UNIX C Shell

简体中文版

原文出处:

http://netlab.cse.yzu.edu.tw/~statue/freebsd/docs/csh/index-cshell.html

以下内容摘自上述网站

作者：黄天彦（网名：网络农夫）

目录

认识 Shell

在我个人接触 UNIX 的初期，时常会在学习中要求系统管理者帮我做一些系统环境上的改变或权限上的开放。当我开始累积 UNIX 的使用经验后，回头猛然发现，事实上绝大部份当初菜鸟时期的我，问的问题简直是有够呆。当时的我有一点深刻的体认 —— 认识 shell 是使用 UNIX 作业系统的一个基础，这点如果一个身为使用者的人无法做到的话，UNIX 的庞大与强悍将会成为使用者最深的负担与恐惧。

一个使用者如果能先建立对 UNIX OS 机能与适用性的正确认识，保持与系统管理者之间的良性互动。我相信，这样的工作经验将会是愉快的。

Shell 是什么

开门见山的说， shell 是一个程序，负责解译及执行用户所下达的指令。

由于身负如此之重任，所以通常她必须在使用者签入（login）系统后，便必须载入记忆体中，这个 shell 有一个专有名词，我们叫她做 login shell。她会为使用者处理输入、输出及系统的错误信息显示（standard input, standard output and standard error）；并读取特殊的起始档案（startup files）用来设定使用者个人所制订的环境变数与预设变数（environment variables and predefined variables）。当然只要使用者还停留在系统中，shell 便会为使用者解译输入的指令。直到使

用者退出系统前，login shell 都会存在内存中为使用者默默地服务。如果您是身为 UNIX 的使用者而不知道 shell 为您做如此多的事，实在是太对不起她了。

三个主要的 Shell

UNIX 作业系统在这 20 几年的发展过程当中，实际上产生过的 shell 实在是不计其数的多。但在各版本之间通用且具有重要的地位的，只有三个。如果按照产生的前后次序来排列的话，它们分别是 Bourne shell、C shell 及 Korn shell。以下是一个简单的对照表。

Shell	创造者	符号	指令名称
Bourne	S.R.Bourne	$	sh
C	Bill Joy	%	csh
Korn	David G. Kron	$	ksh

Bourne Shell

UNIX 系统最早出现的 Shell。开发于贝尔实验室（Bell Laboratories, Murray Hill, New Jersey.），它的创造者是 SR Bourne，所以命名为“Bourne Shell”。一直到今天，在各版 UNIX 作业系统内所采用的 Standard Shell 一直均是以它为主。所谓的 standard shell 的意思便是

指当一个使用者在没有指定要使用何种 shell 的情况下，作业系统会自动帮你设定的 shell，我们称它为 standard shell。当然它也已经被视为是 UNIX 作业系统必备的一部份了。

因为 Bourne Shell 在执行效率上优于其他的 Shell，所以你也可在 UNIX 系统中找寻到以它的语法所写成的执行档。但是它并不支持 aliases 与 history 功能，同时在 Job control 的功能上也比较简单。它的前台与后台工作是无法任意调换的。在整体的功能上，对今日的使用者而言已明显地有些不足之处。所以也有一些 shell 以它为基础，发展出包含新功能的新版本。譬如，你可以在 UNIX System V Release 4 版本中发现到有一个 shell 叫做“jsh”。它便是改进了关于 job control 功能的 Bourne Shell 版本。另外还有 Free Software Foundation 也有发展一个 shell 叫“bash”，缩写的意思是“bourne-again”。它的整体功能趋近于后来所发展出来的“Korn shell”。不过它并不包括在各 UNIX 作业系统中。

Bourne Shell 所使用的起始档案名称为“.profile”和“.login”。在变数上支援局部变数（local variables）与整体变数（global variables）两种。整体变数必须以 export glo_var_name 的方式宣告之。所支持的控制程序有“if-then-else”，“case”，“for”，“while”及“until”等。但请注意在 Bourne Shell 中是没有“goto”功能的（这是比较严谨的作法）。

Korn Shell

Korn Shell 出现在 Bourne Shell 与 C Shell 之后，在功能上则吸收了 C Shell 的 aliases, history 等。而语法则是与 Bourne Shell 兼容，所以在 Bourne Shell 之下所开发的 shell script，也可以在其中执行。它也是以作者的名字命名的 shell。原作者是 David G. Kron。 Korn Shell 也是开发于美国贝尔实验室（AT&T Bell Laboratories, Murray Hill, New Jersey.）。因为在开发的时间上比较晚，现今有支持它的版本，除了 UNIX System V Release 4 版本已将它列为标准配备外，其他版本的 UNIX 作业系统中并不多见。在大部份的 BSD 版本的 UNIX 作业系统并不支持。

C Shell

C Shell 发展于美国加州州立大学柏克莱分校（University of California, Berkeley, California.），原始版本的 C Shell 创作者是 Bill Joy（这位大哥后来任职于 Sun Microsystems），当时这位大哥还是柏克莱的研究生。由于这个 shell 的语法与 C 语言（C Language）极相似而得此名。此 Shell 对惯用 C 语言的使用者而言无疑是一大福音。它出现的时间相当早，早到在 UNIX Time-Sharing System, Sixth Edition 中便已支援。所以如今各版本的 UNIX 作业系统中，BSD 版本的 UNIX 作业系统均有支持，而 SYSTEM V 版本的 UNIX 作业系统，要找到不支持 C Shell 也实在是“很难”（因为我目前找不到不支持 C Shell 的 UNIX OS）。所以可以说 C shell 已经是今日 UNIX 作业系统的一部份了。

在特殊功能上，它是最早提供“别名（aliases）”、“指令使用记录（history）”与“工作控制（Job Control）”功能的 shell。在今天，这几样功能几乎可以说是 UNIX 作业系统的重要特色。 C Shell 所使用的启始档案名称与前面的两个 shell 不同。它的特殊档案为“.cshrc”、“.login”与“.logout”。在其中所使用的设定语法也不相同，所以在使用上并不能兼容。 C Shell 所提供的变量有预设变量（predefined variables）及环境变量（environment variables）两种。分别以内建指令 set 及 setenv 去设定它们。

在目前 C Shell 也有他的更新版，不过并非由原作者操刀，而是在 1980 年左右，由一群工商与学术界的人士共同合作的成果 -- 也就是今天的'T' C Shell，简称与指令相同叫 tcsh。 tcsh 目前发展的相当不错，除了修更 csh 的一些 bugs 外，还增加了不少新的功能，有空的话不妨换他还试试看有何不同。

C Shell 的运用面

对于一个使用 C shell 作为人机使用界面的 UNIX 作业系统使用者而言，C shell 到底提供了你什么样的功能呢？让我们分别从“指令行模式下”、“制定使用环境”与“程序设计”这三个方面来为你说明。希望您能借此先行建立与了解 C Shell 应用面的概念。

指令行模式下

我相信这应该是使用者最为熟悉的运用层面。所以我使用几个关

键性的功能名词来作为区隔为您分项扼要的说明：

输入/输出复位向（I/O redirection）：

提供一套规则让使用者可将执行所需的输入或执行所得的输出做重导向的组合运用。如重导向输出为档案，或将档案重导向作为指令的输入等。这个项目可以说相当基础，是有必要全盘了解的。

正则表达式（Regular Expressions）：

C Shell 定义了一套关于特殊符号的功能定义叫做 Regular Expressions。一般所提到的通配符以及各项的括符引号、斜线、倒斜线等定义及使用的法则均包含在里面。这一套符号规则例程要说简单吗？可算是简单，只要理解的话，不管在指令的使用或者是 sed、awk 的应用都可事半功倍。（不过老实说，一段时日没接触的我看到了，回头看以前自己写的符号堆，有时也颇具催眠作用）

管道（Pipes）：

Pipes 的功能则是让你轻易地将一个指令的输出结果，作为另一个指令的输入。个人建议您善用这项功能。

过去指令使用记录（history）：

可以迅速地将已使用过的指令再次执行，或者是做部份引数上的修改并重新加以执行。看似简单的理念，不过可以做到相当复杂的变化，可以说是一项非常好用但可以使人懒到极点的好功能。

别名（aliases）：

提供用户自行设定简单的字符串来取代复杂的指令选项或多个指令的连续组合。这一项使用者绝对不可不会。

工作控制（job control）：

在单一的终端机荧幕模式下，掌控具分时多任务特性的 UNIX 作业系统。此功能可能在现今 X-Windows 接口下变的比较不重要了，不过还是建议您多少看一下，因为事实上，这项功能还是相当好用的。

内建指令（Build-in command）：

在这个模式下，C Shell 负责解译与执行 UNIX 指令，同时也提供一些常用的内建指令提升系统的使用效率。这些所谓的内建功能我认为多少还是要了解一点，虽然你有可能常使用而不自觉。

制定使用环境

当一个使用者想要主动去了解如何制订与如何改变使用环境时，便算是已经来到进阶门槛之前了。

以一个使用 C Shell 的使用者而言，登录（login）UNIX 系统后，系统会帮你启动一个 C Shell 作为你的 login shell，此 login shell 将会常驻在记忆体中，一直到你注销（logout）UNIX 系统为止。

C shell 定义了预设变量（predefined variables）及环境变量（environment variables）用来控制使用者的使用环境，设定的方式是在使用者登录系统时由 login shell 读取特殊的起始文件（startup files）作为设定的依据，如果使用者没有属于自己的起始文件，则 login shell 将会读取系统的预设文件来加以设定。

C shell 所定义的三个起始文件分别是“.login”、 “.cshrc”、“.logout”，这三个起始文件的格式是一般的文字档，使用 vi 便可编

辑。在使用者 login 后 login shell 会先读取“.cshrc”然后再呼叫“.login”，依据这两个起始文件的内容制订出使用者的使用环境（至于两类变量应设定在那一个起始档案中，稍后再做讨论）；当使用者要 logout 时 login shell 则会读取“.logout”并执行该特殊文件的设定，然后才会退出系统。

使用者个人的启始文件必须放在自己的 home 目录下，如果你用指令 ls 看不到的话请不用惊讶，因为我们前面已经提过它们是特殊文件，你必须使用指令 ls -a 才会显示出来。如果还看不到，这也用不着奇怪，这可能是因为你的系统管理者为你建立 account 时忘了帮你拷贝的缘故吧。如果您使用了 UNIX 作业系统已有一段时日，到今天您才发觉到您根本没有自己的“.cshrc” 及“.login” 启始文件的话，你一定会质疑它们必须存在的必要性与重要性？因为过去你没有这些档案还不是用的好好的!事实上并非如此，因为当你个人的 home 目录下没有这些启始文件时，shell 依旧是必须去读取系统为你准备的原始的启始文件，所以说如果你没有这些启始文件，你可以在 UNIX 的文件系统找到系统原始的启始文件，以 Sun OS 4.1.X 而言文件的位于/usr/lib 目录内，档案名称是 Cshrc 及 Login。你可以将它们拷贝一份到你的 home 目录下作为参考，以便于你在学习中做为设定的范本。（在此说明，以后我们所讨论的启始文件都将是以你的 home 目录下的必须有这些启始文件为前提。）

以下针对这三个档案分为做功能上的区隔简介（说明的部份仅只是建议，因为这三个档案的使用具有相当大的弹性）。

“.cshrc”文件

对我个人而言，这是个令我印象深刻的特殊文件，也就是曾经把菜鸟时代的我最喜欢乱设定，然后电脑就罢工的特殊文件“.cshrc”（其意大概是为 C Shell resource control ）。对 login shell 而言，这是第一个读取的启始文件（启始文件的位置在用户的 home dir.下）。使用者必须知道一点就是 login shell 产生 sub shell 时，sub shell 也会读取“.cshrc” 文件，但 sub shell 不会再读取“.login”，此运作法则请务必注意。

一般而言“.cshrc”使用者必要作改变的环境变量（environment variables），由于前面提到过 login shell 产生 sub shell 时也会读取“.cshrc”，所以设定的变量是会遗传的这点也请切记（所以在不确定的因素下，不建议自己作不必要的环境变量设定，以免后患无穷）。至于预设变量（predefined variables）的部份，牵涉到的大都是属于 C shell 本身所提供的操作功能，建议您视变量的特性作必要的设定（因为并非全部的变数都适合在“.cshrc”中设定的）。

设定上的实际例子我放在第四、五章中有相关内容时再作说明。

“.login”文件

login 进入 UNIX 系统的方式大致说来可分为从 console login，remote login 两类。前者比较单纯，也就是在主机登录系统，所以屏幕、键盘等设定使用的是主机系统的周边，一般而言都不会有问题。而后者是藉由网络或连线装置从系统外登录 UNIX 系统，譬如由另一个 UNIX 主机使用 rlogin 登录系统；又譬如，从 MS Windows 的个人

电脑使用 telnet 登录系统，或者是从 VT220 的终端机登录系统等等，由于使用的并非是主机的周边，有时候必须针对这种情况先加以判断，然后再加以设定特殊的周边装置参数，否则时常会发生如键盘或者编辑上（特别是使用 vi 指令时）的小问题。

适当的“.login”文件设定可以应付上述的情况，设定的情况得视实际的状况而定。

“.logout”文件

这个特殊档案是 C Shell 专有的，它的用途我想你应该猜的到，就是在你退出系统前才会执行的特殊文件。也就是在你的 login shell 要终止之前，C shell 会到你的 home 目录之下去寻找这个特殊文件，并依其内容的设定加以执行。由于在功能上属于非并必要性，所以在一般的 UNIX 系统中，这个特殊文件并不像“.cshrc”及“.login”在系统中的有预设文件的存在。如果使用者个人需要的话，必须自己使用文字编辑工具作自己的设定。

我个人认为 C shell 的这项功能非常适合来做一些暂存文件的清除，或者是个人工作资料与记录的整理及备份，或者是一些备忘录资料的显示等等。

对于 UNIX 系统的起始文件，最好能有系统的预设文件，这部份应该是系统管理者的责任。使用者个人在有正确的认知情况下，可依据个人的需求再加以调整与设定，当然调整设定后的后果，正确的说使用者应自行负责。对于起始档案，使用者应该要有自己设定的做修改与设定。如果一个 UNIX 系统的管理者与使用者无法共同做到此

点，便应该再自我提升对系统的使用能力。

程序设计

C Shell 虽然是一个负责解译及执行指令的用户界面，但是它的功能却不仅只限于此而已（事实上，在 UNIX 系统中，三个 shell 均具有此特性，而这点特性，便是我喜欢 UNIX 系统的主要原因）。 C Shell 提供在语法上类似于 C Language 的流程控制（control flow），也正是 C Shell 得其名的主因。由于具有此项功能，在程序设计上对于一个学习过 C Language 的使用者而言相当容易适应。同时 C Shell 的包容性上也比 Bourne Shell 及 Korn Shell 来得好。不过事实上，有些 UNIX 作业系统的专家相当反对使用 C shell 来撰写程序，如 Tim O'Reilly 及 Jerry Peek 两位 UNIX 的先进大师在其 UNIX 的伟大著作 UNIX POWER TOOLS 中，就以一个章节的篇幅来说明 C Shell 在撰写程序时会产生的限制及 bugs。（UNIX POWER TOOLS 第 49 章“C Shell Programming ... NOT”）。原因事实上很简单，就是不够严谨，且有一些致命性的 bug 存在。原本我的写作计画中，想对此部份做比较深入的介绍，后来也受到此篇文章的影响，修改过不少 script 范例，不过仍然无法摆脱其阴影。所以目前此部份正在重写中。建议有心想要了解这部份的使用者，可以读一读 GAIL ANDERSON 与 PAUL ANDERSON 合着的『The UNIX C SHELL FIELD GUIDE』，我相信您绝对能有相当大的收获。

C Shell 的基本运用

在 UNIX 系统中，指令均以文件的方式分类存放在文件系统的数个目录中。而 C Shell 也其中的一个“指令”。但它的功能却不止于此，指令们必须透过它所设定的搜寻路径（path）来执行，变数 path 设定错误或指令不在 path 支持的路径之内，指令是无法执行的。除此外亦可设定别名（aliases）来取代惯用的某个甚至数个特定的、一连串的指令，以免除指令太长或复杂难记之痛苦。但在这些复杂的功能未介绍之前先来看看光是指令在运用上的变化。

单一指令（single command）

就整体而言 UNIX 系统的指令的用法通常都有一个固定的架构。指令使用语法的第一项是指令名称，然后接着是指令的选项（options）。有时选项后需加上所需的参数。指令的选项大份均可组合或共同连续使用，功能有时相辅相成；当然也有些选项会有不能混用的情况，但比较少。再来多半便是指令所需的引数，如档案名称（filenames）或目录（directory）。此部份则有 C Shell 所提供的“wildcard”特殊功能可运用。在 UNIX 系统指令群的功能上则是复杂多变应有尽有强大无比。甚至某些指令的功能发展到可以写一本专书且欲罢不能（如“sed”“awk”就是），这类指令在指令行模式下虽然很少用到，但却常在撰写 shell 程序发现它们的踪影，所以说对这些功能强大的工具亦应有所了解。在需要运用时才不至于有心无力。

在 UNIX 系统的指令中，关于用户层次的就有二百多个指令，如

果将管理阶层所使用的指令加进来，那更是有够惊人。有鉴于此，诚心地希望读者能养成看 on line manual 的习惯。并且建立自己的指令归类。因为你对指令的熟悉度将会影响撰写一个 Shell 程序的成功机率，抑或是 Shell 程序的执行效率。

以下由简而繁举例说明几个常用指令：

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

连续指令（multiple commands）

在 C shell 中符号“；”有它特殊作用。当你想在同一行连续下指令时，便可在第一个指令结束后用符号“；”来连接下一个指令。如此可一直连接下去。连续指令的使用格式见下：

command1 ； command2 ； command3；.....

以下我们来看几个实际的例子

1% cd ~/backup ; mkdir startup ; cp ~/.* startup/.

