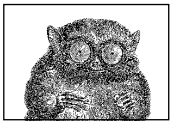

第五章

C shell

本章介绍C shell，之所以如此命名，是因为它很多的编程结构与符号和C编程语言相似。其中包括以下内容：

- 功能概述
- 语法
- 变量
- 表达式
- 命令历史
- 作业控制
- 调用 shell
- 内置的C shell命令

要想得到有关C shell的更多信息，可以阅读在参考文献中列出的《Using csh & tcsh》。

功能概述

C shell具备以下功能：

- 输入输出重定向
- 用于文件名缩写的通配符（元字符）

- 定制用户环境的 shell 变量
- 整数运算
- 访问以前的命令（命令历史）
- 命令名缩写（别名）
- 用于写 shell 程序的内置命令集
- 作业控制
- 文件名完成（可选）

语法

本部分介绍了针对 C shell 的很多符号。包括以下内容：

- 特殊文件
- 文件名元字符
- 引用
- 命令方式
- 重定向方式

特殊文件

`~/.cshrc` 在 shell 调用每一个实例时执行。

`~/.login` 在 `.cshrc` 执行之后由登录 shell 执行。

`~/.logout` 在退出时由登录 shell 执行。

`~/.history` 来自以前的登录中存储的历史列表。

`/etc/passwd` `~`*name* 缩写的主目录的来源（可能来自 NIS 或 NIS+）。

文件名元字符

元字符	描述
*	匹配任何有 0 个或多个字符的字符串
?	匹配任何单个字符
[*abc*...]	匹配被括起来的字符中的任何一个；可以用连字符指定一个范围（例如，a-z、A-Z、0-9）

元字符	描述
{*abc*,*xxx*,...}	扩展括号中的每一个由逗号分隔的字符串。这些字符串不一定匹配实际的文件名
~	当前用户的主目录
~*name*	用户 *name* 的主目录

示例

```
% ls new*          匹配 new、new.1 等
% cat ch?          匹配 ch9 等，但不匹配 ch10
% vi [D-R]*        匹配文件名开头是大写 D 到大写 R 的文件
% ls {ch,app}?     先进行扩展，然后匹配 ch1、ch2、app1、app2 等
% mv info{,.old}   扩展成 mv info info.old
% cd ~tom          将目录修改为用户 tom 的主目录
```

引用

引用可以禁止字符的特殊意义，使字符按其本来意思加以使用，下表中的字符在 C shell 中有特殊的意义。

字符	意义
;	命令分隔符
&	后台执行
( )	命令分组
|	管道
* ? [ ] ~	文件名元字符
{ }	字符串扩展字符，通常不要求引用
< > & !	重定向符号
! ^	历史替代，快速替代
" ' \	用于引用其他字符
`	命令替代
$	变量替代
space tab newline	单词之间的间隔

下列的字符用于引用：

"" 位于 " 和 " 之间的所有的字符都按其字面意义加以采用，下面这些具有特殊意义的字符除外：

`$` 产生变量替代。

`` ` `` 产生命令替代。

`"` 表示双引号的结束。

`\` 转义下一个字符。

`!` 历史字符。

`newline`

换行字符。

`'...'` 除!(历史）和另一个 ' 以及换行符之外，位于 ' 和 ' 之间的所有字符都按其字面意义加以采用。

`\` 在其后的字符会按字面意义采用。通常用于 `"..."` 中以转义 `"`、`$`、`` ` `` 和换行符。用于 `'...'` 内可转义换行符。经常用来转义一个历史字符（通常是!)。

示例

```
% echo 'Single quotes "protect" double quotes'
Single quotes "protect" double quotes

% echo "Don't double quotes protect single quotes too?"
Don't double quotes protect single quotes too?

% echo "You have `ls|wc -l` files in `pwd`"
You have 43 files in /home/bob

% echo The value of \$x is $x
The value of $x is 100
```

命令方式

cmd `&`	在后台执行 *cmd*。
cmd1 `;` *cmd2*	命令序列，在同一行执行多个命令。
`(`*cmd1* `;` *cmd2*`)`	子 shell，将 *cmd1* 和 *cmd2* 视为一个命令组。
cmd1 `|` *cmd2*	管道，用 *cmd1* 的输出作为 *cmd2* 的输入。
cmd1 `` `cmd2` ``	命令替代，用 *cmd2* 的输出作为 *cmd1* 的参数。
cmd1 `&&` *cmd2*	逻辑与关系，执行 *cmd1*（如果 *cmd1* 执行成功）后再执行 *cmd2*。这是一种短路操作，如果 *cmd1* 没有成功执行，*cmd2* 将永远不能执行。

cmd1 || *cmd2*　逻辑或关系，执行 *cmd1* 或（如果 *cmd1* 执行失败）*cmd2*。这是一个短路操作，如果 *cmd1* 执行成功，*cmd2* 将永远不能执行。

示例

```
% nroff file > file.out &              后台格式化
% cd; ls                               顺序执行
% (date; who; pwd) > logfile           重定向所有输出
% sort file | pr -3 | lp               先对文件排序，再分页输出，然后打印
% vi `grep -l ifdef *.c`               编辑 grep 找到的文件
% egrep '(yes|no)' `cat list`          指定一个搜索文件列表
% grep XX file && lp file              如果包含了该模式，则打印文件
% grep XX file || echo XX not found    否则，回显一个错误消息
```

