第十七章

個人化的環境設定

與作業系統溝通是相當耗費時間的，因為使用者必須時常改變目錄、列出檔案以及編譯程式，並且一直重覆這些動作，所以如果有合適的捷徑的話，一定會方便許多。不正確或設計不良的環境設定，對UNIX程式設計師及系統管理者而言是個夢魘，而身為使用者，當然會希望給自己建立一個舒適的環境。也許很多人還不了解UNIX強大的力量，它在環境控制上確實是獨一無二的。

本章著重於shell的環境相關功能，包括四種shell。只要操作設定這些shell，UNIX是可以高度個人化的，指令可以改變其預設的行為，使用者可以重新呼叫、編輯或重新執行先前的指令，甚至可以設計指令的捷徑，或是為自己製訂shell指令或檔名；也可以在shell永久保留你要的設定，存在script中，讓你每次在登入時就設定好。這些個人化的設定可到何種程度，取決於所使用的shell，在讀完本章後，使用者可以選擇一個適合自己的shell來使用。

目　標

- 了解Bourne、Korn、bash及C shells的環境相關功能。(17.1)
- 了解環境變數(environment variable)的意義與重要性。（17.2及17.3）
- 使用別名(aliases)來縮短指令長度。(17.4)
- 使用歷史(history) 功能來叫回、編輯及重新執行先前的指令。(17.5)
- 使用Korn shell或bash中，與vi或emacs功能相似的線上編輯功能。(17.6)
- 使用Korn shell或bash中檔名及指令名稱自動展開的功能。(17.7)
- 使用**noclobber**保護檔案，防止不小心被覆寫，及使用**ignoreeof**避免意外地登出。（17.8.1及17.8.2）
- 使用~（tilde，波浪符號）來代表家目錄。(17.8.3)
- 使用.或source指令來執行shell script，而不產生sub-shell。(17.9.1)
- 設定起始檔案(startup files)，在登入或開啟sub-shell時執行相關指令。（17.9.3到17.9.6）

17.1 該使用那個shell?

在先前的章節中，常常會碰到指令在某個shell可執行，但在另一個shell卻不能執行的狀況，所以有時使用這些指令（像echo及sed）時，必須加上\才可以執行。這表示即使在同一系統下，由於指令執行方式的不同，所以會表現不一樣的功能，可能有如下的原因：

- 使用者正使用一個shell的內部指令而不自知。每個shell都有自己的一組內部指令，所以指令在其他shell中表現出的行為可能不同。
- 不同的shell其共同環境參數的設定可能不同。
- shell並不允許執行該指令，所以先前已經遇到過在按*[Enter]*前必須加上\的情況。

shell決定了使用者最原始的環境，也決定了使用者希望能達成之舒適度。現在的shell，像是Korn shell及bash，擁有豐富的指令集，及高度的可個人化，如果選對了登入的shell，就會比別人有更好的開始。但無論如何，人們總有自己喜歡使用的shell，而C shell是最受歡迎的，因為它是最早提供某些好用指令集的shell，而這些功能在後來興起的shell中也都有加入，這就是為什麼到現在C shell仍然存活的原因。

使用者可以請系統管理者來設定自己的登入shell，或系統會提供chsh指令讓使用者自己設定。下列方式可以讓使用者自己設定Korn為登入的shell：

```
$ chsh
Password: ********
Changing the login shell for henry
Enter the new value, or press return for the default
Login Shell [/bin/csh]: /bin/ksh                    使用這個shell!
$ _
```

這個指令更改了**/etc/passwd**最後一欄為**/bin/ksh**，而系統管理者也可手動、或使用usermod指令來更改：

```
henry:x:501:100:henry blofeld:/home/henry:/bin/ksh
```

在更改自己的個人環境之前，最好以下列的指令來確定現在所使用的shell：

```
$ echo $SHELL
/bin/ksh
```

在本章中，會介紹下列shell的相關環境功能：

- Bourne shell(**sh**)：UNIX系統第一個shell。
- C shell(**csh**) ：為Berkeley發展之最好的解譯器(interpreter)，但較不適合做為程式語言(programming language)。
- Korn shell(**ksh**)之*ksh93*版本：相較於上兩者是最好的shell，如果使用者的系統

沒有這個版本的話，可從*http://www.kornshell.com*來下載免費的非商業版本，只要將它放置於**/bin**下即可（如果有其他版本的話，可將它重新命名）。但在本章所介紹的大部分Korn shell的功能，在較早的版本，如*ksh88*，也是可以使用的，且它可能已安裝於使用者電腦中。在17.7.1節的「小技巧」有介紹如何知道Korn shell版本的方法。

- Bourne Again shell(bash)：Linux系統所使用之標準版本，功能相當於Korn shell，在UNIX上也有一些使用者特別喜歡使用它。

表17.1比較了本章會介紹之shell的各個功能，每個shell在各自的章節會介紹到，但每一章節可能會介紹兩個或多個shell的共同功能。

表 17.1　**各**shell**之功能比較表**

功　能	**sh**	**csh**	**ksh**	**bash**
路徑名稱	/bin/sh	/bin/csh	/bin/ksh	/bin/bash
定義區域變數*var*	*var=value*	set *var=value*	var=*value*	var=*value*
定義環境變數*var*	setenv *var*	setenv *var value*	export *var=value*	export *var=value*
顯示環境變數	export	setenv	export	export
別名	—	有提供	有提供	有提供
定義別名	—	alias *name value*	alias *name=value*	alias *name=value*
指令歷史及取代	—	有提供	有提供	有提供
取出先前指令的參數	—	有提供	—	有提供
截取路徑名稱	—	有提供	—	有提供
線上指令編輯	—	—	有提供	有提供
檔名展開	—	有提供	有提供	有提供
使用cd -進行目錄	—	—	有提供	有提供
用cd ~user回家目錄	—	有提供	有提供	有提供
執行script時不要建立sub-shell	.	source	.	.或source
登入檔案	.profile	.login	.profile	.bash_profile, .profile 或 .bash_login
環境設定檔	—	.c	由ENV變數決定（通常是 . kshrc）	.bashrc或通常為BASH_ENV決定
登出檔案	—	.logout	—	.bash_logout


Note

雖然每個shell分別都有自己的章節來介紹，但有很多概念性的細節在每個主題的介紹資料中都可被找到，包括以Bourne shell為主的章節，這樣可以避免介紹太多重複的資訊。如果使用者只想閱讀介紹某個shell的章節，還是必須參閱其他章節，以獲得較完整的資訊。

17.2 環境變數

UNIX系統是由一些特殊的shell變數來控制(8.11.1)，有些在電腦啟動時就已設定，有些則等到登入時才設定。當使用者開啟一新的sub-shell時，有些sub-shell的變數會繼承自parent。在本節中，將會討論這些**環境變數(environment variables)**或系統變數的意義，並學習如何更改，以方便自己使用。

環境變數在範圍上和簡單的（區域的，local）shell變數是不同的，它們可在全區域中(globally)被引用（被輸出 export）。也就是說，它們在使用者整個環境下都是可看見的(visible)，不論是執行script時所產生的sub-shell、mail指令或是在編輯器裡。最主要及特別的shell環境變數分類於表17.2中。

表 17.2 事先定義好的系統變數

sh	csh	ksh	bash	功 能
HOME	home	HOME	HOME	家目錄，即使用者登入時所進入的目錄
PATH	path	PATH	PATH	shell搜尋指令時所尋找的目錄
LOGNAME	user	LOGNAME	USER或LOGNAME	使用者的登入名稱
MAIL	mail	MAIL	MAIL	使用者郵件信箱的絕對路徑
MAILCHECK	mail	MAILCHECK	MAILCHECK	檢查新進郵件之時間間隔
—	history	—	HISTSIZE	儲存於記憶體中的指令數目
—	savehist	HISTSIZE	HISTFILESIZE	儲存於歷史檔案中的指令數目
—	永遠為history	HISTFILE	HISTFILE	歷史檔案
TERM	term	TERM	TERM	螢幕型態
—	cwd	PWD	PWD	目前目錄的絕對路徑
CDPATH	cdpath	CDPATH	CDPATH	當使用**cd**指令加上一非絕對路徑時，所要搜尋的目錄列表
PS1	prompt	PS1	PS1	主要的提示符號字串
PS2	永遠為?	PS2	PS2	次要的提示符號字串
SHELL	shell	SHELL	SHELL	使用者的登入shell，及使用跳脫指令時，會執行的shell
—	永遠為.cshrc	ENV	BASH_ENV	執行sub-shell時，所使用的環境設定檔

17.2.1　在Bourne、Korn及bash shell中使用變數

使用set敘述句可以顯示所有的變數，不管是簡單的或是環境變數(1.7.5)，而env指令（或export敘述）則只顯示環境變數。在Bourne shell中使用env指令會顯示如下的資訊：

```
$ env
CDPATH=.:..:$HOME
HOME=/home/romeo
LOGNAME=romeo
MAIL=/var/mail/romeo
MAILCHECK=60
PAGER=/usr/bin/more
PATH=/bin:/usr/bin:/usr/dt/bin:/home/romeo/bin:.
PWD=/home/romeo/project5
PS1=’$ ‘
SHELL=/bin/sh
TERM=ansi
....
```

習慣上，環境變數會使用大寫字母，所以使用者在定義自己變數時，可以使用小寫字母，上述指令的輸出也顯示了變數定義的方法，或是重新指定其值的方式。例如，以下為改變螢幕型態的方式：

```
$ TERM=vt220
$ _
```

一個變數可以在命令列以此種方式定義，並且在script仍可使用嗎？是的，像**TERM**這種變數就可以，但是像下面這種指定方式呢？

```
x=5                                    x 為區域變數(local variable)
```

上述x並無法使用於全區域(globally)，只能使用於目前的shell。但如果使用export指令的話，就可以將它轉換為環境變數：

```
export x                               x變為環境變數
```

Korn shell及bash可將這兩個動作以單一的敘述句來表示(export x=5)。而由env指令顯示出的變數（及大部分用set指令可顯示的變數），為何不必使用export呢？原因就是這件工作早就已經做了，亦即當使用者登入時，系統變數已被登入的shell執行之起始script輸出了。在19.6.1節，將使用export指令來認識已經及未經輸出之指令有何不同。

Tip

為避免與系統變數混淆，使用者應該以小寫字母來定義自己的變數。

17.2.2　在C Shell中使用變數

C shell在設定全區域及非全區域變數值時，使用的方式是不同的。非全區域變

數其值之設定是使用set敘述：

```
% set x = 20                                    等號(=)前後不需要空白字元
% echo $x
20
```

C shell在等號(=)前後不需要空白字元，但為了可讀性，所以加上了空白。這樣的設定方式是非全區域的。而要讓使用者之所有script均可使用，則需使用setenv敘述句來指定：

```
setenv x 30                                    沒有等號(=)
```

setenv並不需使用等號(=)（初學者通常會搞混），變數名稱及其值只需一個接一個放上去。此處我們使用set及setenv設定了相同的變數名稱，那麼，**x**的值到底為何？

```
% echo $x
20                                             為區域變數值(local value)
```

echo顯示了區域變數值，但在sub-shell中，使用相同的敘述所得到的卻是全域變數值：

```
% csh                                          建立了一個C的sub-shell
% echo $x
30                                             全域變數值
```

在兩者均有定義的shell中，區域變數覆蓋了全域變數，但是，如果只定義全區域變數（使用setenv）的話，在目前的shell仍可使用這個值。所以為了避免混淆，區域及環境變數最好使用不一樣的名稱。

set敘述句顯示了目前shell所有的變數，但並非是全區域變數。這裡沒有顯示等號(=)，變數名稱及其值是以空白鍵來分隔：

```
% set
cwd         /home/julie/docs
history     100
home        /home/julie
mail        /var/mail/julie
path        (/bin /usr/bin /usr/ucb .)
prompt      %
savehist    100
shell       /bin/csh
term        AT386
user        julie
x           20                                 顯示區域變數值
```

注意此處使用set顯示了**x**的值為20，我們也看到了一些像是環境變數的變數，但實際上卻不是，只有使用setenv所顯示的才真正是環境變數，使用的方式為直接輸入

setenv而不加參數：

```
% setenv
HOME=/home/julie
PATH=/bin:/usr/bin:/usr/ucb:.
LOGNAME=julie
TERM=AT386
SHELL=/bin/csh
MAIL=/var/mail/julie
PWD=/home/julie/docs
USER=julie
OPENWINHOME=/usr/openwin
x=30
```

不一樣的格式

顯示環境變數值

在其他shell使用set指令所顯示的是相同的輸出格式，setenv顯示了**x**的值為全區域值(30)（set則顯示區域值20），注意這裡一些原本為小寫的系統變數，像**home**、**term**及**path**，此時卻用大寫來表示，這是一切混淆的開始，所以先來討論這個議題。

當一些像是vi的程式讀取了大寫的環境變數時，C shell本身則使用了小寫的區域變數(local variable)，例如，它使用**path**而不用**PATH**來找出指令位於何處。很多這種區域變數都有相對應的環境變數，例如echo $TERM及echo $term輸出的是相同的值。在同一系統中，改變**term**及**user**會自動更新**TERM**及**USER**的值，但反之通常是行不通的。當環境變數是取自相對應的shell變數時，這兩者的關係就有點複雜了，本書並不打算討論這個主題。

這　有一變數其“set”的格式與“setenv”格式是不一樣的；那就是**path**，它顯示了用括號(parentheses)包住，且以空白分隔的串列。當學習如何指定C shell的array值後（附錄A），就可以了解其原因。

