

目录

第一部分 热身	7	
第1章 为什么使用shell编程?	9	9.4 指定变量的类型: 使用declare或者typeset 143
第2章 带着一个Sha-Bang(#!)出发	11	9.5 变量的间接引用 146
2.1 调用一个脚本	16	9.6 \$RANDOM: 产生随机整数 150
2.2 初步的练习	17	9.7 双圆括号结构 163
第二部分 基础	18	第10章 循环与分支 165
第3章 特殊字符	20	10.1 循环 166
第4章 变量和参数的介绍	41	10.2 嵌套循环 179
4.1 变量替换	42	10.3 循环控制 180
4.2 变量赋值	46	10.4 测试与分支(case与select结构) 185
4.3 Bash变量是不区分类型的	48	
4.4 特殊的变量类型	50	
第5章 引用	55	第11章 内部命令与内建命令 194
5.1 引用变量	57	11.1 作业控制命令 225
5.2 转义	59	
第6章 退出和退出状态码	65	第12章 外部过滤器, 程序和命令 230
第7章 条件判断	68	12.1 基本命令 231
7.1 条件测试结构	69	12.2 复杂命令 237
7.2 文件测试操作符	77	12.3 时间/日期命令 249
7.3 其他比较操作符	80	12.4 文本处理命令 252
7.4 嵌套的if/then条件测试	86	12.5 文件与归档命令 276
7.5 检测你对测试知识的掌握情况	87	12.6 通讯命令 296
第8章 操作符与相关主题	88	12.7 终端控制命令 314
8.1 操作符	89	12.8 数学计算命令 315
8.2 数字常量	97	12.9 混杂命令 328
第三部分 进阶	99	第13章 系统与管理命令 342
第9章 变量重游	101	13.1 分析一个系统脚本 374
9.1 内部变量	102	
9.2 操作字符串	123	
9.2.1 使用awk来处理字符串	130	第14章 命令替换 376
9.2.2 更深入的讨论	130	
9.3 参数替换	131	第15章 算术扩展 384
		第16章 I/O重定向 386
		16.1 使用exec 390
		16.2 代码块重定向 395
		16.3 重定向的应用 401
		第17章 Here Document 403
		17.1 Here String 415
		第18章 休息片刻 419

第四部分 高级主题	420	第32章 脚本编程风格	542
第19章 正则表达式	422	32.1 非官方的Shell脚本编写风格	543
19.1 一份简要的正则表达式介绍	423		
19.2 通配(globbing)	428		
第20章 子shell	430	第33章 杂项	547
第21章 受限shell	434	33.1 交互与非交互式的交互与非交互式的shell和脚本	548
第22章 进程替换	436	33.2 Shell包装	550
第23章 函数	439	33.3 测试和比较: 一种可选的方法	556
23.1 复杂函数和函数复杂性	443	33.4 测试和比较: 一种可选的方法	557
23.2 局部变量	456	33.5 将脚本彩色化	561
23.3 不使用局部变量的递归	460	33.6 优化	578
第24章 别名	463	33.7 各种小技巧	579
第25章 列表结构	466	33.8 安全问题	592
第26章 数组	470	33.9 可移植性问题	593
第27章 /dev和/proc	504	33.10 Windows下的shell脚本	594
27.1 /dev	505		
27.2 /proc	508		
第28章 Zero与Null	514	第34章 杂项	595
第29章 调试	518	34.1 Bash, 版本2	596
第30章 选项	530	34.2 Bash, 版本3	603
第31章 陷阱	532	第35章 后记	606
		35.1 作者后记	607
		35.2 关于作者	608
		35.3 译者后记	609
		35.4 在哪里可以获得帮助	610
		35.5 用来制作这本书的工具	611
		35.6 致谢	612
		35.7 译者致谢	614
		第五部分 参考文献	615

高级Bash脚本编程指南

原书作者: Mendel Cooper

中文译者: 杨春敏 黄毅

中文译本版本: V3.9.1-HTML-2006年5月26日

这本书假定你没有任何关于脚本或一般程序的编程知识, 但是如果你具备相关的知识, 那么你将很容易就能够达到中高级的水平... 所有这些只是*UNIX®*浩瀚知识的一小部分。你可以把本书作为教材, 自学手册, 或者是关于shell脚本技术的文档。书中的练习和样例脚本中的注释将会与读者进行更好的互动, 但是最关键的前提是: 想真正学习脚本编程的唯一途径就是亲自动手编写脚本。

这本书也可作为教材来讲解一般的编程概念。

本文档的最新版本是作为一个归档文件bzzip2-ed, "tar包"来发布的, 其中还包括SGML源代码和编译好的HTML版本。读者可以从作者的主页上下载。pdf版本也可以从作者的主页上下载。查看change log来查看校订历史。

贡献

献给Anita, 我所有动力的源泉!

原书作者致中国读者(英文)

Greetings to Linux users in the great nation of China!

I hope that the Advanced Bash Scripting Guide will help you learn the intricacies of Linux and appreciate its utility. Time spent writing scripts will reward you in increased understanding of the operating system and appreciation of the power at your fingertips.

In past eras China was a shining light when much of the rest of the world lay in darkness. Perhaps a new generation of Chinese computer science scholars can help liberate us from the darkness of the Microsoft monopoly.

The translators of this book deserve a great deal of credit and praise for all the time and effort they have invested in this project. I wish to give special thanks to them.

Mendel Cooper

Author, Advanced Bash Scripting Guide

原书作者致中国读者(译文)

向伟大的中华民族的Linux用户致意!

我希望这本书能够帮助你们学习和理解Linux的复杂之处，并充分认识和使用这些工具。花在编写脚本上的时间不会白费，这会增强你对操作系统的了解，还能够稳步提高你的水平。

在过去时代中，当世界上大多数的地方都处于黑暗的时候，中国发出了耀眼的光芒。或许新一代的中国计算机科学学者们可以帮助我们摆脱黑暗，摆脱微软的垄断。

这本书的译者们在这个项目上投入了很多的时间和精力，他们的行为应该得到很大的赞赏和肯定。我特此感谢他们。

Mendel Cooper

本书作者

本书译者黄毅

毫无疑问，UNIX/Linux最重要的软件之一就是shell，目前最流行的shell被称为Bash(Bourne Again Shell)，几乎所有的Linux和绝大部分的UNIX都可以使用Bash。作为系统与用户之间的交互接口，shell几乎是你在UNIX工作平台上最亲密的朋友，因此，学好shell，是学习Linux/UNIX的开始，并且它会始终伴随你的工作和学习。

shell是如此地重要，但令人惊奇的是，介绍shell的书没有真正令人满意的。所幸的是，我看到了这本被人称为abs的书，这本书介绍了Bash大量的细节和广阔的范围，我遇到的绝大部分的技术问题—无论是我忘记的或是以前没有发现的—都可以在这本书里找到答案。这本使用大量的例子详细地介绍了Bash的语法，各种技巧，调试等等的技术，以循序渐进的学习方式，让你了解Bash的所有特性，在书中还有许多练习可以引导你思考，以得到更深入的知识。无论你是新手还是老手，或是使用其他语言的程序员，我能肯定你能在此书用受益。而本书除了介绍BASH的知识之外，也有许多有用的关于Linux/UNIX的知识和其他shell的介绍。

在看到本书的英文版后，我决定把它翻译出来，在Linuxsir论坛上结识了译者之一杨春敏，我们一起把这本书翻译了出来。

关于版权的问题，英文版的作者Mendel Cooper对英文版的版权做了详细的约定，请参考：Q. 版权。中文版版权由译者杨春敏和黄毅共同所有，在遵守英文版版权相应条款的条件下，欢迎在保留本书译者名字和版权说明以非盈利的方式自由发布此中文版，以盈利目的的所有行为必须联系英文作者和两位中文译者以获得许可。

本书得以成稿, 我(黄毅)要多谢我的女朋友, 本该给予她的时间我用来翻译, 多谢你的理解, 你是一个很棒的女朋友!

本书译者杨春敏

这本书如此的有名, 根本用不着我在这里多说什么. 我只希望编写Bash脚本的朋友们, 在遇到问题之前, 先在此书中查阅一番. 我相信, 如果每个人都能做到这点的话, 那么与Bash脚本相关的论坛, 就会是另外一番景象! 在这里非常感谢我的老婆——常宇. 我俩长期在两地生活, 由于翻译本书, 甚至抢占了我俩打电话的时间. 如果没有老婆的支持, 我想也不会有这份译本.

第一部分

热身

shell是一个命令解释器. 是介于操作系统内核与用户之间的一个绝缘层. 准确地说,它也是能力很强的计算机语言, 一种shell程序, 同时也被称为一种脚本语言. 它是非常容易使用的工具, 它可以通过将系统调用, 公共程序, 工具, 和编译过的二进制程序”粘合”在一起建立应用. 事实上, 所有的UNIX命令和工具再加上公共程序, 对于shell脚本来说,都是可调用的. 如果这些你还觉得不够,那么shell内建命令, 比如条件测试与循环结构, 也会给脚本添加强力的支持和增加灵活性. Shell脚本对于管理系统任务和其它的重复工作的例程来说, 表现的非常好, 根本不需要那些华而不实的成熟紧凑的程序语言.

本章目录

1. [为什么使用shell编程?](#)
 2. [带着一个Sha-Bang\(#!\)出发](#)
-

第1章 为什么使用shell编程?

没有程序语言是完美的.
甚至没有一个唯一最好的语言,
只有对于特定目的,
比较适合和不适合的程序语言.
— Herbert Mayer

对于任何想适当精通一些系统管理知识的人来说, 掌握shell脚本知识都是最基本的, 即使这些人可能并不打算真正的编写一些脚本. 想一下Linux机器的启动过程, 在这个过程中, 必将运行/etc/rc.d目录下的脚本来存储系统配置和建立服务. 详细的理解这些启动脚本对于分析系统的行为是非常重要的, 并且有时候可能必须修改它.

学习如何编写shell脚本并不是一件很困难的事, 因为脚本可以分为很小的块, 并且相对于shell特性的操作和选项¹部分, 只需要学习很小的一部分就可以了. 语法是简单并且直观的, 编写脚本很像是在命令行上把一些相关命令和工具连接起来, 并且只有很少的一部分“规则”需要学习. 绝大部分脚本第一次就可以正常的工作, 而且即使调试一个长一些的脚本也是很直观的.

一个shell脚本是一个类似于“小吃店的(quick and dirty)”方法, 在你使用原型设计一个复杂的应用的时候. 在工程开发的第一阶段, 即使从功能中取得很有限的一个子集放到shell脚本中来完成往往都是非常有用的. 使用这种方法, 程序的结果可以被测试和尝试运行, 并且在处理使用诸如C/C++, Java或者Perl语言编写的最终代码前, 主要的缺陷和陷阱往往就被发现了.

Shell脚本遵循典型的UNIX哲学, 就是把大的复杂的工程分成小规模的子任务, 并且把这些部件和工具组合起来. 许多人认为这种办法更好一些, 至少这种办法比使用那种高\大\全的语言更美, 更愉悦, 更适合解决问题. 比如Perl就是这种能干任何事能适合任何人的语言, 但是代价就是你需要强迫自己使用这种语言来思考解决问题的办法.

什么时候不适合使用Shell脚本

- 资源密集型的任务, 尤其在需要考虑效率时(比如, 排序, hash等等).
- 需要处理大任务的数学操作, 尤其是浮点运算, 精确运算, 或者复杂的算术运算(这种情况一般使用C++或FORTRAN来处理).
- 有跨平台移植需求(一般使用C或Java).
- 复杂的应用, 在必须使用结构化编程的时候(需要变量的类型检查, 函数原型, 等等).
- 至关重要的应用, 比如说为了这个应用, 你需要赌上自己的农场, 甚至赌上你们公司的未来.
- 对于安全有很高要求的任务, 比如你需要一个健壮的系统来防止入侵, 破解, 恶意破坏等等.
- 工程的每个组成部分之间, 需要连锁的依赖性.
- 需要大规模的文件操作(Bash受限于顺序地进行文件访问, 而且只能使用这种笨拙的效率低下的一行接一行的处理方式.).
- 需要多维数组的支持.

¹这些将在内建章节被引用, 这些都是shell的内部特征.

-
- 需要数据结构的支持,比如链表或数组等数据结构.
 - 需要产生或操作图形化界面GUI.
 - 需要直接操作系统硬件.
 - 需要I/O或socket接口.
 - 需要使用库或者遗留下来的旧代码的接口.
 - 个人的, 闭源的应用(shell脚本把代码就放在文本文件中, 全世界都能看到).

如果你的应用符合上边的任意一条, 那么就考虑一下更强大的语言吧-或许是Perl, Tcl, Python, Ruby – 或者是更高层次的编译语言比如C/C++, 或者是Java. 即使如此, 你会发现, 使用shell来原型开发你的应用, 在开发步骤中也是非常有用的.

我们将开始使用Bash, Bash是”Bourne-Again shell”首字母的缩写, 也是Stephen Bourne的经典Bourne shell的一个双关语, (译者: 说实话, 我一直搞不清这个双关语是什么意思, 为什么叫”Bourn-Again shell”, 这其中应该有个什么典故吧, 哪位好心, 告诉我一下^^). 对于所有UNIX上的shell脚本来说, Bash已经成为了事实上的标准了. 同时这本书也覆盖了绝大部分的其他一些shell的原则, 比如Korn Shell, Bash从ksh中继承了一部分特性, ² C Shell和它的变种. (注意: C Shell编程是不被推荐的, 因为一些特定的内在问题, Tom Christiansen在1993年10月上的Usenet post指出了这个问题).

接下来是脚本的一些说明. 在展示shell不同的特征之前, 它可以减轻一些阅读书中例子的负担. 本书中的例子脚本, 都在尽可能的范围内进行了测试, 并且其中的一些将使用在真实的生活. 读者可以运行这些例子脚本(使用scriptname.sh或者scriptname.bash的形式), ³ 并给这些脚本执行权限(chmod u+rx scriptname), 然后执行它们, 看看发生了什么. 如果你没有相应的源代码, 那么就从本书的HTML, pdf, 或者text版本中将这些源代码拷贝出来. 考虑到这些脚本中的内容在我们还没解释它之前就被列在这里, 可能会影响读者的理解, 这就需要读者暂时忽略这些内容.

除非特别注明, 本书作者编写了本书中的绝大部分例子脚本.

²ksh88的许多特性,甚至是一些ksh93的特性都被合并到Bash中了.

³根据惯例, 用户编写的Bourne shell脚本应该在脚本的名字后边加上.sh扩展名. 一些系统脚本, 比如那些在/etc/rc.d中的脚本, 则不遵循这种命名习惯.

第2章 带着一个Sha-Bang(#!)出发

Shell程序设计是1950年的光盘机. . .

— Larry Wall

子章节

1. 调用一个脚本
 2. 初步的练习
-

在一个最简单的例子中, 一个shell脚本其实就是将一堆系统命令列在一个文件中. 它的最基本的用处就是, 在你每次输入这些特定顺序的命令时可以少敲一些字.

例子2-1. 清除: 清除/var/log下的log文件

```
1 # 清除
2 # 当然要使用root身份来运行这个脚本.
3
4 cd /var/log
5 cat /dev/null > messages
6 cat /dev/null > wtmp
7 echo "Logs cleaned up."
```

这根本就没什么稀奇的, 只不过是命令的堆积, 来让从console或者xterm中一个一个的输入命令更方便一些. 好处就是把所有命令都放在一个脚本中, 不用每次都敲它们. 这样的话, 这个脚本就成为了一个工具, 对于特定的应用来说, 这个脚本就很容易被修改或定制.

例子2-2. 清除: 一个改良的清除脚本

```
1#!/bin/bash
2# 一个Bash脚本的正确的开头部分.
3
4# Cleanup, 版本 2
5
6# 当然要使用root身份来运行.
7# 在此处插入代码, 来打印错误消息, 并且在不是root身份的时候退出.
8
9LOG_DIR=/var/log
10# 如果使用变量, 当然比把代码写死的好.
11cd $LOG_DIR
12
13cat /dev/null > messages
14cat /dev/null > wtmp
15
16
17echo "Logs cleaned up."
```

```
18
19 exit # 这个命令是一种正确并且合适的退出脚本的方法.
```

现在,让我们看一下一个真正意义的脚本.而且我们可以走得更远. . .

例子2-3. 清除: 一个增强的和广义的删除logfile的脚本

```
1#!/bin/bash
2# 清除, 版本 3
3
4# 警告:
5# -----
6# 这个脚本有好多特征,
7#+ 这些特征是在后边章节进行解释的.
8# 大概是进行到本书的一半的时候,
9#+ 你就会觉得它没有什么神秘的了.
10
11
12LOG_DIR=/var/log
13ROOT_UID=0      # $UID为0的时候, 用户才具有root用户的权限
14LINES=50        # 默认的保存行数
15E_XCD=66        # 不能修改目录?
16E_NOTROOT=67    # 非root用户将以error退出
17
18
19# 当然要使用root用户来运行.
20if [ "$UID" -ne "$ROOT_UID" ]
21then
22    echo "Must be root to run this script."
23    exit $E_NOTROOT
24fi
25
26if [ -n "$1" ]
27# 测试是否有命令行参数(非空).
28then
29    lines=$1
30else
31    lines=$LINES # 默认, 如果不在命令行中指定.
32fi
33
34
35# Stephane Chazelas 建议使用下边
36#+ 的更好方法来检测命令行参数.
37#+ 但对于这章来说还是有点超前.
38#
39#     E_WRONGARGS=65 # 非数值参数(错误的参数格式)
40#
```

```

41 #      case "$1" in
42 #        "") lines=50;;
43 #        *[!0-9]*) echo "Usage: `basename $0` file-to-cleanup"; \
44 #                           exit $E_WRONGARGS;;
45 #        *) lines=$1;;
46 #      esac
47 #
48 #* 直到"Loops"的章节才会对上边的内容进行详细的描述.
49
50
51 cd $LOG_DIR
52
53 if [ `pwd` != "$LOG_DIR" ] # 或者if[ "$PWD" != "$LOG_DIR" ]
54 # 不在 /var/log中?
55 then
56   echo "Can't change to $LOG_DIR."
57   exit $E_XCD
58 fi # 在处理log file之前,再确认一遍当前目录是否正确.
59
60 # 更有效率的做法是:
61 #
62 # cd /var/log || {
63 #   echo "Cannot change to necessary directory." >&2
64 #   exit $E_XCD;
65 # }
66
67
68 tail -$lines messages > mesg.temp # 保存log file消息的最后部分.
69 mv mesg.temp messages           # 变为新的log目录.
70
71
72 # cat /dev/null > messages
73 #* 不再需要了,使用上边的方法更安全.
74
75 cat /dev/null > wtmp # ': > wtmp' 和 '> wtmp' 具有相同的作用
76 echo "Logs cleaned up."
77
78 exit 0
79 # 退出之前返回0,
80 #+ 返回0表示成功.

```

因为你可能希望将系统log全部消灭, 这个版本留下了log消息最后的部分. 你将不断地找到新的方法来完善这个脚本, 并提高效率.

要注意, 在每个脚本的开头都使用sha-bang (#!), 这意味着告诉你的系统这个文件的执行需要指定一个解释器. #! 实际上是一个2字节的[\[1\]](#)魔法数字, 这是指定一个文件类型的特殊标记,

换句话说, 在这种情况下, 指的就是一个可执行的脚本(键入man magic来获得关于这个迷人话题的更多详细信息). 在sha-bang之后接着是一个路径名. 这个路径名就是解释脚本中命令的解释程序所在的路径, 可能是一个shell, 也可能是一个程序语言, 也可能是一个工具包中的命令程序. 这个解释程序从头开始解释并且执行脚本中的命令(从sha-bang行下边的一行开始), 忽略注释.[\[2\]](#)

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

上边每一个脚本头的行都指定了一个不同的命令解释器, 如果是/bin/sh, 那么就是默认shell(在Linux系统上默认就是bash), 否则的话就是其他解释器.[\[3\]](#)使用#!/bin/sh, 因为大多数的商业UNIX系统上都是以Bourne shell作为默认shell, 这样可以使脚本移植到non-Linux的机器上, 虽然这将会牺牲Bash一些独特的特征. 但是脚本将与POSIX[\[4\]](#)的sh标准相一致.

注意”sha-bang”后边给出的路径名必须是正确的, 否则将会出现一个错误消息– 通常也是”Command not found” – 这将是运行这个脚本时所得到的唯一结果.

当然#!也可以被忽略, 不过这样你的脚本文件就只能是一些命令的集合, 不能够使用shell内建的指令了. 上边第二个例子必须以#!开头, 是因为分配变量了, lines=50, 这就使用了一个shell特有的用法. [\[5\]](#)再次提醒你#!/bin/sh将会调用默认的shell解释器, 在Linux机器上默认是/bin/bash.

★ 这个例子鼓励你使用模块化的方式来编写脚本. 平时也要多注意收集一些比较有代表性的”模版”代码, 这些零碎的代码可能用在你将来编写的脚本中. 最后你就能生成一个很好的可扩展的例程库. 以下边这个脚本为例, 这个脚本用来测试脚本被调用的参数数量是否正确.

```
1 E_WRONG_ARGS=65
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` 是这个脚本的文件名.
9     exit $E_WRONG_ARGS
10 fi
```

大多数情况下, 你需要编写一个脚本来执行一个特定的任务, 在本章中第一个脚本就是一个这样的例子, 然后你会修改它来完成一个不同的, 但比较相似的任务. 使用变量来代替写死(”硬编码(hard-wired)”)的常量, 就是一个很好的习惯, 将重复的代码放到一个函数中, 也是一种好习惯.

注意事项:

1. 那些具有UNIX味道的脚本(基于4.2BSD)需要一个4字节的魔法数字, 在!后边需要一个空格– #! /bin/sh.
2. 脚本中的#!所在的行的最重要的任务就是告诉系统本脚本是使用哪种命令解释器. (sh或者bash). 因为这行是以#作为行的开头, 当命令解释器执行这个脚本的时候, 会把它作为一个注释行. 当然, 在这之前, 这行语句已经完成了它的任务, 就是调用命令解释器.

如果脚本中还包含有其他的#!行, 那么bash将会把它看成是一个一般的注释行.

```
1 #!/bin/bash
2
3 echo "Part 1 of script."
4 a=1
5
6 #!/bin/bash
7 # 这将*不会*启动一个新脚本.
8
9 echo "Part 2 of script."
10 echo $a # Value of $a stays at 1.
```

3. 这里可以玩一些小技巧.

```
1 #!/bin/rm
2 # 自删除脚本.
3
4 # 当你运行这个脚本时, 基本上什么都不会发生. . .
5 # 当然这个文件消失不见了.
6
7 WHATEVER=65
8
9 echo "This line will never print (betcha!)."
10
11 exit $WHATEVER # 不要紧, 脚本是不会在这退出的.
```

当然, 你还可以试试在一个README文件的开头加上一个#!/bin/more, 并让它具有执行权限. 结果将是文档自动列出自己的内容. (一个使用cat命令的here document在这里可能是一个更好的选择, 参见[例子17-3](#)).

4. Portable Operating System Interface(可移植的操作系统接口), 标准化类UNIX操作系统的
一种尝试. POSIX规范可以在[Open Group site](#)中进行查阅.
5. 如果Bash是你的默认shell, 那么脚本的开头也不用非得写上#!. 但是如果你使用不同的
的shell来开启一个脚本的话, 比如tcsh, 那么你就必须需要#!了.

1 调用一个脚本

编写完脚本之后,你可以使用`sh scriptname`, [1] 或者`bash scriptname`来调用这个脚本. (不推荐使用`sh <scriptname`, 因为这禁用了脚本从`stdin`中读数据的功能.) 更方便的方法是让脚本本身就具有可执行权限, 通过`chmod`命令可以修改.

比如:

`chmod 555 scriptname` (允许任何人都具有可读和执行权限) [2]

或者

`chmod +rx scriptname` (允许任何人都具有可读和执行权限)

`chmod u+rx scriptname` (只给脚本的所有者可读和执行权限)

既然脚本已经具有了可执行权限, 现在你可以使用`./scriptname`[3]来测试这个脚本了. 如果这个脚本以一个"sha-bang"行开头, 那么脚本将会调用合适的命令解释器来运行.

最后一步, 在脚本被测试和debug之后, 你可能想把它移动到`/usr/local/bin`下, (当然是以root身份), 来让你的脚本对所有用户都有用. 这样以来, 用户就可以在命令行上简单的输入`scriptname [ENTER]` 就可以运行这个脚本了.

注意事项:

1. 小心: 使用`sh scriptname`来调用脚本的时候将会关闭一些Bash特定的扩展, 脚本可能因此而调用失败.
2. 脚本需要读和可执行的权限, 因为shell需要读这个脚本.
3. 为什么不直接使用`scriptname`来调用脚本? 如果你当前的目录下(`$PWD`) 正好是`scriptname`所在的目录, 为什么它运行不了呢? 失败的原因是出于安全考虑, 当前目录并没有被加在用户的`$PATH`环境变量中. 因此, 在当前目录下调用脚本必须使用`./scriptname`这种形式.

2 初步的练习

1. 系统管理员经常会为了自动化一些常用的任务而编写脚本. 举出几个这种有用的脚本的实例.
2. 编写一个脚本, 显示时间和日期, 列出所有登陆的用户, 然后给出系统的更新时间. 最后这个脚本将会把这些信息保存到一个log file中.

```
1 # This line is a comment.
```

第二部分

基础

内容

- 3. 特殊字符
 - 4. 变量和参数的介绍
 - 5. 引用
 - 6. 退出和退出状态码
 - 7. 条件判断
 - 8. 操作符与相关主题
-

第3章 特殊字符

用在脚本和其他地方的特殊字符

: [井号], 注释..

行首以#(#!是个例外)开头是注释.

```
1 # This line is a comment.
```

注释也可以放在本行命令的后边.

```
1 echo "A comment will follow." # 注释在这里.  
2 # ^ 注意#前边的空白
```

注释也可以放在本行行首空白的后面.

```
1 # A tab precedes this comment.
```

命令是不能放在同一行上注释的后边的. 因为没有办法把注释结束掉, 好让同一行上后边的"代码生效". 只能够另起一行来使用下一个命令.

当然, 在echo中转义的#是不能作为注释的. 同样的, #也可以出现在特定的参数替换结构中, 或者是出现在数字常量表达式中.

```
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 # 这里开始一个注释.  
5  
6 echo ${PATH##*:}      # 参数替换, 不是一个注释.  
7 echo $(( 2#101011 )) # 数制转换, 不是一个注释.  
8  
9 # 感谢, S.C.
```

标准的[引用和转义](#)字符(“ ’ \)可以用来转义#.

某些特定的模式匹配操作也可以使用#.

; : [分号], 命令分隔符..

可以在同一行上写两个或两个以上的命令.

```
1 echo hello; echo there  
2  
3  
4 if [ -x "$filename" ]; then    # 注意: "if"和"then"需要分隔.  
5           # 为什么?  
6   echo "File $filename exists."; cp $filename $filename.bak  
7 else
```

```
8 echo "File $filename not found."; touch $filename  
9 fi; echo "File test complete."
```

注意一下”;”某些情况下需要[转义](#).

;; : [双分号], 终止case选项..

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

. : [点], 作为命令时等价于source命令..

这是一个bash的内建命令 (参见[例子11-21](#)).

. : [点].”点”作为文件名的一部分.

如果点放在文件名的开头的话,那么这个文件将会成为”隐藏”文件, 并且ls命令将不会正常的显示出这个文件.

```
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
```

如果作为目录名的话,一个单独的点代表当前的工作目录,而两个点表示上一级目录.

```
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/
```

点经常会出现文件移动命令的目的参数(目录)的位置上.

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

. : [点].”点”字符匹配.

当用作匹配字符的作用时, 通常都是作为正则表达式的一部分来使用, ”点”用来匹配任何的单个字符.

” : [双引号].部分引用.

STRING”将会阻止(解释)STRING中大部分特殊的字符. 参见??.

' : [单引号].全引用.

'STRING'将会阻止STRING中所有特殊字符的解释. 这是一种比使用”更强烈的形式. 参见??.

, : [逗号].逗号操作符(混杂的操作符).

逗号操作符链接了一系列的算术操作. 虽然里边所有的内容都被运行了,但只有最后一项被返回.

```
1 let "t2 = ((a = 9, 15 / 3))" # Set "a = 9" and "t2 = 15 / 3"
```

: [反斜线].一种对单字符的引用机制.

\X将会”转义”字符X. 这等价于”X”, 也等价于'X'. \通常用来转义”和', 这样双引号和但引号就不会被解释成特殊含义了.

参见??来深入地了解转义符的详细解释.

/ : [斜线].文件名路径分隔符.

分隔文件名不同的部分(比如/home/bozo/projects/Makefile).

也可以用来作为除法[算术操作符](#).

‘ : [斜点?].命令替换.

‘command’结构可以将命令的输出赋值到一个变量中去. 我们在后边的后置引用(backquotes)或后置标记(backticks)中也会讲解.

:: : [冒号].空命令..

等价于”NOP” (no op, 一个什么也不干的命令). 也可以被认为与shell的内建命令true作用相同. ”:”命令是一个bash的内建命令, 它的退出码(exit status)是”true”(0).

```
1 :
2 echo $? # 0
```

死循环:

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

在if/then中的占位符:

```
1 if condition
2 then : # 什么都不做, 引出分支.
3 else
4     take-some-action
5 fi
```

在一个二元命令中提供一个占位符, 具体参见[例子8-2](#), 和默认参数.

```
1 : ${username='whoami'}
2 # ${username='whoami'} 如果没有开头的":"
3 #                         的话, 将会给出一个错误,
#                         除非"username"是一个命令或者内建命令...
```

在here document中提供一个命令所需的占位符参见[例子17-10](#).

使用参数替换来评估字符串变量参见[例子9-15](#).

```
1 : ${HOSTNAME?} ${USER?} ${MAIL?}
2 # 如果一个或多个必要的环境变量没被设置的话,
3 #+ 就打印错误信息.
```

变量扩展/子串替换.

在与>重定向操作符结合使用时, 将会把一个文件清空, 但是并不会修改这个文件的权限. 如果之前这个文件并不存在, 那么就创建这个文件.

```
1 : > data.xxx # 文件"data.xxx"现在被清空了.
2
3 # 与 cat /dev/null >data.xxx 的作用相同
4 # 然而, 这并不会产生一个新的进程, 因为":>"是一个内建命令.
```

参见[例子12-14](#).

在与`>>`重定向操作符结合使用时, 将不会对预先存在的目标文件(`: >> target_file`)产生任何影响. 如果这个文件之前并不存在, 那么就创建它.

► 这只适用于正规文件, 而不适用于管道, 符号连接, 和某些特殊文件.

也可能用来作为注释行, 虽然我们不推荐这么做. 使用`#`来注释的话, 将关闭剩余行的错误检查, 所以可以在注释行中写任何东西. 然而, 使用`:`的话将不会这样.

```
1 : This is a comment that generates an error, ( if [ $x -eq 3] ).
```

`:`还用来在`/etc/passwd`和`$PATH`变量中做分隔符.

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

[! : \[感叹号\].取反操作符.](#)

`!`操作符将会反转命令的退出码的结果, (具体参见[例子6-2](#)). 也会反转测试操作符的意义, 比如修改”等号”(`=`)为”不等号”(`!=`). `!`操作符是Bash的关键字.

在一个不同的上下文中, `!`也会出现在变量的间接引用中.

在另一种上下文中, 如命令行模式下, `!`还能反转bash的历史机制(参见Appendix J). 需要注意的是, 在一个脚本中, 历史机制是被禁用的.

[* : \[星号\].通配符.](#)

*可以用来做文件名匹配(这个东西有个专有名词叫[globbing](#))的”通配符”. 含义是, 可以用来匹配给定目录下的任何文件名.

```
1 bash$ echo *
2 abs-book.sgml add-drive.sh agram.sh alias.sh
```

*也可以用在[正则表达式](#)中, 用来匹配任意个数(包含0个)的字符.

[* : \[星号\].算术操作符.](#)

在算术操作符的上下文中, `*`号表示乘法运算.

如果要做求幂运算, 使用`**`, 这是求幂操作符.

[? : \[问号\].测试操作符.](#)

在一个特定的表达式中, `?`用来测试一个条件的结果.

在一个[双括号结构](#)中, `?`就是C语言的三元操作符. 参见[例子9-31](#).

在参数替换表达式中, `?`用来测试一个变量是否被set了.

[? : \[问号\].通配符.](#)

?在通配([globbing](#))中, 用来做匹配单个字符的”通配符”, 在[正则表达式](#)中, 也是用来表示一个字符.

\$: [美元符号].变量替换.

```

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

```

在一个变量前面加上\$用来引用这个变量的值.

\$: [美元符号].行结束符.

在正则表达式中, "\$"表示行结束符.

\$: [美元符号+ 大括号].参数替换.

\$*,\$@: [?].位置参数.

\$? : [美元+问号].退出状态码变量.

\$?变量保存了一个命令, 一个[函数](#), 或者是脚本本身的[退出状态码](#).

\$\$: [双美元].进程ID变量.

这个[\\$\\$变量](#)保存了它所在脚本的进程ID[\[1\]](#).

() : [括号].命令组.

```

1 (a=hello; echo $a)

```

④ 在括号中的命令列表, 将会作为一个[子shell](#)来运行.

在括号中的变量, 由于是在子shell中, 所以对于脚本剩下的部分是不可用的. 父进程, 也就是脚本本身, 将不能够读取在子进程中创建的变量, 也就是在子shell中创建的变量.

```

1 a=123
2 ( a=321; )
3
4 echo "a = $a"    # a = 123
5 # 在圆括号中a变量, 更像是一个局部变量.

```

初始化数组.

```

1 Array=(element1 element2 element3)

```

XXX,YYY,ZZZ,...

大括号扩展.

```
1 cat {file1,file2,file3} > combined_file
2 # 把file1, file2, file3连接在一起，并且重定向到combined_file中.
3
4 cp file22.{txt,backup}
5 # 拷贝 "file22.txt" 到 "file22.backup" 中
```

一个命令可能会对大括号[2]中的以逗号分割的文件列表起作用. (通配(globbing))将对大括号中的文件名做扩展.

④ 在大括号中, 不允许有空白, 除非这个空白被引用或转义.

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

{ } : [花括号].代码块. .

又被称为内部组, 这个结构事实上创建了一个匿名函数(一个没有名字的函数). 然而, 与”标准”函数不同的是, 在其中声明的变量, 对于脚本其他部分的代码来说还是可见的.

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

```
1 a=123
2 { a=321; }
3 echo "a = $a" # a = 321 (说明在代码块中对变量a所作的修改,
4                 # 影响了外边的变量)
5
6 # 感谢, S.C.
```

下边的代码展示了在大括号结构中代码的I/O 重定向.

例子3-1,I/O 重定向

```
1#!/bin/bash
2# 从/etc/fstab中读行.
3
4File=/etc/fstab
5
6{
7read line1
8read line2
9}< $File
10
11echo "First line in $File is:"
12echo "$line1"
13echo
14echo "Second line in $File is:"
```

```
15 echo "$line2"
16
17 exit 0
18
19 # 现在，你怎么分析每行的分割域？
20 # 小提示：使用awk.
```

例子3-2. 将一个代码块的结果保存到文件

```
1#!/bin/bash
2# rpm-check.sh
3
4# 这个脚本的目的是为了描述，列表，和确定是否可以安装一个rpm包。
5# 在一个文件中保存输出。
6#
7# 这个脚本使用一个代码块来展示。
8
9SUCCESS=0
10E_NOARGS=65
11
12if [ -z "$1" ]
13then
14    echo "Usage: `basename $0` rpm-file"
15    exit $E_NOARGS
16fi
17
18{
19    echo
20    echo "Archive Description:"
21    rpm -qpi $1      # 查询说明.
22    echo
23    echo "Archive Listing:"
24    rpm -qpl $1      # 查询列表.
25    echo
26    rpm -i --test $1  # 查询rpm包是否可以被安装.
27    if [ "$?" -eq $SUCCESS ]
28    then
29        echo "$1 can be installed."
30    else
31        echo "$1 cannot be installed."
32    fi
33    echo
34} > "$1.test"      # 把代码块中的所有输出都重定向到文件中.
35
36echo "Results of rpm test in file $1.test"
37
```

```
38 # 查看rpm的man页来查看rpm的选项.  
39  
40 exit 0
```

④ 与上面所讲到的()中的命令组不同的是, 大括号中的代码块将不会开启一个新的子shell.
[3]

{} \; : [?].路径名.

一般都在find命令中使用. 这不是一个shell内建命令.

⑤ ";"用来结束find命令序列的-exec选项. 它需要被保护以防止被shell所解释.

[] : [方括号].条件测试.

条件测试表达式放在[]中. 值得注意的是[是shell内建test命令的一部分, 并不是/usr/bin/test中的外部命令的一个链接.

[[]] : [双重方括号].测试.

测试表达式放在[[]]中. (shell关键字).

具体参见关于[[...]]结构的讨论.

[] : [方括号].数组元素.

在一个array结构的上下文中, 中括号用来引用数组中每个元素的编号.

```
Array[1]=slot_1  
echo ${Array[1]}
```

[] : [方括号].字符范围.

用作正则表达式的一部分, 方括号描述一个匹配的字符范围).

(()) : [双重圆括号].整数扩展.

扩展并计算在(())中的整数表达式.

请参考关于((...)) 结构的讨论.

> &> >& >> < <> : [箭头 (组合)].

重定向.

scriptname > filename 重定向scriptname的输出到文件filename中. 如果filename存在的话, 那么将会被覆盖.

command &> filename 重定向command的stdout和stderr到filename中.

command >&2 重定向command的stdout到stderr中.

scriptname >> filename 把scriptname的输出追加到文件filename中. 如果filename不存在的话, 将会被创建.

[i]&filename 打开文件filename用来读写, 并且分配[文件描述符](#)i给这个文件. 如果filename不存在, 这个文件将会被创建.

[进程替换](#).

(command)>

<(command)

在一种不同的上下文中, "<"和">"可用来做字符串比较操作.

在另一种上下文中, "<"和">"可用来做整数比较操作. 参见[例子12-9](#).

<< : [双箭头].

用在[here document](#)中的重定向.

<<< : [三箭头].

用在[here string](#)中的重定向.

<, > : [大于, 小于号].

ASCII 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 "this implies nothing about my culinary preferences."
8 else
9   echo "What kind of dictionary are you using, anyhow?"
10 fi
```

\<, \>

正则表达式中的单词边界.

```
1 bash$ grep '\<the\>' myfile
```

| : [竖线].管道..

分析前边命令的输出, 并将输出作为后边命令的输入. 这是一种产生命令链的好方法.

```
1 echo ls -l | sh
```

```
2 # 传递"echo ls -l"的输出到shell中,
3 #+ 与一个简单的"ls -l"结果相同.
4
5
6 cat *.lst | sort | uniq
7 # 合并和排序所有的".lst"文件, 然后删除所有重复的行.
```

管道是进程间通讯的一个典型办法, 将一个进程的stdout放到另一个进程的stdin中. 标准的方法是将一个一般命令的输出, 比如cat或者echo, 传递到一个"过滤命令"(在这个过滤命令中将处理输入)中, 然后得到结果.

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

当然输出的命令也可以传递到脚本中.

```
1#!/bin/bash
2# uppercase.sh : 修改输入, 全部转换为大写.
3
4 tr 'a-z' 'A-Z'
5 # 字符范围必须被""引用起来
6 #+ 来阻止产生单字符的文件名.
7
8 exit 0
```

现在让我们输送ls -l的输出到一个脚本中.

```
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
```

④ 管道中的每个进程的stdout必须被下一个进程作为stdin来读入. 否则, 数据流会阻塞, 并且管道将产生一些非预期的行为.

```
1 cat file1 file2 | ls -l | sort
2 # 从"cat file1 file2"中的输出并没出现.
```

作为子进程的运行的管道, 不能够改变脚本的变量.

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

如果管道中的某个命令产生了一个异常, 并中途失败, 那么这个管道将过早的终止. 这种行为被叫做*broken pipe*, 并且这种状态下将发送一个SIGPIPE信号.

> | : 强制重定向(即使设置了noclobber选项—就是-C选项)

这将强制的覆盖一个现存文件.

|| :或逻辑操作

在一个条件测试结构中,如果条件测试结构两边中的任意一边结果为true的话, ||操作就会返回0(代表执行成功).

& :后台运行命令

一个命令后边跟一个& 表示在后台运行.

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

在一个脚本中,命令和循环都可能运行在后台.

例子3-3. 在后台运行一个循环

```
1#!/bin/bash
2# background-loop.sh
3
4for i in 1 2 3 4 5 6 7 8 9 10          # 第一个循环.
5do
6  echo -n "$i "
7done & # 在后台运行这个循环.
8      # 在第2个循环之后, 将在某些时候执行.
9
10echo # 这个'echo'某些时候将不会显示.
11
12for i in 11 12 13 14 15 16 17 18 19 20    # 第二个循环.
13do
14  echo -n "$i "
15done
16
17echo # 这个'echo'某些时候将不会显示.
18
19# =====
20
21# 期望的输出应该是:
22# 1 2 3 4 5 6 7 8 9 10
23# 11 12 13 14 15 16 17 18 19 20
24
25# 然而实际的结果有可能是:
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2个'echo'没执行, 为什么?)
29
30# 也可能是:
31# 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
32# (第1个'echo'没执行, 为什么?)
```

```

34 # 非常少见的执行结果, 也有可能是:
35 # 11 12 13 1 2 3 4 5 6 7 8 9 10 14 15 16 17 18 19 20
36 # 前台的循环先于后台的执行.
37
38 exit 0
39
40 # Nasimuddin Ansari 建议加一句 sleep 1
41 #+ 在6行和14行的 echo -n "$i" 之后加这句.
42 #+ 为了真正的乐趣.

```

④ 在一个脚本内后台运行一个命令, 有可能造成这个脚本的挂起, 等待一个按键响应. 幸运的是, 我们有针对这个问题的[解决办法](#).

&& :与逻辑操作

[与逻辑操作](#). 在一个[条件测试结构](#)中, 只有在条件测试结构的两边结果都为true的时候, &&操作才会返回0(代表success).

- :[短横].选项, 前缀

在所有的命令内如果想使用选项参数的话, 前边都要加上”-”.

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

ls -al

sort -dfu \$filename

set - \$variable

```

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

```

- :[破折号].选项, 前缀

在所有的命令内如果想使用选项参数的话, 前边都要加上”-”.

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

ls -al

```

sort -dfu $filename
set - $variable

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

```

- :[破折号].用于重定向stdin或stdout

用于重定向stdin或stdout[破折号, 即-].

```

1 (cd /source/directory && tar cf - .)|(cd /dest/directory && tar xpvf -)
2 # 从一个目录移动整个目录树到另一个目录
3 # [感谢Alan Cox <a.cox@swansea.ac.uk>, 作出了部分修改]
4
5 # 1) cd /source/directory 源目录
6 # 2) &&      "与列表": 如果'cd'命令成功了, 那么就执行下边的命令.
7 # 3) tar cf - . 'c'创建一个新文档, 'f'后边跟'-'指定目标文件作为stdout
8 #           '-'后边的'f'(file)选项, 指明作为stdout的目标文件.
9 #           并且在当前目录('..')执行.
10 # 4) |       管道...
11 # 5) ( ... ) 一个子shell
12 # 6) cd /dest/directory 改变当前目录到目标目录.
13 # 7) &&      "与列表", 同上
14 # 8) tar xpvf - 'x'解档, 'p'保证所有权和文件属性,
15 #           'v'发完整消息到stdout,
16 #           'f'后边跟'-'从stdin读取数据.
17 #
18 #           注意:'x' 是一个命令, 'p', 'v', 'f' 是选项.
19 # Whew!
20
21
22
23 # 更优雅的写法应该是:
24 #   cd source/directory

```

```
25 # tar cf - . | (cd .. /dest/directory; tar xpf -)
26 #
27 #     当然也可以这么写:
28 # cp -a /source/directory/* /dest/directory
29 #     或者:
30 # cp -a /source/directory/* /source/directory/.[^.]* /dest/directory
31 #     如果在/source/directory中有隐藏文件的话.
```

```
1 bunzip2 -c linux-2.6.16.tar.bz2 | tar xvf -
2 # --未解压的tar文件-- | --然后把它传递到"tar"中--
3 # 如果 "tar" 不能够正常的处理"bzipped",
4 #+ 这就需要使用管道来执行2个单独的步骤来完成它.
5 # 这个练习的目的是解档"bzipped"的kernel源文件.
```

注意, 在这个上下文中”-”本身并不是一个Bash操作, 而是一个可以被特定的UNIX工具识别的选项, 这些特定的UNIX工具特指那些可以写输出到stdout的工具, 比如tar, cat, 等等.

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

在需要一个文件名的位置, -重定向输出到stdout(有时候会在tar和cf中出现), 或者从stdin接受输入, 而不是从一个文件中接受输入. 这是在管道中使用文件导向(file-oriented)工具来作为过滤器的一种方法.

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

在命令行上单独给出一个file, 会给出一个错误信息.

添加一个”-”将得到一个更有用的结果. 这会使shell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
```

现在命令从stdin中接受了输入, 并分析它.

”-”可以被用来将stdout通过管道传递到其他命令中. 这样就允许使用[在一个文件开头添加几行的技巧](#).

使用diff命令来和另一个文件的某一段进行比较:

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

最后, 来展示一个使用-的tar命令的一个真实的例子.

例子3-4. 备份最后一天所有修改的文件

```
1
2 #!/bin/bash
3
```

```

4 # 在一个"tarball"中(经过tar和gzip处理过的文件)
5 #+ 备份最后24小时当前目录下d所有修改的文件.
6
7 BACKUPFILE=backup-$(date +%m-%d-%Y)
8 # 在备份文件中嵌入时间.
9 # Thanks, Joshua Tschida, for the idea.
10 archive=${1:-$BACKUPFILE}
11 # 如果在命令行中没有指定备份文件的文件名,
12 #+ 那么将默认使用"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 $PWD backed up in archive file \"\$archive.tar.gz\"."
17
18
19 # Stephane Chazelas指出上边代码,
20 #+ 如果在发现太多的文件的时候, 或者是如果文件
21 #+ 名包括空格的时候, 将执行失败.
22
23 # Stephane Chazelas建议使用下边的两种代码之一:
24 # -----
25 #   find . -mtime -1 -type f -print0 | xargs -0 tar rvf "$archive.tar"
26 #     使用gnu版本的"find".
27
28
29 #   find . -mtime -1 -type f -exec tar rvf "$archive.tar" '{}' \;
30 #     对于其他风格的UNIX便于移植, 但是比较慢.
31 # -----
32
33
34 exit 0

```

⑧ 以”-“开头的文件名在使用”-“作为重定向操作符的时候, 可能会产生问题. 应该写一个脚本来检查这个问题, 并给这个文件加上合适的前缀. 比如: ./FILENAME, \$PWD/-FILENAME, 或者\$PATHNAME/-FILENAME.

如果变量以-开头进行命名, 可能也会引起问题.

```

1 var="-n"
2 echo $var
3 # 具有"echo -n"的效果了,这样什么都不会输出的.

```

- :[短横线].先前的工作目录

*cd -*将会回到先前的工作目录. 它使用了\$OLDPWD 环境变量.

* 不要混淆这里所使用的”-“和先前我们所讨论的”-“重定向操作符. 对于”-“的具体解释只能依赖于具体的上下文.

- :[减号].

减号属于[算术操作](#).

= :[等号].

[赋值操作](#).

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

在另一种上下文环境中, ”=”也用来做[字符串比较](#)操作.

+ :[加号].

[加法算术操作](#).

+ :[加号].选项

一个命令或者过滤器的选项标记.

某些命令内建命令使用+来打开特定的选项, 用-来禁用这些特定的选项.

% :[百分号].取模

取模(一次除法的余数)[算术操作](#).

在不同的上下文中, %也是一种模式匹配操作.

~ :[波浪号].home目录

相当于\$HOME内部变量. ~bozo是bozo的home目录, 并且ls ~bozo将列出其中的内容. ~/就是当前用户的home目录, 并且ls ~/将列出其中的内容

```
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

相当于\$PWD内部变量.

~-:先前工作目录

相当于\$OLDPWD内部变量.

=~:正则表达式匹配

这个操作将会在[version 3](#)版本的Bash部分进行讲解.

^:行首

在正则表达式中, “^”表示定位到文本行的行首.

控制字符

修改终端或文本显示的行为. 控制字符以CONTROL + key这种方式进行组合(同时按下). 控制字符也可以使用8进制或16进制表示法来进行表示, 但是前边必须要加上转义符.

控制字符在脚本中不能正常使用.

- Ctl-B

退格(非破坏性的), 就是退格但是不删掉前面的字符.

- Ctl-C

break. 终结一个前台作业.

- Ctl-D

从一个shell中登出(与exit很相像). ”EOF”(文件结束). 这也能从stdin中终止输入. 在console或者在xterm窗口中输入的时候, Ctl-D将删除光标下字符. 当没有字符时, Ctl-D将退出当前会话, 在一个xterm窗口中, 则会产生关闭此窗口的效果.

- Ctl-G

”哔”(beep). 在一些老式的打字机终端上, 它会响一下铃.

- Ctl-H

”退格”(破坏性的), 就是在退格之后, 还要删掉前边的字符.

```
1 #!/bin/bash
2 # Embedding Ctl-H in a string.
3
4 a="^H^H"                      # 两个 Ctl-H's (backspaces).
5 echo "abcdef"                  # abcdef
6 echo -n "abcdef$a"            # abcd f
7 # Space at end ^              ^ 两次退格.
8 echo -n "abcdef$a"            # abcdef
```

```
9 # 结尾没有空格, 没有 backspace 的效果了(why?).  
10 # 结果并不像期望的那样.  
11 echo; echo
```

- Ctl-I

水平制表符.

- Ctl-J

重起一行(换一行并到行首). 在脚本中, 也可以使用8进制表示法-- '\012' 或者16进制表示法-- '\x0a' 来表示.

- Ctl-K

垂直制表符. 当在console或者xterm窗口中输入文本时, Ctl-K将会删除从光标所在处到行为的全部字符. 在脚本中, Ctl-K的行为有些不同, 具体请参见下边的Lee Maschmeyer的例子程序.

- Ctl-L

清屏(清除终端的屏幕显示). 在终端中, 与clear命令的效果相同. 当发送到打印机上时, Ctl-L会让打印机将打印纸卷到最后.

- Ctl-M

回车.

```
1#!/bin/bash  
2# Thank you, Lee Maschmeyer, for this example.  
3  
4read -n 1 -s -p $'Control-M leaves cursor  
5      at beginning of this line. Press Enter. \x0d'  
6      # 当然, '0d'就是二进制的回车.  
7echo >&2 # '-s'参数使得任何输入都不将回显出来.  
8      #+ 所以, 明确的重起一行是必要的.  
9  
10read -n 1 -s -p $'Control-J leaves cursor on next line. \x0a'  
11      # '0a' 等价于Control-J, 换行.  
12echo >&2  
13  
14###  
15  
16read -n 1 -s -p $'And Control-K\x0bgoes straight down.'  
17echo >&2 # Control-K 是垂直制表符.  
18  
19# 关于垂直制表符效果的一个更好的例子见下边:  
20  
21var=$'\x0aThis is the bottom line\x0bThis is the top line\x0a'  
22echo "$var"  
23# 这句与上边的例子使用的是同样的办法, 然而:  
24echo "$var" | col  
25# 这将造成垂直制表符右边的部分比左边部分高.
```

```
26 # 这也解释了为什么我们要在行首和行尾加上一个换行符 --
27 #+ 这样可以避免屏幕显示混乱.
28
29 # Lee Maschmeyer的解释:
30 # -----
31 # 在这里[第一个垂直制表符的例子中] . . .
32 #+ 这个垂直制表符使得还没回车就直接打印下来.
33 # 这只能在那些不能"后退"的设备中才行,
34 #+ 比如说Linux的console.
35 # 垂直制表符的真正意义是向上移, 而不是向下.
36 # 它可以用来让打印机打印上标.
37 # col工具可以模拟垂直制表符的正确行为.
38
39 exit 0
```

- Ctl-Q

恢复(XON). 在一个终端中恢复stdin.

- Ctl-S

挂起(XOFF). 在一个终端中冻结stdin. (使用Ctl-Q可以恢复输入.)

- Ctl-U

删除光标到行首的所有字符. 在某些设置下, 不管光标的所在位置Ctl-U都将删除整行输入.

- Ctl-V

当输入字符时, Ctl-V允许插入控制字符. 比如, 下边的两个例子是等价的:

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

Ctl-V主要用于文本编辑.

- Ctl-W

当在控制台或一个xterm窗口敲入文本时, Ctl-W将会删除当前光标到左边最近一个空格间的全部字符. 在某些设置下, Ctl-W将会删除当前光标到左边第一个非字母或数字之间的全部字符.

- Ctl-Z

暂停前台作业.

空白

用来分隔函数, 命令或变量. 空白包含空格, tab, 空行, 或者是它们之间任意的组合体.[\[4\]](#)在某些上下文中, 比如[变量赋值](#), 空白是不被允许的, 会产生语法错误.

空行不会影响脚本的行为, 因此使用空行可以很好的划分独立的函数段以增加可读性.

特殊变量[\\$IFS](#)用来做一些输入命令的分隔符, 默认情况下是空白.

如果想在字符串或变量中使用空白, 那么应该使用[引用](#).

注意事项

1. PID, 或进程ID, 是分配给运行进程的一个数字. 要想察看所有运行进程的PID可以使用ps命令.
2. The shell does the brace expansion. The command itself acts upon the result of the expansion.
3. 例外: 在pipe中的一个大括号中的代码段可能运行在一个子shell中.

```
1 ls | { read firstline; read secondline; }
2 # 错误. 在大括号中的代码段, 将运行到子shell中,
3 #+ 所以"ls"的输出将不能传递到代码块中.
4 echo "First line is $firstline; second line is $secondline"
5 # 不能工作.
6
7 # 感谢, S.C.
```

4. 一个换行符("重起一行")也被认为是空白符. 这也就解释了为什么一个只包含换行符的空行也被认为是空白.

第4章 变量和参数的介绍

本节目录

1. [变量替换](#)
 2. [变量赋值](#)
 3. [Bash变量是不区分类型的](#)
 4. [特殊的变量类型](#)
-

变量是脚本编程中进行数据表现的一种方法。说白了，变量不过是计算机为了保留数据项，在内存中分配的一个位置或一组位置的标识或名字。

变量既可以出现在算术操作中，也可以出现在字符串分析过程中。

1 变量替换

变量的名字就是变量保存值的地方. 引用变量的值就叫做变量替换.

\$

让我们仔细的区别变量的名字和变量的值. 如果variable1是一个变量的名字, 那么\$variable1就是引用这变量的值, 即这个变量所包含的数据.

```
1 bash$ variable=23
2
3 bash$ echo variable
4 variable
5
6 bash$ echo $variable
7 23
```

当变量”裸体”出现的时候-- 也就是说没有\$前缀的时候-- 那么变量可能存在如下几种情况. 变量被声明或被赋值, 变量被unset, 变量被export, 或者是变量处在一种特殊的情况-- 变量代表一种信号(参见[例子29-5](#)). 变量赋值可以使用=(比如var1=27), 也可以在read命令中或者循环头进行赋值(for var2 in 1 2 3).

被一对双引号(“ ”)括起来的变量替换是不会被阻止的. 所以双引号被称为部分引用, 有时候又被称为”弱引用”. 但是如果使用单引号的话(' '), 那么变量替换就会被禁止了, 变量名只会被解释成字面的意思, 不会发生变量替换. 所以单引号被称为全引用, 有时候也被称为”强引用”. 详细讨论参见[5](#).

注意\$variable事实上只是\$variable的简写形式. 在某些上下文中\$variable可能会引起错误, 这时候你就需要用\$variable了(参见下边的[9.3](#)).

例子4-1. 变量赋值和替换

```
1 #!/bin/bash
2
3 # 变量赋值和替换
4
5 a=375
6 hello=$a
7
8 #-----
9 # 强烈注意, 在赋值的时候, 等号前后一定不要有空格.
10 # 如果出现空格会怎么样?
11
12 # "VARIABLE =value"
13 #           ^
14 #% 脚本将尝试运行一个"VARIABLE"的命令, 带着一个"value"参数.
15
16 # "VARIABLE= value"
17 #           ^
18 #% 脚本将尝试运行一个"value"的命令,
19 #+ 并且带着一个被赋值成""的环境变量"VARIABLE".
```

```
20 #-----
21
22
23 echo hello      # 没有变量引用，只是个hello字符串。
24
25 echo $hello
26 echo ${hello} # 同上。
27
28 echo "$hello"
29 echo "${hello}"
30
31 echo
32
33 hello="A B   C   D"
34 echo $hello    # A B C D
35 echo "$hello" # A B   C   D
36 # 就象你看到的echo $hello 和 echo "$hello" 将给出不同的结果。
37 # -----
38 # 引用一个变量将保留其中的空白，当然，如果是变量替换就不会保留了。
39 # -----
40
41 echo
42
43 echo '$hello' # $hello
44 #   ^       ^
45 # 全引用的作用将会导致"$"被解释为单独的字符，
46 #+ 而不是变量前缀。
47
48 # 注意这两种引用所产生的不同的效果。
49
50
51 hello=      # 设置为空值。
52 echo "\$hello (null value) = $hello"
53 # 注意设置一个变量为null，与unset这个变量，并不是一回事
54 #+ 虽然最终的结果相同(具体见下边)。
55
56 # -----
57
58 # 可以在同一行上设置多个变量，
59 #+ 但是必须以空白进行分隔。
60 # 慎用，这么做会降低可读性，并且不可移植。
61
62 var1=21  var2=22  var3=$V3
63 echo
64 echo "var1=$var1  var2=$var2  var3=$var3"
```

```

65
66 # 在老版本的"sh"上可能会引起问题.
67
68 # -----
69
70 echo; echo
71
72 numbers="one two three"
73 #           ^   ^
74 other_numbers="1 2 3"
75 #           ^   ^
76 # 如果在变量中存在空白,
77 #+ 那么就必须加上引用.
78 # other_numbers=1 2 3          # 给出一个错误消息.
79 echo "numbers = $numbers"
80 echo "other_numbers = $other_numbers"  # other_numbers = 1 2 3
81 # 不过也可以采用将空白转义的方法.
82 mixed_bag=2\ ---\ Whatever
83 #           ^   ^ 在转义符后边的空格(\).
84
85 echo "$mixed_bag"      # 2 --- Whatever
86
87 echo; echo
88
89 echo "uninitialized_variable = $uninitialized_variable"
90 # Uninitialized变量为null(就是没有值).
91 uninitialized_variable=  # 声明, 但是没有初始化这个变量,
92                      #+ 其实和前边设置为空值的作用是一样的.
93 echo "uninitialized_variable = $uninitialized_variable"
94                      # 还是一个空值.
95
96 uninitialized_variable=23      # 赋值.
97 unset uninitialized_variable  # Unset这个变量.
98 echo "uninitialized_variable = $uninitialized_variable"
99                      # 还是一个空值.
100 echo
101
102 exit 0

```

④ 一个未初始化的变量将会是"null"值- 就是未赋值(但并不是代表值是0!). 在给变量赋值之前就使用这个变量通常都会引起问题.

但是在执行算术操作的时候, 仍然有可能使用未初始化过的变量.

```

1 echo "$uninitialized"      # (blank line)
2 let uninitialized += 5"   # Add 5 to it.
3 echo "$uninitialized"     # 5

```

```
4  
5 # 结论:  
6 # 一个未初始化的变量是没有值的,  
7 #+ 但是在做算术操作的时候, 这个未初始化的变量看起来值为0.  
8 # 这是一个未文档化(并且可能不具可移植性)的行为.
```

参见[例子11-22](#).

2 变量赋值

=

赋值操作(前后都不能有空白)

④ 因为=和-eq都可以用做条件测试操作, 所以不要与这里的赋值操作相混淆.

注意: =既可以用做条件测试操作, 也可以用于赋值操作, 这需要视具体的上下文而定.

例子4-2. 简单的变量赋值

```
1 #!/bin/bash
2 # "裸体"变量
3
4 echo
5
6 # 变量什么时候是"裸体"的, 比如前边少了$的时候?
7 # 当它被赋值的时候, 而不是被引用的时候.
8
9 # 赋值
10 a=879
11 echo "The value of \"a\" is $a."
12
13 # 使用'let'赋值
14 let a=16+5
15 echo "The value of \"a\" is now $a."
16
17 echo
18
19 # 在'for'循环中(事实上, 这是一种伪赋值):
20 echo -n "Values of \"a\" in the loop are: "
21 for a in 7 8 9 11
22 do
23     echo -n "$a "
24 done
25
26 echo
27 echo
28
29 # 使用'read'命令进行赋值(这也是一种赋值的类型):
30 echo -n "Enter \"a\" "
31 read a
32 echo "The value of \"a\" is now $a."
33
34 echo
35
36 exit 0
```

例子4-3. 简单和复杂, 两种类型的变量赋值

```
1 #!/bin/bash
2
3 a=23      # 简单的赋值
4 echo $a
5 b=$a
6 echo $b
7
8 # 现在让我们来点小变化(命令替换).
9
10 a='echo Hello;  # 把'echo'命令的结果传给变量'a,
11 echo $a
12 # 注意, 如果在一个#+的命令替换结构中包含一个(!)的话,
13 #+ 那么在命令行下将无法工作.
14 #+ 因为这触发了Bash的"历史机制."
15 # 但是, 在脚本中使用的话, 历史功能是被禁用的, 所以就能够正常的运行.
16
17 a='ls -l'    # 把'ls -l'的结果赋值给'a,
18 echo $a      # 然而, 如果没有引号的话将会删除ls结果中多余的tab和换行符.
19 echo
20 echo "$a"    # 如果加上引号的话, 那么就会保留ls结果中的空白符.
21           # (具体请参阅"引用"的相关章节.)
22
23 exit 0
```

使用\$(...)机制来进行变量赋值(这是一种比后置引用(反引号`更新的一种方法). 事实上这两种方法都是**命令替换**的一种形式.

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

3 Bash变量是不区分类型的

不像其他程序语言一样, Bash并不对变量区分“类型”. 本质上, Bash变量都是字符串. 但是依赖于具体的上下文, Bash也允许比较操作和整数操作. 其中的关键因素就是, 变量中的值是否只有数字.

例子4-4. 整型还是字符串?

```
1 #!/bin/bash
2 # int-or-string.sh: 整型还是字符串?
3
4 a=2334          # 整型.
5 let "a += 1"
6 echo "a = $a"      # a = 2335
7 echo          # 还是整型.
8
9
10 b=${a/23/BB}      # 将"23"替换成"BB".
11          # 这将把变量b从整型变为字符串.
12 echo "b = $b"      # b = BB35
13 declare -i b        # 即使使用declare命令也不会对此有任何帮助.
14 echo "b = $b"      # b = BB35
15
16 let "b += 1"        # BB35 + 1 =
17 echo "b = $b"      # b = 1
18 echo
19
20 c=BB34
21 echo "c = $c"      # c = BB34
22 d=${c/BB/23}      # 将"BB"替换成"23".
23          # 这使得变量$d变为一个整形.
24 echo "d = $d"      # d = 2334
25 let "d += 1"        # 2334 + 1 =
26 echo "d = $d"      # d = 2335
27 echo
28
29 # null变量会如何呢?
30 e=""
31 echo "e = $e"      # e =
32 let "e += 1"        # 算术操作允许一个null变量?
33 echo "e = $e"      # e = 1
34 echo          # null变量将被转换成一个整形变量.
35
36 # 如果没有声明变量会怎样?
37 echo "f = $f"      # f =
38 let "f += 1"        # 算术操作能通过么?
```

```
39 echo "f = $f"          # f = 1
40 echo                      # 未声明的变量将转换成一个整型变量.
41
42
43 # 所以说Bash中的变量都是不区分类型的.
44
45 exit 0
```

不区分变量的类型既是幸运的事情也是悲惨的事情. 它允许你在编写脚本的时候更加的灵活(但是也足够把你搞晕!), 并且可以让你能够更容易的编写代码. 然而, 这也很容易产生错误, 并且让你养成糟糕的编程习惯.

这样的话, 程序员就承担了区分脚本中变量类型的责任. Bash是不会为你区分变量类型的.

4 特殊的变量类型

局部变量

这种变量只有在[代码块](#) 或者中(参见[函数](#)中的[局部变量](#))才可见.

环境变量

这种变量将影响用户接口和shell的行为.

► 在通常情况下, 每个进程都有自己的"环境", 这个环境是由一组变量组成的, 这些变量中存有进程可能需要引用的信息. 在这种情况下, shell与一个一般的进程没什么区别.

每次当一个shell启动时, 它都将创建适合于自己环境变量的shell变量. 更新或者添加一个新的环境变量的话, 这个shell都会立刻更新它自己的环境(译者注: 换句话说, 更改或增加的变量会立即生效), 并且所有的shell子进程(即这个shell所执行的命令) 都会继承这个环境. (译者注: 准确地说, 应该是后继生成的子进程才会继承Shell的新环境变量, 已经运行的子进程并不会得到它的新环境变量).

④ 分配给环境变量的空间是有限的. 创建太多环境变量, 或者给一个环境变量分配太多的空间都会引起错误.

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

(感谢Stephane Chazelas 对这个问题的澄清, 并且提供了上边的例子程序.)

如果一个脚本要设置一个环境变量, 那么需要将这些变量"export"出来, 也就是需要通知到脚本本地的环境. 这是[export](#)命令的功能.

► 一个脚本只能够export变量到这个脚本所产生的子进程, 也就是说只能够对这个脚本所产生的命令和进程起作用. 如果脚本是从命令行中调用的, 那么这个脚本所export的变量是不能影响命令行环境的. 也就是说, 子进程是不能够export变量来影响产生自己的父进程的环境的.

位置参数

从命令行传递到脚本的参数: \$0, \$1, \$2, \$3 . . .

\$0就是脚本文件自身的名字, \$1 是第一个参数, \$2是第二个参数, \$3是第三个参数, 然后是第四个. [1] \$9之后的位置参数就必须用大括号括起来了, 比如, \${10}, \${11}, \${12}.

两个比较特殊的变量\$*和\$@ 表示所有的位置参数.

例子4-5. 位置参数

```
1#!/bin/bash
2
3# 作为用例, 调用这个脚本至少需要10个参数, 比如:
4# ./scriptname 1 2 3 4 5 6 7 8 9 10
5MINPARAMS=10
6
7echo
8
9echo "The name of this script is \"$0\"."
10# 添加./是表示当前目录
```

```

11 echo "The name of this script is \"`basename $0`\"."
12 # 去掉路径名, 剩下文件名, (参见'basename')
13
14 echo
15
16 if [ -n "$1" ]           # 测试变量被引用.
17 then
18   echo "Parameter #1 is $1" # 需要引用才能够转义"#
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 $3"
29 fi
30
31 # ...
32
33
34 if [ -n "${10}" ]  # 大于$9的参数必须用{}括起来.
35 then
36   echo "Parameter #10 is ${10}"
37 fi
38
39 echo "-----"
40 echo "All the command-line parameters are: \"$*\""
41
42 if [ $# -lt "$MINPARAMS" ]
43 then
44   echo
45   echo "This script needs at least $MINPARAMS command-line arguments!"
46 fi
47
48 echo
49
50 exit 0

```

{}标记法提供了一种提取从命令行传递到脚本的最后一个位置参数的简单办法. 但是这种方法同时还需要使用间接引用.

```

1 args=$#          # 位置参数的个数.
2 lastarg=${!args}

```

```

3 # 或:      lastarg=${!#}
4 #          (感谢, Chris Monson.)
5 # 注意, 不能直接使用 lastarg=${!$#} , 这会产生错误.

```

一些脚本可能会依赖于使用不同的调用名字, 来表现出不同的行为. 如果想要达到这种目的, 一般都需要在脚本中检查\$0. 因为脚本只能够有一个真正的文件名, 如果要产生多个名字, 必须使用符号链接. 参见[例子12-2](#).

④ 如果脚本需要一个命令行参数, 而在调用的时候, 这个参数没被提供, 那么这就可能造成给这个参数赋一个null变量, 通常情况下, 这都会产生问题. 一种解决这个问题的办法就是使用添加额外字符的方法, 在使用这个位置参数的变量和位置参数本身的后边全部添加同样的额外字符.

```

1 variable1_=$1_ # 而不是 variable1=$1
2 # 这将阻止报错, 即使在调用时没提供这个位置参数.
3
4 critical_argument01=$variable1_
5
6 # 这个扩展的字符是可以被消除掉的, 就像这样.
7 variable1=${variable1/_/_}
8 # 副作用就是$variable1_多了一个下划线.
9 # 这里使用了参数替换模版的一种形式(后边会有具体的讨论).
10 # (在一个删除动作中, 节省了一个替换模式.)
11
12 # 处理这个问题的一个更简单的办法就是
13 #+ 判断一下这个位置参数是否传递下来了.
14 if [ -z $1 ]
15 then
16     exit $E_MISSING_POS_PARAM
17 fi
18
19
20 # 然而, 象Fabian Kreutz所指出的那样,
21 #+ 上边的方法将可能产生一个意外的副作用.
22 # 参数替换才是更好的方法:
23 #      ${1:-$DefaultVal}
24 # 具体参见"参数替换"的相关章节
25 #+ 在"变量重游"那章.

```

例子4-6. wh, whois节点名字查询

```

1#!/bin/bash
2# ex18.sh
3
4# 是否'whois domain-name'能够找到如下3个服务之一:
5#                   ripe.net, cw.net, radb.net
6
7# 把这个脚本重命名为'wh', 然后放到/usr/local/bin目录下.
8

```

```

9 # 需要符号链接:
10 # ln -s /usr/local/bin/wh /usr/local/bin/wh-ripe
11 # ln -s /usr/local/bin/wh /usr/local/bin/wh-cw
12 # ln -s /usr/local/bin/wh /usr/local/bin/wh-radb
13
14 E_NOARGS=65
15
16
17 if [ -z "$1" ]
18 then
19   echo "Usage: `basename $0` [domain-name]"
20   exit $E_NOARGS
21 fi
22
23 # 检查脚本名字, 然后调用合适的服务.
24 case `basename $0` in      # Or:    case ${0##*/} in
25   "wh"      ) whois $1@whois.ripe.net;;
26   "wh-ripe") whois $1@whois.ripe.net;;
27   "wh-radb") whois $1@whois.radb.net;;
28   "wh-cw"   ) whois $1@whois.cw.net;;
29   *         ) echo "Usage: `basename $0` [domain-name]";;
30 esac
31
32 exit $?

```

shift命令会重新分配位置参数, 其实就是把所有的位置参数都向左移动一个位置. . .

\$1 <--- \$2, \$2 <--- \$3, \$3 <--- \$4, 等等.

原来的\$1就消失了, 但是\$0 (脚本名)是不会改变的. 如果传递了大量的位置参数到脚本中, 那么shift命令允许你访问的位置参数的数量超过10个, 当然{}标记法也提供了这样的功能.

例子4-7. 使用shift命令

```

1#!/bin/bash
2# 使用'shift'来逐步存取所有的位置参数.
3
4# 给脚本命个名, 比如shft,
5#+ 然后给脚本传递一些位置参数, 比如:
6#       ./shft a b c def 23 skidoo
7
8until [ -z "$1" ] # 直到所有的位置参数都被存取完...
9do
10  echo -n "$1 "
11  shift
12done
13
14echo          # 额外的换行.

```

```
15
16 exit 0
```

- ▶ 在将参数传递到函数中的时候, shift命令的工作方式也差不多. 参考[例子33-15](#).

注意事项

1. \$0参数是由调用这个脚本的进程所设置的. 按照约定, 这个参数一般就是脚本的名字. 具体请参考execv的man页.

第5章 引用

本章目录

1. [引用变量](#)
 2. [转义](#)
-

引用的字面意思就是将字符串用双引号括起来。它的作用就是保护字符串中的特殊字符不被shell或者shell脚本重新解释，或者扩展。（我们这里所说的“特殊”指的是一些字符在shell中具有特殊的含义，而不是字符的字面意思，比如通配符-- *。）

```
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
```

在日常的演讲和写作中，当我们“引用”一个短语的时候，这意味着这个短语被区分以示它有特别的含义。但是在Bash脚本中，当我们引用一个字符串的时候，我们区分这个字符串是为了保护它的字面含义。

某些程序和工具能够重新解释或者扩展被引用的特殊字符。引用的一个重要作用就是保护命令行参数不被shell解释，但是还是能够让正在调用的程序来扩展它。

```
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.
```

注意一下未引用的grep [Ff]irst *.txt 在Bash shell下的行为。[\[1\]](#)

引用还可以改掉echo's不换行的“毛病”。

```
1 bash$ echo $(ls -l)
2 total 8 -rw-rw-r-- 1 bozo bozo 130 \ # 未换行
3 Aug 21 12:57 t222.sh -rw-rw-r-- 1 bozo bozo 78 \ # 未换行
4 Aug 21 12:57 t71.sh
5
6
7 bash$ echo "$(ls -l)"
8 total 8
9 -rw-rw-r-- 1 bozo bozo 130 Aug 21 12:57 t222.sh
10 -rw-rw-r-- 1 bozo bozo 78 Aug 21 12:57 t71.sh
```

注意事项

-
1. 除非正好当前工作目录下有一个名字为first的文件. 然而这是引用的另一个原因(感谢, Harald Koenig, 指出这一点).

1 引用变量

在一个双引号中通过直接使用变量名的方法来引用变量, 一般情况下都是没问题的. 这么做将阻止所有在引号中的特殊字符被重新解释– 包括变量名[1] – 但是\$,‘(后置引用), 和\(转义符)除外. [2] 保留\$作为特殊字符的意义是为了能够在双引号中也能够正常的引用变量("\$variable"), 也就是说, 这个变量将被它的值所取代(参见上边的[例子4-1](#)).

使用双引号还能够阻止单词分割(word splitting). [3] 如果一个参数被双引号扩起来的话, 那么这个参数将认为是一个单元, 即使这个参数包含有空白, 那里面的单词也不会被分隔开.

```
1 variable1="a variable containing five words"
2 COMMAND This is $variable1    # 用下面7个参数执行COMMAND命令:
3 # "This" "is" "a" "variable" "containing" "five" "words"
4
5 COMMAND "This is $variable1"  # 用下面1个参数执行COMMAND命令:
6 # "This is a variable containing five words"
7
8
9 variable2=""      # Empty.
10
11 # COMMAND将不带参数执行.
12 COMMAND $variable2 $variable2 $variable2
13
14 # COMMAND将以3个空参数来执行.
15 COMMAND "$variable2" "$variable2" "$variable2"
16
17 # COMMAND将以1个参数来执行(2空格).
18 COMMAND "$variable2 $variable2 $variable2"
19
20 # 感谢, Stephane Chazelas.
```

④ 在echo语句中, 只有在单词分割(word splitting)或者需要保留空白的时候, 才需要把参数用双引号括起来.

例子5-1. echo出一些诡异变量

```
1#!/bin/bash
2# weirdvars.sh: echo出一些诡异变量.
3
4var='[]\\{}\\$\\"'
5echo $var        # '[]\{}$'
6echo "$var"      # '[]\{}$' 和上一句没什么区别.
7
8echo
9
10IFS='\''
11echo $var        # '[] {}$' \ 字符被空白符替换了, 为什么?
12echo "$var"      # '[]\{}$'
```

```
13  
14 # 这个例子由Stephane Chazelas提供.  
15  
16 exit 0
```

单引号(')操作与双引号基本一样,但是不允许引用变量,因为\$的特殊意义被关闭了。在单引号中,任何特殊字符都按照字面的意思进行解释,除了'。所以说单引号("全引用")是一种比双引号("部分引用")更严格的引用方法。

④ 因为即使是转义符(\)在单引号中也是按照字面意思解释的,所以如果想在一对单引号中显示一个单引号是不行的(译者注:因为单引号对是按照就近原则完成的)。

```
1 echo "Why can't I write 's between single quotes"  
2  
3 echo  
4  
5 # 一种绕弯的方法。  
6 echo 'Why can'\''t I write '""'s between single quotes'  
7 # |-----| |-----| |-----|  
8 # 三个被单引号引用的字符串,  
9 # 在这三个字符串之间有一个用转义符转义的单引号,  
10 # 和一个用双引号括起来的单引号。  
11  
12 # 这个例子由Stephane Chazelas所捐赠.
```

注意事项

1. 即使是变量的值也会有副作用的(见下边)
2. 当在命令行中使用时,如果在双引号中包含!"的话,那么会产生一个错误(译者注:比如,echo "hello!")。这是因为感叹号被解释成历史命令了。但是如果在脚本中,就不会存在这个问题,因为在脚本中Bash历史机制是被禁用的。

在双引号中使用\"也可能会出现一些不一致的行为。

```
1 bash$ echo hello\!  
2 hello!  
3  
4 bash$ echo "hello\!"  
5 hello\!  
6  
7 bash$ echo -e x\ty  
8 xty  
9  
10 bash$ echo -e "x\ty"  
11 x      y
```

感谢,Wayne Pollock,指出这个问题。

3. ”单词分割(Word splitting)”,在这种上下文中,意味着将一个字符串分隔成一些不连续的,分离的参数。

2 转义

转义是一种引用单个字符的方法。一个前面放上转义符(\)的字符就是告诉shell这个字符按照字面的意思进行解释，换句话说，就是这个字符失去了它的特殊含义。

④ 在某些特定的命令和工具中，比如echo和sed，转义符往往会起到相反效果- 它反倒可能会引发出这个字符的特殊含义。

特定的转义符的特殊的含义

echo和sed命令中使用

\n	表示新的一行
\r	表示回车
\t	表示水平制表符
\v	表示垂直制表符
\b	表示后退符\a 表示”alert”(蜂鸣或者闪烁)
\0xx	转换为八进制的ASCII码, 等价于0xx

例子5-2. 转义符

```
1 #!/bin/bash
2 # escaped.sh: 转义符
3
4 echo; echo
5
6 echo "\v\v\v\v"      # 逐字的打印\v\v\v\v.
7 # 使用-e选项的'echo'命令来打印转义符。
8 echo "====="
9 echo "VERTICAL TABS"
10 echo -e "\v\v\v\v"   # 打印4个垂直制表符。
11 echo "====="
12
13 echo "QUOTATION MARK"
14 echo -e "\042"        # 打印" (引号, 8进制的ASCII 码就是42).
15 echo "====="
16
17 # 如果使用$'\X'结构, 那-e选项就不必要了.
18 echo; echo "NEWLINE AND BEEP"
19 echo $'\n'              # 新行.
20 echo $'\a'              # 警告(蜂鸣).
```

```

22 echo "=====
23 echo "QUOTATION MARKS"
24 # 版本2以后Bash允许使用$'\nnn'结构.
25 # 注意在这里，'\nnn\'是8进制的值.
26 echo $'\t \042 \t'  # 被水平制表符括起来的引号(").
27
28 # 当然，也可以使用16进制的值，使用$'\xhhh' 结构.
29 echo $'\t \x22 \t'  # 被水平制表符括起来的引号(").
30 # 感谢，Greg Keraunen，指出了这点.
31 # 早一点的Bash版本允许，\x022，这种形式.
32 echo "=====
33 echo
34
35
36 # 分配ASCII字符到变量中.
37 # -----
38 quote=$'\042'          # " 被赋值到变量中.
39 echo "$quote This is a quoted string, \[未换行]
40     $quote and this lies outside the quotes."
41
42 echo
43
44 # 变量中的连续的ASCII字符.
45 triple_underline=$'\137\137\137'  # 137是八进制的'_'.
46 echo "$triple_underline UNDERLINE $triple_underline"
47
48 echo
49
50 ABC=$'\101\102\103\010'  # 101, 102, 103是八进制码的A, B, C.
51 echo $ABC
52
53 echo; echo
54
55 escape=$'\033'          # 033 是八进制码的esc.
56 echo "\"escape\" echoes as $escape"
57 #                               没有变量被输出.
58
59 echo; echo
60
61 exit 0

```

参考[例子34-1](#)，这是关于\$' '字符串扩展结构的一个例子.

\"

表示引号字面的意思.

1	echo "Hello" # Hello
---	--------------------------------

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

\\$

表示\$本身子面的含义(跟在\\$后边的变量名将不能引用变量的值).

\\

表示反斜线字面的意思.

```
1 echo "\\\" # 结果是\
2
3 # 反之 . . .
4
5 echo "\" # 如果从命令行调用的话, 会出现SP2, 也就是2级提示符
6          # (译者注: 提示你命令不全, 在添加一个"就好了.
7          # 如果在脚本中调用的话, 那么会报错.
```

\的行为依赖于它自身是否被转义, 被引用(""), 或者是否出现[命令替换](#)或[here document](#)中.

```
1 # 简单的转义和引用
2 echo \z           # z
3 echo \\z          # \z
4 echo '\z'         # \z
5 echo '\\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 # 这些例子是由Stephane Chazelas所提供的.
```

赋值给变量的字符串的元素也会被转义, 但是不能把一个单独的转义符赋值给变量.

```
1 variable=\
```

```

2 echo "$variable"
3 # 不能正常运行 - 会报错:
4 # test.sh: : command not found
5 # 一个"裸体的"转义符是不能够安全的赋值给变量的.
6 #
7 # 事实上在这里"\\"转义了一个换行符(变成了续航符的含义),
8 #+ 效果就是variable=echo "$variable"
9 #+ 不可用的变量赋值
10
11 variable=\
12 23skidoo
13 echo "$variable"      # 23skidoo
14                      # 这句是可以的, 因为
15                      #+ 第2行是一个可用的变量赋值.
16
17 variable=\
18 #     \^    转义一个空格
19 echo "$variable"      # 显示空格
20
21 variable=\\
22 echo "$variable"      # \
23
24 variable=\\\\
25 echo "$variable"
26 # 不能正常运行 - 报错:
27 # test.sh: \: command not found
28 #
29 # 第一个转义符把第2个\转义了,但是第3个又变成"裸体的"了,
30 #+ 与上边的例子的原因相同.
31
32 variable=\\\\\\
33 echo "$variable"      # \\
34                      # 第2和第4个反斜线被转义了.
35                      # 这是正确的.

```

转义一个空格会阻止命令行参数列表的“单词分割”问题.

```

1 file_list="/bin/cat /bin/gzip /bin/more /usr/bin/less \\未换行
2                 /usr/bin/emacs-20.7"
3 # 列出的文件都作为命令的参数.
4
5 # 加两个文件到参数列表中, 列出所有的文件信息.
6 ls -l /usr/X11R6/bin/xsetroot /sbin/dump $file_list
7
8 echo "-----"
9

```

```
10 # 如果我们将上边的两个空格转义了会产生什么效果?  
11 ls -l /usr/X11R6/bin/xsetroot\ /sbin/dump\ $file_list  
12 # 错误: 因为前3个路径被合并成一个参数传递给了'ls -l'  
13 #       而且两个经过转义的空格组织了参数(单词)分割.
```

转义符也提供续行功能, 也就是编写多行命令的功能. 一般的, 每一个单独行都包含一个不同的命令, 但是每行结尾的转义符都会转义换行符, 这样下一行会与上一行一起形成一个命令序列.

```
1 (cd /source/directory && tar cf - . ) | \  
2 (cd /dest/directory && tar xpvf -)  
3 # 重复Alan Cox的目录数拷贝命令,  
4 # 但是分成两行是为了增加可读性.  
5  
6 # 也可以使用如下方式:  
7 tar cf - --C /source/directory . |  
8 tar xpvf - --C /dest/directory  
9 # 察看下边的注意事项.  
10 # (感谢, Stephane Chazelas.)
```

④ 如果一个脚本以—结束, 管道符, 那么就不用非的加上转义符\了. 但是一个好的编程风格, 还是应该在行尾加上转义符.

