Scripts大集合

Cyril Huang
cyril@gyoza.homeip.net
餃子出版社

June 14, 2002

Contents

Chapter 1

前言

scripts其實是很有用基本的工具，很多事情其實靠scripts就解決了， 主要在做"環境"上的configuration。而且可以訓練程式寫作， 大型程式的寫作裡面Makefile，package製作，網頁製作， System Admin等等，都要用到scripts，甚至也可以用perl scripts寫個小server呢。 所以scripts是最基本功夫。其實扣除要寫系統和應用程式的話，像巨集型態的script 算是使用電腦的最高境界了。

script其實一般人不見得會用到，如果工作上沒有routine的工作。 爲甚麼大部分的人的電腦裡不會用到，例如修改一篇文章或者畫圖， 主要是這些工作不是常常在做，懶得去想一個general的規則。 例如我用script將測量得到的資料利用gnuplot作x-y曲線圖， 一般人會認爲幹麼這麼麻煩，用Excel畫不是簡單，問題是一般人可能一年裡面 只做他個幾次這種事，用老鼠搖啊搖也沒浪費多少時間，但是如果常常作測量， 處理很多次時，一般人也會去用巨集(macro)，不管word, excel macro， 這其實就是一種script。

scripts不需要編譯，所以可以很快的寫出來再修改，不像C/C++需要理會記憶體的問題 。（scripts就像BASIC語言，專注在人類思考的變數就好，C/C++就在那邊把資料在 記憶體移來移去， 組語就是專注在CPU內的小運算單位 - registers。這個世界就 是各個分層架構出來的，例如建築師不一定要會挑磚頭拿鐵鎚，但沒有這些基本工 具也不行，就算會拿鐵鎚，那鐵鎚怎樣做出來的??） 每一層都依賴下面一層的提供interface來使用，然後再提供上一層的使用者 interface。每一層每一層都有專家，所以Einstein說專家就是訓練有素的狗啊，不過 全世界能叫人的大概100年才有一個人吧。 所以scripts在做修改和一些routine job有很快速的彈性。

Again! 如果你的電腦基礎不是很好，可能這邊有的東西你會看得覺得很難，也可能找不到想要的資訊，例如.profile .bashrc這種初始檔名， 這不是給完完全全不會的初學者的，不過如果你想要多一點的觀念，希望這篇 文章對你有所幫助。

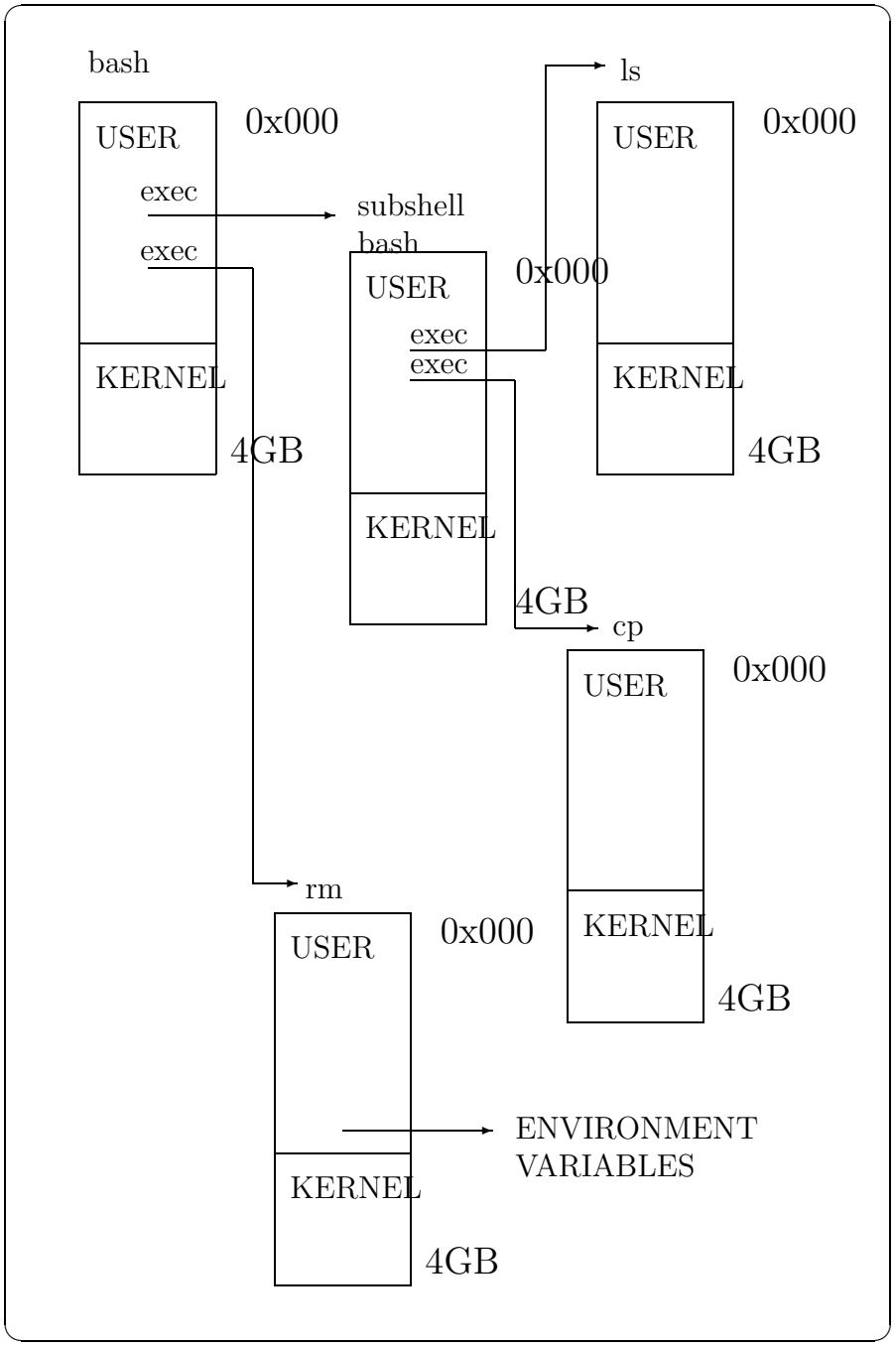
Chapter 2

中階Bourne shell

2.1 簡介

2.1.1 shell是甚麼

shell是一個接受使用者命令後，請求作業系統執行的一個應用程式，有人會把shell 畫畫包在kernel外面然後在包一層AP，我比較不贊同這種畫法啦， 對於我來說shell也只是另外一種application程式而已，其實它跟gimp, netscape等等是一樣的東西，只是會被login叫出來。所以他是可以替換的一種應用程式。 用C program來想，這樣子還可以幫你了解環境變數。


bash
USER
0x000
exec
exec
subshell
bash
KERNEL
4GB
USER
0x000
KERNEL
4GB
ls
USER
0x000
KERNEL
4GB
exec
exec
cp
USER
0x000
KERNEL
4GB
rm
USER
0x000
KERNEL
4GB
ENVIRONMENT
VARIABLES

寫一隻shell其實非常簡單，主要是捉到user打的命令後，用fork跑出另一process （記住unix like是多人多工的然後在要這個child process用exec去執行這個 外部程式例如ls, rm...等等。所以shell內部的運作大概是這樣的）

1. shell程式控制住鍵盤螢幕
2. 打出prompt等待使用者輸入命令
3. 捉到使用者輸入
4. fork()一個新process
5. exec()命令
6. 執行完這個命令又回到shell

在這邊您會看到unix like的執行一個程式都是藉由exec這個system call來執行的， 這是作業系統提供使用者程式執行程式的機制。system call是user程式跟kernel 溝通的唯一管道。

在以前MS DOS時代，一開機後會去執行command.com這隻程式就是shell。他就會跑出

```
C:\>
```

等使用者輸入命令，這個C:就是提示符號(prompt)，bourne shell通常prompt是$，只不過這個dos shell功能相當的差，所以當時有個很有名的4DOS這個shell可以取代 他。(4DOS這個shell其實拿了很多unix上shell的特性。當然跟眞的unix shell還是 不能比) 如果您只是用老鼠按一按一些icon，後來的MS-Windows時代，X的時代的 command line shell就被圖形化的介面給取代了。不過執行另一個程式的機制還是一 樣，還是要exec()。

unix like 的系統的命令交談式的shell基本上除了能執行外部命令，主要他可以"解釋"一些字串來執行，這些字串script的好處是可以設定一些環境。 把多個外部命令集合起來完成一件工作。 例如在Windows下我們也會設定TMPDIR這樣的目錄指到那個目錄去，有些windows程式 會把執行中的一些暫時檔案放在這裡。unix like的shell scripts語法可以有if, while, for loop等等，搭配一些外部命令，可以隨著環境不同時有不同的設定， 使得一些日常工作維護更得心應手，比老舊的dos shell更 爲強大。

sed, awk....，這些外部工具久而久之已經算是標準的unix系統必備的命 令工具，就像winzip可能變成每個windows都會裝的工具。很多工具算是不成文的內定 輔助shell programming工具了。

2.1.2 執行shell script與subshell

兩種方法

- 喚起新shell再執行shell scripts

餃子出品必屬佳作

- 在目前shell執行shell scripts

喚起另一個shell來執行的scripts在scripts檔頭最前面前要加

```
#! /bin/sh
```

第一種方法是在shell script 文字檔前指出shell scripts解讀的程式在那(也就是我們的shell)然後把文字檔的執行權限打開，照一般執行可執行檔方式執行或者叫 一個shell來解釋文字檔test.sh。

```
$ test.sh
$ /bin/sh test.sh
$ ( . test.sh; )
$ exec test.sh
```