重定向方式

文件描述符	名称	一般缩写	默认
0	标准输入	stdin	键盘
1	标准输出	stdout	终端
2	标准错误	stderr	终端

重定向方式可以改变一般的输入源和输出目标，参见下面的内容。

简单重定向

cmd > *file*

将 *cmd* 的输出发送到文件 *file* 中（覆盖）。

cmd >! *file*

和前一个命令意义相同，另外，还会忽略 noclobber 选项。

cmd >> *file*

将 *cmd* 的输出发送到文件 *file* 中（追加）。

cmd >>! *file*

和前一个命令意义相同，但写文件的时候忽略 noclobber 选项。

cmd < *file*

cmd 从文件 *file* 中获取输入 。

cmd << *text*

读取标准输入，直到遇到一个和 *text* 相等的行（*text* 可以存储在一个 shell 变量中）。输入通常使用终端键入或存储在shell程序中。使用这类语法的命令通常有

cat、echo、ex 和 sed。如果文本 *text* 被引起来（用任何 shell 引用机制），则该输入会按其原来的意思逐字地通过。

多重重定向

cmd >& *file*	将标准输出和标准错误发送到 *file* 中。
cmd >&! *file*	和上条命令相同，但忽略 noclobber 的设置。
cmd >>& *file*	将标准输出和标准错误添加到 *file* 的结尾。
cmd >>&! *file*	和上条命令相同，只是在追加或创建 *file* 时忽略 noclobber 选项。
cmd1 |& *cmd2*	标准错误和标准输出一起执行管道功能 。
(*cmd* > *f1*) >& *f2*	将标准输出发送到文件 *f1* 中；将标准错误发送到文件 *f2* 中。
cmd | tee *files*	将 *cmd* 的输出发送到标准输出（通常是终端）和 *files* 中（参见第二章中 **tee** 下面的示例）。

示例

```
% cat part1 > book
% cat part2 part3 >> book
% mail tim < report
% cc calc.c >& error_out
% cc newcalc.c >&! error_out
% grep Unix ch* |& pr
% (find / -print > filelist) >& no_access
```

```
% sed 's/^/XX /g' << "END_ARCHIVE"
This is often how a shell archive is "wrapped"
bundling text for distribution. You would normally
run sed from a shell program, not from the command line.
"END_ARCHIVE"
XX This is often how a shell archive is "wrapped",
XX bundling text for distribution. You would normally
XX run sed from a shell program, not from the command line.
```

变量

本部分介绍以下内容：

- 变量替代
- 变量修饰符

- 预定义的 shell 变量
- `.cshrc` 文件示例
- 环境变量

变量替代

在下列替代中，大括号(`{}`)是可选的，除非必须用它来将变量名与后面的字符分开。

`${`*`var`*`}`	变量 *var* 的值。
`${`*`var`*`[i]}`	在 *var* 的 *i* 位置处选择 1 个或几个单词。*i* 可能是一个单个的数，也可能是范围 *m*-*n*、范围 -*n*（缺 *m* 暗示为 1）、范围 *m*-（缺 *n* 暗示其余的所有单词）中的数，或者是 `*`（选择所有单词）。*i* 也可以是扩展成此类值之一的一个变量。
`${#`*`var`*`}`	*var* 中单词的个数。
`${#`*`argv`*`}`	参数的个数。
`$0`	程序的名称（通常不会在交互式 shell 中设定）。
`${argv[n]}`	命令行上单个的参数（位置参数）。*n* =1~9。
`${n}`	与 `${argv[n]}`作用相同。
`${argv[*]}`	命令行上所有的参数。
`$*`	与 `$argv[*]`作用相同。
`$argv[$#argv]`	最后一个参数。
`${?`*`var`*`}`	如果设置了 *var*，则返回 1；否则返回 0。
`$$`	当前 shell 的进程号；在创建具有惟一名字的临时文件时，作为文件名的一部分是有用的。
`$?0`	如果知道输入文件名，则返回 1；否则返回 0。
`$<`	从标准输入中读一行。

示例

对第三到最后之间的参数（文件）进行排序，并且将输出存储到一个惟一的临时文件中：

```
sort $argv[3-]>tmp.$$
```

只有当 shell 是交互式时处理 `.cshrc` 命令（即必须设置 `prompt` 变量）：

```
if ($?prompt) then
   set commands,
   alias commands,
   etc.
endif
```

变量修饰符

除 `$?var`、`$$`、`$?0` 和 `$<` 之外，前面进行变量替代时，后面都可能跟着下面的修饰符之一。如果使用括号，修饰符在它的里面。

`:r` 返回变量的根。

`:e` 返回变量的扩展。

`:h` 返回变量的头。

`:t` 返回变量的尾。

`:gr` 返回所有的根。

`:ge` 返回所有的扩展。

`:gh` 返回所有的头。

`:gt` 返回所有的尾。

`:q` 将一个单词列表变量引起来，将其各部分分隔开。当变量中包含不应进行扩展的文件名元字符时，该选项比较有用。

`:x` 将一个模式引起来，并将它扩展成单词列表。

使用路径名修饰符的示例

下表展示了有关下面变量路径名修饰符的用法：

```
set aa=(/progs/num.c /book/chap.ps)
```

变量部分	格式	输出结果
正常变量	`echo $aa`	`/progs/num.c /book/chap.ps`
第二个根	`echo $aa[2]:r`	`/book/chap`
第二个头	`echo $aa[2]:h`	`/book`
第二个尾	`echo $aa[2]:t`	`chap.ps`

变量部分	格式	输出结果
第二个扩展	`echo $aa[2]:e`	`ps`
根	`echo $aa:r`	`/progs/num /book/chap.ps`
全部根	`echo $aa:gr`	`/progs/num /book/chap`
头	`echo $aa:h`	`/progs /book/chap.ps`
全部头	`echo $aa:gh`	`/progs /book`
尾	`echo $aa:t`	`num.c /book/chap.ps`
全部尾	`echo $aa:gt`	`num.c chap.ps`
扩展	`echo $aa:e`	`c /book/chap.ps`
全部扩展	`echo $aa:ge`	`c ps`

使用引用修饰符的示例

```
% set a="[a-z]*" A="[A-Z]*"
% echo "$a" "$A"
[a-z]* [A-Z]*

% echo $a $A
at cc m4 Book Doc

% echo $a:x $A
[a-z]* Book Doc

% set d=($a:q $A:q)
% echo $d
at cc m4 Book Doc

% echo $d:q
[a-z]* [A-Z]*

% echo $d[1] +++ $d[2]
at cc m4 +++ Book Doc

% echo $d[1]:q
[a-z]*
```