```
1 echo "foo  
2 bar"  
3 #foo  
4 #bar  
5  
6 echo  
7  
8 echo 'foo  
9 bar'    # 没什么区别.  
10 #foo  
11 #bar  
12  
13 echo  
14  
15 echo foo\  
16 bar      # 换行符被转义.  
17 #foobar  
18  
19 echo  
20  
21 echo "foo\  
22 bar"    # 与上边一样, \在部分引用中还是被解释为续行符.  
23 #foobar  
24  
25 echo
```

```
26
27 echo 'foo\
28 bar'      # 由于是全引用，所以\没有被解释成续行符.
29 #foo\
30 #bar
31
32 # 由Stephane Chazelas所建议的用例.
```

第6章 退出和退出状态码

...在Bourne shell中有许多黑暗的角落,但是人们也会利用它们.

— Chet Ramey

exit 被用来结束一个脚本,就像在C语言中一样. 它也返回一个值,并且这个值会传递给脚本的父进程,父进程会使用这个值做下一步的处理.

每个命令都会返回一个退出状态码(有时候也被称为返回状态). 成功的命令返回0,而不成功的命令返回非零值,非零值通常都被解释成一个错误码. 行为良好的UNIX命令,程序,和工具都会返回0作为退出码来表示成功,虽然偶尔也会有例外.

同样的,脚本中的函数和脚本本身也会返回退出状态码. 在脚本或者是脚本函数中执行的最后的命令会决定退出状态码. 在脚本中, exit nnn命令将会把nnn退出码传递给shell (nnn必须是十进制数,范围必须是0 - 255).

当脚本以不带参数的exit命令来结束时,脚本的退出状态码就由脚本中最后执行的命令来决定(就是exit之前的命令).

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

不带参数的exit命令与exit \$?的效果是一样的,甚至脚本的结尾不写exit,也与前两者的效果相同.

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

```
1 #!/bin/bash
2
3 COMMAND1
```

```
4  
5 . . .  
6  
7 # 将以最后的命令来决定退出状态码.  
8 COMMAND_LAST
```

\$?保存了最后所执行的命令的退出状态码。当函数返回之后，\$?保存函数中最后所执行的命令的退出状态码。这就是bash对函数“返回值”的处理方法。当一个脚本退出，\$?保存了脚本的退出状态码，这个退出状态码也就是脚本中最后一个执行命令的退出状态码。一般情况下，0表示成功，在范围1 - 255的整数表示错误。

例子6-1. 退出/退出状态码

```
1#!/bin/bash  
2  
3 echo hello  
4 echo $?      # 退出状态为0, 因为命令执行成功.  
5  
6 lskdf      # 无效命令.  
7 echo $?      # 非零的退出状态, 因为命令执行失败.  
8  
9 echo  
10  
11 exit 113    # 返回113退出状态给shell.  
12          # 为了验证这个结果, 可以在脚本结束的地方使用"echo $?".  
13  
14 # 一般的, 'exit 0' 表示成功,  
15 #+ 而一个非零的退出码表示一个错误, 或者是反常的条件.
```

\$?用于测试脚本中的命令结果的时候，往往显得特别有用(见[例子12-32](#)和[例子12-17](#))。

④ !，逻辑“非”操作符，将会反转命令或条件测试的结果，并且这会影响退出状态码。

例子6-2. 反转一个条件的用法!

```
1 true # "true" 是内建命令.  
2 echo "exit status of \"true\" = $"      # 0  
3  
4 ! true  
5 echo "exit status of \"! true\" = $"     # 1  
6 # 注意: "!" 需要一个空格.  
7 #      !true  将导致"command not found"错误  
8 #  
9 # 如果一个命令以'!'开头, 那么会启用Bash的历史机制.  
10  
11 true  
12 !true  
13 # 这次就没有错误了, 也没有反转结果.  
14 # 它只是重复了之前的命令(true).
```

16 | # 感谢 , Stephane Chazelas和Kristopher Newsome.

④ 特定的退出状态码具有保留含义, 所以用户不应该在脚本中指定它.

第7章 条件判断

本章目录

1. [条件测试结构](#)
2. [文件测试操作符](#)
3. [其他比较操作符](#)
4. [嵌套的if/then条件测试](#)
5. [检测你对测试知识的掌握情况](#)

每个完整并且合理的程序语言都具有条件判断的功能，并且可以根据条件测试的结果做下一步的处理。Bash有test命令，各种中括号和圆括号操作，和if/then结构。

1 条件测试结构

- if/then结构用来判断命令列表的退出状态码是否为0 (因为在UNIX惯例, 0表示“成功”), 如果成功的话, 那么就执行接下来的一个或多个命令.
- 有一个专有命令[(左中括号, 特殊字符). 这个命令与test命令等价, 并且出于效率上的考虑, 这是一个内建命令. 这个命令把它的参数作为比较表达式或者作为文件测试, 并且根据比较的结果来返回一个退出状态码(0 表示真, 1表示假).
- 在版本2.02的Bash中, 引入了[[...]]扩展测试命令, 因为这种表现形式可能对某些语言的程序员来说更容易熟悉一些. 注意[[是一个关键字, 并不是一个命令.

Bash把[[\$a -lt \$b]]看作一个单独的元素, 并且返回一个退出状态码.

((...))和let ...结构也能够返回退出状态码, 当它们所测试的算术表达式的结果为非零的时候, 将会返回退出状态码0. 这些算术扩展结构被用来做算术比较.

```
1 let "1<2" returns 0 (as "1<2" expands to "1")
2 (( 0 && 1 )) returns 1 (as "0 && 1" expands to "0")
```

- if命令能够测试任何命令, 并不仅仅是中括号中的条件.

```
1 if cmp a b &> /dev/null # 禁止输出.
2 then echo "Files a and b are identical."
3 else echo "Files a and b differ."
4 fi
5
6 # 非常有用的"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 # "-q" 选项是用来禁止输出的.
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 succeeded."
25 else echo "Command failed."
```

26 fi

- 一个if/then结构可以包含嵌套的比较操作和条件判断操作.

```
1 if echo "Next *if* is part of the comparison for the first *if*."
2
3   if [[ $comparison = "integer" ]]
4     then (( a < b ))
5   else
6     [[ $a < $b ]]
7   fi
8
9 then
10   echo '$a is less than $b'
11 fi
```

谦虚的Stephane Chazelas解释了”if-test”的详细细节.

例子7-1. 什么是真?

```
1 #!/bin/bash
2
3 # 小技巧:
4 # 如果你不能够确定一个特定的条件该如何进行判断,
5 #+ 那么就使用if-test结构.
6
7 echo
8
9 echo "Testing \"0\""
10 if [ 0 ]      # zero
11 then
12   echo "0 is true."
13 else
14   echo "0 is false."
15 fi          # 0 为真.
16
17 echo
18
19 echo "Testing \"1\""
20 if [ 1 ]      # one
21 then
22   echo "1 is true."
23 else
24   echo "1 is false."
25 fi          # 1 为真.
26
27 echo
```

```
29 echo "Testing \\"-1\\"
30 if [ -1 ]      # 负1
31 then
32     echo "-1 is true."
33 else
34     echo "-1 is false."
35 fi          # -1 为真.
36
37 echo
38
39 echo "Testing \\\"NULL\\\""
40 if [ ]        # NULL (空状态)
41 then
42     echo "NULL is true."
43 else
44     echo "NULL is false."
45 fi          # NULL 为假.
46
47 echo
48
49 echo "Testing \\\"xyz\\\""
50 if [ xyz ]    # 字符串
51 then
52     echo "Random string is true."
53 else
54     echo "Random string is false."
55 fi          # 随便的一串字符为真.
56
57 echo
58
59 echo "Testing \\\"$xyz\\\""
60 if [ $xyz ]   # 判断$xyz是否为null, 但是...
61                 # 这只是一个未初始化的变量.
62 then
63     echo "Uninitialized variable is true."
64 else
65     echo "Uninitialized variable is false."
66 fi          # 未定义的初始化为假.
67
68 echo
69
70 echo "Testing \\\"-n \\$xyz\\\""
71 if [ -n "$xyz" ]           # 更加正规的条件检查.
72 then
73     echo "Uninitialized variable is true."
```

```
74 else
75     echo "Uninitialized variable is false."
76 fi          # 未初始化的变量为假.
77
78 echo
79
80
81 xyz=        # 初始化了，但是赋null值.
82
83 echo "Testing \"-n \$xyz\""
84 if [ -n "$xyz" ]
85 then
86     echo "Null variable is true."
87 else
88     echo "Null variable is false."
89 fi          # null变量为假.
90
91
92 echo
93
94
95 # 什么时候"false"为真?
96
97 echo "Testing \"false\""
98 if [ "false" ]           # 看起来"false"只不过是一个字符串而已.
99 then
100    echo "\"false\"" is true." #+ 并且条件判断的结果为真.
101 else
102    echo "\"false\"" is false."
103 fi          # "false" 为真.
104
105 echo
106
107 echo "Testing \"\$false\"" # 再来一个，未初始化的变量.
108 if [ "$false" ]
109 then
110    echo "\"\$false\"" is true."
111 else
112    echo "\"\$false\"" is false."
113 fi          # "$false" 为假.
114          # 现在，我们得到了预期的结果.
115
116 # 如果我们测试一下未初始化的变量"$true"会发生什么呢?
117
118 echo
```

```
119  
120 exit 0
```

练习. 解释上边的[例子7-1](#)的行为.

```
1 if [ condition-true ]  
2 then  
3     command 1  
4     command 2  
5     ...  
6 else  
7     # 可选的(如果不需要可以省去).  
8     # 如果原始的条件判断的结果为假, 那么在这里添加默认的代码块来执行.  
9     command 3  
10    command 4  
11    ...  
12 fi
```

如果if和then在条件判断的同一行上的话, 必须使用分号来结束if表达式. if和then都是[关键字](#). 关键字(或者命令)如果作为表达式的开头, 并且如果想在同一行上再写一个新的表达式的话, 那么必须使用分号来结束上一句表达式.

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

Else if和elif

elif

elif是else if的缩写形式. 作用是在外部的判断结构中再嵌入一个内部的if/then结构.

```
1 if [ condition1 ]  
2 then  
3     command1  
4     command2  
5     command3  
6 elif [ condition2 ]  
7 # 与else if一样  
8 then  
9     command4  
10    command5  
11 else  
12     default-command  
13 fi
```

if test condition-true结构与if [condition-true]完全相同. 就像我们前面所看到的, 左中括号, [, 是调用test命令的标识. 而关闭条件判断用的右中括号,], 在if/test结构中并不是严格必需的, 但是在Bash的新版本中必须要求使用.

► test命令在Bash中是[内建命令](#), 用来测试文件类型, 或者用来比较字符串. 因此, 在Bash脚本中, test命令并不会调用外部的/usr/bin/test中的test命令, 这是sh-utils工具包中的一部分. 同

样的, [也并不会调用/usr/bin/[, 这是/usr/bin/test的符号链接.

```
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
```

例子7-2. test, /usr/bin/test, [], 和/usr/bin/[都是等价命令

```
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"      # 与内建的"test"命令结果相同.
15 then
16   echo "No command-line arguments."
17 else
18   echo "First command-line argument is $1."
19 fi
20
21 echo
22
23 if [ -z "$1" ]                # 与上边的代码块作用相同.
24 #  if [ -z "$1" ]              应该能够运行, 但是...
25 #+ Bash报错, 提示缺少关闭条件测试的右中括号.
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" ]          # 再来一个, 与上边的代码块作用相同.
36 # if /usr/bin/[ -z "$1"         # 能够工作, 但是还是给出一个错误消息.
37 #
38 #                                         # 注意:
39 then                                     在版本3.x的Bash中, 这个bug已经被修正了.
40   echo "No command-line arguments."
41 else
42   echo "First command-line argument is $1."
43 fi
44
45 echo
46
47 exit 0

```

[[]]结构 . 比[]结构更加通用. 这是一个扩展的test命令, 是从ksh88中引进的.

► 在[[和]]之间所有的字符都不会发生文件名扩展或者单词分割, 但是会发生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

```

⑧ 使用[[...]]条件判断结构, 而不是[...], 能够防止脚本中的许多逻辑错误. 比如, &&, ||, <, 和> 操作符能够正常存在于[[]]条件判断结构中, 但是如果出现在[]结构中的话, 会报错.

► 在if后面也不一定非得是test命令或者是用于条件判断的中括号结构([] 或[[]]).

```

1 dir=/home/bozo
2
3 if cd "$dir" 2>/dev/null; then    # "2>/dev/null" 会隐藏错误信息.
4   echo "Now in $dir."
5 else
6   echo "Can't change to $dir."
7 fi

```

"if COMMAND"结构将会返回COMMAND的退出状态码.

与此相似, 在中括号中的条件判断也不一定非得要if不可, 也可以使用[列表结构](#).

```

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."

```

(())结构扩展并计算一个算术表达式的值。如果表达式的结果为0，那么返回的退出状态码为1，或者是“假”。而一个非零值的表达式所返回的退出状态码将为0，或者是“true”。这种情况和先前所讨论的test命令和[]结构的行为正好相反。

例子7-3. 算术测试需要使用(())

```
1 #!/bin/bash
2 # 算术测试。
3
4 # (( ... ))结构可以用来计算并测试算术表达式的结果。
5 # 退出状态将会与[ ... ]结构完全相反！
6
7 (( 0 ))
8 echo "Exit status of \"(( 0 ))\" is $?."          # 1
9
10 (( 1 ))
11 echo "Exit status of \"(( 1 ))\" is $?."         # 0
12
13 (( 5 > 4 ))                                     # 真
14 echo "Exit status of \"(( 5 > 4 ))\" is $?."      # 0
15
16 (( 5 > 9 ))                                     # 假
17 echo "Exit status of \"(( 5 > 9 ))\" is $?."      # 1
18
19 (( 5 - 5 ))                                     # 0
20 echo "Exit status of \"(( 5 - 5 ))\" is $?."      # 1
21
22 (( 5 / 4 ))                                     # 除法也可以。
23 echo "Exit status of \"(( 5 / 4 ))\" is $?."      # 0
24
25 (( 1 / 2 ))                                     # 除法的计算结果 < 1.
26 echo "Exit status of \"(( 1 / 2 ))\" is $?."      # 截取之后的结果为 0.
27                                         # 1
28
29 (( 1 / 0 )) 2>/dev/null                         # 除数为0，非法计算。
30 #           ~~~~~
31 echo "Exit status of \"(( 1 / 0 ))\" is $?."      # 1
32
33 # "2>/dev/null"起了什么作用？
34 # 如果这句被删除会怎样？
35 # 尝试删除这句，然后在运行这个脚本。
36
37 exit 0
```

2 文件测试操作符

如果下面的条件成立将会返回真.

-e 文件存在

-a 文件存在

这个选项的效果与-e相同. 但是它已经被”弃用”了, 并且不鼓励使用.

-f 表示这个文件是一个一般文件(并不是目录或者设备文件)

-s 件大小不为零

-d 表示这是一个目录

-b 表示这是一个块设备(软盘, 光驱, 等等.)

-c 表示这是一个字符设备(键盘, modem, 声卡, 等等.)

-p 这个文件是一个管道

-h 这是一个符号链接

-L 这是一个符号链接

-S 表示这是一个socket

-t 文件(描述符)被关联到一个终端设备上

这个测试选项一般被用来检测脚本中的stdin([-t 0]) 或者stdout([-t 1])是否来自于一个终端.

-r 文件是否具有可读权限(指的是正在运行这个测试命令的用户是否具有读权限)

-w 文件是否具有可写权限(指的是正在运行这个测试命令的用户是否具有写权限)

-x 文件是否具有可执行权限(指的是正在运行这个测试命令的用户是否具有可执行权限)

-g set-group-id(sgid)标记被设置到文件或目录上

如果目录具有sgid标记的话, 那么在这个目录下所创建的文件将属于拥有这个目录的用户组, 而不必是创建这个文件的用户组. 这个特性对于在一个工作组中共享目录非常有用.

-u set-user-id (suid)标记被设置到文件上

如果一个root用户所拥有的二进制可执行文件设置了set-user-id标记位的话, 那么普通用户也会以root权限来运行这个文件. [1] 这对于需要访问系统硬件的执行程序(比如pppd和cdrecord)非常有用. 如果没有suid标志的话, 这些二进制执行程序是不能够被非root用户调用的.

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

对于设置了suid标志的文件, 在它的权限列中将会以s表示.

-k 设置粘贴位

对于”粘贴位”的一般了解, save-text-mode标志是一个文件权限的特殊类型. 如果文件设置了这个标志, 那么这个文件将会被保存到缓存中, 这样可以提高访问速度. [2] 粘贴位如果设置在目录中, 那么它将限制写权限. 对于设置了粘贴位的文件或目录, 在它们的权限标记列中将会显示t.

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

如果用户并不拥有这个设置了粘贴位的目录, 但是他在这个目录下具有写权限, 那么这个用户只能在这个目录下删除自己所拥有的文件. 这将有效的防止用户在一个公共目录中不慎覆盖或者删除别人的文件. 比如说/tmp目录. (当然, 目录的所有者或者root用户可以随意删除或重命名其中的文件.)

-O 判断你是否是文件的拥有者-G 文件的group-id是否与你的相同-N 从文件上一次被读取到现在为止, 文件是否被修改过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是相同文件的硬链接! ”非” – 反转上边所有测试的结果(如果没给出条件, 那么返回真).

例子7-4. 测试那些断掉的链接文件

```
1 #!/bin/bash
2 # broken-link.sh
3 # 由Lee bigelow所编写 <ligelowbee@yahoo.com>
4 # 已经征得作者的授权, 引用到本书.
5
6 #一个纯粹的shell脚本用来找出那些断掉的符号链接文件并且输出它们所指向的文件.
7 #以便于它们可以把输出提供给xargs来进行处理 :)
8 #比如. broken-link.sh /somedir /someotherdir|xargs rm
9 #
10 #下边的方法, 不管怎么说, 都是一种更好的办法:
11 #
12 #find "somedir" -type l -print0|\
13 #xargs -r0 file|\
14 #grep "broken symbolic"|
15 #sed -e 's/^\\|: *broken symbolic.*$/"/g'
16 #
17 #但这不是一个纯粹的bash脚本, 最起码现在不是.
18 #注意: 谨防在/proc文件系统和任何死循环链接中使用!
19 #####
20
21
22 #如果没有参数被传递到脚本中, 那么就使用
23 #当前目录. 否则就是用传递进来的参数作为目录
24 #来搜索.
25 #####
26 [ $# -eq 0 ] && directorys='pwd' || directorys=$@
27
28 #编写函数linkchk用来检查传递进来的目录或文件是否是链接,
29 #并判断这些文件或目录是否存在. 然后打印它们所指向的文件.
30 #如果传递进来的元素包含子目录,
31 #那么把子目录也放到linkcheck函数中处理, 这样就达到了递归的目的.
32 #####
33 linkchk () {
34     for element in $1/*; do
35         [ -h "$element" -a ! -e "$element" ] && echo \"\$element\"
36         [ -d "$element" ] && linkchk $element
37         # 当然, '-h'用来测试符号链接, '-d'用来测试目录.
38         done
39     }
40
41 #把每个传递到脚本的参数都送到linkchk函数中进行处理,
42 #检查是否有可用目录. 如果没有, 那么就打印错误消息和
43 #使用信息.
```

```
44 #####
45 for directory in $directorys; 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 0
```

[例子28-1](#), [例子10-7](#), [例子10-3](#), [例子28-3](#), 和[例子A-1](#) 也会演示文件测试操作的使用过程

注意事项

1. 在将suid标记设置到二进制可执行文件的时候, 一定要小心. 因为这可能会引发安全漏洞.
但是suid标记不会影响shell脚本.
2. 在当代UNIX系统中, 文件中已经不使用粘贴位了, 粘贴位只使用在目录中.

3 其他比较操作符

二元比较操作符用来比较两个变量或数字。注意整数比较与字符串比较的区别。

整数比较

```
1 -eq 等于      if [ "$a" -eq "$b" ]
2 -ne 不等于    if [ "$a" -ne "$b" ]
3 -gt 大于      if [ "$a" -gt "$b" ]
4 -ge 大于等于  if [ "$a" -ge "$b" ]
5 -lt 小于      if [ "$a" -lt "$b" ]
6 -le 小于等于  if [ "$a" -le "$b" ]
7 < 小于(在双括号中使用) (( "$a" < "$b" ))
8 <= 小于等于(在双括号中使用) (( "$a" <= "$b" ))
9 > 大于(在双括号中使用) (( "$a" > "$b" ))
10 >= 大于等于(在双括号中使用) (( "$a" >= "$b" ))
```

字符串比较

=

等于

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

==

等于

```
if [ "$a" == "$b" ] 与 = 等价.
```

► == 比较操作符在双中括号对和单中括号对中的行为是不同的。

```
1 [[ $a == z* ]]      # 如果 $a 以 "z" 开头(模式匹配)那么结果将为真
2 [[ $a == "z*" ]]    # 如果 $a 与 z* 相等(就是字面意思完全一样), 那么结果为真.
3
4
5 [ $a == z* ]        # 文件扩展匹配(file globbing)和单词分割有效.
6 [ "$a" == "z*" ]    # 如果 $a 与 z* 相等(就是字面意思完全一样), 那么结果为真.
7
8 # 感谢, Stephane Chazela
```

!=

不等号

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

这个操作符将在 [[...]] 结构中使用模式匹配。

<

小于, 按照ASCII字符进行排序

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

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

注意“\”使用在 [] 结构中的时候需要被转义。

>

大于, 按照ASCII字符进行排序

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

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

注意”>”使用在[]结构中的时候需要被转义.

参考[例子26-11](#), 这个例子展示了如何使用这个比较操作符.

-z

字符串为”null”, 意思就是字符串长度为零

-n

字符串不为”null”.

当-n使用在中括号中进行条件测试的时候, 必须要把字符串用双引号引用起来. 如果采用了未引用的字符串来使用! -z, 甚至是在条件测试中括号(参见[例子7-6](#)) 中只使用未引用的字符串的话, 一般也是可以工作的, 然而, 这是一种不安全的习惯. 习惯于使用引用的测试字符串才是正路[1].

例子7-5. 算术比较与字符串比较

```
1 #!/bin/bash
2
3 a=4
4 b=5
5
6 # 这里的 "a" 和 "b" 既可以被认为是整型也可被认为是字符串.
7 # 这里在算术比较与字符串比较之间是容易让人产生混淆,
8 #+ 因为 Bash 变量并不是强类型的.
9
10 # Bash 允许对于变量进行整形操作与比较操作.
11 #+ 但前提是变量中只能包含数字字符.
12 # 不管怎么样, 还是要小心.
13
14 echo
15
16 if [ "$a" -ne "$b" ]
17 then
18     echo "$a is not equal to $b"
19     echo "(arithmetic comparison)"
20 fi
21
22 echo
23
24 if [ "$a" != "$b" ]
25 then
26     echo "$a is not equal to $b."
27     echo "(string comparison)"
28     #      "4" != "5"
29     # ASCII 52 != ASCII 53
30 fi
31
32 # 在这个特定的例子中, "-ne" 和 "!=" 都可以.
33
```

```
34 echo  
35  
36 exit 0
```

例子7-6. 检查字符串是否为null

```
1 #!/bin/bash  
2 # str-test.sh: 检查null字符串和未引用的字符串,  
3 #+ but not strings and sealing wax, not to mention cabbages and kings . . .  
4 #+ 但不是字符串和封蜡, 也并没有提到卷心菜和国王. . . ??? (没看懂, rojy bug)  
5  
6 # 使用 if [ ... ]  
7  
8  
9 # 如果字符串并没有被初始化, 那么它里面的值未定义.  
10 # 这种状态被称为"null" (注意这与零值不同).  
11  
12 if [ -n $string1 ]      # $string1 没有被声明和初始化.  
13 then  
14   echo "String \"string1\" is not null."  
15 else  
16   echo "String \"string1\" is null."  
17 fi  
18 # 错误的结果.  
19 # 显示$string1为非null, 虽然这个变量并没有被初始化.  
20  
21  
22 echo  
23  
24  
25 # 让我们再试一下.  
26  
27 if [ -n "$string1" ]  # 这次$string1被引号扩起来了.  
28 then  
29   echo "String \"string1\" is not null."  
30 else  
31   echo "String \"string1\" is null."  
32 fi                  # 注意一定要将引用的字符放到中括号结构中!  
33  
34  
35 echo  
36  
37  
38 if [ $string1 ]       # 这次, 就一个$string1, 什么都不加.  
39 then  
40   echo "String \"string1\" is not null."
```

```

41 else
42   echo "String \"string1\" is null."
43 fi
44 # 这种情况运行的非常好.
45 # [ ] 测试操作符能够独立检查string是否为null.
46 # 然而, 使用("$string1")是一种非常好的习惯.
47 #
48 # 就像Stephane Chazelas所指出的,
49 #   if [ $string1 ]      只有一个参数, "]"
50 #   if [ "$string1" ]    有两个参数, 一个为空的"$string1", 另一个是"]"
51
52
53
54 echo
55
56
57
58 string1=initialized
59
60 if [ $string1 ]      # 再来, 还是只有$string1, 什么都不加.
61 then
62   echo "String \"string1\" is not null."
63 else
64   echo "String \"string1\" is null."
65 fi
66 # 再来试一下, 给出了正确的结果.
67 # 再强调一下, 使用引用的("$string1")还是更好一些, 原因我们上边已经说过了.
68
69
70 string1="a = b"
71
72 if [ $string1 ]      # 再来, 还是只有$string1, 什么都不加.
73 then
74   echo "String \"string1\" is not null."
75 else
76   echo "String \"string1\" is null."
77 fi
78 # 未引用的"$string1", 这回给出了错误的结果!
79
80 exit 0
81 # 也感谢Florian Wisser, 给出了上面这个"足智多谋"的例子.

```

例子7-7. zmore

```

1#!/bin/bash
2# zmore

```

```

3
4 #使用'more'来查看gzip文件
5
6 NOARGS=65
7 NOTFOUND=66
8 NOTZIP=67
9
10 if [ $# -eq 0 ] # 与if [ -z "$1" ]效果相同
11 # (译者注：上边这句注释有问题)，$1是可以存在的，可以为空，如：zmore "" arg2 arg3
12 then
13   echo "Usage: `basename $0` filename" >&2
14   # 错误消息输出到stderr.
15   exit $NOARGS
16   # 返回65作为脚本的退出状态的值(错误码).
17 fi
18
19 filename=$1
20
21 if [ ! -f "$filename" ]    # 将$filename引用起来，这样允许其中包含空白字符.
22 then
23   echo "File $filename not found!" >&2
24   # 错误消息输出到stderr.
25   exit $NOTFOUND
26 fi
27
28 if [ ${filename##*.} != "gz" ]
29 # 在变量替换中使用中括号结构.
30 then
31   echo "File $1 is not a gzipped file!"
32   exit $NOTZIP
33 fi
34
35 zcat $1 | more
36
37 # 使用过滤命令'more.'
38 # 当然，如果你愿意，也可以使用'less'.
39
40
41 exit $?  # 脚本将把管道的退出状态作为返回值.
42 # 事实上，也不一定非要加上"exit $?", 因为在任何情况下,
43 # 脚本都会将最后一条命令的退出状态作为返回值.

```

compound comparison

-a

逻辑与

`exp1 -a exp2` 如果表达式`exp1`和`exp2`都为真的话, 那么结果为真.

-o

逻辑或

`exp1 -o exp2` 如果表达式`exp1`和`exp2`中至少有一个为真的话, 那么结果为真.

这与Bash中的比较操作符`&&`和`||`非常相像, 但是这个两个操作符是用在**双中括号结构**中的.

```
1 "[[ condition1 && condition2 ]]"
```

-o和-a操作符一般都是和test命令或者是单中括号结构一起使用的.

```
1 if [ "$exp1" -a "$exp2" ]
```

请参考[例子8-3](#), [例子26-16](#), 和[例子A-29](#), 这几个例子演示了混合比较操作符的行为.

注意事项

1. 就像S.C.所指出的那样, 在一个混合测试中, 即使使用引用的字符串变量也可能还不够. 如果`$string`为空的话, `[-n "$string" -o "$a" = "$b"]` 可能会在某些版本的Bash中产生错误. 安全的做法是附加一个额外的字符给可能的空变量, `["x$string" != x -o "x$a" = "x$b"]` ("x"字符是可以相互抵消的).

4 嵌套的if/then条件测试

可以通过if/then结构来使用嵌套的条件测试. 最终的结果和上面使用&&混合比较操作符的结果是相同的.

```
1 if [ condition1 ]
2 then
3     if [ condition2 ]
4     then
5         do-something # But only if both "condition1" and "condition2" valid.
6     fi
7 fi
```

参考[例子34-4](#), 里边有一个使用if/then结构进行条件测试的例子.

5 检测你对测试知识的掌握情况

系统范围的xinitrc文件可以用来启动X server. 这个文件包含了相当多的if/then条件测试, 下面是这个文件的部分节选.

```
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     # 失败后的安全设置. 虽然我们永远都不会走到这来.
7     # (我们在Xclients中也提供了相同的机制) 保证它不会被破坏.
8     xclock -geometry 100x100-5+5 &
9     xterm -geometry 80x50-50+150 &
10    if [ -f /usr/bin/netscape -a -f /usr/share/doc/HTML/index.html ]
11    then
12        netscape /usr/share/doc/HTML/index.html &
13    fi
14 fi
```

解释上边节选中”条件测试”结构中的内容, 然后检查整个文件, /etc/X11/xinit/xinitrc, 并且分析其中的if/then测试结构. 你可能需要查阅一下后边讲解的知识, 比如说[grep](#), [sed](#), 和[正则表达式](#).

第8章 操作符与相关主题

本章目录

1. 操作符
 2. 数字常量
-

1 操作符

赋值

变量赋值

初始化或者修改变量的值

=

通用赋值操作符, 可用于算术和字符串赋值.

```
1 var=27
2 category=minerals # 在"="之后是不允许出现空白字符的.
```

④ 不要混淆”=”赋值操作符与=测试操作符.

```
1 #      = 在这里是测试操作符
2
3 if [ "$string1" = "$string2" ]
4 # if [ "X$string1" = "X$string2" ] 是一种更安全的做法,
5 # 这样可以防止两个变量中的一个为空所产生的错误.
6 # (字符"X"作为前缀在等式两边是可以相互抵消的.)
7 then
8     command
9 fi
```

算术操作符

+

加法计算

-

减法计算

*

乘法计算

/

除法计算

**

幂运算

```
1 # 在Bash, 版本2.02, 中开始引入了"**" 幂运算符.
2
3 let "z=5**3"
4 echo "z = $z" # z = 125
```

模运算, 或者是求余运算(返回一次除法运算的余数)

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

$5/3 = 1$ 余数为2

模运算经常在其他的一些情况中出现, 比如说产生特定范围的数字(参见例子9-25和例子9-28), 或者格式化程序的输出(参见例子26-15和例子A-6). 它甚至可以用来产生质数, (参见例子A-16). 事实上模运算在算术运算中的使用频率高得惊人.

例子8-1. 最大公约数

```
1 #!/bin/bash
2 # gcd.sh: 最大公约数
3 #       使用Euclid的算法
4
5 # 两个整数的"最大公约数" (gcd),
6 #+ 就是两个整数所能够同时整除的最大的数.
7
8 # Euclid算法采用连续除法.
9 # 在每一次循环中,
10 #+ 被除数 <--- 除数
11 #+ 除数 <--- 余数
12 #+ 直到 余数 = 0.
13 #+ 在最后一次循环中, gcd = 被除数.
14 #
15 # 关于Euclid算法的更精彩的讨论, 可以到
16 #+ Jim Loy的站点, http://www.jimloy.com/number/euclids.htm.
17
18
19 # -----
20 # 参数检查
21 ARGS=2
22 E_BADARGS=65
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            #+ 在这里, 哪个值给的大都没关系.
37               # 为什么没关系?
38
39   remainder=1            # 如果在循环中使用了未初始化的变量,
40                         #+ 那么在第一次循环中,
41                         #+ 它将会产生一个错误消息.
```

```

42
43     until [ "$remainder" -eq 0 ]
44     do
45         let "remainder = $dividend % $divisor"
46         dividend=$divisor          # 现在使用两个最小的数来重复.
47         divisor=$remainder
48     done                      # Euclid的算法
49
50 }                           # Last $dividend is the gcd.
51
52
53 gcd $1 $2
54
55 echo; echo "GCD of $1 and $2 = $dividend"; echo
56
57
58 # Exercise :
59 # -----
60 # 检查传递进来的命令行参数来确保它们都是整数.
61 #+ 如果不是整数, 那就给出一个适当的错误消息并退出脚本.
62
63 exit 0

```

+=

”加-等于” (把变量的值增加一个常量然后再把结果赋给变量)

let ”var += 5” var变量的值会在原来的基础上加5.

-=

”减-等于” (把变量的值减去一个常量然后再把结果赋给变量)

***=**

”乘-等于” (先把变量的值乘以一个常量的值, 然后再把结果赋给变量)

let ”var *= 4” var变量的结果将会在原来的基础上乘以4.

/=

”除-等于” (先把变量的值除以一个常量的值, 然后再把结果赋给变量)

”取模-等于” (先对变量进行模运算, 即除以一个常量取模, 然后把结果赋给变量)

算术操作符经常会出现expr或let表达式中.

例子8-2. 使用算术操作符

```

1#!/bin/bash
2# 使用10种不同的方法计数到11.
3
4n=1; echo -n "$n "
5
6let "n = $n + 1"    # let "n = n + 1" 也可以.
7echo -n "$n "
8
9

```

```

10 : $((n = $n + 1))
11 # ":" 是必需的, 因为如果没有":"的话,
12 #+ Bash将会尝试把"$((n = $n + 1))"解释为一个命令.
13 echo -n "$n"
14
15 (( n = n + 1 ))
16 # 上边这句是一种更简单方法.
17 # 感谢, David Lombard, 指出这点.
18 echo -n "$n"
19
20 n=$((n + 1))
21 echo -n "$n"
22
23 : $[ n = $n + 1 ]
24 # ":" 是必需的, 因为如果没有":"的话,
25 #+ Bash将会尝试把"$[ n = $n + 1 ]"解释为一个命令.
26 # 即使"n"被初始化为字符串, 这句也能够正常运行.
27 echo -n "$n"
28
29 n=$[ $n + 1 ]
30 # 即使"n"被初始化为字符串, 这句也能够正常运行.
31 #* 应该尽量避免使用这种类型的结构, 因为它已经被废弃了, 而且不具可移植性.
32 # 感谢, Stephane Chazelas.
33 echo -n "$n"
34
35 # 现在来一个C风格的增量操作.
36 # 感谢, Frank Wang, 指出这点.
37
38 let "n++"           # let "+n" 也可以.
39 echo -n "$n"
40
41 (( n++ ))          # (( +n )) 也可以.
42 echo -n "$n"
43
44 : $(( n++ ))       # : $(( +n )) 也可以.
45 echo -n "$n"
46
47 : $[ n++ ]          # : $[ +n ] 也可以.
48 echo -n "$n"
49
50 echo
51
52 exit 0

```

► 在Bash中的整型变量事实上是一个有符号的long(32-bit)整型值, 所表示的范围是-

2147483648到2147483647. 如果超过这个范围进行算术操作的话, 那么将不会得到你期望的结果.(译者注: 溢出)

```
1 a=2147483646
2 echo "a = $a"      # a = 2147483646
3 let "a+=1"          # 变量"a"加1.
4 echo "a = $a"      # a = 2147483647
5 let "a+=1"          # 变量"a"再加1, 就会超出范围限制了.
6 echo "a = $a"      # a = -2147483648
7                      # 错误(超出范围了)
```

在2.05b版本之后, Bash开始支持64位整型了.

⑧ Bash不能够处理浮点运算. 它会把包含小数点的数字看作字符串.

```
1 a=1.5
2
3 let "b = $a + 1.3" # 错误.
4 # t2.sh: let: b = 1.5 + 1.3: 表达式的语法错误(错误标志为".5 + 1.3")
5
6 echo "b = $b"      # b=1
```

如果非要做浮点运算的话, 可以在脚本中使用bc, 这个命令可以进行浮点运算, 或者调用数学库函数.

位操作符.

位操作符在shell脚本中很少被使用, 它们最主要的用途就是操作和测试从端口或者sockets中读取的值. 位翻转”Bit flipping”与编译语言的联系很紧密, 比如C/C++, 在这种语言中它可以运行的足够快.

<<

左移一位(每次左移都相当于乘以2)

<<=

”左移-赋值”

let ”var <<= 2” 这句的结果就是变量var左移2位(就是乘以4)

>>

右移一位(每次右移都将除以2)

>>=

”右移-赋值” (与<<=正好相反)

&

按位与

&=

”按位与-赋值”

|

按位或

| =

”按位或-赋值”

~

按位反

!
按位非

按位异或XOR
= ”按位异或-赋值”

逻辑操作符

&&
与(逻辑)

```
1 if [ $condition1 ] && [ $condition2 ]
2 # 与 if [ $condition1 -a $condition2 ] 相同
3 # 如果condition1和condition2都为true, 那结果就为true.
4
5 if [[ $condition1 && $condition2 ]]    # 也可以.
6 # 注意: &&不允许出现在[ ... ]结构中.
```

* &&也可以用在与列表中, 但是使用在连接命令中时, 需要依赖于具体的上下文.

||
或(逻辑)

```
1 if [ $condition1 ] || [ $condition2 ]
2 # 与 if [ $condition1 -o $condition2 ] 相同
3 # 如果condition1或condition2中的一个为true, 那么结果就为true.
4
5 if [[ $condition1 || $condition2 ]]    # 也可以.
6 # 注意||操作符是不能够出现在[ ... ]结构中的.
```

► Bash将会测试每个表达式的退出状态码, 这些表达式由逻辑操作符连接起来.

例子8-3]. 使用&&和||进行混合条件测试

```
1#!/bin/bash
2
3a=24
4b=47
5
6if [ "$a" -eq 24 ] && [ "$b" -eq 47 ]
7then
8  echo "Test #1 succeeds."
9else
10 echo "Test #1 fails."
11fi
12
13# ERROR:  if [ "$a" -eq 24 && "$b" -eq 47 ]
14#+      尝试运行, [ "$a" -eq 24 ,
15#+      因为没找到匹配的',所以失败了.
16#
```

```
17 # 注意: if [[ $a -eq 24 && $b -eq 24 ]] 也能正常运行.
18 # 双中括号的if-test结构要比
19 #+ 单中括号的if-test结构更加灵活.
20 # (在第17行"&&"与第6行的"&&"具有不同的含义.)
21 # 感谢, Stephane Chazelas, 指出这点.
22
23
24 if [ "$a" -eq 98 ] || [ "$b" -eq 47 ]
25 then
26     echo "Test #2 succeeds."
27 else
28     echo "Test #2 fails."
29 fi
30
31
32 # -a和-o选项提供了
33 #+ 一种可选的混合条件测试的方法.
34 # 感谢Patrick Callahan指出这点.
35
36
37 if [ "$a" -eq 24 -a "$b" -eq 47 ]
38 then
39     echo "Test #3 succeeds."
40 else
41     echo "Test #3 fails."
42 fi
43
44
45 if [ "$a" -eq 98 -o "$b" -eq 47 ]
46 then
47     echo "Test #4 succeeds."
48 else
49     echo "Test #4 fails."
50 fi
51
52
53 a=rhino
54 b=crocodile
55 if [ "$a" = rhino ] && [ "$b" = crocodile ]
56 then
57     echo "Test #5 succeeds."
58 else
59     echo "Test #5 fails."
60 fi
61
```

```
62 exit 0
```

&&和||操作符也可以用在算术上下文中.

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

混杂的操作符

,

逗号操作符

逗号操作符可以连接两个或多个算术运算. 所有的操作都会被运行(可能会有副作用), 但是只会返回最后操作的结果.

```
1 let "t1 = ((5 + 3, 7 - 1, 15 - 4))"
2 echo "t1 = $t1"           # t1 = 11
3
4 let "t2 = ((a = 9, 15 / 3))"  # 设置"a"并且计算"t2".
5 echo "t2 = $t2    a = $a"    # t2 = 5    a = 9
```

逗号操作符主要用在for循环中. 参见[例子10-12](#).

2 数字常量

shell脚本在默认情况下都是把数字作为10进制数来处理,除非这个数字采用了特殊的标记或者前缀.如果数字以0开头的话那么就是8进制数.如果数字以0x开头的话那么就是16进制数.如果数字中间嵌入了#的话,那么就被认为是BASE#NUMBER形式的标记法(有范围和符号限制).

例子8-4. 数字常量表示法

```
1 #!/bin/bash
2 # numbers.sh: 几种不同数制的数字表示法.
3
4 # 10进制: 默认情况
5 let "dec = 32"
6 echo "decimal number = $dec"          # 32
7 # 这没什么特别的.
8
9
10 # 8进制: 以'0'(零)开头
11 let "oct = 032"
12 echo "octal number = $oct"           # 26
13 # 表达式结果是用10进制表示的.
14 # -----
15
16 # 16进制: 以'0x'或者'0X'开头的数字
17 let "hex = 0x32"
18 echo "hexadecimal number = $hex"    # 50
19 # 表达式结果是用10进制表示的.
20
21 # 其他进制: BASE#NUMBER
22 # BASE的范围在2到64之间.
23 # NUMBER的值必须使用BASE范围内的符号来表示, 具体看下边的示例.
24
25
26 let "bin = 2#111100111001101"
27 echo "binary number = $bin"          # 31181
28
29 let "b32 = 32#77"
30 echo "base-32 number = $b32"         # 231
31
32 let "b64 = 64#@_"
33 echo "base-64 number = $b64"          # 4031
34 # 这个表示法只能工作于受限的ASCII字符范围(2 - 64).
35 # 10个数字 + 26个小写字母 + 26个大写字符 + @_ +
36
37
38 echo
```

```
39
40 echo $((36#zz)) $((2#10101010)) $((16#AF16)) $((53#1aA))
41                                     # 1295 170 44822 3375
42
43
44 # 重要的注意事项：
45 # -----
46 # 使用一个超出给定进制的数字的话,
47 #+ 将会引起一个错误.
48
49 let "bad_oct = 081"
50 # (部分的) 错误消息输出:
51 # bad_oct = 081: value too great for base (error token is "081")
52 #          Octal numbers use only digits in the range 0 - 7.
53
54 exit 0      # 感谢, Rich Bartell 和 Stephane Chazelas的指正.
```

第三部分

进阶

内容

- 9. 变量重游
 - 10. 循环与分支
 - 11. 内部命令与内建命令
 - 12. 外部过滤器, 程序和命令
 - 13. 系统与管理命令
 - 14. 命令替换
 - 15. 算术扩展
 - 16. I/O重定向
 - 17. Here Document
 - 18. 休息片刻
-

第9章 变量重游

本章目录

1. 内部变量
 2. 操作字符串
 3. 参数替换
 4. 指定变量的类型: 使用declare或者typeset
 5. 变量的间接引用
 6. \$RANDOM: 产生随机整数
 7. 双圆括号结构
-

如果变量使用的恰当, 将会使得脚本更加强大和有弹性. 但这要求我们学习变量的精妙之处及其细微的差别.

1 内部变量

内建变量

这些变量将会影响bash脚本的行为.

\$BASH

Bash的二进制程序文件的路径

```
1 bash$ echo $BASH
2 /bin/bash
```

\$BASH_ENV

这个[环境变量](#)会指向一个Bash的启动文件, 当一个脚本被调用的时候, 这个启动文件将会被读取.

\$BASH_SUBSHELL

这个变量用来提示子shell的层次. 这是一个Bash的新特性, 直到版本3的Bash才被引入近来.

参考[例子20-1](#)中的用法.

\$BASH_VERSINFO[n]

这是一个含有6个元素的数组, 它包含了所安装的Bash的版本信息. 这与下边的\$BASH_VERSION很相像, 但是这个更加详细一些.

```
1 # Bash version info:
2
3 for n in 0 1 2 3 4 5
4 do
5   echo "BASH_VERSINFO[$n] = ${BASH_VERSINFO[$n]}"
6 done
7
8 # BASH_VERSINFO[0] = 3      # 主版本号.
9 # BASH_VERSINFO[1] = 00     # 次版本号.
10 # BASH_VERSINFO[2] = 14    # 补丁次数.
11 # BASH_VERSINFO[3] = 1      # 编译版本.
12 # BASH_VERSINFO[4] = release # 发行状态.
13 # BASH_VERSINFO[5] = i386-redhat-linux-gnu # 结构体系
14                                # (与变量$MACHTYPE相同).
```

\$BASH_VERSION

安装在系统上的Bash版本号

```
1 bash$ echo $BASH_VERSION
2 3.00.14(1)-release
```

```
1 tcsh% echo $BASH_VERSION
```

```
2 BASH_VERSION: Undefined variable.
```

检查\$BASH_VERSION对于判断系统上到底运行的是哪个shell来说是一种非常好的方法. 变量\$SHELL有时候不能够给出正确的答案.

\$DIRSTACK

在目录栈中最顶端的值. (将会受到pushd和popd的影响)

这个内建变量与dirs命令相符, 但是dirs命令会显示目录栈的整个内容.

\$EDITOR

脚本所调用的默认编辑器, 通常情况下是vi或者是emacs.

\$EUID

”有效”用户ID

不管当前用户被假定成什么用户, 这个数都用来表示当前用户的标识号, 也可能使用su命令来达到假定的目的.

警告: \$EUID并不一定与\$UID相同.

\$FUNCNAME

当前函数的名字

```
1 xyz23 ()  
2 {  
3     echo "$FUNCNAME now executing." # 打印: xyz23 now executing.  
4 }  
5  
6 xyz23  
7  
8 echo "FUNCNAME = $FUNCNAME"      # FUNCNAME =  
9                      # 超出函数的作用域就变为null值了.
```

\$GLOBIGNORE

一个文件名的模式匹配列表, 如果在通配(globbing)中匹配到的文件包含有这个列表中的某个文件, 那么这个文件将被从匹配到的结果中去掉.

\$GROUPS

目前用户所属的组

这是一个当前用户的组id列表(数组), 与记录在/etc/passwd文件中的内容一样.

```
1 root# echo $GROUPS  
2 0  
3  
4 root# echo ${GROUPS[1]}  
5 1  
6  
7 root# echo ${GROUPS[5]}
```

\$HOME

用户的home目录, 一般是/home/username(参见例子9-15)

\$HOSTNAME .

`hostname`放在一个初始化脚本中, 在系统启动的时候分配一个系统名字. 然而, `gethostname()`函数可以用来设置这个Bash内部变量\$HOSTNAME. 参见[例子9-15](#).

\$HOSTTYPE

主机类型

就像\$MACHTYPE, 用来识别系统硬件.

```
1 bash$ echo $HOSTTYPE
2 i686
```

\$IFS

内部域分隔符

这个变量用来决定Bash在解释字符串时如何识别域, 或者单词边界.

\$IFS默认为空白(空格, 制表符, 和换行符), 但这是可以修改的, 比如, 在分析逗号分隔的数据文件时, 就可以设置为逗号. 注意\$*使用的是保存在\$IFS中的第一个字符. 参见[例子5-1](#).

```
1 bash$ echo $IFS | cat -vte
2 $
3 (显示缩进和行末的\$字符。)
4
5
6 bash$ bash -c 'set w x y z; IFS=":-"; echo "$*"
7 w:x:y:z
8 (从字符串中读取命令, 并分配参数给位置参数.)
```

⑧ \$IFS处理其他字符与处理空白字符不同.

例子9-1. \$IFS与空白字符

```
1#!/bin/bash
2# $IFS 处理空白与处理其他字符不同.
3
4output_args_one_per_line()
5{
6    for arg
7        do echo "[${arg}]"
8    done
9}
10
11echo; echo "IFS=\\" \""
12echo "-----"
```

```

14 IFS=" "
15 var=" a b c "
16 output_args_one_per_line $var
17 # output_args_one_per_line `echo " a b c   "`
18 #
19 # [a]
20 # [b]
21 # [c]
22
23
24 echo; echo "IFS=:"
25 echo "-----"
26
27 IFS=:
28 var=":a:b:c:::"          # 与上边一样，但是用" "替换了":".
29 output_args_one_per_line $var
30 #
31 # []
32 # [a]
33 # []
34 # [b]
35 # [c]
36 # []
37 # []
38 # []
39
40 # 同样的事情也会发生在awk的"FS"域中。
41
42 # 感谢，Stephane Chazelas.
43
44 echo
45
46 exit 0

```

(感谢, S. C., 进行了澄清与举例.)

参见[例子12-37](#), [例子10-7](#), 和[例子17-14](#)都是展示如何使用\$IFS的例子.

\$LC_COLLATE

常在.bashrc或/etc/profile中设置, 这个变量用来控制文件名扩展和模式匹配的展开顺序. 如果\$LC_COLLATE设置得不正确的话, LC_COLLATE会在文件名匹配(filename globbing)中产生不可预料的结果.

在2.05以后的Bash版本中, 文件名匹配(filename globbing)将不在区分中括号结构中的字符范围里字符的大小写. 比如, ls [A-M]* 既能够匹配为File1.txt也能够匹配为file1.txt. 为了能够恢复中括号里字符的匹配行为(即区分大小写), 可以设置变量LC_COLLATE为C, 在文件/etc/profile或 /.bashrc中使用export LC_COLLATE=C, 可以达到这个目的.

\$LC_CTYPE .

这个内部变量用来控制通配(globbing)和模式匹配中的字符串解释.

\$LINENO .

这个变量用来记录自身在脚本中所在的行号. 这个变量只有在脚本使用这个变量的时候才有意义, 并且这个变量一般用于调试目的.

```
1 # *** 调试代码块开始 ***
2 last_cmd_arg=$_ # Save it.
3
4 echo "At line number $LINENO, variable \"v1\" = $v1"
5 echo "Last command argument processed = $last_cmd_arg"
6 # *** 调试代码块结束 ***
```

\$MACHTYPE .

机器类型

标识系统的硬件.

```
1 bash$ echo $MACHTYPE
2 i686
```

\$OLDPWD .

之前的工作目录("OLD-print-working-directory", 就是之前你所在的目录)

\$OSTYPE .

操作系统类型

```
1 bash$ echo $OSTYPE
2 linux
```

\$PATH .

可执行文件的搜索路径, 一般为/usr/bin/, /usr/X11R6/bin/, /usr/local/bin, 等等.

当给出一个命令时, shell会自动生成一张哈希(hash)表, 并且在这张哈希表中按照path变量中所列出的路径来搜索这个可执行命令. 路径会存储在环境变量中, \$PATH变量本身就是一个以冒号分隔的目录列表. 通常情况下, 系统都是在/etc/profile和/.bashrc中存储\$PATH的定义. (参考[Appendix G](#)).

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

PATH=\$PATH:/opt/bin将会把目录/opt/bin附加到当前目录列表中. 在脚本中, 这是一种把目录临时添加到\$PATH中的权宜之计. 当这个脚本退出时, \$PATH将会恢复以前的值(一个子进程, 比如说一个脚本, 是不能够修改父进程的环境变量的, 在这里也就是不能够修改shell本身的环境变量, - 译者注: 也就是脚本所运行的这个shell).

► 当前的"工作目录", ./, 通常是不会出现在\$PATH中的, 这样做的目的是出于安全的考虑.

\$PIPESTATUS

这个数组变量将保存最后一个运行的前台管道的退出状态码. 相当有趣的是, 这个退出状态码和最后一个命令运行的退出状态码并不一定相同.

```
1 bash$ echo $PIPESTATUS
2 0
3
4 bash$ ls -al | bogus_command
5 bash: bogus_command: command not found
6 bash$ echo $PIPESTATUS
7 141
8
9 bash$ ls -al | bogus_command
10 bash: bogus_command: command not found
11 bash$ echo $?
12 127
```

\$PIPESTATUS数组的每个成员都保存了运行在管道中的相应命令的退出状态码. \$PIPESTATUS[0]保存管道中第一个命令的退出状态码. \$PIPESTATUS[1]保存第二个命令的退出状态码, 依此类推.

⑧ \$PIPESTATUS变量在一个登陆的shell中可能会包含一个不正确0值(在3.0以下版本).

```
1 tcsh% bash
2
3 bash$ who | grep nobody | sort
4 bash$ echo ${PIPESTATUS[*]}
5 0
```

\$PIPESTATUS变量在一个登陆的shell中可能会包含一个不正确0值(在3.0以下版本).

```
1 tcsh% bash
2
3 bash$ who | grep nobody | sort
4 bash$ echo ${PIPESTATUS[*]}
5 0
```

如果一个脚本包含了上边的这行, 那么将会产生我们所期望的0 1 0的输出.

感谢, Wayne Pollock指出这一点并提供了上边的例子.

► 在某些上下文中, 变量\$PIPESTATUS可能不会给出期望的结果.

```
1 bash$ echo $BASH_VERSION
2 3.00.14(1)-release
3
4 bash$ $ ls | bogus_command | wc
5 bash: bogus_command: command not found
6 0      0      0
7
8 bash$ echo ${PIPESTATUS[@]}
```

9 | 141 127 0

Chet Ramey把上边输出不正确的原因归咎于ls的行为. 因为如果把ls的结果放到管道上, 并且这个输出并没有被读取, 那么SIGPIPE将会杀掉它, 同时退出状态码变为141. 而不是我们所期望的0. 这种情况也会发生在tr命令中.

⑧ \$PIPESTATUS是一个”不稳定”变量. 这个变量需要在任何命令干涉之前, 并在管道询问之后立刻被查询.

```
1 bash$ $ ls | bogus_command | wc
2 bash: bogus_command: command not found
3 0      0      0
4
5 bash$ echo ${PIPESTATUS[@]}
6 0 127 0
7
8 bash$ echo ${PIPESTATUS[@]}
9 0
```

\$PPID .

程的\$PPID就是这个进程的父进程的进程ID(pid). [1]
和pidof命令比较一下.

\$PROMPT_COMMAND .

这个变量保存了在主提示符\$PS1显示之前需要执行的命令.

\$PS1 .

这是主提示符, 可以在命令行中见到它.

\$PS2 .

第二提示符, 当你需要额外输入的时候, 你就会看到它. 默认显示”>”.

\$PS3 .

第三提示符, 它在一个select循环中显示(参见[例子10-29](#)).

\$PS4 .

第四提示符, 当你使用-x选项来调用脚本时, 这个提示符会出现在每行输出的开头. 默认显示”+”.

\$PWD .

工作目录(你当前所在的目录)
这与内建命令pwd作用相同.

```
1#!/bin/bash
2
3 E_WRONG_DIRECTORY=73
4
```

```

5 clear # 清屏.
6
7 TargetDirectory=/home/bozo/projects/GreatAmericanNovel
8
9 cd $TargetDirectory
10 echo "Deleting stale files in $TargetDirectory."
11
12 if [ "$PWD" != "$TargetDirectory" ]
13 then    # 防止偶然删错目录.
14     echo "Wrong directory!"
15     echo "In $PWD, rather than $TargetDirectory!"
16     echo "Bailing out!"
17     exit $E_WRONG_DIRECTORY
18 fi
19
20 rm -rf *
21 rm .[A-Za-z0-9]*    # 删除点文件(译者注: 隐藏文件).
22 # rm -f .[^.]* ..?* 为了删除以多个点开头的文件.
23 # (shopt -s dotglob; rm -f *) 也可以.
24 # 感谢, S.C. 指出这点.
25
26 # 文件名可以包含ascii中0 - 255范围内的所有字符, 除了"/".
27 # 删除以各种诡异字符开头的文件将会作为一个练习留给大家.
28
29 # 如果必要的话, 这里预留给其他操作.
30
31 echo
32 echo "Done."
33 echo "Old files deleted in $TargetDirectory."
34 echo
35
36
37 exit 0

```

\$REPLY 当没有参数变量提供给read命令的时候, 这个变量会作为默认变量提供给read命令. 也可以用于select菜单, 但是只提供所选择变量的编号, 而不是变量本身的值.

```

1#!/bin/bash
2# reply.sh
3
4# REPLY是提供给'read'命令的默认变量.
5
6echo
7echo -n "What is your favorite vegetable? "
8read

```

```

9
10 echo "Your favorite vegetable is $REPLY."
11 # 当且仅当没有变量提供给"read"命令时,
12 #+ REPLY才保存最后一个"read"命令读入的值.
13
14 echo
15 echo -n "What is your favorite fruit? "
16 read fruit
17 echo "Your favorite fruit is $fruit."
18 echo "but..."
19 echo "Value of \$REPLY is still $REPLY."
20 # $REPLY还是保存着上一个read命令的值,
21 #+ 因为变量$fruit被传入到了这个新的"read"命令中.
22
23 echo
24
25 exit 0

```

\$SECONDS .

这个脚本已经运行的时间(以秒为单位).

```

1#!/bin/bash
2
3 TIME_LIMIT=10
4 INTERVAL=1
5
6 echo
7 echo "Hit Control-C to exit before $TIME_LIMIT seconds."
8 echo
9
10 while [ "$SECONDS" -le "$TIME_LIMIT" ]
11 do
12     if [ "$SECONDS" -eq 1 ]
13     then
14         units=second
15     else
16         units=seconds
17     fi
18
19     echo "This script has been running $SECONDS $units."
20     # 在一台比较慢或者是附载过大的机器上,
21     #+ 在单次循环中, 脚本可能会忽略计数.
22     sleep $INTERVAL
23 done
24

```

```
25 echo -e "\a" # Beep!(哔哔声!)
```

```
26
```

```
27 exit 0
```

\$SHELLOPTS .

shell中已经激活的选项的列表, 这是一个只读变量.

```
1 bash$ echo $SHELLOPTS
```

```
2 braceexpand:hashall:histexpand:monitor:history:interactive-comments:emacs
```

\$SHLVL .

Shell级别, 就是Bash被嵌套的深度. 如果是在命令行中, 那么\$SHLVL为1, 如果在脚本中那么\$SHLVL为2.

\$TMOUT .

如果\$TMOUT环境变量被设置为非零值time的话, 那么经过time秒后, shell提示符将会超时. 这将会导致登出(logout).

在2.05b版本的Bash中, \$TMOUT变量与命令read可以在脚本中结合使用.

```
1 # 只能够在Bash脚本中使用, 必须使用2.05b或之后版本的Bash.
```

```
2
```

```
3 TMOUT=3      # 提示输入时间为3秒.
```

```
4
```

```
5 echo "What is your favorite song?"
```

```
6 echo "Quickly now, you only have $TMOUT seconds to answer!"
```

```
7 read song
```

```
8
```

```
9 if [ -z "$song" ]
```

```
10 then
```

```
11   song="(no answer)"
```

```
12   # 默认响应.
```

```
13 fi
```

```
14
```

```
15 echo "Your favorite song is $song."
```

还有更加复杂的方法可以在脚本中实现定时输入. 一种办法就是建立一个定时循环, 当超时的时候给脚本发个信号. 不过这也需要有一个信号处理例程能够捕捉(参见[例子29-5](#)) 由定时循环所产生的中断. (哇欧!).

例子9-2. 定时输入

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

```
2# timed-input.sh
```

```
3
```

```
4# TMOUT=3      在新一些的Bash版本上也能运行的很好.
```

```
5
```

```
6
```

```
7TIMELIMIT=3 # 这个例子中设置的是3秒. 也可以设置为其他的时间值.
```

```

8
9 PrintAnswer()
10 {
11     if [ "$answer" = TIMEOUT ]
12     then
13         echo $answer
14     else      # 别和上边的例子弄混了.
15         echo "Your favorite veggie is $answer"
16         kill $!  # 不再需要后台运行的TimerOn函数了, kill了吧.
17             # $! 变量是上一个在后台运行的作业的PID.
18     fi
19
20 }
21
22
23
24 TimerOn()
25 {
26     sleep $TIMELIMIT && kill -s 14 $$ &
27     # 等待3秒, 然后给脚本发送一个信号.
28 }
29
30 Int14Vector()
31 {
32     answer="TIMEOUT"
33     PrintAnswer
34     exit 14
35 }
36
37 trap Int14Vector 14    # 定时中断(14)会暗中给定时间限制.
38
39 echo "What is your favorite vegetable "
40 TimerOn
41 read answer
42 PrintAnswer
43
44
45 # 无可否认, 这是一个定时输入的复杂实现,
46 #+ 然而"read"命令的"-t"选项可以简化这个任务.
47 # 参考后边的"t-out.sh".
48
49 # 如果你需要一个真正优雅的写法...
50 #+ 建议你使用C或C++来重写这个应用,
51 #+ 你可以使用合适的函数库, 比如'alarm'和'setitimer'来完成这个任务.
52

```

53 | exit 0

另一种选择是使用[stty](#).

例子9-3. 再来一个, 定时输入

```
1 #!/bin/bash
2 # timeout.sh
3
4 # 由Stephane Chazelas所编写,
5 #+ 本书作者做了一些修改.
6
7 INTERVAL=5          # 超时间隔
8
9 timedout_read() {
10    timeout=$1
11    varname=$2
12    old_tty_settings='stty -g'
13    stty -icanon min 0 time ${timeout}0
14    eval read $varname      # 或者仅仅读取$varname变量
15    stty "$old_tty_settings"
16    # 参考"stty"的man页.
17 }
18
19 echo; echo -n "What's your name? Quick! "
20 timedout_read $INTERVAL your_name
21
22 # 这种方法可能并不是在每种终端类型上都可以正常使用的.
23 # 最大的超时时间依赖于具体的中断类型.
24 #+ (通常是25.5秒).
25
26 echo
27
28 if [ ! -z "$your_name" ]  # 如果在超时之前名字被键入...
29 then
30     echo "Your name is $your_name."
31 else
32     echo "Timed out."
33 fi
34
35 echo
36
37 # 这个脚本的行为可能与脚本"timed-input.sh"的行为有些不同.
38 # 每次按键, 计时器都会重置(译者注: 就是从0开始).
39
40 exit 0
```

可能最简单的办法就是使用-t选项来read了.

例子9-4. 定时read

```
1 #!/bin/bash
2 # t-out.sh
3 # 从"syngin seven"的建议中得到的灵感 (感谢).
4
5
6 TIMELIMIT=4          # 4秒
7
8 read -t $TIMELIMIT variable <&1
9 #           ^
10 # 在这个例子中, 对于Bash 1.x和2.x就需要"<&1"了,
11 # 但是Bash 3.x就不需要.
12
13 echo
14
15 if [ -z "$variable" ] # 值为null?
16 then
17   echo "Timed out, variable still unset."
18 else
19   echo "variable = $variable"
20 fi
21
22 exit 0
```

\$UID .

用户ID号, 当前用户的用户标识号, 记录在/etc/passwd文件中

这是当前用户的真实id, 即使只是通过使用su命令来临时改变为另一个用户标识, 这个id也不会被改变. \$UID是一个只读变量, 不能在命令行或者脚本中修改它, 并且和id内建命令很相像.

例子9-5. 我是root么?

```
1 #!/bin/bash
2 # am-i-root.sh: 我是不是root用户?
3
4 ROOT_UID=0    # Root的$UID为0.
5
6 if [ "$UID" -eq "$ROOT_UID" ] # 只有真正的"root"才能经受得住考验?
7 then
8   echo "You are root."
9 else
10  echo "You are just an ordinary user (but mom loves you just the same)."
11 fi
12
13 exit 0
14
```

```

15
16 # ===== #
17 # 下边的代码不会执行, 因为脚本在上边已经退出了.
18
19 # 下边是另外一种判断root用户的方法:
20
21 ROOTUSER_NAME=root
22
23 username='id -nu'          # 或者...    username='whoami'
24 if [ "$username" = "$ROOTUSER_NAME" ]
25 then
26   echo "Rooty, toot, toot. You are root."
27 else
28   echo "You are just a regular fella."
29 fi

```

也请参考一下[例子2-3](#).

变量\$ENV, \$LOGNAME, \$MAIL, \$TERM, \$USER, 和\$USERNAME都不是Bash的[内建变量](#). 然而这些变量经常在Bash的启动文件中被当作[环境变量](#)来设置. \$SHELL是用户登陆shell的名字, 它可以在/etc/passwd中设置, 或者也可以在"init"脚本中设置, 并且它也不是Bash内建的.

```

1 tcsh% echo $LOGNAME
2 bozo
3 tcsh% echo $SHELL
4 /bin/tcsh
5 tcsh% echo $TERM
6 rxvt
7
8 bash$ echo $LOGNAME
9 bozo
10 bash$ echo $SHELL
11 /bin/tcsh
12 bash$ echo $TERM
13 rxvt

```

位置参数

\$0, \$1, \$2, 等等.

位置参数, 从命令行传递到脚本, 或者传递给函数, 或者set给变量(参见[例子4-5](#)和[例子11-15](#))

\$#

命令行参数[\[2\]](#)或者位置参数的个数(参见[例子33-2](#))

\$*

所有的位置参数都被看作为一个单词.

► "\$*"必须被引用起来.

\$@

与\$*相同,但是每个参数都是一个独立的引用字符串,这就意味着,参数是被完整传递的,并没有被解释或扩展。这也意味着,参数列表中每个参数都被看作为单独的单词。

►当然,”\$@”应该被引用起来。

例子9-6. arglist: 通过\$*和\$@列出所有的参数

```
1 #!/bin/bash
2 # arglist.sh
3 # 多使用几个参数来调用这个脚本,比如"one two three".
4
5 E_BADARGS=65
6
7 if [ ! -n "$1" ]
8 then
9   echo "Usage: `basename $0` argument1 argument2 etc."
10  exit $E_BADARGS
11 fi
12
13 echo
14
15 index=1      # 起始计数。
16
17 echo "Listing args with \"\$*\":"
18 for arg in "$*"  # 如果"$*"不被""引用,那么将不能正常地工作。
19 do
20   echo "Arg #$index = $arg"
21   let "index+=1"
22 done          # $* 将所有的参数看成一个单词。
23 echo "Entire arg list seen as single word."
24
25 echo
26
27 index=1      # 重置计数(译者注:从1开始)。
28          # 如果你写这句会发生什么?
29
30 echo "Listing args with \"\$@\":"
31 for arg in "$@"
32 do
33   echo "Arg #$index = $arg"
34   let "index+=1"
35 done          # $@ 把每个参数都看成是单独的单词。
36 echo "Arg list seen as separate words."
37
38 echo
39
```

```

40 index=1          # 重置计数(译者注：从1开始).
41
42 echo "Listing args with \$* (unquoted):"
43 for arg in $*
44 do
45     echo "Arg #\$index = \$arg"
46     let "index+=1"
47 done           # 未引用的\$*将会把参数看成单独的单词.
48 echo "Arg list seen as separate words."
49
50 exit 0

```

shift命令执行以后, \$@将会保存命令行中剩余的参数, 但是没有之前的\$1, 因为被丢弃了.

```

1#!/bin/bash
2# 使用 ./scriptname 1 2 3 4 5 来调用这个脚本
3
4echo "$@"      # 1 2 3 4 5
5shift
6echo "$@"      # 2 3 4 5
7shift
8echo "$@"      # 3 4 5
9
10# 每次"shift"都会丢弃$1.
11# "$@" 将包含剩下的参数.

```

\$@也可以作为工具使用, 用来过滤传递给脚本的输入. cat "\$@"结构既可以接受从stdin传递给脚本的输入, 也可以接受从参数中指定的文件中传递给脚本的输入. 参见[例子12-21](#)和[例子12-22](#).

\$*和\$@中的参数有时候会表现出不一致而且令人迷惑的行为, 这都依赖于\$IFS的设置.

[例子9-7. \\$*和\\$@的不一致的行为](#)

```

1#!/bin/bash
2
3# 内部Bash变量"$*"和"$@"的古怪行为,
4#+ 都依赖于它们是否被双引号引用起来.
5# 单词拆分与换行的不一致的处理.
6
7
8set -- "First one" "second" "third:one" "" "Fifth: :one"
9# 设置这个脚本的参数, $1, $2, 等等.
10
11echo
12
13echo 'IFS unchanged, using "$*",
14c=0
15for i in "$*"                  # 引用起来
16do echo "${((c+=1))}: [$i]"    # 这行在下边每个例子中都一样.

```

```
17          # 打印参数.
18 done
19 echo ---
20
21 echo 'IFS unchanged, using $*'
22 c=0
23 for i in $*           # 未引用
24 do echo "$((c+=1)): [$i]"
25 done
26 echo ---
27
28 echo 'IFS unchanged, using "$@",'
29 c=0
30 for i in "$@"
31 do echo "$((c+=1)): [$i]"
32 done
33 echo ---
34
35 echo 'IFS unchanged, using $@'
36 c=0
37 for i in $@
38 do echo "$((c+=1)): [$i]"
39 done
40 echo ---
41
42 IFS=:
43 echo 'IFS=":", using "$*"'
44 c=0
45 for i in "$*"
46 do echo "$((c+=1)): [$i]"
47 done
48 echo ---
49
50 echo 'IFS=":", using $*'
51 c=0
52 for i in $*
53 do echo "$((c+=1)): [$i]"
54 done
55 echo ---
56
57 var=$*
58 echo 'IFS=":", using "$var" (var=$*)'
59 c=0
60 for i in "$var"
61 do echo "$((c+=1)): [$i]"
```

```
62 done
63 echo ---
64
65 echo 'IFS=":", using $var (var=$*)'
66 c=0
67 for i in $var
68 do echo "$((c+=1)): [$i]"
69 done
70 echo ---
71
72 var="$*"
73 echo 'IFS=":", using $var (var=\"$*\")'
74 c=0
75 for i in $var
76 do echo "$((c+=1)): [$i]"
77 done
78 echo ---
79
80 echo 'IFS=":", using "$var" (var=\"$*\")'
81 c=0
82 for i in "$var"
83 do echo "$((c+=1)): [$i]"
84 done
85 echo ---
86
87 echo 'IFS=":", using "$@"
88 c=0
89 for i in "$@"
90 do echo "$((c+=1)): [$i]"
91 done
92 echo ---
93
94 echo 'IFS=":", using $@"
95 c=0
96 for i in $@
97 do echo "$((c+=1)): [$i]"
98 done
99 echo ---
100
101 var=$@
102 echo 'IFS=":", using $var (var=$@)'
103 c=0
104 for i in $var
105 do echo "$((c+=1)): [$i]"
106 done
```

```

107 echo ---
108
109 echo 'IFS=:, using "$var" (var=$@)'
110 c=0
111 for i in "$var"
112 do echo "${((c+=1))}: [$i]"
113 done
114 echo ---
115
116 var="$@"
117 echo 'IFS=:, using "$var" (var="$@")'
118 c=0
119 for i in "$var"
120 do echo "${((c+=1))}: [$i]"
121 done
122 echo ---
123
124 echo 'IFS=:, using $var (var="$@")'
125 c=0
126 for i in $var
127 do echo "${((c+=1))}: [$i]"
128 done
129
130 echo
131
132 # 使用ksh或者zsh -y来试试这个脚本.
133
134 exit 0
135
136 # 这个例子脚本是由Stephane Chazelas所编写,
137 # 并且本书作者做了轻微改动.

```

\$_@与\$_*中的参数只有在被双引号引用起来的时候才会不同.

```

1#!/bin/bash
2
3# 如果$IFS被设置, 但其值为空,
4#+ 那么"$*"和"$@"将不会像期望的那样显示位置参数.
5
6mecho ()      # 打印位置参数.
7{
8echo "$1,$2,$3";
9}
10
11
12IFS=""        # 设置了, 但值为空.

```

```

13 set a b c      # 位置参数.
14
15 mecho "$*"      # abc,,
16 mecho $*        # a,b,c
17
18 mecho $@        # a,b,c
19 mecho "$@"      # a,b,c
20
21 # 当$IFS值为空时, $*和$@的行为依赖于
22 #+ 正在运行的Bash或者sh的版本.
23 # 因此在脚本中使用这种"特性"是不明智的.
24
25
26 # 感谢, Stephane Chazelas.
27
28 exit 0

```

其他的特殊参数

\$- .

传递给脚本的标记(使用set命令). 参见[例子11-15](#).

⑧ 这本来是ksh的结构, 后来被引进到Bash中, 但是不幸的是, 看起来它不能够可靠的用在Bash脚本中. 一种可能的用法是让一个脚本[测试自身是不是可交互的](#).

\$!

运行在后台的最后一个作业的PID(进程ID)

```

1 LOG=$0.log
2
3 COMMAND1="sleep 100"
4
5 echo "Logging PIDs background commands for script: $0" >> "$LOG"
6 # 所以它们是可以被监控的, 并且可以在必要的时候kill掉它们.
7 echo >> "$LOG"
8
9 # 记录命令.
10
11 echo -n "PID of \"\$COMMAND1\": " >> "$LOG"
12 ${COMMAND1} &
13 echo $! >> "$LOG"
14 # "sleep 100"的PID: 1506
15
16 # 感谢, Jacques Lederer, 对此的建议.

```

1 | possibly_hanging_job &\未换行

```
2 {sleep ${TIMEOUT}; eval 'kill -9 $!' &> /dev/null;}  
3 # 强制结束一个出错程序.  
4 # 很有用, 比如用在init脚本中.  
5  
6 # 感谢, Sylvain Fourmanoit, 发现了"!"变量的创造性用法.
```

\$_

这个变量保存之前执行的命令的最后一个参数的值.

例子9-9. 下划线变量

```
1#!/bin/bash  
2  
3echo $_          # /bin/bash  
4                  # 只是调用/bin/bash来运行这个脚本.  
5  
6du >/dev/null    # 这么做命令行上将没有输出.  
7echo $_          # du  
8  
9ls -al >/dev/null # 这么做命令行上将没有输出.  
10echo $_         # -al (这是最后的参数)  
11  
12:  
13echo $_          # :
```

\$?

命令, 函数, 或者是脚本本身的(参见例子23-7)退出状态码

\$\$

脚本自身的进程ID. \$\$变量在脚本中经常用来构造”唯一的”临时文件名(参见[例子A-13](#), [例子29-6](#), [例子12-28](#), 和[例子11-26](#)). 这么做通常比调用mktemp命令来的简单.

注意事项

1. 当然, 当前运行脚本的PID就是\$\$
2. 术语”argument”和”parameter”通常情况下都可以互换使用. 在本书的上下文中, 它们的意思完全相同, 意思都是传递给脚本或者函数的变量, 或者是位置参数. (译者注: 翻译时, 基本上就未加区分.)

2 操作字符串

Bash所支持的字符串操作的数量多的令人惊讶. 但是不幸的是, 这些工具缺乏统一的标准. 一些是参数替换的子集, 而另外一些则受到UNIX expr命令的影响. 这就导致了命令语法的不一致, 还会引起冗余的功能, 但是这些并没有引起混乱.

字符串长度

```
1 ${#string}
2 expr length $string
3 expr "$string" : '.*'
4
5 stringZ=abcABC123ABCabc
6
7 echo ${#stringZ}          # 15
8 echo `expr length $stringZ` # 15
9 echo `expr "$stringZ" : '.*'` # 15
```

例子9-10. 在一个文本文件的段落之间插入空行

```
1#!/bin/bash
2# paragraph-space.sh
3
4# 在一个单倍行距的文本文件中插入空行.
5# Usage: $0 <FILENAME
6
7MINLEN=45      # 可能需要修改这个值.
8# 假定行的长度小于$MINLEN所指定的长度的时候
9#+ 才认为此段结束.
10
11while read line  # 提供和输入文件一样多的行...
12do
13    echo "$line"  # 输入所读入的行本身.
14
15    len=${#line}
16    if [ "$len" -lt "$MINLEN" ]
17        then echo #在短行(译者注:也就是小于$MINLEN个字符的行)后面添加一空行.
18    fi
19done
20
21exit 0
```

匹配字符串开头的子串长度

```
expr match "$string" '$substring'
$substring是一个正则表达式.
expr "$string" : '$substring'
$substring是一个正则表达式.
```

```
1 stringZ=abcABC123ABCabc
```

```
2 #      |-----|
3
4 echo `expr match "$stringZ" 'abc[A-Z]*.2'`    # 8
5 echo `expr "$stringZ" : 'abc[A-Z]*.2'`        # 8
```

索引

expr index \$string \$substring

在字符串\$string中所匹配到的\$substring第一次所出现的位置.

```
1 stringZ=abcABC123ABCabc
2 echo `expr index "$stringZ" C12`           # 6
3                                         # C 字符的位置.
4
5 echo `expr index "$stringZ" 1c`            # 3
6 # 'c' (in #3 position) matches before '1'.
```

这与C语言中的strchr()函数非常相似.

提取子串 .

`\${string:position}`

在\$string中从位置\$position开始提取子串.

如果\$string是”*”或者”@”, 那么将会提取从位置\$position开始的[位置参数. \[1\]](#)

`\${string:position:length}`

在\$string中从位置\$position开始提取\$length长度的子串.

```
1 stringZ=abcABC123ABCabc
2 #      0123456789.....
3 #      0-based indexing.
4
5 echo ${stringZ:0}                         # abcABC123ABCabc
6 echo ${stringZ:1}                         # bcABC123ABCabc
7 echo ${stringZ:7}                         # 23ABCabc
8
9 echo ${stringZ:7:3}                      # 23A
10                                         # 提取子串长度为3.
11
12
13
14 # 能不能从字符串的右边(也就是结尾)部分开始提取子串?
15
16 echo ${stringZ:-4}                      # abcABC123ABCabc
17 # 默认是提取整个字符串, 就象${parameter:-default}一样.
18 # 然而 . . .
19
20 echo ${stringZ:(-4)}                   # Cabc
21 echo ${stringZ: -4}                    # Cabc
22 # 这样, 它就可以工作了.
23 # 使用圆括号或者添加一个空格可以"转义"这个位置参数.
```

```
24  
25 # 感谢, Dan Jacobson, 指出这点
```

如果\$string参数是”*”或”@”, 那么将会从\$position位置开始提取\$length个位置参数, 但是由于可能没有\$length个位置参数了, 那么就有几个位置参数就提取几个位置参数.

```
1 echo ${*:2}          # 打印出第2个和后边所有的位置参数.  
2 echo ${@:2}          # 同上.  
3  
4 echo ${*:2:3}        # 从第2个开始, 连续打印3个位置参数.
```

expr substr \$string \$position \$length

在\$string中从\$position开始提取\$length长度的子串.

```
1 stringZ=abcABC123ABCabc  
2 #      123456789.....  
3 #      以1开始计算.  
4  
5 echo `expr substr $stringZ 1 2`          # ab  
6 echo `expr substr $stringZ 4 3`          # ABC
```

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

从\$string的开始位置提取\$substring, \$substring是正则表达式.

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

从\$string的开始位置提取\$substring, \$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 # 上边的每个echo都打印出相同的结果.
```

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

从\$string的结尾提取\$substring, \$substring是正则表达式.

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

从\$string的结尾提取\$substring, \$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
```

子串削除 .

\$string#substring

从\$string的开头位置截掉最短匹配的\$substring.

\$string##substring

从\$string的开头位置截掉最长匹配的\$substring.

```
1 stringZ=abcABC123ABCabc
```

```

2 #      |---|
3 #      |-----|
4
5 echo ${stringZ#a*C}      # 123ABCabc
6 # 截掉'a'到'C'之间最短的匹配字符串.
7
8 echo ${stringZ##a*C}      # abc
9 # 截掉'a'到'C'之间最长的匹配字符串.

```

\$string%substring

从\$string的结尾位置截掉最短匹配的\$substring.

\$string%%substring

从\$string的结尾位置截掉最长匹配的\$substring.

```

1 stringZ=abcABC123ABCabc
2 #          ||
3 #      |-----|
4
5 echo ${stringZ%b*c}      # abcABC123ABCa
6 # 从$stringZ的结尾位置截掉'b'到'c'之间最短的匹配.
7
8 echo ${stringZ%%b*c}      # a
9 # 从$stringZ的结尾位置截掉'b'到'c'之间最长的匹配.

```

当你需要构造文件名的时候, 这个操作就显得特别有用.

例子9-11. 转换图片文件格式, 同时更改文件名

```

1#!/bin/bash
2# cvt.sh:
3# 将一个目录下的所有MacPaint格式的图片文件都转换为"pbm"各式的图片文件.
4
5# 使用"netpbm"包中的"macptopbm"程序进行转换,
6#+ 这个程序主要是由Brian Henderson(bryanh@giraffe-data.com)来维护的.
7# Netpbm绝大多数Linux发行版的标准套件.
8
9OPERATION=macptopbm
10SUFFIX=pbm      # 新的文件名后缀.
11
12if [ -n "$1" ]
13then
14    directory=$1      # 如果目录名作为参数传递给脚本...
15else
16    directory=$PWD    # 否则使用当前的工作目录.
17fi
18
19# 假定目标目录中的所有文件都是MacPaint格式的图像文件,
20#+ 并且都是以".mac"作为文件名后缀.
21

```

```

22 for file in $directory/*      # 文件名匹配(filename globbing).
23 do
24   filename=${file%.*c}          # 去掉文件名的".mac"后缀
25           #+# ('.*c' 将会匹配
26           #+# '.,'和'c'之间任意字符串).
27   $OPERATION $file > "$filename.$SUFFIX"
28           # 把结果重定向到新的文件中.
29   rm -f $file                  # 转换后删除原始文件.
30   echo "$filename.$SUFFIX"     # 从stdout输出转换后文件的文件名.
31 done
32
33 exit 0
34
35 # 练习:
36 # -----
37 # 就像它现在的样子, 这个脚本把当前
38 #+ 目录下的所有文件都转换了.
39 # 修改这个脚本, 让它只转换以".mac"为后缀名的文件.

```

例子9-12. 将音频流文件转换为ogg格式的文件

```

1#!/bin/bash
2# ra2ogg.sh: 将音频流文件(*.ra)转换为ogg格式的文件.
3
4# 使用"mplayer"媒体播放器程序:
5#     http://www.mplayerhq.hu/homepage
6#     可能需要安装合适的编解码程序(codec)才能够正常的运行这个脚本.
7# 需要使用"ogg"库和"oggenc":
8#     http://www.xiph.org/
9
10
11OFILEPREF=${1%*.ra}      # 去掉"ra"后缀.
12OFILESUFF=wav            # wav文件的后缀.
13OUTFILE="$OFILEPREF""$OFILESUFF"
14E_NOARGS=65
15
16if [ -z "$1" ]            # 必须要指定一个需要转换的文件名.
17then
18  echo "Usage: `basename $0` [filename]"
19  exit $E_NOARGS
20fi
21
22
23#####
24mplayer "$1" -ao pcm:file=$OUTFILE
25oggenc "$OUTFILE"  # oggenc编码后会自动加上正确的文件扩展名.

```

```

26 #####
27
28 rm "$OUTFILE"      # 删除中介的*.wav文件.
29                      # 如果你想保留这个文件的话，可以把上边这行注释掉.
30
31 exit $?
32
33 # 注意：
34 # -----
35 # 在网站上，简单的在*.ram流音频文件上单击的话，
36 #+ 一般都只会下载真正音频流文件(就是*.ra文件)的URL.
37 # 你可以使用"wget"或者一些类似的工具
38 #+ 来下载*.ra文件本身.
39
40
41 # 练习：
42 # -----
43 # 像上面所看到的，这个脚本只能转换*.ra文件.
44 # 给这个脚本添加一些灵活性，让它能够转换*.ram and other filenames.
45 #
46 # 如果你觉得这还不过瘾，那么你可以扩展这个脚本，
47 #+ 让它自动下载并转换音频流文件.
48 # 给出一个URL，(使用"wget")批处理下载音频流文件,
49 #+ 然后转换它们.

```

一个简单的`getopt`命令的模拟，使用子串提取结构。

例子9-13. 模拟getopt

```

1#!/bin/bash
2# getopt-simple.sh
3# 作者：Chris Morgan
4# 已经经过授权，可以使用在本书中。
5
6
7getopt_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}          # 去掉开头的'/' . .
17           parameter=${tmp%%=*} # 提取参数名。
18           value=${tmp##*=}     # 提取参数值。

```

```

19         echo "Parameter: '$parameter', value: '$value'"
20         eval $parameter=$value
21     fi
22     shift
23 done
24 }
25
26 # 把所有选项传给函数getopt_simple().
27 getopt_simple $*
28
29 echo "test is '$test'"
30 echo "test2 is '$test2'"
31
32 exit 0
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

使用\$replacement来替换第一个匹配的\$string.

\$string//substring/replacement

使用\$replacement来替换所有匹配的\$string.

```

1 stringZ=abcABC123ABCabc
2
3 echo ${stringZ/abc/xyz}          # xyzABC123ABCabc
4                                         # 使用'xyz'来替换第一个匹配的'abc'.
5
6 echo ${stringZ//abc/xyz}         # xyzABC123ABCxyz
7                                         # 用'xyz'来替换所有匹配的'abc'.

```

\$string/#substring/replacement

如果\$string匹配\$string的开头部分, 那么就用\$replacement来替换\$string.

\$string/%substring/replacement

如果\$string匹配\$string的结尾部分, 那么就用\$replacement来替换\$string.

1 使用awk来处理字符串

Bash脚本也可以调用[awk](#) 的字符串操作功能来代替它自己内建的字符串操作.

例子9-14. 提取字符串的另一种方法

```
1 #!/bin/bash
2 # substring-extraction.sh
3
4 String=23skidoo1
5 #      012345678      Bash
6 #      123456789      awk
7 # 注意不同的字符串索引系统:
8 # Bash的第一个字符是从'0'开始记录的.
9 # Awk的第一个字符是从'1'开始记录的.
10
11 echo ${String:2:4} # 位置 3 (0-1-2), 4 个字符长
12                               # skid
13
14 # awk中等价于${string:pos:length}的命令是substr(string,pos,length).
15 echo | awk '
16 { print substr("'"${String}"'",3,4)      # skid
17 }
18 '
19 # 使用一个空的"echo"通过管道传递给awk一个假的输入,
20 #+ 这样就不必提供一个文件名给awk.
21
22 exit 0
```

2 更深入的讨论

如果想了解关于在脚本中使用字符串的更多细节, 请参考[9.3](#)和[expr](#)命令列表的[相关章节](#). 相关脚本的例子, 参见:

[例子12-9](#)

[例子9-17](#)

[例子9-18](#)

[例子9-19](#)

[例子9-21](#)

注意事项

1. 这适用于命令行参数或[函数参数](#).

3 参数替换

处理和(或)扩展变量

`${parameter}`

与`$parameter`相同, 也就是变量`parameter`的值. 在某些上下文中, `${parameter}`很少会产生混淆.

可以把变量和字符串组合起来使用.

