Script 指令技巧

內容

內建指令 VS 外部指令

內建指令的執行權優先於外部指令，例如 exit 為 Shell 的內建指令，有個外部指令 /usr/bin/exit，當我們想讓 Script 執行/usr/bin/exit指令時，必須在編輯 Script 時寫出絕對路徑 /usr/bin/exit 而不是 exit，若您寫 exit 則系統將會去執行內建指令 exit 而不是外部指令 /usr/bin/exit。

如何判斷指令為內建指令或外部指令？

```
#which exit
exit: shell built-in command.      //顯示 exit 為內建指令
#which mkdir
/bin/mkdir                         //顯示 mkdir 為外部指令
```

內建指令 VS 函數名稱

定義的 Shell 函數名稱執行優先權大為內建指令，因此時想執行內建指令時該怎辦？ 此時只要配合 Command 即可順利執行內建指令

```
command 內建指令
```

所以我們可以知道 Shell Script 指令執行的優先權排序為

```
函數名稱 > 內建指令 > 外部指令
```

執行指令的方式

想讓 Shell Script 去執行你想要執行指令的時候，有如下三種方式 (以指令 uname 舉例)

```
uname            //直接使用指令 (環境變數需設定)
uname -a         //指令加上參數 (環境變數需設定)
/usr/bin/uname   //使用絕對路徑來執行指令
```

單引號、雙引號、反斜線

當 Shell Script 在執行指令若想輸出空白字元或特殊符號時，便需要雙引號、單引號、反斜線來輸出，利用下列範例來看比較容易了解

```
#!/bin/sh
echo "Good     Morning"
echo 'Good \ \ \ \ \ Morning'
echo  Good     Morning
echo "<TAB>"
echo '<TAB>'
echo \<TAB\>
echo <TAB>
```

執行結果

```
Good     Morning               //視為字串
Good     Morning               //視為字串
Good Morning                   //第二個空白字元系統會自動忽略掉
<TAB>                          //視為字串
<TAB>                          //視為字串
<TAB>                          //反斜線後會視為字串
012.sh: 9: Syntax error: newline unexpected  //系統會看成 <TAB 及 > ，因為找不到輸出所以就
```

單、雙引號差別在哪？ 哪時該使用單引號又哪時該使用雙引號？ 答案在於會不會把**變數展開**或**視為字串**

```
#!/bin/sh
number="Hello   World!!"
echo "$number"
echo '$number'
echo $number
```

執行結果

```
Hello   World!!     //雙引號會把變數展開並保留空白字元
$number             //單引號會把變數視為字串
Hello World!!       //直接呼叫會將變數展開但空白字元將視為間隔字元
```

查看指令是否執行成功【$?】

指令在執行之後會把執行結果存到變數 $? 內，而指令執行成功則會把 0 寫入到變數 $? 內，而失敗則會存入 1 ~ 255 寫入到變數 $?，不同的失敗數字代表相對應的失敗原因，若有多個【|】pipe 執行時則將是最後的指令執行結果寫入變數 $? 內。參考失敗數字及原因對應

例如在家目錄下有個檔案叫 1.sh 而沒有 2.sh 看下列執行結果即可知

```
# cat 2.sh | wc -l ; echo $?
cat: 2.sh: No such file or directory   //找不到 2.sh 檔案
```

```
      0                                  //沒有行數
1                                        //指令執行失敗數字
# cat 1.sh | wc -l ; echo $?
      3                                  //1.sh內有三行指令
0                                        //指令執行成功
```

把指令丟到背景執行【&】

在指令後面加上【&】即可把指令丟到背景去執行，但會隨著下指令的使用者登出之後便結束指令的執行，若希望在登出後指令仍能繼續執行請使用下面介紹的【nohup】。

一行執行多個指令【;】

原則上一行只能執行一行指令，若想有同一行執行多個指令的話，便在指令與指令之間加上分號【;】，便可一行執行多個指令

當指令執行成功才執行下一個指令【&&】

當執行的指令利用分號來間隔時系統無論如何 (即不管指令執行是否成功) 都會執行，與分號不同的是若以【&&】來間隔指令則只有當前面的指令執行成功時會才繼續執行下一個指令

例如在家目錄下有二個檔案 1.sh 及 2.sh

```
#rm 2.sh && touch 3.sh && ls      //刪除 2.sh 後建立 3.sh 再執行 ls
1.sh   3.sh
#rm 2.sh && touch 3.sh && ls      //由於 2.sh 已經不存在了所以指令執行不成功後面的指令便不再
rm: 2.sh: No such file or directory
```

當指令執行失敗才執行下一個指令【||】

若以【||】來間隔指令則只有當前面的指令執行失敗時會才繼續執行下一個指令

例如在家目錄下有個檔案 1.sh

```
#rm 2.sh || touch 3.sh ; ls       //刪除 2.sh 但不存在指令執行失敗，所以執行建立 3.sh 檔案再
rm: 2.sh: No such file or directory
1.sh   3.sh
#rm 3.sh || touch 2.sh ; ls       //刪除 3.sh 指令執行成功，不執行建立 2.sh，再執行ls
1.sh
```