预定义的 shell 变量

变量可以按两种方式来设置，第一种方式是分配一个值：

```
set var=value
```

第二种方式是仅仅打开它：

```
set var
```

下表展示了变量的赋值是通过一个等号（该等号后是变量接受的值的类型）来完成的，然后再对值进行描述。（注意，无论如何都不能对 argv、cwd 或 status 之类的变量进行显式赋值。）对于那些只能打开或关闭的变量来说，该表也相应介绍了在设置它们时所做的工作。C shell 会自动设置下面一些变量：argv、cwd、home、path、promt、shell、status、term 和 user。

变量	描述
argv=(*args*)	传递给当前命令的参数列表，默认是()
cdpath=(*dirs*)	当给 cd、popd 或 pushd 定位参数时，轮流搜索的目录列表
cwd=*dir*	当前目录的完整路径名
echo	在执行前重新显示每个命令行，和 csh -x 命令相同
fignore=(*chars*)	在完成文件名时要忽略的文件名后缀列表（参见 filec）
filec	如果设置了它，当按下 Escape 键的时候，在命令行上输入的部分文件名可以扩展成其完整的名称。如果匹配不止一个文件名，则键入 *EOF* 会列出可能的完全文件名
hardpaths	告诉 dirs 要显示那些是一个符号链接的目录的实际路径名
histchars=ab	一个两字符的字符串，用来设置用于历史替代和快速替代中的字符（默认是!^）
history=*n*	要存储到历史列表中的命令的个数
home=*dir*	用户的主目录，从 HOME 中初始化。~ 字符是该值的缩写形式
ignoreeof	忽略来自终端的文件结束标记（*EOF*），阻止意外退出
mail=(*n file*)	每 5 分钟或（如果指定 *n*）每 *n* 秒钟对一个或多个文件进行新邮件检查
nobeep	对不明确的文件完整性操作不振铃
noclobber	不重定向输出到一个已存在的文件，用来防止对文件的意外毁坏
noglob	关闭文件名扩展，用于 shell 脚本中
nonomatch	将文件名元字符当作字面上的字符来对待。例如，vi ch* 是创建新文件 ch* 而不是输出“ No match ”
notify	立刻通知已完成作业的用户，而不用等待下一个提示符
path=(*dirs*)	列出搜索执行命令的路径名。可用 PATH 初始化。SVR4 默认为（./usr/ucb/usr/bin）。在 Solaris 上，默认路径为（/usr/bin.）。然后标准启动脚本再将它修改为（/bin /usr/bin /usr/ucb/ 等等）

变量	描述
prompt='*str*'	提示用于交互式输入的字符串，默认是 %
savehist=*n*	退出时存储在~/.history目录中的历史命令的个数，在下一次登录时可以访问到它
shell=*file*	当前使用的 shell 程序的路径名，默认是 /bin/csh
status=*n*	最后命令的退出状态。成功时，内置命令返回 0；若失败，则返回 1
term=*ID*	终端类型的名称，和 TERM 一样
time='*n*%*c*'	如果命令执行时间超过 *n* 个 CPU 秒数，则报告用户时间、系统时间、消耗时间和 CPU 占有率。支持可选的 %*c* 标志来显示其他数据
user=*name*	用户的登录名称，在 USER 中初始化
verbose	进行历史替代之后，显示一个命令，和命令 csh -v 作用相同

.csbrc 文件示例

```
# 预定义的变量
set path=(~ ~/bin /usr/ucb /bin /usr/bin . )
set mail=(/var/mail/tom)

if ($?prompt) then              # 为交互式使用而设置
  set echo
  set filec
  set noclobber ignoreeof

  set cdpath=(/usr/lib /var/spool/uucp)
 # 现在可以键入cd宏
 # 代替cd /usr/lib/macros

  set fignore=.o                # 对filec忽略目标文件
  set history=100 savehist=25
  set prompt='tom \!% '         # 包括历史号
  set time=3

 # MY VARIABLES

  set man1="/usr/man/man1"      # 允许我执行cd $man1、ls $man1
  set a="[a-z]*"                # 允许我执行vi $a
  set A="[A-Z]*"                # 或grep string $A

 # ALIASES

  alias c "clear; dirs"         # 使用引号保护;或|
  alias h "history | more"
  alias j jobs -l
  alias ls ls -sFC              # 重新定义ls命令
```

```
  alias del 'mv \!* ~/tmp_dir'   # rm的安全替代
endif
```

环境变量

C shell维护了一系列环境变量（environment variable），它们明确地区别于shell变量，并且不是C shell的部分。shell变量只是在当前shell内部有意义，而环境变量则会自动输出，从而使环境变量成为全局变量。例如，C shell变量只在定义它们的特定脚本中才是可访问的，而环境变量可以应用到任何shell脚本、mail实用程序或用户调用的编辑器中。

可按下列方式给环境变量赋值：

```
setenv VAR value
```

按惯例，环境变量名都是大写的。你可以创建自己的环境变量，也可使用下面预定义的环境变量。

这些环境变量有一些相应的C shell变量，如下所示：

HOME
: 主目录，和home作用相同。可以彼此无关地进行修改。

PATH
: 搜索命令的路径，和path作用相同。改变任意一个路径则会修改存储在另一个中的值。

TERM
: 终端类型，和term作用相同。改变term可修改TERM，但反之则不行。

USER
: 用户名，和user作用相同。改变user可修改USER，但反之则不行。

其他环境变量包括：

EXINIT
: 类似于在.exrc启动文件中的字符串（例如，set ai），是一个ex命令的字符串。由vi和ex使用。

LOGNAME
: USER变量的另一个名字。

MAIL

拥有邮件的文件。由邮件程序使用。它与C shell中只用于检查新邮件的`mail`变量不一样。

PWD

当前目录，该值从`cwd`拷贝而来。

SHELL

默认情况下未定义，一旦初始化到shell，两者是一样的。

表达式

表达式用于在`@`（C shell数学运算符）、`if`和`while`语句中执行算术运算、字符串比较、文件测试等。`exit`和`set`也能指定表达式。表达式由运算符将变量和常量组合构成，就如同在C编程语言中一样。运算符优先权与在C中一样。记住下面的优先权规则是非常容易的：

