

阮一峰 Bash 脚本教程



本教程介绍 Linux 命令行 Bash 的基本用法和脚本编程。



下载手机APP
畅享精彩阅读

目 录

致谢

README

简介

基本语法

模式扩展

引号和转义

变量

字符串操作

算术运算

行操作

目录堆栈

脚本入门

read 命令

条件判断

循环

函数

数组

set 命令

脚本除错

mktemp 命令，trap 命令

启动环境

命令提示符

归档和备份

异步任务

标准I/O

文件操作

文件系统

硬件操作

主机管理

命名管道

进程管理

重定向

正则表达式

系统信息

文本处理

时间管理

用户管理

Shell 的命令

alias

awk

cal

cat

clear

cp 命令

cut

date

dd

df

du

egrep

export

file

find

fmt

grep

gunzip

gzcat

gzip

kill

killall

last

lpq

lpr

ls

nl

ps

scp

sed

sort

tr

uname

uniq

uptime

w

WC

whereis

which

who

致谢

当前文档《阮一峰 Bash 脚本教程》由 进击的皇虫 使用 书栈网(BookStack.CN) 进行构建，生成于 2020-05-12。

书栈网仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈网难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到书栈网，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到书栈网获取最新的文档，以跟上知识更新换代的步伐。

内容来源：阮一峰 <https://github.com/wangdoc/bash-tutorial>

文档地址：<http://www.bookstack.cn/books/bash-tutorial>

书栈官网：<https://www.bookstack.cn>

书栈开源：<https://github.com/TruthHun>

分享，让知识传承更久远！感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

本教程介绍 Linux 命令行 Bash 的基本用法和脚本编程。

Bash 简介

Bash 是 Unix 系统和 Linux 系统的一种 Shell（命令行环境），是目前绝大多数 Linux 发行版的默认 Shell。

Shell 的含义

学习 Bash，首先需要理解 Shell 是什么。Shell 这个单词的原意是“外壳”，跟 kernel（内核）相对应，比喻内核外面的一层，即用户跟内核交互的对话界面。

具体来说，Shell 这个词有多种含义。

首先，Shell 是一个程序，提供一个与用户对话的环境。这个环境只有一个命令提示符，让用户从键盘输入命令，所以又称为命令行环境（commandline，简称为 CLI）。Shell 接收到用户输入的命令，将命令送入操作系统执行，并将结果返回给用户。本书中，除非特别指明，Shell 指的就是命令行环境。

其次，Shell 是一个命令解释器，解释用户输入的命令。它支持变量、条件判断、循环操作等语法，所以用户可以用 Shell 命令写出各种小程序，又称为脚本（script）。这些脚本都通过 Shell 的解释执行，而不通过编译。

最后，Shell 是一个工具箱，提供了各种小工具，供用户方便地使用操作系统的功能。

Shell 的种类

Shell 有很多种，只要能给用户提供了命令行环境的程序，都可以看作是 Shell。

历史上，主要的 Shell 有下面这些。

- Bourne Shell (sh)
- Bourne Again shell (bash)
- C Shell (csh)
- TENEX C Shell (tcsh)
- Korn shell (ksh)
- Z Shell (zsh)
- Friendly Interactive Shell (fish)

Bash 是目前最常用的 Shell，除非特别指明，下文的 Shell 和 Bash 当作同义词使用，可以互换。

下面的命令可以查看当前运行的 Shell。

```
1. $ echo $SHELL
2. /bin/bash
```

下面的命令可以查看当前的 Linux 系统安装的所有 Shell。

```
1. $ cat /etc/shells
```

上面两个命令中，`$` 是命令行环境的提示符，用户只需要输入提示符后面的内容。

Linux 允许每个用户使用不同的 Shell，用户的默认 Shell 一般都是 Bash，或者与 Bash 兼容。

命令行环境

终端模拟器

如果是不带有图形环境的 Linux 系统（比如专用于服务器的系统），启动后就直接是命令行环境。

不过，现在大部分的 Linux 发行版，尤其是针对普通用户的发行版，都是图形环境。用户登录系统后，自动进入图形环境，需要自己启动终端模拟器，才能进入命令行环境。

所谓“终端模拟器”（terminal emulator）就是一个模拟命令行窗口的程序，让用户在一个窗口中使用命令行环境，并且提供各种附加功能，比如调整颜色、字体大小、行距等等。

不同 Linux 发行版（准确地说是不同的桌面环境）带有的终端程序是不一样的，比如 KDE 桌面环境的终端程序是 konsole，Gnome 桌面环境的终端程序是 gnome-terminal，用户也可以安装第三方的终端程序。所有终端程序，尽管名字不同，基本功能都是一样的，就是让用户可以进入命令行环境，使用 Shell。

命令行提示符

进入命令行环境以后，用户会看到 Shell 的提示符。提示符往往是一串前缀，最后以一个美元符号 `$` 结尾，用户可以在这个符号后面输入各种命令。

```
1. [user@hostname] $
```

上面例子中，完整的提示符是 `[user@hostname] $`，其中前缀是用户名（`user`）加上 `@`，再加主机名（`hostname`）。比如，用户名是 `bill`，主机名是 `home-machine`，前缀就是 `bill@home-machine`。

注意，根用户（root）的提示符，不以美元符号（`$`）结尾，而以井号（`#`）结尾，用来提醒用户，现在具有根权限，可以执行各种操作，务必小心，不要出现误操作。这个符号是可以自己定义的，详见《命令提示符》一章。

为了简洁，后文的命令行提示符都只使用 `$` 表示。

进入和退出方法

进入命令行环境以后，一般就已经打开 Bash 了。如果你的 Shell 不是 Bash，可以输入 `bash` 命令启动 Bash。

```
1. $ bash
```

退出 Bash 环境，可以使用 `exit` 命令，也可以同时按下 `Ctrl + d`。

```
1. $ exit
```


Bash 的基本用法就是在命令行输入各种命令，非常直观。作为练习，可以试着输入 `pwd` 命令。按下回车键，就会显示当前所在的目录。

```
1. $ pwd
2. /home/me
```

如果不小心输入了 `pwe`，会返回一个提示，表示输入出错，没有对应的可执行程序。

```
1. $ pwe
2. bash: pwe: 未找到命令
```

Shell 和 Bash 的历史

Shell 伴随着 Unix 系统的诞生而诞生。

1969年，Ken Thompson 和 Dennis Ritchie 开发了第一版的 Unix。

1971年，Ken Thompson 编写了最初的 Shell，称为 Thompson shell，程序名是 `sh`，方便用户使用 Unix。

1973年至1975年间，John R. Mashey 扩展了最初的 Thompson shell，添加了编程功能，使得 Shell 成为一种编程语言。这个版本的 Shell 称为 Mashey shell。

1976年，Stephen Bourne 结合 Mashey shell 的功能，重写一个新的 Shell，称为 Bourne shell。

1978年，加州大学伯克利分校的 Bill Joy 开发了 C shell，为 Shell 提供 C 语言的语法，程序名是 `csh`。它是第一个真正替代 `sh` 的 UNIX shell，被合并到 Berkeley UNIX 的 2BSD 版本中。

1979年，UNIX 第七版发布，内置了 Bourne Shell，导致它成为 Unix 的默认 Shell。注意，Thompson shell、Mashey shell 和 Bourne shell 都是贝尔实验室的产品，程序名都是 `sh`。对于用户来说，它们是同一个东西，只是底层代码不同而已。

1983年，David Korn 开发了Korn shell，程序名是 `ksh`。

1985年，Richard Stallman 成立了自由软件基金会（FSF），由于 Shell 的版权属于贝尔公司，所以他决定写一个自由版权的、使用 GNU 许可证的 Shell 程序，避免 Unix 的版权争议。

1988年，自由软件基金会的第一个付薪程序员 Brian Fox 写了一个 Shell，功能基本上是 Bourne shell 的克隆，叫做 Bourne-Again SHell，简称 Bash，程序名为 `bash`，任何人都可以免费使用。后来，它逐渐成为 Linux 系统的标准 Shell。

1989年，Bash 发布1.0版。

1996年，Bash 发布2.0版。

2004年，Bash 发布3.0版。

2009年，Bash 发布4.0版。

2019年，Bash 发布5.0版。

用户可以通过 `bash` 命令的 `--version` 参数或者环境变量 `$BASH_VERSION` ，查看本机的 Bash 版本。

```
1. $ bash --version
2. GNU bash, 版本 5.0.3(1)-release (x86_64-pc-linux-gnu)
3.
4. # 或者
5. $ echo $BASH_VERSION
6. 5.0.3(1)-release
```

Bash 的基本语法

本章介绍 Bash 的最基本语法。

echo 命令

由于后面的例子会大量用到 `echo` 命令，这里先介绍这个命令。

`echo` 命令的作用是在屏幕输出一行文本，可以将该命令的参数原样输出。

```
1. $ echo hello world
2. hello world
```

上面例子中，`echo` 的参数是 `hello world`，可以原样输出。

如果想要输出的是多行文本，即包括换行符。这时需要把多行文本放在引号里面。

```
1. $ echo "<HTML>
2.     <HEAD>
3.         <TITLE>Page Title</TITLE>
4.     </HEAD>
5.     <BODY>
6.         Page body.
7.     </BODY>
8. </HTML>"
```

上面例子中，`echo` 可以原样输出多行文本。

-n 参数

默认情况下，`echo` 输出的文本末尾会有一个回车符。`-n` 参数可以取消末尾的回车符，使得下一个提示符紧跟在输出内容的后面。

```
1. $ echo -n hello world
2. hello world$
```

上面例子中，`world` 后面直接就是下一行的提示符 `$`。

```
1. $ echo a;echo b
2. a
3. b
4.
5. $ echo -n a;echo b
6. ab
```

上面例子中，`-n` 参数可以让两个 `echo` 命令的输出连在一起，出现在同一行。

`-e` 参数

`-e` 参数会解释引号（双引号和单引号）里面的特殊字符（比如换行符 `\n`）。如果不使用 `-e` 参数，即默认情况下，引号会让特殊字符变成普通字符，`echo` 不解释它们，原样输出。

```
1. $ echo "Hello\nWorld"
2. Hello\nWorld
3.
4. # 双引号的情况
5. $ echo -e "Hello\nWorld"
6. Hello
7. World
8.
9. # 单引号的情况
10. $ echo -e 'Hello\nWorld'
11. Hello
12. World
```

上面代码中，`-e` 参数使得 `\n` 解释为换行符，导致输出内容里面出现换行。

命令格式

命令行环境中，主要通过使用 Shell 命令，进行各种操作。Shell 命令基本都是下面的格式。

```
1. $ command [ arg1 ... [ argN ]]
```

上面代码中，`command` 是具体的命令或者一个可执行文件，`arg1 ... argN` 是传递给命令的参数，它们是可选的。

```
1. $ ls -l
```

上面这个命令中，`ls` 是命令，`-l` 是参数。

有些参数是命令的配置项，这些配置项一般都以一个连词线开头，比如上面的 `-l`。同一个配置项往往有长和短两种形式，比如 `-l` 是短形式，`--list` 是长形式，它们的作用完全相同。短形式便于手动输入，长形式一般用在脚本之中，可读性更好，利于解释自身的含义。

```
1. # 短形式
2. $ ls -r
3.
4. # 长形式
5. $ ls --reverse
```

上面命令中，`-r` 是短形式，`--reverse` 是长形式，作用完全一样。前者便于输入，后者便于理解。

Bash 单个命令一般都是一行，用户按下回车键，就开始执行。有些命令比较长，写成多行会有利于阅读和编辑，这

时可以在每一行的结尾加上反斜杠，Bash 就会将下一行跟当前行放在一起解释。

```
1. $ echo foo bar
2.
3. # 等同于
4. $ echo foo \
5. bar
```

空格

Bash 使用空格（或 Tab 键）区分不同的参数。

```
1. $ command foo bar
```

上面命令中，`foo` 和 `bar` 之间有一个空格，所以 Bash 认为它们是两个参数。

如果参数之间有多个空格，Bash 会自动忽略多余的空格。

```
1. $ echo this is a test
2. this is a test
```

上面命令中，`a` 和 `test` 之间有多个空格，Bash 会忽略多余的空格。

分号

分号（`;`）是命令的结束符，使得一行可以放置多个命令，上一个命令执行结束后，再执行第二个命令。

```
1. $ clear; ls
```

上面例子中，Bash 先执行 `clear` 命令，执行完成后，再执行 `ls` 命令。

注意，使用分号时，第二个命令总是接着第一个命令执行，不管第一个命令执行成功或失败。

命令的组合符 `&&` 和 `||`

除了分号，Bash 还提供两个命令组合符 `&&` 和 `||`，允许更好地控制多个命令之间的继发关系。

```
1. Command1 && Command2
```

上面命令的意思是，如果 `Command1` 命令运行成功，则继续运行 `Command2` 命令。

```
1. Command1 || Command2
```

上面命令的意思是，如果 `Command1` 命令运行失败，则继续运行 `Command2` 命令。

下面是一些例子。

```
1. $ cat filelist.txt ; ls -l filelist.txt
```

上面例子中，只要 `cat` 命令执行结束，不管成功或失败，都会继续执行 `ls` 命令。

```
1. $ cat filelist.txt && ls -l filelist.txt
```

上面例子中，只有 `cat` 命令执行成功，才会继续执行 `ls` 命令。如果 `cat` 执行失败（比如不存在文件 `filelist.txt`），那么 `ls` 命令就不会执行。

```
1. $ mkdir foo || mkdir bar
```

上面例子中，只有 `mkdir foo` 命令执行失败（比如 `foo` 目录已经存在），才会继续执行 `mkdir bar` 命令。如果 `mkdir foo` 命令执行成功，就不会创建 `bar` 目录了。

type 命令

Bash 本身内置了很多命令，同时也可以执行外部程序。怎么知道一个命令是内置命令，还是外部程序呢？

`type` 命令用来判断命令的来源。

```
1. $ type echo
2. echo is a shell builtin
3. $ type ls
4. ls is hashed (/bin/ls)
```

上面代码中，`type` 命令告诉我们，`echo` 是内部命令，`ls` 是外部程序（`/bin/ls`）。

`type` 命令本身也是内置命令。

```
1. $ type type
2. type is a shell builtin
```

如果要查看一个命令的所有定义，可以使用 `type` 命令的 `-a` 参数。

```
1. $ type -a echo
2. echo is shell builtin
3. echo is /usr/bin/echo
4. echo is /bin/echo
```

上面代码表示，`echo` 命令即是内置命令，也有对应的外部程序。

`type` 命令的 `-t` 参数，可以返回一个命令的类型：别名（alias），关键词（keyword），函数（function），内置命令（builtin）和文件（file）。

```
1. $ type -t bash
2. file
3. $ type -t if
4. keyword
```

上面例子中，`bash` 是文件，`if` 是关键词。

快捷键

Bash 提供很多快捷键，可以大大方便操作。下面是一些最常用的快捷键，完整的介绍参见《行操作》一章。

- `Ctrl + L` : 清除屏幕并将当前行移到页面顶部。
- `Ctrl + C` : 中止当前正在执行的命令。
- `Shift + PageUp` : 向上滚动。
- `Shift + PageDown` : 向下滚动。
- `Ctrl + U` : 从光标位置删除到行首。
- `Ctrl + K` : 从光标位置删除到行尾。
- `Ctrl + D` : 关闭 Shell 会话。
- `↑` , `↓` : 浏览已执行命令的历史记录。

除了上面的快捷键，Bash 还具有自动补全功能。命令输入到一半的时候，可以按下 `Tab` 键，Bash 会自动完成剩下的部分。比如，输入 `pw`，然后按一下 `Tab` 键，Bash 会自动补上 `d`。

除了命令的自动补全，Bash 还支持路径的自动补全。有时，需要输入很长的路径，这时只需要输入前面的部分，然后按下 `Tab` 键，就会自动补全后面的部分。如果有多个可能的选择，按两次 `Tab` 键，Bash 会显示所有选项，让你选择。

Bash 的模式扩展

简介

Shell 接收到用户输入的命令以后，会根据空格将用户的输入，拆分成一个个词元（token）。然后，Shell 会扩展词元里面的特殊字符，扩展完成后才会调用相应的命令。

这种特殊字符的扩展，称为模式扩展（globbing）。其中有些用到通配符，又称为通配符扩展（wildcard expansion）。Bash 一共提供八种扩展。

- 波浪线扩展
- `?` 字符扩展
- `*` 字符扩展
- 方括号扩展
- 大括号扩展
- 变量扩展
- 子命令扩展
- 算术扩展

本章介绍这八种扩展。

Bash 是先进行扩展，再执行命令。因此，扩展的结果是由 Bash 负责的，与所要执行的命令无关。命令本身并不存在参数扩展，收到什么参数就原样执行。这一点务必需要记住。

`globbing` 这个词，来自于早期的 Unix 系统有一个 `/etc/glob` 文件，保存扩展的模板。后来 Bash 内置了这个功能，但是这个名字就保留了下来。

模式扩展与正则表达式的关系是，模式扩展早于正则表达式出现，可以看作是原始的正则表达式。它的功能没有正则那么强大灵活，但是优点是简单和方便。

Bash 允许用户关闭扩展。

```
1. $ set -o noglob
2. # 或者
3. $ set -f
```

下面的命令可以重新打开扩展。

```
1. $ set +o noglob
2. # 或者
3. $ set +f
```

波浪线扩展

波浪线 `~` 会自动扩展成当前用户的主目录。


```
1. $ echo ~  
2. /home/me
```

`~/dir` 表示扩展成主目录的某个子目录，`dir` 是主目录里面的一个子目录名。

```
1. # 进入 /home/me/foo 目录  
2. $ cd ~/foo
```

`~user` 表示扩展成用户 `user` 的主目录。

```
1. $ echo ~foo  
2. /home/foo  
3.  
4. $ echo ~root  
5. /root
```

上面例子中，Bash 会根据波浪号后面的用户名，返回该用户的主目录。

如果 `~user` 的 `user` 是不存在的用户名，则波浪号扩展不起作用。

```
1. $ echo ~nonExistedUser  
2. ~nonExistedUser
```

`~+` 会扩展成当前所在的目录，等同于 `pwd` 命令。

```
1. $ cd ~/foo  
2. $ echo ~+  
3. /home/me/foo
```

? 字符扩展

`?` 字符代表文件路径里面的任意单个字符，不包括空字符。比如，`Data???` 匹配所有 `Data` 后面跟着三个字符的文件名。

```
1. # 存在文件 a.txt 和 b.txt  
2. $ ls ?.txt  
3. a.txt b.txt
```

上面命令中，`?` 表示单个字符，所以会同时匹配 `a.txt` 和 `b.txt`。

如果匹配多个字符，就需要多个 `?` 连用。

```
1. # 存在文件 a.txt、b.txt 和 ab.txt  
2. $ ls ???.txt  
3. ab.txt
```

上面命令中，`??` 匹配了两个字符。

`?` 字符扩展属于文件名扩展，只有文件确实存在的前提下，才会发生扩展。如果文件不存在，扩展就不会发生。

```
1. # 当前目录有 a.txt 文件
2. $ echo ?.txt
3. a.txt
4.
5. # 当前目录为空目录
6. $ echo ?.txt
7. ?.txt
```

上面例子中，如果 `?.txt` 可以扩展成文件名，`echo` 命令会输出扩展后的结果；如果不能扩展成文件名，`echo` 就会原样输出 `?.txt`。

*

字符扩展

`*` 字符代表文件路径里面的任意数量的字符，包括零个字符。

```
1. # 存在文件 a.txt、b.txt 和 ab.txt
2. $ ls *.txt
3. a.txt b.txt ab.txt
4.
5. # 输出所有文件
6. $ ls *
```

下面是 `*` 匹配空字符的例子。

```
1. # 存在文件 a.txt、b.txt 和 ab.txt
2. $ ls a*.txt
3. a.txt ab.txt
4.
5. $ ls *b*
6. b.txt ab.txt
```

注意，`*` 不会匹配隐藏文件（以 `.` 开头的文件）。

```
1. # 显示所有隐藏文件
2. $ echo .*
3.
4. # 与方括号扩展结合使用，
5. # 只显示正常的隐藏文件，不显示 . 和 .. 这两个特殊文件
6. $ echo .[!..]*
```

`*` 字符扩展也属于文件名扩展，只有文件确实存在的前提下才会扩展。如果文件不存在，就会原样输出。

```
1. # 当前目录不存在 c 开头的文件
2. $ echo c*.txt
```

```
3. c*.txt
```

上面例子中，当前目录里面没有 `c` 开头的文件，导致 `c*.txt` 会原样输出。

* 只匹配当前目录，不会匹配子目录。

```
1. # 子目录有一个 a.txt
2. # 无效的写法
3. $ ls *.txt
4.
5. # 有效的写法
6. $ ls */*.txt
```

上面的例子，文本文件在子目录，`*.txt` 不会产生匹配，必须写成 `*/*.txt`。有几层子目录，就必须写几层星号。

Bash 4.0 引入了一个参数 `globstar`，当该参数打开时，允许 `**` 匹配零个或多个子目录。因此，`**/*.txt` 可以匹配顶层的文本文件和任意深度子目录的文本文件。详细介绍请看后面 `shopt` 命令的介绍。

方括号扩展

方括号扩展的形式是 `[...]`，只有文件确实存在的前提下才会扩展。如果文件不存在，就会原样输出。括号之中的任意一个字符。比如，`[aeiou]` 可以匹配五个元音字母中的任意一个。

```
1. # 存在文件 a.txt 和 b.txt
2. $ ls [ab].txt
3. a.txt b.txt
4.
5. # 只存在文件 a.txt
6. $ ls [ab].txt
7. a.txt
```

上面例子中，`[ab]` 可以匹配 `a` 或 `b`，前提是确实存在相应的文件。

方括号扩展属于文件名匹配，即扩展后的结果必须符合现有的文件路径。如果不存在匹配，就会保持原样，不进行扩展。

```
1. # 不存在文件 a.txt 和 b.txt
2. $ ls [ab].txt
3. ls: 无法访问 '[ab].txt': 没有那个文件或目录
```

上面例子中，由于扩展后的文件不存在，`[ab].txt` 就原样输出了，导致 `ls` 命名报错。

方括号扩展还有两种变体：`[^...]` 和 `[!...]`。它们表示匹配不在方括号里面的字符，这两种写法是等价的。比如，`[^abc]` 或 `[!abc]` 表示匹配除了 `a`、`b`、`c` 以外的字符。

```
1. # 存在 aaa、bbb、aba 三个文件
2. $ ls ?[!a]?
3. aba bbb
```

上面命令中，`[!a]` 表示文件名第二个字符不是 `a` 的文件名，所以返回了 `aba` 和 `bbb` 两个文件。

注意，如果需要匹配 `[` 字符，可以放在方括号内，比如 `[[aeiou]`。如果需要匹配连字号 `-`，只能放在方括号内部的开头或结尾，比如 `[-aeiou]` 或 `[aeiou-]`。

[start-end] 扩展

方括号扩展有一个简写形式 `[start-end]`，表示匹配一个连续的范围。比如，`[a-c]` 等同于 `[abc]`，`[0-9]` 匹配 `[0123456789]`。

```
1. # 存在文件 a.txt、b.txt 和 c.txt
2. $ ls [a-c].txt
3. a.txt
4. b.txt
5. c.txt
6.
7. # 存在文件 report1.txt、report2.txt 和 report3.txt
8. $ ls report[0-9].txt
9. report1.txt
10. report2.txt
11. report3.txt
12. ...
```

下面是一些常用简写的例子。

- `[a-z]`：所有小写字母。
- `[a-zA-Z]`：所有小写字母与大写字母。
- `[a-zA-Z0-9]`：所有小写字母、大写字母与数字。
- `[abc]*`：所有以 `a`、`b`、`c` 字符之一开头的文件名。
- `program.[co]`：文件 `program.c` 与文件 `program.o`。
- `BACKUP.[0-9][0-9][0-9]`：所有以 `BACKUP.` 开头，后面是三个数字的文件名。

这种简写形式有一个否定形式 `[!start-end]`，表示匹配不属于这个范围的字符。比如，`[!a-zA-Z]` 表示匹配非英文字母的字符。

```
1. $ echo report[!1-3].txt
2. report4.txt report5.txt
```

上面代码中，`[!1-3]` 表示排除1、2和3。

大括号扩展

大括号扩展 `{...}` 表示分别扩展成大括号里面的所有值，各个值之间使用逗号分隔。比如，`{1,2,3}` 扩展成 `1 2 3`。

```
1. $ echo {1,2,3}
2. 1 2 3
```

```

3.
4. $ echo d{a,e,i,u,o}g
5. dag deg dig dug dog
6.
7. $ echo Front-{A,B,C}-Back
8. Front-A-Back Front-B-Back Front-C-Back

```

注意，大括号扩展不是文件名扩展。它会扩展成所有给定的值，而不管是否有对应的文件存在。

```

1. $ ls {a,b,c}.txt
2. ls: 无法访问 'a.txt': 没有那个文件或目录
3. ls: 无法访问 'b.txt': 没有那个文件或目录
4. ls: 无法访问 'c.txt': 没有那个文件或目录

```

上面例子中，即使不存在对应的文件，`{a,b,c}` 依然扩展成三个文件名，导致 `ls` 命令报了三个错误。

另一个需要注意的地方是，大括号内部的逗号前后不能有空格。否则，大括号扩展会失效。

```

1. $ echo {1 , 2}
2. {1 , 2}

```

上面例子中，逗号前后有空格，Bash 就会认为这不是大括号扩展，而是三个独立的参数。

逗号前面可以没有值，表示扩展的第一项为空。

```

1. $ cp a.log{,.bak}
2.
3. # 等同于
4. # cp a.log a.log.bak

```

大括号可以嵌套。

```

1. $ echo {j{p,pe}g,png}
2. jpg jpeg png
3.
4. $ echo a{A{1,2},B{3,4}}b
5. aA1b aA2b aB3b aB4b

```

大括号也可以与其他模式联用，并且总是先于其他模式进行扩展。

```

1. $ echo {cat,d*}
2. cat dawg dg dig dog doug dug

```

上面例子中，会先进行大括号扩展，然后进行 `*` 扩展。

大括号可以用于多字符的模式，方括号不行（只能匹配单字符）。

```

1. $ echo {cat,dog}

```

```
2. cat dog
```

由于大括号扩展 `{...}` 不是文件名扩展，所以它总是会扩展的。这与方括号扩展 `[...]` 完全不同，如果匹配的文件不存在，方括号就不会扩展。这一点要注意区分。

```
1. # 不存在 a.txt 和 b.txt
2. $ echo [ab].txt
3. [ab].txt
4.
5. $ echo {a,b}.txt
6. a.txt b.txt
```

上面例子中，如果不存在 `a.txt` 和 `b.txt`，那么 `[ab].txt` 就会变成一个普通的文件名，而 `{a,b}.txt` 可以照样扩展。

{start..end} 扩展

大括号扩展有一个简写形式 `{start..end}`，表示扩展成一个连续序列。比如，`{a..z}` 可以扩展成26个小写英文字母。

```
1. $ echo {a..c}
2. a b c
3.
4. $ echo d{a..d}g
5. dag dbg dcg ddg
6.
7. $ echo {1..4}
8. 1 2 3 4
9.
10. $ echo Number_{1..5}
11. Number_1 Number_2 Number_3 Number_4 Number_5
```

这种简写形式支持逆序。

```
1. $ echo {c..a}
2. c b a
3.
4. $ echo {5..1}
5. 5 4 3 2 1
```

注意，如果遇到无法理解的简写，大括号模式就会原样输出，不会扩展。

```
1. $ echo {a1..3c}
2. {a1..3c}
```

这种简写形式可以嵌套使用，形成复杂的扩展。

```
1. $ echo .{mp{3..4},m4{a,b,p,v}}
2. .mp3 .mp4 .m4a .m4b .m4p .m4v
```

大括号扩展的常见用途为新建一系列目录。

```
1. $ mkdir {2007..2009}-{01..12}
```

上面命令会新建36个子目录，每个子目录的名字都是“年份-月份”。

这个写法的另一个常见用途，是直接用于 `for` 循环。

```
1. for i in {1..4}
2. do
3.     echo $i
4. done
```

上面例子会循环4次。

如果整数前面有前导 `0`，扩展输出的每一项都有前导 `0`。

```
1. $ echo {01..5}
2. 01 02 03 04 05
3.
4. $ echo {001..5}
5. 001 002 003 004 005
```

这种简写形式还可以使用第二个双点号（`start..end..step`），用来指定扩展的步长。

```
1. $ echo {0..8..2}
2. 0 2 4 6 8
```

上面代码将 `0` 扩展到 `8`，每次递增的长度为 `2`，所以一共输出5个数字。

多个简写形式连用，会有循环处理的效果。

```
1. $ echo {a..c}{1..3}
2. a1 a2 a3 b1 b2 b3 c1 c2 c3
```

变量扩展

Bash 将美元符号 `$` 开头的词元视为变量，将其扩展成变量值，详见《Bash 变量》一章。

```
1. $ echo $SHELL
2. /bin/bash
```

变量名除了放在美元符号后面，也可以放在 `${}` 里面。

```
1. $ echo ${SHELL}
2. /bin/bash
```

`${!string*}` 或 `${!string@}` 返回所有匹配给定字符串 `string` 的变量名。

```
1. $ echo ${!S*}
2. SECONDS SHELL SHELLOPTS SHLVL SSH_AGENT_PID SSH_AUTH_SOCK
```

上面例子中，`${!S*}` 扩展成所有以 `S` 开头的变量名。

子命令扩展

`$(...)` 可以扩展成另一个命令的运行结果，该命令的所有输出都会作为返回值。

```
1. $ echo $(date)
2. Tue Jan 28 00:01:13 CST 2020
```

上面例子中，`$(date)` 返回 `date` 命令的运行结果。

还有另一种较老的语法，子命令放在反引号之中，也可以扩展成命令的运行结果。

```
1. $ echo `date`
2. Tue Jan 28 00:01:13 CST 2020
```

`$(...)` 可以嵌套，比如 `$(ls $(pwd))`。

算术扩展

`$(...)` 可以扩展成整数运算的结果，详见《Bash 的算术运算》一章。

```
1. $ echo $((2 + 2))
2. 4
```

字符类

`[:class:]` 表示一个字符类，扩展成某一类特定字符之中的一个。常用的字符类如下。

- `[:alnum:]`：匹配任意英文字母与数字
- `[:alpha:]`：匹配任意英文字母
- `[:blank:]`：空格和 Tab 键。
- `[:cntrl:]`：ASCII 码 0-31 的不可打印字符。
- `[:digit:]`：匹配任意数字 0-9。
- `[:graph:]`：A-Z、a-z、0-9 和标点符号。
- `[:lower:]`：匹配任意小写字母 a-z。

- `[:print:]` : ASCII 码 32-127 的可打印字符。
- `[:punct:]` : 标点符号 (除了 A-Z、a-z、0-9 的可打印字符)。
- `[:space:]` : 空格、Tab、LF (10)、VT (11)、FF (12)、CR (13)。
- `[:upper:]` : 匹配任意大写字母 A-Z。
- `[:xdigit:]` : 16进制字符 (A-F、a-f、0-9)。

请看下面的例子。

```
1. $ echo [:upper:]*
```

上面命令输出所有大写字母开头的文件名。

字符类的第一个方括号后面, 可以加上感叹号 `!`, 表示否定。比如, `[![:digit:]]` 匹配所有非数字。

```
1. $ echo [![:digit:]]*
```

上面命令输出所有不以数字开头的文件名。

字符类也属于文件名扩展, 如果没有匹配的文件名, 字符类就会原样输出。

```
1. # 不存在以大写字母开头的文件
2. $ echo [:upper:]*
3. [:upper:]*
```

上面例子中, 由于没有可匹配的文件, 字符类就原样输出了。

使用注意点

通配符有一些使用注意点, 不可不知。

(1) 通配符是先解释, 再执行。

Bash 接收到命令以后, 发现里面有通配符, 会进行通配符扩展, 然后再执行命令。

```
1. $ ls a*.txt
2. ab.txt
```

上面命令的执行过程是, Bash 先将 `a*.txt` 扩展成 `ab.txt`, 然后再执行 `ls ab.txt`。

(2) 文件名扩展在不匹配时, 会原样输出。

文件名扩展在没有可匹配的文件时, 会原样输出。

```
1. # 不存在 r 开头的文件名
2. $ echo r*
3. r*
```

上面代码中，由于不存在 `r` 开头的文件名，`r*` 会原样输出。

下面是另一个例子。

```
1. $ ls *.csv
2. ls: *.csv: No such file or directory
```

另外，前面已经说过，大括号扩展 `{...}` 不是文件名扩展。

（3）只适用于单层路径。

所有文件名扩展只匹配单层路径，不能跨目录匹配，即无法匹配子目录里面的文件。或者说，`?` 或 `*` 这样的通配符，不能匹配路径分隔符（`/`）。

如果要匹配子目录里面的文件，可以写成下面这样。

```
1. $ ls */*.txt
```

Bash 4.0 新增了一个 `globstar` 参数，允许 `**` 匹配零个或多个子目录，详见后面 `shopt` 命令的介绍。

（4）文件名可以使用通配符。

Bash 允许文件名使用通配符，即文件名包括特殊字符。这时引用文件名，需要把文件名放在单引号里面。

```
1. $ touch 'fo*'
2. $ ls
3. fo*
```

上面代码创建了一个 `fo*` 文件，这时 `*` 就是文件名的一部分。

量词语法

量词语法用来控制模式匹配的次数。它只有在 Bash 的 `extglob` 参数打开的情况下才能使用，不过一般是默认打开的。下面的命令可以查询。

```
1. $ shopt extglob
2. extglob          on
```

量词语法有下面几个。

- `?(pattern-list)`：匹配零个或一个模式。
- `*(pattern-list)`：匹配零个或多个模式。
- `+(pattern-list)`：匹配一个或多个模式。
- `@(pattern-list)`：只匹配一个模式。
- `!(pattern-list)`：匹配零个或一个以上的模式，但不匹配单独一个的模式。

```
1. $ ls abc?(.)txt
2. abctxt abc.txt
```

上面例子中，`?(.)` 匹配零个或一个点。

```
1. $ ls abc?(def)
2. abc abcdef
```

上面例子中，`?(def)` 匹配零个或一个 `def` 。

```
1. $ ls abc+(.txt|.php)
2. abc.php abc.txt
```

上面例子中，`+(.txt|.php)` 匹配文件有一个 `.txt` 或 `.php` 后缀名。

```
1. $ ls abc+(.txt)
2. abc.txt abc.txt.txt
```

上面例子中，`+(.txt)` 匹配文件有一个或多个 `.txt` 后缀名。

量词语法也属于文件名扩展，如果不存在可匹配的文件，就会原样输出。

```
1. # 没有 abc 开头的文件名
2. $ ls abc?(def)
3. ls: 无法访问 'abc?(def)': 没有那个文件或目录
```

上面例子中，由于没有可匹配的文件，`abc?(def)` 就原样输出，导致 `ls` 命令报错。

shopt 命令

`shopt` 命令可以调整 Bash 的行为。它有好几个参数跟通配符扩展有关。

`shopt` 命令的使用方法如下。

```
1. # 打开某个参数
2. $ shopt -s [optionname]
3.
4. # 关闭某个参数
5. $ shopt -u [optionname]
6.
7. # 查询某个参数关闭还是打开
8. $ shopt [optionname]
```

(1) dotglob 参数

`dotglob` 参数可以让扩展结果包括隐藏文件（即点开头的文件）。

正常情况下，扩展结果不包括隐藏文件。

```
1. $ ls *
2. abc.txt
```

打开 `dotglob`，就会包括隐藏文件。

```
1. $ shopt -s dotglob
2. $ ls *
3. abc.txt .config
```

(2) nullglob 参数

`nullglob` 参数可以让通配符不匹配任何文件名时，返回空字符。

默认情况下，通配符不匹配任何文件名时，会保持不变。

```
1. $ rm b*
2. rm: 无法删除 'b*': 没有那个文件或目录
```

上面例子中，由于当前目录不包括 `b` 开头的文件名，导致 `b*` 不会发生文件名扩展，保持原样不变，所以 `rm` 命令报错没有 `b*` 这个文件。

打开 `nullglob` 参数，就可以让不匹配的通配符返回空字符串。

```
1. $ shopt -s nullglob
2. $ rm b*
3. rm: 缺少操作数
```

上面例子中，由于没有 `b*` 匹配的文件名，所以 `rm b*` 扩展成了 `rm`，导致报错变成了“缺少操作数”。

(3) failglob 参数

`failglob` 参数使得通配符不匹配任何文件名时，Bash 会直接报错，而不是让各个命令去处理。

```
1. $ shopt -s failglob
2. $ rm b*
3. bash: 无匹配: b*
```

上面例子中，打开 `failglob` 以后，由于 `b*` 不匹配任何文件名，Bash 直接报错了，不再让 `rm` 命令去处理。

(4) extglob 参数

`extglob` 参数使得 Bash 支持 ksh 的一些扩展语法。它默认应该是打开的。

```
1. $ shopt extglob
2. extglob          on
```

它的主要应用是支持量词语法。如果不希望支持量词语法，可以用下面的命令关闭。

```
1. $ shopt -u extglob
```

(5) nocaseglob 参数

`nocaseglob` 参数可以让通配符扩展不区分大小写。

```
1. $ shopt -s nocaseglob
2. $ ls /windows/program*
3. /windows/ProgramData
4. /windows/Program Files
5. /windows/Program Files (x86)
```

上面例子中，打开 `nocaseglob` 以后，`program*` 就不区分大小写了，可以匹配 `ProgramData` 等。

(6) globstar 参数

`globstar` 参数可以使得 `**` 匹配零个或多个子目录。该参数默认是关闭的。

假设有下面的文件结构。

```
1. a.txt
2. sub1/b.txt
3. sub1/sub2/c.txt
```

上面的文件结构中，顶层目录、第一级子目录 `sub1`、第二级子目录 `sub1\sub2` 里面各有一个文本文件。请问怎样才能使用通配符，将它们显示出来？

默认情况下，只能写成下面这样。

```
1. $ ls *.txt */*.txt */**/*.txt
2. a.txt sub1/b.txt sub1/sub2/c.txt
```

这是因为 `*` 只匹配当前目录，如果要匹配子目录，只能一层层写出来。

打开 `globstar` 参数以后，`**` 匹配零个或多个子目录。因此，`**/*.txt` 就可以得到想要的结果。

```
1. $ shopt -s globstar
2. $ ls **/*.txt
3. a.txt sub1/b.txt sub1/sub2/c.txt
```

参考链接

- [Think You Understand Wildcards? Think Again](#)
- [Advanced Wildcard Patterns Most People Don't Know](#)

