C Shell
使用簡介

(1.0b 版　2002/05/06)

(作者：網路農夫)

獻給

指導過我的師友

前言

這份文件轉自於過去斷斷續續編寫的 UNIX C shell 網頁。寫作本文時，使用的作業系統是 SunOS 4.1.3，距今已將近是六、七年前的事了。僅有少部份在回應網友詢問後略作修改並在 Solaris 8 for Intel x86 環境下驗證。

由於這最近(2002)的網頁改版，我也決定將原先 UNIX C Shell 的文字僅保留基本的運用與環境設定兩個章節。其他去除的章節，日後將另做整理。

如果您發現文件中有任何謬誤，請務必來信通知我修改。

網路農夫 2002/05/06
farmer@www.europa.idv.tw

目錄

C Shell 的指令運用

在 UNIX 系統中，指令均以檔案的方式分類存放在檔案系統的數個目錄中。而 C Shell 也其中的一個“指令”。但它的功能卻不止於此，指令們必須透過它所設定的搜尋路徑（path）來執行，變數 path 設定錯誤或指令不在 path 支援的路徑之內，指令是無法執行的。除此外亦可設定別名（aliases）來取代慣用的某個甚至數個特定的、一連串的指令，以免除指令太長或複雜難記之痛苦。但在這些複雜的功能未介紹之前先來看看光是指令在運用上的變化。

單一指令（single command）

就整體而言 UNIX 系統的指令的用法通常都有一個固定的架構。指令使用語法的第一項是指令名稱，然後接著是指令的選項（options）。有時選項後需加上所需的參數。指令的選項大份均可組合或共同連續使用，功能有時相輔相成；當然也有些選項會有不能混用的情況，但比較少。再來多半便是指令所需的引數，如檔案名稱（filenames）或目錄（directory）。此部份則有 C Shell 所提供的“wildcard”特殊功能可運用。在 UNIX 系統指令群的功能上則是複雜多變應有盡有強大無比。甚至某些指令的功能發展到可以寫一本專書且欲罷不能（如“sed” “awk”就是），這類指令在指令行模式下雖然很少用到，但卻常在撰寫 shell 程式發現它們的蹤影，所以說對這些功能強大的工具亦應有所瞭解。在需要運用時才不致於有心無力。

在 UNIX 系統的指令中，關於使用者層次的就有二百多個指令，如果將管理階層所使用的指令加進來，那更是有夠驚人。有鑑於此，誠心地希望讀者能養成看 on line manual 的習慣。並且建立自己的指令歸類。因爲你對指令的熟悉度將會影響撰寫一個 Shell 程式的成功機率，亦或是 Shell 程式的執行效率。

以下由簡而繁舉例說明幾個常用指令：

1. ls
2. ps -axu
3. cp -R ~/akbin akbin.bak
4. find / -type f -name "mem*.c" -print

連續指令（multiple commands）

在 C shell 中符號“；”有它特殊作用。當你想在同一行連續下指令時，便可在第一個指令結束後用符號“；”來連接下一個指令。如此可一直連接下去。連續指令的使用格式見下：

command1 ； command2 ； command3；.....

以下我們來看幾個實際的例子

```
1% cd ~/backup ; mkdir startup ; cp ~/.* startup/.
2% cd ~/akbin ; tar cvf /dev/rst8 *.* >& tar.tmp ; rm -r *.* ; logout
```

在事件 1 我們改變工作目錄並緊接著建立一個新目錄然後將要備份的檔案拷貝到新的目錄中。事件 2，我們先改變到要備份的工作目錄然後下備份指令並接著在備份完畢後將檔案清除，然後退出系統 logout。以這種方式來連續執行一連串的指令，可讓你一口氣下完一整個流程的工作，在當你有事想離開時，想要的整個流程作業一次完整下完，相當便利。在實際的應用上，常見到他運用在別名的設定上，關於這點我們將在別名中再爲你說明。

群體指令（commands group）

“群體指令”的作用看起來很像“連續指令”，兩者均可將一長串指令加以執行。但就執行“環境”而言，兩者卻有很大的差異。連續指令在執行時並不先產生一個 subshell 來執行指令。故工作環境會隨著指令的執行而產生變動（如上例連續指令執行完畢後工作目錄便已改變）。但群體指令並不會產生上述的情況。在執行完指令後它會在保留下指令時的工作環境，不會有任何改變。見下例：

```
% pwd
/home1/akira
% (cd ~/akbin;tar cvf /dev/rst8 *.*)
% pwd
/home1/akira
```

這是因爲群體指令的執行是先自動產生一個 subshell，再將所有指令交由這個 subshell 來執行。當這些指令雖在執行過程中改變了環境變數，但僅是改變 subshell 的環境罷了，一旦指令執行完畢後，subshell 便會自動結束並回到原來的 shell 中，因 subshell 的變數無法影響 parent shell，所以原執行的環境將不會改變。

條件式的指令執行（conditional command exection）

在 C Shell 中支援兩個特殊符號“||”、“&&”，用來幫助你做簡單的指令執行控制。它是運用一個特殊符號來連接兩個指令，以第一個指令執行狀態的成功與否來決定是不是要繼續執行第二個指令。讓我們來列表說明它的用法：

第一個指令執行狀態	運用符號	第二個指令反應狀態
執行成功	||	不執行
失敗	||	執行
執行成功	&&	執行
失敗	&&	不執行

特殊符號“||”相當於 else、而特殊符號“&&”則相當於 than。這種相當特殊的指令控制，或許

在正常的指令行模式下難得用上，甚至有可能根本用不上。但在 C Shell 程式設計上可是一項相當重要的功能呢。我們來兩個實際應用的例子：

```
% grep error ecs.sum || lpr ecs.sum
```

當指令 grep 如果在檔案“ecs.sum”中有找到字串“error”，則不執行列印。如果沒找到任何“error”則執行列印檔案“ecs.sum”。

```
% ps axu | grep rpc.pcnfsd | grep -v grep > /dev/null && wall pcnfs.run
```

上面這個例子第一個指令的部份比較複雜，整個指令的功能是找尋系統的處理程序中是否有“rpc.pcnfsd”在執行，如果有則執行第二個指令 wall 向上機的使用者說明，說明的文字內容便是檔案“pcnfs.run”。如果找尋不到，則不執行第二個指令 wall。

相信讀者可能會質疑有可能下這樣複雜的指令嗎？老實說在指令行模式下真得相當少。但這項功能在 C Shell 程式設計上則會運用得上，千萬別忽視。切記！切記！

引號的運用與指令的關係

在 UNIX 系統中如果在指令、檔名等運用，或者是在變數的設定上，碰到了空白（space）或 TAB 字元夾雜在當中時，如果你還用一般的方式來處理，則往往會造成一些不必要的指令語法錯誤或者是設定上的 bug。如果你實際的運用上也常碰上這種困擾的話，請不用擔心，仔細看以下這些符號的運用，相信你所需要得答案與運用法則便在其中。

C shell 對於這方面的處理上，提供了三個功能相近的符號（'"`）來做字串的處理。這三個符號必須以“成對的”方式使用才有效用。換句話說，就是用這些符號來『括住』我們所要處理的字串。當我們用它們來括住待處理的字串時，使用不同的符號便會有不同的結果。但有時會因功能相近且符號也相近的情況下，常常會讓使用者分辨不清，使用上請小心注意符號本身所代表的特性。

就它們整體的使用面來看，可區分成指令、檔名與變數這三個方面，以下我們先就前兩項來說明之，而關於變數方面，將在下一章中再加以討論。

單引號（ ' ）的運用（single-quotes）

當我們以單引號來來括住數個以 space 字元所分格開來的字串時，最主要的目地便是將它們便成是一個單一的字串來運用。譬如說，在 UNIX 系統中，檔案名稱是允許使用 space 字元在當中的。像“data a1”這樣的檔名是合法的。但如果我們想將某個指令的輸出重導成“data a1”檔案，用一般的方式一定會產生錯誤的結果，如下：

```
8 % ls -l > data a1
a1 not found
```

像以上的語法使用“data a1”因爲有 space 字元的分格而被當成是兩個獨立的字串，所以該指令的“data”與“a1”分別被 C shell 解譯成，“a1”是指令 ls -l 的檔案引數；“data”則是重導向的檔案名稱。如果該目錄下有檔案“a1”存在的話，該指令將不會有錯誤訊息傳出。而是成功地將“ls -l a1”的資料輸出重導到檔案“data”中。這指令的作用實際上已相差了十萬八千里。如果是使用在 C shell 程式設計中，這在 debug 上，可真的會累死人。當然像這種檔名上的特殊問題，一般是不會這樣做的。它的用義是想告訴讀者含有 space 字元的字串，在實際上是與一般有所差別的。一但使用觀念錯誤，可能會有相當麻煩的情況產生。這一點請多加注意。接下來我們再來看一個比較常碰到的情況。

當我們想到檔案“find.data”中，找出含有“Permission denied”的每一行時，如果我們用下面的方式來下指令：

```
9 % grep Permission denied find.data > datafile
```

實際上你所得到的結果將不是含有“Permission denied”的字串，而是僅含有“Permission”字串的每一行。而且在指令中的“denied”字串將會被 C shell 解譯爲是指令 grep 的檔案引數，如果該目錄下沒有“denied”這個檔案，將會有像下面這樣的錯誤訊息輸出：

```
grep: denied: No such file or directory
```

要想正確地得到你所要的結果，以上的兩個指令可使用單引數來括住字串，便可解決上述的種種問題。如下：

```
10 % ls -l > 'data aa'
11 % grep 'Permission denied' find.data > datafile
```

另外我們也時常運用單引號在設定較複雜的別名上。如下：

```
alias psg 'ps axu | grep \!:1 | grep -v grep'
```

像如此的整個指令串的部份，必須用單引號來將它們括住，要不然在語法上會產生錯誤。當然在單引號之內，語法上也允許再使用像雙引號那樣的符號在裡面。如下面這個別名的設定的例子：

```
alias lsd ' ls -l | grep "^d" '
```

除此別名的設定之外，像指令 echo 也常會運用得上，如下面的情況：

```
12 % echo ID Title Name Note:
ID Title Name Note:
```

如果你使用 echo 指令要輸出一個帶有 TAB 字元的訊息。如上例你會發現，原本設定好的格式，在輸出時完全變了。這是因爲 C shell 處理指令後的引數，並不認識 TAB 字元，僅僅會讀入引數本身。輸出結果時各個引數之間則用一個 space 字元來加以區格開來。所以造成像上面這樣的結果。如果你使用單引號括住，情況便會有所改變。請看下面的例子：

```
12 % echo 'ID Title Name Note::'
ID Title Name Note::
```

因爲有單引號括住的原因，使得四個引數及它們之間的 TAB 字元，合爲一體，形成爲同一個引數。指令的輸出，便因此而保留了 TAB 字元，輸出結果才會與原來的格式相同。另外一點值得一提的是，變數的符號“$”在單引號內是無法發揮變數作用的，它將僅僅是一個普通的字元，而無任何的特殊意義。如下所示：

```
3 % set d = date
4 % echo $d
date
5 % echo 'variable $d'
variable $d
```

關於此一特性，請特別牢記在心。因爲這也就是單引號有別於雙引號在運用上的最大差別處。

雙引號（ " ）的運用（double-quotes）

讓我們延續上面的問題，來看看改用雙引號的效果。

```
6 % echo "variable $d"
variable date
```

從輸出的結果可清楚地看到 $d 變數，在雙引號之內依然可發揮它變數的功能。其實不光是符號 “$” 有如此的差異，就連以下我們將要爲你介紹的倒引號 ` 與倒斜線 \ ，也均是如此。所以請讀者注意到，在雙引號內的特殊符號，如 $、\、倒引號等，其特殊功能均不會喪失。但如果將它們放入單引號中，則符號的特殊意義與功能會消失，而僅只是符號而已。

不過，不管是雙引號或者是單引號，它們對於 space 字元與 TAB 字元的處理形式上，其功能完全相同。如先前我們所提到的例子：

```
10 % ls -l > 'data aa'
11 % grep 'Permission denied' find.data > datafile
```

如果將單引號改成雙引號，其效果完全相同，兩者都會將它們變成是一個字串。

倒引號（ ` ）的運用（backquote）

倒引號的功能是讓我們去括住指令，此功能常用在 echo 指令的內容中，或者是一些設定變數的場合上。使用的方式相當簡單，如下所示：

```
`command`
```

讓我們先來以下面的例子來說明倒引號的功能。

```
3 % echo There are `who|wc -l` users logged on
There are 2 users logged on
```

`who|wc -l` 是被倒引號括住的一組指令，我們將它放在指令 echo 所要輸出的文字之中。在 C shell 在解譯執行整行指令時，會先執行 `who|wc -l`，然後將其結果傳回 echo 指令中，再由 echo 指令將整個結果輸出到螢幕上。

倒引號在使用上請小心，千萬別將它放在單引號之內，如下面這個例子：

```
4 % echo 'There are `who|wc -l` users logged on'
There are `who|wc -l` users logged on
5 % echo "There are `who|wc -l` users logged on"
```

```
There are 2 users logged on
```

當然倒引號在雙引號內還是有功用的，但請仔細比較指令 5 與指令 2 的輸出，在指令 5 的輸出有一大段空格，這是因爲倒引號所括住的指令的輸出代入指令所佔用的字元長度的結果。可別忘了雙引號是會保留原來格式的特性。而指令 2 的輸出會沒有多餘的空格，便是因爲在指令讀取引數時已經被乎略掉的結果。

倒引號在使用上，常常被運用在變數的設定上。有關於這一點，我們將會在下一個章節中爲你詳細介紹。

倒斜線“ \ ”的運用（backslash）

我們使用倒斜線的功能是擋去某些俱有特殊意義的字元，讓它們的特殊意義失效。譬如某個別名的設定、重導向的符號、變數的符號等等。讓我們來逐一舉例說明之。首先我們來看一下關於前面我們曾經說明過的別名設定：

```
7 % alias rm
rm -i
8 % \rm -r
```

指令 7 顯示了“rm”已被別名的功能設定成爲指令 rm -i，如果我們要使用“rm”，並且要避開別名的功能，你可使用倒斜線“ \ ”擋在“rm”之前，便能除去別名的設定，使用原系統指令“rm”了。

此外在指令 echo 的運用上，最常會運用到倒斜線。如要使用 echo 輸出一個特殊的符號時（像 >, <, \, *,...），如下：

```
9 echo >
Missing name for redirect.
10 % echo \>
>
```

在指令 9 中，很明顯的錯誤訊息告訴我們，“>”是個重導向的符號，所以指令執行失敗。在指令 10，我們以倒斜線來擋在符號“>”之前，便順利地輸出該符號“>”。這便是倒斜線的功用。當然如果你想要以指令 echo 來輸出“\”也是可以的，你只須用兩個連續的倒斜線便可。如下：

```
11 % echo \\
\
```

兩個連續的倒斜線意思便是，擋去倒斜線的特殊意義。在上例中，如果只用了一個倒斜線，這個在行尾的倒斜線所代表的將不是上面我們所提到過的意義了。行尾的倒斜線的功能則是連接下一行的意思。譬如當有下指令時相當長，想要以兩行的方式來完成它，則可以在第一行的行尾加上倒斜線，

然後便可在下一行繼續再見鍵入未完的指令。我們來看下面例子的用法就會明白了。

```
1 % echo "a NEWLINE preceded by a'\' (backslash) \
gives a true NEWLINE character."
a NEWLINE preceded by a'\' (backslash)
gives a true NEWLINE character.
2 %
```

輸入、輸出重導向 (I/O Redirection)

“輸入/輸出重導向”是 shell 用來處理標準輸入（standard input）、標準輸出（standard output）與標準錯誤（standard error）等訊息的重要功能。它可將指令執行的輸出導向到檔案中，亦可把執行程式或指令所需的引數或輸入由檔案導入。或把指令執行錯誤時所產生的錯誤訊息導向 /dev/null。其應用範圍可說相當廣範。

輸入重導向（Input Redirection）的運用

符號“<”－－重導向標準輸入

如右圖所示，通常一個程式或指令所須輸入的參數便稱爲標準輸入（standard input）。在導入的運用上，可用來導入檔案。語法如下：

使用語法 command < file

下面的例子，第一是將檔案“mail.file”重導入指令 mail 中，做爲傳送mail給使用者 akk 的內容。第二則是將檔案“file.data”重導入指令“wc -l”中，用以計算該檔案的行數、字數（word）與字元數。

```
1 % mail akk < mail.file
2 % wc < file.data
1 11 66
```

雖然以上重導入的使用方法是正確的，但對於一般的指令的運用來說實在是多此一舉，因爲像這類指令的語法均會支援檔案輸入的功能。但是重導入的運用卻依然不可輕意忽視，有許多軟體公司所發展的應用程式或自行開發的 shell 程式（一般稱爲 shell script），大部份多採用交談式的方式來要求使用者輸入所需的資料，像此類的交談資料，可預先做成一個檔案再以重導入的方式執行，不但省去交談的程序更可進一步將簡化作業程序。

符號“<<”－－字元的重導向

```
使用語法 command << word
word or data keyin
....
word
```

在上面我們所提到的重導向符號“<”它所能處理的對像是檔案，如果不是檔案便無法處理。而在這裡將爲你介紹的符號“<<”則是用來重導向文字使用的。我們來看它運用在指令行模式下的情況。

```
% mail akk << EOF!
testting I/O Redirection function
testting I/O Redirection function
testting I/O Redirection function
EOF!
%
```

在上面我們使用了符號 “<<” 來將我們想要傳 mail 給使用者 akk 的訊息一一由鍵盤鍵入。指令行最後的“EOF!”代表當我們見要鍵入內容的結尾用字－－我們可稱它爲“關鍵字”。當我們鍵入指令“mail akk << EOF!”後 return，便可開始鍵入想要傳送的訊息內容，如果想要結束僅需在新的一行鍵入“關鍵字”便可。

這個符號“<<”常被運用在檔案編輯的指令上，如指令 ed、ex。如下例子：

```
% ed sed1.f << ok
g/root/s/0/1/
g/akira/d
w sed2.f
q
ok
```

上例我們運用這種重導入的方式來編輯檔案“sed1.f”。第二行與第三行是指令 ed 的編輯指令，前者是找尋檔案內所有的“root”字串，並將該行的“0”代換爲“1”。後者則是找尋檔案內所有的“akira”字串，並將該行清除。第四行也是編輯指令，用意是將編輯的結果寫到另一個檔案“sed2.f”中。第五行則是退出指令 ed 編輯器。第六行便是重導入的“關鍵字”，告訴導入終止。

如果有機會試一試這種用法！使用它來編輯檔案，有時候比你進入 vi 編輯器做還快速。

輸出重導向（Output Redirection）的運用

在重導出的使用比較起重導入的運用複雜，也比較常使用到。通常指令的輸出訊號包含到標準輸出（standard output）與標準錯誤（standard error）二種。關於重導出所使用的符號也比較多，總共有 8 個符號。就功能的區別，大致上可歸類成兩組。如下表所示：

第一組	“>”	“>&”	“>>”	“>>&”
第二組	“>!”	“>&!”	“>>!”	“>>&!”