- `* / %`
- `+ -`
- 将所有在`()`里的其他表达式分组，如果表达式中包括`<`、`>`、`&`或`|`，则圆括号是必需的

运算符

运算符可以是下面类型之一。

赋值运算符

运算符	描述
=	赋值
+= -=	在加/减后重新赋值
*= /= %=	在乘/除/求余后重新赋值
&= ^= |=	在对位进行AND/XOR/OR后重新赋值
++	递增
--	递减

算术运算符

运算符	描述
* / %	乘；整除；求余
+ -	加；减

位和逻辑运算符

运算符	描述
~	二进制求反（二进制反码）
!	逻辑非
<< >>	位左移；位右移
&	按位与
^	按位异或
|	按位或
&&	逻辑与（短路操作）
||	逻辑或（短路操作）
{command}	如果命令成功，则返回1；否则返回0。注意，这个情况和*command*的正常返回码是不一样的。$status变量可能更实用一些

比较运算符

运算符	描述
== !=	相等；不等
<= >=	小于或等于；大于或等于
< >	小于；大于
=~	左边字符串匹配一个包含*、?或[...]的文件名模式
!~	左边字符串不匹配一个包括*、?或[...]的文件模式

文件查询操作符

在执行检测之前对文件*file*执行命令替代和文件名扩展。

操作符	描述
`-d file`	该文件是一个目录
`-e file`	该文件存在
`-f file`	该文件是一个平常的文件
`-o file`	用户拥有该文件
`-r file`	用户有读权限
`-w file`	用户有写权限
`-x file`	用户有执行权限
`-z file`	文件长度为 0
`!`	对上面的每个操作符取反

示例

下面的示例介绍了 `@` 命令，并且假定 `n=4`。

表达式	`$x` 的值
`@ x = ($n > 10 || $n < 5)`	1
`@ x = ($n >= 0 && $n < 3)`	0
`@ x = ($n << 2)`	16
`@ x = ($n >> 2)`	1
`@ x = $n % 2`	0
`@ x = $n % 3`	1

下面的示例一般用于 `if` 和 `while` 语句的第一行。

表达式	意义
`while ($#argv != 0)`	当有参数的时候……
`if ($today[1] == "Fri")`	如果第一个词是“ Fri ”……
`if ($file !~ *.[zZ])`	如果文件不是以 `.z` 或 `.Z` 为结尾……
`if ($argv[1] =~ chap?)`	如果第一个参数是 chap 后跟着一个单字符……
`if (-f $argv[1])`	如果第一个参数是一个平常的文件……
`if (! -d $tmpdir)`	如果 `$tmpdir` 不是一个目录……

命令历史

前面执行过的命令存储在一个历史列表中。C shell 可以让你访问该列表，从而可以对命令进行校验，重复执行它们，或对命令修改后再执行。`history`内置命令可显示历史列表，预定义的变量`histchars`、`history`和`savhist`也可以影响历史机制。访问历史列表涉及三件事情：

- 进行命令替代（用 ! 和 ^）
- 进行参数替代（一个命令内的特定单词）
- 用修饰符提取或替换一个命令或单词的部分

命令替代

`!`	开始一个历史替代
`!!`	先前命令
`!`*`N`*	历史表中第 *N* 条命令
`!-`*`N`*	从当前命令开始往后第 *N* 条命令
`!`*`string`*	以 *string* 开始的最近的命令
`!?`*`string`*`?`	包含了 *string* 的最近的命令
`!?`*`string`*`?%`	包含了 *string* 的最近命令参数
`!$`	先前命令的最后一个参数
`!!`*`string`*	上一条命令，然后追加 *string*
`!`*`N string`*	第 *N* 条命令，然后追加 *string*
`!{`*`s1`*`}`*`s2`*	以字符串 *s1* 开始的最近的命令，然后追加字符串 *s2*
`^`*`old`*`^`*`new`*`^`	快速替代，在先前命令中将字符串 *old* 改成 *new*，执行修改后的命令

命令替代示例

事先假定下面的命令：

```
3% vi cprogs/01.c ch002 ch03
```

事件序号	键入的命令	执行的命令
4	^00^0	vi cprogs/01.c ch02 ch03
5	nroff !*	nroff cprogs/01.c ch02 ch03
6	nroff !$	nroff ch03
7	!vi	vi cprogs/01.c ch02 ch03
8	!6	nroff ch03
9	!?01	vi cprogs/01.c ch02 ch03
10	!{nr}.new	nroff ch03.new
11	!!|lp	nroff ch03.new | lp
12	more !?pr?%	more cprogs/01.c

单词替代

单词说明符允许从先前命令行中提取单个的单词。冒号可以放在任何单词说明符的前面。在一个事件序号之后，除非是这里显示的，否则冒号是可选择的。

`:0` 命令名

`:`*n* 参数序号 *n*

`^` 第一个参数

`$` 最后一个参数

`:`*n*`-`*m* 参数 *n* 到 *m*

`-m` 单词 0 到 *m*，和：`0-`*m* 相同

`:n-` 参数 *n* 至最后一个的前一个

`:n*` 参数 *n* 至最后一个，和 *n*`-$` 相同

`*` 所有的参数，和 `^-$` 或 `1-$` 相同

`#` 当前命令行直到这点，很少用

单词替代示例

假定有下列命令：

```
13% cat ch01 ch02 ch03 biblio back
```

事件序号	键入的命令	执行的命令
14	`ls !13^`	`ls ch01`
15	`sort !13:*`	`sort ch01 ch02 ch03 biblio back`
16	`lp !cat:3*`	`lp ch03 biblio back`
17	`!cat:0-3`	`cat ch01 ch02 ch03`
18	`vi !-5:4`	`vi biblio`