2% cd ~/akbin ; tar cvf /dev/rst8 *.* >& tar.tmp ; rm -r *.* ; logout

在事件 1 我们改变工作目录并紧接着建立一个新目录然后将要备份的文件拷贝到新的目录中。事件 2，我们先改变到要备份的工作目

录然后下备份指令并接着在备份完毕后将档案清除，然后退出系统 logout。以这种方式来连续执行一连串的指令，可让你一口气下完一整个流程的工作，在当你有事想离开时，想要的整个流程作业一次完整下完，相当便利。在实际的应用上，常见到他运用在别名的设定上，关于这点我们将在别名中再为你说明。

群体指令（commands group）

“群体指令”的作用看起来很像“连续指令”，两者均可将一长串指令加以执行。但就执行的“环境”而言，两者却有很大的差异。连续指令在执行时并不先产生一个 subshell 来执行指令。故工作环境会随着指令的执行而产生变动（如上例连续指令执行完毕后工作目录便已改变）。但群体指令并不会产生上述的情况。在执行完指令后它会在保留下指令时的工作环境，不会有任何改变。见下例：

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

这是因为群体指令的执行是先自动产生一个 subshell，再将所有指令交由这个 subshell 来执行。当这些指令虽在执行过程中改变了环境变数，但仅是改变 subshell 的环境罢了，一旦指令执行完毕后，subshell 便会自动结束并回到原来的 shell 中，因 subshell 的变数无

法影响 parent shell，所以原执行的环境将不会改变。

条件式的指令执行（conditional command exection ）

在 C Shell 中支持两个特殊符号“||”、“&&”，用来帮助你做简单的指令执行控制。它是运用一个特殊符号来连接两个指令，以第一个指令执行状态的成功与否来决定是不是要继续执行第二个指令。让我们来列表说明它的用法：

第一个指令执行状态	运用符号	第二个指令执行状态
执行成功	||	不执行
失败	||	执行
执行成功	&&	执行
失败	&&	不执行

特殊符号“||”相当于 else、而特殊符号“&&”则相当于 than。这种相当特殊的指令控制，或许在正常的指令行模式下难得用上，甚至有可能根本用不上。但在 C Shell 程序设计上可是一项相当重要的功能呢。我们来两个实际应用的例子：

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

当指令 grep 如果在档案“ecs.sum”中有找到字符串“error”，则不执行打印。如果没找到任何“error”则执行打印档案“ecs.sum”。

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

上面这个例子第一个指令的部份比较复杂，整个指令的功能是找寻系统的处理程序中是否有“rpc.pcnfsd”在执行，如果有则执行第二个指令 wall 向上机的用户说明，说明的文字内容便是档案“pcnfs.run”。如果找寻不到，则不执行第二个指令 wall。

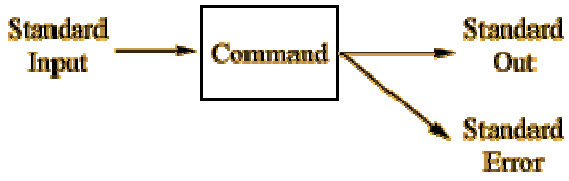
相信读者可能会质疑有可能下这样复杂的指令吗？老实说在指令行模式下真得相当少。但这项功能在 C Shell 程序设计上则会运用得上，千万别忽视。切记！切记！

输入/输出重导向（I/O Redirection）

“输入/输出重导向”是 shell 用来处理标准输入（standard input）、标准输出（standard output）与标准错误（standard error）等信息的重要功能。它可将指令执行的输出导向到档案中，亦可把执行程序或指令所需的引数或输入由档案导入。或把指令执行错误时所产生的错误信息导向/dev/null。其应用范围可说相当广范。

输入重导向（Input Redirection）的运用

符号“<”－－重导向标准输入。



如上图所示，通常一个程序或指令所须输入的参数便称为标准输入（standard input）。在导入的运用上，可用来导入档案。语法如下：

使用语法 command < file

下面的例子，第一是将档案“mail.file”重导入指令 mail 中，作为传送 mail 给使用者 akk 的内容。第二则是将档案“file.data”重导入指令“wc -l”中，用以计算该档案的行数、字数（word）与字符数。

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

虽然以上重导入的使用方法是正确的，但对于一般的指令的运用来说实在是多此一举，因为像这类指令的语法均会支持档案输入的功能。但是重导入的运用却依然不可轻意忽视，有许多软体公司所发展的应用程序或自行开发的 shell 程序（一般称为 shell script），大部份多采用交谈式的方式来要求使用者输入所需的资料，像此类的交谈资料，可预先做成一个档案再以重导入的方式执行，不但省去交谈的程序更可进一步将简化作业程序。

符号“<<”——字符的重导向

使用语法 command << word

```
word or data keyin
....
word
```

在上面我们所提到的重导向符号“<”它所能处理的对像是档案，如果不是档案便无法处理。而在这里将为你介绍的符号“<<”则是用来重导向文字使用的。我们来看它运用在指令行模式下的情况。

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

在上面我们使用了符号“<<” 来将我们想要传 mail 给使用者 akk 的信息一一由键盘键入。指令行最后的“EOF!”代表当我们见要键入内容的结尾用字－－我们可称它为“关键字”。当我们键入指令“mail akk << EOF!”后 return，便可开始键入想要传送的信息内容，如果想要结束仅需在新的一行键入“关键字”便可。

这个符号“<<”常被运用在档案编辑的指令上，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
```

上例我们运用这种重导入的方式来编辑档案“sed1.f”。第二行与第三行是指令 ed 的编辑指令，前者是找寻档案内所有的“root”字符串，并将该行的“ 0 ”代换为“ 1 ”。后者则是找寻档案内所有的

“akira”字符串，并将该行清除。第四行也是编辑指令，用意是将编辑的结果写到另一个档案“sed2.f”中。第五行则是退出指令 ed 编辑器。第六行便是重导入的“关键字”，告诉导入终止。

如果有机会试一试这种用法！使用它来编辑档案，有时候比你进入 vi 编辑器做还快速。

输出重导向（Output Redirection）的运用

在重导出的使用比较起重导入的运用复杂，也比较常使用到。通常指令的输出讯号包含到标准输出（standard output）与标准错误（standard error）二种。关于重导出所使用的符号也比较多，总共有 8 个符号。就功能的区别，大致上可归类成两组。如下表所示：

第一组	“>”	“>&”	“>>”	“>>&”
第二组	“>!”	“>&!”	“>>!”	“>>&!”

符号“>”与“>&”——重导向标准输出与标准错误

使用语法 command > file

command >& file

先让我们来下一个指令 find，从“root”以下找寻档名为“Cshrc”，并将结果列出来。

```
7 % find / -name Cshrc -print
```

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

指令所输出的信息有三行，有两行是指令找到我们指定档名的资料输出，这正是我们所要的。而这些输出信息的格式属于标准输出。但却有一行信息显示“/var/spool/mqueue”目录不允许我们进入，换句话说该目录我们没有读取的权限。这行信息会混在两行标准输出的信息中是因为指令 find 搜寻目录的次序关系，并无特殊意义。但信息本身的格式却不属于标准输出，而是标准错误。这一点我们可用以下的指令来获得证实：

```
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
```

当我们再一次执行同样的 find 指令，但使用重导向符号“>”，将输出导向到档案“file.tmp”中。结果发现只剩下警告信息依然出现在荧幕上。用指令 cat 去看档案“file.tmp”证实需要的资料确实在档案中。这证实了警告信息是不同于一般的输出信息的。因为它的模式是标准错误。其实就整体而言，在 C Shell 的环境中警告性的、错误

性的输出信息均属于此类——标准错误。千万别忘了。

如果说你想要将这了两种输出信息一起重导向到档案“file.tmp”中，可改用重导向符号“>&”。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
```

情况便如我们所要的，所有信息都在档案“file.tmp”中了。在这里我们要告诉你一件事，其实在输出重导向这一系列的符号中，“&”符号便是代表标准错误，只要是重导向符号中含有“&”，就表是可重导标准错误。如“>&”“>>&”“>&!”“>>&!”都是。

对于重导向符号“>”与“>&”在运用上有一个特性，那就是当你所指令的档案实际上不存在时，重导向功能会帮你产生该档案，并将资料写入。但如果该档案在执行前便以存在，则当指令执行重导向功能完后，该档案原来的资料将会被重导入的资料所重写。换句话说，便是原来的资料将会完全不见，且无法挽回。使用上请读者务必注意到此特点。我们来看下面的实际例子：

```
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:::

我们可清楚地看出，当指令“tail -1 /etc/passwd > file.tmp”执行完后，档案“file.tmp”的内容已变成是“/etc/passwd”档案的最后一行。而原来的资料以不见了。

在重导向的运用上有一项限制，请先看到下面的例子：

% cat acct.data > acct.data

cat: input acct.data is output

当指令 cat 所产生的标准输出的正是档案“acct.data”的内容，而我们又要将这个输出重导向到档案“acct.data”中。于是造成指令 cat 的执行错误。在这里我们要告诉你在运用重导向时，不能有输出档名与重导向的档名相同的情况产生。

符号“>>”与“>>&”

使用语法 command >> file

command >>& file

这两个重导向的符号比起先前我们所介绍的符号“>”与“>&”，在功能上有一个很大的差别。就是符号“>>”与“>>&”在重导向输出字符串到文件时，不会重写原文件内容。它们只会将输出资料重导向到原档案内容的后面。

有了这两个符号，我们便可将多个指令的输出一起重导到同一个档案中，而不必再担心内容会被重写了。我们来看个例子：

3 % who >> data.tmp

4 % ps axu >> data.tmp

5 % df >> data.tmp

第一个执行的指令“who >> data.tmp”如果档案“data.tmp”不存在，重导向符号“>>”与符号“>”相同，会自动产生该档案，并将输出重导向到档案中。如果“data.tmp”档案已存在，则指令输出便直接在写到原有的内容后面。而第二、三个指令的输出，一样地会依序串写到该档案中。如果你觉得三行指令太麻烦了，改成以下的执行方式有也行：

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

（如果连标准错误资料也要写入档案则必须使用符号“>>&”）

设定$noclobber 预设变数改变输出重导向特性

变数设定语法 set noclobber

取消变数设定语法 unset noclobber

以上对输出重导向所使用的符号已介绍了四个（第一组），但其实还有四个尚未介绍（第二组），它们的功能与上述的符号有点对映关系。如下：

第一组	“>”	“>&”	“>>”	“>>&”
第二组	“>!”	“>&!”	“>>!”	“>>&!”

以下的四个符号在未设定$noclobber 变数时功能与上四个完全相同。但如果你设定$noclobber 变数，使用符号“>”与“>&”将无法

重写（overwrite）已存在的档案，而必需使用符号“>!”与“>&!”才行。同时使用符号“>>”与“>>&”亦不能将指令输出导向不存在的档案中，得使用“>>!”与“>>&!”。

在下面的例子当中，我们先设定了$noclobber 变数，然后指令 ls 的结果告诉我们该目录下并无任何档案存在。接着我们操作几个运用重导向的指令看看它们的结果。

```
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
```

当指令 3 执行失败的原因是“passwd.bak”档案不存在。这如果在没有设定预设变数 noclobber 的情况下，它是不会产生错误的。接着指令 4 将重导符号由“>>”改为“>>!”便顺利地执行成功了。这时以指令 ls -l 便可清楚地看到该档案的各项资料。

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

在这种环境下如果我们键入了以下的指令：

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

则因为该档名已经存在，不允许我们用符号“>”来将该档的内容重写。这便是设定预设变数 noclobber 所产生的保护作用。如果想重写该档案则必须使用符号“>!”才行，如下所示：

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

以上所模拟的情况仅只是设定前与设定后的最大不同处。至余其它的情况请自行在电脑上依序演练，注意到与原先未设定变数前的种种差别，必可加深在使用上的经验。此项功能将对于撰写 shell 程序有相当的帮助。如果你想取消该变数的设定只要在指令行键入“unset noclobber”便可以了。在上面的各个重导向的情况中，如果须连标准错误信息也一起重导的话，只要重导向符号代有“&”便可。如“>&!”及“>>&!”。千万注意！位置不能放错。

实用的输入/输出重导向运用

标准输出与标准错误的分离重导向

就如我们所知，一个指令的输出信息包含了标准输出与标准错误两种型态。有时我们会有需要将某指令的输出信息储存建档管理，除了正确的输出资料外，那些错误讯往往也是相当重要的，为了将有两种信息分开来存放，重导向的使用手法就显得格外的重要，针对此种情况常用的重导向方法是将指令与重导向标准输出用括号括起来，如此一来此部份会先执行，并且指令的标准输出信息也会先储存到指定的档案中，输出信息剩下的部份就是标准错误信息，我们再用重导符号“>&”重导到指定的错误信息档案便大功告成了。使用语法如下：

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

让我们来看一个实际的运用：

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

上例中我们用指令 find 去搜寻目录/user1 之下所有超过 50000 字符的档案，并将结果以档名 big.file 储存。但指令执行的错误信息则存到 err.file 档案中。如此便将指令 find 的标准输出与标准错误信息分开来存放，让我们更容易整理。

输出重导向与/dev/null－－UNIX 系统的垃箕筒

真正在实际的运用上，有时候执行一个指令并不是要它的输出结

果，而是要它的指令执行状态讯号。（这种使用手法在撰写 Shell 程序时常使用的上）。这时输出的讯号反而便成一不需要的，这时有没有可能将指令的输出，让它“消失”呢？有的！在 UNIX 系统中的早就为你准备了一个垃圾桶，专门供你处理不必要输出信息。你只要将输出重导到“/dev/null”这个垃圾桶来，输出讯号自然便消失，而且也不会占用硬盘或其它资源。来看一个运用指令 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
```

在上面我们将指令所输出的系统处理程序的资料，用管线引导到指令 grep 中去筛选“cron”，并去除 grep 指令本身的处理程序。但这并非我们主要的所须的，真正所要的是指令 grep 的执行状态，如上用指令“echo $status”所显示的输出“0”，便是指令 grep 的执行状态。 “0”表示指令 grep 有找到指定的字符串，如果是“1”则表示没有找到所指定的字符串。我们可看见第一行指令是会有讯号输出的，让我们将它改一改。如下：

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

指令的输出讯号不再出现在萤幕，且指令的执行情况正常。这便是“/dev/null”的妙用。当然你也可以用它来重导向根本不需要的要输出讯号。如下：

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

像如此便能将看也看不来的指令输出去掉，只保留标准错误信息可输出到萤幕上。如此一来，指令执行时是否有错误信息更可一目了然。

输出、输入重导向的混用

输出、输入重导向是允许混合起来运用的，如前面提到输入重导向的运用上，将程序或指令所需的资料用重导入，再将执行结果重导到档案中。这便是常见的使用方式。见以下使用语法：

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

我们来看一个例子：比方有一份报表要将大写字母转换成小写字母。

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

在上例，我们将档案“report.org”倒导入指令 tr 去做大写字母转换成小写字母的动作，然后将 tr 指令的结果重导向到档案“report.low”中储存。而原始的档案“report.org”则毫无改变地保

留。档案“report.low”便是我们想要的结果。

另外也有一种使用情况，让我们来看下面的例子：

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

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

假定你有一串数字要建档并且要你排序排好，相信以上的使用方法你一定会喜欢得不得了！

重导向符号说明

符号	说明
>	将 stdout 重导向到文件（command > file）
>>	将 stdout 字符串加到文件结尾之后（command >> file）
>&	stdout 及 stderr 重导向到文件（command >& file）
>>&	将 steout 及 stderr 字符串加到文件内容之后（command >>& file）
>!	将 stdout 重导向到文件，有设定 $noclobber 时，可重写文件。
>>!	将 stdout 字符串加到文件内容之后，有设定 $noclobber 时，可重写文件。
>&!	stdout 及 stderr 重导向到文件，重导向到档案，有设定$noclobber 时，可重写文件。
>>&!	将 steout 及 stderr 字符串加到文件内容之后，有设定$noclobber 时，可重写文件。

文件名扩展（filename Expansion）运用

在 UNIX 系统中有时难免会用上长长的路径，长长的档案名称或一大堆的档名在指令中，所以在 C Shell 中提供了几个符号来帮助我们解决这使用上的困扰。这就是档名扩展的功能，也有人称它为 wildcard。相信你对“*”这个符号一定不会陌生才对。你一定常运用它来代表所有相关的档案名称，它便是包含在这个档名扩展的功能之一。以下让我们通盘地看一看整个的档名扩展功能。

符号“＊”与“？”

一般使用者使用符号“＊”的比例可能远多过使用符号“？”。这可能是因为符号“＊”代表任何的字符（包含 null）。而符号“？”仅用以代表任何一个单一的字符。这差别是相当大的。让我们来看一个例子：

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

第一个指令 ls 告诉我们共有五个档案，当我们以“a*”来显示相关档案时，五个档案全出现了，其中档案 a 的出现证明了符号“＊”连 null 都代表的特性。而档案 aaa 的出现告诉了我们一个符号“＊”

便可代表多个字符。接下来我们把符号“*”换成符号“？”来看看结果的差异性。

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

你可发现档案 a 没出现，因为符号“？”不代表 null；档案 aaa 也没出现，这是因为符号“？”仅代表一个字符。如果你必需用 a?? 才行，如下：

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

看清楚仅有档案 aaa 出现而已，a1、a2、aa 等只有两个字符不符合条件。所以说使用者为了在使用上的方便，会运用符号“*”多过符号“？”大概就是这个原因吧！但有一点必须说明当你在撰写 shell 程序时，如果需要使用类似的功能时，别忘了符号“？”是比较严谨的安全的作法，不妨改便你的使用习惯看看。因为撰写 shell 程序要应付的情况较多，严谨一点的作法比较不会产生 bug。

对于“*”这个符号，真可说是懒人之“星”。像指令 cd 到某目录时，目录名称打到一半接着“*”代表之。 ls、vi、lpr、cp、rm 等等的指令也均如法炮制。

在符号“*”的运用上也不尽然是所有指令都能接受的，像指令 mv、cp 便会有语法上无法接受“*”代表档案的情况，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！太懒了！ ！被电脑“教训”了一顿！ ！ ！不过说实在的作者我一直认为懒惰是人类发明创造的原动力！一切的伟大发明皆来自于伟大的懒惰。指令不能满足我们的懒惰，则我们自己便应该为自己创造出一个属于自己的“懒惰功能”来。在 Shell 程序设计单元中你将可为自己的懒惰感到无比的光荣。

符号“[...]”