符號“>”與“>&” －－重導向標準輸出與標準錯誤

使用語法 command > file
command >& file

先讓我們來下一個指令 find，從“root”以下找尋檔名爲“Cshrc”，並將結果列出來。

```
7 % find / -name Cshrc -print
/usr/lib/Cshrc
find: cannot chdir to /var/spool/mqueue: Permission denied
/home1/akira/cshell/Cshrc
8 %
```

指令所輸出的訊息有三行，有兩行是指令找到我們指定檔名的資料輸出，這正是我們所要的。而這些輸出訊息的格式屬於標準輸出。但卻有一行訊息顯示“/var/spool/mqueue”目錄不允許我們進入，換句話說該目錄我們沒有讀取的權限。這行訊息會混在兩行標準輸出的訊息中是因爲指令 find 搜尋目錄的次序關係，並無特殊意義。但訊息本身的格式卻不屬於標準輸出，而是標準錯誤。這一點我們可用以下的指令來獲得證實：

```
9 % find / -name Cshrc -print > file.tmp
find: cannot chdir to /var/spool/mqueue: Permission denied
10 % cat file.tmp
/usr/lib/Cshrc
/home1/akira/cshell/Cshrc
```

當我們再一次執行同樣的 find 指令，但使用重導向符號“>”，將輸出導向到檔案“file.tmp”中。結果發現只剩下警告訊息依然出現在螢幕上。用指令 cat 去看檔案“file.tmp”證實需要的資料確實在檔案中。這證實了警告訊息是不同於一般的輸出訊息的。因爲它的模式是標準錯誤。其實就整體而言，在 C Shell 的環境中警告性的、錯誤性的輸出訊息均屬於此類－－標準錯誤。千萬別忘了。

如果說你想要將這了兩種輸出訊息一起重導向到檔案“file.tmp”中，可改用重導向符號“>&”。如下：

```
11 % find / -name Cshrc -print >& file.tmp
12 % cat file.tmp
/usr/lib/Cshrc
find: cannot chdir to /var/spool/mqueue: Permission denied
/home1/akira/cshell/Cshrc
```

情況便如我們所要的，所有訊息都在檔案 “file.tmp”中了。在這裡我們要告訴你一件事，其實在輸出重導向這一系列的符號中，“&”符號便是代表標準錯誤，只要是重導向符號中含有“&”，就表是可重導標準錯誤。如“>&” “>>&” “>&!” “>>&!”都是。

對於重導向符號“>”與“>&”在運用上有一個特性，那就是當你所指令的檔案實際上不存在時，重導向功能會幫你產生該檔案，並將資料寫入。但如果該檔案在執行前便以存在，則當指令執行重導向功能完後，該檔案原來的資料將會被重導入的資料所重寫。換句話說，便是原來的資料將會完全不見，且無法挽回。使用上請讀者務必注意到此一特點。我們來看下面的實際例子：

```
13 % cat file.tmp
/usr/lib/Cshrc
find: cannot chdir to /var/spool/mqueue: Permission denied
```

```
/home1/akira/cshell/Cshrc
14 % tail -1 /etc/passwd > file.tmp
15 % cat file.tmp
+::0:0:::
```

我們可清楚地看出，當指令“tail -1 /etc/passwd > file.tmp”執行完後，檔案“file.tmp”的內容已變成是“/etc/passwd”檔案的最後一行。而原來的資料以不見了。

在重導向的運用上有一項限制，請先看到下面的例子：

```
% cat acct.data > acct.data
cat: input acct.data is output
```

當指令 cat 所產生的標準輸出的正是檔案“acct.data”的內容，而我們又要將這個輸出重導向到檔案“acct.data”中。於是造成指令 cat 的執行錯誤。在這裡我們要告訴你在運用重導向時，不能有輸出檔名與重導向的檔名相同的情況產生。

符號“>>”與“>>&”

使用語法 command >> file
command >>& file

這兩個重導向的符號比起先前我們所介紹的符號“>”與“>&”，在功能上有一個很大的差別。就是符號“>>”與“>>&”在重導向輸出資料到檔案時，不會重寫原檔案內容。它們只會將輸出資料重導向到原檔案內容的後面。

有了這兩個符號，我們便可將多個指令的輸出一起重導到同一個檔案中，而不必再擔心內容會被重寫了。我們來看個例子：

```
3 % who >> data.tmp
4 % ps axu >> data.tmp
5 % df >> data.tmp
```

第一個執行的指令“who >> data.tmp”如果檔案“data.tmp”不存在，重導向符號“>>”與符號“>”相同，會自動產生該檔案，並將輸出重導向到檔案中。如果“data.tmp”檔案已存在，則指令輸出便直接在寫到原有的內容後面。而第二、三個指令的輸出，一樣地會依序串寫到該檔案中。如果你覺得三行指令太麻煩了，改成以下的執行方式有也行：

```
% ( who ; ps axu ; df ) >> data.tmp
```

（如果連標準錯誤資料也要寫入檔案則必須使用符號“>>&”）

設定 $noclobber 預設變數改變輸出重導向特性

變數設定語法 set noclobber
取消變數設定語法 unset noclobber

以上對輸出重導向所使用的符號已介紹了四個（第一組），但其實還有四個尚未介紹（第二組），它們的功能與上述的符號有點對映關係。如下：

第一組	“>”	“>&”	“>>”	“>>&”
第二組	“>!”	“>&!”	“>>!”	“>>&!”

以下的四個符號在未設定 $noclobber 變數時功能與上四個完全相同。但如果你設定 $noclobber 變數，使用符號“>”與“>&”將無法重寫（overwrite）已存在的檔案，而必需使用符號“>!”與“>&!”才行。同時使用符號“>>”與“>>&”亦不能將指令輸出導向不存在的檔案中，得使用“>>!”與“>>&!”。

在下面的例子當中，我們先設定了 $noclobber 變數，然後指令 ls 的結果告訴我們該目錄下並無任何檔案存在。接著我們操作幾個運用重導向的指令看看它們的結果。

```
1 % set noclobber
2 % ls
total 0
3 % cat /etc/passwd >> passwd.bak
passwd.bak: No such file or directory
4 % cat /etc/passwd >>! passwd.bak
```

當指令 3 執行失敗的原因是“passwd.bak”檔案不存在。這如果在沒有設定預設變數 noclobber 的情況下，它是不會產生錯誤的。接著指令4將重導符號由“>>”改爲“>>!”便順利地執行成功了。這時以指令 ls -l 便可清楚地看到該檔案的各項資料。

```
% ls -l
total 1
1 -rw-r--r-- 1 akira 1182 Sep 9 15:19 passwd.bak
```

在這種環境下如果我們鍵入了以下的指令：

```
% ps axu > passwd.bak
file_a: File exists.
```

則因爲該檔名已經存在，不允許我們用符號“>”來將該檔的內容重寫。這便是設定預設變數 noclobber 所產生的保護作用。如果想重寫該檔案則必須使用符號“>!”才行，如下所示：

```
% ps axu >! passwd.bak
```

以上所模擬的情況僅只是設定前與設定後的最大不同處。至餘其它的情況請自行在電腦上依序演練，注意到與原先未設定變數前的種種差別，必可加深在使用上的經驗。此項功能將對於撰寫 shell

程式有相當的幫助。如果你想取消該變數的設定只要在指令行鍵入“unset noclobber”便可以了。在上面的各個重導向的情況中，如果須連標準錯誤訊息也一起重導的話，只要重導向符號代有“&”便可。如“>&!”及“>>&!”。千萬注意！位置不能放錯。

實用的輸入/輸出重導向運用

標準輸出與標準錯誤的分離重導向

就如我們所知，一個指令的輸出訊息包含了標準輸出與標準錯誤兩種型態。有時我們會有需要將某指令的輸出訊息儲存建檔管理，除了正確的輸出資料外，那些錯誤訊往往也是相當重要的，爲了將有兩種訊息分開來存放，重導向的使用手法就顯得格外的重要，針對此種情況常用的重導向方法是將指令與重導向標準輸出用括號括起來，如此一來此部份會先執行，並且指令的標準輸出訊息也會先儲存到指定的檔案中，輸出訊息剩下的部份就是標準錯誤訊息，我們再用重導符號“>&”重導到指定的錯誤訊息檔案便大功告成了。使用語法如下：

```
( command > stdout.file ) >& err.data
```

讓我們來看一個實際的運用：

```
% (find /user1 -type f -size +50000c -ls > big.file) >& err.file
```

上例中我們用指令 find 去搜尋目錄 /user1 之下所有超過 50000 字元的檔案，並將結果以檔名 big.file 儲存。但指令執行的錯誤訊息則存到 err.file 檔案中。如此便將指令 find 的標準輸出與標準錯誤訊息分開來存放，讓我們更容易整理。

輸出重導向與/dev/null－－UNIX系統的垃箕筒

真正在實際的運用上，有時候執行一個指令並不是要它的輸出結果，而是要它的指令執行狀態訊號。（這種使用手法在撰寫 Shell 程式時常使用的上）。這時輸出的訊號反而便成一不需要的，這時有沒有可能將指令的輸出，讓它“消失”呢？有的！在 UNIX 系統中的早就爲你準備了一個垃箕筒，專門供你處理不必要輸出訊息。你只要將輸出重導到“/dev/null”這個垃箕筒來，輸出訊號自然便消失，而且也不會佔用硬碟或其它資源。來看一個運用指令 grep 的例子：

```
1 % ps axu | grep cron | grep -v grep
root 136 0.0 0.0 56 0 ? IW Oct 3 0:00 cron
2 % echo $status
0
```

在上面我們將指令所輸出的系統處理程序的資料，用管線引導到指令 grep 中去篩選“cron”，並去除 grep 指令本身的處理程序。但這並非我們主要的所須的，真正所要的是指令 grep 的執行狀態，如上用指令“echo $status”所顯示的輸出“0”，便是指令 grep 的執行狀態。“0”表示指

令 grep 有找到指定的字串，如果是“1”則表示沒有找到所指定的字串。我們可看見第一行指令是會有訊號輸出的，讓我們將它改一改。如下：

```
3 % ps axu | grep cron | grep -v grep >& /dev/null
4 % echo $status
0
```

指令的輸出訊號不再出現在螢幕，且指令的執行情況正常。這便是“/dev/null”的妙用。當然你也可以用它來重導向根本不需要的要輸出訊號。如下：

```
5 % tar cvf /dev/rst8 /user[1-3] > /dev/null
```

像如此便能將看也看不來的指令輸出去掉，只保留標準錯誤訊息可輸出到螢幕上。如此一來，指令執行時是否有錯誤訊息更可一目了然。

輸出、輸入重導向的混用

輸出、輸入重導向是允許混合起來運用的，如前面提到輸入重導向的運用上，將程式或指令所需的資料用重導入，再將執行結果重導到檔案中。這便是常見的使用方式。見以下使用語法：

```
command < file.input > file.output
```

我們來看一個例子：比方有一份報表要將大寫字母轉換成小寫字母。

```
% tr "[A-Z]" "[a-z]" < report.org > report.low
```

在上例，我們將檔案“report.org”倒導入指令 tr 去做大寫字母轉換成小寫字母的動作，然後將 tr 指令的結果重導向到檔案“report.low”中儲存。而原始的檔案“report.org”則毫無改變地保留。檔案“report.low”便是我們想要的結果。

另外也有一種使用情況，讓我們來看下面的例子：

```
1 % sort -n > munber.data << end!
3838
5
29
128
end!

2 %cat munber.data
5
29
128
3838
%
```

假定你有一串數字要建檔並且要你排序排好，相信以上的使用方法你一定會喜歡得不得了！

重導向符號說明

符號	說 明
>	將 stdout 重導向到檔案（command > file）
>>	將 stdout 資料串加到檔案內容之後（command >> file）
>&	stdout 及 stderr 重導向到檔案（command >& file）
>>&	將 steout 及 stderr 資料串加到檔案內容之後（command >>& file）
>!	將 stdout 重導向到檔案，有設定 $noclobber 時，可重寫檔案。
>>!	將 stdout 資料串加到檔案內容之後，有設定 $noclobber 時，可重寫檔案。
>&!	stdout 及 stderr 重導向到檔案，有設定 $noclobber 時，可重寫檔案。
>>&!	將 steout 及 stderr 資料串加到檔案內容之後，有設定 $noclobber 時，可重寫檔案。

檔名擴展（Filename expansion）運用

在 UNIX 系統中有時難免會用上長長的路徑，長長的檔案名稱或一大堆的檔名在指令中，所以在 C Shell 中提供了幾個符號來幫助我們解決這使用上的困擾。這就是檔名擴展的功能，也有人稱它爲 wildcard。相信你對“*”這個符號一定不會陌生才對。你一定常運用它來代表所有相關的檔案名稱，它便是包含在這個檔名擴展的功能之一。以下讓我們通盤地看一看整個的檔名擴展功能。

符號“＊”與“？”

一般使用者使用符號“＊”的比例可能遠多過使用符號“？”。這可能是因爲符號“＊”代表任何的字元（包含 null）。而符號“？”僅用以代表任何一個單一的字元。這差別是相當大的。讓我們來看一個例子：

```
1 % ls
a a1 a2 aa aaa
2 % ls a*
a a1 a2 aa aaa
```

第一個指令 ls 告訴我們共有五個檔案，當我們以“a*”來顯示相關檔案時，五個檔案全出現了，其中檔案a的出現證明了符號“＊”連 null 都代表的特性。而檔案 aaa 的出現告訴了我們一個符號“＊”便可代表多個字元。接下來我們把符號“＊”換成符號“？”來看看結果的差異性。

```
3 % ls a?
a1 a2 aa
```

你可發現檔案 a 沒出現，因爲符號“？”不代表 null；檔案 aaa 也沒出現，這是因爲符號“？”僅代表一個字元。如果你必需用 a?? 才行，如下：

```
4 % ls a??
aaa
```

看清楚僅有檔案 aaa 出現而已，a1、a2、aa 等只有兩個字元不符合條件。所以說使用者爲了在使用上的方便，會運用符號“＊”多過符號“？”大概就是這個原因吧！但有一點必須說明當你在撰寫 shell 程式時，如果需要使用類似的功能時，別忘了符號“？”是比較嚴謹的安全的作法，不妨改便你的使用習慣看看。因爲撰寫 shell 程式要應付的情況較多，嚴謹一點的作法比較不會產生 bug。

對於“＊”這個符號，真可說是懶人之“星”。像指令 cd 到某目錄時，目錄名稱打到一半接著“＊”代表之。ls、vi、lpr、cp、rm 等等等的指令也均如法泡製。

在符號“＊”的運用上也不盡然是所有指令都能接受的，像指令 mv、cp 便會有語法上無法接受“＊”代表檔案的情況，看下面：

```
% mv acct.* acctold.*
usage: mv [-if] f1 f2 or mv [-if] f1 ... fn d1 (`fn' is a file or directory)
% cp acct.* acctold.*
Usage: cp [-ip] f1 f2; or: cp [-ipr] f1 ... fn d2
```

哈哈！太懶了！！被電腦“教訓”了一頓！！！不過說實在的作者我一直認爲懶惰是人類發明創造的原動力！一切的偉大發明皆來自於偉大的懶惰。指令不能滿足我們的懶惰，則我們自己便應該爲自己創造出一個屬於自己的“懶惰功能”來。在 Shell 程式設計單元中你將可爲自己的懶惰感到無比的光榮。

符號“[...]”

這個符號的功能在爲你定義字元、數字的範圍；它可以代表一個範圍內的一群數字或者是字母，也可代表很多個獨立的字元，或者是數字。在使用上相當具有彈性。要使用該符號“[...]”代表一個範圍時，範圍必須由小到大，兩者之間以符號“-”連接。

[0-9] 代表0、1、2、3、4、5、6、7、8、9等數字
[a-z] 代表字母a、b、c、d、...、x、y、z
[A-Za-z] 代表大寫字母A到Z或小寫字母a到z，既大小寫字母全部。
[A-z] 同上
[9-4] 範圍錯誤（錯誤示範）
[z-a] 範圍錯誤（錯誤示範）

要代表多個單獨的字元，只要一個個字元加入括號中便可。如下所示：

[abs] 代表 a 或 b 或 s
[1235] 代表 1 或 2 或 3 或 5
[ab][12] 代表 a1 或 a2 或 b1 或 b2

再提醒你一次，範圍只能由小到大，不可由大到小。如果用由大到小的方式雖然不會產生錯誤訊息，但結果一定是錯誤的。這一點自己可上機試試！以下我們來看一些實際運用的例子：

想一次顯示兩種尾號不同的所有檔案時，如下例所示：

```
% ls *.[sc]
backup1.c backup3.s backup2.c backup4.s
```

想將某個範圍的資料檔案彙整成一個新檔案，比方如上面這四個檔案合起來建一個新檔案。用法如下：

```
% cat backup[1-4].[sc] > backup.new
```

又譬如要將 aa1、aa2、aa3 三個目錄下的檔案拷貝到另一個目錄 all 之下，也可以使用得上。如下例：

```
% cp aa[1-3]/* ~/all/.
```

但是這個符號如果運用到 mkdir 指令時會產生錯誤的，這點請注意。比方你想一次新建五個目錄 disk1、disk2、disk3、disk4、disk5。假定你使用以下的方式：

```
% mkdir disk[1-5]
No match.
```

錯誤訊息告訴你“No match.”，錯在那裡呢？事實上，在使用的語法來說並沒有錯。只不過是 mkdir 指令無法接受這種用法，所以造成指令無法執行成功，產生了錯亂的訊息輸出。雖是如此，但這個法則的可組合運用還是相當好用的。請愛用善用之。

符號“{,,,}”

在前一個符號的最後指令 mkdir 的例子中，該符號做不到的，這個符號“{,,,}”可幫你做到。語法如下：

```
% mkdir disk{1,2,3,4,5}
```

就這樣一次新建五個目錄 disk1、disk2、disk3、disk4、disk5。或者是像以下的運用：

```
% mkdir man/{cat,man}{1,2,3,4,5,6,7,8}
```

一道 mkdir 指令一次建立 16 個目錄。簡單好用的很。

有時它的功能會讓人覺得蠻接進符號“[...]”的，但它們所適合處理的情況其實並不太相同。符號“[...]”所適合處理的是比較簡單化的字元，較複雜無規則性的字元則使用符號“{,,,}”比較恰當。假如我們要將三個存放在 /usr/include 目錄下的檔案一起拷貝到到 ~/time 目錄下，檔案名稱分別爲 time.h、stdlib.h、termio.h，這如果要運用前面符號“[...]”來“簡化”指令，根本是不可能的事。但是，我們可以運用符號“{,,,}”來處理，請看下例：

```
% cp /usr/include/{time,stdlib,termio}.h ~/time
```

這就是符號“{,,,}”所適用的強處。不過符號“{,,,}”對於 “範圍” 的處理能力並不能像符號“[...]”那樣強，可以使用到符號“-”來做到像這樣“[0-9]”簡潔地代表 0 到 9 的“範圍”功能，這點倒叫人在使用上感到有些遺憾。以下讓我們來看看幾個運用符號“{,,,}”來 精簡化指令的實際例子：

```
% mkdir disk1-1 disk1-2 disk1-3 disk2
```

```
% mkdir disk{1{-1,-2,-3},2}
% mv disk2 disk3
% mv disk{2,3}
% diff echo2.c echo2.c.org
% diff echo2.c{,org}
```

符號 “~”

符號 “~” 代表的是使用者自己的 home 目錄。它有一個比較特殊的用法，就是在符號 “~” 之後如果接上另一個使用者名稱，則便代表該使用者的 home 目錄。這點在路徑名稱的運用上幫助相當大。比方你要到另一個使用者 yeats 的 home 目錄下，你可以毫不考慮目前的工作目錄，及你是否知到該使用者的 home 目錄，馬上使用這個用法在 cd 指令上，便可改變工作目錄到該使用者的 home 目錄之下。如下所示：

```
% pwd
/user1/akira/project
% cd ~yeats
% pwd
/user2/group1/yeats
```

像此類的運用最適合同一個工作團體，使用者與使用者之間的檔案傳輸，如下示：

```
% cp backup.c ~yeats
% cp backup.c /user2/group1/yeats
```

以上這兩個指令做的是同一件事情，你會喜歡下那一個呢？尤其當你會搞不清楚那一個使用者的目錄在那裡時，你將會格外的喜歡上它。

管線（pipeline）觀念與運用

將一個指令的標準輸出重導向，做爲下一個指令的標準輸入，像這種連結兩個指令間標準輸出、輸入的特殊功能，便稱爲管線（pipeline）。在 C shell 中定義了兩個關於管線功能的特殊符號。如下：

“|” 只讓標準輸出（stdout）通過
“|&” 標準輸出與標準錯誤（stdout and stderr）均會通過

先讓我們來看看一個例子：

```
2 % who
akira ttyp0 Sep 9 17:49 (akirahost)
akk ttyp1 Sep 9 17:49 (akirahost)
simon ttyp2 Sep 9 17:49 (akirahost)
3 % who > file.a ; wc -l < file.a ; \rm -f file.a
3
4 % who | wc -l
3
```

從上面的例子中，要計算 who 指令的輸出有幾行，以輸出重導向的做法是，先將 who 指令輸出重導向到一個檔案“file.a”中，再將該檔案重導向到指令 wc -l 來計算行數。最後再將產生的資料檔案“file.a”清除。這過程中有三個硬磁的 I/O 動作。但如果使用管線的方法如行 4 所示，它根本沒有硬磁的 I/O 動作，是不是比行 3 所下的指令方便多了呢。而且不需產生任何檔案，省略的 I/O 動作，對使用效率來說是比較好的使用方式。