历史修饰符

可以通过一个或多个修饰符来修饰命令替代和单词替代：

输出、替代和引用

`:p` 只显示命令但不执行。

`:s/`*`old`*`/`*`new`* 用字符串 *new* 替代 *old*，只影响第一个实例。

`:gs/`*`old`*`/`*`new`* 用字符串 *new* 替代 *old*，适用于所有的实例。

`:&` 重复先前的替代（`:s` 或 `^` 命令），只影响第一个实例。

`:g&` 重复先前的替代，适用于所有实例。

`:q` 将一个单词列表引起来。

`:x` 将分隔的单词引起来。

截取

`:r` 提取第一个可用的路径名的根。

`:gr` 提取所有的路径名的根。

`:e` 提取第一个可用的路径名的扩展名。

`:ge` 提取所有的扩展名。

`:h` 提取第一个可用的路径名的头。

`:gh` 提取所有路径名的头。

`:t` 提取第一个可用的路径名的尾。

`:gt` 提取所有路径名的尾。

历史修饰符的示例

以“单词替代示例”中的第17条命令为例：

```
17% cat ch01 ch02 ch03
```

事件#	键入的命令	执行的命令
19	!17:s/ch/CH/	cat CH01 ch02 ch03
20	!:g&	cat CH01 CH02 CH03
21	!more:p	more cprogs/01.c（只显示）
22	cd !$:h	cd cprogs
23	vi !mo:$:t	vi 01.c
24	grep stdio !$	grep stdio 01.c
25	^stdio^include stdio^:q	grep"include stdio" 01.c
26	nroff !21:t:p	nroff 01.c（是我需要的吗?）
27	!!	nroff 01.c（执行它）

作业控制

作业控制使用户可以将前台作业放到后台，将后台作业转到前台，或挂起（暂时停止）正在运行的作业。C shell对作业控制提供了下面的命令。要想获得这些命令的更多信息，参见本章后面的“内置的C shell命令”。

`bg` 将一个作业放至后台。

`fg` 将一个作业放至前台。

`jobs`
列出活动的作业。

`kill`
终止一个作业。

`notify`
当一个后台作业完成时通知。

`stop`
挂起一个后台作业。

`CTRL-Z`
挂起一个前台作业。

很多作业控制命令采用 *jobID* 作为参数。可以按以下方式指定参数：

`%`*n* 作业号 *n*

`%`*s* 作业的命令行是以字符串 *s* 开始的

`%?`*s* 作业的命令行中包含字符串 *s*

`%%` 当前作业

`%` 当前作业（和上面相同）

`%+` 当前作业（和上面相同）

`%-` 先前的作业

调用 shell

可以按下面的方式来调用 C shell 命令解释器：

```
csh [options] [arguments]
```

`csh` 从一个终端或文件中执行命令。在调试脚本的时候，选项 `-n`、`-v` 和 `-x` 是有用的。

下面对这些选项进行了详细的解释：

`-b` 允许余下的命令行选项作为一个指定命令的选项来解释，而不是作为csh本身的选项。

`-c` 将第一个参数作为一个要执行的命令的字符串。余下的参数通过argv数组可以使用。

`-e` 如果一个命令产生错误，则退出。

`-f` 快速启动，不用执行 `.cshrc` 或 `.login` 就可以启动 `csh`。

`-i` 调用交互式 shell（可提示输入）。

`-n` 解析命令但不执行。

`-s` 从标准输入读命令。

`-t` 执行一个命令后退出。

`-v` 执行命令前显示命令，扩展历史替代但不扩展其他的替代（如文件名、变量和命令）。与设置 `verbose` 作用相同。

`-V` 与设置 `-v` 作用相同，但也显示 `.cshrc`。

`-x` 执行命令之前先显示命令，但扩展所有的替代。与设置 `echo` 作用相同。经常将 `-x` 和 `-v` 结合使用。

`-X` 与 `-x` 作用相同，但也显示 `.cshrc`。

内置的 C shell 命令

#	# 忽略在同一行中跟在它后面的所有文本。shell 脚本把它作为注释符而不是一个真正的命令来使用。另外，一些旧的系统有时会将第一个字符为 # 的文件作为一个 C shell 脚本来解释。
#！	#!*shell* [*option*] 用于一个脚本的第一行来调用指定 shell。该行的其余部分作为单个的参数传递给 shell。该功能一般由内核实现，但一些旧系统可能不支持该功能。一些系统对 shell的最大长度有32个字符的限制。例如： `#!/bin/csh -f`
:	: 空命令（什么也不做）。返回一个退出状态 0。
alias	alias [*name*[*command*]] 指定 *name* 为命令 *command* 的缩写名称或别名。如果省略了参数 *command*，则输出 *name* 别名，如果也省略了 *name*，则输出所有的别名。别名可以在命令行上定义，但更常存储在 `.cshrc` 中，以便在登录之后生效（参见本章前面的“ `.cshrc` 文件示例 ”）。别名定义可以引用命令行参数，与历史列表相似。用 `/!*` 指所有的命令行参数，用 /!^ 指第一个参数，用 `/!$` 指最后一个参数等等。一个别名 *name* 可以是任何有效的 Unix 命令，然而，如果不键入 *name*，则会丧失其最初的意思。参见 **unalias**。 示例 在 X window 系统中设置 `xterm` 窗口的尺寸：

alias

```
alias R 'set noglob; eval `resize`; unset noglob'
```

显示包含了字符串 *ls* 的别名：

```
alias | grep ls
```

在所有的命令行参数上运行 nroff：

```
alias ms 'nroff -ms \!*'
```

拷贝被指定为第一个参数的文件：

```
alias back 'cp \!^ \!^.old'
```

使用常规的 ls，而不是它的别名：

```
% \ls
```

bg

bg [*jobIDs*]

将当前的作业或 *jobIDs* 放至后台。参见前面的“ 作业控制 ”部分。

示例

为了将一个消耗时间的进程放在后台，用户可以使用下面命令：

```
4% nroff -ms report | col > report.txt
CTRL-Z
```

