

目 录

致谢

介绍

翻译术语表

第一部分 初见shell

1. 为什么使用shell编程
2. 和Sha-Bang(#!)一起出发
 - 2.1 调用一个脚本
 - 2.2 牛刀小试

第二部分 shell基础

3. 特殊字符
4. 变量与参数
 - 4.1 变量替换
 - 4.2 变量赋值
 - 4.3 Bash弱类型变量
 - 4.4 特殊变量类型
5. 引用
 - 5.1 引用变量
 - 5.2 转义
6. 退出与退出状态
7. 测试
 - 7.1 测试结构
 - 7.2 文件测试操作
 - 7.3 其他比较操作
 - 7.4 嵌套 if/then 条件测试
 - 7.5 牛刀小试
8. 运算符相关话题
 - 8.1 运算符
 - 8.2 数字常量
 - 8.3 双圆括号结构
 - 8.4 运算符优先级

第三部分 shell进阶

10. 变量处理

10.1 字符串处理	
10.1.1 使用 awk 处理字符串	
10.1.2 参考资料	
10.2 参数替换	
11. 循环与分支	
11.1 循环	
11.2 嵌套循环	
11.3 循环控制	
11.4 测试与分支	
12. 命令替换	
13. 算术扩展	
14. 休息时间	
第五部分 进阶话题	
18 正则表达式	
18.1 正则表达式简介	
19. 嵌入文档	
20. I/O 重定向	
20.1 使用 exec	
20.2 重定向代码块	
20.3 应用程序	
22. 限制模式的Shell	
23. 进程替换	
26. 列表结构	
24 函数	
24.1 复杂函数和函数复杂性	
24.2 局部变量	
24.3 不使用局部变量的递归	
25. 别名	
27 数组	
30 网络编程	
32 调试	
33 选项	
34 陷阱	
36 杂项	

36.1 交互和非交互shell以及脚本

36.2 shell wrappers

36.3 测试和比较的其他方法

36.4 递归：可以调用自己的脚本

38 后记

38.1 作者后记

38.2 关于作者

38.3 在哪里可以获得帮助

38.4 用来制作这本书的工具

38.5 致谢

38.6 免责声明

致谢

当前文档 《高级Bash脚本编程指南 (Advanced Bash-Scripting Guide)》 由 进击的皇虫 使用 书栈 (BookStack.CN) 进行构建，生成于 2018-04-26。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

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

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

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：<http://www.bookstack.cn/books/Advanced-Bash-Scripting-Guide-in-Chinese>

书栈官网：<http://www.bookstack.cn>

书栈开源：<https://github.com/TruthHun>

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

介绍

- [《Advanced Bash-Scripting Guide》 in Chinese](#)
 - [联系/加入我们](#)
 - [原著及早期翻译作品](#)
 - [原著](#)
 - [译著](#)
 - [翻译作品](#)
 - [翻译进度](#)
 - [翻译校审流程](#)
 - [初始化](#)
 - [翻译校审流程](#)
 - [翻译校审建议](#)
 - [关于版权](#)
 - [来源\(书栈小编注\)](#)

《Advanced Bash-Scripting Guide》 in Chinese

《高级Bash脚本编程指南》Revision 10中文版

联系/加入我们

- 邮箱: absguide@linuxstory.org(将#替换为@)
- QQ群: 535442421

原著及早期翻译作品

原著

- 原著链接: <http://tldp.org/LDP/abs/html/>
- 原作: Mendel Cooper
- 原著版本: Revision 10, 10 Mar 2014

译著

- 早期译著连接: <http://www.linuxsir.org/bbs/thread256887.html>
- 译者: 杨春敏 黄毅
- 译著版本: Revision 3.7, 23 Oct 2005
- 最新 Revision 10 由 Linux Story 社区的 imcmy 同学发起并组织翻译
- Linux Story 通告地址: <http://www.linuxstory.org/advanced-bash-scripting-guide-in-chinese/>

翻译作品

翻译作品放在[GitBook](#)上, 欢迎阅读!

翻译进度

- 第一部分 初见Shell[@imcmy][@zihengcat]
 - 1. 为什么使用shell编程[@imcmy][@zihengcat]
 - 2. Sha-Bang (#!) 一起出发[@imcmy][@zihengcat]
- 第二部分 Shell基础[@imcmy][@zihengcat]
 - 3. 特殊字符[@imcmy][@zihengcat]
 - 4. 变量与参数[@imcmy][@zihengcat]
 - 5. 引用[@mr253727942][@zihengcat]
 - 6. 退出与退出状态[@samita2030][@zihengcat]

- 7. 测试[@imcmy][@zihengcat]
- 8. 运算符相关话题[@samita2030][@zihengcat]
- 第三部分 Shell进阶[@imcmy]
 - 9. Another Look at Variables[@Ninestd]
 - 10. 变量处理[@imcmy]
 - 11. 循环与分支[@imcmy]
 - 12. 命令替换[@imcmy]
 - 13. 算术扩展[@imcmy]
 - 14. 休息时间[@imcmy]
- 第四部分. 命令[@zhaozq]
 - 15. 内建命令[@zhaozq]
 - 16. 外部过滤器, 程序与命令[@zhaozq]
 - 17. 系统与高级命令[@zhaozq]
- 第五章. Advanced Topics
 - 18. 正则表达式[@Zjie]
 - 18.1 正则表达式简介[@Zjie]
 - 18.2 文件名替换[@Zjie]
 - 19. 嵌入文档[@mingmings]
 - 20. I/O 重定向[@mingmings]
 - 21. Subshells[@mingmings]
 - 22. Restricted Shells[@panblack]
 - 23. Process Substitution[@panblack]
 - 24. Functions[@zy416548283]
 - 25. 别名[@mingmings]
 - 26. List Constructs[@panblack]
 - 27. Arrays[@zy416548283]
 - 28. Indirect References[@panblack]
 - 29. /dev and /proc[@panblack]

- 30. Network Programming[@Zjie]
- 31. Of Zeros and Nulls[@panblack]
- 32. Debugging[@wuqichao]
- 33. Options[@zy416548283]
- 34. Gotchas[@liuburn]
- 35. Scripting With Style[@chuchingkai]
- 36. Miscellany[@richard-ma]
- 37. Bash, versions 2, 3, and 4
- 38. Endnotes[@zy416548283]
 - 38.1 Author's Note
 - 38.2 About the Author
 - 38.3 Where to Go For Help
 - 38.4 Tools Used to Produce This Book
 - 38.5 Credits
 - 38.6 Disclaimer
- Bibliography
- Appendix
 - A. Contributed Scripts
 - B. Reference Cards
 - C. A Sed and Awk Micro-Primer[@wuqichao]
 - C.1 Sed[@wuqichao]
 - C.2 Awk[@wuqichao]
 - D. Parsing and Managing Pathnames
 - E. Exit Codes With Special Meanings
 - F. A Detailed Introduction to I/O and I/O Redirection
 - G. Command-Line Options
 - G.1 Standard Command-Line Options

- G.2 Bash Command-Line Options
- H. Important Files
- I. Important System Directories
- J. An Introduction to Programmable Completion
- K. Localization
- L. History Commands
- M. Sample .bashrc and .bash_profile Files
- N. Converting DOS Batch Files to Shell Scripts
- O. Exercises
 - 0.1 Analyzing Scripts
 - 0.2 Writing Scripts
- P. Revision History
- Q. Download and Mirror Sites
- R. To Do List
- S. Copyright
- T. ASCII Table
- Index
- List of Tables
- List of Examples

翻译校审流程

初始化

1. 首先fork项目
2. 把fork过去的项目clone到本地
3. 命令行下运行 `git checkout -b dev` 创建一个新分支
4. 运行 `git remote add upstream`

`https://github.com/LinuxStory/Advanced-Bash-Scripting-Guide-in-`

`Chinese.git` 添加远端库

5. 运行 `git remote update` 更新
6. 运行 `git fetch upstream master` 拉取更新到本地
7. 运行 `git rebase upstream/master` 将更新合并到你的分支

初始化只需要做一遍，之后请在dev分支进行修改。

如果修改过程中项目有更新，请重复5、6、7步。

翻译校审流程

1. 保证在dev分支中
2. 打开README.md，在翻译进度后加上你自己的github名

1. Shell Programming! [@翻译人][@校审人]
3. 本地提交修改，写明提交信息
4. push到你fork的项目中，然后登录GitHub
5. 在你fork的项目的首页可以看到一个 `pull request` 按钮，点击它，填写说明信息，然后提交即可

为了不重复工作，请等待我们确认了你的pull request(即你的名字出现在项目中时)，再进行翻译校审工作
6. 进行翻译校审，重复3-5步提交翻译校审的作品

新手可以参阅针对github小白的[《翻译流程详解》](#)，妹子写的哟~

翻译校审建议

1. 使用markdown进行翻译校审，文件名必须使用英文
2. 翻译校审后的文档请放到source文件夹下的对应章节中，然后pull request即可
3. 有任何问题随时欢迎发issue

4. 术语尽量保证和已翻译的一致，也可以查询[微软术语搜索](#)或[Linux 中国术语词典](#)
5. 你可以将你认为术语的词汇加入术语表 `TERM.md` 中

关于版权

根据原著作者的要求，翻译成果属于公有领域(CC0)，翻译参与人员及原著作者Mendel Cooper享有署名权

来源(书栈小编注)

<https://github.com/LinuxStory/Advanced-Bash-Scripting-Guide-in-Chinese>

翻译术语表

- [翻译术语表](#)

翻译术语表

TERM	术语
script	脚本
utility	实用程序
prototype	原型
builtin	内建命令
hard-coded	硬编码
magic number	幻数
generalize	泛化
POSIX	可移植操作系统接口
stdin	标准输入

第一部分 初见shell

- [第一部分 初见Shell](#)
 - [内容目录](#)

第一部分 初见Shell

脚本：文章；书面文档

——韦伯斯特字典1913年版

Shell是一种命令解释器，它不仅分离了用户层与操作系统内核，更是一门强大的编程语言。我们称为shell编写的程序为脚本（script）。脚本是一种易于使用的工具，它能够将系统调用、工具软件、实用程序（utility）和已编译的二进制文件联系在一起构建程序。实际上，shell脚本可以调用所有的UNIX命令、实用程序以及工具软件。如果你觉得这还不够，使用像 `test` 命令和循环结构这样的shell内建命令能够让脚本更加灵活强大。Shell脚本特别适合完成系统管理任务和那些不需要复杂结构性语言实现的重复工作。

内容目录

- [1. 为什么使用shell编程](#)
- [2. 和Sha-Bang（#!）一起出发](#)
 - [2.1 调用一个脚本](#)
 - [2.2 牛刀小试](#)

1. 为什么使用shell编程

- 第一章 为什么使用shell编程
 - 什么时候不应该使用shell脚本

第一章 为什么使用shell编程

没有任何一种程序设计语言是完美的，甚至没有一个最好的语言。只有在特定环境下适合的语言。

— Herbert Mayer

无论你是否打算真正编写shell脚本，只要你想要在一定程度上熟悉系统管理，了解掌握shell脚本的相关知识都是非常有必要的。例如Linux系统在启动的时候会执行 `/etc/rc.d` 目录下的shell脚本来恢复系统配置和准备服务。详细了解这些启动脚本对分析系统行为大有益处，何况，你很有可能会去修改它们呢。

编写shell脚本并不困难，shell脚本由许多小的部分组成，而其中只有数量相当少的与shell本身特性，操作和选项^{^1}有关的部分才需要去学习。Shell语法非常简单朴素，很像是在命令行中调用和连接工具，你只需遵循很少一部分的“规则”就可以了。大部分短小的脚本通常在第一次就可以正常工作，即使是一个稍长一些的脚本，调试起来也十分简单。

在个人计算机发展的早期，BASIC语言让计算机专业人士能够在早期的微机上编写程序。几十年后，Bash脚本可以让所有仅对Linux或UNIX系统有初步了解的用户在现代计算机上做同样的事。

我们现在已经可以做出一些又小又快的单板机，比如树莓派。Bash脚本提供了一种发掘这些有趣设备潜力的方式。

使用shell脚本构建一个复杂应用原型（prototype），不失为是一种虽有缺陷但非常快速的方式。在项目开发初期，使用脚本实现部分功

能往往显得十分有用。在使用C/C++，Java，Perl或Python编写最终代码前，可以使用shell脚本测试，修补应用结构，提前发现重大缺陷。

Shell脚本与经典的UNIX哲学相似，将复杂的任务划分为简单的子任务，将组件与工具连接起来。许多人认为比起新一代功能强大、高度集成的语言，例如Perl，shell脚本至少是一种在美学上更加令人愉悦的解决问题的方式，Perl试图做到面面俱到，但你必须强迫自己改变思维方式适应它。

Herbert Mayer曾说：“有用的语言需要数组、指针以及构建数据结构的通用机制”。如果依据这些标准，那shell脚本距“有用”还差得很远，甚至是“无用”的。

什么时候不应该使用shell脚本

- 资源密集型的任务，尤其是对速度有要求（如排序、散列、递归^{^2}等）
- 需要做大量的数学运算，例如浮点数运算，高精度运算或者复数运算（使用C++或FORTRAN代替）
- 有跨平台需求（使用C或者Java代替）
- 必须使用结构化编程的复杂应用（如变量类型检查、函数原型等）
- 影响系统全局的关键性任务
- 对安全性有高要求，需要保证系统的完整性以及阻止入侵、破解、恶意破坏
- 项目包含有连锁依赖关系的组件
- 需要大量的文件操作（Bash只能访问连续的文件，并且是以一种非常笨拙且低效的逐行访问的方式进行的）
- 需要使用多维数组
- 需要使用如链表、树等数据结构

- 需要产生或操作图像和图形用户接口（GUI）
- 需要直接访问系统硬件或外部设备
- 需要使用端口或套接字输入输出端口（Socket I/O）
- 需要使用库或旧程序的接口
- 私有或闭源的项目（Shell脚本直接将源代码公开，所有人都可以看到）

如果你的应用满足上述任意一条，你可以考虑使用更加强大的脚本语言，如Perl, Tcl, Python, Ruby等，或考虑使用编译型语言，如C, C++或Java等。即使如此，在开发阶段使用shell脚本建立应用原型也是十分有用的。

我们接下来将使用Bash。Bash是“Bourne-Again shell”的首字母缩略词³，Bash来源于Stephen Bourne开发的Bourne shell（sh）。如今Bash已成为了大部分UNIX衍生版中shell脚本事实上的标准。本书所涉及的大部分概念在其他shell中也是适用的，例如Korn Shell，Bash从它当中继承了一部分的特性⁴；又如C Shell及其变体（需要注意的是，1993年10月Tom Christiansen在[Usenet帖子](#)中指出，因C Shell内部固有的问题，不推荐使用C Shell编程）

接下来的部分将是一些编写shell脚本的指导。这些指导很大程度上依赖于实例来阐述shell的特性。本书所有的例子都能够正常工作，并在尽可能的范围内进行过测试，其中的一部分已经运用在实际生产生活中。读者们可以使用这些在存档中的例子（文件名为 `scriptname.sh` 或 `scriptname.bash`）⁵，赋予它们可执行权限（`chmod u+rx scriptname`），然后执行它们看看会发生什么。如果存档不可用，读者朋友也可以从本书的HTML或者PDF版本中复制粘贴代码出来。需要注意的是，在部分例子中使用了一些暂时还未被解释的特性，这需要读者暂时跳过它们。

除特别说明，本书所有例子均由[本书作者](#)编写。

His countenance was bold and bashed not.

— *Edmund Spenser*

[^5]：按照惯例，用户编写的Bourne shell脚本应该在文件名后加上 `.sh` 的扩展名。而那些系统脚本，比如在 `/etc/rc.d` 中的脚本通常不遵循这种规范。

2. 和Sha-Bang(#!)一起出发

- [第二章 和Sha-Bang \(#!\) 一起出发](#)
 - [本章目录](#)

第二章 和Sha-Bang (#!) 一起出发

Shell编程声名显赫

— *Larry Wall*

本章目录

- [2.1 调用一个脚本](#)
- [2.2 牛刀小试](#)

一个最简单的脚本其实就是将一连串系统命令存储在一个文件中。最起码，它能帮你省下重复输入这一连串命令的功夫。

样例 2-1. *cleanup*: 清理 `/var/log` 目录下的日志文件

```
1. # Cleanup
2. # 请使用root权限执行
3.
4. cd /var/log
5. cat /dev/null > messages
6. cat /dev/null > wtmp
7. echo "Log files cleaned up."
```

这支脚本仅仅是一些可以很容易从终端或控制台输入的命令的集合罢了，没什么特殊的地方。将命令放在脚本中的好处是，你不用再一遍遍重复输入这些命令啦。脚本成了一支程序、一款工具，它可以很容易的被修改或为特殊需求定制。

样例 2-2. *cleanup*: 改进的清理脚本

```

1.  #!/bin/bash
2.  # Bash脚本标准起始行。
3.
4.  # Cleanup, version 2
5.
6.  # 请使用root权限执行。
7.  # 这里可以插入代码来打印错误信息，并在未使用root权限时退出。
8.
9.  LOG_DIR=/var/log
10. # 使用变量比硬编码（hard-coded）更合适
11. cd $LOG_DIR
12.
13. cat /dev/null > messages
14. cat /dev/null > wtmp
15.
16.
17. echo "Logs cleaned up."
18.
19. exit # 正确终止脚本的方式。
20.      # 不带参数的exit返回上一条指令的运行结果。

```

现在看到了一个真正意义上的脚本！让我们继续前进...

样例 2-3. *cleanup*: 改良、通用版

```

1.  #!/bin/bash
2.  # Cleanup, version 3
3.
4.  # 注意：
5.  # -----
6.  # 此脚本涉及到许多后边才会解释的特性。
7.  # 当你阅读完整本书的一半以后，理解它们就没有任何困难了。
8.
9.
10. LOG_DIR=/var/log

```

```

11.  ROOT_UID=0      # UID为0的用户才拥有root权限。
12.  LINES=50        # 默认保存messages日志文件行数。
13.  E_XCD=86        # 无法切换工作目录的错误码。
14.  E_NOTROOT=87    # 非root权限用户执行的错误码。
15.
16.
17.
18.  # 请使用root权限运行。
19.  if [ "$UID" -ne "$ROOT_UID" ]
20.  then
21.      echo "Must be root to run this script."
22.      exit $E_NOTROOT
23.  fi
24.
25.  if [ -n "$1" ]
26.  # 测试命令行参数（保存行数）是否为空
27.  then
28.      lines=$1
29.  else
30.      lines=$LINES # 如果为空则使用默认设置
31.  fi
32.
33.
34.  # Stephane Chazelas 建议使用如下方法检查命令行参数,
35.  # 但是这已经超出了此阶段教程的范围。
36.  #
37.  #   E_WRONGARGS=85 # Non-numerical argument (bad argument
38.  #   case "$1" in
39.  #       ""          ) lines=50;;
40.  #       *[!0-9]*) echo "Usage: `basename $0` lines-to-cleanup";
41.  #       exit $E_WRONGARGS;;
42.  #       *           ) lines=$1;;
43.  #   esac
44.  #
45.  #* 在第十一章“循环与分支”中会对此作详细的阐述。
46.
47.

```

```

48. cd $LOG_DIR
49.
50. if [ `pwd` != "$LOG_DIR" ] # 也可以这样写 if [ "$PWD" != "$LOG_DIR"
    ]
51.                               # 检查工作目录是否为 /var/log ?
52. then
53.     echo "Can't change to $LOG_DIR"
54.     exit $E_XCD
55. fi # 在清理日志前，二次确认是否在正确的工作目录下。
56.
57. # 更高效的写法：
58. #
59. # cd /var/log || {
60. #     echo "Cannot change to necessary directory." >&2
61. #     exit $E_XCD;
62. # }
63.
64.
65. tail -n $lines messages > mesg.temp # 保存messages日志文件最后一部分
66. mv mesg.temp messages                # 替换系统日志文件以达到清理目的
67.
68. # cat /dev/null > messages
69. #* 我们不需要使用这个方法了，上面的方法更安全
70.
71. cat /dev/null > wtmp # ': > wtmp' 与 '> wtmp' 有同样的效果
72. echo "Log files cleaned up."
73. # 注意在/var/log目录下的其他日志文件不会被这个脚本清除
74.
75. exit 0
76. # 返回0表示脚本运行成功

```

也许你并不希望清空全部的系统日志，这个脚本保留了messages日志的最后一部分。随着学习的深入，你将明白更多提高脚本运行效率的方法。

脚本起始行`sha-bang (#!)` ^{^1}告诉系统这个脚本文件需要使用指定的

命令解释器来执行。#!实际上是一个占两字节[^2]的幻数 (magic number) , 幻数可以用来标识特殊的文件类型, 在这里则是标记可执行 shell 脚本 (你可以在终端中输入 `man magic` 了解更多信息)。紧随 #! 的是一个路径名。此路径指向用来解释此脚本的程序, 它可以是 shell, 可以是程序设计语言, 也可以是实用程序。这个解释器从头 (#! 的下一行) 开始执行整个脚本的命令, 同时忽略注释。[^3]

```
1. #!/bin/sh
2. #!/bin/bash
3. #!/usr/bin/perl
4. #!/usr/bin/tcl
5. #!/bin/sed -f
6. #!/bin/awk -f
```

上面每一条脚本起始行都调用了不同的解释器, 比如 `/bin/sh` 调用了系统默认 shell (Linux 系统中默认是 bash)[^4]。大部分 UNIX 商业发行版中默认的是 Bourne shell, 即 `#!/bin/sh`。你可以以牺牲 Bash 特性为代价, 在非 Linux 的机器上运行 sh 脚本。当然, 脚本得遵循 POSIX[^5] sh 标准。

需要注意的是 `#!` 后的路径必须正确, 否则当你运行脚本时只会得到一条错误信息, 通常是 "Command not found."[^6]

当脚本仅包含一些通用的系统命令而不使用 shell 内部指令时, 可以省略 `#!`。第三个例子需要 `#!` 是因为当对变量赋值时, 例如 `lines=50`, 使用了与 shell 特性相关的结构⁷。再重复一次, `#!/bin/sh` 调用的是系统默认 shell 解释器, 在 Linux 系统中默认为 `/bin/bash`。

这个例子鼓励读者使用模块化的方式编写脚本, 并在平时记录和收集一些在以后可能会用到的代码模板。最终你将拥有一个相当丰富易用的代码库。以下的代码可以用来测试脚本被调用时的参数数量是否正确。

```

1. E_WRONG_ARGS=85
2. script_parameters="-a -h -m -z"
3.             # -a = all, -h = help 等等
4.
5. if [ $# -ne $Number_of_expected_args ]
6. then
7.     echo "Usage: `basename $0` $script_parameters"
8.     # `basename $0` 是脚本的文件名
9.     exit $E_WRONG_ARGS
10. fi

```

大多数情况下，你会针对特定的任务编写脚本。本章的第一个脚本就是这样。然后你也许会泛化（generalize）脚本使其能够适应更多相似的任务，比如用变量代替硬编码，用函数代替重复代码。

[^2]：一些UNIX的衍生版（基于4.2 BSD）声称他们使用四字节的幻数，在#!后增加一个空格，即 `#!/bin/sh`。而Sven Mascheck指出这是虚构的。

[^3]：

命令解释器首先将会解释#!这一行，而因为#!以#打头，因此解释器将其视作注释。起始行作为调用解释器的作用已经完成了。

事实上即使脚本中含有不止一个#!，bash也会将除第一个 `#!` 以外的解释为注释。

```

1. #!/bin/bash

echo "Part 1 of script."
a=1

#!/bin/bash
# 这并不会启动新的脚本

echo "Part 2 of script."

```



```
echo $a # $a的值仍旧为1
```

[^4]:

这里允许使用一些技巧。

```
1. #!/bin/rm
# 自我删除的脚本

# 当你运行这个脚本，除了这个脚本本身消失以外并不会发生什么。

WHATEVER=85

echo "This line will never print (betcha!)."
```

```
exit $WHATEVER # 这没有任何关系。脚本将不会从这里退出。
               # 尝试在脚本终止后打印echo $a。
               # 得到的值将会是0而不是85。
```

当然你也可以建立一个起始行是 `#!/bin/more` 的README文件，并且使它可以执行。结果就是这个文件成为了一个可以打印本身的文件。（查看样例 19-3，使用 `cat` 命令的here document也许是一个更好的选择）

[^5]: 可移植操作系统接口（POSIX）尝试标准化类UNIX操作系统。POSIX规范可以在[Open Group site](#)中查看。

[^6]: 为了避免这种情况的发生，可以使用 `#!/bin/env bash` 作为起始行。这在bash不在 `/bin` 的UNIX系统中会有效果。

2.1 调用一个脚本

- 2.1 调用一个脚本

2.1 调用一个脚本

写完一个脚本以后，你可以通过 `sh scriptname` ^[^1] 或 `bash scriptname` 来调用它（不推荐使用 `sh <scriptname` 调用脚本，因为这会禁用脚本从标准输入（stdin）读入数据）。更方便的方式是使用 `chmod` 命令使脚本可以被直接执行。

执行命令：

```
chmod 555 scriptname
```

（给予所有用户读取/执行的权限）^{^2}

或

```
chmod +rx scriptname
```

（给予所有用户读取/执行的权限）

```
chmod u+rx scriptname
```

（仅给予脚本所有者读取/执行的权限）

当脚本的权限被设置好后，你就可以直接使用 `./scriptname` ^{^3} 进行调用测试了。如果脚本文件以 `sha-bang` 开头，那么它将自动调用指定的命令解释器运行脚本。

完成调试与测试后，你可能会将脚本文件移至 `/usr/local/bin`（使用 `root` 权限）中，使脚本可以被所有用户调用。这时你可以直接在命令行中输入 `scriptname [ENTER]` 执行脚本。

^[^1]：注意，当你使用 `sh scriptname` 调用 *Bash* 脚本时，将会禁用与 *Bash* 特性相关的功能，脚本有可能会执行失败。

2.2 牛刀小试

- [2.2 牛刀小试](#)

2.2 牛刀小试

1. 系统管理员通常会写一些脚本来完成自动化工作。试举例说明使用脚本的便利之处。
2. 请尝试写一个脚本。调用脚本，会打印当前系统时间和日期，所有已登录的用户和系统运行时间。并将这些信息保存到一个日志文件中。

第二部分 shell基础

- 第二部分 shell基础
 - 目录

第二部分 shell基础

目录

- 3. 特殊字符
- 4. 变量与参数
 - 4.1 变量替换
 - 4.2 变量赋值
 - 4.3 Bash弱类型变量
 - 4.4 特殊变量类型
- 5. 引用
 - 5.1 引用变量
 - 5.2 转义
- 6. 退出与退出状态
- 7. 测试
 - 7.1 测试结构
 - 7.2 文件测试操作
 - 7.3 其他比较操作
 - 7.4 嵌套 if/then 条件测试
 - 7.5 牛刀小试
- 8. 运算符和相关话题
 - 8.1 运算符
 - 8.2 数字常量
 - 8.3 双圆括号结构

◦ 8.4 运算符优先级

3. 特殊字符

- 第三章 特殊字符

- #
- ;
- ;;
- ;;&, ;&
- .
- .
- .
- "
- '
- ,
- ,, ,
- \
- /
- `
- :
- !
- *
- *
- ?
- ?
- \$
- \$
- \${}
- \$' ...'
- \$*, \$@

- `$?`
- `$$`
- `()`
- `{xxx,yyy,zzz,...}`
- `{a..z}`
- `{ }`
- `{ }`
- `{ } \;`
- `[]`
- `[[]]`
- `[]`
- `[]`
- `$(...)`
- `(())`
- `> &> >& >> < <>`
- `<<`
- `<<<`
- `<, >`
- `\<, >`
- `|`
- `>|`
- `||`
- `&`
- `&&`
- `-`
- `—`
- `-`
- `-`

- -
- =
- +
- +
- %
- ~
- ~+
- ~ -
- =~
- ^
- ^, ^^
- 控制字符
 - Ctrl-A
 - Ctrl-B
 - Ctrl-C
 - Ctrl-D
 - Ctrl-E
 - Ctrl-F
 - Ctrl-G
 - Ctrl-H
 - Ctrl-I
 - Ctrl-J
 - Ctrl-K
 - Ctrl-L
 - Ctrl-M
 - Ctrl-N
 - Ctrl-O
 - Ctrl-P

- Ctrl-Q
- Ctrl-R
- Ctrl-S
- Ctrl-T
- Ctrl-U
- Ctrl-V
- Ctrl-W
- Ctrl-X
- Ctrl-Y
- Ctrl-Z
- 空白符

第三章 特殊字符

是什么让一个字符变得特殊呢？如果一个字符不仅具有字面意义，而且具有元意（*meta-meaning*），我们就称它为特殊字符。特殊字符同命令和关键词（keywords）一样，是bash脚本的组成部分。

你在脚本或其他地方都能够找到特殊字符。

#

注释符。如果一行脚本的开头是#（除了#!），那么代表这一行是注释，不会被执行。

```
1. # 这是一行注释
```

注释也可能会在一行命令结束之后出现。

```
1. echo "A comment will follow." # 这儿可以写注释
2. # ^ 注意在#之前有空格
```

注释也可以出现在一行开头的空白符 (whitespace) 之后。

```
1.      # 这个注释前面存在着一个制表符 (tab)
```

注释甚至可以嵌入到管道命令 (pipe) 之中。

```
1. initial=( `cat "$startfile" | sed -e '/#/d' | tr -d '\n' | \
2. # 删除所有带'#'注释符号的行
3.      sed -e 's/\./\./g' -e 's/_/_/g'` )
4. # 摘录自脚本 life.sh
```



命令不能写在同一行注释之后。因为没有任何方法可以结束注释(仅支持单行注释)，为了让新命令正常执行，另起一行写吧。



当然，在 `echo` 语句中被引用或被转义的#不会被认为是注释。同样，在某些参数替换式或常量表达式中的#也不会被认为是注释。

```
1. echo "The # here does not begin a comment."
2. echo 'The # here does not begin a comment.'
3. echo The \# here does not begin a comment.
4. echo The # here begins a comment.
5.
6. echo ${PATH#*:}      # 参数替换而非注释
7. echo $( ( 2#101011 ) ) # 进制转换而非注释
8.
9. # 感谢S.C.
```

因为引用符和转义符 (" ' \) 转义了#。

一些模式匹配操作同样使用了#。

;

命令分隔符[分号]。允许在同一行内放置两条或更多的命令。

```

1. echo hello; echo there
2.
3. if [ -x "$filename" ]; then    # 注意在分号以后有一个空格
4.     ++                        ^^
5.     echo "File $filename exists."; cp $filename $filename.bak
6. else    #                        ^^
7.     echo "File $filename not found."; touch $filename
8. fi; echo "File test complete."

```

注意有时候";"需要被转义才能正常工作。

;;

case 条件语句终止符[双分号]。

```

1. case "$variable" in
2.     abc) echo "\$variable = abc" ;;
3.     xyz) echo "\$variable = xyz" ;;
4. esac

```

;;&, ;&

case 条件语句终止符 (Bash4+ 版本)。

.

句点命令[句点]。等价于 `source` 命令 (查看样例 15-22)。这是一个bash的内建命令。

.

句点可以作为文件名的一部分。如果它在文件名开头，那说明此文件是隐藏文件。使用不带参数的 `ls` 命令不会显示隐藏文件。

```

1. bash$ touch .hidden-file
2. bash$ ls -l
3. total 10
4.  -rw-r--r--    1 bozo      4034 Jul 18 22:04 data1.addressbook
5.  -rw-r--r--    1 bozo      4602 May 25 13:58 data1.addressbook.bak
6.  -rw-r--r--    1 bozo       877 Dec 17  2000 employment.addressbook
7.
8.
9. bash$ ls -al
10. total 14
11.  drwxrwxr-x    2 bozo   bozo    1024 Aug 29 20:54 ./
12.  drwx-----   52 bozo   bozo    3072 Aug 29 20:51 ../
13.  -rw-r--r--    1 bozo   bozo    4034 Jul 18 22:04
    data1.addressbook
14.  -rw-r--r--    1 bozo   bozo    4602 May 25 13:58
    data1.addressbook.bak
15.  -rw-r--r--    1 bozo   bozo       877 Dec 17  2000
    employment.addressbook
16.  -rw-rw-r--    1 bozo   bozo         0 Aug 29 20:54 .hidden-file

```

当句点出现在目录中时，单个句点代表当前工作目录，两个句点代表上级目录。

```

1. bash$ pwd
2. /home/bozo/projects
3.
4. bash$ cd .
5. bash$ pwd
6. /home/bozo/projects
7.
8. bash$ cd ..
9. bash$ pwd
10. /home/bozo/

```

句点通常代表文件移动的目的地（目录），下式代表的是当前目录。

```
1. bash$ cp /home/bozo/current_work/junk/* .
```

复制所有的“垃圾文件”到 [当前目录](#)

.

句点匹配符。在正则表达式中，点号意味着匹配任意单个字符。

//

部分引用[双引号]。在字符串中保留大部分特殊字符。详细内容将在[第五章](#)介绍。

'

全引用[单引号]。在字符串中保留所有的特殊字符。是部分引用的强化版。详细内容将在[第五章](#)介绍。

,

逗号运算符。逗号运算符[`^1`]将一系列的算术表达式串联在一起。算术表达式依次被执行，但只返回最后一个表达式的值。

```
1. let "t2 = ((a = 9, 15 / 3))"
2. # a被赋值为9, t2被赋值为15 / 3
```

逗号运算符也可以用来连接字符串。

```
1. for file in /{,usr/}bin/*calc
2. #           ^    在 /bin 与 /usr/bin 目录中
3. #+         找到所有的以"calc"结尾的可执行文件
4. do
5.     if [ -x "$file" ]
6.     then
7.         echo $file
8.     fi
```

```

9.  done
10.
11.  # /bin/ipcalc
12.  # /usr/bin/kcalc
13.  # /usr/bin/oidcalc
14.  # /usr/bin/oocalc
15.
16.  # 感谢Rory Winston提供的执行结果

```

// /

在参数替换中进行小写字母转换（ Bash4 新增 ）。

\

转义符[反斜杠]。转义某字符的标志。

`\x` 转义了字符x。双引号""内的x与单引号内的x具有同样的效果。
转义符也可以用来转义"与'，使它们表达其字面含义。

第五章将更加深入的解释转义字符。

/

文件路径分隔符[正斜杠]。起分割路径的作用。（ 比如

`/home/bozo/projects/Makefile` ）

它也在算术运算中充当除法运算符。

`

命令替换符。 ``command`` 结构可以使得命令的输出结果赋值给一个变量。通常也被称作后引号（ backquotes ）或反引号（ backticks ）。

:

空命令[冒号]。它在shell中等价于“NOP”（即no op，空操作）与shell内建命令true有同样的效果。它本身也是Bash的内建命令之一，返回值是true（0）。

```
1. :
2. echo $? # 返回0
```

在无限循环中的应用：

```
1. while :
2. do
3.     operation-1
4.     operation-2
5.     ...
6.     operation-n
7. done
8.
9. # 等价于
10. # while true
11. # do
12. #     ...
13. # done
```

可在 `if/then` 中充当占位符：

```
1. if condition
2. then : # 什么都不做，跳出判断执行下一条语句
3. else
4.     take-some-action
5. fi
```

在二元操作中作占位符：查看样例 8-2或默认参数部分。

```
1. : ${username=`whoami`}
```



```

2. # ${username=`whoami`}    如果没有:就会报错
3. #                        除非 "username" 是系统命令或内建命令
4.
5. : ${1?"Usage: $0 ARGUMENT"}    # 摘自样例脚本 "usage-message.sh"

```

查看样例 19-10 了解空命令在 here document 中作为占位符的情况。

使用参数替换为字符串变量赋值（查看样例 10-7）。

```

1. : ${HOSTNAME?} ${USER?} ${MAIL?}
2. # 如果其中一个或多个必要的环境变量没有被设置
3. # 将会打印错误

```

查看变量扩展或字符串替换章节了解空命令在其中的作用。

与 `>` 重定向操作符结合，可以在不改变文件权限的情况下清空文件。如果文件不存在，那么将创建这个文件。

```

1. : > data.xxx    # 文件 "data.xxx" 已被清空
2.
3. # 与 cat /dev/null >data.xxx 作用相同
4. # 但是此操作不会产生一个新进程，因为 ":" 是 shell 内建命令。

```

也可查看样例 16-15。

与 `>>` 重定向操作符结合，将不会清空任何已存在的文件（`:>>target_file`）。如果文件不存在，将创建这个文件。



以上操作仅适用于普通文件，不适用于管道、符号链接和特殊文件。

空命令可以用来作为一行注释的开头，尽管我们并不推荐这么做。使用 `#` 可以使解释器关闭该行的错误检测，所以几乎所有的内容都可以出现在注释 `#` 中。使用空命令却不是这样的：

1. `:` 这一行注释将会产生一个错误, (`if [$x -eq 3]`)。

`:`也可以作为一个域分隔符, 比如在 `/etc/passwd` 和 `$PATH` 变量中。

```
1. bash$ echo $PATH
2. /usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/sbin:/usr/sbin:/usr/games
```

将冒号作为函数名也是可以的。

```
1. :()
2. {
3.     echo "The name of this function is "$FUNCNAME" "
4.     # 为什么要使用冒号作函数名?
5.     # 这是一种混淆代码的方法.....
6. }
7.
8. :
9.
10. # 函数名是 :
```

这种写法并不具有可移植性, 也不推荐使用。事实上, 在Bash的最近的版本更新中已经禁用了这种用法。但我们还可以使用下划线 `_` 来替代。

冒号也可以作为非空函数的占位符。

```
1. not_empty ()
2. {
3.     :
4. } # 含有空指令, 这并不是一个空函数。
```

!

取反 (或否定) 操作符[感叹号]。! 操作符反转已执行的命令的返回

状态（查看样例 6-2）。它同时可以反转测试操作符的意义，例如可以将相等（=）反转成不等（!=）。它是一个Bash关键词。

在一些特殊场景下，它也会出现在间接变量引用中。

在另外一些特殊场景下，即在命令行下可以使用 `!` 调用Bash的历史记录（附录 L）。需要注意的是，在脚本中，这个机制是被禁用的。

★

通配符[星号]。在文件匹配（globbing）操作时扩展文件名。如果它独立出现，则匹配该目录下的所有文件。

```
1. bash$ echo *
```

```
2. abs-book.sgm1 add-drive.sh agram.sh alias.sh
```

在正则表达式中表示匹配任意多个（包括0）前个字符。

★

算术运算符。在进行算术运算时，表示乘法运算。

★★ 双星号可以表示乘方运算或扩展文件匹配。

?

测试操作符[问号]。在一些特定的语句中，`?` 表示一个条件测试。

在一个双圆括号结构中，`?` 可以表示一个类似C语言风格的三元（trinary）运算符的一个组成部分。[^2]

```
condition?result-if-true:result-if-false
```

```
1. (( var0 = var1<98?9:21 ))
```

```
2. #不要加空格，紧挨着写
```

```

3.
4. #等价于
5. # if [ "$var1" -lt 98 ]
6. # then
7. #   var0=9
8. # else
9. #   var0=21
10. # fi

```

在参数替换表达式中，`?` 用来测试一个变量是否已经被赋值。

`?`

通配符。它在进行文件匹配（globbing）时以单字符通配符扩展文件名。

在扩展正则表达式中匹配一个单字符。

`$`

取值符号[钱字符]，用来进行变量替换（即取出变量的内容）。

```

1. var1=5
2. var2=23skidoo
3.
4. echo $var1      # 5
5. echo $var2      # 23skidoo

```

如果在变量名前有 `$`，则表示此变量的值。

`$`

行结束符[EOF]。

在正则表达式中，`$` 匹配行尾字符串。

`${}`

参数替换。

`$'...'`

引用字符串扩展。这个结构将转义八进制或十六进制的值转换成ASCII^[^3]或Unicode字符。

`$*, $@`

位置参数。

`$?`

返回状态变量。此变量保存一个命令、一个函数或该脚本自身的返回状态。

`$$`

进程ID变量。此变量保存该运行脚本的进程ID^[^4]。

`()`

命令组。

```
(a=hello; echo $a)
```



通过括号执行一系列命令会产生一个子shell (subshell)。括号中的变量，即在子shell中的变量，在脚本的其他部分是不可见的。父进程脚本不能访问子进程 (子shell) 所创建的变量。

```
1. a=123
2. ( a=321; )
```

- 3.
4. `echo "a = $a" # a = 123`
5. `#` 在括号中的 `"a"` 就像个局部变量。

数组初始化。

1. `Array=(element1 element2 element3)`

`{xxx, yyy, zzz, ...}`

花括号扩展结构。

1. `echo \"{These, words, are, quoted}\" # " 将作为单词的前缀和后缀`
2. `# "These" "words" "are" "quoted"`
- 3.
- 4.
5. `cat {file1, file2, file3} > combined_file`
6. `# 将 file1, file2 与 file3 拼接在一起后写入 combined_file 中。`
- 7.
8. `cp file22.{txt, backup}`
9. `# 将 "file22.txt" 拷贝为 "file22.backup"`

这个命令可以作用于花括号内由逗号分隔的文件描述列表。⁴⁵ 文件名扩展（匹配）作用于大括号间的各个文件。



除非被引用或被转义，否则空白符不应在花括号中出现。

1. `echo {file1, file2}\ : {\ A, " B", ' C'}`
2. `file1 : A file1 : B file1 : C file2 : A file2 : B file2 : C`

`{a..z}`

扩展的花括号扩展结构。

```

1. echo {a..z} # a b c d e f g h i j k l m n o p q r s t u v w x y z
2. # 输出 a 到 z 之间所有的字母。
3.
4. echo {0..3} # 0 1 2 3
5. # 输出 0 到 3 之间所有的数字。
6.
7.
8. base64_charset=( {A..Z} {a..z} {0..9} + / = )
9. # 使用扩展花括号初始化一个数组。
10. # 摘自 vladz 编写的样例脚本 "base64.sh"。

```

Bash第三版中引入了 `{a..z}` 扩展的花括号扩展结构。

`{ }`

代码块[花括号]，又被称作内联组 (inline group)。它实际上创建了一个匿名函数 (anonymous function)，即没有名字的函数。但是，不同于那些“标准”函数，代码块内的变量在脚本的其他部分仍旧是可见的。

```

1. bash$ { local a;
2.             a=123; }
3. bash: local: can only be used in a
4. function

```

```

1. a=123
2. { a=321; }
3. echo "a = $a"    # a = 321    (代码块内赋值)
4.
5. # 感谢S.C.

```

代码块可以经由I/O重定向进行输入或输出。

样例 3-1. 代码块及I/O重定向

```

1.  #!/bin/bash
2.  # 读取文件 /etc/fstab
3.
4.  File=/etc/fstab
5.
6.  {
7.  read line1
8.  read line2
9.  } < $File
10.
11. echo "First line in $File is:"
12. echo "$line1"
13. echo
14. echo "Second line in $File is:"
15. echo "$line2"
16.
17. exit 0
18.
19. # 你知道如何解析剩下的行吗？
20. # 提示：使用 awk 或...
21. # Hans-Joerg Diers 建议：使用Bash的内建命令 set。

```

样例 3-2. 将代码块的输出保存至文件中

```

1.  #!/bin/bash
2.  # rpm-check.sh
3.
4.  # 查询一个rpm文件的文件描述、包含文件列表，以及是否可以被安装。
5.  # 将输出保存至文件。
6.  #
7.  # 这个脚本使用代码块来描述。
8.
9.  SUCCESS=0
10. E_NOARGS=65
11.
12. if [ -z "$1" ]
13. then

```



```

14.     echo "Usage: `basename $0` rpm-file"
15.     exit $E_NOARGS
16. fi
17.
18. { # 代码块起始
19.     echo
20.     echo "Archive Description:"
21.     rpm -qpi $1          # 查询文件描述。
22.     echo
23.     echo "Archive Listing:"
24.     rpm -qpl $1          # 查询文件列表。
25.     echo
26.     rpm -i --test $1     # 查询是否可以被安装。
27.     if [ "$?" -eq $SUCCESS ]
28.     then
29.         echo "$1 can be installed."
30.     else
31.         echo "$1 cannot be installed."
32.     fi
33.     echo                 # 代码块结束。
34. } > "$1.test"           # 输出重定向至文件。
35.
36. echo "Results of rpm test in file $1.test"
37.
38. # rpm各项参数的具体含义可查看man文档
39.
40. exit 0

```



与由圆括号包裹起来的命令组不同，由花括号包裹起来的代码块不产生子进程。[^6]

也可以使用非标准的 `for` 循环语句来遍历代码块。

{ }

文本占位符。在 `xargs -i` 后作为输出的占位符使用。

```
1. ls . | xargs -i -t cp ./{} $1
2. #           ^^           ^^
3.
4. # 摘自 "ex42.sh" (copydir.sh)
```

{ } \;

路径名。通常在 `find` 命令中使用，但这不是shell的内建命令。

定义：路径名是包含完整路径的文件名，例

如 `/home/bozo/Notes/Thursday/schedule.txt`。我们通常又称之为绝对路径。



在执行 `find -exec` 时最后需要加上 `;`，但是分号需要被转义以保证其不会被shell解释。

[]

测试。在 `[]` 之间填写测试表达式。值得注意的是，`[` 是shell内建命令 `test` 的一个组成部分，而不是外部命令 `/usr/bin/test` 的链接。

[[]]

测试。在 `[[]]` 之间填写测试表达式。相比起单括号测试 (`[]`)，它更加的灵活。它是一个shell的关键字。

详情查看关于 `[[]]` 结构的讨论。

[]

数组元素。在数组中，可以使用中括号的偏移量来用来访问数组中的每一个元素。

```
1. Array[1]=slot_1
```

```
2. echo ${Array[1]}
```

[]

字符集、字符范围。

在正则表达式中，中括号用来匹配指定字符集或字符范围内的任意字符。

\$[...]

整数扩展符。在 `$[]` 中可以计算整数的算术表达式。

```
1. a=3
2. b=7
3.
4. echo $[a+b]    # 10
5. echo $[a*b]    # 21
```

(())

整数扩展符。在 `(())` 中可以计算整数的算术表达式。

详情查看关于 `((...))` 结构的讨论。

> &> >& >> < <>

重定向。

`scriptname >filename` 将脚本 *scriptname* 的输出重定向到 *filename* 中。如果文件存在，那么覆盖掉文件内容。

`command &>filename` 将命令 *command* 的标准输出(stdout) 和标准错误输出(stderr) 重定向到 *filename*。



重定向在用于清除测试条件的输出时特别有效。例如测试一个特定的命令是否存在。

```
1. bash$ type bogus_command &>/dev/null
2.
3.
4. bash$ echo $?
5. 1
```

或写在脚本中：

```
1. command_test () { type "$1" &>/dev/null; }
2. #
3.
4. cmd=rmdir # 存在的命令。
5. command_test $cmd; echo $? # 返回0
6.
7.
8. cmd=bogus_command # 不存在的命令。
9. command_test $cmd; echo $? # 返回1
```

`command >&2` 将命令的标准输出重定向至标准错误输出。

`scriptname >>filename` 将脚本 *scriptname* 的输出追加到 *filename* 文件末尾。如果文件不存在，那么将创建这个文件。

`[i]<>filename` 打开文件 *filename* 用来读写，并且分配一个文件描述符*i*指向它。如果文件不存在，那么将创建这个文件。

进程替换：

`(command)>`

`<(command)`

在某些情况下，“<”与“>”将用作字符串比较。

在另外一些情况下，“<”与“>”将用作数字比较。详情查看样例 16-9。

<<

在here document中进行重定向。

<<<

在here string中进行重定向。

<, >

ASCII码比较。

```
1. veg1=carrots
2. veg2=tomatoes
3.
4. if [[ "veg1" < "veg2" ]]
5. then
6.     echo "Although $veg1 precede $veg2 in the dictionary,"
7.     echo -n "this does not necessarily imply anything "
8.     echo "about my culinary preferences."
9. else
10.    echo "What kind of dictionary are you using, anyhow?"
11. fi
```

\<, >

正则表达式中的单词边界 (word boundary)。

```
1. bash$ grep '\<the\>' textfile
```

|

管道 (pipe)。管道可以将上一个命令的输出作为下一个命令的输入，或者直接输出到shell中。管道是一种可以将一系列命令连接在一起的绝妙方式。

1. `echo ls -l | sh`
2. # 将 "echo ls -l" 的结果输出到shell中，
3. # 与直接输入 "ls -l" 的结果相同。
- 4.
- 5.
6. `cat *.lst | sort | uniq`
7. # 将所有后缀名为 `lst` 的文件合并后排序，接着删掉所有重复行。

管道是一种在进程间通信的典型方法。它将一个进程的输出作为另一个进程的输入。举一个经典的例子，像 `cat` 或者 `echo` 这样的命令，可以通过管道将它们产生的数据流导入到过滤器 (*filter*) 中。过滤器是可以用来处理输入流的命令。[⁷]

```
cat $filename1 $filename2 | grep $search_word
```

查看[UNIX FAQ第三章](#)获取更多关于使用UNIX管道的信息。

命令的输出同样可以通过管道输入到脚本中。

1. `#!/bin/bash`
2. # `uppercase.sh` : 将所有输入变成大写
- 3.
4. `tr 'a-z' 'A-Z'`
5. # 为了防止产生单字符文件名，
6. # 必须使用单引号引用字符范围。
- 7.
8. `exit 0`

现在，让我们将 `ls -l` 的输出通过管道导入到脚本中。

1. `bash$ ls -l | ./uppercase.sh`
2. `-RW-RW-R-- 1 BOZO BOZO 109 APR 7 19:49 1.TXT`
3. `-RW-RW-R-- 1 BOZO BOZO 109 APR 14 16:48 2.TXT`
4. `-RW-R--R-- 1 BOZO BOZO 725 APR 20 20:56 DATA-FILE`



在管道中，每一个进程的输出必须作为下个进程的输入被正确读入，如果不这样，数据流会被阻塞（block），管道就无法按照预期正常工作。

```
1. cat file1 file2 | ls -l | sort
2. # "cat file1 file2" 的输出会消失。
```

管道是在一个子进程中运行的，因此它并不能修改父进程脚本中的变量。

```
1. variable="initial_value"
2. echo "new_value" | read variable
3. echo "variable = $variable"      # variable = initial_value
```

如果管道中的任意一个命令意外中止了，管道将会提前中断，我们称其为管道破裂(Broken Pipe)。出现这种情况，系统将发送一个 `SIGPIPE` 信号。

>|

强制重定向。即使在 `noclobber` 选项被设置的情况下，重定向也会覆盖已存在的文件。

||

或（OR）逻辑运算符。在测试结构中，任意一个测试条件为真，整个表达式为真。返回 0（成功标志位）。

&

后台运行操作符。如果命令后带&，那么此命令将转至后台运行。

```

1. bash$ sleep 10 &
2. [1] 850
3. [1]+  Done                  sleep 10

```

在脚本中，命令甚至循环都可以在后台运行。

样例 3-3. 在后台运行的循环

```

1. #!/bin/bash
2. # background-loop.sh
3.
4. for i in 1 2 3 4 5 6 7 8 9 10          # 第一个循环
5. do
6.     echo -n "$i "
7. done & # 这个循环在后台运行。
8.       # 有时会在第二个循环结束之后才执行此后台循环。
9.
10. echo  # 此'echo' 有时不显示
11.
12. for i in 11 12 13 14 15 16 17 18 19 20  # 第二个循环
13. do
14.     echo -n "$i "
15. done
16.
17. echo  # 此'echo' 有时不显示
18.
19. # =====
20.
21. # 脚本期望输出结果：
22. # 1 2 3 4 5 6 7 8 9 10
23. # 11 12 13 14 15 16 17 18 19 20
24.
25. # 一些情况下可能会输出：
26. # 11 12 13 14 15 16 17 18 19 20
27. # 1 2 3 4 5 6 7 8 9 10 bozo $
28. # 第二个 'echo' 没有被执行，为什么？
29.

```



```

30. # 另外一些情况下可能会输出：
31. # 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
32. # 第一个 'echo' 没有被执行，为什么？
33.
34. # 非常罕见的情况下，可能会输出：
35. # 11 12 13 1 2 3 4 5 6 7 8 9 10 14 15 16 17 18 19 20
36. # 前台循环抢占（preempt）了后台循环。
37.
38. exit 0
39.
40. # Nasimuddin Ansari 建议：在第6行和第14行的
41. # echo -n "$i " 后增加 sleep 1,
42. # 会得到许多有趣的结果。

```



脚本在后台执行命令时可能因为等待键盘事件被挂起。幸运的是，有一套方案可以解决这个问题。

&&

与（AND）逻辑操作符。在测试结构中，所有测试条件都为真，表达式才为真，返回 0（成功标志位）。

-

选项与前缀。它可以作为命令的选项标志，也可以作为一个操作符的前缀，也可以作为在参数代换中作为默认参数的前缀。

```
COMMAND -[Option1][Option2][..]
```

```
ls -al
```

```
sort -dfu $filename
```

```

1. if [ $file1 -ot $file2 ]
2. then #      ^
3.     echo "File $file1 is older than $file2."

```

```

4.  fi
5.
6.  if [ "$a" -eq "$b" ]
7.  then #      ^
8.      echo "$a is equal to $b."
9.  fi
10.
11. if [ "$c" -eq 24 -a "$d" -eq 47 ]
12. then #      ^          ^
13.     echo "$c equals 24 and $d equals 47."
14. fi
15.
16.
17. param2=${param1:-$DEFAULTVAL}
18. #      ^

```

双横线一般作为命令长选项的前缀。

```
sort --ignore-leading-blanks
```

双横线与Bash内建命令一起使用时，意味着该命令选项的结束。



下面提供了一种删除文件名以横线开头文件的简单方法。

```

1. bash$ ls -l
2. -rw-r--r-- 1 bozo bozo 0 Nov 25 12:29 -badname
3.
4.
5. bash$ rm -- -badname
6.
7. bash$ ls -l
8. total 0

```

双横线通常也和 `set` 连用。

`set -- $variable` (查看样例 15-18)。

-

重定向输入输出[短横线]。

```
1. bash$ cat -
2. abc
3. abc
4.
5. ...
6.
7. Ctl-D
```

在这个例子中，`cat -` 输出由键盘读入的标准输入(stdin) 到 标准输出(stdout)。但是在真实应用的 I/O 重定向中是否有使用 `'-'` ？

```
1. (cd /source/directory && tar cf - . ) | (cd /dest/directory && tar
   xpvf -)
2.
3. # 将整个文件树从一个目录移动到另一个目录。
4. # 感谢 Alan Cox <a.cox@swansea.ac.uk> 所作出的部分改动
5.
6. # 1) cd /source/directory
7. #    工作目录定位到文件所属的源目录
8. # 2) &&
9. #    "与链": 如果 'cd' 命令操作成功, 那么执行下一条命令
10. # 3) tar cf - .
11. #    'tar c' (create 创建) 创建一份新的档案
12. #    'tar f -' (file 指定文件) 在 '-' 后指定一个目标文件作为输出
13. #    '.' 代表当前目录
14. # 4) |
15. #    通过管道进行重定向
16. # 5) ( ... )
17. #    在建立的子进程中执行命令
```

```

18. # 6) cd /dest/directory
19. #     工作目录定位到目标目录
20. # 7) &&
21. #     与 2) 相同
22. # 8) tar xpvf -
23. #     'tar x' 解压档案
24. #     'tar p' (preserve 保留) 保留档案内文件的所有权及文件权限
25. #     'tar v' (verbose 冗余) 发送全部信息到标准输出
26. #     'tar f -' (file 指定文件) 在 '-' 后指定一个目标文件作为输入
27. #
28. #     注意 'x' 是一个命令, 而 'p', 'v', 'f' 是选项。
29.
30. # 干的漂亮!
31.
32.
33. # 更加优雅的写法是:
34. #     cd source/directory
35. #     tar cf - . | (cd ../dest/directory; tar xpvf -)
36. #
37. # 同样可以写成:
38. #     cp -a /source/directory/* /dest/directory
39. # 或:
40. #     cp -a /source/directory/* /source/directory/.[^.]*
41. #     /dest/directory
41. # 可以在源目录中有隐藏文件时使用

```

```

1. bunzip2 -c linux-2.6.16.tar.bz2 | tar xvf -
2. # --未解压的 tar 文件--          | --将解压出的 tar 传递给 "tar"--
3. # 如果不使用管道让 "tar" 处理 "bunzip2" 得到的文件,
4. # 那么就需要使用单独的两步来完成。
5. # 目的是了解压 "bziped" 压缩的内核源代码。

```

下面的例子中, "-" 并不是一个Bash的操作符, 它仅仅是 `tar` ,
`cat` 等一些特定UNIX命令中将结果输出到标准输出的选项。

```

1. bash$ echo "whatever" | cat -
2. whatever

```

当需要文件名的时候，`-` 可以用来代替某个文件而重定向到标准输出（通常出现在 `tar cf` 中）或从 `stdin` 中接受数据。这是一种在管道中使用面向文件（file-oriented）工具作为过滤器的方法。

```
1. bash$ file
2. Usage: file [-bciknvzL] [-f namefile] [-m magicfiles] file...
```

单独执行 `file` 命令，将会得到一条错误信息。

在命令后增加一个 `"-"` 可以得到一个更加有用的结果。它会使得 shell 暂停等待用户输入。

```
1. bash$ file -
2. abc
3. standard input:          ASCII text
4.
5.
6. bash$ file -
7. #!/bin/bash
8. standard input:          Bourne-Again shell script text
   executable
```

现在命令能够接受标准输入并且处理它们了。

`"-"` 能够通过管道将标准输出重定向到其他命令中。这就可以做到像在某个文件前添加几行这样的事情。

使用 `diff` 比较两个文件的部分内容：

```
1. grep Linux file1 | diff file2 -
```

最后介绍一个使用 `-` 的 `tar` 命令的实际案例。

样例 3-4. 备份最近一天修改过的所有文件

```

1.  #!/bin/bash
2.
3.  # 将当前目录下24小时之内修改过的所有文件备份成一个
4.  # "tarball" (经 tar 打包与 gzip 压缩) 文件
5.
6.  BACKUPFILE=backup-$(date +%m-%d-%Y)
7.  #                      在备份文件中嵌入时间
8.  #                      感谢 Joshua Tschida 提供的建议
9.
10. archive=${1:-$BACKUPFILE}
11. # 如果没有在命令行中特别制定备份格式,
12. # 那么将会默认设置为 "backup-MM-DD-YYYY.tar.gz"。
13.
14. tar cvf - `find . -mtime -1 -type f -print` > $archive.tar
15. gzip $archive.tar
16. echo "Directory $PWD backed up in archive file
    \"$archive.tar.gz\"."
17.
18. # Stephane Chazeles 指出如果目录中有非常多的文件,
19. # 或文件名中包含空白符时, 上面的代码会运行失败。
20.
21. # 他建议使用以下的任何一种方法:
22. # -----
23. # --
24. # find . -mtime -1 -type f -print0 | xargs -0 tar rvf
    "$archive.tar"
25. # 使用了 GNU 版本的 "find" 命令。
26.
27. # find . -mtime -1 -type f -exec tar rvf "$archive.tar" '{}' \;
28. # 兼容其他的 UNIX 发行版, 但是速度会比较慢
29. # -----
30. # --
31.
32. exit 0

```

⚠ 以 “-” 开头的文件在和 “-” 重定向操作符一起使用时可能会导致一些问题。因此合格的脚本必须首先检查这种情况。如果遇到，就需要给文件名加一个合适的前缀，比如 `./-FILENAME`, `$PWD/-FILENAME` 或者 `$PATHNAME/-FILENAME`。

如果变量的值以 “-” 开头，也可能造成类似问题。

```
1. var='-n'
2. echo $var
3. # 等同于 "echo -n", 不会输出任何东西。
```

-

先前的工作目录。使用 `cd -` 命令可以返回先前的工作目录。它实际上是使用了 `$OLDPWD` 环境变量。

⚠ 不要将这里的 “-” 与先前的 “-” 重定位操作符混淆。“-” 的具体含义需要根据上下文来解释。

-

减号。算术运算符中的减法标志。

=

等号。赋值操作符。

```
1. a=28
2. echo $a    # 28
```

在一些情况下，“=” 可以作为字符串比较操作符。

+

加号。加法算术运算。

在一些情况下，+ 是作为正则表达式中的一个操作符。

+

选项操作符。作为一个命令或过滤器的选项标记。

特定的一些指令和内建命令使用 + 启用特定的选项，使用 - 禁用特定的选项。在参数代换中，+ 是作为变量扩展的备用值 (alternate value) 的前缀。

%

取模。取模操作运算符。

```
1. let "z = 5 % 3"
2. echo $z # 2
```

在另外一些情况下，% 是一种模式匹配的操作符。

~

主目录[波浪号]。它相当于内部变量 `$HOME`。 `~bozo` 是 bozo 的主目录，执行 `ls ~bozo` 将会列出他的主目录中内容。 `~/` 是当前用户的主目录，执行 `ls ~/` 将会列出其中所有的内容。

```
1. bash$ echo ~bozo
2. /home/bozo
3.
4. bash$ echo ~
5. /home/bozo
```



```

6.
7.  bash$ echo ~/
8.  /home/bozo/
9.
10. bash$ echo ~:
11. /home/bozo:
12.
13. bash$ echo ~nonexistent-user
14. ~nonexistent-user

```

~+

当前工作目录。它等同于内部变量 `$PWD` 。

~-

先前的工作目录。它等同于内部变量 `$OLDPWD` 。

=~

正则表达式匹配。将在 *Bash version 3* 章节中介绍。

^

行起始符。在正则表达式中，“^” 代表一行文本的开始。

^, ^^

参数替换中的大写转换符（在Bash第4版新增）。

控制字符

改变终端或文件显示的一些行为。一个控制符是由 *CTRL* + *key* 组成的（同时按下）。控制字符同样可以通过转义以八进制或十六进制

的方式显示。

控制符不能在脚本中使用。

Ctrl-A

移动光标至行首。

Ctrl-B

非破坏性退格（即不删除字符）。

Ctrl-C

中断指令。终止当前运行的任务。

Ctrl-D

登出shell（类似 `exit`）

键入 `EOF`（end-of-file，文件终止标记），中断 *stdin* 的输入。

当你在终端或 *xterm* 窗口中输入字符时，`Ctrl-D` 将会删除光标上的字符。当没有字符时，`Ctrl-D` 将会登出shell。在 *xterm* 中，将会关闭整个窗口。

Ctrl-E

移动光标至行末。

Ctrl-F

光标向前移动一个字符。

Ctrl-G

响铃 `BEL`。在一些老式打字机终端上，将会响铃。而在 *xterm* 中，将会产生“哔”声。

Ctrl-H

抹除（破坏性退格）。退格删除前面的字符。

```

1.  #!/bin/bash
2.  # 在字符串中嵌入 Ctrl-H
3.
4.  a="^H^H"                                # 两个退格符 Ctrl-H
5.                                     # 在 vi/vim 中使用 Ctrl-V Ctrl-H 来键入
6.  echo "abcdef"                            # abcdef
7.  echo
8.  echo -n "abcdef$a "                     # abcd f
9.  #                                     ^      ^ 末尾有空格退格两次的结果
10. echo
11. echo -n "abcdef$a"                      # abcdef
12. #                                     ^ 末尾没有空格时为什么退格无效了？
13.                                     # 并不是我们期望的结果。
14. echo; echo
15.
16. # Constantin Hagemeyer 建议尝试一下：
17. # a=$'\010\010'
18. # a=$'\b\b'
19. # a=$'\x08\x08'
20. # 但是这些并不会改变结果。
21.
22. #####
23.
24. # 现在来试试这个。
25.
26. rubout="^H^H^H^H^H"                     # 5个 Ctrl-H
27.
28. echo -n "12345678"
```

```
29. sleep 2
30. echo -n "$rubout"
31. sleep 2
```

Ctrl-I

水平制表符。

Ctrl-J

另起一行（换行）。在脚本中，你也可使用八进制 ‘\012’ 或者十六进制 ‘\x0a’ 来表示。

Ctrl-K

垂直制表符。

当你在终端或 *xterm* 窗口中输入字符时，`Ctrl-K` 将会删除光标上及其后的所有字符。而在脚本中，`Ctrl-K` 的作用有些不同。具体查看下方 Lee Lee Maschmeyer 写的样例。

Ctrl-L

清屏、走纸。在终端中等同于 `clear` 命令。在打印时，`Ctrl-L` 将会使纸张移动到底部。

Ctrl-M

回车（CR）。

```
1. #!/bin/bash
2. # 感谢 Lee Maschmeyer 提供的样例。
3.
4. read -n 1 -s -p \
5. '$Control-M leaves cursor at beginning of this line. Press Enter.'
```

```

    \x0d'
6.          # '0d' 是 Control-M 的十六进制的值
7. echo >&2  # '-s' 参数禁用了回显，所以需要显式的另起一行。
8.
9. read -n 1 -s -p '$Control-J leaves cursor on next line. \x0a'
10.         # '0a' 是 Control-J 换行符的十六进制的值
11. echo >&2
12.
13. ###
14.
15. read -n 1 -s -p '$And Control-K\x0bgoes straight down.'
16. echo >&2  # Control-K 是垂直制表符。
17.
18. # 一个更好的垂直制表符例子是：
19.
20. var='$\x0aThis is the bottom line\x0bThis is the top line\x0a'
21. echo "$var"
22. # 这将会产生与上面的例子类似的结果。但是
23. echo "$var" | col
24. # 这却会使得右侧行高于左侧行。
25. # 这也解释了为什么我们需要在行首和行尾加上换行符
26. # 来避免显示的混乱。
27.
28. # Lee Maschmeyer 的解释：
29. # -----
30. # 在第一个垂直制表符的例子中，垂直制表符使其
31. # 在没有回车的情况下向下打印。
32. # 这在那些不能回退的设备上，例如 Linux 的终端才可以。
33. # 而垂直制表符的真正目的是向上而非向下。
34. # 它可以用来在打印机中用来打印上标。
35. # col 工具可以用来模拟真实的垂直制表符行为。
36.
37. exit 0

