ch03 bash shell基础命令

很多Linux发行版的默认shell是GNU bash shell。本章将介绍bash shell的基本特性，比如bash手册、命令行补全以及如何显示文件内容。我们会带你逐步了解怎样用bash shell提供的基础命令来处理Linux文件和目录（我骗你的，我会跳过很多基础，这一章是为了完整性出的。。。）

1. shell！！启动！！！！！！

GNU bash shell是一个程序，提供了对Linux系统的交互式访问。它是作为普通程序运行的，通常是在用户登录终端时启动。系统启动的shell程序取决于用户帐户的配置。
/etc/passwd 文件包含了所有系统用户帐户以及每个用户的基本配置信息。下面是从 */etc/passwd* 文件中摘取的样例条目：

```
lxc@Lxc:~/scripts/ch03-bash shell基础命令$ cat /etc/passwd | grep lxc
lxc:x:1000:1000:Lxc,,,:/home/lxc:/bin/bash
```

其中，每个条目包含7个字段，字段之间使用冒号作为分隔。最后一个字段指定了该用户使用的shell程序。lxc用户使用的是/bin/bash。
尽管bash shell会在登陆时自行启动，但是否会出现命令行界面（CLI）取决于所使用的登录方式。如果采用的是虚拟控制台终端登录，那么CLI提示符会自动出现，接受shell命令输入。但如果是通过图形化桌面环境登录Linux系统，则需要启动图形化终端仿真器来访问shell CLI提示符（你多半用的是图形化桌面环境，按Ctrl+Alt+t就行了）。

2. 使用shell提示符

shell提示符并非是一成不变的。你可以根据需要修改提示符（通过 `PS1` 环境变量）。shell CLI提示符用于告诉你什么时候shell可以接受新的命令。

3. 与bash手册交互

手册页将与命令相关的信息分成了多段。每一段的惯用名称标准如下表所示。

如果不记得命令名了，可以使用关键字来搜索手册页。语法为 `man -k keyword`。例如，要查找与终端相关的命令，可以输入 `man -k terminal`。

除了按照惯例命名的各段，手册页还有不同的节。每节都分配了一个数字，从1开始，一直到9，如下表所示。

`man` 命令通常显示的是指定命令的最低的节。例如，我们在下图中输入的是 `man hostname`，注意在显示内容的左上角和右上角，单词 `HOSTNAME` 后的圆括号中有一个数字：（1）。这表示所显示的手册页来自第1节（可执行程序或shell命令）。

> **注意：** 你的Linux系统手册页可能包含一些非标准的节编号。例如 1p 对应于可移植操作系统接口（portable operating system interface，POSIX）命令，3n 对应于网络函数。

一个命令偶尔会在多个节中都有对应的手册页。比如，`hostname` 命令的手册页既包含该命令的相关信息，也包括对系统主机名的概述。要想查看所需的页面，可以使用下面格式的命令：

```
man section# topicname
```

因此，输入 `man 7 hostname`，可以查看手册页中的第7节。也可以使用 `man -a hostname`，你用一次就知道了。
你也可以只看各节内容的简介：输入 `man 1 intro` 阅读第1节的简介（就是说明手册页的第一节一般会放置什么样的内容）；`man 2 intro` 阅读第2节的简介；.....`man 8 intro` 阅读第8节的简介。
输入 `man man` 来查看与手册页相关的信息。
手册页并非唯一的参考资料。还有另一种称作 `info` 页面的信息。可以输入 `info info` 来了解 info 页面的相关内容。
内建命令有自己的帮助页面。有关帮助页面的的更多信息可以输入 `help help` 来了解。
另外，大多数命令接受 `-h` 或 `--help` 选项。例如，可以输入 `hostname --help` 来查看简要的帮助信息。

4. 浏览文件系统

1. Linux文件系统

Linux会将文件存储在名为 **虚拟目录（virtual directory）** 的单个目录结构中。虚拟目录会将计算机中所有存储设备的文件路径都纳入单个目录结构。
Linux虚拟目录结构只包含一个称为 **根（root）** 目录的基础目录。根目录下的目录和文件会按照其访问路径一一列出，路径本身并没有提供任何有关文件究竟存放在哪个物理磁盘中的信息。
Linux虚拟目录中比较复杂的部分是它如何来协调管理各个存储设备。我们称在Linux系统中安装的第一块硬盘为 **根驱动器**。根驱动器包含了虚拟目录的核心，其他目录都是从那里开始构建的。
Linux会使用根驱动器上一些特别的目录作为 **挂载点（mount point）**。挂载点是虚拟目录中分配给额外存储设备的目录。Linux会让文件和目录出现在这些挂载点目录中，即便它们位于其他物理驱动器中。系统文件通常存储在根驱动器中，而用户文件则存储在其它存储器中。如下图所示。

上图展示了计算机中的两块硬盘。一块硬盘（Disk 1）与虚拟目录的根目录关联。其他硬盘可以挂载到虚拟目录结构中的任何地方。在这个例子中，另一块硬盘（Disk 2）被挂载到了 /home，这是用户主目录所在的位置。
Linux文件系统演进自Unix文件系统。在Linux文件系统中，采用通用的目录名表示一些常见的功能。下表列出了一些较常见的Linux顶层虚拟目录名及其内容。

2. 遍历目录

在Linux文件系统中，你可以使用目录切换（ `cd` ）命令来将shell会话切换到另一个目录。`cd` 命令的语法非常简单：

```
cd destination
```

`cd` 命令可以接受单个参数的 *destination*，用以指定你想切换到的目录名。如果没有为 `cd` 命令指定目标路径，则会切换到你的用户主目录。
destination 参数可以用两种方法表示，一种是绝对路径，一种是相对路径。

1. 绝对路径

绝对路径 总是以正斜线开始，以指明虚拟文件系统的根目录。

2. 相对路径

相对路径 允许你指定一个基于当前位置的目标路径。相对路径不以代表根目录的正斜线开头，而是以目录名（如果你准备切换到当前目录下的某个目录的话）或是一个特殊字符开始。

有两个特殊字符可以用于相对路径中。

- 单点号（.），表示当前目录
- 双点号（..），表示当前目录的父目录

5. 列出文件和目录

本节将讲述 `ls` 命令和可用来格式化其输出信息的选项。

1. 显示基本列表

`ls` 命令最基本的形式会显示当前目录下的文件和目录。

```
lxc@Lxc:~/scripts/ch03-bash shell基础命令$ ls
directory.png  fs.png  hsotname.png  man.png  README.md  section1.png
section2.png
```

注意，`ls` 命令输出的列表是按字母排序的（按列而不是按行排序）。如果你使用终端仿真器支持色彩显示，那么 `ls` 命令还可以用不同的颜色来区分不同类型的文件。`LS_COLORS` 环境变量（第6章会介绍环境变量）控制着这个特性。不同的Linux发行版会根据各自终端仿真器的能力来设置该环境变量。
也可以使用 `ls` 命令的 `-F` 选项来轻松地区分文件和目录。使用 `-F` 选项会得到以下输出：

```
lxc@Lxc:~$ ls -F
1.txt*   模板/  图片/  下载/  桌面/  C++/
```

`-F` 选项会在目录名之后添加正斜线，以方便用户在输出中分辨。类似地，它还会在可执行文件之后添加星号（如上面的1.txt）。
Linux经常使用 **隐藏文件** 来保存配置信息。在Linux中，隐藏文件通常是文件名以点号开始的文件。这些文件并不会在 `ls` 命令的默认输出中出现。因此，我们称其为隐藏文件。要显示隐藏文件，可以使用 `-a` 选项。

```
lxc@Lxc:~$ ls -a
.       .bash_aliases  .dbus       Go          .mysql_history     .redhat
               .themes
..      .bash_history  .designer   .gtkrc-2.0  .npm
.rediscli_history          .thunderbird
公共的  .bashrc        Documents   .icons      .nvm               .ros
              tt
```

`-R` 是递归选项，可以列出当前目录及其子目录所包含的文件和目录。

2. 显示长列表

`ls` 命令 `-l` 选项可以显示长列表格式的输出。

```
lxc@Lxc:~$ ls -lF
总用量 96
drwxr-xr-x  2 lxc lxc 4096 6月   8 17:04 公共的/
drwxr-xr-x  2 lxc lxc 4096 12月  1 12:10 模板/
drwxr-xr-x  2 lxc lxc 4096 6月  26 20:36 视频/
drwxrwxr-x  5 lxc lxc 4096 12月  1 21:27 图片/
drwxr-xr-x  4 lxc lxc 4096 10月 22 12:07 文档/
drwx------ 36 lxc lxc 4096 12月  1 12:11 下载/
drwxr-xr-x  2 lxc lxc 4096 6月   8 17:04 音乐/
drwxr-xr-x  2 lxc lxc 4096 12月  1 12:10 桌面/
```

输出的第一行显示了为该目录中的文件所分配的总块数。此后的每一行都包含了关于文件或目录的下列信息。

- 文件类型，比如目录（d）、文件（-）、链接文件（l）、字符设备（c）、块设备（b）
- 文件的权限（参加第7章）
- 文件的硬链接数（参加3.6.4）
- 文件属主
- 文件属组
- 文件大小（以字节为单位）
- 文件的上次修改时间
- 文件名或目录名

如果想查看单个文件的长列表，那么只需在 `ls -l` 命令之后跟上该文件名即可。如果想查看目录的相关信息，而非目录所包含的内容，则除了 `-l` 选项之外，还得添加 `-d` 选项。即 `ls -ld Directory-Name`。

```
lxc@Lxc:~$ ls -ld tt/
drwxrwxr-x 2 lxc lxc 4096 11月 29 14:44 tt/
```

3. 过滤输出列表

当指定特定的文件名作为过滤器时，`ls` 命令只显示该文件的信息。有时你可能不知道要找的那个文件的确切名称。`ls` 命令也能识别标准通配符。

- 问号(?)代表任意单个字符
- 星号(*)代表零个或多个字符

问号可以代表过滤器字符串中任意位置的单个字符。

```
lxc@Lxc:~/tt$ ls
fall  fell  fill  full  my_file  my_scrapt  my_script
lxc@Lxc:~/tt$ ls -l my_scr?pt
-rw-rw-r-- 1 lxc lxc 0 12月  2 11:41 my_scrapt
-rw-rw-r-- 1 lxc lxc 0 12月  2 11:41 my_script
```

星号可用来匹配零个或多个字符。

```
lxc@Lxc:~/tt$ ls my*
my_file  my_scrapt  my_script
```

在过滤器中使用星号和问号被称作 **通配符匹配（globbing）**，是指使用通配符进行模式匹配的过程。通配符正式的名称叫做 **元字符通配符（metacharacter wildcard）**。除了星号和问号，还有更多的元字符通配符可做文件匹配之用。

> 这里说明一下 globbing 和 wildcard 的区别：globbing 是对 wildcard 进行扩展的过程。在贝尔实验室诞生的Unix中，有一个名为 glob(global的简写) 的独立程序(/etc/glob)。早期的Unix版本（第1～6版，1960年～1975年）的命令解释器（也就是shell）都要依赖于该程序扩展命令中未被引用的 wildcard，然后将扩展后的结果传给命令执行。因此，本书将 globbing 译作 "通配符匹配"（也就是展开命令的过程），将 "wildcard" 译作 "通配符"。

关于通配符与正则表达式，参见第20章。

可以试试方括号，方括号代表单个字符位置上的多个可能的选择：

```
lxc@Lxc:~/tt$ touch my_scrypt
lxc@Lxc:~/tt$ ls -l my_scr[ay]pt
-rw-rw-r-- 1 lxc lxc 0 12月  2 11:41 my_scrapt
-rw-rw-r-- 1 lxc lxc 0 12月  2 11:48 my_scrypt
```

你可以像上面那样将可能的字符逐一列出，也可以指定字符范围，比如字母范围[a-i]：

```
lxc@Lxc:~/tt$ ls -l f*ll
-rw-rw-r-- 1 lxc lxc 0 12月  2 11:41 fall
-rw-rw-r-- 1 lxc lxc 0 12月  2 11:41 fell
-rw-rw-r-- 1 lxc lxc 0 12月  2 11:41 fill
-rw-rw-r-- 1 lxc lxc 0 12月  2 11:41 full
lxc@Lxc:~/tt$ ls -l f[a-i]ll
-rw-rw-r-- 1 lxc lxc 0 12月  2 11:41 fall
-rw-rw-r-- 1 lxc lxc 0 12月  2 11:41 fell
-rw-rw-r-- 1 lxc lxc 0 12月  2 11:41 fill
```

还可以使用惊叹号（!）来将不需要的内容排除：

```
lxc@Lxc:~/tt$ ls -l f[!a]ll
-rw-rw-r-- 1 lxc lxc 0 12月  2 11:41 fell
-rw-rw-r-- 1 lxc lxc 0 12月  2 11:41 fill
-rw-rw-r-- 1 lxc lxc 0 12月  2 11:41 full
```

6. 处理文件

本节将带你逐步了解文件处理所需要的一些基本的shell命令。

1. 创建文件

`touch` 命令可以创建空文件。该命令会将你的用户名作为文件的属主。
`touch` 命令还可以用来改变文件的修改时间。该操作不会改变文件内容。

```
lxc@Lxc:~/tt$ ls -l fall
-rw-rw-r-- 1 lxc lxc 0 12月  2 11:41 fall
lxc@Lxc:~/tt$ touch fall
lxc@Lxc:~/tt$ ls -l fall
-rw-rw-r-- 1 lxc lxc 0 12月  2 12:03 fall
```

2. 复制文件

`cp` 命令最基本的用法只需要两个参数，即源对象和目标对象：`cp source destination`。当参数 *source* 和 *destination* 都是文件名时，`cp` 命令会将源文件复制成一个新的目标文件，并以 *destination* 命名。新文件在形式上就像全新文件一样，有新的修改时间。

```
lxc@Lxc:~/tt$ ls
fall  fell  fill  full  my_file  my_scrapt  my_script  my_scrypt  test_one
lxc@Lxc:~/tt$ cp test_one test_two
lxc@Lxc:~/tt$ ls -l test_one test_two
-rw-rw-r-- 1 lxc lxc 0 12月  2 14:38 test_one
-rw-rw-r-- 1 lxc lxc 0 12月  2 14:39 test_two
```

如果目标文件已经存在，则 `cp` 命令可能并不会提醒你这一点。最好加上 `-i` 选项，强制shell询问是否需要覆盖某个已有文件：

```
lxc@Lxc:~/tt$ cp -i test_one test_two
cp: 是否覆盖'test_two'？ n
```

如果不回答 y，则停止文件复制。

看几个例子吧：

```
lxc@Lxc:~/tt$ mkdir Documents
lxc@Lxc:~/tt$ ls -F
Documents/  fall  fell  fill  full  my_file  my_scrapt  my_script  my_scrypt
test_one  test_two
lxc@Lxc:~/tt$ cp test_one Documents/
lxc@Lxc:~/tt$ cp test_two /home/lxc/tt/Documents/
lxc@Lxc:~/tt$ cd Documents/
lxc@Lxc:~/tt/Documents$ cp ../fall .
lxc@Lxc:~/tt/Documents$ ls
fall  test_one  test_two
lxc@Lxc:~/tt/Documents$
```

`-R` 选项可以在单个命令中递归地复制整个目录的内容。

```
lxc@Lxc:~/tt$ cp -R Documents/ NewDocuments/
lxc@Lxc:~/tt$ ls -F
Documents/  fall  fell  fill  full  my_file  my_scrapt  my_script  my_scrypt
NewDocuments/  test_one  test_two
lxc@Lxc:~/tt$ ls -lF NewDocuments/
总用量 0
-rw-rw-r-- 1 lxc lxc 0 12月  2 14:51 fall
-rw-rw-r-- 1 lxc lxc 0 12月  2 14:51 test_one
-rw-rw-r-- 1 lxc lxc 0 12月  2 14:51 test_two
```

在执行 `cp -R` 命令之前，*NewDocuments* 目录并不存在。它是随着 `cp -R` 命令被创建的，整个 *Documents* 目录的内容都被复制到其中。注意，新的 *NewDocuments* 目录中的所有文件都有对应的新日期。

也可以在 `cp` 命令中使用通配符。

```
lxc@Lxc:~/tt$ ls
Documents  fall  fell  fill  full  my_file  my_scrapt  my_script  my_scrypt
NewDocuments  test_one  test_two
lxc@Lxc:~/tt$ cp my* NewDocuments/
lxc@Lxc:~/tt$ ls -lF NewDocuments/
总用量 0
-rw-rw-r-- 1 lxc lxc 0 12月  2 14:51 fall
-rw-rw-r-- 1 lxc lxc 0 12月  2 14:55 my_file
-rw-rw-r-- 1 lxc lxc 0 12月  2 14:55 my_scrapt
-rw-rw-r-- 1 lxc lxc 0 12月  2 14:55 my_script
-rw-rw-r-- 1 lxc lxc 0 12月  2 14:55 my_scrypt
-rw-rw-r-- 1 lxc lxc 0 12月  2 14:51 test_one
-rw-rw-r-- 1 lxc lxc 0 12月  2 14:51 test_two
```

3. 命令行补全

使用制表键补全的技巧在于要给shell提供足够的文件名信息，使其能够将所需文件与其他文件名区分开来。连按两下可以看当前的匹配结果：

```
lxc@Lxc:~/tt$ ls f # 这里连按了两下制表键
fall  fell  fill  full
lxc@Lxc:~/tt$ ls f
```

4. 链接文件

在Linux中有两种类型的文件链接。

- 符号链接
- 硬链接

符号链接（软链接） 是一个实实在在的文件，该文件指向存放在虚拟目录结构中某个地方的另一个文件。要为一个文件创建一个符号链接，原始文件必须存在。然后可以使用 `ln` 命令以及 `-s` 选项来创建符号链接。

```
lxc@Lxc:~/tt$ ls -l test_file
-rw-rw-r-- 1 lxc lxc 68 12月  2 15:08 test_file
lxc@Lxc:~/tt$ ln -s test_file slink_test_file
lxc@Lxc:~/tt$ ls -l *test_file
lrwxrwxrwx 1 lxc lxc  9 12月  2 15:08 slink_test_file -> test_file
-rw-rw-r-- 1 lxc lxc 68 12月  2 15:08 test_file
```

在这个例子中，注意符号链接文件名 *slink_test_file* 位于 `ln` 命令的第二个参数。长列表（`ls -l`）中显示的符号文件名后的->符号表示该文件是链接到文件 *test_file* 的一个符号链接。
另外，还要注意符号链接文件与数据文件的大小。符号链接文件 *slink_test_file* 只有9个字节，而 *test_file* 有68个字节。这是因为 *slink_test_file* 仅仅是指向 *test_file* 而已，它们的内容并不相同。
另一种证明链接文件是一个独立文件的方法是查看inode编号。文件或目录的inode编号是内核分配给文件系统中每一个对象的唯一标识。要查看文件或目录的inode编号，可以使用 `ls` 命令的 `-i` 选项。

```
lxc@Lxc:~/tt$ ls -li *test_file
1713477 lrwxrwxrwx 1 lxc lxc  9 12月  2 15:08 slink_test_file -> test_file
1718916 -rw-rw-r-- 1 lxc lxc 68 12月  2 15:08 test_file
```

两者的inode编号不同，所以说两者是不同的文件。

修改已存在的软连接，可以使用 `ln` 命令的 `-f` 选项。如果软链接文件已经存在的情况下，`-f` 选项会移除该软链接文件，创建新的同名文件。或者使用 `-i` 选项，该选项会询问是否覆盖已存在文件。
修改 *slink_test_file* 文件，使其指向 *test_file2* 文件。

```
lxc@Lxc:~/tt$ ln -sf test_file2 slink_test_file
lxc@Lxc:~/tt$ ls -li
总用量 8
1719135 lrwxrwxrwx 1 lxc lxc 10 12月  2 15:31 slink_test_file -> test_file2
1718916 -rw-rw-r-- 1 lxc lxc 68 12月  2 15:08 test_file
1719212 -rw-rw-r-- 1 lxc lxc 44 12月  2 15:20 test_file2
#
lxc@Lxc:~/tt$ ln -si test_file slink_test_file
ln: 是否替换'slink_test_file'？ y
lxc@Lxc:~/tt$ ls -li
总用量 8
1713477 lrwxrwxrwx 1 lxc lxc  9 12月  2 15:33 slink_test_file -> test_file
1718916 -rw-rw-r-- 1 lxc lxc 68 12月  2 15:08 test_file
1719212 -rw-rw-r-- 1 lxc lxc 44 12月  2 15:20 test_file2
```

软链接也可以链接到目录。为此，要使用 `ln` 命令的 `-n` 选项。`-n` 选项会将指向目录的软链接视为普通文件。

```
lxc@Lxc:~/tt$ ls -liF
总用量 12
2117992 drwxrwxr-x 2 lxc lxc 4096 12月  2 15:36 Documents/
1713477 lrwxrwxrwx 1 lxc lxc    9 12月  2 15:33 slink_test_file -> test_file
1718916 -rw-rw-r-- 1 lxc lxc   68 12月  2 15:08 test_file
1719212 -rw-rw-r-- 1 lxc lxc   44 12月  2 15:20 test_file2
lxc@Lxc:~/tt$ ln -snf Documents/ slink_test_file
lxc@Lxc:~/tt$ ls -li
总用量 12
2117992 drwxrwxr-x 2 lxc lxc 4096 12月  2 15:36 Documents
1719135 lrwxrwxrwx 1 lxc lxc   10 12月  2 15:37 slink_test_file -> Documents/
1718916 -rw-rw-r-- 1 lxc lxc   68 12月  2 15:08 test_file
1719212 -rw-rw-r-- 1 lxc lxc   44 12月  2 15:20 test_file2
```

硬链接 创建的是一个独立的虚拟文件，其中包含了原始文件的信息以及位置。但两者就根本而言是同一个文件。要想创建硬链接，原始文件也必须事先存在，只不过这次使用 `ln` 命令时，不需要再加入额外的选项了。

```
lxc@Lxc:~/tt$ ls -li
总用量 12
1718916 -rw-rw-r-- 1 lxc lxc   68 12月  2 15:08 test_file
lxc@Lxc:~/tt$ ln test_file hlink_test_file
lxc@Lxc:~/tt$ ls -li
总用量 16
1718916 -rw-rw-r-- 2 lxc lxc   68 12月  2 15:08 hlink_test_file
1718916 -rw-rw-r-- 2 lxc lxc   68 12月  2 15:08 test_file
```

注意，以硬链接相连的文件共享同一个inode编号。这是因为两者其实就是同一个文件。另外，两者的文件的大小也一样。

> **注意：** 只能对处于同一存储设备的文件创建硬链接。要想在位于不同存储设备的文件之间创建链接，只能使用符号链接（也就是说硬链接不能跨文件系统，软链接可以跨文件系统）。

5. 文件重命名

在Linux中，重命名文件称为 ***移动（moving）***。`mv` 命令可以将文件或目录从一个位置移动到另一个位置同时/或是重命名。

重命名文件：

```
lxc@Lxc:~/tt$ ls -li
总用量 16
2117992 drwxrwxr-x 2 lxc lxc 4096 12月  2 15:38 Documents
1718916 -rw-rw-r-- 2 lxc lxc   68 12月  2 15:08 hlink_test_file
1719135 lrwxrwxrwx 1 lxc lxc   10 12月  2 15:44 slink_test_file -> Documents/
1718916 -rw-rw-r-- 2 lxc lxc   68 12月  2 15:08 test_file
1719212 -rw-rw-r-- 1 lxc lxc   44 12月  2 15:20 test_file2
lxc@Lxc:~/tt$ mv test_file test_file0
lxc@Lxc:~/tt$ ls -li
总用量 16
2117992 drwxrwxr-x 2 lxc lxc 4096 12月  2 15:38 Documents
1718916 -rw-rw-r-- 2 lxc lxc   68 12月  2 15:08 hlink_test_file
1719135 lrwxrwxrwx 1 lxc lxc   10 12月  2 15:44 slink_test_file -> Documents/
1718916 -rw-rw-r-- 2 lxc lxc   68 12月  2 15:08 test_file0
1719212 -rw-rw-r-- 1 lxc lxc   44 12月  2 15:20 test_file2
```

test_file 文件名改为了 *test_file0*。注意，inode编号和时间戳保持不变。这是因为 `mv` 只影响文件名。

也可用 `mv` 来移动文件的位置，或者在移动文件位置的同时重命名：

```
lxc@Lxc:~$ mkdir tt2
lxc@Lxc:~$ mv tt/test_file0 tt2/
lxc@Lxc:~$ ls -li tt2/
总用量 4
1718916 -rw-rw-r-- 2 lxc lxc 68 12月  2 15:08 test_file0
lxc@Lxc:~$ mv tt/test_file2 tt2/test_file2_
lxc@Lxc:~$ ls -li tt2/
总用量 8
1718916 -rw-rw-r-- 2 lxc lxc 68 12月  2 15:08 test_file0
1719212 -rw-rw-r-- 1 lxc lxc 44 12月  2 15:20 test_file2_
lxc@Lxc:~$ ls -li tt/
总用量 8
```

```
2117992 drwxrwxr-x 2 lxc lxc 4096 12月  2 15:38 Documents
1718916 -rw-rw-r-- 2 lxc lxc   68 12月  2 15:08 hlink_test_file
1719135 lrwxrwxrwx 1 lxc lxc   10 12月  2 15:44 slink_test_file -> Documents/
```

注意，同样的只是文件的位置和名称发生了改变。时间戳与inode编号均未发生改变。

移动目录也是同理：

```
lxc@Lxc:~$ ls -ldi tt2/
2118044 drwxrwxr-x 2 lxc lxc 4096 12月  2 16:02 tt2/
lxc@Lxc:~$ mv tt2/ tt/
lxc@Lxc:~$ ls -liF tt/
总用量 12
2117992 drwxrwxr-x 2 lxc lxc 4096 12月  2 15:38 Documents/
1718916 -rw-rw-r-- 2 lxc lxc   68 12月  2 15:08 hlink_test_file
1719135 lrwxrwxrwx 1 lxc lxc   10 12月  2 15:44 slink_test_file ->
Documents//
2118044 drwxrwxr-x 2 lxc lxc 4096 12月  2 16:02 tt2/
# 在移动的同时重命名目录
lxc@Lxc:~$ mv tt/tt2/ ./tt0
lxc@Lxc:~$ ls -lid tt0/
2118044 drwxrwxr-x 2 lxc lxc 4096 12月  2 16:02 tt0/
```

> **提示：** 和 `cp` 命令、`ln` 命令一样，你也可以在 `mv` 命令中加入 `-i` 选项，这样在 `mv` 命令试图覆盖已存在的文件或目录时会发出询问。

6. 删除文件

在Linux中，删除（deleting）叫作 **移除（removing）**。bash shell中用于移除文件的命令是 `rm`。

`rm` 命令非常简单：

```
lxc@Lxc:~/tt$ ls
fall  fell  fill  full
lxc@Lxc:~/tt$ rm -i fall
rm: 是否删除普通空文件 'fall'？ y
lxc@Lxc:~/tt$ ls
fell  fill  full
```

同样，`-i` 选项会询问你是否删除该文件。shell没有回收站或垃圾箱这样的东西，文件一旦被删除，就再也找不回来了。
也可以使用通配符删除一组文件。

```
lxc@Lxc:~/tt$ rm -i f?ll
rm: 是否删除普通空文件 'fell'？ y
rm: 是否删除普通空文件 'fill'？ y
rm: 是否删除普通空文件 'full'？ y
lxc@Lxc:~/tt$ ls
lxc@Lxc:~/tt$
```

如果你要删除很多文件，又不想被命令提示符干扰，可以使用 `-f` 选项来强制删除。

7. 管理目录

在Linux中，有些命令（比如 `cp` 命令）对文件和目录都有效，有些命令则只对目录有效。这节讲讲目录的增删。

1. 创建目录

不多bb了。

```
lxc@Lxc:~/tt$ mkdir New_dir
lxc@Lxc:~/tt$ ls -lid New_dir/
2117812 drwxrwxr-x 2 lxc lxc 4096 12月  2 16:25 New_dir/
lxc@Lxc:~/tt$ mkdir -p New_dir2/New_dir3
lxc@Lxc:~/tt$ ls -RlFi
.:
总用量 8
2117812 drwxrwxr-x 2 lxc lxc 4096 12月  2 16:25 New_dir/
2117823 drwxrwxr-x 3 lxc lxc 4096 12月  2 16:26 New_dir2/

./New_dir:
总用量 0

./New_dir2:
总用量 4
2117825 drwxrwxr-x 2 lxc lxc 4096 12月  2 16:26 New_dir3/
```

2. 删除目录

也不想bb了。

```
lxc@Lxc:~/tt$ ls
New_dir  New_dir2
lxc@Lxc:~/tt$ rmdir New_dir # 注，New_dir 目录为空，所以可使用该命令直接删除
lxc@Lxc:~/tt$ ls
New_dir2
lxc@Lxc:~/tt$ rmdir New_dir2/
rmdir: 删除 'New_dir2/' 失败: 目录非空
lxc@Lxc:~/tt$ rm -r New_dir2/
lxc@Lxc:~/tt$ ls
lxc@Lxc:~/tt$
```

对于 `rm` 命令 `-R` 和 `-r` 选项的效果是一样，都可以递归地删除目录中的文件。
粗暴点的话，就使用 `rm -rf` 就行，不过要小心点。

8. 查看文件内容

1. 查看文件类型

`file` 命令是一个方便的小工具，能够探测文件的内部并判断文件类型。

```
lxc@Lxc:~$ file .bashrc
.bashrc: UTF-8 Unicode text
```

`file` 命令不仅能够确定文件包含的是文本信息，还能确定该文件的字符编码是UTF-8。

```
lxc@Lxc:~$ file tt0/
tt0/: directory
lxc@Lxc:~/tt$ file slink_test_file
slink_test_file: symbolic link to test_file
lxc@Lxc:~/scripts/ch13-更多的结构化命令$ file test10.sh
test10.sh: Bourne-Again shell script, ASCII text executable
lxc@Lxc:~/scripts/ch13-更多的结构化命令$ file /usr/bin/ls
/usr/bin/ls: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,.....省略
```

2. 查看整个文件

Linux有3个命令来完成这个任务。

1. `cat` 命令

```
lxc@Lxc:~/tt$ cat test_file
This is the first line.
This is the second line.
This is the third line.

This is the last line.
```

`-n` 选项会给所有行加上行号。

```
lxc@Lxc:~/tt$ cat -n test_file
     1  This is the first line.
     2  This is the second line.
     3  This is the third line.
     4
     5  This is the last line.
```

`-b` 选项会给有文本的行加上行号。

```
lxc@Lxc:~/tt$ cat -b test_file
     1  This is the first line.
     2  This is the second line.
     3  This is the third line.

     4  This is the last line.
```

2. `more` 命令

`more` 命令是一个分页工具，只支持文本文件中基本的移动。如果想要更多的高级特性，可以使用 `less` 命令。

3. `less` 命令

`less` 命令为 `more` 命令的升级版。提供了多个非常实用的特性，能够实现在文本文件中前后翻动，还有一些高级搜索功能。
`less` 命令还可以完成在整个文件的读取之前显示文件的内容。`cat` 命令和 `more` 命令则无法做到这一点。
`less` 命令通常为手册页提供分页服务。你对 `less` 了解的越多，阅读各种命令手册页的时候就越得心应手。更多选项请查看 `less` 命令的手册页。

3. 查看部分文件

1. `tail` 命令

默认情况下，`tail` 命令会显示文件的末尾10行内容。

`-n` 选项可以修改所显示的行数。例如 `-n 2` 就是只显示最后2行。

```
lxc@Lxc:~$ cat /etc/passwd | wc -l
57
lxc@Lxc:~$ tail -n 2 /etc/passwd
_rpc:x:134:65534::/run/rpcbind:/usr/sbin/nologin
nobody:x:65534:65534::/nonexistent:/usr/sbin/nologin
```

`-n digit` 在 *digit* 前可以加一个加号，表示从第几行直到末尾。例如 `-n +50` 是从第50行（included）到末尾。

```
lxc@Lxc:~$ tail -n +50 /etc/passwd
mysql:x:130:137:MySQL Server,,,:/nonexistent:/bin/false
consul:x:998:997::/home/consul:/bin/false
kong:x:1001:1001:Kong default user:/home/kong:/bin/sh
postgres:x:131:138:PostgreSQL administrator,,,:/var/lib/postgresql:/bin/bash
nginx:x:132:140:nginx user,,,:/nonexistent:/bin/false
sshd:x:133:65534::/run/sshd:/usr/sbin/nologin
_rpc:x:134:65534::/run/rpcbind:/usr/sbin/nologin
nobody:x:65534:65534::/nonexistent:/usr/sbin/nologin
```

`tail` 命令有一个实用的特性：`-f` 选项，该选项允许你在其他进程使用此文件时查看文件的内容，`tail` 命令会保持活动状态并持续显示添加到文件中的内容。这是实时检测系统日志的绝佳方式。

来个例子：

该示例的脚本文件参考test1.sh

```
lxc@Lxc:~/scripts/ch03-bash shell基础命令$ . test1.sh
2023-12-02 17:39:07
2023-12-02 17:39:12
2023-12-02 17:39:17
......省略后续每隔5秒增添的输出。
# 此时，打开另一个终端，执行如下命令，可以看到输出会不断的跟进。
lxc@Lxc:~/scripts/ch03-bash shell基础命令$ tail -f test.log
2023-12-02 17:39:12
2023-12-02 17:39:17
2023-12-02 17:39:22
2023-12-02 17:39:27
```

```
2023-12-02 17:39:32
2023-12-02 17:39:37
2023-12-02 17:39:42
2023-12-02 17:39:47
2023-12-02 17:39:52
2023-12-02 17:39:57
......省略
```

`-F` 选项相对于 `-f` 选项增加了重试的机制。 会有这么一种情况，在我们这个脚本中，输出被保存到 *test.log* 文件中。如果我们删除或者重命名了 *test.log* 文件，那么 `-f` 选项就不会继续跟进脚本的输出（即便5秒后，该脚本再次创建了同名的文件）。而 `-F` 选项会重新跟进新出现的同名文件，当原文件丢失时，`-F` 选项还会报告丢失，当同名文件再次出现时，它也会报告并跟进。

```
lxc@Lxc:~/scripts/ch03-bash shell基础命令$ . test1.sh
2023-12-02 17:54:33
2023-12-02 17:54:38
2023-12-02 17:54:43
2023-12-02 17:54:48
2023-12-02 17:54:53
......省略输出
# 在第二个终端上进行跟进
lxc@Lxc:~/scripts/ch03-bash shell基础命令$ tail -F test.log
2023-12-02 17:40:58
2023-12-02 17:41:03
2023-12-02 17:54:33
2023-12-02 17:54:38
2023-12-02 17:54:43
2023-12-02 17:54:48
2023-12-02 17:54:53
2023-12-02 17:54:58
2023-12-02 17:55:03
2023-12-02 17:55:08
2023-12-02 17:55:13
2023-12-02 17:55:18
2023-12-02 17:55:23
2023-12-02 17:55:28
2023-12-02 17:55:33
2023-12-02 17:55:38
2023-12-02 17:55:43
# 在第三个终端上进行重命名
lxc@Lxc:~/scripts/ch03-bash shell基础命令$ mv test.log test1.log
# 查看第二个终端的输出:
tail: 'test.log' 已不可访问: 没有那个文件或目录
tail: 'test.log' 已被建立；正在跟随新文件的末尾
2023-12-02 17:55:48
2023-12-02 17:55:53
2023-12-02 17:55:58
2023-12-02 17:56:03
```

如你所见，`-F` 选项符合我们预期。
`-f` 选项不会进行重试：

```
lxc@Lxc:~/scripts/ch03-bash shell基础命令$ . test1.sh
2023-12-02 17:58:50
```

```
2023-12-02 17:58:55
2023-12-02 17:59:00
2023-12-02 17:59:05
2023-12-02 17:59:10
2023-12-02 17:59:15
2023-12-02 17:59:20
# 在第二个终端进行跟进
lxc@Lxc:~/scripts/ch03-bash shell基础命令$ tail -f test.log
2023-12-02 17:55:48
2023-12-02 17:55:53
2023-12-02 17:55:58
2023-12-02 17:56:03
2023-12-02 17:58:50
2023-12-02 17:58:55
2023-12-02 17:59:00
2023-12-02 17:59:05
# 在第三个终端进行重命名
lxc@Lxc:~/scripts/ch03-bash shell基础命令$ mv test.log test2.log
# 查看第二个终端:
没有输出，只有一个闪烁的光标。
```

2. `head` 命令

默认情况下，`head` 命令会显示文件的前10行文本。`head` 命令没什么好讲的其实，就跟 `more` 相对于 `less` 一样。

`-n` 选项可以修改所显示的行数。`-n 2` 显示前两行。

```
lxc@Lxc:~$ head -n 2 /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
```

`-n digit`，*digit* 参数前可以加一个负号（-），表示从开头显示到倒数第几行。`head -n -50` 表示从开头显示到倒数第50行（included）。

```
lxc@Lxc:~$ head -n -50 /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
```

ch04 更多的bash shell命令

本章会讲解Linux系统管理命令，演示如何通过命令行命令来探查Linux系统的内部信息，然后会讲解一些可用于处理系统数据文件的命令。

1. 检测程序

1. 探查进程

`ps` 命令默认只显示运行在当前终端中属于当前用户的那些进程。在这个例子中，只有bash shell在运行（记住，shell只是运行在系统中的另一个程序而已），当然 `ps` 命令本身也在运行（这是因为 `ps` 是外部命令，参见 第5章）。

```
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ ps
    PID TTY          TIME CMD
   8793 pts/2    00:00:00 bash
  12458 pts/2    00:00:00 ps
```

`ps` 命令的基本输出显示了程序的ID（process ID，PID）、进程运行在哪个终端（TTY）及其占用的CPU时间。

> **注意：** `ps` 命令令人头疼的地方（也正是它如此复杂的原因）在于它曾经有两个版本。每个版本都有自己的一套命令行选项，控制着显示哪些信息以及如何显示。最近Linux开发人员已经将这两种 `ps` 命令格式合并到单个 `ps` 命令中（当然，同时也加入了他们自己的风格）。

Linux系统中使用的 `GNU ps` 命令支持以下3种类型的命令行选项：

- Unix 风格选项，选项前加单连字符
- BSD 风格选项，选项前不加连字符
- GNU 长选项。选项前加双连字符

1. Unix 风格选项

Unix风格选项源自贝尔实验室开发的 AT&T Unix 系统中的 `ps` 命令。这些选项如下表所示。

(我比较懒。。。)

查看系统中运行的所有进程，常用 `-elf` 选项。

`-e` 选项指定系统中运行的所有进程。`-f` 选项则扩充输出内容以显示一些有用的信息列。

`-l` 选项产生长格式的输出。

2. BSD风格选项

伯克利软件发行版（Berkeley Software Distribution，BSD）是加州大学伯克利分校开发的一个Unix版本。BSD与AT&T Unix系统有许多细微的差别，由此引发了多年来的诸多Unix纷争。BSD版的 `ps` 命令选项如下表所示。

如你所见，Unix和BSD风格的选项有很多的重叠之处。从一种风格的选项中得到的信息基本上也能从另一种风格中获取。
下面是使用 `l` 选项的输出。

```
lxc@Lxc:~$ ps l
F   UID     PID    PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
4  1000    2252    2152  20   0 164884  6188 do_sys Ssl+ tty2       0:00 /usr/lib/gdm3/gdm-wayland-session env GNOME_SHE
0  1000    2257    2252  20   0 191484 13920 do_sys Sl+  tty2       0:00 /usr/libexec/gnome-session-binary --systemd --s
0  1000    3776    3771  20   0  17944  9000 do_wai Ss   pts/0      0:00 bash
0  1000    4167    4151  20   0  17980  9232 do_sel Ss+  pts/1      0:00 /usr/bin/bash --init-file /usr/share/code/resou
0  1000    4676    4151  20   0  17980  9048 do_sel Ss+  pts/2      0:00 /usr/bin/bash --init-file /usr/share/code/resou
4  1000    5468    3776  20   0  14508  3196 -      R+   pts/0      0:00 ps l
```

下面列出每一列的含义。

- VSZ： 进程占用的虚拟内存的大小（以KB为单位）。
- RSS： 进程在未被交换出时占用的物理内存大小。
- STAT： 代表当前进程状态的多字符状态码。

很多系统管理员喜欢BSD风格的 `l` 选项，因为能输出更详细的进程状态码（STAT列）。多字符状态码能比Unix风格输出的单字符状态码更清楚的表明当前进程的状态。
第一个字符采用了和Unix风格输出的 S 输出列相同的值，表明进程是在休眠、运行还是等待。第二个字符进一步说明了进程的状态。

从这个例子中可以看出，bash 命令处于休眠状态，同时它也是一个控制进程（会话中的主进程），而 `ps` 命令则运行在系统前台。

3. GNU 长选项

GNU开发人员在经过改进的新 `ps` 命令中加入了另外一些选项，其中一些GNU长选项复制了现有的Unix或BSD风格选项的效果，而另外一些则提供了新功能。下表列出了可用的GNU长选项。

可以混用GNU长选项和Unix或BSD风格的选项来定制输出。作为一个GNU长选项， `--forest` 选项着实讨人喜欢。该选项能够使用ASCII字符来绘制图表以显示进程的层级关系。

```
lxc@Lxc:~$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
lxc         3776    3771  0 10:59 pts/0    00:00:00 bash
lxc         6886    3776  0 11:28 pts/0    00:00:00 ps -f
lxc@Lxc:~$ bash
lxc@Lxc:~$ bash
lxc@Lxc:~$ ps --forest
    PID TTY          TIME CMD
   3776 pts/0    00:00:00 bash
   6887 pts/0    00:00:00  \_ bash
   7070 pts/0    00:00:00      \_ bash
   7253 pts/0    00:00:00          \_ ps
```

这种格式可以轻易的跟踪子进程和父进程。

2. 实时检测进程

`ps` 命令只能显示特定时间点的信息。`top` 命令也可以显示进程信息，但采用的是实时的方式。

输出的第一部分显示的是系统的概况：第一行显示了当前时间、系统的运行时长、登录的用户数以及系统的平均负载。平均负载有3个值，分别是最近1分钟、最近5分钟和最近15分钟的平均负载。值越大说明系统的负载越高。由于进程短期的突发性活动，出现最近1分钟的高负载值也很常见。但如果最近15分钟内的平均负载都很高，就说明系统可能有问题了。

> **注意：** Linux系统管理员的难点在于定义究竟到什么程度才算是高负载。这个值取决于系统的硬件配置以及系统中通常运行的程序。某个系统的高负载可能对于其他系统来说就是普通水平。最好都做法是注意在正常情况下系统的负载情况，这样更容易判断系统何时负载不足。

第二行显示了进程（`top` 称其为 task）概况：多少进程处于运行、休眠、停止以及僵化状态（僵化状态是指进程已经结束，但其父进程没有响应）。
第三行显示了CPU概况。`top` 会根据进程的属主（用户还是系统）和进程的状态（运行、空闲或等待）将CPU利用率分成几类输出：

- us： 用户空间占用CPU百分比
- sy： 内核空间占用CPU百分比
- ni： 用户空间内改变过优先级的进程占用CPU百分比
- id： 空闲CPU时间百分比
- wa： 等待输入输出的CPU时间百分比
- hi： 硬件CPU中断占用百分比
- si： 软中断占用百分比
- st： 虚拟机占用百分比

第四、五行详细说明了系统内存的状态。前一行显示了系统的物理内存状态：总共有多少内存、当前用了多少、还有多少空闲。后一行显示了系统交换空间（如果分配了的话）的状态。
最后一部分显示了当前处于运行状态的进程的详细列表，有些列跟 `ps` 命令的输出类似。

- PID： 进程的PID
- USER： 进程属主的用户名
- PR: 进程的优先级
- NI： 进程的谦让度
- VIRT： 进程占用的虚拟内存总量
- RES： 进程占用的物理内存总量
- SHR： 进程和其他进程共享的内存总量
- S： 进程的状态（D代表可中断的休眠状态，R代表在运行状态，S代表休眠状态，T代表被跟踪或停止状态，Z代表僵化）
- %CPU： 进程使用的CPU时间比例
- %MEM： 进程使用的可用物理内存占比
- TIME+： 自进程启动到目前为止的CPU时间总量
- COMMAND： 进程对应的命令行名称，也就是启动的程序名。

在默认情况下，`top` 命令在启动时会按照%CPU的值来对进程进行排序，你可以在 `top` 命令运行时使用多种交互式命令来重新排序。每个交互式命令都是单字符，在 `top` 命令运行时键入可以改变 `top` 的行为。键入 `f` 允许你选择用于对输出进行排序的字段，键入 `d` 允许你修改轮询间隔（polling interval），键入 `q` 可以退出 `top`。

3. 结束进程

Linux沿用了Unix的进程间通信方法。在Linux中，进程之间通过 **信号** 来通信。通信的信号是一个预定义好的一个消息，进程能够识别该消息并决定忽略还是做出反应。进程如何处理信号是由开发人员通过编程决定的。常见的信号如下表所示。

信号	名称	描述
1	HUP	挂起
2	INT	中断
3	QUIT	结束运行
9	KILL	无条件终止
11	SEGV	断错误
15	TERM	尽可能终止
17	STOP	无条件停止运行，但不终止
18	TSTP	停止或暂停，但继续在后台运行
19	CONT	在STOP或TSTP之后恢复执行

在Linux中有两个命令可以向运行中的进程发出进程信号：`kill` 和 `pkill`。

1. `kill` 命令

`kill` 命令可以通过PID向进程发出信号。在默认情况下，`kill` 命令会向命令行中列出的所有PID发送 `TERM` 信号。你只能使用进程的PID而不能使用其对应的程序名，这使得 `kill` 命令有时并不好用。要发送进程信号，必须是进程的属主或root用户。

```
$ kill 3940
bash: kill: (3940) - Operation not permitted
```

`TERM` 信号会告诉进程终止运行。但不服从管教的进程通常会忽略这个请求。如果要强制终止，则 `-s` 选项支持指定其他信号（用信号名或信号值）。

```
$ kill -s HUP 3940
$
```

2. `pkill` 命令

`pkill` 命令可以使用程序名代替PID来终止进程。除此之外，`pkill` 命令也允许使用通配符。

```
root@Lxc:/home/lxc# pkill http*
root@Lxc:/home/lxc#
```

该命令将杀死所有名称以 http 起始的进程。

> **警告：** 以root身份使用 `pkill` 命令时要格外小心。命令中的通配符很容易意外地将系统的重要进程终止。这可能导致文件系统损坏。

2. 检测磁盘系统空间

有几个命令行命令可以帮助你管理Linux系统中的存储设备。本节将介绍在日常系统管理中会用到的核心命令。

1. 挂载存储设备

如第3章所述，Linux文件系统会将所有的磁盘都并入单个虚拟目录。在使用新的存储设备之前，需要将其放在虚拟目录中。这项工作称为 **挂载（mounting）**。
在今天图形化桌面环境中，大多数Linux发行版能自动挂载特定类型的 **可移动存储设备**。所谓可移动存储设备（显然）指的是那种可以从PC中轻易移除的媒介，比如DVD和U盘。

1. `mount` 命令

用于挂载存储设备的命令叫作 `mount`。在默认情况下，`mount` 命令会输出当前系统已挂载的设备列表。但是，除了标准存储设备，较新版本的内核还会挂载大量用作管理目的的虚拟文件系统。这使得 `mount` 命令的默认输出非常杂乱，让人摸不着头脑。如果知道设备分区使用的文件系统类型，可以像下面这样过滤输出。

```
lxc@Lxc:~$ mount -t ext4
/dev/nvme0n1p8 on / type ext4 (rw,relatime,errors=remount-ro)
```

`mount` 命令提供了4部分信息。

- 设备文件名
- 设备在虚拟目录中的挂载点
- 文件系统类型
- 已挂载设备的访问状态

要手动在虚拟目录中挂载设备，需要以root用户身份登录，或是以root用户身份运行。下面是手动挂载设备的基本命令：

```
mount -t type device directory
```

其中，*type* 参数指定了磁盘格式化所使用的文件系统类型。Linux可以识别多种文件系统类型。如果与Windows PC 共用移动存储设备，那么需要使用下列文件系统类型。

- vfat： Windows FAT32文件系统，支持长文件名。
- ntfs： Windows NT 及后续操作系统中广泛使用的高级文件系统。
- exfat： 专门为可移动存储设备优化的Windows文件系统。
- iso9660： 标准CD—ROM和DVD文件系统。

大多数U盘会使用vfat文件系统格式化。如果需要挂载数据CD或DVD，则必须使用iso-9660文件系统类型。
后面两个参数指定了该存储设备文件位置以及挂在点在虚拟目录中的位置。例如，手动将U盘/dev/sdb1挂载到/media/disk，可以使用以下命令：

```
mount -t vfat /dev/sdb1 /media/disk
```

一旦存储设备被挂载到虚拟目录，root用户就拥有了对该设备的所有访问权限，而其他用户的访问则会被限制。可以通过目录权限（参见 第7章）指定用户对设备的访问权限。
如果需要使用 `mount` 命令的一些高级特性，可以参见下表：

2. `umount` 命令

移除可移动设备时，不能直接将设备拔下，应该先卸载。

> **提示：** Linux不允许直接弹出已挂载的CD或DVD。如果从光驱中移除CD或DVD时遇到麻烦，那么最大的可能就是它还在虚拟目录中挂载着。应该先卸载，然后再尝试弹出。
> 卸载设备的命令是 `umount`。该命令的格式非常简单。

```
umount [directory | device]
```

`umount` 命令支持通过设备文件或者挂载点来指定要卸载的设备。如果有任何程序正在使用设备上的文件，则系统不允许卸载该设备。

在本例中，因为命令行提示符仍然位于已挂载设备的文件系统中，所以 `umount` 命令无法卸载该镜像文件。一旦命令行提示符移出其镜像文件系统，`umount` 命令就能成功卸载镜像文件了。

3. 使用 `df` 命令

`df` 命令可以方便的查看所有已挂载磁盘的使用情况。

```
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ df -t ext4 -t vfat
文件系统           1K-块      已用      可用 已用% 挂载点
/dev/nvme0n1p8 107014756 62707580 38824888   62% /
/dev/nvme0n1p1     98304    68636    29668   70% /boot/efi
```

`df` 命令会逐个显示已挂载的文件系统。与 `mount` 命令类似，`df` 命令会输出内核挂载的所有虚拟文件系统，因此可以使用 `-t` 选项来指定文件系统类型，进而过滤输出结果。该命令的输出如下。

- 设备文件位置
- 包含多少以1024字节为单位的块
- 使用了多少以1024字节为单位的块
- 还有多少以1024字节为单位的块可用
- 已用空间所占的百分比
- 设备挂载点

`df` 命令的大部分选项我们不会用到。常用选项之一是 `-h`，该选项会以人类易读（human-readable）的形式显示磁盘空间，通常用M来替代兆字节，用G来替代吉字节。`-T` 选项会打印文件系统类型。

```
lxc@Lxc:~$ df -hT
文件系统        类型      容量  已用  可用 已用% 挂载点
udev            devtmpfs  7.5G     0  7.5G    0% /dev
tmpfs           tmpfs     1.5G  2.2M  1.5G    1% /run
/dev/nvme0n1p8 ext4      103G   60G   38G   62% /
......
```

> **注意：** 记住，Linux系统后台一直有进程在处理文件。`df` 命令的输出值反映的是Linux系统认为的当前值。正在运行的进程有可能创建或删除了某个文件，但尚未释放该文件。这个值是不会被计算进空闲空间的。

3. 使用 `du` 命令

`du` 命令可以显示某个特定目录（默认情况下是当前目录）的磁盘使用情况。这有助于你快速判断系统中是否存在磁盘占用大户。
在默认情况下，`du` 命令会显示当前目录下所有的文件、目录和子目录的磁盘使用情况，并以磁盘块为单位来表明每个文件或目录占用了多大存储空间。

```
lxc@Lxc:~$ du
32  ./.presage
......省略很长很长的输出
```

每行最左侧的数字是每个文件或目录所占用的磁盘块数。注意，这个列表是从目录层级的最底部开始，然后沿着其中包含的文件和子目录逐级向上的。
下面这些选项可以使 `du` 命令的输出更易读。

- -c： 显示所有已列出文件的总大小。
- -h： 以人类易读的格式输出大小，分别用K表示千字节、M表示兆字节、G表示吉字节。
- -s： 输出每个参数的汇总信息。

3. 处理数据文件

希望这些命令可以使我们的生活更轻松吧。

1. 数据排序

在默认情况下，`sort` 命令会依据会话所指定的默认语言的排序规则来对文本文件中的数据进行排序。

```
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ cat file1
one
two
three
four
five
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ sort file1
five
four
one
three
two
```

默认情况下，`sort` 命令会将数字视为字符并执行标准的字符排序。这种结果可能不是你想要的，可以使用 `-n` 选项来告诉 `sort` 将数字按值排序：

```
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ cat file2
1
2
100
45
3
10
145
75
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ sort file2
1
10
100
145
2
3
45
75
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ sort -n file2
1
2
3
10
45
75
100
145
```

下表列出了 `sort` 命令的其他一些方便的选项。

短选项	长选项	描述
-b	--ignore-leading-blanks	排序时忽略起始的空白字符
-C	--check=quiet	不排序，如果数据无序也不要报告
-c	--check	不排序，但检查输入数据是不是已排序；无序的话就报告
-d	--directionary-order	仅考虑空白字符和字母数字字符，不考虑特殊字符
-f	--ignore-case	大写字母默认先出现，该选项会忽略大小写
-g	--general-numeric-sort	按通用数值来排序（跟-n不同，把值当浮点数来排序，支持科学计数法表示的值
-i	--ignore-nonprinting	在排序时忽略不可打印字符
-k	--key=*POS1*[,*POS2*]	排序从POS1位置开始，到POS2位置结束（如果指定了POS2的话）

短选项	长选项	描述
-M	--month-sort	用三字符月份名按月份排序
-m	--merge	将两个已排序数据文件合并
-n	--numeric-sort	按字符串数值来排序（并不转换为浮点数）
-o	--output=*file*	将排序的结果写到指定的文件中
-R	--random-sort 或 --random-source=*FILE*	根据随机哈希排序 或 指定-R选项用到的随机字节文件
-r	--reverse	逆序排序（升序变成降序）
-S	--buffer-size=*SIZE*	指定使用的内存大小
-s	--stable	禁止last-sort比较，实现稳定排序
-T	--temporary-directory=*DIR*	指定一个位置来存储临时工作文件
-t	--field-separator=*SEP*	指定字段分隔符
-u	--unique	和-c选项合用时，检查严格排序；不和-c选项合用时，相同的行仅输出一次（即去重相当于 sort | uniq）
-z	--zero-terminated	在行尾使用NULL字符替代换行符

例如，要根据用户ID对 /etc/passwd 按数值排序。

```
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ sort -t ':' -k 3 -n /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
......
```

现在数据已经按第三个字段（用户的ID数值）排序妥当了。`-t` 选项指定冒号为字段分隔符，-k 选项指定第3个字段。
`-n` 选项适合于排序数值型输出，比如 `du` 命令的输出。

```
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ du -sh * | sort -hr
140K    top.png
112K    bsd2.png
88K GNU-1.png
84K mount2.png
76K ps2.png
56K ps-l.png
52K ps1.png
44K ps-e.png
40K mount1.png
28K GNU-2.png
28K bsd-s.png
24K bsd1.png
20K README.md
16K umount.png
```

```
4.0K    file3
4.0K    file2
4.0K    file1
```

`-r` 选项降序排序，这样你就能轻易看出目录中哪些文件占用磁盘空间最多。

2. 数据搜索

所谓的Linux三剑客之一：`grep`。另外两个在 第19章。
`grep` 命令的格式如下：

```
grep [options] pattern [file]
```

`grep` 命令会在输入或指定文件中逐行搜索匹配指定模式的文本。该命令的输出是包含了匹配模式的行。

```
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ grep three file1
three
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ grep t file1
two
three
```

如果要进行反向搜索（即搜索不匹配指定模式的行），可以使用 `-v` 选项。

```
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ grep -v t file1
one
four
five
```

如果要显示匹配指定模式的行，可以使用 `-n` 选项。

```
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ grep -n t file1
2:two
3:three
```

如果只想知道有多少行含有匹配的模式，可以使用 `-c` 选项。

```
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ grep -c t file1
2
```

如果要指定多个匹配模式，可以使用 `-e` 选项来逐个指定。

```
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ grep -e t -e f file1
two
three
four
five
```

在默认情况下，`grep` 命令使用基本的Unix风格正则表达式（BRE）（参见 第20章）来匹配模式。

```
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ grep [tf] file1
two
three
four
five
```

In addition, the variant programs egrep, fgrep and rgrep are the same as grep -E, grep -F, and grep -r,
respectively. These variants are deprecated, but are provided for backward compatibility.

- -E, --extended-regexp
  Interpret PATTERNS as extended regular expressions (EREs, see below).
- -F, --fixed-strings
  Interpret PATTERNS as fixed strings, not regular expressions.
- -r, --recursive
  Read all files under each directory, recursively, following symbolic links only if they are on the command line. Note that if no file operand is given, grep searches the working directory. This is equivalent to the -d recurse option.
- -d ACTION, --directories=ACTION
  If an input file is a directory, use ACTION to process it. By default, ACTION is read, i.e., read directories just as if they were ordinary files. If ACTION is skip, silently skip directories. If ACTION is recurse, read all files under each directory, recursively, following symbolic links only if they are on the command line. This is equivalent to the -r option.