Note

外部程式會讀取環境變數，而不是簡單的shell變數。一些像是TERM、PATH及USER等變數，都是從相對應的shell變數複製而來，相當奇特的是，C shell本身使用小寫的變數，其中有些不能使用於全區域。

17.3　環境（系統）變數的意義

使用set、env、export或setenv敘述所顯示的變數，控制了系統的行為。在很多狀況下，使用者都會遇到這些指令，而在使用UNIX的過程與經驗中，也會修改其中很多的設定。首先，我們將討論Bourne shell，它的變數亦可使用於Korn及bash，其中一些在C shell亦可使用。但不管使用那一個shell，都必須閱讀以下的章節，來了解這些變數的意義。

17.3.1 Bourne、Korn及bash shell的環境變數

指令搜尋路徑 (PATH) **PATH** 是一個很重要的系統變數，它決定了在執行指令時，需不需要使用絕對路徑。使用下列方式可以知道它目前的值：

```
$ echo $PATH
/bin:/usr/bin:/usr/dt/bin:/home/romeo/bin:.
```

這個變數告訴shell在尋找任何可執行的指令時，該依照哪個路線(route)來尋找，這裡列出五個用冒號(:)分隔的目錄。注意最後有個點(.)，它代表現在的目錄(6.8)，將它列在最後則代表了當在所有目錄均找不到想要尋找的指令時，也要尋找現在的目錄，這在先前已討論過了(2.2)。

如果使用者想要將目錄**/usr/xpg4/bin**放置於搜尋串列中，可使用下列方式來重新定義變數：

```
PATH=$PATH:/usr/xpg4/bin                    將原來的值加入新值中—OK
```

注意這個最新加入的目錄最後才會被搜尋到，而且是在現在的目錄之後。這個新目錄放置的是Solaris之POSIX相容的工具程式，其中的grep指令與**/bin**或**/usr/bin**下的行為是不太一樣的。


Note

如果在不同目錄下的兩指令有兩個相同的名稱，則較先出現在PATH串列中的會被執行，這也就是為什麼使用者在現在目錄下有一cat指令，但在執行cat foo時，卻會執行/bin/cat的原因，因為在PATH設定中，現在目錄是被排在/bin這個目錄之後。

家目錄(HOME) 當登入時，UNIX一般會將使用者放置於一個以登入名稱為名的目錄，這個目錄稱為家(home)或登入(login)目錄(6.5.1)，這個值存放於**HOME**變數中：

```
$ echo $HOME
/home/romeo
```

使用者的家目錄是設定於**/etc/passwd**，其中有一行是屬於此使用者。此檔案中，每一行記錄著一使用者的相關資訊，共有七個欄位，每一行看起來就像這樣：

```
romeo:x:208:50::/home/romeo:/bin/sh
```

家目錄記錄於最後第二個欄位，當一使用者登入時，login程式會讀取這個檔案(10.4)，並據此設定**HOME**及**SHELL**這兩個變數。**/etc/passwd**只能由系統管理者來編輯，可手動或使用useradd或usermod指令，如果使用者想要更改**HOME**的話，必須請系統管理者來更改。

使用者可更改**HOME**的值，但更改的不是家目錄，而是使用cd指令且不加參數時，會更動到那個目錄去，一個單獨的cd指令意思是cd $HOME。


Tip

如果想要使用某人的.xinitrc(12.13)**來使用他所選擇及放置之X客戶端程式，可以使用以下方式來啟動X：**

```
HOME=/home/julie xinit                                    在同一行
```

使用者並不需複製julie的.xinitrc到自己的目錄，X會自動讀取這個檔案，而非使用者自己的。當依上述方式在命令列設定變數值時（不使用分號來分隔），變數值只是暫時被更改，當程式執行完畢時，就會回復原值。

使用者名稱(LOGNAME)　這個變數代表使用者名稱，當使用者瀏覽於檔案系統中時，有時會忘記自己的登入名稱(login name)（這聽起來很詭異，但的確有可能會發生，尤其同時擁有多個帳號時！），但不管何時，最好能確定自己知道登入的帳號是哪一個：

```
$ echo $LOGNAME                                    或試試 whoami
romeo
```

在何處還會用到這個變數呢？在script中，如果同一script會根據使用者之不同而做出不同反應的話，就會用到這個變數。

man指令所使用的分頁器**(**PAGER**)**　在大多數系統中，man指令會使用**PAGER**變數值來決定作為分頁輸出的程式。在現代的UNIX系統中，其值通常會是**more**，而舊系統則使用**pg**。如果使用者的系統顯示**pg**，但也提供more的話，可依下列方式來更改**PAGER**的值：

```
$ PAGER=/usr/bin/more ; echo $PAGER
/usr/bin/more
```

如果系統也提供less的話（在Linux系統中，它是man的預設值），那麼使用者最好使用less。less在各方面的功能都比more強大（更別提pg了），而且更好用。


Note

有些UNIX**系統並不使用這個變數，而是將它定義於/etc/default/man檔案中。如果使用者的系統中有這個檔案，就會看到像是這樣的一行：PAGER=/usr/bin/more。另外，如果在**Solaris**中沒有定義PAGER變數的話，則會使用** more -s。

信箱**(mailbox)**位置及檢查頻率（MAIL及MAILCHECK）　UNIX的信件處理系統並不主動通知使用者是否有信件寄到，這項工作是由shell來負責。shell由MAIL變數來得知使用者的信箱位置，通常是位於**/var/mail**、**/var/spool/mail**或**/usr/spool/mail**（在較舊的系統中）。romeo的信件在SVR4系統中是儲存於**/var/mail/romeo**。

MAILCHECK變數決定了shell該多久檢查一次信箱，以得知是否有新信件到達，通常在大型系統中這項設定為600秒。如果shell發現在上次之檢查後，此檔案已被修改，則表示有新信件到達，並會使用像是如下的訊息來通知使用者：

```
You have mail in /var/mail/romeo
```

如果romeo正在執行一個指令，則會在完成這個指令後才會顯示這個訊息。

提示字串符號（PS1及PS2） shell在**PS1**及**PS2**中各存有一個提示符號，正常看見的是**PS1**，但需要在下一行繼續一指令時，shell會回應>符號：

```
$ sed ‘ s/STRONG/BOLD/g
> s/html/HTML/g ’
```

這個>符號就是**PS2**所代表的次要的提示字串符號。在Bourne shell中，PS1及**PS2**通常分別為**$**及>符號。

如果使用者較習慣Windows環境，也可以使用下列方式來更改主要提示字串符號為C>：

```
$ PS1= ”C> “
C> _
```

由於主要提示字串符號最常使用**$**，為避免混淆，系統管理者通常會使用**#**作為提示符號。在介紹Korn及bash shell時（17.3.3及17.3.4），會更進一步討論**PS1**的使用方式。

目錄搜尋路徑(CDPATH) 使用者也許會時常使用其他位置的目錄，例如使用者在家目錄下有兩個目錄**bar1**及**bar2**，而目前正位於**bar1**，如果需要移至**bar2**的話，通常會使用cd ../bar2，但如果事先設定好**CDPATH**的話，就可以省下幾個按鍵動作：

```
CDPATH=.:..:/home/romeo/project5
```

這記錄的又是一個包含三個目錄名稱的字串，shell就是依此順序來搜尋，但不同的是，此時搜尋的是目錄名稱。所以當在**bar1**中使用cd bar2指令時，shell首先會搜尋目前的目錄(.)，來看看是否有**bar2**，找不到的話再搜尋父目錄(..)，因為**bar1**及**bar2**是位於同一階層，所以此時就可找到：

```
$ pwd
/home/romeo/bar1
$ cd bar2 ; pwd
/home/romeo/bar2
```

而當shell在**/home/romeo**還是找不到**bar2**的話，就會再尋找**/home/romeo/project5**。

使用**shell**跳脫**(Escape)**指令而跳至的**shell(**SHELL**)**　**SHELL**變數告訴使用者目前所使用的shell，先前已看過像vi、emacs及telnet等指令會使用跳脫路徑(escape route)跳到某個shell，**SHELL**變數則定決定了將跳至哪個shell。即使vi使用:sh指令來跳脫至某個shell，但實際上會呼叫的shell還是取決於**SHELL**變數，並不一定必須是Bourne shell(sh)。

像**HOME**變數一樣，系統管理者新增一使用者時，會在**/etc/passwd**檔案設定使用者登入的shell。回顧先前討論**HOME**變數時看到的範例，最後一欄設定了**SHELL**的值，而chsh指令會更改這個欄位的值。

螢幕型態(TERM)　**TERM**變數指定了所使用的螢幕型態，每一螢幕在**/usr/lib/terminfo**目錄（Solaris則為**/usr/share/lib/terminfo**）都有一相對的控制檔，來定義其特性，這個目錄包含一些子目錄，每個子目錄其名稱都是一個英文字母，而螢幕控制檔案儲存於何目錄，則取決於螢幕名稱之第一個字母，例如，ansi螢幕使用**/usr/lib/terminfo/a/ansi**的控制檔。

有些工具程式，像是vi，與螢幕型態是息息相關的(terminal-dependent)，它們必須知道使用者所使用的螢幕型態，如果**TERM**的設定不正確的話，vi就無法正確顯示。**TERM**變數在使用者登入遠端電腦時也很重要，如果設定不正確，很多UNIX工具程式就無法正常運作，或是會在螢幕上產生亂碼。

17.3.2　C shell的特殊變數

雖然C shell使用大寫名稱來命名環境變數，使用者在C shell中還是使用了一些小寫變數值，事實上，雖然有些變數沒有相對的大寫變數，但還是需要交替使用的，可以利用set敘述句來指定這些變數。

主要的提示符號(prompt)　C shell將提示符號字串儲存於**prompt**變數中，通常會是一個百分比符號(%)，可使用set來自行設定：

```
% set prompt = "[C>] "
[C>] _
```

次要的提示符號字串為問號(?)，但它並不儲存於任何環境變數中。另外，**prompt**變數並沒有相對的大寫變數（雖然如此，在sub shell中仍是可使用的）。

在提示符號中加入事件號碼**(event number)**(!)　C shell是首先使用指令歷史(history)概念的，它可讓使用者重新執行先前的指令，而不必重新輸入，我們將在稍候再介紹。現在，假設我們已經知道，在歷史指令串列中，每一指令都有其相對的事件號碼，那麼就可使用此事件號碼來存取或執行一指令。

很多人利用這個特性，對所有指令均指定了一事件號碼，所需的字元為驚嘆號(!)，並且加上跳脫字元：

```
% set prompt = ' [\!] '
[12] _
```

set prompt*在歷史指令串列中，是第11個指令*

注意此處!前有個\，因為在shell中，它有特別的意義（現在加上跳脫字元，但稍後不須使用!也可跳脫）。下次使用者輸入指令時，就會將此指令加入歷史串列(history list)中，而事件號碼也會跟著增加。稍後我們將會探討歷史指令的用法。

Note

定義提示符號時，我們使用\來跳脫!，因為它後面接著一個非空白字元(])。C shell將任何接在!後面的字串解譯為指令，並會試圖去重覆執行先前以這個字串為開頭的歷史指令。

指令搜尋路徑(path)　這是C shell用來表示Bourne之**PATH**變數的方法，而其顯示出來的路徑串列(path list)也不一樣：

```
% echo $path
/bin /usr/bin /usr/lib/java/bin /usr/dt/bin
```

和Bourne不同的是，這裡所顯示的目錄串列是以空白來分隔的，而更特別的地方在於，使用set指令所顯示的輸出，一樣是用括號括起來：

```
path          (/bin /usr/bin /usr/lib/java/bin /usr/dt/bin)
```

這實際上是一個有四個元素的陣列，在此，我們不打算討論處理陣列的細節，附錄A中會有相關的討論。我們必須知道的是重新指定這些變數值的方式，例如，要在路徑串列中加入**/usr/xpg4/bin**，可以這樣做：

```
% set path = ($path /usr/xpg4/bin)
% echo $path
/bin /usr/bin /usr/lib/java/bin /usr/dt/bin /usr/xpg4/bin
```

setenv指令顯示**PATH**之值為同樣的目錄串列，但是是以Bourne shell的格式(17.2.2)，改變**path**同樣也會改變**PATH**。

郵件信箱及檢查時間間隔(mail)　C shell結合了**MAIL**及**MAILCHECK**兩個Bourne變數於**mail**變數中，即使在17.2.2節中使用set選項顯示**mail**之值只包含一個檔名(**/var/mail/julie**)，但這個變數是可以設定多個檔名的，其前面可選擇放置一數字：

```
set mail = (600 /var/mail/julie /opt/Mail/julie)
```

每600秒，shell就會檢查這兩個檔最後修改的時間，以確定是否有新進之信件，環境變數**MAIL**只儲存單一檔名，用來讓一些外部程式讀取。

其它的變數　以下簡單介紹一些其它的變數：
cwd——儲存目前的目錄。
user——使用者登入的名稱（Bourne為LOGNAME）。
home、**cdpath**、**shell**及**term**——和Bourne中之相對應的大寫部分具有相同的意義，C shell也使用SHELL及TERM為環境變數。
history及**savehist**——用於歷史功能(history facility)，在17.5.1節會介紹。
除了這些變數之外，還有一群特殊的變數，它們沒有值，只有被設定(set)及未被設定(unset)兩種狀態，它們設定的格式為set *variable*，以下是其中四個：
notify——通知使用者一背景執行的程式已完成。
filec——啟動檔名展開，在17.7.2節會討論到。
noclobber及**ignoreeof**——在17.8.1節會討論到。
現在，我們已看過三種C shell變數的型態，像**prompt**這種變數是設定為單一值，第二種型態為陣列，它可設定多個值（像**path**），而第三種則完全沒有任何值（像**noclobber**）。

