SHELL 十三问

来源：www.chinaunix.net

作者：**網中人**

整理：HAWK.Li

原文出处：http://bbs.chinaunix.net/forum/24/20031209/218853.html

目　录

1) 為何叫做 shell ？.....2
2) shell prompt(PS1) 與 Carriage Return(CR) 的關係？.....4
3) 別人 echo、你也 echo ，是問 echo 知多少？.....6
4) " "(雙引號) 與 ' '(單引號)差在哪？.....9
5) var=value？export 前後差在哪？.....14
6) exec 跟 source 差在哪？.....19
7) ( ) 與 { } 差在哪？.....23
8) $(( )) 與 $( ) 還有${ } 差在哪？.....25
9) $@ 與 $* 差在哪？.....30
10) && 與 || 差在哪？.....33
11) > 與 < 差在哪？.....38
12) 你要 if 還是 case 呢？.....47
13) for what? while 與 until 差在哪？.....51

1) 為何叫做 shell ？

在介紹 shell 是甚麼東西之前，不妨讓我們重新檢視使用者與電腦系統的關係：
圖(FIXME)

我們知道電腦的運作不能離開硬體，但使用者卻無法直接對硬體作驅動，
硬體的驅動只能透過一個稱為"作業系統(Operating System)"的軟體來控管，
事實上，我們每天所談的 linux ，嚴格來說只是一個作業系統，我們稱之為"核心(kernel)"。
然而，從使用者的角度來說，使用者也沒辦法直接操作 kernel ，
而是透過 kernel 的"外殼"程式，也就是所謂的 shell ，來與 kernel 溝通。
這也正是 kernel 跟 shell 的形像命名關係。如圖：
圖(FIXME)

從技術角度來說，shell 是一個使用者與系統的互動界面(interface)，
主要是讓使用者透過命令行(command line)來使用系統以完成工作。
因此，shell 的最簡單的定義就是---命令解譯器(Command Interpreter)：
* 將使用者的命令翻譯給核心處理，
* 同時，將核心處理結果翻譯給使用者。

每次當我們完成系統登入(log in)，我們就取得一個互動模式的 shell ，也稱為 login shell 或 primary shell。
若從行程(process)角度來說，我們在 shell 所下達的命令，均是 shell 所產生的子行程。這現像，我們暫可稱之為 fork 。
如果是執行腳本(shell script)的話，腳本中的命令則是由另外一個非互動模式的子 shell (sub shell)來執行的。
也就是 primary shell 產生 sub shell 的行程，sub shell 再產生 script 中所有命令的行程。
(關於行程，我們日後有機會再補充。)

這裡，我們必須知道：kernel 與 shell 是不同的兩套軟體，而且都是可以被替換的：
* 不同的作業系統使用不同的 kernel ，
* 而在同一個 kernel 之上，也可使用不同的 shell 。
在 linux 的預設系統中，通常都可以找到好幾種不同的 shell ，且通常會被列於如下檔案裡：
/etc/shells
不同的 shell 有著不同的功能，且也彼此各異、或說"大同小異"。
常見的 shell 主要分為兩大主流：
sh：
burne shell (sh)
burne again shell (bash)
csh：
c shell (csh)

tc shell (tcsh)
korn shell (ksh)
(FIXME)

大部份的 Linux 系統的預設 shell 都是 bash ，其原因大致如下兩點：
* 自由軟體
* 功能強大
bash 是 gnu project 最成功的產品之一，自推出以來深受廣大 Unix 用戶喜愛，且也逐漸成為不少組織的系統標準。

2) shell prompt(PS1) 與 Carriage Return(CR) 的關係？

當你成功登錄進一個文字界面之後，大部份情形下，
你會在熒幕上看到一個不斷閃爍的方塊或底線(視不同版本而別)，
我們稱之為*遊標*(coursor)。
遊標的作用就是告訴你接下來你從鍵盤輸入的按鍵所插入的位置，
且每輸如一鍵遊標便向右邊移動一個格子，若連續輸入太多的話，則自動接在下一行輸入。

假如你剛完成登錄還沒輸入任何按鍵之前，你所看到的遊標所在位置的同一行的左邊部份，
我們稱之為*提示符號*(prompt)。
提示符號的格式或因不同系統版本而各有不同，在 Linux 上，只需留意最接近遊標的一個可見的提示符號，通常是如下兩者之一：
$：給一般使用者帳號使用
#：給 root (管理員)帳號使用

事實上，shell prompt 的意思很簡單：
* 是 shell 告訴使用者：您現在可以輸入命令行了。
我們可以說，使用者只有在得到 shell prompt 才能打命令行，
而 cursor 是指示鍵盤在命令行所輸入的位置，使用者每輸入一個鍵，cursor 就往後移動一格，
直到碰到命令行讀進 CR(Carriage Return，由 Enter 鍵產生)字符為止。
CR 的意思也很簡單：
* 是使用者告訴 shell：老兄你可以執行我的命令行了。
嚴格來說：
* 所謂的命令行，就是在 shell prompt 與 CR 字符之間所輸入的文字。
(思考：為何我們這裡堅持使用 CR 字符而不說 Enter 鍵呢？答案在後面的學習中揭曉。)

不同的命令可接受的命令行格式或有不同，一般情況下，一個標準的命令行格式為如下所列：
command-name options argument

若從技術細節來看，shell 會依據 IFS(Internal Field Seperator) 將 command line 所輸入的文字給拆解為"字段"(word)。
然後再針對特殊字符(meta)先作處理，最後再重組整行 command line 。
(注意：請務必理解上兩句話的意思，我們日後的學習中會常回到這裡思考。)

其中的 IFS 是 shell 預設使用的欄位分隔符號，可以由一個及多個如下按鍵組成：
* 空白鍵(White Space)
* 表格鍵(Tab)
* 回車鍵(Enter)

系統可接受的命令名稱(command-name)可以從如下途逕獲得：
* 明確路逕所指定的外部命令
* 命令別名(alias)
* 自定功能(function)
* shell 內建命令(built-in)
* $PATH 之下的外部命令

每一個命令行均必需含用命令名稱，這是不能缺少的。

3) 別人 echo、你也 echo ，是問 echo 知多少？

承接上一章所介紹的 command line ，這裡我們用 echo 這個命令加以進一步說明。
溫習---標準的 command line 包含三個部件：
* command_name option argument

echo 是一個非常簡單、直接的 Linux 命令：
* 將 argument 送出至標準輸出(STDOUT)，通常就是在監視器(monitor)上輸出。
(註：stdout 我們日後有機會再解說，或可先參考如下討論：
http://www.chinaunix.net/forum/viewtopic.php?t=191375)

為了更好理解，不如先讓我們先跑一下 echo 命令好了：

```
代码:
$ echo

$
```

你會發現只有一個空白行，然後又回到 shell prompt 上了。
這是因為 echo 在預設上，在顯示完 argument 之後，還會送出一個換行符號(new-line charactor)。
但是上面的 command 並沒任何的 argument ，那結果就只剩一個換行符號了...
若你要取消這個換行符號，可利用 echo 的 -n option ：

```
代码:
$ echo -n
$
```

不妨讓我們回到 command line 的概念上來討論上例的 echo 命令好了：
* command line 只有 command_name(echo) 及 option(-n)，並沒有任何 argument 。
要想看看 echo 的 argument ，那還不簡單！接下來，你可試試如下的輸入：

```
代码:
$ echo first line
first line
$ echo -n first line
first line $
```

於上兩個 echo 命令中，你會發現 argument 的部份顯示在你的熒幕，而換行符號則視 -n option 的有無而別。
很明顯的，第二個 echo 由於換行符號被取消了，接下來的 shell prompt 就接在輸出結果同一行了... ^_^

事實上，echo 除了 -n options 之外，常用選項還有：

-e ：啟用反斜線控制字符的轉換(參考下表)
-E：關閉反斜線控制字符的轉換(預設如此)
-n ：取消行末之換行符號(與 -e 選項下的 \c 字符同意)

關於 echo 命令所支援的反斜線控制字符如下表：
\a：ALERT / BELL (從系統喇叭送出鈴聲)
\b：BACKSPACE ，也就是向左刪除鍵
\c：取消行末之換行符號
\E：ESCAPE，跳脫鍵
\f：FORMFEED，換頁字符
\n：NEWLINE，換行字符
\r：RETURN，回車鍵
\t：TAB，表格跳位鍵
\v：VERTICAL TAB，垂直表格跳位鍵
\n：ASCII 八進位編碼(以 x 開首為十六進位)
\\：反斜線本身
(表格資料來自 O'Reilly 出版社之 Learning the Bash Shell, 2nd Ed.)

或許，我們可以透過實例來了解 echo 的選項及控制字符：

例一：

```
代码:
$ echo -e "a\tb\tc\nd\te\tf"
a       b       c
d       e       f
```

上例運用 \t 來區隔 abc 還有 def ，及用 \n 將 def 換至下一行。

例二：

```
代码:
$ echo -e "\141\011\142\011\143\012\144\011\145\011\146"
a       b       c
d       e       f
```

與例一的結果一樣，只是使用 ASCII 八進位編碼。

例三：

```
代码:
$ echo -e "\x61\x09\x62\x09\x63\x0a\x64\x09\x65\x09\x66"
a       b       c
d       e       f
```

與例二差不多，只是這次換用 ASCII 十六進位編碼。

例四：

```
代码:
$ echo -ne "a\tb\tc\nd\te\bf\a"
a           b           c
d           f $
```

因為 e 字母後面是刪除鍵(\b)，因此輸出結果就沒有 e 了。
在結束時聽到一聲鈴響，那是 \a 的傑作！
由於同時使用了 -n 選項，因此 shell prompt 緊接在第二行之後。
若你不用 -n 的話，那你在 \a 後再加個 \c ，也是同樣的效果。

事實上，在日後的 shell 操作及 shell script 設計上，echo 命令是最常被使用的命令之一。
比方說，用 echo 來檢查變量值：

```
代码:
$ A=B
$ echo $A
B
$ echo $?
0
```

(註：關於變量概念，我們留到下兩章才跟大家說明。)

好了，更多的關於 command line 的格式，以及 echo 命令的選項，
就請您自行多加練習、運用了...