```

Ctrl-N

在命令行历史记录中调用下一条历史命令[⁸]。

Ctrl-O

在命令行中另起一行。

Ctrl-P

在命令行历史记录中调用上一条历史命令。

Ctrl-Q

恢复 (XON)。

终端恢复读入 *stdin*。

Ctrl-R

在命令行历史记录中进行搜索。

Ctrl-S

挂起 (XOFF)。

终端冻结 *stdin*。（可以使用 `Ctrl-Q` 恢复）

Ctrl-T

交换光标所在字符与其前一个字符。

Ctrl-U

删除光标所在字符之前的所有字符。

在一些情况下，不管光标在哪个位置，`Ctrl-U` 都会删除整行文字。

Ctrl-V

输入时，使用 `Ctrl-V` 允许插入控制字符。例如，下面两条语句是等

价的：

```
1. echo -e '\x0a'
2. echo <Ctrl-V><Ctrl-J>
```

`Ctrl-V` 在文本编辑器中特别有用。

Ctrl-W

当你在终端或 *xterm* 窗口中输入字符时，`Ctrl-W` 将会删除光标所在字符之前到其最近的空白符之间的所有字符。

在一些情况下，`Ctrl-W` 会删除到之前最近的非字母或数字的字符。

Ctrl-X

在一些特定的文本处理程序中，剪切高亮文本并复制到剪贴板（clipboard）。

Ctrl-Y

粘贴之前使用 `Ctrl-U` 或 `Ctrl-W` 删除的文字。

Ctrl-Z

暂停当前运行的任务。

在一些特定的文本处理程序中是替代操作。

在 MSDOS 文件系统中作为 `EOF`（end-of-file，文件终止标记）。

空白符

作为命令或变量之间的分隔符。空白符包含空格、制表符、换行符或它

们的任意组合。⁹在一些地方，比如变量赋值时，空白符不应该出现，否则会造成语法错误。

空白行在脚本中不会有任何实际作用，但是可以划分代码，使代码更具可读性。

特殊变量 `$IFS` 是作为一些特定命令的输入域 (field) 分隔符，默认值为空白符。

定义：域是字符串中离散的数据块。使用空白符或者指定的字符（通常由 `$IFS` 决定）来分隔临近域。在一些情况下，域也可以被称作记录 (record)。

如果想在字符串或者变量中保留空白符，请引用。

UNIX 过滤器可以使用 POSIX 字符类 `[]` 来寻找和操作空白符。

[^1]：操作符 (operator) 用来执行表达式 (operation)。最常见的例子就是算术运算符 + - * /。在 Bash 中，操作符和关键字的概念有一些重叠。

[^2]：它更被人熟知的名字是三元 (ternary) 操作符。但是读起来不清晰，而且容易令人混淆。trinary 是一种更加优雅的写法。

[^3]：美国信息交换标准代码 (American Standard Code for Information Interchange)。这是一套可以由计算机存储和处理的 7 位 (bit) 字符 (包含字母、数字和一系列有限的符号) 编码系统。

[^4]：进程标识符 (PID)，是分配给正在运行进程的唯一数字标识。可以使用 `ps` 命令查看进程的 PID。

定义：进程是正在执行的命令或程序，通常也称作任务。

[^6]：例外：作为管道的一部分的大括号中的代码块可能会运行在子进程中。


```
1. ls | { read firstline; read secondline; }  
# 错误。大括号中的代码块在子进程中运行，  
#+ 因此 “ls” 命令输出的结果不能传递到代码块中。  
echo "First line is $firstline; second line is  
$secondline" # 无效。  
  
# 感谢 S.C.
```

[^7]：正如在古代催情剂（philtre）被认为是一种能引发神奇变化的药剂一样，UNIX 中的过滤器（filter）也是有类似的作用的。

（如果一个程序员做出了一个能够在 Linux 设备上运行的 “love philtre”，那么他将会获得巨大的荣誉。）

[^8]：Bash将之前在命令行中执行过的命令存储在缓存（buffer）中，或者一块内存区域里。可以使用内建命令 `history` 来查看。

4. 变量与参数

- [第四章 变量与参数](#)
 - [本章目录](#)

第四章 变量与参数

本章目录

- [4.1 变量替换](#)
- [4.2 变量赋值](#)
- [4.3 Bash变量弱类型](#)
- [4.4 特殊变量类型](#)

变量 (variable) 在编程语言中用来表示数据。它本身只是一个标记，指向数据在计算机内存中的一个或一组地址。

变量通常出现在算术运算，数量操作及字符串解析中。

4.1 变量替换

- 4.1 变量替换
 - \$

4.1 变量替换

变量名是其所指向值的一个占位符 (placeholder)。引用变量值的过程我们称之为变量替换 (variable substitution)。

\$

接下来我们仔细区分一下变量名与变量值。如果变量名是

`variable1`，那么 `$variable1` 就是对变量值的引用。[^1]

```
1. bash$ variable1=23
2.
3.
4. bash$ echo variable1
5. variable1
6.
7. bash$ echo $variable1
8. 23
```

变量仅仅在声明时、赋值时、被删除时 (`unset`)、被导出时 (`export`)，算术运算中使用双括号结构 (`((...))`) 时或在代表信号时 (`signal`，查看样例 32-5) 才不需要有 `$` 前缀。赋值可以是使用 `=` (比如 `var1=27`)，可以是在 `read` 语句中，也可以是在循环的头部 (`for var2 in 1 2 3`)。

在双引号 `""` 字符串中可以使用变量替换。我们称之为部分引用，有时候也称弱引用。而使用单引号 `''` 引用时，变量只会作为字符串显示，

变量替换不会发生。我们称之为全引用，有时也称强引用。更多细节将在第五章讲解。

实际上，`$variable` 这种写法是 `${variable}` 的简化形式。在某些特殊情况下，使用 `$variable` 写法会造成语法错误，使用完整形式会更好（查看章节 10.2）。

样例 4-1. 变量赋值与替换

```

1.  #!/bin/bash
2.  # ex9.sh
3.
4.  # 变量赋值与替换
5.
6.  a=375
7.  hello=$a
8.  #   ^ ^
9.
10. #-----
11. # 初始化变量时，赋值号 = 的两侧绝不允许有空格出现。
12. # 如果有空格会发生什么？
13.
14. #   "VARIABLE =value"
15. #           ^
16. #% 脚本将会尝试运行带参数 "=value" 的 "VARIABLE " 命令。
17.
18. #   "VARIABLE= value"
19. #           ^
20. #% 脚本将会尝试运行 "value" 命令，
21. #+ 同时设置环境变量 "VARIABLE" 为 ""。
22. #-----
23.
24.
25. echo hello      # hello
26. # 没有引用变量，"hello" 只是一个字符串...
27.
28. echo $hello     # 375

```

```

29. # ^          这是变量引用。
30.
31. echo ${hello} # 375
32. #           与上面的类似，变量引用。
33.
34. # 字符串内引用变量
35. echo "$hello" # 375
36. echo "${hello}" # 375
37.
38. echo
39.
40. hello="A B C D"
41. echo $hello # A B C D
42. echo "$hello" # A B C D
43. # 正如我们所见，echo $hello 与 echo "$hello" 的结果不同。
44. # =====
45. # 字符串内引用变量将会保留变量的空白符。
46. # =====
47.
48. echo
49.
50. echo '$hello' # $hello
51. # ^ ^
52. # 单引号会禁用掉（转义）变量引用，这导致 "$" 将以普通字符形式被解析。
53.
54. # 注意单双引号字符串引用效果的不同。
55.
56. hello= # 将其设置为空值
57. echo "\$hello (null value) = $hello" # $hello (null value) =
58. # 注意
59. # 将一个变量设置为空与删除(unset)它不同，尽管它们的表现形式相同。
60.
61. # -----
62.
63. # 使用空白符分隔，可以在一行内对多个变量进行赋值。
64. # 但是这会降低程序的可读性，并且可能会导致部分程序不兼容的问题。
65.
66. var1=21 var2=22 var3=$V3

```

```

67. echo
68. echo "var1=$var1   var2=$var2   var3=$var3"
69.
70. # 在一些老版本的 shell 中这样写可能会有问题。
71.
72. # -----
73.
74. echo; echo
75.
76. numbers="one two three"
77. #           ^   ^
78. other_numbers="1 2 3"
79. #           ^   ^
80. # 如果变量中有空白符号, 那么必须用引号进行引用。
81. # other_numbers=1 2 3                # 出错
82. echo "numbers = $numbers"
83. echo "other_numbers = $other_numbers" # other_numbers = 1 2 3
84. # 也可以转义空白符。
85. mixed_bag=2\ ---\ Whatever
86. #           ^   ^ 使用 \ 转义空格
87.
88. echo "$mixed_bag"                # 2 --- Whatever
89.
90. echo; echo
91.
92. echo "uninitialized_variable = $uninitialized_variable"
93. # 未初始化的变量是空值(null表示不含有任何值)。
94. uninitialized_variable= # 只声明而不初始化, 等同于设为空值。
95. echo "uninitialized_variable = $uninitialized_variable" # 仍旧为空
96.
97. uninitialized_variable=23        # 设置变量
98. unset uninitialized_variable     # 删除变量
99. echo "uninitialized_variable = $uninitialized_variable"
100.                                # uninitialized_variable =
101.                                # 变量值为空
102. echo
103.
104. exit 0

```



一个未被赋值或未初始化的变量拥有空值 (*null value*)。注意: *null*值不等同于0。

```
1. if [ -z "$unassigned" ]
2. then
3.     echo "\$unassigned is NULL."
4. fi      # $unassigned is NULL.
```

在赋值前使用变量可能会导致错误。但在算术运算中使用未赋值变量是可行的。

```
1. echo "$uninitialized"           # 空行
2. let "uninitialized += 5"        # 加5
3. echo "$uninitialized"           # 5
4. # 结论:
5. # 一个未初始化的变量不含值(null), 但在算术运算中会被作为0处理。
```

也可参考样例 15-23。

[^1]: 实际上, 变量名是被称作左值 (lvalue), 意思是出现在赋值表达式的左侧的值, 比如 `VARIABLE=23`。变量值被称作右值 (rvalue), 意思是出现在赋值表达式右侧的值, 比如 `VAR2=$VARIABLE`。

事实上, 变量名只是一个引用, 一枚指针, 指向实际存储数据内存地址的指针。

4.2 变量赋值

- 4.2 变量赋值

- =

4.2 变量赋值

=

赋值操作符（在其前后没有空白符）。



不要混淆 = 与 -eq，后者用来进行比较而非赋值。

同时也要注意 = 根据使用场景既可作赋值操作符，也可作比较操作符。

样例 4-2. 变量赋值

```

1.  #!/bin/bash
2.  # 非引用形式变量
3.
4.  echo
5.
6.  # 什么时候变量是非引用形式，即变量名前没有 '$' 符号的呢？
7.  # 当变量在被赋值而不是被引用时。
8.
9.  # 赋值
10. a=879
11. echo "The value of \"a\" is $a."
12.
13. # 使用 'let' 进行赋值
14. let a=16+5
15. echo "The value of \"a\" is now $a."
16.
17. echo
18.
```



```

19. # 在 'for' 循环中赋值（隐式赋值）
20. echo -n "Values of \"a\" in the loop are: "
21. for a in 7 8 9 11
22. do
23.     echo -n "$a "
24. done
25.
26. echo
27. echo
28.
29. # 在 'read' 表达式中（另一种赋值形式）
30. echo -n "Enter \"a\" "
31. read a
32. echo "The value of \"a\" is now $a."
33.
34. echo
35.
36. exit 0

```

样例 4-3. 奇妙的变量赋值

```

1. #!/bin/bash
2.
3. a=23                # 简单形式
4. echo $a
5. b=$a
6. echo $b
7.
8. # 我们来玩点炫的（命令替换）。
9.
10. a=`echo Hello!`    # 将 'echo' 命令的结果赋值给 'a'
11. echo $a
12. # 注意在命令替换结构中包含感叹号(!)在命令行中使用将会失效,
13. #+ 因为它将会触发 Bash 的历史(history)机制。
14. # 在shell脚本内, Bash 的历史机制默认关闭。
15.
16. a=`ls -l`          # 将 'ls -l' 命令的结果赋值给 'a'
17. echo $a            # 不带引号引用, 将会移除所有的制表符与分行符

```

```
18. echo
19. echo "$a"          # 引号引用变量将会保留空白符
20.                   # 查看 "引用" 章节。
21.
22. exit 0
```

使用 `$(...)` 形式进行赋值（与反引号不同的新形式），与命令替换形式相似。

```
1. # 摘自 /etc/rc.d/rc.local
2. R=$(cat /etc/redhat-release)
3. arch=$(uname -m)
```

4.3 Bash弱类型变量

- 4.3 Bash变量是弱类型的

4.3 Bash变量是弱类型的

不同于许多其他编程语言，Bash 并不区分变量的类型。本质上说，*Bash* 变量是字符串，但在某些情况下，Bash 允许对变量进行算术运算和比较。决定因素则是变量值是否只含有数字。

样例 4-4. 整数还是字符串？

```

1.  #!/bin/bash
2.  # int-or-string.sh
3.
4.  a=2334                                # 整数。
5.  let "a += 1"
6.  echo "a = $a "                        # a = 2335
7.  echo                                  # 依旧是整数。
8.
9.
10. b=${a/23/BB}                          # 将 "23" 替换为 "BB"。
11.                                       # $b 变成了字符串。
12. echo "b = $b"                         # b = BB35
13. declare -i b                          # 将其声明为整数并没有什么卵用。
14. echo "b = $b"                         # b = BB35
15.
16. let "b += 1"                          # BB35 + 1
17. echo "b = $b"                         # b = 1
18. echo                                  # Bash 认为字符串的"整数值"为0。
19.
20. c=BB34
21. echo "c = $c"                         # c = BB34
22. d=${c/BB/23}                          # 将 "BB" 替换为 "23"。
23.                                       # $d 变为了一个整数。
24. echo "d = $d"                         # d = 2334

```

```

25. let "d += 1"           # 2334 + 1
26. echo "d = $d"         # d = 2335
27. echo
28.
29.
30. # 如果是空值会怎样呢？
31. e=''                  # ...也可以是 e="" 或 e=
32. echo "e = $e"         # e =
33. let "e += 1"          # 空值是否允许进行算术运算？
34. echo "e = $e"         # e = 1
35. echo                  # 空值变为了一个整数。
36.
37. # 如果时未声明的变量呢？
38. echo "f = $f"         # f =
39. let "f += 1"          # 是否允许进行算术运算？
40. echo "f = $f"         # f = 1
41. echo                  # 未声明变量变为了一个整数。
42. #
43. # 然而.....
44. let "f /= $undekl_var" # 可以除以0么？
45. # let: f /= : syntax error: operand expected (error token is " ")
46. # 语法错误！在这里 $undekl_var 并没有被设置为0！
47. #
48. # 但是，仍旧.....
49. let "f /= 0"
50. # let: f /= 0: division by 0 (error token is "0")
51. # 预期之中。
52.
53.
54. # 在执行算术运算时，Bash 通常将其空值的整数值设为0。
55. # 但是不要做这种事情！
56. # 因为这可能会导致一些意外的后果。
57.
58.
59. # 结论：上面的结果都表明 Bash 中的变量是弱类型的。
60.
61. exit $?

```

弱类型变量有利有弊。它可以使编程更加灵活、更加容易（给你足够的想象空间）。但它也同样的容易造成一些小错误，容易养成粗心大意的编程习惯。

为了减轻脚本持续跟踪变量类型的负担，Bash 不允许变量声明。

4.4 特殊变量类型

- 4.4 特殊的变量类型
 - 局部变量
 - 环境变量
 - 位置参数

4.4 特殊的变量类型

局部变量

仅在代码块或函数中才可见的变量（参考函数章节的局部变量部分）。

环境变量

会影响用户及shell行为的变量。



一般情况下，每一个进程都有自己的“环境”（*environment*），也就是一组该进程可以访问到的变量。从这个意义上来说，*shell*表现出与其他进程一样的行为。

每当*shell*启动时，都会创建出与其环境对应的*shell*环境变量。改变或增加*shell*环境变量会使*shell*更新其自身的环境。子进程（由父进程执行产生）会继承父进程的环境变量。



分配给环境变量的空间是有限的。创建过多环境变量或占用空间过大的环境变量有可能会造成问题。

```
1. bash$ eval "`seq 10000 | sed -e 's/.*/export
   var&=ZZZZZZZZZZZZZZZZ/'`"
2.
3. bash$ du
4. bash: /usr/bin/du: Argument list too long
```

注意，上面的“错误”已经在Linux内核版本号为2.6.23的系统中修复了。

（感谢 *Stéphane Chazelas* 对此问题的解释并提供了上面的例子。）

如果在脚本中设置了环境变量，那么这些环境变量需要被“导出”，也就是通知脚本所在的环境做出相应的更新。这个“导出”操作就是

`export` 命令。



脚本只能将变量导出到子进程，即在这个脚本中所调用的命令或程序。在命令行中调用的脚本不能够将变量回传给命令行环境，即子进程不能将变量回传给父进程。

定义：子进程（*child process*）是由另一个进程，即其父进程（*parent process*）所启动的子程序。

位置参数

从命令行中传递给脚本的参数¹：`$0, $1, $2, $3 ...`

即命令行参数。

`$0` 代表脚本名称，`$1` 代表第一个参数，`$2` 代表第二个，`$3` 代表第三个，以此类推^[^2]。在 `$9` 之后的参数必须被包含在大括号中，如 `${10}, ${11}, ${12}`。

特殊变量 `$*` 与 `$@` 代表所有位置参数。

样例 4-5. 位置参数

```
1. #!/bin/bash
2.
3. # 调用脚本时使用至少10个参数，例如
4. # ./scriptname 1 2 3 4 5 6 7 8 9 10
5. MINPARAMS=10
6.
7. echo
8.
9. echo "The name of this script is \"$0\"."
10. # 附带 ./ 代表当前目录
11. echo "The name of this script is "`basename $0`"."
12. # 除去路径信息（查看 'basename'）
13.
```

```
14. echo
15.
16. if [ -n "$1" ]           # 测试变量是否存在
17. then
18.     echo "Parameter #1 is $1" # 使用引号转义#
19. fi
20.
21. if [ -n "$2" ]
22. then
23.     echo "Parameter #2 is $2"
24. fi
25.
26. if [ -n "$3" ]
27. then
28.     echo "Parameter #3 is $3"
29. fi
30.
31. # ...
32.
33. if [ -n "${10}" ] # 大于 $9 的参数必须被放在大括号中
34. then
35.     echo "Parameter #10 is ${10}"
36. fi
37.
38. echo "-----"
39. echo "All the command-line parameters are: "$*"
40.
41. if [ $# -lt "$MINPARAMS" ]
42. then
43.     echo
44.     echo "This script needs at least $MINPARAMS command-line
        arguments!"
45. fi
46.
47. echo
48.
49. exit 0
```


在位置参数中使用大括号助记符提供了一种非常简单的方式来访问传入脚本的最后一个参数。在其中会使用到间接引用。

```
1. args=$#           # 传入参数的个数
2. lastarg=${!args}
3. # 这是 $args 的一种间接引用方式
4.
5. # 也可以使用：   lastarg=${!#}           (感谢 Chris Monson.)
6. # 这是 $# 的一种间接引用方式。
7. # 注意 lastarg=${!$#} 是无效的。
```

一些脚本能够根据调用时文件名的不同来执行不同的操作。要达到这样的效果，脚本需要检测 `$0`，也就是调用时的文件名^[^3]。同时，也必须存在指向这个脚本所有别名的符号链接文件（symbolic links）。详情查看样例 16-2。



如果一个脚本需要一个命令行参数但是在调用的时候却没有传入，那么这将会造成一个空变量赋值。这通常不是我们想要的。一种避免的方法是，在使用期望的位置参数时候，在赋值语句两侧添加一个额外的字符。

```
1. variable1_=$1_ # 而不是 variable1=$1
2. # 使用这种方法可以在没有位置参数的情况下避免产生错误。
3.
4. critical_argument01=$variable1_
5.
6. # 多余的字符可以被去掉，就像下面这样：
7. variable1=${variable1_/_/_}
8. # 仅仅当 $variable1_ 是以下划线开头时候才会有一些副作用。
9. # 这里使用了我们稍后会介绍的参数替换模板中的一种。
10. # （将替换模式设为空等价于删除。）
11.
12. # 更直接的处理方法就是先检测预期的位置参数是否被传入。
13. if [ -z $1 ]
14. then
15.     exit $E_MISSING_POS_PARAM
16. fi
```

```

17.
18.
19. # 但是, 正如 Fabian Kreutz 指出的,
20. #+ 上面的方法会有一些意想不到的副作用。
21. # 更好的方法是使用参数替换:
22. #      ${1:-$DefaultVal}
23. # 详情查看第十章“操作变量”的第二节“变量替换”。

```

样例 4-6. *wh*, *whois* 域名查询

```

1. #!/bin/bash
2. # ex18.sh
3.
4. # 在下面三个可选的服务器中进行 whois 域名查询:
5. # ripe.net, cw.net, radb.net
6.
7. # 将这个脚本重命名为 'wh' 后放在 /usr/local/bin 目录下
8.
9. # 这个脚本需要进行符号链接:
10. # ln -s /usr/local/bin/wh /usr/local/bin/wh-ripe
11. # ln -s /usr/local/bin/wh /usr/local/bin/wh-apnic
12. # ln -s /usr/local/bin/wh /usr/local/bin/wh-tucows
13.
14. E_NOARGS=75
15.
16.
17. if [ -z "$1" ]
18. then
19.     echo "Usage: `basename $0` [domain-name]"
20.     exit $E_NOARGS
21. fi
22.
23. # 检查脚本名, 访问对应服务器进行查询。
24. case `basename $0` in      # 也可以写:    case ${0##*/} in
25.     "wh"                ) whois $1@whois.tucows.com;;
26.     "wh-ripe"           ) whois $1@whois.ripe.net;;
27.     "wh-apnic"          ) whois $1@whois.apnic.net;;
28.     "wh-cw"             ) whois $1@whois.cw.net;;

```

```

29.         *           ) echo "Usage: `basename $0` [domain-name]";;
30.     esac
31.
32.     exit $?

```

使用 `shift` 命令可以将全体位置参数向左移一位，重新赋值。

`$1 <--- $2` , `$2 <--- $3` , `$3 <--- $4` , 以此类推。

原先的 `$1` 将会消失，而 `$0`（脚本名称）不会有任何改变。如果你在脚本中使用了大量的位置参数，`shift` 可以让你不使用{大括号}助记法也可以访问超过10个的位置参数。

样例 4-7. 使用 `shift` 命令

```

1.  #!/bin/bash
2.  # shft.sh: 使用 `shift` 命令步进访问所有的位置参数。
3.
4.  # 将这个脚本命名为 shft.sh, 然后在调用时跟上一些参数。
5.  # 例如：
6.  #   sh shft.sh a b c def 83 barndoor
7.
8.  until [ -z "$1" ] # 直到访问完所有的参数
9.  do
10.     echo -n "$1 "
11.     shift
12. done
13.
14. echo           # 换行。
15.
16. # 那些被访问完的参数又会怎样呢？
17. echo "$2"
18. # 什么都不会被打印出来。
19. # 当 $2 被移动到 $1 且没有 $3 时，$2 将会保持空。
20. # 因此 shift 是移动参数而非复制参数。
21.
22. exit

```

```

23.
24. # 可以参考 echo-params.sh 脚本, 在不使用 shift 命令的情况下,
25. #+ 步进访问所有位置参数。

```

`shift` 命令也可以带一个参数来指明一次移动多少位。

```

1. #!/bin/bash
2. # shift-past.sh
3.
4. shift 3      # 移动3位。
5. # 与 n=3; shift $n 效果相同。
6.
7. echo "$1"
8.
9. exit 0
10.
11. # ===== #
12.
13.
14. $ sh shift-past.sh 1 2 3 4 5
15. 4
16.
17. # 但是就像 Eleni Fragkiadaki 指出的那样,
18. # 如果尝试将位置参数 ($#) 传给 'shift',
19. # 将会导致脚本错误的结束, 同时位置参数也不会发送改变。
20. # 这也许是因为陷入了一个死循环...
21. # 比如:
22. #     until [ -z "$1" ]
23. #     do
24. #         echo -n "$1 "
25. #         shift 20      # 如果少于20个位置参数,
26. #         done          #+ 那么循环将永远不会结束。
27. #
28. # 当你不确定是否有这么多的参数时, 你可以加入一个测试:
29. #     shift 20 || break
30. #         ^^^^^^^^

```



使用 `shift` 命令同给函数传参相类似。详情查看样例 36-18。

[^2]：是调用脚本的进程设置了 `$0` 参数。就是脚本的文件名。详情可以查看 `execv` 的使用手册。

在命令行中，`$0` 是shell的名称。

```
1. bash$ echo $0
bash

tcsch% echo $0
tcsch
```

[^3]：如果脚本被引用（`sourced`）执行或者被链接（`symlinked`）执行时会失效。安全的方法是检测变量 `$BASH_Source`。

5. 引用

- [第五章 引用](#)
 - [本章目录](#)

第五章 引用

本章目录

- [5.1 引用变量](#)
- [5.2 转义](#)

引用就是将一个字符串用引号括起来。这样做是为了保护Shell/Shell脚本中被重新解释过或带扩展功能的[特殊字符](#)（如果一个字符带有其特殊意义而不仅仅是字面量的话，这个字符就能称为“特殊字符”。比如星号“*”就能表示[正则表达式](#)中的一个[通配符](#)）。

```
1. bash$ ls -l [Vv]*
2. -rw-rw-r-- 1 bozo bozo 324 Apr 2 15:05 VIEWDATA.BAT
3. -rw-rw-r-- 1 bozo bozo 507 May 4 14:25 vartrace.sh
4. -rw-rw-r-- 1 bozo bozo 539 Apr 14 17:11 viewdata.sh
5.
6. bash$ ls -l '[Vv]*'
7. ls: [Vv]*: No such file or directory
```

可以看到，提示不存在该文件。这里的 `'[Vv]*'` 被当成了文件名。

在日常沟通和写作中，当我们引用一个短语的时候，我们会将它单独隔开并赋予它特殊的意义，而在bash脚本中，当我们引用一个字符串，意味着保留它的字面量。

很多程序和公用代码会展开被引用字符串中的特殊字符。引用的一个重用用途是保护Shell中的命令行参数，但仍然允许调用的程序扩展它。

```
1. bash$ grep '[Ff]irst' *.txt
```

2. file1.txt:This is the first line of file1.txt.
3. file2.txt:This is the First line of file2.txt.

在所有.txt文件中找出包含*first*或者*First*字符串的行

注意，不加引号的 `grep [Ff]irst *.txt` 在Bash下也同样有效。
`[^1]`

引用也可以控制echo命令的断行符。

1. bash\$ echo \$(ls -l)
2. total 8 -rw-rw-r-- 1 bo bo 13 Aug 21 12:57 t.sh -rw-rw-r-- 1 bo bo
78 Aug 21 12:57 u.sh
- 3.
- 4.
5. bash\$ echo "\$(ls -l)"
6. total 8
7. -rw-rw-r-- 1 bo bo 13 Aug 21 12:57 t.sh
8. -rw-rw-r-- 1 bo bo 78 Aug 21 12:57 u.sh

`[^1]`：前提是当前目录下有文件名为First或first的文件。这也是使用引用的另一个原因。（感谢 Harald Koenig 指出了这一点）

5.1 引用变量

- 5.1 引用变量

5.1 引用变量

引用变量时，通常建议将变量包含在双引号中。因为这样可以防止除

`$`，```（反引号）和`\`（转义符）之外的其他特殊字符被重新解释。`[^1]`在双引号中仍然可以使用`$`引用变量（`"$variable"`），也就是将变量名替换为变量值（详情查看样例 4-1）。

使用双引号可以防止字符串被分割。`[^2]`即使参数中拥有很多空白分隔符，被包在双引号中后依旧是算作单一字符。

```

1. List="one two three"
2.
3. for a in $List      # 空白符将变量分成几个部分。
4. do
5.     echo "$a"
6. done
7. # one
8. # two
9. # three
10.
11. echo "---"
12.
13. for a in "$List"    # 在单一变量中保留所有空格。
14. do #      ^      ^
15.     echo "$a"
16. done
17. # one two three

```

下面是一个更加复杂的例子：


```

1. variable1="a variable containing five words"
2. COMMAND This is $variable1      # 带上7个参数执行COMMAND命令：
3. # "This" "is" "a" "variable" "containing" "five" "words"
4.
5. COMMAND "This is $variable1"    # 带上1个参数执行COMMAND命令：
6. # "This is a variable containing five words"
7.
8.
9. variable2=""                  # 空值。
10.
11. COMMAND $variable2 $variable2 $variable2
12.                             # 不带参数执行COMMAND命令。
13. COMMAND "$variable2" "$variable2" "$variable2"
14.                             # 带上3个参数执行COMMAND命令。
15. COMMAND "$variable2 $variable2 $variable2"
16.                             # 带上1个参数执行COMMAND命令（2空格）。
17.
18. # 感谢 Stéphane Chazelas。

```



当字符分割或者保留空白符出现问题时，才需要在 `echo` 语句中用双引号包住参数。

样例 5-1. 输出一些奇怪的变量

```

1. #!/bin/bash
2. # weirdvars.sh: 输出一些奇怪的变量
3.
4. echo
5.
6. var="'([\\}\\$\"'"
7. echo $var          # '([\\}]$'
8. echo "$var"        # '([\\}]$'    没有任何区别。
9.
10. echo
11.
12. IFS='\'
13. echo $var          # '([ }$'    \ 被转换成了空格，为什么？
14. echo "$var"        # '([\\}]$'

```

```

15.
16. # 上面的例子由 Stephane Chazelas 提供。
17.
18. echo
19.
20. var2="\\"
21. echo $var2      # "
22. echo "$var2"    # "\"
23. echo
24. # 但是...var2="\\" 不是合法的语句，为什么？
25. var3='\\'
26. echo "$var3"    # \\
27. # 强引用是可以的。
28.
29.
30. # ***** #
31. # 就像第一个例子展示的那样，嵌套引用是允许的。
32.
33. echo "$(echo '')"      # "
34. #   ^               ^
35.
36.
37. # 在有些时候这种方法非常有用。
38.
39. var1="Two bits"
40. echo "\$var1 = \"$var1\""      # $var1 = Two bits
41. #   ^               ^
42.
43. # 或者，可以像 Chris Hiestand 指出的那样：
44.
45. if [[ "$(du "$My_File1")" -gt "$(du "$My_File2")" ]]
46. #   ^   ^           ^ ^   ^   ^           ^ ^
47. then
48.   ...
49. fi
50. # ***** #

```

单引号 (' ') 与双引号类似，但是在单引号中不能引用变量，因为

`$` 不再具有特殊含义。在单引号中，除 `'` 之外的所有特殊字符都将会被直接按照字面意思解释。可以认为单引号（“全引用”）是双引号（“部分引用”）的一种更严格的形式。



因为在单引号中转义符（\）都已经按照字面意思解释了，因此尝试在单引号中包含单引号将不会产生你所预期的结果。

```
1. echo "Why can't I write 's between single quotes"
2.
3. echo
4.
5. # 可以采取迂回的方式。
6. echo 'Why can\'\'t I write \''s between single quotes'
7. #      |-----|  |-----|  |-----|
8. # 由三个单引号引用的字符串，再加上转义以及双引号包住的单引号组成。
9.
10. # 感谢 Stéphane Chazelas 提供的例子。
```

[^1]：在命令行里，如果双引号包含了 “!” 将会产生错误。这是因为shell将其解释为查看历史命令。而在脚本中，因为历史机制已经被关闭，所以不会产生这个问题。

我们更加需要注意的是在双引号中 `\` 的反常行为，尤其是在使用 `echo -e` 命令时。

```
1. bash$ echo hello\!
hello!
bash$ echo "hello\!"
hello\!

bash$ echo \
>
bash$ echo "\
>
bash$ echo \a
a
```

```
bash$ echo "\a"
\a

bash$ echo x\tty
xty
bash$ echo "x\tty"
x\tty

bash$ echo -e x\tty
xty
bash$ echo -e "x\tty"
x      y
```

在 `echo` 后的双引号中一般会转义 `\`。并且 `echo -e` 会将 `"\t"` 解释成制表符。

（感谢 Wayne Pollock 提出这些；感谢 Geoff Lee 与 Daniel Barclay 对此做出的解释。）

[^2]：字符分割（word splitting）在本文中的意思是指将一个字符串分割成独立的、离散的变量。

5.2 转义

- 5.2 转义

- `\n`
- `\r`
- `\t`
- `\v`
- `\b`
- `\a`
- `\0xx`
- `\"`
- `\$`
- `\\`

5.2 转义

转义是一种引用单字符的方法。通过在特殊字符前加上转义符 `\` 来告诉shell按照字面意思去解释这个字符。



需要注意的是，在一些特定的命令和工具，比如 `echo` 和 `sed` 中，转义字符通常会起到相反的效果，即可能会使得那些字符产生特殊含义。

在 `echo` 与 `sed` 命令中，转义字符的特殊含义

`\n`

换行 (line feed)。

`\r`

回车 (carriage return)。

\t

水平制表符。

\v

垂直制表符。

\b

退格。

\a

警报、响铃或闪烁。

\0xx

ASCII码的八进制形式，等价于 `\0nn`，其中 `nn` 是数字。



在 `$' ... '` 字符串扩展结构中可以通过转义八进制或十六进制的ASCII码形式给变量赋值，比如 `quote=$'\042'`。

样例 5-2. 转义字符

```

1.  #!/bin/bash
2.  # escaped.sh: 转义字符
3.
4.  #####
5.  ### 首先让我们先看一下转义字符的基本用法。 ###
6.  #####
7.
8.  # 转义新的一行。
9.  # -----
10.
```

```

11. echo ""
12.
13. echo "This will print
14. as two lines."
15. # This will print
16. # as two lines.
17.
18. echo "This will print \
19. as one line."
20. # This will print as one line.
21.
22. echo; echo
23.
24. echo "====="
25.
26.
27. echo "\v\v\v\v"      # 按字面意思打印 \v\v\v\v
28. # 使用 echo 命令的 -e 选项来打印转义字符。
29. echo "====="
30. echo "VERTICAL TABS"
31. echo -e "\v\v\v\v"   # 打印四个垂直制表符。
32. echo "====="
33.
34. echo "QUOTATION MARK"
35. echo -e "\042"       # 打印 "（引号，八进制ASCII码为42）。
36. echo "====="
37.
38.
39.
40. # 使用 '$\x' 这样的形式后可以不需要加 -e 选项。
41.
42. echo; echo "NEWLINE and (maybe) BEEP"
43. echo $\n'           # 新的一行。
44. echo $\a'           # 警报（响铃）。
45.                     # 根据不同的终端版本，也可能是闪屏。
46.
47. # 我们之前介绍了 '$\nnn' 字符串扩展，而现在我们要看到的是...
48.

```

```

49. # ===== #
50. # 自 Bash 第二个版本开始的 '$\nnn' 字符串扩展结构。
51. # ===== #
52.
53. echo "Introducing the \$\ ' ... \' string-expansion construct . . .
    "
54. echo ". . . featuring more quotation marks."
55.
56. echo $\t \042 \t'    # 在制表符之间的引号。
57. # 需要注意的是 '\nnn' 是一个八进制的值。
58.
59. # 字符串扩展同样适用于十六进制的值，格式是 '$\xhhh'。
60. echo $\t \x22 \t'    # 在制表符之间的引号。
61. # 感谢 Greg Keraunen 指出这些。
62. # 在早期的 Bash 版本中允许使用 '\x022' 这样的形式。
63.
64. echo
65.
66.
67. # 将 ASCII 码字符赋值给变量。
68. # -----
69. quote=$'\042'        # 将 " 赋值给变量。
70. echo "$quote Quoted string $quote and this lies outside the
    quotes."
71.
72. echo
73.
74. # 连接多个 ASCII 码字符给变量。
75. triple_underline=$'\137\137\137' # 137是 '_' ASCII码的八进制形式
76. echo "$triple_underline UNDERLINE $triple_underline"
77.
78. echo
79.
80. ABC=$'\101\102\103\010'          # 101, 102, 103是 A, B, C
81.                                  # ASCII码的八进制形式。
82. echo $ABC
83.
84. echo

```



```

85.
86. escape=$'\033'                # 033 是 ESC 的八进制形式
87. echo "\"escape\" echoes an $escape"
88.                                # 没有可见输出
89.
90. echo
91.
92. exit 0

```

下面是一个更加复杂的例子：

样例 5-3. 检测键盘输入

```

1.  #!/bin/bash
2.  # 作者：Sigurd Solaas, 作于2011年4月20日
3.  # 授权在《高级Bash脚本编程指南》中使用。
4.  # 需要 Bash 版本高于4.2。
5.
6.  key="no value yet"
7.  while true; do
8.      clear
9.      echo "Bash Extra Keys Demo. Keys to try:"
10.     echo
11.     echo "* Insert, Delete, Home, End, Page_Up and Page_Down"
12.     echo "* The four arrow keys"
13.     echo "* Tab, enter, escape, and space key"
14.     echo "* The letter and number keys, etc."
15.     echo
16.     echo "    d = show date/time"
17.     echo "    q = quit"
18.     echo "===== "
19.     echo
20.
21.     # 将独立的Home键值转换为数字7上的Home键值：
22.     if [ "$key" = $'\x1b\x4f\x48' ]; then
23.         key=$'\x1b\x5b\x31\x7e'
24.         # 引用字符扩展结构。
25.     fi

```

```
26.  
27.  # 将独立的End键值转换为数字1上的End键值：  
28.  if [ "$key" = $\x1b\x4f\x46' ]; then  
29.      key=$'\x1b\x5b\x34\x7e'  
30.  fi  
31.  
32.  case "$key" in  
33.      $\x1b\x5b\x32\x7e') # 插入  
34.          echo Insert Key  
35.          ;;  
36.      $\x1b\x5b\x33\x7e') # 删除  
37.          echo Delete Key  
38.          ;;  
39.      $\x1b\x5b\x31\x7e') # 数字7上的Home键  
40.          echo Home Key  
41.          ;;  
42.      $\x1b\x5b\x34\x7e') # 数字1上的End键  
43.          echo End Key  
44.          ;;  
45.      $\x1b\x5b\x35\x7e') # 上翻页  
46.          echo Page_Up  
47.          ;;  
48.      $\x1b\x5b\x36\x7e') # 下翻页  
49.          echo Page_Down  
50.          ;;  
51.      $\x1b\x5b\x41') # 上箭头  
52.          echo Up arrow  
53.          ;;  
54.      $\x1b\x5b\x42') # 下箭头  
55.          echo Down arrow  
56.          ;;  
57.      $\x1b\x5b\x43') # 右箭头  
58.          echo Right arrow  
59.          ;;  
60.      $\x1b\x5b\x44') # 左箭头  
61.          echo Left arrow  
62.          ;;  
63.      $\x09') # 制表符
```

```

64.     echo Tab Key
65.     ;;
66.     $'\x0a') # 回车
67.     echo Enter Key
68.     ;;
69.     $'\x1b') # ESC
70.     echo Escape Key
71.     ;;
72.     $'\x20') # 空格
73.     echo Space Key
74.     ;;
75.     d)
76.     date
77.     ;;
78.     q)
79.     echo Time to quit...
80.     echo
81.     exit 0
82.     ;;
83.     *)
84.     echo Your pressed: \'"$key\"'
85.     ;;
86. esac
87.
88. echo
89. echo "=====
90.
91. unset K1 K2 K3
92. read -s -N1 -p "Press a key: "
93. K1="$REPLY"
94. read -s -N2 -t 0.001
95. K2="$REPLY"
96. read -s -N1 -t 0.001
97. K3="$REPLY"
98. key="$K1$K2$K3"
99.
100. done
101.

```

```
102. exit $?
```

还可以查看样例 37-1。

\"

转义引号，指代自身。

```
1. echo "Hello"           # Hello
2. echo "\"Hello\" ... he said." # "Hello" ... he said.
```

\\$

转义美元符号（跟在 `\\$` 后的变量名将不会被引用）。

```
1. echo "\$variable01"      # $variable01
2. echo "The book cost \$7.98." # The book cost $7.98.
```

\\

转义反斜杠，指代自身。

```
1. echo "\\" # 结果是 \
2.
3. # 然而...
4.
5. echo "\" # 在命令行中会出现第二行并提示输入。
6.          # 在脚本中会出错。
7.
8. # 但是...
9.
10. echo '\'
```



根据转义符所在的上下文（强引用、弱引用，命令替换或者在 *here document*）的不

同，它的行为也会有所不同。

```

1.                                     # 简单转义与引用
2. echo \z                            # z
3. echo \\z                          # \z
4. echo '\z'                          # \z
5. ehco '\\z'                        # \\z
6. echo "z"                          # \z
7. echo "\\z"                        # \z
8.
9.                                     # 命令替换
10. echo `echo \z`                   # z
11. echo `echo \\z`                  # z
12. echo `echo \\z`                  # \z
13. echo `echo \\z`                  # \z
14. echo `echo \\z`                  # \z
15. echo `echo \\z`                  # \z
16. echo `echo "z"`                  # \z
17. echo `echo "\\z"`                # \z
18.
19.                                     # Here Document
20. cat <<EOF
21. \z
22. EOF                              # \z
23.
24. cat <<EOF
25. \\z
26. EOF                              # \z
27.
28. # 以上例子由 Stéphane Chazelas 提供。

```

含有转义字符的字符串可以赋值给变量，但是仅仅将单一的转义符赋值给变量是不可行的。

```

1. variable=\
2. echo "$variable"
3. # 这样做会报如下错误：
4. # tesh.sh: : command not found
5. # 单独的转义符不能够赋值给变量。
6. #
7. # 事实上，"\ " 转义了换行，实际效果是：
8. #+ variable=echo "$variable"
9. #+ 这是一个非法的赋值方式。

```

```

10.
11. variable=\
12. 23skidoo
13. echo "$variable"           # 23skidoo
14.                             # 因为第二行是一个合法的赋值，因此不会报错。
15.
16. variable=\
17. #      \^      转义符后有一个空格
18. echo "$variable"           # 空格
19.
20. variable=\
21. echo "$variable"           # \
22.
23. variable=\\
24. echo "$variable"
25. # 这样做会报如下错误：
26. # tesh.sh: \: command not found
27. #
28. # 第一个转义符转义了第二个，但是第三个转义符仍旧转义的是换行，
29. #+ 跟开始的那个例子一样，因此会报错。
30.
31. variable=\\\
32. echo "$variable"           # \
33.                             # 第二个和第四个转义符被转义了，因此可行。

```

转义空格能够避免在命令参数列表中的字符分割问题。

```

1. file_list="/bin/cat /bin/gzip /bin/more /usr/bin/less
   /usr/bin/emacs-20.7"
2. # 将一系列文件作为命令的参数。
3.
4. # 增加两个文件到列表中，并且列出整个表。
5. ls -l /usr/X11R6/bin/xsetroot /sbin/dump $file_list
6.
7. echo "-----"
   -----"
8.
9. # 如果我们转义了这些空格会怎样？
10. ls -l /usr/X11R6/bin/xsetroot\ /sbin/dump\ $file_list
11. # 错误：因为转义了两个空格，因此前三个文件被连接成了一个参数传递给了 'ls -l'

```

转义符也提供一种可以撰写多行命令的方式。通常，每一行是一个命令，但是转义换行后命令就可以在下一行继续撰写。

```
1. (cd /source/directory && tar cf - . ) | \
2. (cd /dest/directory && tar xpvf -)
3. # 回顾 Alan Cox 的目录树拷贝命令，但是把它拆成了两行。
4.
5. # 或者你也可以：
6. tar cf - -C /source/directory . |
7. tar xpvf - -C /dest/directory
8. # 可以看下方的注释。
9. # （感谢 Stéphane Chazelas。）
```



在脚本中，如果以 “|” 管道作为一行的结束字符，那么不需要加转义符 \ 也可以写多行命令。但是一个好的编程习惯就是在写多行命令的事后，无论什么情况都要在行尾加上转义符 \。

```
1. echo "foo
2. bar"
3. #foo
4. #bar
5.
6. echo
7.
8. echo 'foo
9. bar'      # 没有区别。
10. #foo
11. #bar
12.
13. echo
14.
15. echo foo\
16. bar      # 转义换行。
17. #foobar
18.
19. echo
20.
```

```
21. echo "foo\  
22. bar"      # 没有区别，在弱引用中，\ 转义符仍旧转义了换行。  
23. #foobar  
24.  
25. echo  
26.  
27. echo 'foo\  
28. bar'      # 在强引用中，\ 就按照字面意思来解释了。  
29. #foo\  
30. #bar  
31.  
32. # 由 Stéphane Chazelas 提供的例子。
```


6. 退出与退出状态

- [第六章 退出与退出状态](#)

第六章 退出与退出状态

*Bourne shell*里存在不明确之处，但人们也会使用它们。

— *Chat Ramey*

跟C程序类似，`exit` 命令被用来结束脚本。同时，它也会返回一个值，返回值可以被交给父进程。

每个命令都会返回一个退出状态 (`exit status`)，有时也叫做返回状态 (`return status`) 或退出码 (`exit code`)。命令执行成功返回0，如果返回一个非0值，通常情况下会被认为是一个错误代码。一个运行状态良好的UNIX命令、程序和工具在正常执行退出后都会返回一个0的退出码，当然也有例外。

同样地，脚本中的函数和脚本本身也会返回一个退出状态。在脚本或者脚本函数中执行的最后的命令会决定它们的退出状态。在脚本中，`exit nnn` 命令将会把nnn退出状态码传递给shell (`nnn` 必须是 0-255 之间的整型数)。



当一个脚本以不带参数的 `exit` 来结束时，脚本的退出状态由脚本最后执行命令决定 (`exit` 命令之前)。

```
1. #!/bin/bash
2.
3. COMMAND_1
4.
5. ...
6.
7. COMMAND_LAST
```

```

8.
9.  # 将以最后的命令来决定退出状态
10.
11. exit

```

`exit` , `exit $?` 以及省略 `exit` 效果等同。

```

1.  #!/bin/bash
2.
3.  COMMAND_1
4.
5.  ...
6.
7.  COMMAND_LAST
8.
9.  #将以最后的命令来决定退出状态
10.
11. exit $?

```

```

1.  #!/bin/bash
2.
3.  COMMAND_1
4.
5.  ...
6.
7.  COMMAND_LAST
8.
9.  #将以最后的命令来决定退出状态

```

`$?` 读取上一个执行命令的退出状态。在一个函数返回后，`$?` 给出函数最后执行的那条命令的退出状态。这就是Bash函数的“返回值”。^{[^1](#)}

在管道执行后，`$?` 给出最后执行的那条命令的退出状态。

在脚本终止后，命令行下键入 `$?` 会给出脚本的退出状态，即在脚本中

最后一条命令执行后的退出状态。一般情况下，0为成功，1-255为失败。

样例 6-1. 退出与退出状态

```

1.  #!/bin/bash
2.
3.  echo hello
4.  echo $?      # 返回值为0，因为执行成功。
5.
6.  lskdf        # 不认识的命令。
7.  echo $?      # 返回非0值，因为失败了。
8.
9.  echo
10.
11. exit 113     # 将返回113给shell
12.              # 为了验证这些，在脚本结束的地方使用“echo $?”
13.
14. # 按照惯例，'exit 0' 意味着执行成功，
15. #+ 非0意味着错误或者异常情况。
16. # 查看附录章节“退出码的特殊含义”

```

`$?` 对于测试脚本中的命令的执行结果特别有用（查看样例 16-35 和样例 16-20）。



逻辑非操作符 `!` 将会反转测试或命令的结果，并且这将会影响退出状态。

样例 6-2. 否定一个条件使用 `!`

```

1. true        # true 是 shell 内建命令。
2. echo "exit status of \"true\" = $?"      # 0
3.
4. ! true
5. echo "exit status of \"! true\" = $?"      # 1
6. # 注意在命令之间的 "!" 需要一个空格。
7. # !true 将导致一个"command not found"错误。

```

```

8. #
9. # 如果一个命令以'!'开头，那么将调用 Bash 的历史机制，显示这个命令被使用的历史。
10.
11. true
12. !true
13. # 这次就没有错误了，但是同样也没有反转。
14. # 它不过是重复之前的命令（true）。
15.
16.
17. # ===== #
18. # 在 _pipe_ 前使用 ! 将改变返回的退出状态。
19. ls | bogus_command      #bash: bogus_command: command not found
20. echo $?                 #127
21. >
22. ! ls | bogus_command    #bash: bogus_command:command not found
23. echo $?                 #0
24. # 注意 ! 不会改变管道的执行。
25. # 只改变退出状态。
26. #===== #
27. >
28. # 感谢 Stéphane Chazelas 和 Kristopher Newsome。

```



某些特定的退出码具有一些特定的**保留含义**，用户不应该在自己的脚本中重新定义它们。

7. 测试

- [第七章 测试](#)
 - [本章目录](#)

第七章 测试

本章目录

- [7.1 测试结构](#)
- [7.2 文件测试操作](#)
- [7.3 其他比较操作](#)
- [7.4 嵌套 if/then 条件测试](#)
- [7.5 牛刀小试](#)

每一个完备的程序设计语言都可以对一个条件进行判断，然后根据判断结果执行相应的指令。Bash 拥有 `test` 命令，[双方括号](#)、[双圆括号](#) 测试操作符以及 `if/then` 测试结构。

7.1 测试结构

- 7.1 测试结构
 - Else if 与 elif

7.1 测试结构

- `if/then` 结构是用来检测一系列命令的 [退出状态](#) 是否为0（按 UNIX 惯例, 退出码 0 表示命令执行成功），如果为0，则执行接下来的一个或多个命令。
- 测试结构会使用一个特殊的命令 `[`（参看特殊字符章节 [左方括号](#)）。等同于 `test` 命令，它是一个[内建命令](#)，写法更加简洁高效。该命令将其参数视为比较表达式或文件测试，以比较结果作为其退出状态码返回（0 为真，1 为假）。
- Bash 在 2.02 版本中引入了扩展测试命令 `[[...]]`，它提供了一种与其他语言语法更为相似的方式进行比较操作。注意，`[[` 是一个 [关键字](#) 而非一个命令。

Bash 将 `[[$a -lt $b]]` 视为一整条语句，执行并返回退出状态。

- 结构 `((...))` 和 `let ...` 根据其执行的算术表达式的结果决定退出状态码。这样的 [算术扩展](#) 结构可以用来进行 [数值比较](#)。

```
1. (( 0 && 1 ))           # 逻辑与
2. echo $?               # 1      ***
3. # 然后 ...
4. let "num = (( 0 && 1 ))"
5. echo $num             # 0
6. # 然而 ...
```

```

7. let "num = (( 0 && 1 ))"
8. echo $?      # 1      ***
9.
10.
11. (( 200 || 11 ))      # 逻辑或
12. echo $?      # 0      ***
13. # ...
14. let "num = (( 200 || 11 ))"
15. echo $num    # 1
16. let "num = (( 200 || 11 ))"
17. echo $?      # 0      ***
18.
19.
20. (( 200 | 11 ))      # 按位或
21. echo $?      # 0      ***
22. # ...
23. let "num = (( 200 | 11 ))"
24. echo $num    # 203
25. let "num = (( 200 | 11 ))"
26. echo $?      # 0      ***
27.
28. # "let" 结构的退出状态与双括号算术扩展的退出状态是相同的。

```



注意，双括号算术扩展表达式的退出状态码不是一个错误的值。算术表达式为0，返回1；算术表达式不为0，返回0。

```

1. var=-2 && (( var+=2 ))
2. echo $?      # 1
3.
4. var=-2 && (( var+=2 )) && echo $var
5.              # 并不会输出 $var，因为((var+=2))的状态码为1

```

- `if` 不仅可以用来测试括号内的条件表达式，还可以用来测试其他任何命令。

```

1. if cmp a b &> /dev/null # 消去输出结果

```

```

2. then echo "Files a and b are identical."
3. else echo "Files a and b differ."
4. fi
5.
6. # 下面介绍一个非常实用的 "if-grep" 结构：
7. # -----
8. if grep -q Bash file
9. then echo "File contains at least one occurrence of Bash."
10. fi
11.
12. word=Linux
13. letter_sequence=inu
14. if echo "$word" | grep -q "$letter_sequence"
15. # 使用 -q 选项消去 grep 的输出结果
16. then
17.     echo "$letter_sequence found in \"$word\""
18. else
19.     echo "$letter_sequence not found in $word"
20. fi
21.
22.
23. if COMMAND_WHOSE_EXIT_STATUS_IS_0_UNLESS_ERROR_OCCURRED
24. then echo "Command succeed."
25. else echo "Command failed."
26. fi

```

- 感谢 Stéphane Chazelas 提供了后两个例子。

样例 7-1. 什么才是真？

```

1. #!/bin/bash
2.
3. # 提示：
4. # 如果你不确定某个表达式的布尔值，可以用 if 结构进行测试。
5.
6. echo
7.
8. echo "Testing \"0\""

```



```
9.  if [ 0 ]
10. then
11.     echo "0 is true."
12. else
13.     echo "0 is false."
14. fi          # 0 为真。
15.
16. echo
17.
18. echo "Testing \"1\""
19. if [ 1 ]
20. then
21.     echo "1 is true."
22. else
23.     echo "1 is false."
24. fi          # 1 为真。
25.
26. echo
27.
28. echo "Testing \"-1\""
29. if [ -1 ]
30. then
31.     echo "-1 is true."
32. else
33.     echo "-1 is false."
34. fi          # -1 为真。
35.
36. echo
37.
38. echo "Testing \"NULL\""
39. if [ ]          # NULL, 空
40. then
41.     echo "NULL is true."
42. else
43.     echo "NULL is false."
44. fi          # NULL 为假。
45.
46. echo
```

```

47.
48. echo "Testing \"xyz\""
49. if [ xyz ]      # 字符串
50. then
51.     echo "Random string is true."
52. else
53.     echo "Random string is false."
54. fi              # 随机字符串为真。
55.
56. echo
57.
58. echo "Testing \"$xyz\""
59. if [ $xyz ]     # 原意是测试 $xyz 是否为空，但是
60.                 # 现在 $xyz 只是一个没有初始化的变量。
61. then
62.     echo "Uninitialized variable is true."
63. else
64.     echo "Uninitialized variable is flase."
65. fi              # 未初始化变量含有null空值，为假。
66.
67. echo
68.
69. echo "Testing \"-n $xyz\""
70. if [ -n "$xyz" ]      # 更加准确的写法。
71. then
72.     echo "Uninitialized variable is true."
73. else
74.     echo "Uninitialized variable is false."
75. fi                  # 未初始化变量为假。
76.
77. echo
78.
79.
80. xyz=              # 初始化为空。
81.
82. echo "Testing \"-n $xyz\""
83. if [ -n "$xyz" ]
84. then

```

```

85.     echo "Null variable is true."
86. else
87.     echo "Null variable is false."
88. fi          # 空变量为假。
89.
90. echo
91.
92. # 什么时候 "false" 为真？
93.
94. echo "Testing \"false\""
95. if [ "false" ]          # 看起来 "false" 只是一个字符串
96. then
97.     echo "\"false\" is true." #+ 测试结果为真。
98. else
99.     echo "\"false\" is false."
100. fi          # "false" 为真。
101.
102. echo
103.
104. echo "Testing \"\$false\"" # 未初始化的变量。
105. if [ "$false" ]
106. then
107.     echo "\"\$false\" is true."
108. else
109.     echo "\"\$false\" is false."
110. fi          # "$false" 为假。
111.             # 得到了我们想要的结果。
112.
113. # 如果测试空变量 "$true" 会有什么样的结果？
114.
115. echo
116.
117. exit 0

```

练习：理解 样例 7-1

1. `if [condition-true]`
2. `then`

```

3.     command 1
4.     command 2
5.     ...
6. else # 如果测试条件为假, 则执行 else 后面的代码段
7.     command 3
8.     command 4
9.     ...
10. fi

```



如果把 `if` 和 `then` 写在同一行时, 则必须在 `if` 语句后加上一个分号来结束语句。因为 `if` 和 `then` 都是 **关键字**。以关键字 (或者命令) 开头的语句, 必须先结束该语句(分号;), 才能执行下一条语句。

```
1. if [ -x "$filename" ]; then
```

Else if 与 elif

elif

`elif` 是 `else if` 的缩写。可以把多个 `if/then` 语句连到外边去, 更加简洁明了。

```

1. if [ condition1 ]
2. then
3.     command1
4.     command2
5.     command3
6. elif [condition2 ]
7. # 等价于 else if
8. then
9.     command4
10.    command5
11. else
12.    default-command

```

13. `fi`

`if test condition-true` 完全等价于 `if [condition-true]`。当语句开始执行时，左括号 `[` 是作为调用 `test` 命令的标记^[^1]，而右括号则不严格要求，但在新版本的 Bash 里，右括号必须补上。



`test` 命令是 Bash 的 **内建命令**，可以用来检测文件类型和比较字符串。在 Bash 脚本中，`test` 不调用 `sh-utils` 包下的文件 `/usr/bin/test`。同样，`[` 也不会调用链接到 `/usr/bin/test` 的 `/usr/bin/[` 文件。

```
1. bash$ type test
2. test is a shell builtin
3. bash$ type '['
4. [ is a shell builtin
5. bash$ type '['
6. [[ is a shell keyword
7. bash$ type ']'
8. ]] is a shell keyword
9. bash$ type ']'
10. bash: type: ]: not found
```

如果你想在 Bash 脚本中使用 `/usr/bin/test`，那你必须把路径写全。

样例 7-2. `test`，`/usr/bin/test`，`[` 和 `/usr/bin/[[` 的等价性

```
1. #!/bin/bash
2.
3. echo
4.
5. if test -z "$1"
6. then
```

```

7.   echo "No command-line arguments."
8.   else
9.   echo "First command-line argument is $1."
10.  fi
11.
12.  echo
13.
14.  if /usr/bin/test -z "$1"      # 等价于内建命令 "test"
15.  #  ^^^^^^^^^^^^^^^          # 指定全路径
16.  then
17.   echo "No command-line arguments."
18.  else
19.   echo "First command-line argument is $1."
20.  fi
21.
22.  echo
23.
24.  if [ -z "$1" ]                # 功能和上面的代码相同。
25.  #   if [ -z "$1"             理论上可行, 但是 Bash 会提示缺失右括号
26.  then
27.   echo "No command-line arguments."
28.  else
29.   echo "First command-line argument is $1."
30.  fi
31.
32.  echo
33.
34.
35.  if /usr/bin/[ -z "$1" ]       # 功能和上面的代码相同。
36.  # if /usr/bin/[ -z "$1"       # 理论上可行, 但是会报错
37.  #                             # 已经在 Bash 3.x 版本被修复了
38.  then
39.   echo "No command-line arguments."
40.  else
41.   echo "First command-line argument is $1."
42.  fi
43.
44.  echo

```

```
45.
46. exit 0
```

在 Bash 里，`[[]]` 是比 `[]` 更加通用的写法。其作为扩展 `test` 命令从 ksh88 中被继承了过来。

在 `[[` 和 `]]` 中不会进行文件名扩展或字符串分割，但是可以进行参数扩展和命令替换。

```
1. file=/etc/passwd
2.
3. if [[ -e $file ]]
4. then
5.     echo "Password file exists."
6. fi
```

使用 `[[...]]` 代替 `[...]` 可以避免很多逻辑错误。比如可以在 `[[]]` 中使用 `&&`，`||`，`<` 和 `>` 操作符，而在 `[]` 中使用则会报错。

在 `[[]]` 中会自动执行八进制和十六进制的进制转换操作。

```
1. # [[ 八进制和十六进制进制转换 ]]
2. # 感谢 Moritz Gronbach 提出。
3.
4.
5. decimal=15
6. octal=017    # = 15 (十进制)
7. hex=0x0f    # = 15 (十进制)
8.
9. if [ "$decimal" -eq "$octal" ]
10. then
11.     echo "$decimal equals $octal"
12. else
13.     echo "$decimal is not equal to $octal"    # 15 不等于 017
14. fi      # 在单括号 [ ] 之间不会进行进制转换。
```

```

15.
16.
17. if [[ "$decimal" -eq "$octal" ]]
18. then
19.     echo "$decimal equals $octal"                # 15 等于 017
20. else
21.     echo "$decimal is not equal to $octal"
22. fi        # 在双括号 [[ ]] 之间会进行进制转换。
23.
24. if [[ "$decimal" -eq "$hex" ]]
25. then
26.     echo "$decimal equals $hex"                  # 15 等于 0x0f
27. else
28.     echo "$decimal is not equal to $hex"
29. fi        # 十六进制也可以进行转换。

```



语法上并不严格要求在 `if` 之后一定要写 `test` 命令或者测试结构 (`[]` 或 `[[]]`)。

```

1. dir=/home/bozo
2.
3. if cd "$dir" 2>/dev/null; then    # "2>/dev/null" 重定向消去错误输出。
4.     echo "Now in $dir."
5. else
6.     echo "Can't change to $dir."
7. fi

```

`if COMMAND` 的退出状态就是 `COMMAND` 的退出状态。

同样的，测试括号也不一定需要与 `if` 一起使用。其可以同 [列表结构](#) 结合而不需要 `if`。

```

1. var1=20
2. var2=22
3. [ "$var1" -ne "$var2" ] && echo "$var1 is not equal to $var2"
4.

```



```

5. home=/home/bozo
6. [ -d "$home" ] || echo "$home directory does not exist."

```

`(())` 结构 扩展和执行算术表达式。如果执行结果为0，其返回的退出状态码 为1（假）。非0表达式返回的退出状态为0（真）。这与上述所使用的 `test` 和 `[]` 结构形成鲜明的对比。

样例 7-3. 使用 `(())` 进行算术测试

```

1. #!/bin/bash
2. # arith-tests.sh
3. # 算术测试。
4.
5. # (( ... )) 结构执行并测试算术表达式。
6. # 与 [ ... ] 结构的退出状态正好相反。
7.
8. (( 0 ))
9. echo "Exit status of \"(( 0 ))\" is $?."          # 1
10.
11. (( 1 ))
12. echo "Exit status of \"(( 1 ))\" is $?."          # 0
13.
14. (( 5 > 4 ))
15. echo "Exit status of \"(( 5 > 4 ))\" is $?."      # 真
16.
17. (( 5 > 9 ))
18. echo "Exit status of \"(( 5 > 9 ))\" is $?."      # 1
19.
20. (( 5 == 5 ))
21. echo "Exit status of \"(( 5 == 5 ))\" is $?."      # 0
22. # (( 5 = 5 )) 会报错。
23.
24. (( 5 - 5 ))
25. echo "Exit status of \"(( 5 - 5 ))\" is $?."      # 1
26.
27. (( 5 / 4 ))
28. echo "Exit status of \"(( 5 / 4 ))\" is $?."      # 合法

```

```

29.
30. (( 1 / 2 ))                                # 结果小于1
31. echo "Exit status of \"(( 1 / 2 ))\" is $?." # 舍入至0。
32.                                           # 1
33.
34. (( 1 / 0 )) 2>/dev/null                    # 除0, 非法
35. #          ^^^^^^^^^^^^^
36. echo "Exit status of \"(( 1 / 0 ))\" is $?." # 1
37.
38. # "2>/dev/null" 的作用是什么？
39. # 如果将其移除会发生什么？
40. # 尝试移除这条语句并重新执行脚本。
41.
42. # ===== #
43.
44. # (( ... )) 在 if-then 中也非常有用
45.
46. var1=5
47. var2=4
48.
49. if (( var1 > var2 ))
50. then #^      ^      注意不是 $var1 和 $var2, 为什么？
51.     echo "$var1 is greater than $var2"
52. fi      # 5 大于 4
53.
54. exit 0

```

[^1]: 标记是一个具有特殊意义 (元语义) 的符号或者短字符串。在 Bash 里像 `[` 和 `.` (点命令) 这样的标记可以扩展成关键字和命令。

7.2 文件测试操作

- 7.2 文件测试操作

- -e
- -a
- -f
- -s
- -d
- -b
- -c
- -p
- -h
- -L
- -S
- -t
- -r
- -w
- -x
- -g
- -u
- -k
- -O
- -G
- -N
- f1 -nt f2
- f1 -ot f2
- f1 -ef f2
- !

