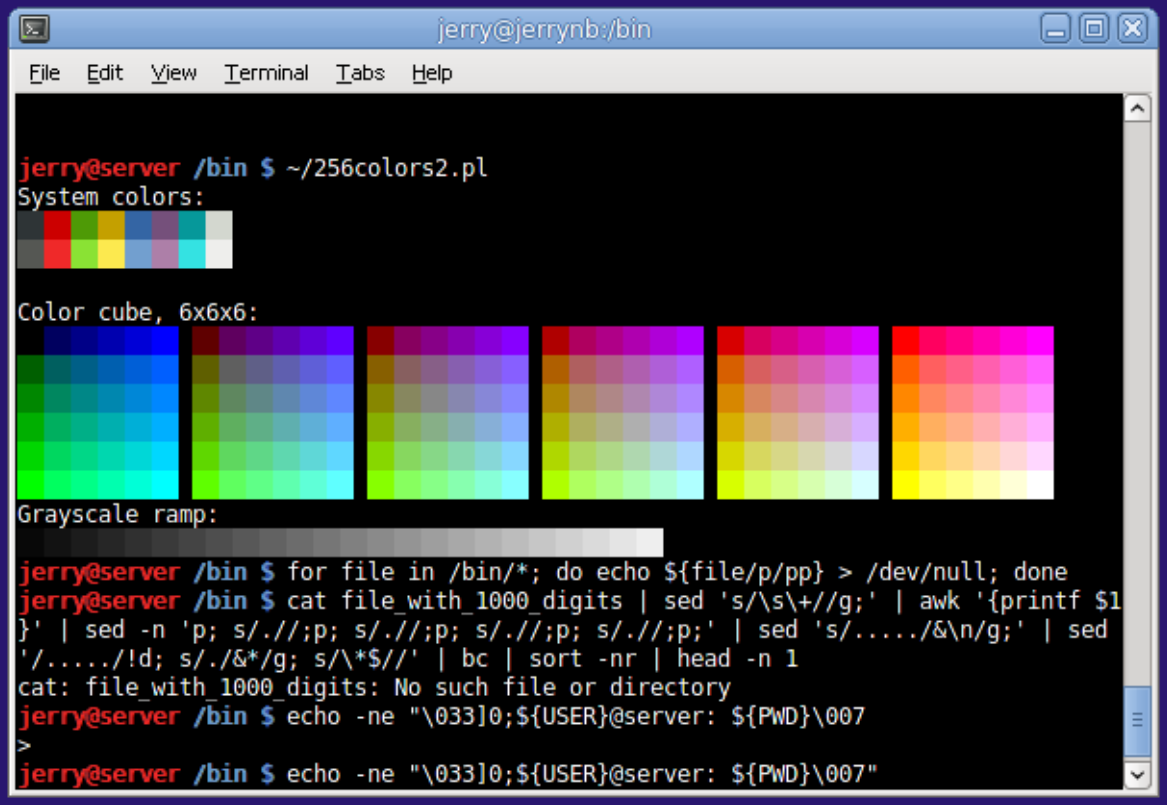
最权威的 BASH 官方文档

- 最翔实的参考
- 最权威的描述
- 最新颖的内容



BASH

中文文档

Chet Ramey / Brain Fox

邵加超(*Jerry Fleming*) 译注

BASH 中文文档

Chet Ramey, Brian Fox 著
邵加超(Jerry Fleming) 译注

最后更新于 2009-10-19

Bash 4.0 参考文档

jerryfleming2006@gmail.com

献给

我的妻子和孩子们

感谢他们的谅解，使我有足够的业余时间翻译完这本书

译者序

人们在学习新事物时往往急于看到它的全貌，而看到全貌时却又失之于细节。学习和使用 Bash 尤其如此。尽管它是 Linux 的首选 shell，但是因为它与我们所熟知的“高级语言”是如此不一致，以至于即使是从业多年的老程序员，写起 shell 程序来也会觉得力不从心。而 Bash 最权威的参考资料，即其程序作者所著的《Bash Reference Manual》，又因为语言的问题不能尽为中国程序员和 IT 从业人士所知悉。所以在很多时候，我们对刚刚崛起的 Linux 只能望洋兴叹。我正是试图改变这种现状才翻译这本书的。事实上，最初促使我翻译的动机并不是来自 Bash 本身，而是来自 LaTeX。在学习 LaTeX 多年以后，我很想做出一些东西来，于是就选择了翻译这本书，因为我使用 Bash 有七八年了，颇有一些心得。这些心得会以脚注或或者备注等形式体现出来。

为了准确的呈现原文的内容，本文在很多地方都做了特殊处理。首先是某些地方中文和英语的表达习惯不一样，为了使中文更具可读性，在不改变原意的情况下，我对原来的语句做了适当的调整。有些地方还加了脚注。原文是用 `texinfo` 来排版的；这便于在计算机上阅读，而排版并翻译成中文却有诸多不便。所以我对原稿的显示风格做了大量的改动，使得翻译出来的中文版更方便阅读和打印。此外，本书原文中有很多地方都对命令进行交叉引用，而命令又很多，只找到章节标题并不能很快的定位到想要查看的命令。所以在交叉引用命令时，我大都做一些改动，以方便读者。

每个页面的页脚中都中指向目录的链接；这样如果在电脑上阅读本书而不是打印，将会非常方便。*1234567890* 是一个数字。 ⏎ 是键盘上的一个键；有的键，例如 C-a 需要和 Control 等其它键一起输入。`abc` 是代码行中的一个或多个单词，可以在命令行中输入或写入到文件中。而 助记词：Mnemonic，助记词 是用来帮助记忆命令或选项的；这些助记词大多数情况下都是来自它所解释对象的英文表达，有些时候则纯粹是为了记忆写臆造的。另外，本书中的交叉引用都是采用“章节号 标题 页码”的完整格式，例如§[译者序], *pi*。

这是应该在命令行中输入的文本。

这段是命令格式的说明。其中的高亮字体部分将在后面的详细讲解中用到。所有的命令格式都在附录中列出。

你知道吗?

这是和上下文相关的作息。

使用小窍门

这是一些小窍门。

注意事项

这是一些典型的错误用法。

*** 代码清单 0：代码清单说明**

这是一些可以直接使用的代码片段。它采用了 Vim 形式的高亮显示，并且加上了行号。

问 0. 这是“常见问题”中的问题与答案。

本书是在 Gentoo Linux 下的 TexLive 环境中使用 Vim 编辑的。成书以后，用 CTexLive2008 最后编译了一次，因为那里的字体好看 (感谢 CTexLive2008 的制作者们)。

本书尚未译完时就有很多读者朋友要求印刷，因为在电脑上看书很伤眼睛。我也正在考虑这件事。如果您愿意购买纸质版本，请尽早发邮件和我联系。

译者
2009年秋

目录

代码列表

第一章 总体介绍

§1.1 什么是 Bash?

Bash 是一个用于 GNU[1] 操作系统的 shell，也就是命令解释器。这个名字是“Borune-Again SHell”[2]的缩略词，意在调侃斯蒂芬•波恩；他写的 sh 是目前 Unix 命令行解释器的前身，最初出现于被贝尔实验室研究用 Unix 的第七版。

Bash 整体上保持与 sh 兼容，并且从科恩 shell ksh 和 C shell csh 引进了一些有用的功能。它的设计力求遵循 IEEE POSIX 规范中的《Shell 和实用工具》一节 (IEEE 标准第 1003.1 号) 的规范，并且在交互和编程运行两方面对 sh 做了功能上的改进。

虽然 GNU 操作系统还提供了其它 shell，包括 csh 的一个版本，但 Bash 是默认的 shell。此外，Bash 和其它 GNU 程序一样，具有很好的移植性。它目前几乎能在任何版本的 Unix 和一些其它操作系统上运行；并且在 MS-DOS，OS/2 和 Windows 等平台上还有独立维护的移植版本。

§1.2 什么是 Shell?

从本质上来说，shell 是一个能执行各种命令的宏处理器。这里，宏处理是指扩展文本和符号以创建更大的表达式的功能。

Unix shell 不仅是一个命令解释器，还是一种编程语言。作为命令解释器，shell 提供了包含众多 GNU 实用工具的用户界面。可编程的特性使得这些实用工具能够被组织起来。可以创建包含若干命令的文件，而这些文件本身又可以作为命令。这些命令和 /bin 等目录下的系统命令具有同等的地位，从而使得用户和用户组能定制运行环境并自动完成他们的常规任务。

Shell 还提供了少量的内部命令 (称为“builtin”)，它们实现的功能是外部工具不方便或者不可能完成的。例如，cd、break、continue 和 exec，它们不能通过 shell 以外的方式实现，因为它们要直接操纵 shell 本身。而诸如 history、getopts、kill 或 pwd 等内部命令，虽然可以在外部单独实现，但是作为内部命令会更便于使用。所有这些内部命令都将在后续章节中介绍。

虽然执行命令是其关键任务，shell 的强大 (和灵活) 之处却在于其中内置了编程语言。和其它高级语言一样，shell 提供了变量、流程控制结构、引用[3]和函数。

Shell 提供了一些专为交互式使用而设计的功能，它们不是为了增强 shell 的编程特性。这些交互式的功能包括作业控制、命令行编辑、命令行历史以及 (命令) 别名。所有这些功能都将在本手册中一一介绍。

[1] 即 Linux。
[2] 意为“波恩 shell 的再生”。
[3] 这里是指字符串周围的引号 (quotes)，而不是如 C++ 中那样的对象地址的引用 — 后者叫 reference。

第二章　术语定义

在本手册的全文中使用了下面的定义：

POSIX　基于 Unix 的一系列操作系统可移植性[1]的标准。Bash 主要和 POSIX 标准第1003.1号中的《Shell 和实用工具》部分有关。

空白符　一个空格或者制表符。

内部命令　在 shell 内部而不是文件系统中由某个可执行文件实现的一些命令。

控制运算符　实现控制功能的一些符号，包括换行符[2] 和下面的任意一个符号：||、&&、&、;、;;、|、|&、(或)。

退出状态　命令返回给调用者的一个值。这个值不得超过八位[3]，所以其最大值是 255。

字段　执行某个 shell 扩展后所得到的文本的一个部分。执行一个命令时，经过 Shell 扩展后得到的各字段分别作为命令的名称和参数。

文件名　用以标志一个文件的字符串。

作业　组成一个管道的一系列进程，以及其衍生出的进程；这些进程都属于同一个进程组。

作业控制　用户可以有选择的终止 (挂起) 和重启 (恢复) 进程执行的一种机制。

元字符　当没有引用时能够分隔开单词的字符。包括空白符和下面的字符之一：|、&、;、(、)、< 以及 >。

名称　只包括数字、字母、下划线，并且以字母或下划线开头的单词。这些名称用作变量和函数的名称，又叫做标志符。

运算符　包括控制运算符和重定向运算符。重定向运算符列表请参见 §3.6[重定向], *p21* 。它至少包括一个未被引用的元字符。

进程组　一系列拥有相同进程组号 的相关的进程。

进程组号　在进程组的生命周期内，能唯一代表该组的一个标志符。

[1] 原文“open system”主要是指面向可移植性的“开放性”，故这里采取意译。
[2] 即 newline。在 Windows 上面，它是 \n\r；在 Linux 上是 \n；在 Mac 上是 \r。
[3] 这里指的是二进位数，尽管实际返回的是十进制数。

保留字 对 shell 来说具有特殊意义的一些单词；它们大部分是用来构建 shell 的控制结构的，例如 `for` 和 `while`。

返回状态 退出状态的同义词。

信号 当系统中发生某个事件时，内核用以通知 (用户) 进程的一种机制。

特殊内部命令 被POSIX标准认为具有特殊作用的命令。

符号 被 shell 当成一个单独单位的一串字符。它要么是一个单词，要么是一个运算符。

单词 被 shell 当成一个单位处理的一串字符；它不能包含未被引用的元字符。

第三章 Shell 的基本功能

Bash 是“Bourne-Again SHell”的缩略词，而Bourne (波恩) shell是原来由斯蒂芬•波恩所作的传统 Unix shell。所有波恩 shell 的内部命令在 Bash 中同样可用，而求值和引用的规则却是来自 POSIX 规范中定义的“标准” Unix shell。

本间简要介绍了 Bash 的结构：命令、控制结构、shell 函数、shell 变量、shell 扩展、重定向 — 即把输入和输出定向到 (自) 文件，以及 shell 是怎么执行命令的。

§ 3.1 Shell 语法

Shell 在读取输入时，要经过一系列的操作。如果在输入中开始了一个注释，shell 会把注释符 (#) 以及后面的一整行都忽略掉。否则，概括的说，shell 会读取输入并将之分解为一个个单词和运算符，并使用引用规则来决定每个单词和字符的不同含义。然后 shell 会把这些解析为命令和其它结构，去除一些特定单词的特殊含义，对另外一些进行扩展，根据需要进行重定向，执行指定的命令，等待其退出状态，并让这个状态能用于后续检查或处理。

§ 3.1.1 Shell 操作

下面简要说明了 shell 读取和执行命令时所进行的操作。简单的说，shell 执行了下面的操作：

1. 从文件 (参见 § 3.8[Shell 脚本], *p26*)，或启动“-c”选项的字符串参数 (参见 § 6.1[Bash 的启动], *p55*) 中，或者用户的终端上读取输入。
2. 按照 § 3.1.1.1[引用], *p5* 中所述规则把输入分解为单词和运算符。这些符号用元字符分隔。该步骤还进行别名扩展 (参见 § 6.6[别名], *p62*)。
3. 把符号解析为简单和复杂命令 (参见 § 3.2[Shell 命令], *p6*)。
4. 进行各种 shell 扩展 (参见 § 3.5[Shell 扩展], *p14*)，并把扩展后的符号分解为文件名 (参见 § 3.5.8[文件名扩展], *p19*)、命令和参数的列表。
5. 进行必要的重定向 (参见 § 3.6[重定向], *p21*)，并把重定向运算符及其参数从参数列表中去掉。
6. 执行得到的命令 (参见 § 3.7[命令的执行], *p24*)。
7. (可选的) 等待命令结束并收集其退出状态 (§ 3.7.5[退出状态], *p26*)。

§3.1.1.1 引用

引用在 shell 中用以去除某些字符或单词的特殊含义。它可以用来禁止对特殊字符的特殊处理，使得保留字不再被认为是保留字，或者禁止参数扩展。

Shell 的每个元字符在 shell 中都有特殊的含义，必须引用后才能代表其自身。如果使用了命令历史扩展的功能 (§9.3[历史扩展], *p92*)，则历史扩展字符 (通常是 `!`) 也引用起来以取消历史扩展。有关历史扩展的更多细节，请参见§9.1[Bash 的历史功能], *p90*。

Bash 中有三种引用机制：转义字符，单引用和双引用。

§3.1.1.2 转义字符

在 Bash 中，没有转义的反斜杠 `\` 是转义字符，它能保留其下一个字符的字面含义，除非这个字符是换行符。如果出现 `\newline`[1]这样的序列，并且反斜杠本身没有被引用，则 `\newline` 就是行连续符；也就是说，它们将会从输入流中被删除并被完全忽略掉。

§3.1.1.3 单引用

把字符串用单引号 (`'`) 引用能保留引号内各个字符的字面含义。在单引号中不允许再出现单引号，即使它已经由反斜杠转义。

§3.1.1.4 双引用

把字符串用双引号 (`"`) 引用能保留引号内各个字符的字面含义，除非这些字符是 `$`、`'`、`\\`、以及 `!` (如果已经打开历史扩展)。在双引号中，`$` 和 `'` 继续保留其特殊的功能 (参见§3.5[Shell 扩展], *p14*)。而反斜杠，只有当其后面的字符是 `$`、`'`、`"`、`\\` 或者换行符时才保留其特殊的含义。在双引号中，如果反斜杠后面是这些字符中的任意一个，则这个反斜杠就会被删除。而它后面字符如果没有特殊的含义，则它将被保留。在双引号中可以出现另外一个双引号，只要它在反斜杠后面。如果打开了历史扩展，`!` 将导致历史扩展，除非它由反斜杠转义。在 `!` 前的反斜杠不会被删除。

特殊变量 `*` 和 `@` 在双引号中有特殊的含义；请参见§3.5.3[Shell 参数扩展], *p16*。

§3.1.2 ANSI 标准 C 引用

形如 `$'string'` 的单词会被特殊处理。这个词将会扩展成一个字符串，其中的转义字符会按照 ANSI C 标准被替换。如果其中出现转义字符序列，则按照下面的规则解释：

`\a` 警告 (响铃)

`\b` 退格删除

`\e` 转义字符 (不属于 ANSI C)

`\f` 走纸换页

`\n` 新行

`\r` 换行

`\t` (水平) 制表符

`\v` 垂直制表符

`\\` 反斜杠

[1] 即`\`是某行的最后一个字符。

`\'` 单引号

`\nn` 由八进制数 *nnn* (一个到三个数字) 代表的一个八位字符。

`\xHH` 由十六进制数 *HH* (一个到两个数字) 代表的一个八位字符。

`\cx` 一个控制字符 CTRL-X

扩展的结果是一个单引用，就好像美元符号原本就不存在一样。

不能用双引号转义

在很多命令中都需要指定单个字符，例如 `tr` 或 `awk` 中的 `IFS` 等变量。这时就应该使用 ANSI 标准 C 引用，而不能用双引号转义。例如

```
tr $'\n' ' ' file
```

可以把文件 `file` 的所有行用空格连在一起。

§3.1.2.1 Locale 专用的翻译

双引用的字符串在美元符号 (`$`) 后面将使得该字符串根据当前的 locale 被翻译过来。如果当前 locale 是 `C` 或者 `POSIX`，则美元符号将被忽略。如果字符串被翻译或者替换，则替换后的字符串是双引用的。

有些系统使用 `LC_MESSAGES` 这个 shell 变量来选择消息目录。也有些系统根据 `TEXTDOMAIN` 这个 shell 变量来决定消息目录的名称，有可能还要加上 `.mo` 后缀。如果使用了 `TEXTDOMAIN` 变量，可能还需要把 `TEXTDOMAINDIR` 设为存放消息目录文件的路径。更有一些系统这样使用这两个变量：

`TEXDOMAINDIR`/`LC_MESSAGES`/*LC_MESSAGES*/`TEXTDOMAIN`.*mo*

§3.1.3 注释

在非交互运行的 shell 中，或者交互运行的 shell 如果打开了内部命令 `shopt` 的 `interactive_comments` 选项 (参见§87[内部命令 shopt], *p46*)，以 `#` 开头的单词将使得该单词及本行中所有其它单词都被忽略。如果交互运行的 shell 没有打开 `interactive_comments` 选项则不允许注释。在默认情况下，交互运行的 shell 已经打开了 `interactive_comments` 选项。至于如何让 shell 变成交互式的，请参见§6.3[交互式的 shell], *p58*。

§3.2 Shell 命令

一个简单的 shell 命令，例如 `echo a b c`，包含该命令本身，其后还有一些参数；它们都用空格分隔。

复杂一些的命令由有简单的命令通过各种方式组合而成的：通过管道命令，这时一个命令的输出成为另一个命令的输入；或者通过循环或条件命令；或者通过其它组合方式。

§3.2.1 简单命令

简单命令使用得最频繁。它仅仅包括空白符分隔的多个单词，其结尾是一个 shell 控制运算符。其中的第一个单词通过指定要执行的命令，而后续单词都是这个命令的参数。

简单命令的返回状态 (参见§3.7.5[退出状态], *p26*) 是 POSIX 1003.1 中的 `waitpid` 函数规定的退出状态；如果该命令由一个信号 *n* 终止，则其退出状态是 *128+n*。

§3.2.2 管道

管道是由控制字符 | 或 |& 分隔开的一系列简单命令。
管道的其格式为：

```
[time [-p]] [!] 命令一 [ [|或|&] 命令二 ··· ]
```

管道里面每个命令的输出都经由管道与下一个命令的输入相连接；也就是说，每个命令都去读取上一个命令的输出。这种连接早在命令中指定的任何重定向之前就已经进行了。

如果使用了 |&，则命令一的标准错误输出将会和命令二的标准输出相连，这是 2>&1 | 的简写形式。这种对标准错误输出的隐含重定向是在命令中指定的任何重定向之后进行的。

保留字 time 能够在管道执行完毕后输出其执行时间的统计信息。这个统计目前包括执行该命令所花费的总时间 (钟表时间) 以及用户和系统时间。-p 选项助记词：POSIX使得输出的形式和 POSIX 中的规定相同。可以设置 TIMEFORMAT 变量为一个格式化字符串，以指定时间输出的形式。至于所有可用的格式，请参见§5.2[Bash 的变量], *p51*。把 time 作为保留字允许我们统计内部命令，shell 函数，以及管道的执行时间；而如果 time 是外部命令就不能很容易的做到这一点。

> **time 和 times**
>
> 在 Bash 中，time 是用于管道的保留字，而 times 是一个内部命令。它们的作用相同，使用的场合却不一样。参见§4.1[波恩 Shell 的内部命令 times], *p33*。

如果该管道不是异步 (参见§3.2.3[命令队列], *p7*) 执行的，则 shell 会等待管道中所有命令执行的结束。

管道里面的每个命令都是在自己的子 shell (参见§3.7.3[命令执行的环境], *p25*) 里面执行的。管道的退出状态是其中的最后一个命令的退出状态，除非打开了 pipefail 选项 (参见§86[内部命令 set], *p43*)。如果打开了 pipefail 选项，则管道的返回状态是其中最后一个 (最靠右的) 返回非零的那个命令的状态；如果所有命令都成功执行，则返回零。如果管道的前端有保留字 !，则其返回状态是按照如上据说再进行逻辑取反。Shell 等待管道里面的所有命令的结束，然后才返回一个值。

§3.2.3 命令队列

命令队列是由一个或者管道通过运算符 ;、&、&&、|| 连接而成，最后还可以 (可选的) 由 ;、&、或换行符结束。

在之些队列运算符中，&& 和 || 具有同样的优先级；其次是 ; 和 &，这两个也有同样的优先级。

在命令队列中可以使用一个或多个换行符分隔命令，这与分号是等价的。

如果一个命令是由控制字符 & 结束，则 shell 会不同步的在子 shell 中执行该命令。我们通常称之为在“后台”运行该命令。这时 shell 并不等待命令的结束，而其返回状态 *0* (即逻辑真)。如果没有启用作业控制 (参见§7[作业控制], *p69*)，并且也没有显式指定重定向，则在异步执行的命令的标准输入将被重定向到 /dev/null。

由 ; 分隔的命令将相继执行。Shell 依次等待每个命令的结束。整个返回状态是最后一个要执行命令的返回状态。

“与”和“或”命令队列是分别由控制运算符 && 和 || 分隔的一个或多个管道。“与”和“或”按照左结合的方法执行。

“与”队列具有如下形式：

```
命令一 && 命令二
```

其中，当且仅当命令一返回值为零时才执行。

“或”队列具有如下形式：

```
命令一 || 命令二
```

其中，当且仅当命令一返回值为非零时才执行。

“与”和“或”队列的返回值是其中最后一个被执行的命令的返回值。

§3.2.4 复合命令

复合命令是 shell 的编程结构体。每个结构体都是以保留字或者控制运算符开头，然后以与之对应的保留字或控制运算符结束。任何与复合命令相关的重定向 (参见§3.6[重定向], *p21*) 都作用于该复合命令里面的所有命令，除非显式覆盖。

Bash 提供了循环结构，条件结构，以及将命令分组并将之整体执行的机制。

§3.2.4.1 循环结构

Bash 支持以下的循环结构。

请注意：在介绍命令的语法时不管在哪里使用了 ;，都可以用一个或多个换行符来代替。

A. until

until 命令的语法格式是

```
until 测试命令; do 命令块; done
```

只要测试命令返回非零值就执行命令块。其返回值是命令块中最后一个被执行的命令的返回值。如果命令块没有被执行则返回零。

B. while

while 命令的语法格式是

```
while 测试命令; do 命令块; done
```

只要测试命令返回零就执行命令块。其返回值是命令块中最后一个被执行的命令的返回值。如果命令块没有被执行则返回零。

C. for

for 命令的语法格式是

```
for 变量 [in 单词 ]; do 命令块; done
```

将单词扩展成一个列表，然后把结果中列表的每个元素都赋值给变量并执行一次命令块。如果没有“in 单词”这部分，则for 依次对每个位置参数都执行一次命令块，就好像指定了“in $@”一样 (参见§3.4.2[特殊参数], *p14*)。其返回值是命令块中最后一个被执行的命令的返回值。如果对单词的扩展没有得到任何元素，则不执行任何命令，并返回零。

for 命令还支持另外一种格式：

```
for (( 表达式一 ; 表达式二 ; 表达式三 )); do 命令块 ; done
```

首先，按照下面将要介绍 (参见§6.5[Shell 的算术运算], *p61*) 规则对算术表达式一进行求值。然后不断的对算术表达式二进行求值，直到其结果为零。每次求值时，如果表达式二的值不是零，则执行一次命令块，并且计算算术表达式三的值。如果省略了任意一个表达式，则效果就好像该表达式总是返回一。其返回值是命令块中最后一个被执行的命令的返回值。如果表达式的值都是假的，则返回假。

[分节结束]

可以使用内置命令break和continue (参见§4.1[波恩 Shell 的内部命令 break], *p28* 和§4.1[波恩 Shell 的内部命令 continue], *p30*) 来控制循环命令的执行。

§3.2.4.2 条件结构

A. if

if 命令的语法格式是

```
if 测试命令一 ; then
   命令块一 ;
[elif 测试命令二 ; then
   命令块二 ;]
   ⋮
[else
   其它命令块 ;]
fi
```

首先执行测试命令一这个命令列表，如果其返回值为零，则执行命令列表命令块一。如果测试命令一返回非零值，则依次执行每个elif列表，如果其返回值为零，则执行其对应的命令块，并结束整个命令。如果有“else 其它命令块”，并且 if 或者 elif 子句的最后一个命令返回值为非零，则执行其它命令块。整个命令的返回值是最后一个被执行的命令的返回值。如果没有一个条件为真，则返回零。

B. case

case 命令的语法格式是

```
case 单词 in
   [ [(] 模式一 [| 模式二 ] ··· )
      命令块
      ;;]
   ⋮
esac
```

case 命令会选择性的执行与单词所匹配的第一个模式对应的命令块。如果打开了 shell 的 nocasematch 选项 (参见§87[内部命令 shopt], p47的描述)，则匹配时将忽略字母的大小写。“|”用来分隔多个模式，而“)”用来结束模式列表。[2]模式列表和其对应的命令块叫做一个“分句”。每个分句都必须由“;;”，“;&”或者“;;&”结束。这里的单词在匹配之前要经过波浪号扩展、参数扩展、命令替换、算术扩展以及引用去除，而每个模式也要经过波浪号扩展、参数扩展、命令替换、算术扩展等步骤。

case 分句的数量量不限的，但是每个分句都以“;;”，“;&”或者“;;&”结束。最先匹配的模式决定了哪个命令块被执行。下面是 case 在脚本中使用的一个例子，它描述一些动物的有趣特征：

* 代码清单 1：case 的例子

```
1 echo -n "请输入一个动物的名称："
2 read ANIMAL
3 echo -n "$ANIMAL有"
4 case $ANIMAL in
5     ( 马 | 狗 | 猫 ) echo -n "四";;
6     人 | 袋鼠 ) echo -n "两";;
7     *) echo -n "未知数目的";;
8 esac
9 echo "腿。"
```

[2]模式列表前面可以加一个与“)”匹配的“(”，但不是必须的。

如果使用了“;;” (来结束分句) ,则匹配第一个模式以后就不会再匹配其它模式。如果用“;&”来代替“;;”，则执行命令块后，如果下面还有其它分句，就继续执行该分句。如果用“;;&”来代替“;;”，则执行命令块后，如果下面还有其它分句，就检查其模式 (如果有的话) ；如果其模式为真，继续执行其对应的命令块。

如果任何模式都不匹配，该命令的返回状态是零；否则，返回最后一个被执行的命令的返回值。

C. select

select 结构使得菜单的生成变得简单。它的语法格式几乎和 for 命令一样：

```
select 名称 [in 单词表 ··· ] ; do 命令块; done
```

in 后面的单词表被扩展并生成一个项目列表；这个扩展后的列表将会打印到标准错误输出流中，并且每个项目前面都会加上一个序数。如果省略了“in 单词表”部分，则打印位置参数，就好像使用了“in $@”一样。之后，将显示 PS3 提示符，并且从标准输入读取一行的输入。如果输入的行含有一个与所打印出的词项对应的数字，则 name 的值就被置为该单词。如果输入的为空，则重新显示单词表和提示符。如果输入了 EOF 字符，select 命令将结束。输入任何其它值都会使名称被置为空值。读取到地值被存放在变量 REPLY 中。每次选择之后都会执行命令块，直到遇到一个 break 命令为止—这时 select 命令将结束。

下面是 select 的一个例子。该例允许用户从当前目录中选择一个文件，并显示用户选择的文件名及其序号。

❋ 代码清单 2: select 的例子

```
1 select fname in *;
2 do
3     echo you picked $fname \($REPLY\)
4     break;
5 done
```

D. ((···))

```
(( 算术表达式 ))
```

根据后面将要介绍的规则 (参见§6.5[Shell 的算术运算], *p61*) 对算术表达式求值。如果这个值不是零，则返回状态是零，否则返回*1*。这和下面的命令完全等价：

```
let "表达式"
```

关于内部命令 let 的完整介绍，请参见§4.2[Bash 的内部命令 let], *p38*。

E. [[···]]

```
[[ 条件表达式 ]]
```

对条件表达式求值，并根据其结果返回 *0* 或者 *1*。条件表达式是由§6.4[Bash 条件表达式], *p59*中介绍的原子成分组成的。在 “[[” 和 “]]” 中间的单词不会进行单词和文件名扩展，但却进行波浪号扩展、参数和变量扩展、算术扩展、命令替换、进程替换以及引用去除。诸如“-f”等条件运算符不能被引用，否则它们就不是原子算术表达式了。

如果使用了“==”和“!=”运算符，则运算符的右边会被看作是一个模式，并且按照§3.5.8.1[模式匹配], *p20* 中所介绍的匹配规则进行匹配。如果打开了 shell 的 nocasematch 选项 (参见§87[内部命令 shopt],

p47)，则匹配时不考虑字母的大小写。如果使用了“==”并且字符串匹配，或者使用了“!=”并且字符串不匹配，则其返回值是 0，否则返回 1。模式的任何部分都可以被引用以强制把其当作字符串来匹配。

还可以使用另外一个双目运算符“=~”；它和“==”以及“!=”具有同样的优先级。如果使用了它，则其右边的字符串就被认为是一个扩展的正则表达式来匹配 (如 regex *3* 一样[3])。如果字符串和模式匹配，则返回值是 0，否则返回 1。如果这个正则表达式有语法错误，则整个条件表达式的返回值是 2。如果打开了 shell 的 nocasematch 选项 (参见§87[内部命令 shopt], *p47*)，则匹配时不考虑字母的大小写。模式的任何部分都可以被引用以强制把其当作字符串来匹配。由正则表达式中括号里面的子模式匹配的字符串被保存在数组变量 BASH_REMATCH 中。BASH_REMATCH 中下标为 0 的元素是字符串中与整个正则表达式匹配的部分。BASH_REMATCH 中下标为 n 的元素是字符串中与第 *n* 个括号里面的子模式匹配的部分。

表达式可以通过下面的运算符 (按优先级降序排列) 组合在一起：

(表达式) 返回表达式的值。这样写可以改变运算符的正常优先级。

! 表达式 如果表达式为假，则返回真。

表达式一 && 表达式二 如果表达式一和表达式二同时为真则返回真。

表达式一 || 表达式二 如果表达式一或者表达式二为真则返回真。

如果表达式一的值已经足以判断整个条件表达式的返回值，则&& 和 || 运算符就不再对表达式二进行求值。

§3.2.4.3 命令组合

Bash 提供了两种方式来把一系列命令放在一起作为整体执行。当命令被组织在一起时，可以对整个命令列表进行重定向。例如，命令列表中所有命令的输出都可以重定向到一个单一的流中。

A. ()

```
( 表达式 )
```

把一列命令放在括号中间就会创建一个子 shell 环境 (参见§3.7.3[命令执行的环境], *p25*) 并在这个子 shell 中执行该列表中的每个俱。就是因为命令列表是在子 shell 中执行的，所以在子 shell 结束后，其中的变量赋值将不再有效。

B. {}

```
{ 表达式; }
```

把一列命令放在大括号中间，这列命令就会在当前 shell 中执行，而不是创建子 shell。命令列表后面的逗号 (或者换行符) 是必须的。

[3]指 manpage。

命令的分隔符

在 Bash 的分隔符中，“;”的使用是最频繁的。它除了具有分隔的作用，没有其它任何含义；所以，任何命令的末尾都可以使用它。但如果一个命令单独成行，这个符号完全可以省略。例如，

```
for name in *;do echo ${name/.png/.jpg}; done
```

和

```
for name in *
do
    echo ${name/.png/.jpg}
done
```

是完全等价的。

[分节结束]

除了子 shell 的创建，上述两种结构之间由于历史的原因还有微妙的差别。大括号是保留字，所以它们与命令列表之间必须用空白符或其它 shell 的元字符分开；而圆括号是运算符，所以即使它们和命令列表之间没有用空白符分开也会被 shell 当作独立的符号。

这两种结构的命令返回值都是其中命令列表的返回值。

§3.2.5 协同进程

协同进程 *coprocess* 是指一个 shell 命令前面有 `coproc` 保留字；它是在子 shell 中异步执行的，就好像这个命令后面有控制运算符 `&` 一样。协同进程和其父 shell 进程之间有双向的管道。这个命令的格式是：

```
coproc [ NAME ] 命令 [ 重定向 ]
```

上述命令创建了一个名为 `NAME` 的协同进程。如果没有指定 `NAME`，默认的名称是 `COPROC`。如果这里的命令只是一个简单命令 (参见§3.2.1[简单命令], *p6*)，则不能指定 `NAME`，否则它会被当作命令的第一个单词。

当 `coproc` 执行时，shell 会在父进程中创建一个名为 `NAME` 的数组变量 (参见§6.7[数组], *p62*)。命令的标准输出通过管道和父进程的一个文件描述符相连；该文件描述符被赋给 `NAME[0]`。命令的标准输入通过管道和父进程的一个文件描述符相连；该文件描述符被赋给 `NAME[1]`。这个管道是在命令当中指定的任何重定向 (参见§3.6[重定向], *p21*) 之前就建立了。这些文件描述符可以在 shell 命令和重定向中通过标准的单词扩展而当作参数使用。

用来执行协同进程的子 shell 的进程号保留在数组变量 `NAME[PID]` 中。可以使用内部命令 `wait` 来等待协同命令的结束。协同进程的返回状态是其中命令的返回状态。

§3.3 Shell 函数

Shell 函数把一组命令与单一的名称相关联，以便以后执行。在执行时，它们就和“常规”命令一样。如果 shell 函数的名称被当作一个简单命令使用，与它相关联的命令就会被执行。Shell 函数是在当前的 shell 环境中执行的，而不是创建新的进程来执行。

函数通过下面的语法来定义：

```
[ function ] 名称 () 复合命令块 [ 重定向 ]
```

上面定义了一个叫做名称的函数。保留字 `funnction` 是可选的。如果有 `function` 这个保留字，则可以省略括号。复合命令块 (参见§3.2.4[复合命令], *p8*) 是函数体；它通常是包含在 { 和 } 之间的命令列表，也可以是上面列出的任何复合命令。每当名称被指定为一个命令名时，复合命令块就会被执行。当函数被执行时，与之相关的重定向 (参见§3.6[重定向], *p21*) 也会同时被执行。

可以使用内部命令 unset 的“-f”选项 (参见§4.1[波恩 Shell 的内部命令 unset], p34) 来取消函数的定义。

除非发生语法错误，或者一个同名并且为只读的函数已经存在，函数定义的返回值是零。执行时，函数的返回值是函数体内最后一个被执行命令的返回值。

注意，由于历史的原因，通常情况下，函数体外的大括号与函数体之间必须用空白符或者换行符分开。因为大括号是保留字，但是只有它们与其中间的命令列表空格或其它 shell 元字符分隔时才能被识别为保留字。此外，使用大括号时，其中间的命令列表必须用逗号、“&”或者换行符结束。