第二種方法是用命令"."或者source執行。

```
$ . test.sh
$ source test.sh
$ { test.sh; }
$ eval '. test.sh'
```

差別在於一些設定只有在這個shell下的才算數，而喚起另一個shell就是另一個不相干的世界， 也就是用第一種方法執行的script中變數的設定，不會影響到原來的shell變數。 這個相當重要。ksh沒有source這個命令，所以最好不要用source。 中括號( )表示用另一個subshell大括號，{ }表示用目前shell。例如

```
$ ( VAR='testvar'; )
$ echo $VAR

$ { VAR='testvar'; }
$ echo $VAR
testvar
```

只有{ }內的VAR中的值被設定。其中用 . 的方法要很小心， 不要在script裡面用

```
$ . test.sh arg1 arg2
```

因爲arg1 arg2會繼承呼叫這個script的arg1 arg2來，用 . 的方式最好是要執行的script只是一團script library不帶參數。 另外如果 . test.sh執行，test.sh離開時，呼叫. test.sh的shell也跟著離開。

餃子出品必屬佳作

2.2 變數函數與模組化寫作

2.2.1 變數使用

給變數值用

```
var="value"
var='value'
var=`command`
```

要把值拿出來要加個錢符號$

```
$ var="I am a gyoza"
$ echo $var
I am a gyoza
$ var=`ls *`
$ echo ${var}

$ var='$var'
$ echo $var
$var
```

以上要注意的是空格是有差別的還有引號的不同與作用。請往下看

2.2.2 引號quote

鍵盤上的double quote, single quote, backquote要弄清楚。

- backquote `command` 這個是命令取代，等於 $(command)。 所以常常看到var=`command或者var=$(command)， 就是把command執行結果設給var
- single quote 在' '內的文字通通保持原樣不做代換，有些保留符號在單引號 內不再保留特殊意義。
- double quote 在" "內的文字$ ` "三種意義會代換其它的保持原樣
- escape char 在\ 後的單一字元也可以保持原樣

2.2.3 中括號的變數取代

加上括號{}是parameter的方法， ${var}是比較好的寫法，如果沒有要加其他字串可以用$var就好。 小心空格是有差的，他不像其他程式寫作，多空一個與少空一格是有差的。 其他變數一些用法：

- ${var:-value} 如果var有值了那麼就用原本的值，不然用value的值
- ${var:+value} 如果var有值就用value的值

餃子出品必屬佳作

- ${var:?message} var有值那麼就用原本的值，不然就印出message 值到螢幕並且跳出。
- ${var:%pattern} 如果pattern與var後面的部份吻合，傳回剩下沒有 吻合部份給var
- ${var:#pattern} 如果pattern與var前面的部份吻合，傳回剩下沒有 吻合部份給var
- ${var/pattern/substitute} 如果var有pattern吻合就代換成substitute

請看一個例子

```
# remote shell $RSH會等於/usr/bin/rsh  如果RSH當初沒有給值
RSH=${RSH:-/usr/bin/rsh}

# :+ 這種方式用在可有可無的option很好用
# 以下面這個副程式爲例子
# 如果傳參數給他則VAR變成"-o 參數"，沒有參數則VAR沒有值
# $1 是第一個傳進來的參數  如果有值VAR就用"-o $1"
func()
{
    VAR=${1:+" -o $1"}
}

# :% 這種可以用在擷取字串的某部份
# 例如  find 或者有的命令取回的結果往往是絕對路徑名
# 但我們只想要最後面的那個檔案名時可以用這個來擷取
# 不過傳統的bourne shell沒有% # pattern match，這只有在
# ksh bash才有 最好不要用改用sed比較保險一點

PRIV_HOST=fermion-priv
PUB_HOST=${PRIV_HOST:%-priv}
echo $PUB_HOST
(PUB_HOST會等於fermion)

# 代換也是  在新的Korn shell與Bourne Again Shell上才有

BLOCK_DEVICE=/dev/vx/dsk/oracledg/vol_0
RAW_DEVICE=${BLOCK_DEVICE/dsk/rdsk}
```

2.2.4 環境變數

普通的變數 如果Bourne shell, korn shell下用

$ export var

餃子出品必屬佳作

c shell, turbo c shell用

$ setenv var

則這個變數就變成一個環境變數。在C裡面的

main(int argc, char **argv, char **envp)

我們可以在shell裡設環境變數，而這個值是每個由這個shell fork出的程式，經由envp都看得到的。這個argv, envp指的都在virtual memory的下面stack 往上長的開始處。請看拙作"用Open Source tools開發軟體"。 所以shell也不過是一個"User space的C program"，環境變數藏在C image user space最下面，也可以從getenv這個library function call拿到。 所以一般光設變數，沒有設環境變數沒有辦法把值告訴其它的程式。 這邊要注意的是原本的Bourne shell的export沒有支援

$ export var=value

的寫法，所以看到一些shell scripts爲了portable起見都用

$ var=value
$ export var

2.2.5 一些內定變數

- $? 前一個命令執行完的status， 0表示沒問題，在程式設計裡，0表示FALSE也表示No ERROR，所以程式的exit 與return要處理好
- $# 表示傳給shell的arguments數量
- $0 這個shell命令名字
- $1 $2 $3 ... function傳進來的參數arguments，
- $@ 表示function 傳進來的所有arguments，會等於 "$1" "$2" "$3"
- $* 表示function 傳進來的所有arguments，會等於"$1 $2 $3"
- $$ 表示目前的process ID

這邊$@ $*是不一樣的要小心single guote, double quote的不一樣。 $?用在測試條件的判定是最常用的。

shell一開始也會設定一些內定環境變數，這些環境變數會有一些程式自動會來讀取， 例如mail程式用的MAIL，還有指明命令所在搜尋目錄的PATH等等。 這些是寫bash, ksh, csh的C一開始寫在C程式內的。 看環境變數用env這個命令，看變數用set命令就看得到不一樣結果。

餃子出品必屬佳作

2.3 函數與多檔模組化

函數的定義很直接就是

```
func()
{
    cmd
    cmd
    ...
}
```

傳值也是一樣的道理$1 $2 ...$@ 等用法是一樣的。 呼叫時就是

```
func arg1 arg2 ...
```

要注意的是變數是沒有像C一樣的有local variable，通通是global的。 簡單的perl也是這樣。

2.3.1 模組化程式

function是最簡單的modulized寫作，再來就是多個模組檔當成shell lib檔。 用了多個檔的問題跟C是一樣的就是變數怎樣在一個檔定義了，然後讓別的檔參照。 通常export 成一個環境變數或者用eval這個內部命令執行一個個shell script使得 這變數是大家都看得到的。

2.4 重要外部與內建命令

我們常用的ls, cp, sh....等等這些都是一個可執行檔，相對於一個外部可執行檔， shell有一些內建的命令藏在shell裡面，這些是built-in的內部命令是當初用C寫 shell這隻程式時就寫在裡面的，像cd, export這就是。

bash其實有很多以前的外部命令他都包含進來作成內部命令了，例如echo, getopts， 不過爲了程式可攜性，我們還是把他們稱爲外部命令。這些內建命令可以用enable, disable在bash內取消掉。不過別的sh, ksh或許就沒有了。要小心。

2.4.1 內建命令

比較重要且難一點的內建命令有

- : 假指令，甚麼都不作，等於CPU內的NOP指令。
- eval 大部分用來作兩次變數替換
- exec 執行某種命令並且替換掉目前shell(原本的執行不會替換掉shell)

餃子出品必屬佳作

- read 得到使用者從standard in(鍵盤)輸入
- set shell的精細設定，例如debug shell scripts
- test shell scripts的控制語法內的測試條件
- trap 接到某種訊號的處理
- shift arguments的處理

: magic numbers與註解

:是不做任何事的，在Unix中系統要執行程式會去檢查檔案前的字串叫 magic number，古老的Bourne Shell的執行magic number是 : 而不是 #! 所以看到這種以:開頭的shell script可就要肅然起敬，同時:也有人拿來作註解用 而不用#的，這些老習慣的shell script可都是很厲害的。

eval與脫逃字元

eval可以拿來執行一個命令，不過他最常用的是拿來作兩次變數的代換， 主要是每次執行一個shell 命令他會先evaluate一次，看到有$這個東西的就把值換一次把變數換掉， 然後再執行一遍。這種double scan的方法對一些變數代換很有用， 因爲eval不是喚起另一個shell來執行，而是在本來這個shell內多evaluate一次， 所以代換結果可以保留下來。 例如如果我們要兩次代換

```
count=1
var1=I
var2=am
var3=a
var4=gyoza
while [ $count -lt 5 ]; do
   eval "echo \$var$count"
   let 'count=count + 1'
done
```

count可以一直變化1. 2. 3 ....要產生一個新變數var1 var2 var3....然後再對 var1 var2取值。其中因爲第一個var不想被運算，所以先用escape字元\， 然後第二次運算時才被解釋。 那如果要三次以上變數代換在一行內解決呢? 想想看吧。

eval主要還用來evaluate執行一個shell script檔，可以像C一樣寫成很多的 模組shell script在同一個shell下run，則變數在此shell內通通有效。

```
$ eval ". foo.sh"
```

不過如果變數太多，名字會打架。

餃子出品必屬佳作

exec

exec 會把"目前"的shell整個process拿掉，換成後面的命令，其實這就是用exec()這個system call置換掉子行程的意義是一樣的。最常看到是在 /.xinitrc這個scripts中置換掉成window manager。例如 exec twm。所以如果你在shell中執行

```
exec cmd
```