17.3.3　其他Korn shell的環境變數

Korn shell是Bourne的父集(superset)，因此先前介紹之所有的Bourne變數都可應用於Korn，但是Korn也有一些自己的變數。

提示符號下之現在目錄（PWD及PS1）　Korn shell使用**PWD**變數來儲存現在目錄的路徑名稱，它比C shell的**cwd**還好用，**$PWD**可使用於**PS1**的指定句：

```
$ PS1= '[$PWD] '                                          只有單引號
[/home/romeo] _
```

提示符號會從**$**變成[**/home/romeo**]，現在改變目錄為**cgi**：

```
[/home/romeo] cd cgi
[/home/romeo/cgi] _                          提示符號會反應目錄的改變
```

PWD是較特殊的變數，它在每一次工作目錄改變時，都會重新設定一次，而提示符號也因此改變來反應**PWD**的新值。在C shell中試著使用**cwd**來做相同的事，但它無法執行（可使用別名，它可設定來做相同的動作）。

在PS1中使用事件號碼(!) Korn shell也提供歷史功能來叫回及重新執行先前的指令，使用者可設定**PS1**之提示符號來顯示現在的事件號碼，這項功能是從C shell而來，並且使用!來表示這項指定動作的事件號碼：

```
$ PS1="[!] "                                          在Korn中並不需要\
[42] _
```

或是更進一步加上**PWD**變數來顯示現在的目錄。這次，我們必須加上單引號，因為要使用**$PWD**之變數值時，不能使用雙引號：

```
$ PS1='[! $PWD]'                                      在單引號中計算變數值
[43 /home/romeo/project3]
```

每次執行指令時，事件號碼就會自動增加（在這裡是從42變為43），認識事件號碼的意義是相當有用的，因為只要使用這個號碼就可再度執行先前的指令。

其他的變數 Korn shell也使用了一些其他的變數，使用set敘述句時，這些變數或許有些不會顯示出來：

HISTFILE及 **HISTSIZE**使用於歷史指令機制，在17.5節會討論。

ENV描述了在script執行時，所引用的shell，在17.9.5節會討論。

EDITOR及**VISUAL**：決定（vi或emacs）線上編輯(in-line editing)所使用的模式，在17.6節會討論。

17.3.4 其他bash的環境變數

像Korn shell一樣，bash也是Bourne的父集，所以在討論Bourne時所介紹的一些功能也可以使用於bash，而bash也有自己的一些變數。

提示符號下之現在目錄（PWD及PS1） 像Korn一樣，bash也使用**PWD**變數來代替pwd指令，我們可以使用完全相同的方式：

```
$ PS1='$PWD>'                                         必須使用單引號！
/home/juliet>
```

每次改變現在目錄時，**PWD**之值就會被重新設定，也就是說，如果改變目錄的話，提示符號也會跟著改變：

```
/home/juliet> cd cgi                                  提示符號顯示改變後的目錄
/home/juliet/cgi>
```

PWD是較特殊的變數，它在每一次工作目錄改變時，都會重新設定一次。在C shell中也可使用**cwd**來做相同的事，但其值卻不會被重新設定。

在PS1中使用事件號碼(!)　bash shell也提供歷史功能，而且很像C shell。使用者可以叫回先前的指令，每一指令都設定了一事件號碼。!用來表示這個號碼，但必須使用跳脫符號，這樣shell才不會將其後之字串視為指令：

```
$ PS1='<\!>'                                          在bash中需要加上\
<508> _
```

也可使用下面方法來加入PWD變數：

```
$ PS1=’<\! $PWD> ‘
<509 /home/juliet/cgi>
```

這裡使用單引號是沒問題的，而這是在歷史指令集中第508個指令，執行完PS1的指定動作後，此數字就自動增加了。事件號碼在叫回先前已執行過的指令時，是相當有用的。

進一步製定PS1，bash使用一串跳脫序列來讓提示符號更為易於理解，例如，\h字串顯示了電腦的主機名稱：

```
$ PS1=”\h> “
saturn> _                                          saturn 為主機名稱
```

當使用telnet來登入網路上其他的電腦時，使用者通常會搞不清楚現在究竟在哪裡，而這個提示能夠隨時提醒這項訊息。如果在PS1中設定句含您的主機名稱，但在提示符號看見的卻是其他電腦的名稱，那使用者一定是登入了遠端電腦了。

以下的跳脫序列都是可以使用的：

\s—— Shell的名稱。

\t—— 現在的時間，以 HH:MM:SS 格式顯示（\T 為十二小時制）。

\@——現在的時間，以 am/pm 格式。

\w——相對於家目錄之現在目錄。

\u——目前使用者的名稱。

結合上述一些跳脫序列，就可以擁有一個合理、易於理解（雖然稍長）的提示字串：

```
$ PS1="\h \@ \w>"
saturn 09:55am ~/project5/cgi> _
```

除了Bourne shell之外，所有的shell都以~字元（波浪符號）來代表家目錄，目前上面這個使用者正在project5/cgi目錄下（相對於家目錄），~字元稍後會討論到(17.8.3)。

其它的變數　bash還使用了其它一些變數：

HISTFILE、HISTSIZE 及 HISTFILESIZE：使用於歷史指令機制中，第17.5.1節會討

論到。

BASH_ENV：指定當呼叫shell時，所要執行的script，這在第17.9.6節會討論到。

USER：儲存使用者登入的名稱，也可使用LOGNAME。

17.4 別名(aliases)

除了Bourne之外，所有的shell均提供別名**(aliases)**的使用，以供使用者指定常用指令的簡略表達形式，也就是說，如果l是ls -lids的別名，則就可使用l來代替ls -lids。使用者可能想要重新定義一現存的指令，以便在執行時，都可引用一些選項。例如，很多人自訂ls指令，讓它永遠執行ls -xF。設定別名這項動作可使用alias敘述句，但在C shell的定義方式與Korn及bash不太一樣，但下面的功能都是相同的：

- 當使用alias指令而不加參數時，會顯示所有已定義的別名。
- 當使用alias指令而加上一已定義別名之名稱時，會顯示這項別名之定義。
- 一別名可使用unalias敘述句來移除此定義。
- 一別名可遞迴定義，也就是說，如果a為b的別名，而b為c的別名，則執行a就等於執行c。

現在就來討論這三個shell所提供之設定別名的功能，對於在C shell中的使用方式會討論較多，因為Korn及bash也提供shell函式(function)(19.10)，它在各方面均優於別名的使用。C shell並沒有提供函式，所以使用別名是除了使用script之外，唯一可以用來縮短指令長度的方式。

17.4.1 在C shell中使用別名

C shell提供了很多別名可用的擴充功能，包括命令列參數。alias敘述句使用了兩個參數：別名名稱及其定義。下面的敘述句為ls -l的縮寫：

```
alias l ls -l
```

在C shell中不使用=符號

一旦使用這種方式定義後，想執行ls -l時，直接輸入l即可，即使這個別名和一外部指令，或是內建指令相同也可以。現在就來看看，當使用這個別名及一些檔名時，會發生什麼事：

```
$ l relaydenied.html blankdel.sh
-rwx------   1 sumit    dialout      75 May 23 16:22 blankdel.sh
-rw-r--r--   1 sumit    dialout    7899 Jun 14 20:04 relaydenied.html
```

使用者也許會以為，別名l使用了兩個參數來完成工作，但很不幸的，事實並非如此。在執行這個指令時，shell純粹只是將這個別名置換成它所代表的指令，然後再

執行置換後加上參數之整串命令而已。但是C shell的alias指令的確可以接受參數，它會被讀入alias內一些特殊的位置參數中，以下兩個參數是必須要知道的：

\!* ——代表命令列所有的參數
\!$ ——代表命令列最後一個參數

這些表示式是從歷史指令機制而來，它們有著類似的意義，除了這裡的表示式是相關於上一個指令之外。我們可使用最後一個表示式來設計找尋一檔案的別名：

```
alias where ‘ find / -name \!$ -print ’                    ! 是特殊的
```

現在，我們可以執行這個別名來找尋一個檔案，就從根目錄開始找起：

```
where pearl.jpg                          此時!$記錄著pearl.jpg
```

我們必須讓參數**!$**跳脫，以防止shell將之置換成上一指令的最後一個參數，如此，**\!$**才能記錄目前指令的最後一個參數。如果使用者一直遵循著先前的一句箴言「單引號防護所有特殊字元」，那現在就是第一個例外，因為那並不防護**!**，只有****才是。實際上，這裡完全不需要引號。

但是，如果必須使用其它特殊字元的話，引號當然是需要的，就像**|**。以下是用來列出檔案的別名，一次顯示一頁：

```
alias lsl 'ls -l \!* | more'
```

這次我們必須使用**\!***，因為這個別名必須和多個檔名同時工作：

```
lsl 123.html thanks.ppt r*.html                    可使用萬用字元
```

如果這裡使用了**\!$**，那麼就只會列出最後一個檔名。這個別名不必加上參數也可執行，如此它就會執行ls -l | more。但如果我們執行where而不加上參數呢？結果可能不是使用者所期待的，因為where這個別名會尋找一叫做“ where ”的檔案，它將**\!$**解譯為指令列最後一個字，而這個字就是這個別名where本身！

C shell能做的不只是這樣，它還可讓使用者個別存取每一參數，這項功能也是從歷史指令機制而來，並使用了**\!:***n*這個敘述句，這裡的*n*可代表一數字或是一範圍，甚至是一些特殊字元。現在就來更改where別名，以讓它可以接受第二個參數，而且能夠儲存其輸出：

```
alias where 'find / -name \!:1 -print > \!:2'
```

這次**\!**後面接著一個**:**以及一個代表命令列參數位置的數字。這裡也可以使用範圍敘述，**\!:3-6**代表第**3**到第**6**的參數。這個數字化系統使用於歷史指令功能，而這些

特性的完整列表在第17.5.1節會介紹。

alias也有簡單的應用，當使用者時常進入某個有很長路徑名稱的目錄時，就可設定一個別名來進入這個目錄：

```
alias cddoc cd /usr/documentation/packages
```

或者使用者也想重新定義ls指令，使用永遠以多行來顯示，並且標示可執行檔及目錄：

```
alias ls ls -xF
```

使用unalias指令可以移除別名的設定：

```
unalias where
```

使用alias指令而不加參數的話，會列出所有別名之定義，但是，也可使用一參數來顯示某一特定別名的定義：

```
% alias where
find / -name !:1 -print > !:2                    這裡沒有顯示 \
```

如果使用者沒有選擇而必須使用C shell，就一定要精通如何使用別名。有些使用者可能單純只是想要藉由使用別名，來作為學習shell函式的墊腳石，這些函式使用位置參數$1及$2而不用\!:1及\!:2。不管相不相信，C shell的別名也使用*if-then-else-endif*這種條件判斷敘述式，除了必須加上大量的*s*之外。


Note

如果使用者利用別名來定義一外部或內建指令，則原指令仍可在其前面加上\而執行，也就是說，如果where**這個指令存在於系統中的話，使用者仍可執行這個指令，只需使用**\where。

17.4.2 在Korn shell及bash中使用別名

在Korn及bash shell中，也可在執行任何工作之前使用alias指令，任何使用者使用最多的可能是ls -l指令，如果在系統中沒有l指令的話，就可自己建立一個別名：

```
alias l=' ls -l '                    這裡需要加上 = 號
```

這裡的alias需要加上=號，但是其前後不可以有空格，如果這個值包含空格的話(就像這個例子)，就必須使用引號將其括起來。現在就可執行ls -l這個指令，只需執行：

```
l                                    這指令會執行 ls -l
```

我們時常會使用cd指令來進入有著長路徑名稱的目錄，如果有一目錄是經常使用的話，設定一別名來轉換這個指令會是明智的抉擇。請看以下這個別名：

```
alias intcd=”cd /usr/spool/lp/interface”
```

這些別名可像script一樣，可接受命令列的參數嗎？我們可以使用l別名加上一些檔名嗎？現在就來試試下面的指令：

```
$ l addbook.ldif apacheFAQ.html
-rw-r--r--   1 sumit    dialout      1555 Feb  8 14:03 addbook.ldif
-rw-r--r--   1 sumit    dialout    103485 Dec 13  1999 apacheFAQ.html
```

這個指令可正常執行，但那是因為shell將l別名取代為ls -l，然後再執行ls -l加上這兩個參數。雖然那看起來像是使用了一個別名再加上兩參數，其實不是，Korn及bash的別名並不能使用參數，但如果一個指令使用檔名來當作它最後的參數，我們就可使用這個性質來重新定義這個指令，這就是接下來要介紹的部分。

當目的地檔案已存在時，cp -i指令就會詢問使用者，而使用別名時，就可讓cp指令永遠詢問使用者。同樣地，為了小心起見，也可使用相同概念來重新定義rm：

```
alias cp=”cp -i”
alias rm=”rm -i”
```

現在，每次執行這些指令時，就會執行已別名化的版本，那麼，如何使用原來的外部指令呢？只要在前面加上\即可，也就是說，使用者必須使用\cp foo1 foo2來覆蓋先前別名的設定。