我們再來看看下面這個例子：

```
% find ~ -name "*.dat" -print | rm
```

上面的指令是用指令 find 找尋 home 目錄下檔名爲“*.dat”並將它列出，再將列出的訊息運用管線通入指令 rm 去將這些檔案清除。說起來好像是一點問題也沒有，其實是一點用也沒有。要不然馬上上機試一試。這個指令的錯不在於指令 find 或運用了管線，而是在於指令 rm 身上，因爲它的引數不接受標準輸入。所以你無法這樣使用。不過指令 find 本身的語法有支援像上面的運用，如下示：

```
% find ~ -name "*.dat" -exec rm -i "{}" \;
```

當指令 find 在找到一個相符的檔名時會執行指令 rm -i，讓你鍵入 y 或 n 來決定是否要清除該檔案，之後，指令 find 會再繼續尋找相符的檔名來做同樣的動作。這樣的功能便與上面那個使用管線不成的指令作用相同了。

所以在管線的運用時一定得注意前後指令的關係，及是否指令接受標準輸入等問題。你越用心去嘗試將可得到越方便的越佳的使用效率。看下面這個運用：

```
% ps axu | egrep 'cron|nfsd|pcnfsd' | egrep -v egrep | lpr
```

從一個系統訊息經過數個管線的連接，將這個系統訊息篩選處理成我們所要的資訊，並直接將它由列表機印出。或者是像下面的運用：

```
% cat files | grep pattern | sort -u | more
```

像這類的運用都是相當得宜的作法。

相關的輔助指令 － tee 指令的功能

對管線的功能而言，並無法像輸出重導向可產生檔案。這是管線功能的不足與限制之處。但不要緊，UNIX 有一指令可填補這項缺撼，這便是 tee 指令。當行 4 指令修改爲 “who | tee file.a | wc -l” ，則指令 who 的輸出一方面由 tee 指令儲存到檔案 “file.a” 中，一方面又 pipe 到指令 wc -l 去計算 who 輸出的行數。這個指令你可用在備份檔案的指令，特別是當你想一面在螢幕上查看檔案備份的情況，而又希望將備份檔案的訊息儲存成檔案時。如下例：

```
1 % tar cvf /dev/rst8 /user1 /user2 |& tee backup.data
```

另外該指令也提供了一個 -a 的選項，來讓你能將訊息加到指定檔案的內容之後。請看以下的運用。

```
2 % date | tee -a log.log
Sun Nov 26 22:15:15 CST 1995
```

在指令 2 中，我們再一次使用 tee 指令將指令 date 的輸出加到檔案 “backup.data” 中，但因爲指令 tee 有加上選項 -a，所以並不會將原檔案的內容清除掉。怎麼樣是不是很好用呢？

history 的設定與運用

history 說起來可算是 C Shell 的重大功能之一。history 這個字如果以它的整體功能來說，或許我們可稱它爲 UNIX 的指令使用記錄。它負責記錄使用過的指令，供使用者“再利用”。這個“再利用”的動作可說是靈活多變，可運用的方面除了指令行外，連別名（aliases）、預設變數（Predefined Variables）的設定或 shell script 的程式設計等均可見到 history 的運用蹤跡。實在是一項使用者不能忽略的好功能。

製定 history 的使用環境

要能善用 history，首先得熟悉關於 history 使用環境的三個重要的預設變數。以下讓我們來爲你一一地介紹：

$history 變數

變數 history 是使用 history 功能之前必須先設定的。通常都在 C Shell 的啓始檔案“.cshrc”中設定此變數，當然也可用手動的方式來設定或更改。設定的語法如下：

set history = n （n是數字且必須是整數）

如果你把 n 設定爲 30，則 C Shell 將會隨時爲你保留最後所下的 30 道指令，供你呼叫顯示到螢幕上，這便是 history list，它可供你運用 history 的其它功能。如果你想查看此變數的設定值，可使用指令

echo $history

來顯示 history 設定值。要重新設定可修改“.cshrc”檔案內的設定，再以指令 source .cshrc 便能更新設定值。也可直接下指令 set history = n 來更改。但後者設定的設定值會隨著該 Shell 的終結而消失，此點請注意。

$savehist 變數

設定這個變數的作用是在 Shell 終結後，將最後某幾道指令儲存到使用者的 home 目錄下的“.history”特殊檔案中，以供你下次 login 或另一個還在工作的 shell 來運用，上一個 shell 所記錄下來的最後幾道指令。如你設定爲下：

set savehist = 50

則每一個 shell 的終結，都會對“~/.history”檔案做資料寫入、更新的動作。請注意！再重覆一遍，是每一個 shell 的終結都會更新 home 目錄下的“.history”檔案。所以當你下次 login 時所

見到的 history list，便是“.history”檔案的內容。

如果你常在 X Windows 或者是 Open Windows 等 GUI Windows 界面下工作的話，相信你一定會開好幾個 Windows 同時工作。在這種工作環境下如果你不去加以控制要儲存那一個 shell 的話，建議你不要使用它，或許會比較好些。

$histchars 變數

這個變數便是用來改變設定 history 的運用符號。用來解決符號的使用習慣問題。history 的專屬符號有兩個，第一個使用的符號“!”第二個符號“^”。如果你如下設定：

```
set histchars = "#/"
```

則符號“#”取代符號“!”；而符號“/”取代符號“^”。老實說來 UNIX系統對特殊符號的運用可說到了一“符”多“棲”的田地了，小心改出毛病來。建議你謹慎使用（最好是不用）。

history 的運用說明

設定 history 的數量與顯示的關係

在一般使用的情況下，history 的設定數量大約都在 20 ~ 50 之間。這是因爲設定太大了會佔用系統資源，用不上的話實在不划算。適中夠用就是好的設定值。你也可就螢幕顯示的列數來做爲設定值的參考。免得列出history 時超過螢幕所能容納的行數。要在螢幕上列出 history 相當簡單，鍵入 history 便可。如果你嫌“history”字太長不好記，設個別名“alias h history”，相信也不會有人罵你懶！

在指令行模式下鍵入 history 可顯示出 history list。當然你也可以在指令 history 後加上一個數字，用來控制顯示 history list 的數量。如下：

```
25 % history 5
21 ls
22 cd subdir1
23 ls
24 cl
25 history 5
```

在 BSD 版本的 UNIX 系統中，C Shell 會提供 history 一項比較特殊的選項，就是“-r”選項。它的作用是將 hisroty list 顯示的事件次序倒過來。如下所示：

```
26 % history -r 5
26 history -r 5
25 history 5
24 cl
23 ls
```

```
22 cd subdir1
27 %
```

較長的設定值在顯示上會有超出螢幕行數的情況產生，這個問題很好解決，提供幾種別名設定供你參考使用：

```
alias hup 'history | head -15' （前15道history）
alias hdn 'history | tail -15' （後15道history）
alias hm 'history | more' （一頁一頁看）
```

history 對過去指令的處理方式說明

C Shell 的 history 功能，對於使用者所執行過的每個指令，基本上都把它當成一個“事件（event）”來處理。而每個事件都以數字來編號代表之。讓我們來在設定預設變數 $prompt 中加入 history 功能來說明，這種事件的處理情況。在指令行中鍵入如下：

```
% set prompt = '\! % '
```

這樣你便可在提詞（prompt）中清楚地看到 history 對“事件”的編號情況。如以下實際例子：

```
% set prompt = '\! % '
13 % history
4 cd test
5 \rm -r test
6 tar cvf /dev/rst8 backup &
7 ls
8 vi backup.task
9 set history = 50
10 ps axu | grep cron
11 cd
12 set prompt = '! % '
13 history
14 % echo $history
10
```

從以上實例中看到 history 對過去指令的編號情況。通常 history 對指令的編號由 1 開始編起，每次自動加 1。但如果你設定預設變數 $savehist，則由此變數的數字加 1 算起。此點請注意。

到此相信你對 history 的運作應該有一個概略的瞭解了吧！接下來便開始來運用 history 的功能，來“便利”我們的指令操作。如果你能善用將可大大改變你的使用效率。它至少可以降低你因爲打多了鍵盤而得到職業病的機率。

history 的符號說明與基本運用

我們已經知道 history 對指令是以事件來處理，且加以編號。要如何簡單地“再利用”這些過去的“事件”呢？history 定義了一群符號來供我們變化使用。這些符號都是組合式的，基礎的第一個

符號是“!”。此符號用來啓動 history 功能，但緊接在符號後的不能是下列這些符號：TAB 鍵、空白鍵、換行、等號或“(”及“.”等。接上這些符號將會產生錯誤。

history 在指令行運作下的常用到的符號運用組合，我們以功能分類來加以說明：

1. 指定事件執行

符號“!!”

“!!”執行上一個指令。在運用上除了執行上個指令外，並可在符號後在加入與指令語法不相沖的字元來修飾上個指令。如下例：

```
% ls
akira.sch passwd
```

以上是指令 ls 的執行情況，當你覺得所顯示的訊息不足，想要加上選項-l時，你可在運用 history 時再加上該選項，如下用“!! -l”便相當於下指令“ls -l”：

```
% !! -l
total 2
1 -rw-r--r-- 1 akira 184 Sep 16 11:32 akira.sch
1 -rw-r--r-- 1 akira 65 Sep 16 11:31 passwd
```

符號“!n”

“!n”執行第 n 個事件。注意到這第 n 個事件一定得還在 history list 內才能順利執行。

假設我們的 history 變數設定值爲 10，當我們執行到第 22 個事件時，想要用符號“!n”來執行第 5 個事件，會產生什麼情況呢？請看下面：

```
21 % echo $history
10
22 % !5
5: Event not found.
22 %
```

“5: Event not found.”這個錯誤訊息便明顯的告訴了我們，指令所要執行的 history 事件以不在 history list 的範圍之內。如果你常會有這種情況發生，可以考慮將 history 變數的設定值加大到適當的數字。另外請注意到在事件 22 執行指令“!5”時，因產生“5: Event not found.”的錯誤，history 對此錯誤的指令並不記錄下來，所以事件的編號停留“22”。如果你所指定的事件在 history list 的範圍內的話，指令便能順利的執行過去的事件。如下：

```
22 % !17
```

```
ls
total 4
2 cshrc* 1 file 1 passwd
```

符號“!-n”

“!-n” 執行在 history list 中倒數第 n 個事件。

我們用下面的例子說明它的使用情況：

```
24 % h
15 ps axu | grep cron
16 pwd
17 ls
18 vi passwd
19 h
20 cl
21 echo $history
22 vi ~/.cshrc
23 ls
24 h
25 % !-4
echo $history
10
26 %
```

看到沒有事件 25 的指令“!-4”所執行的是 history list 的事件 21，剛好是倒數第四個。

2. 搜尋指定執行

“!string” 搜尋以某字串 string 爲開頭的過去指令並加以執行。
“!?string?” 搜尋過去指令行中有某字串 ?string? 並加以執行。

以上兩種使用方式均是在 history list 的 event 中，搜尋 event 的開頭或 event 中間有指定的字串，來加以執行。但均是執行第一個符合條件的 event。搜尋時以由 event 數字大到小。如下例所示：

```
34 % h
25 h
26 cd
27 tar cvf /dev/rst8 akbin &
28 ps
29 vi ~/.cshrc
30 tar tvf /dev/rst8
31 cd test
32 ls
33 vi passwd
34 h
35 % !v
```

在上例中最後指令“!v”所執行的是第33個“vi passwd”，而不會是第27個“vi ~/.cshrc”。如果要執行第 27 個 event，你可用“!?cshrc”來執行，當然你也可直接用“!27”來執行，但會使用搜尋方式，自然是在不記的第幾個 event 的情況下才會採用的，不是嗎？又如果你設定的 $history 爲 100 個，要用眼睛在 history list 中找尋，倒不如“奴役”電腦還來得好。是嗎？

接下來介紹一個比較特殊的符號“!{...}”，用來應付一種比較特殊的情況。譬如你想在重執行以下這個指令：

```
% cat /etc/hosts > ~/hosts
```

並且要緊接著在指令後加入一些字串。如將重導向的“~/hosts”這個檔案名稱改爲“~/hosts.bak”， 如果你下指令如下：

```
% !cat.bak
cat.bak: Event not found.
```

這種錯誤是因爲搜尋的字串與添加的字串沒有區隔開來的結果。解決的方法是使用大括號“{ }”將搜尋的字串括起來，再僅接著添加的字串便可。如下所示：

```
% !{cat}.bak
cat /etc/passwd > ~/hosts.bak
```

3. 修改執行過去指令

下錯指令、打錯字的情況是難免會發生的，或大同小異的指令也偶而會連續使用，像此類的情況運用修改的方式相當適當。（由其指令又臭又長的時候，你使用時一定會感激造令者的偉大）先來看一個例子：

```
% rsh hosta -l user "screendump -x 20 -y 20 -X 300 -Y 300 scrfileA"
```

如果是像這種指令打錯了一個字，比方說指令 screendump 少打了一個字母 e，變成 screndump，天呀！主管在旁邊看耶！！重打一次？如果再打錯保證你年終獎金少一半！！！這可怎麼辦？？不用急，露一手什麼叫“知錯能改”，如下：

```
語法 ^old^new^ 修改上個指令的old爲new後執行。
實際指令 % ^re^ree^
```

就這樣把你漏打的字母補了進去。簡單幾個字就搞定，方便吧！不過這種方式的修改是有限制的。它只能還是以上面那個長的不得了的指令中更改一個地方，而且還得是上一個“事件”才行。如果不是上一道指令，要做類似的修改運用，可用以下的語法均可做到：

```
!n:s/old/new/ 修改第 n 個事件的 old 爲 new 後執行。
!string:s/old/new/ 搜尋以 string 爲開頭的事件並修改 old 爲 new 後執行之。
```

到目前所介紹的三種方式，均有個共通性，就是只能更改一組字串。而且字串內不可有空白鍵。如果有空白鍵會造成修改錯誤。如果要同時修改兩個以上相同的字串，對後面兩種語法中再加一個“g”便可，如下：

!n:gs/old/new/ 修改第 n 個事件中所有的 old 爲 new 後執行。
!string:gs/old/new/ 搜尋以 string 爲開頭的事件並修改所有的 old 爲 new 後執行之。

用這種方法便可輕易地將那個臭長指令中的 20，一起更改爲想要的數字了。如下示：

```
% !rsh:gs/20/50/
```

這時我們所得到的將相當於下：

```
% rsh hosta -l user "screendump -x 50 -y 50 -X 300 -Y 300 scrfileA"
```

自己找例子試一試吧！它們真的是太好用了。

history 對於事件的引數（argument）運用

1. 引數的區分方式

在前面我們曾經提到 history 對指令是以編號的事件來處理，但你除了可再利用這些事件外，同時你也可再利用事件中的每一個引數。先讓我們來看一個底下這個例子：

事件的引數由 0 開始編起，所以 0 便代表事件開頭的指令。接下來是以空白鍵來隔開的字串或以特殊符號依序編號。注意到引數的編號，並非全以空白鍵隔開才算，因爲只要有特殊符號，不管它與前後有無空白鍵都得算是一個單獨的引數。（這些特殊符號有 |,<,<<,>,>>,& 等）history 對引數安排了多種方式來代表引數的位置，如數字（0，1，2，...）或符號（“^” “$” “*” “x*” “x-”）。對引數的運用可單一的或者是一個範圍（x-y）均可。

以下我們以表列來說明引數的符號使用方式：

符號	說 明
!!:n	使用上個事件的第 n 個引數，如“!!:3”
!!:x-y	使用上個事件的第 x 個引數到第 y 個引數，如“!!:0-4”
!n:^	使用第 n 個事件的第 1 個引數，符號“^”代表第一個引數
!^	使用上個事件的第 1 個引數
!$	使用上個事件的最後一個引數，符號“$”代表第最後個引數
!*	使用上個事件的第 1 個引數到最後一個引數
!!:x*	使用上個事件的第 x 個引數到最後一個引數，（相當於“!!:x-$”）如“!!:2*”即代

	表使用第 2 個之後的所有引數
!!:x-	與上相似不同處爲不包含最後的引數，如“!!:2-”即代表使用第 2個到最後的倒數第一個引數。（語法“!!:x-”就是“!!:x-$”去掉“$”後所剩下來的樣子）

2. 引數在指令行的運用

在指令行模式下有些時候會連續好幾個指令都用到同一個或一群檔名，也就事說，你可能會有一個字串要一再重複鍵入。比方有以下情況：

```
1 % ls -l ch1.doc ch2.doc ch3.doc
2 % chmod g+w ch1.doc ch2.doc ch3.doc
3 % tar cvf /dev/rst8 ch1.doc ch2.doc ch3.doc
```

像以上這接連的三個指令都用到相同的檔案名稱 ch1.doc，ch2.doc， ch3.doc，利用 C Shell 的 history 功能，我們可在指令中將過去事件的引數利用符號代入，以簡化指令的長度，如以下所示：

```
4 % ls -l ch1.doc ch2.doc ch3.doc
5 % chmod g+w !!:2*
6 % tar cvf /dev/rst8 !!:2*
```

指令 5 中的“!!:2*”就代表指令 4 中的第 2 到最後的引數，所以指令 5 相當於指令 2。同理指令 6 相當於指令 3。

再舉一個例子：

```
7 % ls -l *.doc *.txt
8 % rm -r !$
9 % cat *.doc > doc.files
10 % !6:0-2 !!$
```

指令 8 中的“!$”既代表指令 7 中的最後的引數，就是“*.txt”。指令 10 的使用在語法上是成立的。連用 history 功能來組成指令，第一組“!6:0-2”是使用事件 6 的第 0 到第 2 個引數，接著空白鍵，接著第二組“!!$”使用前一個事件的最後一個引數。所以此指令相當於下：

```
10 % tar cvf /dev/rst8 doc.file
```

這便是 history 的引數的組合運用。對了！相信你還記得下面這個又臭又長的指令吧！

```
% rsh hosta -l user "screendump -x 50 -y 50 -X 300 -Y 300 scrfileA"
```

如果現在要你運用 history 的修改功能來更改第二個“50”變爲“60”的話，你該如何下指令？回憶一下過去所介紹過的“修改功能”中，只能更改第一個或全部都改，這只改第二個可怎麼改才好呢？相信上面的指令 10會代給你靈感的。想到了沒，答案如下：

```
% !rsh:0-7 60 !rsh:9*
```

我們用該事件的引數 0 到 7，而引數 8 便是要修改的對像，我們不引用。在“!rsh:0-7”之後，我們便加上“□60□”，再接上該事件的引數 9 到最後的引數。如此便能成功地將某個事件中的某個引數，代換成所需（“□”代表空白鍵）。運用這樣較麻煩的方法可補原先與修改功能的不足與缺憾。希望有一天這種方法能幫你解決一些困擾。

在 history 功能中有一項是僅將指令或所須的功能列出，而不執行。使用的符號爲“:p”。這種列出不執行的方式，對比較複雜或比較沒有把握的組合運用是有幫助的。我們來修改第 10 個的使用語法成爲只顯示而不執行，來看看它的用法：

```
10 % !6:0-2:p !!:$:p
tar cvf /dev/rst8 doc.file
11 %
```

如上指令 10 所得的結果顯示在下行，但並不執行。如果錯誤則可加以更正，如果正確無誤想要執行則再鍵入“!!”便可。

3. 引數的特殊符號運用

在 history 的引數運用上有一項是比較特殊的，它是關於路徑（path）的運用。一般也並不常運用在指令行模式之下，倒是在 shell 的程式設計上相當有用。不過我們在介紹時，還是將它運用在指令行模式之下，讀者可借此瞭解它的功能。

```
27 % !cat:p
cat /usr/adm/messages > message.1
28 % !!:1:t:p
messages
29 % !cat:1:h:p
/usr/adm
```

由上例中我們可藉由 history 的功能來將一個路徑分離成兩個部份，如 “/usr/adm/messages” 可分離成，“/usr/adm”與“messages”。如果以前面這個情況來說，一個絕對路徑（full path）可將它分離成檔案所在的路徑及檔案名稱這兩項資料吧！這個運用情況，我們將在 shell 程式設計中再爲你舉例說明。

關於 history list 的說明

history list 實際上便是你啓動 History 功能之後所產生的一個指令暫存器。這個指令暫存器會記錄的你下過的指令，也可稱之爲事件（even）。儲存事件的數量便是你使用內建指令「set history = xx」所指定的數量。 history list 的內容允許被呼叫出來加以再次使用或者是修改使用，被呼叫出來的事件可以是整個事件也可以是事件的部份引數，在使用上有相當多的方式及符號，當然組合