然后使用下面的任何一种方式：

```
5% bg
5% bg %          当前作业
5% bg %1         作业号 1
5% bg %nr        匹配词首字符串 nroff
5% % &
```

break

break

继续执行距离 while 或 foreach 最近的、在 end 命令后的命令。

breaksw

breaksw

终止一个 switch 语句，然后继续执行 endsw 之后的命令。

case	case *pattern* 在 switch 语句中标识模式 pattern。
cd	cd [*dir*] 将工作目录修改为*dir*，默认为用户的主目录。如果*dir*是一个相对的路径名而不是当前目录，则搜索变量 cdpath。参见本章前面的“ .cshrc 文件示例 ”部分。
chdir	chdir [*dir*] 与 cd 命令作用相同。如果希望重新将 cd 定义为别名，则可以使用该命令。
continue	continue 继续执行 while 和 foreach 的下一次迭代循环。
default	default： 规定 switch 语句中默认的情况（通常在最后）。
dirs	dirs[-l] 输出目录堆栈，首先显示当前目录，用 -l 将主目录符号（~）扩展为实际的目录名。参见 **popd** 和 **pushd**。
echo	echo [-n] *string* 将字符串 *string* 写到标准输出中，如果指定了 -n，输出不会被换行符中断。和 Unix 版本及 Bourne shell 版本不一样（/bin/echo），C shell 中的 echo 不支持转义字符。也可以参见第二章和第四章中的 **echo** 命令。
end	end 用于结束 foreach 和 while 语句的保留字。

endif

endif

用于结束 if 语句的保留字。

endsw

endsw

用于结束 switch 语句的保留字。

eval

eval *args*

一般地，eval用于 shell 脚本中，并且 *args* 是一行包含了 shell 变量的代码。eval首先强制变量扩展，然后运行生成的命令。当shell变量包含了输入/输出重定向符号、别名或其他shell变量的时候，这种两次搜索是有用的。(例如，正常情况下重定向发生在变量扩展之前，因此一个包含了重定向符号的变量必须首先用eval扩展，否则会无法解释那个重定向符号。) 可以参见第四章中eval下面的示例。下面也列出了 eval的其他用法。

示例

下面的代码用于 .login 文件中来确定终端属性。

```
set noglob
eval 'tset -s xterm'
unset noglob
```

下面的命令显示了 eval 的用法：

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

% echo $b        读命令行一次
$a
% eval echo $b   读命令行两次
hello
```

exec

exec *command*

执行 *command* 来代替当前shell，这将终止当前的shell，而不是在它下面创建一个新的进程。

exit

`exit [(expr)]`

退出一个shell脚本，退出时带有*expr*指定的状态。零状态意味着成功，非零状态意味着失败。如果没有指定*expr*，则退出值就是`status`变量的值。可以在命令行执行`exit`以关闭一个窗口（退出）。

fg

`fg [`*`jobIDs`*`]`

将当前的作业或*jobIDs*放到前台。可以参见前面的“作业控制”部分。

示例

如果用户挂起了一个`vi`编辑会话（通过按`CTRL-Z`），那么用户就可以用下面命令中的任何一种继续`vi`的执行：

```
8% %
8% fg
8% fg %
8% fg %vi          匹配词首的字符串
```

foreach

```
foreach name (wordlist)
   commands
end
```

将变量*name*赋给*wordlist*中的每个值，并且执行在`foreach`和`end`之间的命令。用户可以将`foreach`作为在C shell 提示符下的多行命令来使用（第一个示例），也可以将它用在一个shell脚本中（第二个示例）。

示例

对所有以大写字母开头的文件重命名：

```
% foreach i ([A-Z]*)
? mv $i $i.new
? end
```

检查每个命令行参数是否是一个选项：

```
foreach arg ($argv)
   # does it begin with - ?
   if ("$arg" =~ -*) then
```

foreach

```
            echo "Argument is an option"
          else
          echo "Argument is a filename"
         endif
      end
```

glob

glob *wordlist*

对 *wordlist* 作文件名替代、变量替代和历史替代。这种扩展更像 echo，只是不能识别\转义，并且单词间是用空字符作为定界符的。glob 通常用于在 shell 脚本中对一个值进行“硬编码”，以使其在脚本的其余部分保持一致。

goto

goto *string*

跳到第一个非空字符是 *string* 且其后跟着一个：的一行，然后继续执行该行下的代码。在 goto 行上，*string* 可能是一个变量或文件名模式，但要分支去的标签必须是一个字面上的、被扩展的值，且不能出现在一个 foreach 或 while 语句的内部。

hashstat

hashstat

显示一个统计，该统计表明了通过 path 变量来定位命令的散列表的成功级别。

history

history [*option*]

显示历史事件的列表（在前面的“命令历史”中曾讨论了历史语法）。

注意：多行复合命令（例如 foreach...end）不存储在历史列表中。

options

-h 不带事件序号输出历史列表。

-r 按倒置顺序输出，最后显示最早的命令。

n 只显示最后的 *n* 条历史命令，而不是由 shell 变量 history 设置的数字。

示例

保存并执行最后 5 条命令：

history

```
history -h 5 > do_it
source do_it
```

if

`if`

开始一个条件语句，简单的格式是：

```
if (expr) cmd
```

下面并排显示了三个可能的其他格式：

```
if (expr) then      if (expr) then      if (expr) then
    cmds                cmds1               cmds1
endif               else                else if (expr)then
                        cmds2               cmds2
                    endif               else
                                            cmds3
                                        endif
```

在最简单的形式中，如果 *expr* 为真，则执行 *cmd*；否则什么也不做（重定向仍旧发生，这是一个 bug）。在其他形式中，执行一个或多个命令。如果 *expr* 为真，则继续执行 `then` 后的命令；如果 *expr* 为假，则分支到 `else`（或 `else if` 之后并且继续检查）后面的命令。更多的示例可以参见前面的“ 表达式 ”，或 **shift** 和 **while**。

示例

如果没有指定命令行参数，则执行一个默认的行为：