引号和转义

Bash 只有一种数据类型，就是字符串。不管用户输入什么数据，Bash 都视为字符串。因此，字符串相关的引号和转义，对 Bash 来说就非常重要。

转义

某些字符在 Bash 里面有特殊含义（比如 `$`、`&`、`*`）。

```
1. $ echo $date
2.
3. $
```

上面例子中，输出 `$date` 不会有任何结果，因为 `$` 是一个特殊字符。

如果想要原样输出这些特殊字符，就必须在它们前面加上反斜杠，使其变成普通字符。这就叫做“转义”（escape）。

```
1. $ echo \ $date
2. $date
```

上面命令中，只有在特殊字符 `$` 前面加反斜杠，才能原样输出。

反斜杠本身也是特殊字符，如果想要原样输出反斜杠，就需要对它自身转义，连续使用两个反斜线（`\\`）。

```
1. $ echo \\
2. \
```

上面例子输出了反斜杠本身。

反斜杠除了用于转义，还可以表示一些不可打印的字符。

- `\a`：响铃
- `\b`：退格
- `\n`：换行
- `\r`：回车
- `\t`：制表符

如果想要在命令行使用这些不可打印的字符，可以把它们放在引号里面，然后使用 `echo` 命令的 `-e` 参数。

```
1. $ echo a\tb
2. atb
3.
4. $ echo -e "a\tb"
5. a      b
```

上面例子中，命令行直接输出不可打印字符，Bash 不能正确解释。必须把它们放在引号之中，然后使用 `echo` 命令的 `-e` 参数。

由于反斜杠可以对换行符转义，使得 Bash 认为换行符是一个普通字符，从而可以将一行命令写成多行。

```
1. $ mv \  
2. /path/to/foo \  
3. /path/to/bar  
4.  
5. # 等同于  
6. $ mv /path/to/foo /path/to/bar
```

上面例子中，如果一条命令过长，就可以在行尾使用反斜杠，将其改写成多行。这是常见的多行命令的写法。

单引号

Bash 允许字符串放在单引号或双引号之中，加以引用。

单引号用于保留字符的字面含义，各种特殊字符在单引号里面，都会变为普通字符，比如星号（`*`）、美元符号（`$`）、反斜杠（`\`）等。

```
1. $ echo '*'  
2. *  
3.  
4. $ echo '$USER'  
5. $USER  
6.  
7. $ echo '$((2+2))'  
8. $((2+2))  
9.  
10. $ echo '$(echo foo)'  
11. $(echo foo)
```

上面命令中，单引号使得 Bash 扩展、变量引用、算术运算和子命令，都失效了。如果不使用单引号，它们都会被 Bash 自动扩展。

由于反斜杠在单引号里面变成了普通字符，所以如果单引号之中，还要使用单引号，不能使用转义，需要在外层的单引号前面加上一个美元符号（`$`），然后再对里层的单引号转义。

```
1. # 不正确  
2. $ echo it's  
3.  
4. # 不正确  
5. $ echo 'it\'s'  
6.  
7. # 正确  
8. $ echo $'it\'s'
```

不过，更合理的方法是改在双引号之中使用单引号。

```
1. $ echo "it's"  
2. it's
```

双引号

双引号比单引号宽松，可以保留大部分特殊字符的本来含义，但是三个字符除外：美元符号（`$`）、反引号（```）和反斜杠（`\`）。也就是说，这三个字符在双引号之中，会被 Bash 自动扩展。

```
1. $ echo "*"
2. *
```

上面例子中，通配符 `*` 放在双引号之中，就变成了普通字符，会原样输出。这一点需要特别留意，双引号里面不会进行文件名扩展。

```
1. $ echo "$SHELL"  
2. /bin/bash  
3.  
4. $ echo "`date`"  
5. Mon Jan 27 13:33:18 CST 2020
```

上面例子中，美元符号和反引号在双引号中，都保持特殊含义。美元符号用来引用变量，反引号则是执行子命令。

```
1. $ echo "I'd say: \"hello!\""
2. I'd say: "hello!"
3.  
4. $ echo "\"\""  
5. \
```

上面例子中，反斜杠在双引号之中保持特殊含义，用来转义。所以，可以使用反斜杠，在双引号之中插入双引号，或者插入反斜杠本身。

由于双引号将换行符解释为普通字符，所以可以利用双引号，在命令行输入多行文本。

```
1. $ echo "hello  
2. world"  
3. hello  
4. world
```

上面命令中，Bash 正常情况下会将换行符解释为命令结束，但是换行符在双引号之中就是普通字符，所以可以输入多行。`echo` 命令会将换行符原样输出，显示的时候正常解释为换行。

双引号的另一个常见的使用场合是，文件名包含空格。这时就必须使用双引号，将文件名放在里面。

```
1. $ ls "two words.txt"
```

上面命令中，`two words.txt` 是一个包含空格的文件名，否则就会被 Bash 当作两个文件。

双引号会原样保存多余的空格。

```
1. $ echo "this is a    test"
2. this is a    test
```

双引号还有一个作用，就是保存原始命令的输出格式。

```
1. # 单行输出
2. $ echo $(cal)
3. 一月 2020 日 一 二 三 四 五 六 1 2 3 ... 31
4.
5. # 原始格式输出
6. $ echo "$(cal)"
7.      一月 2020
8. 日 一 二 三 四 五 六
9.      1  2  3  4
10.   5  6  7  8  9 10 11
11. 12 13 14 15 16 17 18
12. 19 20 21 22 23 24 25
13. 26 27 28 29 30 31
```

上面例子中，如果 `$(cal)` 不放在双引号之中，`echo` 就会将所有结果以单行输出，丢弃了所有原始的格式。

Here 文档

Here 文档 (here document) 是一种输入多行字符串的方法，格式如下。

```
1. << token
2. text
3. token
```

它的格式分成开始标记 (`<< token`) 和结束标记 (`token`)。开始标记是两个小于号 + Here 文档的名称，名称可以随意取，后面必须是一个换行符；结束标记是单独一行顶格写的 Here 文档名称，如果不是顶格，结束标记不起作用。两者之间就是多行字符串的内容。

下面是一个通过 Here 文档输出 HTML 代码的例子。

```
1. $ cat << _EOF_
2. <html>
3. <head>
4.     <title>
5.         The title of your page
6.     </title>
7. </head>
8.
9. <body>
10.     Your page content goes here.
11. </body>
12. </html>
```

```
13. _EOF_
```

Here 文档内部会发生变量替换和通配符扩展，但是双引号和单引号都失去语法作用，变成了普通字符。

```
1. $ foo='hello world'
2. $ cat << _example_
3. $foo
4. "$foo"
5. '$foo'
6. _example_
7.
8. hello world
9. "hello world"
10. 'hello world'
```

上面例子中，变量 `$foo` 发生了替换，但是双引号和单引号都原样输出了，表明它们已经失去了引用的功能。

如果不希望发生变量替换和通配符扩展，可以把 Here 文档的开始标记放在单引号之中。

```
1. $ foo='hello world'
2. $ cat << '_example_'
3. $foo
4. "$foo"
5. '$foo'
6. _example_
7.
8. $foo
9. "$foo"
10. '$foo'
```

上面例子中，Here 文档的开始标记 (`_example_`) 放在单引号之中，导致变量替换失效了。

Here 文档的本质是重定向，它将字符串重定向输出给某个命令，相当于包含了 `echo` 命令。

```
1. $ command << token
2.   string
3. token
4.
5. # 等同于
6.
7. $ echo string | command
```

上面代码中，Here 文档相当于 `echo` 命令的重定向。

所以，Here 字符串只适合那些可以接受标准输入作为参数的命令，对于其他命令无效，比如 `echo` 命令就不能用 Here 文档作为参数。

```
1. $ echo << _example_
2. hello
3. _example_
```

上面例子不会有任何输出，因为 Here 文档对于 `echo` 命令无效。

此外，Here 文档也不能作为变量的值，只能用于命令的参数。

Here 字符串

Here 文档还有一个变体，叫做 Here 字符串 (Here string)，使用三个小于号 (`<<<`) 表示。

```
1. <<< string
```

它的作用是将字符串通过标准输入，传递给命令。

有些命令直接接受给定的参数，与通过标准输入接受参数，结果是不一样的。所以才有了这个语法，使得将字符串通过标准输入传递给命令更方便，比如 `cat` 命令只接受标准输入传入的字符串。

```
1. $ cat <<< 'hi there'
2. # 等同于
3. $ echo 'hi there' | cat
```

上面的第一种语法使用了 Here 字符串，要比第二种语法看上去语义更好，也更简洁。

```
1. $ md5sum <<< 'ddd'
2. # 等同于
3. $ echo 'ddd' | md5sum
```

上面例子中，`md5sum` 命令只能接受标准输入作为参数，不能直接将字符串放在命令后面，会被当作文件名，即 `md5sum ddd` 里面的 `ddd` 会被解释成文件名。这时就可以用 Here 字符串，将字符串传给 `md5sum` 命令。

Bash 变量

Bash 变量分成环境变量和自定义变量两类。

简介

环境变量是 Bash 环境自带的变量，进入 Shell 时已经定义好了，可以直接使用。它们通常是系统定义好的，也可以由用户从父 Shell 传入子 Shell。

`env` 命令或 `printenv` 命令，可以显示所有环境变量。

```
1. $ env
2. # 或者
3. $ printenv
```

下面是一些常见的环境变量。

- `BASHPID` : Bash 进程的进程 ID。
- `BASHOPTS` : 当前 Shell 的参数，可以用 `shopt` 命令修改。
- `DISPLAY` : 图形环境的显示器名字，通常是 `:0`，表示 X Server 的第一个显示器。
- `EDITOR` : 默认的文本编辑器。
- `HOME` : 用户的主目录。
- `HOST` : 当前主机的名称。
- `IFS` : 词与词之间的分隔符，默认为空格。
- `LANG` : 字符集以及语言编码，比如 `zh_CN.UTF-8`。
- `PATH` : 由冒号分开的目录列表，当输入可执行程序名后，会搜索这个目录列表。
- `PS1` : Shell 提示符。
- `PS2` : 输入多行命令时，次要的 Shell 提示符。
- `PWD` : 当前工作目录。
- `RANDOM` : 返回一个0到32767之间的随机数。
- `SHELL` : Shell 的名字。
- `SHELLOPTS` : 启动当前 Shell 的 `set` 命令的参数，参见《set 命令》一章。
- `TERM` : 终端类型名，即终端仿真器所用的协议。
- `UID` : 当前用户的 ID 编号。
- `USER` : 当前用户的用户名。

很多环境变量很少发生变化，而且是只读的，可以视为常量。由于它们的变量名全部都是大写，所以传统上，如果用户要自己定义一个常量，也会使用全部大写的变量名。

注意，Bash 变量名区分大小写，`HOME` 和 `home` 是两个不同的变量。

查看单个环境变量的值，可以使用 `printenv` 命令或 `echo` 命令。

```
1. $ printenv PATH
2. # 或者
3. $ echo $PATH
```

注意， `printenv` 命令后面的变量名，不用加前缀 `$` 。

自定义变量是用户在当前 Shell 里面自己定义的变量，必须先定义后使用，而且仅在当前 Shell 可用。一旦退出当前 Shell，该变量就不存在了。

`set` 命令可以显示所有变量（包括环境变量和自定义变量），以及所有的 Bash 函数。

```
1. $ set
```

创建变量

用户创建变量的时候，变量名必须遵守下面的规则。

- 字母、数字和下划线字符组成。
- 第一个字符必须是一个字母或一个下划线，不能是数字。
- 不允许出现空格和标点符号。

变量声明的语法如下。

```
1. variable=value
```

上面命令中，等号左边是变量名，右边是变量。注意，等号两边不能有空格。

如果变量的值包含空格，则必须将值放在引号中。

```
1. myvar="hello world"
```

Bash 没有数据类型的概念，所有的变量值都是字符串。

下面是一些自定义变量的例子。

```
1. a=z                # 变量 a 赋值为字符串 z
2. b="a string"       # 变量值包含空格，就必须放在引号里面
3. c="a string and $b" # 变量值可以引用其他变量的值
4. d="\t\\ta string\\n" # 变量值可以使用转义字符
5. e=$(ls -l foo.txt)  # 变量值可以是命令的执行结果
6. f=$((5 * 7))       # 变量值可以是数学运算的结果
```

变量可以重复赋值，后面的赋值会覆盖前面的赋值。

```
1. $ foo=1
2. $ foo=2
3. $ echo $foo
4. 2
```

上面例子中，变量 `foo` 的第二次赋值会覆盖第一次赋值。

读取变量

读取变量的时候，直接在变量名前加上 `$` 就可以了。

```
1. $ foo=bar
2. $ echo $foo
3. bar
```

每当 Shell 看到以 `$` 开头的单词时，就会尝试读取这个变量名对应的值。

如果变量不存在，Bash 不会报错，而会输出空字符。

由于 `$` 在 Bash 中有特殊含义，把它当作美元符号使用时，一定要非常小心，

```
1. $ echo The total is $100.00
2. The total is 00.00
```

上面命令的原意是输入 `$100`，但是 Bash 将 `$1` 解释成了变量，该变量为空，因此输入就变成了 `00.00`。所以，如果要使用 `$` 的原义，需要在 `$` 前面放上反斜杠，进行转义。

```
1. $ echo The total is \$100.00
2. The total is $100.00
```

读取变量的时候，变量名也可以使用花括号 `{}` 包围，比如 `$a` 也可以写成 `${a}`。这种写法可以用于变量名与其他字符连用的情况。

```
1. $ a=foo
2. $ echo $a_file
3.
4. $ echo ${a}_file
5. foo_file
```

上面代码中，变量名 `a_file` 不会有任何输出，因为 Bash 将其整个解释为变量，而这个变量是不存在的。只有用花括号区分 `$a`，Bash 才能正确解读。

事实上，读取变量的语法 `$foo`，可以看作是 `${foo}` 的简写形式。

如果变量的值本身也是变量，可以使用 `${!varname}` 的语法，读取最终的值。

```
1. $ myvar=USER
2. $ echo ${!myvar}
3. ruanyf
```

上面的例子中，变量 `myvar` 的值是 `USER`，`${!myvar}` 的写法将其展开成最终的值。

删除变量

`unset` 命令用来删除一个变量。

```
1. unset NAME
```

这个命令不是很有用。因为不存在的 Bash 变量一律等于空字符串，所以即使 `unset` 命令删除了变量，还是可以读取这个变量，值为空字符串。

所以，删除一个变量，也可以将这个变量设成空字符串。

```
1. $ foo=''
2. $ foo=
```

上面两种写法，都是删除了变量 `foo`。由于不存在的值默认为空字符串，所以后一种写法可以在等号右边不写任何值。

输出变量，`export` 命令

用户创建的变量仅可用于当前 Shell，子 Shell 默认读取不到父 Shell 定义的变量。为了把变量传递给子 Shell，需要使用 `export` 命令。这样输出的变量，对于子 Shell 来说就是环境变量。

`export` 命令用来向子 Shell 输出变量。

```
1. NAME=foo
2. export NAME
```

上面命令输出了变量 `NAME`。变量的赋值和输出也可以在一个步骤中完成。

```
1. export NAME=value
```

上面命令执行后，当前 Shell 及随后新建的子 Shell，都可以读取变量 `$NAME`。

子 Shell 如果修改继承的变量，不会影响父 Shell。

```
1. # 输出变量 $foo
2. $ export foo=bar
3.
4. # 新建子 Shell
5. $ bash
6.
7. # 读取 $foo
8. $ echo $foo
9. bar
10.
11. # 修改继承的变量
12. $ foo=baz
13.
14. # 退出子 Shell
15. $ exit
```

```
16.  
17. # 读取 $foo  
18. $ echo $foo  
19. bar
```

上面例子中，子 Shell 修改了继承的变量 `$foo`，对父 Shell 没有影响。

特殊变量

Bash 提供一些特殊变量。这些变量的值由 Shell 提供，用户不能进行赋值。

(1) `$?`

`$?` 为上一个命令的退出码，用来判断上一个命令是否执行成功。返回值是 `0`，表示上一个命令执行成功；如果是非零，上一个命令执行失败。

```
1. $ ls doesnotexist  
2. ls: doesnotexist: No such file or directory  
3.  
4. $ echo $?  
5. 1
```

上面例子中，`ls` 命令查看一个不存在的文件，导致报错。`$?` 为1，表示上一个命令执行失败。

(2) `$$`

`$$` 为当前 Shell 的进程 ID。

```
1. $ echo $$  
2. 10662
```

这个特殊变量可以用来命名临时文件。

```
1. LOGFILE=/tmp/output_log.$$
```

(3) `$_`

`$_` 为上一个命令的最后一个参数。

```
1. $ grep dictionary /usr/share/dict/words  
2. dictionary  
3.  
4. $ echo $_  
5. /usr/share/dict/words
```

(4) `#!`

`#!` 为最近一个后台执行的异步命令的进程 ID。


```
1. $ firefox &
2. [1] 11064
3.
4. $ echo $!
5. 11064
```

上面例子中，`firefox` 是后台运行的命令，`$!` 返回该命令的进程 ID。

(5) `$0`

`$0` 为当前 Shell 的名称（在命令行直接执行时）或者脚本名（在脚本中执行时）。

```
1. $ echo $0
2. bash
```

上面例子中，`$0` 返回当前运行的是 Bash。

(6) `$-`

`$-` 为当前 Shell 的启动参数。

```
1. $ echo $-
2. himBHS
```

(7) `$@` 和 `$#`

`$@` 和 `$#` 表示脚本的参数数量，参见脚本一章。

变量的默认值

Bash 提供四个特殊语法，跟变量的默认值有关，目的是保证变量不为空。

```
1. ${varname:-word}
```

上面语法的含义是，如果变量 `varname` 存在且不为空，则返回它的值，否则返回 `word`。它的目的是返回一个默认值，比如 `${count:-0}` 表示变量 `count` 不存在时返回 `0`。

```
1. ${varname:=word}
```

上面语法的含义是，如果变量 `varname` 存在且不为空，则返回它的值，否则将它设为 `word`，并且返回 `word`。它的目的是设置变量的默认值，比如 `${count:=0}` 表示变量 `count` 不存在时返回 `0`，且将 `count` 设为 `0`。

```
1. ${varname:+word}
```

上面语法的含义是，如果变量名存在且不为空，则返回 `word`，否则返回空值。它的目的是测试变量是否存在，比如 `${count:+1}` 表示变量 `count` 存在时返回 `1`（表示 `true`），否则返回空值。

```
1. ${varname:?message}
```

上面语法的含义是，如果变量 `varname` 存在且不为空，则返回它的值，否则打印出 `varname: message`，并中断脚本的执行。如果省略了 `message`，则输出默认的信息“parameter null or not set.”。它的目的是防止变量未定义，比如 `${count:? "undefined!"}` 表示变量 `count` 未定义时就中断执行，抛出错误，返回给定的报错信息 `undefined!`。

上面四种语法如果用在脚本中，变量名的部分可以用到数字 `1` 到 `9`，表示脚本的参数。

```
1. filename=${1:? "filename missing."}
```

上面代码出现在脚本中，`1` 表示脚本的第一个参数。如果该参数不存在，就退出脚本并报错。

declare 命令

`declare` 命令可以声明一些特殊类型的变量，为变量设置一些限制，比如声明只读类型的变量和整数类型的变量。

它的语法形式如下。

```
1. declare OPTION VARIABLE=value
```

`declare` 命令的主要参数（OPTION）如下。

- `-a`：声明数组变量。
- `-f`：输出所有函数定义。
- `-F`：输出所有函数名。
- `-i`：声明整数变量。
- `-l`：声明变量为小写字母。
- `-p`：查看变量信息。
- `-r`：声明只读变量。
- `-u`：声明变量为大写字母。
- `-x`：该变量输出为环境变量。

`declare` 命令如果用在函数中，声明的变量只在函数内部有效，等同于 `local` 命令。

不带任何参数时，`declare` 命令输出当前环境的所有变量，包括函数在内，等同于不带有任何参数的 `set` 命令。

```
1. $ declare
```

（1）`-i` 参数

`-i` 参数声明整数变量以后，可以直接进行数学运算。

```
1. $ declare -i val1=12 val2=5
2. $ declare -i result
3. $ result=val1*val2
4. $ echo $result
```

```
5. 60
```

上面例子中，如果变量 `result` 不声明为整数，`val1*val2` 会被当作字面量，不会进行整数运算。另外，`val1` 和 `val2` 其实不需要声明为整数，因为只要 `result` 声明为整数，它的赋值就会自动解释为整数运算。

注意，一个变量声明为整数以后，依然可以被改写为字符串。

```
1. $ declare -i var=12
2. $ var=foo
3. $ echo $var
4. 0
```

上面例子中，变量 `var` 声明为整数，覆盖以后，Bash 不会报错，但会赋以不确定的值，上面的例子中可能输出0，也可能输出的是3。

(2) `-x` 参数

`-x` 参数等同于 `export` 命令，可以输出一个变量为子 Shell 的环境变量。

```
1. $ declare -x foo
2. # 等同于
3. $ export foo
```

(3) `-r` 参数

`-r` 参数可以声明只读变量，无法改变变量值，也不能 `unset` 变量。

```
1. $ declare -r bar=1
2.
3. $ bar=2
4. bash: bar: 只读变量
5. $ echo $?
6. 1
7.
8. $ unset bar
9. bash: bar: 只读变量
10. $ echo $?
11. 1
```

上面例子中，后两个赋值语句都会报错，命令执行失败。

(4) `-u` 参数

`-u` 参数声明变量为大写字母，可以自动把变量值转成大写字母。

```
1. $ declare -u foo
2. $ foo=upper
3. $ echo $foo
4. UPPER
```

(5) -l 参数

-l 参数声明变量为小写字母，可以自动把变量值转成小写字母。

```
1. $ declare -l bar
2. $ bar=LOWER
3. $ echo $bar
4. lower
```

(6) -p 参数

-p 参数输出变量信息。

```
1. $ foo=hello
2. $ declare -p foo
3. declare -- foo="hello"
4. $ declare -p bar
5. bar : 未找到
```

上面例子中，`declare -p` 可以输出已定义变量的值，对于未定义的变量，会提示找不到。

如果不提供变量名，`declare -p` 输出所有变量的信息。

```
1. $ declare -p
```

(7) -f 参数

-f 参数输出当前环境的所有函数，包括它的定义。

```
1. $ declare -f
```

(8) -F 参数

-F 参数输出当前环境的所有函数名，不包含函数定义。

```
1. $ declare -F
```

readonly 命令

readonly 命令等同于 `declare -r`，用来声明只读变量，不能改变变量值，也不能 `unset` 变量。

```
1. $ readonly foo=1
2. $ foo=2
3. bash: foo : 只读变量
4. $ echo $?
5. 1
```

上面例子中，更改只读变量 `foo` 会报错，命令执行失败。

`readonly` 命令有三个参数。

- `-f`：声明的变量为函数名。
- `-p`：打印出所有的只读变量。
- `-a`：声明的变量为数组。

let 命令

`let` 命令声明变量时，可以直接执行算术表达式。

```
1. $ let foo=1+2
2. $ echo $foo
3. 3
```

上面例子中，`let` 命令可以直接计算 `1 + 2`。

`let` 命令的参数表达式如果包含空格，就需要使用引号。

```
1. $ let "foo = 1 + 2"
```

`let` 可以同时为多个变量赋值，赋值表达式之间使用空格分隔。

```
1. $ let "v1 = 1" "v2 = v1++"
2. $ echo $v1,$v2
3. 2,1
```

上面例子中，`let` 声明了两个变量 `v1` 和 `v2`，其中 `v2` 等于 `v1++`，表示先返回 `v1` 的值，然后 `v1` 自增。

这种语法支持的运算符，参考《Bash 的算术运算》一章。

字符串操作

本章介绍 Bash 字符串操作的语法。

字符串的长度

获取字符串长度的语法如下。

```
1. ${#varname}
```

下面是一个例子。

```
1. $ myPath=/home/cam/book/long.file.name
2. $ echo ${#myPath}
3. 29
```

大括号 `{}` 是必需的，否则 Bash 会将 `$#` 理解成脚本的参数个数，将变量名理解成文本。

```
1. $ echo $#myvar
2. 0myvar
```

上面例子中，Bash 将 `$#` 和 `myvar` 分开解释了。

子字符串

字符串提取子串的语法如下。

```
1. ${varname:offset:length}
```

上面语法的含义是返回变量 `$varname` 的子字符串，从位置 `offset` 开始（从 `0` 开始计算），长度为 `length`。

```
1. $ count=frogfootman
2. $ echo ${count:4:4}
3. foot
```

上面例子返回字符串 `frogfootman` 从4号位置开始的长度为4的子字符串 `foot`。

这种语法不能直接操作字符串，只能通过变量来读取字符串，并且不会改变原始字符串。变量前面的美元符号可以省略。

```
1. # 报错
2. $ echo ${"hello":2:3}
```

上面例子中，`"hello"` 不是变量名，导致 Bash 报错。

如果省略 `length`，则从位置 `offset` 开始，一直返回到字符串的结尾。

```
1. $ count=frogfootman
2. $ echo ${count:4}
3. footman
```

上面例子是返回变量 `count` 从4号位置一直到结尾的子字符串。

如果 `offset` 为负值，表示从字符串的末尾开始算起。注意，负数前面必须有一个空格，以防止与 `${variable:-word}` 的变量的设置默认值语法混淆。这时，如果还指定 `length`，则 `length` 不能小于零。

```
1. $ foo="This string is long."
2. $ echo ${foo: -5}
3. long.
4. $ echo ${foo: -5:2}
5. lo
```

上面例子中，`offset` 为 `-5`，表示从倒数第5个字符开始截取，所以返回 `long.`。如果指定长度为 `2`，则返回 `lo`。

搜索和替换

Bash 提供字符串搜索和替换的多种方法。

(1) 字符串头部的模式匹配。

以下两种语法可以检查字符串开头，是否匹配给定的模式。如果匹配成功，就删除匹配的部分，返回剩下的部分。原始变量不会发生变化。

```
1. # 如果 pattern 匹配变量 variable 的开头，
2. # 删除最短匹配（非贪婪匹配）的部分，返回剩余部分
3. ${variable#pattern}
4.
5. # 如果 pattern 匹配变量 variable 的开头，
6. # 删除最长匹配（贪婪匹配）的部分，返回剩余部分
7. ${variable##pattern}
```

上面两种语法会删除变量字符串开头的匹配部分（将其替换为空），返回剩下的部分。区别是一个是最短匹配（又称非贪婪匹配），另一个是最长匹配（又称贪婪匹配）。

匹配模式 `pattern` 可以使用 `*`、`?`、`[]` 等通配符。

```
1. $ myPath=/home/cam/book/long.file.name
2.
3. $ echo ${myPath#/*/}
4. cam/book/long.file.name
5.
```

```
6. $ echo ${myPath##*/}
7. long.file.name
```

上面例子中，匹配的模式是 `/*`，其中 `*` 可以匹配任意数量的字符，所以最短匹配是 `/home/`，最长匹配是 `/home/cam/book/`。

下面写法可以删除文件路径的目录部分，只留下文件名。

```
1. $ path=/home/cam/book/long.file.name
2.
3. $ echo ${path##*/}
4. long.file.name
```

上面例子中，模式 `*/` 匹配目录部分，所以只返回文件名。

下面再看一个例子。

```
1. $ phone="555-456-1414"
2. $ echo ${phone#*-}
3. 456-1414
4. $ echo ${phone##*-}
5. 1414
```

如果匹配不成功，则返回原始字符串。

```
1. $ phone="555-456-1414"
2. $ echo ${phone#444}
3. 555-456-1414
```

上面例子中，原始字符串里面无法匹配模式 `444`，所以原样返回。

如果要将头部匹配的部分，替换成其他内容，采用下面的写法。

```
1. # 模式必须出现在字符串的开头
2. ${variable/#pattern/string}
3.
4. # 示例
5. $ foo=JPG.JPG
6. $ echo ${foo/#JPG/jpg}
7. jpg.JPG
```

上面例子中，被替换的 `JPG` 必须出现在字符串头部，所以返回 `jpg.JPG`。

（2）字符串尾部的模式匹配。

以下两种语法可以检查字符串结尾，是否匹配给定的模式。如果匹配成功，就删除匹配的部分，返回剩下的部分。原始变量不会发生变化。

```
1. # 如果 pattern 匹配变量 variable 的结尾，
```



```

2. # 删除最短匹配（非贪婪匹配）的部分，返回剩余部分
3. ${variable%pattern}
4.
5. # 如果 pattern 匹配变量 variable 的结尾，
6. # 删除最长匹配（贪婪匹配）的部分，返回剩余部分
7. ${variable%%pattern}

```

上面两种语法会删除变量字符串结尾的匹配部分（将其替换为空），返回剩下的部分。区别是一个是最短匹配（又称非贪婪匹配），另一个是最长匹配（又称贪婪匹配）。

```

1. $ path=/home/cam/book/long.file.name
2.
3. $ echo ${path%.*}
4. /home/cam/book/long.file
5.
6. $ echo ${path%%.*}
7. /home/cam/book/long

```

上面例子中，匹配模式是 `.*`，其中 `*` 可以匹配任意数量的字符，所以最短匹配是 `.name`，最长匹配是 `.file.name`。

下面写法可以删除路径的文件名部分，只留下目录部分。

```

1. $ path=/home/cam/book/long.file.name
2.
3. $ echo ${path%/*}
4. /home/cam/book

```

上面例子中，模式 `/*` 匹配文件名部分，所以只返回目录部分。

下面的写法可以替换文件的后缀名。

```

1. $ file=foo.png
2. $ echo ${file%.png}.jpg
3. foo.jpg

```

上面的例子将文件的后缀名，从 `.png` 改成了 `.jpg`。

下面再看一个例子。

```

1. $ phone="555-456-1414"
2. $ echo ${phone%-*}
3. 555-456
4. $ echo ${phone%%-*}
5. 555

```

如果匹配不成功，则返回原始字符串。

如果要将尾部匹配的部分，替换成其他内容，采用下面的写法。

```

1. # 模式必须出现在字符串的结尾
2. ${variable/%pattern/string}
3.
4. # 示例
5. $ foo=JPG.JPG
6. $ echo ${foo/%JPG/jpg}
7. JPG.jpg

```

上面例子中，被替换的 `JPG` 必须出现在字符串尾部，所以返回 `JPG.jpg`。

（3）任意位置的模式匹配。

以下两种语法可以检查字符串内部，是否匹配给定的模式。如果匹配成功，就删除匹配的部分，换成其他的字符串返回。原始变量不会发生变化。

```

1. # 如果 pattern 匹配变量 variable 的一部分，
2. # 最长匹配（贪婪匹配）的那部分被 string 替换，但仅替换第一个匹配
3. ${variable/pattern/string}
4.
5. # 如果 pattern 匹配变量 variable 的一部分，
6. # 最长匹配（贪婪匹配）的那部分被 string 替换，所有匹配都替换
7. ${variable//pattern/string}

```

上面两种语法都是最长匹配（贪婪匹配）下的替换，区别是前一个语法仅仅替换第一个匹配，后一个语法替换所有匹配。

```

1. $ path=/home/cam/foo/foo.name
2.
3. $ echo ${path/foo/bar}
4. /home/cam/bar/foo.name
5.
6. $ echo ${path//foo/bar}
7. /home/cam/bar/bar.name

```

上面例子中，前一个命令只替换了第一个 `foo`，后一个命令将两个 `foo` 都替换了。

下面的例子将分隔符从 `:` 换成换行符。

```

1. $ echo -e ${PATH//:/'\n'}
2. /usr/local/bin
3. /usr/bin
4. /bin
5. ...

```

上面例子中，`echo` 命令的 `-e` 参数，表示将替换后的字符串的 `\n` 字符，解释为换行符。

模式部分可以使用通配符。

```

1. $ phone="555-456-1414"

```

```
2. $ echo ${phone/5?4/-}
3. 55-56-1414
```

上面的例子将 `5-4` 替换成 `-`。

如果省略了 `string` 部分，那么就相当于匹配的部分替换成空字符串，即删除匹配的部分。

```
1. $ path=/home/cam/foo/foo.name
2.
3. $ echo ${path/./*/}
4. /home/cam/foo/foo
```

上面例子中，第二个斜杠后面的 `string` 部分省略了，所以模式 `.*` 匹配的部分 `.name` 被删除后返回。

前面提到过，这个语法还有两种扩展形式。

```
1. # 模式必须出现在字符串的开头
2. ${variable/#pattern/string}
3.
4. # 模式必须出现在字符串的结尾
5. ${variable/%pattern/string}
```

改变大小写

下面的语法可以改变变量的大小写。

```
1. # 转为大写
2. ${varname^^}
3.
4. # 转为小写
5. ${varname,,}
```

下面是一个例子。

```
1. $ foo=heLLo
2. $ echo ${foo^^}
3. HELLO
4. $ echo ${foo,,}
5. hello
```

Bash 的算术运算

算术表达式

`((...))` 语法可以进行整数的算术运算。

```
1. $ ((foo = 5 + 5))
2. $ echo $foo
3. 10
```

`((...))` 会自动忽略内部的空格，所以下面的写法都正确，得到同样的结果。

```
1. $ ((2+2))
2. $ (( 2+2 ))
3. $ (( 2 + 2 ))
```

这个语法不返回值，命令执行的结果根据算术运算的结果而定。只要算术结果不是 `0`，命令就算执行成功。

```
1. $ (( 3 + 2 ))
2. $ echo $?
3. 0
```

上面例子中，`3 + 2` 的结果是5，命令就算执行成功，环境变量 `$?` 为 `0`。

如果算术结果为 `0`，命令就算执行失败。

```
1. $ (( 3 - 3 ))
2. $ echo $?
3. 1
```

上面例子中，`3 - 3` 的结果是 `0`，环境变量 `$?` 为 `1`，表示命令执行失败。

如果要读取算术运算的结果，需要在 `((...))` 前面加上美元符号 `$((...))`，使其变成算术表达式，返回算术运算的值。

```
1. $ echo $((2 + 2))
2. 4
```

`((...))` 语法支持的算术运算符如下。

- `+`：加法
- `-`：减法
- `*`：乘法
- `/`：除法（整除）

- `%` : 余数
- `**` : 指数
- `++` : 自增运算（前缀或后缀）
- `--` : 自减运算（前缀或后缀）

注意，除法运算符的返回结果总是整数，比如 `5` 除以 `2`，得到的结果是 `2`，而不是 `2.5`。

```
1. $ echo $((5 / 2))
2. 2
```

`++` 和 `--` 这两个运算符有前缀和后缀的区别。作为前缀是先运算后返回值，作为后缀是先返回值后运算。

```
1. $ i=0
2. $ echo $i
3. 0
4. $ echo $((i++))
5. 0
6. $ echo $i
7. 1
8. $ echo $((++i))
9. 2
10. $ echo $i
11. 2
```

上面例子中，`++` 作为后缀是先返回值，执行 `echo` 命令，再进行自增运算；作为前缀则是先进行自增运算，再返回值执行 `echo` 命令。

`$((...))` 内部可以用圆括号改变运算顺序。

```
1. $ echo $(( (2 + 3) * 4 ))
2. 20
```

上面例子中，内部的圆括号让加法先于乘法执行。

`$((...))` 结构可以嵌套。

```
1. $ echo $(((5**2) * 3))
2. 75
3. # 等同于
4. $ echo $((($(5**2)) * 3))
5. 75
```

这个语法只能计算整数，否则会报错。

```
1. # 报错
2. $ echo $((1.5 + 1))
3. bash: 语法错误
```

`$(...)` 的圆括号之中，不需要在变量名之前加上 `$`，不过加上也不报错。

```
1. $ number=2
2. $ echo $(( $number + 1 ))
3. 3
```

上面例子中，变量 `number` 前面有没有美元符号，结果都是一样的。

如果在 `$(...)` 里面使用字符串，Bash 会认为那是一个变量名。如果不存在同名变量，Bash 就会将其作为空值，因此不会报错。

```
1. $ echo $(( "hello" + 2 ))
2. 2
3. $ echo $(( "hello" * 2 ))
4. 0
```

上面例子中，`"hello"` 会被当作变量名，返回空值，而 `$(...)` 会将空值当作 `0`，所以乘法的运算结果就是 `0`。同理，如果 `$(...)` 里面使用不存在的变量，也会当作 `0` 处理。

如果一个变量的值为字符串，跟上面的处理逻辑是一样的。即该字符串如果不对应已存在的变量，在 `$(...)` 里面会被当作空值。

```
1. $ foo=hello
2. $ echo $(( foo + 2 ))
3. 2
```

上面例子中，变量 `foo` 的值是 `hello`，而 `hello` 也会被看作变量名。这使得有可能写出动态替换的代码。

```
1. $ foo=hello
2. $ hello=3
3. $ echo $(( foo + 2 ))
4. 5
```

上面代码中，`foo + 2` 取决于变量 `hello` 的值。

最后，`$[...]` 是以前的语法，也可以做整数运算，不建议使用。

```
1. $ echo ${2+2}
2. 4
```

数值的进制

Bash 的数值默认都是十进制，但是在算术表达式中，也可以使用其他进制。

- `number`：没有任何特殊表示法的数字是十进制数（以10为底）。
- `0number`：八进制数。
- `0xnumber`：十六进制数。

- `base#number` : `base` 进制的数。

下面是一些例子。

```
1. $ echo $((0xff))
2. 255
3. $ echo $((2#11111111))
4. 255
```

上面例子中，`0xff` 是十六进制数，`2#11111111` 是二进制数。

位运算

`$((...))` 支持以下的二进制位运算符。

- `<<` : 位左移运算，把一个数字的所有位向左移动指定的位。
- `>>` : 位右移运算，把一个数字的所有位向右移动指定的位。
- `&` : 位的“与”运算，对两个数字的所有位执行一个 `AND` 操作。
- `|` : 位的“或”运算，对两个数字的所有位执行一个 `OR` 操作。
- `~` : 位的“否”运算，对一个数字的所有位取反。
- `!` : 逻辑“否”运算
- `^` : 位的异或运算 (exclusive or)，对两个数字的所有位执行一个异或操作。

下面是右移运算符 `>>` 的例子。

```
1. $ echo $((16>>2))
2. 4
```

下面是左移运算符 `<<` 的例子。

```
1. $ echo $((16<<2))
2. 64
```

下面是 `17` (二进制 `10001`) 和 `3` (二进制 `11`) 的各种二进制运算的结果。

```
1. $ echo $((17&3))
2. 1
3. $ echo $((17|3))
4. 19
5. $ echo $((17^3))
6. 18
```

逻辑运算

`$((...))` 支持以下的逻辑运算符。

- `<` : 小于
- `>` : 大于
- `<=` : 小于或相等
- `>=` : 大于或相等
- `==` : 相等
- `!=` : 不相等
- `&&` : 逻辑与
- `||` : 逻辑或
- `expr1?expr2:expr3` : 三元条件运算符。若表达式 `expr1` 的计算结果为非零值（算术真），则执行表达式 `expr2`，否则执行表达式 `expr3`。