这个符号的功能在为你定义字符、数字的范围；它可以代表一个范围内的一群数字或者是字母，也可代表很多个独立的字符，或者是数字。在使用上相当具有弹性。要使用该符号“[...]”代表一个范围时，范围必须由小到大，两者之间以符号“-”连接。

[0-9] 代表 0、1、2、3、4、5、6、7、8、9 等数字

[a-z] 代表字母 a、b、c、d、...、x、y、z

[A-Za-z]代表大写字母 A-Z 或小写字母 a-z，既大小写字母全部。

[A-z] 同上。

[9-4] 范围错误（错误示范）

[z-a] 范围错误（错误示范）

要代表多个单独的字符，只要一个个字符加入括号中便可。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，范围只能由小到大，不可由大到小。如果用由大到小的方式虽然不会产生错误信息，但结果一定是错误的。这一点自己可上机试试！以下我们来看一些实际运用的例子：

想一次显示两种尾号不同的所有档案时，如下例所示：

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

想将某个范围的资料档案汇整成一个新档案，比方如上面这四个档案合起来建一个新档案。用法如下：

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

又譬如要将 aa1、aa2、aa3 三个目录下的档案拷贝到另一个目录 all 之下，也可以使用得上。如下例：

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

但是这个符号如果运用到 mkdir 指令时会产生错误的，这点请注意。比方你想一次新建五个目录 disk1、disk2、disk3、disk4、disk5。假定你使用以下的方式：

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

错误信息告诉你“No match.”，错在那里呢？事实上，在使用的语法来说并没有错。只不过是 mkdir 指令无法接受这种用法，所以造成指令无法执行成功，产生了错乱的信息输出。虽是如此，但这个法则的可组合运用还是相当好用的。请爱用善用之。

符号“{,,,}”

在前一个符号的最后指令 mkdir 的例子中，该符号做不到的，这个符号“{,,,}”可帮你做到。语法如下：

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

就这样一次新建五个目录 disk1、disk2、disk3、disk4、disk5。或者是像以下的运用：

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

一道 mkdir 指令一次建立 16 个目录。简单好用的很。

有时它的功能会让人觉得蛮接进符号“[...]”的，但它们所适合处理的情况其实并不太相同。符号“[...]”所适合处理的是比较简单化的字符，较复杂无规则性的字符则使用符号“{,,,}”比较恰当。假如我们要将三个存放在/usr/include 目录下的档案一起拷贝到~/time 目录下，档案名称分别为 time.h、stdlib.h、termio.h，这如果要运用前面符号“[...]”来“简化”指令，根本是不可能的事。但是，我们可以运用符号“{,,,}”来处理，请看下例：

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

这就是符号“{,,,}”所适用的强处。不过符号“{,,,}”对于“范围”的处理能力并不能像符号“[...]”那样强，可以使用到符号“-”来做到像这样“[0-9]”简洁地代表 0 到 9 的“范围”功能，这点倒叫人在使用上感到有些遗憾。以下让我们来看看几个运用符号“{,,,}”来精简化指令的实际例子：

```
% 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}
```

符号“~”

符号“~”代表的是使用者自己的 home 目录。它有一个比较特殊的用法，就是在符号“~”之后如果接上另一个使用者名称，则便代表该使用者的 home 目录。这点在路径名称的运用上帮助相当大。比方你要到另一个使用者 yeats 的 home 目录下，你可以毫不考虑目前的工作目录，及你是否知到该使用者的 home 目录，马上使用这个用法在 cd 指令上，便可改变工作目录到该使用者的 home 目录之下。如下所示：

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

像此类的运用最适合同一个工作团体，使用者与使用者之间的档案传输，如下示：

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

以上这两个指令做的是同一件事情，你会喜欢下那一个呢？尤其当你会搞不清楚那一个使用者的目录在那里时，你将会格外的喜欢上它。

管线（pipeline）观念与运用

将一个指令的标准输出重导向，作为下一个指令的标准输入，像这种连结两个指令间标准输出、输入的特殊功能，便称为管线（pipeline）。在 C shell 中定义了两个关于管线功能的特殊符号。如下：

“|” 只导向标准输出（stdout）

“|&” 导向标准输出与标准错误（stdout and stderr）

先让我们来看看一个例子：

```
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
```

从上面的例子中，要计算 who 指令的输出有几行，以输出重导向的做法是，先将 who 指令输出重导向到一个档案“file.a”中，再将该档案重导向到指令 wc -l 来计算行数。最后再将产生的资料档案“file.a”清除。这过程中有三个硬磁的 I/O 动作。但如果使用管线的方法如行 4 所示，它根本没有硬磁的 I/O 动作，是不是比行 3 所下的指令方便多了呢。而且不需产生任何档案，省略的 I/O 动作，对使用效率来说是比较好的使用方式。

我们再来看看下面这个例子：

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

上面的指令是用指令 find 找寻 home 目录下档名为“*.dat”并将它列出，再将列出的信息运用管线通入指令 rm 去将这些档案清除。

说起来好像是一点问题也没有，其实是一点用也没有。要不然马上上机试一试。这个指令的错不在于指令 find 或运用了管线，而是在于指令 rm 身上，因为它的引数不接受标准输入。所以你无法这样使用。不过指令 find 本身的语法有支持像上面的运用，如下示：

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

当指令 find 在找到一个相符的档名时会执行指令 rm -i，让你键入 y 或 n 来决定是否要清除该档案，之后，指令 find 会再继续寻找相符的档名来做同样的动作。这样的功能便与上面那个使用管线不成的指令作用相同了。

所以在管线的运用时一定得注意前后指令的关系，及是否指令接受标准输入等问题。你越用心去尝试将可得到越方便的越佳的使用效率。看下面这个运用：

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

从一个系统信息经过数个管线的连接，将这个系统信息筛选处理成我们所要的资讯，并直接将它由列表机印出。或者是像下面的运用：

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

像这类的运用都是相当得宜的作法。

相关的辅助指令－ tee 指令的功能

对管线的功能而言，并无法像输出重导向可产生档案。这是管线功能的不足与限制之处。但不要紧，UNIX 有一指令可填补这项缺撼，这便是 tee 指令。当行 4 指令修改为“who | tee file.a | wc -l”，则指令 who 的输出一方面由 tee 指令储存到档案“file.a”中，一方面又 pipe 到指令 wc -l 去计算 who 输出的行数。这个指令你可用在备份档案的指令，特别是当你想一面在萤幕上查看档案备份的情况，而又希望将备份档案的信息储存成档案时。如下例：

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

另外该指令也提供了一个-a 的选项，来让你能将信息加到指定档案的内容之后。请看以下的运用。

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

在指令 2 中，我们再一次使用 tee 指令将指令 date 的输出加到档案“backup.data”中，但因为指令 tee 有加上选项-a，所以并不会将原档案的内容清除掉。怎么样是不是很好用呢？

history 的设定与运用

history 说起来可算是 C Shell 的重大功能之一。history 这个字如果以它的整体功能来说，或许我们可称它为 UNIX 的指令使用记录。它负责记录使用过的指令，供用户“再利用”。这个“再利用”的动作可说是灵活多变，可运用的方面除了指令行外，连别名（aliases）、预设变数（Predefined Variables）的设定或 shell script 的程序设计等均可见到 history 的运用踪迹。实在是一项使用者不能忽略的好功能。

制定 history 的使用环境

要能善用 history，首先得熟悉关于 history 使用环境的三个重要的预设变数。以下让我们来为你一一地介绍：

$history 变量

变数 history 是使用 history 功能之前必须先设定的。通常都在 C Shell 的启始档案“.cshrc”中设定此变数，当然也可用手动的方式来设定或更改。设定的语法如下：

set history = n （n 是数字且必须是整数）

如果你把 n 设定为 30，则 C Shell 将会随时为你保留最后所下的 30 道指令，供你呼叫显示到荧幕上，这便是 history list，它可供你运用 history 的其它功能。如果你想查看此变数的设定值，可使用指令

```
echo $history
```

来显示 history 设定值。要重新设定可修改“.cshrc”档案内的设定，再以指令 source .cshrc 便能更新设定值。也可直接下指令 set history = n 来更改。但后者设定的设定值会随着该 Shell 的终结而消失，此点请注意。

$savehist 变量

设定这个变数的作用是在 Shell 终结后，将最后某几道指令储存到用户的 home 目录下的“.history”特殊档案中，以供你下次 login 或另一个还在工作的 shell 来运用，上一个 shell 所记录下来的最后几道指令。如你设定为下：

```
set savehist = 50
```

则每一个 shell 的终结，都会对“~/.history”档案做资料写入、更新的动作。请注意!再重复一遍，是每一个 shell 的终结都会更新 home 目录下的“.history”档案。所以当你下次 login 时所见到的 history list，便是“.history”档案的内容。

如果你常在 X Windows 或者是 Open Windows 等 GUI Windows 界面下工作的话，相信你一定会开好几个 Windows 同时工作。在这种工作环境下如果你不去加以控制要储存那一个 shell 的话，建议你不要使用它，或许会比较好些。

$histchars 变数

这个变数便是用来改变设定 history 的运用符号。用来解决符号的使用习惯问题。 history 的专属符号有两个，第一个使用的符号“!”第二个符号“^”。如果你如下设定：

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

则符号“#”取代符号“!”；而符号“/”取代符号“^”。老实说来 UNIX 系统对特殊符号的运用可说到了一“符”多“栖”的田地了，小心改出毛病来。建议你谨慎使用（最好是不用）。

history 的运用说明

设定 history 的数量与显示的关系

在一般使用的情况下，history 的设定数量大约都在 20 ~ 50 之间。这是因为设定太大了会占用系统资源，用不上的话实在不划算。适中够用就是好的设定值。你也可就屏幕显示的列数来做为设定值的参考。免得列出 history 时超过屏幕所能容纳的行数。要在屏幕上列出 history 相当简单，键入 history 便可。如果你嫌“history”字太长不好记，设个别名“alias h history”，相信也不会有人骂你懒！

在指令行模式下键入 history 可显示出 history list。当然你也可以在指令 history 后加上一个数字，用来控制显示 history list 的数量。

如下：

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

在 BSD 版本的 UNIX 系统中，C Shell 会提供 history 一项比较特殊的选项，就是“-r”选项。它的作用是将 hisroty list 显示的事件次序倒过来。如下所示：

```
26 % history -r 5
26 history -r 5
25 history 5
24 cl
23 ls
22 cd subdir1
27 %
```

较长的设定值在显示上会有超出萤幕行数的情况产生，这个问题很好解决，提供几种别名设定供你参考使用：

alias hup 'history | head -15' （前 15 道 history）

alias hdn 'history | tail -15' （后 15 道 history）

alias hm 'history | more' （一页一页看）

history 对过去指令的处理方式说明

C Shell 的 history 功能，对于用户所执行过的每个指令，基本上都把它当成一个“事件（event）”来处理。而每个事件都以数字来编号代表之。让我们来在设定预设变数$prompt 中加入 history 功能来说明，这种事件的处理情况。在指令行中键入如下：

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

这样你便可在提词（prompt）中清楚地看到 history 对“事件”的编号情况。如以下实际例子：

```
% 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
```

从以上实例中看到 history 对过去指令的编号情况。通常 history 对指令的编号由 1 开始编起，每次自动加 1。但如果你设定预设变数 $savehist，则由此变数的数字加 1 算起。此点请注意。

到此相信你对 history 的运作应该有一个概略的了解了吧！接下来便开始来运用 history 的功能，来“便利”我们的指令操作。如果你能善用将可大大改变你的使用效率。它至少可以降低你因为打多了键盘而得到职业病的机率。

history 的符号说明与基本运用

我们已经知道 history 对指令是以事件来处理，且加以编号。要如何简单地“再利用”这些过去的“事件”呢？ history 定义了一群符号来供我们变化使用。这些符号都是组合式的，基础的第一个符号是“!”。此符号用来启动 history 功能，但紧接在符号后的不能是下列这些符号：TAB 键、空白键、换行、等号或“(”及“.”等。接上这些符号将会产生错误。

history 在指令行运作下的常用到的符号运用组合，我们以功能分类来加以说明：

1. 指定事件执行

符号“!!”

“!!”执行上一个指令。在运用上除了执行上个指令外，并可在符号后在加入与指令语法不相冲的字符来修饰上个指令。如下例：

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

以上是指令 ls 的执行情况，当你觉得所显示的信息不足，想要加上选项-l 时，你可在运用 history 时再加上该选项，如下用“!! -l”便相当于下指令“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
```

符号“!n”

“!n”执行第 n 个事件。注意到这第 n 个事件一定得还在 history list 内才能顺利执行。

假设我们的 history 变数设定值为 10，当我们执行到第 22 个事件时，想要用符号“!n”来执行第 5 个事件，会产生什么情况呢？请看下面：

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

“5: Event not found.”这个错误信息便明显的告诉了我们，指令所要执行的 history 事件以不在 history list 的范围之内。如果你常会有这种情况发生，可以考虑将 history 变数的设定值加大到适当的数字。另外请注意到在事件 22 执行指令“!5”时，因产生“5: Event not found.”的错误，history 对此错误的指令并不记录下来，所以事件的编号停留“22”。如果你所指定的事件在 history list 的范围内的话，指令便能顺利的执行过去的事件。如下：

22 % !17

ls

total 4

2 cshrc* 1 file 1 passwd

符号“!-n”

“!-n” 执行在 history list 中倒数第 n 个事件。

我们用下面的例子说明它的使用情况：

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 %

看到没有事件 25 的指令“!-4”所执行的是 history list 的事件 21，刚好是倒数第四个。

2. 搜寻指定执行

“!string” 搜寻以某字符串 string 为开头的过去指令并加以执行。

“!?string?” 搜寻过去指令行中有某字符串?string? 并加以执行。

以上两种使用方式均是在 history list 的 event 中，搜寻 event 的开头或 event 中间有指定的字符串，来加以执行。但均是执行第一个符合条件的 event。搜寻时以由 event 数字大到小。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

在上例中最后指令“!v”所执行的是第 33 个“vi passwd”，而不会是第 27 个“vi ~/.cshrc”。如果要执行第 27 个 event，你可用“!?cshrc”来执行，当然你也可直接用“!27”来执行，但会使用搜寻方式，自然是在不记的第几个 event 的情况下才会采用的，不是吗？又如果你设定的$history 为 100 个，要用眼睛在 history list 中找寻，倒不如“奴役”电脑还来得好。是吗？

接下来介绍一个比较特殊的符号“!{...}”，用来应付一种比较特殊的情况。譬如你想在重执行以下这个指令：

% cat /etc/hosts > ~/hosts

并且要紧接着在指令后加入一些字符串。如将重导向的“~/hosts”这个档案名称改为“~/hosts.bak”， 如果你下指令如下：

% !cat.bak

cat.bak: Event not found.

这种错误是因为搜寻的字符串与添加的字符串没有区隔开来的结

果。解决的方法是使用大括号“{}”将搜寻的字符串括起来，再仅接着添加的字符串便可。如下所示：

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

3. 修改执行过去指令

下错指令、打错字的情况是难免会发生的，或大同小异的指令也偶尔会连续使用，像此类的情况运用修改的方式相当适当。（由其指令又臭又长的时候，你使用时一定会感激造令者的伟大）先来看一个例子：

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

如果是像这种指令打错了一个字，比方说指令 screendump 少打了一个字母 e，变成 screndump，天呀！主管在旁边看耶！！重打一次？如果再打错保证你年终奖金少一半！！！这可怎么办？？不用急，露一手什么叫“知错能改”，如下：

语法^old^new^ 修改上个指令的 old 为 new 后执行。

实际指令 % ^re^ree^

就这样把你漏打的字母补了进去。简单几个字就搞定，方便吧！不过这种方式的修改是有限制的。它只能还是以上面那个长的不得了

的指令中更改一个地方，而且还得是上一个“事件”才行。如果不是上一道指令，要做类似的修改运用，可用以下的语法均可做到：

!n:s/old/new/ 修改第 n 个事件的 old 为 new 后执行。

!string:s/old/new/ 搜寻以 string 为开头的事件并修改 old 为 new 后执行之。

到目前所介绍的三种方式，均有个共通性，就是只能更改一组字符串。而且字符串内不可有空白键。如果有空白键会造成修改错误。如果要同时修改两个以上相同的字符串，对后面两种语法中再加一个“g”便可，如下：

!n:gs/old/new/ 修改第 n 个事件中所有的 old 为 new 后执行。

!string:gs/old/new/ 搜寻以 string 为开头的事件并修改所有的 old 为 new 后执行之。

用这种方法便可轻易地将那个臭长指令中的 20，一起更改为想要的数字了。如下示：

% !rsh:gs/20/50/

这时我们所得到的将相当于下：

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

自己找例子试一试吧！它们真的是太好用了。

history 对于事件的引数（argument）运用

1. 引数的区分方式

在前面我们曾经提到 history 对指令是以编号的事件来处理，但你除了可再利用这些事件外，同时你也可再利用事件中的每一个引数。先让我们来看一个底下这个例子：

事件的引数由 0 开始编起，所以 0 便代表事件开头的指令。接下来是以空白键来隔开的字符串或以特殊符号依序编号。注意到引数的编号，并非全以空白键隔开才算，因为只要有特殊符号，不管它与前后有无空白键都得算是一个单独的引数。 （这些特殊符号有 |,<,<<,>,>>,& 等）history 对引数安排了多种方式来代表引数的位置，如数字（0，1，2，...）或符号（“^”“$”“*”“x*”“x-”）。对引数的运用可单一的或者是一个范围（xy）均可。

以下我们以表列来说明引数的符号使用方式：

符号	说　　明
!!:n	使用上个事件的第 n 个自变量，如“!!:3”
!!:x-y	使用上个事件的第 x 个自变量到第 y 个自变量，如“!!:0-4”

!n:^	使用第 n 个事件的第 1 个自变量，符号“^”代表第一个自变量
!^	使用上个事件的第 1 个自变量
!$	使用上个事件的最后一个自变量，符号“$”代表第最后个自变量
!*	使用上个事件的第 1 个自变量到最后一个自变量
!!:x*	使用上个事件的第 x 个自变量到最后一个自变量，（相当于“!!:x-$”）如“!!:2*”即代表使用第 2 个之后的所有自变量
!!:x-	与上相似不同处为不包含最后的自变量，如“!!:2-”即代表使用第 2 个到最后的倒数第一个自变量。（语法“!!:x-”就是“!!:x-$”去掉“$”后所剩下来的样子）

2. 引数在指令行的运用

在指令行模式下有些时候会连续好几个指令都用到同一个或一群档名，也就事说，你可能会有一个字符串要一再重复键入。比方有以下情况：

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

像以上这接连的三个指令都用到相同的档案名称 ch1.doc，ch2.doc， ch3.doc，利用 C Shell 的 history 功能，我们可在指令中将过去事件的引数利用符号代入，以简化指令的长度，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 到最后的引数，所以指令 5 相当于指令 2。同理指令 6 相当于指令 3。

再举一个例子：

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

指令 8 中的“!$”既代表指令 7 中的最后的引数，就是“*.txt”。指令 10 的使用在语法上是成立的。连用 history 功能来组成指令，第一组“!6:0-2”是使用事件 6 的第 0 到第 2 个引数，接着空白键，接着第二组“!!$”使用前一个事件的最后一个引数。所以此指令相当于下：

10 % tar cvf /dev/rst8 doc.file

这便是 history 的引数的组合运用。对了！相信你还记得下面这个又臭又长的指令吧！

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

如果现在要你运用 history 的修改功能来更改第二个“50”变为“60”的话，你该如何下指令？回忆一下过去所介绍过的“修改功能”中，只能更改第一个或全部都改，这只改第二个可怎么改才好呢？相信上面的指令 10 会代给你灵感的。想到了没，答案如下：

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

我们用该事件的引数 0 到 7，而引数 8 便是要修改的对像，我们不引用。在“!rsh:0-7”之后，我们便加上“□60□”，再接上该事件的引数 9 到最后的引数。如此便能成功地将某个事件中的某个引数，代换成所需（“□”代表空白键）。运用这样较麻烦的方法可补原先与修改功能的不足与缺憾。希望有一天这种方法能帮你解决一些困扰。

在 history 功能中有一项是仅将指令或所须的功能列出，而不执行。使用的符号为“:p”。这种列出不执行的方式，对比较复杂或比较没有把握的组合运用是有帮助的。我们来修改第 10 个的使用语法

成为只显示而不执行，来看看它的用法：

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

如上指令 10 所得的结果显示在下行，但并不执行。如果错误则可加以更正，如果正确无误想要执行则再键入“!!”便可。

3. 引数的特殊符号运用

在 history 的引数运用上有一项是比较特殊的，它是关于路径（path）的运用。一般也并不常运用在指令行模式之下，倒是在 shell 的程序设计上相当有用。不过我们在介绍时，还是将它运用在指令行模式之下，读者可借此了解它的功能。

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

由上例中我们可藉由 history 的功能来将一个路径分离成两个部份，如“/usr/adm/messages”可分离成，“/usr/adm”与“messages”。如果以前面这个情况来说，一个绝对路径（full path）可将它分离成档案所在的路径及档案名称这两项资料吧！这个运用情况，我们将在 shell 程序设计中再为你举例说明。

关于 history list 的说明

history list 实际上便是你启动 History 功能之后所产生的一个指令暂存器。这个指令暂存器会记录的你下过的指令，也可称之为事件（even）。储存事件的数量便是你使用内建指令「set history = xx」所指定的数量。 history list 的内容允许被呼叫出来加以再次使用或者是修改使用，被呼叫出来的事件可以是整个事件也可以是事件的部份引数，在使用上有相当多的方式及符号，当然组合的变化也相当多。同时这个 history list 的内容也可以在你 logout 前储存成特殊档案供你下一次 login 时再利用。

如何传递 hisrory list 到另一个 C shell 中

在前面我们曾提到变数 savehist，它的功能是可以在你结束 C shell 之后，把该 shell 的 history list 储存起来供你下一次 login 时，或者是另一个 C shell 来使用，但如果你并不想终结现在正在操作的 C

shell，却需要将 history list 储存起来，提供另一个 C shell 运用时。此时变数 savehist 的功能就不能适用了。我们得寻求变通的方式来达到这个目的。首先，将所需的 history list 数量重导向到一个档案中储存，如下所示：

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

接着便是使用编辑器清除档案内不必要的指令。然后退出 vi。这时另一个 C shell 便可以看见这个档案“history.15”。这时我们便可使用内建指令 source -h 来将该读取“history.15”档案内的各行指令，将它们加入到这个 C shell 的 history list 中，如此便可以供我们来运用了。请见实际的指令操作：

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

（请注意，加上选项-h 的作用在于只读入指令，但并不执行。关于详细的情况请参考内建指令 source 说明）

别名（aliases）的设定与运用

设定语法 alias name 'command'

解除设定语法 unalias name

显示设定语法 alias （显示所有别名设定）

alias name （显示 name 的别名设定）

别名（aliases）是 C Shell 的内建指令。主要的功能是设定一个“小名”来取代常用的或复杂的指令组合。在应用上可说是最多样、方便的好功能（此 aliases 功能在 Bourne Shell 中不支持）。在下例中便是别名的设定、显示、使用情况：

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

当我们在使用常用的指令时，总不免会固定地使用指令的某一些选项。所以有人便使用设定 aliases 的方式来代替之。这是 aliases 最常见的应用。如上例我们设定一个叫“ls”的 aliases，它所代表的是指令“ls -aF”。在设定完之后我们所执行的 ls 指令其实已经不是原来系统的指令 ls，而是“ls -aF”。这全与 C Shell 对指令执行的解译流程

问题有关。因为 C Shell 在替使用者解译键入的指令时，如果是属于 C Shell 的内建指令，则会直接执行。如果指令是以符号“/”为开头，则将会被认定为是路径名称，并会到该路径下去搜寻该指令并加以执行。如果开头没有“/”符号，则会有两种情况，依变数 path 里所设定的目录逐序加以搜寻及执行，这是一般的情况。但如果该指令名称在内建指令 alias 所建立的资料内被发现，则 C Shell 会先将该指令名称转换成内建指令 alias 所建立的对映资料，再依此资料内容到变数 path 里所设定的目录群中去搜寻与执行。这便是设定 aliases 后会取代原指令执行的原因。

这种 aliases 名称与系统指令名称相同的设定，虽然带给我们在指令使用上的方便，但也会造成一些在使用指令上的负面影响，使用者不可不小心谨慎。譬如在“ls”已被设定为“ls -aF”的情况下，我们键入“ls -l”，则真正执行的其实是“ls -aF -l”。而不是原指令 ls 的“ls -l”。这一点请读者务必认识清楚。

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