4) " "(雙引號) 與 ' '(單引號)差在哪？

還是回到我們的 command line 來吧...
經過前面兩章的學習，應該很清楚當你在 shell prompt 後面敲打鍵盤、直到按下 Enter 的時候，
你輸入的文字就是 command line 了，然後 shell 才會以行程的方式執行你所交給它的命令。
但是，你又可知道：你在 command line 輸入的每一個文字，對 shell 來說，是有類別之分的呢？

簡單而言(我不敢說這是精確的定議，註一)，command line 的每一個 charactor ，分為如下兩種：
* literal：也就是普通純文字，對 shell 來說沒特殊功能。
* meta：對 shell 來說，具有特定功能的特殊保留字元。
(註一：關於 bash shell 在處理 command line 時的順序說明，
請參考 O'Reilly 出版社之 Learning the Bash Shell, 2nd Edition，第 177 - 180 頁的說明，
尤其是 178 頁的流程圖 Figure 7-1 ...)

Literal 沒甚麼好談的，凡舉 abcd、123456 這些"文字"都是 literal ... (easy？)
但 meta 卻常使我們困惑..... (confused?)
事實上，前兩章我們在 command line 中已碰到兩個機乎每次都會碰到的 meta ：
* IFS：由 <space> 或 <tab> 或 <enter> 三者之一組成(我們常用 space)。
* CR：由 <enter> 產生。
IFS 是用來拆解 command line 的每一個詞(word)用的，因為 shell command line 是按詞來處理的。
而 CR 則是用來結束 command line 用的，這也是為何我們敲 <enter> 命令就會跑的原因。
除了 IFS 與 CR ，常用的 meta 還有：
= ： 設定變量。
$ ： 作變量或運算替換(請不要與 shell prompt 搞混了)。
> ：重導向 stdout。
< ：重導向 stdin。
|：命令管線。
& ：重導向 file descriptor ，或將命令置於背境執行。
( )：將其內的命令置於 nested subshell 執行，或用於運算或命令替換。
{ }：將其內的命令置於 non-named function 中執行，或用在變量替換的界定範圍。
; ：在前一個命令結束時，而忽略其返回值，繼續執行下一個命令。
&& ：在前一個命令結束時，若返回值為 true，繼續執行下一個命令。
|| ：在前一個命令結束時，若返回值為 false，繼續執行下一個命令。
!：執行 history 列表中的命令
....

假如我們需要在 command line 中將這些保留字元的功能關閉的話，就需要 quoting 處理了。
在 bash 中，常用的 quoting 有如下三種方法：
* hard quote：''(單引號)，凡在 hard quote 中的所有 meta 均被關閉。
* soft quote： " "(雙引號)，在 soft quoe 中大部份 meta 都會被關閉，但某些則保留(如 $)。(註二)
* escape ： \(反斜線)，只有緊接在 escape (跳脫字符)之後的單一 meta 才被關閉。
(註二：在 soft quote 中被豁免的具體 meta 清單，我不完全知道，
有待大家補充，或透過實作來發現及理解。)

下面的例子將有助於我們對 quoting 的了解：

代码:
```
$ A=B C     # 空白鍵未被關掉，作為 IFS 處理。
$ C: command not found. (FIXME)
$ echo $A

$ A="B C"     # 空白鍵已被關掉，僅作為空白鍵處理。
$ echo $A
B C
```

在第一次設定 A 變量時，由於空白鍵沒被關閉，command line 將被解讀為：
* A=B 然後碰到<IFS>，再執行 C 命令
在第二次設定 A 變量時，由於空白鍵被置於 soft quote 中，因此被關閉，不再作為 IFS ：
* A=B<space>C
事實上，空白鍵無論在 soft quote 還是在 hard quote 中，均會被關閉。Enter 鍵亦然：

代码:
```
$ A='B
> C
> '
$ echo $A
B
C
```

在上例中，由於 <enter> 被置於 hard quote 當中，因此不再作為 CR 字符來處理。
這裡的 <enter> 單純只是一個斷行符號(new-line)而已，由於 command line 並沒得到 CR 字符，
因此進入第二個 shell prompt (PS2，以 > 符號表示)，command line 並不會結束，直到第三行，我們輸入的 <enter> 並不在 hard quote 裡面，因此並沒被關閉，此時，command line 碰到 CR 字符，於是結束、交給 shell 來處理。

上例的 <enter> 要是被置於 soft quote 中的話，也會同樣被關閉，用 escape 亦可：

```
代码:
    $ A=B\
    > C\
    >
    $ echo $A
    B
    C
```

上例中，第一個 <enter> 跟第二個 <enter> 均被 escape 字符關閉了，因此也不作為 CR 來處理，
但第三個 <enter> 由於沒被跳脫，因此作為 CR 結束 command line 。

至於 soft quote 跟 hard quote 的不同，主要是對於某些 meta 的關閉與否，以 $ 來作說明：

```
代码:
    $ A=B\ C
    $ echo "$A"
    B C
    $ echo '$A'
    $A
```

在第一個 echo 命令行中，$ 被置於 soft quote 中，將不被關閉，因此繼續處理變量替換，
因此 echo 將 A 的變量值輸出到熒幕，也就得到 "B C" 的結果。
在第二個 echo 命令行中，$ 被置於 hard quote 中，則被關閉，因此 $ 只是一個 $ 符號，
並不會用來作變量替換處理，因此結果是 $ 符號後面接一個 A 字母：$A 。

練習與思考：如下結果為何不同？

```
代码:
    $ A=B\ C
    $ echo "'$A'"     # 最外面的是單引號
    "$A"
    $ echo '"$A"'     # 最外面的是雙引號
    'B C'
```

(提示：單引號及雙引號，在 quoting 中均被關?#93;了。)

在 CU 的 shell 版裡，我發現有很多初學者的問題，都與 quoting 理解的有關。
比方說，若我們在 awk 或 sed 的命令參數中調用之前設定的一些變量時，常會問及為何不能的問題。
要解決這些問題，關鍵點就是：
* 區分出 shell meta 與 command meta

前面我們提到的那些 meta ，都是在 command line 中有特殊用途的，
比方說 { } 是將其內一系列 command line 置於不具名的函式中執行(可簡單視為 command block)，
但是，awk 卻需要用 { } 來區分出 awk 的命令區段(BEGIN, MAIN, END)。
若你在 command line 中如此輸入：

```
代码:
$ awk {print $0} 1.txt
```

由於 { } 在 shell 中並沒關閉，那 shell 就將 {print $0} 視為 command block ，但同時又沒有" ; "符號作命令區隔，因此就出現 awk 的語法錯誤結果。

要解決之，可用 hard quote ：

```
代码:
$ awk '{print $0}' 1.txt
```

上面的 hard quote 應好理解，就是將原本的 {、<space>、$(註三)、} 這幾個 shell meta 關閉，
避免掉在 shell 中遭到處理，而完整的成為 awk 參數中的 command meta 。
(註三：而其中的 $0 是 awk 內建的 field number ，而非 awk 的變量，
awk 自身的變量無需使用 $ 。)
要是理解了 hard quote 的功能，再來理解 soft quote 與 escape 就不難：

```
代码:
awk "{print \$0}" 1.txt
awk \{print\ \$0\} 1.txt
```

然而，若你要改變 awk 的 $0 的 0 值是從另一個 shell 變量讀進呢？
比方說：已有變量 $A 的值是 0 ，那如何在 command line 中解決 awk 的 $$A 呢？
你可以很直接否定掉 hard quoe 的方案：

```
代码:
$ awk '{print $$A}' 1.txt
```

那是因為 $A 的 $ 在 hard quote 中是不能替換變量的。

聰明的讀者(如你!)，經過本章學習，我想，應該可以解釋為何我們可以使用如下操作了吧：

```
代码:
A=0
awk "{print \$$A}" 1.txt
awk \{print\ \$$A\} 1.txt
awk '{print $'$A'}' 1.txt
awk '{print $'"$A"'}' 1.txt        # 注："$A" 包在 soft quote 中
```

或許，你能舉出更多的方案呢.... ^_^

5) var=value？export 前後差在哪？

這次讓我們暫時丟開 command line ，先來了解一下 bash 變量(variable)吧...

所謂的變量，就是就是利用一個特定的"名稱"(name)來存取一段可以變化的"值"(value)。

設定(set)
在 bash 中，你可以用 "=" 來設定或重新定義變量的內容：
name=value
在設定變量的時侯，得遵守如下規則：
* 等號左右兩邊不能使用區隔符號(IFS)，也應避免使用 shell 的保留字元(meta charactor)。
* 變量名稱不能使用 $ 符號。
* 變量名稱的第一個字母不能是數字(number)。
* 變量名稱長度不可超過 256 個字母。
* 變量名稱及變量值之大小寫是有區別的(case sensitive)。

如下是一些變量設定時常見的錯誤：
A= B ： 不能有 IFS
1A=B ： 不能以數字開頭
$A=B ： 名稱不能有 $
a=B ： 這跟 a=b 是不同的
如下則是可以接受的設定：
A=" B" ： IFS 被關閉了 (請參考前面的 quoting 章節)
A1=B ： 並非以數字開頭
A=$B ： $ 可用在變量值內
This_Is_A_Long_Name=b ： 可用 _ 連接較長的名稱或值，且大小寫有別。

變量替換(substitution)
Shell 之所以強大，其中的一個因素是它可以在命令行中對變量作替換(substitution)處理。
在命令行中使用者可以使用 $ 符號加上變量名稱(除了在用 = 號定義變量名稱之外)，
將變量值給替換出來，然後再重新組建命令行。
比方：

代码:
```
$ A=ls
$ B=la
$ C=/tmp
$ $A -$B $C
```

(注意：以上命令行的第一個 $ 是 shell prompt ，並不在命令行之內。)
必需強調的是，我們所提的變量替換，只發生在 command line 上面。(是的，讓

我們再回到 command line 吧！)
仔細分析最後那行 command line ，不難發現在被執行之前(在輸入 CR 字符之前)，
$ 符號會對每一個變量作替換處理(將變量值替換出來再重組命令行)，最後會得出如下命令行：

```
代码:
    ls -la /tmp
```