```
if ($#argv == 0) then
   echo "No filename given. Sending to Report."
   set outfile = Report
else
   set outfile = $argv[1]
endif
```

jobs

`jobs [-l]`

列出所有正在运行或停止的作业，`-l` 包括进程 ID。例如，用户可以检查长编译或文本格式化是否仍在运行。在退出前也是有用的。

kill

`kill [options] ID`

终止每个指定的进程 *ID* 或作业 *ID*。用户必须拥有该进程，或者是特权用户。这个内置命令与第二章中的 `/usr/bin/kill` 类似，但也允

kill

许符号作业名称。难处理的进程可以用信号9去终止。参见前面的“ 作业控制 ”。

options

`-l` 列出信号名（用于本身）。

`-signal`

信号编号（来自`/usr/include/sys/signal.h`）或信号名（来自`kill -l`）。采用了信号 9 的 kill 是 “ 绝对的 ”。

信号

在`/usr/include/sys/signal.h`中定义了信号, 在这里不带SIG前缀列出了这些信号。在用户的系统中可能有比这更多的信号。

HUP	1	挂起
INT	2	中断
QUIT	3	退出
ILL	4	非法指令
TRAP	5	跟踪陷阱
IOT	6	IOT 指令
EMT	7	EMT 指令
FPE	8	浮点异常
KILL	9	终止
BUS	10	总线错误
SEGV	11	段冲突
SYS	12	系统调用了错误参数
PIPE	13	向管道中写, 但没有进程来读
ALRM	14	时钟报警
TERM	15	软件终止(默认信号)
USR1	16	用户定义信号 1
USR2	17	用户定义信号 2
CLD	18	子进程终止
PWR	19	电源失败后重启

示例

如果你使用了如下命令：

```
44% nroff -ms report > report.txt &
[1] 19536          csh 输出作业和进程 ID
```

你可以使用下面的任何一种方式终止它:

```
45% kill 19536          进程 ID
```

kill

```
45% kill %          当前作业
45% kill %1         作业号 1
45% kill %nr        词首的字符串
45% kill %?report   匹配字符串
```

limit

limit [-h] [*resource* [*limit*]]

显示限制，或对当前进程和它创建的进程使用的资源进行限制。如果不给出 *limit*，则只输出对 *resource* 的当前限制。如果也省略了 *resource*，则输出所有的限制。默认情况下会显示或设置当前的限制。使用 -h 时，则使用硬限制。一个硬限制强加一个不能超越的绝对限制。只有特权用户才可以提高它。参见 **unlimit**。

resource（资源）

cputime
: CPU 能花费的最大秒数，可以缩写为 cpu。

filesize
: 任何一个文件的最大长度。

datasize
: 数据（包括栈）的最大长度。

stacksize
: 栈的最大长度。

coredumpsize
: 核心转储文件的最大长度。

limit（限制）

一个数字后跟着一个可选的字符（单位说明符）。

对于 cputime： *n*h（*n* 小时）
*n*m（*n* 分钟）
mm:ss（分钟和秒）

对于其他的： *n*k（*n* 千字节，默认）
*n*m（*n* 兆字节）

login	`login [`*user*`|-p]` 用 `/bin/login` 代替用户 *user* 的登录 shell。`-p` 保持了环境变量。
logout	`logout` 终止登录的 shell 。
nice	`nice [±`*n*`]` *command* 改变 *command* 的执行优先权；如果什么也没给出，则改变当前 shell 的优先权（参见第二章中的 **nice**），该优先权的范围是 −20 至 20，默认为 4。该范围的优先顺序和我们所想像的相反：−20 给出最高的优先权（最快地执行），20 给出最低的优先权。 `+`*n*　对优先权值加 *n*（降低作业优先权）。 `-`*n*　对优先权值减 *n*（提高作业优先权）。只能由特权用户使用。
nohup	`nohup [`*command*`]` “ 没有挂起的信号 ”。在关闭终端行后（即，当挂断一个电话或退出时）不终止 *command*。在 shell 脚本中使用没有 *command* 的 nohup 命令，以使脚本不被终止（参见第二章中的 **nohup**）。
notify	`notify [`*jobID*`]` 当一个后台作业完成后立即报告（而不是等待用户退出一个长编辑会话）。如果省略了 *jobID*，则假定是当前的后台作业。

onintr	`onintr` *label*
	`onintr -`
	`onintr`
	“ 正在中断 ”。用于在 shell 脚本中处理中断信号（与 Bourne shell 中的 `trap 2` 和 `trap ""2` 命令相似）。第一种形式类似 `goto` *label*，此时如果捕捉到中断信号（例如，`CTRL-C`），则脚本分支至 *label*；第二种形式使脚本忽略中断。在一个其运行不能被干扰的脚本的开头或代码段之前，第二种形式是有用的；第三种形式用于存储由先前的命令 `onintr-` 禁止的中断处理。

onintr

示例

```
onintr cleanup          中断发生时转至“ cleanup ”
 .
 .                      shell 脚本命令
 .
cleanup:                中断标号
  onintr -              忽略相应的中断
  rm -f $tmpfiles       删除任何创建的文件
  exit 2                返回一个错误状态并退出
```

popd

popd [+*n*]

删除目录栈的当前条目，或从栈中删除第 *n* 条。当前条目有序号 0 并显示在左边。可参见 **dirs** 和 **pushed**。

pushd

pushd *name*
pushd +*n*
pushd

第一种形式将工作目录修改为 *name*，并且将其添加到目录栈中；第二种形式将目录的第 *n* 项移动到开始位置，使之成为工作目录（条目序号从 0 开始）。如果不带参数，pushd 会交换目录的前两项，并变成新的当前目录。可以参见 **dirs** 和 **popd**。

示例

```
5% dirs
/home/bob /usr
6% pushd /etc                   将 /etc 添加到目录栈
/etc /home/bob /usr
7% pushd +2                     将目录第三项转变成第一项
/usr /etc /home/bob
8% pushd                        交换目录的前两项
/etc /usr /home/bob
9% popd                         丢弃当前项，并移动到下一项
/usr /home/bob
```

rehash

rehash

重新计算 path 变量的散列表。在当前会话期间无论何时创建一个新命令，都应用 rehash。这可以允许 shell 定位并执行该命令。（如果新命令没有列在 path 目录中，在执行 rehash 前，应将目录添加到 path 中。）可以参见 **unhash** 命令。

repeat

repeat *n command*

执行 *command* 的 *n* 个实例。

示例

通过在一个文件中存储/usr/dict/words的25次拷贝来产生一个测试文件。