我們可以使用alias加上別名名稱來顯示這個別名的定義：

```
$ alias cp
cp=’cp -i’
```

***bash* 之輸出有點不同**

如果使用alias而不加上任何參數的話，就會列出所有的別名定義，而使用unalias敘述句則可取消別名的設定。要取消cp別名的設定，可使用unalias cp。使用別名可大大提升效率，以下是本書作者所使用的一些重要別名：

```
alias ..=’cd ..’
alias ...=’cd ../..’
alias dial=’/usr/sbin/dip -v $HOME/internet/int3.dip’
alias dialk=’/usr/sbin/dip -k’
alias mailg=’fetchmail && elm’
alias mails=’/usr/sbin/sendmail -q’
```

這些別名大部分都是事先定義好的，也有一些是作者定義用來加速網際網路之運作。如果常常使用cd ..及cd ../..的話，就會發現..及...別名相當有用。dial別名使用int3.dip這個script來使用數據機登入ISP的電腦，dialk則會中斷這個連線。mailg會從郵件伺服器收取信件，如果有新郵件的話就呼叫elm來處理，而mails會

將未寄出去的郵件傳送給外寄伺服器。所有這些功能在本書中都會介紹，但只要注意這些基本的網際網路動作，是如何使用少數幾個別名就可處理。


Tip

但在定義別名時，要稍微克制一下，太多的別名定義常容易造成混淆，且不好記。此外，別名可完全由shell函式(19.10)來取代，它可以提供比別名更好的功能。使用別名是很好的開始，但最後你會選擇使用shell函式。

17.5 指令歷史(history)

Bourne shell一個很嚴重的缺點是，如果想要重新執行某個指令的話，就必須重新輸入一次，而另外三個shell則提供了許多不同的**history**功能，可讓使用者叫回先前已執行過之指令（即使是在上個shell就已執行的指令），並且修改及重新執行它們。shell會指定一事件號碼**(event number)**給每一個指令，並且可能（取決於所使用的shell）會將所有指令都儲存於歷史檔案中。

history這個指令會顯示每個已執行過的指令，及其相對的事件號碼，而使用一個符號，就像!或r，再加上事件號碼，就可以再度執行某個指令。這個歷史指令的最大容量，及其所儲存的檔案，決定於兩個shell特定的變數，C shell及bash的功能較相同，而Korn則使用了不同的符號，表17.3中概述了這些歷史指令的功能。

17.5.1 C shell及bash的歷史功能

就像alias一樣，history也是一個內建(built-in)的指令，預設上，這個指令顯示了它所維持之串列中所有的事件：

```
12 % history
      .....這裡省略了幾行 .......
   8   grep “William Joy” uxadv??
   9   cd
  10   pwd
  11   doscp *.awk /dev/fd1135ds18
  12   history
```

每個指令前都有一個事件號碼，而每個執行過的指令都會加入到這個串列之中，最後一個指令則顯示了這個history指令本身。這個串列可能會相當大，所以可加上一個數字參數來限制所顯示的大小，例如，history 7只會顯示最後七個指令。

shell可將指令儲存於記憶體或是檔案中，它會使用一變數來決定儲存於記憶體中之串列的大小，並會使用另一變數來決定所儲存檔案的大小。它的區別如下：

csh	bash	**意　義**
$HOME/.history	$HOME/.bash_history	儲存歷史指令的檔案
savehist	HISTFILESIZE	儲存歷史指令檔案的大小
history	HISTSIZE	儲存於記憶體中之串列的大小

表17.3　歷史指令功能

csh,bash	ksh	意　義
history 12	history -12	列出最後12個指令
!!	r	重複執行上一個指令
!7	r 7	重複執行事件號碼為7的指令
!24:p	—	列出事件號碼為24的指令，但不執行
!-2	r -2	重複執行上一指令之前一個指令
!ja	r ja	重複執行上一個以ja為開頭的指令
!?size?	—	重複執行上一個內含size的指令
!find:s/pl/java	r find pl=java	重複執行上一個find指令，並將pl置換為java
^mtime^atime	r mtime=atime	重複執行上一個指令，並將mtime置換為atime
!cp:gs/doc/html	—	重複執行上一個cp指令，並將所有的doc置換為html
!! | sort	r | sort	重複執行上一個指令，並將結果輸出至sort指令
!find | sort	r find | sort	重複執行上一個find指令，並將結果輸出至sort指令
cd !$	cd $_	變換目錄至上一個指令的最後一個參數（bash也使用$_）
rm !*	—	移除上一指令之所有參數所代表的檔案
vi !:2	—	執行vi，並以上一指令之第二個指令為參數
grep http !30:4	—	執行grep http，並以事件號碼為30的指令之第四個參數為參數
more !:$	—	執行more，並以上一指令之最後一個參數為參數（並不一定需要：）
!:0 foo	—	重覆執行上一個指令，並以foo作為參數
echo !$:h	—	顯示上一指令之路徑名稱的頭部
echo !$:t	—	顯示上一指令之路徑名稱的尾部（檔名）
echo !$:r	—	移除上一指令之檔名的副檔名

此外，C shell及bash也有差不多相同的指令及敘述句來處理歷史指令的功能，通常使用者不會更改這個歷史檔案，但可取代其預設的大小：

```
set savehist = 1000          在C shell 中儲存於 .history
HISTFILESIZE=1000            在bash 中儲存於.bash_history
```

同樣地，也可設定儲存於記憶體中的指令數目：

```
set history = 500            儲存於記憶體—C shell
HISTSIZE=500                 儲存於記憶體—bash
```


Tip

以下可設定別名，使其只顯示最後十五個指令：

```
alias h ‘ history 15 ’       C shell
alias h= ’history 15 ’       bash
```

除非設定了**history**變數，否則它的歷史功能是不能啟動的。如果沒有設定這項功能，則只有最後一個指令會被儲存，即使它是存在記憶體中也一樣。

和C shell不同的地方在於，它儲存歷史指令的檔案取決於**HISTFILE**變數的設定，如果這變數未設定的話，則使用**$HOME/.bash_history**檔。

重複先前的指令(**!**) **!**這個指令是用來重複先前的指令，它是歷史指令機制的主要符號，可以搭配正或負的整數、字串或其他的!符號來使用。當開始使用這個符號時，就等於開啟了嶄新的世界，接下來的幾頁就會帶領使用者進入這個世界。
想要重複最後一個指令，只需使用**!**兩次：

```
!!                                          重複最後一個指令
```

也可以在!後面加上事件號碼，來重複歷史指令串列中的任何指令，在!及事件號碼中間不可以有任何空白：

```
% !11                                       重複事件號碼為11的指令
11 doscp *.awk /dev/fd1135ds18
```

這個指令列的內容會顯示並執行，但這種方式可能會導致執行了一個錯誤的指令（像**rm**），所以可加上一個修飾句（**modifier**或運算子）**p**（列印出來），就可以只顯示而不執行：

```
% !11:p
11 doscp *.awk /dev/fd1135ds18
```

doscp指令可在硬碟及磁片間複製檔案，在使用前最好能使用**:p**修飾句來確定是否為所要執行的動作，如果是的話，再使用**!!**就可執行這個動作。

使用者也可使用指令的相對位置來執行最近才執行過的指令，方法是在**!**後面加上一負整數，整數部分代表與最後指令相隔的指令數目：

```
!-2                                         在 !及-之間沒有空格
```

很有可能使用者會不記得指令的事件號碼，除了最近執行的兩三個，但至少會記得指令是以那個字元或是那個字串作為開頭的，例如，如果使用者記得最後執行以**v**開頭的指令為vi，則可使用**!**加上**v**或**vi**：

```
!v                                          重複最後一個以 v 為開頭的指令
```

如果這種方式還不能滿足所需的話，還可以比對字串中的某些字，只需在所要比對之樣式的前後加上**?**即可，以下這種方式會比對先前任何一位置含有**xvf**的指令：

```
!?xvf?                                          執行最後一個含有 xvf 的指令
```

tar會使用**xvf**這個選項從一軟碟(floppy)取出檔案，而**cvf**選項則會寫入檔案，所以使用!tar指令來重新執行最後一個tar指令時會很危險，因為想要執行的可能是從磁片中讀取資料，但卻不小心將這些資料覆蓋了。另外，使用!?xvf會是更安全的。是的，也許這樣還是可能會執行錯誤的指令，但是卻沒有任何一指令會使用xvf這個選項啊！這個地方要注意的是，如果指令輸入後馬上按*[Enter]*的話，那麼第二個**?**是不需要的。

在Korn shell中沒有比對歷史指令之部分樣式的功能，但有另外一些指令可用來從執行代換及取出一路徑名稱中的某一部分，我們接著會討論這項功能。


Tip

在!接著一字串可用來叫回最後以此字串為開頭的指令，而使用?將這個字串前後括住則可執行任何含有這字串（包含以這個字串為開始）的指令。當使用一指令而加上很多種的選項時，第二個形式會是較有用的，而且搜尋指令的參數會是這裡所需。如果使用者擔心以這種方式來執行歷史指令的話，會危害整個系統，那麼可使用:p來顯示指令，確定後再使用!!來執行它。

代換先前的指令**(:s)**　這裡會介紹一些用來修改操作動作的修飾句(modifier)，它們很簡單，也不需使用相互對話的方式。例如下面的指令可使用**:s**修飾句(modifier)來再度執行先前的grep指令，但會將**William**置換為**Bill**：

```
!grep:s/William/Bill                                    :為定義符號
```

使用sed時，這個代換動作就不是應用於整個指令列之字串了，此時只有第一個出現的樣式會被置換，否則就要加上**g**參數（也可用於sed）才能應用於整體代換(global substitution)。下面的指令會重複最後一個cp指令，並將所有的**doc**全部取代為**bak**：

```
!cp:gs/doc/bak                                  在Korn是不能這樣做的
```

而要代換上一個指令時，有一捷徑可用，不需使用**!**，那就是使用^符號作為字串的分隔符號而不需使用正規表示式。以下的指令可再度執行原來的指令，並且將**bak**置換為**doc**：

```
^bak^doc                                        只置換第一個
```

使用先前指令的參數（!$及!*）　我們可使用mkdir foo來建立一目錄，接著使用cd foo來進入這個目錄。C shell及bash使用一特殊的樣式來縮短在命令列所需輸入的指令長度，那就是**!$**。這個表示式代表上一指令之最後一個參數，在別名中已使

用過它，但這裡我們要使用**!$**來作為先前指令所使用之目錄的縮寫：

```
mkdir programs                                    將目錄變換為 programs
cd !$
```

考慮另一個例子，如果使用者已使用vi或emacs編輯了一指令手稿檔案cronfind.sh，只要這樣輸入就可執行這個檔：

```
!$                                                執行 cronfind.sh
```

這是多麼棒的方法啊！可以想像執行一剛由vi或emacs所編輯完成的shell或perl的script檔案會是這麼簡單嗎！

!*代表上一指令所有的參數，先前也使用過了，所以如果使用下列方式來檢查是否某些檔案已存在：

```
ls runj count.pl script.sh
```

就可使用!*作為 **rm** 的參數來移除它們：

```
rm !*                                             這個指令執行的是 rm runj count.pl script.sh
```

這個功能在Korn shell中是不存在的。但當然，這裡也有個限制，如果使用ls -l再接著使用rm指令，則rm執行時就會使用-l參數，並且會產生錯誤。

使用歷史修飾句**(modifier)(**:n**)** 假設使用者剛剛使用下列指令來列出一些檔案：

```
ls runj count.pl script.sh add.sh binary.pl
```

現在就可使用**:n**這個修飾句(modifier)來取出上一指令之個別的參數，**n**的值由**0**（指令本身）開始，也可以是任意一範圍間，或是使用*****來取出直到最後一個參數間之所有的參數。**^**及**$**分別代表第一及最後的參數，表17.4列出如何使用vi來編輯其中一或多個這種檔案。

:n修飾句(modifier)並不侷限只能單獨使用於最後一個指令，也可以執行一特定事件的指令，再加上另一事件之所選擇的參數。我們已經存取了一指令的參數，而要存取先前指令則要使用**:0**修飾句(modifier)，例如：

```
!com:0 !47:1-3
```

這個指令會執行上一個以**com**（也許指的是compress）為起始的指令，再加上事件號碼為47之前三個參數。也可執行上一指令，而加上其它指令的參數：

```
!:0 !-2:*                                         !:0 是上一指令但不加參數
```

表 17.4　使用歷史修飾句 (上一指令：ls runj count.pl script.sh add.sh binary.pl)

使用歷史修飾句的指令	實際執行的指令
vi !:1	vi runj
vi !^	vi runj（不需要：）
vi !:2-3	vi count.pl script.sh
vi !:3-$	vi script.sh add.sh binary.pl
vi !:3*	同上
vi !*	vi 加上所有參數
vi !:3-	vi script.sh add.sh（不包含最後一個參數）
vi !$	vi binary.pl（不需要:）

這指令會取出上一命令中之指令部分，並使用再上一個指令的所有參數。要適應這個難懂的參數需要多花些時間，但這個功能也許是人們仍然使用C shell的唯一理由。

除了**!$**之外，bash也使用**$_**來代表最後指令的最後一個參數，這個變數和Korn shell 所使用的一樣。

如果使用者已對一群檔案執行了一些動作，就可使用!*這個符號來執行另一指令來對同一群檔案執行動作，但如果只想使用最後一個檔名，那麼可使用!$，而使用:0則可單獨取出指令名稱。

路徑名稱修飾句**(modifier)**（:h、:t及:r）　想要取出路徑名稱之一部分的話，這裡有三個重要的運算子可用。如果使用其中一個於先前指令的話，就可執行相同（或不同）指令，且加上從這路徑名稱取出之一部分。