函数执行时，传递给它的参数成为它执行期间的位置参数 (参见§3.4.1[位置参数], p14)。能扩展为位置参数个数的特殊参数“#”将随之更新。特殊参数 0[4] 不变。在函数执行时，变量 FUNCNAME 的第一名元素被设为函数的名称。除了 DEBUG 和 RETURN 这两个陷阱没有被继承以外，函数和其调用者之间在Shell 执行环境所有其它方面都完全一样。如果用内部命令 declare 设置了函数的 trace 属性，或者设置了内部命令 set 的 functrace 选项，以便使，则函数也会继承调用者的 DEBUG 和 RETURN 陷阱。参见§4.1[波恩 Shell 的内部命令 trap], p33 的描述。

如果在函数里面执行了内部命令 return，则函数的执行将结束，并且返回到调用函数那里的下一个命令。任何与 RETURN 陷阱相关联的命令都将在执行恢复前被执行。函数结束时，位置参数以及特殊参数“#”的值恢复到函数被执行以前的状态。如果 return 带有一个数值型参数，则这个参数就是函数的返回值；否则，函数的返回状态是其返回前最后一个被执行命令的返回状态。

函数本地的变量可以用内部命令 local 来声明。这些变量只对函数及它使用的命令是可见的。

函数名称及其定义可以用内部命令 typeset 或者 declare 加上“-f”选项 (参见§4.2[Bash 的内部命令 declare], p36) 来列出；而typeset 或者 declare 加上“-F”选项只列出函数名 (如果打开了 shell 的 extdebug 选项，则还会列出源文件和行号)。通过内部命令 export 的“-f”选项 (参见§4.1[波恩 Shell 的内部命令 export], p30)，还可以把函数导出，使得它们在子 shell 中自动得以定义。注意，如果 shell 函数和变量同名，则可能导致传给 shell 子进程的环境中有多个完全一样的名字。如果这样会引发问题，就需要避免同名。

函数可以是递归的。对递归调用的次数没有限制。

§3.4 Shell 参数

参数是能存储值的实体；它可以是一个名称、一个数字、或者下面列出的特殊字符之一。变量是名称所代表的参数。每个变量都有值以及零个或多个属性。属性通过内部命令 declare (参见§4.2[Bash 的内部命令 declare], p36) 来设置。

参数通过赋值来设置。空字符串也是一个有效的值。参数一旦设置以后，只能通过内部命令 unset 才能取消设置。

可以通过下面的语句形式给参数赋值[5]：

```
名称 =[值]
```

如果没有给定值，则变量被赋于空字符串。所有的值都会进行大括号扩展、参数和变量扩展、命令替换、算术扩展、以及引用去除 (详见下述)。如果启用了变量的 integer 属性，则把该变量当作算术表达式求值，即使没有使用 $((···)) 扩展 (参见§3.5.4[命令替换], p18)。除了下述的“$@”，否则不会进行单词扩展。文件名扩展也不会进行。赋值语句还可以作为内部命令 alias、declare、typeset、export、readonly 和 local 的参数。

在赋值语句给 shell 变量或数组 (参见§6.7[数组], p62) 元素[6]赋值的行文中，可以使用“+=”运算符附加或增加到变量原来的值中。如果变量启用了 integer 属性并且使用“+=”，则按照算术表达式对值进行求值，并把它加入到变量原来的值中后再求值。如果对数组变量进行复合赋值 (参见§6.7[数组], p62) 时使用了“+=”，则变量原来的值不会被覆盖 (像用“=”那样)，新的值被附加到数组中下标最大的那个元素的后面 (对下标数组而言)，或者在键值数组中新增一个键值对。如果对字符串变量使用它，则把值进行扩展并附加在变量 的值后面。

[4] 即 $0，通常表示脚本的名称。

[5] 紧挨着等号前后的第一个字符不能是空格，否则在进行下述的七种扩展时会遇到麻烦。如果的确想在值中使用空格，可以将值放在引号中。

[6] 原文为 *index*，实际应该是指数组中位于 *index* 的某个元素。

§3.4.1 位置参数

位置参数是由除了单个 *0* 以外的一个或多个数字表示的参数；它是在 shell 启动时由其参数赋值的，并且可以用内部命令 `set` 来重新赋值。第 *N* 个位置参数可以表示为 `${N}`；如果 *N* 只含有一个数字，也可以表示为 `$N`。位置参数不可以通过赋值语句来赋值；而应该用内部命令 `set` 或者 `shift` (参见§4.3.1[内部命令 set], *p42*和§4.1[波恩 Shell 的内部命令 `shift`], *p32*) 来设置或删除。在执行 shell 函数时 (参见§3.3[Shell 函数], *p12*)，位置参数会暂时被更换。

含有多于一个数字的位置参数在扩展时必须放在大括号中。

§3.4.2 特殊参数

Shell 会对一些参数特殊处理。这些参数只能使用而不能对它们赋值。

`*` 扩展为从 *1* 开始的所有位置参数。如果它出现在双引号中，则扩展为一个包含每个参数的单词，参数之间用特殊变量 `IFS` 的第一个字符分隔。也就是说，`"$*"` 和 `"$1c$2c···"` 是等价的；其中，`c` 是特殊变量 `IFS` 的第一个字符。如果 `IFS` 没有设置，则参数之间用空格分隔。如果 `IFS` 为空，则参数直接相连，中间没有分隔。

`@` 扩展为从 *1* 开始的所有位置参数。如果它出现在双引号中，则每个参数都扩展为一个单词；也就是说，`"$@"` 和 `"$1c" "$2c" ···` 是等价的。其中，`c` 是特殊变量 `IFS` 的第一个字符。如果 `IFS` 没有设置，则参数之间用空格分隔。如果 `IFS` 为空，则参数直接相连，中间没有分隔。如果这这样的双引号扩展发生中单词里面，则第一个参数扩展后与原单词的开始部分连在一起，而最后一个参数扩展后与原单词的最后一个部分连在一起。如果没有位置参数，则 `"$@"` 和 `$@` 扩展后为空，也即它们会被删除。

`#` 扩展为位置参数的个数，用十进制表示。

`?` 扩展为最近在前台执行的命令的退出状态。

`-` (连字符) 扩展为当前的所有选项；这些选项是启动时给定的，或者通过内部命令 `set` 打开的，或者由 shell 本身打开的 (例如 `-i` 选项)。

`$` 扩展为当前 shell 的进程号。在子 shell `()` 中，扩展为启动 shell 的进程号，而不是子 shell 的进程号。

`!` 扩展为最近在后台 (异步) 执行的命令的退出状态。

`0` 扩展为 shell 或者 shell 脚本的名称。它是在 shell 初始化时设置的。如果 Bash 启动时带有包含命令的文件名参数 (参见§6.1[Bash 的启动], *p55*)，`$0` 就被设为该文件名。如果 Bash 启动时带有“`-c`”选项 (参见§6.1[Bash 的启动], *p55*)，则 `$0` 被设为待执行字符串后面的第一个参数 (如果这个参数存在)。否则，它就是用来启动 Bash 的文件名，即 (命令行的) 第一个参数。

`_` (下划线) 在 shell 启动时，设为启动 shell 的绝对路径，或者在执行环境或参数列表中所传递的待执行的 shell 脚本的绝对路径。随后，扩展为前一条命令的最后一个参数扩展后的值。还可设为每个已执行命令的绝对路径，这些路径是启动时指定的并且导入到命令的执行环境中。检查邮件时，这个变量保存邮箱文件的文件名。

§3.5 Shell 扩展

命令行被拆分成符号以后要进行扩展；扩展的方式有七种：

- ☞ 大括号扩展
- ☞ 波浪号扩展

- ☞ 参数和变量扩展
- ☞ 命令替换
- ☞ 算术扩展
- ☞ 单词拆分
- ☞ 文件名扩展

扩展的顺序是：大括号扩展，波浪号扩展，参数、变量和算术扩展以及命令替换 (按从左到右的顺序)，单词拆分，以及文件名扩展。如果系统支持，则还有另外一种扩展，即进程替换；它和参数、变量和算术扩展以及命令替换是同时进行的。

只有大括号扩展，单词扩展，以及文件名扩展在扩展时能够改变单词的数目。其它的扩展都是单个单词扩展成单个单词。唯一例外的是对“$@” (参见§3.4.2[特殊参数], *p14*) 和“${name[@]}” (参见§6.7[数组], *p62*) 的扩展。所有扩展完成后再进行引用去除 (参见§3.5.9[引用去除], *p21*)。

§3.5.1 大括号扩展

大括号扩展是一种能够生成任意字符串的机制；它和文件名扩展 (参见§3.5.8[文件名扩展], *p19*) 是相似的，但是生成的文件名不一定存在。进行大括号扩展的模式在形式上有一个可选的前缀；其后是一组用逗号分隔的字符串，或者是一个序列表达式，它们都在一对大括号之间；最后是一个可行的后缀。前缀部分将放在大括号中每个字符串的前面，而后缀将放在每个结果的后面，它们都是从左到右进行扩展的。例如，

```
bash$ echo a{d,c,b}e
ade ace abe
```

序列表达式的形式是

```
{x .. y [ 增量 ] }
```

其中，x 和 y 是整数或者单个字符；而可选的增量是一个整数。如果使用了整数，则扩展成 x 的 y 之间的每个整数，包括 x 和 y。所使用的整数可以用“*0*”开头，以使每个量都有同样的宽度。当 x 或者 y 有前导的零时，shell 会试图强制让每个生成的量都含有同样多的数位，如果不同就在前面补零。如果使用了字符，那么表达式就扩展成在字母表中 x 和 y 中间的每个字母，包括 x 和 y。注意，x 和 y 必须有同样的类型。如果指定了增量，它就是每个量之间的差值。默认的增量是 *1* 或者 *-1*，根据情况[7]而定。

大括号扩展在其它所有扩展之前进行；在其它扩展中特殊的字符都被保留下来。它完全是字面上的扩展。Bash 不会对扩展的上下文字或大括号之间的文本进行任何语义解释。为了避免与参数扩展冲突，大括号扩展不会识别字符串中的“${”。

格式正确的大括号扩展必须包含没有被引用的起始和结束大括号，还有至少一个未被引用的逗号或者序列表达式。格式不正确的大括号扩展不会被处理。

为了防止被认为是大括号扩展的一部分，{ 或者“,”可以用反斜杠转义。为了避免与参数扩展冲突，大括号扩展不会识别字符串中的“${”。

当要生成的字符串具有的公共前缀比上面的例子更长时，大括号扩展通常用来简写：

```
mkdir /usr/local/src/bash/{old,new,dist,bugs}
chown root /usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}}
```

§3.5.2 波浪号扩展

如果一个单词以未被引用的波浪号“~”开头，则其后的所有字符，直到第一个未被引用的斜杠 (如果有未被引用的斜杠)，都被看作波浪号前缀。如果波浪号前缀里面的字符都没有被引用，则其中波浪号后面的所有字符就被当成一个可能存在的登录用户名。如果这个登录名是个空字符串，波浪号就被替换成 shell 变

[7] 具体来说，如果 x 比 y 小，则增量是 *1*；如果 x 比 y 大，则增量是 *-1*。

量中 HOME 的值。如果没有设置 HOME，则替换成执行该脚本那个用户的主目录。否则，波浪号前缀就被替换成其中指定的那个登录名的主目录。

如果波浪号前缀是“~+”，它就会被 shell 变量 PWD 的值取代。如果波浪号前缀是“~-”，它就会被 shell 变量 OLDPWD 的值取代 (如果这个值已经设置)。在波浪号前缀中，如果波浪号后面的字符包含数字 *N*，前面可能还有“+”或“-”，那么这个波浪号前缀就会被目录栈 (参见§6.8[目录栈], *p63*) 中相应的元素取代，这个元素是内部命令 dirs 以波浪号前缀中波浪号后面各字符作为参数所要显示的内容。如果波浪号前缀中除去波浪号的部分仅仅包含数字，而没有前导的“+”或“-”，则认为指定了“+”。

如果登录名是无效的，或者波浪号扩展失败，则该单词不会被处理。

每个变量在赋值时都会检查紧接着“:”或者“=”是否含有波浪号前缀。如果找到，就进行波浪号扩展。所以，在给 PATH、MAILPATH 以及 CDPATH 赋值时，可以使用带波浪号的文件名；这时，shell 会用扩展后的量赋值。

下面的例子显示了 Bash 是如何处理未被引用的波浪号前缀：

`~` $HOME 的值

`~/foo` “$HOME/foo”

`~fred/foo` 用户 fred 主目录中的子目录 foo，即“$PWD/foo”

`~+/foo` “$PWD/foo”

`~-/foo` “${OLDPWD-'~-'}/foo”

`~N` 命令“dirs +*N*”所显示的字符串。

`~+N` 命令“dirs +*N*”所显示的字符串。

`~-N` 命令“dirs -*N*”所显示的字符串。

§3.5.3 Shell 参数扩展

字符“$”引导参数扩展，命令替换和算术扩展。将要扩展的变量名或符号可以放在大括号中。大括号虽然是可选的，但却可以保护待扩展的变量，使得紧跟在大括号后面的部分名称不会被扩展。如果使用了大括号，则与这匹配的结束半边是第一个没有用反斜杠转义或不属于引用字符串的“}”，这个结束符不能嵌入在算术扩展、命令替换、或者参数扩展之中。

参数扩展的基本形式是

```
${参数}
```

结果用参数的值替换。如果参数是包含多个数位的位置参数，或者参数后面的字符不应该当成是整个名称的一部分，则大括号是必须的。

如果参数的第一个字符是个感叹号，就表示某个级别的间接变量。Bash 使用后续变量的值作为新变量的名称，然后扩展这个新的变量，并用其值进行替换，而不是后续变量的值。这叫做间接扩展。下面将介绍的 ${!前缀*} 和 ${!名称[@]} 属于例外情况。感叹号必须紧跟在大括号后面才表示间接变量。

正确使用间接变量

在很多其它语言中，可以用 $$A 来表示以 $A 为名称的间接变量，而 Bash 中不可以，即使 ${$A} 也不可以；Bash 只识别感叹号形式的间接变量。不过，这个功能在其它的 shell 中可能没有。所以，为了增强可移植性，可以这样写：

```
eval echo \$$B
```

在下面介绍的每种情况中，名称都要进行波浪号扩展、参数扩展、命令替换和算术扩展。如果不是进行字符串扩展，使用下面的形式时，Bash 会检查参数是否已经设置或者为空；只有当参数尚未设置时才会在测试时忽略冒号后面的结果。换句话说，如果包含了冒号，则这个运算符会测试参数是否存在，同时还会检查它是否为空；如果忽略了冒号，就只检查变量参数是否存在。

`${参数:-单词}` 如果参数没有设置或者为空，则替换为单词；否则替换为参数的值。

`${参数:=单词}` 如果参数没有设置或者为空，则把扩展后的单词赋给参数，然后替换为参数的值。对位置参数和特殊参数，不可以这样进行赋值。

`${参数:?单词}` 如果参数没有设置或者为空，就把扩展后的单词 (如果没有给出单词，则代之以一条大意相同的相信) 写到标准错误输出中。如果当前的 shell 是交互式的，退出 shell。否则，替换为参数的值。

`${参数:+单词}` 如果参数没有设置或者为空，不进行任何替换；否则，替换为扩展后的单词。

`${参数:偏移量}`

`${参数:偏移量:长度}` 扩展为参数中从偏移量开始的不超过长度个字符。如果没有指定长度，扩展为参数中从偏移量开始的子字符串。长度和偏移量都是算术表达式 (参见§6.5[Shell 的算术运算], *p61*)。这又叫做“子字符串扩展”。

长度的值必须是个大于或等于零的数字。如果长度的值是个小于零的数，它就会被当成参数所表示的字符串中从结尾开始的偏移量。如果参数是“`@`”，结果就是从偏移量开始的第长度个位置参数。如果参数是带有“`@`”或“`*`”下标的下标数组名，则结果是该数组中从 `${参数[偏移量]}` 开始的长度个元素。负的偏移量是从数组中比最大的下标大一的数字开始的。对键值数组进行子字符串扩展的结果没有定义。注意，负数的偏移量与冒号之间至少得有一个空格，这样可以避免与“`:-`”扩展相混淆。查找子字符串的下标是从 *0* 开始的；但是如果使用了位置参数，则默认从 *1* 开始。如果使用位置参数时偏移量是 *0*，则把 `$@`添加到结果前面[8]。

`${!前缀*}`

`${!前缀@}` 扩展为名称中含有前缀的变量，以特殊变量 `IFS` 的第一个字符分隔。如果使用了“`@`”，并且在双引号内扩展，则每个变量都扩展成单独的单词。

`${!名称@}`

`${!名称*}` 如果名称是个数组变量，扩展成名称内数组下标或者键名列表。如果名称不是数组变量，则如果名称已经设置就扩展为 *0*，否则扩展为空值。如果使用了“`@`”，并且在双引号内扩展，则每个键或下标都扩展成单独的单词。

`${#参数}` 替换为参数扩展后所含的字符数目。如果参数是“`*`”或“`@`”，则替换为位置参数的个数。如果参数是带有下标“`*`”或“`@`”的数组名，则在做的为该数组中元素的个数。

`${参数#单词}`

`${参数##单词}` 单词被扩展成一个模式，就像文件名扩展 (参见§3.5.8[文件名扩展], *p19*) 一样。如果这个模式与参数扩展后的值开始部分匹配，则替换的结果是该模式与参数扩展后的值最短的匹配部分 (指“`#`”) 或者最长的匹配部分 (指“`##`”) 删除以后的字符串。如果参数是“`*`”或“`@`”，则模式删除操作就对位置参数依次进行，扩展的结果就是所得到的位置参数列表。如果参数是带有下标“`*`”或“`@`”的数组名，则模式删除操作就对数组元素依次进行，扩展的结果就是所得到的数组元素列表。

[8] 这个尚待验证。

`${参数%单词}`

`${参数%%单词}` 单词被扩展成一个模式，就像文件名扩展一样。如果这个模式与参数扩展后的值开始部分匹配，则替换的结果是该模式与参数扩展后的值最短的匹配部分 (指“%”) 或者最长的匹配部分 (指“%%”) 删除以后的字符串。如果参数是“*”或“@”，则模式删除操作就对位置参数依次进行，扩展的结果就是所得到的位置参数列表。如果参数是带有下标“*”或“@”的数组名，则模式删除操作就对数组元素依次进行，扩展的结果就是所得到的数组元素列表。

`${参数/模式/字符串}` 模式被扩展成一个模式，就像文件名扩展一样。参数被扩展后，与模式匹配的最长部分用字符串来取代。如果模式以“/”开头，则所有与之匹配的部分都用字符串来取代；而通常情况下，只取代第一个匹配的部分。如果模式以“#”开头，它就只能和参数扩展后的开始部分匹配；。如果模式以“%”开头，它就只能和参数扩展后的结尾部分匹配。如果字符串为空，则与模式匹配的部分将被删除；这时，模式后面的 / 也可以省略。如果参数是带有下标“*”或“@”的数组名，则模式删除操作就对数组元素依次进行，扩展的结果就是所得到的数组元素列表。

`${参数^模式}`

`${参数^^模式}`

`${参数,模式}`

`${参数,,模式}` 这种扩展会改变参数中字母符号的大小写。模式被扩展成一个模式，就像文件名扩展一样。运算符“^”把匹配部分的小写字母转换为大写；运算符“,”把匹配部分的大写字母转换为小写。“^^”和“,,”扩展转换参数扩展后的每个匹配的字符，而“^”和“,”扩展只转换参数扩展后第一个匹配的字符。如果省略了模式，则它则当成“?”，这就与每个字符都匹配。如果参数是“*”或“@”，则大小写转换操作就对位置参数依次进行，扩展的结果就是所得到的位置参数列表。如果参数是带有下标“*”或“@”的数组名，则模式删除操作就对数组元素依次进行，扩展的结果就是所得到的数组元素列表。

§3.5.4 命令替换

命令替换用命令的输出取代命令本身，它的形式可以是下面任意一种：

```
$( 命令 )
```

或者

```
` 命令 `
```

进行扩展时，Bash 先执行命令，并把该命令的标准输出中最后面的换行符删除，用结果取代命令替换。这个时候中间的换行符不删除，但是它们可能在单词拆分时被删除。命令替换形式 `$(cat 文件名)` 可以用与其等价但速度更快的 `$(<␣文件名)` 来取代。

如果使用了旧式的反引号，反斜杠就保留其字面含义，除非它后面是“$”、“`”或者“\”。第一个不是出现在反斜杠后面的反引号结束命令替换。如果使用了 `$(命令)` 的形式，则括号中间的所有字符组成命令，没有任何一个会被特殊处理。命令替换可以嵌套。使用反引号形式进行嵌套时，里面的反引号需要用反斜杠转义。如果命令替换出现在双引号之间，则其结果不会进行单词拆分和文件名扩展。

§3.5.5 算术扩展

算术扩展可以对算术表达式求值并替换成的求值的结果。它的格式是：

```
$(( 表达式 ))
```

处理表达式时，就好像它在双引号之间，但圆括号中间的双引号不会被特殊处理。表达式中的所有符号都会进行参数扩展，命令替换和引用去除。算术表达式也可以嵌套。求值时按照后面要介绍的规则 (参见§6.5[Shell 的算术运算], *p61*) 进行。如果表达式是无效的，Bash 就会在标准错误输出中打印一条错误信息，并且不会进行任何替换。

§3.5.6 进程替换

如果系统支持命名管道 (fifo) 或能够以“/dev/fd”方式来命名打开的文件，则也就支持进程替换。进程替换的语法形式是下面任意一种

```
<( 命令列表 )
```

或者

```
>( 命令列表 )
```

在运行时，进程的命令列表的输入和输出与一个命名管道或者“/dev/fd”目录里面的某个文件相关联。扩展的结果是，该文件的名称作为一个参数转递给当前的命令。如果使用了 >(命令列表) 这种形式，对该文件的写入就为命令列表提供了输入。如果使用了 <(命令列表) 这种形式，在需要得到命令列表的输出时，应该去读取作为参数转递的文件。注意，< 或 > 与左边括号之间不能有任何空格，否则这种结构就会被解释成重定向。

如果系统支持，进程替换就和参数及变量扩展、命令替换、还有算术扩展同时进行。

§3.5.7 单词拆分

在单词拆分时，shell 会扫描参数扩展、命令替换和算术运算的结果，如果它们不是在双引号之间进行的。

Shell 会把 $IFS 中的每个字符都当成分隔符，并按照这些字符把其它扩展的结果拆分成单词。如果 $IFS 没有设置，或者它的值和默认的 <space><tab><newline> 完全一样，那么在前述扩展的结果中开头或结尾出现的由 <space>、<tab>、或者 <newline> 构成的序列就会被忽略；而由 $IFS 中任意字符组成的序列如果不是出现在头部或尾部就成为单词的分隔符。如果 $IFS 的值不是默认的并且含有由空格和制表符这两个空白符 (称为 $IFS 分隔符) 构成的空白符，则如果在单词的头部或尾部出现就会被删除。$IFS 中除了空白符以外的任何字符，包括与其毗邻的空白符，就成为字段的分隔符；由 $IFS 空白符组成的序列也是分隔符。如果 $IFS 的值为空，则不拆分单词。

明确表示的空参数 ("" 或 '') 会被保留下来。由没有设置值的参数扩展后得到的未被引用的隐含空参数会被删除。如果没有设置值的参数在双引号之间扩展，则结果的空值会被保留。

注意，如果没有进行扩展，则也不会进行单词拆分。

§3.5.8 文件名扩展

单词拆分以后，Bash 会在每个单词中搜索字符“*”、“?”、和“[”，除非打开了“-f”选项 (参见§86[内部命令 set], *p42*)。如果找到其中一个，则把这个单词当作一个模式，并把与之匹配的文件名按字母顺序排列来取代它。如果没有找到匹配的文件名，并且禁止了 shell 的 nullglob 选项，则不处理该单词；如果打开了 nullglob 选项并且没有找到匹配的文件名，这个单词就会被删除。如果打开了 failglob 选项并且没有找到匹配的文件名，则打印一条错误信息，并且不执行当前命令。如果打开了 nocaseglob 选项，则匹配时不区分字母字符的大小写。如果一个模式用于生成文件名，则对于文件名开头或紧跟在斜杠后面的“.”必须明确匹配，除非打开了 shell 的 dotglob 选项。匹配文件名时，斜杠也必须明确匹配。在其它情况下，“.”不会特殊处理。关于 nocaseglob、nullglob、failglob、以及 dotglob 选项，请参见§87[内部命令 shopt], *p47*。

Shell 变量 GLOBIGNORE 可以用来限制匹配的文件名。如果设置了 GLOBIGNORE，并且匹配的文件名中又与 GLOBIGNORE 中的模式匹配的，将会从列表中删除。如果设置了 GLOBIGNORE，并且它的值不为空，则文件名“.”和“..”总是被忽略。但同时，给 GLOBIGNORE 设置一个非空的值会让 shell 的 dotglob 选项也生效，使得所有以“.”开头的文件名也会被匹配。为了像以往一样忽略以“.”开头的文件名，可以让“.*”成为 GLOBIGNORE 的模式之一。如果没有设置 GLOBIGNORE，则 dotglob 也会被取消。

§3.5.8.1 模式匹配

除了下面介绍的特殊模式字符，模式中出现的任何其它字符都与其自身相匹配。模式中不能使用 nul 字符。反斜杠可以用来转义其后面的字符；用于转义的字符在匹配时会被忽略。特殊模式字符必须转义以后才能按照字面含义匹配。

特殊模式字符有如下含义：

`*` 匹配任何字符串，包括空字符串。如果打开了 shell 的 globstar 选项，并且在文件名扩展的行文中使用了“*”，连续使用两个“*”字符的模式会匹配所有文件以及零个或多个文件夹和子文件夹。如果连续两个“*”并且后面有个“/”，那么它只匹配文件夹和子文件夹。

`?` 匹配任意单个字符。

`[···]` 匹配方括号中的任一字符。由连字符分隔的一对字符是一个“范围表达式”；在当前语言区域 (locale) 和字符集中，按顺序在这两个字符之间的任一字符，包括这两个字符本身，都会被匹配。如果“[”之后的第一个字符是“!”或者“^”，则匹配没有出现在方括号中的任一字符。如果要匹配“-”，可以把它放在方括号中第一个或最后一个位置。如果要匹配“]”，可以把它放在方括号中第一个位置。范围表达式中字符地排列顺序是由当前的语言和 shell 变量 LC_COLLATE (如果已经设置) 决定的。

例如，在默认的 C 语言区域中，“[a-dx-z]”和“[abcdxyz]”是等价的。在许多语言区域中，字符都是按词典顺序排列的；在这些语言区域中，“[a-dx-z]”和“[abcdxyz]”通常是不等价的，例如，它可能与“[aBbCcDdxXyYz]”等价。为了方括号表达式中使用在传统意义上的范围，可以把环境变量 LC_COLLATE 或 LC_ALL 设为“C”以强制使用 C 语言区域。

在“[”和“]”中间，可以用 [:类别:] 这样的语法形式来指定“字符类别”；其中的类别是 POSIX 标准中定义的下列类别之一：

alnum	匹配所有字母和数字
alpha	匹配所有字母
ascii	匹配所有 (ASCII) 字符
blank	匹配所有空白符
cntrl	匹配所有控制字符 (即 ASCII 中的二十个字符)
digit	匹配所有的数字 (*0-9*)
graph	匹配所有可显示字符 (可打印字符中，空格和退格符不可显示)
lower	匹配所有小写字母
print	匹配可打印字符 (非控制字符都可打印)
punct	匹配所有标点符号
space	匹配空格
upper	匹配所有大写字母
word	匹配单词里面的字符 (大小写字母)
xdigit	匹配所有十六进制数字 (*0-9* 和 *A-F*)

字符类别和该类别中的任一字符匹配。字符类别 word 匹配字母，数字和下划线。在“[”和“]”中间，还可以用 [=c=] 这样的语法形式来指定一个等价的字符类，它匹配当前语言中区域为 c 的所有字符。在“[”和“]”中间，[.名称.]这样的语法形式和名称为名称的语言区域相匹配。

如果用内部命令 shopt 打开了 shell 的 extglob 选项，Bash 就可以识别几个扩展的模式匹配运算符。在下面的叙述中，模式列表是由“|”分隔的一个或多个模式。可以使用下面的一个或多个子模式构成复合模式。

`?(模式列表)` 与模式列表匹配零次或一次。

`*(模式列表)` 与模式列表匹配零次或多次。

`+(模式列表)` 与模式列表匹配一次或多次。

`@(模式列表)` 与模式列表中的模式之一匹配。

`!(模式列表)` 与模式列表中的任一模式之外的字符匹配。

§3.5.9 引用去除

经过上述扩展以后，对于所有没有被引用的字符“\”、“'”、以及 "，如果它们不是由上述任何一种扩展产生的，就会被删除。

§3.6 重定向

在执行命令之前，shell 可能会用特殊的方式把它的输入和输出重新定向。重定向还可以用来在当前的 shell 执行环境中打开和关闭文件。下面讲的重定向运算符可以出现在一个简单命令的前面或中间的任何地方，也可以出现在命令的后面。重定向按照出现的顺序从左到右处理。

在下文中，如果省略了文件描述符，并且重定向运算符的第一个字符是“<”，则指的是重定向标准输入 (文件描述符为 *0*)；重定向运算符的第一个字符是“>”，则指的是重定向标准输出 (文件描述符为 *1*)。

在下文中，重定向运算符后的单词，如果没有特别说明，要进行大括号扩展、波浪号扩展、参数扩展、命令替换、算术扩展、引用去除、文件名扩展、以及单词拆分。如果扩展后产生多个单词，Bash 就会报错。

注意，重定向的顺序是很重要的。例如，命令

```
ls > 目录列表 2>&1
```

会把标准输出 (文件描述符为 *1*) 和标准错误输出 (文件描述符为 *2*) 重定向到文件目录列表中，而命令

```
ls 2>&1 > 目录列表
```

仅把标准输出重定向到文件目录列表中，因为在标准输出被重定向到文件目录列表中之前，标准错误输出已经被复制到标准输出中了。

在进行重定向时，Bash 会对下表列出的几个文件特殊处理：

`/dev/fd/fd` 如果fd是个有效的整数，则复制文件描述符fd。

`/dev/stdin` 复制文件描述符 *0*。

`/dev/stdout` 复制文件描述符 *1*。

`/dev/stderr` 复制文件描述符 *2*。

`/dev/tcp/主机名/端口号` 如果主机名是个有效的主机名称或因特网地址，并且端口号是整数型的端口号或服务名称，Bash 会试图打开一个到相应套接字端口的 TCP 连接。

`/dev/udp/主机名/端口号` 如果主机名是个有效的主机名称或因特网地址，并且端口号是整数型的端口号或服务名称，Bash 会试图打开一个到相应套接字端口的 UDP 连接。

如果不能打开或者创建文件，重定向就会失败。要谨慎使用比 *9* 大的文件描述符进行重定向，因为它们可能会和 shell 内部使用的文件描述符相冲突。

§3.6.1 输入重定向

输入重定向会打开单词扩展后所形成的文件名以备读取，并将其作为文件描述符 *n*；如果没有指定 *n* 则将其作为标准输入 (文件描述符为 *0*)。输入重定向的一般格式是：

```
[n]<单词
```

§3.6.2 输出重定向

输出重定向会打开单词扩展后所形成的文件名以备写入，并将其作为文件描述符 n；如果没有指定 n 则将其作为标准输出 (文件描述符为 *1*)。如果文件不存在，就首先创建它；如果已存在，就将它清空使得长度为零。输出重定向的一般格式是：

```
[n]>[|]单词
```

如果重定向运算符是“>”，并且已经打开了内部命令 `set` 的 `noclobber` 选项，则当单词扩展后所形成的文件名且是一个普通文件时，重定向将失败。如果重定向运算符是“>|”，或者重定向运算符是“>”且没有打开 `noclobber` 选项，则即使单词扩展后所形成的文件名存在，重定向也会进行。

§3.6.3 输出重定向的追加

这种类型的输出重定向会打开单词扩展后所形成的文件名以备追加，并将其作为文件描述符 n；如果没有指定 *n* 则将其作为标准输出 (文件描述符为 *1*)。如果文件不存在，就首先创建它。追加输出重定向的一般格式是：

```
[n]>>单词
```

§3.6.4 输出和错误输出重定向

这种结构把标准输出 (文件描述符为 *1*) 和标准错误输出 (文件描述符为 *2*) 同时重定向到单词扩展后所形成的文件中。标准输出和标准错误输出重定向的形式有两种：

```
&>单词
```

和

```
>&单词
```

这两种形式中，优先使用第一种。它们和下面的形式在语法上是等价的：

```
>单词 2>&1
```

§3.6.5 输出和错误输出重定向的追加

这种结构把标准输出 (文件描述符为 *1*) 和标准错误输出 (文件描述符为 *2*) 同时追加重定向到单词扩展后所形成的文件中。标准输出和标准错误输出重定向的追加用下面的形式：

```
&>>单词
```

这和下面的形式在语法上是等价的：

```
>>单词 2>&1
```

§3.6.6 即插即用文本

这种形式的重定向指示 shell 从当前文本源中读取输入，直到遇到其中一行只包含单词 (结尾不含空白符)。这时，已经读取的所有行都被当作命令的标准输入。即插即用文本的格式是：

```
<<[-]单词
    即插即用文本
结束符
```

单词不会进行参数扩展，命令替换，算术扩展，或文件名扩展。如果单词中任一字符被引用，则结束符是单词进行引用去除后的结果，这时不会对即插即用文本进行扩展。如果单词没有被引用，则即插即用文本中的所有行都会进行参数扩展、命令替换、和算术扩展；这时，字符序列 \newline就会被忽略，并且只能用“\”来引用字符“\”、“$”和“`”。

如果重定向运算符是“<<-”，则输入行和结束符所在行中所有的在行开头的制表符都会被删除。这样，在 shell 脚本中的即插即用文本就能够自然的缩进。

§3.6.7 即插即用字符串

即插即用文本的一种变体，其形式是

```
<<< 单词
```

扩展单词并作为标准输入提供给命令。

§3.6.8 文件描述符的复制

重定向运算符

```
[n]<&单词
```

用来复制输入文件描述符。如果单词扩展后是一个或多个数字，则文件描述符 n 就是与这个数字对应的文件描述符的拷贝。如果单词中的数字没有指定要打开并读取的文件描述符，就会发生重定向错误。如果单词扩展后的值是“-”，就会关闭文件描述符 n。如果没有指定 n，则使用标准输入 (文件描述符为 *0*)。

运算符

```
[n]>&单词
```

与之相似，用来复制输出文件描述符。如果没有指定 n，则使用标准输出 (文件描述符为 *1*)。如果单词中的数字没有指定要打开并写入的文件描述符，就会发生重定向错误。 作为特殊情况，如果省略 n，并且单词扩展后不是一个或多个数字，则标准输出和标准错误输出就会按前面说的规则重定向。

§3.6.9 文件描述符的移动

重定向运算符

```
[n]<&数字-
```

把文件描述符数字转移到文件描述符 n 上；如果没有指定 n，则转移到标准输入 (文件描述符为 *0*) 上。转移到 n 后， (文件描述符) 数字就会被关闭。

类似的，重定向运算符

```
[n]>&数字-
```

把文件描述符数字转移到文件描述符 n 上；如果没有指定 n，则转移到标准输出 (文件描述符为 *1*) 上。

§3.6.10 打开文件描述符以备读出和写入

重定向运算符

```
[n]<>单词
```

可以打开单词扩展后的文件名以同时准备读取和写入，其文件描述符为 n；如果没有指定 n，则使用文件描述符 *0*。如果文件不存在，则首先创建它。

§3.7 命令的执行

§3.7.1 简单命令的扩展

执行简单命令时，shell 会从左到右地进行下列扩展，赋值和重定向：

1. 被分析器当作 (在命令名称之前的) 变量赋值和重定向的单词，将被保存下来以备后续处理。
2. 不是变量赋值和重定向的单词会被 (参见§3.5[Shell 扩展], *p14*)。如果扩展以后还有单词，则其中的第一个会被当作命令的名称，剩余的当作该命令的参数。
3. 进行前面所说的重定向 (参见§3.6[重定向], *p21*)。
4. 每个变量赋值语句中“=”后面的文本在赋给变量之前会进行大括号扩展、参数扩展、命令替换、算术扩展和引用去除。