的變化也相當多。同時這個 history list 的內容也可以在你 logout 前儲存成特殊檔案供你下一次 login 時再利用。

如何傳遞 hisrory list 到另一個 C shell 中

在前面我們曾提到變數 savehist，它的功能是可以在你結束 C shell 之後，把該 shell 的 history list 儲存起來供你下一次 login 時，或者是另一個 C shell 來使用，但如果你並不想終結現在正在操作的 C shell，卻需要將 history list 儲存起來，提供另一個 C shell 運用時。此時變數 savehist 的功能就不能適用了。我們得尋求變通的方式來達到這個目的。首先，將所需的 history list 數量重導向到一個檔案中儲存，如下所示：

```
33 % history -h 15 > history.15
34 % vi history.15
```

接著便是使用編輯器清除檔案內不必要的指令。然後退出 vi。這時另一個 C shell 便可以看見這個檔案“history.15”。這時我們便可使用內建指令 source -h 來將該讀取“history.15”檔案內的各行指令，將它們加入到這個 C shell 的 history list 中，如此便可以供我們來運用了。請見實際的指令操作：

```
10 % source -h history.15
22 %
```

（請注意，加上選項 -h 的作用在於只讀入指令，但並不執行。關於詳細的情況請參考內建指令 source 說明）

別名（aliases）的設定與運用

設定語法 alias name 'command'
解除設定語法 unalias name
顯示設定語法 alias （顯示所有別名設定）
alias name （顯示name的別名設定）

別名（aliases）是 C Shell 的內建指令。主要的功能是設定一個“小名”來取代常用的或複雜的指令組合。在應用上可說是最多樣、方便的好功能（此 aliases 功能在 Bourne Shell 中不支援）。在下例中便是別名的設定、顯示、使用情況：

```
2 % alias ls 'ls -aF'
3 % alias ls
ls -aF
4 % ls
./ ../ .cshrc* a b file login logout
```

當我們在使用常用的指令時，總不免會固定地使用指令的某一些選項。所以有人便使用設定 aliases 的方式來代替之。這是 aliases 最常見的應用。如上例我們設定一個叫“ls”的 aliases，它所代表的是指令“ls -aF”。在設定完之後我們所執行的 ls 指令其實已經不是原來系統的指令 ls，而是“ls -aF”。這全與 C Shell 對指令執行的解譯流程問題有關。因爲 C Shell 在替使用者解譯鍵入的指令時，如果是屬於 C Shell 的內建指令，則會直接執行。如果指令是以符號“/”爲開頭，則將會被認定爲是路徑名稱，並會到該路徑下去搜尋該指令並加以執行。如果開頭沒有“/”符號，則會有兩種情況，依變數 path 裡所設定的目錄逐序加以搜尋及執行，這是一般的情況。但如果該指令名稱在內建指令 alias 所建立的資料內被發現，則 C Shell 會先將該指令名稱轉換成內建指令 alias 所建立的對映資料，再依此資料內容到變數 path 裡所設定的目錄群中去搜尋與執行。這便是設定 aliases 後會取代原指令執行的原因。

這種 aliases 名稱與系統指令名稱相同的設定，雖然帶給我們在指令使用上的方便，但也會造成一些在使用指令上的負面影響，使用者不可不小心謹慎。譬如在“ls”已被設定爲“ls -aF”的情況下，我們鍵入“ls -l”，則真正執行的其實是“ls -aF -l”。而不是原指令 ls 的“ls -l”。這一點請讀者務必認識清楚。

如果你想要將“ls”的 aliases 設定消除，可用指令 unalias 在指令行模式下解除“ls”的設定，如下所示：

```
5 % unalias ls
```

如果說你在不想解除 ls 的 aliases 設定情況之下，想要運用原指令 ls。有兩種方式，第一：指令要以絕對路徑的方式來執行，以指令 ls 爲例，須鍵入 /bin/ls 來執行。但這種方法老實說是比較笨的人用的。第二種方法是運用 C Shell 爲我們所提供的特殊符號“\”擋去 alias 指令的設

定，達到你要使用原指令的需求。如下所示：

```
6 % \ls
a b file login logout
```

如此你便可在指令的使用上得到更大的彈性，做更有效率的運用。

aliases 通常均設定在 C Shell 的啓始檔案“.cshrc”中；也可簽入後在設定，不夠設定值會隨著 logout 或 shell 的終止而消失。最好的方式還是設定在“.cshrc”檔案中，對使用上比較方便。

簡單的 aliases 設定如“alias ls ls -asF”或“alias rm rm -i”等可不用符號來括住指令，但如果是指令組合中有運用到連續指令的符號“;”時，則一定得用符號來將指令組合括住，否則設定將出現問題。所以在設定 aliases 最好養成用符號將指令組合括住的好習慣。如下例，第一個 aliases 設定便是錯誤；而第二個有括住指令組合的，才是正確的。

```
% alias ww who ; date
alias ww who
date
Mon Sep 12 03:06:32 CST 1994
% ww
who
akira ttyp0 Sep 12 02:34 (akirahost)
% alias ww 'who ; date'
alias ww who ; date
% ww
who
akira ttyp0 Sep 12 02:34 (akirahost)
date
Mon Sep 12 03:09:28 CST 1994
```

別名設定中運用事件的引數

在別名的設定是允許代入變數或使用引數的。先來看看一般使用者常用來查看系統是否執行某個 process 的指令組合：

```
% ps axu | grep pattern （pattern代表process的名稱）
```

如果像以上指令設定爲下：

```
% alias psg 'ps axu | grep'
```

讓我們來實際運用這個 psg 來尋找 cron 這個 process 看看：

```
/home1/akira> set echo
/home1/akira> alias psg 'ps axu | grep'
alias psg ps axu | grep
/home1/akira> psg cron
```

```
ps axu
grep cron
root 136 0.0 0.0 56 0 ? IW 00:41 0:00 cron
akira 172 0.0 1.4 32 196 p0 S 01:01 0:00 grep cron
```

上例中指令 set echo 的在顯示執行的指令。（set echo 是 shell 的一個預設變數在後面將會有詳細的說明）在例子中可清楚的看到設定 aliases 後所執行的 psg cron 情況，因爲 psg=ps axu | grep，所以 psg cron 便相當於 ps axu | grep cron。實際上這個指令組合在 grep 的處理的並不完美，因爲會連自己的 process 也顯示出來。所以我們將指令組合再加一道指令來去掉這個 bug。如下：

```
% ps axu | grep pattern | grep -v grep
```

用指令 grep -v grep 來去掉不需要的訊息。但如此一來 grep 所需 pattern 要如何帶入 aliases 的設定呢？這便是如何將下指令的引數帶入 aliases 的設定中。關於這種情況你可以運用以下這幾個特殊符號來導入所需的引數，以達到你所想要設定功能。

符號	說明
!*	代表指令行的第一個到最後一個引數
!^	代表指令行的第一個引數
!:n	代表指令行的第 n 個引數

所以關於上一個 aliases 設定，便可使用“!^”或“!:1”來帶入第一個引數。aliases 設定如下所列兩種方式均可：

```
% alias psg 'ps axu | grep \!^ | grep -v grep'
% alias psg 'ps axu | grep \!:1 | grep -v grep'
```

如果要同時運用多個引數，我們以常用的 find 指令複雜的運用來做爲例子，如下：

```
% find path -name filename -type f rm -i {} \;
% alias ffrm 'find \!:1 -name \!:2 -type f rm -i {} \;'
% ffrm / cshrc.old
```

如果要同時將所有引數全部帶入且無法預期引數的數量，符號“!*”正好符合需求，例如：

```
% chmod u+x filenames
% alias cux 'chmod u+x \!*'
% ls -l dirnames | grep "^d"
% alias lsd 'ls \!* | grep "^d"
```

（這個例子中的 grep "^d" 是指每行開頭的第一個字母爲 d，才顯示。整個指令組合運用來顯示指定目錄下層子目錄。）

別名設定中引用別名

在 aliases 設定的指令組合中允許使用已設定的 aliases。如下說明：

```
% alias a alias
% a h history
% a hten 'h | head -10'
```

第一行設定 a = alias，第二行馬上運用第一行所設定的 a = alias 來設定 h = history，第三行中則運用了第一行及第二行的的設定。像此類的設定雖然成立，但實際運用時可得小心。

別名設定的迴圈錯誤（alias loop）現象

```
% alias date 'clear;date'
% date
Alias loop.
% alias ps 'who; ps'
% ps
Alias loop.
```

看看上面這幾個 aliases 的設定，雖然在語法上是合理的，設定時也不會產生錯誤訊息，但爲何執行時卻都會產生錯誤呢？請仔細想想看！其實原因均是在“指令組合”中運用連續指令時產生的，其現象就是當 aliases 的名稱與所設定的 aliases 內容的最後一個指令相同所造成的。請特別注意下例的ps：

```
% alias ps 'who; ps'
```

展開來執行 ps 便等於 who; ps，而連續指令中的 ps 會因爲 aliases 已設定成功的因素，也等於who ; ps，如此設定後的 ps 在執行時便會產生一個怪異的結果，也就是一直在執行who ; who ; ...，這就是所謂的 Alias loop.。

要避開 Alias loop. 有幾種方式，可將指令組合中的指令改爲絕對路徑，如下：

```
% alias ps 'who; /bin/ps'
```

同理但迅速的方法便是利用符號“\”，如下所示：

```
% alias ps 'who; \ps'
```

以上的兩個避開 Alias loop. 的方法，如果碰到要使用 C Shell 的內建指令，就無效了。這時您可以嘗試將指令組合中的指令，次序上對調如下亦可解除“Alias loop.”的情況。（雖然看起來也像會造成執行的迴圈，但其實不會，請放心使用）

```
% alias ps 'ps;who'
```

在前面我們提過 alias 與指令名稱相衝的情況是可允許的，但也是必須要加以特別小心的。因爲在 alias 設定中你可能一不小心便造成這種錯得不知不覺的情況產生。同時別名與指令名稱相沖的情況，最容易影響到設計考慮不週全的 shell script，造成執行錯誤或增加移植上的困擾。此點不可不慎重其事。

幾個實用的別名實例

好的 aliases 設定往往會令人錯覺是一個“新指令”的誕生。而 aliases 的設定方式也可說是千奇百怪，花招無窮。每當我想到或看到有創意的 aliases 時，總不免惹人一笑。我們來看看下面所列的幾個 aliases 的例子：

```
alias vicsh 'vi \!:1 ; chmod u+x \!:1'
```

上面這個 aliase 是將指令 vi 與 chmod 的功能結合在一起，用來免除我們用 vi 編輯器編寫一個新的 C Shell 程式時，還要自己再用指令 chmod 來改便該檔案的“執行權限”的使用困擾。這對常撰寫 C Shell 程式的使用者來說真的相當方便。

```
alias cd 'cd \!*;set prompt = "\! <$cwd> "'
```

工作控制（Job control）的運用

在一個多工的（multi-tasking）環境中，允許你同時執行好幾項工作。就如使用應用軟體、監看系統情況、備份資料、編輯文件、列印文件等等。這些工作你可同時進行，譬如當您從一部個人電腦使用一般的 telnet 工具簽入 UNIX 主機時，你只有一個螢幕又不可能有視窗介面（X Window、Motif、Open Window或 SunView 等等）可使用，你便可以借助 job control 來執行同時他們。這些同時進行的工作就像是排好的卡片一樣。一個在前面，其它的都依次序排一個接一個的排在後面。前面的我們稱它爲前景工作（foreground jobs），後面的我們稱爲背景工作（background jobs）。

前景工作（foreground jobs）

在一般正常的情況下，假設你在鍵入一道指令，如下：

```
1 % find /home -type f -size +50000c -print >& file.big
```

則你必須等待指令執行訊息出現、執行完畢，到下一個提語（prompt）出來之後，你才可以再繼續你的下一道指令鍵入。像這樣的操作模式，所執行的工作就是屬於前景工作。如果指令執行中途想要中斷（interrupt），可用 control-c 的方式中斷指令的執行。或用 control-z 來停止指令。如果你用中斷的方式，當然這個指令或程式的結果有可能是不正確的。像指令 find 的執行的方法是到檔案系統中去搜尋你所指定的某種形態的檔案，在執行過程中，你只能眼看著它佔用這個 shell，除了等它搜尋完畢外，你什麼事也別想再做了。還好指令 find 的執行時間通常還不會太久。但如果像 tar、find、cpio、...等等指令或某些應用軟體的程式，往往執行下來會花費一斷不算短的時間，有時說不定會等上數個小時甚至於數天。這麼長的時間你也要等嗎？別忘了 UNIX 系統可不是 MS-DOS，它是多工的。像這樣的工作便不應該讓它在前景工作中執行，而必需交給背景工作去執行。而前景工作則可繼續正常運作，不須做任何等待。

背景工作（background jobs）

要將一個指令放到背景工作去執行，其實非常簡單。只要在指令的最後面加上符號“&”便可。當你 rutern 後會將馬上顯示一個訊息，並且下一個提語（prompt）也會馬上出現。如下例：

```
2 % tar cvf /dev/rst8 /home |& tee tar.tmp >& /dev/null &
[1] 293 294
3 %
```

在上面的指令中你可看到三個“&”符號，只有第三個才將指令放到背景工作中執行（第一個與第二個它代表的義意是標準錯誤訊息，千萬別搞亂了）。在指令執行後所出現的訊息，“[1]”則是代表背景工作的工作號碼；“293”與 “294”則表示該工作在系統中所執行的 PID（process ID）

號碼。這指令的作用在備份目錄“/home”下所有的資料，並將指令 tar 的輸出（含標準錯誤訊息）以指令 tee 存放到 tar.tmp 檔案，並將輸出重導向到“/dev/null”這個無底垃圾箱去，避免輸出訊息干擾到前景工作的進行。

當指令以放到背景工作中執行，我們就不需等待該指令執行完成，便可馬上繼續下指令來查詢 process 的處理情況。我們所得如下：

```
3 % ps
PID TT STAT TIME COMMAND
290 p0 S 0:00 -csh (csh)
293 p0 D 0:00 tar cvf /dev/rst8 /home1
294 p0 S 0:00 tee tar.tmp
295 p0 R 0:00 ps
```

由上的結果顯示該背景工作的 process 在系統中的處理狀態。另外我們可用 C Shell 的內建指令 job 來顯示背景工作的狀態。如下：

```
4 % jobs
[1] + Running tar cvf /dev/rst8 /home1 |& tee tar.tmp >& /dev/null
```

指令 jobs 的訊息：“[1]”代表背景工作號碼，“+”符號代表“current job”，如果是出現符號“-”則代表“previous job”。再來便是背景工作的執行情況“Running”，最後則是背景工作的指令。當指令在背景工作中執行時，當它正常執行完畢後會產生訊息，告訴你背景工作已經執行完畢。如下指令 5 後的第二行訊息便是。

```
5 % pwd
/home1/akira
[1] Done tar cvf /dev/rst8 /home1 |& tee tar.tmp >& /dev/null
6 %
```

當然背景工作不是止能執行一個而以，你可連續將各種工作用符號“&”將它們放到背景工作內執行。

背景工作的控制管理

當我們將數個工作放到背景工作執行時，我們要如何去管理這些背景工作呢？又如何知道背景工作的執行情況？如何終止或暫時停止背景工作的執行？不要急先讓我們來丟幾個指令到背景工作去執行，再來逐一說明：

```
6 % du -a /user > user.data &
[1] 237
7 % find / -name core -type f -ls > core.data &
[2] 238
8 % grep '^[^:]*::' /etc/passwd > nopasswd &
[3] 239
```

如上我們將 3 個指令放到背景（background）中執行。用指令 jobs -l 顯示背景工作的執行情況如下：

```
% jobs -l
[1] + 237 Running du -a /user > user.data
[2] - 238 Running find / -name core -type f -ls > core.data
[3] 239 Running grep '^[^:]*::' /etc/passwd > nopasswd
```

首先我們爲你介紹一個工作控制特有的名辭：current job。再上例中的 current job 是相信讀者一定能一眼看出是“du -a /user > user.data”，也就是背景工作號碼“[1]”。如果當第一個背景工作順利執行完畢，第二個與第三個背景工作均還在執行中時，current job 便會自動變成是背景工作號碼“[2]”的背景工作。所以 current job 是會變動的。當你下 fg、bg、stop 等指令時，如果不加任何引數則所變動的均是 current job，關於這點在下指令時千萬得注意。同時代表 current job 的特殊符號也有是多個如“%”“%+”或“%%”，選一個習慣的運用便可。

管理背景工作處理程序

終止背景工作

指令總是有下錯或下了而又發覺不必要了的時候，當這樣的情況，當這樣的指令是產生在已執行的背景工作時，你可用 C Shell 的內建指令 kill來終結它，而且還有很多種方式呢。在例子中，假如你想要“終結”背景工作“du -a /user > user.data”。見下面幾種指令 kill 的用法：

內建指令 kill 使用語法：	
kill pid_number	將某個號碼的處理程序終止執行
kill %	終止執行 current job
kill %+	同上
kill %-	終止執行 previous job
kill %m	終止執行第 m 個背景工作
kill %string	搜尋以 string 爲開頭的背景工作並將它終止執行（爲避免錯誤，此法少用）

瞭解上述的情況後，要終止第一個背景工作就很簡單了。隨便以底下的每一行指令均可將它終止。

```
% kill 237 （237爲第一個背景工作的PID號碼）
% kill %1
```

如果想終止第二個背景工作，以下兩種方法均可。

```
% kill %2
% kill %-
```

前景、背景工作的停止及再繼續執行

有些時候在工作中，手邊正做到一半的事，時常會被打斷，要求你先處理別的事。譬如你正用 vi 編輯某個檔案或用指令 find 在整理檔案系統的過時檔案，你的主管十萬火急地跑過來，馬上要你處理一份資料，因爲他的主管馬上要，人又坐在你旁邊，好罷！把正在處理的工作先佔停下來，先做他的吧。請想想，你是如何暫時停止你的工作的？在這個時候，我們可使用 control-z 來暫停工作。且我們會得到一個訊息，如下：

```
Stopped
20 %
```

暫停的工作已被 C Shell 放入背景工作中了，且保留原狀態停止執行。這時便可以馬上做你主管插入的高優先工作，假設如下：

```
20 % cd ~/acct
21 % lpr mooon.acct
```

這時候你雖然把工作目錄更改了，但其實並不要緊。你可以先用 C Shell 的內建指令 jobs 來顯示剛才被暫停的工作是第幾個背景工作，假設你就只有那一個背景工作，則你將見到類似下列的訊息：

```
22 % jobs
[1] + Stopped vi home.data
```

這時你可以不用顧慮工作目錄改變的問題，因爲 C Shell 的工作控制會幫你處理的，你只管用 C Shell 的內建指令 fg 來把背景工作切換到前景來執行便可。如下：

```
23 % fg （註：指令後不加引數既代表current job）
vi home.data (wd: ~/test)
```

上面訊息“(wd: ~/test)”便是告訴你 vi 的工作目錄，請放心它絕對與你當初的工作目錄相同。這訊息出現的時間很短，馬上便回到你原來的 vi 模式中，當你編輯完後，存檔後會出現下面的訊息：

```
(wd now: /home1/acct)
24 %
```

工作目錄又回到你所改變的目錄去了。自己好好試一試，碰上時可幫你解決不少困擾。

以上爲你介紹的是暫停前景工作，接下來我們來談談如何暫停背景工作的執行。 C shell 的內建指令 stop 可用來暫停背景工作。語法如下：

stop 指令使用語法：	
stop %	停止第一個背景工作執行
stop %n	停止第 n 個背景工作執行

瞭解指令 stop 的語法後，你便可將想要停止的背景工作暫時停止執行，譬如你要將一份整年度的月報表用 nroff 指令整理，因爲資料量龐大，所以你將它放到背景中執行。指令如下示：

```
% nroff -ms moonth[1-12].acct > year93.acct &
[4] 240
```

當你處理到一半的時候發覺 moonth12.acct 檔案是舊的需要更新，這時你用 jobs 指令查看到背景工作“[4]”還在執行中，但無法知道指令以處理到那一個月份時，你可馬上下 stop 指令，將背景工作“[4]”先暫時停止執行再說：

```
% stop %4
```

然後馬上查看檔案 year93.acct 的尾部判斷是否以處理到第 12 月份，如果還沒有執行到，馬上將 moonth12.acct 檔案資料更新。再用指令 bg 來將已被暫時停止執行的背景工作“[4]”再接著繼續執行下去。這樣非但可更正錯誤檔案，又不用將已處理完的工作放棄，重新重頭再執行一次。可說是一舉數得。這是工作控制中最令人“感激”的一項功德無量的功能。千萬不可不知。