3. 数据压缩

下表列出了可用的Linux文件压缩工具。

工具	文件扩展名	描述
bzip2	.bz2	采用Burrows-Wheeler块排序文本压缩算法和霍夫曼编码
compress	.Z	最初的Unix文件压缩工具，已经快要无人使用了
gzip	.gz	GNU压缩工具，用Lempel-Zivwelch编码
xz	.xz	日渐流行的通用压缩工具
zip	.zip	Windows中PKZIP工具的Unix实现

gzip是Linux中最流行的压缩工具。gzip软件包是GNU项目的产物，旨在编写一个能够替代原先Unix中compress工具的免费版本。这个软件包包括以下文件。

- gzip： 用于压缩文件
- gzcat： 用于查看压缩过的文本文件的内容（命令名是zcat）
- gunzip： 用于解压文件

这些工具基本上和bzip2一样：

```
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ gzip file1
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ ls -l file1.gz
-rw-rw-r-- 1 lxc lxc 50 12月  1 15:28 file1.gz
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ zcat file1.gz
one
two
three
four
five
```

gzip会压缩命令行中指定的文件。也可以指定多个文件名或是用通配符来一次性压缩多个文件。

```
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ gzip file*
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ ls file*
file2.gz  file3.gz
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ gunzip file*
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ ls file*
file2  file3
```

4. 数据归档

目前，Unix和Linux中最流行的归档工具是 `tar` 命令。`tar` 命令最开始是用来将文件写到磁带设备上归档的，但它也可以将输出写入文件，这种用法成了在Linux中归档数据的普遍用法。
`tar` 命令的格式如下：

```
tar function [options] object1 object2 ....
```

function 参数定义了 `tar` 命令要执行的操作，如下表所示。

操作	长选项	描述
-A	--concatenate	将一个tar归档文件追加到另一个tar归档文件的末尾
-c	--create	创建新的tar归档文件
-d	--diff 或 --delete	检查归档文件与文件系统的不同之处 或 从tar归档文件中删除文件
-r	--append	追加文件到已有tar归档文件末尾
-t	--list	列出已有tar归档文件的内容
-u	--update	将比tar归档文件中已有的同名文件更新的文件追加到该归档文件
-x	--extract	从tar归档文件中提取文件

每种操作都使用 *option* （选项）来定义对tar归档文件的具体行为，下表列出了常用的选项。

选项	描述
-C *dir*	切换到指定目录
-f *file*	将结果输出到文件（或设备）
-j	将输出传给bzip2命令进行压缩

选项	描述
-J	将输出传给xz命令进行压缩
-p	保留文件的所有权限
-v	在处理文件时显示文件名
-z	将输出传给gzip命令进行压缩
-Z	将输出传给compress命令进行压缩

这些选项经常合并使用。
可以使用下面的命令创建归档文件。

```
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ tar -cvf file.tar file*
file1
file2
file3
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ ls file*
file1  file2  file3  file.tar
```

归档并压缩（使用gzip）。

```
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ tar -cvzf file.tgz file1 file2
file1
file2
```

下面的命令列出（但不提取）归档文件的内容。

```
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ tar -tf file.tar
file1
file2
file3
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ tar -tf file.tgz
file1
file2
```

下面的命令从归档文件中提取内容。

```
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ tar -xvf file.tar
file1
file2
file3
```

下面的命令解压缩并提取已压缩的归档文件的内容。

```
lxc@Lxc:~/scripts/ch04-更多的bash shell命令$ tar -xzvf file.tgz
file1
file2
```

常见的就是这几个了。希望能够使你的生活更轻松。

> **提示：** 在下载开源软件时经常会看到文件名以 .tgz 或 .tar.gz 结尾，这是经gzip压缩过的tar文件。提取这类文件的方法见上面。

ch05 理解shell

本章会带你全面学习shell进程及其在各种情况下的运作方式。我们将探究如何创建子shell及其与父shell之间的关系，了解各种会创建子进程和不会创建子进程的命令。另外，本章还会涉及一些提高命令行效率的shell窍门和技巧。

1. shell的类型

当你登录系统时，系统启动什么样的shell取决于你的个人用户配置。在 /etc/passwd 文件的第7个字段列出了该用户的默认shell程序。只要用户登录某个虚拟控制台终端或是在GUI中启动终端仿真器，默认的shell程序就会启动。

```
lxc@Lxc:~/scripts/ch05$ cat /etc/passwd
lxc:x:1000:1000:Lxc,,,:/home/lxc:/bin/bash
......
```

在现代Linux系统中，bash shell程序通常位于 /usr/bin 目录。不过，在你的Linux系统中，也有可能位于 /bin 目录。`which bash` 命令可以帮助我们找出bash shell的位置：

```
lxc@Lxc:~/scripts/ch05$ which bash
/usr/bin/bash
lxc@Lxc:~/scripts/ch05$ ls -lF /usr/bin/bash
-rwxr-xr-x 1 root root 1183448 4月  18  2022 /usr/bin/bash*
```

`ls` 命令的长列表尾部的 `*` 表明bash文件是一个可执行程序。

> **注意：** 在现代Linux系统中，/bin 目录通常是 /usr/bin 目录的符号链接，/sbin目录通常是/usr/sbin目录的符号链接。这就是为什么用户lxc用户的默认shell程序是/bin/bash，但bash shell程序实际位于/usr/bin目录。

在大多数Linux系统中，/etc/shells 文件列出了各种已安装的shell，这些shell可以作为用户的默认shell。

```
lxc@Lxc:~/scripts/ch05$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/bash
/usr/bin/bash
/bin/rbash
/usr/bin/rbash
/bin/dash
/usr/bin/dash
/bin/zsh
/usr/bin/zsh
```

默认的交互式shell（default interactive shell） 也称 **登录shell（login shell）**，只要用户登录某个虚拟控制台终端或是在GUI中启动终端仿真器，该shell就会启动。
作为 **默认的系统shell（default system shell）**，sh(/bin/bash)用于那些需要在启动时使用的系统shell脚本。

你经常会看到一些发行版使用软连接将默认的系统shell指向bash shell，比如CentOS发行版：

```
lxc@Lxc:~/scripts/ch05$ ls -lF `which sh`
lrwxrwxrwx 1 root root 13 11月  9 21:35 /usr/bin/sh -> /usr/bin/bash*
```

但要注意，在有些发行版。默认的系统shell并不指向bash shell，比如Ubuntu发行版：

```
lxc@Lxc:~/scripts/ch05$ ls -lF `which sh`
lrwxrwxrwx 1 root root 13 11月  9 21:35 /usr/bin/sh -> /usr/bin/dash*
# 本书使用的是Ubuntu发行版，不过更改了这个软连接使其指向了bash shell。
```

并不是非得使用默认的交互式shell，可以启动任意一种已安装的shell，只需输入其名称即可。但屏幕上不会有任何提示或消息表明你当前使用的是哪种shell。`$0` 变量可以助你一臂之力。命令 `echo $0` 会显示当前shell的名称，提供必要的参考。

> **注意：** 使用 `echo $0` 显示当前所用shell的做法仅限在shell命令行中使用。如果在shell脚本中使用，则显示的是该脚本的名称。可以参见第14章。

```
lxc@Lxc:~/scripts/ch05$ echo $0
-bash
lxc@Lxc:~/scripts/ch05$ dash
$ echo $0
dash
```

> **注意：** 在上面的例子中，注意第一个命令的输出：bash之前有一个连字符(-)。这表明该shell是用户的登录shell。

`$` 是dash shell的CLI提示符。输入命令 `exit` 就可以退出dash shell程序（对于bash shell也是如此）：

```
$ exit
lxc@Lxc:~/scripts/ch05$ echo $0
-bash
```

2. shell的父子关系

用户登录某个虚拟控制台终端或在GUI中运行终端仿真器时所启动的默认的交互式shell（登录shell）是一个 **父shell**。到目前为止，都是由父shell提供CLI提示符并等待命令输入。
当你在CLI提示符处输入 `bash` 命令（或是其他shell程序名）时，会创建新的shell程序。这是一个 **子shell**。子shell也拥有CLI提示符，同样会等待命令输入。

```
lxc@Lxc:~/scripts/ch05$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
lxc         3313    3308  0 10:28 pts/0    00:00:00 bash
lxc        10040    3313  0 11:31 pts/0    00:00:00 ps -f
lxc@Lxc:~/scripts/ch05$ bash
lxc@Lxc:~/scripts/ch05$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
lxc         3313    3308  0 10:28 pts/0    00:00:00 bash
lxc        10041    3313  2 11:31 pts/0    00:00:00 bash
lxc        10225   10041  0 11:31 pts/0    00:00:00 ps -f
```

> **注意：** 进程就是正在运行的程序。bash shell是一个程序，当它运行的时候，就成了进程。一个 运行中的shell同样是进程。因此，在说到运行bash shell的时候，你会经常看到 "shell" 和 "进程" 这两个词交换使用。

输入命令 `bash` 之后，就创建了一个子shell（可以通过PPID来确认父子关系）。你可以从显示结果中看到有两个bash shell程序在运行。在生成子shell进程时，只有部分父进程的环境被复制到了子shell环境中。下图展示了这种关系。

子shell既可以从父shell中创建，也可以从另一个子shell中创建：

```
lxc@Lxc:~/scripts/ch05$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
lxc         3313    3308  0 10:28 pts/0    00:00:00 bash
lxc        10993    3313  0 11:39 pts/0    00:00:00 ps -f
lxc@Lxc:~/scripts/ch05$ bash
lxc@Lxc:~/scripts/ch05$ bash
lxc@Lxc:~/scripts/ch05$ bash
lxc@Lxc:~/scripts/ch05$ ps --forest
    PID TTY          TIME CMD
   3313 pts/0    00:00:00 bash
  10994 pts/0    00:00:00  \_ bash
  11177 pts/0    00:00:00      \_ bash
  11360 pts/0    00:00:00          \_ bash
  11543 pts/0    00:00:00              \_ ps
```

下图展示了这种关系：

可以使用 `exit` 命令有条不紊的退出子shell。

```
lxc@Lxc:~/scripts/ch05$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
lxc         3313    3308  0 10:28 pts/0    00:00:00 bash
lxc        10994    3313  0 11:39 pts/0    00:00:00 bash
lxc        11177   10994  0 11:39 pts/0    00:00:00 bash
lxc        11360   11177  0 11:39 pts/0    00:00:00 bash
lxc        11766   11360  0 11:42 pts/0    00:00:00 ps -f
lxc@Lxc:~/scripts/ch05$ exit
exit
lxc@Lxc:~/scripts/ch05$ ps --forest
    PID TTY          TIME CMD
   3313 pts/0    00:00:00 bash
  10994 pts/0    00:00:00  \_ bash
  11177 pts/0    00:00:00      \_ bash
  11768 pts/0    00:00:00          \_ ps
lxc@Lxc:~/scripts/ch05$ exit
exit
lxc@Lxc:~/scripts/ch05$ exit
exit
lxc@Lxc:~/scripts/ch05$ ps --forest
    PID TTY          TIME CMD
   3313 pts/0    00:00:00 bash
```

```
23    11770 pts/0    00:00:00  \_ ps
```

bash shell程序可以使用命令行选项来修改shell的启动方式。下表列出了bash命令的部分可用选项。

选项	描述
-c *string*	从 *string* 中读取命令并进行处理
-i	启动一个能够接收用户输入的交互式shell
-r	启动一个受限shell，将用户限制在默认目录中
-s	从标准输入中读取命令

可以输入 `man bash` 获取关于 `bash` 命令的更多帮助信息，了解更多的命令行选项。`bash --help` 命令也会提供一些额外的帮助。

1. 查看进程列表

可以在单行中指定要一次运行的一系列命令。这可以通过命令列表来实现，只需将命令之间以分号分隔即可：

```
lxc@Lxc:~/scripts/ch05$ pwd; ls *.png; cd /etc; pwd; cd; pwd
/home/lxc/scripts/ch05
parchildshell2.png  parchishell.png
/etc
/home/lxc
```

可以将命令列表放入圆括号内形成 **进程列表**。

```
lxc@Lxc:~/scripts/ch05$ (pwd; ls *.png; cd /etc; pwd; cd; pwd)
/home/lxc/scripts/ch05
parchildshell2.png  parchishell.png
/etc
/home/lxc
```

圆括号的加入使命令列表变成了进程列表，进程列表会生成一个子shell来执行这些命令。

> **注意：** 进程列表是 **命令分组（command grouping）** 的一种。另一种命令分组是将命令放入花括号内，并在命令尾部以分号作结。语法为 `{ command; }`。使用花括号进行命令分组并不会像进程列表那样创建子shell。

要想知道是否生成了子shell，需要使用一个命令输出环境变量的值。这个命令就是 `echo $BASH_SUBSHELL`。如果命令返回0，那么表明没有子shell。如果命令返回1或者更大的数字，则表明存在子shell。

```
lxc@Lxc:~/scripts/ch05$ pwd; ls *.png; cd /etc; pwd; cd; pwd; echo
$BASH_SUBSHELL
/home/lxc/scripts/ch05
parchildshell2.png  parchishell.png
/etc
/home/lxc
0
lxc@Lxc:~$
```

在输出结果的最后是数字0，这表明并未创建子shell来执行这些命令。
如果改用进程列表，则结果就不一样了。

```
lxc@Lxc:~/scripts/ch05$ (pwd; ls *.png; cd /etc; pwd; cd; pwd; echo
$BASH_SUBSHELL)
/home/lxc/scripts/ch05
parchildshell2.png  parchishell.png
/etc
/home/lxc
1
lxc@Lxc:~/scripts/ch05$
```

这次输出结果的最后数字是1。这表明的确创建了子shell来执行这些命令。此外，还有一点你可以发现，使用命令列表时不使用子shell执行命令，在执行完命令列表时，当前的目录成了`lxc@Lxc:~$`。而使用进程列表，会创建子shell来执行这些命令，命令执行完毕后的目录保持不变`lxc@Lxc:~/scripts/ch05$`。
因此，**进程列表就是使用圆括号包围起来的一组命令，它能够创建子shell来执行这些命令。**
你还可以在进程列表中嵌套圆括号来创建子shell的子shell：

```
lxc@Lxc:~/scripts/ch05$ (pwd; echo $BASH_SUBSHELL)
/home/lxc/scripts/ch05
1
lxc@Lxc:~/scripts/ch05$ (pwd; (echo $BASH_SUBSHELL))
/home/lxc/scripts/ch05
2
```

在第一个进程列表中，数字1表明有一个子shell，这个结果符合预期。在第二个进程列表中，其中嵌套了一个进程列表，这个嵌套的进程列表在子shell进程中产生了另一个子shell来执行该命令。因此数字2表示的就是这个子shell。
子shell在shell脚本中经常用于多进程处理。但是，创建子shell的成本不菲，会明显拖慢任务进度。在交互式CLI会话中，子shell存在同样的问题，它并非真正的多进程处理，原因在于终端与子shell的I/O绑定在了一起。

2. 别出心裁的子shell用法

进程列表、协程和管道都用到了子sehll，各自都可以有效运用于交互式shell。
在交互式shell中，一种高效的子shell用法是后台模式。在讨论如何配合使用后台模式和子shell之前，需要先搞明白什么是后台模式。

1. 探究后台模式

在后台模式中运行命令可以在处理命令的同时让出CLI，以供他用。要想将命令置于后台模式，可以在命令末尾加上字符 `&`。
以后台模式运行脚本，可参见 第16章。

来个例子：

```
lxc@Lxc:~/scripts/ch05$ sleep 30&
[1] 7828
lxc@Lxc:~/scripts/ch05$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
lxc         6137    6132  0 14:01 pts/0    00:00:00 bash
lxc         7828    6137  0 14:12 pts/0    00:00:00 sleep 30
lxc         7829    6137  0 14:12 pts/0    00:00:00 ps -f
```

`sleep` 命令会在后台（&）睡眠30秒。当被置于后台时，在shell CLI提示符返回之前，屏幕上会出现两条信息。第一条信息是方括号中的后台作业号（1）。第二条信息是后台进程ID（7828）。
除了 `ps` 命令，也可以使用 `jobs` 命令来显示后台作业信息。`jobs` 命令能够显示当前运行在后台模式中属于你的所有进程（作业）：

```
lxc@Lxc:~/scripts/ch05$ jobs
[1]+  已完成               sleep 30
```

`jobs` 命令会在方括号中显示作业号（1）。除此之外，还有作业的当前状态（已完成）以及对应的命令（sleep 30）。

利用 `jobs` 命令的 `-l` 选项，还可以看到更多的相关信息。除了默认信息，`-l` 选项还会显示命令的PID。

```
lxc@Lxc:~/scripts/ch05$ sleep 1&
[1] 8190
lxc@Lxc:~/scripts/ch05$ jobs -l
[1]+  8190 已完成               sleep 1
```

> **提示：** 如果运行多个后台进程，则还有一些额外信息可以显示哪个后台作业是最近启动的。在 `jobs` 命令的显示中，最近启动的作业是在其作业号之后会有一个加号（+），在它之前启动的进程（the second newest process）则以减号（-）表示。这两个符号还被用于表示默认作业，参见 第16章 的注意部分。

2. 将进程列表置于后台

通过将进程列表置入后台，可以在子shell中进行大量的多进程处理。由此带来的一个好处是终端不再和子shell的I/O绑定在一起。
之前说过，进程列表是在子shell中运行的一系列命令。

```
lxc@Lxc:~/scripts/ch05$ (sleep 2; echo $BASH_SUBSHELL; sleep 2)
1
```

在上面的例子中，执行命令后，2秒后，显示数字1，再过2秒后，返回命令行提示符。这符合我们的预期。

```
lxc@Lxc:~/scripts/ch05$ (sleep 2; echo $BASH_SUBSHELL; sleep 2)&
[1] 8704
lxc@Lxc:~/scripts/ch05$ 1

[1]+  已完成               ( sleep 2; echo $BASH_SUBSHELL; sleep 2 )
```

在这个例子中，执行命令后，立刻返回作业号（1）和进程ID（8704），返回命令行提示符。两秒后，数字1出现在提示符右侧。再次按下Enter键，就会得到另一个提示符了（如果自显示数字1之后，再过2秒，按下Enter键，就会显示后台作业的结束状态）。如下：

```
lxc@Lxc:~/scripts/ch05$ (sleep 2; echo $BASH_SUBSHELL; sleep 2)&
[1] 9013
lxc@Lxc:~/scripts/ch05$ # 这里是按下了Enter键
lxc@Lxc:~/scripts/ch05$ # 这里是按下了Enter键
lxc@Lxc:~/scripts/ch05$ # 这里是按下了Enter键
lxc@Lxc:~/scripts/ch05$ 1

lxc@Lxc:~/scripts/ch05$ # 这里是按下了Enter键
[1]+  已完成               ( sleep 2; echo $BASH_SUBSHELL; sleep 2 )
```

当然，这里的只是示例而已，你可以使用后台进程列表执行你想完成的任务。

```
lxc@Lxc:~/tt$ (tar -czf Doc.tar.gz text1.txt text2.txt ; tar -czf Doc2.tgz
text2.txt text3.txt )&
[1] 9143
lxc@Lxc:~/tt$ ls
Doc2.tgz  Doc.tar.gz  text1.txt  text2.txt  text3.txt
[1]+  已完成               ( tar -czf Doc.tar.gz text1.txt text2.txt; tar -czf
Doc2.tgz text2.txt text3.txt )
# 注意，后台作业的结束状态会在下一个显示提示符的命令执行后打印出。
# 你如果想单独看后台作业的结束状态，单纯的按一下Enter键也行。
```

将进程列表置入后台并不是子shell在CLI中仅有的创造性方法，还有一种方法是协程。

3. 协程

协程同时做两件事：一是在后台生成一个子shell；二是在该子shell中执行命令。
要进行协程处理，可以结合 `coproc` 命令以及要在子shell中执行的命令。

```
lxc@Lxc:~/scripts/ch05$ coproc sleep 10
[1] 9544
lxc@Lxc:~/scripts/ch05$ jobs
[1]+  运行中               coproc COPROC sleep 10 &
```

除了会创建子shell，协程基本上就是将命令置入后台。当输入 `coproc` 命令及其参数之后，你会发现后台启用了一个作业。屏幕上会显示该后台作业号（1）以及进程ID（2689）。
从上面的例子中可以看到，在子shell中执行的后台命令 `coproc COPROC sleep 10 &`。COPROC是 `coproc` 命令给进程起的名字。可以使用命令的扩展语法来自己设置这个命令：

```
lxc@Lxc:~/scripts/ch05$ coproc My_Job { sleep 10; }
[1] 9864
lxc@Lxc:~/scripts/ch05$ jobs
[1]+  运行中               coproc My_Job { sleep 10; } &
```

使用扩展语法，协程名被设置成了 My_Job。这里要注意，扩展语法写起来有点小麻烦。你必须确保在左花括号和内部命令名之间有一个空格，内部命令必须以分号结尾，分号和右花括号之间也得有一个空格。

> **注意：** 协程能够让你尽情的发挥想象力，发送或接收来自子shell中进程的信息。只有在拥有多个协程时才需要对协程进行命名，因为你要和它们进程通信。否则的话，让 *coproc* 命令将其设置成默认名称 COPROC 即可。

你可以将协程和进程列表结合起来创建嵌套子shell。只需将命令 `coproc` 放在进程列表之前即可：

```
lxc@Lxc:~/scripts/ch05$ coproc ( sleep 10; sleep 2 )
[1] 10293
lxc@Lxc:~/scripts/ch05$ jobs
[1]+  运行中               coproc COPROC ( sleep 10; sleep 2 ) &
lxc@Lxc:~/scripts/ch05$ ps --forest
    PID TTY          TIME CMD
   6975 pts/2    00:00:00 bash
  10293 pts/2    00:00:00  \_ bash
  10294 pts/2    00:00:00  |   \_ sleep
  10345 pts/2    00:00:00  \_ ps
```

记住，生成子shell的成本可不低，而且速度很慢。创建嵌套子shell更是火上浇油。
子shell提供了灵活性和便利性。要想获得这些优势，重要的是要理解子shell的行为方式。对于命令也是如此。下一节将研究内建命令与外部命令之间的行为差异。

3. 理解外部命令和内建命令

搞明白shell的内建命令和外部命令非常重要。两者的操作方式大不相同。

1. 外部命令

外部命令（有时也称为文件系统命令） 是存在于bash shell之外的程序。也就是说，它并不属于shell程序的一部分。外部命令程序通常位于 /bin、/usr/bin、/sbin、/usr/sbin目录中。

`ps` 命令就是一个外部命令。可以使用 `which` 或 `type` 命令来找到对应的文件名：

```
lxc@Lxc:~/scripts/ch05$ which ps
/usr/bin/ps
lxc@Lxc:~/scripts/ch05$ type ps
ps 已被录入哈希表 (/usr/bin/ps)
lxc@Lxc:~/scripts/ch05$ ls -l /usr/bin/ps
-rwxr-xr-x 1 root root 137688 10月 31 19:35 /usr/bin/ps
```

每当执行外部命令时，就会创建一个子进程。这种操作称为 **衍生（forking）**。外部命令 `ps` 会显示其父进程以及自己所对应的衍生子进程。

```
lxc@Lxc:~/scripts/ch05$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
lxc         6137    6132  0 14:01 pts/0    00:00:00 bash
lxc        10941    6137  0 15:24 pts/0    00:00:00 ps -f
lxc@Lxc:~/scripts/ch05$ echo $$
6137
```

作为外部命令，`ps` 命令在执行时会产生一个子进程。在这里，`ps` 命令的PID是 10941，PPID是6137。下图展示了外部命令执行时的衍生过程。

只要涉及进程衍生，就需要耗费时间和资源来设置子进程的环境。因此，外部命令系统开销较高。

> 无论是衍生出子进程还是创建了子进程，都仍然可以通过信号与其互通，这一点无论是在命令行还是在编写脚本时都及其有用。进程间以发送信号的方式彼此通信。第16章 将介绍信号和信号发送。

在使用内建命令时，不需要衍生子进程。因此，内建命令的系统开销较低。

2. 内建命令

与外部命令不同，**内建命令无须使用子进程来执行。内建命令已经和shell编译成一体，作为shell的组成部分存在，无须借助外部程序文件来执行。**

`cd` 命令和 `exit` 命令都内建于bash shell。可以使用 `type` 命令来判断某个命令是否为内建：

```
lxc@Lxc:~/scripts/ch05$ type cd
cd 是 shell 内建
lxc@Lxc:~/scripts/ch05$ type exit
exit 是 shell 内建
lxc@Lxc:~/scripts/ch05$ type type
type 是 shell 内建
```

因为内建命令既不需要衍生出子进程来执行，也不用打开程序文件，所以执行速度更快，效率也更高。注意，有些命令有多种实现。例如，`echo` 和 `pwd` 既有内建命令也有外部命令。两种实现有差异。要查看命令的不同实现，可以使用 `type` 的 `-a` 选项：

```
lxc@Lxc:~/scripts/ch05$ type -a echo
echo 是 shell 内建
echo 是 /usr/bin/echo
echo 是 /bin/echo
lxc@Lxc:~/scripts/ch05$ which echo
/usr/bin/echo
lxc@Lxc:~/scripts/ch05$ type -a pwd
pwd 是 shell 内建
pwd 是 /usr/bin/pwd
pwd 是 /bin/pwd
lxc@Lxc:~/scripts/ch05$ which pwd
/usr/bin/pwd
```

`type -a` 显示每个命令的两种实现（内建和外部）。`which` 命令只显示外部命令文件。

> **提示：** 对于有多种实现的命令，如果想使用其外部命令实现，直接指明对应的文件即可。例如，要使用外部命令 `pwd`，可以输入 `/usr/bin/pwd`。

1. 使用 `history` 命令

bash shell会跟踪你最近使用过的命令。你可以重新唤回这些命令，加以重用。`history` 是一个实用的内建命令，能帮助你管理先前执行过的命令。
要查看最近用过的命令，可以使用不带任何选项的 `history` 命令：

```
lxc@Lxc:~/scripts/ch05$ history
    1  cd tt/
    2  ls
    3  cd ..
    ......省略了很长的输出
```

> **提示：** 你可以修改保存在bash历史记录中的命令数量。为此，需要修改名为 `HISTSIZE` 的环境变量（参见 第6章）。

你可以唤回并重用历史记录列表中最近的那条命令。输入 `!!`，然后按Enter键即可。

```
lxc@Lxc:~/scripts/ch05$ ps --forest
    PID TTY          TIME CMD
  12552 pts/2    00:00:00 bash
  14143 pts/2    00:00:00  \_ ps
lxc@Lxc:~/scripts/ch05$ !!
ps --forest
    PID TTY          TIME CMD
  12552 pts/2    00:00:00 bash
  14155 pts/2    00:00:00  \_ ps
```

当输入 `!!` 时，bash会先显示从shell的历史记录中唤回的命令，然后再执行该命令。
历史记录中的命令被保存在 `~/.bash_history` 文件中。
这里要注意的是，在CLI会话期间，bash命令的历史记录被保存在内存之中。当shell退出时才被写入历史文件中，这个问题在 第6章 已有论述。

可以在不退出shell的情况下强制将命令历史记录写入.bash_history文件。为此，需要使用 `history` 命令的 `-a` 选项：

```
lxc@Lxc:~/scripts/ch05$ history -a
lxc@Lxc:~/scripts/ch05$ history | tail
  569  (tar -czf Doc.tar.gz text1.txt text2.txt ; tar -czf Doc2.tgz text2.txt
text3.txt )&
  570  ls
  571  ls Doc.tar.gz
  572  cd
  573  cd scripts/ch05/
  574  ps -f
  575  echo $$
  576  man history
  577  history -a
  578  history | tail
lxc@Lxc:~/scripts/ch05$ cat ~/.bash_history | tail
ls
(tar -czf Doc.tar.gz text1.txt text2.txt ; tar -czf Doc2.tgz text2.txt
text3.txt )&
ls
```

```
ls Doc.tar.gz
cd
cd scripts/ch05/
ps -f
echo $$
man history
history -a
```

注意，`history` 命令的输出与 ~/.bash_history 文件内容是一样的。除了最后的那条 `history | tail` 命令。

> **注意：** 如果打开了多个终端会话，则仍然可以使用 `history -a` 命令在每个打开的会话中向 .bash_history 文件添加记录。但是历史记录并不会在其他打开的终端会话中自动更新。这是因为.bash_history 文件只在首次启动终端会话的时候才会被读取。要想强制重新读取 .bash_history 文件，更新内存中的终端会话历史记录，可以使用 `history -n` 命令。

你可以唤回历史记录中的任意命令。只需输入惊叹号和命令在历史记录中的编号即可：

```
lxc@Lxc:~/scripts/ch05$ history | tail
  572  cd
  573  cd scripts/ch05/
  574  ps -f
  575  echo $$
  576  man history
  577  history -a
  578  history | tail
  579  cat ~/.bash_history | tail
  580  man history
  581  history | tail
lxc@Lxc:~/scripts/ch05$ !574
ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
lxc         6137    6132  0 14:01 pts/0    00:00:00 bash
lxc        15119    6137  0 16:37 pts/0    00:00:00 ps -f
```

和执行最近的命令一样，bash shell会先显示从历史记录中唤回的命令，然后再执行该命令。

> **提示：** 如果要清除历史记录，很简单，输入 `history -c` 即可。接下来再输入 `history -a`，清除.bash_history文件。

内建命令 `history` 能做到的事情远不止这些。可以通过 `man history` 来查看 `history` 的手册页。

2. 使用命令别名

`alias` 命令是另一个实用的内建命令。命令别名允许为常用命令及其参数创建另一个名称。
你所使用的Linux发行版很有可能已经为你设置好了一些常用命令的别名。使用 `alias` 命令以及选项 `-p` 可以查看当前可用的别名：

```
lxc@Lxc:~/scripts/ch05$ alias -p
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -alF'
alias ls='ls --color=auto'
......
```

注意在该Ubuntu Linux发行版中，有一个别名取代了标准命令 `ls`。该别名中加入了 `--color=auto` 选项，以便在终端支持彩色的情况下，`ls` 命令可以使用色彩编码（比如，使用蓝色表示目录）。`LS_COLORS` 环境变量控制着所用的彩色编码。

> **提示：** 如果经常跳转于不同的发行版，那么在使用色彩编码来分辨某个名称究竟是目录还是文件时，一定要小心。因为色彩编码并未实现标准化。所以最好用 `ls -F` 来判断文件类型。

可以使用 `alias` 命令创建自己的别名：

```
lxc@Lxc:~/scripts/ch05$ alias li='ls -i'
lxc@Lxc:~/scripts/ch05$ li
2123732 extern.png  2123049 parchildshell2.png  2125585 parchishell.png
 2121945 README.md
lxc@Lxc:~/scripts/ch05$ bash
lxc@Lxc:~/scripts/ch05$ li
li: 未找到命令
lxc@Lxc:~/scripts/ch05$ exit
lxc@Lxc:~/scripts/ch05$ li
2123732 extern.png  2123049 parchildshell2.png  2125585 parchishell.png
 2121945 README.md
```

定义好别名之后，就可以随时在shell或者shell脚本中使用了。要注意，因为命令别名属于内建命令，所以别名仅在其被定义的shell进程中才有效。
可以使用 unlias *alias_name* 来删除命令别名。
如果想持久化别名的效果，可以通过修改环境文件来实现。可以参考第6章。

ch06 Linux环境变量

很多程序和脚本通过环境变量来获取系统信息、存储临时信息和配置信息。在Linux系统中，有很多地方可以设置环境变量，了解去哪里设置相应的环境变量很重要。
本章先带你逐步了解Linux环境变量：它们存储在哪里、如何使用，以及如何创建自己的环境变量，最后以数组变量的用法作结。

1. 什么是环境变量

bash shell使用 **环境变量** 来存储shell会话和工作环境的相关信息（这也是被称作环境变量的原因）。环境变量允许在内存中存储数据，以便shell中运行的程序或脚本能够轻松访问到这些数据。这也是存储持久数据的一种简便方法。
bash shell中有两种环境变量。

- 全局变量
- 局部变量

> **注意：** 尽管bash shell使用的专有环境变量是一致的，但不同的Linux发行版经常会添加自己的环境变量，可以查看你的Linux发行版文档。

1. 全局环境变量

全局环境变量对于shell会话和所有生成的子shell都是可见的。局部环境变量则只对创建它的shell可见。
如果程序创建的子shell需要获取父shell的信息，那么全局环境变量就能派上用场。
Linux系统在你启动bash会话时就设置好了一些全局环境变量（6.6节将展示具体都有哪些变量）。系统环境变量基本上会使用大写字母，以区别用户自定义的环境变量。
可以使用 `env` 命令或 `printenv` 命令来查看全局变量：

```
lxc@Lxc:~/scripts/ch06$ env
SHELL=/bin/bash
......
```

Linux系统为bash shell设置的全局环境变量数目众多，上面省略了输出。其中很多是在登录的过程中设置的，另外，你的登录方式也会影响所设置的环境变量。
要显示个别环境变量可以使用 `printenv` 命令，不能使用 `env` 命令。也可以使用 `echo` 命令。

```
lxc@Lxc:~/scripts/ch06$ printenv HOME
/home/lxc
lxc@Lxc:~/scripts/ch06$ echo $HOME
/home/lxc
lxc@Lxc:~/scripts/ch06$ env HOME
env: “HOME”: 没有那个文件或目录
```

在变量名前加上 `$` 可不仅仅是能够显示变量的当前值，它还能让变量作为其它命令的参数：

```
lxc@Lxc:~/scripts/ch06$ ls $HOME
公共的  模板  视频  图片  文档  下载  音乐  桌面 .......
```

如前所述，全局环境变量可用于进程的子shell。

```
lxc@Lxc:~/scripts/ch06$ bash
lxc@Lxc:~/scripts/ch06$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
lxc         5670    3319  0 14:51 pts/3    00:00:00 bash
lxc         6039    5670  0 14:51 pts/3    00:00:00 bash
lxc         6225    6039  0 14:52 pts/3    00:00:00 ps -f
lxc@Lxc:~/scripts/ch06$ echo $HOME
/home/lxc
lxc@Lxc:~/scripts/ch06$ exit
exit
```

在这个例子中，用 `bash` 命令生成一个子shell后，显示了HOME环境变量的当前值。这个值和父shell中的值一模一样。

2. 局部环境变量

顾名思义，**局部环境变量** 只能在定义它的进程中可见。Linux系统默认定义了标准的局部环境变量。在命令行中查看局部环境变量有点棘手。遗憾的是，没有哪个命令可以只显示这类变量。
`set` 命令可以显示特定进程的所有环境变量，既包括局部变量、全局变量也包括用户自定义变量：

```
lxc@Lxc:~$ set
BASH=/usr/bin/bash
......省略了很长很长的输出
```

> **注意：** `env` 命令、`printenv` 命令和 `set` 命令之间的差异很细微。`set` 命令既会显示全局和局部环境变量、用户自定义变量以及局部shell函数，还会按照字母顺序对结果进行排序。与 `set` 命令不同，`env` 和 `printenv` 命令不会对变量进行排序，也不会输出局部环境变量、局部用户自定义变量以及局部shell函数。在这种情况下 `env` 和 `printenv` 命令的输出是重复的。不过 `env` 命令有 `printenv` 命令不具备的一个功能，这使其略胜一筹。

2. 设置用户自定义变量

你可以在bash shell中直接设置自己的变量。

1. 设置局部用户自定义变量

启动bash shell后，就能创建仅对该shell进程可见的局部用户自定义变量。可以使用等号为变量赋值，值可以是数值或字符串。

```
lxc@Lxc:~/scripts/ch06$ my_variable=haha
lxc@Lxc:~/scripts/ch06$ echo $my_variable
haha
```

如果用于赋值的字符号包含空格，则必须用单引号或双引号来界定字符串的起止：

```
lxc@Lxc:~/scripts/ch06$ my_variable="hello world"
lxc@Lxc:~/scripts/ch06$ echo $my_variable
hello world
lxc@Lxc:~/scripts/ch06$ my_variable=hello world

Command 'world' not found, but can be installed with:

sudo snap install world

```

如你所见，如果没有引号，那么bash shell会将下一个单词视为要执行的命令。

> **提示：** bash shell的惯例是所有的环境变量均使用大写字母命令。如果是你自己创建或在shell脚本中使用的局部变量，则使用小写字母命名。变量名区分大小写。坚持使用小写字母命名用户自定义的局部变量可以让你避免不小心与系统环境变量同名可能带来的灾难。

记住，在变量名、等号和值之间没有空格，这一点非常重要。如果在赋值表达式加上了空格，那么bash shell会将其视为单独的命令。

设置好局部变量之后，就能在shell进程中使用了。但如果又生成了另一个shell，则该变量在子shell中不可用，反之，也成立：

```
lxc@Lxc:~/scripts/ch06$ my_variable="Hello World"
lxc@Lxc:~/scripts/ch06$ bash
lxc@Lxc:~/scripts/ch06$ echo $my_variable

lxc@Lxc:~/scripts/ch06$ exit
lxc@Lxc:~/scripts/ch06$ echo $my_variable
Hello World
# 反之也是如此。
lxc@Lxc:~/scripts/ch06$ echo $my_child_variable

lxc@Lxc:~/scripts/ch06$ bash
lxc@Lxc:~/scripts/ch06$ my_child_variable="Hello Little World"
lxc@Lxc:~/scripts/ch06$ echo $my_child_variable
Hello Little World
lxc@Lxc:~/scripts/ch06$ exit
exit
lxc@Lxc:~/scripts/ch06$ echo $my_child_variable

```

```
    # 注，该行是一个空行
# 返回父shell后，子shell中设置的局部变量就不存在了。
```

2. 设置全局环境变量

全局环境变量在设置该变量的父进程所创建的子进程中都是可见的。创建全局环境变量的方法是先创建局部变量，然后再将其导出到全局环境变量中。当然，也可以将设置变量和导出变量放在一个命令里完成。
这可以通过 `export` 命令以及要导出的变量名（不加$符号）来实现：

```
lxc@Lxc:~/scripts/ch06$ my_variable="I am Global now"
lxc@Lxc:~/scripts/ch06$ export my_variable
lxc@Lxc:~/scripts/ch06$ echo $my_variable
I am Global now
lxc@Lxc:~/scripts/ch06$ bash
lxc@Lxc:~/scripts/ch06$ echo $my_variable
I am Global now
lxc@Lxc:~/scripts/ch06$ exit
lxc@Lxc:~/scripts/ch06$ echo $my_variable
I am Global now
```

修改子shell中的全局环境变量并不会影响父shell中该变量的值：

```
lxc@Lxc:~/scripts/ch06$ export my_variable="I am global now"
lxc@Lxc:~/scripts/ch06$ echo $my_variable
I am global now
lxc@Lxc:~/scripts/ch06$ bash
lxc@Lxc:~/scripts/ch06$ echo $my_variable
I am global now
lxc@Lxc:~/scripts/ch06$ my_variable="Null"
lxc@Lxc:~/scripts/ch06$ echo $my_variable
Null
lxc@Lxc:~/scripts/ch06$ exit
lxc@Lxc:~/scripts/ch06$ echo $my_variable
I am global now
```

子shell无法使用 `export` 命令改变父shell中全局环境变量中的值：

```
lxc@Lxc:~/scripts/ch06$ export my_variable="I am global now hahaha"
lxc@Lxc:~/scripts/ch06$ echo $my_variable
I am global now hahaha
lxc@Lxc:~/scripts/ch06$ bash
lxc@Lxc:~/scripts/ch06$ export my_variable="Null"
lxc@Lxc:~/scripts/ch06$ echo $my_variable
Null
lxc@Lxc:~/scripts/ch06$ exit
lxc@Lxc:~/scripts/ch06$ echo $my_variable
I am global now hahaha
```

3. 删除环境变量

可以使用 `unset` 命令来删除已有的环境变量。

```
lxc@Lxc:~/scripts/ch06$ my_variable="I am going to be removed"
lxc@Lxc:~/scripts/ch06$ echo $my_variable
I am going to be removed
lxc@Lxc:~/scripts/ch06$ unset my_variable
lxc@Lxc:~/scripts/ch06$ echo $my_variable
    # 这是一个空行
```

> **提示：：** 在涉及环境变量名时，什么时候该使用 `$`，什么时候不该使用 `$`，可以按以下考虑：如果要用到（doing anything with）变量，就是用 `$`；如果要操作（doing anything to）变量。则不使用 `$`。唯一例外是使用 `printenv` 命令显示某个变量的值。

如果在子进程中删除了一个全局环境变量，那么该操作仅对子进程有效。该全局环境变量在父进程中依然可用：

```
lxc@Lxc:~/scripts/ch06$ export my_variable="I am global now"
lxc@Lxc:~/scripts/ch06$ echo $my_variable
I am global now
lxc@Lxc:~/scripts/ch06$ bash
lxc@Lxc:~/scripts/ch06$ echo $my_variable
I am global now
lxc@Lxc:~/scripts/ch06$ unset my_variable
lxc@Lxc:~/scripts/ch06$ echo $my_variable

lxc@Lxc:~/scripts/ch06$ exit
lxc@Lxc:~/scripts/ch06$ echo $my_variable
I am global now
```

4. 默认的shell环境变量

在默认情况下，bash shell 会用一些特定的环境变量来定义系统环境。这些变量在你的Linux系统中都已设置好，只管放心使用就行了。由于bash shell源于最初的Unix Bourne shell，因此也保留了其中定义的那些环境变量。

除了默认的Bourne shell环境变量，bash shell还提供了一些自有的变量，如下表所示：

这里详细说明一下 `HISTFILESIZE` 和 `HISTSIZE` 这两个环境变量的区别。先要区别 "历史记录列表" 和 "历史记录文件"。前者位于内存中，在bash会话期间更新。后者位于硬盘上，在bash shell中通常是 `~/.bash_history` 文件。会话结束后，历史记录列表中的内容会被写入历史记录文件。如果 `HISTFILESIZE` = 200，表示历史记录文件中最多能保存200条命令；如果 `HISTSIZE` = 20，则在一个shell会话中，不管输入多少条命令，历史记录列表中只记录20条命令，最终也只有这20条命令会在会话结束后被写入历史记录文件中。

你可能已经注意到，不是所有的默认环境变量都会在 `set` 命令中列出。如果用不到，默认环境变量并不要求必须有值。

> **注意：** 系统使用的默认环境变量有时取决于bash shell的版本。例如，`EPOCHREALTIME` 仅在bash shell版本5及更高的版本中可用。可以在CLI中输入 `bash --version` 来查看bash shell的版本号。

5. 设置 PATH 环境变量

当你在shell CLI中输入一个外部命令（参见第5章）时，shell必须搜索系统，从中找到对应的程序。
PATH 环境变量定义了用于查找命令和程序的目录。

```
lxc@Lxc:~/scripts/ch06$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:.....省略
```

PATH 中的目录之间以冒号分隔。shell会在其中查找命令和程序。
如果命令和程序所在位置没有包括在 PATH 变量中，那么在不使用绝对路径的情况下，shell是无法找到的。
你可以把新的搜索目录加入到现有的 PATH 环境变量中，无须从头定义。PATH 中各个目录之间以冒号分隔。只需引用原来的 PATH 值，添加冒号（:），然后再使用绝对路径输入新目录即可。

```
export PATH=$PATH:$HADOOP_HOME/bin
```

将该目录加入到 PATH 环境变量之后，就可以在虚拟目录结构的任意位置执行这个程序了。如果你希望程序位置也可以应用于子shell，就应该像上面那样使用 export 将其导出。

> **注意：** 有些脚本编写人员使用 env 命令作为bash shell脚本（参见第11章）的第一行，就像这样：`#!/usr/bin/env bash`。这样的方法的优点在于 env 命令会在 $PATH 中搜索bash，使脚本具备更好的移植性。

6. 定位系统环境变量

本节介绍如何持久化环境变量。
当你登录Linux系统启动bash shell时，默认情况下bash会从几个文件中查找命令。这些文件称作 **启动文件** 或 **环境文件**。bash进程的启动文件取决于你启动bash shell的方式。启动bash shell有以下3种方式：

- 登录时作为默认登录shell；
- 作为交互式shell，通过生成子shell启动；
- 作为运行脚本的非交互式shell。

下面介绍bash shell在不同的启动方式下执行的启动文件。

1. 登录shell

当你登录Linux系统时，bash shell会作为 **登录shell** 启动。登录shell通常会从5个不同的启动文件中读取命令。

- /etc/profile
- $HOME/.bash_profile
- $HOME/.bashrc
- $HOME/.bash_login
- $HOME/.profile

/etc/profile 文件是系统中默认的bash shell的主启动文件。系统中的每个用户登陆时都会执行这个启动文件。

> **注意：** 要留意的是有些Linux发行版使用了可拆卸式认证模块(pluggable authentication module，PAM)。在这种情况下，PAM文件会在bash shell启动之前被处理，前者中可能会包含环境变量。PAM文件包括 /etc/environment 文件和 $HOME/.pam_environment文件。

另外四个启动文件是针对用户的，位于用户主目录中，可根据个人具体定制。下面来看看这几个文件。

1. /etc/profile 文件

/etc/profile 文件是bash shell默认的主启动文件。只要登录Linux系统，bash就会执行 /etc/profile 启动文件中的命令。不同的Linux发行版在这个文件放置了不同的命令。在本书所用的Ubuntu Linux系统中，该文件如下所示：

```
lxc@Lxc:~$ cat /etc/profile
# /etc/profile: system-wide .profile file for the Bourne shell (sh(1))
# and Bourne compatible shells (bash(1), ksh(1), ash(1), ...).

if [ "${PS1-}" ]; then
  if [ "${BASH-}" ] && [ "$BASH" != "/bin/sh" ]; then
    # The file bash.bashrc already sets the default PS1.
    # PS1='\h:\w\$ '
    if [ -f /etc/bash.bashrc ]; then
      . /etc/bash.bashrc
    fi
  else
    if [ "`id -u`" -eq 0 ]; then
      PS1='# '
    else
      PS1='$ '
    fi
  fi
fi

if [ -d /etc/profile.d ]; then
  for i in /etc/profile.d/*.sh; do
    if [ -r $i ]; then
      . $i
    fi
  done
  unset i
fi
```

你应该是能看懂这个文件的。
CentOS发行版的 /etc/profile 文件不再展示（我没装那个系统，懒得复制）。

这两种发行版的 /etc/profile 文件都使用了 `for` 语句来迭代 /etc/profile.d 目录下的所有文件。**这为Linux 系统提供了一个位置放置特定应用程序启动文件或管理员自定义启动文件的地方**，shell会在用户登录时执行这些文件。在本书Ubuntu Linux系统中，/etc/profile.d 目录下包含下列文件：

```
lxc@Lxc:~/scripts/ch06$ ls /etc/profile.d/
01-locale-fix.sh  apps-bin-path.sh  bash_completion.sh  cedilla-portuguese.sh
flatpak.sh  gawk.csh  gawk.sh  im-config_wayland.sh  vte-2.91.sh  vte.csh
xdg_dirs_desktop_session.sh
```

不难发现，有些文件与系统中的特定应用程序有关。大部分应用程序会创建两个启动文件：一个供bash shell使用（扩展名为*.sh），另一个供C shell使用（扩展名 .csh）。

2. $HOME目录下的启动文件

其余的这四个启动文件都用于同一个目的：提供用户专属的启动文件来定义该用户所用到的环境变量。大多数Linux发行版只用到这4个启动文件中的一两个。

- $HOME/.bash_profile
- $HOME/.bashrc
- $HOME/.bash_login
- $HOME/.profile

这些文件都以点号开头，是隐藏文件。因为它们位于用户的$HOME目录下，所以每个用户可以对其进行编辑并添加自己的环境变量，其中的环境变量会在每次启动shell会话时生效。

> **注意：** Linux发行版在环境文件方面存在的差异巨大。本节所列出的$HOME文件下的那些文件并非每个用户都有。例如，有些用户可能只有一个$HOME/.bashrc文件。这很正常。

shell会按照下列顺序执行第一个被找到的文件，其余的则被忽略：

1. $HOME/.bash_profile
2. $HOME/.bash_login
3. $HOME/.profile

你会发现这个列表中没有 $HOME/.bashrc 文件，这是因为该文件通常通过其他文件运行。

> **提示：** 记住，$HOME 代表某个用户的主目录，和波浪号(~)的效果一样。

2. 交互式shell进程

如果不是在登陆系统时启动的bash shell（比如在命令行中输入bash），那么这时的shell称作 **交互式shell**。与登录shell一样，交互式shell提供了命令行提示符供用户输入命令。
作为交互式shell启动的bash并不处理 /etc/profile 文件，只检查用户$HOME目录中的.bashrc文件。.bashrc 文件会做两件事：首先，检查 /etc 目录下的通用bashrc文件；其次，为用户提供一个定制自己的命令别名和脚本函数的地方。

3. 非交互式shell

系统执行脚本时使用的shell就是 **非交互式shell**。不同之处在于它没有命令行提示符。

你在系统中运行脚本时，也许希望能够运行一些特定的启动命令。为了处理这种情况，bash shell提供了 `BASH_ENV` 环境变量。可以查看 "bashshell1.png"。 `BASH_ENV` 环境变量：如果设置了的话，每个bash脚本会在运行前先尝试运行该变量定义的启动文件。当shell启动一个非交互式shell进程时，会检查这个环境变量以查看要执行的启动文件名。

```
lxc@Lxc:~$ echo $BASH_ENV
    # 这是一个空行
```

如果未设置 `$BASH_ENV`，shell脚本到哪里去获取其环境变量呢？别忘了有些shell脚本是通过启动一个子shell来执行的。子shell会继承父shell的导出变量。如果父shell是登录shell，在 /etc/profile 文件、/etc/profile.d/*.sh 文件和 $HOME/.bashrc 文件中设置并导出了变量，那么用于执行脚本的子shell就能继承这些变量。

> **提示：** 任何由父shell设置但未导出的变量都是局部变量，不会被子shell继承。

对于那些不启动子shell的脚本，变量已经存在于当前shell了。就算没有设置 `BASH_ENV`，也可以使用当前shell的局部变量和全局变量。

4. 环境变量持久化

对于全局环境变量（Linux系统的所有用户都要用到的变量）来说，可能更倾向于将新的或修改过的变量放在 /etc/profile 文件中，但这可不是什么好主意。如果升级了所用的发行版，则该文件也会随之更新，这样一来，所有定制过的变量设置可就都没有了。
最好是在 /etc/profile.d 目录中创建一个以 .sh 结尾的文件。把所有新的或修改过的全局环境变量设置都放在这个文件中。
在大多数发行版中，保存个人用户永久性bash shell变量的最佳地点是 $HOME/.bashrc 文件。这适用于所有类型的shell进程。但如果设置了 `BASH_ENV` 环境变量，请记住：除非值为 $HOME/.bashrc，否则，应该将非交互式shell的用户变量放在别的地方。

> **注意：** 图形化界面组成部分（比如GUI客户端）的环境变量可能需要在另外一些配置文件中设置，这和设置bash shell环境变量的文件不一样。

第5章讲过，`alias` 命令设置无法持久生效。你可以把个人的 `alias` 设置放在 $HOME/.bashrc 启动文件中(该启动文件推荐你把命令别名放在 ~/.bash_aliases 这个文件下，~/.bashrc 文件会执行这个别名文件，这样做更清晰)，使其效果永久化。

7. 数组变量

环境变量的一个很酷的特性是可以作为数组使用。数组是能够存储多个值的变量。这些值既可以单独引用，也可以作为整体引用。
要为某个环境变量设置多个值，可以把值放在圆括号中，值与值之间以空格分隔：

```
lxc@Lxc:~/scripts/ch06$ mytest=(zero one two three four)
lxc@Lxc:~/scripts/ch06$ echo $mytest
zero
lxc@Lxc:~/scripts/ch06$ echo ${mytest[2]}
two
```

要引用单个数组元素，必须使用表示其在数组中位置的索引。注意，**环境变量数组的索引都是从0开始的**。索引要写在方括号中，$ 符号之后的所有内容都要放入花括号中。
要显示整个数组变量，可以使用通配符 * 作为索引。

```
lxc@Lxc:~/scripts/ch06$ echo ${mytest[*]}
zero one two three four
```

也可以改变某个索引位置上的值：

```
lxc@Lxc:~/scripts/ch06$ echo ${mytest[2]}
2
```

你可以用 `unset` 命令来删除数组中的某个值，但是要小心，这有点复杂。

```
lxc@Lxc:~/scripts/ch06$ unset mytest[2]
lxc@Lxc:~/scripts/ch06$ echo ${mytest[*]}
zero one three four
lxc@Lxc:~/scripts/ch06$ echo ${mytest[2]}

lxc@Lxc:~/scripts/ch06$ echo ${mytest[3]}
three
```

这个例子使用 `unset` 命令来删除索引位置2上的值。显示整个数组时，看起来好像其他索引已经填补了这个位置，但如果专门显示索引位置2上的值时，你会发现这个位置是空的。

可以在 `unset` 命令后跟上数组名来删除整个数组：

```
lxc@Lxc:~/scripts/ch06$ unset mytest
lxc@Lxc:~/scripts/ch06$ echo ${mytest[*]}
    # 这是一个空行
```

有时候，数组变量只会把事情搞得更复杂，所以在shell脚本编程时并不常用。数组并不太方便移植到其它shell环境，如果需要在不同的shell环境中从事大量的shell脚本编写工作，这是一个不足之处。有些bash系统环境变量用到了数组（比如 `BASH_VERSINFO`），但总体而言，你不会经常碰到数组。

ch07 理解Linux文件权限

Linux系统沿用了Unix文件权限的做法，允许用户和组根据每个文件和目录的安全性设置来访问文件。本章将讨论如何根据需要利用Linux文件安全系统来保护数据。

1. Linux安全性

Linux安全系统的核心是 **用户账户**。每个能访问Linux系统的用户都会被分配一个唯一的用户账户。用户对系统中各种对象的访问权限取决于他们登录系统时所用的账户。
用户权限是通过创建用户时的分配的 **用户ID（user ID，UID）** 来跟踪的。UID是个数值，每个用户都有一个唯一的UID。但用户在登陆系统时是使用 **登录名（login name）** 来代替UID登录的。登录名是用户用来登录系统的 **最长8字符** 的字符串（字符可以是数字或字母），同时会关联一个对应的密码。
本节将介绍管理用户帐户所需要的文件和工具。

1./etc/passwd 文件

Linux系统使用一个专门的文件 */etc/passwd* 来匹配登录名与对应的UID值。该文件包含了一些与用户有关的信息。

```
lxc@Lxc:~/scripts/ch07$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
......
lxc:x:1000:1000:Lxc,,,:/home/lxc:/bin/bash
......
```

文件内容很长，我们做了部分删减。root用户账户是系统的管理员，为其固定分配的UID是0。如你所见，Linux系统会为各种各样的功能创建不同的用户账户，而这些账户并非真正的人类用户。我们称其为 **系统账户**，它们是系统中运行的各种服务进程访问资源使用的特殊账户。所有运行在后台的服务都需要通过一个系统用户帐户登录到Linux系统中。
Linux为系统账户预留了500以下的UID。有些服务甚至要用特定的UID才能正常工作。为普通用户创建账

户时，大多数Linux系统会从500开始，将第一个可用UID分配给这个账户。（并非所有的Linux发行版都是这样，比如Ubuntu就是从1000开始的。）

/etc/passwd 文件中各个字段（字段分隔符为冒号）的含义如下：

- 登录用户名
- 用户密码
- 用户帐户的UID（数字形式）
- 用户帐户的组ID（数字形式）
- 用户帐户的文本描述（称为备注字段）
- 用户$HOME目录的位置
- 用户的默认shell

/etc/passwd 文件的密码字段都被设置为x，这可不是说所有的用户帐户都使用相同的密码。在早期的Linux系统中，*/etc/passwd* 文件包含经过加密的用户密码。但鉴于很多程序要从这个文件中获取用户信息，这在某种程度上成了一个安全问题。随着可以轻松破解加密密码的软件的出现，那些居心不良的人试图破解保存在 */etc/passwd* 中的密码时可谓如鱼得水。Linux开发者重新考虑了这一策略。现在，绝大多数Linux系统将用户密码保存在单独的文件中（称为shadow文件，位于 */etc/shadow*）只有特定的程序才能访问该文件。
如你所见，*/etc/passwd* 是一个标准的文本文件。你可以使用任何文本编辑器直接在其中手动用户管理（比如添加、修改或删除用户帐户），但这样做极其危险。如果 */etc/passwd* 文件受损，系统无法读取文件内容，则会导致用户（即便是root用户）无法正常登录。选择标准的Linux用户管理工具来执行这些用户管理任务会安全很多。

2. /etc/shadow 文件

/etc/shadow 文件对Linux系统密码管理提供了更多的控制。只有root用户才能访问 */etc/shadow* 文件，这使其相比 */etc/passwd* 文件安全许多。

/etc/shadow 文件为系统中的每个用户都保存了一条记录。

```
lxc:$6$bkKjxqw72Br.mk5YQ.NYnvLShA8L/yH7iDGJbT1:19263:0:99999:7:::
```

该文件每条记录都包含9个字段（同样使用冒号作为字段分隔符）。

1. 登录名
2. 加密后的密码
3. 自上次修改密码后已经过去的天数（从1970年1月1日开始计算）
4. 多少天后才能更改密码
5. 多少天后必须更改密码
6. 密码过期前提前多少天提醒用户更改密码
7. 密码过期后多少天禁用用户帐户
8. 用户帐户被禁用的日期（以从1970年1月1日到当时的天数表示）
9. 预留给以后使用的字段

有了shadow密码系统，Linux系统就可以更好地控制用户密码了，比如控制用户多久更改一次密码，以及如果密码未更新的话，什么时候禁用账户。

3. 添加新用户

添加新用户的主要工具是 `useradd`。该命令可以一次性轻松创建新用户账户并设置用户的$HOME目录结构。`useradd` 命令使用系统的默认值以及命令行参数来设置用户账户。要想查看所使用的Linux发行版的系统默认值。可以使用加了 `-D` 选项的 `useradd` 命令。

```
lxc@Lxc:~/scripts/ch07$ useradd -D
GROUP=100
HOME=/home
INACTIVE=-1
EXPIRE=
SHELL=/bin/sh
SKEL=/etc/skel
CREATE_MAIL_SPOOL=no
```

> **注意：** `useradd` 命令的默认值使用 */etc/default/useradd* 文件设置。另外，进一步的安全设置在 */etc/login.defs* 文件中设置。你可以调整这些文件，改变Linux系统默认的安全行为。

这些默认值的含义如下：

- 新用户会被添加到GID为100的公共组
- 新用户的主目录会位于 /home/*loginname*
- 新用户的账户密码在过期后不会被禁用
- 新用户账户不设置过期日期
- 新用户账户将 /bin/sh 指向的shell作为默认shell
- 系统会将 /etc/skel 目录的内容复制到用户的 $HOME 目录
- 系统不会为该用户账户在mail目录下创建一个用于接收邮件的文件

`useradd` 命令允许管理员创建默认的$HOME目录配置，然后将其作为创建新用户$HOME目录的模板。这样就能自动在每个新用户的$HOME目录里放置默认的系统文件了。在Ubuntu系统中，/etc/skel目录包含以下文件：

```
lxc@Lxc:~$ ls -al /etc/skel
总用量 32
drwxr-xr-x   3 root root  4096 10月 30 11:08 .
drwxr-xr-x 150 root root 12288 11月 25 21:22 ..
drwxr-xr-x   2 root root  4096 10月 18 20:50 模板
-rw-r--r--   1 root root   220 2月  25  2020 .bash_logout
-rw-r--r--   1 root root  3771 2月  25  2020 .bashrc
-rw-r--r--   1 root root   807 2月  25  2020 .profile
```

根据第6章的内容，你应该知道这些文件是做什么的。它们是bash shell环境的标准启动文件。系统自动将这些默认文件复制到你创建的每个用户的$HOME目录。

要想在创建新用户时改变默认行为，可以使用相应的命令行选项，如表7-1所示。

4. 删除用户

如果想从系统中删除账户，`userdel` 命令可以满足这个需求。在默认情况下，`userdel` 命令只删除/etc/passwd和/etc/shadow文件中的用户信息，属于该账户的文件会被保留。
如果加入 `-r` 选项，则 `userdel` 命令会删除用户的$HOME目录以及邮件目录。然而，系统中仍可能存有已删除用户的其他文件。这在有些环境中会造成问题。

```
root@Lxc:/home/lxc/scripts/ch07# userdel -r test1
userdel: test1 邮件池 (/var/mail/test1) 未找到
# 因为注册test1账户时并没有创建mail目录，所以在删除时，其邮件池也未找到。
```

> **警告：** 在有大量用户的环境中使用 `-r` 选项要特别小心。你永远不知道用户是否在个人的$HOME目录中存放了其他用户或程序要用到的重要文件。在删除用户的$HOME目录之前一定要检查清楚。

5. 修改用户

每种工具都提供了特定的功能来修改用户账户信息。接下来将介绍这些工具。

1. `usermod`

`usermod` 命令是用户账户修改工具中最强大的一个，提供了修改/etc/passwd文件中大部分字段的相关选项，只需指定相应的选项即可。大部分选项与 `useradd` 命令选项一样（比如，`-c` 用于修改备注字段，`-e` 修改过期日期，`-g` 修改默认的登录组）。除此之外，还有另外一些也许能派上用场的选项。

- -l： 修改用户帐户的登录名
- -L： 锁定账户，禁止用户登录。
- -p： 修改账户密码
- -U： 解除锁定，恢复用户登录

`-L` 选项尤为实用。该选项可以锁定账户，使用户无法登录，无须删除账户和用户数据。要恢复账户，只需使用 `-U` 选项即可。

2. `passwd` 和 `chpasswd`

`passwd` 命令可以方便的修改用户密码：

如果只使用 `passwd` 命令，则修改的是你自己的密码。系统中的任何用户都能修改自己的密码，但只有root用户才有权限修改别人的密码。
`-e` 选项可以强制用户下次登陆时修改密码。你可以先给用户设置一个简单的密码，之后强制用户在下次登陆时改成他们的密码。