7.2 文件测试操作

下列每一个 `test` 选项在满足条件时，返回0（真）。

-e

检测文件是否存在

-a

检测文件是否存在

等价于 `-e`。不推荐使用，已被弃用^[^1]。

-f

文件是常规文件(regular file)，而非目录或 [设备文件](#)

-S

文件大小不为0

-d

文件是一个目录

-b

文件是一个 [块设备](#)

-c

文件是一个 [字符设备](#)

```

1. device0="/dev/sda2"    # /    (根目录)
2. if [ -b "$device0" ]
3. then
4.     echo "$device0 is a block device."
5. fi
6.
7. # /dev/sda2 是一个块设备。
8.
9.
10.
11. device1="/dev/ttyS1"   # PCMCIA 调制解调卡
12. if [ -c "$device1" ]
13. then
14.     echo "$device1 is a character device."
15. fi
16.
17. # /dev/ttyS1 是一个字符设备。

```

-p

文件是一个 **管道设备**

```

1. function show_input_type()
2. {
3.     [ -p /dev/fd/0 ] && echo PIPE || echo STDIN
4. }
5.
6. show_input_type "Input"           # STDIN
7. echo "Input" | show_input_type    # PIPE
8.
9. # 这个例子由 Carl Anderson 提供。

```

-h

文件是一个 **符号链接**

-L

文件是一个符号链接

-S

文件是一个 套接字

-t

文件（文件描述符）与终端设备关联

该选项通常被用于 测试 脚本中的 `stdin [-t 0]` 或 `stdout [-t 1]` 是否为终端设备。

-r

该文件对执行测试的用户可读

-w

该文件对执行测试的用户可写

-x

该文件可被执行测试的用户所执行

-g

文件或目录设置了 `set-group-id` `sgid` 标志

如果一个目录设置了 `sgid` 标志，那么在该目录中所有的新建文件的权限组都归属于该目录的权限组，而非文件创建者的权限组。该标志对共享文件夹很有用。

-u

文件设置了 `set-user-id` `suid` 标志。

一个属于 `root` 的可执行文件设置了 `suid` 标志后，即使是一个普通用户执行也拥有 `root` 权限^[^2]。对需要访问硬件设备的可执行文件（例如 `pppd` 和 `cdrecord`）很有用。如果没有 `suid` 标志，这些可执行文件就不能被非 `root` 用户所调用了。

```
1. -rwsr-xr-t    1 root      178236 Oct  2  2000 /usr/sbin/pppd
```

设置了 `suid` 标志后，在权限中会显示 `s`。

-k

设置了粘滞位(sticky bit)。

标志粘滞位是一种特殊的文件权限。如果文件设置了粘滞位，那么该文件将会被存储在高速缓存中以便快速访问^[^3]。如果目录设置了该标记，那么它将会对目录的写权限进行限制，目录中只有文件的拥有者可以修改或删除文件。设置标记后你可以在权限中看到 `t`。

```
1. drwxrwxrwt    7 root      1024 May 19 21:26 tmp/
```

如果一个用户不是设置了粘滞位目录的拥有者，但对该目录有写权限，那么他仅仅可以删除目录中他所拥有的文件。这可以防止用户不经意间删除或修改其他人的文件，例如 `/tmp` 文件夹。（当然目录的所有者可以删除或修改该目录下的所有文件）

-0

执行用户是文件的拥有者

-G

文件的组与执行用户的组相同

-N

文件在上次访问后被修改过了

`f1 -nt f2`

文件 `f1` 比文件 `f2` 新

`f1 -ot f2`

文件 `f1` 比文件 `f2` 旧

`f1 -ef f2`

文件 `f1` 和文件 `f2` 硬链接到同一个文件

!

取反——对测试结果取反(如果条件缺失则返回真)。

样例 7-4. 检测链接是否损坏

```

1. #!/bin/bash
2. # broken-link.sh
3. # Lee bigelow <ligelowbee@yahoo.com> 编写。
4. # ABS Guide 经许可可以使用。
5.
6. # 该脚本用来发现输出损坏的链接。输出的结果是被引用的,
7. #+ 所以可以直接导到 xargs 中进行处理 :)
8. # 例如: sh broken-link.sh /somedir /someotherdir|xargs rm
9. #

```



```

10. # 更加优雅的方式：
11. #
12. # find "somedir" -type l -print0|\
13. # xargs -r0 file|\
14. # grep "broken symbolic"|
15. # sed -e 's/^\ |: *broken symbolic.*$/g'
16. #
17. # 但是这种方法不是纯 Bash 写法。
18. # 警告：小心 /proc 文件下的文件和任意循环链接！
19. #####
20.
21.
22. # 如果不给脚本传任何参数，那么 directories-to-search 设置为当前目录
23. #+ 否则设置为传进的参数
24. #####
25.
26. [ $# -eq 0 ] && directory=`pwd` || directory=$@
27.
28.
29. # 函数 linkchk 是用来检测传入的文件夹中是否包含损坏的链接文件，
30. #+ 并引用输出他们。
31. # 如果文件夹中包含子文件夹，那么将子文件夹继续传给 linkchk 函数进行检测。
32. #####
33.
34. linkchk () {
35.     for element in $1/*; do
36.         [ -h "$element" -a ! -e "$element" ] && echo "\"$element\""
37.         [ -d "$element" ] && linkchk $element
38.         # -h 用来检测是否是链接，-d 用来检测是否是文件夹。
39.     done
40. }
41.
42. # 检测传递给 linkchk() 函数的参数是否是一个存在的文件夹，
43. #+ 如果不是则报错。
44. #####
45. for directory in $direcotrys; do
46.     if [ -d $directory ]
47.     then linkchk $directory

```

```

48.         else
49.             echo "$directory is not a directory"
50.             echo "Usage $0 dir1 dir2 ..."
51.         fi
52.     done
53.
54. exit $?

```

样例 31-1, 样例 11-8, 样例 11-3, 样例 31-3和样例 A-1 也包含了文件测试操作符的使用。

[^1]: 摘自1913年版本的韦氏词典

```

1. Deprecate
...

To pray against, as an evil;
to seek to avert by prayer;
to desire the removal of;
to seek deliverance from;
to express deep regret for;
to disapprove of strongly.

```

[^2]: 注意使用 suid 的可执行文件可能会带来安全问题。suid 标记对 shell 脚本没有影响。

[^3]: 在 Linux 系统中, 文件已经不使用粘滞位了, 粘滞位只作用于目录。

7.3 其他比较操作

- 7.3 其他比较操作

- 整数比较

- -eq
- -ne
- -gt
- -ge
- -lt
- -le
- <
- <=
- >
- >=

- 字符串比较

- =
- ==
- !=
- <
- >
- -z
- -n

- 复合比较

- -a
- -o

7.3 其他比较操作

二元比较操作可以比较变量或者数量。

需要注意的是，整数和字符串比较使用的是两套不同的操作符。

整数比较

-eq

等于

```
if [ "$a" -eq "$b" ]
```

-ne

不等于

```
if [ "$a" -ne "$b" ]
```

-gt

大于

```
if [ "$a" -gt "$b" ]
```

-ge

大于等于

```
if [ "$a" -ge "$b" ]
```

-lt

小于

```
if [ "$a" -lt "$b" ]
```

-le

小于等于

```
if [ "$a" -le "$b" ]
```

<

小于（使用 [双圆括号](#)）

```
(( "$a" < "$b" ))
```

<=

小于等于（使用双圆括号）

```
(( "$a" <= "$b" ))
```

>

大于（使用双圆括号）

```
(( "$a" > "$b" ))
```

>=

大于等于（使用双圆括号）

```
(( "$a" >= "$b" ))
```

字符串比较

=

等于

```
if [ "$a" = "$b" ]
```



注意在 `=` 前后要加上[空格](#)

`if ["$a"="$b"]` 和上面不等价。

`==`

等于

`if ["$a" == "$b"]`

和 `=` 同义



`==` 操作符在 [双方括号](#) 和单方括号里的功能是不同的。

1. `[[$a == z*]]` # \$a 以 "z" 开头时为真（模式匹配）
2. `[[$a == "z*"]]` # \$a 等于 z* 时为真（字符匹配）
- 3.
4. `[$a == z*]` # 发生文件匹配和字符分割。
5. `["$a" == "z*"]` # \$a 等于 z* 时为真（字符匹配）
- 6.
7. # 感谢 Stéphane Chazelas

`!=`

不等于

`if ["$a" != "$b"]`

在 `[[...]]` 结构中会进行模式匹配。

`<`

小于，按照 [ASCII码](#) 排序。

`if [["$a" < "$b"]]`

`if ["$a" \< "$b"]`

注意在 `[]` 结构里 `<` 需要被 [转义](#)。

>

大于，按照 ASCII 码排序。

```
if [[ "$a" > "$b" ]]
```

```
if [ "$a" \> "$b" ]
```

注意在 `[]` 结构里 `>` 需要被转义。

样例 27-11 包含了比较操作符。

-Z

字符串为空，即字符串长度为0。

```
1. String='' # 长度为0的字符串变量。
2.
3. if [ -z "$String" ]
4. then
5.     echo "\$String is null."
6. else
7.     echo "\$String is NOT null."
8. fi # $String is null.
```

-n

字符串非空（ `null` ）。



使用 `-n` 时字符串必须是在括号中且被引用的。使用 `! -z` 判断未引用的字符串或者直接判断（[样例 7-6](#)）通常可行，但是非常危险。判断字符串时一定要引用`[^1]`。

样例 7-5. 算术比较和字符串比较

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

```

2.
3.  a=4
4.  b=5
5.
6.  # 这里的 "a" 和 "b" 可以是整数也可以是字符串。
7.  # 因为 Bash 的变量是弱类型的，因此字符串和整数比较有很多相同之处。
8.
9.  # 在 Bash 中可以用处理整数的方式来处理全是数字的字符串。
10. # 但是谨慎使用。
11.
12. echo
13.
14. if [ "$a" -ne "$b" ]
15. then
16.     echo "$a is not equal to $b"
17.     echo "(arithmetic comparison)"
18. fi
19.
20. echo
21.
22. if [ "$a" != "$b" ]
23. then
24.     echo "$a is not equal to $b."
25.     echo "(string comparison)"
26.     #      "4"  != "5"
27.     # ASCII 52 != ASCII 53
28. fi
29.
30. # 在这个例子里 "-ne" 和 "!=" 都可以。
31.
32. echo
33.
34. exit 0

```

样例 7-6. 测试字符串是否为空 (`null`)

```

1.  #!/bin/bash
2.  # str-test.sh: 测试是否为空字符串或是未引用的字符串。

```



```

3.
4. # 使用 if [ ... ] 结构
5.
6. # 如果字符串未被初始化，则其值是未定义的。
7. # 这种状态就是空 "null"（并不是 0）。
8.
9. if [ -n $string1 ]      # 并未声明或是初始化 string1。
10. then
11.     echo "String \"$string1\" is not null."
12. else
13.     echo "String \"$string1\" is null."
14. fi
15. # 尽管没有初始化 string1，但是结果显示其非空。
16.
17. echo
18.
19. # 再试一次。
20.
21. if [ -n "$string1" ]    # 这次引用了 $string1。
22. then
23.     echo "String \"$string1\" is not null."
24. else
25.     echo "String \"$string1\" is null."
26. fi                      # 在测试括号内引用字符串得到了正确的结果。
27.
28. echo
29.
30. if [ $string1 ]         # 这次只有一个 $string1。
31. then
32.     echo "String \"$string1\" is not null."
33. else
34.     echo "String \"$string1\" is null."
35. fi                      # 结果正确。
36. # 独立的 [ ... ] 测试操作符可以用来检测字符串是否为空。
37. # 最好将字符串进行引用 (if [ "$string1" ])。
38. #
39. # Stephane Chazelas 指出：
40. #     if [ $string1 ]    只有一个参数 "]"

```

```

41. #   if [ "$string1" ] 则有两个参数, 空的 "$string1" 和 "]"
42.
43.
44. echo
45.
46.
47. string1=initialized
48.
49. if [ $string1 ]          # $string1 这次仍然没有被引用。
50. then
51.     echo "String \"$string1\" is not null."
52. else
53.     echo "String \"$string1\" is null."
54. fi                      # 这次的结果仍然是正确的。
55. # 最好将字符串引用 (" $string1")
56.
57.
58. string1="a = b"
59.
60. if [ $string1 ]          # $string1 这次仍然没有被引用。
61. then
62.     echo "String \"$string1\" is not null."
63. else
64.     echo "String \"$string1\" is null."
65. fi                      # 这次没有引用就错了。
66.
67. exit 0   # 同时感谢 Florian Wisser 的提示。

```

样例 7-7. zmore

```

1. #!/bin/bash
2. # zmore
3.
4. # 使用筛选器 'more' 查看 gzipped 文件。
5.
6. E_NOARGS=85
7. E_NOTFOUND=86
8. E_NOTGZIP=87

```

```

9.
10. if [ $# -eq 0 ] # 作用和 if [ -z "$1" ] 相同。
11. # $1 可以为空： zmore "" arg2 arg3
12. then
13.     echo "Usage: `basename $0` filename" >&2
14.     # 将错误信息通过标准错误 stderr 进行输出。
15.     exit $E_NOARGS
16.     # 脚本的退出状态为 85。
17. fi
18.
19. filename=$1
20.
21. if [ ! -f "$filename" ]    # 引用字符串以防字符串中带有空格。
22. then
23.     echo "File $filename not found!" >&2    # 通过标准错误 stderr 进行输出。
24.     exit $E_NOTFOUND
25. fi
26.
27. if [ ${filename##*.} != "gz" ]
28. # 在括号内使用变量代换。
29. then
30.     echo "File $1 is not a gzipped file!"
31.     exit $E_NOTGZIP
32. fi
33.
34. zcat $1 | more
35.
36. # 使用筛选器 'more'
37. # 也可以用 'less' 替代
38.
39. exit $?    # 脚本的退出状态由管道 pipe 的退出状态决定。
40. # 实际上 "exit $?" 不一定要写出来，
41. ## 因为无论如何脚本都会返回最后执行命令的退出状态。

```

复合比较

-a

逻辑与

`exp1 -a exp2` 返回真当且仅当 `exp1` 和 `exp2` 均为真。

-o

逻辑或

如果 `exp1` 或 `exp2` 为真，则 `exp1 -o exp2` 返回真。

以上两个操作和 [双方括号](#) 结构中的 Bash 比较操作符号 `&&` 和 `||` 类似。

```
1. [[ condition1 && condition2 ]]
```

测试操作 `-o` 和 `-a` 可以在 `test` 命令或在测试括号中进行。

```
1. if [ "$expr1" -a "$expr2" ]
2. then
3.     echo "Both expr1 and expr2 are true."
4. else
5.     echo "Either expr1 or expr2 is false."
6. fi
```



rihad 指出：

```
1. [ 1 -eq 1 ] && [ -n "`echo true 1>&2`" ] # 真
2. [ 1 -eq 2 ] && [ -n "`echo true 1>&2`" ] # 没有输出
3. # ^^^^^^^ 条件为假。到这里为止，一切都按预期执行。
4.
5. # 但是
6. [ 1 -eq 2 -a -n "`echo true 1>&2`" ] # 真
7. # ^^^^^^^ 条件为假。但是为什么结果为真？
```

```

8.
9. # 是因为括号内的两个条件子句都执行了么？
10. [[ 1 -eq 2 && -n "`echo true 1>&2`" ]]      # 没有输出
11. # 并不是。
12.
13. # 所以显然 && 和 || 具备“短路”机制，
14. #+ 例如对于 &&，若第一个表达式为假，则不执行第二个表达式直接返回假，
15. #+ 而 -a 和 -o 则不是。

```

复合比较操作的例子可以参考 [样例 8-3](#)，[样例 27-17](#) 和 [样例 A-29](#)。

[^1]：S.C. 指出在复合测试中，仅仅引用字符串可能还不够。比如表达式 `[-n "$string" -o "$a" = "$b"]` 在某些 Bash 版本下，如果 `$string` 为空可能会出错。更加安全的方式是，对于可能为空的字符串，添加一个额外的字符，例如 `["x$string" != x -o "x$a" = "x$b"]`（其中的 `x` 互相抵消）。

7.4 嵌套 if/then 条件测试

- 7.4 嵌套 if/then 条件测试

7.4 嵌套 if/then 条件测试

可以嵌套 `if/then` 条件测试结构。嵌套的结果等价于使用 `&&` 复合比较操作符。

```
1. a=3
2.
3. if [ "$a" -gt 0 ]
4. then
5.     if [ "$a" -lt 5 ]
6.     then
7.         echo "The value of \"a\" lies somewhere between 0 and 5."
8.     fi
9. fi
10.
11. # 和下面的结果相同
12.
13. if [ "$a" -gt 0 ] && [ "$a" -lt 5 ]
14. then
15.     echo "The value of \"a\" lies somewhere between 0 and 5."
16. fi
```

在 样例 37-4 和 样例 17-11 中展示了嵌套 `if/then` 条件测试结构。

7.5 牛刀小试

- 7.5 牛刀小试

7.5 牛刀小试

系统文件 `xinitrc` 可以用来启动软件 X Server。该文件包含了许多 `if/then` 测试结构。下面的代码摘录自较早版本的 `xinitrc`（大约在 Red Hat 7.1 版本）。

```

1.  if [ -f $HOME/.Xclients ]; then
2.      exec $HOME/.Xclients
3.  elif [ -f /etc/X11/xinit/Xclients ]; then
4.      exec /etc/X11/xinit/Xclients
5.  else
6.      # 安全分支。尽管程序不会执行这个分支。
7.      # （我们在 Xclients 中也提供了相同的机制）增强程序可靠性。
8.      xclock -geometry 100x100-5+5 &
9.      xterm -geometry 80x50-50+150 &
10.     if [ -f /usr/bin/netcape -a -f /usr/share/doc/HTML/index.html
        ]; then
11.         netcape /usr/share/doc/HTML/index.html
12.     fi
13. fi

```

试着解释代码片段中的条件测试结构，然后试着在 `/etc/X11/xinit/xinitrc` 查看最新版本，并且分析其中的 `if/then` 条件测试结构。为了更好的进行分析，你可能需要继续阅读后面章节中对 `grep`，`sed` 和 [正则表达式](#) 的讨论。

8. 运算符相关话题

- [第八章 运算符相关话题](#)
 - [本章目录](#)

第八章 运算符相关话题

本章目录

- [8.1 运算符](#)
- [8.2 数字常量](#)
- [8.3 双圆括号结构](#)
- [8.4 运算符优先级](#)

8.1 运算符

- 8.1. 运算符

- 赋值运算符

- =

- 算术运算符

- +

- -

- *

- /

- **

- %

- +=

- -=

- *=

- /=

- %=

- 小结

- 位运算

- <<

- <<=

- >>

- >>=

- &

- &=

- |

- |=

- ~

- ^
- ^=
- 逻辑(布尔)运算符
 - !
 - &&
 - ||
 - 小结
- 其他运算符
 - ,

8.1. 运算符

赋值运算符

变量赋值，初始化或改变一个变量的值。

=

等号 `=` 赋值运算符，既可用于算术赋值，也可用于字符串赋值。

```
1. var=27
2. category=minerals # "="左右不允许有空格
```



注意，不要混淆 `=` 赋值运算符与 `=` 测试操作符。

```
1. # = 作为测试操作符
2.
3. if [ "$string1" = "$string2" ]
4. then
5.     command
6. fi
7.
```

8. # ["X\$string1" = "X\$string2"] 这样写是安全的,
9. # 这样写可以避免任意一个变量为空时的报错。
10. # (变量前加的"x"字符规避了变量为空的情况)

算术运算符

+

加

-

减

*

乘

/

除

**

幂运算

1. # Bash, 2.02版本, 推出了"("**)"幂运算操作符。
- 2.
3. `let "z=5**3"` # 5 * 5 * 5
4. `echo "z = $z"` # z = 125

%

取余(返回整数除法的余数)

```
1. bash$ expr 5 % 3
2. 2
```

5/3=1, 余2

取余运算符经常被用于生成一定范围内的数(案例9-11, 案例9-15), 以及格式化程序输出(案例 27-16, 案例 A-6)。

取余运算符还可以用来产生素数(案例A-15), 取余的出现大大扩展了整数的算术运算。

样例 8-1. 最大公约数

```
1. #!/bin/bash
2. # gcd.sh: 最大公约数
3. #      使用欧几里得算法
4.
5. # 两个整数的最大公约数 (gcd)
6. # 是两数能同时整除的最大数
7.
8. # 欧几里得算法使用辗转相除法
9. #   In each pass,
10. #       dividend <--- divisor
11. #       divisor  <--- remainder
12. #   until remainder = 0.
13. #   The gcd = dividend, on the final pass.
14. #
15. # 关于欧几里得算法更详细的讨论, 可以查看:
16. #   Jim Loy's site, http://www.jimloy.com/number/euclids.htm.
17.
18.
19. # -----
20. # 参数检查
21. ARGS=2
22. E_BADARGS=85
23.
```

```

24. if [ $# -ne "$ARGS" ]
25. then
26.     echo "Usage: `basename $0` first-number second-number"
27.     exit $E_BADARGS
28. fi
29. # -----
30.
31.
32. gcd ()
33. {
34.
35.     dividend=$1          # 随意赋值,
36.     divisor=$2           # 两数谁大谁小是无关紧要的,
37.                          # 为什么?
38.
39.     remainder=1          # 如果在测试括号里使用了一个未初始化的变量,
40.                          # 会报错的。
41.
42.     until [ "$remainder" -eq 0 ]
43.     do # ^^^^^^^^^^^ 该变量必须在使用前初始化!
44.         let "remainder = $dividend % $divisor"
45.         dividend=$divisor # 对被除数, 除数重新赋值
46.         divisor=$remainder
47.     done # 欧几里得算法
48.
49. } # 最后的 $dividend 就是最大公约数 (gcd)
50.
51.
52. gcd $1 $2
53.
54. echo; echo "GCD of $1 and $2 = $dividend"; echo
55.
56.
57. # 练习 :
58. # -----
59. # 1) 检查命令行参数, 保证其为整数,
60. #+ 如果有错误, 捕捉错误并在脚本退出前打印出适当的错误信息。
61. # 2) 使用本地变量(local variables)重写gcd()函数。

```

```
62.
63. exit 0
```

+=

加等 （加上一个数）[^1]

`let "var += 5"` 的结果是 `var` 变量的值增加了5。

--

减等 （减去一个数）

*=

乘等 （乘以一个数）

`let "var *= 4"` 的结果是 `var` 变量的值乘了4。

/=

除等 （除以一个数）

%=

余等 （取余赋值）

小结

算术运算符常用于 `expr` 或 `let` 表达式中。

样例 8-2. 使用算术运算符

```
1.  #!/bin/bash
2.  # 使变量自增1, 10种不同的方法实现
3.
```

```

4.  n=1; echo -n "$n "
5.
6.  let "n = $n + 1"    # 可以使用 let "n = n + 1"
7.  echo -n "$n "
8.
9.
10. : $(n = $n + 1)
11. # ":" 是必要的, 不加的话, bash会将
12. #+ "$(n = $n + 1)"看做一条命令。
13. echo -n "$n "
14.
15. (( n = n + 1 ))
16. # 更简洁的写法。
17. # 感谢 David Lombard指出。
18. echo -n "$n "
19.
20. n=$(( $n + 1 ))
21. echo -n "$n "
22.
23. : $[ n = $n + 1 ]
24. # ":" 是必要的, 不加的话, bash会将
25. #+ "$[ n = $n + 1 ]"看做一条命令。
26. # 即使"n"是字符串, 也是可行的。
27. echo -n "$n "
28.
29. n=$[ $n + 1 ]
30. # 即使"n"是字符串, 也是可行的。
31. #* 不要用这种写法, 它已被废弃不具有兼容性。
32. # 感谢 Stephane Chazelas.
33. echo -n "$n "
34.
35. # 使用C风格的自增运算符也是可以的
36. # 感谢 Frank Wang 指出。
37.
38. let "n++"           # let "++n" 可行
39. echo -n "$n "
40.
41. (( n++ ))           # (( ++n )) 可行

```

```

42. echo -n "$n "
43.
44. : $(( n++ ))      # : $(( ++n )) 可行
45. echo -n "$n "
46.
47. : ${ n++ }        # : ${ ++n } 可行
48. echo -n "$n "
49.
50. echo
51.
52. exit 0

```

在早期的Bash版本中，整型变量是带符号的长整型数（32-bit），取值范围从 -2147483648 到 2147483647。如果算术操作超出了整数的取值范围，结果会不准确。

```

1. echo $BASH_VERSION    # Bash 1.14版本
2.
3. a=2147483646
4. echo "a = $a"          # a = 2147483646
5. let "a+=1"             # 自增 "a".
6. echo "a = $a"          # a = 2147483647
7. let "a+=1"             # 再次自增"a", 超出取值范围。
8. echo "a = $a"          # a = -2147483648
9.                        #      错误：超出范围，
10.                      #+      最左边的符号位被重置，
11.                      #+      结果变负

```

Bash版本 \geq 2.05b，Bash支持了64-bit整型数。



注意，Bash并不支持浮点运算，Bash会将带小数点的数看做字符串。

```

1. a=1.5
2.
3. let "b = $a + 1.3"    # 报错
4. # t2.sh: let: b = 1.5 + 1.3: syntax error in expression

```



```

5. # (error token is ".5 + 1.3")
6.
7. echo "b = $b"      # b=1

```

如果你想在脚本中使用浮点数运算，借助**bc**或外部数学函数库吧。

位运算

位运算很少出现在shell脚本中，在bash中加入位运算的初衷似乎是为了操控和检测来自 `ports` 或 `sockets` 的数据。位运算在编译型语言中能发挥更大的作用，比如C/C++，位运算提供了直接访问系统硬件的能力。然而，聪明的vladz在他的base64.sh(案例 A-54)脚本中也用到了位运算。

下面介绍位运算符。

<<

左移运算符(左移1位相当于乘2)

<<=

左移赋值

`let "var <<= 2"` 的结果是var变量的值向左移了2位(乘以4)

>>

右移运算符(右移1位相当于除2)

>>=

右移赋值

$\&$

按位与 (AND)

$\&=$

按位与等 (AND-equal)

$|$

按位或 (OR)

$|=$

按位或等 (OR-equal)

\sim

按位取反

\wedge

按位异或 (XOR)

$\wedge=$

按位异或等 (XOR-equal)

逻辑(布尔)运算符

$!$

非 (NOT)

```

1. if [ ! -f $FILENAME ]
2. then
3.    ...

```

&&

与(AND)

```

1. if [ $condition1 ] && [ $condition2 ]
2. # 等同于: if [ $condition1 -a $condition2 ]
3. # 返回true如果 condition1 和 condition2 同时为真...
4.
5. if [[ $condition1 && $condition2 ]] # 可行
6. # 注意, && 运算符不能用在[ ... ]结构里。

```



&&也可以被用在 `list` 结构中连接命令。

||

或(OR)

```

1. if [ $condition1 ] || [ $condition2 ]
2.
3. # 等同于: if [ $condition1 -a $condition2 ]
4. # 返回true如果 condition1 和 condition2 任意一个为真...
5.
6. if [[ $condition1 || $condition2 ]] # 可行
7. # 注意, || 运算符不能用在[ ... ]结构里。

```

小结

样例 8-3. 在条件测试中使用 && 和 ||

```

1. #!/bin/bash
2.

```

```
3.  a=24
4.  b=47
5.
6.  if [ "$a" -eq 24 ] && [ "$b" -eq 47 ]
7.  then
8.      echo "Test #1 succeeds."
9.  else
10.     echo "Test #1 fails."
11. fi
12.
13. # 错误:  if [ "$a" -eq 24 && "$b" -eq 47 ]
14. #         这样写的话, bash会先执行'[ "$a" -eq 24'
15. #         然后就找不到右括号']'了...
16. #
17. # 注意:  if [[ $a -eq 24 && $b -eq 24 ]] 这样写是可以的
18. # 双方括号测试结构比单方括号更加灵活。
19. # (双方括号中的"&&"与单方括号中的"&&"意义不同)
20. # 感谢 Stephane Chazelas 指出。
21.
22.
23. if [ "$a" -eq 98 ] || [ "$b" -eq 47 ]
24. then
25.     echo "Test #2 succeeds."
26. else
27.     echo "Test #2 fails."
28. fi
29.
30.
31. # 使用 -a 和 -o 选项也具有同样的效果。
32. # 感谢 Patrick Callahan 指出。
33.
34.
35. if [ "$a" -eq 24 -a "$b" -eq 47 ]
36. then
37.     echo "Test #3 succeeds."
38. else
39.     echo "Test #3 fails."
40. fi
```

```

41.
42.
43. if [ "$a" -eq 98 -o "$b" -eq 47 ]
44. then
45.     echo "Test #4 succeeds."
46. else
47.     echo "Test #4 fails."
48. fi
49.
50.
51. a=rhino
52. b=crocodile
53. if [ "$a" = rhino ] && [ "$b" = crocodile ]
54. then
55.     echo "Test #5 succeeds."
56. else
57.     echo "Test #5 fails."
58. fi
59.
60. exit 0

```

`&&` 和 `||` 运算符也可以用在算术运算中。

```

1. bash$ echo $(( 1 && 2 )) $((3 && 0)) $((4 || 0)) $((0 || 0))
2. 1 0 1 0

```

其他运算符

/

逗号运算符

逗号运算符用于连接两个或多个算术操作，所有的操作会被依次求值（可能会有副作用）。²

```

1. let "t1 = ((5 + 3, 7 - 1, 15 - 4))"

```

```
2. echo "t1 = $t1"          ^^^^^^ # t1 = 11
3. # 这里的t1 被赋值了11, 为什么?
4.
5. let "t2 = ((a = 9, 15 / 3))" # 对"a"赋值并对"t2"求值。
6. echo "t2 = $t2    a = $a"    # t2 = 5    a = 9
```

逗号运算符常被用在 `for` 循环中。参看案例 11-13。

[^1]： 取决与不同的上下文，+= 也可能作为字符串连接符。它可以很方便地修改环境变量。

8.2 数字常量

- 8.2. 数字常量

8.2. 数字常量

通常情况下，shell脚本会把数字以十进制整数看待(base 10)，除非数字加了特殊的前缀或标记。

带前缀0的数字是八进制数(base 8)；带前缀0x的数字是十六进制数(base 16)。

内嵌 # 的数字会以 BASE#NUMBER 的方式进行求值（不能超出当前shell支持整数的范围）。

样例 8-4. 数字常量的表示

```

1.  #!/bin/bash
2.  # numbers.sh: 不同进制数的表示
3.
4.  # 十进制数：默认
5.  let "dec = 32"
6.  echo "decimal number = $dec"           # 32
7.  # 一切正常。
8.
9.
10. # 八进制数：带前导'0'的数
11. let "oct = 032"
12. echo "octal number = $oct"           # 26
13. # 结果以 十进制 打印输出了。
14. # -----
15.
16.
17. # 十六进制数：带前导'0x'或'0X'的数
18. let "hex = 0x32"
19. echo "hexadecimal number = $hex"     # 50

```

```

20.
21. echo $((0x9abc))                # 39612
22. #      ^^      ^^      双圆括号进行表达式求值
23. # 结果以十进制打印输出。
24.
25.
26.
27. # 其他进制数: BASE#NUMBER
28. # BASE 范围: 2 - 64
29. # NUMBER 必须以 BASE 规定的正确形式书写, 如下:
30.
31. let "bin = 2#111100111001101"
32. echo "binary number = $bin"      # 31181
33.
34. let "b32 = 32#77"
35. echo "base-32 number = $b32"    # 231
36.
37. let "b64 = 64#@_"
38. echo "base-64 number = $b64"    # 4031
39.
40. # 这种表示法只对进制范围(2 - 64)内的 ASCII 字符有效。
41. # 10 数字 + 26 小写字母 + 26 大写字母 + @ + _
42.
43.
44. echo
45.
46. echo $((36#zz)) $((2#10101010)) $((16#AF16)) $((53#1aA))
47.                                     # 1295 170 44822 3375
48.
49. # 重要提醒:
50. # -----
51. # 使用超出进制范围以外的符号会报错。
52.
53. let "bad_oct = 081"
54.
55. # (可能的) 报错信息:
56. # bad_oct = 081: value too great for base (error token is "081")
57. #           Octal numbers use only digits in the range 0 - 7.

```



```
58.  
59.  exit $?          # 退出码 = 1 (错误)  
60.  
61.  # 感谢 Rich Bartell 和 Stephane Chazelas 的说明。
```

8.3 双圆括号结构

- 双圆括号结构

双圆括号结构

与 `let` 命令类似，`((...))` 结构允许对算术表达式的扩展和求值。它是 `let` 命令的简化形式。例如，`a=$((5 + 3))` 会将变量 `a` 赋值成 `5 + 3`，也就是8。在Bash中，双圆括号结构也允许以C风格的方式操作变量。例如，`((var++))`。

样例 8-5. 以C风格的方式操作变量

```

1.  #!/bin/bash
2.  # c-vars.sh
3.  # 以C风格的方式操作变量，使用(( ... ))结构
4.
5.
6.  echo
7.
8.  (( a = 23 )) # C风格的变量赋值，注意"="等号前后都有空格
9.
10. echo "a (initial value) = $a" # 23
11.
12. (( a++ )) # 后缀自增'a', C-style.
13. echo "a (after a++) = $a" # 24
14.
15. (( a-- )) # 后缀自减'a', C-style.
16. echo "a (after a--) = $a" # 23
17.
18.
19. (( ++a )) # 前缀自增'a', C-style.
20. echo "a (after ++a) = $a" # 24
21.
22. (( --a )) # 前缀自减'a', C-style.
```

```

23. echo "a (after --a) = $a"          # 23
24.
25. echo
26.
27. #####
28. # 注意, C风格的++, --运算符, 前缀形式与后缀形式有不同的
29. #+ 副作用。
30.
31. n=1; let --n && echo "True" || echo "False" # False
32. n=1; let n-- && echo "True" || echo "False" # True
33.
34. # 感谢 Jeroen Domburg。
35. #####
36.
37. echo
38.
39. (( t = a<45?7:11 )) # C风格三目运算符。
40. #      ^  ^  ^
41. echo "If a < 45, then t = 7, else t = 11." # a = 23
42. echo "t = $t "                          # t = 7
43.
44. echo
45.
46.
47. # -----
48. # 复活节彩蛋!
49. # -----
50. # Chet Ramey 偷偷往Bash里加入了C风格的语句结构,
51. # 还没写文档说明 (实际上很多是从ksh中继承过来的)。
52. # 在Bash 文档中, Ramey把 (( ... ))结构称为shell 算术运算,
53. # 但是这种表述并不准确...
54. # 抱歉啊, Chet, 把你的秘密抖出来了。
55.
56. # 参看 "for" 和 "while" 循环章节关于 (( ... )) 结构的部分。
57.
58. # (( ... )) 结构在Bash 2.04版本之后才能正常工作。
59.
60. exit

```

还可以参看 样例 **11-13** 与 样例 **8-4**。

8.4 运算符优先级

- [运算符优先级](#)

运算符优先级

在脚本中，运算执行的顺序被称为优先级：高优先级的操作会比低优先级的操作先执行。^{[^1](#)}

表 8-1. 运算符优先级(从高到低)

运算符	含义	注解	
var++ var-	后缀自增/ 自减	C风格运算符	
++var -var	前缀自增/ 自减		
! ~	按位取反/ 逻辑取反	对每一比特位取反/对逻辑判断的结果取反	
**	幂运算	算数运算符	
* / %	乘，除，取余	算数运算符	
+ -	加，减	算数运算符	
<< >>	左移，右移	比特位运算符	
-z -n	一元比较	字符串是/否为空	
-e -f -t -x, etc	一元比较	文件测试	
-lt -gt -le -ge <= >=	复合比较	字符串/整数比较	
-nt -ot -ef	复合比较	文件测试	
&	AND(按位与)	按位与操作	
^	XOR(按位异或)	按位异或操作	
\		OR(按位或)	按位或操作

&& -a	AND(逻辑与)	逻辑与, 复合比较		
\	\	-o	OR(逻辑或)	逻辑或, 复合比较
?:	if/else三目运算符	C风格运算符		
=	赋值	不要与test中的等号混淆		
*= /= %= += -= <<= >>= &=	赋值运算	先运算后赋值		
,	逗号运算符	连接一系列语句		

实际上, 你只需要记住以下规则就可以了:

- 先乘除取余, 后加减, 与算数运算相似
- 复合逻辑运算符, &&, ||, -a, -o 优先级较低
- 优先级相同的操作按从左至右顺序求值

现在, 让我们利用运算符优先级的知识来分析一下 *Fedora Core Linux* 中的 `/etc/init.d/functions` 文件。

```

1. while [ -n "$remaining" -a "$retry" -gt 0 ]; do
2.
3. # 初看之下很恐怖...
4.
5.
6. # 分开来分析
7. while [ -n "$remaining" -a "$retry" -gt 0 ]; do
8. #      --condition 1-- ^^ --condition 2--
9.
10. # 如果变量"$remaining" 长度不为0
11. #+      并且AND (-a)
12. #+ 变量 "$retry" 大于0
13. #+ 那么
14. #+ [ 方括号表达式 ] 返回成功(0)
15. #+ while-loop 开始迭代执行语句。
16. # =====

```

```

17. # "condition 1" 和 "condition 2" 在 AND之前执行, 为什么?
18. # 因为AND(-a)优先级比-n, -gt来得低, 逻辑与会在最后求值。
19. #####
20.
21. if [ -f /etc/sysconfig/i18n -a -z "${NOLOCALE:-}" ] ; then
22.
23.
24. # 同样, 分开来分析
25. if [ -f /etc/sysconfig/i18n -a -z "${NOLOCALE:-}" ] ; then
26. #     --condition 1----- ^^ --condition 2-----
27.
28. # 如果文件"/etc/sysconfig/i18n" 存在
29. #+      并且AND (-a)
30. #+ 变量 $NOLOCALE 长度不为0
31. #+ 那么
32. #+ [ 方括号表达式 ] 返回成功(0)
33. #+ 执行接下来的语句。
34. #
35. # 和之前的情况一样, 逻辑与AND(-a)最后求值。
36. # 因为在方括号测试结构中, 逻辑运算的优先级是最低的。
37. # =====
38. # 注意:
39. # ${NOLOCALE:-} 是一个参数扩展式, 看起来有点多余。
40. # 但是, 如果 $NOLOCALE 没有提前声明, 它会被设成null,
41. # 在某些情况下, 这会有点问题。

```



为了避免在复杂比较运算中的错误, 可以把运算分散到几个括号结构中。

```

1. if [ "$v1" -gt "$v2" -o "$v1" -lt "$v2" -a -e "$filename" ]
2. # 这样写不清晰...
3.
4. if [[ "$v1" -gt "$v2" ]] || [[ "$v1" -lt "$v2" ]] && [[ -e
   "$filename" ]]
5. # 好多了 -- 把逻辑判断分散到多个组之中

```


第三部分 shell进阶

- 第三部分 shell进阶
 - 目录

第三部分 shell进阶

目录

- 9. 换个角度看变量
 - 9.1 内部变量
 - 9.2 指定变量属性: `declare` 或 `typeset`
 - 9.3 `$RANDOM`: 随机产生整数
- 10. 变量处理
 - 10.1 字符串处理
 - 10.1.1 使用 `awk` 处理字符串
 - 10.1.2 参考资料
 - 10.2 参数替换
- 11. 循环与分支
 - 11.1 循环
 - 11.2 嵌套循环
 - 11.3 循环控制
 - 11.4 测试与分支
- 12. 命令替换
- 13. 算术扩展
- 14. 休息时间

10. 变量处理

- [第十章 变量处理](#)
 - [本章目录](#)

第十章 变量处理

本章目录

- [10.1 字符串处理](#)
 - [10.1.1 使用 `awk` 处理字符串](#)
 - [10.1.2 参考资料](#)
- [10.2 参数替换](#)

10.1 字符串处理

- 10.1 字符串处理

- 字符串长度

- `${#string}`
- `expr length $string`
- `expr "$string" : '.*'`

- 起始部分字符串匹配长度

- `expr match "$string" '$substring'`
- `expr "$string" : '$substring'`

- 索引

- `expr index $string $substring`

- 截取字符串（字符串分片）

- `${string:position}`
- `${string:position:length}`
- `expr substr $string $position $length`
- `expr match "$string" '\($substring\).'`
- `expr "$string" : '\($substring\).'`
- `expr match "$string" '.*\($substring\).'`
- `expr "$string" : '.*\($substring\).'`

- 删除子串

- `${string#substring}`
- `${string##substring}`
- `${string%substring}`
- `${string%%substring}`

- 子串替换

- `${string/substring/replacement}`
- `${string//substring/replacement}`

- `${string/#substring/replacement}`
- `${string/%substring/replacement}`

10.1 字符串处理

Bash 支持的字符串操作数量达到了一个惊人的数目。但可惜的是，这些操作工具缺乏一个统一的核心。他们中的一些是[参数代换](#)的子集，另外一些则是 UNIX 下 `expr` 函数的子集。这将会导致语法前后不一致或者功能上出现重叠，更不用说那些可能导致的混乱了。

字符串长度

```
${#string}
```

```
expr length $string
```

上面两个表达式等价于C语言中的 `strlen()` 函数。

```
expr "$string" : '.*'
```

```
1. stringZ=abcABC123ABCabc
2.
3. echo ${#stringZ}           # 15
4. echo `expr length $stringz` # 15
5. echo `expr "$stringZ" : '.*'` # 15
```

样例 10-1. 在文本的段落之间插入空行

```
1. #!/bin/bash
2. # paragraph-space.sh
3. # 版本 2.1, 发布日期 2012年7月29日
4.
5. # 在无空行的文本文件的段落之间插入空行。
6. # 像这样使用: $0 <FILENAME
7.
8. MINLEN=60 # 可以试试修改这个值。它用来做判断。
```

```

9.  # 假设一行的字符数小于 $MINLEN, 并且以句点结束段落。
10. #+ 结尾部分有练习！
11.
12. while read line # 当文件有许多行的时候
13. do
14.     echo "$line" # 输出行本身。
15.
16.     len=${#line}
17.     if [[ "$len" -lt "$MINLEN" && "$line" =~ [\.\.]+$ ]]
18. # if [[ "$len" -lt "$MINLEN" && "$line" =~ \[*\.\.]+$ ]]
19. # 新版Bash将不能正常运行前一个版本的脚本。Ouch！
20. # 感谢 Halim Srama 指出这点, 并且给出了修正版本。
21.     then echo # 在该行以句点结束时,
22.     fi #+ 增加一行空行。
23. done
24.
25. exit
26.
27. # 练习：
28. # -----
29. # 1) 该脚本通常会在文件的最后插入一个空行。
30. #+ 尝试解决这个问题。
31. # 2) 在第17行仅仅考虑到了以句点作为句子终止的情况。
32. #+ 修改以满足其他的终止符, 例如 ?, ! 和 "。

```

起始部分字符串匹配长度

```
expr match "$string" '$substring'
```

其中, `$substring` 是一个正则表达式。

```
expr "$string" : '$substring'
```

其中, `$substring` 是一个正则表达式。

```

1.
2. stringZ=abcABC123ABCabc
3. # |-----|

```

```

4. #      12345678
5.
6. echo `expr match "$stringZ" 'abc[A-Z]*.2'` # 8
7. echo `expr "$stringZ" : 'abc[A-Z]*.2'`      # 8

```

索引

```
expr index $string $substring
```

返回在 `$string` 中第一个出现的 `$substring` 字符所在的位置。

```

1. stringZ=abcABC123ABCabc
2. #      123456 ...
3. echo `expr index "$stringZ" C12`          # 6
4.                                           # C 的位置。
5. echo `expr index "$stringZ" 1c`           # 3
6. # 'c' (第三号位) 较 '1' 出现的更早。

```

几乎等价于C语言中的 `strchr()` 。

截取字符串（字符串分片）

```
${string:position}
```

在 `$string` 中截取自 `$position` 起的字符串。

如果参数 `$string` 是 “*” 或者 “@”，那么将会截取自

`$position` 起的[位置参数](#)。¹

```

${stringlength}

```

在 `$string` 中截取自 `$position` 起，长度为 `$length` 的字符串。

1.

```

2. stringZ=abcABC123ABCabc
3. #      0123456789.....
4. #      索引位置从0开始。
5.
6. echo ${stringZ:0}           # abcABC123ABCabc
7. echo ${stringZ:1}           # bcABC123ABCabc
8. echo ${stringZ:7}           # 23ABCabc
9.
10. echo ${stringZ:7:3}         # 23A
11.                             # 三个字符的子字符串。
12.
13.
14.
15. # 从右至左进行截取可行么？
16.
17. echo ${stringZ:-4}          # abcABC123ABCabc
18. # ${parameter:-default} 将会得到整个字符串。
19. # 但是.....
20.
21. echo ${stringZ:(-4)}         # Cabc
22. echo ${stringZ: -4}         # Cabc
23. # 现在可以了。
24. # 括号或者增加空格都可以"转义"位置参数。
25.
26. # 感谢 Dan Jacobson 指出这些。

```

其中，参数 `position` 与 `length` 可以传入一个变量而不一定需要传入常量。

样例 10-2. 产生一个8个字符的随机字符串

```

1. #!/bin/bash
2. # rand-string.sh
3. # 产生一个8个字符的随机字符串。
4.
5. if [ -n "$1" ] # 如果在命令行中已经传入了参数,
6. then          #+ 那么就以它作为起始字符串。

```



```

7.   str0="$1"
8.   else                # 否则，就将脚本的进程标识符PID作为起始字符串。
9.   str0="$ $"
10.  fi
11.
12.  POS=2 # 从字符串的第二位开始。
13.  LEN=8 # 截取八个字符。
14.
15.  str1=$( echo "$str0" | md5sum | md5sum )
16.  #                ^^^^^^  ^^^^^^
17.  # 将字符串通过管道计算两次 md5 来进行两次混淆。
18.
19.  randstring="${str1:$POS:$LEN}"
20.  #                ^^^^^  ^^^^^
21.  # 允许传入参数
22.
23.  echo "$randstring"
24.
25.  exit $?
26.
27.  # bozo$ ./rand-string.sh my-password
28.  # 1bdd88c4
29.
30.  # 不过不建议将其作为一种能够抵抗黑客的生成密码的方法。

```

如果参数 `$string` 是 “*” 或者 “@”，那么将会截取自 `$position` 起，最大个数为 `$length` 的位置参数。

```

1.  echo ${*:2}           # 输出第二个及之后的所有位置参数。
2.  echo ${@:2}           # 同上。
3.
4.  echo ${*:2:3}         # 从第二个位置参数起，输出三个位置参数。

```

```
expr substr $string $position $length
```

在 `$string` 中截取自 `$position` 起，长度为 `$length` 的字符串。

```

1. stringZ=abcABC123ABCabc
2. #      123456789.....
3. #      索引位置从1开始。
4.
5. echo `expr substr $stringZ 1 2`      # ab
6. echo `expr substr $stringZ 4 3`      # ABC

```

```
expr match "$string" '\($substring\).'
```

在 `$string` 中截取自 `$position` 起的字符串，其中 `$substring` 是正则表达式。

```
expr "$string" : '\($substring\).'
```

在 `$string` 中截取自 `$position` 起的字符串，其中 `$substring` 是正则表达式。

```

1. stringZ=abcABC123ABCabc
2. #      =====
3.
4. echo `expr match "$stringZ" '\([b-c]*[A-Z]..[0-9]\)'\`      # abcABC1
5. echo `expr "$stringZ" : '\([b-c]*[A-Z]..[0-9]\)'\`      # abcABC1
6. echo `expr "$stringZ" : '\(.....\)'\`      # abcABC1
7. # 上面所有的形式都给出了相同的结果。

```

```
expr match "$string" '.*\($substring\).'
```

从 `$string` 结尾部分截取 `$substring` 字符串，其中 `$substring` 是正则表达式。

```
expr "$string" : '.*\($substring\).'
```

从 `$string` 结尾部分截取 `$substring` 字符串，其中 `$substring` 是正则表达式。

```

1. stringZ=abcABC123ABCabc
2. #      =====

```

```

3.
4. echo `expr match "$stringZ" '.*\([A-C][A-C][A-C][a-c]*\) '`      #
    ABCabc
5. echo `expr "$stringZ" : '.*\([.....\)`                          #
    ABCabc

```

删除子串

`${string#substring}`

删除从 `$string` 起始部分起，匹配到的最短的 `$substring`。

`${string##substring}`

删除从 `$string` 起始部分起，匹配到的最长的 `$substring`。

```

1. stringZ=abcABC123ABCabc
2. #      |----|      最长
3. #      |-----|    最短
4.
5. echo ${stringZ#a*C}      # 123ABCabc
6. # 删除 'a' 与 'c' 之间最短的匹配。
7.
8. echo ${stringZ##a*C}     # abc
9. # 删除 'a' 与 'c' 之间最长的匹配。
10.
11.
12.
13. # 你可以使用变量代替 substring。
14.
15. x='a*C'
16.
17. echo ${stringZ#$X}      # 123ABCabc
18. echo ${stringZ##$X}     # abc
19.                        # 同上。

```

`${string%substring}`

删除从 `$string` 结尾部分起，匹配到的最短的 `$substring`。

例如：

```
1. # 将当前目录下所有后缀名为 "TXT" 的文件改为 "txt" 后缀。
2. # 例如 "file1.TXT" 改为 "file1.txt"。
3.
4. SUFF=TXT
5. suff=txt
6.
7. for i in $(ls *.$SUFF)
8. do
9.     mv -f $i $(i%.$SUFF).$suff
10.    # 除了从变量 $i 右侧匹配到的最短的字符串之外，
11.    #+ 其他一切都保持不变。
12. done ### 如果需要，循环可以压缩成一行的形式。
13.
14. # 感谢 Rory Winston。
```

`${string%%substring}`

删除从 `$string` 结尾部分起，匹配到的最长的 `$substring`。

```
1.
2. stringZ=abcABC123ABCabc
3. #                      ||      最短
4. #          |-----|      最长
5.
6. echo ${stringZ%b*c}      # abcABC123ABCa
7. # 从结尾处删除 'b' 与 'c' 之间最短的匹配。
8.
9. echo ${stringZ%%b*c}      # a
10. # 从结尾处删除 'b' 与 'c' 之间最长的匹配。
```

这个操作对生成文件名非常有帮助。

样例 10-3. 改变图像文件的格式及文件名

```

1.  #!/bin/bash
2.  #  cvt.sh:
3.  #  将目录下所有的 MacPaint 文件转换为 "pbm" 格式。
4.
5.  #  使用由 Brian Henderson (bryanh@giraffe-data.com) 维护的
6.  #+ "netpbm" 包下的 "macptobpm" 二进制工具。
7.  #  Netpbm 是大多数 Linux 发行版的标准组成部分。
8.
9.  OPERATION=macptobpm
10. SUFFIX=pbm          # 新的文件名后缀。
11.
12. if [ -n "$1" ]
13. then
14.     directory=$1      # 如果已经通过脚本参数传入了目录名的情况.....
15. else
16.     directory=$PWD     # 否则就使用当前工作目录。
17. fi
18.
19. #  假设目标目录下的所有 MacPaint 图像文件都拥有
20. #+ ".mac" 的文件后缀名。
21.
22. for file in $directory/*    # 文件名匹配。
23. do
24.     filename=${file%.*c}    # 从文件名中删除 ".mac" 后缀
25.                             #+ ('.*c' 匹配 '.' 与 'c' 之间的
26.                             # 所有字符，包括其本身)。
27.     $OPERATION $file > "$filename.$SUFFIX"
28.                             # 将转换结果重定向到新的文件。
29.     rm -f $file            # 在转换后删除原文件。
30.     echo "$filename.$SUFFIX" # 将记录输出到 stdout 中。
31. done
32.
33. exit 0
34.
35. # 练习：
36. # -----
37. # 这个脚本会将当前工作目录下的所有文件进行转换。
38. # 修改脚本，使得它仅转换 ".mac" 后缀的文件。

```

```

39.
40.
41.
42. # *** 还可以使用另外一种方法。 *** #
43.
44. #!/bin/bash
45. # 将图像批处理转换成不同的格式。
46. # 假设已经安装了 imagemagick。（在大部分 Linux 发行版中都有）
47.
48. INFMT=png # 可以是 tif, jpg, gif 等等。
49. OUTFMT=pdf # 可以是 tif, jpg, gif, pdf 等等。
50.
51. for pic in *"$INFMT"
52. do
53.     p2=$(ls "$pic" | sed -e s/\.$INFMT//)
54.     # echo $p2
55.     convert "$pic" $p2.$OUTFMT
56. done
57.
58. exit $?

```

样例 10-4. 将流音频格式转换成 ogg 格式

```

1. #!/bin/bash
2. # ra2ogg.sh: 将流音频文件 (*.ra) 转换成 ogg 格式。
3.
4. # 使用 "mplayer" 媒体播放器程序：
5. #     http://www.mplayerhq.hu/homepage
6. # 使用 "ogg" 库与 "oggenc"：
7. #     http://www.xiph.org/
8. #
9. # 脚本同时需要安装一些解码器，例如 sipr.so 等等一些。
10. # 这些解码器可以在 compat-libstdc++ 包中找到。
11.
12.
13. OFILEPREF=${1%ra} # 删除 "ra" 后缀。
14. OFILESUFF=wav     # wav 文件后缀。
15. OUTFILE="$OFILEPREF"$OFILESUFF

```

```

16. E_NOARGS=85
17.
18. if [ -z "$1" ]           # 必须指定一个文件进行转换。
19. then
20.     echo "Usage: `basename $0` [filename]"
21.     exit $E_NOARGS
22. fi
23.
24.
25. #####
26. mplayer "$1" -ao pcm:file=$OUTFILE
27. oggenc "$OUTFILE" # 由 oggenc 自动加上正确的文件后缀名。
28. #####
29.
30. rm "$OUTFILE"           # 立即删除 *.wav 文件。
31.                         # 如果你仍需保留原文件，注释掉上面这一行即可。
32.
33. exit $?
34.
35. # 注意：
36. # -----
37. # 在网站上，点击一个 *.ram 的流媒体音频文件
38. #+ 通常只会下载到 *.ra 音频文件的 URL。
39. # 你可以使用 "wget" 或者类似的工具下载 *.ra 文件本身。
40.
41.
42. # 练习：
43. # -----
44. # 这个脚本仅仅转换 *.ra 文件。
45. # 修改脚本增加适应性，使其可以转换 *.ram 或其他文件格式。
46. #
47. # 如果你非常有热情，你可以扩展这个脚本使其
48. #+ 可以自动下载并且转换流媒体音频文件。
49. # 给定一个 URL，自动下载流媒体音频文件（使用 "wget"），
50. #+ 然后转换它。

```

下面是使用字符串截取结构对 `getopt` 的一个简单模拟。

样例 10-5. 模拟 `getopt`

```

1.  #!/bin/bash
2.  # getopt-simple.sh
3.  # 作者: Chris Morgan
4.  # 允许在高级脚本编程指南中使用。
5.
6.
7.  getopt_simple()
8.  {
9.      echo "getopt_simple()"
10.     echo "Parameters are '$*'"
11.     until [ -z "$1" ]
12.     do
13.         echo "Processing parameter of: '$1'"
14.         if [ ${1:0:1} = '/' ]
15.         then
16.             tmp=${1:1}           # 删除开头的 '/'
17.             parameter=${tmp%%=*} # 取出名称。
18.             value=${tmp##*=}     # 取出值。
19.             echo "Parameter: '$parameter', value: '$value'"
20.             eval $parameter=$value
21.         fi
22.         shift
23.     done
24. }
25.
26. # 将所有参数传递给 getopt_simple()。
27. getopt_simple $*
28.
29. echo "test is '$test'"
30. echo "test2 is '$test2'"
31.
32. exit 0 # 可以查看该脚本的修改版 UseGetOpt.sh。
33.
34. ---
35.
36. sh getopt_example.sh /test=value1 /test2=value2

```



```

37.
38. Parameters are '/test=value1 /test2=value2'
39. Processing parameter of: '/test=value1'
40. Parameter: 'test', value: 'value1'
41. Processing parameter of: '/test2=value2'
42. Parameter: 'test2', value: 'value2'
43. test is 'value1'
44. test2 is 'value2'

```

子串替换

```
${string/substring/replacement}
```

替换匹配到的第一个 `$substring` 为 `$replacement`。[^2]

```
${string//substring/replacement}
```

替换匹配到的所有 `$substring` 为 `$replacement`。

```

1. stringZ=abcABC123ABCabc
2.
3. echo ${stringZ/abc/xyz}      # xyzABC123ABCabc
4.                             # 将匹配到的第一个 'abc' 替换为 'xyz'。
5.
6. echo ${stringZ//abc/xyz}    # xyzABC123ABCxyz
7.                             # 将匹配到的所有 'abc' 替换为 'xyz'。
8.
9. echo -----
10. echo "$stringZ"            # abcABC123ABCabc
11. echo -----
12.                             # 字符串本身并不会被修改！
13.
14. # 匹配以及替换的字符串可以是参数么？
15. match=abc
16. repl=000
17. echo ${stringZ/$match/$repl} # 000ABC123ABCabc
18. #           ^       ^       ^^^
19. echo ${stringZ//$match/$repl} # 000ABC123ABC000

```

```

20. # Yes!           ^      ^      ^^^      ^^^
21.
22. echo
23.
24. # 如果没有给定 $replacement 字符串会怎样？
25. echo ${stringZ/abc}           # ABC123ABCabc
26. echo ${stringZ//abc}          # ABC123ABC
27. # 仅仅是将其删除而已。

```

```
${string/#substring/replacement}
```

替换 `${string}` 中最前端匹配到的 `$substring` 为 `$replacement`。

```
${string/%substring/replacement}
```

替换 `${string}` 中最末端匹配到的 `$substring` 为 `$replacement`。

```

1. stringZ=abcABC123ABCabc
2.
3. echo ${stringZ/#abc/XYZ}           # XYZABC123ABCabc
4.                                     # 将前端的 'abc' 替换为 'XYZ'
5.
6. echo ${stringZ/%abc/XYZ}           # abcABC123ABCXYZ
7.                                     # 将末端的 'abc' 替换为 'XYZ'

```

[^2]：注意根据使用时上下文的不同，`$substring` 和 `$replacement` 可以是文本字符串也可以是变量。可以参考第一个样例。

10.1.1 使用 awk 处理字符串

- 10.1.1 使用 `awk` 处理字符串

10.1.1 使用 `awk` 处理字符串

在 Bash 脚本中可以调用字符串处理工具 `awk` 来替换内置的字符串处理操作。

样例 10-6. 使用另一种方式来截取和定位子字符串

```

1. #!/bin/bash
2. # substring-extraction.sh
3.
4. String=23skidoo1
5. #      012345678      Bash
6. #      123456789      awk
7. # 注意不同字符串索引系统：
8. # Bash 中第一个字符的位置为0。
9. # Awk 中第一个字符的位置为1。
10.
11. echo ${String:2:4} # 从第3位开始（0-1-2），4个字符的长度
12.                      # skid
13.
14. # Awk 中与 ${string:pos:length} 等价的是 substr(string,pos,length)。
15. echo | awk '
16. { print substr("'"${String}"'",3,4)      # skid
17. }
18. '
19. # 将空的 "echo" 通过管道传递给 awk 作为一个模拟输入，
20. #+ 这样就不需要提供一个文件名来操作 awk 了。
21.
22. echo "-----"
23.
24. # 同样的：
25.
```

```
26. echo | awk '  
27. { print index("'"${String}"'", "skid")      # 3  
28. }                                           # (skid 从第3位开始)  
29. '      # 这里使用 awk 等价于 "expr index".  
30.  
31. exit 0
```

10.1.2 参考资料

- [10.1.2 参考资料](#)

10.1.2 参考资料

更多关于脚本中处理字符串的资料，可以查看 [章节 10.2](#) 以及

`expr` 命令的[相关章节](#)。

脚本样例：

1. [样例 16-9](#)
2. [样例 10-9](#)
3. [样例 10-10](#)
4. [样例 10-11](#)
5. [样例 10-13](#)
6. [样例 A-36](#)
7. [样例 A-41](#)

10.2 参数替换

• 10.2 参数替换

- `${parameter}`
- `${parameter-default}, ${parameter:-default}`
- `${parameter=default}, ${parameter:=default}`
- `${parameter+alt_value}, ${parameter:+alt_value}`
- `${parameter?err_msg}, ${parameter:?err_msg}`
- 变量长度 / 删除子串
 - `${#var}`
 - `${var#Pattern}, ${var##Pattern}`
 - `${var%Pattern}, ${var%%Pattern}`
- 变量扩展 / 替换子串
 - `${var:pos}`
 - `${var:pos:len}`
 - `${var/Pattern/Replacement}`
 - `${var//Pattern/Replacement}`
 - `${var/#Pattern/Replacement}`
 - `${var/%Pattern/Replacement}`
 - `${!varprefix*}, ${!varprefix@}`

10.2 参数替换

参数替换用来处理或扩展变量。

`${parameter}`

等同于 `$parameter`，是变量 `parameter` 的值。在一些特定的环境下，只允许使用不易混淆的 `${parameter}` 形式。

可以用于连接变量与字符串。

```
1. your_id=${USER}-on-${HOSTNAME}
2. echo "$your_id"
3. #
4. echo "Old \ $PATH = $PATH"
5. PATH=${PATH}:/opt/bin # 在脚本执行过程中临时在 $PATH 中加入 /opt/bin。
6. echo "New \ $PATH = $PATH"
```

`${parameter-default}`, `${parameter:-default}`

在没有设置变量的情况下使用缺省值。

```
1. var1=1
2. var2=2
3. # 没有设置 var3。
4.
5. echo ${var1-$var2} # 1
6. echo ${var3-$var2} # 2
7. #           ^      注意前面的 $ 前缀。
8.
9.
10.
11. echo ${username-`whoami`}
12. # 如果变量 $username 没有被设置, 输出 `whoami` 的结果。
```



`${parameter-default}` 与 `${parameter:-default}` 的作用几乎相同, 唯一不同的情况就是当变量 `parameter` 已经被声明但值为空时。

```
1. #!/bin/bash
2. # param-sub.sh
3.
4. # 无论变量的值是否为空, 其是否已被声明决定了缺省设置的触发。
5.
6. username0=
7. echo "username0 has been declared, but is set to null."
8. echo "username0 = ${username0-`whoami`}"
```

```

9.  # 将不会输出 `whoami` 的结果。
10.
11. echo
12.
13. echo username1 has not been declared.
14. echo "username1 = ${username1-`whoami`}"
15. # 将会输出 `whoami` 的结果。
16.
17. username2=
18. echo "username2 has been declared, but is set to null."
19. echo "username2 = ${username2:-`whoami`}"
20. #                                     ^
21. # 因为这里是 :- 而不是 -, 所以将会输出 `whoami` 的结果。
22. # 与上面的 username0 比较。
23.
24.
25. #
26.
27. # 再来一次：
28.
29. variable=
30. # 变量已被声明，但其值为空。
31.
32. echo "${varibale-0}"      # 没有输出。
33. echo "${variable:-1}"    # 1
34. #                         ^
35.
36. unset variable
37.
38. echo "${variable-2}"      # 2
39. echo "${variable:-3}"    # 3
40.
41. exit 0

```

当传入的命令行参数的数量不足时，可以使用这种缺省参数结构。

```

1. DEFAULT_FILENAME=generic.data
2. filename=${1:-$DEFAULT_FILENAME}

```



```

3. # 如果没有其他特殊情况, 下面的代码块将会操作文件 "generic.data"。
4. # 代码块开始
5. # ...
6. # ...
7. # ...
8. # 代码块结束
9.
10.
11.
12. # 摘自样例 "hanoi2.bash" :
13. DISKS=${1:-E_NOPARAM}    # 必须指定碟子的个数。
14. # 将 $DISKS 设置为传入的第一个命令行参数,
15. #+ 如果没有传入第一个参数, 则设置为 $E_NOPARAM。

```

可以查看 [样例 3-4](#), [样例 31-2](#) 和 [样例 A-6](#)。

可以同 [使用与链设置缺省命令行参数](#) 做比较。

```
${parameter=default}, ${parameter:=default}
```

在没有设置变量的情况下, 将其设置为缺省值。

两种形式的作用几乎相同, 唯一不同的情况与上面类似, 就是当变量 `parameter` 已经被声明但值为空时。[^1]

```

1. echo ${var=abc}    # abc
2. echo ${vat=xyz}    # abc
3. # $var 已经在第一条语句中被赋值为 abc, 因此第二条语句将不会改变它的值。

```

```
${parameter+alt_value}, ${parameter:+alt_value}
```

如果变量已被设置, 使用 `alt_value`, 否则使用空值。

两种形式的作用几乎相同, 唯一不同的情况就是当变量 `parameter` 已经被声明但值为空时, 看下面的例子。

```
1. echo "##### \${parameter+alt_value} #####"
```

```

2.  echo
3.
4.  a=${param1+xyz}
5.  echo "a = $a"      # a =
6.
7.  param2=
8.  a=${param2+xyz}
9.  echo "a = $a"      # a = xyz
10.
11. param3=123
12. a=${param3+xyz}
13. echo "a = $a"      # a = xyz
14.
15. echo
16. echo "##### \${parameter:+alt_value} #####"
17. echo
18.
19. a=${param4:+xyz}
20. echo "a = $a"      # a =
21.
22. param5=
23. a=${param5:+xyz}
24. echo "a = $a"      # a =
25. # 不同于 a=${param5+xyz}
26.
27. param6=123
28. a=${param6:+xyz}
29. echo "a = $a"      # a = xyz

```

```

${parameter?err_msg}, ${parameter:?err_msg}

```

如果变量已被设置，那么使用原值，否则输出 `err_msg` 并且终止脚本，返回 [错误码 1](#)。

两种形式的作用几乎相同，唯一不同的情况与上面类似，就是当变量 `parameter` 已经被声明但值为空时。

样例 10-7. 如何使用变量替换和错误信息

```

1.  #!/bin/bash
2.
3.  # 检查系统环境变量。
4.  # 这是一种良好的预防性维护措施。
5.  # 如果控制台用户的名称 $USER 没有被设置，那么主机将不能够识别用户。
6.
7.  : ${HOSTNAME?} ${USER?} ${HOME?} ${MAIL?}
8.  echo
9.  echo "Name of the machine is $HOSTNAME."
10. echo "You are $USER."
11. echo "Your home directory is $HOME."
12. echo "Your mail INBOX is located in $MAIL."
13. echo
14. echo "If you are reading this message,"
15. echo "critcial environmental variables have been set."
16. echo
17. echo
18.
19. # -----
20.
21. # ${variablename?} 结构统一可以检查脚本中的变量是否被设置。
22.
23. ThisVariable=Value-of-ThisVariable
24. # 顺带一提，这个字符串的值可以被设置成名称中不可以使用的禁用字符。
25. : ${ThisVariable?}
26. echo "Value of ThisVariable is $ThisVariable."
27.
28. echo; echo
29.
30.
31. : ${ZZXy23AB?"ZZXy23AB has not been set."}
32. # 因为 ZXy23AB 没有被设置，所以脚本会终止同时显示错误消息。
33.
34. # 你可以指定错误消息。
35. # : ${variablename?"ERROR MESSAGE"}
36.