使指令以 Daemon 方式在背景執行【nohup】

正常情形下，C Shell 在系統中所執行的處理程序 Process 會隨著 Logout 而終止，這是因為 UNIX 系統會隨著 Logout 送出 Hangup 的訊號將你所有的 Process 終止，例如家目下 time.sh 內容如下

```
#!/bin/sh
while true;do                                  //do~done之間為迴圈部份
     date '+%H:%M:%S' > "$HOME/nowtime"        //迴圈部份
     sleep 1                                   //迴圈部份
done
```

使用 nohup command & 將指令用 Daemon 方式在背景執行

```
#nohup ~/time.sh &  //以 Daemon 方式在背景執行
[1] 81957           //顯示啟動程序 ID
```

可登出並再次登入後以 cat 方式查看 nowtime 可發現 time.sh 仍在執行中，可利用 kill 終結執行序。

補充：使用 nohup command & 仍無法把指令丟到背景執行？

狀況是這樣的主機 A 控制遠端主機 B 去執行 .sh，雖然遠端主機 B 已經使用 nohup command & 的方式將 Script 丟到背景執行，且在遠端主機 B 是可正常運作的，但當主機 A 去控制遠端主機 B 執行 Script 時雖然遠端主機 B 可以把 Script 丟到背景執行但主機 A 這裡的 Script 卻回不來，所以我們可以用個簡單的判斷把主機 A 這裡等待遠端主機回應的 Process 砍掉 (當然要確定遠端機主 B 有確實執行該 Script).，以下就是在主機 A 中的內容

```
echo 'Starting Logging Server on Host B...'          //顯示字串
ssh hostb.weithenn.org 'sudo /home/logging.sh' &     //控制遠端主機 B 執行 logging.sh，
pid_of_q="$!"                                        //抓取上一行丟到背景執行的 Process
sleep 5                                              //休息 5 秒 (不然下一行會砍不掉)
kill -9 ${pid_of_q}                                  //強制砍掉剛才丟到背景執行的 Proce
```

把變數內容當成指令【eval】

可以把變數內容當成指令來執行例如

```
#!/bin/sh
D="date"
eval "$D"      //輸出指令 2006年 4月20日 周四 17時14分16秒 CST
echo "$D"      //輸出字串 date
```

把指令執行結果變成下一個指令的輸入資料【|】

pipe 來把指令 1 的執行結果變成指令 2 的輸入資料，例如下例為查看開機訊息並尋找關鍵字有 ad0 的行，不過要注意的是若在 pipe 內設定的變數的值在執行後會消失了。

```
#!/bin/sh
cat /var/run/dmesg.boot | grep 'ad0'   //輸出 ad0: 38172MB <MAXTOR 6L040J2 A93.0500> at
```

把指令執行結果寫入檔案內【> 重新寫入、 >> 最尾寫入】

使用 > 則是把執行結果重新寫入檔案內(先清空內容在寫入)，而 >> 則是把執行結果從檔案最後面開始寫入

```
#!/bin/sh
echo "test1" > 1.txt     //此時 1.txt 內容為 test1
echo "test2" > 1.txt     //此時 1.txt 內容為 test2
echo "test3" >> 1.txt    //此時 1.txt 內容為 test2 test3
```

而 tee 則是包含了 > 功能並同時把執行結果顯示到畫面上，所以就寫入檔案功能上 tee 等於 > ，而 tee -a 等於 >> 所以上面的例子可以改寫為

```
#!/bin/sh
echo "test1" | tee  1.txt
echo "test2" | tee  1.txt
echo "test3" | tee -a 1.txt
```

把檔案變成指令執行的依據【<】

上面的技巧是把指令執行的結果寫入到指定的檔案內，而反過來思考我們可以把檔案變成是指令執行的依據，如下變成從檔案 1.txt 去找 test2 是否存在

```
#!/bin/sh
grep 'test2' < 1.txt    //顯示 test2
```

標準輸入輸出【/dev/fd/number】

什麼是標準輸入、標準輸出、標準錯誤輸出？有什麼差別？

Handle	Show	Read / Write
0：標準輸入	Keyboard Input	Read
1：標準輸出	terminal	Write
2：標準錯誤輸出	terminal(錯誤訊息是寫到此)	Write

以下為大致說明如何使用標準輸入輸出

```
command > error.log     //當指令執行失敗時是直接顯示在 terminal 而不會寫入 error.log 內
command 2> error.log    //當指令執行失敗時會將錯誤訊息寫入 error.log (此時 terminal 不會顯
command 2> /dev/null    //當指令執行失敗時把錯誤訊息寫入黑洞(此時 terminal 不會顯示錯誤訊息)
command > 1.log 2>&1    //當指令執行失敗或是有標準輸出的訊息皆寫入 1.log 內
command 2>&1 > 1.log    //當 1.log 內容為標準輸出，而錯誤訊息顯示到 terminal
```