如果逻辑表达式为真，返回 `1`，否则返回 `0`。

```
1. $ echo $((3 > 2))
2. 1
3. $ echo $(( (3 > 2) || (4 <= 1) ))
4. 1
```

三元运算符执行一个单独的逻辑测试。它用起来类似于 `if/then/else` 语句。

```
1. $ a=0
2. $ echo $((a<1 ? 1 : 0))
3. 1
4. $ echo $((a>1 ? 1 : 0))
5. 0
```

上面例子中，第一个表达式为真时，返回第二个表达式的值，否则返回第三个表达式的值。

赋值运算

算术表达式 `$((...))` 可以执行赋值运算。

```
1. $ echo $((a=1))
2. 1
3. $ echo $a
4. 1
```

上面例子中，`a=1` 对变量 `a` 进行赋值。这个式子本身也是一个表达式，返回值就是等号右边的值。

`$((...))` 支持的赋值运算符，有以下这些。

- `parameter = value` : 简单赋值。
- `parameter += value` : 等价于 `parameter = parameter + value`。
- `parameter -= value` : 等价于 `parameter = parameter - value`。
- `parameter *= value` : 等价于 `parameter = parameter * value`。
- `parameter /= value` : 等价于 `parameter = parameter / value`。
- `parameter %= value` : 等价于 `parameter = parameter % value`。

- `parameter <= value` : 等价于 `parameter = parameter < value` 。
- `parameter >= value` : 等价于 `parameter = parameter > value` 。
- `parameter &= value` : 等价于 `parameter = parameter & value` 。
- `parameter |= value` : 等价于 `parameter = parameter | value` 。
- `parameter ^= value` : 等价于 `parameter = parameter ^ value` 。

下面是一个例子。

```
1. $ foo=5
2. $ echo $((foo*=2))
3. 10
```

如果在表达式内部赋值，可以放在圆括号中，否则会报错。

```
1. $ echo $(( a<1 ? (a+=1) : (a-=1) ))
```

求值运算

逗号 `,` 在 `$((...))` 内部是求值运算符，执行前后两个表达式，并返回后一个表达式的值。

```
1. $ echo $((foo = 1 + 2, 3 * 4))
2. 12
3. $ echo $foo
4. 3
```

上面例子中，逗号前后两个表达式都会执行，然后返回后一个表达式的值 `12` 。

expr 命令

`expr` 命令支持算术运算，可以不使用 `((...))` 语法。

```
1. $ expr 3 + 2
2. 5
```

`expr` 命令支持变量替换。

```
1. $ foo=3
2. $ expr $foo + 2
3. 5
```

`expr` 命令也不支持非整数参数。

```
1. $ expr 3.5 + 2
2. expr: 非整数参数
```

上面例子中，如果有非整数的运算，`expr` 命令就报错了。

Bash 行操作

简介

Bash 内置了 Readline 库，具有这个库提供的很多“行操作”功能，比如命令的自动补全，可以大大加快操作速度。

这个库默认采用 Emacs 快捷键，也可以改成 Vi 快捷键。

```
1. $ set -o vi
```

下面的命令可以改回 Emacs 快捷键。

```
1. $ set -o emacs
```

如果想永久性更改编辑模式 (Emacs / Vi)，可以将命令写在 `~/.inputrc` 文件，这个文件是 Readline 的配置文件。

```
1. set editing-mode vi
```

本章介绍的快捷键都属于 Emacs 模式。Vi 模式的快捷键，读者可以参考 Vi 编辑器的教程。

Bash 默认开启这个库，但是允许关闭。

```
1. $ bash --noediting
```

上面命令中，`--noediting` 参数关闭了 Readline 库，启动的 Bash 就不带有行操作功能。

光标移动

Readline 提供快速移动光标的快捷键。

- `Ctrl + a`：移到行首。
- `Ctrl + b`：向行首移动一个字符，与左箭头作用相同。
- `Ctrl + e`：移到行尾。
- `Ctrl + f`：向行尾移动一个字符，与右箭头作用相同。
- `Alt + f`：移动到当前单词的词尾。
- `Alt + b`：移动到当前单词的词首。

上面快捷键的 Alt 键，也可以用 ESC 键代替。

清除屏幕

`Ctrl + l` 快捷键可以清除屏幕，即将当前行移到屏幕的第一行，与 `clear` 命令作用相同。

编辑操作

下面的快捷键可以编辑命令行内容。

- `Ctrl + d` : 删除光标位置的字符 (delete)。
- `Ctrl + w` : 删除光标前面的单词。
- `Ctrl + t` : 光标位置的字符与它前面一位的字符交换位置 (transpose)。
- `Alt + t` : 光标位置的词与它前面一位的词交换位置 (transpose)。
- `Alt + l` : 将光标位置至词尾转为小写 (lowercase)。
- `Alt + u` : 将光标位置至词尾转为大写 (uppercase)。

使用 `Ctrl + d` 的时候，如果当前行没有任何字符，会导致退出当前 Shell，所以要小心。

剪切和粘贴快捷键如下。

- `Ctrl + k` : 剪切光标位置到行尾的文本。
- `Ctrl + u` : 剪切光标位置到行首的文本。
- `Alt + d` : 剪切光标位置到词尾的文本。
- `Alt + Backspace` : 剪切光标位置到词首的文本。
- `Ctrl + y` : 在光标位置粘贴文本。

同样地，Alt 键可以用 Esc 键代替。

自动补全

命令输入到一半的时候，可以按一下 Tab 键，Readline 会自动补全命令或路径。比如，输入 `cle`，再按下 Tab 键，Bash 会自动将这个命令补全为 `clear`。

如果符合条件的命令或路径有多个，就需要连续按两次 Tab 键，Bash 会提示所有符合条件的命令或路径。

除了命令或路径，Tab 还可以补全其他值。如果一个值以 `$` 开头，则按下 Tab 键会补全变量；如果以 `~` 开头，则补全用户名；如果以 `@` 开头，则补全主机名 (hostname)，主机名以列在 `/etc/hosts` 文件里面的主机为准。

自动补全相关的快捷键如下。

- Tab: 完成自动补全。
- `Alt + ?` : 列出可能的补全，与连续按两次 Tab 键作用相同。
- `Alt + /` : 尝试文件路径补全。
- `Ctrl + x /` : 先按 `Ctrl + x`，再按 `/`，等同于 `Alt + ?`，列出可能的文件路径补全。
- `Alt + !` : 命令补全。
- `Ctrl + x !` : 先按 `Ctrl + x`，再按 `!`，等同于 `Alt + !`，命令补全。
- `Alt + ~` : 用户名补全。
- `Ctrl + x ~` : 先按 `Ctrl + x`，再按 `~`，等同于 `Alt + ~`，用户名补全。
- `Alt + $` : 变量名补全。
- `Ctrl + x $` : 先按 `Ctrl + x`，再按 `$`，等同于 `Alt + $`，变量名补全。
- `Alt + @` : 主机名补全。

- `Ctrl + x @` : 先按 `Ctrl + x` , 再按 `@` , 等同于 `Alt + @` , 主机名补全。
- `Alt + *` : 在命令行一次性插入所有可能的补全。
- `Alt + Tab` : 尝试用 `.bash_history` 里面以前执行命令, 进行补全。

上面的 `Alt` 键也可以用 `ESC` 键代替。

操作历史

基本用法

Bash 会保留用户的操作历史, 即用户输入的每一条命令都会记录。退出当前 Shell 的时候, Bash 会将用户在当前 Shell 的操作历史写入 `~/.bash_history` 文件, 该文件默认储存500个操作。

环境变量 `HISTFILE` 总是指向这个文件。

```
1. $ echo $HISTFILE
2. /home/me/.bash_history
```

有了操作历史以后, 就可以使用方向键的 `↑` 和 `↓` , 快速浏览上一条和下一条命令。

下面的方法可以快速执行以前执行过的命令。

```
1. $ echo Hello World
2. Hello World
3.
4. $ echo Goodbye
5. Goodbye
6.
7. $ !e
8. echo Goodbye
9. Goodbye
```

上面例子中, `!e` 表示找出操作历史之中, 最近的那一条以 `e` 开头的命令并执行。Bash 会先输出那一条命令 `echo Goodbye` , 然后直接执行。

同理, `!echo` 也会执行最近一条以 `echo` 开头的命令。

```
1. $ !echo
2. echo Goodbye
3. Goodbye
4.
5. $ !echo H
6. echo Goodbye H
7. Goodbye H
8.
9. $ !echo H G
10. echo Goodbye H G
11. Goodbye H G
```

注意，`!string` 语法只会匹配命令，不会匹配参数。所以 `!echo H` 不会执行 `echo Hello World`，而是会执行 `echo Goodbye`，并把参数 `H` 附加在这条命令之后。同理，`!echo H G` 也是等同于 `echo Goodbye` 命令之后附加 `H G`。

最后，按下 `Ctrl + r` 会显示操作历史，可以用方向键上下移动，选择其中要执行的命令。也可以键入命令的首字母，Shell 就会自动在历史文件中，查询并显示匹配的结果。

history 命令

`history` 命令能显示操作历史，即 `.bash_history` 文件的内容。

```
1. $ history
2. ...
3. 498 echo Goodbye
4. 499 ls ~
5. 500 cd
```

使用该命令，而不是直接读取 `.bash_history` 文件的好处是，它会在所有的操作前加上行号，最近的操作在最后面，行号最大。

通过定制环境变量 `HISTTIMEFORMAT`，可以显示每个操作的时间。

```
1. $ export HISTTIMEFORMAT='%F %T '
2. $ history
3. 1 2013-06-09 10:40:12 cat /etc/issue
4. 2 2013-06-09 10:40:12 clear
```

上面代码中，`%F` 相当于 `%Y - %m - %d`，`%T` 相当于 `%H : %M : %S`。

只要设置 `HISTTIMEFORMAT` 这个环境变量，就会在 `.bash_history` 文件保存命令的执行时间戳。如果不设置，就不会保存时间戳。

如果不希望保存本次操作的历史，可以设置环境变量 `HISTSIZE` 等于0。

```
1. export HISTSIZE=0
```

如果 `HISTSIZE=0` 写入用户主目录的 `~/.bashrc` 文件，那么就不会保留该用户的操作历史。如果写入 `/etc/profile`，整个系统都不会保留操作历史。

如果想搜索某个以前执行的命令，可以配合 `grep` 命令搜索操作历史。

```
1. $ history | grep /usr/bin
```

上面命令返回 `.bash_history` 文件里面，那些包含 `/usr/bin` 的命令。

操作历史的每一条记录都有编号。知道了命令的编号以后，可以用 `感叹号 + 编号` 执行该命令。如果想要执行 `.bash_history` 里面的第8条命令，可以像下面这样操作。

```
1. $ !8
```

`history` 命令的 `-c` 参数可以清除操作历史。

```
1. $ history -c
```

相关快捷键

下面是一些与操作历史相关的快捷键。

- `Ctrl + p` : 显示上一个命令, 与向上箭头效果相同 (previous)。
- `Ctrl + n` : 显示下一个命令, 与向下箭头效果相同 (next)。
- `Alt + <` : 显示第一个命令。
- `Alt + >` : 显示最后一个命令, 即当前的命令。
- `Ctrl + o` : 执行历史文件里面的当前条目, 并自动显示下一条命令。这对重复执行某个序列的命令很有帮助。

感叹号 `!` 的快捷键如下。

- `!!` : 执行上一个命令。
- `!n` : 执行历史文件里面行号为 `n` 的命令。
- `!-n` : 执行当前命令之前 `n` 条的命令。
- `!string` : 执行最近一个以指定字符串 `string` 开头的命令。
- `!?string` : 执行最近一条包含字符串 `string` 的命令。
- `^string1^string2` : 执行最近一条包含 `string1` 的命令, 将其替换成 `string2`。

其他快捷键

- `Ctrl + j` : 等同于回车键 (LINEFEED)。
- `Ctrl + m` : 等同于回车键 (CARRIAGE RETURN)。
- `Ctrl + o` : 等同于回车键, 并展示操作历史的下一个命令。
- `Ctrl + v` : 将下一个输入的特殊字符变成字面量, 比如回车变成 `^M`。
- `Ctrl + [` : 等同于 ESC。
- `Alt + .` : 插入上一个命令的最后一个词。
- `Alt + _` : 等同于 `Alt + .`。

上面的 `Alt + .` 快捷键, 对于很长的文件路径, 有时会非常方便。因为 Unix 命令的最后一个参数通常是文件路径。

```
1. $ mkdir foo_bar
2. $ cd #按下 Alt + .
```

上面例子中, 在 `cd` 命令后按下 `Alt + .`, 就会自动插入 `foo_bar`。

目录堆栈

为了方便用户在不同目录之间切换，Bash 提供了目录堆栈功能。

cd -

Bash 可以记忆用户进入过的目录。默认情况下，只记忆前一次所在的目录，`cd -` 命令可以返回前一次的目录。

```
1. # 当前目录是 /path/to/foo
2. $ cd bar
3.
4. # 重新回到 /path/to/foo
5. $ cd -
```

上面例子中，用户原来所在的目录是 `/path/to/foo`，进入子目录 `bar` 以后，使用 `cd -` 可以回到原来的目录。

pushd, popd

如果希望记忆多重目录，可以使用 `pushd` 命令和 `popd` 命令。它们用来操作目录堆栈。

`pushd` 命令的用法类似 `cd` 命令，可以进入指定的目录。

```
1. $ pushd dirname
```

上面命令会进入目录 `dirname`，并将该目录放入堆栈。

第一次使用 `pushd` 命令时，会将当前目录先放入堆栈，然后将所要进入的目录也放入堆栈，位置在前一个记录的上方。以后每次使用 `pushd` 命令，都会将所要进入的目录，放在堆栈的顶部。

`popd` 命令不带有参数时，会移除堆栈的顶部记录，并进入新的堆栈顶部目录（即原来的第二条目录）。

下面是一个例子。

```
1. # 当前处在主目录，堆栈为空
2. $ pwd
3. /home/me
4.
5. # 进入 /home/me/foo
6. # 当前堆栈为 /home/me/foo /home/me
7. $ pushd ~/foo
8.
9. # 进入 /etc
10. # 当前堆栈为 /etc /home/me/foo /home/me
11. $ pushd /etc
12.
13. # 进入 /home/me/foo
```



```

14. # 当前堆栈为 /home/me/foo /home/me
15. $ popd
16.
17. # 进入 /home/me
18. # 当前堆栈为 /home/me
19. $ popd
20.
21. # 目录不变，当前堆栈为空
22. $ popd

```

这两个命令的参数如下。

（1）-n 参数

`-n` 的参数表示仅操作堆栈，不改变目录。

```
1. $ popd -n
```

上面的命令仅删除堆栈顶部的记录，不改变目录，执行完成后还停留在当前目录。

（2）整数参数

这两个命令还可以接受一个整数作为参数，该整数表示堆栈中指定位置的记录（从0开始），作为操作对象。这时不会切换目录。

```

1. # 从栈顶算起的3号目录（从0开始），移动到栈顶
2. $ pushd +3
3.
4. # 从栈底算起的3号目录（从0开始），移动到栈顶
5. $ pushd -3
6.
7. # 删除从栈顶算起的3号目录（从0开始）
8. $ popd +3
9.
10. # 删除从栈底算起的3号目录（从0开始）
11. $ popd -3

```

上面例子的整数编号都是从0开始计算，`popd +0` 是删除第一个目录，`popd +1` 是删除第二个，`popd -0` 是删除最后一个目录，`popd -1` 是删除倒数第二个。

（3）目录参数

`pushd` 可以接受一个目录作为参数，表示将该目录放到堆栈顶部，并进入该目录。

```
1. $ pushd dir
```

`popd` 没有这个参数。

dirs 命令

`dirs` 命令可以显示目录堆栈的内容，一般用来查看 `pushd` 和 `popd` 操作后的结果。

```
1. $ dirs
```

它有以下参数。

- `-c` : 清空目录栈。
- `-l` : 用户主目录不显示波浪号前缀，而打印完整的目录。
- `-p` : 每行一个条目打印目录栈，默认是打印在一行。
- `-v` : 每行一个条目，每个条目之前显示位置编号（从0开始）。
- `+N` : `N` 为整数，表示显示堆顶算起的第 `N` 个目录，从零开始。
- `-N` : `N` 为整数，表示显示堆底算起的第 `N` 个目录，从零开始。

Bash 脚本入门

脚本（script）就是包含一系列命令的一个文本文件。Shell 读取这个文件，依次执行里面的所有命令，就好像这些命令直接输入到命令行一样。所有能够在命令行完成的任务，都能够用脚本完成。

脚本的好处是可以重复使用，也可以指定在特定场合自动调用，比如系统启动或关闭时自动执行脚本。

Shebang 行

脚本的第一行通常是指定解释器，即这个脚本必须通过什么解释器执行。这一行以 `#!` 字符开头，这个字符称为 Shebang，所以这一行就叫做 Shebang 行。

`#!` 后面就是脚本解释器的位置，Bash 脚本的解释器一般是 `/bin/sh` 或 `/bin/bash`。

1. `#!/bin/sh`
2. `# 或者`
3. `#!/bin/bash`

`#!` 与脚本解释器之间有没有空格，都是可以的。

如果 Bash 解释器不放在目录 `/bin`，脚本就无法执行了。为了保险，可以写成下面这样。

1. `#!/usr/bin/env bash`

上面命令使用 `env` 命令（这个命令总是在 `/usr/bin` 目录），返回 Bash 可执行文件的位置。`env` 命令的详细介绍，请看后文。

Shebang 行不是必需的，但是建议加上这行。如果缺少该行，就需要手动将脚本传给解释器。举例来说，脚本是 `script.sh`，有 Shebang 行的时候，可以直接调用执行。

1. `$./script.sh`

上面例子中，`script.sh` 是脚本文件名。脚本通常使用 `.sh` 后缀名，不过这不是必需的。

如果没有 Shebang 行，就只能手动将脚本传给解释器来执行。

1. `$ /bin/sh ./script.sh`
2. `# 或者`
3. `$ bash ./script.sh`

执行权限和路径

前面说过，只要指定了 Shebang 行的脚本，可以直接执行。这有一个前提条件，就是脚本需要有执行权限。可以使用下面的命令，赋予脚本执行权限。

```
1. # 给所有用户执行权限
2. $ chmod +x script.sh
3.
4. # 给所有用户读权限和执行权限
5. $ chmod +rx script.sh
6. # 或者
7. $ chmod 755 script.sh
8.
9. # 只给脚本所有者读权限和执行权限
10. $ chmod u+rx script.sh
```

脚本的权限通常设为 `755`（拥有者有所有权限，其他人有读和执行权限）或者 `700`（只有拥有者可以执行）。

除了执行权限，脚本调用时，一般需要指定脚本的路径（比如 `path/script.sh`）。如果将脚本放在环境变量 `$PATH` 指定的目录中，就不需要指定路径了。因为 Bash 会自动到这些目录中，寻找是否存在同名的可执行文件。

建议在主目录新建一个 `~/bin` 子目录，专门存放可执行脚本，然后把 `~/bin` 加入 `$PATH`。

```
1. export PATH=$PATH:~/bin
```

上面命令改变环境变量 `$PATH`，将 `~/bin` 添加到 `$PATH` 的末尾。可以将这一行加到 `~/.bashrc` 文件里面，然后重新加载一次 `.bashrc`，这个配置就可以生效了。

```
1. $ source ~/.bashrc
```

以后不管在什么目录，直接输入脚本文件名，脚本就会执行。

```
1. $ script.sh
```

上面命令没有指定脚本路径，因为 `script.sh` 在 `$PATH` 指定的目录中。

env 命令

`env` 命令总是指向 `/usr/bin/env` 文件，或者说，这个二进制文件总是在目录 `/usr/bin`。

`#!/usr/bin/env NAME` 这个语法的意思是，让 Shell 查找 `$PATH` 环境变量里面第一个匹配的 `NAME`。如果你不知道某个命令的具体路径，或者希望兼容其他用户的机器，这样的写法就很有用。

`/usr/bin/env bash` 的意思就是，返回 `bash` 可执行文件的位置，前提是 `bash` 的路径是在 `$PATH` 里面。其他脚本文件也可以使用这个命令。比如 Node.js 脚本的 Shebang 行，可以写成下面这样。

```
1. #!/usr/bin/env node
```

`env` 命令的参数如下。

- `-i`，`--ignore-environment`：不带环境变量启动。

- `-u` , `--unset=NAME` : 从环境变量中删除一个变量。
- `--help` : 显示帮助。
- `--version` : 输出版本信息。

下面是一个例子，新建一个不带任何环境变量的 Shell。

```
1. $ env -i /bin/sh
```

注释

Bash 脚本中，`#` 表示注释，可以放在行首，也可以放在行尾。

```
1. # 本行是注释
2. echo 'Hello World!'
3.
4. echo 'Hello World!' # 井号后面的部分也是注释
```

建议在脚本开头，使用注释说明当前脚本的作用，这样有利于日后的维护。

脚本参数

调用脚本的时候，脚本文件名后面可以带有参数。

```
1. $ script.sh word1 word2 word3
```

上面例子中，`script.sh` 是一个脚本文件，`word1`、`word2` 和 `word3` 是三个参数。

脚本文件内部，可以使用特殊变量，引用这些参数。

- `$0` : 脚本文件名，即 `script.sh`。
- `$1` ~ `$9` : 对应脚本的第一个参数到第九个参数。
- `$#` : 参数的总数。
- `$@` : 全部的参数，参数之间使用空格分隔。
- `$*` : 全部的参数，参数之间使用变量 `$IFS` 值的第一个字符分隔，默认为空格，但是可以自定义。

如果脚本的参数多于9个，那么第10个参数可以用 `${10}` 的形式引用，以此类推。

注意，如果命令是 `command -o foo bar`，那么 `-o` 是 `$1`，`foo` 是 `$2`，`bar` 是 `$3`。

下面是一个脚本内部读取命令行参数的例子。

```
1. #!/bin/bash
2. # script.sh
3.
4. echo "全部参数：" $@
5. echo "命令行参数数量：" $#
6. echo '$0 = ' $0
```

```
7. echo '$1 = ' $1
8. echo '$2 = ' $2
9. echo '$3 = ' $3
```

执行结果如下。

```
1. $ ./script.sh a b c
2. 全部参数 : a b c
3. 命令行参数数量 : 3
4. $0 = script.sh
5. $1 = a
6. $2 = b
7. $3 = c
```

用户可以输入任意数量的参数，利用 `for` 循环，可以读取每一个参数。

```
1. #!/bin/bash
2.
3. for i in "$@"; do
4.     echo $i
5. done
```

上面例子中，`$@` 返回一个全部参数的列表，然后使用 `for` 循环遍历。

如果多个参数放在双引号里面，视为一个参数。

```
1. $ ./script.sh "a b"
```

上面例子中，Bash 会认为 `"a b"` 是一个参数，`$1` 会返回 `a b`。注意，返回时不包括双引号。

shift 命令

`shift` 命令可以改变脚本参数，每次执行都会移除脚本当前的第一个参数（`$1`），使得后面的参数向前一位，即 `$2` 变成 `$1`、`$3` 变成 `$2`、`$4` 变成 `$3`，以此类推。

`while` 循环结合 `shift` 命令，也可以读取每一个参数。

```
1. #!/bin/bash
2.
3. echo "一共输入了 $# 个参数"
4.
5. while [ "$1" != "" ]; do
6.     echo "剩下 $# 个参数"
7.     echo "参数 : $1"
8.     shift
9. done
```

上面例子中，`shift` 命令每次移除当前第一个参数，从而通过 `while` 循环遍历所有参数。

`shift` 命令可以接受一个整数作为参数，指定所要移除的参数个数，默认为 `1`。

```
1. shift 3
```

上面的命令移除前三个参数，原来的 `$4` 变成 `$1`。

getopts 命令

`getopts` 命令用在脚本内部，可以解析复杂的脚本命令行参数，通常与 `while` 循环一起使用，取出脚本所有的带有前置连词线（`-`）的参数。

```
1. getopts optstring name
```

它带有两个参数。第一个参数 `optstring` 是字符串，给出脚本所有的连词线参数。比如，某个脚本可以有三个配置项参数 `-l`、`-h`、`-a`，其中只有 `-a` 可以带有参数值，而 `-l` 和 `-h` 是开关参数，那么 `getopts` 的第一个参数写成 `lha:`，顺序不重要。注意，`a` 后面有一个冒号，表示该参数带有参数值，`getopts` 规定带有参数值的配置项参数，后面必须带有一个冒号（`:`）。`getopts` 的第二个参数 `name` 是一个变量名，用来保存当前取到的配置项参数，即 `l`、`h` 或 `a`。

下面是一个例子。

```
1. while getopts 'lha:' OPTION; do
2.     case "$OPTION" in
3.         l)
4.             echo "linuxconfig"
5.             ;;
6.
7.         h)
8.             echo "h stands for h"
9.             ;;
10.
11.        a)
12.            avalue="$OPTARG"
13.            echo "The value provided is $OPTARG"
14.            ;;
15.        ?)
16.            echo "script usage: $(basename $0) [-l] [-h] [-a somevalue]" >&2
17.            exit 1
18.            ;;
19.        esac
20.    done
21. shift "$(($OPTIND - 1))"
```

上面例子中，`while` 循环不断执行 `getopts 'lha:' OPTION` 命令，每次执行就会读取一个连词线参数（以及对应的参数值），然后进入循环体。变量 `OPTION` 保存的是，当前处理的那一个连词线参数（即 `l`、`h` 或 `a`）。如果用户输入了没有指定的参数（比如 `-x`），那么 `OPTION` 等于 `?`。循环体内使用 `case` 判断，处理这四种不同的情况。

如果某个连词线参数带有参数值，比如 `-a foo`，那么处理 `a` 参数的时候，环境变量 `$OPTARG` 保存的就是参数值。

注意，只要遇到不带连词线的参数，`getopts` 就会执行失败，从而退出 `while` 循环。比如，`getopts` 可以解析 `command -l foo`，但不可以解析 `command foo -l`。另外，多个连词线参数写在一起的形式，比如 `command -lh`，`getopts` 也可以正确处理。

变量 `$OPTIND` 在 `getopts` 开始执行前是 `1`，然后每次执行就会加 `1`。等到退出 `while` 循环，就意味着连词线参数全部处理完毕。这时，`$OPTIND - 1` 就是已经处理的连词线参数个数，使用 `shift` 命令将这些参数移除，保证后面的代码可以用 `$1`、`$2` 等处理命令的主参数。

配置项参数终止符 `--`

变量当作命令的参数时，有时希望指定变量只能作为实体参数，不能当作配置项参数，这时可以使用配置项参数终止符 `--`。

```
1. $ myPath=~/.docs"
2. $ ls -- $myPath
```

上面例子中，`--` 强制变量 `$myPath` 只能当作实体参数（即路径名）解释。

如果变量不是路径名，就会报错。

```
1. $ myPath="-l"
2. $ ls -- $myPath
3. ls: 无法访问 '-l': 没有那个文件或目录
```

上面例子中，变量 `myPath` 的值为 `-l`，不是路径。但是，`--` 强制 `$myPath` 只能作为路径解释，导致报错“不存在该路径”。

exit 命令

`exit` 命令用于终止当前脚本的执行，并向 Shell 返回一个退出值。

```
1. $ exit
```

上面命令中止当前脚本，将最后一条命令的退出状态，作为整个脚本的退出状态。

`exit` 命令后面可以跟参数，该参数就是退出状态。

```
1. # 退出值为0（成功）
2. $ exit 0
3.
4. # 退出值为1（失败）
5. $ exit 1
```


退出时，脚本会返回一个退出值。脚本的退出值，`0` 表示正常，`1` 表示发生错误，`2` 表示用法不对，`126` 表示不是可执行脚本，`127` 表示命令没有发现。如果脚本被信号 `N` 终止，则退出值为 `128 + N`。简单来说，只要退出值非0，就认为执行出错。

下面是一个例子。

```
1. if [ $(id -u) != "0" ]; then
2.     echo "根用户才能执行当前脚本"
3.     exit 1
4. fi
```

上面的例子中，`id -u` 命令返回用户的 ID，一旦用户的 ID 不等于 `0`（根用户的 ID），脚本就会退出，并且退出码为 `1`，表示运行失败。

`exit` 与 `return` 命令的差别是，`return` 命令是函数的退出，并返回一个值给调用者，脚本依然执行。`exit` 是整个脚本的退出，如果在函数之中调用 `exit`，则退出函数，并终止脚本执行。

命令执行结果

命令执行结束后，会有一个返回值。`0` 表示执行成功，非 `0`（通常是 `1`）表示执行失败。环境变量 `$?` 可以读取前一个命令的返回值。

利用这一点，可以在脚本中对命令执行结果进行判断。

```
1. cd $some_directory
2. if [ "$?" = "0" ]; then
3.     rm *
4. else
5.     echo "无法切换目录!" 1>&2
6.     exit 1
7. fi
```

上面例子中，`cd $some_directory` 这个命令如果执行成功（返回值等于 `0`），就删除该目录里面的文件，否则退出脚本，整个脚本的返回值变为 `1`，表示执行失败。

由于 `if` 可以直接判断命令的执行结果，执行相应的操作，上面的脚本可以改写成下面的样子。

```
1. if cd $some_directory; then
2.     rm *
3. else
4.     echo "Could not change directory! Aborting." 1>&2
5.     exit 1
6. fi
```

更简洁的写法是利用两个逻辑运算符 `&&`（且）和 `||`（或）。

```
1. # 第一步执行成功，才会执行第二步
2. cd $some_directory && rm *
```

```
3.  
4. # 第一步执行失败, 才会执行第二步  
5. cd $some_directory || exit 1
```

source 命令

`source` 命令用于执行一个脚本，通常用于重新加载一个配置文件。

```
1. $ source .bashrc
```

`source` 命令最大的特点是在当前 Shell 执行脚本，不像直接执行脚本时，会新建一个子 Shell。所以，`source` 命令执行脚本时，不需要 `export` 变量。

```
1. #!/bin/bash  
2. # test.sh  
3. echo $foo
```

上面脚本输出 `$foo` 变量的值。

```
1. # 当前 Shell 新建一个变量 foo  
2. $ foo=1  
3.  
4. # 打印输出 1  
5. $ source test.sh  
6. 1  
7.  
8. # 打印输出空字符串  
9. $ bash test.sh
```

上面例子中，当前 Shell 的变量 `foo` 并没有 `export`，所以直接执行无法读取，但是 `source` 执行可以读取。

`source` 命令的另一个用途，是在脚本内部加载外部库。

```
1. #!/bin/bash  
2.  
3. source ./lib.sh  
4.  
5. function_from_lib
```

上面脚本在内部使用 `source` 命令加载了一个外部库，然后就可以在脚本里面，使用这个外部库定义的函数。

`source` 有一个简写形式，可以使用一个点（`.`）来表示。

```
1. $ . .bashrc
```

别名，alias 命令

`alias` 命令用来为一个命令指定别名，这样更便于记忆。下面是 `alias` 的格式。

```
1. alias NAME=DEFINITION
```

上面命令中，`NAME` 是别名的名称，`DEFINITION` 是别名对应的原始命令。注意，等号两侧不能有空格，否则会报错。

一个常见的例子是为 `grep` 命令起一个 `search` 的别名。

```
1. alias search=grep
```

`alias` 也可以用来为长命令指定一个更短的别名。下面是通过别名定义一个 `today` 的命令。

```
1. $ alias today='date +%A, %B %-d, %Y'
2. $ today
3. 星期一, 一月 6, 2020
```

有时为了防止误删除文件，可以指定 `rm` 命令的别名。

```
1. $ alias rm='rm -i'
```

上面命令指定 `rm` 命令是 `rm -i`，每次删除文件之前，都会让用户确认。

`alias` 定义的别名也可以接受参数，参数会直接传入原始命令。

```
1. $ alias echo='echo It says: '
2. $ echo hello world
3. It says: hello world
```

上面例子中，别名定义了 `echo` 命令的前两个参数，等同于修改了 `echo` 命令的默认行为。

指定别名以后，就可以像使用其他命令一样使用别名。一般来说，都会把常用的别名写在 `~/.bashrc` 的末尾。另外，只能为命令定义别名，为其他部分（比如很长的路径）定义别名是无效的。

直接调用 `alias` 命令，可以显示所有别名。

```
1. $ alias
```

`unalias` 命令可以解除别名。

```
1. $ unalias lt
```

参考链接

- [How to use getopts to parse a script options](#), Egidio Docile

read 命令

用法

有时，脚本需要在执行过程中，由用户提供一部分数据，这时可以使用 `read` 命令。它将用户的输入存入一个变量，方便后面的代码使用。用户按下回车键，就表示输入结束。

`read` 命令的格式如下。

```
1. read [-options] [variable...]
```

上面语法中，`options` 是参数选项，`variable` 是用来保存输入数值的一个或多个变量名。如果没有提供变量名，环境变量 `REPLY` 会包含用户输入的一整行数据。

下面是一个例子 `demo.sh`。

```
1. #!/bin/bash
2.
3. echo -n "输入一些文本 > "
4. read text
5. echo "你的输入：$text"
```

上面例子中，先显示一行提示文本，然后会等待用户输入文本。用户输入的文本，存入变量 `text`，在下一行显示出来。

```
1. $ bash demo.sh
2. 输入一些文本 > 你好，世界
3. 你的输入：你好，世界
```

`read` 可以接受用户输入的多个值。

```
1. #!/bin/bash
2. echo Please, enter your firstname and lastname
3. read FN LN
4. echo "Hi! $LN, $FN !"
```

上面例子中，`read` 根据用户的输入，同时为两个变量赋值。

如果用户的输入项少于 `read` 命令给出的变量数目，那么额外的变量值为空。如果用户的输入项多于定义的变量，那么多余的输入项会包含到最后一个变量中。

如果 `read` 命令之后没有定义变量名，那么环境变量 `REPLY` 会包含所有的输入。

```
1. #!/bin/bash
2. # read-single: read multiple values into default variable
```

```

3. echo -n "Enter one or more values > "
4. read
5. echo "REPLY = '$REPLY'"

```

上面脚本的运行结果如下。

```

1. $ read-single
2. Enter one or more values > a b c d
3. REPLY = 'a b c d'

```

`read` 命令除了读取键盘输入，可以用来读取文件。

```

1. while read myline
2. do
3.     echo "$myline"
4. done < $filename

```

上面的例子通过 `read` 命令，读取一个文件的内容。`done` 命令后面的定向符 `<`，将文件导向 `read` 命令，每次读取一行，存入变量 `myline`，直到文件读取完毕。

参数

`read` 命令的参数如下。

(1) -t 参数

`read` 命令的 `-t` 参数，设置了超时的秒数。如果超过了指定时间，用户仍然没有输入，脚本将放弃等待，继续向下执行。

```

1. #!/bin/bash
2.
3. echo -n "输入一些文本 > "
4. if read -t 3 response; then
5.     echo "用户已经输入了"
6. else
7.     echo "用户没有输入"
8. fi

```

上面例子中，输入命令会等待3秒，如果用户超过这个时间没有输入，这个命令就会执行失败。`if` 根据命令的返回值，转入 `else` 代码块，继续往下执行。

环境变量 `TMOUT` 也可以起到同样作用，指定 `read` 命令等待用户输入的时间（单位为秒）。

```

1. $ TMOUT=3
2. $ read response

```

上面例子也是等待3秒，如果用户还没有输入，就会超时。

(2) -p 参数

-p 参数指定用户输入的提示信息。

```

1. read -p "Enter one or more values > "
2. echo "REPLY = '$REPLY'"

```

上面例子中，先显示 `Enter one or more values >`，再接受用户的输入。

(3) -a 参数

-a 参数把用户的输入赋值给一个数组，从零号位置开始。

```

1. $ read -a people
2. alice duchess dodo
3. $ echo ${people[2]}
4. dodo

```

上面例子中，用户输入被赋值给一个数组 `people`，这个数组的2号成员就是 `dodo`。

(4) -n 参数

-n 参数指定只读取若干个字符作为变量值，而不是整行读取。

```

1. $ read -n 3 letter
2. abcdefghij
3. $ echo $letter
4. abc

```

上面例子中，变量 `letter` 只包含3个字母。

(5) -e 参数

-e 参数允许用户输入的时候，使用 `readline` 库提供的快捷键，比如自动补全。具体的快捷键可以参阅《行操作》一章。

```

1. #!/bin/bash
2.
3. echo Please input the path to the file:
4.
5. read -e fileName
6.
7. echo $fileName

```

上面例子中，`read` 命令接受用户输入的文件名。这时，用户可能想使用 Tab 键的文件名“自动补全”功能，但是 `read` 命令的输入默认不支持 `readline` 库的功能。`-e` 参数就可以允许用户使用自动补全。

(6) 其他参数

- **-d delimiter**：定义字符串 `delimiter` 的第一个字符作为用户输入的结束，而不是一个换行符。

- `-r` : raw 模式，表示不把用户输入的反斜杠字符解释为转义字符。
- `-s` : 使得用户的输入不显示在屏幕上，这常常用于输入密码或保密信息。
- `-u fd` : 使用文件描述符 `fd` 作为输入。

IFS 变量

`read` 命令读取的值，默认是以空格分隔。可以通过自定义环境变量 `IFS`（内部字段分隔符，Internal Field Separator 的缩写），修改分隔标志。

`IFS` 的默认值是空格、Tab 符号、换行符号，通常取第一个（即空格）。

如果把 `IFS` 定义成冒号（`:`）或分号（`;`），就可以分隔以这两个符号分隔的值，这对读取文件很有用。

```
1. #!/bin/bash
2. # read-ifs: read fields from a file
3.
4. FILE=/etc/passwd
5.
6. read -p "Enter a username > " user_name
7. file_info="$(grep "^$user_name:" $FILE)"
8.
9. if [ -n "$file_info" ]; then
10.     IFS=":" read user pw uid gid name home shell <<< "$file_info"
11.     echo "User = '$user'"
12.     echo "UID = '$uid'"
13.     echo "GID = '$gid'"
14.     echo "Full Name = '$name'"
15.     echo "Home Dir. = '$home'"
16.     echo "Shell = '$shell'"
17. else
18.     echo "No such user '$user_name'" >&2
19.     exit 1
20. fi
```

上面例子中，`IFS` 设为冒号，然后用来分解 `/etc/passwd` 文件的一行。`IFS` 的赋值命令和 `read` 命令写在一行，这样的话，`IFS` 的改变仅对后面的命令生效，该命令执行后 `IFS` 会自动恢复原来的值。如果不写在一行，就要采用下面的写法。

```
1. OLD_IFS="$IFS"
2. IFS=":"
3. read user pw uid gid name home shell <<< "$file_info"
4. IFS="$OLD_IFS"
```