```
1 your_id=${USER}-on-${HOSTNAME}
2 echo "$your_id"
3 #
4 echo "Old \$PATH = $PATH"
5 PATH=${PATH}:/opt/bin #在脚本的生命周期中,/opt/bin会被添加到$PATH变量中.
6 echo "New \$PATH = $PATH"
```

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

`${parameter-default}` – 如果变量`parameter`没被声明, 那么就使用默认值.

`${parameter:-default}` – 如果变量`parameter`没被设置, 那么就使用默认值.

```
1 echo ${username-'whoami'}
2 # 如果变量$username还没有被声明,
3 # 那么就echo出'whoami'的结果(译
4 # 者注: 也就是把'whoami'的结果赋值给变量$username).
```

► `${parameter-default}` 和 `${parameter:-default}`在绝大多数的情况下都是相同的. 只有在`parameter`已经被声明, 但是被赋`null`值得时候, 这个额外的`:``才会产生不同的结果.

```
1#!/bin/bash
2# param-sub.sh
3
4# 一个变量是否被声明或设置,
5#+ 将会影响这个变量是否使用默认值,
6#+ 即使这个变量值为空(null).
7
8username0=
9echo "username0 has been declared, but is set to null."
10echo "username0 = ${username0-'whoami'}"
11# 不会有输出.
12
13echo
14
15echo username1 has not been declared.
16echo "username1 = ${username1-'whoami'}"
17# 将会输出默认值.
18
19username2=
20echo "username2 has been declared, but is set to null."
```

```

21 echo "username2 = ${username2:-'whoami'}"
22 #
23 # 会输出, 因为:-会比-多一个条件测试.
24 # 可以与上边的例子比较一下.
25
26
27 #
28
29 # 再来一个:
30
31 variable=
32 # 变量已经被声明, 但是设为空值.
33
34 echo "${variable-0}"      # (没有输出)
35 echo "${variable:-1}"     # 1
36 #
37
38 unset variable
39
40 echo "${variable-2}"      # 2
41 echo "${variable:-3}"     # 3
42
43 exit 0

```

如果脚本并没有接收到来自命令行的参数, 那么默认参数结构将会提供一个默认值给脚本.

```

1 DEFAULT_FILENAME=generic.data
2 filename=${1:-$DEFAULT_FILENAME}
3 # 如果没有指定值, 那么下面的代码块将会使用filename
4 #+ 变量的默认值"generic.data".
5 #
6 # 后续的命令.

```

参考[例子3-4](#), [例子28-2](#), 和[例子A-6](#).

与使用[一个与列表来提供一个默认的命令行参数](#)的方法相比较.

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

`${parameter=default}` – 如果变量parameter没声明, 那么就把它的值设为default.

`${parameter:=default}` – 如果变量parameter没设置, 那么就把它的值设为default.

这两种形式基本上是一样的. 只有在变量\$parameter被声明并且被设置为null值的时候, :才会引起这两种形式的不同.[\[1\]](#) 如上边所示.

```

1 echo ${username='whoami'}
2 # 变量"username"现在被赋值为'whoami'.

```

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

`${parameter+alt_value}` – 如果变量parameter被声明了, 那么就使用alt_value, 否则就使用null字符串.

`${parameter:+alt_value}` – 如果变量parameter被设置了, 那么就使用alt_value, 否则就使用null字符串.

这两种形式绝大多数情况下都一样. 只有在parameter被声明并且设置为null值的时候, 多出来的这个:才会引起这两种形式的不同, 具体请看下边的例子.

```
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 # 产生与a=\${param5+xyz}不同的结果.
26
27 param6=123
28 a=\${param6:+xyz}
29 echo "a = $a"      # a = xyz
```

`\${parameter?err_msg}, \${parameter:+err_msg}`

`\${parameter?err_msg}` – 如果parameter已经被声明, 那么就使用设置的值, 否则打印err_msg错误消息.

`\${parameter:+err_msg}` – 如果parameter已经被设置, 那么就使用设置的值, 否则打印err_msg错误消息.

这两种形式绝大多数情况都是一样的. 和上边所讲的情况一样, 只有在parameter被声明并设置为null值的时候, 多出来的:才会引起这两种形式的不同.

例子9-15. 使用参数替换和错误消息

```
1#!/bin/bash
2
3# 检查一些系统环境变量.
```

```

4 # 这是一种可以做一些预防性保护措施的好习惯。
5 # 比如，如果$USER(用户在控制台上中的名字)没有被设置的话，
6 #+ 那么系统就会不认你。
7
8 : ${HOSTNAME?} ${USER?} ${HOME?} ${MAIL?}
9 echo
10 echo "Name of the machine is $HOSTNAME."
11 echo "You are $USER."
12 echo "Your home directory is $HOME."
13 echo "Your mail INBOX is located in $MAIL."
14 echo
15 echo "If you are reading this message,"
16 echo "critical environmental variables have been set."
17 echo
18 echo
19
20 # -----
21
22 # ${variablename?}结构
23 #+ 也能够检查脚本中变量的设置情况。
24
25 ThisVariable=Value-of-ThisVariable
26 # 注意，顺便提一下，
27 #+ 这个字符串变量可能会被设置一些非法字符。
28 : ${ThisVariable?}
29 echo "Value of ThisVariable is $ThisVariable".
30 echo
31 echo
32
33
34 : ${ZZXy23AB?"ZXy23AB has not been set."}
35 # 如果变量ZXy23AB没有被设置的话，
36 #+ 那么这个脚本会打印一个错误信息，然后结束。
37
38 # 你可以自己指定错误消息。
39 # : ${variablename?"ERROR MESSAGE"}
40
41
42 # 等价于： dummy_variable=${ZXy23AB?}
43 #           dummy_variable=${ZXy23AB?"ZXy23AB has not been set."}
44 #
45 #           echo ${ZXy23AB?} >/dev/null
46
47 # 使用命令"set -u"来比较这些检查变量是否被设置的方法。
48 #

```

```

49
50
51
52 echo "You will not see this message, because script already terminated."
53
54 HERE=0
55 exit $HERE # 不会在这里退出.
56
57 # 事实上, 这个脚本将会以返回值1作为退出状态(echo $?).

```

例子9-16. 参数替换和"usage"消息(译者注: 通常就是帮助信息)

```

1#!/bin/bash
2# usage-message.sh
3
4: ${1?"Usage: $0 ARGUMENT"}
5# 如果没有提供命令行参数的话, 那么脚本就在这里退出了,
6#+ 并且打印如下错误消息.
7#     usage-message.sh: 1: Usage: usage-message.sh ARGUMENT
8
9echo "These two lines echo only if command-line parameter given."
10echo "command line parameter = \"$1\""
11
12exit 0 # 如果提供了命令行参数, 那么脚本就会在这里退出.
13
14# 分别检查有命令行参数时和没有命令行参数时, 脚本的退出状态.
15# 如果有命令行参数, 那么 "$?" 就是0.
16# 如果没有的话, 那么 "$?" 就是1.

```

参数替换与(或)扩展. 下边这些表达式都是对如何在expr字符串操作中进行match的补充. 参考[例子12-9](#)). 这些特定的使用方法一般都用来解析文件所在的目录名.

变量长度/子串删除

`#{#var}`

字符串长度(变量\$var得字符个数). 对于array来说, `#{#array}`表示的是数组中第一个元素的长度.

► 例外情况:

- `#{#*}`和`#{#@}`表示位置参数的个数.
- 对于数组来说, `#{#array[*]}`和`#{#array[@]}`表示数组中元素的个数.

例子9-17. 变量长度

```

1#!/bin/bash
2# length.sh
3
4E_NO_ARGS=65
5

```

```

6 if [ $# -eq 0 ] # 这个演示脚本必须有命令行参数.
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, 让我们试试在变量中嵌入一个空格.
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}`

从变量\$var的开头删除最短或最长匹配\$Pattern的子串. (译者注: 这是一个很常见的用法, 请读者牢记, 一个”#”表示匹配最短, ”##”表示匹配最长.)

例子A-7中的一个用法示例:

```

1 # 摘自例子"days-between.sh"的一个函数.
2 # 去掉传递进来参数开头的0.
3
4 strip_leading_zero () # 去掉从参数中传递进来的,
5 {                      #+ 可能存在的开头的0(也可能有多个0).
6     return=${1#0}        # "1"表示的是"$1" -- 传递进来的参数.
7 }                      # "0"就是我们想从"$1"中删除的子串 -- 去掉零.
8 \end{everbatim}
9
10 下边是Manfred Schwab给出的一个更加详细的例子:
11
12 \begin{everbatim}
13 strip_leading_zero2 () # 去掉开头可能存在的0(也可能有多个0),
14                     # 因为如果不取掉的话,
15 {                   # Bash就会把这个值当作8进制的值来解释.
16     shopt -s extglob    # 打开扩展的通配(globbing).
17     local val=${1##+(0)} # 使用局部变量, 匹配最长连续的一个或多个0.
18     shopt -u extglob    # 关闭扩展的通配(globbing).
19     _strip_leading_zero2=${val:-0}
20                     # 如果输入为0, 那么返回0来代替"".
21 }

```

另一个用法示例:

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

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

从变量\$var的结尾删除最短或最长匹配\$Pattern的子串. (译者注: 这是一个很常见的用法, 请读者牢记, 一个”%”表示匹配最短, ”%%”表示匹配最长.)

Bash的[版本2](#)添加了一些额外选项.

例子9-18. 参数替换中的模式匹配

```
1#!/bin/bash
2# patt-matching.sh
3
4# 使用# ## % %%来进行参数替换操作的模式匹配.
5
6var1=abcd12345abc6789
7pattern1=a*c  # *(通配符)匹配a - c之间的任意字符.
8
9echo
10echo "var1 = $var1"          # abcd12345abc6789
11echo "var1 = ${var1}"         # abcd12345abc6789
12                           # (另一种形式)
13echo "Number of characters in ${var1} = ${#var1}"
14echo
15
16echo "pattern1 = $pattern1"   # a*c  (匹配'a'到'c'之间的任意字符)
17echo "-----"
18echo '${var1#$pattern1}' = "${var1#$pattern1}"      # d12345abc6789
19# 最短的可能匹配, 去掉abcd12345abc6789的前3个字符.
20#           | - |           ^^^^^^
21echo '${var1##$pattern1}' = "${var1##$pattern1}"      # 6789
22# 最长的可能匹配, 去掉abcd12345abc6789的前12个字符
23#           | ----- |       ^^^^^^
24
25echo; echo; echo
26
27pattern2=b*9                # 匹配'b'到'9'之间的任意字符
28echo "var1 = $var1"          # 还是abcd12345abc6789
```

```

29 echo
30 echo "pattern2 = $pattern2"
31 echo "-----"
32 echo '${var1%pattern2} =' "${var1%\${pattern2}}"      # abcd12345a
33 # 最短的可能匹配, 去掉abcd12345abc6789的最后6个字符
34 #           |----| ~~~~~~|
35 echo '${var1%%pattern2} =' "${var1%\${pattern2}}"      # a
36 # 最长的可能匹配, 去掉abcd12345abc6789的最后12个字符
37 #           |-----| ~~~~~~|
38
39 # 牢记, #和##是从字符串左边开始, 并且去掉左边的字符串,
40 #       %和%%从字符串的右边开始, 并且去掉右边的字符串.
41 # (译者注: 有个好记的方法, 那就是察看键盘顺序, 记住#在%的左边. ^_^)
42 echo
43
44 exit 0

```

例子9-19. 修改文件扩展名:

```

1#!/bin/bash
2# rfe.sh: 修改文件扩展名.
3#
4# 用法:          rfe old_extension new_extension
5#
6# 示例:
7# 将指定目录中所有的*.gif文件都重命名为*.jpg,
8# 用法:          rfe gif jpg
9
10
11 E_BADARGS=65
12
13 case $# in
14   0|1)          # 竖线"|"在这里表示"或"操作.
15   echo "Usage: `basename $0` old_file_suffix new_file_suffix"
16   exit $E_BADARGS # 如果只有0个或1个参数的话, 那么就退出脚本.
17 ;;
18 esac
19
20
21 for filename in *.$1
22 # 以第一个参数为扩展名的全部文件的列表.
23 do
24   mv $filename ${filename%$1}$2
25   # 把筛选出来的文件的扩展名去掉,
26   # 因为筛选出来的文件的扩展名都是第一个参数,
27   #+ 然后把第2个参数作为扩展名, 附加到这些文件的后边.

```

```
28 done  
29  
30 exit 0
```

变量扩展/子串替换

这些结构都是从ksh中引入的.

`${var:pos}`

变量var从位置pos开始扩展(译者注: 也就是pos之前的字符都丢弃).

`${var:pos:len}`

变量var从位置pos开始, 并扩展len个字符. 参考例子A-14, 这个例子展示了这种操作的一个创造性的用法.

`${var/Pattern/Replacement}`

使用Replacement来替换变量var中第一个匹配Pattern的字符串.

如果省略Replacement, 那么第一个匹配Pattern的字符串将被替换为空, 也就是被删除了.

`${var//Pattern/Replacement}`

全局替换. 所有在变量var匹配Pattern的字符串, 都会被替换为Replacement.

和上边一样, 如果省略Replacement, 那么所有匹配Pattern的字符串, 都将被替换为空, 也就是被删除掉.

例子9-20. 使用模式匹配来解析任意字符串

```
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#*-} 也一样,  
9 #+ 因为#匹配最短的字符串,  
10 #+ 同时*匹配任意前缀, 包括空字符串.  
11 # (感谢, Stephane Chazelas, 指出这点.)  
12  
13 t=${var1##*-}  
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 # 在这个特例中, 与t='basename $path_name'          效果相同.
29 # t=${path_name%/}; t=${t##*/}    是更一般的解决方法.
30 #+ 但有时还是会失败.
31 # 如果$path_name以一个换行符结尾的话,
32 # 那么 'basename $path_name' 就不能正常工作了,
33 #+ 但是上边的表达式可以.
34 # (感谢, S.C.)
35
36 t=${path_name%/*.*}
37 # 与t='dirname $path_name' 效果相同.
38 echo "path_name, stripped of suffixes = $t"
39 # 在某些情况下将失效, 比如 "../", "/foo////", # "foo/", "/".
40 # 删除后缀, 尤其是在basename没有后缀的情况下,
41 #+ 但是dirname可以, 不过这同时也使问题复杂化了.
42 # (感谢, S.C.)
43
44 echo
45
46 t=${path_name:11}
47 echo "$path_name, with first 11 chars stripped off = $t"
48 t=${path_name:11:5}
49 echo "$path_name, with first 11 chars stripped off, length 5 = $t"
50
51 echo
52
53 t=${path_name/bozo/clown}
54 echo "$path_name with \"bozo\" replaced by \"clown\" = $t"
55 t=${path_name/today/}
56 echo "$path_name with \"today\" deleted = $t"
57 t=${path_name//o/O}
58 echo "$path_name with all o's capitalized = $t"
59 t=${path_name//o/}
60 echo "$path_name with all o's deleted = $t"
61
62 exit 0

```

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

如果变量var的前缀匹配Pattern, 那么就使用Replacement来替换匹配到Pattern的字符串.

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

如果变量var的后缀匹配Pattern, 那么就使用Replacement来替换匹配到Pattern的字符串.

例子9-21. 对字符串的前缀和后缀使用匹配模式

```

1#!/bin/bash
2# var-match.sh:

```

```

3 # 对字符串的前缀和后缀进行模式替换的一个演示.
4
5 v0=abc1234zip1234abc      # 变量原始值.
6 echo "v0 = $v0"           # abc1234zip1234abc
7 echo
8
9 # 匹配字符串的前缀(开头).
10 v1=${v0/#abc/ABCDEF}     # abc1234zip1234abc
11                                # |-|
12 echo "v1 = $v1"           # ABCDEF1234zip1234abc
13                                # |----|
14
15 # 匹配字符串的后缀(结尾).
16 v2=${v0/%abc/ABCDEF}     # abc1234zip123abc
17                                #           |-|
18 echo "v2 = $v2"           # abc1234zip1234ABCDEF
19                                #           |----|
20
21 echo
22
23 # -----
24 # 必须匹配字符串的开头或结尾,
25 #+ 否则是不会产生替换结果的.
26 #
27 v3=${v0/#123/000}        # 匹配, 但不是在开头.
28 echo "v3 = $v3"          # abc1234zip1234abc
29                                # 不会发生替换.
30 v4=${v0/%123/000}        # 匹配, 但不是在结尾.
31 echo "v4 = $v4"          # abc1234zip1234abc
32                                # 不会发生替换.
33
34 exit 0

```

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

匹配所有之前声明过的, 并且以varprefix开头的变量.

```

1 xyz23=whatever
2 xyz24=
3
4 a=${!xyz*}      # 展开所有以"xyz"开头的, 并且之前声明过的变量名.
5 echo "a = $a"    # a = xyz23 xyz24
6 a=${!xyz@}      # 同上.
7 echo "a = $a"    # a = xyz23 xyz24
8
9 # Bash, 版本2.04, 添加了这个功能.

```

注意事项

1. 如果在一个非交互脚本中, \$parameter被设置为null的话, 那么这个脚本将会返回127作为退出状态码 (127返回码对应的Bash错误码为命令未发现”command not found”).

4 指定变量的类型: 使用declare或者typeset

declare或者typeset [内建命令](#)(这两个命令是完全一样的)允许指定变量的具体类型. 在某些编程语言中, 这是指定变量类型的一种很弱的形式. declare命令是从Bash 2.0之后才被引入的命令. typeset也可以用在ksh的脚本中.

declare/typeset选项

-r 只读

```
1 declare -r var1
```

(declare -r var1与readonly var1是完全一样的)

这和C语言中的const关键字一样, 都用来指定变量为只读. 如果你尝试修改一个只读变量的值, 那么会产生错误信息.

-i 整型

```
1 declare -i number
2 # 脚本将会把变量"number"按照整型进行处理.
3
4 number=3
5 echo "Number = $number"      # Number = 3
6
7 number=three
8 echo "Number = $number"      # Number = 0
9 # 脚本尝试把字符串"three"作为整数来求值(译者注:当然会失败,所以出现值为0).
```

如果把一个变量指定为整型的话, 那么即使没有expr或者let命令, 也允许使用特定的算术运算.

```
1 n=6/3
2 echo "n = $n"      # n = 6/3
3
4 declare -i n
5 n=6/3
6 echo "n = $n"      # n = 2
```

-a 数组

```
1 declare -a indices
```

变量indices将被视为数组.

-f 函数

```
1 declare -f
```

如果在脚本中使用declare -f, 而不加任何参数的话, 那么将会列出这个脚本之前定义的所有函数.

```
1 declare -f function_name
```

如果在脚本中使用declare -f function_name这种形式的话, 将只会列出这个函数的名字.

-x export

```
1 declare -x var3
```

这句将会声明一个变量，并作为这个脚本的环境变量被导出.

```
-x var=$value
```

```
1 declare -x var3=373
```

declare命令允许在声明变量类型的同时给变量赋值.

例子9-22. 使用declare来指定变量的类型

```
1#!/bin/bash
2
3func1 ()
4{
5echo This is a function.
6}
7
8declare -f      # 列出前面定义的所有函数.
9
10echo
11
12declare -i var1  # var1是个整型变量.
13var1=2367
14echo "var1 declared as $var1"
15var1=var1+1    # 整型变量的声明并不需要使用'let'命令.
16echo "var1 incremented by 1 is $var1."
17# 尝试修改一个已经声明为整型变量的值.
18echo "Attempting to change var1 to floating point value, 2367.1."
19var1=2367.1     # 产生错误信息，并且变量并没有被修改.
20echo "var1 is still $var1"
21
22echo
23
24declare -r var2=13.36      # 'declare' 允许设置变量的属性,
25                                #+ 同时给变量赋值.
26echo "var2 declared as $var2" # 试图修改只读变量的值.
27var2=13.37                # 产生错误消息，并且从脚本退出.
28
29echo "var2 is still $var2"   # 将不会执行到这行.
30
31exit 0                      # 脚本也不会从此处退出.
```

⑧ 使用declare内建命令可以限制变量的作用域.

```
1foo ()
2{
3FOO="bar"
4}
5
6bar ()
7{
```

```
8 foo
9 echo $FOO
10 }
11
12 bar # 打印bar.
```

然而. . .

```
1 foo (){
2 declare FOO="bar"
3 }
4
5 bar ()
6 {
7 foo
8 echo $FOO
9 }
10
11 bar # 什么都不打印.
12
13
14 # 感谢, Michael Iatrou, 指出这点.
```

5 变量的间接引用

假设一个变量的值是第二个变量的名字。那么我们如何从第一个变量中取得第二个变量的值呢？比如，如果a=letter_of_alphabet并且letter_of_alphabet=z，那么我们能够通过引用变量a来获得z么？这确实是能做到的，它被称为间接引用。它使用eval var1=\\$\\$var2这种不平常的形式。

例子9-23. 间接引用

```
1 #!/bin/bash
2 # ind-ref.sh: 间接变量引用。
3 # 访问一个以另一个变量内容作为名字的变量的值。(译者注：怎么译都不顺)
4
5 a=letter_of_alphabet    # 变量"a"的值是另一个变量的名字。
6 letter_of_alphabet=z
7
8 echo
9
10 # 直接引用。
11 echo "a = $a"           # a = letter_of_alphabet
12
13 # 间接引用。
14 eval a=\$\$a
15 echo "Now a = $a"       # 现在 a = z
16
17 echo
18
19
20 # 现在，让我们试试修改第二个引用的值。
21
22 t=table_cell_3
23 table_cell_3=24
24 echo "\"table_cell_3\" = $table_cell_3"          # "table_cell_3" = 24
25 echo -n "dereferenced \"t\" = "; eval echo \$\$t   # 解引用 "t" = 24
26 # 在这个简单的例子中，下面的表达式也能正常工作么(为什么？)。
27 #         eval t=\$\$t; echo "\"t\" = \$t"
28
29 echo
30
31 t=table_cell_3
32 NEW_VAL=387
33 table_cell_3=$NEW_VAL
34 echo "Changing value of \"table_cell_3\" to $NEW_VAL."
35 echo "\"table_cell_3\" now $table_cell_3"
36 echo -n "dereferenced \"t\" now "; eval echo \$\$t
37 # "eval" 带有两个参数 "echo" 和 "\$\$t" (与$table_cell_3等价)
```

```

38
39 echo
40
41 # (感谢, Stephane Chazelas, 澄清了上边语句的行为.)
42
43
44 # 另一个方法是使用${!t}符号, 见"Bash, 版本2"小节的讨论.
45 # 也请参考 ex78.sh.
46
47 exit 0

```

变量的间接引用到底有什么应用价值? 它给Bash添加了一种类似于C语言指针的功能, 比如, 在[表格查找](#)中的用法. 另外, 还有一些其他非常有趣的应用. . . .

Nils Radtke展示了如何建立“动态”变量名并取出它们的值. 当使用[source](#)命令加载配置文件的时候, 很有用.

```

1#!/bin/bash
2
3
4# -----
5# 这部分内容可以用单独文件通过使用"source"命令来单独加载.
6isdnMyProviderRemoteNet=172.16.0.100
7isdnYourProviderRemoteNet=10.0.0.10
8isdnOnlineService="MyProvider"
9# -----
10
11
12remoteNet=$(eval "echo \$$(echo isdn${isdnOnlineService}RemoteNet)")
13remoteNet=$(eval "echo \$$(echo isdnMyProviderRemoteNet)")
14remoteNet=$(eval "echo \$isdnMyProviderRemoteNet")
15remoteNet=$(eval "echo $isdnMyProviderRemoteNet")
16
17echo "$remoteNet"    # 172.16.0.100
18
19# =====
20
21# 能够做得更好.
22
23# 注意下面的脚本, 给出了变量getSparc,
24#+ 但是没有变量getIa64:
25
26chkMirrorArchs () {
27    arch="$1";
28    if [ "$(eval "echo \$${(echo get$echo -ne $arch |
29        sed 's/^(\.).*/\1/g' | tr 'a-z' 'A-Z'; echo $arch |
30        sed 's/^(\.*\)/\1/g')):-false}")" = true ]

```

```

31 then
32     return 0;
33 else
34     return 1;
35 fi;
36 }
37
38 getSparc="true"
39 unset getIa64
40 chkMirrorArchs sparc
41 echo $?      # 0
42             # True
43
44 chkMirrorArchs Ia64
45 echo $?      # 1
46             # False
47
48 # 注意:
49 # -----
50 # 变量名中由替换命令产生的部分被准确地生成了.
51 # chkMirrorArchs函数的参数全都是小写字母.
52 # 新产生的变量名由两部分组成: "get"和"Sparc" . . .
53 # (译者注: 此处是将chkMirrorArchs函数参数的第一个字母转为大写,
54 # 然后与"get"组合形成新的变量名.)

```

例子9-24. 传递一个间接引用给awk

```

1#!/bin/bash
2
3# 这是"求文件中指定列的总和"脚本的另一个版本,
4#+ 这个脚本可以计算目标文件中指定列(此列的内容必须都是数字)所有数字的和.
5# 这个脚本使用了间接引用.
6
7ARGS=2
8E_WRONGARGS=65
9
10if [ $# -ne "$ARGS" ] # 检查命令行参数的个数是否合适.
11then
12    echo "Usage: `basename $0` filename column-number"
13    exit $E_WRONGARGS
14fi
15
16filename=$1
17column_number=$2
18
19===== 在这一行上边的部分, 与原始脚本是相同的 =====#

```

```
20
21
22 # 多行的awk脚本的调用方法为: awk '.....'
23
24
25 # awk脚本开始.
26 # -----
27 awk "
28
29 { total += \$\$column_number} # 间接引用
30 }
31 END {
32     print total
33 }
34
35 " "$filename"
36 # -----
37 # awk脚本结束.
38
39 # 间接变量引用避免了在一个内嵌awk脚本中引用shell变量的混乱行为.
40 # 感谢, Stephane Chazelas.
41
42
43 exit 0
```

⑧ 这种使用间接引用的方法是一个小技巧. 如果第二个变量更改了它的值, 那么第一个变量必须被适当的解除引用(就像上边的例子一样). 幸运的是, 在Bash版本2中引入的\${!variable}形式使得使用间接引用更加直观了. (参考[例子34-2](#)和[例子A-23](#)).

Bash并不支持指针运算操作, 因此这极大的限制了间接引用的使用. 事实上, 在脚本语言中, 间接引用是一个蹩脚的东西.

6 \$RANDOM: 产生随机整数

\$RANDOM是Bash的内部[函数](#) (并不是常量), 这个函数将返回一个伪随机[\[1\]](#) 整数, 范围在0 - 32767之间. 它不应该被用来产生密匙.

例子9-25. 产生随机整数

```
1 #!/bin/bash
2
3 # 每次调用$RANDOM都会返回不同的随机整数.
4 # 一般范围为: 0 - 32767 (有符号的16-bit整数).
5
6 MAXCOUNT=10
7 count=1
8
9 echo
10 echo "$MAXCOUNT random numbers:"
11 echo "-----"
12 while [ "$count" -le $MAXCOUNT ]      # 产生10 ($MAXCOUNT)个随机整数.
13 do
14     number=$RANDOM
15     echo $number
16     let "count += 1"  # 增加计数.
17 done
18 echo "-----"
19
20 # 如果你需要在特定范围内产生随机整数, 那么使用'modulo'(模)操作.
21 # (译者注: 事实上, 这不是一个非常好的办法. 理由见man 3 rand)
22 # 取模操作会返回除法的余数.
23
24 RANGE=500
25
26 echo
27
28 number=$RANDOM
29 let "number %= $RANGE"
30 #      ^^
31 echo "Random number less than $RANGE --- $number"
32
33 echo
34
35
36
37 # 如果你需要产生一个大于某个下限的随机整数.
38 #+ 那么建立一个test循环来丢弃所有小于此下限值的整数.
39
```

```

40 FLOOR=200
41
42 number=0    #初始化
43 while [ "$number" -le $FLOOR ]
44 do
45     number=$RANDOM
46 done
47 echo "Random number greater than $FLOOR --- $number"
48 echo
49
50 # 让我们对上边的循环尝试一个小改动，如下：
51 #         let "number = $RANDOM + $FLOOR"
52 # 这将不再需要那个while循环，并且能够运行的更快。
53 # 但是，这可能会产生一个问题，思考一下是什么问题？
54
55
56
57 # 结合上边两个例子，来在指定的上下限之间来产生随机数。
58 number=0    #initialize
59 while [ "$number" -le $FLOOR ]
60 do
61     number=$RANDOM
62     let "number %= $RANGE"  # 让$number依比例落在$RANGE的范围内。
63 done
64 echo "Random number between $FLOOR and $RANGE --- $number"
65 echo
66
67
68
69 # 产生二元值，就是，"true" 或 "false" 两个值。
70 BINARY=2
71 T=1
72 number=$RANDOM
73
74 let "number %= $BINARY"
75 # 注意 let "number >>= 14"    将会给出一个更好的随机分配。
76 # (译者注：正如man页中提到的，更高位的随机分布更加平均)
77 #+ (右移14位将把所有的位全部清空，除了第15位，因为有符号，
78 # 第16位是符号位). #取模操作使用低位来产生随机数会相对不平均)
79 if [ "$number" -eq $T ]
80 then
81     echo "TRUE"
82 else
83     echo "FALSE"
84 fi

```

```

85
86 echo
87
88
89 # 抛骰子.
90 SPOTS=6    # 模6给出的范围是0 - 5.
91          # 加1会得到期望的范围1 - 6.
92          # 感谢, Paulo Marcel Coelho Aragao, 对此进行的简化.
93 die1=0
94 die2=0
95 # 是否让SPOTS=7会比加1更好呢? 解释行或者不行的原因?
96
97 # 每次抛骰子, 都会给出均等的机会.
98
99     let "die1 = $RANDOM % $SPOTS +1" # 抛第一次.
100    let "die2 = $RANDOM % $SPOTS +1" # 抛第二次.
101    # 上边的算术操作中, 哪个具有更高的优先级呢 --
102    #+ 模(%) 还是加法操作(+)?
103
104
105 let "throw = $die1 + $die2"
106 echo "Throw of the dice = $throw"
107 echo
108
109
110 exit 0

```

例子9-26. 从一幅扑克牌中取出一张随机的牌

```

1#!/bin/bash
2# pick-card.sh
3
4# 这是一个从数组中取出随机元素的一个例子.
5
6
7# 抽取一张牌, 任何一张.
8
9Suites="Clubs
10Diamonds
11Hearts
12Spades"
13
14Denominations="2
153
164
175

```

```

18 6
19 7
20 8
21 9
22 10
23 Jack
24 Queen
25 King
26 Ace"
27
28 # 注意变量的多行展开.
29
30
31 suite=($Suites)           # 读入一个数组.
32 denomination=($Denominations)
33
34 num_suites=${#suite[*]}      # 计算有多少个数组元素.
35 num_denominations=${#denomination[*]}
36
37 echo -n "${denomination[$((RANDOM%num_denominations))]} of "
38 echo ${suite[$((RANDOM%num_suites))]}
39
40
41 # $bozo sh pick-cards.sh
42 # Jack of Clubs
43
44
45 # 感谢, "jipe," 指出$RANDOM的这个用法.
46 exit 0

```

Jipe展示了一套技巧来在一个指定范围内产生随机数.

```

1 # 在6 到 30之间产生随机数.
2 rnumber=$((RANDOM%25+6))
3
4 # 还是在6 - 30之间产生随机数,
5 #+ 但是这个数还必须能够被3整除.
6 rnumber=$(((RANDOM%30/3+1)*3))
7
8 # 注意, 并不是在所有情况下都能正确运行.
9 # 如果$RANDOM返回0, 那么就会失败.
10
11 # Frank Wang 建议用下边的方法:
12 rnumber=$(( RANDOM%27/3*3+6 ))

```

Bill Gradwohl给出了一个改良公式, 这个公式只适用于正数.

```
1 rnumber=$((RANDOM%(max-min+divisibleBy))/divisibleBy*divisibleBy+min))
```

这里Bill展示了一个通用公式, 这个函数返回两个指定值之间的随机数.

例子9-27. 两个指定值之间的随机数

```
1#!/bin/bash
2# random-between.sh
3# 产生两个指定值之间的随机数.
4# 由Bill Gradwohl编写, 本书作者做了一些修改.
5# 脚本作者允许在这里使用.
6
7
8randomBetween() {
9# 在$min和$max之间,
10#+ 产生一个正的或负的随机数.
11#+ 并且可以被$divisibleBy所整除.
12# 给出一个合理的随机分配的返回值.
13#
14# Bill Gradwohl - Oct 1, 2003
15
16syntax() {
17# 在函数中内嵌函数
18echo
19echo "Syntax: randomBetween [min] [max] [multiple]"
20echo
21echo "Expects up to 3 passed parameters, but all are completely optional."
22echo "min is the minimum value"
23echo "max is the maximum value"
24echo "multiple specifies that the answer must be a multiple of this value."
25echo "    i.e. answer must be evenly divisible by this number."
26echo
27echo "If any value is missing, defaults area supplied as: 0 32767 1"
28echo "Successful completion returns 0, unsuccessful completion returns"
29echo "function syntax and 1."
30echo "The answer is returned in the global variable randomBetweenAnswer"
31echo "Negative values for any passed parameter are handled correctly."
32}
33
34local min=${1:-0}
35local max=${2:-32767}
36local divisibleBy=${3:-1}
37# 默认值分配, 用来处理没有参数传递进来的情况.
38
39local x
40local spread
```

```

41
42 # 确认divisibleBy是正值.
43 [ ${divisibleBy} -lt 0 ] && divisibleBy=$((0-divisibleBy))
44
45 # 完整性检查.
46 if [ $# -gt 3 -o ${divisibleBy} -eq 0 -o  ${min} -eq ${max} ]; then
47     syntax
48     return 1
49 fi
50
51 # 查看min和max是否颠倒了.
52 if [ ${min} -gt ${max} ]; then
53     # 交换它们.
54     x=${min}
55     min=${max}
56     max=$x
57 fi
58
59 # 如果min自己并不能够被$divisibleBy所整除,
60 #+ 那么就调整max的值, 使其能够被$divisibleBy所整除, 前提是不能放大范围.
61 if [ $((min/divisibleBy*divisibleBy)) -ne ${min} ]; then
62     if [ ${min} -lt 0 ]; then
63         min=$((min/divisibleBy*divisibleBy))
64     else
65         min=$((((min/divisibleBy)+1)*divisibleBy))
66     fi
67 fi
68
69 # 如果min自己并不能够被$divisibleBy所整除,
70 #+ 那么就调整max的值, 使其能够被$divisibleBy所整除, 前提是不能放大范围.
71 if [ $((max/divisibleBy*divisibleBy)) -ne ${max} ]; then
72     if [ ${max} -lt 0 ]; then
73         max=$((((max/divisibleBy)-1)*divisibleBy))
74     else
75         max=$((max/divisibleBy*divisibleBy))
76     fi
77 fi
78
79 # -----
80 # 现在, 来做点真正的工作.
81
82 # 注意, 为了得到对于端点来说合适的分配,
83 #+ 随机值的范围不得不落在
84 #+ 0 和 abs(max-min)+divisibleBy 之间, 而不是 abs(max-min)+1.
85

```

```

86 # 对于端点来说,
87 #+ 这个少量的增加将会产生合适的分配.
88
89 # 如果修改这个公式, 使用abs(max-min)+1来代替 abs(max-min)+divisibleBy的话,
90 #+ 也能够得到正确的答案, 但是在这种情况下所生成的随机值对于正好为端点倍数
91 #+ 的这种情况来说将是不完美的, 因为正好为端点倍数情况下的随机率比较低,
92 #+ 因为你才加1而已, 这比正常公式下所产生的几率要小的多(正常为加divisibleBy).
93 # -----
94
95 spread=$((max-min))
96 [ ${spread} -lt 0 ] && spread=$((0-spread))
97 let spread+=divisibleBy
98 randomBetweenAnswer=$(((RANDOM%spread)/divisibleBy*divisibleBy+min))
99
100 return 0
101
102 # 然而, Paulo Marcel Coelho Aragao 指出
103 #+ 当 $max 和 $min 不能够被$divisibleBy所整除时,
104 #+ 这个公式将会失败.
105 #
106 # 他建议使用如下公式:
107 #     rnumber = $(((RANDOM%(max-min+1)+min)/divisibleBy*divisibleBy))
108
109 }
110
111 # 让我们测试一下这个函数.
112 min=-14
113 max=20
114 divisibleBy=3
115
116
117 # 产生一个所期望的数组answers, 数组下标用来表示在范围内可能出现的值,
118 #+ 而元素内容记录的是这个值所出现的次数, 如果我们循环足够多次,
119 # 那么我们一定会得到至少一次出现机会.
120
121 declare -a answer
122 minimum=${min}
123 maximum=${max}
124 if [ $((minimum/divisibleBy*divisibleBy)) -ne ${minimum} ]; then
125     if [ ${minimum} -lt 0 ]; then
126         minimum=$((minimum/divisibleBy*divisibleBy))
127     else
128         minimum=$((((minimum/divisibleBy)+1)*divisibleBy))
129     fi
130 fi

```

```

131
132
133 # 如果max本身并不能够被$divisibleBy整除,
134 #+ 那么就调整max的值, 使其能够被$divisibleBy整除, 前提是不能放大范围.
135
136 if [ $((maximum/divisibleBy*divisibleBy)) -ne ${maximum} ]; then
137     if [ ${maximum} -lt 0 ]; then
138         maximum=$(((maximum/divisibleBy)-1)*divisibleBy)
139     else
140         maximum=$((maximum/divisibleBy*divisibleBy))
141     fi
142 fi
143
144
145 # 我们需要产生一个下标全部为正的数组.
146 #+ 所以我们需要一个displacement,
147 #+ 这样就可以保证结果都为正.
148
149 displacement=$((0-minimum))
150 for ((i=${minimum}; i<=${maximum}; i+=divisibleBy)); do
151     answer[i+displacement]=0
152 done
153
154
155 # 现在, 让我们循环足够多的次数, 来得到我们想要的答案.
156 loopIt=1000    # 脚本作者建议循环 100000 次,
157                 #+ 但是这需要的时间太长了.
158
159 for ((i=0; i<$loopIt; ++i)); do
160
161     # 注意, 我们在这里调用randomBetweenAnswer函数时, 估计将min和max颠倒顺序.
162     #+ 这是为了测试在这种情况下, 此函数是否还能正确的运行.
163
164     randomBetween ${max} ${min} ${divisibleBy}
165
166     # 如果答案不是我们所期望的, 就报错.
167     [ ${randomBetweenAnswer} -lt ${min} \
168     -o ${randomBetweenAnswer} -gt ${max} ] \
169     && echo MIN or MAX error - ${randomBetweenAnswer}!
170
171     [ $((randomBetweenAnswer%${divisibleBy})) -ne 0 ] \
172     && echo DIVISIBLE BY error - ${randomBetweenAnswer}!
173
174     # 将统计值保存到answer中.
175     answer[randomBetweenAnswer+displacement]= \

```

```

176    $((answer[randomBetweenAnswer+displacement]+1))
177 done
178
179
180
181 # 让我们来察看一下结果.
182
183 for ((i=${minimum}; i<=${maximum}; i+=divisibleBy)); do
184     [ ${answer[i+displacement]} -eq 0 ] \
185     && echo "We never got an answer of $i." \
186     || echo "${i} occurred ${answer[i+displacement]} times."
187 done
188
189
190 exit 0

```

\$RANDOM到底有多随机? 最好的方法就是编写脚本来测试一下, 跟踪\$RANDOM所产生的”随机”数的分布情况. 让我们用\$RANDOM来摇骰子. . .

例子9-28. 用随机数来摇单个骰子

```

1#!/bin/bash
2# RANDOM到底有多随机?
3
4RANDOM=$$          # 使用脚本的进程ID来作为随机数的种子.
5
6PIPS=6              # 一个骰子有6个面.
7MAXTHROWS=600       # 如果你没别的事做, 可以增加这个数值.
8throw=0             # 抛骰子的次数.
9
10ones=0             # 必须把所有的count都初始化为0,
11twos=0             #+ 因为未初始化的变量为null, 不是0.
12threes=0
13fours=0
14fives=0
15sixes=0
16
17print_result ()
18{
19echo
20echo "ones =      $ones"
21echo "twos =      $twos"
22echo "threes =    $threes"
23echo "fours =     $fours"
24echo "fives =     $fives"
25echo "sixes =     $sixes"
26echo

```

```

27 }
28
29 update_count()
30 {
31 case "$1" in
32     0) let "ones += 1";;    # 因为骰子没有"零", 所以给1.
33     1) let "twos += 1";;   # 把这个设为2, 后边也一样.
34     2) let "threes += 1";;
35     3) let "fours += 1";;
36     4) let "fives += 1";;
37     5) let "sixes += 1";;
38 esac
39 }
40
41 echo
42
43
44 while [ "$throw" -lt "$MAXTHROWS" ]
45 do
46     let "die1 = RANDOM % $PIPS"
47     update_count $die1
48     let "throw += 1"
49 done
50
51 print_result
52
53 exit 0
54
55 # 如果RANDOM是真正的随机, 那么摇出来结果应该是平均的.
56 # 把$MAXTHROWS设为600, 那么每个面应该是100, 上下的出入不应该超过20.
57 #
58 # 记住RANDOM毕竟是一个伪随机数,
59 #+ 并且不是十分完美.
60
61 # 随机数的生成是一个十分深奥并复杂的问题.
62 # 足够长的随机序列, 不但会展现其杂乱无章的一面,
63 #+ 同样也会展现其机会均等的一面.
64
65 # 练习 (很简单):
66 # -----
67 # 重写这个脚本, 做成抛1000次硬币的形式.
68 # 分为"头"和"字"两面.

```

就像我们在上边的例子中所看到的, 最好在每次产生RANDOM的时候都使用新的种子. 因为如果使用同样种子的话, 那么RANDOM将会产生相同的序列. [2] (C语言中的random()函数也

会有这样的行为.)

例子9-29. 重新分配随机数种子

```
1 #!/bin/bash
2 # seeding-random.sh: 设置RANDOM变量作为种子.
3
4 MAXCOUNT=25      # 决定产生多少个随机数.
5
6 random_numbers () {
7
8 count=0
9 while [ "$count" -lt "$MAXCOUNT" ]
10 do
11     number=$RANDOM
12     echo -n "$number "
13     let "count += 1"
14 done
15 }
16
17 echo; echo
18
19 RANDOM=1          # 为随机数的产生来设置RANDOM种子.
20 random_numbers
21
22 echo; echo
23
24 RANDOM=1          # 设置同样的种子...
25 random_numbers    # ...将会和上边产生的随机序列相同.
26
27           # 复制一个相同的"随机"序列在什么情况下有用呢?
28
29 echo; echo
30
31 RANDOM=2          # 在试一次, 但是这次使用不同的种子...
32 random_numbers    # 这次将得到一个不同的随机序列.
33
34 echo; echo
35
36 # RANDOM=$$ 使用脚本的进程ID来作为产生随机数的种子.
37 # 从 'time' 或 'date' 命令中取得RANDOM作为种子也是常用的做法.
38
39 # 一个很有想象力的方法...
40 SEED=$(head -1 /dev/urandom | od -N 1 | awk '{ print $2 }')
41 # 首先从/dev/urandom(系统伪随机设备文件)中取出一行,
42 #+ 然后将这个可打印行转换为8进制数, 使用"od"命令来转换.
43 #+ 最后使用"awk"来获得一个数,
```

```

44 #+ 这个数将作为产生随机数的种子.
45 RANDOM=$SEED
46 random_numbers
47
48 echo; echo
49
50 exit 0

```

/dev/urandom设备文件提供了一种比单独使用\$RANDOM更好的, 能够产生更加”随机”的随机数的方法. dd if=/dev/urandom of=targetfile bs=1 count=XX能够产生一个很分散的伪随机数序列. 然而, 如果想要将这个数赋值到一个脚本文件的变量中, 还需要可操作性, 比如使用od命令(就像上边的例子, 还有[例子12-13](#)), 或者使用dd命令(参见[例子12-55](#)), 或者通过管道传递到md5sum命令中(参见[例子33-14](#)).

当然还有其他的产生伪随机数的方法. awk就能提供一个方便的方法来做到这点.

例子9-30. 使用awk来产生伪随机数

```

1#!/bin/bash
2# random2.sh: 产生一个范围在 0 - 1 之间的伪随机数.
3# 使用了awk的rand()函数.
4
5AWKSCRIPT=' { srand(); print rand() } '
6#           Command(s) / 传递到awk中的参数
7# 注意, srand()是awk中用来产生伪随机数种子的函数.
8
9
10echo -n "Random number between 0 and 1 = "
11
12echo | awk "$AWKSCRIPT"
13# 如果你省去'echo', 会怎样?
14
15exit 0
16
17
18# 练习:
19# -----
20
21# 1) 使用循环结构, 打印出10个不同的随机数.
22#   (提示: 在每次循环过程中, 你必须使用"srand()"函数来生成不同的种子,
23#+   如果你不这么做会怎样?)
24
25# 2) 使用整数乘法作为一个比例因子, 在10到100的范围之间,
26#+   来产生随机数.
27
28# 3) 同上边的练习#2, 但是这次产生随机整数.

```

[date命令也可以用来产生伪随机整数序列](#).

注意事项

1. 真正的”随机事件,”在它存在的范围内, 只发生在特定的几个未知的自然界现象中, 比如放射性衰变. 计算机只能产生模拟的随机事件, 并且计算机产生的”随机”数只能称为伪随机数.
2. 计算机用来产生伪随机数的种子可以被看成是一种标识标签. 比如, 使用种子23所产生的随机序列就被称为序列#23.

一个伪随机序列的特点就是在这个序列开始重复之前的所有元素个数的总和, 也就是这个序列的长度. 一个好的伪随机产生算法可以产生一个非常长的不重复序列.

7 双圆括号结构

与let命令很相似, ((...))结构允许算术扩展和赋值. 举个简单的例子, a=\$((5 + 3)), 将把变量”a”设为”5 + 3”, 或者8. 然而, 双圆括号结构也被认为是在Bash中使用C语言风格变量操作的一种处理机制.

例子9-31. C语言风格的变量操作

```
1 #!/bin/bash
2 # 使用((...))结构操作一个变量, C语言风格的变量操作.
3
4
5 echo
6
7 (( a = 23 )) # C语言风格的变量赋值, "="两边允许有空格.
8 echo "a (initial value) = $a"
9
10 (( a++ )) # C语言风格的后置自加.
11 echo "a (after a++) = $a"
12
13 (( a-- )) # C语言风格的后置自减.
14 echo "a (after a--) = $a"
15
16
17 (( ++a )) # C语言风格的前置自加.
18 echo "a (after ++a) = $a"
19
20 (( --a )) # C语言风格的前置自减.
21 echo "a (after --a) = $a"
22
23 echo
24 #####
25 # 注意: 就像在C语言中一样, 前置或后置自减操作
26 #+ 会产生一些不同的副作用.
27
28
29 n=1; let --n && echo "True" || echo "False" # False
30 n=1; let n-- && echo "True" || echo "False" # True
31
32 # 感谢, Jeroen Domburg.
33 #####
34
35 echo
36
37 (( t = a<45?7:11 )) # C语言风格的三元操作.
38 echo "If a < 45, then t = 7, else t = 11."
```

```
39 echo "t = $t "          # Yes!
40
41 echo
42
43
44 # -----
45 # 复活节彩蛋!
46 # -----
47 # Chet Ramey显然偷偷摸摸的将一些未公开的C语言风格的结构
48 #+ 引入到了Bash中(事实上是从ksh中引入的,这更接近些).
49 # 在Bash的文档中,Ramey将((...))称为shell算术运算,
50 #+ 但是它所能做的远远不止于此.
51 # 不好意思,Chet,现在秘密被公开了.
52
53 # 你也可以参考一些"for"和"while"循环中使用((...))结构的例子.
54
55 # 这些只能够在Bash 2.04或更高版本的Bash上才能运行.
56
57 exit 0
```

参见[例子10-12](#).

第10章 循环与分支

本章目录

1. [循环](#)
 2. [嵌套循环](#)
 3. [循环控制](#)
 4. [测试与分支（case与select结构）](#)
-

对代码块的操作是构造和组织shell脚本的关键。循环和分支结构为脚本编程提供了操作代码块的工具。

1 循环

循环就是迭代(重复)一些命令的代码块, 如果循环控制条件不满足的话, 就结束循环.

for .

for arg **in** [list]

 这是一个基本的循环结构. 它与C语言中的for循环结构有很大的不同.

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

在循环的每次执行中, arg将顺序的访问list中列出的变量.

```
1 for arg in "$var1" "$var2" "$var3" ... "$varN"
2 # 在第1次循环中, arg = $var1
3 # 在第2次循环中, arg = $var2
4 # 在第3次循环中, arg = $var3
5 # ...
6 # 在第N此循环中, arg = $varN
7
8 # 在[list]中的参数加上双引号是为了阻止单词分割.
```

list中的参数允许包含通配符.

如果do和for想在同一行中出现, 那么在它们之间需要添加一个分号.

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

例子10-1. 一个简单的for循环

```
1 #!/bin/bash
2 # 列出所有的行星名称. (译者注: 现在的太阳系行星已经有了变化^_^)
3 # (PDFer再注: 貌似是冥王星被踢了:(, 感觉被骗了呀。)
4
5 for planet in Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune Pluto
6 do
7     echo $planet # 每个行星都被单独打印在一行上.
8 done
9
10 echo
11
12 for planet in "Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune Pluto"
13 # 所有的行星名称都打印在同一行上.
14 # 整个'list'都被双引号封成了一个变量.
15 do
16     echo $planet
17 done
18
```

```
19 exit 0
```

每个[list]中的元素都可能包含多个参数. 在处理参数组时, 这是非常有用的. 在这种情况下, 使用set命令(参见例子11-15)来强制解析每个[list]中的元素, 并且将每个解析出来的部分都分配到一个位置参数中.

例子10-2. 每个[list]元素中都带有两个参数的for循环

```
1 #!/bin/bash
2 # 还是行星.
3
4 # 用行星距太阳的距离来分配行星的名字.
5
6 for planet in "Mercury 36" "Venus 67" "Earth 93" "Mars 142" "Jupiter 483"
7 do
8     set -- $planet # 解析变量"planet"并且设置位置参数.
9     # "--" 将防止$planet为空, 或者是以一个破折号开头.
10
11    # 可能需要保存原始的位置参数, 因为它们被覆盖了.
12    # 一种方法就是使用数组.
13    #         original_params=("$@")
14
15    echo "$1           $2,000,000 miles from the sun"
16    #-----two tabs---把后边的0和2连接起来
17 done
18
19 # (感谢, S.C., 对此问题进行的澄清.)
20
21 exit 0
```

可以将一个变量放在for循环的[list]位置上.

例子10-3. 文件信息: 对包含在变量中的文件列表进行操作

```
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/badblocks
9 /sbin/ypbind"      # 这是你所关心的文件列表.
10                  # 扔进去一个假文件, /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                      # 继续下一个.
21 fi
22
23 ls -l $file | awk '{ print $9 "           file size: " $5 }' # 打印两个域.
24 whatis `basename $file`    # 文件信息.
25 # 注意whatis数据库需要提前建立好.
26 # 要想达到这个目的, 以root身份运行/usr/bin/makewhatis.
27 echo
28 done
29
30 exit 0

```

如果在for循环的[list]中有通配符(*和?), 那么将会发生[通配\(globbing\)](#), 也就是文件名扩展.

例子10-4. 在for循环中操作文件

```

1#!/bin/bash
2# list-glob.sh: 使用"globbing", 在for循环中产生[list]
3
4echo
5
6for file in *
7#           ^ 在表达式中识别文件名匹配时,
8#+           Bash将执行文件名扩展.
9do
10  ls -l "$file"  # 列出在$PWD(当前目录)中的所有文件.
11  # 回想一下,通配符"*"能够匹配所有文件,
12  #+# 然而在"globbing"中,是不能比配"."文件的.
13
14  # 如果没匹配到任何文件,那它将扩展成自己.
15  # 为了不让这种情况发生,那就设置nullglob选项
16  #+# (shopt -s nullglob).
17  # 感谢, S.C.
18done
19
20echo; echo
21
22for file in [jx]*
23do
24  rm -f $file  # 只删除当前目录下以"j"或"x"开头的文件.
25  echo "Removed file \"\$file\"".
26done
27
28echo

```

```
29  
30 exit 0
```

在一个for循环中忽略in [list]部分的话，将会使循环操作\$@ – 从命令行传递给脚本的位置参数。一个非常好的例子，参见[例子A-16](#)。参见[例子11-16](#)。

例子10-5. 在for循环中省略in [list]部分

```
1 #!/bin/bash  
2  
3 # 使用两种方式来调用这个脚本，一种带参数，另一种不带参数，  
4 #+ 并观察在这两种情况下，此脚本的行为。  
5  
6 for a  
7 do  
8 echo -n "$a "  
9 done  
10  
11 # 省略'in list'部分，因此循环将会操作'$@'  
12 #+ (包括空白的命令行参数列表)。  
13  
14 echo  
15  
16 exit 0
```

也可以使用命令替换来产生for循环的[list]。参见[例子12-49](#)，[例子10-10](#) 和[例子12-43](#)。

```
1 #!/bin/bash  
2 # for-loopcmd.sh: 带[list]的for循环，  
3 #+ [list]是由命令替换所产生的。  
4  
5 NUMBERS="9 7 3 8 37.53"  
6  
7 for number in `echo $NUMBERS` # for number in 9 7 3 8 37.53  
8 do  
9   echo -n "$number "  
10 done  
11  
12 echo  
13 exit 0
```

下边是一个用命令替换来产生[list]的更复杂的例子。

例子10-7. 对于二进制文件的grep替换

```
1 #!/bin/bash  
2 # bin-grep.sh: 在一个二进制文件中定位匹配字串。  
3  
4 # 对于二进制文件的"grep"替换。  
5 # 与 "grep -a" 的效果相似  
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'      # 由Paulo Marcel Coelho Aragao提出的建议.
24                 # 也就是: 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 # 输出到管道交给"grep",然后由grep命令来过滤字符串.
28 do
29     echo $word
30 done
31
32 # S.C. 指出, 行23 - 29 可以被下边的这行来代替,
33 #   strings "$2" | grep "$1" | tr -s "$IFS" '[\n*]', 
34
35
36 # 试试用 "./bin-grep.sh mem /bin/ls" 来运行这个脚本.
37
38 exit 0

```

大部分相同.

例子10-8. 列出系统上的所有用户

```

1#!/bin/bash
2# userlist.sh
3
4PASSWORD_FILE=/etc/passwd
5n=1          # User number
6
7for name in $(awk 'BEGIN{FS=":"}{print $1}' < "$PASSWORD_FILE" )
8# 域分隔 = :           ~~~~~
9# 打印出第一个域          ~~~~~
10# 从password文件中取得输入           ~~~~~

```

```

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 #30 = bozo
22
23 exit 0
24
25 # 练习:
26 # -----
27 # 一个普通用户(或者是一个普通用户运行的脚本)
28 #+ 怎么才能够读取/etc/passwd呢?
29 # 这是否是一个安全漏洞? 为什么是?为什么不是?

```

关于用命令替换来产生[list]的最后一个例子.

例子10-9. 在目录的所有文件中查找源字符串

```

1#!/bin/bash
2# findstring.sh:
3# 在一个指定目录的所有文件中查找一个特定的字符串.
4
5directory=/usr/bin/
6fstring="Free Software Foundation" # 查看哪个文件中包含FSF.
7
8for file in $( find $directory -type f -name '*' | sort )
9do
10  strings -f $file | grep "$fstring" | sed -e "s%$directory%%"
11  # 在"sed"表达式中,
12  #+ 我们必须替换掉正常的替换分隔符"/",
13  #+ 因为"/"碰巧是我们需要过滤的字符串之一.
14  # 如果不用"%"代替"/"作为分隔符,
15  # 那么这个操作将失败,并给出一个错误消息.(试一试).
16done
17
18exit 0
19
20# 练习(很简单):
21# -----
22# 转换这个脚本,用命令行参数
23#+ 代替内部用的$directory和$fstring.

```

for循环的输出也可以通过管道传递到一个或多个命令中.

例子10-10. 列出目录中所有的符号链接

```
1 #!/bin/bash
2 # symlinks.sh: 列出目录中所有的符号链接文件.
3
4
5 directory=${1-'pwd'}
6 # 如果没有其他特殊的指定,
7 #+ 默认为当前工作目录.
8 # 下边的代码块, 和上边这句等价.
9 #
10 # ARG$=1           # 需要一个命令行参数.
11 #
12 # if [ $# -ne "$ARGS" ] # 如果不是单个参数的话...
13 # then
14 #   directory='pwd'      # 当前工作目录
15 # else
16 #   directory=$1
17 # fi
18 #
19
20 echo "symbolic links in directory \\"$directory\\"
21
22 for file in "$( find $directory -type l )"    # -type l = 符号链接
23 do
24   echo "$file"
25 done | sort      # 否则的话, 列出的文件都是未经排序的.
26 # 严格意义上说, 这里并不一定非要一个循环不可.
27 #+ 因为"find"命令的输出将被扩展成一个单词.
28 # 然而, 这种方式很容易理解也很容易说明.
29
30 # 就像Dominik 'Aeneas' Schnitzer所指出的,
31 #+ 如果没将$( find $directory -type l )用""引用起来的话,
32 #+ 那么将会把一个带有空白部分的文件名拆分成
33 # 以空白分隔的两部分(文件名允许有空白).
34 # 即使这里只会取出每个参数的第一个域.
35
36 exit 0
37
38
39 # Jean Helou建议采用下边的方法:
40
41 echo "symbolic links in directory \\"$directory\\"
42 # 当前IFS的备份. 要小心使用这个值.
43 OLDIFS=$IFS
```

```

44 IFS=:
45
46 for file in $(find $directory -type l -printf "%p$IFS")
47 do      #                                     ~~~~~
48     echo "$file"
49     done|sort

```

循环的stdout可以重定向到文件中, 我们对上边的例子做了一点修改.

例子10-11. 将目录中所有符号链接文件的名字保存到一个文件中

```

1#!/bin/bash
2# symlinks.sh: 列出目录中所有的符号链接文件.
3
4OUTFILE=symlinks.list # 保存符号链接文件名的文件
5
6directory=${1-'pwd'}
7# 如果没有其他特殊的指定,
8#+ 默认为当前工作目录.
9
10
11echo "symbolic links in directory \"\$directory\" > \"\$OUTFILE"
12echo "-----" >> "\$OUTFILE"
13
14for file in "$( find \$directory -type l )"    # -type l = 符号链接
15do
16    echo "$file"
17done | sort >> "\$OUTFILE" # 循环的stdout
18#           ~~~~~          重定向到一个文件中.
19
20exit 0

```

有一种非常像C语言for循环的语法形式. 需要使用(()).

例子10-12. 一个C风格的for循环

```

1#!/bin/bash
2# 两种循环到10的方法.
3
4echo
5
6# 标准语法.
7for a in 1 2 3 4 5 6 7 8 9 10
8do
9    echo -n "$a "
10done
11
12echo; echo
13
14# +=====

```

```

15
16 # 现在, 让我们用C风格语法来做相同的事情.
17
18 LIMIT=10
19
20 for ((a=1; a <= LIMIT ; a++)) # 双圆括号, 并且"LIMIT"变量前面没有"$".
21 do
22   echo -n "$a "
23 done          # 这是一个借用'ksh93'的结构.
24
25 echo; echo
26
27 # +=====+
28
29 # 让我们使用C语言的"逗号操作符", 来同时增加两个变量的值.
30
31 for ((a=1, b=1; a <= LIMIT ; a++, b++)) # 逗号将同时进行两条操作.
32 do
33   echo -n "$a-$b "
34 done
35
36 echo; echo
37
38 exit 0

```

参考[例子26-15](#), [例子26-16](#), 和[例子A-6](#).

—

现在, 让我们来看一个“现实生活”中使用for循环的例子.

例子10-13. 在batch mode中使用efax

```

1#!/bin/bash
2# Faxing (前提是'fax'必须已经安装好).
3
4EXPECTED_ARGS=2
5E_BADARGS=65
6
7if [ $# -ne $EXPECTED_ARGS ]
8# 检查命令行参数的个数是否正确.
9then
10  echo "Usage: `basename $0` phone# text-file"
11  exit $E_BADARGS
12fi
13
14
15if [ ! -f "$2" ]
16then

```

```

17 echo "File $2 is not a text file"
18 exit $E_BADARGS
19 fi
20
21
22 fax make $2          # 从纯文本文件中创建传真格式的文件.
23
24 for file in $(ls $2.0*) # 连接转换过的文件.
25             # 在变量列表中使用通配符.
26 do
27   fil="$fil $file"
28 done
29
30 efax -d /dev/ttyS3 -o1 -t "T$1" $fil    # 干活的地方.
31
32
33 # S.C. 指出, 通过下边的命令可以省去for循环.
34 #   efax -d /dev/ttyS3 -o1 -t "T$1" $2.0*
35 # 但这并不十分具有讲解意义[嘿嘿].
36
37 exit 0

```

while .

这种结构在循环的开头判断条件是否满足, 如果条件一直满足, 那么就一直循环下去(返回0作为退出状态码). 与for循环的区别是, while循环更适合在循环次数未知的情况下使用.

```

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

```

与for循环一样, 如果想把do和条件判断放到同一行上的话, 还是需要一个分号.

while [condition] ; do

需要注意一下某种特定的while循环, 比如getopts结构, 好像和这里所介绍的模版有点脱节.

例子10-14. 简单的while循环

```

1#!/bin/bash
2
3var0=0
4LIMIT=10
5
6while [ "$var0" -lt "$LIMIT" ]
7do
8  echo -n "$var0 "          # -n 将会阻止产生新行.
9  #                      ^      空格, 数字之间的分隔.
10
11 var0='expr $var0 + 1'    # var0=$((var0+1)) 也可以.

```

```

12          # var0=$((var0 + 1)) 也可以.
13          # let "var0 += 1"    也可以.
14 done           # 使用其他的方法也行.
15
16 echo
17
18 exit 0

```

例子10-15. 另一个while循环

```

1#!/bin/bash
2
3echo
4          # 等价于:
5while [ "$var1" != "end" ]      # while test "$var1" != "end"
6do
7  echo "Input variable #1 (end to exit) "
8  read var1                  # 为什么不使用'read $var1'?
9  echo "variable #1 = $var1"   # 因为包含"#", 所以需要"""
10 # 如果输入为'end', 那么就在这里echo.
11 # 不在这里判断结束, 在循环顶判断.
12 echo
13done
14
15exit 0

```

一个while循环可以有多个判断条件. 但是只有最后一个才能够决定是否能够退出循环. 然而这里需要一种有点特殊的循环语法.

例子10-16. 多条件的while循环

```

1#!/bin/bash
2
3var1	unset
4previous=$var1
5
6while echo "previous-variable = $previous"
7  echo
8  previous=$var1
9  [ "$var1" != end ] # 纪录之前的$var1.
10 # 这个"while"中有4个条件, 但是只有最后一个能够控制循环.
11 # *最后*的退出状态就是由这最后一个条件来决定.
12do
13echo "Input variable #1 (end to exit) "
14  read var1
15  echo "variable #1 = $var1"
16done
17
18# 尝试理解这个脚本的运行过程.

```

```
19 # 这里还是有点小技巧的.  
20  
21 exit 0
```

与for循环一样, while循环也可以通过(())来使用C风格的语法. (参考[例子9-31](#)).

例子10-17. C风格的while循环

```
1#!/bin/bash  
2# wh-loopc.sh: 循环10次的"while"循环.  
3  
4LIMIT=10  
5a=1  
6  
7while [ "$a" -le $LIMIT ]  
8do  
9    echo -n "$a "  
10   let "a+=1"  
11done      # 到目前为止都没有什么令人惊奇的地方.  
12  
13echo; echo  
14  
15# +=====+  
16  
17# 现在, 重复C风格的语法.  
18  
19((a = 1))      # a=1  
20# 双圆括号允许赋值两边的空格, 就像C语言一样.  
21  
22while (( a <= LIMIT ))  # 双圆括号, 变量前边没有"$".  
23do  
24    echo -n "$a "  
25    ((a += 1))  # let "a+=1"  
26    # Yes, 看到了吧.  
27    # 双圆括号允许像C风格的语法一样增加变量的值.  
28done  
29  
30echo  
31  
32# 现在, C程序员可以在Bash中找到回家的感觉了吧.  
33  
34exit 0
```



- while循环的stdin可以使用<来重定向到一个文件.
- while循环的stdin支持管道.

until .

这个结构在循环的顶部判断条件, 并且如果条件一直为false, 那么就一直循环下去.
(与while循环相反).

```
1 until [condition-is-true]
2 do
3   command...
4 done
```

注意, until循环的条件判断在循环的顶部, 这与某些编程语言是不同的.

与for循环一样, 如果想把do和条件判断放在同一行里, 那么就需要使用分号.

until [condition-is-true] ; do

例子10-18. until循环

```
1#!/bin/bash
2
3END_CONDITION=end
4
5until [ "$var1" = "$END_CONDITION" ]
6# 在循环的顶部进行条件判断.
7do
8  echo "Input variable #1 "
9  echo "($END_CONDITION to exit)"
10 read var1
11 echo "variable #1 = $var1"
12 echo
13done
14
15exit 0
```

2 嵌套循环

嵌套循环就在一个循环中还有一个循环，内部循环在外部循环体中。在外部循环的每次执行过程中都会触发内部循环，直到内部循环执行结束。外部循环执行了多少次，内部循环就完成多少次。当然，无论是内部循环还是外部循环的break语句都会打断处理过程。

例子10-19. 嵌套循环

```
1 #!/bin/bash
2 # nested-loop.sh: 嵌套的"for"循环。
3
4 outer=1          # 设置外部循环计数。
5
6 # 开始外部循环。
7 for a in 1 2 3 4 5
8 do
9   echo "Pass $outer in outer loop."
10  echo "-----"
11  inner=1          # 重置内部循环计数。
12
13  # =====
14  # 开始内部循环。
15  for b in 1 2 3 4 5
16  do
17    echo "Pass $inner in inner loop."
18    let "inner+=1"  # 增加内部循环计数。
19  done
20  # 内部循环结束。
21  # =====
22
23  let "outer+=1"  # 增加外部循环的计数。
24  echo          # 每次外部循环之间的间隔。
25 done
26 # 外部循环结束。
27
28 exit 0
```

关于嵌套的while循环 请参考[例子26-11](#)，关于while循环中嵌套until循环的例子请参考[例子26-13](#)。

3 循环控制

影响循环行为的命令

break, continue

break和continue这两个循环控制命令[1] 与其他语言的类似命令的行为是相同的. break命令用来跳出循环, 而continue命令只会跳过本次循环, 忽略本次循环剩余的代码, 进入循环的下一次迭代.

例子10-20. break和continue命令在循环中的效果

```
1 #!/bin/bash
2
3 LIMIT=19 # 上限
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 ] || [ "$a" -eq 11 ] # 除了3和11.
15     then
16         continue      # 跳过本次循环剩余的语句.
17     fi
18
19     echo -n "$a "    # 在$a等于3和11的时候, 这句将不会执行.
20 done
21
22 # 练习:
23 # 为什么循环会打印出20?
24
25 echo; echo
26
27 echo Printing Numbers 1 through 20, but something happens after 2.
28
29 ######
30
31 # 同样的循环, 但是用'break'来代替'continue'.
32
33 a=0
34
35 while [ "$a" -le "$LIMIT" ]
36 do
```

```
37 a=$((a+1))
38
39 if [ "$a" -gt 2 ]
40 then
41     break # 将会跳出整个循环.
42 fi
43
44 echo -n "$a "
45 done
46
47 echo; echo; echo
48
49 exit 0
```

break命令可以带一个参数. 一个不带参数的break命令只能退出最内层的循环, 而break N可以退出N层循环.

例子10-21. 多层循环的退出

```
1#!/bin/bash
2# break-levels.sh: 退出循环.
3
4# "break N" 退出N层循环.
5
6for outerloop in 1 2 3 4 5
7do
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 # 试试 break 2 来看看发生什么事.
18               # (内部循环和外部循环都被"Break"了. )
19       fi
20   done
21   # -----
22
23   echo
24done
25
26echo
27
28exit 0
```

continue命令也可以象break命令一样带一个参数。一个不带参数的continue命令只会去掉本次循环的剩余代码。而continue N将会把N层循环的剩余代码都去掉，但是循环的次数不变。

例子10-22. 多层循环的continue

```
1 #!/bin/bash
2 # "continue N" 命令，将让N层的循环全部被continue.
3
4 for outer in I II III IV V           # 外部循环
5 do
6   echo; echo -n "Group $outer: "
7
8   # -----
9   for inner in 1 2 3 4 5 6 7 8 9 10 # 内部循环
10  do
11
12    if [ "$inner" -eq 7 ]
13    then
14      continue 2 # 在第2层循环上的continue，也就是"外部循环".
15          # 使用"continue"来替代这句，
16          # 然后看一下一个正常循环的行为。
17    fi
18
19    echo -n "$inner " # 7 8 9 10 将不会被echo.
20  done
21  # -----
22  # 译者注：如果在此处添加echo的话，当然也不会输出。
23 done
24
25 echo; echo
26
27 # 练习：
28 # 在脚本中放入一个有意义的"continue N".
29
30 exit 0
```

例子10-23. 在实际的任务中使用"continue N"

```
1 # Albert Reiner 给出了一个关于使用"continue N"的例子：
2 # -----
3
4 # 假定我有很多作业需要运行，这些任务要处理一些数据，
5 #+ 这些数据保存在某个目录下的文件里，文件是以预先给定的模式进行命名的。
6 #+ 有几台机器要访问这个目录，
7 #+ 我想把工作都分配给这几台不同的机器，
8 #+ 然后我一般会在每台机器里运行类似下面的代码：
9
10 while true
```

```

11 do
12   for n in .iso.*
13   do
14     [ "$n" = ".iso.opts" ] && continue
15     beta=${n#.iso.}
16     [ -r .Iso.$beta ] && continue
17     [ -r .lock.$beta ] && sleep 10 && continue
18     lockfile -r0 .lock.$beta || continue
19     echo -n "$beta: " `date`
20     run-isotherm $beta
21     date
22     ls -alF .Iso.$beta
23     [ -r .Iso.$beta ] && rm -f .lock.$beta
24     continue 2
25   done
26   break
27 done
28
29 # 在我的应用中的某些细节是很特殊的，尤其是sleep N,
30 #+ 但是更一般的模式是：
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  # 而所谓的'sleep 600'只不过是想避免程序太快结束，而达不到演示效果。
41 done
42
43 # 脚本只有在所有任务都完成之后才会停止运行，
44 #+ (包括那些运行时新添加的任务).
45 #+ 通过使用合适的lockfiles,
46 #+ 可以使几台机器协作运行而不会产生重复的处理,
47 #+ [在我的这个例子中，重复处理会使处理时间延长一倍时间,
48 #+ 因此我很想避免这个问题].
49 #+ 同样，如果每次都从头开始搜索，可以由文件名得到处理顺序.
50 #+ 当然，还有一种办法，也可以不使用'continue 2',
51 #+ 但这样的话，就不得不检查相同的任务是不是已经被完成过了,
52 #+ (而我们应该马上来找到下一个要运行的任务)
53 #+ (在演示的情况下，检查新任务前我们终止或睡眠了很长一段时间).
54 #+

```

⑧ continue N结构如果用在有意义的场合中,往往都很难理解,并且技巧性很高.所以最好的方法就是尽量避免使用它.

注意事项

1. 这两个命令是shell的[内建命令](#),而不象其他的循环命令那样,比如[while](#)和[case](#),这两个是关键字.

4 测试与分支(case与select结构)

case和select结构在技术上说并不是循环, 因为它们并不对可执行代码块进行迭代. 但是和循环相似的是, 它们也依靠在代码块顶部或底部的条件判断来决定程序的分支.

在代码块中控制程序分支

case (in) / esac

在shell中的case结构与C/C++中的switch结构是相同的. 它允许通过判断来选择代码块中多条路径中的一条. 它的作用和多个if/then/else语句的作用相同, 是它们的简化结构, 特别适用于创建菜单.

```
1 case "$variable" in
2
3     "$condition1" )
4         command...
5     ;;
6
7     "$condition2" )
8         command...
9     ;;
10
11 esac
```

- 对变量使用””并不是强制的, 因为不会发生单词分割.
- 每句测试行, 都以右小括号)来结尾.
- 每个条件判断语句块都以一对分号结尾;;.
- case块以esac (case的反向拼写)结尾.

例子10-24. 使用case

```
1#!/bin/bash
2# 测试字符串范围.
3
4echo; echo "Hit a key, then hit return."
5read Keypress
6
7case "$Keypress" in
8    [[:lower:]] ) echo "Lowercase letter";;
9    [[:upper:]] ) echo "Uppercase letter";;
10   [0-9]        ) echo "Digit";;
11   *            ) echo "Punctuation, whitespace, or other";;
12esac          # 允许字符串的范围出现在[中括号]中,
13                  #+ 或者出现在POSIX风格的[[双中括号]中.]
```

```

14
15 # 在这个例子的第一个版本中,
16 #+ 测试大写和小写字符串的工作使用的是
17 #+ [a-z] 和 [A-Z].
18 # 这种用法在某些特定场合的或某些Linux发行版中不能够正常工作.
19 # POSIX 的风格更具可移植性.
20 # 感谢Frank Wang指出了这点.

21
22 # 练习:
23 #
24 # 就像这个脚本所表现出来的, 它只允许单次的按键, 然后就结束了.
25 # 修改这个脚本, 让它能够接受重复输入,
26 #+ 报告每次按键, 并且只有在"X"被键入时才结束.
27 # 暗示: 将这些代码都用"while"循环圈起来.

28
29 exit 0

```

例子10-25. 使用case来创建菜单

```

1#!/bin/bash
2
3# 未经处理的地址资料
4
5clear # 清屏.
6
7echo "          Contact List"
8echo "          ----- ----"
9echo "Choose one of the following persons:"
10echo
11echo "[E]vans, Roland"
12echo "[J]ones, Mildred"
13echo "[S]mith, Julie"
14echo "[Z]ane, Morris"
15echo
16
17read person
18
19case "$person" in
20# 注意, 变量是被""引用的.
21
22"E" | "e" )
23# 接受大写或者小写输入.
24echo
25echo "Roland Evans"
26echo "4321 Floppy Dr."
27echo "Hardscrabble, CO 80753"

```

```

28 echo "(303) 734-9874"
29 echo "(303) 734-9892 fax"
30 echo "revans@zyy.net"
31 echo "Business partner & old friend"
32 ;;
33 # 注意, 每个选项后边都要以双分号;;结尾.
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 # 后边的 Smith 和 Zane 的信息在这里就省略了.
48
49     * )
50 # 默认选项.
51 # 空输入(敲回车RETURN), 也适用于这里.
52 echo
53 echo "Not yet in database."
54 ;;
55
56 esac
57
58 echo
59
60 # 练习:
61 #
62 # 修改这个脚本, 让它能够接受多个输入,
63 #+ 并且能够显示多个地址.
64
65 exit 0

```

一个case的非常聪明的用法, 用来测试命令行参数.

```

1 #! /bin/bash
2
3 case "$1" in
4 "") echo "Usage: ${0##*/} <filename>"; exit $E_PARAM;; # 没有命令行参数,
5 # 或者第一个参数为空.

```

```

6 # 注意: ${0##*/} 是 ${var##pattern} 的一种替换形式. 得到的结果为$0.
7
8 -*) FILENAME=./$1;;      # 如果传递进来的文件名参数($1)以一个破折号开头,
9      ##+ 那么用./$1来代替.
10     ##+ 这样后边的命令将不会把它作为一个选项来解释.
11
12 * ) FILENAME=$1;;      # 否则, $1.
13 esac

```

这是一个命令行参数处理的更容易理解的例子:

```

1#!/bin/bash
2
3
4while [ $# -gt 0 ]; do      # 直到你用完所有的参数 . .
5  case "$1" in
6    -d|--debug)
7      # 是 "-d" 或 "--debug" 参数?
8      DEBUG=1
9      ;;
10   -c|--conf)
11     CONFFILE="$2"
12     shift
13     if [ ! -f $CONFFILE ]; then
14       echo "Error: Supplied file doesn't exist!"
15       exit $E_CONFFILE      # 错误: 文件未发现.
16     fi
17     ;;
18  esac
19  shift      # 检查剩余的参数.
20 done
21
22 # 来自 Stefano Falsetto 的 "Log2Rot" 脚本,
23 #+ 并且是他的"rottlog"包的一部分.
24 # 已得到使用许可.

```

例子10-26. 使用命令替换来产生case变量

```

1#!/bin/bash
2# case-cmd.sh: 使用命令替换来产生"case"变量.
3
4case $( arch ) in      # "arch" 返回机器体系的类型.
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结构也可以过滤[通配\(globbing\)](#)模式的字符串.

例子10-27. 简单的字符串匹配

```
1 #!/bin/bash  
2 # match-string.sh: 简单的字符串匹配  
3  
4 match_string ()  
5 {  
6     MATCH=0  
7     NOMATCH=90  
8     PARAMS=2      # 此函数需要2个参数.  
9     BAD_PARAMS=91  
10  
11     [ $# -eq $PARAMS ] || return $BAD_PARAMS  
12  
13     case "$1" in  
14         "$2") return $MATCH;;  
15         *) return $NOMATCH;;  
16     esac  
17  
18 }  
19  
20  
21 a=one  
22 b=two  
23 c=three  
24 d=two  
25  
26  
27 match_string $a      # 参数个数错误.  
28 echo $?                # 91  
29  
30 match_string $a $b    # 不匹配  
31 echo $?                # 90  
32  
33 match_string $b $d    # 匹配  
34 echo $?                # 0  
35  
36  
37 exit 0
```

例子10-28. 检查输入字符是否为字母

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

```

2 # isalpha.sh: 使用"case"结构来过滤字符串.
3
4 SUCCESS=0
5 FAILURE=-1
6
7 isalpha () # 检查输入的 *第一个字符* 是不是字母表上的字符.
8 {
9 if [ -z "$1" ] # 没有参数传进来?
10 then
11     return $FAILURE
12 fi
13
14 case "$1" in
15 [a-zA-Z]*) return $SUCCESS;; # 以一个字母开头?
16 *) return $FAILURE;;
17 esac
18 } # 同C语言的"isalpha ()"函数比较一下.
19
20
21 isalpha2 () # 测试 *整个字符串* 是否都是字母表上的字符.
22 {
23 [ $# -eq 1 ] || return $FAILURE
24
25 case $1 in
26 *[!a-zA-Z]*|"") return $FAILURE;;
27 *) return $SUCCESS;;
28 esac
29 }
30
31 isdigit () # 测试 *整个字符串* 是否都是数字.
32 { # 换句话说, 就是测试一下是否是整数变量.
33 [ $# -eq 1 ] || return $FAILURE
34
35 case $1 in
36 *[!0-9]*|"") return $FAILURE;;
37 *) return $SUCCESS;;
38 esac
39 }
40
41
42
43 check_var () # 测试isalpha().
44 {
45 if isalpha "$@"
46 then

```

```
47 echo "\"$*\\" begins with an alpha character."
48 if isalpha2 "$@"
49 then      # 不需要测试第一个字符是否是non-alpha.
50     echo "\"$*\\" contains only alpha characters."
51 else
52     echo "\"$*\\" contains at least one non-alpha character."
53 fi
54 else
55     echo "\"$*\\" begins with a non-alpha character."
56             # 如果没有参数传递进来, 也是"non-alpha".
57 fi
58
59 echo
60
61 }
62
63 digit_check () # 测试isdigit().
64 {
65 if isdigit "$@"
66 then
67     echo "\"$*\\" contains only digits [0 - 9]."
68 else
69     echo "\"$*\\" has at least one non-digit character."
70 fi
71
72 echo
73
74 }
75
76 a=23skidoo
77 b=H3ll0
78 c=-What?
79 d=What?
80 e='echo $b'    # 命令替换.
81 f=AbcDef
82 g=27234
83 h=27a34
84 i=27.34
85
86 check_var $a
87 check_var $b
88 check_var $c
89 check_var $d
90 check_var $e
91 check_var $f
```

```

92 check_var      # 没有参数传递进来, 将会发生什么?
93 #
94 digit_check $g
95 digit_check $h
96 digit_check $i
97
98
99 exit 0          # S.C改进了这个脚本.
100
101 # 练习:
102 # -----
103 # 编写一个'isfloat ()', 函数来测试浮点数.
104 # 提示: 这个函数基本上与'isdigit ()', 相同,
105 #+ 但是要添加一些小数点部分的处理.

```

select

select结构是建立菜单的另一种工具, 这种结构是从ksh中引入的.

```

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

```

提示用户输入选择的内容(比如放在变量列表中). 注意: select命令使用PS3提示符, 默认为(#{?}, 当然, 这可以修改.

例子10-29. 使用select来创建菜单

```

1#!/bin/bash
2
3PS3='Choose your favorite vegetable: ' # 设置提示符字符串.
4
5echo
6
7select vegetable in "beans" "carrots" "potatoes" "onions" "rutabagas"
8do
9  echo
10 echo "Your favorite veggie is $vegetable."
11 echo "Yuck!"
12 echo
13 break # 如果这里没有 'break' 会发生什么?
14done
15
16exit 0

```

如果忽略了in list列表, 那么select命令将会使用传递到脚本的命令行参数(\$@), 或者是函数参数(当select是在函数中时).

与忽略in list的

for variable [in list]

结构比较一下.

例子10-30. 使用函数中的select结构来创建菜单

```
1 #!/bin/bash
2
3 PS3='Choose your favorite vegetable: '
4
5 echo
6
7 choice_of()
8 {
9     select vegetable
10    # [in list]被忽略, 所以'select'使用传递给函数的参数.
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 carrots radishes tomatoes spinach
21 #           $1      $2      $3      $4      $5      $6
22 #           传递给choice_of()的参数
23
24 exit 0
```

第11章 内部命令与内建命令

内建命令指的就是包含在Bash工具包中的命令，从字面意思上看就是built in. 这主要是考虑到执行效率的问题— 内建命令将比外部命令执行的更快，一部分原因是由于外部命令通常都需要fork出一个单独的进程来执行— 另一部分原因是特定的内建命令需要直接访问shell的内核部分。

当一个命令或者是shell本身需要初始化(或者创建)一个新的子进程来执行一个任务的时候，这种行为被称为fork. 这个新产生的进程被叫做子进程，并且这个进程是从父进程中fork出来的。当子进程执行它的任务时，父进程也在运行。

注意：当父进程获得了子进程的进程ID时，父进程可以给子进程传递参数，然而反过来却不行。这将会产生不可思议的并且很难追踪的问题。

例子11-1. 一个fork出多个自身实例的脚本

```
1 #!/bin/bash
2 # spawn.sh
3
4
5 PIDS=$(pidof sh $0)  # 这个脚本不同实例的进程ID。
6 P_array=( $PIDS )    # 把它们放到数组里(为什么?).
7 echo $PIDS           # 显示父进程和子进程的进程ID。
8 let "instances = ${#P_array[*]} - 1"  # 计算元素个数，至少为1。
9                                # 为什么减1?
10 echo "$instances instance(s) of this script running."
11 echo "[Hit Ctl-C to exit.]"; echo
12
13
14 sleep 1              # 等一下。
15 sh $0                 # 再来一次，Sam。
16
17 exit 0               # 没必要；脚本永远不会运行到这里。
18                                # 为什么运行不到这里？
19
20 # 在使用Ctl-C退出之后，
21 #+ 是否所有产生的进程都会被kill掉？
22 # 如果是这样的话，为什么？
23
24 # 注意：
25 # -----
26 # 小心，不要让这个脚本运行太长时间。
27 # 它最后会吃掉你绝大多数的系统资源。
28
29 # 是否有合适的脚本技术，
```

```
30 #+ 用于产生脚本自身的大量实例.  
31 # 为什么或为什么不?
```

通常情况下,脚本中的Bash内建命令在运行的时候是不会fork出一个子进程的.但是脚本中的外部或者过滤命令通常会fork出一个子进程.

一个内建命令通常会与一个系统命令同名,但是Bash在内部重新实现了这些命令.比如,Bash的echo命令与/bin/echo就不尽相同,虽然它们的行为在绝大多数情况下都是一样的.