指令 stty 設定的影響

在一般的環境下，背景工作的輸出均會直接出現在螢幕上，如果想要讓背景工作的輸出不直接插入到前景工作中，用指令 stty 來設定可解決這項問題。指令 stty 有一項參數叫“tostop”它的作用是將背景工作的輸出不直接輸出到前景去，只送出訊息告訴前景說某背景工作有輸出訊息。並且它會將那個背景工作先暫時停止執行。當你知道某背景工作有訊息輸出時，可用指令 fg 來將該背景工作叫回前景觀看其輸出訊息，同時該工作將會繼續執行。如果你想再將這個工作送回背景去執行，你可以用 control-z 先暫停，再用 bg 將該工作放入背景中繼續執行。這種運用方式僅適用於輸出訊息少的工作，訊息輸出量大且頻繁的工作可能就不適合。因爲當一有訊息要輸出，該工作便會停止執行，等待你去查看後才會再繼續執行，前景、後景這樣切換多次會叫人受不了的。讓我們來看一下實際的例子：

如果你不知 stty 指令現在的設定，可用鍵入 stty 來顯示設定值，如下：

```
% stty
speed 38400 baud; evenp
-inpck imaxbel -tabs
iexten crt
```

顯示的參數中可看到並無“tostop”，於是下指令來設定它：

```
% stty tostop
% stty
speed 38400 baud; evenp
-inpck imaxbel -tabs
iexten crt tostop
```

在這樣的環境設定下，我們將一個指令放入背景中執行，如下：

```
% find / -name Cshrc -ls &
```

當背景工作找到檔案時，會在你前景出現以下的訊息：

```
[1] + Stopped (tty output) find / -name Cshrc -ls
```

這時你便可用指令 fg 將該背景工作叫回前景，便可看見輸出內容，如下示：

```
% fg
find / -name Cshrc -ls
24255 3 -rw-r--r-- 1 bin staff 2897 Jul 24 1992 /usr/lib/Cshrc
```

此時你可以將它 control-z 暫停，再以指令 bg 送回背景中執行。就可在前景繼續工作，並等待下一個輸出訊息出現。

如果你想將 “tostop” 這個參數去掉，指令語法如下：

```
% stty -tostop
% stty
speed 38400 baud; evenp
-inpck imaxbel -tabs
iexten crt
```

指令 notify 與預設變數 $notify 的運用

C shell 的工作控制對於背景工作的執行結果通知，通常是在前景中等待你任何一次 return 的時候，順便將訊息輸出到螢幕上。譬如當你在前景中用 vi 在 編輯檔案時，某個背景工作的執行結束了，這項訊息會一直等待你編輯完檔案退出 vi 模式後，才有機會將會訊息輸出到螢幕上。這個好處是不影響你編輯檔案，但也可能延誤你對需最優先工作的處理時效。如果你要改變這項工作控制的執行特性， C Shell 提供了內建指令 notify 與預設變數 $notify 來供你運用。先來看指令 notify 的用法。比方說你把一個指令放到背景中執行，如下：

```
% cc -o backup backup.c &
```

當指令放入背景中執行時，如果你想讓它在執行結束時馬上通知你，你可運用內建指令 notify 來做到這點，如下鍵入指令：

```
% notify
```

這時指令不加引數是因爲該背景工作是 current job，如果不是 current job，你下指令時便得加背景工作號碼的引數，如下：

% notify %3 （假設背景工作號碼爲3）

當你下完指令 notify 之後，便去做另一個程式的編輯工作，當該背景工作執行完畢時，如果你還未離開 vi 模式，輸出訊息便會插入螢幕中，插入的訊息僅是通知你背景工作的情況，並不會加入你的 vi buffer 中，這點請放一百個心。訊息如下：

[1] Done cc -o backup backup.c

以上是使用指令的方式，如果使用模式比較適合你的使用習慣的話，建議你不妨改用設定預設變數 $notify 的方式，將下行加入“~/.cshrc”檔案中，則你每個背景工作均會以此方式通知你其執行情況。

set notify

關於背景工作使用的注意事項

資源使用限制（limit指令）

如果在你的使用環境中，有運用 C Shell 的內建指令 limit 來限制你的系統資源的使用時，對一項執行時間會較長的背景工作而言可能會比較不利。如果你想查詢資源使用限制可使用 C Shell 內建指令 limit，如下：

```
% limit
cputime 10:00
filesize unlimited
datasize 524280 kbytes
stacksize 8192 kbytes
coredumpsize unlimited
memoryuse unlimited
descriptors 64
```

這時你可清楚地看到各項系統資源的使用限制。如果你要取消某項限制，可用指令unlimit。譬如取消 CPU 的限制，方法如下：

% unlimit cputime

但如此運用會影響所有的 process，如果你只是要對放到背景中執行的工作單獨取消的話，你可用我們前面所提到的群體指令來執行該工作。如下例所示：

% (unlimit cputime ; find / -nouser -ls >& nouser.file) &

工作控制與退出的關係（指令nohup）

在正常情況下， 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

爲 UNIX 系統會隨著 logout 送出 hangup 的訊號將你所有的 process 終止。但在背景工作則會隨著你使用的 UNIX 系統不同而有所差異。有些 UNIX 系統對於執行背景工作會主動以 nohup 的方式執行，它可讓背景工作不受 logout 的影響而終止執行。但如果你的系統是屬於不主動爲你的背景工作加上nohup 的話，則你一但 logout 則背景工作一樣逃不過被 hangup 訊號終止的命運。當然你也不可能得到任何的結果了。這一點請特別特別注意，務必在下指令到背景內執行前搞清楚系統的特性。如果發現有必要自己手動的話，下指令的方式如下：

```
% nohup command &
```

C Shell 的內建指令（Built-in Commands）

C shell 的內建指令，它們其實是存在於 C shell 這個程式本身之內。當你的 login shell 被載入記憶體的時候，他們也就隨之存在記憶體中了。所以當 C shell 要執行它們時， C shell 會直接從記憶體中讀取並加以執行，不需要像執行系統指令那樣要經過搜尋檔案後，用 fork() 一個新的處理程序，然後用 exec() 來執行它。所以在執行效率上會快過系統指令。這是內建指令與一般的系統指令最大的不同處。

在 C shell 這個程式中有不少內建的指令，如常用的 cd、kill、echo、exit 便都是 C shell 的內建指令。或者像 alias、history、limit、set、setenv、source、以及關於工作控制（job control）的 fg、bg、stop 等也都是 C shell 的內建指令。甚至於在後面我們將會爲你介紹的控制流程（control flow）功能，也全部是內建指令。

以下我們所要爲你介紹是幾個比較獨立性且非常重要的內建指令。至於在前面我們已經介紹過的 history、alias、控制流程（control flow）功能，在此便不再贅述。至於 C shell 的完整的內建指令請參考附錄。

umask 指令

```
umask （顯示設定值）
umask nnn （設定umask，設定值爲000~777的整數）
```

umask 指令的功能是用來“限定”每一個新增的檔案、目錄的基本使用權限（permission）。譬如說當使用者以編輯器新產生的檔案，或者是從系統的某處拷貝來的新檔案，或者是以輸出重導向的方式產生的新檔案，或是以指令 mkdir 新建的目錄等等，一切新產生的檔案、目錄，它們的最初使用權限，均會受到這個內建指令 umask 的設定值所影響。就是我所說的“限定”。

指令裡的 nnn 所代表的意義與 chmod 指令的 nnn 相似。不同的是 chmod 指令 nnn 是“給於”使用或者是將要改變的許可權限，而 umask 則是“取消”nnn 的使用許可權限。這點是根本性的差異，使用者必須分清楚。

指令 umask 的設定值以三個八進位的數字“nnn”代表。第一個設定數字給使用者自己（owner user），第二個則是設定給用使用者所屬的群體（group），第三個給不屬於同群體的其它使用者（other）。每一位數字的設定值都是三項不同權限的數值加總，read 權限數值爲 4；write 權限數值爲 2；execute 權限數值爲 1。結合了前三者的權限數值，單一的數字可設定的範圍是 0 ~ 7；整體的可設定範圍是 000 ~ 777。

要知道設定後會得到什麼結果。原則上，方法很簡單。就是用最大值減去設定值即可得到你想要知道的結果。對目錄而言最大值是 777；對檔案而言，最大值則是 666。這個方法對目錄而言完全正確；但對檔案而言會有無法應付的意外。

以下爲了說明上的方便，我將以實際運用上，不可能會採用的設定值 067 作爲本節例子來加以說明。

譬如當你設定 umask 爲 670 ，使用檔案的最大值 666 減去設定值 670，得到的是數值是 負4，已超出數值的定義範圍變成沒有任何意義的數值。但真正使用者會得到的結果卻是檔案對 other 開放 rw 權限；對 owner 與 group 關閉所有權限。所以使用者無法使用減去的方法來獲得完全正確的結果。

表面上，C shell 讓系統使用者使用 umask 時只需輸入一組 3 個數字的設定值。但是，這組設定值對於目錄及檔案卻有著不同的作用結果。所以農夫我打算更進一步地說明其中運作的細節，讓看官們能完全的理解。

對系統程式而言，內建指令 umask 的設定值實際上是群組化的參數，也就是 S_IRWXU、S_IRWXG、S_IRWXO。代表的群組情況如下：

```
S_IRWXU = S_IRUSR | S_IWUSR | S_IXUSR
S_IRWXG = S_IRGRP | S_IWGRP | S_IXGRP
S_IRWXO = S_IROTH | S_IWOTH | S_IXOTH
```

也就是說 umask 所設定的三個數字，其實包含了九個不同意涵的參數，用來對映九種不同的使用權限，這些參數會被要產生檔案的程式，或者是要產生目錄的程式帶入並執行出結果。一個 C shell 的使用者必須要有能力完全掌握 umask 的設定，並演算出設定後所得到的結果。

系統在產生一個新目錄，會完全使用到上述的九種權限的參數，所以最大值是 777 ，這對 umask 內建指令而言，可以很容易地使用減去權限的方法來獲得正確的結果。但在產生一個新檔案時，就不是如此。

系統在產生一個新檔案時，creat function 只取用 read 與 write 權限相關參數，也就是 access permission bits 裡的 S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, S_IWOTH 來定義產生的檔案應該具有何種程度的權限。由於 read（數值 4） 與 write（數值 2） 的權限相加的結果是 6。所以最大的有效數值爲 666。至於所有的 execute 權限(數值 1)在此被忽略(access permission bits 爲 S_IXUSR, S_IXGRP, S_IXOTH），所以 creat function 在輸出時一律定義爲 0，也就是無執行的權限。會對檔案如此限制的理由其實很容易理解。因爲幾乎不可能有一個使用者他的所有檔案都絕對必要被固定成爲給予 execute 權限(檔案包含的型態相當多，如文字檔、資料檔、圖形檔、執行檔等等)，如果功能被如此設定的話，反而會造成相當多的系統漏洞，所以必須管制成必要時再由使用者自己來打開 execute 權限（這就是爲什麼每當你新編輯完成的 C

Shell script ，還要用指令 chmod 來加上可執行使用權限才能執行的原因）。

明瞭上述的原因之後，看官們您能理解，指令 umask 的設定值裡面，如果包含了 excute 權限，又運算產生檔案的使用權限時，應該要先減去（或者應該稱爲 disable），才來作運算。所以當設定值爲 670，其中是第二位數字是奇數，明顯的包含了 S_EXGRP 的數值，所以先減去 1，所已有效設定爲 660。此時使用最大值 666 減去 660，得到的便是正確的檔案使用權限。

農夫我使用最大值減去設定值的方式來說明，對一般的使用者應該能較容易理解與運用。在前一版的文字說明中，我曾經提到計算的方式此用的是 XOR 運算法則，它才是實際上程式的運算法則。也就是設定值被程式拆成參數後的參數演算法則。我想學習過邏輯運算的人都清楚 XOR 的運算方式（如右圖所示）。

Exclusive-OR

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	0

以下我使用設定值 067 當作例子，以 XOR 的演算法則來實際設定會應該得到的結果。

對檔案的演算

Owner	Group	Other	
rwx	rwx	rwx	
111	111	111	777
110	111	000	670
001	000	000	107

對目錄的演算

Owner	Group	Other	
rwx	rwx	rwx	
11-	11-	11-	666
110	111	000	670
000	000	110	006

希望看官們，能理解以上的說明（農夫我發覺改寫後的版本，看起來實在相當囉唆）。以下提供一些比較常用的設定值供看官們參考：

檔案權限的最大值	設定值	結果	代表的使用權限
666	002	664	rw-rw-r--
666	022	644	rw-r--r--
666	037	640	rw-r-----

666	077	600	rw-------
目錄權限的最大值	設定值	結果	代表的使用權限
777	002	775	rwxrwxr-x
777	022	755	rwxr-xr-x
777	037	740	rwxr-----
777	077	700	rwx------

順便附上單一設定值與檔案及目錄的互動對照表，如下：

umask 設定值	檔案使用許可權	目錄使用許可權
0	rw-	rwx
1	rw-	rw-
2	r--	r-x
3	r--	r--
4	-w-	-wx
5	-w-	r--
6	---	--x
7	---	---

假如使用者想要顯示 umask 的設定值，可鍵入指令 umask，設定值即會顯示出：

```
% umask
22
```

以上所顯示的設定值“22”即代表“022”。因第一個數字爲“0”時不顯示。假如顯示值爲“2”則代表“002”，顯示值爲“0”則代表“000”。一般系統的常用的設定值有“002”、“022”、“037”或“077”等幾種。

接著我們實際來設定指令並觀看其使用結果：

```
5 % umask 022 ; umask
22
6 % ls -l > aa ; ls -l aa
1 -rw-r--r-- 1 akira 61 Aug 31 11:32 aa
7 % mkdir dd ; ls -l
total 2
1 -rw-r--r-- 1 akira 61 Aug 31 11:32 aa
1 drwxr-xr-x 2 akira 512 Aug 31 11:33 dd/
```

一般而言如果使用者要自行設定或更改這個指令的設定值，最好的方式是將這個指令放在“~/.cshrc”檔案中，讓 C Shell 來爲你執行。如果 login 後還有須要更動，可直接在指令行模式下鍵入指令重新設定之。如果使用者不自行設定則系統會給於系統的設定值，一般均爲“022”。

exit [status value] 指令

這個指令會終止 C shell 的執行，並退出該 C shell。如果你 login shell 中執行，則功能便相當於 logout。如果在 subshell 中執行則回到其 parent shell，並且可以在退出 subshell 時，給於一個執行狀態變數的參數值。在不指定的情況下退出的執行狀態變數爲“0”。這個功能在 C Shell 程式設計時，常用來設定程式不正常結束的狀態值，以供我們做執行狀態的查詢使用，是一個非常有用的功能。請讀者特別注意之。

下例中，我們在 login shell 中下指令 csh，其作用便是產生一個 subshell，如此一來我們的執行環境便由 login shell 轉移到 subshell 中了。然後我們執行指令“exit 1” 來結束這個 subshell 的執行，並且設定 $status 變數的值爲“1”。當我們回到 login shell 中以指令“echo $status”顯示該 subshell 的執行狀態時，所得的結果便是我們執行指令 exit 時所傳回的設定值“1”。在最後使用指令 exit，我們將會退出系統。

```
7 % csh
8 % ps
PID TT STAT TIME COMMAND
293 p0 S 0:01 -csh (csh)
361 p0 S 0:00 -sh (csh)
362 p0 R 0:00 ps
1 % exit 1
9 % echo $status
1
10 % ps
PID TT STAT TIME COMMAND
293 p0 S 0:01 -csh (csh)
364 p0 R 0:00 ps
11 % exit
```

source [-h] filename 指令

[-h] 選項 將所讀取的指令列入過去指令使用記錄（history list）中，但並不執行所讀入的指令。

source 指令能從指定的檔案中讀取指令來執行，常用以執行修改過後的特殊檔案，如“.cshrc”、“.login”檔案等。比方你以 vi 指令更改“.cshrc”的 path 變數後，要如何來“執行”呢？你必須使用 source 這個內建指令來執行它。如下：

```
% source ~/.cshrc
```

執行後 path 變數便是你所更新的設定值。這可是一個相當重要的內建指令。

在使用 source 指令來讀取檔案的執行過程中，請注意一個特殊情況。就是一但產生指令無法執行或產生錯誤時，則執行的動作將會在該指令行被終止，未執行部份將不再執行。關於這種情況，我們用指令 source 來執行一個分離的別名檔案來做說明：

假定檔案“.aliases”內容爲下：

```
2 % cat -n .aliases
1 alias rmr rm -r
2 alias cd 'cd \!*;set prompt = "\! <$cwd>
3 alias vicsh 'vi \!:1 ; chmod u+x \!:1'
4 alias lsa ls -asF
3 % source .aliases
Unmatched '.
4 % alias
rmr rm -r
5 %
```

當我們以執行指令“source .aliases”時，產生錯誤訊息“Unmatched '.”。表示檔案“.aliases”中有不合語法的 alias 設定，產生了無法執行的情況。此時我們用指令 alias 來看我們執行成功的別名時，發現僅第一行設定成功，而第二至第四行均沒有被設定。這是因爲我們用指令 source 執行檔案“.aliases”時，讀取檔案的第二行要執行時，產生了的語法錯誤，指令 source 於是便停止以下各行指令的“執行”所造成的結果。當然第二行以後便不會執行讀取的動作了。

此外在不加選項情況下使用 source 指令，執行時所讀取執行的指令並不會加入 history list 中。如果有需要加入 history list 中，必需加選項 -h。但加上選項 -h 的執行方式與不加選項 -h 時，有相當大的差異。首先是它只讀取整個檔案的所有指令行進入 history list 中，但並不執行指令行。其次是它也不會檢查指令行的語法是否正確。所以在產生像上述的錯誤時，加上選項 -h 的指令 source，依然會繼續讀取下一行直到整個檔案讀取完畢爲止。讓我們再利用上面的檔案“.aliases”來說明：

```
18 % csh -v
1 % alias
alias
```

首先我們執行 csh -v，用意在產生一個可觀看執行情況的 subshell。當執行指令 alias 查看時發現到並沒有任何別名已設定。

```
2 % source -h .aliases
source -h .aliases
alias rmr rm -r
alias cd 'cd \!*;set prompt = "\! <$cwd>
alias vicsh 'vi \!:1 ; chmod u+x \!:1'
alias ls ls -asF
```

此時我們執行指令 source -h .aliases，可明顯得看出檔案內容被全部讀取，而且也沒有錯誤訊息產生了。

```
7 % alias
alias
8 % history ; exit
history ; exit
```

```
1 alias
2 source -h .aliases
3 alias rmr rm -r
4 alias cd 'cd \!*;set prompt = "\! <$cwd>
5 alias vicsh 'vi \!:1 ; chmod u+x \!:1'
6 alias ls ls -asF
7 alias
8 history ; exit
9 % 19 %
```

我們再用指令 alias 查看別名設定，結果依舊是沒有任何設定。然後用指令 history 卻清楚看到所讀取的指令行已列入其中。我們可清楚地了解到 source 指令加上選項 -h 之後的執行情況，與原先的差別是相當大的。在使用上請多加注意。

limit [resource [max-use]]、unlimit [resource] 指令

這兩個內建指令分別是用來設定（limit）或消除（unlimit）系統資源在使用上的限制，限定的系統資源與使用單位如下所示。

可設限的系統資源（resource）種類如下：

cputime 佔用的CPU執行時間
filesize 可使用的單一檔案的最大值
datasize 限制處理的資料上限，包含堆疊（stack）
stacksize 處理的堆疊大小
coredumpsize 核心資料轉存爲檔案之上限
descriptors 檔案描述詞的上限
memoryuse 記憶體使用上限
設限的最大使用值（max-use）單位如下：

nh 單位：小時（僅用於cputime）
nm 百萬位元（Megabytes）或單位：分（僅用於cputime）
mm:ss 分：秒（僅用於cputime）
nk 單位：千位元（kilobytes）此單位爲基本設定值

首先我們可使用指令 limit 來顯示系統現在的設限情況。

```
3 % limit
cputime unlimited
filesize unlimited
datasize 524280 kbytes
stacksize 8192 kbytes
coredumpsize unlimited
memoryuse unlimited
descriptors 64
```

假設要設定 cputime limit 爲 1 分鐘，鍵入指令如下：

```
4 % limit cputime 1
```

當執行指令使用 cputime 超過限制時，指令將自動被終結，並顯示出警告訊息。如下所示：

```
5 % find / -name Cshrc -print
Cputime limit exceeded
```