```

```

37.
38. # 与这些结果相同: dummy_variable=${ZZXy23AB?}
39. #           dummy_variable=${ZZXy23AB?"ZZXy23AB has not been
    set."}
40. #
41. #           echo ${ZZXy23AB?} >/dev/null
42.
43. # 将上面这些检查变量是否被设置的方法同 "set -u" 作比较。
44.
45.
46.
47. echo "You will not see this message, because script already
    terminated."
48.
49. HERE=0
50. exit $HERE # 将不会从这里退出。
51.
52. # 事实上, 这个脚本将会返回退出码 (echo $? ) 1。

```

样例 10-8. 参数替换与 “usage” 消息

```

1. #!/bin/bash
2. # usage-message.sh
3.
4. : ${1?"Usage: $0 ARGUMENT"}
5. # 如果命令行参数缺失, 脚本将会在这里结束, 并且返回下面的错误信息。
6. #     usage-message.sh: 1: Usage: usage-message.sh ARGUMENT
7.
8. echo "These two lines echo only if command-line parameter given."
9. echo "command-line parameter = \"$1\""
10.
11. exit 0 # 仅当命令行参数存在是才会从这里退出。
12.
13. # 在传入和未传入命令行参数的情况下查看退出状态。
14. # 如果传入了命令行参数, 那么 "$?" 的结果是0。
15. # 如果没有, 那么 "$?" 的结果是1。

```

参数替换用来处理或扩展变量。下面的表达式是对 `expr` 处理字符串的操作的补足（查看样例 16-9）。这些特殊的表达式通常用来解析文件的路径名。

变量长度 / 删除子串

`${#var}`

字符串的长度（`$var` 中字符的个数）。对任意 [数组](#) `array`，`${#array}` 返回数组中第一个元素的长度。



以下情况例外：

- `${#*}` 和 `${#@}` 返回位置参数的个数。
- 任意数组 `array`，`${#array[*]}` 和 `${#array[@]}` 返回数组中元素的个数。

样例 10-9. 变量长度

```
1.  #!/bin/bash
2.  # length.sh
3.
4.  E_NO_ARGS=65
5.
6.  if [ $# -eq 0 ] # 脚本必须传入参数。
7.  then
8.      echo "Please invoke this script with one or more command-line
          arguments."
9.      exit $E_NO_ARGS
10. fi
11.
12. var01=abcdEFGH28ij
13. echo "var01 = ${var01}"
14. echo "Length of var01 = ${#var01}"
15. # 现在我们尝试加入空格。
16. var02="abcd EFGH28ij"
17. echo "var02 = ${var02}"
18. echo "Length of var02 = ${#var02}"
```

```

19.
20. echo "Number of command-line arguments passed to script = ${#@}"
21. echo "Number of command-line arguments passed to script = ${#*}"
22.
23. exit 0

```

```
${var#Pattern}, ${var##Pattern}
```

`${var#Pattern}` 删除 `$var` 前缀部分匹配到的最短长度的 `$Pattern`。

`${var##Pattern}` 删除 `$var` 前缀部分匹配到的最长长度的 `$Pattern`。

摘自 [样例 A-7](#) 的例子：

```

1. # 函数摘自样例 "day-between.sh"。
2. # 删除传入的参数中的前缀0。
3.
4. strip_leading_zero () # 删除传入参数中可能存在的
5. {                    #+ 前缀0。
6.     return=${1#0}    # "1" 代表 "$1"，即传入的参数。
7. }                   # 从 "$1" 中删除 "0"。

```

下面是由 Manfred Schwarb 提供的上述函数的改进版本：

```

1. strip_leading_zero2 () # 删除前缀0,
2. {                      # 否则 Bash 会将其解释为8进制数。
3.     shopt -s extglob    # 启用扩展通配特性。
4.     local val=${1##+(0)} # 使用本地变量，匹配前缀中所有的0。
5.     shopt -u extglob    # 禁用扩展通配特性。
6.     _strip_leading_zero2=${var:-0}
7.     # 如果输入的为0，那么返回 0 而不是 ""。

```

另外一个样例：

```

1. echo `basename $PWD`           # 当前工作目录的目录名。
2. echo "${PWD##*/}"              # 当前工作目录的目录名。
3. echo
4. echo `basename $0`             # 脚本名。
5. echo $0                       # 脚本名。
6. echo "${0##*/}"                # 脚本名。
7. echo
8. filename=test.data
9. echo "${filename##*.}"          # data
10.                               # 文件扩展名。

```

```
${var%Pattern}, ${var%%Pattern}
```

`${var%Pattern}` 删除 `$var` 后缀部分匹配到的最短长度的 `Pattern`。

`${var%%Pattern}` 删除 `$var` 后缀部分匹配到的最长长度的 `Pattern`。

在 Bash 的 [第二个版本](#) 中增加了一些额外的选择。

样例 10-10. 参数替换中的模式匹配

```

1. #!/bin/bash
2. # patt-matching.sh
3.
4. # 使用 # ## % %% 参数替换操作符进行模式匹配
5.
6. var1=abcd12345abc6789
7. pattern1=a*c # 通配符 * 可以匹配 a 与 c 之间的任意字符
8.
9. echo
10. echo "var1 = $var1"           # abcd12345abc6789
11. echo "var1 = ${var1}"         # abcd12345abc6789
12.                               # （另一种形式）
13. echo "Number of characters in ${var1} = ${#var1}"
14. echo

```

```

15.
16. echo "pattern1 = $pattern1"    # a*c  (匹配 'a' 与 'c' 之间的一切)
17. echo "-----"
18. echo '${var1#$pattern1}' = ' "${var1#$pattern1}"    #
    d12345abc6789
19. # 匹配到首部最短的3个字符
    abcd12345abc6789
20. #          ^                      | - |
21. echo '${var1##$pattern1}' = ' "${var1##$pattern1}"    #
    6789
22. # 匹配到首部最长的12个字符
    abcd12345abc6789
23. #          ^                      | -----
    -- |
24.
25. echo; echo; echo
26.
27. pattern2=b*9                # 匹配 'b' 与 '9' 之间的任意字符
28. echo "var1 = $var1"        # 仍旧是 abcd12345abc6789
29. echo
30. echo "pattern2 = $pattern2"
31. echo "-----"
32. echo '${var1%pattern2}' = ' "${var1%$pattern2}"    #      abcd12345a
33. # 匹配到尾部最短的6个字符
    abcd12345abc6789
34. #          ^                      | ---- |
35. echo '${var1%%pattern2}' = ' "${var1%%$pattern2}"    #      a
36. # 匹配到尾部最长的12个字符
    abcd12345abc6789
37. #          ^                      | -----
    ---- |
38.
39. # 牢记 # 与 ## 是从字符串左侧开始,
40. #      % 与 %% 是从右侧开始。
41.
42. echo
43.

```



```
44. exit 0
```

样例 10-11. 更改文件扩展名：

```
1.  #!/bin/bash
2.  # rfe.sh: 更改文件扩展名。
3.  #
4.  #          rfe old_extension new_extension
5.  #
6.  # 如：
7.  # 将当前目录下所有 *.gif 文件重命名为 *.jpg,
8.  #          rfe gif jpg
9.
10.
11. E_BADARGS=65
12.
13. case $# in
14.   0|1)          # 竖线 | 在这里表示逻辑或关系。
15.     echo "Usage: `basename $0` old_file_suffix new_file_suffix"
16.     exit $E_BADARGS # 如果只有0个或1个参数，那么退出脚本。
17.   ;;
18. esac
19.
20.
21. for filename in *. $1
22. # 遍历以第一个参数作为后缀名的文件列表。
23. do
24.   mv $filename ${filename%$1}$2
25.   # 删除文件后缀名，增加第二个参数作为后缀名。
26. done
27.
28. exit 0
```

变量扩展 / 替换子串

下面这些结构采用自 ksh。

```
${var:pos}
```

扩展为从偏移量 `pos` 处截取的变量 `var`。

```
${var:pos:len}
```

扩展为从偏移量 `pos` 处截取变量 `var` 最大长度为 `len` 的字符串。

```
${var/Pattern/Replacement}
```

替换 `var` 中第一个匹配到的 `Pattern` 为 `Replacement`。

如果 `Replacement` 被省略，那么匹配到的第一个 `Pattern` 将被替换为空，即删除。

```
${var//Pattern/Replacement}
```

全局替换。替换 `var` 中所有匹配到的 `Pattern` 为 `Replacement`。

跟上面一样，如果 `Replacement` 被省略，那么匹配到的所有 `Pattern` 将被替换为空，即删除。

样例 10-12. 使用模式匹配解析任意字符串

```
1.  #!/bin/bash
2.
3.  var1=abcd-1234-defg
4.  echo "var1 = $var1"
5.
6.  t=${var1#*-}*}
7.  echo "var1 (with everything, up to and including first - stripped
   out) = $t"
8.  #   t=${var1#*-} 效果相同,
9.  #+ 因为 # 只匹配最短的字符串,
10. #+ 并且 * 可以任意匹配, 其中也包括空字符串。
11. # (感谢 Stephane Chazelas 指出这一点。)
```

```

12.
13. t=${var##*- *}
14. echo "If var1 contains a \"-\", returns empty string...   var1 =
    $t"
15.
16.
17. t=${var1%*- *}
18. echo "var1 (with everything from the last - on stripped out) = $t"
19.
20. echo
21.
22. # -----
23. path_name=/home/bozo/ideas/thoughts/for.today
24. # -----
25. echo "path_name = $path_name"
26. t=${path_name##*/}
27. echo "path_name, stripped of prefixes = $t"
28. # 在这里与 t=`basename $path_name` 效果相同。
29. # t=${path_name%/*}; t=${t##*/} 是更加通用的方法,
30. #+ 但有时仍旧也会出现问题。
31. # 如果 $path_name 以换行结束, 那么 `basename $path_name` 将会失效,
32. #+ 但是上面这种表达式却可以。
33. # (感谢 S.C.)
34.
35. t=${path_name%/*.*}
36. # 同 t=`dirname $path_name` 效果相同。
37. echo "path_name, stripped of suffixes = $t"
38. # 在一些情况下会失效, 比如 "../", "/foo///", # "foo/", "/"。
39. # 在删除后缀时, 尤其是当文件名没有后缀, 目录名却有后缀时,
40. #+ 事情会变的非常复杂。
41. # (感谢 S.C.)
42.
43. echo
44.
45. t=${path_name:11}
46. echo "$path_name, with first 11 chars stripped off = $t"
47. t=${path_name:11:5}
48. echo "$path_name, with first 11 chars stripped off, length 5 = $t"

```

```

49.
50. echo
51.
52. t=${path_name/bozo/clown}
53. echo "$path_name with \"bozo\" replaced by \"clown\" = $t"
54. t=${path_name/today/}
55. echo "$path_name with \"today\" deleted = $t"
56. t=${path_name//o/O}
57. echo "$path_name with all o's capitalized = $t"
58. t=${path_name//o/}
59. echo "$path_name with all o's deleted = $t"
60.
61. exit 0

```

```
${var/#Pattern/Replacement}
```

替换 var 前缀部分匹配到的 Pattern 为 Replacement。

```
${var/%Pattern/Replacement}
```

替换 var 后缀部分匹配到的 Pattern 为 Replacement。

样例 10-13. 在字符串首部或尾部进行模式匹配

```

1.  #!/bin/bash
2.  # var-match.sh:
3.  # 演示在字符串首部或尾部进行模式替换。
4.
5.  v0=abc1234zip1234abc      # 初始值。
6.  echo "v0 = $v0"          # abc1234zip1234abc
7.  echo
8.
9.  # 在字符串首部进行匹配
10. v1=${v0/#abc/ABCDEF}     # abc1234zip123abc
11.                          # |-|
12. echo "v1 = $v1"          # ABCDEF1234zip1234abc
13.                          # |----|
14.

```

```

15. # 在字符串尾部进行匹配
16. v2=${v0/%abc/ABCDEF}      # abc1234zip123abc
17.                             #          | - |
18. echo "v2 = $v2"            # abc1234zip1234ABCDEF
19.                             #          | ---- |
20.
21. echo
22.
23. # -----
24. # 必须在字符串的最开始或者最末尾的地方进行匹配,
25. #+ 否则将不会发生替换。
26. # -----
27. v3=${v0/#123/000}          # 虽然匹配到了, 但是不在最开始的地方。
28. echo "v3 = $v3"            # abc1234zip1234abc
29.                             # 没有替换。
30. v4=${v0/%123/000}          # 虽然匹配到了, 但是不在最末尾的地方。
31. echo "v4 = $v4"            # abc1234zip1234abc
32.                             # 没有替换。
33.
34. exit 0

```

```

${!varprefix*}, ${!varprefix@}

```

匹配先前声明过所有以 `varprefix` 作为变量名前缀的变量。

```

1. # 这是带 * 或 @ 的间接引用的一种变换形式。
2. # 在 Bash 2.04 版本中加入了这个特性。
3.
4. xyz23=whatever
5. xyz23=
6.
7. a=${!xyz*}                # 扩展为声明变量中以 "xyz"
8. # ^ ^ ^                  + 开头变量名。
9. echo "a = $a"              # a = xyz23 xyz24
10. a=${!xyz@}                 # 同上。
11. echo "a = $a"              # a = xyz23 xyz24
12.
13. echo "..."

```

```
14.  
15. abc23=something_else  
16. b=${!abc*}  
17. echo "b = $b"      # b = abc23  
18. c=${!b}            # 这是我们熟悉的间接引用的形式。  
19. echo $c            # something_else
```

[^1]：如果在非交互的脚本中，`$parameter` 为空，那么程序将会终止，并且返回 **错误码 127**（意为“找不到命令”）。

11. 循环与分支

- [第十一章 循环与分支](#)
 - [本章目录](#)

第十一章 循环与分支

奥赛罗夫人，您为什么把这句话说了又说呢？

— 《奥赛罗》，莎士比亚

本章目录

- [11.1 循环](#)
- [11.2 嵌套循环](#)
- [11.3 循环控制](#)
- [11.4 测试与分支](#)

对代码块的处理是结构化和构建 shell 脚本的关键。循环与分支结构恰好提供了这样一种对代码块处理的工具。

11.1 循环

- 11.1 循环
 - for 循环
 - `for arg in [list]`
 - while 循环
 - until

11.1 循环

循环是当循环控制条件为真时，一系列命令迭代^{^1}执行的代码块。

for 循环

```
for arg in [list]
```

这是 shell 中最基本的循环结构，它与C语言形式的循环有着明显的不同。

```
1. for arg in [list]
2. do
3.     command(s)...
4. done
```



在循环的过程中，`arg` 会从 `list` 中连续获得每一个变量的值。

```
1. for arg in "$var1" "$var2" "$var3" ... "$varN"
2. # 第一次循环中, arg = $var1
3. # 第二次循环中, arg = $var2
4. # 第三次循环中, arg = $var3
5. # ...
6. # 第 N 次循环中, arg = $varN
7.
8. # 为了防止可能的字符分割问题, [list] 中的参数都需要被引用。
```


参数 `list` 中允许含有 [通配符](#)。

如果 `do` 和 `for` 写在同一行时，需要在 `list` 之后加上一个分号。

```
for arg in [list] ; do
```

样例 11-1. 简单的 `for` 循环

```
1.  #!/bin/bash
2.  # 列出太阳系的所有行星。
3.
4.  for planet in Mercury Venus Earth Mars Jupiter Saturn Uranus
    Neptune Pluto
5.  do
6.      echo $planet # 每一行输出一个行星。
7.  done
8.
9.  echo; echo
10.
11. for planet in "Mercury Venus Earth Mars Jupiter Saturn Uranus
    Neptune Pluto"
12.     # 所有的行星都输出在一行上。
13.     # 整个 'list' 被包裹在引号中时是作为一个单一的变量。
14.     # 为什么？因为空格也是变量的一部分。
15. do
16.     echo $planet
17. done
18.
19. echo; echo "Whoops! Pluto is no longer a planet!"
20.
21. exit 0
```

`[list]` 中的每一个元素中都可能含有多个参数。这在处理参数组中非常有用。在这种情况下，使用 `set` 命令（查看 [样例 15-16](#)）强制解析 `[list]` 中的每一个元素，并将元素的每一个部分分配给位置参

数。

样例 11-2. `for` 循环 `[list]` 中的每一个变量有两个参数的情况

```

1.  #!/bin/bash
2.  # 让行星再躺次枪。
3.
4.  # 将每个行星与其到太阳的距离放在一起。
5.
6.  for planet in "Mercury 36" "Venus 67" "Earth 93" "Mars 142"
    "Jupiter 483"
7.  do
8.      set -- $planet # 解析变量 "planet"
9.                      #+ 并将其每个部分赋值给位置参数。
10.     # "--" 防止一些极端情况，比如 $planet 为空或者以破折号开头。
11.
12.     # 因为位置参数会被覆盖掉，因此需要先保存原先的位置参数。
13.     # 你可以使用数组来保存
14.     #         original_params=("$@")
15.
16.     echo "$1          $2,000,000 miles from the sun"
17.     #-----两个制表符---将后面的一系列 0 连到参数 $2 上。
18. done
19.
20. # （感谢 S.C. 做出的额外注释。）
21.
22. exit 0

```

一个单一变量也可以成为 `for` 循环中的 `[list]`。

样例 11-3. 文件信息：查看一个单一变量中含有的文件列表的文件信息

```

1.  #!/bin/bash
2.  # fileinfo.sh
3.
4.  FILES="/usr/sbin/accept

```

```

5. /usr/sbin/pwck
6. /usr/sbin/chroot
7. /usr/bin/fakefile
8. /sbin/badbblocks
9. /sbin/ypbind"      # 你可能会感兴趣的一系列文件。
10.                  # 包含一个不存在的文件, /usr/bin/fakefile。
11.
12. echo
13.
14. for file in $FILES
15. do
16.
17.     if [ ! -e "$file" ]      # 检查文件是否存在。
18.     then
19.         echo "$file does not exist."; echo
20.         continue           # 继续判断下一个文件。
21.     fi
22.
23.     ls -l $file | awk '{ print $8 "          file size: " $5 }' # 输出
    其中的两个域。
24.     whatis `basename $file` # 文件信息。
25.     # 脚本正常运行需要注意提前设置好 whatis 的数据。
26.     # 使用 root 权限运行 /usr/bin/makewhatis 可以完成。
27.     echo
28. done
29.
30. exit 0

```

`for` 循环中的 `[list]` 可以是一个参数。

样例 11-4. 操作含有一系列文件的参数

```

1. #!/bin/bash
2.
3. filename="*txt"
4.
5. for file in $filename
6. do

```

```

7.  echo "Contents of $file"
8.  echo "---"
9.  cat "$file"
10. echo
11. done

```

如果在匹配文件扩展名的 `for` 循环中的 `[list]` 含有通配符（`*` 和 `?`），那么将会进行文件名扩展。

样例 11-5. 在 `for` 循环中操作文件

```

1.  #!/bin/bash
2.  # list-glob.sh: 通过文件名扩展在 for 循环中产生 [list]。
3.  # 通配 = 文件名扩展。
4.
5.  echo
6.
7.  for file in *
8.  #           ^ Bash 在检测到通配表达式时，
9.  #+         会进行文件名扩展。
10. do
11.  ls -l "$file" # 列出 $PWD（当前工作目录）下的所有文件。
12.  # 回忆一下，通配符 "*" 会匹配所有的文件名，
13.  #+ 但是，在文件名扩展中，他将不会匹配以点开头的文件。
14.
15.  # 如果没有匹配到文件，那么它将会扩展为它自身。
16.  # 为了防止出现这种情况，需要设置 nullglob 选项。
17.  #+ (shopt -s nullglob)。
18.  # 感谢 S.C.
19. done
20.
21. echo; echo
22.
23. for file in [jx]*
24. do
25.  rm -f $file # 删除当前目录下所有以 "j" 或 "x" 开头的文件。
26.  echo "Removed file \"$file\"".

```

```

27. done
28.
29. echo
30.
31. exit 0

```

如果在 `for` 循环中省略 `in [list]` 部分，那么循环将会遍历位置参数 (`$@`)。样例 A-15 中使用到了这一点。也可以查看 样例 15-17。

样例 11-6. 缺少 `in [list]` 的 `for` 循环

```

1. #!/bin/bash
2.
3. # 尝试在带参数和不带参数两种情况下调用这个脚本，观察发生了什么。
4.
5. for a
6. do
7.   echo -n "$a "
8. done
9.
10. # 缺失 'in list' 的情况下，循环会遍历 '$@'
11. #+（命令行参数列表，包括空格）。
12.
13. echo
14.
15. exit 0

```

可以在 `for` 循环中使用 [命令代换](#) 生成 `[list]`。查看 样例 16-54，样例 11-11 和 样例 16-48。

样例 11-7. 在 `for` 循环中使用命令代换生成 `[list]`

```

1. #!/bin/bash
2. # for-loopcmd.sh: 带命令代换所生成 [list] 的 for 循环
3.

```

```

4.  NUMBERS="9 7 3 8 37.53"
5.
6.  for number in `echo $NUMBERS` # for number in 9 7 3 8 37.53
7.  do
8.      echo -n "$number "
9.  done
10.
11. echo
12. exit 0

```

下面是使用命令代换生成 [list] 的更加复杂的例子。

样例 11-8. 一种替代 `grep` 搜索二进制文件的方法

```

1.  #!/bin/bash
2.  # bin-grep.sh: 在二进制文件中定位匹配的字符串。
3.
4.  # 一种替代 `grep` 搜索二进制文件的方法
5.  # 与 "grep -a" 的效果类似
6.
7.  E_BADARGS=65
8.  E_NOFILE=66
9.
10. if [ $# -ne 2 ]
11. then
12.     echo "Usage: `basename $0` search_string filename"
13.     exit $E_BADARGS
14. fi
15.
16. if [ ! -f "$2" ]
17. then
18.     echo "File \"$2\" does not exist."
19.     exit $E_NOFILE
20. fi
21.
22.
23. IFS=$'\012' # 按照 Anton Filippov 的意见应该是
24.             # IFS="\n"

```

```

25. for word in $( strings "$2" | grep "$1" )
26. # "strings" 命令列出二进制文件中的所有字符串。
27. # 将结果通过管道输出到 "grep" 中，检查是不是匹配的字符串。
28. do
29.     echo $word
30. done
31.
32. # 就像 S.C. 指出的那样，第 23-30 行可以换成下面的形式：
33. #     strings "$2" | grep "$1" | tr -s "$IFS" '[\n*]'
34.
35.
36. # 尝试运行脚本 "./bin-grep.sh mem /bin/ls"
37.
38. exit 0

```

下面的例子同样展示了如何使用命令代换生成 [list]。

样例 11-9. 列出系统中的所有用户

```

1. #!/bin/bash
2. # userlist.sh
3.
4. PASSWORD_FILE=/etc/passwd
5. n=1          # 用户数量
6.
7. for name in $(awk 'BEGIN{fs=":"}{print $1}' < "$PASSWORD_FILE" )
8. # 分隔符 = :          ^^^^^^
9. # 输出第一个域          ^^^^^^^^
10. # 读取密码文件 /etc/passwd          ^^^^^^^^^^^^^^^^^^^^^^^^^^^
11. do
12.     echo "USER #n = $name"
13.     let "n += 1"
14. done
15.
16.
17. # USER #1 = root
18. # USER #2 = bin
19. # USER #3 = daemon

```

```

20. # ...
21. # USER #33 = bozo
22.
23. exit $?
24.
25. # 讨论：
26. # -----
27. # 一个普通用户是如何读取 /etc/passwd 文件的？
28. # 提示：检查 /etc/passwd 的文件权限。
29. # 这算不算是一个安全漏洞？为什么？

```

另外一个关于 `[list]` 的例子也来自于命令代换。

样例 11-10. 检查目录中所有二进制文件的原作者

```

1. #!/bin/bash
2. # findstring.sh
3. # 在指定目录的二进制文件中寻找指定的字符串。
4.
5. directory=/usr/bin
6. fstring="Free Software Foundation" # 查看哪些文件来自于 FSF。
7.
8. for file in $( find $directory -type f -name '*' | sort )
9. do
10.     strings -f $file | grep "$fstring" | sed -e "s%$directory%"
11.     # 在 "sed" 表达式中，你需要替换掉 "/" 分隔符，
12.     #+ 因为 "/" 是一个会被过滤的字符。
13.     # 如果不做替换，将会产生一个错误。（你可以尝试一下。）
14. done
15.
16. exit $?
17.
18. # 简单的练习：
19. # -----
20. # 修改脚本，使其可以从命令行参数中获取 $directory 和 $fstring。

```

最后一个关于 `[list]` 和命令代换的例子，但这个例子中的命令是一

个函数。

```

1. generate_list ()
2. {
3.     echo "one two three"
4. }
5.
6. for word in $(generate_list) # "word" 获得函数执行的结果。
7. do
8.     echo "$word"
9. done
10.
11. # one
12. # two
13. # three

```

`for` 循环的结果可以通过管道导向至一个或多个命令中。

样例 11-11. 列出目录中的所有符号链接。

```

1. #!/bin/bash
2. # symlinks.sh: 列出目录中的所有符号链接。
3.
4. directory=${1-`pwd`}
5. # 如果没有特别指定，缺省目录为当前工作目录。
6. # 等价于下面的代码块。
7. # -----
8. # ARGS=1                # 只有一个命令行参数。
9. #
10. # if [ $# -ne "$ARGS" ] # 如果不是只有一个参数的情况下
11. # then
12. #     directory=`pwd`    # 设为当前工作目录。
13. # else
14. #     directory=$1
15. # fi
16. # -----
17.

```

```

18. echo "symbolic links in directory \"$directory\""
19.
20. for file in "$( find $directory -type 1 )" # -type 1 = 符号链接
21. do
22.     echo "$file"
23. done | sort # 否则文件顺序会是乱序。
24. # 严格的来说这里并不需要使用循环,
25. ## 因为 "find" 命令的输出结果已经被扩展成一个单一字符串了。
26. # 然而, 为了方便大家理解, 我们使用了循环的方式。
27.
28. # Dominik 'Aeneas' Schnitzer 指出,
29. ## 不引用 $( find $directory -type 1 ) 的话,
30. # 脚本将在文件名包含空格时阻塞。
31.
32. exit 0
33.
34.
35. # -----
36. # Jean Helou 提供了另外一种方法:
37.
38. echo "symbolic links in directory \"$directory\""
39. # 备份当前的内部字段分隔符。谨慎永远没有坏处。
40. OLDIFS=$IFS
41. IFS=:
42.
43. for file in $(find $directory -type 1 -printf "%p$IFS")
44. do # ^^^^^^^^^^^^^^^^^^^^^^^^^
45.     echo "$file"
46. done | sort
47.
48. # James "Mike" Conley 建议将 Helou 的代码修改为:
49.
50. OLDIFS=$IFS
51. IFS=' ' # 空的内部字段分隔符意味着将不会分隔任何字符串
52. for file in $( find $directory -type 1 )
53. do
54.     echo $file
55. done | sort

```

```

56.
57. # 上面的代码可以在目录名包含冒号（前一个允许包含空格）
58. #+ 的情况下仍旧正常工作。

```

只需要对上一个样例做一些小小的改动，就可以把在标准输出 `stdout` 中的循环 [重定向](#) 到文件中。

样例 11-12. 将目录中的所有符号链接保存到文件中。

```

1.  #!/bin/bash
2.  # symlinks.sh: 列出目录中的所有符号链接。
3.
4.  OUTFILE=symlinks.list
5.
6.  directory=${1-`pwd`}
7.  # 如果没有特别指定，缺省目录为当前工作目录。
8.
9.
10. echo "symbolic links in directory \"$directory\"" > "$OUTFILE"
11. echo "-----" >> "$OUTFILE"
12.
13. for file in "$( find $directory -type 1 )"    # -type 1 = 符号链接
14. do
15.     echo "$file"
16. done | sort >> "$OUTFILE"                    # 将 stdout 的循环结果
17. #          ^^^^^^^^^^^^^^^^^              重定向到文件。
18.
19. # echo "Output file = $OUTFILE"
20.
21. exit $?

```

还有另外一种看起来非常像C语言中循环那样的语法。你需要使用到 [双圆括号](#) 语法。

样例 11-13. C语言风格的循环

```

1.  #!/bin/bash

```

```
2.  # 用多种方式数到10。
3.
4.  echo
5.
6.  # 基础版
7.  for a in 1 2 3 4 5 6 7 8 9 10
8.  do
9.      echo -n "$a "
10. done
11.
12. echo; echo
13.
14. # +=====+
15.
16. # 使用 "seq"
17. for a in `seq 10`
18. do
19.     echo -n "$a "
20. done
21.
22. echo; echo
23.
24. # +=====+
25.
26. # 使用大括号扩展语法
27. # Bash 3+ 版本有效。
28. for a in {1..10}
29. do
30.     echo -n "$a "
31. done
32.
33. echo; echo
34.
35. # +=====+
36.
37. # 现在用类似C语言的语法再实现一次。
38.
39. LIMIT=10
```

```

40.
41. for ((a=1; a <= LIMIT ; a++)) # 双圆括号语法, 不带 $ 的 LIMIT
42. do
43.     echo -n "$a "
44. done # 从 ksh93 中学习到的特性。
45.
46. echo; echo
47.
48. # +=====+
49.
50. # 我们现在使用C语言中的逗号运算符来使得两个变量同时增加。
51.
52. for ((a=1, b=1; a <= LIMIT ; a++, b++))
53. do # 逗号连接操作。
54.     echo -n "$a-$b "
55. done
56.
57. echo; echo
58.
59. exit 0

```

还可以查看 [样例 27-16](#), [样例 27-17](#) 和 [样例 A-6](#)。

--

接下来, 我们将展示在真实环境中应用的循环。

样例 11-14. 在批处理模式下使用 `efax`

```

1. #!/bin/bash
2. # 传真(必须提前安装了 'efax' 模块)。
3.
4. EXPECTED_ARGS=2
5. E_BADARGS=85
6. MODEM_PORT="/dev/ttyS2" # 你的电脑可能会不一样。
7. #          ^^^^^ PCMCIA 调制解调卡缺省端口。
8.
9. if [ $# -ne $EXPECTED_ARGS ]

```

```

10. # 检查是不是传入了适当数量的命令行参数。
11. then
12.     echo "Usage: `basename $0` phone# text-file"
13.     exit $E_BADARGS
14. fi
15.
16.
17. if [ ! -f "$2" ]
18. then
19.     echo "File $2 is not a text file."
20.     #   File 不是一个正常文件或者文件不存在。
21.     exit $E_BADARGS
22. fi
23.
24.
25. fax make $2                # 根据文本文件创建传真格式文件。
26.
27. for file in $(ls $2.0*)    # 连接转换后的文件。
28.                        # 在参数列表中使用通配符（文件名通配）。
29. do
30.     fil="$fil $file"
31. done
32.
33. efax -d "$MODEM_PORT" -t "T$1" $fil    # 最后使用 efax。
34. # 如果上面一行执行失败，尝试添加 -o1。
35.
36.
37. # S.C. 指出，上面的 for 循环可以被压缩为
38. #   efax -d /dev/ttyS2 -o1 -t "T$1" $2.0*
39. #+ 但是这并不是一个好主意。
40.
41. exit $?    # efax 同时也会将诊断信息传递给标准输出。

```



关键字 `do` 和 `done` 圈定了 `for` 循环代码块的范围。但是在一些特殊的情况下，也可以被 **大括号** 取代。

```

1. for((n=1; n<=10; n++))
2. # 没有 do !

```

```

3. {
4.     echo -n "*" $n "*"
5. }
6. # 没有 done !
7.
8.
9. # 输出 :
10. # * 1 ** 2 ** 3 ** 4 ** 5 ** 6 ** 7 ** 8 ** 9 ** 10 *
11. # 并且 echo $? 返回 0, 因此 Bash 并不认为这是一个错误。
12.
13.
14. echo
15.
16.
17. # 但是注意在典型的 for 循环 for n in [list] ... 中,
18. #+ 需要在结尾加一个分号。
19.
20. for n in 1 2 3
21. { echo -n "$n "; }
22. #           ^
23.
24.
25. # 感谢 Yongye 指出这一点。

```

while 循环

`while` 循环结构会在循环顶部检测循环条件，若循环条件为真（[退出状态](#) 为 0）则循环持续进行。与 `for` [循环](#) 不同的是，`while` 循环是在不知道循环次数的情况下使用的。

```

1. while [ condition ]
2. do
3.     command(s)...
4. done

```

在 `while` 循环结构中，你不仅可以使使用像 `if/test` 中那样的 [括号结构](#)，也可以使用用途更广泛的 [双括号结构](#)（`while [[condition]]`）。

就像在 `for` 循环中那样，将 `do` 和循环条件放在同一行时需要加一个分号。

```
while [ condition ] ; do
```

在 `while` 循环中，括号结构 [并不是必须存在的](#)。比如说 `getopts` 结构。

样例 11-15. 简单的 `while` 循环

```
1.  #!/bin/bash
2.
3.  var0=0
4.  LIMIT=10
5.
6.  while [ "$var0" -lt "$LIMIT" ]
7.  #      ^                ^
8.  #  必须有空格，因为这是测试结构
9.  do
10.    echo -n "$var0 "      # -n 不会另起一行
11.    #      ^            空格用来分开输出的数字。
12.
13.    var0=`expr $var0 + 1`  # var0=$((var0+1)) 效果相同。
14.                        # var0=$((var0 + 1)) 效果相同。
15.                        # let "var0 += 1" 效果相同。
16.  done                    # 还有许多其他的方法也可以达到相同的效果。
17.
18.  echo
19.
20.  exit 0
```

样例 11-16. 另一个例子

```
1.  #!/bin/bash
2.
3.  echo
4.                                # 等价于：
```



```

5. while [ "$var1" != "end" ]      # while test "$var1" != "end"
6. do
7.     echo "Input variable #1 (end to exit) "
8.     read var1                  # 不是 'read $var1' (为什么?)。
9.     echo "variable #1 = $var1"  # 因为存在 "#", 所以需要使用引号。
10.    # 如果输入的是 "end", 也将会在这里输出。
11.    # 在结束本轮循环之前都不会再测试循环条件了。
12.    echo
13. done
14.
15. exit 0

```

一个 `while` 循环可以有多个测试条件，但只有最后的那一个条件决定了循环是否终止。这是一种你需要注意到的不同于其他循环的语法。

样例 11-17. 多条件 `while` 循环

```

1. #!/bin/bash
2.
3. var1=unset
4. previous=$var1
5.
6. while echo "previous-variable = $previous"
7.     echo
8.     previous=$var1
9.     [ "$var1" != end ] # 记录下 $var1 之前的值。
10.    # 在 while 循环中有4个条件，但只有最后的那个控制循环。
11.    # 最后一个条件的退出状态才会被记录。
12. do
13.     echo "Input variable #1 (end to exit) "
14.     read var1
15.     echo "variable #1 = $var1"
16. done
17.
18. # 猜猜这是怎样实现的。
19. # 这是一个很小的技巧。
20.

```

```
21. exit 0
```

就像 `for` 循环一样，`while` 循环也可以使用双圆括号结构写得像C语言那样（也可以查看[样例 8-5](#)）。

样例 11-18. C语言风格的 `while` 循环

```
1. #!/bin/bash
2. # wh-loopc.sh: 在 "while" 循环中计数到10。
3.
4. LIMIT=10                # 循环10次。
5. a=1
6.
7. while [ "$a" -le $LIMIT ]
8. do
9.     echo -n "$a "
10.    let "a+=1"
11. done                    # 没什么好奇怪的吧。
12.
13. echo; echo
14.
15. # +=====+
16.
17. # 现在我们用C语言风格再写一次。
18.
19. ((a = 1))               # a=1
20. # 双圆括号结构允许像C语言一样在赋值语句中使用空格。
21.
22. while (( a <= LIMIT ))  # 双圆括号结构，
23. do                      #+ 并且没有使用 "$"。
24.     echo -n "$a "
25.     ((a += 1))          # let "a+=1"
26.     # 是的，就是这样。
27.     # 双圆括号结构允许像C语言一样自增一个变量。
28. done
29.
30. echo
```

```

31.
32. # 这可以让C和Java程序员感觉更加舒服。
33.
34. exit 0

```

在测试部分，`while` 循环可以调用 [函数](#)。

```

1. t=0
2.
3. condition ()
4. {
5.     ((t++))
6.
7.     if [ $t -lt 5 ]
8.     then
9.         return 0 # true 真
10.    else
11.        return 1 # false 假
12.    fi
13. }
14.
15. while condition
16. #     ^^^^^^^^^^^
17. #     调用函数循环四次。
18. do
19.     echo "Still going: t = $t"
20. done
21.
22. # Still going: t = 1
23. # Still going: t = 2
24. # Still going: t = 3
25. # Still going: t = 4

```

和 `if 测试` 结构一样，`while` 循环也可以省略括号。

```

1. while condition
2. do
3.     command(s) ...

```

4. done

在 `while` 循环中结合 `read` 命令，我们就得到了一个非常易于使用的 `while read` 结构。它可以用来读取和解析文件。

```

1. cat $filename |      # 从文件获得输入。
2. while read line      # 只要还有可以读入的行，循环就继续。
3. do
4.     ...
5. done
6.
7. # ===== 摘自样例脚本 "sd.sh" ===== #
8.
9. while read value     # 一次读入一个数据。
10. do
11.     rt=$(echo "scale=$SC; $rt + $value" | bc)
12.     (( ct++ ))
13. done
14.
15. am=$(echo "scale=$SC; $rt / $ct" | bc)
16.
17. echo $am; return $ct  # 这个功能“返回”了2个值。
18. # 注意：这个技巧在 $ct > 255 的情况下会失效。
19. # 如果要操作更大的数字，注释掉上面的 "return $ct" 就可以了。
20. } <"$datafile"      # 传入数据文件。

```



在 `while` 循环后面可以通过 `<` 将标准输入 [重定位到文件](#) 中。

`while` 循环同样可以 [通过管道](#) 传入标准输入中。

until

与 `while` 循环相反，`until` 循环测试其顶部的循环条件，直到其中的条件为真时停止。

```

1. until [ condition-is-true ]
2. do

```

```
3.   commands(s)...
4.  done
```

注意到，跟其他的一些编程语言不同，`until` 循环的测试条件在循环顶部。

就像在 `for` 循环中那样，将 `do` 和循环条件放在同一行时需要加一个分号。

```
until[ condition-is-true ] ; do
```

样例 11-19. `until` 循环

```
1.  #!/bin/bash
2.
3.  END_CONDITION=end
4.
5.  until [ "$var1" = "$END_CONDITION" ]
6.  # 在循环顶部测试条件。
7.  do
8.    echo "Input variable #1 "
9.    echo "($END_CONDITION to exit)"
10.   read var1
11.   echo "variable #1 = $var1"
12.   echo
13. done
14.
15. #           ---           #
16.
17. # 就像 "for" 和 "while" 循环一样，
18. #+ "until" 循环也可以写的像C语言一样。
19.
20. LIMIT=10
21. var=0
22.
23. until (( var > LIMIT ))
24. do # ^^ ^      ^      ^^  没有方括号，没有 $ 前缀。
25.   echo -n "$var "
```

```
26.    (( var++ ))  
27. done    # 0 1 2 3 4 5 6 7 8 9 10  
28.  
29.  
30. exit 0
```

如何在 `for` ， `while` 和 `until` 之间做出选择？我们知道在C语言中，在已知循环次数的情况下更加倾向于使用 `for` 循环。但是在 Bash 中情况可能更加复杂一些。Bash 中的 `for` 循环相比起其他语言来说，结构更加松散，使用更加灵活。因此使用你认为最简单的就好。

11.2 嵌套循环

- 11.2 嵌套循环

11.2 嵌套循环

嵌套循环，顾名思义就是在循环里面还有循环。外层循环会不断的触发内层循环直到外层循环结束。当然，你仍然可以使用 `break` 可以终止外层或内层的循环。

样例 11-20. 嵌套循环

```
1.  #!/bin/bash
2.  # nested-loop.sh: 嵌套 "for" 循环。
3.
4.  outer=1                # 设置外层循环计数器。
5.
6.  # 外层循环。
7.  for a in 1 2 3 4 5
8.  do
9.      echo "Pass $outer in outer loop."
10.     echo "-----"
11.     inner=1             # 重设内层循环计数器。
12.
13.     # =====
14.     # 内层循环。
15.     for b in 1 2 3 4 5
16.     do
17.         echo "Pass $inner in inner loop."
18.         let "inner+=1" # 增加内层循环计数器。
19.     done
20.     # 内层循环结束。
21.     # =====
22.
23.     let "outer+=1"      # 增加外层循环计数器。
24.     echo                # 在每次外层循环输出中加入空行。
```

```
25. done
26. # 外层循环结束。
27.
28. exit 0
```

查看 [样例 27-11](#) 详细了解嵌套 `while` 循环。查看 [样例 27-13](#) 详细了解嵌套 `until` 循环。

11.3 循环控制

- 11.3 循环控制
 - `break`, `continue`

11.3 循环控制

*Tournez cent tours, tournez mille tours,
Tournez souvent et tournez toujours . . .*

—保尔·魏尔伦，《木马》

本节介绍两个会影响循环行为的命令。

`break`, `continue`

`break` 和 `continue` 命令^[^1]的作用和在其他编程语言中的作用一样。`break` 用来中止（跳出）循环，而 `continue` 则是略过未执行的循环部分，直接进行下一次循环。

样例 11-21. 循环中 `break` 与 `continue` 的作用

```
1.  #!/bin/bash
2.
3.  LIMIT=19 # 循环上界
4.
5.  echo
6.  echo "Printing Numbers 1 through 20 (but not 3 and 11)."
```

```
7.
8.  a=0
9.
10. while [ $a -le "$LIMIT" ]
11. do
12.     a=$((a+1))
13.
```

```

14.  if [ "$a" -eq 3 ] || [ "$a" -eq 11 ] # 除了 3 和 11。
15.  then
16.      continue      # 略过本次循环的剩余部分。
17.  fi
18.
19.  echo -n "$a " # 当 a 等于 3 和 11 时，将不会执行这条语句。
20.  done
21.
22.  # 思考：
23.  # 为什么循环不会输出到20？
24.
25.  echo; echo
26.
27.  echo Printing Numbers 1 through 20, but something happens after 2.
28.
29.  #####
30.
31.  # 用 'break' 代替了 'continue'。
32.
33.  a=0
34.
35.  while [ "$a" -le "$LIMIT" ]
36.  do
37.      a=$((a+1))
38.
39.      if [ "$a" -gt 2 ]
40.      then
41.          break # 中止循环。
42.      fi
43.
44.      echo -n "$a"
45.  done
46.
47.  echo; echo; echo
48.
49.  exit 0

```

break

命令接受一个参数。普通的

break

命令仅仅跳出其所在的

那层循环，而 `break N` 命令则可以跳出其上 N 层的循环。

样例 11-22. 跳出多层循环

```

1.  #!/bin/bash
2.  # break-levels.sh: 跳出循环.
3.
4.  # "break N" 跳出 N 层循环。
5.
6.  for outerloop in 1 2 3 4 5
7.  do
8.      echo -n "Group $outerloop:  "
9.
10.     # -----
11.     for innerloop in 1 2 3 4 5
12.     do
13.         echo -n "$innerloop "
14.
15.         if [ "$innerloop" -eq 3 ]
16.         then
17.             break # 尝试一下 break 2 看看会发生什么。
18.                 # （它同时中止了内层和外层循环。）
19.         fi
20.     done
21.     # -----
22.
23.     echo
24. done
25.
26. echo
27.
28. exit 0

```

与 `break` 类似，`continue` 也接受一个参数。普通的 `continue` 命令仅仅影响其所在的那层循环，而 `continue N` 命令则可以影响其上 N 层的循环。

样例 11-23. `continue` 影响外层循环

```

1.  #!/bin/bash
2.  # "continue N" 命令可以影响其上 N 层循环。
3.
4.  for outer in I II III IV V          # 外层循环
5.  do
6.      echo; echo -n "Group $outer: "
7.
8.      # -----
9.      for inner in 1 2 3 4 5 6 7 8 9 10 # 内层循环
10.     do
11.
12.         if [[ "$inner" -eq 7 && "$outer" = "III" ]]
13.         then
14.             continue 2 # 影响两层循环，包括“外层循环”。
15.                         # 将其替换为普通的 "continue"，那么只会影响内层循环。
16.         fi
17.
18.         echo -n "$inner " # 7 8 9 10 将不会出现在 "Group III."中。
19.     done
20.     # -----
21.
22. done
23.
24. echo; echo
25.
26. # 思考：
27. # 想一个 "continue N" 在脚本中的实际应用情况。
28.
29. exit 0

```

样例 11-24. 真实环境中的 `continue N`

1. # Albert Reiner 举出了一个如何使用 "continue N" 的例子：

```

2.  # -----
3.
4.  # 如果我有许多任务需要运行，并且运行所需要的数据都以文件的形
5.  #+ 式存在文件夹中。现在有多台设备可以访问这个文件夹，我想将任
6.  #+ 务分配给这些不同的设备来完成。
7.  # 那么我通常会在每台设备上执行下面的代码：
8.
9.  while true:
10. do
11.     for n in .iso.*
12.     do
13.         [ "$n" = ".iso.opts" ] && continue
14.         beta=${n#.iso.}
15.         [ -r .Iso.$beta ] && continue
16.         [ -r .lock.$beta ] && sleep 10 && continue
17.         lockfile -r0 .lock.$beta || continue
18.         echo -n "$beta: " `date`
19.         run-isotherm $beta
20.         date
21.         ls -alF .Iso.$beta
22.         [ -r .Iso.$beta ] && rm -rf .lock.$beta
23.         continue 2
24.     done
25.     break
26. done
27.
28. exit 0
29.
30. # 这个脚本中出现的 sleep N 只针对这个脚本，通常的形式是：
31.
32. while true
33. do
34.     for job in {pattern}
35.     do
36.         {job already done or running} && continue
37.         {mark job as running, do job, mark job as done}
38.         continue 2
39.     done

```

```

40.     break          # 或者使用类似 `sleep 600` 这样的语句来防止脚本结束。
41. done
42.
43. # 这样做可以保证脚本只会在没有任务时（包括在运行过程中添加的任务）
44. #+ 才会停止。合理使用文件锁保证多台设备可以无重复的并行执行任务（这
45. #+ 在我的设备上通常会消耗好几个小时，所以我想避免重复计算）。并且，
46. #+ 因为每次总是从头开始搜索文件，因此可以通过文件名决定执行的先后
47. #+ 顺序。当然，你可以不使用 'continue 2' 来完成这些，但是你必须
48. #+ 添加代码去检测某项任务是否完成（以此判断是否可以执行下一项任务或
49. #+ 终止、休眠一段时间再执行下一项任务）。

```



`continue N` 结构不易理解并且可能在一些情况下有歧义，因此不建议使用。

[^1]：这两个命令是 **内建命令**，而另外的循环命令，如 `while` 和 `case` 则是 **关键词**。

11.4 测试与分支

• 11.4 测试与分支

- `case (in)` / `esac`
- `select`

11.4 测试与分支

`case` 和 `select` 结构并不属于循环结构，因为它们并没有反复执行代码块。但是和循环结构相似的是，它们会根据代码块顶部或尾部的条件控制程序流。

下面介绍两种在代码块中控制程序流的方法：

`case (in)` / `esac`

在 shell 脚本中，`case` 模拟了 C/C++ 语言中的 `switch`，可以根据条件跳转到其中一个分支。其相当于简写版的 `if/then/else` 语句。很适合用来创建菜单选项哟！

```
1. case "$variable" in
2.   "$condition1" )
3.     command...
4.   ;;
5.   "$condition2" )
6.     command...
7.   ;;
8. esac
```



- 对变量进行引用不是必须的，因为在这里不会进行字符分割。
- 条件测试语句必须以右括号 `)` 结束。[^1]

- 每一段代码块都必须以双分号 `;;` 结束。
- 如果测试条件为真，其对应的代码块将被执行，而后整个 `case` 代码段结束执行。
- `case` 代码段必须以 `esac` 结束（倒着拼写 `case`）。

样例 11-25. 如何使用 `case`

```

1. #!/bin/bash
2. # 测试字符的种类。
3.
4. echo; echo "Hit a key, then hit return."
5. read Keypress
6.
7. case "$Keypress" in
8.     [:lower:] ) echo "Lowercase letter";;
9.     [:upper:] ) echo "Uppercase letter";;
10.    [0-9]      ) echo "Digit";;
11.    *          ) echo "Punctuation, whitespace, or other";;
12. esac        # 字符范围可以用[方括号]表示，也可以用 POSIX 形式的[[双方括号]]
               表示。
13.
14. # 在这个例子的第一个版本中，用来测试是小写还是大写字符使用的是 [a-z] 和 [A-
    Z]。
15. # 这在一些特定的语言环境和 Linux 发行版中不起效。
16. # POSIX 形式具有更好的兼容性。
17. # 感谢 Frank Wang 指出这一点。
18.
19. # 练习：
20. # -----
21. # 这个脚本接受一个单字符然后结束。
22. # 修改脚本，使得其可以循环接受输入，并且检测键入的每一个字符，直到键入 "x" 为
    止。
23. # 提示：将所有东西包在 "while" 中。
24.
25. exit 0

```

样例 11-26. 使用 `case` 创建菜单


```
1.  #!/bin/bash
2.
3.  # 简易的通讯录数据库
4.
5.  clear # 清屏。
6.
7.  echo "          Contact List"
8.  echo "          -----"
9.  echo "Choose one of the following persons:"
10. echo
11. echo "[E]vans, Roland"
12. echo "[J]ones, Mildred"
13. echo "[S]mith, Julie"
14. echo "[Z]ane, Morris"
15. echo
16.
17. read person
18.
19. case "$person" in
20. # 注意变量是被引用的。
21.
22.     "E" | "e" )
23.     # 同时接受大小写的输入。
24.     echo
25.     echo "Roland Evans"
26.     echo "4321 Flash Dr."
27.     echo "Hardscrabble, CO 80753"
28.     echo "(303) 734-9874"
29.     echo "(303) 734-9892 fax"
30.     echo "revans@zzy.net"
31.     echo "Business partner & old friend"
32.     ;;
33. # 注意用双分号结束这一个选项。
34.
35.     "J" | "j" )
36.     echo
37.     echo "Mildred Jones"
38.     echo "249 E. 7th St., Apt. 19"
```

```

39.     echo "New York, NY 10009"
40.     echo "(212) 533-2814"
41.     echo "(212) 533-9972 fax"
42.     echo "milliej@loisaida.com"
43.     echo "Ex-girlfriend"
44.     echo "Birthday: Feb. 11"
45.     ;;
46.
47.     # Smith 和 Zane 的信息稍后添加。
48.
49.     *          )
50.     # 缺省设置。
51.     # 空输入（直接键入回车）也是执行这一部分。
52.     echo
53.     echo "Not yet in database."
54.     ;;
55.
56. esac
57.
58. echo
59.
60. # 练习：
61. # -----
62. # 修改脚本，使得其可以循环接受多次输入而不是只显示一个地址后终止脚本。
63.
64. exit 0

```

你可以用 `case` 来检测命令行参数。

```

1.  #!/bin/bash
2.
3.  case "$1" in
4.      "") echo "Usage: ${0##*/} <filename>"; exit $E_PARAM;;
5.          # 没有命令行参数，或者第一个参数为空。
6.          # 注意 ${0##*/} 是参数替换 ${var##pattern} 的一
           种形式。
7.          # 最后的结果是 $0.
8.

```

```

9.     -*) FILENAME=./$1;; # 如果传入的参数以短横线开头，那么将其替换为 ./$1
10.                                     #+ 以避免后续的命令将其解释为一个选项。
11.
12.     * ) FILENAME=$1;;    # 否则赋值为 $1。
13. esac

```

下面是一个更加直观的处理命令行参数的例子：

```

1.  #!/bin/bash
2.
3.  while [ $# -gt 0 ]; do    # 遍历完所有参数
4.      case "$1" in
5.          -d|--debug)
6.              # 检测是否是 "-d" 或者 "--debug"。
7.              DEBUG=1
8.              ;;
9.          -c|--conf)
10.             CONFFILE="$2"
11.             shift
12.             if [ ! -f $CONFFILE ]; then
13.                 echo "Error: Supplied file doesn't exist!"
14.                 exit $E_CONFFILE    # 找不到文件。
15.             fi
16.             ;;
17.         esac
18.         shift    # 检测下一个参数
19.     done
20.
21. # 摘自 Stefano Falsetto 的 "Log2Rot" 脚本中 "roottlog" 包的一部分。
22. # 已授权使用。

```

样例 11-27. 使用命令替换生成 `case` 变量

```

1.  #!/bin/bash
2.  # case-cmd.sh: 使用命令替换生成 "case" 变量。
3.
4.  case $( arch ) in    # $( arch ) 返回设备架构。

```

```

5.          # 等价于 'uname -m'。
6.    i386 ) echo "80386-based machine";;
7.    i486 ) echo "80486-based machine";;
8.    i586 ) echo "Pentium-based machine";;
9.    i686 ) echo "Pentium2+-based machine";;
10.   *    ) echo "Other type of machine";;
11. esac
12.
13. exit 0

```

`case` 还可以用来做字符串模式匹配。

样例 11-28. 简单的字符串匹配

```

1.  #!/bin/bash
2.  # match-string.sh: 使用 'case' 结构进行简单的字符串匹配。
3.
4.  match_string ()
5.  { # 字符串精确匹配。
6.    MATCH=0
7.    E_NOMATCH=90
8.    PARAMS=2      # 需要2个参数。
9.    E_BAD_PARAMS=91
10.
11.    [ $# -eq $PARAMS ] || return $E_BAD_PARAMS
12.
13.    case "$1" in
14.        "$2") return $MATCH;;
15.        *    ) return $E_NOMATCH;;
16.    esac
17.
18. }
19.
20.
21. a=one
22. b=two
23. c=three
24. d=two

```

```

25.
26. match_string $a      # 参数个数不够
27. echo $?              # 91
28.
29. match_string $a $b   # 匹配不到
30. echo $?              # 90
31.
32. match_string $a $d   # 匹配成功
33. echo $?              # 0
34.
35.
36. exit 0

```

样例 11-29. 检查输入

```

1.  #!/bin/bash
2.  # isalpha.sh: 使用 "case" 结构检查输入。
3.
4.  SUCCESS=0
5.  FAILURE=1    # 以前是FAILURE=-1,
6.               #+ 但现在 Bash 不允许返回负值。
7.
8.  isalpha () # 测试字符串的第一个字符是否是字母。
9.  {
10. if [ -z "$1" ]                # 检测是否传入参数。
11. then
12.     return $FAILURE
13. fi
14.
15. case "$1" in
16.     [a-zA-Z]*) return $SUCCESS;; # 是否以字母形式开始?
17.     *) return $FAILURE;;
18. esac
19. } # 可以与 C 语言中的函数 "isalpha ()" 作比较。
20.
21.
22. isalpha2 () # 测试整个字符串是否都是字母。
23. {

```

```

24.  [ $# -eq 1 ] || return $FAILURE
25.
26.  case $1 in
27.      *[^a-zA-Z]*|") return $FAILURE;;
28.      *) return $SUCCESS;;
29.  esac
30. }
31.
32. isdigit ()    # 测试整个字符串是否都是数字。
33. {            # 换句话说，也就是测试是否是一个整型变量。
34.  [ $# -eq 1 ] || return $FAILURE
35.
36.  case $1 in
37.      *[^0-9]*|") return $FAILURE;;
38.      *) return $SUCCESS;;
39.  esac
40. }
41.
42.
43.
44. check_var () # 包装后的 isalpha ()。
45. {
46.  if isalpha "$@"
47.  then
48.      echo "\"$*\" begins with an alpha character."
49.      if isalpha2 "$@"
50.      then          # 其实没必要检查第一个字符是不是字母。
51.          echo "\"$*\" contains only alpha characters."
52.      else
53.          echo "\"$*\" contains at least one non-alpha character."
54.      fi
55.  else
56.      echo "\"$*\" begins with a non-alpha character."
57.          # 如果没有传入参数同样返回“存在非字母”。
58.  fi
59.
60.  echo
61.

```

```
62. }
63.
64. digit_check () # 包装后的 isdigit ()。
65. {
66.     if isdigit "$@"
67.     then
68.         echo "\"$*" contains only digits [0 - 9].\"
69.     else
70.         echo "\"$*" has at least one non-digit character.\"
71.     fi
72.
73.     echo
74.
75. }
76.
77.
78. a=23skidoo
79. b=H3llo
80. c=-What?
81. d=What?
82. e=$(echo $b) # 命令替换。
83. f=AbcDef
84. g=27234
85. h=27a34
86. i=27.34
87.
88. check_var $a
89. check_var $b
90. check_var $c
91. check_var $d
92. check_var $e
93. check_var $f
94. check_var # 如果不传入参数会发送什么？
95. #
96. digit_check $g
97. digit_check $h
98. digit_check $i
99.
```

```

100.
101.  exit 0          # S.C. 改进了本脚本。
102.
103.  # 练习：
104.  # -----
105.  # 写一个函数 'isfloat ()' 来检测输入值是否是浮点数。
106.  # 提示：可以参考函数 'isdigit ()'，在其中加入检测合法的小数点即可。

```

select

select 结构是学习自 Korn Shell。其同样可以用来构建菜单。

```

1.  select variable [in list]
2.  do
3.      command...
4.  break
5.  done

```

而效果则是终端会提示用户输入列表中的一个选项。注意，`select` 默认使用提示字符串3 (Prompt String 3, `$PS3`，即#?)，但同样可以被修改。

样例 11-30. 使用 `select` 创建菜单

```

1.  #!/bin/bash
2.
3.  PS3='Choose your favorite vegetable: ' # 设置提示字符串。
4.                                     # 否则默认为 #?。
5.
6.  echo
7.
8.  select vegetable in "beans" "carrots" "potatoes" "onions"
9.      "rutabagas"
10. do
11.     echo "Your favorite veggie is $vegetable."
12.     echo "Yuck!"

```



```

13.     echo
14.     break # 如果没有 'break' 会发生什么？
15. done
16.
17. exit
18.
19. # 练习：
20. # -----
21. # 修改脚本，使得其可以接受其他输入而不是 "select" 语句中所指定的。
22. # 例如，如果用户输入 "peas,"，那么脚本会通知用户 "Sorry. That is not on
    the menu."

```

如果 *in list* 被省略，那么 `select` 将会使用传入脚本的命令行参数 (`$@`) 或者传入函数的参数作为 *list*。

可以与 `for variable [in list]` 中 *in list* 被省略的情况做比较。

样例 11-31. 在函数中使用 `select` 创建菜单

```

1.  #!/bin/bash
2.
3.  PS3='Choose your favorite vegetable: '
4.
5.  echo
6.
7.  choice_of()
8.  {
9.      select vegetable
10.     # [in list] 被省略，因此 'select' 将会使用传入函数的参数作为 list。
11.     do
12.         echo
13.         echo "Your favorite veggie is $vegetable."
14.         echo "Yuck!"
15.         echo
16.         break
17.     done
18. }

```

```

19.
20. choice_of beans rice carrorts radishes rutabaga spinach
21. #          $1    $2    $3      $4      $5      $6
22. #          传入了函数 choice_of()
23.
24. exit 0

```

还可以参照 [样例37-3](#)。

[^1]：在写匹配行的时候，可以在左边加上左括号（，使整个结构看起来更加优雅。

```

1. case $( arch ) in    # $( arch ) 返回设备架构。
    ( i386 ) echo "80386-based machine";;
    # ^          ^
    ( i486 ) echo "80486-based machine";;
    ( i586 ) echo "Pentium-based machine";;
    ( i686 ) echo "Pentium2+-based machine";;
    (      * ) echo "Other type of machine";;
esac

```

12. 命令替换

- 第十二章 命令替换

第十二章 命令替换

命令替换重新指定一个`[^1]`或多个命令的输出。其实就是将命令的输出导出到另外一个地方`^2`。

命令替换的通常形式是 (``...``)，即用反引号引用命令。

```
1. script_name=`basename $0`
2. echo "The name of this script is $script_name."
```

命令的输出可以作为另一个命令的参数，也可以赋值给一个变量。甚至在 `for` 循环中可以用输出产生参数表。

```
1. rm `cat filename`    # "filename" 中包含了一系列需要被删除的文件名。
2. #
3. # S.C. 指出这样写可能会导致出现 "arg list too long" 的错误。
4. # 更好的写法应该是 xargs rm -- < filename
5. # ( -- 可以在 "filename" 文件名以 "-" 为开头时仍旧正常执行 )
6.
7. textfile_listing=`ls *.txt`
8. # 变量中包含了当前工作目录下所有的名为 *.txt 的文件。
9. echo $textfile_listing
10.
11. textfile_listing2=$(ls *.txt)    # 命令替换的另一种形式。
12. echo $textfile_listing2
13. # 结果相同。
14.
15. # 这样将一系列文件名赋值给一个单一字符串可能会出现换行。
16. #
17. # 而更加安全的方式是将这一系列文件存入数组。
18. #      shopt -s nullglob        # 设置后，如果没有匹配到文件，那么变量会被赋值
```

为空。

```
19. #      textfile_listing=( *.txt )
20. #
21. # 感谢 S.C.
```



命令替换本质上是调用了一个 **子进程** 来执行。



命令替换有可能出现 **字符分割** 的情况。

```
1. COMMAND `echo a b`      # 2个参数：a和b
2.
3. COMMAND "`echo a b`"    # 1个参数："a b"
4.
5. COMMAND `echo`          # 没有参数
6.
7. COMMAND "`echo`"        # 一个空参数
8.
9.
10. # 感谢 S.C.
```

但即使不存在字符分割的情况，使用命令替换也会出现丢失尾部换行符的情况。

```
1. # cd "`pwd`" # 你是不是认为这条语句在任何情况下都不会出现错误？
2. # 但事实却不是这样的。
3.
4. mkdir 'dir with trailing newline
5. '
6.
7. cd 'dir with trailing newline
8. '
9.
10. cd "`pwd`" # Bash 会出现如下错误提示：
11. # bash: cd: /tmp/file with trailing newline: No such file or
    directory
12.
13. cd "$PWD" # 这样写是对的。
14.
15.
16.
17.
18.
```

```

19. old_tty_setting=$(stty -g)    # 保存旧的设置。
20. echo "Hit a key "
21. stty -icanon -echo            # 禁用终端的 canonical 模式。
22.                               # 同时禁用 echo。
23. key=$(dd bs=1 count=1 2> /dev/null) # 使用 'dd' 获得键值。
24. stty "$old_tty_setting"        # 恢复旧的设置。
25. echo "You hit ${#key} key."    # ${#variable} 表示 $variable 中的字符个数。
26. #
27. # 除了按下回车键外，其余情况都会输出 "You hit 1 key."
28. # 按下回车键会输出 "You hit 0 key."
29. # 因为唯一的换行符在命令替换中被丢失了。
30.
31. # 这段代码摘自 Stéphane Chazelas。

```



使用 `echo` 输出未被引用的命令代换的变量时会删掉尾部的换行。这可能会导致非常不好的情况出现。

```

1. dir_listing=`ls -l`
2. echo $dir_listing    # 未被引用
3.
4. # 你希望会出现按行显示出文件列表。
5.
6. # 但是，你却看到了：
7. # total 3 -rw-rw-r-- 1 bozo bozo 30 May 13 17:15 1.txt -rw-rw-r-
   # - 1 bozo
8. # bozo 51 May 15 20:57 t2.sh -rwxr-xr-x 1 bozo bozo 217 Mar 5
   # 21:13 wi.sh
9.
10. # 所有换行都消失了。
11.
12.
13. echo "$dir_listing" # 被引用
14. # -rw-rw-r--      1 bozo      30 May 13 17:15 1.txt
15. # -rw-rw-r--      1 bozo      51 May 15 20:57 t2.sh
16. # -rwxr-xr-x      1 bozo      217 Mar  5 21:13 wi.sh

```

你甚至可以使用 [重定向](#) 或者 `cat` 命令把一个文件的内容通过命令代换赋值给一个变量。

```

1. variable1=<file1`      # 将 "file1" 的内容赋值给 variable1。
2. variable2=`cat file2`  # 将 "file2" 的内容赋值给 variable2。
3.                        # 使用 cat 命令会开一个新进程，因此执行速度会比重
    定向慢。
4.
5. # 需要注意的是，这些变量中可能包含一些空格或者控制字符。
6.
7. # 无需显示的赋值给一个变量。
8. echo "` <$0`"         # 输出脚本自身。

```

```

1. # 摘录自系统文件 /etc/rc.d/rc.sysinit
2. #+ (Red Hat Linux 发行版)
3.
4.
5. if [ -f /fsckoptions ]; then
6.     fsckoptions=`cat /fsckoptions`
7. ...
8. fi
9. #
10. #
11. if [ -e "/proc/ide/${disk[$device]}/media" ] ; then
12.     hdmedia=`cat /proc/ide/${disk[$device]}/media`
13. ...
14. fi
15. #
16. #
17. if [ ! -n "`uname -r | grep -- "-"``" ]; then
18.     ktag="`cat /proc/version`"
19. ...
20. fi
21. #
22. #
23. if [ $usb = "1" ]; then
24.     sleep 5
25.     mouseoutput=`cat /proc/bus/usb/devices 2>/dev/null|grep -E
        "^I.*Cls=03.*Prot=02"`
26.     kbdoutput=`cat /proc/bus/usb/devices 2>/dev/null|grep -E

```

```

    "^I.*Cls=03.*Prot=01" `
27. ...
28. fi

```



尽量不要将一大段文字赋值给一个变量，除非你有足够的理由。也绝不要将一个二进制文件的内容赋值给一个变量。

样例 12-1. 蠢蠢的脚本

```

1. #!/bin/bash
2. # stupid-script-tricks.sh: 不要在自己的电脑上尝试。
3. # 摘自 "Stupid Script Tricks" 卷一。
4.
5. exit 99 ### 如果你有胆，就注释掉这行。:)
6.
7. dangerous_variable=`cat /boot/vmlinuz` # 压缩的 Linux 内核。
8.
9. echo "string-length of \${dangerous_variable} =
    \${#dangerous_variable}"
10. # ${dangerous_variable} 的长度为 794151
11. # （更新版本的内核可能更大。）
12. # 与 'wc -c /boot/vmlinuz' 的结果不同。
13.
14. # echo "${dangerous_variable}"
15. # 不要作死。否则脚本会挂起。
16.
17.
18.
19. # 将二进制文件的内容赋值给一个变量没有任何意义。
20.
21. exit 0

```

尽管脚本会挂起，但并不会出现缓存溢出的情况。而这正是像 *Bash* 这样的解释型语言相比起编译型语言能够提供更多保护的一个例子。

命令替换允许将 **循环** 的输出结果赋值给一个变量。这其中的关键在于循环内部的 `echo` 命令。

样例 12-2. 将循环的输出结果赋值给变量

```

1.  #!/bin/bash
2.  # csubloop.sh: 将循环的输出结果赋值给变量。
3.
4.  variable1=`for i in 1 2 3 4 5
5.  do
6.      echo -n "$i"                # 在这里, 'echo' 命令非常关键。
7.  done`
8.
9.  echo "variable1 = $variable1" # variable1 = 12345
10.
11.
12.  i=0
13.  variable2=`while [ "$i" -lt 10 ]
14.  do
15.      echo -n "$i"                # 很关键的 'echo'。
16.      let "i += 1"                # i 自增。
17.  done`
18.
19.  echo "variable2 = $variable2" # variable2 = 0123456789
20.
21.  # 这个例子表明可以在变量声明时嵌入循环。
22.
23.  exit 0

```

命令替换能够让 *Bash* 做更多的事情。而这仅仅需要在书写程序或者脚本时将结果输出到标准输出 `stdout` 中, 然后将这些输出结果赋值给变量即可。

```

1.  #include <stdio.h>
2.
3.  /* "Hello, world." C program */
4.
5.  int main()
6.  {
7.      printf( "Hello, world.\n" );
8.      return (0);
9.  }

```

```
1. bash$ gcc -0 hello hello.c
```



```

1. #!/bin/bash
2. # hello.sh
3.
4. greeting=`./hello`
5. echo $greeting

```

```

1. bash$ sh hello.sh
2. Hello, world.