```
1#!/bin/bash  
2  
3echo "This line uses the \"echo\" builtin."  
4/bin/echo "This line uses the /bin/echo system command."
```

关键字的意思就是保留字,对于shell来说关键字具有特殊的含义,并且用来构建shell语法结构.比如,"for","while","do",和"!"都是关键字.与内建命令相似的是,关键字也是Bash的骨干部分,但是与内建命令不同的是,关键字本身并不是一个命令,而是一个比较大的命令结构的一部分.[\[1\]](#)

I/O

echo .

打印(到stdout)一个表达式或者变量([参考例子4-1](#)).

```
1 echo Hello  
2 echo $a
```

echo命令需要-e参数来打印转义字符. [参考例子5-2](#).

通常情况下,每个echo命令都会在终端上新起一行,但是-n参数会阻止新起一行.

► echo命令可以作为输入,通过管道传递到一系列命令中去.

```
1 if echo "$VAR" | grep -q txt  # if [[ $VAR = *txt* ]]  
2 then  
3   echo "$VAR contains the substring sequence \"txt\""  
4 fi
```

echo命令可以与[命令替换](#)组合起来,这样可以用来设置一个变量.

a='echo "HELLO" — tr A-Z a-z'

[参考例子12-19](#),[例子12-3](#),[例子12-42](#),和[例子12-43](#).

小心echo 'command'将会删除任何由command所产生的换行符.

\$IFS (内部域分隔符)一搬都会将\n(换行符)包含在它的空白字符集合中.Bash因此会根据参数中的换行来分离command的输出,然后echo.最后echo将以空格代替换行来输出这些参数.

```
1 bash$ ls -l /usr/share/apps/kjezz/sounds  
2 -rw-r--r-- 1 root      root          1407 Nov  7 2000 reflect.au  
3 -rw-r--r-- 1 root      root          362 Nov  7 2000 seconds.au  
4  
5 bash$ echo 'ls -l /usr/share/apps/kjezz/sounds'  
6 total 40 -rw-r--r-- 1 root root 716 Nov  7 2000 \  
7 reflect.au -rw-r--r-- 1 root root 362 Nov  7 2000 seconds.au
```

所以,我们怎么做才能够在一个需要echo出来的字符串中嵌入换行呢?

```
1 # 嵌入一个换行?
```

```
2 echo "Why doesn't this string \n split on two lines?"  
3 # 上边这句的\n将被打印出来. 达不到换行的目的.  
4  
5 # 让我们再试试其他方法.  
6  
7 echo  
8  
9 echo $"A line of text containing  
10 a linefeed."  
11 # 打印出两个独立的行(嵌入换行成功了).  
12 # 但是, 是否必须有"$"作为变量前缀?  
13  
14 echo  
15  
16 echo "This string splits  
17 on two lines."  
18 # 不, 并不是非有"$"不可.  
19  
20 echo  
21 echo "-----"  
22 echo  
23  
24 echo -n $"Another line of text containing  
25 a linefeed."  
26 # 打印出两个独立的行(嵌入换行成功了).  
27 # 即使使用了-n选项, 也没能阻止换行. (译者注: -n 阻止了第2个换行)  
28  
29 echo  
30 echo  
31 echo "-----"  
32 echo  
33 echo  
34  
35 # 然而, 下边的代码就没能像期望的那样运行.  
36 # 为什么失败? 提示: 因为分配到了变量.  
37 string1=$"Yet another line of text containing  
38 a linefeed (maybe)."  
39  
40 echo $string1  
41 # Yet another line of text containing a linefeed (maybe).  
42 #  
43 # 换行变成了空格.  
44  
45 # 感谢, Steve Parker, 指出了这点.
```

这个命令是shell的一个内建命令, 与/bin/echo不同, 虽然行为相似.

```
1 bash$ type -a echo
2 echo is a shell builtin
3 echo is /bin/echo
```

printf .

printf命令, 格式化输出, 是echo命令的增强版. 它是C语言printf()库函数的一个有限的变形, 并且在语法上有些不同.

printf format-string... parameter...

这是Bash的内建版本, 与/bin/printf或者/usr/bin/printf命令不同. 如果想更深入的了解, 请察看printf(系统命令)的man页.

④ 老版本的Bash可能不支持printf.

例子11-2. 使用printf的例子

```
1 #!/bin/bash
2 # printf 示例
3
4 PI=3.14159265358979
5 DecimalConstant=31373
6 Message1="Greetings,"
7 Message2="Earthling."
8
9 echo
10
11 printf "Pi to 2 decimal places = %1.2f" $PI
12 echo
13 printf "Pi to 9 decimal places = %1.9f" $PI # 都能够正确的结束.
14
15 printf "\n"                                # 打印一个换行,
16                                         # 等价于 'echo' . . .
17
18 printf "Constant = \t%d\n" $DecimalConstant # 插入一个 tab (\t).
19
20 printf "%s %s \n" $Message1 $Message2
21
22 echo
23
24 # =====#
25 # 模拟C函数, sprintf().
26 # 使用一个格式化的字符串来加载一个变量.
27
28 echo
29
30 Pi12=$(printf "%1.12f" $PI)
31 echo "Pi to 12 decimal places = $Pi12"
```

```

32
33 Msg='printf "%s %s \n" $Message1 $Message2'
34 echo $Msg; echo $Msg
35
36 # 像我们所看到的一样，现在'sprintf'可以
37 #+ 作为一个可被加载的模块，
38 #+ 但是不具可移植性。
39
40 exit 0

```

使用printf的最主要的应用就是格式化错误消息.

```

1 E_BADDIR=65
2
3 var=nonexistent_directory
4
5 error()
6 {
7     printf "$@" >&2
8     # 格式化传递进来的位置参数，并把它们送到stderr.
9     echo
10    exit $E_BADDIR
11 }
12
13 cd $var || error $"Can't cd to %s." "$var"
14
15 # 感谢，S.C.

```

read .

从stdin中“读取”一个变量的值，也就是，和键盘进行交互，来取得变量的值。使用-a参数可以read数组变量(参考[例子26-6](#)).

例子11-3. 使用read来进行变量分配

```

1#!/bin/bash
2# "Reading" 变量.
3
4echo -n "Enter the value of variable 'var1': "
5# -n 选项，阻止换行。
6
7read var1
8# 注意：在var1前面没有'$'，因为变量正在被设置。
9
10echo "var1 = $var1"
11
12
13echo

```

```

15 # 一个单独的'read'语句可以设置多个变量.
16 echo -n "Enter the values of variables 'var2' and 'var3'\未换行
17 (separated by a space or tab): "
18 read var2 var3
19 echo "var2 = $var2      var3 = $var3"
20 # 如果你只输入了一个值, 那么其他的变量还是处于未设置状态(null).
21
22 exit 0

```

一个不带变量参数的read命令, 将会把来自键盘的输入存入到专用变量\$REPLY中.

例子11-4. 当使用一个不带变量参数的read命令时, 将会发生什么?

```

1#!/bin/bash
2# read-novar.sh
3
4echo
5
6# -----
7echo -n "Enter a value: "
8read var
9echo "\"var\" = \"$var\""
10# 到这里为止, 都与期望的一样.
11# -----
12
13echo
14
15# -----
16echo -n "Enter another value: "
17read      # 没有变量分配给'read'命令, 所以...
18          #+# 输入将分配给默认变量, $REPLY.
19var="$REPLY"
20echo "\"var\" = \"$var\""
21# 这部分代码和上边的代码等价.
22# -----
23
24echo
25
26exit 0

```

一般的, 当输入给read时, 输入一个 然后回车, 将会阻止产生一个新行. -r选项将会让 转义.

例子11-5. read命令的多行输入

```

1#!/bin/bash
2
3echo
4
5echo "Enter a string terminated by a \\, then press <ENTER>."
6echo "Then, enter a second string, and again press <ENTER>."

```

```

7 read var1      # 当 read $var1 时, "\" 将会阻止产生新行.
8          #      first line \
9          #      second line
10
11 echo "var1 = $var1"
12 #      var1 = first line second line
13
14 # 对于每个以 "\"" 结尾的行,
15 #+ 你都会看到一个下一行的提示符, 让你继续向var1输入内容.
16
17 echo; echo
18
19 echo "Enter another string terminated by a \"\", then press <ENTER>."
20 read -r var2 # -r 选项会让 "\"" 转义.
21          #      first line \
22
23 echo "var2 = $var2"
24 #      var2 = first line \
25
26 # 第一个 <ENTER> 就会结束var2变量的录入.
27
28 echo
29
30 exit 0

```

read命令有些有趣的选项, 这些选项允许打印出一个提示符, 然后在不输入ENTER的情况下, 可以读入你所按下的字符的内容.

```

1 # 不敲回车, 读取一个按键字符.
2
3 read -s -n1 -p "Hit a key " keypress
4 echo; echo "$keypress was \"$keypress\"."
5
6 # -s 选项意味着不打印输入.
7 # -n N 选项意味着只接受N个字符的输入.
8 # -p 选项意味着在读取输入之前打印出后边的提示符.
9
10 # 使用这些选项是有技巧的, 因为你需要用正确的顺序来使用它们.

```

read命令的-n选项也可以检测方向键, 和一些控制按键.

例子11-6. 检测方向键

```

1#!/bin/bash
2# arrow-detect.sh: 检测方向键, 和一些非打印字符的按键.
3# 感谢, Sandro Magi, 告诉了我们怎么做到这点.
4
5# -----
6# 按键所产生的字符编码.

```

```
7 arrowup='\[A'
8 arrowdown='\[B'
9 arrowrt='\[C'
10 arrowleft='\[D'
11 insert='\[2'
12 delete='\[3'
13 # -----
14
15 SUCCESS=0
16 OTHER=65
17
18 echo -n "Press a key... "
19 # 如果不是上边列表所列出的按键,
20 # 可能还是需要按回车. (译者注: 因为一般按键是一个字符)
21 read -n3 key           # 读取3个字符.
22
23 echo -n "$key" | grep "$arrowup" # 检查输入字符是否匹配.
24 if [ "$?" -eq $SUCCESS ]
25 then
26   echo "Up-arrow key pressed."
27   exit $SUCCESS
28 fi
29
30 echo -n "$key" | grep "$arrowdown"
31 if [ "$?" -eq $SUCCESS ]
32 then
33   echo "Down-arrow key pressed."
34   exit $SUCCESS
35 fi
36
37 echo -n "$key" | grep "$arrowrt"
38 if [ "$?" -eq $SUCCESS ]
39 then
40   echo "Right-arrow key pressed."
41   exit $SUCCESS
42 fi
43
44 echo -n "$key" | grep "$arrowleft"
45 if [ "$?" -eq $SUCCESS ]
46 then
47   echo "Left-arrow key pressed."
48   exit $SUCCESS
49 fi
50
51 echo -n "$key" | grep "$insert"
```

```

52 if [ "$?" -eq $SUCCESS ]
53 then
54   echo "\\"Insert\" key pressed."
55   exit $SUCCESS
56 fi
57
58 echo -n "$key" | grep "$delete"
59 if [ "$?" -eq $SUCCESS ]
60 then
61   echo "\\"Delete\" key pressed."
62   exit $SUCCESS
63 fi
64
65
66 echo " Some other key pressed."
67
68 exit $OTHER
69
70 # 练习:
71 # -----
72 # 1) 使用'case'结构来代替'if'结构,
73 #+ 这样可以简化这个脚本.
74 # 2) 添加 "Home", "End", "PgUp", 和 "PgDn" 这些按键的检查.

```

► 对于read命令来说, -n选项不会检测ENTER(新行)键.

read命令的-t选项允许时间输入(参考[例子9-4](#)).

read命令也可以从重定向的文件中“读取”变量的值. 如果文件中的内容超过一行, 那么只有第一行被分配到这个变量中. 如果read命令的参数个数超过一个, 那么每个变量都会从文件中取得一个分配的字符串作为变量的值, 这些字符串都是以定义的[空白字符](#)来进行分隔的. 小心使用!

[例子11-7.](#) 通过文件重定向来使用read命令

```

1#!/bin/bash
2
3read var1 <data-file
4echo "var1 = $var1"
5# var1将会把"data-file"的第一行的全部内容都为它的值.
6
7read var2 var3 <data-file
8echo "var2 = $var2    var3 = $var3"
9# 注意, 这里的"read"命令将会产生一种不直观的行为.
10# 1) 重新从文件的开头开始读入变量.
11# 2) 每个变量都设置成了以空白分割的字符串.
12# 而不是之前的以整行的内容作为变量的值.
13# 3) 而最后一个变量将会取得第一行剩余的全部部分(译者注: 不管是否以空白分割).
14# 4) 如果需要赋值的变量个数比文件中第一行以空白分割的字符串个数还多的话,
15# 那么这些变量将会被赋空值.

```

```

16
17 echo "-----"
18
19 # 如何用循环来解决上边所提到的问题:
20 while read line
21 do
22     echo "$line"
23 done <data-file
24 # 感谢, Heiner Steven 指出了这点.
25
26 echo "-----"
27
28 # 使用$IFS(内部域分隔变量)来将每行的输入单独的放到"read"中,
29 # 前提是如果你不想使用默认空白的话.
30
31 echo "List of all users:"
32 OIFS=$IFS; IFS=:      # /etc/passwd 使用 ":" 作为域分隔符.
33 while read name passwd uid gid fullname ignore
34 do
35     echo "$name ($fullname)"
36 done </etc/passwd    # I/O 重定向.
37 IFS=$OIFS            # 恢复原始的$IFS.
38 # 这段代码也是Heiner Steven编写的.
39
40
41
42 # 在循环内部设置$IFS变量,
43 #+ 而不用把原始的$IFS
44 #+ 保存到临时变量中.
45 # 感谢, Dim Segebart, 指出了这点.
46 echo "-----"
47 echo "List of all users:"
48
49 while IFS=: read name passwd uid gid fullname ignore
50 do
51     echo "$name ($fullname)"
52 done </etc/passwd    # I/O 重定向.
53
54 echo
55 echo "\$IFS still $IFS"
56
57 exit 0

```

► 管道输出到read命令中, 使用管道echo输出来设置变量将会失败. 然而, 使用管道cat输出

看起来能够正常运行.

```
1 cat file1 file2 |  
2 while read line  
3 do  
4 echo $line  
5 done
```

但是, 就像Bj Eriksson所指出的:

例子11-8. 管道输出到read中的问题

```
1#!/bin/sh  
2# readpipe.sh  
3# 这个例子是由Bjorn Eriksson所编写的.  
4  
5last="(null)"  
6cat $0 |  
7while read line  
8do  
9    echo "{$line}"  
10   last=$line  
11done  
12printf "\nAll done, last:$last\n"  
13  
14exit 0 # 代码结束.  
15      # 下边是脚本的(部分)输出.  
16      # 'echo'出了多余的大括号.  
17  
18#####  
19  
20./readpipe.sh  
21  
22{#!/bin/sh}  
23{last="(null)"}  
24{cat $0 |}  
25{while read line}  
26{do}  
27{echo "{$line}"}  
28{last=$line}  
29{done}  
30{printf "nAll done, last:$lastn"}  
31  
32  
33All done, last:(null)  
34  
35变量(last)被设置在子shell中, 并没有被设置在外边.
```

在许多Linux发行版上, gendiff脚本通常都在/usr/bin下, 将find的输出通过管道传到while

read结构中.

```
1 find $1 \(-name "*$2" -o -name ".$*$2" \) -print |  
2 while read f; do  
3 . . .
```

文件系统

cd .

cd, 修改目录命令, 在脚本中用的最多的时候就是当命令需要在指定目录下运行时, 需要用它来修改当前工作目录.

```
1 (cd /source/directory && tar cf - . ) | (cd /dest/directory && tar xpvf -)
```

[来自于[之前引用过](#)的一个例子, 是由Alan Cox编写的]

-P (physical)选项对于cd命令的意义是忽略符号链接.

cd - 将会把工作目录修改至\$OLDPWD, 也就是之前的工作目录.

当我们使用两个"/"来作为cd命令的参数时, 结果却出乎我们的意料. .

```
1 bash$ cd //  
2 bash$ pwd  
3 //
```

输出应该是, 并且当然应该是/. 无论在命令下还是在脚本中, 这都是个问题.

pwd .

打印出当前的工作目录. 这将给出用户(或脚本)的当前工作目录([参考例子11-9](#)). 使用这个命令的结果和从内建变量\$PWD中所读取的值是相同的.

pushd, popd, dirs .

这几个命令可以使得工作目录书签化, 就是可以按顺序向前或向后移动工作目录. 压栈的动作可以保存工作目录列表. 选项可以允许对目录栈做不同的操作.

pushd dir-name把路径dir-name压入目录栈, 同时修改当前目录到dir-name.

popd将目录栈最上边的目录弹出, 同时将当前目录修改为刚弹出来的那个目录.

dirs列出所有目录栈的内容(与\$DIRSTACK变量相比较). 一个成功的pushd或者popd将会自动调用dirs命令.

对于那些并没有对当前目录做硬编码, 并且需要对当前工作目录做灵活修改的脚本来说, 使用这些命令是再好不过了. 注意内建\$DIRSTACK数组变量, 这个变量可以在脚本中进行访问, 并且它们保存了目录栈的内容.

例子11-9. 修改当前工作目录

```
1 #!/bin/bash  
2  
3 dir1=/usr/local  
4 dir2=/var/spool  
5  
6 pushd $dir1  
7 # 将自动运行一个 'dirs' (把目录栈的内容列到stdout上).  
8 echo "Now in directory `pwd`." # 使用后置引用的 'pwd'.
```

```

9
10 # 现在对'dir1'做一些操作.
11 pushd $dir2
12 echo "Now in directory `pwd`."
13
14 # 现在对'dir2'做一些操作.
15 echo "The top entry in the DIRSTACK array is $DIRSTACK."
16 popd
17 echo "Now back in directory `pwd`."
18
19 # 现在, 对'dir1'做更多的操作.
20 popd
21 echo "Now back in original working directory `pwd`."
22
23 exit 0
24
25 # 如果你不使用 'popd' 将会发生什么 -- 然后退出这个脚本?
26 # 你最后将落在哪个目录中? 为什么?

```

变量

`let .`

`let`命令将执行变量的算术操作. 在许多情况下, 它被看作是复杂的`expr`命令的一个简化版本.

例子11-10. 使用“`let`”命令来做算术运算.

```

1#!/bin/bash
2
3echo
4
5let a=11          # 与 'a=11' 相同
6let a=a+5         # 等价于 let "a = a + 5"
7                      # (双引号和空格是这句话更具可读性.)
8echo "11 + 5 = $a" # 16
9
10let "a <= 3"      # 等价于 let "a = a << 3"
11echo "\$a" (=16) left-shifted 3 places = $a"
12                      # 128
13
14let "a /= 4"       # 等价于 let "a = a / 4"
15echo "128 / 4 = $a" # 32
16
17let "a -= 5"       # 等价于 let "a = a - 5"
18echo "32 - 5 = $a" # 27
19
20let "a *= 10"      # 等价于 let "a = a * 10"
21echo "27 * 10 = $a" # 270

```

```

22
23 let "a %= 8"          # 等价于 let "a = a % 8"
24 echo "270 modulo 8 = $a  (270 / 8 = 33, remainder $a)"
25             # 6
26
27 echo
28
29 exit 0

```

eval .

eval arg1 [arg2] ... [argN]

将表达式中的参数, 或者表达式列表, 组合起来, 然后评价它们(译者注: 通常用来执行). 任何被包含在表达式中的变量都将被扩展. 结果将会被转化到命令中. 如果你想从命令行中或者是从脚本中产生代码, 那么这个命令就非常有用.

```

1 bash$ process=xterm
2 bash$ show_process="eval ps ax | grep $process"
3 bash$ $show_process
4 1867 tty1      S      0:02 xterm
5 2779 tty1      S      0:00 xterm
6 2886 pts/1      S      0:00 grep xterm

```

例子11-11. 展示eval命令的效果

```

1#!/bin/bash
2
3y='eval ls -l'  # 与 y='ls -l' 很相似
4echo $y          #+ 但是换行符将会被删除, 因为"echo"的变量未被""引用.
5echo
6echo "$y"        # 用""将变量引用起来, 换行符就不会被空格替换了.
7
8echo; echo
9
10y='eval df'     # 与 y='df' 很相似
11echo $y          #+ 换行符又被空格替换了.
12
13# 当没有LF(换行符)出现时, 如果使用"awk"这样的工具来分析输出的结果,
14#+ 应该能更容易一些.
15
16echo
17echo =====
18echo
19
20# 现在, 来看一下怎么用"eval"命令来"扩展"一个变量 . .
21
22for i in 1 2 3 4 5; do
23    eval value=$i

```

```

24 # value=$i 具有相同的效果, 在这里并不是非要使用"eval"不可.
25 # 一个缺乏特殊含义的变量将被评价为自身 -- 也就是说,
26 #+ 这个变量除了能够被扩展成自身所表示的字符外, 不能被扩展成任何其他的含义.
27 echo $value
28 done
29
30 echo
31 echo "----"
32 echo
33
34 for i in ls df; do
35   value=eval $i
36   # value=$i 在这里就与上边这句有了本质上的区别.
37   # "eval" 将会评价命令 "ls" 和 "df" . .
38   # 术语 "ls" 和 "df" 就具有特殊含义,
39   #+ 因为它们被解释成命令,
40   #+ 而不是字符串本身.
41   echo $value
42 done
43
44
45 exit 0

```

例子11-12. 强制登出(log-off)

```

1#!/bin/bash
2# 结束ppp进程来强制登出log-off.
3
4# 本脚本应该以root用户的身份来运行.
5
6killppp="eval kill -9 `ps ax | awk '/ppp/ { print $1 }'`"
7# ----- ppp的进程ID -----
8
9$killppp          # 这个变量现在成为了一个命令.
10
11
12# 下边的命令必须以root用户的身份来运行.
13
14chmod 666 /dev/ttys3      # 恢复读写权限, 否则什么?
15# 因为在ppp上执行一个SIGKILL将会修改串口的权限,
16#+ 我们把权限恢复到之前的状态.
17
18rm /var/lock/LCK..ttys3  # 删除串口锁文件.为什么?
19
20exit 0
21

```

```

22 # 练习:
23 #
24 # 1) 编写一个脚本来验证是否root用户正在运行它.
25 # 2) 做一个检查, 在杀掉某个进程之前,
26 #+ 检查一下这个将要被杀掉的进程是否正在运行.
27 # 3) 基于'fuser'来编写达到这个目的的另一个版本的脚本
28 #+ if [ fuser -s /dev/modem ]; then . .

```

例子11-13. 另一个"rot13"版本

```

1#!/bin/bash
2# 使用'eval'的一个"rot13"的版本,(译者:rot13就是把26个字母,从中间分为2半,各13个).
3# 与脚本"rot13.sh" 比较一下.
4
5setvar_rot_13()           # "rot13" 函数
6{
7    local varname=$1 varvalue=$2
8    eval $varname='$(echo "$varvalue" | tr a-z n-za-m)'
9}
10
11
12setvar_rot_13 var "foobar"   # 将 "foobar" 传递到 rot13函数中.
13echo $var                   # sbbone
14
15setvar_rot_13 var "$var"     # 传递 "sbbone" 到rot13函数中.
16                           # 又变成了原始值.
17echo $var                   # foobar
18
19# 这个例子是Segebart Chazelas编写的.
20# 作者又修改了一下.
21
22exit 0

```

Rory Winston 捐献了下边的脚本, 关于使用eval命令.

例子11-14. 在Perl脚本中使用eval命令来强制变量替换

```

1 In the Perl script "test.pl":
2 ...
3     my $WEBROOT = <WEBROOT_PATH>;
4 ...
5
6 To force variable substitution try:
7     $export WEBROOT_PATH=/usr/local/webroot
8     $sed 's/<WEBROOT_PATH>/\$WEBROOT_PATH/' < test.pl > out
9
10 But this just gives:
11     my $WEBROOT = $WEBROOT_PATH;
12

```

```

13 However:
14     $export WEBROOT_PATH=/usr/local/webroot
15     $eval sed 's%<WEBROOT_PATH>%$WEBROOT_PATH%' < test.pl > out
16 #
17 #
18 That works fine, and gives the expected substitution:
19     my $WEBROOT = /usr/local/webroot;
20
21
22 ### Paulo Marcel Coelho Aragao校正了这个原始例子.

```

④ eval命令是有风险的, 如果你有更合适的方法来实现功能的话, 尽量避免使用它. eval \$COMMANDS将会执行命令COMMANDS的内容, 如果命令中含有rm -rf *这样的东西, 可能就不是你想要的了. 当你运行一个包含有eval命令的陌生人所编写的代码片段的时候, 这是一件很危险的事情.

set .

set命令用来修改内部脚本变量的值. 它的一个作用就是触发[选项标志位](#)来帮助决定脚本的行为. 另一个作用是以一个命令的结果(set ‘command’)来重新设置脚本的[位置参数](#). 脚本将会从命令的输出中重新分析出位置参数.

例子11-15. 使用set命令来改变脚本的位置参数

```

1#!/bin/bash
2
3# script "set-test"
4
5# 使用3个命令行参数来调用这个脚本,
6# 比如, "./set-test one two three".
7
8echo
9echo "Positional parameters before set `uname -a` :"
10echo "Command-line argument #1 = $1"
11echo "Command-line argument #2 = $2"
12echo "Command-line argument #3 = $3"
13
14
15set `uname -a` # 把`uname -a`的命令输出设置
16# 为新的位置参数.
17
18echo $_          # unknown(译者注: 这要看你的uname -a输出了,
19#这句打印出的就是输出的最后一个单词.)
20# 在脚本中设置标志.
21
22echo "Positional parameters after set `uname -a` :"
23# $1, $2, $3, 等等. 这些位置参数将被重新初始化为`uname -a`的结果
24echo "Field #1 of `uname -a` = $1"

```

```

25 echo "Field #2 of 'uname -a' = $2"
26 echo "Field #3 of 'uname -a' = $3"
27 echo ---
28 echo $_          # ---
29 echo
30
31 exit 0

```

关于位置参数更多有趣的事情.

例子11-16. 反转位置参数

```

1#!/bin/bash
2# revposparams.sh: 反转位置参数.
3# 本脚本由Dan Jacobson所编写，本书作者做了一些格式上的修正.
4
5
6set a\ b c d\ e;
7#      ^      ^    转义的空格
8#      ^ ^     未转义的空格
9OIFS=$IFS; IFS=:;
10#           ^ 保存旧的IFS，然后设置新的IFS.
11
12echo
13
14until [ $# -eq 0 ]
15do      # 步进位置参数.
16  echo "### k0 = "$k""      # 步进之前
17  k=$1:$k; # 将每个位置参数都附加在循环变量的后边.
18#
19  echo "### k = "$k""      # 步进之后
20  echo
21  shift;
22done
23
24set $k # 设置一个新的位置参数.
25echo -
26echo $# # 察看位置参数的个数.
27echo -
28echo
29
30for i  # 省略 "in list" 结构,
31    #+ 为位置参数设置变量 -- i --.
32do
33  echo $i # 显示新的位置参数.
34done
35

```

```

36 IFS=$OIFS # 恢复 IFS.
37
38 # 问题:
39 # 是否有必要设置新的IFS, 内部域分隔符,
40 #+ 能够让这个脚本正常运行? (译者注: 当然有必要.)
41 # 如果你没设置新的IFS, 会发生什么? 试一下.
42 # 并且, 在第17行, 为什么新的IFS要使用 -- 一个冒号 -- ,
43 #+ 来将位置参数附加到循环变量中?
44 # 这么做的目的是什么?

45
46 exit 0
47
48 $ ./revposparams.sh
49
50 ##### k0 =
51 ##### k = a b
52
53 ##### k0 = a b
54 ##### k = c a b
55
56 ##### k0 = c a b
57 ##### k = d e c a b
58
59 -
60 3
61 -
62
63 d e
64 c
65 a b

```

不使用任何选项或参数来调用set命令的话, 将会列出所有的[环境变量](#)和其他所有的已经初始化过的变量.

```

1 bash$ set
2 AUTHORITY=/home/bozo/posts
3 BASH=/bin/bash
4 BASH_VERSION=$'2.05.8(1)-release'
5 ...
6 XAUTHORITY=/home/bozo/.Xauthority
7 _=/etc/bashrc
8 variable22=abc
9 variable23=xzy

```

如果使用参数-来调用set命令的话, 将会明确的分配位置参数. 如果-选项后边没有跟变量名

的话, 那么结果就使得所有位置参数都被unset了.

例子11-17. 重新分配位置参数

```
1 #!/bin/bash
2
3 variable="one two three four five"
4
5 set -- $variable
6 # 将位置参数的内容设为变量"$variable"的内容.
7
8 first_param=$1
9 second_param=$2
10 shift; shift      # 将最前面的两个位置参数移除.
11 remaining_params="$*"
12
13 echo
14 echo "first parameter = $first_param"          # one
15 echo "second parameter = $second_param"        # two
16 echo "remaining parameters = $remaining_params" # three four five
17
18 echo; echo
19
20 # 再来一次.
21 set -- $variable
22 first_param=$1
23 second_param=$2
24 echo "first parameter = $first_param"          # one
25 echo "second parameter = $second_param"        # two
26
27 # =====
28
29 set --
30 # 如果没指定变量,那么将会unset所有的位置参数.
31
32 first_param=$1
33 second_param=$2
34 echo "first parameter = $first_param"          # (null value)
35 echo "second parameter = $second_param"        # (null value)
36
37 exit 0
```

参考[例子10-2](#)和[例子12-51](#).

unset .

unset命令用来删除一个shell变量, 这个命令的效果就是把这个变量设为null. 注意: 这个命

令对位置参数无效.

```
1 bash$ unset PATH  
2  
3 bash$ echo $PATH  
4  
5 bash$
```

例子11-18. "Unset"一个变量

```
1#!/bin/bash  
2# unset.sh: Unset 一个变量.  
3  
4variable=hello          # 初始化.  
5echo "variable = $variable"  
6  
7unset variable          # Unset.  
8                      # 与 variable= 效果相同.  
9echo "(unset) variable = $variable" # $variable 设为 null.  
10  
11exit 0
```

export .

export命令将会使得被export的变量在所运行脚本(或shell)的所有子进程中都可用. 不幸的是, 没有办法将变量export到父进程中, 这里所指的父进程就是调用这个脚本的脚本或shell. 关于export命令的一个重要的用法就是使用在启动文件中, 启动文件用来初始化和设置环境变量, 这样, 用户进程才能够访问环境变量.

例子11-19. 使用export命令来将一个变量传递到一个内嵌awk的脚本中

```
1#!/bin/bash  
2  
3# 这是"求列的和"脚本的另外一个版本(col-totaler.sh)  
4#+ 那个脚本可以把目标文件中的指定的列上的所有数字全部累加起来,求和.  
5# 这个版本将把一个变量通过export的形式传递到'awk'中 . . .  
6#+ 并且把awk脚本放到一个变量中.  
7  
8  
9ARGS=2  
10E_WRONGARGS=65  
11  
12if [ $# -ne "$ARGS" ] # 检查命令行参数的个数.  
13then  
14    echo "Usage: `basename $0` filename column-number"  
15    exit $E_WRONGARGS  
16fi  
17  
18filename=$1
```

```

19 column_number=$2
20
21 ##### 上边的这部分,与原始脚本完全一样 #####
22
23 export column_number
24 # 将列号export出来,这样后边的进程就可用了.
25
26
27 # -----
28 awkscript='{ total += $ENVIRON["column_number"] }'
29 END { print total }'
30 # 是的, 变量可以保存awk脚本.
31 # -----
32
33 # 现在, 运行这个awk脚本.
34 awk "$awkscript" "$filename"
35
36 # 感谢, Stephane Chazelas.
37
38 exit 0

```

► 可以在一个操作中同时进行赋值和export变量, 比如: export var1=xxx.

然而, 就像Greg Keraunen所指出的, 在某些情况下, 如果使用上边这种形式的话, 将与先设置变量, 然后export变量效果不同.

```

1 bash$ export var=(a b); echo ${var[0]}
2 (a b)
3
4 bash$ var=(a b); export var; echo ${var[0]}
5 a

```

declare, typeset .

declare和**typeset**命令被用来指定或限制变量的属性.

readonly .

与**declare -r**作用相同, 设置变量的只读属性, 或者可以认为这个变量就是一个常量. 设置了这种属性之后, 如果你还要修改它, 那么将会得到一个错误信息. 这种情况与C语言中的**const**常量类型是相同的.

getopts .

可以说这个命令是分析传递到脚本中命令行参数的最有力的工具. 这个命令与外部命令 **getopt**, 还有C语言中的库函 **getopt**的作用是相同的. 它允许传递和连接多个选项[\[2\]](#) 到脚本中, 并且能够分配多个参数到脚本中(比如: `scriptname -abc -e /usr/local`).

getopts结构使用两个隐含变量. **\$OPTIND**是参数指针(选项索引) 和**\$OPTARG**(选项参数)(可选的)可以在选项后边附加一个参数. 在声明标签中, 选项名后边的冒号用来提示这个选项名已经分配了一个参数.

getopts结构通常都组成一组放在一个while循环中，循环过程中每次处理一个选项和参数，然后增加隐含变量\$OPTIND的值，再进行下一次的处理。

1. 通过命令行传递到脚本中的参数前边必须加上一个减号(-). -是一个前缀，这样getopts命令把这个参数看作为一个选项。事实上，getopts不会处理不带-前缀的参数，如果第一个参数就没有-，那么将会结束选项的处理。
2. getopts的while循环模板与标准的while循环模板有些不同，没有标准循环中的中括号[]判断条件。
3. getopts结构将会取代外部命令 getopt.

```
1 while getopts ":abcde:fg" Option
2 # 开始的声明.
3 # a, b, c, d, e, f, 和 g 被认为是选项(标志).
4 # 'e' 选项后边的 : 提示这个选项需要带一个参数.
5 # 译者注：解释一下 'a' 前边的那个 : 的作用.
6 # 如果选项'e'不带参数进行调用的话，会产生一个错误信息.
7 # 这个开头的 : 就是用来屏蔽掉这个错误信息的,
8 # 因为我们一般都会有默认处理，所以并不需要这个错误信息.
9 do
10 case $Option in
11   a ) # 对选项'a'作些操作.
12   b ) # 对选项'b'作些操作.
13   ...
14   e) # 对选项'e'作些操作，同时处理一下$OPTARG,
15       # 这个变量里边将保存传递给选项"e"的参数.
16   ...
17   g ) # 对选项'g'作些操作.
18 esac
19 done
20 shift $(( $OPTIND - 1 ))
21 # 将参数指针向下移动.
22
23 # 所有这些远没有它看起来的那么复杂.<嘿嘿>.
24
```

例子11-20. 使用getopts命令来读取传递给脚本的选项/参数

```
1#!/bin/bash
2# 练习 getopts 和 OPTIND
3# 在Bill Gradwohl的建议下，这个脚本于 10/09/03 被修改.
4
5
6# 在这里我们将学习如何使用 'getopts' 来处理脚本的命令行参数.
7# 参数被作为"选项"(标志)来解析，并且对选项分配参数.
8
9# 试一下，使用如下方法来调用这个脚本
```

```

10 # 'scriptname -mn'
11 # 'scriptname -oq qOption' (qOption 可以是任意的哪怕有些诡异字符的字符串.)
12 # 'scriptname -qXXX -r'
13 #
14 # 'scriptname -qr'      - 意外的结果, "r" 将被看成是选项 "q" 的参数.
15 # 'scriptname -q -r'    - 意外的结果, 同上.
16 # 'scriptname -mnop -mnop' - 意外的结果
17 # (OPTIND在选项刚传递进来的地方是不可靠的).
18 # (译者注: 也就是说OPTIND只是一个参数指针, 指向下一个参数的位置.
19 # 比如: -mnop 在mno处理的位置OPTION都为1, 而到p的处理就变成2,
20 #           -m -n -o 在m的时候OPTION为2, 而n为3, o为4,
21 #           也就是说它总指向下一个位置).
22 #
23 # 如果选项需要一个参数的话("flag:"), 那么它将获取
24 #+ 命令行上紧挨在它后边的任何字符.
25
26 NO_ARGS=0
27 E_OPTERROR=65
28
29 if [ $# -eq "$NO_ARGS" ] # 不带命令行参数就调用脚本?
30 then
31     echo "Usage: 'basename $0' options (-mnopqrs)"
32     exit $E_OPTERROR # 如果没有参数传递进来, 那么就退出脚本, 并且解释此脚本的用法.
33 fi
34 # 用法: scriptname -options
35 # 注意: 必须使用破折号 (-)
36
37
38 while getopts ":mnopq:rs" Option
39 do
40     case $Option in
41         m      ) echo "Scenario #1: option -m-  [OPTIND=${OPTIND}]";;
42         n | o ) echo "Scenario #2: option -$Option-  [OPTIND=${OPTIND}]";;
43         p      ) echo "Scenario #3: option -p-  [OPTIND=${OPTIND}]";;
44         q      ) echo "Scenario #4: option -q-\`"
45         with argument \"\$OPTARG\"  [OPTIND=${OPTIND}]";;
46         # 注意, 选项'q'必须分配一个参数,
47         #+ 否则, 默认将失败.
48         r | s ) echo "Scenario #5: option -$Option-";;
49         *      ) echo "Unimplemented option chosen.";;   # 默认情况的处理
50     esac
51 done
52
53 shift ${((OPTIND - 1))}
54 # (译者注: shift命令是可以带参数的, 参数就是移动的个数)

```

```

55 # 将参数指针减1, 这样它将指向下一个参数.
56 # $1 现在引用的是命令行上的第一个非选项参数,
57 #+ 如果有一个这样的参数存在的话.
58
59 exit 0
60
61 # 就像 Bill Gradwohl 所描述的,
62 # "getopts"机制允许指定一个参数,
63 #+ 但是scriptname -mnop -mnop就是一种比较特殊的情况,
64 #+ 因为在使用OPTIND的时候, 没有可靠的方法来区分到底传递进来了什么东西."

```

脚本行为

source, . (点 命令)

当在命令行中调用的时候, 这个命令将会执行一个脚本. 当在脚本中调用的时候, source file-name 将会加载file-name文件. source一个文件(或点命令)将会在脚本中引入代码, 并将这些代码附加到脚本中(与C语言中的#include指令效果相同). 最终的结果就像是在使用"source"的行上插入了相应文件的内容. 在多个脚本需要引用相同的数据, 或者需要使用函数库的情况下, 这个命令非常有用.

例子11-21. "includ"一个数据文件

```

1 #!/bin/bash
2
3 . data-file    # 加载一个数据文件.
4 # 与 "source data-file"效果相同, 但是更具可移植性.
5
6 # 文件"data-file"必须存在于当前工作目录,
7 #+ 因为这个文件是使用'basename'来引用的.
8
9 # 现在, 引用这个文件中的一些数据.
10
11 echo "variable1 (from data-file) = $variable1"
12 echo "variable3 (from data-file) = $variable3"
13
14 let "sum = $variable2 + $variable4"
15 echo "Sum of variable2 + variable4 (from data-file) = $sum"
16 echo "message1 (from data-file) is \"\$message1\""
17 # 注意:                                将双引号转义
18
19 print_message This is the message-print function in the data-file.
20
21
22 exit 0

```

上边例子11-21所使用的数据文件data-file, 必须和上边的脚本放在同一目录下.

```

1 # 这是需要被脚本加载的数据文件.
2 # 这种文件可以包含变量, 函数, 等等.

```

```

3 # 在脚本中可以通过'source'或者'.'命令来加载.
4
5 # 让我们初始化一些变量.
6
7 variable1=22
8 variable2=474
9 variable3=5
10 variable4=97
11
12 message1="Hello, how are you?"
13 message2="Enough for now. Goodbye."
14
15 print_message ()
16 {
17 # echo出所有传递进来的消息.
18
19 if [ -z "$1" ]
20 then
21     return 1
22     # 如果没有参数的话，会出错.
23 fi
24
25 echo
26
27 until [ -z "$1" ]
28 do
29     # 循环处理传递到函数中的参数.
30     echo -n "$1"
31     # 每次 echo 一个参数，-n禁止换行.
32     echo -n " "
33     # 在参数之间插入空格.
34     shift
35     # 切换到下一个.
36 done
37
38 echo
39
40 return 0
41 }

```

如果source进来的文件本身就一个可执行脚本的话，那么它将运行起来，然后将控制权交还给调用它的脚本。一个source进来的可执行脚本可以使用[return](#) 命令来达到这个目的。

(可选的)也可以向source文件中传递参数，这些参数将被看作[位置参数](#)。

```
1 source $filename $arg1 arg2
```

你甚至可以在脚本文件中source它自身, 虽然这么做看不出有什么实际的应用价值.

例子11-22. 一个(没什么用的)source自身的脚本

```
1 #!/bin/bash
2 # self-source.sh: 一个脚本"递归"的source自身.
3 # 来自于"Stupid Script Tricks," 卷 II.
4
5 MAXPASSCNT=100      # 最大的可执行次数.
6
7 echo -n "$pass_count"
8 # 在第一次运行的时候,这句只不过echo出2个空格,
9 #+ 因为$pass_count还没被初始化.
10
11 let "pass_count += 1"
12 # 假定这个未初始化的变量$pass_count
13 #+ 可以在第一次运行的时候+1.
14 # 这句可以正常工作在Bash和pdksh下, 但是
15 #+ 它依赖于不可移植(并且可能危险)的行为.
16 # 更好的方法是在使用$pass_count之前, 先把这个变量初始化为0.
17
18 while [ "$pass_count" -le $MAXPASSCNT ]
19 do
20     . $0    # 脚本"source"自身, 而不是调用自己.
21         # ./$0 (应该能够正常递归)不能在这正常运行. 为什么?
22 done
23
24 # 这里发生的动作并不是真正的递归,
25 #+ 因为脚本成功的展开了自己,换句话说,
26 #+ 在每次循环的过程中
27 #+ 在每个'source'行(第20行)上
28 # 都产生了新的代码.
29 #
30 # 当然, 脚本会把每个新'source'进来文件的"#!"行
31 #+ 都解释成注释, 而不会把它看成是一个新的脚本.
32
33 echo
34
35 exit 0    # 最终的效果就是从1数到100.
36         # 真是让人印象深刻.
37
38 # 练习:
39 # -----
40 # 使用这个小技巧编写一些真正能够干些事情的脚本.
```

exit .

无条件的停止一个脚本的运行. exit命令可以随意的取得一个整数参数, 然后把这个参数作

为这个脚本的[退出状态码](#). 在退出一个简单脚本的时候, 使用exit 0的话, 是种好习惯, 因为这表明成功运行.

如果不带参数调用exit命令退出的话, 那么退出状态码将会是脚本中最后一个命令的退出状态码. 等价于exit \$?.

exec .

这个shell内建命令将使用一个特定的命令来取代当前进程. 一般的当shell遇到一个命令, 它会[forks off](#)一个子进程来真正的运行命令. 使用exec内建命令, shell就不会fork了, 并且命令的执行将会替换掉当前shell. 因此, 在脚本中使用时, 一旦exec所执行的命令执行完毕, 那么它就会强制退出脚本. [3]

例子11-23. exec命令的效果

```
1 #!/bin/bash
2
3 exec echo "Exiting \"$0\"."    # 脚本应该在这里退出.
4
5 # -----
6 # The following lines never execute.
7
8 echo "This echo will never echo."
9
10 exit 99                      # 脚本是不会在这里退出的.
11                                # 脚本退出后会使用'echo $?'
12                                #+ 来检查一下退出码.
13                                # 一定 *不是* 99.
```

例子11-24. 一个exec自身的脚本

```
1 #!/bin/bash
2 # self-exec.sh
3
4 echo
5
6 echo "This line appears ONCE in the script, yet it keeps echoing."
7 echo "The PID of this instance of the script is still $$."
8 # 上边这行展示了并没有fork出子shell.
9
10 echo "===== Hit Ctl-C to exit ====="
11
12 sleep 1
13
14 exec $0  # 产生了本脚本的另一个实例,
15          #+ 但是这个新产生的实例却代替了原来的实例.
16
17 echo "This line will never echo!" # 为什么不是这样?
18
19 exit 0
```

exec命令还能够用来[重新分配文件描述符](#). 比如, exec jzzz-file将会用zzz-file来代替stdin.

► **find**命令的`-exec`选项与shell内建的`exec`命令是不同的.

shopt .

这个命令允许shell在空闲时修改shell选项(见[例子24-1](#)和[例子24-2](#)). 它经常出现在启动文件中,但在一般脚本中也常出现. 需要在[版本2](#)之后的Bash中才支持.

```
1 shopt -s cdspell
2 # 使用'cd'命令时, 允许产生少量的拼写错误.
3 cd /hpme # 噢! 应该是'/home'.
4 pwd      # /home
5          # 拼写错误被纠正了.
```

caller .

将`caller`命令放到[函数](#)中, 将会在`stdout`上打印出函数的调用者信息.

```
1#!/bin/bash
2
3function1 ()
4{
5    # 在 function1 () 内部.
6    caller 0    # 显示调用者信息.
7}
8
9function1    # 脚本的第9行.
10
11# 9 main test.sh
12# ^          函数调用者所在的行号.
13# ^~~~       从脚本的"main"部分开始调用的.
14# ~~~~~     调用脚本的名字.
15
16caller 0    # 没效果, 因为这个命令不在函数中.
```

`caller`命令也可以在一个被[source](#)的脚本中返回调用者信息. 当然这个调用者就是[source](#)这个脚本的脚本. 就像函数一样, 这是一个"子例程调用".

你会发现这个命令在调试的时候特别有用.

命令

true .

这是一个返回(零)成功退出状态码的命令, 但是除此之外不做任何事.

```
1 # 死循环
2 while true    # 这里的true可以用":"来替换
3 do
4     operation-1
5     operation-2
6     ...
7     operation-n
8     # 需要一种手段从循环中跳出来, 或者是让这个脚本挂起.
9 done
```

false .

这是一个返回失败[退出状态码](#)的命令, 但是除此之外不做任何事.

```
1 # 测试 "false"
2 if false
3 then
4   echo "false evaluates \"true\""
5 else
6   echo "false evaluates \"false\""
7 fi
8 # 失败会显示 "false"
9
10
11 # while "false" 循环 (空循环)
12 while false
13 do
14   # 这里面的代码不会被执行.
15   operation-1
16   operation-2
17   ...
18   operation-n
19   # 什么事都没发生!
20 done
```

type [cmd] .

与外部命令[which](#)很相像, type cmd将会给出"cmd"的完整路径. 与which命令不同的是, type命令是Bash内建命令. -a是type命令的一个非常有用的选项, 它用来鉴别参数是关键字还是内建命令, 也可以用来定位同名的系统命令.

```
1 bash$ type '['
2 [ is a shell builtin
3 bash$ type -a '['
4 [ is a shell builtin
5 [ is /usr/bin/[
```

hash [cmds] .

在shell的hash表中,[4] 记录指定命令的路径名, 所以在shell或脚本中调用这个命令的话, 就不需要再在\$PATH中重新搜索这个命令了. 如果不带参数的调用hash命令, 它将列出所有已经被hash的命令. -r选项会重新设置hash表.

bind .

bind内建命令用来显示或修改readlin [5] 的键绑定.

help .

获得shell内建命令的一个小的使用总结. 与whatis命令比较象, 但help命令是内建命令.

```
1 bash$ help exit
2 exit: exit [n]
3     Exit the shell with a status of N.  If N is omitted, the exit status
4     is that of the last command executed.
```

注意事项

1. 其中有一个例外就是[time](#)命令, Bash的官方文档说这个命令是一个关键字.
2. 一个选项就是一个行为上比较象标志位的参数, 可以用来打开或关闭脚本的某些行为. 而和某个特定选项相关的参数就是用来控制这个选项(标志)功能是开启还是关闭.
3. 除非exec命令被用来[重新分配文件描述符](#).
4. Hash是一种处理数据的方法, 这种方法就是为表中的数据建立查找键. 而数据项本身是"不规则"的, 这样就需要通过一个简单的数学算法来产生一个数字, 这个数字被用来作为查找键. 使用hash的一个最有利的优点就是提高了速度. 而缺点就是会产生"冲撞" – 也就是说, 可能会有多个数据元素使用同一个主键. possible.
关于hash的例子请参考[例子A-21](#)和[例子A-22](#).
5. 在一个交互的shell中, readline库就是Bash用来读取输入的. (译者注: 比如默认的Emacs风格的输入, 当然也可以改为vi风格的输入)

1 作业控制命令

下边的作业控制命令需要一个”作业标识符”作为参数. 请参考本章结尾部分的[表格](#).

jobs .

在后台列出所有正在运行的作业, 给出作业号. 并不象ps命令那么有用.



作业和进程的概念太容易混淆了. 特定的[内建命令](#), 比如kill, disown, 和wait命令即可以接受作业号为参数, 也可以接受进程号为参数. 但是fg, bg和jobs命令就只能接受作业号为参数.

```
1 bash$ sleep 100 &
2 [1] 1384
3
4 bash $ jobs
5 [1]+  Running                  sleep 100 &
```

”1”是作业号(作业是被当前shell所维护的), 而”1384”是进程号(进程是被系统维护的). 为了kill掉作业/进程, 或者使用kill %1或者使用kill 1384. 这两个命令都行.

感谢, S.C.

disown .

从shell的激活作业表中删除作业.

fg, bg .

fg命令可以把一个在后台运行的作业放到前台来运行. 而bg命令将会重新启动一个挂起的作业, 并且在后台运行它. 如果使用fg或者bg命令的时候没有指定作业号, 那么默认将对当前正在运行的作业进行操作.

wait .

停止脚本的运行, 直到后台运行的所有作业都结束为止, 或者如果传递了作业号或进程号为参数的话, 那么就直到指定作业结束为止. 返回等待命令的[退出状态码](#).

你可以使用wait命令来防止在后台作业没完成(这会产生一个孤儿进程)之前退出脚本.

例子11-25. 在继续处理之前, 等待一个进程的结束

```
1#!/bin/bash
2
3ROOT_UID=0    # 只有$UID为0的用户才拥有root权限.
4E_NOTROOT=65
5E_NOPARAMS=66
6
7if [ "$UID" -ne "$ROOT_UID" ]
8then
9    echo "Must be root to run this script."
10   # "Run along kid, it's past your bedtime."
11   exit $E_NOTROOT
12fi
```

```

13
14 if [ -z "$1" ]
15 then
16   echo "Usage: `basename $0` find-string"
17   exit $E_NOPARAMS
18 fi
19
20
21 echo "Updating 'locate' database..."
22 echo "This may take a while."
23 updatedb /usr &      # 必须使用root身份来运行.
24
25 wait
26 # 将不会继续向下运行, 除非'updatedb'命令执行完成.
27 # 你希望在查找文件名之前更新database.
28
29 locate $1
30
31 # 如果没有'wait'命令的话, 而且在比较糟的情况下,
32 #+ 脚本可能在'updatedb'命令还在运行的时候退出,
33 #+ 这将会导致'updatedb'成为一个孤儿进程.
34
35 exit 0

```

可选的, wait也可以接受一个作业标识符作为参数, 比如, wait%1或者wait \$PPID. 请参考[作业标识符表](#).

在一个脚本中, 使用后台运行命令(&)可能会使这个脚本挂起, 直到敲ENTER, 挂起的脚本才会被恢复. 看起来只有在这个命令的结果需要输出到stdout的时候, 这种现象才会出现. 这是个很烦人的现象.

```

1 #!/bin/bash
2 # test.sh
3
4 ls -l &
5 echo "Done."
6 -----
7 bash$ ./test.sh
8 Done.
9 [bozo@localhost test-scripts]$ total 1
10 -rwxr-xr-x    1 bozo      bozo     34 Oct 11 15:09 test.sh
11 -

```

看起来只要在后台运行命令的后边加上一个wait命令就会解决这个问题.

```

1 #!/bin/bash
2 # test.sh
3
4 ls -l &

```

```
5 echo "Done."
6 wait
7 -----
8 bash$ ./test.sh
9 Done.
10 [bozo@localhost test-scripts]$ total 1
11 -rwxr-xr-x    1 bozo      bozo     34 Oct 11 15:09 test.sh
```

如果将后台运行命令的输出重定向到文件中或/dev/null中, 也能解决这个问题.

suspend .

这个命令的效果与Control-Z很相像, 但是它挂起的是这个shell(这个shell的父进程应该在合适的时候重新恢复它).

logout .

退出一个已经登陆上的shell, 也可以指定一个[退出状态码](#).

times .

给出执行命令所占用的时间, 使用如下的形式进行输出:

```
1 0m0.020s 0m0.020s
```

这只能给出一个很有限的值, 因为它很少在shell脚本中出现.

kill .

通过发送一个适当的结束信号, 来强制结束一个进程(请参考[例子13-6](#)).

例子11-26. 一个结束自身的脚本程序

```
1 #!/bin/bash
2 # self-destruct.sh
3
4 kill $$ # 脚本将在此处结束自己的进程.
5       # 回忆一下,"$$"就是脚本的PID.
6
7 echo "This line will not echo."
8 # 而且shell将会发送一个"Terminated"消息到stdout.
9
10 exit 0
11
12 # 在脚本结束自身进程之后,
13 #+ 它返回的退出码是什么?
14 #
15 # sh self-destruct.sh
16 # echo $?
17 # 143
18 #
19 # 143 = 128 + 15
```

► kill -l将会列出所有[信号](#). kill -9是”必杀”命令, 这个命令将会结束顽固的不想被kill掉的进程. 有时候kill -15也能干这个活. 一个”僵尸进程”, 僵尸进程就是子进程已经结束了, 但是[父进程](#)还没kill掉这个子进程, 不能被登陆的用户kill掉– 因为你不能杀掉一些已经死了的东西– 但是init进程迟早会把它清除干净.

killall .

killall命令将会通过名字来杀掉一个正在运行的进程, 而不是通过[进程ID](#). 如果某个特定的命令有多个实例正在运行, 那么执行一次killall命令就会把这些实例全部杀掉.

► 这里所指的killall命令是在/usr/bin中, 而不是/etc/rc.d/init.d中的[killall脚本](#).

command .

对于命令”COMMAND”, command COMMAND会直接禁用别名和函数的查找.

译者注, 注意一下Bash执行命令的优先级:

1	别名
2	关键字
3	函数
4	内建命令
5	脚本或可执行程序(\$PATH)

► 这是shell用来影响脚本命令处理效果的三个命令之一. 另外两个分别是[builtin](#) 和[enable](#).
(译者注: 当你想运行的命令或函数与内建命令同名时, 由于内建命令比外部命令的优先级高, 而函数比内建命令的优先级高, 所以Bash将总会执行优先级比较高的命令. 这样当你想执行优先级低的命令的时候, 就没有选择的余地了. 这三个命令就是用来为你提供这样的机会.)

builtin .

当你使用builtin BUILTIN_COMMAND的时候, 只会调用shell内建命令”BUILTIN_COMMAND”, 而暂时禁用同名的函数, 或者是同名的扩展命令.

enable .

这个命令或者禁用[内建命令](#)或者恢复内建命令. 比如, enable -n kill将禁用内建命令[kill](#), 所以当我们调用kill命令时, 使用的将是/bin/kill外部命令.

-a选项会enable所有作为参数的shell内建命令, 不管它们之前是否被enable了. (译者注: 如果不带参数的调用enable -a, 那么会恢复所有内建命令.) -f filename选项将会从适当的编译过的目标文件[1] 中, 让enable命令以共享库的形式来加载内建命令.

autoload .

这是从ksh中的autoloader命令移植过来的. 一个带有”autoload”声明的函数, 在它第一次被调用的时候才会被加载. [2] 这样做是为了节省系统资源.

注意, autoload命令并不是Bash核心安装时候的一部分. 这个命令需要使用命令enable -f来加载(参考上边的enable命令).

表格11-1. 作业标识符

记法	含义
%N	作业号[N]
%S	以字符串S开头的被(命令行)调用的作业
%?S	包含字符串S的被(命令行)调用的作业
%%+	”当前”作业(前台最后结束的作业, 或后台最后启动的作业)
%-	最后的作业
\$!	最后的后台进程

注意事项

1. 一些可加载的内建命令的C源代码通常都放在/usr/share/doc/bash-?.?/functions目录下.
注意, enable的-f选项并不是所有系统都支持的.
2. autoload命令与typeset -fu效果相同.

第12章 外部过滤器, 程序和命令

本章目录

1. 基本命令
 2. 复杂命令
 3. 时间/日期命令
 4. 文本处理命令
 5. 文件与归档命令
 6. 通讯命令
 7. 终端控制命令
 8. 数学计算命令
 9. 混杂命令
-

1 基本命令

新手必须要掌握的初级命令

ls .

”列出”文件的基本命令. 但是往往就是因为这个命令太简单, 所以我们总是低估它. 比如, 使用-R选项, 递归选项, ls将会以目录树的形式列出所有文件. 另一个很有用的选项-S, 将会按照文件尺寸列出所有文件, -t, 将会按照修改时间来列出文件, -i选项会显示文件的inode(请参考[例子12-4](#)).

例子12-1. 使用ls命令来创建一个烧录CDR的内容列表

```
1 #!/bin/bash
2 # ex40.sh (burn-cd.sh)
3 # 自动刻录CDR的脚本.
4
5
6 SPEED=2          # 如果你的硬件支持的话, 你可以选用更高的速度.
7 IMAGEFILE=cdimage.iso
8 CONTENTSFILE=contents
9 DEVICE=cdrom
10 # DEVICE="0,0"      为了是用老版本的CDR
11 DEFAULTDIR=/opt  # 这是包含需要被刻录内容的目录.
12             # 必须保证目录存在.
13             # 小练习: 测试一下目录是否存在.
14
15 # 使用 Joerg Schilling 的 "cdrecord" 包:
16 # http://www.fokus.fhg.de/usr/schilling/cdrecord.html
17
18 # 如果一般用户调用这个脚本的话, 可能需要root身份
19 #+ chmod u+s /usr/bin/cdrecord
20 # 当然, 这会产生安全漏洞, 虽然这是一个比较小的安全漏洞.
21
22 if [ -z "$1" ]
23 then
24     IMAGE_DIRECTORY=$DEFAULTDIR
25     # 如果命令行没指定的话, 那么这个就是默认目录.
26 else
27     IMAGE_DIRECTORY=$1
28 fi
29
30 # 创建一个"内容列表"文件.
31 ls -lRF $IMAGE_DIRECTORY > $IMAGE_DIRECTORY/$CONTENTSFILE
32 # "l" 选项将给出一个"长"文件列表.
33 # "R" 选项将使这个列表递归.
34 # "F" 选项将标记出文件类型 (比如: 目录是以 / 结尾, 而可执行文件以 * 结尾).
```

```
35 echo "Creating table of contents."
36
37 # 在烧录到CDR之前创建一个镜像文件。
38 mkisofs -r -o $IMAGEFILE $IMAGE_DIRECTORY
39 echo "Creating ISO9660 file system image ($IMAGEFILE)."
40
41 # 烧录CDR。
42 echo "Burning the disk."
43 echo "Please be patient, this will take a while."
44 cdrecord -v -isosize speed=$SPEED dev=$DEVICE $IMAGEFILE
45
46 exit $?
```

cat, tac .

cat, 是单词concatenate的缩写, 把文件的内容输出到stdout. 当与重定向操作符(>或>&)时, 一般都是用来将多个文件连接起来.

```
1 # Uses of 'cat'
2 cat filename          # 打印出文件内容.
3
4 cat file.1 file.2 file.3 > file.123 # 把三个文件连接到一个文件中.
```

cat命令的-n选项是为了在目标文件中的所有行前边插入行号. -b也是用来加行号的, 但是不对空行进行编号. -v选项可以使用~标记法来echo出不可打印字符. -s选项可以把多个空行压缩成一个空行.

请参考[例子12-25](#)和[例子12-21](#).

► 在一个管道中, 有一种把stdin重定向到一个文件中更有效的方法, 这种方法比使用cat文件的方法更高效.

```
1 cat filename | tr a-z A-Z
2
3 tr a-z A-Z < filename  # 效果相同, 但是处理更少,
4                      #+ 并且连管道都省掉了.
```

tac命令, 就是cat命令的反转, 这个命令将会从文件结尾部分列出文件的内容.

rev .

把每一行中的内容反转, 并且输出到stdout上. 这个命令与tac命令的效果是不同的, 因为它并不反转行序, 而是把每行的内容反转.

```
1 bash$ cat file1.txt
2 This is line 1.
3 This is line 2.
4
5
6 bash$ tac file1.txt
7 This is line 2.
8 This is line 1.
```

```
9  
10  
11 bash$ rev file1.txt  
12 .1 enil si sihT  
13 .2 enil si sihT
```

cp .

这是文件拷贝命令. cp file1 file2把文件file1拷贝到file2, 如果file2存在的话, 那么file2将被覆盖(请参考[例子12-6](#)).

④ 特别有用的选项就是-a选项, 这是归档标志(目的是为了copy一个完整的目录树), -u是更新选项, -r和-R选项是递归标志

```
1 cp -u source_dir/* dest_dir  
2 # 把源目录"同步"到目标目录上,  
3 #+ 也就是拷贝所有更新的文件和之前不存在的文件.
```

mv .

这是文件移动命令. 它等价于cp和rm命令的组合. 它可以把多个文件移动到目录中,甚至将目录重命名. 想察看mv在脚本中使用的例子, 请参考[例子9-19](#)和[例子A-2](#).

当使用非交互脚本时, 可以使用mv的-f(强制)选项来避免用户的输入.

当一个目录被移动到一个已存在的目录时, 那么它将成为目标目录的子目录.

```
1 bash$ mv source_directory target_directory  
2  
3 bash$ ls -lF target_directory  
4 total 1  
5 drwxrwxr-x    2 bozo  bozo  1024 May 28 19:20 source_directory/
```

rm .

删除(清除)一个或多个文件. -f选项将强制删除文件, 即使这个文件是只读的. 并且可以用来避免用户输入(在非交互脚本中使用).

④ rm将无法删除以破折号开头的文件.

```
1 bash$ rm -badname  
2 rm: invalid option -- b  
3 Try 'rm --help' for more information.
```

解决这个问题的一个方法就是在要删除的文件的前边加上./.

```
1 bash$ rm ./badname
```

另一种解决的方法是在文件名前边加上"-".

```
1 bash$ rm -- -badname
```

当使用递归参数-r时, 这个命令将会删除整个目录树. 如果不慎的使用rm -rf *的话, 那整个目录树就真的完了.

rmdir .

删除目录. 但是只有这个目录中没有文件的时候- 当然会包含”不可见的”点文件[1] – 这个命令才会成功.

mkdir .

生成目录, 创建一个空目录. 比如, mkdir -p project/programs/December将会创建指定的目录, 即使project目录和programs目录都不存在. -p选项将会自动产生必要的父目录, 这样也就同时创建了多个目录.

chmod .

修改一个现存文件的属性(请参考例子11-12).

```
1 chmod +x filename
2 # 使得文件"filename"对所有用户都可执行.
3
4 chmod u+s filename
5 # 设置"filename"文件的"suid"位.
6 # 这样一般用户就可以在执行"filename"的时候, 拥有和文件宿主相同的权限.
7 # (这并不适用于shell脚本.)
```

```
1 chmod 644 filename
2 # 对文件"filename"的宿主设置r/w权限,
3 # 对一般用户设置读权限
4 # (8进制模式).
```

```
1 chmod 1777 directory-name
2 # 对这个目录设置r/w和可执行权限, 并开放给所有人.
3 # 同时设置 "粘贴位".
4 # 这意味着, 只有目录宿主,
5 # 文件宿主, 当然,
6 # 还有root可以删除这个目录中的任何特定的文件.
```

chattr .

修改文件属性. 这个命令与上边的chmod命令项类似, 但是有不同的选项和不同的调用语法, 并且这个命令只能工作在ext2文件系统中.

chattr命令的一个特别有趣的选项是i. chattr +i filename将使得这个文件被标记为永远不变. 这个文件将不能被修改, 连接, 或删除, 即使是root也不行. 这个文件属性只能被root设置和删除. 类似的, a选项将会把文件标记为只能追加数据.

```
1 root# chattr +i file1.txt
2
3 root# rm file1.txt
4
5 rm: remove write-protected regular file 'file1.txt'? y
6 rm: cannot remove 'file1.txt': Operation not permitted
```

如果文件设置了s(安全)属性, 那么当这个文件被删除时, 这个文件所在磁盘的块将全部被0填充.

如果文件设置了u(不可删除)属性,那么当这个文件被删除后,这个文件的内容还可以被恢复(不可删除).

如果文件设置了c(压缩)属性,那么当这个文件在进行写操作时,它将自动被压缩,并且在读的时候,自动解压.

- ▶ 使用chattr命令设置过属性的文件将不会显示在文件列表中(ls -l).

ln .

创建文件链接,前提是这个文件是存在的.”链接”就是一个文件的引用,也就是这个文件的另一个名字. ln命令允许对同一个文件引用多个链接,并且是避免混淆的一个很好的方法(请参考[例子4-6](#)).

ln对于文件来说只不过是创建了一个引用,一个指针而已,因为创建出来的连接文件只有几个字节.

绝大多数使用ln命令时,使用的是-s选项,可以称为符号链接,或”软”链接. 使用-s标志的一个优点是它可以穿越文件系统来链接目录.

关于使用这个命令的语法还是有点小技巧的. 比如: ln -s oldfile newfile将对之前存在的oldfile产生一个新的连接,newfile.

- ④ 如果之前newfile已经存在的话,将会产生一个错误信息.

```
1 使用链接中的哪种类型?  
2  
3 就像John Macdonald解释的那样:  
4  
5 不论是那种类型的链接,都提供了一种双向引用的手段 -- 也就是说,  
6 不管你用文件的那个名字对文件内容进行修改,  
7 你修改的效果都即会影响到原始名字的文件,也会影响到链接名字的文件.  
8 当你工作在更高层次的时候,才会发生软硬链接的不同.  
9 硬链接的优点是,原始文件与链接文件之间是相互独立的 --  
10 如果你删除或者重命名旧文件,那么这种操作将不会影响硬链接的文件,  
11 硬链接的文件讲还是原来文件的内容.  
12 然而如果你使用软链接的话,当你把旧文件删除或重命名后,  
13 软链接将再也找不到原来文件的内容了.  
14 而软链接的优点是它可以跨越文件系统  
15 (因为它只不过是文件名的一个引用,而并不是真正的数据).  
16 与硬链接的另一个不同是,一个符号链接可以指向一个目录.
```

链接给出了一种可以用多个名字来调用脚本的能力(当然这也适用于任何其他可执行的类型),并且脚本的行为将依赖于脚本是如何被调用的.

例子12-2. 到底是Hello还是Good-bye

```
1#!/bin/bash  
2# hello.sh: 显示"hello"还是"goodbye"  
3#+          依赖于脚本是如何被调用的.  
4  
5# 在当前目录下($PWD)为这个脚本创建一个链接:  
6#     ln -s hello.sh goodbye  
7# 现在,通过如下两种方法来调用这个脚本:  
8# ./hello.sh
```

```
9 # ./goodbye
10
11
12 HELLO_CALL=65
13 GOODBYE_CALL=66
14
15 if [ $0 = "./goodbye" ]
16 then
17   echo "Good-bye!"
18   # 当然, 在这里你也可以添加一些其他的goodbye类型的命令.
19   exit $GOODBYE_CALL
20 fi
21
22 echo "Hello!"
23 # 当然, 在这里你也可以添加一些其他的hello类型的命令.
24 exit $HELLO_CALL
```

man, info .

这两个命令用来查看系统命令或安装工具的手册和信息. 当两者都可用时, info页一般会比man页包含更多的细节描述.

注意事项

1. 点文件就是文件名以点开头的文件, 比如 /.Xdefaults. 当使用一般的ls命令时, 这样的文件是不会被显示出来的. (当然ls -a会显示它们), 所以它们也不会被意外的rm -rf *命令所删除. 在用户的home目录中, 点文件一般被用作安装和配置文件.

2 复杂命令

更高级的用户命令

find .

-exec COMMAND \;

在每一个find匹配到的文件执行COMMAND命令。命令序列以;结束(";"是转义符 以保证shell传递到find命令中的字符不会被解释为其他的特殊字符)。

```
1 bash$ find ~/ -name '*txt'  
2 /home/bozo/.kde/share/apps/karm/karmdata.txt  
3 /home/bozo/misc/irmeyc.txt  
4 /home/bozo/test-scripts/1.txt
```

如果COMMAND中包含，那么find命令将会用所有匹配文件的路径名来替换”。

```
1 find ~/ -name 'core*' -exec rm {} \;  
2 # 从用户的 home 目录中删除所有的 core dump文件.
```

```
1 find /home/bozo/projects -mtime 1  
2 # 列出最后一天被修改的  
3 #+ 在/home/bozo/projects目录树下的所有文件.  
4 #  
5 # mtime = 目标文件最后修改的时间  
6 # ctime = 修改后的最后状态(通过'chmod'或其他方法)  
7 # atime = 最后访问时间  
8  
9 DIR=/home/bozo/junk_files  
10 find "$DIR" -type f -atime +5 -exec rm {} \;  
11 #  
12 # 大括号就是"find"命令用来替换目录的地方.  
13 #  
14 # 删除至少5天内没被访问过的  
15 #+ "/home/bozo/junk_files" 中的所有文件.  
16 #  
17 # "-type filetype", where  
18 # f = regular file  
19 # d = directory, etc.  
20 # ('find' 命令的man页包含有完整的选项列表.)
```

```
1 find /etc -exec grep \  
2   '[0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-9]*[.] [0-9][0-9]*' {} \;  
3  
4 # 在 /etc 目录中的文件找到所有包含 IP 地址(xxx.xxx.xxx.xxx) 的文件.  
5 # 可能会查找到一些多余的匹配. 我们如何去掉它们呢?  
6
```

```

7 # 或许可以使用如下方法:
8
9 find /etc -type f -exec cat '{}' \; | tr -c '.[:digit:]' '\n' \
10 | grep '^[^.][^.]*\.[^.][^.]*\.[^.][^.]*\.[^.][^.]*$'
11 #
12 # [:digit:] 是一种字符类.
13 #+ 关于字符类的介绍请参考 POSIX 1003.2 标准化文档.
14
15 # 感谢, Stephane Chazelas.

```

find命令的-exec选项不应该与shell中的内建命令exec相混淆.

例子12-3. 糟糕的文件名,

删除当前目录下文件名中包含一些糟糕字符(包括空白的文件).

```

1#!/bin/bash
2# badname.sh
3# 删除当前目录下文件名中包含一些特殊字符的文件.
4# (这些特殊字符指的是不应该出现在文件名中的字符)
5
6for filename in *
7do
8    badname='echo "$filename" | sed -n /[\\+\\{\\;\\\"\\\\=\\?~\\(\\)\\<\\>\\&\\*\\|\\$/p'
9# badname='echo "$filename" | sed -n '/+[{};\"\\=\\?~()\\<\\>&\\*\\|$]/p' 这句也行.
10# 删除文件名包含这些字符的文件: + { ; " \ = ? ~ ( ) < > & * | $#
11#
12    rm $badname 2>/dev/null
13#           ~~~~~ 错误消息将被抛弃.
14done
15
16# 现在, 处理文件名中以任何方式包含空白的文件.
17find . -name "* *" -exec rm -f {} \;
18# "find"命令匹配到的目录名将替换到"{}"的位置.
19# '\'是为了保证';'被正确的转义, 并且放到命令的结尾.
20
21exit 0
22
23#-----
24# 这行下边的命令将不会运行, 因为有 "exit" 命令.
25
26# 下边这句可以用来替换上边的脚本:
27find . -name '*[+{};"\\=\\?~()\\<\\>&\\*\\|$]*' -exec rm -f '{}' \;
28# (感谢, S.C.)

```

例子12-4. 通过文件的inode号来删除文件

```

1#!/bin/bash
2# idelete.sh: 通过文件的inode号来删除文件.
3

```

```

4 # 当文件名以一个非法字符开头的时候, 这就非常有用了,
5 #+ 比如 ? 或 -.
6
7 ARGCOUNT=1                      # 文件名参数必须被传递到脚本中.
8 E_WRONGARGS=70
9 E_FILE_NOT_EXIST=71
10 E_CHANGED_MIND=72
11
12 if [ $# -ne "$ARGCOUNT" ]
13 then
14     echo "Usage: `basename $0` filename"
15     exit $E_WRONGARGS
16 fi
17
18 if [ ! -e "$1" ]
19 then
20     echo "File \"\$1\" does not exist."
21     exit $E_FILE_NOT_EXIST
22 fi
23
24 inum='ls -i | grep "$1" | awk '{print $1}''
25 # inum = inode 文件的(索引节点)号.
26 # -----
27 # 每个文件都有一个inode号, 这个号用来记录文件物理地址信息.
28 # -----
29
30 echo; echo -n "Are you absolutely sure you want to delete \"\$1\" (y/n)? "
31 # 'rm' 命令的 '-v' 选项得询问也会出现这句话.
32 read answer
33 case "$answer" in
34 [nN]) echo "Changed your mind, huh?"
35         exit $E_CHANGED_MIND
36         ;;
37 *)    echo "Deleting file \"\$1\". .";
38 esac
39
40 find . -inum $inum -exec rm {} \;
41 #          ^
42 #      大括号就是"find"命令
43 #+      用来替换文本输出的地方.
44 echo "File \"\$1\" deleted!"
45
46 exit 0

```

请参考[例子12-27](#), [例子3-4](#), 和[例子10-9](#) 这些例子展示了如何使用find命令. 对于这个强大而

又复杂的命令来说, 查看man页可以获得更多的细节.

xargs .

这是给命令传递参数的一个过滤器, 也是组合多个命令的一个工具. 它把一个数据流分割为一些足够小的块, 以方便过滤器和命令进行处理. 由此这个命令也是[后置引用](#)的一个强有力的替换. 当在一般情况下使用过多参数的命令替换都会产生失败的现象, 这时候使用xargs命令来替换, 一般都能成功. [1] 一般的, xargs从stdin或者管道中读取数据, 但是它也能够从文件的输出中读取数据.

xargs的默认命令是echo. 这意味着通过管道传递给xargs的输入将会包含换行和空白, 不过通过xargs的处理, 换行和空白将被空格取代.

```
1 bash$ ls -l
2 total 0
3 -rw-rw-r--    1 bozo  bozo          0 Jan 29 23:58 file1
4 -rw-rw-r--    1 bozo  bozo          0 Jan 29 23:58 file2
5
6 bash$ ls -l | xargs
7 total 0 -rw-rw-r-- 1 bozo bozo 0 Jan 29 23:58 file1\续行
8 -rw-rw-r-- 1 bozo bozo 0 Jan 29 23:58 file2
9
10 bash$ find ~/mail -type f | xargs grep "Linux"
11 ./misc:User-Agent: slrn/0.9.8.1 (Linux)
12 ./sent-mail-jul-2005: hosted by the Linux Documentation Project.
13 ./sent-mail-jul-2005: (Linux Documentation Project Site, rtf version)
14 ./sent-mail-jul-2005: Subject: Criticism of Bozo's Windows/Linux article
15 ./sent-mail-jul-2005: while mentioning that the Linux ext2/ext3 filesystem
16 . . .
```

ls — xargs -p -l gzip 使用[gzsps](#) 压缩当前目录下的每个文件, 每次压缩一个, 并且在每次压缩前都提示用户.

► 一个有趣的xargs选项是-n NN, NN用来限制每次传递进来参数的个数.

ls — xargs -n 8 echo以每行8列的形式列出当前目录下的所有文件.

► 另一个有用的选项是-0, 使用find -print0grep -lZ这两种组合方式. 这允许处理包含空白或引号的参数.

```
find / -type f -print0 | xargs -0 grep -liwZ GUI | xargs -0 rm -f
grep -rliwZ GUI / | xargs -0 rm -f
```

上边两行都可用来删除任何包含"GUI"的文件. (感谢, S.C.)

例子12-5. Logfile: 使用xargs来监控系统log

```
1#!/bin/bash
2
3# 从/var/log/messages生成的尾部开始
4# 产生当前目录下的一个lof文件.
5
6# 注意: 如果这个脚本被一个一般用户调用的话,
7#/var/log/messages 必须是全部可读的.
8#      #root chmod 644 /var/log/messages
```

```

9
10 LINES=5
11
12 ( date; uname -a ) >>logfile
13 # 时间和机器名
14 echo ----- >>logfile
15 tail -$LINES /var/log/messages | xargs | fmt -s >>logfile
16 echo >>logfile
17 echo >>logfile
18
19 exit 0
20
21 # 注意：
22 # -----
23 # 像 Frank Wang 所指出,
24 #+ 在原文件中的任何不匹配的引号(包括单引号和双引号)
25 #+ 都会给xargs造成麻烦.
26 #
27 # 他建议使用下边的这行来替换上边的第15行:
28 # tail -$LINES /var/log/messages | tr -d "\'"' | xargs | fmt -s \
29 #     >>logfile
30
31
32 # 练习:
33 # -----
34 # 修改这个脚本, 使得这个脚本每个20分钟
35 #+ 就跟踪一下 /var/log/messages 的修改记录.
36 # 提示: 使用 "watch" 命令.

```

和find命令一样, 将作为待替换字符的占位符。

例子12-6. 把当前目录下的文件拷贝到另一个文件中

```

1#!/bin/bash
2# copydir.sh
3
4# 将当前目录下($PWD)的所有文件都拷贝到
5#+ 命令行所指定的另一个目录中去.
6
7E_NOARGS=65
8
9if [ -z "$1" ]    # 如果没有参数传递进来那就退出.
10then
11    echo "Usage: `basename $0` directory-to-copy-to"
12    exit $E_NOARGS
13fi
14

```

```

15 ls . | xargs -i -t cp ./{} $1
16 #           ^~ ^~      ^
17 # -t 是 "verbose" (输出命令行到stderr) 选项.
18 # -i 是"替换字符串"选项.
19 # {} 是输出文本的替换点.
20 # 这与在"find"命令中使用{}的情况很相像.
21 #
22 # 列出当前目录下的所有文件(ls .),
23 #+ 将 "ls" 的输出作为参数传递到 "xargs"(-i -t 选项) 中,
24 #+ 然后拷贝(cp)这些参数({})到一个新目录中($1).
25 #
26 # 最终的结果和下边的命令等价,
27 #+ cp * $1
28 #+ 除非有文件名中嵌入了"空白"字符.
29
30 exit 0

```

例子12-7. 通过名字kill进程

```

1#!/bin/bash
2# kill-byname.sh: 通过名字kill进程.
3# 与脚本kill-process.sh相比较.
4
5# 例如,
6#+ 试一下 "./kill-byname.sh xterm" --
7#+ 并且查看你系统上的所有xterm都将消失.
8
9# 警告:
10# -----
11# 这是一个非常危险的脚本.
12# 运行它的时候一定要小心. (尤其是以root身份运行时)
13#+ 因为运行这个脚本可能会引起数据丢失或产生其他一些不好的效果.
14
15E_BADARGS=66
16
17if test -z "$1" # 没有参数传递进来?
18then
19  echo "Usage: `basename $0` Process(es)_to_kill"
20  exit $E_BADARGS
21fi
22
23
24PROCESS_NAME="$1"
25ps ax | grep "$PROCESS_NAME" | awk '{print $1}' | xargs -i kill {} 2>/dev/null
26#
27

```

```

28 # -----
29 # 注意:
30 # -i 参数是xargs命令的"替换字符串"选项.
31 # 大括号对的地方就是替换点.
32 # 2>/dev/null 将会丢弃不需要的错误消息.
33 # -----
34
35 exit $?
36
37 # 在这个脚本中, "killall"命令具有相同的效果,
38 #+ 但是这么做就没有教育意义了.

```

例子12-8. 使用xargs分析单词出现的频率

```

1#!/bin/bash
2# wf2.sh: 分析一个文本文件中单词出现的频率.
3
4# 使用 'xargs' 将文本行分解为单词.
5# 与后边的 "wf.sh" 脚本相比较.
6
7
8# 检查命令行上输入的文件.
9ARGS=1
10E_BADARGS=65
11E_NOFILE=66
12
13if [ $# -ne "$ARGS" ]
14# 纠正传递到脚本中的参数个数?
15then
16    echo "Usage: `basename $0` filename"
17    exit $E_BADARGS
18fi
19
20if [ ! -f "$1" ]      # 检查文件是否存在.
21then
22    echo "File \"\$1\" does not exist."
23    exit $E_NOFILE
24fi
25
26
27
28#####
29cat "$1" | xargs -n1 | \
30# 列出文件, 每行一个单词.
31tr A-Z a-z | \
32# 将字符转换为小写.

```

```

33 sed -e 's/\.\//g' -e 's/\,,/ /g' -e 's/ /\n/g' | \
34 # 过滤掉句号和逗号,
35 #+ 并且将单词间的空格修改为换行,
36 sort | uniq -c | sort -nr
37 # 最后统计出现次数, 把数字显示在第一列, 然后显示单词, 并按数字排序.
38 #####
39 #####
40 #####
41 # 这个例子的作用与 "wf.sh" 的作用是一样的,
42 #+ 但是这个例子比较臃肿, 并且运行起来更慢一些(为什么?).
43 #####
44 exit 0

```

expr .

通用求值表达式: 通过给定的操作(参数必须以空格分开)连接参数, 并对参数求值. 可以使算术操作, 比较操作, 字符串操作或者是逻辑操作.

expr 3 + 5

返回8

expr 5 % 3

返回2

expr 1 / 0

返回错误消息, expr: division by zero

不允许非法的算术操作.

expr 5 * 3

返回15

在算术表达式expr中使用乘法操作时, 乘法符号必须被转义.

y='expr \$y + 1'

增加变量的值, 与let y = y + 1和y = \$(\$y + 1)的效果相同. 这是使用算术表达式的一个例子.

z='expr substr \$string \$position \$length'

在位置\$position上提取\$length长度的子串.

例子12-9. 使用expr

```

1#!/bin/bash
2
3# 展示一些使用'expr'的例子
4# =====
5
6echo
7
8# 算术 操作
9# ---- -
10
11echo "Arithmetice Operators"
12echo

```

```
13 a='expr 5 + 3'
14 echo "5 + 3 = $a"
15
16 a='expr $a + 1'
17 echo
18 echo "a + 1 = $a"
19 echo "(incrementing a variable)"
20
21 a='expr 5 % 3'
22 # 取模操作
23 echo
24 echo "5 mod 3 = $a"
25
26 echo
27 echo
28
29 # 逻辑 操作
30 # ----- -----
31
32 # true返回1, false返回0,
33 #+ 而Bash的使用惯例则相反.
34
35 echo "Logical Operators"
36 echo
37
38 x=24
39 y=25
40 b='expr $x = $y'          # 测试相等.
41 echo "b = $b"              # 0 ($x -ne $y)
42 echo
43
44 a=3
45 b='expr $a \> 10'
46 echo 'b='expr $a \> 10', therefore...'
47 echo "If a > 10, b = 0 (false)"
48 echo "b = $b"                # 0 (3 ! -gt 10)
49 echo
50
51 b='expr $a \< 10'
52 echo "If a < 10, b = 1 (true)"
53 echo "b = $b"                # 1 (3 -lt 10)
54 echo
55 # 注意转义操作.
56
57 b='expr $a \<= 3'
```

```
58 echo "If a <= 3, b = 1 (true)"
59 echo "b = $b"          # 1 ( 3 -le 3 )
60 # 也有 "\>=" 操作 (大于等于).
61
62
63 echo
64 echo
65
66
67
68 # 字符串 操作
69 # -----
70
71 echo "String Operators"
72 echo
73
74 a=1234zipper43231
75 echo "The string being operated upon is \"$a\"."
76
77 # 长度: 字符串长度
78 b='expr length $a'
79 echo "Length of \"$a\" is $b."
80
81 # 索引: 从字符串的开头查找匹配的子串,
82 #       并取得第一个匹配子串的位置.
83 b='expr index $a 23'
84 echo "Numerical position of first \"2\" in \"$a\" is \"$b\"."
85
86 # substr: 从指定位置提取指定长度的字串.
87 b='expr substr $a 2 6'
88 echo "Substring of \"$a\", starting at position 2,
89 and 6 chars long is \"$b\"."
90
91
92 # 'match' 操作的默认行为就是从字符串的开始进行搜索,
93 #+ 并匹配第一个匹配的字符串.
94 #
95 #      使用正则表达式
96 b='expr match "$a" "[0-9]*"'           # 数字的个数.
97 echo Number of digits at the beginning of \"$a\" is $b.
98 b='expr match "$a" "\([0-9]*\)"'        # 注意, 需要转义括号
99 #             == == + 这样才能触发子串的匹配.
100 echo "The digits at the beginning of \"$a\" are \"$b\"."
101
102 echo
```

```
103
104 exit 0
```

:操作可以替换match命令. 比如, b='expr \$a : [0-9]'与上边所使用的b='expr match \$a [0-9]*'完全等价.