想要解除 cputime 限制，或者是要加以設定的話，可用 unlimit 指令來處理之。

```
6 % unlimit cputime
```

其它的各項設定方法均與此例相似。

對大部份的系統而言，最常設限的系統資源可能以 cputime 及 filesize 這兩者居多。通常是用以防止不明的或無法預期的錯誤處理程序佔用掉有限的系統資源，如硬碟空間或以秒計費的 CPU 時間(或許在今日這種環境的使用限制事實上已經不多了)。假定你對 cputime 設定限制爲 60 秒，則大部份的處理程序如果執行的 cputime 超過 60 秒，執行會自動終止。但如果你要花費常時間去編輯一個檔案，碰上執行的 cputime 超過 60 秒的限制時，我敢保證你一定會哭出來（或“媽”出來）。所以，有預期這種情況會產生時，建議你先取消設定的 cputime 限制，再執行 vi 的編輯工作，下指令的方式可用下面的方法：

```
% unlimt cputime ; vi long_job ; limit cputime 60
```

如果你想一勞永逸地免除 vi 到一半被這種錯誤所終結的惡夢，你也可設定一個別名來代替 vi 指令，來免除像上面那種使用方法，建議的設定如下：

```
alias viul 'unlimit cputime ; /bin/vi \!:1 ; limit cputime 60'
```

關於 hard limit 的限制

有些 UNIX 作業系統基於實際面與管理層次上的理由，對於 process 允許使用的系統資源是有所限制的。以 stacksize 這項參數爲例，嚴格說來，其可使用的硬體系統資源是有限，她以電腦擁有的實體記憶體以及 Swap 空間大小而定。隨便允許一般使用者做錯誤的參數設定，時常會影響整體系統的穩定性。所以有必要有所管制。也因此所謂的 "hard limit" 因應而生。像現今的 Solaris 8 就有此管制性的功能。

由於有的 "hard limit" 的限制，使得一般的使用者在 unlimit 時有所限制。因此一般使用者的 c shell process 實際的 limit 就被"hard limit" 所取代。而且這項 hard limit ，一般使用者沒有權限修改。只有系統管理者可以經由以下設定取消

```
% unlimit -h stacksize
% unlimit stacksize
```

第一道是取消 hard limit 的限制，第二道才真正重設 stacksize 爲 unlimit

dirs [-l] 指令

此內建指令功能與 pwd 指令相似均是顯示目前的工作目錄， pwd 指令顯示的是絕對路徑，如“/home1/akira/test”，而內建指令 dirs 顯示的工作目錄的表示方式，則比較不同。在 home 以下的工作目錄，用符號“~/dirname”來表示路徑。如果加上選項 [-l] 則與 pwd 指令功能完全相同。使用情況如下：

```
% dirs
~/test
% dirs -l
/home1/akira/test
% cd /bin ; dirs
/bin
```

由於它會以符號“~”來表示路徑的特性，假定說你的 home 目錄相當長，而你想把目錄顯示在題詞中，你也可以使用這個內建指令來設定，請參考以下的例子：

```
% alias cd ' ch \!* && set prompt = " `dirs` % " '
% cd test
~/test % cd /usr/bin
/usr/bin %
```

這樣的設定也不賴吧！

事實上內建指令 dirs 所顯示出來的資料其實是目錄堆疊（directory stack），關於這個目錄堆疊，我們留到下面的 pushd、popd 時再來說明。

echo [-n] 指令

內建指令 echo 是一個寫 C shell 程式的相當常用的指令。通常在指令行模式下亦會使用來顯示變數的內容，如下：

```
6 % set akira = yeats
7 % echo $akira
yeats
8 % echo '$akira'
$akira
```

指令 echo 所提供的選項 [-n] 功能，是將你的游標停在 echo 所顯示出的訊息之後，也就是說選項 [-n] 不在指令結束後加上 NEWLINE 的訊號。這個選項常用於交談式的 C shell 程式上。請注意下面的例子：

```
9 % echo "data in :: "
data in ::
10 % echo -n "data in :: "
```

```
data in :: 11 %
```

指令 10 後的提詞跑到 echo 的內容後面去了，很明顯的與指令 9 有所差別。

time 指令

在今日的 UNIX 系統內事實上存在兩個 time 的指令，一個是系統所提供的 /bin/time，這個 time 指令是 System V 版本所提供的工具，這是爲了 Bourne shell 使用者所寫的工具；而另一個則是以下所要爲你介紹的 C shell 內建指令 – time。內建指令 time 的功能是用來計算指令執行時所使用的各種系統資源的0資料。指令使用的語法爲下：

```
指令語法 time command
```

內建指令 time 後面所接的便是你真正要執行的指令。 System V 版本的 time 指令只能得到三種資料，分別是實耗時間、使用者時間及系統時間。而 C Shell 的內建指令 time 比 System V 的 time 指令能得到更多的訊息。它所顯示的資料一共分成七個部份。讓我們來看下面的例子：

```
% time find / -name Cshrc -ls
24255 3 -rw-r--r-- 1 bin staff 2897 Jul 24 1992 /usr/lib/Cshrc
0.9u 19.5s 0:39 52% 0+372k 1018+143io 599pf+0w

第一部份 “0.9u” 爲使用者時間（user time）
第二部份 “19.5s” 爲系統時間（system time）
第三部份 “0:39” 爲實耗時間（elapsed time）
第四部份 “52%” 是第一部份的使用者時間加上第二部份系統時間後除以第三部份實耗時間的百分比
第五部份 “0+372k” 爲系統的平均分配（shared）記憶體與 unshared 記憶體的大小
第六部份 “1018+143io” 爲輸入與輸出的資料 block 數量
第七部份 “599pf+0w” 則代表 page fault 次數與 swap 的大小。
```

有關於這個 time 內建指令的一些相關的參數意義的詳細說明，我將它們放在第四篇的 C shell 變數的整體介紹內的預設變數的設定影響中，請參閱該部份的說明。

nice [+n | -n] 指令

UNIX 作業系統是一個多人多工的分時（time-sharing）作業系統。所有人的所執行的所有程式均可在系統中同時運作。每一個執行的程式對系統而言都會給於相對映到處理程序（process）。系統以程序定序（process scheduling）的管理方式來安排這些程序，讓它們依序、循環地進到核心程式中執行。這其中的程序運作實際上是非常複雜的，但我們只需要注意到一個觀念，就是程序定序中有設定優先權的處理方式。而內建指令 nice 的作用便是能讓使用者透過它，去調變程序優先權（process priority）中關於計算處理的一項參數。借以控制該程序在同一時間內所能享用的CPU資源的多寡。 nice 內建指令用數字 “-20” 到 “+19” 來代表程序優先權的高低。數字 “-20” 優先權最高，而數字 “+19” 優先權則是最低。一般使用者所能調變的範圍是 “0” 到 “+19” ，固定

的設定值是“0”，也就是一般使用者所能得到的最高程序優先權。而 super-user 所能調變就比一般的使用者爲大，範圍是“-20”到“+19”（這便是我們稱他爲 super-user 的原因之一）。

當我們使用指令 ps -l 來顯示系統的處理程序時， NI 欄爲所代表的數字就是程序優先權的數值。我們先來看一般指令的程序優先權情況，如下：

```
3 % ps -l
F UID PID PPID CP PRI NI SZ RSS WCHAN STAT TT TIME COMMAND
20488201 101 3398 3397 0 15 0 56 376 kernelma S p0 0:00 -csh (csh)
20000001 101 3402 3398 9 27 0 216 456 R p0 0:00 ps -l
```

我們可清楚地看觀察到 NI 欄的數字均爲“0”。現在讓我們降低程序優先權來執行指令 ps -l 看看結果：

```
5 % nice +10 ps -l
F UID PID PPID CP PRI NI SZ RSS WCHAN STAT TT TIME COMMAND
20488201 101 3398 3397 1 15 0 56 216 kernelma S p0 0:00 -csh (csh)
20000001 101 3412 3398 18 49 10 216 452 R N p0 0:00 ps -l
```

看 NI 欄的程序優先權爲“10”，正如我們所下的指令效果。

能妥善地運用調整指令的程序優先權，將有助於提昇系統的使用效率。這點在一個忙的不得了的機器上，善用指令 nice 整體的工作效率將會有所改善。譬如你 Open Windows 的環境下，一面在執行比較高優先的工作的同時，一面在操作一般的系統指令，如果你能適當地將次要的系統指令操作降低程序優先權，就會縮短高優先的工作時間。比較方便的做法是以 nice 指令來執行一個較低優先的 subshell 或者是開一個較低優先 xterm，則你在這個 subshell 或 xterm 之下所操作的任何指令，都會得到較低優先的程序處理權。

C shell 的內建指令 nice 固定的程序優先權數值爲“4”。如下所示：

```
7 % nice ps -l
F UID PID PPID CP PRI NI SZ RSS WCHAN STAT TT TIME COMMAND
20488201 101 3609 3608 0 15 0 56 228 kernelma S p2 0:00 -csh (csh)
20000001 101 3619 3609 19 37 4 216 452 R N p2 0:00 ps -l
```

當你運用 nice 指令時請注意一點，在 nice 指令不接受別名所設定的“指令”，也就是說要用 nice 來下指令時，別名的功能將失去它的效用。請小心使用之。另外與 nice 指令相關的是 renice 指令。 nice 指令是要在下指令時用的，而 renice 指令則是當你下完指令後，才想到要更改程序優先權時使用。一般使用者要使用 renice 指令只能將程序優先權降低，且一但降低後便不能再將它提高。而 super-user 則不受這些限制，並可針對使用者會或者是某個群體程序（process group）或者是某個程序做程序優先權的提高與降低。如果有非常重要的工作急著要提早執行完畢，你可以去求求 super-user 用 renice 來加速一下，你便可知道“電腦特權階級”的滋味！

rehash、unhash、hashstat 指令

這三個內建指令 rehash、unhash、hashstat 在功能上息息相關於一項 C shell 對尋找指令檔案的加速做法。這項加速法叫“internal hash table”。這三個內建指令便是用來做有關於“internal hash table”的資料更新、不使用的設定及顯示“internal hash table”的資料狀態。

C Shell 運用了 internal hash table 的方式，來加速在 path 變數的目錄群中搜尋指令執行的速度。也就是因爲使用了 internal hash table 的來記憶指令的位置，以至於產生了一個使用上必須注意的小小限制。這個限制便是當你新加入一個可執行的程式到 path 變數的目錄中，如果你不將 internal hash table 更新的話，你只能在該程式所在的目錄下執行。當你改變工作目錄到別的目錄中要執行那個新程式，雖然它確實是在 path 變數的搜尋目錄中，在理論上應該會找到它，但其實不是這回事。你執行時將會驚訝地發現“: Command not found.”的錯誤訊息。千萬別以爲電腦又出毛病了，這是正常現像。你在這個時候只要用指令 rehash 將 internal hash table 更新便可以搜尋到該新增的程式了。讓我們來設定一個比較單純的環境來實際試一試：

```
2 % set path = ( /usr/ucb /bin /usr/bin ~/bin)
3 % cd ~/bin
4 % echo ps > test ; chmod u+x test
5 % cd
6 % test
test: Command not found.
7 % rehash
8 % test
PID TT STAT TIME COMMAND
4049 p0 S 0:00 -csh (csh)
4075 p0 S 0:00 /bin/sh /home1/akira/bin/test
4076 p0 R 0:00 ps
```

從上例中，我們清楚地看到指令 6 的執行結果是“test: Command not found.”，而經過指令 7 用 rehash 內建指令將 internal hash table 更新後，新增的執行檔 test 便能被搜尋到，並由執行訊息中我們清楚地看到 test 檔所在的目錄。

最後爲你介紹的是指令 hashstat，它會顯示 internal hash table 的三項統計性資料。如下例：

```
20 % hashstat
15 hits, 5 misses, 75%
```

第一部份是是使用 internal hash table 的有效數值。第二部份是則是使用失效記錄。第三部份是成功的百分比。關於第二部份爲何會產生失效的情況，我們將爲你稍加說明。

當我們鍵入指令後， C shell 經判斷不是內建指令或別名後，便使用 internal hash table 搜尋該指令位置，如果發現沒有該指令，便輸出訊息“xx: Command not found.”。如果說你的 path 變數中沒有“.”（dot）這個符號的話，便不屬於“失效”。因爲失效的意義是指 C shell 有使用到 exec() 這個 system call 爲你執行該指令卻產生錯誤時，才算是“失效”。譬如執行時產生

Permission denied 的情況，便是屬於失效的情況。或許你會感到奇怪，爲何我們在前面所提到的 xx: Command not found.”的情況中，爲何要排除“.”（dot）這個符號呢？因爲它在執行上比較特殊。當 C shell 使用 internal hash table 在搜尋指令位置時，碰到“.”（dot）這個符號便會以 exec() 這個 system call 來執行“./command”。如果執行失敗，將馬上爲你記上一筆。如果你所設定的 path 變數有加上符號“.”（dot），而且還是放在最前面，在此建議你最好考慮將它移到 path 變數的最後面。因爲在你每次使用到系統指令時都會發生一次不必要的錯誤，在無形中將會降低系統的效率。當然最好是放棄在 path 變數使用符號“.”，需要時在以“./command”的方式來執行也可以。

由於 C shell 對於“internal hash table”的使用系統所定義的初始值是使用的狀態，所以當你不想要有上述這樣擾人的情況，要放棄使用 internal hash table 這項功能也可以。 C shell 提供你內建指令 unhash 來解除這個問題。但就整體的使用效率而言，最好還是別輕易放棄使用 internal hash table。因爲當你放棄使用 internal hash table，搜尋指令執行的方式將會改便成最原始的狀態。就是要執行一個指令便逐一的到 path 變數的目錄群 中去嘗試著執行。就如下面的例子：

```
19 % hashstat
10 hits, 0 misses, 100%
20 % unhash
21 % grep akira /etc/passwd
akira:lv8u/EYmXK5kU:101:100:SYMBAD USER:/home1/akira:/bin/csh
22 % hashstat
11 hits, 1 misses, 91%
23 % echo $path
/usr/ucb /bin /usr/bin /home1/akira/bin
24 % rehash
```

在放棄使用 internal hash table 前，指令 hashstat 告訴我們“失效”爲“0”。然後我們鍵入 unhash，放棄使用 internal hash table。接著我們操作一般的系統指令如 grep，在顯示使用情況，卻得到失效”爲“1”，這是爲什麼呢？因爲不使用 hash table後，要執行指令 grep 的動作變成到 path 變數的目錄群中去“嘗試執行”，第一先以“/usr/ucb/grep”執行，發現沒有該指令，所以 exec() 的動作失敗。再來以“/bin/grep”執行，該指令存在， C shell 便爲你執行該指令。這便是放棄使用 internal hash table 的指令執行方式。相當地“原始”是吧！請考慮清楚再使用。如果要恢復使用 internal hash table，鍵入 rehash 便可。

exec 指令

指令語法 exec command

內建指令 exec 的功能與用途是相當特殊的。如果使用 exec 來執行“指令”，在“指令”執行完畢結果輸出之後，原先的 C shell 也會跟著終結。來看下面的例子：

```
6 % exec date
Sat Oct 15 11:29:05 CST 1994
login:
```

當我們在 login shell 中以 exec 來執行指令 date，在指令結果輸出後， login shell也終結。換句話說，就是執行完指令後便自動 logout。因爲以 exec 執行指令時，並不會另外呼叫 fork() 這個 system call，所以不會產生新的處理程序（process）。而 exec 所執行“指令”的處理程序會“佔用”原來呼叫 exec 內建指令的處理程序，而且程序號碼（PID）及環境變數等執行條件均不會改變。就是因爲這個會產生這種“代換”的動作，所以當 exec 所執行的“指令”結束後，便無法再回到原來的執行環境中了。這便是在 login shell 中執行此內建指令後會 logout 的原因。所以千萬別在你的 login shell 中以 exec 來執行指令。如果你還不想 logout 的話！

其實在指令行模式下會使用 exec 來執行指令的可能性是相當小。因爲這個指令大都是運用在“.login”檔案或 C shell 程式設計中。譬如說一個使用者想要對自己的簽入過程加上一些選項，用來執行自己的程式（如備份資料或清理舊資料等）。而且在執行這些程式後，並不希望再進入到系統中。也就是說進入這些特別的選項，執行完特殊的工作後便要離開系統了。在做法上，你便可在“.login”檔案中以這個內建指令 exec 來取代一般的執行方式來達到你所要的特殊執行效果。

當然你也可以利用這個內建指令 exec 的執行特性，在你最後要離開系統前，以它來執行結束前的最後工作，它會爲你自動 logout。這倒也相當方便。

eval 指令

指令語法 eval argument ...

內建指令 eval 的功能是將引數（argument）讀入 C shell 中，然後再加以執行。這個指令，在 C shell script 寫作時，比較會運用到。讓我們先來看下面的情況：

```
6 % set vcom = 'ls -l ; date'
7 % $vcom
; not found
date not found
```

在指令 6，設定變數 vcom 爲 'ls -l ; date'。當我們用變數的形態來執行“$vocm”，卻發現有兩個錯誤訊息，告訴我們“; not found”及“date not found”。會造成這種錯誤的原因，是因爲 C shell 對於這種變數的解析語法，無法辨視特殊符號所造成的。如上例的變數，符號“;”與指令 date 均 C shell 被誤解成是指令 ls -l 後得“檔案名稱”。所以才會有“not found”的訊息傳出。

內建指令 eval 便是用來應付這種情況。我們將上面的變數改用 eval 來執行：

```
8 % eval $vcom
total 1
-r--r--r-- 1 akira 1296 Oct 12 07:29 search.c
Tue Oct 18 12:13:53 CST 1994
```

由於指令 eval 將 $vcom 當成引數讀入再加以執行，所以“not found”的誤解情況便消失了。其

實在作法上使用指令 eval 便相當於以下的用法：

```
9 % echo $vcom | csh
total 1
-r--r--r-- 1 akira 1296 Oct 12 07:29 search.c
Tue Oct 18 12:13:54 CST 1994
```

如果你是在 C shell script 裡運用的話，你也可應用以下的方式：

```
/bin/csh << EOF
$vcom
EOF
```

不過這些變通的方法都不如使用內建指令 eval 來的方便。另外在使用內建指令 eval 上也有相當多的技巧，讓我們來看一個變數互換的技巧：

```
% set a = '$b'
% set b = 'swapping'
% echo $a
$b
%eval echo $a
swapping
```

在上例中，變數 a 的內容是“$b”，當我們以一般的 echo 指令執行時，我們發現僅是將變數的內容顯示螢幕罷了。但是，當我們以指令 eval 來執行該 echo 指令時，卻輸出變數 b 的內容。相信你已經知道要如何運用了吧！

對於這個內建指令，在這裡要告訴你一個比較不好的消息，就是如果你所使用的 UNIX 作業系統是 SUN OS 4.1.3 的話，它有存在不少 bug，使用上請多多小心。

repeat 指令

這個指令相當的特殊（當然也是懶人最常用的用），它的功能是，“重複執行”你所指定的指令。使用的語法如下：

```
repeat 執行次數 執行指令
```

讓我們來看一個例子：

```
% repeat 3 echo "C shell repeat command"
C shell repeat command
C shell repeat command
C shell repeat command
%
```

如果當你真有需要重複執行某個指令很多次時，這個內建指令可好用的很。這種例子並非沒有，譬

如說執行多次才令人安心的指令 sync，便是使用 repeat 的最好時機。

pushd [+n | dir]、popd [+n] 指令

這兩個內建指令在使用上有很大的關連性，所以我們放在一起爲你說明。當你必須在幾個固定的目錄來回地操作時，你便可以用得上這些改善你使用效率的貼心的內建指令。以下讓我們來看一個實際的例子：

例子一

```
% cd ~/project/project-a （改變工作目錄）
... （修改或編輯一些檔案，此部份省略）
% cd /usr/etc/local/bin （改變工作目錄）
... （修改或編輯一些檔案，此部份省略）
% cd ~/project/project-a （改變工作目錄）
... （修改或編輯一些檔案，此部份省略）
% cd /usr/etc/local/bin （改變工作目錄）
... （修改或編輯一些檔案，此部份省略）
```

在上面的例子中我們可以很清楚的看到使用者爲了短暫的工作型態需要，必須到三個不同目錄下去修改一些檔案，而且可能會重複多次。而上面用的是 cd 指令來更改工作目錄，這事實上是相當累人的沒有效率的。 C shell 特別爲了這種情況開發了三個內建指令，也就是 popd、pushd 及 suspend。