還記得第二章我請大家"務必理解"的那兩句嗎？若你忘了，那我這裡再重貼一遍：
引用:
若從技術細節來看，shell 會依據 IFS(Internal Field Seperator) 將 command line 所輸入的文字給拆解為"字段"(word)。
然後再針對特殊字符(meta)先作處理，最後再重組整行 command line 。

這裡的 $ 就是 command line 中最經典的 meta 之一了，就是作變量替換的！
在日常的 shell 操作中，我們常會使用 echo 命令來查看特定變量的值，例如：

```
代码:
    $ echo $A -$B $C
```

我們已學過， echo 命令只單純將其 argument 送至"標準輸出"(STDOUT，通常是我們的熒幕)。
所以上面的命令會在熒幕上得到如下結果：

```
代码:
    ls -la /tmp
```

這是由於 echo 命令在執行時，會先將 $A(ls)、$B(la)、跟 $C(/tmp) 給替換出來的結果。

利用 shell 對變量的替換處理能力，我們在設定變量時就更為靈活了：
A=B
B=$A
這樣，B 的變量值就可繼承 A 變量"當時"的變量值了。
不過，不要以"數學羅輯"來套用變量的設定，比方說：
A=B
B=C
這樣並不會讓 A 的變量值變成 C 。再如：
A=B
B=$A
A=C
同樣也不會讓 B 的值換成 C 。
上面是單純定義了兩個不同名稱的變量：A 與 B ，它們的值分別是 B 與 C 。
若變量被重復定義的話，則原有舊值將被新值所取代。(這不正是"可變的量"嗎？^_^)

當我們在設定變量的時侯，請記著這點：
* 用一個名稱儲存一個數值
僅此而已。

此外，我們也可利用命令行的變量替換能力來"擴充"(append)變量值：
A=B:C:D
A=$A:E
這樣，第一行我們設定 A 的值為 "B:C:D"，然後，第二行再將值擴充為 "A:B:C:E" 。
上面的擴充範例，我們使用區隔符號(:)來達到擴充目的，
要是沒有區隔符號的話，如下是有問題的：
A=BCD
A=$AE
因為第二次是將 A 的值繼承 $AE 的提換結果，而非 $A 再加 E ！
要解決此問題，我們可用更嚴謹的替換處理：
A=BCD
A=${A}E
上例中，我們使用 {} 將變量名稱的範圍給明確定義出來，
如此一來，我們就可以將 A 的變量值從 BCD 給擴充為 BCDE 。

(提示：關於 ${name} 事實上還可做到更多的變量處理能力，這些均屬於比較進階的變量處理，
現階段暫時不介紹了，請大家自行參考資料。如 CU 的貼子：
http://www.chinaunix.net/forum/viewtopic.php?t=201843
)

* export *

嚴格來說，我們在當前 shell 中所定義的變量，均屬於"本地變量"(local variable)，只有經過 export 命令的"輸出"處理，才能成為環境變量(environment variable)：

代码:
```
$ A=B
$ export A
```

或：

代码:
```
$ export A=B
```

經過 export 輸出處理之後，變量 A 就能成為一個環境變量供其後的命令使用。
在使用 export 的時侯，請別忘記 shell 在命令行對變量的"替換"(substitution)處理，
比方說：

代码:
```
$ A=B
```

```
$ B=C
$ export $A
```

上面的命令並未將 A 輸出為環境變量，而是將 B 作輸出，
這是因為在這個命令行中，$A 會首先被提換出 B 然後再"塞回"作 export 的參數。

要理解這個 export ，事實上需要從 process 的角度來理解才能透徹。
我將於下一章為大家說明 process 的觀念，敬請留意。

取消變量

要取消一個變量，在 bash 中可使用 unset 命令來處理：

代码:
```
unset A
```

與 export 一樣，unset 命令行也同樣會作變量替換(這其實就是 shell 的功能之一)，
因此：

代码:
```
$ A=B
$ B=C
$ unset $A
```

事實上所取消的變量是 B 而不是 A 。

此外，變量一旦經過 unset 取消之後，其結果是將整個變量拿掉，而不僅是取消其變量值。
如下兩行其實是很不一樣的：

代码:
```
$ A=
$ unset A
```

第一行只是將變量 A 設定為"空值"(null value)，但第二行則讓變量 A 不在存在。
雖然用眼睛來看，這兩種變量狀態在如下命令結果中都是一樣的：

代码:
```
$ A=
$ echo $A

$ unset A
$ echo $A
```

請學員務必能識別 null value 與 unset 的本質區別,這在一些進階的變量處理上是很嚴格的。
比方說：

代码:

```
$ str=          # 設為 null
$ var=${str=expr}     # 定義 var
$ echo $var

$ echo $str

$ unset str     # 取消
$ var=${str=expr}     # 定義 var
$ echo $var
expr
$ echo $str
expr
```

聰明的讀者(yes, you!)，稍加思考的話，
應該不難發現為何同樣的 var=${str=expr} 在 null 與 unset 之下的不同吧？
若你看不出來，那可能是如下原因之一：
a. 你太笨了
b. 不了解 var=${str=expr} 這個進階處理
c. 對本篇說明還沒來得及消化吸收
e. 我講得不好
不知，你選哪個呢？.... ^_^

6) exec 跟 source 差在哪？

這次先讓我們從 CU Shell 版的一個實例貼子來談起吧：
(http://www.chinaunix.net/forum/viewtopic.php?t=194191)

例中的提問是：
引用:

cd /etc/aa/bb/cc 可以執行
但是把這條命令寫入 shell 時 shell 不執行！
這是什么原因呀！

我當時如何回答暫時別去深究，先讓我們了解一下行程(process)的觀念好了。
首先，我們所執行的任何程式，都是由父行程(parent process)所產生出來的一個子行程(child process)，
子行程在結束後，將返回到父行程去。此一現像在 Linux 系統中被稱為 fork 。
(為何要程為 fork 呢？嗯，畫一下圖或許比較好理解... ^_^)
當子行程被產生的時候，將會從父行程那裡獲得一定的資源分配、及(更重要的是)繼承父行程的環境 ！
讓我們回到上一章所談到的"環境變量"吧：
* 所謂環境變量其實就是那些會傳給子行程的變量。
簡單而言，"遺傳性"就是區分本地變量與環境變量的決定性指標。
然而，從遺傳的角度來看，我們也不難發現環境變量的另一個重要特徵：
* 環境變量只能從父行程到子行程單向繼承。換句話說：在子行程中的環境如何變更，均不會影響父行程的環境。

接下來，再讓我們了解一下命令腳本(shell script)的概念。
所謂的 shell script 講起來很簡單，就是將你平時在 shell prompt 後所輸入的多行 command line 依序寫入一個文件去而已。
其中再加上一些條件判斷、互動界面、參數運用、函數調用、等等技巧，得以讓 script 更加"聰明"的執行，
但若撇開這些技巧不談，我們真的可以簡單的看成 script 只不過依次執行預先寫好的命令行而已。

再結合以上兩個概念(process + script)，那應該就不難理解如下這句話的意思了：
* 正常來說，當我們執行一個 shell script 時，其實是先產生一個 sub-shell 的子行程，然後 sub-shell 再去產生命令行的子行程。
然則，那讓我們回到本章開始時所提到的例子再從新思考：
引用:

cd /etc/aa/bb/cc 可以執行
但是把這條命令寫入 shell 時 shell 不執行！

這是什么原因呀！

我當時的答案是這樣的：
引用:

因為，一般我們跑的 shell script 是用 subshell 去執行的。
從 process 的觀念來看，是 parent process 產生一個 child process 去執行，
當 child 結束後，會返回 parent ，但 parent 的環境是不會因 child 的改變而改變的。
所謂的環境元數很多，凡舉 effective id, variable, workding dir 等等...
其中的 workding dir ($PWD) 正是樓主的疑問所在：
當用 subshell 來跑 script 的話，sub shell 的 $PWD 會因為 cd 而變更，
但當返回 primary shell 時，$PWD 是不會變更的。

能夠了解問題的原因及其原理是很好的，但是？如何解決問題恐怕是我們更感興趣的！是吧？^_^
那好，接下來，再讓我們了解一下 source 命令好了。
當你有了 fork 的概念之後，要理解 source 就不難：
* 所謂 source 就是讓 script 在當前 shell 內執行、而不是產生一個 sub-shell 來執行。
由於所有執行結果均於當前 shell 內完成，若 script 的環境有所改變，當然也會改變當前環境了！
因此，只要我們要將原本單獨輸入的 script 命令行變成 source 命令的參數，就可輕易解決前例提到的問題了。
比方說，原本我們是如此執行 script 的：

代码:
```
./my.script
```

現在改成這樣即可：
代码:
```
source ./my.script
```
或：
```
. ./my.script
```

說到這裡，我想，各位有興趣看看 /etc 底下的眾多設定文件，
應該不難理解它們被定議後，如何讓其他 script 讀取並繼承了吧？
若然，日後你有機會寫自己的 script ，應也不難專門指定一個設定文件以供不同的 script 一起"共用"了... ^_^

okay，到這裡，若你搞得懂 fork 與 source 的不同，那接下來再接受一個挑戰：

---- 那 exec 又與 source/fork 有何不同呢？
哦... 要了解 exec 或許較為複雜，尤其扯上 File Descriptor 的話...
不過，簡單來說：
* exec 也是讓 script 在同一個行程上執行，但是原有行程則被結束了。
也就是簡而言之：原有行程會否終止，就是 exec 與 source/fork 的最大差異了。

嗯，光是從理論去理解，或許沒那麼好消化，不如動手"實作+思考"來的印像深刻哦。
下面讓我們寫兩個簡單的 script ，分別命令為 1.sh 及 2.sh ：

1.sh

代码:

```
#!/bin/bash
A=B
echo "PID for 1.sh before exec/source/fork:$$"
export A
echo "1.sh: \$A is $A"
case $1 in
        exec)
                echo "using exec..."
                exec ./2.sh ;;
        source)
                echo "using source..."
                . ./2.sh ;;
        *)
                echo "using fork by default..."
                ./2.sh ;;
esac
echo "PID for 1.sh after exec/source/fork:$$"
echo "1.sh: \$A is $A"
```

2.sh

代码:

```
#!/bin/bash
echo "PID for 2.sh: $$"
echo "2.sh get \$A=$A from 1.sh"
A=C
export A
echo "2.sh: \$A is $A"
```