```
5 % unalias ls
```

如果说你在不想解除 ls 的 aliases 设定情况之下，想要运用原指令 ls。有两种方式，第一：指令要以绝对路径的方式来执行，以指令 ls 为例，须键入/bin/ls 来执行。但这种方法老实说是比较笨的人用

的。第二种方法是运用 C Shell 为我们所提供的特殊符号“\”挡去 alias 指令的设定，达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弹性，做更有效率的运用。

aliases 通常均设定在 C Shell 的启始档案“.cshrc”中；也可签入后在设定，不够设定值会随着 logout 或 shell 的终止而消失。最好的方式还是设定在“.cshrc”档案中，对使用上比较方便。

简单的 aliases 设定如“alias ls ls -asF”或“alias rm rm -i”等可不用符号来括住指令，但如果是指令组合中有运用到连续指令的符号“;”时，则一定得用符号来将指令组合括住，否则设定将出现问题。所以在设定 aliases 最好养成用符号将指令组合括住的好习惯。如下例，第一个 aliases 设定便是错误；而第二个有括住指令组合的，才是正确的。

```
% 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
```

别名设定中运用事件的引数

在别名的设定是允许代入变数或使用引数的。先来看看一般使用者常用来查看系统是否执行某个 process 的指令组合：

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

如果像以上指令设定为下：

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

让我们来实际运用这个 psg 来寻找 cron 这个 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 的在显示执行的指令。（set echo 是 shell 的一个预设变数在后面将会有详细的说明）在例子中可清楚的看到设定 aliases 后所执行的 psg cron 情况，因为 psg=ps axu | grep，所以 psg cron 便相当于 ps axu | grep cron。实际上这个指令组合在 grep 的处理的并不完美，因为会连自己的 process 也显示出来。所以我们将指令组合再加一道指令来去掉这个 bug。如下：

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

用指令 grep -v grep 来去掉不需要的信息。但如此一来 grep 所需 pattern 要如何带入 aliases 的设定呢？这便是如何将下指令的引数带入 aliases 的设定中。关于这种情况你可以运用以下这几个特殊符

号来导入所需的引数，以达到你所想要设定功能。

符　号	说　　明
!*	代表指令行的第一个到最后一个自变量
!^	代表指令行的第一个自变量
!:n	代表指令行的第 n 个自变量

所以关于上一个 aliases 设定，便可使用“!^”或“!:1”来带入第一个引数。 aliases 设定如下所列两种方式均可：

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

如果要同时运用多个引数，我们以常用的 find 指令复杂的运用来做为例子，如下：

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

如果要同时将所有引数全部带入且无法预期引数的数量，符号“!*”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"
```

（这个例子中的 grep "^d" 是指每行开头的第一个字母为 d，才显示。整个指令组合运用来显示指定目录下层子目录。）

别名设定中引用别名

在 aliases 设定的指令组合中允许使用已设定的 aliases。如下说明：

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

第一行设定 a = alias，第二行马上运用第一行所设定的 a = alias 来设定 h = history，第三行中则运用了第一行及第二行的的设定。像此类的设定虽然成立，但实际运用时可得小心。

别名设定的循环错误（alias loop）现象

```
% alias date 'clear;date'
% date
```

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

看看上面这几个 aliases 的设定，虽然在语法上是合理的，设定时也不会产生错误信息，但为何执行时却都会产生错误呢？请仔细想想看！其实原因均是在“指令组合”中运用连续指令时产生的，其现象就是当 aliases 的名称与所设定的 aliases 内容的最后一个指令相同所造成的。请特别注意下例的 ps：

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

展开来执行 ps 便等于 who; ps，而连续指令中的 ps 会因为 aliases 已设定成功的因素，也等于 who ; ps，如此设定后的 ps 在执行时便会产生一个怪异的结果，也就是一直在执行 who ; who ; ...，这就是所谓的 Alias loop.。

要避开 Alias loop. 有几种方式，可将指令组合中的指令改为绝对路径，如下：

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

同理但迅速的方法便是利用符号“\”，如下所示：

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

以上的两个避开 Alias loop. 的方法，如果碰到要使用 C Shell 的内建指令，就无效了。这时您可以尝试将指令组合中的指令，次序上对调如下亦可解除“Alias loop.”的情况。 （虽然看起来也像会造成执行的回圈，但其实不会，请放心使用）

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

在前面我们提过 alias 与指令名称相冲的情况是可允许的，但也是必须要加以特别小心的。因为在 alias 设定中你可能一不小心便造成这种错得不知不觉的情况产生。同时别名与指令名称相冲的情况，最容易影响到设计考虑不周全的 shell script，造成执行错误或增加移植上的困扰。此点不可不慎重其事。

几个实用的别名实例

好的 aliases 设定往往会令人错觉是一个“新指令”的诞生。而 aliases 的设定方式也可说是千奇百怪，花招无穷。每当我想到或看到有创意的 aliases 时，总不免惹人一笑。我们来看看下面所列的几个 aliases 的例子：

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

上面这个 aliase 是将指令 vi 与 chmod 的功能结合在一起，用来

免除我们用 vi 编辑器编写一个新的 C Shell 程序时，还要自己再用指令 chmod 来改便该档案的“执行权限”的使用困扰。这对常撰写 C Shell 程序的用户来说真的相当方便。

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

工作控制（job control）的运用

在一个多任务的（multi-tasking）环境中，允许你同时执行好几项工作。就如使用应用软体、监看系统情况、备份资料、编辑文件、打印文件等等。这些工作你可同时进行，譬如当您从一部个人电脑使用一般的 telnet 工具签入 UNIX 主机时，你只有一个萤幕又不可能有视窗接口（X Window、Motif、Open Window 或 SunView 等等）可使用，你便可以借助 job control 来执行同时他们。这些同时进行的工作就像是排好的卡片一样。一个在前面，其它的都依次序排一个接一个的排在后面。前面的我们称它为前台工作（foreground jobs），后面的我们称为后台工作（foreground jobs）。

前台工作（foreground jobs）

在一般正常的情况下，假设你在键入一道指令，如下：

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

则你必须等待指令执行信息出现、执行完毕，到下一个提语（prompt）出来之后，你才可以再继续你的下一道指令键入。像这样的操作模式，所执行的工作就是属于前台工作。如果指令执行中途想要中断（interrupt），可用 control-c 的方式中断指令的执行。或用 control-z 来停止指令。如果你用中断的方式，当然这个指令或程序的结果有可能是不正确的。像指令 find 的执行的方法是到档案系统中去搜寻你所指定的某种形态的档案，在执行过程中，你只能眼看着它占用这个 shell，除了等它搜寻完毕外，你什么事也别想再做了。还好指令 find 的执行时间通常还不会太久。但如果像 tar、find、cpio、...等等指令或某些应用软体的程序，往往执行下来会花费一断不算短的时间，有时说不定会等上数个小时甚至于数天。这么长的时间你也要等吗？别忘了 UNIX 系统可不是 MS-DOS，它是多任务的。像这样的工作便不应该让它在前台工作中执行，而必需交给后台工作去执行。而前台工作则可继续正常运作，不须做任何等待。

后台工作（background jobs）

要将一个指令放到后台工作去执行，其实非常简单。只要在指令的最后面加上符号“&”便可。当你 rutern 后会将马上显示一个信息，并且下一个提语（prompt）也会马上出现。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个“&”符号，只有第三个才将指令放到后台工作中执行（第一个与第二个它代表的义意是标准错误信息，千万别搞乱了）。在指令执行后所出现的信息，“[1]”则是代表后台工作的工作号码；“293”与“294”则表示该工作在系统中所执行的 PID（process ID）号码。这指令的作用在备份目录“/home”下所有的资料，并将指令 tar 的输出（含标准错误信息）以指令 tee 存放到 tar.tmp 档案，并将输出重导向到“/dev/null”这个无底垃圾箱去，避免输出信息干扰到前景工作的进行。

当指令以放到后台工作中执行，我们就不需等待该指令执行完成，便可马上继续下指令来查询 process 的处理情况。我们所得如下：

```
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
```

由上的结果显示该后台工作的 process 在系统中的处理状态。另外我们可用 C Shell 的内建指令 job 来显示后台工作的状态。如下：

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

指令 jobs 的信息：“[1]”代表后台工作号码，“+”符号代表“current job”，如果是出现符号“-”则代表“previous job”。再来便是后台工作的执行情况“Running”，最后则是后台工作的指令。当指令在后台工作中执行时，当它正常执行完毕后会产生信息，告诉你后台工作已经执行完毕。如下指令 5 后的第二行信息便是。

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

当然后台工作不是止能执行一个而以，你可连续将各种工作用符号“&”将它们放到后台工作内执行。

后台工作的控制管理

当我们将数个工作放到背景工作执行时，我们要如何去管理这些背景工作呢？又如何知道背景工作的执行情况？如何终止或暂时停止背景工作的执行？不要急先让我们来丢几个指令到背景工作去执行，再来逐一说明：

```
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
```

如上我们将 3 个指令放到背景（background）中执行。用指令 jobs -l 显示背景工作的执行情况如下：

```
% 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
```

首先我们为你介绍一个工作控制特有的名辞：current job。再上例中的 current job 是相信读者一定能一眼看出是“du -a /user > user.data”，也就是背景工作号码“[1]”。如果当第一个背景工作顺利执行完毕，第二个与第三个背景工作均还在执行中时，current job 便会自动变成是背景工作号码“[2]”的背景工作。所以 current job 是会变动的。当你下 fg、bg、stop 等指令时，如果不加任何引数则所变动的均是 current job，关于这点在下指令时千万得注意。同时代表 current job 的特殊符号也有是多个如“%”“%+”或“%%”，选一个

习惯的运用便可。

管理背景工作处理程序

终止背景工作

指令总是有下错或下了而又发觉不必要了的时候，当这样的情况，当这样的指令是产生在已执行的背景工作时，你可用 C Shell 的内建指令 kill 来终结它，而且还有很多种方式呢。在例子中，假如你想要“终结”背景工作“du -a /user > user.data”。见下面几种指令 kill 的用法：

内建指令 kill 使用语法：	
kill pid_number	将某个代码的处理程序终止执行
kill %	终止执行 current job
kill %+	同上
kill %-	终止执行 previous job
kill %m	终止执行第 m 个背景工作
kill %string	搜索以 string 为开头的背景工作并将它终止执行（为避免错误，此法少用）

了解上述的情况后，要终止第一个背景工作就很简单了。随便以底下的每一行指令均可将它终止。

```
% kill 237 （237 为第一个背景工作的 PID 号码）
% kill %1
```

如果想终止第二个背景工作，以下两种方法均可。

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

前景、背景工作的停止及再继续执行

有些时候在工作中，手边正做到一半的事，时常会被打断，要求你先处理别的事。譬如你正用 vi 编辑某个档案或用指令 find 在整理档案系统的过时档案，你的主管十万火急地跑过来，马上要你处理一份资料，因为他的主管马上要，人又坐在你旁边，好罢！把正在处理的工作先占停下来，先做他的吧。请想想，你是如何暂时停止你的工作的？在这个时候，我们可使用 control-z 来暂停工作。且我们会得到一个信息，如下：

```
Stopped
20 %
```

暂停的工作已被 C Shell 放入背景工作中了，且保留原状态停止执行。这时便可以马上做你主管插入的高优先工作，假设如下：

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

这时候你虽然把工作目录更改了，但其实并不要紧。你可以先用 C Shell 的内建指令 jobs 来显示刚才被暂停的工作是第几个背景工作，假设你就只有那一个背景工作，则你将见到类似下列的信息：

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

这时你可以不用顾虑工作目录改变的问题，因为 C Shell 的工作控制会帮你处理的，你只管用 C Shell 的内建指令 fg 来把背景工作切换到前景来执行便可。如下：

```
23 % fg （注：指令后不加引数既代表 current job）
vi home.data (wd: ~/test)
```

上面信息“(wd: ~/test)”便是告诉你 vi 的工作目录，请放心它绝对与你当初的工作目录相同。这信息出现的时间很短，马上便回到你原来的 vi 模式中，当你编辑完后，存档后会出现下面的信息：

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

工作目录又回到你所改变的目录去了。自己好好试一试，碰上时可帮你解决不少困扰。

以上为你介绍的是暂停前景工作，接下来我们来谈谈如何暂停背景工作的执行。 C shell 的内建指令 stop 可用来暂停背景工作。语法如下：

stop 指令使用语法：	
stop %	停止第一个背景工作执行
stop %n	停止第 n 个背景工作执行

了解指令 stop 的语法后，你便可将想要停止的背景工作暂时停止执行，譬如你要将一份整年度的月报表用 nroff 指令整理，因为资料量庞大，所以你将它放到背景中执行。指令如下示：

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

当你处理到一半的时候发觉 moonth12.acct 档案是旧的需要更新，这时你用 jobs 指令查看到背景工作“[4]”还在执行中，但无法知道指令以处理到那一个月份时，你可马上下 stop 指令，将背景工作“[4]”先暂时停止执行再说：

```
% stop %4
```

然后马上查看档案 year93.acct 的尾部判断是否以处理到第 12 月份，如果还没有执行到，马上将 moonth12.acct 档案资料更新。再用指令 bg 来将已被暂时停止执行的背景工作“[4]”再接着继续执行下去。这样非但可更正错误档案，又不用将已处理完的工作放弃，重新重头再执行一次。可说是一举数得。这是工作控制中最令人“感激”的一项功德无量的功能。千万不可不知。

指令 stty 设定的影响

在一般的环境下，背景工作的输出均会直接出现在萤幕上，如果想要让背景工作的输出不直接插入到前景工作中，用指令 stty 来设定可解决这项问题。指令 stty 有一项参数叫“tostop”它的作用是将背景工作的输出不直接输出到前景去，只送出信息告诉前景说某背景工作有输出信息。并且它会将那个背景工作先暂时停止执行。当你知道某背景工作有信息输出时，可用指令 fg 来将该背景工作叫回前景观看其输出信息，同时该工作将会继续执行。如果你想再将这个工作送回背景去执行，你可以用 control-z 先暂停，再用 bg 将该工作放入背景中继续执行。这种运用方式仅适用于输出信息少的工作，信息输出量大且频繁的工作可能就不适合。因为当一有信息要输出，该工作便会停止执行，等待你去查看后才会再继续执行，前景、后景这样切换多次会叫人受不了的。让我们来看一下实际的例子：