現在先來探討**:h**（頭部）這個修飾句，它會取出一路徑名稱的頭部，也就是說如果剛剛使用下列方式來列出一目錄的話：

```
ls /var/spool/lp/adains
```

就可重複相同的過程，但這次處理的是它的父目錄，或甚至是轉換到這個父目錄去：

```
ls !$:h                                          # 執行 ls /var/spool/lp
cd !$:h                                          # 執行 cd /var/spool/lp
```

:t（尾部）修飾句會取出基本檔名，以下方式可在將一檔案複製到其他目錄時，順便加上副檔名：

```
% cat /etc/skel/profile
      ... 列出檔案內容 ...
% cp !$ !$:t.txt                                 複製為 profile.txt
cp /etc/skel/profile profile.txt                 系統回應的訊息
```

:r（根部）修飾句可取出檔名，但會捨棄其副檔名，這個功能在執行Java程式時很有用：

```
% javac hello.java                                    編譯 hello.java
hello.java compiled successfully
% java !$:r                                          執行 java hello
java hello
```

還有一個修飾句**:e**，它可取出一檔案的副檔名。當C shell及bash使用這些修飾句於上一指令之參數時，C shell還可加上變數來使用：

```
% set domain=planets.com
% echo $domain:e                                     只使用於C shell
com
```

Note

C shell及bash有四個Korn shell所沒有的重要指令歷史功能：

- **在歷史檔案中尋找一內含的字串（就像!?xvf?），Korn只能尋找以一字串為起始的指令。**
- **可置換一先前指令（就像!cp:gs/doc/bak）中所有出現的樣式，Korn只能重複先前以一字串為起始的指令，並只置換一行中第一次出現的樣式。**
- **可使用!*這個特殊的符號來代表先前指令所有的參數，及使用:*n*修飾句來選擇性地存取某些參數，Korn只能存取最後一個參數。**
- **Korn沒有提供路徑名稱之修飾句，但擁有更強的樣式比對功能**(19.8.1)。

17.5.2 Korn shell的指令歷史功能

使用history指令而不加參數時，會列出儲存於歷史檔案中之最後十六個指令，而加上**-5**作為參數時，可列出最後五個指令：

```
$ history -5                                         注意 - 這個符號
36 exit
37 alias l=' ls -l '
38 doscp *.pl a:
39 fc -l
40 history -5                                        也包含顯示這個列表的指令
```

這些指令儲存在由**HISTFILE**變數所定義的檔案當中，如果沒有設定這個變數的話，則使用**$HOME/.sh_history**的定義。儲存於事件串列之指令的數目決定於**HISTSIZE**，預設上，它會儲存至少**128**個指令，但為了記錄多個登入的所有指令，可將它設為更大的數目：

```
HISTSIZE=1200                                        儲存至少 1200 個指令
```

每次執行指令時，事件號碼就會自動增加，對於事件號碼的了解對您會很有幫助，

因為在重新執行先前指令時，參照的就是這個號碼。

重複先前指令(r)　Korn使用**r**指令來重複先前的指令，不加參數時，它會重複最後的指令：

```
r                                          重複上一個指令，亦即與 r -1相同
```

使用事件號碼作為r的參數時，就可重複其它指令：

```
$ r 38                                     在 r和38之間有空格
38 doscp *.pl a:
```

有時候使用者會重複交替使用兩個指令，例如用於編輯並編譯一C程式的vi及cc指令，此時可使用相對位置來執行上一指令之前一個指令，以交替執行這兩個指令：

```
r -2                                       另一個要交替執行的指令
```

有時候，使用事件號碼來存取一指令是很重要的工作，特別是想要呼叫一先前事件的時候。有時也可利用字串來存取指令，但這指令必須要以這字串為開頭（不允許嵌入其中）。例如，如果使用者記得最後一個以**v**開頭的指令是vi的話，就可使用r加上**v**或vi來執行：

```
r v                                        重複最後一個以 v為開頭的指令
```

先前指令的置換(=)　有時使用者會想要重新執行先前的指令，但是想要置換其中一些字串。使用r指令加上這個指令名稱，再加上想被置換的字串、一個=符號，最後再加上想要置換的字串即可。例如，想要重複先前已執行的cp指令，並將**pl**字串置換為**awk**，則可使用下列的方式：

```
r cp pl=awk
```

然而，這個置換動作只作用於第一個出現的位置，它並不會置換第二或其他的項目（在C shell及bash中是可以的），如果此時想要置換所有的項目的話，就應該使用線上(in-line)編輯指令，下一節會討論這個指令。


Tip

對於一個程式設計者而言，使用先前指令再加上一部分的字串是非常有用的，特別是需要重複編輯和編譯程式的動作時。例如，如果一直交替使用vi appl.java**及**javac appl.java**指令，則只需輸入這兩個指令各一次，接著在**Korn**中只需分別使用**r v**及**r j**即可，而其它**shell**則只需使用**!v**及**!j**，不再需要記住它們的事件號碼！**

使用先前指令的最後一個參數($_)　使用者通常會使用數個命令作用於同一檔案上，而這個檔案通常會是最後一個參數。除了每次都要指定檔名外，也可使用$_來

代表上一指令之最後一個參數。例如，如果剛剛已執行vi applet.java這個指令，就可執行javac並使用**$_**作為參數來編譯這個程式：

```
vi applet.java                                  $_ 變成 applet.java
javac $_                                        執行 javac applet.java
```

那麼，要如何執行剛剛才由vi foo指令編輯完成的script呢？那很簡單，只需使用**$_**來執行即可：

```
$_                                              執行最後一個參數，以作為指令
```

和C shell不同之處在於，Korn shell並沒有太多的歷史功能，但目前所討論的應可滿足大部分的需求，其它的部分可使用線上編輯功能(in-line editing)來補足。vi及emacs的使用者可能喜歡使用*/pattern*或*[Ctrl-r] pattern*來呼叫先前已執行的指令，bash及Korn有提供這項功能，並且在下一章會討論到。

17.6 Korn shell及bash的線上指令編輯

Korn及bash提供像是vi及emacs那樣命令列編輯功能，可使用於現在及先前的指令，這個功能就是所謂的線上編輯**(in-line editing)**。這兩個編輯器的基本功能都內建於這幾個shell當中，在使用這個功能之前，必須先執行以下的設定：

```
set -o vi                                       只能設定一個，不能兩者同時設定
set -o emacs
```

使用**+o**選項可關閉任一狀態，現在就來看看shell中像vi的編輯功能如何加強命令列編輯的能力。如果知道emacs的話，就會知道如何使自己熟悉這個環境，表17.5列出第四及第五章介紹之一些編輯指令。

現在如果使用者在命令列發現了一個錯誤，而由於已設定使用vi的編輯功能，所以就不必將前面所有已輸入的文字刪除掉，而將游標移至錯誤點。首先，按下*[Esc]*鍵以進入vi的「命令模式」，此時就可使用所有vi的瀏覽指令在這個指令行移動（使用^、$、b及e），如果需要的話，也可加上重複因子(repeat factor)。

此時也可引用這些已使用的指令，以進入「輸入狀態」，像是i、a、A等。使用x可刪除一字元，而使用dw可刪除一字(word)，使用p或P則可將這個字放在這一行的某個地方（或不同行！）。修改完畢後按下*[Enter]*就可執行這個指令。

k指令在vi中可將游標往上移，而在這裡則可重新呼叫先前的指令（但需先按*[Esc]*鍵）。因為這裡也可使用重複因子(repeat factor)，所以5k就代表最後五個指令（除了現在這個之外）。如果這不是想要的指令，則使用j可顯示下一個指令。

使用者可以使用vi的搜尋技術來呼叫先前的指令，即使它的搜尋方向是相反的，還是必須使用*/pattern*這個順序，如果需要的話，也可使用正規表示式：

/find*[Enter]*　　***找出find這個字串最後出現的位置***
/^find　　***找出最後一個 find 指令***

在執行一反向(backward)搜尋時，如果想要轉向而執行順向(forward)搜尋的話，可使用?*pattern*。持續按下n可重複執行上一個搜尋動作，如果過頭了，可按下N來轉向。編輯此行指令，並再一次執行，是不是很特別呢？

表 17.5　有用的線上編輯指令

vi指令	emacs指令	動　作
i	—	插入文字
A	—	插入文字於本行最後
j	*[Ctrl-n]*	向下移動
k	*[Ctrl-p]*	向上移動
l	*[Ctrl-f]*	向右移動一字元
h	*[Ctrl-b]*	向左移動一字元
w	*[Alt-f]*	往前一字組(word)
b	*[Alt-b]*	往後一字組(word)
0(零)	*[Ctrl-a]*	回到本行最前面
$	*[Ctrl-e]*	回到本行最後面
x	*[Delete]* 或 *[Ctrl-d]*	刪除一字元
dw	*[Alt-d]*	刪除一字組(word)
D	*[Ctrl-k]*	刪除至本行最後面
/*pat*	*[Ctrl-r]* ***pat***	找出最後一個包含***pat***的指令
?*pat*	—	往前找出包含***pat***的指令
n	*[Ctrl-r]*	以同方向重覆上一搜尋動作
N	—	以反方向重覆上一搜尋動作
u	*[Ctrl-_]*	取消上一編輯動作

先前曾經說過，線上編輯功能可彌補讓C shell很受歡迎之歷史修飾句(modifier)的不足，一個熟練的vi（或emacs）使用者可輕易利用這些編輯功能來執行一指令，並且加上先前指令的某部分參數。如果使用者記得曾使用過vi foo1 foo2 foo3 foo4這個指令，就可以輕易地執行compress這個指令，並且加上其中一或多個這些參數。按下*[Esc]*鍵，使用/vi來尋找這個指令，並使用cw指令將**vi**置換為**compress**，接著按下*[Enter]*，如果必須刪除最後一個參數，可使用4w移至第四個參數，接著按下dw，就是這麼簡單！

除了使用set -o之外，Korn shell也可使用命令列編輯功能，只需設定以下兩個變數：

```
EDITOR=/usr/bin/vi
VISUAL=/usr/bin/emacs
```

很奇怪的，這裡的路徑名稱並不需要正確設定，VISUAL不論是設為/usr/bin/vi或abcd/vi都沒有關係，shell分別尋找VISUAL及EDITOR變數，看看是否有一字串是以vi或emacs作為結尾，並據此設定相對應的模式。

bash可讓使用者使用游標移動按鍵來重新呼叫先前的指令，就像"DOSKEY"一樣。在這個模式下不能使用set -o指令，如果已使用set -o vi指令，那就再使用set +o vi來閉關這個設定。

17.7 檔名展開(completion)

Korn及bash提供一個功能，稱為檔名展開(**filename completion**)，在這些shell的最新版本中，又增加了以下的功能：

- 可展開作為指令之參數的檔名。
- 可展開指令本身。

也就是說，使用者因此不必輸入完整的指令或檔名，即使是第一次使用它們。只要輸入指令或檔名的某一部分，shell就會盡可能的將其餘部分展開，如果先前已使用過emacs(5.10)之類似的功能，那麼現在對這項功能應不陌生。C shell大部分的版本都只提供檔名的展開，但無法應用於指令。表17.6列出在三個shell中展開功能所需要的按鍵順序。

17.7.1 Korn shell中的檔名及指令展開功能

對於Korn shell的討論是依據*ksh93*版本，因為它同時提供這兩個功能，如果使用的是較舊的版本，就無法使用指令展開的功能。在使用這些功能之前，要先輸入任何能夠啟動像是vi之指令編輯功能的指令，才能開始使用展開機制。

```
set -o vi
EDITOR=vi                                    只允許像是vi的編輯
VISUAL=vi
```

要使用展開功能的話，只需要兩個指令，兩者都需要*[Esc]*這個按鍵。例如使用者有以下以p為開頭的檔案：

```
$ ls -x p*
passwd              patlist             pattern.lst
planets.local.sam   planets.master.sam   planets.reverse.sam
problems.sam        profile.sam         pstree.sam
```

表 17.6　檔名及指令名稱展開功能所使用的按鍵

	csh	ksh	bash
檔名展開	*[Esc]*	*[Esc]* \	*[Alt-/]*或*[Tab]*
顯示檔案列表	*[Ctrl-d]*	*[Esc]* =	*[Ctrl-x]* / 或*[Tab][Tab]*
指令名稱展開	—	*[Esc]* \	*[Alt-!]* 或 *[Tab]*
顯示指令列表	—	*[Esc]* =	*[Tab][Tab]*

而現在想要使用vi來編輯**planets.master.sam**這個檔案。首先輸入**vi**，接著輸入這個字串的一部分，先輸入**pl**，然後按下鍵*[Esc]*\：

```
$ vi pl[Esc]\                                    會變成 planets.
```

命令列馬上變成**vi planets.**（包括.），shell此時會發現現在目錄中有三個檔案都是以**pl**為開頭，而且**planets.**也是它們共同的字串，現在，可再輸入一個**m**，接著再按*[Esc]*\鍵：

```
$ vi planets.m[Esc]\                             會變成 vi planets.master.sam
```

在共同字串**planets.**之後，只有一個檔案接下來的字元是**m**（第二個），使用者會發現shell已自動展開了整行命令列所需輸入的資料，而使用者自己才輸入三個字元及兩次*[Esc]*\鍵，總共就是這樣而已！檔名展開功能列於圖17.1。

還有另外一個使用展開功能的方式，除了要求shell幫忙完成之外，也可要求列出符合輸入字串樣式的檔案列表，這種狀況下，就要使用*[Esc]*=。使用這項功能來重複上個例子，會顯示如下的資訊：