如果需要为系统中的大量用户修改密码，那么 `chpasswd` 命令可以助你事半功倍。`chpasswd` 命令可以从标准输入自动读取一系列以冒号分隔的登录名和密码对偶（login name and password pair），自动对密码加密，然后为用户帐户设置密码。你也可以重定向命令将包含 *username:password* 对偶的文件重定向给该命令。

```
root@Lxc:/home/lxc/scripts/ch07# chpasswd < users.txt
root@Lxc:/home/lxc/scripts/ch07#
```

3. `chsh`、`chfn` 和 `chage`

2. 使用Linux组

组权限允许多个用户对系统对象（比如文件、目录或设备等）共享一组权限。
Linux发行版在处理默认组的成员关系时略有差异。有些Linux发行版会创建一个组，将所有用户都作为该组成员。遇到这种情况要特别小心，因为你的文件有可能对于其他用户也是可读的。有些发行版会为每个用户创建一个单独的组，这样会更安全一点。
每个组都有唯一的GID，和UID类似，该值在系统中是唯一的。除了GID，每个组还有一个唯一的组名。
Linux系统中有一些组工具可用于创建和管理组。

3. /etc/group 文件

与用户账户类似，组信息也保存在一个文件中。/etc/group 文件包含系统中每个组的信息。

```
lxc@Lxc:~/scripts/ch07$ cat /etc/group
root:x:0:
daemon:x:1:
bin:x:2:
......
```

/etc/group 文件的4个字段分别是：

- 组名
- 组密码
- GID
- 属于该组的用户列表

与UID类似，GID在分配时也采用了特定的格式。对于系统账户组，为其分配的GID低于500，而普通用户组的GID则从500开始分配（有的发行版是1000）。
组密码允许非组内成员使用密码临时性的成为该组成员。这个功能用的不多，但确实存在。
同样，/etc/group 文件是一个文本文件。手动编辑出现错误时可能会损坏文件，引发系统故障。更安全的做法是使用 `usermod` 命令（参见7.1.5节）。

> **注意：** 用户帐户列表多少有些误导人。你会发现列表中的有一些组没有任何用户。这并不是说这些组没有成员。当一个用户在 /etc/passwd 文件中指定某个组作为主要组时，该用户不会作为该组成员再出现在 /etc/group 文件中。多年了被这个问题困扰的系统管理员可不止一两个。

2. 创建新组

3. 修改组

3. 理解文件权限

`ls` 命令可以查看Linux系统中的文件、目录和设备的权限。

```
lxc@Lxc:~/scripts/ch07$ ls -l
总用量 680
-rw-rw-r-- 1 lxc lxc  91572 11月 27 14:34 chage.png
-rw-rw-r-- 1 lxc lxc  12099 11月 27 15:08 README.md
......
```

- - 代表文件
- d 代表目录
- l 代表链接
- c 代表字符设备
- b 代表块设备
- p 代表具名管道
- s 代表网络套接字

之后的三组三字符的编码。每一组定义了3种访问权限。

- r 代表对象是可读的
- w 代表对象是可写的
- x 代表对象是可执行的

如果没有某种权限，则在该权限位会出现连字符。
这三组权限分别对应对象的三个安全级别。

- 对象的属主
- 对象的属组
- 系统的其他用户

2. 默认文件权限

`umask` 命令用来设置新建文件和目录的默认权限。`umask` 命令可以显示和设置默认权限：

```
lxc@Lxc:~/scripts/ch07$ umask
0022
```

其中四位数字中的第一个数位代表了一项特别的安全特性，在 7.5节 会详述。接下来三个数位表示文件或目录对应的umask八进制值。比如664代表属主和属组成员都有读取和写入的权限。而其他用户只有读取的权限。
umask只是个掩码，它会屏蔽掉不想授予该安全级别的权限。要把umask值从对象的全权限值减掉。剩下的是对象的权限。对文件而言，全权限值是666（所有用户都有读取和写入的权限）；对目录而言全权限值是777（所有用户都有读取、写入和执行的权限，对于目录而言，目录的执行权限就是进入该目录）。 所以，在上面的例子中，文件一开始的权限是666，减去掩码umask的值022之后，剩下的文件权限就变成了664。

umask的值通常会被设置在 /etc/profile 文件中，你可以使用 umask 命令指定其他的umask默认值。当然，该指定在shell退出后就会失效，你可以更改 /etc/profile 文件中的值来持久化。

```
lxc@Lxc:~/scripts/ch07$ umask
0002
lxc@Lxc:~/scripts/ch07$ umask 026
lxc@Lxc:~/scripts/ch07$ touch newfile1
lxc@Lxc:~/scripts/ch07$ ls -l newfile1
-rw-r----- 1 lxc lxc 0 11月 27 15:31 newfile1
lxc@Lxc:~/scripts/ch07$ umask
0026
lxc@Lxc:~/scripts/ch07$ mkdir newdir
lxc@Lxc:~/scripts/ch07$ ls -ld newdir/
drwxr-x--x 2 lxc lxc 4096 11月 27 15:35 newdir/
```

4. 更改安全设置

本节将介绍如何修改文件和目录的已有权限、默认属主以及默认属组。

1. 修改权限

`chmod` 命令可以修改文件和目录的安全设置。该命令的格式如下：

```
chmod options mode files
```

mode 参数允许使用八进制模式或符号模式来进行安全设置。
八进制模式设置非常直观，直接用打算赋予文件的标准3位八进制权限编码即可。

```
lxc@Lxc:~/scripts/ch07$ chmod 760 newfile
lxc@Lxc:~/scripts/ch07$ ls -l newfile
-rwxrw---- 1 lxc lxc 0 11月 27 15:51 newfile
```

符号模式指定权限的格式如下：

```
[ugoa...][[+-=][rwxXstugo...]]
```

第一组字符指定了定义权限作用的对象。

- u 代表用户
- g 代表组
- o 代表其他用户
- a 代表上述所有

接下来的符号表示你想在现有的权限基础上：增加权限（+）、移除权限（-），还是设置权限（=）。
最后，第三个符号代表要设置的权限。你会发现，可取的值要比通常的rwx多。这些额外值如下。

- X 仅当对象是目录或者已有执行权限时才赋予执行权限
- s 在执行时设置SUID或SGID
- t 设置粘滞位(sticky bit)
- u 设置属主权限
- g 设置属组权限
- o 设置其他用户权限

来个例子：

```
lxc@Lxc:~/scripts/ch07$ chmod o+r newfile
lxc@Lxc:~/scripts/ch07$ ls -l newfile
-rwxrw-r-- 1 lxc lxc 0 11月 27 15:51 newfile
lxc@Lxc:~/scripts/ch07$ chmod u-x newfile
lxc@Lxc:~/scripts/ch07$ ls -l newfile
-rw-rw-r-- 1 lxc lxc 0 11月 27 15:51 newfile
```

options 为 `chmod` 命令提供了额外的增强特性。`-R` 选项能够以递归的方式修改文件或目录的权限。你可以使用通配符指定多个文件名，然后用单个命令批量修改权限。

2. 改变所属关系

`chown` 命令用来修改文件的属主; `chgrp` 命令用来修改文件的默认属组。

`chown` 的命令格式如下：

```
chown options owner[.group] file
```

可以使用登录名或UID来指定文件的新属主：

```
root@Lxc:/home/lxc/scripts/ch07# ls -l newfile
-rw-rw-r-- 1 lxc lxc 0 11月 27 15:51 newfile
root@Lxc:/home/lxc/scripts/ch07# chown test newfile
root@Lxc:/home/lxc/scripts/ch07# ls -l newfile
-rw-rw-r-- 1 test lxc 0 11月 27 15:51 newfile
```

也支持同时修改文件的属主和属组：

```
root@Lxc:/home/lxc/scripts/ch07# chown lxc.test newfile
root@Lxc:/home/lxc/scripts/ch07# ls -l newfile
-rw-rw-r-- 1 lxc test 0 11月 27 15:51 newfile
```

也可以只修改文件的默认属组：

```
root@Lxc:/home/lxc/scripts/ch07# chown .lxc newfile
root@Lxc:/home/lxc/scripts/ch07# ls -l newfile
-rw-rw-r-- 1 lxc lxc 0 11月 27 15:51 newfile
```

最后，如果你的Linux系统使用与用户登录名相同的组名，则可以同时修改二者：

```
root@Lxc:/home/lxc/scripts/ch07# chown test.test newfile
root@Lxc:/home/lxc/scripts/ch07# ls -l newfile
-rw-rw-r-- 1 test test 0 11月 27 15:51 newfile
root@Lxc:/home/lxc/scripts/ch07# chown lxc. newfile
root@Lxc:/home/lxc/scripts/ch07# ls -l newfile
-rw-rw-r-- 1 lxc lxc 0 11月 27 15:51 newfile
```

> **注意：** 只有root用户能修改文件的属主。任何用户都能修改文件的属组，但前提是该用户必须是原属组和新属组的成员。

`chgrp` 命令可以方便的修改文件或目录的默认属组：

```
root@Lxc:/home/lxc/scripts/ch07# chgrp test newfile
root@Lxc:/home/lxc/scripts/ch07# ls -l newfile
-rw-rw-r-- 1 lxc test 0 11月 27 15:51 newfile
```

5. 共享文件

Linux系统中共享文件的方法是创建组。
Linux系统为每个文件和目录存储了3个额外的信息位。

- SUID(set user ID)：当用户执行此文件时，程序会以文件属主的权限运行。
- SGID(set group ID)：对文件而言，程序会以文件属组的权限运行；对目录而言，该目录中创建的新文件会以目录的属组作为默认属组。
- 粘滞位(sticky bit)：应用于目录时，只有文件属主可以删除或重命名该目录中的文件。

SGID位对文件共享非常重要。启用SGID位后，可以强制在共享目录中创建的新文件都属于该目录的属组，这个组也就成了每个用户的属组。
可以通过 `chmod` 命令设置 SGID，将其添加到标准3位八进制之前（组成4位八进制值），或者在符号模式下使用符号 `s`。

如果使用的是八进制模式，则需要知道这些位的排列。自左向右依次为：SUID位、SGID位、粘滞位。如下表所示：

因此，要创建一个共享目录，使目录中的所有新文件都沿用目录的属组，只需设置该目录的SGID位。

```
lxc@Lxc:~/scripts/ch07$ mkdir testdir
lxc@Lxc:~/scripts/ch07$ ls -ld testdir/
drwxr-xr-x 2 lxc lxc 4096 11月 27 17:01 testdir/
root@Lxc:/home/lxc/scripts/ch07# chgrp shared testdir/
root@Lxc:/home/lxc/scripts/ch07# chmod g+s testdir/
root@Lxc:/home/lxc/scripts/ch07# ls -ld testdir/
drwxr-sr-x 2 lxc shared 4096 11月 27 17:01 testdir/
root@Lxc:/home/lxc/scripts/ch07# umask 002
root@Lxc:/home/lxc/scripts/ch07# cd testdir/
root@Lxc:/home/lxc/scripts/ch07/testdir# touch testfile
root@Lxc:/home/lxc/scripts/ch07/testdir# ls -l
总用量 0
-rw-rw-r-- 1 root shared 0 11月 27 17:03 testfile
```

首先，使用 `mkdir` 命令创建希望共享的目录。然后，通过 `chgrp` 命令修改目录的默认属组，使其包含所有需要共享文件的用户。最后设置目录的SGID位，保证目录中的新建文件都以 *shared* 作为默认属组。为了让这个环境正常工作，所有组成员都要设置他们的umask值。使文件对属组成员可写。在这个例子中，umask被改成了002，所以文件对属组是可写的。
完成这些步骤之后，组成员都能在共享目录下创建新文件了。跟期望的一样，新文件会沿用目录的默认属组，而不是账户的默认属组。现在 *shared* 组的所有用户都能访问这个文件了。

6. 访问控制列表

Linux的基本权限方法有一个缺点：局限性。你只能将文件或目录的权限分配给单个组或用户账户。在一个复杂的商业环境中，对于文件和目录，不同的组需要不同的权限，基本的权限方法解决不了这个问题。

Linux开发者设计出了一种更为先进的目录和文件的安全方法：**访问控制列表（access control list, ACL）**。ACL允许指定包含多个用户或组的列表以及为其分配的权限。和基本安全方法一样，ACL权限使用相同的读取、写入和执行权限位，但现在可以分配给多个用户和组。

可以使用 `setfacl` 命令和 `getfacl` 命令在Linux系统中实现ACL特性。`getfacl` 命令能够查看分配给文件或目录的ACL：

```
lxc@Lxc:~/scripts/ch07$ touch test
lxc@Lxc:~/scripts/ch07$ ls -l test
-rw-r--r-- 1 lxc lxc 0 11月 27 17:57 test
lxc@Lxc:~/scripts/ch07$ getfacl test
# file: test
# owner: lxc
# group: lxc
user::rw-
group::r--
other::r--
```

如果只为文件分配了基本的安全权限，则这些权限就会像上面例子所显示的那样，出现在 `getfacl` 命令的输出中。

`setfacl` 命令可以为用户或组分配权限：

```
setfacl [options] rule filenames
```

`setfacl` 命令可以使用 `-m` 选项修改分配给文件或目录的权限，或使用 `-x` 选项删除特定的权限。可以使用以下3种格式定义 *rule*：

```
u[ser]:uid:perms
g[roup]:gid:perms
o[ther]::perms
```

要为用户分配权限，可以使用 user 格式；要为组分配权限，可以使用 group 格式；要为其他用户分配权限，可以使用 other 格式。对于 uid 或 gid，可以使用数字值或名称。

```
lxc@Lxc:~/scripts/ch07$ touch test
lxc@Lxc:~/scripts/ch07$ sudo groupadd tg
lxc@Lxc:~/scripts/ch07$ setfacl -m g:tg:rw test
lxc@Lxc:~/scripts/ch07$ ls -l test
-rw-rw-r--+ 1 lxc lxc 0 11月 27 18:09 test
```

这个例子为 test 文件添加了 tg 组的读写权限。注意，`setfacl` 命令不产生输出。在列出文件时，只显示标准的属主、属组和其它用户权限，但在权限列末尾多了一个加号（+），指明该文件还应用了ACL。可以再次使用 `getfacl` 命令查看ACL：

```
lxc@Lxc:~/scripts/ch07$ getfacl test
# file: test
# owner: lxc
# group: lxc
user::rw-
group::r--
group:tg:rw-
mask::rw-
other::r--
```

`getfacl` 的输出显示为两个组分配了权限。默认组(lxc)对文件有读权限，tg 组对文件有读写权限。要想删除权限，可以使用 `-x` 选项：

```
lxc@Lxc:~/scripts/ch07$ setfacl -x g:tg test
lxc@Lxc:~/scripts/ch07$ getfacl test
# file: test
# owner: lxc
# group: lxc
user::rw-
group::r--
mask::r--
other::r--
```

Linux也允许对目录设置默认ACL，在该目录中创建的文件会自动继承。这个特性称为ACL继承。
要想设置目录的默认ACL，可以在正常的规则定义前加上 `d:`，如下所示：

```
lxc@Lxc:~/scripts/ch07$ mkdir tt
lxc@Lxc:~/scripts/ch07$ sudo setfacl -m d:g:tg:rw tt
lxc@Lxc:~/scripts/ch07$ getfacl tt/
# file: tt/
# owner: lxc
# group: lxc
user::rwx
group::r-x
other::r-x
default:user::rwx
default:group::r-x
default:group:tg:rw-
default:mask::rwx
default:other::r-x

lxc@Lxc:~/scripts/ch07$ cd tt/
lxc@Lxc:~/scripts/ch07/tt$ touch 1.txt
lxc@Lxc:~/scripts/ch07/tt$ getfacl 1.txt
# file: 1.txt
# owner: lxc
# group: lxc
user::rw-
group::r-x                      #effective:r--
group:tg:rw-
mask::rw-
other::r--
```

这个例子为 tt 目录添加了 tg 组的读写权限。在该目录中创建的文件都会自动为 tg 组分配读写权限。

ch11 构建基础脚本

11.1 使用多个命令

如果想让两个命令一起运行，可以将其放在同一行，彼此用分号隔开:

```
date ; who
```

可以将多个命令串在一起使用，只要不超过命令行最大字符数 **255** 就行，shell会按顺序逐个执行。其实就是命令列表，参见 第5章。

11.2 创建shell脚本文件

在创建shell时，必须在第一行指定要使用的shell，格式如下:

```
#!/bin/bash
```

在普通的shell脚本中，`#`用作注释行。shell并不会处理shell脚本中的注释行。然而，shell脚本的第一行是个例外，`#`号后面的`!`用于告诉shell用哪个shell来运行脚本。（可以使用bash shell，然后使用另一个shell来运行你的脚本。）

11.3 显示消息

大多数shell命令会产生自己的输出，这些输出会显示在脚本运行的控制台显示器上。
你可以通过 `echo` 命令来显示自己添加的消息。`echo` 命令可用单引号或者双引号来划定字符串。如果你在字符串中用到某种引号，可以使用另一种引号来划定字符串。

例如:

```
lxc@Lxc:~/scripts/ch11$ echo 'lxc says "scripting is easy"'
lxc says "scripting is easy"
```

> 使用 `echo` 命令时，若不想输出尾随换行符，可以使用 `-n` 选项。

11.4 使用变量

1. *环境变量*

shell维护着一组用于记录特定系统信息的环境变量，比如系统名称、已登陆系统的用户名、用户的系统ID（UID）、用户的默认主目录以及shell查找程序的搜索路径等。
你可以使用 `set` 命令显示一份完整的当前环境变量列表。参见 第6章

在脚本中，可以在环境变量名之前加上 `$` 符号来引用这些环境变量。参见 test2

如果你想在双引号中原生的显示 `$` 符号，必须在它前面放置一个反斜线:

```
echo "The cost of the item is \$15"
# output: The cost of the item is $15
```

> 也可以使用 `${variavle}` 形式引用变量。花括号通常用于帮助界定`$`后面的变量名.

2. *用户自定义变量*

- 用户自定义变量的名称可以是由任意 **字母、数字、下划线** 组成的字符串，长度不超过 **20** 个字符，区分大小写。
- 注意：使用等号为变量赋值，在变量、等号和值之间 **不能** 出现空格。
- shell脚本会以 **字符串** 的形式存储所有的变量值，脚本中的各个命令自行决定变量值的数据类型。
- shell脚本中定义的变量在脚本的整个生命周期里会一直保持它们的值，在脚本结束时被删除。

test3 test4

3. *命令替换*

有两种方法可以将命令输出赋给变量。

- 反引号 `
- `$()` 格式

例如：

```
testing=`date`
```

或者

```
testing=$(date)
```

test5

> **警告：** 命令替换会创建出子shell来运行指定命令，这是由运行脚本的shell所生成的一个独立的shell。**因此，在子shell中运行的命令无法使用脚本中的变量**。 如果在命令行中使用`./`路径执行命令，就会创建子shell，但如果不加路径，则不会创建子shell。不过，内建的shell命令也不会创建子shell。

11.5 重定向输入输出

1. 输出重定向

命令格式：

```
command > outputfile
```

例如：

```
date > test6.txt
```

最基本的重定向会将命令输出发送至文件。bash shell使用 `>` 来实现该操作（若文件已存在，则覆盖）。
若你不想覆盖文件原有内容，可以使用 `>>` 来将命令输出追加到已有文件中。

2. 输入重定向

输入重定向将文件内容重定向至命令，使用 `<` 实现。

命令格式：

```
command < inputfile
```

例如：

```
wc < test6.txt
```

内联输入重定向（inline input redirection） 使用 `<<` 号，无须使用文件进行重定向，只需在命令行中指定用于输入重定向的数据即可。必须指定一个文本标记，任何字符串都可以作为文本标记，文本标记用来划分输入数据的起止(起止的文本标记必须一致)：

命令格式：

```
command << marker
data
data
marker
```

例如：

```
wc << EOF
> test string 1
> test string 2
> test string 3
> EOF

# output 3 9 42
```

次提示符（由 `PS2` 环境变量定义）会持续显示，直到输入了作为文本标记的那个字符串。`wc` 命令统计了数据的 行数、单词数、字节数。

11.6 管道

管道 可以将一个命令的输出作为另一个命令的输入，管道可以串联的命令数量没有限制。

命令格式：

```
command1 | command2
```

> **不要认为管道串联起来的命令会依次执行，Linux系统会同时运行这两个命令**，当第一个命令产生输出时，
> 它会被立即传给第二个命令。数据传输不会用到任何中间文件或者缓冲区。

有些命令的输出会在屏幕上一闪而过，我们可以使用管道将其输出传给文本分页命令（`more` 或者 `less`），来强行将输出按屏显示。

11.7 执行数学运算

shell脚本在执行数学运算这方面多少有点不尽如人意。在shell脚本中执行数学运算有两种方式。

1. expr 命令

~~该命令十分笨拙~~，可以识别少量算术运算符和字符串运算符，如下表所示：

运算符	描述
ARG1 | ARG2	如果 ARG1 既不为null也不为0，就返回 ARG1 ； 否则，返回 ARG2
ARG1 & ARG2	如果 ARG1 和 ARG2 都不为null或者0，就返回 ARG1；否则返回 0
ARG1 < ARG2	如果 ARG1 小于 ARG2，返回1；否则，返回0
ARG1 <= Arg2	如果 ARG1 小于或者等于 ARG2，返回1；否则，返回0
ARG1 = ARG2	如果 ARG1 等于 ARG2，返回1；否则，返回0
ARG1 != ARG2	如果 ARG1 不等于 ARG2，返回1；否则，返回0
ARG1 >= ARG2	如果 ARG1 大于或者等于 ARG2，返回1；否则，返回0
ARG1 > ARG2	如果 ARG1 大于 ARG2，返回1；否则，返回0
ARG1 + ARG2	返回 ARG1 和 ARG2 之和
ARG1 - ARG2	返回 ARG1 和 ARG2 之差
ARG1 * ARG2	返回 ARG1 和 ARG2 之积
ARG1 / ARG2	返回 ARG1 和 ARG2 之商
ARG1 % ARG2	返回 ARG1 和 ARG2 之余数
STRING : REGEXP	如果 REGEXP 模式匹配 STRING ，则返回该模式匹配的内容
match STRING REGEXP	如果 REGEXP 模式匹配 STRING ，则返回该模式匹配的内容
substr STRING POS LENGTH	返回起始位置为POS（从1开始计数）、长度为LENGTH的子串
index STRING CHARS	返回 CHARS 在字符串 STRING 中所处的位置；否则，返回0
length STRING	返回字符串 STRING 的长度
+ TOKEN	将 TOKEN 解释成字符串，即使 TOKEN 属于关键字
(EXPRESSION)	返回 EXPRESSION 的值

> 许多 *expr* 命令运算符在shell中另有他意（比如 `*`）。当这些符号出现在 *expr* 命令中时，会产生诡异的结果，**所以，一些符号需要转义**。

```
expr 5 * 2
# output: syntax error
```

正确的应为:

```
expr 5 \* 2
# output 10
```

确实难看，在shell脚本中使用 `expr` 命令也同样麻烦。test6
所以我们使用下面的方法。

2. *使用方括号*

- 为了兼容Bourne shell，所以bash shell 保留了 *expr* 命令，同时也提供了更方便的方式来执行数学运算。
- 在bash中，要将数学运算的结果赋给变量，可以使用 `$` 和 `[]`，就像这样: `$[ operation ]`。在使用方括号时，无须担心shell会误解乘号或其他符号。

例如:

test7.sh

```
#!/bin/bash

var1=100
var2=50
var3=45
var4=$[$var1 * ($var2 - $var3)]
echo "The final result is $var4"
```

> bash shell的数学运算符只支持整数运算，z shell(zsh)提供了完整的浮点数操作。如果需要在shell脚本中执行浮点数运算，那么不妨考虑一下z shell（参见 第23章）。

3. *浮点数解决方案*

可以使用内建的bash计算器 `bc`

1.bc的基本用法

`bc` 计算器实际上是一种编程语言，允许在命令行输入浮点数表达式，然后解释并计算该表达式并返回结果。
`bc` 计算器能够识别以下内容。

- 数字（整数和浮点数）
- 变量（简单变量和数组）
- 注释（以 `#` 或C语言中的 `/**/` 开始的行）
- 表达式
- 编程语句（比如 *if-then* 语句）
- 函数

你可以在shell提示符下通过 `bc` 命令访问bash计算器，输入 `quit` 退出。
`scale` 变量控制 `bc` 输出结果保留的位数，默认为0。

```
lxc@Lxc:~/scripts/ch11$ bc
bc 1.07.1
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006, 2008, 2012-2017 Free
Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
12 * 5.4
64.8
3.156 * (3 + 5)
25.248
quit
```

浮点数运算由内建变量 `scale` 控制，默认值为0。你必须将该变量的值设置为希望在结果中保留的小数位数。

```
lxc@Lxc:~/scripts/ch11$ bc -q
3.44 / 5
0
scale=4
3.44/5
.6880
quit
```

`-q` 选项可以不显示bash计算器冗长的欢迎信息。

除了普通数字，bash计算器还支持变量：

```
lxc@Lxc:~/scripts/ch11$ bc -q
var1=10
var1 * 4
40
var2 = var1 / 5
print var2
2
quit
```

2.在脚本中使用bc

可以使用 *命令替换* 将 `bc` 命令的输出赋给变量。

命令格式：

```
variable=$(echo "options; expression" | bc)
```

例如：

test9.sh

```
#!/bin/bash
var1=$(echo "scale=4; 3.44 / 5" | bc)
echo "The answer is $var1"
# output:
lxc@Lxc:~/scripts/ch11$ ./test9
The answer is .6880
```

test10 test11

这种方法适用于较短的运算，但有时你需要涉及更多的数字。如果要进行大量运算，那么在一个命令行中列出多个表达式容易让人犯晕。最好的方法是使用内联输入重定向。

test12.sh

```
#!/bin/bash
var1=10.46
var2=43.67
var3=33.2
var4=71

var5=$(bc << EOF
scale=4
a1 = $var1 * $var2
a2 = $var3 * $var4
a1 + b1
EOF
)
echo The final answer for this mess is $var5
```

还要注意到，在这个例子中，可以在bash计算器中为变量赋值，但是在bash计算器中创建的变量仅在计算器中有效，不能在shell脚本中使用。

11.8 退出脚本

shell中运行的每个命令都使用 **退出状态码** 来告诉shell自己已经运行完毕。
退出状态码是一个 **0~255** 的整数值，在命令结束运行时传给shell，你可以获取这个值并在脚本中使用。

1. 查看退出状态码

Linux提供了专门的变量 `$?` 来保存最后一个已执行命令的退出状态码。按照惯例，成功结束的命令，其退出状态码为0。因错误而结束的命令，其退出状态码为正整数。
Linux错误退出状态码没有什么标准可循。但有一些可用的指南，如下：

状态码	描述
0	命令成功结束
1	一般性未知错误(如命令参数错误)
2	不适合的shell命令
126	命令无法执行(没有权限)
127	没找到命令(你打错字了)

状态码	描述
128	无效的退出参数
128+*x*	与Linux信号*x*相关的严重错误
130	通过 `Ctrl+C` 终止的命令
255	正常范围之外的退出状态码

2. exit 命令

在默认情况下，shell脚本会以脚本中的最后一个命令的退出状态码退出。
你可以改变这种行默认为，返回自己的退出状态码。 `exit` 命令允许在脚本结束时指定一个退出状态码:

test13.sh

```
#!/bin/bash
# testing the exit status
var1=10
var2=30
var3=$[ $var1 + $var2 ]
echo The final result is $var3
exit 5
# output:
lxc@Lxc:~/scripts/ch11$ ./test13
The final result is 40
lxc@Lxc:~/scripts/ch11$ echo $?
5
```

也可以使用变量作为 `exit` 命令的参数:

test14

```
#!/bin/bash
# testing the exit status
var1=10
var2=30
var3=$[ $var1 + $var2 ]
echo The final result is $var3
exit $var3
# output:
lxc@Lxc:~/scripts/ch11$ ./test14
The final result is 40
lxc@Lxc:~/scripts/ch11$ echo $?
40
```

> 注意：退出状态码最大为 255 ， 若大于该值，则为除以256的余数。

test14b.sh

```
#!/bin/bash
# testing the exit status
var1=10
var2=30
var3=$[ $var1 * $var2 ]
echo The final result is $var3
exit $var3 # 300 % 256 = 44
# output:
lxc@Lxc:~/scripts/ch11$ ./test14b
The final result is 300
lxc@Lxc:~/scripts/ch11$ echo $?
44
```

9. 实战演练

我们搞一个脚本来计算两个日期之间相隔的天数：

mydate.sh

```
#!/bin/bash
# calculate the number of days between two dates
date1="Jan 1, 2020"
date2="May 1, 2020"

time1=$(date -d "$date1" +%s)
time2=$(date -d "$date2" +%s)
diff=$(expr $time2 - $time1)
secondsinday=$(expr 24 \* 60 \* 60)
days=$(expr $diff \/ $secondsinday)

echo "The difference in $date1 and $date2 is $days days"
```

`date` 命令允许使用 `-d` 选项指定特定日期（以任意格式）。为了执行日期运算，要利用到 **纪元时间（epoch time）** 这个Linux特性。纪元时间将时间指定为自1970年1月1日后的整秒数。`date` 命令的 `%s` 格式：seconds since 1970-01-01 00:00:00 UTC。得到两个日期的纪元时间，相减，除以一天中的秒数，以获得两个日期之间的天数差异。

ch12 结构化命令

有一类命令允许脚本根据条件跳过部分命令，改变执行流程。这样的命令通常称为 **结构化命令（structured command）**。

12.1 使用 *if-then* 语句

语句格式:

```
if command
then
    commands
fi
```

test1.sh test2.sh test3.sh

bash shell的 `if` 语句会运行 `if` 之后的命令。如果该命令的退出状态码为0（命令成功运行），那么位于 `then` 部分的命令就会被执行。若为其它值，则不会被执行。

注意，*`if-then`* 语句还有另一种形式：

```
if command; then
    commands
fi
```

通过将分号写在待求值的命令尾部，可以将 *`then`* 语句写在同一行，这样看起来更像其他编程语言中的 *`if-then`* 语句。

12.2 *if-then-else* 语句

当 *`if`* 语句中的命令返回退出状态码为0时，*`then`* 部分中的命令会被执行，这跟普通的 *`if-then`* 语句一样。当 *`if`* 语句中的命令返回非0的退出状态码时，bash shell会执行 *`else`* 部分的命令。

语句格式:

```
if command
then
    commands
else
    commands
fi
```

test4.sh

12.3 嵌套 *if* 语句

语句格式:

```
if command1
then
    command set 1
elif command2
then
    command set 2
elif comman3
then
    command set 3
fi
```

test5.sh

12.4 test 命令

`if-then` 语句不能测试命令退出状态码之外的条件。
我们可以使用 `test` 测试不同条件，如果 `test` 命令中列出的条件成立，那么 `test` 命令就会退出并返回退出状态码 0 。这样 `if-then` 语句的工作方式就和其他编程语言中的 `if-then` 语句差不多了。

`test` *命令格式:*

```
test condition
```

当用在 `if-then` 语句中时：

语句格式:

```
if test condition
then
    commands
fi
```

例如:

test6.sh test6b.sh

```
my_variable="Full"
# 这里my_variable变量非空，所以test会返回0。否则，则会执行else分支。
if test $my_variable
then
    echo "The my_variable variable has content and returns a True."
    echo "The my_variable variable content is $my_variable."
else
    echo "The my_variable variable doesn't have content,"
    echo "and return a False."
fi
```

bash shell中提供了另一种 *条件测试* 方式，无须在 `if-then` 语句中写明 `test` 命令：

```
if [ condition ]
then
    commands
fi
```

> **注意:** 第一个方括号之后和第二个方括号之前**必须留有空格**，否则会出错.

`test` 命令和 *测试条件* 可以判断以下3类条件

1. 数值比较

比较	描述
n1 -eq *n2*	检查 *n1* 是否等于 *n2*
n1 -ge *n2*	检查 *n1* 是否大于或等于 *n2*
n1 -gt *n2*	检查 *n1* 是否大于 *n2*
n1 -le *n2*	检查 *n1* 是否小于或等于 *n2*
n1 -lt *n2*	检查 *n1* 是否小于 *n2*
n1 -ne *n2*	检查 *n1* 是否不等于 *n2*

numberic_test.sh

```bash
#!/bin/bash
# using numeric test evaluations

value1=10
value2=11
#
if [ $value1 -gt 5 ]
then
    echo "The test value $value1 is greater than 5."
fi

if [ $value1 -eq $value2 ]
then
    echo "The value are equal."
else
    echo "The value are different."
fi
```

> **注意：** 对于条件测试。bash shell只能处理整数，尽管可以将浮点值用于某些命令(比如 `echo`)，但它们在条件测试下无法正常工作。

2. 字符串比较

比较	描述
str1 = *str2*	检查 *str1* 是否和 *str2*相同
str1 != *str2*	检查 *str1* 是否和 *str2*不同
str1 < *str2*	检查 *str1* 是否小于 *str2*
str1 > *str2*	检查 *str1* 是否大于 *str2*
-n *str1*	检查 *str1* 的长度是否不为 0
-z *str1*	检查 *str1* 的长度是否为 0

1. 字符串相等性比较

string_test.sh

```bash
#!/bin/bash
# using string test evaluations
#
testUser=lxc
#
if [ $testUser = lxc ]
then
    echo "The testUser variable contains: lxc"
else
    echo "ths testUser variable contains: $testUser"
fi
```

2. 字符串顺序

使用条件测试的大于或小于功能时，会出现两个经常困扰shell程序员的问题。

- 在 *条件测试* 中大于号和小于号 **必须** 转义，否则会被shell视为重定向符，将字符串当作文件名。
- 大于和小于顺序和 `sort` 命令采用的比较不同

`sort` 命令使用的是系统语言环境设置中定义的排序顺序。比较测试中使用的是标准的Unicode顺序，根据每个字符的Unicode编码值来决定排序结果.

good_string_comparison.sh

```
#!/bin/bash
# Properly using string comparisons
#
string1=soccer
string2=zorbfootball
#
if [ $string1 \> $string2 ]
then
     echo "$string1 is greater than $string2"
else
     echo "$string1 is less than $string2"
fi
```

第二个问题更细微。`sort` 命令处理大写字母的方法刚好与 `test` 命令相反。

sort_order_comparison.sh

```
lxc@Lxc:~/scripts/ch12$ cat SportsFile.txt
Soccer
soccer
lxc@Lxc:~/scripts/ch12$ sort SportsFile.txt
soccer
Soccer
lxc@Lxc:~/scripts/ch12$ ./sort_order_comparison.sh
Soccer <= soccer.
lxc@Lxc:~/scripts/ch12$ cat sort_order_comparison.sh
#!/bin/bash
# Testing string sort order
#
string1=Soccer
string2=soccer
#
if [ $string1 \> $string2 ]
then
    echo "$string1 > $string2."
else
    echo "$string1 <= $string2."
fi
# Shell 使用Unicode编码值比较大小
```

在脚本的比较测试中大写字母被认为是小于小写字母的。但 `sort` 命令正好相反。这是由于各个命令使用了不同的排序技术。
比较测试中（即脚本中的条件测试）使用的是标准的Unicode顺序，根据每个字符的Unicode编码值来决定排序结果。`sort` 命令使用的系统的语言环境设置中定义的顺序。对于英语，语言环境设置指定了在排序顺序中小写字母出现在大写字母之前。

3. 字符串大小

`-n` 和 `-z` 可以很方便地用于检查一个变量是否为空：

variable_content_eval.sh

```
#!/bin/bash
# Testing string length
# -n 长度是否不为0、-z 长度是否为0
string1="soccer"
string2=''

if [ -n $string1 ]
then
    echo "The string '$string1' is NOT empty."
else
    echo "The string '$string1' IS empty."
fi

if [ -z $string2 ]
then
    echo "The string '$string2' IS empty."
else
    echo "The string '$string2' is NOT empty"
fi

if [ -z $string3 ]
then
    echo "The string '$string2' IS empty."
else
    echo "The string '$string2' is NOT empty"
fi
# output:
lxc@Lxc:~/scripts/ch12$ ./variable_content_eval.sh
The string 'soccer' is NOT empty.
The string '' IS empty.
The string '' IS empty.
```

> **警告：** 空变量和未初始化的变量会对shell脚本测试造成灾难性的影响。如果不确定变量的内容，那么最好在将其用于数值或字符串比较之前先通过 `-n` 或 `-z` 来测试一下变量是否为空。

3. 文件比较

比较	描述
-d *file*	检查 *file* 是否存在且为目录
-e *file*	检查 *file* 是否存在

比较	描述
-f *file*	检查 *file* 是否存在且为文件
-r *file*	检查 *file* 是否存在且可读
-s *file*	检查 *file* 是否存在且非空
-w *file*	检查 *file* 是否存在且可写
-x *file*	检查 *file* 是否存在且可执行
-O *file*	检查 *file* 是否存在且属当前用户所有
-G *file*	检查 *file* 是否存在且默认组与当前用户相同
file1 -nt *file2*	检查 *file1* 是否比 *file2* 新
file1 -ot *file2*	检查 *file1* 是否比 *file2* 旧

1. 检查目录

jump_point.sh

```
#!/bin/bash
# Look before you leap
#
jump_directory=/home/Torfa
#
if [ -d $jump_directory ]
then
     echo "The $jump_directory directory exists."
     cd $jump_directory
     ls
else
     echo "The $jump_directory directory does NOT exist."
fi
```

2. 检查对象是否存在

`-e` 允许在使用文件或目录前先检查其是否存在：

update_file.sh

```
#!/bin/bash
# Check if either a directory or file exists
#
location=$HOME
file_name="sentinel"
#
if [ -d $location ]
then
     echo "OK on the $location directory"
     echo "Now checking on the file, $file_name..."
     if [ -e $location/$file_name ]
     then
```

```
          echo "OK on the file, $file_name."
          echo "Updating file's contents."
          date >> $location/$file_name
     #
     else
          echo "File, $location/$file_name, does NOT exist."
          echo "Nothing to update."
     fi
#
else
     echo "Directory, $location, does NOT exist."
     echo "Nothing to update."
fi
```

3. 检查文件

`-e` 测试可用于文件和目录。如果要确定指定对象为文件，那就必须使用 `-f` 测试：

dir-or-file.sh

```
#!/bin/bash
# Check if object exists and is a directory or a file
#
object_name=$HOME/lxc
echo
echo "The object being checked: $object_name"
echo
#
if [ -e $object_name ]
then
     echo "The object, $object_name, does exist,"
     #
     if [ -f $object_name ]
     then
          echo "and $object_name is a file."
     #
     else
          echo "and $object_name is a directory."
     fi
#
else
     echo "The object, $object_name, does NOT exist."
fi
```

4. 检查是否可读

can-i-read-it.sh

5. 检查空文件

is-it-empty.sh

6. 检查是否可写

can-i-write-to-it.sh

7. 检查文件是否可以执行

can-i-run-it.sh

8. 检查所有权

do-i-own-it.sh

9. 检查默认属组关系

`-G` 测试可以检查文件的属组，如果与用户的默认组匹配，则测试成功。`-G` 只会检查默认组，而非用户所属的所有组。

check-default-group.sh

10. 检查文件日期

check_file_dates.sh

`-nt` 与 `-ot` 选项都不会检查文件是否存在，即在文件不存在的情况下，该选项会得出错误的结果，所以，在使用时需要确保文件已经存在.

12.5 复合条件测试

`if-then`语句允许使用布尔逻辑将测试条件组合起来。可以使用以下两种布尔运算符。

- `[ condition1 ] && [ condition2 ]`
- `[ condition1 ] || [ condition2 ]`

例如:

AndBoolean.sh

```
if [ -d $HOME ] && [ -w $HOME/newfile ]
then
    echo "The file $HOME/newfile exists and you can write to it"
#
else
    echo "You cannot write to the file."
#
fi
```

12.6 *if-then* 的高级特性

bash shell 还提供了3个可在 `if-then` 语句中使用的高级特性。

- 在子shell中执行命令的 `单括号`
- 用于数学表达式的 `双括号`
- 用于高级字符串处理功能的 `双方括号`

1. *使用单括号*

单括号允许在 `if` 语句中使用子shell（子shell的用法参见第5章）。

命令格式:

```
(command)
```

在bash shell执行 *command* 之前，会先创建一个子shell，然后在其中执行命令。如果命令成功结束，则退出状态码（参见第11章）会被设为0，`then` 部分的命令就会被执行。如果命令的退出状态码不为0，则不执行 `then` 部分的命令。

例如:

SingleParentheses.sh SingleParenthese_b.sh

```
echo $BASH_SUBSHELL
if ( echo $BASH_SUBSHELL )
then
    echo "The Subshell command operated successfully."
else
    echo "The Subshell command NOT successful."
fi
# output:
# 0
# 1
# The Subshell command operated successfully.
```

当脚本第一次（在 `if` 语句之前）执行 `echo $BASH_SUBSHELL` 命令时，是在当前shell中完成的。该命令会输出0，表明没有使用子shell（`$BASH_SUBSHELL` 环境变量参见第5章）。在 `if` 语句内，脚本在子shell中执行 `echo $BASH_SUBSHELL` 命令，该命令输出1，表明使用了子shell。子shell操作成功，接下来是执行 `then` 部分的命令。

> 注意：在`if test`语句中使用进程列表(参见第5章)时，哪怕进程列表中除最后一个命令之外的命令全都失败，即只有进程列表中最后一个命令执行成功，子shell仍会将退出码设置为0.

2. *使用双括号*

双括号 命令允许在比较过程中使用高级数学表达式。`test` 命令在进行比较的时候只能使用简单的算数操作。双括号命令提供了更多的数学符号，这些符号对有过其他编程语言经验的程序员并不陌生。

命令格式:

```
(( expression ))
```

例如:

DoubleParentheses.sh

```
#
var1=10
#
if (($var1 ** 2 > 90)); then
    ((var2 = $var1 ** 2))
    echo "The square of $var1 = $var2,"
    echo "which is greater than 90."
fi
# output:
# lxc@Lxc:~/scripts/ch12$ ./DoubleParentheses.sh
# The square of 10 = 100,
# which is greater than 90.
```

以下列出双括号中可用的部分运算符。

符号	描述
val++	后增
val--	后减
++val	先增
--val	先减
!	逻辑求反
~	位求反
**	幂运算
<<	左位移
>>	右位移
&	位布尔AND
|	位布尔OR
&&	逻辑AND
||	逻辑OR

注意在双括号中大于小于号不用转义。

3. *使用双方括号*

双方括号 命令提供了针对字符串比较的高级特性。

命令格式:

```
[[ expression ]]
```

expression 可以使用 `test` 命令中的标准字符串比较。除此之外，它还提供了 `test` 命令所不具备的另一个特性： **模式匹配** 。
在进行模式匹配时，可以定义通配符或正则表达式（参见 第20章）来匹配字符串。
注意，当在双方括号内使用 `==` 运算符或 `!=` 运算符时，运算符的右侧被视为通配符。如果使用的是 `~=` 运算符，则运算符右侧被视为POSIX扩展正则表达式。

DoubleBrackets.sh

```
# DoubleBrackets.sh
if [[ $BASH_VERSION == 5.* ]]
then
    echo "You are using bash shell 5 series."
fi
```

> **注意：** 双方括号在bash shell中运作良好。不过要小心，不是所有的shell的支持双方括号。

12.7 *case* 命令

命令格式:

```
case variable in
pattern1 | pattern2) commands1;;
pattern3) commands2;;
*) default commands;;
esac
```

来个例子:

ShortCase.sh

```
#!/bin/bash
# Using a short case statement.
#
case $USER in
rich | christine)
    echo "Welcome $USER"
    echo "Please enjoy your visit."
    ;;
barbara | lxc)
    echo "Hi, there, $USER"
    echo "We are glad you could join us."
    ;;
testing)
    echo "Please log out when done with test."
    ;;
*)
    echo "Sorry, you are not allowed here."
    ;;
esac
```

8. 实战演练

一个脚本，该脚本会将本章的结构化命令付诸实践，确定当前系统中可用的软件包管理器，以已安装的软件包管理器为指导，猜测当前系统是基于哪个Linux发行版。

PackageMgrCheck.sh

脚本中用到了输出重定向。`which` 命令的标准输出和标准错误输出都通过 `&>` 符号重定向到 */dev/null*，这个地方被幽默地称为黑洞，因为被送往这里的东西从来都是有来无回。第15章 介绍了错误重定向。

ch13 更多的结构化命令

13.1 *for* 命令

命令格式:

```
for var in list
do
    commands
done
```

也可以将 `do` 语句和 `for` 语句放在一行，使用分号隔开：

```
for var in list; do
    commands
done
```

之后的循环语句也是同理，循环语句和后面的 `do` 语句均可放在同一行，使用分号隔开。

1. 读取列表中的值

在最后一次迭代结束后，*$var* 变量中的值在shell脚本的剩余部分依然有效。它会一直保持最后一次迭代时的值（除非做了修改）。

例如:

test1b.sh

```
#!/bin/bash
# Testing the for variable after the looping

for test in AlaBama Alaska Arizona Arkansas California Colorado
do
    echo "The next state is $test"
done
echo "The last state we visited was $test"
test=Connecticut
echo "Wait, now we're visiting $test"
```

2. 读取列表中的复杂值

shell看到 *list* 中的单引号会尝试使用他们来定义一个单独的数据值，这会导致一些问题。

badtest1.sh

```
#!/bin/bash
# another example of how not to use the for command

for test in I don't know if this'll work
do
    echo "word:$test"
done
# output:
lxc@Lxc:~/scripts/ch13$ ./badtest1.sh
word:I
word:dont know if thisll
word:work
```

我们有两种方法解决:

- 使用转义字符将单引号转义
- 使用双引号来定义含有单引号的值

例如:

test2.sh

```
#!/bin/bash
# another example of how not to use the for command

for test in I don\'t know if "this'll" work; do
    echo "word:$test"
done
# 在第一个有问题的地方使用反斜线进行了转义，第二个有问题的地方，将this'll放在了双引号内,
# 这两种方法都管用.
# output:
lxc@Lxc:~/scripts/ch13$ ./test2.sh
word:I
word:don't
word:know
word:if
word:this'll
word:work
```

for 循环假定 *list* 中各个值是以空格（制表符、换行符）分隔的，如果某个值含有空格，则必须将其放入双引号内:

test3.sh

```
 #!/bin/bash
 # an example of how to properly define values
for test in Nevada "New Hampshire" "New Mexico" "New York"
do
   echo "Now going to $test"
done
```

3. 从变量中读取列表值

test4.sh

```
#!/bin/bash
# using a variable to hold the list

list="Alabama Alaska Arizona Arkansas Colorado"
list=$list" Connecticut" # 这里是字符串拼接

for state in $list
do
   echo "Have you ever visited $state?"
done
# output:
lxc@Lxc:~/scripts/ch13$ bash test4.sh
Have you ever visited Alabama?
Have you ever visited Alaska?
Have you ever visited Arizona?
Have you ever visited Arkansas?
Have you ever visited Colorado?
Have you ever visited Connecticut?
```

4. 从命令中读取值列表

生成值列表的另一种途径是使用命令的输出。你可以使用 *命令替换* （参考第11章）来生成值列表。

例如：

test5.sh

```
#!/bin/bash
# reading values from a file

file="states.txt"
for state in $(cat $file)
do
    echo "Visit beautiful $state"
done
# output:
lxc@Lxc:~/scripts/ch13$ ./test5.sh
Visit beautiful Alabama
Visit beautiful Alaska
Visit beautiful Arizona
Visit beautiful Arkansas
Visit beautiful Colorado
Visit beautiful Connecticut
```

```
Visit beautiful Delaware
Visit beautiful Florida
Visit beautiful Georgia
Visit beautiful New # 注意这个单词和下一行的单词是一个值，被分开了。
Visit beautiful York
Visit beautiful "New
Visit beautiful Hampshire" # 这个用引号包围的值也被分开了
Visit beautiful North
Visit beautiful Carolina
# 在 states.txt 文件中每个值一行，如果某个值中以空格分隔，
# 则仍旧会出现以空格分隔的一个值被识别成两个值的情况
```

5. 更改字段分隔符

造成这个问题的原因是特殊的环境变量 `IFS（internal field separator，内部字段分隔符）`。`IFS` 环境变量定义了bash shell用作字段分隔符的一系列字符。在默认情况下，bash shell会将 空格、制表符、换行符作为字段分隔符。

解决这个问题的办法是在shell脚本中临时更改 `IFS` 环境变量的值来限制被bash shell视为字段分隔符的字符。

例如：

```
IFS=$'\n' # 字段分隔符仅为换行符
IFS=: # 字段分隔符仅为冒号，用于遍历文件中以冒号分隔的值(比如/etc/passwd)
IFS=$'\n:;"' # 也可以指定多个IFS字符。该语句将换行符、冒号、分号和双引号作为字段分隔符
```

test5b.sh

```
#!/bin/bash
# reading values from a file

file="states.txt"

IFS=$'\n'
for state in $(cat $file)
do
    echo "Visit beautiful $state"
done
# output:
lxc@Lxc:~/scripts/ch13$ ./test5b.sh
Visit beautiful Alabama
Visit beautiful Alaska
Visit beautiful Arizona
Visit beautiful Arkansas
Visit beautiful Colorado
Visit beautiful Connecticut
Visit beautiful Delaware
Visit beautiful Florida
Visit beautiful Georgia
Visit beautiful New York
Visit beautiful "New Hampshire"
Visit beautiful North Carolina
```

有时候，我们需要 **在一个地方修改IFS的值，然后再将其复原** 可以使用下面的手法：

```
IFS.OLD=$IFS
IFS=$'\n' #更改为新的IFS值
<在代码中使用新的IFS值>
IFS=$IFS.OLD
```

6. 使用通配符读取目录

可以使用 *for* 命令来遍历目录中的文件。为此，必须使用在文件名或者路径名中使用通配符，这会强制shell使用 **文件名通配符匹配(file globbing)**。文件名通配符匹配是生成与指定通配符匹配的文件名或路径名的过程。

例如：

test6.sh
test7.sh

```
#!/bin/bash
# iterate through all the files in a directory

for file in /home/lxc/scripts/* /home/lxc/Go/* # 可以列出多个目录通配符.
do
    if [ -d "$file" ]
    then
        echo "$file is a directory."
    elif [ -f "$file" ]
    then
        echo "$file is a file."
    fi
done
```

注意这个处理，`if [ -d "$file" ]`，因为在Linux中，文件名或者目录名中包含空格是完全合法的，要应对这种情况，应该将 `$file` 变量放在双引号内。否则在遇到含有空格的目录名或文件名时会产生错误：

```
./test6: line6: [: too many arguments
./test6: line9: [: too many arguments
```

2. C语言风格的 *for* 命令

命令格式:

```
for (( variable assignment; condition; iteration process ))
```

该命令格式和bash shell标准的 `for` 命令并不一致

- 变量赋值可以有空格
- 迭代条件中的变量不以美元符号开头
- 迭代过程的算式不使用 `expr` 命令格式

例如：

test8.sh
test9.sh

```
#!/bin/bash
# multiple variables

for (( a=1, b=10; a <= 10; a++, b-- ))
do
    echo "$a - $b"
done
```

3. *while* 命令

只有在命令产生的退出状态码为0时，`while` 循环才会继续迭代。

1. while的基本格式

```
while test command
do
    other commands
done
```

`while` 命令中定义的 *test command* 与 `if-then` 语句中的格式一模一样（参见第12章）。

当然，如之前所言

```
while test command; do
    other commands
done
```

这样的形式也可以。

`test command` (即 *条件测试* 命令)最常见的用法是使用方括号:

test10.sh

```
#!/bin/bash
# while command test

var1=10
while [ $var1 -gt 0 ]
do
    echo $var1
    var1=$[ $var1 - 1 ]
done
```

2. 使用多个测试命令

> 注意: `while` 命令允许在 `while` 语句行定义多个测试命令。**只有最后一个测试命令的退出状态码会被用于决定是否结束循环**。

test11.sh

```

```

```
#!/bin/bash
# testing a multicommand while loop

var1=3

while echo $var1
      [ $var1 -ge 0 ]
do
   echo "This is inside the loop"
   var1=$[ $var1 - 1 ]
done
# output
# 3
# This is inside the loop
# 2
# This is inside the loop
# 1
# This is inside the loop
# 0
# This is inside the loop
# -1 注意额外多输出的这个 -1
```

> 注意： 在指定多个测试命令时，注意要把每个测试命令单独放在一行中(除了 [ ] &&/| | [ ] 这种情况)，否则出错。

4. *until* 命令

`until` 命令和 `while` 命令的工作方式正相反。只要测试命令的退出状态码不为0，bash shell就会执行循环中的命令。直至测试命令返回了状态码0，循环结束。

命令格式:

```
until test command
do
    other commands
done
```

- 如前所述，`do` 也可以与 `until` 放在同一行，中间用分号隔开。
- 与 `while` 命令类似，你可以在 `until` 命令语句中放入多个 `test command`。最后一个命令的退出状态码决定了bash shell是否执行已定义的 `other commands`。

> 注意： 在指定多个测试命令时，注意要把每个测试命令单独放在一行中(除了 [ ] &&/| | [ ] 这种情况)，否则出错。（与 `while` 一样）。

test12.sh
test13.sh

```
#!/bin/bash
# using the until command

var1=100

until echo $var1
        [ $var1 -eq 0 ]
do
    echo "Inside the loop: $var1"
    var1=$[ $var1 - 25 ]
done
```

5. 嵌套循环

循环语句可以在循环内使用任意类型的命令，包括其他循环命令，这称为 **嵌套循环**。
举两个例子吧：

test14.sh

```
#!/bin/bash
# nesting for loops

for (( a = 1; a <= 3; a++ ))
do
    echo "Starting loop $a:"
    for (( b = 1; b <= 3; b++ ))
    do
        echo "      Inside loop: $b"
    done
done
```

test15.sh
test16.sh

```
# !/bin/bash
# using until and while loops

var1=3

until [ $var1 -eq 0 ]
do
    echo "Outer loop: $var1"
    var2=1
    while [ $var2 -lt 5 ]
    do
        var3=$(echo "scale=4; $var1 / $var2" | bc)
        echo "Inner loop: $var1 / $var2 = $var3"
        var2=$[ $var2 + 1 ]
    done
    var1=$[ $var1 - 1 ]
done
```

6. 循环处理文件数据

例如：

test16 .sh

```
#!/bin/bash
# changing the IFS value

IFS.OLD=$IFS
IFS=$'\n'
for entry in $(cat /etc/passwd)
do
    echo "Values in $entry -"
    IFS=:
    for value in $entry
    do
        echo "  $value"
    done
done
```

7. 循环控制

1. break命令

你可以使用 `break` 命令退出任意类型的循环。

1. 跳出单个循环

例如：

test17.sh
test18.sh

```
#!/bin/bash
# breaking out of a for loop

for var1 in 1 2 3 4 5 6 7 8 9 10
do
    if [ $var1 -eq 5 ]
    then
        break
    fi
    echo "Iteration number: $var1"
done
echo "The for loop is completed."
```

2. 跳出内层循环

例如：

test19.sh

```
#!/bin/bash
# breaking out of an inner loop
```

```
3
4  for (( a = 1; a <= 3; a++ ))
5  do
6      echo "Outer loop: $a"
7      for(( b = 1; b <= 100; b++ ))
8      do
9          if [ $b -eq 5 ]
10         then
11             break
12         fi
13         echo "  Inner loop: $b"
14     done
15 done
16 echo "done."
```

3. 跳出外层循环

有时你位于内层循环，但需结束外层循环。`break`命令接受单个命令行参数：

```
1  break n
```

其中 `n` 指定了要跳出的循环层级，默认情况下，`n` 为1，表明跳出的是当前循环。如果将 `n` 设为2，则 `break` 命令会停止下一级的外层循环：

test20.sh

```
1  #!/bin/bash
2  # breaking out of an outer loop
3
4  for (( a = 1; a <= 3; a++ ))
5  do
6      echo "Outer loop: $a"
7      for((b = 1; b <= 100; b++ ))
8      do
9          if [ $b -eq 5 ]
10         then
11             break 2
12         fi
13         echo "  Inner loop: $b"
14     done
15 done
16 echo "done."
```

2. continue命令

跳过某次循环，但不终止循环。

test21.sh

```bash
#!/bin/bash
# using the continue command.

for (( var1 = 1; var1 <= 20; var1++ ))
do
    if [ $var1 -gt 5 ] && [ $var1 -lt 10 ]
    then
        continue
    fi
    echo "Iteration number: $var1"
done
echo "Done."
```

与`break`命令一样，`continue`命令也允许通过命令行参数指定要继续执行哪一级的循环：

```
continue n
```

下面是一个继续执行外层`for`循环的例子：

test22.sh

```bash
#!/bin/bash
# continuing an outer loop

for ((a = 1; a <= 8; a++)); do
    echo "Outer loop: $a"
    for ((b = 1; b <= 3; b++)); do
        if [ $a -gt 3 ] && [ $a -lt 6 ]; then
            continue 2
        fi
        var3=$(($a * $b))
        echo "  The result of $a * $b is $var3."
    done
done
echo "Done."
# output:
lxc@Lxc:~/scripts/ch13$ ./test22.sh
Outer loop: 1
  The result of 1 * 1 is 1.
  The result of 1 * 2 is 2.
  The result of 1 * 3 is 3.
Outer loop: 2
  The result of 2 * 1 is 2.
  The result of 2 * 2 is 4.
  The result of 2 * 3 is 6.
Outer loop: 3
  The result of 3 * 1 is 3.
  The result of 3 * 2 is 6.
  The result of 3 * 3 is 9.
Outer loop: 4
Outer loop: 5
Outer loop: 6
  The result of 6 * 1 is 6.
  The result of 6 * 2 is 12.
```

```
  The result of 6 * 3 is 18.
Outer loop: 7
  The result of 7 * 1 is 7.
  The result of 7 * 2 is 14.
  The result of 7 * 3 is 21.
Outer loop: 8
  The result of 8 * 1 is 8.
  The result of 8 * 2 is 16.
  The result of 8 * 3 is 24.
Done.
```

8. 处理循环的输出

在shell脚本中，**可以对循环的输出使用管道或进行重定向。这可以通过在 `done` 命令之后添加一个处理命令来实现**

下面这个例子将 `for` 命令的输出重定向至文件：

test23.sh

```
#!/bin/bash
# redirecting the for output to a file

for (( a = 1; a <= 5; a++ ))
do
    echo "Iteration number: $a"
done > output.txt
echo "Done."
```

下面这个例子使用了输入重定向:

test26.sh

```
#!/bin/bash
# process new user accounts

input="users.csv"

while IFS=',' read -r userid name; do
    echo "adding $userid"
    useradd -c "$name" -m $userid
done <"$input"

```

下面这个例子使用管道，将循环的结果传输到另一个命令：

test24.sh

```
#!/bin/bash
# piping a loop to another command

for state in "North Dakota" Connecticut Illinois AlaBama Tenessee
do
    echo "$state is the next place to go."
done | sort
echo "Done."
```