如果你不知 stty 指令现在的设定，可用键入 stty 来显示设定值，如下：

% stty

speed 38400 baud; evenp

-inpck imaxbel -tabs

iexten crt

显示的参数中可看到并无“tostop”，于是下指令来设定它：

% stty tostop

% stty

speed 38400 baud; evenp

-inpck imaxbel -tabs

iexten crt tostop

在这样的环境设定下，我们将一个指令放入背景中执行，如下：

% find / -name Cshrc -ls &

当背景工作找到档案时，会在你前景出现以下的信息：

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

这时你便可用指令 fg 将该背景工作叫回前景，便可看见输出内容，如下示：

% fg

find / -name Cshrc -ls

24255 3 -rw-r--r-- 1 bin staff 2897 Jul 24 1992 /usr/lib/Cshrc

此时你可以将它 control-z 暂停，再以指令 bg 送回背景中执行。就可在前景继续工作，并等待下一个输出信息出现。

如果你想将“tostop”这个参数去掉，指令语法如下：

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

指令 notify 与预设变数 $notify 的运用

C shell 的工作控制对于背景工作的执行结果通知，通常是在前景中等待你任何一次 return 的时候，顺便将信息输出到萤幕上。譬如当你在前景中用 vi 在编辑档案时，某个背景工作的执行结束了，这项信息会一直等待你编辑完档案退出 vi 模式后，才有机会将会信息输出到萤幕上。这个好处是不影响你编辑档案，但也可能延误你对需最优先工作的处理时效。如果你要改变这项工作控制的执行特性， C Shell 提供了内建指令 notify 与预设变数$notify 来供你运用。先来看指令 notify 的用法。比方说你把一个指令放到背景中执行，如下：

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

当指令放入背景中执行时，如果你想让它在执行结束时马上通知你，你可运用内建指令 notify 来做到这点，如下键入指令：

```
% notify
```

这时指令不加引数是因为该背景工作是 current job，如果不是 current job，你下指令时便得加背景工作号码的引数，如下：

```
% notify %3 （假设背景工作号码为 3）
```

当你下完指令 notify 之后，便去做另一个程序的编辑工作，当该背景工作执行完毕时，如果你还未离开 vi 模式，输出信息便会插入萤幕中，插入的信息仅是通知你背景工作的情况，并不会加入你的 vi buffer 中，这点请放一百个心。信息如下：

[1] Done cc -o backup backup.c

以上是使用指令的方式，如果使用模式比较适合你的使用习惯的话，建议你不妨改用设定预设变数$notify 的方式，将下行加入“~/.cshrc”档案中，则你每个背景工作均会以此方式通知你其执行情况。

set notify

关于背景工作使用的注意事项

资源使用限制（limit 指令）

如果在你的使用环境中，有运用 C Shell 的内建指令 limit 来限制你的系统资源的使用时，对一项执行时间会较长的背景工作而言可能会比较不利。如果你想查询资源使用限制可使用 C Shell 内建指令 limit，如下：

% limit

cputime 10:00

filesize unlimited

datasize 524280 kbytes

stacksize 8192 kbytes

coredumpsize unlimited

memoryuse unlimited

descriptors 64

这时你可清楚地看到各项系统资源的使用限制。如果你要取消某项限制，可用指令 unlimit。譬如取消 CPU 的限制，方法如下：

% unlimit cputime

但如此运用会影响所有的 process，如果你只是要对放到背景中执行的工作单独取消的话，你可用我们前面所提到的群体指令来执行该工作。如下例所示：

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

工作控制与退出的关系（指令 nohup）

在正常情况下， C Shell 在系统中所执行的处理程序（process）会随着 logout 而终止，这是因为 UNIX 系统会随着 logout 送出 hangup 的讯号将你所有的 process 终止。但在背景工作则会随着你使用的 UNIX 系统不同而有所差异。有些 UNIX 系统对于执行背景工作会主动以 nohup 的方式执行，它可让背景工作不受 logout 的影响而终止执行。但如果你的系统是属于不主动为你的背景工作加上 nohup 的话，则你一但 logout 则背景工作一样逃不过被 hangup 讯号终止的命运。当然你也不可能得到任何的结果了。这一点请特别特别注意，务必在下指令到背景内执行前搞清楚系统的特性。如果发现有必要自己手动的话，下指令的方式如下：

% nohup command &

C Shell 的内建指令（Built-in Commands）

C shell 的内建指令，它们其实是存在于 C shell 这个程序本身之内。当你的 login shell 被载入记忆体的时候，他们也就随之存在记忆体中了。所以当 C shell 要执行它们时， C shell 会直接从记忆体中读取并加以执行，不需要像执行系统指令那样要经过搜寻档案后，用 fork() 一个新的处理程序，然后用 exec( ) 来执行它。所以在执行效率上会快过系统指令。这是内建指令与一般的系统指令最大的不同处。

在 C shell 这个程序中有不少内建的指令，如常用的 cd、kill、echo、exit 便都是 C shell 的内建指令。或者像 alias、history、limit、set、setenv、source、以及关于工作控制（job control）的 fg、bg、stop 等也都是 C shell 的内建指令。甚至于在后面我们将会为你介绍的控制流程（control flow）功能，也全部是内建指令。

以下我们所要为你介绍是几个比较独立性且非常重要的内建指令。至于在前面我们已经介绍过的 history、alias、控制流程（control flow）功能，在此便不再赘述。至于 C shell 的完整的内建指令请参考附录。[* 作者网站未提供]

umask 指令

umask （显示设定值）

umask nnn （设定 umask，设定值为 000~777 的整数）

umask 指令的功能是用来“限定”每一个新增的档案、目录的基本使用权限（permission）。譬如说当使用者以编辑器新产生的档案，或者是从系统的某处拷贝来的新档案，或者是以输出重导向的方式产生的新档案，或是以指令 mkdir 新建的目录等等，一切新产生的档案、目录，它们的最初使用权限，均会受到这个内建指令 umask 的设定值所影响。就是我所说的“限定”。

指令里的 nnn 所代表的意义与 chmod 指令的 nnn 相似。不同的是 chmod 指令 nnn 是“给于”使用或者是将要改变的许可权限，而 umask 则是“取消”nnn 的使用许可权限。这点是根本性的差异，使用者必须分清楚。

指令 umask 的设定值以三个八进位的数字“nnn”代表。第一个设定数字给使用者自己（owner user），第二个则是设定给用使用者所属的群体（group），第三个给不属于同群体的其它使用者（other）。每一位数字的设定值都是三项不同权限的数值加总，read 权限数值为 4；write 权限数值为 2；execute 权限数值为 1。结合了前三者的权限数值，单一的数字可设定的范围是 0 ~ 7；整体的可设定范围是 000 ~ 777。

要知道设定后会得到什么结果。原则上，方法很简单。就是用最大值减去设定值即可得到你想要知道的结果。对目录而言最大值是777；对档案而言，最大值则是 666。这个方法对目录而言完全正确；但对档案而言会有无法应付的意外。

以下为了说明上的方便，我将以实际运用上，不可能会采用的设定值 067 作为本节例子来加以说明。

譬如当你设定 umask 为 670 ，使用档案的最大值 666 减去设定值 670，得到的是数值是负 4，已超出数值的定义范围变成没有任何意义的数值。但真正使用者会得到的结果却是档案对 other 开放 rw 权限；对 owner 与 group 关闭所有权限。所以使用者无法使用减去的方法来获得完全正确的结果。

表面上，C shell 让系统使用者使用 umask 时只需输入一组 3 个数字的设定值。但是，这组设定值对于目录及档案却有着不同的作用结果。所以农夫我打算更进一步地说明其中运作的细节，让看官们能完全的理解。

对系统程序而言，内建指令 umask 的设定值实际上是群组化的参数，也就是 S_IRWXU、S_IRWXG、S_IRWXO。代表的群组情况如下：

S_IRWXU = S_IRUSR | S_IWUSR | S_IXUSR

S_IRWXG = S_IRGRP | S_IWGRP | S_IXGRP

S_IRWXO = S_IROTH | S_IWOTH | S_IXOTH

也就是说 umask 所设定的三个数字，其实包含了九个不同意涵的参数，用来对映九种不同的使用权限，这些参数会被要产生档案的程

序，或者是要产生目录的程序带入并执行出结果。一个 C shell 的使用者必须要有能力完全掌握 umask 的设定，并演算出设定后所得到的结果。

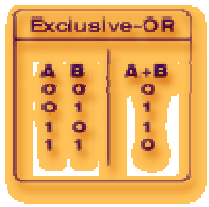
系统在产生一个新目录，会完全使用到上述的九种权限的参数，所以最大值是 777 ，这对 umask 内建指令而言，可以很容易地使用减去权限的方法来获得正确的结果。但在产生一个新档案时，就不是如此。

系统在产生一个新档案时，creat function 只取用 read 与 write 权限相关参数，也就是 access permission bits 里的 S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, S_IWOTH 来定义产生的档案应该具有何种程度的权限。由于 read（数值 4） 与 write（数值 2） 的权限相加的结果是 6。所以最大的有效数值为 666。至于所有的 execute 权限（数值 1）在此被忽略（access permission bits 为 S_IXUSR, S_IXGRP, S_IXOTH），所以 creat function 在输出时一律定义为 0，也就是无执行的权限。会对档案如此限制的理由其实很容易理解。因为几乎不可能有一个使用者他的所有档案都绝对必要被固定成为给予 execute 权限(档案包含的型态相当多，如文字档、资料档、图形档、执行档等等)，如果功能被如此设定的话，反而会造成相当多的系统漏洞，所以必须管制成必要时再由使用者自己来打开 execute 权限（这就是为什么每当你新编辑完成的 C Shell script ，还要用指令 chmod 来加上可执行使用权限才能执行的原因）。

明了上述的原因之后，看官们您能理解，指令 umask 的设定值里面，如果包含了 excute 权限，又运算产生档案的使用权限时，应该要先减去（或者应该称为 disable），才来作运算。所以当设定值为 670，其中是第二位数字是奇数，明显的包含了 S_EXGRP 的数值，所以先减去 1，所已有效设定为 660。此时使用最大值 666 减去 660，得到的便是正确的档案使用权限。

农夫我使用最大值减去设定值的方式来说明，对一般的使用者应该能较容易理解与运用。在前一版的文字说明中，我曾经提到计算的方式此用的是 XOR 运算法则，它才是实际上程序的运算法则。也就是设定值被程序拆成参数后的参数算法则。我想学习过逻辑运算的人都清楚 XOR 的运算方式（如右图所示）。

以下我使用设定值 067 当作例子，以 XOR 的算法则来实际设定会应该得到的结果。



希望看官们，能理解以上的说明（农夫我发觉改写后的版本，看

起来实在相当啰唆)。以下提供一些比较常用的设定值供看官们参考：

文件权限的最大值	设定值	结果	代表的使用权限
666	002	664	rw-rw-r--
666	022	644	rw-r--r--
666	037	640	rw-r-----
666	077	600	rw-------
目录权限的最大值	设定值	结果	代表的使用权限
777	002	775	rwxrwxr-x
777	022	755	rwxr-xr-x
777	037	740	rwxr-----
777	077	700	rwx------

顺便附上单一设定值与文件及目录的互动对照表，如下：

Umask 设定值	文件 使用许可权	目录 使用许可权
0	rw-	rwx
1	rw-	rw-
2	r--	r-x

3	r--	r--
4	-w-	-wx
5	-w-	r--
6	---	--x
7	---	---

假如使用者想要显示 umask 的设定值，可键入指令 umask，设定值即会显示出：

% umask

22

以上所显示的设定值“22”即代表“022”。因第一个数字为“0”时不显示。假如显示值为“2”则代表“002”，显示值为“0”则代表“000”。一般系统的常用的设定值有“002”、“022”、“037”或“077”等几种。接着我们实际来设定指令并观看其使用结果：

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设定或更改这个指令的设定值，最好的方式是将这个指令放在“~/.cshrc”档案中，让 C Shell 来为你执行。如果 login 后还有须要更动，可直接在指令行模式下键入指令重新设定之。如果使用者不自行设定则系统会给于系统的设定值，一般均为“022”。

exit [status value] 指令

这个指令会终止 C shell 的执行，并退出该 C shell。如果你 login shell 中执行，则功能便相当于 logout。如果在 subshell 中执行则回到其 parent shell，并且可以在退出 subshell 时，给于一个执行状态变数的参数值。在不指定的情况下退出的执行状态变数为“0”。这个功能在 C Shell 程序设计时，常用来设定程序不正常结束的状态值，以供我们做执行状态的查询使用，是一个非常有用的功能。请读者特别注意之。

下例中，我们在 login shell 中下指令 csh，其作用便是产生一个 subshell，如此一来我们的执行环境便由 login shell 转移到 subshell 中了。然后我们执行指令“exit 1” 来结束这个 subshell 的执行，并且设定$status 变数的值为“1”。当我们回到 login shell 中以指令“echo $status”显示该 subshell 的执行状态时，所得的结果便是我们执行指

令 exit 时所传回的设定值“1”。在最后使用指令 exit，我们将会退出系统。

```
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] 选项将所读取的指令列入过去指令使用记录（history list）中，但并不执行所读入的指令。

source 指令能从指定的档案中读取指令来执行，常用以执行修改过

后的特殊档案，如“.cshrc”、“.login”档案等。比方你以 vi 指令更改“.cshrc”的 path 变数后，要如何来“执行”呢？你必须使用 source 这个内建指令来执行它。如下：

% source ~/.cshrc

执行后 path 变数便是你所更新的设定值。这可是一个相当重要的内建指令。

在使用 source 指令来读取档案的执行过程中，请注意一个特殊情况。就是一但产生指令无法执行或产生错误时，则执行的动作将会在该指令行被终止，未执行部份将不再执行。关于这种情况，我们用指令 source 来执行一个分离的别名档案来做说明：

假定档案“.aliases”内容为下：

```
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 %
```

当我们以执行指令“source .aliases”时，产生错误信息“Unmatched

'.”。表示档案“.aliases”中有不合语法的 alias 设定，产生了无法执行的情况。此时我们用指令 alias 来看我们执行成功的别名时，发现仅第一行设定成功，而第二至第四行均没有被设定。这是因为我们用指令 source 执行档案“.aliases”时，读取档案的第二行要执行时，产生了的语法错误，指令 source 于是便停止以下各行指令的“执行”所造成的结果。当然第二行以后便不会执行读取的动作了。

此外在不加选项情况下使用 source 指令，执行时所读取执行的指令并不会加入 history list 中。如果有需要加入 history list 中，必需加选项-h。但加上选项-h 的执行方式与不加选项-h 时，有相当大的差异。首先是它只读取整个档案的所有指令行进入 history list 中，但并不执行指令行。其次是它也不会检查指令行的语法是否正确。所以在产生像上述的错误时，加上选项-h 的指令 source，依然会继续读取下一行直到整个档案读取完毕为止。让我们再利用上面的档案“.aliases”来说明：

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

首先我们执行 csh -v，用意在产生一个可观看执行情况的 subshell。当执行指令 alias 查看时发现到并没有任何别名已设定。

```
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
```

此时我们执行指令 source -h .aliases，可明显得看出档案内容被全部读取，而且也没有错误信息产生了。

```
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 %
```

我们再用指令 alias 查看别名设定，结果依旧是没有任何设定。然后用指令 history 却清楚看到所读取的指令行已列入其中。我们可清楚地了解到 source 指令加上选项-h 之后的执行情况，与原先的差别是相当大的。在使用上请多加注意。

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

这两个内建指令分别是用来设定（limit）或消除（unlimit）系统资源在使用上的限制，限定的系统资源与使用单位如下所示。

可设限的系统资源（resource）种类如下：

cputime 占用的 CPU 执行时间

filesize 可使用的单一档案的最大值

datasize 限制处理的资料上限，包含堆叠（stack）

stacksize 处理的堆叠大小

coredumpsize 核心资料转存为档案之上限

descriptors 档案描述词的上限

memoryuse 记忆体使用上限

设限的最大使用值（max-use）单位如下：

nh 单位：小时（仅用于 cputime）

nm 百万位（Megabytes）或单位：分（仅用于 cputime）

mm:ss 分：秒（仅用于 cputime）

nk 单位：千位（kilobytes）此单位为基本设定值

首先我们可使用指令 limit 来显示系统现在的设限情况。

3 % limit

cputime unlimited

filesize unlimited

datasize 524280 kbytes

stacksize 8192 kbytes

coredumpsize unlimited

memoryuse unlimited

descriptors 64

假设要设定 cputime limit 为 1 分钟，键入指令如下：

4 % limit cputime 1

当执行指令使用 cputime 超过限制时，指令将自动被终结，并显示出警告信息。如下所示：

5 % find / -name Cshrc -print

Cputime limit exceeded

想要解除 cputime 限制，或者是要加以设定的话，可用 unlimit 指令来处理之。

6 % unlimit cputime

其它的各项设定方法均与此例相似。

对大部份的系统而言，最常设限的系统资源可能以 cputime 及 filesize 这两者居多。通常是用以防止不明的或无法预期的错误处理程序占用掉有限的系统资源，如硬盘空间或以秒计费的 CPU 时间(或许在今日这种环境的使用限制事实上已经不多了) 。假定你对 cputime 设定限制为 60 秒，则大部份的处理程序如果执行的 cputime 超过 60 秒，执行会自动终止。但如果你要花费常时间去编辑一个档案，碰上执行的 cputime 超过 60 秒的限制时，我敢保证你一定会哭出来（或“妈”出来）。所以，有预期这种情况会产生时，建议你先取消设定的 cputime 限制，再执行 vi 的编辑工作，下指令的方式可

用下面的方法：

% unlimt cputime ; vi long_job ; limit cputime 60

如果你想一劳永逸地免除 vi 到一半被这种错误所终结的恶梦，你也可设定一个别名来代替 vi 指令，来免除像上面那种使用方法，建议的设定如下：

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

dirs [-l] 指令

此内建指令功能与 pwd 指令相似均是显示目前的工作目录，pwd 指令显示的是绝对路径，如“/home1/akira/test”，而内建指令 dirs 显示的工作目录的表示方式，则比较不同。在 home 以下的工作目录，用符号“~/dirname”来表示路径。如果加上选项[-l] 则与 pwd 指令功能完全相同。使用情况如下：

% dirs

~/test

% dirs -l

/home1/akira/test

% cd /bin ; dirs

/bin

由于它会以符号“~”来表示路径的特性，假定说你的 home 目录相当长，而你想把目录显示在题词中，你也可以使用这个内建指令来

设定，请参考以下的例子：

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

这样的设定也不赖吧！

事实上内建指令 dirs 所显示出来的资料其实是目录堆叠（directory stack），关于这个目录堆叠，我们留到下面的 pushd、popd 时再来说明。

echo [-n] 指令

内建指令 echo 是一个写 C shell 程序的相当常用的指令。通常在指令行模式下亦会使用来显示变数的内容，如下：

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

指令 echo 所提供的选项[-n] 功能，是将你的游标停在 echo 所显示出的信息之后，也就是说选项[-n] 不在指令结束后加上 NEWLINE 的讯号。这个选项常用于交谈式的 C shell 程序上。请注意下面的例子：