```
1 #!/bin/bash
2
3 echo
4 echo "String operations using \"expr \$string : \" construct"
5 echo "====="
6 echo
7
8 a=1234zipper5FLIPPER43231
9
10 echo "The string being operated upon is \"`expr \"$a\" : '\.(.*\)`\"."
11 #      转义括号对的操作. == ==
12
13 # *****
14 #+          转义括号对
15 #+          用来匹配一个子串
16 # *****
17
18
19 # 如果不转义括号的话...
20 #+ 那么'expr'将把string操作转换为一个整数.
21
22 echo "Length of \"\$a\" is `expr \"$a\" : '.*'`.`" # 字符串长度
23
24 echo "Number of digits at the beginning of \"\$a\" is `expr \"$a\" : '[0-9]*'`.`"
25
26 # ----- #
27
28 echo
29
30 echo "The digits at the beginning of \"\$a\" are `expr \"$a\" : '\([0-9]*\)``.`"
31 #           ==
32 echo "The first 7 characters of \"\$a\" are `expr \"$a\" : '\(.....\)``.`"
33 #           =====           ==
34 # 再来一个, 转义括号对强制一个子串匹配.
35 #
36 echo "The last 7 characters of \"\$a\" are `expr \"$a\" : '.*\\(.....\\)``.`"
37 #           =====           字符串操作的结尾^
38 # (最后这个模式的意思是忽略前边的任何字符,直到最后7个字符,
39 #+ 最后7个点就是需要匹配的任意7个字符的字串)
40
```

```
41 echo  
42  
43 exit 0
```

上边的脚本展示了expr如何使用转义括号对- ... - 和[正则表达式](#)一起来分析和匹配子串. 下边是另外一个例子, 这次的例子是真正的”应用用例”.

```
1 # 去掉字符串开头和结尾的空白.  
2 LRFDATE='expr "$LRFDATE" : '[:space:]*\(.*)[:space:]*$'  
3  
4 # 来自于Peter Knowle的"booklistgen.sh"脚本  
5 #+ 用来将文件转换为Sony Librie格式.  
6 # (http://booklistgensh.peterknowles.com)
```

[Perl](#), [sed](#), 和[awk](#)是更强大的字符串分析工具. 在脚本中加入一段比较短的sed或者awk”子程序” ([参考Section 33.2](#)), 比使用expr更有吸引力.

参考[Section 9.2](#)可以了解到更多使用expr进行字符串操作的例子.

注意事项

- 即使在某些不必非得强制使用xargs的情况下, 使用这个命令也会明显的提高多文件批处理执行命令的速度.

3 时间/日期命令

时间/日期和计时

date .

直接调用date命令就会把日期和时间输出到stdout上。这个命令有趣的地方在于它的格式化和分析选项上。

例子12-10. 使用date命令

```
1 #!/bin/bash
2 # 练习'date'命令
3
4 echo "The number of days since the year's beginning is `date +%j`."
5 # 需要在调用格式的前边加上一个'+'号。
6 # %j用来给出今天是本年度的第几天。
7
8 echo "The number of seconds elapsed since 01/01/1970 is `date +%s`."
9 # %s将产生从"UNIX 元年"到现在为止的秒数,
10 #+ 但是这东西现在还有用么?
11
12 prefix=temp
13 suffix=$(date +%s) # 'date'命令的"+%s"选项是GNU特性.
14 filename=$prefix.$suffix
15 echo $filename
16 # 这是一种非常好的产生"唯一"临时文件的办法,
17 #+ 甚至比使用$$都强.
18
19 # 如果想了解'date'命令的更多选项, 请查阅这个命令的man页.
20
21 exit 0
```

-u选项将给出UTC时间(Universal Coordinated Time)。

```
1 bash$ date
2 Fri Mar 29 21:07:39 MST 2002
3
4 bash$ date -u
5 Sat Mar 30 04:07:42 UTC 2002
```

date命令有许多的输出选项。比如%N将以十亿分之一为单位表示当前时间。这个选项的一个有趣的用法就是用来产生一个6位的随机数。

```
1 date +%N | sed -e 's/000$//' -e 's/^0//'
2
3 # 去掉开头和结尾的0.
```

当然, 还有许多其他的选项(请察看man date)。

```
1 date +%j
```

```

2 # 显示今天是本年度的第几天(从1月1日开始计算).
3
4 date +%k%M
5 # 使用24小时的格式来显示当前小时数和分钟数.
6
7
8
9 # 'TZ'参数允许改变当前的默认时区.
10 date           # Mon Mar 28 21:42:16 MST 2005
11 TZ=EST date    # Mon Mar 28 23:42:16 EST 2005
12 # 感谢, Frank Kannemann 和 Pete Sjoberg 提供了这个技巧.
13
14
15 SixDaysAgo=$(date --date='6 days ago')
16 OneMonthAgo=$(date --date='1 month ago') # 四周前(不是一个月).
17 OneYearAgo=$(date --date='1 year ago')

```

请参考[例子3-4](#).

zdump .

时区dump: 查看特定时区的当前时间.

```

1 bash$ zdump EST
2 EST  Tue Sep 18 22:09:22 2001 EST

```

time .

输出统计出来的命令执行的时间.

time ls -l / 给出的输出大概是如下格式:

```

1 0.00user 0.01system 0:00.05elapsed 16%CPU (0avgtext+0avgdata 0maxresident)k
2 0inputs+0outputs (149major+27minor)pagefaults 0swaps

```

请参考前边章节所讲的一个类似命令[times](#).

► 在Bash的[2.0版本](#)中, time成为了shell的一个保留字, 并且在一个带有管道的命令行中, 这个命令的行为有些小的变化.

touch .

这是一个用来更新文件被访问或修改的时间的工具, 这个时间可以是当前系统的时间, 也可以是指定的时间, 这个命令也用来产生一个新文件. 命令touch zzz将产生一个zzz为名字的0字节长度文件, 当然前提是zzz文件不存在. 为了存储时间信息, 就需要一个时间戳为空的文件, 比如当你想跟踪一个工程的修改时间的时候, 这就非常有用.

touch命令等价于: >> newfile或>> newfile(对于一个普通文件).

at .

at命令是一个作业控制命令, 用来在指定时间点上执行指定的命令集合. 它有点像[cron](#)命令, 然而, at命令主要还是用来执行那种一次性执行的命令集合.

at 2pm January 15将会产生提示, 提示你输入需要在这个时间上需要执行的命令序列. 这些命令应该是可以和shll脚本兼容的, 因为实际上在一个可执行的脚本中, 用户每次只能输入一行. 输入将以**Ctrl-D**结束.

你可以使用-f选项或者使用(j)重定向操作符, 来让at命令从一个文件中读取命令集合. 这个文件其实就一个可执行的脚本, 虽然它是一个不可交互的脚本. 在文件中包含一个run-parts命令, 对于执行一套不同的脚本来说是非常聪明的做法.

```
1 bash$ at 2:30 am Friday < at-jobs.list
2 job 2 at 2000-10-27 02:30
```

batch .

batch作业控制命令与at令的行为很相像, 但是batch命令被用来在系统平均负载量降到. 8以下时执行一次性的任务. 与at命令相似的是, 它也可以使用-f选项来从文件中读取命令.

cal .

从stdout中输出一个格式比较整齐的日历. 既可以指定当前年度, 也可以指定过去或将来的某个年度.

sleep .

这个命令与一个等待循环的效果一样. 你可以指定需要暂停的秒数, 这段时间将什么都不干. 当一个后台运行的进程需要偶尔检测一个事件时, 这个功能很有用. 也可用于计时. 请参考[例子29-6](#).

```
1 sleep 3      # 暂停3秒.
```

► sleep默认是以秒为单位, 但是你也可以指定分钟, 小时, 或者天数为单位.

```
1 sleep 3 h    # 暂停3小时!
```

► 如果你想每隔一段时间来运行一个命令的话, 那么[watch](#)命令将比sleep命令好得多.

usleep .

指定需要sleep的微秒数("u"会被希腊人读成"mu", 或者是micro- 前缀). 与上边的sleep命令相同, 但这个命令以微秒为单位. 当需要精确计时, 或者需要非常频繁的监控一个正在运行进程的时候, 这个命令非常有用.

```
1 usleep 30      # 暂停30微妙.
```

这个命令是Red Hat的initscripts / rc-scripts包的一部分.

事实上usleep命令并不能提供非常精确的计时, 所以如果你需要运行一个实时的任务的话, 这个命令并不适合.

hwclock, clock .

hwclock命令可以访问或调整硬件时钟. 这个命令的一些选项需要具有root权限. 在系统启动的时候, /etc/rc.d/rc.sysinit, 会使用hwclock来从硬件时钟中读取并设置系统时间.

clock命令与hwclock命令完全相同.

4 文本处理命令

..
处理文本和文本文件的命令

sort .

文件排序, 通常用在管道中当过滤器来使用. 这个命令可以依据指定的关键字或指定的字符位置, 对文件行进行排序. 使用-m选项, 它将会合并预排序的输入文件. 想了解这个命令的全部参数请参考这个命令的info页. 请参考[例子10-9](#), [例子10-10](#), 和[例子A-8](#).

tsort

拓扑排序, 读取以空格分隔的有序对, 并且依靠输入模式进行排序.

uniq

这个过滤器将会删除一个已排序文件中的重复行. 这个命令经常出现在[sort](#)命令的管道后边.

```
1 cat list-1 list-2 list-3 | sort | uniq > final.list
2 # 将3个文件连接起来,
3 # 将它们排序,
4 # 删除其中重复的行,
5 # 最后将结果重定向到一个文件中.
```

-c用来统计每行出现的次数, 并把次数作为前缀放到输出行的前面.

```
1 bash$ cat testfile
2 This line occurs only once.
3 This line occurs twice.
4 This line occurs twice.
5 This line occurs three times.
6 This line occurs three times.
7 This line occurs three times.
8
9
10 bash$ uniq -c testfile
11      1 This line occurs only once.
12      2 This line occurs twice.
13      3 This line occurs three times.
14
15
16 bash$ sort testfile | uniq -c | sort -nr
17      3 This line occurs three times.
18      2 This line occurs twice.
19      1 This line occurs only once.
```

sort INPUTFILE — uniq -c — sort -nr 命令先对INPUTFILE文件进行排序, 然后统计每行出现的次数(sort命令的-nr选项会产生一个数字的反转排序). 这种命令模板一般都用来分析log文

件或者用来分析字典列表, 或者用在那些需要检查文本词汇结构的地方.

例子12-11. 分析单词出现的频率

```
1 #!/bin/bash
2 # wf.sh: 分析文本文件中词汇出现的频率.
3 # "wf2.sh"脚本是一个效率更高的版本.
4
5
6 # 从命令行中检查输入的文件.
7 ARGS=1
8 E_BADARGS=65
9 E_NOFILE=66
10
11 if [ $# -ne "$ARGS" ] # 检验传递到脚本中参数的个数.
12 then
13   echo "Usage: `basename $0` filename"
14   exit $E_BADARGS
15 fi
16
17 if [ ! -f "$1" ] # 检查传入的文件是否存在.
18 then
19   echo "File \"\$1\" does not exist."
20   exit $E_NOFILE
21 fi
22
23
24
25 ######
26 # main ()
27 sed -e 's/\.\//g' -e 's/\,,//g' -e 's/ /\n/g' "$1" | tr 'A-Z' 'a-z' | sort | uniq -c | sort -nr
28 # =====
29 # 检查单词出现的频率
30
31
32 # 过滤掉句号和逗号,
33 #+ 并且把单词间的空格转化为换行,
34 #+ 然后转化为小写,
35 #+ 最后统计单词出现的频率并按频率排序.
36
37 # Arun Giridhar建议将上边的代码修改为:
38 # . . . | sort | uniq -c | sort +1 [-f] | sort +0 -nr
39 # 这句添加了第2个排序主键, 所以
40 #+ 这个与上边等价的例子将按照字母顺序进行排序.
41 # 就像他所解释的:
42 # "这是一个有效的根排序, 首先对频率最少的
43 #+ 列进行排序
```

```

44 #+ (单词或者字符串，忽略大小写)
45 #+ 然后对频率最高的列进行排序."
46 #
47 # 像Frank Wang所解释的那样，上边的代码等价于：
48 #+     . . . | sort | uniq -c | sort +0 -nr
49 #+ 用下边这行也行：
50 #+     . . . | sort | uniq -c | sort -k1nr -k
51 ##########
52
53 exit 0
54
55 # 练习：
56 # -----
57 # 1) 使用'sed'命令来过滤其他的标点符号，
58 #+ 比如分号。
59 # 2) 修改这个脚本，添加能够过滤多个空格或者
60 # 空白的能力。

```

```

1 bash$ cat testfile
2 This line occurs only once.
3 This line occurs twice.
4 This line occurs twice.
5 This line occurs three times.
6 This line occurs three times.
7 This line occurs three times.
8
9
10 bash$ ./wf.sh testfile
11      6 this
12      6 occurs
13      6 line
14      3 times
15      3 three
16      2 twice
17      1 only
18      1 once

```

expand, unexpand.

expand命令将会把每个tab转化为一个空格. 这个命令经常用在管道中.

unexpand命令将会把每个空格转化为一个tab. 效果与expand命令相反.

cut

一个从文件中提取特定域的工具. 这个命令与awk中使用的print \$N命令很相似, 但是更受限. 在脚本中使用cut命令会比使用awk命令来得容易一些. 最重要的选项就是-d(字段定界符)和-f(域分隔符)选项.

使用cut来获得所有mount上的文件系统的列表:

```
1 cut -d ' ' -f1,2 /etc/mtab
```

使用cut命令列出OS和内核版本:

```
1 uname -a | cut -d" " -f1,3,11,12
```

使用cut命令从e-mail中提取消息头:

```
1 bash$ grep '^Subject:' read-messages | cut -c10-80
2 Re: Linux suitable for mission-critical apps?
3 MAKE MILLIONS WORKING AT HOME!!!
4 Spam complaint
5 Re: Spam complaint
```

使用cut命令来分析一个文件:

```
1 # 列出所有在/etc/passwd中的用户.
2
3 FILENAME=/etc/passwd
4
5 for user in $(cut -d: -f1 $FILENAME)
6 do
7   echo $user
8 done
9
10 # 感谢Oleg Philon对此的建议.
```

```
1 cut -d ' ' -f2,3 filename等价于awk -F'[ ]' '{ print $2, $3 }' filename
```

你甚至可以指定换行符作为字段定界符. 这个小伎俩实际上就是在命令行上插入一个换行(RETURN). (译者: linux使用lf作为换行符).

```
1 bash$ cut -d'
2   -f3,7,19 testfile
3 This is line 3 of testfile.
4 This is line 7 of testfile.
5 This is line 19 of testfile.
```

感谢, Jaka Kranjc指出这点.

请参考[例子12-43](#).

paste

将多个文件, 以每个文件一列的形式合并到一个文件中, 合并后文件中的每一列就是原来的一个文件. 与cut结合使用, 经常用于创建系统log文件.

join

这个命令与paste命令属于同类命令. 但是它能够完成某些特殊的目地. 这个强力工具能够以一种特殊的形式来合并两个文件, 这种特殊的形式本质上就是一个关联数据库的简单版本.

join命令只能够操作两个文件. 它可以将那些具有特定标记域(通常是一个数字标签)的行合并起来, 并且将结果输出到stdout. 被加入的文件应该事先根据标记域进行排序以便于能够正确的匹配.

```
1 File: 1.data
```

```
2
```

```
3 100 Shoes
```

```
4 200 Laces
```

```
5 300 Socks
```

```
1 File: 2.data
```

```
2
```

```
3 100 $40.00
```

```
4 200 $1.00
```

```
5 300 $2.00
```

```
1 bash$ join 1.data 2.data
```

```
2 File: 1.data 2.data
```

```
3
```

```
4 100 Shoes $40.00
```

```
5 200 Laces $1.00
```

```
6 300 Socks $2.00
```

► 在输出中标记域将只会出现一次.

head

把文件的头部内容打印到stdout上(默认为10行, 可以自己修改). 这个命令有一些比较有趣的选项.

例子12-12. 哪个文件是脚本?

```
1#!/bin/bash
2# script-detector.sh: 在一个目录中检查所有的脚本文件.
3
4TESTCHARS=2      # 测试前两个字符.
5SHABANG='#!'     # 脚本都是以"#!"开头的.
6
7for file in *    # 遍历当前目录下的所有文件.
8do
9  if [[ `head -c$TESTCHARS "$file"` = "$SHABANG" ]]
10 #      head -c2                      #!
11 # '-c' 选项将从文件头输出指定个数的字符,
12 ##+ 而不是默认的行数.
13 then
14   echo "File \"\$file\" is a script."
15 else
16   echo "File \"\$file\" is *not* a script."
17 fi
18done
```

```
19
20 exit 0
21
22 # 练习：
23 # -----
24 # 1) 修改这个脚本,
25 #+ 让它可以指定扫描的路径.
26 #+ (而不是只搜索当前目录).
27 #
28 # 2) 以这个脚本目前的状况, 它不能正确识别出
29 #+ Perl, awk, 和其他一些脚本语言的脚本文件.
30 # 修正这个问题
```

例子12-13. 产生10-进制随机数

```
1#!/bin/bash
2# rnd.sh: 输出一个10进制随机数
3
4# 由Stephane Chazelas所编写的这个脚本.
5
6head -c4 /dev/urandom | od -N4 -tu4 | sed -ne '1s/.*/p'
7
8
9# ===== #
10
11# 分析
12# -----
13
14# head:
15# -c4 选项将取得前4个字节.
16
17# od:
18# -N4 选项将限制输出为4个字节.
19# -tu4 选项将使用无符号10进制格式来输出.
20
21# sed:
22# -n 选项, 使用"s"命令与"p"标志组合的方式,
23# 将会只输出匹配的行.
24
25
26
27# 本脚本作者解释,sed命令的行为如下.
28
29# head -c4 /dev/urandom | od -N4 -tu4 | sed -ne '1s/.*/p'
30# -----> |
31
```

```
32 # 假设一直处理到"sed"命令时的输出--> |
33 # 为 0000000 1198195154\n
34
35 # sed命令开始读取字符串: 0000000 1198195154\n.
36 # 这里它发现一个换行符,
37 #+ 所以sed准备处理第一行 (0000000 1198195154).
38 # sed命令开始匹配它的<range>和<action>. 第一个匹配的并且只有这一个匹配的:
39
40 # range      action
41 # 1          s/.*/ //p
42
43 # 因为行号在range中, 所以sed开始执行action:
44 #+ 替换掉以空格结束的最长的字符串, 在这行中这个字符串是
45 # ("0000000 "), 用空字符串(//)将这个匹配到的字符串替换掉,
46 # 如果成功, 那就打印出结果
47 # ("p"在这里是"s"命令的标志, 这与单独的"p"命令是不同的).
48
49 # sed命令现在开始继续读取输入. (注意在继续之前,
50 #+ continuing, 如果没使用-n选项的话, sed命令将再次
51 #+ 将这行打印一遍).
52
53 # 现在, sed命令读取剩余的字符串, 并且找到文件的结尾.
54 # sed命令开始处理第2行(这行也被标记为'$'
55 # 因为这已经是最后一行).
56 # 所以这行没被匹配到<range>中, 这样sed命令就结束了.
57
58 # 这个sed命令的简短的解释是:
59 # "在第一行中删除第一个空格左边全部的字符,
60 #+ 然后打印出来."
61
62 # 一个更好的来达到这个目的的方法是:
63 #           sed -e 's/.*/ //;q'
64
65 # 这里, <range>和<action>分别是(也可以写成
66 #           sed -e 's/.*/ /' -e q):
67
68 # range              action
69 # nothing (matches line)  s/.*/ //
70 # nothing (matches line)  q (quit)
71
72 # 这里, sed命令只会读取第一行的输入.
73 # 将会执行2个命令, 并且会在退出之前打印出(已经替换过的)这行(因为"q" action),
74 #+ 因为没使用"-n"选项.
75
76 # ===== #
```

```
77  
78 # 也可以使用如下一个更简单的语句来代替:  
79 #           head -c4 /dev/urandom| od -An -tu4  
80  
81 exit 0
```

请参考[例子12-35](#).

tail .

将一个文件结尾部分的内容输出到stdout中(默认为10行). 通常用来跟踪一个系统logfile的修改情况, 如果使用-f选项的话, 这个命令将会继续显示添加到文件中的行.

例子12-14. 使用tail命令来监控系统log

```
1#!/bin/bash  
2  
3filename=sys.log  
4  
5cat /dev/null > $filename; echo "Creating / cleaning out file."  
6# 如果文件不存在的话就创建文件,  
7#+ 然后将这个文件清空.  
8# : > filename 和 > filename 也能完成这个工作.  
9  
10tail /var/log/messages > $filename  
11# /var/log/messages 必须具有全局的可读权限才行.  
12  
13echo "$filename contains tail end of system log."  
14  
15exit 0
```

为了列出一个文本文件中的指定行的内容, 可以将head命令的输出通过管道传递到tail -1中. 比如head -8 database.txt | tail -1将会列出database.txt文件第8行的内容.

下边是将一个文本文件中指定范围的所有行都保存到一个变量中:

```
1var=$(head -$m $filename | tail -$n)  
2  
3# filename = 文件名  
4# m = 从文件开头到块结尾的行数  
5# n = 想保存到变量中的指定行数(从块结尾开始截断)
```

请参考[例子12-5](#), [例子12-35](#)和[例子29-6](#).

grep .

使用正则表达式的一个多用途文本搜索工具. 这个命令本来是ed行编辑器中的一个命令/过滤器: g/re/p – global - regular expression - print.

grep pattern [file...]

在文件中搜索所有pattern出现的位置, pattern既可以是要搜索的字符串, 也可以是一个正则表达式.

```
1bash$ grep '[rst]ystem.$' osinfo.txt
```

```
2 The GPL governs the distribution of the Linux operating system.
```

如果没有指定文件参数, grep通常用在管道中对stdout进行过滤.

```
1 bash$ ps ax | grep clock
2 765 tty1      S      0:00 xclock
3 901 pts/1      S      0:00 grep clock
```

-i 选项在搜索时忽略大小写.

-w 选项用来匹配整个单词.

-l 选项仅列出符合匹配的文件, 而不列出匹配行.

-r (递归) 选项不仅在当前工作目录下搜索匹配, 而且搜索子目录.

-n 选项列出所有匹配行, 并显示行号.

```
1 bash$ grep -n Linux osinfo.txt
2 2:This is a file containing information about Linux.
3 6:The GPL governs the distribution of the Linux operating system.
```

-v (或者--invert-match) 选项将会显示所有不匹配的行.

```
1 grep pattern1 *.txt | grep -v pattern2
2
3 # 匹配在"*.txt"中所有包含 "pattern1" 的行,
4 # 而***不显示***匹配包含"pattern2"的行.
```

-c (--count) 选项将只会显示匹配到的行数的总数, 而不会列出具体的匹配.

```
1 grep -c txt *.sgml    # (在 "*.sgml" 文件中, 匹配"txt"的行数的总数.)
2
3
4 #   grep -cz .
5 #           ^ 点
6 # 意思是计数 (-c) 所有以空字符分割(-z) 的匹配 ".." 的项
7 # ".." 是正则表达式的一个符号, 表达匹配任意一个非空字符(至少要包含一个字符).
8 #
9 printf 'a b\nc  d\n\n\n\n\n\n\n\n000\n\n000e\n000\n000\nnf' | grep -cz .    # 3
10 printf 'a b\nc  d\n\n\n\n\n\n\n\n000\n\n000e\n000\n000\nnf' | grep -cz '$'    # 5
11 printf 'a b\nc  d\n\n\n\n\n\n\n\n000\n\n000e\n000\n000\nnf' | grep -cz '^'    # 5
12 #
13 printf 'a b\nc  d\n\n\n\n\n\n\n\n000\n\n000e\n000\n000\nnf' | grep -c '$'    # 9
14 # 默认情况下, 是使用换行符(\n)来分隔匹配项.
15
16 # 注意 -z 选项是 GNU "grep" 特定的选项.
17
18
19 # 感谢, S.C.
```

当有多个文件参数的时候, grep将会指出哪个文件中包含具体的匹配.

```
1 bash$ grep Linux osinfo.txt misc.txt
2 osinfo.txt:This is a file containing information about Linux.
```

```
3 osinfo.txt:The GPL governs the distribution of the Linux operating system.  
4 misc.txt:The Linux operating system is steadily gaining in popularity.
```

如果在grep命令只搜索一个文件的时候,那么可以简单的把/dev/null作为第二个文件参数传递给grep.

```
1 bash$ grep Linux osinfo.txt /dev/null  
2 osinfo.txt:This is a file containing information about Linux.  
3 osinfo.txt:The GPL governs the distribution of the Linux operating system.
```

如果存在一个成功的匹配,那么grep命令将会返回0作为[退出状态码](#),这样就可以将grep命令的结果放在脚本的条件测试中来使用,尤其和-q(禁止输出)选项组合时特别有用.

```
1 SUCCESS=0 # 如果grep匹配成功  
2 word=Linux  
3 filename=data.file  
4  
5 grep -q "$word" "$filename" # "-q"选项将使得什么都不输出到stdout上.  
6  
7 if [ $? -eq $SUCCESS ]  
8 # if grep -q "$word" "$filename" 这句话可以代替行 5 - 7.  
9 then  
10 echo "$word found in $filename"  
11 else  
12 echo "$word not found in $filename"  
13 fi
```

[例子29-6](#) 展示了如何使用grep命令在一个系统logfile中进行一个单词的模式匹配.

例子12-15. 在脚本中模拟”grep”的行为

```
1#!/bin/bash  
2# grp.sh: 一个非常粗糙的'grep'命令的实现.  
3  
4 E_BADARGS=65  
5  
6 if [ -z "$1" ] # 检查传递给脚本的参数.  
7 then  
8 echo "Usage: 'basename $0' pattern"  
9 exit $E_BADARGS  
10 fi  
11  
12 echo  
13  
14 for file in * # 遍历$PWD下的所有文件.  
15 do  
16 output=$(sed -n "/$1/p" $file) # 命令替换.  
17  
18 if [ ! -z "$output" ] # 如果"$output"不加双引号将会发生什么?  
19 then
```

```
20     echo -n "$file: "
21     echo $output
22 fi           # sed -ne "/$1/s|^|${file}: |p" 这句与上边这段等价.
23
24 echo
25 done
26
27 echo
28
29 exit 0
30
31 # 练习:
32 # -----
33 # 1) 在任何给定的文件中,如果有超过一个匹配的话,在输出中添加新行.
34 # 2) 添加一些特征.
```

如何使用grep命令来搜索两个(或两个以上)独立的模式?如果你想在一个或多个文件中显示既匹配”pattern1”又匹配”pattern2”的所有匹配的话,那又该如何做呢?(译者:这是取交集的情况,如果取并集该怎么办呢?)

一个方法是通过[管道](#)来将grep pattern1的结果传递到grep pattern2中.

比如,给定如下文件:

```
1 # 文件名: tstfile
2
3 This is a sample file.
4 This is an ordinary text file.
5 This file does not contain any unusual text.
6 This file is not unusual.
7 Here is some text.
```

现在,让我们在这个文件中搜索既包含”file”又包含”text”的所有行.

```
1 bash$ grep file tstfile
2 # 文件名: tstfile
3 This is a sample file.
4 This is an ordinary text file.
5 This file does not contain any unusual text.
6 This file is not unusual.
7
8 bash$ grep file tstfile | grep text
9 This is an ordinary text file.
10 This file does not contain any unusual text.
```

egrep - 扩展的grep - 这个命令与grep -E等价.这个命令用起来有些不同,由于使用正则表达式的扩展集合,将会使得搜索更具灵活性.它也允许逻辑—(或)操作.

```
1 bash $ egrep 'matches|Matches' file.txt
2 Line 1 matches.
3 Line 3 Matches.
```

4 Line 4 contains matches, but also Matches

fgrep - 快速的grep - 这个命令与grep -F等价. 这是一种按照字符串字面意思进行的搜索(即不允许使用[正则表达式](#)), 这样有时候会使搜索变得容易一些.

► 在某些Linux发行版中, egrep和fgrep都是grep命令的符号链接或者别名, 只不过在调用的时候分别使用了-E和-F选项罢了.

例子12-16. 在1913年的韦氏词典中查找定义

```
1 #!/bin/bash
2 # dict-lookup.sh
3
4 # 这个脚本在1913年的韦氏词典中查找定义.
5 # 这本公共词典可以通过不同的
6 #+ 站点来下载, 包括
7 #+ Project Gutenberg (http://www.gutenberg.org/etext/247).
8 #
9 # 在使用本脚本之前,
10 #+ 先要将这本字典由DOS格式转换为UNIX格式(只以LF作为行结束符).
11 # 将这个文件存储为纯文本形式, 并且保证是未压缩的ASCII格式.
12 # 将DEFAULT_DICTFILE变量以path/filename形式设置好.
13
14
15 E_BADARGS=65
16 MAXCONTEXTLINES=50 # 显示的最大行数.
17 DEFAULT_DICTFILE="/usr/share/dict/webster1913-dict.txt"
18 # 默认的路径和文件名.
19 # 在必要的时候可以进行修改.
20 # 注意:
21 # -----
22 # 这个特定的1913年版的韦氏词典
23 #+ 在每个入口都是以大写字母开头的
24 #+ (剩余的字符都是小写).
25 # 只有每部分的第一行是以这种形式开始的,
26 #+ 这也就是为什么搜索算法是下边的这个样子.
27
28
29
30 if [[ -z $(echo "$1" | sed -n '/^ [A-Z]/p') ]]
31 # 必须指定一个要查找的单词,
32 #+ 并且这个单词必须以大写字母开头.
33 then
34   echo "Usage: `basename $0` Word-to-define [dictionary-file]"
35   echo
36   echo "Note: Word to look up must start with capital letter,"
37   echo "with the rest of the word in lowercase."
38   echo "-----"
```

```

39 echo "Examples: Abandon, Dictionary, Marking, etc."
40 exit $E_BADARGS
41 fi
42
43
44 if [ -z "$2" ]                                # 也可以指定不同的词典
45                                        #+ 作为这个脚本的第2个参数传递进来.
46 then
47     dictfile=$DEFAULT_DICTFILE
48 else
49     dictfile="$2"
50 fi
51
52 # -----
53 Definition=$(fgrep -A $MAXCONTEXTLINES "$1 \\\" \"$dictfile")
54 #                                     以 "Word \..." 这种形式定义
55 #
56 # 当然, 即使搜索一个特别大的文本文件的时候
57 #+ "fgrep"也是足够快的.
58
59
60 # 现在, 剪掉定义块.
61
62 echo "$Definition" |
63 sed -n '1,/^[A-Z]/p' |
64 # 从输出的第一行
65 #+ 打印到下一部分的第一行.
66 sed '$d' | sed '$d'
67 # 删除输出的最后两行
68 #+ (空行和下一部分的第一行).
69 #
70
71 exit 0
72
73 # 练习:
74 # -----
75 # 1) 修改这个脚本, 让它具备能够处理任何字符形式的输入
76 # + (大写, 小写, 或大小写混合), 然后将其转换为
77 # + 能够处理的统一形式.
78 #
79 # 2) 将这个脚本转化为一个GUI应用,
80 # + 使用一些比如像"gdialog"的东西 . . .
81 #      这样的话, 脚本将不再从命令行中
82 # + 取得这些参数.
83 #

```

```
84 # 3) 修改这个脚本让它具备能够分析另外一个人  
85 # + 公共词典的能力, 比如 U.S. Census Bureau Gazetteer.
```

agrep (近似grep)扩展了grep近似匹配的能力. 搜索的字符串可能会与最终匹配结果所找到字符串有些不同. 这个工具并不是核心Linux发行版的一部分.

► 为了搜索压缩文件, 应使用zgrep, zegrep, 或zfgrep. 这些命令也可以对未压缩的文件进行搜索, 只不过会比一般的grep, egrep, 和fgrep慢上一些. 当然, 在你要搜索的文件中如果混合了压缩和未压缩的文件的话, 那么使用这些命令是非常方便的.

如果要搜索bzipped类型的文件, 使用bzgrep.

look .

look命令与grep命令很相似, 但是这个命令只能做“字典查询”, 也就是它所搜索的文件必须是已经排过序的单词列表. 默认情况下, 如果没有指定搜索哪个文件, look命令就默认搜索/usr/dict/words(译者: 感觉好像应该是/usr/share/dict/words), 当然也可以指定其他目录下的文件进行搜索.

例子12-17. 检查列表中单词的正确性

```
1#!/bin/bash  
2# lookup: 对指定数据文件中的每个单词都做一遍字典查询.  
3  
4file=words.data # 指定的要搜索的数据文件.  
5  
6echo  
7  
8while [ "$word" != end ] # 数据文件中最后一个单词.  
9do  
10    read word      # 从数据文件中读, 因为在循环的后边重定向了.  
11    look $word > /dev/null # 不想将字典文件中的行显示出来.  
12    lookup=$?      # 'look'命令的退出状态.  
13  
14    if [ "$lookup" -eq 0 ]  
15    then  
16        echo "\"$word\" is valid."  
17    else  
18        echo "\"$word\" is invalid."  
19    fi  
20  
21done <"$file"      # 将stdin重定向到$file, 所以"reads"来自于$file.  
22  
23echo  
24  
25exit 0  
26  
27# -----  
28# 下边的代码行将不会执行, 因为上边已经有"exit"命令了.  
29
```

```
30
31 # Stephane Chazelas建议使用下边更简洁的方法:
32
33 while read word && [[ $word != end ]]
34 do if look "$word" > /dev/null
35     then echo "\"$word\" is valid."
36     else echo "\"$word\" is invalid."
37 fi
38 done <"$file"
39
40 exit 0
```

sed, awk

这两个命令都是独立的脚本语言, 尤其适合分析文本文件和命令输出. 既可以单独使用, 也可以结合管道和在shell脚本中使用.

sed .

非交互式的”流编辑器”, 在批处理模式下, 允许使用多个ex命令. 你会发现它在shell脚本中非常有用.

awk .

可编程的文件提取器和文件格式化工具, 在结构化的文本文件中, 处理或提取特定域(特定列)具有非常好的表现. 它的语法与C语言很类似.

wc.

wc可以统计文件或I/O流中的”单词数量”:

```
1 bash $ wc /usr/share/doc/sed-4.1.2/README
2 13 70 447 README
3 [13 lines 70 words 447 characters]
```

wc -w 统计单词数量.

wc -l 统计行数量.

wc -c 统计字节数量.

wc -m 统计字符数量.

wc -L 给出文件中最长行的长度.

使用wc命令来统计当前工作目录下有多少个.txt文件:

```
1 $ ls *.txt | wc -l
2 # 因为列出的文件名都是以换行符区分的, 所以使用-l来统计.
3
4
5 # 另一种方法:
6 #     find . -maxdepth 1 -name '*.txt' -print0 | grep -cz .
7 #     (shopt -s nullglob; set -- *.txt; echo $#)
```

```
9 # 感谢, S.C.
```

wc命令来统计所有以d - h 开头的文件的大小.

```
1 bash$ wc [d-h]* | grep total | awk '{print $3}'  
2 71832
```

```
1 使用wc命令来查看指定文件中包含"Linux"的行一共有多少.
```

```
1 bash$ grep Linux abs-book.sgml | wc -l  
2 50
```

请参考[例子12-35](#)和[例子16-8](#).

某些命令的某些选项其实已经包含了wc命令的部分功能.

```
1 ... | grep foo | wc -l  
2 # 这个命令使用的非常频繁, 但事实上它有更简便的写法.  
3  
4 ... | grep -c foo  
5 # 只要使用grep命令的"-c"(或"--count")选项就能达到同样的目的.  
6  
7 # 感谢, S.C.
```

tr .

字符转换过滤器.

(*)

必须使用[引用或中括号](#), 这样做才是合理的. 引用可以阻止shell重新解释出现在tr命令序列中的特殊字符. 中括号应该被引用起来防止被shell扩展.

无论tr "A-Z" "*" &filename还是tr A-Z &filename都可以将filename中的大写字符修改为星号(写到stdout). 但是在某些系统上可能就不能正常工作了, 而tr A-Z '[**]'在任何系统上都可以正常工作.

-d选项删除指定范围的字符.

```
1 echo "abcdef"          # abcdef  
2 echo "abcdef" | tr -d b-d    # aef  
3  
4  
5 tr -d 0-9 <filename  
6 # 删除"filename"中所有的数字.
```

--squeeze-repeats (或-s)选项用来在重复字符序列中除去除第一个字符以外的所有字符. 这个选项在删除多余[空白](#)的时候非常有用.

```
1 bash$ echo "XXXXXX" | tr --squeeze-repeats 'X'  
2 X
```

-c"complement"选项将会反转匹配的字符集. 通过这个选项, tr将只会对那些不匹配的字符起作用.

```
1 bash$ echo "acfdeb123" | tr -c b-d +  
2 +c+d+b+++++
```

注意tr命令支持POSIX字符类[1].

```
1 bash$ echo "abcd2ef1" | tr '[[:alpha:]]' -
2 ----2--1
```

例子12-18. 转换大写: 把一个文件的内容全部转换为大写.

```
1 #!/bin/bash
2 # 把一个文件的内容全部转换为大写.
3
4 E_BADARGS=65
5
6 if [ -z "$1" ] # 检查命令行参数.
7 then
8   echo "Usage: `basename $0` filename"
9   exit $E_BADARGS
10 fi
11
12 tr a-z A-Z <"$1"
13
14 # 与上边的作用相同, 但是使用了POSIX字符集标记方法:
15 #       tr '[[:lower:]]' '[[:upper:]]' <"$1"
16 # 感谢, S.C.
17
18 exit 0
19
20 # 练习:
21 # 重写这个脚本, 通过选项可以控制脚本或者
22 #+ 转换为大写或者转换为小写.
```

例子12-19. 转换小写: 将当前目录下的所有文全部转换为小写.

```
1 #!/bin/bash
2 #
3 # 将当前目录下的所有文全部转换为小写.
4 #
5 # 灵感来自于John Dubois的脚本,
6 #+ Chet Ramey将其转换为Bash脚本,
7 #+ 然后被本书作者精简了一下.
8
9
10 for filename in *           # 遍历当前目录下的所有文件.
11 do
12   fname=`basename $filename`
13   n=`echo $fname | tr A-Z a-z` # 将名字修改为小写.
14   if [ "$fname" != "$n" ]      # 只对那些文件名不是小写的文件进行重命名.
15   then
16     mv $fname $n
```

```

17     fi
18 done
19
20 exit $?
21
22
23 # 下边的代码将不会被执行，因为上边的"exit".
24 #-----#
25 # 删除上边的内容，来运行下边的内容.
26
27 # 对于那些文件名中包含空白和新行的文件，上边的脚本就不能工作了.
28 # Stephane Chazelas因此建议使用下边的方法：
29
30
31 for filename in *      # 不必非得使用basename命令,
32                 # 因为 "*" 不会返回任何包含 "/" 的文件.
33 do n='echo "$filename/" | tr '[:upper:]' '[:lower:]'
34 #                         POSIX 字符集标记法.
35 #                         添加的斜线是为了在文件名结尾换行不会被
36 #                         命令替换删掉.
37 # 变量替换:
38 n=${n%/}                # 从文件名中将上边添加在结尾的斜线删除掉.
39 [[ $filename == $n ]] || mv "$filename" "$n"
40                         # 检查文件名是否已经是小写.
41 done
42
43 exit $?

```

例子12-20. Du: DOS到UNIX文本文件的转换.

```

1#!/bin/bash
2# Du.sh: DOS到UNIX文本文件的转换.
3
4E_WRONGARGS=65
5
6if [ -z "$1" ]
7then
8    echo "Usage: `basename $0` filename-to-convert"
9    exit $E_WRONGARGS
10fi
11
12NEWFILENAME=$1.unx
13
14CR='\015' # 回车.
15          # 015是8进制的ASCII码的回车.
16          # DOS中文本文件的行结束符是CR-LF.

```

```

17      # UNIX中文本文件的行结束符只是LF.
18
19 tr -d $CR < $1 > $NEWFILENAME
20 # 删除回车并且写到新文件中.
21
22 echo "Original DOS text file is \"\$1\"."
23 echo "Converted UNIX text file is \"\$NEWFILENAME\"."
24
25 exit 0
26
27 # 练习:
28 # -----
29 # 修改上边的脚本完成从UNIX到DOS的转换.

```

例子12-21. rot13: rot13, 弱智加密.

```

1#!/bin/bash
2# rot13.sh: 典型的rot13算法,
3#           使用这种方法加密至少可以愚弄一下3岁小孩.
4
5# 用法: ./rot13.sh filename
6# 或    ./rot13.sh <filename
7# 或    ./rot13.sh and supply keyboard input (stdin)
8
9cat "$@" | tr 'a-zA-Z' 'n-za-mN-ZA-M' # "a"变为"n", "b"变为"o", 等等.
10# 'cat "$@"'结构
11#+ 允许从stdin或者从文件中获得输入.
12
13exit 0

```

例子12-22. 产生"Crypto-Quote"游戏(译者: 一种文字游戏)

```

1#!/bin/bash
2# crypto-quote.sh: 加密
3
4# 使用单码替换(单一字母替换法)来进行加密.
5# 这个脚本的结果与"Crypto Quote"游戏
6#+ 的行为很相似.
7
8
9key=ETAOINSHRDLUBCFGJMQPVWZYXK
10# "key"不过是一个乱序的字母表.
11# 修改"key"就会修改加密的结果.
12
13# 'cat "$@"' 结构既可以从stdin获得输入, 也可以从文件中获得输入.
14# 如果使用stdin, 那么要想结束输入就使用 Control-D.
15# 否则就要在命令行上指定文件名.

```

```

16
17 cat "$@" | tr "a-z" "A-Z" | tr "A-Z" "$key"
18 #           | 转化为大写   | 加密
19 # 小写, 大写, 或混合大小写, 都可以正常工作.
20 # 但是传递进来的非字母字符将不会起任何变化.
21
22
23 # 用下边的语句试试这个脚本:
24 # "Nothing so needs reforming as other people's habits."
25 # --Mark Twain
26 #
27 # 输出为:
28 # "CFPHRCS QF CIIQ MINFMBRCS EQ FPHIM GIFGUI'Q HETRPQ."
29 # --BEML PZERC
30
31 # 解密:
32 # cat "$@" | tr "$key" "A-Z"
33
34
35 # 这个简单的密码可以轻易的被一个12岁的小孩
36 #+ 用铅笔和纸破解.
37
38 exit 0
39
40 # 练习:
41 # -----
42 # 修改这个脚本, 让它可以用命令行参数
43 #+ 来决定加密或解密.

```

tr的不同版本

tr工具在历史上有2个重要版本. BSD版本不需要使用中括号(tr a-z A-Z), 但是SysV版本则需要中括号(tr '[a-z]' '[A-Z]'). GNU版本的tr命令与BSD版本比较象, 所以最好使用中括号来引用字符范围.

fold

将输入按照指定宽度进行折行. 这里有一个非常有用的选项-s, 这个选项可以使用空格进行断行(译者: 事实上只有外文才需要使用空格断行, 中文是不需要的) (请参考[例子12-23](#)和[例子A-1](#)).

fmt .

一个简单的文件格式器, 通常用在管道中, 将一个比较长的文本行输出进行”折行”.

```

1#!/bin/bash
2
3 WIDTH=40          # 设为40列宽.
4

```

```

5 b='ls /usr/local/bin'      # 取得文件列表...
6
7 echo $b | fmt -w $WIDTH
8
9 # 也可以使用如下方法, 作用是相同的.
10 #   echo $b | fold -s -w $WIDTH
11
12 exit 0

```

请参考[例子12-5](#).

► 如果想找到一个更强力的fmt工具可以选择Kamil Toman的工具par, 这个工具可以从后边的这个网址取得<http://www.cs.berkeley.edu/amc/Par/>.

col .

这个命令用来滤除标准输入的反向换行符号. 这个工具还可以将空白用等价的tab来替换. col工具最主要的应用还是从特定的文本处理工具中过滤输出, 比如groff和tbl. (译者: 主要用来将man页转化为文本.)

column .

列格式化工具. 通过在合适的位置插入tab, 这个过滤工具会将列类型的文本转化为"易于打印"的表格式进行输出.

例子12-24. 使用column来格式化目录列表

```

1#!/bin/bash
2# 这是"column" man页中的一个例子, 作者对这个例子做了很小的修改.
3
4
5 (printf "PERMISSIONS LINKS OWNER GROUP SIZE MONTH DAY HH:MM PROG-NAME\n" \
6 ; ls -l | sed 1d) | column -t
7
8 # 管道中的"sed 1d"删除输出的第一行,
9 #+ 第一行将是"total      N",
10 #+ 其中"N"是"ls -l"找到的文件总数.
11
12 # "column"中的-t选项用来转化为易于打印的表形式.
13
14 exit 0

```

colrm .

列删除过滤器. 这个工具将会从文件中删除指定的列(列中的字符串)并且写到文件中, 如果指定的列不存在, 那么就回到stdout. colrm 2 4 filename将会删除filename文件中每行的第2到第4列之间的所有字符.

⑧ 如果这个文件包含tab和不可打印字符, 那将会引起不可预期的行为. 在这种情况下, 应该通过管道的手段使用[expand](#)和[unexpand](#)来预处理colrm.

nl .

计算行号过滤器. nl filename将会把filename文件的所有内容都输出到stdout上, 但是会在每个非空行的前面加上连续的行号. 如果没有filename参数, 那么就操作stdin.

nl命令的输出与cat -n非常相似, 然而, 默认情况下nl不会列出空行.

例子12-25. nl: 一个自己计算行号的脚本.

```
1 #!/bin/bash
2 # line-number.sh
3
4 # 这个脚本将会echo自身两次, 并显示行号.
5
6 # 'nl'命令显示的时候你将会看到, 本行是第4行, 因为它不计空行.
7 # 'cat -n'命令显示的时候你将会看到, 本行是第6行.
8
9 nl `basename $0'
10
11 echo; echo # 下边, 让我们试试 'cat -n'
12
13 cat -n `basename $0'
14 # 区别就是'cat -n'对空行也进行计数.
15 # 注意'nl -ba'也会这么做.
16
17 exit 0
18 # -----
```

pr .

格式化打印过滤器. 这个命令会将文件(或stdout)分页, 将它们分成合适的小块以便于硬拷贝打印或者在屏幕上浏览. 使用这个命令的不同的参数可以完成好多任务, 比如对行和列的操作, 加入行, 设置页边, 计算行号, 添加页眉, 合并文件等等. pr命令集合了许多命令的功能, 比如nl, paste, fold, column, 和expand.

pr -o 5 --width=65 fileZZZ | more 这个命令对fileZZZ进行了比较好的分页, 并且打印到屏幕上. 文件的缩进被设置为5, 总宽度设置为65.

一个非常有用的选项-d, 强制隔行打印(与sed -G效果相同).

gettext .

GNU gettext包是专门用来将程序的输出翻译或者[本地化](#)为不同国家语言的工具集. 在最开始的时候仅仅支持C语言, 现在已经支持了相当数量的其它程序语言和脚本语言.

想要查看gettext程序如何在shell脚本中使用. 请参考info页.

msgfmt .

一个产生二进制消息目录的程序. 这个命令主要用来本地化.

iconv .

一个可以将文件转化为不同编码格式(字符集)的工具. 这个命令主要用来本地化.

```
1 # 将字符串由UTF-8格式转换为UTF-16并且打印到BookList中
2 function write_utf8_string {
```

```

3  STRING=$1
4  BOOKLIST=$2
5  echo -n "$STRING" | iconv -f UTF8 -t UTF16 | \
6    cut -b 3- | tr -d \\n >> "$BOOKLIST"
7 }
8
9 # 来自于Peter Knowles的"booklistgen.sh"脚本
10 #+ 目的是把文件转换为Sony Librie格式.
11 # (http://booklistgensh.peterknowles.com)

```

recode .

可以认为这个命令是上边iconv命令的专业版本. 这个非常灵活的并可以把整个文件都转换为不同编码格式的工具并不是Linux标准安装的一部分.

TeX, gs .

TeX和Postscript都是文本标记语言, 用来对打印和格式化的视频显示进行预拷贝.

TeX是Donald Knuth精心制作的排版系统. 通常情况下, 通过编写脚本的手段来把所有的选项和参数封装起来一起传到标记语言中是一件很方便的事情.

Ghostscript (gs) 是一个遵循GPL的Postscript解释器.

enscript .

将纯文本文件转换为PostScript的工具

比如, enscrip filename.txt -p filename.ps 产生一个PostScript 输出文件filename.ps.

groff, tbl, eqn .

另一种文本标记和显示格式化语言是groff. 这是一个对传统UNIX roff/troff显示和排版包的GNU增强版本. Man页使用的就是groff.

tbl表处理工具可以认为是groff的一部分, 它的功能就是将表标记转化到groff命令中.

eqn等式处理工具也是groff的一部分, 它的功能是将等式标记转化到groff命令中.

例子12-26. manview: 查看格式化的man页

```

1#!/bin/bash
2# manview.sh: 将man页源文件格式化以方便查看.
3
4# 当你想阅读man页的时候, 这个脚本就有用了.
5# 它允许你在运行的时候查看
6#+ 中间结果.
7
8E_WRONGARGS=65
9
10if [ -z "$1" ]
11then
12  echo "Usage: `basename $0` filename"
13  exit $E_WRONGARGS
14fi

```

```
15
16 # -----
17 groff -Tascii -man $1 | less
18 # 来自于groff的man页.
19 # -----
20
21 # 如果man页中包括表或者等式,
22 #+ 那么上边的代码就够呛了.
23 # 下边的这行代码可以解决上边的这个问题.
24 #
25 #    gtbl < "$1" | geqn -Tlatin1 | groff -Tlatin1 -mtty-char -man
26 #
27 # 感谢, S.C.
28
29 exit 0
```

lex, yacc .

lex是用于模式匹配的词汇分析产生程序. 在Linux系统上这个命令已经被flex取代了.

yacc工具基于一系列的语法规范, 产生一个语法分析器. 在Linux系统上这个命令已经被bison取代了.

注意事项

1. 对于GNU版本的tr命令来说, 这是唯一一处比那些商业UNIX系统上的一般版本更好的地方.

5 文件与归档命令

归档命令

tar.

标准的UNIX归档工具. [1] 起初这只是一个磁带归档程序, 而现在这个工具已经被开发为通用打包程序, 它能够处理所有设备的所有类型的归档文件, 包括磁带设备, 正常文件, 甚至是stdout(请参考[例子3-4](#)). GNU的tar工具现在可以接受不同种类的压缩过滤器, 比如tar czvf archive_name.tar.gz *, 并且可以递归的处理归档文件, 还可以用gzips压缩目录下的所有文件, 除了当前目录下(\$PWD)的[点文件](#).[2]

一些有用的tar命令选项:

- c 创建(一个新的归档文件)
- x 解压文件(从存在的归档文件中)
- delete 删除文件(从存在的归档文件中)

Caution

这个选项不能用于磁带类型设备.

- r 将文件添加到现存的归档文件的尾部
- A 将tar文件添加到现存的归档文件的尾部
- t 列出现存的归档文件中包含的内容
- u 更新归档文件
- d 使用指定的文件系统, 比较归档文件
- z 用gzip压缩归档文件

(压缩还是解压, 依赖于是否组合了-c或-x)选项

- j 用bzip2压缩归档文件

⑥ 如果想从损坏的用gzipped压缩过的tar文件中取得数据, 那将是非常困难的. 所以当我们归档重要的文件的时候, 一定要保留多个备份.

shar.

Shell归档工具. 存在于shell归档文件中的所有文件都是未经压缩的, 并且本质上是一个shell脚本, 以#!/bin/sh开头, 并且包含所有必要的解档命令. Shar 归档文件至今还在Internet新闻组中使用, 否则的话, shar早就被tar/gzip所取代了. unshar命令用来解档shar归档文件.

ar.

创建和操作归档文件的工具, 主要在对二进制目标文件打包成库时才会用到.

rpm.

Red Hat包管理器, 或者说rpm工具提供了一种对源文件或二进制文件进行打包的方法. 除此之外, 它还包括安装命令, 并且还检查包的完整性.

一个简单的rpm -i package_name.rpm命令对于安装一个包来说就足够了, 虽然这个命令还有好多其它的选项.

- ▶ rpm -qf 列出一个文件属于那个包.

1 bash\$ rpm -qf /bin/ls

2 coreutils-5.2.1-31

rpm -qa将会列出给定系统上所有安装了的rpm包. rpm -qa package_name命令将会列出与给定名字package_name相匹配的包.

```
1 ash$ rpm -qa
2 redhat-logos-1.1.3-1
3 glibc-2.2.4-13
4 cracklib-2.7-12
5 dosfstools-2.7-1
6 gdbm-1.8.0-10
7 ksymoops-2.4.1-1
8 mktemp-1.5-11
9 perl-5.6.0-17
10 reiserfs-utils-3.x.0j-2
11 ...
12
13
14 bash$ rpm -qa docbook-utils
15 docbook-utils-0.6.9-2
16
17
18 bash$ rpm -qa docbook | grep docbook
19 docbook-dtd31-sgml-1.0-10
20 docbook-style-dsssl-1.64-3
21 docbook-dtd30-sgml-1.0-10
22 docbook-dtd40-sgml-1.0-11
23 docbook-utils-pdf-0.6.9-2
24 docbook-dtd41-sgml-1.0-10
25 docbook-utils-0.6.9-2
```

cpio.

这个特殊的归档拷贝命令(拷贝输入和输出, copy input and output) 现在已经很少能见到了, 因为它已经被tar/gzip所替代了. 现在这个命令只在一些比较特殊的地方还在使用, 比如拷贝一个目录树. 如果指定一个合适尺寸的块(用于拷贝), 那么这个命令会比tar命令快一些.

例子12-27. 使用cpio来拷贝一个目录树

```
1#!/bin/bash
2
3# 使用cpio来拷贝目录树.
4
5# 使用'cpio'的优点:
6# 加速拷贝. 比通过管道使用'tar'命令快一些.
7# 很适合拷贝一些'cp'命令
8#+ 搞不定的的特殊文件(比如名字叫pipes的文件, 等等)
9
10ARGS=2
11E_BADARGS=65
```

```

12
13 if [ $# -ne "$ARGS" ]
14 then
15   echo "Usage: `basename $0` source destination"
16   exit $E_BADARGS
17 fi
18
19 source=$1
20 destination=$2
21
22
23 find "$source" -depth | cpio -admvP "$destination"
24 #           ^^^^          ^^^^
25 # 阅读'find'和'cpio'的man页来了解这些选项的意义.
26
27
28 # 练习:
29 # -----
30
31 # 添加一些代码来检查'find | cpio'管道命令的退出码($?)
32 #+ 并且如果出现错误的时候输出合适的错误码.
33
34 exit 0

```

rpm2cpio.

这个命令可以从rpm归档文件中解出一个cpio归档文件.

例子12-28. 解包一个rpm归档文件

```

1#!/bin/bash
2# de-rpm.sh: 解包一个'rpm'归档文件
3
4: ${1?"Usage: `basename $0` target-file"}
5# 必须指定'rpm'归档文件名作为参数.
6
7
8 TEMPFILE=$$.cpio                      # Tempfile必须是一个"唯一"的名字.
9                                # $$是这个脚本的进程ID.
10
11 rpm2cpio < $1 > $TEMPFILE          # 将rpm归档文件转换为cpio归档文件.
12 cpio --make-directories -F $TEMPFILE -i # 解包cpio归档文件.
13 rm -f $TEMPFILE                      # 删除cpio归档文件.
14
15 exit 0
16
17 # 练习:

```

```
18 # 添加一些代码来检查    1) "target-file"是否存在  
19 #+                      2) 这个文件是否是一个rpm归档文件.  
20 # 暗示:                  分析'file'命令的输出.
```

压缩命令

gzip.

标准的GNU/UNIX压缩工具, 取代了比较差的compress命令. 相应的解压命令是gunzip, 与gzip -d是等价的.

-c选项将会把gzip的输出打印到stdout上. 当你想通过管道传递到其他命令的时候, 这就非常有用了.

zcat过滤器可以将一个gzip文件解压到stdout, 所以尽可能的使用管道和重定向. 这个命令事实上就是一个可以工作于压缩文件(包括一些的使用老的compress工具压缩的文件)的cat命令. zcat命令等价于gzip -dc.

⑧ 在某些商业的UNIX系统上, zcat与uncompress -c等价, 并且不能工作于gzip文件.

请参考[例子7-7](#).

bzip2.

用来压缩的一个可选的工具, 通常比gzip命令压缩率更高(所以更慢), 适用于比较大的文件. 相应的解压命令是bunzip2.

► 新版本的tar命令已经直接支持bzip2了.

compress, uncompress.

这是一个老的, 私有的压缩工具, 一般的商业UNIX发行版都会有这个工具. 更有效率的gzip工具早就把这个工具替换掉了. Linux发行版一般也会包含一个兼容的compress命令, 虽然gunzip也可以解压用compress工具压缩的文件.

► znew命令可以将compress压缩的文件转换为gzip压缩的文件.

sq.

另一种压缩工具, 一个只能工作于排过序的ASCII单词列表的过滤器. 这个命令使用过滤器标准的调用语法, sq < input-file > output-file. 速度很快, 但是效率远不及gzip. 相应的解压命令为unsq, 调用方法与sq相同.

► sq的输出可以通过管道传递给gzip, 以便于进一步的压缩.

zip, unzip.

跨平台的文件归档和压缩工具, 与DOS下的pkzip.exe兼容. "Zip"归档文件看起来在互联网上比"tar包"更流行.

unarc, unarj, unrar.

这些Linux工具可以用来解档那些用DOS下的arc.exe, arj.exe, 和rar.exe 程序进行归档的文件.

文件信息

file.

确定文件类型的工具。命令file file-name将会用ascii文本或数据的形式返回file-name文件的详细描述。这个命令会使用/usr/share/magic, /etc/magic, 或/usr/lib/magic中定义的魔法数字来标识包含某种魔法数字的文件，上边所举出的这3个文件需要依赖于具体的Linux/UNIX发行版。

-f选项将会让file命令运行于批处理模式，也就是说它会分析-f后边所指定的文件，从中读取需要处理的文件列表，然后依次执行file命令。-z选项，当对压缩过的目标文件使用时，将会强制分析压缩的文件类型。

```
1 bash$ file test.tar.gz
2 test.tar.gz: gzip compressed data, deflated, \
3 last modified: Sun Sep 16 13:34:51 2001, os: Unix
4
5 bash file -z test.tar.gz
6 test.tar.gz: GNU tar archive \
7 (gzip compressed data, deflated, \
8 last modified: Sun Sep 16 13:34:51 2001, os: Unix)
```

```
1 # 在给定的目录中找出sh和Bash脚本文件:
2
3 DIRECTORY=/usr/local/bin
4 KEYWORD=Bourne
5 # Bourne和Bourne-Again shell脚本
6
7 file $DIRECTORY/* | fgrep $KEYWORD
8
9 # 输出:
10
11 # /usr/local/bin/burn-cd:           Bourne-Again shell script text executable
12 # /usr/local/bin/burnit:            Bourne-Again shell script text executable
13 # /usr/local/bin/cassette.sh:       Bourne shell script text executable
14 # /usr/local/bin/copy-cd:           Bourne-Again shell script text executable
15 # . . .
```

例子12-29. 从C文件中去掉注释

```
1#!/bin/bash
2# strip-comment.sh: 去掉C程序中的注释 /* 注释 */.
3
4E_NOARGS=0
5E_ARGERROR=66
6E_WRONG_FILE_TYPE=67
7
8if [ $# -eq "$E_NOARGS" ]
9then
```

```
10 echo "Usage: 'basename $0' C-program-file" >&2 # 将错误消息发到stderr.
11 exit $E_ARGERROR
12 fi
13
14 # 检查文件类型是否正确.
15 type='file $1 | awk '{ print $2, $3, $4, $5 }'
16 # "file $1" echo出文件类型 . .
17 # 然后awk会删掉第一个域, 就是文件名 . .
18 # 然后结果将会传递到变量"type"中.
19 correct_type="ASCII C program text"
20
21 if [ "$type" != "$correct_type" ]
22 then
23   echo
24   echo "This script works on C program files only."
25   echo
26   exit $E_WRONG_FILE_TYPE
27 fi
28
29
30 # 相当隐秘的sed脚本:
31 #-----
32 sed '
33 /^\/\*/d
34 /.*\*\//d
35 ' $1
36 #-----
37 # 如果你花上几个小时来学习sed语法的话, 上边这个命令还是很好理解的.
38
39
40 # 如果注释和代码在同一行上, 上边的脚本就不行了.
41 #+ 所以需要添加一些代码来处理这种情况.
42 # 这是一个很重要的练习.
43
44 # 当然, 上边的代码也会删除带有"*/"的非注释行 --
45 #+ 这也不是一个令人满意的结果.
46
47 exit 0
48
49
50 # -----
51 # 下边的代码不会执行, 因为上边已经'exit 0'了.
52
53 # Stephane Chazelas建议使用下边的方法:
54
```

```

55 usage() {
56     echo "Usage: `basename $0` C-program-file" >&2
57     exit 1
58 }
59
60 WEIRD='echo -n -e '\377' # 或者WEIRD=$'\377'
61 [[ $# -eq 1 ]] || usage
62 case '$1' in
63     *"C program text") sed -e "s%/\*${WEIRD}%g;s%\*/%${WEIRD}%g" "$1" \
64         | tr '\377\n' '\n\377' \
65         | sed -ne 'p;n' \
66         | tr -d '\n' | tr '\377' '\n';;
67     *) usage;;
68 esac
69
70 # 如果是下列的这些情况，还是很糟糕：
71 # printf("/*");
72 # 或者
73 # /* /* buggy embedded comment */
74 #
75 # 为了处理上边所有这些特殊情况(字符串中的注释，含有 \"， \\\" ...
76 #+ 的字符串中的注释)唯一的方法还是写一个C分析器
77 #+ (或许可以使用lex或者yacc?).
78
79 exit 0

```

which.

which command-xxx将会给出”command-xxx”的完整路径. 当你想在系统中准确定位一个特定的命令或工具的时候, 这个命令就非常用了.

```

1 $bash which rm
2
3 /usr/bin/rm

```

whereis.

与上边的which很相似, whereis command-xxx不只会给出”command-xxx”的完整路径, 而且还会给出这个命令的man页的完整路径.

```

1 $bash whereis rm
2
3 rm: /bin/rm /usr/share/man/man1/rm.1.bz2

```

whatis .

whatis filexxx将会在whatis数据库中查询”filexxx”. 当你想确认系统命令和重要的配置文件

的时候,这个命令就非常重要了.可以把这个命令认为是一个简单的man命令.

```
1 $bash whatis whatis
2
3 whatis (1) - search the whatis database for complete words
```

例子12-30. 浏览/usr/X11R6/bin

```
1#!/bin/bash
2
3# 所有在/usr/X11R6/bin中的神秘2进制文件都是些什么东西?
4
5DIRECTORY="/usr/X11R6/bin"
6# 也试试 "/bin", "/usr/bin", "/usr/local/bin", 等等.
7
8for file in $DIRECTORY/*
9do
10    whatis `basename $file` # 将会echo出这个2进制文件的信息.
11done
12
13exit 0
14
15# 你可能希望将这个脚本的输出重定向, 像这样:
16# ./what.sh >>whatis.db
17# 或者一页一页的在stdout上察看,
18# ./what.sh | less
```

请参考[例子10-3](#).

vdir.

显示详细的目录列表.与ls -l的效果相似.

这是一个GNU fileutils.

```
1 bash$ vdir
2 total 10
3 -rw-r--r-- 1 bozo bozo 4034 Jul 18 22:04 data1.xrolo
4 -rw-r--r-- 1 bozo bozo 4602 May 25 13:58 data1.xrolo.bak
5 -rw-r--r-- 1 bozo bozo 877 Dec 17 2000 employment.xrolo
6
7 bash ls -l
8 total 10
9 -rw-r--r-- 1 bozo bozo 4034 Jul 18 22:04 data1.xrolo
10 -rw-r--r-- 1 bozo bozo 4602 May 25 13:58 data1.xrolo.bak
11 -rw-r--r-- 1 bozo bozo 877 Dec 17 2000 employment.xrolo
```

locate.

locate命令将会在预先建立好的档案数据库中查询文件. slocate命令是locate的安全版

本(locate命令很有可能已经被关联到slocate命令上了).

```
1 $bash locate hickson  
2  
3 /usr/lib/xephem/catalogs/hickson.edb
```

readlink.

显示符号链接所指向的文件.

```
1 bash$ readlink /usr/bin/awk  
2 ../../bin/gawk
```

strings.

使用strings命令在二进制或数据文件中找出可打印字符. 它将在目标文件中列出所有找到的可打印字符的序列. 这个命令对于想进行快速查找n个字符的打印检查来说是很方便的, 也可以用来检查一个未知格式的图片文件(strings image-file — more可能会搜索出像JFIF这样的字符串, 那么这就意味着这个文件是一个jpeg格式的图片文件). 在脚本中, 你可能会使用grep或者sed命令来分析strings命令的输出. 请参考[例子10-7](#)和[例子10-9](#).

例子12-31. 一个“改进过”的strings命令

```
1#!/bin/bash  
2# wstrings.sh: "word-strings" (增强的"strings"命令)  
3#  
4# 这个脚本将会通过排除标准单词列表的形式  
5#+ 来过滤"strings"命令的输出.  
6# 这将有效的过滤掉无意义的字符,  
7#+ 并且只会输出可以识别的字符.  
8  
9# =====  
10# 脚本参数的标准检查  
11ARGS=1  
12E_BADARGS=65  
13E_NOFILE=66  
14  
15if [ $# -ne $ARGS ]  
16then  
17    echo "Usage: `basename $0` filename"  
18    exit $E_BADARGS  
19fi  
20  
21if [ ! -f "$1" ] # 检查文件是否存在.  
22then  
23    echo "File \"$1\" does not exist."  
24    exit $E_NOFILE  
25fi  
26# =====
```

```

27
28
29 MINSTRLEN=3                      # 最小的字符串长度.
30 WORDFILE=/usr/share/dict/linux.words # 字典文件.
31 # 也可以指定一个不同的单词列表文件,
32 #+ 但这种文件必须是以每个单词一行的方式进行保存.
33
34
35 wlist='strings "$1" | tr A-Z a-z | tr '[space:]' Z | \
36 tr -cs '[alpha:]' Z | tr -s '\173-\377' Z | tr Z , '
37
38 # 将'strings'命令的输出通过管道传递到多个'tr'命令中.
39 # "tr A-Z a-z" 全部转换为小写字符.
40 # "tr '[space:]'" 转换空白字符为多个Z.
41 # "tr -cs '[alpha:]' Z" 将非字母表字符转换为多个Z,
42 #+ 然后去除多个连续的Z.
43 # "tr -s '\173-\377' Z" 把所有Z后边的字符都转换为Z.
44 #+ 并且去除多余重复的Z.
45 #+ (注意173(123 ascii "{})和377(255 ascii 最后一个字符)都是8进制)
46 #+ 这样处理之后, 我们所有之前需要处理的令我们头痛的字符
47 #+ 就全都转换为字符Z了.
48 # 最后"tr Z , , " 将把所有的Z都转换为空格,
49 #+ 这样我们在下边循环中用到的变量wlist中的内容就全部以空格分隔了.
50
51 # ****
52 # 注意, 我们使用管道来将多个'tr'的输出传递到下一个'tr'时
53 #+ 每次都使用了不同的参数.
54 # ****
55
56
57 for word in $wlist                  # 重要:
58                                         # $wlist 这里不能使用双引号.
59                                         # "$wlist" 不能正常工作.
60                                         # 为什么不行?
61 do
62
63     strlen=${#word}                  # 字符串长度.
64     if [ "$strlen" -lt "$MINSTRLEN" ] # 跳过短的字符串.
65     then
66         continue
67     fi
68
69     grep -Fw $word "$WORDFILE"       # 只匹配整个单词.
70 #     ^^^
71                                         # "固定字符串" 和
72                                         #+ "整个单词" 选项.

```

```
72  
73 done  
74  
75  
76 exit $?
```

比较 .

diff, patch .

diff: 一个非常灵活的文件比较工具. 这个工具将会以一行接一行的形式来比较目标文件. 在某些应用中, 比如说比较单词词典, 在通过管道将结果传递给diff命令之前, 使用诸如sort和uniq命令来对文件进行过滤将是非常有用的. diff file-1 file-2 将会输出两个文件中不同的行, 并会通过符号标识出每个不同行所属的文件.

diff命令的-side-by-side选项将会按照左右分隔的形式, 把两个比较中的文件全部输出, 并且会把不同的行标记出来. -c和-u选项也会使得diff命令的输出变得容易解释一些.

还有一些diff命令的变种, 比如sdiff, wdiff, xdiff, 和mgdiff.

► 如果比较的两个文件是完全一样的话, 那么diff命令会返回0作为退出状态码, 如果不同的话就返回1作为退出码. 这样diff命令就可以用在shell脚本的测试结构中了. (见下边).

diff命令的一个重要用法就是产生区别文件, 这个文件将用作patch命令的-e选项的参数, -e选项接受ed或ex脚本.

patch: 灵活的版本工具. 给出一个用diff命令产生的区别文件, patch命令可以将一个老版本的包更新为一个新版本的包. 因为你发布一个小的”区别”文件远比重新发布一个大的软件包来的容易得多. 对于频繁更新的Linux内核来说, 使用内核”补丁包”的形式来发布是一种非常好的办法.

```
1 patch -p1 <patch-file  
2 # 在'patch-file'中取得所有的修改列表,  
3 # 然后把它们更新到相应的文件中.  
4 # 那么这个包就被更新为新版本了.
```

更新内核:

```
1 cd /usr/src  
2 gzip -cd patchXX.gz | patch -p0  
3 # 使用'patch'来更新内核源文件.  
4 # 来自于linux内核文档"README",  
5 # 这份文档由匿名作者(Alan Cox?)所编写.
```

► diff也可以递归的比较目录下的所有文件(包含子目录).

```
1 bash$ diff -r ~/notes1 ~/notes2  
2 Only in /home/bozo/notes1: file02  
3 Only in /home/bozo/notes1: file03  
4 Only in /home/bozo/notes2: file04
```

► 使用zdiff来比较gzip文件.

diff3 .

这是diff命令的扩展版本, 可以同时比较三个文件. 如果成功执行那么这个命令就返回0, 但不幸的是这个命令不给出比较结果的信息.

```
1 bash$ diff3 file-1 file-2 file-3
2 ====
3 1:1c
4     This is line 1 of "file-1".
5 2:1c
6     This is line 1 of "file-2".
7 3:1c
8     This is line 1 of "file-3"
```

sdiff .

比较和(或)编辑两个文件, 将它们合并到一个输出文件中. 由于这个命令的交互特性, 所以在脚本中很少使用这个命令.

cmp .

cmp命令是上边diff命令的一个简单版本. diff命令会报告两个文件的不同之处, 而cmp命令仅仅指出哪些位置有所不同, 不会显示不同之处的具体细节.

⑧ 就像diff命令那样, 如果两个文件相同的话, cmp将返回0作为退出状态码, 如果不同就返回1. 这样能用在shell脚本的测试结构中了.

例子12-32. 在一个脚本中使用cmp命令来比较两个文件.

```
1 #!/bin/bash
2
3 ARGS=2 # 脚本需要两个参数.
4 E_BADARGS=65
5 E_UNREADABLE=66
6
7 if [ $# -ne "$ARGS" ]
8 then
9   echo "Usage: `basename $0` file1 file2"
10  exit $E_BADARGS
11 fi
12
13 if [[ ! -r "$1" || ! -r "$2" ]]
14 then
15   echo "Both files to be compared must exist and be readable."
16   exit $E_UNREADABLE
17 fi
18
19 cmp $1 $2 &> /dev/null # /dev/null将会禁止"cmp"命令的输出.
20 #   cmp -s $1 $2 与上边这句的结果相同("-s"选项是禁止输出(silent)标志)
21 #   感谢Anders Gustavsson指出这点.
22 #
23 # 使用'diff'命令也可以, 比如,   diff $1 $2 &> /dev/null
```

```
24
25 if [ $? -eq 0 ]          # 测试"cmp"命令的退出状态.
26 then
27   echo "File \"$1\" is identical to file \"$2\"."
28 else
29   echo "File \"$1\" differs from file \"$2\"."
30 fi
31
32 exit 0
```

► 使用zcmp处理gzip文件.

comm .

多功能的文件比较工具. 使用这个命令之前必须先排序.

comm -options first-file second-file

comm file-1 file-2 将会输出3列:

第1列= 只在file-1中存在的行

第2列= 只在file-2中存在的行

第3列= 两边相同的行.

下列选项可以禁止一列或多列的输出.

-1 禁止显示第1列(译者: 在File1中的行)

-2 禁止显示第2列(译者: 在File2中的行)

-3 禁止显示第3列(译者: 在File3中的行)

-12 禁止第1列和第2列, 等等. (译者: 就是说选项可以组合)

工具 .

basename .

从文件名中去掉路径信息, 只打印出文件名. 结构basename \$0可以让脚本获得它自己的名字, 也就是, 它被调用的名字. 可以用来显示"用法"信息, 比如如果你调用脚本的时候缺少参数, 可以使用如下语句:

```
1 echo "Usage: `basename $0` arg1 arg2 ... argn"
```

dirname .

从带路径的文件名字符串中去掉文件名(basename), 只打印出路径信息.

► basename和dirname可以操作任意字符串. 它们的参数不一定是一个真正存在的文件, 甚至可以不是一个文件名. (请参考[例子A-7](#)).

例子12-33. basename和dirname

```
1#!/bin/bash
2
3 a=/home/bozo/daily-journal.txt
4
5 echo "Basename of /home/bozo/daily-journal.txt = `basename $a`"
```

```
6 echo "Dirname of /home/bozo/daily-journal.txt = `dirname $a`"
7 echo
8 echo "My own home is `basename ~/.`"          # `basename ~` 也可以.
9 echo "The home of my home is `dirname ~/.`"    # `dirname ~` 也可以.
10
11 exit 0
```

split .

将一个文件分割为几个小段的工具. 这些命令通常会将大的文件分割, 然后备份到软盘上, 或者是为了将大文件切成合适的尺寸, 然后用email上传.

csplit命令会根据上下文来切割文件, 切割的位置将会发生在模式匹配的地方.

sum, cksum, md5sum, sha1sum .

这些都是用来产生checksum的工具. checksum是对文件的内容进行数学计算而得到的, 它的目的是用来检验文件的完整性, 出于安全目的一个脚本可能会有一个checksum列表, 这样可以确保关键系统文件的内容不会被修改或损坏. 对于需要安全性的应用来说, 应该使用md5sum (message digest 5 checksum)命令, 或者使用更好更新的sha1sum命令(安全Hash算法).

```
1 bash$ cksum /boot/vmlinuz
2 1670054224 804083 /boot/vmlinuz
3
4 bash$ echo -n "Top Secret" | cksum
5 3391003827 10
6
7
8
9 bash$ md5sum /boot/vmlinuz
10 0f43eccea8f09e0a0b2b5cf1dcf333ba  /boot/vmlinuz
11
12 bash$ echo -n "Top Secret" | md5sum
13 8babc97a6f62a4649716f4df8d61728f  -
```

► cksum将会显示目标尺寸(以字节为单位), 目标可以是stdout, 也可以是文件.

md5sum和sha1sum命令在它们收到来自于stdout的输入的时候, 显示一个dash.

例子12-34. 检查文件完整性

```
1#!/bin/bash
2# file-integrity.sh: 检查一个给定目录下的文件
3#                   是否被改动了.
4
5E_DIR_NOMATCH=70
6E_BAD_DBFILE=71
7
8dbfile=File_record.md5
9# 存储记录的文件名(数据库文件).
10
```

```
11 11
12 12 set_up_database ()
13 13 {
14 14   echo ""$directory"" > "$dbfile"
15 15   # 把目录名写到文件的第一行.
16 16   md5sum "$directory"/* >> "$dbfile"
17 17   # 在文件中附上md5 checksum和filename.
18 18 }
19 19
20 20 check_database ()
21 21 {
22 22   local n=0
23 23   local filename
24 24   local checksum
25 25
26 26   # -----
27 27   # 这个文件检查其实是不必要的,
28 28   #+ 但是能更安全一些.
29 29
30 30   if [ ! -r "$dbfile" ]
31 31   then
32 32     echo "Unable to read checksum database file!"
33 33     exit $E_BAD_DBFILE
34 34   fi
35 35   # -----
36 36
37 37   while read record[n]
38 38   do
39 39
40 40     directory_checked="${record[0]}"
41 41     if [ "$directory_checked" != "$directory" ]
42 42     then
43 43       echo "Directories do not match up!"
44 44       # 换个目录试一下.
45 45       exit $E_DIR_NOMATCH
46 46     fi
47 47
48 48     if [ "$n" -gt 0 ]    # 不是目录名.
49 49     then
50 50       filename[n]=$( echo ${record[$n]} | awk '{ print $2 }' )
51 51       # md5sum向后写记录,
52 52       #+ 先写checksum, 然后写filename.
53 53       checksum[n]=$( md5sum "${filename[n]}" )
54 54
55 55
```

```
56     if [ "${record[n]}" = "${checksum[n]}" ]
57     then
58         echo "${filename[n]} unchanged."
59
60     elif [ `basename ${filename[n]}` != "$dbfile" ]
61         # 跳过checksum数据库文件,
62         #+ 因为在每次调用脚本它都会被修改.
63     #
64         # 这不幸的意味着当我们在$PWD中运行这个脚本时候,
65         #+ 篡改这个checksum数
66         #+ 数据库文件将不会被检测出来.
67         # 练习: 修正这个问题.
68     then
69         echo "${filename[n]} : CHECKSUM ERROR!"
70         # 从上次的检查之后, 文件已经被修改.
71     fi
72
73     fi
74
75
76
77     let "n+=1"
78 done <"$dbfile"          # 从checksum数据库文件中读.
79
80 }
81
82 # ===== #
83 # main ()
84
85 if [ -z "$1" ]
86 then
87     directory="$PWD"      # 如果没指定参数的话,
88 else
89     directory="$1"          #+ 那么就使用当前的工作目录.
90 fi
91
92 clear                  # 清屏.
93 echo " Running file integrity check on $directory"
94 echo
95
96 # ----- #
97 if [ ! -r "$dbfile" ] # 是否需要建立数据库文件?
98 then
99     echo "Setting up database file, \\""$directory"/"$dbfile"\\"; echo
100    set_up_database
```

```
101 101 fi
102 # -----
103 103
104 104 check_database      # 调用主要处理函数.
105 105
106 106 echo
107 107
108 108 # 你可能想把这个脚本的输出重定向到文件中,
109 109 #+ 尤其在这个目录中有很多文件的时候.
110 110
111 111 exit 0
112 112
113 113 # 如果要对数量非常多的文件做完整性检查,
114 114 #+ 可以考虑一下"Tripwire"包,
115 115 #+ http://sourceforge.net/projects/tripwire/.
116 116
```

请参考[例子A-19](#)和[例子33-14](#), 这两个例子展示了md5sum命令的用法.

④ 到目前为止, 已经有128-bit的md5sum被破解的报告了,

所以现在更安全的160-bit的sha1sum非常受欢迎, 这个命令已经被加入到checksum工具包中了.

一些安全顾问认为即使是sha1sum也是一种折衷的做法. 所以, 下一个工具是什么呢? -- 512-bit的checksum工具?

```
1 bash$ md5sum testfile
2 e181e2c8720c60522c4c4c981108e367  testfile
3
4 bash$ sha1sum testfile
5 5d7425a9c08a66c3177f1e31286fa40986ffc996  testfile
```

shred .

用随机字符填充文件, 使得文件无法恢复, 这样就可以保证文件安全的被删除. 这个命令的效果与[例子12-55](#)一样, 但是使用这个命令是一种更优雅更彻底的方法.

这是GNU的文件工具之一.

④ 即使使用了shred命令, 高级的辨别技术也还是能够恢复文件的内容.

编码和解码 .

uuencode .

这个工具用来把二进制文件编码成ASCII字符串, 这个工具适用于编码e-mail消息体, 或者新闻组消息.

uudecode .

这个工具用来把uuencode后的ASCII字符串恢复为二进制文件.

例子12-35. Uudecode编码后的文件

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

```

2 # 在当前目录下用Uudecode解码所有用uuencode编码的文件.
3
4 lines=35          # 允许读头部的35行(范围很宽).
5
6 for File in *    # 测试所有$PWD下的文件.
7 do
8   search1='head -$lines $File | grep begin | wc -w'
9   search2='tail -$lines $File | grep end | wc -w'
10  # 用Uuencode编码过的文件在文件开始的地方都有个"begin",
11  #+ 在文件结尾的地方都有"end".
12  if [ "$search1" -gt 0 ]
13  then
14    if [ "$search2" -gt 0 ]
15    then
16      echo "uudecoding - $File -"
17      uudecode $File
18    fi
19  fi
20 done
21
22 # 小心不要让这个脚本运行自己,
23 #+ 因为它也会把自身也认为是一个经过uuencode编码过的文件,
24 #+ 这都是因为这个脚本自身也包含"begin"和"end".
25
26 # 练习:
27 # -----
28 # 修改这个脚本, 让它可以检查一个新闻组的每个文件,
29 #+ 并且如果下一个没找到的话就跳过.
30
31 exit 0

```

fold .

-s命令在处理从Usenet新闻组下载下来的长的uudecode文本消息的时候可能会有用(可能在管道中).

mimencode .

mimencode和mmencode命令用来处理多媒体编码的email附件.

虽然mail用户代理(比如pine或kmail)通常情况下都会自动处理, 但是这些特定的工具允许从命令行或shell脚本中来手动操作这些附件.

crypt .

这个工具曾经是标准的UNIX文件加密工具[3]. 政府由于政策上的动机规定禁止加密软件的输出, 这样导致了crypt命令从UNIX世界消失, 并且在大多数的Linux发行版中也没有这个命令. 幸运的是, 程序员们想出了一些替代它的方法, 在这些方法中有作者自己的crupt (请参考[例](#)

子A-4).

其他杂项 .

mktemp .

使用一个“唯一”的文件名来创建一个临时文件. [4] 如果不带参数的在命令行下调用这个命令时, 将会在/tmp目录下产生一个零长度的文件.

```
1 bash$ mktemp  
2 /tmp/tmp.zzsSQL3154
```

```
1 PREFIX=filename  
2 tempfile='mktemp $PREFIX.XXXXXX'  
3 #           ^~~~~~ 在这个临时的文件名中  
4 #+          至少需要6个占位符.  
5 #  如果没有指定临时文件的文件名,  
6 #+ 那么默认就是 "tmp.XXXXXXXXXXX".  
7  
8 echo "tempfile name = $tempfile"  
9 # tempfile name = filename.QA2ZpY  
10 #          或者一些其他的相似的名字...  
11  
12 # 使用600为文件权限  
13 #+ 来在当前工作目录下创建一个这样的文件.  
14 # 这样就不需要 "umask 177" 了.  
15 # 但不管怎么说, 这也是一个好的编程风格.
```

make .

build(建立)和compile(编译)二进制包的工具. 当源文件被增加或修改时就会触发一些操作, 这个工具用来控制这些操作.

make命令会检查Makefile, makefile是文件的依赖和操作列表.

install .

特殊目的的文件拷贝命令, 与cp命令相似, 但是具有设置拷贝文件的权限和属性的能力. 这个命令看起来是为了安装软件包所定制的, 而且就其本身而言, 这个命令经常出现在Makefiles中(在make install : 区域中). 在安装脚本中也会看到这个命令的使用.

dos2unix .

这个工具是由Benjamin Lin及其同事共同编写的, 目的是将DOS格式的文本文件(以CR-LF为行结束符)转换为UNIX格式(以LF为行结束符), 反过来也一样.

ptx .

ptx [targetfile]命令将输出目标文件的序列改变索引(交叉引用列表). 如果必要的话, 这个命令可以在管道中进行更深层次的过滤和格式化.

more, less .

分页显示文本文件或stdout, 一次一屏. 可以用来过滤stdout的输出... 或过滤一个脚本的输出.

more命令的一个有趣的应用就是测试一个命令序列的执行, 这样做的目的是避免可能发生的糟糕的结果.