另外，上面例子中，`<<<` 是 Here 字符串，用于将变量值转为标准输入，因为 `read` 命令只能解析标准输入。

如果 `IFS` 设为空字符串，就等同于将整行读入一个变量。

```
1. #!/bin/bash
2. input="/path/to/txt/file"
```

```
3. while IFS= read -r line
4. do
5.     echo "$line"
6. done < "$input"
```

上面的命令可以逐行读取文件，每一行存入变量 `line` ，打印出来以后再读取下一行。

条件判断

本章介绍 Bash 脚本的条件判断语法。

if 结构

`if` 是最常用的条件判断结构，只有符合给定条件时，才会执行指定的命令。它的语法如下。

```
1. if commands; then
2.     commands
3. [elif commands; then
4.     commands...]
5. [else
6.     commands]
7. fi
```

这个命令分成三个部分：`if`、`elif` 和 `else`。其中，后两个部分是可选的。

`if` 关键字后面是主要的判断条件，`elif` 用来添加在主条件不成立时的其他判断条件，`else` 则是所有条件都不成立时要执行的部分。

```
1. if test $USER = "foo"; then
2.     echo "Hello foo."
3. else
4.     echo "You are not foo."
5. fi
```

上面的例子中，判断条件是环境变量 `$USER` 是否等于 `foo`，如果等于就输出 `Hello foo.`，否则输出其他内容。

`if` 和 `then` 写在同一行时，需要分号分隔。分号是 Bash 的命令分隔符。它们也可以写成两行，这时不需要分号。

```
1. if true
2. then
3.     echo 'hello world'
4. fi
5.
6. if false
7. then
8.     echo 'it is false' # 本行不会执行
9. fi
```

上面的例子中，`true` 和 `false` 是两个特殊命令，前者代表操作成功，后者代表操作失败。`if true` 意味着命令部分总是会执行，`if false` 意味着命令部分永远不会执行。

除了多行的写法，`if` 结构也可以写成单行。

```
1. $ if true; then echo 'hello world'; fi
2. hello world
3.
4. $ if false; then echo "It's true."; fi
```

注意，`if` 关键字后面也可以是一条命令，该条命令执行成功（返回值 `0` ），就意味着判断条件成立。

```
1. $ if echo 'hi'; then echo 'hello world'; fi
2. hi
3. hello world
```

上面命令中，`if` 后面是一条命令 `echo 'hi'` 。该命令会执行，如果返回值是 `0` ，则执行 `then` 的部分。

`if` 后面可以跟任意数量的命令。这时，所有命令都会执行，但是判断真伪只看最后一个命令，即使前面所有命令都失败，只要最后一个命令返回 `0` ，就会执行 `then` 的部分。

```
1. $ if false; true; then echo 'hello world'; fi
2. hello world
```

上面例子中，`if` 后面有两条命令（`false;true;` ），第二条命令（`true` ）决定了 `then` 的部分是否会执行。

`elif` 部分可以有多个。

```
1. #!/bin/bash
2.
3. echo -n "输入一个1到3之间的数字（包含两端）> "
4. read character
5. if [ "$character" = "1" ]; then
6.     echo 1
7. elif [ "$character" = "2" ]; then
8.     echo 2
9. elif [ "$character" = "3" ]; then
10.    echo 3
11. else
12.    echo 输入不符合要求
13. fi
```

上面例子中，如果用户输入 `3` ，就会连续判断3次。

test 命令

`if` 结构的判断条件，一般使用 `test` 命令，有三种形式。

```
1. # 写法一
2. test expression
3.
4. # 写法二
5. [ expression ]
```

```

6.
7. # 写法三
8. [[ expression ]]

```

上面三种形式是等价的，但是第三种形式还支持正则判断，前两种不支持。

上面的 `expression` 是一个表达式。这个表达式为真，`test` 命令执行成功（返回值为 `0`）；表达式为伪，`test` 命令执行失败（返回值为 `1`）。注意，第二种和第三种写法，`[` 和 `]` 与内部的表达式之间必须有空格。

```

1. $ test -f /etc/hosts
2. $ echo $?
3. 0
4.
5. $ [ -f /etc/hosts ]
6. $ echo $?
7. 0

```

上面的例子中，`test` 命令采用两种写法，判断 `/etc/hosts` 文件是否存在，这两种写法是等价的。命令执行后，返回值为 `0`，表示该文件确实存在。

实际上，`[` 这个字符是 `test` 命令的一种简写形式，可以看作是一个独立的命令，这解释了为什么它后面必须有空格。

下面把 `test` 命令的三种形式，用在 `if` 结构中，判断一个文件是否存在。

```

1. # 写法一
2. if test -e /tmp/foo.txt ; then
3.     echo "Found foo.txt"
4. fi
5.
6. # 写法二
7. if [ -e /tmp/foo.txt ] ; then
8.     echo "Found foo.txt"
9. fi
10.
11. # 写法三
12. if [[ -e /tmp/foo.txt ]] ; then
13.     echo "Found foo.txt"
14. fi

```

判断表达式

`if` 关键字后面，跟的是一个命令。这个命令可以是 `test` 命令，也可以是其他命令。命令的返回值为 `0` 表示判断成立，否则表示不成立。因为这些命令主要是为了得到返回值，所以可以视为表达式。

常用的判断表达式有下面这些。

文件判断

以下表达式用来判断文件状态。

- `[-a file]` : 如果 file 存在, 则为 `true` 。
- `[-b file]` : 如果 file 存在并且是一个块 (设备) 文件, 则为 `true` 。
- `[-c file]` : 如果 file 存在并且是一个字符 (设备) 文件, 则为 `true` 。
- `[-d file]` : 如果 file 存在并且是一个目录, 则为 `true` 。
- `[-e file]` : 如果 file 存在, 则为 `true` 。
- `[-f file]` : 如果 file 存在并且是一个普通文件, 则为 `true` 。
- `[-g file]` : 如果 file 存在并且设置了组 ID, 则为 `true` 。
- `[-G file]` : 如果 file 存在并且属于有效的组 ID, 则为 `true` 。
- `[-h file]` : 如果 file 存在并且是符号链接, 则为 `true` 。
- `[-k file]` : 如果 file 存在并且设置了它的“sticky bit”, 则为 `true` 。
- `[-L file]` : 如果 file 存在并且是一个符号链接, 则为 `true` 。
- `[-N file]` : 如果 file 存在并且自上次读取后已被修改, 则为 `true` 。
- `[-O file]` : 如果 file 存在并且属于有效的用户 ID, 则为 `true` 。
- `[-p file]` : 如果 file 存在并且是一个命名管道, 则为 `true` 。
- `[-r file]` : 如果 file 存在并且可读 (当前用户有可读权限), 则为 `true` 。
- `[-s file]` : 如果 file 存在且其长度大于零, 则为 `true` 。
- `[-S file]` : 如果 file 存在且是一个网络 socket, 则为 `true` 。
- `[-t fd]` : 如果 fd 是一个文件描述符, 并且重定向到终端, 则为 `true` 。
- `[-u file]` : 如果 file 存在并且设置了 setuid 位, 则为 `true` 。
- `[-w file]` : 如果 file 存在并且可写 (当前用户拥有可写权限), 则为 `true` 。
- `[-x file]` : 如果 file 存在并且可执行 (有效用户有执行 / 搜索权限), 则为 `true` 。
- `[file1 -nt file2]` : 如果 FILE1 比 FILE2 的更新时间最近, 或者 FILE1 存在而 FILE2 不存在, 则为 `true` 。
- `[file1 -ot file2]` : 如果 FILE1 比 FILE2 的更新时间更旧, 或者 FILE2 存在而 FILE1 不存在, 则为 `true` 。
- `[FILE1 -ef FILE2]` : 如果 FILE1 和 FILE2 引用相同的设备和 inode 编号, 则为 `true` 。

下面是一个示例。

```
1. #!/bin/bash
2.
3. FILE=~/.bashrc
4.
5. if [ -e "$FILE" ]; then
6.     if [ -f "$FILE" ]; then
7.         echo "$FILE is a regular file."
8.     fi
9.     if [ -d "$FILE" ]; then
10.        echo "$FILE is a directory."
11.    fi
12.    if [ -r "$FILE" ]; then
13.        echo "$FILE is readable."
14.    fi
15.    if [ -w "$FILE" ]; then
```

```

16.     echo "$FILE is writable."
17. fi
18. if [ -x "$FILE" ]; then
19.     echo "$FILE is executable/searchable."
20. fi
21. else
22.     echo "$FILE does not exist"
23.     exit 1
24. fi

```

上面代码中，`$FILE` 要放在双引号之中。这样可以防止 `$FILE` 为空，因为这时 `[-e]` 会判断为真。而放在双引号之中，返回的就总是一个空字符串，`[-e ""]` 会判断为伪。

字符串判断

以下表达式用来判断字符串。

- `[string]`：如果 `string` 不为空（长度大于0），则判断为真。
- `[-n string]`：如果字符串 `string` 的长度大于零，则判断为真。
- `[-z string]`：如果字符串 `string` 的长度为零，则判断为真。
- `[string1 = string2]`：如果 `string1` 和 `string2` 相同，则判断为真。
- `[string1 == string2]` 等同于 `[string1 = string2]`。
- `[string1 != string2]`：如果 `string1` 和 `string2` 不相同，则判断为真。
- `[string1 '>' string2]`：如果按照字典顺序 `string1` 排列在 `string2` 之后，则判断为真。
- `[string1 '<' string2]`：如果按照字典顺序 `string1` 排列在 `string2` 之前，则判断为真。

注意，`test` 命令内部的 `>` 和 `<`，必须用引号引起来（或者用反斜杠转义）。否则，它们会被 `shell` 解释为重定向操作符。

下面是一个示例。

```

1. #!/bin/bash
2.
3. ANSWER=maybe
4.
5. if [ -z "$ANSWER" ]; then
6.     echo "There is no answer." >&2
7.     exit 1
8. fi
9. if [ "$ANSWER" = "yes" ]; then
10.    echo "The answer is YES."
11. elif [ "$ANSWER" = "no" ]; then
12.    echo "The answer is NO."
13. elif [ "$ANSWER" = "maybe" ]; then
14.    echo "The answer is MAYBE."
15. else
16.    echo "The answer is UNKNOWN."
17. fi

```

上面代码中，首先确定 `$ANSWER` 字符串是否为空。如果为空，就终止脚本，并把退出状态设为 `1`。注意，这里

的 `echo` 命令把错误信息 `There is no answer.` 重定向到标准错误，这是处理错误信息的常用方法。如果 `$ANSWER` 字符串不为空，就判断它的值是否等于 `yes`、`no` 或者 `maybe`。

注意，字符串判断时，变量要放在双引号之中，比如 `[-n "$COUNT"]`，否则变量替换成字符串以后，`test` 命令可能会报错，提示参数过多。另外，如果不放在双引号之中，变量为空时，命令会变成 `[-n]`，这时会判断为真。如果放在双引号之中，`[-n ""]` 就判断为伪。

整数判断

下面的表达式用于判断整数。

- `[integer1 -eq integer2]`：如果 `integer1` 等于 `integer2`，则为 `true`。
- `[integer1 -ne integer2]`：如果 `integer1` 不等于 `integer2`，则为 `true`。
- `[integer1 -le integer2]`：如果 `integer1` 小于或等于 `integer2`，则为 `true`。
- `[integer1 -lt integer2]`：如果 `integer1` 小于 `integer2`，则为 `true`。
- `[integer1 -ge integer2]`：如果 `integer1` 大于或等于 `integer2`，则为 `true`。
- `[integer1 -gt integer2]`：如果 `integer1` 大于 `integer2`，则为 `true`。

下面是一个用法的例子。

```
1. #!/bin/bash
2.
3. INT=-5
4.
5. if [ -z "$INT" ]; then
6.     echo "INT is empty." >&2
7.     exit 1
8. fi
9. if [ $INT -eq 0 ]; then
10.    echo "INT is zero."
11. else
12.    if [ $INT -lt 0 ]; then
13.        echo "INT is negative."
14.    else
15.        echo "INT is positive."
16.    fi
17.    if [ $((INT % 2)) -eq 0 ]; then
18.        echo "INT is even."
19.    else
20.        echo "INT is odd."
21.    fi
22. fi
```

上面例子中，先判断变量 `$INT` 是否为空，然后判断是否为 `0`，接着判断正负，最后通过求余数判断奇偶。

正则判断

`[[expression]]` 这种判断形式，支持正则表达式。

```
1. [[ string1 =~ regex ]]
```

上面的语法中，`regex` 是一个正则表示式，`==` 是正则比较运算符。

下面是一个例子。

```
1. #!/bin/bash
2.
3. INT=-5
4.
5. if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
6.     echo "INT is an integer."
7.     exit 0
8. else
9.     echo "INT is not an integer." >&2
10.    exit 1
11. fi
```

上面代码中，先判断变量 `INT` 的字符串形式，是否满足 `^-?[0-9]+$` 的正则模式，如果满足就表明它是一个整数。

test 判断的逻辑运算

通过逻辑运算，可以把多个 `test` 判断表达式结合起来，创造更复杂的判断。三种逻辑运算 `AND`，`OR`，和 `NOT`，都有自己的专用符号。

- `AND` 运算：符号 `&&`，也可使用参数 `-a`。
- `OR` 运算：符号 `||`，也可使用参数 `-o`。
- `NOT` 运算：符号 `!`。

下面是一个 `AND` 的例子，判断整数是否在某个范围之内。

```
1. #!/bin/bash
2.
3. MIN_VAL=1
4. MAX_VAL=100
5.
6. INT=50
7.
8. if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
9.     if [[ $INT -ge $MIN_VAL && $INT -le $MAX_VAL ]]; then
10.        echo "INT is within $MIN_VAL to $MAX_VAL."
11.    else
12.        echo "INT is out of range."
13.    fi
14. else
15.    echo "INT is not an integer." >&2
16.    exit 1
17. fi
```

上面例子中，`&&` 用来连接两个判断条件：大于等于 `$MIN_VAL`，并且小于等于 `$MAX_VAL`。

使用否定操作符 `!` 时，最好用圆括号确定转义的范围。

```
1. if [ ! \ ( $INT -ge $MIN_VAL -a $INT -le $MAX_VAL \ ) ]; then
2.     echo "$INT is outside $MIN_VAL to $MAX_VAL."
3. else
4.     echo "$INT is in range."
5. fi
```

上面例子中，`test` 命令内部使用的圆括号，必须使用引号或者转义，否则会被 Bash 解释。

算术判断

Bash 还提供了 `((...))` 作为算术条件，进行算术运算的判断。

```
1. if ((3 > 2)); then
2.     echo "true"
3. fi
```

上面代码执行后，会打印出 `true`。

注意，算术判断不需要使用 `test` 命令，而是直接使用 `((...))` 结构。这个结构的返回值，决定了判断的真伪。

如果算术计算的结果是非零值，则表示判断成立。这一点跟命令的返回值正好相反，需要小心。

```
1. $ if ((1)); then echo "It is true."; fi
2. It is true.
3. $ if ((0)); then echo "It is true."; else echo "it is false."; fi
4. It is false.
```

上面例子中，`((1))` 表示判断成立，`((0))` 表示判断不成立。

算术条件 `((...))` 也可以用于变量赋值。

```
1. $ if (( foo = 5 ));then echo "foo is $foo"; fi
2. foo is 5
```

上面例子中，`((foo = 5))` 完成了两件事情。首先把 `5` 赋值给变量 `foo`，然后根据返回值 `5`，判断条件为真。

注意，赋值语句返回等号右边的值，如果返回的是 `0`，则判断为假。

```
1. $ if (( foo = 0 ));then echo "It is true.";else echo "It is false."; fi
2. It is false.
```

下面是用算术条件改写的数值判断脚本。


```

1. #!/bin/bash
2.
3. INT=-5
4.
5. if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
6.     if ((INT == 0)); then
7.         echo "INT is zero."
8.     else
9.         if ((INT < 0)); then
10.            echo "INT is negative."
11.        else
12.            echo "INT is positive."
13.        fi
14.        if (( (INT % 2) == 0 )); then
15.            echo "INT is even."
16.        else
17.            echo "INT is odd."
18.        fi
19.    fi
20. else
21.     echo "INT is not an integer." >&2
22.     exit 1
23. fi

```

只要是算术表达式，都能用于 `((...))` 语法，详见《Bash 的算术运算》一章。

普通命令的逻辑运算

如果 `if` 结构使用的不是 `test` 命令，而是普通命令，比如上一节的 `((...))` 算术运算，或者 `test` 命令与普通命令混用，那么可以使用 Bash 的命令控制操作符 `&&`（AND）和 `||`（OR），进行多个命令的逻辑运算。

```

1. $ command1 && command2
2. $ command1 || command2

```

对于 `&&` 操作符，先执行 `command1`，只有 `command1` 执行成功后，才会执行 `command2`。对于 `||` 操作符，先执行 `command1`，只有 `command1` 执行失败后，才会执行 `command2`。

```

1. $ mkdir temp && cd temp

```

上面的命令会创建一个名为 `temp` 的目录，执行成功后，才会执行第二个命令，进入这个目录。

```

1. $ [ -d temp ] || mkdir temp

```

上面的命令会测试目录 `temp` 是否存在，如果不存在，就会执行第二个命令，创建这个目录。这种写法非常有助于在脚本中处理错误。

```

1. [ ! -d temp ] && exit 1

```

上面的命令中，如果 `temp` 子目录不存在，脚本会终止，并且返回值为 `1`。

下面就是 `if` 与 `&&` 结合使用的写法。

```
1. if [ condition ] && [ condition ]; then
2.     command
3. fi
```

下面是一个示例。

```
1. #! /bin/bash
2.
3. filename=$1
4. word1=$2
5. word2=$3
6.
7. if grep $word1 $filename && grep $word2 $filename
8. then
9.     echo "$word1 and $word2 are both in $filename."
10. fi
```

上面的例子只有在指定文件里面，同时存在搜索词 `word1` 和 `word2`，就会执行 `if` 的命令部分。

下面的示例演示如何将一个 `&&` 判断表达式，改写成对应的 `if` 结构。

```
1. [[ -d "$dir_name" ]] && cd "$dir_name" && rm *
2.
3. # 等同于
4.
5. if [[ ! -d "$dir_name" ]]; then
6.     echo "No such directory: '$dir_name'" >&2
7.     exit 1
8. fi
9. if ! cd "$dir_name"; then
10.    echo "Cannot cd to '$dir_name'" >&2
11.    exit 1
12. fi
13. if ! rm *; then
14.    echo "File deletion failed. Check results" >&2
15.    exit 1
16. fi
```

case 结构

`case` 结构用于多值判断，可以为每个值指定对应的命令，跟包含多个 `elif` 的 `if` 结构等价，但是语义更好。它的语法如下。

```
1. case expression in
2.     pattern )
```

```

3.     commands ;;
4.     pattern )
5.     commands ;;
6.     ...
7. esac

```

上面代码中，`expression` 是一个表达式，`pattern` 是表达式的值或者一个模式，可以有多条，用来匹配多个值，每条以两个分号（`;`）结尾。

```

1. #!/bin/bash
2.
3. echo -n "输入一个1到3之间的数字（包含两端）> "
4. read character
5. case $character in
6.     1 ) echo 1
7.         ;;
8.     2 ) echo 2
9.         ;;
10.    3 ) echo 3
11.        ;;
12.    * ) echo 输入不符合要求
13. esac

```

上面例子中，最后一条匹配语句的模式是 `*`，这个通配符可以匹配其他字符和没有输入字符的情况，类似 `if` 的 `else` 部分。

下面是另一个例子。

```

1. #!/bin/bash
2.
3. OS=$(uname -s)
4.
5. case "$OS" in
6.     FreeBSD) echo "This is FreeBSD" ;;
7.     Darwin) echo "This is Mac OSX" ;;
8.     AIX) echo "This is AIX" ;;
9.     Minix) echo "This is Minix" ;;
10.    Linux) echo "This is Linux" ;;
11.    *) echo "Failed to identify this OS" ;;
12. esac

```

上面的例子判断当前是什么操作系统。

`case` 的匹配模式可以使用各种通配符，下面是一些例子。

- `a)`：匹配 `a`。
- `a|b)`：匹配 `a` 或 `b`。
- `[:alpha:])`：匹配单个字母。
- `???)`：匹配3个字符的单词。
- `*.txt)`：匹配 `.txt` 结尾。

- `*)`：匹配任意输入，通过作为 `case` 结构的最后一个模式。

```

1. #!/bin/bash
2.
3. echo -n "输入一个字母或数字 > "
4. read character
5. case $character in
6.     [[:lower:]] | [[:upper:]] ) echo "输入了字母 $character"
7.
8.     [0-9] ) echo "输入了数字 $character"
9.
10.    * ) echo "输入不符合要求"
11. esac

```

上面例子中，使用通配符 `[[:lower:]] | [[:upper:]]` 匹配字母，`[0-9]` 匹配数字。

Bash 4.0之前，`case` 结构只能匹配一个条件，然后就会退出 `case` 结构。Bash 4.0之后，允许匹配多个条件，这时可以用 `;;&` 终止每个条件块。

```

1. #!/bin/bash
2. # test.sh
3.
4. read -n 1 -p "Type a character > "
5. echo
6. case $REPLY in
7.     [[:upper:]]) echo "'$REPLY' is upper case." ;;&
8.     [[:lower:]]) echo "'$REPLY' is lower case." ;;&
9.     [[:alpha:]] ) echo "'$REPLY' is alphabetic." ;;&
10.    [[:digit:]] ) echo "'$REPLY' is a digit." ;;&
11.    [[:graph:]] ) echo "'$REPLY' is a visible character." ;;&
12.    [[:punct:]] ) echo "'$REPLY' is a punctuation symbol." ;;&
13.    [[:space:]] ) echo "'$REPLY' is a whitespace character." ;;&
14.    [[:xdigit:]] ) echo "'$REPLY' is a hexadecimal digit." ;;&
15. esac

```

执行上面的脚本，会得到下面的结果。

```

1. $ test.sh
2. Type a character > a
3. 'a' is lower case.
4. 'a' is alphabetic.
5. 'a' is a visible character.
6. 'a' is a hexadecimal digit.

```

可以看到条件语句结尾添加了 `;;&` 以后，在匹配一个条件之后，并没有退出 `case` 结构，而是继续判断下一个条件。

参考链接

- [The Linux Command Line](#), William Shotts

循环

Bash 提供三种循环语法 `for` 、 `while` 和 `until` 。

while 循环

`while` 循环有一个判断条件，只要符合条件，就不断循环执行指定的语句。

```
1. while condition; do
2.     commands
3. done
```

上面代码中，只要满足条件 `condition` ，就会执行命令 `commands` 。然后，再次判断是否满足条件 `condition` ，只要满足，就会一直执行下去。只有不满足条件，才会退出循环。

循环条件 `condition` 可以使用 `test` 命令，跟 `if` 结构的判断条件写法一致。

```
1. #!/bin/bash
2.
3. number=0
4. while [ "$number" -lt 10 ]; do
5.     echo "Number = $number"
6.     number=$((number + 1))
7. done
```

上面例子中，只要变量 `$number` 小于10，就会不断加1，直到 `$number` 等于10，然后退出循环。

关键字 `do` 可以跟 `while` 不在同一行，这时两者之间不需要使用分号分隔。

```
1. while true
2. do
3.     echo 'Hi, while looping ...';
4. done
```

上面的例子会无限循环，可以按下 `Ctrl + c` 停止。

`while` 循环写成一行，也是可以的。

```
1. $ while true; do echo 'Hi, while looping ...'; done
```

`while` 的条件部分也可以是执行一个命令。

```
1. $ while echo 'ECHO'; do echo 'Hi, while looping ...'; done
```

上面例子中，判断条件是 `echo 'ECHO'` 。由于这个命令总是执行成功，所以上面命令会产生无限循环。

`while` 的条件部分可以执行任意数量的命令，但是执行结果的真伪只看最后一个命令的执行结果。

```
1. $ while true; false; do echo 'Hi, looping ...'; done
```

上面代码运行后，不会有任何输出，因为 `while` 的最后一个命令是 `false`。

until 循环

`until` 循环与 `while` 循环恰好相反，只要不符合判断条件（判断条件失败），就不断循环执行指定的语句。一旦符合判断条件，就退出循环。

```
1. until condition; do
2.     commands
3. done
```

关键字 `do` 可以与 `until` 不写在同一行，这时两者之间不需要分号分隔。

```
1. until condition
2. do
3.     commands
4. done
```

下面是一个例子。

```
1. $ until false; do echo 'Hi, until looping ...'; done
2. Hi, until looping ...
3. Hi, until looping ...
4. Hi, until looping ...
5. ^C
```

上面代码中，`until` 的部分一直为 `false`，导致命令无限运行，必须按下 `Ctrl + c` 终止。

```
1. #!/bin/bash
2.
3. number=0
4. until [ "$number" -ge 10 ]; do
5.     echo "Number = $number"
6.     number=$((number + 1))
7. done
```

上面例子中，只要变量 `number` 小于10，就会不断加1，直到 `number` 大于等于10，就退出循环。

`until` 的条件部分也可以是一个命令，表示在这个命令执行成功之前，不断重复尝试。

```
1. until cp $1 $2; do
2.     echo 'Attempt to copy failed. waiting...'
3.     sleep 5
```

```
4. done
```

上面例子表示，只要 `cp $1 $2` 这个命令执行不成功，就5秒钟后再尝试一次，直到成功为止。

`until` 循环都可以转为 `while` 循环，只要把条件设为否定即可。上面这个例子可以改写如下。

```
1. while ! cp $1 $2; do
2.     echo 'Attempt to copy failed. waiting...'
3.     sleep 5
4. done
```

一般来说，`until` 用得比较少，完全可以统一都使用 `while`。

for...in 循环

`for...in` 循环用于遍历列表的每一项。

```
1. for variable in list
2. do
3.     commands
4. done
```

上面语法中，`for` 循环会依次从 `list` 列表中取出一项，作为变量 `variable`，然后在循环体中进行处理。

关键词 `do` 可以跟 `for` 写在同一行，两者使用分号分隔。

```
1. for variable in list; do
2.     commands
3. done
```

下面是一个例子。

```
1. #!/bin/bash
2.
3. for i in word1 word2 word3; do
4.     echo $i
5. done
```

上面例子中，`word1 word2 word3` 是一个包含三个单词的列表，变量 `i` 依次等于 `word1`、`word2`、`word3`，命令 `echo $i` 则会相应地执行三次。

列表可以由通配符产生。

```
1. for i in *.png; do
2.     ls -l $i
3. done
```


上面例子中，`*.png` 会替换成当前目录中所有 PNG 图片文件，变量 `i` 会依次等于每一个文件。

列表也可以通过子命令产生。

```
1. #!/bin/bash
2.
3. count=0
4. for i in $(cat ~/.bash_profile); do
5.     count=$((count + 1))
6.     echo "Word $count ($i) contains $(echo -n $i | wc -c) characters"
7. done
```

上面例子中，`cat ~/.bash_profile` 命令会输出 `~/.bash_profile` 文件的内容，然后通过遍历每一个词，计算该文件一共包含多少个词，以及每个词有多少个字符。

`in list` 的部分可以省略，这时 `list` 默认等于脚本的所有参数 `$@`。但是，为了可读性，最好还是不要省略，参考下面的例子。

```
1. for filename; do
2.     echo "$filename"
3. done
4.
5. # 等同于
6.
7. for filename in "$@" ; do
8.     echo "$filename"
9. done
```

在函数体中也是一样的，`for...in` 循环省略 `in list` 的部分，则 `list` 默认等于函数的所有参数。

for 循环

`for` 循环还支持 C 语言的循环语法。

```
1. for (( expression1; expression2; expression3 )); do
2.     commands
3. done
```

上面代码中，`expression1` 用来初始化循环条件，`expression2` 用来决定循环结束的条件，`expression3` 在每次循环迭代的末尾执行，用于更新值。

注意，循环条件放在双重圆括号之中。另外，圆括号之中使用变量，不必加上美元符号 `$`。

它等同于下面的 `while` 循环。

```
1. (( expression1 ))
2. while (( expression2 )); do
3.     commands
4. (( expression3 ))
```

```
5. done
```

下面是一个例子。

```
1. for (( i=0; i<5; i=i+1 )); do
2.     echo $i
3. done
```

上面代码中，初始化变量 `i` 的值为0，循环执行的条件是 `i` 小于5。每次循环迭代结束时，`i` 的值加1。

`for` 条件部分的三个语句，都可以省略。

```
1. for (;;))
2. do
3.     read var
4.     if [ "$var" = "." ]; then
5.         break
6.     fi
7. done
```

上面脚本会反复读取命令行输入，直到用户输入了一个点（`.`）位为止，才会跳出循环。

break, continue

Bash 提供了两个内部命令 `break` 和 `continue`，用来在循环内部跳出循环。

`break` 命令立即终止循环，程序继续执行循环块之后的语句，即不再执行剩下的循环。

```
1. #!/bin/bash
2.
3. for number in 1 2 3 4 5 6
4. do
5.     echo "number is $number"
6.     if [ "$number" = "3" ]; then
7.         break
8.     fi
9. done
```

上面例子只会打印3行结果。一旦变量 `$number` 等于3，就会跳出循环，不再继续执行。

`continue` 命令立即终止本轮循环，开始执行下一轮循环。

```
1. #!/bin/bash
2.
3. while read -p "What file do you want to test?" filename
4. do
5.     if [ ! -e "$filename" ]; then
6.         echo "The file does not exist."
7.         continue
```

循环

```
8.     fi
9.
10.    echo "You entered a valid file.."
11.    done
```

上面例子中，只要用户输入的文件不存在，`continue` 命令就会生效，直接进入下一轮循环（让用户重新输入文件名），不再执行后面的打印语句。

select 结构

`select` 结构主要用来生成简单的菜单。它的语法与 `for...in` 循环基本一致。

```
1. select name
2. [in list]
3. do
4.     commands
5. done
```

Bash 会对 `select` 依次进行下面的处理。

1. `select` 生成一个菜单，内容是列表 `list` 的每一项，并且每一项前面还有一个数字编号。
2. Bash 提示用户选择一项，输入它的编号。
3. 用户输入以后，Bash 会将该项的内容存在变量 `name`，该项的编号存入环境变量 `REPLY`。如果用户没有输入，就按回车键，Bash 会重新输出菜单，让用户选择。
4. 执行命令体 `commands`。
5. 执行结束后，回到第一步，重复这个过程。

下面是一个例子。

```
1. #!/bin/bash
2. # select.sh
3.
4. select brand in Samsung Sony iphone symphony Walton
5. do
6.     echo "You have chosen $brand"
7. done
```

执行上面的脚本，Bash 会输出一个品牌的列表，让用户选择。

```
1. $ ./select.sh
2. 1) Samsung
3. 2) Sony
4. 3) iphone
5. 4) symphony
6. 5) Walton
7. #?
```

如果用户没有输入编号，直接按回车键。Bash 就会重新输出一遍这个菜单，直到用户按下 `Ctrl + c`，退出执行。

`select` 可以与 `case` 结合，针对不同项，执行不同的命令。

```
1. #!/bin/bash
2.
3. echo "Which Operating System do you like?"
4.
5. select os in Ubuntu LinuxMint Windows8 Windows7 WindowsXP
6. do
7.     case $os in
8.         "Ubuntu"|"LinuxMint")
9.             echo "I also use $os."
10.        ;;
11.        "Windows8" | "Windows10" | "WindowsXP")
12.            echo "Why don't you try Linux?"
13.        ;;
14.        *)
15.            echo "Invalid entry."
16.            break
17.        ;;
18.    esac
19. done
```

上面例子中，`case` 针对用户选择的不同项，执行不同的命令。

参考链接

- [Bash Select Command](#), Fahmida Yesmin

Bash 函数

本章介绍 Bash 函数的用法。

简介

函数 (function) 是可以重复使用的代码片段，有利于代码的复用。它与别名 (alias) 的区别是，别名只适合封装简单的单个命令，函数则可以封装复杂的多行命令。

函数总是在当前 Shell 执行，这是跟脚本的一个重大区别，Bash 会新建一个子 Shell 执行脚本。如果函数与脚本同名，函数会优先执行。但是，函数的优先级不如别名，即如果函数与别名同名，那么别名优先执行。

Bash 函数定义的语法有两种。

```
1. # 第一种
2. fn() {
3.     # codes
4. }
5.
6. # 第二种
7. function fn() {
8.     # codes
9. }
```

上面代码中，`fn` 是自定义的函数名，函数代码就写在大括号之中。这两种写法是等价的。

下面是一个简单函数的例子。

```
1. hello() {
2.     echo "Hello $1"
3. }
```

上面代码中，函数体里面的 `$1` 表示函数调用时的第一个参数。

调用时，就直接写函数名，参数跟在函数名后面。

```
1. $ hello world
2. hello world
```

下面是一个多行函数的例子，显示当前日期时间。

```
1. today() {
2.     echo -n "Today's date is: "
3.     date +"%A, %B %-d, %Y"
4. }
```

删除一个函数，可以使用 `unset` 命令。

```
1. unset -f functionName
```

查看当前 Shell 已经定义的所有函数，可以使用 `declare` 命令。

```
1. $ declare -f
```

上面的 `declare` 命令不仅会输出函数名，还会输出所有定义。输出顺序是按照函数名的字母表顺序。由于会输出很多内容，最好通过管道命令配合 `more` 或 `less` 使用。

`declare` 命令还支持查看单个函数的定义。

```
1. $ declare -f functionName
```

`declare -F` 可以输出所有已经定义的函数名，不含函数体。

```
1. $ declare -F
```

参数变量

函数体内可以使用参数变量，获取函数参数。函数的参数变量，与脚本参数变量是一致的。

- `$1` ~ `$9` : 函数的第一个到第9个的参数。
- `$0` : 函数所在的脚本名。
- `$#` : 函数的参数总数。
- `$@` : 函数的全部参数，参数之间使用空格分隔。
- `$*` : 函数的全部参数，参数之间使用变量 `$IFS` 值的第一个字符分隔，默认为空格，但是可以自定义。

如果函数的参数多于9个，那么第10个参数可以用 `${10}` 的形式引用，以此类推。

下面是一个示例脚本 `test.sh` 。

```
1. #!/bin/bash
2. # test.sh
3.
4. function alice {
5.     echo "alice: $@"
6.     echo "$0: $1 $2 $3 $4"
7.     echo "$# arguments"
8.
9. }
10.
11. alice in wonderland
```

运行该脚本，结果如下。

```
1. $ bash test.sh
2. alice: in wonderland
3. test.sh: in wonderland
4. 2 arguments
```

上面例子中，由于函数 `alice` 只有第一个和第二个参数，所以第三个和第四个参数为空。

下面是一个日志函数的例子。

```
1. function log_msg {
2.     echo "[`date '+ %F %T'` ]: $@"
3. }
```

使用方法如下。

```
1. $ log_msg "This is sample log message"
2. [ 2018-08-16 19:56:34 ]: This is sample log message
```

return 命令

`return` 命令用于从函数返回一个值。函数执行到这条命令，就不再往下执行了，直接返回了。

```
1. function func_return_value {
2.     return 10
3. }
```

函数将返回值返回给调用者。如果命令行直接执行函数，下一个命令可以用 `$?` 拿到返回值。

```
1. $ func_return_value
2. $ echo "Value returned by function is: $?"
3. Value returned by function is: 10
```

`return` 后面不跟参数，只用于返回也是可以的。

```
1. function name {
2.     commands
3.     return
4. }
```

全局变量和局部变量，local 命令

Bash 函数体内直接声明的变量，属于全局变量，整个脚本都可以读取。这一点需要特别小心。

```
1. # 脚本 test.sh
2. fn () {
3.     foo=1
```

```
4.     echo "fn: foo = $foo"
5. }
6.
7. fn
8. echo "global: foo = $foo"
```

上面脚本的运行结果如下。

```
1. $ bash test.sh
2. fn: foo = 1
3. global: foo = 1
```

上面例子中，变量 `$foo` 是在函数 `fn` 内部声明的，函数体外也可以读取。

函数体内不仅可以声明全局变量，还可以修改全局变量。

```
1. foo=1
2.
3. fn () {
4.     foo=2
5. }
6.
7. echo $foo
```

上面代码执行后，输出的变量 `$foo` 值为2。

函数里面可以用 `local` 命令声明局部变量。

```
1. # 脚本 test.sh
2. fn () {
3.     local foo
4.     foo=1
5.     echo "fn: foo = $foo"
6. }
7.
8. fn
9. echo "global: foo = $foo"
```

上面脚本的运行结果如下。

```
1. $ bash test.sh
2. fn: foo = 1
3. global: foo =
```

上面例子中，`local` 命令声明的 `$foo` 变量，只在函数体内有效，函数体外没有定义。

参考链接

- [How to define and use functions in Linux Shell Script](#), by Pradeep Kumar

数组

数组（array）是一个包含多个值的变量。成员的编号从0开始，数量没有上限，也没有要求成员被连续索引。

创建数组

数组可以采用逐个赋值的方法创建。

```
1. ARRAY[INDEX]=value
```

上面语法中，`ARRAY` 是数组的名字，可以是任意合法的变量名。`INDEX` 是一个大于或等于零的整数，也可以是算术表达式。注意数组第一个元素的下标是0，而不是1。

下面创建一个三个成员的数组。

```
1. $ array[0]=val
2. $ array[1]=val
3. $ array[2]=val
```

数组也可以采用一次性赋值的方式创建。

```
1. ARRAY=(value1 value2 ... valueN)
2.
3. # 等同于
4.
5. ARRAY=(
6.     value1
7.     value2
8.     value3
9. )
```

采用上面方式创建数组时，可以按照默认顺序赋值，也可以在每个值前面指定位置。

```
1. $ array=(a b c)
2. $ array=([2]=c [0]=a [1]=b)
3.
4. $ days=(Sun Mon Tue Wed Thu Fri Sat)
5. $ days=([0]=Sun [1]=Mon [2]=Tue [3]=Wed [4]=Thu [5]=Fri [6]=Sat)
```

只为某些值指定位置，也是可以的。

```
1. names=(hatter [5]=duchess alice)
```

上面例子中，`hatter` 是数组的0号位置，`duchess` 是5号位置，`alice` 是6号位置。

没有赋值的数组元素的默认值是空字符串。

定义数组的时候，可以使用通配符。

```
1. $ mp3s=( *.mp3 )
```

上面例子中，将当前目录的所有 MP3 文件，放进一个数组。

先用 `declare -a` 命令声明一个数组，也是可以的。

```
1. $ declare -a ARRAYNAME
```

`read -a` 命令则是将用户的命令行输入，读入一个数组。

```
1. $ read -a dice
```

上面命令将用户的命令行输入，读入数组 `dice`。

读取数组

读取单个元素

读取数组指定位置的成员，要使用下面的语法。

```
1. $ echo ${array[i]}      # i 是索引
```

上面语法里面的大括号是必不可少的，否则 Bash 会把索引部分 `[i]` 按照原样输出。

```
1. $ array[0]=a
2.
3. $ echo ${array[0]}
4. a
5.
6. $ echo $array[0]
7. a[0]
```