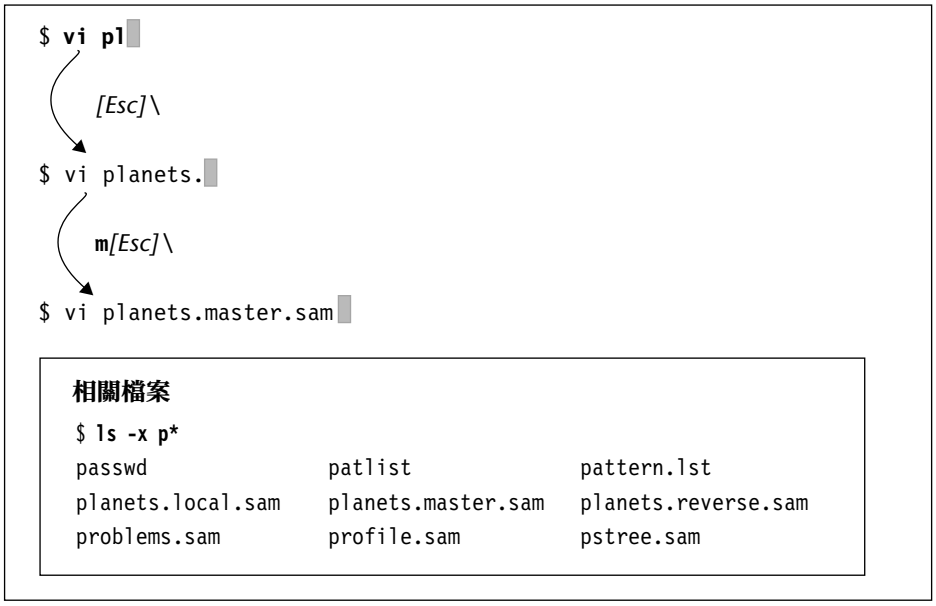
```
$ vi pl[Esc]=
1) planets.local.sam
2) planets.master.sam
3) planets.reverse.sam
113 $ vi pl
```

這次shell列出了三個檔案，游標會停留在未完成之命令列的最後一個字元上面，現在眼前有三個檔名，按下*[Esc]*\鍵將此字串擴展為**planets.**，接著輸入**m**，然後再次按下*[Esc]*\鍵，就完成了這項工作。

在*ksh93*中，已將檔名展開功能擴展為亦可使用於命令名稱之展開。假如使用者想要使用uncompress指令將一檔案解壓縮，只需輸入前三個字元，接著輪流使用*[Esc]*\及*[Esc]*=鍵，就像下面這種方式

```
unc[Esc]\
```

圖17.1 Korn shell之檔名展開功能



shell會將umc擴展為uncompress，這個指令只需五個按鍵動作就可完成。另外如果按下的是*[Esc]*=鍵，則只會看見一檔案符合這個字串樣式。可試著練習使用更短的字串來試試看。


Tip

想知道所使用之Korn shell的版本，只要使用在本節一開始所介紹的命令列編輯功能，然後按下*[Ctrl-v]*，就可看見像是Version 12/28/93f的資訊，這指的是*ksh93*。

17.7.2 bash中的檔名及指令名稱展開功能

Korn中所使用的功能，在bash中一樣可以使用，但是bash之展開功能更強，它甚至可展開主機名稱及電子郵件位址。這裡，我們只討論檔名及指令的展開功能，並使用不同的按鍵順序（使用set -o vi），此時線上編輯模式必須關閉。現在就來重複上一節在Korn 中的例子：

```
$ ls -x pl*
planets.local.sam    planets.master.sam    planets.reverse.sam
```

想要編輯檔案planets.master.sam，只需輸入這個字串的獨特部分，亦即此時只需輸入到pl。接著，使用*[Alt-/]*：

```
$ vi pl[Alt-/]                                    會變成 planets.
```

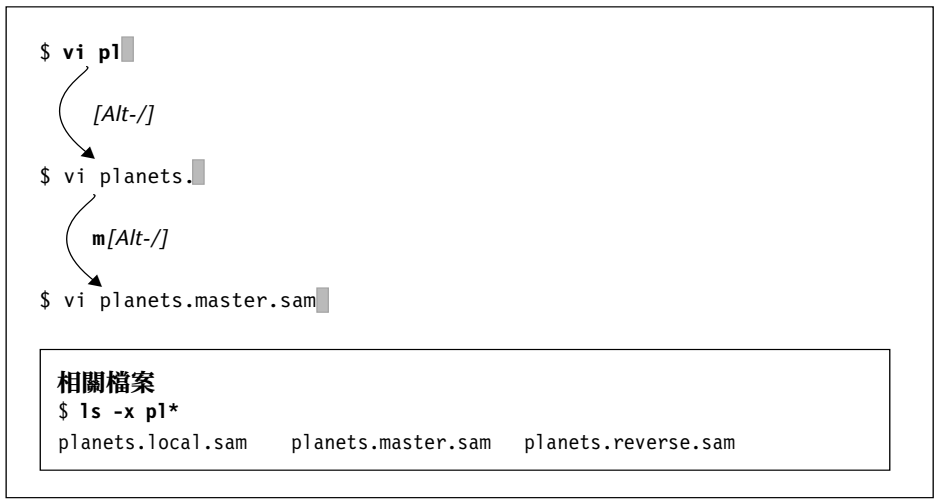
bash會執行指令展開的功能，並將**pl**擴展為**planets.**（包括.）。既然有三個檔案都以**pl**為開頭，而且都有一共同字串**planets.**，接著就可輸入**m**，並使用*[Alt-/]*來展開這個字串：

```
$ vi planets.m[Alt-/]                    會變成 vi planets.master.sam
```

由於**planets.**字串後，只有唯一一個檔名接下來的字元為**m**（第二個），所以只須輸入三個字元及按下兩次*[Alt-/]*鍵，就能完成這個擁有十八個字元的檔名！檔名展開功能列於圖17.2。

除了使用ls來列出檔案之外，也可讓bash來做這件事。這次，必須使用另外一個按鍵順序，即*[Ctrl-x]/*。再來重複一次上一個練習，但這次使用按鍵順序：

圖17.2　bash中的檔名補齊功能




```
$ vi pl[Ctrl-x]/                          在按下 / 要先放掉 [Ctrl-x] 鍵
planets.local.sam    planets.master.sam   planets.reverse.sam
$ vi pl
```

在看過這個列表後，您可以理解下一步應該要做什麼，只要按下*[Alt-/]*鍵就可以將這個字串擴展為**planets.**，接著輸入**m**，並再次按下*[Alt-/]*鍵即可，這就是所有需要做的動作。

bash也提供指令名稱展開的功能，例如使用者想要使用gunzip指令將一檔案解壓縮，首先輸入前兩個字元，接著按下*[Alt-!]*：

gu*[Alt-!]*需按下 *[Shift]*鍵

shell會將**gu**擴展為**gunzip**，如果想要列出能夠比對整個字串之指令，而這指令是記錄於**PATH**中之目錄下所有的指令之一，則只需在這個字串後按下*[Tab]*鍵兩次即可。


Tip

這個只含有一半說明文件的bash功能，可讓使用者按下*[Tab]*鍵就可執行檔名或指令名稱展開的功能。輸入字串的一部分，按下*[Tab]*鍵一次，檔名或指令名稱就會儘可能地展開，想要顯示這個串列，則只需按下*[Tab]*鍵兩次。毫無疑問的，這會是最受歡迎的方式，因為它不管是否已經使用`set -o vi`指令來啟動線上編輯功能，都可使用這個展開功能。


C Shell

C shell大部分的版本都有提供檔名展開的功能，但是卻不提供指令名稱的展開功能，不過，只要這個指令是位於現在的目錄，就可將之當成檔案來處理，而使用這項功能。如果使用者使用的版本有提供檔名展開的功能，則可使用`set filec`指令來開啟這個功能。按下*[Esc]*鍵可執行展開功能，而按下*[Ctrl-d]*鍵則可列出所有符合的檔案。

17.8 其它各式各樣的功能

在繼續探討shell用來初始化的script之前，先來考慮可能有時會用到的一些功能：

- 保護檔案以防意外覆蓋。這項功能就是noclobber，一旦設定了這項功能，就可防止shell使用>及>>符號將這個檔案覆蓋掉。
- 防止使用*[Ctrl-d]*來登出系統。使用者想要中斷標準輸出時，通常會不小心按到*[Ctrl-d]*，所以就登出了系統，而ignoreeof這項功能可提供一安全的防護機制來防止這件事的發生。
- 使用波浪符號~當作家目錄的標記。

這些功能在Bourne shell中並不提供，而在其他shell中實作的方式也不同。

17.8.1 C shell中的noclobber及ignoreeof

C shell使用set敘述句加上**noclobber**參數來防止不小心覆蓋了檔案：

```
set noclobber                                    不能再使用 > 來覆蓋檔案
```

如果現在將指令輸出導向一個已存在的檔案**foo**，那麼shell會回應一訊息：

```
foo: File exists.
```

要使這項保護失效的話，必須在>後面加上!：

```
head -5 emp.lst >! foo
```

另外，使用**ignoreeof**參數可防止不小心使用了*[Ctrl-d]*而登出系統：

```
set ignoreeof                                    按下 [Ctrl-d] 無法登出系統
```

現在如果使用*[Ctrl-d]*來終止連結的話，就會看到下面shell典型的回應：

```
Use “ logout ” to logout.
```

現在就必須使用C shell的logout指令來登出，使用exit指令亦可。

17.8.2 Korn及bash中的noclobber及ignoreeof

set敘述句提供-o選項，以便可利用這些功能。要防止意外覆蓋掉一檔案的內容，則必須以下列方式來使用**noclobber**參數：

```
set -o noclobber                                 不能再使用 > 來覆蓋檔案
```

也就是說，如果現在將輸出導向一已存在的檔案**foo**，那麼shell會回應一訊息：

```
bash: foo: cannot overwrite existing file                    bash
ksh: foo: file already exists                                Korn
```

要使這項保護失效的話，必須在>後面加上|：

```
head -5 emp.lst >| foo
```

使用set -o加上**ignoreeof**參數可防止不小心使用了*[Ctrl-d]*而登出系統：

```
set -o ignoreeof                                 按下 [Ctrl-d] 無法登出系統
```

現在如果使用*[Ctrl-d]*來終止連結的話，就會看到下面shell典型的回應：

```
Use ‘ exit ’ to terminate this shell
```

現在就必須使用exit指令來登出，*[Ctrl-d]*已經不會產生作用了（除非重複一直執行）。

set的選項可使用set +o來關閉，要再度恢復noclobber功能，可使用set +o noclobber。**set**完整的選項列表可使用set -o或set +o來獲得，不需其它額外的參數。

Bash提供像是C shell的logout指令，它可用來終止一連結。

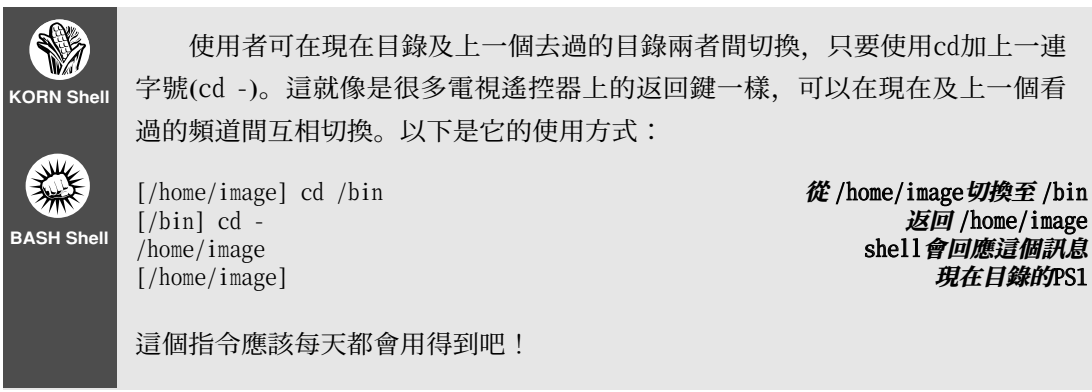
17.8.3 波浪符號之代換

波浪符號~是家目錄的標記，除了Bourne shell之外，其它shell都可使用。當使用~再加上一登入名稱時，如下所示：

```
cd ~juliet                                    效果等於 cd $HOME/juliet
```

就會切換到juliet的家目錄去，如果$HOME之值為/home/juliet/html的話，cd ~juliet指令就會將使用者位置變換至這個目錄去。

有趣的是，當單獨使用~時，它會參照到家目錄，如果使用者登入名稱為juliet的話，就可在家目錄下使用cd ~/html來存取html目錄，這也就是為什麼我們常常看到像是.profile這種組態設定檔，會被參照成$HOME/.profile和~/.profile的原因。



使用者可在現在目錄及上一個去過的目錄兩者間切換，只要使用cd加上一連字號(cd -)。這就像是很多電視遙控器上的返回鍵一樣，可以在現在及上一個看過的頻道間互相切換。以下是它的使用方式：

```
[/home/image] cd /bin                  從 /home/image 切換至 /bin
[/bin] cd -                            返回 /home/image
/home/image                            shell會回應這個訊息
[/home/image]                          現在目錄的PS1
```

這個指令應該每天都會用得到吧！

17.9 初始化script

直到現在，我們已經設定了很多環境變數、定義了別名及使用了set選項，這些設定只在一連線期間內有效，但是一旦登出後，全部又會還原為預設值。想要永久改變這些值，就要將之放在系統的起始script之中。