```
9 % echo "data in :: "
data in ::
10 % echo -n "data in :: "
data in :: 11 %
```

指令 10 后的提词跑到 echo 的内容后面去了，很明显的与指令 9 有所差别。

time 指令

在今日的 UNIX 系统内事实上存在两个 time 的指令，一个是系统所提供的/bin/time，这个 time 指令是 System V 版本所提供的工具，这是为了 Bourne shell 使用者所写的工具；而另一个则是以下所要为你介绍的 C shell 内建指令 – time。内建指令 time 的功能是用来计算指令执行时所使用的各种系统资源的 0 资料。指令使用的语法为下：

指令语法 time command

内建指令 time 后面所接的便是你真正要执行的指令。 System V 版本的 time 指令只能得到三种资料，分别是实耗时间、使用者时间及系统时间。而 C Shell 的内建指令 time 比 System V 的 time 指令能得到更多的信息。它所显示的资料一共分成七个部份。让我们来看下面的例子：

```
% 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”为使用者时间（user time）

第二部份“19.5s”为系统时间（system time）

第三部份“0:39”为实耗时间（elapsed time）

第四部份“52%”是第一部份的使用者时间加上第二部份系统时间后除以第三部份实耗时间的百分比

第五部份“0+372k”为系统的平均分配（shared）记忆体与 unshared 记忆体的大小

第六部份“1018+143io”为输入与输出的资料 block 数量

第七部份“599pf+0w”则代表 page fault 次数与 swap 的大小。

有关于这个 time 内建指令的一些相关的参数意义的详细说明，我将它们放在第四篇的 C shell 变数的整体介绍内的预设变数的设定影响中，请参阅该部份的说明。

nice [+n | -n] 指令

UNIX 作业系统是一个多人多任务的分时（time-sharing）作业系统。所有人的所执行的所有程序均可在系统中同时运作。每一个执行的程序对系统而言都会给于相对映到处理程序（process）。系统以程序定序（process scheduling）的管理方式来安排这些程序，让它们依序、循环地进到核心程序中执行。这其中的程序运作实际上是非常复杂的，但我们只需要注意到一个观念，就是程序定序中有设定优先权的处理方式。而内建指令 nice 的作用便是能让使用者透过它，去调变程序优先权（process priority）中关于计算处理的一项参数。借以控制该程序在同一时间内所能享用的 CPU 资源的多寡。 nice 内建指令用数字“-20”到“+19”来代表程序优先权的高低。数字“-20”优先权最高，而数字“+19”优先权则是最低。一般使用者所能调变的范围是“0”到“+19”，固定的设定值是“0”，也就是一般使用者所能得到的最高程序优先权。而 super-user 所能调变就比一般的使用者为大，范围是“-20”到“+19”（这便是我们称他为 super-user 的原因之一）。

当我们使用指令 ps -l 来显示系统的处理程序时， NI 栏为所代表的数字就是程序优先权的数值。我们先来看一般指令的程序优先权情况，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

我们可清楚地看观察到 NI 栏的数字均为“0”。现在让我们降低程序优先权来执行指令 ps -l 看看结果：

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 RN p0 0:00 ps -l

看 NI 栏的程序优先权为“10”，正如我们所下的指令效果。

能妥善地运用调整指令的程序优先权，将有助于提升系统的使用效率。这点在一个忙的不得了的机器上，善用指令 nice 整体的工作效率将会有所改善。譬如你 Open Windows 的环境下，一面在执行比较高优先的工作的同时，一面在操作一般的系统指令，如果你能适当地将次要的系统指令操作降低程序优先权，就会缩短高优先的工作时间。比较方便的做法是以 nice 指令来执行一个较低优先的 subshell 或者是开一个较低优先 xterm，则你在这个 subshell 或 xterm 之下所操作的任何指令，都会得到较低优先的程序处理权。

C shell 的内建指令 nice 固定的程序优先权数值为“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 RN p2 0:00 ps -l

当你运用 nice 指令时请注意一点，在 nice 指令不接受别名所设定的“指令”，也就是说要用 nice 来下指令时，别名的功能将失去它的效用。请小心使用之。另外与 nice 指令相关的是 renice 指令。 nice 指令是要在下指令时用的，而 renice 指令则是当你下完指令后，才想到要更改程序优先权时使用。一般用户要使用 renice 指令只能将程序优先权降低，且一但降低后便不能再将它提高。而 super-user 则不受这些限制，并可针对使用者会或者是某个群体程序（process group）或者是某个程序做程序优先权的提高与降低。如果有非常重要的工作急着要提早执行完毕，你可以去求求 super-user 用 renice 来加速一下，你便可知道“电脑特权阶级”的滋味！

rehash、unhash、hashstat 指令

这三个内建指令 rehash、unhash、hashstat 在功能上息息相关于一项 C shell 对寻找指令档案的加速做法。这项加速法叫“internal hash table”。这三个内建指令便是用来做有关于“internal hash table”的资料更新、不使用的设定及显示“internal hash table”的资料状态。

C Shell 运用了 internal hash table 的方式，来加速在 path 变数的目录群中搜寻指令执行的速度。也就是因为使用了 internal hash table 的来记忆指令的位置，以至于产生了一个使用上必须注意的小小限

制。这个限制便是当你新加入一个可执行的程序到 path 变数的目录中，如果你不将 internal hash table 更新的话，你只能在该程序所在的目录下执行。当你改变工作目录到别的目录中要执行那个新程序，虽然它确实是在 path 变数的搜寻目录中，在理论上应该会找到它，但其实不是这回事。你执行时将会惊讶地发现“: Command not found.”的错误信息。千万别以为电脑又出毛病了，这是正常现像。你在这个时候只要用指令 rehash 将 internal hash table 更新便可以搜寻到该新增的程序了。让我们来设定一个比较单纯的环境来实际试一试：

```
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
```

从上例中，我们清楚地看到指令 6 的执行结果是“test: Command not found.”，而经过指令 7 用 rehash 内建指令将 internal hash table 更新后，新增的执行档 test 便能被搜寻到，并由执行信息中我们清楚地看到 test 档所在的目录。

最后为你介绍的是指令 hashstat，它会显示 internal hash table 的三项统计性资料。如下例：

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

第一部份是是使用 internal hash table 的有效数值。第二部份是则是使用失效记录。第三部份是成功的百分比。关于第二部份为何会产生失效的情况，我们将为你稍加说明。

当我们键入指令后， C shell 经判断不是内建指令或别名后，便使用 internal hash table 搜寻该指令位置，如果发现没有该指令，便输出信息“xx: Command not found.”。如果说你的 path 变数中没有“.”（dot）这个符号的话，便不属于“失效”。因为失效的意义是指 C shell 有使用到 exec() 这个 system call 为你执行该指令却产生错误时，才算是“失效”。譬如执行时产生 Permission denied 的情况，便是属于失效的情况。或许你会感到奇怪，为何我们在前面所提到的 xx: Command not found.”的情况中，为何要排除“.”（dot）这个符号呢？因为它在执行上比较特殊。当 C shell 使用 internal hash table 在搜寻

指令位置时，碰到“.”（dot）这个符号便会以 exec() 这个 system call 来执行“./command”。如果执行失败，将马上为你记上一笔。如果你所设定的 path 变数有加上符号“.”（dot），而且还是放在最前面，在此建议你最好考虑将它移到 path 变数的最后面。因为在你每次使用到系统指令时都会发生一次不必要的错误，在无形中将会降低系统的效率。当然最好是放弃在 path 变数使用符号“.”，需要时在以“./command”的方式来执行也可以。

由于 C shell 对于“internal hash table”的使用系统所定义的初始值是使用的状态，所以当你不想要有上述这样扰人的情况，要放弃使用 internal hash table 这项功能也可以。 C shell 提供你内建指令 unhash 来解除这个问题。但就整体的使用效率而言，最好还是别轻易放弃使用 internal hash table。因为当你放弃使用 internal hash table，搜寻指令执行的方式将会改便成最原始的状态。就是要执行一个指令便逐一的到 path 变数的目录群中去尝试着执行。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

在放弃使用 internal hash table 前，指令 hashstat 告诉我们“失效”为“0”。然后我们键入 unhash，放弃使用 internal hash table。接着我们操作一般的系统指令如 grep，在显示使用情况，却得到失效”为“1”，这是为什么呢？因为不使用 hash table 后，要执行指令 grep 的动作变成到 path 变数的目录群中去“尝试执行”，第一先以“/usr/ucb/grep”执行，发现没有该指令，所以 exec() 的动作失败。再来以“/bin/grep”执行，该指令存在， C shell 便为你执行该指令。这便是放弃使用 internal hash table 的指令执行方式。相当地“原始”是吧！请考虑清楚再使用。如果要恢复使用 internal hash table，键入 rehash 便可。

exec 指令

指令语法 exec command

内建指令 exec 的功能与用途是相当特殊的。如果使用 exec 来执行“指令”，在“指令”执行完毕结果输出之后，原先的 C shell 也会跟着终结。来看下面的例子：

6 % exec date

Sat Oct 15 11:29:05 CST 1994

login:

当我们在 login shell 中以 exec 来执行指令 date，在指令结果输出后，login shell 也终结。换句话说，就是执行完指令后便自动 logout。因为以 exec 执行指令时，并不会另外呼叫 fork() 这个 system call，所以不会产生新的处理程序（process）。而 exec 所执行“指令”的处理程序会“占用”原来呼叫 exec 内建指令的处理程序，而且程序号码（PID）及环境变数等执行条件均不会改变。就是因为这个会产生这种“代换”的动作，所以当 exec 所执行的“指令”结束后，便无法再回到原来的执行环境中了。这便是在 login shell 中执行此内建指令后会 logout 的原因。所以千万别在你的 login shell 中以 exec 来执行指令。如果你还不想 logout 的话！

其实在指令行模式下会使用 exec 来执行指令的可能性是相当小。因为这个指令大都是运用在“.login”档案或 C shell 程序设计中。譬如说一个使用者想要对自己的签入过程加上一些选项，用来执行自己的程序（如备份资料或清理旧资料等）。而且在执行这些程序后，并不希望再进入到系统中。也就是说进入这些特别的选项，执行完特殊的工作后便要离开系统了。在做法上，你便可在“.login”档案中以这个内建指令 exec 来取代一般的执行方式。来达到你所要的特殊执行效果。

当然你也可以利用这个内建指令 exec 的执行特性，在你最后要离开系统前，以它来执行结束前的最后工作，它会为你自动 logout。这倒也相当方便。

eval 指令

指令语法 eval argument ...

内建指令 eval 的功能是将引数（argument）读入 C shell 中，然后在加以执行。在 C shell 程序设计运用上，比较常看见。让我们先来看下面的情况：

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

在指令 6，设定变数 vcom 为'ls -l ; date'。当我们用变数的形态来执行“$vocm”，却发现有两个错误信息，告诉我们“; not found”及“date not found”。会造成这种错误的原因，是因为 C shell 对于这种变数的解析语法，无法辨视特殊符号所造成的。如上例的变数，符号“;”与指令 date 均 C shell 被误解成是指令 ls -l 后得“档案名称”。所以才会有“not found”的信息传出。

内建指令 eval 便是用来应付这种情况。我们将上面的变数改用 eval 来执行：

```
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
```

由于指令 eval 将$vcom 当成引数读入再加以执行，所以“not found”的误解情况便消失了。其实在作法上使用指令 eval 便相当于以下的用法：

```
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 程序设计里运用的话，你也可应用以下的方式：

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

不过这些变通的方法都不如使用内建指令 eval 来的方便。另外在使用内建指令 eval 上也有相当多的技巧，让我们来看一个变数互换的技巧：

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

```
% echo $a
$b
%eval echo $a
swapping
```

在上例中，变数 a 的内容是“$b”，当我们以一般的 echo 指令执行时，我们发现仅是将变数的内容显示萤幕罢了。但是，当我们以指令 eval 来执行该 echo 指令时，却输出变数 b 的内容。相信你已经知道要如何运用了吧！

对于这个内建指令，在这里要告诉你一个比较不好的消息，就是如果你所使用的 UNIX 作业系统是 SUN OS 4.1.3 的话，它有存在不少 bug，使用上请多多小心。

repeat 指令

这个指令相当的特殊（当然也是懒人最常用的用），它的功能是，“重复执行”你所指定的指令。使用的语法如下：

repeat 执行次数 执行指令

让我们来看一个例子：

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

C shell repeat command

C shell repeat command

%

如果当你真有需要重复执行某个指令很多次时，这个内建指令可好用的很。这种例子并非没有，譬如说执行多次才令人安心的指令 sync，便是使用 repeat 的最好时机。

pushd [+n | dir]、popd [+n] 指令

这两个内建指令在使用上有很大的关连性，所以我们放在一起为你说明。当你必须在几个固定的目录来回地操作时，你便可以用得上这些改善你使用效率的贴心的内建指令。以下让我们来看一个实际的例子：

例子一

% cd ~/project/project-a （改变工作目录）

... （修改或编辑一些档案，此部份省略）

% cd /usr/etc/local/bin （改变工作目录）

... （修改或编辑一些档案，此部份省略）

% cd ~/project/project-a （改变工作目录）

... （修改或编辑一些档案，此部份省略）

% cd /usr/etc/local/bin （改变工作目录）

... （修改或编辑一些档案，此部份省略）

在上面的例子中我们可以很清楚的看到使用者为了短暂的工作型态需要，必须到三个不同目录下去修改一些档案，而且可能会重复多次。而上面用的是 cd 指令来更改工作目录，这事实上是相当累人的没有效率的。 C shell 特别为了这种情况开发了三个内建指令，也就是 popd、pushd 及 suspend。

这三个指令的使用概念是为你建立一个目录堆栈（directory stack），用来供你放置必须常去的工作目录，以便让你的更改工作目录更加方便及有效率。让我们先来看看以下各个指令使用时所产生的目录堆栈的数据变化情况：

1 % pwd

/home1/akira/project/project-a

2 % pushd ../project-b

~/project/project-b ~/project/project-a (directory stack 的内容)

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，工作目录是在 /home1/akira/project/project-a，我们请读者注意到事件 2 也就是内建指令 pushd；它不但兼具 cd 指令的作用，同时也会将你所键入的目录放入缓存器中，所以当你如事件 2 的方式执行后，所显示出的信息便是已被放入缓存器中的目录，而第一个目录是现在的工作目录，其他的目录在出现的次序上也请加以注意。接下来让我们用 pushd 指令来重新做一次例子一，看看有什么效率上的改变。

```
% cd ~/project/project-a （改变工作目录）
... （修改或编辑一些档案，此部份省略）
% pushd /usr/etc/local/bin （改变工作目录）
/usr/etc/local/bin ~/project/project-a
... （修改或编辑一些档案，此部份省略）
% pushd （改变工作目录）
~/project/project-a /usr/etc/local/bin
...
% pushd （改变工作目录）
/usr/etc/local/bin ~/project/project-a
... （修改或编辑一些档案，此部份省略）
```

是不是简单、省事多了呢？对了还记不记得我们提过的内建指令 dirs，它可以为你显示目录堆栈（directory stack）的内容，所以当你

放很多目录在目录堆栈中时，想要知道目录堆栈的内容就可以用的上。让我们来看看下面的例子，顺便为你介绍另一个 C Shell 内建指令 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 ” 所代表的是目录堆栈中的第三个目录，pushd 指令在不加上任何自变量时，所代表的便是目录堆栈中的第一个目录。内建指令 popd 的功能就如同事件 60 及 61 所显示出来的，就是将目录堆栈中的的某个目录去除掉。事件 60 是除掉目录堆栈中的第二个目录；事件 61 是除去目录堆栈中的第一项数据。但是请注意！当你如事件 61 去除掉第一个目录堆栈的内容时，你的工作目录也会自动改变到目录堆栈中的第二个目录，因为这时它已经变成目录堆栈的第一项了。关于这一点，你可以从事件 61 及 62 看出来。

相信以上的例子中你可能会发觉，当你放入目录堆栈中的目录在两个的情况下，这项功能还称得上非常好用；但是一但目录堆栈的内容多了，恐怕就不是人人都能用得顺心如意的了。对于这种情况，建议你不妨采用设定 cdpath 变量的方式会来的好些。

某些旧版的 UNIX 操作系统的 directory stack 对于 symbol link 的功能会产生一个小小的 bug，由于这个 bug 会对内建指令 pushd 及 popd 的使用会有影响，使用上应注意到此点，关于这一点 bug 稍后我们会在变量 hardpaths 中还为你说明。

引号的运用与指令的关系

在 UNIX 系统中如果在指令、文件名等运用，或者是在变量的设定上，碰到了空白（space）或 TAB 字符夹杂在当中时，如果你还用一般的方式来处理，则往往会造成一些不必要的指令语法错误或者是设定上的 bug。如果你实际的运用上也常碰上这种困扰的话，请不用担心，仔细看以下这些符号的运用，相信你所需要得答案与运用法则便在其中。

C shell 对于这方面的处理上，提供了三个功能相近的符号（'"`）来做字符串的处理。这三个符号必须以“成对的”方式使用才有效用。换句话说，就是用这些符号来『括住』我们所要处理的字符串。当我们用它们来括住待处理的字符串时，使用不同的符号便会有不同的结果。但有时会因功能相近且符号也相近的情况下，常常会让使用者分辨不清，使用上请小心注意符号本身所代表的特性。

就它们整体的使用面来看，可区分成指令、文件名与变量这三个方面，以下我们先就前两项来说明之，而关于变量方面，将在下一章中再加以讨论。

单引号（ ' ）的运用（single-quotes）

当我们以单引号来来括住数个以 space 字符所分格开来的字符串时，最主要的目地便是将它们便成是一个单一的字符串来运用。譬如说，在 UNIX 系统中，文件名是允许使用 space 字符在当中的。

像“data a1”这样的档名是合法的。但如果我们想将某个指令的输出重导成“data a1”档案，用一般的方式一定会产生错误的结果，如下：

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

像以上的语法使用“data a1”因为有 space 字符的分格而被当成是两个独立的字符串，所以该指令的“data”与“a1”分别被 C shell 解译成，“a1”是指令 ls -l 的档案自变量；“data”则是重导向的文件名。如果该目录下有档案“a1”存在的话，该指令将不会有错误信息传出。而是成功地将“ls -l a1”的数据输出重导到档案“data”中。这指令的作用实际上已相差了十万八千里。如果是使用在 C shell 程序设计中，这在 debug 上，可真的会累死人。当然像这种档名上的特殊问题，一般是不会这样做的。它的用义是想告诉读者含有 space 字符的字符串，在实际上是与一般有所差别的。一但使用观念错误，可能会有相当麻烦的情况产生。这一点请多加注意。接下来我们再来看一个比较常碰到的情况。

当我们想到档案“find.data”中，找出含有“Permission denied”的每一行时，如果我们用下面的方式来下指令：

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

实际上你所得到的结果将不是含有“Permission denied”的字符串，而是仅含有“Permission”字符串的每一行。而且在指令中的“denied”

字符串将会被 C shell 解译为是指令 grep 的档案自变量，如果该目录下没有“denied”这个档案，将会有像下面这样的错误信息输出：

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

要想正确地得到你所要的结果，以上的两个指令可使用单自变量来括住字符串，便可解决上述的种种问题。如下：

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

另外我们也时常运用单引号在设定较复杂的别名上。如下：

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

像如此的整个指令串的部份，必须用单引号来将它们括住，要不然在语法上会产生错误。当然在单引号之内，语法上也允许再使用像双引号那样的符号在里面。如下面这个别名的设定的例子：

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

除此别名的设定之外，像指令 echo 也常会运用得上，如下面的情况：

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

如果你使用 echo 指令要输出一个带有 TAB 字符的信息。如上例你会发现，原本设定好的格式，在输出时完全变了。这是因为 C shell 处理指令后的自变量，并不认识 TAB 字符，仅仅会读入自变量本身。输出结果时各个自变量之间则用一个 space 字符来加以区格开来。所以造成像上面这样的结果。如果你使用单引号括住，情况便会有所改变。请看下面的例子：

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

因为有单引号括住的原因，使得四个自变量及它们之间的 TAB 字符，合为一体，形成为同一个自变量。指令的输出，便因此而保留了 TAB 字符，输出结果才会与原来的格式相同。另外一点值得一提的是，变量的符号“$”在单引号内是无法发挥变量作用的，它将仅仅是一个普通的字符，而无任何的特殊意义。如下所示：