這三個指令的使用概念是爲你建立一個目錄堆疊（directory stack），用來供你放置必須常去的工作目錄，以便讓你的更改工作目錄更加方便及有效率。讓我們先來看看以下各個指令使用時所產生的目錄堆疊的資料變化情況：

```
1 % pwd
/home1/akira/project/project-a
2 % pushd ../project-b
~/project/project-b ~/project/project-a (directory stack的內容)
3 % pwd
/home1/akira/project/project-b
4 % pushd /etc
/etc ~/project/project-b ~/project/project-a
```

由上例中在事件 1，工作目錄是在 /home1/akira/project/project-a ，我們請讀者注意到事件 2 也就是內建指令 pushd；它不但兼具cd指令的作用，同時也會將你所鍵入的目錄放入暫存器中，所以當你如事件 2 的方式執行後，所顯示出的訊息便是已被放入暫存器中的目錄，而第一個目錄是現在的工作目錄，其他的目錄在出現的次序上也請加以注意。接下來讓我們用 pushd 指令來重新做一次例子一，看看有什麼效率上的改變。

```
% cd ~/project/project-a （改變工作目錄）
... （修改或編輯一些檔案，此部份省略）
```

```
% pushd /usr/etc/local/bin （改變工作目錄）
/usr/etc/local/bin ~/project/project-a
... （修改或編輯一些檔案，此部份省略）
% pushd （改變工作目錄）
~/project/project-a /usr/etc/local/bin
...
% pushd （改變工作目錄）
/usr/etc/local/bin ~/project/project-a
... （修改或編輯一些檔案，此部份省略）
```

是不是簡單、省事多了呢？對了還記不記得我們提過的內建指令 dirs，它可以爲你顯示目錄堆疊（directory stack）的內容，所以當你放很多目錄在目錄堆疊中時，想要知道目錄堆疊的內容就可以用的上。讓我們來看看下面的例子，順便爲你介紹另一個 C Shell 內建指令 popd。

```
1 % cd ~/project/project-b
...
8 % pushd /usr/etc/local
/usr/etc/local ~/project/project-b
...
19 % pushd /bin
/bin /usr/etc/local ~/project/project-b
...
35 % pushd /etc
/etc /bin /usr/etc/local ~/project/project-b
...
43 % pushd +2
/usr/etc/local ~/project/project-b /etc /bin
...
59 % dirs
/usr/etc/local ~/project/project-b /etc /bin
60 % popd +1
/usr/etc/local /etc /bin
61 % popd
/etc /bin
62 % pwd
/etc
```

在事件 43 中 “ +2 ” 所代表的是目錄堆疊中的第三個目錄，pushd 指令在不加上任何引數時，所代表的便是目錄堆疊中的第一個目錄。內建指令 popd 的功能就如同事件 60 及 61 所顯示出來的，就是將目錄堆疊中的的某個目錄去除掉。事件 60 是除掉目錄堆疊中的第二個目錄；事件 61 是除去目錄堆疊中的第一項資料。但是請注意！當你如事件 61 去除掉第一個目錄堆疊的內容時，你的工作目錄也會自動改變到目錄堆疊中的第二個目錄，因爲這時它已經變成目錄堆疊的第一項了。關於這一點，你可以從事件 61 及 62 看出來。

相信以上的例子中你可能會發覺，當你放入目錄堆疊中的目錄在兩個的情況下，這項功能還稱得上非常好用；但是一但目錄堆疊的內容多了，恐怕就不是人人都能用得順心如意的了。對於這種情況，建議你不妨採用設定 cdpath 變數的方式會來的好些。

某些舊版的 UNIX 作業系統的 directory stack 對於 symbol link 的功能會產生一個小小的

bug，由於這個 bug 會對內建指令 pushd 及 popd 的使用會有影響，使用上應注意到此點，關於這一點 bug 稍後我們會在變數 hardpaths 中還爲你說明

C shell 對其本身在整體的環境控制與部份功能的設定和使用上，都有專屬的變數，提供使用者自己設定與應用。同時在變數的型態上，也區分爲環境變數（environment variables）與預設變數（predefined variables）兩種。前者相當於整體變數，而後者相當於區域變數。以下我們將爲以這兩大分類來爲你分別介紹 C shell 本身所制定的各種變數。

環境變數的設定（environment variables）

環境變數的設定目的在於管理 shell，這是它之所以重要的原因。它的特性相當於整體變數（global variable）。也就是說，你僅需要把環境變數設定在你的“.cshrc”檔案中，由 login shell 所產生的 subshell 或者是執行的 shell 文稿、程式或指令等，均不需再重新設定，便可以直接呼叫或使用該變數。所以環境變數是具有遺傳（inherited）性的。因爲在 UNIX 作業系統中，由一個處理程序（process）會將它全部的環境變數遺傳給它所衍生出的子處理程序（child preoceese）。

譬如你在 login shell 之下執行一個 vi 指令，設定的 TERM 變數會決定使用何種終端機模式，同時 vi 程式本身也會繼承了原來的 login shell 所定義的所有環境變數，所以當你想要在 vi 程式中用指令“:sh”的方式產生一個新的 shell 時， vi 程式還會依據你所定義的 SHELL 變數，產生那個你所指定的 shell 的原因。當然因 vi 程式所產生的 new subshell，依然會繼承來自於 vi 程式的所有環境變數。

C shell 的環境變數全部都是以大寫字母命名。事實上這也是一個不成文的規定。所以當你要自行定義一些環境變數時，請你也能夠這樣做。設定環境變數的使用語法如以下所示：

設定語法 setenv ENVNAME string
解除設定語法 unsetenv variable
顯示所有設定 env

C shell 的環境變數並不多，僅有基本且重要的特殊資訊才被列入。如使用者的簽入目錄（login directory），存放郵件的目錄，終端機的模式，執行指令依據的搜尋路徑等。在這些環境變數中，部份會由系統依據某些特殊檔案內的資料，爲使用者自動設定初始值。如 HOME 變數以及 USER 變數（有些 UNIX 版本不叫做 USER 變數，改稱爲 LOGNAME 變數）的初設值便是來自於“/etc/passwd”檔案。又如 TERM 變數初始值是來自於檔案“/etc/ttytab”。除此之外，環境變數中的 HOME, PATH, MAIL, TERM 等，還會將它們的內容拷貝到相同名稱的預設變數中，以做爲預設變數的初始值。不僅如此，這兩者之間還保有一種互動的關係，也就是其中的任何一方有改變，另一方變數也會自動地將變數內容更新。這些都是環境變數的特點。以下讓我逐一地爲你介紹每一個 C shell 的環境變數。

C shell 對其本身在整體的環境控制與部份功能的設定和使用上，都有專屬的變數，提供使用者自己設定與應用。同時在變數的型態上，也區分爲環境變數（environment variables）與預設變數（predefined variables）兩種。前者相當於整體變數，而後者相當於區域變數。以下我們將爲以這兩大分類來爲你分別介紹 C shell 本身所制定的各種變數。

PATH 環境變數

環境變數 PATH 的初使值來自於系統設定值（各版本的 UNIX 作業系統均有差異）。設定的語法如下所示：

```
setenv PATH /usr/ucb:/bin:/usr/bin:~/akbin:
```

設定的目錄必須用符號“：”來加以區隔開來。在上例的最後的符號“：”，它的後面沒有填上任何的目錄，這樣在語法上並沒有錯，因爲“空乏”在這裡所代表的意義是“目前的工作目錄”，相當於在設定預設變數中所使用的符號“.”。使用時請注意此點。

當設定環境變數 $PATH 後，預設變數 $path 會隨之更改。反之改變 $path 設定， $PATH 亦會自動改變。

```
5 % echo $PATH
/usr/ucb:/bin:/usr/bin:
6 % echo $path
/usr/ucb /bin /usr/bin .
7 % set path = ( ${path} ~/{akbin,project} )
8 % echo $path
/usr/ucb /bin /usr/bin . /home1/akira/akbin /home1/akira/project
9 % echo $PATH
.:/usr/ucb:/bin:/usr/bin::/home1/akira/akbin:/home1/akira/project
```

因爲有這種互動的關係，所以在“.cshrc”檔案中只需要設定其中的一個便可以了。通常的選擇是設定預設變數 $path。

環境變數 HOME 與預設變數 home

一個使用者在簽入後，環境變數 HOME 的初使值來自 /etc/passwd 檔案中的，並將設定值拷貝給預設變數 home，當成是 $home 的初始值。所以對一般的使用者而言，這兩個相關的變數都不需要特別去設定它們。

假使有一天，你因爲執行計畫的需求，想將原 home 目錄 /home1/akira 改到目錄 /home1/akira/project，最簡單的作法是在“.cshrc”檔案中加入下行：

```
set home = /home1/akira/project
```

你或許會感到奇怪，爲何設定是預設變數，而不是環境變數呢？這個原因是，當 login shell 產生之後，再去做修改環境變數 HOME 的動作時，預設變數 home 已經不會隨著它改變了。所以會造成環境變數 HOME 與預設變數 home 所分別設定的 home 目錄不同的情況。這樣便會造成很多問題。因爲由 login shell 所產生的 subshell 或執行任何程式，它們所繼承的是 HOME 變數的設定值，所以他們的 home 目錄爲 /home1/akira/project。而 login shell 的 home 目錄則是爲

/home1/akira。這便是我們不採用設定環境變數的原因。

當你在 login shell 中再重新設定 home 的預設變數，它會將設定傳給環境變數 HOME。如此兩個變數的設定值才會一致。自然便不會產生上述的問題。

另外我們在此要爲你釐清一項重要的觀念，那就是 C shell 用來代表 home 目錄的符號“~”，它實際上所代表 home 目錄是來自於預設變數 home 的設定值。而與環境變數 HOME 一點關係也沒有。讓我們來看下面的例子：

```
1 % setenv HOME /home1/akira/project ; echo $HOME
/home1/akira/project
2 % echo $home
/home1/akira
3 % cd ~ ; pwd
/home1/akira
```

由指令 1 到指令 2，我們可清楚地看到兩個變數的設定值已經不同，指令 3 則明顯地看出特殊符號“~”的設定值和預設變數 home 是相同。這樣可夠清楚了吧！所以說，更改 home 目錄對使用環境而言是一件非常非常重大的大事。使用者在未能真正地全盤性掌握自己的所有環境設定之前，最好不要輕易去更動它。如果真有需要更改，除了上述的情況之外，關於 C shell 的各種起始檔案與所有的特殊起始檔案，最好也將它們拷貝一份或者是使用連結的方式，將他們放一份在新的 home 目錄下，這樣會比較安全些。想要“搬家”，請千萬小心！！

環境變數 SHELL 與預設變數 shell

此變數的系統初始值便是 /bin/csh，此變數在用途上並不廣泛，所以一般的使用者也不太會注意到這個變數的設定。此變數的所定義的 shell，是設定給 UNIX 系統的部份公用程式（如 vi 、 ex 或 mail 等指令），在程式操作中需要產生 subshell 時，會依據這個變數所定義的 shell，產生你所設定的 subshell。在變數的設定上，你所指定的 shell 必須使用絕對路徑。如果你在使用上，有必要改變系統設定值的話，設定的語法如下所示：

```
% setenv SHELL /bin/sh （如果是其他的 shell 也一樣必需使用絕對路徑來設定）
```

此環境變數 SHELL 的特性和環境變數 HOME 相同，就是 login 以後再更此變數的話，並不會同步更改預設變數 shell。而預設變數 shell 的設定語法如下：

```
% set shell /bin/sh
```

如果沒有特殊的因素或需要，此變數請保持系統的設定值。

LOGNAME 與 USER 環境變數

LOGNAME 環境變數爲簽入者的使用者名字，USER 環境變數則爲現在使用者名字。

```
1 % echo "$LOGNAME $USER"
akira akira
2 % su akk
lee% echo "$LOGNAME $USER"
akira akk
```

由以上的例子中，相信你一定能夠清楚地認清，這變數兩兄弟的在定義上真正的區別了吧。如果想要運用這兩個環境變數在 C shell 文稿中，請確實注意它們之間的差異性。

環境變數 MAIL 與預設變數 mail

環境變數 MAIL 所設定的參數是一個絕對路徑的 mail 檔名。設定值便是提供給指令 mail 作爲該讀取那個 mail file 的依據。這個環境變數你沒有設定，系統會自動給一個系統的設定值。系統的設定值如下：

```
setenv MAIL /usr/spool/mail/$USER
```

在 C shell 中與 mail 功能相關的變數，除了 MAIL 環境變數之外，還有一個預設變數 mail 。事實上這個預設變數 mail，使用者而言是比較重要的。因爲這個變數的作用是爲使用者檢查是否有新的 mail 傳入。此預設變數的設定方法如下：

```
使用語法 set mail = ( /user/spool/mail/akira )
set mail = ( 60 /usr/spool/mail/akira )
```

預設變數 mail 的設定方式有兩種，一個是僅指定絕對路徑的 mail 檔名；另一種則是，設定搜尋時間及絕對路徑的 mail 檔名（時間的單位爲秒）。如果你沒有加上時間的參數的話，則系統會給予“10分鐘”設定值，也就是相當於你設定爲 600 秒。另外指定的mail檔案的數量有並不限定只能有一個，可以是一個以上。完全視你所需而定。以下是一個設定的例子：

```
set mail = ( 60 /usr/spool/mail/akira /usr/spool/uucp/akira )
```

最後爲你說明一點，環境變數 MAIL 的設定只會影響“children process”，與預設變數 mail 在設定上並沒有“互動”的關係。也就是說它們的設定值是獨立的不互相影響的。

EXINIT 環境變數

這個變數是專屬於 vi 與 ex 指令所使用，它設定的參數會作爲這兩個指令的環境初始設定。設定的語法如下：

setenv EXINIT 'set 選項'

由於此變數的選項超過 40 個以上，而且都是關於編輯器方面的參數，所以在此並不準備爲你做詳盡的介紹，僅列出選項的資料供你參考運用，如下表：

EXINIT變數的設定選項一覽表		
Noautoindent	Number	noslowopen
autoprint	Nonovice	nosourceany
noautowrite	Nooptimize	tabstop=8
nobeautify	Paragraphs	taglength=0
directory=/var/tmp	Prompt	tags=tags /usr/lib/tags
noedcompatible	Noreadonly	tagstack
noerrorbells	Redraw	term=vt220
flash	Remap	noterse
hardtabs=8	report=5	timeout
noignorecase	scroll=11	ttytype=vt220
nolisp	Sections=	warn
nolist	shell=/bin/csh	window=23
magic	Shiftwidth=8	wrapscan
mesg	Noshowmatch	wrapmargin=0
nomodeline	Noshowmode	nowriteany

關於這項設定資料，你可以在 vi 模式下，用“:set all”的方式顯示所有的選項設定情況。（本項設定資料會因爲 UNIX 作業系統的不同而有所差異，但不會相差太大）

以下我們來舉個較常使用的設定值，供你參考：

% setenv EXINIT 'set nu ai sm sw=8'

上例中我們設定了四個選項，nu=number, ai=autoindent, sm=showmatch, sw=shiftwidth=8。

事實上，在你啓動 vi 編輯器， vi 程式會先在你的 home 目錄下找尋一個叫“.exrc”的特殊檔案。這個特殊檔案作用就是用來設定這些選項，同時它還可以做類似於 aliases 功能的動作，叫做“map”。如果你要設定得相當繁雜的話，你可以考慮放棄設定這個變數。並將你想要設定的所有選項編輯到“.exrc”檔案中。或許還比較適合些呢！提供你作參考。

TERM 環境變數

這個環境變數與 vi 編輯器、more 指令及相關於螢幕游標位置的指令有相當大的關係。一般設定

在“.login”檔案中。設定的語法爲：

設定語法 setenv TERM 螢幕模式
例子 setenv TERM vt220

這個變數對於在“console”操作的使用者而言，可能會忽略。但是對使用個人電腦或者是終端機的使用者來說，它的影響就比較大了。

預設變數的設定影響（predefined variables）

在 C shell 本身所制定的兩種變數當中，真正用來控制 C shell 功能的是以下我們所要爲你介紹的這些預設變數。制定這些變數關係到指令的執行、hisroty 功能的使用、 指令 time 的顯示、job control 的顯示、指令行模式下的資訊顯示、wildcard 功能的使用、退出 C shell 的設定、filename completion 功能的使用、輸出/輸入重導向功能的重寫保護等 C shell 的特殊功能。所以對一個使用者或對於 C shell 的使用而言，這些變數是相當重要的。因爲在上述的特殊功能中，有一小部份功能如果不自行設定，系統的初始設定值是不使用的，關於此點請讀者稍加注意。

C shell 的預設變數在設定上爲了要和環境變數有明顯的區別，一般的習慣均使用小寫字母來做爲變數名稱。各項設定的語法如下所示：

設定語法 set variable
set variable = string
解除設定語法 unset variable
顯示所有設定 set

C shell 的預設變數一般均集中設定在“.cshrc”檔案中。爲什麼要集中到“.cshrc”檔案中呢？因爲預設變數的特性是局部性的，在實際上當你在設定這些變數之後，這些預設變數的設定狀態，並不像環境變數那樣，會將設定值“遺傳”給在它之下所產生的 subshell。如果你想要每個 C shell 及它的 subshell 均要保有你想要的預設變數的設定狀態，你便應該將它設定在 C shell 的起始檔案“.cshrc”中。因爲 C shell 本身的在產生 subshell 時，會自動地去讀取“.cshrc”檔案，以該檔案內的設定做爲 subshell 的初始環境設定。所以，你想要的各項預設變數的設定，爲了要能做到，自login shell 到所有的 subshell 均能完全一致，最佳的方式便是將它們設定在“.cshrc”檔案中。如果是臨時性的、短暫性的預設變數設定，當然還是以在指令行模式下手動設定方式比較適合。所以這類的變數在設定上，請務必留意使用上的需要，來選擇適合的、正確的設定方式。

以下便讓我們來逐一介紹 C shell 的各個預設變數。

path 指令搜尋路徑變數

```
set path = (/usr/ucb /bin /usr/bin ~/bin .)
```

paht 變數所定義的路徑次序便是執行指令時搜尋次序的依據。如果不自行設定，則大部份的 UNIX 系統均自行設定爲（. /usr/ucb /bin /usr/bin）。其中符號“.”代表“目前所在的工作目錄（current directory）”。將目前所在的工作目錄放在其他目錄之前所得到的結果是，在目前所在的工作目錄下所有可執行的程式優先執行。此法乃用以解決自行開發的程式名稱與系統指令相同時，在執行上的問題。但對指令搜尋執行的效率而言，並不理想。因爲對你每次執行的系統指令都會先對目前的工作目錄作搜尋再往 /usr/ucb 等目錄去尋找系統指令。這種設定次序對整體指令的使用效率是較

差的。在自行設定時請注意此點，儘量將你的路徑安排的有效率一點。

在設定這個變數時，有些軟體或者是使用者，可能會因爲某種理由而採用變數 path 加到自己本身的設定中。如果你在“.cshrc”檔案中，發現有類似下面你所看見的 path 設定方式，則請你特別小心：

```
set path = ( $path ~/project )
```

這樣的設定方式在語法上是沒有問題的。設定變數 path 的路徑是原變數的內容加上一個新的路徑“~/project”。這樣的設定方式或許在設定上相當好用，但在實際的運用上卻也有可能造成非常不良的連鎖效應出現。比如，你修改“.cshrc”檔案之後要使用內建指令 source 做更新設定的這個動作來說，原變數的內容會再一次被帶入設定的中，造成路徑“~/project”重復設定，影響到指令搜尋路徑的精簡。如果你重複做上許多次，後果可就不麼美妙了。這種設定的影響在使用上請小心。

cdpath 改變工作目錄搜尋路徑變數

設定這個變數的作用與 path 變數有點相似，不同的是變數 path 是供你找尋指令用的；而 cdpath 變數則是讓你在改變工作目錄時找尋目錄用的。它可以讓我們在任何的目錄下，很容易地到我們想到的工作目錄中。譬如說你會常到自己 home 目錄之下的 akbin 目錄中修改或編輯 C shell 文稿。那麼你便可將 home 目錄設定到變數 cdpath 中。如此一來，不管你是位在那個目錄之下，你都可以當成 akbin 這個目錄便在於目前的工作目錄之下，只要使用“cd akbin”，便可以到該目錄了。請看以下的實際例子：

```
2 % set cdpath = (~)
3 % cd /bin
4 /bin % cd akbin
~/akbin
5 /home1/akira/akbin %
```

覺得這功能如何？有夠好用吧！沒有設定的你趕快設定吧！！他的設定語法和變數 path 完全相同。如果要設定兩個 path 以上，用 space 區格開來便可。不過有一點請注意，就是利用它來找尋目錄時，是依據你所設定的路徑先後次序。萬一碰到相同名稱的目錄時，第一個選擇是目前的工作目錄，再來便是你所設定 cdpath 目錄的次序了。所以在規劃目錄名稱時，最好不要有相同名稱的情況產生。