如果结果没有产生命令名称，则变量赋值会影响到当前的 shell 环境；否则，变量被加到命令的执行环境中，而不会影响当前的 shell 环境。如果某个赋值语句试图给只读变量赋值，就会发生错误，命令就会退出并返回一个非零的状态。

如果结果没有产生命令名称，也会进行重定向，但是不会影响到当前的 shell 环境。如果重定向发生错误，命令就会退出并返回一个非零的状态。

如果扩展以后产生了命令名称，则按如前所说的步骤执行；否则，命令结束。如果某个扩展含有命令替换，则命令的退出状态是替换中所执行的最后一个命令的退出状态。如果没有命令替换，则结束命令并返回状态零。

§3.7.2 命令的搜索和执行

命令拆分成单词以后，结果就得到一个简单命令及其可选的参数列表。这时，执行下面的操作。

1. 如果命令名中不含斜杠，shell 试图找到它。如果有一个同名的 shell 函数，则如§3.3[Shell 函数], *p12* 所说的执行该函数。
2. 如果这个名称不是函数，shell 会在内部命令列表中搜索它。如果找到，则执行该内部命令。
3. 如果这个名称既不是函数名，也不是内部命令，并且不含有斜杠，Bash 会搜索 `$PATH` 中每个目录里面同名的可执行文件。Bash 使用一个散列表来保存可执行文件的绝对路径 (参见§4.1[波恩 Shell 的内部命令 `hash`], *p31*)，以避免重复搜索 `$PATH` 路径。只有当命令不在散列表中时才去搜索 `$PATH` 中的所有目录。如果搜索失败，shell 就会去找一个叫 `command_not_found_handle` 的 shell 函数定义。如果这个函数存在，就调用这个函数，并把原来的命令及其参数当作参数传递给它；这个函数的就成为 shell 的退出状态。如果这个函数也没有定义，shell 会打印一条错误信息并返回状态 *127*。
4. 如果搜索成功，或者命令名中含有一个或多个斜杠，shell 就会在独立的执行环境中执行这个命令。参数 *0* 设为指定的命令名，如果有剩余的参数，将传作为参数 递给命令。
5. 如果因为文件格式和可执行文件不同导致执行失败，并且该文件不是一个目录名称，它就会被当成一个 shell 脚本并按照§3.8[Shell 脚本], *p26* 中所说的方法执行。
6. 如果命令不是异步执行的，shell 会等待它的结束并收集其返回状态。

§3.7.3 命令执行的环境

Shell 的执行环境包括下列内容：

- Shell 启动时，打开它所继承的文件；这些文件可以用内部命令 `exec` 来重定向
- 当前工作目录；它是由 `cd`，`pushd` 或 `popd` 设置的，或者 shell 启动时就继承的
- 创建文件所用的掩码；它是由 `umask` 设置，或者继承自 shell 的父进程
- 当前的陷阱，由 `trap` 设置
- Shell 参数；它是通过变量赋值或者 `set` 设置的，或者从 shell 的父进程环境中继承来的
- Shell 函数；它是在执行期间定义的，或者从 shell 的父进程环境中继承来的
- 启动时 (默认的或通过命令行参数) 或通过 `set` 打开的选项
- 用 `shopt` (参见§4.3.2[内部命令 shopt], *p45*) 打开的选项
- 用 `alias` (参见§6.6[别名], *p62*) 定义的 shell 别名
- 各个进程号，包括后台作业 (参见§3.2.3[命令队列], *p7*)，以及 `$$` 和 `$PPID` 的值

要执行一个不是内部命令或者 shell 函数的简单命令时，它会在包含下列内容的单独环境中执行。除非特别说明，下面的值都从 shell 中继承。

- Shell 打开的文件，包括命令重定向时所作的修改和增加
- 当前工作目录
- 创建文件所用的掩码
- 标志为可以导出的 shell 变量和函数，以及命令从环境 (参见§3.7.4[环境], *p25*) 中导入的变量。
- Shell 捕获的陷阱被重置为从 shell 父进程中继承来的值；shell 忽略的陷阱也会被忽略

在这种单独的环境里启动的命令不会影响 shell 的执行环境。

命令替换，组合在括号中的命令，以及异步命令都在子 shell 环境中；子 shell 复制了 shell 环境，不同的是 shell 捕获的陷阱被重置为从 shell 父进程中继承来的值。属于管道中一部分的内部命令也在子 shell 环境中执行。对子 shell 环境的改变不会影响到 shell 的执行环境。

用来执行命令替换的子 shell 会继承父 shell 的“`-e`”选项。如果不是在 POSIX 模式下，Bash 会在这些子 shell 中去掉“`-e`”选项。

如果启用了作业控制，并且命令后面有个“`&`”，则命令的默认标准输入是空文件“`/dev/null`”；否则，命令将会继承 shell 的重定向后的文件描述符。

§3.7.4 环境

在一个程序启动时，它会得到一个字符串数组，这就叫做环境。这个数组有一列成对的名称和值，即“名称=值”。

Bash 提供了几种操纵环境的方法。启动时，shell 会检查自己的环境，每找到一个名称就创建一个变量，并自动标志这个变量，使得它可以导入到子进程中。执行的命令都继承这个环境。`export` 命令 和“`declare -x`”可以在环境中增加和删除变量或函数。在环境中，如果一个参数的值被修改，这个新的值取代旧的值就会成为环境的一部分。执行的命令所继承的环境包括 shell 的初始环境，这些值可能在 shell 中被修改了，其中不包括用 `unset` 或 “`export -n`” 删除了的，但包含 `export` 和 “`declare -x`” 命令所增加的部分。

简单变量或函数的环境都可以通过加上参数赋值的前缀来暂时修改 (参展§5[Shell 变量], *p48*)。这些赋值语句只影响该命令所能访问到的环境。

如果打开了“`-k`”选项 (参见§86[内部命令 set], *p42*)，则所有赋值的变量都被加入到命令的环境中，而不仅仅是命令名前面的那些变量。

当 Bash 启动外部命令时，`$_` 就会被设置为这个命令的绝对路径，并传递到变量的环境中。

§3.7.5 退出状态

命令执行以后，它的退出状态就是系统调用 waitpid 或与其等价的函数所返回的值。返回状态总是介于 *0* 和 255 之间，具体将在下面解释；*125* 以上的值使用比较特殊。Shell 的内部命令和复合命令的退出状态也是位于这个范围。在某些情况下，shell 将使用特殊的状态来表示具体的错误状态。

为了便于 shell 处理，对成功执行的命令，它的退出状态是零；而非零的退出状态表示失败了。使用这种看起来不直观的方法是因为，这样就可以很好的区分成功和多种不同的失败状态。如果命令接收到一个值为 *N* 的关键信号而退出，Bash 就会把 128+N 作为它的退出状态。

如果命令没有找到，用来执行它的子进程就会返回状态 127。如果命令跃然找到但却不是可执行的，就返回状态 126。

如果命令因为扩展或重定向时发生错误而失败，则它的返回状态比零大。

Bash 的条件命令 (参见§3.2.4.2[条件结构], *p9*) 以及部分命令队列 (参见§3.2.3[命令队列], *p7*) 使用了退出状态。

Bash 的所有内部命令都会在成功时返回状态零，失败时返回非零，所以它们可以用于条件命令和命令队列中。如果使用不正确，所有的内部命令都会返回状态 *2*。

§3.7.6 信号

如果 Bash 是交互运行的，并且没有任何陷阱，它就会忽略 SIGTERM (所以“kill 0”不会杀死一个交互式的 shell)，但会捕获并处理 SIGINT (所以内部命令 wait 是可以中断的)。当 Bash 接收到 SIGINT 时，会退出任何正在进行的循环。在任何情况下，Bash 都会忽略 SIGQUIT。如果启用了作业控制 (参见§7[作业控制], *p69*)，Bash 会忽略 SIGTTIN、SIGTTOU、SIGTSTP。

Bash 启动的非内部命令会使用 shell 从其父进程继承的信号处理程序。如果没有启用作业控制，异步执行的命令除了有这些信号处理程序，还会忽略 SIGINT 和 SIGQUIT。因为命令替换而运行的命令会忽略键盘产生的作业控制信号 SIGTTIN，SIGTTOU 和 SIGTSTP。

默认情况下，shell 接收到 SIGHUP 后会退出。在退出之间，交互运行的 shell 会向所有的作业，不管是正在运行还是已经停止，重新发送 SIGHUP。对已经停止的作业，shell 还会发送 SIGCONT 以确保它能够接收到 SIGHUP。为了阻止 shell 向某个特定的作业发送 SIGHUP 信号，可以用内部命令 disown (参见§7.2[作业控制内部命令], *p70*) 将它从作业表中删除，或者用 disown -h 让它下接收 SIGHUP。

如果使用 shopt (参见§87[内部命令 shopt], *p46*) 打开了 shell 的 huponexit 选项，当一个交互运行的登录 shell 退出时，会向所有的作业发送 SIGHUP。

如果 Bash 在等待命令结束时接收到了一个信号，并且已经给这个信号设置了陷阱，则该陷阱直到命令结束时才会执行。如果 Bash 通过内部命令 wait 在等待一个异步命令时接收到了一个信号，并且已经给这个信号设置了陷阱，则内部命令 wait 在执行完该陷阱后将立即返回一个大于 *128* 的状态。

§3.8 Shell 脚本

Shell 脚本是包含 shell 命令的文本文件。如果 Bash 启动时把这个文件用作为第一个不是选项的参数，并且没有使用“-c”或“-s”选项 (参见§6.1[Bash 的启动], *p55*)，Bash 会从该文件中读取命令并执行，然后退出。这种方式会产生一个非交互运行的 shell；这个 shell 会首先在当前目录中搜索该文件，如果没有找到就会继续搜索 $PATH 中的目录。

Bash 运行 shell 脚本时，会把特殊参数 *0* 设为该脚本的名称，而不是 shell 的名称。如果还有其它参数，就把其余的参数当作位置参数；如果没有其它参数，就不设置位置参数。

可以用 chown 命令打开 shell 脚本的执行位使得它成为可执行文件。如果 Bash 在 $PATH 中搜索命令时找到了这个文件，就会产生一个子 shell 来执行它。换句话说，如果文件名是一个可执行的 shell 脚本，则执行

```
文件名 参数
```

就相当于执行

```
bash 文件名 参数
```

这个子 shell 会重新初始化，效果就等同于启动了一个新的 shell 来执行该脚本；所不同的是，父进程存储的各命令的路径 (参见§4.1[波恩 Shell 的内部命令 hash], *p31*) 将会在子进程中沿用。

大多数 Unix 版本在执行命令时都包含这样的机制：如果脚本的第一行由“#!”这两个字符开头，则这一行其余的内容就指定该脚本的解释器。因此，可以指定 Bash、awk、Perl、或者其它解释器，并在脚本的这行后面用这些语言来写。

在脚本文件的第一行，解释器名称后面可以包含一个单一的选项，以作为该解释器的参数；后面是脚本的名称，再后面是其余参数。在不能处理这些参数的操作系统里面，Bash 会处理它们。注意，有些比较老的 Unix 版本规定，解释器名称和其参数总长度不得超过 *32* 个字符。

Bash 脚本的开头通常是 `#!/bin/bash` (假定 Bash 安装在“/bin”下面)，因为这样能保证该脚本用 Bash 来解释，即使它在其它 shell 下执行。

第四章 Shell 内部命令

内部命令是由 shell 自身提供的。如果某个内部命令的名称是一个简单命令 (参见§3.2.1[简单命令], *p6*) 的第一个单词，shell 会直接执行这个命令，而不会启动其它程序。对于一些不可能或者不方便通过外部程序实现的功能，内部命令是非常必要的。

本章简要介绍了 Bash 从波恩 shell 继承的内部命令，以及 Bash 扩展过的独特内部命令。还有几个内部命令在其它章节中介绍：Bash 对作业控制功能 (参见§7.2[作业控制内部命令], *p70*) 提供的界面，目录栈 (参见§6.8[目录栈], *p63*)，`history` 命令 (参见§9.2[Bash 历史内部命令], *p90*)，以及可编程补全功能 (参见§8.7[可编程补全的内部命令], *p87*)。很多内部命令在 POSIX 或者 Bash 中都得到了扩展。

下面介绍的内部命令，除非特别说明，如果说它接受“-”开头的选项，则“--”就表示选项的结束。例如，内部命令 `:`、`true`、`false` 和 `test` 不接受选项。

§4.1 波恩 Shell 的内部命令

下面的内部命令是从波恩 shell 继承来的；它们是根据 POSIX 规范现实的。

A. : (逗号)

```
: [参数]
```

除了扩展参数和执行重定向外不做任何操作。返回状态是零。

B. . (点号)

```
. 文件名 [参数]
```

在当前的 shell 环境中，从文件名中读取并执行命令。如果文件名不包括斜杠，则使用 `PATH` 变量去搜索文件。如果 Bash 不是在 POSIX 模式下运行，则在 `$PATH` 中找不支后就会在当前目录中搜索。如果提供了参数，它们就成为执行文件名时的位置参数；否则，位置参数不会被改变。返回状态是最后一个被执行命令的退出状态；如果没有命令被执行，则返回零。如果文件名没有找到，或者不能读取，返回状态就是非零值。该命令和 `source` 是等价的。

C. break

```
break [n]
```

从 `for`，`while`，`until` 或 `select` 循环中退出。如果给定 n，则退出外围的第 n 层循环。n 必须大于或等于 *1*。返回状态是零，除非 n 不是大于或等于 *1*。

D. cd

```
cd [-L|-P] [目录]
```

把当前工作目录改为目录；如果目录没有指定，则使用 shell 变量 $HOME 的值。如果 shell 变量 CDPATH 存在，则用它作为搜索路径。如果目录以斜杠开头就不再使用 CDPATH。“-P”选项助记词：Physical，物理路径的意思是不跟踪符号链接；而默认情况下，或使用了“-L”助记词：symLink，符号链接会跟踪符号链接。如果目录是“-”，就相当于 $OLDPWD。如果使用 CDPATH 中一个非空的目录，或者“-”上第一个参数，并且改变目录成功，则在标准输出中打印新的工作目录的绝对路径。如果改变目录成功，返回状态就是零；否则就是非零。

✻ 代码清单 3：改进 cd 实现目录导航

```
#!/bin/bash

# 改进 Bash 中的 cd 命令。
# 另请参见 dirs，popd 和 pushd 命令。
# 作者 Jerry Fleming <jerryfleming2006@gmail.com>，2009 年 8 月

cd()
{
    if [[ $1 =~ ^([+-])([0-9]*)$ ]]
    then
        CDR=$( expr $CDR ${BASH_REMATCH[1]} ${BASH_REMATCH[2]:-1} )
        [[ $CDR -lt 1 ]] && CDR=1
        [[ $CDR -gt ${#CDP[@]} ]] && CDR=${#CDP[@]}
        I=$(($CDR-1))
        builtin cd ${CDP[$I]}
    elif [[ $1 =~ ^[+][+]([0-9]*)$ ]]
    then
        CDR=${BASH_REMATCH[1]:-1}
        [[ $CDR -lt 1 ]] && CDR=1
        [[ $CDR -gt ${#CDP[@]} ]] && CDR=${#CDP[@]}
        I=$(($CDR-1))
        builtin cd ${CDP[$I]}
    elif [[ $1 == -i ]]
    then
        unset CDP
        CDP[0]=$PWD
        CDR=1
    elif [[ $1 == -l ]]
    then
        I=0
        for item in ${CDP[@]}
        do
            I=$(($I+1))
            echo -n [$I]
            if [[ $I -eq $CDR ]]
            then
                echo -n '*'
            else
                echo -n ' '
            fi
```

```
41              echo -n ' '
42              echo $item
43          done
44      else
45          builtin cd $@
46          CDP[$CDR]=$PWD
47          CDR=$(($CDR + 1))
48      fi
49  }
```

E. continue

```
continue [n]
```

继续执行外围的 for，while，until 或 select 的下一次循环。如果指定 n，则继续执行外围第 n 层的循环。n 必须大于或等于 *1*。返回状态是零，除非 n 不是大于等于 *1*。

F. eval

```
eval [参数表]
```

把参数表中的参数连在一起形成一个命令，然后读取并执行这个命令，它的退出状态就是 eval 的退出状态。如果没有参数表，或者参数表为空，则退出状态为零。

G. exec

```
exec [-cl] [-a 名称] [命令 [参数表]]
```

如果指定了命令，它就会取代 (当前的) shell 而不是创建新的进程。如果给定了“-l”选项助记词：Login，登录，shell 就会在传给命令的第零个参数前加上一个短横。这就是登录程序所做的。“-c”选项助记词：Clear，清除使得命令在空的环境中执行。如果指定“-a”助记词：Alter，改变，shell 会把名称作为第零个参数传给命令。如果没有指定命令，可以用重定向来影响当前的 shell 环境。如果没有重定向错误，返回状态就是零，否则就是非零。

H. exit

```
exit [n]
```

退出 shell，并在 shell 的父进程中返回状态 n。如果省略了 n，则退出状态是最后被执行的命令的退出状态。EXIT 陷阱是在 shell 结束前执行的。

I. export

```
export [-fn] [-p] [名称[=值]]
```

把每个名称都传到子进程的环境中。如果给定了“-f”选项助记词：Function，函数，则名称就是 shell 函数；否则，它是 shell 变量。“-n”选项助记词：No，不表示不再把名称导出到子进程。如果没有给定名称，或者给定了“-p”选项助记词：Pretty-Print，格式化打印，则显示已经导出的名称列表。“-p”选项能够把输出格式化成可以重新作为输入的形式。如果名称后面是“=值”的形式，那么这个值就会赋给名称。

这个命令的返回状态是零，除非指定了无效的选项，或者其中一个名称不是有效的 shell 变量名，或者“-f”指定的不是一个 shell 函数的名称。

J. getopts

```
getopts 选项字符串 名称 [参数表]
```

Shell 脚本用 getopts 来分析位置参数。选项字符串包含待识别的选项字符；如果一个字符后面有个冒号，这个选项就要接受一个参数，这个参数和选项之间用空格分开。选项字符不可以是冒号 (“:”) 或问号 (“?”)。每次使用时，getopts 都会把下一个选项放在 shell 变量名称中 (如果名称不存在就初始化它)，并把下一个要处理的参数的下标放在变量 OPTIND 中。每次启动当前的 shell 或 shell 脚本时，都把 OPTIND 初始化为 *1*。如果某个选项需要参数，getopts 就把该参数放在变量 OPTARG 中。Shell 不会自动把 OPTIND 重置；如果在同一个 shell 中多次调用 getopts 时要使用新的参数，则必须手动重置它。

处理完选项结尾时，getopts 会退出并返回一个大于零的状态；OPTIND 会指向第一个非选项参数的下标；而名称被设为“?”。

getopts 参演会分析位置参数，但如果参数表中的参数更多，getopts 就会转而分析参数表。

getopts 可以通过两种方式报错。如果选项字符串的第一个字符是冒号，则忽略报错。正常情况下，如果遇到无效选项，或者忽略了选项的参数，就会打印诊断信息。如果把变量 OPTERR 设为 *0*，则不会打印错误信息，即使选项字符串的第一个字符不是冒号。

如果遇到了无效的选项，getopts 就会在名称里面放入“?”。这时如果不忽略报错，就打印一条错误信息并重置 OPTARG；如果忽略报错，就把该选项放在 OPTARG 中而不打印错误信息。如果某个必要参数没有找到，并且 getopts 没有忽略报错，则在名称中放入问号 (“?”) 并重置 OPTARG且打印错误信息。如果 getopts 忽略错误，则在名称中放入冒号 (“:”)，并把 OPTARG 设为找到的选项字符。

K. hash

```
hash [-r] [-p 文件名] [-dtl] [名称]
```

记住参数名称所指定的命令的完整路径，使得以后再启动这个命令时不需要再搜索它。命令是通过搜索 $PATH 中列出的目录而找到的。“-p”选项助记词：Path，指定路径可以禁止路径搜索，并用文件名指定命令的位置。“-r”选项助记词：Redo，重做使得 shell 忘记所有已经记住的位置。“-d”选项助记词：Destination，目标使得 shell 忘记它已经记住的名称的每个位置。如果指定了“-t”选项助记词：prinT，打印，则打印每个名称对应的完整路径；如果“-t”指定了多个名称参数，则打印完整路径时还在前面打印记住的命令名称。“-l”选项助记词：List，列出把输出格式化成可以重新作为输入的形式。如果没有给定参数，或者只给定了“-l”，则打印与每个所记住命令的有关信息。返回状态是零，除非没有找到名称，或者遇到了无效的选项。

L. pwd

```
pwd [-LP]
```

打印出当前工作目录的绝对路径。如果给定了“-P”选项助记词：Physical，物理路径，则打印的路径中不会包含符号链接。如果给定了“-L”选项助记词：symLink，符号链接，则打印的路径中可能包含符号链接。返回状态是零，除非在计算当前工作目录时遇到了错误，或者指定了无效的选项。

M. readonly

```
readonly [-aApf] [名称[=值]] ···
```

把每个名称标志为只读。后续语句就不可以更改这些名称的值。如果指定了“-f”助记词：Function，函数，则每个名称都是指 shell 函数。“-a”选项助记词：Array，数组的意思是每个名称都是下标数组变量；“-A”选项助记词：AArray，名称数组的意思是每个名称都是键值数组变量。如果没有给定参数名称，或者给定了“-p”选项助记词：Print，打印，则打印所有只读的名称。“-p”选项能够把输出格式化成可以重新作为输入的形式。如果变量名称后面是“=值”的形式，变量名称的值就会被设为值。返回状态是零，除非遇到了无效的选项，某个参数名称是无效的 shell 变量或函数名，或者“-f”选项指定了一个不是 shell 函数的名称。

N. return

```
return [n]
```

使得 shell 函数退出并返回状态 n。如果没有指定 n，则返回状态是函数中最后一个被执行的命令的退出状态。这个命令也可以用来结束执行一个用内部命令 . 或 source 执行的脚本，并把 n 或者这个脚本中最后一个被执行的命令的退出状态作为脚本的退出状态。如果关联了 RETURN 陷阱，它就会参函数或脚本退出以后执行。如果在函数以外，或者在不是用 . 或 source 执行的脚本中使用，它的退出状态就是非零，

O. shift

```
shift [n]
```

把位置参数向左移动 n 个位置，位置参数 n+1···$# 被重命名为 $1···$#-n，而从 $# 到 $#-n+1 的位置参数被重置了。n 必须是个小于或等于 $# 的非负数。如果 n 是零，或者比 $# 大，则不会改变位置参数。如果没有指定 n，则默认为 1。返回状态是零；除非 n 比 $# 大或者小于零，这时返回零。

P. test 和 [

计算条件表达式表达式的值。运算符和运算数都要是独立的参数。表达式是由原子表达式 (参见 §6.4[Bash 条件表达式], *p59*) 组成的。test 不接受任何选项，也不能用参数“--”来表示选项的结束。如果使用 [的形式，则这个命令的最后一个参数必须是]。可以用下面的运算符把表达式连起来，这些运算符是按照优先级的降序排列的。求值结果依赖于参数的个数，如下所示。

! 表达式 如果表达式为假，则真。

(表达式) 返回表达式的值。这可以用来改变运算符的正常优先级。

表达式一 -a 表达式二 如果表达式一和表达式二都为真，则返回真。

表达式一 -o 表达式二 如果表达式一或者表达式二有一个为真，则返回真。

内部命令 test 和 [按照基于参数个数的规则，对表达式进行求值。

***0* 个参数 (没有参数)** 表达式是假的。

***1* 个参数** 当且仅当这个参数非空时，表达式的值为真。

2 个参数 如果第一个参数是“!”，则当且仅当第二个参数为空时表达式的值才为真。如果第一个参数是个单目条件运算符 (参见§6.4[Bash 条件表达式], p59)，则如果单目测试为真，表达式就为真。如果第一个参数不是个有效的单目运算符，则表达式为假。

3 个参数 如果第二个参数是双目运算符 (参见§6.4[Bash 条件表达式], p59)，则表达式的值是把第一和第三个参数作为运算数的双目测试的结果。如果有三个参数，则“-a”和“-o”就是双目运算符。如果第一个参数是“!”，则结果是把第二和第三个参数作为运算数的双目测试的结果。如果第一个参数是“(”且第三个参数是“)”，则结果是第二个参数的单目运算的结果。否则，表达式是假的。

4 个参数 如果第一个参数是“!”，则结果就和其余三个参数构成的表达式的值相反。否则根据上面列出的优先级和规则分析和计算表达式的值。

5 个参数 根据上面列出的优先级和规则分析和计算表达式的值。

Q. times

```
times
```

打印出 shell 及其子进程所使用的用户时间和系统时间。返回状态是零 (参见§2[管道], p7)。

R. trap

```
trap [-lp] [参数] [信号指示···]
```

当 shell 接收到信号指示中的信号时，就会读取和执行参数中指定的命令。如果省略了参数 (这时只有一个信号指示) 或者参数是“-”，则每个指定信号的处理都重置为 shell 启动时的值。如果参数是空字符串，则 shell 及它所启动的命令会忽略每个信号指示指定的信号。如果省略了参数并给定了“-p”选项助记词：Print，打印，则 shell 会显示每个信号指示所关联的 trap 命令。如果没有给定参数或者只给定了“-p”选项，trap 会把每个信号数字所关联的命令输出格式化成可以重新作为 shell 输入的形式。“-l”选项助记词：List，列出会使 shell 打印信号名称和它们所对应数字的列表。每个信号指示都是一个信号名称或者信号数字。信号名称对大小写不敏感，其前面的 SIG 可以省略。如果信号指示是 0 或者 EXIT，则当 shell 退出时就会执行参数。如果参数是 DEBUG，则每次执行简单命令、for 命令、case 命令、select 命令、算术 for 命令的每个算术表达式，以及 shell 函数的第一个命令之前，都会执行命令参数。关于内部命令 shopt 的 extglob 选项如何影响 DEBUG 陷阱的细节，请参见§87[内部命令 shopt], p46。如果信号指示是 ERR，则当一个简单命令因为下列原因返回非零的值时就执行命令参数。如果失败的命令是紧跟在关键词 until 或 while 后的命令队列的一部分，或者是在关键词 if 或 elif 后的测试命令的一部分，或者是 && 或 || 命令队列中所执行命令的一部分，或者该命令的返回状态经由 ! 反转，则不会执行 ERR 陷阱。errexit 选项也服从同样的条件。如果信号指示是 RETURN，则每当由内部命令 . 或 source 执行的 shell 函数或脚本执行结束时，都会执行命令参数。进入 shell 时就被忽略的信号不可以再用于陷阱或者被重置。当创建了子进程时，已经设了陷阱没有被忽略的信号将在子进程中被设为原来的值。

返回状态是零，除非信号指示指定的不是有效信号。

S. umask

```
umask [-p] [-S] [模式]
```

把 shell 进程的文件创建掩码设为模式。如果模式以数字开头，它就当八进制数解释；否则，它就被当作类似于 chmod 命令所接受的掩码模式符号。如果忽略了模式，则打印当前的掩码值。如果给定了“-S”选项助记词：Symbolic，模式符号却没有给定参数模式，则以符号形式打印掩码。如果给定“-p”选项助记词：Pretty-Print，格

式化打印并省略了模式，则把输出格式化成可以重新作为输入的形式。如果模式更改成功或者没有给定模式，则返回零；否则返回非零。

注意，如果模式当八进制数解释，则掩码中的每个数都要用 *7* 去减。所以，掩码为 *022* 就会得到 *755* 的权限。

T. unset

```
unset [-fv] [名称]
```

删除各个指定的变量或函数名称。如果没有指定选项，或者指定了“-v”选项助记词：Variable，变量，则每个名称都是指 shell 变量。如果指定了“-f”选项助记词：Function，函数，则每个名称都是指 shell 函数，这些函数的定义将被删除。只读变量和函数不可以删除。返回状态是零，除非名称是只读的。

§4.2 Bash 的内部命令

本节介绍在 Bash 中独特或者经过扩展的内部命令。这些命令中有些在 POSIX 标准中已经作了规范。

A. alias

```
alias [-p] [名称[=值] ···]
```

如果没有参数或者给定了“-p”选项助记词：Print，打印，alias 就会在屏幕上把别名列表以便于重新输入的形式打印出来。如果给定参数，则对于每个给定的名称，都用对应的值作为别名；如果没有给出值，则打印名称指定的别名。别名在§6.6[别名], *p62*中介绍。

B. bind

```
bind [-m 键映射] [-lpsvPSV]
bind [-m 键映射] [-q 函数] [-u 函数] [-r 键序列]
bind [-m 键映射] -f 文件名
bind [-m 键映射] -x 键序列:shell 命令
bind [-m 键映射] 键序列:函数名
bind Readline 命令
```

显示当前 Readline (参见 §8[编辑命令行], *p72*) 中键和功能函数的绑定，或者把键序列绑定到 Readline 函数或宏，或者设置 Readline 变量。每个非选项的参数都是可以用于 Readline 初始化文件 (参见§8.3[Readline 的启动脚本], *p74*) 中的命令，但是每个绑定和命令都必须单独作为参数；例如，“"\C-X\C-r":re-read-init-file”。如果给定了选项，就会有下面的含义：

-m 键映射 用键映射作为后续绑定所使用的键映射。可用的键映射名称包括 emacs、emacs-standard、emacs-meta、emacs-ctlx、vi、vi-move、vi-command、和 vi-insert；其中，vi 和 vi-command 是等价的，emacs 和 emacs-startdard 也是等价的 (参见§86[内部命令 set], *p43*)。助记词：Map，映射

-l 列出所有的 Readline 函数名称。助记词：List，列出

-p 以便于重新作为输入或者可以用于 Readline 初始化文件中的格式显示 Readline 函数名。助记词：Pretty-Print，格式化输出

-P 列出当前的 Readline 函数名和绑定。助记词：PPrint，打印

`-v` 以便于重新作为输入或者可以用于 Readline 初始化文件中的格式显示 Readline 变量名及其值。助记词：Variable，变量

`-V` 列出当前的 Readline 变量名和值。助记词：VVariable，变量

`-s` 以便于重新作为输入或者可以用于 Readline 初始化文件中的格式显示 Readline 中绑定到宏的键序列及其输出的字符串。助记词：Sequence，序列

`-S` 列出绑定到宏的键序列及其输出的字符串。助记词：SSequence，序列

`-f 文件名` 从文件名中读取键绑定。助记词：File，文件

`-q 函数` 查询绑定到指定函数的键。助记词：Query，查询

`-u 函数` 取消所有对函数的绑定。助记词：Unbind，取消绑定

`-r 键序列` 删除键序列的当前绑定。助记词：Remove，删除

`-x 键序列:shell 命令` 使得每次输入键序列都执行 shell 命令。当执行 shell 命令时，shell 会把变量 READLINE_LINE 设为 Readline 中的行缓存，把变量 READLINE_POINT 设为当前标志点的位置。如果执行的 shell 命令改变了 READLINE_LINE 或 READLINE_POINT，则其新的值将在编辑状态中反映出来。助记词：eXecute，执行

返回状态是零，除非给定了无效的选项或者发生了错误。

C. builtin

```
builtin [shell 内部命令 [参数表]]
```

运行一个shell 内部命令，把参数表传给它，并返回它的退出状态。这可以用来定义一个与 shell 内部命令同名的函数，并在函数内部保留这个内部命令的功能。如果 shell 内部命令不是有效的 shell 内部命令，则返回状态是非零。

D. caller

```
caller [表达式]
```

返回当前活动的子程序 (即 shell 函数，或通过内部命令 . 或 source 执行的 shell 脚本) 调用。

如果没有表达式，caller 显示当前子程序调用的行号和源文件名。如果用非负整数作为表达式，caller 就会显示行号、子程序名称、以及与当前调用堆栈位置相对应的源文件。这些额外信息可以用来打印堆栈跟踪信息。当前的帧是第 *0* 帧。

返回值是 *0*，除非 shell 并没有在执行子程序调用，或者表达式不对应调用堆栈中的有效位置。

E. command

```
command [-pVv] 命令 [参数表 ···]
```

把参数传给命令并执行这个命令，而忽略与之同名的 shell 函数。只执行 shell 内部命令或者通过搜索 PATH 而找到的命令。如果有一个 shell 函数叫 ls，则在该函数中执行“command ls”会运行外部命令 ls 而不是递归调用该函数。“-p”选项助记词：Path，默认路径的含义是使用默认的 PATH 以确保能够找到所有的标准命令。这时，如果没有找到命令或者发生错误，返回状态就是 *127*，否则就返回命令的返回状态。

如果指定“-V”选项或者“-v”，则打印关于命令的描述。“-v”选项助记词：Verbose，详细显示启动该命令的命令或文件名，它是一个单词；而“-V”选项助记词：VVerbose，更详细的输入更详细。这时，如果找到命令则返回零，否则返回非零。

F. declare

```
declare [-aAfFilrtux] [-p] [名称[=值] ···]
```

声明变量并设置其属性。如果没有给定名称，则显示变量的值。

“-p”选项助记词：Print，打印会显示每个名称的属性和值。如果使用了“-p”和参数名称，则忽略其余参数。如果给定“-p”却没有给定参数名称，则显示所有通过其它选项选定了属性的变量的属性和值。如果给定“-p”而没有给定其它选项，declare 将会显示所有 shell 变量的属性和值。“-f”选项助记词：Function，函数限制输出只显示 shell 函数。“-F”选项助记词：no-Function-def，无定义禁止显示函数的定义，只显示函数的名称和属性。如果用 shopt（参见§4.3.2[内部命令 shopt], *p45*）打开了 shell 的 extdebug 选项，则还会显示函数定义所在的源文件名和行号。“-F”选项隐含了“-f”。下列选项限制含有指定属性的变量输出或者给变量设置属性：

`-a` 每个名称都是下标数组变量（参见§6.7[数组], *p62*）。助记词：Array，数组

`-A` 每个名称都是键值数组变量（参见§6.7[数组], *p62*）。助记词：AArray，键值数组

`-f` 只使用函数名称。助记词：Function，函数

`-i` 把变量当成整数；对它赋值时会进行算术求值[1]（参见§6.5[Shell 的算术运算], *p61*）。助记词：Integer，整数

`-l` 对变量名称赋值，把大写字母转化为小写。清除 upper-case 属性。助记词：Lowercase，小写

`-r` 把名称设为只读。以后不可以用赋值语句对这些名称赋值或者把它们重置。助记词：Readonly，只读

`-t` 给每个名称都设置 trace 属性。设置了 trace 属性函数继承调用它的 shell 的 DEBUG 和 RETURN 陷阱。trace 属性对变量来说没有特殊意义。助记词：Trace，跟踪

`-u` 对变量名称赋值，把小写字母转化为大写。清除 lower-case 属性。助记词：Uppercase，大写

`-x` 通过环境把每个名称导出给后续命令。助记词：eXport，导出

（在选项前）使用“+”而不是“-”会关闭属性；例外的是，“+a”不会清除数组变量，“+r”也不会去掉只读属性。在函数中使用时，declare会让每个q名称都是局部可见的，就和 local 命令一样。如果变量名称后面有 =值，这个值就会赋给名称。

返回状态是零，除了遇到无效的选项、或者试图用“-f foo=bar”的形式定义函数、或者试图给只读变量赋值、或者没有使用复合赋值的语法（参见§6.7[数组], *p62*）给数组变量赋值、或者某个名称不是有效的 shell 变量名、或者试图去掉只读变量的只读属性，或者试图去掉数组变量的数组状态、或者试图用“-f”去显示一个不存在的函数。