`for`命令的输出通过管道传给了`sort`命令，由后者对输出结果进行排序。

9. 实战演练

搞两个脚本。

1. 查找可执行文件

test25.sh

```
#!/bin/bash
# finding files in the PATH

IFS=:
for folder in $PATH
do
    echo "$folder:"
    for file in $folder
    do
        if [ -x $file ]
        then
            echo "    $file"
        fi
    done
done
# output:
lxc@Lxc:~/scripts/ch13$ ./test25.sh
/home/lxc/.nvm/versions/node/v18.16.0/bin:
    /home/lxc/.nvm/versions/node/v18.16.0/bin
/usr/lib/jvm/jdk1.8.0_162/bin:
    /usr/lib/jvm/jdk1.8.0_162/bin
/usr/local/sbin:
......
```

2. 创建多个用户帐户

test26.sh

```
#!/bin/bash
# process new user accounts

input="users.csv"

while IFS=',' read -r userid name; do
    echo "adding $userid"
    useradd -c "$name" -m $userid
done <"$input"
```

"users.csv" 这个文本文件的格式如下：

```
loginame, name
```

第一项是为新用户账户所选用的用户id。第二项是用户的全名。两个值之间以逗号分隔，这样就形成了一种叫作CSV(comma-separated value, 逗号分隔值)的文件格式。我们将 `IFS` 设置成逗号即可。

ch14 处理用户输入

bash shell提供了一些不同的方法来从用户处获取数据，包括命令行参数（添加在命令行后的数据）、命令行选项（可改变命令行为的单个字母）以及从键盘读取输入。

1. 传递参数

向shell脚本传递参数的最基本方法是使用 **命令行参数**。格式如下：

```
./addem 10 30
```

1. 读取参数

bash shell会将所有的命令行参数都指派给称作 **位置参数** 的特殊变量。这也包括shell脚本名称。位置变量的名称都是标准数字：`$0` 对应脚本名，`$1` 对应第一个命令行参数，`$2` 对应第二个，第9个之后，必须在变量名两侧加上花括号用于界定名称，例如 `${10}` 表示第10个位置参数。

例如：

positional1.sh positional2.sh
stringparam.sh

```
#!/bin/bash
# Usint the command-line string parameter
#
echo Hello $1, glad to meet you.
exit
```

参数之间是以空格分隔的，如果你的命令行参数中包含空格，需以引号（单或者双）包围：

```
./stringparam 'big world'
# output: Hello big world, glad to meet you.
```

2. 读取脚本名

使用位置变量 `$0` 来获取运行中的shell脚本名。

如果你使用另一个命令来运行shell脚本，则命令名和脚本名会混在一起，出现在位置变量`$0`中：
当然你在脚本所在目录以 `bash 脚本名` 的命令运行脚本， `$0` 中是只有脚本名的。

```
bash position0.sh
# output: This script name is position0.sh.

./position0.sh
# output This script name is ./position0.sh.
```

如果你运行脚本时使用的是绝对路径，那么位置变量 `$0` 就会包含整个路径

如果你编写脚本只打算使用脚本名，可以使用 `basename` 命令，该命令可以返回不包含路径的脚本名：

posbasename.sh

```
#!/bin/bash
# Using basename with the $0 command-line parameter.
#
name=$( basename $0 )
#
echo The script name is $name
exit
# output: The script name is posbasename.sh
```

3. 参数测试

当脚本认为位置变量中应该有数据，而实际根本没有的时候，脚本很可能会产生错误消息。这种编写脚本的方法并不可取。在使用位置变量之前一定要检查是否为空：

checkpositional1.sh

```
#!/bin/bash
# Using one command-line parameter.
#
if [ -n "$1" ]
then
    factorial=1
    for (( number = 1; number <= $1; number++ ))
    do
        factorial=$[ $factorial * $number ]
    done
    echo "The factorial of $1 is $factorial."
else
    echo "You did not provide a parameter."
fi
exit
```

上述例子使用了 `-n` 测试命令行参数 `$1` 是否为空。

2. 特殊参数变量

在bash shell中有一些跟踪命令行参数的特殊变量。

1. 参数统计

特殊变量 `$#` 含有脚本运行时携带的命令行参数的个数。

例如:

counteparameters.sh

```
#!/bin/bash
# Counting command-line parameters
#
if [ $# -eq 1 ]; then
    fragment="parameter was"
else
    fragment="parameter were"
fi
echo $# $fragment supplied.
exit
# output:
lxc@Lxc:~/scripts/ch14$ ./countparameters.sh
0 parameter were supplied.
lxc@Lxc:~/scripts/ch14$ ./countparameters.sh Hello
1 parameter was supplied.
lxc@Lxc:~/scripts/ch14$ ./countparameters.sh Hello World
2 parameter were supplied.
lxc@Lxc:~/scripts/ch14$ ./countparameters.sh "Hello World"
1 parameter was supplied.
```

你可能认为如果 `$#` 变量含有命令行参数的总数，那么变量 `${$#}` 应该就代表最后一个位置变量了。实则不然，想使用最后一个位置变量，必须将花括号内的`$`换成`!`（很奇怪。。可能因为实现的原因吧）。

goodlastparametest.sh

```
#!/bin/bash
# Testing grabbing the last parameter
#
echo The number of parameters is $#
echo The last parameter is ${!#}
exit
# output:
lxc@Lxc:~/scripts/ch14$ ./goodlastparamtest.sh one two three four
The number of parameters is 4
The last parameter is four
```

需要注意的是，当命令行中没有任何参数时，`$#` 的值即为0，但 `${!#}` 会返回命令行中的脚本名。

2. 获取所有数据

`$@` 和 `$*` 变量都包含里所有的命令行参数。

区别:

- `$*` 变量会将所有的命令行参数视为一个单词。这个单词含有命令行中出现的每一个参数。基本上，`$*` 变量会将这些参数视为一个整体，而不是一系列个体。当 `$*` 出现在双引号内时，会被扩展成由多个命令行参数组成的单个单词，每个参数之间以IFS变量值的第一个字符分隔，也就是说，$*会被扩展为 "$1c$2c..."（其中c是IFS变量值的第一个字符）。
- `$@` 变量会将所有的命令行参数视为同一字符串中的多个独立的单词，以便你能遍历并处理全部参数。这通常使用 `for` 命令完成。当 `$@` 出现在双引号时，其所包含的各个命令行参数会被扩展成为独立的单词，也就是说"$@"会被扩展为"$1""$2"...。

例如:

grabdisplayallparams.sh

```
#!/bin/bash
# Exploring different methods for grabbing all the parameters.
#
echo
echo "Using the \$* method: $*"
count=1
for param in "$*"
do
    echo "\$* Parameter #$count = $param"
    count=$[ $count + 1 ]
done
#
echo
echo "using the \$@ method: $@"
count=1
for param in "$@"
do
    echo "\$@ Parameter #$count = $param"
    count=$[ $count + 1 ]
done
echo
exit
# ./grabdisplayallparams.sh alpha beta charlie delta
# ouput:

Using the $* method: alpha beta charlie delta
$* Parameter #1 = alpha beta charlie delta

using the $@ method: alpha beta charlie delta
$@ Parameter #1 = alpha
$@ Parameter #2 = beta
$@ Parameter #3 = charlie
$@ Parameter #4 = delta
```

`$*` 变量会将所有参数视为单个参数，而 `$@` 变量会单独处理每个参数。

3. 移动参数

`shift` 命令默认情况下会将每个位置的变量值都向左移动一个位置。因此，变量$3会被移入$2，$2会被移入$1，$1会被移出，如果某个参数被移出，那么它的值就被丢弃了，无法再恢复。

例如：

shiftparams.sh

```
#!/bin/bash
# Shifting through the parameters
#
echo
echo "Using the shift method: "
count=1
while [ -n "$1" ]
do
    echo "Parameter #$count = $1"
    count=$[ $count + 1 ]
    shift
done
echo
# output:
lxc@Lxc:~/scripts/ch14$ ./shiftparams.sh alpha bravo charlie delta

Using the shift method:
Parameter #1 = alpha
Parameter #2 = bravo
Parameter #3 = charlie
Parameter #4 = delta

```

当然，可以给`shift`命令提供一个参数，指明要移动的位置数：

bigshiftparams.sh

```
#!/bin/bash
# Shifting mulitiple positions through the parameters
#
echo
echo "The original parameters: $*"
echo "Now shifting 2..."
shift 2
echo "Here's the new first parameter: $1"
echo
exit
# output:
lxc@Lxc:~/scripts/ch14$ ./bigshiftparams.sh 1 2 3 4 5 6 7

The original parameters: 1 2 3 4 5 6 7
Now shifting 2...
Here's the new first parameter: 3

```

4. 处理选项

选项 是在连字符之后出现的单个字母（还有以双连字符起始的，后跟字符串的 **长选项** ），能够改变命令的行为。
命令行参数 是在脚本名之后出现的各个单词，其中以连字符（-）或双连字符（--）起始的参数。因其能改变命令的行为，称作 **命令行选项**。所以，命令行选项是一种特殊形式的命令行参数。

1. 查找选项

1. 处理简单选项

上个例子吧:

extractoptions.sh

```
#!/bin/bash
# Extract command-line options.
#

while [ -n "$1" ]
do
    case "$1" in
    -a) echo "Found the -a option." ;;
    -b) echo "Found the -b option." ;;
    -c) echo "Found the -c option." ;;
    *) echo "$1 is not an options.";;
    esac
    shift

done
echo
exit
# ./extractoptions.sh -a -b -c -d
# output:
# Found the -a option.
# Found the -b option.
# Found the -c option.
# -d is not an options.
```

2. 分离参数和选项

在Linux中使用双连字符将 **选项** 与 **参数** 分开，该字符告诉脚本 *选项* 何时结束，*参数* 何时开始。

例如:

extractoptionsparams.sh

```
#!/bin/bash
# Extract command-line options and parameters.
#
echo
while [ -n "$1" ]
do
    case "$1" in
    -a) echo "Found the -a option.";;
```

```
    -b) echo "Found the -b option.";;
    -c) echo "Found the -c option.";;
    --) shift
        break;;
    *) echo "'$1' is not an option";;
    esac
    shift
done
echo
count=1
for param in $@
do
    echo "Parameter #$count: $param"
    count=$[ $count + 1 ]
done
echo
exit
# ./extractoptionsparams.sh -a -b -c -- test1 test2 test3
# output:
# Found the -a option.
# Found the -b option.
# Found the -c option.

# Parameter #1: test1
# Parameter #2: test2
# Parameter #3: test3
```

3. 处理含值的选项

有些选项需要一个额外参数值。像下面这样:

```
./testing.sh -a param1 -b -c -d param2
```

当命令行选项要求额外的参数时，脚本必须能够检测到并正确的加以处理。来看下面的处理方法。

extractoptionsvalues.sh

```
#!/bin/bash
# Extract command-line options and values
#
echo
while [ -n "$1" ]
do
    case "$1" in
    -a) echo "Found the -a option";;
    -b) param=$2
        echo "Found the -b option with parameter value $param"
        shift;;
    -c) echo "Found the -c options";;
    --) shift
        break;;
    *) echo "$1 is not an options";;
    esac
    shift
done
```

```
#
echo
count=1
for param in $@
do
    echo "Parameter #$count: $param"
    count=$[ $count + 1 ]
done
exit
# ./extractoptionsvalues.sh -a -b Bvalues -c
# output:

# Found the -a option
# Found the -b option with parameter value Bvalues
# Found the -c options
```

在这个例子中，`case` 语句定义了3个该处理的选项。`-b` 选项还需要一个额外的参数值。由于要处理的选项位于 `$1`，因此额外的参数值就应该位于 `$2`。只要将参数值从 `$2` 变量中提取出来就可以了。当然，因为这个选项占用了两个位置，所以还需要使用 `shift` 命令多移动一次。

有时，我们需要合并选项，像下面这样：

```
./testing.sh -ab Bvalues -cd
```

我们使用下面的方法。

2. 使用 `getopt` 命令

1. 命令格式

```
getopt optstring parameters
```

其中，*optstring* 定义了有效的命令行选项字母，以及哪些选项字母需要参数值。如果 *optstring* 未包含你指定的选项，则在默认情况下，`getopt` 命令会产生一条错误消息。如果想忽略这条消息，可以使用 `getopt` 的 `-q` 选项。

首先，在 *optstring* 中列出脚本中用到的每个命令行选项字母。然后，在每个需要参数值的选项字母后面加一个冒号。`getopt` 命令会基于你定义的 *optstring* 解析提供的参数。

例如：

```
getopt ab:cd -a -b Bvalue -cde test1 test2
# output:
# getopt: invalid option -- 'e'
# -a -b Bvalue -c -d -- test1 test2
```

2. 在脚本中使用 `getopt`

我们可以在脚本中使用 `getopt` 命令来格式化脚本所携带的任何命令行参数或选项，但用起来略显复杂。
难点在于要使用 `getopt` 命令生成的格式化版本替换已有的命令行选项和参数。要使用 `getopt` 命令生成的格式化版本替换已有的命令行选项和参数。要求助于 `set` 命令。
`set` 命令有一个选项是双连字符(--)，可以将 *位置变量* 的值替换成 `set` 命令所指定的值。

具体做法是，将脚本的命令行参数传给 `getopt` 命令，然后再将 `getopt` 命令的输出传给 `set` 命令，用 `getopt` 格式化后的命令行参数来替换原始的命令行参数，如下

```
set -- $(getopt -a ab:cd "$@")
```

现在，位置变量原先的值会被 `getopt` 命令的输出替换掉，后者已经为我们格式化好了命令行参数。

例如:

extractwithgetopt.sh

```
#!/bin/bash
# Extract command-line options and values with getopt
#
set -- $(getopt -q ab:cd "$@")
#
echo
while [ -n "$1" ]
do
    case "$1" in
    -a) echo "Found the -a option.";;
    -b) param=$2
        echo "Found the -b option with parameter value $param"
        shift;;
    -c) echo "Found the -c option";;
    --) shift
        break;;
    *) echo "$1 is not an option.";;
    esac
    shift
done
#
echo
count=1
for param in $@
do
    echo "Parameter #$count: $param"
    count=$[ $count + 1 ]
done
exit
# ./extractwithgetopt.sh -ac -b Bvalue -d test1 test2
# output:
#
# Found the -a option.
# Found the -c option
# Found the -b option with parameter value 'Bvalue'
# -d is not an option.

# Parameter #1: 'test1'
# Parameter #2: 'test2'
```

不过，`getops` 命令存在一个小问题。看下面这个例子:

```
lxc@Lxc:~/scripts/ch14$ ./extractwithgetopt.sh -c -d -b Bvalue -a "test1
test2" test3

Found the -c option
-d is not an option.
Found the -b option with parameter value 'Bvalue'
Found the -a option.

Parameter #1: 'test1
Parameter #2: test2'
Parameter #3: 'test3'
```

`getopt` 命令会使用空格作为参数分隔符，而不是根据引号将二者当作一个参数。在命令行参数中，即便用引号将带有空格的参数包围，也会出现问题。`getopt` 命令并不擅长处理带有空格和引号的值。所以，有下面的 `getopts` 命令。

3. 使用 `getopts` 命令

`getopts` 命令是bash shell的内建命令。与 `getopt` 的不同之处在于，`getopt` 在将命令行中的选项和参数处理完后只生成一个输出，而 `getopts` 能够和已有的shell *位置变量* 配合默契。
`getopts` 每次只处理一个检测到的命令行参数。在处理完所有参数后 `getopts` 命令会退出并返回一个大于0的退出状态码。这使其非常适合用在解析命令行参数的循环中。

`getopts` 的命令格式如下:

```
getopts optstring variable
```

optstring 值与 `getopt` 命令中使用的值类似。有效的选项字段会在 *optstring* 中列出，如果选项字母要求有参数值，就在其后加一个冒号。如果不想显示错误消息的话，可以在 *optstring* 之前加一个冒号。
`getopts` 命令会将当前参数保存在命令行中定义的 *variable* 中。
`getopts` 命令要用到两个环境变量。如果选项需要加带参数值，那么 **`OPTARG`** 环境变量保存的就是这个值。
`OPTIND` 环境变量保存着参数列表中 `getopts` 正在处理的参数位置。这样在处理完当前选项之后就能继续处理其他命令行参数了。

例如:

extractwithgetopts.sh

```
#!/bin/bash
# Extract command-line options and parameters with getopts
#
echo
while getopts :ab:cd opt
do
    case "$opt" in
    a) echo "Found the -a option";;
    b) echo "Found the -b option with parameter value $OPTARG";;
    c) echo "Found the -c option";;
    d) echo "Found the -d option";;
    *) echo "Unkonwn option: $opt";;
    esac
done
```

```
#
shift $[ $OPTIND - 1 ]
#
echo
count=1
for param in "$@"
do
    echo "Parameter #$count: $param"
    count=$[ $count + 1 ]
done
exit
# ./extractoptsparamswithgetopts.sh -ab "Bvalue1 Bvalue2" -de test1 test2
# output:
#
# Found the -a option
# Found the -b option with parameter value Bvalue1 Bvalue2
# Found the -d option
# Unkonwn option: ?

# Parameter #1: test1
# Parameter #2: test2
```

`getopts` 命令能够从 `-b` 选项中正确解析出 Bvalue值，注意，这个Bvalue值中使用了双引号包围的空格分隔的形式，`getopts` 正确的处理了该形式。`getopts` 命令知道何时停止处理选项，并将参数留给你处理。在处理每个选项时，`getopts` 会将 `OPTIND` 环境变量值增1。处理完选项后，可以使用 `shift` 命令和 `OPTIND` 值来移动参数。注意，`getopts` 命令将在命令行找到的所有未定义的选项统一输出为问号。
至此，你拥有了一个能在所有shell脚本中使用的全功能命令行选项和参数处理工具。

5. 选项标准化

在编写shell脚本时，选用那些选项字母以及选项的具体用法，完全由你掌握。
但在Linux中，有些选项字母在某种程度上已经有了标准含义。如果能在shell脚本中支持这些选项，则你的脚本会对用户更友好。
下表列出Linux中一些命令行选项的常用含义。

选项	描述
-a	显示所有对象
-c	生成计数
-d	指定目录
-e	扩展对象
-f	指定读入数据的文件
-h	显示命令的帮助信息
-i	忽略文本大小写
-l	产生长格式的输出
-n	使用非交互模式(批处理)

选项	描述
-o	将所有输出重定向至文件
-q	以静默模式运行
-r	递归处理文件或目录
-s	以静默模式运行
-v	生成详细输出
-x	排除某个对象
-y	对所有问题回答yes

6. 获取用户输入

有时候脚本需要一些交互性。你可能想在脚本运行期间询问用户并等待用户回答。

1. 基本的读取

`read` 命令从标准输入（键盘）或另一个文件描述符中接受输入。获取输入后，`read` 命令会将数据存入变量。

例如:

askname.sh

```
#!/bin/bash
# Using the read command
#
echo -n "Enter your name: "
read name
echo "Hello $name, welcome to my script."
exit
# echo 的 -n 选项是不输出尾随换行符
# ./askname.sh
# output:
# Enter your name: l xc
# Hello l xc, welcome to my script.
```

`read` 命令也提供了 `-p` 选项，允许直接指定提示符:

askage.sh

```
#!/bin/bash
# Using the read command with the -p option
#
read -p "Please enter your age: " age
days=$[ $age * 365 ]
echo "That means you are over $days days old!"
exit
```

在第一个例子中输入姓名时，`read` 命令会将姓氏和名字（两者以空格分开了，如果你在该例中输入以引号包围的以空格分隔的字符串的话，那么最终的 *name* 变量中也会保存引号）保存在同一个变量中。`read` 命令会将提示符后输入的所有数据分配给单个变量。如果指定多个变量，则输入的每个数据值都会分配给列表中的下一个变量。如果变量数量不够，那么剩下的数据就全都分配给最后一个变量：

askfirstlastname.sh

```
#!/bin/bash
# Using the read command for multiple variables.
#
read -p "Enter your first and last name: " first last
echo "Checking data for $last, $first..."
exit
# ./askfirstlastname.sh
# output:
# Enter your first and last name: l xc yyds
# Checking data for xc yyds, l...
```

也可以在 `read` 命令中不指定任何变量，这样 `read` 命令便会将接收到的所有数据都放进特殊环境变量 `REPLY` 中：

asknamereply.sh

```
#!/bin/bash
# Using the read command with REPLY variable
#
read -p "Enter your name: "
echo
echo "Hello $REPLY, welcome to my script."
exit
# ./asknamereply.sh
# output:
# Enter your name: l xc
#
# Hello l xc, welcome to my script.
```

`REPLY` 环境变量包含输入的所有数据，其可以在shell脚本中像其他变量一样使用。

2. 超时

你可以使用 `-t` 选项来指定一个定时器。`-t` 选项会指定 `read` 命令等待输入的秒数。如果计时器超时，则 `read` 命令会返回非0的退出状态码：

asknametimed.sh

```
#!/bin/bash
# Using the read command with a timer
#
if read -t 5 -p "Enter your name: " name
then
    echo "Hello $name, welcome to my script."
else
    echo
    echo "Sorry, no longer waiting for time."
fi
exit
```

你也可以通过 `-n` 选项让 `read` 命令统计输入的字符数。当字符数达到预设值时，就自动退出，将已输入的数据赋给变量：

continueornot.sh

```
#!/bin/bash
# Using the read command for one character
#
read -n 1 -p "Do you want to continue [Y/N]? " answer
#
case $answer in
Y | y)
    echo
    echo "Okay. Continue on...";;
N | n)
    echo
    echo "Okay. Goodbay"
    exit;;
esac
echo "This is the end of the script."
exit
```

本例中使用了 `-n` 选项和数值1，告诉 `read` 命令在接收到单个字符后退出。只要按下单个字符进行应答， `read` 命令就会接受输入并将其传给变量，无须按Enter键。

3. 无显示读取

有时你需要从脚本用户处得到输入，但又不想在屏幕上显示输入信息。典型的例子就是输入密码，但除此之外还有很多种需要隐藏的数据。
`-s` 选项可以避免在 `read` 命令中输入的数据出现在屏幕上（其实数据还是会显示，只不过 `read` 命令将文本颜色设成了跟背景色一样）。

来个例子：

askpassword.sh

```
#!/bin/bash
# Hiding input date
#
read -s -p "Enter your password:" passwd
echo
echo "Your password is $passwd"
exit
# ./askpassword.sh
# output:
# Enter your password:
# Your password is 66666
```

4. 从文件中读取

我们也可以使用 `read` 命令读取文件。每次调用 `read` 命令都会从指定文件中读取一行文本。当文件中没有内容可读时， `read` 命令会退出并返回非0的状态码。
其中麻烦的地方是将文件数据传给 `read` 命令。最常见的方法是对文件使用 `cat` 命令，将结果通过管道直接传给含有 `read` 命令的 `while` 明令。 来个例子:

readfile.sh

```
#!/bin/bash
# Using the read command to read a file
#
count=1
cat $HOME/scripts/ch14/test.txt | while read line
do
    echo "Line $count: $line"
    count=$[ $count + 1 ]
done
echo "Finished processing the file."
exit
```

`while` 循环会持续通过 `read` 命令处理文件的各行，直到 `read` 命令以非0状态码退出。

7. 实战演练

本节搞一个脚本， 该脚本在处理用户输入的同时，使用 `ping` 命令或 `ping6` 命令来测试与其他网络主机的连通性。

CheckSystems.sh

```
#!/bin/bash
# Check systems on local network
# allowing for a variety of input
# methods.
#
#
############ Determine Input Method ###################
#

# Check for command-line options here using getopts.
# If none, then go on to File Input Method
#
```

```
while getopts t: opt
do
     case "$opt" in
          t) # Found the -t option
             if [ $OPTARG = "IPv4" ]
             then
                  pingcommand=$(which ping)
             #
             elif [ $OPTARG = "IPv6" ]
             then
                  pingcommand=$(which ping6)
             #
             else
                  echo "Usage: -t IPv4 or -t IPv6"
                  echo "Exiting script..."
                  exit
             fi
             ;;
          *) echo "Usage: -t IPv4 or -t IPv6"
             echo "Exiting script..."
             exit;;
     esac
     #
     shift $[ $OPTIND - 1 ]
     #
     if [ $# -eq 0 ]
     then
          echo
          echo "IP Address(es) parameters are missing."
          echo
          echo "Exiting script..."
          exit
     fi
#
     for ipaddress in "$@"
     do
          echo
          echo "Checking system at $ipaddress..."
          echo
          $pingcommand -q -c 3 $ipaddress
          echo
     done
     exit
done
#
############ File Input Method ###################
#
echo
echo "Please enter the file name with an absolute directory reference..."
echo
choice=0
while [ $choice -eq 0 ]
do
     read -t 60 -p "Enter name of file: " filename
     if [ -z $filename ]
     then
```

```
          quitanswer=""
          read -t 10 -n 1 -p "Quit script [Y/n]? " quitanswer
          #
          case $quitanswer in
          Y | y) echo
                 echo "Quitting script..."
                 exit;;
          N | n) echo
                 echo "Please answer question: "
                 choice=0;;
          *)     echo
                 echo "No response. Quitting script..."
                 exit;;
          esac
     else
          choice=1
     fi
done
#
if [ -s $filename ] && [ -r $filename ]
     then
          echo "$filename is a file, is readable, and is not empty."
          echo
          cat $filename | while read line
          do
               ipaddress=$line
               read line
               iptype=$line
               if [ $iptype = "IPv4" ]
               then
                    pingcommand=$(which ping)
               else
                    pingcommand=$(which ping6)
               fi
               echo "Checking system at $ipaddress..."
               $pingcommand -q -c 3 $ipaddress
               echo
          done
          echo "Finished processing the file. All systems checked."
     else
          echo
          echo "$filename is either not a file, is empty, or is"
          echo "not readable by you. Exiting script..."
fi
#
#################### Exit Script ####################
#
exit
```

下面是输出:

```
lxc@Lxc:~/scripts/ch14$ ./CheckSystems.sh

Please enter the file name with an absolute directory reference...

```

```
Enter name of file: /home/lxc/scripts/ch14/addresses.txt
/home/lxc/scripts/ch14/addresses.txt is a file, is readable, and is not
empty.

Checking system at 192.168.1.102...
PING 192.168.1.102 (192.168.1.102) 56(84) bytes of data.

--- 192.168.1.102 ping 统计 ---
已发送 3 个包，  已接收 0 个包，+3 错误，100% 包丢失，耗时 2002 毫秒


Checking system at 192.168.1.103...
PING 192.168.1.103 (192.168.1.103) 56(84) bytes of data.

--- 192.168.1.103 ping 统计 ---
已发送 3 个包，  已接收 0 个包，+3 错误，100% 包丢失，耗时 2003 毫秒


Checking system at 192.168.1.104...
PING 192.168.1.104 (192.168.1.104) 56(84) bytes of data.

--- 192.168.1.104 ping 统计 ---
已发送 3 个包，  已接收 0 个包，+3 错误，100% 包丢失，耗时 2003 毫秒


Finished processing the file. All systems checked.
########################################################################
lxc@Lxc:~/scripts/ch14$ ./CheckSystems.sh -t IPv4 192.168.1.108

Checking system at 192.168.1.108...

PING 192.168.1.108 (192.168.1.108) 56(84) bytes of data.

--- 192.168.1.108 ping 统计 ---
已发送 3 个包，  已接收 0 个包，+3 错误，100% 包丢失，耗时 2003 毫秒

################################################################################
lxc@Lxc:~/scripts/ch14$ ./CheckSystems.sh -t IPv4 192.168.1.108
192.168.101.111

Checking system at 192.168.1.108...

PING 192.168.1.108 (192.168.1.108) 56(84) bytes of data.

--- 192.168.1.108 ping 统计 ---
已发送 3 个包，  已接收 0 个包，+3 错误，100% 包丢失，耗时 2003 毫秒



Checking system at 192.168.101.111...

PING 192.168.101.111 (192.168.101.111) 56(84) bytes of data.

--- 192.168.101.111 ping 统计 ---
已发送 3 个包，  已接收 0 个包，100% 包丢失，耗时 2033 毫秒

```

```
########################################################################
lxc@Lxc:~/scripts/ch14$ ./CheckSystems.sh -t IPv4

IP Address(es) parameters are missing.

Exiting script...
```

ch15 呈现数据

本章将演示如何将脚本的输出重定向到Linux系统的不同位置。

1. 理解输入输出

到目前位置，你已经知道了两种显示脚本输出的方法。

- 在显示器屏幕上显示输出。
- 将输出重定向到文件。

这两种方法要么将数据全部显示出来，要么什么都不显示。但有时将一部分数据显示在屏幕上，另一部分数据保存到文件中更合适。对此，了解Linux如何处理输入输出有助于将脚本输出送往所需的位置。

1. 标准文件描述符

Linux系统会将每个对象当作文件来处理，这包括输入输出。Linux用 **文件描述符** 来标识每个文件对象。文件描述符是一个非负整数，唯一会标识的是会话中打开的文件。每个进程一次最多可以打开 **9** (这个数量并不是固定的)个文件描述符。出于特殊目的，bash shell保留了前 **3** 个文件描述符(0、1和2)。见下表。

文件描述符	缩写	描述
0	STDIN	标准输入
1	STDOUT	标准输出
2	STDERR	标准错误

1. STDIN

STDIN 文件描述符代表标准输入，于终端界面而言，标准输入就是键盘。在使用输入重定向符（<）时，Linux会用重定向指定的文件替换标准输入文件描述符。于是，命令就会从文件中读取数据，就好像这些数据就是从键盘键入的。

例如：

```
# 使用 cat 命令来处理STDIN的输入:
$ cat
This is a test
This is a test
This is a second test
This is a second test
# 当在命令行中输入cat命令时，它会从STDIN接受输入。输入一行，cat命令就显示一行。
# 也可以通过输入重定向符强制cat命令接受来自STDIN之外的文件输入:
$ cat < testfile
This is the first line.
This is the second line.
This is the third line.
```

内联输入重定向 **<<** 见第11章笔记。

2. STDOUT

STDOUT 文件描述符代表shell的标准输出，在终端界面上，标准输出就是显示器。shell的所有输出（包括shell中运行的程序和脚本）都会被送往标准输出。
可以通过输出重定向符 **>**，将输出重定向到指定的文件，也可以使用 **>>** 将数据追加到某个文件。

例如:

```
lxc@Lxc:~/scripts/ch15$ ls -l > test2
lxc@Lxc:~/scripts/ch15$ cat test2
总用量 108
-rwxrw-r-- 1 lxc lxc  145 11月  5 14:24 badtest.sh
-rw-rw-r-- 1 lxc lxc  186 11月  5 15:57 members.csv
-rw-rw-r-- 1 lxc lxc  554 11月  5 15:57 members.sql
-rw-rw-r-- 1 lxc lxc 1096 11月  6 18:11 README.md
...省略
lxc@Lxc:~/scripts/ch15$ who >> test2
lxc@Lxc:~/scripts/ch15$ cat test2
总用量 108
-rwxrw-r-- 1 lxc lxc  145 11月  5 14:24 badtest.sh
-rw-rw-r-- 1 lxc lxc  186 11月  5 15:57 members.csv
...省略
lxc      tty2         2023-11-06 18:58 (tty2)
```

shell对于错误消息的处理和普通输出是分开的。因此，当对脚本使用标准输出重定向时，如果脚本产生错误消息，错误消息会被显示在屏幕上，而输出的文件中只有标准输出的消息。

3. STDERR

STDERR 代表shell的标准错误输出。shell或运行在shell的程序和脚本报错时，生成的错误消息都会被送往这个位置。
在默认情况下，*STDOUT* 和 *STDERR* 指向同一个地方（屏幕），但 *STDERR* 并不会随着 *STDOUT* 的重定向而发生改变。在使用脚本时，我们常常想改变这种情况。

2. 重定向错误

重定向 *STDERR* 和重定向 *STDOUT* 没太大区别，只要在使用重定向符时指定 *STDERR* 文件描述符就可以了。

1. 只重定向错误

STDERR 的文件描述符为2，可以将该文件描述符索引值放在重定向符号之前，只重定向错误消息。注意，两者必须挨着，否则出错。

```
$ ls -al test2 badtestfile 2> test5 #注释，badtestfile文件不存在
-rw-rw-r-- 1 lxc lxc 1589 11月  6 20:04 test2
$ cat test5
ls: 无法访问 'badtestfile': 没有那个文件或目录
# 如你所见，标准输出显示在屏幕上，标准错误输出重定向了到文件中。
```

2. 重定向错误消息和正常输出

如果想重定向错误消息和正常输出，则必须使用两个重定向符号。你需要在重定向符号之前放上需要重定向的 文件描述符，然后让它们指向用于保存数据的输出文件：

```
$ ls -al test2 test5 badtest 2> test6 1> test7 #注释，badtest文件不存在
$ cat test6
ls: 无法访问 'badtest': 没有那个文件或目录
$ cat test7
-rw-rw-r-- 1 lxc lxc 1589 11月  6 20:04 test2
-rw-rw-r-- 1 lxc lxc   60 11月  6 20:26 test5
# 如你所见，标准错误输出被重定向到 test6 文件，而标准输出被重定向到 test7 文件。
```

你也可以将 *STDOUT* 和 *STDERR* 重定向到同一个文件。为此bash shell提供了特殊的重定向符号 `&>`：

```
$ ls -al test2 test5 badtest &> test7 #注释，badtest文件不存在
$ cat test7
ls: 无法访问 'badtest': 没有那个文件或目录
-rw-rw-r-- 1 lxc lxc 1589 11月  6 20:04 test2
-rw-rw-r-- 1 lxc lxc   60 11月  6 20:26 test5
# 如你所见，两者均被重定向到test7文件。
```

> 注意，其中的一条错误消息出现的顺序和预想不同。badtest(列出的最后一个文件)的这条错误消息出现在了输出文件的第一行。这是因为为了避免错误消息散落在输出文件中，相较于标准输出，bash shell自动赋予了错误消息更高的优先级。这样，便于你集中浏览错误消息。

2. 在脚本中重定向输出

在脚本中重定向输出方法有两种：

- 临时重定向一行
- 永久重定向脚本中的所有命令

1. 临时重定向

如果你有意在脚本中生成错误消息，可以将单独的一行输出重定向到 *STDERR* 。这只需要使用输出重定向符号将输出重定向到 *STDERR* 文件描述符。在重定向到文件描述符时，必须在文件描述符索引值之前加一个`&`:

```
echo "This is an err message" >&2
```

来个例子:

test8.sh

```
#!/bin/bash
# Testring STDERR messages

echo "This is an error" >&2
echo "This is normal output"
```

如果你向往常一样运行这个脚本，你看不出任何区别:

```
./test8.sh
This is an error
This is normal output
```

这是因为，**默认情况下， *STDOUT* 和 *STDERR* 指向的位置（屏幕）是一样的。** 但是如果你在运行脚本时重定向了 *STDERR* ，那么脚本中所有送往 *STDERR* 的文本都会被重定向。

```
./test8.sh 2> test9
This is normal output
$ cat test9
This is an error
# 符合预期，标准输出显示在屏幕上，标准错误输出出现在test9文件里。
```

2. 永久重定向

如果脚本中有大量数据需要重定向，那么逐条重定向所有`echo`语句就会很烦琐。这时可以使用`exec`命令，它会启动一个新shell，并在脚本执行期间重定向某个特定文件描述符。

例如:

test10.sh

```
#!/bin/bash
# redirecting all output to a file
exec 1> testout

echo "This is a test of redirecting all output"
echo "from a script to another file."
echo "without having to redirect every individual line."
# output:
# ./test10.sh
# $ cat testout
# This is a test of redirecting all output
# from a script to another file.
# without having to redirect every individual line.
```

test11.sh

```
#!/bin/bash
# Redirecting output to different locations.

exec 2> testerror

echo "This is start of the script."
echo "now redirecting all output to another location."

exec 1> testout

echo "This output should go to the testout file."
echo "but this should go to testerror file" >&2
# output:
# ./test11.sh
# This is start of the script.
# now redirecting all output to another location.
# $ cat testout
# This output should go to the testout file.
# $ cat testerror
# but this should go to testerror file
# 注意观察结果。 在重定向标准输出之前，标准输出是输出在屏幕上的。
```

一旦重定向了 *STDERR* 或者 *STDOUT* 那么就不太容易将其恢复到原先的位置。如果需要在重定向中来回切换，那么请看 15.4节。

3. 在脚本中重定向输入

可以使用与重定向 *STDOUT* 和 *STDERR* 相同的方法，将 *STDIN* 从键盘重定向到其他位置。在Linux系统中，`exec` 命令允许将 *STDIN* 重定向为文件：

```
exec 0< filname
```

来个例子：

test12.sh

```
#!/bin/bash
```

```
# redirecting file input

exec 0< testfile
count=1

while read line
do
   echo "Line #$count: $line"
   count=$[ $count + 1 ]
done
# output:
# ./test12.sh
# Line #1: This is the first line.
# Line #2: This is the second line.
# Line #3: This is the third line.
```

这是从日志文件中读取并处理数据最简单的方法。

4. 创建自己的重定向

前文提到过，在shell中可以打开9个文件描述符。替代性文件描述符从3到8共6个，均可用作输入或输出重定向。这些文件描述符中的任意一个都可以分配给文件并用在脚本中。

1. 创建输出文件描述符

可以用 `exec` 命令分配用于输出的文件描述符。和标准的文件描述符一样，一旦将替代性文件描述符指向文件，此重定向就会一直生效，直至重新分配。

来个例子吧:

test13

```
#!/bin/bash
# Using a alternative file descriptor

exec 3> test13out

echo "This should display on monitor"
echo "and this should be stored in the file" >&3
echo "Then this should be back on the monitor."
# output:
# ./test13.sh
# This should display on monitor
# Then this should be back on the monitor.
# $ cat test13out
# and this should be stored in the file
```

当然你也可以不创建新文件，而是使用 `exec` 命令将数据追加到现有文件：

```
exec 3>>test13out
```

2. 重定向文件描述符

有一个技巧可以帮助你恢复已重定向的文件描述符。你可以将另一个文件描述符分配给标准文件描述符，反之亦可。这意味着可以将 *STDOUT* 的原先位置先重定向到另一个文件描述符，然后再利用该文件描述符恢复 *STDOUT*。

来个例子:

test14.sh

```
#!/bin/bash
# storing STDOUT, then coming back to it

exec 3>&1
exec 1>test14out

echo "This should store in the output file."
echo "alone with this line."

exec 1>&3

echo "Now thing should be back normal."
# output:
# ./test14.sh
# Now thing should be back normal.
# $ cat test14out
# This should store in the output file.
# alone with this line.
# 我们先将3重定向到了1，这意味着任何送往文件描述符3的输出都会出现在屏幕上，
# 然后我们将标准输出重定向到了我们需要的文件中，
# 所以下面的两句echo的输出都出现在test14out文件中，
# 随后我们将1重定向到3（3现在是标准输出），所以后续的echo又输出在了屏幕上（标准输出）
```

3. 创建输入文件描述符

可以采用和重定向输出文件描述符同样的办法来重定向输入文件描述符。在重定向到文件之前，先将 *STDIN* 指向的位置保存到另一个文件描述符，然后在读取完文件之后将 *STDIN* 恢复到原先的位置。

来个例子:

test15.sh

```
#!/bin/bash
# redirecting input file descriptors.

exec 6<&0
exec 0<testfile

count=0
while read line; do
    echo "Line #$count: $line"
    count=$(($count + 1))
done

exec 0<&6
read -p "Are you done now?" answer
```

```
case $answer in
Y | y) echo "GoodBye" ;;
N | n) echo "Sorry, this is the end." ;;
esac
# output:
# ./test15.sh
# Line #0: This is the first line.
# Line #1: This is the second line.
# Line #2: This is the third line.
# Are you done now?y
# GoodBye
```

4. 创建读/写文件描述符

你可以打开单个文件描述符兼做输入和输出，这样就能用同一个文件描述符对文件进行读和写两种操作了。
不过，在使用这种方法时要小心。由于对同一个文件进行读和写两种操作，因此shell会维护一个内部指针，指明文件的当前位置。任何读或者写都会从文件指针上次的位置开始。如果粗心的话，会产生一些意外的结果。

test16.sh

```
#!/bin/bash
# testing input/output file descriptor

exec 3<> testfile

read line <&3
echo "Read line: $line"
echo "This is a test line" >&3
# output:
# $cat testfile
# This is the first line.
# This is the second line.
# This is the third line.
# $ ./test16.sh
# Read line: This is the first line.
# $ cat testfile
# This is the first line.
# This is a test line
# ine.
# This is the third line.
# 注意testfile文件内容的修改。read 命令读取了第一行数据，这使得文件指针指向了第二行数据的第一个字符
# 当echo语句将数据输出到文件时，会将数据写入文件指针的当前位置，覆盖该位置上已有的数据。
```

5. 关闭文件描述符

如果创建了新的输入/输出文件描述符，那么shell会在脚本退出时自动将其关闭。然而在一些情况下，我们需要在脚本结束前手动关闭文件描述符。
要关闭文件描述符，只需将其重定向到特殊符号 `&-` 即可。例如我想关闭文件描述符3：

```
exec 3>&-
```

来个例子：

badtest.sh

```
#!/bin/bash
# testing closing file descriptors

exec 3> test17file

echo "This is a test line of data" >&3

exec 3>&-

echo "This won't work" >&3
# output:
# $./badtest.sh
# ./badtest.sh: 行 10: 3: 错误的文件描述符
# 因为文件描述符3已经关闭，所以报错。
```

> - 一旦关闭了文件描述符，就不能再脚本中向其写入任何数据，否则shell会发出错误消息。
> - 在关闭文件描述符时还要注意另一件事。如果随后你在脚本中又打开了同一个输出文件，那么shell就会用一个新文件来替换已有文件。这意味着如果你输出数据，他就会覆盖已有文件。

test17.sh

```
#!/bin/bash
# testing closing file descriptors

exec 3> test17file
echo "This is a test line of data" >&3
exec 3>&-

cat test17file

exec 3>test17file
echo "This'll be bad" >&3

cat test17file
# output:
# ./test17.sh
# This is a test line of data
# This'll be bad
# 在向test17file文件发送字符串并关闭该文件描述符之后，脚本使用cat命令显示文件内容。
# 到这一步，一切都还好。
# 接下来，脚本重新打开了该输出文件并向它发送了另一个字符串。
# 再显示文件内容的时候，你就只能看到第二个字符串了。shell覆盖了原来的输出文件。
```

5. 列出打开的文件描述符

能用的文件描述符只有9个，你可能会觉得没有什么复杂的。但有时要记住哪个文件描述符被重定向到了哪里就没那么容易了。为了帮你厘清条理，bash shell提供了 `lsof`（**l**ist **o**pened **f**ile）命令。`lsof` 命令会列出整个Linux系统打开的所有文件描述符，这包括所有后台进程以及登录用户打开的文件。

有大量的命令行参数可以过滤 `lsof` 命令的输出。最常用的选项是 `-p` 和 `-d` ，前者允许指定进程ID（PID），后者允许指定要显示的文件描述符编号（多个编号之间以逗号隔开）。
要想知道当前进程的PID。可以使用特殊环境变量 `$$` （shell会将其设为当前PID）。`-a` 选项可用于对另外两个选项的结果执行AND（取交集）运算。

```
lsof -a -p $$ -d 0,1,2
COMMAND  PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
bash    9454  lxc    0u   CHR  136,1      0t0    4 /dev/pts/1
bash    9454  lxc    1u   CHR  136,1      0t0    4 /dev/pts/1
bash    9454  lxc    2u   CHR  136,1      0t0    4 /dev/pts/1
# 显示了当前进程的默认文件描述符。
```

`lsof` 的默认输出中包含多列信息，含义如下表所示：

列	描述
COMMAND	进程对应的命令名的前9个字符
PID	进程的PID
USER	进程属主的登录名
FD	文件描述符编号以及访问类型(*r* 代表可读，*w* 代表可写，*u* 代表读/写)
TYPE	文件的类型（*CHR* 代表字符型，*BLK* 代表块型，*DIR* 代表目录，*REG* 代表常规文件）
DEVICE	设备好（主设备号和从设备号）
SIZE	如果有的话，代表文件的大小
NODE	本地文件的节点号
NAME	文件名

与 *STDIN*、*STDOUT* 和 *STDERR* 关联的文件类型是字符型，因为这三个文件描述符都指向终端，所以输出文件名就是终端的设备名。这3个标准文件描述符都支持读和写（尽管向 *STDIN* 写数据以及从 *STDOUT* 读数据看起来有点奇怪）。

test18.sh

```
#!/bin/bash
# testing lsof with file descriptors

exec 3> test18file1
exec 6> test18file2
exec 7< testfile

lsof -a -p $$ -d 0,1,2,3,6,7
# output:
 ./test18.sh
COMMAND     PID USER   FD   TYPE DEVICE SIZE/OFF    NODE NAME
test18.sh 10474  lxc    0u   CHR  136,1      0t0       4 /dev/pts/1
test18.sh 10474  lxc    1u   CHR  136,1      0t0       4 /dev/pts/1
test18.sh 10474  lxc    2u   CHR  136,1      0t0       4 /dev/pts/1
```

```
15 test18.sh 10474  lxc    3w   REG  259,8        0 2245976
   /home/lxc/scripts/ch15/test18file1
16 test18.sh 10474  lxc    6w   REG  259,8        0 2245978
   /home/lxc/scripts/ch15/test18file2
17 test18.sh 10474  lxc    7r   REG  259,8       73 2250411
   /home/lxc/scripts/ch15/testfile
```

6. 抑制命令输出

有时候，你可能不想显示脚本输出。将脚本作为后台进程运行时这很常见（参见第16章）。如果在后台运行的脚本出现错误消息，那么shell会将其通过邮件发送给进程属主。这会很麻烦，尤其是当运行的脚本输出很多烦琐的小错误时。

要解决这个问题，可以将 *STDERR* 重定向到一个名为`null`文件的特殊文件 (该文件已有讲述，见第12章)跟它的名字很像，null文件里什么都没有。shell输出到null文件的任何数据都不会被保存，全部会被丢弃。

在Linux系统中，null文件的位置是 */dev/null* 。重定向到该位置的任何数据都会被丢弃，不再显示。

```
$ ls -al > /dev/null
$ cat /dev/null
$
```

这是抑制错误消息出现且无须保存它们的一种方法。

也可以在输入重定向中将 */dev/null* 作为输入文件。由于 */dev/null* 文件不包含任何内容，因此我们通常用它来快速清除现有文件中的数据，这样就不用删除文件再重新创建了：

```
$ cat testfile
This is the first line.
This is the second line.
This is the third line.
$ cat /dev/null > testfile
$ cat testfile
$
# testfile文件仍然存在，但现在是一个空文件。
```

这是清除日志文件的常用方法，因为日志文件必须时刻等待应用程序操作

7. 使用临时文件

Linux系统有一个专供临时文件使用的特殊目录 */tmp*，其中存放那些不需要永久保留的文件。大多数Linux发行版配置系统在启动时会自动删除 */tmp* 目录的所有文件。

系统中的任何用户都有权限读写 */tmp* 目录中的文件。这个特性提供了一种创建临时文件的简单方法，而且还无须担心清理工作。

还有一个专门用于创建临时文件的命令 **`mktemp`**，该命令可以直接在 */tmp* 目录中创建唯一临时文件。所创建的临时文件不使用默认的 *umask*（参见第7章）值。作为临时文件属主，你拥有该文件的读写权限，但其他用户无法访问（当然，root用户除外）。

1. 创建本地临时文件

在默认情况下，`mktemp` 会在当前目录中创建一个文件。在使用 `mktemp` 命令时，只需指定一个文件名模板即可。模板可以包含任意文本字符，同时在文件名末尾要加上6个`X`：

```
mktemp testing.XXXXXX
testing.2x7Ykb
ls -al testing.2x7Ykb
-rw------- 1 lxc lxc 0 11月  7 17:14 testing.2x7Ykb
```

`mktemp` 命令会任意地将6个 `X` 替换为同等数量的字符，以保证文件名在目录中是唯一的。`mktemp` 命令输出的就是它所创建的文件名。在脚本中使用 `mktemp` 命令时，可以将文件名保存到变量中，这样就能在随后的脚本中引用了：

test19.sh

```
#!/bin/bash
# creating and using a temp file

tempfile=$(mktemp test19.XXXXXX)

exec 3>$tempfile

echo "This script writes to temp file $tempfile"

echo "This is the first line" >&3
echo "This is the second line." >&3
echo "This is the third line." >&3
exec 3>&-

echo "Done creating temp file. The contains are:"
cat $tempfile

rm -f $tempfile 2>/dev/null
# output:
#  ./test19.sh
# This script writes to temp file test19.WHmkCQ
# Done creating temp file. The contains are:
# This is the first line
# This is the second line.
# This is the third line.
```

2. 在 /tmp 目录中创建临时文件

`-t` 选项会强制 `mktemp` 命令在系统的临时目录中创建文件。在使用这个特性时，`mktemp` 命令返回的是所创建的临时文件的完整路径名，而不是文件名。

test20.sh

```
#!/bin/bash
# creating a temp file in /tmp

tempfile=$(mktemp -t tmp.XXXXXX)

echo "This is a test file." >$tempfile
echo "This is the second line of the test." >>$tempfile

echo "The temp file is located at: $tempfile"
cat $tempfile
rm -f $tempfile
```

3. 创建临时目录

`-d` 选项会告诉 `mktemp` 命令创建一个临时目录。你可以根据需要使用该目录，比如在其中创建其他临时文件。

test21.sh

```
#!/bin/bash
# using a temporary directory

tempdir=$(mktemp -d dir.XXXXXX)
cd $tempdir

tempfile1=$(mktemp temp.XXXXXX)
tempfile2=$(mktemp temp.XXXXXX)

exec 7> $tempfile1
exec 8> $tempfile2

echo "Sending data to directory $tempdir."
echo "This is a test line of data for $tempfile1." >&7
echo "This is a test line of data for $tempfile2." >&8

echo "Done."
```

8. 记录消息

有时候我们需要将输出同时送往显示器和文件。与其对输出进行两次重定向，不如使用 `tee` 命令。`tee` 命令就像是连接管道的T型接头，它能将来自 *STDIN* 的数据同时送往两处。一处是 *STDOUT*，另一处是 `tee` 命令行所指定的文件名。

> `tee` 命令便于将输出同时发往标准输出和日志文件。这样你就可以在屏幕上显示脚本消息的同时将其保存在日志文件中。

命令格式:

```
tee filename
```

来个例子:

```
date | tee testfile
2023年 11月 07日 星期二 17:30:43 CST
cat testfile
2023年 11月 07日 星期二 17:30:43 CST
```

输出出现在了 *STDOUT* 中，同时写入了指定的文件。注意，在默认情况下，`tee` 命令会在每次使用时覆盖指定文件的原先内容。如果想将数据追加到指定文件中，必须使用 `-a` 选项。

```
date | tee -a testfile
2023年 11月 07日 星期二 17:34:00 CST
$ cat testfile
2023年 11月 07日 星期二 17:33:39 CST
2023年 11月 07日 星期二 17:34:00 CST
```

test22.sh

```
#!/bin/bash
# Using the tee command for logging

tempfile=test22file

echo "This is the start of the test." | tee $tempfile
echo "This is the second line of the test." | tee -a $tempfile
echo "This is the end of the test." | tee -a $tempfile
```

9. 实战演练

搞个脚本，读取CSV格式的数据文件，输出SQL INSERT语句。

test23.sh

```
#!/bin/bash
# read file and create INSERT statements for MYSQL

outfile='members.sql'
IFS=','
while read lname fname address city state zip
do
    cat >> $outfile << EOF
    INSERT INTO members(lname, fname, address, city, state, zip) VALUES
('$lname', '$fname', '$address', '$city', '$state', '$zip');
EOF
done < ${1}
```

脚本中出现了3处重定向操作。`while` 循环使用 `read` 语句从数据文件中读取文本。注意 `done` 语句中出现的重定向符号：

```
done < ${1}
```

脚本中另外两处重定向操作出现在同一条语句中：

```
cat >> $outfile << EOF
```

这条语句包含一个输出重定向（追加）和一个内联输入重定向（使用EOF字符串作为起止标志）。输出重定向将 `cat` 命令的输出追加到由 `$outfile` 变量指定的文件中，可以这样看这个语句：

```
cat >> $outfile
```

`cat` 命令的输入使用内联输入重定向，使用EOF字符串作为起止的标志，或许这样看更清晰一些：

```
cat << EOF
INSERT INTO members(lname, fname, address, city, state, zip) VALUES ('$lname',
'$fname', '$address', '$city', '$state', '$zip');
EOF
```

不再解释。

下面是输出：

```
lxc@Lxc:~/scripts/ch15$ ./test23.sh members.sql
lxc@Lxc:~/scripts/ch15$
```

当然，运行脚本时，显示器上不会有任何输出。可以在输出文件 `members.sql` 中查看输出。

ch16 脚本控制

到目前为止，运行脚本的唯一方式就是有需要时直接在命令行启动。当然这并不是唯一的方式，还有很多方式可以用来运行shell脚本。你也可以对脚本加以控制，包括向脚本发送信号、修改脚本的优先级，以及切换脚本的运行模式。本章将逐一介绍这些控制方法。

1. 处理信号

Linux利用信号与系统中的进程进行通信。第4章介绍过不同的Linux信号以及Linux如何用这些信号来停止、启动以及杀死进程。你可以对脚本进行编程，使其在收到特定信号时执行某些命令，从而控制shell脚本的操作。

1. 重温Linux信号

Linux系统和应用程序可以产生超过30个信号。下表列出在shell脚本编程时会遇到的最常见的Linux系统信号。

信号	值	描述
1	SIGHUP	挂起（hang up）进程
2	SIGINT	中断（interrupt）进程
3	SIGQUIT	停止进程（stop）
9	SIGKILL	无条件终止（terminate）进程
15	SIGTERM	尽可能终止进程
18	SIGCONT	继续（continue）运行停止的进程
19	SIGSTOP	无条件停止，但不终止进程

信号	值	描述
20	SIGTSTP	停止或暂停（pause），但不终止进程

在默认情况下bash shell会忽略收到的任何 *SIGQUIT(3)* 和 *SIGTERM(15)* 信号（因此交互式shell才不会被意外终止）。但是bash shell会处理收到的所有 *SIGHUP(1)* 和 *SIGINT(2)* 信号。
如果收到了 *SIGHUP* 信号（比如在离开交互式shell时），bash shell就会退出。但在退出之前，它会将 *SIGHUP* 信号传给所有由该shell启动的进程，包括正在运行的shell脚本。
随着收到 *SIGINT* 信号，shell会被中断。Linux内核将不再为shell分配CPU处理时间。当出现这种情况时，shell会将 *SIGINT* 信号传给由其启动的所有进程，以此告知出现的状况。
你可能也注意到了，shell会将这些信号传给shell脚本来处理。而shell脚本的默认行为是忽略这些信号，因为可能不利于脚本运行。要避免这种情况，可以在脚本中加入识别信号的代码，并做相应的处理。

2. 产生信号

bash shell允许键盘上的两种组合键来生成两种基本的Linux信号。这个特性在需要停止或暂停失控脚本时非常方便。

1. 中断进程

Ctrl+C组合键会生成 *SIGINT* 信号，并将其发送给当前在shell中运行的所有进程。

```
$ sleep 60
^C
$
```

在超时前（60秒）按下Ctrl+C组合键，就可以提前终止 `sleep` 命令

2. 暂停进程

Ctrl+Z组合键可以生成 *SIGTSTP* 信号，停止shell中运行的任何进程。停止（stopping）进程跟终止（terminating）进程不同，前者让程序继续驻留在内存中，还能从上次停止的位置继续运行。16.4节将介绍如何重启一个已经停止的进程。
当使用Ctrl+Z组合键时，shell会通知你进程已经被停止了：

```
$ sleep 60
^Z
[1]+  已停止               sleep 60
```

方括号中分配的数字是 **作业号**。shell将运行的各个进程称为 **作业**，并为作业在当前shell内分配了唯一的作业号。作业号从1开始，然后是2，以此递增。
如果shell会话中有一个已停止的作业，那么在退出shell时，bash会发出提醒：

```
$ sleep 70
^Z
[2]+  已停止               sleep 70
$ exit
exit
有停止的任务。
```

可以用 `ps` 命令来查看已停止的作业：

```
$ ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    5616    5599  0  80   0 -  4495 do_wai pts/1    00:00:00 bash
0 T  1000    6751    5616  0  80   0 -  2791 do_sig pts/1    00:00:00 sleep
0 T  1000    6849    5616  0  80   0 -  2791 do_sig pts/1    00:00:00 sleep
4 R  1000    6994    5616  0  80   0 -  3628 -      pts/1    00:00:00 ps
```

在S列（进程状态）中，`ps` 命令将已停止作业的状态显示为T。这说明命令要么被跟踪，要么被停止。如果在有已停止作业的情况下仍旧想退出，则只需要再输入一边 `exit` 命令即可。shell会退出，终止已停止作业。或者，如果知道已停止作业的PID，那就可以用kill命令发送 *SIGKILL（9）*信号将其终止：

```
$ kill -9 6751
[1]-  已杀死               sleep 60
$ kill -9 6849
[2]+  已杀死               sleep 70
```

每当shell生成命令行提示符时，也会显示shell中状态发生改变的作业。杀死作业后，shell会显示一条消息，表示运行中的作业已被杀死，然后生成提示符。

> **注意:** 在某些Linux系统中，杀死作业时并不会得到任何回应，但当下次执行能让shell生成命令行提示符的操作时（比如，按下Enter键），你会看到一条消息，表示作业已被杀死。

3. 捕获信号

你也可以用其他命令在信号出现时将其捕获，而不是忽略信号。**`trap`** 命令可以指定shell脚本需要侦测并拦截的Linux信号。如果脚本收到了 `trap` 命令中列出的信号，则该信号不再由shell处理，而是由本地处理。

命令格式：

```
trap commands signals
```

在`tarp`命令中，需要在 *commands* 部分列出想要shell执行的命令，在 *signals* 部分列出想要捕获的信号（多个信号之间以空格分隔）。指定信号的时候，可以使用信号的值或者信号名。

来个例子：

trapsignal.sh

```
#!/bin/bash
# Testing signal trapping
#
trap "echo ' Sorry! I have trapped Ctrl-C'" SIGINT
#
echo This is a test script.
#
count=1
while [ $count -le 5 ]
do
    echo "Loop #$count."
    sleep 1
    count=$[ $count + 1 ]
done
#
```

```
echo This is the end of the script.
exit
# output:
 ./trapsignal.sh
This is a test script.
Loop #1.
^C Sorry! I have trapped Ctrl-C
Loop #2.
^C Sorry! I have trapped Ctrl-C
Loop #3.
^C Sorry! I have trapped Ctrl-C
Loop #4.
^C Sorry! I have trapped Ctrl-C
Loop #5.
This is the end of the script.
# 每次使用Ctrl+C组合键，脚本都会执行trap命令中指定的echo语句，而不是忽略信号并让shell停止该脚本。
```

> **警告：** 如果脚本中的命令被信号中断，使用带有指定命令的 `trap` 未必能让被中断的命令继续执行。为了保证脚本中的关键操作不被打断，请使用带有空操作命令的 `trap` 命令以及要捕获的信号列表，例如：
> `trap "" SIGINT`
> 这种形式的 `trap` 命令允许脚本完全忽略 SIGINT 信号，继续执行重要的工作。

4. 捕获脚本退出

除了在shell脚本中捕获信号，也可以在shell脚本退出时捕获信号。这是在shell完成任务时执行命令的一种简便方法。
要捕获shell脚本的退出，只需在 `trap` 命令后加上 `EXIT` 信号即可：

trapexit.sh

```
#!/bin/bash
# Testing exit trapping
#
trap "echo GoodBye..." EXIT
#
count=1
while [ $count -le 5 ]
do
    echo "Loop #$count"
    sleep 1
    count=$[ $count + 1 ]
done
#
exit
# output:
./trapexit.sh
Loop #1
Loop #2
Loop #3
Loop #4
Loop #5
GoodBye...
```

```
# 当脚本运行到正常的退出位置时，触发了EXIT，shell执行了在trap命令中指定的命令。
# 如果提前退出脚本，则依然能捕获到EXIT：
./trapexit.sh
Loop #1
Loop #2
^CGoodBye...
# 因为SIGINT信号并未在trap命令的信号列表中，所以当按下Ctrl+C组合键发送SIGINT信号时，脚本就退出了。但在退出之前已经触发了EXIT，于是shell会执行trap命令。
```