而cmd這個命令不存在就會回到login去，因爲整個shell被換到cmd，但卻沒有cmd這個執行檔。所以執行程式的方法兩種是不一樣的

```
$ exec cmd
$ cmd
```

放在scripts的執行當然也不一樣。不過exec在script有另一個相當重要的用途就是跟file descriptor的連結，這個等下面再來討論。

read

```
$ read input
read from standard input
$ echo $input
read from standard input
```

則會停在這邊等待鍵盤輸入，輸入字串變成$input這個變數的值。

set

set是拿來設定這個shell的執行環境的。比較重要的script會用到的大概

- set -- 正常看到-後面是option，現在不再是option 而是一個命令參數。如-1 -2 ...
- set -a 從這邊以後變數自動變成環境變數。
- set -f 不要解釋檔名的特殊字元例如wildcard *不再解釋爲所有的意思了。
- set -x debug shell scripts
- set -o ignoreeof 一定要用exit離開shell,本來按Ctrl-D(eof)也可以
- set -o noclobber 關掉I/O導向不準overwrite檔案
- set -o notify shell結束時報告background job的status
- set -o noglob 關掉wildcard字元解釋 如 * ? [ ]
- set +o 把-o的反向操作

餃子出品必屬佳作

- set - 關掉-v -x –三種選項

set -- 或者set - 常常用在shell scripts裡面。 set -o 是很常用的例如set -o vi設定shell的操作方式用vi方法， 取回上個命令就是按ESC再按k囉。 set -o emacs就是用emacs的方法。

test

test是控制語法中的測試條件，這個在下面討論。

trap

捉到某個signal時shell做的對應，

```
    通式
    $ trap "command" signo
    其中signo是
    cyril@grill:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR      31) SIGSYS
    例如
    $ trap "" 2
    $ trap "rm $TMPFILE" EXIT 1 2 15
```

如果""的command則表示shell不處理這個signal， 2號就是INT通常就是按了Ctrl-C打斷shell script的執行，15就是 TERM(process被kill了)，再看難一點的例子

```
trap_init()
{
        trap '  scriptcleanup
                [ "$SCRIPT_DISP" = ABORT ] && exit 100
                [ "$SCRIPT_DISP" = PASSED ]; exit $?' EXIT
        if [ -z "$__TC_INTERACTIVE" ]
        then
                for sig in HUP INT TERM; do
                        trap "  trap - HUP INT TERM;
                        echo 'Signaled - cleanup after script ...' >&2;
                        scriptcleanup $sig; kill -$sig $$; exit 101"  $sig
                done
        fi
        trap : PIPE
```

餃子出品必屬佳作

```
}
```

這個例子的前面如果抓到EXIT這個signal就執行scriptclean到exit $?的code，就是單quote內的東西， 如果$__TC_INTERACTIVE不是空字串，就執行下面的signal處理， 最後如果是SIGPIPE(13號)就不做任何事(冒號:)

shift

shift是用來把進來的arguments移一個位移， shift n 移n個位移， shift可以拿來用超過10個以上的argrments，因爲$1 ... $9只有9個而已。 來看一個shell的queue list的append

```
# list_append list_name item ... - append items to a list
list_append()
{
    _list_a_name=$1; shift
    eval "set -- \${$_list_a_name} \$*"
    eval "$_list_a_name=\$*"
}
```

2.4.2 外部命令

有些外部命令常用到

- basename 拿掉副檔名的處理
- echo 印出message的方法
- find 找檔案
- getopts 處理傳進來的arguments，跟C函式庫的getopt()很像
- xargs pipe時的命令執行與參數arguments的處理

另外sed awk會有專門介紹。

basename

捉出檔名副檔名工具。

```
$ basename foo.c .c
```

則副檔名 .c 會被幹掉

餃子出品必屬佳作

echo

echo是印出東西到螢幕的手段(也有print, printf可以用)， 這邊要講的是sh, ksh內的echo與bash的不一樣，所以設定有些不一樣。

```
echo -n  "messages"      不要換行(newline)
echo -e  "messages\c"    解釋escape字元\c，等於-n用法
echo -e  "\t messages"   tab在messages前
```

傳統sh, ksh是沒有-n -e選項的，直接用echo就好。

```
echo "messages"      要換行(newline)
echo "messages\c"    解釋escape字元\c 不要換行(newline)
echo "\t messages"   tab在messages前
```

getopt

getopts其實是ksh, bash的內部命令，getopt才是眞的外部命令。 這很像C裡的getopt()。請看個例子

```
while getopts :dvt:n: c
do
   case $c in
      d)
          DEBUG=yes
      ;;
      t)
          TYPE=$OPTARG
      ;;
      n)
          NUMBER=$OPTARG
      ;;
      v)
          VERBOSE='yes'
      ;;
   esac
done
```

其中冒號:是說這個參數一定要有跟著的參數值，沒有冒號像dv表示後面沒有帶著參 數值， 例如最常看到-v是說程式執行時是verbose mode就是這樣。$OPTARG就是跟在後面的 參數，getopts自動幫我處理好，並且一個一個的丟進 $OPTARG來。他也有$OPTIND跟C library的用法很像

find

尋找檔案用的程式命令，

餃子出品必屬佳作

```
find . -name "regex"
find / -type d 找directory
             f 找plain file
             l 找symbolic link
             p 找named pipe file
             b 找block device
             c 找char device
find . -perm 755
find / -user cyril
gzip ‘find . \! -name ’*.gz’ -print‘
```

not ! 有的shell要加\ 因爲!有特別意義，例如bash表示取回以前的命令。

```
find -name -type -o -name
```

在script中很多時候都是脫逃字元問題，常看到\時都要想到這是脫逃字元。在script中 ()也有特別意義表示喚起另一個subshell來執行所以碰到()時也要用脫逃字元。或者合sed awk一起使用時也是一樣的道理。

xargs

xargs用來處理一些輸出結果要來當另一個輸入的arguments碰到的問題。 例如

```
# find /usr/include/ -name "*.h" | xargs -n 2 diff
```

-n2是指定有兩兩當成輸出變成diff的argumnets

```
# find /usr/include/ -name "*.h" | xargs grep ’#ifdef’
```

正常內定是輸出的一拖拉窟的結果，有用xargs時是一個一個餵給後面的命令

```
# find /usr/include/ -name "*.h" | xargs -i cp {} ~/include/
```

-i 與 { }可以把find的輸出的每一個當成cp的第一個argument， /.就可以當成第二個argument。 其實find裡面有-exec這個選項後面也可以用{ }表示一個一個餵給後面程式，而不是 一拖拉庫的餵給後面程式。

```
# find /usr/include/ -name "*.h" -exec cp {} ~/include/
```

2.4.3 簡單數學

主要是拿來作counter的，有很多種方法，下面介紹三種， bash有個內建的let， 例如

餃子出品必屬佳作

```
let 'x = x + 1'
let 'x = x * 1'
```

注意!!這邊等號右邊的x取值時不用加錢符號，而且在單quote內也可以有空格。 ksh也有內建的，不過這不是每種shell都有的， 比較保險的portable作法是用expr這個外部命令。 請看例子

```
 if [ "$KSH" ]
 then
        eval '
        add()
        {
                result=$((${1} + ${2}))
        }
        sub()
        {
                result=$(($1 - $2))
        }
        mul()
        {
                result=$(($1 * $2))
        }
        div()
        {
                result=$(($1 / $2))
        }
        inc()
        {
                eval "$1=\$((\${$1} + ${2:-1}))"
        }
        dec()
        {
                eval "$1=\$((\${$1} - ${2:-1}))"
        }
        '
else
        add()
        {
                result=`expr $1 + $2`
        }
        mul()
        {
                result=`expr $1 * $2`
        }
        div()
        {
```

餃子出品必屬佳作

```
        result=‘expr $1 / $2‘
}
sub()
{
        result=‘expr $1 - $2‘
}
inc()
{
        eval "$1=\‘expr \${$1} + ${2:-1}\‘"
}
dec()
{
        eval "$1=\‘expr \${$1} - ${2:-1}\‘"
}
```

這上面定義了一些副程式，可以在script裡面一樣像呼叫一般命令呼叫。$((1+1))是一個subshell執行1+1並且傳回結果，下面例子讓我想起我的basic程式

```
i=0
while [ $((i=$i+1)) -lt 10 ]; do echo $i; done
```

不過$(( ))這跟${VAR%value}一樣只有ksh bash有，最好不要用在需要 portable的shell程式上。 副程式的position parameters $1, $2 ....跟平常script程式原則一樣。

如果要更多的數學例如小數點的運算或者sin log等函數使用， 就用bc和awk吧。

ksh與bash都支援typeset，不過傳統的sh並不支援，bash還有declare這個內部命令， 這兩個都是拿來定義這個變數是甚麼性質的很像C的宣告。不過爲了portable能不用 就不要用，還是用最基本的sh就好，如果眞的要寫得很複雜就用perl吧，如果非常的 嚴謹就用C囉。

```
$ typeset -i int_var
$ declare -r constant
```

int_var被設成整數，如果給字串則int_var的值會是0。如果用-r 則constant的值從現在起不能再被改了。

2.5 File descriptor與I/O導向

一個程式最基本的會自動開啓三個檔，操作時相對應的代號就是file descriptor， 其實也就是作業系統每次要到的file descriptor，是per process的

- 0 standard input 通常就是鍵盤

餃子出品必屬佳作

- 1 standard output 通常就是螢幕 buffer I/O
- 2 standard error 通常就是螢幕

我們可以用I/O導向

```
$ cat file1 > file2
$ dirs 2>&1 > /dev/null
```

執行dirs時所產生的standard error丟到standard out再到黑洞/dev/null去， 就是甚麼也看不到。 exec 最常用在與file descriptor的關聯這裡

```
$ exec 3> $LOGFILE
$ exec 3>> $LOGFILE
$ exec 2>&-
$ exec < infile
```