[1] Bash 的算术运算只能处理整数。

G. echo

```
echo [-neE] [参数 ···]
```

输出每个参数，中间用空格分隔，结束时输出换行符。返回状态永远是 *0*。如果给定了“-n”助记词：noNewline，不换行，则结束时不会输出换行符。如果指定了“-e”选项助记词：Evaluate，求值，则会解释下列用斜杠转义的字符；而“-E”选项助记词：noEvaluate，不求值会禁止解释它们，即使系统默认会去解释。可以使用 shell 的 xpg_echo 选项来动态决定 echo 是否解释这些转义字符。echo 不会所“--”当成选项的结束。

echo 会解释下列转义序列 (参见§3.1.2[ANSI 标准 C 引用], *p5*)：

- \a 警告 (响铃)
- \b 退格删除
- \c 禁止继续输出
- \e 转义字符
- \f 走纸换页
- \n 新行
- \r 换行
- \t (水平) 制表符
- \v 垂直制表符
- \\ 反斜杠
- *nn* 由八进制数 *nnn* (一个到三个数字) 代表的一个八位字符。
- \x*HH* 由十六进制数 *HH* (一个到两个数字) 代表的一个八位字符。

H. enable

```
enable [-a] [-dnps] [-f 文件名 ] [名称 ···]
```

启用或禁止 shell 内部命令。禁止了内部命令，就可以执行与之同名的外部命令而不必指定它的完整路径，即使正常情况下 shell 会行搜索内部命令再搜索外部命令。如果指定了“-n”助记词：Not-enabled，禁止，则禁止名称的使用；否则就启用它们。例如，要使用在 $PATH 中找到的 test 命令而不是 shell 内部的那个，就要输入“enable -n test”。

如果给定了“-p”选项助记词：Print，打印或者没有参数名称，则列出 shell 内部命令。如果没有其它参数，则列出所有已经启用的 shell 内部命令。“-a”选项会在列出每个内部命令的时候显示它是否已经启用。

在支持动态加载的系统上，“-f”选项助记词：Filename，文件名会从共享库文件名中加载新的内部命令名称。而“-d”选项助记词：Delete，删除会删除通过“-f”加载的选项。

如果没有选项，则列出 (一些) shell 内部命令。“-s”选项助记词：Special，特殊的限制 enable 只列出 POSIX 的特殊内部命令。如果“-s”和“-f”一起使用，则新的内部命令就成为特殊命令 (参见§4.4[特殊内部命令], *p47*)。

返回状态是零，除非名称不是 shell 内部命令，或者在从共享库中加载新的内部命令时发生错误。

I. help

```
help [-dms] [模式]
```

显示内部命令的帮助信息。如果给定了模式，则 help 会显示所有与模式匹配的命令帮助，否则就列出内部命令。

如果给定了选项，就会有下列含义：

`-d` 对与模式匹配的每个命令，显示一条简短的描述。助记词：Description，描述

`-m` 对与模式匹配的每个命令，显示像联机帮助 (manpage) 那样的帮助信息。助记词：Manpage，联机帮助

`-s` 对与模式匹配的每个命令，简短的显示使用概要。助记词：Synopsis，使用概要

返回状态是零，除非没有命令与模式匹配。

J. let

```
let 表达式 [表达式]
```

内部命令 let 可以对 shell 变量进行算术运算。每个表达式都根据 §6.5[Shell 的算术运算], *p61* 不求值。如果最后一个表达式的值为 *0*，则返回 *1*；否则返回 *0*。

K. local

```
local [选项] 名称[=值] ···
```

对于每个参数名称都创建一个局部变量并给它赋值。选项可以是任何 declare 可以接受的选项。local 只能在函数里面使用，它使得变量名称只在函数或其子进程中可见。返回状态是零，除非 local 在函数外部使用、或者指定了无效的名称、或者名称是只读变量。

L. logout

```
logout [n]
```

退出当前的登录 shell，向 shell 的父进程返回状 n。

M. mapfile

```
mapfile [-n 行数] [-O 原下标] [-s 忽略行数] [-t] [-u 文件描述符] [-C 回调程序]
[-c 数量] [数组]
```

从标准输入读取文本行并存入数组变量数组中；如果给定了“-u”选项，则从文件描述符中读取。默认的数组是 MAPFILE。如果给定了选项，就会有下列含义：

`-n` 最多复制的行数。如果行数是 *0*，则复制所有行。助记词：Number，行数

`-O` 从原下标开始给数组赋值。默认的原下标是 *0*。助记词：Origin，原下标

`-s` 从头面不要读取的忽略行数。助记词：diSgard，忽略行数

`-t` 从每次读取的行中删除行结尾。助记词：Tail，删除结尾

`-u` 从文件描述符而不是标准输入中读取文本行。助记词：soUrce，来源

`-C` 每当读取的行达到数量时就执行回调函数一次。“-c”选项指定数量。助记词：Callback，回调函数

`-c` 指定每次调用回调函数之间所要读取文本行的数量。助记词：Count，行数

如果指定“-C”而没有“-c”，则默认的数量是 *5000*。每次调用回调函数时，都把数组中下一个元素的下标作为额外参数传给它。回调函数的求值是在的文本行读取以后，数组元素赋值之前。

如果没有明确指定原下标，`mapfile` 会给数组赋值之前把它清空。

`mapfile` 会返回真，除非指定了无效的选项或参数，或者数组是无效的或没有赋值。

N. printf

```
printf [-v 变量] 格式 [参数表]
```

在标准输出中打印格式化后的参数表；格式化是由格式来控制的，它是包含三种数据的字符串：普通字符 (它们会直接复制到标准输出中)、转义序列 (转化以后再复制到标准输出中)、以及格式指示 (每个字符都会按次序打印下一个参数)。除了标准的 `printf(1)` 中的格式，还能识别的格式有：“%b”让 `printf` 解释对应参数中的转义序列 (除了“\c”结束输出、保留“\'”、“\"”和“\?”中的斜杠、以及以“\0”开头的八进制转义序列中可以包含最多四个数字)。“%q”把对应参数用一种能够重新作为 shell 输入的格式输出。

“-v”选项助记词：Variabe，赋给变量把输出赋给变量，而不是打印到标准输出中。

格式会被重复使用以处理所有参数。如果格式要求比实际更多的参数，多余的格式就会恰当选用零值或空字符串作为参数，就好像它们实际指定了一样。如果操作成功则返回值是零，失败时返回非零。

O. read

```
read [-ers] [-a 数组名称] [-d 分隔符] [-i 文本] [-n 字符数] [-p 提示符] [-t 超时时间] [-u 文件描述符] [名称 ···]
```

从标准输入或者“-u”选项指定的文件描述符参数中读取一行文本，并把第一个单词赋值给第一个名称，把第二个单词赋值给第二个名称，依此类推；剩余的单词连同分隔符一起赋值给最后一个名称。如果从输入流中读取的单词数比名称少，就把空值赋值给剩余的名称。`IFS` 变量中的字符用来把文本行分隔成单词。“\”字符可以用来去掉下一个字符的特殊含义，或者用来续行。如果没有指定名称，就把读取到的变量赋值给 `REPLY` 变量。返回状态是零，除非遇到文件结束标志、或者超时 (这时返回值大于 *128*)、或者“-u”选项指定了无效的文件描述符。

如果给定了选项，就会有下列含义：

`-a 数组名称` 把文本赋值给数组变量数组名称中从 *0* 开始的连续下标。赋值之前把数组名称中的所有元素都删除。忽略其它参数名称。助记词：数组

`-d 分隔符` 用分隔符的第一个字符来结束输入行，而不是用换行符。助记词：Delimiter，分隔符

`-e` 用 Readline (参见§8[编辑命令行], *p72*) 来读取文本。Readline 会使用当前的编辑设置；如果之前没有启用行编辑功能，则使用默认设置。助记词：Edit，编辑

`-i 文本` 如果使用 Readline 来读取文本，则在开始编辑之前先把它放在编辑缓存中。助记词：Initial，初始文本

-n 字符数 读取达到字符数时就返回，而不是读入一整行时才返回。助记词：Number，字符数

-p 提示符 在试图读取输入之前显示提示符，不换行。只有从终端读取输入时才会显示提示符。助记词：Prompt，提示符

-r 如果指定该选项，则反斜杠就不是转义字符，而是文本的一部分。特别的，一对反斜杠和换行符不是续行符。助记词：Raw，原始字符串

-s 安静模式。如果从终端读取输入，则不回显字符。助记词：Silent，安静

-t 超时时间 如果在超时时间指定的秒数内还没有读入完整的一行，则读取超时并返回失败。超时时间可以是个带有小数的十进制数。这个选项只有在 read 从终端，管道，或者其它特殊文件中读取输入时才有效；从普通文件中读取时没有作用。如果超时时间是 *0*，则当指定的文件描述符可用时返回成功，不可用则返回失败。如果超时则返回状态大于 *128*。助记词：Timeout，超时

-u 文件描述符 从文件描述符中读取输入。助记词：soUrce，文本源

P. readarray

```
readarray [-n 行数] [-O 原下标] [-s 忽略行数] [-t] [-u 文件描述符] [-C 回调程序]
[-c 数量] [数组]
```

从标准输入读取文本行并存入数组变量数组中；如果给定了“-u”选项，则从文件描述符中读取。

它由 mapfile 是同义词。

Q. source

```
source 文件名
```

. 的同义词 (参见§4.1[波恩 Shell 的内部命令 . (点号)], *p28*)。

R. type

```
type [-afptP] [名称 ···]
```

对于每个名称，如果把它用作命令名，指示如何解释。

如果使用了“-t”选项助记词：Type，类型，并且名称分别是别名、shell 函数、shell 内部命令、磁盘文件、或者 shell 保留字，则分别打印 alias、function、builtin、file、keyword。如果没有找到名称，则不输出任何信息，这时 type 返回错误。如果使用了“-p”选项助记词：Print，打印，则 type 要么返回将要执行的磁盘文件名称，要么什么也不返回 (如果这时“-t”不是返回 file)。“-P”选项助记词：Path，路径强制查找每个名称的路径，即使“-t”不会返回 file。如果命令在散列表中，则“-p”和“-P”打印散列表中的值，而不一定是 $PATH 中第一个找到的文件。如果使用了“-a”选项助记词：All，所有，则 type 返回可执行文件的所有路径。当且仅当没有同时使用“-p”选项时，这些路径中才会包括别名和函数。如果使用了“-f”选项助记词：Function，函数，则 type 不会试图去搜索 shell 函数，以及内部命令。

如果能找到所有的名称就返回零，否则返回非零。

S. typeset

```
typeset [-afFrxi] [-p] [名称[=值] ···]
```

typeset 命令是为了与科恩 shell 相兼容而设的。但是它已经被内部命令 declare 取代而不推荐使用。

T. ulimit

```
ulimit [-abcdefilmnpqrstuvxHST] [限制数]
```

如果系统支持，ulimit 能够控制由 shell 启动并提供给进程的资源。如果给定选项，就按下面解释：

- -S 更改并报告与资源相关联的软限制。助记词：Soft-limit，软限制
- -H 更改并报告与资源相关联的硬限制。助记词：Hard-limit，硬限制
- -a 报告当前的所有限制。助记词：All，所有
- -b 套接字缓存的最大长度。助记词：Buffer，缓存
- -c 可以创建的 core 文件的最大长度。助记词：Core，内在映像
- -d 进程数据段的最大长度。助记词：Data-segment，数据段
- -e 最大的调度优先级 (“nice”)。助记词：nicE，优先级调度
- -f Shell 及其子进程写文件时的最大长度。助记词：File，文件
- -i 最多可以延迟的信号。助记词：sIgnal，信号
- -l 可以锁定在内存中的最大长度。助记词：Lock，锁定
- -m 常驻内存集合的最大大小 (许多系统并没有实现这个限制)。助记词：Memory，内存
- -n 最多可以打开的文件描述符数目 (许多系统都不允许设置这个值)。助记词：Number，文件数目
- -p 管道的缓存大小。助记词：Pipe，管道
- -q POSIX 消息队列的最大字节数。助记词：Queue，队列
- -r 实时调度的最大优先级。助记词：Real-time，实时
- -s 堆栈的最大长度。助记词：Stack，堆栈
- -t CPU 时间的最大秒数。助记词：Time，CPU 时间
- -u 一个用户所能拥有的最多进程数。助记词：User，用户里程
- -v 进程所拥有的最大虚拟内存。助记词：Virtual-memory，虚拟内存
- -x 文件锁的最多个数。助记词：X，锁定
- -T 线程的最多个数。助记词：Thread，线程

如果给定了限制数，它就是指定资源的新值。hard，soft 和 unlimited 等特殊的值分别代表当前的硬限制，软限制和无限制。硬限制一旦设定了以后除了 root 用户就不能修改；而软限制则可以增加到硬限制的值。如果没有给定限制数，则打印指定资源的软限制的值，除非指定了“-H”选项。设置新的限制时，如果既没有“-H”也没有“-S”，则默认为“-f”。限制值一般都是按 1024 字节增加，但是“-t”是以秒增加的，“-p”是以 *512* 字节的块增加，而“-n”和“-u”不是按比例增加的。

返回值是零，除非指定了无效的选项或参数，或者设置新限制时发生了错误。

U. unalias

```
unalias [-a] [名称 ···]
```

从别名列表中删除每个名称。如果给定了“-a”助记词：All，所有的，则删除所有别名。别名在§6.6[别名], *p62*中介绍。

§4.3 改变 Shell 的行为

§4.3.1 内部命令 set

这个内部命令非常复杂，必须在单独的章节中介绍。它可以用来改变 shell 选项或者设置位置参数，也可以显示 shell 变量的名称和值。

```
set [--abefhkmnptuvxBCEHPT] [-o 选项] [参数 ···]
set [+abefhkmnptuvxBCEHPT] [+o 选项] [参数 ···]
```

如果没有指定选项或参数，set 就会显示所有 shell 变量与函数的名称和值，显示时按照当前的语言区域排序，其格式可以用来重新设置这些已经设置的变量。只读变量不可以重设。在 POSIX 模式下，只列出 shell 变量。

如果指定了选项，它们就会打开或关闭 shell 的属性。它们具有下列含义：

-a 对已经更改或新创建的变量和函数，把它们标志为可以导出到后续命令的环境中。助记词：Alter，改变的

-b 后台作业结束时立即报告，而不是在下次打印第一提示符时报告。助记词：Background，后台

-e 助记词：Exit，退出如果管道 (参见§3.2.2[管道], *p7*) 返回非零的状态则立即退出；管道可以包含简单命令 (参见§3.2.1[简单命令], *p6*)，在圆括号中间的子 shell 命令 (参见§3.2.4.3[命令组合], *p11*)，大括号之间的命令队列 (参见§3.2.3[命令队列], *p7*) 中的命令。如果失败的命令是紧接着 while 或 until 关键词后面的命令队列的一部分、或者属于 if 语句中测试部分、或者 && 或 || 队列中除了最后一个 && 或 || 后面的任何命令、或者管道中除了最后一个以外的任何命令、或者命令的返回状态经由 ! 反转，则不会退出。如果设置了 ERR 陷阱，则会在 shell 退出之间执行。

这个选项分别作用于 shell 环境和每个子 shell 环境 (参见§3.7.3[命令执行的环境], *p25*)，并会导致子 shell 在尚未执行其中所有命令之前退出。

-f 禁止生成文件名 (文件通配)。助记词：Filename，文件名

-h 搜索执行命令时把它们记录在散列表中 hash 中。这个选项默认就打开了。助记词：Hash，散列表

-k 把所有以赋值形式出现的变量都放入命令的环境中，而不仅仅是命令名称前面的那些。助记词：Keep，保留

-m 启用作业控制 (参见§7[作业控制], *p69*)。助记词：Monitor，监控作业

-n 读取命令但不执行。这可以用来检查脚本的语法。在交互式的 shell 中，这个选项会被忽略掉。助记词：No-execution，不执行

-o 选项名称 设置与选项名称对应的选项：助记词：Option，选项

allexport 与 -a 相同。

`braceexpand` 与 `-B` 相同。

`emacs` 使用 emacs 风格的行编辑界面 (参见§8[编辑命令行], *p72*)。这个选项会还会影响 `read -e` 所使用的编辑界面。

`errexit` 与 `-e` 相同。

`errtrace` 与 `-E` 相同。

`functrace` 与 `-T` 相同。

`hashall` 与 `-h` 相同。

`histexpand` 与 `-H` 相同。

`history` 启用命令历史 (参见§9.1[Bash 的历史功能], *p90*)。在交互式的 shell 中，这个选项默认是打开的。

`ignoreeof` 交互式的 shell 在读取到 `EOF` 时不退出。

`keyword` 与 `-k` 相同。

`monitor` 与 `-m` 相同。

`noclobber` 与 `-C` 相同。

`noexec` 与 `-n` 相同。

`noglob` 与 `-f` 相同。

`nolog` 目前忽略这个选项 (尚不支持)。

`notify` 与 `-b` 相同。

`nounset` 与 `-u` 相同。

`onecmd` 与 `-t` 相同。

`physical` 与 `-P` 相同。

`pipefail` 如果打开，则管道的退出状态与其中最后一个 (最右边的) 退出状态为非零的命令相同；如果管道中的所有命令都成功执行，则返回零。这个选项默认是关闭的。

`posix` 更改 Bash 的行为以遵循 POSIX 标准 (参见§6.11[Bash 的 POSIX 模式], *p67*)；默认不是 POSIX 模式的。这个选项的目的是让 Bash 成为严格遵循 POSIX 的超集。

`privileged` 与 `-p` 相同。

`verbose` 与 `-v` 相同。

`vi` 使用 vi 风格的行编辑界面。这个选项会还会影响 `read -e` 所使用的编辑界面。

`xtrace` 与 `-x` 相同。

`-p` 助记词：Privileged，特权的打开特权模式。在这种模式下，不处理 `$BASH_ENV` 和 `$ENV` 文件，不从环境中继承 shell 函数，如果环境中有 `SHELLOPTS`，`CDPATH` 和 `GLOBIGNORE` 变量也会忽略。如果启动 shell 时的有效用户 (组) 和实际用户 (组) 不一样，并且没有指定“`-p`”选项，则除了这些，还把有效用户设为实际用户。如果指定了“`-p`”选项，则不改变有效用户。关闭这个选项会把有效用户和有效用户组设为实际用户和实际用户组。

`-t` 读取并执行一条命令后就退出。助记词：exiT，退出

`-u` 在进行参数扩展时，如果变量没有设置就报错。把错误信息写入到标准错误输出中，并退出非交互运行的 shell。助记词：Unset，未定义的

`-v` 在 shell 读取输入行时把它打印出来。助记词：Verbose，详细

`-x` 对于简单命令，`for` 命令，`case` 命令，`select` 命令，命令与其参数或者关联的单词列表里面的算术运算，则在扩展以后、执行之前，打印跟踪信息。在打印命令及其扩展过的参数之前，先打印扩展后的 `PS4` 变量。助记词：X，跟踪

`-B` 进行大括号扩展 (参见§3.5.1[大括号扩展], *p15*)。这个选项默认是打开的。助记词：Brace，大括号

`-C` 防止使用“`>`”，“`>&`”和“`<>`”的重定向覆盖已经存在的文件。助记词：Collision，文件冲突

`-E` 如果打开，则 shell 函数、命令替换、以及子 shell 环境中执行的命令会继承 `ERR` 陷阱。通常，这些情况下不会继承 `ERR` 陷阱。助记词：ERR，错误陷阱

`-H` 打开“`!`”风格的历史替换 (参见§9[历史的交互使用], *p90*)。在交互式的 shell 中，这个选项默认是打开的。助记词：History，历史

`-P` 助记词：Physical，物理路径如果打开，执行诸如 `cd` 等改变当前目录的命令时不跟踪符号链接，而使用物理路径。默认情况下，Bash 执行能够改变当前目录的命令时会使用逻辑路径。例如，如果“`/usr/sys`”是指向“`/usr/local/sys`”的符号链接，则

```
$ cd /usr/sys; echo $PWD
/usr/sys
$ cd ..; pwd
/usr
```

如果打开 `-P`，则

```
$ cd /usr/sys; echo $PWD
/usr/local/sys
$ cd ..; pwd
/usr/local
```

`-T` 如果打开，则 shell 函数、命令替换、以及子 shell 环境中执行的命令，会继承 `DEBUG` 和 `RETURN` 陷阱。通常，这些情况下不会继承 `DEBUG` 和 `RETURN` 陷阱。

`--` 如果后面没有其它参数，则重置位置参数；否则，把它们赋值给位置参数，即使它们开头是“`-`”。

`-` 表示选项结束，后续参数都会赋值给位置参数。这会关闭“`-x`”和“`-v`”选项。如果没有后续参数，则不改变位置参数。

如果在选项中使用“`+`”而不是“`-`”，这些选项就会关闭。这些选项还可以在 shell 启动时使用。当前的选项保存在变量 `$-`中。

剩余的 *N* 个参数是位置参数，它们会按顺序分别赋给 `$1`, `$2`, ··· `$N`。特殊变量 `$#` 设为 N。

返回状态总是零，除非指定了无效的选项。

§4.3.2 内部命令 shopt

本内部命令可以改变 shell 额外选项的行为。

```
shopt [-pqsu] [-o] [选项名称 ···]
```

打开或关闭 shell 选项以控制 shell 的行为。如果没有选项，或者有“-p”选项助记词：Pretty-Print，格式输出，则显示所有可以设置的选项，并指示每个选项是否打开。“-p”选项使输出的形式可以重新用于输入。其它选项有下列含义：

`-s` 设置 (打开) 选项名称。助记词：Set，设置

`-u` 重置 (关闭) 选项名称。助记词：Unset，重置

`-q` 禁止正常输出。它的返回状态可以提示选项名称是否打开。如果“-q”带有多个选项名称，则当所有的选项名称都打开时返回零，否则返回非零。助记词：Quiet，安静模式

`-o` 限制选项名称为内部命令 set (参见§4.3.1[内部命令 set], *p42*) 所定义的那些。助记词：Option，SET的选项

如果指定了“-s”或者“-u”选项而没有选项名称，则只 (分别) 显示已经打开或关闭的选项。除非特别说明，shopt 选项默认都是关闭的。列出选项时，如果所有选项都打开了，则返回零，否则返回非零。设置或重置选项时返回零，除非遇到无效的 shell 选项。shopt 有下列选项：

`autocd` 如果打开，则使用目录名称作为命令就和把这个目录作为 cd 命令的参数一样。这个选项只在交互式的 shell 中使用。

`cdable_vars` 如果打开，并且内部命令 cd 的参数不是个目录，就把这个参数当成值为目录的变量，并进入该目录。

`cdspell` 如果打开，则自动改正 cd 命令中对目录名轻微的拼写错误。这些错误包括颠倒的字符，缺少一个字符，或者多一个字符。如果改正了错误，则打印改正后的路径，并继续处理命令。这个选项只在交互式的 shell 中使用。

`checkhash` 如果打开，Bash 会在执行命令前检查散列表中的命令是否存在。如果已经不存在，则进行正常搜索。

`checkjobs` 如果打开，Bash 会在退出交互式的 shell 前列出所有正在运行或已经停止的作业状态。如果有作业正在运行，则延迟退出，直到使用干涉命令 (参见§7[作业控制], *p69*) 再次尝试退出。如果有已经停止的作业，shell 总会延迟退出。

`checkwinsize` 如果打开，Bash 会在执行每个命令以后都检查 (终端) 窗口大小；如果必要，就更新 LINES 和 COLUMNS 的值。

`cmdhist` 如果打开，Bash 会试图把多行的命令的所有行都保存在同样的历史文件中。这使得再次编辑多行命令变得容易。

`compat31` 如果打开，Bash 会在处理条件命令的 =~ 时，以第 *3.1* 版的方式处理引用。

`dirspell` 如果打开，Bash 会在单词补全时遇到原本不存在的目录后试图改正目录名的拼写。

`dotglob` 如果打开，Bash 会在文件名扩展的结果中包含以“.”开头的文件名。

`execfail` 如果打开，则非交互式运行的 shell 在用内部命令 exec 不能执行指定的文件时不会退出。exec 失败时，交互式的 shell 都不会退出。

`expand_aliases` 如果打开，则按照§6.6[别名], *p62* 中所说的对别名进行扩展。这个选项在交互式的 shell 中默认就是打开的。

`extdebug` 如果打开，则启动用于调试的行为：

1. 内部命令 `declare` 的“`-F`”选项 (参见§4.2[Bash 的内部命令 `declare`], *p36*) 会显示与每个参数所指定的函数名对应的源文件名和行号。
2. 如果 `DEBUG` 陷阱运行的命令返回的值为非零，则忽略下一条命令的执行。
3. 如果 `DEBUG` 陷阱运行的命令返回的值为 *2*，并且 shell 正在执行子程序 (即 shell 函数，或者用内部命令 `.` 或 `source` 执行的 shell 脚本)，则模拟 `return` 的调用。
4. 更新 `BASH_ARGC` 和 `BASH_ARGV` 变量 (参见§5.2[Bash 的变量], *p49*)。
5. 启用函数跟踪：通过“`(命令)`”而执行的命令替换，shell 函数和子 shell 都会继承 `DEBUG` 和 `RETURN` 陷阱。
6. 启用错误跟踪：通过“`(命令)`”而执行的命令替换，shell 函数和子 shell 都会继承 `ERROR` 陷阱。

`extglob` 如果打开，则使用扩展的模式匹配 (参见§3.5.8.1[模式匹配], *p20*)。

`extquote` 如果打开，则对于双引号之间的“`${参数}`”，还会处理参数里面的“`$"字符串"`”和“`$'字符串'`”引用。这个选项默认就是打开的。

`failglob` 如果打开，则在文件名扩展中如果模式与文件名不匹配就会导致扩展错误。

`force_fignore` 如果打开，则在单词补全时会忽略 shell 变量 `FIGNORE` 指定的后缀，即使被忽略的单词是唯一可以补全的 (参见§5.2[Bash 的变量], *p51*)。这个选项默认就是打开的。

`globstar` 如果打开，则在文件名扩展中“`*`”字符会匹配文件以及零个或多个文件夹和子文件夹。如果这个模式后面有个“`/`”，则只匹配文件夹和子文件夹。

`gnu_errfmt` 如果打开，则用标准的 GNU 错误信息格式输出 shell 错误信息。

`histappend` 如果打开，则当 shell 退出时，把历史记录添加到 `HISTFILE` 变量指定的文件中，而不是覆盖这个文件。

`histreedit` 如果打开，并且使用了 Readline，则当历史替换失败时给用户提供重新编辑的机会。

`histverify` 如果打开，并且使用了 Readline，则历史替换的结果不会立即传给 shell 来解释，而是把结果的文本加载到 Readline 的编辑缓存中以便进一步修改。

`hostcomplete` 如果打开，并且使用了 Readline，则当补全一个含有“`@`”的单词时，Bash 会试图进行主机名补全 (参见§8.4.6[补全命令], *p84*)。这个选项默认就是打开的。

`huponexit` 如果打开，则在交互式的登录 shell 退出时，Bash 会向每个作业发送 `SIGHUP` (参见§3.7.6[信号], *p26*)。

`interactive_comments` 允许忽略以“`#`”开头的单词以及其所在行中所有剩余的单词。这个选项默认就是打开的。

`lithist` 如果打开，并且打开了 `cmdhist` 选项，则保存在历史中的多行命令就带有内部换行符，而不是在必要时使用分号分隔。

`login_shell` 如果作为登录 shell 启动就打开它 (参见§6.1[Bash 的启动], *p55*)。这个选项不可以更改。

`mailwarn` 如果打开，并且 Bash 用来检查新邮件的文件在它上次检查过后已经被访问，则打印“邮箱文件已经被读取过”。

`no_empty_cmd_completion` 如果打开，并且使用了 Readline，则 Bash 不会为补全空行而试图搜索 `PATH`。

`nocaseglob` 如果打开，则 Bash 在进行文件名扩展时对文件名的大小写不敏感。

`nocasematch` 如果打开，则 Bash 在执行 `case` 命令或 `[[` 条件命令并进行模式匹配时对模式的大小写不敏感。

`nullglob` 如果打开，则 Bash 允许不匹配任何文件的模式扩展为空字符串，而不是扩展成自身。

`progcomp` 如果打开，则启用可编程的补全功能 (参见§8.6[可编程的补全], *p86*)。这个选项默认就是打开的。

`promptvars` 如果打开，则对提示符按下述方法扩展以后 (参见§6.9[提示符的控制], *p64*) 还要进行参数扩展，命令替换，算术扩展和引用去除。这个选项默认就是打开的。

`restricted_shell` 如果 shell 以受限制的模式 (参见§6.10[受限制的 shell], *p66*) 启动就打开它。这个选项不可以更改。在执行初始化文件时也不会重置它，这使得初始化文件可以检测一个 shell 是不是受限制的。

`shift_verbose` 如果打开，则当移动的数目超过位置参数的数目时，内部命令 `shift` 会打印一条错误信息。

`sourcepath` 如果打开，则内部命令 `source` 会用 `PATH` 会搜索参数中指定文件所在的目录。这个选项默认就是打开的。

`xpg_echo` 如果打开，则内部命令 `echo` 默认会扩展转义字符序列。

列出选项时，如果所有选项都打开了，则返回零，否则返回非零。设置或重置选项时返回零，除非遇到无效的 shell 选项。

§4.4 特殊内部命令

由于历史的原因，POSIX 标准把几个内部命令归入到特殊的类别。当 Bash 在 POSIX 模式下运行时，这些特殊的内部命令和其它内部命令有三个不同的方面：

1. 在搜索命令时，特殊命令的搜索先于 shell 函数。
2. 如果特殊命令返回错误状态，则非交互运行的 shell 就会退出。
3. 命令前面的赋值语句在命令结束以后仍然对 shell 环境有影响。

当 Bash 不是在 POSIX 模式下运行时，这些内部命令和其它内部命令的行为没有什么不同。Bash 的 POSIX 模式在§6.11[Bash 的 POSIX 模式], *p67*中详细介绍。

下面是 POSIX 规定的特殊内部命令：

```
break     :       .
continue  eval    exec
exit      export  readonly
return    set     shift
trap      unset
```

第五章　Shell 变量

本章介绍 shell 中所使用的变量。Bash 会自动给其中一些变量赋默认值。

§5.1　波恩 Shell 的变量

Bash 使用一些和波恩 shell 同样的变量。有时，Bash 会给它赋默认值。

`CDPATH`　冒号分隔的一组目录名，用作内部命令 `cd` 的搜索路径。

`HOME`　当前用户的主目录，也是内部命令 `cd` 的默认值。这个变量的值还用在波浪号扩展中 (参见§3.5.2[波浪号扩展], *p15*)。

`IFS`　用来分隔字段的一组字符；在扩展时，shell 用它来拆分单词。

`MAIL`　如果这个变量设为一个文件名，并且没有设置 `MAILPATH` 变量，Bash 将通知用户在指定文件中有新邮件。

`MAILPATH`　冒号分隔的一级文件名，shell 会定期在这些文件中检查新邮件。每个文件名都可以名称后面用“`?`”分隔，然后指定一条消息，当新邮件到达时将把这条消息打印出来。在消息文本中，`$_` 扩展成当前邮箱文件名[1]。

`OPTARG`　内部命令 `getopts` 处理的最后一个选项。

`OPTIND`　内部命令 `getopts` 处理的最后一个选项参数。

`PATH`　冒号分隔的一组目录，shell 用它来搜索命令。长度为零的 (空目录) 目录名表示当前目录，它可以作为两个连在一起的冒号出现，也可以作为开始或结束的冒号出现。

`PS1`　主提示符，它的默认值是“`\s-\v$␣`”[2]。关于在显示 `PS1` 之前就被扩展的转义序列完整列表，请参见§6.9[提示符的控制], *p64*。

`PS2`　第二提示符。默认值是“`>␣`”[3]。

[1] 传统的 `mbox` 邮箱中能保存多封邮件，而对于 `maildir` 则是每个文件只存放一封邮件。所以如果邮件系统是 `maildir` 类型的，例如 qmail，则应该指定 `MAILPATH` 而不是 `MAIL`。
[2] 注意最后的空格。
[3] 注意，最后面也有个空格。

§5.2 Bash 的变量

下面的变量是 Bash 设置和使用的，但是其它 shell 通常都不会对它们进行特殊处理。有些 Bash 变量在其它章节中介绍，例如作业控制所使用的变量 (参见§7.3[作业控制变量], *p71*)。

`BASH` 执行当前 Bash 所用的完整路径。

`BASHPID` 扩展为当前 Bash 进程的进程号。在有些情况下，这和 `$$` 是不同的；例如子 shell 中，这时 Bash 不会重新初始化。

`BASH_ALIASES` 一个键值数组变量，其中的元素和内部命令 `alias` 所维护的别名列表相对应 (参见§4.2[Bash 的内部命令 `alias`], *p34*)。这个数组中添加的元素将也出现在别名列表中；而删除元素将会同时删除别名列表中的别名。

`BASH_ARGC` 一个数组变量，其中的元素是当前 Bash 的执行栈中每个帧里面的参数个数。当前子程序 (即 shell 函数，或者用 `.` 或 `source` 执行的 shell 脚本) 的参数个数在栈的顶端。执行子程序时，转递给它的参数个数被压入 `BASH_ARGC` 数组。只有 shell 运行在扩展的调试模式中 (参见§87[内部命令 shopt], *p46*) 才会设置这个数组。

`BASH_ARGV` 一个数组变量，包含当前 Bash 的执行栈中的所有变量。最后一个子程序调用的最后一个参数在栈的顶端，而第一个子程序调用的第一个参数在栈的底部。在执行子程序时，向它传递的参数被压入到 `BASH_ARGV` 中。只有 shell 运行在扩展的调试模式中 (参见§87[内部命令 shopt], *p46*) 才会设置这个数组。

`BASH_CMDS` 一个键值数组变量，其中的元素和内部命令 `hash` 所维护的命令散列表相对应 (参见§4.1[波恩 Shell 的内部命令 `hash`], *p31*)。这个数组中添加的元素将也出现在散列表中；而删除元素将会同时删除散列表中的命令。

`BASH_COMMAND` 当前正在或即将执行的命令。如果这个命令是从属于陷阱，则表示触引发陷阱的命令。

`BASH_ENV` 如果在启动 Bash 来执行一个 shell 脚本时设置了这个变量，它的值就会在执行脚本前被扩展并当作初始化文件来读取 (参见§6.2[Bash 的启动脚本], *p56*)。

`BASH_EXECUTION_STRING` 启动选项“`-c`”的命令参数。

`BASH_LINENO` 一个键值数组变量，其中的元素是与每个 `FUNCNAME` 对应的源文件中的行号。`${BASH_LINENO[$i]}` 是源文件中调用 `${FUNCNAME[$i]}` 地方的行号 (如果是在其它 shell 函数中调用，则是 `${BASH_LINENO[$i-1]}`)，而对应的源文件名是 `${BASH_SOURCE[$i]}`。可以用 `LINENO` 来获得当前的行号。

`BASH_REMATCH` 一个数组变量，其中的元素由条件结构命令 `[[` 的双目运算符“`=~`” (参见§3.2.4.2[条件结构 `[[···]]`], *p10*) 来赋值。下标为 *0* 的元素是字符串中与整个正则表达式匹配的部分；下标为 *n* 的元素是字符串中与第 *n* 个括号中的子模式匹配的部分。这个变量是只读的。

`BASH_SOURCE` 一个数组变量，其中的元素是与数组变量 `FUNCNAME` 中元素对应的源文件名。

`BASH_SUBSHELL` 每次创建一个子 shell 或子 shell 环境中都把这个变量增加一。它的初始值是 *0*。

`BASH_VERSINFO` 一个只读数组变量 (参见§6.7[数组], *p62*)，其中的元素保存了当前 Bash 的版本信息。元素的值如下：

`BASH_VERSINFO[0]` 主版本号 (发布号)。

`BASH_VERSINFO[1]` 次版本号 (版本号 `BASH_VERSION`)。

`BASH_VERSINFO[2]` 补丁级别。

`BASH_VERSINFO[3]` 编译版本号。

`BASH_VERSINFO[4]` 发布状态 (例如 beta1)。

`BASH_VERSINFO[5]` `MACHTYPE` 的值。

`BASH_VERSION` 当前 Bash 的版本号。

`COLUMNS` 内部命令 `select` 在打印待选列表时用它来决定终端的宽度。它会在接收到 `SIGWINCH` 信号时自动改变。

`COMP_CWORD` 在包含当前光标位置的单词 `${COMP_WORDS}` 中的下标。这个变量只能在可编程补全的 shell 函数中使用 (参见§8.6[可编程的补全], *p86*)。