然後，分別跑如下參數來觀察結果：

代码:

```
$ ./1.sh fork
$ ./1.sh source
$ ./1.sh exec
```

或是，你也可以參考 CU 上的另一貼子：
http://www.chinaunix.net/forum/viewtopic.php?t=191051

好了，別忘了仔細比較輸出結果的不同及背後的原因哦...
若有疑問，歡迎提出來一起討論討論~~~

happy scripting! ^_^

7) ( ) 與 { } 差在哪？

嗯，這次輕鬆一下，不講太多... ^_^

先說一下，為何要用 ( ) 或 { } 好了。
許多時候，我們在 shell 操作上，需要在一定條件下一次執行多個命令，
也就是說，要麼不執行，要麼就全執行，而不是每次依序的判斷是否要執行下一個命令。
或是，需要從一些命令執行優先次順中得到豁免，如算術的 2*(3+4) 那樣...
這時候，我們就可引入"命令群組"(command group)的概念：將多個命令集中處理。

在 shell command line 中，一般人或許不太計較 ( ) 與 { } 這兩對符號的差異，
雖然兩者都可將多個命令作群組化處理，但若從技術細節上，卻是很不一樣的：
( ) 將 command group 置於 sub-shell 去執行，也稱 nested sub-shell。
{ } 則是在同一個 shell 內完成，也稱為 non-named command group。
若，你對上一章的 fork 與 source 的概念還記得了的話，那就不難理解兩者的差異了。
要是在 command group 中扯上變量及其他環境的修改，我們可以根據不同的需求來使用 ( ) 或 { } 。
通常而言，若所作的修改是臨時的，且不想影響原有或以後的設定，那我們就 nested sub-shell ，
反之，則用 non-named command group 。

是的，光從 command line 來看，( ) 與 { } 的差別就講完了，夠輕鬆吧~~~ ^_^
然而，若這兩個 meta 用在其他 command meta 或領域中(如 Regular Expression)，還是有很多差別的。
只是，我不打算再去說明了，留給讀者自己慢慢發掘好了...
我這裡只想補充一個概念，就是 function 。
所謂的 function ，就是用一個名字去命名一個 command group ，然後再調用這個名字去執行 command group 。
從 non-named command group 來推斷，大概你也可以猜到我要說的是 { } 了吧？(yes! 你真聰明！ ^_^)

在 bash 中，function 的定義方式有兩種：
方式一：

```
代码:
function function_name {
        command1
        command2
        command3
        ....
}
```

方式二：

```
代码:
fuction_name () {
        command1
        command2
        command3
        ....
}
```

用哪一種方式無所謂，只是若碰到所定意的名稱與現有的命令或別名(Alias)衝突的話，方式二或許會失敗。
但方式二起碼可以少打 function 這一串英文字母，對懶人來說(如我)，又何樂不為呢？... ^_^

function 在某一程度來說，也可稱為"函式"，但請不要與傳統編程所使用的函式(library)搞混了，畢竟兩者差異很大。
惟一相同的是，我們都可以隨時用"已定義的名稱"來調用它們...
若我們在 shell 操作中，需要不斷的重覆質行某些命令，我們首先想到的，或許是將命令寫成命令稿(shell script)。
不過，我們也可以寫成 function ，然後在 command line 中打上 function_name 就可當一版的 script 來使用了。
只是若你在 shell 中定義的 function ，除了可用 unset function_name 取消外，一旦退出 shell ，function 也跟著取消。
然而，在 script 中使用 function 卻有許多好處，除了可以提高整體 script 的執行效能外(因為已被載入)，
還可以節省許多重覆的代碼...

簡單而言，若你會將多個命令寫成 script 以供調用的話，那，你可以將 function 看成是 script 中的 script ... ^_^
而且，透過上一章介紹的 source 命令，我們可以自行定義許許多多好用的 function ，再集中寫在特定文件中，
然後，在其他的 script 中用 source 將它們載入並反覆執行。
若你是 RedHat Linux 的使用者，或許，已經猜得出 /etc/rc.d/init.d/functions 這個文件是作啥用的了~~~ ^_^

okay，說要輕鬆點的嘛，那這次就暫時寫到這吧。祝大家學習愉快！ ^_^

8) $(( )) 與 $( ) 還有${ } 差在哪？

我們上一章介紹了 ( ) 與 { } 的不同，這次讓我們擴展一下，看看更多的變化：
$( ) 與 ${ } 又是啥玩意兒呢？

在 bash shell 中，$( ) 與 ` ` (反引號) 都是用來做命令替換用(command substitution)的。
所謂的命令替換與我們第五章學過的變量替換差不多，都是用來重組命令行：
* 完成引號裡的命令行，然後將其結果替換出來，再重組命令行。
例如：

```
代码:
$ echo the last sunday is $(date -d "last sunday" +%Y-%m-%d)
```

如此便可方便得到上一星期天的日期了... ^_^

在操作上，用 $( ) 或 ` ` 都無所謂，只是我"個人"比較喜歡用 $( ) ，理由是：

1, ` ` 很容易與 ' ' (單引號)搞混亂，尤其對初學者來說。
有時在一些奇怪的字形顯示中，兩種符號是一模一樣的(直豎兩點)。
當然了，有經驗的朋友還是一眼就能分變兩者。只是，若能更好的避免混亂，又何樂不為呢？ ^_^

2, 在多層次的復合替換中，` ` 須要額外的跳脫(\`)處理，而 $( ) 則比較直觀。
例如：
這是錯的：

```
代码:
command1 `command2 `command3` `
```

原本的意圖是要在 command2 `command3` 先將 command3 提換出來給 command 2 處理，
然後再將結果傳給 command1 `command2 ...` 來處理。
然而，真正的結果在命令行中卻是分成了 `command2 ` 與 `` 兩段。
正確的輸入應該如下：

```
代码:
command1 `command2 \`command3\` `
```

要不然，換成 $( ) 就沒問題了：

```
代码:
command1 $(command2 $(command3))
```

只要你喜歡，做多少層的替換都沒問題啦~~~ ^_^

不過，$( ) 並不是沒有斃端的...
首先，`` 基本上可用在全部的 unix shell 中使用，若寫成 shell script ，其移植性比較高。
而 $( ) 並不見的每一種 shell 都能使用，我只能跟你說，若你用 bash2 的話，肯定沒問題... ^_^

接下來，再讓我們看 ${ } 吧... 它其實就是用來作變量替換用的啦。
一般情況下，$var 與 ${var} 並沒有啥不一樣。
但是用 ${ } 會比較精確的界定變量名稱的範圍，比方說：

代码:
```
$ A=B
$ echo $AB
```

原本是打算先將 $A 的結果替換出來，然後再補一個 B 字母於其後，
但在命令行上，真正的結果卻是只會提換變量名稱為 AB 的值出來...
若使用 ${ } 就沒問題了：

代码:
```
$ echo ${A}B
BB
```

不過，假如你只看到 ${ } 只能用來界定變量名稱的話，那你就實在太小看 bash 了！
有興趣的話，你可先參考一下 cu 本版的精華文章：
http://www.chinaunix.net/forum/viewtopic.php?t=201843

為了完整起見，我這裡再用一些例子加以說明 ${ } 的一些特異功能：
假設我們定義了一個變量為：
file=/dir1/dir2/dir3/my.file.txt
我們可以用 ${ } 分別替換獲得不同的值：
${file#*/}：拿掉第一條 / 及其左邊的字串：dir1/dir2/dir3/my.file.txt
${file##*/}：拿掉最後一條 / 及其左邊的字串：my.file.txt
${file#*.}：拿掉第一個 . 及其左邊的字串：file.txt
${file##*.}：拿掉最後一個 . 及其左邊的字串：txt
${file%/*}：拿掉最後條 / 及其右邊的字串：/dir1/dir2/dir3
${file%%/*}：拿掉第一條 / 及其右邊的字串：(空值)
${file%.*}：拿掉最後一個 . 及其右邊的字串：/dir1/dir2/dir3/my.file
${file%%.*}：拿掉第一個 . 及其右邊的字串：/dir1/dir2/dir3/my
記憶的方法為：

是去掉左邊(在鑑盤上 # 在 $ 之左邊)
% 是去掉右邊(在鑑盤上 % 在 $ 之右邊)

單一符號是最小匹配；兩個符號是最大匹配。

${file:0:5}：提取最左邊的 5 個字節：/dir1
${file:5:5}：提取第 5 個字節右邊的連續 5 個字節：/dir2

我們也可以對變量值裡的字串作替換：
${file/dir/path}：將第一個 dir 提換為 path：/path1/dir2/dir3/my.file.txt
${file//dir/path}：將全部 dir 提換為 path：/path1/path2/path3/my.file.txt

利用 ${ } 還可針對不同的變數狀態賦值(沒設定、空值、非空值)：
${file-my.file.txt} ：假如 $file 為空值，則使用 my.file.txt 作默認值。(保留沒設定及非空值)
${file:-my.file.txt} ：假如 $file 沒有設定或為空值，則使用 my.file.txt 作默認值。(保留非空值)
${file+my.file.txt} ：不管 $file 為何值，均使用 my.file.txt 作默認值。 (不保留任何值)
${file:+my.file.txt} ：除非 $file 為空值，否則使用 my.file.txt 作默認值。 (保留空值)
${file=my.file.txt} ：若 $file 沒設定，則使用 my.file.txt 作默認值，同時將 $file 定義為非空值。 (保留空值及非空值)
${file:=my.file.txt} ：若 $file 沒設定或為空值，則使用 my.file.txt 作默認值，同時將 $file 定義為非空值。 (保留非空值)
${file?my.file.txt} ：若 $file 沒設定，則將 my.file.txt 輸出至 STDERR。 (保留空值及非空值))
${file:?my.file.txt} ：若 $file 沒設定或為空值，則將 my.file.txt 輸出至 STDERR。 (保留非空值)

還有哦，${#var} 可計算出變量值的長度：
${#file} 可得到 27 ，因為 /dir1/dir2/dir3/my.file.txt 剛好是 27 個字節...