其中各意義分別爲 開一個3號file descriptor相對於$LOGFILE。把standard err關掉。 把目前shell的standard input變成infile

```
$ cat something >&3
$ read <3
```

現在3這個file descriptor就相對於LOGFILE了，所以cat something的內容會到LOGFILE去。 注意file descriptor 1,2,3與導向符號的空格。

一些重要的I/O導向:

```
>       重新導向並且先把檔案虧空(我眞是敗家子:-))
>>      把結果導向並且append到檔案後
<       重新導向standard input
>&n     把standard out丟給file descriptor n
<&n     從file descriptor n拿東西給standard in
>&-     close standard out
<<text 把stanadard in導向，直到讀到text爲止。
```

要注意的是有些後面只能接檔案，有的接file descriptor，還有 接file descriptor時不能有空格。 自動ftp的scripts，在很早很早以前，中山大學的ftp站可是有名的X情網站， 不過他只會在半夜 才開放目錄的存取，所以我們就叫電腦在半夜時cron job自動幫我們ftp。哈哈

```
#!/bin/ksh
# A script to automate FTP transfer

HOST=ftp
USER=user
```

餃子出品必屬佳作

```
PASSWORD=password
FILENAME_PATTERN=remote_files
REMOTE_DIR=/usr/doc
LOCAL_DIR=/usr/local
# -i = non-interactive, -n = disable auto-login

ftp -i -n <<HERE
     open $HOST
     user $USER $PASSWORD
     cd $REMOTE_DIR
     lcd $LOCAL_DIR
     mget $FILENAME_PATTERN
     close
     bye
HERE
```

注意<<的用法，ftp的input會一直讀到HERE爲止，open, user, cd, mget 都是ftp中的命令。不過也可以把 這些ftp命令寫成一個檔案然後ftp < 檔案也行。

2.6 流程控制與測試條件

2.6.1 測試條件

在Bourne shell的內部命令裡面有測試條件的語法test給if while用

```
test condition
或者
[ condition ]
```

爲了與autoconf不要混淆，programmer比較喜歡用test 不用中括號condition的方法。括號的用法請注意空格。 例如

```
檔案測試條件
test -f /etc/file       檔案是個一般檔案不是其他特殊檔案嗎
[ -e file ]             檔案存在嗎
[ -d /etc/ ]            目錄存在嗎
[ -s file ]             檔案大小大於0嗎
[ -r file ]             檔案可讀嗎

字串測試條件
[ "$string" ]           string有東西就返回true
[ -n "$string" ]        string有東西(non-zero)返回true
[ -z "$string" ]        string沒東西(zero)返回true
[ "$s1"  = "$s2" ]      s1 等於 s2時返回true
```

餃子出品必屬佳作

```
[ "$s1" != "$s2" ]     s1 不等於 s2時返回true

數值條件  小心有多個-喔
[ $num1 -eq $num2 ]    num1相等    num2 爲true
[ $num1 -ne $num2 ]        不等    num2 爲true
[ $num1 -lt $num2 ]        小於    num2 爲true
[ $num1 -ge $num2 ]        大於等於num2 爲true


多重條件
[ ! -f "testfile" -o ! -r "testfile" ]
[ test condition -o test condition ]
!               表示not
-a     表示and
-o     表示or
```

通常比較常用的又有portable的就是上面一些用法。美觀上來說用中括號condition比較好看，很多人爲了autoconf的語法portable起見，盡量用test的寫法。測試的結果當然放在$?中，0表示成功，其他值表示失敗。

在shell script中也有可能看到有人用

```
if [ "x$VAR" = "xvalue" ]; then .....
```

來作$VAR是否是空的測試，尤其你如果先測試是否爲空字串，再測試是那個值要做什麼，這樣就會作兩次測試划不來，用這樣作比較經濟。

另外這跟perl字串與數值測試容易混淆，他跟perl剛好相反，而且perl沒有多-

```
perl語法
if ($str1 eq $str2)
if ($num1 == $num2)
```

最後有個大比較，會一起列出shell perl c的差異來。(人老了記不住，我是這樣記，以perl爲基準eq是字，所以前後是string，==是符號所以前後是number，bourne shell是怪胎，剛好顛倒，eq還要加個-符號。==要少一個=)

```
C語法
if (!strcmp(string1, string2))
if (num1 == num2)
```

2.6.2 if 條件判斷

if test;then cmd1; elif test; cmd2; else cmd3 fi

餃子出品必屬佳作

```
if test
then
  cmd1
  ...
elif test
  cmd2
  ...
else
  cmd3
  ...
fi
```

同樣注意寫成一行時的分號位置。尤其在寫Makefile時會用到。 另外他的else if是elif， 跟C的else if不一樣喔。

另外有種簡單的if用法

```
[ "$VERBOSE" ] && echo 'everything'  && 表示傳回true時執行&&後面動作
[ "$VERBOSE" ] || echo 'nothing'     || 表示傳回false時執行||後面動作
```

這邊有點要注意

```
[ "$var" ]
如果是空字串傳回false 這其實很像
[ -n "$var" ]
```

通常括號裡面是這些判斷的方式，但其實每次他如果看到有命令代換的式子， 會去執行一次命令， 但是記住他不是C，可以在if while裡面執行式子(expression)還回return值， 他所還回來的是命令的standard out，所以不可以嘗試著同時執行命令， 然後要判斷執行成功與否，來決定下一步怎麼作。不過有的命令成功了會在standard out 印出一堆字串，我們可以利用這樣的行爲來 作判斷，例如

```
[ "$(find . -name gyoza.txt)" ] && echo "I Find It !!!"
```

用$(cmd) 等於用‘cmd‘，如果找到了，會印出這個檔案來所以雙引號裡面” ”有值， 中括號

就還回true囉。如果沒有雙引號，如果只有一個字串則會當成字串永遠爲true，所以

餃子出品必屬佳作

```
$ [ 1 ]
$ echo $?
0
$ [ 0 ]
$ echo $?
0
```

如果有兩個以上，則shell把find丟出來的字串又當成命令，想去執行他，就發生 錯誤了。

2.6.3 for loop

for var in $list; do cmd1; cmd2; done

```
for var in $somelist
do
  cmd1
  cmd2
done
```

注意寫成一行時的分號位置。尤其在寫Makefile時會用到。 list是shell裡常用的一種方法，就只是很多小單位用空格或tab分開的資料就是。 例如

```
ME="I am a gyoza"
```

ME就是一個list(bash的man page用list表示一堆commnands， 把word表示這裡的list，有的書或者ksh, sh的man page又不一樣。 所以我很不喜歡去定義這些名詞，有時候爭論這些沒有意義 捉到key point比較重要)for 敘述可以一次把一個元素餵給前面的var

```
for ELEMENT in $ME
do
  echo $ELEMENT
done
```

就會出現

```
I
am
a
gyoza
```

注意錢符號與設定變數使用變數的差異。

餃子出品必屬佳作

2.6.4 while loop

while test ; do cmd1; cmd2; done

```
while test
do
  cmd1
  cmd2
done
```

同樣注意寫成一行時的分號位置。尤其在寫Makefile時會用到。

2.6.5 case switch

```
case $var in *)cmd1; cmd2 ;; xx?)cmd3; cmd4 ;; esac
```

```
case $string in
     *) cmd1; cmd2
     ;;
     xx?)
        cmd3
        cmd4
     ;;
esac
case $answer in
     [Yy])
        echo 'yes'
     ;;
     [Nn])
        echo 'no'
     ;;
esac
```

case switch的用法，不像C的是整數的case switch而是可以有字串的喔， 其中符合的字串條件用globbing的方式，也就是wildcard *, ?, [ ] ，有的書上寫regular express是錯誤的。注意用一行寫出的方法。

2.7 Portable Shell與remote shell

remote shell執行是眞的把遠端的standard out 連到自己的standard out 所以他不像一般telnet程式

其實如果工作環境確定只在bash上那也不用考慮一些跨平台的考量。 不過如果作development時要考慮到portable的寫作，應該用最原始的sh就好，

餃子出品必屬佳作

2.8 結語

一般標準用shell是Bourne Shell後來延伸出Korn Shell，很多老一輩的工程師 還比較喜歡用korn shell寫scripts。另外還有BSD的C shell， c shell也造成一股風潮，尤其在很多system admin，後來tcsh加進很多不錯的特點， Bourne Again Shell把大家有的沒的特點拉拉紮紮放在一起。拜GNU/Linux之賜 年輕一代的工程師很多喜歡bash，不過要注意portable的shell scripts寫作。 sh, ksh bash是同一家族的，這幾個語法比較接近，不像tcsh csh是另一家族， 差很多。一般說來sys admin喜歡用c shell，傳統的programmer用sh的較多。

Chapter 3

Regular Express

3.1 簡介-RE與glob

regular express是一種表示方法，用一些特殊的字元來表示一些電腦裡的特殊意義，並且做爲尋找的一種樣式比對 (pattern match)。 在MS dos/windows中也有相當的字元，例如 * 代表所有字所以你用del *.* 就可以砍 掉所有檔案，* 就叫wildcard字元。不過這在shell中的處理只是叫做globbing 並不是regular express。其實這當然是從Unix學去的，UNIX上的regular express更強大，而且適用於很多Unix 軟體裡面，像grep, sed, awk, vi, emacs perl...等等，可以說regular express就是使用UNIX的靈魂。 不過要來學unix請先把DOS那套不general東西給忘了。