上面例子中，数组的第一个元素是 `a`。如果不加大括号，Bash 会直接读取 `$array` 首成员的值，然后将 `[0]` 按照原样输出。

读取所有成员

`@` 和 `*` 是数组的特殊索引，表示返回数组的所有成员。

```
1. $ foo=(a b c d e f)
2. $ echo ${foo[@]}
3. a b c d e f
```

这两个特殊索引配合 `for` 循环，就可以用来遍历数组。

```
1. for i in "${names[@]}; do
2.     echo $i
3. done
```

`@` 和 `*` 放不放在双引号之中，是有差别的。

```
1. $ activities=( swimming "water skiing" canoeing "white-water rafting" surfing )
2. $ for act in ${activities[@]}; \
3. do \
4. echo "Activity: $act"; \
5. done
6.
7. Activity: swimming
8. Activity: water
9. Activity: skiing
10. Activity: canoeing
11. Activity: white-water
12. Activity: rafting
13. Activity: surfing
```

上面的例子中，数组 `activities` 实际包含5个元素，但是 `for...in` 循环直接遍历 `${activities[@]}`，会导致返回7个结果。为了避免这种情况，一般把 `${activities[@]}` 放在双引号之中。

```
1. $ for act in "${activities[@]}; \
2. do \
3. echo "Activity: $act"; \
4. done
5.
6. Activity: swimming
7. Activity: water skiing
8. Activity: canoeing
9. Activity: white-water rafting
10. Activity: surfing
```

上面例子中，`${activities[@]}` 放在双引号之中，遍历就会返回正确的结果。

`${activities[*]}` 不放在双引号之中，跟 `${activities[@]}` 不放在双引号之中是一样的。

```
1. $ for act in ${activities[*]}; \
2. do \
3. echo "Activity: $act"; \
4. done
5.
6. Activity: swimming
7. Activity: water
8. Activity: skiing
9. Activity: canoeing
```

```
10. Activity: white-water
11. Activity: rafting
12. Activity: surfing
```

`${activities[*]}` 放在双引号之中，所有元素就会变成单个字符串返回。

```
1. $ for act in "${activities[*]"; \
2. do \
3. echo "Activity: $act"; \
4. done
5.
6. Activity: swimming water skiing canoeing white-water rafting surfing
```

所以，拷贝一个数组的最方便方法，就是写成下面这样。

```
1. $ hobbies=( "${activities[@]}" )
```

上面例子中，数组 `activities` 被拷贝给了另一个数组 `hobbies`。

这种写法也可以用来为新数组添加成员。

```
1. $ hobbies=( "${activities[@}" diving )
```

上面例子中，新数组 `hobbies` 在数组 `activities` 的所有成员之后，又添加了一个成员。

默认位置

如果读取数组成员时，没有读取指定哪一个位置的成员，默认使用 `0` 号位置。

```
1. $ declare -a foo
2. $ foo=A
3. $ echo ${foo[0]}
4. A
```

上面例子中，`foo` 是一个数组，赋值的时候不指定位置，实际上是给 `foo[0]` 赋值。

引用一个不带下标的数组变量，则引用的是 `0` 号位置的数组元素。

```
1. $ foo=(a b c d e f)
2. $ echo ${foo}
3. a
4. $ echo $foo
5. a
```

上面例子中，引用数组元素的时候，没有指定位置，结果返回的是 `0` 号位置。

数组的长度

要想知道数组的长度（即一共包含多少成员），可以使用下面两种语法。

```
1. ${#array[*]}
2. ${#array[@]}
```

下面是一个例子。

```
1. $ a[100]=foo
2.
3. $ echo ${#a[*]}
4. 1
5.
6. $ echo ${#a[@]}
7. 1
```

上面例子中，把字符串赋值给 `100` 位置的数组元素，这时的数组只有一个元素。

注意，如果用这种语法去读取具体的数组成员，就会返回该成员的字符串长度。这一点必须小心。

```
1. $ a[100]=foo
2. $ echo ${#a[100]}
3. 3
```

上面例子中，`${#a[100]}` 实际上是返回数组第100号成员 `a[100]` 的值（`foo`）的字符串长度。

提取数组序号

`${!array[@]}` 或 `${!array[*]}`，可以返回数组的成员序号，即哪些位置是有值的。

```
1. $ arr=( [5]=a [9]=b [23]=c )
2. $ echo ${!arr[@]}
3. 5 9 23
4. $ echo ${!arr[*]}
5. 5 9 23
```

上面例子中，数组的5、9、23号位置有值。

利用这个语法，也可以通过 `for` 循环遍历数组。

```
1. arr=(a b c d)
2.
3. for i in ${!arr[@]}; do
4.     echo ${arr[i]}
5. done
```

提取数组成员

`${array[@]:position:length}` 的语法可以提取数组成员。

```
1. $ food=( apples bananas cucumbers dates eggs fajitas grapes )
2. $ echo ${food[@]:1:1}
3. bananas
4. $ echo ${food[@]:1:3}
5. bananas cucumbers dates
```

上面例子中，`${food[@]:1:1}` 返回从数组1号位置开始的1个成员，`${food[@]:1:3}` 返回从1号位置开始的3个成员。

如果省略长度参数 `length`，则返回从指定位置开始的所有成员。

```
1. $ echo ${food[@]:4}
2. eggs fajitas grapes
```

上面例子返回从4号位置开始到结束的所有成员。

追加数组成员

数组末尾追加成员，可以使用 `+=` 赋值运算符。它能够自动地把值追加到数组末尾。否则，就需要知道数组的最大序号，比较麻烦。

```
1. $ foo=(a b c)
2. $ echo ${foo[@]}
3. a b c
4.
5. $ foo+=(d e f)
6. $ echo ${foo[@]}
7. a b c d e f
```

删除数组

删除一个数组成员，使用 `unset` 命令。

```
1. $ foo=(a b c d e f)
2. $ echo ${foo[@]}
3. a b c d e f
4.
5. $ unset foo[2]
6. $ echo ${foo[@]}
7. a b d e f
```

上面例子中，删除了数组中的第三个元素，下标为2。

删除成员也可以将这个成员设为空值。

```
1. $ foo=(a b c d e f)
2. $ foo[1]=''
3. $ echo ${foo[@]}
4. a c d e f
```

上面例子中，将数组的第二个成员设为空字符串，就删除了这个成员。

由于空值就是空字符串，所以下面这样写也可以，但是不建议这种写法。

```
1. $ foo[1]=
```

上面的写法也相当于删除了数组的第二个成员。

直接将数组变量赋值为空字符串，相当于删除数组的第一个成员。

```
1. $ foo=(a b c d e f)
2. $ foo=''
3. $ echo ${foo[@]}
4. b c d e f
```

上面的写法相当于删除了数组的第一个成员。

`unset ArrayName` 可以清空整个数组。

```
1. $ unset ARRAY
2.
3. $ echo ${ARRAY[*]}
4. <--no output-->
```

关联数组

Bash 的新版本支持关联数组。关联数组使用字符串而不是整数作为数组索引。

`declare -A` 可以声明关联数组。

```
1. declare -A colors
2. colors["red"]="ff0000"
3. colors["green"]="00ff00"
4. colors["blue"]="0000ff"
```

整数索引的数组，可以直接使用变量名创建数组，关联数组则必须用带有 `-A` 选项的 `declare` 命令声明创建。

访问关联数组成员的方式，几乎与整数索引数组相同。

```
1. echo ${colors["blue"]}
```


set 命令

`set` 命令是 Bash 脚本的重要环节，却常常被忽视，导致脚本的安全性和可维护性出问题。本章介绍 `set` 的基本用法，帮助你写出更安全的 Bash 脚本。

简介

我们知道，Bash 执行脚本时，会创建一个子 Shell。

```
1. $ bash script.sh
```

上面代码中，`script.sh` 是在一个子 Shell 里面执行。这个子 Shell 就是脚本的执行环境，Bash 默认给定了这个环境的各种参数。

`set` 命令用来修改子 Shell 环境的运行参数，即定制环境。一共有十几个参数可以定制，[官方手册](#)有完整清单，本章介绍其中最常用的几个。

顺便提一下，如果命令行下不带任何参数，直接运行 `set`，会显示所有的环境变量和 Shell 函数。

```
1. $ set
```

set -u

执行脚本时，如果遇到不存在的变量，Bash 默认忽略它。

```
1. #!/usr/bin/env bash
2.
3. echo $a
4. echo bar
```

上面代码中，`$a` 是一个不存在的变量。执行结果如下。

```
1. $ bash script.sh
2.
3. bar
```

可以看到，`echo $a` 输出了一个空行，Bash 忽略了不存在的 `$a`，然后继续执行 `echo bar`。大多数情况下，这不是开发者想要的行为，遇到变量不存在，脚本应该报错，而不是一声不响地往下执行。

`set -u` 就用来改变这种行为。脚本在头部加上它，遇到不存在的变量就会报错，并停止执行。

```
1. #!/usr/bin/env bash
2. set -u
3.
```

```
4. echo $a
5. echo bar
```

运行结果如下。

```
1. $ bash script.sh
2. bash: script.sh:行4: a: 未绑定的变量
```

可以看到，脚本报错了，并且不再执行后面的语句。

`-u` 还有另一种写法 `-o nounset`，两者是等价的。

```
1. set -o nounset
```

set -x

默认情况下，脚本执行后，只输出运行结果，没有其他内容。如果多个命令连续执行，它们的运行结果就会连续输出。有时会分不清，某一段内容是什么命令产生的。

`set -x` 用来在运行结果之前，先输出执行的那一行命令。

```
1. #!/usr/bin/env bash
2. set -x
3.
4. echo bar
```

执行上面的脚本，结果如下。

```
1. $ bash script.sh
2. + echo bar
3. bar
```

可以看到，执行 `echo bar` 之前，该命令会先打印出来，行首以 `+` 表示。这对于调试复杂的脚本是很有用的。

`-x` 还有另一种写法 `-o xtrace`。

```
1. set -o xtrace
```

脚本当中如果要关闭命令输出，可以使用 `set +x`。

```
1. #!/bin/bash
2.
3. number=1
4.
5. set -x
6. if [ $number = "1" ]; then
7.     echo "Number equals 1"
```

```
8. else
9.     echo "Number does not equal 1"
10. fi
11. set +x
```

上面的例子中，只对特定的代码段打开命令输出。

Bash 的错误处理

如果脚本里面有运行失败的命令（返回值非 `0` ），Bash 默认会继续执行后面的命令。

```
1. #!/usr/bin/env bash
2.
3. foo
4. echo bar
```

上面脚本中，`foo` 是一个不存在的命令，执行时会报错。但是，Bash 会忽略这个错误，继续往下执行。

```
1. $ bash script.sh
2. script.sh:行3: foo: 未找到命令
3. bar
```

可以看到，Bash 只是显示有错误，并没有终止执行。

这种行为很不利于脚本安全和除错。实际开发中，如果某个命令失败，往往需要脚本停止执行，防止错误累积。这时，一般采用下面的写法。

```
1. command || exit 1
```

上面的写法表示只要 `command` 有非零返回值，脚本就会停止执行。

如果停止执行之前需要完成多个操作，就要采用下面三种写法。

```
1. # 写法一
2. command || { echo "command failed"; exit 1; }
3.
4. # 写法二
5. if ! command; then echo "command failed"; exit 1; fi
6.
7. # 写法三
8. command
9. if [ "$?" -ne 0 ]; then echo "command failed"; exit 1; fi
```

另外，除了停止执行，还有一种情况。如果两个命令有继承关系，只有第一个命令成功了，才能继续执行第二个命令，那么就要采用下面的写法。

```
1. command1 && command2
```

set -e

上面这些写法多少有些麻烦，容易疏忽。`set -e` 从根本上解决了这个问题，它使得脚本只要发生错误，就终止执行。

```
1. #!/usr/bin/env bash
2. set -e
3.
4. foo
5. echo bar
```

执行结果如下。

```
1. $ bash script.sh
2. script.sh:行4: foo: 未找到命令
```

可以看到，第4行执行失败以后，脚本就终止执行了。

`set -e` 根据返回值来判断，一个命令是否运行失败。但是，某些命令的非零返回值可能不表示失败，或者开发者希望在命令失败的情况下，脚本继续执行下去。这时可以暂时关闭 `set -e`，该命令执行结束后，再重新打开 `set -e`。

```
1. set +e
2. command1
3. command2
4. set -e
```

上面代码中，`set +e` 表示关闭 `-e` 选项，`set -e` 表示重新打开 `-e` 选项。

还有一种方法是使用 `command || true`，使得该命令即使执行失败，脚本也不会终止执行。

```
1. #!/bin/bash
2. set -e
3.
4. foo || true
5. echo bar
```

上面代码中，`true` 使得这一行语句总是会执行成功，后面的 `echo bar` 会执行。

`-e` 还有另一种写法 `-o errexit`。

```
1. set -o errexit
```

set -o pipefail

`set -e` 有一个例外情况，就是不适用于管道命令。

所谓管道命令，就是多个子命令通过管道运算符（`|`）组合成为一个大的命令。Bash 会把最后一个子命令的返回值，作为整个命令的返回值。也就是说，只要最后一个子命令不失败，管道命令总是会执行成功，因此它后面命令依然会执行，`set -e` 就失效了。

请看下面这个例子。

```
1. #!/usr/bin/env bash
2. set -e
3.
4. foo | echo a
5. echo bar
```

执行结果如下。

```
1. $ bash script.sh
2. a
3. script.sh:行4: foo: 未找到命令
4. bar
```

上面代码中，`foo` 是一个不存在的命令，但是 `foo | echo a` 这个管道命令会执行成功，导致后面的 `echo bar` 会继续执行。

`set -o pipefail` 用来解决这种情况，只要一个子命令失败，整个管道命令就失败，脚本就会终止执行。

```
1. #!/usr/bin/env bash
2. set -eo pipefail
3.
4. foo | echo a
5. echo bar
```

运行后，结果如下。

```
1. $ bash script.sh
2. a
3. script.sh:行4: foo: 未找到命令
```

可以看到，`echo bar` 没有执行。

其他参数

`set` 命令还有一些其他参数。

- `set -n`：等同于 `set -o noexec`，不运行命令，只检查语法是否正确。
- `set -f`：等同于 `set -o noglob`，表示不对通配符进行文件名扩展。
- `set -v`：等同于 `set -o verbose`，表示打印 Shell 接收到的每一行输入。

上面的 `-f` 和 `-v` 参数，可以分别使用 `set +f`、`set +v` 关闭。

set 命令总结

上面重点介绍的 `set` 命令的四个参数，一般都放在一起使用。

```
1. # 写法一
2. set -euxo pipefail
3.
4. # 写法二
5. set -eux
6. set -o pipefail
```

这两种写法建议放在所有 Bash 脚本的头部。

另一种办法是在执行 Bash 脚本的时候，从命令行传入这些参数。

```
1. $ bash -euxo pipefail script.sh
```

shopt 命令

`shopt` 命令用来调整 Shell 的参数，跟 `set` 命令的作用很类似。之所以会有这两个类似命令的主要原因是，`set` 是从 Ksh 继承的，属于 POSIX 规范的一部分，而 `shopt` 是 Bash 特有的。

直接输入 `shopt` 可以查看所有参数，以及它们各自打开和关闭的状态。

```
1. $ shopt
```

`shopt` 命令后面跟着参数名，可以查询该参数是否打开。

```
1. $ shopt globstar
2. globstar off
```

上面例子表示 `globstar` 参数默认是关闭的。

(1) -s

`-s` 用来打开某个参数。

```
1. $ shopt -s optionNameHere
```

(2) -u

`-u` 用来关闭某个参数。

```
1. $ shopt -u optionNameHere
```

举例来说，`histappend` 这个参数表示退出当前 Shell 时，将操作历史追加到历史文件中。这个参数默认是打开

的，如果使用下面的命令将其关闭，那么当前 Shell 的操作历史将替换掉整个历史文件。

```
1. $ shopt -u histappend
```

(3) -q

`-q` 的作用也是查询某个参数是否打开，但不是直接输出查询结果，而是通过命令的执行状态 (`$?`) 表示查询结果。如果状态为 `0` ，表示该参数打开；如果为 `1` ，表示该参数关闭。

```
1. $ shopt -q globstar
2. $ echo $?
3. 1
```

上面命令查询 `globstar` 参数是否打开。返回状态为 `1` ，表示该参数是关闭的。

这个用法主要用于脚本，供 `if` 条件结构使用。

```
1. if shopt -q globstar; then
2.   ...
3. fi
```

参考链接

- [The Set Builtin](#)
- [Safer bash scripts with 'set -euxo pipefail'](#)
- [Writing Robust Bash Shell Scripts](#)

脚本除错

本章介绍如何对 Shell 脚本除错。

常见错误

编写 Shell 脚本的时候，一定要考虑到命令失败的情况，否则很容易出错。

```
1. #! /bin/bash
2.
3. dir_name=/path/not/exist
4.
5. cd $dir_name
6. rm *
```

上面脚本中，如果目录 `$dir_name` 不存在，`cd $dir_name` 命令就会执行失败。这时，就不会改变当前目录，脚本会继续执行下去，导致 `rm *` 命令删光当前目录的文件。

如果改成下面的样子，也会有问题。

```
1. cd $dir_name && rm *
```

上面脚本中，只有 `cd $dir_name` 执行成功，才会执行 `rm *`。但是，如果变量 `$dir_name` 为空，`cd` 就会进入用户主目录，从而删光用户主目录的文件。

下面的写法才是正确的。

```
1. [[ -d $dir_name ]] && cd $dir_name && rm *
```

上面代码中，先判断目录 `$dir_name` 是否存在，然后才执行其他操作。

如果不放心删除什么文件，可以先打印出来看一下。

```
1. [[ -d $dir_name ]] && cd $dir_name && echo rm *
```

上面命令中，`echo rm *` 不会删除文件，只会打印出来要删除的文件。

bash 的 `-x` 参数

`bash` 的 `-x` 参数可以在执行每一行命令之前，打印该命令。这样就不用自己输出执行的命令，一旦出错，比较容易追查。

下面是一个脚本 `script.sh`。


```
1. # script.sh
2. echo hello world
```

加上 `-x` 参数，执行每条命令之前，都会显示该命令。

```
1. $ bash -x script.sh
2. + echo hello world
3. hello world
```

上面例子中，行首为 `+` 的行，显示该行是所要执行的命令，下一行才是该命令的执行结果。

下面再看一个 `-x` 写在脚本内部的例子。

```
1. #! /bin/bash -x
2. # trouble: script to demonstrate common errors
3.
4. number=1
5. if [ $number = 1 ]; then
6.     echo "Number is equal to 1."
7. else
8.     echo "Number is not equal to 1."
9. fi
```

上面的脚本执行之后，会输出每一行命令。

```
1. $ trouble
2. + number=1
3. + '[' 1 = 1 ']'
4. + echo 'Number is equal to 1.'
5. Number is equal to 1.
```

输出的命令之前的 `+` 号，是由系统变量 `PS4` 决定，可以修改这个变量。

```
1. $ export PS4='$LINENO + '
2. $ trouble
3. 5 + number=1
4. 7 + '[' 1 = 1 ']'
5. 8 + echo 'Number is equal to 1.'
6. Number is equal to 1.
```

另外，`set` 命令也可以设置 Shell 的行为参数，有利于脚本除错，详见《set 命令》一章。

环境变量

有一些环境变量常用于除错。

LINENO

变量 `LINENO` 返回它在脚本里面的行号。

```
1. #!/bin/bash
2.
3. echo "This is line $LINENO"
```

执行上面的脚本 `test.sh`，`$LINENO` 会返回 `3`。

```
1. $ ./test.sh
2. This is line 3
```

FUNCNAME

变量 `FUNCNAME` 返回一个数组，内容是当前的函数调用堆栈。该数组的0号成员是当前调用的函数，1号成员是调用当前函数的函数，以此类推。

```
1. #!/bin/bash
2.
3. function func1()
4. {
5.     echo "func1: FUNCNAME0 is ${FUNCNAME[0]}"
6.     echo "func1: FUNCNAME1 is ${FUNCNAME[1]}"
7.     echo "func1: FUNCNAME2 is ${FUNCNAME[2]}"
8.     func2
9. }
10.
11. function func2()
12. {
13.     echo "func2: FUNCNAME0 is ${FUNCNAME[0]}"
14.     echo "func2: FUNCNAME1 is ${FUNCNAME[1]}"
15.     echo "func2: FUNCNAME2 is ${FUNCNAME[2]}"
16. }
17.
18. func1
```

执行上面的脚本 `test.sh`，结果如下。

```
1. $ ./test.sh
2. func1: FUNCNAME0 is func1
3. func1: FUNCNAME1 is main
4. func1: FUNCNAME2 is
5. func2: FUNCNAME0 is func2
6. func2: FUNCNAME1 is func1
7. func2: FUNCNAME2 is main
```

上面例子中，执行 `func1` 时，变量 `FUNCNAME` 的0号成员是 `func1`，1号成员是调用 `func1` 的主脚本 `main`。执行 `func2` 时，变量 `FUNCNAME` 的0号成员是 `func2`，1号成员是调用 `func2` 的 `func1`。

BASH_SOURCE

变量 `BASH_SOURCE` 返回一个数组，内容是当前的脚本调用堆栈。该数组的0号成员是当前执行的脚本，1号成员是调用当前脚本的脚本，以此类推，跟变量 `FUNCNAME` 是一一对应关系。

下面有两个子脚本 `lib1.sh` 和 `lib2.sh`。

```
1. # lib1.sh
2. function func1()
3. {
4.     echo "func1: BASH_SOURCE0 is ${BASH_SOURCE[0]}"
5.     echo "func1: BASH_SOURCE1 is ${BASH_SOURCE[1]}"
6.     echo "func1: BASH_SOURCE2 is ${BASH_SOURCE[2]}"
7.     func2
8. }
```

```
1. # lib2.sh
2. function func2()
3. {
4.     echo "func2: BASH_SOURCE0 is ${BASH_SOURCE[0]}"
5.     echo "func2: BASH_SOURCE1 is ${BASH_SOURCE[1]}"
6.     echo "func2: BASH_SOURCE2 is ${BASH_SOURCE[2]}"
7. }
```

然后，主脚本 `main.sh` 调用上面两个子脚本。

```
1. #!/bin/bash
2. # main.sh
3.
4. source lib1.sh
5. source lib2.sh
6.
7. func1
```

执行主脚本 `main.sh`，会得到下面的结果。

```
1. $ ./main.sh
2. func1: BASH_SOURCE0 is lib1.sh
3. func1: BASH_SOURCE1 is ./main.sh
4. func1: BASH_SOURCE2 is
5. func2: BASH_SOURCE0 is lib2.sh
6. func2: BASH_SOURCE1 is lib1.sh
7. func2: BASH_SOURCE2 is ./main.sh
```

上面例子中，执行函数 `func1` 时，变量 `BASH_SOURCE` 的0号成员是 `func1` 所在的脚本 `lib1.sh`，1号成员是主脚本 `main.sh`；执行函数 `func2` 时，变量 `BASH_SOURCE` 的0号成员是 `func2` 所在的脚本 `lib2.sh`，1号成员是调用 `func2` 的脚本 `lib1.sh`。

BASH_LINENO

变量 `BASH_LINENO` 返回一个数组，内容是每一轮调用对应的行号。`${BASH_LINENO[$i]}` 跟 `${FUNCNAME[$i]}` 是一一对应关系，表示 `${FUNCNAME[$i]}` 在调用它的脚本文件 `${BASH_SOURCE[$i+1]}` 里面的行号。

下面有两个子脚本 `lib1.sh` 和 `lib2.sh`。

```
1. # lib1.sh
2. function func1()
3. {
4.     echo "func1: BASH_LINENO is ${BASH_LINENO[0]}"
5.     echo "func1: FUNCNAME is ${FUNCNAME[0]}"
6.     echo "func1: BASH_SOURCE is ${BASH_SOURCE[1]}"
7.
8.     func2
9. }
```

```
1. # lib2.sh
2. function func2()
3. {
4.     echo "func2: BASH_LINENO is ${BASH_LINENO[0]}"
5.     echo "func2: FUNCNAME is ${FUNCNAME[0]}"
6.     echo "func2: BASH_SOURCE is ${BASH_SOURCE[1]}"
7. }
```

然后，主脚本 `main.sh` 调用上面两个子脚本。

```
1. #!/bin/bash
2. # main.sh
3.
4. source lib1.sh
5. source lib2.sh
6.
7. func1
```

执行主脚本 `main.sh`，会得到下面的结果。

```
1. $ ./main.sh
2. func1: BASH_LINENO is 7
3. func1: FUNCNAME is func1
4. func1: BASH_SOURCE is main.sh
5. func2: BASH_LINENO is 8
6. func2: FUNCNAME is func2
7. func2: BASH_SOURCE is lib1.sh
```

上面例子中，函数 `func1` 是在 `main.sh` 的第7行调用，函数 `func2` 是在 `lib1.sh` 的第8行调用的。

mktemp 命令，trap 命令

Bash 脚本有时需要创建临时文件或临时目录。常见的做法是，在 `/tmp` 目录里面创建文件或目录，这样做有很多弊端，使用 `mktemp` 命令是最安全的做法。

临时文件的安全问题

直接创建临时文件，尤其在 `/tmp` 目录里面，往往会导致安全问题。

首先，`/tmp` 目录是所有人可读写的，任何用户都可以往该目录里面写文件。创建的临时文件也是所有人可读的。

```
1. $ touch /tmp/info.txt
2. $ ls -l /tmp/info.txt
3. -rw-r--r-- 1 ruanyf ruanyf 0 12月 28 17:12 /tmp/info.txt
```

上面命令在 `/tmp` 目录直接创建文件，该文件默认是所有人可读的。

其次，如果攻击者知道临时文件的文件名，他可以创建符号链接，链接到临时文件，可能导致系统运行异常。攻击者也可能向脚本提供一些恶意数据。因此，临时文件最好使用不可预测、每次都不一样的文件名，防止被利用。

最后，临时文件使用完毕，应该删除。但是，脚本意外退出时，往往会忽略清理临时文件。

生成临时文件应该遵循下面的规则。

- 创建前检查文件是否已经存在。
- 确保临时文件已成功创建。
- 临时文件必须有权限的限制。
- 临时文件要使用不可预测的文件名。
- 脚本退出时，要删除临时文件（使用 `trap` 命令）。

mktemp 命令的用法

`mktemp` 命令就是为安全创建临时文件而设计的。虽然在创建临时文件之前，它不会检查临时文件是否存在，但是它支持唯一文件名和清除机制，因此可以减轻安全攻击的风险。

直接运行 `mktemp` 命令，就能生成一个临时文件。

```
1. $ mktemp
2. /tmp/tmp.4GcsWSG4vj
3.
4. $ ls -l /tmp/tmp.4GcsWSG4vj
5. -rw----- 1 ruanyf ruanyf 0 12月 28 12:49 /tmp/tmp.4GcsWSG4vj
```

上面命令中，`mktemp` 命令生成的临时文件名是随机的，而且权限是只有用户本人可读写。

Bash 脚本使用 `mktemp` 命令的用法如下。

```
1. #!/bin/bash
2.
3. TMPFILE=$(mktemp)
4. echo "Our temp file is $TMPFILE"
```

为了确保临时文件创建成功，`mktemp` 命令后面最好使用 `OR` 运算符（`||`），保证创建失败时退出脚本。

```
1. #!/bin/bash
2.
3. TMPFILE=$(mktemp) || exit 1
4. echo "Our temp file is $TMPFILE"
```

为了保证脚本退出时临时文件被删除，可以使用 `trap` 命令指定退出时的清除操作。

```
1. #!/bin/bash
2.
3. trap 'rm -f "$TMPFILE"' EXIT
4.
5. TMPFILE=$(mktemp) || exit 1
6. echo "Our temp file is $TMPFILE"
```

mktemp 命令的参数

`-d` 参数可以创建一个临时目录。

```
1. $ mktemp -d
2. /tmp/tmp.Wcau5UjmN6
```

`-p` 参数可以指定临时文件所在的目录。默认是使用 `$TMPDIR` 环境变量指定的目录，如果这个变量没设置，那么使用 `/tmp` 目录。

```
1. $ mktemp -p /home/ruanyf/
2. /home/ruanyf/tmp.F0KEtvs2H3
```

`-t` 参数可以指定临时文件的文件名模板，模板的末尾必须至少包含三个连续的 `x` 字符，表示随机字符，建议至少使用六个 `x`。默认的文件名模板是 `tmp.` 后接十个随机字符。

```
1. $ mktemp -t mytemp.XXXXXXX
2. /tmp/mytemp.yZ1HgZV
```

trap 命令

`trap` 命令用来在 Bash 脚本中响应系统信号。

最常见的系统信号就是 `SIGINT`（中断），即按 `Ctrl + C` 所产生的信号。`trap` 命令的 `-l` 参数，可以列出所

有的系统信号。

```

1. $ trap -l
2.  1) SIGHUP      2) SIGINT      3) SIGQUIT      4) SIGILL      5) SIGTRAP
3.  6) SIGABRT     7) SIGBUS     8) SIGFPE     9) SIGKILL    10) SIGUSR1
4. 11) SIGSEGV    12) SIGUSR2    13) SIGPIPE    14) SIGALRM    15) SIGTERM
5. 16) SIGSTKFLT   17) SIGCHLD   18) SIGCONT    19) SIGSTOP    20) SIGTSTP
6. 21) SIGTTIN     22) SIGTTOU    23) SIGURG     24) SIGXCPU    25) SIGXFSZ
7. 26) SIGVTALRM   27) SIGPROF    28) SIGWINCH   29) SIGIO      30) SIGPWR
8. 31) SIGSYS      34) SIGRTMIN   35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
9. 38) SIGRTMIN+4  39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
10. 43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
11. 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
12. 53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
13. 58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2
14. 63) SIGRTMAX-1 64) SIGRTMAX

```

`trap` 的命令格式如下。

```
1. $ trap [动作] [信号1] [信号2] ...
```

上面代码中，“动作”是一个 Bash 命令，“信号”常用的有以下几个。

- HUP: 编号1, 脚本与所在的终端脱离联系。
- INT: 编号2, 用户按下 Ctrl + C, 意图让脚本中止运行。
- QUIT: 编号3, 用户按下 Ctrl + 斜杠, 意图退出脚本。
- KILL: 编号9, 该信号用于杀死进程。
- TERM: 编号15, 这是 `kill` 命令发出的默认信号。
- EXIT: 编号0, 这不是系统信号, 而是 Bash 脚本特有的信号, 不管什么情况, 只要退出脚本就会产生。

`trap` 命令响应 `EXIT` 信号的写法如下。

```
1. $ trap 'rm -f "$TMPFILE"' EXIT
```

上面命令中, 脚本遇到 `EXIT` 信号时, 就会执行 `rm -f "$TMPFILE"`。

`trap` 命令的常见使用场景, 就是在 Bash 脚本中指定退出时执行的清理命令。

```

1. #!/bin/bash
2.
3. trap 'rm -f "$TMPFILE"' EXIT
4.
5. TMPFILE=$(mktemp) || exit 1
6. ls /etc > $TMPFILE
7. if grep -qi "kernel" $TMPFILE; then
8.   echo 'find'
9. fi

```

上面代码中, 不管是脚本正常执行结束, 还是用户按 Ctrl + C 终止, 都会产生 `EXIT` 信号, 从而触发删除临时文件。

注意， `trap` 命令必须放在脚本的开头。否则，它上方的任何命令导致脚本退出，都不会被它捕获。

如果 `trap` 需要触发多条命令，可以封装一个 Bash 函数。

```
1. function egress {  
2.     command1  
3.     command2  
4.     command3  
5. }  
6.  
7. trap egress EXIT
```

参考链接

- [Working with Temporary Files and Directories in Shell Scripts](#), Steven Vona
- [Using Trap to Exit Bash Scripts Cleanly](#)
- [Sending and Trapping Signals](#)

Bash 启动环境

Session

用户每次使用 Shell，都会开启一个与 Shell 的 Session（对话）。

Session 有两种类型：登录 Session 和非登录 Session，也可以叫做 login shell 和 non-login shell。

登录 Session

登录 Session 是用户登录系统以后，系统为用户开启的原始 Session，通常需要用户输入用户名和密码进行登录。

登录 Session 一般进行整个系统环境的初始化，启动的初始化脚本依次如下。

- `/etc/profile`：所有用户的全局配置脚本。
- `/etc/profile.d` 目录里面所有 `.sh` 文件
- `~/.bash_profile`：用户的个人配置脚本。如果该脚本存在，则执行完就不再往下执行。
- `~/.bash_login`：如果 `~/.bash_profile` 没找到，则尝试执行这个脚本（C shell 的初始化脚本）。如果该脚本存在，则执行完就不再往下执行。
- `~/.profile`：如果 `~/.bash_profile` 和 `~/.bash_login` 都没找到，则尝试读取这个脚本（Bourne shell 和 Korn shell 的初始化脚本）。

Linux 发行版更新的时候，会更新 `/etc` 里面的文件，比如 `/etc/profile`，因此不要直接修改这个文件。如果想修改所有用户的登陆环境，就在 `/etc/profile.d` 目录里面新建 `.sh` 脚本。

如果想修改你个人的登录环境，一般是写在 `~/.bash_profile` 里面。下面是一个典型的 `.bash_profile` 文件。

```
1. # .bash_profile
2. PATH=/sbin:/usr/sbin:/bin:/usr/bin:/usr/local/bin
3. PATH=$PATH:$HOME/bin
4.
5. SHELL=/bin/bash
6. MANPATH=/usr/man:/usr/X11/man
7. EDITOR=/usr/bin/vi
8. PS1='\h:\w\$ '
9. PS2='> '
10.
11. if [ -f ~/.bashrc ]; then
12.     . ~/.bashrc
13. fi
14.
15. export PATH
16. export EDITOR
```

可以看到，这个脚本定义了一些最基本的环境变量，然后执行了 `~/.bashrc`。

`bash` 命令的 `--login` 参数，会强制执行登录 Session 会执行的脚本。

```
1. $ bash --login
```

`bash` 命令的 `--noprofile` 参数，会跳过上面这些 Profile 脚本。

```
1. $ bash --noprofile
```

非登录 Session

非登录 Session 是用户进入系统以后，手动新建的 Session，这时不会进行环境初始化。比如，在命令行执行 `bash` 命令，就会新建一个非登录 Session。

非登录 Session 的初始化脚本依次如下。

- `/etc/bash.bashrc`：对全体用户有效。
- `~/.bashrc`：仅对当前用户有效。

对用户来说，`~/.bashrc` 通常是最重要的脚本。非登录 Session 默认会执行它，而登陆 Session 一般也会通过调用执行它。由于每次执行 Bash 脚本，都会新建一个非登录 Session，所以 `~/.bashrc` 也是每次执行脚本都会执行的。

`bash` 命令的 `--norc` 参数，可以禁止在非登录 Session 执行 `~/.bashrc` 脚本。

```
1. $ bash --norc
```

`bash` 命令的 `--rcfile` 参数，指定另一个脚本代替 `.bashrc`。

```
1. $ bash --rcfile testrc
```

.bash_logout

`~/.bash_logout` 脚本在每次退出 Session 时执行，通常用来做一些清理工作和记录工作，比如删除临时文件，记录用户在本次 Session 花费的时间。

如果没有退出时要执行的命令，这个文件也可以不存在。

启动选项

为了方便 Debug，有时在启动 Bash 的时候，可以加上启动参数。

- `-n`：不运行脚本，只检查是否有语法错误。
- `-v`：输出每一行语句运行结果前，会先输出该行语句。
- `-x`：每一个命令处理完以后，先输出该命令，再进行下一个命令的处理。

```
1. $ bash -n scriptname
```

```
2. $ bash -v scriptname
3. $ bash -x scriptname
```

键盘绑定

Bash 允许用户定义自己的快捷键。全局的键盘绑定文件默认为 `/etc/inputrc`，你可以在主目录创建自己的键盘绑定文件 `.inputrc` 文件。如果定义了这个文件，需要在其中加入下面这行，保证全局绑定不会被遗漏。

```
1. $include /etc/inputrc
```

`.inputrc` 文件里面的快捷键，可以像这样定义，`"\C-t":"pwd\n"` 表示将 `Ctrl + t` 绑定为运行 `pwd` 命令。

命令提示符

用户进入 Bash 以后，Bash 会显示一个命令提示符，用来提示用户在该位置后面输入命令。

环境变量 PS1

命令提示符通常是美元符号 `$`，对于根用户则是井号 `#`。这个符号是环境变量 `PS1` 决定的，执行下面的命令，可以看到当前命令提示符的定义。

```
1. $ echo $PS1
```

Bash 允许用户自定义命令提示符，只要改写这个变量即可。改写后的 `PS1`，可以放在用户的 Bash 配置文件 `.bashrc` 里面，以后新建 Bash 对话时，新的提示符就会生效。要在当前窗口看到修改后的提示符，可以执行下面的命令。

```
1. $ source ~/.bashrc
```

命令提示符的定义，可以包含特殊的转义字符，表示特定内容。

- `\a`：响铃，计算机发出一记声音。
- `\d`：以星期、月、日格式表示当前日期，例如“Mon May 26”。
- `\h`：本机的主机名。
- `\H`：完整的主机名。
- `\j`：运行在当前 Shell 会话的工作数。
- `\l`：当前终端设备名。
- `\n`：一个换行符。
- `\r`：一个回车符。
- `\s`：Shell 的名称。
- `\t`：24小时制的 `hours:minutes:seconds` 格式表示当前时间。
- `\T`：12小时制的当前时间。
- `\@`：12小时制的 `AM/PM` 格式表示当前时间。
- `\A`：24小时制的 `hours:minutes` 表示当前时间。
- `\u`：当前用户名。
- `\v`：Shell 的版本号。
- `\V`：Shell 的版本号和发布号。
- `\w`：当前的工作路径。
- `\W`：当前目录名。
- `\!`：当前命令在命令历史中的编号。
- `\#`：当前 shell 会话中的命令数。
- `\$`：普通用户显示为 `$` 字符，根用户显示为 `#` 字符。
- `\[`：非打印字符序列的开始标志。
- `\]`：非打印字符序列的结束标志。

举例来说，`[\u@\h \w]\$` 这个提示符定义，显示出来就是 `[user@host ~]$`（具体的显示内容取决于你的系统）。

```
1. [user@host ~]$ echo $PS1
2. [\u@\h \w]\$
```