另外有一點要說明的是，這個變數僅對“cd”“chdir”“pushd”“popd”等指令有作用，其他的指令就沒有任何的效用了。

prompt 提詞變數

C shell 的題詞設定是相當富彈性與變化的，你可以設定的很簡單；也可以設定得什麼資訊全都在

上頭出現，只要你高興。不過基於整個系統的使用效率來衡量它的話，最好別設定的太複雜。花進心思寫個程式來做一個題詞，實在有點小題大作。如果每個人都這樣使用系統的話，總有一天吃虧的還是自己。以下我們僅提供幾種簡單實用的題詞設定，供讀者做爲參考：

```
% set prompt = "\! % "
16 %
```

上例是將 history 的 even 數字加入題詞中。

```
% set prompt = "`hostname`::$user % "
akhost::akira %
```

上例是將 hostname 與使用者名字加入題詞中，此設定很適合使用網路的工作者。

```
% alias cd 'cd \!* ; set prompt = "\! $cwd % " '
%cd
5 /home1/akira % cd /usr/etc
6 /usr/etc %
```

上例相當適合常忘記身在何處的使用者。我們將提詞變數設定到 aliases 中，當每次執行指令 cd 時，便自動再設定一次，結果下兩行所示。

以上的例子均是簡單實用型的題詞，爲了喜歡複雜的讀者，特地提供下面這個較複雜的例子，供你在設定上的一些靈感。

```
alias myprompt 'set prompt = "\\`hostname`::${user}_${cwd}\\\! %" '
alias cd "`chdir` \!* && myprompt"myprompt # run aliases for initial prompt
```

上例的題詞分成三個部份，第一部份是設定提詞的 aliases，第二部份是設定指定 cd，並且將設定題詞的 aliases 加入其中，第三部份是則是執行第一個提詞的 aliases 設定。你可以將這三部份加到“.cshrc”檔案中。以下便是出現的題詞樣本：

```
（空白行）
akhost::akira_/etc/adm/acct
1 % cd ~
（空白行）
akhost::akira_/home1/akira
2 %
```

history 儲存指令使用記錄變數

使用語法 set history = 30

指令使用記錄（history）是 C Shell 的內建指令。如果這個變數沒有設定的話， history 的功能便不會啓動，請使用者加以注意。此變數所設定的數字，就是儲存過去的事件的數目，數目越大雖然更有助於你利用過去的指令，但是相對的越佔記憶體的空間。所以一般的設定值差不多在 30 ~

100 左右。

histchars 指令使用記錄之特殊符號變數

使用語法 set histchars = "!^"

關於 history 功能的特殊符號有二個，系統設定值分別爲“!”及“^”。利用本變數則可改變其特殊符號的原始設定值。

set histchars = "#/"

經過以上設定後，符號“#”取代“!”；符號“/”取代“^”。這將改變你使用 history 的習慣，如下所示：

% set _test = "test aa"

如果你要修改上個指令的 aa 變爲 bb，你必須使用符號“/”，來做修改，如下：

% /aa/bb/

savehist 指令使用記錄檔案儲存變數

使用語法 set savehist = 20

此變數的作用與 history 變數作用相近。不同處是 history 變數所記錄的資料會隨 logout 而消失，而本變數則會在 home 錄下建一個特殊檔案“.history”，供下次 login 時呼叫上回 login 的指令使用記錄；或者是其他的 C shell 使用之。

time 執行時間變數

預設變數 time 與內建指令 time 是息息相關的。本變數是用來定義內建指令 time 的輸出格式及設定自動顯示 time 訊息的基本時間。所以實際上此變數有兩組設定的資料，一個是時間，另一個是輸出格式。設定的組合如下所示：

設定時間 set time = 時間
設定輸出格式 set time = ("輸出格式")
完全設定 set time = (時間 "輸出格式")

時間的單位一般以秒計算之，假定你設定 CPU time 爲 3 秒，則你每個執行時間超過 3 秒的指令，便會自動顯示出執行的時間狀態，而不需用 time 指令來執行命令。

在輸出格式上，總共有十個符號，以下便是每個符號所代表的意義：

%D	佔用系統記憶體的 unshared 平均值（單位：千位元）
%E	實際執行的消耗時間，一般又稱爲 wall clock time（分:秒）
%F	分頁錯誤（page faults）數量。在此我們爲你簡單地介紹分頁錯誤的產生。在 UNIX 作業系統中，記憶體的架構是以頁 page）爲程序移轉的處理單位。所以一個程式或指令的執行過程有可能會因爲 kernal 分頁系統的需要，被多次地載入與載出主記憶體。當處理程序找不到某個使用過的頁次時，便會產生分頁錯誤。這時便會更新使用過的頁次集資料。如果有必要的話，會到硬碟中將需要的頁次讀入。有就是說產生了越多次的分頁錯誤，就有可能會做較多次的 disk I/O 動作。越是如此則執行的效率就越差。所以你必須留意這個項目所輸出的數值。如果常常會有很多的分頁錯誤產生的話，有可能表示你的記憶體已經不夠系統使用了，應該做硬體上的投資了。
%I	程式從硬碟所讀入的資料量（單位：block）
%K	unshared stack 的平均值（單位：千位元）
%M	處理程序執行期間所需的最大記憶體（physical memory） （單位：千位元）
%O	程式輸出到硬碟的資料量（單位：block）
%S	kernal process 的 CPU 使用時間（單位：秒）
%U	user's process 的 CPU 使用時間（單位：秒）
%P	系統使用時間的總和與實際消耗時間的百分比，也就是(%S+%U)/%E 的結果。
%W	swaps 的次數
%X	shared 記憶體的平均值（單位：千位元）

這十個符號你可以依照你的需要設定是否要將它輸出，以及輸出時的格式。以下我們來看一個實例：

```
% set time = ( 5 "Elapsed time: %E System time: %S User time: %U" )
9 % find / -name core -type f -ls
Elapsed time: 1:53 System time: 50.3 User time: 1.8
```

如果你不設定輸出格式的話，內建指令 time 會以 C shell 所定義的輸出格式來輸出訊息。這個格式爲（"%Uu %Ss %E %P% %X+%D %I+%O %F+%W"）。下面是不經過設定的輸出格式：

```
% time find / -name Csherc -ls
（輸出部份省略）
9u 36.1s1:39 38% 0+340k 2859+663io 1250pf+0w
```

echo 與 verbose 指令顯示變數

使用語法 set echo
set verbose

這兩個變數上使用上非常相近，而且在使用及功能上也有互補的作用，所以我們放在一起爲你介紹。設定 echo 變數的作用是將所“真正執行的指令”顯示出來。爲何說是真正執行的指令呢？

因爲在 UNIX 作業系統中有一些“指令”，可能是經過 aliases 功能重新定義過了，所以執行的並不是原來的指令。如果你設定這個變數，你便能很清楚地看到“指令”所執行的內容了。

```
30 ~ % set echo
31 ~ % cd test
cd
set prompt = ! `dirs`%
dirs
32 ~/test % alias cd
alias cd
cd !*;set prompt = "! `dirs`% "
```

其實 C shell 本身也提供一個與此功能相同的選項，就是“-x”。當你下指令來產生一個 subshell 時，如果加上“-x”選項也能得到相同的結果。如下所示：

```
33 % csh -x
1 % cd /bin
cd /bin
set prompt = ! `dirs`%
dirs
2 /bin %
```

verbose 變數的作用與變數 echo 相近。 verbose 變數所處理的並不是指令，而是變數的顯示。如果指令中沒牽涉到變數的問題，它只會將所執行的指令原封不變地顯示在螢幕上。就算是有 aliases 的情況它也不會有任何的作用。

```
2 % set verbose
3 % cd /bin
cd /bin
4 bin %
```

但如果所執行的指令含有變數的話，它的處理情況就和設定echo的顯示有所不同了。讓我們看下面的例子：

```
1 % set vb = 'set echo variable'
2 % set echo
3 % echo $vb
echo set echo variable
set echo variable
4 %
```

以上是設定了變數 echo 時所產生的情況。請注意到指令 3 的輸出訊息的第一行，變數 vb 的內容已被帶入。接下來我們再來看設定 verbose 變數的情形：

```
1 % set vb = 'set verbose variable'
2 % set verbose
3 % echo $vb
echo $vb
set verbose variable
```

```
4 %
```

我們清楚地看到 $vb 的內容並沒有帶入變數中，依然保持了 echo 指令的字元模式“$vb”。這便是兩者之間的差異所在。其時設定 verbose 變數就相當於是 C shell 的“-x”選項。

這兩個變數的最常用在撰寫 C shell 文稿產生錯誤時，相互配合使用來 debug。一般在指令行模式下比較少有必要設定它。

status 執行狀態變數

變數 status 乃是用以顯示最近的指令執行狀態。執行成功狀態爲 0；失敗狀態爲 1。本變數常運用於 shell 程式設計中判斷上一個指令的執行情況。譬如用指令 grep 在資料檔中找尋某個字串的狀態。

```
% grep akira /etc/passwd
akira:lv8u/EYmXK5kU:101:100:SYMBAD USER:/home1/akira:/bin/csh
% echo $status
0
% cd aaaa
aaaa: No such file or directory
/home1/akira> !e
echo $status
1
```

cwd 目前工作目錄變數

cwd 變數所代表的便是目前的工作目錄。如果我們用指令 echo 來顯示該變數的內容，其實它和指令 pwd 的效果相當接近。如下：

```
% echo $cwd
/home1/akira
% pwd
/home1/akira
```

一般最常應用此變數來設定題詞（prompt）的顯示內容，或者是運用在 C shell 文稿內顯示目前所在的工作目錄。讓我們來設定一個可以顯示目前工做目錄的題詞，如下：

```
% set prompt = "$cwd % "
/home1/akira %
```

如上所示，在題詞中我們就可以得到工作目錄的訊息、而不再需要以指令 pwd 來得到此訊息。不過這種設定方法並不會再你更改工作目錄時，自動地更改題詞的資料。所以最好的設定方式還是以下的方式：

```
/home1/akira % alias cd 'cd \!* ; set prompt = "$cwd % " '
```

```
% cd
/home1/akira % cd /
/ % cd
/home1/akira %
```

我們使用 aliases 的功能重新設定 cd 的功能，讓 cd 不但只執行原來的cd指令而且還重新設定一次 prompt 變數的內容。如此便能在你每次更改目錄時將目前的工作目錄反應在題詞中了。也許你會問，爲何不用指令 pwd 而要用變數 cwd 呢？其時這是執行效率的問題。因爲指令 pwd 並不是內建指令，必須做 I/O 動作，而使用 C shell 變數則不需要這個動作。所以使用變數 cwd 是比較好的選擇。

hardpaths 實體路徑變數

此變數 hardpaths 和變數 cwd 極爲相似，兩者都與顯示工作目錄的變數，然而 hardpaths 變數可以說是針對修正變數 cwd 在顯示上的一項錯誤，所產生的一個比較特殊的變數。這項錯誤產生在 UNIX 作業系統的一項重要的功能上，就是檔案系統的目錄連結（symbolic links或稱爲soft links）。這是因爲這項技術發展於 C shell 定義變數 cwd 之後的緣故。首先讓我們延用上例的提詞設定來看下面的例子：

```
/home1/akira % ln -s /usr/share/man man
/home1/akira % ls -l man
lrwxrwxrwx 1 akira 8 Oct 31 01:57 man -> /usr/share/man
/home1/akira % cd man
/home1/akira/man % pwd
/usr/share/man
```

在上例中我們將“/usr/share/man”以指令 ln 連結到目錄“man”，然後我們以指令“ls -l”顯示該目錄，訊息很明顯地告訴我們該目錄的連結狀態。當我們以指令 cd 到該目錄之下時發現題詞所顯示的工作目錄是“/home1/akira/man”，但指令 pwd 的結果卻告訴我們目前的工作目錄是“/usr/share/man”。這便是變數 cwd 無法應付這種 symbolic links 所造成的錯誤。如果你的 UNIX 作業系統版本有這樣的問題，你便可設定這個 hardpaths 變數除這個“bug”。使用語法如下：

```
% set hardpaths
%
```

當然它最好也是設定在“.cshrc”檔案中。

因爲啓動這項功能的作用與會使用到 directory stack 的內建指令 pushd 及 popd 有相當大的關係。這就是爲何要修改這項 bug 的最主要原因。

ignoreeof 忽略使用 eof 退出變數

使用語法 set ignoreeof

在一般的情況下，如果我們在 C shell 中使用 CTRL-d，所造成的結果便是將該 C shell 終結，如果在 login shell 中使用則會退出系統。這樣難免會有誤動作的情況產生。想要避免這種不幸的錯誤，你可以設定 ignoreeof 變數，如此便不會因爲誤用 CTRL-d 而退出系統了。如下所示；

```
% set ignoreeof
% ^D
Use "logout" to logout.
%
```

如果你曾有使用 CTRL-d 造成誤動作而退出系統的記錄的話，建議你好設定此變數。我想應該可以減少一些使用習慣所造成的麻煩。

noclobber 禁止覆寫變數

這個 noclobber 變數我們在前面的輸出重導向章節中已經提過，它的功能便是停止重導向符號“>”的覆寫（overwiting）已存在檔案以及符號“>>”要將資料寫入一個不存在的檔案時，自動產生該檔案的特性。我想在此便不再贅述，僅用兩個例子讓讀者回憶一下，設定後的實際使用狀況。

例子一：

```
% ps axu > testfile
% set noclobber
% echo "test set noclobber" > testfile
testfile: File exists.
% echo "test set noclobber" >! testfile
%
```

例子二：

```
% set noclobber
% cat /etc/passwd >> nopass
nopass: No such file or directory
% cat /etc/passwd >>! nopass
%
```

noglob 變數

設定這變數 noglob 的作用是停止 wildcard 功能，也就是說像符號 * ? [ ] { } ~ 等等，它們所代表的特殊作用都將失去效用。而僅僅只是代表一般的字元而已。如下面的例子所示：

```
% echo ~
/home1/akira
% echo *
akbin bourne cshell project soft
% set noglob
% echo ~
~
% echo *
```

*

看到沒，在設定完變數 noglob 後，代表 home 目錄的“~”與符號“*”等均失去其原有的特殊效用。所以要使用這個變數請務必瞭解自己在做什麼！否則你會以爲電腦壞了？

建議您如果需要將整個 wildcard 功能暫時停用時再手動設定這個變數是最好的使用方式。如果只是二、三行指令的話我建議使用倒斜線“\”來暫時消除特殊符號的功能。這個方法同樣可行。如果選擇設定 noglob 變數的話，別忘了不用時您只要 unset noglob 便可以回覆到設定前的使用模式了。

有時候在我們撰寫 C shell 文稿會因爲要常常需要將特殊符號當成一般符號使用，您可以設定這項變數將終止 wildcard 的功能，關於這點我們在下一章中再爲你舉例說明之。

nonomatch 變數

在我們使用 C shell 的一般的情況下，大都不會放棄使用 wildcard 功能，因爲它實在帶給我們相當多的便利。不過使用這項功能對某些指令的執行會造成一些負面的影響，讓我們來看幾個指令執行的錯誤例子：

```
% ls
aaa abc akira core
% rm *.tmp core
No match.
% ls
aaa abc akira core
```

我們看到指令“rm *.tmp core”在執行第一個引數“*.tmp”產生錯誤，因爲工作目錄下並沒有這類檔名的檔案存在。不過指令 rm 卻會因爲這項錯誤而終止執行下一個引數“core”。所以當我們在一次以指令 ls 來查看執行狀況時，你會驚訝地發現檔案“core”並沒有被清除。這或許在指令行模式下會被我們發現，但如果是在 C shell 文稿中，這可就是個嚴重的 bug 了。

在使用 C shell 時要消除這類因使用 wildcard 所產生的問題，有兩種方式。一是設定變數 noglob 乾脆放棄使用 wildcard 功能。另一個便是設定 nonomatch 變數，它可以消除這類問題，同時你也能繼續保有使用 wildcard 的功能。

```
% set nonomatch
% rm *.tmp core ; ls
rm: *.tmp: No such file or directory
aa abc test
```

我們可以看到在設定 nonomatch 變數後，檔案“core”被清除了。

當然在設定 nonomatch 變數後，因爲指令執行上也的改變，也有一種情況會與未設定前不同，那

就是指令的執行狀態也改變了。見下面說明：

```
% ls *.tmp ; echo $status
No match.
1
% set nonomatch
% ls *.tmp ; echo $status
*.tmp not found
0
```

在設定 nonomatch 變數之前，指令 ls 如果無法找到合符條件的檔案，它的指令執行狀態是失敗的，所以爲“1”。但是設定了 nonomatch 變數之後，指令 ls 在相同的情況下，指令的執行狀態則是成功的，所以爲“0”。這可是相當大的變化，請讀者在使用上也最好注意到這種設定後的變化情況。萬一你要使用指令的執行狀態來作爲判斷依據時，可得小心！免得因爲這個變化，使程式多出一個 bug。

notify 變數

變數 notify 與工作控制（Job control）的背景工作顯示有著密切的關係。由於這個變數的運用不太適合片段式的介紹，所以我把它和工作控制的介紹一起放在第三章中了。簡單的說：它的功能便是會將背景工作的處理完成訊息插播到前景工作中。詳細請參考第三章的工作控制。

filec 檔名自動續接變數

設定 filec 變數對於冗長的檔名或者是非常特別“懶”的人幫助是相當大的。它的使用方式是你在指令行模下先鍵入該檔名的前幾個字母，然後按ESC鍵，C shell 便會幫助你將符合的檔名自動地補上。讓我們來看下面這個例子：

```
2 % set filec
3 % ls
echoerr.c echoout.c screenprint.c
4 % cc scrESC
```

當指令4檔案名稱鍵入到一半時，你按下ESC鍵，檔名便自動接上。如下：

```
4 % cc screenprint.c
```

方便吧！當然啦它有會有不靈光的時候，比方說鍵入的字首部份，經過 C shell 偵測發現，符合條件的檔案超過一個的時候， C shell 便會發出“嗶”聲警告你。在這種情況下，所鍵入的指令行便不做任何改變，保持原來的情況等待你繼續再鍵入資料。如下例的情況：

```
5 % cc echoESC （發出“嗶”聲警告）
```

另外它還會有一個功能，就是你可以鍵入檔名的前面的幾個字母，然後使用“CTRL-d”來顯示出

符合該條件的檔案。請見下面的例子：

```
6 % cc echoCTRL-d
echoerr.c echoout.c
6 % cc echo
```

在顯示出相關的檔案名稱之後，回復到原來的指令行狀態等待你輸入。感覺怎樣！用用看，你一定會喜歡的。如果你對於這樣的功能覺得很適用的話，你可以在“.cshrc”檔案中設定它，或者是要使用時才在指令行中設定也可以。

fignore 變數

這個變數 fignore 是配合變數 filec 使用的。對於在上面的 filec 變數所支援的功能，在 UNIX 中稱爲“filename completion”，該功能的作法就是利用鍵入的字首部份，由 C shell 自動判斷並補上完整的檔案名稱。 filec 變數便是設定啓用“filename completion”。而變數 fignore 則是設定該功能忽略掉某種檔案的尾名。我們來看下面的例子：

```
2 % set filec ; set fignore = (.o .out)
3 % ls
screenprint.c screenprint.o screenprint.out
4 % cc screenESC
4 % cc screenprint.c （按完 ESC 之後的情況）
```

如果我們沒有設定變數 fignore 的話，指令 4 會因爲有三個檔案符合條件，而產生嗶聲警告我們。但因爲已經設定了忽略“.o”及“.out”檔名，所以只剩下一個檔案符合條件，於是便自動將符合條件的檔案名字補上了。這便是設定後所產生的功能變化。不過設定此變數並不會對 CTRL-d 的顯示造成任何的影響，這點請讀者注意。

nobeep 不准叫變數

設定這個變數的作用是取消“嗶”的警告聲。使用方法如下：

```
% set nobeep
```

討厭聲音的人可以設定此變數叫電腦“閉嘴”。以往在深夜無人的辦公室內打電腦的我，習慣手動設定這個變數，以免嚇到警衛伯伯。

參考資料、書籍：

Sun OS 4.1.3 on-line manual

Unix C Shell Field Guide by Gail Anderson, Paul Anderson

UNIX Power Tools by Jerry D. Peek, Tim O'Reilly, Mike Loukides