
```

`case` 敘述的標準語法是：

```
case string in
expr1)
  body1
  ;;
expr2)
  body2
```

```
  ;;
  ...
exprN)
  bodyN
  ;;
*)
  bodyelse
  ;;
esac
```

其中的 *string* 可以是任何值，但通常是類似 `$myvar` 這樣變數值；*expr1*、*expr ...* 到 *exprN* 是測試結果的樣式（細節請參閱 `info bash reserved case`），而最後的「`*`」代表前述樣式都不對的情況，它相當於 `if` 敘述中最後的 `else`。每一組命令集合都必須以 `;;` 收尾，就像下面這樣：

```
case $letter in
X)
  echo "$letter is an X"
  ;;
[aeiou])
  echo "$letter is a vowel"
  ;;
[0-9])
  echo "$letter is a digit, silly"
  ;;
*)
echo "I cannot handle  that"
;;
esac
```

40.6. 迴圈

`while` 迴圈由一個測試條件與一組命令構成，只要測試條件持續成立，就重覆執行迴圈內的命令。

```
while command   當 command 的結束狀態為 0 時
do
   body
done
```

舉例來說，若 `myscript` script 的內容是：

```
i=0
while [ $i -lt 3 ]
do
  echo "again"
  i=`expr $i + 1`
done
```

執行結果：

```
$ ./myscript
0
1
2
```

通常，while 迴圈的主體，應該包含能夠改變受測條件的命令，否則會造成無窮迴圈。

until 迴圈的構成也是一個測試條件與一組命令，但是它與 while 迴圈相反，until 迴圈會重複執行那一組命令，直到測試條件成立為止：

```
until command   當 command 的結束狀態不是 0
do
  body
done
```

範例：

```
i=0
until [ $i -gt 3 ]
do
  echo "again"
  i=`expr $i + 1`
done
```

執行結果：

```
$ ./myscript
0
1
2
```

for 迴圈由一個變數、一組資料（變數值）、一組命令構成，資料值會被依序代入變數，然後執行一次迴圈主體，直到所有資料值都被處理過為止。

```
for variable in list
do
  body
done
```

範例：

```
for name in Tom Jack Harry
do
  echo "$name is my friend"
done
```

執行結果：

```
$ ./myscript
Tom is my friend
Jack is my friend
Harry is my friend
```

for 迴圈特別適合用於處理一系列檔案，例如，目前工作目錄下的特定類型檔案：

```
for file in *.doc
do
  echo "$file is a stinky Microsoft Word file"
done
```

某些情況下，你或許會需要無窮迴圈。while 與 until 都有無窮迴圈的效果，你只要提供一個永遠成立（或永遠不成立）的條件即可：

```
while true
do
  echo "forever"
done

until false
do
  echo "forever again"
done
```

通常，你會想在無窮迴圈內放一個測試條件，並以 break 或 exit 來結束迴圈。真正的“無窮”迴圈其實很少見。

40.7. break 與 continue

break 跳出它所在的最內層迴圈。假設有一個 myscript：

```
for name in Tom Jack Harry
do
  echo $name
  echo "again"
done
echo "all done"
```

它的執行結果原本是：

```
$ ./myscript
Tom
again
Jack
again
Harry
again
all done
```

現在加上 break：

```
for name in Tom Jack Harry
do
  echo $name
  if [ "$name" = "Jack" ]
  then
    break
  fi
  echo "again"
done
echo "all done"
```

看看會發生什麼事：

```
$ ./myscript
Tom
again
Jack    發生了 break
all done
```

continue 迫使迴圈立刻跳過本回合未完成的部份，直接進入下一回合。同樣以先前的 myscript 為例：

```
for name in Tom Jack Harry
```

```
do
  echo $name
  if [ "$name" = "Jack" ]
  then
    continue
  fi
  echo "again"
done
echo "all done"
```

看看會怎樣：

```
$ ./myscript
Tom
again
Jack    發生 continue
Harry
again
all done
```

break 和 continue 都可以接受一個數值引數（break *N*、continue *N*），對於 break，*N* 代表要跳出多少層迴圈，對於 continue，則代表要略過多少回合。不過，實務上很少這樣做，因為那會導致你的 script 成為一坨義大利麵，所以我們也建議你最好盡量避免。

40.8. Shell Scripts 的製作與執行

Shell script 本質上只是普通文字檔，凡是可以在 bash 提示符號後鍵入的命令，都可以出現在 script 檔案裡。要執行 script 檔，你有三種選擇：

標準方法

將下列加到 script 檔的頂端（第一列，靠左對齊）：