```



在命令替换中，你可以使用 `$(...)` 来替代反引号。

```

1. output=$(sed -n /"$1"/p $file) # 摘自 "grp.sh"。
2.
3. # 将文本文件的内容赋值给一个变量。
4. File_contents1=$(cat $file1)
5. File_contents2=$(<$file2) # 这么做也是可以的。

```

`$(...)` 和反引号在处理双反斜杠上有所不同。

```

1. bash$ echo `echo \\\`
2.
3.
4. bash$ echo $(echo \\\)
5. \

```

`$(...)` 允许嵌套。[^3]

```

1. word_count=$( wc -w $(echo * | awk '{print $8}') )

```

样例 12-3. 寻找变位词 (*anagram*)

```

1. #!/bin/bash
2. # agram2.sh
3. # 嵌套命令替换的例子。
4.
5. # 其中使用了作者写的工具包 "yaw1" 中的 "anagram" 工具。
6. # http://ibiblio.org/pub/Linux/libs/yaw1-0.3.2.tar.gz
7. # http://bash.deta.in/yaw1-0.3.2.tar.gz
8.

```

```

9. E_NOARGS=86
10. E_BADARG=87
11. MINLEN=7
12.
13. if [ -z "$1" ]
14. then
15.     echo "Usage $0 LETTERSET"
16.     exit $E_NOARGS          # 脚本需要命令行参数。
17. elif [ ${#1} -lt $MINLEN ]
18. then
19.     echo "Argument must have at least $MINLEN letters."
20.     exit $E_BADARG
21. fi
22.
23.
24.
25. FILTER='.....'          # 至少需要7个字符。
26. #      1234567
27. Anagrams=( $(echo $(anagram $1 | grep $FILTER) ) )
28. #      $(      $(      嵌套命令集      ) )
29. #      (      赋值给数组      )
30.
31. echo
32. echo "${#Anagrams[*]} 7+ letter anagrams found"
33. echo
34. echo ${Anagrams[0]}        # 第一个变位词。
35. echo ${Anagrams[1]}        # 第二个变位词。
36.                            # 以此类推。
37.
38. # echo "${Anagrams[*]}"    # 将所有变位词在一行里面输出。
39.
40. # 可以配合后面的数组章节来理解上面的代码。
41.
42. # 建议同时查看另一个寻找变位词的脚本 agram.sh。
43.
44. exit $?

```

以下是包含命令替换的样例：

1. 样例 11-8
2. 样例 11-27

- 3. 样例 9-16
- 4. 样例 16-3
- 5. 样例 16-22
- 6. 样例 16-17
- 7. 样例 16-54
- 8. 样例 11-14
- 9. 样例 11-11
- 0. 样例 16-32
- 1. 样例 20-8
- 2. 样例 A-16
- 3. 样例 29-3
- 4. 样例 16-47
- 5. 样例 16-48
- 6. 样例 16-49

[^1]: 在命令替换中可以使用外部系统命令, [内建命令](#) 甚至是 [脚本函数](#)。

[^3]: 事实上, 使用反引号进行嵌套也是可行的。但是 John Default 提醒到需要将内部的反引号进行转义。

```
1. word_count=`wc -w \`echo * | awk '{print $8}'\` `
```

13. 算术扩展

- 第十三章 算术扩展
 - 差异比较
 - 使用 反引号 的算术扩展（通常与 `expr` 一起使用）
 - 使用 双圆括号 或 `let` 的算术扩展。

第十三章 算术扩展

算术扩展为脚本中的（整数）算术操作提供了强有力的工具。你可以使用反引号、双圆括号或者 `let` 将字符串转换为数学表达式。

差异比较

使用 反引号 的算术扩展（通常与 `expr` 一起使用）

```
1. z=`expr $z + 3`           # 'expr' 命令执行了算术扩展。
```

使用 双圆括号 或 `let` 的算术扩展。

事实上，在算术扩展中，反引号已经被双圆括号 `((...))` 和 `$(...)` 以及 `let` 所取代。

```
1. z=$(( $z + 3 ))
2. z=$(( z + 3 ))           # 同样正确。
3.                          # 在双圆括号内，参数引用形式可用可不用。
4.
5. # $((EXPRESSION)) 是算术扩展。 # 不要与命令替换混淆。
6.
7.
8.
9. # 双圆括号不是只能用作赋值算术结果。
10.
```

```
11.    n=0
12.    echo "n = $n"                # n = 0
13.
14.    (( n += 1 ))                  # 自增。
15.    # (( $n += 1 )) 是错误用法！
16.    echo "n = $n"                # n = 1
17.
18.
19.    let z=z+3
20.    let "z += 3" # 引号允许在赋值表达式中使用空格。
21.                                # 'let' 事实上执行的算术运算而非算术扩展。
```

以下是包含算术扩展的样例：

1. 样例 16-9
2. 样例 11-15
3. 样例 27-1
4. 样例 27-11
5. 样例 A-16

14. 休息时间

- [第十四章 休息时间](#)
 - [原作者致所有读者](#)

第十四章 休息时间

作者开始玩不转不是外国人的游戏了。亲爱的读者可以藉此休息一下，如果可以，请帮助我们推广一下本书原作和译作。

原作者致所有读者

各位 Linux 用户，你们好！你们现在正阅读的这本书能够给你们带来好运。

所以赶紧打开你们的邮箱，将本文的访问链接发给你的10位朋友。

但是在发邮件之前，记得粘贴一段大约100行的 Bash 脚本在邮件后面。

千万不要打断这个传递，并且一定要在48小时内发送邮件！

布鲁克林区的 Wilfred P. 没有发出10封邮件。当他第三天起床时发现他变成了一名 COBOL 程序员。

纽波特纽斯港的 Howard L. 按时发出了10封邮件。然后一个月内，他就有了足够的硬件来搭建一个100个节点的 Beowulf 集群来玩 Tuxracer。

芝加哥的 Amelia V. 看到以后不屑一顾，置之不理。不久之后，她的终端炸了。现在她不得不为微软工作，撰写文档。

千万不要打断这个传递！马上去发邮件吧！

Courtesy 'NIX "fortune cookies", with some alterations and many apologies

第五部分 进阶话题

- [第五部分 高级话题](#)
 - [目录](#)

第五部分 高级话题

目录

- [18. 正则表达式](#)
 - [18.1 正则表达式简介](#)
 - [18.2 文件名替换](#)
- [19. 嵌入文档](#)
- [20. I/O 重定向](#)
 - [20.1 使用 exec](#)
 - [20.2 重定向代码块](#)
 - [20.3 应用程序](#)
- [22. 限制模式的Shell](#)
- [24. 函数](#)
 - [24.1 复杂函数和函数复杂性](#)
 - [24.2 局部变量](#)
 - [24.3 不适用局部变量的递归](#)
- [25. 别名](#)
- [27. 数组](#)
- [30. 网络编程](#)
- [33. 选项](#)
- [34. 陷阱](#)
- [38. 后记](#)
 - [38.1 作者后记](#)

- [38.2 关于作者](#)
- [38.3 从哪里可以获得帮助](#)
- [38.4 用来制作这本书的工具](#)
- [38.5 致谢](#)
- [38.6 免责声明](#)

18 正则表达式

- [18 正则表达式](#)

18 正则表达式

...the intellectual activity associated with software development is largely one of gaining insight.

—Stowe Boyd

目录

- [18.1 正则表达式简介](#)
- [18.2 文件名替换](#)

为了充分利用shell脚本，您需要熟练掌握正则表达式。有一些在脚本中常用的特定的命令和工具，例如grep、expr、sed和awk，这些命令会解释和使用正则表达式。版本三的bash实现了它独特的正则匹配符：`=~`。

18.1 正则表达式简介

- [18.1 正则表达式简介](#)

18.1 正则表达式简介

正则表达式是一系列的字符串。这些包含超过其字面含义的字符串被称之为元字符。例如，一个符号前面的引用符代表一个人的言语能力，或者按照上面的说法，代表着meta-meaning[\[1\]](#)。正则表达式是一组字符串和（或者）一组匹配（特定的）模式的元字符。

一个正则表达式包含下面的一个或多个选项：

- 一组字符串。这是仅仅表示字面意思的字符串。最简单形式的正则表达式仅仅包含一组字符串。
- 一个锚字符。锚节点指定了正则表达式在一行文本中的匹配位置。例如，`^`和`$`就是锚字符。
- 修饰符。修饰符扩展或者限定（修改）了正则表达式在文本中的匹配范围。修饰符包括星号、方括号和反斜线。

正则表达式的主要用在文本搜索和字符串操作。一个正则表达式匹配单个字符或者一组字符 — 一系列的字符或者字符串的一部分。

- 星号 `*` 匹配前面的子表达式任意次，包括0次

`"1133*"匹配"11"加一个或多个"3"：113, 1133, 1133333, 及以后`

- 点号 `.` 匹配任意字符，除了新的一行[\[2\]](#)

`"13."匹配"13"加至少一个字符（包括空格）：1133, 11333, 但不是13（缺少额外的字符）`

参见例子16-18，展示单字符匹配

- 脱字符 `^` 匹配行的起始位置，但有时候会根据上下文环境匹配其相反的意义（译者注：例如`^[a]`匹配任意一个非a的字符）

- 美元符 `$` 匹配行的结束位置

`"XXX$"`匹配行尾处的`"XXX"`

`"^$"`匹配空行

- 方括号 `[...]` 匹配所包含的任意一个字符

`"[xyz]"`匹配x、y或z中的任意一个字符

`"[c-n]"`匹配c到n之间的任意一个字符

`"[B-Pk-y]"`匹配B到P和k到y之间任意一个字符

`"[a-z0-9]"`匹配任意一个小写字符和任意一个数字

`"[^b-d]"`匹配任意一个不在b到d之间的字符。这是一个很好的例子，展示了`"^"`的匹配了正则表达式的反义（类似在其他环境下的`"!"`符号所起的作用）

组合一连串的用方括号括起来的字符能匹配非常多的词组模

式。`"[Yy][Ee][Ss]"`匹配yes、Yes、YES、yEs等等。`"[0-9]`

`[0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]"`匹配任何一个社会保险号

- 反斜线 `\` 转义一个特殊字符，意味着这个字符被解释为字面意义（因此不再包含特殊意思）

`"\$"`表示回它的字面意义`"$"`，而不是它原本在正则表达式中代表行尾的意义。同样，`"\"`表示字面意义`"\"`

- 转义后的尖括号 `\<.>` 代表词组的边界

尖括号必须进行转义，否则它们就代表其字面意义

“\”匹配词组“the”，而不是词组“them,” “there,”
“other,”等等

```

1. bash$ cat textfile
2. This is line 1, of which there is only one instance.
3. This is the only instance of line 2.
4. This is line 3, another line.
5. This is line 4.
6.
7.
8. bash$ grep 'the' textfile
9. This is line 1, of which there is only one instance.
10. This is the only instance of line 2.
11. This is line 3, another line.
12.
13.
14. bash$ grep '\<the\>' textfile
15. This is the only instance of line 2.
```

唯一判断一个特定的正则表达式是否有效的方法就是测试它。

```

1. 测试文件: tstfile                                # No match.
2.                                                    # No match.
3. 运行 grep "1133*" tstfile                        # Match.
4.                                                    # No match.
5.                                                    # No match.
6. This line contains the number 113.                # Match.
7. This line contains the number 13.                 # No match.
8. This line contains the number 133.                # No match.
9. This line contains the number 1133.               # Match.
10. This line contains the number 113312.            # Match.
11. This line contains the number 1112.              # No match.
12. This line contains the number 113312312.         # Match.
13. This line contains no numbers at all.            # No match.
```

```

1. bash$ grep "1133*" tstfile
2. Run grep "1133*" on this file.                  # Match.
```

```

3.  This line contains the number 113.          # Match.
4.  This line contains the number 1133.         # Match.
5.  This line contains the number 113312.       # Match.
6.  This line contains the number 113312312.   # Match.

```

注解

[1] 元意义指的是一个词组或者表达式在更高层次的抽象上的意义。例如，正则表达式的字面意思就是所有人接受其用法的普通表达式。元意义则完全不同，正如在本章最终讨论的那样。

[2]

Since sed, awk, and grep process single lines, there will usually not be a newline to match. In those cases where there is a newline in a multiple line expression, the dot will match the newline.

```

1.  #!/bin/bash
2.
3.  sed -e 'N;s/.*/[&]/' << EOF    # Here Document
4.  line1
5.  line2
6.  EOF
7.  # OUTPUT:
8.  # [line1
9.  # line2]
10.
11.
12.
13. echo
14.
15. awk '{ $0=$1 "\n" $2; if (/line.1/) {print}}' << EOF
16. line 1
17. line 2
18. EOF
19. # OUTPUT:

```

```
20. # line
21. # 1
22.
23.
24. # Thanks, S.C.
25.
26. exit 0
```

19. 嵌入文档

- 19 嵌入文档

19 嵌入文档

Here and now, boys.
—Aldous Huxley, Island

嵌入文档是一段有特殊作用的代码块，它用 **I/O 重定向** 在交互程序和交互命令中传递和反馈一个命令列表，例如 **ftp**，**cat** 或者是 **ex** 文本编辑器

```
1. COMMAND <<InputComesFromHERE
2. ...
3. ...
4. ...
5. InputComesFromHERE
```

嵌入文档用限定符作为命令列表的边界，在限定符前需要一个指定的标识符 `<<`，这会将一个程序或命令的标准输入(`stdin`)进行重定向，它类似 `交互程序 < 命令文件` 的方式，其中命令文件内容如下

```
1. command #1
2. command #2
3. ...
```

嵌入文档的格式大致如下

```
1. interactive-program <<LimitString
2. command #1
3. command #2
4. ...
5. LimitString
```


限定符的选择必须保证特殊以确保不会和命令列表里的内容发生混淆。

注意嵌入文档有时候用作非交互的工具和命令有着非常好的效果，例如 `wall`

样例 19-1. `broadcast`：给每个登陆者发送信息

```

1.  #!/bin/bash
2.
3.  wall <<zzz23EndOfMessagezzz23
4.  E-mail your noontime orders for pizza to the system administrator.
5.      (Add an extra dollar for anchovy or mushroom topping.)
6.  # 额外的信息文本.
7.  # 注意: 'wall' 会打印注释行.
8.  zzz23EndOfMessagezzz23
9.
10. # 更有效的做法是通过
11. #      wall < 信息文本
12. # 然而，在脚本里嵌入信息模板不乏是一种迅速而又随性的解决方式.
13.
14. exit
```

样例：19-2. `dummyfile`：创建一个有两行内容的虚拟文件

```

1.  #!/bin/bash
2.
3.  # 非交互的使用 `vi` 编辑文件.
4.  # 仿照 'sed'.
5.
6.  E_BADARGS=85
7.
8.  if [ -z "$1" ]
9.  then
10.     echo "Usage: `basename $0` filename"
11.     exit $E_BADARGS
12. fi
13.
```

```

14. TARGETFILE=$1
15.
16. # 插入两行到文件中保存
17. #-----Begin here document-----#
18. vi $TARGETFILE <<x23LimitStringx23
19. i
20. This is line 1 of the example file.
21. This is line 2 of the example file.
22. ^[
23. ZZ
24. x23LimitStringx23
25. #-----End here document-----#
26.
27. # 注意 "^" 对 "[" 进行了转义
28. #+ 这段起到了和键盘上按下 Control-V <Esc> 相同的效果.
29.
30. # Bram Moolenaar 指出这种情况下 'vim' 可能无法正常工作
31. #+ 因为在与终端交互的过程中可能会出现问題.
32.
33. exit

```

上述脚本实现了 `ex` 的功能，而不是 `vi`。嵌入文档包含了 `ex` 足够通用的命令列表来形成自有的类别，所以又称之为 `ex` 脚本。

```

1. #!/bin/bash
2. # 替换所有的以 ".txt" 后缀结尾的文件的 "Smith" 为 "Jones"
3.
4. ORIGINAL=Smith
5. REPLACEMENT=Jones
6.
7. for word in $(fgrep -l $ORIGINAL *.txt)
8. do
9.     # -----
10.     ex $word <<EOF
11.     :%s/$ORIGINAL/$REPLACEMENT/g
12.     :wq

```

```

13. EOF
14.   # :%s is the "ex" substitution command.
15.   # :wq is write-and-quit.
16.   # -----
17. done

```

类似的 `ex` 脚本 是 `cat` 脚本 。

样例 19-3. 使用 `cat` 的多行信息

```

1. #!/bin/bash
2.
3. # 'echo' 可以输出单行信息,
4. #+ 但是如果是输出消息块就有点问题了.
5. # 'cat' 嵌入文档却能解决这个局限.
6.
7. cat <<End-of-message
8. -----
9. This is line 1 of the message.
10. This is line 2 of the message.
11. This is line 3 of the message.
12. This is line 4 of the message.
13. This is the last line of the message.
14. -----
15. End-of-message
16.
17. # 替换上述嵌入文档内的 7 行文本
18. #+ cat > $Newfile <<End-of-message
19. #+ ^^^^^^^^^^^
20. #+ 将输出追加到 $Newfile, 而不是标准输出.
21.
22. exit 0
23.
24.
25. #-----
26. # 由于上面的 "exit 0", 下面的代码将不会生效.
27.
28. # S.C. points out that the following also works.

```

```

29. echo "------"
30. This is line 1 of the message.
31. This is line 2 of the message.
32. This is line 3 of the message.
33. This is line 4 of the message.
34. This is the last line of the message.
35. -----"
36. # 然而，文本可能不包括双引号除非出现了字符串逃逸。

```

`-` 的作用是标记了一个嵌入文档限制符 (`<<-LimitString`)，它能抑制输出的行首的 `tab` (非空格)。这在脚本可读性方面可能非常有用。

样例 19-4. 抑制 `tab` 的多行信息

```

1. #!/bin/bash
2. # 和之前的样例一样，但...
3.
4. # 嵌入文档内的 '-'，也就是 <<-
5. #+ 抑制了文档行首的 'tab'，
6. #+ 但 *不是* 空格。
7.
8. cat <<-ENDOFMESSAGE
9.     This is line 1 of the message.
10.    This is line 2 of the message.
11.    This is line 3 of the message.
12.    This is line 4 of the message.
13.    This is the last line of the message.
14. ENDOFMESSAGE
15. # 脚本的输出将左对齐。
16. # 行首的 tab 将不会输出。
17.
18. # 上面 5 行的 "信息" 以 tab 开始，不是空格。
19. # 空格不会受影响 <<- .
20.
21. # 注意这个选项对 *内嵌的* tab 没有影响。
22.

```

23. `exit 0`

嵌入文档支持参数和命令替换。因此可以向嵌入文档传递不同的参数，变向的改其输出。

样例 19-5. 可替换参数的嵌入文档

```

1.  #!/bin/bash
2.  # 另一个使用参数替换的 'cat' 嵌入文档.
3.
4.  # 试一试没有命令行参数,    ./scriptname
5.  # 试一试一个命令行参数,    ./scriptname Mortimer
6.  # 试试用一两个单词引用命令行参数,
7.  #                                ./scriptname "Mortimer Jones"
8.
9.  CMDLINEPARAM=1      # Expect at least command-line parameter.
10.
11. if [ $# -ge $CMDLINEPARAM ]
12. then
13.     NAME=$1          # If more than one command-line param,
14.                     #+ then just take the first.
15. else
16.     NAME="John Doe"  # Default, if no command-line parameter.
17. fi
18.
19. RESPONDENT="the author of this fine script"
20.
21.
22. cat <<Endofmessage
23.
24. Hello, there, $NAME.
25. Greetings to you, $NAME, from $RESPONDENT.
26.
27. # 这个注释在输出时显示 (为什么?).
28.
29. Endofmessage
30.

```

```

31. # 注意输出了空行.
32. # 所以可以这样注释.
33.
34. exit

```

这个包含参数替换的嵌入文档是相当有用的

样例 19-6. 上传文件对到 Sunsite 入口目录

```

1. #!/bin/bash
2. # upload.sh
3.
4. # 上传文件对 (Filename.lsm, Filename.tar.gz)
5. #+ 到 Sunsite/UNC (ibiblio.org) 的入口目录.
6. # Filename.tar.gz 是个 tarball.
7. # Filename.lsm is 是个描述文件.
8. # Sunsite 需要 "lsm" 文件, 否则将会退回给发送者
9.
10.
11. E_ARGERROR=85
12.
13. if [ -z "$1" ]
14. then
15.     echo "Usage: `basename $0` Filename-to-upload"
16.     exit $E_ARGERROR
17. fi
18.
19.
20. Filename=`basename $1`           # Strips pathname out of file
    name.
21.
22. Server="ibiblio.org"
23. Directory="/incoming/Linux"
24. # 脚本里不需要硬编码,
25. #+ 但最好可以替换命令行参数.
26.
27. Password="your.e-mail.address"  # Change above to suit.
28.

```

```

29. ftp -n $Server <<End-Of-Session
30. # -n 禁用自动登录
31.
32. user anonymous "$Password"      # If this doesn't work, then try:
33.                                # quote user anonymous
    "$Password"
34. binary
35. bell                          # Ring 'bell' after each file
    transfer.
36. cd $Directory
37. put "$Filename.lsm"
38. put "$Filename.tar.gz"
39. bye
40. End-Of-Session
41.
42. exit 0

```

在嵌入文档头部引用或转义“限制符”来禁用参数替换。原因是 引用/转义 限定符能有效的转义 “\$”，“`”，和 “\” 这些特殊符号，使它们维持字面上的意思。（感谢 Allen Halsey 指出这点。）

样例 19-7. 禁用参数替换

```

1. #!/bin/bash
2. # A 'cat' here-document, but with parameter substitution disabled.
3.
4. NAME="John Doe"
5. RESPONDENT="the author of this fine script"
6.
7. cat <<'Endofmessage'
8.
9. Hello, there, $NAME.
10. Greetings to you, $NAME, from $RESPONDENT.
11.
12. Endofmessage
13.
14. # 当'限制符'引用或转义时不会有参数替换。

```

```

15. # 下面的嵌入文档也有同样的效果
16. # cat <<"Endofmessage"
17. # cat <<\Endofmessage
18.
19.
20.
21. # 同样的:
22.
23. cat <<"SpecialCharTest"
24.
25. Directory listing would follow
26. if limit string were not quoted.
27. `ls -l`
28.
29. Arithmetic expansion would take place
30. if limit string were not quoted.
31. $((5 + 3))
32.
33. A a single backslash would echo
34. if limit string were not quoted.
35. \\
36.
37. SpecialCharTest
38.
39.
40. exit

```

生成脚本或者程序代码时可以用禁用参数的方式来输出文本。

样例 19-8. 生成其他脚本的脚本

```

1. #!/bin/bash
2. # generate-script.sh
3. # Based on an idea by Albert Reiner.
4.
5. OUTFILE=generated.sh          # Name of the file to generate.
6.
7.

```



```
8. # -----
9. # '嵌入文档涵盖了生成脚本的主体部分.
10. (
11. cat <<'EOF'
12. #!/bin/bash
13.
14. echo "This is a generated shell script."
15. # 注意我们现在在一个子 shell 内,
16. #+ 我们不能访问 "外部" 脚本变量.
17.
18. echo "Generated file will be named: $OUTFILE"
19. # 上面这行并不能按照预期的正常工作
20. #+ 因为参数扩展已被禁用.
21. # 相反的, 结果是文字输出.
22.
23. a=7
24. b=3
25.
26. let "c = $a * $b"
27. echo "c = $c"
28.
29. exit 0
30. EOF
31. ) > $OUTFILE
32. # -----
33.
34. # 在上述的嵌入文档内引用 '限制符' 防止变量扩展
35.
36. if [ -f "$OUTFILE" ]
37. then
38.     chmod 755 $OUTFILE
39.     # 生成可执行文件.
40. else
41.     echo "Problem in creating file: \"$OUTFILE\""
42. fi
43.
44. # 这个方法适用于生成 C, Perl, Python, Makefiles 等等
45.
```

```
46. exit 0
```

可以从嵌入文档的输出设置一个变量的值。这实际上是种灵活的 **命令替换**。

```
1. variable=$(cat <<SETVAR
2. This variable
3. runs over multiple lines.
4. SETVAR
5. )
6.
7. echo "$variable"
```

同样的脚本里嵌入文档可以作为函数的输入。

样例 19-9. 嵌入文档和函数

```
1. #!/bin/bash
2. # here-function.sh
3.
4. GetPersonalData ()
5. {
6.     read firstname
7.     read lastname
8.     read address
9.     read city
10.    read state
11.    read zipcode
12. } # 可以肯定的是这应该是个交互式的函数，但 . . .
13.
14.
15. # 作为函数的输入。
16. GetPersonalData <<RECORD001
17. Bozo
18. Bozeman
19. 2726 Nondescript Dr.
20. Bozeman
```

```

21. MT
22. 21226
23. RECORD001
24.
25.
26. echo
27. echo "$firstname $lastname"
28. echo "$address"
29. echo "$city, $state $zipcode"
30. echo
31.
32. exit 0

```

可以这样使用：作为一个虚构的命令接受嵌入文档的输出。这样实际上就创建了一个“匿名”嵌入文档。

样例 19-10. “匿名” 嵌入文档

```

1. #!/bin/bash
2.
3. : <<TESTVARIABLES
4. ${HOSTNAME?}${USER?}${MAIL?} # Print error message if one of the
   variables not set.
5. TESTVARIABLES
6.
7. exit $?

```

- 上面技巧的一种变体允许“可添加注释”的代码块。

样例 19-11. 可添加注释的代码块

```

1. #!/bin/bash
2. # commentblock.sh
3.
4. : <<COMMENTBLOCK
5. echo "This line will not echo."
6. 这些注释没有 "#" 前缀。

```

```

7. 则是另一种没有 "#" 前缀的注释方法.
8.
9.  &*@!!+=
10. 上面这行不会产生报错信息,
11. 因为 bash 解释器会忽略它.
12.
13. COMMENTBLOCK
14.
15. echo "Exit value of above \"COMMENTBLOCK\" is $?."    # 0
16. # 没有错误输出.
17. echo
18.
19. # 上面的技巧经常用于工作代码的注释用作排错目的
20. # 这省去了在每一行开头加上 "#" 前缀,
21. #+ 然后调试完不得不删除每行的前缀的重复工作.
22. # 注意我们用了 ":", 在这之上, 是可选的.
23.
24. echo "Just before commented-out code block."
25. # 下面这个在双破折号之间的代码不会被执行.
26. #
=====
27. : <<DEBUGXXX
28. for file in *
29. do
30.   cat "$file"
31. done
32. DEBUGXXX
33. #
=====
34. echo "Just after commented-out code block."
35.
36. exit 0
37.
38.
39.
40. #####
41. # 注意, 然而, 如果将变量中包含一个注释的代码块将会引发问题
42. # 例如:

```

```

43.
44.
45.  #!/bin/bash
46.
47.  : <<COMMENTBLOCK
48.  echo "This line will not echo."
49.  &*@!!+=
50.  ${foo_bar_bazz?}
51.  $(rm -rf /tmp/foobar/)
52.  $(touch my_build_directory/cups/Makefile)
53.  COMMENTBLOCK
54.
55.
56.  $ sh commented-bad.sh
57.  commented-bad.sh: line 3: foo_bar_bazz: parameter null or not set
58.
59.  # 有效的补救办法就是在 49 行的位置加上单引号, 变为 'COMMENTBLOCK'.
60.
61.  : <<'COMMENTBLOCK'
62.
63.  # 感谢 Kurt Pfeifle 指出这一点.

```

- 另一个漂亮的方法使得“自文档化”的脚本成为可能

样例 19-12. 自文档化的脚本

```

1.  #!/bin/bash
2.  # self-document.sh: self-documenting script
3.  # Modification of "colm.sh".
4.
5.  DOC_REQUEST=70
6.
7.  if [ "$1" = "-h" -o "$1" = "--help" ]      # 请求帮助.
8.  then
9.      echo; echo "Usage: $0 [directory-name]"; echo
10.     sed --silent -e '/DOCUMENTATIONXX$/,/^DOCUMENTATIONXX$/p' "$0" |
11.     sed -e '/DOCUMENTATIONXX$/d'; exit $DOC_REQUEST; fi

```

```

12.
13.
14. : <<DOCUMENTATIONXX
15. List the statistics of a specified directory in tabular format.
16. -----
17. The command-line parameter gives the directory to be listed.
18. If no directory specified or directory specified cannot be read,
19. then list the current working directory.
20.
21. DOCUMENTATIONXX
22.
23. if [ -z "$1" -o ! -r "$1" ]
24. then
25.     directory=.
26. else
27.     directory="$1"
28. fi
29.
30. echo "Listing of "$directory":"; echo
31. (printf "PERMISSIONS LINKS OWNER GROUP SIZE MONTH DAY HH:MM PROG-
    NAME\n" \
32. ; ls -l "$directory" | sed 1d) | column -t
33.
34. exit 0

```

使用 `cat script` 是另一种可行的方法。

```

1. DOC_REQUEST=70
2.
3. if [ "$1" = "-h" -o "$1" = "--help" ]      # Request help.
4. then                                         # Use a "cat script" . .
    .
5.     cat <<DOCUMENTATIONXX
6. List the statistics of a specified directory in tabular format.
7. -----
8. The command-line parameter gives the directory to be listed.
9. If no directory specified or directory specified cannot be read,
10. then list the current working directory.

```

```

11.
12. DOCUMENTATIONXX
13. exit $DOC_REQUEST
14. fi

```

另请参阅 [样例 A-28](#), [样例 A-40](#), [样例 A-41](#), and [样例 A-42](#) 更多样例请阅读脚本附带的注释文档。

- 嵌入文档创建了临时文件，但这些文件在打开且不可被其他程序访问后删除。

```

1. bash$ bash -c 'lsof -a -p $$ -d0' << EOF
2. > EOF
3. lsof      1213 bozo      0r    REG      3,5      0 30386 /tmp/t1213-0-sh
      (deleted)

```

- 某些工具在嵌入文档内部并不能正常运行。
- 在嵌入文档的最后关闭限定符必须在起始的第一个字符的位置开始。行首不能是空格。限制符后尾随空格同样会导致意想不到的行为。空格可以防止限制符被当做其他用途。 [\[1\]](#)

```

1. #!/bin/bash
2.
3. echo "-----"
4.
5. cat <<LimitString
6. echo "This is line 1 of the message inside the here document."
7. echo "This is line 2 of the message inside the here document."
8. echo "This is the final line of the message inside the here
   document."
9.      LimitString
10. #^^^^^限制符的缩进。出错！这个脚本将不会如期运行。
11.
12. echo "-----"

```

```

13.
14. # 这些评论在嵌入文档范围外并不能输出
15.
16. echo "Outside the here document."
17.
18. exit 0
19.
20. echo "This line had better not echo." # 紧跟着个 'exit' 命令.

```

- 有些人非常聪明的使用了一个单引号(!)做为限制符。但这并不是个好主意

```

1. # 这个可以运行.
2. cat <<!
3. Hello!
4. ! Three more exclamations !!!
5. !
6.
7.
8. # 但是 . . .
9. cat <<!
10. Hello!
11. Single exclamation point follows!
12. !
13. !
14. # Crashes with an error message.
15.
16.
17. # 然而, 下面这样也能运行.
18. cat <<EOF
19. Hello!
20. Single exclamation point follows!
21. !
22. EOF
23. # 使用多字符限制符更为安全.

```

为嵌入文档设置这些任务有些复杂，可以考虑使用 `expect`，一种专

门用来和程序进行交互的脚本语言。

Notes:

除此之外，Dennis Benzinger 指出，使用 `<<-` 抑制 `tab.`

20. I/O 重定向

- [20 I/O 重定向](#)
 - [注意](#)

20 I/O 重定向

目录

- [20.1 使用 exec](#)
- [20.2 重定向代码块](#)
- [20.3 应用程序](#)

有三个默认打开的文件[\[1\]](#)，`stdin`（标准输入，键盘），`stdout`（标准输出， 屏幕）和 `stderr`（标准错误，屏幕上输出的错误信息）。这些和任何其他打开的文件都可以被重定向。重定向仅仅意味着捕获输出文件，命令，脚本，甚至是一个脚本的代码块([样例 3-1](#))和([样例 3-2](#)) 作为另一个文件，命令，程序或脚本的输入。

每个打开的文件都有特定的文件描述符。[\[2\]](#)，而

`stdin`，`stdout`，`stderr` 的文件描述符分别为 0, 1, 2。当然了，还有附件的文件描述符 3 - 9。有时候为 `stdin`，`stdout`，`stderr` 临时性的复制链接分配这些附加的文件描述符会非常有用。[\[3\]](#)。这简化了复杂重定向和重组后的恢复（见[样例 20-1](#)）

```
1.    COMMAND_OUTPUT >
2.        # 重定向标准输出到一个文件.
3.        # 如果文件不存在则创建，否则覆盖.
4.
5.    ls -lR > dir-tree.list
6.        # 创建了一个包含目录树列表的文件.
```

```

7.
8.      : > filename
9.      # ">" 清空了文件.
10.     # 如果文件不存在, 则创建了一个空文件 (效果类似 'touch').
11.     # ":" 是个虚拟占位符, 不会有输出.
12.
13.     > filename
14.     # ">" 清空了文件.
15.     # 如果文件不存在, 则创建了一个空文件 (效果类似 'touch').
16.     # (结果和上述的 ":">" 一样, 但在某些 shell 环境中不能正常运行.)
17.
18.     COMMAND_OUTPUT >>
19.     # 重定向标准输出到一个文件.
20.     # 如果文件不存在则创建, 否则新内容在文件末尾追加.
21.
22.
23.     # 单行重定向命令 (只作用于本身所在的那行):
24.     # -----
-----
25.
26.     1>filename
27.     # 以覆盖的方式将 标准错误 重定向到文件 "filename."
28.     1>>filename
29.     # 以追加的方式将 标准输出 重定向到文件 "filename."
30.     2>filename
31.     # 以覆盖的方式将 标准错误 重定向到文件 "filename."
32.     2>>filename
33.     # 以追加的方式将 标准错误 重定向到文件 "filename."
34.     &>filename
35.     # 以覆盖的方式将 标准错误 和 标准输出 同时重定向到文件 "filename."
36.     # 在 bash 4 中才有这个新功能.
37.
38.     M>N
39.     # "M" 是个文件描述符, 如果不明确指定, 默认为 1.
40.     # "N" 是个文件名.
41.     # 文件描述符 "M" 重定向到文件 "N."
42.     M>&N
43.     # "M" 是个文件描述符, 如果不设置默认为 1.

```

```

44.      # "N" 是另一个文件描述符.
45.
46.
=====
47.
48.      # 重定向 标准输出, 一次一行.
49.      LOGFILE=script.log
50.
51.      echo "This statement is sent to the log file, \"$LOGFILE\"."
1>$LOGFILE
52.      echo "This statement is appended to \"$LOGFILE\"."
1>>$LOGFILE
53.      echo "This statement is also appended to \"$LOGFILE\"."
1>>$LOGFILE
54.      echo "This statement is echoed to stdout, and will not appear
in \"$LOGFILE\"."
55.      # 这些重定向命令在每行结束后自动"重置".
56.
57.
58.
59.      # 重定向 标准错误, 一次一行.
60.      ERRORFILE=script.errors
61.
62.      bad_command1 2>$ERRORFILE      # Error message sent to
$ERRORFILE.
63.      bad_command2 2>>$ERRORFILE    # Error message appended to
$ERRORFILE.
64.      bad_command3                  # Error message echoed to
stderr,
65.                                     #+ and does not appear in
$ERRORFILE.
66.      # 这些重定向命令每行结束后会自动"重置".
67.
=====

```

1. 2>&1
2. # 重定向 标准错误 到 标准输出.

```

3.      # 错误信息发送到标准输出相同的位置.
4.      >>filename 2>&1
5.      bad_command >>filename 2>&1
6.      # 同时将 标准输出 和 标准错误 追加到文件 "filename" 中 ...
7.      2>&1 | [command(s)]
8.      bad_command 2>&1 | awk '{print $5}'    # found
9.      # 通过管道传递 标准错误.
10.     # bash 4 中可以将 "2>&1 |" 缩写为 "|&".
11.
12.     i>&j
13.     # 重定向文件描述符 i 到 j.
14.     # 文件描述符 i 指向的文件输出将会重定向到文件描述符 j 指向的文件
15.
16.     >&j
17.     # 默认的标准输出 (stdout) 重定向到 j.
18.     # 所有的标准输出将会重定向到 j 指向的文件.

```

```

1.     0< FILENAME
2.     < FILENAME
3.     # 从文件接收输入.
4.     # 类似功能命令是 ">", 经常会组合使用.
5.     #
6.     # grep search-word <filename
7.
8.
9.     [j]<>filename
10.    # 打开并读写文件 "filename" ,
11.    #+ 并且分配文件描述符 "j".
12.    # 如果 "filename" 不存在则创建.
13.    # 如果文件描述符 "j" 未指定, 默认分配文件描述符 0, 标准输入.
14.    #
15.    # 这是一个写指定文件位置的应用程序.
16.    echo 1234567890 > File      # 写字符串到 "File".
17.    exec 3<> File              # 打开并分配文件描述符 3 给 "File" .
18.    read -n 4 <&3              # 读取 4 字符.
19.    echo -n . >&3              # 写一个小数点.
20.    exec 3>&-                  # 关闭文件描述符 3.
21.    cat File                  # ==> 1234.67890

```

```

22.      # 随机访问.
23.
24.
25.
26.      |
27.      # 管道.
28.      # 一般是命令和进程的链接工具.
29.      # 类似 ">", 但更一般.
30.      # 在连接命令, 脚本, 文件和程序方面非常有用.
31.      cat *.txt | sort | uniq > result-file
32.      # 所有 .txt 文件输出进行排序并且删除复制行,
33.      # 最终保存结果到 "result-file".

```

可以用单个命令行表示输入和输出的多个重定向或管道。

```

1.  command < input-file > output-file
2.  # 或者等价:
3.  < input-file command > output-file  # 尽管这不标准.
4.
5.  command1 | command2 | command3 > output-file

```

更多详情见[样例 16-31](#) and [样例 A-14](#)。

多个输出流可以重定向到一个文件。

```

1.  ls -yz >> command.log 2>&1
2.  # 捕获不合法选项 "yz" 的结果到文件 "command.log."
3.  # 因为 标准错误输出 被重定向到了文件,
4.  #+ 任何错误信息都会在这.
5.
6.  # 注意, 然而, 接下来的这个案例并 "不能" 同样的结果.
7.  ls -yz 2>&1 >> command.log
8.  # 输出一条错误信息, 但是不会写入到文件.
9.  # 恰恰的, 命令输出(这个例子里为空)写入到文件, 但错误信息只会在 标准输出 输出.
10.
11. # 如果同时重定向 标准输出 和 标准错误输出,
12. #+ 命令的顺序不同会导致不同.

```

关闭文件描述符

```

1.  n<&-
2.      关闭输入文件描述符 n.
3.
4.  0<&-, <&-
5.      关闭标准输入.
6.
7.  n>&-
8.      关闭输出文件描述符 n.
9.
10. 1>&-, >&-
11.      关闭标准输出.

```

子进程能继承文件描述符. 这就是管道符能工作的原因. 通过关闭文件描述符来防止继承 .

```

1.  # 只重定向到 标准错误 到管道.
2.
3.  exec 3>&1                                # 保存当前 标准输出 "值".
4.
5.  ls -l 2>&1 >&3 3>&- | grep bad 3>&-      # 关闭 'grep' 文件描述符 3 (但
    不是 'ls').
6.  #                ^^^^^  ^^^^^
7.  exec 3>&-                                # 现在关闭它.
8.
9.  # 感谢, S.C.

```

更多关于 I/O 重定向详情见 [Appendix F](#).

注意

[1] 在 UNIX 和 Linux 中, 数据流和周边外设([device files](#))都被看做文件.

[2] 文件描述符 仅仅是操作系统分配的一个可追踪的打开的文件号.

可以认为是一个简化的文件指针。类似于 C 语言的 `文件句柄`。

[3] 当 bash 创建一个子进程的时候使用 `文件描述符 5` 会有问题。例如 `exec`，子进程继承了文件描述符 5（详情见 Chet Ramey's 归档的 e-mail, [SUBJECT: RE: File descriptor 5 is held open](#)）。最好将这个文件描述符单独规避。

20.1 使用 exec

- 20.1 使用 exec

20.1 使用 exec

一个 `exec < filename` 命令重定向了 标准输入 到一个文件。自此所有 标准输入 都来自该文件而不是默认来源(通常是键盘输入)。在使用 `sed` 和 `awk` 时候这种方式可以逐行读文件并逐行解析。

样例 20-1. 使用 `exec` 重定向 标准输入

```

1. #!/bin/bash
2. # 使用 'exec' 重定向 标准输入 .
3.
4.
5. exec 6<&0          # 链接文件描述符 #6 到标准输入.
6.                   # .
7.
8. exec < data-file   # 标准输入被文件 "data-file" 替换
9.
10. read a1           # 读取文件 "data-file" 首行.
11. read a2           # 读取文件 "data-file" 第二行
12.
13. echo
14. echo "Following lines read from file."
15. echo "-----"
16. echo $a1
17. echo $a2
18.
19. echo; echo; echo
20.
21. exec 0<&6 6<&-
22. # 现在在之前保存的位置将从文件描述符 #6 将 标准输出 恢复.
23. #+ 且关闭文件描述符 #6 ( 6<&- ) 让其他程序正常使用.
24. #
```

```

25. # <&6 6<&-    also works.
26.
27. echo -n "Enter data  "
28. read b1 # 现在按预期的, 从正常的标准输入 "read".
29. echo "Input read from stdin."
30. echo "-----"
31. echo "b1 = $b1"
32.
33. echo
34.
35. exit 0

```

同理, `exec >filename` 重定向 标准输出 到指定文件. 他将所有的命令输出通常是 标准输出 重定向到指定的位置.

`exec N > filename` 影响整个脚本或当前 shell。PID 从重定向脚本或 shell 的那时候已经发生了改变. 然而 `N > filename` 影响的就新派生的进程, 而不是整个脚本或 shell。

样例 20-2. 使用 exec 重定向标准输出

```

1. #!/bin/bash
2. # reassign-stdout.sh
3.
4. LOGFILE=logfile.txt
5.
6. exec 6>&1          # 链接文件描述符 #6 到标准输出.
7.                  # 保存标准输出.
8.
9. exec > $LOGFILE    # 标准输出被文件 "logfile.txt" 替换.
10.
11. # ----- #
12. # 所有在这个块里的命令的输出都会发送到文件 $LOGFILE.
13.
14. echo -n "Logfile: "
15. date
16. echo "-----"

```

```

17. echo
18.
19. echo "Output of \"ls -al\" command"
20. echo
21. ls -al
22. echo; echo
23. echo "Output of \"df\" command"
24. echo
25. df
26.
27. # ----- #
28.
29. exec 1>&6 6>&-      # 关闭文件描述符 #6 恢复 标准输出.
30.
31. echo
32. echo "== stdout now restored to default == "
33. echo
34. ls -al
35. echo
36.
37. exit 0

```

样例 20-3. 用 exec 在一个脚本里同时重定向 标准输入 和 标准输出

```

1. #!/bin/bash
2. # upperconv.sh
3. # 转化指定的输入文件成大写.
4.
5. E_FILE_ACCESS=70
6. E_WRONG_ARGS=71
7.
8. if [ ! -r "$1" ]      # 指定的输入文件是否可读?
9. then
10.    echo "Can't read from input file!"
11.    echo "Usage: $0 input-file output-file"
12.    exit $E_FILE_ACCESS

```

```

13. fi                                # 同样的错误退出
14.                                  #+ 等同如果输入文件 ($1) 未指定 (为什么?).
15.
16. if [ -z "$2" ]
17. then
18.     echo "Need to specify output file."
19.     echo "Usage: $0 input-file output-file"
20.     exit $E_WRONG_ARGS
21. fi
22.
23.
24. exec 4<&0
25. exec < $1                        # 将从输入文件读取.
26.
27. exec 7>&1
28. exec > $2                        # 将写入输出文件.
29.                                # 假定输出文件可写 (增加检测?).
30.
31. # -----
32.     cat - | tr a-z A-Z           # 转化大写.
33. #   ^^^^^                      # 读取标准输入.
34. #           ^^^^^^^^^^^^^      # 写到标准输出.
35. # 然而标准输入和标准输出都会被重定向.
36. # 注意 'cat' 可能会被遗漏.
37. # -----
38.
39. exec 1>&7 7>&-                    # 恢复标准输出.
40. exec 0<&4 4<&-                    # 恢复标准输入.
41.
42. # 恢复后, 下面这行会预期从标准输出打印.
43. echo "File \"$1\" written to \"$2\" as uppercase conversion."
44.
45. exit 0

```

I/O 重定向是种明智的规避 `inaccessible variables within a subshell` 问题的方法.

样例 20-4. 规避子 shell

```

1.  #!/bin/bash
2.  # avoid-subshell.sh
3.  # Matthew Walker 的建议.
4.
5.  Lines=0
6.
7.  echo
8.
9.  cat myfile.txt | while read line;
10.
11.      do {
12.          echo $line
13.          (( Lines++ )); # 递增变量的值趋近外层循环
14.                        # 使用子 shell 会有问题.
15.      }
16.      done
17.  echo "Number of lines read = $Lines"      # 0
18.                                          # 报错!
19.
20.  echo "-----"
21.
22.
23.  exec 3<> myfile.txt
24.  while read line <&3
25.  do {
26.      echo "$line"
27.      (( Lines++ )); # 递增变量的值趋近外层循环.
28.                  # 没有子 shell, 就不会有问题.
29.  }
30.  done
31.  exec 3>&-
32.
33.  echo "Number of lines read = $Lines"      # 8
34.
35.  echo
36.
37.  exit 0
38.

```

```
39. # 下面的行并不在脚本里.  
40.  
41. $ cat myfile.txt  
42.  
43. Line 1.  
44. Line 2.  
45. Line 3.  
46. Line 4.  
47. Line 5.  
48. Line 6.  
49. Line 7.  
50. Line 8.
```

20.2 重定向代码块

- 20.2 重定向代码块

20.2 重定向代码块

有如 `while`, `until`, 和 `for` 循环, 甚至 `if/then` 也可以重定向 标准输入 测试代码块. 甚至连一个函数都可以用这个方法进行重定向 (见 样例 24-11). 代码块的末尾部分的 “<” 就是用来完成这个的.

样例 20-5. `while` 循环的重定向

```

1. #!/bin/bash
2. # redir2.sh
3.
4. if [ -z "$1" ]
5. then
6.     Filename=names.data      # 如果不指定文件名的默认值.
7. else
8.     Filename=$1
9. fi
10. #+ Filename=${1:-names.data}
11. # can replace the above test (parameter substitution).
12.
13. count=0
14.
15. echo
16.
17. while [ "$name" != Smith ] # 为什么变量 "$name" 加引号?
18. do
19.     read name              # 从 $Filename 读取值, 而不是 标准输入.
20.     echo $name
21.     let "count += 1"
22. done <"$Filename"         # 重定向标准输入到文件 $Filename.
```

```

23. #      ^^^^^^^^^^^^^^^
24.
25. echo; echo "$count names read"; echo
26.
27. exit 0
28.
29. # 注意在一些老的脚本语言中,
30. #+ 循环的重定向会跑在子 shell 的环境中.
31. # 因此, $count 返回 0, 在循环外已经初始化过值.
32. # Bash 和 ksh *只要可能* 会避免启动子 shell ,
33. #+ 所以这个脚本作为样例运行成功.
34. # (感谢 Heiner Steven 指出这点.)
35.
36. # 然而 . . .
37. # Bash 有时候 *能* 在 "只读的 while" 循环启动子进程 ,
38. #+ 不同于 "while" 循环的重定向.
39.
40. abc=hi
41. echo -e "1\n2\n3" | while read l
42.     do abc="$l"
43.         echo $abc
44.     done
45. echo $abc
46.
47. # 感谢, Bruno de Oliveira Schneider 上面的演示代码.
48. # 也感谢 Brian Onn 纠正了注释的错误.

```

样例 20-6. 另一种形式的 while 循环重定向

```

1. #!/bin/bash
2.
3. # 这是之前的另一种形式的脚本.
4.
5. # Heiner Steven 提议在重定向循环时候运行在子 shell 可以作为一个变通方案
6. #+ 因此直到循环终止时循环内部的变量不需要保证他们的值
7.
8.
9. if [ -z "$1" ]

```



```

10. then
11.     Filename=names.data      # 如果不指定文件名的默认值.
12. else
13.     Filename=$1
14. fi
15.
16.
17. exec 3<&0                    # 保存标准输入到文件描述符 3.
18. exec 0<"$Filename"         # 重定向标准输入.
19.
20. count=0
21. echo
22.
23.
24. while [ "$name" != Smith ]
25. do
26.     read name                # 从重定向的标准输入($Filename)读取值.
27.     echo $name
28.     let "count += 1"
29. done                          # 从 $Filename 循环读
30.                             #+ 因为第 20 行.
31.
32. # 这个脚本的早期版本在 "while" 循环 done <"$Filename" 终止
33. # 练习:
34. # 为什么这个没必要?
35.
36.
37. exec 0<&3                    # 恢复早前的标准输入.
38. exec 3<&-                    # 关闭临时的文件描述符 3.
39.
40. echo; echo "$count names read"; echo
41.
42. exit 0

```

样例 20-7. until 循环的重定向

```

1. #!/bin/bash
2. # 同先前的脚本一样, 不过用的是 "until" 循环.

```

```

3.
4.  if [ -z "$1" ]
5.  then
6.      Filename=names.data          # 如果不指定文件的默认值.
7.  else
8.      Filename=$1
9.  fi
10.
11. # while [ "$name" != Smith ]
12. until [ "$name" = Smith ]        # 变 != 为 =.
13. do
14.     read name                    # 从 $Filename 读取值, 而不是标准输入.
15.     echo $name
16. done <"$Filename"                # 重定向标准输入到文件 "$Filename".
17. #      ^^^^^^^^^^^^^^^
18.
19. # 和之前的 "while" 循环样例相同的结果.
20.
21. exit 0

```

样例 20-8. for 循环的重定向

```

1.
2.  #!/bin/bash
3.
4.  if [ -z "$1" ]
5.  then
6.      Filename=names.data          # 如果不指定文件的默认值.
7.  else
8.      Filename=$1
9.  fi
10.
11. line_count=`wc $Filename | awk '{ print $1 }'`
12. #          目标文件的行数.
13. #
14. # 非常作和不完善, 然而这只是证明 "for" 循环中的重定向标准输入是可行的
15. #+ 如果你足够聪明的话.
16. #

```

```

17. # 简介的做法是      line_count=$(wc -l < "$Filename")
18.
19.
20. for name in `seq $line_count` # 回忆下 "seq" 可以输入数组序列.
21. # while [ "$name" != Smith ] -- 比 "while" 循环更复杂的循环 --
22. do
23.     read name                # 从 $Filename 读取值, 而不是标准输入.
24.     echo $name
25.     if [ "$name" = Smith ]    # 这需要所有这些额外的设置.
26.     then
27.         break
28.     fi
29. done <"$Filename"           # 重定向标准输入到文件 "$Filename".
30. #      ^^^^^^^^^^^^^^^
31.
32. exit 0

```

我们可以修改先前的样例也可以重定向循环的输出。

样例 20-9. for 循环的重定向（同时重定向标准输入和标准输出）

```

1. #!/bin/bash
2.
3. if [ -z "$1" ]
4. then
5.     Filename=names.data      # 如果不指定文件的默认值.
6. else
7.     Filename=$1
8. fi
9.
10. Savefile=$Filename.new      # 报错的结果的文件名.
11. FinalName=Jonah            # 停止 "read" 的终止字符.
12.
13. line_count=`wc $Filename | awk '{ print $1 }'` # 目标文件行数.
14.
15.
16. for name in `seq $line_count`
17. do

```

```

18.   read name
19.   echo "$name"
20.   if [ "$name" = "$FinalName" ]
21.   then
22.       break
23.   fi
24. done < "$Filename" > "$Savefile"      # 重定向标准输入到文件 $Filename,
25. #   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^   并且报错结果到备份文件.
26.
27. exit 0

```

样例 20-10. if/then test的重定向

```

1.  #!/bin/bash
2.
3.  if [ -z "$1" ]
4.  then
5.      Filename=names.data    # 如果不指定文件的默认值.
6.  else
7.      Filename=$1
8.  fi
9.
10. TRUE=1
11.
12. if [ "$TRUE" ]             # if true    和    if :    都可以工作.
13. then
14.     read name
15.     echo $name
16. fi <"$Filename"
17. #   ^^^^^^^^^^^^^^^^^
18.
19. # 只读取文件的首行.
20. # "if/then" test 除非嵌入在循环内部否则没办法迭代.
21.
22. exit 0

```

样例 20-11. 上述样例的数据文件 names.data

```
1.
2. Aristotle
3. Arrhenius
4. Belisarius
5. Capablanca
6. Dickens
7. Euler
8. Goethe
9. Hegel
10. Jonah
11. Laplace
12. Maroczy
13. Purcell
14. Schmidt
15. Schopenhauer
16. Semmelweiss
17. Smith
18. Steinmetz
19. Tukhashevsky
20. Turing
21. Venn
22. Warshawski
23. Znosko-Borowski
24.
25. #+ 这是 "redir2.sh", "redir3.sh", "redir4.sh", "redir4a.sh",
    "redir5.sh" 的数据文件.
```

代码块的标准输出的重定向影响了保存到文件的输出。见样例 [样例 3-2](#)。

嵌入文档 是种特别的重定向代码块的方法。既然如此,它使得在 `while` 循环的标准输入里传入嵌入文档的输出变得可能。

```
1. # 这个样例来自 Albert Siersema
2. # 得到了使用许可 (感谢!).
3.
4. function doesOutput()
```

```
5.  # 当然这也是个外部命令.
6.  # 这里用函数进行演示会更好一点.
7.  {
8.    ls -al *.jpg | awk '{print $5,$9}'
9.  }
10.
11.
12.  nr=0          # 我们希望在 'while' 循环里可以操作这些
13.  totalSize=0   #+ 并且在 'while' 循环结束时看到改变.
14.
15.  while read fileSize fileName ; do
16.    echo "$fileName is $fileSize bytes"
17.    let nr++
18.    totalSize=$((totalSize+fileSize))  # Or: "let
    totalSize+=fileSize"
19.  done<<EOF
20.  $(doesOutput)
21.  EOF
22.
23.  echo "$nr files totaling $totalSize bytes"
```

20.3 应用程序

- [20.3 应用程序](#)

20.3 应用程序

使用 I/O 重定向可以同时解析和固定命令输出的片段(see [样例 15-7](#))。这也使得可以生成报告和日志文件。

样例 20-12. 日志记录事件

```
1. #!/bin/bash
2. # logevents.sh
3. # 作者: Stephane Chazelas.
4. # 用于 ABS 许可指南.
5.
6. # 事件记录到文件.
7. # 必须 root 身份执行 (可以写入 /var/log).
8.
9. ROOT_UID=0      # 只有 $UID 为 0 的用户具有 root 权限.
10. E_NOTROOT=67    # 非 root 会报错.
11.
12.
13. if [ "$UID" -ne "$ROOT_UID" ]
14. then
15.     echo "Must be root to run this script."
16.     exit $E_NOTROOT
17. fi
18.
19.
20. FD_DEBUG1=3
21. FD_DEBUG2=4
22. FD_DEBUG3=5
23.
24. # === 取消下面两行注释来激活脚本. ===
25. # LOG_EVENTS=1
```

```

26. # LOG_VARS=1
27.
28.
29. log() # 时间和日期写入日志文件.
30. {
31. echo "$(date)  $" ">7      # *追加* 日期到文件.
32. #      ^^^^^^^  命令替换
33.                                     # 见下文.
34. }
35.
36.
37.
38. case $LOG_LEVEL in
39. 1) exec 3>&2          4> /dev/null 5> /dev/null;;
40. 2) exec 3>&2          4>&2          5> /dev/null;;
41. 3) exec 3>&2          4>&2          5>&2;;
42. *) exec 3> /dev/null 4> /dev/null 5> /dev/null;;
43. esac
44.
45. FD_LOGVARS=6
46. if [[ $LOG_VARS ]]
47. then exec 6>> /var/log/vars.log
48. else exec 6> /dev/null          # 清空输出.
49. fi
50.
51. FD_LOGEVENTS=7
52. if [[ $LOG_EVENTS ]]
53. then
54. # exec 7 >(exec gawk '{print strftime(), $0}' >>
    /var/log/event.log)
55. # 上述行在最近高于 bash 2.04 版本会失败, 为什么?
56. exec 7>> /var/log/event.log      # 追加到 "event.log".
57. log                             # 写入时间和日期.
58. else exec 7> /dev/null          # 清空输出.
59. fi
60.
61. echo "DEBUG3: beginning" >&${FD_DEBUG3}
62.

```



```
63. ls -l >&5 2>&4 # 命令1 >&5 2>&4
64.
65. echo "Done" # 命令2
66.
67. echo "sending mail" >&${FD_LOGEVENTS}
68. # 输出信息 "sending mail" 到文件描述符 #7.
69.
70.
71. exit 0
```

22. 限制模式的Shell

- 第二十二章． 限制模式的Shell
 - 限制模式下被禁用的命令
 - 例 22-1． 在限制模式运行脚本

第二十二章． 限制模式的Shell

限制模式下被禁用的命令

- 在限制模式下运行一个脚本或部分脚本将禁用一些命令，尽管这些命令在正常模式下是可用的。这是个安全措施，可以限制脚本用户的权限，减少运行脚本可能带来的损害。

被禁用的命令和功能：

- 使用 `cd` 来改变工作目录。
- 修改 `$PATH`, `$SHELL`, `$BASH_ENV` 或 `$ENV` 等环境变量
- 读取或修改 `$SHELLOPTS`, `shell`环境选项。
- 输出重定向。
- 调用包含 `/` 的命令。
- 调用 `exec` 来替代`shell`进程。
- 其他各种会造成混乱或颠覆脚本用途的命令。
- 在脚本中跳出限制模式。

例 22-1． 在限制模式运行脚本

```
1. #!/bin/bash
2.
3. # 在脚本开头用#!/bin/bash -r
4. #+ 可以让整个脚本在限制模式运行。
5.
```

```
6.  echo
7.
8.  echo "改变目录。"
9.  cd /usr/local
10. echo "现在是在 `pwd`"
11. echo "回到家目录。"
12. cd
13. echo "现在是在 `pwd`"
14. echo
15.
16. # 到此为止一切都是正常的，非限制模式。
17.
18. set -r
19. # set --restricted 效果相同。
20. echo "==> 现在是限制模式 <=="
21.
22. echo
23. echo
24.
25. echo "在限制模式试图改变目录。"
26. cd ..
27. echo "依旧在 `pwd`"
28.
29. echo
30. echo
31.
32. echo "\$SHELL = $SHELL"
33. echo "试图在限制模式改变Shell 。"
34. SHELL="/bin/ash"
35. echo
36. echo "\$SHELL= $SHELL"
37.
38. echo
39. echo
40.
41. echo "试图在限制模式重定向输出内容。"
42. ls -l /usr/bin > bin.files
43. ls -l bin.files      # 尝试列出试图创建的文件。
```

```
44.  
45.  echo  
46.  
47.  exit 0
```

23. 进程替换

- [第二十三章． 进程替换](#)

第二十三章． 进程替换

用管道 将一个命令的 标准输出 输送到另一个命令的 标准输入 是个强大的技术。但是如果你需要用管道输送多个命令的 标准输出 怎么办？这时候 进程替换 就派上用场了。

进程替换 把一个（或多个）进程 的输出送到另一个进程的 标准输入。

样板

命令列表要用括号括起来

1. `>(command_list)`
2. `<(command_list)`

进程替换使用 `/dev/fd/<n>` 文件发送括号内进程的结果到另一个进程。[1]



“<”或“>”与括号之间没有空格，加上空格或报错。

1. `bash$ echo >(true)`
2. `/dev/fd/63`
- 3.
4. `bash$ echo <(true)`
5. `/dev/fd/63`
- 6.
7. `bash$ echo >(true) <(true)`
8. `/dev/fd/63 /dev/fd/62`
- 9.

```

10. bash$ wc <(cat /usr/share/dict/linux.words)
11.   483523   483523  4992010 /dev/fd/63
12.
13. bash$ grep script /usr/share/dict/linux.words | wc
14.     262     262     3601
15.
16. bash$ wc <(grep script /usr/share/dict/linux.words)
17.     262     262     3601 /dev/fd/63

```



Bash用两个文件描述符创建管道，`--fIn` 和 `fOut--`。 `true` 的 `标准输入` 连接 `fOut(dup2(fOut, 0))`，然后Bash 传递一个 `/dev/fd/fIn` 参数给 `echo`。在不使用 `/dev/fd/<n>` 的系统里，Bash可以用临时文件（感谢 S.C. 指出这点）。

进程替换可以比较两个不同命令的输出，或者同一个命令使用不同选项的输出。

```

1. bash$ comm <(ls -l) <(ls -al)
2. total 12
3. -rw-rw-r--    1 bozo bozo    78 Mar 10 12:58 File0
4. -rw-rw-r--    1 bozo bozo    42 Mar 10 12:58 File2
5. -rw-rw-r--    1 bozo bozo   103 Mar 10 12:58 t2.sh
6.      total 20
7.      drwxrwxrwx    2 bozo bozo   4096 Mar 10 18:10 .
8.      drwx-----   72 bozo bozo   4096 Mar 10 17:58 ..
9.      -rw-rw-r--    1 bozo bozo    78 Mar 10 12:58 File0
10.     -rw-rw-r--    1 bozo bozo    42 Mar 10 12:58 File2
11.     -rw-rw-r--    1 bozo bozo   103 Mar 10 12:58 t2.sh

```

进程替换可以比较两个目录的内容——来检查哪些文件在这个目录而不在那个目录。

```

1. diff <(ls $first_directory) <(ls $second_directory)

```

进程替换的一些其他用法：

```
1. read -a list <<( od -Ad -w24 -t u2 /dev/urandom )
2. # 从 /dev/urandom 读取一个随机数列表
3. #+ 用 "od" 处理
4. #+ 输送到 "read" 的标准输入. . .
5. # 来自 "insertion-sort.bash" 示例脚本。
6. # 致谢：JuanJo Ciarlante。
```

```
1. PORT=6881 # bittorrent (BT端口)
2.
3. # 扫描端口，确保没有恶意行为
4. netcat -l $PORT | tee>(md5sum ->mydata-orig.md5) |
5. gzip | tee>(md5sum - | sed 's/-$/mydata.lz2/'>mydata-
   gz.md5)>mydata.gz
6.
7. # 检查解压缩结果：
8. gzip -d<mydata.gz | md5sum -c mydata-orig.md5)
9. # 对原件的MD5校验用来检查标准输入，并且探测压缩当中出现的问题。
10.
11. # Bill Davidsen 贡献了这个例子
12. #+ (ABS指南作者做了轻微修改)。
```

```
1. cat <(ls -l)
2. # 等价于    ls -l | cat
3.
4. sort -k 9 <(ls -l /bin) <(ls -l /usr/bin) <(ls -l /usr/X11R6/bin)
5. # 列出 3 个主要 'bin' 目录的文件，按照文件名排序。
6. # 注意，有三个（数一下）单独的命令输送给了 'sort'。
7.
8. diff <(command1) <(command2) # 比较命令输出结果的不同之处。
9.
10. tar cf >(bzip2 -c > file.tar.bz2) $directory_name
11.
12. # 调用 "tar cf /dev/fd/?? $directory_name", 然后 "bzip2 -c >
   file.tar.bz2"。
13. #
```

```

14. # 因为 /dev/fd/<n> 系统特性
15. # 不需要在两个命令之间使用管道符
16. #
17. # 这个可以模拟
18. #
19. bzip2 -c < pipe > file.tar.bz2&
20. tar cf pipe $directory_name
21. rm pipe
22. # 或者
23. exec 3>&1
24. tar cf /dev/fd/4 $directory_name 4>&1 >&3 3>&- | bzip2 -c >
    file.tar.bz2 3>&-
25. exec 3>&-
26.
27. # 致谢：Stéphane Chazelas

```

在子shell中 `echo` 命令用管道输送给 `while-read` 循环时会出现问题，下面是避免的方法：

例23-1 不用 `fork` 的代码块重定向。

```

1. #!/bin/bash
2.
3. # wr-ps.bash: 使用进程替换的 while-read 循环。
4.
5. # 示例由 Tomas Pospisek 贡献。
6. # （ABS指南作者做了大量改动。）
7.
8. echo
9.
10. echo "random input" | while read i
11. do
12.     global=3D": Not available outside the loop."
13.     # ... 因为在子 shell 中运行。
14. done
15.
16. echo "\$global (从子进程之外) = $global"

```



```

17. # $global (从子进程之外) =
18.
19. echo; echo "--"; echo
20.
21. while read i
22. do
23.     echo $i
24.     global=3D": Available outside the loop."
25.     # ... 因为没有在子 shell 中运行。
26. done <<( echo "random input" )
27. #     ^ ^
28.
29. echo "\$global (使用进程替换) = $global"
30. # 随机输入
31. # $global (使用进程替换)= 3D: Available outside the loop.
32.
33.
34. echo; echo "#####"; echo
35.
36.
37.
38. # 同样道理 . . .
39.
40. declare -a inloop
41. index=0
42. cat $0 | while read line
43. do
44.     inloop[$index]="$line"
45.     ((index++))
46.     # 在子 shell 中运行, 所以 ...
47. done
48. echo "OUTPUT = "
49. echo ${inloop[*]}          # ... 什么也没有显示。
50.
51.
52. echo; echo "--"; echo
53.
54.