```
1 ls /home/bozo | awk '{print "rm -rf \"\$1\""}' | more
2 #                                     ^^^^
3
4 # 检查下边(灾难性的)命令行的效果:
5 #     ls /home/bozo | awk '{print "rm -rf \"\$1\""}' | sh
6 #     推入shell中执行 . . .
```

注意事项 .

1. 在这里所讨论的归档文件, 只不过是存储在一个单一位置上的一些相关文件的集合.
2. tar czvf archive_name.tar.gz *可以包含当前目录下的点文件.
这是一个未文档化的GNUTar的”特性”.
3. 这是一个对称的块密码, 过去曾在单系统或本地网络中用来加密文件, 用来对抗”public key”密码类, pgp就是一个众所周知的例子.
4. 使用-d选项可以创建一个临时的目录.

6 通讯命令

下边命令中的某几个命令你会在追踪垃圾邮件练习中找到其用法, 用来进行网络数据的转换和分析.

信息与统计

host .

通过名字或IP地址来搜索一个互联网主机的信息, 使用DNS.

```
1 bash$ host surfacemail.com
2 surfacemail.com. has address 202.92.42.236
```

ipcalc .

显示一个主机IP信息. 使用-h选项, ipcalc将会做一个DNS的反向查询, 通过IP地址找到主机(服务器)名.

```
1 bash$ ipcalc -h 202.92.42.236
2 HOSTNAME=surfacemail.com
```

nslookup .

通过IP地址在一个主机上做一个互联网的”名字服务查询”. 事实上, 这与ipcalc -h或dig -x等价. 这个命令既可以交互运行也可以非交互运行, 换句话说, 就是在脚本中运行.

nslookup命令据说已经被慢慢的”忽视”了, 但事实上它是有一定的作用.

```
1 bash$ nslookup -sil 66.97.104.180
2 nslookup kuhleersparnis.ch
3 Server:          135.116.137.2
4 Address:         135.116.137.2#53
5
6 Non-authoritative answer:
7 Name:      kuhleersparnis.ch
```

dig .

Domain Information Groper(域信息查询). 与nslookup很相似, dig也可以在一个主机上做互联网的”名字服务查询”. 这个命令既可以交互运行也可以非交互运行, 换句话说, 就是在脚本中运行.

下面是一些dig命令有趣的选项, +time=N选项用来设置查询超时为N秒, +nofail选项用来持续查询服务器直到收到一个响应, -x会做反向地址查询.

比较下边这3个命令的输出, dig -x, ipcalc -h和nslookup.

```
1 bash$ dig -x 81.9.6.2
2 ;; Got answer:
3 ;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 11649
4 ;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 0
```

```

5
6 ;; QUESTION SECTION:
7 ;2.6.9.81.in-addr.arpa.      IN      PTR
8
9 ;; AUTHORITY SECTION:
10 6.9.81.in-addr.arpa.    3600    IN      SOA     ns_eltel.net. noc_eltel.net.
11 2002031705 900 600 86400 3600
12
13 ;; Query time: 537 msec
14 ;; SERVER: 135.116.137.2#53(135.116.137.2)
15 ;; WHEN: Wed Jun 26 08:35:24 2002
16 ;; MSG SIZE rcvd: 91

```

例子12-36. 查找滥用的链接来报告垃圾邮件发送者

```

1#!/bin/bash
2# spam-lookup.sh: 查找滥用的连接来报告垃圾邮件发送者.
3# 感谢Michael Zick.
4
5# 检查命令行参数.
6ARGCOUNT=1
7E_WRONGARGS=65
8if [ $# -ne "$ARGCOUNT" ]
9then
10 echo "Usage: `basename $0` domain-name"
11 exit $E_WRONGARGS
12fi
13
14
15dig +short $1.contacts.abuse.net -c in -t txt
16# 也试试:
17#     dig +nssearch $1
18#     尽量找到"可信赖的名字服务器"并且显示SOA记录.
19
20# 下边这句也可以:
21#     whois -h whois.abuse.net $1
22#             ^~ ~~~~~ 指定主机.
23#     使用这个命令也可以查找多个垃圾邮件发送者, 比如:
24#     whois -h whois.abuse.net $spamdomain1 $spamdomain2 . .
25
26
27# 练习:
28# -----
29# 扩展这个脚本的功能,
30#+ 让它可以自动发送e-mail来通知
31#+ 需要对此负责的ISP的联系地址.

```

```

32 # 暗示：使用"mail"命令。
33
34 exit $?
35
36 # spam-lookup.sh chinatietong.com
37 #           一个已知的垃圾邮件域. (译者：中国铁通 . . .)
38
39 # "crnet_mgr@chinatietong.com"
40 # "crnet_tec@chinatietong.com"
41 # "postmaster@chinatietong.com"
42
43
44 # 如果想找到这个脚本的一个更详尽的版本,
45 #+ 请访问SpamViz的主页，http://www.spamviz.net/index.html.

```

例子12-37. 分析一个垃圾邮件域

```

1 #! /bin/bash
2 # is-spammer.sh: 鉴别一个垃圾邮件域
3
4 # $Id: is-spammer, v 1.4 2004/09/01 19:37:52 mszick Exp $
5 # 上边这行是RCS ID信息.
6 #
7 # 这是附件中捐献脚本is_spammer.bash
8 #+ 的一个简单版本.
9
10 # is-spammer <domain.name>
11
12 # 使用外部程序：'dig'
13 # 测试版本：9.2.4rc5
14
15 # 使用函数.
16 # 使用IFS来分析分配在数组中的字符串.
17 # 做一些有用的事：检查e-mail黑名单.
18
19 # 使用来自文本体中的domain.name:
20 # http://www.good\_stuff.spammer.biz/just\_ignore\_everything\_else
21 # ^^^^^^^^^^
22 # 或者使用来自任意e-mail地址的domain.name:
23 # Really_Good_Offer@spammer.biz
24 #
25 # 并将其作为这个脚本的唯一参数.
26 #(另：你的Inet连接应该保证连接好)
27 #
28 # 这样，在上边两个实例中调用这个脚本：
29 #       is-spammer.sh spammer.biz

```

```
30
31
32 # Whitespace == :Space:Tab:Line Feed:Carriage Return:
33 WSP_IFS=$'\x20'$'\x09'$'\x0A'$'\x0D'
34
35 # No Whitespace == Line Feed:Carriage Return
36 No_WSP=$'\x0A'$'\x0D'
37
38 # 域分隔符为点分10进制ip地址
39 ADR_IFS=${No_WSP}('.')
40
41 # 取得dns文本资源记录.
42 # get_txt <error_code> <list_query>
43 get_txt() {
44
45     # 分析在"."中分配的$1.
46     local -a dns
47     IFS=$ADR_IFS
48     dns=( $1 )
49     IFS=$WSP_IFS
50     if [ "${dns[0]}" == '127' ]
51     then
52         # 查看此处是否有原因.
53         echo $(dig +short $2 -t txt)
54     fi
55 }
56
57 # 取得dns地址资源纪录.
58 # chk_adr <rev_dns> <list_server>
59 chk_adr() {
60     local reply
61     local server
62     local reason
63
64     server=${1}${2}
65     reply=$( dig +short ${server} )
66
67     # 假设应答可能是一个错误码 . .
68     if [ ${#reply} -gt 6 ]
69     then
70         reason=$(get_txt ${reply} ${server} )
71         reason=${reason:-${reply}}
72     fi
73     echo ${reason:-' not blacklisted.'}
74 }
```

```
75
76 # 需要从名字中取得 IP 地址.
77 echo 'Get address of: '$1
78 ip_adr=$(dig +short $1)
79 dns_reply=${ip_adr:-' no answer '}
80 echo ' Found address: '${dns_reply}
81
82 # 一个可用的应答至少是4个数字加上3个点.
83 if [ ${#ip_adr} -gt 6 ]
84 then
85     echo
86     declare query
87
88     # 通过点中的分配进行分析.
89     declare -a dns
90     IFS=$ADR_IFS
91     dns=( ${ip_adr} )
92     IFS=$WSP_IFS
93
94     # 用8进制表示法将dns查询循序记录起来.
95     rev_dns="${dns[3]}${dns[2]}${dns[1]}${dns[0]}"
96
97 # 查看: http://www.spamhaus.org (传统地址, 维护的很好)
98 echo -n 'spamhaus.org says: '
99 echo $(chk_adr ${rev_dns} 'sbl-xbl.spamhaus.org')
100
101 # 查看: http://ordb.org (开放转发Open mail relay)
102 echo -n ' ordbo.org says: '
103 echo $(chk_adr ${rev_dns} 'relays.ordb.org')
104
105 # 查看: http://www.spamcop.net/ (你可以在这里报告spammer)
106 echo -n ' spamcop.net says: '
107 echo $(chk_adr ${rev_dns} 'bl.spamcop.net')
108
109 # # # 其他的黑名单操作 # # #
110
111 # 查看: http://cbl.abuseat.org.
112 echo -n ' abuseat.org says: '
113 echo $(chk_adr ${rev_dns} 'cbl.abuseat.org')
114
115 # 查看: http://dsbl.org/usage (不同的邮件转发mail relay)
116 echo
117 echo 'Distributed Server Listings'
118 echo -n ' list.dsbl.org says: '
119 echo $(chk_adr ${rev_dns} 'list.dsbl.org')
```

```

120
121     echo -n '    multihop.dsbl.org says: '
122     echo $(chk_adr ${rev_dns} 'multihop.dsbl.org')
123
124     echo -n 'unconfirmed.dsbl.org says: '
125     echo $(chk_adr ${rev_dns} 'unconfirmed.dsbl.org')
126
127 else
128     echo
129     echo 'Could not use that address.'
130 fi
131
132 exit 0
133
134 # 练习:
135 # -----
136
137 # 1) 检查脚本参数,
138 # 并且如果必要的话, 可以使用合适的错误消息退出.
139
140 # 2) 察看调用这个脚本的时候是否在线,
141 # 并且如果必要的话, 可以使用合适的错误消息退出.
142
143 # 3) 用一般变量来替换掉"硬编码"的BHL domain.
144
145 # 4) 通过对'dig'命令使用"+time="选项
146 # 来给这个脚本设置一个暂停.

```

想获得比上边这个脚本更详细的版本, 请参考[例子A-28](#).

traceroute .

跟踪包发送到远端主机过程中的路由信息. 这个命令在LAN, WAN, 或者在Internet上都可以正常工作. 远端主机可以通过IP地址来指定. 这个命令的输出也可以通过管道中的grep或sed命令来过滤.

```

1 bash$ traceroute 81.9.6.2
2 traceroute to 81.9.6.2 (81.9.6.2), 30 hops max, 38 byte packets
3  1 tc43.xjbnnbrb.com (136.30.178.8)  191.303 ms  179.400 ms  179.767 ms
4  2 or0.xjbnnbrb.com (136.30.178.1)  179.536 ms  179.534 ms  169.685 ms
5  3 192.168.11.101 (192.168.11.101)  189.471 ms  189.556 ms *
6  ...

```

ping .

广播一个"ICMP ECHO_REQUEST"包到其他主机上, 既可以是本地网络也可以是远端网络. 这是一个测试网络连接的诊断工具, 应该小心使用.

如果ping成功之行, 那么返回的退出状态码为0. 可以用在脚本的测试语句中.

```
1 bash$ ping localhost
2 PING localhost.localdomain (127.0.0.1) from 127.0.0.1 : 56(84) bytes of data.
3 64 bytes from localhost.localdomain (127.0.0.1): \
4     icmp_seq=0 ttl=255 time=709 usec
5 64 bytes from localhost.localdomain (127.0.0.1): \
6     icmp_seq=1 ttl=255 time=286 usec
7
8 --- localhost.localdomain ping statistics ---
9 2 packets transmitted, 2 packets received, 0% packet loss
10 round-trip min/avg/max/mdev = 0.286/0.497/0.709/0.212 ms
```

whois .

执行DNS(域名系统)查询. -h选项允许指定需要查询的特定whois服务器. 请参考[例子4-6](#)和[例子12-36](#).

finger .

取得网络上的用户信息. 另外这个命令可以显示一个用户的 /.plan, /.project, 和 /.forward文件, 当然, 前提是如果这些文件存在的话.

```
1 bash$ finger
2 Login   Name          Tty      Idle  Login Time   Office       Office Phone
3 bozo    Bozo Bozeman  tty1        8 Jun 25 16:59
4 bozo    Bozo Bozeman  tttyp0      Jun 25 16:59
5 bozo    Bozo Bozeman  tttyp1      Jun 25 17:07
6
7
8 bash$ finger bozo
9 Login: bozo                      Name: Bozo Bozeman
10 Directory: /home/bozo             Shell: /bin/bash
11 Office: 2355 Clown St., 543-1234
12 On since Fri Aug 31 20:13 (MST) on tty1   1 hour 38 minutes idle
13 On since Fri Aug 31 20:13 (MST) on pts/0   12 seconds idle
14 On since Fri Aug 31 20:13 (MST) on pts/1
15 On since Fri Aug 31 20:31 (MST) on pts/2   1 hour 16 minutes idle
16 No mail.
17 No Plan.
```

出于安全上的考虑, 许多网络都禁用了finger, 以及和它相关的幽灵进程. [\[1\]](#)

chfn .

修改finger命令所显示出来的用户信息.

vrfy .

验证一个互联网的e-mail地址.

远端主机接入

sx, rx .

sx和rx命令使用xmodem协议, 置服务来向远端主机传输文件和接收文件. 这些都是通讯安装包的一般部分, 比如minicom.

sz, rz .

sz和rz命令使用zmodem协议, 设置服务来向远端主机传输文件和接收文件. Zmodem协议在某些方面比xmodem协议强, 比如使用更快的传输波特率, 并且可以对中断的文件进行续传. 与sx和rx一样, 这些都是通讯安装包的一般部分.

ftp .

向远端服务器上传或下载的工具, 也是一种协议. 一个ftp会话可以写到脚本中自动运行. (请参考例子17-6, 例子A-4, 和例子A-13).

uucp, uux, cu .

uucp: UNIX到UNIX拷贝. 这是一个通讯安装包, 目的是为了在UNIX服务器之间传输文件. 使用shell脚本来处理uucp命令序列是一种有效的方法.

因为互联网和电子邮件的出现, uucp现在看起来已经很落伍了, 但是这个命令在互联网连接不可用或者不适合使用的地方, 这个命令还是可以完美的运行. uucp的优点就是它的容错性, 即使有一个服务将拷贝操作中断了, 那么当连接恢复的时候, 这个命令还是可以在中断的地方续传.

—
uux: UNIX到UNIX执行. 在远端系统上执行一个命令. 这个命令是uucp包的一部分.

—
cu: Call Up 一个远端系统并且作为一个简单终端进行连接. 这是一个telnet的缩减版本. 这个命令是uucp包的一部分.

telnet .

连接远端主机的工具和协议.

⑧

telnet协议本身包含安全漏洞, 因此我们应该适当的避免使用.

wget .

wget工具使用非交互的形式从web或ftp站点上取得或下载文件. 在脚本中使用正好.

```
1 wget -p http://www.xyz23.com/file01.html
2 # -p或--page-requisite选项将会使得wget取得所有在显示指定页时
3 #+ 所需要的文件. (译者: 比如内嵌图片和样式表等.)
4
5 wget -r ftp://ftp.xyz24.net/~bozo/project_files/ -O $SAVEFILE
6 # -r选项将会递归的从指定站点
7 #+ 上下载所有连接.
```

例子12-38. 获得一份股票报价

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

```

2 # quote-fetch.sh: 下载一份股票报价.
3
4
5 E_NOPARAMS=66
6
7 if [ -z "$1" ] #必须指定需要获取的股票(代号).
8   then echo "Usage: `basename $0` stock-symbol"
9   exit $E_NOPARAMS
10 fi
11
12 stock_symbol=$1
13
14 file_suffix=.html
15 # 获得一个HTML文件, 所以要正确命名它.
16 URL='http://finance.yahoo.com/q?s='
17 # Yahoo金融板块, 后缀是股票查询.
18
19 # -----
20 wget -O ${stock_symbol}${file_suffix} "${URL}${stock_symbol}"
21 # -----
22
23
24 # 在http://search.yahoo.com上查询相关材料:
25 # -----
26 # URL="http://search.yahoo.com/search?fr=ush-news&p=${query}"
27 # wget -O "$savefilename" "${URL}"
28 # -----
29 # 保存相关URL的列表.
30
31 exit $?
32
33 # 练习:
34 # -----
35 #
36 # 1) 添加一个测试来验证用户是否在线.
37 #   (暗示: 对"ppp"或"connect"来分析'ps -ax'的输出.
38 #
39 # 2) 修改这个脚本, 让这个脚本具有获得本地天气预报的能力,
40 #+ 将用户的zip code作为参数.

```

请参考[例子A-30](#)和[例子A-31](#).

lynx .

lynx是一个网页浏览器, 也是一个文件浏览器. 它可以(通过使用-dump选项)在脚本中使用.

它的作用是可以非交互的从Web或ftp站点上获得文件.

```
1 lynx -dump http://www.xyz23.com/file01.html >$SAVEFILE
```

使用-traversal选项, lynx将会从参数中指定的HTTP URL开始, ”遍历”指定服务器上的所有连接. 如果与-crawl选项一起用的话, 将会把每个输出的页面文本都放到一个log文件中.

rlogin .

远端登陆, 在远端的主机上开启一个会话. 这个命令存在安全隐患, 所以要使用ssh来代替.

rsh .

远端shell, 在远端的主机上执行命令. 这个命令存在安全隐患, 所以要使用ssh来代替.

rcp .

远端拷贝, 在网络上的不同主机间拷贝文件.

rsync .

远端同步, 在网络上的不同主机间(同步)更新文件.

```
1 bash$ rsync -a ~/sourcedir/*txt /node1/subdirectory/
```

例子12-39. 更新FC4(Fedora 4)

```
1 #!/bin/bash
2 # fc4upd.sh
3
4 # 脚本作者: Frank Wang.
5 # 本书作者作了少量修改.
6 # 授权在本书中使用.
7
8
9 # 使用rsync命令从镜像站点上下载Fedora 4的更新.
10 # 为了节省空间, 如果有多个版本存在的话,
11 #+ 只下载最新的包.
12
13 URL=rsync://distro.ibiblio.org/fedora-linux-core/updates/
14 # URL=rsync://ftp.kddilabs.jp/fedora/core/updates/
15 # URL=rsync://rsync.planetmirror.com/fedora-linux-core/updates/
16
17 DEST=${1:-/var/www/html/fedora/updates/}
18 LOG=/tmp/repo-update-$(/bin/date +%Y-%m-%d).txt
19 PID_FILE=/var/run/${0##*/}.pid
20
21 E_RETURN=65      # 某些意想不到的错误.
22
23
24 # 一般rsync选项
25 # -r: 递归下载
```

```
26 # -t: 保存时间
27 # -v: verbose
28
29 OPTS="-rtv --delete-excluded --delete-after --partial"
30
31 # rsync include模式
32 # 开头的"/"会导致绝对路径名匹配.
33 INCLUDE=(
34     "/4/i386/kde-i18n-Chinese*"
35 # ^
36 # 双引号是必须的, 用来防止globbing.
37 )
38
39
40 # rsync exclude模式
41 # 使用#"临时注释掉一些不需要的包.
42 EXCLUDE=(
43     /1
44     /2
45     /3
46     /testing
47     /4/SRPMS
48     /4/ppc
49     /4/x86_64
50     /4/i386/debug
51     "/4/i386/kde-i18n-*"
52     "/4/i386/openoffice.org-langpack-*"
53     "/4/i386/*i586.rpm"
54     "/4/i386/GFS-*"
55     "/4/i386/cman-*"
56     "/4/i386/dlm-*"
57     "/4/i386/gnbd-*"
58     "/4/i386/kernel-smp*"
59 #     "/4/i386/kernel-xen*"
60 #     "/4/i386/xen-*"
61 )
62
63
64 init () {
65     # 让管道命令返回可能的rsync错误, 比如, 网络延时(stalled network).
66     set -o pipefail
67
68     TMP=${TMPDIR:-/tmp}/${0##*/}.$$      # 保存精炼的下载列表.
69     trap "{
70         rm -f $TMP 2>/dev/null
```

```

71     }" EXIT                                # 删除存在的临时文件.
72 }
73
74
75 check_pid () {
76 # 检查进程是否存在.
77     if [ -s "$PID_FILE" ]; then
78         echo "PID file exists. Checking ..."
79         PID=$(./bin/egrep -o "^[[:digit:]]+" $PID_FILE)
80         if /bin/ps --pid $PID &>/dev/null; then
81             echo "Process $PID found. ${0##*/} seems to be running!"
82             /usr/bin/logger -t ${0##*/} \
83                 "Process $PID found. ${0##*/} seems to be running!"
84             exit $E_RETURN
85         fi
86         echo "Process $PID not found. Start new process . . ."
87     fi
88 }
89
90
91 # 根据上边的模式,
92 #+ 设置整个文件的更新范围, 从root或$URL开始.
93 set_range () {
94     include=
95     exclude=
96     for p in "${INCLUDE[@]}"; do
97         include="$include --include \"$p\""
98     done
99
100    for p in "${EXCLUDE[@]}"; do
101        exclude="$exclude --exclude \"$p\""
102    done
103 }
104
105
106 # 获得并提炼rsync更新列表.
107 get_list () {
108     echo $$ > $PID_FILE || {
109         echo "Can't write to pid file $PID_FILE"
110         exit $E_RETURN
111     }
112
113     echo -n "Retrieving and refining update list . . ."
114
115     # 获得列表 -- 作为单个命令来运行rsync的话需要'eval'.

```

```

116 # $3和$4是文件创建的日期和时间.
117 # $5是完整的包名字.
118 previous=
119 pre_file=
120 pre_date=0
121 eval /bin/nice /usr/bin/rsync \
122 -r $include $exclude $URL | \
123 egrep '^dr.x|^-r' | \
124 awk '{print $3, $4, $5}' | \
125 sort -k3 | \
126 { while read line; do
127     # 获得这段运行的秒数, 过滤掉不用的包.
128     cur_date=$(date -d "$(echo $line | awk '{print $1, $2}')" +%s)
129     # echo $cur_date
130
131     # 取得文件名.
132     cur_file=$(echo $line | awk '{print $3}')
133     # echo $cur_file
134
135     # 如果可能的话, 从文件名中取得rpm的包名字.
136     if [[ $cur_file == *rpm ]]; then
137         pkg_name=$(echo $cur_file | sed -r -e \
138             's/(^([-_]+[-_])+)[:digit:]+.*[-_].*$/\1/')
139     else
140         pkg_name=
141     fi
142     # echo $pkg_name
143
144     if [ -z "$pkg_name" ]; then    # 如果不是一个rpm文件,
145         echo $cur_file >> $TMP    #+ 然后添加到下载列表里.
146     elif [ "$pkg_name" != "$previous" ]; then    # 发现一个新包.
147         echo $pre_file >> $TMP          # 输出最新的文件.
148         previous=$pkg_name            # 保存当前状态.
149         pre_date=$cur_date
150         pre_file=$cur_file
151     elif [ "$cur_date" -gt "$pre_date" ]; then    # 如果是相同的包,
152                                         # 但是这个包更新一些,
153         pre_date=$cur_date           #+ 那么就更新最新的.
154         pre_file=$cur_file
155     fi
156     done
157     echo $pre_file >> $TMP          # TMP现在包含所有
158                                         #+ 提炼过的列表.
159     # echo "subshell=$BASH_SUBSHELL"
160

```

```

161     }
162     # 这里的大括号是为了让最后这句"echo $pre_file >> $TMP"
163     # 也能与整个循环一起放到同一个子shell ( 1 )中.
164
165     RET=$?  # 取得管道命令的返回状态.
166
167     [ "$RET" -ne 0 ] && {
168         echo "List retrieving failed with code $RET"
169         exit $E_RETURN
170     }
171
172     echo "done"; echo
173 }
174
175 # 真正的rsync下载部分.
176 get_file () {
177
178     echo "Downloading..."
179     /bin/nice /usr/bin/rsync \
180         $OPTS \
181         --filter "merge,+/" $TMP" \
182         --exclude '*' \
183         $URL $DEST \
184         | /usr/bin/tee $LOG
185
186     RET=$?
187
188     # --filter merge,+/ 对于这个目的来说, 这句是至关重要的.
189     # + 修饰语意为着包含, / 意味着绝对路径.
190     # 然后$TMP中排过序的列表将会包含升序的路径名,
191     #+ 并从"简化的流程"(shortcutting the circuit)中阻止下边的 --exclude '*'.
192
193     echo "Done"
194
195     rm -f $PID_FILE 2>/dev/null
196
197     return $RET
198 }
199
200 # -----
201 # Main
202 init
203 check_pid
204 set_range
205 get_list
206 get_file

```

```

206 RET=$?
207 # -----
208
209 if [ "$RET" -eq 0 ]; then
210     /usr/bin/logger -t ${0##*/} "Fedora update mirrored successfully."
211 else
212     /usr/bin/logger -t ${0##*/} "Fedora update mirrored with failure code: $RET"
213 fi
214
215 exit $RET

```

在使用rcp, rsync, 还有另外一些有安全问题的类似工具的时候, 一定要小心, 因为将这些工具用在shell脚本中是不明智的. 你应该考虑使用ssh, scp, 或者expect脚本来代替这些不安全的工具.

ssh .

安全shell, 登陆远端主机并在其上运行命令. 这个工具具有身份认证和加密的功能, 可以安全的替换telnet, rlogin, rcp, 和rsh等工具. 请参考这个工具的man页来获取详细信息.

例子12-40. 使用ssh

```

1#!/bin/bash
2# remote.bash: 使用ssh.
3
4# 这个例子是Michael Zick编写的.
5# 授权在本书中使用.
6
7
8# 假设的一些前提:
9# -----
10# fd-2(文件描述符2)的内容并没有被丢弃('2>/dev/null').
11# ssh/sshd假设stderr('2')将会显示给用户.
12#
13# 假设sshd正运行在你的机器上.
14# 对于绝大多数, 标准, 的发行版, 都是有sshd的,
15#+ 并且没有稀奇古怪的ssh-keygen.
16
17# 在你的机器上从命令行中试着运行一下ssh:
18#
19# $ ssh $HOSTNAME
20# 不需要特别的设置, 也会要求你输入密码.
21# 接下来输入密码,
22# 完成后, $ exit
23#
24# 能够正常运行么? 如果正常的话, 接下来你可以获得更多的乐趣了.
25
26# 尝试在你的机器上以'root'身份来运行ssh:

```

```

27 #
28 # $ ssh -l root $HOSTNAME
29 # 当要求询问密码时，输入root的密码，注意别输入你的用户密码。
30 #           Last login: Tue Aug 10 20:25:49 2004 from localhost.localdomain
31 # 完成后键入'exit'.
32
33 # 上边的动作将会带给你一个交互的shell.
34 # 也可以在'single command'模式下建立sshd,
35 #+ 但是这已经超出本例所讲解的范围了.
36 # 唯一需要注意的是，下面的命令都可以运行在
37 #+ 'single command'模式下.
38
39
40 # 基本的，写stdout(本地)命令.
41
42 ls -l
43
44 # 这样远端机器上就会执行相同的命令.
45 # 如果你想的话，可以传递不同的'USERNAME'和'HOSTNAME':
46 USER=${USERNAME:-$(whoami)}
47 HOST=${HOSTNAME:-$(hostname)}
48
49 # 现在，在远端主机上执行上边的命令，
50 #+ 当然，所有的传输都会被加密.
51
52 ssh -l ${USER} ${HOST} " ls -l "
53
54 # 期望的结果就是在远端主机上列出
55 #+ 你的用户名所拥有的主目录下的所有文件.
56 # 如果想看点不一样的东西，
57 #+ 那就在别的地方运行这个脚本，别在你自己的主目录下运行这个脚本.
58
59 # 换句话说，Bash命令已经作为一个引用行
60 #+ 被传递到了远端shell中，这样远端机器就会运行它.
61 # 在这种情况下，sshd代表你运行了，bash -c "ls -l" .
62
63 # 如果你想不输入密码，
64 #+ 或者想更详细的了解相关的问题，请参考：
65 #+ man ssh
66 #+ man ssh-keygen
67 #+ man sshd_config.
68
69 exit 0

```

⑧ 在循环中，ssh可能会引起一些异常问题。根据comp.unix上的shell文档Usenet post所描述

的内容, ssh继承了循环的stdin. 为了解决这个问题, 请使用ssh的-n或者-f选项.

感谢, Jason Bechtel, 为我们指出这个问题.

scp .

安全拷贝, 在功能上与rcp很相似, 就是在两个不同的网络主机之间拷贝文件, 但是要使用鉴权的方式, 并且要使用与ssh类似的安全层.

本地网络

write .

这是一个端到端通讯的工具. 这个工具可以从你的终端上(console或者xterm)发送整行数据到另一个用户的终端上. mesg命令当然也可以用来禁用对于一个终端的写权限.

因为write命令是需要交互的, 所以这个命令在脚本中很少使用.

netconfig .

用来配置网络适配器(使用DHCP)的命令行工具. 这个命令对于红帽发行版来说是内置的.

邮件

mail .

发送或者读取e-mail消息.

如果把这个命令行的mail客户端当成一个脚本中的命令来使用的话, 效果非常好.

例子12-41. 一个mail自身的脚本

```
1 #!/bin/sh
2 # self-mailer.sh: mail自身的脚本.
3
4 adr=${1:-`whoami`}      # 如果没有指定的话, 默认是当前用户.
5 # 键入'self-mailer.sh wiseguy@superdupergenius.com'
6 #+ 将脚本发送到这个地址.
7 # 如果只键入'self-mailer.sh'(不给参数)的话,
8 #+ 那么这个脚本就会被发送给调用者, 比如, 比如, bozo@localhost.localdomain.
9 #
10 # 如果想了解${parameter:-default}结构的更多细节,
11 #+ 请参考"变量重游"那章中的
12 #+ "参数替换"小节.
13
14 # =====
15 cat $0 | mail -s "Script `basename $0` \
16 mailed itself to you." "$adr"
17 # =====
18
19 # -----
20 # 来自self-mailing脚本的一份祝福.
21 # 一个喜欢恶搞的家伙运行了这个脚本,
22 #+ 这导致了他自己收到了这份mail.
```

```
23 # 显然的，有些人确实没什么事好做，  
24 #+ 就只能浪费他们自己的时间玩了。  
25 # -----  
26  
27 echo "At `date`, script `basename $0` mailed to \"$adr\"."  
28  
29 exit 0
```

mailto .

与mail命令很相似, mailto可以使用命令行或在脚本中发送e-mail消息. 而且mailto也可以发送MIME(多媒体)消息.

vacation .

这个工具可以自动回复e-mail给发送者, 表示邮件的接受者正在度假暂时无法收到邮件. 这个工具与sendmail一起运行于网络上, 并且这个工具不支持拨号的POPmail帐号.

注意事项

1. 一个幽灵进程指的是并未附加在终端会话中的后台进程. 幽灵进程在指定的时间执行指定的服务, 或者由特定的事件触发来执行指定的服务.

希腊文中的”daemon”意思是幽灵, 这个词充满了神秘感和神奇的力量, 在UNIX中幽灵进程总是在后台默默地执行着分配给它们的任务.

7 终端控制命令

影响控制台或终端的命令

tput

初始化终端或者从terminfo数据中取得终端信息. 这个命令有许多选项, 每个选项都允许特定操作. tput clear与后边所介绍的clear命令等价, tput reset与后边所介绍的reset命令等价, tput sgr0可以复位终端, 但是并不清除屏幕.

```
1 bash$ tput longname
2 xterm terminal emulator (XFree86 4.0 Window System)
```

使用tput cup X Y将会把光标移动到当前终端的(X,Y)坐标上, 使用这个命令之前一般都要先用clear命令清屏.

注意: stty提供了一个更强大的命令专门用来设置如何控制终端.

infocmp

这个命令会打印出大量当前终端的信息. 事实上它是引用了terminfo数据库的内容.

```
1 bash$ infocmp
2 #      通过infocmp显示出来, 内容都来自于文件:
3 /usr/share/terminfo/r/rxvt
4 rxvt|rxvt terminal emulator (X Window System),
5     am, bce, eo, km, mir, msgr, xenl, xon,
6     colors#8, cols#80, it#8, lines#24, pairs#64,
7     acsc='aaffggjjkkllmmnnooppqrrssttuuvwxyz{{||}}~~',
8     bel=\E[5m, blink=\E[1m,
9     civis=\E[?25l,
10    clear=\E[H\E[2J, cnorm=\E[?25h, cr=\^M,
11    ...
```

reset

复位终端参数并且清除屏幕. 与clear命令一样, 光标和提示符将会重新出现在终端的左上角.

clear

clear命令只不过是简单的清除控制台或者xterm的屏幕. 光标和提示符将会重新出现在屏幕或者xterm window的左上角. 这个命令既可以用在命令行中也可以用在脚本中. 请参考[例子10-25](#).

script

这个工具将会记录(保存到一个文件中)所有的用户按键信息(在控制台下的或在xterm window下的按键信息). 这其实就是创建了一个会话记录.

8 数学计算命令

”操作数字”

factor

将一个正数分解为多个素数.

```
1 bash$ factor 27417
2 27417: 3 13 19 37
```

bc

Bash不能处理浮点运算，并且缺乏特定的一些操作，这些操作都是一些重要的计算功能。幸运的是，bc可以解决这个问题。

bc不仅仅是个多功能灵活的精确计算工具，且它还提供许多编程语言才具备的一些方便功能。

bc比较类似于C语言的语法。

因为它是一个完整的UNIX工具，所以它可以用在[pipe](#)中，bc在脚本中也是很常用的。

这里有一个简单的使用bc命令的模版，可以用来计算脚本中的变量。这个模版经常用于[命令替换](#)中。

```
1 variable=$(echo "OPTIONS; OPERATIONS" | bc)
```

例子12-42. 按月偿还贷款

```
1#!/bin/bash
2# monthlypmt.sh: 计算按月偿还贷款的数量。
3
4
5# 这份代码是一份修改版本，原始版本在"mcalc"(贷款计算)包中，
6#+ 这个包的作者是Jeff Schmidt和Mendel Cooper(本书作者)。
7#   http://www.ibiblio.org/pub/Linux/apps/financial/mcalc-1.6.tar.gz [15k]
8
9echo
10echo "Given the principal, interest rate, and term of a mortgage,"
11echo "calculate the monthly payment."
12
13bottom=1.0
14
15echo
16echo -n "Enter principal (no commas) "
17read principal
18echo -n "Enter interest rate (percent)"#如果是12%，那就键入"12"，而不是".12"。
19read interest_r
20echo -n "Enter term (months) "
21read term
```

```

23
24 interest_r=$(echo "scale=9; $interest_r/100.0" | bc) # 转换成小数.
25         # "scale"指定了有效数字的个数.
26
27
28 interest_rate=$(echo "scale=9; $interest_r/12 + 1.0" | bc)
29
30
31 top=$(echo "scale=9; $principal*$interest_rate^$term" | bc)
32
33 echo; echo "Please be patient. This may take a while."
34
35 let "months = $term - 1"
36 # =====
37 for ((x=$months; x > 0; x--))
38 do
39     bot=$(echo "scale=9; $interest_rate^$x" | bc)
40     bottom=$(echo "scale=9; $bottom+$bot" | bc)
41 # bottom = $($bottom + $bot")
42 done
43 # =====
44
45 # -----
46 # Rick Boivie给出了一个对上边循环的修改方案,
47 #+ 这个修改更加有效率, 将会节省大概2/3的时间.
48
49 # for ((x=1; x <= $months; x++))
50 # do
51 #     bottom=$(echo "scale=9; $bottom * $interest_rate + 1" | bc)
52 # done
53
54
55 # 然后他又想出了一个更加有效率的版本,
56 #+ 将会节省95%的时间!
57
58 # bottom={
59 #     echo "scale=9; bottom=$bottom; interest_rate=$interest_rate"
60 #     for ((x=1; x <= $months; x++))
61 #     do
62 #         echo 'bottom = bottom * interest_rate + 1'
63 #     done
64 #     echo 'bottom'
65 # } | bc'      # 在命令替换中嵌入一个'for循环'.
66 #
67 # -----
# 另一方面, Frank Wang建议:

```

```

68 # bottom=$(echo "scale=9; \
69 nterest_rate^$term-1)/($interest_rate-1)" | bc)
70
71 # 因为 . .
72 # 在循环后边的算法
73 #+ 事实上是一个等比数列的求和公式.
74 # 求和公式是  $e_0(1-q^n)/(1-q)$ ,
75 #+  $e_0$  是第一个元素,  $q=e(n+1)/e(n)$ ,
76 #+  $n$  是元素数量.
77 # -----
78
79
80 # let "payment = $top/$bottom"
81 payment=$(echo "scale=2; $top/$bottom" | bc)
82 # 使用2位有效数字来表示美元和美分.
83
84 echo
85 echo "monthly payment = \$\$payment" # 在总和的前边显示美元符号.
86 echo
87
88
89 exit 0
90
91
92 # 练习:
93 # 1) 处理输入允许本金总数中的逗号.
94 # 2) 处理输入允许按照百分号和小数点的形式输入利率.
95 # 3) 如果你真正想好好编写这个脚本,
96 # 那么就扩展这个脚本让它能够打印出完整的分期付款表.

```

例子12-43. 数制转换

```

1#!/bin/bash
2#####
3# 脚本      : base.sh - 用不同的数制来打印数字 (Bourne Shell)
4# 作者      : Heiner Steven (heiner.steven@odn.de)
5# 日期      : 07-03-95
6# 类型      : 桌面
7# $Id: base.sh,v 1.2 2000/02/06 19:55:35 heiner Exp $
8# ==> 上边这行是RCS ID信息.
9#####
10# 描述
11#
12# 修改纪录
13# 21-03-95 stv fixed error occurring with 0xb as input (0.2)
14#####

```

```

15
16 # ==> 在本书中使用这个脚本通过了原作者的授权.
17 # ==> 注释是本书作者添加的.
18
19 NOARGS=65
20 PN='basename "$0"' # 程序名
21 VER='echo '$Revision: 1.2 $' | cut -d' ' -f2' # ==> VER=1.2
22
23 Usage () {
24     echo "$PN - print number to different bases, $VER (stv '95)
25 usage: $PN [number ...]
26
27 If no number is given, the numbers are read from standard input.
28 A number may be
29     binary (base 2)           starting with 0b (i.e. 0b1100)
30     octal (base 8)            starting with 0 (i.e. 014)
31     hexadecimal (base 16)      starting with 0x (i.e. 0xc)
32     decimal                  otherwise (i.e. 12)" >&2
33     exit $NOARGS
34 } # ==> 打印出用法信息的函数.
35
36 Msg () {
37     for i    # ==> 省略[list].
38     do echo "$PN: $i" >&2
39     done
40 }
41
42 Fatal () { Msg "$@"; exit 66; }
43
44 PrintBases () {
45     # 决定数字的数制
46     for i      # ==> 省略[list]...
47     do        # ==> 所以是对命令行参数进行操作.
48         case "$i" in
49             0b*)
50                 ibase=2;;
51                 # 2进制
52             0x*[a-f]*|[A-F]*)
53                 ibase=16;;
54                 # 16进制
55             0*)
56                 ibase=8;;
57                 # 8进制
58             [1-9]*)
59                 ibase=10;;
56                 # 10进制
57             *)
58                 Msg "illegal number $i - ignored"
59                 continue;;
60             esac
61
62     # 去掉前缀, 将16进制数字转换为大写(bc命令需要这么做)
63     number='echo "$i" | sed -e 's:^0[bBxX]::' | tr '[a-f]' '[A-F]''

```

```

60      # ==> 使用 ":" 作为 sed分隔符，而不使用 "/" .
61
62      # 将数字转换为10进制
63      dec='echo "ibase=$ibase; $number" | bc' # ==> 'bc'是个计算工具 .
64      case "$dec" in
65          [0-9]*);;                                # 数字没问题
66          *)           continue;;                # 错误：忽略
67      esac
68
69      # 在一行上打印所有转换后的数字 .
70      # ==> 'here document' 提供命令列表给 'bc' .
71      echo 'bc <<!
72          obase=16; "hex="; $dec
73          obase=10; "dec="; $dec
74          obase=8;  "oct="; $dec
75          obase=2;  "bin="; $dec
76      !
77      ' | sed -e 's: : :g'
78
79      done
80 }
81
82 while [ $# -gt 0 ]
83 # ==> 这里必须使用一个"while循环",
84 # ==>+ 因为所有的case都可能退出循环或者
85 # ==>+ 结束脚本 .
86 # ==> (感谢, Paulo Marcel Coelho Aragao.)
87 do
88     case "$1" in
89         --)    shift; break;;
90         -h)    Usage;;          # ==> 帮助信息 .
91         -*)   Usage;;
92         *)    break;;          # 第一个数字
93     esac # ==> 对于非法输入进行更严格检查是非常有用的 .
94     shift
95 done
96
97 if [ $# -gt 0 ]
98 then
99     PrintBases "$@"
100 else                                # 从stdin中读取
101     while read line
102     do
103         PrintBases $line
104     done

```

```
105 fi  
106  
107  
108 exit 0
```

调用bc的另一种方法就是[here document](#), 并把它嵌入到[命令替换](#)块中. 当一个脚本需要将一个选项列表和多个命令传递到bc中时, 这种方法就显得非常合适了.

```
1 variable='bc << LIMIT_STRING  
2 options  
3 statements  
4 operations  
5 LIMIT_STRING  
6 '  
7  
8 ...or...  
9  
10  
11 variable=$(bc << LIMIT_STRING  
12 options  
13 statements  
14 operations  
15 LIMIT_STRING  
16 )
```

例子12-44. 使用”here document”来调用bc

```
1#!/bin/bash  
2# 使用命令替换来调用'bc'  
3# 并与'here document'相结合.  
4  
5  
6var1='bc << EOF  
718.33 * 19.78  
8EOF  
9'  
10echo $var1      # 362.56  
11  
12  
13# 使用$(...)这种标记法也可以.  
14v1=23.53  
15v2=17.881  
16v3=83.501  
17v4=171.63  
18  
19var2=$(bc << EOF  
20scale = 4  
21a = ( $v1 + $v2 )
```

```

22 b = ( $v3 * $v4 )
23 a * b + 15.35
24 EOF
25 )
26 echo $var2      # 593487.8452
27
28
29 var3=$(bc -l << EOF
30 scale = 9
31 s ( 1.7 )
32 EOF
33 )
34 # 返回弧度为1.7的正弦.
35 # "-l"选项将会调用'bc'算数库.
36 echo $var3      # .991664810
37
38
39 # 现在, 在函数中试一下...
40 hyp=          # 声明全局变量.
41 hypotenuse () # 计算直角三角形的斜边.
42 {
43     hyp=$(bc -l << EOF
44     scale = 9
45     sqrt ( $1 * $1 + $2 * $2 )
46     EOF
47 )
48 # 不幸的是, 不能从bash函数中返回浮点值.
49 }
50
51 hypotenuse 3.68 7.31
52 echo "hypotenuse = $hyp"    # 8.184039344
53
54
55 exit 0

```

例子12-45. 计算圆周率

```

1#!/bin/bash
2# cannon.sh: Approximating PI by firing cannonballs.
3
4# 这事实上是一个"Monte Carlo"蒙特卡洛模拟的非常简单的实例:
5#+ 蒙特卡洛模拟是一种由现实事件抽象出来的数学模型,
6#+ 由于要使用随机抽样统计来估算数学函数, 所以使用伪随机数来模拟真正的随机数.
7
8# 想象有一个完美的正方形土地, 边长为10000个单位.
9# 在这块土地的中间有一个完美的圆形湖,

```

```

10  #+ 这个湖的直径是10000个单位.
11 # 这块土地的绝大多数面积都是水, 当然只有4个角上有一些土地.
12 # (可以把这个湖想象成为这个正方形的内接圆.)
13 #
14 # 我们将使用老式的大炮和铁炮弹
15 #+ 向这块正方形的土地上开炮.
16 # 所有的炮弹都会击中这块正方形土地的某个地方.
17 #+ 或者是打到湖上, 或者是打到4个角的土地上.
18 # 因为这个湖占据了这个区域大部分地方,
19 #+ 所以大部分的炮弹都会"扑通"一声落到水里.
20 # 而只有很少的炮弹会"砰"的一声落到4个
21 #+ 角的土地上.
22 #
23 # 如果我们发出的炮弹足够随机的落到这块正方形区域中的话,
24 #+ 那么落到水里的炮弹与打出炮弹总数的比率,
25 #+ 大概非常接近于PI/4.
26 #
27 # 原因是所有的炮弹事实上都
28 #+ 打在了这个土地的右上角,
29 #+ 也就是, 笛卡尔坐标系的第一象限.
30 # (之前的解释只是一个简化.)
31 #
32 # 理论上来说, 如果打出的炮弹越多, 就越接近这个数字.
33 # 然而, 对于shell 脚本来说一定会做些让步的,
34 #+ 因为它肯定不能和那些内建就支持浮点运算的编译语言相比.
35 # 当然就会降低精度.

36
37
38 DIMENSION=10000 # 这块土地的边长.
39 # 这也是所产生随机整数的上限.

40
41 MAXSHOTS=1000 # 开炮次数.
42 # 10000或更多次的话, 效果应该更好, 但有点太浪费时间了.
43 PMULTIPLIER=4.0 # 接近于PI的比例因子.

44
45 get_random ()
46 {
47 SEED=$(head -1 /dev/urandom | od -N 1 | awk '{ print $2 }')
48 RANDOM=$SEED # 来自于"seeding-random.sh"
49 #+ 的例子脚本.
50 let "rnum = $RANDOM % $DIMENSION" # 范围小于10000.
51 echo $rnum
52 }
53
54 distance= # 声明全局变量.

```

```

55 hypotenuse ()      # 从"alt-bc.sh"例子来的,
56 {                  # 计算直角三角形的斜边的函数.
57     distance=$(bc -l << EOF
58     scale = 0
59     sqrt ( $1 * $1 + $2 * $2 )
60     EOF
61   )
62   # 设置 "scale" 为 0 , 好让结果四舍五入为整数值,
63   #+ 这也是这个脚本中必须折中的一个地方.
64   # 不幸的是, 这将降低模拟的精度.
65 }
66
67
68 # main() {
69
70 # 初始化变量.
71 shots=0
72 splashes=0
73 thuds=0
74 Pi=0
75
76 while [ "$shots" -lt  "$MAXSHOTS" ]          # 主循环.
77 do
78
79     xCoord=$(get_random)                      # 取得随机的 X 与 Y 坐标.
80     yCoord=$(get_random)
81     hypotenuse $xCoord $yCoord                # 直角三角形斜边 =
82                               #+ distance.
83     ((shots++))
84
85     printf "#%4d    " $shots
86     printf "Xc = %4d   " $xCoord
87     printf "Yc = %4d   " $yCoord
88     printf "Distance = %5d   " $distance        # 到湖中心的
89                               #+ 距离 --
90                               # 起始坐标点 --
91                               #+ (0,0).
92
93     if [ "$distance" -le "$DIMENSION" ]
94     then
95         echo -n "SPLASH!  "
96         ((splashes++))
97     else
98         echo -n "THUD!      "
99         ((thuds++))

```

```

100 fi
101
102 Pi=$(echo "scale=9; $PMULTIPLIER*$splashes/$shots" | bc)
103 # 将比例乘以4.0.
104 echo -n "PI ~ $Pi"
105 echo
106
107 done
108
109 echo
110 echo "After $shots shots, PI looks like approximately $Pi."
111 # 如果不太准的话, 那么就提高一下运行的次数. . .
112 # 可能是由于运行错误和随机数随机程度不高造成的.
113 echo
114
115 #
116
117 exit 0
118
119 # 要想知道一个shell脚本到底适不适合对计算应用进行模拟的话?
120 #+ (一种需要对复杂度和精度都有要求的计算应用).
121 #
122 # 一般至少需要两个判断条件.
123 # 1) 作为一种概念的验证: 来显示它可以做到.
124 # 2) 在使用真正的编译语言来实现一个算法之前,
125 #+ 使用脚本来测试和验证这个算法.

```

dc

dc(桌面计算器desk calculator)工具是面向栈的, 并且使用RPN(逆波兰表达式”Reverse Polish Notation”又叫”后缀表达式”). 与bc命令很相似, 但是这个工具具备好多只有编程语言才具备的能力.

```

1 (
2 译者注: 正常表达式      逆波兰表达式
3      a+b                  a,b,+ 
4      a+(b-c)              a,b,c,-,+ 
5      a+(b-c)*d            a,d,b,c,-,*,+ 
6 )
7

```

绝大多数人都避免使用这个工具, 因为它需要非直观的RPN输入, 但是, 它却有特定的用途.

例子12-46. 将10进制数字转换为16进制数字

```

1#!/bin/bash
2# hexconvert.sh: 将10进制数字转换为16进制数字.
3
4E_NOARGS=65 # 缺少命令行参数错误.

```

```

5 BASE=16      # 16进制.
6
7 if [ -z "$1" ]
8 then
9     echo "Usage: $0 number"
10    exit $E_NOARGS
11    # 需要一个命令行参数.
12 fi
13 # 练习：添加命令行参数检查.
14
15
16 hexcvt ()
17 {
18 if [ -z "$1" ]
19 then
20     echo 0
21     return    # 如果没有参数传递到这个函数中的话就"return" 0.
22 fi
23
24 echo ""$1" "$BASE" o p" | dc
25 #           "o" 设置输出的基数(数制).
26 #           "p" 打印栈顶.
27 # 参考dc的man页来了解其他的选项.
28 return
29 }
30
31 hexcvt "$1"
32
33 exit 0

```

通过仔细学习dc的info页，可以更深入的理解这个复杂的命令。但是，有一些精通dc巫术小组经常会炫耀他们使用这个强大而又晦涩难懂的工具时的一些技巧，并以此为乐。

```

1 bash$ echo "16i[q]sa[ln0=aln100%Pln100/sn1bx]sbA0D68736142sn1bxq" | dc"
2 Bash

```

例子12-47. 因子分解

```

1#!/bin/bash
2# factr.sh: 分解约数
3
4 MIN=2      # 如果比这个数小就不行了.
5 E_NOARGS=65
6 E_TOOSMALL=66
7
8 if [ -z $1 ]
9 then
10    echo "Usage: $0 number"

```

```

11    exit $E_NOARGS
12 fi
13
14 if [ "$1" -lt "$MIN" ]
15 then
16     echo "Number to factor must be $MIN or greater."
17     exit $E_TOOSMALL
18 fi
19
20 # 练习：添加类型检查(防止非整型的参数).
21
22 echo "Factors of $1:"
23 # -----
24 echo "$1[p]s2[lip/dli%0=1dvsr]s12sid2%0=\`"
25 sidvsr[dli%0=1lrlid+dsi!>.]ds.xd1<2" | dc
26 # -----
27 # 上边这行代码是Michel Charpentier编写的<charpov@cs.unh.edu>.
28 # 在此使用经过授权(感谢).
29
30 exit 0

```

awk

在脚本中使用浮点运算的另一种方法是使用awk内建的数学运算函数，可以用在shell包装中。
例子12-48. 计算直角三角形的斜边

```

1#!/bin/bash
2# hypotenuse.sh: 返回直角三角形的斜边.
3#           (直角边长的平方和, 然后对和取平方根)
4
5ARGS=2          # 需要将2个直角边作为参数传递进来.
6E_BADARGS=65    # 错误的参数值.
7
8if [ $# -ne "$ARGS" ] # 测试传递到脚本中的参数值.
9then
10   echo "Usage: `basename $0` side_1 side_2"
11   exit $E_BADARGS
12fi
13
14
15AWKSCRIPT=' { printf( "%3.7f\n", sqrt($1*$1 + $2*$2) ) } '
16#           命令 / 传递给awk的参数
17
18
19# 现在, 将参数通过管道传递给awk.
20echo -n "Hypotenuse of $1 and $2 = "

```

```
21 echo $1 $2 | awk "$AWKSCRIPT"
22
23 exit 0
```

9 混杂命令

一些不好归类的命令

jot, seq .

这些工具用来生成一系列整数, 用户可以指定生成范围.

每个产生出来的整数一般都占一行, 但是可以使用-s选项来改变这种设置.

```
1 bash$ seq 5
2 1
3 2
4 3
5 4
6 5
7
8
9
10 bash$ seq -s : 5
11 1:2:3:4:5
```

jot和seq命令经常用在for循环中.

例子12-49. 使用seq命令来产生循环参数

```
1#!/bin/bash
2# 使用"seq"
3
4echo
5
6for a in `seq 80` # or for a in $( seq 80 )
7# 与for a in 1 2 3 4 5 ... 80 相同(少敲了好多字!).
8# 也可以使用'jot'(如果系统上有的话).
9do
10 echo -n "$a "
11done      # 1 2 3 4 5 ... 80
12# 这也是一个通过使用命令输出
13# 来产生"for"循环中[list]列表的例子.
14
15echo; echo
16
17
18COUNT=80 # 当然, 'seq'也可以使用一个可替换的参数.
19
20for a in `seq $COUNT` # 或者 for a in $( seq $COUNT )
21do
22 echo -n "$a "
23done      # 1 2 3 4 5 ... 80
```

```

24
25 echo; echo
26
27 BEGIN=75
28 END=80
29
30 for a in `seq $BEGIN $END`
31 # 传给"seq"两个参数, 从第一个参数开始增长,
32 #+ 一直增长到第二个参数为止.
33 do
34   echo -n "$a "
35 done      # 75 76 77 78 79 80
36
37 echo; echo
38
39 BEGIN=45
40 INTERVAL=5
41 END=80
42
43 for a in `seq $BEGIN $INTERVAL $END`
44 # 传给"seq"三个参数, 从第一个参数开始增长,
45 #+ 并以第二个参数作为增量,
46 #+ 一直增长到第三个参数为止.
47 do
48   echo -n "$a "
49 done      # 45 50 55 60 65 70 75 80
50
51 echo; echo
52
53 exit 0

```

一个简单一些的例子:

```

1 # 产生10个连续扩展名的文件,
2 #+ 名字分别是 file.1, file.2 . . . file.10.
3 COUNT=10
4 PREFIX=file
5
6 for filename in `seq $COUNT`
7 do
8   touch $PREFIX.$filename
9 # 或者, 你可以做一些其他的操作,
10 #+ 比如rm, grep, 等等.
11 done

```

例子12-50. 字母统计

```

1 #!/bin/bash

```

```
2 # letter-count.sh: 统计一个文本文件中某些字母出现的次数.
3 # 由Stefano Palmeri所编写.
4 # 经过授权可以使用在本书中.
5 # 本书作者做了少许修改.
6
7 MINARGS=2          # 本脚本至少需要2个参数.
8 E_BADARGS=65
9 FILE=$1
10
11 let LETTERS=$#-1    # 指定了多少个字母(作为命令行参数).
12                 # (从命令行参数的个数中减1.)
13
14
15 show_help(){
16     echo
17     echo Usage: `basename $0` file letters
18     echo Note: `basename $0` arguments are case sensitive.
19     echo Example: `basename $0` foobar.txt G n U L i N U x.
20     echo
21 }
22
23 # 检查参数个数.
24 if [ $# -lt $MINARGS ]; then
25     echo
26     echo "Not enough arguments."
27     echo
28     show_help
29     exit $E_BADARGS
30 fi
31
32
33 # 检查文件是否存在.
34 if [ ! -f $FILE ]; then
35     echo "File \"\$FILE\" does not exist."
36     exit $E_BADARGS
37 fi
38
39
40
41 # 统计字母出现的次数.
42 for n in `seq $LETTERS`; do
43     shift
44     if [[ `echo -n "$1" | wc -c` -eq 1 ]]; then           # 检查参数.
45         echo "$1" -> `cat $FILE | tr -cd "$1" | wc -c` # 统计.
46     else
```

```

47         echo "$1 is not a single char."
48     fi
49 done
50
51 exit $?
52
53 # 这个脚本在功能上与letter-count2.sh完全相同,
54 #+ 但是运行得更快.
55 # 为什么?

```

getopt .

getopt命令将会分析以破折号开头的命令行选项. 这个外部命令与Bash的内建命令getopts作用相同. 通过使用-l标志, getopt可以处理超长(多个字符的)选项, 并且也允许参数重置.

例子12-51. 使用getopt来分析命令行选项

```

1#!/bin/bash
2# 使用getopt.
3
4# 尝试使用下边的不同的方法来调用这脚本:
5# sh ex33a.sh -a
6# sh ex33a.sh -abc
7# sh ex33a.sh -a -b -c
8# sh ex33a.sh -d
9# sh ex33a.sh -dXYZ
10# sh ex33a.sh -d XYZ
11# sh ex33a.sh -abcd
12# sh ex33a.sh -abcdZ
13# sh ex33a.sh -z
14# sh ex33a.sh a
15# 解释上面每一次调用的结果.
16
17E_OPTERR=65
18
19if [ "$#" -eq 0 ]
20then  # 脚本需要至少一个命令行参数.
21    echo "Usage $0 -[options a,b,c]"
22    exit $E_OPTERR
23fi
24
25set -- `getopt "abcd:" "$@"` 
26# 为命令行参数设置位置参数.
27# 如果使用"$*"来代替"$@"的话, 会发生什么?
28
29while [ ! -z "$1" ]
30do

```

```

31 case "$1" in
32     -a) echo "Option \"a\"";;
33     -b) echo "Option \"b\"";;
34     -c) echo "Option \"c\"";;
35     -d) echo "Option \"d\" $2";;
36     *) break;;
37 esac
38
39 shift
40 done
41
42 # 通常来说在脚本中使用内建的'getopts'命令,
43 #+ 会比使用'getopt'好一些.
44 # 参考"ex33.sh".
45
46 exit 0

```

请参考[例子9-13](#), 这是对getopt命令的一个简单模拟.

run-parts .

run-parts命令[1] 将会执行目标目录中所有的脚本, 这些脚本会以ASCII码的循序进行排列. 当然, 这些脚本都需要具有可执行权限.

cron 幽灵进程会调用run-parts来运行/etc/cron.*下的所有脚本.

yes .

yes命令的默认行为是向stdout连续不断的输出字符y, 每个y单独占一行. 可以使用control-c来结束输出. 如果想换一个输出字符的话, 可以使用yes different string, 这样就会连续不断的输出different string到stdout. 那么这样的命令究竟能用来做什么呢? 在命令行或者脚本中, yes的输出可以通过重定向或管道来传递给一些命令, 这些命令的特点是需要用户输入来进行交互. 事实上, 这个命令可以说是expect命令(译者注: 这个命令本书未介绍, 一个自动实现交互的命令)的一个简化版本.

yes — fsck /dev/hda1将会以非交互的形式运行fsck(译者注: 因为需要用户输入的y全由yes命令搞定了)(小心使用!).

yes — rm -r dirname 与rm -rf dirname 效果相同(小心使用!).

⑧: 当用yes命令的管道形式来使用一些可能具有潜在危险的系统命令的时候一定要深思熟虑, 比如fsck或fdisk. 可能会产生一些令人意外的副作用.

►: yes命令也可用来分析变量. 比如:

```

1 bash$ yes $BASH_VERSION
2 3.00.16(1)-release
3 3.00.16(1)-release
4 3.00.16(1)-release
5 3.00.16(1)-release
6 3.00.16(1)-release
7 . .

```

这个”特性”估计也不会特别有用.

banner .

将会把传递进来的参数字符串用一个ASCII字符(默认是'#')给画出来(就是将多个'#'拼出一副字符的图形), 然后输出到stdout. 可以作为硬拷贝重定向到打印机上. (译者注: 可以使用-w选项设置宽度.)

printenv .

显示某个特定用户所有的环境变量.

```
1 bash$ printenv | grep HOME  
2 HOME=/home/bozo
```

lp .

lp和lpr命令将会把文件发送到打印队列中, 并且作为硬拷贝来打印. [2] 这些命令会记录它们名字的起点, 直到行打印机的另一个阶段.

bash\$ lp file1.txt 或者bash\$ lp >file1.txt

通常情况下都是将pr的格式化输出传递到lp中.

bash\$ pr -options file1.txt — lp

格式化的包, 比如groff和Ghostscript就可以将它们的输出直接发送给lp.

bash\$ groff -Tascii file.tr — lp

bash\$ gs -options — lp file.ps

还有一些相关的命令, 比如lpq, 可以用来查看打印队列, 而lprm, 可以从打印队列中删除作业.

tee .

[这是UNIX从管道行业借来的主意.]

这是一个重定向操作, 但是与之前所看到的有点不同. 就像管道中的"三通"一样, 这个命令可以将命令或者管道命令的输出"抽出"到一个文件中, 而且不影响结果. 当你想将一个运行中进程的输出保存到文件时, 或者为了debug而保存输出记录的时候, 这个命令就显得非常有用.

```
1 (重定向)  
2 |----> 到文件  
3 |  
4 ======|=====  
5 命令 ---> 命令 ---> | tee ---> 命令 ---> ---> 管道的输出  
6 ======
```

```
1 cat listfile* | sort | tee check.file | uniq > result.file
```

(在对排序的结果进行uniq(去掉重复行)之前, 文件check.file保存了排过序的"listfiles".)

mkfifo .

这个不大引人注意的命令可以创建一个命名管道, 并产生一个临时的先进先出的buffer, 用来在两个进程之间传递数据. [3] 典型的应用是一个进程向FIFO中写数据, 另一个进程读出来. 请参考[例子A-15](#).

pathchk .

这个命令用来检查文件名的有效性. 如果文件名超过了最大允许长度(255个字符), 或者它所在的一个或多个路径搜索不到, 那么就会产生一个错误结果.

不幸的是, pathchk并不能够返回一个可识别的错误码, 因此它在脚本中几乎没有什么用. 可以考虑使用文件测试操作来替代这个命令.

dd .

这也是一个不太出名的工具, 但却是一个令人恐惧的”数据复制”命令. 最开始, 这个命令被用来在UNIX微机和IBM大型机之间通过磁带来交换数据, 这个命令现在仍然有它的用途. dd命令只不过是简单的拷贝一个文件(或者stdin/stdout), 但是它会做一些转换. 下边是一些可能的转换, 比如ASCII/EBCDIC,[4] 大写/小写, 在输入和输出之间的字节对的交换, 还有对输入文件做一些截头去尾的工作. dd -help列出了所有转换, 还列出了这个强大工具的其他一些选项.

```
1 # 将一个文件转换为大写:  
2  
3 dd if=$filename conv=ucase > $filename.uppercase  
4 #                      lcase   # 转换为小写
```

例子12-52. 一个拷贝自身的脚本

```
1#!/bin/bash  
2# self-copy.sh  
3  
4# 这个脚本会拷贝自身.  
5  
6file_subscript=copy  
7  
8dd if=$0 of=$0.$file_subscript 2>/dev/null  
9# 阻止dd产生的消息:           ~~~~~  
10  
11exit $?
```

例子12-53. 练习dd

```
1#!/bin/bash  
2# exercising-dd.sh  
3  
4# 由Stephane Chazelas编写.  
5# 本文作者做了少量修改.  
6  
7input_file=$0    # 脚本自身.  
8output_file=log.txt  
9n=3  
10p=5  
11  
12dd if=$input_file of=$output_file bs=1 \  
13p=$((n-1)) count=$((p-n+1)) 2> /dev/null  
14# 从脚本中把位置n到p的字符提取出来.  
15
```

```

16 # -----
17
18 echo -n "hello world" | dd cbs=1 conv=unblock 2> /dev/null
19 # 垂直地echo "hello world".
20
21 exit 0

```

为了展示dd的多种用途, 让我们使用它来记录按键.

例子12-54. 记录按键

```

1#!/bin/bash
2# dd-keypress.sh: 记录按键, 不需要按回车.
3
4
5 keypresses=4          # 记录按键的个数.
6
7
8 old_tty_setting=$(stty -g)      # 保存旧的终端设置.
9
10 echo "Press $keypresses keys."
11 stty -icanon -echo           # 禁用标准模式.
12                         # 禁用本地echo.
13 keys=$(dd bs=1 count=$keypresses 2> /dev/null)
14 # 如果不指定输入文件的话, 'dd'使用标准输入.
15
16 stty "$old_tty_setting"       # 恢复旧的终端设置.
17
18 echo "You pressed the \"$keys\" keys."
19
20 # 感谢Stephane Chazelas, 演示了这种方法.
21 exit 0

```

dd命令可以在数据流上做随机访问.

```

1 echo -n . | dd bs=1 seek=4 of=file conv=notrunc
2 # "conv=notrunc"选项意味着输出文件不能被截短.
3
4 # 感谢, S.C.

```

dd命令可以将数据或磁盘镜像拷贝到设备中, 也可以从设备中拷贝数据或磁盘镜像, 比如说磁盘或磁带设备都可以([例子A-5](#)). 通常用来创建启动磁盘.

dd if=kernel-image of=/dev/fd0H1440

同样的, dd可以拷贝软盘的整个内容(甚至是“其他”操作系统的磁盘格式), 到硬盘驱动器上(以镜像文件的形式).

dd if=/dev/fd0 of=/home/bozo/projects/floppy.img

dd命令还有一些其他用途, 包括可以初始化临时交换文件([例子28-2](#))和ramdisks(内存虚拟硬盘)([例子28-3](#)). 它甚至可以做一些对整个硬盘分区的底层拷贝, 虽然不建议这么做.

某些(可能是比较无聊的)人总会想一些关于dd命令的有趣应用.

例子12-55. 安全的删除一个文件

```
1 #!/bin/bash
2 # blot-out.sh: 删除一个文件"所有"的记录.
3
4 # 这个脚本会使用随机字节交替的覆盖目标文件,
5 #+ 并且在最终删除这个文件之前清零.
6 # 这么做之后, 即使你通过传统手段来检查磁盘扇区
7 #+ 也不能把文件原始数据重新恢复.
8
9 PASSES=7          # 破坏文件的次数.
10                  # 提高这个数字会减慢脚本运行的速度,
11                  #+ 尤其是对尺寸比较大的目标文件进行操作的时候.
12 BLOCKSIZE=1       # 带有/dev/urandom的I/O需要单位块尺寸,
13                  #+ 否则你可能会获得奇怪的结果.
14 E_BADARGS=70      # 不同的错误退出码.
15 E_NOT_FOUND=71
16 E_CHANGED_MIND=72
17
18 if [ -z "$1" ]    # 没指定文件名.
19 then
20   echo "Usage: `basename $0` filename"
21   exit $E_BADARGS
22 fi
23
24 file=$1
25
26 if [ ! -e "$file" ]
27 then
28   echo "File \"\$file\" not found."
29   exit $E_NOT_FOUND
30 fi
31
32 echo; echo -n "Are you absolutely sure
33 want to blot out \"\$file\" (y/n)? "
34 read answer
35 case "$answer" in
36 [nN]) echo "Changed your mind, huh?"
37   exit $E_CHANGED_MIND
38   ;;
39 *)   echo "Blotting out file \"\$file\".";;
40 esac
41
42
43 flength=$(ls -l "$file" | awk '{print $5}') # 5是文件长度.
```

```
44 pass_count=1
45
46 chmod u+w "$file"    # 允许覆盖/删除这个文件.
47
48 echo
49
50 while [ "$pass_count" -le "$PASSES" ]
51 do
52     echo "Pass #$pass_count"
53     sync      # 刷新buffers.
54     dd if=/dev/urandom of=$file bs=$BLOCKSIZE count=$flength
55             # 使用随机字节进行填充.
56     sync      # 再次刷新buffer.
57     dd if=/dev/zero of=$file bs=$BLOCKSIZE count=$flength
58             # 用0填充.
59     sync      # 再次刷新buffer.
60     let "pass_count += 1"
61     echo
62 done
63
64
65 rm -f $file    # 最后, 删除这个已经被破坏得不成样子的文件.
66 sync      # 最后一次刷新buffer.
67
68 echo "File \"$file\" blotted out and deleted."; echo
69
70
71 exit 0
72
73 # 这是一种真正安全的删除文件的办法,
74 #+ 但是效率比较低, 运行比较慢.
75 # GNU文件工具包中的"shred"命令,
76 #+ 也可以完成相同的工作, 不过更有效率.
77
78 # 使用普通的方法是不可能重新恢复这个文件了.
79 # 然而 . .
80 #+ 这个简单的例子是不能够抵抗
81 #+ 那些经验丰富并且正规的分析.
82
83 # 这个脚本可能不会很好的运行在日志文件系统上(JFS).
84 # 练习 (很难): 像它做的那样修正这个问题.
85
86
87 # Tom Vier的文件删除包可以更加彻底的删除文件,
```

```
89 #+ 比这个例子厉害的多.  
90 #      http://www.ibiblio.org/pub/Linux/utils/file/wipe-2.0.0.tar.bz2  
91  
92 #  如果想对安全删除文件这一论题进行深入的分析,  
93 #+ 可以参见Peter Gutmann的网页,  
94 #+ "Secure Deletion of Data From Magnetic and Solid-State Memory".  
95 #      http://www.cs.auckland.ac.nz/~pgut001/pubs/secure_del.html
```

od

od, 或者octal dump过滤器, 将会把输入(或文件)转换为8进制或者其他进制. 在你需要查看或处理一些二进制数据文件或者一个不可读的系统设备文件的时候, 这个命令非常有用, 比如/dev/urandom, 或者是一个二进制数据过滤器. 请参考[例子9-29](#)和[例子12-13](#).

hexdump .

对二进制文件进行16进制, 8进制, 10进制, 或者ASCII码的查阅动作. 这个命令大体上与上边的od命令的作用相同, 但是远没有od命令有用.

objdump .

显示编译后的二进制文件或二进制可执行文件的信息, 以16进制的形式显示, 或者显示反汇编列表(使用-d选项).

```
1 bash$ objdump -d /bin/ls  
2 /bin/ls:      file format elf32-i386  
3  
4 Disassembly of section .init:  
5  
6 080490bc <.init>:  
7  80490bc:    55          push    %ebp  
8  80490bd:    89 e5       mov     %esp,%ebp  
9  . . .
```

mcookie .

这个命令会产生一个”magic cookie”, 这是一个128-bit(32-字符)的伪随机16进制数字, 这个数字一般都用来作为X server的鉴权”签名”. 这个命令还可以用在脚本中作为一种生成随机数的手段, 当然这是一种”小吃店”(译者注: 虽然不太正统, 但是方便快捷)的风格.

```
1 random000=$(mcookie)
```

当然, 要想达到同样的目的还可以使用[md5](#)命令.

```
1 # 产生关于脚本自身的md5 checksum.  
2 random001='md5sum $0 | awk '{print $1}''  
3 # 使用 'awk' 来去掉文件名.
```

mcookie命令还给出了另一种产生”唯一”文件名的方法.

例子12-56. 文件名产生器

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

```

2 # tempfile-name.sh: 临时文件名产生器
3
4 BASE_STR='mcookie'    # 32-字符的magic cookie.
5 POS=11                 # 字符串中随便的一个位置.
6 LEN=5                  # 取得$LEN长度连续的字符串.
7
8 prefix=temp            # 最终的一个"临时"文件.
9                      # 如果想让这个文件更加"唯一",
10                     ##+ 可以对这个前缀也采用下边的方法进行生成.
11
12 suffix=${BASE_STR:POS:LEN}
13                      # 提取从第11个字符之后的长度为5的字符串.
14
15 temp_filename=$prefix.$suffix
16                      # 构造文件名.
17
18 echo "Temp filename = \"$temp_filename\""
19
20 # sh tempfile-name.sh
21 # Temp filename = temp.e19ea
22
23 # 与使用'date'命令(参考 ex51.sh)来创建"唯一"文件名
24 ##+ 的方法相比较.
25
26 exit 0

```

units .

这个工具用来在不同的计量单位之间互相转换. 当你在交互模式下正常调用时, 会发现在脚本中units命令也是非常有用的.

例子12-57. 将长度单位-米, 转化为英里

```

1#!/bin/bash
2# unit-conversion.sh
3
4
5 convert_units ()  # 通过参数取得需要转换的单位.
6{
7    cf=$((units "$1" "$2" | sed --silent -e '1p' | awk '{print $2}'))
8    # 除了真正需要转换的部分保留下外, 其他的部分都去掉.
9    echo "$cf"
10}
11
12 Unit1=miles
13 Unit2=meters
14 cfactor='convert_units $Unit1 $Unit2'

```

```

15 quantity=3.73
16
17 result=$(echo $quantity*$cfactor | bc)
18
19 echo "There are $result $Unit2 in $quantity $Unit1."
20
21 # 如果你传递了两个不匹配的单位会发生什么?
22 #+ 比如分别传入"英亩"和"英里"?
23
24 exit 0

```

m4 .

一个隐藏的财宝, m4是一个强大的宏处理过滤器, [5] 差不多可以说是一种语言了. 虽然最开始这个工具是用来作为RatFor的预处理器而编写的, 但是后来证明m4即使作为独立的工具来使用也是非常有用的. 事实上, m4结合了许多工具的功能, 比如eval, tr, 和awk, 除此之外, 它还使得宏扩展变得更加容易.

在2004年4月的Linux Journal 问题列表中有一篇关于m4命令用法的好文章.

例子12-58. 使用m4

```

1#!/bin/bash
2# m4.sh: 使用m4宏处理器
3
4# 字符串操作
5string=abcdA01
6echo "len($string)" | m4                      # 7
7echo "substr($string,4)" | m4                  # A01
8echo "regexp($string,[0-1][0-1],\&Z)" | m4    # 01Z
9
10# 算术操作
11echo "incr(22)" | m4                         # 23
12echo "eval(99 / 3)" | m4                     # 33
13
14exit 0

```

doexec

doexec命令允许将一个随便的参数列表传递到一个二进制可执行文件中. 比较特殊的, 甚至可以传递argv[0](相当于脚本中的\$0), 这样就可以使用不同的名字来调用这个可执行文件, 并且通过不同的调用名字, 还可以让这个可执行文件执行不同的动作. 这也可以说是一种将参数传递到可执行文件中的比较绕圈子的做法.

比如, /usr/local/bin目录可能包含一个”aaa”的二进制文件. 使用doexec /usr/local/bin/aaa list可以列出当前工作目录下所有以”a”开头的文件, 而使用doexec /usr/local/bin/aaa delete 将会删除这些文件.

►: 可执行文件的不同行为必须定义在可执行文件自身的代码中, 可以使用如下的shell脚本来做类比:

```

1 case `basename $0` in

```

```
2 "name1" ) do_something;;
3 "name2" ) do_something_else;;
4 "name3" ) do_yet_another_thing;;
5 *       ) bail_out;;
6 esac
```

dialog .

dialog工具集提供了一种从脚本中调用交互对话框的方法。dialog更好的变种版本是gdialog, Xdialog, 和kdialog – 事实上是调用X-Windows的界面工具集。请参考[例子33-19](#)。

sox .

sox命令, 也就是"sound exchange"命令, 可以进行声音文件的转换。事实上, 可执行文件/usr/bin/play(现在不建议使用)只不过是sox的一个shell包装器而已。

举个例子, sox soundfile.wav soundfile.au将会把一个WAV文件转换成(Sun音频格式)AU声音文件。

Shell脚本非常适合于使用sox的声音操作来批处理声音文件。比如, Linux Radio Timeshift HOWTO和MP3do Project.

注意事项

1. 这个工具事实上是从Debian Linux发行版中的一个脚本借鉴过来的。
2. 打印队列就是"在线等待"打印的作业组。
3. 对于本话题的一个完美的介绍, 请参考Andy Vaught的文章, 命名管道的介绍, 这是Linux Journal1997年9月的一个主题。
4. EBCDIC (发音是"ebb-sid-ick") 是单词(Extended Binary Coded Decimal Interchange Code)的首字母缩写。这是IBM的数据格式, 现在已经不常见了。dd命令的conv=ebcdic选项有一种比较古怪的用法, 那就是对一个文件进行快速容易但不太安全的编码。

```
1 cat $file | dd conv=swab,ebcdic > $file_encrypted
2 # 编码(看起来好像没什么用).
3 # 应该交换字节(swab), 有点晦涩.
4
5 cat $file_encrypted | dd conv=swab,ascii > $file_plaintext
6 # 解码.
```

5. 宏是一个符号常量, 将会被扩展成一个命令字符串或者一系列的参数进行操作。

第13章 系统与管理命令

本章目录

1. 分析一个系统脚本

在/etc/rc.d目录中的启动和关机脚本中包含了好多有用的(和没用的)系统管理命令. 这些命令通常总是被root用户使用, 用于系统维护或者是紧急系统文件修复. 一定要小心使用这些工具, 因为如果滥用的话, 它们会损坏你的系统.

User和Group类

users .

显示所有的登录用户. 这个命令与who -q基本一致.

groups .

列出当前用户和他所属的组. 这相当于\$GROUPS内部变量, 但是这个命令将会给出组名字, 而不是数字.

```
1 bash$ groups
2 bozita cdrom cdwriter audio xgrp
3
4 bash$ echo $GROUPS
5 501
```

chown, chgrp .

chown命令将会修改一个或多个文件的所有权. 对于root用户来说, 如果他想将文件的所有权从一个用户换到另一个用户的话, 那么使用这个命令是非常好的选择. 一个普通用户不能修改文件的所有权, 即使他是文件的宿主也不行. [1]

```
1 root# chown bozo *.txt
```

chgrp将会修改一个或多个文件的group所有权. 但前提是你必须是这些文件的宿主, 并且必须是目的组的成员(或者是root), 这样你才能够使用这个命令.

```
1 chgrp --recursive dunderheads *.data
2 # "dunderheads"(译者: 晕, 蠢才...) 组现在拥有了所有的"*.data"文件.
3 #+ 包括所有$PWD目录下的子目录中的文件(--recursive的作用就是包含子目录).
```

useradd, userdel .

useradd管理命令将会在系统上添加一个用户帐号, 并且如果指定的话, 还会为特定的用户创建home目录. 相应的, userdel命令将会从系统上删除一个用户帐号, [2] 并且会删除相应的文件.

►: adduser与useradd是完全相同的, adduser通常仅仅是个符号链接.

usermod .

修改用户帐号. 可以修改给定用户帐号的密码, 组身份, 截止日期, 或者其他一些属性. 使用这个命令, 用户的密码可能会被锁定, 因为密码会影响到帐号的有效性.

groupmod .

修改指定组. 组名字或者ID号都可以用这个命令来修改.

id .

id命令将会列出当前进程真实有效的用户ID, 还有用户的组ID. 这与Bash内部变量\$UID, \$EUID, 和\$GROUPS很相像.

```
1 bash$ id
2 uid=501(bozo) gid=501(bozo) groups=501(bozo),22(cdrom),80(cdwriter),81(audio)
3
4 bash$ echo $UID
5 501
```

►: id命令只有在有效ID与实际ID不符时, 才会显示有效ID.

请参考[例子9-5](#).

who .

显示系统上所有已经登录的用户.

```
1 bash$ who
2 bozo  tty1      Apr 27 17:45
3 bozo  pts/0      Apr 27 17:46
4 bozo  pts/1      Apr 27 17:47
5 bozo  pts/2      Apr 27 17:49
```

-m选项将会给出当前用户的详细信息. 将任意两个参数传递到who中, 都等价于who -m, 就像who am i或who The Man.

```
1 bash$ who -m
2 localhost.localdomain!bozo  pts/2      Apr 27 17:49
```

whoami与who -m很相似, 但是只列出用户名.

```
1 bash$ whoami
2 bozo
```

w .

显示所有的登录用户和属于它们的进程. 这是一个who命令的扩展版本. w的输出可以通过管道传递到grep命令中, 这样就可以查找指定的用户或进程.

```
1 bash$ w | grep startx
2 bozo  tty1      -          4:22pm  6:41   4.47s  0.45s  startx
```

logname .

显示当前用户的登录名(可以在/var/run/utmp中找到). 这与上边的whoami很相近.

```
1 bash$ basename  
2 bozo  
3  
4 bash$ whoami  
5 bozo
```

然而...

```
1 bash$ su  
2 Password: .....  
3  
4 bash# whoami  
5 root  
6 bash# basename  
7 bozo
```

►: basename只会打印出登录的用户名, 而whoami将会给出附着到当前进程的用户名. 就像我们上边看到的那样, 这两个名字有时会不同.

su .

使用替换的用户(substitute user)身份来运行一个程序或脚本. su rjones将会以用户rjones的身份来启动shell. 使用su命令时, 如果不使用任何参数的话, 那默认就是root用户. 请参考[例子A-15](#).

sudo .

以root(或其他用户)的身份来运行一个命令. 这个命令可以用在脚本中, 这样就允许以正规的用户身份来运行脚本.

```
1 #!/bin/bash  
2  
3 # 某些命令.  
4 sudo cp /root/secretfile /home/bozo/secret  
5 # 其余的命令.
```

文件/etc/sudoers中保存有允许调用sudo命令的用户名.

passwd .

设置, 修改, 或者管理用户的密码.

passwd命令可以用在脚本中, 但是估计你不想这么用, 呵呵.

例子13-1. 设置一个新密码

```
1 #!/bin/bash  
2 # setnew-password.sh: 这个脚本仅仅用于说明passwd命令.  
3 # 如果你真想运行这个脚本, 很遗憾, 这可不是个好主意.  
4 # 这个脚本必须以root身份来运行.  
5  
6 ROOT_UID=0          # Root的$UID为0.  
7 E_WRONG_USER=65     # 不是root用户?
```

```

8
9 E_NOSUCHUSER=70
10 SUCCESS=0
11
12
13 if [ "$UID" -ne "$ROOT_UID" ]
14 then
15     echo; echo "Only root can run this script."; echo
16     exit $E_WRONG_USER
17 else
18     echo
19     echo "You should know better than to run this script, root."
20     echo "Even root users get the blues... "
21     echo
22 fi
23
24
25 username=bozo
26 NEWPASSWORD=securityViolation
27
28 # 检查bozo是否在这里.
29 grep -q "$username" /etc/passwd
30 if [ $? -ne $SUCCESS ]
31 then
32     echo "User $username does not exist."
33     echo "No password changed."
34     exit $E_NOSUCHUSER
35 fi
36
37 echo "$NEWPASSWORD" | passwd --stdin "$username"
38 # 'passwd'命令的'--stdin'选项允许
39 #+ 从stdin(或者管道)中获得一个新的密码.
40
41 echo; echo "User $username's password changed!"
42
43 # 在脚本中使用'passwd'命令是非常危险的.
44
45 exit 0

```

passwd命令的-l, -u, 和-d选项允许锁定, 解锁, 和删除一个用户的密码. 只有root用户可以使用这些选项.

ac .

显示用户登录的连接时间, 就像从/var/log/wtmp中读取一样. 这是一个GNU统计工具.

```
1 | bash$ ac
```

```
2      total      68.08
```

last .

用户最后登录的信息, 就像从/var/log/wtmp中读出来一样. 这个命令也可以用来显示远端登录.

比如, 显示最后几次系统的重启信息:

```
1 bash$ last reboot
2 reboot    system boot  2.6.9-1.667      Fri Feb  4 18:18      (00:02)
3 reboot    system boot  2.6.9-1.667      Fri Feb  4 15:20      (01:27)
4 reboot    system boot  2.6.9-1.667      Fri Feb  4 12:56      (00:49)
5 reboot    system boot  2.6.9-1.667      Thu Feb  3 21:08      (02:17)
6 . . .
7
8 wtmp begins Tue Feb  1 12:50:09 2005
```

newgrp .

不用登出就可以修改用户的组ID. 并且允许访问新组的文件. 因为用户可能同时属于多个组, 这个命令很少被使用.

终端类命令

tty .

显示当前用户终端的名字. 注意每一个单独的xterm窗口都被算作一个不同的终端.

```
1 bash$ tty
2 /dev/pts/1
```

stty .

显示并(或)修改终端设置. 这个复杂命令可以用在脚本中, 并可以用来控制终端的行为和其显示输出的方法. 参见这个命令的info页, 并仔细学习它.

例子13-2. 设置一个擦除字符

```
1#!/bin/bash
2# erase.sh: 在读取输入时使用"stty"来设置一个擦除字符.
3
4echo -n "What is your name? "
5read name          # 试试用退格键
6                     #+ 来删除输入的字符.
7                      # 有什么问题?
8echo "Your name is $name."
9
10stty erase '#'      # 将"hashmark"(#)设置为退格字符.
11echo -n "What is your name? "
12read name          # 使用#来删除最后键入的字符.
13echo "Your name is $name."
```

```
14  
15 # 警告：即使在脚本退出后，新的键值还是保持着这个设置。  
16 # (译者：可以使用stty erase '^?'进行恢复)  
17  
18 exit 0
```

例子13-3. 保密密码: 关闭终端对于密码的echo

```
1#!/bin/bash  
2# secret-pw.sh: 保护密码不被显示  
3  
4echo  
5echo -n "Enter password "  
6read passwd  
7echo "password is $passwd"  
8echo -n "If someone had been looking over your shoulder, "  
9echo "your password would have been compromised."  
10  
11echo && echo # 在一个"与列表"中产生两个换行。  
12  
13  
14stty -echo # 关闭屏幕的echo。  
15  
16echo -n "Enter password again "  
17read passwd  
18echo  
19echo "password is $passwd"  
20echo  
21  
22stty echo # 恢复屏幕的echo。  
23  
24exit 0  
25  
26# 详细的阅读stty命令的info页，以便于更好的掌握这个有用并且狡猾的工具。
```

一个创造性的stty命令的用法, 检测用户所按的键(不用敲回车).

例子13-4. 按键检测

```
1#!/bin/bash  
2# keypress.sh: 检测用户按键("hot keys").  
3  
4echo  
5  
6old_tty_settings=$(stty -g) # 保存老的设置(为什么?).  
7stty -icanon  
8Keypress=$(head -c1) # 或者使用$(dd bs=1 count=1 2> /dev/null)  
9 # 在非GNU系统上  
10
```

```
11 echo
12 echo "Key pressed was \\"$Keypress"\"."
13 echo
14
15 stty "$old_tty_settings"      # 恢复老的设置.
16
17 # 感谢, Stephane Chazelas.
18
19 exit 0
```

请参考[例子9-3](#).