每個shell至少會使用一起始script，它放置於使用者的家目錄中，就像是在Windows中的AUTOEXEC.BAT檔案一樣。這個指令手稿當使用者一登入時就被執行，有些shell在使用者登出系統時也會執行某個特定檔案。在家目錄下使用ls -a指令，就會看到一個或多個這種檔案：

- .profile (Bourne shell)
- .login, .cshrc 及 .logout (C Shell)
- .profile 及 .kshrc (Korn Shell)
- .bash_profile（或 .profile 或 .bash_login), .bashrc 及 .bash_logout

(bash)

這裡會介紹的script均屬於下列三種之一（亦可參考表17.1之最後三個項目）：

- 登入的script——為使用者一登入時就執行的起始script，C shell使用**.login**，Bourne及Korn使用**.profile**，而bash使用**.bash_profile**來作為登入的script。
- 環境設定的script——在已登入的shell中執行sub shell時會執行的檔案。通常稱為**rc script**，除了Bourne之外，其它所有shell都有一設定檔（**.cshrc**，**.kshrc**及**.bashrc**）。嚴格地說，Korn shell的*rc* script並不限定於**.kshrc**，但通常都是這個檔。
- 登出的script——只有C shell及bash使用登出的script（**.logout** 和 **.bash_logout**），它會在使用者登出時執行。

在接下來的章節中，會介紹這些檔案的運作方式，以及如何在這些檔案中放入有用的項目。但在此之前，我們必須先分清楚登入的是哪個shell，而執行這些script的又是哪一個，也必須學習如何執行這些script，而不需再建立一個sub shell。

17.9.1　.（點）及source指令

在執行script時，現在的shell會產生一sub shell來執行在這個script中的指令，此script中變數之指定及檔案之改變並不會顯示在原先的shell當中，但這是意料中的事，因為在sub shell中所做的改變不會顯示在父shell當中(10.2)。

當登入的shell在執行初始化的檔案時，就全然是另外一種方式了。定義在這些檔案中的變數在登入的shell中是有效的，而即使在這些script執行完畢後，目錄的改變也是永遠的。我們可以說，這些script是由登入的shell來執行，而不是再建立一新的sub shell來執行。

有兩個指令在執行時，不需要建立一新的sub shell，即.（點）及source指令。C shell使用source，Bourne及Korn使用.（點），bash則兩者均使用。與某些人之認知不同之處在於，如果改變了一起始script，並不需要登出並再次登入，才能讓這個script的改變生效，只需用下列方式來執行這個script：

```
. .profile                                  Bourne 和 Korn shell
source .login                                        C shell
. .bash_profile                                       bash
source .bash_profile                  bash—同時使用 . 和 source
```

以這種方式執行後，在目前的shell中這些改變就會生效。在接下來的章節中，也會需要使用這項功能來執行包含shell函式的檔案。


Note

.（點）及source指令可執行一script而不用另外產生一sub shell，它們也不需要擁有可執行的權限。如果PATH變數不包含現在目錄的話，那麼大部分這些指令運作就會

不正常，因為它們無法找到所需的檔案，除非使用了絕對或相對的路徑名稱。如果PATH中沒有.，則. .profile**指令將不會執行，那就必須使用**. ./.profile**。即使如此，在**C shell**中還是能正常執行。**

17.9.2 互動式(interactive)及非互動式(noninteractive)的shell

從另外的觀點來看，每個shell都可更進一步來分成兩個部分，互動式及非互動式。當使用者登入時，顯示提示符號並等待使用者之輸入的shell，就是互動式的**shell**。一個互動式的shell可允許工作排程(job control)、命令列編輯以及歷史指令功能，它也使用與非互動式之shell不同的方式來回應訊號(signal)。有些像是vi及emacs的UNIX指令會提供shell跳脫(escape)功能，讓使用者回到一互動式之sub shell的提示符號下執行動作。

當使用者在shell執行script時，就會呼叫一個非互動式的**shell**，由於很多互動式shell的參數並不傳遞給非互動式shell，所以就必須個別來指定這些參數。互動式的shell會執行起始（登入）的script，以及一個別的*rc*（環境）script檔案，當一非互動式的shell從一互動式的shell中執行的時候，它只執行*rc* script檔案，這個檔案包含所有sub shell必須知道的所有設定。

現在就來討論這個Bourne shell所使用的**.profile**檔案，即使使用者所使用的shell是不一樣的，還是應該要了解這個檔案的內容，因為它也包含可應用於其他shell的相關概念。


Note

環境變數及別名（與set**選項一樣）的區別必須先弄清楚，這些變數在**sub shell**中仍是有效的，但是別名及**set**選項在**sub shell**中就不會自動生效了，因此它們就被放在***rc***檔案中，並傳遞給互動式的**shell**，而在**C shell**中這項功能則比較完善，別名在**C shell**的**script**中都是可使用的。**

17.9.3 Bourne Shell：初始化的script(.profile)

在家目錄下執行ls -a指令，看看**.profile**檔案是否存在，這個登入的script在建立使用者帳號時就應該會被加入才對，但即使沒有這個檔案，使用者還是可以自己來建立。當使用者登入時，shell會執行下列兩個檔案：

- 一全域初始化檔案**/etc/profile**，這個檔案會設定所有使用者共同的變數及可執行的指令。
- 使用者自己的**.profile**檔案，它包含了使用者自己的設定。

依據使用者需求的不同，這個**.profile**檔案可能會相當大，但下面這個例子對於認識這個檔案會很有幫助：

```
$ cat .profile
# 使用者的 $HOME/.profile - 登入時所執行的指令
MAIL=/var/mail/$LOGNAME                          # 郵件信箱位置
IFS=
```

```
PATH=$PATH:$HOME/bin:/usr/ucb:.
CDPATH=.:..:$HOME
PS1='$ '
PS2=>
TERM=ansi
MOZILLA_HOME=/opt/netscape ; export MOZILLA_HOME
calendar
mesg y
stty stop ^S intr ^C erase ^?
```

在這個script中設定了一些系統的變數，最後的幾個敘述句顯示這個檔案已被使用者自行修改過。**PATH**已多加了三個目錄於其中，**CDPATH**現在可讓使用者直接存取家目錄及其父目錄下之所有目錄。使用calendar時時提醒約會的時間，是一個不錯的法子，mesg y表達了可讓其他人使用talk(11.2)指令來交談的意願，而這裡也有一些stty的設定。

在執行**.profile**之前，shell已執行了**/etc/profile**，所以也繼承了很多由這個script檔案所定義的變數。這裡必須要執行**export MOZILLA_HOME**的原因在於這個變數是在此處被定義的，所以必須要被輸出(export)，這樣Netscape才能正確執行。而其他像是**PATH**這些變數早已「被輸出」而改變其值了，所以就不需再次執行輸出(export)的動作。

Bourne shell並不提供別名及任何本章討論的進一步功能，所以它不需要rc script。所有定義在**.profile**中的變數在script本身之中必須被輸出，這樣在sub shell中才可以使用它們。Bourne也不執行登出的script，即使依據shell程式設計的觀點來看，這樣子做會使它的功能更為強大。


Note

shell會在執行完這個大家共用的/etc/profile後，才接著執行.profile。系統管理者會在/etc/profile中放置共同環境設定，所以所有使用者都可使用這些設定。如果系統包含/etc/skel這個目錄的話，在這個目錄下也會有一個.profile檔案，這個檔案在使用useradd指令時，會複製到使用者的家目錄下。

17.9.4　C Shell：初始化script（.cshrc、.login及.logout）

C shell並不使用**.profile**作為起始指令，它使用兩個其他的檔案。**~/.logout**檔案（如果存在的話）在登出系統時會執行，而當一使用者登入時，shell會依照下列順序執行三個script：

- 全域初始化檔案：它可能是**/etc/login**或是**/etc/.login**(Solaris)，所有使用者預設上都會執行這檔案中所有的指令。這個檔案不見得會出現在所有系統之中。
- **~/.cshrc**檔案：包含當C shell啟動時所要執行的指令。
- **~/.login**檔案：只有當使用者登入時才會執行這個檔案。

這裡的~/符號代表家目錄。這個執行的順序必須稍微注意一下，因為它可能不完全是使用者所預期的。它和Korn及bash所執行的順序也不一樣，首先，shell會執行在/etc中的登入檔案，然後執行~/.cshrc，在確定是一登入的shell之後，才接著執行~/.login。每呼叫一sub shell時，就會執行~/.cshrc，而/etc/login及~/.login只執行一次。

C shell的環境設定檔會依循一簡單的行為模式，不管是互動式或非互動式的shell，永遠都會執行.cshrc這個檔案（Korn則不一樣），在.login檔案中應該只能包含環境變數之設定，像TERM及那些只需執行一次的指令：

```
calendar
mesg n
stty stop ^S intr ^C erase ^?
setenv MOZILLA_HOME /opt/netscape                    Netscape需要此設定
setenv TERM vt220
```

MOZILLA_HOME及TERM必須使用setenv敘述句來明確地定義，C shell特殊的變數在sub shell中並不會自動地繼承，必須定義於~/.cshrc檔案中：

```
set prompt = ‘ [\!] ‘
set path = ($path /usr/local/bin)
set history = 500
set savehist = 50
set noclobber
set ignoreeof
```

prompt、path及history是C shell的區域變數，只要是放置在這個檔案中的變數值，都會自動傳遞給sub shell，此外，noclobber及ignoreeof的設定並不會輸出到sub shell中，必須每次在sub shell建立時來設定它們。既然別名的定義無法讓sub shell來繼承，所以就必須放在~/.cshrc檔案中：

```
alias l ls -l
alias ls-l ls -l
alias h “history | more”
alias ls ls -aFx
alias h history
alias rm rm -i
```

在經過一段時間之後，使用者可能會發現~/.cshrc增大的速度比~/.login還快，要注意大部分這些設定只在互動式shell中才會有用，或是有意義，但在script中還是可以使用的。

17.9.5　Korn Shell：初始化檔案（.profile及ENV）

既然Korn shell為Bourne的父集，在17.9.3節中所討論的內容都可應用於Korn shell。然而，Korn可使用的功能更為豐富，也有自己的環境*rc* script，這個script的名稱可以任意取，但仍決定於**ENV**這個變數。它通常是**~/.kshrc**這個檔案，由於很多C shell的使用者轉而使用Korn，所以這個檔案的名稱也以相同的命名方式沿用了下來。

當一使用者登入系統時，以下依序是所要執行之script：

- 全域初始化檔案：即**/etc/profile**，它包含所有使用者共同的設定。
- 使用者自己的**~/.profile**。
- 定義於**ENV**變數中的檔案，通常是**~/.kshrc**。

Korn的行為模式與Bourne稍有不同，難怪它的**.profile**檔案也包含不同，甚至更少的項目：

```
export ENV=$HOME/.kshrc                                    輸出及指定動作合而為一
calendar
mesg n
stty stop ^S int ^C erase ^?
```

在Korn所使用的**.profile**檔案中，必須含有一特殊的變數**ENV**，這個變數有兩個目的：

- 定義在登入及呼叫一sub shell時，所必須執行的檔案。
- 執行環境設定檔。

既然**.profile**檔案只有在登入時才會執行，最好將那些只需執行一次的敘述句放在這個檔案中，其餘的敘述句才放在環境設定檔。以下是**.kshrc**檔案中的一些範例項目：

```
MAIL=/var/mail/$LOGNAME          # 郵件信箱位置
PATH=$PATH:$HOME/bin:/usr/dt/bin:.
CDPATH=.:..:$HOME
PS1=’[$PWD] ‘
PS2=>
HISTSIZE=2000
set -o noclobber
set -o vi
alias h=”history -10”
alias cp=”cp -i”
alias rm=”rm -i”
```

嚴格地說，**.kshrc**所設定的環境變數，也可放在**.profile**中，在這種情況下，這些環境變數大多必須要明確地輸出，因為sub shell並不執行**.profile**，但是set -o及alias指令仍必須放在**.kshrc**中。和在C shell中不一樣的地方在於，Korn shell的別

名功能只在互動式shell中才可使用，而在script中並不能使用（alias的**-x**選項可以使用，但這項功能在*ksh93*中被拿掉了）。


Tip

更清楚地說，如果使用者仍想在script**中使用別名功能，可將別名的定義放置於個別的檔案中，例如~/.aliases檔案，然後將**. ~/.aliases**敘述句放在**script**的最前面，那不管**Korn shell**的版本為何，都可使用這項功能。**

17.9.6 bash：初始化的檔案（.bash_profile、.bash_logout及.bashrc）

像Korn一樣，bash也是Bourne的父集，所有在17.9.3節中所討論的內容也可應用於這個shell。bash的功能也是從C shell而來，因此它提供的初始化檔案功能更為精細。當一使用者登入系統時，以下依序是所要執行之script：

- 全域初始化檔案：即**/etc/profile**，它包含所有使用者共同的設定。
- 使用者自己的“profile”，bash會依序尋找這三個檔案，包括**~/.bash_profile**、**~/.bash_login**及**~/.profile**。只要找到一個，就不會再找其他的檔案。
- 定義於**BASH_ENV**變數中的環境設定檔，但只有在其“profile”中有明確指定才行。這個檔案通常是**~/.bashrc**。

bash在使用者登出系統時，也會執行**~/.bash_logout**檔案。為了方便討論，我們假設**.bash_profile**檔案是bash的“profile”。像是在C shell及Korn中一樣，將那些只有在登入時才會執行的敘述句放置於**.bash_profile**這個檔案中：

```
stty stop ^S int ^C erase ^?
mesg n
export BASH_ENV=$HOME/.bashrc
. ~/.bashrc
```