```

```

55. declare -a outloop
56. index=0
57. while read line
58. do
59.     outloop[$index]="$line"
60.     ((index++))
61.     # 没有在子 shell 中运行, 所以 ...
62. done < <( cat $0 )
63. echo "OUTPUT = "
64. echo ${outloop[*]}           # ... 整个脚本的结果显示出来。
65.
66. exit $?

```

下面是个类似的例子。

例 23-2. 重定向进程替换的输出到一个循环内

```

1. #!/bin/bash
2. # psub.bash
3. # 受 Diego Molina 启发（感谢！）。
4.
5. declare -a array0
6. while read
7. do
8.     array0[${#array0[@]}]="$REPLY"
9. done < <( sed -e 's/bash/CRASH-BANG!/' $0 | grep bin | awk '{print
    $1}' )
10. # 由进程替换来设置'read'默认变量（$REPLY）。
11. #+ 然后将变量复制到一个数组。
12.
13. echo "${array0[@]}"
14.
15. exit $?
16.
17. # ===== #
18. # 运行结果：
19. bash psub.bash
20.

```

```
21.  #!/bin/CRASH-BANG! done #!/bin/CRASH-BANG!
```

一个读者发来一个有趣的进程替换例子，如下：

```
1.  # SuSE 发行版中提取的脚本片段：
2.
3.  # -----#
4.  while read des what mask iface; do
5.  # 一些命令 ...
6.  done < <(route -n)
7.  #    ^ ^  第一个 < 是重定向，第二个是进程替换。
8.
9.  # 为了测试，我们让它来做点儿事情。
10. while read des what mask iface; do
11.     echo $des $what $mask $iface
12. done < <(route -n)
13.
14. # 输出内容：
15. # Kernel IP routing table
16. # Destination Gateway Genmask Flags Metric Ref Use Iface
17. # 127.0.0.0 0.0.0.0 255.0.0.0 U 0 0 0 lo
18. # -----#
19.
20. # 正如 Stéphane Chazelas 指出的，
21. #+ 一个更容易理解的等价代码如下：
22. route -n |
23.     while read des what mask iface; do # 通过管道输出设置的变量
24.         echo $des $what $mask $iface
25.     done # 这段代码的结果更上面的相同。
26.         # 但是，Ulrich Gayer 指出 . . .
27.         #+ 这段简化版等价代码在 while 循环里用了子 shell，
28.         #+ 因此当管道终止时变量都消失了。
29.
30. # -----#
31.
32. # 然而，Filip Moritz 说上面的两个例子有一个微妙的区别，
33. #+ 见下面的代码
34.
```

```

35. (
36. route -n | while read x; do ((y++)); done
37. echo $y # $y is still unset
38.
39. while read x; do ((y++)); done < <(route -n)
40. echo $y # $y has the number of lines of output of route -n
41. )
42.
43. # 更通俗地说（译者注：原文本行少了注释符）
44. (
45. : | x=x
46. # 似乎启动了子 shell ， 就像
47. : | ( x=x )
48. # 而
49. x=x < <( :)
50. # 并没有。
51. )
52. # 这个方法在解析 csv 和类似格式时很有用。
53. # 也就是在效果上，原始 SuSE 系统的代码片段就是做这个用的。

```

注解 [1]

这个与命名管道（使用临时文件）的效果相同，而且事实上，进程替换也曾经用过命名管道。

26. 列表结构

- 第二十六章． 列表结构
 - 链接多个命令

第二十六章． 列表结构

and 列表 和 *or* 列表 结构提供了连续执行若干命令的方法，可以有效地替换复杂的嵌套 *if/then* ，甚至 *case* 语句。

链接多个命令

and 列表

```
command-1 && command-2 && command-3 && ... command-n
```

只要前一个命令返回 *true* (即 0) ，每一个命令就依次执行。当第一个 *false* (即 非0) 返回时，命令链条即终止 (第一个返回 *false* 的命令是最后一个执行的) 。

在YongYe早期版本的俄罗斯方块游戏脚本里，一个有趣的双条件 *and* 列表 用法：

```
1. equation()
2.
3. { # core algorithm used for doubling and halving the coordinates
4.   [[ ${cdx} ]] && ((y=cy+(ccy-cdy){2}2))
5.   eval ${1}+=\ "${x} ${y} \"
6. }
```

例 26-1． 使用 *and* 列表 来测试命令行参数

```
1. #!/bin/bash
2. # and list
```

```

3.
4.  if [ ! -z "$1" ] && echo "Argument #1 = $1" && [ ! -z "$2" ] && \
5.  #             ^^             ^^             ^^
6.  echo "Argument #2 = $2"
7.  then
8.      echo "At least 2 arguments passed to script."
9.      # 链条内的所有命令都返回 true。
10. else
11.     echo "Fewer than 2 arguments passed to script."
12.     # 链条内至少有一个命令返回 false。
13. fi
14. # 注意： "if [ ! -z $1 ]" 是好用的，但是宣传与之等同的
15. #   "if [ -n $1 ]" 并不好用。
16. #   不过，用引号就能解决问题，
17. #   "if [ -n "$1" ]" 好用（译者注：原文本行内第一个引号位置错了）。
18. #           ^  ^   小心！
19. # 被测试的变量放在引号内总是最好的选择。
20.
21.
22. # 下面的代码功能一样，用的是“纯粹”的 if/then 语句。
23. if [ ! -z "$1" ]
24. then
25.     echo "Argument #1 = $1"
26. fi
27. if [ ! -z "$2" ]
28. then
29.     echo "Argument #2 = $2"
30.     echo "At least 2 arguments passed to script."
31. else
32.     echo "Fewer than 2 arguments passed to script."
33. fi
34. # 比起用“and 列表”要更长、更笨重。
35.
36.
37. exit $?

```

例 26-2. 使用 *and* 列表 来测试命令行参数2

```

1.  #!/bin/bash
2.
3.  ARGS=1          # 预期的参数数量。
4.  E_BADARGS=85    # 参数数量错误时返回的值。
5.
6.  test $# -ne $ARGS && \
7.  #      ^^^^^^^^^^^^^ 条件 #1
8.  echo "Usage: `basename $0` $ARGS argument(s)" && exit $E_BADARGS
9.  #                                     ^^
10. # 如果条件 #1 结果为 true (传递给脚本的参数数量错误),
11. #+ 那么执行本行剩余的命令, 脚本终止。
12.
13. # 下面的代码行只有在上面的测试失败时才执行。
14. echo "Correct number of arguments passed to this script."
15.
16. exit 0
17.
18. # 如果要检查退出值, 脚本终止后运行 "echo $?".

```

当然, *and* 列表 也可以给变量设置默认值。

```

1.  arg1=$@ && [ -z "$arg1" ] && arg1=DEFAULT
2.
3.          # 如果有命令行参数, 则把参数值赋给 $arg1 。
4.          # 但是... 如果没有参数, 则使用DEFAULT给 $arg1 赋值。

```

or 列表

```

1.  command-1 || command-2 || command-3 || ... command-n

```

只要前一个命令返回`false`, 每一个命令就依次执行。当第一个`true`返回时, 命令链条即终止(第一个返回`true`的命令是最后一个执行的)。很明显它与“*and* 列表”相反。

例 26-3. *or* 列表 与 *and* 列表 结合使用

```

1.  #!/bin/bash
2.
3.  #  delete.sh, 不那么巧妙的文件删除工具。
4.  #  用法:  delete 文件名
5.
6.  E_BADARGS=85
7.
8.  if [ -z "$1" ]
9.  then
10.     echo "Usage: `basename $0` filename"
11.     exit $E_BADARGS  # No arg? Bail out.
12. else
13.     file=$1          # Set filename.
14. fi
15.
16.
17. [ ! -f "$file" ] && echo "File \"$file\" not found. \
18. Cowardly refusing to delete a nonexistent file."
19. # AND 列表, 如果文件不存在则显示出错信息。
20. # 注意, echo 消息内容分成了两行, 中间通过转义符 (\) 连接。
21.
22. [ ! -f "$file" ] || (rm -f $file; echo "File \"$file\" deleted.")
23. # OR 列表, 删除存在的文件。
24.
25. # 注意上面的逻辑颠倒。 Note logic inversion above.
26. # “AND 列表” 在得到 true 时执行, “OR 列表”在得到 false 时执行。
27.
28. exit $?

```



如果 *or* 列表 第一个命令返回 true, 它会执行。

```

1.  # ==> 下面的代码段来自 /etc/rc.d/init.d/single
2.  #+==> 作者 Miquel van Smoorenburg
3.  #+==> 说明了 "and" 和 "or" 列表。
4.  # ==> 带箭头的注释是本文作者添加的。
5.
6.  [ -x /usr/bin/clear ] && /usr/bin/clear

```



```

7.      # ==> 如果 /usr/bin/clear 存在, 则调用它。
8.      # ==> 调用命令之前检查它是否存在,
9.      #+=> 可以避免出错消息和其他怪异的结果。
10.
11.     # ==> . . .
12.
13.    # If they want to run something in single user mode, might as well
    run it...
14.    for i in /etc/rc1.d/S[0-9][0-9]* ; do
15.        # 检查脚本是否存在。
16.        [ -x "$i" ] || continue
17.        # ==> 如果对应的文件在 $PWD 里*没有*找到,
18.        #+=> 则跳回到循环顶端“继续运行”。
19.
20.        # 丢弃备份文件和 rpm 生成的文件。
21.        case "$i" in
22.            *.rpmsave|*.rpmorig|*.rpmnew|*~|*.orig)
23.                continue;;
24.        esac
25.        [ "$i" = "/etc/rc1.d/S00single" ] && continue
26.        # ==> 设置脚本名, 但先不要执行
27.        $i start
28.    done
29.
30.     # ==> . . .

```



and 列表 或 *or* 列表 的退出状态就是最后一个执行的命令的退出状态。

聪明地结合 *and* 列表 和 *or* 列表 是可能的, 但是程序逻辑会很容易地变得令人费解, 需要密切注意操作符优先规则, 而且, 会带来大量的调试工作。

```

1.  false && true || echo false          # false
2.

```

```
3. # 下面的代码结果相同
4. ( false && true ) || echo false      # false
5. # 但这个就不同了
6. false && ( true || echo false )      # (什么都不显示)
7.
8. # 注意语句是从左到右组合和解释的。
9.
10. # 通常情况下最好避免这种复杂性。
11.
12. # 感谢, S.C.
```

例 A-7 和 例 7-4 解释了用 *and* 列表 / *or* 列表 来测试变量。

24 函数

- [24 函数](#)
 - [本章目录](#)

24 函数

本章目录

- [24.1 复杂函数和函数复杂性](#)
- [24.2 局部变量](#)
- [24.3 不使用局部变量的递归](#)

和其它“真正”的编程语言一样，Bash也有函数，尽管它在实现方面有一些限制。一个函数就是一个子程序，实现一系列操作的[代码块](#)，执行一个特定任务的“黑盒子”。有重复代码的地方，当一个过程只需要轻微修改任务就会重复执行的时候，那么你就需要考虑使用函数了。

```
1. function function_name {  
2.   command...  
3. }
```

或者

```
1. function_name () {  
2.   command...  
3. }
```

第二种形式可能会更受C程序员的喜爱（并且它更具有[可移植性](#)）。在C语言里面，函数的圆括号可以出现在第二行。

```
1. function_name () {
```

```
2. command...
3. }
```



一个函数可能被“压缩”到一个单独行里。

```
1. fun () { echo "This is a function"; echo; }
2. # ^ ^
```

然而，在这种情况下，函数里的最后一个命令必须跟有一个分号。

```
1. fun () { echo "This is a function"; echo } # Error!
2. # ^
3. fun2 () { echo "Even a single-command function? Yes!"; }
4. # ^
```

只需要引用函数名字就可以调用或者触发函数。一个函数调用相当于一个命令。

例子 24-1. 简单的函数

```
1. #!/bin/bash
2. # ex59.sh: 练习函数(简单的).
3.
4. JUST_A_SECOND=1
5.
6. funky ()
7. { # 这是一个简单的函数
8.     echo "This is a funky function."
9.     echo "Now exiting funky function."
10. } # 函数必须在调用前声明.
11.
12.
13. fun ()
14. { # 一个稍微复杂点的函数.
15.     i=0
16.     REPEATS=30
```

```

17.
18.     echo
19.     echo "And now the fun really begins."
20.     echo
21.
22.     sleep $JUST_A_SECOND    # Hey, 等一秒钟!
23.     while [ $i -lt $REPEATS ]
24.     do
25.         echo "------FUNCTIONS----->"
26.         echo "<-----ARE-----"
27.         echo "<-----FUN----->"
28.         echo
29.         let "i+=1"
30.     done
31. }
32.
33. # 现在, 调用这些函数.
34.
35. funky
36. fun
37.
38. exit $?

```

函数定义必须在第一次函数调用之前。没有声明函数的方法，比如像C语言中一样。

```

1.  f1
2.  # 将会产生一个错误消息, 因为"f1"函数还没有定义。
3.
4.  declare -f f1    # 这样也不会有帮助。
5.  f1              # 仍然会产生一个错误消息。
6.
7.  # 然而...
8.
9.
10. f1 () {
11.     echo "Calling function \"f2\" from within function \"f1\"."

```

```

12.     f2
13. }
14.
15. f2 () {
16.     echo "Function \"f2\"."
17. }
18.
19. f1 # 在此之前，事实上函数“f2”是没有被调用的，
20.    #+ 尽管在它定义之前被引用了。
21.    # 这是可以的。
22.    # 感谢，S.C.

```



函数不能为空！

```

1.  #!/bin/bash
2.  # empty-functionn.sh
3.
4.  empty ()
5.  {
6.  }
7.
8.  exit 0 # 这里将不会退出！
9.
10.
11. # $ sh empty-function.sh
12. # empty-function.sh: line 6: syntax error near unexpected token `}'
13. # empty-function.sh: line 6: `}'
14.
15. # $ echo $?
16. # 2
17.
18. # 请注意，只包含注释的函数也是空函数。
19.
20. func ()
21. {
22.     # 注释 1.
23.     # 注释 2.

```

```

24.      # 这仍然是一个空函数。
25.      # 感谢, Mark Bova将这一点指出来。
26.  }
27.  # 结果会出现和上面一样的错误信息。
28.
29.  # 然而 ...
30.
31.  not_quite_empty ()
32.  {
33.      illegal_command
34.  } # 一个包含这个函数的脚本将不会出错
35.      #+ 只要这个函数没有被调用。
36.  not_empty ()
37.  {
38.      :
39.  } # 包含一个 : (空命令符), 这样是可以的。
40.
41.  # 感谢, Dominick Geyer 和 Thiemo Kellner.

```

甚至, 把一个函数嵌套在另外一个函数里也是可行的, 尽管这并没有什么用。

```

1.  f1 ()
2.  {
3.      f2 () # 嵌套函数
4.      {
5.          echo "Function \"f2\", inside \"f1\"."
6.      }
7.  }
8.
9.  f2 # 将会产生一个错误消息。
10.   # 即使有一个前置的 "declare -f f2" 也不会有什么作用。
11.
12.  echo
13.
14.  f1 # 不会做任何事情, 因为调用"f1"的时候, 并不会自动调用"f2"。
15.   # 现在, 调用"f2"是可以的,

```

```

16.      #+ 因为通过调用“f1”，它的定义现在已是可见的。
17.
18.      # 感谢，S.C.

```

函数定义可能出现在不太可能出现的地方，甚至出现在本应该是命令出现的地方。

```

1.  ls -l | foo() { echo "foo"; } # 可行的，尽管没有什么作用。
2.
3.
4.  if [ "$USER" = bozo ]
5.  then
6.      bozo_greet () # 函数定义嵌套在if/then的结构体中。
7.      {
8.          echo "Hello, Bozo."
9.      }
10. fi
11.
12. bozo_greet          # 只有Bozo用户工作，其它用户会得到一个错误消息。
13.
14.
15. # 在某些场景中，像下面这些东西可能会很有用。
16. NO_EXIT=1          # 将会激活下面的函数定义。
17.
18. [[ $NO_EXIT -eq 1 ]] && exit() { true; } # 函数定义出现在“与列表”中。
19. # 如果 $NO_EXIT 等于 1，定义 "exit ()"。
20. # 通过把exit函数别名为“true”，这样把内置的exit命令给禁用了。
21.
22. exit # 调用 "exit ()" 函数，而不是内置的 "exit" 命令。
23.
24.
25. # 或者，类似地：
26. filename=file1
27.
28. [ -f "$filename" ] &&
29. foo () { rm -f "$filename"; echo "File \"$filename\" deleted."; } ||

```



```

30. foo () { echo "File "$filename" not found."; touch bar; }
31.
32. foo
33.
34. # 感谢, S.C. 和 Christopher Head

```

函数名字可以呈现各种奇怪的形式。

```

1. _(){ for i in {1..10}; do echo -n "$FUNCNAME"; done; echo; }
2. # ^^^^          函数名字和圆括号之间没有空格。
3. #              这并不会总是会正常工作。为什么呢？
4.
5. # 现在, 我们来调用函数。
6. _          # _____
7. #          ^^^^^^^^^^^ 10 个下划线 (10 倍的函数名字) !
8. # 一个“假”的下划线也是一个可以接受的函数名字。
9.
10. # 事实上, 一个分号也是一个可以接受的函数名字。
11.
12. :(){ echo ":"; }; :
13.
14. # 这有什么作用呢？
15. # 这是一个狡诈的方式去混淆脚本中的代码。

```

也可以参见 [Example A-56](#)

小提示：当一个函数的不同版本出现在一个脚本中，会发生什么事情呢？

```

1. # 正如Yan Chen 指出的那样,
2. # 当一个函数被多次定义的时候,
3. # 最后一个函数是被调用的那个。
4. # 然而这并不是特别有用。
5.
6. func ()
7. {
8.     echo "First version of func ()."

```

```
9.  }
10.
11. func ()
12. {
13.     echo "Second version of func ()."
14. }
15.
16. func    # 调用的是第二个 func () 函数版本。
17.
18. exit $?
19.
20. # 甚至, 可能用函数去覆盖
21. #+ 或者占用系统命令。
22. # 当然, 这并不是可取的。
```

24.1 复杂函数和函数复杂性

- 24.1 复杂函数和函数复杂性
 - 退出与返回码
 - 退出状态码
 - `return`

24.1 复杂函数和函数复杂性

函数可以处理传递给它的参数，并且能返回它的退出状态码给脚本，以便后续处理。

```
1. function_name $arg1 $arg2
```

函数通过位置来引用传递过来的参数（就好像它们是位置参数），例如，\$1，\$2，等等。

例子 24-2. 带参数的函数

```
1. #!/bin/bash
2. # 函数和参数
3.
4. DEFAULT=default                # 默认参数值。D
5.
6. func2 () {
7.     if [ -z "$1" ]              # 第一个参数长度是否为零？
8.     then
9.         echo "-Parameter #1 is zero length. -" # 或者没有参数传递进来。
10.    else
11.        echo "-Parameter #1 is \"$1\". -"
12.    fi
13.
14.    variable=${1-$DEFAULT}
15.    echo "variable = $variable"    # 这里的参数替换
```

```

16.                                     #+ 表示什么？
17.                                     # -----
18.                                     # 为了区分没有参数的情况
19.                                     #+ 和只有一个null参数的情况。
20.
21.     if [ "$2" ]
22.     then
23.         echo "-Parameter #2 is \"$2\".-"
24.     fi
25.
26.     return 0
27. }
28.
29. echo
30.
31. echo "Nothing passed."
32. func2                                     # 不带参数调用
33. echo
34.
35.
36. echo "Zero-length parameter passed."
37. func2 ""                                # 使用0长度的参数进行调用
38. echo
39.
40. echo "Null parameter passed."
41. func2 "$uninitialized_param"           # 使用未初始化的参数进行调用
42. echo
43.
44.
45. echo "One parameter passed."
46. func2 first                            # 带一个参数的调用
47. echo
48.
49. echo "Two parameters passed."
50. func2 first second                    # 带两个参数的调用
51. echo
52.
53. echo "\"\" \"second\" passed."

```

```

54. func2 "" second      # 第一个调用参数为0长度参数,
55. echo                 # 第二个是ASCII码的字符串参数。
56.
57. exit 0

```



也可以使用`shift`命令来处理传递给函数的参数（请参考[例子 33-18](#)。

但是，传递给脚本的命令行参数怎么办？在函数内部，可以看见这些命令行参数么？好，现在让我们弄清楚这个困惑。

例子 34-3. 函数以及传递给脚本的命令行参数。

```

1. #!/bin/bash
2. # func-cmdlinearg.sh
3. # 带一个命令行参数来执行这个脚本,
4. #+ 类似于 $0 arg1.
5.
6.
7. func ()
8. {
9.     echo "$1"    # 显示传递给这个函数的第一个参数。
10. }              # 命令行参数可以么？
11.
12. echo "First call to function: no arg passed."
13. echo "See if command-line arg is seen."
14. func
15. # 不！没有见到命令行参数.
16.
17. echo "===== "
18. echo
19. echo "Second call to function: command-line arg passed explicitly."
20.
21. func $1
22. # 现在，见到命令行参数了！
23.
24. exit 0

```

和其它的编程语言相比，shell脚本一般只会传值给函数。如果把变量名（事实上就是指针）作为参数传递给函数的话，那将被解释为字面含义，也就是被看做字符串。 函数只会以字面含义来解释函数参数。

[变量的间接引用](#)（请参考[例子 37-2](#)）提供了一种笨拙的机制，来将变量指针传递给函数。

例子 24-4. 将一个间接引用传递给函数

```

1.  #!/bin/bash
2.  # ind-func.sh: 将一个间接引用传递给函数。
3.
4.  echo_var ()
5.  {
6.      echo "$1"
7.  }
8.
9.  message=Hello
10. Hello=Goodbye
11.
12. echo_var "$message"           # Hello
13. # 现在，让我们传递一个间接引用给函数。
14. echo_var "${!message}"       # Goodbye
15. echo "-----"
16.
17. # 如果我们改变“hello”的值会发生什么？
18. Hello="Hello, again!"
19. echo_var "$message"          # Hello
20. echo_var "${!message}"       # Hello, again!
21.
22. exit 0

```

接下来的一个逻辑问题就是，将参数传递给函数之后，参数能否被解除引用。

例子 24-5. 对一个传递给函数的参数进行解除引用的操作

```

1.  #!/bin/bash
2.  # dereference.sh
3.  # 对一个传递给函数的参数进行解除引用的操作。
4.  # 此脚本由Bruce W. Clare编写。
5.
6.  dereference ()
7.  {
8.      y=\${$1}    # 变量名（而不是值）。
9.      echo $y      # $Junk
10.
11.      x=`eval "expr \"\$y\" "`
12.      echo $1=$x
13.      eval "$1=\"Some Different Text \"" # 赋新值。
14.  }
15.
16.  Junk="Some Text"
17.  echo $Junk "before"          # Some Text before
18.
19.  dereference Junk
20.  echo $Junk "after"           # Some Different Text after
21.
22.  exit 0

```

例子 24-6. 再来一次，对一个传递给函数的参数进行解除引用的操作

```

1.  #!/bin/bash
2.  # ref-params.sh: 解除传递给函数的参数引用。
3.  # (复杂的例子C)
4.
5.  ITERATIONS=3 # 取得输入的次数。
6.  icount=1
7.
8.  my_read () {
9.      # 用my_read varname这种形式来调用,
10.     ##+ 将之前用括号括起的值作为默认值输出,
11.     ##+ 然后要求输入一个新值。
12.

```

```

13.     local local_var
14.
15.     echo -n "Enter a value "
16.     eval 'echo -n "[$'$1'] "' # 之前的值.
17. # eval echo -n "[\$$1] "      # 更容易理解,
18.                               #+ 但会丢失用户在尾部输入的空格。
19.     read local_var
20.     [ -n "$local_var" ] && eval $1=\$local_var
21.
22.     # "与列表": 如果 "local_var" 的测试结果为true, 则把变量"$1"的值赋给
    它。
23. }
24.
25. echo
26.
27. while [ "$icount" -le "$ITERATIONS" ]
28. do
29.     my_read var
30.     echo "Entry #$icount = $var"
31.     let "icount += 1"
32.     echo
33. done
34.
35. # 感谢Stephane Chazelas 提供这个例子。
36.
37. exit 0

```

退出与返回码

退出状态码

函数返回一个值，被称为退出状态码。这和一条命令返回的[退出状态码](#)类似。退出状态码可以由**return** 命令明确指定，也可以由函数中最后一条命令的退出状态码来指定（如果成功，则返回0，否则返回非0值）。可以在脚本中使用**\$?**来引用[退出状态码](#)。因为有了这种机制，

所以脚本函数也可以像C函数一样有“返回值”。

return

终止一个函数。一个return命令¹ 可选的允许带一个整形参数，这个整形参数将作为函数的“退出状态码”返回给调用这个函数的脚本，并且这个证书也被赋值给变量\$?。

例子 24-7. 取两个数中的最大值

```


1.  #!/bin/bash
2.  # max.sh: 取两个Maximum of two integers.
3.  E_PARAM_ERR=250      # 如果传给函数的参数少于两个时，就返回这个值。
4.  EQUAL=251            # 如果两个参数相等时，就返回这个值。
5.  # 任意超出范围的
6.  #+ 参数值都可能传递到函数中。
7.
8.  max2 ()              # 返回两个数中的最大值。
9.  {                   # 注意：参与比较的数必须小于250.
10.     if [ -z "$2" ]
11.     then
12.         return $E_PARAM_ERR
13.     fi
14.
15.     if [ "$1" -eq "$2" ]
16.     then
17.         return $EQUAL
18.     else
19.         if [ "$1" -gt "$2" ]
20.         then
21.             return $1
22.         else
23.             return $2
24.         fi
25.     fi
26. }
27.

```

```

28. max2 33 34
29. return_val=$?
30.
31. if [ "$return_val" -eq $E_PARAM_ERR ]
32. then
33.     echo "Need to pass two parameters to the function."
34. elif [ "$return_val" -eq $EQUAL ]
35. then
36.     echo "The two numbers are equal."
37. else
38.     echo "The larger of the two numbers is $return_val."
39. fi
40.
41. exit 0
42. # 练习 (easy):
43. # -----
44. # 把这个脚本转化为交互脚本,
45. #+ 也就是, 修改这个脚本, 让其要求调用者输入2个数。

```

 为了让函数可以返回字符串或者是数组，可以使用一个在函数外可见的专用全局变量。

```

1. count_lines_in_etc_passwd()
2. {
3.     [[ -r /etc/passwd ]] && REPLY=$(echo $(wc -l < /etc/passwd))
4.     # 如果 /etc/passwd 可读, 让 REPLY 等于 文件的行数.
5.     # 这样就可以同时返回参数值与状态信息。
6.     # 'echo' 看上去没什么用, 可是...
7.     #+ 它的作用是删除输出中的多余空白符。
8. }
9.
10. if count_lines_in_etc_passwd
11. then
12.     echo "There are $REPLY lines in /etc/passwd."
13. else
14.     echo "Cannot count lines in /etc/passwd."
15. fi

```

```
16.
17. # 感谢, S.C.
```

例子 24-8. 将阿拉伯数字转化为罗马数字

```
1. #!/bin/bash
2. # 将阿拉伯数字转化为罗马数字。
3. # 范围: 0 - 200
4. # 比较粗糙, 但可以正常工作。
5. # 扩展范围, 并且完善这个脚本, 作为练习。
6. # 用法: roman number-to-convert
7.
8. LIMIT=200
9. E_ARG_ERR=65
10. E_OUT_OF_RANGE=66
11.
12. if [ -z "$1" ]
13. then
14.     echo "Usage: `basename $0` number-to-convert"
15. exit $E_ARG_ERR
16. fi
17.
18. num=$1
19. if [ "$num" -gt $LIMIT ]
20. then
21.     echo "Out of range!"
22.     exit $E_OUT_OF_RANGE
23. fi
24.
25. to_roman ()                # 在第一次调用函数前必须先定义它。
26. {
27.     number=$1
28.     factor=$2
29.     rchar=$3
30.     let "remainder = number - factor"
31.     while [ "$remainder" -ge 0 ]
32.     do
33.         echo -n $rchar
```

```

34.         let "number -= factor"
35.         let "remainder = number - factor"
36.     done
37.
38.     return $number
39.     # 练习:
40.     # -----
41.     # 1) 解释这个函数如何工作
42.     #     提示: 依靠不断的除, 来分割数字。
43.     # 2) 扩展函数的范围:
44.     #     提示: 使用echo和substitution命令。
45. }
46.
47. to_roman $num 100 C
48. num=$?
49. to_roman $num 90 LXXXX
50. num=$?
51. to_roman $num 50 L
52. num=$?
53. to_roman $num 40 XL
54. num=$?
55. to_roman $num 10 X
56. num=$?
57. to_roman $num 9 IX
58. num=$?
59. to_roman $num 5 V
60. num=$?
61. to_roman $num 4 IV
62. num=$?
63. to_roman $num 1 I
64. # 成功调用了转换函数。
65. # 这真的是必须的么? 这个可以简化么?
66.
67. echo
68.
69. exit

```

也可以参见[例子 11-29](#)



函数所能返回最大的正整数是255。return命令和[退出状态码](#)的概念紧密联系在一起，并且退出状态码的值受此限制。幸运的是，如果想让函数返回大整数的话，有好多种不同的[变通方法](#)能够应对这个情况。

例子24-9. 测试函数最大的返回值

```

1.  #!/bin/bash
2.  # return-test.sh
3.  # 函数所能返回的最大正整数为255.
4.
5.  return_test ()          # 传给函数什么值，就返回什么值。
6.  {
7.      return $1
8.  }
9.
10. return_test 27           # o.k.
11. echo $?                 # 返回27.
12.
13. return_test 255         # Still o.k.
14. echo $?                 # 返回 255.
15.
16. return_test 257         # 错误！
17. echo $?                 # 返回 1 (对应各种错误的返回码).
18.
19. # =====
20. return_test -151896     # 能返回一个大负数么？
21. echo $?                 # 能否返回 -151896？
22.                         # 不行！返回的是 168.
23.
24. # Bash 2.05b 之前的版本
25. #+ 允许返回大负数。
26. # 这可能是个有用的特性。
27. # Bash之后的新版本修正了这个漏洞。
28. # 这可能会影响以前所编写的脚本。
29. # 一定要小心！

```

```

30. # =====
31.
32. exit 0

```

如果你想获得大整数“返回值”的话，简单的方法就是将“要返回的值”保存到一个全局变量中。

```

1. Return_Val=    # 用于保存函数特大返回值的全局变量。
2.
3. alt_return_test ()
4. {
5.     fvar=$1
6.     Return_Val=$fvar
7.     return    # 返回 0 (成功).
8. }
9.
10. alt_return_test 1
11. echo $?          #0
12. echo "return value = $Return_Val"      #1
13.
14. alt_return_test 256
15. echo "return value = $Return_Val"      # 256
16.
17. alt_return_test 257
18. echo "return value = $Return_Val"      # 257
19.
20. alt_return_test 25701
21. echo "return value = $Return_Val"      #25701

```

一种更优雅的做法是在函数中使用echo命令将“返回值输出到 stdout”，然后用命令替换来捕捉此值。请参考[36.7小节](#) 中关于这种用法的讨论。

例子 24-10. 比较两个大整数

```

1. #!/bin/bash

```

```

2.  # max2.sh: 取两个大整数中的最大值。
3.
4.  # 这是前一个例子 "max.sh" 的修改版,
5.  #+ 这个版本可以比较两个大整数。
6.
7.  EQUAL=0                # 如果两个值相等, 那就返回这个值。
8.  E_PARAM_ERR=-99999     # 没有足够多的参数, 那就返回这个值。
9.  #          ^^^^^^      任意超出范围的参数都可以传递进来。
10.
11. max2 ()                # "返回" 两个整数中最大的那个。
12. {
13.     if [ -z "$2" ]
14.     then
15.         echo $E_PARAM_ERR
16.         return
17.     fi
18.
19.     if [ "$1" -eq "$2" ]
20.     then
21.         echo $EQUAL
22.         return
23.     else
24.         if [ "$1" -gt "$2" ]
25.         then
26.             retval=$1
27.         else
28.             retval=$2
29.         fi
30.     fi
31.
32.     echo $retval        # 输出 (到 stdout), 而没有用返回值。
33.                        # 为什么?
34. }
35.
36.
37. return_val=$(max2 33001 33997)
38. #          ^^^^^      函数名
39. #          ^^^^^^ ^^^^^ 传递进来的参数

```

```

40. # 这其实是命令替换的一种形式：
41. #+ 可以把函数看作一个命令，
42. #+ 然后把函数的stdout赋值给变量“return_val”。
43.
44.
45. # ===== OUTPUT =====
46. if [ "$return_val" -eq "$E_PARAM_ERR" ]
47. then
48.     echo "Error in parameters passed to comparison function!"
49. elif [ "$return_val" -eq "$EQUAL" ]
50. then
51.     echo "The two numbers are equal."
52. else
53.     echo "The larger of the two numbers is $return_val."
54. fi
55. # =====
56.
57. exit 0
58.
59. # 练习：
60. # -----
61. # 1) 找到一种更优雅的方法，
62. #+ 去测试传递给函数的参数。
63. # 2) 简化“输出”段的if/then结构。
64. # 3) 重写这个脚本，使其能够从命令行参数中获得输入。

```

这是另一个能够捕捉函数“返回值”的例子。要想搞明白这个例子，需要一些awk的知识。

```

1. month_length () # 把月份作为参数。
2. {               # 返回该月包含的天数。
3.     monthD="31 28 31 30 31 30 31 31 30 31 30 31" # 作为局部变量声明？
4.     echo "$monthD" | awk '{ print $'"${1}"' }' # 小技巧。
5. #                                     ^^^^^^^^^^
6. # 传递给函数的参数 ($1 -- 月份)，然后传给 awk。
7. # Awk 把参数解释为"print $1 . . . print $12" (这依赖于月份号)
8. # 这是一个模板，用于将参数传递给内嵌awk的脚本：

```



```

9. #                                     "${script_parameter}"
10. #     这里是一个简单的awk结构：
11. #     echo $monthD | awk -v month=$1 '{print $(month)}'
12. #     使用awk的-v选项，可以把一个变量值赋给
13. #+   awk程序块的执行体。
14. #     感谢 Rich.
15. #     需要做一些错误检查，来保证月份好正确，在范围（1-12）之间，
16. #+   别忘了检查闰年的二月。
17. }
18. # -----
19. # 用例：
20. month=4           # 以四月为例。
21. days_in=$(month_length $month)
22. echo $days_in    # 30
23. # -----

```

也请参考[例子 A-7](#) 和[例子A-37](#)。

练习：使用目前我们已经学到的知识，来扩展之前的例子 [将阿拉伯数字转化为罗马数字](#)，让它能够接受任意大的输入。

重定向

重定向函数的stdin

函数本质上其实就是一个[代码块](#)，这就意味着它的stdin可以被重定向（比如[例子3-1](#)）。

例子 24-11. 从username中取得用户的真名

```

1. #!/bin/bash
2. # realname.sh
3. #
4. # 依靠username，从/etc/passwd 中获得“真名”。
5.
6.
7. ARGCOUNT=1      # 需要一个参数。
8. E_WRONGARGS=85

```

```

9.
10. file=/etc/passwd
11. pattern=$1
12.
13. if [ $# -ne "$ARGCOUNT" ]
14. then
15.     echo "Usage: `basename $0` USERNAME"
16.     exit $E_WRONGARGS
17. fi
18.
19. file_excerpt ()      # 按照要求的模式来扫描文件,
20. {                    #+ 然后打印文件的相关部分。
21.     while read line # "while" 并不一定非得有 [ 条件 ] 不可。
22.     do
23.         echo "$line" | grep $1 | awk -F":" '{ print $5 }'
24.         # awk用":" 作为界定符。
25.     done
26. } <$file # 重定向到函数的stdin。
27.
28. file_excerpt $pattern
29. # 是的, 整个脚本其实可以被缩减为
30. #      grep PATTERN /etc/passwd | awk -F":" '{ print $5 }'
31. # or
32. #      awk -F: '/PATTERN/ {print $5}'
33. # or
34. #      awk -F: '($1 == "username") { print $5 }' # 从username中获取
    真名
35. # 但是, 这些起不到示例的作用。
36.
37. exit 0

```

还有一个办法, 或许能够更好的理解重定向函数的stdin。 它在函数内添加了一对大括号, 并且将重定向stdin的行为放在这对添加的大括号上。

1. # 用下面的方法来代替它:
2. `Function ()`

```

3. {
4.     ...
5. } < file
6.
7. # 试试这个:
8. Function ()
9. {
10.     {
11.         ...
12.     } < file
13. }
14.
15. # 同样的,
16.
17. Function () # 没问题.
18. {
19.     {
20.         echo $*
21.     } | tr a b
22. }
23.
24. Function () # 不行.
25. {
26.     echo $*
27. } | tr a b # 这儿的内嵌代码块是被强制的。
28. # 感谢, S.C.

```



Emmanuel Rouat的 `sample bash` 文件包含了一些很有指导性意义的函数例子。

24.2 局部变量

- 24.2 局部变量
 - 24.2.1 局部变量和递归
 - 注释

24.2 局部变量

怎样使一个变量变成“局部”变量？

局部变量

如果变量用`local`来声明，那么它就只能够在该变量被声明的[代码块](#)中可见。这个代码块就是[局部范围](#)。在一个函数中，一个局部变量只有在函数代码中才有意义。[\[1\]](#)

例子 24-12. 局部变量的可见范围

```

1.  #!/bin/bash
2.  # ex62.sh: 函数内部的局部变量与全局变量。
3.
4.  func () {
5.      local loc_var=23          # 声明为局部变量。
6.      echo                      # 使用 'local' 内建命令
7.      echo "\"loc_var\" in function = $loc_var"
8.      global_var=999            # 没有声明为局部变量。
9.      # 默认为全局变量。
10.
11.     echo "\"global_var\" in function = $global_var"
12. }
13.
14. func
15.
16. # 现在，来看看局部变量“loc_var”在函数外部是否可见。
17.
```

```

18. echo
19. echo "\"loc_var\" outside function = $loc_var"
20.                                     # $loc_var outside function =
21.                                     # 不行, $loc_var 不是全局可见的.
22. echo "\"global_var\" outside function = $global_var"
23.                                     # $在函数外部global_var = 999
24.                                     # $global_var 是全局可见的.
25. echo
26.
27. exit 0
28. # 与C语言相比, 在函数内声明的Bash变量
29. #+ 除非它被明确声明为local时, 它才是局部的。

```



在函数被调用之前, 所有在函数中声明的变量, 在函数外部都是不可见的, 当然也包括那些被明确声明为local的变量。

```

1. #!/bin/bash
2.
3. func ()
4. {
5.     global_var=37      # 变量只在函数体内可见
6.                        #+ 在函数被调用之前。
7. }                      # 函数结束
8.
9. echo "global_var = $global_var" # global_var =
10.                                # 函数 "func" 还没被调用,
11.                                #+ 所以$global_var 在这里还不是可见的.
12. func
13. echo "global_var = $global_var" # global_var = 37
14.                                # 已经在函数调用的时候设置。

```



正如Evgeniy Ivanov指出的那样, 当在一条命令中定义和给一个局部变量赋值时, 显然操作的顺序首先是给变量赋值, 之后限定变量的局部范围。这可以通过[返回值](#)来反应。

```

1.  #!/bin/bash
2.
3.  echo "=="OUTSIDE Function (global)=="
4.  t=$(exit 1)
5.  echo $?      # 1
6.              # 如预期一样。
7.
8.  echo
9.  function0 ()
10. {
11.     echo "=="INSIDE Function=="
12.     echo "Global"
13.     t0=$(exit 1)
14.     echo $?      # 1
15.                 # 如预期一样。
16.
17.     echo
18.     echo "Local declared & assigned in same command."
19.     local t1=$(exit 1)
20.     echo $?      # 0
21.                 # 意料之外!
22. # 显然, 变量赋值发生在Apparently,
23. #+ 局部声明之前。
24. #+ 返回值是为了latter。
25.
26.     echo
27.     echo "Local declared, then assigned (separate commands)."
28.     local t2
29.     t2=$(exit 1)
30.     echo $?
31. }
32.
33. function0

```

24.2.1 局部变量和递归

递归是一个有趣并且有时候非常有用的自己调用自己的形式。

Herbert Mayer 是这样定义递归的，“。。。表示一个算法通过使用一个简单的相同算法版本。。。 ”

想象一下，一个定义是从自身考虑的，[2] 一个表达包含了自身的表达，[3] 一条蛇吞下自己的尾巴，[4] 或者。。。 一个函数调用自身。[5]

例子 24-13. 一个简单的递归函数表示

```

1.  #!/bin/bash
2.  # recursion-demo.sh
3.  # 递归演示.
4.
5.  RECURSIONS=9    # 递归的次数.
6.  r_count=0        # 必须是全局变量, 为什么?
7.
8.  recurse ()
9.  {
10.     var="$1"
11.
12.     while [ "$var" -ge 0 ]
13.     do
14.         echo "Recursion count = "$r_count"  +-+  \ $var = "$var"
15.         (( var-- )); (( r_count++ ))
16.         recurse "$var" # 函数调用自身(递归)
17.     done              #+ 直到遇到什么样的终止条件?
18. }
19.
20. recurse $RECURSIONS
21.
22. exit $?
23. `
```

例子 24-14. 另一个简单的例子

```

1.  #!/bin/bash
```

```

2. # recursion-def.sh
3. # 另外一个描述递归的比较生动的脚本。
4.
5. RECURSIONS=10
6. r_count=0
7. sp=" "
8.
9. define_recursion ()
10. {
11.     ((r_count++))
12.     sp="$sp" " "
13.     echo -n "$sp"
14.     echo "\"The act of recurring ... \"" # Per 1913
        Webster's dictionary.
15.
16.     while [ $r_count -le $RECURSIONS ]
17.     do
18.         define_recursion
19.     done
20. }
21.
22. echo
23. echo "Recursion: "
24. define_recursion
25. echo
26.
27. exit $?

```

局部变量是一个写递归代码有效的工具，但是这种方法一般会包含大量的计算负载，显然在shell脚本中并不推荐递归。[\[6\]](#)

例子24-15. 使用局部变量进行递归

```

1. #!/bin/bash
2.
3. # 阶乘
4. # -----

```



```

5.
6.  # Bash允许递归么？
7.  # 恩，允许，但是...
8.  # 他太慢了，所以恐怕你难以忍受。
9.
10. MAX_ARG=5
11. E_WRONG_ARGS=85
12. E_RANGE_ERR=86
13.
14.
15. if [ -z "$1" ]
16. then
17.     echo "Usage: `basename $0` number"
18.     exit $E_WRONG_ARGS
19. fi
20.
21. if [ "$1" -gt $MAX_ARG ]
22. then
23.     echo "Out of range ($MAX_ARG is maximum)."

```

```

43.     return $factorial
44. }
45.
46. fact $1
47. echo "Factorial of $1 is $?."
48.
49. exit 0

```

也可以参考[例子 A-15](#)，一个包含递归例子的脚本。我们意识到递归同时也意味着巨大的资源消耗和缓慢的运行速度，因此它并不适合在脚本中使用。

注释

[\[1\]](#) 然而，如Thomas Braunberger 指出的那样，一个函数里定义的局部变量对于调用它的父函数也是可见的。

```

1.  #!/bin/bash
2.
3.  function1 ()
4.  {
5.      local func1var=20
6.
7.      echo "Within function1, \${func1var} = ${func1var}."
8.
9.      function2
10. }
11.
12. function2 ()
13. {
14.     echo "Within function2, \${func1var} = ${func1var}."
15. }
16.
17. function1
18.
19. exit 0

```

```

20.
21.
22. # 脚本的输出：
23.
24. # Within function1, $func1var = 20.
25. # Within function2, $func1var = 20.

```

在Bash手册里是这样描述的：

“局部变量只能在函数内部使用；它让变量名的可见范围限制在了函数内部以及它的孩子里”
[emphasis added]
 The ABS Guide的作者认为这个行为一个bug.

[2] 被熟知为冗余。

[3] 被熟知为同义反复。

[4] 被熟知为暗喻。

[5] 被熟知为递归函数。

[6] 太多的递归层次可能会引发一个脚本的段错误。

```

1. #!/bin/bash
2.
3. # 提醒：运行这个脚本可能会让你的系统卡死。
4. # 如果你够好运的话，在耗尽可用内存之前，它会发生一个段错误。
5.
6. recursive_function ()
7. {
8. echo "$1"      # 让函数做一些事情，加快发生段错误。
9. (( $1 < $2 )) && recursive_function $(( $1 + 1 )) $2;
10. # 只要第一个参数小于第二个参数，
11. #+ 让第一个参数加1，然后递归。
12. }
13.
14. recursive_function 1 50000 # 递归 50,000层！
15. # 很可能发生段错误(依赖于栈的大小，通过ulimit -m可以设置栈的大小)

```

```
16.  
17. # 即使是C语言，递归调用这么多层也会发生段错误，  
18. #+ 通过分配栈耗尽所有的内存。  
19.  
20.  
21. echo "This will probably not print."  
22. exit 0 # 这个脚本可能不会正常退出。  
23.  
24. # 感谢，Stéphane Chazelas.
```

24.3 不使用局部变量的递归

- 24.3 不使用局部变量的递归

24.3 不使用局部变量的递归

即使不适用局部变量，函数也可以递归的调用自身。

例子24-16. 斐波那契序列

```

1.  #!/bin/bash
2.  # fibo.sh : 斐波那契序列 (递归)
3.  # 作者: M. Cooper
4.  # License: GPL3
5.
6.  # -----算法-----
7.  # Fibo(0) = 0
8.  # Fibo(1) = 1
9.  # else
10. #   Fibo(j) = Fibo(j-1) + Fibo(j-2)
11. # -----
12.
13. MAXTERM=15      # 要产生的计算次数。
14. MINIDX=2        # 如果下标小于2, 那么 Fibo(idx) = idx.
15.
16. Fibonacci ()
17. {
18.     idx=$1      # 不需要是局部变量, 为什么?
19.     if [ "$idx" -lt "$MINIDX" ]
20.     then
21.         echo "$idx" # 前两个下标是0和1 ... 从上面的算法可以看出来。
22.     else
23.         (( --idx )) # j-1
24.         term1=$( Fibonacci $idx ) # Fibo(j-1)
25.         (( --idx )) # j-2
26.         term2=$( Fibonacci $idx ) # Fibo(j-2)

```

```

27.         echo $(( term1 + term2 ))
28.     fi
29.     # 一个丑陋的实现
30.     # C语言里, 一个更加优雅的斐波那契递归实现
31.     #+ 是一个简单的只需要7-10代码的算法翻译。
32. }
33.
34. for i in $(seq 0 $MAXTERM)
35. do # 计算 $MAXTERM+1 次。
36.     FIBO=$(Fibonacci $i)
37.     echo -n "$FIBO "
38. done
39. # 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
40. # 要花费一段时间, 不是么? 一个递归脚本是有些慢的。
41.
42. echo
43.
44. exit 0

```

例子 24-17. 汉诺塔

```

1.  #! /bin/bash
2.  #
3.  # 汉诺塔
4.  # Bash script
5.  # Copyright (C) 2000 Amit Singh. All Rights Reserved.
6.  # http://hanoi.kernelthread.com
7.  #
8.  # 在 Bash version 2.05b.0(13)-release下通过测试。
9.  # 同样在Bash3.x版本下工作正常。
10. #
11. # 在 "Advanced Bash Scripting Guide" 一书中使用
12. #+ 经过了脚本作者的许可。
13. # ABS的作者对脚本进行了轻微的修改和注释。
14. #=====#
15. # 汉诺塔是由Edouard Lucas,提出的数学谜题,
16. #+ 他是19世纪的法国数学家。
17. #

```

```

18. # 有三个直立的柱子竖在地面上。
19. # 第一个柱子上有一组盘子套在上面。
20. # 这些盘子是平的，中间有孔，
21. #+ 可以套在柱子上面。
22. # 这些盘子的直径不同，它们从下而上，
23. #+ 按照尺寸递减的顺序摆放。
24. # 也就是说，最小的在最上边，最大的在最下面。
25. #
26. # 现在的任务是要把这组盘子
27. #+ 从一个柱子上全部搬到另一个柱子上
28. # 你每次只能将一个盘子从一个柱子移到另一个柱子上。
29. # 你也可以把盘子从其他的柱子上移回到原来的柱子上。
30. # 你只能把小的盘子放到大的盘子上。
31. #+ 反过来就不行。
32. # 切记，绝对不能把大盘子放到小盘子的上面。
33. # 如果盘子的数量比较少，那么移不了几次就能完成。
34. #+ 但是随着盘子数量的增加，
35. #+ 移动次数几乎成倍的增长，
36. #+ 而且移动的“策略”也会变得越来越复杂。
37. #
38. # 想了解更多信息的话，请访问http://hanoi.kernelthread.com
39. #+ 或者 pp. 186-92 of _The Armchair Universe_ by A.K. Dewdney.
40. #
41. #
42. #          ...          ...          ...
43. #          | |          | |          | |
44. #         _|_|_         | |          | |
45. #        |_____|        | |          | |
46. #       |_____|        | |          | |
47. #      |_____|        | |          | |
48. #     |_____|        | |          | |
49. #    |          |        | |          | |
50. # .----- .
51. # |*****|
52. #          #1          #2          #3
53. #
54. #=====#

```

```

55. E_NOPARAM=66 # 没有参数传给脚本。
56. E_BADPARAM=67 # 传给脚本的盘子个数不符合要求。
57. Moves=      # 保存移动次数的全局变量。
58.             # 这里修改了原来的脚本。
59.
60. dohanoi() {  # 递归函数
61.     case $1 in
62.         0)
63.             ;;
64.         *)
65.             dohanoi "$(($1-1))" $2 $4 $3
66.             echo move $2 "-->" $3
67.             ((Moves++))      # 这里修改了原来的脚本。
68.             dohanoi "$(($1-1))" $4 $3 $2
69.             ;;
70.     esac
71. }
72.
73. case $# in
74.     1) case $((($1>0)) in      # 至少要有有一个盘子
75.         1) # Nested case statement.
76.             dohanoi $1 1 3 2
77.             echo "Total moves = $Moves"      # 2^n - 1, where n = #
of disks.
78.             exit 0;
79.             ;;
80.         *)
81.             echo "$0: illegal value for number of disks";
82.             exit $E_BADPARAM;
83.             ;;
84.     esac ;;
85.     *)
86.         echo "usage: $0 N"
87.         echo "      Where \"N\" is the number of disks."
88.         exit $E_NOPARAM;
89.         ;;
90.     esac
91.

```



```
92. # 练习：
93. # -----
94. # 1) 这个位置以下的代码会不会执行？
95. #     为什么不(容易)
96. # 2) 解释一下这个 "dohanoi" 函数的运行原理。
97. #     (比较难 可以参考上面的Dewdney 的引用)
```

25. 别名

- 25. 别名
 - 注意

25. 别名

Bash 别名 本质上不外乎是键盘上的快捷键，缩写呢是避免输入很长的命令串的一种手段。举个例子，在 `~/.bashrc` 文件中包含别名 `lm="ls -l | more`，而后每个命令行输入的 `lm` [1] 将会自动被替换成 `ls -l | more`。这可以节省大量的命令行输入和避免记住复杂的命令和选项。设定别名 `rm="rm -i"`（交互的删除模式）防止无意的删除重要文件，也许可以少些悲痛。

脚本中别名作用十分有限。如果别名可以有一些 C 预处理器的功能会更好，例如宏扩展，但不幸的是 bash 别名中没有扩展参数。[2] 另外，脚本在“复合结构”中并不能扩展自身的别名，例如 `if/then`，循环和函数。另一个限制是，别名不能递归扩展。基本上是我们无论怎么喜欢用别名都不如函数 `function` 来的更有效。

样例 25-1. 脚本中的别名

```
1. #!/bin/bash
2. # alias.sh
3.
4. shopt -s expand_aliases
5. # 必须设置此选项，否则脚本不能别名扩展。
6.
7.
8. # 首先来点好玩的东西。
9. alias Jesse_James='echo "\"Alias Jesse James\" was a 1959 comedy
    starring Bob Hope."'
10. Jesse_James
```

```
11.
12. echo; echo; echo;
13.
14. alias ll="ls -l"
15. # 可以任意使用单引号 (') 或双引号 (") 把别名括起来.
16.
17. echo "Trying aliased \"ll\":"
18. ll /usr/X11R6/bin/mk*    /* 别名可以运行.
19.
20. echo
21.
22. directory=/usr/X11R6/bin/
23. prefix=mk* # See if wild card causes problems.
24. echo "Variables \"directory\" + \"prefix\" = $directory$prefix"
25. echo
26.
27. alias lll="ls -l $directory$prefix"
28.
29. echo "Trying aliased \"lll\":"
30. lll          # 所有 /usr/X11R6/bin 文件清单以 mk 开始.
31. # 别名可以处理连续的变量 -- 包含 wild card -- o.k.
32.
33.
34.
35.
36. TRUE=1
37.
38. echo
39.
40. if [ TRUE ]
41. then
42.     alias rr="ls -l"
43.     echo "Trying aliased \"rr\" within if/then statement:"
44.     rr /usr/X11R6/bin/mk*    /* 结果报错!
45.     # 别名在复合的表达式中并没有生效.
46.     echo "However, previously expanded alias still recognized:"
47.     ll /usr/X11R6/bin/mk*
48. fi
```

```

49.
50. echo
51.
52. count=0
53. while [ $count -lt 3 ]
54. do
55.     alias rrr="ls -l"
56.     echo "Trying aliased \"rrr\" within \"while\" loop:"
57.     rrr /usr/X11R6/bin/mk*    /* 这里的别名也没生效.
58.                               # alias.sh: 行 57: rrr: 命令未找到
59.     let count+=1
60. done
61.
62. echo; echo
63.
64. alias xyz='cat $0'    # 列出了自身.
65.                       # 注意强引.
66. xyz
67. # 这看起来能工作,
68. #+ 尽管 bash 文档不介意这么做.
69. #
70. # 然而, Steve Jacobson 指出,
71. #+ "$0" 参数的扩展在上面的别名申明后立刻生效.
72.
73. exit 0

```

取消别名的命令删除之前设置的别名。

样例 25-2. unalias: 设置和取消一个别名

```

1. #!/bin/bash
2. # unalias.sh
3.
4. shopt -s expand_aliases # 开启别名扩展.
5.
6. alias llm='ls -al | more'
7. llm
8.

```

```
9. echo
10.
11. unalias llm          # 取消别名.
12. llm
13. # 'llm' 不再被识别后的报错信息.
14.
15. exit 0
16. bash$ ./unalias.sh
17. total 6
18. drwxrwxr-x    2 bozo    bozo    3072 Feb  6 14:04 .
19. drwxr-xr-x   40 bozo    bozo    2048 Feb  6 14:04 ..
20. -rwxr-xr-x    1 bozo    bozo     199 Feb  6 14:04 unalias.sh
21.
22. ./unalias.sh: llm: 命令未找到
```

注意

[1] ... 作为命令行的第一个词。显然别名只在命令的开始有意义。

[2] 然而，别名可用来扩展位置参数。

27 数组

- 27 数组

27 数组

新版本的Bash支持一维数组。 数组元素可以使用符号`variable[xx]`来初始化。另外，脚本可以使用`declare -a variable`语句来制定一个数组。 如果想引用一个数组元素（也就是取值），可以使用大括号，访问形式为 `${element[xx]}` 。

例子 27-1. 简单的数组使用

```

1.  #!/bin/bash
2.
3.  area[11]=23
4.  area[13]=37
5.  area[51]=UF0s
6.
7.  # 数组成员不一定非得是相邻或连续的。
8.
9.  # 数组的部分成员可以不被初始化。
10. # 数组中允许空缺元素。
11. # 实际上，保存着稀疏数据的数组（“稀疏数组”）
12. #+ 在电子表格处理软件中是非常有用的。
13.
14. echo -n "area[11] = "
15. echo ${area[11]}      # 需要{大括号}。
16.
17. echo -n "area[13] = "
18. echo ${area[13]}
19.
20. echo "Contents of area[51] are ${area[51]}."
21.
22. # 没被初始化的数组成员打印为空值（null变量）。

```

```

23. echo -n "area[43] = "
24. echo ${area[43]}
25. echo "(area[43] unassigned)"
26.
27. echo
28.
29. # 两个数组元素的和被赋值给另一个数组元素。
30. area[5]=`expr ${area[11]} + ${area[13]}`
31. echo "area[5] = area[11] + area[13]"
32. echo -n "area[5] = "
33. echo ${area[5]}
34.
35. area[6]=`expr ${area[11]} + ${area[51]}`
36. echo "area[6] = area[11] + area[51]"
37. echo -n "area[6] = "
38. echo ${area[6]}
39. # 这里会失败，是因为不允许整数与字符串相加。
40.
41. echo; echo; echo
42.
43. # -----
44. # 另一个数组，"area2".
45.
46. # 另一种给数组变量赋值的方法...
47. # array_name=( XXX YYY ZZZ ... )
48.
49. area2=( zero one two three four )
50.
51. echo -n "area2[0] = "
52. echo ${area2[0]}
53. # 啊哈，从0开始计算数组下标（也就是，数组的第一个元素为[0]，而不是[1]）。
54.
55. echo -n "area2[1] = "
56. echo ${area2[1]}      # [1] 是数组的第二个元素。
57. # -----
58.
59. echo; echo; echo
60.

```

```

61. # -----
62. # 第三个数组, "area3".
63. # 另外一种给数组元素赋值的方法...
64. # array_name=( [xx]=XXX [yy]=YYY ...)
65.
66. area3=( [17]=seventeen [24]=twenty-four )
67.
68. echo -n "area3[17] = "
69. echo ${area3[17]}
70.
71. echo -n "area3[24] = "
72. echo ${area3[24]}
73. # -----
74.
75. exit 0

```

我们可以看出, 初始化整数的一个简单的方法是 `array=(element1 element2 ... elementN)`。

1. `base64_charset=({A..Z} {a..z} {0..9} + / =)`
2. # 使用扩展的一对范围 Using extended brace expansion
3. `#+` 去初始化数组的元素。to initialize the elements of the array.
4. # 从 vladz's "base64.sh" 脚本中摘录过来。
5. `#+` 在"Contributed Scripts" 附录中可以看到。

Bash允许把变量当成数据来操作, 即使这个变量没有明确地被声明为数组。

1. `string=abcABC123ABCabc`
2. `echo ${string[@]}` # abcABC123ABCabc
3. `echo ${string[*]}` # abcABC123ABCabc
4. `echo ${string[0]}` # abcABC123ABCabc
5. `echo ${string[1]}` # 没有输出!
6. # 为什么?
7. `echo $#string[@]` # 1
8. # 数组中只有一个元素。
9. # 就是这个字符串本身。


```

10.
11. # 感谢你, Michael Zick, 指出这一点.

```

类似的示范可以参考 [Bash变量是无类型的](#) 。

例子 27-2. 格式化一首诗

```

1.  #!/bin/bash
2.  # poem.sh: 将本书作者非常喜欢的一首诗, 漂亮的打印出来。
3.
4.  # 诗的行数 (单节).
5.  Line[1]="I do not know which to prefer,"
6.  Line[2]="The beauty of inflections"
7.  Line[3]="Or the beauty of innuendoes,"
8.  Line[4]="The blackbird whistling"
9.  Line[5]="Or just after."
10. # 注意 引用允许嵌入的空格。
11.
12. # 出处.
13. Attrib[1]=" Wallace Stevens"
14. Attrib[2]="\"Thirteen Ways of Looking at a Blackbird\""
15. # 这首诗已经是公共版权了 (版权已经过期了).
16.
17. echo
18.
19. tput bold    # 粗体打印.
20.
21. for index in 1 2 3 4 5    # 5行.
22. do
23.     printf "        %s\n" "${Line[index]}"
24. done
25.
26. for index in 1 2          # 出处为2行。
27. do
28.     printf "        %s\n" "${Attrib[index]}"
29. done
30.
31. tput sgr0    # 重置终端。Reset terminal.

```

```

32.                # 查看 'tput' 文档.
33.  echo
34.
35.  exit 0
36.
37.  # 练习:
38.  # -----
39.  # 修改这个脚本, 使其能够从一个文本数据文件中提取出一首诗的内容, 然后将其漂亮的打
    印出来。

```

数组元素有它们独特的语法, 甚至标准Bash命令和操作符, 都有特殊的选项用以配合数组操作。

例子 27-3. 多种数组操作

```

1.  #!/bin/bash
2.  # array-ops.sh: 更多有趣的数组用法.
3.
4.  array=( zero one two three four five )
5.  # 数组元素 0    1    2    3    4    5
6.
7.  echo ${array[0]}      # 0
8.  echo ${array:0}       # 0
9.                        # 第一个元素的参数扩展,
10.                       #+ 从位置0(#0)开始 (即第一个字符) .
11.  echo ${array:1}       # ero
12.                       # 第一个元素的参数扩扎,
13.                       #+ 从位置1 (#1) 开始 (即第二个字符) 。
14.
15.  echo "-----"
16.
17.  echo ${#array[0]}      # 4
18.                       # 第一个数组元素的长度。
19.  echo ${#array}         #4
20.                       # 第一个数组元素的长度。
21.                       # (另一种表示形式)
22.