`COMP_LINE` 当前的命令行。这个变量只能在可编程补全的 shell 函数中使用 (参见§8.6[可编程的补全], *p86*)。

`COMP_POINT` 当前光标位置相对于当前命令行开头的下标。如果当前光标位置在当前命令行的尾部，则这个变量的值就与 `${#COMP_LINE}` 相同。这个变量只能在可编程补全的 shell 函数中使用 (参见§8.6[可编程的补全], *p86*)。

`COMP_TYPE` 一个整数值，与触发调用补全函数时试图进行补全的类型相对应：正常补全时为 `TAB`，连续输入制表符后的补全列表为 `?`，列出其它部分补全条目时为 `!`，没有修改单词而列出补全条目时为 `@`，补全菜单时为 `%`。这个变量只能在可编程补全的 shell 函数中使用 (参见§8.6[可编程的补全], *p86*)。

`COMP_KEY` 触发当前补全函数的键 (或键序列中的最后一个键)。

`COMP_WORDBREAKS` Readline 库进行单词补全时用作单词分隔的字符。如果没有设置这个变量，它就会失去特殊作用，即使以后再设置。

`COMP_WORDS` 一个数组变量，包含当前命令行的每个单词。当前行被上面所说的 `COMP_WORKBREAKS` 拆分成单词，和 Readline 的方式一样。这个变量只能在可编程补全的 shell 函数中使用 (参见§8.6[可编程的补全], *p86*)。

`COMPREPLY` 一个数组变量；Bash 从这个变量中读取可编程补全所调用的 shell 函数生成的补全条目 (参见§8.6[可编程的补全], *p86*)。

`DIRSTACK` 一个数组变量，包含了当前目录栈的内容。目录在栈中按照内部命令 `dirs` 所显示的顺序保存。可以通过对这个数组中元素赋值来修改栈中已有的目录，但必须用 `pushd` 或 `popd` 才能删除栈中的目录。对这个变量赋值不会改变当前目录。如果没有设置这个变量，则它就会失去其特殊作用，即使以后再设置。

`EMACS` 如果 Bash 启动时在环境中有这个变量并且其值为“`t`”，就会认为它正在 emacs 的 shell 缓存中运行，并禁止行编辑。

`EUID` 当前用户的有效用户号。这个变量是只读的。

`FCEDIT` 内部命令 `fc` 带有“`-e`”选项时默认要使用的编辑器。

FIGNORE 进行文件名扩展时要忽略的冒号分隔的后缀。如果一个文件名的后缀和这个变量某个条目匹配，它就不会出现在匹配的文件名列表中。例如，它可以取值为“.o:~”。

FUNCNAME 一个数组变量，包含当前执行栈中所有 shell 函数的名称。下标为 *0* 的元素是当前正在执行的 shell 函数；而栈底是“main”。这个变量只有在执行 shell 函数时才有。对 FUNCNAME 赋值不会生效并且会返回一个错误状态。如果没有设置这个变量，它就会推动特殊作用，即使以后再设置。

GLOBIGNORE 冒号分隔的一组模式，它定义了文件名扩展时所要忽略的文件名。在文件名扩展中，如果匹配了一个文件名，而这个文件名又和 GLOBIGNORE 中的一个模式匹配，则它将被从匹配列表中删除。

GROUPS 一个数组变量，包含当前用户所属的用户组。对 GROUPS 赋值不会生效并且会返回一个错误的状态。如果没有设置这个变量，它就会失去其特殊作用，即使以后再设置。

histchars 不超过三个字符，用来控制历史扩展、快速替换、符号化 (参见§9[历史的交互使用], *p90*)。其中的第一个是历史扩展字符，即表示历史扩展开始的字符，它通常是“!”。第二个字符如果出现在行的开头，则表示要进行“快速替换”，它通常是“^”。第三个字符是可选的，它如果是一个单词的第一个字符，就表示本行中剩余的部分是注释；它通常是“#”。历史注释字符[4]使得历史替换忽略本行中剩余的单词；而不一定表示 shell 解释器会把剩余部分当成注释。

HISTCMD 历史号，即当前命令在历史中的索引号。如果没有设置这个变量，它就失去特殊的作用，即使以后再设置。

HISTCONTROL 冒号分隔的一组值，它控制命令怎么在历史中保存。如果值中包含“ignorespace”，则不在历史中保存以空格开头的行。值为“ignoredups”表示不保存与前一历史条目匹配的行。值“ignoreboth”是“ignorespace”和“ignoredups”两者的简写。值“erasedups”表示在保存本行时首先删除与本行匹配的前一行。此外的所有值都会被忽略。如果没有设置这个变量，或它不包含有效的值，则 shell 解释器会根据 HISTIGNORE 的值在历史中保存所有读取的行。多行命令的第二和其余行不会进行如上的检测，不管这个变量如何设置都会保存到历史中。

HISTFILE 用来保存命令历史的文件，默认值是 ~/.bash_history。

HISTFILESIZE 历史文件中包含的最多行数。如果给这个变量赋值，历史文件就会在必要时删除最早的记录，以保证不超过指定的行数。当交互式的 shell 退出运行时也会删除这个文件中的最早记录并只保留不超过指定的行数。这个变量的默认值是 *500*。

HISTIGNORE 冒号分隔的一组模式，用以决定哪些命令可以保存在历史中。每个模式都定位在行的开头，必须和整行匹配 (没有隐式加上“*”)。在进行 HISTCONTROL 指定的检查后，每个模式都会和当前行匹配。除了 shell 中通常的模式匹配字符，还可以用“&”来匹配历史中的前一行。如果用使用“&”本身，可以用反斜杠转义；在匹配之前，这个反斜杠将被删除。多行命令的第二和其余行不会进行检测，不管这个变量如何设置都会保存到历史中。

HISTIGNORE 包括了 HISTCONTROL 的功能。“&”模式与 ignoredups 是等价的，而“[␣]*”模式和 ignorespace 是等价的。把这两个模式用冒号连接起来就和 ignoreboth 是等价的。

HISTSIZE 历史中保存的最多命令个数。默认值是 *500*。

HISTTIMEFORMAT 如果设置了这个变量且不为空，它的值将用作 strftime 的格式字符串，并被内部命令 history 显示历史条目时打印每条历史的时间戳。如果设置了这个变量，时间戳也会写入到历史文件中以便在不同的 shell 会话中保留这些信息。它使用历史注释字符以便把时间戳和其它历史行区别开来。

[4] 即刚刚说的第三个字符。

HOSTFILE 指定一个格式与 /etc/hosts 相同的文件，以便 shell 补全主机名时使用。在 shell 运行时可以更改主机名补全的条目。改变这个值后再进行主机名补全时，Bash 会把新文件中的内容加入到已有的列表中。如果设置了这个变量却没有指定值[5]，Bash 就试图读取 /etc/hosts 来获得主机名补全的条目。如果重置这个变量，就会清空主机名列表。

HOSTNAME 当前主机的名称。

HOSTTYPE 一个字符串，它描述了运行 Bash 的机器的类型。

IGNOREEOF 控制着 shell 读取到 EOF 字符作为整个输入时的行为。如果设置了，它的值表示连续 EOF 字符的个数；如果 shell 输入的行首有这么多 EOF，就会退出。如果设置了这人变量，而其值却不是数值 (或没有值)，则默认为 *10*。如果这个变量不存在，EOF 字符就表示 shell 输入的结束。这只有在交互式的 shell 中才有效。

INPUTRC Readline 初始化文件的名称，用来覆盖默认值 ~/.inputrc。

LANG 用来指定语言类别，如果这个类别没有特别地用 LC_ 开头的变量指定。

LC_ALL 这个变量覆盖 LANG 和所有其它 LC_ 变量指定的语言类别。

LC_COLLATE 这个变量决定文件名扩展结果的排序顺序，以及文件名扩展和文件名匹配中的范围表达式、等价字符类、语言区域序列 (参见§3.5.8[文件名扩展], *p19*)。

LC_CTYPE 这个变量决定文件名扩展和模式匹配中对字符的解释和字符类的行为 (参见§3.5.8[文件名扩展], *p19*)。

LC_MESSAGES 这个变量决定翻译 $ 后面的双引用字符串 (参见§3.1.2.1[Locale 专用的翻译], *p6*) 时所使用的语言区域。

LC_NUMERIC 这个变量决定格式化数字时所使用的语言区域。

LINENO 当前执行的脚本或 shell 函数中的行号。

LINES 内部命令 select 在打印待选条目时所用的列宽度。这个值会在接收到 SIGWINCH 信号后自动更新。

MACHTYPE 对正在运行 Bash 的系统，一个能完整描述系统类型的字符串，它的格式是 GNU 标准的“CPU-COMPANY-SYSTEM”。

MAILCHECK Shell 检查邮件的频度 (以秒为单位)，邮件存放在 MAILPATH 或 MAIL 变量指定的文件中。默认值是 *60* 秒。到达检查邮件的时间时，shell 会先检查邮件再显示主提示符。如果这个变量没有设置，或设为一个不是大于或等于零的数，则 shell 就禁用邮件检查。

OLDPWD 内部命令 cd 所设置的以前工作目录。

OPTERR 如果设为 *1*，Bash 会显示内部命令 getopts 所产生的错误。

OSTYPE 一个字符串，描述了正运行 Bash 的操作系统的类型。

PIPESTATUS 一个数组变量 (参见§6.7[数组], *p62*)，包含最近在前台执行的管道 (可能只有一个命令) 中进程的退出状态。

[5] 例如，赋与了空值。

`POSIXLY_CORRECT` 如果 Bash 启动时这个变量在环境中，shell 就会在读取初始化文件之前进入 POSIX 模式 (参见§6.11[Bash 的 POSIX 模式], *p67*)，就好像指定了“`--posix`”启动选项一样。如果在 shell 运行中设置这个变量，Bash 也会进入 POSIX 模式，就好像执行了 `set -o posix` 命令一样。

`PPID` Shell 父进程的进程号。这个变量是只读的。

`PROMPT_COMMAND` 如果设置了，它的值就被当作每次显示主提示符 (`$PS1`) 之前要执行的命令。

`PROMPT_DIRTRIM` 如果设为一个大于零的数，这个值就是扩展提示符中的转义字符 `\w` 和 `\W` (参见§6.9[提示符的控制], *p64*) 时要保留的目录节点数目。删除的字符用省略号来代替。

`PS3` 这个变量的值用作内部命令 `select` 的提示符。如果没有设置这个变量，`select` 命令就把“`#?`”作为提示符。

`PS4` 它的值用作回显命令时的提示符；可以用内部命令 `set` 的“`-x`”选项打开命令回显 (参见§86[内部命令 set], *p44*)。这个变量的第一个字符在必要时将被重复显示以表示多层间接变量 (参见§3.5.3[Shell 参数扩展], *p16*)。默认值是“`+␣`”。

`PWD` 内部命令 `cd` 所设置的当前工作目录。

`RANDOM` 每次使用这个变量都会随机生成一个 *0* 到 *32767* 之间的整数。对这个变量赋值就会给随机数发生器赋于不同的种子。

`REPLY` 内部命令 `read` 的默认变量。

`SECONDS` 这个变量扩展为 shell 从启动到现在经过的秒数。对这个变量赋值就会把这个时间计数器设为赋与的值，而这个变量的扩展值就成为赋与的值加上赋值后所经过的秒数。

`SHELL` 这个环境变量保存 shell 的完整路径。如果 shell 启动时没有设置它，Bash 就把当前用户的登录 shell 的完整路径赋给它。

`SHELLOPTS` 冒号分隔的所有启用的 shell 选项；其中的每个单词都是内部命令 `set` 的“`-o`”选项的参数 (参见§86[内部命令 set], *p42*)。这些选项是 `set -o` 命令显示为“`on`”的条目。如果启动 Bash 时环境中有这个变量，则 shell 会在读取初始化文件之前打开其中的每个选项。这个变量是只读的。

`SHLVL` 每次启动一个新的 Bash 时这个变量都增加一。它是为计算 shell 的嵌套层次而设的。

`TIMEFORMAT` 这全变量的值用作格式化字符串上，它决定带有 `time` 关键字的管道怎么显示时间信息。“`%`”引导一个转义序列，可以扩展为时间值或其它信息。转义序列的含义如下；其中的方括号表示可选的部分。

`%%` 表示“`%`”本身。

`%[p][l]R` 已经经历的时间，以秒为单位。

`%[p][l]U` 用户模式下所使用的 CPU 时间，以秒为单位。

`%[p][l]S` 系统模式下所使用的 CPU 时间，以秒为单位。

`%P` CPU 使用百分比，即 $(\%U + \%S)/\%R$。

其中可选的 p 是一个指定显示精度的数字，即小数点后的小数位数。*0* 表示不显示小数点和后面的小数。最多只能指定显示三位；比 *3* 大的数值将被改为 *3*。如果没有指定 p，则使用 *3*。可选的 l 指定使用长格式 `MM mSS.FFs`，其中包括分钟。p 的值决定是否包括小数部分。

如果这个变量没有设置，Bash 就把它当成

```
$'\nreal\t%3lR\nuser\t%3lU\nsys\t%3lS'
```

如果这个值为空，则不显示时间信息。在显示格式化后的信息时，会自动在末尾加上一个换行符。

`TMOUT` 如果设为一个大于零的值，它就是内部命令 `read` (参见§4.2[Bash 的内部命令 `read`], *p39*) 默认的超时时间。如果输入是来自终端并且在 `TMOUT` 秒内没有输入，则`select` 命令 (参见§3.2.4.2[条件结构 `select`], *p10*) 也会结束。在交互式的 shell 中，这个值表示打印提示符以后等待输入时的秒数；如果在这段时间内没有输入，Bash 就会退出。

`TMPDIR` 如果设置了，Bash 就用它的值作为目录名并根据 shell 需要在这个目录中创建临时文件。

`UID` 当前用户的实际用户号。这个变量是只读的。

第六章 Bash 的功能

本章介绍 Bash 的特色功能。

§6.1 Bash 的启动

```
bash [长选项] [-ir] [-abefhkmnptuvxdBCDHP] [-o 选项] [-O shopt 选项] [参数 ···]

bash [长 选 项] [-abefhkmnptuvxdBCDHP] [-o 选 项] [-O shopt 选 项] -c string [参数 ···]

bash [长选项] -s [-abefhkmnptuvxdBCDHP] [-o 选项] [-O shopt 选项] [参数 ···]
```

除了单字符的命令行选项 (参见§4.3.1[内部命令 set], *p42*) 外，还可以使用一些多字符选项。要想正确解析命令行，多字符选项必须出现在单字符选项的前面。

`--debugger` 在 shell 启动前准备调试器分析。打开扩展的调试模式 (参见§87[内部命令 shopt], *p46*) 和 shell 函数的跟踪 (参见§86[内部命令 set], *p43*)。

`--dump-po-strings` 在标准输出中打印“`$`”后面的双引用字符串 (它们是 GNU gettext 的 PO 格式，即可移植目标文件格式)。除了输出的格式，其它和“`-D`”选项是等价的。

`--dump-strings` 和“`-D`”选项等价。

`--help` 在标准输出中打印使用帮助后成功退出。

`--init-file 文件名`

`--rcfile 文件名` 在交互式的 shell 中执行文件名 (而不是 `~/.bashrc`) 中的命令。

`--login` 和“`-l`”选项等价。

`--noediting` 当 shell 交互式运行时，不使用 GNU Readline (参见§8[编辑命令行], *p72*) 库来读取命令行。

`--noprofile` 当 Bash 作为登录 shell 启动时，不加载系统或个人的初始化文件 `/etc/profile`、`~/.bash_profile`、`~/.bash_login`、`~/.profile`。

`--norc` 在交互式的 shell 中，不读取初始化文件 `~/.bashrc`。如果用 `sh` 来启动 shell，这个选项默认就打开了。

`--posix` 如果 Bash 的默认行为与 POSIX 不同，就遵循 POSIX 规范。这个选项是用来让 Bash 成为该规范的一个超集 (参见§6.11[Bash 的 POSIX 模式], *p67*)。

`--restricted` 打开受限制模式 (参见§6.10[受限制的 shell], *p66*)。

`--verbose` 和"`-v`"选项等价，回显读取的输入行。

`--version` 在标准输出中显示当前 Bash 的版本信息后成功退出。

启动时还可以指定一些单字符选项；这些选项是内部命令 `set` 所没有提供的。

`-c 字符串` 处理完选项以后从字符串中读取命令并执行，然后退出。剩余的参数赋值给从 `$0` 开始的位置参数。助记词：Command，命令字符串

`-i` 强制 shell 交互式的运行。交互式的 shell 在§6.3[交互式的 shell], *p58*中介绍。助记词：Interactive，交互的

`-l` 使当前 shell 表现得像登录后直接启动的那样。在交互式的 shell 中，这和用 `exec -l bash` 命令启动的登录 shell 是等价的。如果不是交互式的 shell，则执行登录 shell 的初始化文件。`exec bash -l` 或 `exec bash --login` 将把当前的 shell 替换成一个登录 shell。关于登录 shell 的特殊行为，请参见§6.2[Bash 的启动脚本], *p56*。

`-r` 把当前 shell 变为受限制的 shell (参见§6.10[受限制的 shell], *p66*)。助记词：Login，登录的

`-s` 如果给定了这个选项，或者处理选项以后没有剩余的参数，则从标准输入读取命令。这个选项可以用来在启动交互式的 shell 时指定位置参数。助记词：Startupfile，初始化文件

`-D` 在标准输出中打印所有"`$`"后面的双引用字符串。如果当前的语言区域不是 C 或 POSIX (参见§3.1.2.1[Locale 专用的翻译], *p6*)，则要对这些字符串进行翻译。这个选项隐含了"`-n`"选项，并且不执行命令。助记词：Debug，语言翻译调试

`[-+]O [shopt 选项]` shopt 选项是内部命令 `shopt` 所接受的选项 (参见§4.3.2[内部命令 shopt], *p45*)。如果指定了shopt 选项，则"`+O`"就会设置这个选项，而"`-O`"重置它。如果没有指定shopt 选项，则在标准输出中显示 `shopt` 所接受的选项名称和值。如果这时选项是"`+O`"，则以一种可以重新作为输入的格式来显示。助记词：Option，选项

`--` 单个 `--` 表示选项的结束并停止继续处理选项。它后面的任何参数都被当成文件名或参数。

"登录 shell"是指其第零个参数的第一个字符是"`-`"，或者通过"`--login`"选项启动的 shell。"交互式"的 shell 是指启动时没有非选项的参数 (除非指定了"`-s`"选项)，没有指定"`-c`"选项，并且其输入和输出都与终端相连 (通过 `isatty(3)` 查看)，或者用"`-i`"选项启动的 shell。更多细节请参见§6.3[交互式的 shell], *p58*。

如果选项处理完毕还剩余参数，这时又没有指定"`-c`"或"`-s`"选项，则第一个参数就当作是包含 shell 命令的文件名 (参见§3.8[Shell 脚本], *p26*)。如果通过这种方式启动 Bash，则 `$0` 就设为该文件名，而位置参数就设为其余的参数。Bash 会从这个文件中读取命令并执行，然后退出。Bash 的退出状态是脚本中最后一个被执行命令的退出状态。如果没有执行任何命令，则退出状态为 *0*。

§6.2 Bash 的启动脚本

本节介绍 Bash 如果执行其初始化文件。如果这些文件存在但是不可读，Bash 就会报错。文件名中的波浪号会被扩展 (参见§3.5.2[波浪号扩展], *p15*)。交互式的 shell 在§6.3[交互式的 shell], *p58*中介绍。

§6.2.1 作为交互式的登录 shell 启动，或带有“--login”选项

当 Bash 作为交互式的 shell 启动时，或作为非交互式的 shell 但带有“--login”选项，它将先读取 /etc/profile 里面的命令并执行 (如果这个文件存在)，然后依次搜索 ~/.bash_profile、~/.bash_login、~/.profile，然后读取并执行第一个存在并且可读的文件。可以在 shell 启动时指定“--noprofile”选项来禁止这种行为。

当一个登录 shell 退出时，Bash 会读取并执行 ~/.bash_logout 里面的命令 (如果这个文件存在)。

§6.2.2 作为交互式的非登录 shell 启动

当启动一个交互式的非登录 shell 时，Bash 会读取并执行 ~/.bashrc 文件里面的命令 (如果这个文件存在)。这个行为可以用“--norc”选项来禁止。而“--rcfile 文件名”选项强制 Bash 从文件名中读取命令并执行，而不是从 ~/.bashrc 中。所以，~/.bash_profile 文件通常在进行登录相关的初始化之前 (或之后) 包含下面这行：

```
if [ -f ~/.bashrc ]; then .  ~/.bashrc; fi
```

§6.2.3 非交互式的启动

如果 Bash 非交互式的启动，例如为了运行一个脚本，它就会在环境中寻找 BASH_ENV 变量 (参见§5.2[Bash 的变量], *p49*)，如果找到就把这个变量扩展后的值当作一个文件名，并且从中读取命令并执行。这就好像执行了下面的命令：

```
if [ -n "$BASH_ENV" ]; then .  "$BASH_ENV"; fi
```

只不过这时并没有使用 PATH 来搜索这个文件。

如上所说，如果非交互式的 shell 启动时指定了“--login”选项，Bash 就会试图从初始化文件中读取命令并执行。

§6.2.4 作为 sh 启动

如果把 Bash 作为 sh 来启动，它就会尽量去模仿历史上的 sh 的启动行为，同时还保证遵循 POSIX 标准。

如果作为交互式的登录 shell 启动，或者作为非交互式的 shell 启动但却指定了“--login”选项，Bash 会依次试图去读取 /etc/profile 和 ~/.profile 并执行其中的命令。可以使用“--noprofile”选项来制止这种行为。如果有交互式的 shell 中用名称 sh 来启动时，Bash 将会寻找 ENV 变量，如果找到就扩展它的值并把这个值当成文件名来读取和执行。因为作为 sh 启动的 shell 不会读取和执行任何其它的初始化文件，所以“--rcfile”选项不起作用。如果用名称 sh 启动一个非交互的 shell，它就不会读取任何初始化文件。

作为 sh 启动时，Bash 会在读取初始化文件以后进入 POSIX 模式。

§6.2.5 启动 POSIX 模式

如果通过“--posix”选项以 POSIX 模式启动 Bash，它就会按照 POSIX 规范去使用初始化文件。在这种模式下，交互式的 shell 会扩展 ENV 变量并在扩展结果所指示的文件在读取命令并执行。它不会读取其它初始化文件。

§6.2.6 由远程的 shell 守护进程启动

Bash 会试图把它的标准输出和一个网络连接相关联，就好像它由远程的 shell 守护进程 (通常是 rshd 或 sshd) 启动的一样。如果 Bash 以这种方式启动，它就会读取并执行 ~/.bashrc 里面的命令 (如果这个文件存在并且可读)。而如果以 sh 来启动就不这么做。可以使用“--norc”选项来制止这种行为，或者用“--rcfile”选项来强制读取另外一个文件，但 rshd 在启动 shell 时通常都不带这些选项，或者不允许指定它们。

§6.2.7 启动时实际用户 (组) 号和有效用户 (组) 号不同

如果 Bash 启动时实际用户 (组) 号和有效用户 (组) 号不同，并且没有指定“-p”选项，它就不会读取初始化文件，也不从环境中继承 shell 函数；如果环境中有 SHELLOPTS 变量，也会被忽略。这时，把有效用户号设为实际用户。如果启动时指定了“-p”选项，则启动行为仍然这样，但不设置有效用户号。

§6.3 交互式的 shell

§6.3.1 什么是交互式的 shell

交互式的 shell 是指它启动时没有非选项的参数 (除非指定了“-s”选项)，也没有指定“-c”选项，并且标准输出和标准错误输出都和终端关联 (可以用 isatty(3) 查看)，或者通过“-i”选项启动。通常，交互式的 shell 都读取和写入用户的终端。“-s”选项可以在启动交互式 shell 时设置位置参数。

§6.3.2 当前的 shell 是交互式的吗?

如果想在初始化脚本中检测 Bash 是否以交互方式运行，可以检测特殊变量 - 的值。如果它的值包含“i”，那么它就是交互的。另外一种方法是，检查变量 PS1 的值；这个变量在交互式的 shell 中设置，而非交互的 shell 重置了它。

❋ 代码清单 4：Readline 启动脚本的例子

```
# 第一种方法
case "$-" in
    *i*)
        echo 这个是交互式的 shell。
        ;;
    *)
        echo 这个不是交互式的 shell。
        ;;
esac

# 第二种方法
if [ -z "$PS1" ]; then
    echo 这个不是交互式的 shell。
else
    echo 这个是交互式的 shell。
fi
```

§6.3.3 交互式 shell 的行为

Shell 在交互式运行时，会改变几个方面的行为：

1. 读取并执行初始化文件 (参见§6.2[Bash 的启动脚本], *p56*)。
2. 默认启用作业控制 (参见§7[作业控制], *p69*)。已经启用作业控制时，Bash 就会忽略来自键盘的作业控制信号 SIGTTIN、SIGTTOU、SIGTSTP。
3. Bash 会在读取第一行命令之前先扩展并显示 PS1，在读取多行命令的第二和其余行之前扩展并显示 PS2。
4. Bash 会在打印主提示符 PS1 之前把变量 PROMPT_COMMAND 的值当成一个命令去执行 (参见§5.2[Bash 的变量], *p53*)。

5. 用 Readline (参见§8[编辑命令行], *p72*) 从用户的终端读取命令。
6. 在读取命令时，Bash 会检查 `set -o` 命令的 `ignoreeof` 选项 (参见§86[内部命令 set], *p43*) 的值，而不是接收到标准输入中的 `EOF` 后就立即退出。
7. 默认启用命令历史 (参见§9.1[Bash 的历史功能], *p90*) 和历史补全 (参见§9[历史的交互使用], *p90*)。在退出时，Bash 会把命令历史写入到 `$HISTFILE` 指定的文件中。
8. 默认进行别名 (参见§6.6[别名], *p62*) 扩展。
9. 如果没有定义任何陷阱，Bash 就忽略 `SIGTERM` 信号 (参见§3.7.6[信号], *p26*)。
10. 如果没有定义任何陷阱，Bash 就捕获并处理 `SIGINT` 信号 (参见§3.7.6[信号], *p26*)。`SIGINT` 信号可能会中断某些内部命令。
11. 如果打开了 `huponexit` 选项 (参见§87[内部命令 shopt], *p46*)，交互式的登录 shell 在退出时会向所有的作业发送 `SIGHUP` 信号。
12. 忽略启动选项“`-n`”。“`set -n`”也不会起作用 (参见§86[内部命令 set], *p42*)。
13. Bash 会根据变量 `MAIL`、`MAILPATH`、`MAILCHECK` 的值定期检查邮件 (参见§5.1[波恩 Shell 的变量], *p48*)。
14. 设置 `set -u` (参见§86[内部命令 set], *p44*) 后，扩展一个未定义的变量发生的错误不会导致 shell 退出。
15. Shell 不会因为在扩展 `${var:?word}` 时发生变量 `var` 未定义错误而退出 (参见§3.5.3[Shell 参数扩展], *p16*)。
16. Shell 内部命令的重定向错误不会导致退出。
17. 如果在 `POSIX` 模式下运行，特殊命令返回错误状态不会导致 shell 退出 (参见§6.11[Bash 的 POSIX 模式], *p67*)。
18. 执行 `exec` 失败不会导致 shell 退出 (参见§4.1[波恩 Shell 的内部命令 exec], *p30*)。
19. 解析器遇到语法错误不会导致 shell 退出。
20. 默认打开了内部命令 `cd` 对目录参数的简单拼写更正功能 (参见§87[内部命令 shopt], *p45*)。
21. Shell 会检查 `TMOUNT` 变量 (参见§5.2[Bash 的变量], *p54*) 的值，如果打印提示符后 `$PS1` 后在其指定的秒数内没有读取到命令就会退出。

§6.4 Bash 条件表达式

条件表达式由复合命令 `[[` 和内部命令 `test` 与 `[` 使用。这些表达式可以是单目或者双目的。单目表达式常常用来检测文件的状态。此外还有字符串运算符和数值比较运算符。如果某个基本表达式的文件参数格式为 `/dev/fd/N`，则测试的是文件描述符 *N*。如果某个基本表达式的文件参数是 `/dev/stdin`、`/dev/stdout`、`/dev/stderr` 之一，则测试的分别是文件描述符 *0*、*1*、*2*。

下面除非特别说明，下列操作文件的表达式将跟随符号链接去操作其指向的目标，而不是操作符号链接本身。

`-a 文件` 如果文件存在则为真。

`-b 文件` 如果文件存在并且是个块设备文件则为真。

-c 文件 如果文件存在并且是个字符设备文件则为真。

-d 文件 如果文件存在并且是个目录则为真。

-e 文件 如果文件存在则为真。

-f 文件 如果文件存在并且是个常规文件则为真。

-g 文件 如果文件存在并且设置了有效组号则为真。

-h 文件 如果文件存在并且是个符号链接则为真。

-k 文件 如果文件存在并且设置了“滞留位”则为真。

-p 文件 如果文件存在并且是个命名管道 (FIFO) 则为真。

-r 文件 如果文件存在并且可读则为真。

-s 文件 如果文件存在并且其大小不为零则为真。

-t 文件描述符 如果文件描述符已打开并且指向终端则为真。

-u 文件 如果文件存在并且设置了有效用户号则为真。

-w 文件 如果文件存在并且可写则为真。

-x 文件 如果文件存在并且可执行则为真。

-O 文件 如果文件存在并且被其有效用户号所拥有则为真。

-G 文件 如果文件存在并且被其有效组号所拥有则为真。

-L 文件 如果文件存在并且是个符号链接则为真。

-S 文件 如果文件存在并且是个套接字文件则为真。

-N 文件 如果文件存在并且大上次读取过后被修改过则为真。

文件一 -nt 文件二 如果文件一比文件二新 (根据修改时间)或者文件一存在而文件二不存在则为真。

文件一 -ot 文件二 如果文件一比文件二旧 (根据修改时间)或者文件二存在而文件一不存在则为真。

文件一 -ef 文件二 如果文件一和文件二指向同样的设备或文件节点则为真。

-o 选项名称 如果 shell 的选项名称已设置则为真。可以用内部命令 `set` 的“`-o`”选项 (参见§86[内部命令 set], *p42*) 列出所有选项。

-z 字符串 如果字符串的长度是零则为真。

字符串

-n 字符串 如果字符串的长度不是零则为真。

字符串一 == 字符串二 如果字符串一与字符串二相等则为真。可以把 == 换成 = 以保证与 POSIX 一致。

字符串一 != 字符串二 如果字符串一与字符串二不相等则为真。

字符串一 < 字符串二 在当前语言区域中排序时，如果字符串一排在字符串二前面则为真。

字符串一 > 字符串二 在当前语言区域中排序时，如果字符串一排在字符串二后面则为真。

数值一 运算符 数值二 运算符是“-eq”、“-ne”、“-lt”、“-le”、“-gt”、“-ge”之一。在这些算术双目运算中，如果数值一分别为等于、不等于、小于、小于或等于、大于、大于或等于数值二则为真。数值一和数值二可以是正或负整数。

§6.5 Shell 的算术运算

Shell 可以对算术表达式求值，它可以是 shell 扩展的结果，也可以由内部命令 let，或者 declare 的“-i”选项来实现。

求值时使用固定宽度的整数，并且不检查溢出，虽然它可以捕获到除以零的情况并报错。运算符的优先级、结合性、以及值都和 C 语言相同。下列运算符按优先级分组，并按优先级从高到低的顺序列出。

id++ id--	后增和后减
++id --id	先增和先减
- +	单目负号和正号
! ~	逻辑取反，按位取反
**	指数
* / %	乘，除，求余
+ -	加，减
<< >>	按位左移，按位右移
<= >= < >	比较
== !=	相等，不等
&	按位与
^	按位异或
|	按位或
&&	逻辑与
||	逻辑或
cond ? expr1 : expr2	条件运算符
= *= /= %= += -= <<= >>= &= ^= |=	赋值
expr1, expr2	逗号运算

可以使用 shell 变量作为运算数。在进行求值之前会进行参数扩展。在表达式中，还可以通过名称直接使用变量而无需参数扩展的语法形式[1]。如果是通过名称而不是参数扩展来使用变量，则对于未设置或设为空值的变量在求值时为 *0*。在引用到一个变量时，或者对一个用 declare -i 设置了其 integer 属性的变量进行赋值时，就把这个变量当成算术表达式进行求值。空值运算结果为 *0*。要在表达式中使用一个 shell 变量，不需要设置它的 interger 属性。

以 *0* 开头的常量当作八进制数解释，而以“0x”或“0X”开头表示十六进制数。此外，数值的格式是 [base#]n；其中的 base 是 *2* 到 *64* 之间的一个十进制数，它表示算术进制基数；而 n 是这个进制中的一个

[1] 即不需要变量前面的“$”。

数。如果省略了 base# 部分，则表示 *10* 进制。大于 *9* 的数字依次用小写字母、大写字母、“@”、“_”来表示。如果 base 小于或等于 *36*，则可以混合使用大小写字母来表示 *10* 到 *35* 之间的数。

求值时按运算符的优先顺序进行。括号中的子表达式先求值，可以用来改变运算顺序。

§6.6 别名

对于简单命令中的第一个单词，别名可以把它替换成一个字符串。Shell 维护一个别名列表，可以用内部命令 alias 或 unalias 进行设置或删除。

对于每条简单命令的第一个单词，如果没有引用，shell 都会去检查它是否有别名。如果有，这个单词就用别名中的文本替换。“/”、“$”、“'”、“=” 以及前面列出的任一 shell 元字符都不能在别名的名称中出现；而要替换的文本中可以含有任何有效的 shell 输入，包括 shell 元字符。检查别名时只测试替换文件中的第一个单词，但是与别名扩展后完全一样的单词不会再次被扩展。例如，这就意味着可以把 ls 作为 ls -F 的别名，但是 Bash 不会递归的去扩展要替换的文本。如果别名文本中的最后一个字符是空格或制表符，则还要对命令中别名之后单词进行别名扩展。

别名可以用 alias 命令来创建或列出，并用 unalias 命令删除。在替换的文本中没有办法像 csh 那样使用参数。如果需要参数，则应该使用 shell 函数 (参见§3.3[Shell 函数], *p12*)。在非交互运行的 shell 中不会进行别名扩展，除非打开了 shell 的 expand_aliases 选项 (参见§87[内部命令 shopt], *p46*)。

定义和使用别名的规则有点含糊。Bash 在执行任何命令之前总是至少读取一整行。别名是在读取命令时扩展的，而不是在执行时。所以，同一行中另外一个命令定义的别名直到读取下一行输入时才能生效，而本行中该别名之后的命令不受它的影响。执行函数时也有同样的问题。别名是在读取函数定义时扩展的，而不是在函数执行时，因为函数定义本身就是一条复合命令。这样的结果就是，在函数内部定义的别名直到函数执行以后才能使用[2]。所有为了安全，永远都在单独的行中定义别名，并且不要在复合命令中使用别名。

不管出于什么目的，都应该优先使用 shell 函数而不是别名。

§6.7 数组

Bash 中支持一维的下标数组变量和键值数组变量。任何变量都可以作为下标数组来使用；内部命令 declear 可以显式的声明一个数组。数组的元素个数不受限制，也不限制数组的下标或赋值时要连续。下标数组使用整数或算术表达式 (参见§6.5[Shell 的算术运算], *p61*) 来访问元素，下标从零开始，而键值数组使用任意字符串来访问元素。

如果用下面的语法形式给任意变量赋值就自动创建了一个下标数组：

```
数组名[下标]=值
```

其中下标被当成算术表达式，它的求值结果必须是一个大于或等于零的数。如果要显式的声明一个数组，则

```
declare -a 数组名
```

下面的语法格式也是允许的。下标会被忽略。

```
declare -a 数组名[下标]
```

键值数组用下面的形式来创建：

```
declare -A 数组名
```

可以用内部命令 declare 或 readonly 来设置数组变量的属性。每个属性都会作用于数组中的每个元素。可以使用下面的复合赋值语句给数组赋值

```
数组名=([下标=]值 ··· [下标=]值 )
```