接下來，再為大家介稍一下 bash 的組數(array)處理方法。
一般而言，A="a b c def" 這樣的變量只是將 $A 替換為一個單一的字串，
但是改為 A=(a b c def) ，則是將 $A 定義為組數...
bash 的組數替換方法可參考如下方法：
${A[@]} 或 ${A[*]} 可得到 a b c def (全部組數)
${A[0]} 可得到 a (第一個組數)，${A[1]} 則為第二個組數...
${#A[@]} 或 ${#A[*]} 可得到 4 (全部組數數量)
${#A[0]} 可得到 1 (即第一個組數(a)的長度)，${A[3]} 可得到 3 (第一個組數(def)的長度)
A[3]=xyz 則是將第 4 個組數重新定義為 xyz ...

諸如此類的....
能夠善用 bash 的 $( ) 與 ${ } 可大大提高及簡化 shell 在變量上的處理能力哦~~~ ^_^

好了，最後為大家介紹 $(( )) 的用途吧：它是用來作整數運算的。
在 bash 中，$(( )) 的整數運算符號大致有這些：
+ - * / ：分別為 "加、減、乘、除"。
% ：餘數運算
& | ^ !：分別為 "AND、OR、XOR、NOT" 運算。

例：

代码:
```
$ a=5; b=7; c=2
$ echo $(( a+b*c ))
19
$ echo $(( (a+b)/c ))
6
$ echo $(( (a*b)%c))
1
```

在 $(( )) 中的變量，可用 $ 符號來替換，也可以不用，如：
$(($a + $b * $c)) 也可得到 19 的結果

此外，$(( )) 還可作不同進位(如二進位、八進位、十六進位)作運算呢，只是，輸出結果皆為十進位而已：
echo $((16#2a)) 結果為 42 (16 進位轉十進位)
以一個實用的例子來看看吧：
假如當前的 umask 是 022 ，那麼新建文件的權限即為：

代码:
```
$ umask 022
$ echo "obase=8;$(( 8#666 & (8#777 ^ 8#$(umask)) ))" | bc
644
```

事實上，單純用 (( )) 也可重定義變量值，或作 testing：
a=5; ((a++)) 可將 $a 重定義為 6
a=5; ((a--)) 則為 a=4
a=5; b=7; ((a < b)) 會得到 0 (true) 的返回值。
常見的用於 (( )) 的測試符號有如下這些：

<：小於
>：大於
<=：小於或等於
>=：大於或等於
==：等於
!=：不等於

不過，使用 (( )) 作整數測試時，請不要跟 [ ] 的整數測試搞混亂了。(更多的測試我將於第十章為大家介紹)

怎樣？好玩吧.. ^_^ okay，這次暫時說這麼多...
上面的介紹，並沒有詳列每一種可用的狀態，更多的，就請讀者參考手冊文件囉...

9) $@ 與 $* 差在哪？

要說 $@ 與 $* 之前，需得先從 shell script 的 positional parameter 談起...
我們都已經知道變量(variable)是如何定義及替換的，這個不用再多講了。
但是，我們還需要知道有些變量是 shell 內定的，且其名稱是我們不能隨意修改的，
其中就有 positional parameter 在內。

在 shell script 中，我們可用 $0, $1, $2, $3 ... 這樣的變量分別提取命令行中的如下部份：

```
代码:
script_name parameter1 parameter2 parameter3 ...
```

我們很容易就能猜出 $0 就是代表 shell script 名稱(路逕)本身，而 $1 就是其後的第一個參數，如此類推....
須得留意的是 IFS 的作用，也就是，若 IFS 被 quoting 處理後，那麼 positional parameter 也會改變。
如下例：

```
代码:
my.sh p1 "p2 p3" p4
```

由於在 p2 與 p3 之間的空白鍵被 soft quote 所關閉了，因此 my.sh 中的 $2 是 "p2 p3" 而 $3 則是 p4 ...

還記得前兩章我們提到 fucntion 時，我不是說過它是 script 中的 script 嗎？ ^_^
是的，function 一樣可以讀取自己的(有別於 script 的) postitional parameter ，惟一例外的是 $0 而已。
舉例而言：假設 my.sh 裡有一個 fucntion 叫 my_fun ，若在 script 中跑 my_fun fp1 fp2 fp3 ，
那麼，function 內的 $0 是 my.sh ，而 $1 則是 fp1 而非 p1 了...

不如寫個簡單的 my.sh script 看看吧：

```
代码:
#!/bin/bash

my_fun() {
      echo '$0 inside function is '$0
      echo '$1 inside function is '$1
      echo '$2 inside function is '$2
}

echo '$0 outside function is '$0
echo '$1 outside function is '$1
```

```
echo '$2 outside function is '$2

my_fun fp1 "fp2 fp3"
```

然後在 command line 中跑一下 script 就知道了：

```
代码:
chmod +x my.sh
./my.sh p1 "p2 p3"
$0 outside function is ./my.sh
$1 outside function is p1
$2 outside function is p2 p3
$0 inside function is ./my.sh
$1 inside function is fp1
$2 inside function is fp2 fp3
```

然而，在使用 positional parameter 的時候，我們要注意一些陷阱哦：
* $10 不是替換第 10 個參數，而是替換第一個參數($1)然後再補一個 0 於其後！
也就是，my.sh one two three four five six seven eigth nine ten 這樣的 command line ，
my.sh 裡的 $10 不是 ten 而是 one0 哦... 小心小心！
要抓到 ten 的話，有兩種方法：

方法一是使用我們上一章介紹的 ${ } ，也就是用 ${10} 即可。

方法二，就是 shift 了。
用通俗的說法來說，所謂的 shift 就是取消 positional parameter 中最左邊的參數($0 不受影響)。
其預設值為 1 ，也就是 shift 或 shift 1 都是取消 $1 ，而原本的 $2 則變成 $1、$3 變成 $2 ...
若 shift 3 則是取消前面三個參數，也就是原本的 $4 將變成 $1 ...
那，親愛的讀者，你說要 shift 掉多少個參數，才可用 $1 取得 ${10} 呢？ ^_^

okay，當我們對 positional parameter 有了基本概念之後，那再讓我們看看其他相關變量吧。
首先是 $# ：它可抓出 positional parameter 的數量。
以前面的 my.sh p1 "p2 p3" 為例：
由於 p2 與 p3 之間的 IFS 是在 soft quote 中，因此 $# 可得到 2 的值。
但如果 p2 與 p3 沒有置於 quoting 中話，那 $# 就可得到 3 的值了。
同樣的道理在 function 中也是一樣的...

因此，我們常在 shell script 裡用如下方法測試 script 是否有讀進參數：

```
代码:
```

```
[ $# = 0 ]
```

假如為 0 ，那就表示 script 沒有參數，否則就是有帶參數...

接下來就是 $@ 與 $* ：
精確來講，兩者只有在 soft quote 中才有差異，否則，都表示"全部參數"($0 除外)。
舉例來說好了：
若在 command line 上跑 my.sh p1 "p2 p3" p4 的話，
不管是 $@ 還是 $* ，都可得到 p1 p2 p3 p4 就是了。
但是，如果置於 soft quote 中的話：
"$@" 則可得到 "p1" "p2 p3" "p4" 這三個不同的詞段(word)；
"$*" 則可得到 "p1 p2 p3 p4" 這一整串單一的詞段。

我們可修改一下前面的 my.sh ，使之內容如下：

```
代码:
#!/bin/bash

my_fun() {
      echo "$#"
}

echo 'the number of parameter in "$@" is '$(my_fun "$@")
echo 'the number of parameter in "$*" is '$(my_fun "$*")
```

然後再執行 ./my.sh p1 "p2 p3" p4 就知道 $@ 與 $* 差在哪了 ... ^_^

10) && 與 || 差在哪？

好不容易，進入兩位數的章節了... 一路走來，很辛苦吧？也很快樂吧？ ^_^

在解答本章題目之前，先讓我們了解一個概念：return value ！
我們在 shell 下跑的每一個 command 或 function ，在結束的時候都會傳回父行程一個值，稱為 return value 。
在 shell command line 中可用 $? 這個變量得到最"新"的一個 return value ，也就是剛結束的那個行程傳回的值。
Return Value(RV) 的取值為 0-255 之間，由程式(或 script)的作者自行定議：
* 若在 script 裡，用 exit RV 來指定其值，若沒指定，在結束時以最後一道命令之 RV 為值。
* 若在 function 裡，則用 return RV 來代替 exit RV 即可。

Return Value 的作用，是用來判斷行程的退出狀態(exit status)，只有兩種：
* 0 的話為"真"(true)
* 非 0 的話為"假"(false)

舉個例子來說明好了：
假設當前目錄內有一份 my.file 的文件，而 no.file 是不存在的：

```
代码:
$ touch my.file
$ ls my.file
$ echo $?     # first echo
0
$ ls no.file
ls: no.file: No such file or directory
$ echo $?     # second echo
1
$ echo $?     # third echo
0
```

上例的第一個 echo 是關於 ls my.file 的 RV ，可得到 0 的值，因此為 true ；
第二個 echo 是關於 ls no.file 的 RV ，則得到非 0 的值，因此為 false ；
第三個 echo 是關於第二個 echo $? 的 RV ，為 0 的值，因此也為 true 。

請記住：每一個 command 在結束時都會送回 return value 的！不管你跑甚麼樣的命令...
然而，有一個命令卻是"專門"用來測試某一條件而送出 return value 以供 true 或 false 的判斷，
它就是 test 命令了！
若你用的是 bash ，請在 command line 下打 man test 或 man bash 來了解這個 test 的用法。
這是你可用作參考的最精確的文件了，要是聽別人說的，僅作參考就好...

下面我只簡單作一些輔助說明，其餘的一律以 man 為準：

首先，test 的表示式我們稱為 expression ，其命令格式有兩種：

```
代码:
test expression
or:
[ expression ]
```

(請務必注意 [ ] 之間的空白鍵！)
用哪一種格式沒所謂，都是一樣的效果。(我個人比較喜歡後者...)