5. 修改或移除信号捕获

要想在脚本的不同位置进行不同的信号捕获，只需重新使用带有新选项的 `trap` 命令即可：

trapmod.sh

```
#!/bin/bash
# Modifying a set trap
#
trap "echo ' Sorry...Ctrl-C is trapped.'" SIGINT
#
count=1
while [ $count -le 3 ]
do
    echo "Loop #$count"
    sleep 1
    count=$[ $count + 1 ]
done
#
trap "echo ' I have modified the trap!'" SIGINT
#
count=1
while [ $count -le 3 ]
do
    echo "Loop #$count"
    sleep 1
    count=$[ $count + 1 ]
done
#
exit
# output:
./trapmod.sh
Loop #1
^C Sorry...Ctrl-C is trapped.
Loop #2
Loop #3
Loop #1
^C I have modified the trap!
Loop #2
Loop #3
# 如你所见，在前三次和后三次循环捕获信号所执行命令产生的消息是不同的。
```

> **提示：** 如果在交互式shell会话中使用 `trap` 命令，可以使用 `trap -p` 查看被捕获的信号。如果什么都没有显示，则说明shell会话按照默认方式处理信号。

也可以移除已设置好的信号捕获。在 `trap` 命令与希望恢复默认行为的信号列表之间加上两个连字符即可。

trapremoval.sh

```
#!/bin/bash
# Removing a set trap
#
trap "echo ' Sorry...Ctrl-C is trapped.'" SIGINT
#
count=1
while [ $count -le 3 ]
do
    echo "Loop #$count"
    sleep 1
    count=$[ $count + 1 ]
done
#
trap -- SIGINT
echo "The trap is now removed."
#
count=1
while [ $count -le 3 ]
do
    echo "Loop #$count"
    sleep 1
    count=$[ $count + 1 ]
done
#
exit
# output:
./trapremoval.sh
Loop #1
^C Sorry...Ctrl-C is trapped.
Loop #2
Loop #3
The trap is now removed.
Loop #1
^C
# 如你所见，在移除信号捕获后，使用Ctrl+C提前终止了脚本的执行。
```

移除信号捕获后，脚本会按照默认行为处理 *SIGINT* 信号，也就是终止脚本运行。

> 也可以在`trap`命令后使用单连字符来恢复信号的默认行为。单连字符和双连字符的效果一样。

2. 以后台模式运行脚本

在后台模式中，进程运行时不和终端会话的 *STDOUT*、*STDIN* 以及 *STDERR* 关联。

1. 后台模式运行脚本

以后台模式运行shell脚本非常简单，只需在脚本名后面加上 `&` 即可：
以后台模式运行shell命令，可参见 第5章。

backgroundscript.sh

```
#!/bin/bash
#Test running in the background
#
count=1
while [ $count -le 5 ]
do
     sleep 1
     count=$[ $count + 1 ]
done
#
exit
# output:
./backgroundscript.sh &
[1] 5073
$
# 在脚本名之后加上&会将脚本与当前shell分离开来，并将脚本作为一个独立的后台进程运行。
```

方括号中的数字1是shell分配给后台进程的作业号，之后的数字（5073）是Linux系统为进程分配的进程ID（PID）。Linux系统中的每个进程都必须有的唯一PID。
一旦显示了这些内容，就会出现新的命令行界面提示符。返回到当前shell，刚才执行的脚本则会以后台模式安全地退出。这时，就可以继续在命令行中输入新的命令了。
当后台进程结束时，终端上会显示一条消息：

```
$
[1]+  已完成               ./backgroundscript.sh
```

其中，指明了作业号、作业状态（Done），以及用于启动该作业的命令（删除了`&`）。

> 注意，当后台进程运行时，它仍然会使用终端显示器来显示 *STDOUT* 和 *STDERR* 消息。

backgroundoutput.sh

```
#!/bin/bash
#Test running in the background
#
echo "Starting the script..."
count=1
while [ $count -le 5 ]
do
     echo "Loop #$count"
     sleep 1
     count=$[ $count + 1 ]
done
#
echo "Script is completed."
exit
# output:
./backgroundoutput.sh &
[1] 5833
Starting the script...
Loop #1
$ # 这里是按了一下Enter键
$ # 这里是按了一下Enter键
Loop #2
```

```
Loop #3
Loop #4
Loop #5
Script is completed.

[1]+  已完成               ./backgroundoutput.sh
# 你会注意到脚本的输出与shell提示符混在了一起，
# 这个时候你如果执行其他命令，则命令的输出也会和脚本的输出混在一起
# 最好是将后台脚本的STDOUT和STDERR进行重定向，避免这种杂乱的输出。
```

2. 运行多个后台作业

在使用命令行提示符的情况下，可以同时启动多个作业：

```
./testAscript.sh &
[1] 6946
This is Test Script #1.
$ ./testBscript.sh &
[2] 6953
This is Test Script #2.
$ ./testCscript.sh &
[3] 6957
$ And... another Test script.

$ ./testDscript.sh &
[4] 6961
$ Then...there was one more Test script.
$ ps
    PID TTY          TIME CMD
   5588 pts/1    00:00:00 bash
   6747 pts/1    00:00:00 sleep
   6946 pts/1    00:00:00 testAscript.sh
   6948 pts/1    00:00:00 sleep
   6953 pts/1    00:00:00 testBscript.sh
   6955 pts/1    00:00:00 sleep
   6957 pts/1    00:00:00 testCscript.sh
   6959 pts/1    00:00:00 sleep
   6961 pts/1    00:00:00 testDscript.sh
   6963 pts/1    00:00:00 sleep
   6982 pts/1    00:00:00 ps
# 通过ps命令可以看到，所有脚本都处于运行状态
```

> **注意：** 在 `ps` 命令的输出中，每一个后台进程都和终端会话（pts/1）终端关联在一起。如果终端会话退出，那么后台进程也会随之退出。
> 注意，本章先前提到过，当要退出终端会话时，如果还有被停止的进程，就会出现警告信息。但如果是后台进程，则只有部分终端仿真器会在退出终端会话前提醒你尚有后台进程在运行。

如果在登出控制台后，仍希望运行在后台模式的脚本继续运行，则需要借助其他手段。下一节讨论实现方法。

3. 在非控制台下运行脚本

有时候，即便退出了终端会话，你也想在终端会话中启动shell脚本，让脚本一直以后台模式运行结束。这可以用 **`nohup`** 命令来实现。

`nohup` 命令能够阻断发给特定进程的 `SIGHUP` 信号。当退出终端会话时，可以避免进程退出。

命令格式：

```
nohup command
```

来个例子：

```
 nohup ./testAscript.sh &
[1] 7511
nohup: 忽略输入并把输出追加到'nohup.out'

# 退出终端后，再进行查询:
ps -elf | grep testA
0 S lxc         7511    2189  0  80   0 -  3145 do_wai 11:28 ?        00:00:00 /bin/bash ./testAscript.sh
0 R lxc         7907    7640  0  80   0 -  3001 -      11:29 pts/1    00:00:00 grep --color=auto testA
# 发现testAscript.sh脚本并未退出
```

和普通后台进程一样，shell会给 *command* 分配一个作业号，Linux系统会为其分配一个PID号。区别在于，当使用 `nohup` 命令时，如果关闭终端会话，则脚本会忽略其发送的 *SIGHUP* 信号。
由于 `nohup` 命令会解除终端与进程之间的关联，因此进程不再同 *STDOUT* 和 *STDERR* 绑定在一起。为了保存该命令产生的输出，`nohup` 命令会自动将 *STDOUT* 和 *STDERR* 产生的消息重定向到一个名为 *nohup.out* 的文件中。

> 注意， *nohup.out* 文件一般在当前目录创建，否则会在 *$HOME* 目录创建。

```
cat nohup.out
This is Test Script #1.
```

> 注意，如果使用 `nohup` 命令运行了另一命令，那么该命令的输出会被追加到已有的 *nohup.out* 文件中。当运行同一目录中的多个命令时，要注意，因为所有的命令输出都会发送到同一个 *nohup.out* 文件中，结果会让人摸不到头脑

借助 `nohup` 命令，可以在无须停止脚本进程的情况下，登出终端会话去完成其他任务，随后可以检查结果。下一节将介绍更为灵活的后台作业管理方法。

4. 作业控制

作业控制 包括启动、停止、杀死、以及恢复作业。通过作业控制，你能完全控制shell环境中所有进程的运行方式。

1. 查看作业

`jobs` 命令允许用户查看的当前正在处理的作业。

jobcontrol.sh

```
./jobcontrol.sh
Script process id: 8162
Loop #1.
^Z
[1]+  已停止               ./jobcontrol.sh
```

```
$ ./jobcontrol.sh > jobcontrol.out &
[2] 8302
$ jobs
[1]+  已停止               ./jobcontrol.sh
[2]-  运行中               ./jobcontrol.sh > jobcontrol.out &
$ jobs -l
[1]+  8162 停止                 ./jobcontrol.sh
[2]-  8302 运行中              ./jobcontrol.sh > jobcontrol.out &
$ kill -9 8162
[1]+  已杀死               ./jobcontrol.sh
```

> **注意：** 关于`jobs`命令输出中的加号与减号。带有加号的作业为 **默认作业**。如果作业控制命令没有指定作业号，则引用的就是该默认作业。
> 带有减号的作业会在默认作业结束后成为下一个默认作业。
> 任何时候，不管shell运行着多少作业，带加号的作业只能有一个，带减号的作业也只能有一个。

`jobs`命令提供了一些命令行选项，如下表：

选项	描述
-l	列出进程的PID以及作业号
-n	只列出上次shell发出通知后状态发生改变的作业
-p	只列出进程的PID
-r	只列出运行中的作业
-s	只列出已停止的作业

2. 重启已停止的作业

在bash作业控制中，可以将已停止的作业作为后台进程或前台进程重启。前台进程会接管当前使用的终端，因此在使用该特性时要小心。
要以后台模式重启作业，可以使用 **`bg`** 命令：

```
./restartjob.sh
^Z
[1]+  已停止               ./restartjob.sh
$ bg
[1]+ ./restartjob.sh &
$ jobs -l
[1]+  6014 运行中               ./restartjob.sh &
# 因为该作业是默认作业（从加号可以看出），所以仅使用bg命令就可以将其以后台模式重启。
```

如果存在多个作业，则需要在`bg`命令后加上作业号，以便于控制。
要以前台模式重启作业，可以使用带有作业号的 **`fg`** 命令：

```
$ jobs
[1]-  已停止               ./restartjob.sh
[2]+  已停止               ./newrestartjob.sh
$ fg 2
./newrestartjob.sh
This is the script's end.
```

因为作业是前台运行的，因此直到该作业完成后，命令行界面的提示符才会出现。

5. 调整谦让度

在多任务操作系统比如（Linux）中，内核负责为每个运行的进程分配CPU时间。**调度优先级**（也称为**谦让度**（nice value））是指内核为进程分配的CPU时间（相对于其他进程）。在Linux系统中，由shell启动的所有进程的调度优先级默认都是相同的。
调度优先级是一个整数值，**取值范围为-20（最高优先级）到+19（最低优先级）**。在默认情况下，bash shell以优先级0来启动所有进程。

> -20（最低值）代表最高优先级，+19代表最低优先级，这很容易记住。只要记住那句俗话 "Nice guys finish last" 即可，越是谦让（值越大）获得的CPU的机会就越低。

1.`nice` 命令

`nice` 命令允许在启动命令时设置其调度优先级。要想让命令以 *更低* 的优先级运行，只需用 `nice` 命令的 `-n` 选项指定优先级即可

```
nice -n 10 ./jobcontrol.sh > jobcontrol.out &
[1] 6935
ps -p 6935 -o pid,ppid,ni,cmd
    PID    PPID  NI CMD
   6935    5329  10 /bin/bash ./jobcontrol.sh
```

`nice` 命令使得脚本以更低的优先级运行，它会阻止普通用户提高命令的优先级。只有root用户或者特权用户才能提高命令的优先级。
`nice` 命令的 `-n` 选项不是必须的，直接在连字符后面跟上优先级也可以：

```
nice -10 ./jobcontrol.sh > jobcontrol.out &
[1] 7628
$ ps -p 7628 -o pid,ppid,ni,cmd
    PID    PPID  NI CMD
   7628    7338  10 /bin/bash ./jobcontrol.sh
```

当然，当要设置的优先级是负数时，这种写法很容易造成混淆，因为出现了双连字符。在这种情况下，最好还是使用 `-n` 选项。

2. `renice` 命令

有时候，你想修改系统中已运行命令的优先级。`renice` 命令可以帮你搞定。它通过指定运行进程的PID来改变其优先级：

```
./jobcontrol.sh > jobcontrol.out &
[1]+  已完成               nice -10 ./jobcontrol.sh > jobcontrol.out
[1] 7798
$ ps -p 7798 -o pid,ppid,ni,cmd
    PID    PPID  NI CMD
   7798    7338   0 /bin/bash ./jobcontrol.sh
$ renice -p 7798 -n 10
renice: invalid priority '-p'
Try 'renice --help' for more information.
$ renice -n 10 -p 7798
7798 (process ID) 旧优先级为 0，新优先级为 10
```

```
$ ps -p 7798 -o pid,ppid,ni,cmd
    PID    PPID  NI CMD
   7798    7338  10 /bin/bash ./jobcontrol.sh
```

`renice` 命令会自动更新运行进程的调度优先级。和 `nice` 命令一样，`renice` 命令对于非特权用户也有一些限制：只能对属主自己的进程使用 `renice` 且只能降低调度优先级。但是root用户和特权用户可以使用`renice` 命令对任意进程的优先级做任意调整。

6. 定时运行脚本

Linux系统提供了多个在预选时间运行脚本的方法：`at` 命令、`cron` 表以及 `anacron`。

1. 使用 `at` 命令调度作业

`at` 命令允许指定Linux系统何时运行脚本。该命令会将作业提交到队列中，指定shell何时运行该作业。`at` 的守护进程 `atd` 在后台运行，在作业列表中检查待运行的作业。
`atd` 守护进程会检查系统的一个特殊目录(通常位于 */var/spool/at* 或 */var/spool/cron/atjobs*)，从中获取 `at` 命令提交的作业。在默认情况下，`atd` 守护进程每隔60秒检查一次这个目录。如果其中有作业，那么 `atd` 守护进程就会查看此作业的运行时间。如果时间跟当前时间一致，就运行此作业。

1. `at` 命令的格式

命令格式：

```
at [-f filename] time
```

`-f` 选项指定用于从中读取命令（脚本文件）的文件名。`time` 选项指定你希望何时运行该作业。指定时间的方式非常灵活。`at` 命令能识别多种时间格式。

- 标准的小时和分钟，比如 10:15
- AM/PM指示符，比如 10:15 PM
- 特定的时间名称，比如 now、noon、midnight、teatime(4:00 p.m.)。
  除了指定运行作业的时间，也可以通过不同的日期格式指定特定的日期。
- 标准日期，比如 MMDDYY、MM/DD/YY、DD.MM.YY
- 文本日期，比如 Jul 4、Dec 25，加不加年份均可。
- 时间增量
  - Now+25 minutes
  - 10:15 PM tomorrow
  - 10:15 + 7 days

> **提示：** `at` 命令可用的日期和时间格式有很多种，具体参见 */usr/share/doc/at/timespec* 文件

在使用 `at` 命令时，该作业会被提交至 **作业队列**。作业队列保存着通过 `at` 命令提交的待处理作业。针对不同的优先级，有52种作业队列。作业队列通常用小写字母a～z和大写字母A～Z来指代，A队列和a队列是两个不同的队列。

> 在几年前，`batch` 命令也能指定脚本的执行时间。这是个很独特的命令，因为它可以安排脚本在系统处于低负载时运行。现在 `batch` 命令只不过是一个脚本而已(*/usr/bin/batch*)，它会调用 `at` 命令将作业提交到b队列中。

作业队列的字母排序越高，此队列中的作业运行优先级就越低（谦让度越大）。在默认情况下，`at` 命令提交的作业会被放入a队列。如果想以较低的优先级运行作业，可以使用 `-q` 选项指定其他的队列。如果相较于其他进程你希望你的作业尽可能少的占用CPU，可以将其放入到z队列。

2. 获取作业的输出

当在Linux系统中运行 `at` 命令时，显示器并不会关联到该作业。Linux系统反而会将提交该作业的用户email地址作为 *STDOUT* 和 *STDERR*。任何送往 *STDOUT* 和 *STDERR* 的输出都会通过邮件系统传给该用户。

来个例子:

tryat.sh

```
#!/bin/bash
# Trying out the at command
#
echo "This script ran at $(date +%B%d,%T)"
echo
echo "This script is using the $SHELL shell."
echo
sleep 5
echo "This is the script's end."
#
exit
# output:
$ at -f tryat.sh now
warning: commands will be executed using /bin/sh
job 8 at Wed Nov  8 20:50:00 2023
```

无需在意 `at` 命令输出的告警信息，因为脚本的第一行是 #!/bin/bash，该命令由bash shell执行。使用email作为 `at` 命令的输出极不方便。`at` 命令通过 `sendmail` 应用程序发送email。如果系统中没有安装sendmail，那就无法获得任何输出。因此在使用 `at` 命令时，最好在脚本中对 *STDOUT* 和 *STDERR* 进行重定向。

tryatout.sh

```
#!/bin/bash
# Trying out the at command redirecting output
#
outfile=$HOME/scripts/ch16/tryat.out
#
echo "This script ran at $(date +%B%d,%T)" > $outfile
echo >> $outfile
echo "This script is using the $SHELL shell." >> $outfile
echo >> $outfile
sleep 5
echo "This is the script's end." >> $outfile
#
exit
# output:
$ at -M -f tryatout.sh now # -M 选项用于禁止给用户发送邮件
warning: commands will be executed using /bin/sh
job 9 at Wed Nov  8 20:57:00 2023
$ cat tryat.out
```

```
This script ran at 十一月08,20:57:27

This script is using the  shell.

This is the script's end.
```

3. 列出等待的作业

`atq` 命令可以查看系统中有哪些作业在等待:

```
at -M -f tryatout.sh teatime
warning: commands will be executed using /bin/sh
job 10 at Thu Nov  9 16:00:00 2023
lxc@Lxc:~/scripts/ch16$ at -M -f tryatout.sh tomorrow
warning: commands will be executed using /bin/sh
job 11 at Thu Nov  9 21:01:00 2023
lxc@Lxc:~/scripts/ch16$ at -M -f tryatout.sh 21:30
warning: commands will be executed using /bin/sh
job 12 at Wed Nov  8 21:30:00 2023
lxc@Lxc:~/scripts/ch16$ at -M -f tryatout.sh now+1hour
warning: commands will be executed using /bin/sh
job 13 at Wed Nov  8 22:01:00 2023
lxc@Lxc:~/scripts/ch16$ atq
12      Wed Nov  8 21:30:00 2023 a lxc
13      Wed Nov  8 22:01:00 2023 a lxc
11      Thu Nov  9 21:01:00 2023 a lxc
10      Thu Nov  9 16:00:00 2023 a lxc
# 作业列表中显示了作业号、系统运行该作业的日期和时间，以及该作业所在的作业队列。
```

4. 删除作业

一旦知道了哪些作业正在作业队列中等待，就可以用 `atrm` 命令删除等待中的作业。指定要删除的作业号即可:

```
lxc@Lxc:~/scripts/ch16$ atq
12      Wed Nov  8 21:30:00 2023 a lxc
13      Wed Nov  8 22:01:00 2023 a lxc
11      Thu Nov  9 21:01:00 2023 a lxc
10      Thu Nov  9 16:00:00 2023 a lxc
lxc@Lxc:~/scripts/ch16$ atrm 10
lxc@Lxc:~/scripts/ch16$ atq
12      Wed Nov  8 21:30:00 2023 a lxc
13      Wed Nov  8 22:01:00 2023 a lxc
11      Thu Nov  9 21:01:00 2023 a lxc
lxc@Lxc:~/scripts/ch16$ atrm 11 12 13
lxc@Lxc:~/scripts/ch16$ atq
```

只能删除自己提交的作业，不能删除其他人的。

2. 调度需要定期运行的脚本

Linux系统使用 `cron` 程序调度需要定期执行的作业。`cron` 在后台运行，并会检查一个特殊的表(**cron时间表**)，从中获知已安排执行的作业。

1. `cron`时间表

`cron` 时间表通过一种特别的格式指定作业何时运行，其格式如下：

```
minutepasthour hourofday dayofmonth month dayofweek command
```

`cron` 时间表允许使用特定值、取值范围（比如1～5）或者通配符（星号）来指定各个字段。

来几个例子：

如果想在每天的10:15分运行一个命令，可以使用如下的时间表字段：

```
15 10 * * * command
```

每周一的下午4:15分执行命令，可以使用军事时间(24小时制):

```
15 16 * * 1 command
```

可以使用三字符的文本值(mon、tue、wed、thu、fri、sat、sun)或数值（0或7代表周日6代表周六）来指定 *dayofweek* 字段。

每个月的第一天的中午12点执行命令：

```
00 12 1 * * command
```

dayofmonth 字段指定的是月份中的日期值(1 ~ 31)。

> **提示：** 如何设置命令在每个月的最后一天执行，因为无法设置一个 *dayofmonth* 值，涵盖所有月份的最后一天。
>
> - 常用的解决方法是加一个 `if-then` 语句，在其中使用 `date` 命令来检查明天的日期是不是某个月份的第一天(01):
>   00 12 28-31 * * if ["$(date +%d -d tomorrow)" = 01] ; then command; fi
>   这行脚本会在每天中午12点检查当天是不是当月的最后一天（28～31），如果是，就由 `cron` 执行 *command*
> - 另一种方法是将 *command* 替换成一个控制脚本（controlling script），在可能是每月最后一天的时候运行。控制脚本包含 `if-then` 语句，用于检查第二天是否为某个月的第一天。如果是，则由控制脚本发出命令，执行必须在当月最后一天执行的内容。

命令列表必须指定要运行的命令或者脚本的完整路径。你可以像在命令行中那样，添加所需的任何选项和重定向符：

```
15 10 * * * /home/lxc/scripts/ch16/backup.sh > backup.out
```

`cron` 程序会以提交作业的用户身份运行该脚本，因此你必须有访问该脚本、命令以及输出文件的权限。

2. 构建 `cron` 时间表

每个用户都可以使用自己的 `cron` 时间表运行已安排好的任务。Linux提供了 **`crontab`** 命令来处理 `cron时间表`。要列出自己的 `cron` 时间表，可以用 `-l` 选项：

```
lxc@Lxc:~/scripts/ch16$ crontab -l
no crontab for lxc
```

在默认情况下，用户的 `cron` 时间表文件并不存在。可以使用 `-e` 选项向 `cron` 时间表添加字段。在添加字段时，`crontab` 命令会启动一个文本编辑器（参见第十章），使用已有的 `cron` 时间表作为文件内容（如果时间表不存在，就是一个空文件）。

3. 浏览 `cron` 目录

如果创建的脚本对于执行时间的精确性要求不高，则用预配置的 `cron` 脚本目录会更方便。预配置的基础目录共有4个： *hourly*、*daily*、*monthly*、*weekly*。

```
lxc@Lxc:~/scripts/ch16$ ls /etc/cron.*ly
/etc/cron.daily:
0anacron  apport  apt-compat  aptitude  bsdmainutils  cracklib-runtime  dpkg
google-chrome  logrotate  man-db  mlocate  popularity-contest  sysstat
update-notifier-common

/etc/cron.hourly:

/etc/cron.monthly:
0anacron

/etc/cron.weekly:
0anacron  apt-xapian-index  man-db  update-notifier-common
```

如果你的脚本每天运行一次，那么将脚本复制到 *daily* 目录， `cron` 就会每天运行它。

4. `anacron` 程序

`cron` 程序唯一的问题是它假定Linux系统是7*24小时运行的。除非你的Linux系统运行在服务器环境，否则这种假设未必成立。
如果某个作业在 `cron` 时间表中设置的时间已到，但这时候Linux系统处于关闭状态，那么该作业就不会运行。当再次启动系统时，`cron` 程序不会再去运行那些错过的作业。为了解决这个问题，许多Linux发行版提供了 **`anacron`** 程序。
如果 `anacron` 程序判断出某个作业错过了设置的运行时间，它会尽快运行该作业。这意味着如果Linux系统关闭了几天，等到再次启动时，原计划在关机期间运行的作业会自动运行。有了 `anacron`，就能确保作业一定能运行，这正是通常使用 `anacron` 代替 `cron` 调度作业的原因。
`anacron` 程序只处理位于 `cron` 目录的程序，比如 */etc/cron.monthly*。它通过时间戳来判断作业是否在正确的时间间隔内运行了。每个 `cron` 目录都有一个时间戳文件，该文件位于 */var/spool/anacron:*

```
lxc@Lxc:~/scripts/ch16$ ls -lF /var/spool/anacron/
总用量 12
-rw------- 1 root root 9 11月  8 16:46 cron.daily
-rw------- 1 root root 9 10月 29 13:11 cron.monthly
-rw------- 1 root root 9 11月  8 16:51 cron.weekly
lxc@Lxc:~/scripts/ch16$ sudo cat /var/spool/anacron/cron.daily
[sudo] lxc 的密码:
20231108
```

`anacron` 程序使用自己的时间表（通常位于 */etc/anacrontab*）来检查作业目录：

```
lxc@Lxc:~/scripts/ch16$ cat /etc/anacrontab
# /etc/anacrontab: configuration file for anacron

# See anacron(8) and anacrontab(5) for details.

SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin
HOME=/root
LOGNAME=root

# These replace cron's entries
1       5       cron.daily      run-parts --report /etc/cron.daily
7       10      cron.weekly     run-parts --report /etc/cron.weekly
@monthly        15      cron.monthly    run-parts --report /etc/cron.monthly
```

`anacron` 时间表的基本格式和 `cron` 时间表略有不同：

```
period delay identifier command
```

period 字段定义了作业的运行频率（以天为单位）。`anacron` 程序使用该字段检查作业的时间戳文件。*delay*字段指定了在系统启动后，`anacron` 程序需要等待多少分钟再开始运行错过的脚本。

> 注意：`anacron` 不会运行位于 */etc/cron.hourly*目录的脚本。这是因为 `anacron` 并不处理执行时间需求少于一天的脚本。

identifier 字段是一个独特的非空字符串，比如 *cron.weekly*。它唯一的作用是标识出现在日志消息和email中的作业。*command* 字段包含了 `run-parts` 程序和一个 `cron` 脚本目录名。`run-parts` 程序负责运行指定目录中的所有脚本。

7. 使用新Shell启动脚本

如果每次用户启动新的bash shell时都能运行相关的脚本，那将会非常方便，因为有时候你希望为shell会话设置某些shell特性，或者希望已经设置了某个文件。
这时可以回想一下当用户登录bash shell时要运行的启动文件（参见第6章）。另外别忘了，不是所有的发行版都包含这些启动文件。基本上，以下所列文件中的第一个文件会被运行，其余的则被忽略（参考第6章）。

- $HOME/.bash_profile
- $HOME/.bash_login
- $HOME/.profile

因此，应该将需要在 登陆时 运行的脚本放在上述的第一个文件中。
每次启动新shell，bash shell都会运行.bashrc文件。对此进行验证，可以使用这种方法：在主目录下的.bashrc文件中加入一条简单的 `echo` 语句，然后启动一个新shell。

.bashrc文件通常也借由某个bash启动文件来运行，因为.bashrc文件会运行两次：一次是当用户登录bash shell时，另一次是当用户启动bash shell时，如果需要某个脚本在两个时刻都运行，则可以将其放入该文件。

8. 实战演练

搞一个脚本，捕获信号，处理信号。

[trapandrun.sh]

```
#!/bin/bash
# Set specified signal traps; then run script in background
#
####################### Check Signals to Trap ######################
#
while getopts S: opt   #Signals to trap listed with -S option
do
     case "$opt" in
          S) # Found the -S option
             signalList="" #Set signalList to null
             #
             for arg in $OPTARG
             do
                  case $arg in
                  1)   #SIGHUP signal is handled
                       signalList=$signalList"SIGHUP "
                  ;;
                  2)   #SIGINT signal is handled
                       signalList=$signalList"SIGINT "
                  ;;
                  20)  #SIGTSTP signal is handled
                       signalList=$signalList"SIGTSTP "
                  ;;
                  *)   #Unknown or unhandled signal
                       echo "Only signals 1 2 and/or 20 are allowed."
                       echo "Exiting script..."
                       exit
                  ;;
                  esac
             done
             ;;
          *) echo 'Usage: -S "Signal(s)" script-to-run-name'
             echo 'Exiting script...'
             exit
             ;;
     esac
     #
done
#
####################### Check Script to Run ######################
#
```

```
shift $[ $OPTIND - 1 ] #Script name should be in parameter
#
if [ -z $@ ]
then
     echo
     echo 'Error: Script name not provided.'
     echo 'Usage: -S "Signal(s)"  script-to-run-name'
     echo 'Exiting script...'
     exit
elif [ -O $@ ] && [ -x $@ ]
then
     scriptToRun=$@
     scriptOutput="$@.out"
else
     echo
     echo "Error: $@ is either not owned by you or not excecutable."
     echo "Exiting..."
     exit
fi
#
########################## Trap and Run ############################
#
echo
echo "Running the $scriptToRun script in background"
echo "while trapping signal(s): $signalList"
echo "Output of script sent to: $scriptOutput"
echo
trap "" $signalList  #Ignore these signals
#
source $scriptToRun > $scriptOutput &  #Run script in background
#
trap -- $signalList  #Set to default behavior
#
######################## Exit script ######################
#
exit
```

下面是输出:

```
lxc@Lxc:~/scripts/ch16$ ./trapandrun.sh -S "1 2 20" testTandR.sh

Running the testTandR.sh script in background
while trapping signal(s): SIGHUP SIGINT SIGTSTP
Output of script sent to: testTandR.sh.out

lxc@Lxc:~/scripts/ch16$ ps
    PID TTY          TIME CMD
   4178 pts/1    00:00:00 bash
   6224 pts/1    00:00:00 trapandrun.sh
   6225 pts/1    00:00:00 sleep
   6231 pts/1    00:00:00 ps
lxc@Lxc:~/scripts/ch16$ kill -1 6224
lxc@Lxc:~/scripts/ch16$ cat testTandR.sh.out
This is a test script.
Loop #1
```

```
Loop #2
lxc@Lxc:~/scripts/ch16$ kill -2 6224
lxc@Lxc:~/scripts/ch16$ cat testTandR.sh.out
This is a test script.
Loop #1
Loop #2
Loop #3
lxc@Lxc:~/scripts/ch16$ kill -20 6224
lxc@Lxc:~/scripts/ch16$ ps
    PID TTY          TIME CMD
   4178 pts/1    00:00:00 bash
   6224 pts/1    00:00:00 trapandrun.sh
   6376 pts/1    00:00:00 sleep
   6395 pts/1    00:00:00 ps
lxc@Lxc:~/scripts/ch16$ cat testTandR.sh.out
This is a test script.
Loop #1
Loop #2
Loop #3
Loop #4
lxc@Lxc:~/scripts/ch16$ cat testTandR.sh.out
This is a test script.
Loop #1
Loop #2
Loop #3
Loop #4
Loop #5
lxc@Lxc:~/scripts/ch16$ cat testTandR.sh.out
This is a test script.
Loop #1
Loop #2
Loop #3
Loop #4
Loop #5
This is the end of test script.
lxc@Lxc:~/scripts/ch16$ ps
    PID TTY          TIME CMD
   4178 pts/1    00:00:00 bash
   6495 pts/1    00:00:00 ps
lxc@Lxc:~/scripts/ch16$
```

在阅读该控制脚本时，有一个地方需要注意：检查文件是否有执行权限并不是必需的。当使用 `source` 命令运行脚本时，就像bash一样，无须在文件中设置执行权限。这种方式与使用 `bash` 运行脚本差不多，只是不会创建子shell。

补充：shell脚本执行命令时，`source` 与 `bash` 的区别：

- bash xxx.sh
- ./xxx.sh(使用shebang)
  以上命令是开启一个新子进程去执行sh脚本。
- source xxx.sh
- . xxx.sh(点号是 `source` 命令的别名，叫做 **点号操作符**（参见第17章）)。
  以上命令是在当前进程来进行运行。

ch17 创建函数

本章将带你逐步了解如何创建自己的shell脚本函数，并演示如何将其用于其他shell脚本。

1. 脚本函数基础

函数 是一个脚本代码块，你可以为其命名并在脚本中的任何位置重用它。每当需要在脚本中使用该代码块时，直接写函数名即可（这叫作 **调用函数**）。

1. 创建函数

在bash shell脚本中创建函数的语法有两种。

- 第一种是使用关键字 `function`，随后跟上分配给该代码块的函数名：

```
function name {
    commands
}
```

name 定义了该函数的唯一名称。脚本中的函数名不能重复。*commands* 是组成函数的一个或多个bash shell命令。调用该函数时，bash shell会依次执行函数内的命令，就像在普通脚本中一样。

- 第二种在bash shell脚本中创建函数的语法更接近其他编程语言中的定义：

```
name () {
    commands
}
```

函数名后的空括号表明正在定义的是一个函数。这种语法的命名规则与第一种语法一样。

2. 使用函数

要在脚本中使用函数，只需像其他shell命令一样写出函数名即可：

test1.sh

```
#!/bin/bash
# Using a function in a script

function func1 {
    echo "This is an example of function."
}

count=1
while [ $count -le 3 ]
do
    func1
    count=$[ $count + 1 ]
done

echo "This is the end of the loop"
func1
echo "Now this is the end of the script"
# output:
```

```
./test1.sh
This is an example of function.
This is an example of function.
This is an example of function.
This is the end of the loop
This is an example of function.
Now this is the end of the script
```

函数不一定要放在shell脚本的最开始部分，但是要注意这种情况。如果试图在函数被定义之前使用它，则出错：

test2.sh

```
#!/bin/bash
# using a function located in the middle of a script

count=1
echo "This line comes before the function definition"

function func1 {
   echo "This is an example of a function"
}

while [ $count -le 5 ]
do
   func1
   count=$[ $count + 1 ]
done
echo "This is the end of the loop"
func2
echo "Now this is the end of the script"

function func2 {
   echo "This is an example of a function"
}
# output:
bash test2.sh
This line comes before the function definition
This is an example of a function
This is an example of a function
This is an example of a function
This is an example of a function
This is an example of a function
This is the end of the loop
test2.sh: 行 17: func2: 未找到命令
Now this is the end of the script
```

另外也要注意函数名。函数名必须是唯一的，否则就会出现问题。如果定义了同名的函数，那么新定义就会覆盖函数原先的定义，而这一切不会有任何错误消息：

test3.sh

```
#!/bin/bash
# Testing use a deplicate function name

```

```
function func1 {
    echo "This is the first definition of the function name"
}

func1

function func1 {
    echo "This is a repeat of the same function name"
}

func1
echo "This is the end of the script."
# output:
./test3.sh
This is the first definition of the function name
This is a repeat of the same function name
This is the end of the script.
```

func1函数最初的定义工作正常，但重新定义该函数后，后续的函数调用会使用第二个定义。

2. 函数返回值

bash shell把函数视为一个小型脚本，运行结束时会返回一个退出状态码（参见第11章）。有3种方法能为函数生成退出状态码。

1. 默认的退出状态码

在默认情况下，函数的退出状态码是函数中最后一个命令返回的退出状态码。函数执行结束后，可以使用标准变量 `$?` 来确定函数的退出状态码：

test4.sh

```
#!/bin/bash
# Testing the exit statue of a function

func1() {
    echo "trying to display a non-existent file"
    ls -lF badile
}

echo "testing the function: "
func1
echo "The exit status is: $?"
# output:
 ./test4.sh
testing the function:
trying to display a non-existent file
ls: 无法访问 'badile': 没有那个文件或目录
The exit status is: 2
```

该函数的退出状态码非0，因为函数中最后一个命令执行失败了。同时，你也无法知道该函数中的其他命令是否执行成功。来看下面这个例子：

test4b.sh

```bash
#!/bin/bash
# testing the exit status of a function

func1 () {
    ls -lF badfile
    echo "This is a test of a bad command."
}

echo "testing the function:"
func1
echo "The exit status is: $?"
# output:
bash test4b.sh
testing the function:
ls: 无法访问 'badfile': 没有那个文件或目录
This is a test of a bad command.
The exit status is: 0
```

这次，由于函数最后一个命令 `echo` 执行成功，因此该函数的退出状态码为0。不过其中的其他命令执行失败。使用函数的默认退出状态码是一种危险的做法。不过，有几种方法可以解决这个问题。

2. 使用 `return` 命令

bash shell会使用 `return` 命令以特定的退出状态码退出函数。`return` 命令允许指定一个整数值作为函数的退出状态码。

test5.sh

```bash
#!/bin/bash
# Using the return command in a funtion

dbl() {
    read -p "Enter a value: " value
    echo "doubling the value"
    return $[ $value * 2 ]
}

dbl
echo "The new value is $?"
# output:
./test5.sh
Enter a value: 122
doubling the value
The new value is 244
```

当使用这种方法从函数中返回值时，一定要小心。为了避免出现问题，牢记以下两个技巧。

- 函数执行一结束就立刻读取返回值
- 退出状态码必须介于0~255

如果在用 `$?` 变量提取函数返回值之前执行了其他命令，那么函数的返回值会丢失。记住：`$?` 变量保存的是最后执行的那个命令的退出状态码。
第二个技巧界定了返回值的取值范围。由于退出状态码必须小于256，因此函数结果也必须为一个小于256的值。大于255的任何数值都会产生错误的值：

```
 ./test5.sh
Enter a value: 200
doubling the value
The new value is 144
```

如果需要返回较大的整数值或字符串，就不能使用 `return` 方法。接下来介绍另一种方法。

3. 使用函数输出

正如可以将命令的输出保存到shell变量中一样，也可以将函数的输出保存到shell变量中。

来个例子：

test5b.sh

```
#!/bin/bash
# Using the echo to return a value

dbl() {
    read -p "Enter a value: " value
    echo $[ $value * 2 ]
}

result=$(dbl)
echo "The new value is $result"
# output:
./test5b.sh
Enter a value: 200
The new value is 400
```

新函数会使用 `echo` 语句来显示计算结果。该脚本会获取 *dbl* 函数的输出，而不是查看退出状态码。
这个例子演示了一个不易察觉的技巧。注意，*dbl* 函数实际上输出了两条消息。`read` 命令输出了一条简短的消息来向用户询问输入值。bash shell脚本非常聪明，并不将其作为 *STDOUT* 输出的一部分，而是直接将其忽略。如果用 `echo` 语句来生成这条消息来询问用户，那么它会与输出值一起被读入shell变量。

> **注意：** 这种方法还可以返回浮点值和字符串，这使其成为一种获取函数返回值的强大方法。

3. 在函数中使用变量

在函数中使用变量时，需要注意它们的定义方式和处理方式。这是shell脚本中常见的错误的根源。

1. 向函数传递参数

17.2节提到过，bash shell会将函数当作小型脚本来对待。这意味着你可以像普通脚本那样向函数传递参数（参见第14章）。
函数可以使用标准的位置变量来表示在命令行中传给函数的任何参数。例如，函数名保存在 `$0` 变量中，函数参数依次保存在 `$1、$2` 等变量中。也可以使用特殊变量 `$#` 来确定传给函数的参数数量。
在脚本中调用函数时，必须将参数和函数名放在同一行，就像下面这样：

```
func1 $value1 10
```

然后函数可以使用 *位置变量* 来获取参数值。

来个例子：

test6.sh

```
#!/bin/bash
# passing parameters to a function

function addem {
   if [ $# -eq 0 ] || [ $# -gt 2 ]
   then
      echo -1
   elif [ $# -eq 1 ]
   then
      echo $[ $1 + $1 ]
   else
      echo $[ $1 + $2 ]
   fi
}

echo -n "Adding 10 and 15: "
value=$(addem 10 15)
echo $value
echo -n "Let\'s try adding just one number: "
value=$(addem 10)
echo $value
echo -n "Now trying adding no numbers: "
value=$(addem)
echo $value
echo -n "Finally, try adding three numbers: "
value=$(addem 10 15 20)
echo $value
# output:
./test6.sh
Adding 10 and 15: 25
Let's try adding just one number: 20
Now trying adding no numbers: -1
Finally, try adding three numbers: -1
```

由于函数使用 *位置变量* 访问函数参数，因此无法直接获取脚本的命令行参数。因此下面的例子无法成功运行：

badtest1.sh

```
#!/bin/bash
# trying to access script parameters inside a function

function badfunc1 {
   echo $[ $1 * $2 ]
}

if [ $# -eq 2 ]
then
   value=$(badfunc1)
   echo "The result is $value"
else
   echo "Usage: badtest1 a b"
fi
```

```
# output:
./badtest1.sh
Usage: badtest1 a b
lxc@Lxc:~/scripts/ch17$ ./badtest1.sh 10 15
./badtest1.sh: 行 5: *  : 语法错误: 需要操作数 (错误符号是 "*  ")
The result is
```

尽管函数使用了 `$1` 变量和 `$2` 变量，但它们和脚本主体中的 *$1* 变量和 *$2* 变量不是一回事。要在函数中使用脚本的命令行参数，必须在调用函数时手动将其传入:

test7.sh

```
#!/bin/bash
# trying to access script parameters inside a function

function func7 {
   echo $[ $1 * $2 ]
}

if [ $# -eq 2 ]
then
   value=$(func7 $1 $2)
   echo "The result is $value"
else
   echo "Usage: badtest1 a b"
fi
# output:
./test7.sh
Usage: badtest1 a b
lxc@Lxc:~/scripts/ch17$ ./test7.sh 10 13
The result is 130
```

在将 *$1* 变量和 *$2* 变量传给函数后，它们就能跟其他变量一样，可供函数使用了。

2. 在函数中处理变量

给shell脚本程序员带来麻烦情况之一就是变量的 **作用域**。作用域是变量的有效区域。在函数中定义的变量与普通变量的作用域不同。也就是说，对脚本的其他部分而言，在函数中定义的变量是无效的。
函数有两种类型的变量。

- 全局变量
- 局部变量

1. 全局变量

全局变量 就是在shell脚本内任何地方都有效的变量。如果在脚本的主体部分定义了一个全局变量，那么就可以在函数内读取它的值。类似的，如果在函数内定义了一个全局变量，那么也可以在脚本的主体部分读取它的值。
在默认情况下，在脚本中定义的任何变量都是全局变量。在函数外定义的变量可以在函数内正常访问。

test8.sh

```
#!/bin/bash
# using a global variable to pass a value

```

```
dbl() {
    value=$[ $value * 2 ]
}

read -p "Enter a value: " value
dbl
echo "The new value is $value"
# output:
./test8.sh
Enter a value: 12
The new value is 24
```

有时我们会在函数内不经意修改全局变量的值，这可能导致我们意外的结果：

badtest2.sh

```
#!/bin/bash
# demonstrating a bad use of variables

function func1 {
   temp=$[ $value + 5 ]
   result=$[ $temp * 2 ]
}

temp=4
value=6

func1
echo "The result is $result"
if [ $temp -gt $value ]
then
   echo "temp is larger"
else
   echo "temp is smaller"
fi
# output:
./badtest2.sh
The result is 22
temp is larger
```

所以，我们可以使用局部变量。

2. 局部变量

无须在函数内使用全局变量，任何在函数内使用的变量都可以被声明为局部变量。为此，只需在变量声明之前加上关键字 `local` 即可：

```
local variable
```

也可以在变量赋值语句中使用 `local` 关键字：

```
local temp=$[ $value + 5 ]
```

`local` 关键字保证了变量仅在该函数中有效。如果函数之外有同名变量，那么shell会保持这两个变量的值互不干扰。这意味着你可以轻松地将函数变量和脚本变量分离开，只共享需要共享的变量：

test9.sh

```
#!/bin/bash
# demonstrating the local keyword

function func1 {
   local temp=$[ $value + 5 ]
   result=$[ $temp * 2 ]
}

temp=4
value=6

func1
echo "The result is $result"
if [ $temp -gt $value ]
then
   echo "temp is larger"
else
   echo "temp is smaller"
fi
# output:
./test9.sh
The result is 22
temp is smaller
```

现在，当你在 *func1* 函数中使用 *$temp* 变量时，该变量的值不会影响到脚本主体中赋给 *$temp* 的值

4. 数组变量和函数

第5章讨论过使用数组在单个变量中保存多个值的高级用法。在函数中使用数组变量有点儿麻烦，需要做一些特殊考虑。

1. 向函数传递数组

向脚本函数传递数组变量的方法有点难以理解。将数组变量当作单个参数传递的话，它不会起作用：

badtest3.sh

```
#!/bin/bash
# trying to pass an array variable

function testit {
   echo "The parameters are: $@"
   thisarray=$1
   echo "The received array is ${thisarray[*]}"
}

myarray=(1 2 3 4 5)
echo "The original array is: ${myarray[*]}"
testit $myarray
# output:
```

```
./badtest3.sh
The original array is: 1 2 3 4 5
The parameters are: 1
The received array is 1
```

如果试图将数组作为函数参数进行传递，则函数只会提取数组变量的第一个元素。
要解决这个问题，必须先将数组变量拆解成多个数组元素，然后将这些数组元素作为函数参数传递。最后在函数内部，将所有的参数重新组合成一个新的数组变量。来看下面的例子：

test10.sh

```
#!/bin/bash
# array variable to function test

function testit {
   local newarray
   newarray=(`echo "$@"`)
   echo "The new array value is: ${newarray[*]}"
}

myarray=(1 2 3 4 5)
echo "The original array is ${myarray[*]}"
testit ${myarray[*]}
# output:
./test10.sh
The original array is 1 2 3 4 5
The new array value is: 1 2 3 4 5
```

在函数内部，数组可以照常使用：

test11.sh

```
#!/bin/bash
# adding values in an array

function addarray {
    local sum=0
    local newarray
    newarray=(`echo "$@"`)
    for value in ${newarray[*]}
    do
        sum=$[ $sum + $value ]
    done
    echo $sum
}

myarray=(1 2 3 4 5)
echo "The original array is ${myarray[*]}"
result=$(addarray ${myarray[*]})
echo "The result is $result"
# output:
lxc@Lxc:~/scripts/ch17$ ./test11.sh
The original array is 1 2 3 4 5
The result is 15
```

2. 从函数返回数组

函数向shell脚本返回数组变量也采用类似的方法。函数先用 `echo` 语句按正确的顺序输出数组的各个元素，然后脚本再将数组元素重组成一个新的数组变量。

test12.sh

```
#!/bin/bash
# returing an array value

function arrayblr {
    local origarray
    local newarray
    local elements
    local i
    origarray=($(echo "$@"))
    newarray=(`echo "$@"`)
    elements=$[ $# - 1 ]
    for(( i = 0; i <= elements; i++ ))
    {
        newarray[$i]=$[ ${origarray[$i]} * 2 ]
    }
    echo ${newarray[*]}
}

myarray=(1 2 3 4 5)
echo "The orignal array is ${myarray[*]}"
arg1=$(echo ${myarray[*]})
result=($(arrayblr $arg1))
echo "The new array is ${result[*]}"
# output:
./test12.sh
The orignal array is 1 2 3 4 5
The new array is 2 4 6 8 10
```

注意，要想使一个变量为数组变量必须在该变量被赋值时使用圆括号包围值。在该脚本中 `result` 变量就是如此。

5. 函数递归

局部变量的函数的一个特性是 **自成体系（self-containment）**。除了获取函数参数外，自成体系的函数不需要任何外部资源。这个特性使得函数可以递归地调用。

来个例子：

test13.sh

```
#!/bin/bash
# using recursion

function factorial {
    if [ $1 -eq 1 ]
    then
        echo 1
    else
```

```
        local temp=$[ $1 - 1 ]
        local result=$(factorial $temp)
        echo $[ $result * $1 ]
    fi
}

read -p "Enetr value: " value
result=$(factorial $value)
echo "The factorial of $value is $result."
# output:
./test13.sh
Enetr value: 10
The factorial of 10 is 3628800.
```

6. 创建库

bash shell允许创建 **库文件**，允许多个脚本文件引用此库文件。

1. 创建库文件的第一步就是创建一个包含所需函数的共用库文件。

来看一个库文件 *myfuncs*，其中定义了3个简单函数：

myfuncs

```
# my script functions

function addem {
   echo $[ $1 + $2 ]
}

function multem {
   echo $[ $1 * $2 ]
}

function divem {
   if [ $2 -ne 0 ]
   then
      echo $[ $1 / $2 ]
   else
      echo -1
   fi
}
```

2. 第二步就是在需要用到这些函数的脚本文件中包含 *myfuncs* 库文件。这里有点小问题：
   问题出在shell的作用域上。和环境变量一样，shell函数仅在定义它的shell会话内有效。如果在shell命令行界面运行 *myfuncs* 脚本，那么shell会创建一个新的shell并在其中运行这个脚本。在这种情况下，以上3个函数会定义在新shell中，当你运行另一个要用到这些函数的脚本时，它们是无法使用的。这同样适用于脚本。如果尝试像普通脚本文件那样运行库文件，那么这3个函数也不会出现在脚本中。

badtest4.sh

```
#!/bin/bash
# using a library file the wrong way
./myfuncs

result=$(addem 10 15)
echo "The result is $result."
# output:
./badtest4.sh
./badtest4.sh: 行 5: addem: 未找到命令
The result is .
```

使用函数库的关键在于 `source` 命令。`source` 命令会在当前shell的上下文中执行命令，而不是创建新shell并在其中执行命令。可以用 `source` 命令在脚本中运行库文件，这样脚本就可以使用库中的函数了。
`source` 命令有个别名，称作 **点号操作符**。要在shell脚本中运行 *myfuncs* 库文件，只需添加下面这一行代码：

```
. ./myfuncs
```

这个例子假定 *myfuncs* 库文件和shell脚本文件位于同一目录(*实际上*，你运行 *test14.sh* 这个脚本时，必须在ch17这个目录下，否则还是同样找不到，这个相对路径是相对于命令执行时的路径的，并不是相对于脚本文件的路径)。如果不是，则需要使用正确路径访问该文件。

test14.sh

```
#!/bin/bash
# using functions defined in a library file
. ./myfuncs

value1=10
value2=5
result1=$(addem $value1 $value2)
result2=$(multem $value1 $value2)
result3=$(divem $value1 $value2)
echo "The result of adding them is: $result1"
echo "The result of multiplying them is: $result2"
echo "The result of dividing them is: $result3"
# output:
./test14.sh
The result of adding them is: 15
The result of multiplying them is: 50
The result of dividing them is: 2
```

7. 在命令行中使用函数

可以用脚本函数执行一些十分复杂的操作，但有时候，在命令行界面直接使用这些函数也很有必要。

1. 在命令行中创建函数

因为shell会解释用户输入的命令，所以可以在命令行中直接定义一个函数。有两种方法。

1. 一种方法是采用单行的方式来定义函数

```
lxc@Lxc:~/scripts$ function divem { echo $[ $1 / $2 ]; }
lxc@Lxc:~/scripts$ divem 100 5
20
```

当你在命令行中定义函数时，必须在每个命令后面加个分号，这样shell就能知道哪里是命令的起止了：

```
lxc@Lxc:~/scripts$ function doubleit { read -p "Enter value: " value; echo $[ $value * 2 ]; }
lxc@Lxc:~/scripts$ doubleit
Enter value: 10
20
```

2. 另一种方法是采用多行方式来定义函数。在定义时，bash shell会采用次提示符来提示输入更多命令。
   使用这种方法，无须在每条命令的末尾放置分号，只需按下回车键即可：

```
lxc@Lxc:~/scripts$ function multem {
> echo $[ $1 * $2 ]
> }
lxc@Lxc:~/scripts$ multem 2 5
10
```

输入函数尾部的花括号后，shell就知道你已经完成函数的定义了。

> **警告：** 在命令行创建函数时要特别小心。如果给函数起了一个跟内建命令或另一个命令相同的名字，那么函数就会覆盖原来的命令。

2. 在 .bashrc 文件中定义函数

在命令行中定义函数的一个明显缺点是，在退出shell时，函数也会消失。我们可以将函数定义在每次新shell启动时都会重新读取该函数的地方。而 *.bashrc* 文件就是这个最佳位置。不管是交互式shell还是从现有shell启动新的shell，bash shell在每次启动时都会在用户主目录中查找这个文件。

1. 直接定义函数

只需将你要定义的函数放在 *.bashrc* 文件的末尾即可。该函数会在下次启动新的bash shell时生效，或者你在当前已经启动的shell中执行一遍 `source ~/.bashrc` 也行。

```
# 一些自定义函数
function addem {
    echo $[ $1 + $2 ]
}
# 该函数来自 ~/.bashrc文件
# 以下来自命令行终端
lxc@Lxc:~/scripts/ch17$ addem 1 2
3
```

2. 源引函数文件

只要是在shell脚本文件中，就可以用 `source` 命令（或其别名，即点号操作符）将库文件中的函数添加到 *.bashrc* 文件中。同样下次启动新的shell时生效，或者你在当前已经启动的shell中执行一边 `source ~/.bashrc` 也行。

更棒的是，shell还会将定义好的函数传给子shell进程，这样一来，这些函数就能够自动用于该shell会话中的任何shell脚本了。你可以写个脚本，试试在不定义或不源引函数的情况下直接使用函数会是什么结果：

test15.sh

```
#!/bin/bash
# using a function defined in the .bashrc file

value1=10
value2=5
result1=$(addem $value1 $value2)
result2=$(multem $value1 $value2)
result3=$(divem $value1 $value2)
echo "The result of adding them is: $result1"
echo "The result of multiplying them is: $result2"
echo "The result of dividing them is: $result3"
# 按照书上所说，输出应该是:
./test15.sh
The result of adding them is: 15
The result of multiplying them is: 50
The result of dividing them is: 2
# 但是当我实践时，只有以source命令执行test15.sh这个脚本时这个脚本才能找到函数的定义，用其他命令均出错（找不到函数定义）:
lxc@Lxc:~/scripts/ch17$ ./test15.sh
./test15.sh: 行 8: addem: 未找到命令
./test15.sh: 行 9: multem: 未找到命令
./test15.sh: 行 10: divem: 未找到命令
The result of adding them is:
The result of multiplying them is:
The result of dividing them is:
lxc@Lxc:~/scripts/ch17$ bash test15.sh
test15.sh: 行 8: addem: 未找到命令
test15.sh: 行 9: multem: 未找到命令
test15.sh: 行 10: divem: 未找到命令
The result of adding them is:
The result of multiplying them is:
The result of dividing them is:
lxc@Lxc:~/scripts/ch17$ source test15.sh
The result of adding them is: 15
The result of multiplying them is: 50
The result of dividing them is: 2
# 只有在使用 source 命令时未出错
```

参考 ch16 README.md 末尾对 `source` 命令和 `bash` 命令的区别。

按照书上所说，`shell 还会将定义好的函数传给子shell进程`，那么即使是用 `bash` 命令执行脚本不应该不出错吗？？？？为啥出错呢。。。。

ch18 图形化桌面环境中的脚本编程

本章将深入介绍一些可以为交互式脚本增添活力的方法，让它们看起来不再那么古板。

1. 创建文本菜单

菜单式脚本通常会清空显示区域，然后显示可用的菜单项列表。用户可以按下与每个菜单项关联的字母或数字来选择相应的选项。
shell脚本菜单的核心是 `case` 命令（参见第12章）。`case` 命令会根据用户在菜单上的选择来执行相应的命令。

1. 创建菜单布局

创建菜单的第一步显然是确定在菜单上显示的元素以及想要显示的布局方式。
在创建菜单之前，最好先清除屏幕上已有的内容。`clear` 命令使用终端会话的终端设置信息（`clear` 命令会查看由环境变量 *TERM* 给出的终端类型，然后在 *terminfo* 数据库中确定如何清除屏幕上的内容，参见 *man clear*）来清除屏幕上的文本。运行 `clear` 命令之后，可以使用 `echo` 命令来显示菜单。
在默认情况下， `echo` 命令只显示可打印文本字符。要在 `echo` 命令中包含非可打印字符（制表符、换行符等），必须加入 `-e` 选项。

来个例子:

```
lxc@Lxc:~/scripts/ch18$ echo -e "1.\tDisplay disk space"
1.      Display disk space
```

2. 创建菜单函数

shell脚本菜单作为一项独立的函数实现起来更为容易。这样你就能创建出简洁、准确且容易理解的 `case` 命令了。为此，你要为每个菜单项创建单独的shell函数。创建菜单函数的第一步是确定脚本要实现的功能，然后将这些功能以函数的形式放在代码中。
通常我们会为还没有实现的函数创建一个 **桩函数** 。桩函数既可以是一个空函数，也可以只包含一个 `echo` 语句，用于说明最终这里需要什么内容。

来个例子:

```
function diskspace {
    clear
    echo "This is where the diskspace commands will go"
}
```

这个桩函数允许在实现某个函数的同时，菜单仍能正常操作，无须等到写出所有函数之后才让菜单投入使用。你会注意到，函数从 `clear` 命令开始。这是为了能在一个干净的屏幕上执行函数，让它不受原先菜单的干扰。
将菜单布局本身作为一个函数来创建，有助于制作shell脚本菜单：

```
function menu {
    clear
    echo
    echo -e "\t\tSys Admin Menu\n"
    echo -e "\t1. Display disk space"
    echo -e "\t2. Display logged on users"
    echo -e "\t3. Display memory usage"
    echo -e "\t0. Exit program\n\n"
    echo -en "\t\tEnter option: "
    read -n 1 option
}
```

这样，任何时候，你都能调用 *menu* 函数来重现菜单。

3. 添加菜单逻辑

现在你已经创建了菜单布局和 *menu* 函数，只需创建程序逻辑将二者结合起来即可。前面提到过，这要使用 `case` 命令。下面展示了菜单中 `case` 命令的典型用法：

```
menu
case $option in
0)
    break;;
1)
    diskspcae;;
2)
    whoseon;;
3)
    memusage;;
*)
    clear
    echo "Sorry, wrong  selection";;
esac
```

4. 整合shell脚本菜单

下面是一个完整的菜单脚本示例：

menu1.sh

```
#!/bin/bash
# simple script menu

function diskspace {
    clear
    df -k
}

function whoseon {
    clear
    who
}

function menusage {
```

```
    clear
    cat /proc/meminfo
}

function menu {
    clear
    echo
    echo -e "\t\tSys Admin Menu\n"
    echo -e "\t1. Display disk space"
    echo -e "\t2. Display logged on users"
    echo -e "\t3. Display memory usage"
    echo -e "\t0. Exit program\n\n"
    echo -en "\t\tEnter option: "
    read -n 1 option
}

while [ 1 ]
do
    menu
    case $option in
    0)
        break;;
    1)
        diskspace;;
    2)
        whoseon;;
    3)
        menusage;;
    *)
        clear
        echo "Sorry, wrong selection";;
    esac
    echo -en "\n\n\t\tHit any key to continue"
    read -n 1 line
done
clear
```

5. 使用 `select` 命令

`select` 命令只需要一个命令就可以创建出菜单，然后获取输入并自动处理。`select` 命令的格式如下：

```
select variable in list
do
    commands
done
```

list 参数是由空格分隔的菜单项列表，该列表构成整个菜单。`select` 命令会将每个列表项显示成一个带编号的菜单项，然后显示一个由 `PS3` 环境变量定义的特殊提示符，指示用户做出选择。