```
3 % set d = date
4 % echo $d
```

```
date
5 % echo 'variable $d'
variable $d
```

关于此一特性，请特别牢记在心。因为这也就是单引号有别于双引号在运用上的最大差别处。

双引号（ " ）的运用（double-quotes）

让我们延续上面的问题，来看看改用双引号的效果。

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

从输出的结果可清楚地看到 $d 变量，在双引号之内依然可发挥它变量的功能。其实不光是符号“$”有如此的差异，就连以下我们将要为你介绍的倒引号 ` 与倒斜线 \ ，也均是如此。所以请读者注意到，在双引号内的特殊符号，如 $、\、倒引号等，其特殊功能均不会丧失。但如果将它们放入单引号中，则符号的特殊意义与功能会消失，而仅只是符号而已。

不过，不管是双引号或者是单引号，它们对于 space 字符与 TAB 字符的处理形式上，其功能完全相同。如先前我们所提到的例子：

10 % ls -l > 'data aa'

11 % grep 'Permission denied' find.data > datafile

如果将单引号改成双引号，其效果完全相同，两者都会将它们变成是一个字符串。

倒引号（ ` ）的运用（backquote）

倒引号的功能是让我们去括住指令，此功能常用在 echo 指令的内容中，或者是一些设定变量的场合上。使用的方式相当简单，如下所示：

`command`

让我们先来以下面的例子来说明倒引号的功能。

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

There are 2 users logged on

`who|wc -l` 是被倒引号括住的一组指令，我们将它放在指令 echo 所要输出的文字之中。在 C shell 在解译执行整行指令时，会先执行 `who|wc -l`，然后将其结果传回 echo 指令中，再由 echo 指令

将整个结果输出到屏幕上。

倒引号在使用上请小心，千万别将它放在单引号之内，如下面这个例子：

```
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
```

当然倒引号在双引号内还是有功用的，但请仔细比较指令 5 与指令 2 的输出，在指令 5 的输出有一大段空格，这是因为倒引号所括住的指令的输出代入指令所占用的字符长度的结果。可别忘了双引号是会保留原来格式的特性。而指令 2 的输出会没有多余的空格，便是因为在指令读取自变量时已经被乎略掉的结果。

倒引号在使用上，常常被运用在变量的设定上。有关于这一点，我们将会在下一个章节中为你详细介绍。

反斜线“ \ ”的运用（backslash）

我们使用倒斜线的功能是挡去某些俱有特殊意义的字符，让它们的特殊意义失效。譬如某个别名的设定、重导向的符号、变量的符号等等。让我们来逐一举例说明之。首先我们来看一下关于前面我们曾

经说明过的别名设定：

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

指令 7 显示了“rm”已被别名的功能设定成为指令 rm -i，如果我们要使用“rm”，并且要避开别名的功能，你可使用倒斜线“ \ ”挡在“rm”之前，便能除去别名的设定，使用原系统指令“rm”了。

此外在指令 echo 的运用上，最常会运用到倒斜线。如要使用 echo 输出一个特殊的符号时（像 >, <, \, *,...），如下：

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

在指令 9 中，很明显的错误信息告诉我们，“>”是个重导向的符号，所以指令执行失败。在指令 10，我们以倒斜线来挡在符号“>”之前，便顺利地输出该符号“>”。这便是倒斜线的功用。当然如果你想要以指令 echo 来输出“\”也是可以的，你只须用两个连续的倒斜线便可。如下：

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

两个连续的倒斜线意思便是，挡去倒斜线的特殊意义。在上例中，如果只用了一个倒斜线，这个在行尾的倒斜线所代表的将不是上面我们所提到过的意义了。行尾的倒斜线的功能则是连接下一行的意思。譬如当有下指令时相当长，想要以两行的方式来完成它，则可以在第一行的行尾加上倒斜线，然后便可在下一行继续再见键入未完的指令。我们来看下面例子的用法就会明白了。

```
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 %
```

C Shell 变量的整体介绍

C shell 对其本身在整体的环境控制与部份功能的设定和使用上，都有专属的变量，提供用户自己设定与应用。同时在变量的型态上，也区分为环境变量（environment variables）与默认变量（predefined variables）两种。前者相当于整体变量，而后者相当于局部变量。以下我们将为以这两大分类来为你分别介绍 C shell 本身所制定的各种变量。

环境变量的设定影响（environment variables）

环境变量的设定目的在于管理 shell，这是它之所以重要的原因。它的特性相当于整体变量（global variable）。也就是说，你仅需要把环境变量设定在你的“.cshrc”档案中，由 login shell 所产生的 subshell 或者是执行的 shell 文稿、程序或指令等，均不需再重新设定，便可以直接呼叫或使用该变量。所以环境变量是具有遗传（inherited）性的。因为在 UNIX 操作系统中，由一个处理程序（process）会将它全部的环境变量遗传给它所衍生出的子处理程序（child preoceese）。

譬如你在 login shell 之下执行一个 vi 指令，设定的 TERM 变量会决定使用何种终端机模式，同时 vi 程序本身也会继承了原来的 login shell 所定义的所有环境变量，所以当你想要在 vi 程序中用指令“:sh”的方式产生一个新的 shell 时， vi 程序还会依据你所定义

的 SHELL 变量，产生那个你所指定的 shell 的原因。当然因 vi 程序所产生的 new subshell，依然会继承来自于 vi 程序的所有环境变量。

C shell 的环境变量全部都是以大写字母命名。事实上这也是一个不成文的规定。所以当你要自行定义一些环境变量时，请你也能够这样做。设定环境变量的使用语法如以下所示：

设定语法 setenv ENVNAME string
解除设定语法 unsetenv variable
显示所有设定 env

C shell 的环境变量并不多，仅有基本且重要的特殊信息才被列入。如用户的签入目录（login directory），存放邮件的目录，终端机的模式，执行指令依据的搜寻路径等。在这些环境变量中，部份会由系统依据某些特殊档案内的数据，为用户自动设定初始值。如 HOME 变量以及 USER 变量（有些 UNIX 版本不叫做 USER 变量，改称为 LOGNAME 变数）的初设值便是来自于“/etc/passwd”档案。又如 TERM 变量初始值是来自于档案“/etc/ttytab”。除此之外，环境变量中的 HOME, PATH, MAIL, TERM 等，还会将它们的内容拷贝到相同名称的默认变量中，以做为默认变量的初始值。不仅如此，这两者之间还保有一种互动的关系，也就是其中的任何一方有改变，另一方变量也会自动地将变量内容更新。这些都是环境变量的特点。以下让我逐一地为你介绍每一个 C shell 的环境变量。

环境变量 HOME 与默认变量 home

一个用户在签入后，环境变量 HOME 的初使值来自 /etc/passwd 档案中的，并将设定值拷贝给默认变量 home，当成是 $home 的初始值。所以对一般的使用者而言，这两个相关的变量都不需要特别去设定它们。

假使有一天，你因为执行计划的需求，想将原 home 目录 /home1/akira 改到目录 /home1/akira/project，最简单的作法是在“.cshrc”档案中加入下行：

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

你或许会感到奇怪，为何设定是默认变量，而不是环境变量呢？这个原因是，当 login shell 产生之后，再去做修改环境变量 HOME 的动作时，默认变量 home 已经不会随着它改变了。所以会造成环境变量 HOME 与默认变量 home 所分别设定的 home 目录不同的情况。这样便会造成很多问题。因为由 login shell 所产生的 subshell 或执行任何程序，它们所继承的是 HOME 变量的设定值，所以他们的 home 目录为 /home1/akira/project。而 login shell 的 home 目录则是为 /home1/akira。这便是我们不采用设定环境变量的原因。

当你在 login shell 中再重新设定 home 的默认变量，它会将设定传给环境变量 HOME。如此两个变量的设定值才会一致。自然便不会产生上述的问题。

另外我们在此要为你厘清一项重要的观念，那就是 C shell 用来代表 home 目录的符号“~”，它实际上所代表 home 目录是来自于默认变量 home 的设定值。而与环境变量 HOME 一点关系也没有。让我们来看下面的例子：

```
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，我们可清楚地看到两个变量的设定值已经不同，指令 3 则明显地看出特殊符号“~”的设定值和默认变量 home 是相同。这样可够清楚了吧！所以说，更改 home 目录对使用环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盘性掌握自己的所有环境设定之前，最好不要轻易去更动它。如果真有需要更改，除了上述的情况之外，关于 C shell 的各种起始档案与所有的特殊起始档案，最好也将它们拷贝一份或者是使用连结的方式，将他们放一份在新的 home 目录下，这样会比较安全些。想要“搬家”，请千万小心！！

环境变量 SHELL 与默认变量 shell

此变量的系统初始值便是 /bin/csh，此变量在用途上并不广泛，所以一般的使用者也不太会注意到这个变量的设定。此变量的所定义的 shell，是设定给 UNIX 系统的部份公用程序（如 vi 、 ex 或 mail 等指令），在程序操作中需要产生 subshell 时，会依据这个变量所定义的 shell，产生你所设定的 subshell。在变量的设定上，你所指定的 shell 必须使用绝对路径。如果你在使用上，有必要改变系统设定值的话，设定的语法如下所示：

% setenv SHELL /bin/sh （如果是其他的 shell 也一样必需使用绝对路径来设定）

此环境变量 SHELL 的特性和环境变量 HOME 相同，就是 login 以后再更此变量的话，并不会同步更改默认变数 shell。而默认变量 shell 的设定语法如下：

% set shell /bin/sh

如果没有特殊的因素或需要，此变量请保持系统的设定值。

LOGNAME 与 USER 环境变量

LOGNAME 环境变量为签入者的用户名字，USER 环境变量则为现在用户名字。

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

由以上的例子中，相信你一定能够清楚地认清，这变量两兄弟的在定义上真正的区别了吧。如果想要运用这两个环境变量在 C shell 文稿中，请确实注意它们之间的差异性。

环境变量 MAIL 与默认变量 mail

环境变量 MAIL 所设定的参数是一个绝对路径的 mail 文件名。设定值便是提供给指令 mail 作为该读取那个 mail file 的依据。这个环境变量你没有设定，系统会自动给一个系统的设定值。系统的设定值如下：

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

在 C shell 中与 mail 功能相关的变量，除了 MAIL 环境变量之外，还有一个默认变量 mail 。事实上这个默认变量 mail，用户而言是比较重要的。因为这个变量的作用是为用户检查是否有新的 mail 传入。此默认变量的设定方法如下：

使用语法

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

默认变量 mail 的设定方式有两种，一个是仅指定绝对路径的 mail 文件名；另一种则是，设定搜寻时间及绝对路径的 mail 文件名（时间的单位为秒）。如果你没有加上时间的参数的话，则系统会给予“10 分钟”设定值，也就是相当于你设定为 600 秒。另外指定的 mail 档案的数量有并不限定只能有一个，可以是一个以上。完全视你所需而定。以下是一个设定的例子：

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

最后为你说明一点，环境变量 MAIL 的设定只会影响“children process”，与默认变量 mail 在设定上并没有“互动”的关系。也就是说它们的设定值是独立的不互相影响的。

EXINIT 环境变量

这个变量是专属于 vi 与 ex 指令所使用，它设定的参数会作为这两个指令的环境初始设定。设定的语法如下：

setenv EXINIT 'set 选项'

由于此变量的选项超过 40 个以上，而且都是关于编辑器方面的参数，所以在此并不准备为你做详尽的介绍，仅列出选项的数据供你参考运用，如下表：

EXINIT 变量的设定选项一览表		
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

关于这项设定数据，你可以在 vi 模式下，用“:set all”的方式显示所有的选项设定情况。（本项设定数据会因为 UNIX 操作系统的不同而有所差异，但不会相差太大）

以下我们来举个较常使用的设定值，供你参考：

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

上例中我们设定了四个选项，nu=number, ai=autoindent, sm=showmatch, sw=shiftwidth=8。

事实上，在你启动 vi 编辑器， vi 程序会先在你的 home 目录下找寻一个叫“.exrc”的特殊档案。这个特殊档案作用就是用来设定这些选项，同时它还可以做类似于 aliases 功能的动作，叫做“map”。如果你要设定得相当繁杂的话，你可以考虑放弃设定这个变量。并将你想要设定的所有选项编辑到“.exrc”档案中。或许还比较适合些呢！提供你作参考。

TERM 环境变量

这个环境变量与 vi 编辑器、more 指令及相关于屏幕光标位置的指令有相当大的关系。一般设定在“.login”档案中。设定的语法为：

设定语法 setenv TERM 屏幕模式

例子

setenv TERM vt220

这个变量对于在“console”操作的使用者而言，可能会忽略。但是对使用个人计算机或者是终端机的用户来说，它的影响就比较大了。

默认变量的设定影响（predefined variables）

在 C shell 本身所制定的两种变量当中，真正用来控制 C shell 功能的是以下我们所要为你介绍的这些默认变量。制定这些变量关系到指令的执行、hisroty 功能的使用、 指令 time 的显示、job control 的显示、指令行模式下的信息显示、wildcard 功能的使用、退出 C shell 的设定、filename completion 功能的使用、输出/输入重导向功能的重写保护等 C shell 的特殊功能。所以对一个使用者或对于 C shell 的使用而言，这些变量是相当重要的。因为在上述的特殊功能中，有一小部份功能如果不自行设定，系统的初始设定值是不使用的，关于此点请读者稍加注意。

C shell 的默认变量在设定上为了要和环境变量有明显的区别，一般的习惯均使用小写字母来做为变量名称。各项设定的语法如下所示：

设定语法 set variable

set variable = string

解除设定语法 unset variable

显示所有设定 set

C shell 的默认变量一般均集中设定在“.cshrc”档案中。为什么要集中到“.cshrc”档案中呢？因为默认变量的特性是局部性的，在实际上当你在设定这些变量之后，这些默认变量的设定状态，并不像环境变量那样，会将设定值“遗传”给在它之下所产生的 subshell。如果你想要每个 C shell 及它的 subshell 均要保有你想要的默认变量的设定状态，你便应该将它设定在 C shell 的起始档案“.cshrc”中。因为 C shell 本身的在产生 subshell 时，会自动地去读取“.cshrc”档案，以该档案内的设定做为 subshell 的初始环境设定。所以，你想要的各项默认变量的设定，为了要能做到，自 login shell 到所有的 subshell 均能完全一致，最佳的方式便是将它们设定在“.cshrc”档案中。如果是临时性的、短暂性的默认变量设定，当然还是以在指令行模式下手动设定方式比较适合。所以这类的变量在设定上，请务必留意使用上的需要，来选择适合的、正确的设定方式。

以下便让我们来逐一介绍 C shell 的各个默认变量。

path 指令搜寻路径变量

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

paht 变量所定义的路径次序便是执行指令时搜寻次序的依据。如果不自行设定，则大部份的 UNIX 系统均自行设定为（. /usr/ucb /bin /usr/bin）。其中符号“.”代表“目前所在的工作目录（current directory）”。将目前所在的工作目录放在其他目录之前所得到的结果是，在目前所在的工作目录下所有可执行的程序优先执行。此法乃用以解决自行开发的程序名称与系统指令相同时，在执行上的问题。但对指令搜寻执行的效率而言，并不理想。因为对你每次执行的系统指令都会先对目前的工作目录作搜寻再往 /usr/ucb 等目录去寻找系统指令。这种设定次序对整体指令的使用效率是较差的。在自行设定时请注意此点，尽量将你的路径安排的有效率一点。

在设定这个变量时，有些软件或者是用户，可能会因为某种理由而采用变量 path 加到自己本身的设定中。如果你在“.cshrc”档案中，发现有类似下面你所看见的 path 设定方式，则请你特别小心：

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

这样的设定方式在语法上是没有问题的。设定变量 path 的路径是原变量的内容加上一个新的路径“~/project”。这样的设定方式或许在设定上相当好用，但在实际的运用上却也有可能造成非常不良的连锁

效应出现。比如，你修改“.cshrc”档案之后要使用内建指令 source 做更新设定的这个动作来说，原变量的内容会再一次被带入设定的中，造成路径“~/project”重复设定，影响到指令搜寻路径的精简。如果你重复做上许多次，后果可就不么美妙了。这种设定的影响在使用上请小心。

cdpath 改变工作目录搜寻路径变量

设定这个变量的作用与 path 变量有点相似，不同的是变数 path 是供你找寻指令用的；而 cdpath 变量则是让你在改变工作目录时找寻目录用的。它可以让我们在任何的目录下，很容易地到我们想到的工作目录中。譬如说你会常到自己 home 目录之下的 akbin 目录中修改或编辑 C shell 文稿。那么你便可将 home 目录设定到变量 cdpath 中。如此一来，不管你是位在那个目录之下，你都可以当成 akbin 这个目录便在于目前的工作目录之下，只要使用“cd akbin”，便可以到该目录了。请看以下的实际例子：