[2] 指在函数中使用。而在其它地方使用则不受此限制。

对下标数组，不必要指定下标。如果给下标数组赋值时指定了下标，就赋值给指定下标的元素；否则，要赋值元素的下标就是该语句所赋值的最后一个下标加上一。下标是从零开始的。给键值数组赋值时，必须指定下标。这种语法形式对内部命令 declear 同样有用。对单个元素赋值可以最上面介绍的形式：

```
数组名[下标]=值
```

数组的任何元素都可以用

```
${数组名[下标]}
```

的形式来引用。这里必须使用大括号以免与 shell 的文件名扩展运算符想冲突。如果下标是“@”或“*”，则这个单词就扩展为数组中的所有元素。只有在这个单词位于双引号之间时这两个下标才会有区别。如果这个单词位于双引号中间，${变量名[*]} 就扩展为一个单独的单词，它是用 IFS 变量的第一个字符把把数组名所有的元素连接而成的；而 ${数组名[*]} 把 数组名中的每个元素都扩展成一个独立的单词。如果数组中没有元素，则 ${数组名[*]} 的扩展结果为空。如果双引用的扩展在单词中进行，则第一个参数扩展后就和原来单词的开头部分相连，而最后一个参数扩展后就和原来单词的结尾部分相连。这和特殊变量“@”及“*”的扩展方法是类似的。${#数组名[下标]} 会扩展成 ${数组名[下标]} 的长度。如果下标是“@”或“*”，则它就扩展为数组的长度。使用数组时如果没有指定下标，就相当于指定了 *0* 作为下标。

可以用内部命令 unset 来删除数组。如果删除时指定下标，则只删除该下标处的元素。这时要注意文件名扩展带来的负面效果。如果删除时只指定了数组名，则删除整个数组。删除时指定下标为“@”或“*”也会删除整个数组。

内部命令 declare、local、readonly 都接受“-a”选项来指定下标数组，或“-A”选项来指定键值数组。可以用内部命令 read 的“-a”选项把从标准输入中读取的一组单词赋给数组，也可以用它从标准输入中读取值后赋给指定的数组元素。内部命令 set 和 declare 可以以便于重新作为输入的格式来显示数组的值。

§6.8 目录栈

目录栈是一组最近访问过的目录。内部命令 pushd 可以在更改当前目录时把目录压入到栈中；内部命令 popd 可以把指定的目录从栈中移除并把当前目录设为被移除的那个目录；而内部命令 dirs 可以显示目录栈的内容。目录栈的内容还可以从 shell 变量 DIRSTACK 中获得。

§6.8.1 用于目录栈的内部命令

A. dirs

```
dirs [+N — -N] [-clpv]
```

列出当前记住的目录。可以用 pushd 命令添加目录，而用 popd 删除。

`+N` 显示从零开始的第 N 个目录 (在不带参数执行 dirs 所列出的内容中从左开始数)。

`-N` 显示从零开始的第 N 个目录 (在不带参数执行 dirs 所列出的内容中从右开始数)。

`-c` 删除目录栈中所有目录。助记词：Clean，清除

`-l` 显示长列表；默认的列表会用波浪号来表示主目录。助记词：LongList，长列表

`-p` 列出目录时每个目录占一行。助记词：wraP，分行

`-v` 列出目录时每个目录占一行，每行前面都显示这个目录在栈中的位置。助记词：wraP，分行

B. popd

```
popd [+N — -N] [-n]
```

(如果没有参数) 删除目录栈中的顶端目录，并用 cd 命令进入到新的栈顶目录中。dirs 命令列出的目录中第一个序号为 *0*。所以，popd 就相当于 popd +0。

`+N` 删除从零开始的第 N 个目录 (在不带参数执行 dirs 所列出的内容中从左开始数)。

`-N` 删除从零开始的第 N 个目录 (在不带参数执行 dirs 所列出的内容中从右开始数)。

`-n` 在目录栈中删除目录时，禁止改变目录，即只操纵目录栈。

C. pushd

```
pushd [-n] [+N — -N — 目录]
```

在目录栈的顶端保存当前目录并进入目录中。如果没有参数，则交换栈顶的两个目录。

`-n` 在目录栈中添加目录时不按常规改变当前工作目录，而只对目录栈进行操作。

`+N` 轮转目录栈，从而把第 N 个目录移到栈顶 (在 dirs 所列出的内容中从左开始数，第一个为零)。

`+N` 轮转目录栈，从而把第 N 个目录移到栈顶 (在 dirs 所列出的内容中从右开始数，第一个为零)。

`目录` 把当前工作目录加入到栈顶，然后进入目录。

§6.9 提示符的控制[3]

Bash 在每次打印主提示符之前都会检查 PROMPT_COMMAND 变量的值。如果设置了这个变量并且其值不为空，就执行这个值中的命令，就好像这些命令是从命令行输入的一样。此外，下表列出了可以用于这个变量的特殊字符：

`\a` 响铃字符。

`\d` 当前日期，格式为“周 月 日”，例如“Tue May 26”。

`\D{格式}` 把格式传给 strftime3 并把其结果插入到提示符中；空的格式将会用当前语言区域的格式显示时间。大括号是必须的。

`\e` 转义字符。

`\h` 主机名中第一个“.”之前的部分。

`\H` 主机名。

`\j` 当前 shell 管理的作业数目。

[3] 正文标题是“提示符的控制”，但很多引用到这里的链接却认为标题是“打印提示符”。

`\l` Shell 所在终端设备的文件基名。

`\n` 换行符。

`\r` 回车符。

`\s` Shell 的名称，`$0` 的基名 (即完整文件名中最后一个斜杠后面的部分)。

`\t` *24* 小时制的当前时间，格式为“HH:MM:SS”。

`\T` *12* 小时制的当前时间，格式为“HH:MM:SS”。

`\@` *12* 小时制的当前时间，区分上午和下午。

`\A` *24* 小时制的当前时间，格式为“HH:MM”。

`\u` 当前用户的用户名。

`\v` Bash 的版本号，例如 *2.00*。

`\V` Bash 的发行号，即版本号加上补丁级别，例如 *2.00.0*。

`\w` 当前工作目录，其中的 `$HOME` 部分省略成一个波浪号 (使用 `$PROMPT_DIRTRIM` 变量)。

`\W` `$PWD` 的基名，其中的 `$HOME` 部分省略成一个波浪号。

`\!` 当前命令的历史编号。

`\#` 当前命令的命令编号。

`\$` 如果有效用户号为 *0* 就是 `#` 字符，否则就是 `$` 字符。

`\nnn` ASCII 代码为八进制数 nnn 的字符。

`\\` 一个反斜杠。

`\[` 开始一个不可打印字符的转义序列；可以在提示符中插入终端控制字符序列。

`\]` 结束一个不可打印字符的转义序列。

命令编号和历史编号通常是不一样的：一个命令的历史编号是它在历史中的位置，历史中可能包含了从历史文件中读取的命令 (参见§9.1[Bash 的历史功能], *p90*)；而命令编号是指在当前 shell 会话中已经执行的所有命令中的序号。

这个字符串在解析以后还要根据 shell 的 `promptvars` 选项 (参见§87[内部命令 shopt], *p47*) 进行参数扩展、命令替换、算术扩展、以及引用去除 ()

✻ 代码清单 5：控制提示符实例

```
CL="\[\e[0m\]"
GREEN="$CL\[\e[0;32m\]"
BGREEN="$CL\[\e[0;32;1m\]"
XORG="$CL\[\e[0;36m\]"
```

```
XRED="$CL\[\e[0;35m\]"
BRED="$CL\[\e[0;35;1m\]"
ORG="$CL\[\e[0;33m\]"
DARK_GRAY="$CL\[\e[1;30m\]"
CYAN="$CL\[\e[1;36m\]"
BLUE="$CL\[\e[1;34m\]"

# 为了显示方便，下面一行用命令替换进行赋值。实际使用时应该用单引号。
PROMPT_COMMAND=$(
    echo -ne "\033]0;${USER}@${HOSTNAME}:  ${PWD}\007"
    NTTY=$(tty | cut -d"/" -f3-4)
    LS=$(ls | wc -l)
    LSA=$(ls -a | wc -l)

    L1a="$BLUE[$BGREEN\u$GREEN@\h:$NTTY\s$BLUE]$BLUE"
    L1b="$ORG\t$BLUE"
    L1c="$BLUE<$BRED\w$BLUE>$BLUE"
    L1d="($XRED$LS/$LSA$BLUE)$BLUE"
    L2="$CYAN\\\$$CL"
    export PS1="$L1a-$L1b-$L1c-$L1d-\n$L2 "
    #export PS1="[\u@\h:$NTTY\s]-\t-<\w:$LS/$LSA>-\n\\$ "
    history -a
)
# 实际使用时，上面一行也应该改成单引号
```

§6.10 受限制的 shell

如果通过 rbash 来启动 Bash，或者启动时指定了“--restricted”或“-r”选项，则 shell 就进入受限模式。受限制的 shell 可以用来设置一个控制更严格的环境；它在行为上和 bash 完全一样，除了下面允许和不允许的操作：

- ☞ 用内部命令 cd 改变目录。
- ☞ 设置或重置变量 SHELL、PATH、ENV、BASH_ENV 的值。
- ☞ 使用包含斜杠的命令名。
- ☞ 在内部命令 . 的参数中指定带有斜杠的文件名。
- ☞ 在内部命令 hash 的“-p”选项参数中指定带有斜杠的文件名。
- ☞ 启动时从 shell 环境中导入函数定义。
- ☞ 启动时解析 shell 环境中的 SHELLOPTS。
- ☞ 使用重定向运算符“>”、“>|”、“<>”、“>&”、‘&>”进行重定向。
- ☞ 使用内部命令 exec 把当前的 shell 换成另外一个命令。
- ☞ 使用内部命令 enable 的“-f”或“-d”选项增加或删除内部命令。
- ☞ 使用内部命令 enable 来启用已经禁用的 shell 内部命令。
- ☞ 指定内部命令 command 的“-p”选项。

☞ 使用 `set +r` 或者 `set +o restricted` 来取消受限模式。

这些限制是在读取启动文件以后生效的。如果要执行的命令是个 shell 脚本 (参见§3.8[Shell 脚本], *p26*)，`rbash` 就会创建一个不带任何限制的 shell 来执行这个脚本。

§6.11 Bash 的 POSIX 模式

在 Bash 启动时指定“`--posix`”命令行选项，或者在 Bash 运行时执行 `set -o posix` 命令，都会使它改变那些与 POSIX 规范不一致的行为，从而更接近于 POSIX 规范。如果作为 `sh` 来启动，Bash 就会在读取启动文件之后进入 POSIX 模式。下面是这种模式中改变的行为：

1. 如果散表列中的命令不存在了，Bash 会在 `$PATH` 中重新搜索基路径。这个行为也可以用 `shopt -s checkhash` 得到。
2. 如果作业退出时返回状态不为零，作业控制代码及其内部命令打印的信息是“已完成 (状态号)”。
3. 作业退出时，作业控制代码及其内部命令打印的信息是“已停止 (信号名)”，其中信号名是诸如 `SIGTSTP` 等名称。
4. 内部命令 `bg` 使用指定的格式显示后台的每个作业，但不没有指明哪个是当前作业，哪个是前一个作业。
5. 在能够识别保留字的上下文中出现的保留字不会进行别名扩展。
6. 启用了 POSIX 方式来扩展 `PS1` 和 `PS2`，使得 `!` 扩展成历史编号，而 `!!` 扩展成 `!`。并且不管 `promptvars` 选项如何设置，都会对 `PS1` 和 `PS2` 的值进行参数扩展。
7. 执行 POSIX 的启动文件 (`$ENV`) 而不是正常的 Bash 文件。
8. 只对命令名之前的赋值进行波浪号扩展，而不是对本行中所有赋值都进行。
9. 默认的历史文件是 `\/.sh_history` (这是 `$HISTFILE` 的默认值)。
10. `kill -l` 的输出是在单行中打印所有信号名，其间用空格分隔，并且不带 `SIG` 前缀。
11. 内部命令 `kill` 不接受带有 `SIG` 前缀的信号名。
12. 在执行“`.␣文件名`”时如果文件名不存在，则非交互的 shell 将会退出。
13. 非交互的 shell 在进行算术扩展时如果遇到无效的表达式而产生语法错误就会退出。
14. 重定向运算符不会对重定向中的单词进行文件名扩展，除非在交互式的 shell 中。
15. 重定向运算符不会对重定向中的单词进行单词拆分。
16. 函数名必须是有效的 shell 标志符；即它们不能包含字母、数字、下划线以外的字符，也不能以数字开头。在非交互的 shell 中如果定义的函数名无效则会导致一个严重错误。
17. 在搜索命令时，POSIX 的特殊命令会在函数名之前找到。
18. 如果 POSIX 的特殊命令返回错误的状态，则非交互的 shell 就会退出。严重错误是指 POSIX 规范中列出的那些，包括指定了不正确的选项，重定向错误，命令名之前的变量赋值错误，等等。
19. 如果设置了 `CDPATH`，内部命令 `cd` 就不会在其后隐式附加当前目录。这意味着如果在 `$CDPATH` 的任一条目中都找不到一个有效的目录，`cd` 就会失败，即使其参数指定的目录在当前目录中存在。
20. 如果变量赋值时发生错误并且其后没有跟着命令名，非交互的 shell 就会退出并返回错误状态。变量赋值会发生错误的例子包括试图给只读变量赋值。

21. 如果 `for` 语句中的循环变量或者 `select` 语句中的选择变量是只读变量，则非交互的 shell 就会退出雅返回错误状态。
22. 不可以使用远程替换。
23. 在 POSIX 特殊内部命令之前的赋值语句，在命令执行完毕以后还将持续保留在 shell 环境中。
24. 在 shell 函数调用之前的赋值语句，在函数返回以后还将持续保留在 shell 环境中，不好像执行了 POSIX 特殊内部命令。
25. 内部命令 `export` 和 `readonly` 会按照 POSIX 要求的格式输出内容。
26. 内部命令 `trap` 显示的信号名称不带有 `SIG` 前缀。
27. 内部命令 `trap` 不会检查其第一个参数是否是一个信号指示并且在是信号指示时恢复该信号的处理，除非这个参数只含有数字并且指定了一个有效的信号。如果用户希望把某个信号的处理重围为原来的程序，则应该用“`-`”作为第一个参数。
28. 如果 `PATH` 中不包括当前路径，则内部命令 `.` 和 `source` 就不会在当前目录中搜索文件名参数。
29. 为执行命令替换而创建的子 shell 会继承父 shell 的“`-e`”选项。如果不是在 POSIX 模式中，Bash 会在这种子 shell 中重围“`-e`”选项。
30. 总是启用别名扩展，即使是在非交互的 shell 中。
31. 当用内部命令 `alias` 显示别名定义时，不会在每条输出前加上前导的“alias␣”，除非指定了“`-p`”选项。
32. 如果启动内部命令 `set` 时没有指定选项，它不会显示 shell 函数的名称和定义。
33. 如果启动内部命令 `set` 时没有指定选项，它在显示变量值时就不用引用，即使其中含有不可打印字符，除非这个值中含有 shell 元字符。
34. 如果使用了内部命令 `cd` 的逻辑路径模式，并且由 `$PWD` 和其参数指定的目录名构成的路径不存在，`cd` 将会失败，而不是继续尝试物理路径模式。
35. 如果内部命令 `pwd` 指定了“`-P`”选项，它就会把 `$PWD` 转换成不带符号链接的路径后输出。
36. 内部命令 `pwd` 会检查其打印的路径是否和当前目录相同，即使没有用“`-P`”选项让它去检查文件系统。
37. 在列出历史条目时，内部命令 `fc` 不会指示每个条目是否已经被修改过。
38. `fc` 默认使用的编辑器是 `ed`。
39. 内部命令 `type` 和 `command` 不会报告不可执行的文件，尽管在 `$PATH` 中只有这个文件的名称匹配时 shell 会试图去执行它。
40. 在 vi 编辑模式中，“`v`”命令会直接启动 vi 编辑器，而不是去检查 `$VISUAL` 和 `EDITOR`。
41. 如果打开了 `xpg_echo` 选项，Bash 不会把 `echo` 命令的任何参数当前选项；而是对每个参数进行转义处理以后直接显示。
42. 内部命令 `ulimit` 的“`-c`”和“`-f`”选项使用 *512* 字节的块。

还有一些 POSIX 行为，即使是 Bash 的 `posix` 模式在默认情况下也没有实现的。特别的：

1. 当一个程序要编辑历史条目并且这时没有设置 `FCEDIT` 变量时，内部命令 `fc` 会再去检查 `$EDITOR`，而不是直接使用默认的 `ed`。只有在没有设置 `EDITOR` 时 `fc` 才会使用 `ed`。
2. 如上所说，Bash 需要打开 `xpg_echo` 选项才能让内部命令 `echo` 完全遵循规范。

在编译时，可以指定 `--enable-strict-posix-default` (参见§10.8[配置选项], *p98*) 把 Bash 配置成默认就支持 POSIX。

第七章　作业控制

本节讨论作业控制是什么、它怎么工作、以及 Bash 里面怎么使用这些功能。

§7.1　作业控制基础

作业控制是指有选择的停止 (暂停) 并在后来继续 (恢复) 执行某个进程的能力。通常，用户通过 Bash 和系统的终端驱动共同提供的功能，在交互式的界面上使用作业控制。Shell 会把每个管道都和一个关联。它会维护一个当前正在执行的作业表，这个表可以用 `jobs` 命令列出。当 Bash 异步的启动一个作业时，它会打印一条如下的信息：

```
[1] 25647
```

表示这个作业的作业号是 *1*，而这个作业所关联的管道中最后一个进程的进程号是 *25647*。一个管道中的所有进程都是同一作业的组成部分。Bash 使用作业这种抽象的机制作为作业控制的基础。

为了便于实现作业控制的用户界面，操作系统维护一个叫当前终端进程组号的概念。这个进程组的成员 (即进程组号与当前终端进程组号相等的进程) 接收诸如 `SIGINT` 的键盘信号，它们被称为前台进程。而后台进程是指那些进程组号与终端进程组号不同的进程，它们不受键盘信号的影响。只有前台进程才可以读取和写入终端。如果后台进程试图读取 (或写入) 终端，终端驱动就会向它们发送一个 `SIGTTIN` 或 `SIGTTOU` 信号；这时，如果没有捕获这个信号，这个进程就会暂停。

如果运行 Bash 的操作系统支持作业控制，Bash 就会提供作业控制的功能。在某个进程运行时输入暂停字符 (通常是 ^Z，即 C-Z)，这个进程就会停止并且把控制权返回给 Bash。如果输入延迟暂停字符 (通常是 ^Y 或 C-Y)，这个进程就会在试图从终端读取输入时停止并且把控制权返回给 Bash。这时用户就可以控制该作业的状态：用 `bg` 命令在后台继续运行作业，用 `fg` 命令在前台继续运行作业，或者用 `kill` 命令结束作业。^Z 会立即生效，并有一个副作用，即丢弃剩余的输出和尚未提交的输入。

有好多方法来表示 shell 中的作业。“`%`” 字符引导一个作业指示 (`jobspec`)。作业号 *n* 可以记为 `%n`。`%%` 和 `%+` 代表 shell 概念中的当前作业，即前台中停止的最后一个作业或后台中最后一个开始的作业。单个 `%` (后面没有作业指示) 也表示当前作业。而前一个作业可以用 `%-` 来表示。如果只有一个作业，则 `%+` 和 `%-` 都可以表示它。在 `jobs` 命令的输出中，当前作业总是标为 `+`，而前一个作业总是 `-`。

还可以用启动时名称中的前缀来表示一个作业，或者用命令行中的子字符串。例如，`%ce` 表示已停止的进程 `ce`。而使用 `%?ce` 却表示任何包含字符串 `ce` 的命令。如果前缀或子字符串匹配多个作业，Bash 就会报错。

简单的称呼一个作业可以把它调到前台，例如，`%1` 和 `fg %1` 是同义的，它们都把后台中的第一个作业调到前台。类似的，`%1 &` 可以在后台继续执行第一个作业，它和 `bg %1` 是等价的。

Shell 会即时知悉作业状态的改变。通常，Bash 会等待打印提示符时再报告作业状态的改变，以避免干扰用户的其它输出。如果打开了内部命令 `set` 的“`-k`”选项 (参见 §86[内部命令 set], *p42*)，Bash 会立即报告状态的改变。如果有 `SIGCHLD` 陷阱，则在每个子进程退出时执行。

如果作业停止时试图退出 Bash (如果打开了 `checkjobs` 选项，则也包括作业正在运行时；参见 §87[内部命令 shopt], *p45*)，shell 就会打印警告信息；这时如果打开了 `checkjobs` 选项，则列出每个作业和它们

的状态。然后，可以用 jobs 命令查看它们的状态[1]。如果没有输入其它命令而再次试图退出，Bash 就不再打印警告信息，并结束所有已停止的作业。

§7.2 作业控制内部命令

A. bg

```
bg [作业指示 ···]
```

在后台继续执行每个暂停的作业指示，就好像启动它们时带有“&”一样。如果没有给定作业指示，则使用当前的作业。返回状态是零，除非运行时没有启用作业控制，或者虽然启用了作业控制而有些作业指示没有找到或它们在启动时没有使用作业控制。

B. fg

```
fg [作业指示]
```

在前台继续执行作业指示，并把它作为当前作业。如果没有给定作业指示，则使用当前的作业。返回状态是放到前台的命令的返回状态；除非运行时没有启用作业控制，或者虽然启用了作业控制而有些作业指示没有找到或它们在启动时没有使用作业控制，这时返回状态是非零。

C. jobs

```
jobs [-lnprs] [作业指示]
jobs -x 命令 [参数表]
```

第一种形式列出活动和作业；其选项具有下列含义：

`-l` 除了正常要显示的信息外，还列出进程号。助记词：List，列出

`-n` 只显示上次把状态通知用户以后，已经改变了状态的作业。助记词：Notify，上次通知

`-p` 只列出作业进程组中首领进程的进程号。助记词：Process，进程号

`-r` 只显示正在运行的作业。助记词：Running，正在运行

`-s` 只显示已经停止的作业。助记词：Stopped，已停止

如果给定作业指示，则只显示该作业的信息。否则，列出全部作业的状态信息。

如果指定了“-x”选项，jobs 就会把命令或参数表中的作业指示用对应的进程组号替换，然后把参数表传给命令并执行它，最后返回这个命令的返回状态。

[1] 如果没有打开 checkjobs 选项。

D. kill

```
kill [-s 信号指示] [-n 信号数字] [-信号指示] 作业指示或进程号
kill -l [退出状态]
```

把信号指示或信号数字指定的信号发送给作业指示或进程号指定的进程。信号指示是个不区分大小写的信号名称，例如 SIGINT (带有或没有 SIG 前缀)，或者是个信号代码；信号数字是个信号代码。如果没有指定信号指示或信号数字，则使用 SIGTERM。“-l”选项助记词：List，列出可以列出所有信号名称。如果“-l”选项还带有一些参数，则只列出这些参数对应的信号，这时返回状态是零。退出状态是个信号代码或者是能导致进程结束并返回这个退出状态的信号。如果至少成功发送一个信号，则返回状态是零；如果发生错误，或遇到了无效的选项，则返回非零。

E. wait

```
wait [作业指示或进程号 ···]
```

等待由作业指示或进程号指定的进程退出并返回等待的最后一个命令的退出状态。如果给定作业指示，则等待这个作业的所有进程。如果没有给定参数，则等待当前所有的活动子进程，其返回值为零。如果作业指示或进程号都没有指定 shell 的活动子进程，则返回状态是 *127*。

F. disown

```
disown [-ar] [-h] [作业指示 ···]
```

如果没有选项，则从活动作业表中移除第一个作业指示。如果指定了“-h”选项助记词：Hang，挂起，则并不移除作业，而是给它一个标志，使得 shell 在接收到 SIGHUP 信号时不会把这个信号转发给它。如果没有指定作业指示，并且也没有指定“-a”或“-r”选项，则使用当前作业。如果没有指定作业指示，则“-a”选项助记词：All，所有会移除或标志所有作业；而“-r”限制只操作正在运行的作业。

G. suspend

```
suspend [-f]
```

暂停执行当前的 shell，直到向它发送 SIGCOUNT 信号。登录 shell 不可以暂停；但是可以使用“-f”选项助记词：Force，强制来强制暂停。

[分节结束]

如果没有启用作业控制，内部命令 kill 和 wait 就不能把作业指示作为参数，它们只能接受进程号。

§7.3 作业控制变量

A. auto_resume

这个变量控制 shell 与用户和作业控制的交互。如果这个变量存在，则没有重定向且只包含单个单词的简单命令就被当作恢复已有作业的候选命令。这时不允许有岐义；如果有多个命令以输入的字符串开头，则选择最近访问的作业。在这种情况下，对于已经停止的作业，其名称就是启动时的命令行。如果把这个变量的值设为 exact，则输入的字符串必须和已停止的作业名完全一致；如果把它设为 substring，输入的字符串只要和已停止作业名的部分匹配就可以了；substring 这个值在功能上和作业号 %? 相类似 (参见§7.1[作业控制基础], *p69*)。如果设为任何其它的值，则必须是一个已停止作业名称中的开头部分；这和作业号 % 在功能上是类似的。

第八章　编辑命令行

本章介绍 GNU 命令行编辑界面的基本功能。命令行编辑是 Readline 库提供的；这个库被几个不同的程序共用，Bash 是其中一个。使用交互式的 shell 时，默认已经打开了命令行编辑，除非启动 shell 时指定了“`--noediting`”选项。当使用内部命令 `read` 的“`-e`”选项 (参见§4.2[Bash 的内部命令 `read`], *p39*) 时也会使用行编辑。默认情况下，行编辑命令和 emacs 的很相似；但也可以使用 vi 风格的行编辑界面。在任何时候，都可以使用内部命令 `set` (参见§4.3.1[内部命令 set], *p42*) 的“`-o emacs`”或“`-o vi`”选项来打开行编辑，或者使用 `set` 的“`+o emacs`”或“`+o vi`”选项来关闭。

§8.1　行编辑介绍

下面几段介绍键的表示方法。

字符 C-k 读作“`Control-K`”，它表示按下 Control 键时再按 k 键所得到的字符。符号 M-k 读作“`Meta-K`”，它表示按下 Meta 键 (如果有这个键) 时再按 k 键所得到的字符。在很多键盘上，Meta 键都标上 ALT 。如果一个键盘有两个标为 ALT 的键 (通常在空格键的两旁)，则一般左边的那个可以当 Meta 键使用。右边的那个 ALT 键也可以配置成 Meta 键，或者配置成其它的修饰键，例如用来输入字母音符的 Compose 键。如果没有 Meta 或 ALT 键，也没有其它可以当成 Meta 键的，则可以先按下 ESC 再按下 k 来得到同样的键。这两种键都叫 Meta 化的 k 键。

字符 M-C-k 读作“`Meta-Control-K`”，它表示 Meta 化的 C-k 。

此外，还有几个键有独特的名称。特别的，本文或初始化文件 (参见§8.3[Readline 的启动脚本], *p74*) 中出现的 DEL 、 ESC 、 LFD 、 SPC 、 RET 和 TAB 都表示其自身。如果键盘上没有 LFD 键，输入 C-j 也会得到同样的字符。在有些键盘上，RET 键可能被标为 Return 或 Enter 键。

§8.2　与 Readline 的交互

在交互式的会话中，常常是输入了很长的一行文本，却发现这行的第一个单词拼写错了。Readline 提供了一套在输入时控制文本的命令，用来改正输入错误，而不必要重新输入大部分内容。使用这些编辑命令，可以把光标移动到需要更正的地方，删除或者插入更正的内容；然后，如果文本行还令人满意，就按下 RET 。按下 RET 不一定要在行的结尾；整行都会读入，不管光标在行中的什么地方。

§8.2.1　Readline 的基础

要在行中输入字符，只需要按下对应的键。输入的字符在光标后出现，然后光标就向后移动一格的位置。如果输错了一个字符，可以用删除字符后退并删除这个输错的字符。有时，输错了不会马上发现，直到又输入了好个几个字符。这种情况下，可以输入 C-b 向左移动光标然后更正错误。接下来，可以用 C-f 向右移动光标。

如果在行的中间输入，就会发现光标右边的字符被“推过去”以腾出空间容纳刚刚输入的文本。同样的，删除光标下的字符时，光标右边的字符也会被“推过来”以填入删除后留出的空白中。下面是编辑输入文本行用到的最基础的命令：

C-b 向后移动一个字符 (的位置)。

C-f 向后移动一个字符 (的位置)。

DEL 或 Backspace 删除光标左边的字符。

C-d 删除光标下面的字符。

可打印字符 在光标处插入该字符。

C-_ 或 C-x C-u 取消最后一次输入命令。可以一直取消直到行为空。

(取决于配置，Backspace 可以设为删除光标左边的字符，而 DEL 设为和 C-d 一样删除光标下面的字符，而不是其左边的。)

§8.2.2 Readline 的移动命令

上面列出了编辑输入行时要用到的最基本的键。为了方便，除了 C-b 、C-f 、C-d 、DEL 以外，还有其它编辑命令。下面这些命令可以用来在行内部快速移动。

C-a 移到行的开头。

C-e 移到行的结尾。

M-f 向前移动一个单词 (的位置)；单词是由字母和数字构成的。

M-b 向后移动一个单词 (的位置)。

C-l 清除屏幕，然后在顶端打印当前行。

注意 C-f 向前移动一个字符，而 M-f 向前移动一个单词。有个很松散的习惯，就是 Control 键操纵字符而 Meta 键操纵单词。

§8.2.3 Readline 的删除命令

删除 (“kill”) 是指把文本从行中移除并保存下来以备后用，通常是后来重新插入 (“yank”) 到行中 (现在常说“剪切”[1]和“复制”)。如果介绍命令时说它“删除”文本，则可以保证在另外一个地方 (或者同一地方) 恢复这些文本。使用删除命令时，文本被放在删除环中。后面连续删除的文本也会被放在一起，所以重新插入时，会一下子恢复所有文本。删除环不是针对行的；在前面一行删除的文本可以在以后输入另外一行时恢复。

下面的命令用来删除文本。

C-k 从当前光标的位置删除到行的结尾。

M-d 从光标的位置删除到当前单词的结尾；如果是在单词之间，则删除到下一个单词的结尾。单词界限和 M-f 所使用的一样。

[1] 剪切的文本一般直接丢弃，而删除的会保存在删除环中。

M-DEL 从光标删除到当前单词的开头；如果是在单词之间，则删除到止上个单词的开头。单词界限和 M-b 所使用的一样。

C-w 从光标删除到上一个空白符。这和 M-DEL 使用的单词界限是不一样的。

下面这些命令用来在行中恢复文本。重新插入是指从删除缓存中复制最后被删除的文本。

C-y 把最近删除的文本插入到光标所在的缓存中。

M-y 循环到删除环，并插入新的顶端文本。只有当前一个命令是 C-y 或者 M-y 时才可以用这个命令。

§8.2.4 Readline 的参数

Readline 命令可以接受数值型参数。有时这些参数可以表示重复次数，有时数值的正负符号很重要。如果一个通常向前进行的命令得到了一个负的参数，则它会向后进行。例如，删除到行开头的文本，可以使用“M-- C-k”。

给命令指定参数的一般方法是在命令前面输入 Meta 化的数字。如果输入的第一个“数位”是负号“-”，则这个参数将是负数。开始输入了参数的第一个 Meta 化的数字后，就可以输入其余数字，然后再输入命令。例如，给 C-d 命令以参数 *10*，则可以输入“M-1 0 C-d”，这会删除输入行中后面十个字符。

§8.2.5 在历史中搜索命令

Readline 提供的可以搜索历史 (参见§9.1[Bash 的历史功能], *p90*) 的命令以寻找包含指定字符串的行。搜索命令有两种，增量的和非增量的。

增量搜索在用户输入搜索字符串结束前就开始搜索。第输入字符串的一个字符时，Readline 都会显示历史中与已经输入的字符串匹配的行。增量搜索时，实际需要输入多少字符就输入多少，就能找到想要的历史。在历史中向后搜索包含特定字符串的行可以用 C-r；而 C-s 表示向前搜索。`isearch-terminators` 变量中的字符可以用来结束增量搜索。如果这个变量没有设置值，就用 ESC 或 C-j 来结束增量搜索。C-g 可以退出增量搜索并恢复之前的行。搜索结束以后，历史中包含搜索字符串的行就成为当前行。

如果要在历史中搜索其它匹配的行，可以根据需要再输入 C-r 或 C-s；这会在历史中向前或向后搜索与已经输入的搜索字符串匹配的下一行。与 Readline 命令绑定的任何其它键序列都会结束搜索并执行搜索到的命令。例如，RET 会结束搜索并提交一整行，所以会执行历史中的对应命令。移动命令也会结束搜索，并把最后找到的行作为当前行后开始编辑。

Readline 会记住最后一次增量搜索的字符串。如果输入的两个 C-r 之间没有其它搜索字符串，就会使用已经记住的字符串。

非增量搜索在开始搜索匹配的历史行之前要读取整个搜索字符串。搜索字符串可以由用户输入，也可以的当前行的部分内容。

§8.3 Readline 的启动脚本

虽然 Readline 库默认安装了一套 eamcs 风格的键绑定，使用其它的键绑定也是可以的。任何用户都可以在一个 `inputrc` 文件中放入 Readline 命令来对使用 Readline 的程序进行自定义；这个文件通常在用户的主目录内。文件的名称来自 shell 变量 `INPUTRC` 的值。如果这个变量没有设置，则使用默认的“`~/.inputrc`”；如果这个文件不存在或者不可读取，则最终的默认值是“`/etc/inputrc`”。

一个使用 Readline 库的程序在启动时会读取初始化文件并绑定键。此外，C-x C-r 会重新读入初始化文件，所以会使用任何改动生效。

§8.3.1 Readline 启动脚本的语法

Readline 的初始化文件里面只能使用一些基本的结构。空行会被忽略掉。以“#”开头的行是注释。以“$”开头的行表示条件结构 (参见§8.3.2[Readline 启动脚本的条件结构], *p78*)。其它行表示变量赋值和键绑定。

A. 变量赋值

可以用在初始化脚本中用 Readline 的 set 命令修改变量的值来改变 Readline 的运行时行为。其语法很简单：

```
set 变量 值
```

例如，下面就示范了怎么把默认的 eamcs 风格键绑定改成使用 vi 风格的行编辑命令：

```
set editing-mode vi
```

在解析变量的名称和值时会根据情况忽略大小写。不能识别的变量会被忽略。对于布尔型的变量 (即可以打开或关闭的)，如果其值没有设置、或者为空、或者是 on (不区分大小写)、或者是 *1*，就会打开。任何其它值都会关闭这个变量。命令 bind -V 可以列出当前的 Readline 变量和值 (参见§4.2[Bash 的内部命令], *p34*)。

大量的运行时行为都可以由下列变量来更改。

bell-style 控制 Readline 想让终端铃声响起时发生的动作。如果设为 none，则不响铃；如果设为 visible，则如果有可见响铃就使用可见响铃[2]；如果设为 audible (默认的)，Readline 会试图让终端的铃声响起。助记词：响铃方式

bind-tty-special-chars 如果设为 on，Readline 会试图把内核的终端驱动程序要特殊处理的字符映射到自己对应的字符上。助记词：绑定终端特殊字符

comment-begin 在执行 insert-comment 命令时要插入行开头的字符；默认值是“#”。助记词：注释开始

completion-ignore-case 如果设为 on，Readline 在进行文件名匹配和补全时会忽略大小写。默认值是 off。助记词：忽略补全大小写

completion-prefix-display-length 补全列表中不加修改而直接显示的公共前缀的字符长度。如果设为大于零的值，则比这个值大的公共前缀将会在列出补全时被替换成省略号。助记词：补全前缀显示长度

completion-query-items 决定询问用户是否需要显示补全列表时补全列表的长度。如果补全列表的长度大于这个值，Readline 会询问用户是否想查看；否则，就直接显示列表。这个变量必须设置一个大于或等于 *0* 的整数。负值表示 Readline 不会询问。默认值是 *100*。助记词：补全询问长度

convert-meta 如果设为 on，Readline 会把设置了高八位的字符的第八位 (最高位) 去掉并加上一个前导的 ESC 字符，从而把它们转换成 Meta 化的 ASCII 字符序列。默认值是 on。助记词：Meta 转换

disable-completion 如果设为 on，Readline 将不会补全单词。补全字符会被插入行中，就好像这些字符也绑定了 self-insert 命令。默认值是 off。助记词：禁用补全

editing-mode 这个变量控制默认使用哪种键绑定。默认情况下，Readline 启动 emacs 风格的编辑模式，其中的键很像 emacs。这个变量可以被设为 emacs 或 vi。助记词：编辑模式