```

```

23. echo ${#array[1]}           # 3
24.                             # 第二个数组元素的长度。
25.                             # Bash中的数组是从0开始索引的。
26.
27. echo ${#array[*]}           # 6
28.                             # 数组中的元素个数。
29. echo ${#array[@]}           # 6
30.                             # 数组中的元素个数。
31. echo "-----"
32.
33. array2=( [0]="first element" [1]="second element" [3]="fourth
    element" )
34. #           ^      ^      ^      ^      ^      ^      ^      ^
    ^
35. # 引用允许嵌入的空格, 在每个单独的数组元素中。
36.
37. echo ${array2[0]}           # 第一个元素
38. echo ${array2[1]}           # 第二个元素
39. echo ${array2[2]}           #
40.                             # 因为并没有被初始化, 所以此值为null。
41. echo ${array2[3]}           # 第四个元素。
42. echo ${#array2[0]}           # 13      (第一个元素的长度)
43. echo ${#array2[*]}           # 3        (数组中元素的个数)
44.
45. exit

```

大部分标准字符串操作 都可以用于数组中。

例子27-4. 用于数组的字符串操作

```

1.  #!/bin/bash
2.  # array-strops.sh: 用于数组的字符串操作。
3.
4.  # 本脚本由Michael Zick 所编写。
5.  # 通过了授权在本书中使用。
6.  # 修复: 05 May 08, 04 Aug 08.
7.
8.  # 一般来说, 任何类似于 ${name ... } (这种形式) 的字符串操作

```

```

9.  #+ 都能够应用于数组中的所有字符串元素,
10. #+ 比如说${name[@] ... } 或者 ${name[*] ...} 这两种形式。
11.
12. arrayZ=( one two three four five five )
13.
14. echo
15.
16. # 提取尾部的子串。
17. echo ${arrayZ[@]:0}      # one two three four five five
18. #           ^          所有元素
19.
20. echo ${arrayZ[@]:1}      # two three four five five
21. #           ^          element[0]后边的所有元素。
22.
23. echo ${arrayZ[@]:1:2}    # two three
24. #           ^          只提取element[0]后边的两个元素。
25.
26. echo "-----"
27.
28.
29. # 子串删除
30.
31. # 从字符串的开头删除最短的匹配。
32.
33. echo ${arrayZ[@]#f*r}    # one two three five five
34. #           ^          # 匹配将应用于数组的所有元素。
35. #                               # 匹配到了"four", 并且将它删除。
36.
37. # 从字符串的开头删除最长的匹配
38. echo ${arrayZ[@]##t*e}   # one two four five five
39. #           ^^         # 匹配将应用于数组的所有元素
40. #                               # 匹配到了 "three" , 并且将它删除。
41.
42. # 从字符串的结尾删除最短的匹配
43. echo ${arrayZ[@]%h*e}    # one two t four five five
44. #           ^          # 匹配将应用于数组的所有元素
45. #                               # 匹配到了 "hree" , 并且将它删除。
46.

```

```

47. # 从字符串的结尾删除最长的匹配
48. echo ${arrayZ[@]%%t*e} # one two four five five
49. #                ^^      # 匹配将应用于数组的所有元素
50. #                # 匹配到了 "three" , 并且将它删除。
51.
52. echo "-----"
53.
54. # 子串替换
55.
56. # 第一个匹配到的子串将会被替换。
57. echo ${arrayZ[@]/fiv/XYZ} # one two three four XYZe XYZe
58. #                ^      # 匹配将应用于数组的所有元素
59.
60. # 所有匹配到的子串将会被替换。
61. echo ${arrayZ[@]//iv/YY} # one two three four fYYe fYYe
62. #                # 匹配将应用于数组的所有元素
63.
64. # 删除所有的匹配子串
65. # 如果没有指定替换字符串的话, 那就意味着'删除'...
66. echo ${arrayZ[@]//fi/} # one two three four ve ve
67. #                ^^      # 匹配将应用于数组的所有元素
68.
69. # 替换字符串前端子串
70. echo ${arrayZ[@]/#fi/XY} # one two three four XYve XYve
71. #                ^      # 匹配将应用于数组的所有元素
72.
73. # 替换字符串后端子串
74. echo ${arrayZ[@]/%ve/ZZ} # one two three four fiZZ fiZZ
75. #                ^      # 匹配将应用于数组的所有元素
76.
77. echo ${arrayZ[@]/%o/XX} # one twXX three four five five
78. #                ^      # 为什么?
79.
80. echo "-----"
81.
82. replacement() {
83.     echo -n "!!!"
84. }

```

```

85.
86. echo ${arrayZ[@]/%e/${replacement}}
87. #                ^  ^^^^^^^^^^^^^^^^^
88. # on!!! two thre!!! four fiv!!! fiv!!!
89. # replacement()的标准输出就是那个替代字符串。
90. # Q.E.D: 替换动作实际上是一个‘赋值’。
91.
92. echo "-----"
93.
94. # 使用"for-each"之前:
95. echo ${arrayZ[@]//*/${replacement optional_arguments}}
96. #                ^^ ^^^^^^^^^^^^^^^^^
97. # !!! !!! !!! !!! !!! !!!
98.
99. # 现在, 如果Bash只将匹配到的字符串
100. #+ 传递给被调用的函数...
101.
102. echo
103.
104. exit 0
105.
106. # 在将处理后的结果发送到大工具之前, 比如-- Perl, Python, 或者其它工具
107. # 回忆一下:
108. #   $( ... ) 是命令替换。
109. #   一个函数作为子进程运行。
110. #   一个函数将结果输出到stdout。
111. #   赋值, 结合"echo"和命令替换,
112. #+ 可以读取函数的stdout。
113. #   使用name[@]表示法指定了一个 "for-each"
114. #+ 操作。
115. #   Bash比你想象的更加强力。

```

命令替换 可以构造数组的独立元素。

例子 27-5. 将脚本中的内容赋值给数组

```

1. #!/bin/bash
2. # script-array.sh: 将脚本中的内容赋值给数组。

```

```

3. # 这个脚本的灵感来自于 Chris Martii 的邮件 (感谢!).
4.
5. script_contents=( $(cat "$0") ) # 将这个脚本的内容($0)
6.                                #+ 赋值给数组
7. for element in $(seq 0 ${#script_contents[@]} - 1))
8. do                               # ${#script_contents[@]}
9.                                #+ 表示数组元素的个数
10.                               #
11.                               # 问题:
12.                               # 为什么必须使用seq 0 ?
13.                               # 用seq 1来试一下.
14.     echo -n "${script_contents[$element]}"
15.                                # 在同一行上显示脚本中每个域的内容。
16. # echo -n "${script_contents[element]}" also works because of ${
17. ... }.
18.     echo -n " -- " # 使用 " -- " 作为域分隔符。
19. done
20.
21. exit 0
22. # 练习:
23. # -----
24. # 修改这个脚本,
25. #+ 让这个脚本能够按照它原本的格式输出,
26. #+ 连同空格, 换行, 等等。

```

在数组环境中, 某些Bash [内建命令](#) 的含义可能会有些轻微的改变。比如, [unset](#) 命令可以删除数组元素, 甚至能够删除整个数组。

例子 27-6. 一些数组的专有特性

```

1. #!/bin/bash
2.
3. declare -a colors
4. # 脚本中所有的后续命令都会把
5. #+ "colors" 当做数组
6.

```

```

7.  echo "Enter your favorite colors (separated from each other by a
    space)."
8.
9.  read -a colors      # 至少需要键入3种颜色，以便于后边的演示。
10. # 'read'命令的特殊选项，
11. #+ 允许给数组元素赋值。
12.
13. echo
14.
15. element_count=${#colors[@]}
16. # 提取数组元素个数的特殊语法
17. #     用element_count=${#colors[*]} 也可以。
18. #
19. #  "@" 变量允许在引用中存在单次分割，
20. #+ (依靠空白字符来分割变量)。
21. #
22. # 这就好像"$@" 和 "$*"
23. #+ 在位置参数中所表现出来的行为一样。
24.
25. index=0
26.
27. while [ "$index" -lt "$element_count" ]
28. do    # 列出数组中的所有元素
29.     echo ${colors[$index]}
30.     #   ${colors[index]} 也可以工作，因为它${ ... }之中。
31.     let "index = $index + 1"
32.     # Or:
33.     #   ((index++))
34. done
35. # 每个数组元素被列为单独的一行
36. # 如果没有这种要求的话，可以使用echo -n "${colors[$index]} "
37. #
38. # 也可以使用“for”循环来做：
39. #   for i in "${colors[@]}"
40. #   do
41. #       echo "$i"
42. #   done
43. # (Thanks, S.C.)

```



```

44.
45. echo
46.
47. # 再次列出数组中的所有元素，不过这次的做法更为优雅。
48.   echo ${colors[@]}           # echo ${colors[*]} 也可以工作。
49.
50. echo
51.
52. # "unset"命令既可以删除数组数据，也可以删除整个数组。
53. unset colors[1]              # 删除数组的第2个元素。
54.                               # 作用等同于colors[1]=
55. echo  ${colors[@]}           # 再次列出数组内容，第2个元素没了。
56.
57. unset colors                  # 删除整个数组。
58.                               # unset colors[*] 以及
59.                               #+ unset colors[@] 都可以。
60. echo; echo -n "Colors gone."
61. echo ${colors[@]}            # 再次列出数组内容，内容为空。
62. exit 0

```

正如我们在前面的例子中所看到的，`${array_name[@]}` 或者 `${array_name[*]}` 都与数组中的所有元素相关。同样的，为了计算数组的元素个数，可以使用 `${array_name[@]}` 或者 `${array_name[*]}`。`${#array_name}` 是数组第一个元素的长度，也就是 `${array_name[0]}` 的长度（字符个数）。

例子 27-7. 空数组与包含空元素的数组

```

1.  #!/bin/bash
2.  # empty-array.sh
3.
4.  # 感谢Stephane Chazelas制作这个例子的原始版本。
5.  #+ 同时感谢Michael Zick 和 Omair Eshkenazi 对这个例子所作的扩展。
6.  # 以及感谢Nathan Coulter 作的声明和感谢。
7.
8.  # 空数组与包含有空元素的数组，这两个概念不同。

```

```

9.
10. array0=( first second third )
11. array1=( ' ' )           # "array1" 包含一个空元素.
12. array2=( )               # 没有元素. . . "array2"为空
13. array3=( )               # 这个数组呢?
14.
15. echo
16. ListArray()
17. {
18.     echo
19.     echo "Elements in array0:  ${array0[@]}"
20.     echo "Elements in array1:  ${array1[@]}"
21.     echo "Elements in array2:  ${array2[@]}"
22.     echo "Elements in array3:  ${array3[@]}"
23.     echo
24.     echo "Length of first element in array0 = ${#array0}"
25.     echo "Length of first element in array1 = ${#array1}"
26.     echo "Length of first element in array2 = ${#array2}"
27.     echo "Length of first element in array3 = ${#array3}"
28.     echo
29.     echo "Number of elements in array0 = ${#array0[*]}" # 3
30.     echo "Number of elements in array1 = ${#array1[*]}" # 1
    (Surprise!)
31.     echo "Number of elements in array2 = ${#array2[*]}" # 0
32.     echo "Number of elements in array3 = ${#array3[*]}" # 0
33. }
34.
35. #
    =====
36.
37. ListArray
38.
39. # 尝试扩展这些数组。
40.
41. # 添加一个元素到这个数组。
42. array0=( "${array0[@]}" "new1" )
43. array1=( "${array1[@]}" "new1" )
44. array2=( "${array2[@]}" "new1" )

```

```

45. array3=( "${array3[@]}" "new1" )
46.
47. ListArray
48.
49. # 或者
50. array0[${#array0[*]}]="new2"
51. array1[${#array1[*]}]="new2"
52. array2[${#array2[*]}]="new2"
53. array3[${#array3[*]}]="new2"
54.
55. ListArray
56.
57. # 如果你按照上边的方法对数组进行扩展的话，数组比较像‘栈’
58. # 上边的操作就是‘压栈’
59. # ‘栈’的高度为：
60. height=${#array2[@]}
61. echo
62. echo "Stack height for array2 = $height"
63.
64. # ‘出栈’就是：
65. unset array2[${#array2[@]}-1]    # 数组从0开始索引
66. height=${#array2[@]}             #+ 这就意味着数组的第一个下标是0
67. echo
68. echo "POP"
69. echo "New stack height for array2 = $height"
70.
71. ListArray
72.
73. # 只列出数组array0的第二个和第三个元素。
74. from=1                          # 从0开始索引。
75. to=2
76. array3=( ${array0[@]:1:2} )
77. echo
78. echo "Elements in array3: ${array3[@]}"
79.
80. # 处理方式就像是字符串（字符数组）。
81. # 试试其他的“字符串”形式。
82.

```

```

83. # 替换：
84. array4=( ${array0[@]/second/2nd} )
85. echo
86. echo "Elements in array4: ${array4[@]}"
87.
88. # 替换掉所有匹配通配符的字符串
89. array5=( ${array0[@]//new?/old} )
90. echo
91. echo "Elements in array5: ${array5[@]}"
92.
93. # 当你觉得对此有把握的时候...
94. array6=( ${array0[@]#*new} )
95. echo # This one might surprise you.
96. echo "Elements in array6: ${array6[@]}"
97.
98. array7=( ${array0[@]#new1} )
99. echo # 数组array6之后就没有惊奇了。
100. echo "Elements in array7: ${array7[@]}"
101.
102. # 看起来非常像...
103. array8=( ${array0[@]/new1/} )
104. echo
105. echo "Elements in array8: ${array8[@]}"
106.
107. # 所以，让我们怎么形容呢？
108.
109. # 对数组var[@]中的每个元素The string operations are performed on
110. #+ 进行连续的字符串操作。each of the elements in var[@] in succession.
111. # 因此：Bash支持支持字符串向量操作，
112. # 如果结果是长度为0的字符串
113. #+ 元素会在结果赋值中消失不见。
114. # 然而，如果扩展在引用中，那个空元素会仍然存在。
115.
116. # Michael Zick: 问题--这些字符串是强引用还是弱引用？
117. # Nathan Coulter: 没有像弱引用的东西
118. #! 真正发生的事情是
119. #!+ 匹配的格式发生在
120. #!+ [word]的所有其它扩展之后

```

```

121.  #!+  比如像${parameter#word}.
122.
123.  zap='new*'
124.  array9=( ${array0[@]/$zap/} )
125.  echo
126.  echo "Number of elements in array9: ${#array9[@]}"
127.  array9=( "${array0[@]/$zap/}" )
128.  echo "Elements in array9: ${array9[@]}"
129.  # 此时, 空元素仍然存在
130.  echo "Number of elements in array9: ${#array9[@]}"
131.
132.  # 当你还在认为你身在Kansas州时...
133.  array10=( ${array0[@]#$zap} )
134.  echo
135.  echo "Elements in array10: ${array10[@]}"
136.  # 但是, 如果被引用的话, *号将不会被解释。
137.  array10=( ${array0[@]#"$zap"} )
138.  echo
139.  echo "Elements in array10: ${array10[@]}"
140.  # 可能, 我们仍然在Kansas...
141.  # (上面的代码块Nathan Coulter所修改.)
142.
143.  # 比较 array7 和array10.
144.  # 比较array8 和array9.
145.
146.  # 重申: 所有所谓弱引用的东西
147.  # Nathan Coulter 这样解释:
148.  # word在${parameter#word}中的匹配格式在
149.  #+ 参数扩展之后和引用移除之前已经完成了。
150.  # 在通常情况下, 格式匹配在引用移除之后完成。
151.
152.  exit

```


`${array_name[@]}` 和 `${array_name[*]}` 的关系非常类似于 `$@` 和 `$*`。这种数组用法非常广泛。

1. # 复制一个数组

```

2. array2=( "${array1[@]}" )
3. # 或者
4. array2="${array1[@]}"
5. #
6. # 然而，如果在“缺项”数组中使用的话，将会失败
7. #+ 也就是说数组中存在空洞（中间的某个元素没赋值），
8. #+ 这个问题由Jochen DeSmet 指出。
9. # -----
10. array1[0]=0
11. # array1[1] not assigned
12. array1[2]=2
13. array2=( "${array1[@]}" )      # 拷贝它？
14. echo ${array2[0]}           # 0
15. echo ${array2[2]}           # (null)，应该是 2
16. # -----
17. # 添加一个元素到数组。
18. array=( "${array[@]}" "new element" )
19. # 或者
20. array[${#array[*]}]="new element"
21. # 感谢，S.C.

```

 **array=(element1 element2 ... elementN)** 初始化操作，如果有**命令替换**的帮助，就可以将一个文本文件的内容加载到数组。

```

1. #!/bin/bash
2. filename=sample_file
3. #          cat sample_file
4. #
5. #          1  a  b  c
6. #          2  d  e  fg
7.
8. declare -a array1
9.
10. array1=( `cat "$filename"` )      # 将$filename的内容
11. #          把文件内容展示到输出    #+ 加载到数组array1.
12. #
13. # array1=( `cat "$filename" | tr '\n' ' '` )

```

```

14. #                                把文件中的换行替换为空格
15. # 其实这样做是没必要的，因为Bash在做单词分割的时候，
16. #+ 将会把换行转换为空格。
17.
18. echo ${array1[@]}                # 打印数组
19. #                                1 a b c 2 d e fg
20. #
21. # 文件中每个被空白符分割的“单词”
22. #+ 都被保存到数组的一个元素中。
23.
24. element_count=${#array1[*]}
25. echo $element_count              # 8

```

出色的技巧使得数组的操作技术又多了一种。

例子 27-8. 初始化数组

```

1.  #! /bin/bash
2.  # array-assign.bash
3.
4.  # 数组操作是Bash所特有的，
5.  #+ 所以才使用".bash" 作为脚本扩展名
6.
7.  # Copyright (c) Michael S. Zick, 2003, All rights reserved.
8.  # License: Unrestricted reuse in any form, for any purpose.
9.  # Version: $ID$
10. #
11. # 说明与注释由 William Park所添加.
12.
13. # 基于 Stephane Chazelas所提供的例子
14. #+ 它是在ABS中的较早版本。
15.
16. # 'times' 命令的输出格式：
17. # User CPU <space> System CPU
18. # User CPU of dead children <space> System CPU of dead children
19.
20. # Bash有两种方法，
21. #+ 可以将一个数组的所有元素都赋值给一个新的数组变量。

```

```

22. # 这两个方法都会丢弃数组中的“空引用”（null值）元素
23. #+ 在2.04和以后的Bash版本中。
24. # 另一种给数组赋值的方法将会被添加到新版本的Bash中，
25. #+ 这种方法采用[subscript]=value 形式，来维护数组下标与元素值之间的关系。
26.
27. # 可以使用内部命令来构造一个大数组，
28. #+ 当然，构造一个包含上千元素数组的其它方法
29. #+ 也能很好的完成任务
30.
31. declare -a bigOne=( /dev/* ) # /dev下的所有文件 . . .
32. echo
33. echo 'Conditions: Unquoted, default IFS, All-Elements-Of'
34. echo "Number of elements in array is ${#bigOne[@]}"
35.
36. # set -vx
37.
38. echo
39. echo '- - testing: =( ${array[@]} ) - -'
40. times
41. declare -a bigTwo=( ${bigOne[@]} )
42. # 注意括号：      ^          ^
43. times
44. echo
45.
46. echo '- - testing: =${array[@]} - -'
47. times
48. declare -a bigThree=${bigOne[@]}
49. # 这次没用括号。
50. times
51. # 通过比较，可以发现第二种格式的赋值更快一些，
52. #+ 正如 Stephane Chazelas指出的那样
53. #
54. # William Park 解释：
55. #+ bigTwo数组是作为一个单个字符串被赋值的(因为括号)
56. #+ 而BigThree数组，则是一个元素一个元素进行的赋值。
57. # 所以，实质上是：
58. #                               bigTwo=( [0]="..." [1]="..." [2]="..." ... )
59. #                               bigThree=( [0]="... .." )

```



```

60. #
61. # 通过这样确认:  echo ${bigTwo[0]}
62. #                      echo ${bigThree[0]}
63. # 在本书的例子中, 我还是会继续使用第一种形式,
64. #+ 因为, 我认为这种形式更有利于将问题阐述清楚。
65.
66. # 在我所使用的例子中, 在其中复用的部分,
67. #+ 还是使用了第二种形式, 那是因为这种形式更快。
68.
69. # MSZ: 很抱歉早先的疏忽。
70.
71. # 注意:
72. # ----
73. # 32和44的"declare -a" 语句其实不是必需的,
74. #+ 因为Array=(...)形式
75. #+ 只能用于数组
76. # 然而, 如果省略这些声明的话,
77. #+ 会导致脚本后边的相关操作变慢。
78. # 试试看, 会发生什么。
79.
80. exit 0

```



在数组声明的时候添加一个额外的**declare -a**语句, 能够加速后续的数组操作速度。

例子 27-9. 拷贝和连接数组

```

1.  #! /bin/bash
2.  # CopyArray.sh
3.  #
4.  # 这个脚本由Michael Zick所编写。
5.  # 这里已经通过作者的授权
6.
7.  # 如何“通过名字传值&通过名字返回”
8.  #+ 或者“建立自己的赋值语句”。
9.
10. CpArray_Mac() {

```

```

11.      # 建立赋值命令
12.      echo -n 'eval '
13.      echo -n "$2"                # 目的名字
14.      echo -n '=( ${'
15.      echo -n "$1"                # 源名字
16.      echo -n '[@]} )'
17.
18. # 上边这些语句会构成一条命令。
19. # 这仅仅是形式上的问题。
20. }
21.
22. declare -f CopyArray
23. CopyArray=CpArray_Mac
24.
25. Hype() {
26. # "Pointer"函数
27. # 状态产生器
28. # 需要连接的数组名为$1.
29. # (把这个数组与字符串"Really Rocks"结合起来, 形成一个新数组.)
30. # 并将结果从数组$2中返回.
31.
32.     local -a TMP
33.     local -a hype=( Really Rocks )
34.     ${CopyArray $1 TMP}
35.     TMP=( ${TMP[@]} ${hype[@]} )
36.     ${CopyArray TMP $2}
37. }
38.
39. declare -a before=( Advanced Bash Scripting )
40. declare -a after
41.
42. echo "Array Before = ${before[@]}"
43.
44. Hype before after
45.
46. echo "Array After = ${after[@]}"
47.
48. # 连接的太多了?

```

```

49.
50. echo "What ${after[@]:3:2}?"
51. declare -a modest=( ${after[@]:2:1} ${after[@]:3:2} )
52. #          ----- 子串提取 -----
53.
54. echo "Array Modest = ${modest[@]}"
55.
56. # 'before' 发生了什么变化 ?
57.
58. echo "Array Before = ${before[@]}"
59.
60. exit 0

```

例子27-10. 关于串联数组的更多信息

```

1.  #! /bin/bash
2.  # array-append.bash
3.
4.  # Copyright (c) Michael S. Zick, 2003, All rights reserved.
5.  # License: Unrestricted reuse in any form, for any purpose.
6.  # Version: $ID$
7.  #
8.  # 在格式上, 由M.C做了一些修改.
9.
10. # 数组操作是Bash特有的属性。
11. # 传统的UNIX /bin/sh 缺乏类似的功能。
12.
13. # 将这个脚本的输出通过管道传递给'more',
14. #+ 这样做的目的是防止输出的内容超过终端能够显示的范围,
15. # 或者, 重定向输出到文件中。
16.
17. declare -a array1=( zero1 one1 two1 )
18. # 依次使用下标
19. declare -a array2=( [0]=zero2 [2]=two2 [3]=three2 )
20. # 数组中存在空缺的元素-- [1] 未定义
21.
22. echo
23. echo '- Confirm that the array is really subscript sparse. -'

```

```

24. echo "Number of elements: 4"           # 为了演示, 这里作了硬编码
25. for (( i = 0 ; i < 4 ; i++ ))
26. do
27.     echo "Element [$i]: ${array2[$i]}"
28. done
29. # 也可以参考一个更通用的例子,  basics-reviewed.bash.
30.
31.
32. declare -a dest
33.
34. # 将两个数组合并到第3个数组中。
35. echo
36. echo 'Conditions: Unquoted, default IFS, All-Elements-Of operator'
37. echo '- Undefined elements not present, subscripts not maintained.
    -'
38. # # 那些未定义的元素不会出现; 组合时会丢弃这些元素。
39.
40. dest=( ${array1[@]} ${array2[@]} )
41. # dest=${array1[@]}${array2[@]}           # 奇怪的结果, 可能是个bug。
42.
43. # 现在, 打印结果。
44. echo
45. echo '- - Testing Array Append - -'
46. cnt=${#dest[@]}
47.
48. echo "Number of elements: $cnt"
49. for (( i = 0 ; i < cnt ; i++ ))
50. do
51.     echo "Element [$i]: ${dest[$i]}"
52. done
53.
54. # 将数组赋值给一个数组中的元素 (两次)
55. dest[0]=${array1[@]}
56. dest[1]=${array2[@]}
57.
58. # 打印结果
59. echo
60. echo '- - Testing modified array - -'

```

```

61. cnt=${#dest[@]}
62.
63. echo "Number of elements: $cnt"
64. for (( i = 0 ; i < cnt ; i++ ))
65. do
66. echo "Element [$i]: ${dest[$i]}"
67. done
68.
69. # 检查第二个元素的修改状况.
70. echo
71. echo '- - Reassign and list second element - -'
72.
73. declare -a subArray=${dest[1]}
74. cnt=${#subArray[@]}
75.
76. echo "Number of elements: $cnt"
77. for (( i = 0 ; i < cnt ; i++ ))
78. do
79.     echo "Element [$i]: ${subArray[$i]}"
80. done
81.
82. # 如果你使用 '${ ... }'形式
83. ## 将一个数组赋值到另一个数组的一个元素中,
84. ## 那么这个数组的所有元素都会被转换为一个字符串,
85. ## 这个字符串中的每个数组元素都以空格进行分隔(其实是IFS的第一个字符).
86.
87. # 如果原来数组中的所有元素都不包含空白符 . . .
88. # 如果原来的数组下标都是连续的 . . .
89. # 那么我们就可以将原来的数组进行恢复.
90.
91. # 从修改过的第二个元素中, 将原来的数组恢复出来.
92. echo
93. echo '- - Listing restored element - -'
94.
95. declare -a subArray=( ${dest[1]} )
96. cnt=${#subArray[@]}
97.
98. echo "Number of elements: $cnt"

```

```

99. for (( i = 0 ; i < cnt ; i++ ))
100. do
101.     echo "Element [$i]: ${subArray[$i]}"
102. done
103.
104. echo '- - Do not depend on this behavior. - -'
105. echo '- - This behavior is subject to change - -'
106. echo '- - in versions of Bash newer than version 2.05b - -'
107.
108. # MSZ: 抱歉，之前混淆了一些要点。
109.
110. exit 0

```

有了数组，我们就可以在脚本中实现一些比较熟悉的算法。这么做，到底是不是一个好主意，我们在这里不做讨论，还是留给读者决定吧。

例子 27-11. 冒泡排序

```

1.  #!/bin/bash
2.  # bubble.sh: 一种排序方式，冒泡排序。
3.
4.  # 回忆一下冒泡排序的算法。我们在这里要实现它...
5.
6.  # 依靠连续的比较数组元素进行排序，
7.  #+ 比较两个相邻元素，如果顺序不对，就交换这两个元素的位置。
8.  # 当第一轮比较结束之后，最"重"的元素就会被移动到底部。
9.  # 当第二轮比较结束之后，第二"重"的元素就会被移动到次底部的位置。
10. # 依此类推。
11. # 这意味着每轮比较不需要比较之前已经"沉淀"好的数据。
12. # 因此你会注意到后边的数据在打印的时候会快一些。
13.
14.
15. exchange() {
16.     # 交换数组中的两个元素。
17.     local temp=${Countries[$1]} # 临时保存
18.                                   #+ 要交换的那个元素
19.     Countries[$1]=${Countries[$2]}

```

```

20.     Countries[$2]=$temp
21.
22.     return
23. }
24.
25. declare -a Countries # 声明数组,
26.                        #+ 此处是可选的, 因为数组在下面被初始化
27. # 我们是否可以使用转义符(\)
28. #+ 来将数组元素的值放在不同的行上?
29. # 可以.
30.
31. Countries=(Netherlands Ukraine Zaire Turkey Russia Yemen Syria \
32. Brazil Argentina Nicaragua Japan Mexico Venezuela Greece England \
33. Israel Peru Canada Oman Denmark Wales France Kenya \
34. Xanadu Qatar Liechtenstein Hungary)
35.
36. # "Xanadu" 虚拟出来的世外桃源.
37. #+ Kubla Khan做了个愉快的决定
38.
39.
40. clear # 开始之前的清屏动作
41.
42. echo "0: ${Countries[*]}" # 从索引0开始列出整个数组.
43.
44. number_of_elements=${#Countries[@]}
45. let "comparisons = $number_of_elements - 1"
46.
47. count=1 # Pass number.
48.
49. while [ "$comparisons" -gt 0 ] # 开始外部循环
50. do
51.
52.     index=0 # 在每轮循环开始之前, 重置索引。
53.
54.     while [ "$index" -lt "$comparisons" ] # 开始内部循环。
55.     do
56.         if [ ${Countries[$index]} \> ${Countries[`expr $index + 1`]} ]
57.         # 如果原来的排序次序不对...

```

```

58.      # 回想一下，在单括号中，
59.      #+ \>是ASCII码的比较操作符.
60.
61.      # if [[ ${Countries[$index]} > ${Countries[`expr $index + 1`]}
    ]]
62.      #+ 这样也行.
63.      then
64.          exchange $index `expr $index + 1` # 交换
65.      fi
66.      let "index += 1" #或者，index+=1 在Bash 3.1之后的版本才能这么用.
67.  done # 内部循环结束
68.
69.      # -----
    -----
70. # Paulo Marcel Coelho Aragao 建议我们可以使用更简单的for循环
71. #
72. # for (( last = $number_of_elements - 1 ; last > 0 ; last-- ))
73. ##              Fix by C.Y. Hunt              ^   (Thanks!)
74. # do
75. #     for (( i = 0 ; i < last ; i++ ))
76. #     do
77. #         [[ "${Countries[$i]}" > "${Countries[$((i+1))]}" ]] \
78. #         && exchange $i $((i+1))
79. #     done
80. # done
81. # -----
    -----
82.
83.
84. let "comparisons -= 1" # 因为最"重"的元素到了底部，
85.      #+ 所以每轮我们可以少做一次比较。
86.
87. echo
88. echo "$count: ${Countries[@]}" # 每轮结束后，都打印一次数组。
89. echo
90. let "count += 1"          # 增加传递计数。
91.
92. done                    # 外部循环结束

```



```

93.                                     # 至此，全部完成。
94. exit 0

```

我们可以在数组中嵌套数组么？

```

1.  #!/bin/bash
2.  # "嵌套" 数组。
3.
4.  # Michael Zick 提供了这个用例。
5.  #+ William Park做了一些修正和说明。
6.
7.  AnArray=( $(ls --inode --ignore-backups --almost-all \
8.             --directory --full-time --color=none --time=status \
9.             --sort=time -l ${PWD} ) ) # Commands and options.
10.
11. # 空格是有意义的 . . . 并且不要在上边用引号引用任何东西。
12.
13. SubArray=( ${AnArray[@]:11:1} ${AnArray[@]:6:5} )
14. # 这个数组有六个元素：
15. #+ SubArray=( [0]=${AnArray[11]} [1]=${AnArray[6]}
16.             [2]=${AnArray[7]}
17.             [3]=${AnArray[8]} [4]=${AnArray[9]} [5]=${AnArray[10]} )
18. #
19. # Bash数组是字符串(char *)类型
20. # 的(循环)链表
21. # 因此，这不是真正意义上的嵌套数组，
22. # 只不过功能很相似而已。
23.
24. echo "Current directory and date of last status change:"
25. echo "${SubArray[@]}"
26. exit 0

```

如果将“嵌套数组”与[间接引用](#) 组合起来使用的话，将会产生一些非常有趣的用法。

例子 27-12. 嵌套数组与间接引用

```

1.  #!/bin/bash
2.  # embedded-arrays.sh
3.  # 嵌套数组和间接引用.
4.
5.  # 本脚本由Dennis Leeuw 编写.
6.  # 经过授权, 在本书中使用.
7.  # 本书作者做了少许修改.
8.
9.  ARRAY1=(
10.      VAR1_1=value11
11.      VAR1_2=value12
12.      VAR1_3=value13
13.  )
14.
15.  ARRAY2=(
16.      VARIABLE="test"
17.      STRING="VAR1=value1 VAR2=value2 VAR3=value3"
18.      ARRAY21=${ARRAY1[*]}
19.  )    # 将ARRAY1嵌套到这个数组中.
20.
21.  function print () {
22.      OLD_IFS="$IFS"
23.      IFS=$'\n'          # 这么做是为了每行
24.                          #+ 只打印一个数组元素.
25.      TEST1="ARRAY2[*]"
26.      local ${!TEST1} # 删除这一行, 看看会发生什么?
27.      # 间接引用.
28.      # 这使得$TEST1
29.      #+ 只能够在函数内被访问.
30.
31.      # 让我们看看还能干点什么.
32.
33.      echo
34.      echo "\$TEST1 = $TEST1"          # 仅仅是变量名字.
35.      echo; echo
36.      echo "${!TEST1} = ${!TEST1}"    # 变量内容.

```

```

37.                                     # 这就是
38.                                     #+ 间接引用的作用。
39.     echo
40.     echo "-----"; echo
41.     echo
42.
43.     # 打印变量
44.     echo "Variable VARIABLE: $VARIABLE"
45.
46.     # 打印一个字符串元素
47.     IFS="$OLD_IFS"
48.     TEST2="STRING[*]"
49.     local ${!TEST2}      # 间接引用(同上)。
50.     echo "String element VAR2: $VAR2 from STRING"
51.
52.     # Print an array element
53.     TEST2="ARRAY21[*]"
54.     local ${!TEST2}      # 间接引用(同上)。
55.     echo "Array element VAR1_1: $VAR1_1 from ARRAY21"
56. }
57.
58. print
59. echo
60.
61. exit 0
62.
63. # 脚本作者注,
64. #+ "你可以很容易的将其扩展成一个能创建hash 的Bash 脚本。"
65. # (难) 留给读者的练习: 实现它。

```

数组使得埃拉托色尼素数筛子有了shell版本的实现。当然，如果你需要的是追求效率的应用，那么就 应该使用编译行语言来实现，比如C语言。因为脚本运行的太慢了。

例子 27-13. 埃拉托色尼素数筛子

```

1.  #!/bin/bash
2.  # sieve.sh (ex68.sh)
3.
4.  # 埃拉托色尼素数筛子
5.  # 找素数的经典算法.
6.
7.  # 在同等数值的范围内,
8.  #+ 这个脚本运行的速度比C版本慢的多.
9.
10. LOWER_LIMIT=1      # 从1开始.
11. UPPER_LIMIT=1000    # 到1000.
12. # (如果你时间很多的话 . . . 你可以将这个数值调的很高.)
13.
14. PRIME=1
15. NON_PRIME=0
16.
17. let SPLIT=UPPER_LIMIT/2
18. # 优化:
19. # 只需要测试中间到最大的值, 为什么?
20.
21. declare -a Primes
22. # Primes[] 是个数组.
23.
24.
25. initialize ()
26. {
27.     # 初始化数组.
28.     i=LOWER_LIMIT
29.     until [ "$i" -gt "$UPPER_LIMIT" ]
30.     do
31.         Primes[i]=$PRIME
32.         let "i += 1"
33.     done
34.     # 假定所有数组成员都是需要检查的(素数)
35.     #+ 直到检查完成.
36. }
37.
38. print_primes ()

```

```

39. {
40.     # 打印出所有数组Primes[]中被标记为素数的元素.
41.
42.     i=$LOWER_LIMIT
43.
44.     until [ "$i" -gt "$UPPER_LIMIT" ]
45.     do
46.         if [ "${Primes[i]}" -eq "$PRIME" ]
47.         then
48.             printf "%8d" $i
49.             # 每个数字打印前先打印8个空格, 在偶数列才打印.
50.         fi
51.
52.         let "i += 1"
53.
54.     done
55. }
56.
57. sift () # 查出非素数.
58. {
59.     let i=$LOWER_LIMIT+1
60.     # 我们从2开始.
61.
62.     until [ "$i" -gt "$UPPER_LIMIT" ]
63.     do
64.
65.         if [ "${Primes[i]}" -eq "$PRIME" ]
66.         # 不要处理已经过滤过的数字(被标识为非素数).
67.         then
68.             t=$i
69.
70.             while [ "$t" -le "$UPPER_LIMIT" ]
71.             do
72.                 let "t += $i "
73.                 Primes[t]=$NON_PRIME
74.                 # 标识为非素数.
75.             done
76.         fi

```

```

77.
78.     let "i += 1"
79.     done
80. }
81.
82. # =====
83. # main ()
84. # 继续调用函数.
85. initialize
86. sift
87. print_primes
88. # 这里就是被称为结构化编程的东西.
89. # =====
90. echo
91.
92. exit 0
93.
94. # ----- #
95. # 因为前面的 'exit' 语句, 所以后边的代码不会运行
96.
97. # 下边的代码, 是由Stephane Chazelas 所编写的埃拉托色尼素数筛子的改进版本,
98. #+ 这个版本可以运行的快一些.
99.
100. # 必须在命令行上指定参数(这个参数就是: 寻找素数的限制范围)
101.
102. UPPER_LIMIT=$1                # 来自于命令行.
103. let SPLIT=UPPER_LIMIT/2        # 从中间值到最大值.
104.
105. Primes=( '' $(seq $UPPER_LIMIT) )
106.
107. i=1
108. until (( ( i += 1 ) > SPLIT )) # 仅需要从中间值检查.
109. do
110.     if [[ -n ${Primes[i]} ]]
111.     then
112.         t=$i
113.         until (( ( t += i ) > UPPER_LIMIT ))
114.         do

```

```

115.     Primes[t]=
116.     done
117. fi
118. done
119. echo ${Primes[*]}
120.
121. exit $?

```

例子 27-14. 埃拉托色尼素数筛子，优化版

```

1.  #!/bin/bash
2.  # 优化过的埃拉托色尼素数筛子
3.  # 脚本由Jared Martin编写，ABS Guide 的作者作了少许修改.
4.  # 在ABS Guide 中经过了许可而使用(感谢!).
5.
6.  # 基于Advanced Bash Scripting Guide中的脚本.
7.  # http://tldp.org/LDP/abs/html/arrays.html#PRIMES0 (ex68.sh).
8.
9.  # http://www.cs.hmc.edu/~oneill/papers/Sieve-JFP.pdf (引用)
10. # Check results against http://primes.utm.edu/lists/small/1000.txt
11.
12. # Necessary but not sufficient would be, e.g.,
13. #      (($sieve 7919 | wc -w) == 1000) && echo "7919 is the 1000th
    prime"
14.
15. UPPER_LIMIT=${1:? "Need an upper limit of primes to search."}
16.
17. Primes=( ' ' $(seq ${UPPER_LIMIT}) )
18.
19. typeset -i i t
20. Primes[i=1]=' ' # 1不是素数
21. until (( ( i += 1 ) > (${UPPER_LIMIT}/i) )) # 只需要ith-way 检查.
22. do                                           # 为什么?
23.     if ((${Primes[t=i*(i-1), i]}))
24.     # 很少见, 但是很有指导意义, 在下标中使用算术扩展。
25.     then
26.         until (( ( t += i ) > ${UPPER_LIMIT} ))
27.         do Primes[t]=; done

```

```

28.     fi
29.     done
30.
31. # echo ${Primes[*]}
32. echo  # 改回原来的脚本，为了漂亮的打印(80-col. 展示).
33. printf "%8d" ${Primes[*]}
34. echo; echo
35.
36. exit $?

```

上边的这个例子是基于数组的素数产生器，还有不使用数组的素数产生器[例子A-15](#) 和[例子 16-46](#)，让我们来比较一番。

数组可以进行一定程度上的扩展，这样就可以模拟一些Bash原本不支持的数据结构。

例子 27-15. 模拟一个压入栈

```

1.  #!/bin/bash
2.  # stack.sh: 模拟压入栈
3.
4.  # 类似于CPU 栈，压入栈依次保存数据项，
5.  #+ 但是取数据时，却反序进行，后进先出。
6.
7.  BP=100          # 栈数组的基址指针。
8.                  # 从元素100 开始。
9.
10. SP=$BP          # 栈指针。
11.                  # 将其初始化为栈"基址"(栈底)
12.
13. Data=            # 当前栈的数据内容。
14.                  # 必须定义为全局变量，
15.                  #+ 因为函数所能够返回的整数存在范围限制。
16.
17.                  # 100      基址                <-- Base Pointer
18.                  # 99       第一个数据元素

```



```

19.          # 98      第二个数据元素
20.          # ...     更多数据
21.          #         最后一个数据元素    <-- Stack pointer
22.
23. declare -a stack
24.
25. push()      # 压栈
26. {
27.     if [ -z "$1" ]          # 没有可压入的数据项?
28.     then
29.         return
30.     fi
31.
32.     let "SP -= 1"           # 更新栈指针.
33.     stack[$SP]=$1
34.     return
35. }
36.
37. pop()       #从栈中弹出数据项.
38. {
39.     Data=                        # 清空保存数据项的中间变量
40.
41.     if [ "$SP" -eq "$BP" ]     # 栈空?
42.     then
43.         return
44.     fi                        # 这使得SP不会超过100,
45.                             #+ 例如, 这可以防止堆栈失控.
46.
47.
48.     Data=${stack[$SP]}
49.     let "SP += 1"             # 更新栈指针
50.     return
51. }
52.
53. status_report()              # 打印当前状态
54. {
55.     echo "-----"
56.     echo "REPORT"

```

```

57.     echo "Stack Pointer = $SP"
58.     echo "Just popped \"'$Data'\" off the stack."
59.     echo "- - - - -"
60.     echo
61. }
62.
63.
64. # =====
65. # 现在，来点乐子。
66.
67. echo
68.
69. # 看你是否能从空栈里弹出数据项来。
70. pop
71. status_report
72.
73. echo
74.
75. push garbage
76. pop
77. status_report          # 压入Garbage，弹出garbage。
78.
79. value1=23;             push $value1
80. value2=skidoo;         push $value2
81. value3=LAST;           push $value3
82.
83. pop                    # LAST
84. status_report
85. pop                    # skidoo
86. status_report
87. pop                    # 23
88. status_report          # 后进，先出！
89.
90. # 注意：栈指针在压栈时减，
91. #+ 在弹出时加。
92.
93. echo
94.

```

```

95.  exit 0
96.
97.
98.  # =====
99.  #
100. # 练习：
101. #
102. # 1) 修改"push()"函数，
103. #     + 使其调用一次就能够压入多个数据项。
104.
105. # 2) 修改"pop()"函数，
106. #     + 使其调用一次就能弹出多个数据项。
107.
108. # 3) 给那些有临界操作的函数添加出错检查。
109. #     说明白一些，就是让这些函数返回错误码，
110. #     + 返回的错误码依赖于操作是否成功完成，
111. #     + 如果没有成功完成，那么就需要启动合适的处理动作。
112.
113. # 4) 以这个脚本为基础，
114. #     + 编写一个用栈实现的四则运算计算器。

```

如果想对数组“下标”做一些比较诡异的操作，可能需要使用中间变量。对于那些有这种需求的项目来说，还是应该考虑使用功能更加强大的编程语言，比如Perl或C。

例子 27-16. 复杂的数组应用：探索一个神秘的数学序列

```

1.  !/bin/bash
2.
3.  # Douglas Hofstadter 的声名狼藉的序列"Q-series":
4.
5.  # Q(1) = Q(2) = 1
6.  # Q(n) = Q(n - Q(n-1)) + Q(n - Q(n-2)), 当 n>2时
7.
8.  # 这是一个令人感到陌生的，没有规律的"乱序"整数序列
9.  #+ 并且行为不可预测

```

```

10. # 序列的头20项，如下所示：
11. # 1 1 2 3 3 4 5 5 6 6 6 8 8 8 10 9 10 11 11 12
12.
13. # 请参考相关书籍，Hofstadter的，"_Goedel, Escher, Bach: An Eternal
    Golden Braid_",
14. #+ 第137页.
15.
16.
17. LIMIT=100      # 需要计算的数列长度.
18. LINEWIDTH=20   # 每行打印的个数.
19.
20. Q[1]=1          # 数列的头两项都为1.
21. Q[2]=1
22.
23. echo
24. echo "Q-series [$LIMIT terms]:"
25. echo -n "${Q[1]} "          # 输出数列头两项.
26. echo -n "${Q[2]} "
27.
28. for ((n=3; n <= $LIMIT; n++)) # C风格的循环条件.
29. do # Q[n] = Q[n - Q[n-1]] + Q[n - Q[n-2]] for n>2
30. # 需要将表达式拆开，分步计算，
31. #+ 因为Bash 不能够很好的处理复杂数组的算术运算.
32.
33.     let "n1 = $n - 1"          # n-1
34.     let "n2 = $n - 2"          # n-2
35.
36.     t0=`expr $n - ${Q[n1]}`    # n - Q[n-1]
37.     t1=`expr $n - ${Q[n2]}`    # n - Q[n-2]
38.
39.     T0=${Q[t0]}                # Q[n - Q[n-1]]
40.     T1=${Q[t1]}                # Q[n - Q[n-2]]
41.
42.
43.     Q[n]=`expr $T0 + $T1`      # Q[n - Q[n-1]] + Q[n - Q[n-2]]
44.     echo -n "${Q[n]} "
45.
46.     if [ `expr $n % $LINEWIDTH` -eq 0 ]          # 格式化输出

```

```

47.     then    #      ^ 取模操作
48.         echo # 把每行都拆为20个数字的小块.
49.     fi
50.
51. done
52.
53. echo
54.
55. exit 0
56.
57. # 这是Q-series的一个迭代实现.
58. # 更直接明了的实现是使用递归, 请读者作为练习完成.
59. # 警告: 使用递归的方法来计算这个数列的话, 会花费非常长的时间.
60. #+ C/C++ 将会计算的快一些。

```

Bash仅仅支持一维数组，但是我们可以使用一个小手段，这样就可以模拟多维数组了。

例子 27-17. 模拟一个二维数组，并使它倾斜

```

1.  #!/bin/bash
2.  # twodim.sh: 模拟一个二维数组.
3.
4.  # 一维数组由单行组成.
5.  # 二维数组由连续的多行组成.
6.
7.  Rows=5
8.  Columns=5
9.  # 5 X 5 的数组.
10.
11. declare -a alpha          # char alpha [Rows] [Columns];
12.                           # 没必要声明. 为什么?
13.
14. load_alpha ()
15. {
16.     local rc=0
17.     local index

```

```

18.
19.     for i in A B C D E F G H I J K L M N O P Q R S T U V W X Y
20.     do         # 你可以随你的心意，使用任意符号。
21.         local row=`expr $rc / $Columns`
22.         local column=`expr $rc % $Rows`
23.         let "index = $row * $Rows + $column"
24.         alpha[$index]=$i
25.     # alpha[$row][$column]
26.     let "rc += 1"
27. done
28.     # 更简单的方法：
29.     #+ declare -a alpha=( A B C D E F G H I J K L M N O P Q R S T
    U V W X Y )
30.     #+ 但是如果写的话，就缺乏二维数组的"风味"了。
31. }
32.
33. print_alpha ()
34. {
35.     local row=0
36.     local index
37.     echo
38.     while [ "$row" -lt "$Rows" ]    # 以"行序为主"进行打印：
39.     do                                #+ 行号不变(外层循环)，
40.                                    #+ 列号进行增长。
41.         local column=0
42.
43.         echo -n "          "        # 按照行方向打印"正方形"数组。
44.
45.         while [ "$column" -lt "$Columns" ]
46.         do
47.             let "index = $row * $Rows + $column"
48.             echo -n "${alpha[index]} " # alpha[$row][$column]
49.             let "column += 1"
50.         done
51.
52.         let "row += 1"
53.         echo
54.     done

```

```

55.     # 更简单的等价写法为:
56.     #     echo ${alpha[*]} | xargs -n $Columns
57.     echo
58. }
59.
60. filter ()      # 过滤掉负的数组下标.
61. {
62.
63.     echo -n " " # 产生倾斜.
64.             # 解释一下, 这是怎么做到的.
65.     if [[ "$1" -ge 0 && "$1" -lt "$Rows" && "$2" -ge 0 && "$2" -lt
"$Columns" ]]
66.     then
67.         let "index = $1 * $Rows + $2"
68.         # 现在, 按照旋转方向进行打印.
69.         echo -n " ${alpha[index]}"
70.         #             alpha[$row][$column]
71.     fi
72.
73. }
74.
75. rotate () # 将数组旋转45度 --
76. {         #+ 从左下角进行"平衡".
77.     local row
78.     local column
79.
80.     for (( row = Rows; row > -Rows; row-- ))
81.     do      # 反向步进数组, 为什么?
82.
83.         for (( column = 0; column < Columns; column++ ))
84.         do
85.
86.             if [ "$row" -ge 0 ]
87.             then
88.                 let "t1 = $column - $row"
89.                 let "t2 = $column"
90.             else
91.                 let "t1 = $column"

```

```

92.         let "t2 = $column + $row"
93.     fi
94.     filter $t1 $t2          # 将负的数组下标过滤出来.
95.                             # 如果你不做这一步, 将会怎样?
96. done
97.     echo; echo
98.
99. done
100.
101. # 数组旋转的灵感来源于Herbert Mayer 所著的
102. #+ "Advanced C Programming on the IBM PC"的例子(第143-146页)
103. #+ (参见参考书目).
104. # 由此可见, C语言能够做到的好多事情,
105. #+ 用shell 脚本一样能够做到.
106. }
107.
108.
109. #----- 现在, 让我们开始吧. -----#
110. load_alpha          # 加载数组
111. print_alpha         # 打印数组.
112. rotate              # 逆时针旋转45度打印.
113. #-----#
114.
115. exit 0
116.
117. # 这有点做作, 不是那么优雅.
118.
119. # 练习:
120. # -----
121. # 1) 重新实现数组加载和打印函数,
122. #    让其更直观, 可读性更强.
123. #
124. # 2) 详细地描述旋转函数的原理.
125. #    提示: 思考一下倒序索引数组的实现.
126. #
127. # 3) 重写这个脚本, 扩展它, 让不仅仅能够支持非正方形的数组.
128. #    比如6 X 4的数组.
129. #    尝试一下, 在数组旋转时, 做到最小"失真".

```


二维数组本质上其实就是一个一维数组，只不过是添加了行和列的寻址方式，来引用和操作数组的元素而已。

这里有一个精心制作的模拟二维数组的例子，请参考[例子 A-10](#)。

还有更多使用数组的有趣的脚本，请参考：

- [例子 12-3](#)
- [例子 16-46](#)
- [例子 A-22](#)
- [例子 A-44](#)
- [例子 A-41](#)
- [例子 A-42](#)

30 网络编程

- 30 网络编程

30 网络编程

The Net's a cross between an elephant and a white elephant sale: it never forgets, and it's always crap.

—Nemo

Linux系统拥有一系列的工具，用于访问、操作和调解网络连接。我们能够把其中的一部分工具整合到脚本中 — 这些脚本能扩展我们对网络的认知，有用的脚本还能方便网络管理。

以下是一个简单的CGI脚本，阐述如何连接到远程服务器。

例子 30-1. 打印服务器环境

```
1. #!/bin/bash
2. # test-cgi.sh
3. # by Michael Zick
4. # Used with permission
5.
6. # 您应该根据您的情况修改相应的Bash路径
7. # （在ISP的服务器中，Bash一般不会放在正常的位置）
8. # 其他位置： /usr/bin 或者 /usr/local/bin
9. # 甚至应该在sha-bang中不用任何路径运行它
10.
11. # 取消通配符
12. set -f
13.
14. # Http Header（译者注：此头信息是告诉浏览器服务器返回的内容格式）
15. echo Content-type: text/plain
16. echo
17.
```

```
18. echo CGI/1.0 test script report:
19. echo
20.
21. echo environment settings:
22. set
23. echo
24.
25. echo whereis bash?
26. whereis bash
27. echo
28.
29.
30. echo who are we?
31. echo ${BASH_VERSINFO[*]}
32. echo
33.
34. echo argc is $#. argv is "$*".
35. echo
36.
37. # CGI/1.0 预期的环境变量。
38.
39. echo SERVER_SOFTWARE = $SERVER_SOFTWARE
40. echo SERVER_NAME = $SERVER_NAME
41. echo GATEWAY_INTERFACE = $GATEWAY_INTERFACE
42. echo SERVER_PROTOCOL = $SERVER_PROTOCOL
43. echo SERVER_PORT = $SERVER_PORT
44. echo REQUEST_METHOD = $REQUEST_METHOD
45. echo HTTP_ACCEPT = "$HTTP_ACCEPT"
46. echo PATH_INFO = "$PATH_INFO"
47. echo PATH_TRANSLATED = "$PATH_TRANSLATED"
48. echo SCRIPT_NAME = "$SCRIPT_NAME"
49. echo QUERY_STRING = "$QUERY_STRING"
50. echo REMOTE_HOST = $REMOTE_HOST
51. echo REMOTE_ADDR = $REMOTE_ADDR
52. echo REMOTE_USER = $REMOTE_USER
53. echo AUTH_TYPE = $AUTH_TYPE
54. echo CONTENT_TYPE = $CONTENT_TYPE
55. echo CONTENT_LENGTH = $CONTENT_LENGTH
```

```

56.
57.  exit 0
58.
59.  # 在这里，文档给出一些简短的指令。
60.  :<<- '_test_CGI_'
61.
62.  1) 将此文档放到http://domain.name/cgi-bin的目录。
63.  2) 然后，访问http://domain.name/cgi-bin/test-cgi.sh.
64.
65.  _test_CGI_

```

出于安全的考虑，确认连接计算机的IP地址是有用的。

例子 30-2. IP地址

```

1.  #!/bin/bash
2.  # ip-addresses.sh
3.  # 列出您的计算机所连接的IP地址。
4.
5.  # 受Greg Bledsoe的ddos.sh脚本所启发，
6.  # Linux Journal, 2011年3月9号。
7.  # URL:
8.  # http://www.linuxjournal.com/content/back-dead-simple-bash-
  complex-ddos
9.  # Greg licensed his script under the GPL2,
10. #+ and as a derivative, this script is likewise GPL2.
11.
12. connection_type=TCP      # 也可以使用UDP
13. field=2                  # Which field of the output we're interested in.
14. no_match=LISTEN          # 过滤出包含此字符串的记录，为什么？
15. lsof_args=-ni            # -i 列出互联网相关的文件。
16.                          # -n 使用数值IP地址。
17.                          # 如果不使用-n选项，会发生什么情况？试试看。
18. router="[0-9][0-9][0-9][0-9][0-9]->"
19. # 删除路由信息。
20.
21. lsof "$lsof_args" | grep $connection_type | grep -v "$no_match" |
22.     awk '{print $9}' | cut -d : -f $field | sort | uniq |

```

```

23.      sed s/"^$router"//
24.
25.  # Bledsoe的脚本将过滤出的IP地址结果赋给一个变量（类似上面的19行到22行）。
26.  # 他检查连接到一个IP地址的多个连接，
27.  # 使用：
28.  #
29.  # iptables -I INPUT -s $ip -p tcp -j REJECT --reject-with tcp-
    reset
30.  #
31.  # ... 在每一次的60秒延迟循环中，拒绝来自DDOS攻击的数据包。
32.
33.
34.  # 练习：
35.  # -----
36.  # 使用'iptables'命令扩展这个脚本
37.  #+ 来拒绝一些来自垃圾邮件发送者的IP域名的连接请求。

```

更多网络编程的例子：

- [Getting the time from nist.gov](#)
- [Downloading a URL](#)
- [A GRE tunnel](#)
- [Checking if an Internet server is up](#)
- [例子 16-41](#)
- [例子 A-28](#)
- [例子 A-29](#)
- [例子 29-1](#)

更多资料请看《[System and Administrative Commands](#)》的章节“[网络命令](#)”，以及《[External Filters, Programs and Commands](#)》的章节“[通信命令](#)”。

32 调试

- 32 调试
- `#!/bin/bash`
- `assert.sh`
 - `#`

32 调试

调试代码要比写代码困难两倍。因此，你写代码时越多的使用奇技淫巧（自做聪明），顾名思义，你越难以调试它。 —Brian Kernighan

Bash shell中不包含内置的debug工具，甚至没有调试专用的命令和结构。当调试非功能脚本，产生语法错误或者有错别字时，往往是无用的错误提示消息。

例子 32-1. 一个错误脚本

```
1. #!/bin/bash
2. # ex74.sh
3.
4. # 这是一个错误脚本，但是它错在哪？
5.
6. a=37
7.
8. if [ $a -gt 27 ]
9. then
10.     echo $a
11. fi
12.
13. exit $?    # 0! 为什么？
```

脚本的输出：

```
1. ./ex74.sh: [37: command not found
```

上边的脚本究竟哪错了(提示: 注意if的后边)

例子 32-2. 缺少关键字

```
1. #!/bin/bash
2. # missing-keyword.sh
3. # 这个脚本会提示什么错误信息?
4.
5. for a in 1 2 3
6. do
7.     echo "$a"
8. # done      #所需关键字'done'在第8行被注释掉.
9.
10. exit 0      # 将不会在这退出!
11.
12. #在命令行执行完此脚本后
13. 输入: echo $?
14. 输出: 2
```

脚本的输出:

```
1. missing-keyword.sh: line 10: syntax error: unexpected end of file
```

注意, 其实不必参考错误信息中指出的错误行号. 这行只不过是Bash解释器最终认定错误的地方.

出错信息在报告产生语法错误的行号时, 可能会忽略脚本的注释行. 如果脚本可以执行, 但并不如你所期望的那样工作, 怎么办? 通常情况下, 这都是由常见的逻辑错误所产生的.

例子 32-3.

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



```

2.
3. # 这个脚本应该删除在当前目录下所有文件名中含有空格的文件
4. # 它不能正常运行，为什么？
5.
6. badname=`ls | grep ' '`
7.
8. # Try this:
9. # echo "$badname"
10.
11. rm "$badname"
12.
13. exit 0

```

可以通过把echo “\$badname”行的注释符去掉，找出例子 29-3中的错误， 看一下echo出来的信息，是否按你期望的方式运行。

在这种特殊的情况下，rm “\$badname”不能得到预期的结果，因为\$badname不应该加双引号。加上双引号会让rm只有一个参数(这就只能匹配一个文件名)。一种不完善的解决办法是去掉\$badname外面的引号，并且重新设置IFS，让IFS只包含一个换行符，IFS=\$'\n'。但是，下面这个方法更简单。

```

1. # 删除包含空格的文件的正确方法。
2. rm *\ *
3. rm *" " *
4. rm *' ' *
5. # 感谢. S.C.

```

总结一下这个问题脚本的症状：

>

1. 由于“syntax error”(语法错误)使得脚本停止运行，
2. 或者脚本能够运行，但是并不是按照我们所期望的那样运行(逻辑错误)。

- 脚本能够按照我们所期望的那样运行，但是有烦人的副作用(逻辑炸弹)。

如果想调试脚本，可以用以下方式：

- echo语句可以放在脚本中存在疑问的位置上，观察变量的值，来了解脚本运行时的情况。

```

1.  ### debecho (debug-echo), by Stefano Falsetto ###
2.  ### Will echo passed parameters only if DEBUG is set to a
   value. ###
3.  debecho () {
4.      if [ ! -z "$DEBUG" ]; then
5.          echo "$1" >&2
6.          # ^^^ to stderr
7.      fi
8.  }
9.
10. DEBUG=on
11. Whatever=whatnot
12. debecho $Whatever    # whatnot
13.
14. DEBUG=
15. Whatever=notwhat
16. debecho $Whatever    # (Will not echo.)

```

- 使用过滤器tee来检查临界点上的进程或数据流。
- 设置选项 -n -v -x

sh -n scriptname不会运行脚本，只会检查脚本的语法错误。这等于把set -n或set -o noexec插入脚本中。注意，某些类型的语法错误不会被这种方式检查出来。

sh -v scriptname将会在运行脚本之前，打印出每一个命令。这等于把set -v或set -o verbose插入到脚本中。

选项 `-n` 和 `-v` 可以同时使用。 `sh -nv scriptname` 将会给出详细的语法检查。

`sh -x scriptname` 会打印出每个命令执行的结果，但只使用缩写形式。这等价于在脚本中插入 `set -x` 或 `set -o xtrace`。

把 `set -u` 或 `set -o nounset` 插入到脚本中，并运行它，就会在每个试图使用未声明变量的地方给出一个 `unbound variable` 错误信息。

```

1.  set -u    # Or    set -o nounset
2.
3.  # Setting a variable to null will not trigger the
    error/abort.
4.  # unset_var=
5.
6.  echo $unset_var    # Unset (and undeclared) variable.
7.  echo "Should not echo!"
8.
9.  #sh t2.sh
10. #t2.sh: line 6: unset_var: unbound variable

```

3. 使用“断言”功能在脚本的关键点进行测试的变量或条件。（这是从C借来的一个想法）

Example 32-4. Testing a condition with an assert

...

`!/bin/bash`

`assert.sh`

#

```
assert () # If condition false,
{ #+ exit from script
```

```
1.                                     #+ with appropriate error message.
```

```
E_PARAM_ERR=98
```

```
E_ASSERT_FAILED=99
```

```
1.  if [ -z "$2" ]           # Not enough parameters passed
2.  then                     #+ to assert() function.
3.      return $E_PARAM_ERR  # No damage done.
4.  fi
5.
6.  lineno=$2
7.
8.  if [ ! $1 ]
9.  then
10.     echo "Assertion failed: \"$1\""
11.     echo "File \"$0\", line $lineno"    # Give name of file and
line number.
12.     exit $E_ASSERT_FAILED
13.     # else
14.     # return
15.     # and continue executing the script.
16.     fi
17. } # Insert a similar assert() function into a script you need to
debug.
18. #####
19.
20.
21. a=5
22. b=4
23. condition="$a -lt $b"    # Error message and exit from script.
24.                          # Try setting "condition" to something
else
```

```

25.                                     #+ and see what happens.
26.
27. assert "$condition" $LINENO
28. # The remainder of the script executes only if the "assert" does
    not fail.
29.
30.
31. # Some commands.
32. # Some more commands . . .
33. echo "This statement echoes only if the \"assert\" does not fail."
34. # . . .
35. # More commands . . .
36.
37. exit $?
38. ```

```

1. 使用变量\$LINENO和内建命令caller.
2. 捕获exit返回值.

The exit command in a script triggers a signal 0, terminating the process, that is, the script itself. [1] It is often useful to trap the exit, forcing a “printout” of variables, for example. The trap must be the first command in the script.

捕获信号

trap

Specifies an action on receipt of a signal; also useful for debugging.

A signal is a message sent to a process, either by

the kernel or another process, telling it to take some specified action (usually to terminate). For example, hitting a Control-C sends a user interrupt, an INT signal, to a running program.