来个例子：

smenu1.sh

```
#!/bin/bash
# using select in the menu

function diskspace {
    clear
    df -k
}

function whoseon {
    clear
    who
}

function menusage {
    clear
    cat /proc/meminfo
}

PS3="Enter option: "
select option in "Display disk space" "Display logged on users" "Display memory usage" "Exit program"
do
    case $option in
    "Exit program")
        break;;
    "Display disk space")
        diskspace;;
    "Display logged on users")
        whoseon;;
    "Display memory usage")
        menusage;;
    *)
        clear
        echo "Sorry, wrong selection";;
    esac
done
clear
# output:
lxc@Lxc:~/scripts/ch18$ ./smenu1.sh
1) Display disk space
2) Display logged on users
3) Display memory usage
4) Exit program
Enter option:
```

`select` 语句中的所有内容必须作为一行出现。在使用 `select` 命令时，存储在指定变量中的值是整个字符串，而不是跟菜单选项相关联的数字。字符串才是要在 `case` 语句中进行比较的内容。

2. 创建文本窗口部件

dialog 软件包最早是 Savio Lam 编写的一款小巧的工具，现在由 Thomas E.Dickey 负责维护。 *dialog* 能够用 ANSI 转义控制字符，在文本环境中创建标准的窗口对话框。你可以轻而易举地将这些对话框融入自己的shell脚本中，以实现与用户的交互。本节将介绍 *dialog* 软件包并演示如何在shell脚本中使用它。

1. *dialog* 软件包

dialog 命令使用命令行选项来决定生成哪种窗口**部件(widget)**。部件是代表某类窗口元素的术语。
dialog 软件包目前支持的部件类型如下所示：

部件名	描述
calendar	提供可选择日期的日历
checklist	显示多个条目，其中每个条目都可以打开或者关闭
form	构建一个带有标签以及文本字段(可以填写内容)的表单
fselect	提供一个文件选择窗口来浏览选择文件
gauge	显示一个进度条，指明已完成的百分比
infobox	显示一条消息，但不用等待回应
inputbox	显示一个文本框，以输入一个文本
inputmenu	提供一个可编辑的菜单
menu	显示一系列可供选择的菜单项
msgbox	显示一条消息，要求用户点选OK按钮
pause	显示一个进度条，指明暂停期间的状态
passwordbox	显示一个文本框，但会隐藏输入的文本
passworform	显示一个带标签和隐藏文本字段的表单
radiolist	提供一组菜单项，但只能选择其中一个
tailbox	用`tail`命令在滚动窗口中显示文件的内容
tailboxbg	与*tailbox*一样，但运行在后台
textbox	在滚动窗口中显示文件的内容
timebox	提供一个选择小时、分钟和秒钟的窗口
yesno	提供一条带有选择Yes按钮和No按钮的简单消息

要在命令行中指定某个特定部件，需要使用双连字符格式：

```
dialog --widget parameters
```

其中 *widget* 是表中的部件名，*parameters* 定义了部件窗口的大小以及部件需要的文本。
每个 *dialog* 部件都提供了两种输出形式。

- 使用 *STDERR*
- 使用退出状态码

dialog 命令的退出状态码能显示出用户选择的按钮。如果选择了Yes按钮或OK按钮，*dialog* 命令就会返回退出状态码0。如果选择了Cancel按钮或No按钮，*dialog* 命令就会返回退出状态码1。可以用 `$?` 变量来确定用户选择了 *dialog* 部件中的那个按钮。
如果部件返回了数据（比如菜单选项），那么 *dialog* 命令会将数据发送到 *STDERR*。*STDERR* 的输出可以重定向到另一个文件或文件描述符：

```
dialog --inputbox "Enter your age:" 10 20 2>age.txt
```

该命令会将文本框中输入的文本重定向到 age.txt 文件。

1. msgbox部件

msgbox 部件使用的命令格式如下：

```
dialog --msgbox text height width
```

text 参数是你想在窗口中显示的字符串。*dialog* 命令会根据由 *height* 参数和 *width* 参数创建的窗口大小来自动换行。如果想在窗口顶部放一个标题，可以使用 `--title` 参数，后面跟上标题文本。

来个例子：

```
lxc@Lxc:~/scripts/ch18$ dialog --title Testing --msgbox "This is a test" 10 20
```

2. yesno部件

来个例子：

```
lxc@Lxc:~/scripts/ch18$ dialog --title "Please answer" --yesno "Is this thing
on?" 10 20
lxc@Lxc:~/scripts/ch18$ echo $?
0
# 如果用户选择了No按钮，那么退出状态码为1；如果用户选择Yes按钮，那么退出状态码为0。
```

3. inputbox部件

inputbox 部件为用户提供了一个简单的文本区域来输入文本字符串。*dialog* 命令会将文本字符串发送到 *STDERR*你必须重定向 *STDERR* 来获取用户的输入。如果选择了Cancel按钮，那么 *dialog* 命令的退出状态码为1;否则0。

```
lxc@Lxc:~/scripts/ch18$ dialog --inputbox "Enter your age:" 10 20 2>age.txt
lxc@Lxc:~/scripts/ch18$ echo $?
0
```

4. textbox部件

textbox 部件是在窗口显示大量信息的好方法。它会生成一个滚动窗口来显示指定文件的内容。

```
lxc@Lxc:~/scripts/ch18$ dialog --textbox /etc/passwd 15 45
```

5. menu部件

menu 部件可以创建一个文本菜单的窗口版本。只要为每个菜单项提供选择标号和文本就行。

```
lxc@Lxc:~/scripts/ch18$ dialog --menu "Sys Admin Menu" 20 30 10 1 "Display
disk space" 2 "Display users" 3 "Display memory usage" 4 "Exit" 2>test.txt
```

第一个参数 *--menu "Sys Admin Menu"* 定义了菜单的标题，后续两个参数20和30定义了菜单窗口的高和宽，第四个参数定义了一次在窗口中显示的菜单项总数。如果还有更多的菜单项，可以用方向键来滚动显示。

如果用户通过按下标号对应的键来选择了某个菜单项，则菜单项会高亮显示但不会被选定。直到用户用鼠标或者Enter键选择了OK按钮，该菜单项才算最终选定。*dialog* 命令会将选定的菜单项文本发送到 *STDERR*。可以根据需要重定向 *STDERR*。

6. fselect部件

命令格式：

```
dialog --title "Select a file" --fselect $HOME/ 10 50 2>file.txt
```

--fselect 之后的第一个参数指定了窗口使用的起始目录位置。*fselect* 部件窗口由左侧的目录列表、右侧的文件列表（显示了选定目录下的所有文件）和含有当前选定文件或目录的简单文本框组成。可以手动在文本框中输入文件名，也可以用目录和文件列表来选定（使用空格键选定文件，将其加入文本框中）。

2. dialog选项

除了标准部件，还可以在 *dialog*命令中定制很多不同的选项。我们已经使用过 *--title* 选项，该选项允许设置出现在窗口顶部的部件标题。
另外还有许多其他选项，可以全面定制窗口的外观和操作。详见书上 P413-415。

3. 在脚本中使用dialog命令

在脚本中使用 *dialog* 时要记住两条规则：

- 如果有Cancel或No按钮，请检查 *dialog* 命令的退出状态码。
- 重定向 *STDERR* 来获取输出值。

menu3.sh

```
#!/bin/bash
# using dialog to create a menu
```

```
3
4  temp=$(mktemp -t test.XXXXXX)
5  temp2=$(mktemp -t test2.XXXXXX)
6
7  function diskspace {
8      df -k > $temp
9      dialog --textbox $temp 20 60
10 }
11
12 function whoseon {
13     who > $temp
14     dialog --textbox $temp 20 50
15 }
16
17 function menusage {
18     cat /proc/meminfo > $temp
19     dialog --textbox $temp 20 50
20 }
21
22 while [ 1 ]
23 do
24     dialog --menu "Sya Admin Menu" 20 30 10 1 "Display disk space" 2 "Display
   users" 3 "Display memory usage" 0 "Exit" 2>$temp2
25 if [ $? -eq 1 ]
26 then
27     break
28 fi
29 selection=$(cat $temp2)
30 case $selection in
31 1)
32     diskspace;;
33 2)
34     whoseon;;
35 3)
36     menusage;;
37 0)
38     break;;
39 *)
40     dialog --msgbox "Sorry, invalid selection" 10 30
41 esac
42 done
43
44 rm -f $temp 2> /dev/null
45 rm -f $temp2 2> /dev/null
```

3. 图形化窗口部件

KDE桌面环境和GNOME桌面环境（参见第一章）都扩展了 *dialog* 命令的思路，提供了可以在各自环境中生成 X Window图形化部件的命令。
本节将介绍 *kdialog* 软件包和 *zenity* 软件包（这句话有一半是假的，你猜），两者分别为KDE桌面和GNOME桌面提供了图形化窗口部件。

1. KDE环境

。。。见书上P417。

2. GNOME环境

GNOME图形化环境支持两种流行的可生成标准窗口的软件包。

- gdialog
- zenity

到目前为止，zenity是大多数GNOME桌面Linux发行版中最常见的软件包（在Ubuntu和Centos中默认安装）。

1. zenity部件

zenity可以使用命令行选项创建不同的窗口部件。下表列出了zenity能够生成的不同部件。
zenity的命令行程序和kdialog程序和dialog程序的工作方式有些不同，zenity的许多部件类型用额外的命令行选项定义，而不是作为某个选项的参数。

选项	描述
--calendar	显示整月日历
--entry	显示文本输入对话窗口
--error	显示错误消息对话窗口
--file-selection	显示完整的路径和文件名对话窗口
--info	显示信息对话窗口
--list	显示多选列表或单选列表对话窗口
--notification	显示通知图标
--progress	显示进度条对话窗口
--question	显示yes/no对话窗口
--scale	显示一个带有滑动条的比例尺对话窗口，可以选择一定范围内的数值
--text-info	显示含有文本的文本框
--warning	显示警告对话窗口

来看几个例子：

当在日历中选择了日期时，zenity命令会将该日期的值发送到 *STDOUT*。

2. 使用zenity

menu5.sh

4. 实战演练

dialog软件包提供了表单功能，但是相当基础，只允许将多个文本框组合到单个窗口中，以此输入多个数据项。`--form` 选项的格式如下:

```
--form text height width formheight [ label y x item y x flen ilen ].....
```

`--form` 选项的各个参数含义如下:

- *text*： 表单顶部的标题
- *height*： 表单窗口的高度
- *width*： 表单窗口的宽度
- *formheight*： 窗口内的表单的高度
- *label*： 表单字段的标签
- *y*： 表单内的标签或字段的Y坐标
- *x*： 表单内的标签或字段的X坐标
- *item*： 分配给表单字段的默认值
- *flen*： 要显示的表单字段的长度
- *ilen*： 可输入表单字段的最大数据长度

newemployee.sh

```
#!/bin/bash
temp=$(mktemp -t record.XXXX)

function newrecord {
    dialog --form "Enter new employee" 19 50 0 \
        "Last name " 1 1 "" 1 15 30 0 \
        "First name " 3 1 "" 3 15 30 0 \
        "Address " 5 1 "" 5 15 30 0 \
        "City " 7 1 "" 7 15 30 0 \
        "State " 9 1 "" 9 15 30 0 \
        "Zip " 11 1 "" 11 15 30 0 2>$temp

    last=$(cat $temp | head -1)
    first=$(cat $temp | head -2 | tail -1)
    address=$(cat $temp | head -3 | tail -1)
    city=$(cat $temp | head -4 | tail -1)
    state=$(cat $temp | head -5 | tail -1)
    zip=$(cat $temp | head -6 | tail -1)
    record="INSERT INTO employees (last, first, address, city, state, zip)
VALUES
('$last', '$first', '$address', '$city', '$state', '$zip');"
    echo $record >>newrecords.txt
}
```

```
23
24 function listrecords {
25     dialog --title "New Data" --textbox data.txt 20 50
26 }
27
28 while [ 1 ]; do
29     dialog --menu "Employee Data" 20 30 5 \
30         1 "Enter new employee" \
31         2 "Display records" \
32         3 "Exit" 2>$temp
33
34     if [ $? -eq 1 ]; then
35         break
36     fi
37
38     selection=$(cat $temp)
39
40     case $selection in
41     1)
42         newrecord
43         ;;
44     2)
45         listrecords
46         ;;
47     3)
48         break
49         ;;
50     *)
51         dialog --msgbox "Invalid selection" 10 30
52         ;;
53     esac
54 done
55
56 rm -f $temp 2>/dev/null
```

ch19 初识sed和gawk

传说中的Linux三剑客之二。

1. 文本处理

本节将介绍Linux中应用最为广泛的命令行编辑器：sed和gawk

1. sed编辑器

sed编辑器被称作 **流编辑器**（stream editor），与普通的交互式文本编辑器（如Vim）截然不同，流编辑器则是根据事先设计好的一组规则编辑数据流。
sed编辑器根据命令来处理数据流中的数据，这些命令要么从命令行输入，要么保存在命令文本文件中。
sed编辑器可以执行以下操作。

1. 从输入中读取一行数据。
2. 根据所提供的编辑器命令匹配数据。

3. 按照命令修改数据流中的数据。
4. 将新的数据输出到 *STDOUT*。

在流编辑器匹配并针对一行数据执行所有命令之后，会读取下一行数据并重复这个过程。在流编辑器处理完数据流中的所有行后，就结束运行。
由于命令是按顺序执行的，因此sed编辑器只需对数据流处理一遍(one pass through)即可完成编辑操作。这使得sed编辑器要比交互式编辑器快的多，并且可以快速完成对数据的自动修改。

sed命令格式如下:

```
sed options script file
```

options 参数允许修改sed命令的行为，下表列出了可用的选项。

选项	描述
-e *commands*	在处理输入时，加入额外的sed命令
-f *file*	在处理输入时，将 *file* 中指定的命令添加到已有的命令中
-n	不产生命令的输出，使用p(print)命令完成输出

script 参数指定了应用于数据流中的单个命令。如果需要多个命令，则要么使用 `-e` 选项在命令行中指定，要么使用 `-f` 选项在单独的文件中指定。本章将介绍一些sed编辑器的基础命令，然后会在第21章介绍另外一些高级命令。

1. 在命令行中定义编辑器命令

在默认情况下，sed编辑器会将指定的命令应用于 *STDIN* 输入流中。因此，可以直接将数据通过管道传入sed编辑器进行处理。

来个例子:

```
lxc@Lxc:~/scripts/ch19$ echo "This is a test" | sed 's/test/big test/'
This is a big test
```

这个例子在sed编辑器中使用了替换(s)命令。替换命令会用斜线间指定的第二个字符串替换第一个字符串。在本例中，*big test* 替换了 *test* 。

```
lxc@Lxc:~/scripts/ch19$ cat data1.txt
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
lxc@Lxc:~/scripts/ch19$ sed 's/dog/cat/' data1.txt
The quick brown fox jumps over the lazy cat.
The quick brown fox jumps over the lazy cat.
The quick brown fox jumps over the lazy cat.
The quick brown fox jumps over the lazy cat.
```

> 重要的是，sed编辑器并 **不会** 修改文本文件的数据。它只是将修改后的数据发送到 *STDOUT*。

2. 在命令行中使用多个编辑器命令

如果要在sed命令行中执行多个命令，可以使用 `-e` 选项。

```
lxc@Lxc:~/scripts/ch19$ sed -e 's/brown/red/; s/dog/cat/' data1.txt
The quick red fox jumps over the lazy cat.
The quick red fox jumps over the lazy cat.
The quick red fox jumps over the lazy cat.
The quick red fox jumps over the lazy cat.
```

两个命令都应用于文件的每一行数据。命令之间必须以分号分隔，**并且在命令末尾和分号之间不能有空格**。
如果不想使用分号，那么也可以用bash shell的次提示符来分割命令。只要输入第一个单引号标示出sed程序脚本（也称作sed编辑器命令列表）的起始，bash就会提示继续输入命令，直到输入了标示结束的单引号。

```
lxc@Lxc:~/scripts/ch19$ sed -e '
> s/brown/red/
> s/fox/toad/
> s/dof/cat/' data1.txt
The quick red toad jumps over the lazy dog.
The quick red toad jumps over the lazy dog.
The quick red toad jumps over the lazy dog.
The quick red toad jumps over the lazy dog.
```

必须记住，要在闭合单引号所在的行结束命令。bash shell一旦发现了闭合单引号，就会执行命令。sed命令会将你指定的所有命令应用于文本文件的每一行。

3. 从文件中读取编辑器命令

如果有大量要执行的sed命令，那么将其放进单独的文件通常会更方便一些。可以在sed命令中用 `-f` 选项来指定文件。

```
lxc@Lxc:~/scripts/ch19$ cat script1.sed
s/brown/red/
s/fox/toad/
s/dog/cat/
lxc@Lxc:~/scripts/ch19$ sed -f script1.sed data1.txt
The quick red toad jumps over the lazy cat.
The quick red toad jumps over the lazy cat.
The quick red toad jumps over the lazy cat.
The quick red toad jumps over the lazy cat.
```

在这种情况下，不用在每条命令后面加分号。sed编辑器知道每一行都是一条单独的命令。和在命令行输入命令一样，sed编辑器会从指定文件中读取命令并应用于文件中的每一行。

> 提示：sed编辑器脚本文件很容易与bash shell脚本文件混淆。为了避免这种情况，可以使用.sed作为sed脚本文件的扩展名。

2. gawk编辑器

gawk是Unix中最初的awk的GNU版本。gawk比sed的流编辑器提升了一个段位，它提供了一种编程语言，而不仅仅是编辑器命令。在gawk编程语言中，可以实现以下操作。

- 定义变量来保存数据。
- 使用算术和字符串运算来处理数据。
- 使用结构化编程概念（比如 *if-then* 语句和循环）为数据处理添加处理逻辑。
- 提取文件中的数据并将其重新排列组合，最后生成格式化报告。

gawk的报告生成能力多用于从大文本文件中提取数据并将其格式化为可读性报告。最完美的应用案例是格式化日志文件。

1.gawk的命令格式

gawk的基本格式如下:

```
gawk options program file
```

下表列出了gawk的可用选项。

选项	描述
-F *fs*	指定行中划分数据字段的字段分隔符
-f *file*	从指定文件中读取gawk脚本代码
-v *var=value*	定义gawk脚本中的变量及其默认值
-L *[keyword]*	指定gawk的兼容模式或警告级别

2. 从命令行读取gawk脚本

gawk脚本用一对花括号来定义。必须将脚本命令放到一对花括号之间。gawk命令行假定脚本是单个文本字符串，因此还必须将脚本放到单引号中。

来个例子:

```
lxc@Lxc:~/scripts/ch19$ gawk '{print "Hello World"}'
# 这里按下了Enter键
Hello World
# 这里按下了Enter键
Hello World
我输入了一行文本并按下了Enter键
Hello World
```

这个脚本定义了一个命令：`print`。该命令会将文本打印到 *STDOUT*。如果运行这个脚本，什么都不会发生。因为没有在命令行中指定文件名，因此gawk程序会从 *STDIN* 接受数据。在脚本运行时，它会一直等待来自 *STDIN* 的文本，如果你输入一行文本并按下Enter键，则gawk脚本会对这行文本执行一遍脚本。和sed编辑器一样gawk会对数据流中的每一行文本都会执行脚本。所以，你每输入一行文本，都会打印 *Hello World* 的输出。
要终止这个gawk程序，必须表明数据流已经结束了。bash shell提供了Ctrl+D组合键来生成 EOF 字符。使用该组合键可以终止gawk程序并返回到命令行界面。

3. 使用数据字段变量

gawk的主要特性之一是处理文本文件中的数据。它会自动为每一行的各个数据元素分配一个变量。在默认情况下，gawk会将下列变量分配给文本行中的数据字段。

- `$0` 代表整行文本
- `$1` 代表文本行中的第一个数据字段
- `$2` 代表文本行中的第二个数据字段
- `$n` 代表文本行中的第n个数据字段

文本行中的数据字段是通过 **字段分隔符** 来划分的。在读取一行文本时，gawk会用预先定义好的字段分隔符划分出各个数据字段。在默认情况下，字段分隔符是任意的空白字符(比如空格、制表符等)

来几个例子:

```
lxc@Lxc:~/scripts/ch19$ cat data2.txt
One line of the test text.
Two lines of the test text.
Three lines of the test text.
lxc@Lxc:~/scripts/ch19$ gawk '{print $1}' data2.txt
One
Two
Three
```

这个例子中，gawk脚本会读取文本文件，只显示第一个数据字段的值。该脚本使用 `$1` 字段变量来显示每行文本的第一个数据字段。

如果要读取的文件采用了其他的字段分隔符，可以通过 `-F` 选项指定:

```
lxc@Lxc:~/scripts/ch19$ gawk -F: '{print $1}' /etc/passwd
root
daemon
bin
sys
...
```

由于 */etc/passwd* 文件使用冒号来分隔数据字段，因此想要划出数据字段，就必须在gawk选项中将冒号指定为字段分隔符(-F:)。

4. 在脚本中使用多条命令

gawk编程语言允许将多条命令组合成一个常规脚本。要在命令行指定脚本使用多条命令，只需在命令之间加入分号即可。

```
lxc@Lxc:~/scripts/ch19$ echo "My name is lxc" | gawk '{$4="lxcYYDS"; print$0}'
My name is lxcYYDS
```

第一条命令为字段 `$4` 赋值。第二条命令会打印整个文本行。注意，gawk在输出中已经将原文中的第四个数据字段替换成了新值。

当然也可以一次一行地输入脚本命令:

```
lxc@Lxc:~/scripts/ch19$ gawk '{
> $4="lxcYYDS"
> print $0
> }'
My name is lxc
My name is lxcYYDS
Your name is son
Your name is lxcYYDS
```

因为没有在命令行中指定文件名，所以gawk程序会从 *STDIN* 中获取数据。当运行这个脚本时，它会等着读取来自 *STDIN* 的文本。要退出的话，只需按下 Ctrl+D 组合键表明数据结束即可。

5. 从文件中读取脚本

跟sed编辑器一样，gawk允许将脚本保存在文件中，然后在命令行中引用脚本。

```
lxc@Lxc:~/scripts/ch19$ cat script2.gawk
{print $1 "'s home is " $6}
lxc@Lxc:~/scripts/ch19$ gawk -F: -f script2.gawk /etc/passwd
root's home is /root
daemon's home is /usr/sbin
bin's home is /bin
sys's home is /dev
sync's home is /bin
....
```

当然也可以在脚本文件中指定多条命令。为此，只需一行写一条命令即可，且无须加分号。

```
lxc@Lxc:~/scripts/ch19$ cat script3.gawk
{
text = "'s home is "
print $1 text $6
}
lxc@Lxc:~/scripts/ch19$ gawk -F: -f script3.gawk /etc/passwd
root's home is /root
daemon's home is /usr/sbin
bin's home is /bin
sys's home is /dev
....
```

注意在gawk脚本中，引用变量值时无须像shell脚本那样使用美元符号。

6. 在处理数据前运行脚本

gawk还允许指定脚本何时运行。在默认情况，gawk会从输入读取一行文本，然后对这一行数据执行脚本。但在有些时候，可能需要在处理数据前先运行脚本，比如要为报告创建一个标题。`BEGIN` 关键字就是用来做这个的。它会强制gawk在读取数据前执行 `BEGIN` 关键字之后指定的脚本：

```
$ gawk 'BEGIN {print "Hello World"}'
Hello World
```

这次 `print` 命令会在读取数据前显示文本。但在显示过文本之后，脚本就直接结束了，不等待任何数据。
原因在于 `BEGIN` 关键字在处理任何数据之前仅应用指定的脚本。如果想使用正常的脚本来处理数据，则必须用另一个区域来定义脚本：

```
lxc@Lxc:~/scripts/ch19$ cat data3.txt
Line 1
Line 2
Line 3
lxc@Lxc:~/scripts/ch19$ gawk 'BEGIN {print "The date3 file Contents: "}
> {print $0}' data3.txt
The date3 file Contents:
Line 1
Line 2
Line 3
```

7. 在处理数据后运行脚本

和 `BEGIN` 关键字类似， `END` 关键字允许指定一段脚本，gawk会在处理完数据后执行这段脚本。

```
lxc@Lxc:~/scripts/ch19$ gawk 'BEGIN {print "The data3 Dile Contents:"}
> {print $0}
> END {print "End of File"}' data3.txt
The data3 Dile Contents:
Line 1
Line 2
Line 3
End of File
```

gawk在打印完文件内容后，会执行 `END` 脚本中的命令。这是在处理完所有正常数据后给报告添加页脚的最佳方法。

来个例子：

script4.gawk

```
BEGIN {
print "The latest list of users and shells"
print "UerID   \t Shell"
print "-----   \t -----"
FS=":"
}

{
print $1 "       \t " $7
}

END {
print "The concludes the listing"
}
# output:
lxc@Lxc:~/scripts/ch19$ gawk -f script4.gawk /etc/passwd
The latest list of users and shells
UerID            Shell
```

```
-----               -----
root                /bin/bash
daemon              /usr/sbin/nologin
bin                 /usr/sbin/nologin
....
nobody              /usr/sbin/nologin
The concludes the listing
```

`print` 命令中的 `-t` 负责生成美观的 **选项卡式输出（tabbed output）**。

2. sed编辑器基础命令

1. 更多的替换选项

前面已经讲过如何用替换命令在文本行替换文本。这个命令还有另外一些能简化操作的选项。

1. 替换标志

替换命令在替换多行中的文本时也能正常工作，但在默认情况下它只替换每行中出现的第一处匹配文本。要想替换每行中所有的匹配文本，必须使用 **替换标志(substitution flag)** 。替换标志在替换命令字符串之后设置。

```
s/pattern/replacement/flags
```

有4种可用的替换标志。

- 数字，指明新文本将替换行中的第几处匹配。
- g，指明新文本将替换行中所有的匹配。
- p，指明打印出替换后的行。
- w *file* ，将替换的结果写入文件。

第一种替换表示，你可以告诉sed编辑器用新文本替换第几处匹配文本：

```
lxc@Lxc:~/scripts/ch19$ cat data4.txt
This is a test of the test script.
This is the second test of the test script.

lxc@Lxc:~/scripts/ch19$ sed 's/test/trial/2' data4.txt
This is a test of the trial script.
This is the second test of the trial script.
# 如你所见，替换了每行中第二处匹配文本。
```

替换标志 `g` 可以替换文本行中所有的匹配文本。

```
lxc@Lxc:~/scripts/ch19$ cat data4.txt
This is a test of the test script.
This is the second test of the test script.

lxc@Lxc:~/scripts/ch19$ sed 's/test/trial/g' data4.txt
This is a trial of the trial script.
This is the second trial of the trial script.
```

替换标志 `p` 会打印出包含替换命令中指定匹配模式的文本行。该标志通常和sed的 `-n` 选项配合使用。

```
lxc@Lxc:~/scripts/ch19$ cat data5.txt
This is a test line.
This is a different line.

lxc@Lxc:~/scripts/ch19$ sed -n 's/test/trial/p' data5.txt
This is a trial line.
```

`-n` 选项会抑制sed编辑器的输出，而替换标志 `p` 会输出替换后的行。将二者配合使用的结果就是只输出被替换命令修改过的行。

替换标志 `-w` 会产生同样的输出，不过会将输出保存到指定文件中。

```
lxc@Lxc:~/scripts/ch19$ sed  's/test/trial/w test.txt' data5.txt
This is a trial line.
This is a different line.

lxc@Lxc:~/scripts/ch19$ cat test.txt
This is a trial line.
```

sed编辑器的正常输出会被保存在 *STDOUT* 中，只有那些包含匹配模式的行才会被保存在指定的输出文件中。

2. 使用地址

在默认情况下，在sed编辑器中使用的命令会应用于所有的文本行。如果只想将命令应用于特定的某一行或某些行，则必须使用 **行寻址**。
在sed编辑器中有两种形式的行寻址。

- 以数字形式表示的行区间。
- 匹配行内文本的模式。

以上两种形式使用相同的格式来指定地址。

```
[address]command
```

也可以将针对特定地址的多个命令分组：

```
address {
    command1
    command2
    command3
}
```

sed编辑器会将指定的各个命令应用于匹配指定地址的文本行。

1. 数字形式的行寻址

在使用数字形式的行寻址时，可以用行号来引用文本流中的特定行。sed编辑器会将文本流中的第一行编号为1，第二行编号为2，以此类推。
在命令行中指定的行地址可以是单个行号，也可以是用起始行号、逗号以及结尾行号指定的行区间。

来几个例子：

```
lxc@Lxc:~/scripts/ch19$ cat data1.txt
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.

lxc@Lxc:~/scripts/ch19$ sed '2s/dog/cat/' data1.txt
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy cat.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
```

sed编辑器只修改了地址所指定的第二行文本。

```
lxc@Lxc:~/scripts/ch19$ sed '2,3s/dog/cat/' data1.txt
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy cat.
The quick brown fox jumps over the lazy cat.
The quick brown fox jumps over the lazy dog.
```

这次使用了行区间，替换了第2、3行。

如果想将命令应用于从某行开始到结尾的所有行，可以使用美元符号作为结尾行号。

```
lxc@Lxc:~/scripts/ch19$ sed '2,$s/dog/cat/' data1.txt
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy cat.
The quick brown fox jumps over the lazy cat.
The quick brown fox jumps over the lazy cat.
```

如你所见，替换了从第2行开始到最后一行。因为往往不知道文本中有多少行，所以美元符号用起来很方便。

2. 使用文本模式过滤

另一种限制命令应用于哪些行的方法略显复杂。sed编辑器允许指定文本模式来过滤出命令所应用的行，其格式如下：

```
/pattern/command
```

必须将指定的模式放入正斜线内。sed编辑器会将该命令应用于包含匹配模式的行。

来个例子：

```
$ grep /bin/bash /etc/passwd
lxc:x:1000:1000:Lxc,,,:/home/lxc:/bin/bash

lxc@Lxc:~/scripts/ch19$ sed '/lxc/s/bash/csh/' /etc/passwd
lxc:x:1000:1000:Lxc,,,:/home/lxc:/bin/csh
....
```

如你所见，该命令只应用于包含匹配模式的行。虽然使用固定的文本模式有助于过滤出特定的值，就跟上面的例子一样，但难免有所局限。sed编辑器在文本模式中引入了正则表达式（见第20章）来创建匹配效果更好的模式。

3. 命令组

如果需要在单行中执行多条命令，可以用花括号将其组合在一起，sed编辑器会执行匹配地址中列出的所有命令。

```
lxc@Lxc:~/scripts/ch19$ sed '2{
> s/fox/toad/
> s/dog/cat/
> }' data1.txt
The quick brown fox jumps over the lazy dog.
The quick brown toad jumps over the lazy cat.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
```

这两条命令都会应用于该地址。当然，也可以在一组命令前指定区间。

```
lxc@Lxc:~/scripts/ch19$ sed '2,${
> s/brown/red/
> s/fox/toad/
> s/lazy/sleeping/
> }' data1.txt
The quick brown fox jumps over the lazy dog.
The quick red toad jumps over the sleeping dog.
The quick red toad jumps over the sleeping dog.
The quick red toad jumps over the sleeping dog.
# 如你所见，我就不用解释了吧。
```

3. 删除行

如果需要删除文本流中的特定行，可以使用删除命令(d)。
删除命令很简单，它会删除匹配指定模式的所有行。使用该命令时要小心，如果忘记加入寻址模式，则流中的所有文本行都会被删除：

```
lxc@Lxc:~/scripts/ch19$ cat data1.txt
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.

lxc@Lxc:~/scripts/ch19$ sed 'd' data1.txt
# 都删除，当然没输出啦。
```

当和指定地址一起使用时，删除命令显然能发挥出最大的功用。可以从数据流中删除特定的文本行，这些文本行要么通过行号指定：

```
lxc@Lxc:~/scripts/ch19$ cat data6.txt
This is line number 1.
This is line number 2.
This is the 3rd line.
This is the 4th line.

lxc@Lxc:~/scripts/ch19$ sed '3d' data6.txt
This is line number 1.
This is line number 2.
This is the 4th line.
```

要么通过特定行区间指定：

```
lxc@Lxc:~/scripts/ch19$ sed '2,3d' data6.txt
This is line number 1.
This is the 4th line.
```

要么通过特殊的末行字符指定：

```
lxc@Lxc:~/scripts/ch19$ sed '3,$d' data6.txt
This is line number 1.
This is line number 2.
```

当然模式匹配特性也适用于删除命令：

```
lxc@Lxc:~/scripts/ch19$ sed '/number 1/d' data6.txt
This is line number 2.
This is the 3rd line.
This is the 4th line.
```

sed编辑器会删除掉与指定模式相匹配的文本行。

> **注意：** sed编辑器 **不会修改原始文件**。你删除的行只是从sed编辑器的输出中消失了。

也可以使用两个文本模式来删除某个区间内的行。但这么做时要小心，你指定的第一个模式会启用行删除功能，第二个模式会关闭行删除功能，而sed编辑器会删除两个指定行之间的所有行（当然包括指定的行）：

```
lxc@Lxc:~/scripts/ch19$ sed '/1/,/3/d' data6.txt
This is the 4th line.
```

除此之外，要特别小心，因为只要sed编辑器在数据流中匹配到了开始模式，就会启用删除功能，如果没有找到停止模式，就会一直执行删除，直到尾行。

```
lxc@Lxc:~/scripts/ch19$ cat data7.txt
This is line number 1.
This is line number 2.
This is the 3rd line.
This is the 4th line.
This is line number 1 again; we want to keep it.
This is more text we want to keep.
Last line in the file; we want to keep it.

lxc@Lxc:~/scripts/ch19$ sed '/1/,/3/d' data7.txt
This is the 4th line.
```

第二个包含数字 "1" 的行再次触发了删除指令，因为没有找到停止模式，所以数据流中的剩余文本行全部被删除了。当然，如果指定的停止模式始终未在文本中出现，就会出现删除到尾行的情况。

```
lxc@Lxc:~/scripts/ch19$ sed '/3/,/5/d' data7.txt
This is line number 1.
This is line number 2.
```

删除功能在匹配到开始模式的时候就启用了，但由于一直未能匹配到结束模式，因此没有关闭，最终整个数据流都被删除了。

4. 插入和附加文本

sed编辑器也可以向数据流中插入和附加文本行。

- 插入(insert)(i)命令会在指定行 **前** 增加一行。
- 附加(append)(a)命令会在指定行 **后** 增加一行。

这两条命令不能在单个命令行中使用。必须指定是将行插入还是附加到另一行，其格式如下：

```
sed '[address]command\
new line'
```

new line 中的文本会出现在你所指定的sed编辑器的输出位置。当使用插入命令时，文本会出现在数据流文本之前：

```
lxc@Lxc:~/scripts/ch19$ echo "Test Line 2" | sed 'i\Test Line 1'
Test Line 1
Test Line 2
```

当使用附加命令时，文本会出现在数据流文本之后：

```
lxc@Lxc:~/scripts/ch19$ echo "Test Line 2" | sed 'a\Test Line 1'
Test Line 2
Test Line 1
```

在命令行界面使用sed编辑器时，你会看到次提示符，它会提醒输入新一行的数据。必须在此行完成sed编辑器命令。一旦输入表示结尾的后单引号，bash shell就会执行该命令：

```
lxc@Lxc:~/scripts/ch19$ echo "Test Line 2" | sed 'i\
> Test Line 1'
Test Line 1
Test Line 2
```

要向数据流内部插入或附加数据，必须用地址告诉sed编辑器希望数据出现在什么位置。用这些命令时只能指定一个行地址。使用行号或文本模式都行，但不能用行区间。这也说的通，因为只能将文本插入或附加到某一行而不是行区间的前面或后面。

来个例子：

```
lxc@Lxc:~/scripts/ch19$ cat data6.txt
This is line number 1.
This is line number 2.
This is the 3rd line.
This is the 4th line.

lxc@Lxc:~/scripts/ch19$ sed '3i\
> This is an inserted line.
> ' data6.txt
This is line number 1.
This is line number 2.
This is an inserted line.
This is the 3rd line.
This is the 4th line.
# 如你所见，这个例子将新文本行插入到第3行之前。
```

下面这个例子将新行附加到数据流中的第三行之后：

```
lxc@Lxc:~/scripts/ch19$ sed '3a\
> This is an insertetd line.
> ' data6.txt
This is line number 1.
This is line number 2.
This is the 3rd line.
This is an insertetd line.
This is the 4th line.
```

同样，如果你想将新行附加到数据流的末尾，那么只需用代表数据流最后一行的美元符号即可：

```
lxc@Lxc:~/scripts/ch19$ sed '$a\
> This line was added to the end of the file.
> ' data6.txt
This is line number 1.
This is line number 2.
This is the 3rd line.
This is the 4th line.
This line was added to the end of the file.
```

当然，如果你想在第一行之前增加一个新行。这只要在第一行之前插入新行就可以。

要插入或附加多行文本，必须在要插入或附加的每行新文本末尾使用反斜线：

```
lxc@Lxc:~/scripts/ch19$ sed '1i\
> This is an inserted line.\
> This is another inserted line.
> ' data6.txt
This is an inserted line.
This is another inserted line.
This is line number 1.
This is line number 2.
This is the 3rd line.
This is the 4th line.
```

5. 修改行

修改(c)命令允许修改数据流中整行文本的内容。它跟插入和附加命令的工作机制一样，必须在sed命令中单独指定一行：

```
$ sed '2c\
> This is a changed line of text.
> ' data6.txt
This is line number 1.
This is a changed line of text.
This is the 3rd line.
This is the 4th line.
```

也可以用文本模式来寻址，这个例子中，sed编辑器会修改第3行文本：

```
$ sed '/3rd line/c\
> This is a changed line of text.
> ' data6.txt
This is line number 1.
This is line number 2.
This is a changed line of text.
This is the 4th line.
```

文本模式会修改所匹配到的任意文本行：

```
$ cat data8.txt
I have 2 Infinity Stones
I need 4 more Infinity Stones
I have 6 Infinity Stones!
I need 4 Infinity Stones
I have 6 Infinity Stones...
I want 1 more Infinity Stone

$ sed '/have 6 Infinity Stones/c\
> Snap! This is changed line of text.
> ' data8.txt
I have 2 Infinity Stones
I need 4 more Infinity Stones
Snap! This is changed line of text.
I need 4 Infinity Stones
Snap! This is changed line of text.
I want 1 more Infinity Stone
```

可以在修改命令中使用地址区间，但sed编辑器会用指定的文本替换指定地址区间的文本，而不是逐一修改：

```
$ cat data6.txt
This is line number 1.
This is line number 2.
This is the 3rd line.
This is the 4th line.

lxc@Lxc:~/scripts/ch19$ sed '2,3c\
> This is a changed line of text.
> ' data6.txt
This is line number 1.
This is a changed line of text.
This is the 4th line.
```

6. 转换命令

转换（y）命令是唯一一个可以处理单个字符的sed编辑器命令。

命令格式：

```
[address]y/inchars/outchars/
```

转换命令会对 *inchars* 和 *outchars* 进行一对一的映射。 *inchars* 中的第一个字符会被转换成为 *outchars* 中的第一个字符， *inchars* 中的第二个字符会被转换成为 *outchars* 的第二个字符。这个映射过程会一直持续到处理完指定字符。如果 *inchars* 和 *outchars* 的长度不同，则sed编辑器会产生一条错误消息。

来个例子：

```
lxc@Lxc:~/scripts/ch19$ cat data9.txt
This is line 1.
This is line 2.
This is line 3.
This is line 4.
This is line 5.
This is line 1 again.
This is line 3 again.
This is the last file line.

lxc@Lxc:~/scripts/ch19$ sed 'y/123/789/' data9.txt
This is line 7.
This is line 8.
This is line 9.
This is line 4.
This is line 5.
This is line 7 again.
This is line 9 again.
This is the last file line.
# 如你所见，inchars中的各个字符都会被替换成outchars中相同位置的字符。
```

转换命令是一个全局命令，也就是说，它会对文本行中匹配到的所有指定字符进行转换，不考虑字符出现的位置：

```
lxc@Lxc:~/scripts/ch19$ echo "Test #1 of try #1." | sed 'y/123/678/'
Test #6 of try #6.
```

7. 再探打印

之前介绍过如何使用p标志和替换命令显示sed编辑器修改过的行。另外，还有3个命令也能打印数据流中的信息。

- 打印（p）命令用于打印文本行。
- 等号（=）命令用于打印行号。
- 列出（l）命令用于列出行。

接下来介绍这三个sed编辑器的打印命令。

1. 打印行

和替换命令中的 `p` 标志类似，打印命令用于打印sed编辑器输出中的一行。如果只用这个命令，倒也没什么特别的：

```
lxc@Lxc:~/scripts/ch19$ echo "This is a test" | sed 'p'
This is a test
This is a test
```

它所做的就是打印已有的数据文本。打印命令最常见的用法是打印包含匹配文本模式的行：

```
lxc@Lxc:~/scripts/ch19$ cat data6.txt
This is line number 1.
This is line number 2.
This is the 3rd line.
This is the 4th line.

lxc@Lxc:~/scripts/ch19$ sed -n '/3rd line/p' data6.txt
This is the 3rd line.
# 在命令行中使用-n选项可以抑制其他行的输出，只打印包含匹配文本模式的行。
```

也可以用它来快速打印数据流中的部分行：

```
lxc@Lxc:~/scripts/ch19$ sed -n '2,3p' data6.txt
This is line number 2.
This is the 3rd line.
```

如果需要在使用替换或修改命令做出改动之前查看相应的行，可以使用打印命令。

```
lxc@Lxc:~/scripts/ch19$ sed -n '/3/{
> p
> s/line/test/p
> }' data6.txt
This is the 3rd line.
This is the 3rd test.
```

这个例子中，sed编辑器会首先查找包含数字3的行，然后执行两条命令。第一条命令打印出原始的匹配行。第二条命令用替换命令替换文本并通过p标志打印出替换结果。输出同时显示了原始的文本行和新的文本行。

2. 打印行号

等号命令会打印文本行在数据流中的行号。行号由数据流中的换行符决定。数据流中每出现一个换行符，sed编辑器就会认为有一行文本结束了。

```
lxc@Lxc:~/scripts/ch19$ cat data1.txt
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.

lxc@Lxc:~/scripts/ch19$ sed '=' data1.txt
1
The quick brown fox jumps over the lazy dog.
2
The quick brown fox jumps over the lazy dog.
3
The quick brown fox jumps over the lazy dog.
4
The quick brown fox jumps over the lazy dog.
```

sed编辑器在实际文本行之前会先打印行号。如果要在数据流中查找特定文本，那么等号命令用起来特别方便：

```
lxc@Lxc:~/scripts/ch19$ cat data7.txt
This is line number 1.
This is line number 2.
This is the 3rd line.
This is the 4th line.
This is line number 1 again; we want to keep it.
This is more text we want to keep.
Last line in the file; we want to keep it.

lxc@Lxc:~/scripts/ch19$ sed -n '/text/{
> =
> p
> }' data7.txt
6
This is more text we want to keep.
```

利用 `-n` 选项，就能让sed编辑器只显示包含匹配文本模式的文本行的行号和内容。

3. 列出行

列出命令可以打印数据流中的文本和不可打印字符。在显示不可打印字符的时候，要么在其八进制前加一个反斜线，要么使用标准的C语言命名规范（用于常见的不可打印字符），比如\t用于代表制表符：

```
lxc@Lxc:~/scripts/ch19$ cat data10.txt
This    line    contains        tabs.
This line does contain tabs.

lxc@Lxc:~/scripts/ch19$ sed -n 'l' data10.txt
This\tline\tcontains\ttabs.$
This line does contain tabs.$
```

制表符所在的位置显示为\t。行尾的美元符号表示换行符。如果数据流包含转义字符，则列出命令会在必要时用八进制值显示：

```
lxc@Lxc:~/scripts/ch19$ cat data11.txt
This line contains an escape character.

lxc@Lxc:~/scripts/ch19$ sed -n 'l' data11.txt
This line contains an escape character. \a$
```

data11.txt文本文件含有一个用于产生铃声的转移控制码。当用 `cat` 命令显示文本文件时，转义控制码不会显示出来，你只能听到声音（如果打开了音箱的话）。但利用列出命令，就能显示出所使用的转义控制码。

8. 使用sed处理文件

替换命令包含一些文件处理标志。一些常规的sed编辑器命令也可以让你无须替换文本即可完成此操作。

1. 写入文件

写入（w）命令用向文件写入行。

命令格式：

```
[address]w filename
```

filename 可以使用相对路径或者绝对路径，但不管使用哪种，运行sed编辑器的用户都必须有文件的写入权限。*address* 可以是sed支持的任意类型的寻址方式，比如单个行号、文本模式、行区间或文本模式区间。

下面的例子会将数据流中的前两行写入文本文件：

```
lxc@Lxc:~/scripts/ch19$ sed -n '1,2w test.txt' data6.txt
lxc@Lxc:~/scripts/ch19$ cat test.txt
This is line number 1.
This is line number 2.
```

如果要根据一些共用的文本值，从主文件中创建一份数据文件，则使用写入命令会非常方便：

```
lxc@Lxc:~/scripts/ch19$ cat data12.txt
Blum, R       Browncoat
McGuiness, A  Alliance
Bresnahan, C  Browncoat
Harken, C     Alliance

lxc@Lxc:~/scripts/ch19$ sed -n '/Browncoat/w Browncoat.txt' data12.txt
lxc@Lxc:~/scripts/ch19$ cat Browncoat.txt
Blum, R       Browncoat
Bresnahan, C  Browncoat
```

如你所见，sed编辑器会将匹配文本模式的数据行写入文件。

2. 从文件读取数据

你已经知道如何通过sed命令行向数据流插入文本或附件文本。读取（r）命令允许将一条独立文件中的数据插入数据流。

命令格式如下：

```
[address]r filename
```

filename 参数指定了数据文件的绝对路径或相对路径。读取命令无法使用地址区间，只能指定单个文本号或文本模式地址。sed编辑器会将文件内容插入到指定地址之后：

```
lxc@Lxc:~/scripts/ch19$ cat data13.txt
This is an added line.
This is a second added line.

lxc@Lxc:~/scripts/ch19$ sed '3r data13.txt' data6.txt
This is line number 1.
This is line number 2.
This is the 3rd line.
This is an added line.
This is a second added line.
This is the 4th line.
# 该命令是读取 data13.txt 的数据插入到 data6.txt 第3行之后。
```

sed编辑器会将数据文件中的所有文本行都插入数据流。在使用文本模式地址时，同样的方法也适用：

```
lxc@Lxc:~/scripts/ch19$ sed '/number 2/r data13.txt' data6.txt
This is line number 1.
This is line number 2.
This is an added line.
This is a second added line.
This is the 3rd line.
This is the 4th line.
# 该命令是在 data6.txt 文件中查找匹配 number 2 模式的文本行，并在该行之后插入
data13.txt 文件的全部数据。

lxc@Lxc:~/scripts/ch19$ cat data6.txt
This is line number 1.
This is line number 2.
This is the 3rd line.
```

```
This is line number 2.
This is the 4th line.

lxc@Lxc:~/scripts/ch19$ sed '/number 2/r data13.txt' data6.txt
This is line number 1.
This is line number 2.
This is an added line.
This is a second added line.
This is the 3rd line.
This is line number 2.
This is an added line.
This is a second added line.
This is the 4th line.
```

要在数据流末尾添加文本，只需使用美元符号地址即可：

```
lxc@Lxc:~/scripts/ch19$ sed '$r data13.txt' data6.txt
This is line number 1.
This is line number 2.
This is the 3rd line.
This is the 4th line.
This is an added line.
This is a second added line.
```

该命令还有一种很酷的用法是和删除命令配合使用，利用另一个文件中的数据来替换文件中的占位文本。
假如你保存在文本文件中的套用信件如下所示：

```
lxc@Lxc:~/scripts/ch19$ cat notice.std
Would the following people:
LIST
please report to the ship's captain.
```

套用信件将通用占位文本 *LIST* 放在了人物名单的位置。要在占位文本后插入名单，只需使用读取命令即可。但这样的话，占位文本仍然会留在输出中。为此，可以用删除命令删除占位文本，其结果如下：

```
lxc@Lxc:~/scripts/ch19$ sed '/LIST/{
> r data12.txt
> d
> }' notice.std
Would the following people:
Blum, R       Browncoat
McGuiness, A  Alliance
Bresnahan, C  Browncoat
Harken, C     Alliance
please report to the ship's captain.
```

如你所见，现在占位文本已经被替换成了数据文件中的名单。

3. 实战演练

在第11章，我们讲过shell脚本文件的第一行：

```
#!/bin/bash
```

第一行有时被称为 `shebang`（`shebang` 这个词其实是两个字符名称(sharp-bang)的简写。在Unix专业术语中，用 sharp 或 hash（有时候是mesh）来称呼字符 "#"，用 bang 来称呼惊叹号 "!"，因而 shebang 合起来就代表了这两个字符 #!），在传统的Unix shell脚本中其形式如下：

```
#!/bin/sh
```

这种传统通常也延续到了Linux的bash shell脚本，这在过去不是问题，大多数发行版将 */bin/sh* 链接到了bash shell(/bin/bash)，因此，如果在脚本中使用 */bin/sh* 就相当于写的是 */bin/bash*：

```
lxc@Lxc:~/scripts/ch19$ ls -l /bin/sh
lrwxrwxrwx 1 root root 13 11月  9 21:35 /bin/sh -> /usr/bin/bash
# 我使用的是Ubuntu，但如你所见，我在11月13号的晚上21:35分修改了该软连接，使其指向了bash
```

在某些Linux发行版(比如Ubuntu)中，*/bin/sh* 文件并没有链接到bash shell。

```
lxc@Lxc:~/scripts/ch19$ ls -l /bin/sh
lrwxrwxrwx 1 root root 13 11月  9 21:35 /bin/sh -> dash
```

在这类系统中运行的shell脚本，如果使用 */bin/sh* 作为 shebang，则脚本会运行在 dash shell 而非 bash shell 中。这可能会造成很多shell脚本命令执行失败。

现在我们搞一个脚本，使以dash shell运行的脚本文件运行在bash shell上。

ChangeScriptShell.sh

```
#!/bin/bash
# Change the shebang used for a directory of scripts
#
################### Function Declarations ############################
#
function errorOrExit {
    echo
    echo $message1
    echo $message2
    echo "Exiting script..."
    exit
}
#
function modifyScripts {
    echo
    read -p "Directory name in which to store new scripts? " newScriptDir
    #
    echo "Modifying the scripts started at $(date +%N) nanoseconds"
    #
    count=0
    for filename in $(grep -l "/bin/sh" $scriptDir/*.sh)
    do
        newFilename=$(basename $filename)
        cat $filename |
        sed '1c\#!/bin/bash' > $newScriptDir/$newFilename
        count=$[$count + 1]
```

```
    done
    echo "$count modifications completed at $(date +%N) nanoseconds"
}
#
################## Check for Script Directory ######################
if [ -z $1 ]
then
    message1="The name of the directory containing scripts to check"
    message2="is missing. Please provide the name as a parameter."
        errorOrExit
else
    scriptDir=$1
fi
#
################ Create Shebang Report ############################
#
sed -sn '1F;
1s!/bin/sh!/bin/bash!' $scriptDir/*.sh |
gawk 'BEGIN {print ""
print "The following scripts have /bin/sh as their shebang:"
print "================================================="}
{print $0}
END {print ""
print "End of Report"}'
#
################## Change Scripts? #################################
#
#
echo
read -p "Do you wish to modify these scripts' shebang? (Y/n)? " answer
#
case $answer in
Y | y)
    modifyScripts
    ;;
N | n)
    message1="No scripts will be modified."
    message2="Run this script later to modify, if desired."
    errorOrExit
    ;;
*)
    message1="Did not answer Y or n."
    message2="No scripts will be modified."
    errorOrExit
    ;;
esac
```

sed命令行中的 `-s` 选项可以告知sed将目录内的各个文件作为单独的流，这样可以检查目录下每个文件的第一行。`-n` 选项则会抑制输出，这样就不会看到脚本的内容了：

```
sed -sn '1s!/bin/sh!/bin/bash!' OldScripts.*.sh
```

sed命令行中用到的另一个命令是 `F`。该命令会告知sed打印出当前正在处理的文件名，且不受 `-n` 选项的影响。因为脚本文件名只需显示一次即可，所以要在 `F` 命令之前加上数字1（否则的话，所处理的每个文件的每一行都会显示文件名）。现在，我们可以知道哪些脚本使用的是旧式的shebang：

```
sed -sn '1F;
> 1s!/bin/sh!/bin/bash!' OldScripts/*.sh
```

ch20 正则表达式

本章将介绍如何在sed和gawk中创建正则表达式，以得到所需要的数据。

1. 正则表达式基础

1. 定义

正则表达式是一种可供Linux工具过滤文本的自定义模板。Linux工具（比如sed和gawk）会在读取数据时使用正则表达式对数据进行模式匹配。如果数据匹配模式，它就会被接受并进行处理。如果数据不匹配模式，它就会被弃用。
正则表达式使用元字符（原书中使用的是 wildcard character(通配符)，准确的说，通配符和正则表达式并不是一回事，虽然正则表达式中也有 `*` 和 `?`，但是作用完全不一样。）来描述数据流中的一个或多个字符。
Linux中很多场景使用特殊字符来描述具体内容不确定的数据。比如在 `ls` 命令中使用通配符列出文件和目录（准确的过程是这样的：shell负责处理通配符，在本例中，将 *.sh扩展为当前目录下以 .sh 结尾的所有文件名，`ls` 命令会列出由通配符匹配的那些文件信息。**通配符是由shell处理的，命令看到的只是经过处理后的匹配结果**）

```
lxc@Lxc:~/scripts/ch20$ ls -al *.sh
-rwxrw-r-- 1 lxc lxc 324 11月 15 20:37 countfiles.sh
-rwxrw-r-- 1 lxc lxc 146 11月 15 21:41 isemail.sh
-rwxrw-r-- 1 lxc lxc 139 11月 15 21:03 isphone.sh
```

正则表达式的工作方式与通配符类似。正则表达式包含文本和/或特殊字符（这些特殊字符在正则表达式中称作 **元字符(metacharacter)**），定义了sed和gawk匹配时使用的模板。你可以在正则表达式中使用不同的特殊字符来第一特定的数据过滤模式。

2. 正则表达式的类型

使用正则表达式最大的问题在于不止一种类型的正则表达式。在Linux中，不同的应用程序可能使用不同类型的正则表达式。
正则表达式是由 **正则表达式引擎** 实现的。这是一种底层软件，负责解释正则表达式并用这些模式进行文本匹配。
尽管在Linux世界有很多不同的正则表达式引擎，但最流行的是以下两种：

- POSIX基础正则表达式(basic regular expression，BRE)引擎。
- POSIX扩展正则表达式(extended regular expression，ERE)引擎。

大多数Linux工具至少符合POSIX BRE引擎规范，能够识别该规范定义的所有模式符号。有些工具（比如sed）仅符合BRE引擎规范的一个子集，这是出于速度方面的考虑导致的，因为sed希望尽可能快的处理数据流中的文本。POSIX ERE引擎多见于依赖正则表达式过滤文本的编程语言中。它为常见模式（比如数字、单词、以及字母数字字符）提供了高级模式符号和特殊符号。gawk使用ERE引擎来处理正则表达式。

2. 定义BRE模式

最基本的BRE模式是匹配数据流中的文本字符。本节将介绍在正则表达式中定义文本的方法及其预期的匹配结果。

1. 普通文本

正则表达式区分大小写，它只会匹配大小写也相符的模式。

```
lxc@Lxc:~/scripts/ch20$ echo "This is a test" | sed -n '/This/p'
This is a test
lxc@Lxc:~/scripts/ch20$ echo "This is a test" | sed -n '/this/p'
lxc@Lxc:~/scripts/ch20$
```

在正则表达式中无须写出整个单词。只要定义的文本出现在数据流中，正则表达式就能够匹配。

```
lxc@Lxc:~/scripts/ch20$ echo "The books are expensive" | sed -n '/book/p'
The books are expensive
```

尽管数据流中文本是 *books*，但数据中含有正则表达式 *book*，因此正则表达式能匹配数据。当然，反过来就不行了。

```
lxc@Lxc:~/scripts/ch20$ echo "The book is expensive" | sed -n '/books/p'
lxc@Lxc:~/scripts/ch20$
```

你也无须在正则表达式中只使用单个文本单词，空格和数字也是可以的。在正则表达式中，空格和其他字符没有什么区别。

```
lxc@Lxc:~/scripts/ch20$ echo "This is line number 1" | sed -n '/ber 1/p'
This is line number 1
```

如果正则表达式中定义了空格，那么它必须出现在数据流中。你甚至可以创建匹配多个连续空格的正则表达式：

```
lxc@Lxc:~/scripts/ch20$ cat data1
This is a normal line of text.
This is  a line with too many spaces.
lxc@Lxc:~/scripts/ch20$ sed -n '/  /p' data1
This is  a line with too many spaces.
```

2. 特殊字符

正则表达式中的一些字符具有特别的含义。正则表达式能识别的特殊字符如下所示：

`.*[]^${}\+?|()`

如果要将某个特殊字符视为普通字符，则必须将其转义，需要在其前面加个反斜线。

来几个例子：

```
lxc@Lxc:~/scripts/ch20$ cat data2
The cost is $4.00
lxc@Lxc:~/scripts/ch20$ sed -n '/\$/p' data2
The cost is $4.00
```

```
lxc@Lxc:~/scripts/ch20$ echo "\ is a special character" | sed -n '/\\/p'
\ is a special character
```

尽管正斜线不是正则表达式中的特殊字符，但在sed或gawk正则表达式中要用到它一样也要转义。

```
lxc@Lxc:~/scripts/ch20$ echo "3 / 2" | sed -n '/\//p'
3 / 2
```

3. 锚点字符

在默认情况下，当指定一个正则表达式模式时，只要模式出现在数据流中的任何地方，它就能匹配。有两个特殊字符可以用来将模式锁定在数据流中的行首或行尾。

1. 锚定行首

脱字符(^)可以指定位于数据流中文本行行首的模式。如果模式出现在行首之外的位置，则正则表达式无法匹配。要使用脱字符，就必须将其置于正则表达式之前：

```
lxc@Lxc:~/scripts/ch20$ echo "The book store" | sed -n '/^book/p'
lxc@Lxc:~/scripts/ch20$ echo "Books are great" | sed -n '/^Book/p'
Books are great
```

脱字符使得正则表达式引擎在每行（由换行符界定）的行首检查模式：

```
lxc@Lxc:~/scripts/ch20$ cat data3
This is a test line.
this is another test line.
A line that tests this feature.
Yet more testing of this
lxc@Lxc:~/scripts/ch20$ sed -n '/^this/p' data3
this is another test line.
```

如果将脱字符放在正则表达式开头之外的位置，那么它就跟普通字符一样，没什么特殊含义了。

```
lxc@Lxc:~/scripts/ch20$ echo "This ^ is a test" | sed -n '/s ^/p'
This ^ is a test
```

如果正则表达式中只有脱字符，就不必用反斜线转义。但如果在正则表达式中先指定脱字符，随后还有其他文本，那就必须在脱字符前用转义字符：

```
lxc@Lxc:~/scripts/ch20$ echo "This ^ is a test" | sed -n '/s ^/p'
This ^ is a test
lxc@Lxc:~/scripts/ch20$ echo "This ^ is a test" | sed -n '/^/p'
This ^ is a test
# 注意看这个示例，如果正则表达式中只有脱字符就不必用反斜线转义。

lxc@Lxc:~/scripts/ch20$ echo "I love ^regex" | sed -n '/\^regex/p'
I love ^regex
# 但如果在正则表达式中先指定脱字符，随后还有其他文本，那就必须在脱字符前用转义字符。这也相当容易理解
# 因为你不转义，那不就又成了锚定行首了嘛，就变成了查找以...开头的模式。

lxc@Lxc:~/scripts/ch20$ echo "I love ^regex" | sed -n '/ ^/p'
I love ^regex
```