[2] 大部分情况下都是让终端闪烁几下。

`enable-keypad` 如果设为 `on`，Readline 在启动时启用小键盘。有些系统依赖小键盘上的方向键。默认值是 `off`。助记词：启用小键盘

`expand-tilde` 如果设为 `on`，Readline 在试图对单词进行补全时要进行波浪号扩展。默认值是 `off`。助记词：扩展波浪号

`history-preserve-point` 如果设为 `on`，则对于通过 `previous-history` 或 `next-history` 得到的每个历史行，标志点 (光标的当前位置) 将会停留在行中同样的地方。默认值是 `off`。助记词：保持历史标志点

`history-size` 历史列表中所保存的历史行的最大数目。如果设为零，则历史列表中保存的历史行不受限制。助记词：历史长度

`horizontal-scroll-mode` 可以设为 `on` 或 `off`。如果设为 `on`，则如果正在编辑的行比屏幕宽，就会在单行内水平滚动，而不是开始新行。默认值是 `off`。助记词：水平滚动模式

`input-meta` 如果设为 `on`，Readline 会启用八位字符的输入 (即不会清除每个所读入字符的第八位)，不管终端支持与否。默认值是 `off`。`meta-flag` 和它是同义的。助记词：Meta 输入

`isearch-terminators` 结束增量搜索而不会把字符当作命令去执行 (参见§8.2.5[在历史中搜索命令], *p74*) 的一个字符串。如果没有设置这个变量，则 ESC 和 C-J 字符会结束增量搜索。助记词：增量搜索结束符

`keymap` 设置 Readline 当前用来绑定命令的键映射。可用的键映射有 `emacs`、`emacs-standard`、`emacs-meta`、`emacs-ctlx`、`vi`、`vi-move`、`vi-command`、`vi-insert`。其中，`vi` 和 `vi-command` 是等价的，`emacs` 和 `emacs-standard` 也是等价的。默认值是 `emacs`。`editing-mode` 变量的值也会影响默认的键映射。助记词：键映射

`mark-directories` 如果设为 `on`，在补全后的目录后面加上斜杠。默认值是 `off`。助记词：标志出目录

`mark-modified-lines` 如果设为 `on`，Readline 将会在已被修改的历史行开头显示一个星号 (“`*`”)。默认值是 `off`。助记词：标志已修改的行

`mark-symlinked-directories` 如果设为 `on`，并且补全后的名称是个指向目录的符号链接，则在后面加上斜杠 (还要看 `mark-directories` 的值)。默认值是 `off`。助记词：标志目录的符号链接

`match-hidden-files` 如果设为 `on`，Readline 在补全文件名时会匹配以“`.`”开头的文件 (即隐藏文件)，除非用户在要补全的文件名开头指定了“`.`”。默认值是 `on`。助记词：匹配隐藏文件

`output-meta` 如果设为 `on`，Readline 会直接显示设置了高八位的字符，而不是显示 Meta 化的序列。默认值是 `off`。助记词：Meta 输出

`page-completions` 如果设为 `on`，Readline 会使用其内部类似于 `more` 命令的分页程序来显示一次性满屏的补全。默认值是 `on`。助记词：补全分页

`print-completions-horizontally` 如果设为 `on`，Readline 会把匹配的补全条目按字母顺序排列并水平显示，而不是在屏幕中垂直显示。默认值是 `off`。助记词：水平显示补全

`revert-all-at-newline` 如果设为 `on`，Readline 会在执行 `accept-line` 时恢复历史行的所有改动。默认情况下，改动过的历史行在多次使时可以各自恢复。默认值是 `off`。助记词：提交时恢复所有历史

`show-all-if-ambiguous` 这会改变补全命令的默认行为。如果设为 `on`，则如一个单词有多个匹配将会立即被全部列出，而不是响铃。默认值是 `off`。助记词：如有岐义就显示全部

`show-all-if-unmodified` 这会以类似于 `show-all-if-ambiguous` 的方式改变补全命令的默认行为。如果设为 `on`，则如一个单词有多个匹配，并且它们都不是部分匹配 (即匹配的条目不含有公共前缀) 将会立即被全部列出，而不是响铃。默认值是 `off`。助记词：如未修改则显示全部

`visible-stats` 如果设为 `on`，则在列出补全条目时在文件名后面加上一个用以显示其类别的字符。默认值是 `off`。助记词：可见的类型

B. 键绑定

初始化文件中控制键绑定的语法是很简单的。首先要找到需要使用的命令名。下面各节中有命令名的列表，以及命令功能的简单描述；如果有键绑定，也一并列出。知道了命令名称以后，只要在初始化文件的某一行中写入要和该命令绑定的键，后面加个冒号，然后再写上这个命令就可以了。在键名和冒号之间不可以有空格，否则这些空格也会成为键名的一部分。键名可以有不同的表示方法，自己看着舒服的那种就行。

除了可以绑定命令名称，Readline 还允许绑定一个字符串，这样键入绑定的键名 (宏) 后就能插入这个字符串。`bind -p` 命令可以输出 Readline 中的命令和绑定；输出的格式可以直接放在初始化文件中 (参见§4.2[Bash 的内部命令 `bind`], *p34*)。

键名：命令名或宏 键名是英语中一个键的名称。例如：

```
Control-u: universal-argument
Meta-Rubout: backward-kill-word
Control-o: "> output"
```

在上面的例子中，C-u 绑定到了 `universal-argument` 命令上，M-DEL 绑定到了 `backward-kill-word` 命令，而 C-o 绑定可以运行其右边所写的宏 (即把文本"`> output`"插入到行中)。在处理绑定时可以使用一些键符号：DEL、ESC、ESCAPE、LFD、NEWLINE、RET、RETURN、RUBOUT、SPACE、SPC、以及 TAB。

键序列：命令名或宏 键序列和上面的键名不同之处在于它可以把键名序列放在双引号中间的字符串来表示整个键序列。这里可以使用 emacs 风格的转义键名，如下例所示，但不能使用特殊的键名。

```
"\\C-u": universal-argument
"\\C-x\\C-r": re-read-init-file
"\\e[11~": "Function Key 1"
```

在上例中，C-u 仍绑定到 `universal-argument` 命令 (同前一个例子)；C-x C-r 绑定了 `re-read-init-file` 命令；而 ESC [11 ~ 绑定可以插入文本"Function Key 1"。

指定键序列时可以使用下列 emacs 风格的转义序列：

- \C- 控制键前缀。
- \M- Meta 前缀。
- \e 一个转义字符。
- \\ 反斜杠。
- \" 双引号。
- \' 单引号或省字符。

除了 emacs 风格的转义序列，还可以用另外一套转义序列：

`\a` 警告 (响铃)

`\b` 退格删除

`\d` 删除

`\f` 走纸换页

`\n` 新行

`\r` 换行

`\t` (水平) 制表符

`\v` 垂直制表符

`\nn` 由八进制数 *nnn* (一个到三个数字) 代表的一个八位字符。

`\xHH` 由十六进制数 *HH* (一个或两个十六进制数字) 代表的一个八位字符。

在宏中写入文本时必须中单引号或双引号表示宏定义。没有引用的文本会被当成命令名称。在宏里面，上面列出的转义字符会被扩展。反斜杠可以转义后面的任意字符，包括“"”和“'”。例如，下面的绑定会使用 C-x \ 能够在行中输入“\”：

```
"\C-x\\":  "\\"
```

§8.3.2 Readline 启动脚本的条件结构

Readline 实现了与 C 语言预编译器中的条件编译很神似的功能，它使得键绑定和变量赋值根据测试结果进行。它使用了四种方法指示解释器。

`$if` 这种结构根据编辑模式，所有终端或使用 readline 的程序进行绑定。测试所用的文本一直延伸到行的结尾，不需要字符去隔离它。

`mode` `mode=` 形式的 `$if` 指令用来测试 readline 使用的模式是 emacs 还是 vi。例如，它可以和“set 键映射”命令一起使用，使 readline 只有在 emacs 模式中才绑定 `emacs-standard` 和 `emacs-ctlx` 的键。

`term` `term=` 的形式可以用来绑定与终端相关联的键，例如绑定终端功能键的输出。“=”右边的单词用来和终端的完整名称以及名称中第一个“-”前面的部分进行匹配。例如，`sun` 会与 `sun` 和 `sun-cmd` 匹配。

`application` 这种结构用来包含与应用程序相关联的设置。每个使用 readline 库的程序都会设置程序名称，这个名称可以用于测试，并用来绑定只针对特定程序的键序列。例如，下面的命令在 Bash 中加入了一个键绑定用来给当前和前一个单词加引号：

```
$if Bash
# Quote the current or previous word
"\C-xq":  "\eb\"\ef\""
$endif
```

`$endif` 如上例中看到的那样，这个命令用来结束 `$if` 命令。

`$else` 这个命令 `$if` 结构的分支，当 `$if` 测试失败时执行。

`$include` 这个结构带一个文件名参数并从文件中读取和执行命令。例如，下面的命令读取“/etc/inputrc”：

```
$include /etc/inputrc
```

§8.3.3 Readline 启动脚本的例子

下面是 inputrc 文件的一个实例。它示范了键绑定，变量赋值和条件语法。

❋ 代码清单 6：Readline 启动脚本的例子

```
# vim:filetype=readline
#
# 本文件控制所有使用 readline 库的程序的行输入行为。
# 这些程序包括 FTP，Bash 和 GDB。
#
# 可以用 C-x C-r 命令重新加载该文件。
# 以 '#' 开头的行是注释。
#
# 首先，包含 /etc/Inputrc 中的任何系统绑定和变量。
$include /etc/Inputrc
#
# 设置 emacs 风格的绑定。
set editing-mode emacs

$if mode=emacs
Meta-Control-h:  backward-kill-word 命令后面的文本会被忽略掉
#
# 小键盘上的方向键
#
#"\M-OD":         backward-char
#"\M-OC":         forward-char
#"\M-OA":         previous-history
#"\M-OB":         next-history
#
# ANSI 模式的方向键
#
"\M-[D":         backward-char
"\M-[C":         forward-char
"\M-[A":         previous-history
"\M-[B":         next-history
#
# 八位小键盘上的方向键
#
#"\M-\C-OD":      backward-char
#"\M-\C-OC":      forward-char
#"\M-\C-OA":      previous-history
#"\M-\C-OB":      next-history
#
# 八位 ANSI 模式的方向键
#
#"\M-\C-[D":      backward-char
#"\M-\C-[C":      forward-char
#"\M-\C-[A":      previous-history
#"\M-\C-[B":      next-history
C-q:  quoted-insert
$endif

```

```
# 旧式的绑定。这恰好也是默认的。
TAB: complete
# 便于 shell 交互的宏。

$if Bash
# 编辑 PATH 路径
"\C-xp":  "PATH=${PATH}\e\C-e\C-a\ef\C-f"
# 准备输入引用的单词：插入引号的开始和结束，然后移到开始引号的后面。
"\C-x\"":  "\"\"\C-b"
# 插入反斜杠（测试反斜杠转义序列和宏）。
"\C-x\\":  "\\"
# 用引号引用当前或前一个单词。
"\C-xq":  "\eb\"\ef\""
# 绑定刷新本行的命令；这原来是没有绑定的。
"\C-xr":  redraw-current-line
# 编辑本行中的变量。
"\M-\C-v":  "\C-a\C-k$\C-y\M-\C-e\C-a\C-y="
$endif

# 如果可以响铃就使用
set bell-style visible
# 读取输入时不要把字符截成 7 位。
set input-meta on
# 允许插入 iso-latin1 字符，而不是把它们变成 Meta 化的序列。
set convert-meta off
# 直接显示八位的字符，而不是把它们当成 Meta 化的字符来显示。
set output-meta on
# 如果可以补全的项目超过 150 条，询问用户是否要显示全部。
set completion-query-items 150

# 用于 FTP
$if Ftp
"\C-xg":  "get \M-?"
"\C-xt":  "put \M-?"
"\M-.":  yank-last-arg
$endif
```

§8.4 可以绑定的 Readline 命令

本章介绍可以绑定键的 Readline 命令。可以使用 `bind -P` 命令列出自己的键绑定；或者用 `bind -p` 更紧凑的列出，这样还适合放到 `inputrc` 文件中 (参见§4.2[Bash 的内部命令 bind], *p34*)。

在下文中，标志点是指当前光标的位置[3]，而记号是指用 `set:mark` 保存的光标位置。标志点和记号之间的文本叫做区域。没有附带键序列的命令默认是没有绑定的。

§8.4.1 Readline 的移动命令

`beginning-of-line` (C-a) 移到到行的开头。

[3]严格的说，标志点是指活动光标的位置，这也是编辑命令要操纵的文本位置。光标会停留在一个字符的上面，而标志点可以看作是在光标下的字符与其前一个字符之间。例如，当光标在文本“frob”中的“b”上面时，标志点就在“o”和“b”之间。如果这时插入一个字符“!”，结果就是“fro!b”。这时，标志点在“!”和“b”之间，而光标还停留在“b”上面。

end-of-line (C-e) 移动到行的结尾。

forward-char (C-f) 向前移动一个字符。

backward-char (C-b) 向后移动一个字符。

forward-word (M-f) 向前移动到下一个单词的结尾。单词是由字母和数字构成的。

backward-word (M-b) 向后移动到前一个单词的开头。单词是由字母和数字构成的。

shell-forward-word 向前移动到下一个单词的结尾。单词是由未被引用的 shell 元字符分隔的。

shell-backward-word 向后移动到前一个单词的开头。单词是由未被引用的 shell 元字符分隔的。

clear-screen (C-l) 清屏并在屏幕顶端重新显示当前行，

redraw-current-line 刷新当前行。这个命令默认是没有绑定的。

§8.4.2 Readline 的历史操作命令

accept-line (Newline 或 Return) 不管光标在哪都提交本行。如果本行不是空行，就按照 HISTCONTROL 和 HISTIGNORE 变量的设置把它加入到历史中。如果本行是修改历史行得到的，就恢复历史中原来的行。

previous-history (C-p) 在历史中向“后”移动得到上一个命令。

next-history (C-n) 在历史中向“前”移动得到下一个命令。

beginning-of-history (M-<) 移动到历史的第一行 ¿aa 。

end-of-history (M->) 移动到历史的最后一行，即当前正在输入的行。

reverse-search-history (C-r) 从当前行开始向后搜索，如有必要则向“上”移动。这是增量搜索。

forward-search-history (C-s) 从当前行开始向前搜索，如有必要则向“下”移动。这是增量搜索。

non-incremental-reverse-search-history (M-p) 从当前行开始向前搜索，如有必要则向“上”移动；移动时，使用非增量搜索搜索查找用户提供的字符串。

non-incremental-forward-search-history (M-n) 从当前行开始向后搜索，如有必要则向“下”移动；移动时，使用非增量搜索搜索查找用户提供的字符串。

history-search-forward 在历史中当前行的开头和标志点之间向前搜索字符串。这是非增量搜索。这个命令默认没有绑定。

history-search-backward 在历史中当前行的开头和标志点之间向后搜索字符串。这是非增量搜索。这个命令默认没有绑定。

yank-nth-arg (M-C-y) 在标志点上插入前一个命令的第一个参数 (通常是前一行的第二个单词)。如果有参数 n，插入前一个命令的第 n 个单词 (前一行的单词是从 0 开始数的)。负的参数将插入从前一个命令的结尾开始的第 n 个单词。计算 n 的值以后，就会截取对应的参数，就好像指定了历史扩展“!n”一样。

yank-last-arg (M-. 或 M-_) 插入前一个命令的最后一个参数 (前一行的最后一个单词)。如果有参数，其行为就和 yank-nth-arg 完全一样。连续调用 yank-last-arg 会在历史中向后移动，并依次插入每一行的最后一个参数。最后一个参数是通过历史扩展机制取出来的，就好像指定了历史扩展“!$”一样。

§8.4.3 Readline 的文本修改命令

delete-char (C-d) 删除标志点处的字符。如果标志点在行的开头，行中没有字符，并且最后输入的字符没有绑定到 delete-char，则返回 EOF。

backward-delete-char (Rubout) 删除光标后面的字符。带有数值参数表示剪切这个字符，而不是删除。

forward-backward-delete-char 删除光标下面的字符；如果光标在行的结尾则删除光标后面的字符。这个命令默认没有绑定。

quoted-insert (C-q 或 C-v) 按字面意思插入后面输入的一个字符。例如，这样可以用来输入 C-q 等键序列。

self-insert (a 、 b 、 A 、 1 、 ! 、 ···) 插入这些字符本身。

transpose-chars (C-t) 把光标前的字符向前拖动到光标下的字符上，同时把光标向前移动。如果插入点在行的结尾，则交换行中最后两个字符的位置。负的参数不起作用。

transpose-words (M-t) 把标志点前的单词拖过标志点后的单词，并把标志点移过这个单词。如果插入点在行的结尾，则交换行中最后两个单词的位置。

upcase-word (M-u) 把当前 (或下一个) 单词变为大写。如果有负的参数，则把前一个单词变为大写，但不移动光标。

downcase-word (M-l) 把当前 (或下一个) 单词变为小写。如果有负的参数，则把前一个单词变为小写，但不移动光标。

capitalize-word (M-c) 把当前 (或下一个) 单词变为首字母大写。如果有负的参数，则把前一个单词首字母变为小写，但不移动光标。

overwrite-mode 切换覆盖模式。如果指定正数作为参数，则切换到覆盖模式；如果指定非正数参数，则切换到插入模式。这个命令只影响 emacs 模式，而 vi 模式有不同的覆盖方法。每次启动 Readline 时都进入插入模式。在覆盖模式中，绑定到 self-insert 的字符会覆盖标志点后的字符，而不是把文本向右推动。绑定到 backward-delete-char 的字符会把标志点前的字符用空格替换。这个命令默认没有绑定。

§8.4.4 删除和复制

kill-line (C-k) 删除从标志点开始到行结尾的文本。

backward-kill-line (C-x Rubout) 向后删除到行的开头。

unix-line-discard (C-u) 从光标位置向后删除到当前行的开始。

kill-whole-line 删除当前行中的所有字符，不管标志点在哪。这个命令默认没有绑定。

kill-word (M-d) 从标志点删除到当前单词的结尾；如果在单词之间，则删除到下一个单词的结尾。单词界限和 forward-word 相同。

backward-kill-word (M-DEL) 删除标志点后的单词。单词界限和 backward-word 相同。

shell-kill-word 从标志点删除到当前单词的结尾；如果在单词之间，则删除到下一个单词的结尾。单词界限和 shell-forward-word 相同。

backward-kill-word 删除标志点后的单词。单词界限和 shell-backward-word 相同。[4]

unix-word-rubout (C-w) 删除标志点后的单词，用空格作为单词的界限。删除后的文本保存在删除环中[5]。

unix-filename-rubout 删除标志点后的单词，用空格和斜杠作为单词的界限。删除后的文本保存在删除环中。

delete-horizontal-space 删除标志点左右的空格和制表符。这个命令默认没有绑定。

kill-region 删除当前区域中的文本。这个命令默认没有绑定。

copy-region-as-kill 把区域中的文本复制到删除环缓存中，以便可以立即粘贴。这个命令默认没有绑定。

copy-backward-word 把标志点前的单词复制到删除环缓存中，单词界限和 backward-word 相同。这个命令默认没有绑定。

copy-forward-word 把标志点后的单词复制到删除环缓存中，单词界限和 forward-word 相同。这个命令默认没有绑定。

yank (C-y) 把删除环顶端的文本复制到标志点所在的缓存中。

yank-pop (M-y) 轮询删除环[6]并复制新的顶端文本。只能在 yank 或 yank-pop 之后使用这个命令。

§8.4.5 指定数字参数

digit-argument (M-0 、 M-1 、 ··· M--) 把这个数字加入到已收集数字的参数中，或者开始一个新的数字参数。 M-- 开始一个负的参数。

universal-argument 这是另外一种指定数字的方法。如果这个命令后面有一个或多个数字，数字前面可能还有负号，这些数字就是命令的参数。如果这个命令后面是数字，则再次执行这个命令就会结束数字参数，否则就忽略这个命令。特别的，如果紧接着这个命令后的字符既不是数字也不是负号，则后面的命令参数就扩大四倍。参数初始值是一，所以第一次执行这个命令时参数就变成四，第二次就变成十六，以此类推。这个命令默认没有绑定。

[4] 这和前面的描述相冲突，应该是原文中的错误。这里可能是shell-backward-kill-word，但是 Readline 中好像没有这个命令。以前的文档中没有这句话。

[5] 事实上删除命令一般都会把删除的文本保存在删除环中。如果不这样，一般翻译成“剪切”。

[6] 即依次使用下一条文本，把原顶行移动末尾，同时把原第二行移动顶端；这样，删除环中各行文本好像在一个有方向的环中。

§8.4.6 补全命令[7]

`complete` (TAB) 试图补全标志点前的文本。实际进行的补全是应用程序相关的。补全时，Bash 依次把文本当作变量 (如果以“$”开头)、或用户名 (如果以“~”开头)、或主机名 (如果以“@”开头)、或命令名 (包括别名和函数)。如果这些都不匹配，则试图进行文件名补全。

`possible-completions` (M-?) 列出能够补全标志点前的条目。

`insert-completions` (M-*) 把 `possible-completions` 命令能生成的所有文本条目插入到标志点前。

`menu-complete` 和 `complete` 类似，但是把要补全的文本替换成补全列表中的一个条目。连续执行 `menu-complete` 会在补全列表中依次前进，每次都插入当前条目。前进到补全列表的结尾时就响铃 (取决于 `bell-style` 的设置) 并恢复原来的文本。参数 n 表示在补全列表中向前移动 n 步；负参数表示向后移动。这个命令本来是要绑定到 TAB 键，但默认没有绑定。

`delete-char-or-list` 如果不是在行的结尾或开头则删除光标下面的字符 (和 `delete-char` 一样)。如果在行的结尾，其行为就和 `possible-completions` 完全一样。这个命令默认没有绑定。

`complete-filename` (M-/) 试图对标志点前的文本进行文件名补全。

`possible-filename-completions` (C-x /) 把标志点前的文本当成文件名并列出可以补全的条目。

`complete-username` (M-~) 把标志点前的文本当成用户名并试图进行补全。

`possible-username-completions` (C-x ~) 把标志点前的文本当成用户名并列出可以补全的条目。

`complete-variable` (M-$) 把标志点前的文本当成 shell 变量并试图进行补全。

`possible-variable-completions` (C-x $) 把标志点前的文本当成 shell 变量并列出可以补全的条目。

`complete-hostname` (M-@) 把标志点前的文本当成主机名并试图进行补全。

`possible-hostname-completions` (C-x @) 把标志点前的文本当成主机名并列出可以补全的条目。

`complete-command` (M-!) 把标志点前的文本当成命令名并试图进行补全。进行命令名补全时会依次使用别名、保留字、shell 函数、shell 内部命令，最后是可执行文件名。

`possible-command-completions` (C-x !) 把标志点前的文本当成命令名并列出可以补全的条目。

`dynamic-complete-history` (M-TAB) 把标志点前的文本与历史记录里的文本行进行比较以寻找匹配并试图进行补全。

`dabbrev-expand` 把标志点前的文本与历史记录里的文本行进行比较以寻找匹配并试图列出可以进行补全的条目菜单。

`complete-into-braces` (M-{) 进行文件名补全，把可以补全的条目列表放在大括号之间，以使这个列表可以在 shell 中使用 (参见 §3.5.1[大括号扩展], *p15*)。

[7] 原文中，本节标题是“让 Readline 为你输入”，但多处对本节的引用却认为标题是“补全命令”，所以这里做了改动。

§8.4.7 键盘宏定义

start-kbd-macro (C-x ( ) 开始把输入的字符保存在当前的键盘宏中。

end-kbd-macro (C-x)) 结束把输入的字符保存在当前的键盘宏中，并保存键盘宏定义。

call-last-kbd-macro (C-x e) 重新执行刚刚定义的键盘宏，使得键盘宏的文本和用键盘输入的一样。

§8.4.8 其它功能

re-read-init-file (C-x C-r) 读入 inputrc 文件的内容，并把其中任何绑定和变量赋值合并到当前会话中。

abort (C-g) 中止当前编辑的命令并响铃 (取决于 bell-style 的配置)。

do-uppercase-version (M-a 、 M-b 、 M-x ···) 如果 Meta 化的字符是小写的，就运行对应大写字母绑定的命令。

prefix-meta (ESC) 把下一个字符 Meta 化。这是为没有 Meta 键的键盘准备的。键入 ESC f 就等价于输入 M-f 。

undo (C-_ 或 C-x C-u) 增量撤销，可以分别对每行进行。

revert-line (M-r) 撤销对本行的所有修改。这就和多次执行 undo 命令以恢复到开始是一样的。

tilde-expand (M-&) 对当前的单词进行大括号扩展。

set:mark (C-@) 在标志点设置记号。如果给定数值参数，则在指定位置设置记号。

exchange-point-and-mark (C-x C-x) 交换标志点的记号的位置。当前光标的位置变成已保存的位置；原来光标的位置设成记号。

character-search (C-]) 读取一个字符 (作为参数) 并把标志点移动到下一个同样的字符旁。负参数则向前移动。

character-search-backward (M-C-]) 读取一个字符 (作为参数) 并把标志点移动到上一个同样的字符旁。负参数则后前移动。

insert-comment (M-#) 如果没有数值参数，则把 comment-begin 变量的值插入到当前行的开始。如果指定数值参数，这个数就作为开关：如果行开头的字符和 comment-begin 变量的值不一样就插入这个值，否则就把 comment-begin 变量里的字符从行的开头删除。不管哪种情况，都提交本行，就好像输入了 Newline 一样。如果使用 comment-begin 的默认值，这个命令就会把当前行变成 shell 注释。如果带有数值参数，就删除注释字符并把本行交给 shell 执行。

dump-functions 把所有函数和它们的键绑定打印到 Readline 的输出流中。如果给定数值参数，就把输出格式化意以便于作为 inputrc 文件的一部分。这个命令默认没有绑定。

dump-variables 把所有可设置变量和它们的值打印到 Readline 的输出流中。如果给定数值参数，就把输出格式化意以便于作为 inputrc 文件的一部分。这个命令默认没有绑定。

`dump-macros` 把 Readline 中所有绑定到宏的键序和它们绑定的字符串列打印出来。如果给定数值参数，就把输出格式化意以便于作为 `inputrc` 文件的一部分。这个命令默认没有绑定。

`glob-complete-word` (M-g) 把标志点前的单词当成模式并隐式的在后面加上一个星号，然后进行文件名扩展。这个模式可以生成一系列匹配的文件名以用来补全。

`glob-expand-word` (C-x *) 把标志点前的单词当成模式并进行文件名扩展，把匹配的文件名列表插入进来并替换原来的单词。如果指定数值参数，就在文件名扩展之前先插入“*”。

`glob-list-expansions` (C-x g) 列出 `glob-expand-word` 可以生成的扩展并重新显示本行。如果指定数值参数，就在文件名扩展之前先插入“*”。

`display-shell-version` (C-x C-v) 显示当前 Bash 的版本信息。

`shell-expand-line` (M-C-e) 像 shell 一样扩展本行。它除了进行所有的 shell 扩展 (参见§3.5[Shell 扩展], *p14*) 以外还要进行别名的历史扩展。

`history-expand-line` (M-^) 对当前行进行历史扩展。

`magic-space` 对当前行进行历史扩展并插入一全空格 (参见§9[历史的交互使用], *p90*)。

`alias-expand-line` 对当前行进行别名扩展 (参见§6.6[别名], *p62*)。

`history-and-alias-expand-line` 对当前行进行历史和别名扩展。

`insert-last-argument` (M-. 或 M-_) 和 `yank-last-arg` 是同义的。

`operate-and-get-next` (C-o) 提交并执行当前行，然后从历史中取出相对于当前行的下一行进行编辑。忽略任何参数。

`edit-and-execute-command` (C-x C-e) 启动一个编辑器来编辑当前行，并把结果当作 shell 命令来执行。Bash 会试图依次启动 `$VISUAL`，`$EDITOR` 和 emacs 作为编辑器。

§8.5 Readline 的 vi 模式

尽管 Readline 库里面没有完整的 vi 编辑命令，却已经包含了足够的命令进行简单的行编辑。Readline 的 vi 模式在行为上遵循 POSIX 1003.2 标准。可以使用 `set -o emacs` 和 `set -o vi` 命令 (参见§86[内部命令 set], *p43*) 在 `emacs` 和 `vi` 模式之间交互的切换。默认的是 `emacs` 模式。

在 `vi` 模式下输入文本行时已经进入了“插入”模式，就好像已经输入了 i 一样。按下 ESC 会切换到“命令”模式，这时可以用标准的 vi 移动键来编辑文本，用“k”移动到上一个历史行，用“j”移动到下一行，等等。

§8.6 可编程的补全

如果已经使用内部命令 `complete` (参见§8.7[可编程补全的内部命令], *p87*) 定义了补全的方法 (即 `compspec`)，则在对命令中的参数进行单词补全时，就会启用可编程的补全功能。

首先识别命令名。如果已经为这个命令定义了 `compspec` 补全方法，就用 `compspec` 来生成一个可以补全当前单词的条目列表。如果命令词是一个完整路径，则首先用 `compspec` 生成一个完整路径的补全列表；如果没有指定生成完整路径的 `compspec`，就会试图找到一个能生成路径中最后一个斜杠后面的部分路径的

补全列表。找到 compspec 以后，就用它来生成和当前单词匹配的条目列表。如果没有找到，Bash 就生成默认的补全列表 (参见§8.4.6[补全命令], *p84*)。

首先[8]，执行 compspec 指定的动作。这时只返回以待补全单词作为前缀的条目。如果在补全文件或目录名时指定了“-f”或“-d”选项，就会使用 shell 变量 FIGNORE (参见§5.2[Bash 的变量], *p51*) 来过滤匹配的条目。

其次，扩展“-G”选项的文件名模式并生成补全列表。根据模式生成的单词不一定要和待补全的单词匹配。这时不会使用 shell 变量 GLOBIGNORE 来过滤匹配的条目，而使用 FIGNORE 过滤。

然后，处理“-W”选项指定的字符串参数。首先用特殊变量 IFS 中的字符作为分隔符把这个字符串分开。这时会使用 shell 的引用机制。然后对每个单词进行在括号扩展、波浪号扩展、参数和变量扩展、命令替换、以及算术扩展 (参见§3.5[Shell 扩展], *p14*)，并对扩展结果进行单词拆分 (参见§3.5.7[单词拆分], *p19*)。扩展的结果加上待补全的单词作为前缀；这样匹配的单词就成为可以补全的条目。

生成匹配的条目以后，就执行“-F”和“-C”选项指定的任何 shell 函数和命令。执行这些函数或命令时，会对 COMP_LINE、COMP_POINT、COMP_KEY、和 COMP_TYPE 变量进行赋值 (参见§5.2[Bash 的变量], *p50*)。如果执行的是 shell 函数，则还会设置 COMP_WORDS 和 COMP_CWORD 变量。执行函数或命令时，第一个参数是等待补全其参数的命令的名称，第二个参数是要补全的单词，第三个参数是在当前命令中待补全单词的前一个单词。这时不会按照待补全的单词对生成的补全条目进行过滤；而所执行的函数或命令可以完全自由的控制生成的条目。“-F”指定的函数将行执行。这个函数可以使用任何 shell 功能来生成匹配条目，包括下面的内部命令 compgen 和 compopt (参见§8.7[可编程补全的内部命令], *p87*)，但是它必须把补全条目放在数组变量 COMPREPLY 中。

接下来，在类似于命令替换的环境中执行“-C”选项指定的命令。这个命令应该在标准输出中打印出匹配条目，每个一行。如有必要，可以使用反斜杠来转义换行符。

当所有补全条目都生成好以后，“-X”选项指定的过滤器过滤条目列表。这个过滤器是一个模式，就和文件名扩展的模式一样。模式中的“&”将替换成待补全的文本；而“&”本身可以用反斜杠转义，在匹配之前反斜杠会被去除。与这个模式匹配的条目将从列表中删除。模式中前导的“!”表示否定，这种情况下会删除所有不与模式匹配的条目。

最后，对补全列表的每个条目加上“-P”和“-S”选项指定的前缀和后缀，并把结果作为最终补全列表返回给 Readline 的补全代码。

如果上一次执行的动作没有生成任何条目，并且在定义 compspec 时指定了“-o dirnames”，则将试图进行目录名称补全。如果定义 compspec 时指定了“-o plusdirs”选项，则会试图进行目录名称补全，并加上其它动作生成的任何补全条目。

默认情况下，如果找到一个 compspec，则不管它生成什么，都将完整的返回给补全代码；这时不会进行 Bash 默认的补全，也不会进行 Readline 默认的文件名补全。在定义 compspec 时，可以使用“-o bashdefault”选项，这样如果 compspec 没有生成任何条目就会试图进行 Bash 默认的补全。在定义 compspec 时，可以使用“-o default”选项，这样如果 compspec 没有生成任何条目 (并且如果试图进行 Bash 默认的补全也没有生成任何条目) 就会试图进行 Readline 默认的补全。

如果 compspec 指示进行目录名称补全，则可编程的补全功能会强制 Readline 在每个指向目录的符号链接后面加上斜杠 (取决于 Readline 的 mark-directories 变量)，而不管 Readline 的 mark-symlinked-directories 变量如何设置。

§8.7 可编程补全的内部命令

可编程补全的功能是由两个命令实现的。此外，还可以特殊的设置补全的选项。

A. compgen

```
compgen [选项] [单词]
```

根据选项生成与单词相匹配的补全，并写到标准输出中；这些选项可以是内部命令 complete 所能接受的任何选项，但不能是“-p”和“-r”。如果使用了“-F”或“-C”选项，则由可编程补全功能设置的各个 shell 变量虽

[8] 原文的章节安排有点乱，这里做了适当改动。

然仍可以使用，它们的值却没有什么作用。生成的补全条目就好像可编程补全的代码用相同的选项按照补全方法直接生成的一样。如果指定了单词，则只显示匹配该单词的条目。返回状态是真，除非指定了无效的选项，或没有生成任何匹配条目。

B. complete

```
complete [-abcdefgjksuv] [-o 补全选项] [-E] [-A 动作] [-G 模式] [-W 单词列表]
[-F 函数] [-C 命令] [-X 过滤模式] [-P 前缀] [-S 后缀] 名称 [名称 ···]

complete -pr [-E] [名称 ···]
```

指定如何对每个名称进行补全。如果指定了“-p”选项，或者没有指定任何选项，则把已有的补全方法用一种便于重新作为输入的格式打印出来。“-r”选项会把每个名称的补全方法删除。“-E”选项告诉后续选项和动作要补全“空”命令，即补全空白行。对单词进行补全时处理补全方法的过程已经在上文中介绍 (参见§8.6[可编程的补全], *p86*)。

如果指定了其它选项，则会有如下的含义。“-G”、“-W”、和“-X”选项 (如有必要，还有“-P”和“-S”选项) 的参数需要引用，以防止它们在补全开始前被扩展。