改写 `PS1` 变量，就可以改变这个命令提示符。

```
1. $ PS1="\A \h \$ "
2. 17:33 host $
```

注意，`$` 后面最好跟一个空格，这样的话，用户的输入与提示符就不会连在一起。

颜色

默认情况下，命令提示符是显示终端预定义的颜色。Bash 允许自定义提示符颜色。

使用下面的代码，可以设定其后文本的颜色。

- `\033[0;30m` : 黑色
- `\033[1;30m` : 深灰色
- `\033[0;31m` : 红色
- `\033[1;31m` : 浅红色
- `\033[0;32m` : 绿色
- `\033[1;32m` : 浅绿色
- `\033[0;33m` : 棕色
- `\033[1;33m` : 黄色
- `\033[0;34m` : 蓝色
- `\033[1;34m` : 浅蓝色
- `\033[0;35m` : 粉红
- `\033[1;35m` : 浅粉色
- `\033[0;36m` : 青色
- `\033[1;36m` : 浅青色
- `\033[0;37m` : 浅灰色
- `\033[1;37m` : 白色

举例来说，如果要提示符设为红色，可以将 `PS1` 设成下面的代码。

```
1. PS1='\[\033[0;31m\]<\u@\h \w>\$'
```

但是，上面这样设置以后，用户在提示符后面输入的文本也是红色的。为了解决这个问题，可以在结尾添加另一个特殊代码 `\[\033[00m\]`，表示将其后的文本恢复到默认颜色。

```
1. PS1='\[\033[0;31m\]<\u@\h \w>\$'\[\033[00m\]'
```

除了设置前景颜色，Bash 还允许设置背景颜色。

- `\033[0;40m` : 蓝色
- `\033[1;44m` : 黑色

- `\033[0;41m` : 红色
- `\033[1;45m` : 粉红
- `\033[0;42m` : 绿色
- `\033[1;46m` : 青色
- `\033[0;43m` : 棕色
- `\033[1;47m` : 浅灰色

下面是一个带有红色背景的提示符。

```
1. PS1='\[\033[0;41m\]<\u@\h \w>\$\' \[\033[0m\] '
```

环境变量 PS2, PS3, PS4

除了 `PS1` , Bash 还提供了提示符相关的另外三个环境变量。

环境变量 `PS2` 是命令行折行输入时系统的提示符, 默认为 `>` 。

```
1. $ echo "hello"
2. > world"
```

上面命令中, 输入 `hello` 以后按下回车键, 系统会提示继续输入。这时, 第二行显示的提示符就是 `PS2` 定义的 `>` 。

环境变量 `PS3` 是使用 `select` 命令时, 系统输入菜单的提示符。

环境变量 `PS4` 默认为 `+` 。它是使用 Bash 的 `-x` 参数执行脚本时, 每一行命令在执行前都会先打印出来, 并且在行首出现的那个提示符。

比如下面是脚本 `test.sh` 。

```
1. #!/bin/bash
2.
3. echo "hello world"
```

使用 `-x` 参数执行这个脚本。

```
1. $ bash -x test.sh
2. + echo 'hello world'
3. hello world
```

上面例子中, 输出的第一行前面有一个 `+` , 这就是变量 `PS4` 定义的。

归档和备份

gzip

gzip 程序用来压缩文件，原文件的压缩版（添加 `gz` 后缀名）会替代原文件。gunzip 程序用来还原压缩版本。

```
1. $ gzip foo.txt
2. $ gunzip foo.txt.gz
```

`gzip` 的参数如下。

- `-c` 把输出写入到标准输出，并且保留原始文件。也有可能用`-stdout` 和`-to-stdout` 选项来指定。
- `-d` 解压缩。正如 `gunzip` 命令一样。也可以用`-decompress` 或者`-uncompress` 选项来指定。
- `-f` 强制压缩，即使原始文件的压缩文件已经存在了，也要执行。也可以用`-force` 选项来指定。
- `-h` 显示用法信息。也可用`-help` 选项来指定。
- `-l` 列出每个被压缩文件的压缩数据。也可用`-list` 选项。
- `-r` 若命令的一个或多个参数是目录，则递归地压缩目录中的文件。也可用`-recursive` 选项来指定。
- `-t` 测试压缩文件的完整性。也可用`-test` 选项来指定。
- `-v` 显示压缩过程中的信息。也可用`-verbose` 选项来指定。
- `-number` 设置压缩指数。number 是一个在1（最快，最小压缩）到9（最慢，最大压缩）之间的整数。 数值1和9也可以各自用`-fast` 和`-best` 选项来表示。默认值是整数6。

下面是一些例子。

```
1. # 查看解压缩后的内容
2. $ gunzip -c foo.txt | less
```

`zcat` 程序等同于带有`-c` 选项的 `gunzip` 命令。它可以像 `cat` 命令那样，用来查看 `gzip` 压缩文件。

```
1. $ zcat foo.txt.gz | less
```

bzip2

`bzip2` 程序与 `gzip` 程序相似，但是使用了不同的压缩算法，舍弃了压缩速度，实现了更高的压缩级别。在大多数情况下，它的工作模式等同于 `gzip` 。 由 `bzip2` 压缩的文件，用扩展名 `.bz2` 表示。

```
1. $ bzip2 foo.txt
2. $ bunzip2 foo.txt.bz2
```

gzip程序的所有选项（除了 `-r` ），bzip2 程序同样也支持。同样有 `bunzip2` 和 `bzcat` 程序来解压缩文件。bzip2 文件也带有 `bzip2recover` 程序，其会 试图恢复受损的 `.bz2` 文件。

zip

`zip` 程序既是压缩工具，也是一个打包工具，读取和写入 .zip 文件。

```
1. $ zip options zipfile file...
```

它的用法如下。

```
1. # 将指定目录压缩成zip文件
2. $ zip -r playground.zip playground
```

`zip` 与 `tar` 命令有一个相反之处。如果压缩文件已存在，其将被更新而不是被替代。这意味着会保留此文件包，但是会添加新文件，同时替换匹配的文件。

解压使用 `unzip` 命令。

```
1. $ unzip ../playground.zip
```

`unzip` 命令的参数如下。

- `-l` 列出文件包中的内容而不解压
- `-v` 显示冗余信息
- `-p` 输出发送到标准输出

```
1. $ unzip -p ls-etc.zip | less
```

tar

`tar` 是tape archive的简称，原来是一款制作磁带备份的工具，现在主要用于打包。一个 `tar` 包可以由一组独立的文件，一个或者多个目录，或者两者混合体组成。

`tar` 程序的语法如下。

```
1. $ tar mode[options] pathname...
```

`tar`支持以下模式。

- `c` 表示create，为文件和 / 或目录列表创建归档文件。
- `x` 抽取归档文件。
- `r` 追加具体的路径到归档文件的末尾。
- `t` 列出归档文件的内容。

支持的参数如下。

- `f` 表示file，用来指定生成的文件。

模式和参数可以写在一起，而且不需要开头的短横线。注意，必须首先指定模式，然后才是其它的选项。

```
1. # 创建子目录的tar包
2. $ tar cf playground.tar playground
3.
4. # 查看tar包内容
5. $ tar tf playground.tar
6.
7. # 查看更详细的列表信息
8. $ tar tvf playground.tar
9.
10. # 还原归档文件
11. $ tar xf playground.tar
12.
13. # 还原单个文件
14. $ tar xf archive.tar pathname
15.
16. # 还原文件到指定目录
17. $ tar xvf archive.tar -C /home/me/
18.
19. # 追加文件
20. $ tar rf archive.tar file.txt
21.
22. # 验证归档文件内容是否正确
23. $ tar tvfW archive.tar
24.
25. # 支持通配符
26. $ tar xf ../playground2.tar --wildcards 'home/me/playground/*.txt'
```

注意，`tar` 命令还原的时候，总是还原为相对路径。如果归档的时候，保存的是绝对路径，那么还原的时候，这个绝对路径会整个变成相对路径。

`find` 命令可以与 `tar` 命令配合使用。

```
1. $ find playground -name 'file.txt' -exec tar rf playground.tar '{}' '+'
```

上面的命令先用 `find` 程序找到所有名为 `file.txt` 的文件，然后使用追加模式（`r`）的 `tar` 命令，把匹配的文件添加到归档文件 `playground.tar` 里面。

这种 `tar` 和 `find` 的配合使用，可以创建逐渐增加的目录树或者整个系统的备份。通过 `find` 命令匹配新于某个时间戳的文件，我们就能够创建一个归档文件，其只包含新于上一个 `tar` 包的文件。

`tar`支持压缩功能。

```
1. # 打成gzip压缩包
2. $ tar czvf assets.tar.gz dist
3.
4. # 打成bz2压缩包
5. $ tar cvfj assets.tar.bz2 dist
6.
7. # 解压 tar.gz 文件
```

```
8. $ tar xzv archive.tar.gz
9. $ tar xvf archive.tar.gz
10.
11. # 解压bz2压缩包
12. $ tar xvf archive.tar.bz2
13.
14. # 显示gzip压缩包内容
15. $ tar tvf archive.tar.gz
16.
17. # 显示bz2压缩包内容
18. $ tar tvf archive.tar.bz2
19.
20. # 从gzip压缩包取出单个文件
21. $ tar zxvf archive.tar.gz file.txt
22.
23. # 从bz2压缩包取出单个文件
24. $ tar jxvf archive.tar.bz2 file.txt
25.
26. # 按通配符取出文件
27. $ tar zxvf archive.tar.gz --wildcards '*.php'
28. $ tar jxvf archive.tar.bz2 --wildcards '*.php'
29.
30. # 追加文件到压缩包
31. $ tar rvf archive.tar.gz xyz.txt
32. $ tar rvf archive.tar.bz2 xyz.txt
```

rsync

`rsync` 命令用于在多个目录之间、或者本地与远程目录之间同步。字母 `r` 表示 `remote`。

```
1. $ rsync options source destination
```

`source` 和 `destination` 是下列选项之一：

- 一个本地文件或目录
- 一个远端文件或目录，以 `[user@]host:path` 的形式存在
- 一个远端 `rsync` 服务器，由 `rsync://[user@]host[:port]/path` 指定

注意 `source` 和 `destination` 两者之一必须是本地文件。`rsync` 不支持远端到远端的复制。

`rsync` 命令的参数如下。

- `-a` 递归和保护文件属性
- `-v` 冗余输出
- `--delete` 删除可能在备份设备中已经存在但却不再存在于源设备中的文件
- `--rsh=ssh` 使用 `ssh` 程序作为远程 `shell`，目的地必须标注主机名。

```
1. # 同步两个本地目录
2. $ rsync -av playground foo
3.
```

```
4. # 删除源设备不存在的文件
5. $ sudo rsync -av --delete /etc /home /usr/local /media/BigDisk/backup
6.
7. # 远程同步
8. $ sudo rsync -av --delete --rsh=ssh /etc /home /usr/local remote-sys:/backup
9.
10. # 与远程rsync主机同步
11. $ rsync -av --delete rsync://rsync.gtlib.gatech.edu/path/to/oss fedora-devel
```

异步任务

Bash脚本有时候需要同时执行多个任务。通常这涉及到启动一个脚本，依次，启动一个或多个子脚本来执行额外的任务，而父脚本继续运行。然而，当一系列脚本 以这种方式运行时，要保持父子脚本之间协调工作，会有一些问题。也就是说，若父脚本或子脚本依赖于另一方，并且 一个脚本必须等待另一个脚本结束任务之后，才能完成它自己的任务，这应该怎么办？

bash 有一个内置命令，能帮助管理诸如此类的异步执行的任务。wait 命令导致一个父脚本暂停运行，直到一个 特定的进程（例如，子脚本）运行结束。

首先我们将演示一下 wait 命令的用法。为此，我们需要两个脚本，一个父脚本：

```
1. #!/bin/bash
2. # async-parent : Asynchronous execution demo (parent)
3. echo "Parent: starting..."
4. echo "Parent: launching child script..."
5. async-child &
6. pid=$!
7. echo "Parent: child (PID= $pid) launched."
8. echo "Parent: continuing..."
9. sleep 2
10. echo "Parent: pausing to wait for child to finish..."
11. wait $pid
12. echo "Parent: child is finished. Continuing..."
13. echo "Parent: parent is done. Exiting."
```

和一个子脚本：

```
1. #!/bin/bash
2. # async-child : Asynchronous execution demo (child)
3. echo "Child: child is running..."
4. sleep 5
5. echo "Child: child is done. Exiting."
```

在这个例子中，我们看到该子脚本是非常简单的。真正的操作通过父脚本完成。在父脚本中，子脚本被启动， 并被放置到后台运行。子脚本的进程 ID 记录在 pid 变量中，这个变量的值是 \$! shell 参数的值，它总是 包含放到后台执行的最后一个任务的进程 ID 号。

父脚本继续，然后执行一个以子进程 PID 为参数的 wait 命令。这就导致父脚本暂停运行，直到子脚本退出， 意味着父脚本结束。

当执行后，父子脚本产生如下输出：

```
1. $ async-parent
2. Parent: starting...
3. Parent: launching child script...
4. Parent: child (PID= 6741) launched.
5. Parent: continuing...
```

```
6. Child: child is running...
7. Parent: pausing to wait for child to finish...
8. Child: child is done. Exiting.
9. Parent: child is finished. Continuing...
10. Parent: parent is done. Exiting.
```

标准I/O

echo

`echo` 命令用于将指定内容输出到显示屏（标准输出）。

```
1. $ echo this is a test
2. this is a test
```

它的参数如下。

- `-e` 解释转义字符。
- `-n` 不输出行尾的换行符

```
1. $ echo "a\nb"
2. a\nb
3.
4. $ echo -e "a\nb"
5. a
6. b
```

上面代码中，如果不加 `-e` 参数，`\n` 就会按字面形式输出；加了以后，就被解释成了换行符。

引号之中可以包括多个换行符，即可以输出多行文本。

```
1. echo "<HTML>
2.     <HEAD>
3.         <TITLE>Page Title</TITLE>
4.     </HEAD>
5.     <BODY>
6.         Page body.
7.     </BODY>
8. </HTML>"
```

`echo ' Page body. '`

read

`read` 命令被用来从标准输入读取单行数据。这个命令可以用来读取键盘输入，当使用重定向的时候，读取文件中的一行数据。

```
1. read [-options] [variable...]
```

上面的 `variable` 用来存储输入数值的一个或多个变量名。如果没有提供变量名，shell 变量 `REPLY` 会包含数据

行。

基本上，`read` 会把来自标准输入的字段赋值给具体的变量。

```
1. echo -n "Please enter an integer -> "
2. read int
```

`read` 可以给多个变量赋值。

```
1. #!/bin/bash
2. # read-multiple: read multiple values from keyboard
3. echo -n "Enter one or more values > "
4. read var1 var2 var3 var4 var5
5. echo "var1 = '$var1'"
6. echo "var2 = '$var2'"
7. echo "var3 = '$var3'"
8. echo "var4 = '$var4'"
9. echo "var5 = '$var5'"
```

上面脚本的用法如下。

```
1. $ read-multiple
2. Enter one or more values > a b c d e
3. var1 = 'a'
4. var2 = 'b'
5. var3 = 'c'
6. var4 = 'd'
7. var5 = 'e'
8.
9. $ read-multiple
10. Enter one or more values > a
11. var1 = 'a'
12. var2 = ''
13. var3 = ''
14. var4 = ''
15. var5 = ''
16.
17. $ read-multiple
18. Enter one or more values > a b c d e f g
19. var1 = 'a'
20. var2 = 'b'
21. var3 = 'c'
22. var4 = 'd'
23. var5 = 'e f g'
```

如果 `read` 命令接受到变量值数目少于期望的数字，那么额外的变量值为空，而多余的输入数据则会 被包含到最后一个变量中。

如果 `read` 命令之后没有列出变量名，则一个 `shell` 变量 `REPLY`，将会包含所有的输入。

```

1. #!/bin/bash
2. # read-single: read multiple values into default variable
3. echo -n "Enter one or more values > "
4. read
5. echo "REPLY = '$REPLY'"

```

上面脚本的输出结果如下。

```

1. $ read-single
2. Enter one or more values > a b c d
3. REPLY = 'a b c d'

```

read命令的参数如下。

- `-a array` 把输入赋值到数组 `array` 中，从索引号零开始。
- `-d delimiter` 用字符串 `delimiter` 中的第一个字符指示输入结束，而不是一个换行符。
- `-e` 使用 Readline 来处理输入。这使得与命令行相同的方式编辑输入。
- `-n num` 读取 `num` 个输入字符，而不是整行。
- `-p prompt` 为输入显示提示信息，使用字符串 `prompt`。
- `-r` Raw mode. 不把反斜杠字符解释为转义字符。
- `-s` Silent mode. 不会在屏幕上显示输入的字符。当输入密码和其它确认信息的时候，这会很有帮助。
- `-t seconds` 超时。几秒钟后终止输入。read 会返回一个非零退出状态，若输入超时。
- `-u fd` 使用文件描述符 `fd` 中的输入，而不是标准输入。

`-p` 的例子。

```

1. read -p "Enter one or more values > "
2. echo "REPLY = '$REPLY'"

```

`-t` 和 `-s` 的例子。

```

1. if read -t 10 -sp "Enter secret pass phrase > " secret_pass; then
2.     echo -e "\nSecret pass phrase = '$secret_pass'"

```

上面这个脚本提示用户输入一个密码，并等待输入10秒钟。如果在特定的时间内没有完成输入，则脚本会退出并返回一个错误。因为包含了一个 `-s` 选项，所以输入的密码不会出现在屏幕上。

Shell的内部变量 `IFS` 可以控制输入字段的分离。例如，这个 `/etc/passwd` 文件包含的数据行 使用冒号作为字段分隔符。通过把 `IFS` 的值更改为单个冒号，我们可以使用 `read` 读取 `/etc/passwd` 中的内容，并成功地把字段分给不同的变量。

```

1. #!/bin/bash
2. # read-ifs: read fields from a file
3. FILE=/etc/passwd
4. read -p "Enter a user name > " user_name
5. file_info=$(grep "^$user_name:" $FILE)
6. if [ -n "$file_info" ]; then
7.     IFS=":" read user pw uid gid name home shell <<< "$file_info"

```



```

8.     echo "User = '$user'"
9.     echo "UID = '$uid'"
10.    echo "GID = '$gid'"
11.    echo "Full Name = '$name'"
12.    echo "Home Dir. = '$home'"
13.    echo "Shell = '$shell'"
14. else
15.     echo "No such user '$user_name'" >&2
16.     exit 1
17. fi

```

Shell 允许在一个命令之前立即发生一个或多个变量赋值。这些赋值为跟随着的命令更改环境变量。 这个赋值的影响是暂时的；只是在命令存在期间改变环境变量。

虽然通常 `read` 命令接受标准输入，但是你不能这样做：

```
1. $ echo "foo" | read
```

我们期望这个命令能生效，但是它不能。这个命令将显示成功，但是 `REPLY` 变量 总是为空。为什么会这样？

答案与 `shell` 处理管道线的方式有关系。在 `bash` (和其它 `shells`, 例如 `sh`) 中, 管道线 会创建子 `shell`。它们是 `shell` 的副本, 且用来执行命令的环境变量在管道线中。 上面示例中, `read` 命令将在子 `shell` 中执行。

在类 Unix 的系统中, 子 `shell` 执行的时候, 会为进程创建父环境的副本。当进程结束 之后, 环境副本就会被破坏掉。这意味着一个子 `shell` 永远不能改变父进程的环境。`read` 赋值变量, 然后会变为环境的一部分。在上面的例子中, `read` 在它的子 `shell` 环境中, 把 `foo` 赋值给变量 `REPLY`, 但是当命令退出后, 子 `shell` 和它的环境将被破坏掉, 这样赋值的影响就会消失。

使用 `here` 字符串是解决此问题的一种方法。

下面是生成菜单的一个例子。

```

1. #!/bin/bash
2. # read-menu: a menu driven system information program
3. clear
4. echo "
5. Please Select:
6.
7.     1. Display System Information
8.     2. Display Disk Space
9.     3. Display Home Space Utilization
10.    0. Quit
11. "
12. read -p "Enter selection [0-3] > "
13.
14. if [[ $REPLY =~ ^[0-3]$ ]]; then
15.     if [[ $REPLY == 0 ]]; then
16.         echo "Program terminated."
17.         exit
18.     fi

```

```
19.     if [[ $REPLY == 1 ]]; then
20.         echo "Hostname: $HOSTNAME"
21.         uptime
22.         exit
23.     fi
24.     if [[ $REPLY == 2 ]]; then
25.         df -h
26.         exit
27.     fi
28.     if [[ $REPLY == 3 ]]; then
29.         if [[ $(id -u) -eq 0 ]]; then
30.             echo "Home Space Utilization (All Users)"
31.             du -sh /home/*
32.         else
33.             echo "Home Space Utilization ($USER)"
34.             du -sh $HOME
35.         fi
36.         exit
37.     fi
38. else
39.     echo "Invalid entry." >&2
40.     exit 1
41. fi
```

文件操作

cp

cp 命令用于将文件（或目录）拷贝到目的地。

```
1. # 拷贝单个文件
2. $ cp source dest
3.
4. # 拷贝多个文件
5. $ cp source1 source2 source3 dest
6.
7. # -i 目的地有同名文件时会提示确认
8. $ cp -i file1 file2
9.
10. # -r 递归拷贝, 将dir1拷贝到dir2, 完成后dir2生成一个子目录dir1
11. # dir2如果不存在, 将被创建
12. # 拷贝目录时, 该参数是必需的
13. $ cp -r dir1 dir2
14.
15. # -u --update 只拷贝目的地没有的文件, 或者比目的地同名文件更新的文件
16. $ cp -u *.html destination
```

其他参数

- **-a** 拷贝时保留所有属性, 包括所有者与权限
- **-v** 显示拷贝的详细信息

mkdir

mkdir 命令用于新建目录。

```
1. # 新建多个目录
2. $ mkdir dir1 dir2 dir3
```

mv

mv 命令用于将源文件移动到目的地。

```
1. # 移动单个文件
2. $ mv item1 item2
3.
4. # 移动多个文件
5. $ mv file1 file2 dir1
6.
```

```
7. # 将dir1拷贝进入dir2, 完成后dir2将多出一个子目录dir1
8. # 如果dir2不存在, 将会被创建
9. $ mv dir1 dir2
```

参数

- `-i` 覆盖已经存在的文件时, 会提示确认
- `-u` 只移动目的地不存在的文件, 或比目的地更新的文件

rm

`rm` 命令用于删除文件。

参数。

- `-i` 文件存在时, 会提示确认。
- `-r` 递归删除一个子目录
- `-f` 如果删除不存在的文件, 不报错
- `-v` 删除时展示详细信息

ln

`ln` 命令用于建立链接文件。

```
1. # 新建硬链接
2. $ ln file link
3.
4. # 新建软链接
5. $ ln -s item link
```

文件系统

pwd

pwd 命令显示列出当前所在的目录。

```
1. $ pwd
```

cd

cd 命令用来改变用户所在的目录。

```
1. # 进入用户的主目录
2. $ cd
3.
4. # 进入前一个工作目录
5. $ cd -
6.
7. # 进入指定用户的主目录
8. $ cd ~user_name
```

ls

ls 目录可以显示指定目录的内容。不加参数时，显示当前目录的内容。

```
1. $ ls
```

上面命令显示当前目录的内容。

ls 命令也可以显示指定文件是否存在。

```
1. $ ls foo.txt
2. foo.txt
```

-l 参数可以显示文件的详细信息。

```
1. $ ls -l foo.txt
2. -rw-rw-r-- 1 me me 0 2016-03-06 14:52 foo.txt
```

上面命令输出结果的第一栏，是文件的类型和权限。

文件类型分为以下几种。

- **-** 普通文件
- **d** 目录
- **l** 符号链接。注意，对于符号链接文件，剩余的文件属性总是“rwxrwxrwx”。
- **c** 字符设备文件，指按照字节流处理数据的设备，比如调制解调器。
- **b** 块设备文件，指按照数据块处理数据的设备，比如硬盘。

其他参数的用法。

```

1. # 显示多个目录的内容
2. $ ls ~ /usr
3.
4. # -a --all 显示隐藏文件
5. $ ls -a
6.
7. # -A 与-a类似，但是不显示当前目录和上一级目录两个点文件
8. $ ls -A
9.
10. # -l 显示详细信息
11. $ ls -l
12.
13. # -l 单列显示，每行只显示一个文件
14. $ ls -l
15.
16. # -d 显示当前目录本身，而不是它的内容
17. # 通常与-l配合使用，列出一个目录本身的详细信息
18. $ ls -dl
19.
20. # -F 目录名之后添加斜杠，可执行文件后面添加星号
21. $ ls -F
22.
23. # -h 与-l配合使用，将文件大小显示为人类可读的格式
24.
25. # -t 按文件修改时间排序，修改晚的排在前面
26. $ ls -t
27.
28. # -s 按文件大小排序，
29.
30. # --reverse 显示结果倒序排列
31. $ ls -lt --reverse

```

如果只显示一个目录里面的子目录，不显示文件，可以使用下面这些命令。

```

1. # 只显示常规目录
2. $ ls -d */
3. $ ls -F | grep /
4. $ ls -l | grep ^d
5. $ tree -dL 1
6.
7. # 只显示隐藏目录
8. $ ls -d .* /
9.

```

```
10. # 隐藏目录和非隐藏目录都显示
11. $ find -maxdepth 1 -type d
```

另一个简便方法是利用自动补全功能，先键入 `cd` 命令，然后连接两下 `tab` 键。

stat

`stat` 命令是加强版的 `ls` 命令，可以显示一个文件的详细信息。

```
1. $ stat timestamp
2. File: 'timestamp'
3. Size: 0 Blocks: 0 IO Block: 4096 regular empty file
4. Device: 803h/2051d Inode: 14265061 Links: 1
5. Access: (0644/-rw-r--r--) Uid: ( 1001/ me) Gid: ( 1001/ me)
6. Access: 2008-10-08 15:15:39.000000000 -0400
7. Modify: 2008-10-08 15:15:39.000000000 -0400
8. Change: 2008-10-08 15:15:39.000000000 -0400
```

touch

`touch` 用来设置或更新文件的访问，更改，和修改时间。然而，如果一个文件名参数是一个不存在的文件，则会创建一个空文件。

```
1. $ touch timestamp
```

上面命令创建了一个名为 `timestamp` 空文件。如果该文件已经存在，就会把它的修改时间设置为当前时间。

```
1. $ mkdir -p playground/dir-{00{1..9},0{10..99},100}
2. $ touch playground/dir-{00{1..9},0{10..99},100}/file-{A..Z}
```

上面的命令创建了一个包含一百个子目录，每个子目录中包含了26个空文件。

file

`file` 命令显示指定文件的类型。

```
1. $ file picture.jpg
2. picture.jpg: JPEG image data, JFIF standard 1.01
```

chmod

`chmod` 命令用于更改文件的权限，是“change mode”的缩写。

```
1. $ chmod 600 foo.txt
```

上面命令将 `foo.txt` 的权限改成了600。

`chmod` 还可以接受四个缩写，为不同的对象单独设置权限。

- `u` 所有者“user”的简写
- `g` 用户组“group”的缩写
- `o` 其他所有人“others”的简写
- `a` 所有人“all”的简写

```
1. # 为所有者添加可执行权限
2. $ chmod u+x foo.txt
3.
4. # 删除所有者的可执行权限
5. $ chmod u-x foo.txt
6.
7. # 为所有人添加可执行权限，等价于 a+x
8. $ chmod +x foo.txt
9.
10. # 删除其他人的读权限和写权限。
11. $ chmod o-rw foo.txt
12.
13. # 设定用户组和其他人的权限是读权限和写权限
14. $ chmod go=rw foo.txt
15.
16. # 为所有者添加执行权限，设定用户组和其他人为读权限和写权限，多种设定用逗号分隔
17. $ chmod u+x,go=rw foo.txt
```

添加权限。

- `+x` 添加执行权限
- `+r` 设置读权限
- `+w` 设置写权限
- `+rwx` 设置所有读、写和执行权限。

删除权限只需将 `+` 更改为 `-`，就可以删除任何已设置的指定权限。可以使用 `-R`（或 `--recursive`）选项来递归地操作目录和文件。

设置精确权限，可以使用 `=` 代替 `+` 或 `-` 来实现此操作。如果想为用户、组或其他用户设置不同的权限，可以使用逗号将不同表达式分开（例如 `ug=rwx,o=rx`）。

由于一共有3种可能的权限。也可以使用八进制数代替符号来设置权限。通过这种方式设置的权限最多使用3个八进制数。第1个数定义用户权限，第2个数定义组权限，第3个数定义其他权限。这3个数中的每一个都通过添加想要的权限设置来构造：读（4）、写（2）和执行（1）。

- `rwx` 7
- `rw-` 6
- `r-x` 5
- `r--` 4
- `-wx` 3

- -w- 2
- -x 1
- -- 0

umask

umask 用来查看和设置权限掩码。

```
1. $ umask
2. 0022
```

上面命令显示当前系统之中，默认的文件掩码是 **0022**，转为二进制就是 **000 000 010 010**。

可以看到，这个掩码是一个12位的二进制数，后面的9位分别代表文件三种使用对象的三类权限。只要对应位置上是 **1**，就表示关闭该项权限，所以 **010** 就表示关闭读权限。

新建文件时，通常不会带有执行权限，也就是说，新建文件的默认权限是 **rw-rw-rw-**。如果文件掩码是 **0022**，那么用户组和其他人的写权限也会被拿掉。

```
1. $ touch new.txt
2. $ ls -l new.txt
3. -rw-r--r-- 1 me me 0 2016-03-06 14:52 new.txt
```

上面代码中，**new.txt** 的用户组和其他人的写权限就没了。

umask 后面跟着参数，就表示设置权限掩码。

```
1. $ umask 0000
```

上面命令将权限掩码设为 **0000**，实际上就是关闭了权限掩码。

umask 命令设置的掩码值只能在当前Shell会话中生效，若当前Shell会话结束后，则必须重新设置。

du

du 命令用于查看指定目录的大小。

```
1. $ du -hs /path/to/directory
```

显示第一层子目录的大小。

```
1. $ du -h --max-depth=1 /path/to/folder
```

参数的含义。

- **-h** 表示人类可读的格式

- `-s` 表示总结信息，否则会显示该目录内所有文件和子目录的信息。

`tree` 命令也可以显示子目录大小。

```
1. $ tree --du -h /path/to/directory
```

md5sum

`md5sum` 命令用来显示一个文件的md5校验码。

```
1. $ md5sum image.iso
2. 34e354760f9bb7fbf85c96f6a3f94ece    image.iso
```

locate

`locate` 程序快速搜索本机的路径名数据库，并且输出每个与给定字符串相匹配的文件名。

```
1. $ locate bin/zip
2. /usr/bin/zip
3. /usr/bin/zipcloak
4. /usr/bin/zipgrep
5. /usr/bin/zipinfo
6. /usr/bin/zipnote
7. /usr/bin/zipsplit
```

`locate` 数据库由另一个叫做 `updatedb` 的程序创建。大多数装有 `locate` 的系统会每隔一天运行一回 `updatedb` 程序。因为数据库不能被持续地更新，所以当使用 `locate` 时，你会发现 目前最新的文件不会出现。为了克服这个问题，可以手动运行 `updatedb` 程序， 更改为超级用户身份，在提示符下运行 `updatedb` 命令。

`locate` 支持正则查找。 `--regex` 参数支持基本的正则表达式， `--regex` 参数支持扩展的正则表达式。

```
1. $ locate --regex 'bin/(bz|gz|zip)'
```

find

`locate` 程序只能依据文件名来查找文件，而 `find` 程序能基于各种各样的属性，搜索一个给定目录（以及它的子目录），来查找文件。

```
1. # 输出当前目录的所有子目录和文件（含子目录）
2. $ find
3. $ find .
4.
5. # 显示当前目录的文件总数
6. $ find . | wc -l
7.
```

```

8. # 当前目录的子目录总数
9. $ find . -type d | wc -l
10.
11. # 当前目录的文件总数（不含子目录）
12. $ find . -type f | wc -l
13.
14. # 当前目录的文件名匹配 "*.JPG" 且大于1M的文件总数
15. $ find . -type f -name "*.JPG" -size +1M | wc -l

```

-type 参数支持的文件类型。

- **b** 块设备文件
- **c** 字符设备文件
- **d** 目录
- **f** 普通文件
- **l** 符号链接

-size 参数支持的文件大小类型。

- **b** 512 个字节块。如果没有指定单位，则这是默认值。
- **c** 字节
- **w** 两个字节的字
- **k** 千字节
- **M** 兆字节
- **G** 千兆字节

find 程序支持的查询参数。

- **-cmin n** 匹配的文件和目录的内容或属性最后修改时间正好在 **n** 分钟之前。指定少于 **n** 分钟之前，使用 **-n**，指定多于 **n** 分钟之前，使用 **+n**。
- **-cnewer file** 匹配的文件和目录的内容或属性最后修改时间早于那些文件。
- **-ctime n** 匹配的文件和目录的内容和属性最后修改时间在 **n*24**小时之前。
- **-empty** 匹配空文件和目录。
- **-group name** 匹配的文件和目录属于一个组。组可以用组名或组 **ID** 来表示。
- **-iname pattern** 就像 **-name** 测试条件，但是不区分大小写。
- **-inum n** 匹配的文件 **inode** 号是 **n**。这对于找到某个特殊 **inode** 的所有硬链接很有帮助。
- **-mmin n** 匹配的文件或目录的内容被修改于 **n** 分钟之前。
- **-mtime n** 匹配的文件或目录的内容被修改于 **n*24**小时之前。
- **-name pattern** 用指定的通配符模式匹配的文件和目录。
- **-newer file** 匹配的文件和目录的内容早于指定的文件。当编写 **shell** 脚本，做文件备份时，非常有帮助。每次你制作一个备份，更新文件（比如说日志），然后使用 **find** 命令来决定自从上次更新，哪一个文件已经更改了。
- **-nouser** 匹配的文件和目录不属于一个有效用户。这可以用来查找属于删除帐户的文件或监测攻击行为。
- **-nogroup** 匹配的文件和目录不属于一个有效的组。
- **-perm mode** 匹配的文件和目录的权限已经设置为指定的 **mode**。**mode** 可以用八进制或符号表示法。
- **-samefile name** 相似于 **-inum** 测试条件。匹配和文件 **name** 享有同样 **inode** 号的文件。
- **-size n** 匹配的文件大小为 **n**。
- **-type c** 匹配的文件类型是 **c**。

- `-user name` 匹配的文件或目录属于某个用户。这个用户可以通过用户名或用户 ID 来表示。
- `-depth` 指导 `find` 程序先处理目录中的文件，再处理目录自身。当指定 `-delete` 行为时，会自动应用这个选项。
- `-maxdepth levels` 当执行测试条件和行为的时候，设置 `find` 程序陷入目录树的最大级别数
- `-mindepth levels` 在应用测试条件和行为之前，设置 `find` 程序陷入目录数的最小级别数。
- `-mount` 指导 `find` 程序不要搜索挂载到其它文件系统上的目录。
- `-regex` 指定正则表达式

```
1. # 找出包括空格或其它不规范字符的文件名或路径名
2. $ find . -regex '.*[^\_\.\/0-9a-zA-Z].*'
```

`find` 程序还支持逻辑操作符。

- `-and` 如果操作符两边的测试条件都是真，则匹配。可以简写为 `-a`。注意若没有使用操作符，则默认使用 `-and`。
- `-or` 若操作符两边的任一个测试条件为真，则匹配。可以简写为 `-o`。
- `-not` 若操作符后面的测试条件是真，则匹配。可以简写为一个感叹号 (!)。
- `()` 把测试条件和操作符组合起来形成更大的表达式。这用来控制逻辑计算的优先级。注意 因为圆括号字符对于 `shell` 来说有特殊含义，所以在命令行中使用它们的时候，它们必须用引号引起来，才能作为实参传递给 `find` 命令。通常反斜杠字符被用来转义圆括号字符。

```
1. # 或关系
2. ( expression 1 ) -or ( expression 2 )
3.
4. # 找出不是600权限的文件，或者不是700权限的目录
5. $ find ~ \( -type f -not -perm 0600 \) -or \( -type d -not -perm 0700 \)
```

`find` 程序的逻辑表达式，具有“短路运算”的特点，即对于 `expr1 -operator expr2` 这个表达式，`expr2` 不一定执行。这是为了提高运行速度。

- `expr1` 为真，且操作符为 `-and`，`expr2` 总是执行
- `expr1` 为假，且操作符为 `-and`，`expr2` 从不执行
- `expr1` 为真，且操作符为 `-or`，`expr2` 从不执行
- `expr1` 为假，且操作符为 `-or`，`expr2` 总是执行

为了方便执行一些常见操作，`find` 程序定义了一些预定义操作。

- `-delete` 删除当前匹配的文件。
- `-ls` 对匹配的文件执行等同的 `ls -dils` 命令。并将结果发送到标准输出。
- `-print` 把匹配文件的全路径名输送到标准输出。如果没有指定其它操作，这是默认操作。
- `-quit` 一旦找到一个匹配，退出。

```
1. # 找到匹配的文件，并显示在标准输出
2. # -print 是默认操作，可以省略
3. $ find . -print
4.
5. # 删除后缀名为BAK的文件
6. # 执行 delete 操作前，最好先执行 print 操作，确认要删除哪些文件
```

```
7. $ find . -type f -name '*.BAK' -delete
```

预定义操作可以与逻辑表达式， 结合使用。

```
1. $ find ~ -type f -and -name '*.BAK' -and -print
```

除了预定义操作以外，用户还可以使用 `-exec` 参数自定义操作。

```
1. -exec command {} ;
```

上面的命令中， `command` 是一个命令行命令， `{}` 用来指代当前路径，分号表示命令结束。

```
1. # 预定义的 -delete 操作，等同于下面的操作
2. -exec rm '{}' ';' 
```

`-exec` 使用时，每次找到一个匹配的文件，会启动一个新的指定命令的实例。

```
1. $ find ~ -type f -name 'foo*' -exec ls -l '{}' ';' 
```

执行上面的命令， `ls` 程序可能会被调用多次。

```
1. $ ls -l file1
2. $ ls -l file2
```

如果想改成 `ls` 程序只调用一次，要把 `find` 命令里面的分号，改成加号。

```
1. $ ls -l file1 file2
2. # 相当于
3. $ find ~ -type f -name 'foo*' -exec ls -l '{}' +
```

xargs

`xargs` 命令从标准输入接受输入，并把输入转换为一个特定命令的参数列表。

```
1. $ find ~ -type f -name 'foo*' -print | xargs ls -l
```

硬件操作

df

df 命令查看硬盘信息。

```
1. $ df
2. Filesystem 1K-blocks Used Available Use% Mounted on
3. /dev/sda2 15115452 5012392 9949716 34% /
4. /dev/sda5 59631908 26545424 30008432 47% /home
5. /dev/sda1 147764 17370 122765 13% /boot
```

free

free 命令查看内存占用情况。

```
1. $ free
2. total used free shared buffers cached
3. Mem: 513712 503976 9736 0 5312 122916
4. -/+ buffers/cache: 375748 137964
5. Swap: 1052248 104712 947536
```