C程式裡面處理regular express如果是POSIX的系統則用 `regcomp() regexec()`，System V的有`regcmp(), regex()`，BSD系統有`re_comp(), re_exec()`。

在shell中的pattern match叫globbing，shell是不認得regular express的 通常用於路徑名的？* [] 的比對，對應用`fnmatch()`來處理。 不過這不是要談的重點，只是以後進階用C寫程式時可以想到用甚麼function call。

通常很多使用regular express的時後會用一對//夾住。

```
/reg express/
```

來表示找到合乎reg express條件的字串。perl裡面說其實是

```
m//
s///
```

m表示match，s表是substitude，不過m可以省略不寫就變成//了 通常用法是m//命令 s///命令。命令常看到的有

- p -> print 通常也不用寫

- d -> delete 這通常是跟在m命令後 符合的就砍掉
- c -> confirm 通常用在s命令 詢問是不是眞要取代
- g -> global 通通換 因爲match到時內定只有第一個符合的有效
- i -> ignore 不管大小寫

不過也不一定，像grep命令就沒有，用雙引號就可以了。 anyway,只是很多都是這樣表示。

3.2 字元處理

3.2.1 基本字元表示

字元，就是單一一個英文字母的處理。先來個基本的字元

- . 代表任意"一個"字元除了換行newline字元不過awk可以match換行記住
- ? 代表"前面"字元出現0次或1次
- + 代表"前面"字元出現1次以上 1 2 3 4 ... 不出現return false
- * 代表"前面"字元出現0次以上 0 1 2 3 ...

這邊初學者比較常犯的錯誤是在shell下的wildcard * 的習慣，以爲只要給個* 就代表所有字元，其實是 ? + * 都要前面搭配 . 來使用才能代表任意字元

- ^ 代表行首
- $ 代表行尾
- \ 脫逃字元(escape)取消特定字元的涵義用的

這邊比較要注意的是字在行首起頭的^，行尾的$，與在行中的case分屬不同的regular express，小心有的程式會把這三種當成不同的regular express， 這會不一樣喔。不過有的不會。

- /^#/ 有#在一行的第一字元時 就還回TRUE
- /^bag/ bag在行首的行
- /bag$/ bag結尾的行
- /bag/ bag在中間的行 不過有的程式處理沒有這麼嚴格像sed
- /b.g/ 表示bag beg bug都行
- /b\.g/ 表示b.g
- /^..g/ 表示行首bag beg bug xxg都符合 但一定要三個字母
- /^.*g/ 表示行首有任何字元的然後有個g的都可以還回true

餃子出品必屬佳作

- /beg*/ be, beg, begg, beggg, ...
- /beg+/ beg, begg, beggg, begggg, ...
- / */ 空白字元也行

3.2.2 字元處理 - 括號與範圍表示

- [ ] []內的任一"單一字元"符合就還回TRUE
- [^] []的反效果 不含[]內的任一單一字元 還回true， 所以這邊^在[]內有不同的意義
- [1-10] -號在[]內可以表示一段範圍不用打到手酸死， 所以如果要表示-號必須放在第一或最後字元
- w{n,m} 連續字元出現表示法
  比用 /.../好用 跟LaTeX中的table的用法很像，表示有"連續"符合 w字元出現"連續" n次到m次
  都會還回TRUE
- 0{3,} 0至少出現3次 -> /xxxx000xxxxxx/

在字串下{} ()是表示眞的這些字元的，不像[ ]會被regular express當成一種運算，所以不要忘了用脫逃字元 變成\{ \} \(\)。

3.3 字串處理 - 不同括號表示

一些字串的處理上

- ( ) Group operator
- (str1|str2|str3) str1或者str2或者str3
  () 與 |是extend的regular expression 只有一些軟體如egrep才有支援。所以在用軟體的regex 時必須知道他能處理的regex能力。
- & 表示找到的字串
- \1 \2 \3 ... 代表s///中前面用括號\(\)括起來的字串， 這通常也是找到的字串，不過&只有一個， 用 \1 \2 \3 可以有很多個。
  \1 表第一個括號內字串
  \2 表第二個括號內字串

通常\1 \2 \3是用來對match到的字串還要再處理時用的

```
/[Yy]es/          Yes 和 yes
/80[23]?86/       8086 80286 或者80386
/[A-Za-z0-9]/     字元可以有這樣的連續表示法
/compan(y|ies)/   company companies
/0\{3,\}/         表示0要出現三次以上
```

餃子出品必屬佳作

```
s/.*/(&)/    將原本的行加上括號( )
s/\(str1\) \(str2\)/\2 \1/  把兩個字串對調  注意\1 \2的用法
                        其中& \1 \2 \3 ...這些常用在代換(substitue)中
                        注意括號在前面有不同意義，所以必須用\來escape。
```

3.4 範圍定址(address)

在sed, vi等editor裡我們可以指定要處理的行範圍， 這種指定要處理的條件行或某段range也叫address(定址)。 尤其是替換命令很常用。

range(addressing)s///

```
sed
$ sed '1,3d' file    在第一行到第三行間幹掉整行

vi
:%s/xx/yy/g          把整篇文章的xx換成yy
```

常用的範圍符號

- , 行數限制 1,5 表示從第一行到第五行
- 0 最上一行
- . 目前行
- $ 最後一行 5,$ 表示從第五行到最後一行 .,$目前行到最後一行
- % 整篇文章也等於1,$
- x-n x往上數n行 .,.+10 表示目前行到目前行加十行
- x+n x往下數n行

3.5 greedy的regular express

所謂greedy是說如果一行裡面符合regular expression的情況有很多，也就是 同時有很多pattern都符合時，通常會很貪心的符合最長的那個pattern，不過 並不是我們要的，尤其在HTML的tag處理上例如用/t.*t/去找，<tt>this is a tag</tt>在同一行裡 有<tt>也有<txxxxxxxxxt>，greedy的處理只有perl有比較簡單的方法。這將在perl裡面談到。

Chapter 4

sed

4.1 簡介

sed是一種Stream EDitor，也就是餵給它一串資料流跟命令完成編輯的動作，這很適合自動化的scripts作業

```
file -> stream -> sed -> stream -> file
```

由於它必須有個in， 處理完後有個out所以不能同時改個檔案， 必須先處理舊檔案後送出成新檔名，把舊檔殺掉，再把新檔改名成舊檔， 又因爲他內定輸出是standard out所以常看到的方法是

```
$ sed 'sed_syntax' old_file > new_file
$ mv new_file old_file
```

不過sed通常是用來對字串的處理顯示，不是像一般editor來modify檔案的， 而且記住是一行一行處理的所以空白, tab, 換行這些字元很重要。

sed的一般式是

```
sed [address],[address][!]command[args] file
```

其中address,command，用單引號'xxx'包住。 address就是上面regular express的東西，常用的命令有d(delete), s///(代換) 一些簡單例子

```
$ sed 's/yes/no/g' file                    把檔案中yes換成no
$ sed '/save/!d' file                      把沒有save字眼的行幹掉
$ sed -e 's/\(.*\)\(#.*\)/\1/' xxx.sh      把#後面的註解幹掉
```

其中-e是常用的一個option，通常是用在兩個以上的選項時。 其中，由於( )有特別意義所以需要反斜線escape一下。

4.2 符合條件的addressing

address是sed裡的特殊用語，用來設定符合條件的行，以便對它執行編輯命令， 符合條件的條件指定方法有1. range(範圍) 2.regular express。

sed的address除了之前regular expression介紹的 1,3 . $ %這種表示法外 還可以直接用 /regex/找到要的行

```
$ sed '2d'        file   找到第二行幹掉
$ sed '/delete/d' file   找到delete這個字幹掉整行
$ sed '/RE/,$d'   file   從有RE字眼的行到檔案尾通通幹掉
$ sed '/^$/d'     file   幹掉空白行
$ sed '/START/,/END/!s/xx/yy/g' file > file1  把xx換成yy
```

4.3 pattern/hold space, 多regex條件與script檔

如果有兩個條件以上那可以加-e

```
$ sed -e xxxx -e xxxx file
```

例如先將註解行變成空白行然後砍掉所有空白行

```
$ sed -e 's/\(.*\)\(#.*\)/\1/' -e /^$/d file xxx.sh
```

sed其實每次都把一行放到pattern space，然後對他操作 執行結果再先放到pattern space這個地方，如果有 -e， 就接著對pattern space的行再做script的執行，所以好幾個條件是有順序的。 做完一行後，再來一遍，所以最通用的一般式應該是

```
[address],[address]{
command[args]
command[args]
...
}
```

或者

```
[address],[address]{command[args];command[args];...;command[args];}
```

每一行command相當於有個-e的作用會先把結果放到一個pattern space， 可以寫成一個sed script file然後用-f 執行

```
 $ sed -f script_file file
```

或者每個command的最後面要放個分號;。 另外sed有個hold space是個暫存東西的buffer，sed有些命令可以利用 hold space與pattern space，很像vi裡面的yy dd 先放到一個看不見的 buffer，再用p命令把它叫出來。