```
1 终端与模式terminals and modes
2
3
4 一般情况下, 一个终端都是工作在canonical(标准)模式下.
5 当用户按键后, 事实上所产生的字符并没有马上传递到运行在当前终端上的程序.
6 终端上的一个本地缓存保存了这些按键.
7 当用按下回车键的时候, 才会将所有保存的按键信息传递到运行的程序中.
8 这就意味着在终端内部存在一个基本的行编辑器.
9
10
11 bash$ stty -a
12 speed 9600 baud; rows 36; columns 96; line = 0;
13 intr = ^C; quit = ^\; erase = ^H; kill = ^U; eof = ^D;
14 eol = <undef>; eol2 = <undef>; start = ^Q; stop = ^S;
15 susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V; flush = ^O;
16 ...
17 isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt
18
19
20 在使用canonical模式的时候, 可以对本地终端行编辑器所定义的特殊按键进行重新定义.
21
22
23 bash$ cat > filexxx
24 wha<ctl-W>I<ctl-H>foo bar<ctl-U>hello world<ENTER>
25 <ctl-D>
26 bash$ cat filexxx
27 hello world
28 bash$ wc -c < filexxx
29 12
30
31
32 控制终端的进程只保存了12个字符(11个字母加上一个换行), 虽然用户敲了26个按键.
33 在non-canonical("raw")模式下, 每次按键(包括特殊定义的按键,
34 比如ctrl-H)都将会立即发送一个字符到控制进程中.
```

```
35
36 Bash提示符禁用了icanon和echo，因为它用自己的行编辑器代替了终端的基本行编辑器，  
37 因为Bash的行编辑器更好。  
38 比如，当你在Bash提示符下敲ctl-A的时候，终端将不会显示^A，  
39 但是Bash将会获得\1字符，然后解释这个字符，这样光标就移动到行首了。  
40
41 Stephane Chazelas
```

setterm .

设置特定的终端属性。这个命令将向它所在终端的stdout发送一个字符串，这个字符串将修改终端的行为。

```
1 bash$ setterm -cursor off
2 bash$
```

setterm命令可以放在脚本中用来修改写入到stdout上的文本的外观。当然，如果你只想完成这个目的的话，还有[更合适的工具](#)可以用。

```
1 setterm -bold on
2 echo bold hello
3
4 setterm -bold off
5 echo normal hello
```

tset .

显示或初始化终端设置。可以把它看成一个功能比较弱的stty命令。

```
1 bash$ tset -r
2 Terminal type is xterm-xfree86.
3 Kill is control-U (^U).
4 Interrupt is control-C (^C).
```

setserial .

设置或者显示串口参数。这个脚本只能被root用户来运行，并且通常都在系统安装脚本中使用。

```
1 # 来自于/etc/pcmcia/serial脚本：
2
3 IRQ='setserial /dev/$DEVICE | sed -e 's/.*/IRQ: //'
4 setserial /dev/$DEVICE irq 0 ; setserial /dev/$DEVICE irq $IRQ
```

getty, agetty .

一个终端的初始化过程通常都是使用getty或agetty来建立，这样才能让用户登录。这些命令并不用在用户的shell脚本中。它们的行为与stty很相似。

mesg .

启用或禁用当前用户终端的访问权限. 禁用访问权限将会阻止网络上的另一用户向这个终端写消息.

当你正在编写文本文件的时候, 在文本中间突然来了一个莫名其妙的消息, 你会觉得非常烦人. 在多用户的网络环境下, 如果你不想被打断, 那么你必须关闭其他用户对你终端的写权限.

wall .

这是一个缩写单词"write all", 也就是, 向登录到网络上的所有终端的所有用户都发送一个消息. 最早这是一个管理员的工具, 很有用, 比如, 当系统有问题的时候, 管理可以警告系统上的所有人暂时离开(请参考[例子17-1](#)).

```
1 bash$ wall System going down for maintenance in 5 minutes!
2 Broadcast message from bozo (pts/1) Sun Jul  8 13:53:27 2001...
3
4 System going down for maintenance in 5 minutes!
```

►: 如果某个特定终端使用mesg来禁止了写权限, 那么wall将不会给它发消息.

信息与统计类

uname .

显示系统信息(OS, 内核版本, 等等.) ,输出到stdout上. 使用-a选项, 将会给出详细的系统信息(请参考[例子12-5](#)). 使用-s选项只会输出OS类型.

```
1 bash$ uname -a
2 Linux localhost.localdomain 2.2.15-2.5.0 \
3 #1 Sat Feb 5 00:13:43 EST 2000 i686 unknown
4
5 bash$ uname -s
6 Linux
```

arch .

显示系统的硬件体系结构. 等价于uname -m. 请参考[例子10-26](#).

```
1 bash$ arch
2 i686
3
4 bash$ uname -m
5 i686
```

lastcomm .

给出前一个命令的信息, 存储在/var/account/pacct文件中. 命令名字和用户名字都可以通过选项来指定. 这是GNU的一个统计工具.

lastlog .

列出系统上所有用户最后登录的时间. 然后保存到/var/log/lastlog文件中.

```
1 bash$ lastlog
2 root          tty1                      Fri Dec  7 18:43:21 -0700 2001
```

```

3 bin                                **Never logged in**
4 daemon                            **Never logged in**
5 ...
6 bozo      tty1                      Sat Dec  8 21:14:29 -0700 2001
7
8
9 bash$ lastlog | grep root
10 root     tty1                      Fri Dec  7 18:43:21 -0700 2001

```

⑧: 如果用户对于文件/var/log/lastlog没有读权限的话, 那么调用这个命令就会失败.

lsof .

列出打开的文件. 这个命令将会把所有当前打开的文件都列出到一份详细的表格中, 包括文件的宿主信息, 尺寸, 还有与它们相关的信息等等. 当然, lsof也可以通过管道输出到grep和(或)awk中, 来分析它的内容.

```

1 bash$ lsof
2 COMMAND   PID   USER   FD   TYPE   DEVICE   SIZE   NODE NAME
3 init       1   root    mem   REG      3,5    30748  30303 /sbin/init
4 init       1   root    mem   REG      3,5    73120  8069  /lib/ld-2.1.3.so
5 init       1   root    mem   REG      3,5   931668  8075  /lib/libc-2.1.3.so
6 cardmgr    213  root    mem   REG      3,5   36956  30357 /sbin/cardmgr
7 ...

```

strace .

系统跟踪(System trace): 是跟踪系统调用和信号的诊断和调试工具. 如果你想了解特定的程序或者工具包为什么运行失败的话, 那么这个命令和下边的ltrace命令就显得非常有用了, . . .当然, 这种失败现象可能是由缺少相关的库, 或者其他问题所引起.

```

1 bash$ strace df
2 execve("/bin/df", ["df"], /* 45 vars */) = 0
3 uname({sys="Linux", node="bozo.localdomain", ...}) = 0
4 brk(0)                               = 0x804f5e4
5
6 ...

```

这是Solaris truss命令的Linux的等价工具.

ltrace .

库跟踪工具(Library trace): 跟踪给定命令的调用库的相关信息.

```

1 bash$ ltrace df
2 __libc_start_main(0x804a910, 1, 0xbfb589a4, 0x804fb70, 0x804fb68 <unfinished. . .>)
3 setlocale(6, "")                     = "en_US.UTF-8"
4 bindtextdomain("coreutils", "/usr/share/locale") = "/usr/share/locale"
5 textdomain("coreutils")               = "coreutils"
6 __cxa_atexit(0x804b650, 0, 0, 0x8052bf0, 0xbfb58908) = 0
7 getenv("DF_BLOCK_SIZE")              = NULL

```

```
8  
9 ...  
10
```

nmap .

网络映射(Network mapper)与端口扫描程序. 这个命令将会扫描一个服务器来定位打开的端口, 并且定位与这些端口相关的服务. 这个命令也能够上报一些包过滤与防火墙的信息. 这是一个防止网络被黑客入侵的非常重要的安全工具.

```
1 #!/bin/bash  
2  
3 SERVER=$HOST # localhost.localdomain (127.0.0.1).  
4 PORT_NUMBER=25 # SMTP端口.  
5  
6 nmap $SERVER | grep -w "$PORT_NUMBER" # 察看指定端口是否打开?  
7 # grep -w 匹配整个单词.  
8 #+ 这样就不会匹配类似于1025这种含有25的端口了.  
9  
10 exit 0  
11  
12 # 25/tcp open smtp
```

nc .

nc(netcat)工具是一个完整的工具包, 可以用它连接和监听TCP和UDP端口. 它能作为诊断和测试工具, 也能作为基于脚本的HTTP客户端和服务器组件.

```
1 bash$ nc localhost.localdomain 25  
2 220 localhost.localdomain ESMTP Sendmail 8.13.1/8.13.1;  
3 Thu, 31 Mar 2005 15:41:35 -0700
```

例子13-5. 扫描远程机器上的identd服务进程

```
1 #! /bin/sh  
2 ## 使用netcat工具写的和DaveG写的ident-scan扫描器有同等功能的东西.  
3 ## 噢, 他会被气死的.  
4 ## 参数: target port [port port port ...]  
5 ## 标准输出和标准输入被混到一块.  
6 ##  
7 ## 优点: 运行起来比ident-scan慢,  
8 ## 这样使远程机器inetd进程更不易注意而不会产生警告,  
9 ##+ 并且只有很少的知名端口会被指定.  
10 ## 缺点: 要求数字端口参数, 输出中无法区分标准输出和标准错误,  
11 ##+ 并且当远程服务监听在很高的端口时无法工作.  
12 # 脚本作者: Hobbit <hobbit@avian.org>  
13 # 已征得作者同意在ABS指南中使用.  
14  
15 # -----
```

```

16 E_BADARGS=65      # 至少需要两个参数.
17 TWO_WINKS=2       # 睡眠多长时间.
18 THREE_WINKS=3
19 IDPORT=113        # indent协议的认证端口.
20 RAND1=999
21 RAND2=31337
22 TIMEOUT0=9
23 TIMEOUT1=8
24 TIMEOUT2=4
25 # -----
26
27 case "${2}" in
28   "") echo "Need HOST and at least one PORT." ; exit $E_BADARGS ;;
29 esac
30
31 # 测试目标主机看是否运行了identd守护进程.
32 nc -z -w $TIMEOUT0 "$1" $IDPORT || \
33 { echo "Oops, $1 isn't running identd." ; exit 0 ; }
34 # -z 选项扫描监听进程.
35 # -w $TIMEOUT = 尝试连接多长时间.
36
37 # 产生一个随机的本地起点源端口.
38 RP='expr $$ % $RAND1 + $RAND2'
39
40 TRG="$1"
41 shift
42
43 while test "$1" ; do
44   nc -v -w $TIMEOUT1 -p ${RP} "$TRG" ${1} < /dev/null > /dev/null &
45   PROC=$!
46   sleep $THREE_WINKS
47   echo "${1},${RP}" | nc -w $TIMEOUT2 -r "$TRG" $IDPORT 2>&1
48   sleep $TWO_WINKS
49
50 # 这看上去是不是像个残疾的脚本或是其他类似的东西... ?
51 # ABS作者评注："这不是真的那么糟糕,
52 #+          事实上，做得非常聪明."
53
54   kill -HUP $PROC
55   RP='expr ${RP} + 1'
56   shift
57 done
58
59 exit $?
60

```

```
61 # 注意事项:  
62 # -----  
63  
64 # 试着把第30行去掉,  
65 #+ 然后以"localhost.localdomain 25"为参数来运行脚本.  
66  
67 # 关于Hobbit写的更多'nc'例子脚本,  
68 #+ 可以在以下文档中找到:  
69 #+ /usr/share/doc/nc-X.XX/scripts 目录.
```

并且,当然,这里还有Dr. Andrew Tridgell在BistKeeper事件中臭名卓著的一行脚本:

```
1 echo clone | nc thunk.org 5000 > e2fsprogs.dat
```

free .

使用表格形式来显示内存和缓存的使用情况. 这个命令的输出非常适合于使用[grep](#), [awk](#)或者Perl来分析. [procinfo](#)将会显示free命令所能显示的所有信息, 而且更加详细.

```
1 bash$ free  
2  
3          total        used        free      shared      buffers      cached  
4   Mem:       30504       28624       1880      15820       1608      16376  
5 -/+ buffers/cache:       10640       19864  
6 Swap:       68540       3128      65412
```

打印出未使用的RAM内存:

```
1 bash$ free | grep Mem | awk '{ print $4 }'  
2 1880
```

procinfo .

从[/proc pseudo-fs](#)中提取并显示所有信息和统计资料. 这个命令将给出更详细的信息.

```
1 bash$ procinfo | grep Bootup  
2 Bootup: Wed Mar 21 15:15:50 2001      Load average: 0.04 0.21 0.34 3/47 6829
```

lsdev .

列出系统设备,也就是显示所有安装的硬件.

```
1 bash$ lsdev  
2 Device          DMA    IRQ  I/O Ports  
3 -----  
4 cascade          4      2  
5 dma              0080-008f  
6 dma1             0000-001f  
7 dma2             00c0-00df  
8 fpu              00f0-00ff  
9 ide0            14     01f0-01f7 03f6-03f6
```

```
10 ...
```

du .

递归的显示(磁盘)文件的使用状况. 除非特殊指定, 否则默认是当前工作目录.

```
1 bash$ du -ach
2 1.0k    ./wi.sh
3 1.0k    ./tst.sh
4 1.0k    ./random.file
5 6.0k    .
6 6.0k    total
```

df .

使用列表的形式显示文件系统的使用状况.

```
1 bash$ df
2 Filesystem      1k-blocks   Used Available Use% Mounted on
3 /dev/hda5        273262    92607   166547  36% /
4 /dev/hda8        222525   123951   87085  59% /home
5 /dev/hda7        1408796  1075744  261488  80% /usr
```

dmesg .

将所有的系统启动消息输出到stdout上, 方便除错, 并且可以查出安装了哪些设备驱动和察看使用了哪些系统中断. dmesg命令的输出当然也放在脚本中, 并使用[grep](#), [sed](#), 或[awk](#)来进行分析.

```
1 bash$ dmesg | grep hda
2 Kernel command line: ro root=/dev/hda2
3 hda: IBM-DLGA-23080, ATA DISK drive
4 hda: 6015744 sectors (3080 MB) w/96KiB Cache, CHS=746/128/63
5 hda: hda1 hda2 hda3 < hda5 hda6 hda7 > hda4
```

stat .

显示一个或多个给定文件(也可以是目录文件或设备文件)的详细统计信息(statistic).

```
1 bash$ stat test.cru
2 File: "test.cru"
3 Size: 49970          Allocated Blocks: 100           Filetype: Regular File
4 Mode: (0664/-rw-rw-r--)
5 Device:  3,8   Inode: 18185   Links: 1
6 Access: Sat Jun  2 16:40:24 2001
7 Modify: Sat Jun  2 16:40:24 2001
8 Change: Sat Jun  2 16:40:24 2001
```

如果目标文件不存在, stat将会返回一个错误消息.

```
1 bash$ stat nonexistent-file
```

```
2 nonexistent-file: No such file or directory
```

vmstat .

显示虚拟内存的统计信息.

```
1 bash$ vmstat
2 procs          memory      swap      io system      cpu
3   r   b   w   swpd   free   buff   cache   si   so   bi   bo   in   cs   us   sy id
4   0   0   0     0 11040   2636  38952   0   0    33    7 271   88   8   3 89
```

netstat .

显示当前网络的统计状况和信息, 比如路由表和激活的连接, 这个工具将访问[/proc/net](#)中的信息. 请参考[例子27-3](#).

netstat -r等价于route命令.

uptime .

显示系统运行的时间, 还有其他的一些统计信息.

```
1 bash$ uptime
2 10:28pm  up  1:57,  3 users,  load average: 0.17, 0.34, 0.27
```

load average 如果小于或等于1, 那么就意味着系统会马上处理. 如果大于1, 那么就意味着进程需要排队. 如果大于3, 那么就意味着, 系统性能已经显著下降了.

hostname .

显示系统的主机名字. 这个命令在/etc/rc.d安装脚本(或类似的/etc/rc.d/rc.sysinit)中设置主机名. 等价于uname -n, 并且与\$HOSTNAME内部变量很相像.

与hostname命令很相像的命令还有domainname, dnsdomainname, nisdomainname, 和ypdomainname命令. 使用这些命令来显示(或设置)系统DNS或NIS/YP域名. 对于hostname命令来说, 使用不同的选项就可以分别达到上边这些命令的目的.

hostid .

用16进制表示法来显示主机的32位ID.

```
1 bash$ hostid
2 7f0100
```

这个命令据说对于特定系统可以获得一个“唯一”的序号. 某些产品的注册过程可能会需要这个序号来作为用户的许可证. 不幸的是, hostid只会使用字节对转换的方法来返回机器的网络地址, 网络地址用16进制表示.

对于一个典型的没有网络的Linux机器来说, 它的网络地址保存在/etc/hosts中.

```
1 bash$ cat /etc/hosts
2 127.0.0.1           localhost.localdomain localhost
```

碰巧, 通过对127.0.0.1进行字节转换, 我们获得了0.127.1.0, 用16进制表示就是007f0100, 这就是上边hostid命令返回的结果. 这样几乎所有的无网络的Linux机器都会得到这个hostid.

sar . sar(System Activity Reporter系统活动报告)命令将会给出系统统计的一个非常详细概要. Santa Cruz Operation("以前的"SCO)公司在1999年4月份以开源软件的形式发布了sar.

这个命令并不是基本Linux发行版的一部分,但是你可以从sysstat utilities所编写的Sebastien Godard包中获得这个工具.

```
1 bash$ sar
2 Linux 2.4.9 (brooks.seringas.fr)          09/26/03
3
4 10:30:00      CPU    %user    %nice   %system   %iowait   %idle
5 10:40:00      all     2.21    10.90    65.48     0.00    21.41
6 10:50:00      all     3.36    0.00     72.36     0.00    24.28
7 11:00:00      all     1.12    0.00     80.77     0.00    18.11
8 Average:      all     2.23    3.63     72.87     0.00    21.27
9
10 14:32:30    LINUX RESTART
11
12 15:00:00      CPU    %user    %nice   %system   %iowait   %idle
13 15:10:00      all     8.59    2.40     17.47     0.00    71.54
14 15:20:00      all     4.07    1.00     11.95     0.00    82.98
15 15:30:00      all     0.79    2.94     7.56     0.00    88.71
16 Average:      all     6.33    1.70     14.71     0.00    77.26
```

readelf .

这个命令会显示elf格式的二进制文件的统计信息. 这个工具是binutils工具包的一部分.

```
1 bash$ readelf -h /bin/bash
2 ELF Header:
3   Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
4   Class: ELF32
5   Data: 2's complement, little endian
6   Version: 1 (current)
7   OS/ABI: UNIX - System V
8   ABI Version: 0
9   Type: EXEC (Executable file)
10  . . .
```

size .

size [/path/to/binary]命令可以显示2进制可执行文件或归档文件每部分的尺寸. 这个工具主要提供给程序员使用.

```
1 bash$ size /bin/bash
2   text     data     bss     dec      hex filename
3 495971   22496   17392  535859   82d33 /bin/bash
```

logger .

附加一个用户产生的消息到系统日志中(/var/log/messages). 即使不是root用户, 也可以调用logger.

```
1 logger Experiencing instability in network connection at 23:10, 05/21.  
2 # 现在, 运行'tail /var/log/messages'.
```

通过在脚本中调用logger命令, 就可以将调试信息写到/var/log/messages中.

```
1 logger -t $0 -i Logging at line "$LINENO".  
2 # "-t"选项可以为日志入口指定标签.  
3 # "-i"选项记录进程ID.  
4  
5 # tail /var/log/message  
6 # ...  
7 # Jul 7 20:48:58 localhost ./test.sh[1712]: Logging at line 3.
```

logrotate .

这个工具用来管理系统的log文件, 可以在合适的时候轮换, 压缩, 删除, 或(和)e-mail它们. 这个工具将从旧的log文件中取得一些杂乱的记录, 并保存到/var/log中. 一般的, 每天都是通过cron来运行logrotate.

在/etc/logrotate.conf中添加合适的入口就可以管理自己的log文件了, 就像管理系统log文件一样.

►: Stefano Falsetto创造了rottlog, 他认为这是logrotate的改进版本.

作业控制类

ps .

进程统计(Process Statistics): 通过进程宿主或PID(进程ID)来列出当前正在执行的进程. 通常都是使用ax或aux选项来调用这个命令, 并且结果可以通过管道传递到grep或sed中来搜索特定的进程(请参考[例子11-12](#)和[例子27-2](#)).

```
1 bash$ ps ax | grep sendmail  
2 295 ? S 0:00 sendmail: accepting connections on port 25
```

如果想使用”树”的形式来显示系统进程: 那么就使用ps afjx或ps ax -forest.

pgrep, pkill .

ps命令可以与grep或kill结合使用.

```
1 bash$ ps a | grep mingetty  
2 2212 tty2 Ss+ 0:00 /sbin/mingetty tty2  
3 2213 tty3 Ss+ 0:00 /sbin/mingetty tty3  
4 2214 tty4 Ss+ 0:00 /sbin/mingetty tty4  
5 2215 tty5 Ss+ 0:00 /sbin/mingetty tty5  
6 2216 tty6 Ss+ 0:00 /sbin/mingetty tty6  
7 4849 pts/2 S+ 0:00 grep mingetty  
8
```

```
9  
10 bash$ pgrep mingetty  
11 2212 mingetty  
12 2213 mingetty  
13 2214 mingetty  
14 2215 mingetty  
15 2216 mingetty
```

pstree .

使用”树”形式列出当前执行的进程. -p选项显示PID, 也就是进程名字.

top .

连续不断的显示cpu占有率最高的进程. -b选项将会以文本方式来显示, 以便于可以在脚本中进行分析或访问.

```
1 bash$ top -b  
2 8:30pm  up 3 min,  3 users,  load average: 0.49, 0.32, 0.13  
3 45 processes: 44 sleeping, 1 running, 0 zombie, 0 stopped  
4 CPU states: 13.6% user, 7.3% system, 0.0% nice, 78.9% idle  
5 Mem: 78396K av, 65468K used, 12928K free, 0K shrd, 2352K buff  
6 Swap: 157208K av, 0K used, 157208K free 37244K cached  
7  
8      PID USER      PRI  NI   SIZE  RSS SHARE STAT %CPU %MEM    TIME COMMAND  
9      848 bozo      17   0   996  996   800 R      5.6  1.2  0:00 top  
10     1 root       8   0   512  512   444 S      0.0  0.6  0:04 init  
11     2 root       9   0     0     0     0 SW      0.0  0.0  0:00 keventd  
12     ...
```

nice .

使用经过修改的优先级来运行一个后台作业. 优先级从19(最低)到-20(最高) 只有root用户可以设置负的(相对比较高的)优先级. 相关的命令还有renice, snice, 和skill.

nohup .

保持一个命令进程处于运行状态, 即使这个命令进程所属的用户登出系统. 这个命令进程将会运行在前台, 除非在它前面加上&. 如果你在脚本中使用nohup命令, 那么你最好同时使用wait命令, 这样就可以避免产生孤儿进程或僵尸进程.

pidof .

获取一个正在运行作业的进程ID(PID). 因为一些作业控制命令, 比如kill和renice只能使用进程的PID(而不是它的名字)作为参数, 所以有的时候必须得取得PID. pidof命令与\$PPID内部变量非常相似.

```
1 bash$ pidof xclock  
2 880
```

例子13-6. 使用pidof命令帮忙kill一个进程

```
1 #!/bin/bash
2 # kill-process.sh
3
4 NOPROCESS=2
5
6 process=xxxxyyzzz # 使用不存在的进程.
7 # 只不过是为了演示...
8 # ... 并不想在这个脚本中杀掉任何真正的进程.
9 #
10 # 举个例子, 如果你想使用这个脚本来断线Internet,
11 #     process=pppd
12
13 t='pidof $process'      # 取得$process的pid(进程id).
14 # 'kill'只能使用pid(不能用程序名)作为参数.
15
16 if [ -z "$t" ]          # 如果没这个进程, 'pidof' 返回空.
17 then
18     echo "Process $process was not running."
19     echo "Nothing killed."
20     exit $NOPROCESS
21 fi
22
23 kill $t                  # 对于某些顽固的进程可能需要使用'kill -9'.
24
25 # 这里需要做一个检查, 看看进程是否允许自身被kill.
26 # 或许另一个 " t='pidof $process' " 或许 ...
27
28
29 # 整个脚本都可以使用下边这句来替换:
30 #     kill $(pidof -x process_name)
31 # 或者
32 #     killall process_name
33 # 但是这就没有教育意义了.
34
35 exit 0
```

fuser .

获取一个正在访问某个或某些文件(或目录)的进程ID. 使用-k选项将会kill这些进程. 对于系统安全来说, 尤其是在脚本中想阻止未被授权的用户访问系统服务的时候, 这个命令就显得非常有用了.

```
1 bash$ fuser -u /usr/bin/vim
2 /usr/bin/vim:            3207e(bozo)
3
```

```
4  
5  
6 bash$ fuser -u /dev/null  
7 /dev/null: 3009(bozo) 3010(bozo) 3197(bozo) 3199(bozo)
```

在进行正常插入或删除保存的媒体(比如CD ROM或者USB闪存设备)的时候, fuser命令非常的有用. 某些情况下, 也就是当你umount一个设备失败的时候, 会出现设备忙错误消息. 这意味着某些用户或进程正在访问这个设备. 可以使用fuser -um /dev/device_name来解决这种问题, 这样你就可以kill所有相关的进程.

```
1 bash$ umount /mnt/usbdrive  
2 umount: /mnt/usbdrive: device is busy  
3  
4  
5  
6 bash$ fuser -um /dev/usbdrive  
7 /mnt/usbdrive: 1772c(bozo)  
8  
9 bash$ kill -9 1772  
10 bash$ umount /mnt/usbdrive
```

fuser命令的-n选项可以获得正在访问某一端口的进程. 当和nmap命令结合使用的时候尤其有用.

```
1 root# nmap localhost.localdomain  
2 PORT      STATE SERVICE  
3 25/tcp    open  smtp  
4  
5  
6  
7 root# fuser -un tcp 25  
8 25/tcp: 2095(root)  
9  
10 root# ps ax | grep 2095 | grep -v grep  
11 2095 ? Ss 0:00 sendmail: accepting connections
```

cron .

管理程序调度器, 用来执行一些日常任务, 比如清除和删除系统log文件, 或者更新slocate数据库. 这是at命令的超级用户版本(虽然每个用户都可以有自己的crontab文件, 并且这个文件可以使用crontab命令来修改). 它以幽灵进程的身份来运行, 并且从/etc/crontab中获得执行的调度入口.

►: 一般Linux风格的系统都使用crond, 用的是Matthew Dillon版本的cron.

启动与进程控制类

init .

init命令是所有进程的父进程. 在系统启动的最后一步调用, init将会依据/etc/inittab来决定

系统的运行级别. 只能使用root身份来运行它的别名- telinit.

telinit .

init命令的符号链接, 这是一种修改系统运行级别的手段, 通常在系统维护的时候, 或者在紧急的文件系统修复的时候才能用. 只能使用root身份调用. 调用这个命令是非常危险的- 在你使用之前确定你已经很好地了解它!

runlevel .

显示当前的和最后的运行级别, 也就是, 判断系统是处于终止状态(runlevel为0), 单用户模式(1), 多用户模式(2或3), X Windows(5), 还是正处于重启状态(6). 这个命令将会访问/var/run/utmp文件.

halt, shutdown, reboot .

设置系统关机的命令, 通常比电源关机的优先级高.

service .

开启或停止一个系统服务. 在/etc/init.d 和/etc/rc.d中的启动脚本使用这个命令来启动服务.

```
1 root# /sbin/service iptables stop
2 Flushing firewall rules:                                     [  OK  ]
3 Setting chains to policy ACCEPT: filter                   [  OK  ]
4 Unloading iptables modules:                                [  OK  ]
```

网络类

ifconfig .

网络的接口配置和调试工具.

```
1 bash$ ifconfig -a
2 lo      Link encap:Local Loopback
3          inet addr:127.0.0.1 Mask:255.0.0.0
4          UP LOOPBACK RUNNING MTU:16436 Metric:1
5          RX packets:10 errors:0 dropped:0 overruns:0 frame:0
6          TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
7          collisions:0 txqueuelen:0
8          RX bytes:700 (700.0 b) TX bytes:700 (700.0 b)
```

ifconfig命令绝大多数情况都是在启动的时候设置接口, 或者在重启的时候关闭它们.

```
1 # 来自于/etc/rc.d/init.d/network中的代码片断
2
3 # ...
4
5 # 检查网络是否启动.
6 [ ${NETWORKING} = "no" ] && exit 0
7
```

```

8 [ -x /sbin/ifconfig ] || exit 0
9
10 # ...
11
12 for i in $interfaces ; do
13     if ifconfig $i 2>/dev/null | grep -q "UP" >/dev/null 2>&1 ; then
14         action "Shutting down interface $i: " ./ifdown $i boot
15     fi
16 # grep命令的GNU指定选项"-q"的意思是"安静", 也就是, 不产生输出.
17 # 这样, 后边重定向到/dev/null的操作就有点多余了.
18
19 # ...
20
21 echo "Currently active devices:"
22 echo '/sbin/ifconfig | grep ^[a-z] | awk '{print $1}''
23 #                                     ^^^^^^ 应该被引用以防止通配(globbing).
24 # 下边这段也能工作.
25 #     echo $(/sbin/ifconfig | awk '/^([a-z])/{ print $1 }')
26 #     echo $(/sbin/ifconfig | sed -e 's/ .*//')
27 # 感谢, S.C., 做了额外的注释.

```

请参考[例子29-6](#).

iwconfig .

这是为了配置无线网络的命令集合. 可以说是上边的ifconfig的无线版本.

route .

显示内核路由表信息, 或者查看内核路由表的修改情况.

1	bash\$ route					
2	Destination Gateway Genmask Flags MSS Window irtt Iface					
3	pm3-67.bozosisp *	255.255.255.255	UH	40 0	0	ppp0
4	127.0.0.0 * 255.0.0.0 U	40 0	0	lo		
5	default pm3-67.bozosisp	0.0.0.0	UG	40 0	0	ppp0

chkconfig

检查网络配置. 这个命令负责显示和管理在启动过程中所开启的网络服务(这些服务都是从/etc/rc?.d目录中开启的).

最开始是从IRIX到Red Hat Linux的一个接口, chkconfig在某些Linux发行版中并不是核心安装的一部分.

1	bash\$ chkconfig --list
2	atd 0:off 1:off 2:off 3:on 4:on 5:on 6:off
3	rwhod 0:off 1:off 2:off 3:off 4:off 5:off 6:off
4	...
5	

tcpdump .

网络包的”嗅探器”. 这是一个用来分析和调试网络上传输情况的工具, 它所使用的手段是把所有匹配指定规则的包头都显示出来.

显示主机bozoville和主机caduceus之间所有传输的ip包.

```
1 bash$ tcpdump ip host bozoville and caduceus
```

当然, tcpdump的输出是可以进行分析的, 可以用我们之前讨论的文本处理工具来分析结果.

文件系统类

mount .

加载一个文件系统, 通常都用来安装外部设备, 比如软盘或CDROM. 文件/etc/fstab将会提供一个方便的列表, 这个列表列出了所有可用的文件系统, 分区和设备, 另外还包括某些选项, 比如是否可以自动或者手动的mount. 文件/etc/mtab显示了当前已经mount的文件系统和分区(包括虚拟的, 比如/proc).

mount -a将会mount所有出现在/etc/fstab中的文件系统和分区, 除了那些标记有noauto(非自动)选项的. 启动的时候, 在/etc/rc.d中的一个启动脚本(rc.sysinit或者一些相似的脚本)将会调用mount -a, 目的是mount所有可用的文件系统和分区.

```
1 mount -t iso9660 /dev/cdrom /mnt/cdrom
2 # 加载CDROM
3 mount /mnt/cdrom
4 # 如果/mnt/cdrom包含在/etc/fstab中的话, 那么这么调用就是一种简便的方法.
```

这个多功能的命令甚至可以将一个普通文件mount到块设备中, 这样一来, 就可以象做操作文件系统一样来操作这个文件. 先将这个文件与一个loopback device设备相关联, 然后Mount就能达到这个目的了. 这种应用一般都是用来mount或检查一个ISO9660镜像, 经过检查后这个镜像会被烧录到CDR上. [3]

例子13-7. 检查一个CD镜像

```
1 # 以root身份...
2
3 mkdir /mnt/cdtest # 准备一个mount入口, 如果你没准备的话.
4
5 mount -r -t iso9660 -o loop cd-image.iso /mnt/cdtest # mount这个镜像.
6 #           "-o loop" option equivalent to "losetup /dev/loop0"
7 cd /mnt/cdtest # 现在检查这个镜像.
8 ls -alr        # 列出目录树中的文件.
9             # 诸如此类.
```

umount .

卸除一个当前已经mount的文件系统. 在删除已经mount上的软盘或CDROM之前, 这个设备必须被umount, 否则文件系统将会被损坏.

automount工具, 如果对这个工具进行了适当的安装, 那么当需要访问或退出磁盘(或软盘)的时候, 就能够自动的mount和unmount它们了. 但是如果在带有软驱或光驱的笔记本电脑上使用的话, 可能会引起问题.

sync .

当你需要更新硬盘buffer中的数据时, 这个命令可以强制将你buffer上的数据立即写入到硬盘上(同步带有buffer的驱动器). 在某些情况下, 一个sync命令可能会挽救你刚刚更新的数据, 比如说突然断电, 所以这个命令可以给系统管理员和普通用户一些保障. 以前, 系统重启前都使用sync; sync (两次, 为了保证绝对可靠), 这是一种谨慎小心的可靠方法.

某些时候, 比如说当你想安全删除文件的时候(请参考[例子12-55](#)), 或者当磁盘灯开始闪烁的时候, 你可能需要强制对buffer进行立即刷新.

losetup .

建立和配置loopback设备.

例子13-8. 在一个文件中创建文件系统

```
1 SIZE=1000000 # 1 meg
2
3 head -c $SIZE < /dev/zero > file # 建立指定尺寸的文件.
4 losetup /dev/loop0 file          # 作为loopback设备来创建.
5 mke2fs /dev/loop0                # 创建文件系统.
6 mount -o loop /dev/loop0 /mnt    # mount.
7
8 # 感谢, S.C.
```

mkswap .

创建一个交换分区或文件. 交换区域随后必须马上使用swapon来启用.

swapon, swapoff .

启用/禁用交换分区或文件. 这两个命令通常在启动和关机的时候才有效.

mke2fs .

创建Linux ext2文件系统. 这个命令必须以root身份调用.

例子13-9. 添加一个新的硬盘驱动器

```
1#!/bin/bash
2
3# 在系统上添加第二块硬盘驱动器.
4# 软件配置. 假设硬件已经安装了.
5# 来自于本书作者的一篇文章.
6# 在"Linux Gazette"的问题#38上, http://www.linuxgazette.com.
7
8ROOT_UID=0      # 这个脚本必须以root身份运行.
9E_NOTROOT=67    # 非root用户将会产生这个错误.
10
11if [ "$UID" -ne "$ROOT_UID" ]
12then
13  echo "Must be root to run this script."
14  exit $E_NOTROOT
```

```

15 fi
16
17 # 要非常谨慎小心的使用!
18 # 如果某步错了, 可能会彻底摧毁你当前的文件系统.
19
20
21 NEWDISK=/dev/hdb          # 假设/dev/hdb空白. 检查一下!
22 MOUNTPOINT=/mnt/newdisk #或者选择其他的mount入口.
23
24
25 fdisk $NEWDISK
26 mke2fs -cv $NEWDISK1    # 检查坏块, 并且详细输出.
27 # 注意: /dev/hdb1, *不是* /dev/hdb!
28 mkdir $MOUNTPOINT
29 chmod 777 $MOUNTPOINT  # 让所有用户都具有全部权限.
30
31
32 # 现在, 测试一下...
33 # mount -t ext2 /dev/hdb1 /mnt/newdisk
34 # 尝试创建一个目录.
35 # 如果运行起来了, umount它, 然后继续.
36
37 # 最后一步:
38 # 将下边这行添加到/etc/fstab.
39 # /dev/hdb1 /mnt/newdisk ext2 defaults 1 1
40
41 exit 0

```

请参考[例子13-8](#)和[例子28-3](#).

une2fs .

调整ext2文件系统. 可以用来修改文件系统参数, 比如mount的最大数量. 必须以root身份调用.

⑧: 这是一个非常危险的命令. 一旦用错, 你需要自己负责, 因为它可能会破坏你的文件系统.

dumpe2fs .

打印(输出到stdout)非常详细的文件系统信息. 必须以root身份调用.

```

1 root# dumpe2fs /dev/hda7 | grep 'ount count'
2 dumpe2fs 1.19, 13-Jul-2000 for EXT2 FS 0.5b, 95/08/09
3 Mount count:           6
4 Maximum mount count:  20

```

hdparm .

显示或修改硬盘参数. 这个命令必须以root身份调用, 如果滥用的话会有危险.

fdisk .

在存储设备上(通常都是硬盘)创建和修改一个分区表. 必须以root身份使用.

Warning ⑧: 谨慎使用这个命令. 如果出错, 会破坏你现存的文件系统.

fsck, e2fsck, debugfs .

文件系统的检查, 修复, 和除错命令集合.

fsck: 检查UNIX文件系统的前端工具(也可以调用其它的工具). 文件系统的类型一般都是默认的ext2.

e2fsck: ext2文件系统检查器.

debugfs: ext2文件系统除错器. 这个功能多- 并且危险的工具, 主要用处之一就是(尝试)恢复删除的文件. 只有高级用户才能用!

Caution

上边的这几个命令都必须以root身份调用, 这些命令都很危险, 如果滥用的话会破坏文件系统.

badblocks .

检查存储设备的坏块(物理损坏). 这个命令在格式化新安装的硬盘时候, 或者在测试"备份媒体"完整性的时候会被用到. [4] 举个例子, badblocks /dev/fd0用来测试软盘.

如果badblocks使用不慎的话, 可能会引起比较糟糕的结果(覆盖所有数据), 但是在只读模式下就不会发生这种情况. 如果root用户要测试某个设备(这是通常情况), 那么root用户必须使用这个命令.

lsusb, usbmodules .

lsusb命令会显示所有USB(Universal Serial Bus通用串行总线)总线和使用USB的设备.

usbmodules命令会输出连接USB设备的驱动模块的信息.

```
1 root# lsusb
2 Bus 001 Device 001: ID 0000:0000
3   Device Descriptor:
4     bLength          18
5     bDescriptorType    1
6     bcdUSB         1.00
7     bDeviceClass      9 Hub
8     bDeviceSubClass    0
9     bDeviceProtocol    0
10    bMaxPacketSize0     8
11    idVendor           0x0000
12    idProduct          0x0000
13
14    . . .
```

lspci .

显示pci总线及其设备.

```
1 bash$ lspci
```

```
2 00:00.0 Host bridge: \
3 Intel Corporation 82845 845 (Brookdale) Chipset Host Bridge (rev 04)
4 00:01.0 PCI bridge: \
5 Intel Corporation 82845 845 (Brookdale) Chipset AGP Bridge (rev 04)
6 00:1d.0 USB Controller: Intel Corporation 82801CA/CAM USB (Hub #1) (rev 02)
7 00:1d.1 USB Controller: Intel Corporation 82801CA/CAM USB (Hub #2) (rev 02)
8 00:1d.2 USB Controller: Intel Corporation 82801CA/CAM USB (Hub #3) (rev 02)
9 00:1e.0 PCI bridge: Intel Corporation 82801 Mobile PCI Bridge (rev 42)
10
11 . . .
```

mkbootdisk .

创建启动软盘, 启动盘可以唤醒系统, 比如当MBR(master boot record)坏掉的时候. mkbootdisk命令其实是一个Bash脚本, 由Erik Troan所编写, 放在/sbin目录中.

chroot .

修改ROOT目录. 一般的命令都是从\$PATH中获得的, 相对的, 默认根目录是/. 这个命令将会把根目录修改为另一个目录(并且也将把工作目录修改到那). 这个命令对于安全目的非常有用, 举个例子, 某些情况下, 系统管理员希望限制一些特定的用户, 比如那些telnet上来的用户, 将他们限定到文件系统上一个安全的地方(有时候, 这被称为将一个guest用户限制在"chroot监牢"中). 注意, 在使用chroot命令后, 系统的二进制可执行文件的目录就不再可用.

chroot /opt将会使得原来的/usr/bin变为/opt/usr/bin. 同样, chroot /aaa/bbb /bin/ls将会使得ls命令以/aaa/bbb作为根目录, 而不是之前的/. 如果使用alias XX 'chroot /aaa/bbb ls', 并把这句放到用户的 /.bashrc文件中的话, 这样可以有效地限制运行命令"XX"时, 命令"XX"可以使用文件系统的范围.

当从启动盘恢复的时候(chroot到/dev/fd0), 或者当系统从死机状态恢复过来并作为lilo选项的时候, chroot命令都是非常方便的. 其它的应用还包括从不同的文件系统进行安装(一个rpm选项)或者从CDROM上运行一个只读文件系统. 只能以root身份调用, 小心使用.

Caution ⑧: 由于正常的\$PATH不再被关联, 所以可能需要将一些特定的系统文件拷贝到chroot之后的目录中,

lockfile .

这个工具是procmail包的一部分(www.procmail.org). 它可以创建一个锁定文件, 锁定文件是一种用来控制访问文件, 设备或资源的标记文件. 锁定文件就像一个标记一样被使用, 如果特定的文件, 设备, 或资源正在被一个特定的进程所使用("busy"), 那么对于其它进程来说, 就只能进行受限访问(或者不能访问).

```
1 lockfile /home/bozo/lockfiles/$0.lock
2 # 创建一个以脚本名字为前缀的写保护锁定文件.
```

锁定文件用在一些特定的场合, 比如说保护系统的mail目录以防止多个用户同时修改, 或者提示一个modem端口正在被访问, 或者显示Netscape的一个实例正在使用它的缓存. 脚本可以做一些检查工作, 比如说一个特定的进程可以创建一个锁定文件, 那么只要检查这个特定的进程是否在运行, 就可以判断出锁定文件是否存在了. 注意如果脚本尝试创建一个已经存在的锁定文件的话, 那么脚本很可能被挂起.

一般情况下, 应用对于锁定文件的创建和检查都放在/var/lock目录中. [5] 脚本可以使用下面的方法来检测锁定文件是否存在.

```
1 appname=xyzip
2 # 应用 "xyzip" 创建锁定文件 "/var/lock/xyzip.lock".
3
4 if [ -e "/var/lock/$appname.lock" ]
5 then
6     ...
```

flock .

flock命令不像lockfile那么有用. 它在一个文件上设置一个”咨询性”的锁, (译者注: ”咨询性”的锁有时也称为”建议性”的锁, 这种锁只对协同进程起作用, 还有一种锁叫”强制性”锁, 这种锁加锁的对象读写操作都会由内核做检查, 更多的细节请参考flock(1), flock(2), /usr/src/linux/Documentation/locks.txt和mandatory.txt), 然后在锁持续的期间可以执行一个命令. 这样可以避免这个命令完成前有另外的进程试图在这个文件上设置锁.

```
1 flock $0 cat $0 > lockfile__$0
2 # 上面这行表示脚本正处于列出自身内容到标准输出的过程中,
3 #+ 设置一把锁锁住脚本文件自身.
```

与lockfile不同, flock不会自动创建一个锁定文件.

mknod .

创建块或者字符设备文件(当在系统上安装新硬盘时, 必须这么做). MADEDEV工具事实上具有mknod的全部功能, 而且更容易使用.

MADEDEV .

创建设备文件的工具. 必须在/dev目录下, 并且以root身份使用.

```
1 root# ./MADEDEV
```

这是mknod的高级版本.

tmpwatch .

自动删除在指定时间内未被访问过的文件. 通常都是被cron调用, 用来删掉旧的log文件.

备份类

dump, restore .

dump命令是一个精巧的文件系统备份工具, 通常都用在比较大的安装版本和网络上. [6] 它读取原始的磁盘分区并且以二进制形式来写备份文件. 需要备份的文件可以保存到各种各样的存储设备上, 包括磁盘和磁带. restore命令用来恢复dump所产生的备份.

fdformat .

对软盘进行低级格式化.

系统资源类

ulimit .

设置系统资源的使用上限. 通常情况下都是使用-f选项来调用, -f用来设置文件尺寸的限制(ulimit -f 1000就是将文件大小限制为1M). -c(译者注: 这里应该是作者笔误, 作者写的是-t)选项来限制coredump尺寸(ulimit -c 0就是不要coredump). 一般情况下, ulimit的值应该设置在/etc/profile和(或) /.bash_profile中(请参考[Appendix G](#)).

合理的使用ulimit命令可以保护系统免受可怕的fork炸弹的迫害.

```
1 #!/bin/bash
2 # 这个脚本只是为了展示用.
3 # 你要自己为运行这个脚本的后果负责 -- 它*将*凝固你的系统.
4
5 while true # 死循环.
6 do
7     $0 &      # 这个脚本调用自身 . .
8         #+ fork无限次 . .
9         #+ 直到系统完全不动, 因为所有的资源都耗尽了.
10    done      # 这就像令人郁闷的"魔术师不断变出雨伞"的场景.
11
12 exit 0      # 这里不会真正的退出, 因为这个脚本不会终止.
```

当这个脚本超过预先设置的限制时, 在/etc/profile中的ulimit -Hu XX(XX就是需要限制的用户进程) 可以终止这个脚本的运行.

quota .

显示用户或组的磁盘配额.

setquota .

从命令行中设置用户或组的磁盘配额.

umask .

设定用户创建文件时缺省的权限mask(掩码). 也可以用来限制特定用户的默认文件属性. 所有用户创建的文件属性都是由umask所指定的. 传递给umask命令的值(8进制)定义了文件的屏蔽权限. 比如, umask 022将会使得新文件的权限最多为755(777与022进行”与非”操作). [7] 当然, 用户随后可以使用chmod来修改指定文件的属性. 用户一般都将设置umask值得地方放在/etc/profile或(和) /.bash_profile中(请参考[Appendix G](#)).

例子13-10. 用umask将输出文件隐藏起来

```
1 #!/bin/bash
2 # rot13a.sh: 与"rot13.sh"脚本相同, 但是会将输出写到"安全"文件中.
3
4 # 用法: ./rot13a.sh filename
5 # 或      ./rot13a.sh <filename
6 # 或      ./rot13a.sh同时提供键盘输入(stdin)
7
8 umask 177          # 文件创建掩码.
9                      # 被这个脚本所创建的文件
```

```

10          #+# 将具有600权限.
11
12 OUTFILE=decrypted.txt    # 结果保存在"decrypted.txt"中
13          #+# 这个文件只能够被
14          # 这个脚本的调用者(或者root)所读写.
15
16 cat "$@" | tr 'a-zA-Z' 'n-za-mN-ZA-M' > $OUTFILE
17 #     ^^ 从stdin 或文件中输入.           ~~~~~~ 输出重定向到文件中.
18
19 exit 0

```

rdev .

取得root device, swap space, 或video mode的相关信息, 或者对它们进行修改. 一般情况下, rdev的功能都是被lilo所使用, 但是在建立一个ram disk的时候, 这个命令也很有用. 小心使用, 这是一个危险的命令.

模块类

lsmod .

显示所有安装的内核模块.

```

1 bash$ lsmod
2 Module           Size  Used by
3 autofs            9456  2 (autoclean)
4 opl3             11376  0
5 serial_cs         5456  0 (unused)
6 sb              34752  0
7 uart401          6384  0 [sb]
8 sound            58368  0 [opl3 sb uart401]
9 soundlow          464   0 [sound]
10 soundcore        2800   6 [sb sound]
11 ds              6448   2 [serial_cs]
12 i82365          22928  2
13 pcmcia_core      45984  0 [serial_cs ds i82365]

```

►: 使用cat /proc/modules可以得到同样的结果.

insmod .

强制安装一个内核模块(如果可能的话, 使用modprobe来代替). 必须以root身份调用.

rmmod .

强制卸载一个内核模块. 必须以root身份调用.

modprobe .

模块装载器, 一般情况下都是在启动脚本中自动调用. 必须以root身份来运行.

depmod .

创建模块依赖文件, 一般都是在启动脚本中调用.

modinfo .

输出一个可装载模块的信息.

```
1 bash$ modinfo hid
2 filename:      /lib/modules/2.4.20-6/kernel/drivers/usb/hid.o
3 description:  "USB HID support drivers"
4 author:        "Andreas Gal, Vojtech Pavlik <vojtech@suse.cz>"
5 license:       "GPL"
```

杂项类

env .

使用设置过的或修改过(并不是修改整个系统环境)的[环境变量](#)来运行一个程序或脚本. 使用[`varname=xxx`]形式可以在脚本中修改环境变量. 如果没有指定参数, 那么这个命令将会显示所有设置的环境变量.

►: 在Bash或其它Bourne shell的衍生物中, 是可以在同一命令行上设置多个变量的.

```
1 1 var1=value1 var2=value2 commandXXX
2 2 # $var1和$var2只设置在'commandXXX'的环境中.
```

当不知道shell或解释器路径的时候, 脚本的第一行("#!"sha-bang"行)可以使用env.

```
1
2
3 当不知道shell或解释器路径的时候, 脚本的第一行(\#!"sha-bang"行)可以使用env.
4
5 #! /usr/bin/env perl
6
7 print "This Perl script will run,\n";
8 print "even when I don't know where to find Perl.\n";
9
10 # 在不知道perl程序路径的时候,
11 # 这么写有利于跨平台移植.
12 # 感谢, S.C.
```

ldd .

显示一个可执行文件和它所需要共享库之间依赖关系.

```
1 bash$ ldd /bin/ls
2 libc.so.6 => /lib/libc.so.6 (0x4000c000)
3 /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x80000000)
```

watch .

以指定的时间间隔来重复运行一个命令.

默认的时间间隔是2秒, 但是可以使用-n选项进行修改.

```
1 watch -n 5 tail /var/log/messages  
2 # 每隔5秒钟显示系统log文件(/var/log/messages)的结尾.
```

strip .

从可执行文件中去掉调试符号的引用. 这样做可以减小可执行文件的尺寸, 但是就不能调试了.

这个命令一般都用在[Makefile](#)中, 但是很少用在shell脚本中.

nm .

列出未strip过的, 经过编译的, 2进制文件的全部符号.

rdist .

远程分布客户端: 在远端服务器上同步, 克隆, 或者备份一个文件系统.

注意事项

1. 这是在Linux机器上或者在带有磁盘配额的UNIX系统上的真实情况.
2. 如果正在被删除的特定用户已经登录了主机, 那么userdel命令将会失败.
3. 关于烧录CDR的更多细节, 可以参考Alex Withers的文章, 创建CD, 这篇文章是1999年10月在Linux Journal上发表的.
4. [mke2fs](#)的-c选项也会进行磁盘坏块检查.
5. 因为只有root用户才具有对/var/lock目录的写权限, 一般的用户脚本不能在那里设置一个锁定文件.
6. 单用户Linux系统的操作更倾向于使用简单的备份工具, 比如tar.
7. NAND"与非"是一种逻辑操作. 事实上, 这种操作与减法很相像.

1 分析一个系统脚本

利用我们所学到的关于管理命令的知识，让我们一起来练习分析一个系统脚本。最简单并且最短的系统脚本之一是“killall”，[\[1\]](#) 这个脚本被用来在系统关机时挂起运行的脚本。

例子13-11. killall, 来自于/etc/rc.d/init.d

```
1 #!/bin/sh
2
3 # --> 本书作者所做的注释全部以 "# -->" 开头。
4
5 # --> 这是由Miquel van Smoorenburg所编写的
6 # --> 'rc' 脚本包的一部分, <miquels@drinkel.nl.mugnet.org>.
7
8 # --> 这个特殊的脚本看起来是Red Hat/FC专用的,
9 # --> (在其它的发行版中可能不会出现).
10
11 # 停止所有正在运行的不必要的服务
12 #+ (不会有多少, 所以这是个合理性检查)
13
14 for i in /var/lock/subsys/*; do
15     # --> 标准的for/in循环, 但是由于"do"在同一行上,
16     # --> 所以必须添加";".
17     # 检查脚本是否在那里.
18     [ ! -f $i ] && continue
19     # --> 这是一种使用"与列表"的聪明方法, 等价于:
20     # --> if [ ! -f "$i" ]; then continue
21
22     # 取得子系统的名称.
23     subsys=${i#/var/lock/subsys/}
24     # --> 匹配变量名, 在这里就是文件名.
25     # --> 与subsys='basename $i' 完全等价.
26
27     # --> 从锁定文件名中获得
28     # -->+ (如果那里有锁定文件的话,
29     # -->+ 那就证明进程正在运行).
30     # --> 参考一下上边所讲的"锁定文件"的内容.
31
32
33     # 终止子系统.
34     if [ -f /etc/rc.d/init.d/$subsys.init ]; then
35         /etc/rc.d/init.d/$subsys.init stop
36     else
37         /etc/rc.d/init.d/$subsys stop
38         # --> 挂起运行的作业和幽灵进程.
39         # --> 注意"stop"只是一个位置参数,
```

```
40      # -->+ 并不是shell内建命令.  
41      fi  
42 done
```

这个没有那么糟. 除了在变量匹配的地方玩了一点花样, 其它也没有别的材料了.

练习1. 在/etc/rc.d/init.d中, 分析halt脚本. 比脚本killall长一些, 但是概念上很相近. 对这个脚本做一个拷贝, 放到你的home目录下并且用它练习一下, (不要以root身份运行它). 使用-vn标志来模拟运行一下(sh -vn scriptname). 添加详细的注释. 将"action"命令修改为"echo".

练习2. 察看/etc/rc.d/init.d下的更多更复杂的脚本. 看看是不是能够理解其中的一些脚本. 使用上边的过程来分析这些脚本. 为了更详细的理解, 你可能也需要分析在/usr/share/doc/initscripts-??.?目录下的文件sysvinitfiles, 这些都是"initscripts"文档的一部分.

注意事项

1. 系统的killall脚本不应该与/usr/bin中的killall命令相混淆.

第14章 命令替换

命令替换能够重新分配一个[1] 甚至是多个命令的输出; 它会将命令的输出如实地添加到另一个上下文中.[2]

命令替换的典型用法形式, 是使用后置引用('...'). 使用后置引用的(反引号)命令会产生命令行文本.

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

这样一来, 命令的输出就能够保存到变量中, 或者传递到另一个命令中作为这个命令的参数, 甚至可以用来产生for循环的参数列表..

```
1 rm `cat filename`    # "filename"包含了需要被删除的文件列表.
2 #
3 # S. C. 指出, 这种使用方法可能会产生"参数列表太长"的错误.
4 # 更好的方法是          xargs rm -- < filename
5 # ( -- 同时涵盖了某些特殊情况,
6 # 这种特殊情况就是, 以"--"开头的文件名会产生不良结果.)
7
8 textfile_listing=`ls *.txt`
9 # 变量中包含了当前工作目录下所有的*.txt文件.
10 echo $textfile_listing
11
12 textfile_listing2=$(ls *.txt)    # 这是命令替换的另一种形式.
13 echo $textfile_listing2
14 # 同样的结果.
15
16 # 如果将文件列表放入到一个字符串中的话,
17 # 可能会混入一个新行.
18 #
19 # 一种安全的将文件列表传递到参数中的方法就是使用数组.
20 #     shopt -s nullglob    # 如果不匹配, 那就不进行文件名扩展.
21 #     textfile_listing=( *.txt )
22 #
23 # 感谢, S.C.
```

►: 命令替换将会调用一个subshell.

►: 命令替换可能会引起单词分割(word split).

```
1 COMMAND `echo a b`      # 两个参数: a and b
2
3 COMMAND "'echo a b'"    # 1个参数: "a b"
4
5 COMMAND `echo`           # 无参数
```

```
6  
7 COMMAND ``echo``      # 一个空参数  
8  
9  
10 # 感谢, S.C.
```

即使没有引起单词分割(word split), 命令替换也会去掉多余的新行.

```
1  
2  
3 命令替换可能会引起单词分割(word split).  
4  
5 COMMAND `echo a b`      # 两个参数: a and b  
6  
7 COMMAND ``echo a b``    # 1个参数: "a b"  
8  
9 COMMAND `echo`        # 无参数  
10  
11 COMMAND ``echo``       # 一个空参数  
12  
13  
14 # 感谢, S.C.  
15  
16 即使没有引起单词分割(word split), 命令替换也会去掉多余的新行.  
17  
18 # cd ``pwd`` # 这句总能正常运行.  
19 # 然而...  
20  
21 mkdir 'dir with trailing newline'  
22 ,  
23  
24 cd 'dir with trailing newline'  
25 ,  
26  
27 cd ``pwd`` # 错误消息:  
28 # bash: cd: /tmp/file with trailing newline: No such file or directory  
29  
30 cd "$PWD"   # 运行良好.  
31  
32  
33  
34 old_tty_setting=$(stty -g)  # 保存旧的终端设置.  
35 echo "Hit a key "  
36 stty -icanon -echo          # 对终端禁用"canonical"模式.  
37                      # 这样的话, 也会禁用了*本地*的echo.  
38 key=$(dd bs=1 count=1 2> /dev/null)  # 使用'dd'命令来取得一个按键.
```

```

39 stty "$old_tty_setting"      # 恢复旧的设置.
40 echo "You hit ${#key} key."  # ${#variable} = number of characters in $variable
41 #
42 # 除了回车, 你随便敲任何按键都会输出"You hit 1 key."
43 # 如果敲回车, 那么输出就是"You hit 0 key."
44 # 新行已经被命令替换吃掉了.
45
46 感谢, S.C.

```

如果用echo命令输出一个未引用变量, 而且这个变量以命令替换的结果作为值, 那么这个变量中的换行符将会被删除. 这可能会引起一些异常状况.

```

1 dir_listing='ls -l'
2 echo $dir_listing      # 未引用, 就是没用引号括起来
3
4 # 期望打印出经过排序的目录列表.
5
6 # 可惜, 我们只能获得这些:
7 # total 3 -rw-rw-r-- 1 bozo bozo 30 May 13 17:15 1.txt -rw-rw-r-- 1 bozo
8 # bozo 51 May 15 20:57 t2.sh -rwxr-xr-x 1 bozo bozo 217 Mar 5 21:13 wi.sh
9
10 # 新行消失了.
11
12
13 echo "$dir_listing"    # 引用起来
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

```

命令替换甚至允许将整个文件的内容放到变量中, 可以使用重定向或者cat命令.

```

1 variable1='<file1'      # 将"file1"的内容放到"variable1"中.
2 variable2='cat file2'   # 将"file2"的内容放到"variable2"中.
3                               # 但是这行将会fork一个新进程,
4                               #+ 所以这行代码将会比第一行代码执行得慢.
5
6 # 注意:
7 # 变量中可以包含空白,
8 #+ 甚至是(厌恶至极的), 控制字符.

```

```

1 # 摘录自系统文件, /etc/rc.d/rc.sysinit
2 #+ (这是红帽系统中的)
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> \
26         /dev/null|grep -E "^I.*Cls=03.*Prot=02"`
27     kbdoutput='cat /proc/bus/usb/devices 2> \
28         /dev/null|grep -E "^I.*Cls=03.*Prot=01"`
29 ...
30 fi

```

►: 不要将一个长文本文件的全部内容设置到变量中，，也不要将二进制文件的内容保存到变量中，即使是开玩笑也不行.

例子14-1. 愚蠢的脚本策略

```

1#!/bin/bash
2# stupid-script-tricks.sh: 朋友, 别在家试这个脚本.
3# 来自于"Stupid Script Tricks," 卷I.
4
5
6dangerous_variable='cat /boot/vmlinuz'    # 这是压缩过的Linux内核自身.
7
8echo "string-length of \$dangerous_variable = ${#dangerous_variable}"
9# 这个字符串变量的长度是$dangerous_variable = 794151
10# (不要使用as 'wc -c /boot/vmlinuz'来计算长度.)
11
12# echo "$dangerous_variable"
13# 千万别尝试这么做! 这样将挂起这个脚本.
14
15
16# 脚本作者已经意识到将二进制文件设置到
17#+ 变量中一点作用都没有.
18
19exit 0

```

注意，在这里不会发生缓冲区溢出错误。因为这是一个解释型语言的实例，Bash就是一种解释型语言，解释型语言会比编译型语言提供更多的对程序错误的保护措施。

变量替换允许将一个loop的输出设置到一个变量中。这么做的关键就是将循环中echo命令的输出全部截取。

例子14-2. 将一个循环输出的内容设置到变量中

```
1 #!/bin/bash
2 # csubloop.sh: 将循环输出的内容设置到变量中。
3
4 variable1='for i in 1 2 3 4 5
5 do
6     echo -n "$i"          # 对于在这里的命令替换来说
7 done'                   #+ 这个'echo'命令是非常关键的。
8
9 echo "variable1 = $variable1"  # variable1 = 12345
10
11
12 i=0
13 variable2='while [ "$i" -lt 10 ]
14 do
15     echo -n "$i"          # 再来一个，'echo'是必需的。
16     let "i += 1"           # 递增。
17 done'
18
19 echo "variable2 = $variable2"  # variable2 = 0123456789
20
21 # 这就证明了在一个变量的声明中
22 #+ 嵌入一个循环是可行的。
23
24 exit 0
```

命令替换使得扩展有效Bash工具集变为可能。这样，写一段小程序或者一段脚本就可以达到目的。因为程序或脚本的输出会传到stdout上(就像一个标准UNIX工具所做的那样)，然后重新将这些输出保存到变量中。(译者：作者的意思就是在这种情况下写脚本和写程序作用是一样的。)

```
1 #include <stdio.h>
2
3 /* "Hello, world." C program */
4
5 int main()
6 {
7     printf( "Hello, world." );
8     return (0);
9 }
10
11 bash$ gcc -o hello hello.c
12
```

```

13
14 #!/bin/bash
15 # hello.sh
16
17 greeting='./hello'
18 echo $greeting
19
20 bash$ sh hello.sh
21 Hello, world.

```

对于命令替换来说, \$(COMMAND)形式已经取代了后置引用”“.

```

1 output=$(sed -n "/$1/p $file)    # 来自于例子"grp.sh".
2
3 # 将文本文件的内容保存到一个变量中.
4 File_contents1=$(cat $file1)
5 File_contents2=$(<$file2)          # Bash也允许这么做.

```

\$(...)形式的命令替换在处理双反斜线(\\)时与‘...’形式不同.

```

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

```

\$(...)形式的命令替换是允许嵌套的. [3]

```

1 word_count=$( wc -w $(ls -l | awk '{print $9}') )

```

或者, 可以更加灵活. . .

例子14-3. 找anagram

(回文构词法, 可以将一个有意义的单词, 变换为1个或多个有意义的单词, 但是还是原来的字母集合)

```

1#!/bin/bash
2# agram2.sh
3# 关于命令替换嵌套的例子.
4
5# 使用"anagram"工具.
6#+ 这是作者的"yaw1"文字表软件包中的一部分.
7# http://ibiblio.org/pub/Linux/libs/yaw1-0.3.2.tar.gz
8# http://personal.riverusers.com/~thegrendel/yaw1-0.3.2.tar.gz
9
10E_NOARGS=66
11E_BADARG=67
12MINLEN=7
13
14if [ -z "$1" ]
15then

```

```

16 echo "Usage $0 LETTERSET"
17 exit $E_NOARGS          # 脚本需要一个命令行参数.
18 elif [ ${#1} -lt $MINLEN ]
19 then
20   echo "Argument must have at least $MINLEN letters."
21   exit $E_BADARG
22 fi
23
24
25
26 FILTER='.....'           # 必须至少有7个字符.
27 #      1234567
28 Anagrams=( $(echo $(anagram $1 | grep $FILTER) ) )
29 #      |     |    嵌套的命令替换.      |
30 #      (             数组分配            )
31
32 echo
33 echo "${#Anagrams[*]} 7+ letter anagrams found"
34 echo
35 echo ${Anagrams[0]}       # 第一个anagram.
36 echo ${Anagrams[1]}       # 第二个anagram.
37                 # 等等.
38
39 # echo "${Anagrams[*]}"  # 在一行上列出所有的anagram . .
40
41 # 考虑到后边还有单独的一章, 对"数组"进行详细的讲解,
42 #+ 所以在这里就不深入讨论了.
43
44 # 可以参考脚本agram.sh, 这也是一个找出anagram的例子.
45
46 exit $?

```

命令替换在脚本中使用的例子:

1. [例子10-7](#)
2. [例子10-26](#)
3. [例子9-29](#)
4. [例子12-3](#)
5. [例子12-19](#)
6. [例子12-15](#)
7. [例子12-49](#)
8. [例子10-13](#)

9. 例子10-10

10. 例子12-29

11. 例子16-8

12. 例子A-17

13. 例子27-2

14. 例子12-42

15. 例子12-43

16. 例子12-44

注意事项

1. 对于命令替换来说, 这个命令既可以是外部的系统命令, 也可以是内部脚本的内建命令, 甚至可以是[脚本函数](#).
2. 从技术的角度来讲, 命令替换将会抽取一个命令的输出, 然后使用=操作将其赋值到一个变量中.
3. 事实上, 对于后置引用的嵌套是可行的, 但是只能将内部的反引号转义才行, 就像John默认指出的那样.

```
1 word_count=' wc -w `ls -l | awk '{print $9}'` '
```

第15章 算术扩展

算术扩展提供了一种强力工具，可以在脚本中执行(整型)算法操作。可以使用backticks, double parentheses, 或let来将字符串转换为数字表达式。

一些变化

使用后置引用的算术扩展(通常都是和expr一起使用)

```
1 z='expr $z + 3'          # 'expr'命令将会执行这个扩展。
```

使用双括号形式的算术扩展，也可以使用let命令

后置引用形式的算术扩展已经被双括号形式所替代了- ((...))和\${((...))} – 当然也可以使用非常方便的let结构。

```
1 z=$((z+3))
2 z=$((z+3))                      # 也正确。
3                                         # 使用双括号的形式,
4                                        #+ 参数解引用
5                                        #+ 是可选的。
6
7 # ${((EXPRESSION))}是算数表达式.      # 不要与命令替换
8                                        #+ 相混淆。
9
10
11
12 # 使用双括号的形式也可以不用给变量赋值。
13
14 n=0
15 echo "n = $n"                      # n = 0
16
17 (( n += 1 ))                      # 递增。
18 # (( $n += 1 )) is incorrect!
19 echo "n = $n"                      # n = 1
20
21
22 let z=z+3
23 let "z += 3"  # 使用引用的形式，允许在变量赋值的时候存在空格。
24                                         # 'let'命令事实上执行得的是算术赋值,
25                                        #+ 而不是算术扩展。
```

下边是一些在脚本中使用算术扩展的例子：

1. [例子12-9](#)
2. [例子10-14](#)
3. [例子26-1](#)

4. 例子26-11

5. 例子A-17

第16章 I/O重定向

本章目录

1. [使用exec](#)
 2. [代码块重定向](#)
 3. [重定向的应用](#)
-

默认情况下始终有3个”文件”处于打开状态, stdin(键盘), stdout(屏幕), 和stderr(错误消息输出到屏幕上). 这3个文件和其他打开的文件都可以被重定向. 对于重定向简单的解释就是捕捉一个文件, 命令, 程序, 脚本, 或者是脚本中的代码块(请参考[例子3-1](#)和[例子3-2](#))的输出, 然后将这些输出作为输入发送到另一个文件, 命令, 程序, 或脚本中.

每个打开的文件都会被分配一个文件描述符. [1] stdin, stdout, 和stderr的文件描述符分别是0, 1, 和2. 除了这3个文件, 对于其他那些需要打开的文件, 保留了文件描述符3到9. 在某些情况下, 将这些额外的文件描述符分配给stdin, stdout, 或stderr作为临时的副本链接是非常有用的. [2] 在经过复杂的重定向和刷新之后需要把它们恢复成正常状态(请参考[例子16-1](#)).

```
1 COMMAND_OUTPUT >
2   # 将stdout重定向到一个文件.
3   # 如果这个文件不存在, 那就创建, 否则就覆盖.
4
5   ls -lR > dir-tree.list
6   # 创建一个包含目录树列表的文件.
7
8 : > filename
9   # >操作, 将会把文件"filename"变为一个空文件(就是size为0).
10  # 如果文件不存在, 那么就创建一个0长度的文件(与'touch'的效果相同).
11  # :是一个占位符, 不产生任何输出.
12
13 > filename
14  # >操作, 将会把文件"filename"变为一个空文件(就是size为0).
15  # 如果文件不存在, 那么就创建一个0长度的文件(与'touch'的效果相同).
16  # (与上边的": >"效果相同, 但是某些shell可能不支持这种形式.)
17
18 COMMAND_OUTPUT >>
19   # 将stdout重定向到一个文件.
20   # 如果文件不存在, 那么就创建它, 如果存在, 那么就追加到文件后边.
21
22
23   # 单行重定向命令(只会影响它们所在的行):
24   # -----
```

```

25
26 1>filename
27      # 重定向stdout到文件"filename".
28
29 1>>filename
30      # 重定向并追加stdout到文件"filename".
31
32 2>filename
33      # 重定向stderr到文件"filename".
34
35 2>>filename
36      # 重定向并追加stderr到文件"filename".
37
38 &>filename
39      # 将stdout和stderr都重定向到文件"filename".
40
41 M>N
42      # "M"是一个文件描述符, 如果没有明确指定的话默认为1.
43      # "N"是一个文件名.
44      # 文件描述符"M"被重定向到文件"N".
45
46 M>&N
47      # "M"是一个文件描述符, 如果没有明确指定的话默认为1.
48      # "N"是另一个文件描述符.
49
50 =====
51      # 重定向stdout, 一次一行.
52 LOGFILE=script.log
53
54 echo "This statement is sent to the log file, \"\$LOGFILE\"." 1>\$LOGFILE
55 echo "This statement is appended to \"\$LOGFILE\"." 1>>\$LOGFILE
56 echo "This statement is also appended to \"\$LOGFILE\"." 1>>\$LOGFILE
57 echo "This statement is echoed to stdout, \
58         and will not appear in \"\$LOGFILE\"."
59      # 每行过后, 这些重定向命令会自动"reset".
60
61
62
63      # 重定向stderr, 一次一行.
64 ERRORFILE=script.errors
65
66 bad_command1 2>\$ERRORFILE      # Error message sent to \$ERRORFILE.
67 bad_command2 2>>\$ERRORFILE    # Error message appended to \$ERRORFILE.
68 bad_command3                  # Error message echoed to stderr,
69                               #+ and does not appear in \$ERRORFILE.

```

```
70 # 每行过后，这些重定向命令也会自动"reset".
71 =====
72
73
74
75 2>&1
76 # 重定向stderr到stdout.
77 # 将错误消息的输出，发送到与标准输出所指向的地方.
78
79 i>&j
80 # 重定向文件描述符i到j.
81 # 指向i文件的所有输出都发送到j.
82
83 >&j
84 # 默认的，重定向文件描述符1(stdout)到j.
85 # 所有传递到stdout的输出都送到j中去.
86
87 0< FILENAME
88 < FILENAME
89 # 从文件中接受输入.
90 # 与">"是成对命令，并且通常都是结合使用.
91 #
92 # grep search-word <filename
93
94 [j]<>filename
95 # 为了读写"filename"，把文件"filename"打开，并且将文件描述符"j"分配给它.
96 # 如果文件"filename"不存在，那么就创建它.
97 # 如果文件描述符"j"没指定，那默认是fd 0, stdin.
98 #
99 # 这种应用通常是为了写到一个文件中指定的地方.
100 echo 1234567890 > File      # 写字符串到"File".
101 exec 3<> File            # 打开"File"并且将fd 3分配给它.
102 read -n 4 <&3              # 只读取4个字符.
103 echo -n . >&3             # 写一个小数点.
104 exec 3>&-
105 cat File                  # ==> 1234.67890
106 # 随机访问.
107
108 |
109 # 管道.
110 # 通用目的处理和命令链工具.
111 # 与">"，很相似，但是实际上更通用.
112 # 对于想将命令，脚本，文件和程序串连起来的时候很有用.
113 cat *.txt | sort | uniq > result-file
114 # 对所有.txt文件的输出进行排序，并且删除重复行.
```

```
115 # 最后将结果保存到"result-file"中.
```

可以将输入输出重定向和(或)管道的多个实例结合到一起写在同一行上.

```
1 command < input-file > output-file  
2  
3 command1 | command2 | command3 > output-file
```

请参考[例子12-28](#)和[例子A-15](#).

可以将多个输出流重定向到一个文件上.

```
1 ls -yz >> command.log 2>&1  
2 # 将错误选项"yz"的结果放到文件"command.log"中.  
3 # 因为stderr被重定向到这个文件中,  
4 #+ 所有的错误消息也就都指向那里了.  
5  
6 # 注意, 下边这个例子就不会给出相同的结果.  
7 ls -yz 2>&1 >> command.log  
8 # 输出一个错误消息, 但是并不写到文件中.  
9  
10 # 如果将stdout和stderr都重定向,  
11 #+ 命令的顺序会有些不同.
```

关闭文件描述符n<&- 关闭输入文件描述符n.

0<&-, <&-

关闭stdin. n>&-

关闭输出文件描述符n.

1>&-, >&-

关闭stdout.

子进程继承了打开的文件描述符. 这就是为什么管道可以工作. 如果想阻止fd被继承, 那么可以关掉它.

```
1 # 只将stderr重定到一个管道.  
2  
3 exec 3>&1 # 保存当前stdout的"值".  
4 ls -l 2>&1 >&3 3>&- | grep bad 3>&- # 对'grep'关闭fd 3(但不关闭'ls').  
5 #  
6 exec 3>&- # 对于剩余的脚本来说, 关闭它.  
7  
8 # 感谢, S.C.
```

如果想了解关于I/O重定向更多的细节, 请参考[Appendix E](#).

注意事项

1. 一个文件描述符说白了就是文件系统为了跟踪这个打开的文件而分配给它的一个数字. 也可以将其理解为文件指针的一个简单版本. 与C语言中文件句柄的概念很相似.
2. 使用文件描述符5可能会引起问题. 当Bash使用exec创建一个子进程的时候, 子进程会继承fd5(参考Chet Ramey的归档e-mail, SUBJECT: RE: File descriptor 5 is held open). 最好还是不要去招惹这个特定的fd.

1 使用exec

exec <filename命令会将stdin重定向到文件中。从这句开始，所有的stdin就都来自于这个文件了，而不是标准输入(通常都是键盘输入)。这样就提供了一种按行读取文件的方法，并且可以使用sed和/或awk来对每一行进行分析。

例子16-1. 使用exec重定向stdin

```
1 #!/bin/bash
2 # 使用'exec'重定向stdin.
3
4
5 exec 6<&0          # 将文件描述符#6与stdin链接起来.
6           # 保存stdin.
7
8 exec < data-file    # stdin被文件"data-file"所代替.
9
10 read a1            # 读取文件"data-file"的第一行.
11 read a2            # 读取文件"data-file"的第二行.
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 # 现在将stdin从fd #6中恢复，因为刚才我们把stdin重定向到#6了，
23 #+ 然后关闭fd #6 ( 6<&- )，好让这个描述符继续被其他进程所使用。
24 #
25 # <&6 6<&-    这么做也可以。
26
27 echo -n "Enter data "
28 read b1  # 现在"read"已经恢复正常了，就是能够正常的从stdin中读取.
29 echo "Input read from stdin."
30 echo -----
31 echo "b1 = $b1"
32
33 echo
34
35 exit 0
```

同样的，exec >filename命令将会把stdout重定向到一个指定的文件中。这样所有命令的输出就都会发送到那个指定的文件，而不是stdout。

►: exec N > filename会影响整个脚本或当前shell. 对于这个指定PID的脚本或shell来说, 从这句命令执行之后, 就会重定向到这个文件中, 然而. . .

N > filename只会影响新fork出来的进程, 而不会影响整个脚本或shell. not the entire script or shell.

感谢你, Ahmed Darwish, 指出这个问题.

例子16-2. 使用exec来重定向stdout

```
1 #!/bin/bash
2 # reassign-stdout.sh
3
4 LOGFILE=logfile.txt
5
6 exec 6>&1          # 将fd #6与stdout链接起来.
7          # 保存stdout.
8
9 exec > $LOGFILE      # stdout就被文件"logfile.txt"所代替了.
10
11 # ----- #
12 # 在这块中所有命令的输出都会发送到文件$LOGFILE中.
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>&-      # 恢复stdout, 然后关闭文件描述符#6.
30
31 echo
32 echo "== stdout now restored to default == "
33 echo
34 ls -al
35 echo
36
37 exit 0
```

例子16-3. 使用exec在同一个脚本中重定向stdin和stdout

```
1 #!/bin/bash
2 # upperconv.sh
3 # 将一个指定的输入文件转换为大写.
4
5 E_FILE_ACCESS=70
6 E_WRONG_ARGS=71
7
8 if [ ! -r "$1" ]      # 判断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                      # 即使输入文件($1)没被指定
14                      #+# 也还是会以相同的错误退出(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          # 将会从输入文件中读取.
26
27 exec 7>&1
28 exec > $2          # 将写到输出文件中.
29                      # 假设输出文件是可写的(添加检查?).
30
31 # -----
32     cat - | tr a-z A-Z  # 转换为大写.
33 # ^^^^^^          # 从stdin中读取.
34 #           ~~~~~~  # 写到stdout上.
35 # 然而, stdin和stdout都被重定向了.
36 # -----
37
38 exec 1>&7 7>&-      # 恢复stdout.
39 exec 0<&4 4<&-      # 恢复stdin.
40
41 # 恢复之后, 下边这行代码将会如预期的一样打印到stdout上.
42 echo "File \"$1\" written to \"$2\" as uppercase conversion."
43
```

```
44 exit 0
```

I/O重定向是一种避免可怕的子shell中不可访问变量问题的方法.

例子16-4. 避免子shell

```
1 #!/bin/bash
2 # avoid-subshell.sh
3 # 由Matthew Walker所提出的建议.
4
5 Lines=0
6
7 echo
8
9 cat myfile.txt | while read line; # (译者注: 管道会产生子shell)
10          do {
11              echo $line
12              (( Lines++ )); # 增加这个变量的值
13                      #+# 但是外部循环却不能访问.
14                      # 子shell问题.
15          }
16          done
17
18 echo "Number of lines read = $Lines"      # 0
19                                     # 错误!
20
21 echo "-----"
22
23
24 exec 3<> myfile.txt
25 while read line <&3
26 do {
27     echo "$line"
28     (( Lines++ ));           # 增加这个变量的值
29                         #+# 现在外部循环就可以访问了.
30                         # 没有子shell, 现在就没问题了.
31 }
32 done
33 exec 3>&-
34
35 echo "Number of lines read = $Lines"      # 8
36
37 echo
38
39 exit 0
40
41 # 下边这些行是这个脚本的结果, 脚本是不会走到这里的.
```

```
42
43 $ cat myfile.txt
44
45 Line 1.
46 Line 2.
47 Line 3.
48 Line 4.
49 Line 5.
50 Line 6.
51 Line 7.
52 Line 8.
```

2 代码块重定向

象while, until, 和for循环代码块, 甚至if/then测试结构的代码块, 都可以对stdin进行重定向. 即使函数也可以使用这种重定向方式(请参考例子23-11). 要想做到这些, 都要依靠代码块结尾的;操作符.

例子16-5. while循环的重定向

```
1 #!/bin/bash
2 # redir2.sh
3
4 if [ -z "$1" ]
5 then
6     Filename=names.data      # 如果没有指定文件名, 则使用这个默认值.
7 else
8     Filename=$1
9 fi
10 #+ Filename=${1:-names.data}
11 # 这句可代替上面的测试(参数替换).
12
13 count=0
14
15 echo
16
17 while [ "$name" != Smith ]  # 为什么变量$name要用引号?
18 do
19     read name          # 从$Filename文件中读取输入, 而不是在stdin中读取输入.
20     echo $name
21     let "count += 1"
22 done <"$Filename"      # 重定向stdin到文件$Filename.
23 #      ^^^^^^^^^^
24
25 echo; echo "$count names read"; echo
26
27 exit 0
28
29 # 注意在一些比较老的shell脚本编程语言中,
30 #+ 重定向的循环是放在子shell里运行的.
31 # 因此, $count 值返回后会是 0, 此值是在循环开始前的初始值.
32 # *如果可能的话*, 尽量避免在Bash或ksh中使用子shell,
33 #+ 所以这个脚本能够正确的运行.
34 # (多谢Heiner Steven指出这个问题.)
35
36 # 然而 . .
37 # Bash有时还是会*在一个使用管道的"while-read"循环中启动一个子shell,
38 #+ 与重定向的"while"循环还是有区别的.
```

```

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
48 #+ 给出上面的代码片段来演示此问题.
49 # 同时,感谢, Brian Onn, 修正了一个注释错误.

```

例子16-6. 重定向while循环的另一种形式

```

1#!/bin/bash
2
3# 这是上个脚本的另一个版本.
4
5# Heiner Steven建议,
6#+ 为了避免重定向循环运行在子shell中(老版本的shell会这么做),
7#+ 最好让重定向循环运行在当前工作区内,
8#+ 这样的话, 需要提前进行文件描述符重定向,
9#+ 因为变量如果在(子shell上运行的)循环中被修改的话,
10#+ 循环结束后并不会保存修改后的值.
11
12if [ -z "$1" ]
13then
14    Filename=names.data      # 如果没有指定文件名则使用默认值.
15else
16    Filename=$1
17fi
18
19
20exec 3<&0                  # 将stdin保存到文件描述符3.
21exec 0<"$Filename"          # 重定向标准输入.
22
23count=0
24echo
25
26
27while [ "$name" != Smith ]
28do
29    read name              # 从stdin(现在已经是$Filename了)中读取.
30    echo $name
31    let "count += 1"
32done                      # 从文件$Filename中循环读取

```

```

33                         #+# 因为文件(译者注: 指默认文件, 在本节最后)有20行.
34
35 # 这个脚本原先在"while"循环的结尾还有一句:
36 #+      done <"$Filename"
37 # 练习:
38 # 为什么不需要这句了?
39
40
41 exec 0<&3                  # 恢复保存的stdin.
42 exec 3<&-                   # 关闭临时文件描述符3.
43
44 echo; echo "$count names read"; echo
45
46 exit 0

```

例子16-7. 重定向until循环

```

1#!/bin/bash
2# 和前面的例子相同, 但使用的是"until"循环.
3
4if [ -z "$1" ]
5then
6    Filename=names.data          # 如果没有指定文件名那就使用默认值.
7else
8    Filename=$1
9fi
10
11# while [ "$name" != Smith ]
12until [ "$name" = Smith ]      # 把!=改为=.
13do
14    read name                  # 从$Filename中读取, 而不是从stdin中读取.
15    echo $name
16done <"$Filename"             # 重定向stdin到文件$Filename.
17#      ^^^^^^^^^^
18
19# 结果和前面例子的"while"循环相同.
20
21exit 0

```

例子16-8. 重定向for循环

```

1#!/bin/bash
2
3if [ -z "$1" ]
4then
5    Filename=names.data          # 如果没有指定文件名就使用默认值.
6else
7    Filename=$1

```

```

8 fi
9
10 line_count='wc $Filename | awk '{ print $1 }''
11 # 目标文件的行数.
12 #
13 # 此处的代码太过做作，并且写得很难看，
14 #+ 但至少展示了"for"循环的stdin可以重定向...
15 #+ 当然，你得足够聪明，才能看得出来.
16 #
17 # 更简洁的写法是      line_count=$(wc -l < "$Filename")
18
19
20 for name in `seq $line_count` # "seq"打印出数字序列.
21 # while [ "$name" != Smith ] -- 比"while"循环更复杂 --
22 do
23     read name          # 从$Filename中，而非从stdin中读取.
24     echo $name
25     if [ "$name" = Smith ]      # 因为用for循环，所以需要这个多余测试.
26     then
27         break
28     fi
29 done <"$Filename"           # 重定向stdin到文件$Filename.
30 # ~~~~~
31
32 exit 0

```

我们也可以修改前面的例子使其能重定向循环的标准输出.

例子16-9. 重定向for循环(stdin和stdout都进行重定向)

```

1#!/bin/bash
2
3 if [ -z "$1" ]
4 then
5     Filename=names.data      # 如果没有指定文件名，则使用默认值.
6 else
7     Filename=$1
8 fi
9
10 Savefile=$Filename.new      # 保存最终结果的文件名.
11 FinalName=Jonah            # 终止"read"时的名称.
12
13 line_count='wc $Filename | awk '{ print $1 }'' # 目标文件的行数.
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" > "$Savefile"      # 重定向stdin到文件$Filename,
25 #      ~~~~~                         并且将它保存到备份文件中.
26
27 exit 0

```

例子16-10. 重定向if/then测试结构

```

1#!/bin/bash
2
3 if [ -z "$1" ]
4 then
5     Filename=names.data    # 如果文件名没有指定，使用默认值.
6 else
7     Filename=$1
8 fi
9
10 TRUE=1
11
12 if [ "$TRUE" ]           # if true    和   if :    都可以.
13 then
14     read name
15     echo $name
16 fi <"$Filename"
17 # ~~~~~
18
19 # 只读取了文件的第一行.
20 # An "if/then"测试结构不能自动地反复地执行，除非把它们嵌到循环里.
21
22 exit 0

```

例子16-11. 用于上面例子的"names.data"数据文件

```

1 Aristotle
2 Belisarius
3 Capablanca
4 Euler
5 Goethe
6 Hamurabi
7 Jonah
8 Laplace
9 Maroczy
10 Purcell

```

```
11 Schmidt
12 Semmelweiss
13 Smith
14 Turing
15 Venn
16 Wilson
17 Znosko-Borowski
18
19 # 此数据文件用于:
20 #+ "redir2.sh", "redir3.sh", "redir4.sh", "redir4a.sh", "redir5.sh".
```

重定向代码块的stdout, 与”将代码块的输出保存到文件中”具有相同的效果. 请参考[例子3-2](#).
[here document](#) 是重定向代码块的一个特例.

3 重定向的应用

巧妙地运用I/O重定向，能够解析和粘合命令输出的各个片断(请参考[例子11-7](#)). 这样就可以产生报告与日志文件.

例子16-12. 事件纪录

```
1 #!/bin/bash
2 # logevents.sh, 由Stephane Chazelas所编写 .
3
4 # 把事件记录在一个文件中.
5 # 必须以root身份运行 (这样才有权限访问/var/log) .
6
7 ROOT_UID=0      # 只有$UID值为0的用户才具有root权限.
8 E_NOTROOT=67    # 非root用户的退出错误.
9
10
11 if [ "$UID" -ne "$ROOT_UID" ]
12 then
13     echo "Must be root to run this script."
14     exit $E_NOTROOT
15 fi
16
17
18 FD_DEBUG1=3
19 FD_DEBUG2=4
20 FD_DEBUG3=5
21
22 # 去掉下边两行注释中的一行, 来激活脚本.
23 # LOG_EVENTS=1
24 # LOG_VARS=1
25
26
27 log() # 把时间和日期写入日志文件.
28 {
29 echo "$(date) $*" >&7      # 这会把日期*附加*到文件中.
30                      # 参考下边的代码.
31 }
32
33
34
35 case $LOG_LEVEL in
36   1) exec 3>&2          4> /dev/null 5> /dev/null;;
37   2) exec 3>&2          4>&2          5> /dev/null;;
38   3) exec 3>&2          4>&2          5>&2;;
39   *) exec 3> /dev/null 4> /dev/null 5> /dev/null;;
```

```
40 esac
41
42 FD_LOGVARS=6
43 if [[ $LOG_VARS ]]
44 then exec 6>> /var/log/vars.log
45 else exec 6> /dev/null          # 丢弃输出.
46 fi
47
48 FD_LOGEVENTS=7
49 if [[ $LOG_EVENTS ]]
50 then
51     # then exec 7 >(exec gawk '{print strftime(), $0}' >> /var/log/event.log)
52     # 上面这行不能在2.04版本的Bash上运行.
53     exec 7>> /var/log/event.log      # 附加到"event.log".
54     log                           # 记录日期与时间.
55 else exec 7> /dev/null          # 丢弃输出.
56 fi
57
58 echo "DEBUG3: beginning" >&${FD_DEBUG3}
59
60 ls -l >&5 2>&4                # command1 >&5 2>&4
61
62 echo "Done"                      # command2
63
64 echo "sending mail" >&${FD_LOGEVENTS} # 将字符串"Sending mail"写到文件描述符#7.
65
66
67 exit 0
```

第17章 Here Document

Here and now, boys.

— Aldous Huxley, "Island"

一个here document就是一段带有特殊目的的代码段. 它使用I/O重定向的形式将一个命令序列传递到一个交互程序或者命令中, 比如[ftp](#), [cat](#), 或者ex文本编辑器.