硬盘

文件 **/etc/fstab** 配置系统启动时要挂载的设备。

1. LABEL=/12	/	ext3	defaults	1	1
2. LABEL=/home	/home	ext3	defaults	1	2
3. LABEL=/boot	/boot	ext3	defaults	1	2

输出结果一共有6个字段，含义依次如下。

- 设备名：与物理设备相关联的设备文件（或设备标签）的名字，比如说 **/dev/hda1**（第一个 IDE 通道上第一个主设备分区）。
- 挂载点：设备所连接到的文件系统树的目录。
- 文件系统类型：Linux 允许挂载许多文件系统类型。
- 选项：文件系统可以通过各种各样的选项来挂载。
- 频率：一位数字，指定是否和在什么时间用 `dump` 命令来备份一个文件系统。
- 次序：一位数字，指定 `fsck` 命令按照什么次序来检查文件系统。

mount

mount 不带参数时，显示当前挂载的文件系统。

```
1. $ mount
2. /dev/sda2 on / type ext3 (rw)
3. proc on /proc type proc (rw)
4. sysfs on /sys type sysfs (rw)
5. devpts on /dev/pts type devpts (rw,gid=5,mode=620)
6. /dev/sda5 on /home type ext3 (rw)
```

这个列表的格式是：设备 on 挂载点 type 文件系统类型（可选的）。

mount 带参数时，用于将设备文件挂载到挂载点， **-t** 参数用来指定文件系统类型。

```
1. $ mount -t iso9660 /dev/hdc /mnt/cdrom
2.
3. # 挂载一个iso文件
4. $ mount -t iso9660 -o loop image.iso /mnt/iso_image
```

umount

umount 命令用来卸载设备。

```
1. $ umount [设备名]
2.
3. $ umount /dev/hdc
```

fdisk

fdisk 命令用于格式化磁盘。

```
1. $ sudo umount /dev/sdb1
2. $ sudo fdisk /dev/sdb
```

mkfs

mkfs 命令用于在一个设备上新建文件系统。

```
1. $ sudo mkfs -t ext3 /dev/sdb1
2. $ sudo mkfs -t vfat /dev/sdb1
```

fsck

fsck 命令用于检查（修复）文件系统。

```
1. $ sudo fsck /dev/sdb1
```

dd

dd 命令用于将大型数据块，从一个磁盘复制到另一个磁盘。

```
1. $ dd if=input_file of=output_file [bs=block_size [count=blocks]]
2.
3. # 将 /dev/sdb 的所有数据复制到 /dev/sdc
4. $ dd if=/dev/sdb of=/dev/sdc
5.
6. # 将 /dev/sdb 的所有数据拷贝到一个镜像文件
7. $ dd if=/dev/sdb of=flash_drive.img
8.
9. # 从cdrom制作一个iso文件
10. $ dd if=/dev/cdrom of=ubuntu.iso
```

dmidecode

dmidecode 命令用于输出BIOS信息。

```
1. $ sudo dmidecode
```

以上命令会输出全部BIOS信息。为了便于查看，往往需要指定所需信息的类别。

- 0 BIOS
- 1 System
- 2 Base Board
- 3 Chassis 4 Processor
- 5 Memory Controller
- 6 Memory Module
- 7 Cache
- 8 Port Connector
- 9 System Slots
- 10 On Board Devices
- 11 OEM Strings
- 12 System Configuration Options
- 13 BIOS Language
- 14 Group Associations
- 15 System Event Log
- 16 Physical Memory Array
- 17 Memory Device
- 18 32-bit Memory Error
- 19 Memory Array Mapped Address
- 20 Memory Device Mapped Address

- 21 Built-in Pointing Device
- 22 Portable Battery
- 23 System Reset
- 24 Hardware Security
- 25 System Power Controls
- 26 Voltage Probe
- 27 Cooling Device
- 28 Temperature Probe
- 29 Electrical Current Probe
- 30 Out-of-band Remote Access
- 31 Boot Integrity Services
- 32 System Boot
- 33 64-bit Memory Error
- 34 Management Device
- 35 Management Device Component
- 36 Management Device Threshold Data
- 37 Memory Channel
- 38 IPMI Device
- 39 Power Supply

查看内存信息的命令如下。

```
1. $ sudo dmidecode -t 17
2. # 或者
3. $ dmidecode --type 17
```

以下是其他一些选项。

```
1. # 查看BIOS信息
2. $ sudo dmidecode -t 0
3.
4. # 查看CPU信息
5. $ sudo dmidecode -t 4
```

`dmidecode` 也支持关键词查看，关键词与类别的对应关系如下。

- bios 0, 13
- system 1, 12, 15, 23, 32
- baseboard 2, 10
- chassis 3
- processor 4
- memory 5, 6, 16, 17
- cache 7
- connector 8
- slot 9

查看系统信息的命令如下。

```
1. $ sudo dmidecode -t system
```

lspci

lspci 命令列出本机的所有PCI设备。

```
1. $ lspci
```

该命令输出信息的格式如下。

```
1. 03:00.0 Unassigned class [ff00]: Realtek Semiconductor Co., Ltd. RTS5209 PCI Express Card Reader (rev 01)
```

输出信息一共分成三个字段。

- Field 1: PCI bus slot 的编号
- Field 2: PCI slot的名字
- Field 3: 设备名和厂商名

如果想查看更详细信息，可以使用下面的命令。

```
1. $ lspci -vmm
```

lsusb

lsusb 命令用于操作USB端口。

下面命令列出本机所有USB端口。

```
1. $ lsusb
```

它的输出格式如下。

```
1. Bus 002 Device 003: ID 0781:5567 SanDisk Corp. Cruzer Blade
```

各个字段的含义如下。

- Bus 002 : bus编号
- Device 003: bus 002连接的第三个设备
- ID 0781:5567: 当前设备的编号，冒号前是厂商编号，冒号后是设备编号
- SanDisk Corp. Cruzer Blade: 厂商和设备名

找出本机有多少个USB接口可用。

```
1. $ find /dev/bus/  
2. /dev/bus/
```

```
3. /dev/bus/usb
4. /dev/bus/usb/002
5. /dev/bus/usb/002/006
6. /dev/bus/usb/002/005
7. /dev/bus/usb/002/004
8. /dev/bus/usb/002/002
9. /dev/bus/usb/002/001
10. /dev/bus/usb/001
11. /dev/bus/usb/001/007
12. /dev/bus/usb/001/003
13. /dev/bus/usb/001/002
14. /dev/bus/usb/001/001
```

查看某个USB设备的详细情况。

```
1. $ lsusb -D /dev/bus/usb/002/005
```

查看所有设备的详细情况。

```
1. $ lsusb -v
```

查看USB端口的版本。

```
1. $ lsusb -v | grep -i bcdusb
```

主机管理

hostname命令

`hostname` 命令返回当前服务器的主机名。

```
1. $ hostname
```

命名管道

在大多数类似 Unix 的操作系统中，有可能创建一种特殊类型的文件，叫做命名管道。命名管道用来在 两个进程之间建立连接，也可以像其它类型的文件一样使用。

命令管道的行为类似于文件，但实际上形成了先入先出（FIFO）的缓冲。和普通（未命令的）管道一样，数据从一端进入，然后从另一端出现。通过命令管道，有可能像这样设置一些东西：

```
1. process1 > named_pipe
```

和

```
1. process2 < named_pipe
```

表现出来就像这样：

```
1. process1 | process2
```

设置一个命名管道

使用 `mkfifo` 命令能够创建命令管道：

```
1. $ mkfifo pipe1
2. $ ls -l pipe1
3. prw-r--r-- 1 me me 0 2009-07-17 06:41 pipe1
```

这里我们使用 `mkfifo` 创建了一个名为 `pipe1` 的命名管道。使用 `ls` 命令，我们查看这个文件，看到位于属性字段的第一个字母是 “p”，表明它是一个命名管道。

使用命名管道

为了演示命名管道是如何工作的，我们将需要两个终端窗口（或用两个虚拟控制台代替）。在第一个终端中，我们输入一个简单命令，并把命令的输出重定向到命名管道：

```
1. $ ls -l > pipe1
```

我们按下 `Enter` 按键之后，命令将会挂起。这是因为在管道的另一端没有任何接受数据。当这种现象发生的时候，据说是管道阻塞了。一旦我们绑定一个进程到管道的另一端，该进程开始从管道中读取输入的时候，这种情况会消失。使用第二个终端窗口，我们输入这个命令。

```
1. $ cat < pipe1
```

然后产自第一个终端窗口的目录列表出现在第二个终端中，并作为来自 `cat` 命令的输出。在第一个终端窗口中的 `ls` 命令一旦它不再阻塞，会成功地结束。

进程管理

ps

ps 命令用来列出进程信息。

```
1. $ ps
2.  PID TTY          TIME CMD
3.  5198 pts/1    00:00:00 bash
4. 10129 pts/1    00:00:00 ps
```

不带任何参数时，**ps** 只列出与当前Session相关的进程。输出结果中，**PID** 是进程ID、**TTY** 是进程的终端号（如果显示 **?**，则表示进程没有终端），**TIME** 是消耗的CPU时间，**CMD** 是触发进程的命令。

x 参数列出所有进程的详细信息，包括不在当前Session的信息。

```
1. $ ps x
2.  PID TTY   STAT   TIME COMMAND
3.  2799 ?    Ssl    0:00 /usr/libexec/bonobo-activation-server -ac
4.  2820 ?    Sl     0:01 /usr/libexec/evolution-data-server-1.10 --
```

这时的输出结果，会多出 **STAT** 一栏，表示状态。它的各种值如下。

- **R** 正在运行或准备运行
- **S** 正在睡眠，即没有运行，正在等待一个事件唤醒
- **D** 不可中断睡眠。进程正在等待 I/O，比如磁盘驱动器的I/O
- **T** 已停止，即进程停止运行
- **Z** “僵尸”进程。即这是一个已经终止的子进程，但父进程还没有清空它（没有把子进程从进程表中删除）
- **<** 高优先级进程。这可能会授予一个进程更多重要的资源，给它更多的 CPU 时间。
- **N** 低优先级进程。一个低优先级进程（一个“好”进程）只有当其它高优先级进程执行之后，才会得到处理器时间。

aux 参数可以显示更多信息。

```
1. $ ps aux
2.  USER  PID  %CPU  %MEM    VSZ   RSS TTY   STAT   START   TIME  COMMAND
3.  root    1   0.0   0.0   2136   644 ?     Ss     Mar05   0:31   init
4.  root    2   0.0   0.0     0     0 ?     S<lt;   Mar05   0:00  [kt]
```

输出结果包含的列的含义如下。

- **USER** 用户ID，表示进程的所有者
- **%CPU** 百分比表示的 CPU 使用率
- **%MEM** 百分比表示的内存使用率
- **VSZ** 虚拟内存大小

- `RSS` 进程占用的物理内存的大小，以千字节为单位。
- `START` 进程运行的起始时间。若超过24小时，则用天表示。

top

`top` 命令可以查看机器的当前状态。

```
1. $ top
```

它的输出结果分为两部分，最上面是系统概要，下面是进程列表，以 CPU 的使用率排序。

输出结果是动态更新的，默认每三分钟更新一次。

jobs

`jobs` 命令用来查看后台任务。

```
1. $ jobs
2. [1]+  Running                  xlogo &
```

输出结果之中，每个后台任务会有一个编号。上面结果中，`xlogo` 的编号是 `1`，`+` 表示正在运行。

fg

`fg` 命令用于将后台任务切换到前台。

```
1. $ fg %1
```

`fg` 命令之后，跟随着一个百分号和工作序号，用来指定切换哪一个后台任务。如果只有一个后台任务，那么 `fg` 命令可以不带参数。

bg

`bg` 命令用于将一个暂停的前台任务，转移到后台。只有暂停的任务，才能使用 `bg` 命令，因为正在运行的任务，命令行是无法输入的。

```
1. $ bg %1
```

`Ctrl + z` 可以暂停正在运行的前台任务。

kill

`kill` 命令用于杀死进程。它的参数是进程ID。


```
1. $ kill 28401
```

kill 命令的实质是操作系统向进程发送信号。在使用 `Ctrl-c` 的情况下，会发送一个叫做 `INT`（中断）的信号；当使用 `Ctrl-z` 时，则发送一个叫做 `TSTP`（终端停止）的信号。

kill 命令可以用来向进程发送指定信号。

```
1. $ kill [-signal] PID
```

下面是常见信号。

- **HUP**：编号1，表示挂起。发送这个信号到前台程序，程序会终止。许多守护进程也使用这个信号，来重新初始化。这意味着，当发送这个信号到一个守护进程后，这个进程会重新启动，并且重新读取它的配置文件。Apache 网络服务器守护进程就是一个例子。
- **INT**：编号2，中断。实现和 `Ctrl-c` 一样的功能，由终端发送。通常，它会终止一个程序。
- **KILL**：编号9，杀死。进程可能选择忽略这个信号。所以，操作系统不发送该信号到目标进程，而是内核立即终止这个进程。当一个进程以这种方式终止的时候，它没有机会去做些“清理”工作，或者是保存劳动成果。因为这个原因，把 **KILL** 信号看作杀手锏，当其它终止信号失败后，再使用它。
- **TERM**：编号15，终止。这是 `kill` 命令发送的默认信号。如果程序仍然“活着”，可以接受信号，那么这个信号终止。
- **CONT**：编号18，继续。在停止一段时间后，进程恢复运行。
- **STOP**：编号19，停止。这个信号导致进程停止运行，而没有终止。像 **KILL** 信号，它不被发送到目标进程，因此它不能被忽略。
- **QUIT**：编号3，退出
- **SEGV**：编号11，段错误。如果一个程序非法使用内存，就会发送这个信号。也就是说，程序试图写入内存，而这个内存空间是不允许此程序写入的。
- **TSTP**：编号20，终端停止。当按下 `Ctrl-z` 组合键后，终端发送这个信号。不像 **STOP** 信号，**TSTP** 信号由目标进程接收，且可能被忽略。
- **WINCH**：编号28，改变窗口大小。当改变窗口大小时，系统会发送这个信号。一些程序，像 `top` 和 `less` 程序会响应这个信号，按照新窗口的尺寸，刷新显示的内容。

-l 参数可以列出所有信号。

```
1. $ kill -l
```

killall

killall 命令用于向指定的程序或用户发送信号。

```
1. $ killall [-u user] [-signal] name
```

其他进程相关命令

- **pstree** 输出树型结构的进程列表，这个列表展示了进程间父/子关系。

- `vmstat` 输出一个系统资源使用快照，包括内存，交换分区和磁盘 I/O。为了看到连续的显示结果，则在命令后加上延时的时间（以秒为单位）。例如，“`vmstat 5`”。终止输出，按下 `Ctrl-c` 组合键。
- `xload` 一个图形界面程序，可以画出系统负载的图形。
- `tload` 与 `xload` 程序相似，但是在终端中画出图形。使用 `Ctrl-c`，来终止输出。

重定向

重定向指的是将命令行输出写入指定位置。

- `cmd1 | cmd2` : Pipe; take standard output of cmd1 as standard input to cmd2.
- `> file` : Direct standard output to file.
- `< file` : Take standard input from file.
- `>> file` : Direct standard output to file; append to file if it already exists.
- `>| file` : Force standard output to file even if noclobber is set.
- `n>| file` : Force output to file from file descriptor n even if noclobber is set.
- `<> file` : Use file as both standard input and standard output.
- `n<> file` : Use file as both input and output for file descriptor n.
- `<< label` : Here-document; see text.
- `n > file` : Direct file descriptor n to file.
- `n < file` : Take file descriptor n from file.
- `n >> file` : Direct file descriptor n to file; append to file if it already exists.
- `n>&` : Duplicate standard output to file descriptor n.
- `n<&` : Duplicate standard input from file descriptor n.
- `n>&m` : File descriptor n is made to be a copy of the output file descriptor.
- `n<&m` : File descriptor n is made to be a copy of the input file descriptor.
- `&>file` : Directs standard output and standard error to file.
- `<&-` : Close the standard input.
- `>&-` : Close the standard output.
- `n>&-` : Close the output from file descriptor n.
- `n<&-` : Close the input from file descriptor n.
- `n>&word` : If n is not specified, the standard output (file descriptor 1) is used. If the digits in word do not specify a file descriptor open for output, a redirection error occurs. As a special case, if n is omitted, and word does not expand to one or more digits, the standard output and standard error are redirected as described previously.
- `n<&word` : If word expands to one or more digits, the file descriptor denoted by n is made to be a copy of that file descriptor. If the digits in word do not specify a file descriptor open for input, a redirection error occurs. If word evaluates to -, file descriptor n is closed. If n is not specified, the standard input (file descriptor 0) is used.
- `n>&digit-` : Moves the file descriptor digit to file descriptor n, or the standard output (file descriptor 1) if n is not specified.
- `n<&digit-` : Moves the file descriptor digit to file descriptor n, or the standard input (file descriptor 0) if n is not specified. digit is closed after being duplicated to n.

> 用来将标准输出重定向到指定文件。

```
1. $ ls -l /usr/bin > ls-output.txt
```

如果重定向后的指定文件已经存在，就会被覆盖，不会有任何提示。

如果命令没有任何输出，那么重定向之后，得到的是一个长度为 `0` 的文件。因此，`>` 具有创建新文件或改写现存文件、将其改为长度 `0` 的作用。

```
1. $ > ls-output.txt
```

`>>` 用来将标准输出重定向追加到指定文件。

```
1. $ ls -l /usr/bin >> ls-output.txt
```

`2>` 用来将标准错误重定向到指定文件。

```
1. $ ls -l /bin/usr 2> ls-error.txt
```

标准输出和标准错误，可以重定向到同一个文件。

```
1. $ ls -l /bin/usr > ls-output.txt 2>&1
2. # 或者
3. $ ls -l /bin/usr &> ls-output.txt
4.
5. # 追加到同一个文件
6. $ ls -l /bin/usr &>> ls-output.txt
```

如果不希望输出错误信息，可以将它重定向到一个特殊文件 `/dev/null`。

```
1. $ ls -l /bin/usr 2> /dev/null
```

`|` 用于将一个命令的标准输出，重定向到另一个命令的标准输入。

```
1. $ ls -l /usr/bin | less
```

不要将 `>` 与 `|` 混淆。

```
1. $ ls > less
```

上面命令会在当前目录，生成一个名为 `less` 的文本文件。

下面是标准错误重定向的一个例子。

```
1. invalid_input () {
2.     echo "Invalid input '$REPLY'" >&2
3.     exit 1
4. }
5. read -p "Enter a single item > "
6. [[ -z $REPLY ]] && invalid_input
```

tee

tee 命令用于同时将标准输出重定向到文件，以及另一个命令的标准输入。

```
1. $ ls /usr/bin | tee ls.txt | grep zip
```

命令替换

命令替换 (command substitution) 指的是将一个命令的输出，替换进入另一个命令。`$(command)` 表示命令替换，另一种写法是使用反引号。

```
1. $ echo $(ls)
2. # 或者
3. $ echo `ls`
4.
5. $ ls -l $(which cp)
6. # 或者
7. $ ls -l `which cp`
```

basename

basename 命令清除 一个路径名的开头部分，只留下一个文件的基本名称。

```
1. #!/bin/bash
2. # file_info: simple file information program
3. PROGRAMME=$(basename $0)
4. if [[ -e $1 ]]; then
5.     echo -e "\nFile Type:"
6.     file $1
7.     echo -e "\nFile Status:"
8.     stat $1
9. else
10.    echo "$PROGRAMME: usage: $PROGRAMME file" >&2
11.    exit 1
12. fi
```

正则表达式

正则表达式 是表达文本模式的方法。

- `.` : 匹配任何单个字符。
- `?` : 上一项是可选的, 最多匹配一次。
- `*` : 前一项将被匹配零次或多次。
- `+` : 前一项将被匹配一次或多次。
- `{N}` : 上一项完全匹配N次。
- `{N, }` : 前一项匹配N次或多次。
- `{N, M}` : 前一项至少匹配N次, 但不超过M次。
- `--` : 表示范围, 如果它不是列表中的第一个或最后一个, 也不是列表中某个范围的终点。
- `^` : 匹配行首的空字符串; 也代表不在列表范围内的字符。
- `$` : 匹配行尾的空字符串。
- `\b` : 匹配单词边缘的空字符串。
- `\B` : 匹配空字符串, 前提是它不在单词的边缘。
- `\<` : 匹配单词开头的空字符串。
- `\>` : 匹配单词末尾的空字符串。

元字符

元字符 是表示特殊函数的字符, 包括以下这些 `^ $. [] { } - ? * + () | \`。除了元字符, 其他字符在正则表达式中, 都表示原来的含义。

- `.` 匹配任意字符, 但不含空字符
- `^` 匹配文本行开头
- `$` 匹配文本行结尾

```
1. $ grep -h '.zip' dirlist*.txt
```

上面命令在文件中查找包含正则表达式“.zip”的文本行。注意, 上面命令不会匹配 `zip` 程序, 因为 `zip` 只有三个字符, 而 `.zip` 要求四个字符。

```
1. $ grep -h '^zip' dirlist*.txt
2. $ grep -h 'zip$' dirlist*.txt
```

上面命令分别在文件列表中搜索行首, 行尾以及行首和行尾同时包含字符串“zip”(例如, zip 独占一行)的匹配行。 注意正则表达式`^$`(行首和行尾之间没有字符)会匹配空行。

方括号

方括号之中的字符, 表示可以任意匹配其中的一个。

```
1. $ grep -h '[bg]zip' dirlist*.txt
```

上面命令匹配包含字符串“bzip”或者“gzip”的任意行。

注意，元字符放入方括号之中，会失去其特殊含义。但有两种情况除外，`^` 在方括号的开头，表示否定，否则只是一个普通字符，表示原义。

```
1. $ grep -h '^[bg]zip' dirlist*.txt
```

上面命令匹配不以 `b` 或 `g` 开头的 `zip` 字符串。注意，上面命令不会匹配 `zip`，因为一个否定的字符集仍然要求存在一个字符。

- 在方括号之中表示一个字符区域。

```
1. $ grep -h '^[A-Z]' dirlist*.txt
```

上面命令匹配所有以大写字母开头的文本行。类似的，`^[A-Za-z0-9]` 表示以大写字母、小写字母、数字开头的文本行。

注意，连字号如果不构成一个字符区域，则表示其本来的含义。

```
1. $ grep -h '[-AZ]' dirlist*.txt
```

上面命令匹配包含一个连字符，或一个大写字母“A”，或一个大写字母“Z”的文件名。

预定义字符类

由于 `locale` 设置不同，Shell展开正则表达式 `[A-Z]` 时，可能不是解释为所有大写字母，而是解释为包括所有字母的字典顺序。

```
1. $ ls /usr/sbin/[A-Z]*
```

上面命令在某些发行版里面，会返回所有大写字母或小写字母开头的文件。

为了避免这个问题，可以使用正则表达式的预定义字符类。

- `[:alnum:]` 字母数字字符。在 ASCII 中，等价于：`[A-Za-z0-9]`
- `[:word:]` 与 `[:alnum:]` 相同，但增加了下划线字符。
- `[:alpha:]` 字母字符。在 ASCII 中，等价于 `[A-Za-z]`
- `[:blank:]` 包含空格和 tab 字符。
- `[:cntrl:]` ASCII 的控制码。包含了0到31，和127的 ASCII 字符。
- `[:digit:]` 数字0到9
- `[:graph:]` 可视字符。在 ASCII 中，它包含33到126的字符。
- `[:lower:]` 小写字母。
- `[:punct:]` 标点符号字符。
- `[:print:]` 可打印的字符。等于 `[:graph:]` 中的所有字符，再加上空格字符。
- `[:space:]` 空白字符，包括空格，tab，回车，换行，vertical tab，和 form feed.在 ASCII 中，

等价于 `[\t\r\n\v\f]`

- `[:upper:]` 大写字母。
- `[:xdigit:]` 用来表示十六进制数字的字符。在 ASCII 中，等价于 `[0-9A-Fa-f]`

```
1. $ ls /usr/sbin/[[:upper:]]*
```

上面命令返回所有大写字母开头的文件名。

选择

`|` 表示匹配一系列字符串之中的一个。注意与方括号区分，方括号表示匹配一系列字符之中的一个。

```
1. $ echo "AAA" | grep -E 'AAA|BBB'
2. AAA
3. $ echo "BBB" | grep -E 'AAA|BBB'
4. BBB
5. $ echo "CCC" | grep -E 'AAA|BBB'
6. $
```

上面代码中，`AAA|BBB` 表示匹配字符串 `AAA` 或者是字符串 `BBB`。 `grep` 程序使用 `-E` 参数，表示按照正则表达式规则匹配。并且，这个正则表达式放在单引号之中，为的是阻止Shell把 `|` 解释为管道操作符。

`|` 可以多个连用，也可以与其他正则规则结合使用。

```
1. $ echo "AAA" | grep -E 'AAA|BBB|CCC'
2.
3. $ grep -Eh '^(bz|gz|zip)' dirlist*.txt
```

量词操作符

量词操作符表示一个元素被匹配的次数。

- `?` 匹配前面的元素出现0次或1次
- `*` 匹配前面的元素出现0次或多次
- `+` 匹配前面的元素出现1次或多次
- `{n}` 匹配前面的元素出现了 `n` 次
- `{n,m}` 匹配前面的元素它至少出现了 `n` 次，但是不多于 `m` 次
- `{n,}` 匹配前面的元素至少出现了 `n` 次
- `{,m}` 匹配前面的元素，如果它出现的次数不多于 `m` 次。

系统信息

uname

`uname` 命令返回当前机器的信息。

```
1. # 内核的版本
2. $ uname -r
3. 3.2.0-24-virtual
4.
5. # CPU 架构
6. $ uname -m
7. x86_64
```

如果要了解操作系统的版本，可以查看 `/etc/issue` 文件。

```
1. $ cat /etc/issue
2. Debian GNU/Linux 9 \n \l
```

service

`service` 命令可以查看当前正在运行的服务。

```
1. $ service --status-all
2. [ + ] apache2
3. [ ? ] atd
4. [ - ] bootlogd
```

上面代码中，`+` 表示正在运行，`-` 表示已经停止，`?` 表示 `service` 命令不了解相关信息。

文本处理

cat

`cat` 可以文件的内容，显示在标准输出。

```
1. $ cat text1
2. 1 apple
3. 2 pear
4. 3 banana
```

它也可以同时输出多个文件内容。

```
1. $ cat text1 text2
```

它与重定向结合，就可以合并多个文件。

```
1. # 合并文本文件
2. $ cat text* > text.all
3.
4. # 合并二进制文件
5. $ cat movie.mpeg.0* > movie.mpeg
```

如果调用 `cat` 命令时没有任何参数，它将读取标准输入，然后显示到标准输出。按下 `Ctrl + d`，将会结束 `cat` 读取标准输入。利用这一点，可以将键盘输入写入指定文件，按下 `Ctrl + d` 结束输入。

```
1. $ cat > lazy_dog.txt
```

它的参数如下。

- `-n` 输出结果显示行号
- `-s` 将多个连续的空白行，输出为一行
- `-A` 输出结果中显示控制符，比如Tab键显示为 `^I`，行尾显示 `$`

`cat` 支持Here document，显示多行文本。

```
1. cat << _EOF_
2. <HTML>
3.     <HEAD>
4.         <TITLE>$TITLE</TITLE>
5.     </HEAD>
6.     <BODY>
7.         <H1>$TITLE</H1>
8.         <P>$TIME_STAMP</P>
9.     </BODY>
```

```
10. </HTML>
11. _EOF_
```

Here document 常在脚本当中作为输入的手段。

```
1. $ sort -k2 <<END
2. > 1 apple
3. > 2 pear
4. > 3 banana
5. > END
6. 1 apple
7. 3 banana
8. 2 pear
```

如果使用 `<<-` 代替 `<<`，行首的tab键将被剥离。

nl

`nl` 命令为文本文件添加行号，显示在标准输出。

```
1. $ nl example.txt
```

sort

`sort` 命令将文本文件的所有行排序后输出。

```
1. $ sort file1.txt file2.txt file3.txt > final_sorted_list.txt
```

它的参数如下。

- `-b` `--ignore-leading-blanks` 默认情况下，排序用的是每行的第一个字符。这个参数忽略每行开头的空格，从第一个非空白字符开始排序。
- `-f` `--ignore-case` 让排序不区分大小写。
- `-n` `--numeric-sort` 按照数值排序，而不是字符值，用于行首是数值的情况。
- `-r` `--reverse` 按相反顺序排序。结果按照降序排列，而不是升序。
- `-k` `--key=field1[,field2]` 指定按照每行的第几个字段（从1开始）排序，而不是按照行首字符排序。该属性可以多个连用，用于指定多重排序标准，还可以指定每个字段指定排序标准，这些值与全局属性一致，比如 `b`（忽略开头的空格），`n`（数值排序），`r`（逆向排序）等等。
- `-m` `--merge` 把每个参数看作是一个预先排好序的文件。把多个文件合并成一个排好序的文件，而没有执行额外的排序。
- `-o` `--output=file` 把排好序的输出结果发送到文件，而不是标准输出。
- `-t` `--field-separator=char` 定义字段分隔字符。默认情况下，字段由空格或制表符分隔。
- `-u` 输出结果中删除重复行

```
1. $ sort --key=1,1 --key=2n distros.txt
```

上面命令中，第一个 `--key` 指定第一排序标准是只用第一字段（`1,1`），也可以指定使用第一字段第一个字符（`1.1`）；第二排序标准是第二字段，按数值排序。

uniq

`uniq` 命令在排序后的行中，删除所有重复的行，保证所有输出没有重复。

```
1. $ ls /bin /usr/bin | sort | uniq
```

它的参数如下。

- `-c` 输出所有的重复行，并且每行开头显示重复的次数。
- `-d` 只输出重复行，而不是不重复的文本行。
- `-f n` 忽略每行开头的 `n` 个字段，字段之间由空格分隔，正如 `sort` 程序中的空格分隔符；然而，不同于 `sort` 程序，`uniq` 没有选项来设置备用的字段分隔符。
- `-i` 在比较文本行的时候忽略大小写。
- `-s n` 跳过（忽略）每行开头的 `n` 个字符。
- `-u` 只是输出独有的文本行。这是默认的。
- `-V` 按照版本号排序。

`-V` 参数可以按版本号排列（从小到大）。

```
1. $ sort -V input.txt
2. 1.0.15
3. 1.3.0
4. 2.1.2
5. 3.0.0
```

`-rV` 参数可以按版本号逆序排列。

```
1. $ sort -rV input.txt
2. 3.0.0
3. 2.1.2
4. 1.3.0
5. 1.0.15
```

cut

`cut` 程序用来从文本行中抽取文本，并把其输出到标准输出。它能够接受多个文件参数或者标准输入。

它的参数如下。

- `-c char_list` 抽取指定范围的文本
- `-f field_list` 抽取指定字段，字段之间可以tab分隔也可以逗号分隔
- `-d delim_char` 指定字段分隔符，默认是tab键
- `--complement` 抽取整个文本行，除了那些由 `-c` 和 `/` 或 `-f` 选项指定的文本。

```
1. # 抽取每行的第三个字段
2. $ cut -f 3 distros.txt
3.
4. # 抽取每行的第7到第10个字符
5. $ cut -c 7-10 distros.txt
6.
7. # 抽取每行的第23个到结尾的字符1
8. $ cut -c 23- distros.txt
9.
10. # 指定字段分隔符为冒号
11. $ cut -d ':' -f 1 /etc/passwd
```

paste

paste 程序将多个文本文件按行合并，即每一行都由原来文本文件的每一行组成，显示在标准输出。

```
1. $ paste distros-dates.txt distros-versions.txt
```

WC

wc 命令输出一个文本文件的统计信息（word count），一共有三个值，分别为行数、词数和字节数。

```
1. $ wc ls-output.txt
2. 7902 64566 503634 ls-output.txt
```

如果使用 **-l** 参数，则只输出行数。

```
1. $ ls /bin /usr/bin | sort | uniq | wc -l
2. 2728
```

head

head 命令返回文本文件的头部，默认显示10行。

-n 参数指定显示的行数。

```
1. $ head -n 5 ls-output.txt
```

tail

tail 命令返回文本文件的尾部，默认显示10行。

-n 参数指定显示的行数。

```
1. $ tail -n 5 ls-output.txt
```

`-f` 会实时追加显示新增的内容，常用于实时监控日志，按 `Ctrl + c` 停止。

```
1. $ tail -f /var/log/messages
```

grep

`grep` 程序用于在指定文件之中，搜索符合某个模式的行，并把搜索结果输出到标准输出。

```
1. $ grep keyword foo.txt
```

上面命令输出 `foo.txt` 之中匹配 `keyword` 的行。

`grep` 程序可以同时搜索多个文件。

```
1. $ grep keyword f*.txt
```

上面命令输出多个文件中匹配 `keyword` 的行。

`-l` 参数输出匹配的文件名，而不是文件行。

```
1. $ grep -l bzip dirlist*.txt
```

如果想搜索文件名，而不是文件内容，可以使用重定向。

```
1. $ ls /usr/bin | grep zip
```

上面命令会输出 `/usr/bin` 目录中，文件名中包含子字符串 `zip` 的所有文件。

参数的含义。

- `-c` 或 `--count` 输出匹配的数量，而不是匹配的文本行。如果使用了 `-v`，则输出不匹配的数量。
- `-h` 或 `--no-filename` 应用于多文件搜索，不在每行匹配的文本前，输出文件名
- `-i` 或 `--ignore-case` 忽略大小写
- `-l` 或 `--files-with-matches` 输出包含匹配项的文件名，而不是文本行本身
- `-L` 或 `--files-without-match` 类似于 `-l`，但输出不包含匹配项的文件名
- `-n` 或 `--line-number` 每个匹配行之前输出其对应的行号
- `-v` 或 `--invert-match` 只返回不符合模式的行

sed

`sed` 是一个强大的文本编辑工具。

```
1. # 输出前5行
```

```
2. $ sed -n '1,5p' distros.txt
3.
4. # 输出包含指定内容的行
5. $ sed -n '/SUSE/p' distros.txt
6.
7. # 输出不包含指定内容的行
8. $ sed -n '/SUSE/!p' distros.txt
9.
10. # 替换内容（只替换第一个）
11. $ sed 's/regexp/replacement/' distros.txt
12.
13. # 替换内容（全局替换）
14. $ sed 's/regexp/replacement/g' distros.txt
```

时间管理

date 命令

`date` 命令用于输出当前时间

```
1. $ date
2. 2016年 03月 14日 星期一 17:32:35 CST
```

`date` 命令后面用加号 (`+`) 指定显示的格式。

```
1. $ date +%d_%b_%Y
2. 10_Sep_2018
3.
4. $ date +%D
5. 09/10/18
6.
7. $ date +%F-%T
8. 2018-09-10-11:09:51
```

完整的格式参数如下。

- %a 星期名的缩写 (Sun)
- %A 星期名的全称 (Sunday)
- %b 月份的缩写 (Jan)
- %B 月份的全称 (January)
- %c 日期和时间 (Thu Mar 3 23:05:25 2005)
- %C 世纪，就是年份数省略后两位 (20)
- %d 一个月的第几天 (01)
- %D 日期，等同于 `%m/%d/%y`
- %e 一个月的第几天，用空格补零，等同于 `_%d`
- %F 完整的日期，等同于 `%Y-%m-%d`
- %g last two digits of year of ISO week number (see %G)
- %G year of ISO week number (see %V); normally useful only with %V
- %h 等同于 `%b`
- %H 小时 (00..23)
- %I 小时 (01..12)
- %j day of year (001..366)
- %k hour (0..23)
- %l hour (1..12)
- %m month (01..12)
- %M minute (00..59)
- %N nanoseconds (000000000..999999999)
- %p locale's equivalent of either AM or PM; blank if not known

- %P like %p, but lower case
- %r locale's 12-hour clock time (e.g., 11:11:04 PM)
- %R 24-hour hour and minute; same as %H:%M
- %s seconds since 1970-01-01 00:00:00 UTC
- %S second (00..60)
- %T time; same as %H:%M:%S
- %u day of week (1..7); 1 is Monday
- %U week number of year, with Sunday as first day of week (00..53)
- %V ISO week number, with Monday as first day of week (01..53)
- %w day of week (0..6); 0 is Sunday
- %W week number of year, with Monday as first day of week (00..53)
- %x locale's date representation (e.g., 12/31/99)
- %X locale's time representation (e.g., 23:13:48)
- %y last two digits of year (00..99)
- %Y year
- %Z +hhmm numeric timezone (e.g., -0400)
- %:z +hh:mm numeric timezone (e.g., -04:00)
- %::z +hh:mm:ss numeric time zone (e.g., -04:00:00)
- %Z alphabetic time zone abbreviation (e.g., EDT)

cal 命令

cal 命令用于显示日历。不带有参数时，显示的是当前月份。

```

1. $ cal
2.      三月 2016
3. 日 一 二 三 四 五 六
4.      1  2  3  4  5
5.  6  7  8  9 10 11 12
6. 13 14 15 16 17 18 19
7. 20 21 22 23 24 25 26
8. 27 28 29 30 31

```

用户管理

id

id 命令用于查看指定用户的用户名和组名。

```
1. $ id
2. uid=500(me) gid=500(me) groups=500(me)
```

id 输出结果分为三个部分，分别是UID（用户编号和用户名）、GID（组编号和组名），groups（用户所在的所有组）。

用户帐户的信息，存放在 `/etc/passwd` 文件里面；用户组的信息，存放在 `/etc/group` 文件里面。

```
1. # 返回UID
2. $ id -u [UserName]
3.
4. # 返回GID
5. $ id -g [UserName]
6.
7. # 返回用户名
8. $ id -un [UserName]
9.
10. # 返回组名
11. $ id -gn [UserName]
```

上面的命令，如果省略用户名，则返回当前用户的信息。

SU

su 命令允许你以另一个用户的身份，启动一个新的 shell 会话，或者是以这个用户的身份来发布一个命令。

```
1. $ su otherUser
```

执行上面的命令以后，系统会提示输入密码。通过以后，就以另一个用户身份在执行命令了。

如果不加用户名，则表示切换到root用户。

```
1. $ su
```

-l 参数表示启动一个需要登录的新的Shell，这意味着工作目录会切换到该用户的主目录。它的缩写形式是 **-**。

```
1. $ su -
```

上面命令表示，切换到root用户的身份，且工作目录也切换到root用户的主目录。

-c 参数表示只以其他用户的身份，执行单个命令，而不是启动一个新的Session。

```
1. $ su -c 'command'
2.
3. # 实例
4. $ su -c 'ls -l /root/*'
```

sudo

sudo 命令很类似 **su** 命令，但有几点差别。

- 对于管理员来说，**sudo** 命令的可配置性更高
- **sudo** 命令通常只用于执行单个命令，而不是开启另一个Session。
- **sudo** 命令不要求超级用户的密码，而是用户使自己的密码来认证。

sudo 的设置文件在 **/etc/sudoers** 之中。

-l 参数列出用户拥有的所有权限。

```
1. $ sudo -l
```

chown

chown 命令用来更改文件或目录的所有者和用户组。使用这个命令需要超级用户权限。

```
1. $ chown [owner][:[group]] file
```