其次，bash 的 test 目前支援的測試對像只有三種：
* string：字串，也就是純文字。
* integer：整數(0 或正整數，不含負數或小數點)。
* file：文件。
請初學者一定要搞清楚這三者的差異，因為 test 所用的 expression 是不一樣的。
以 A=123 這個變量為例：
* ["$A" = 123]：是字串的測試，以測試 $A 是否為 1、2、3 這三個連續的"文字"。
* ["$A" -eq 123]：是整數的測試，以測試 $A 是否等於"一百二十三"。
* [-e "$A"]：是關於文件的測試，以測試 123 這份"文件"是否存在。

第三，當 expression 測試為"真"時，test 就送回 0 (true) 的 return value ，否則送出非 0 (false)。
若在 expression 之前加上一個 "！"(感嘆號)，則是當 expression 為"假時" 才送出 0 ，否則送出非 0 。
同時，test 也允許多重的覆合測試：
* expression1 -a expression2 ： 當兩個 exrepssion 都為 true ，才送出 0 ，否則送出非 0 。
* expression1 -o expression2 ： 只需其中一個 exrepssion 為 true ，就送出 0 ，只有兩者都為 false 才送出非 0 。
例如：

```
代码:
[ -d "$file" -a -x "$file" ]
```

是表示當 $file 是一個目錄、且同時具有 x 權限時，test 才會為 true 。

第四，在 command line 中使用 test 時，請別忘記命令行的"重組"特性，
也就是在碰到 meta 時會先處理 meta 再重新組建命令行。(這個特性我在第二及第四章都曾反覆強調過)
比方說，若 test 碰到變量或命令替換時，若不能滿足 expression 格式時，將會得到語法錯誤的結果。
舉例來說好了：

關於 [string1 = string2] 這個 test 格式，
在 = 號兩邊必須要有字串，其中包括空(null)字串(可用 soft quote 或 hard quote 取得)。
假如 $A 目前沒有定義，或被定議為空字串的話，那如下的寫法將會失敗：

```
代码:
$ unset A
$ [ $A = abc ]
[: =: unary operator expected
```

這是因為命令行碰到 $ 這個 meta 時，會替換 $A 的值，然後再重組命令行，那就變成了：
[= abc]
如此一來 = 號左邊就沒有字串存在了，因此造成 test 的語法錯誤！
但是，下面這個寫法則是成立的：

```
代码:
$ [ "$A" = abc ]
$ echo $?
1
```

這是因為在命令行重組後的結果為：
["" = abc]
由於 = 左邊我們用 soft quote 得到一個空字串，而讓 test 語法得以通過...

讀者諸君請務必留意這些細節哦，因為稍一不慎，將會導至 test 的結果變了個樣！
若您對 test 還不是很有經驗的話，那在使用 test 時不妨先採用如下這一個"法則"：
* 假如在 test 中碰到變量替換，用 soft quote 是最保險的！
若你對 quoting 不熟的話，請重新溫習第四章的內容吧... ^_^

okay，關於更多的 test 用法，老話一句：請看 man page 吧！ ^_^

雖然洋洋灑灑講了一大堆，或許你還在嘀咕.... 那... 那個 return value 有啥用啊？ ！
問得好！
告訴你：return value 的作用可大了！若你想讓你的 shell 變"聰明"的話，就全靠它了：
* 有了 return value，我們可以讓 shell 跟據不同的狀態做不同的時情...

這時候，才讓我來揭曉本章的答案吧~~~ ^_^
&& 與 || 都是用來"組建"多個 command line 用的：
* command1 && command2 ：其意思是 command2 只有在 RV 為 0 (true) 的條件下執行。
* command1 || command2 ：其意思是 command2 只有在 RV 為非 0 (false) 的條

件下執行。
來，以例子來說好了：

```
代码:
$ A=123
$ [ -n "$A" ] && echo "yes! it's ture."
yes! it's ture.
$ unset A
$ [ -n "$A" ] && echo "yes! it's ture."
$ [ -n "$A" ] || echo "no, it's NOT ture."
no, it's NOT ture.
```

(註：[-n string] 是測試 string 長度大於 0 則為 true 。)
上例的第一個 && 命令行之所以會執行其右邊的 echo 命令，是因為上一個 test 送回了 0 的 RV 值；
但第二次就不會執行，因為為 test 送回非 0 的結果...
同理，|| 右邊的 echo 會被執行，卻正是因為左邊的 test 送回非 0 所引起的。

事實上，我們在同一命令行中，可用多個 && 或 || 來組建呢：

```
代码:
$ A=123
$ [ -n "$A" ] && echo "yes! it's ture." || echo "no, it's NOT ture."
yes! it's ture.
$ unset A
$ [ -n "$A" ] && echo "yes! it's ture." || echo "no, it's NOT ture."
no, it's NOT ture.
```

怎樣，從這一刻開始，你是否覺得我們的 shell 是"很聰明"的呢？ ^_^

好了，最後，佈置一道習題給大家做做看、、、
下面的判斷是：當 $A 被賦與值時，再看是否小於 100 ，否則送出 too big! ：

```
代码:
$ A=123
$ [ -n "$A" ] && [ "$A" -lt 100 ] || echo 'too big!'
too big!
```

若我將 A 取消，照理說，應該不會送文字才對啊(因為第一個條件就不成立了)...

```
代码:
$ unset A
$ [ -n "$A" ] && [ "$A" -lt 100 ] || echo 'too big!'
too big!
```

為何上面的結果也可得到呢？
又，如何解決之呢？

(提示：修改方法很多，其中一種方法可利用第七章介紹過的 command group ...)

快！告我我答案！其餘免談....

11) > 與 < 差在哪？

這次的題目之前我在 CU 的 shell 版已說明過了：
http://bbs.chinaunix.net/forum/24/20031030/191375.html
這次我就不重寫了，將貼子的內容"抄"下來就是了...

11.1
談到 I/O redirection ，不妨先讓我們認識一下 File Descriptor (FD) 。

程式的運算，在大部份情況下都是進行數據(data)的處理，
這些數據從哪讀進？又，送出到哪裡呢？
這就是 file descriptor (FD) 的功用了。

在 shell 程式中，最常使用的 FD 大概有三個，分別為：
0: Standard Input (STDIN)
1: Standard Output (STDOUT)
2: Standard Error Output (STDERR)

在標準情況下，這些 FD 分別跟如下設備(device)關聯：
stdin(0): keyboard
stdout(1): monitor
stderr(2): monitor

我們可以用如下下命令測試一下：

```
代码:
$ mail -s test root
this is a test mail.
please skip.
^d (同時按 crtl 跟 d 鍵)
```

很明顯，mail 程式所讀進的數據，就是從 stdin 也就是 keyboard 讀進的。
不過，不見得每個程式的 stdin 都跟 mail 一樣從 keyboard 讀進，
因為程式作者可以從檔案參數讀進 stdin ，如：

```
代码:
$ cat /etc/passwd
```

但，要是 cat 之後沒有檔案參數則又如何呢？
哦，請您自己玩玩看囉.... ^_^

```
代码:
$ cat
```

(請留意數據輸出到哪裡去了，最後別忘了按 ^d 離開...)

至於 stdout 與 stderr ，嗯... 等我有空再續吧... ^_^
還是，有哪位前輩要來玩接龍呢？

11.2
沿文再續，書接上一回... ^_^

相信，經過上一個練習後，你對 stdin 與 stdout 應該不難理解吧？
然後，讓我們繼續看 stderr 好了。
事實上，stderr 沒甚麼難理解的：說穿了就是"錯誤信息"要往哪邊送而已...
比方說，若讀進的檔案參數是不存在的，那我們在 monitor 上就看到了：

代码:
```
$ ls no.such.file
ls: no.such.file: No such file or directory
```

若，一個命令同時產生 stdout 與 stderr 呢？
那還不簡單，都送到 monitor 來就好了：

代码:
```
$ touch my.file
$ ls my.file no.such.file
ls: no.such.file: No such file or directory
my.file
```

okay，至此，關於 FD 及其名稱、還有相關聯的設備，相信你已經沒問題了吧？
那好，接下來讓我們看看如何改變這些 FD 的預設數據通道，
我們可用 < 來改變讀進的數據通道(stdin)，使之從指定的檔案讀進。
我們可用 > 來改變送出的數據通道(stdout, stderr)，使之輸出到指定的檔案。

比方說：

代码:
```
$ cat < my.file
```

就是從 my.file 讀進數據

代码:
```
$ mail -s test root < /etc/passwd
```

則是從 /etc/passwd 讀進...
這樣一來，stdin 將不再是從 keyboard 讀進，而是從檔案讀進了...
嚴格來說，< 符號之前需要指定一個 FD 的(之間不能有空白)，

但因為 0 是 < 的預設值，因此 < 與 0< 是一樣的！

okay，這個好理解吧？
那，要是用兩個 << 又是啥呢？
這是所謂的 HERE Document ，它可以讓我們輸入一段文本，直到讀到 << 後指定的字串。
比方說：

代码:
```
$ cat <<FINISH
first line here
second line there
third line nowhere
FINISH
```

這樣的話，cat 會讀進 3 行句子，而無需從 keyboard 讀進數據且要等 ^d 結束輸入。

至於 > 又如何呢？
且聽下回分解....

11.3
okay，又到講古時間~~~

當你搞懂了 0< 原來就是改變 stdin 的數據輸入通道之後，相信要理解如下兩個 redirection 就不難了：
* 1>
* 2>
前者是改變 stdout 的數據輸出通道，後者是改變 stderr 的數據輸出通道。
兩者都是將原本要送出到 monitor 的數據轉向輸出到指定檔案去。
由於 1 是 > 的預設值，因此，1> 與 > 是相同的，都是改 stdout 。

用上次的 ls 例子來說明一下好了：

代码:
```
$ ls my.file no.such.file 1>file.out
ls: no.such.file: No such file or directory
```

這樣 monitor 就只剩下 stderr 而已。因為 stdout 給寫進 file.out 去了。

代码:
```
$ ls my.file no.such.file 2>file.err
my.file
```

這樣 monitor 就只剩下 stdout ，因為 stderr 寫進了 file.err 。

```
代码:
$ ls my.file no.such.file 1>file.out 2>file.err
```

這樣 monitor 就啥也沒有，因為 stdout 與 stderr 都給轉到檔案去了...