這裡的**BASH_ENV**指定了環境設定檔，如果這個項目沒有設定的話，必須要將它加上去，因為sub shell也會執行這個檔案，但是登入的shell不會自動這樣做，除非明確地在**.bash_profile**中指定這樣做。這就是為什麼使用者會發現在每一個**.bash_profile**檔案的最後放置這個項目(**. ~/.bashrc**)的原因。

環境設定檔應該要包含所有的環境設定、敘述句以及別名的定義：

```
MAIL=/var/spool/mail/$LOGNAME          # 郵件信箱位置
PATH=$PATH:$HOME/bin:/usr/X11R6/bin:/usr/lib/java/bin:/opt/kde/bin:.
CDPATH=.:..:$HOME
PS1="\u@\h:\w > "
PS2='> '
HISTSIZE=1000
HISTFILESIZE=1000
TERM=linux
```

```
set -o emacs
set -o ignoreeof
alias dial='/usr/sbin/dip -v $HOME/internet/int3.dip'
alias dialk='/usr/sbin/dip -k'
alias mailg='fetchmail && elm'
alias mails='/usr/sbin/sendmail -q'
```

在看過**MAIL**變數之值後，使用者可能會猜到現在是在Linux系統中。注意這裡的**PS1**顯示了使用者名稱、主機名稱及現在的目錄，而別名的定義先前已解釋過(17.4.2)。

嚴格地說，環境變數在**.profile**（或**.bash_profile**）中就可以先設定了，而不必在這裡設定，但那些尚未被輸出的變數就必須被明確地輸出，因為sub shell並不會執行“profile”檔案。**TERM**及**PATH**在這裡被設定，所以不需要輸出，但**HISTSIZE**及**HISTFILESIZE**大概就需要被輸出了。

別名定義及set選項必須放置於**.bashrc**檔案中，因為它們並未被輸出，和在C shell中不同之處在於，bash的別名定義只有在互動式shell中才可使用，而且不會傳遞給script使用。使用者也不能使用將這些定義置於一個別檔案**.aliases**中的技巧，就像在17.9.5節之小技巧中所介紹的一樣。


Tip

使用者可使用.bash_logout檔案來儲存任何想要在登出系統時執行的指令，例如顯示時間。Korn則沒有類似的檔案。

bash及Korn shell都是Bourne shell的父集，而bash也有一些從C shell借來的功能。使用Korn shell或bash（如果使用Linux的話）做為預設的登入shell是明智的抉擇，如果本章的介紹更堅定您的想法的話，請告訴系統管理者代為設定。而不論您決定使用那個shell，都保證能夠得到豐富且有助益的經驗。

摘　要

一般而言

某個shell的指令在其他shell中可能就不能使用，或是會以不同的方式來執行。Bourne shell擁有最少和C shell相同的功能，Korn及bash shell的功能最豐富，所以是最被建議使用的。

環境變數是shell的變數，它必須被輸出才能在sub shell中使用，UNIX系統的行為大部分是由這些變數的設定來決定。

登入的shell，以及從vi、emacs及telnet指令中跳脫的shell都是互動式shell，而script是由非互動式shell來執行，這兩種shell的行為模式不太一樣。

每一初始化script不需另外產生sub shell就可執行，也因此使得定義在此script中的變數，在登入的shell及其sub shell中都可使用。.（點）指令在Bourne、Korn及bash中都可使用，但source指令只有在C shell（bash也可以）中才可使用。

在所有非Bourne shell中，別名可幫助縮短長指令，只需使用alias敘述句。這些shell也提供*history*功能，它可讓使用者再度執行先前執行過的指令（事件），如果需要的話還可執行代換功能。使用**noclobber**可保護檔案，避免不小心覆蓋掉；使用**ignoreeof**可防止不小心按下*[Ctrl-d]*而登出；波浪符號(~，tilde)則代表了起始目錄。

Korn及bash shell則適合互動式及非互動式之使用。

Bourne Shell（也可應用於**Korn**及**bash**）

使用環境變數，就可設定指令搜尋的路徑(**PATH**)，及目錄搜尋的路徑(**CDPATH**)。使用者的家目錄及登入shell決定於**/etc/passwd**檔案，並且記錄於**HOME**及**SHELL**當中。為了讓螢幕正常顯示，**TERM**必須正確設定，shell會知道郵件信箱的位置(**MAIL**)，以及多久才去檢查新進的郵件(**MAILCHECK**)。

export指令會將一般變數轉變為環境變數，這些變數的改變儲存於**$HOME/.profile**中，這個檔案在使用者登入時，會被shell執行。

C Shell

set敘述句會設定區域變數的值，也會顯示它們的定義。setenv指令會設定環境變數，而不加參數時則會顯示這些設定。像**TERM**這些大寫變數會被程式所讀取，但C shell自己則使用小寫的變數。

使用者可設定指令搜尋路徑(**path**)，以及提示符號的字串(**prompt**)。家目錄記錄於**home**變數中，現在目錄記錄於**cwd**變數，shell變數記錄**SHELL**的名稱，**mail**變數則記錄郵件的目錄。很多這些特殊變數和環境變數一樣，也可以是大寫。

別名定義時不需使用等號(=)，**\!$**這個符號代表最後或是唯一的參數，而**\!***則代表所有的參數。

先前執行過的指令可以儲存在記憶體(history)，或是**.history**(**savehist**)檔案中，使用**!**再加上一字串可執行一先前執行過的指令，而**!!**會重複執行最後一個指令，**!$**及**!***則分別代表最後指令之最後一個及所有的參數。

使用**:s**可以代換先前執行過的指令，代換上一指令的話，則可使用^*s1*^*s2*。

:*n*運算子（*n*從0開始）會取出上一指令之一或多個參數，另外也可執行先前指令，但加上另一組參數(**:0**)。而在路徑名稱中，可取出其頭部(**:h**)、尾部(**:t**)及基本檔名但不加副檔名(**:r**)。

使用set noclobber及set ignoreeof可防止不小心覆蓋掉檔案，以及不小心登出。

當使用者登入時，會執行~/.cshrc及~/.login，呼叫sub shell時則會使用.cshrc，而登出時則會使用.logout。別名定義及特殊小寫的變數應該要放置於.cshrc中。

Korn shell

Korn shell使用Bourne shell中所有的變數，使用者可以使用目前的目錄(PWD)及事件號碼(!)作為提示符號字串，以讓使用者能較清楚目前狀態。

定義別名時必須使用等號(=)，而且它不接受任何命令列參數。

先前執行過的指令儲存於~/.sh_history中（或是定義於HISTFILE中的檔案），r加上一字串可重複先前的指令，而單獨使用時，可重複上一指令。$_代表最後指令的最後一個參數。

像是vi或emacs的線上編輯功能可使用set -o （使用vi或emacs作為參數）來啟動，指令及檔名展開功能可讓shell在命令列中擴展所輸入的字串。

set -o指令可用來保護檔案以防不小心覆蓋(noclobber)，及不小心登出(ignoreeof)，使用cd -可返回先前的目錄。

除了.profile之外，登入的shell及互動式sub shell也可執行定義於ENV變數中的環境設定檔，這個檔通常是.kshrc，環境變數、set -o敘述句及別名定義都應放置於這個環境設定檔之中。

bash

Bash是Bourne shell的父集，並且使用Bourne shell所有的變數。提示字串(PS1)可包含目前目錄(PWD)及事件號碼(!)，也可包含使用者名稱、主機名稱以及目前的時間。

定義別名時必須使用等號(=)，而且它不接受任何命令列參數。

先前執行過的指令儲存於.bash_history（或定義HISTFILE於之檔案）中，!後面加上一字串可執行以這個字串為開頭之最近執行過的指令，使用!!可執行上一指令，$_及!$都代表上一指令之最後一個參數，!*則代表上一指令之所有的參數。

使用者可代換先前指令(:s及:gs)，bash也提供一捷徑來執行上一指令的代換(^*s1*^*s2*)。

使用者可取出先前指令一或多個參數(:*n*)，或執行上一指令但加上不同的參數(:o)。對於一路徑名稱，可取出其頭部(:h)、尾部(:t)及基本檔名但不加副檔名(:r)。

可使用像是vi或emacs的線上編輯功能只要使用set -o 加上**vi**或**emacs**作為參數即可，也可使用指令及檔名展開功能，讓shell在命令列中擴展所輸入的字串。

set -o noclobber是一安全的機制，它可防止檔案不小心被覆蓋，如果設定**ignoreeof**的話，不小心按下*[Ctrl-d]*就不會登出，cd -指令可返回先前的目錄。

當一使用者登出時，bash會執行**.bash_profile**、**.profile**，或**.bash_login**中最先找到的那一個，**BASH_ENV**變數決定**sub-shell**執行時所要執行的檔案，通常是**.bashrc**，所有的環境變數、set -o敘述句以及別名定義均放置於這個檔案中。

自我測驗

有些問題假設使用者使用Korn或bash作為登入的shell，並以此來推論以下的問題。

17.1 如果**TERM**的值為**vt220**，那麼相關的控制檔會放在哪裡？

17.2 shell多久會檢查一次郵件信箱以得知是否有新進郵件？

17.3 如果man指令使用了一個變數來設定它的分頁器(pager)，那麼如果將這個分頁器更改為less？

17.4 who am i指令的輸出會放置於哪一個變數？

17.5 如果使用者擔心使用>>符號會不小心覆蓋檔案的話，可以採取什麼預防措施？

17.6 在執行script時，sub shell也會讀取登入檔案（**.login** 或 **.profile**）嗎？

17.7 一個shell變數何時會變成環境變數？

17.8 如果在家目錄下有**.bash_profile**及**.profile**檔案，則在登入bash時，兩個檔案都會讀取嗎？

17.9 在(i) C shell (ii) Korn (iii) bash中，如何設定歷史指令功能並在記憶體中儲存最後二百個指令？

17.10 在(i) C shell (ii) Korn (iii) bash中如何重複上一個指令？

17.11 想要在命令列啟動像是vi的編輯功能，在(i) Korn shell (ii) bash中如何做相關的設定？

17.12 如果不知道henry之家目錄的絕對路徑，那麼在非Bourne shell中如何“cd”到這個目錄呢？

17.13 **:h**及**:t**這兩個符號的意義為何？

練　習

17.1　如果想無更改登入shell的話，需要請系統管理者協助處理嗎？

17.2　那兩個環境變數的設定需要讀取**/etc/passwd**？

17.3　如何將父目錄加入已存在的**PATH**中？

17.4　如果您想要每分鐘檢查一次郵件信箱，該如何設定？

17.5　在bash中如何設定如下面所示之提示符號字串（使用者名稱為romeo，主機名稱為*niccomail*，而**project5**為現在的目錄）？

```
[niccomail-romeo ~/project5]
```

17.6　假設使用者目前在**/home/romeo/cgi**下，並且在執行cd perl時，想要到**/home/romeo/perl**這個目錄下，需要做何相關設定？

17.7　如果有一設定為**PS1=‘ \!$ ’**，那麼，在Korn及bash中會有什麼樣的提示符號字串？

17.8　寫出一Korn或bash的別名，它只會顯示目前目錄下的隱藏檔。如果這隱藏檔為目錄的話，其輸出會有什麼不同？

17.9　寫出一Korn或bash的別名，它可以(i)只列出目前目錄的可執行檔 (ii)顯示最後一個由vi編輯過的檔案。

17.10　在C shell中如何設定提示符號字串，以反應現在目錄的改變？（提示：設計一cd指令的別名）

17.11　如果已設定**noclobber**來防止檔案覆寫的話，則在(i) C shell (ii) Korn 及 bash中如果真正需要覆寫檔案的話，該怎麼做？

17.12　在**/etc**下有一檔案**profile**（沒有.），這個檔案可執行嗎？

17.13　如果將所有別名定義放置於檔案**.alias**中，那麼如何確保在(i) C shell (ii) Korn (iii) bash之所有sub shell中仍可使用這些別名？

17.14　在(i) C shell (ii) Korn (iii) bash中如何只需一些按鍵動作就可快速的重新執行最後一個fdformat指令？

17.15　下面指令的意義為何？

```
^htm^pdf
```

17.16　在剛剛執行tar -cvf /dev/fd0 *.sh完指令後，在(i) C shell (ii) Korn中如何重複執行這個指令，並使用**.pl**檔案做為參數？

17.17　如果執行$_指令時回應一“ **foo.sh: Permission denied.** ”之訊息，代表什麼意思？

17.18 假設已啟動線上編輯功能，那麼在像是(i)vi(ii)emacs的編輯模式下，如何取出最後一個包含**perl**這個名稱的指令？

17.19 在(i) Korn shell (ii) bash中，在**PATH**所指定的目錄中，列出所有以**z**為開頭的指令之最簡單的方式為何？

17.20 在(i) C shell (ii) Korn (iii) bash可以縮短以下的命令嗎？

```
vi script.c ; cc script.c
```

17.21 在(i) C shell (ii) Korn (iii) bash中，如何變換至上一個所在的目錄？

17.22 互動式及非互動式shell有何不同？shell的何項功能只在互動式shell中才有意義？

17.23 當執行. .profile指令時，系統回應一訊息“**ksh: .profile: not found**”，但是現在目錄下的確有一檔案為**.profile**，為何還會產生錯誤？

17.24 如果**.profile**檔案包含一敘述為“**PATH=$PATH:$HOME/bin**”，在修改這個敘述句後，如果使這項修改生效？

17.25 “**vi !un:1:r**”敘述句的意義為何？