餃子出品必屬佳作

4.4 sed command命令

command有25個比較常用的

4.4.1 基本指令

就是先把這幾個編輯命令學好啦

- Append a\ 找到address後append所有a\ 後面所有字串
- Change c\ 改變一段文字
- Delete d 砍掉regex找到的字串那一行，記住是整行砍掉喔
- Insert i\ 找到address後在前面insert字串
- Substitution s/// 代換是最常用的
- Translate y/// 可以一個字元一個字元作不同的代換

a\ 命令與i\命令，注意a\ i\ 都要換行才開始要加入的文字。如果要加的文字 有換行要用\ 這個符號表示換行，如果要加入的文字有\， 可以用\\來escape掉\。不過這邊有很大的問題， 也就是如果文字裡面同時有\ ’怎麼辦。 由於shell中的單引號’沒辦法逃掉，這個只得另寫script檔解決。請看例子

```
cyril@gyoza:~$ cat last
第一行
最後一行

cyril@gyoza:~$ sed 'a\
> 中文試試看
> ' last
第一行
中文試試看
最後一行
中文試試看

cyril@gyoza:~$ sed '$a\
> 中文試試看 並且故意想辦法超過一行看看不做任何處理時  超過一行時
> sed會怎麼樣處理這樣的問題
> ' last
sed: -e expression #1, char 93: Unterminated `s' command

cyril@gyoza:~$ sed '$a\
> 中文試試看 並且故意想辦法超過一行看看不做任何處理時  超過一行時\
> sed會怎麼樣處理這樣的問題
> ' last
第一行
```

餃子出品必屬佳作

最後一行
中文試試看 並且故意想辦法超過一行看看不做任何處理時 超過一行時
sed會怎麼樣處理這樣的問題

```
cyril@gyoza:~$ cat address.txt
台北市建國南路一段
270號
<Michele's Address>
<Cyril's Address>
台北市松江路
一段10號
<Nick's Address>

cyril@gyoza:~$ cat insert.sed
/<Cyril's Address>/i\
100 Gyoza Blvd\
San Jose, CA

cyril@gyoza:~$ sed -f inser.sed address.txt
台北市建國南路一段
270號
<Michele's Address>
100 Gyoza Blvd
San Jose, CA
<Cyril's Address>
台北市松江路
一段10號
<Nick's Address>
```

c\ change這個命令跟a\ i\ 很像，只有一點要注意， 就是如果要改變的文字跨很多行，則 adress是個range 1,10這樣時，所有的range行都會消失，並且只有一行改變了。

```
cyril@gyoza:~$ cat last
第一行  first line
第二行  second line
最後一行  last line

cyril@gyoza:~$ cat change.sed
1,2c\
第一行跟第二行都被幹掉了

cyril@gyoza:~$ sed -f change.sed last
第一行跟第二行都被幹掉了
最後一行  last line
```

餃子出品必屬佳作

小寫變大寫

把第一行到第十行中的小寫變大寫

```
$ sed '1,10y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/' lower.txt
```

代換的命令中有很重要的第二層處理，如果在一行內的字串吻合要代換的有很多 個，那麼內定會用第一個吻合的換掉，如果想換掉特定的就在後面加上想要的 第幾個，想全部換掉用g這個flag

```
cyril@gyoza:~$ cat last
第一行  first line
第二行  second line
最後一行  last line

cyril@gyoza:~$ sed 's/first/line/1' last
第一行  line line
第二行  second line
最後一行  last line

cyril@gyoza:~$ sed 's/first/line/2' last
第一行  first line
第二行  second line
最後一行  last line

cyril@gyoza:~$ cat lines
第一行  first line
第二行  second line
很多行  multiple line line
最後一行  last line

cyril@gyoza:~$ sed 's/line/row/2' lines
第一行  first line
第二行  second line
很多行  multiple line row
最後一行  last line

cyril@gyoza:~$ sed 's/line/row/g' lines
第一行  first row
第二行  second row
很多行  multiple row row
最後一行  last row
```

4.4.2 pattern/hold space的處理

- l list pattern space的東西是甚麼
- h 把hold space裡的東西清掉，把pattern space的東西copy給hold space

餃子出品必屬佳作

- H 把pattern space的東西加在hold space東西後面
- g 把pattern space裡的東西清掉，把hold space東西拿回給pattern space
- G 把hold space的東西加在pattern space東西後面
- p 印出pattern space的東西
- x 交換(exchange)pattern space與hold space

l跟p的差別在對非ASCII的處理

```
cyril@gyoza:~$ cat last
第一行  first line
第二行  second line
最後一行  last line

cyril@gyoza:~$ sed '1,2p' last
第一行  first line
第一行  first line
第二行  second line
第二行  second line
最後一行  last line

cyril@gyoza:~$ sed '1,2l' last
\262\304\244@\246\346  first line$
第一行  first line
\262\304\244G\246\346  second line$
第二行  second line
最後一行  last line
```

這邊可以對pattern space有更深的了解，第一行進到pattern space 看到p把第一行印出，接著pattern space變成output，所以sed又把 他印一遍，接著第二行進來了，在pattern space裡，看到p把第二行印出， 同理又有兩個第二行被印出，第三行由於p沒作用，直接是pattern space 送到output。再看一個例子(抄出來的)

```
cyril@gyoza:~$ cat command.txt
This describes the Linux ls command.
This describes the Linux cp command.

cyril@gyoza:~$ cat command.sed
/Linux/{
h
s/.* Linux \(.*\) .*/\1:/
p
x
}
```

餃子出品必屬佳作

```
cyril@gyoza:~$ sed -f command.sed command.txt
ls:
This describes the Linux ls command.
cp:
This describes the Linux cp command.
```

或者寫成一行

```
cyril@gyoza:~$ sed '/Linux/{h;s/.* Linux \(.*\) .*/\1:/;p;x;}' command.txt
```

這邊看第一個h把第一行推進hold space，不過記住pattern space還保有 原本第一行的句子。然後代換掉變成ls:，現在pattern space只有ls:， 然後p印出ls:，最後把hold space跟ls:交換，所以最後pattern space內 就是原本的東西，被sed當成output送出，cp是同樣的道理。

所以沒什麼，只是兩個眼睛看不到的space在那邊換來換去，把內容在這兩個 地方加加減減copy/paste就好了。如果熟悉vi的人應該很容易體會。

4.4.3 再進一步

多行處理與迴圈

在原本的h, H, g, G, x裡面都只有對單一行處理，如果pattern match要跨行時，就捉襟見肘。下面的命令用來處理兩行以上的pattern match及pattern space, hold space的處理

- n 把pattern space東西送出，讀入下一行，所以下一行變成pattern space中的東西。
- N 讀入下一行跟目前pattern space的 東西連起來變一行，但是裡面有個embedded換行符號。可以用 \n來表示，記住，只有這時才可以用\n來表示。
- D 幹掉到第一embedded newline部份，與N合用喔。 如果D在最底下被執行了，則pattern space東西保留下來， 迴圈到script頂端再來一次。
- P print到有embedded newline的一行，保留所有pattern space東西， 如果P在最底下被執行了，則pattern space東西保留下來， ，迴圈到script頂端再來一次。

Join兩行，

```
/join/ {
N
s/\n/ /g
}
```

把多行空白行幹掉成一行空白，

餃子出品必屬佳作

```
/^$/ {
N
/^\n$/D
}
```

請看，找到空白行後，把下一行加進來，如果只有一行空白，空白行與 下一行併成一行保留在pattern space，不過這時下個命令 /^\n$/D沒作用，pattern space全部送出， 如果有兩行空白，所以Delete掉第一行空白，剩下一行空白，保留下來， 這時因爲D命另有作用了，所以並不送出pattern space的東西， 繼續/^$/，一直會有一行空白行留下來。這邊D與P都有迴圈的作用 ，可以拿來作重複的處理。

flow control branch

這是用來轉移命令執行的就像是c裡的if else, while這些控制命令 這要搭配:一起用。由:設定一個label，然後用b label跳到這個地方執行。

- : label, 設定label
- b label branch 換到label地方執行，如果沒有label就通通不執行 直接跳到script尾巴。
- t label 如果一個代換成功，test 測試條件，成立後跳到label， 所以必須與s///命令一起用，s命令一定在t命令前。

多說不如看例子

```
$ cat join.txt
join
second line
在C語言中反斜線代表字串的延續 \
這一行跟上一行是連在一塊的

$ sed '{
:a
N
s/\\\n//
t a}' join.txt
join
second line
在C語言中反斜線代表字串的延續 這一行跟上一行是連在一塊的
```

多檔處理

- r 讀檔案 將檔案append到目前pattern space中
- w 寫檔案 將pattern space中的東西寫出到一個檔案

餃子出品必屬佳作

```
$ cat insert.sed
/<Cyril's Address>/i\
100 Gyoza Blvd\
San Jose, CA

$ cat lines
第一行  first line
第二行  second line
很多行  multiple line line
最後一行  last line

$ sed '/first/r insert.sed' lines
第一行  first line
/<Cyril's Address>/i\
100 Gyoza Blvd\
San Jose, CA
第二行  second line
很多行  multiple line line
最後一行  last lin

$ sed '/first/w firstline.txt' lines
第一行  first line
第二行  second line
很多行  multiple line line
最後一行  last line

$ cat firstline.txt
第一行  first line
```

其他

- = 印出行號
- q 離開sed

```
$ cat lines
第一行  first line
第二行  second line
很多行  multiple line line
最後一行  last line

$ sed -e '/first/=' -e '/second/q' lines
1
第一行  first line
第二行  second line
```

餃子出品必屬佳作

有很多的例子請看http://www.student.northpark.edu/pemente/sed/sedfaq.html

Chapter 5

awk

5.1 簡介

field的觀念是awk比sed好用的地方，相對於sed一行一行的處理，awk提供了 一行內部欄位拆解的處理，還有更powerful的像C程式語法迴圈控制處理。 通式可以是

awk 'script' var=value files
awk -f scriptfile var=value files

其中var是自定的awk變數，value是給定的值。這可以用來跟shell變數溝通。

另外像sed的regular express都拿一行行來作內定的輸入，再做比對，這樣很沒有 彈性，awk允許用awk的一個變數來作regular express的輸入比對，因此像欄位也 可以輕鬆的拿來比對。主要是用比對算符 來跟一個regular express比對，

var ~ /re/
var !~ /re/

這樣就很方便了，而且這比對也同樣會傳回true, false，可以當成一個condition。

除了欄位處理以外awk還有更多好用的function可以呼叫，另外也有了if,while, for與自定function的能力，這些比sed跟shell合併的能力又大上了許多，也是將來 perl的基本功能的基礎。