```
% repeat 25 cat /usr/dict/words > test_file
```

从终端读 10 行并存储在 item_list 中：

```
% repeat 10 line > item_list
```

向 report 中追加 50 个样本文件：

```
% repeat 50 cat template >> report
```

set

set *variable=value*
set *variable*[*n*]=*value*
set

将变量 *variable* 设置为 *value*；如果指定了多个值，则用值的列表来设置变量。如果给定了下标 *n*，则将变量中的第 *n* 个单词设置为 *value*（变量包含的单词的数量不能少于单词的序号）。如果没有给定参数，则显示所有设置变量的名称和值。参见本章前面的“预定义的 shell 变量”部分。

示例

```
% set list=(yes no maybe)          对一个词的列表赋值
% set list[3]=maybe                对存在的词列表的一项赋值
% set quote="Make my day"          对一个变量赋值
% set x=5 y=10 history=100         对几个变量赋值
% set blank                        对 blank 赋空值
```

setenv

setenv [*name* [*value*]]

对一个环境变量 *name* 赋值 *value*。按惯例，*name* 应用大写。*value* 可能是单个的词或一个引起来的字符串。如果没有给出 *value*，则赋空值。没有给出参数时，显示所有环境变量的名称和值。对 USER、TERM 和 PATH 变量来说，setenv 不是必需的，因为它们会自动地从 user、term 和 path 中输出。可以参见前面的“环境变量”。

shift

shift [*variable*]

假如给出*variable*，则转换一个词列表变量中的词。例如，*name*[2]会变成*name*[1]。如果没有给出参数，则转换位置参数（命令行参数），例如，$2变成$1。在while循环中经常用到shift。可参见**while**下的相应示例。

示例

```
while ($#argv)                  当有参数时
   if (-f $argv[1])
      wc -l $argv[1]
   else
      echo "$argv[1] is not a regular file"
   endif
   shift                        下一个参数
end
```

source

source [-h] *script*

从C shell脚本中读取并且执行命令。如果有-h，则命令会添加到历史列表中，但不会执行。

示例

```
source ~/.cshrc
```

stop

stop [*jobIDs*]

挂起当前的后台作业或由*jobIDs*指定的后台作业，该命令是CTRL-Z或suspend的补充。

suspend

suspend

挂起当前的前台作业，和CTRL-Z相似。经常用于终止su命令。

switch

switch

依靠一个变量的值来处理命令。当用户希望处理三个以上选择的时候，switch是if-then-else语句的最佳替代。如果*string*变量匹配*pattern1*，则执行第一组命令；如果匹配*pattern2*，则执行第二组命令；其他依次类推。如果没有模式匹配，则执行默认情况下的命令。*string*可以由命令替代、变量替代或文件名扩展来指定。模式可以用

switch

模式匹配符号 *、?和[ ]来指定。当执行完 *commands* 之后，可使用 breaksw 来退出 switch 语句。如果省略了 breaksw（很少这样做），则 switch 继续执行另一组命令，直到碰到 breaksw 或 endsw。下面是 switch 的一般语法，同时并列列出了一个示例，该示例用于处理第一个命令行参数。

```
switch (string)      |    switch ($argv[1])
 case pattern1:      |      case -[nN]:
     commands        |          nroff $file | lp
     breaksw         |          breaksw
 case pattern2:      |      case -[Pp]:
     commands        |          pr $file | lp
     breaksw         |          breaksw
 case pattern3:      |      case -[Mm]:
     commands        |          more $file
     breaksw         |          breaksw
     .               |      case -[Ss]:
     .               |          sort $file
     .               |          breaksw
 default:            |      default:
     commands        |          echo "Error-no such option"
                     |          exit 1
     breaksw         |          breaksw
endsw                |      endsw
```

time

time [*command*]

执行 *command* 并且显示它花费了多少时间。如果没有给出参数，则 time 用于 shell 脚本中，并对它进行计时。

umask

umask [*nnn*]

显示文件创建掩码，或将文件创建掩码设为八进制 *nnn*。文件创建掩码决定关闭哪一个权限位。参见第二章该项的示例。

unalias

unalias *name*

从别名列表中删除 *name*。要想获得更多信息，可参见 **alias**。

unhash

unhash

删除内部的散列表。C shell 停止使用被散列的值，并且通过搜索 path 中的目录来定位一个命令。可参见 **rehash**。

unlimit

`unlimit [`*`resource`*`]`

消除对资源*resource*的限制。如果没有给出*resource*，则取消对所有资源的限制。要想获得更多的信息，可以参见 **limit**。

unset

`unset` *`variables`*

删除一个或多个变量。变量名可以用文件名元字符来指定为一个模式。参见 **set**。

unsetenv

`unsetenv` *`variable`*

删除一个环境变量。文件名匹配无效。参见 **setenv**。

wait

`wait`

暂停执行，直到所有后台作业完成或接收到一个中断信号。

while

```
while (expression)
   commands
end
```

只要表达式为真（计算为非零），则执行 `while` 和 `end` 之间的 *commands*。`break` 和 `continue` 可以中断或继续该循环。可以参见 **shift** 下的示例。

示例

```
set user = (alice bob carol ted)
while ($argv[1] != $user[1])
  检查匹配以循环通过每一个用户
   shift user
  如果在循环通过时没有匹配…
   if ($#user == 0) then
     echo "$argv[1] is not on the list of users"
     exit 1
   endif
end
```

@	@ *variable=expression* @ *variable*[*n*]=*expression* @ 将运算表达式 *expression* 的值赋给 *variable*；如果指定了下标 *n*，则对变量的第 *n* 个元素赋值；如果既没有指定 *variable* 也没有给出 *expression*，则输出所有 shell 变量的值（和 set 相同）。表达式运算符和示例都在前面的“ 表达式 ”一节列出。下面两个特殊的形式也是有效的： @ *variable++* 变量加一。 @ *variable--* 变量减一。