-o 补全选项 除了简单的生成补全条目以外，补全选项还控制着 compspec 的多方面行为。补全选项可以是：

- **bashdefault** 如果 compspec 没有生成任何条目就进行 Bash 默认的其它补全。
- **default** 如果 compspec 没有生成任何条目就使用 Readline 默认的文件名补全。
- **dirnames** 如果 compspec 没有生成任何条目就进行目录名称补全。
- **filenames** 告诉 Readline 由 compspec 生成文件名，以便进行与文件名相关的处理 (例如在目录名后面加上斜杠，引用特殊字符，或去掉行尾的空格)。这个选项是为和“-F”选项所指定的函数一起使用而设计的。
- **nospace** 告诉 Readline 不要在结尾补全的单词后添加空格。
- **plusdirs** 生成 compspec 定义的所有匹配条目后，还试图进行目录名补全，并把生成的条目加入到其它动作得到的结果中。

-A 动作 动作可以是下列之一；它用来生成补全条目。

- **alias** 所有别名。还可以指定为“-a”选项。
- **arrayvar** 所有数组变量名。
- **binding** Readline 的所有键绑定名 (参见§8.4[可以绑定的 Readline 命令], *p80*)。
- **builtin** 所有的 shell 内部命令名。还可以指定为“-b”选项。
- **command** 所有的命令名。还可以指定为“-c”选项。
- **directory** 所有的目录名。还可以指定为“-d”选项。
- **disabled** 所有已经禁用的 shell 内部命令。
- **enabled** 所有已经启用的 shell 内部命令。
- **export** Shell 导出的所有变量名。还可以指定为“-e”选项。

`file` 所有的文件名。还可以指定为“`-f`”选项。

`function` 所有的 shell 函数名。

`group` 所有的用户组名。还可以指定为“`-g`”选项。

`helptopic` 内部命令 `help` (参见§4.2[Bash 的内部命令 `help`], *p38*) 所接受的所有帮助主题。

`hostname` Shell 变量 `HOSTFILE` (参见§5.2[Bash 的变量], *p52*) 指定文件中的所有主机名。

`job` 如果使用了作业控制，则是所有作业的名称。还可以指定为“`-j`”选项。

`keyword` Shell 中的所有保留字。还可以指定为“`-k`”选项。

`running` 如果使用了作业控制，则是所有正在运行的作业。

`service` 所有的服务名称。还可以指定为“`-s`”选项。

`setopt` 内部命令 `set` 的“`-o`”选项 (参见§4.3.1[内部命令 set], *p42*) 所接受的所有有效参数。

`shopt` 内部命令 `shopt` (参见§4.3.2[内部命令 shopt], *p45*) 所接受的所有 shell 选项名称。

`signal` 所有信号名称。

`stopped` 如果使用了作业控制，则是所有已停止的作业名称。

`user` 所有用户名。还可以指定为“`-u`”选项。

`variable` 所有 shell 变量的名称。还可以指定为“`-v`”选项。

`-G 模式` 对模式进行文件名扩展来生成补全条目。

`-W 单词列表` 使用特殊变量 `IFS` 中的字符拆分单词列表并扩展拆分后的每个单词。结果中与待补全单词匹配的条目就是补全条目。

`-C 命令` 在子 shell 中执行命令，并把其结果作为补全条目。

`-F 函数` 在当前的 shell 环境中执行 shell 函数。结束执行时，从数组变量 `COMPREPLY` 中获取补全条目。

`-X 过滤模式` 它是进行文件名扩展时使用的模式。它作用于通过前面的选项和参数生成的补全列表，并把每个与过滤模式匹配的条目删除。模式中前导的“`!`”表示否定；这时会删除与过滤模式不匹配的条目。

`-P 前缀` 在处理完所有其它的选项后，给每个补全的条目加上前缀。

`-S 后缀` 在处理完所有其它的选项后，给每个补全的条目加上后缀。

返回状态是真，除非指定了无效的选项、或者指定了“`-p`”或“`-r`”之外的选项却没有指定名称参数、或者修改了名称中没有任何补全方法定义过的一个选项、或者输出时发生错误。

C. `compopt`

```
compopt [-o 选项] [+o 选项] [名称]
```

修改每个名称指定的补全选项；如果没有指定名称则修改当前执行的补全的选项。如果也没有指定选项，则显示每个名称或当前补全所用的选项。选项可能的取值就是上面的内部命令 `complete` 的有效选项。返回状态是真，除非指定了无效的选项、或者修改了名称中没有任何补全方法定义过的一个选项、或者输出时发生错误。

第九章　历史的交互使用

本章从用户的角度介绍了如何使用 GNU 的历史库功能。可以把这里的内容作为用户指南。关于如何在其它程序中使用 GNU 的历史库功能，请参考《GNU Readline 库参考手册》

§9.1　Bash 的历史功能

如果使用了内部命令 set 的“-o history”选项 (参见§86[内部命令 set], *p43*)，shell 就会允许访问历史命令，即以前输入的命令。Shell 变量 HISTSIZE 控制历史中所保存命令的数目，即只保存最后 $HISTSIZE 条 (默认是 *500*) 命令的文本。根据 shell 变量 HISTIGNORE 和 HISTCONTROL 的值，shell 会在进行参数和变量扩展之前及历史扩展之后把每条命令保存在历史中。

Shell 在启动时会用 HISTFILE 变量指定的文件 (默认是 ~/.bash_history) 对历史进行初始化。如果有必要，就截断 HISTFILE 变量指定的文件，使它包含的行数不超过 HISTFILESIZE 变量的值。在交互式的 shell 退出运行时，把历史中最后 HISTSIZE 行复制到 HISTFILE 变量指定的文件中。如果设置了 shell 选项 histappend (参见§87[内部命令 shopt], *p46*)，则把这些行附加到历史文件中，否则就覆盖历史文件。如果重置了 HISTFILE，或者历史文件不可写，则不保存历史。保存了历史以后，就把历史文件截断到不超过 $HISTFILESIZE 行。如果没有设置 HISTFILESIZE 则不截断历史文件。

如果设置了 HISTTIMEFORMAT，则把与每条历史关联的时间戳信息也一并写入到历史文件中，时间信息用历史注释字符标志。在读取历史文件时，以历史注释字符和紧跟其后的数字开头的行会被当作下一条命令[1]的时间戳。

可以用内部命令 fc 来列出、编辑和重新执行部分历史；也可以用内部命令 history 显示和修改历史，或者操纵历史文件。在进行命令行编辑时，每种编辑模式中都有搜索命令 (参见§B[history], *p91*) 来访问历史。

Shell 可以控制在历史文件中保存哪些命令。HISTCONTROL 和 HISTIGNORE 变量只保存输入命令中的一部分条目。如果设置了 shell 选项 cmdhist，shell 就会试图在同一条目中保存多行命令的每一行，必要时可能会加上分号以保证语法的正确性。Shell 选项 lithist 会使 shell 用行内的换行符，而不是分号。可以用 shopt 来设置这些选项。参见§4.3.2[内部命令 shopt], *p45* 的介绍。

§9.2　Bash 历史内部命令

Bash 提供了两个命令来操纵历史和历史文件。

A. fc

```
fc [-e 编辑器] [-lnr] [第一个] [最后一个]
fc -s [模式=替换文本] [命令]
```

[1] 原文说是“上一条”；而实际上，时间信息在命令之前，所以应该是“下一条”。

改变命令。第一种形式将从历史文件中选取从第一个到最后一个之间的命令。第一个和最后一个都可以是字符串 (用来指定最近用这些字符开头的命令) 或数字 (历史中的位置索引；负数表示从当前命令开始索引)。如果没有指定最后一个，它就和第一个相同；如果没有指定第一个，则在编辑时它就是前一个命令，在列表时就是 -16。如果指定了“-l”选项助记词：List，列表，就在标准输出中列出这些命令；“-n”选项助记词：Number，行号可以在列出命令时不显示行号；而“-r”选项助记词：Reverse，倒序使列表倒序排列。否则[2]，启动编辑器并打开包含这些命令的文件。如果没有指定编辑器就使用变量 ${FCEDIT:-${EDITOR:-vi}} 扩展后的值；也就是说，如果设置了 FCEDIT 变量就使用它，或者如果设置了 EDITOR 变量就使用它，如果都没有设置就用 vi。编辑结束以后则显示并执行编辑过的命令。

在第二种形式中，把每个选中的命令中与模式匹配的文本改成替换文本后执行。

fc 命令很有用的别名是 r='fc -s'，这样输入“r cc”就会执行最后一个以 cc 开头的命令，而输入“r”就会执行最后一个命令。参见§6.6[别名], p62。

B. history

```
history [n]
history -c
history -d 偏移量
history [-anrw] [文件名]
history -ps 参数
```

如果没有选项，列出历史和行号。前面带有“*”的行已经被修改过。参数 n 只列出最后 n 行。如果设置了 shell 变量 HISTTIMEFORMAT 且不为空，就用它作为 strftime 的参数来显示每条历史所关联的时间戳。在格式化的时间戳和历史行之间没有空白[3]。

如果指定了选项，就会有下面的含义：

-c 清除历史。可以把它和其它选项一起使用来完全替换历史。助记词：Clear，清除

-d 偏移量 删除偏移量处的历史行。偏移量应该是列出历史时显示的数值。助记词：Delete，删除

-a 把新的历史行 (即当前的 Bash 会话开始后输入的历史行) 附加到历史文件中。助记词：Append，附加

-n 把历史中尚未读取的行附加到当前的历史中。这些行是当前的 Bash 会话开始后附加到历史文件中去的。助记词：New，新行

-r 读取历史文件，把其内容附加到当前的历史中。助记词：Read，读取

-w 把当前的历史写入到历史文件中。助记词：Write，写入

-p 对参数进行历史扩展并在标准输出上显示其结果，而不是把结果存放在历史中。助记词：outPut，输出

-s 把参数作为单个条目附加到历史中。助记词：Single，单个

如果使用了“-w”、“-r”、“-a”、“-n”选项中中的任意一个并且给定了文件名，就把文件名当作历史文件。如果没有给定文件名，则使用 “HISTFILE”变量的值。

[2] 即指定了“-l”选项时。

[3] 所以如果想把时间戳和历史行之间用空白分隔，必须在 HISTTIMEFORMAT 的后面显式指定这些空白。

§9.3 历史扩展

历史库提供了类似于 csh 中的历史扩展功能。本节介绍操纵历史信息的语法。

历史扩展把历史中的单词引入到输入流中，这样易于重复输入命令、在当前输入行中插入以前命令的参数、或者快速修改以前命令中的错误。

历史扩展有两个步骤：第一步决定在替换时应该使用历史中的哪一行，第二步选择选定行的部分文本以包含到当前行中。从历史中选定的行叫做“条目”；该行中要操纵的文本部分叫“单词”，可以使用各种修饰符来控制选中的单词。与 Bash 一样，选中的行被拆分成单词；被引用的多个单词当作一个单词。历史扩展由历史扩展字符 (默认是“!”) 引入。只有“\”和“'”可以对历史扩展字符转义。

有些可以用内部命令 shopt (参见§4.3.2[内部命令 shopt], *p45*) 设置的 shell 选项用来调整历史扩展的行为。如果设置了 shell 选项 histverify，并且使用了 Readline，则历史扩展不会立即传给 shell 解释器，而是把扩展后的命令行重新加载到 Readline 的编辑缓存中以备进一步修改。如果使用了 Readline 并且设置了 shell 选项 histreedit，则历史扩展失败时将重新加载到 Readline 的编辑缓存中以备更正。在进行历史扩展之间，可以用内部命令 history 的“-p”选项来查看历史扩展如何进行。而内部命令 history 的“-s”选项可以用来把命令直接加入到历史文件中却并不择午，这样它就可以在以后使用。这如果和 Readline 一起使用将会非常有用。

Shell 可以通过 histchars 变量控制历史扩展机制所使用的各种字符 (参见§5.2[Bash 的变量], *p51*)，还可以在写入历史文件时用历史注释字符来标志历史中的时间戳。

§9.3.1 条目指示符

条目指示符指向历史中的命令行。

`!` 开始历史替换，除非后面跟着空格、制表符、行结束符、“=”、或“(”(如果用内部命令 shopt 打开了 extglob 选项)。

`!n` 选择命令行 n。

`!-n` 选择向后第 n 行命令。

`!!` 选择前一条命令，它和“!-1”是等价的。

`!字符串` 选择最近以字符串开头的命令。

`!?字符串[?]` 选择最近包含字符串开头的命令。如果字符串后面紧跟着换行符就可以省略结尾的“?”。

`^字符串一^字符串二^` 快速替换。重复最后的命令，并把字符串一替换成字符串二；它和 !!:s/字符串一/字符串二 是等价的。

`!#` 目前已经输入的整个命令。

§9.3.2 单词指示符

单词指示符用来从选定条目中选择指定的单词。条目指示符和单词指示符之间用“:”分隔；如果单词指示符以“^”、“$”、“*”、“-”、“%”开头，则可以省略分隔符。单词从行首开始数起，第一单词序号为 *0*。插入到当前行中时，这些单词用空格分开。例如，

`!!` 指定前一条命令。如果输入这个指示符则整个重复前一条命令。

`!!:$` 指定前一条命令的最后一个参数；可以简写为 !$。

`!fi:2` 指定最近以字母 fi 开头的命令的第二个参数。

下面是单词指示符：

`0` 即零，第零个单词。对大多数命令而言，它是指命令名。

`n` 第 n 个单词。

`^` 第一个参数 (单词)。

`$` 最后一个参数。

`%` 最近“?字符串?”匹配的单词。

`x-y` 单词范围。“0-y”可以简写为“-y”。

`*` 除了第零个以外的所有单词，和“1-$”同义。如果条目中只有一个单词，使用“*”也不会出错，而是返回空字符串。

`x*` “x-$”的简写形式。

`x-` 和“x*”一样，是“x-$”的简写形式，但是忽略最后一个单词。

如果使用单词指示符时没有用条目指示符，则把前一条命令作为条目。

§9.3.3 修饰符

在可选的单词指示符后面，可以加上下列一个或多个修饰符，每个修饰符前都有“:”。

`h` 去掉文件名的尾部，只保留头部。

`t` 去掉文件名的头部，只保留尾头部。

`r` 去掉结尾的扩展名，只保留文件基名。

`e` 去掉扩展名以外的所有部分。

`p` 打印新的命令但不执行。

`q` 引用替换后的单词，以备进一步替换。

`x` 和“q”一样引用替换后的单词，同时还在空格、制表符、换行符的地方把单词分开。

`s/旧词/新词/` 把条目中的第一个旧词替换成新词。在“/”的地方可以使用任何分隔符。在旧词和新词中要用到分隔符的地方可以用一个反斜杠对分隔符转义。如果新词中出现“&”，就替换成旧词；可以使用一个反斜杠来引用“&”。结尾的分隔符如果是输入行的最后一个字符则是可选的。

`&` 重复上次替换。

`g`

`a` 使替换在整个条目中进行，和“s”一起使用，例如 `gs/old/new/`，或者和“&”一起使用。

`G` 对条目中的每个单词都执行一次“s”修饰符。

第十章 Bash 的安装

本节提供了在 Bash 支持的不同系统上的基本安装指导。本版本支持 GNU 操作系统，几乎每个 UNIX 版本，以及几个非 UNIX 系统，例如 BeOS 和 Interix。还有针对 MS-DOS、OS/2、Windows 等系统的独立移植版本。

§ 10.1 基本安装

下面介绍 Bash 的安装步骤。编译 Bash 最简单的方法是：

1. 切换到包含源文件的目录并输入 `./configure` 以便在系统中配置 Bash。如果在老版本的 System V 上使用 `csh`，则需要输入 `sh ./configure` 以防止 `csh` 自己去执行配置。配置要花一点时间。运行中它会输出一些信息告诉用户它正在检查什么功能。
2. 输入 `make` 来编译 Bash 和 `bashbug` — 一个错误报告的脚本。
3. 可选的，输入 `make tests` 来运行 Bash 的测试用例。
4. 输入 `make install` 来安装 `bash` 和 `bashbug`。这一步还会安装帮助手册和 Info 文件[1]。

配置脚本会试图支猜测编译时所需的各个和系统相关的变量值。这些值用来在包的每个目录 (即顶级目录、`builtins`、`doc`、`support`，还有 `lib` 下面的每个目录，以及其它几个目录) 下面生成 `Makefile` 文件。它还会生成一个包含系统相关的定义 `config.h` 文件。最后，它会生成一个叫 `config.status` 的脚本，以后可以用它来重新生成当前的配置；还有一个叫 `config.log` 的文件来存放编译时的输出 (主要用于调度配置脚本)。有时候 `config.cache` 文件包含一些不想要的结果；这时可以删除或修改它。在 Bash 目录的命令提示符下，可以输入下面的命令以获得配置脚本所接受的更多选项和参数：

```
bash$ ./configure --help
```

如果想在编译 Bash 时做一些额外配置，则需要了解配置脚本是怎么决定是否这样配置的，并把代码不同部分以及步骤发送到 bash-maintainers@gnu.org 以便在下次发布时包含这些改动。

`configure.in` 文件是一个叫 Autoconf 的程序用来生成配置脚本的。如果修改配置，或者使用新版本的 Autoconf 重新生成配置，只需要这个文件就行了。这时要确保使用 *2.50* 或更高版本的 Autoconf。

可以用 `make clean` 在源文件目录删除二进制程序文件和目标文件。如果还要删除配置脚本生成的文件 (以便为不同的电脑编译 Bash)，则输入 `make distclean`。

[1] 指可以用 `info` 命令或者 emacs 的 C-h-i 命令查看的 `texinfo` 文件。

§10.2 编译器和选项

有些系统需要在编译和连接时用一些不能被配置配置脚本识别的选项。可以在环境中为配置脚本设置一些变量的初始值。如果使用与波恩 shell 兼容的程序，则可以在命令行中这样写：

```
CC=c89 CFLAGS=-O2 LIBS=-lposix ./configure
```

如果系统中有 env 程序，则可以这样写：

```
env CPPFLAGS=-I/usr/local/include LDFLAGS=-s ./configure
```

如果有 GCC，则配置过程中变用它来编译 Bash。

§10.3 跨平台编译

可以把目标文件各个系统自己的目录中以便同时为多种类型的电脑编译 Bash。为此，必须使用支持 VPATH 变量的 make 版本，例如 GNU make。切换到要存放目标文件和可执行文件的目录并运行源文件目录中的配置脚本。这时可能要指定 --srcdir=PATH 参数告诉配置脚本源文件的位置。配置脚本会自动在其所在的目录和 .. 中寻找源文件。

如果一定要使用不支持 VPATH 变量的 make，则可以在源文件目录中每次只为一个系统编译 Bash。为某个系统安装好 Bash 后要运行 make distclean 才能再为其它系统编译。或者，如果系统支持符号链接，则可以使用 support/mkclone 脚本来生成编译树，其中包含指向源目录每个文件的链接。下面的例子在当前目录中使用 /usr/gnu/src/bash 目录里的源文件来生成编译树：

```
bash /usr/gnu/src/bash/support/mkclone -s\
    /usr/gnu/src/bash .
```

mkclone 脚本要 Bash 才能运行，所以必须已经在至少一个系统上编译了 Bash 才能为其它系统生成编译树。

§10.4 安装路径

默认情况下，make install 会在 /usr/local/bin、/usr/local/man 等目录下安装。可以使用配置选项 --prefix=PATH 来指定 /usr/local 以外的安装根目录，或者在运行 make install 时给 make 的 DESTDIR 变量指定不同的值。

可以为系统相关和系统无关的文件分别指定安装根目录。如果指定了配置选项 --exec-prefix=PATH，则 make install 会使用 PATH 作为安装程序和库文件的根目录；而文档和其它数据文件仍安装在常规的地方。

§10.5 选择系统类型

可能配置脚本不能自动发现某些系统特性，而是由要运行 Bash 的主机类型来决定。配置脚本通常会发现这些特性；如果它没有并打印一条信息说它不能发现系统类型，则需要 --host=类型 选项。类型可以是系统类型的简称，例如 sun4，或者是包含三个字段的典型称呼“CPU-COMPANY-SYSTEM”，如 i386-unknow-freebsd4.2。关于每个字段的可能取值，请参考 support/config.sub 文件。

§10.6 默认设置的共享

如果要让配置共享一些默认值，可以创建一个 config.site 的全局 shell 脚本，并在其中设置诸如 CC、cache_file、prefix 等变量的默认值。配置脚本会依次查找 PREFIX/share/config.site 和 PREFIX/etc/config.site 文件。也可以有环境变量 CONFIG_SITE 来指定全局脚本的位置。注意，Bash 的配置脚本会寻找全局脚本，但不是每个其它配置脚本也都这么做。

§10.7 控制配置脚本

配置脚本会识别下列控制选项。

`--cache-file=文件` 用文件而不是 `./config.cache` 来保存测试结果。为了测试配置脚本，可以把它设为 `/dev/null` 以禁止缓存。

`--help` 打印配置选项的总结后退出。

`--quiet`

`--silent` 不显示正在检查的目标。

`-q`

`--srcdir=目录` 在目录中寻找 Bash 的源文件。配置脚本通常会自动找到这个目录。

`--version` 打印用来生成当前配置脚本的 Autoconf 的版本信息，然后退出。

还有其它不常使用的以及用作模板的选项。可以用 `./configure --help` 列出全部选项。

§10.8 配置选项

Bash 的配置脚本有一些 `--enable-feature` 选项，其中的 `feature` 是 Bash 中可选的功能。还有一些 `--with-package` 选项，其中的 `package` 是诸如 `bash-malloc` 或 `purify` 的模块。如果想禁用一些默认的模块，可以用 `--without-package`；如果想禁用一些默认就打开的功能，可以用 `--disable-feature`。下面是 Bash 配置脚本能识别的所有 `--enable-` 和 `--with-` 选项。

`--with-afs` 如果使用 Transarc 上面的安德鲁文件系统 (AFS) 就打开这个选项。

`--with-bash-malloc` 使用 `lib/malloc` 目录下 Bash 版本的 `malloc`[2]。它和 GNU LIBC 里面的 `malloc` 是不同的，而是直接继承处 BSD *4.2* 的一个老版本。这个版本更快，但每次分配内在时都会浪费一些空间。这个选项默认是打开的。`NOTES` 文件列出了一些应该关闭这个选项的系统；在某些系统中，配置脚本会自动关闭这个选项。

`--with-curses` 使用 `curses` 库而不是 `termcap` 库。如果系统中没有合适的或完整的 `termcap` 数据库，就应该使用它。

`--with-gnu-malloc` 与 `--with-bash-malloc` 同义。

`--with-installed-readline[=前缀]` 定义这个选项会让 Bash 和本地安装的而不时 `lib/readline` 中的 Readline 库连接。这只有使用 Readline *5.0* 或更高版本时才有用。如果没有指定 `PREFIX`，并且 Readline 没有安装在标准的头文件和库目录中，配置脚本会使用 `make` 的变量 `includedir` 和 `libdir` 作为默认的子目录和根目录来搜索 Readline 的安装路径。如果 `PREFIX` 是 `no`，Bash 就和 `lib/readline` 中的那个版本连接。如果 `PREFIX` 设为其它值，则配置脚本会把它当作一个路径，并在这个目录和其子目录中搜索安装好的 Readline (在 `PREFIX/include` 中搜索头文件，在 `PREFIX/lib` 中搜索库文件)。

`--with-purify` 定义这个选项可以使用 Rational 软件中的来进行内在分配检查。

[2] 用来给进程分配内在空间的函数。

`--enable-minimal-config` 生成一个最简小的 shell，和历史上的波恩 shell 很相近。

有一些 `--enable-` 选项能控制怎么编译和连接 Bash，而不是改变它运行时的功能。

`--enable-largefile` 支持大文件；如果操作系统需要特殊的编译器选项才能生成支持大文件在程序就用这个选项。如果操作系统支持大文件，这个选项默认是打开的。

`--enable-profiling` 这个选项会生成一个支持性能分析的 Bash 可执行文件；性能分析信息可以在每次执行时由 `gprof` 处理。

`--enable-static-link` 如果使用 `gcc`， 这个选项可以静态的连接 Bash；它可以作为 `root` 的 shell。

可以用 `minimal-config` 即最小选项来禁用下面所有的选项；这个选项将会优先处理，所以可以用 `--enable-feature` 来打开个别选项。除了 `disabled-builtins` 和 `xpg-echo-default`，下面的所有选项默认都是打开的，除非系统不支持。

`--enable-alias` 允许扩展并包含内部命令 `alias` 和 `unalias` (参见§6.6[别名], *p62*)

`--enable-arith-for-command` 支持另外一种形式的 `for` 命令 (参见§3.2.4.1[循环结构 `for`], *p8*)，它看上去就像 C 语言里面的 `for` 语句。

`--enable-array-variables` 支持一维数组变量 (参见§6.7[数组], *p62*)。

`--enable-bang-history` 支持类似于 `csh` 的历史替换 (参见§9[历史的交互使用], *p90*)。

`--enable-brace-expansion` 支持类似于 `csh` 的大括号扩展，例如 `b{a,b}c` ➜ `bac bbc` (参见§3.5.1[大括号扩展], *p15*)。

`--enable-casemod-attributes` 允许内部命令 `declare` 以及在赋值时更改大小写的属性。例如，具有大写属性的变量在接受赋值时会把值转换成大写。

`--enable-casemod-expansion` 支持更改大小写的单词扩展。

`--enable-command-timing` 支持把 `time` 识别为关键字并显示其后的管道的时间统计信息 (参见§3.2.2[管道], *p7*)。这样可以统计管道以及内部命令和函数的执行时间。

`--enable-cond-command` 支持条件测试命令 `[[` (参见§3.2.4.2[条件结构 `[[···]]`], *p10*)

`--enable-cond-regexp` 支持在条件测试命令 `[[` 的双目运算符 `=~` 后使用 POSIX 正则表达式进行匹配 (参见§3.2.4.2[条件结构 `[[···]]`], *p10*)。

`--enable-coprocesses` 支持协同进程和保留字 `coproc` (参见§3.2.5[协同进程], *p12*)。

`--enable-debugger` 支持 Bash 调度器 (另外发布)。

`--enable-directory-stack` 支持类似于 `csh` 的目录栈以及内部命令 `pushd`、`popd`、`dirs` (参见§6.8[目录栈], *p63*)。

`--enable-disabled-builtins` 支持通过 `builtin XXX` 来启用一个内部命令，即使这个内部命令已经用 `enable -n XXX` 禁用了。参见§4.2[Bash 的内部命令 `enable`], *p37*和§4.2[Bash 的内部命令 `builtin`], *p35* 。

`--enable-dparen-arithmetic` 支持 `((···))` 命令 (参见§3.2.4.2[条件结构 `((···))`], *p10*)。

`--enable-extended-glob` 支持扩展的模式匹配 (参见§3.5.8.1[模式匹配], *p20*)。

`--enable-help-builtin` 支持内部命令 `help`，它能显示 shell 内部命令和变量的帮助信息 (参见§4.2[Bash 的内部命令 `help`], *p38*)。

`--enable-history` 支持命令历史和内部命令 `fc` 与 `history` (参见§9.1[Bash 的历史功能], *p90*)。

`--enable-job-control` 如果系统支持，就启用作业控制功能 (参见§7[作业控制], *p69*)。

`--enable-multibyte` 如果系统提供必要支持，就启用多字节字符。

`--enable-net-redirections` 在重定向中启用对 `/dev/tcp/host/port` 和 `/dev/udp/host/port` 文件名的特殊处理 (参见§3.6[重定向], *p21*)。

`--enable-process-substitution` 如果系统提供必要支持，就启用进程替换 (参见§3.5.6[进程替换], *p19*)。

`--enable-progcomp` 启用可编程的补全功能 (参见§8.6[可编程的补全], *p86*)。如果没有启用 Readline，这个选项将不起作用。

`--enable-prompt-string-decoding` 启用 `$PS1`、`$PS2`、`$PS3`、`$PS4` 提示符字符串中对一些转义字符的解析。关于提示符字符串中的全部转义字符，请参见§6.9[提示符的控制], *p64*。

`--enable-readline` 通过 Bash 版本的 Readline 库支持命令行编辑和历史 (参见§8[编辑命令行], *p72*)。

`--enable-restricted` 支持受限制的 shell。如果启用它，则用 `rbash` 来启动 Bash 时会进入受限模式。关于受限模式，请参见§6.10[受限制的 shell], *p66*。

`--enable-select` 支持内部命令 `select`，它可以生成简单的菜单 (参见§3.2.4.2[条件结构 `select`], *p10*)。

`--enable-separate-helpfiles` 把内部命令 `help` 显示的帮助文档存放在外部文件中，而不是存放在命令内部。

`--enable-single-help-strings` 把内部命令 `help` 显示的帮助文档作为各个帮助主题的单个字符串存放。这样有助于把这些文本翻译成不同语言。如果电脑不能处理很长的文本字符串，可以需要禁用这个选项。

`--enable-strict-posix-default` 使 Bash 默认就遵循 POSIX (参见§6.11[Bash 的 POSIX 模式], *p67*)。

`--enable-usg-echo-default` 与 `--enable-xpg-echo-default` 同义。

`--enable-xpg-echo-default` 让内部命令 `echo` 默认就扩展由斜杠默认的字符，而不需要“`-e`”选项。这个选项会把 shell `xpg_echo` 选项的默认值设为 `on`，从而使得 Bash 的 `echo` 表现得更像是 UNIX 统一规范第三版中指定的那个版本。关于 `echo` 所能识别的转义序列，请参见§4.2[Bash 的内部命令 `echo`], *p37*。

`config-top.h` 文件中包含了那些不能由配置脚本设置的选项，它们是 C 预处理的 `#define` 声明。这些选项中有些是不应该改的；如要修改，请当心其后果。关于这些定义的作用，请阅读与其相关的注释。

附录

附录A　Bash 语法一览表

下面是本书点中介绍的所有命令的语法格式。格式后面的数字表示其所在的页码。

[time [-p]] [!] 命令一 [[|或|&] 命令二 ···]
p7

命令一 && 命令二
p7

命令一 || 命令二
p7

until 测试命令; do 命令块; done
p8

while 测试命令; do 命令块; done
p8

for 变量 [in 单词]; do 命令块; done
p8

for ((表达式一 ; 表达式二 ; 表达式三)); do 命令块 ; done
p8

```
if 测试命令一 ; then
   命令块一 ;
[elif 测试命令二 ; then
   命令块二 ;]
   ⋮
[else
   其它命令块 ;]
fi
```
p9

```
case 单词 in
  [ [(] 模式一 [| 模式二 ] ··· )
     命令块
     ;;]
  ⋮
esac
```
p9

```
select 名称 [in 单词表 ··· ] ; do 命令块; done
```
p10

```
(( 算术表达式 ))
```
p10

```
let "表达式"
```
p10

```
[[ 条件表达式 ]]
```
p10

```
( 表达式 )
```
p11

```
{ 表达式; }
```
p11

```
coproc [ NAME ] 命令 [ 重定向 ]
```
p12

```
[ function ] 名称 () 复合命令块 [ 重定向 ]
```
p12

```
名称 =[值]
```
p13

```
{x ..  y [ 增量 ] }
```
p15

```
${参数}
```
p16

```
$( 命令 )
```
p18

`命令`

p18

$((表达式))

p18

<(命令列表)

p19

>(命令列表)

p19

ls > 目录列表 2>&1

p21

ls 2>&1 > 目录列表

p21

[n]<单词

p21

[n]>[|]单词

p22

[n]>>单词

p22

&>单词

p22

>&单词

p22

>单词 2>&1

p22

&>>单词

p22

>>单词 2>&1

p22

eval [参数表]
p30

exec [-cl] [-a 名称] [命令 [参数表]]
p30

exit [n]
p30

export [-fn] [-p] [名称[=值]]
p30

getopts 选项字符串 名称 [参数表]
p31

hash [-r] [-p 文件名] [-dtl] [名称]
p31

pwd [-LP]
p31

readonly [-aApf] [名称[=值]] ···
p32

return [n]
p32

shift [n]
p32

times
p33

trap [-lp] [参数] [信号指示···]
p33

umask [-p] [-S] [模式]
p33

unset [-fv] [名称]
p34

alias [-p] [名称[=值] ···]
p34

bind [-m 键映射] [-lpsvPSV]
bind [-m 键映射] [-q 函数] [-u 函数] [-r 键序列]
bind [-m 键映射] -f 文件名
bind [-m 键映射] -x 键序列:shell 命令
bind [-m 键映射] 键序列:函数名
bind Readline 命令
p34

builtin [shell 内部命令 [参数表]]
p35

caller [表达式]
p35

command [-pVv] 命令 [参数表 ···]
p35

declare [-aAfFilrtux] [-p] [名称[=值] ···]
p36

echo [-neE] [参数 ···]
p37

enable [-a] [-dnps] [-f 文件名] [名称 ···]
p37

help [-dms] [模式]
p38

let 表达式 [表达式]
p38

local [选项] 名称[=值] ···
p38

logout [n]
p38

mapfile [-n 行数] [-O 原下标] [-s 忽略行数] [-t] [-u 文件描述符] [-C 回调程序] [-c 数量] [数组]
p38

printf [-v 变量] 格式 [参数表]
p39

read [-ers] [-a 数组名称] [-d 分隔符] [-i 文本] [-n 字符数] [-p 提示符] [-t 超时时间] [-u 文件描述符] [名称 ···]
p39

readarray [-n 行数] [-O 原下标] [-s 忽略行数] [-t] [-u 文件描述符] [-C 回调程序] [-c 数量] [数组]
p40

source 文件名
p40

type [-afptP] [名称 ···]
p40

typeset [-afFrxi] [-p] [名称[=值] ···]
p40

ulimit [-abcdefilmnpqrstuvxHST] [限制数]
p41

unalias [-a] [名称 ···]
p42

set [--abefhkmnptuvxBCEHPT] [-o 选项] [参数 ···]
set [+abefhkmnptuvxBCEHPT] [+o 选项] [参数 ···]
p42

shopt [-pqsu] [-o] [选项名称 ···]
p45

bash [长选项] [-ir] [-abefhkmnptuvxdBCDHP] [-o 选项] [-O shopt 选项] [参数 ···]

bash [长选项] [-abefhkmnptuvxdBCDHP] [-o 选项] [-O shopt 选项] -c string [参数 ···]

bash [长选项] -s [-abefhkmnptuvxdBCDHP] [-o 选项] [-O shopt 选项] [参数 ···]
p55

数组名[下标]=值
p62

declare -a 数组名
p62

declare -a 数组名[下标]
p62

declare -A 数组名
p62

数组名=([下标=]值 ··· [下标=]值)
p62

数组名[下标]=值
p63

${数组名[下标]}
p63

dirs [+N — -N] [-clpv]
p63

popd [+N — -N] [-n]
p64

pushd [-n] [+N — -N — 目录]
p64

bg [作业指示 ···]
p70

fg [作业指示]
p70

jobs [-lnprs] [作业指示]
jobs -x 命令 [参数表]
p70

kill [-s 信号指示] [-n 信号数字] [-信号指示] 作业指示或进程号
kill -l [退出状态]
p71

wait [作业指示或进程号 ···]
p71

disown [-ar] [-h] [作业指示 ···]
p71

suspend [-f]
p71

set 变量 值
p75

compgen [选项] [单词]
p87

complete [-abcdefgjksuv] [-o 补全选项] [-E] [-A 动作] [-G 模式] [-W 单词列表] [-F 函数] [-C 命令] [-X 过滤模式] [-P 前缀] [-S 后缀] 名称 [名称 ···]

complete -pr [-E] [名称 ···]
p88

compopt [-o 选项] [+o 选项] [名称]
p89

fc [-e 编辑器] [-lnr] [第一个] [最后一个]
fc -s [模式=替换文本] [命令]
p90

history [n]
history -c
history -d 偏移量
history [-anrw] [文件名]
history -ps 参数
p91

附录B 常见问题

本章内容将会根据网友的反馈不断增补。

问 1. 在终端里面输入的命令会进行别名扩展。怎么才能只针对当前输入的命令禁止这样的扩展？

例如，对于 `alias ls='ls -l'` 这样的别名，可以输入 `/bin/ls` 或 `/textbackslashls` 来绕过这个别名扩展。不过这两种方法虽然简单，却都有弊端。第一种方法限制了命令的路径，对于不是在标准路径下的命令就有移植性的问题；第二种方法使用了转义字符，在现实中往往会带来其它问题，不够通用。通用的办法是：想用别名时使用 `shopt -s expand_alias`，不想用时 `shopt -u expand_alias`。

附录C 索引