2. 锚定行尾

特殊字符美元符号($)定义了行尾锚点。将这个特殊字符放在正则表达式之后则表示数据行必须以该模式结尾。

```
lxc@Lxc:~/scripts/ch20$ echo "This is a good book" | sed -n '/book$/p'
This is a good book
lxc@Lxc:~/scripts/ch20$ echo "This book is good" | sed -n '/book$/p'
```

使用该模式时，你必须清楚到底要查什么：

```
lxc@Lxc:~/scripts/ch20$ echo "There are a lot of books" | sed -n '/$book/p'
lxc@Lxc:~/scripts/ch20$
```

尽管 *book* 这个模式确实存在于数据文本中，但该数据文本并不是以 *book* 的模式结尾的，所以不匹配。

3. 组合锚点

我们可以在同一行中组合使用行首锚点和行尾锚点。比如，我们要查找只含有特定文本模式的数据行：

```
lxc@Lxc:~/scripts/ch20$ cat data4
this is a test of using both anchors
I said this is a test
this is a test
I'm sure this is a test.
lxc@Lxc:~/scripts/ch20$ sed -n '/^this is a test$/p' data4
this is a test
```

我们可以直接将这两个锚点组合在一起，之间不加任何文本。这样可以过出数据流中的空行。

```
lxc@Lxc:~/scripts/ch20$ cat data5
This is one test line.


This is another test line.
lxc@Lxc:~/scripts/ch20$ sed '/^$/d' data5
This is one test line.
This is another test line.
# 我们先过滤出了空行，然后删除了空行。所以输出中没有空行。
```

4. *点号字符*

点号字符可以匹配除换行符之外的任意单个字符。点号必须匹配一个字符，如果点号字符的位置没有可匹配的字符，那么模式不成立。

```
lxc@Lxc:~/scripts/ch20$ cat data6
This is a test of a line.
The cat is sleeping.
That is a very nice hat.
This test is at line four.
at ten o'clock we'll go home.
lxc@Lxc:~/scripts/ch20$ sed -n '/.at/p' data6
The cat is sleeping.
That is a very nice hat.
This test is at line four.
# 最后一处匹配是空格匹配了点号字符。
```

5. *字符组*

点号字符在匹配某个位置上的任意字符时很有用，但如果你想限定要匹配的具体字符，可以使用 **字符组 (character class)**。
你可以在正则表达式中定义用来匹配某个位置的一组字符。如果字符组中的某个字符出现在了数据流中，那就能匹配该模式。方括号用于定义字符组。在方括号中加入你希望出现在该字符组中的所有字符，就可以在正则表达式中像其他特殊字符一样使用字符组了。

来个例子：

```
lxc@Lxc:~/scripts/ch20$ sed -n '/[ch]at/p' data6
The cat is sleeping.
That is a very nice hat.
```

在这个例子中匹配这个模式的只有单词 *cat* 和 *hat* 。

在不确定某个字符的大小写时非常适合使用字符组，在单个正则表达式可以使用多个字符组：

```
lxc@Lxc:~/scripts/ch20$ echo "Yes" | sed -n '/[Yy]es/p'
Yes
lxc@Lxc:~/scripts/ch20$ echo "YEs" | sed -n '/[Yy][Ee][Ss]/p'
YEs
```

字符组中当然并非只能使用字母，也可以在其中使用数字：

```
lxc@Lxc:~/scripts/ch20$ cat data7
This line doesn't contain a number.
This line has 1 number on it.
This line a number 2 on it.
This line has a number 4 on it.
lxc@Lxc:~/scripts/ch20$ sed -n '/[0123]/p' data7
This line has 1 number on it.
This line a number 2 on it.
```

你也可以将多个字符组组合在一起，以检查数字是否具备正确的格式，比如电话号码和邮政编码。当你尝试匹配某种特定格式时，一定要注意。这里有个邮政编码匹配出错的例子：

```
lxc@Lxc:~/scripts/ch20$ cat data8
60633
46201
223001
4353
22203
lxc@Lxc:~/scripts/ch20$ sed -n '/[1234567890][1234567890][1234567890]
[1234567890][1234567890]/p' data8
60633
46201
223001
22203
```

在结果中，错误的保留了一个6位数的结果，尽管我们只定义了5个字符组。记住，正则表达式可以匹配数据流中任意位置的文本。匹配模式之外经常会有其他字符。如果想确保只匹配5位数，可以像下面那样，指明匹配数字的起止位置。

```
lxc@Lxc:~/scripts/ch20$ sed -n '/^[1234567890][1234567890][1234567890]
[1234567890][1234567890]$/p' data8
60633
46201
22203
```

字符组一种常见的用法是解析拼错的单词，比如用户表单输入的数据。

```
lxc@Lxc:~/scripts/ch20$ cat data9
I need to have some maintenence done on my car.
I'll pay that in a seperate invoice.
After I pay for the maintenance my car will be as good as new.
lxc@Lxc:~/scripts/ch20$ sed -n '
> /maint[ea]n[ae]nce/p
> /sep[ea]r[ea]te/p
> ' data9
I need to have some maintenence done on my car.
I'll pay that in a seperate invoice.
After I pay for the maintenance my car will be as good as new.
```

两个sed打印命令利用正则表达式字符组来查找文本中拼错的单词 *maintenence* 和 *seperate*。同样的正则表达式也能匹配正确拼写的结果。

6. 排除型字符组

在正则表达式中，你可以反转字符组的作用：匹配字符组中没有的字符。为此，只需在字符组的开头添加脱字符即可：

```
lxc@Lxc:~/scripts/ch20$ sed -n '/[^ch]at/p' data6
This test is at line four.
```

通过排除型字符组，正则表达式会匹配除 *c* 或 *h* 之外的任何字符以及文本模式。由于空格字符属于这个范围，因此通过了模式匹配。但即使是排除型，字符组必须匹配一个字符，以 *at* 为起始的行不能匹配模式。

7. 区间

可以用单连字符在字符组中表示字符区间。只需指定区间的第一个字符、连字符以及区间的最后一个字符即可。根据Linux系统使用的字符集（参见第二章），字符组会包括此区间（闭区间）内的任意字符。

来几个例子：

```
lxc@Lxc:~/scripts/ch20$ sed -n '/^[0-9][0-9][0-9][0-9][0-9]$/p' data8
60633
46201
22203
```

同样的方法也适用于字母。

```
lxc@Lxc:~/scripts/ch20$ sed -n '/[c-h]at/p' data6
The cat is sleeping.
That is a very nice hat.
# 该模式只会匹配在字母 c 和 h 之间的单词。
# 在这种情况下，只含有单词 at 的行无法匹配该模式。
```

还可以在单个字符组内指定多个不连续的区间：

```
lxc@Lxc:~/scripts/ch20$ sed -n '/[a-ch-m]at/p' data6
The cat is sleeping.
That is a very nice hat.
# 字符组内指定了两个区间，a-c 以及 h-m。
```

8. 特殊的字符组

除了定义自己的字符组，BRE还提供了一些特殊的字符组，以用来匹配特定类型的字符。下表列出了可用的BRE特殊字符组。

字符组	描述
[[:alpha:]]	匹配任意字母字符，无论是大写还是小写
[[:alnum:]]	匹配任意字母数字字符，0-9、A-Z、a-z
[[:blank:]]	匹配空格或制表符
[[:digit:]]	匹配0-9中的数字

字符组	描述
[[:lower:]]	匹配小写字母字符a-z
[[:print:]]	匹配任意可打印字符
[[:punct:]]	匹配标点符号
[[:space:]]	匹配任意空白字符:空格、制表符、换行符、分页符(formfeed)、垂直制表符和回车符
[[:upper:]]	匹配任意大写字母字符A-Z

特殊字符组在正则表达式中的用法和普通字符组一样:

```
lxc@Lxc:~/scripts/ch20$ echo "abc" | sed -n '/[[:digit:]]/p'
lxc@Lxc:~/scripts/ch20$ echo "abc" | sed -n '/[[:alpha:]]/p'
abc
lxc@Lxc:~/scripts/ch20$ echo "abc123" | sed -n '/[[:digit:]]/p'
abc123
lxc@Lxc:~/scripts/ch20$ echo "This, is, a test" | sed -n '/[[:punct:]]/p'
This, is, a test
```

9. 星号

在字符后面放置星号表明该字符必须在匹配模式的文本中出现0次或多次

```
lxc@Lxc:~/scripts/ch20$ echo "ik" | sed -n '/ie*k/p'
ik
lxc@Lxc:~/scripts/ch20$ echo "iek" | sed -n '/ie*k/p'
iek
lxc@Lxc:~/scripts/ch20$ echo "ieek" | sed -n '/ie*k/p'
ieek
lxc@Lxc:~/scripts/ch20$ echo "ieeeeeeeek" | sed -n '/ie*k/p'
ieeeeeeeek
```

这个特殊符号广泛用于处理有常见拼写错误或在不同语言中有拼写变化的单词。比如:

```
lxc@Lxc:~/scripts/ch20$ echo "I am getting a color TV" | sed -n '/colou*r/p'
I am getting a color TV
lxc@Lxc:~/scripts/ch20$ echo "I am getting a colour TV" | sed -n '/colou*r/p'
I am getting a colour TV
```

如果一个单词经常被拼错也可以用星号来容忍这种错误。

```
lxc@Lxc:~/scripts/ch20$ echo "I ate a potatoe with my lunch" | sed -n
'/potatoe*/p'
I ate a potatoe with my lunch
lxc@Lxc:~/scripts/ch20$ echo "I ate a potato with my lunch" | sed -n
'/potatoe*/p'
I ate a potato with my lunch
```

可以将点号字符和星号字符组合起来。**这个组合能够匹配任意数量的任意字符**，通常用在数据流中两个可能相邻或不相邻的字符串之间：

```
lxc@Lxc:~/scripts/ch20$ echo "This is a regular pattern expression" | sed -n '/regular.*expression/p'
This is a regular pattern expression
```

星号还能用于字符组，指定可能在文本中出现0次或多次的字符组或字符区间：

```
lxc@Lxc:~/scripts/ch20$ echo "bt" | sed -n '/b[ae]*t/p'
bt
lxc@Lxc:~/scripts/ch20$ echo "bat" | sed -n '/b[ae]*t/p'
bat
lxc@Lxc:~/scripts/ch20$ echo "bet" | sed -n '/b[ae]*t/p'
bet
lxc@Lxc:~/scripts/ch20$ echo "btt" | sed -n '/b[ae]*t/p'
btt
lxc@Lxc:~/scripts/ch20$ echo "baat" | sed -n '/b[ae]*t/p'
baat
lxc@Lxc:~/scripts/ch20$ echo "beet" | sed -n '/b[ae]*t/p'
beet
lxc@Lxc:~/scripts/ch20$ echo "baet" | sed -n '/b[ae]*t/p'
baet
lxc@Lxc:~/scripts/ch20$ echo "baaeaeeaeaaet" | sed -n '/b[ae]*t/p'
baaeaeeaeaaet
```

3. 扩展正则表达式

POSIX ERE模式提供了一些可供Linux应用程序和工具使用的额外符号。gawk支持ERE模式，但sed不支持。

> 记住，sed和gawk的正则表达式引擎之间是有区别的。gawk可以使用大多数扩展的正则表达式符号，并且提供了一些sed所不具备的额外过滤功能。但正因如此，gawk在处理数据时往往比较慢。

本节将介绍可用于gawk脚本中的常见ERE模式符号。

1. 问号

问号和星号类似，但有一些不同，**问号表明前面的字符可以出现0次或1次**，仅此而已。

```
lxc@Lxc:~/scripts/ch20$ echo "bt" | gawk '/be?t/{print $0}'
bt
lxc@Lxc:~/scripts/ch20$ echo "bet" | gawk '/be?t/{print $0}'
bet
lxc@Lxc:~/scripts/ch20$ echo "beet" | gawk '/be?t/{print $0}'
lxc@Lxc:~/scripts/ch20$ echo "beeet" | gawk '/be?t/{print $0}'
```

跟星号一样，可以将问号和字符组一起使用。

```
lxc@Lxc:~/scripts/ch20$ echo "bt" | gawk '/b[ae]?t/{print $0}'
bt
lxc@Lxc:~/scripts/ch20$ echo "bat" | gawk '/b[ae]?t/{print $0}'
bat
lxc@Lxc:~/scripts/ch20$ echo "bet" | gawk '/b[ae]?t/{print $0}'
bet
lxc@Lxc:~/scripts/ch20$ echo "baet" | gawk '/b[ae]?t/{print $0}'
lxc@Lxc:~/scripts/ch20$ echo "beat" | gawk '/b[ae]?t/{print $0}'
lxc@Lxc:~/scripts/ch20$ echo "baat" | gawk '/b[ae]?t/{print $0}'
```

2. 加号

也类似于星号，**加号表明前面的字符可以出现1次或多次，但必须至少出现一次。**

```
lxc@Lxc:~/scripts/ch20$ echo "beeet" | gawk '/b[ae]+t/{print $0}'
beeet
lxc@Lxc:~/scripts/ch20$ echo "beet" | gawk '/b[ae]+t/{print $0}'
beet
lxc@Lxc:~/scripts/ch20$ echo "bet" | gawk '/b[ae]+t/{print $0}'
bet
lxc@Lxc:~/scripts/ch20$ echo "bt" | gawk '/b[ae]+t/{print $0}'
```

与星号和问号一样，也可用于字符组：

```
lxc@Lxc:~/scripts/ch20$ echo "bt" | gawk '/b[ae]+t/{print $0}'
lxc@Lxc:~/scripts/ch20$ echo "bat" | gawk '/b[ae]+t/{print $0}'
bat
lxc@Lxc:~/scripts/ch20$ echo "bet" | gawk '/b[ae]+t/{print $0}'
bet
lxc@Lxc:~/scripts/ch20$ echo "baet" | gawk '/b[ae]+t/{print $0}'
baet
lxc@Lxc:~/scripts/ch20$ echo "beet" | gawk '/b[ae]+t/{print $0}'
beet
lxc@Lxc:~/scripts/ch20$ echo "beaaeaeaeaet" | gawk '/b[ae]+t/{print $0}'
beaaeaeaeaet
```

3. 花括号

ERE中的花括号允许为正则表达式指定具体的可重复次数，这通常称为 **区间**。可以用两种格式来指定区间。

- m: 正则表达式正好出现m次
- m,n: 正则表达式至少出现m次，至多出现n次。

这个特性可以精确指定字符或字符组在模式中具体出现的次数。

> **注意：**，在默认情况下，gawk不识别正则表达式区间，必须指定gawk的命令行选项 `--re-interval` 才行。

来几个例子：

```
lxc@Lxc:~/scripts/ch20$ echo "bt" | gawk --re-interval '/be{1}t/{print $0}'
lxc@Lxc:~/scripts/ch20$ echo "bet" | gawk --re-interval '/be{1}t/{print $0}'
bet
lxc@Lxc:~/scripts/ch20$ echo "beet" | gawk --re-interval '/be{1}t/{print $0}'
```

指定区间下限和上限:

```
lxc@Lxc:~/scripts/ch20$ echo "bt" | gawk --re-interval '/be{1,2}t/{print $0}'
lxc@Lxc:~/scripts/ch20$ echo "bet" | gawk --re-interval '/be{1,2}t/{print $0}'
bet
lxc@Lxc:~/scripts/ch20$ echo "beet" | gawk --re-interval '/be{1,2}t/{print
$0}'
beet
lxc@Lxc:~/scripts/ch20$ echo "beeet" | gawk --re-interval '/be{1,2}t/{print
$0}'
```

区间也同样适用于字符组:

```
lxc@Lxc:~/scripts/ch20$ echo "bt" | gawk --re-interval '/b[ae]{1,2}t/{print
$0}'
lxc@Lxc:~/scripts/ch20$ echo "bat" | gawk --re-interval '/b[ae]{1,2}t/{print
$0}'
bat
lxc@Lxc:~/scripts/ch20$ echo "bet" | gawk --re-interval '/b[ae]{1,2}t/{print
$0}'
bet
lxc@Lxc:~/scripts/ch20$ echo "baet" | gawk --re-interval '/b[ae]{1,2}t/{print
$0}'
baet
lxc@Lxc:~/scripts/ch20$ echo "baat" | gawk --re-interval '/b[ae]{1,2}t/{print
$0}'
baat
lxc@Lxc:~/scripts/ch20$ echo "beet" | gawk --re-interval '/b[ae]{1,2}t/{print
$0}'
beet
lxc@Lxc:~/scripts/ch20$ echo "beeet" | gawk --re-interval '/b[ae]
{1,2}t/{print $0}'
lxc@Lxc:~/scripts/ch20$ echo "baaeeet" | gawk --re-interval '/b[ae]
{1,2}t/{print $0}'
```

4. 竖线符号

竖线符号允许在检查数据流时，以逻辑OR方式指定正则表达式引擎要使用的两个或多个模式。如果其中任何一个模式匹配了数据流文本，就视为匹配。如果没有模式匹配，则匹配失败。

竖线符号使用格式如下:

```
expr1|expr2|.....
```

来个例子:

```
lxc@Lxc:~/scripts/ch20$ echo "The cat is asleep" | gawk '/cat|dog/{print $0}'
The cat is asleep
lxc@Lxc:~/scripts/ch20$ echo "The dog is asleep" | gawk '/cat|dog/{print $0}'
The dog is asleep
lxc@Lxc:~/scripts/ch20$ echo "The sheep is asleep" | gawk '/cat|dog/{print
$0}'
```

> 注意，正则表达式和竖线符号之间不能有空格，否则这个空格会被认为是正则表达式模式的一部分。

竖线符号两侧的子表达式可以采用正则表达式可用的任何模式符号。

```
lxc@Lxc:~/scripts/ch20$ echo "He has a cat" | gawk '/[ch]at|dog/{print $0}'
He has a cat
```

5. 表达式分组

也可以用圆括号对正则表达式进行分组。分组之后，每一组会被视为一个整体，可以像对普通字符一样，对该组应用特殊字符。

```
lxc@Lxc:~/scripts/ch20$ echo "Sat" | gawk '/Sat(urday)?/{print $0}'
Sat
lxc@Lxc:~/scripts/ch20$ echo "Saturday" | gawk '/Sat(urday)?/{print $0}'
Saturday
```

结尾的 *urday* 分组和问号使得该模式能够匹配 *Saturday* 的全写或缩写 *Sat*。

将分组和竖线符号结合起来创建可选的模式匹配组是很常见的做法。

```
lxc@Lxc:~/scripts/ch20$ echo "cab" | gawk '/(c|b)a(b|t)/{print $0}'
cab
lxc@Lxc:~/scripts/ch20$ echo "cat" | gawk '/(c|b)a(b|t)/{print $0}'
cat
lxc@Lxc:~/scripts/ch20$ echo "bab" | gawk '/(c|b)a(b|t)/{print $0}'
bab
lxc@Lxc:~/scripts/ch20$ echo "bat" | gawk '/(c|b)a(b|t)/{print $0}'
bat
lxc@Lxc:~/scripts/ch20$ echo "eat" | gawk '/(c|b)a(b|t)/{print $0}'
```

4. 实战演练

下面演示shell脚本中一些常见的正则表达式实例。

1. 目录文件计数

countfiles.sh

该脚本可以对PATH环境变量中的各个目录所包含文件数量进行统计。

```
#!/bin/bash
# count number of files in your PATH
mypath=`echo $PATH | sed 's/:/ /g'`
total=0
```

```
count=0
for directory in $mypath
do
    check=$(ls $directory)
    for item in $check
    do
        count=$[ $count + 1 ]
        total=$[ $total + 1 ]
    done
    echo "$directory - $count"
    count=0
done

echo "Total: $total"
```

2. 验证电话号码

isphone.sh

验证是否是一个合法的美国电话号码。

```
#!/bin/bash
# script to filter out bad phone numbers
gawk --re-interval '/^\(?[2-9][0-9]{2}\)?(| |-|.)[0-9]{3}( |-|\.)[0-9]
{4}/{print $0}'
```

3. 解析email地址

isemail.sh

```
#!/bin/bash
# script to filter out bad email address
gawk --re-interval '/^([a-zA-Z0-9_\-\.\+]+)@([a-zA-Z0-9_\-\.]+)\.([a-zA-Z]
{2,5})$/{print $0}'
```

ch21 sed进阶

第19章展示了如何用sed编辑器的基本功能来处理数据流中的文本。sed编辑器的基础命令能满足大多数日常文本编辑的需要。本章将介绍sed编辑器所提供的更多高级特性。

1. 多行命令

在之前使用sed编辑器的基础命令时，你可能注意到了一个局限：所有的命令都是针对单行数据执行操作。在sed编辑器读取数据流时，它会根据换行符的位置将数据分成行。sed编辑器会根据定义好的脚本命令，一次一行处理数据，然后移到下一行重复这个过程。
但有时候，你需要对跨多行的数据执行特定的操作。幸运的是，sed编辑器有对应的解决方案。sed编辑器提供了3个可用于处理多行文本的特殊命令。

- N：加入数据流中的下一行，创建一个多行组进行处理。
- D：删除多行组中的一行。
- P：打印多行组中的一行。

1. `next` 命令

在讲解多行 `next(N)` 命令之前，首先需要知道单行版本的 `next` 命令是如何工作的，这样一来，理解多行版本的 `next` 命令的用法就容易多了。

1. 单行 `next` 命令

单行 `next(n)` 命令会告诉sed编辑器移动到数据流中的下一行，不用再返回到命令列表的最开始位置。记住，通常sed编辑器在移动到数据流中的下一行之前，会在当前行执行完所有定义好的命令，而单行的 `next` 命令改变了这个流程。

来个例子：

比如我们想删除 *data1.txt* 文件首行之后的空行，其他空行保留。

```
lxc@Lxc:~/scripts/ch21$ cat data1.txt
Header Line

Data Line #1

End of Data Lines
lxc@Lxc:~/scripts/ch21$ sed '/^$/d' data1.txt
Header Line
Data Line #1
End of Data Lines
# 执行普通的空行匹配会删除所有空行。
lxc@Lxc:~/scripts/ch21$ sed '/Header/{n; d}' data1.txt
Header Line
Data Line #1

End of Data Lines
```

在这个例子中，先用脚本查找到含有单词 *Header* 的那一行，找到之后，单行 `next` 命令会让sed编辑器移动到文本的下一行，也就是我们想要删除的行，然后继续执行命令列表，使用删除命令删除空行。sed编辑器在执行完命令之后会读取数据流中的下一行文本，从头开始执行脚本。

2. 合并文本行

现在来看看多行版的 `next` 命令。 单行的 `next` 命令会将数据流中的下一行移入sed编辑器的工作空间（称为 **模式空间**）。多行版本的 `next` 命令则是将下一行添加到模式空间已有文本之后。这样的结果就是将数据流中的两行文本合并到同一个模式空间中。在文本行之间仍然用换行符分隔，但sed编辑器现在会将两行文本**当成一行**来处理。

```
lxc@Lxc:~/scripts/ch21$ cat data2.txt
Header Line
First Data Line
Second Data Line
End of Data Lines
lxc@Lxc:~/scripts/ch21$ sed '/First/{N;s/\n/ /}' data2.txt
Header Line
First Data Line Second Data Line
End of Data Lines
```

sed编辑器首先查找到含有单词 *First* 的那行文本，找到该行后，使用 `N` 命令将下一行与该行合并，然后用替换命令将换行符替换成了空格。这样，两行文本在sed编辑器的输出中就成了一行。

如果要在数据文件中查找一个可能会分散在两行中的文本短语，那么这会是一个很管用的方法：

```
lxc@Lxc:~/scripts/ch21$ cat data3.txt
On Tuesday, the Linux System
Admin group meeting will be held.
All System Admins should attend.
Thank you for your cooperation.
lxc@Lxc:~/scripts/ch21$ sed 's/System.Admin/Devops Engineer/' data3.txt
On Tuesday, the Linux System
Admin group meeting will be held.
All Devops Engineers should attend.
Thank you for your cooperation.
lxc@Lxc:~/scripts/ch21$ sed 'N;s/System.Admin/Devops Engineer/' data3.txt
On Tuesday, the Linux Devops Engineer group meeting will be held.
All Devops Engineers should attend.
Thank you for your cooperation.
```

注意，替换命令在 *System* 和 *Admin* 之间用点号模式(.)来匹配空格和换行符这两种符号(前文有讲，说点号模式不匹配换行符，这里又说点号模式匹配了换行符，呃，这是sed编辑器将两行合并为一行的缘故吧应该是)。这导致了两行被合并为一行。

要解决这个问题，可以在sed编辑器使用两个替换命令，一个用来处理短语出现在多行的情况，一个用来处理短语出现在单行的情况。

```
lxc@Lxc:~/scripts/ch21$ sed 'N;
> s/System\nAdmin/DevOps\nEngineer/
> s/System Admin/DevOps Engineer/
> ' data3.txt
On Tuesday, the Linux DevOps
Engineer group meeting will be held.
All DevOps Engineers should attend.
Thank you for your cooperation.
```

但这里还有一个不易察觉的问题，该脚本总是在执行sed编辑器命令前将下一行文本读入模式空间，当抵达最后一行文本时，就没有下一行可读了，这时 `N` 命令会叫停sed编辑器。如果恰好要匹配的文本在最后一行，那么命令就无法找到要匹配的数据：

```
lxc@Lxc:~/scripts/ch21$ cat data4.txt
On Tuesday, the Linux System
Admin group meeting will be held.
All System Admins should attend.
lxc@Lxc:~/scripts/ch21$ sed 'N
> s/System\nAdmin/DevOps\nEngineer/
> s/System Admin/DevOps Engineer/
> ' data4.txt
On Tuesday, the Linux DevOps
Engineer group meeting will be held.
All System Admins should attend.
```

Sytem Admin 文本出现在了数据流中的最后一行，但 `N` 命令会错过它，因为没有其他行可以读入模式空间跟这行合并。这个问题不难解决，将单行编辑器命令放到 `N` 命令前面，将多行编辑器命令放到 `N` 命令后面就可以了。

```
lxc@Lxc:~/scripts/ch21$ sed '
s/System Admin/DevOps Engineer/
N
s/System\nAdmin/DevOps\nEngineer/
' data4.txt
On Tuesday, the Linux DevOps
Engineer group meeting will be held.
All DevOps Engineers should attend.
```

2. 多行删除命令

第19行介绍过单行删除(d)命令。sed编辑器用该命令来删除模式空间中的当前行。然而，如果和 `N` 命令一起使用，则必须小心单行删除命令：

```
lxc@Lxc:~/scripts/ch21$ cat data4.txt
On Tuesday, the Linux System
Admin group meeting will be held.
All System Admins should attend.
lxc@Lxc:~/scripts/ch21$ sed 'N; /System\nAdmin/d' data4.txt
All System Admins should attend.
```

单行删除命令会在不同行中查找单词 *System* 和 *Admin*，然后在模式空间中将两行都删除，这未必是你想要的结果。

sed编辑器提供了多行删除（D）命令，该命令只会删除模式空间中的第一行，即删除该行中的换行符及其之前的内容。

```
lxc@Lxc:~/scripts/ch21$ sed 'N; /System\nAdmin/D' data4.txt
Admin group meeting will be held.
All System Admins should attend.
```

这里有个例子，删除数据流中出现在第一行之前的空行：

```
lxc@Lxc:~/scripts/ch21$ cat data5.txt

Header Line
First Data Line

End of Data Lines
lxc@Lxc:~/scripts/ch21$ sed '/^$/{N; /Header/D}' data5.txt
Header Line
First Data Line

End of Data Lines
```

sed编辑器脚本会查找空行，然后用 `N` 命令将下一行加入模式空间。如果模式空间中含有单词 *Header*，则 `D` 命令会删除模式空间中的第一行。如果不综合使用 `D` 命令和 `N` 命令，无法做到在不删除其他行的情况下只删除第一个空行。

3. 多行打印命令

多行打印命令(P)，它只打印模式空间中的第一行，即打印模式空间中换行符及其之前的所有字符。当用 `-n` 选项抑制脚本输出时，它和显示文本的单行 `p` 命令的用法大同小异:

```
lxc@Lxc:~/scripts/ch21$ cat data3.txt
On Tuesday, the Linux System
Admin group meeting will be held.
All System Admins should attend.
Thank you for your cooperation.
lxc@Lxc:~/scripts/ch21$ sed -n 'N; /System\nAdmin/P' data3.txt
On Tuesday, the Linux System
```

来看一下sed的用户手册是怎样说明 `D` 命令的:

> If pattern space contains no newline, start a normal new cycle as if the d command was issued. Other - wise, delete text in the pattern space up to the first newline, and restart cycle with the resultant pattern space, without reading a new line of input.
> 补充: `d` 命令的用户手册说明
>
> > d Delete pattern space. Start next cycle.

(还是得看手册，终于解决了困惑。)

当出现多行匹配时，匹配命令只打印模式空间中的第一行。该命令的强大之处在于其和 `N` 命令以及和 `D` 命令配合使用的时候。
`D` 命令的独特之处在于其删除模式空间的第一行之后，会强制sed编辑器返回到脚本的起始处（在模式空间中含有新行的情况下，如果不含有新行则会开启新的循环，我觉得书上说的有误，所以补充了这句），对当前模式空间的内容重新执行此循环（`D` 命令不会从数据流中读取新行）。在脚本中加入 `N` 命令，就能单步扫过(single-step through)整个模式空间，对多行进行匹配。接下来，先使用 `P` 命令打印第一行，然后使用 `D` 命令删除第一行并绕回到脚本的起始处，接着 `N` 命令会读取下一行文本并重新开始此过程。这个循环会一直持续到数据流结束。

```
lxc@Lxc:~/scripts/ch21$ cat corruptData.txt
Header Line#
@
Data Line #1
Data Line #2#
@
End of Data Lines#
@
lxc@Lxc:~/scripts/ch21$ sed -n '
N
s/#\n@//
P
D
' corruptData.txt
Header Line
Data Line #1
Data Line #2
End of Data Lines
lxc@Lxc:~/scripts/ch21$ sed -n '
N
s/#\n@//
```

```
p
D
' corruptData.txt
Header Line
Data Line #1
Data Line #2#
Data Line #2
End of Data Lines
```

corruptData.txt 是一个被破坏的数据文件，在一些行的末尾有 #符号，接着在下一行有 @。为了解决这个问题，可以使用sed将 *Header Line#* 行载入模式空间，然后用 `N` 命令载入第二行 @，将其附加到模式空间内的第一行之后。替换命令用空值来替换删除违规数据(#\n@)，然后 `P` 命令打印模式空间中已经清理过的第一行。`D` 命令将第一行从模式空间中删除。因为模式空间中不含有新行，所以本次循环结束。下次循环，sed编辑器读入下一行 *Data Line #1*，然后 `N` 命令读入下一行 *Data Line #2#*，替换命令模式不匹配，`P` 命令打印第一行 *Data Line #1*，然后 `D` 命令删除第一行，因为模式空间仍存有新行，所以重新开始此次循环。`N` 命令读入下一行 @，模式空间中存在的行匹配替换命令的模式，替换后，注意替换命令删除了匹配的模式，`P` 命令打印第一行 *Data Line #2*，`D`命令删除第一行。因为此时模式空间为空，所以本轮循环结束。sed编辑器开始下次循环，读入新行 *End of Data Lines#*，`N` 命令读入新行 @ 附加到模式空间，替换命令模式匹配，`P` 命令打印处理后的第一行 *End of Data Lines*，`D` 命令删除该行，之后模式空间为空，本次循环结束，数据流也已读取完毕，sed编辑器退出。

2. 保留空间

模式空间(pattern space) 是一块活跃的缓冲区，在sed编辑器执行命令时保存着带检查的文本，但它并不是sed编辑器保存文本的唯一空间。
sed编辑器还有另一块称作 **保留空间(hold space)** 的缓冲区。当你在处理模式空间中的某些行时，可以用保留空间临时保存部分行。与保留空间相关的命令有5个，如下表所示：

命令	描述
h	将模式空间复制到保留空间
H	将模式空间追加到保留空间
g	将保留空间复制到模式空间
G	将保留空间追加到模式空间
x	交换模式空间和保留空间的内容

通常，使用 `h` 或 `H` 命令将字符串移入到保留空间之后，最终还要使用 `g`、`G` 或 `x` 命令将保存的字符串移回模式空间。

来个例子：

```
lxc@Lxc:~/scripts/ch21$ sed -n '/First/ {
> h;p;
> n;p;
> g;p}
> ' data2.txt
First Data Line
Second Data Line
First Data Line
```

我们来一步一步讲解这段代码。

1. sed脚本使用正则表达式作为地址，过滤出含有单词 *First* 的行。
2. 当出现含有单词 *First* 的行时，{} 中的第一个命令 `h` 会将该行复制到保留空间（保留空间的默默认值是一个空行，后续会讲到，）。这时，模式空间和保留空间的内容是一样的。
3. `p` 命令会打印模式空间的内容（First Data Line），也就是被复制进保留空间中的那一行。
4. `n` 命令会读取数据流中的下一行（Second Data Line），将其放入模式空间，注意与 `N` 命令将新内容附加到模式空间中已有内容之后不同，`n` 命令放入的内容会覆盖模式空间中的内容。现在模式空间和保留空间的内容不一样了。
5. `p` 命令会打印模式空间的内容（Second Data Line）。
6. `g` 命令会将保留空间的内容放回模式空间，替换模式空间的当前文本。模式空间和保留空间的内容现在又相同了。
7. `p` 命令会打印模式空间的当前行(First Data Line)。

3. 排除命令

第19章展示过sed编辑器如何将命令应用于数据流中的每一行或是由单个地址或地址区间指定的多行。我们也可以指示命令 **不应用于** 数据流中的特定地址或地址区间。
感叹号(!)命令用于排除(negate)命令，也就是让原本会起作用的命令失效。

```
lxc@Lxc:~/scripts/ch21$ sed -n '/Header/!p' data2.txt
First Data Line
Second Data Line
End of Data Lines
```

正常的 `p` 命令只打印 *data2.txt* 文件中包含单词 *Header* 的那一行。加了感叹号之后，情况反过来了，除了包含单词 *Header* 的那一行，文件中的其他行都被打印。

21.1.1节展示过一种情况，sed编辑器无法处理数据流中的最后一行文本，因为之后再没有其他行了。也可以用感叹号来解决这个问题：

```
lxc@Lxc:~/scripts/ch21$ cat data4.txt
On Tuesday, the Linux System
Admin group meeting will be held.
All System Admins should attend.
lxc@Lxc:~/scripts/ch21$ sed '$!N;
> s/System\nAdmin/DevOps\nEngineer/
> s/System Admin/DevOps Engineer/
> ' data4.txt
On Tuesday, the Linux DevOps
Engineer group meeting will be held.
All DevOps Engineers should attend.
```

在该例中，当sed编辑器读到最后一行时，不执行 `N` 命令，但会对其他行执行 `N` 命令。
这种方法可以反转数据流中文本行的先后顺序。要实现这种效果，需要利用保留空间做一些特别的工作。
为此，可以使用sed做以下工作：

1. 在模式空间放置一行文本
2. 将模式空间中的文本复制到保留空间。

3. 在模式空间中放置下一行文本。
4. 将保留空间中的内容附加到模式空间。
5. 将模式空间的所有内容复制到保留空间。
6. 重复执行第3～5步，直到将所有文本行以反序放入保留空间。
7. 提取并打印文本行

在使用这种方法时，你不想在处理行的时候打印。这意味你要使用sed的 `-n` 选项。然后要决定如何将保留空间的内容附加到模式空间的文本之后。这可以使用 `G` 命令实现。唯一的问题是你不想将保留空间的文本附加到要处理的第一行文本之后。这可以使用感叹号命令轻松搞定：

```
1!G
```

接下来就是将新的模式空间（包含已反转的行）放入保留空间。这也不难，用 `h` 命令即可。
将模式空间的所有文本都反转之后，只需打印结果。当到达数据流中的最后一行时，你就得到了模式空间所有内容。要打印结果，可以使用如下命令：

```
$p
```

以上就是创建可以反转文本行的sed编辑器脚本所需要的操作步骤。

```
lxc@Lxc:~/scripts/ch21$ cat data2.txt
Header Line
First Data Line
Second Data Line
End of Data Lines
lxc@Lxc:~/scripts/ch21$ sed -n '{1!G; h; $p}' data2.txt
End of Data Lines
Second Data Line
First Data Line
Header Line
```

> 有一个现成的bash shell命令可以实现同样的效果：`tac` 命令会以倒序显示文本文件。这个命令的名字也很奇妙，因为它的功能正好和 `cat` 命令相反，所以也采用了相反的命令。

4. 改变执行流程

通常，sed编辑器会从脚本的顶部开始，一直执行到脚本的结尾（`D` 命令是个例外，它会强制sed编辑器在不读取新行的情况下返回到脚本的顶部）。sed编辑器提供了一种方法，可以改变脚本的执行流程，其效果与结构化编程类似。

1. 分支

sed编辑器还提供了一种方法，这种方法可以基于地址、地址模式或地址区间排除一整段命令。这允许你只对数据流中的特定部分执行命令。

分支命令格式如下：

```
[address] b[label]
```

address 参数决定了哪些行会触发命令。*label* 参数定义了要跳转到位置。如果没有 *label* 参数，则跳过触发分支命令的行，继续处理余下的文本。

下面这个例子使用了分支命令的 *address* 参数，但未指定 *label*：

```
lxc@Lxc:~/scripts/ch21$ cat data2.txt
Header Line
First Data Line
Second Data Line
End of Data Lines
lxc@Lxc:~/scripts/ch21$ sed '{2,3b
> s/Line/Replacement/
> }' data2.txt
Header Replacement
First Data Line
Second Data Line
End of Data Replacements
```

如你所见，分支命令在第2、3行跳过了替换命令。

如果不想跳转到脚本末尾，可以定义 *label* 参数，指定分支命令要跳转到的位置。标签以冒号开始，最多可以有7个字符：

```
:label2
```

要指定 *label*，把它放在分支命令之后即可。有了标签，就可以使用其他命令处理匹配分支 *address* 的那些行。对于其他行，仍然沿用脚本中原先的命令处理。

```
lxc@Lxc:~/scripts/ch21$ sed '{/First/b jump1;
> s/Line/Replacement/
> :jump1
> s/Line/Jump Replacement/
> }' data2.txt
Header Replacement
First Data Jump Replacement
Second Data Replacement
End of Data Replacements
```

分支命令指定，如果文本行中出现 *First*，则程序应该跳转到标签为 *jump1* 的脚本行。如果文本行不匹配分支 *address*，则sed编辑器会继续执行脚本中的命令，包括标签 *jump1* 之后的命令。（因此，两个替换命令都被应用于不匹配分支 *address* 的行。当然，在第一个替换命令将 *Line* 替换成 *Replacement* 之后，第二个替换命令就不能匹配到 *Line* 的模式了）。如果某行匹配分支 *address*，那么sed编辑器就会跳转到带有分支标签 *jump1* 的那一行，因此只有最后一个替换命令会被执行。

这个例子演示了跳转到sed脚本下方的标签。你也可以像下面这样，跳转到靠前的标签，达到循环的效果：

```
lxc@Lxc:~/scripts/ch21$ echo "This, is, a, test, to, remove, commas." |
> sed -n {'
> :start
> s/,//1p
> b start
> }'
This is, a, test, to, remove, commas.
This is a, test, to, remove, commas.
This is a test, to, remove, commas.
This is a test to, remove, commas.
This is a test to remove, commas.
This is a test to remove commas.
^C
```

脚本每次迭代都会删除文本中的第一个逗号并打印字符串。这个脚本有一个问题。永远不会结束。这就形成一个死循环，不停的查找逗号，直到使用 Ctrl+C 组合键发送信号，手动停止脚本。

为了避免这种情况，可以为分支命令指定一个地址模式。如果模式不匹配，就不会再跳转：

```
lxc@Lxc:~/scripts/ch21$ echo "This, is, a, test, to, remove, commas." |
> sed -n '{
> :start
> s/,//1p
> /,/b start
> }'
This is, a, test, to, remove, commas.
This is a, test, to, remove, commas.
This is a test, to, remove, commas.
This is a test to, remove, commas.
This is a test to remove, commas.
This is a test to remove commas.
```

现在分支命令只会在行中有逗号的情况下跳转。在最后一个逗号被删除后，分支命令不再执行，脚本结束。

2. 测试

与分支命令类似，测试（t）命令也可以改变sed编辑器脚本的执行流程。测试命令会根据先前替换命令的结果跳转到某个 *label* 处，而不是根据 *address* 进行跳转。
如果替换命令成功匹配并完成了替换，测试命令就会跳转到指定的标签。如果替换命令未能匹配指定的模式，测试命令就不会跳转。

测试命令的格式与分支命令相同：

```
[address]t [label]
```

跟分支命令一样，在没有指定 *label* 的情况下，如果测试成功，sed会跳转到脚本结尾。

测试命令提供了一种低成本的方法来对数据流中的文本执行 *if-then* 语句。如果需要做二选一的替换操作，也就是执行这个替换就不执行另一个替换，那么测试命令可以助你一臂之力（无须指定 *label*）：

```
lxc@Lxc:~/scripts/ch21$ sed '{s/First/Matched/; t
> s/Line/Replacement/
> }' data2.txt
Header Replacement
Matched Data Line
Second Data Replacement
End of Data Replacements
```

第一个替换命令会查找模式文本 *First*。如果匹配了行中的模式，就替换文本，而且测试命令会跳过后面的替换命令。如果第一个替换未能匹配，则执行第二个替换命令。

有了替换命令，就能避免之前用分支命令形成的死循环：

```
lxc@Lxc:~/scripts/ch21$ echo "This, is, a, test, to, remove, commas." |
> sed -n '{
> :start
> s/,//1p
> t start
> }'
This is, a, test, to, remove, commas.
This is a, test, to, remove, commas.
This is a test, to, remove, commas.
This is a test to, remove, commas.
This is a test to remove, commas.
This is a test to remove commas.
```

当没有逗号可以替换时，测试命令不再跳转，而是继续执行剩下的脚本（在本例中，也就是结束脚本）。

5. 模式替换

加入你想为行中匹配的单词加上引号。如果只是要匹配某个单词，那非常简单：

```
lxc@Lxc:~/scripts/ch21$ echo "The cat sleeps in his hat" |
> sed 's/cat/"cat"/'
The "cat" sleeps in his hat
```

但如果在模式中用点号来匹配多个单词呢？

```
lxc@Lxc:~/scripts/ch21$ echo "The cat sleeps in his hat." |  sed
's/.at/".at"/g'
The ".at" sleeps in his ".at".
```

结果并不如意，下面介绍解决方法。

1. `&` 符号

sed编辑器提供了一种解决方法。`&` 符号可以代表替换命令中的匹配模式。不管模式匹配到的是什么样的文本，都可以使用 `&` 符号代表这部分内容。这样就能处理匹配模式的任何单词了：

```
lxc@Lxc:~/scripts/ch21$ echo "The cat sleeps in his hat." | sed 's/.at/"&"/g'
The "cat" sleeps in his "hat".
```

2. 替换单独的单词

`&` 符号代表替换命令中指定模式所匹配的字符串。但有时候，你只想获取该字符串的一部分。当然可以这样做，不过有点难度。
sed编辑器使用圆括号来定义替换模式中的子模式。随后使用特殊的字符串来引用（称作 **反向引用(back reference)**）每个子模式所匹配到的文本。反向引用由反斜线和数字组成。数字表明子模式的符号，第一个子模式为 `\1`，第二个子模式为 `\2`，以此类推。

> **注意：**，在替换命令中使用圆括号时，必须使用转义字符，以此表明这不是普通的圆括号，而是用于划分子模式。这跟转义其他特殊字符正好相反。

来个反向引用的例子：

```
lxc@Lxc:~/scripts/ch21$ echo "The Guide to Programming" |  sed '
s/\(Guide to\) Programming/\1 DevOps/'
The Guide to DevOps
```

这个替换命令将 *Guide to* 放入圆括号，将其标示为一个子模式。然后使用 `\1` 来提取此子模式匹配到的文本。

如果需要用一个单词来替换一个短语，而这个单词又正好是该短语的子串，但在子串中用到了特殊的模式字符，那么这时使用子模式将会方便很多：

```
lxc@Lxc:~/scripts/ch21$ echo "That furry cat is pretty." |
> sed 's/furry \(.at\)/\1/'
That cat is pretty.
```

在这种情况下，不能用 `&` 符号，因为其代表的是整个模式所匹配到的文本。而反向引用则允许将某个子模式匹配到的文本作为替换内容。

当需要在两个子模式间插入文本时，这个特性尤其有用。下面的脚本使用子模式在大数中插入逗号：

```
lxc@Lxc:~/scripts/ch21$ echo "1234567" |
> sed '{
> :start
> s/\(.*[0-9]\)\([0-9]\{3\}\)/\1,\2/
> t start}'
1,234,567
```

这个脚本将匹配模式分成了两个子模式：

- .*[0-9]
- [0-9]{3}

sed编辑器会在文本行中查找这两个子模式。第一个子模式是以数字结尾的任意长度字符串。第二个子模式是3位数字。如果匹配到了相应的模式，就在两者之间加一个逗号，每个子模式都通过其序号来标示。这个脚本使用测试命令来遍历这个大数，直到所有的逗号都插入完毕。

6. 在脚本中使用sed

本节将演示一些你应该已经知道的一些特性，在bash shell脚本中使用sed编辑器时能够用到它们。

1. 使用包装器

编写sed脚本的过程很烦琐，尤其是当脚本很长的时候。你可以将sed编辑器命令放入脚本 **包装器**。这样就不用每次都重新键入整个脚本。包装器充当着sed编辑器脚本和命令行之间的中间人的角色。shell脚本包装器 ChangeScriptShell.sh 在第19章的时候作为实例出现过。

来个例子:

reverse.sh

```
#!/bin/bash
# Shell wrapper for sed editor script
# to reverse test file lines.
#
sed -n '{1!G; h; $p}' $1
#
exit
# output:
lxc@Lxc:~/scripts/ch21$ cat data2.txt
Header Line
First Data Line
Second Data Line
End of Data Lines
lxc@Lxc:~/scripts/ch21$ ./reverse.sh data2.txt
End of Data Lines
Second Data Line
First Data Line
Header Line
```

2. 重定向sed的输出

在shell脚本中，你可以用 *命令替换* 来将sed编辑器命令的输出重定向到一个变量中，以备后用。

fact.sh

```
#!/bin/bash
# Shell wrapper for sed editor script
# to calaulate a factorial, and
# format the result with commas.
#
factorial=1
counter=1
number=$1
#
while [ $counter -le $number ]
do
    factorial=$[ $factorial * $counter ]
    counter=$[ $counter + 1 ]
done
#
result=$(echo $factorial |
sed '{
:start
s/\(.*[0-9]\)\([0-9]\{3\}\)/\1,\2/
t start
```

```
}')
#
echo "The result is $result"
#
exit
# output:
lxc@Lxc:~/scripts/ch21$ ./fact.sh 20
The result is 2,432,902,008,176,640,000
```

7. 创建sed实用工具

本节将展示一些方便趁手且众所周知的sed编辑器脚本，从而帮助你完成常见的数据处理工作。

1. 加倍行间距

首先，来看一个向文本文件的行间插入空行的简单sed脚本。

```
lxc@Lxc:~/scripts/ch21$ sed 'G' data2.txt
Header Line

First Data Line

Second Data Line

End of Data Lines

```

这个技巧的关键在于保留空间的默认值。`G` 命令只是将保留空间的内容附加到模式空间内容之后。当启动sed编辑器时，保留空间只有一个空行。将它附加到已有行之后，就创建了空行。但是，在最后一行之后我们也插入了空行，我们可以使用排除命令来确保脚本不会将空行插入到数据流的最后一行之后。

```
lxc@Lxc:~/scripts/ch21$ sed '$!G' data2.txt
Header Line

First Data Line

Second Data Line

End of Data Lines
```

2. 对可能含有空行的文件加倍行间距

将上面的例子再扩展一步，如果文本文件已经有一些空行，但你想给所有行加倍行间距(即每个行除最后一行外后面只有一个空行)，如果沿用前面的脚本，则有些区域会有太多空行，因为已的空行也会被加倍(即空行也被算作一行，空行后面还会再放入空行)。

```
lxc@Lxc:~/scripts/ch21$ cat data6.txt
Line one.
Line two.

Line three.
Line four.
lxc@Lxc:~/scripts/ch21$ sed '$!G' data6.txt
```

```
Line one.

Line two.



Line three.

Line four.
```

原来是有一个空行的的位置现在有3个空行了。解决办法是先删除所有空行，再在每行除最后一行外再插入空行。

```
lxc@Lxc:~/scripts/ch21$ sed '/^$/d; $!G' data6.txt
Line one.

Line two.

Line three.

Line four.
```

3. 给文件中的行编号

第 19 章演示过如何使用等号来显示数据流中行的行号：

```
lxc@Lxc:~/scripts/ch21$ sed '=' data2.txt
1
Header Line
2
First Data Line
3
Second Data Line
4
End of Data Lines
```

这多少有点难看。

解决方法如下：

```
lxc@Lxc:~/scripts/ch21$ sed '=' data2.txt | sed 'N; s/\n/ /'
1 Header Line
2 First Data Line
3 Second Data Line
4 End of Data Lines
```

使用 `N` 命令合并行，使用替换命令将换行符替换成空格或者制表符，不再赘述。

有些bash shell命令也能添加行号。但是会引入一些额外的(可能是不需要的)间隔：

```
lxc@Lxc:~/scripts/ch21$ nl data2.txt
     1  Header Line
     2  First Data Line
     3  Second Data Line
```

```
     4  End of Data Lines
lxc@Lxc:~/scripts/ch21$ cat -n data2.txt
     1  Header Line
     2  First Data Line
     3  Second Data Line
     4  End of Data Lines
lxc@Lxc:~/scripts/ch21$ nl data2.txt | sed 's/     //; s/\t/ /'
1 Header Line
2 First Data Line
3 Second Data Line
4 End of Data Lines
# 在sed编辑器命令中第一个替换是5个空格。
```

4. 打印末尾行

本节展示使用滑动窗口的方法来显示数据流末尾的若干行。

```
lxc@Lxc:~/scripts/ch21$ cat data7.txt
Line1
Line2
Line3
Line4
Line5
Line6
Line7
Line1
Line2
Line3
Line4
Line5
Line6
Line7
Line8
Line9
Line10
Line11
Line12
Line13
Line14
Line15
lxc@Lxc:~/scripts/ch21$ sed '{
> :start
> $q; N; 11,$D
> b start
> }' data7.txt
Line6
Line7
Line8
Line9
Line10
Line11
Line12
Line13
Line14
```

```
Line15
```

该脚本首先检查当前行是否是数据流中的最后一行。如果是，则退出命令(`q`)会停止循环， `N` 命令会将下一行附加到模式空间中的当前行之后。如果当前行在第10行之后，则 `11,$D` 命令会删除模式空间中的第1行。这就在模式空间创建了类似滑动窗口的效果。因此，这个sed脚本只会显示 *data7.txt* 文件的最后10行。

5. 删除行

本节给出了3个简洁的sed编辑器脚本，用来删除数据中不需要的空行。

1. 删除连续的多行

删除连续空行的关键在于创建包含一个空行和非空行的地址空间。如果sed编辑器遇到了这个区间，它不会删除行。但对于不属于该区间的行（两个或者多个空行），则执行删除操作。

下面是完成该操作的脚本：

```
/./,/^$/!d
```

该命令使用了两个模式匹配，第一个模式匹配作为地址区间的起始地址，第二个模式匹配作为地址区间的结束地址。指定的区间是 `/./` 到 `/^$/` 。区间的开始地址会匹配至少含有一个字符的行。区间的结束地址会匹配一个空行。然后使用了排除命令，在这个区间内的行不会被删除。

```
lxc@Lxc:~/scripts/ch21$ cat data8.txt
Line one.


Line two.

Line three.




Line four.
lxc@Lxc:~/scripts/ch21$ sed '/./,/^$/!d' data8.txt
Line one.

Line two.

Line three.

Line four.
```

如你所见，不管文件的数据行之间有多少空行，在输出中只保留一个空行。

2. 删除开头的空行

该脚本用于删除数据流中开头的空行。

```
lxc@Lxc:~/scripts/ch21$ cat data9.txt


Line one.

Line two.
lxc@Lxc:~/scripts/ch21$ sed '/./,$!d' data9.txt
Line one.

Line two.
```

该脚本使用模式匹配作为地址区间的起始地址，`$`为地址区间的结束地址。

3. 删除结尾的空行

删除结尾的空行不像删除开头的空行那么简单。要利用循环实现：

```
lxc@Lxc:~/scripts/ch21$ cat data10.txt
Line one.
Line two.



lxc@Lxc:~/scripts/ch21$ sed '{
> :start
> /^\n*$/{$d; N; b start}
> }' data10.txt
Line one.
Line two.
```

在该脚本中，`/^\n*$/`这是一个模式匹配，其中用到了组合锚点。该组合锚点是想匹配这样的一行：以0个或多个(星号)(实际上一行最多一个换行符嘛)以`\n`开头和结尾的行(0个换行符就是最后一行空行，1个换行符就是最后一行空行之前的空行)。分支命令使用了地址模式，见分支的最后部分。 如果模式匹配，如果是最后一行则执行`d`命令，删除模式空间中的行。如果不是最后一行那么`N`命令会将下一行附加到它后面，然后分支命令跳转到循环开始处重新开始。

该脚本成功删除了文本文件结尾的空行，同时保持了其他空行未变。

6. 删除HTML标签

如题，删除HTML标签，大多数HTML标签是成对出现的：一个起始标签(比如用来加粗)和一个闭合标签(比如用来结束加粗)。

乍一看，你可能认为删除HTML标签就是查找以小于号(<)开头、大于号(>)结尾且其中包含数据的字符串：

```
s/<.*>//g
```

但这个命令可能会造成一些意想不到的结果：

```
lxc@Lxc:~/scripts/ch21$ sed 's/<.*>//g' data11.txt
```

```


This is the  line in the Web page.
This should provide some
information to use in our sed script.

```

注意标题文本以及加粗和倾斜的文本都不见了。sed编辑器忠实地将这个脚本理解为小于号和大于号之间的任何文本，包括前者的小于号和后者的大于号(因为 `*` 属于贪婪型量词)。为此，要让sed编辑器忽略任何嵌入原始标签中的大于号。可以使用排除型字符组来排除大于号，将脚本改为如下形式：

```
s/<[^>]*>//g
```

现在这个脚本就能正常显示Web页面中的数据了：

```
lxc@Lxc:~/scripts/ch21$ sed 's/<[^>]*>//g' data11.txt


This is the page title



This is the first line in the Web page.
This should provide some useful
information to use in our sed script.