下面是一些例子。

```
1. # 更改文件所有者
2. $ sudo chown bob foo.txt
3.
4. # 更改文件所有者和用户组
5. $ sudo chown bob:users foo.txt
6.
7. # 更改用户组
8. $ sudo chown :admins foo.txt
9.
10. # 更改文件所有者和用户组（用户 bob 登录系统时，所属的用户组）
11. $ sudo chown bob: foo.txt
```

chgrp

chgrp 命令更改用户组，用法与 **chown** 命令类似。

useradd

useradd 命令用来新增用户。

```
1. $ useradd -G admin -d /home/bill -s /bin/bash -m bill
```

上面命令新增用户 **bill**，参数 **-G** 指定用户所在的组，参数 **-d** 指定用户的主目录，参数 **-s** 指定用户的 Shell，参数 **-m** 表示如果该目录不存在，则创建该目录。

usermod

usermod 命令用来修改用户的各项属性。

```
1. $ usermod -g sales jerry
```

上面的命令修改用户 **jerry** 属于的主要用户组为 **sales**。

```
1. $ usermod -G sales jerry
```

上面的命令修改用户 **jerry** 属于的次要用户组为 **sales**。

adduser

adduser 命令用来将一个用户加入用户组。

```
1. $ sudo adduser username group1
```

groupadd

groupadd 命令用来新建一个用户组。

```
1. $ sudo groupadd group1
2. $ sudo adduser foobar group1
```

groupdel

groupdel 命令用来删除一个用户组。

```
1. $ sudo groupdel group1
```

passwd

`passwd` 命令用于修改密码。

1. # 修改自己的密码
2. \$ `passwd`
- 3.
4. # 修改其他用户的密码
5. \$ `sudo passwd [user]`

Shell 的命令

命令的类别

Bash可以使用的命令分成四类。

- 可执行程序
- Shell 提供的命令
- Shell 函数
- 前三类命令的别名

type, whatis

type 命令可以显示命令类型。

```
1. $ type command
```

下面是几个例子。

```
1. $ type type
2. type is a shell builtin
3.
4. $ type ls
5. ls is aliased to `ls --color=tty'
6.
7. $ type cp
8. cp is /bin/cp
```

whatis 命令显示指定命令的描述。

```
1. $ whatis ls
2. ls (1) - list directory contents
```

apropos

apropos 命令返回符合搜索条件的命令列表。

```
1. $ apropos floppy
2. create_floppy_devices (8) - udev callout to create all possible
3. fdformat (8) - Low-level formats a floppy disk
4. floppy (8) - format floppy disks
5. gfloppy (1) - a simple floppy formatter for the GNOME
6. mbadblocks (1) - tests a floppy disk, and marks the bad
7. mformat (1) - add an MSDOS filesystem to a low-level
```

alias, unalias

alias 命令用来为命令起别名。

```
1. $ alias foo='cd /usr; ls; cd -'  
2.  
3. $ type foo  
4. foo is aliased to `cd /usr; ls ; cd -'
```

上面命令指定 **foo** 为三个命令的别名。以后，执行 **foo** 就相当于一起执行这三条命令。

注意，默认情况下，别名只在当前Session有效。当前Session结束时，这些别名就会消失。

alias 命令不加参数时，显示所有有效的别名。

```
1. $ alias  
2. alias l.='ls -d .* --color=tty'  
3. alias ll='ls -l --color=tty'  
4. alias ls='ls --color=tty'
```

unalias 命令用来取消别名。

```
1. $ unalias foo  
2. $ type foo  
3. bash: type: foo: not found
```

which

which 命令显示可执行程序的路径。

```
1. $ which ls  
2. /bin/ls
```

which 命令用于Shell内置命令时（比如 **cd** ），将没有任何输出。

help, man

help 命令用于查看Shell内置命令的帮助信息，**man** 命令用于查看可执行命令的帮助信息。

```
1. $ help cd  
2. $ man ls
```

man 里面的文档一共有8类，如果同一个命令，匹配多个文档，**man** 命令总是返回第一个匹配。如果想看指定类型的文档，命令可以采用下面的形式。

```
1. $ man 5 passwd
```

script

`script` 命令会将输入的命令和它的输出，都保存进一个文件。

```
1. $ script [file]
```

如果没有指定文件名，则所有结果会保存进当前目录下 `typescript` 文件。结束录制的时候，可以按下 `Ctrl + d`。

export

`export` 命令用于将当前进程的变量，输出到所有子进程。

命令的连续执行

多个命令可以写在一起。

Bash 提供三种方式，定义它们如何执行。

```
1. # 第一个命令执行完，执行第二个命令
2. command1; command2
3.
4. # 只有第一个命令成功执行完（退出码0），才会执行第二个命令
5. command1 && command2
6.
7. # 只有第一个命令执行失败（退出码非0），才会执行第二个命令
8. command1 || command2
```

上面三种执行方法的退出码，都是最后一条执行的命令的退出码。

bash 允许把命令组合在一起。可以通过两种方式完成；要么用一个 `group` 命令，要么用一个子 `shell`。 这里是每种方式的语法示例：

组命令：

```
1. { command1; command2; [command3; ...] }
```

子 shell

```
1. (command1; command2; [command3;...])
```

这两种形式的不同之处在于，组命令用花括号把它的命令包裹起来，而子 `shell` 用括号。值得注意的是，鉴于 `bash` 实现组命令的方式，花括号与命令之间必须有一个空格，并且最后一个命令必须用一个分号或者一个换行符终止。

那么组命令和子 shell 命令对什么有好处呢？它们都是用来管理重定向的。

```
1. { ls -l; echo "Listing of foo.txt"; cat foo.txt; } > output.txt
```

使用一个子 shell 是相似的。

```
1. (ls -l; echo "Listing of foo.txt"; cat foo.txt) > output.txt
```

组命令和子 shell 真正闪光的地方是与管道线相结合。当构建一个管道线命令的时候，通常把几个命令的输出结果合并成一个流是很有用的。组命令和子 shell 使这种操作变得很简单。

```
1. { ls -l; echo "Listing of foo.txt"; cat foo.txt; } | lpr
```

这里我们已经把我们的三个命令的输出结果合并在一起，并把它们用管道输送给命令 lpr 的输入，以便产生一个打印报告。

虽然组命令和子 shell 看起来相似，并且它们都能用来在重定向中合并流，但是两者之间有一个很重要的不同。然而，一个组命令在当前 shell 中执行它的所有命令，而一个子 shell（顾名思义）在当前 shell 的一个子副本中执行它的命令。这意味着运行环境被复制给了一个新的 shell 实例。当这个子 shell 退出时，环境副本会消失，所以在子 shell 环境（包括变量赋值）中的任何更改也会消失。因此，在大多数情况下，除非脚本要求一个子 shell，组命令比子 shell 更受欢迎。组命令运行很快并且占用的内存也少。

当我们发现管道线中的一个 read 命令不按我们所期望的那样工作的时候。为了重现问题，我们构建一个像这样的管道线：

```
1. echo "foo" | read
2. echo $REPLY
```

该 REPLY 变量的内容总是为空，是因为这个 read 命令在一个子 shell 中执行，所以它的 REPLY 副本会被毁掉，当该子 shell 终止的时候。因为管道线中的命令总是在子 shell 中执行，任何给变量赋值的命令都会遭遇这样的问题。幸运的是，shell 提供了一种奇异的展开方式，叫做进程替换，它可以用来解决这种麻烦。进程替换有两种表达方式：

一种适用于产生标准输出的进程：

```
1. <(list)
```

另一种适用于接受标准输入的进程：

```
1. >(list)
```

这里的 list 是一串命令列表：

为了解决我们的 read 命令问题，我们可以雇佣进程替换，像这样。

```
1. read < <(echo "foo")
```

```
2. echo $REPLY
```

进程替换允许我们把一个子 shell 的输出结果当作一个用于重定向的普通文件。事实上，因为它是一种展开形式，我们可以检验它的真实值：

```
1. [me@linuxbox ~]$ echo <(echo "foo")  
2. /dev/fd/63
```

通过使用 echo 命令，查看展开结果，我们看到子 shell 的输出结果，由一个名为 /dev/fd/63 的文件提供。

alias

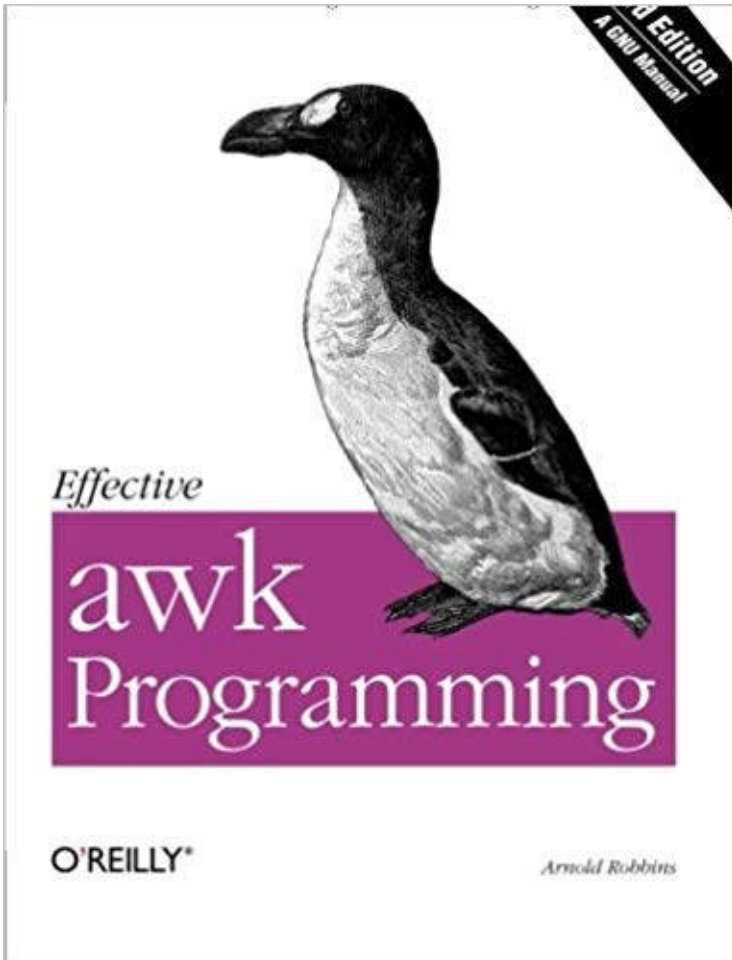
`alias` 命令用于设置别名。通常用于在 Bash 设置文件中，设置别名。

```
1. alias dockerlogin='ssh www-data@adnan.local -p2222'
```

awk

`awk` 是处理文本文件的一个应用程序，几乎所有 Linux 系统都自带这个程序。

它依次处理文件的每一行，并读取里面的每一个字段。对于日志、CSV 那样的每行格式相同的文本文件，`awk` 可能是最方便的工具。



`awk` 其实不仅仅是工具软件，还是一种编程语言。不过，这里只介绍它的命令行用法，对于大多数场合，应该足够用了。

基本用法

`awk` 的基本用法就是下面的形式。

1. `# 格式`
2. `$ awk 动作 文件名`
- 3.
4. `# 示例`
5. `$ awk '{print $0}' demo.txt`

上面示例中，`demo.txt` 是 `awk` 所要处理的文本文件。前面单引号内部有一个大括号，里面就是每一行的处理动作 `print $0`。其中，`print` 是打印命令，`$0` 代表当前行，因此上面命令的执行结果，就是把每一行原样打印出

来。

下面，我们先用标准输入（stdin）演示上面这个例子。

```
1. $ echo 'this is a test' | awk '{print $0}'
2. this is a test
```

上面代码中，`print $0` 就是把标准输入 `this is a test`，重新打印了一遍。

`awk` 会根据空格和制表符，将每一行分成若干字段，依次用 `$1`、`$2`、`$3` 代表第一个字段、第二个字段、第三个字段等等。

```
1. $ echo 'this is a test' | awk '{print $3}'
2. a
```

上面代码中，`$3` 代表 `this is a test` 的第三个字段 `a`。

下面，为了便于举例，我们把 `/etc/passwd` 文件保存成 `demo.txt`。

```
1. root:x:0:0:root:/root:/usr/bin/zsh
2. daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
3. bin:x:2:2:bin:/bin:/usr/sbin/nologin
4. sys:x:3:3:sys:/dev:/usr/sbin/nologin
5. sync:x:4:65534:sync:/bin:/bin/sync
```

这个文件的字段分隔符是冒号（`:`），所以要用 `-F` 参数指定分隔符为冒号。然后，才能提取到它的第一个字段。

```
1. $ awk -F ':' '{ print $1 }' demo.txt
2. root
3. daemon
4. bin
5. sys
6. sync
```

变量

除了 `$ + 数字` 表示某个字段，`awk` 还提供其他一些变量。

变量 `NF` 表示当前行有多少个字段，因此 `$NF` 就代表最后一个字段。

```
1. $ echo 'this is a test' | awk '{print $NF}'
2. test
```

`$(NF-1)` 代表倒数第二个字段。

```
1. $ awk -F ':' '{print $1, $(NF-1)}' demo.txt
2. root /root
```

```
3. daemon /usr/sbin
4. bin /bin
5. sys /dev
6. sync /bin
```

上面代码中，`print` 命令里面的逗号，表示输出的时候，两个部分之间使用空格分隔。

变量 `NR` 表示当前处理的是第几行。

```
1. $ awk -F ':' '{print NR " " $1}' demo.txt
2. 1) root
3. 2) daemon
4. 3) bin
5. 4) sys
6. 5) sync
```

上面代码中，`print` 命令里面，如果原样输出字符，要放在双引号里面。

`awk` 的其他内置变量如下。

- `FILENAME` : 当前文件名
- `FS` : 字段分隔符，默认是空格和制表符。
- `RS` : 行分隔符，用于分割每一行，默认是换行符。
- `OFS` : 输出字段的分隔符，用于打印时分隔字段，默认为空格。
- `ORS` : 输出记录的分隔符，用于打印时分隔记录，默认为换行符。
- `OFMT` : 数字输出的格式，默认为 `%.6g`。

函数

`awk` 还提供了一些内置函数，方便对原始数据的处理。

函数 `toupper()` 用于将字符转为大写。

```
1. $ awk -F ':' '{ print toupper($1) }' demo.txt
2. ROOT
3. DAEMON
4. BIN
5. SYS
6. SYNC
```

上面代码中，第一个字段输出时都变成了大写。

其他常用函数如下。

- `tolower()` : 字符转为小写。
- `length()` : 返回字符串长度。
- `substr()` : 返回子字符串。
- `sin()` : 正弦。
- `cos()` : 余弦。
- `sqrt()` : 平方根。
- `rand()` : 随机数。

awk 内置函数的完整列表，可以查看[手册](#)。

条件

awk 允许指定输出条件，只输出符合条件的行。

输出条件要写在动作的前面。

```
1. $ awk '条件 动作' 文件名
```

请看下面的例子。

```
1. $ awk -F ':' '/usr/ {print $1}' demo.txt
2. root
3. daemon
4. bin
5. sys
```

上面代码中，**print** 命令前面是一个正则表达式，只输出包含 **usr** 的行。

下面的例子只输出奇数行，以及输出第三行以后的行。

```
1. # 输出奇数行
2. $ awk -F ':' 'NR % 2 == 1 {print $1}' demo.txt
3. root
4. bin
5. sync
6.
7. # 输出第三行以后的行
8. $ awk -F ':' 'NR > 3 {print $1}' demo.txt
9. sys
10. sync
```

下面的例子输出第一个字段等于指定值的行。

```
1. $ awk -F ':' '$1 == "root" {print $1}' demo.txt
2. root
3.
4. $ awk -F ':' '$1 == "root" || $1 == "bin" {print $1}' demo.txt
5. root
6. bin
```

if 语句

awk 提供了 **if** 结构，用于编写复杂的条件。

```
1. $ awk -F ':' '{if ($1 > "m") print $1}' demo.txt
```

awk

```
2. root
3. sys
4. sync
```

上面代码输出第一个字段的第一个字符大于 `m` 的行。

`if` 结构还可以指定 `else` 部分。

```
1. $ awk -F ':' '{if ($1 > "m") print $1; else print "---}" demo.txt
2. root
3. ---
4. ---
5. sys
6. sync
```

参考链接

- [An Awk tutorial by Example](#), Greg Grothaus
- [30 Examples for Awk Command in Text Processing](#), Mokhtar Ebrahim

cal

`cal` 命令显示本月的日历。

```
1. $ cal
```

cat

`cat` 命令用于显示一个文本文件的内容。

`cat - >> filename` 用于向一个现有文件的尾部追加内容。

clear

`clear` 命令用来清除当前屏幕的显示，运行后会只留下一个提示符。

```
1. $ clear
```

cp 命令

`cp` 命令用于复制文件。

参数

`-u` 参数只复制那些目标目录里面还不存在的文件，以及那些虽然存在、但是比源目录对应文件更陈旧的文件。

cut

cut 命令用于在命令行输出文本文件的指定位置的内容。

它的使用格式如下。

```
1. $ cut OPTION [FILE]
```

如果没有指定文件名，将读取标准输入。

-b 参数用来指定读取的字节。

```
1. # 输出前三个字节
2. $ cut file1.txt -b1,2,3
3.
4. # 输出前十个字节
5. $ cut file1.txt -b1-10
6.
7. # 输出从第5个字节开始的所有字节
8. $ cut file1.txt -b5-
9.
10. # 输出前5个字节
11. $ cut file1.txt -b-5
```

-c 参数用来指定读取的字符，用法与 **-b** 一样。有的字符是多字节字符，这时候就应该用 **-c** 代替 **-b**。

-d 参数用来指定分隔符，默认分隔符为制表符。

-f 参数用来指定字段。

```
1. # 指定每一行的分隔符为逗号，
2. # 输出第一和第三个字段
3. $ cut file1.txt -d, -f1,3
4.
5. # 输出第一、第二、第四和第五个字段
6. $ cut -f 1-2,4-5 data.txt
```

date

`date` 命令显示当前的日期和时间。

```
1. $ date
```

dd

dd 命令用于复制磁盘或文件系统。

复制磁盘

```
1. $ dd if=/dev/sda of=/dev/sdb
```

上面命令表示将 `/dev/sda` 磁盘复制到 `/dev/sdb` 设备。参数 `if` 表示来源地，`of` 表示目的地。

除了复制，**dd** 还允许将磁盘做成一个镜像文件。

```
1. $ dd if=/dev/sda of=/home/username/sdadisk.img
```

dd 还可以复制单个分区。

```
1. $ dd if=/dev/sda2 of=/home/username/partition2.img bs=4096
```

上面命令中，参数 `bs` 表示单次拷贝的字节数（bytes）。

要将镜像文件复原，也很简单。

```
1. $ dd if=sdadisk.img of=/dev/sdb
```

清除数据

dd 也可以用于清除磁盘数据。

```
1. # 磁盘数据写满 0
2. $ dd if=/dev/zero of=/dev/sda1
3.
4. # 磁盘数据写满随机字符
5. $ dd if=/dev/urandom of=/dev/sda1
```

监控进展

磁盘的复制通常需要很久，为了监控进展，可以使用 Pipe Viewer 工具软件。如果没有安装这个软件，可以使用下面的命令安装。

```
1. $ sudo apt install pv
```

然后，来源地和目的地之间插入管道，就可以看到进展了。

```
1. $ dd if=/dev/urandom | pv | dd of=/dev/sda1
2. 4,14MB 0:00:05 [ 98kB/s] [      <=>      ]
```

参考链接

- David Clinton, [How to use dd in Linux without destroying your disk](#)

df

df 命令显示磁盘信息。

du

du 命令显示某个文件或目录的磁盘使用量。

```
1. $ du filename
```

-h 参数将返回的大小显示为人类可读的格式，即显示单位为 K、M、G 等。

-s 参数表示总结 (summarize)。

-x 参数表示不显示不在当前分区的目录，通常会忽略 `/dev` 、 `/proc` 、 `/sys` 等目录。

-c 参数表示显示当前目录总共占用的空间大小。

```
1. # 显示根目录下各级目录占用的空间大小
2. $ sudo du -shxc /*
```

--exclude 参数用于排除某些目录或文件。

```
1. $ sudo du -shxc /* --exclude=proc
2. $ sudo du -sh --exclude=*.iso
```

--max-depth 参数用于设定目录大小统计到第几层。如果设为 `--max-depth=0` ，那么等同于 **-s** 参数。

```
1. $ sudo du /home/ -hc --max-depth=2
```

egrep

`egrep` 命令用于显示匹配正则模式的行，与 `grep -E` 命令等价。

下面是 `example.txt` 文件的内容。

```
1. Lorem ipsum
2. dolor sit amet,
3. consetetur
4. sadipscing elitr,
5. sed diam nonumy
6. eirmod tempor
7. invidunt ut labore
8. et dolore magna
9. aliquyam erat, sed
10. diam voluptua. At
11. vero eos et
12. accusam et justo
13. duo dolores et ea
14. rebum. Stet clita
15. kasd gubergren,
16. no sea takimata
17. sanctus est Lorem
18. ipsum dolor sit
19. amet.
```

`egrep` 命令显示包括 `Lorem` 或 `dolor` 的行。

```
1. $ egrep '(Lorem|dolor)' example.txt
2. # 或者
3. $ grep -E '(Lorem|dolor)' example.txt
4. Lorem ipsum
5. dolor sit amet,
6. et dolore magna
7. duo dolores et ea
8. sanctus est Lorem
9. ipsum dolor sit
```

export

`export` 命令用于向子Shell输出变量。

```
1. $ export hotellogs="/workspace/hotel-api/storage/logs"
```

然后执行下面的命令，新建一个子 Shell。

```
1. $ bash
2. $ cd hotellogs
```

上面命令的执行结果会进入 `hotellogs` 变量指向的目录。

`export` 命令还可以显示所有环境变量。

```
1. $ export
2. SHELL=/bin/zsh
3. AWS_HOME=/Users/adnanadnan/.aws
4. LANG=en_US.UTF-8
5. LC_CTYPE=en_US.UTF-8
6. LESS=-R
```

如果想查看单个变量，使用 `echo $VARIABLE_NAME` 。

```
1. $ echo $SHELL
2. /usr/bin/zsh
```

file

file 命令用来某个文件的类型。

```
1. $ file index.html
2.  index.html: HTML document, ASCII text
```

file 工具可以对所给的文件一行简短的介绍，它用文件后缀、头部信息和一些其他的线索来判断文件。你在检查一堆你不熟悉的文件时使用 **find** 非常方便：

```
1. $ find -exec file {} \;
2.  .:          directory
3.  ./hanoi:    Perl script, ASCII text executable
4.  ./hanoi.swp: Vim swap file, version 7.3
5.  ./factorial: Perl script, ASCII text executable
6.  ./bits.c:   C source, ASCII text
7.  ./bits:     ELF 32-bit LSB executable, Intel 80386, version ...
```

find

`find` 命令用于寻找文件，会包括当前目录的所有下级目录。

如果不带任何参数，`find` 文件会列出当前目录的所有文件，甚至还包括相对路径。如果把结果导入 `sort` 效果更好。

```
1. $ find | sort
2. .
3. ./Makefile
4. ./README
5. ./build
6. ./client.c
7. ./client.h
8. ./common.h
9. ./project.c
10. ./server.c
11. ./server.h
12. ./tests
13. ./tests/suite1.pl
14. ./tests/suite2.pl
15. ./tests/suite3.pl
16. ./tests/suite4.pl
```

如果想要 `ls -l` 样式的列表，只要在 `find` 后面加上 `-ls`。

```
1. $ find -ls
```

`find` 有它自己的一套复杂的过滤语句。下面列举的是一些最常用的你可以用以获取某些文件列表的过滤器：

- `find -name '*.c'` — 查找符合某 shell 式样式的文件名的文件。用 `iname` 开启大小写不敏感搜索。
- `find -path 'test'` — 查找符合某 shell 式样式的路径的文件。用 `ipath` 开启大小写不敏感搜索。
- `find -mtime -5` — 查找近五天内编辑过的文件。你也可以用 `+5` 来查找五天之前编辑过的文件。
- `find -newer server.c` — 查找比 `server.c` 更新的文件。
- `find -type d` — 查找所有文件夹。如果想找出所有文件，那就用 `-type f`；找符号连接就用 `-type l`。

要注意，上面提到的这些过滤器都是可以组合使用的，例如找出近两天内编辑过的 C 源码：

```
1. $ find -name '*.c' -mtime -2
```

默认情况下，`find` 对搜索结果所采取的动作只是简单地通过标准输出输出一个列表，然而其实还有其他一些有用的后续动作。

- `-ls` — 如前文，提供了一种 `ls -l` 式的列表。
- `-delete` — 删除符合查找条件的文件。
- `-exec` — 对搜索结果里的每个文件都运行某个命令，`{}` 会被替换成适当的文件名，并且命令用 `\;` 终结。

find

```
1. $ find -name '*.pl' -exec perl -c {} \;
```

你也可以使用 `+` 作为终止符来对所有结果运行一次命令。我还发现一个我经常使用的小技巧，就是用 `find` 生成一个文件列表，然后在 Vim 的垂直分窗中编辑：

```
1. $ find -name '*.c' -exec vim {} +
```

fmt

fmt 命令用于对文本指定样式。

下面是 `example.txt` 的内容，是非常长的一行。

```
1. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.
```

fmt 可以将其输出为每行80个字符。

```
1. cat example.txt | fmt -w 20
2. Lorem ipsum
3. dolor sit amet,
4. consetetur
5. sadipscing elitr,
6. sed diam nonumy
7. eirmod tempor
8. invidunt ut labore
9. et dolore magna
10. aliquyam erat, sed
11. diam voluptua. At
12. vero eos et
13. accusam et justo
14. duo dolores et ea
15. rebum. Stet clita
16. kasd gubergren,
17. no sea takimata
18. sanctus est Lorem
19. ipsum dolor sit
20. amet.
```


grep

grep 命令用于文件内容的搜索，返回所有匹配的行。

```
1. $ grep pattern filename
```

下面是一个例子。

```
1. $ grep admin /etc/passwd
2. _kadmin_admin:*:218:-2:Kerberos Admin Service:/var/empty:/usr/bin/false
3. _kadmin_changepw:*:219:-2:Kerberos Change Password Service:/var/empty:/usr/bin/false
4. _krb_kadmin:*:231:-2:Open Directory Kerberos Admin Service:/var/empty:/usr/bin/false
```

一般情况下，应该使用 **grep -R**，递归地找出当前目录下符合 **someVar** 的文件。

```
1. $ grep -FR 'someVar' .
```

别忘了大小不敏感的参数，因为 **grep** 默认搜索是大小写敏感的。

```
1. $ grep -iR 'somevar' .
```

也可以用 **grep -l** 光打印出符合条件的文件名而非文件内容选段。

```
1. $ grep -lR 'somevar' .
```

如果你写的脚本或批处理任务需要上面的输出内容，可以使用 **while** 和 **read** 来处理文件名中的空格和其他特殊字符：

```
1. grep -lR someVar | while IFS= read -r file; do
2.     head "$file"
3. done
```

如果你在你的项目里使用了版本控制软件，它通常会在 **.svn**，**.git**，**.hg** 目录下包含一些元数据。你也可以很容易地用 **grep -v** 把这些目录移出搜索范围，当然得用 **grep -F** 指定一个恰当且确定的字符串，即要移除的目录名：

```
1. $ grep -R 'someVar' . | grep -vF '.svn'
```

部分版本的 **grep** 包含了 **-exclude** 和 **-exclude-dir** 选项，这看起来更加易读。

参数

-i 参数表示忽略大小写。

`-r` 表示搜索某个目录下面的所有文件。

```
1. $ grep -r admin /etc/
```

`-v` 过滤包含某个词的行，即 `grep` 的逆操作。

```
1. # 显示所有包含 vim, 但不包含 grep 的行
2. $ ps | grep vim | grep -v grep
```

gunzip

`gunzip` 命令用于解压 `gzip` 命令压缩的文件。

gzcat

`gzcat` 命令用于查看一个 `gz` 文件，但并不实际解压它。

```
1. $ gzcat filename
```

gzip

`gzip` 命令用于压缩文件。

kill

kill 命令用户终止指定进程。

```
1. $ kill PID
```

killall

killall 命令终止给定名字的一系列相关进程。

```
1. $ killall processname
```

last

last 命令显示用户登录系统的记录。

```
1. $ last
```

last 命令后面加上用户名，会显示该用户上次登录的信息。

```
1. $ last yourUsername
```


lpq

lpq 命令显示打印机队列。

```
1. $ lpq
2. Rank      Owner    Job      File(s)      Total Size
3. active    adnanad  59       demo         399360 bytes
4. 1st       adnanad  60       (stdin)      0 bytes
```

lpr

lpr 命令用于打印文件。

```
1. lpr filename
```

ls

ls 命令用于列出当前目录里面的文件和子目录。

参数

- a: 列出隐藏文件
- l: 以长格式列出文件
- t: 按最后编辑日期排序，最新的最先。这在某个大目录里找出最近修改的文件列表时很有用，比如将结果导入（ pipe ） head 或者 sed 10q。或许加上 -l 会效果更好。当然如果你想获取最旧的文件列表，只要加 -r 反转列表即可。
- X: 按文件类型分类。这在多语言或多后缀的项目中特别方便，比如头文件和源文件分开，或区分开源文件和生成文件或目录。
- v: 按照文件名里的版本号排序。
- S: 按文件大小排序。
- R: 递归地列举文件。这个选项和 -l 组合使用并将结果导出到 less 效果很好。

可以把结果导出给类似 vim 的进程。

```
1. $ ls -XR | vim -
```

nl

nl 命令用于显示行号。

下面是 `example.txt` 文件的内容。

```
1. Lorem ipsum
2. dolor sit amet,
3. consetetur
4. sadipscing elitr,
5. sed diam nonumy
6. eirmod tempor
7. invidunt ut labore
8. et dolore magna
9. aliquyam erat, sed
10. diam voluptua. At
11. vero eos et
12. accusam et justo
13. duo dolores et ea
14. rebum. Stet clita
15. kasd gubergren,
16. no sea takimata
17. sanctus est Lorem
18. ipsum dolor sit
19. amet.
```

nl 命令让上面这段文本显示行号。

```
1. $ nl -s". " example.txt
2.     1. Lorem ipsum
3.     2. dolor sit amet,
4.     3. consetetur
5.     4. sadipscing elitr,
6.     5. sed diam nonumy
7.     6. eirmod tempor
8.     7. invidunt ut labore
9.     8. et dolore magna
10.    9. aliquyam erat, sed
11.   10. diam voluptua. At
12.   11. vero eos et
13.   12. accusam et justo
14.   13. duo dolores et ea
15.   14. rebum. Stet clita
16.   15. kasd gubergren,
17.   16. no sea takimata
18.   17. sanctus est Lorem
19.   18. ipsum dolor sit
20.   19. amet.
```

nl

-s 参数表示行号的后缀。

ps

`ps` 命令列出当前正在执行的进程信息。

由于进程很多，所以为了快速找到某个进程，一般与 `grep` 配合使用。

1. # 找出正在运行 vim 的进程
2. \$ ps | grep vi

参数

`-u` 参数列出指定用户拥有的进程。

1. \$ ps -u yourusername

scp

基本用法

`scp` 是 `secure copy` 的缩写，用来在两台主机之间加密传送文件。它的底层是 `SSH` 协议，默认端口是22。

它主要用于以下三种复制操作。

- 从本地系统到远程系统。
- 从远程系统到本地系统。
- 在本地系统的两个远程系统之间。

使用 `scp` 传输数据时，文件和密码都是加密的，不会泄漏敏感信息。

`scp` 的语法类似 `cp` 的语法。

注意，如果传输的文件在本机和远程系统，有相同的名称和位置，`scp` 会在没有警告的情况下覆盖文件。

（1）本地文件复制到远程系统

复制本机文件到远程系统的基本语法如下。

```
1. # 语法
2. $ scp SourceFile user@host:directory/TargetFile
3.
4. # 示例
5. $ scp file.txt remote_username@10.10.0.2:/remote/directory
```

下面是复制整个目录。

```
1. # 将本机的 documents 目录拷贝到远程主机，
2. # 会在远程主机创建 documents 目录
3. $ scp -r documents username@server_ip:/path_to_remote_directory
4.
5. # 将本机整个目录拷贝到远程目录下
6. $ scp -r localmachine/path_to_the_directory username@server_ip:/path_to_remote_directory/
7.
8. # 将本机目录下的所有内容拷贝到远程目录下
9. $ scp -r localmachine/path_to_the_directory/* username@server_ip:/path_to_remote_directory/
```

（2）远程文件复制到本地

从远程主机复制文件到本地的语法如下。

```
1. # 语法
2. $ scp user@host:directory/SourceFile TargetFile
3.
4. # 示例
```

```
5. $ scp remote_username@10.10.0.2:/remote/file.txt /local/directory
```

下面是复制整个目录的例子。

```
1. # 拷贝一个远程目录到本机目录下
2. $ scp -r username@server_ip:/path_to_remote_directory local-machine/path_to_the_directory/
3.
4. # 拷贝远程目录下的所有内容，到本机目录下
5. $ scp -r username@server_ip:/path_to_remote_directory/* local-machine/path_to_the_directory/
6. $ scp -r user@host:directory/SourceFolder TargetFolder
```

(3) 两个远程系统之间的复制

本机发出指令，从远程主机 A 拷贝到远程主机 B 的语法如下。

```
1. # 语法
2. $ scp user@host1:directory/SourceFile user@host2:directory/SourceFile
3.
4. # 示例
5. $ scp user1@host1.com:/files/file.txt user2@host2.com:/files
```

系统将提示您输入两个远程帐户的密码。数据将直接从一个远程主机传输到另一个远程主机。

参数

-P 用来指定远程主机的 SSH 端口。如果远程主机使用非默认端口22，可以在命令中指定。

```
1. $ scp -P 2222 user@host:directory/SourceFile TargetFile
```

-p 参数用来保留修改时间 (modification time)、访问时间 (access time)、文件状态 (mode) 等原始文件的信息。

```
1. $ scp -C -p ~/test.txt root@192.168.1.3:/some/path/test.txt
```

-l 参数用来限制传输数据的带宽速率，单位是 Kbit/sec。对于多人分享的带宽，这个参数可以留出一部分带宽供其他人使用。

```
1. $ scp -l 80 yourusername@yourserver:/home/yourusername/* .
```

上面代码中，**scp** 命令占用的带宽限制为每秒80K比特位，即每秒10K字节。

-c 参数用来指定加密算法。

```
1. $ scp -c blowfish some_file your_username@remotehost.edu:~
```

上面代码指定加密算法为 **blowfish**。

-c 表示是否在传输时压缩文件。

```
1. $ scp -c blowfish -C local_file your_username@remotehost.edu:~
```

-q 参数用来关闭显示拷贝的进度条。

```
1. $ scp -q Label.pdf mrariantto@202.x.x.x:.
```

-F 参数用来指定 ssh_config 文件。

```
1. $ scp -F /home/pungki/proxy_ssh_config Label.pdf
```

-v 参数用来显示详细的输出。

```
1. $ scp -v ~/test.txt root@192.168.1.3:/root/help2356.txt
```

-i 参数用来指定密钥。

```
1. $ scp -vCq -i private_key.pem ~/test.txt root@192.168.1.3:/some/path/test.txt
```

-r 参数表示是否以递归方式复制目录。

sed

`sed` 命令用于对文本进行过滤和变形处理。

下面是 `example.txt` 文件的内容。

1. `Hello This is a Test 1 2 3 4`
2. `replace all spaces with hyphens`

`sed` 命令将所有的空格换成连词线 `-` 。

1. `$ sed 's/ /-/g' example.txt`
2. `Hello-This-is-a-Test-1-2-3-4`

下面的命令将数字换成字母 `d` 。

1. `$ sed 's/[0-9]/d/g' example.txt`
2. `Hello This is a Test d d d d`

sort

`sort` 命令用于文本文件的排序。

下面是 `example.txt` 文件的内容。

```
1. f
2. b
3. c
4. g
5. a
6. e
7. d
```

执行 `sort` 命令对其进行排序。

```
1. $ sort example.txt
2. a
3. b
4. c
5. d
6. e
7. f
8. g
```

参数

`-R` 参数表示随机排序。

```
1. sort -R example.txt
2. b
3. d
4. a
5. c
6. g
7. e
8. f
```

tr

tr 命令用于按照给定模式转换文本。

下面是 `example.txt` 文件的内容。

```
1. Hello World Foo Bar Baz!
```

tr 命令可以将所有小写字母转换为大写字母。

```
1. $ cat example.txt | tr 'a-z' 'A-Z'
2. HELLO WORLD FOO BAR BAZ!
```

tr 命令还可以将所有空格转为换行符。

```
1. $ cat example.txt | tr ' ' '\n'
2. Hello
3. World
4. Foo
5. Bar
6. Baz!
```

uname

`uname` 命令用来显示内核信息。

```
1. $ uname -a
```

uniq

uniq 用于过滤掉重复的行，该命令只对排序后的文件有效。

下面是 `example.txt` 文件的内容。

```
1. a
2. a
3. b
4. a
5. b
6. c
7. d
8. c
```

对该文件进行排序后，再过滤掉重复的行。

```
1. $ sort example.txt | uniq
2. a
3. b
4. c
5. d
```

参数


-c 参数会显示每行一共出现了多少次。

```
1. sort example.txt | uniq -c
2.   3 a
3.   2 b
4.   2 c
5.   1 d
```

uptime

`uptime` 命令显示本次开机运行的时间。

W

 命令显示当期谁在线。

WC

WC 命令返回某个文件的行数、词数和字符数。

```
1. $ wc demo.txt
2. 7459 15915 398400 demo.txt
```

上面代码中，**7459** 是行数，**15915** 是词数，**398400** 是字符数。

whereis

whereis 用来显示某个命令的位置。如果有多个程序符合条件，会全部列出。

1. `$ whereis node`
2. `/usr/bin/node /usr/sbin/node`

which

`which` 命令根据 `PATH` 环境变量指定的顺序，返回最早发现某个命令的位置。即不指定路径时，实际执行的命令的完整路径。

1. `$ which node`
2. `/usr/bin/node`

who

who 命令显示已经登录的用户。

参数

-b 参数显示上一次系统启动的时间。

1. `$ who -b`
2. `system boot 2017-06-20 17:41`