```
#!/bin/bash
```

然後改變檔案存取模式，使其成為可執行檔：

```
$ chmod +x myscript
```

為了方便，你可將寫好的 script 放在搜尋路徑（非必要步驟）；習慣上，個人寫的 script 是放在 ~/bin 目錄下，若

要給其他使用者也可以用，則是放在 /usr/local/bin 目錄下。放在搜尋路徑中的 script，可以當成普通命令來執行：

```
$ myscript
```

若 script 不是放在搜尋路徑中，而是位於工作目錄下，而且搜尋路徑中也沒包含「.」（工作目錄）【註】，則必須在 script 名稱之前加上「**./**」，shell 才能找到你的 script：

```
$ ./myscript
```

以 subshell 執行

bash 會將它的引數視為 script 檔的名稱，並予以執行：

```
$ bash myscript
```

請注意，由於 script 是在 subshell 的環境裡執行，所以，script 對於環境所做的任何改變（設定 shell 變數、改變工作目錄 ... ），僅止於 subshell，而不影響 login shell。

以 login shell 執行

對於會影響 shell 環境的 script，應該交給目前的 shell 去執行：

```
$ . myscript
```

應該採用哪種方法，取決於 script 本身的性質。一般而言，工具性的 script，應該用 #!/bin/bash 咒語保護好。為了應付臨時工作而寫的一次性 script，那就看需不需要影響 shell 環境來決定。

40.9. 命令列引數

Shell scripts 也都能夠接受命令列引數，就像其它 Linux 命令一樣（事實上，有許多 Linux 命令本身其實是 scripts.）。bash shell 提供了一系列特殊變數，讓你能夠在 script 裡處理引數。

註　將工作目錄納入搜尋路徑，確實很方便，但是基於安全顧慮，Fedora 和許多 Linux distribution 都沒這麼做。

首先，含有所有引數的特殊變數是 **$@**，而 $1、$2、$3 ... 則代表個別引數：

```
$ cat myscript
#!/bin/bash
echo "My name is $1 and I come from $2"
echo "Your info : $@"

$ ./myscript Johnson Wisconsin
My name is Johnson and I come from Wisconsin
Your info : Johnson Wisconsin
```

很顯然，script 無法預先知道使用者會給幾個引數，為此，bash 提供另一個特殊變數 — **$#** — 來代表引數個數：

```
if [ $# -lt 2 ]
then
  echo "$0 error: you must supply two arguments"
else
  echo "My name is $1 and I come from $2"
fi
```

特殊變數 $0 代表 script 自己的名稱。當 script 需要顯示自己的用法或錯誤訊息時，這個變數就可以派上用場：

```
$ ./myscript Bob
./myscript error: you must supply two args
```

用一個簡單的 for 迴圈搭配 $@ 特殊變數，就可以逐一處理每一個引數，不管實際上到底有多少個：

```
for arg in $@
do
  echo "I found the argument $arg"
done
```

40.10. 傳回結束狀態

exit 命令可用於結束 script，並傳回指定的狀態碼給 shell。傳統上，狀態碼 0 代表成功，1（或任何非零值）代表失敗。若 script 結束之前沒呼叫 exit，則 shell 會自動假設狀態碼為 0。

```
if [ $# -lt 2 ]
then
  echo "Error: you must supply two args"
  exit 1
else
  echo "My name is $1 and I come from $2"
fi
exit 0

$ ./myscript Bob
./myscript error: you must supply two args
$ echo $?
1
```

40.11. 除了 Shell Scripting 之外 ...

Shell scripts 的用途相當廣泛，但畢竟不是萬能。所以，Linux 上還有許多更強的命令稿語言與程式語言。

語言	程式	資訊來源
Perl	perl	http://www.perl.com/
Python	python	http://www.python.org/
C/C++	gcc	http://www.gnu.org/software/gcc/
Java	javac、java【註】	http://java.sun.com/
FORTRAN	g77	http://www.gnu.org/software/fortran/fortran.html
Ada	gnat	http://www.gnu.org/software/gnat/gnat.html

註　必須另外安裝，Fedora 與大多數 Linux distribution 都沒隨附。