5.2 script基礎

一個script由 `condition{procedure},condition{procedure},.....`組成。 跟sed很像，找到什麼條件(pattern部份)，做什麼動作(procedure部份)， 有一些內部的變數可以幫我們設定一些條件。condition可以是pattern regular express或某個符合條件，這個請看邏輯運算符號與比對算符 。其中有兩個重要 的內

定condition, BEGIN跟END。一個script通常是這樣

BEGIN {xxx} /re/{xxx} END{xxx}

一般寫程式，一定要有init初始化一些值，就像人有出生年月日八字一樣，程式結束也要有處理善後的routine來處理。awk作爲一個程式型態的script語言， 提供了一個general的初始與結束的處理。

```
算算有幾個空白行
$ awk 'BEGIN{x=0} /^$/{x++} END{print x}' scripts.tex
```

一開始設定x=0後，BEGIN後的就不在執行，然後每讀一行進來，如果是空白行就 x++，最後讀完到檔尾，印出x。

要記住在上面的例子裡，awk的變數跟c是一樣的也就是沒有錢符號的，這邊跟shell perl等script比較不一樣。

5.3 變數與欄位處理

awk裡面很重要的兩個定義

- record 就是一行一行的資料
- field 通常由space或者tab鍵隔開的資料欄就是field

請看例子，並注意shell command line的單引號雙引號

```
$ cat /etc/mnttab
/proc   /proc   proc    dev=31c0000     1022606134
fd      /dev/fd fd      rw,suid,dev=32c0000     1022606198
mnttab  /etc/mnttab     mntfs   dev=3380000     1022606209
swap    /var/run        tmpfs   dev=1   1022606209
swap    /tmp    tmpfs   dev=2   1022606211

$ awk '/swap/{print $2}' /etc/mnttab
/var/run
/tmp

$ awk '{print "Hello World"}' /etc/mnttab
Hello World
Hello World
Hello World
Hello World
Hello World

一二欄位中間是tab二三欄位中間是空白
```

餃子出品必屬佳作

```
$ cat datafile
99 98 3.5
300 298 4.9
498 493 5.9
699 698 7.6
900 748 9.0
1200 703 9.6
1500 651 10.4
1698 627 10.8

記住  這樣是從input兩個欄位輸出output一個欄位
$ awk '{print $1 $2}' datafile
9998
300298
498493
699698
900748
1200703
1500651
1698627

這樣是從input兩個欄位輸出output兩個欄位
用print輸出欄位由逗號 , 分開
$ awk '{print $1,$2}' datafile
99 98
300 298
498 493
699 698
900 748
1200 703
1500 651
1698 627

這樣是從input兩個欄位輸出output一個欄位
$ awk '{print $1 "," $2}' datafile
99,98
300,298
498,493
699,698
900,748
1200,703
1500,651
1698,627

也就是說print $1$2 等於print $1 $2
```

餃子出品必屬佳作

5.3.1 內定變數

除了像C語法的自訂變數，x,y,z....。awk有一些內定的變數名。 欄位上的內定變數

```
ARGC        :  number of arguments on command line
ARGV        :  array containing the command line arguments
ARGIND      :
$0          :  整行record
$1....$n    :  第n個欄位(field)

ENVIRON     :  一個associate array含有全部的環境變數
ERRNO       :  最後system出錯的error number請看/usr/include/errno.h

FILENAME    :  current filename
FIELDWIDTHS :  list of field widths

FS          :  field separator內定是空白或tab
OFS         :  output field seperator內定是空白
RS          :  record separator內定是newline
ORS         :  output record seperator內定是newline
NF          :  number of field
NR          :  number of record
```

其中FS, OFS, RS, ORS, NF, NR是常用到的請看例子

```
$ cat /etc/passwd
list:x:38:38:SmartList:/var/list:/bin/sh
irc:x:39:39:ircd:/var:/bin/sh
nobody:x:65534:65534:nobody:/home:/bin/sh
cyril:x:1000:1000:Cyril Huang,,,:/home/cyril:/bin/bash

$ awk '{FS=":"; print $7}' /etc/passwd
/bin/sh
/bin/sh
/bin/sh
/bin/bash

$ awk '{FS=":"; OFS=","; print $1,$6,$7}' /etc/passwd
list,/var/list,/bin/sh
irc,/var,/bin/sh
nobody,/home,/bin/sh
cyril,/home/cyril,/bin/bash
```

餃子出品必屬佳作

NF NR可以拿來作condition用
$1 $2....也可以拿來作condition用，搭配變數regular express match算符 ~

```
$ cat datafile
# This is the data file from experiment
99      98 3.5
300     298 4.9
498     493 5.9
# Failed data
122     123
699     698 7.6
900     748 9.0
1200    703 9.6
1500    651 10.4
1698    627 10.8

$ awk 'NF == 3 && $1 ~ /[0-9]+/{print $3,$1,$2}' datafile
3.5 99 98
4.9 300 298
5.9 498 493
7.6 699 698
9.0 900 748
9.6 1200 703
10.4 1500 651
10.8 1698 627
```

5.3.2 awk變數與shell變數

awk 與 sed都是幫助shell programming的好工具，因此要跟shell變數溝通， 通式中的 var=value，var就是awk變數，value就是shell變數的值。

```
# fs_mounted $dev $mnt_pt $fs_type
# This is shell script version of fs_mounted, check if the file system is
# mounted.
fs_mounted()
{
    _BINGO=
    [ $# -eq 3 ] || return $ERRNO_EINVAL
    _BINGO="`$AWK '{if (\$1 == _DEVICE && \$2 == _MNT_PT && \$3 == _FS_TYPE) print \$1}' _DEV
    [ "$_BINGO" ] || return -1
    return 0;
}
```

餃子出品必屬佳作

shell script的執行與shell command中的awk對於一些保留字元處理跟sed一樣要小心點 這個例子裡面注意$1 $2 $3，在AWK ' '單引號裡面的是awk的$1$2$3，在 外面_DEVICE=$1...是shell的 $1 $2 $3。_DEVICE _MNT_PT 等是awk的 變數。所以awk的$1 $2 $3有\這個escape在前面。也就是說shell 不處理它們並把這些送給awk處理。

5.3.3 陣列Array與split函數

所有awk的array都是associate array,也就是hash，也就是帶有名字id的陣列。這個名字id就是hash的key。一般我們用1, 2 ,3 ...來當作陣列的id註標，在awk裡面允許我們用名稱來當id index來搜尋陣列。不過awk很聰明如果看到是數 字型的index也可以像在C裡面用的一樣。

注意!陣列從array[1]開始，不是從array[0]開始，跟c不一樣。 我們可以指定x[1], x[2]...的陣列照1,2,3 ...排列順序， 但associate array的排列不是我們可以控制的。注意associate array的注標要用 引號括起來，請看for loop的用法。

可以用split(string, array, [sep])來得到一個陣列，sep沒有給就用FS的值當做這個string的欄位分隔，一個欄位就是一個陣列元素。

```
$ cat split.awk
#! /bin/awk
BEGIN{FS=":"}
/cyril/{
    print $0
    split($0, afield)
    for (i = 0; i < NF; ++i) {
        print i","afield[i]
    }

    delete afield[2]

    for (i = 0; i <= NF; ++i) {
        print i","afield[i]
    }
}

$ awk -f split.awk /etc/passwd
lcyril:x:100:1::/export/lcyril:/usr/bin/bash
0,
1,lcyril
2,x
3,100
4,1
5,
```

餃子出品必屬佳作

```
6,/export/lcyril
7,/usr/bin/bash
0,
1,lcyril
2,
3,100
4,1
5,
6,/export/lcyril
7,/usr/bin/bash
```

5.4 基本控制語法

除了欄位的變數與特定用法外，awk比sed強的在於更接近程式語言的控制方法。 awk的語法很像c，說實在awk蠻簡單的，有c的基礎很容易就會了，當然也可以先學 awk再學c。

5.4.1 數學與邏輯算符

C的一般數學

```
+ - * / % ++ -- ^ **(這是fortran的指數)
```

C的一般指定

```
= += -= *= /= %= ^= **=
```

C的一般邏輯

```
! == < > <= >= != && ||
?: (c的cond ? result1 : result2 如果cond成立返回result1不然返回result2)
```

condition的邏輯算符裡面有個比較特別的就是pattern match算符 ~, !~ ， 將來perl中很有用的類似算符~=。主要是regular express通常match的是 一行行作爲內定的比對，如果想要對某個特定變數做re match時，就要用到這個算 符。

```
如果x是個阿拉伯數字
if (x ~ /^[0-9]+/) {
     x += y
}
```

餃子出品必屬佳作

5.4.2 if

沒什麼了不起就是c而已

```
if (condition) {
} else if (condition) {
} else {
}
```

注意else if跟c語法一樣

5.4.3 for loop

這邊的for有兩個，是c與bourne shell的綜合，perl也是一樣，不過perl用了 c shell的語法。

```
for (i = 0; i < 10; i++) {
}
for x in xarray {
}
```