呵~~~ 看來要理解 > 一點也不難啦！是不？沒騙你吧？ ^_^
不過，有些地方還是要注意一下的。
首先，是 file locking 的問題。比方如下這個例子：

```
代码:
$ ls my.file no.such.file 1>file.both 2>file.both
```

從 file system 的角度來說，單一檔案在單一時間內，只能被單一的 FD 作寫入。
假如 stdout(1) 與 stderr(2) 都同時在寫入 file.both 的話，
則要看它們在寫入時否碰到同時競爭的情形了，基本上是"先搶先贏"的原則。
讓我們用周星馳式的"慢鏡頭"來看一下 stdout 與 stderr 同時寫入 file.out 的情形好了：
* 第 1, 2, 3 秒為 stdout 寫入
* 第 3, 4, 5 秒為 stderr 寫入
那麼，這時候 stderr 的第 3 秒所寫的數據就丟失掉了！
要是我們能控制 stderr 必須等 stdout 寫完再寫，或倒過來，stdout 等 stderr 寫完再寫，那問題就能解決。
但從技術上，較難掌控的，尤其是 FD 在作"長期性"的寫入時...

那，如何解決呢？所謂山不轉路轉、路不轉人轉嘛，
我們可以換一個思維：將 stderr 導進 stdout 或將 stdout 導進 sterr ，而不是大家在搶同一份檔案，不就行了！
bingo！就是這樣啦：
* 2>&1 就是將 stderr 併進 stdout 作輸出
* 1>&2 或 >&2 就是將 stdout 併進 stderr 作輸出
於是，前面的錯誤操作可以改為：

```
代码:
$ ls my.file no.such.file 1>file.both 2>&1
或
$ ls my.file no.such.file 2>file.both >&2
```

這樣，不就皆大歡喜了嗎？ 呵~~~ ^_^

不過，光解決了 locking 的問題還不夠，我們還有其他技巧需要了解的。
故事還沒結束，別走開！廣告後，我們再回來...！

11.4
okay，這次不講 I/O Redirction ，講佛吧...
(有沒搞錯？！網中人是否頭殼燒壞了？...) 嘻~~~ ^_^

學佛的最高境界，就是"四大皆空"。至於是空哪四大塊？我也不知，因為我還沒到那境界...
但這個"空"字，卻非常值得我們返複把玩的：
--- 色即是空、空即是色！
好了，施主要是能夠領會"空"的禪意，那離修成正果不遠矣~~~

在 Linux 檔案系統裡，有個設備檔位於 /dev/null 。
許多人都問過我那是甚麼玩意兒？我跟你說好了：那就是"空"啦！
沒錯！空空如也的空就是 null 了.... 請問施主是否忽然有所頓誤了呢？然則恭喜了~~~ ^_^

這個 null 在 I/O Redirection 中可有用得很呢：
* 若將 FD1 跟 FD2 轉到 /dev/null 去，就可將 stdout 與 stderr 弄不見掉。
* 若將 FD0 接到 /dev/null 來，那就是讀進 nothing 。
比方說，當我們在執行一個程式時，畫面會同時送出 stdout 跟 stderr ，
假如你不想看到 stderr (也不想存到檔案去)，那可以：

代码:
```
$ ls my.file no.such.file 2>/dev/null
my.file
```

若要相反：只想看到 stderr 呢？還不簡單！將 stdout 弄到 null 就行：

代码:
```
$ ls my.file no.such.file >/dev/null
ls: no.such.file: No such file or directory
```

那接下來，假如單純只跑程式，不想看到任何輸出結果呢？
哦，這裡留了一手上次節目沒講的法子，專門贈予有緣人！... ^_^
除了用 >/dev/null 2>&1 之外，你還可以如此：

代码:
```
$ ls my.file no.such.file &>/dev/null
```

(提示：將 &> 換成 >& 也行啦~~!)

okay? 講完佛，接下來，再讓我們看看如下情況：

代码:

```
$ echo "1" > file.out
$ cat file.out
1
$ echo "2" > file.out
$ cat file.out
2
```

看來，我們在重導 stdout 或 stderr 進一份檔案時，似乎永遠只獲得最後一次導入的結果。
那，之前的內容呢？
呵~~~ 要解決這個問提很簡單啦，將 > 換成 >> 就好：

```
代码:
$ echo "3" >> file.out
$ cat file.out
2
3
```

如此一來，被重導的目標檔案之內容並不會失去，而新的內容則一直增加在最後面去。
easy ？ 呵 ... ^_^

但，只要你再一次用回單一的 > 來重導的話，那麼，舊的內容還是會被"洗"掉的！
這時，你要如何避免呢？
----備份！ yes ，我聽到了！不過.... 還有更好的嗎？
既然與施主這麼有緣份，老納就送你一個錦囊妙法吧：

```
代码:
$ set -o noclobber
$ echo "4" > file.out
-bash: file: cannot overwrite existing file
```

那，要如何取消這個"限制"呢？
哦，將 set -o 換成 set +o 就行：

```
代码:
$ set +o noclobber
$ echo "5" > file.out
$ cat file.out
5
```

再問：那... 有辦法不取消而又"臨時"蓋寫目標檔案嗎？
哦，佛曰：不可告也！

啊~~~ 開玩笑的、開玩笑的啦~~~ ^_^ 唉，早就料到人心是不足的了！

代码:
```
$ set -o noclobber
$ echo "6" >| file.out
$ cat file.out
6
```

留意到沒有：在 > 後面再加個" | "就好(注意： > 與 | 之間不能有空白哦)....

呼.... (深呼吸吐納一下吧)~~~ ^_^
再來還有一個難題要你去參透的呢：

代码:
```
$ echo "some text here" > file
$ cat < file
some text here
$ cat < file > file.bak
$ cat < file.bak
some text here
$ cat < file > file
$ cat < file
```

嗯？！注意到沒有？！！
---- 怎麼最後那個 cat 命令看到的 file 竟是空的？！
why? why? why?

同學們：下節課不要遲到囉~~~!

11.5
噹噹噹~~~ 上課囉~~~ ^_^

前面提到：$ cat < file > file 之後原本有內容的檔案結果卻被洗掉了！
要理解這一現像其實不難，這只是 priority 的問題而已：
* 在 IO Redirection 中，stdout 與 stderr 的管道會先準備好，才會從 stdin 讀進資料。
也就是說，在上例中，> file 會先將 file 清空，然後才讀進 < file ，
但這時候檔案已經被清空了，因此就變成讀不進任何資料了...

哦~~~ 原來如此~~~~ ^_^
那... 如下兩例又如何呢？

代码:
```
$ cat <> file
```

```
$ cat < file >> file
```

嗯... 同學們，這兩個答案就當練習題囉，下節課之前請交作業！

好了，I/O Redirection 也快講完了，sorry，因為我也只知道這麼多而已啦~~~ 嘻~~ ^_^
不過，還有一樣東東是一定要講的，各位觀眾(請自行配樂~!#@!$%) ：
---- 就是 pipe line 也！

談到 pipe line ，我相信不少人都不會陌生：
我們在很多 command line 上常看到的" | "符號就是 pipe line 了。
不過，究竟 pipe line 是甚麼東東呢？
別急別急... 先查一下英漢字典，看看 pipe 是甚麼意思？
沒錯！它就是"水管"的意思...
那麼，你能想像一下水管是怎麼一根接著一根的嗎？
又，每根水管之間的 input 跟 output 又如何呢？
嗯？？
靈光一閃：原來 pipe line 的 I/O 跟水管的 I/O 是一模一樣的：
* 上一個命令的 stdout 接到下一個命令的 stdin 去了！
的確如此... 不管在 command line 上你使用了多少個 pipe line ，
前後兩個 command 的 I/O 都是彼此連接的！(恭喜：你終於開竅了！ ^_^)

不過... 然而... 但是... ... stderr 呢？
好問題！不過也容易理解：
* 若水管漏水怎麼辦？
也就是說：在 pipe line 之間，前一個命令的 stderr 是不會接進下一命令的 stdin 的，
其輸出，若不用 2> 導到 file 去的話，它還是送到監視器上面來！
這點請你在 pipe line 運用上務必要注意的。

那，或許你又會問：
* 有辦法將 stderr 也餵進下一個命令的 stdin 去嗎？
(貪得無厭的家夥！)
方法當然是有，而且你早已學過了！ ^_^
我提示一下就好：
* 請問你如何將 stderr 合併進 stdout 一同輸出呢？
若你答不出來，下課之後再來問我吧... (如果你臉皮真夠厚的話...)

或許，你仍意尤未盡！或許，你曾經碰到過下面的問題：
* 在 cm1 | cm2 | cm3 ... 這段 pipe line 中，若要將 cm2 的結果存到某一檔案呢？

若你寫成 cm1 | cm2 > file | cm3 的話，

那你肯定會發現 cm3 的 stdin 是空的！(當然啦，你都將水管接到別的水池了！)
聰明的你或許會如此解決：

```
代码:
cm1 | cm2 > file ; cm3 < file
```

是的，你的確可以這樣做，但最大的壞處是：這樣一來，file I/O 會變雙倍！
在 command 執行的整個過程中，file I/O 是最常見的最大效能殺手。
凡是有經驗的 shell 操作者，都會盡量避免或降低 file I/O 的頻率。

那，上面問題還有更好方法嗎？
有的，那就是 tee 命令了。
* 所謂 tee 命令是在不影響原本 I/O 的情況下，將 stdout 複製一份到檔案去。
因此，上面的命令行可以如此打：

```
代码:
cm1 | cm2 | tee file | cm3
```

在預設上，tee 會改寫目標檔案，若你要改為增加內容的話，那可用 -a 參數達成。

基本上，pipe line 的應用在 shell 操作上是非常廣泛的，尤其是在 text filtering 方面，
凡舉 cat, more, head, tail, wc, expand, tr, grep, sed, awk, ... 等等文字處理工具，
搭配起 pipe line 來使用，你會驚覺 command line 原來是活得如此精彩的！
常讓人有"眾裡尋他千百度，驀然回首，那人卻在燈火闌珊處！"之感... ^_^

....

好了，關於 I/O Redirection 的介紹就到此告一段落。
若日後有空的話，再為大家介紹其它在 shell 上好玩的東西！bye... ^_^

12) 你要 if 還是 case 呢？

放了一個愉快的春節假期，人也變得懶懶散散的... 只是，答應了大家的作業，還是要堅持完成就是了~~~

還記得我們在第 10 章所介紹的 return value 嗎？
是的，接下來介紹的內容與之有關，若你的記憶也被假期的歡樂時光所抵消掉的話，
那，建議您還是先回去溫習溫習再回來...