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

觉得这功能如何？有够好用吧！没有设定的你赶快设定吧!! 他的设定语法和变量 path 完全相同。如果要设定两个 path 以上，用 space 区格开来便可。不过有一点请注意，就是利用它来找寻目录时，是依据你所设定的路径先后次序。万一碰到相同名称的目录时，第一个选择是目前的工作目录，再来便是你所设定 cdpath 目录的次序了。所以在规划目录名称时，最好不要有相同名称的情况产生。

另外有一点要说明的是，这个变量仅对“cd”“chdir”“pushd”“popd”等指令有作用，其他的指令就没有任何的效用了。

prompt 提词变数

C shell 的题词设定是相当富弹性与变化的，你可以设定的很简单；也可以设定得什么信息全都在上头出现，只要你高兴。不过基于整个系统的使用效率来衡量它的话，最好别设定的太复杂。花进心思写个程序来做一个题词，实在有点小题大作。如果每个人都这样使用系统的话，总有一天吃亏的还是自己。以下我们仅提供几种简单实用的题词设定，供读者做为参考：

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

上例是将 history 的 even 数字加入题词中。

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

上例是将 hostname 与使用者名字加入题词中，此设定很适合使用网络的工作者。

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

上例相当适合常忘记身在何处的使用者。我们将提词变量设定到 aliases 中，当每次执行指令 cd 时，便自动再设定一次，结果下两行所示。

以上的例子均是简单实用型的题词，为了喜欢复杂的读者，特地提供下面这个较复杂的例子，供你在设定上的一些灵感。

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

上例的题词分成三个部份，第一部份是设定提词的 aliases，第二部份是设定指定 cd，并且将设定题词的 aliases 加入其中，第三部份是则是执行第一个提词的 aliases 设定。你可以将这三部份加到“.cshrc”档案中。以下便是出现的题词样本：

（空白行）

akhost::akira_/etc/adm/acct

1 % cd ~

（空白行）

akhost::akira_/home1/akira

2 %

history 储存指令使用记录变量

使用语法 set history = 30

指令使用记录（history）是 C Shell 的内建指令。如果这个变量没有设定的话， history 的功能便不会启动，请使用者加以注意。此变量所设定的数字，就是储存过去的事件的数目，数目越大虽然更有助于你利用过去的指令，但是相对的越占内存的空间。所以一般的设定值差不多在 30 ~ 100 左右。

histchars 指令使用记录之特殊符号变量

使用语法 set histchars = "!^"

关于 history 功能的特殊符号有二个，系统设定值分别为“!”及“^”。利用本变量则可改变其特殊符号的原始设定值。

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

经过以上设定后，符号“#”取代“!”；符号“/”取代“^”。这将改变你使用 history 的习惯，如下所示：

```
% set _test = "test aa"
```

如果你要修改上个指令的 aa 变为 bb，你必须使用符号“/”，来做修改，如下：

```
% /aa/bb/
```

savehist 指令使用记录文件储存变量

使用语法

```
set savehist = 20
```

此变量的作用与 history 变量作用相近。不同处是 history 变量所记录的数据会随 logout 而消失，而本变数则会在 home 录下建一个特殊档案“.history”，供下次 login 时呼叫上回 login 的指令使用记

录；或者是其他的 C shell 使用之。

time 运行时间变数

默认变量 time 与内建指令 time 是息息相关的。本变量是用来定义内建指令 time 的输出格式及设定自动显示 time 信息的基本时间。所以实际上此变量有两组设定的数据，一个是时间，另一个是输出格式。设定的组合如下所示：

设定时间 set time = 时间

设定输出格式 set time = ("输出格式")

完全设定 set time = (时间 "输出格式")

时间的单位一般以秒计算之，假定你设定 CPU time 为 3 秒，则你每个运行时间超过 3 秒的指令，便会自动显示出执行的时间状态，而不需用 time 指令来执行命令。

在输出格式上，总共有十个符号，以下便是每个符号所代表的意义：

%D	占用系统内存的 unshared 平均值（单位：千位）
%E	实际执行的消耗时间，一般又称为 wall clock time（分:秒）
%F	页面错误（page faults）数量。在此我们为你简单地介绍页面错误的产生。在 UNIX 操作系统中，内存的架构是以页 page）为程序移转的处理单位。所以一个程序或指令的执行过程有可能会因为 kernal 分页系统的需要，被多次地加载与载出主存储器。当处理程序找不到某个使用过的页次时，便会产生页面错误。这时便会更新使用过的页次集数据。如果有必要的话，会到硬盘中将需要的页次读入。有就是说产生了越多次的页面错误，就有可能会做较多次的 disk I/O 动作。越是如此则执行的效率就越差。所以你必须留意这个项目所输出的数值。如

	果常常会有很多的页面错误产生的话，有可能表示你的内存已经不够系统使用了，应该做硬件上的投资了。
%I	程序从硬盘所读入的数据量（单位：block）
%K	unshared stack 的平均值（单位：千位）
%M	处理程序执行期间所需的最大内存（physical memory）（单位：千位）
%O	程序输出到硬盘的数据量（单位：block）
%S	kernal process 的 CPU 使用时间（单位：秒）
%U	user's process 的 CPU 使用时间（单位：秒）
%P	系统使用时间的总和与实际消耗时间的百分比，也就是(%S+%U)/%E 的结果。
%W	swaps 的次数
%X	shared 内存的平均值（单位：千位）

这十个符号你可以依照你的需要设定是否要将它输出，以及输出时的格式。以下我们来看一个实例：

% 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

如果你不设定输出格式的话，内建指令 time 会以 C shell 所定义的输出格式来输出信息。这个格式为（"%Uu %Ss %E %P% %X+%D %I+%O %F+%W"）。下面是不经过设定的输出格式：

% time find / -name Csherc –ls

（输出部份省略）

9u 36.1s1:39 38% 0+340k 2859+663io 1250pf+0w

echo 与 verbose 指令显示变量

使用语法

set echo

set verbose

这两个变量上使用上非常相近，而且在使用及功能上也有互补的作用，所以我们放在一起为你介绍。设定 echo 变量的作用是将所“真正执行的指令”显示出来。为何说是真正执行的指令呢？因为在 UNIX 操作系统中有一些“指令”，可能是经过 aliases 功能重新定义过了，所以执行的并不是原来的指令。如果你设定这个变量，你便能很清楚地看到“指令”所执行的内容了。

30 ~ % set echo

31 ~ % cd test

cd

set prompt = ! `dirs`%

dirs

32 ~/test % alias cd

alias cd

cd !*;set prompt = "! `dirs`% "

其实 C shell 本身也提供一个与此功能相同的选项，就是“-x”。

当你下指令来产生一个 subshell 时，如果加上“-x”选项也能得到相同的结果。如下所示：

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

verbose 变量的作用与变量 echo 相近。 verbose 变量所处理的并不是指令，而是变量的显示。如果指令中没牵涉到变量的问题，它只会将所执行的指令原封不变地显示在屏幕上。就算是有 aliases 的情况它也不会有任何的作用。

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

但如果所执行的指令含有变量的话，它的处理情况就和设定 echo 的显示有所不同了。让我们看下面的例子：

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

以上是设定了变量 echo 时所产生的情况。请注意到指令 3 的输出信息的第一行，变量 vb 的内容已被带入。接下来我们再来看设定 verbose 变量的情形：

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

我们清楚地看到 $vb 的内容并没有带入变量中，依然保持了 echo 指令的字符模式“$vb”。这便是两者之间的差异所在。其时设定 verbose 变量就相当于是 C shell 的“-x”选项。

这两个变量的最常用在撰写 C shell 文稿产生错误时，相互配合使用来 debug。一般在指令行模式下比较少有必要设定它。

status 执行状态变量

变量 status 乃是用以显示最近的指令执行状态。执行成功状态为 0；失败状态为 1。本变量常运用于 shell 程序设计中判断上一个指令的执行情况。譬如用指令 grep 在数据文件中找寻某个字符串的状态。

```
% 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 目前工作目录变量

cwd 变量所代表的便是目前的工作目录。如果我们用指令 echo 来显示该变量的内容，其实它和指令 pwd 的效果相当接近。如下：

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

一般最常应用此变量来设定题词（prompt）的显示内容，或者是运用在 C shell 文稿内显示目前所在的工作目录。让我们来设定一个可以显示目前工做目录的题词，如下：

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

如上所示，在题词中我们就可以得到工作目录的信息、而不再需要以指令 pwd 来得到此信息。不过这种设定方法并不会再你更改工作目录时，自动地更改题词的数据。所以最好的设定方式还是以下的方式：

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

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

我们使用 aliases 的功能重新设定 cd 的功能，让 cd 不但只执行原来的 cd 指令而且还重新设定一次 prompt 变量的内容。如此便能在你每次更改目录时将目前的工作目录反应在题词中了。也许你会问，为何不用指令 pwd 而要用变量 cwd 呢？其时这是执行效率的问题。因为指令 pwd 并不是内建指令，必须做 I/O 动作，而使用 C shell 变量则不需要这个动作。所以使用变量 cwd 是比较好的选择。

hardpaths 实体路径变量

此变量 hardpaths 和变量 cwd 极为相似，两者都与显示工作目录的变量，然而 hardpaths 变量可以说是针对修正变量 cwd 在显示上的一项错误，所产生的一个比较特殊的变量。这项错误产生在 UNIX 操作系统的一项重要的功能上，就是文件系统的目录链接（symbolic links 或称为 soft links）。这是因为这项技术发展于 C shell 定义变量 cwd 之后的缘故。首先让我们延用上例的提词设定来看下面的例子：

```
/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
```

在上例中我们将“/usr/share/man”以指令 ln 链接到目录“man”，然后我们以指令“ls -l”显示该目录，信息很明显地告诉我们该目录的链接状态。当我们以指令 cd 到该目录之下时发现题词所显示的工作目录是“/home1/akira/man”，但指令 pwd 的结果却告诉我们目前的工作目录是“/usr/share/man”。这便是变量 cwd 无法应付这种 symbolic links 所造成的错误。如果你的 UNIX 操作系统版本有这样的问题，你便可设定这个 hardpaths 变量除这个“bug”。使用语法如下：

```
% set hardpaths
%
```

当然它最好也是设定在“.cshrc”档案中。

因为启动这项功能的作用与会使用到 directory stack 的内建指令 pushd 及 popd 有相当大的关系。这就是为何要修改这项 bug 的最主要原因。

ignoreeof 忽略使用 eof 退出变量

使用语法 set ignoreeof

在一般的情况下，如果我们在 C shell 中使用 CTRL-d，所造成的结果便是将该 C shell 终结，如果在 login shell 中使用则会退出系统。这样难免会有误动作的情况产生。想要避免这种不幸的错误，你可以设定 ignoreeof 变量，如此便不会因为误用 CTRL-d 而退出系统了。如下所示；

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

如果你曾有使用 CTRL-d 造成误动作而退出系统的记录的话，建议你好设定此变量。我想应该可以减少一些使用习惯所造成的麻烦。

noclobber 禁止覆写变数

这个 noclobber 变量我们在前面的输出重导向章节中已经提过，它的功能便是停止重导向符号“>”的覆写（overwiting）已存在档案以及符号“>>”要将数据写入一个不存在的档案时，自动产生该档案的特性。我想在此便不再赘述，仅用两个例子让读者回忆一下，设定

后的实际使用状况。

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 变数

设定这变量 noglob 的作用是停止 wildcard 功能，也就是说像符号 * ? [ ] { } ~ 等等，它们所代表的特殊作用都将失去效用。而仅仅只是代表一般的字符而已。如下面的例子所示：

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

看到没，在设定完变量 noglob 后，代表 home 目录的“~”与符号“*”等均失去其原有的特殊效用。所以要使用这个变量请务必了解自己在做什么！否则你会以为计算机坏了？

建议您如果需要将整个 wildcard 功能暂时停用时再手动设定这个变量是最好的使用方式。如果只是二、三行指令的话我建议使用倒斜线“\”来暂时消除特殊符号的功能。这个方法同样可行。如果选择设定 noglob 变量的话，别忘了不用时您只要 unset noglob 便可以回复到设定前的使用模式了。

有时候在我们撰写 C shell 文稿会因为要常常需要将特殊符号当成一般符号使用，您可以设定这项变量将终止 wildcard 的功能，关于这点我们在下一章中再为你举例说明之。

nonomatch 变数

在我们使用 C shell 的一般的情况下，大都不会放弃使用 wildcard 功能，因为它实在带给我们相当多的便利。不过使用这项功能对某些指令的执行会造成一些负面的影响，让我们来看几个指令执行的错误例子：

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

我们看到指令“rm *.tmp core”在执行第一个自变量“*.tmp”产生错误，因为工作目录下并没有这类档名的档案存在。不过指令 rm 却会因为这项错误而终止执行下一个自变量“core”。所以当我们在一次以指令 ls 来查看执行状况时，你会惊讶地发现档案“core”并没有被清除。这或许在指令行模式下会被我们发现，但如果是在 C shell 文稿中，这可就是个严重的 bug 了。

在使用 C shell 时要消除这类因使用 wildcard 所产生的问题，有两种方式。一是设定变量 noglob 干脆放弃使用 wildcard 功能。另一个便是设定 nonomatch 变量，它可以消除这类问题，同时你也能继续保有使用 wildcard 的功能。

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

我们可以看到在设定 nonomatch 变量后，档案“core”被清除了。

当然在设定 nonomatch 变量后，因为指令执行上也的改变，也有一种情况会与未设定前不同，那就是指令的执行状态也改变了。见下面说明：

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

在设定 nonomatch 变量之前，指令 ls 如果无法找到合符条件的档案，它的指令执行状态是失败的，所以为“1”。但是设定了 nonomatch 变量之后，指令 ls 在相同的情况下，指令的执行状态则是成功的，所以为“0”。这可是相当大的变化，请读者在使用上也最

好注意到这种设定后的变化情况。万一你要使用指令的执行状态来作为判断依据时，可得小心！免得因为这个变化，使程序多出一个 bug。

notify 变数

变量 notify 与工作控制（Job control）的背景工作显示有着密切的关系。由于这个变量的运用不太适合片段式的介绍，所以我把它和工作控制的介绍一起放在第三章中了。简单的说：它的功能便是会将背景工作的处理完成信息插播到前景工作中。详细请参考第三章的工作控制。

filec 文件名自动续接变量

设定 filec 变量对于冗长的文件名或者是非常特别“懒”的人帮助是相当大的。它的使用方式是你在指令行模下先键入该档名的前几个字母，然后按 ESC 键，C shell 便会帮助你将符合的档名自动地补上。让我们来看下面这个例子：

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

当指令 4 文件名键入到一半时，你按下 ESC 键，档名便自动接上。如下：

4 % cc screenprint.c

方便吧！当然啦它有会有不灵光的时候，比方说键入的前缀部份，经过 C shell 侦测发现，符合条件的档案超过一个的时候， C shell 便会发出“哔”声警告你。在这种情况下，所键入的指令行便不做任何改变，保持原来的情况等待你继续再键入数据。如下例的情况：

5 % cc echoESC （发出“哔”声警告）

另外它还会有一个功能，就是你可以键入档名的前面的几个字母，然后使用“CTRL-d”来显示出符合该条件的档案。请见下面的例子：

6 % cc echoCTRL-d

echoerr.c echoout.c

6 % cc echo

在显示出相关的文件名之后，回复到原来的指令行状态等待你输入。感觉怎样！用用看，你一定会喜欢的。如果你对于这样的功能觉得很适用的话，你可以在“.cshrc”档案中设定它，或者是要使用时才在指令行中设定也可以。

fignore 变数

这个变量 fignore 是配合变量 filec 使用的。对于在上面的 filec

变量所支持的功能，在 UNIX 中称为“filename completion”，该功能的作法就是利用键入的前缀部份，由 C shell 自动判断并补上完整的文件名。 filec 变量便是设定启用“filename completion”。而变量 fignore 则是设定该功能忽略掉某种档案的尾名。我们来看下面的例子：

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 之后的情况）

如果我们没有设定变量 fignore 的话，指令 4 会因为有三个档案符合条件，而产生哔声警告我们。但因为已经设定了忽略“.o”及“.out”档名，所以只剩下一个档案符合条件，于是便自动将符合条件的文件名字补上了。这便是设定后所产生的功能变化。不过设定此变量并不会对 CTRL-d 的显示造成任何的影响，这点请读者注意。

nobeep 不准叫变数

设定这个变量的作用是取消“哔”的警告声。使用方法如下：

% set nobeep

讨厌声音的人可以设定此变量叫计算机“闭嘴”，以前在深夜无人的办公室内打计算机的我，一定会设定这个变量，以免吓到警卫伯伯。

设定 C Shell 的使用环境

对一个正常的开放操作系统，一个 UNIX 的使用者绝对有足够的权限来为自己设定与管理自己的 C Shell 使用环境。(我所谓的正常的情况指的是系统管理者不做特殊限制，本篇中所有的例子我都先假设上述的条件成立。)所以，使用者必须要有能力管理与设定自己的 C Shell 使用环境；系统管理者也应视教育使用者管理自己的使用环境为自身应尽的职责。如果你身处于上述的工作环境，请在心中感谢系统管理者，并激励自己多点时间多做点自发性的学习。如果没有如此的环境，那就自力救济吧！

让我们回到本章的主题，UNIX 系统让用户在 login 的过程中可以设定自己想要的使用环境（如果你清楚地知道自己想要做什么！)。而所谓的『使用环境』包含了硬件（如屏幕、键盘、鼠标等）与 shell 的使用环境（第四篇的种种变量设定）两大部份。 Shell 的启始档案（startup file）便是肩负的这项基本而重要的环境设定的『特殊档案』。

在第二篇中，曾经就 C shell 的启始档案做过简要的功能描述，所以不打算在此重述之（有必要参考的网友，请按这里)。

一般而言，使用者个人的启始档案必须放在自己的 home 目录下，如果你用指令 ls 看不到的话请不用惊讶，因为我们前面已经提过它们是『特殊档案』，你必须使用指令 ls -a 才会显示出来。如果还看不到，这也用不着奇怪，这可能是因为你的系统管理者为你建立 account 时忘了帮你拷贝的缘故吧。如果您用了 UNIX 操作系统已有

一段时日，到今天您才发觉到您根本没有自己的“.cshrc”及“.login”启始档案的话，你一定会质疑它们必须存在的必要性与重要性？因为过去你没有这些档案还不是用的好好的！事实上并非如此，因为当你个人的 home 目录下没有这些启始档案时， shell 依旧是必须去读取系统为你准备的原始的启始档案，所以说如果你没有这些启始档案，你可以在 UNIX 的文件系统找到系统原始的启始档案，以 Sun OS 4.1.X 而言档案的位于 /usr/lib 目录内，文件名是 Cshrc 及 Login。你可以将它们拷贝一份到你的 home 目录下作为参考，以便于你在学习中做为设定的模板。（在此说明，以后我们所讨论的启始档案都将是以你的 home 目录下的必须有这些启始档案为前提。）