A simple instance:

```
1. trap '' 2
2. # Ignore interrupt 2 (Control-C), with no action specified.
3.
4. trap 'echo "Control-C disabled."' 2
5. # Message when Control-C pressed.
```

Example 32-5. Trapping at exit

```
1. #!/bin/bash
2. # Hunting variables with a trap.
3.
4. trap 'echo Variable Listing --- a = $a b = $b' EXIT
5. # EXIT is the name of the signal generated upon exit from a
   script.
6. #
7. # The command specified by the "trap" doesn't execute until
8. #+ the appropriate signal is sent.
9.
10. echo "This prints before the \"trap\" --"
11. echo "even though the script sees the \"trap\" first."
12. echo
13.
14. a=39
15. b=36
16.
17. exit 0
18.
19.
20. # Note that commenting out the 'exit' command makes no difference,
```

```
21.  #+ since the script exits in any case after running out of
    commands.
```

Example 32-6. Cleaning up after Control-C

```
1.  #!/bin/bash
2.  # logon.sh: A quick 'n dirty script to check whether you are on-
    line yet.
3.
4.  umask 177 # Make sure temp files are not world readable.
5.
6.
7.  TRUE=1
8.  LOGFILE=/var/log/messages
9.  # Note that $LOGFILE must be readable
10. #+ (as root, chmod 644 /var/log/messages).
11. TEMPFILE=temp.$$
12. # Create a "unique" temp file name, using process id of the
    script.
13. # Using 'mktemp' is an alternative.
14. # For example:
15. # TEMPFILE=`mktemp temp.XXXXXX`
16. KEYWORD=address
17. # At logon, the line "remote IP address xxx.xxx.xxx.xxx"
18. # appended to /var/log/messages.
19. ONLINE=22
20. USER_INTERRUPT=13
21. CHECK_LINES=100
22. # How many lines in log file to check.
23.
24. trap 'rm -f $TEMPFILE; exit $USER_INTERRUPT' TERM INT
25. # Cleans up the temp file if script interrupted by control-c.
26.
27. echo
28.
29. while [ $TRUE ] #Endless loop.
30. do
31.     tail -n $CHECK_LINES $LOGFILE> $TEMPFILE
```

```

32.  # Saves last 100 lines of system log file as temp file.
33.  # Necessary, since newer kernels generate many log messages at
    log on.
34.  search=`grep $KEYWORD $TEMPFILE`
35.  # Checks for presence of the "IP address" phrase,
36.  #+ indicating a successful logon.
37.
38.  if [ ! -z "$search" ] # Quotes necessary because of possible
    spaces.
39.  then
40.      echo "On-line"
41.      rm -f $TEMPFILE      # Clean up temp file.
42.      exit $ONLINE
43.  else
44.      echo -n "."          # The -n option to echo suppresses
    newline,
45.                          #+ so you get continuous rows of dots.
46.  fi
47.
48.  sleep 1
49. done
50.
51.
52. # Note: if you change the KEYWORD variable to "Exit",
53. #+ this script can be used while on-line
54. #+ to check for an unexpected logoff.
55.
56. # Exercise: Change the script, per the above note,
57. #           and prettify it.
58.
59. exit 0
60.
61.
62. # Nick Drage suggests an alternate method:
63.
64. while true
65. do ifconfig ppp0 | grep UP 1> /dev/null && echo "connected" &&
    exit 0

```



```

66.     echo -n "."      # Prints dots (.....) until connected.
67.     sleep 2
68. done
69.
70. # Problem: Hitting Control-C to terminate this process may be
    insufficient.
71. #+          (Dots may keep on echoing.)
72. # Exercise: Fix this.
73.
74.
75.
76. # Stephane Chazelas has yet another alternative:
77.
78. CHECK_INTERVAL=1
79.
80. while ! tail -n 1 "$LOGFILE" | grep -q "$KEYWORD"
81. do echo -n .
82.     sleep $CHECK_INTERVAL
83. done
84. echo "On-line"
85.
86. # Exercise: Discuss the relative strengths and weaknesses
87. #           of each of these various approaches.
88. Example 32-7. A Simple Implementation of a Progress Bar
89.
90. #! /bin/bash
91. # progress-bar2.sh
92. # Author: Graham Ewart (with reformatting by ABS Guide author).
93. # Used in ABS Guide with permission (thanks!).
94.
95. # Invoke this script with bash. It doesn't work with sh.
96.
97. interval=1
98. long_interval=10
99.
100. {
101.     trap "exit" SIGUSR1
102.     sleep $interval; sleep $interval

```

```

103.     while true
104.     do
105.         echo -n '.'      # Use dots.
106.         sleep $interval
107.     done; } &          # Start a progress bar as a background
                           process.
108.
109. pid=$!
110. trap "echo !; kill -USR1 $pid; wait $pid"  EXIT      # To handle
                           ^C.
111.
112. echo -n 'Long-running process '
113. sleep $long_interval
114. echo ' Finished!'
115.
116. kill -USR1 $pid
117. wait $pid          # Stop the progress bar.
118. trap EXIT
119.
120. exit $?

```

Note

The `DEBUG` argument to `trap` causes a specified action to execute after every command in a script. This permits tracing variables, for example.

Example 32-8. Tracing a variable

```

1.
2. #!/bin/bash
3.
4. trap 'echo "VARIABLE-TRACE> \${variable} = \"\${variable}\"" ' DEBUG
5. # Echoes the value of $variable after every command.
6.
7. variable=29; line=$LINENO
8.
9. echo "   Just initialized \${variable} to $variable in line number

```

```

    $line."
10.
11. let "variable *= 3"; line=$LINENO
12. echo "    Just multiplied \${variable} by 3 in line number $line."
13.
14. exit 0
15.
16. # The "trap 'command1 . . . command2 . . .' DEBUG" construct is
17. #+ more appropriate in the context of a complex script,
18. #+ where inserting multiple "echo \${variable}" statements might be
19. #+ awkward and time-consuming.
20.
21. # Thanks, Stephane Chazelas for the pointer.

```

Output of script:

VARIABLE-TRACE> \$variable = ""

VARIABLE-TRACE> \$variable = "29"

Just initialized \$variable to 29.

VARIABLE-TRACE> \$variable = "29"

VARIABLE-TRACE> \$variable = "87"

Just multiplied \$variable by 3.

VARIABLE-TRACE> \$variable = "87"

Of course, the trap command has other uses aside from debugging, such as disabling certain keystrokes within a script (see Example A-43).

Example 32-9. Running multiple processes (on an SMP box)

```

1.
2. #!/bin/bash
3. # parent.sh
4. # Running multiple processes on an SMP box.

```

```
5. # Author: Tedman Eng
6.
7. # This is the first of two scripts,
8. #+ both of which must be present in the current working directory.
9.
10.
11.
12.
13. LIMIT=$1          # Total number of process to start
14. NUMPROC=4         # Number of concurrent threads (forks?)
15. PROCID=1          # Starting Process ID
16. echo "My PID is $$"
17.
18. function start_thread() {
19.     if [ $PROCID -le $LIMIT ] ; then
20.         ./child.sh $PROCID&
21.         let "PROCID++"
22.     else
23.         echo "Limit reached."
24.         wait
25.         exit
26.     fi
27. }
28.
29. while [ "$NUMPROC" -gt 0 ]; do
30.     start_thread;
31.     let "NUMPROC--"
32. done
33.
34.
35. while true
36. do
37.
38. trap "start_thread" SIGRTMIN
39.
40. done
41.
42. exit 0
```

```

43.
44.
45.
46. # ===== Second script follows =====
47.
48.
49. #!/bin/bash
50. # child.sh
51. # Running multiple processes on an SMP box.
52. # This script is called by parent.sh.
53. # Author: Tedman Eng
54.
55. temp=$RANDOM
56. index=$1
57. shift
58. let "temp %= 5"
59. let "temp += 4"
60. echo "Starting $index  Time:$temp" "$@"
61. sleep ${temp}
62. echo "Ending $index"
63. kill -s SIGRTMIN $PPID
64.
65. exit 0
66.
67.
68. # ===== SCRIPT AUTHOR'S NOTES
   ===== #
69. #  It's not completely bug free.
70. #  I ran it with limit = 500 and after the first few hundred
   iterations,
71. #+ one of the concurrent threads disappeared!
72. #  Not sure if this is collisions from trap signals or something
   else.
73. #  Once the trap is received, there's a brief moment while
   executing the
74. #+ trap handler but before the next trap is set.  During this time,
   it may
75. #+ be possible to miss a trap signal, thus miss spawning a child

```

```

process.
76.
77. # No doubt someone may spot the bug and will be writing
78. #+ . . . in the future.
79.
80.
81.
82. #
=====
#
83.
84.
85.
86. # -----
----#
87.
88.
89.
90. #####
91. # The following is the original script written by Vernia Damiano.
92. # Unfortunately, it doesn't work properly.
93. #####
94.
95. #!/bin/bash
96.
97. # Must call script with at least one integer parameter
98. #+ (number of concurrent processes).
99. # All other parameters are passed through to the processes
started.
100.
101.
102. INDICE=8          # Total number of process to start
103. TEMPO=5           # Maximum sleep time per process
104. E_BADARGS=65      # No arg(s) passed to script.
105.
106. if [ $# -eq 0 ] # Check for at least one argument passed to script.
107. then
108.     echo "Usage: `basename $0` number_of_processes [passed params]"

```

```

109.     exit $E_BADARGS
110. fi
111.
112. NUMPROC=$1           # Number of concurrent process
113. shift
114. PARAMETRI=( "$@" )   # Parameters of each process
115.
116. function avvia() {
117.     local temp
118.     local index
119.     temp=$RANDOM
120.     index=$1
121.     shift
122.     let "temp %= $TEMPO"
123.     let "temp += 1"
124.     echo "Starting $index Time:$temp" "$@"
125.     sleep ${temp}
126.     echo "Ending $index"
127.     kill -s SIGRTMIN $$
128. }
129.
130. function parti() {
131.     if [ $INDICE -gt 0 ] ; then
132.         avvia $INDICE "${PARAMETRI[@]}" &
133.         let "INDICE--"
134.     else
135.         trap : SIGRTMIN
136.     fi
137. }
138.
139. trap parti SIGRTMIN
140.
141. while [ "$NUMPROC" -gt 0 ]; do
142.     parti;
143.     let "NUMPROC--"
144. done
145.
146. wait

```

```

147. trap - SIGRTMIN
148.
149. exit $?
150.
151. : <<SCRIPT_AUTHOR_COMMENTS
152. I had the need to run a program, with specified options, on a
    number of
153. different files, using a SMP machine. So I thought [I'd] keep
    running
154. a specified number of processes and start a new one each time . . .
    one
155. of these terminates.
156.
157. The "wait" instruction does not help, since it waits for a given
    process
158. or *all* process started in background. So I wrote [this] bash
    script
159. that can do the job, using the "trap" instruction.
160.   --Vernia Damiano
161. SCRIPT_AUTHOR_COMMENTS

```

Note

trap '' SIGNAL (two adjacent apostrophes) disables SIGNAL for the remainder of the script. trap SIGNAL restores the functioning of SIGNAL once more. This is useful to protect a critical portion of a script from an undesirable interrupt.

```

1. trap '' 2 # Signal 2 is Control-C, now disabled.
2. command
3. command
4. command
5. trap 2    # Reenables Control-C

```


33 选项

- 33 选项

33 选项

选项用来更改shell和脚本的行为。

`set` 命令用来打开脚本中的选项。你可以在脚本中任何你想让选项生效的地方插入 `set -o option-name`，或者使用更简单的形式，`set -option-abbrev`。这两种形式是等价的。

```
1. #!/bin/bash
2.
3. set -o verbose
4. # # 打印出所有执行前的命令。
```

```
1. #!/bin/bash
2.
3. set -v
4. # 与上边的例子具有相同的效果。
```



如果你想在脚本中禁用某个选项，可以使用 `set +o option-name` 或 `set +option-abbrev`。

```
1. #!/bin/bash
2. set -o verbose
3. # 激活命令回显。
4. command
5. ...
6. command
7.
8. set +o verbose
```

```

9.  # 禁用命令回显.
10. command
11. # 没有命令回显了.
12.
13. set -v
14. # 激活命令回显.
15. command
16. ...
17. command
18.
19. set +v
20. # 禁用命令回显.
21. command
22.
23. exit 0

```

还有另一种可以在脚本中启用选项的方法，那就是在脚本头部，`#!`的后边直接指定选项。

```

1. #!/bin/bash -x
2. #
3. # 下边是脚本的主要内容.

```

也可以从命令行中打开脚本的选项。某些不能与`set`命令一起用的选项就可以使用这种方法来打开。`-i`就是其中之一，这个选项用来强制脚本以交互的方式运行。

bash - v script - name

bash - o verbose script - name

下表列出了一些有用的选项。它们都可以使用缩写的方式来指定(开头加一个破折号)，也可以使用完整名字来指定(开头加上双破折号，或者使用`-o`选项来指定)。

表格 33-1. Bash选项

缩写	名称	作用	
-B	brace expansion	开启 大括号展开 (默认 setting = on)	
+B	brace expansion	关闭大括号展开	
-C	noclobber	防止重定向时覆盖文件(可能会被>\	覆盖)
-D	(none)	列出用双引号引用起来的, 以\$为前缀的字符串, 但是不执行脚本中的命令	
-a	all export	export(导出)所有定义过的变量	
-b	notify	当后台运行的作业终止时, 给出通知(脚本中并不常见)	
-c ...	(none)	从...中读取命令	
checkjobs	(none)	通知有活跃shell 任务 的用户退出。 Bash 4 版本中引入, 仍然处于“实验”阶段. 用法: shopt -s checkjobs .(注意: 可能会hang!)	
-e	errexit	当脚本发生第一个错误时, 就退出脚本, 换种说法就是, 当一个命令返回非零值时, 就退出脚本(除了 until 或 while loops , if-tests , list constructs)	
-f	noglob	禁用文件名扩展(就是禁用globbing)	
globstar	globbing star-match	打开 globbing 操作符(Bash 4+). 使用方法: shopt -s globstar	
-i	interactive	让脚本以交互模式运行	
-n	noexec	从脚本中读取命令, 但是不执行它们(做语法检查)	
-o Option-Name	(none)	调用Option-Name选项	
-o posix	POSIX	修改Bash或被调用脚本的行为, 使其符合 POSIX 标准.	
-o pipefail	pipe failure	创建一个管道去返回最后一条命令的 退出状态码 , 这个返回值是一个非0的返回值	
-p	privileged	以“suid”身份来运行脚本(小心!)	
-r	restricted	以受限模式来运行脚本(参考 22).	
-s	stdin	从stdin 中读取命令	
-t	(none)	执行完第一个命令之后, 就退出	
-u	nounset	如果尝试使用了未定义的变量, 就会输出一个错误消息, 然后强制退出	
-v	verbose	在执行每个命令之前, 把每个命令打印到stdout上	
-x	xtrace	与-v选项类似, 但是会打印完整命令	

-	(none)	选项结束标志。后面的参数为 位置参数 。
—	(none)	unset(释放)位置参数。如果指定了参数列表(—arg1 arg2)，那么位置参数将会依次设置到参数列表中。



34 陷阱

- 第34章 陷阱
 - 以下的做法（非推荐）将让你原本平淡无奇的生活激动不已。
 - 注意事项

第34章 陷阱

Turandot: Gli enigmi sono tre, la morte una!
Caleph: No, no! Gli enigmi sono tre, una la vita!
 —Puccini

以下的做法（非推荐）将让你原本平淡无奇的生活激动不已。

- 将保留字或特殊字符声明为变量名。

```
1. case=value0          # 引发错误。
2. 23skidoo=value1      # 也会引发错误。
3. # 以数字开头的变量名是被shell保留使用的。
4. # 试试_23skidoo=value1。以下划线开头的变量名就没问题。
5.
6. # 然而 . . . 只用一个下划线作为变量名就不行。
7. _=25
8. echo $_              # $_是一个特殊变量，代表最后一个命令的最后一个参数。
9. # 但是，_是一个有效的函数名！
10.
11. xyz(!(*=value2      # 引起严重的错误。
12. # Bash3.0之后，标点不能出现在变量名中。
```

- 使用连字符或其他保留字符来做变量名（或函数名）。

```
1. var-1=23
2. # 用 'var_1' 代替。
```

```

3.
4. function-whatever () # 错误
5. # 用 'function_whatever ()' 代替。
6.
7.
8. # Bash3.0之后, 标点不能出现在函数名中。
9. function.whatever () # 错误
10. # 用 'functionWhatever ()' 代替。

```

- 让变量名与函数名相同。 这会使得脚本的可读性变得很差。

```

1. do_something ()
2. {
3.     echo "This function does something with \"$1\"."
4. }
5.
6. do_something=do_something
7.
8. do_something do_something
9.
10. # 这么做是合法的, 但会让人混淆。

```

- 不合时宜的使用空白符。与其他编程语言相比, Bash非常讲究空白符的使用。

```

1. var1 = 23 # 'var1=23'才是正确的。
2. # 对于上边这一行来说, Bash会把"var1"当作命令来执行,
3. # "="和"23"会被看作"命令""var1"的参数。
4.
5. let c = $a - $b # 'let c=$a-$b'或'let "c = $a - $b"'才是正确的。
6.
7. if [ $a -le 5 ] # if [ $a -le 5 ] 是正确的。
8. #           ^^   if [ "$a" -le 5 ] 这么写更好。
9. #           # [[ $a -le 5 ]] 也行。

```

- 在大括号包含的代码块中, 最后一条命令没有以分号结尾。

```

1. { ls -l; df; echo "Done." }
2. # bash: syntax error: unexpected end of file
3.
4. { ls -l; df; echo "Done."; }
5. # ^ ##### 最后的这条命令必须以分号结尾。

```

- 假定未被初始化的变量（赋值前的变量）被“清0”。事实上，未初始化的变量值为“null”，而不是0。

```

1. #!/bin/bash
2.
3. echo "uninitialized_var = $uninitialized_var"
4. # uninitialized_var =
5.
6. # 但是 . . .
7. # if $BASH_VERSION ≥ 4.2; then
8.
9. if [[ ! -v uninitialized_var ]]
10. then
11.     uninitialized_var=0    # Initialize it to zero!
12. fi

```

- 混淆测试符号=和-ep。请记住，=用于比较字符变量，而-ep用来比较整数。

```

1. if [ "$a" = 273 ]      # $a是整数还是字符串？
2. if [ "$a" -eq 273 ]    # $a为整数。
3.
4. # 有些情况下，即使你混用-ep和=，也不会产生错误的结果。
5. # 然而 . . .
6.
7.
8. a=273.0    # 不是一个整数。
9.
10. if [ "$a" = 273 ]
11. then
12.     echo "Comparison works."

```



```

13. else
14.     echo "Comparison does not work."
15. fi    # Comparison does not work.
16.
17. # 与a=" 273"和a="0273"相同。
18.
19.
20. # 类似的, 如果对非整数值使用“-ep”的话, 就会产生问题。
21.
22. if [ "$a" -eq 273.0 ]
23. then
24.     echo "a = $a"
25. fi    # 产生了错误消息而退出。
26. # test.sh: [: 273.0: integer expression expected

```

- 误用了字符串比较操作符。

样例 34-1. 数字比较与字符串比较并不相同

```

1. #!/bin/bash
2. # bad-op.sh: 尝试一下对整数使用字符串比较。
3.
4. echo
5. number=1
6.
7. # 下面的"while循环"有两个过错误:
8. #+ 一个比较明显, 而另一个比较隐蔽。
9.
10. while [ "$number" < 5 ]    # 错! 应该是: while [ "$number" -lt 5 ]
11. do
12.     echo -n "$number "
13.     let "number += 1"
14. done
15. # 如果试图运行这个错误的脚本, 就会得到一个错误信息:
16. #+ bad-op.sh: line 10: 5: No such file or directory
17. # 在单中括号结构([ ])中, "<"必须被转义,
18. #+ 即便如此, 比较两个整数仍是错误的。
19.

```

```

20. echo "-----"
21.
22. while [ "$number" \< 5 ]      # 1 2 3 4
23. do                          #
24.     echo -n "$number "        # 看起来好像可以工作，但是 . . .
25.     let "number += 1"         #+ 事实上是比较ASCII码，
26. done                         #+ 而不是整数比较。
27.
28. echo; echo "-----"
29.
30. # 这么做会产生问题。比如：
31.
32. lesser=5
33. greater=105
34.
35. if [ "$greater" \< "$lesser" ]
36. then
37.     echo "$greater is less than $lesser"
38. fi                          # 105 is less than 5
39. # 事实上，在字符串比较中（按照ASCII码的顺序）
40. #+ "105"小于"5"。
41.
42. echo
43.
44. exit 0

```

- 试图用`let`来设置字符串变量。

```

1. let "a = hello, you"
2. echo "$a"    # 0

```

- 有时候在“test”中括号（`[]`）结构里的变量需要被引用起来（双引号）。如果不这么做的话，可能会引起不可预料的结果。请参考[例子 7-6](#)，[例子 16-5](#)，[例子 9-6](#)。
- [为防分隔](#)，用双引号引用一个包含空白符的变量。 有些情况下，

这会产生意想不到的后果。

- 脚本中的命令可能会因为脚本宿主不具备相应的运行权限而导致运行失败。如果用户在命令行中不能调用这个命令的话，那么即使把它放到脚本中来运行，也还是会失败。这时可以通过修改命令的属性来解决这个问题，有时候甚至要给它设置suid位(当然，要以root身份来设置)。
- 试图使用-作为作为重定向操作符（事实上它不是），通常都会导致令人不快的结果。

```
1. command1 2> - | command2
2. # 试图将command1的错误输出重定向到一个管道中 . . .
3. # . . . 不会工作。
4.
5. command1 2>& - | command2 # 也没效果。
6.
7. 感谢, S.C.
```

- 使用Bash 2.0或更高版本的功能，可以在产生错误信息的时候，引发修复动作。但是比较老的Linux机器默认安装的可能是Bash 1.XX。

```
1. #!/bin/bash
2.
3. minimum_version=2
4. # 因为Chet Ramey经常给Bash添加一些新的特征,
5. # 所以你最好将$minimum_version设置为2.XX, 3.XX, 或是其他你认为比较合适的
   值。
6. E_BAD_VERSION=80
7.
8. if [ "$BASH_VERSION" \< "$minimum_version" ]
9. then
10.   echo "This script works only with Bash, version $minimum or
      greater."
```

```

11.     echo "Upgrade strongly recommended."
12.     exit $E_BAD_VERSION
13. fi
14.
15. ...

```

- 在非Linux机器上的Bourne shell脚本(`#!/bin/sh`)中使用Bash特有的功能，可能会引起不可预料的行为。Linux系统通常都会把bash别名化为sh，但是在一般的UNIX机器上却不一定会这么做。
- 使用Bash未文档化的特征，将是一种危险的举动。本书之前的几个版本就依赖一个这种“特征”，下面说明一下这个“特征”，虽然exit或return所能返回的最大正值为255，但是并没有限制我们使用负整数。不幸的是，Bash 2.05b之后的版本，这个漏洞消失了。请参考例子 24-9。
- 在某些情况下，会返回一个误导性的退出状态。设置一个函数内的局部变量或分配一个算术值给一个变量时，就有可能发生这种情况。
- 算术表达式的退出状态不等同于一个错误代码。

```

1. var=1 && ((--var)) && echo $var
2. #          ^^^^^^^^^^ 在这里，这个与列表返回错误代码1而终止。
3. #          不会打印$var的值！
4. echo $?    # 1

```

- 一个带有DOS风格换行符(`\r\n`)的脚本将会运行失败，因为`#!/bin/bash\r\n`是不合法的，与我们所期望的`#!/bin/bash\n`不同，解决办法就是将这个脚本转换为UNIX风格的换行符。

```

1.  #!/bin/bash
2.
3.  echo "Here"
4.
5.  unix2dos $0      # 脚本先将自己改为DOS格式。
6.  chmod 755 $0     # 更改可执行权限。
7.                  # 'unix2dos'会删除可执行权限
8.
9.  ./$0             # 脚本尝试再次运行自己。
10.                 # 但它作为一个DOS文件，已经不能运行了。
11.
12.  echo "There"
13.
14.  exit 0

```

- 以`#!/bin/sh`开头的Bash脚本，不能在完整的Bash兼容模式下运行。某些Bash特定的功能可能会被禁用。如果脚本需要完整的访问所有Bash专有扩展，那么它需要使用`#!/bin/bash`作为开头。
- 如果在here document中，结尾的`limit string`之前加上空白字符的话，将会导致脚本的异常行为。
- 在一个输出被捕获的函数中放置了不止一个echo语句。

```

1.  add2 ()
2.  {
3.      echo "Whatever ... "  # 删掉zhehan
4.      let "retval = $1 + $2"
5.      echo $retval
6.  }
7.
8.      num1=12
9.      num2=43
10.     echo "Sum of $num1 and $num2 = $(add2 $num1 $num2)"
11.
12.  #   Sum of 12 and 43 = Whatever ...

```

```

13. # 55
14.
15. # 这些echo连在一起了。

```

这是行不通的。

- 脚本不能将变量export到它的父进程(即调用这个脚本的shell)，或父进程的环境中。就好比我们在生物学中所学到的那样，子进程只会继承父进程，反过来则不行。

```

1. WHATEVER=/home/bozo
2. export WHATEVER
3. exit 0

```

```

1. bash$ echo $WHATEVER
2. bash$

```

- 可以确定的是，即使回到命令行提示符，变量\$WHATEVER仍然没有被设置。
- 在子shell中设置和操作变量之后，如果尝试在子shell作用域之外使用同名变量的话，将会产生令人不快的结果。

样例 34-2. 子shell缺陷

```

1. #!/bin/bash
2. # 子shell中的变量缺陷。
3.
4. outer_variable=outer
5. echo
6. echo "outer_variable = $outer_variable"
7. echo
8.
9. (
10. # 开始子shell

```

```

11.
12. echo "outer_variable inside subshell = $outer_variable"
13. inner_variable=inner # Set
14. echo "inner_variable inside subshell = $inner_variable"
15. outer_variable=inner # 会修改全局变量吗？
16. echo "outer_variable inside subshell = $outer_variable"
17.
18. # 如果将变量‘导出’会产生不同的结果么？
19. #     export inner_variable
20. #     export outer_variable
21. # 试试看。
22.
23. # 结束子shell
24. )
25.
26. echo
27. echo "inner_variable outside subshell = $inner_variable" # Unset.
28. echo "outer_variable outside subshell = $outer_variable" #
    Unchanged.
29. echo
30.
31. exit 0
32.
33. # 如果你去掉第19和第20行的注释会怎样？
34. # 会产生不同的结果吗？

```

- 将echo的输出通过管道传递给read命令可能会产生不可预料的结果。在这种情况下，read命令的行为就好像它在子shell中运行一样。可以使用set命令来代替(就好像例子15-18一样)。

样例 34-3. 将echo的输出通过管道传递给read命令

```

1. #!/bin/bash
2. #   badread.sh:
3. #   尝试使用'echo'和'read'命令
4. #+ 非交互的给变量赋值。
5.

```

```

6.  # shopt -s lastpipe
7.
8.  a=aaa
9.  b=bbb
10. c=ccc
11.
12. echo "one two three" | read a b c
13. # 尝试重新给变量a, b, 和c赋值。
14.
15. echo
16. echo "a = $a" # a = aaa
17. echo "b = $b" # b = bbb
18. echo "c = $c" # c = ccc
19. # 重新赋值失败。
20.
21. ### 但如果 . . .
22. ## 去掉第6行的注释:
23. # shopt -s lastpipe
24. ##+ 就能解决这个问题!
25. ### 这是Bash 4.2版本的新特性。
26.
27. # -----
28.
29. # 试试下边这种方法。
30.
31. var=`echo "one two three"`
32. set -- $var
33. a=$1; b=$2; c=$3
34.
35. echo "-----"
36. echo "a = $a" # a = one
37. echo "b = $b" # b = two
38. echo "c = $c" # c = three
39. # 重新赋值成功。
40.
41. # -----
42.
43. # 也请注意, echo到'read'的值只会在子shell中起作用。

```



```

44. # 所以, 变量的值*只*会在子shell中被修改。
45.
46. a=aaa          # 重新开始。
47. b=bbb
48. c=ccc
49.
50. echo; echo
51. echo "one two three" | ( read a b c;
52. echo "Inside subshell: "; echo "a = $a"; echo "b = $b"; echo "c =
    $c" )
53. # a = one
54. # b = two
55. # c = three
56. echo "-----"
57. echo "Outside subshell: "
58. echo "a = $a"  # a = aaa
59. echo "b = $b"  # b = bbb
60. echo "c = $c"  # c = ccc
61. echo
62.
63. exit 0

```

事实上, 也正如Anthony Richardson指出的那样, 通过管道将输出传递到任何循环中, 都会引起类似的问题。

```

1. # 循环的管道问题。
2. # 这个例子由Anthony Richardson编写,
3. #+ 由Wilbert Berendsen补遗。
4.
5.
6. foundone=false
7. find $HOME -type f -atime +30 -size 100k |
8. while true
9. do
10.     read f
11.     echo "$f is over 100KB and has not been accessed in over 30
    days"

```

```

12.     echo "Consider moving the file to archives."
13.     foundone=true
14.     # -----
15.     echo "Subshell level = $BASH_SUBSHELL"
16.     # Subshell level = 1
17.     # 没错，现在是在子shell中运行。
18.     # -----
19. done
20.
21. # 变量foundone在这里肯定是false，
22. #+ 因为它是在子shell中被设置为true的。
23. if [ $foundone = false ]
24. then
25.     echo "No files need archiving."
26. fi
27.
28. # =====现在，下边是正确的方法:=====
29.
30. foundone=false
31. for f in $(find $HOME -type f -atime +30 -size 100k) # 这里没使用管
32. do
33.     echo "$f is over 100KB and has not been accessed in over 30
34.     echo "Consider moving the file to archives."
35.     foundone=true
36. done
37.
38. if [ $foundone = false ]
39. then
40.     echo "No files need archiving."
41. fi
42.
43. # =====这里是另一种方法=====
44.
45. # 将脚本中读取变量的部分放到一个代码块中，
46. #+ 这样一来，它们就能在相同的子shell中共享了。
47. # 感谢，W.B。

```

```

48.
49. find $HOME -type f -atime +30 -size 100k | {
50.     foundone=false
51.     while read f
52.     do
53.         echo "$f is over 100KB and has not been accessed in over 30
    days"
54.         echo "Consider moving the file to archives."
55.         foundone=true
56.     done
57.
58.     if ! $foundone
59.     then
60.         echo "No files need archiving."
61.     fi
62. }

```

- 一个相关的问题：当你尝试将tail -f的stdout通过管道传递给grep时，会产生问题。

```

1. tail -f /var/log/messages | grep "$ERROR_MSG" >> error.log
2. # "error.log"文件将不会写入任何东西。
3. # 正如Samuli Kaipiainen指出的那样，
4. #+ 这一结果是从grep的缓冲区输出的。
5. # 解决的办法就是把"--line-buffered"参数添加到grep中。

```

- 在脚本中使用“suid”命令是非常危险的，因为这会危及系统安全。^{[^suid](#)}
- 使用shell脚本来编写CGI程序是值得商榷的。因为Shell脚本的变量不是“类型安全”的，当CGI被关联的时候，可能会产生令人不快的行为。此外，它还很难抵挡住“破解的考验”。
- Bash不能正确地处理双斜线(//)字符串。
- 在Linux或BSD上编写的Bash脚本，可能需要修改一下，才能使它

们运行在商业的UNIX机器上。这些脚本通常都使用GNU命令和过滤工具，GNU工具通常都比一般的UNIX上的同类工具更加强大。这方面的一个非常明显的例子就是，文本处理工具[tr](#)。

- 遗憾的是，更新Bash本身就会破坏[过去工作完全正常](#)的脚本。让我们回顾一下[使用无正式文件的Bash功能有多危险](#)。

危险正在接近你 —

小心，小心，小心，小心。

许多勇敢的心都在沉睡。

所以一定要小心 —

小心。

—A.J. Lamb and H.W. Petrie

注意事项

36 杂项

- [第36章 杂项](#)

第36章 杂项

Nobody really knows what the Bourne shell's grammar is. Even examination of the source code is little help.

—Tom Duff

目录

- [36.1 交互和非交互shell以及脚本](#)
- [36.2 shell wrappers](#)
- [36.3 测试和比较的其他方法](#)
- [36.4 递归：调用自己的脚本](#)
- [36.5 “彩色”的脚本](#)
- [36.6 优化](#)
- [36.7 其他技巧](#)
- [36.8 安全问题](#)
- [36.9 可移植性问题](#)
- [36.10 Windows系统下的脚本](#)

36.1 交互和非交互shell以及脚本

- 36.1 交互和非交互shell以及脚本

36.1 交互和非交互shell以及脚本

交互shell从tty读取用户输入。shell默认会读取启动文件，显示提示符和打开任务控制等。用户可以和shell交互。

脚本总是运行在非交互的shell上。同样，脚本可以访问它自己的tty，这使得在脚本中依然可以模拟出交互的shell。

```
1.  #!/bin/bash
2.  MY_PROMPT='$ '
3.  while :
4.      do
5.          echo -n "$MY_PROMPT"
6.          read line
7.          eval "$line"
8.      done
9.
10. exit 0
11.
12. # 这个脚本以及以上解释是由Stéphane Chazelas提供的
```

一个要求用户输入的交互脚本，通常会用到read语句（请看例15-3）。不过实际上要复杂一些，一个被用户通过console或者xterm调用的脚本，意味着这个脚本被绑定到了一个tty上。

初始化和启动脚本必须是非交互的，因为在运行过程中不能要求人类的介入。许多管理和系统维护脚本也同样是是非交互的。要求自动运行的重复性任务也是通过非交互脚本实现的。

非交互的脚本可以在后台运行，而交互脚本（在后台运行）则会挂起，

因为要等待永远不可能出现的“输入”。解决这个难题可以使用带有expect命令的脚本，或者将文档嵌入到后台运行的交互脚本中。最简单的例子就是将一个文件重定向到read语句，来提供“输入”。(read variable < file) 这可以创造出一个在交互和非交互两种模式下通用的脚本。

如果脚本需要知道它是否运行在交互模式下，简单的方法就是看提示符变量是否存在，就是\$PS1变量。（如果用户通过提示符输入，那么脚本就需要显示提示符，所以脚本中\$PS1变量会被设置）

```
1. if [ -z $PS1 ] # 是否有提示符（译注：判断脚本是否运行在交互模式下）
2. ### if [ -v PS1 ] # Bash 4.2+ ...
3. then
4.     # 非交互模式
5.     ...
6. else
7.     # 交互模式
8.     ...
9. fi
```

脚本也可以测试\$-变量中是否使用了“i”选项来判读是否运行在交互模式下。

```
1. case $- in
2.     *i*) # 交互shell
3.         ;;
4.     *) # 非交互shell
5.         ;;
6. # (参见 "UNIX F.A.Q.," 1993)
```

John Lange描述了另一种替代方法：使用test -t来测试。

```
1. # 关于终端的测试！
2.
```

```

3.  fd=0    # stdin 标准输入
4.
5.  # 使用test -t测试stdin或者stdout是否是一个终端（如果是则证明该脚本运行于交互模式）。
6.  if [ -t "$fd" ]
7.  then
8.      echo interactive # 译注：交互模式
9.  else
10.     echo non-interactive # 译注：非交互模式
11. fi
12.
13. # 但是John指出：
14. #     if [ -t 0 ] 仅在你本地登录时有效，
15. #     如果通过ssh远程调用就会失效，
16. #     所以还要加上对socket的判断。
17.
18. if [[ -t "$fd" || -p /dev/stdin ]]
19. then
20.     echo interactive # 译注：交互模式
21. else
22.     echo non-interactive # 译注：非交互模式
23. fi

```

笔记

脚本可以使用-i选项或者#!/bin/bash -i的文件头强制进入交互模式执行。这可能导致古怪的脚本行为或者在没有错误的情况下显示错误信息。

36.2 shell wrappers

- 36.2 shell wrappers
 - Example 36-1. shell wrapper
 - Example 36-2. 稍微复杂一点的 shell wrapper
 - Example 36-3. 一个通用的写日志文件的 shell wrapper
 - Example 36-4. 关于awk脚本的 shell wrapper
 - Example 36-5. 另一个关于awk的 shell wrapper
 - Example 36-6. Perl嵌入Bash脚本
 - Example 36-7. Bash和Perl脚本合并
 - Example 36-8. Python嵌入Bash脚本
 - Example 36-9. 会讲话的脚本

36.2 shell wrappers

wrapper是一个包含系统命令和工具的脚本，脚本会把一些参数传递给这些（脚本内的）命令。将一个复杂的命令封装成一个wrapper是为了调用它时比较简单好记，特别在使用sed和awk命令时会这么做。

sed或awk脚本通常在命令行下调用时是sed -e '命令'或者awk '命令'。在Bash脚本中嵌入这些命令会让它们在调用时很简单，并且能够被重用。使用这种方法可以将sed和awk的优势统一起来，比如将sed命令处理的结果通过管道传递给awk继续处理。将这些保存成为一个可执行文件，你可以重复调用它的原始版本或者修改版本，而不用在命令行里反复敲冗长的命令。

Example 36-1. shell wrapper

```

1.  #!/bin/bash
2.
3.  # 这个脚本功能是去除文件中的空白行
4.  # 没有做参数检查
5.  #
6.  # 也许你想添加下面的内容：
7.  #
8.  # E_NOARGS=85
9.  # if [ -z "$1" ]
10. # then
11. #   echo "Usage: `basename $0` target-file"
12. #   exit $E_NOARGS
13. # fi
14.
15. sed -e /^$/d "$1"
16. # 就像这个命令
17. #   sed -e '/^$/d' filename
18. # 通过命令行调用
19.
20. # '-e'意思是后面为编辑命令（这个选项可省略）。
21. # '^'代表行首，'$'代表行尾。
22. # 这个正则表达式表示要匹配出所有行首位没有内容的行，就是空白行。
23. # 是删除命令（译注：就是把刚才选出来的空白行删掉）
24.
25. # 将文件名中的特殊字符和空白进行转译
26.
27. # 这个脚本并不会真正的修改目标文件，如果想对目标文件真正的修改，请将输出重定向
28.
29. exit

```

Example 36-2. 稍微复杂一点的 shell wrapper

```

1.  #!/bin/bash
2.
3.  # subst.sh: 在文件中进行替换字符串的脚本

```

```

4. # 例如 "sh subst.sh Smith Jones letter.txt"
5. # letter.txt 中的所有 Jones 都被替换为 Smith。
6.
7. ARGS=3          # 这个脚本需要三个参数
8. E_BADARGS=85    # 传给脚本的参数数量不正确
9.
10. if [ $# -ne "$ARGS" ]
11. then
12.     echo "Usage: `basename $0` old-pattern new-pattern filename"
13.     exit $E_BADARGS
14. fi
15.
16. old_pattern=$1
17. new_pattern=$2
18.
19. if [ -f "$3" ]
20. then
21.     file_name=$3
22. else
23.     echo "File \"$3\" does not exist."
24.     exit $E_BADARGS
25. fi
26.
27. # -----
28. # 这里是最核心的部分
29. sed -e "s/$old_pattern/$new_pattern/g" $file_name
30. # -----
31.
32. # 's' 是sed中的替换命令
33. # /pattern/调用地址匹配
34. # 'g' 表示要对文件中的所有匹配项目都进行替换操作，而不是仅对第一个这样干。
35. # 如果需要深入了解，请阅读sed命令的相关文档。
36.
37. exit $? # 将这个脚本的输出重定向到一个文件即可记录真正的结果

```

Example 36-3. 一个通用的写日志文件的 shell wrapper

```

1.  #!/bin/bash
2.  #  logging-wrapper.sh
3.  #  一个通用的shell wrapper，在进行操作的同时对操作进行日志记录
4.
5.  DEFAULT_LOGFILE=logfile.txt
6.
7.  # 设置下面两个变量的值
8.  OPERATION=
9.  # 可以是任意操作，比如一个awk脚本或者用管道连接的复杂命令
10.
11. LOGFILE=
12. if [ -z "$LOGFILE" ]
13. then      # 如果没有设置日志文件，则使用默认文件名
14.     LOGFILE="$DEFAULT_LOGFILE"
15. fi
16.
17. # 对于操作命令的参数（可选）
18. OPTIONS="$@"
19.
20.
21. # 日志记录
22. echo "`date` + `whoami` + $OPERATION "$@" ">> $LOGFILE
23. # 进行操作动作
24. exec $OPERATION "$@"
25.
26. # 要在真正执行操作之前写日志
27. # 思考下为什么要先写日志，后操作。

```

Example 36-4. 关于awk脚本的 shell wrapper

```

1.  #!/bin/bash
2.  #  pr-ascii.sh: 打印ASCII码表格
3.
4.  START=33      # 可打印的ASCII码范围（十进制）
5.  END=127       # 不会输出不可打印的ASCII码

```

```

6.
7.  echo "  Decimal    Hex      Character"  # 表头
8.  echo "  - - - - -    - - -    - - - - -"
9.
10. for ((i=START; i<=END; i++))
11. do
12.     echo $i | awk '{printf("   %3d          %2x          %c\n", $1, $1,
13.                          $1)}'
13. # Bash内置的printf命令无法完成下面的操作：（译注：所以这使用awk脚本来实现输出）
14. #     printf "%c" "$i"
15. done
16.
17. exit 0
18.
19.
20. #  Decimal    Hex      Character
21. #  - - - - -    - - -    - - - - -
22. #    33         21        !
23. #    34         22        "
24. #    35         23        #
25. #    36         24        $
26. #
27. #    . . .
28. #
29. #   122         7a        z
30. #   123         7b        {
31. #   124         7c        |
32. #   125         7d        }
33.
34.
35. # 将输出重定向到文件
36. # 或者用管道传递给"more":  sh pr-asc.sh | more

```

Example 36-5. 另一个关于awk的 shell wrapper

```

1.  #!/bin/bash
2.
3.  # 在目标文件中添加一个数字的特殊列
4.  # 十进制浮点数也可以，因为awk可以处理这样的输出。
5.
6.  ARGS=2
7.  E_WRONGARGS=85
8.
9.  if [ $# -ne "$ARGS" ] # Check for proper number of command-line
    args.
10. then
11.     echo "Usage: `basename $0` filename column-number"
12.     exit $E_WRONGARGS
13. fi
14.
15. filename=$1
16. column_number=$2
17.
18. # 将shell脚本的变量传递给awk有点难办。
19. # 第一种方法是用引号将Bash脚本变量在awk脚本中包起来
20. #     '$$BASH_SCRIPT_VAR'
21. #     ^                 ^
22. # 下面的awk脚本就是这么干的。
23. # 详细用法可以查阅awk文档。
24.
25. # 多行的awk脚本可以写成这样
26. # awk '
27. # ...
28. # ...
29. # ...
30. # '
31.
32.
33. # 开始awk脚本
34. # -----
35. awk '
36.
37. { total += $"${column_number}"' # 译注：这就是那个bash脚本变量

```

```

38. }
39. END {
40.     print total
41. }
42.
43. ' "$filename"
44. # -----
45. # 结束awk脚本
46.
47.
48. #   将shell变量传递给awk脚本也许是不安全的
49. #   所以Stephane Chazelas提出了下面的替代方案：
50. #   -----
51. #   awk -v column_number="$column_number" ' # 译注：将shell的值赋给一个awk变量
52. #   { total += $column_number
53. #   }
54. #   END {
55. #       print total
56. #   }' "$filename"
57. #   -----
58.
59.
60. exit 0

```

能满足那些需要瑞士军刀般全能工具的脚本语言，就只有Perl了。Perl集合了sed和awk的能力，并且比C更加精简。它是模块化的并且能支持包括厨房洗碗槽在内的所有面向对象编程所能涉及的事物。短小的Perl脚本可以嵌入shell脚本中，甚至Perl可以完全替代shell脚本。（本书作者对此仍然抱有怀疑）

Example 36-6. Perl嵌入Bash脚本

```

1. #!/bin/bash
2.
3. # shell命令先于Perl脚本执行

```

```

4.  echo "This precedes the embedded Perl script within \"$0\"."
5.  echo
    "=====
6.
7.  perl -e 'print "This line prints from an embedded Perl script.\n";'
8.  # 像sed命令一样, Perl使用'-e'选项
9.
10. echo
    "=====
11. echo "However, the script may also contain shell and system
    commands."
12.
13. exit 0

```

即使能将Bash脚本和Perl脚本合二为一，先执行Bash部分还是Perl部分仍然要取决于调用脚本的方式。

Example 36-7. Bash和Perl脚本合并

```

1.  #!/bin/bash
2.  # bashandperl.sh
3.
4.  echo "Greetings from the Bash part of the script, $0."
5.  # 这里可以写更多的Bash命令
6.
7.  exit
8.  # Bash脚本部分结束
9.
10. # =====
11.
12. #!/usr/bin/perl
13. # 这部分脚本要像下面这样调用
14. #     perl -x bashandperl.sh
15.
16. print "Greetings from the Perl part of the script, $0.\n";
17. # Perl 看起来并不像 "echo" ...
18. # 这里可以写更多的Perl命令

```



```
19.
20. # Perl命令部分结束
```

```
1. bash$ bash bashandperl.sh
2. Greetings from the Bash part of the script.
3.
4. bash$ perl -x bashandperl.sh
5. Greetings from the Perl part of the script.
```

当然还可以用shell wrapper嵌入更多的“外来户”，比如Python或者其他的...

Example 36-8. Python嵌入Bash脚本

```
1. #!/bin/bash
2. # ex56py.sh
3.
4. # shell脚本先于Python脚本执行
5. echo "This precedes the embedded Python script within \"$0.\""
6. echo
   "=====
7.
8. python -c 'print "This line prints from an embedded Python
   script.\n";'
9. # 并不像sed和Perl, Python使用'-c'选项
10. python -c 'k = raw_input( "Hit a key to exit to outer script. " )'
11.
12. echo
   "=====
13. echo "However, the script may also contain shell and system
   commands."
14.
15. exit 0
```

使用脚本封装mplayer或者Google翻译服务器的一些功能，你能做出

给你反馈一些信息的小东西。

Example 36-9. 会讲话的脚本

```

1.  #!/bin/bash
2.  #   参见:
3.  #   http://elinux.org/RPi\_Text\_to\_Speech\_\(Speech\_Synthesis\)
4.
5.  # 为了连接Google翻译服务器, 这个脚本必须连接到互联网才能工作,
6.  # 而且你的计算机上必须装有mplayer。
7.
8.  speak()
9.  {
10.   local IFS=+
11.   # 先调用mplayer, 再连接Google翻译服务器。
12.   /usr/bin/mplayer -ao alsa -really-quiet -noconsolecontrols \
13.   "http://translate.google.com/translate_tts?tl=en&q="$*"
14.   # 可以说话的Google翻译
15.   }
16.
17.  LINES=4
18.
19.  spk=$(tail - $LINES $0) # 同样的结尾
20.  speak "$spk"
21.  exit
22.  # BRowns 很高兴与你谈话。

```

有个有趣的shell wrapper例子是Martin Matusiak的undvd, 为复杂的mencoder工具提供了一个简单易用的命令行接口。另一个例子是Itzchak Rehberg的Ext3Undel, 它为在ext3文件系统上恢复删除的文件提供了一整套工具。

Notes

[1] Linux工具事实上很多是shell wrapper, 比如/usr/bin/pdf2ps, /usr/bin/batch和/usr/bin/xmkmf。

36.3 测试和比较的其他方法

- 36.3 测试和比较的其他方法

36.3 测试和比较的其他方法

对于判断（test命令）来说，[[]]比[]更加合适。同样的，算数运算符（译注：-eq之类的）比(())更有优势。

```

1. a=8
2.
3. # 下面所有这些比较的结果都应该是相等的
4. test "$a" -lt 16 && echo "yes, $a < 16"           # "and list"
5. /bin/test "$a" -lt 16 && echo "yes, $a < 16"
6. [ "$a" -lt 16 ] && echo "yes, $a < 16"
7. [[ $a -lt 16 ]] && echo "yes, $a < 16"           # 为表达式添加
8. (( a < 16 )) && echo "yes, $a < 16"           # [[ ]]和(( ))并不是
   必须的
9.
10. city="New York"
11. # 下面这些结果也是相等的
12. test "$city" \< Paris && echo "Yes, Paris is greater than $city"
13.                               # ASCII字符比较
14. /bin/test "$city" \< Paris && echo "Yes, Paris is greater than
   $city"
15. [ "$city" \< Paris ] && echo "Yes, Paris is greater than $city"
16. [[ $city < Paris ]] && echo "Yes, Paris is greater than $city"
17.                               # 并不需要为$city变量加引号。
18.
19. # 向S.C.致谢

```

36.4 递归：可以调用自己的脚本

- 36.4 递归：可以调用自己的脚本
 - Example 36-10. 可以调用自己的脚本（但没什么实际用途）
 - Example 36-11. 一个有点用的调用自己的脚本
 - Example 36-12. 另一个调用自己的脚本

36.4 递归：可以调用自己的脚本

脚本可以递归的调用自己吗？答案是肯定的。

Example 36-10. 可以调用自己的脚本（但没什么实际用途）

```

1.  #!/bin/bash
2.  # recurse.sh
3.
4.  # 脚本可以调用自己吗？
5.  # 其实是可以的。但这样有什么实际用途吗？
6.  # （请往下看）
7.
8.  RANGE=10
9.  MAXVAL=9
10.
11. i=$RANDOM
12. let "i %= $RANGE" # 在0到$RANGE - 1的范围内产生一个随机数
13.
14. if [ "$i" -lt "$MAXVAL" ]
15. then
16.     echo "i = $i"
17.     ./$0          # 脚本进行递归调用（调用自己）
18. fi               # 每次被调用的脚本做同样的事情，直到$i和$MAXVAL相等。
19.
```

```

20. # 如果使用“while”循环代替“if/then”语句会出问题。请试着解释为什么。
21.
22. exit 0
23.
24. # 笔记：
25. # ----
26. # 这个脚本文件必须有可执行权限。
27. # 即使使用“sh”命令调用，这脚本也可以执行。
28. # 请解释原因。

```

Example 36-11. 一个有点用的调用自己的脚本

```

1. #!/bin/bash
2. # pb.sh: phone book
3.
4. # 用于权限管理的脚本，由Rick Boivie编写。
5. # ABS作者稍有修改
6.
7. MINARGS=1      # 需要至少一个参数
8. DATAFILE=./phonebook
9.                # 当前目录下必须存在名为“phonebook”的数据文件
10. PROGRAMME=$0
11. E_NOARGS=70    # 没有错误
12.
13. if [ $# -lt $MINARGS ]; then
14.     echo "Usage: \"$PROGRAMME\" data-to-look-up"
15.     exit $E_NOARGS
16. fi
17.
18. if [ $# -eq $MINARGS ]; then
19.     grep $1 "$DATAFILE"
20.     # 如果$DATAFILE没有匹配则'grep'命令会报错。
21. else
22.     ( shift; "$PROGRAMME" $* ) | grep $1
23.     # 脚本的递归调用
24. fi
25. exit 0          # 脚本结束

```

```

26.
27. # 下面是一些文件内容
28.
29. # -----
   -----
30. 一个简单的"phonebook"数据文件：
31.
32. John Doe          1555 Main St., Baltimore, MD 21228      (410)
   222-3333
33. Mary Moe          9899 Jones Blvd., Warren, NH 03787      (603)
   898-3232
34. Richard Roe       856 E. 7th St., New York, NY 10009      (212)
   333-4567
35. Sam Roe           956 E. 8th St., New York, NY 10009      (212)
   444-5678
36. Zoe Zenobia       4481 N. Baker St., San Francisco, SF 94338 (415)
   501-1631
37. # -----
   -----
38.
39. $bash pb.sh Roe
40. Richard Roe       856 E. 7th St., New York, NY 10009      (212)
   333-4567
41. Sam Roe           956 E. 8th St., New York, NY 10009      (212)
   444-5678
42.
43. $bash pb.sh Roe Sam
44. Sam Roe           956 E. 8th St., New York, NY 10009      (212)
   444-5678
45.
46. # 当对脚本传入多于一个参数时，脚本只显示包含所有参数的行

```

Example 36-12. 另一个调用自己的脚本

```

1. #!/bin/bash
2. # usrmnt.sh, 由Anthony Richardson编写
3. # 在ABS Guide中用于权限管理

```

```

4.
5. # usage:      usrmnt.sh
6. # description: 想使用挂载设备操作的用户，在/etc/sudoers文件中必须属于
   MNTUSERS组。
7.
8. # -----
9. # 这个脚本会返回加了sudo命令的自身。
10. # 如果一个有权限的用户，则只需要输入：
11. #   usermount /dev/fd0 /mnt/floppy
12. # 而不需要使用下面的方法：
13. #   sudo usermount /dev/fd0 /mnt/floppy
14.
15. # 我对于所有需要sudo的脚本都使用了这个技术，因为我发现这让我感觉非常舒服。
16. # -----
17.
18. # 如果SUDO_COMMAND变量没有被设置，那证明没有使用sudo命令运行。这需要
19. # 再重新运行这个脚本，同时传递用户的ID和组ID...
20.
21. if [ -z "$SUDO_COMMAND" ]
22. then
23.     mntusr=$(id -u) grpusr=$(id -g) sudo $0 $* # 译注：脚本调用自己，并
   且传递参数
24.     exit 0
25. fi
26.
27. # 如果使用了sudo运行，就不会卡在这里了。
28. /bin/mount $* -o uid=$mntusr,gid=$grpusr
29.
30. exit 0
31.
32. # 附加说明：
33. # -----
34.
35. # 1) Linux系统允许/etc/fstab文件中列出的用户挂在移动存储设备。但在服务器上，
   我喜欢让更少的人访问移动存储。我发现使用sudo可以帮我做到。
36.
37. # 2) 我还发现使用sudo比用组权限来实现让人感觉更加舒服。
38.

```



```
39. # 3) 这种方法可以给任何有权限的人使用mount命令，所以要小心处理。  
40. # 你也可以将这种技术用到比如mntfloppy, mntcdrom和mntsamba等脚本上来实现  
    更优雅的挂载管理。
```

过多层次的递归调用会导致脚本的栈空间溢出，引起段错误（`segfault`）。

38 后记

- [38 后记](#)
 - [本章目录](#)

38 后记

本章目录

- [38.1 作者后记](#)
- [38.2 关于作者](#)
- [38.3 从哪里可以获得帮助](#)
- [38.4 用来制作这本书的工具](#)
- [38.5 致谢](#)
- [38.6 免责声明](#)

38.1 作者后记

- 38.1 作者后记

38.1 作者后记

doce ut discas

(Teach, that you yourself may learn.)

我怎么会写这么一本与Bash脚本相关的书？这有一个奇怪的故事。让我们把时间退回到几年前，那时候 我正准备学习shell脚本编程 — 除了阅读一本这方面的好书，还有其他比这更好的学习方法么？我苦苦的寻找一本能够覆盖关于这个主题所有部分内容的书籍。我还希望这本书在讲解那些难懂的概念时，能够做到深入浅出，并且能附以详细的例子，最好这些例子还能有很好的注释。 [1] 事实上，我想要找的是一本完美的书，或者是类似的东西。不幸的是，它根本不存在，如果我想要的话，那我就非得自己 写一本了。正因为如此，所以这本书才会呈现在这里。

这使我想起一个关于疯教授的虚构故事。这个家伙非常的古怪。当他在图书馆，或者在书店，任何地方 都行 — 看到一本书的时候，任何书 — 他都会突发奇想的认为，他也可以写这本书，早就应该写了 — 而且他会写得更好。因此，他会马上冲回家，然后着手开始写书，他甚至将书名都起的和原书的名字差不多。许多年过去，当他去世之后，他写了几千本书，可能Asimov(译者注：美国的一个高产作家)在他面前都会觉得羞愧。这些书可能没有那么好 — 谁知道 — 但是这又有什么关系？这是一个生活在 梦想中的家伙，即使他被梦想所迷惑，所驱使 . . . 但是我还是忍不住有点钦佩这个老笨蛋。

注意事项：

[1] 这真是一种声名狼藉并使人郁闷到死的技术。

38.2 关于作者

- [38.2 关于作者](#)

38.2 关于作者

这家伙到底是谁？

作者没有任何特殊的背景或资格[\[1\]](#)，只有一颗冲动的心，用来写作。[\[2\]](#)

这本书有点偏离他主要的工作范围，[HOW-2 Meet Women: The Shy Man's Guide to Relationships](#)。他还写了另一本书，[Software- Building HOWTO](#)。最近，他正打算编写一些短篇小说：[Dave Dawson Over Berlin \(First Installment\)](#) [Dave Dawson Over Berlin \(Second Installment\)](#)和[Dave Dawson Over Berlin \(Third Installment\)](#)。他还有一些 [Instructables](#)给他的credit。
([here](#),[here](#),[here](#),[here](#),[here](#),[here](#) 和 [here](#))

从1995年成为一个Linux用户以来([Slackware 2.2](#), [kernel 1.2.1](#))，作者已经发表了一些软件包，包括[cruft](#) 一次一密乱码本 (one-time pad)加密工具，[mcalc](#)按揭计算器(mortgage calculator)，软件[judge](#)是Scrabble拼字游戏的自动求解包，软件包[yaw1](#)一起组成猜词表，和[Quacky](#)编程游戏包。他的编程之路是从CDC 3800的机器上编写FORTRAN程序开始的，但是那段日子一点都不值得怀念。

作者和他的妻子，还有他们的狗生活在一个偏远的社区里，他认为人性是脆弱的。[\[3\]](#)

注意事项：

[1] 事实上，他没有任何背景或者特殊资格。他是一个学校丢弃的，没有正式资格或者专业的经验。None.Zero.Nada. 除了ABS Guide这本书，他主要的名气是1958年六月的一次在Colfax Elementary School Field Day的套袋赛跑中获得第一名。

[2] 这种事谁都可以做。但是有些人不能...想要拿到MCSE证书。

[3] 有时候，这看起来就像，他花费了整个人生去藐视传统的智慧以及去否定权威的声音：“嗨，你做不到的！”

38.3 在哪里可以获得帮助

- 38.3 在哪里可以获得帮助

38.3 在哪里可以获得帮助

作者(邮件: thegrendel.abs@gmail.com)不再维护或者更新这份文档。他不会回答关于这本书或者一般脚本主题的任何问题。

如果你的课程作业需要帮助的话, 请阅读相关的章节以及其它的参考书籍。 尽你最大的能力、使用你的智慧和资源去解决问题。 请不要浪费作者的时间。你不会获得任何帮助和同情。[1]

同样的, 友善地拒绝对于作者的恳求, 提供工作机会或者商业机会。 他活的很好, 不需要帮助或者同情。谢谢你。

请注意, 作者不会回答任何来自*Sun/Solaris/Oracle* 或者 *Apple*系统上的脚本问题。这些团队正在和开源社区过不去。*Solaris*或者*Apple*的用户可以直接从客户服务那里去友善的获得关心。

注意事项:

[1] 好吧, 如果你执意坚持, 你可以试试修改[例子 A-44] 来达到你的目的。

38.4 用来制作这本书的工具

- 38.4 用来制作这本书的工具
 - 38.4.1 硬件
 - 38.4.2 软件与排版软件

38.4 用来制作这本书的工具

38.4.1 硬件

一台运行着Red Hat 7.1/7.3的IBM Thinkpad, model 760XL (P166, 104 meg RAM)笔记本. 没错, 它非常慢, 而且还有一个令人胆战心惊的键盘, 但它总比一根铅笔加上一个大写字板强多了.

更新: 已经升级到了770Z Thinkpad (P2-366, 192 meg RAM)笔记本, 并且在上面跑FC3. 有人想捐献 一个新一点的笔记本, 给这个快要饿死的作者么?

更新: 已经升级到T61 Thinkpad, 跑着Mandriva 2011. 不再挨饿了, 但是不太情愿接受捐赠。

38.4.2 软件与排版软件

- i. Bram Moolenaar的强大的SGML软件, [vim](#)文本编辑器.
- ii. [OpenJade](#), 使用DSSSL翻译引擎, 来将SGML文档转换为其他格式的工具.
- iii. [Norman Walsh](#)的DSSSL样式单.

iv. DocBook, The Definitive Guide(译者注：这本书被亲切称为TDG)，这本书由Norman Walsh和 Leonard Mueller编写(O'Reilly, ISBN 1-56592-580-7)。对于任何想要使用Docbook SGML格式编写文档的人来说，这本书到目前为止仍然是一本标准参考手册。(译者注：这本书到现在已经有xml的升级版了。)

38.5 致谢

- 38.5 致谢

38.5 致谢

社区的参与让这个项目有了可能实现。作者深知写这本书，如果没有你们这些社区人的帮助和反馈，是不可能完成的。非常感谢你们。

Philippe Martin(email:feloy@free.fr) 把这个文档的第一版翻译进了 DocBook/SGML。当他不在一个法国小公司上班的时候，他喜欢在GNU/Linux的文档和软件上工作，阅读文学，为了头脑安静而玩音乐，结交朋友。你可能会在法国的某个地方或者在Basque Country 偶遇他。你也可以给他发邮件：feloy@free.fr

Philippe Martin 同时指出传入的\$9位置参数可能使用{bracket}注解。(参考[例子 4-5](#))

Stéphane Chazelas(email:stephane_chazelas@yahoo.fr) 发过来了一长串的修改、添加以及脚本例子。除了作为一个贡献者之外，他实际上担任过一段时间的所谓的文档编辑工作者。非常感谢！

Paulo Marcel Coelho Aragao提供了很多修改，不管是重要的还是次要的，并且贡献了非常多有用的建议。

我这里特别感谢Patrick Callahan, Mike Novak 和 Pal Domokos，他们帮忙找bug、指出不明确之处、在这份文档的0.1版本里提供了一些澄清和改变的`建议`。他们活跃的讨论shell脚本和一些文档问题，这激励着我让这份文档更加可读。

我很感激Jim Van Zandt，他指出了这份文档的0.2版的错误和疏漏之处。他同时贡献了一个非常有建设性的[脚本例子](#)。

非常感谢Jordi Sanfeliu(email:mikaku@fiwix.org) 让本书使用他的不错的tree脚本(例子 A-16),以及Rick Boivie修改了它。

同样的,感谢Michel Charpentier(email:charpov@cs.unh.edu) 让我们使用他的dc 分解脚本(例子16-52)。

赞扬Noah Friedman(email:friedman@prep.ai.mit.edu) 许可了他的字符串函数脚本(例子 A-18)。

Emmanuel Rouat(email: emmanuel.rouat@wanadoo.fr)在命令替换、别名和路径管理上给了一些修改和添加的建议。他同时贡献了一个非常有意思的样例文件—.bashrc (附录M)。

Heiner Steven(email:heiner.steven@odn.de) 友善的让我们使用他的转换脚本, 例子 16-48。他同时提供了许多修改和有帮助的建议。特别感谢。

Rick Boivie 贡献了令人愉快的pb.sh 脚本(例子 36-11), 修改了tree.sh脚本(例子 A-16), 并且提出了mothlypmt.sh脚本的性能改善(例子 16-47)。

Florian Wisser 给我提醒了测试字符串的五点(参考例子 7-6), 及其它的一些事情。

Oleg Philon 关于cut 和 pidof提出了一些建议。

Michael Zick 扩展了空数组的例子, 从而证明了一些令人惊讶的数组特性。他同时贡献了isspammer脚本(例子 16-41 和 例子 A-48)。

Marc-Jano Knopp提出了一些修改和澄清关于DOS的批处理文件。

Hyun Jin Cha发现了文档中的一些排印错误，在做韩语翻译的时候。非常感谢他指出了这些内容。

Andreas Abraham 提出了一长串的排印错误和其它的一些修改。特别感谢！

其它贡献脚本、提出帮助性的意见以及指出错误的有：

Gabor Kiss, Leopold Toetsch, Peter Tillier, Marcus Berglof, Tony Richardson, Nick Drage (script ideas!), Rich Bartell, Jess Thrysoee, Adam Lazur, Bram Moolenaar, Baris Cicek, Greg Keraunen, Keith Matthews, Sandro Magi, Albert Reiner, Dim Segebart, Rory Winston, Lee Bigelow, Wayne Pollock, "jipe," "bojster," "nyal," "Hobbit," "Ender," "Little Monster" (Alexis), "Mark," "Patsie," "vladz," Peggy Russell, Emilio Conti, Ian. D. Allen, Hans-Joerg Diers, Arun Giridhar, Dennis Leeuw, Dan Jacobson, Aurelio Marinho Jargas, Edward Scholtz, Jean Helou, Chris Martin, Lee Maschmeyer, Bruno Haible, Wilbert Berendsen, Sebastien Godard, Björn Eriksson, John MacDonald, John Lange, Joshua Tschida, Troy Engel, Manfred Schwarb, Amit Singh, Bill Gradwohl, E. Choroba, David Lombard, Jason Parker, Steve Parker, Bruce W. Clare, William Park, Vernia Damiano, Mihai Maties, Mark Alexander, Jeremy Impson, Ken Fuchs, Jared Martin, Frank Wang, Sylvain Fourmanoit, Matthew Sage, Matthew Walker, Kenny Stauffer, Filip Moritz, Andrzej Stefanski, Daniel Albers, Jeffrey Haemer, Stefano Palmeri, Nils Radtke, Sigurd Solaas, Serghey Rodin, Jeroen Domburg, Alfredo Pironti, Phil Braham, Bruno de Oliveira Schneider, Stefano Falsetto, Chris Morgan, Walter Dnes, Linc Fessenden, Michael Iatrou, Pharis Monalo, Jesse Gough, Fabian Kreutz, Mark Norman, Harald Koenig, Dan Stromberg, Peter Knowles, Francisco Lobo, Mariusz Gniazdowski, Sebastian Arming, Chetankumar Phulpagare, Benno Schulenberg, Tedman Eng, Jochen DeSmet, Juan Nicolas Ruiz, Oliver Beckstein, Achmed Darwish, Dotan Barak, Richard Neill, Albert Siersema, Omair Eshkenazi, Geoff Lee, Graham Ewart, JuanJo Ciarlante, Cliff Bamford, Nathan Coulter, Ramses Rodriguez Martinez, Evgeniy Ivanov, Craig Barnes, George Dimitriu, Kevin LeBlanc, Antonio Macchi, Tomas Pospisek, David Wheeler, Erik Brandsberg, YongYe, Andreas Kühne, Pádraig Brady, Joseph Steinhauser, and David Lawyer (himself an author of four HOWTOs).

我非常感激Chet Ramey(email:chet@po.cwru.edu)和Brian Fox, 他们写了Bash, 并且把它们编进了优雅的和强力的ksh脚本中。

特别感谢努力工作的[Linux Documentation Project](#) 的志愿者。
LDP 存储了Linux知识和学问，极大程度上让这本书的发行成为了可能。

感谢和感激IBM、Redis Hat、Google、[Free Software Foundation](#)以及所有奋战在让开源软件项目更加自由和开放的一线的人们。

感谢我四年级的老师，Miss Spencer，是她的情感支持让我相信，我可能不是一个完全的loser。

最感谢我的妻子—Anita，感谢她的鼓励、灵感和情感支持。

38.6 免责声明

- 38.6 免责声明

38.6 免责声明

(这是一个LDP 标准免责声明的变体.)

文档的内容不负有任何责任。 你可以自己冒险使用概念、例子和信息。可能存在错误、意外输出以及差错，这些可能会让你丢失数据、损害你的电脑或者产生非自愿的触电，所以请带着适当的小心来做这些事情。对于任何伤害、意外或者其它不好的东西，作者不负有任何责任。

正如发生的那样，你或者你的系统不太可能会遭受坏的影响，或者不可控的小问题。事实上，这本书的主要目的是为了让读者能够分析shell脚本以及确实是否有不可预料的结果。