若你記得 return value ，我想你也應該記得了 && 與 || 是甚麼意思吧？
用這兩個符號再配搭 command group 的話，我們可讓 shell script 變得更加聰明哦。
比方說：

```
代码:
comd1 && {
      comd2
      comd3
} || {
      comd4
      comd5
}
```

意思是說：

假如 comd1 的 return value 為 true 的話，
然則執行 comd3 與 comd4 ，
否則執行 comd4 與 comd5 。

事實上，我們在寫 shell script 的時候，經常需要用到這樣那樣的條件以作出不同的處理動作。
用 && 與 || 的確可以達成條件執行的效果，然而，從"人類語言"上來理解，卻不是那麼直觀。
更多時候，我們還是喜歡用 if .... then ... else ... 這樣的 keyword 來表達條件執行。
在 bash shell 中，我們可以如此修改上一段代碼：

```
代码:
if comd1
then
      comd2
      comd3
else
      comd4
```

```
    comd5
fi
```

這也是我們在 shell script 中最常用到的 if 判斷式：

只要 if 後面的 command line 返回 true 的 return value (我們最常用 test 命令來送出 return value)，
然則就執行 then 後面的命令，否則執行 else 後的命令；fi 則是用來結束判斷式的 keyword 。

在 if 判斷式中，else 部份可以不用，但 then 是必需的。
(若 then 後不想跑任何 command ，可用" : " 這個 null command 代替)。
當然，then 或 else 後面，也可以再使用更進一層的條件判斷式，這在 shell script 設計上很常見。
若有多項條件需要"依序"進行判斷的話，那我們則可使用 elif 這樣的 keyword ：

```
代码:
if comd1; then
    comd2
elif comd3; then
    comd4
else
    comd5
fi
```

意思是說：

若 comd1 為 true ，然則執行 comd2 ；
否則再測試 comd3 ，然則執行 comd4 ；
倘若 comd1 與 comd3 均不成立，那就執行 comd5 。

if 判斷式的例子很常見，你可從很多 shell script 中看得到，我這裡就不再舉例子了...

接下來要為大家介紹的是 case 判斷式。
雖然 if 判斷式已可應付大部份的條件執行了，然而，在某些場合中，卻不夠靈活，
尤其是在 string 式樣的判斷上，比方如下：

```
代码:
QQ () {
    echo -n "Do you want to continue? (Yes/No): "
    read YN
```

```
    if [ "$YN" = Y -o "$YN" = y -o "$YN" = "Yes" -o "$YN" = "yes" -o "$YN" =
"YES" ]
    then
        QQ
    else
        exit 0
    fi
}
QQ
```

從例中，我們看得出來，最麻煩的部份是在於判斷 YN 的值可能有好幾種式樣。
聰明的你或許會如此修改：

```
代码:
...
if echo "$YN" | grep -q '^[Yy]\([Ee][Ss]\)*$'
...
```

也就是用 Regular Expression 來簡化代碼。(我們有機會再來介紹 RE)
只是... 是否有其它更方便的方法呢？
有的，就是用 case 判斷式即可：

```
代码:
QQ () {
    echo -n "Do you want to continue? (Yes/No): "
    read YN
  case "$YN" in
        [Yy]|[Yy][Ee][Ss])
            QQ
            ;;
        *)
            exit 0
            ;;
    esac
}
QQ
```

我們常 case 的判斷式來判斷某一變量在同的值(通常是 string)時作出不同的處理，
比方說，判斷 script 參數以執行不同的命令。
若你有興趣、且用 Linux 系統的話，不妨挖一挖 /etc/init.d/* 裡那堆 script 中的 case 用法。
如下就是一例：

```
代码:
case "$1" in
  start)
```

```
        start
        ;;
  stop)
        stop
        ;;
  status)
        rhstatus
        ;;
  restart|reload)
        restart
        ;;
  condrestart)
        [ -f /var/lock/subsys/syslog ] && restart || :
        ;;
  *)
        echo $"Usage: $0 {start|stop|status|restart|condrestart}"
        exit 1
esac
```

(若你對 positional parameter 的印像已經模糊了，請重看第 9 章吧。)

okay，十三問還剩一問而已，過幾天再來搞定之.... ^_^

13) for what? while 與 until 差在哪？

終於，來到 shell 十三問的最後一問了... 長長吐一口氣~~~~

最後要介紹的是 shell script 設計中常見的"循環"(loop)。
所謂的 loop 就是 script 中的一段在一定條件下反覆執行的代碼。
bash shell 中常用的 loop 有如下三種：
* for
* while
* until

for loop 是從一個清單列表中讀進變量值，並"依次"的循環執行 do 到 done 之間的命令行。
例：

```
代码:
for var in one two three four five
do
        echo -----------
        echo '$var is '$var
        echo
done
```

上例的執行結果將會是：

1) for 會定義一個叫 var 的變量，其值依次是 one two three four five 。
2) 因為有 5 個變量值，因此 do 與 done 之間的命令行會被循環執行 5 次。
3) 每次循環均用 echo 產生三行句子。
而第二行中不在 hard quote 之內的 $var 會依次被替換為 one two three four five 。
4) 當最後一個變量值處理完畢，循環結束。

我們不難看出，在 for loop 中，變量值的多寡，決定循環的次數。
然而，變量在循環中是否使用則不一定，得視設計需求而定。
倘若 for loop 沒有使用 in 這個 keyword 來指定變量值清單的話，其值將從 $@ (或 $*)中繼承：

```
代码:
for var; do
        ....
done
```

(若你忘記了 positional parameter ，請溫習第 9 章...)

for loop 用於處理"清單"(list)項目非常方便，

其清單除了可明確指定或從 positional parameter 取得之外，
也可從變量替換或命令替換取得... (再一次提醒：別忘了命令行的"重組"特性！)
然而，對於一些"累計變化"的項目(如整數加減)，for 亦能處理：

```
代码:
for ((i=1;i<=10;i++))
do
    echo "num is $i"
done
```

除了 for loop ，上面的例子我們也可改用 while loop 來做到：

```
代码:
num=1
while [ "$num" -le 10 ]; do
      echo "num is $num"
      num=$(($num + 1))
done
```

while loop 的原理與 for loop 稍有不同：
它不是逐次處理清單中的變量值，而是取決於 while 後面的命令行之 return value ：
* 若為 ture ,則執行 do 與 done 之間的命令,然後重新判斷 while 後的 return value 。
* 若為 false ，則不再執行 do 與 done 之間的命令而結束循環。

分析上例：

1) 在 while 之前，定義變量 num=1 。
2) 然後測試(test) $num 是否小於或等於 10 。
3) 結果為 true ，於是執行 echo 並將 num 的值加一。
4) 再作第二輪測試，其時 num 的值為 1+1=2 ，依然小於或等於 10，因此為 true ，繼續循環。
5) 直到 num 為 10+1=11 時，測試才會失敗... 於是結束循環。

我們不難發現：
* 若 while 的測試結果永遠為 true 的話，那循環將一直永久執行下去：

```
代码:
while :; do
      echo looping...
done
```

上例的" : "是 bash 的 null command ，不做任何動作，除了送回 true 的 return value 。
因此這個循環不會結束，稱作死循環。
死循環的產生有可能是故意設計的(如跑 daemon)，也可能是設計錯誤。
若要結束死尋環，可透過 signal 來終止(如按下 ctrl-c)。
(關於 process 與 signal ，等日後有機會再補充，十三問暫時略過。)

一旦你能夠理解 while loop 的話，那，就能理解 until loop ：
* 與 while 相反，until 是在 return value 為 false 時進入循環，否則結束。
因此，前面的例子我們也可以輕鬆的用 until 來寫：

```
代码:
num=1
until [ ! "$num" -le 10 ]; do
        echo "num is $num"
        num=$(($num + 1))
done
```

或是：

```
代码:
num=1
until [ "$num" -gt 10 ]; do
        echo "num is $num"
        num=$(($num + 1))
done
```

okay ，關於 bash 的三個常用的 loop 暫時介紹到這裡。
在結束本章之前，再跟大家補充兩個與 loop 有關的命令：
* break
* continue
這兩個命令常用在複合式循環裡，也就是在 do ... done 之間又有更進一層的 loop ，
當然，用在單一循環中也未嘗不可啦... ^_^

break 是用來打斷循環，也就是"強迫結束" 循環。
若 break 後面指定一個數值 n 的話，則"從裡向外"打斷第 n 個循環，
預設值為 break 1 ，也就是打斷當前的循環。
在使用 break 時需要注意的是， 它與 return 及 exit 是不同的：
* break 是結束 loop
* return 是結束 function
* exit 是結束 script/shell

而 continue 則與 break 相反：強迫進入下一次循環動作。
若你理解不來的話，那你可簡單的看成：在 continue 到 done 之間的句子略過

而返回循環頂端...
與 break 相同的是：continue 後面也可指定一個數值 n ，以決定繼續哪一層(從裡向外計算)的循環，
預設值為 continue 1 ，也就是繼續當前的循環。

在 shell script 設計中，若能善用 loop ，將能大幅度提高 script 在複雜條件下的處理能力。
請多加練習吧....

好了，該是到了結束的時候了。
婆婆媽媽的跟大家囉唆了一堆關於 shell 的基礎概念，
目的不是要告訴大家"答案"，而是要帶給大家"啟發"...
在日後關於 shell 的討論中，我或許會經常用"鏈接"方式指引回來十三問中的內容，
以便我們在進行技術探討時彼此能有一些討論基礎，而不至於各說各話、徒費時力。
但，更希望十三問能帶給你更多的思考與樂趣，至為重要的是透過實作來加深理解。

是的，我很重視"實作"與"獨立思考"這兩項學習要素，若你能夠掌握其中真義，那請容我說聲：
--- 恭喜！十三問你沒白看了！ ^_^

p.s.
至於補充問題部份，我暫時不寫了。而是希望：
1) 大家擴充題目。
2) 一起來寫心得。

Good luck and happy studying!