請看例子

```
$ cat gyoza.awk
#! /bin/awk  heehee
BEGIN {
x[1]="I"
x[2]="am"
x[3]="a"
x[4]="gyoza"
y["I"]="I"
y["am"]="am"
y["a"]="a"
y["gyoza"]="gyoza"
for (i = 1; i < 5; i++) {
     print x[i]
}
for item in x {
     print item, x[item]
}
for item in y {
     print item, y[item]
}
}
```

餃子出品必屬佳作

```
後面的gyoza.awk沒有用啦  我只是跑這個BEGIN而已
$ awk -f gyoza.awk gyoza.awk
I
am
a
gyoza
2 am
3 a
4 gyoza
1 I
gyoza gyoza
am am
I I
a a
```

5.4.4 while

沒什麼了不起就是c而已

```
while (condition) {
    statment
}

do {
    statment
} while (condition)
```

for loop與while loop都跟c一樣，可以用break離開，continue不做這loop換到下個 loop。

5.5 function

跟c不太一樣的，這是一種script，不用甚麼type要宣告，也沒有什麼 pass by value, pass by address，變數是global的。

```
# 通式
function fun(arg1, arg2) {
        arg1 = arg2
        xxxx
}

排序的例子
$ cat sort.awk
```

餃子出品必屬佳作

```
# sorting function
function sort(Array, elements, temp, i,j)
    for (i = 2; i<= elements; ++i) {
        for(j = i; array[j - 1] > array[j]; --j) {
            temp = array[j]
            array[j] = array[j - 1]
            array[j - 1] = temp
        }
    }
    return
}
# main routine, 一行一行來
{
for (i = 2; i <= NF; ++i) {
     grades[i - 1] = $i
}
sort(grades, NF-1)
printf("%s: ", $1)
for (j = 1; j <= NF - 1; ++j) {
    printf("%d ", grades[j])
}
printf("\n")
}

$ cat grade.txt
西西  100 60 75 23
美美  100 98 99 89

$ awk -f sort.awk grade.txt
西西: 23 60 75 100
美美: 89 98 99 100
```

5.6 其他函數

5.6.1 字串處理-match與代換

字串上處理不像sed有s///這種代換，得用上一些內建函數。

- 傳回符合regex的在string的位置
  match(string, regex)
- 傳回子字串: 傳回在第m個字元往後數n個字元的子字串，n沒給就到底
  substr(string, m, [,n])

代換: 代換的幾個函數有的傳回值很特別

餃子出品必屬佳作

- 把string2裡面符合regex的第一個字串換成string1。成功傳回1。
  sub(regex, string1, string2)
- 把string2裡面符合regex的字串globally的全部代換成string1， 等於sed的s///g。
  傳回整數表示代換數目。
  gsub(regex, string1, string2)
- 傳回一個字串爲string2裡面符合regex的第n個字串換成string1， 等於sed的s///nth_match。
  但是string2沒有被改變。
  gensub(regex, string1, nth_match, string2)

string2沒給通通是$0，nth_match可以用g或G表示globally(全部)。請看例子

```
$ cat match.awk
#! /bin/awk
BEGIN{FS=":"}
/cyril/{
print $0
print match($0, /cyril/)
print substr($0, match($0, /cyril/), length("cyril"))
print gensub(/cyril/, "mark", 2)
print $0
sub(/^.*cyril/, "mark")
print $0
}

$ awk -f match.awk /etc/passwd
lcyril:x:100:1::/export/lcyril:/usr/bin/bash
2
cyril
lcyril:x:100:1::/export/lmark:/usr/bin/bash
lcyril:x:100:1::/export/lcyril:/usr/bin/bash
mark:/usr/bin/bash
```

這個例子用了length()這個函數，regex不用引號，有string的地方要用 double quote括起來，用gensub不會改變string2的值， 而且請看第三個print $0，這邊還是有greedy的效應在，所以只剩下後面一截。

5.6.2 輸入輸出處理print(f)

輸出上來說print是很簡單的用法，不過有更好的format輸出，就像c裡的printf或者fortran一樣

```
printf()

用法跟c函數一樣
{printf("The sum on line %s is %d.\n", NR, $1+$2)}
```

餃子出品必屬佳作

```
%s     表示一個字串
%d     表示一個整數
%n.mf 表示一個n個整數m個小數的浮點數
```

5.6.3 亂數rand()

產生介於0 到 1之間的亂數

```
BEGIN {
srand(systime())
random_num = rand()
print random_num
}
```

systime()傳回從1970, 1, 1到現在的秒數 srand()產生一個虛擬的亂數陣列以供rand()使用

5.6.4 系統system()

這個很像Unix的system()這個函數

```
BEGIN {if (system("ls -l")) != 0) {
           print "command failed"
       }
}
```

其他的像數學函數等等就不加以介紹了

Chapter 6

perl

相對於sed, awk, shell script等工具，當初Larry Wall覺得要發明一種可以取代 一些雜七雜八script泛通用型的script工具，比起所有的script還要power，處理起regular express也更方便，解決了一些如greedy的問題。perl最後跟 system call的連結與檔案,process，socket program等的處理，讓他變成很強大的script語言。

perl跟其他不管shell, TCL/expect...等script語言其實還是用C寫出來的， 每一種script其實都還是一個執行程式，只不過這個執行程式懂得一些內定語法，會去解釋這些規則然後轉換這些script變數，控制迴圈等等變成自己的c變數跟迴圈 （當然啦!其實最後還是都是assembly的變數跟迴圈）來執行。就跟shell, awk一 樣。

6.1 基本語法與資料型態

6.1.1 變數

不管設定取值都要加個錢符號

6.1.2 條件控制

6.2 更power的regular expression

6.3 常用函數

6.4 系統處理

6.4.1 process

6.4.2 檔案

6.4.3 system call

6.5 module

Chapter 7

Bourne shell, perl與C語法比較

這個其實是我年紀大了有時記不住用的。C, shell 與perl是我常用到的工具，現在對這3種工具語法做個比較。

7.1 變數

定義變數

```
shell
var=

perl
$var =

C
int var =
char var =

Makefile
var =
var :=
```

使用變數

```
shell
$var
${var}

awk
var
```

```
perl
$var

C
var

Makefile
${var} or $(var)
```

要注意的是

- shell的定義不加$ 但要知道變數值時一定要加$ perl就比較一致 不管如何就是加$符號 C就是都不加
- shell的等號前後不可有空白
  var= var = 是不一樣的。 perl與C就沒有限制，Makefile是定義與使用跟shell 一樣，但是等號前後可以有空白，所以有四種情形請默記一下。

7.2 陣列與Hash

shell

```
array="e1 e2 e3"
```

perl

```
@array = {e1, e2 ,e2};
%hash = {key1 => val1,
  key2 => val2,
  key3 => val3};
```

C

```
int array[] = {1, 2, 3};
```

shell 的串列型資料叫list裡面元素用space分開，最常用的場合

```
for var in $array
do
cmd1
done
```

array中的元素就是用space分開的資料。 perl有個很像的foreach

```
foreach $var (@array) {
xxx
}
```

其實這是C shell的寫法。 C沒有像這樣一個一個自動餵array的元素給變數的機制。

餃子出品必屬佳作

7.3 條件敘述

shell

```
if test; then
dosomething
elif test; then
doother
else
allright
fi
```

perl

```
if (c) {
do_something;

} elsif () {
do_other;
} else {
allright;
}
```

C

```
if (c) {
do_something;
} else if () {
do_other;
} else {
allright;
}
```

這邊要注意的是shell不需要括號，else if的寫法三個不一樣。條件式寫法，shell沒有一定要用()包住條件式，只要test是某個可執行的敘述就好

7.3.1 條件比較

shell

字串

```
[ "str1" = "str2" ]
```

數值

```
[ num1 -ne num2 ]
```

perl

字串

```
if ($str1 ne $str2)
```

數值

```
if ($num1 == $num2)
```

餃子出品必屬佳作

C

字串
```
if (strcmp(str1, str2))
```
數值
```
if (num1 == num2)
```

要注意數值比較，字串比較，字串處理與邏輯比較。 由這邊我們可以看出其實perl是非常有彈性與強大的。

7.4 迴圈控制

7.4.1 while迴圈

shell

```
while test
do
cmd
done
```

perl跟c是一樣的，除了變數用法不一樣

```
while (condition) {
statment1;
}

unless (condition) {
statment1;
}
```

C

```
while (condition) {
statment1;
}

do {
statment1;
} while (condition)
```

7.4.2 for loop

Bourne shell的for不是像C裡的for loop

```
for var in "element1 element2 element3..."
do
echo $var
done
```

餃子出品必屬佳作

perl 有兩種for的用法

```
for ($i = 0; $i < 10; $i++) {
statement
}
foreach $var (@array) {
statement;
}
```

C的用法

```
for (i = 0, init = 1; i < 10; i++) {
statement;
}
```

7.5 副程式

shell

```
func1()
{
arg1=$1
arg2=$2
}
```

awk

```
function func1(arg1, arg2)
{
xxx
}
```

perl

```
sub func1(arg1, arg2)
{
xxx
}
```

C

```
int func1(int arg1, char *arg2)
{
xxx
}
```

餃子出品必屬佳作

7.6 註解

shell

```
# comment
```

perl

```
# comment
```

C

```
/* comment */
```

Bibliography

[1] Linux In a Nutshell, Ellen Siever, O'Reilly & Associates, ISBN 0596000251

[2] Sek & Awk, Dale Dougherty, et al, O'Reilly & Associates, ISBN 1565922255

[3] Perl Cookbook, Tom Christiansen, Nathan Torkington O'Reilly & Associate, ISBN 1565922433

[4] The C Programming Language, Brian W. Kernighan, Dennis M. Ritchie, Prentice Hall, ISBN 0131103628