```

可以删除多余的空行来使结果更清晰：

```
lxc@Lxc:~/scripts/ch21$ sed 's/<[^>]*>//g; /^$/d' data11.txt
This is the page title
This is the first line in the Web page.
This should provide some useful
information to use in our sed script.
```

8. 实战演练

搞一个脚本，该脚本扫描bash shell脚本，找出适合放入函数的一些重复行。

NeededFunctionCheck.sh

```
#!/bin/bash
# Checks for 3 duplicate lines in scripts.
# Suggest these lines are possible replaced
# by a function.
#
```

```
tempfile=$2
#
#
sed -n '{
1N; N;
s/ //g; s/\t//g;
s/\n/\a/g; p;
s/\a/\n/; D}' $1 >> $tempfile
#
sort $tempfile | uniq -d | sed 's/\a/\n/g'
#
rm -i $tempfile
#
exit
# output:
lxc@Lxc:~/scripts/ch21$ ./NeededFunctionCheck.sh ScriptDataB.txt TempFile.txt
Line3
Line4
Line5
rm: 是否删除普通文件 'TempFile.txt'?  y
lxc@Lxc:~/scripts/ch21$ ./NeededFunctionCheck.sh CheckMe.sh TempFile.txt
echo"Usage:./CheckMe.shparameter1parameter2"
echo"Exitingscript..."
exit
rm: 是否删除普通文件 'TempFile.txt'?  y
```

ch22 gawk进阶

gawk是一种功能丰富的编程语言，提供了各种用于编写高级数据处理程序的特性。在本章中，你将看到如何使用gawk编写程序，处理可能遇到的各种数据格式化任务。

1. 使用变量

gawk编程语言支持两类变量。

- 内建变量
- 自定义变量

gawk内建变量包含用于处理数据文件中的数据字段和记录的信息。

1. 内建变量

gawk脚本使用内建变量来引用一些特殊的功能。

1. 字段和记录分隔符变量

在第 19 章演示过gawk的一种内建变量——**数据字段变量**。数据字段变量允许使用美元符号和字段在记录中的位置值来引用对应的字段。因此，要引用记录中的第一个数据字段，就用变量 `$1`，要引用第二个数据字段就用 `$2`，以此类推。
数据字段由字段分隔符划定。在默认情况下，字段分隔符就是一个空白字符，也就是空格或者制表符。
第 19 章讲过如何使用命令行选项 `-F`，或是在gawk脚本中使用特殊内建变量 `FS` 修改字段分隔符。
有一组内建变量可以控制gawk对输入数据和输出数据中字段和记录的处理方式。下表列出了这些内建变量。

变量	描述
FIELDWIDTHS	由空格分隔一列数字，定义了每个数据字段的确切宽度
FS	输入字段分隔符
RS	输入记录分隔符
OFS	输出字段分隔符
ORS	输出记录分隔符

前文介绍过如何使用变量 `FS` 定义记录中的字段分隔符，变量 `OFS` 具有相同的功能，只不过是用于 `print` 命令的输出。

默认情况下，gawk会将 `OFS` 变量设置为一个空格。

来个例子：

```
lxc@Lxc:~/scripts/ch22$ cat data1
data11,data12,data13,data14,data15
data21,data22,data23,data24,data25
data31,data32,data33,data34,data35
lxc@Lxc:~/scripts/ch22$ gawk 'BEGIN{FS=","}{print $1,$2,$3}' data1
data11 data12 data13
data21 data22 data23
data31 data32 data33
```

如你所见，`print` 命令自动会将 `OFS` 变量的值置于输出的每个字段之间。通过设置 `OFS` 变量，可以在输出中用任意字符串来分隔字段：

```
lxc@Lxc:~/scripts/ch22$ gawk 'BEGIN{FS=","; OFS="-"}{print $1, $2, $3}' data1
data11-data12-data13
data21-data22-data23
data31-data32-data33
lxc@Lxc:~/scripts/ch22$ gawk 'BEGIN{FS=","; OFS="<-->"}{print $1, $2, $3}'
data1
data11<-->data12<-->data13
data21<-->data22<-->data23
data31<-->data32<-->data33
```

`FIELDWIDTHS` 变量可以不通过字段分隔符读取记录。有些应用程序并没有使用字段分隔符，而是将数据放置在记录中的特定列。在这种情况下，必须设定 `FIELDWIDTHS` 变量来匹配数据在记录中的位置。
一旦设置了 `FIELDWIDTHS` 变量，gawk会忽略 `FS` 变量，并根据提供的字段宽度来计算字段。

来个例子：

```
lxc@Lxc:~/scripts/ch22$ cat data1b
1005.3247596.37
115-2.349194.00
05810.1298100.1
lxc@Lxc:~/scripts/ch22$ gawk 'BEGIN{FIELDWIDTHS="3 5 2 5 "}{print
$1,$2,$3,$4}' data1b
100 5.324 75 96.37
115 -2.34 91 94.00
058 10.12 98 100.1
```

`FIELDWIDTHS` 变量定义了4个数据字段，gawk以此解析记录。每个记录中的数字串会根据已定义好的字段宽度来分割。

> 一定要记住，一旦设定了 `FIELDWIDTHS` 变量的值，就不能再改动了。这种方法并不适用于变长的数据字段。

变量 `RS` 和 `ORS` 定义了gawk对数据流中记录的处理方式。在默认情况下，gawk会将 `RS` 和 `ORS` 设置为换行符。也就是说，默认情况下，输入数据流中的一行文本就是一条记录。

有时，我们会遇到数据流中占据多行的记录。如果使用默认的 `FS` 变量和 `RS` 变量来读取数据，gawk就会把每一行当作一条单独的记录来读取，并将其中的空格作为字段分隔符，这当然不是我们希望看到的。
为此，我们需要把 `FS` 变量设置为换行符。这表明数据流中的每一行都是一个单独的字段。把 `RS` 变量设置为空字符串。然后我们需要在文本数据的记录之间留一个空行。gawk会把每一个空行都视为记录分隔符。

```
lxc@Lxc:~/scripts/ch22$ cat data2
Ima Test
123 Main Street
Chicago, IL  60601
(312)555-1234

Frank Tester
456 Oak Street
Indianapolis, IN  46201
(317)555-9876

Haley Example
4231 Elm Street
Detroit, MI 48201
(313)555-4938
lxc@Lxc:~/scripts/ch22$ gawk 'BEGIN{FS="\n";RS=""}{print $1,$4}' data2
Ima Test (312)555-1234
Frank Tester (317)555-9876
Haley Example (313)555-4938
```

现在，gawk会把文件中的每一行都视为一个字段，将空行视为记录分隔符。

2. 数据变量

除了字段和记录分隔符，gawk还提供了一些其他的内建变量以帮助你了解数据发生了什么变换，并提取shell环境信息。下表列出了gawk中的其它内建变量。

变量	描述
ARGC	命令行参数的数量
ARGIND	当前处理的文件在 *ARGV* 中的索引
ARGV	包含命令行参数的数组
CONVFMT	数字的转换格式(参见 `printf` 语句)，默认值为 %.6g
ENVIRON	当前shell环境变量及其值组成的关联数组
ERRNO	当读取或关闭文件发生错误时的系统错误号
FILENAME	用作gawk输入的数据文件的名称
FNR	当前数据文件中的记录数
IGNORECASE	设成非0值时，忽略gawk命令中出现的字符串的大小写
NF	数据文件中的字段总数
NR	已处理的输入记录数
OFMT	数字的输出显示格式。默认值为%.6g.，以浮点数或科学计数法显示，以较短者为准，最多使用6位小数
RLENGTH	由 `match` 函数所匹配的子串的长度
RSTART	由 `match` 函数所匹配的子串的起始位置

变量 `ARGC` 和 `ARGV` 允许从shell中获取命令行参数的总数及其值。有点麻烦的地方在于gawk并不会将程序脚本视为命令行参数的一部分：

```
lxc@Lxc:~/scripts/ch22$ gawk 'BEGIN{print ARGC, ARGV[0], ARGV[1]}' data1
2 gawk data1
```

`ARGV` 数组从索引0开始，代表的是命令。第一个数组值是gawk命令后的第一个命令行参数。

> 跟shell变量不同，在脚本中引用gawk变量时，变量名前不用加美元符号。

ENVIRON 变量看起来有点陌生。它使用 **关联数组** 来提取shell环境变量。关联数组用文本（而非数值）来作为数组索引。
数组索引中的文本是shell环境变量名，对应的数组元素值是shell环境变量的值。

```
lxc@Lxc:~/scripts/ch22$ gawk '
> BEGIN{
> print ENVIRON["HOME"]
> print ENVIRON["PATH"]
> }'
/home/lxc
/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap
/bin
# 省略了PATH环境变量的输出
```

`NF` 变量表示数据文件中的字段总数。可以在 `NF` 变量之前加上美元符号，将其用作 *字段变量*。

```
lxc@Lxc:~/scripts/ch22$ gawk 'BEGIN{FS=":";OFS=":"}{print $1, $NF}'
/etc/passwd
root:/bin/bash
daemon:/usr/sbin/nologin
bin:/usr/sbin/nologin
......
```

`FNR` 变量和 `NR` 变量类似，但略有不同。*FNR* 变量包含 *当前数据文件中已处理过的记录数*，`NR` 变量则包含 *已处理过的记录总数*。

```
lxc@Lxc:~/scripts/ch22$ gawk '
> BEGIN{FS=","}
> {print $1, "FNR="FNR, "NR="NR}
> END {print "There were", NR, "records processed"}' data1 data1
data11 FNR=1 NR=1
data21 FNR=2 NR=2
data31 FNR=3 NR=3
data11 FNR=1 NR=4
data21 FNR=2 NR=5
data31 FNR=3 NR=6
There were 6 records processed
```

在这个例子中，gawk脚本在命令行指定了两个输入文件（同一个输入文件被指定了两次）。在gawk处理第二个输入文件时，`FNR` 变量的值被重置了，而 `NR` 变量则继续计数。因此，如果只使用一个数据文件作为输入，那么 `FNR` 和 `NR` 的值是相同的；如果使用多个数据文件作为输入，那么 `FNR` 的值会在处理每个数据文件时被重置，`NR` 的值则会继续计数直到处理完所有的数据文件。

2. 自定义变量

gawk自定义变量名称可以由任意数量的字母、数字和下划线组成，但不能以数字开头。gawk变量名区分大小写。

1. 在脚本中给变量赋值

在gawk脚本中给变量赋值与给shell脚本中的变量赋值一样，都用赋值语句：

```
lxc@Lxc:~/scripts/ch22$ gawk '
> BEGIN{testing="This is a test"
> print testing
> }'
This is a test
lxc@Lxc:~/scripts/ch22$ gawk '
> BEGIN{
> testing = "This is a test"
> print testing
> testing = 45
> print testing
> }'
This is a test
45
```

赋值语句还可以包含处理数值的数学算式：

```
lxc@Lxc:~/scripts/ch22$ gawk '
> BEGIN{x = 4; x = x * 2 + 3; print x}'
11
```

gawk编程语言包含了用来处理数值的标准算术运算符，其中包括求余运算符(%)和幂运算符(^或**)。

2. 在命令行中给变量赋值

也可以通过gawk命令行来为脚本中的变量赋值。这允许你在正常的代码之外赋值，即时修改变量值。下面这个例子使用命令行变量来显示文件中特定的数据字段：

```
lxc@Lxc:~/scripts/ch22$ gawk '
> BEGIN{x = 4; x = x * 2 + 3; print x}'
11
lxc@Lxc:~/scripts/ch22$ cat script1
BEGIN{FS=","}
{print $n}
lxc@Lxc:~/scripts/ch22$ gawk -f script1 n=2 data1
data12
data22
data32
```

这个特性可以让你在不修改脚本代码的情况下就改变脚本的行为。

使用命令行参数来定义变量值会产生一个问题，在设置过变量之后，这个值在脚本的 `BEGIN` 部分不可用。可以用 `-v` 选项来解决这个问题，它允许在 `BEGIN` 部分之前设定变量。在命令行中， `-v` 选项必须放在脚本代码之前。

```
lxc@Lxc:~/scripts/ch22$ gawk -f script2 n=2 data1
The starting value is
data12
data22
data32
lxc@Lxc:~/scripts/ch22$ gawk -v n=2 -f script2 data1
The starting value is 2
data12
data22
data32
```

现在， `BEGIN` 部分中的变量n的值就已经是命令行中设定的那个值了。

2. 处理数组

gawk编程语言使用 **关联数组** 来提供数组功能。与数字型数组不同，关联数组的索引可以是任意文本字符串，你不需要用连续的数字来标识数组元素。相反，关联数组用各种字符串来引用数组元素。每个索引字符串都必须能够唯一标识出分配给它的数组元素。类似于其他编程语言中的哈希表或字典。

1. 定义数组变量

可以使用标准的赋值语句来定义数组变量。数组变量赋值的格式如下：

```
var[index] = element
```

其中 *var* 是变量名，*index* 是关联数组的索引值，*element* 是数组元素值。

来个例子：

```
lxc@Lxc:~/scripts/ch22$ gawk 'BEGIN{
> capital["nibaba"] = "niye"
> capital["hahaha"]="lueluelue"
> print capital["nibaba"]
> }'
niye
lxc@Lxc:~/scripts/ch22$ gawk 'BEGIN{
> var[1] = 12
> var[2] = 23
> total = var[1] + var[2]
> print total
> }'
35
```

2. 遍历数组变量

关联数组变量的问题在于，你可能无法预知索引是什么。如果要在gawk脚本中遍历关联数组，可以用 `for` 语句的一种特殊形式：

```
for (var in array)
{
    statement
}
```

这个 `for` 语句会在每次循环时将关联数组 *array* 的下一个索引值赋给变量 *var*，需要注意的是，索引值没有特定的返回顺序，然后执行一遍 *statement*。

```
lxc@Lxc:~/scripts/ch22$ gawk 'BEGIN{
> var["a"] = 1
> var["g"] = 2
> var["m"] = 3
> var["u"] = 4
> for (test in var)
> {
> print "Index:",test, "- Value:",var[test]
> }
> }'
Index: u - Value: 4
Index: m - Value: 3
Index: a - Value: 1
Index: g - Value: 2
```

3. 删除数组变量

从关联数组中删除数组元素要使用一个特殊的命令：

```
delete array[index]
```

`delete` 命令会从关联数组中删除索引值及其相关的数组元素值：

```
lxc@Lxc:~/scripts/ch22$ gawk 'BEGIN{
> var["a"]=1
> var["b"]=2
> for(test in var)
> {
> print "Index:",test,"- Value:", var[test]
> }
> delete var["a"]
> print "----"
> for(test in var)
> {
> print "Index:",test,"- Value:",var[test]
> }
> }'
Index: a - Value: 1
Index: b - Value: 2
----
Index: b - Value: 2
```

3. 使用模式

本节将演示如何在gawk脚本中用匹配模式来限制将脚本作用于哪些记录。

1. 正则表达式

你可以用基础正则表达式（BRE）或扩展正则表达式（ERE）来筛选脚本要作用于数据流中的哪些行。在使用正则表达式时，它必须出现在与其对应脚本的左花括号前。

```
lxc@Lxc:~/scripts/ch22$ gawk 'BEGIN{FS=","} /11/{print $1}' data1
data11
```

2. 匹配操作符

匹配操作符（~）能将正则表达式限制在记录的特定数据字段。你可以指定匹配操作符、数据字段以及要匹配的正则表达式：

```
$1 ~ /^data/
```

`$1` 变量代表记录中的第一个数据字段。该表达式会过滤出第一个数据字段以文本 *data* 开头的所有记录。

来个例子：：

```
lxc@Lxc:~/scripts/ch22$ gawk -F: '$1 ~ /lxc/{print $1,$NF}' /etc/passwd
lxc /bin/bash
```

这个例子会在第一个数据字段查找文本 *lxc*。如果匹配该模式，则打印记录中的第一个数据字段和最后一个数据字段。

也可以用！符号来排除正则表达式的匹配：

```
$1 !~ /expression/
```

来个例子:

```
lxc@Lxc:~/scripts/ch22$ gawk -F: '$1 !~ /lxc/{print $1,$NF}' /etc/passwd
root /bin/bash
daemon /usr/sbin/nologin
bin /usr/sbin/nologin
......
```

在这个例子中，gawk脚本会打印 */etc/passwd* 文件中用户名不是 *lxc* 的那些用户名和登录shell。

3. 数学表达式

除了正则表达式，也可以在匹配模式中使用数学表达式。这个功能在匹配数据字段中的数值时非常方便。

```
lxc@Lxc:~/scripts/ch22$ gawk -F: '$4 == 0{print $1}' /etc/passwd
root
```

该脚本显示所有属于root用户组（组ID为0）的用户，该脚本会检查记录中值为0的第四个字段。在Linux系统中，只有一个用户账户属于root用户组。

可以使用任何常见的数学比较表达式。

- *x* == *y* : *x* 的值等于 *y* 的值
- *x* <= *y* : *x* 的值小于等于 *y* 的值
- *x* < *y* : *x* 的值小于 *y* 的值
- *x* >= *y* : *x* 的值大于等于 *y* 的值
- *x* > *y* : *x* 的值大于 *y* 的值

也可以对文本数据使用表达式，但必须小心。表达式必须完全匹配。数据必须跟模式严格匹配。

```
lxc@Lxc:~/scripts/ch22$ gawk -F, '$1 == "data"{print $1}' data1
lxc@Lxc:~/scripts/ch22$ gawk -F, '$1 == "data11"{print $1}' data1
data11
```

如你所见，第一个测试没有匹配任何记录，因为第一个数据字段的值不在任何记录中。第二个测试用值 *data11* 匹配了一条记录。

4. 结构化命令

gawk编程语言支持常见的结构化编程命令。本节将介绍这些命令并演示如何在gawk编程环境中使用它们。

1. `if` 语句

gawk编程语言支持标准格式的 `if-then-else` 语句。你必须为 `if` 语句定义一个求值的条件，并将其放入圆括号内。

格式如下:

```
if (condition)
    statement
```

也可以写在一行，就像下面这样：

```
if (condition) statement1
```

来个例子：

```
lxc@Lxc:~/scripts/ch22$ cat data4
10
5
13
50
34
lxc@Lxc:~/scripts/ch22$ gawk '{if ($1 > 20) print $1}' data4
50
34
```

如果要在 `if` 语句中执行多条语句，则必须将其放入花括号内：

```
lxc@Lxc:~/scripts/ch22$ gawk '{
> if ($1 > 20)
> {
> x = $1 * 2
> print x
> }
> }' data4
100
68
```

gawk的 `if` 语句也支持 `else` 子句，允许在 `if` 语句不成立的情况下执行一条或多条语句。来个例子：

```
lxc@Lxc:~/scripts/ch22$ gawk '{
> if ($1 > 20)
> {
> x = $1 *2
> print x
> }
> else {
> x = $1 / 2
> print x
> }
> }' data4
5
2.5
6.5
100
68
```

也可以在单行使用 `else` 子句，但必须在 `if` 语句部分之后使用分号：

```
if (condition) statement1; else statement2
```

下面是上一个例子的单行格式版本：

```
lxc@Lxc:~/scripts/ch22$ gawk '{if($1 > 20) print $1 * 2; else print $1 / 2}'
data4
5
2.5
6.5
100
68
```

2. `while` 语句

语句格式：

```
while(condition)
{
    statement
}
```

来个例子：

```
lxc@Lxc:~/scripts/ch22$ cat data5
130 120 135
160 113 140
145 170 215
lxc@Lxc:~/scripts/ch22$ gawk '{
> total = 0
> i = 1
> while(i < 4)
> {
> total += $i
> i++
> }
> avg = total / 3
> print "Avgerage:", avg
> }' data5
Avgerage: 128.333
Avgerage: 137.667
Avgerage: 176.667
```

gawk编程语言支持在 `while` 循环中使用 `break` 语句和 `continue` 语句。

```
lxc@Lxc:~/scripts/ch22$ gawk '{
> total = 0
> i = 1
> while(i < 4)
> {
> total += $i
> if(i == 2)
> break
> i++
> }
> avg = total / 2
> print "Average:",avg
> }' data5
```

```
Average: 125
Average: 136.5
Average: 157.5
```

在 i 等于 2 时，`break` 语句跳出 `while` 循环。

3. `do-while` 语句

语句格式：

```
do
{
    statements
}while(condition)
```

这种格式保证在 *statements* 会在条件被求值前至少被执行一次。

```
lxc@Lxc:~/scripts/ch22$ gawk '{
> total = 0
> i = 1
> do
> {
> total += $i
> i++
> }while(total < 150)
> print total}' data5
250
160
315
```

4. `for` 语句

gawk编程语言支持C风格的 `for` 循环：

```
for(variable assignment; condition; iteration process)
```

来个例子：

```
lxc@Lxc:~/scripts/ch22$ gawk '{
> total = 0
> for(i = 1; i < 4; i++)
> {
> total += $i
> }
> avg = total / 3
> print "Average:", avg
> }' data5
Average: 128.333
Average: 137.667
Average: 176.667
```

2. 格式化打印

gawk编程语言提供了格式化打印命令 `printf`。如果你熟悉C语言，那么gawk中 `printf` 命令的用法也是一样的。

`printf` 命令的格式如下：

```
printf "format string", var1, var2
```

format string 是格式化输出的关键。它会用文本元素和 **格式说明符（format specifier）**来具体指定如何呈现格式化输出。格式说明符是一种特殊的代码，可以指明显示什么类型的变量以及如何显示。gawk脚本会将每个格式说明符作为占位符，供命令中的每个变量使用。第一个格式说明符对应第一个变量，第二个对应第二个变量，以此类推。

格式说明符的格式如下：

```
%[modifier] control-letter
```

其中， *control-letter* 是控制字母，用于指明显示什么类型的数据，*modifier* 是修饰符，定义了可选的格式化特性。

下表列出了在格式说明符中可用的控制字母。

控制字母	描述
c	将数字作为ASCII字符显示
d	显示整数值
i	显示整数值（和 *d* 一样）
e	用科学计数法显示数字
f	显示浮点数
g	用科学计数法或浮点数显示（较短的格式优先）
o	显示八进制
s	显示字符串
x	显示十六进制
X	显示十六进制，但用大写字母A~F

除了控制字母，还有3种修饰符可以进一步控制输出。

- *width* : 指出输出字段的最小宽度。如果输出短于这个值，则 `printf` 语句会将文本右对齐，并用空格进行填充。如果输出比指定的宽度长，则按照实际长度输出。
- *prec* : 指定浮点数中小数点右侧的位数或者字符串中显示的最大字符数。
- -（减号）: 指明格式化空间中的数据采用左对齐而非右对齐。

下面来几个例子：

```
lxc@Lxc:~/scripts/ch22$ gawk 'BEGIN{FS="\n";RS=""}{printf "%s %s\n", $1, $4}'
data2
Ima Test (312)555-1234
Frank Tester (317)555-9876
Haley Example (313)555-4938
```

注意，你需要在 `printf` 命令的末尾手动添加换行符，以便生成新行。否则，`printf` 命令会继续在同一行打印后继续输出。

```
$ gawk 'BEGIN{FS="\n";RS=""}{printf "%16s %s\n", $1, $4}' data2
        Ima Test (312)555-1234
    Frank Tester (317)555-9876
   Haley Example (313)555-4938
```

通过添加一个值为16的修饰符，我们强制第一个字符串的宽度为16字符。在默认情况下，`printf` 命令使用右对齐来讲数据放入格式化空间中。要改为左对齐，只需给修饰符加上一个减号即可：

```
$ gawk 'BEGIN{FS="\n";RS=""}{printf "%-16s %s\n", $1,$4}' data2
Ima Test         (312)555-1234
Frank Tester     (317)555-9876
Haley Example    (313)555-4938
```

`printf` 命令在处理浮点值时也很方便。通过为变量指定格式，可以使输出看起来更为统一：

```
lxc@Lxc:~/scripts/ch22$ gawk '{
> total = 0
> for(i = 1; i < 4; i++)
> {
> total += $i
> }
> avg = total / 3
> printf "Average: %5.1f\n", avg
> }' data5
Average: 128.3
Average: 137.7
Average: 176.7
```

格式说明符 `%5.1f` 强制 `printf` 命令将浮点值近似到小数点后一位。

6. 内建函数

gawk编程语言提供了不少内置函数，以用于执行一些常见的数学、字符串以及时间运算。本节将带你逐步熟悉gawk编程语言中的各种内建函数。

1. 数学函数

下表列出了gawk中内建的数学函数。

函数	描述
atan2(x, y)	x/y的反正切，x和y以弧度为单位

函数	描述
cos(x)	x的余弦，x以弧度为单位
exp(x)	e的x次方
int(x)	x的整数部分，取靠近0的一侧
log(x)	x的自然对数
rand()	比0大且比1小的随机浮点值
sin(x)	x的正弦，x以弧度为单位
sqrt(x)	x的平方根
srand(x)	为计算随机数指定一个种子值

`rand` 函数会返回一个随机数，但这个随机数只在0和1之间（不包括0或1）。要得到更大的数，就需要放大返回值。产生较大随整数的常见方法是综合运用函数 `rand()` 和 `int()` 创建一个算法：

```
x = int(10 * rand())
```

这会返回一个0~9（包括0和9）的随机整数值。只要在程序中用上限值替换等式中的10就可以了。

在使用一些数学函数时要小心，因为gawk编程语言对于能够处理的数值有一个限定区间。如果超出这个区间，就会得到一条错误消息：

```
lxc@Lxc:~/scripts$ gawk 'BEGIN{x=exp(100); print x}'
2688117141816135609425340043596290355468697 6
lxc@Lxc:~/scripts$ gawk 'BEGIN{x=exp(1000); print x}'
gawk: 命令行:1: 警告:  exp: 参数 1000 超出范围
+inf
```

第一个例子计算e的100次幂，虽然这个数值很大但尚在系统的区间以内。第二个例子尝试计算e的1000次幂，这已经超出了系统的数值区间，因此产生了一条错误消息。

除了标准数学函数，gawk还支持一些按位操作数据的函数。

- and(v1, v2) : 对v1和v2执行按位AND运算。
- compl(val) : 对val执行补运算。
- lshift(val, count) : 将val左移count位。
- or(v1, v2) : 对v1和v2执行按位OR运算。
- rshift(val, count) : 将val右移count位。
- xor(v1, v2) : 对v1和v2执行按位XOR运算。

2. 字符串函数

不多bb了，如下表所示：

函数	描述
asort(s [,d])	将数组 *s* 按照元素值排序。索引会被替换成表示新顺序的连续数字。如果指定了 *d*，则排序后的数组会被保存在数组 *d* 中
asorti(s [,d])	将数组 *s* 按索引排序。生成的数组会将索引作为数组元素值，用连续数字索引表明排序顺序。如果指定了 *d*，则排序后的数组会被保存在数组 *d* 中
gensub(r,s, h[,t])	针对变量$0或目标字符串 *t* (如果提供了的话)来匹配正则表达式 *r*。如果 *h* 是一个以 *g* 或 *G* 开头的字符串，就用 *s* 替换匹配的文本。如果 *h* 是一个数字，则表示要替换 *r* 的第 *h* 处匹配
gsub(r,s [,t])	针对变量$0或目标字符串 *t* (如果提供了的话)来匹配正则表达式 *r*。如果找到了，就将所有的匹配之处全部替换成字符串 *s*
index(s,t)	返回字符串 *t* 在字符串 *s* 中的索引位置；如果没找到，就返回0
length(s)	返回字符串 *s* 的长度；如果没有指定，则返回$0的长度
match(s,r [,a])	返回正则表达式 *r* 在字符串 *s* 中匹配位置的索引。如果指定了数组 *a*，则将 *s* 的匹配部分保存到该数组中
split(s, a [,r])	将 *s* 以FS(字段分隔符)或正则表达式 *r* (如果指定了的话)分割并放入数组 *a* 中。返回分割后的字段总数
sprintf(*format*, *variables*)	用提供的 *format* 和 *variables* 返回一个类似于 `printf` 输出的字符串
sub(r,s,[,t])	在变量$0或字符串 *t* 中查找匹配正则表达式 *r* 的部分。如果找到了，就用字符串 *s* 替换第一处匹配
substr(s,i [,n])	返回 *s* 中从索引 *i* 开始、长度为 *n* 的子串。如果未提供 *n*，则返回 *s* 中剩下的部分
tolower(s)	将 *s* 中所有字符都转换为小写
toupper(s)	将 *s* 中所有字符都转换为大写

有些字符串函数的作用显而易见：

```
lxc@Lxc:~/scripts$ gawk 'BEGIN{x = "testing"; print toupper(x); print
length(x)}'
TESTING
7
```

有些字符串函数的用法较为复杂。`asort` 和 `asorti` 是新加入的gawk函数，允许基于数据元素值(asort)或索引(asorti)对数组变量进行排序。

```
lxc@Lxc:~/scripts$ gawk 'BEGIN{
> var["a"] = 1
> var["g"] = 2
> var["m"] = 3
> var["u"] = 4
> asort(var, test)
> for(i in test)
```

```
> {
> print "Index:",i, " - Value:",test[i]
> }
> }'
Index: 1  - Value: 1
Index: 2  - Value: 2
Index: 3  - Value: 3
Index: 4  - Value: 4
```

新数组 *test* 包含经过排序的原数组的数据元素，但数组索引变成了表明正确顺序的数字值。

`split` 函数是将数据字段放入数组以供进一步处理的好方法：

```
$ gawk 'BEGIN{FS=","}{
split($0, var)
print var[1],var[5]
}' data1
data11 data15
data21 data25
data31 data35
```

新数组使用连续数字作为数组索引，从含有第一个数据字段的索引值1开始。

3. 时间函数

时间戳（timestamp）是自1970-01-01 00:00:00 UTC 到现在，以秒为单位的计数，通常称为纪元时（epoch time）。gawk编程语言也有一些处理时间的函数，如下表所示：

函数	描述
mktime(*datespec*)	将一个按YYYY MM DD HH MM SS (DST)格式指定的日期转换为时间戳
strftime(*format* [, *timestamp*])	将当前时间戳或 *timestamp*（如果提供了的话）转换为格式化日期（采用shell `date` 命令的格式）
systime()	返回当前时间的时间戳

时间函数多用于处理日志文件。日志文件中通常含有需要进行比较的日期。通过将日期的文本表示形式转换为纪元时，可以轻松地比较日期。

```
$ gawk 'BEGIN{
> data = systime()
> day = strftime("%A, %B, %d, %Y", date)
> print day
> }'
星期四, 一月, 01, 1970
```

这个例子用 `systime` 函数从系统获取当前的时间戳，然后用 `strftime` 函数将其转换成用户可读的格式，转换过程中用到了shell命令 `date` 的日期格式化字符。

7. 自定义函数

要定义自己的函数，必须使用关键字 `function`：

```
function name([variables])
{
    statements
}
```

函数名必须能够唯一标识函数。你可以在调用该函数的gawk脚本中向其传入一个或多个变量：

```
function printthird()
{
    print $3
}
```

这个函数会打印记录中的第三个字段。

函数还可以使用 `return` 语句返回一个值。

```
return value
```

返回的这个值既可以是变量，也可以是最终能计算出值的算式：

```
function myrand(limit)
{
    return int(limit * rand())
}
```

可以将函数返回值赋给gawk脚本中的变量：

x = myrand(100)

这个变量包含函数的返回值。

2. 使用自定义函数

在定义函数时，它必须出现在所有代码块之前（包括 `BEGIN` 代码块）。

```
lxc@Lxc:~/scripts/ch22$ gawk '
> function myprint()
> {
> printf "%-16s %s\n", $1, $4
> }
> BEGIN{FS="\n";RS=""}
> {
> myprint()
> }' data2
Ima Test         (312)555-1234
Frank Tester     (317)555-9876
Haley Example    (313)555-4938
```

3. 创建函数库

gawk提供了一种方式以将多个函数放入单个库文件中，这样就可以在所有的gawk脚本中使用了。

首先需要创建一个包含所有gawk函数的文件：

```
lxc@Lxc:~/scripts/ch22$ cat funclib1
function myprint()
{
  printf "%-16s - %s\n", $1, $4
}
function myrand(limit)
{
  return int(limit * rand())
}
function printthird()
{
  print $3
}
```

`-f` 选项不能和内联gawk脚本(inline gawk script)一起使用，不过可以在同一命令行中使用多个 `-f` 选项。因此，要使用库，只要创建好gawk脚本文件，然后在命令行中同时指定库文件和脚本文件即可：

```
lxc@Lxc:~/scripts/ch22$ cat script4
BEGIN{ FS="\n"; RS=""}
{
    myprint()
}
lxc@Lxc:~/scripts/ch22$ gawk -f funclib1 -f script4 data2
Ima Test         - (312)555-1234
Frank Tester     - (317)555-9876
Haley Example    - (313)555-4938
```

8. 实战演练

bowling.sh

```
lxc@Lxc:~/scripts/ch22$ cat scores.txt
Rich Blum,team1,100,115,95
Barbara Blum,team1,110,115,100
Christine Bresnahan,team2,120,115,118
Tim Bresnahan,team2,125,112,116
# output:
lxc@Lxc:~/scripts/ch22$ ./bowling.sh
Total for team1 is 635 ,the average is 105.833
Total for team2 is 706 ,the average is 117.667
```

该脚本计算出每队的总分和平均分。

```
#!/bin/bash

for team in $(gawk -F, '{print $2}' scores.txt | uniq)
do
```

```
    gawk -v team=$team 'BEGIN{FS=","; total=0}
    {
        if ($2 == team)
        {
            total += $3 + $4 + $5
        }
    }
    END {
        avg = total / 6
        print "Total for", team, "is", total, ",the average is", avg
    }' scores.txt
done
```

ch23 使用其他shell

本章讲讲 dash 和 zsh。

1. 什么是 dash shell

(我真的不想解释，看书去吧。。。算了还是说说吧)
Debian Linux发行版与其许多衍生产品（比如Ubuntu）一样，使用dash shell作为Linux bash shell的替代品。dash shell是ash shell的直系后裔，是Unix系统中Bourne shell的简易复制品。
Henneth Almquist 这个人为Unix系统开发了Bourne shell的简化版本 Almquist shell，缩写为ash。ash最初的版本体积小，速度奇快，但缺乏许多高级特性，比如命令行编辑和命令历史，这使其很难用作交互式shell。
NetBSD Unix操作系统移植了ash shell，直到今天依然将其作为默认shell。NetBSD开发人员对其进行了定制，增加了一些新特性，使它更接近Bourne shell。ash shell的这个版本也被FreeBSD操作系统用作默认登录shell。
Debian Linux发行版创建了自己的ash shell版本，就叫 Debian shell或 dash。dash复刻了NetBSD版本的ash shell的大多数特性，提供了一些高级命令行编辑功能。你可以认为 dash 是Bourne shell的精简版，因此并不像bash shell那样支持众多特性。
在许多基于Debian的Linux发行版中，dash shell实际上并不是默认shell。由于bash shell在Linux世界广为流行，因此大多数基于Debian的Linux发行版选择将bash shell作为登录shell，而只将dash shell作为安装脚本的快速启动shell，用作发行版安装。
要想知道你的系统属于哪种情况，只需查看 */etc/passwd* 文件中的用户账户信息即可。你可以看看自己的账户的默认交互式shell。比如：

```
lxc@Lxc:~/scripts/ch23$ cat /etc/passwd | grep lxc
lxc:x:1000:1000:Lxc,,,:/home/lxc:/bin/bash
```

Ubuntu系统使用bash shell作为默认的交互式shell。要想知道默认的系统shell是什么，只需使用 `ls` 命令查看 */bin* 目录中的sh文件即可：

```
lxc@Lxc:~/scripts/ch23$ ls -al /bin/sh
lrwxrwxrwx 1 root root 13 11月  9 21:35 /bin/sh -> /usr/bin/bash
# 如你所见，笔者的Ubuntu系统的默认系统shell是bash。之前已有讲述，我更改了这个软连接。
```

Ubuntu系统使用dash shell作为默认的系统shell。这正是许多问题（这个问题在 之前 讲到过）的根源所在。

2. dash shell的特性

尽管bash shell和dash shell均以Bourne shell为样板，但两者之间还是有些差距的。

1. dash命令行选项

dash shell使用命令行选项控制其行为。下表列出了这些选项及其用途。

选项	描述
...	RTFM
-a	导出分配给shell的所有变量
-c	从特定的命令字符串中读取命令
-e	如果是非交互式shell，就在未经测试的命令失败时立即退出
-f	显示路径通配符
-n	如果是非交互式shell，就读取命令但不执行
-u	在尝试扩展一个未设置过的变量时，将错误消息写入 *STDERR*
-v	在读取输入时将输入写到 *STDERR*
-x	在执行命令时，将每个命令写入 *STDERR*
-I	在交互式模式下，忽略输入中的EOF字符
-i	强制shell运行在交互式模式下
-m	启用作业控制（在交互式模式下默认开启）
-s	从 *STDIN* 读取命令（在没有指定读取文件参数时的默认行为）
-E	启用Emacs命令行编辑器
-V	启用vi命令行编辑器

`-E` 命令行选项允许使用Emacs编辑器命令来进行命令行文本编辑（参见第10章）。你可以通过 Ctrl 和 Alt 组合键，使用所有的Emacs命令来处理一行中的文本。
`-V` 命令行选项允许使用vi编辑器命令进行命令行文本编辑（参见第10章）。该功能允许用Esc键在普通模式和vi编辑器模式之间切换。当处于vi模式时，可以使用标准的vi编辑器命令（比如，x 用于删除一个字符，i 用于插入文本）。当完成命令行编辑后，必须再次按下Esc键退出vi编辑器模式。

2. dash环境变量

dash使用大量的默认环境变量来记录信息，你也可以创建自己的环境变量。本节将介绍这些环境变量以及dash如何处理它们。

1. 默认环境变量

dash以简洁为目标，其使用的环境变量比bash shell明显要少。在dash shell环境编写脚本时，要考虑到这一点。

dash shell用 `set` 命令来显示环境变量：

```
$ set
CHROME_DESKTOP='code-url-handler.desktop'
......
```

2. 位置变量

除了默认的环境变量，dash shell还为在命令行中定义的参数分配了特殊的变量。下面是dash shell中可用的位置变量：

- $0 :shell脚本的名称。
- $n :第n个位置变量。
- $* :含有所有命令行参数的单个值，参数之间由 IFS 环境变量中的第一个字符分隔；如果没有定义 IFS，则由空格分隔。
- $@ :扩展为由所有的命令行参数组成的多个参数。
- $# :命令行参数的总数。
- $? :最近一个命令的退出状态码。
- $- :当前选项标志。
- $$:当前shell的进程ID（PID）。
- $! :最后一个后台命令的进程ID（PID）。

dash shell的位置变量的用法和bash shell一样，你可以像在bash shell中那样在dash shell脚本中使用位置变量。

3. 用户自定义环境变量

dash shell还允许你定义自己的环境变量。跟bash一样，可以在命令行中用赋值语句来定义新的环境变量：

```
$ testing=10; export testing
$ echo $testing
10
```

如果不用 `export` 命令，那么用户自定义环境变量就只在当前shell或进程中可见。

> dash变量和bash变量之间有一个巨大的差异：dash shell不支持数组。这个特性给高级shell脚本开发人员带来了各种问题。

3. dash内建命令

见书上 p538页

3. dash脚本编程

dash shell不能完全支持bash shell的脚本编程功能。为bash shell编写的脚本通常无法在dash shell中运行，这给shell脚本程序员带来了各种麻烦。本节将介绍一些值得留意的差别，以便你的shell脚本能够在dash shell环境中正常运行。

1. 创建dash脚本

可以在shell脚本的第一行指定以下内容：

```
#!/bin/dash
```

还可以在这行指定shell命令行选项，参见23.2.1节

2. 不能使用的特性

由于dash shell只是Bourne shell的一个子集，因此bash shell脚本中有些特性无法在dash shell中使用。这些特性通常称作 **bash流派(bashism)**。本节将简单总结那些在bash shell脚本中习惯使用，但在dash shell环境中无法使用的bash shell特性。

1. 算术运算

第11章 介绍过3种在bash shell脚本中执行数学运算的方法。

- 使用 `expr` 命令： expr operation。
- 使用方括号： $[operation]
- 使用双圆括号： $((operation))

dash shell支持使用 `expr` 命令和双圆括号进行数学运算，但不支持使用方括号。如果你的脚本中有大量采用方括号的数学运算，那么这可是个问题。

在dash shell脚本中执行数学运算的正确格式是使用双圆括号：

test1.sh

```
lxc@Lxc:~/scripts/ch23$ cat test1.sh
#!/bin/dash
# testing mathematical operations

value1=10
value2=15

value3=$(( $value1 * $value2 ))
echo "The answer is $value3"
# output:
lxc@Lxc:~/scripts/ch23$ ./test1.sh
The answer is 150
```

2. `test` 命令

虽然dash shell支持 `test` 命令，但务必注意其用法。dash shell版本的 `test` 命令跟bash shell版本略有不同。
bash shell的 `test` 命令使用双等号(==)来测试两个字符串是否相等，而dash shell中的 `test` 命令不能识别用作文本比较的 \== 符号，只能识别 = 符号。如果在bash脚本中使用了 \== 符号，则需要将其换成 \= 符号：

test2.sh

```
lxc@Lxc:~/scripts/ch23$ cat test2.sh
#!/bin/dash
```

```
# testing the = comparison

test1=abcde
test2=abcde

if [ $test1 = $test2 ]
then
    echo "They're the same"
else
    echo "They're the different"
fi
lxc@Lxc:~/scripts/ch23$ ./test2.sh
They're the same
```

3. `function` 命令

第17章 演示过如何在脚本中定义函数。bash shell支持两种定义函数的方法。

- 使用 `function` 语句。
- 只使用函数名。

dash shell不支持 `function` 语句，必须使用函数名和圆括号来定义函数。
如果编写的脚本可能会在dash环境中，就只能使用函数名来定义函数，决不要用 `function` 语句。

test3.sh

```
lxc@Lxc:~/scripts/ch23$ cat test3.sh
#!/bin/dash
# testing functions

func1() {
    echo "This is an example of a function"
}

count=1

while [ $count -le 5 ]
do
    func1
    count=$(( $count + 1 ))

done
echo "This is the end of the loop."
func1
echo "This is the end of script."
# output:
lxc@Lxc:~/scripts/ch23$ ./test3.sh
This is an example of a function
This is an example of a function
This is an example of a function
This is an example of a function
This is an example of a function
This is the end of the loop.
This is an example of a function
This is the end of script.
```

4. zsh shell

另一个流行的shell是 Z shell（称作zsh）。zsh是由 Paul Flastad开发的开源Unix shell。它汲取了所有现存shell的设计理念，增加了许多独有的特性，是为程序员而设计的一款无所不能的高级shell。
下面是zsh shell的一些独有特性：

- 改进的shell选项处理
- shell兼容模式
- 可加载模块

可加载模块是shell设计中最先进的特性。我们在bash 和 dash shell中已经看到过，每种shell都包含一组内建命令，这些命令无须借助外部程序即可使用。内建命令的好处在于执行速度快。shell不必在运行命令前先加载一个外部程序，因为内建命令已经在内存中了，随时可用。
zsh shell提供了一组核心内建命令，比如网络支持和高级数学功能。可以之添加你认为有用的模块。

5. zsh shell的组成

本节将带你逐步学习zsh shell的基础知识，介绍可用的内建命令（或是可以通过命令模块添加的命令）以及命令行选项和环境变量。

1. shell选项

大多数shell采用命令行选项来定义shell的行为。zsh shell也不例外，同样提供了相应的选项。你可以在命令行或shell中用 `set` 命令设置shell选项。
下表列出了zsh shell可用的命令行选项。

选项	描述
-c	只执行指定的命令，然后退出
-i	作为交互式shell启动，提供命令行提示符
-s	强制shell从 *STDIN* 读取命令
-o	指定命令行选项

`-o` 选项允许设置shell选项来定义shell的各种特性。到目前为止，zsh shell是所有shell中可定制性最强的。有大量的特性可供更改shell环境。shell选项可以分为以下几类。

2. 内建命令

本节将介绍核心内建命令以及在写作本书时可用的各种模块。

1. 核心内建命令

下表列出了可用的核心内建命令。

2. 附加模块

3. 查看、添加和删除模块

`zmodload` 命令是zsh模块的管理接口。你可以在zsh shell会话中使用该命令查看、添加或删除模块。不加任何参数的 `zmodload` 命令会显示zsh shell中当前已安装的模块。

```
lxc@Lxc ~/scripts/ch23 % zmodload
zsh/complete
zsh/complist
zsh/computil
zsh/main
zsh/parameter
zsh/stat
zsh/terminfo
zsh/zle
zsh/zutil
```

不同的zsh shell实现在默认情况下包含了不同的安装模块，要添加新模块，只需在 `zmodload` 命令行中指定模块名即可：

```
lxc@Lxc ~/scripts/ch23 % zmodload zsh/net/tcp
lxc@Lxc ~/scripts/ch23 %
```

无显示信息则表明模块已加载成功。再运行一次 `zmodload` 名录令，可以看到新模块出现在已安装模块的列表中。一旦加载了模块，该模块中的命令就成了可用的内建命令。

> **提示：** 将 `zmodload` 命令放入 $HOME/.zshrc 启动文件是一种常见的做法，这样在zsh启动时就会自动加载常用的模块。

6. zsh脚本编程

zsh shell的主要目的是为程序员提供一个高级编程环境。

1. 数学运算

zsh shell可以让你轻松地执行数学函数。目前，zsh shell在所有数学运算中都提供了对浮点数的全面支持。

1. 执行计算

zsh shell提供了执行数学运算的两种方法。

1. `let` 命令
2. 双圆括号

在使用 `let` 命令时，应该在算式前后加上双引号，这样才能使用空格：

```
lxc@Lxc ~/scripts/ch23 % let value1="4 * 5.1 / 3.2 "
lxc@Lxc ~/scripts/ch23 % echo $value1
6.3750000000
```

注意，使用浮点数会有精度问题。要解决这个问题，可以使用 `printf` 命令指定所需的小数点精度。

```
lxc@Lxc ~/scripts/ch23 % printf "%6.3f\n" $value1
6.375
```

第二种方法是使用双圆括号。这种方法结合了两种定义数学运算的方法：

```
lxc@Lxc ~/scripts/ch23 % value1=$((4 * 5.1))
lxc@Lxc ~/scripts/ch23 % (( value2 = 4 * 5.1 ))
lxc@Lxc ~/scripts/ch23 % printf "%6.3f\n" $value1 $value2
20.400
20.400
```

注意，可以将双圆括号放在算式两边（前面加个美元符号）或整个赋值表达式两边。如果将双圆括号放在整个赋值表达式两边的话，那赋值表达式中等号两侧可以有空格（见上面的例子）。两种方式能输出同样的结果。

如果一开始未用 `typeset` 命令声明变量的数据类型，那么zsh shell会尝试自动分配数据类型。这在处理整数和浮点数时很容易出问题。

```
lxc@Lxc ~/scripts/ch23 % value1=10
lxc@Lxc ~/scripts/ch23 % value2=$(( $value1 / 3 ))
lxc@Lxc ~/scripts/ch23 % echo $value2
3.0000000000
```

如果在指定数值时未指定小数部分的话，zsh shell会将其视为整数值并进行整数运算。如果想保证结果是浮点数，则必须指定小数部分：

```
lxc@Lxc ~/scripts/ch23 % value1=10.
lxc@Lxc ~/scripts/ch23 % value2=$(( $value1 / 3. ))
lxc@Lxc ~/scripts/ch23 % echo $value2
3.3333333333
```

2. 数学函数

zsh shell的数学函数可多可少。默认的zsh shell不含任何特殊的数学函数，但如果安装了zsh/mathfunc模块，那么你将拥有的数学函数绝对比需要的多：

```
lxc@Lxc ~/scripts/ch23 % value1=$(( sqrt(9) ))
zsh: unknown function: sqrt
lxc@Lxc ~/scripts/ch23 % zmodload zsh/mathfunc
lxc@Lxc ~/scripts/ch23 % value1=$(( sqrt(9) ))
lxc@Lxc ~/scripts/ch23 % echo $value1
3.
```

> **提示：** zsh支持大量的数学函数。要查看 zsh/mathfunc 模块提供的所有数学函数的清单，可参见zshmodules的手册页。

2. 结构化命令

zsh shell为shell脚本提供了常用的结构化命令。

- `if-then-else` 语句
- `for` 循环（包括C语言风格的）
- `while` 循环
- `until` 循环
- `select` 语句
- `case` 语句

zsh shell中的结构化命令采用的语法和你熟悉的bash shell一样。zsh shell还提供了另一个结构化命令 `repeat`。该命令格式如下：

```
repeat param
do
    commands
done
```

`param` 参数必须是一个数值，或是能计算出一个值的数学运算。然后，`repeat` 命令就会执行那么多次指定的命令：

```
lxc@Lxc ~/scripts/ch23 % cat test4.sh
#!/bin/zsh
# using the repeat command

value1=$(( 10 / 2 ))
repeat $value1
do
    echo "This is a test"
done
# output:
lxc@Lxc ~/scripts/ch23 % ./test4.sh
This is a test
This is a test
This is a test
This is a test
This is a test
```

3. 函数

zsh shell支持使用 `function` 命令或函数名加圆括号的形式来创建自定义函数：

```
lxc@Lxc ~/scripts/ch23 % function functest1 {
lxc@Lxc ~/scripts/ch23 function> echo "This is test1 function"
lxc@Lxc ~/scripts/ch23 function> }
lxc@Lxc ~/scripts/ch23 % function functest2 {
lxc@Lxc ~/scripts/ch23 function> echo "This is the test2 function"
lxc@Lxc ~/scripts/ch23 function> }
lxc@Lxc ~/scripts/ch23 % functest1
This is test1 function
lxc@Lxc ~/scripts/ch23 % functest2
This is the test2 function
```

跟bash shell函数一样（参见第17章），你可以在shell脚本中定义函数，然后使用全局变量或向函数传递参数。

7. 实战演练

zsh shell的tcp模块尤为实用。该模块允许创建TCP套接字，侦听传入的连接，然后与远程系统建立连接。这是在shell脚本之间传输数据的绝佳方式。

首先，打开shell窗口作为服务器。启动zsh，加载tcp模块，然后定义TCP套接字的侦听端口号。

```
lxc@Lxc:~/scripts$ zsh
lxc@Lxc ~/scripts % zmodload zsh/net/tcp
lxc@Lxc ~/scripts % ztcp -l 8888
lxc@Lxc ~/scripts % listen=$REPLY
lxc@Lxc ~/scripts % ztcp -a $listen
```

`ztcp` 命令的 `-l` 选项指定了监听的TCP端口号（在本例中是8888）。特殊的 `$REPLY` 变量包含与网络套接字关联的文件句柄(file handle)。`ztcp` 命令的 `-a` 选项会一直等待传入连接建立完毕。
现在，打开另一个shell窗口作为客户端，输入下列命令来连接服务端shell（注意客户端shell进入了ch23目录，服务器端shell并没有进入ch23目录，以此作为区分）：

```
lxc@Lxc:~/scripts/ch23$ zsh
lxc@Lxc ~/scripts/ch23 % zmodload zsh/net/tcp
lxc@Lxc ~/scripts/ch23 % ztcp localhost 8888
lxc@Lxc ~/scripts/ch23 % remote=$REPLY
lxc@Lxc ~/scripts/ch23 %
```

当建立好连接之后，你会在服务器端的shell窗口看到zsh shell命令行提示符。你可以在服务器端将新连接的句柄保存在变量中：

```
lxc@Lxc ~/scripts % remote=$REPLY
```

这样就可以收发数据了。要想发送消息，可以使用 `print` 语句将文本发送到 `$remote` 连接句柄：

```
lxc@Lxc ~/scripts/ch23 % print "This is a test message" >&$remote
```

在另一个shell窗口中，可以使用 `read` 命令接受发送到 `$remote` 连接的句柄，然后使用 `print` 命令将其显示出来：

```
lxc@Lxc ~/scripts % read -r data <&$remote; print -r $data
This is a test message
```

你也可以用同样的方法反向发送数据。

事成之后，使用 `-c` 选项关闭各个系统中对应的句柄。对于服务器端来说，可以使用下列命令：

```
lxc@Lxc ~/scripts % ztcp -c $listen
lxc@Lxc ~/scripts % ztcp -c $remote
```

对于客户端来说，可以使用如下命令：

```
lxc@Lxc ~/scripts/ch23 % ztcp -c $remote
```

你的shell脚本如今已经具备了联网特性，又上了一个新台阶。

ch24 编写简单的脚本实用工具

主要解释一下本章四个脚本中用到的一些命令。这几个脚本用到的知识在前文都已经讲过了，所以看懂的话肯定是没难度的。

1. 创建按日归档的脚本

Daily_archive

因为tar归档文件会占用大量的磁盘空间，因此最好压缩一下。这只需加一个 `-z` 选项即可。该选项会使用gzip压缩tar归档文件，由此生成的文件称作 tarball。后缀一般是 `.tar.gz` 或 `.tgz`。

```
tar -czf archive.tgz Project/*.* 2>/dev/null
```

在

```
if [ -f $filename -o -d $filename]
```

中，`-o` 选项使得只要其中一个测试为真，那么整个 `if` 语句就成立。其他选项见 第12章。

其他可能有助于备份的工具，可以查看 `man -k archive` 和 `man -k copy`。

2. 创建按小时归档的脚本

Hourly_archive.sh

如果希望文件名总是保留4位小数，则可以将脚本中的 `TIME=$(date +%k%M)` 修改为 `TIME=$(date +%k0%M)`。在 `%k` 后面加入数字0后，所有的单位(single-digit)小时数都会加上一个前导数字0，被填充成两位数字。

3. 删除账户

Delete_User.sh

`cut` 命令的用法。`cut` 命令在手册中说明的很清楚了，看手册吧。

Print selected parts of lines from each FILE to standard output.
With no FILE, or when FILE is -, read standard input.

`cut` 支持三种选中方式，`-b` 选项以字节为单位进行选中，`-c` 选项以字符为单位选中，`-f` 以字段为单位选中，`cut` 命令默认的字段分隔符为TAB。

Use one, and only one of -b, -c or -f. Each LIST is made up of one range, or many ranges separated by commas.
Selected input is written in the same order that it is read, and is written exactly once. Each range is oneof:

- N   N'th byte, character or field, counted from 1
- N-   from N'th byte, character or field, to end of line
- N-M   from N'th to M'th (included) byte, character or field
- -M   from first to M'th (included) byte, character or field

来看几个例子：

```
lxc@Lxc:~/tt$ cat text1.txt
echo "Hello World!"
哈哈
lxc@Lxc:~/tt$ cut -b 1 text1.txt
e
�
lxc@Lxc:~/tt$ cut -b 1-3 text1.txt
ech
哈
lxc@Lxc:~/tt$ cut -b 1- text1.txt
echo "Hello World!"
哈哈
lxc@Lxc:~/tt$ cut -b -3 text1.txt
ech
哈
lxc@Lxc:~/tt$ cut -b 1,3,4 text1.txt
eho
��
xc@Lxc:~/tt$ cut -d " " -f1 text1.txt
echo
哈哈
lxc@Lxc:~/tt$ cut -d " " -f2 text1.txt
"Hello
哈哈
lxc@Lxc:~/tt$ cut -d " " -f1 -s text1.txt
echo
lxc@Lxc:~/tt$ cut -d " " -f 1- -s --output-delimiter "      " text1.txt
echo      "Hello      World!"
lxc@Lxc:~/tt$ cut -d "  " -f 1- -s --output-delimiter "      " text1.txt
cut: 分界符必须是单个字符
请尝试执行 "cut --help" 来获取更多信息。
lxc@Lxc:~/tt$ cat /etc/passwd | grep lxc | cut -d: -f7
/bin/bash
```

`cut` 的 `-s` 选项： do not print lines not containing delimiters

在这个脚本中，

```
lxc@Lxc:~/scripts$ echo "Yes" | cut -c1
Y
```

`cut` 命令的 `-c1` 选项可以删除除第一个字符之外的所有内容，即选中第一个字符。

```
cut -d: -f7
```

`cut` 命令的 `-d` 选项指定字段分隔符为`:`，`-f7` 选项指定记录中的第七个字段。

`xargs` 命令的用法。

`xargs` 命令来自英文词组“extended arguments”的缩写，其功能是用于给其他命令传递参数的过滤器。xargs命令能够处理从标准输入或管道符输入的数据，并将其转换成命令参数，也可以将单行或多行输入的文本转换成其他格式。

```
lxc@Lxc:~/tt$ echo "Hello World" | echo

lxc@Lxc:~/tt$ echo "Hello World" | xargs echo
Hello World
lxc@Lxc:~/tt$ echo "Hello World" | xargs -n 1
Hello
World
lxc@Lxc:~/tt$ echo "Hello#World" | xargs -d "#"
Hello World

lxc@Lxc:~/tt$ echo -n "Hello#World" | xargs -d "#"
Hello World
lxc@Lxc:~/tt$ echo -n "Hello#World" | xargs -d "#" -p
echo Hello World ?...y
Hello World
lxc@Lxc:~/tt$ echo -n "Hello#World" | xargs -d "#" -n 1 -p
echo Hello ?...y
Hello
echo World ?...y
World
lxc@Lxc:~/tt$ echo -n "Hello#World" | xargs -d "#" -n 1 -t
echo Hello
Hello
echo World
World
lxc@Lxc:~/tt$ echo -n "hello#world" | xargs -I {} echo {}
hello#world
lxc@Lxc:~/tt$ echo | xargs echo

lxc@Lxc:~/tt$ echo | xargs -r echo
lxc@Lxc:~/tt$ ll
总用量 12
drwxrwxr-x  2 lxc lxc 4096 11月 26 12:56 ./
drwxr-xr-x 55 lxc lxc 4096 11月 26 13:42 ../
-rw-rw-r--  1 lxc lxc   27 11月 26 12:56 text1.txt
-rw-rw-r--  1 lxc lxc    0 11月 26 12:55 text2.txt
```

```
-rw-rw-r--  1 lxc lxc    0 11月 26 12:55 text3.txt
lxc@Lxc:~/tt$ vim text1.txt
lxc@Lxc:~/tt$ cat text1.txt
lalala
hahaha
heihei haha
\a \n \t
nnn

lxc@Lxc:~/tt$ cat text1.txt | xargs echo
lalala hahaha heihei haha a n t nnn
lxc@Lxc:~/tt$ vim text1.txt
lxc@Lxc:~/tt$ cat text1.txt
lalala
hahaha
heihei haha
'\'a "\"n \\t
nnn

lxc@Lxc:~/tt$ cat text1.txt | xargs echo
lalala hahaha heihei haha \a \n \t nnn
lxc@Lxc:~/tt$ cat text1.txt | xargs -n 2 echo
lalala hahaha
heihei haha
\a \n
\t nnn
lxc@Lxc:~/tt$ cat text1.txt | xargs touch
lxc@Lxc:~/tt$ ls
'\a'   haha   hahaha   heihei   lalala  '\n'   nnn  '\t'   text1.txt
text2.txt   text3.txt
lxc@Lxc:~/tt$ cat text1.txt | xargs rm
lxc@Lxc:~/tt$ ls
text1.txt  text2.txt  text3.txt
```

`xargs` 可以使用从标准输入 *STDIN* 获取的命令行参数并执行指定的命令。它非常适合放在管道的末尾处。

在该脚本中：有下面这一段代码：

```
command_1="ps -u $user_account --no-heading"
#
# Create command_3 to kill processes in variable
command_3="xargs -d \\n /usr/bin/sudo /bin/kill -9"
#
# Kill processes via piping commands together
$command_1 | gawk '{print $1}' | $command_3
```

`xargs` 命令被保存在变量 `command_3` 中。选项 `-d` 指定使用什么样的分隔符。也就是说，`xargs` 从 *STDIN* 处接收多个项作为输入，那么各个项之间要怎么区分呢，在这里，`\n`（换行符）被用作各项的分隔符。当 `ps` 命令的输出PID列被传给 `xargs` 时，后者会将每个PID作为单个项来处理。因为 `xargs` 命令被赋给了变量，所以 \n 中的反斜线必须再加上另一个反斜线进行转义。
注意，在处理PID时，`xargs` 需要使用命令的完整路径名。`xargs` 的现代版本 **不要求** 使用命令的绝对路径。但较旧的Linux发行版可能使用的还是旧版本的 `xargs`，因此我们依然采用了绝对路径写法。

4. 系统监控

Audit_System.sh

系统账户（第7章）用于提供服务或执行特殊任务。一般来说，这类账户需要在 */etc/passwd* 文件中有对应的记录，但禁止登录系统（root账户是一个典型的例外）。
防止有人使用这些账户登录的方法是，将其默认shell设置为 */bin/false* 、*/usr/sbin/nologin* 或 */sbin/nologin*。当系统账户的默认shell从当前设置更改为 */bin/bash* 时，就会出现问题。虽然 **不良行为者** 在没有设置密码的情况下无法登录到该账户，但这仍会削弱系统的安全性。因此，账户设置需要进行审计，以纠正不正确的默认shell。
审计这种潜在问题的一种方法是确定有多少账户的默认shell被设置为false或nologin，然后定期检查这一数量。如果发现数量减少，则有必要进一步调查。

在下面这段代码中，

```
cat /etc/passwd | cut -d: -f7 |
grep -E "(nologin|false)" | wc -l |
tee $accountReport
```

`grep` 命令的 `-E` 选线使grep支持ERE（扩展正则表达式），在扩展正则表达式中，选择结构应该放入圆括号并用竖线分隔开。要想让 `grep` 正常工作，还有必不可少的事要做：**shell引用（shell quoting）**。因为圆括号和竖线在bash shell中具有特殊含义，所以必须将其放入引号中，避免shell误解。
`wc` 命令的 `-l` 选项统计 `grep` 命令生成的结果有多少行。`tee` 命令将结果写入报告文件的同时，还将这一信息显示给脚本用户。

```
sudo chattr +i  $accountReport
```

为了保护报告，需要设置 **不可变属性（immutable attribute）**。只要对文件设置了该属性，任何人（包括超级用户）都无法修改或删除此文件（以及一些其它特性）。要设置不可变属性，需要使用 `chattr` 命令，并且具有超级用户权限。
要想查看属性是否设置成功，可以使用 `lsattr` 命令，并在输出中查找 `i`。要想移除不可变属性，需要再次使用 `chattr` 命令（具有超级用户权限），之后文件就可以被修改或删除了。
可以查看 `chattr` 命令的手册页，看看还有什么其他的可设置属性。

```
prevReport="$(ls -1t $reportDir/PermissionAudit*.rpt | sed -n '2p')"
```

`ls` 命令 `-t` 选项使输出按照修改时间从新到旧的顺序排序，`-1`（里面是数字1，不是字母l）选项使的输出按照每行一列的格式输出。

```
sudo find / -perm /6000 >$permReport 2>/dev/null
```

如上，`find` 命令的搜索起点是根目录，`find` 命令的 `-perm` 选项可以使用八进制值指定要查找的具体权限。因为要检查系统中所有的文件和目录，所以需要超级用户权限。注意 `-perm` 的值是 `/6000`.八进制6表示 `find` 要查找的权限是 `SUID` 和 `SGID`。正斜线以及八进制值 `000` 告诉 `find` 命令忽略文件或目录的其余权限。如果没有正斜线，那么 `find` 命令会查找设置了 SUID 权限和 SGID 权限且其他权限均未设置（000）的文件或目录，这当然不是我们想要的。

> 旧版本的 `find` 命令使用加号 `+` 表示忽略某些权限。如果你使用的Linux版本比较旧，则可能需要把正斜线换成加号。

```
differences=$(diff $permReport $prevReport)
```

`diff` 命令可以比较文件，并将两者之间的差异输出到 *STDOUT*。`diff` 命令对文件进行逐行比较。因此，`diff` 会比较两分报告中的第一行，然后是第二行、第三行、以此类推。如果由于要安装软件，添加了一个或一批新文件，而这些文件需要SUID或SGID权限，那么在下一次审计时，`diff` 命令就会显示大量的差异。为了解决这个潜在的问题，可以在 `diff` 命令中使用 `-q` 选项或 `--brief` 选项，只显示消息，说明这两份报告存在不同。