```
1 COMMAND <<InputComesFromHERE
2 ...
3 InputComesFromHERE
```

limit string用来界定命令序列的范围(译者注: 两个相同的limit string之间就是命令序列). 特殊符号`<<`用来标识limit string. 这个符号的作用就是将文件的输出重定向到程序或命令的stdin中. 与interactive-program `|` command-file很相似, 其中command-file包含:

```
1 command #1
2 command #2
3 ...
```

而here document看上去是下面这个样子:

```
1 #!/bin/bash
2 interactive-program <<LimitString
3 command #1
4 command #2
5 ...
6 LimitString
```

选择一个名字非常诡异limit string能够有效的避免命令列表与limit string重名的问题. 注意, 某些情况下, 把here document用在非交互工具或命令中, 也会取得非常好的效果, 比如, [wall](#).

例子17-1. 广播: 将消息发送给每个登陆的用户

```
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 # 附加的消息文本放在这里.
7 # 注意: 'wall'命令会把注释行也打印出来.
8 zzz23EndOfMessagezzz23
9
10 # 当然, 更有效率的做法是:
11 #         wall <message-file
12 # 然而, 将消息模版嵌入到脚本中
```

```
13 #+ 只是一种"小吃店"(译者注: 方便但是不卫生)的做法, 而且这种做法是一次性的.  
14  
15 exit 0
```

对于某些看上去不太可能的工具, 比如vi, 也能够使用here document.

例子17-2. 虚拟文件: 创建一个2行的虚拟文件

```
1#!/bin/bash  
2  
3# 用非交互的方式来使用'vi'编辑一个文件.  
4# 模仿'sed'.  
5  
6E_BADARGS=65  
7  
8if [ -z "$1" ]  
9then  
10    echo "Usage: 'basename $0' filename"  
11    exit $E_BADARGS  
12fi  
13  
14TARGETFILE=$1  
15  
16# 在文件中插入两行, 然后保存.  
17#-----Begin here document-----#  
18vi $TARGETFILE <<x23LimitStringx23  
19i  
20This is line 1 of the example file.  
21This is line 2 of the example file.  
22^ [  
23ZZ  
24x23LimitStringx23  
25#-----End here document-----#  
26  
27# 注意上边^[是一个转义符, 键入Ctrl+v <Esc>就行,  
28#+ 事实上它是<Esc>键;.  
29  
30# Bram Moolenaar指出这种方法不能使用在'vim'上,  
31# (译者注: Bram Moolenaar是vim作者)  
32#+ 因为可能会存在终端相互影响的问题.  
33  
34exit 0
```

上边的脚本也可以不用vi而改用ex来实现, here document包含ex命令列表的形式足以形成自己的类别了, 称为ex script.

```
1#!/bin/bash  
2# 把所有后缀为".txt"文件  
3#+ 中的"Smith"都替换成"Jones".
```

```

4
5 ORIGINAL=Smith
6 REPLACEMENT=Jones
7
8 for word in $(fgrep -l $ORIGINAL *.txt)
9 do
10   # -----
11   ex $word <<EOF
12   :%s/$ORIGINAL/$REPLACEMENT/g
13   :wq
14 EOF
15   # :%s是"ex"的替换命令. (译者注: 与vi和vim的基本命令相同)
16   # :wq是保存并退出的意思.
17   # -----
18 done

```

与”ex script”相似的是cat script.

例子17-3. 使用cat的多行消息

```

1#!/bin/bash
2
3# 'echo'对于打印单行消息来说是非常好用的,
4#+ 但是在打印消息块时可能就有点问题了.
5# 'cat' here document可以解决这个限制.
6
7cat <<End-of-message
-----
9This is line 1 of the message.
10This is line 2 of the message.
11This is line 3 of the message.
12This is line 4 of the message.
13This is the last line of the message.
14-----
15End-of-message
16
17# 用下边这行代替上边的第7行,
18#+ cat > $Newfile <<End-of-message
19#+ ^^^^^^^^^^
20#+ 那么就会把输出写到文件$Newfile中, 而不是stdout.
21
22exit 0
23
24
25#-----
26# 下边的代码不会运行, 因为上边有"exit 0".
27

```

```

28 # S.C. 指出下边代码也能够达到相同目的.
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, 文本中可能不允许包含双引号, 除非它们被转义.

```

-选项用来标记here document的limit string (jj-LimitString), 可以抑制输出时前边的tab(不是空格). 这么做可以增加一个脚本的可读性.

例子17-4. 带有抑制tab功能的多行消息

```

1#!/bin/bash
2# 与之前的例子相同, 但是...
3
4# - 选项对于here documtment来说,
5#+ <<-可以抑制文档体前边的tab,
6#+ 而*不*是空格.
7
8cat <<-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.
14ENDOFMESSAGE
15# 脚本在输出的时候左边将被刷掉.
16# 就是说每行前边的tab将不会显示.
17
18# 上边5行"消息"的前边都是tab, 而不是空格.
19# 空格是不受<<-影响的.
20
21# 注意, 这个选项对于*嵌在*中间的tab没作用.
22
23exit 0

```

here document支持参数和命令替换. 所以也可以给here document的消息体传递不同的参数, 这样相应的也会修改输出.

例子17-5. 使用参数替换的here document

```

1#!/bin/bash
2# 一个使用'cat'命令的here document, 使用了参数替换.
3
4# 不传命令行参数给它, ./scriptname
5# 传一个命令行参数给它, ./scriptname Mortimer
6# 传一个包含2个单词(用引号括起来)的命令行参数给它,

```

```

7 #                               ./scriptname "Mortimer Jones"
8
9 CMDLINEPARAM=1      # 所期望的最少的命令行参数个数.
10
11 if [ $# -ge $CMDLINEPARAM ]
12 then
13     NAME=$1          # 如果命令行参数超过1个,
14                 #+# 那么就只取第一个参数.
15 else
16     NAME="John Doe" # 默认情况下, 如果没有命令行参数的话.
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 # This comment shows up in the output (why?).
28
29 Endofmessage
30
31 # 注意上边的空行也打印输出,
32 # 而上边那行"注释"当然也会打印到输出.
33 # (译者注: 这就是为什么不翻译那行注释的原因, 尽量保持代码的原样)
34 exit 0

```

例子17-6. 上传一个文件对到“Sunsite”的incoming目录

```

1#!/bin/bash
2# upload.sh
3
4# 上传这一对文件(Filename.lsm, Filename.tar.gz)
5#+ 到Sunsite/UNC (ibiblio.org)的incoming目录.
6# Filename.tar.gz是自身的tar包.
7# Filename.lsm是描述文件.
8# Sunsite需要"lsm"文件, 否则就拒绝上传.
9
10
11 E_ARGERROR=65
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`          # 从文件名中去掉目录字符串.
21
22 Server="ibiblio.org"
23 Directory="/incoming/Linux"
24 # 在这里也不一定非得将上边的参数写死在这个脚本中,
25 #+ 可以使用命令行参数的方法来替换.
26
27 Password="your.e-mail.address" # 可以修改成相匹配的密码.
28
29 ftp -n $Server <<End-Of-Session
30 # -n选项禁用自动登录.
31
32 user anonymous "$Password"
33 binary
34 bell                         # 在每个文件传输后, 响铃.
35 cd $Directory
36 put "$Filename.lsm"
37 put "$Filename.tar.gz"
38 bye
39 End-Of-Session
40
41 exit 0

```

在here document的开头, 引用或转义"limit string", 会使得here document消息体中的参数替换被禁用.

例子17-7. 关闭参数替换

```

1#!/bin/bash
2# 一个使用'cat'的here document, 但是禁用了参数替换.
3
4NAME="John Doe"
5RESPONDENT="the author of this fine script"
6
7cat <<'Endofmessage'
8
9Hello, there, $NAME.
10Greetings to you, $NAME, from $RESPONDENT.
11
12Endofmessage
13
14# 如果"limit string"被引用或转义的话, 那么就禁用了参数替换.
15# 下边的两种方式具有相同的效果.

```

```
16 #  cat <<"Endofmessage"
17 #  cat <<\Endofmessage
18
19 exit 0
```

禁用了参数替换后, 将允许输出文本本身(译者注: 就是未转义的原文). 如果你想产生脚本甚至是程序代码的话, 那么可以使用这种办法.

例子17-8. 生成另外一个脚本的脚本

```
1#!/bin/bash
2# generate-script.sh
3# 这个脚本的诞生基于Albert Reiner的一个主意.
4
5OUTFILE=generated.sh          # 所产生文件的名字.
6
7
8# -----
9# 'Here document'包含了需要产生的脚本的代码.
10(
11cat <<'EOF'
12#!/bin/bash
13
14echo "This is a generated shell script."
15# Note that since we are inside a subshell,
16#+ we can't access variables in the "outside" script.
17
18echo "Generated file will be named: $OUTFILE"
19# Above line will not work as normally expected
20#+ because parameter expansion has been disabled.
21# Instead, the result is literal output.
22
23a=7
24b=3
25
26let "c = $a * $b"
27echo "c = $c"
28
29exit 0
30EOF
31) > $OUTFILE
32# -----
33
34# 将'limit string'引用起来将会阻止上边
35#+ here document消息体中的变量扩展.
36# 这会使得输出文件中的内容保持here document消息体中的原文.
37
```

```
38 if [ -f "$OUTFILE" ]
39 then
40     chmod 755 $OUTFILE
41     # 让所产生的文件具有可执行权限.
42 else
43     echo "Problem in creating file: \\"$OUTFILE\""
44 fi
45
46 # 这个方法也可以用来产生
47 #+ C程序代码, Perl程序代码, Python程序代码, makefile,
48 #+ 和其他的一些类似的代码.
49 # (译者注: 中间一段没译的注释将会被here document打印出来)
50 exit 0
```

也可以将here document的输出保存到变量中.

```
1 variable=$(cat <<SETVAR
2 This variable
3 runs over multiple lines.
4 SETVAR)
5
6 echo "$variable"
```

A here document can supply input to a function in the same script.

例子17-9. Here document与函数

```
1#!/bin/bash
2# here-function.sh
3
4GetPersonalData(){}
5{
6    read firstname
7    read lastname
8    read address
9    read city
10   read state
11   read zipcode
12 } # 这个函数看起来就是一个交互函数, 但是...
13
14
15 # 给上边的函数提供输入.
16 GetPersonalData <<RECORD001
17 Bozo
18 Bozeman
19 2726 Nondescript Dr.
20 Baltimore
21 MD
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
```

也可以这么使用:(冒号), 做一个假命令来从一个here document中接收输出. 这么做事实上就是创建了一个”匿名”的here document.

例子17-10. ”匿名”的here Document

```
1#!/bin/bash
2
3: <<TESTVARIABLES
4 ${HOSTNAME?}${USER?}${MAIL?} # 如果其中某个变量没被设置, 那么就打印错误信息.
5 TESTVARIABLES
6
7 exit 0
```

►: 上边所示技术的一种变化, 可以用来”注释”掉代码块.

例子17-11. 注释掉一段代码块

```
1#!/bin/bash
2# commentblock.sh
3
4: <<COMMENTBLOCK
5echo "This line will not echo."
6This is a comment line missing the "#" prefix.
7This is another comment line missing the "#" prefix.
8
9*&*@!!++=
10The above line will cause no error message,
11because the Bash interpreter will ignore it.
12COMMENTBLOCK
13
14echo "Exit value of above \"COMMENTBLOCK\" is $?." # 0
15# 这里将不会显示任何错误.
16
17
18# 上边的这种技术当然也可以用来注释掉
19#+ 一段正在使用的代码, 如果你有某些特定调试要求的话.
20#+ 这比在每行前边都敲入#"来得方便的多,
21#+ 而且如果你想恢复的话, 还得将添加上的#"删除掉.
22
```

```
23 : <<DEBUGXXX
24 for file in *
25 do
26   cat "$file"
27 done
28 DEBUGXXX
29
30 exit 0
```

►: 关于这种小技巧的另一个应用就是能够产生“自文档化(self-documenting)”的脚本.

例子17-12. 一个自文档化(self-documenting)的脚本

```
1#!/bin/bash
2# self-document.sh: 自文档化(self-documenting)的脚本
3# 修改于"colm.sh".
4
5DOC_REQUEST=70
6
7if [ "$1" = "-h" -o "$1" = "--help" ]      # 请求帮助.
8then
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
15List the statistics of a specified directory in tabular format.
16-----
17The command line parameter gives the directory to be listed.
18If no directory specified or directory specified cannot be read,
19then list the current working directory.
20
21DOCUMENTATIONXX
22
23if [ -z "$1" -o ! -r "$1" ]
24then
25  directory=.
26else
27  directory="$1"
28fi
29
30echo "Listing of \"$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
34exit 0
```

使用[cat脚本](#)也能够完成相同的目的.

```
1 DOC_REQUEST=70
2
3 if [ "$1" = "-h" -o "$1" = "--help" ]      # 请求帮助.
4 then                                         # 使用"cat脚本" . .
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
```

请参考[例子A-28](#)可以看到更多关于“自文档化”脚本的好例子.

►: Here document创建临时文件, 但是这些文件将在打开后被删除, 并且不能够被任何其他进程所访问.

```
1 bash$ bash -c 'lsof -a -p $$ -d0' << EOF
2 > EOF
3 lsof      1213 bozo      Or      REG      3,5      0 30386 /tmp/t1213-0-sh (deleted)
```

►:某些工具是不能放入here document中运行的.

►: 结尾的limit string, 就是here document最后一行的limit string, 必须从第一个字符开始. 它的前面不能够有任何前置的空白. 而在这个limit string后边的空白也会引起异常. 空白将会阻止limit string的识别. (译者注: 下边这个脚本由于结束limit string的问题, 造成脚本无法结束, 所有内容全部被打印出来, 所以注释就不译了, 保持这个例子脚本的原样.)

```
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 ####Indented limit string. Error! This script will not behave as expected.
11
12 echo "-----"
13
14 # These comments are outside the 'here document',
15 #+ and should not echo.
16
17 echo "Outside the here document."
```

```
18
19 exit 0
20
21 echo "This line had better not echo." # Follows an 'exit' command.
```

对于那些使用”here document”，并且非常复杂的任务，最好考虑使用expect脚本语言，这种语言就是为了达到向交互程序添加输入的目的而量身定做的。

1 Here String

here string可以看成是here document的一种定制形式. 除了COMMAND <<<\$WORD, 就什么都没有了, \$WORD将被扩展并且被送入COMMAND的stdin中.

```
1 1 String="This is a string of words."
2 2
3 3 read -r -a Words <<< "$String"
4 4 # "read"命令的-a选项
5 5 #+ 将会把结果值按顺序的分配给数组中的每一项.
6 6
7 7 echo "First word in String is: ${Words[0]}" # This
8 8 echo "Second word in String is: ${Words[1]}" # is
9 9 echo "Third word in String is: ${Words[2]}" # a
10 10 echo "Fourth word in String is: ${Words[3]}" # string
11 11 echo "Fifth word in String is: ${Words[4]}" # of
12 12 echo "Sixth word in String is: ${Words[5]}" # words.
13 13 echo "Seventh word in String is: ${Words[6]}" # (null)
14 14 # $String的结尾.
15 15
16 16 # 感谢, Francisco Lobo的这个建议.
```

例子17-13. 在一个文件的开头添加文本

```
1 1#!/bin/bash
2 2 # prepend.sh: 在文件的开头添加文本.
3 3 #
4 4 # Kenny Stauffer所捐助的脚本例子,
5 5 #+ 本文作者对这个脚本进行了少量修改.
6 6
7 7
8 8 E_NOSUCHFILE=65
9 9
10 10 read -p "File: " file # 'read'命令的-p参数用来显示提示符.
11 11 if [ ! -e "$file" ]
12 12 then # 如果这个文件不存在, 那就进来.
13 13 echo "File $file not found."
14 14 exit $E_NOSUCHFILE
15 15 fi
16 16
17 17 read -p "Title: " title
18 18 cat - $file <<<$title > $file.new
19 19
20 20 echo "Modified file is $file.new"
21 21
22 22 exit 0
```

```
23
24 # 下边是'man bash'中的一段:
25 # Here String
26 #   here document的一种变形, 形式如下:
27 #
28 #           <<<word
29 #
30 #   word被扩展并且被提供到command的标准输入中.
```

例子17-14. 分析一个邮箱

```
1#!/bin/bash
2# 由Francisco Lobo所提供的脚本,
3#+ 本文作者进行了少量修改和注释.
4# 并且经过授权, 可以使用在本书中.(感谢你!)
5
6# 这个脚本不能运行于比Bash version 3.0更低的版本中.
7
8
9E_MISSING_ARG=67
10if [ -z "$1" ]
11then
12  echo "Usage: $0 mailbox-file"
13  exit $E_MISSING_ARG
14fi
15
16mbox_grep() # 分析邮箱文件.
17{
18  declare -i body=0 match=0
19  declare -a date sender
20  declare mail header value
21
22
23  while IFS= read -r mail
24#      ^^^^          重新设置$IFS.
25# 否则"read"会从它的输入中截去开头和结尾的空格.
26
27  do
28    if [[ $mail =~ ^From\ ]]\# 匹配消息中的"From"域.
29    then
30      (( body = 0 ))          # 取消("Zero out"俚语)变量.
31      (( match = 0 ))
32      unset date
33
34    elif (( body ))
35    then
```

```

36      (( match ))
37      # echo "$mail"
38      # 如果你想显示整个消息体的话，那么就打开上面的注释行。
39
40      elif [[ $mail ]]; then
41          IFS=: read -r header value <<< "$mail"
42          #
43          ^^^ "here string"
44
45          case "$header" in
46              [Ff] [Rr] [Oo] [Mm] ) [[ $value =~ "$2" ]] && (( match++ )) ;;
47              # 匹配"From"行。
48              [Dd] [Aa] [Tt] [Ee] ) read -r -a date <<< "$value" ;;
49              #
50              ^^^
51              # 匹配"Date"行。
52              [Rr] [Ee] [Cc] [Ee] [Ii] [Vv] [Ee] [Dd] ) read -r -a sender <<< "$value" ;;
53              #
54              ^^^
53          esac
54
55      else
56          (( body++ ))
57          (( match )) &&
58          echo "MESSAGE ${date:+of: ${date[*]}} "
59          # 整个$date数组
60          echo "IP address of sender: ${sender[1]}"
61          # "Received"行的第二个域
62
63      fi
64
65
66      done < "$1" # 将文件的stdout重定向到循环中。
67 }
68
69
70 mbox_grep "$1" # 将邮箱文件发送到函数中。
71
72 exit $?
73
74 # 练习：
75 # -----
76 # 1) 拆开上面的这个函数，把它分成多个函数，
77 #+ 这样可以提高代码的可读性。
78 # 2) 对这个脚本添加额外的分析，可以分析不同的关键字。
79
80

```

```
81
82 $ mailbox_grep.sh scam_mail
83 --> MESSAGE of Thu, 5 Jan 2006 08:00:56 -0500 (EST)
84 --> IP address of sender: 196.3.62.4
```

练习: 找出here string的其他用法.

第18章 休息片刻

这片刻的休息可以让读者放松一下，并且学习了这么多东西，读者也可以发出会心的微笑了。

Linux同志们，向你们致敬！你正在阅读的这些东西，将会给你们带来好运。把这份文档发给你的10个朋友。在拷贝这份文档之前，在信的结尾加上一个100行的Bash脚本，然后发送给列表上的第一个人。最后在信的底部删除他们的名字，并把你自己的名字添加到列表的尾部。

千万不要打断这个发送的通道！并且在48小时之内发送出去。Brooklyn的Wilfred P.就因为没有成功的发送他的10个拷贝，当他第2天早上醒来，发现他变成了一个“COBOL 程序员”。而Newport News的Howard L.在一个月内才发出了他的10个拷贝，如果有足够的硬件，一个月的时间足以建立一个100个节点的Beowulf cluster来玩Tuxracer了。Chicago的Amelia V.对这封信付之一笑，并且打断了这个发送通道。不久之后，她的终端爆炸了，现在，她不得不每天为MS Windows编写文档。

千万不要打断这个发送的通道！今天就把10个拷贝发送出去！

Courtesy 'NIX "fortune cookies", with some alterations and many apologies

第四部分

高级主题

内容

- 19. 正则表达式
 - 20. 子shell
 - 21. 受限shell
 - 22. 进程替换
 - 23. 函数
 - 24. 别名
 - 25. 列表结构
 - 26. 数组
 - 27. /dev和/proc
 - 28. Zero与Null
 - 29. 调试
 - 30. 选项
 - 31. 陷阱
 - 32. 脚本编程风格
 - 33. 杂项
 - 34. Bash, 版本2与版本3
-

第19章 正则表达式

本章目录

1. [一份简要的正则表达式介绍](#)
 2. [通配\(globbing\)](#)
-

为了充分发挥shell编程的威力, 你必须精通正则表达式. 脚本中经常使用的某些命令, 和工具包通常都支持正则表达式, 比如[grep](#), [expr](#), [sed](#) 和[awk](#)解释器.

1 一份简要的正则表达式介绍

正则表达式就是由一系列特殊字符组成的字符串，其中每个特殊字符都被称为元字符，这些元字符并不表示为它们字面上的含义，而会被解释为一些特定的含义。举个例子，比如引用符号，可能就是表示某人的演讲内容，同上，也可能表示为我们下面将要讲到的符号的元-含义。正则表达式其实是由普通字符和元字符共同组成的集合，这个集合用来匹配(或指定)模式。

一个正则表达式会包含下列一项或多项：

- 一个字符集。这里所指的字符集只包含普通字符，这些字符只表示它们的字面含义。正则表达式的最简单形式就是只包含字符集，而不包含元字符。
- 锚。锚指定了正则表达式所要匹配的文本在文本行中所处的位置。比如，^，和\$就是锚。
- 修饰符。它们扩大或缩小(修改)了正则表达式匹配文本的范围。修饰符包含星号，括号，和反斜杠。

正则表达式最主要的目的就是用于(RE)文本搜索与字符串操作。(译者注：以下正则表达式也会被简称为RE。) RE能够匹配单个字符或者一个字符集即，一个字符串，或者一个字符串的一部分。

- 星号”*”。用来匹配它前面字符的任意多次，包括0次。”1133*”匹配11 + 一个或多个3 + 也允许后边还有其他字符：113, 1133, 111312, 等等。
- 点。用于匹配任意一个字符，除了换行符。[\[1\]](#) ”13.” 匹配13 + 至少一个任意字符(包括空格)：1133, 11333, 但不能匹配13 (因为缺少”.”所能匹配的至少一个任意字符)。
- 脱字符号”^”。匹配行首，但是某些时候需要依赖上下文环境，在RE中，有时候也表示对一个字符集取反。
- 美元符。在RE中用来匹配行尾。”XXX\$” 匹配行尾的XXX。
”^\$” 匹配空行。
- 中括号。在RE中，将匹配中括号字符集中的某一个字符。
”[xyz]” 将会匹配字符x, y, 或z。
”[c-n]” 匹配字符c到字符n之间的任意一个字符。
”[B-Pk-y]” 匹配从B到P, 或者从k到y之间的任意一个字符。
”[a-z0-9]” 匹配任意小写字母或数字。
”[^ b-d]” 将会匹配范围在b到d之外的任意一个字符。
这就是使用^ 对字符集取反的一个实例。(就好像在某些情况下, !所表达的含义)。

将多个中括号字符集组合使用，能够匹配一般的单词或数字。”[Yy][Ee][Ss]”能够匹配yes, Yes, YES, yEs, 等等。

”[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]” 可以匹配社保码(Social Security number)。

- 反斜杠。用来转义某个特殊含义的字符，这意味着，这个特殊字符将会被解释为字面含义。”\\$”将会被解释成字符”\$”，而不是RE中匹配行尾的特殊字符。相似的，”\\”将会被解释为字符”\”。

- 转义的”尖括号”. 用于匹配单词边界. 尖括号必须被转义才含有特殊的含义, 否则它就表示尖括号的字面含义. ”\<the\>” 完整匹配单词”the”, 不会匹配”them”, ”there”, ”other”, 等等.

```

1 bash$ cat myfile
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' myfile
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\>' myfile
15 This is the only instance of line 2.

```

要想确定一个RE能否正常工作, 唯一的办法就是测试它.

```

1 TEST FILE: tstfile                      # 不匹配.
2                                     # 不匹配.
3 Run    grep "1133*"  on this file.      # 匹配.
4                                     # 不匹配.
5                                     # 不匹配.
6 This line contains the number 113.       # 匹配.
7 This line contains the number 13.          # 不匹配.
8 This line contains the number 133.         # 不匹配.
9 This line contains the number 1133.        # 匹配.
10 This line contains the number 113312.     # 匹配.
11 This line contains the number 1112.        # 不匹配.
12 This line contains the number 113312312.  # 匹配.
13 This line contains no numbers at all.     # 不匹配.

```

```

1 bash$ grep "1133*" tstfile
2 Run    grep "1133*"  on this file.      # 匹配.
3 This line contains the number 113.        # 匹配.
4 This line contains the number 1133.        # 匹配.
5 This line contains the number 113312.     # 匹配.
6 This line contains the number 113312312.  # 匹配.

```

扩展的正则表达式. 添加了一些额外的匹配字符到基本集合中. 用于`egrep`, `awk`, 和`Perl`.

- 问号. 匹配它前面的字符, 但是只能匹配1次或0次. 通常用来匹配单个字符.

- 加号. 匹配它前面的字符, 能够匹配一次或多次. 与前面讲的*号作用类似, 但是不能匹配0个字符的情况.

```

1 # GNU版本的sed和awk能够使用"+",
2 # 但是它需要被转义一下.
3
4 echo a111b | sed -ne '/a1\+b/p'
5 echo a111b | grep 'a1\+b'
6 echo a111b | gawk '/a1+b/'"
7 # 上边3句的作用相同.
8
9 # 感谢, S.C.

```

- [转义](#)”大括号”. 在转义后的大括号中加上一个数字, 这个数字就是它前面的RE所能匹配的次数.

大括号必须经过转义, 否则, 大括号仅仅表示字面含意. 这种用法并不是基本RE集合中的一部分, 仅仅是个技巧而已.

”[0-9]{5}” 精确匹配5个数字(所匹配的字符范围是0到9).

►: 使用大括号形式的RE是不能够在”经典”(非POSIX兼容)的awk版本中正常运行的. 然而, gawk命令中有一个--re-interval选项, 使用这个选项就允许使用大括号形式的RE了(无需转义).

```

1 bash$ echo 2222 | gawk --re-interval '/2{3}/'
2 2222

```

Perl与某些版本的egrep不需要转义大括号.

- 圆括号. 括起一组正则表达式. 当你想使用expr进行子字符串提取 (substring extraction)的时候, 圆括号就有用了. 如果和下面要讲的”|”操作符结合使用, 也非常有用.

```

1 bash$ egrep 're(a|e)d' misc.txt
2 People who read seem to be better informed than those who do not.
3 The clarinet produces sound by the vibration of its reed.

```

- 竖线. 就是RE中的”或”操作符, 使用它能够匹配一组可选字符中的任意一个.

与GNU工具一样, 某些版本的sed, ed, 和ex一样能够支持扩展正则表达式, 上边这部分就描述了扩展正则表达式.

POSIX字符类. [:class:]

- [:alnum:]. 匹配字母和数字. 等价于A-Za-z0-9.
- [:alpha:]. 匹配字母. 等价于A-Za-z.
- [:blank:]. 匹配一个空格或是一个制表符(tab).
- [:cntrl:]. 匹配控制字符.
- [:digit:]. 匹配(十进制)数字. 等价于0-9.

- `[:graph:]`. (可打印的图形字符). 匹配ASCII码值范围在33 - 126之间的字符. 与下面所提到的`[:print:]`类似, 但是不包括空格字符(空格字符的ASCII码是32).
- `[:lower:]`. 匹配小写字母. 等价于`a-z`.
- `[:print:]`. (可打印的图形字符). 匹配ASCII码值范围在32 - 126之间的字符. 与上边的`[:graph:]`类似, 但是包含空格.
- `[:space:]`. 匹配空白字符(空格和水平制表符).
- `[:upper:]`. 匹配大写字母. 等价于`A-Z`.
- `[:xdigit:]`. 匹配16进制数字. 等价于`0-9A-Fa-f`.

POSIX字符类通常都要用引号或**双中括号**(`[[]]`)引起起来.

```
1 bash$ grep [[[:digit:]]] test.file
2 abc=723
```

如果在一个受限的范围内, 这些字符类甚至可以用在通配(globbing)中.

```
1 bash$ ls -l ?[[[:digit:]]][[:digit:]]?
2 -rw-rw-r--    1 bozo  bozo        0 Aug 21 14:47 a33b
```

如果想了解POSIX字符类在脚本中的使用情况, 请参考[例子12-18](#)和[例子12-19](#).

Sed, awk, 和Perl在脚本中一般都被用作过滤器, 这些过滤器将会以RE为参数, 对文件或者I/O流进行”过滤”或转换. 请参考[例子A-12](#)和[例子A-17](#), 来详细了解这种用法.

对于RE这个复杂的主题, 标准的参考材料是Friedl的Mastering Regular Expressions. 由Dougherty和Robbins所编写的Sed & Awk这本书, 也对RE进行了清晰的论述. 如果想获得这些书的更多信息, 请察看[参考文献](#).

注意事项

1. 因为sed, awk, 和grep通常用于处理单行, 但是不能匹配一个换行符. 如果你想处理多行输入的话, 那么你可以使用”点”来匹配换行符.

因为sed, awk, 和grep通常用于处理单行, 但是不能匹配一个换行符. 如果你想处理多行输入的话, 那么你可以使用”点”来匹配换行符.

```
1 #!/bin/bash
2
3 sed -e 'N;s/.*/[&]/' << EOF    # Here Document
4 line1
5 line2
6 EOF
7 # 输出:
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 # 输出:
20 # line
21 # 1
22
23
24 # 感谢, S.C.
25
26 exit 0
```

2 通配(globbing)

Bash本身并不会识别正则表达式。在脚本中，使用RE的是命令和工具—比如sed和awk—这些工具能够解释RE。

Bash仅仅做的一件事是文件名扩展(译者注：作者在前面使用的名词是filename globbing，这里又使用filename expansion，造成术语不统一，希望读者不要产生误解。) [1] – 这就是所谓的通配(globbing) – 但是这里所使用的并不是标准的RE，而是使用通配符。通配(globbing)解释标准通配符，*，?，中括号扩起来的字符，还有其他一些特殊字符(比如^用来表示取反匹配)。然而通配(globbing)所使用的通配符有很大的局限性。包含*的字符串不能匹配以“点”开头的文件，比如，.bashrc。[2] 另外，RE中所使用的?，与通配(globbing)中所使用的?，含义并不相同。

```
1 bash$ ls -l
2 total 2
3 -rw-rw-r-- 1 bozo bozo 0 Aug 6 18:42 a.1
4 -rw-rw-r-- 1 bozo bozo 0 Aug 6 18:42 b.1
5 -rw-rw-r-- 1 bozo bozo 0 Aug 6 18:42 c.1
6 -rw-rw-r-- 1 bozo bozo 466 Aug 6 17:48 t2.sh
7 -rw-rw-r-- 1 bozo bozo 758 Jul 30 09:02 test1.txt
8
9 bash$ ls -l t?.sh
10 -rw-rw-r-- 1 bozo bozo 466 Aug 6 17:48 t2.sh
11
12 bash$ ls -l [ab]*
13 -rw-rw-r-- 1 bozo bozo 0 Aug 6 18:42 a.1
14 -rw-rw-r-- 1 bozo bozo 0 Aug 6 18:42 b.1
15
16 bash$ ls -l [a-c]*
17 -rw-rw-r-- 1 bozo bozo 0 Aug 6 18:42 a.1
18 -rw-rw-r-- 1 bozo bozo 0 Aug 6 18:42 b.1
19 -rw-rw-r-- 1 bozo bozo 0 Aug 6 18:42 c.1
20
21 bash$ ls -l [^ab]*
22 -rw-rw-r-- 1 bozo bozo 0 Aug 6 18:42 c.1
23 -rw-rw-r-- 1 bozo bozo 466 Aug 6 17:48 t2.sh
24 -rw-rw-r-- 1 bozo bozo 758 Jul 30 09:02 test1.txt
25
26 bash$ ls -l {b*,c*,*est*}
27 -rw-rw-r-- 1 bozo bozo 0 Aug 6 18:42 b.1
28 -rw-rw-r-- 1 bozo bozo 0 Aug 6 18:42 c.1
29 -rw-rw-r-- 1 bozo bozo 758 Jul 30 09:02 test1.txt
```

Bash只能对未用引号引用起来的命令行参数进行文件名扩展。`echo`命令可以印证这一点。

```
1 bash$ echo *
2 a.1 b.1 c.1 t2.sh test1.txt
3
```

```
4 bash$ echo t*
5 t2.sh test1.txt
```

Bash在通配(globbing)中解释特殊字符的行为是可以修改的. set -f命令可以禁用通配(globbing), 而且`shopt`命令的选项`nocaseglob`和`nullglob`可以修改通配(globbing)的行为. 请参考[例子10-4](#).

注意事项

1. 文件名扩展意味着扩展包含有特殊字符的文件名模式或模版. 比如, `example.???`可能会被扩展成`example.001`或(和)`example.txt`.
2. 文件名扩展能够匹配以”点”开头的文件, 但是, 你必须在模式字符串中明确的写上”点”(.), 才能够扩展.

第20章 子shell

运行一个shell脚本的时候, 会启动命令解释器的另一个实例. 就好像你的命令是在命令行提示下被解释的一样, 类似于批处理文件中的一系列命令. 每个shell脚本都有效地运行在父shell的一个子进程中. 这个父shell指的是在一个控制终端或在一个xterm窗口中给出命令提示符的那个进程.

shell脚本也能启动它自己的子进程. 这些子shell能够使脚本并行的, 有效的, 同时运行多个子任务.

►: 一般来说, 脚本中的外部命令能够生成(fork)一个子进程, 然而Bash的内建命令却不会这么做. 也正是由于这个原因, 内建命令比等价的外部命令要执行的快.

圆括号中的命令列表

```
( command1; command2; command3; ... )
```

圆括号中命令列表的命令将会运行在一个子shell中.

子shell中的变量对于子shell之外的代码块来说, 是不可见的. 当然, 父进程也不能访问这些变量, 父进程指的是产生这个子shell的shell. 事实上, 这些变量都是局部变量.

例子20-1. 子shell中的变量作用域

```
1 #!/bin/bash
2 # subshell.sh
3
4 echo
5
6 echo "Subshell level OUTSIDE subshell = $BASH_SUBSHELL"
7 # Bash, 版本3, 添加了这个新的          $BASH_SUBSHELL 变量.
8 echo
9
10 outer_variable=Outer
11
12 (
13 echo "Subshell level INSIDE subshell = $BASH_SUBSHELL"
14 inner_variable=Inner
15
16 echo "From subshell, \"inner_variable\" = $inner_variable"
17 echo "From subshell, \"outer\" = $outer_variable"
18 )
19
20 echo
21 echo "Subshell level OUTSIDE subshell = $BASH_SUBSHELL"
22 echo
23
24 if [ -z "$inner_variable" ]
25 then
```

```

26 echo "inner_variable undefined in main body of shell"
27 else
28   echo "inner_variable defined in main body of shell"
29 fi
30
31 echo "From main body of shell, \"inner_variable\" = $inner_variable"
32 # $inner_variable将被作为未初始化的变量，被显示出来，
33 #+ 这是因为变量是在子shell里定义的"局部变量".
34 # 还有补救的办法么？
35
36 echo
37
38 exit 0

```

请参考[例子31-2](#).

子shell中的目录更改不会影响到父shell.

例子20-2. 列出用户的配置文件

```

1#!/bin/bash
2# allprofs.sh: 打印所有用户的配置文件
3
4# 由Heiner Steven编写，并由本书作者进行了修改.
5
6FILE=.bashrc # 在原始脚本中，File containing user profile,
7      #+ 包含用户profile的是文件".profile".
8
9for home in `awk -F: '{print $6}' /etc/passwd`
10do
11  [ -d "$home" ] || continue    # 如果没有home目录，跳出本次循环.
12  [ -r "$home" ] || continue    # 如果home目录没有读权限，跳出本次循环.
13  (cd $home; [ -e $FILE ] && less $FILE)
14done
15
16# 当脚本结束时，不必使用'cd'命令返回原来的目录.
17#+ 因为'cd $home'是在子shell中发生的，不影响父shell.
18
19exit 0

```

子shell可用于为一组命令设置一个“独立的临时环境”.

```

1COMMAND1
2COMMAND2
3COMMAND3
4(
5  IFS=:
6  PATH=/bin
7  unset TERMINFO
8  set -C

```

```

9  shift 5
10 COMMAND4
11 COMMAND5
12 exit 3 # 只是从子shell退出.
13 )
14 # 父shell不受任何影响，并且父shell的环境也没有被更改.
15 COMMAND6
16 COMMAND7
17 \end{everbati}
18
19 子shell的另一个应用，是可以用来检测一个变量是否被定义.
20
21 \begin{everbatim}
22 if (set -u; : $variable) 2> /dev/null
23 then
24   echo "Variable is set."
25 fi    # 变量已经在当前脚本中被设置,
26      #+ 或者是一个Bash的内建变量,
27      #+ 或者是在当前环境下的一个可见变量(指已经被export的环境变量).
28
29 # 也可以写成          [[ ${variable-x} != x || ${variable-y} != y ]]
30 # 或                  [[ ${variable-x} != x$variable ]]
31 # 或                  [[ ${variable+x} = x ]]
32 # 或                  [[ ${variable-x} != x ]]

```

子shell还可以用来检测一个加锁的文件:

```

1 if (set -C; : > lock_file) 2> /dev/null
2 then
3   : # lock_file不存在，还没有用户运行这个脚本
4 else
5   echo "Another user is already running that script."
6 exit 65
7 fi
8
9 # 这段程序由Stephane Chazelas所编写,
10 #+ Paulo Marcel Coelho Aragao做了一些修改.

```

进程在不同的子shell中可以并行地执行。这样就可以把一个复杂的任务分成几个小的子问题来同时处理。

例子20-3. 在子shell中进行并行处理

```

1 (cat list1 list2 list3 | sort | uniq > list123) &
2 (cat list4 list5 list6 | sort | uniq > list456) &
3 # 列表的合并与排序同时进行.
4 # 放到后台运行可以确保能够并行执行.
5 #
6 # 等效于

```

```
7 # cat list1 list2 list3 | sort | uniq > list123 &
8 # cat list4 list5 list6 | sort | uniq > list456 &
9
10 wait    # 不再执行下面的命令，直到子shell执行完毕.
11
12 diff list123 list456
```

使用”—”管道操作符, 将I/O流重定向到一个子shell中, 比如ls -al — (command).

►: 在大括号中的命令不会启动子shell.

```
{ command1; command2; command3; . . . commandN; }
```

第21章 受限shell

在受限shell中禁用的命令

在受限模式下运行一个脚本或脚本片断，将会禁用某些命令，这些命令在正常模式下都可以运行。这是一种安全策略，目的是为了限制脚本用户的权限，并且能够让运行脚本所导致的危害降低到最小。

使用cd命令更改工作目录。

更改环境变量\$PATH, \$SHELL, \$BASH_ENV, 或\$ENV的值。

读取或修改环境变量\$SHELLOPTS的值。

输出重定向。

调用的命令路径中包括有一个或多个斜杠(/)。

调用exec，把当前的受限shell替换成另外一个进程。

能够在无意中破坏脚本的命令。

在脚本中企图脱离受限模式的操作。

例子21-1. 在受限模式下运行脚本

```
1 1#!/bin/bash
2
3 # 脚本开头以"#!/bin/bash -r"来调用,
4 #+ 会使整个脚本在受限模式下运行.
5
6 echo
7
8 echo "Changing directory."
9 cd /usr/local
10 echo "Now in `pwd`"
11 echo "Coming back home."
12 cd
13 echo "Now in `pwd`"
14 echo
15
16 # 非受限的模式下, 所有操作都正常.
17
18 set -r
19 # set --restricted 也具有相同的效果.
20 echo "==> Now in restricted mode. <=="
```

```
21
22 echo
23 echo
24
25 echo "Attempting directory change in restricted mode."
26 cd ..
```

```
27 echo "Still in 'pwd'"  
28  
29 echo  
30 echo  
31  
32 echo "\$SHELL = $SHELL"  
33 echo "Attempting to change shell in restricted mode."  
34 SHELL="/bin/ash"  
35 echo  
36 echo "\$SHELL= $SHELL"  
37  
38 echo  
39 echo  
40  
41 echo "Attempting to redirect output in restricted mode."  
42 ls -l /usr/bin > bin.files  
43 ls -l bin.files # 尝试列出刚才创建的文件.  
44  
45 echo  
46  
47 exit 0
```

第22章 进程替换

进程替换与[命令替换](#)很相似。命令替换把一个命令的结果赋值给一个变量，比如`dir_contents=`ls -al``或`xref=$(grep word datafile)`。进程替换把一个进程的输出提供给另一个进程(换句话说，它把一个命令的结果发给了另一个命令)。

命令替换的模版

用圆括号扩起来的命令

```
>(command)  
<(command)
```

启动进程替换。它使用`/dev/fd/n`文件将圆括号中的进程处理结果发送给另一个进程。[\[1\]](#)
(译者注：实际上现代的UNIX类操作系统提供的`/dev/fd/n`文件是与[文件描述符](#)相关的，整数n指的就是进程运行时对应数字的文件描述符)

►：在”`i`”或”`l`”与圆括号之间是没有空格的。如果加了空格，会产生错误。

```
1  
2 bash$ echo >(true)  
3 /dev/fd/63  
4  
5 bash$ echo <(true)  
6 /dev/fd/63
```

Bash在两个文件描述符之间创建了一个管道，`-fIn`和`fOut`。`true`命令的`stdin`被连接到`fOut`(`dup2(fOut, 0)`)，然后Bash把`/dev/fd/fIn`作为参数传给`echo`。如果系统缺乏`/dev/fd/n`文件，Bash会使用临时文件。(感谢，S.C.)

进程替换可以比较两个不同命令的输出，甚至能够比较同一个命令不同选项情况下的输出。

```
1  
2 bash$ comm <(ls -l) <(ls -al)  
3 total 12  
4 -rw-rw-r-- 1 bozo bozo 78 Mar 10 12:58 File0  
5 -rw-rw-r-- 1 bozo bozo 42 Mar 10 12:58 File2  
6 -rw-rw-r-- 1 bozo bozo 103 Mar 10 12:58 t2.sh  
7  
8 total 20  
9 drwxrwxrwx 2 bozo bozo 4096 Mar 10 18:10 .  
10 drwx----- 72 bozo bozo 4096 Mar 10 17:58 ..  
11 -rw-rw-r-- 1 bozo bozo 78 Mar 10 12:58 File0  
12 -rw-rw-r-- 1 bozo bozo 42 Mar 10 12:58 File2  
13 -rw-rw-r-- 1 bozo bozo 103 Mar 10 12:58 t2.sh
```

使用进程替换来比较两个不同目录的内容(可以查看哪些文件名相同，哪些文件名不同)：

```
1 diff <(ls $first_directory) <(ls $second_directory)
```

一些进程替换的其他用法与技巧：

```
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 # 列出系统3个主要'bin'目录中的所有文件，并且按文件名进行排序。
6 # 注意是3个(查一下，上面就3个圆括号)明显不同的命令输出传递给'sort'.
7
8
9 diff <(command1) <(command2)      # 给出两个命令输出的不同之处。
10
11 tar cf >(bzip2 -c > file.tar.bz2) $directory_name
12 # 调用"tar cf /dev/fd/?? $directory_name"，和"bzip2 -c > file.tar.bz2".
13 #
14 # 因为/dev/fd/<n>的系统属性，
15 # 所以两个命令之间的管道不必被命名。
16 #
17 # 这种效果可以被模拟出来。
18 #
19 bzip2 -c < pipe > file.tar.bz2&
20 tar cf pipe $directory_name
21 rm pipe
22 #      或
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
28 # 感谢，Stephane Chazelas

```

一个读者给我发了一个有趣的例子，是关于进程替换的，如下。

```

1 # 摘自SuSE发行版中的代码片断：
2
3 while read des what mask iface; do
4 # 这里省略了一些命令...
5 done < <(route -n)
6
7
8 # 为了测试它，我们让它做点事。
9 while read des what mask iface; do
10   echo $des $what $mask $iface
11 done < <(route -n)
12
13 # 输出：
14 # Kernel IP routing table
15 # Destination Gateway Genmask Flags Metric Ref Use Iface
16 # 127.0.0.0 0.0.0.0 255.0.0.0 U 0 0 0 lo

```

```

17
18
19
20 # 就像Stephane Chazelas所给出的那样，一个更容易理解的等价代码是：
21 route -n |
22     while read des what mask iface; do    # 管道的输出被赋值给了变量.
23         echo $des $what $mask $iface
24     done # 这将产生出与上边相同的输出.
25         # 然而，Ulrich Gayer指出 . .
26         #+ 这个简单的等价版本在while循环中使用了一个子shell,
27         #+ 因此当管道结束后，变量就消失了.
28
29
30
31 # 更进一步，Filip Moritz解释了上面两个例子之间存在一个细微的不同之处，
32 #+ 如下所示。
33
34 (
35 route -n | while read x; do ((y++)); done
36 echo $y # $y 仍然没有被声明或设置
37
38 while read x; do ((y++)); done <<(route -n)
39 echo $y # $y 的值为route -n的输出行数.
40 )
41
42 # 一般来说，(译者注：原书作者在这里并未加注释符号"#", 应该是笔误)
43 (
44 : | x=x
45 # 看上去是启动了一个子shell
46 : | ( x=x )
47 # 但
48 x=x <<(:)
49 # 其实不是
50 )
51
52 # 当你要解析csv或类似东西的时候，这非常有用.
53 # 事实上，这就是SuSE的这个代码片断所要实现的功能.

```

注意事项

1. 这与[命名管道](#)(临时文件)具有相同的作用，并且，事实上，命名管道也被同时使用在进程替换中。

第23章 函数

本章目录

- 1. 复杂函数和函数复杂性
 - 2. 局部变量
 - 3. 不使用局部变量的递归
-

与“真正的”编程语言一样, Bash也有函数, 虽然在某些实现方面稍有限制。一个函数就是一个子程序, 用于实现一系列操作的代码块, 它是完成特定任务的“黑盒子”。当存在重复代码的时候, 或者当一个任务只需要轻微修改就被重复使用的时候, 你就需要考虑使用函数了。

```
function function_name {  
    command...  
}  
或  
function_name () {  
    command...  
}
```

C程序员肯定会更加喜欢第二中格式的写法(并且这种写法可移植性更好)。

在C中, 函数的左大括号也可以写在下一行中。

```
function_name ()  
{  
    command...  
}
```

只需要简单的调用函数名, 函数就会被调用或触发。

简单函数

```
1 #!/bin/bash  
2  
3 JUST_A_SECOND=1  
4  
5 funky ()  
6 { # 这是一个最简单的函数。  
7     echo "This is a funky function."  
8     echo "Now exiting funky function."  
9 } # 函数必须在调用前声明。  
10  
11  
12 fun ()
```

```
13 # 一个稍微复杂一些的函数.
14 i=0
15 REPEATS=30
16
17 echo
18 echo "And now the fun really begins."
19 echo
20
21 sleep $JUST_A_SECOND    # 嘿，暂停一秒！
22 while [ $i -lt $REPEATS ]
23 do
24     echo "-----FUNCTIONS----->"
25     echo "<-----ARE-----"
26     echo "<-----FUN----->"
27     echo
28     let "i+=1"
29 done
30 }
31
32 # 现在，调用这两个函数。
33
34 funky
35 fun
36
37 exit 0
```

函数定义必须在第一次调用函数之前完成。没有像C中函数”声明”的方法。

```
1 f1
2 # 因为函数"f1"还没有被定义，这会产生一个错误。
3
4 declare -f f1      # 这样也没用。
5 f1                  # 仍然会引起错误。
6
7 # 然而...
8
9
10 f1 ()
11 {
12     echo "Calling function \"f2\" from within function \"f1\"."
13     f2
14 }
15
16 f2 ()
17 {
18     echo "Function \"f2\"."
```

```
19 }
20
21 f1 # 虽然在f2在定义前被引用过,
22     #+ 实际上f2到这儿才被调用.
23 # 所以这么做是正常的.
24
25 # 感谢, S.C.
```

甚至可以在一个函数内嵌套另一个函数, 虽然这么做并没有多大用处.

```
1 f1 ()
2 {
3
4   f2 () # nested
5   {
6     echo "Function \"f2\", inside \"f1\"."
7   }
8
9 }
10
11 f2 # 产生一个错误.
12   # 即使你先写出"declare -f f2"也没用.
13
14 echo
15
16 f1 # 什么事都没干, 因为调用"f1"并不会自动调用"f2".
17 f2 # 现在, 可以正确的调用"f2"了,
18     #+ 因为之前调用"f1"使"f2"在脚本中变得可见了.
19
20 # 感谢, S.C.
```

函数声明可以出现在看上去不可能出现的地方, 比如说本应出现命令的地方, 也可以出现函数声明.

```
1 ls -l | foo() { echo "foo"; } # 可以这么做, 但没什么用.
2
3
4
5 if [ "$USER" = bozo ]
6 then
7   bozo_greet ()  # 在if/then结构中定义了函数.
8   {
9     echo "Hello, Bozo."
10  }
11 fi
12
13 bozo_greet      # 只能由Bozo运行, 其他用户使用的话, 会引起错误.
14
```

```
15
16
17 # 在某些上下文中，这样做可能会有用。
18 NO_EXIT=1    # 将会打开下面的函数定义。
19
20 [[ $NO_EXIT -eq 1 ]] && exit() { true; }      # 在"与列表"中定义函数。
21 # 如果$NO_EXIT为1，那就声明"exit ()"。
22 # 把"exit"化名为"true"，将会禁用内建的"exit"命令。
23
24 exit # 这里调用的是"exit ()"函数，而不是"exit"内建命令。
25
26 # 感谢，S.C.
```

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
```

函数以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23-2. 带参数的函数

```
1#!/bin/bash
2# 函数和参数
3
4DEFAULT=default          # 默认参数值。
5
6func2 () {
7    if [ -z "$1" ]          # 第一个参数是否长度为零？
8    then
9        echo "-Parameter #1 is zero length.-" # 或者没有参数被传递进来。
10   else
11       echo "-Param #1 is \"$1\".-"
12   fi
13
14variable=${1-$DEFAULT}    # 这里的参数替换
15echo "variable = $variable" #+ 表示什么？
16#
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
21if [ "$2" ]
22then
23    echo "-Parameter #2 is \"$2\".-"
24fi
25
26return 0
27
28echo
29
30echo "Nothing passed."
31func2          # 不带参数调用
32echo
33
34
35echo "Zero-length parameter passed."
36func2 ""        # 使用0长度的参数进行调用
37echo
```

```

38
39 echo "Null parameter passed."
40 func2 "$uninitialized_param"    # 使用未初始化的参数进行调用
41 echo
42
43 echo "One parameter passed."
44 func2 first          # 带一个参数调用
45 echo
46
47 echo "Two parameters passed."
48 func2 first second   # 带两个参数调用
49 echo
50
51 echo "\"\" \"second\" passed."
52 func2 "" second      # 带两个参数调用,
53 echo                  # 第一个参数长度为0, 第二个参数是由ASCII码组成的字符串.
54
55 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33-15](#)).

但是, 传给脚本的命令行参数怎么办? 在函数内部, 这些传给脚本的命令行参数也可见么? 好, 现在让我们弄清楚这个问题.

例子23-3. 函数与传递给脚本的命令行参数

```

1#!/bin/bash
2# func-cmdlinearg.sh
3# 调用这个脚本, 并且带一个命令行参数.
4#+ 类似于 $0 arg1.
5
6
7func ()
8
9{
10echo "$1"
11}
12
13echo "First call to function: no arg passed."
14echo "See if command-line arg is seen."
15func
16# 不行! 命令行参数不可见.
17
18echo =====
19echo
20echo "Second call to function: command-line arg passed explicitly."
21func $1
22# 现在可见了!

```

```
23  
24 exit 0
```

与别的编程语言相比, 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, 也就是被看作字符串. 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34-2](#))提供了一种笨拙的机制, 来将变量指针传递给函数.

例子23-4. 将一个间接引用传递给函数

```
1 #!/bin/bash  
2 # func-cmdlinearg.sh  
3 # 调用这个脚本, 并且带一个命令行参数.  
4 #+ 类似于 $0 arg1.  
5  
6  
7 func ()  
8  
9 {  
10 echo "$1"  
11 }  
12  
13 echo "First call to function: no arg passed."  
14 echo "See if command-line arg is seen."  
15 func  
16 # 不行! 命令行参数不可见.  
17  
18 echo "=====."  
19 echo  
20 echo "Second call to function: command-line arg passed explicitly."  
21 func $1  
22 # 现在可见了!  
23  
24 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23-5. 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所编写.  
5  
6 dereference ()  
7 {  
8     y=\$"\$1"    # 变量名.  
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23-6. 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)
4
5ITERATIONS=3 # 取得输入的次数.
6icount=1
7
8my_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
13local local_var
14
15echo -n "Enter a value "
16eval '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.
19read 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"$1"的值赋给它.
23}
24
25echo
26
27while [ "$icount" -le "$ITERATIONS" ]
28do
29    my_read var
30    echo "Entry #$icount = $var"
31    let "icount += 1"
32    echo

```

```
33 done
34
35
36 # 感谢Stephane Chazelas提供这个例子.
37
38 exit 0
```

退出与返回

退出状态码 函数返回一个值, 被称为退出状态码. 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象C函数一样有”返回值”.

return

终止一个函数. return命令[1] 可选的允许带一个整型参数, 这个整数将作为函数的”退出状态码”返回给调用这个函数的脚本, 并且这个整数也被赋值给变量\$?.

例子23-7. 取两个数中的最大值

```
1#!/bin/bash
2# max.sh: 取两个整数中的最大值.
3
4E_PARAM_ERR=-198      # 如果传给函数的参数少于2个时, 就返回这个值.
5EQUAL=-199            # 如果两个整数相等时, 返回这个值.
6# 任意超出范围的
7#+ 参数值都可能传递到函数中.
8
9max2 ()                # 返回两个整数中的最大值.
10{                      # 注意: 参与比较的数必须小于257.
11    if [ -z "$2" ]
12    then
13        return $E_PARAM_ERR
14    fi
15
16    if [ "$1" -eq "$2" ]
17    then
18        return $EQUAL
19    else
20        if [ "$1" -gt "$2" ]
21        then
22            return $1
23        else
24            return $2
25        fi
26    fi
27}
28
29max2 33 34
30return_val=$?
```

```

31
32 if [ "$return_val" -eq $E_PARAM_ERR ]
33 then
34     echo "Need to pass two parameters to the function."
35 elif [ "$return_val" -eq $EQUAL ]
36 then
37     echo "The two numbers are equal."
38 else
39     echo "The larger of the two numbers is $return_val."
40 fi
41
42
43 exit 0
44
45 # 练习(简单):
46 # -----
47 # 把这个脚本转化为交互式脚本,
48 #+ 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是可读的, 那么就把REPLY设置为文件的行数.
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字符.
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23-8. 将阿拉伯数字转化为罗马数字

```

1#!/bin/bash
2
3# 将阿拉伯数字转化为罗马数字
4# 范围: 0 - 200
5# 比较粗糙, 但可以正常工作.
6
7# 扩展范围, 并且完善这个脚本, 作为练习.

```

```
8
9 # 用法: roman number-to-convert
10
11 LIMIT=200
12 E_ARG_ERR=65
13 E_OUT_OF_RANGE=66
14
15 if [ -z "$1" ]
16 then
17   echo "Usage: `basename $0` number-to-convert"
18   exit $E_ARG_ERR
19 fi
20
21 num=$1
22 if [ "$num" -gt $LIMIT ]
23 then
24   echo "Out of range!"
25   exit $E_OUT_OF_RANGE
26 fi
27
28 to_roman ()  # 在第一次调用函数前必须先定义它.
29 {
30   number=$1
31   factor=$2
32   rchar=$3
33   let "remainder = number - factor"
34   while [ "$remainder" -ge 0 ]
35   do
36     echo -n $rchar
37     let "number -= factor"
38     let "remainder = number - factor"
39   done
40
41   return $number
42   # 练习:
43   # -----
44   # 解释这个函数是如何工作的.
45   # 提示: 依靠不断的除, 来分割数字.
46 }
47
48
49 to_roman $num 100 C
50 num=$?
51 to_roman $num 90 LXXX
52 num=$?
```

```

53 to_roman $num 50 L
54 num=$?
55 to_roman $num 40 XL
56 num=$?
57 to_roman $num 10 X
58 num=$?
59 to_roman $num 9 IX
60 num=$?
61 to_roman $num 5 V
62 num=$?
63 to_roman $num 4 IV
64 num=$?
65 to_roman $num 1 I
66
67 echo
68
69 exit 0

```

也请参考[例子10-28](#).

函数所能返回最大的正整数是255. return命令与[退出状态码](#)的概念被紧密联系在一起，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[工作区](#)能够应付这个情况.

[例子23-9. 测试函数最大的返回值](#)

```

1 #!/bin/bash
2 # return-test.sh
3
4 # 函数所能返回的最大正整数为255.
5
6 return_test ()          # 传给函数什么值，就返回什么值.
7 {
8     return $1
9 }
10
11 return_test 27          # o.k.
12 echo $?                  # 返回27.
13
14 return_test 255          # 依然是o.k.
15 echo $?                  # 返回255.
16
17 return_test 257          # 错误！
18 echo $?                  # 返回1（对应各种错误的返回码）.
19
20 # =====
21 return_test -151896      # 能返回一个大负数么？
22 echo $?                  # 能否返回-151896?

```

```

23 # 显然不行! 只返回了168.
24 # Bash 2.05b以前的版本
25 #+ 允许返回大负数.
26 # Bash的新版本(2.05b之后)修正了这个漏洞.
27 # 这可能会影响以前所编写的脚本.
28 # 一定要小心!
29 # =====
30
31 exit 0

```

如果你想获得“大整数”返回值的话，其实最简单的办法就是将“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”，然后使用[命令替换](#)来捕捉此值。请参考Section 33.7中关于这种用法的讨论。

例子23-10. 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
7EQUAL=0 # 如果两个值相等, 那就返回这个值.
8E_PARAM_ERR=-99999 # 没有足够的参数传递给函数.
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 #+ 可以把函数看作为一个命令,
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 #+ 来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 -- 月份号), 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 # 这是一个模版, 用于将参数传递给内嵌awk的脚本:
9 #                         $"${script_parameter}"'
10
11 # 需要做一些错误检查, 来保证月份号正确, 在范围(1-12)之间,
12 #+ 别忘了检查闰年的二月.
13 }
14
15 # -----
16 # 用例:
17 month=4          # 以四月为例.
18 days_in=$(month_length $month)
19 echo $days_in   # 30
20 #

```

也请参考[例子A-7](#).

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23-11. 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=65
9
10 file=/etc/passwd
11 pattern=$1
12
13 if [ $# -ne "$ARGCOUNT" ]
14 then
15   echo "Usage: `basename $0` USERNAME"
16   exit $E_WRONGARGS
17 fi
18
19 file_excerpt () # 按照要求的模式来扫描文件, 然后打印文件相关的部分.
20 {
21 while read line # "while"并不一定非得有"[ condition ]"不可.
22 do
23   echo "$line" | grep $1 | awk -F ":" '{ print $5 }' # awk用":"作为界定符.
24 done
25 } <$file # 重定向到函数的stdin.
26
27 file_excerpt $pattern
28
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 # 或
32 #     awk -F: '/PATTERN/ {print $5}'
33 # 或
34 #     awk -F: '($1 == "username") { print $5 }' # 从username中获得真名.
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
29
30 # 感谢, S.C.
```

注意事项

1. return命令是Bash[内建命令builtin](#).

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块中才有意义。

例子23-12. 局部变量的可见范围

```
1 #!/bin/bash
2 # 函数内部的局部变量与全局变量。
3
4 func ()
5 {
6     local loc_var=23      # 声明为局部变量。
7     echo                  # 使用'local'内建命令。
8     echo "\"loc_var\" in function = $loc_var"
9     global_var=999        # 没有声明为局部变量。
10                # 默认为全局变量。
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。
```

函数被调用之前,所有在函数中声明的变量,在函数体外都是不可见的,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      # 在函数被调用之前,
6                         #+ 变量只在函数体内可见。
```

```

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还不能被访问.
12
13 func
14 echo "global_var = $global_var" # global_var = 37
15 # 已经在函数调用的时候设置了变量的值.

```

23.2.1. 局部变量使递归变为可能.

局部变量允许递归, [1] 但是这种方法会产生大量的计算, 因此在shell脚本中, 非常明确的不推荐这种做法. [2]

例子23-13. 使用局部变量的递归

```

1#!/bin/bash
2
3# 阶乘
4# -----
5
6
7# bash允许递归吗?
8# 嗯, 允许, 但是...
9# 他太慢了, 所以恐怕你难以忍受.
10
11
12MAX_ARG=5
13E_WRONG_ARGS=65
14E_RANGE_ERR=66
15
16
17if [ -z "$1" ]
18then
19    echo "Usage: `basename $0` number"
20    exit $E_WRONG_ARGS
21fi
22
23if [ "$1" -gt $MAX_ARG ]
24then
25    echo "Out of range (5 is maximum)."
26    # 现在让我们来了解一些实际情况.
27    # 如果你想计算比这个更大的范围的阶乘,
28    #+ 应该用真正的编程语言来重写它.
29    exit $E_RANGE_ERR
30fi
31

```

```

32 fact () {
33 {
34     local number=$1
35     # 变量"number"必须声明为局部变量,
36     #+ 否则不能正常工作.
37     if [ "$number" -eq 0 ]
38     then
39         factorial=1      # 0的阶乘为1.
40     else
41         let "decrnum = number - 1"
42         fact $decrnum # 递归的函数调用(就是函数调用自己).
43         let "factorial = $number * $?"
44     fi
45
46     return $factorial
47 }
48
49 fact $1
50 echo "Factorial of $1 is $?."
51
52 exit 0

```

也可以参考例子A-16, 这是一个脚本中递归的例子. 必须认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.

注意事项

1. [Herbert Mayer](#) 给递归下的定义为: "... expressing an algorithm by using a simpler version of that same algorithm (使用相同算法的一个简单版本来表达这个算法) ..." 一个递归函数就是调用自身的函数.
2. 过多层次的递归可能会产生段错误, 继而导致脚本崩溃.

```

1 #!/bin/bash
2
3 # 警告: 运行这个脚本可能使你的系统失去响应!
4 # 如果你运气不错, 在它用光所有可用内存之前会因为段错误而退出.
5
6 recursive_function () {
7
8 echo "$1"      # 让这个函数做点事, 以便于加速段错误.
9 (( $1 < $2 )) && recursive_function $(( $1 + 1 )) $2;
10 # 当第一个参数比第二个参数少时,
11 #+ 将第一个参数加1, 然后再递归.
12 }
13
14 recursive_function 1 50000 # Recurse 50,000 levels!

```

```
15 # 极有可能产生段错误(这依赖于栈尺寸, 可以用ulimit -m来设置).
16
17 # 这种深度递归可能会产生C程序的段错误,
18 #+ 这是由于耗光所有的栈内存所引起的.
19
20
21 echo "This will probably not print."
22 exit 0 # 这个脚本是不会在这里正常退出的.
23
24 # 感谢, Stephane Chazelas.
```

3 不使用局部变量的递归

即使不使用局部变量，函数也可以递归的调用自身。

例子23-14. 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 #
10 # 经过脚本原作者同意
11 #+ 可以使用在"Advanced Bash Scripting Guide"中.
12 # 本书作者对此脚本做了少许修改.
13
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, 它们从下到上,
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动到另一个柱子上.
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, 这是规则, 绝对不能把大盘子放到小盘子的上面.
33 #
34 # 如果盘子的数量比较少, 那么移不了几次就能完成.
35 #+ 但是随着盘子数量的增加,
36 #+ 移动次数几乎成倍的增长,
37 #+ 而且移动的"策略"也会变得越来越复杂.
38 #
39 # 想了解更多信息的话, 请访问http://hanoi.kernelthread.com.
40 #
```

```

41 #
42 #      ...
43 #      | |
44 #      _|_|_
45 #      |_____|_
46 #      |_____|
47 #      |_____|_
48 #      |_____|_
49 #      | |
50 #
51 # .-----.
52 # |*****| #1          #2          #3
53 #
54 #=====
55
56
57 E_NOPARAM=66 # 没有参数传给脚本.
58 E_BADPARAM=67 # 传给脚本的盘子个数不符合要求.
59 Moves=         # 保存移动次数的全局变量.
60             # 这里修改了原来的脚本.
61
62 dohanoi() {   # 递归函数.
63     case $1 in
64     0)
65         ;;
66     *)
67         dohanoi "$((\$1-1))" \$2 \$4 \$3
68         echo move \$2 "-->" \$3
69         let "Moves += 1" # 这里修改了原脚本.
70         dohanoi "$((\$1-1))" \$4 \$3 \$2
71         ;;
72     esac
73 }
74
75 case $# in
76 1)
77     case $((\$1>0)) in      # 至少要有一个盘子.
78     1)
79         dohanoi \$1 1 3 2
80         echo "Total moves = $Moves"
81         exit 0;
82         ;;
83     *)
84         echo "\$0: illegal value for number of disks";
85         exit \$E_BADPARAM;

```

```
86      ;;
87      esac
88      ;;
89 *)      echo "usage: $0 N"
90      echo "          Where \"N\" is the number of disks."
91      exit $E_NOPARAM;
92      ;;
93
94 esac
95
96 # 练习:
97 # -----
98 # 1) 这个位置以下的代码会不会被执行?
99 #   为什么不? (容易)
100 # 2) 解释一下这个运行的"dohanoi"函数的运行原理.
101 #   (比较难)
```

第24章 别名

Bash别名本质上来说不过就是个简称，缩写，是一种避免输入长命令序列的手段。举个例子，如果我们添加alias lm="ls -l — more"到文件 /.bashrc中，那么每次在命令行中键入lm就可以自动转换为ls -l — more。这可以让你在命令行上少敲好多次，而且也可以避免记忆复杂的命令和繁多的选项。设置alias rm="rm -i"(删除的时候提示)，可以让你在犯了错误之后也不用悲伤，因为它可以让你避免意外删除重要文件。

在脚本中，别名就没那么重要了。如果把别名机制想象成C预处理器的某些功能的话，就很形象，比如说宏扩展，但不幸的是，Bash不能在别名中扩展参数。[\[1\]](#) 而且在脚本中，别名不能够用在“混合型结构”中，比如if/then结构，循环，和函数。还有一个限制，别名不能递归扩展。绝大多数情况下，我们期望别名能够完成的工作，都能够用函数更高效的完成。

例子24-1. 用在脚本中的别名

```
1 #!/bin/bash
2 # alias.sh
3
4 shopt -s expand_aliases
5 # 必须设置这个选项，否则脚本不会打开别名功能。
6
7
8 # 首先，来点有趣的。
9 alias Jesse_James= \
10 'echo "\"Alias Jesse James\" was a 1959 comedy starring Bob Hope."'
11 Jesse_James
12
13 echo; echo; echo;
14
15 alias ll="ls -l"
16 # 可以使用单引号(')或双引号(")来定义一个别名。
17
18 echo "Trying aliased \"ll\":"
19 ll /usr/X11R6/bin/mk*    ## 别名工作了。
20
21 echo
22
23 directory=/usr/X11R6/bin/
24 prefix=mk*  # 看一下通配符会不会引起麻烦。
25 echo "Variables \"directory\" + \"prefix\" = $directory$prefix"
26 echo
27
28 alias lll="ls -l $directory$prefix"
```

```
30 echo "Trying aliased \"lll\":"  
31 lll          # 详细列出/usr/X11R6/bin目录下所有以mk开头的文件.  
32 # 别名能处理连接变量 -- 包括通配符 -- o.k.  
33  
34  
35  
36  
37 TRUE=1  
38  
39 echo  
40  
41 if [ $TRUE ]  
42 then  
43     alias rr="ls -l"  
44     echo "Trying aliased \"rr\" within if/then statement:"  
45     rr /usr/X11R6/bin/mk*    ## 产生错误信息!  
46     # 别名不能在混合结构中使用.  
47     echo "However, previously expanded alias still recognized:"  
48     ll /usr/X11R6/bin/mk*  
49 fi  
50  
51 echo  
52  
53 count=0  
54 while [ $count -lt 3 ]  
55 do  
56     alias rrr="ls -l"  
57     echo "Trying aliased \"rrr\" within \"while\" loop:"  
58     rrr /usr/X11R6/bin/mk*    ## 这里, 别名也不会扩展.  
59                         # alias.sh: line 57: rrr: command not found  
60     let count+=1  
61 done  
62  
63 echo; echo  
64  
65 alias xyz='cat $0'    # 脚本打印自身内容.  
66                         # 注意是单引号(强引用).  
67 xyz  
68 # 虽然Bash文档建议, 它不能正常运行,  
69 #+ 不过它看起来是可以运行的.  
70 #  
71 # 然而, 就像Steve Jacobson所指出的那样,  
72 #+ 参数"$0"立即扩展成了这个别名的声明.  
73
```

```
74 exit 0
```

unalias命令用来删除之前设置的别名.

例子24-2. unalias: 设置与删除别名

```
1 #!/bin/bash
2 # unalias.sh
3
4 shopt -s expand_aliases # 启用别名扩展.
5
6 alias llm='ls -al | more'
7 llm
8
9 echo
10
11 unalias llm          # 删除别名.
12 llm
13 # 产生错误信息, 因为'llm'已经不再有效了.
14
15 exit 0
```

```
1
2 bash$ ./unalias.sh
3 total 6
4 drwxrwxr-x    2 bozo      bozo        3072 Feb  6 14:04 .
5 drwxr-xr-x   40 bozo      bozo       2048 Feb  6 14:04 ..
6 -rwxr-xr-x    1 bozo      bozo        199 Feb  6 14:04 unalias.sh
7
8 ./unalias.sh: llm: command not found
```

注意事项

1. 然而, 别名好像能够扩展位置参数.

第25章 列表结构

”与列表”和”或列表”结构能够提供一种手段, 这种手段能够用来处理一串连续的命令. 这样就可以有效的替换掉嵌套的if/then结构, 甚至能够替换掉case语句.

把命令连接到一起

与列表

```
1 command-1 && command-2 && command-3 && ... command-n
```

如果每个命令执行后都返回true(0)的话, 那么命令将会依次执行下去. 如果其中的某个命令返回false(非零值)的话, 那么这个命令链就会被打断, 也就是结束执行, (那么第一个返回false的命令, 就是最后一个执行的命令, 其后的命令都不会执行).

例子25-1. 使用”与列表”来测试命令行参数

```
1#!/bin/bash
2# "与列表"
3
4if [ ! -z "$1" ] && echo "Argument #1 = $1" && [ ! -z "$2" ] && \
5echo "Argument #2 = $2"
6then
7    echo "At least 2 arguments passed to script."
8    # 所有连接起来的命令都返回true.
9else
10    echo "Less than 2 arguments passed to script."
11    # 整个命令列表中至少有一个命令返回false.
12fi
13# 注意, "if [ ! -z $1 ]"也可以, 但它是有所假定的等价物.
14#   if [ -n $1 ] 这个不行.
15#   然而, 如果加了引用就行了.
16#   if [ -n "$1" ] 这样就行了.
17#   小心!
18# 最好将你要测试的变量引用起来, 这么做是非常好的习惯.
19
20
21# 下面这段代码与上面代码是等价的,
22# 不过下面这段代码使用的是"纯粹"的if/then结构.
23if [ ! -z "$1" ]
24then
25    echo "Argument #1 = $1"
26fi
27if [ ! -z "$2" ]
28then
29    echo "Argument #2 = $2"
30    echo "At least 2 arguments passed to script."
```

```
31 else
32     echo "Less than 2 arguments passed to script."
33 fi
34 # 这么写的话，行数太多了，没有"与列表"来的精简.
35
36
37 exit 0
```

例子25-2. 使用”与列表”来测试命令行参数的另一个例子

```
1#!/bin/bash
2
3ARGS=1          # 期望的参数个数.
4E_BADARGS=65   # 如果传递的参数个数不正确，那么给出这个退出码.
5
6test $# -ne $ARGS && echo "Usage: `basename $0` $ARGS argument(s)" && \
7exit $E_BADARGS
8# 如果"条件1"测试为true (表示传递给脚本的参数个数不对),
9#+ 则余下的命令会被执行，并且脚本将结束运行.
10
11# 只有当上面的测试条件为false的时候，这行代码才会被执行.
12echo "Correct number of arguments passed to this script."
13
14exit 0
15
16# 为了检查退出码，在脚本结束的时候可以使用"echo $"来查看退出码.
```

当然，与列表也可以给变量设置默认值。

```
1arg1=$@      # 不管怎样，将$arg1设置为命令行参数.
2
3[ -z "$arg1" ] && arg1=DEFAULT
4                      # 如果没有指定命令行参数，则把$arg1设置为DEFAULT.
```

或列表

```
1command-1 || command-2 || command-3 || ... command-n
```

如果每个命令都返回false，那么命令链就会执行下去。一旦有一个命令返回true，命令链就会被打断，也就是结束执行，(第一个返回true的命令将会是最后一个执行的命令)。显然，这和”与列表”完全相反。

例子25-3. 将”或列表”和”与列表”结合使用

```
1#!/bin/bash
2
3# delete.sh, 不是很聪明的文件删除方法.
4# Usage: delete filename
5
6E_BADARGS=65
7
```

```

8 if [ -z "$1" ]
9 then
10   echo "Usage: `basename $0` filename"
11   exit $E_BADARGS # 没有参数? 退出脚本.
12 else
13   file=$1          # 设置文件名.
14 fi
15
16
17 [ ! -f "$file" ] && echo "File \`$file\` not found. \
18 Cowardly refusing to delete a nonexistent file."
19 # 与列表, 在文件不存在时将会给出错误信息.
20 # 注意echo命令使用了一个续行符, 这样下一行的内容, 也会作为echo命令的参数.
21
22 [ ! -f "$file" ] || (rm -f $file; echo "File \`$file\` deleted.")
23 # 或列表, 如果文件存在, 那就删除此文件.
24
25 # 注意, 上边的两个逻辑相反.
26 # 与列表在true的情况下才执行, 或列表在false的时候才执行.
27
28 exit 0

```

►: 如果"或列表"中的第一个命令返回true, 那么,"或列表"中的第一个命令还是会执行.

```

1 # ==> 下面的片断摘自于脚本/etc/rc.d/init.d/single,
2 # ==> 这个脚本由Miquel van Smoorenburg所编写.
3 #+==> 用于展示"与"/"或"列表的使用.
4 # ==> "箭头"注释是由本书作者添加的.
5
6 [ -x /usr/bin/clear ] && /usr/bin/clear
7 # ==> 如果/usr/bin/clear存在, 那么就调用它.
8 # ==> 在调用一个命令前, 检查一下它是否存在.
9 #+==> 这样可以避免错误信息, 和其他愚蠢的结果.
10
11 # ==> . . .
12
13 # 如果他们想在单用户模式下运行某些程序, 可能也会运行它...
14 for i in /etc/rc1.d/S[0-9][0-9]* ; do
15     # 检查一下脚本是否在那里.
16     [ -x "$i" ] || continue
17     # ==> 如果在$PWD中没发现相应的文件,
18     #+==> 则会使用"continue"跳过本次循环.
19
20     # 不接受备份文件, 也不接受由rpm产生的文件.
21     case "$1" in
22         *.rpmsave|*.rpmodig|*.rpmnew|*~|*.orig)

```

```
23         continue;;
24     esac
25     [ "$i" = "/etc/rc1.d/S00single" ] && continue
26     # ==> 设置脚本名，但现在还不执行它。
27     $i start
28 done
29
30 # ==> . . .
```

►: 与列表和或列表的[退出状态码](#)由最后一个命令的退出状态所决定。

可以灵活的将”与”/”或”列表组合在一起，但是这么做的话，会使得逻辑变得很复杂，并且需要经过仔细的测试。

```
1 false && true || echo false      # false
2
3 # 与下面的结果相同
4 ( false && true ) || echo false    # false
5 # But *not*
6 false && ( true || echo false )    # (没有输出)
7
8 # 注意，以从做到右的顺序进行分组与求值，
9 #+ 这是因为逻辑操作符"&&"和"||"具有相同的优先级。
10
11 # 最好避免这么复杂的情况，除非你非常了解你到底在做什么。
12
13 # 感谢，S.C.
```

也请参考[例子A-7](#)和[例子7-4](#)，这两个例子展示了如何使用与/或列表来测试变量。

第26章 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语句来指定一个数组。如果想解引用一个数组元素(也就是取值)，可以使大括号，访问形式为\${variable[xx]}。

例子26-1. 简单的数组使用

```
1 #!/bin/bash
2
3
4 area[11]=23
5 area[13]=37
6 area[51]=UFOs
7
8 # 数组成员不一定非得是相邻或连续的。
9
10 # 数组的部分成员可以不被初始化。
11 # 数组中允许空缺元素。
12 # 实际上，保存着稀疏数据的数组("稀疏数组")
13 #+ 在电子表格处理软件中是非常有用的。
14
15
16 echo -n "area[11] = "
17 echo ${area[11]}      # 需要{大括号}。
18
19 echo -n "area[13] = "
20 echo ${area[13]}
21
22 echo "Contents of area[51] are ${area[51]}."
23
24 # 没被初始化的数组成员打印为空值(null变量)。
25 echo -n "area[43] = "
26 echo ${area[43]}
27 echo "(area[43] unassigned)"
28
29 echo
30
31 # 两个数组元素的和被赋值给另一个数组元素
32 area[5]=$(expr ${area[11]} + ${area[13]})'
33 echo "area[5] = area[11] + area[13]"
34 echo -n "area[5] = "
35 echo ${area[5]}
```

```

36
37 area[6]=`expr ${area[11]} + ${area[51]}`
38 echo "area[6] = area[11] + area[51]"
39 echo -n "area[6] = "
40 echo ${area[6]}
41 # 这里会失败，是因为不允许整数与字符串相加。
42
43 echo; echo; echo
44
45 # -----
46 # 另一个数组，"area2".
47 # 另一种给数组变量赋值的方法...
48 # array_name=( XXX YYY ZZZ ... )
49
50 area2=( zero one two three four )
51
52 echo -n "area2[0] = "
53 echo ${area2[0]}
54 # 阿哈，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])。
55
56 echo -n "area2[1] = "
57 echo ${area2[1]}      # [1]是数组的第2个元素。
58 # -----
59
60 echo; echo; echo
61
62 # -----
63 # 第3个数组，"area3".
64 # 第3种给数组元素赋值的方法...
65 # array_name=( [xx]=XXX [yy]=YYY ... )
66
67 area3=( [17]=seventeen [24]=twenty-four)
68
69 echo -n "area3[17] = "
70 echo ${area3[17]}
71
72 echo -n "area3[24] = "
73 echo ${area3[24]}
74 # -----
75
76 exit 0

```

Bash允许把变量当成数组来操作, 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, 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26-2. 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).
5Line[1]="I do not know which to prefer,"
6Line[2]="The beauty of inflections"
7Line[3]="Or the beauty of innuendoes,"
8Line[4]="The blackbird whistling"
9Line[5]="Or just after."
10
11# 出处.
12Attrib[1]="" Wallace Stevens"
13Attrib[2]=""\Thirteen Ways of Looking at a Blackbird\""
14# 这首诗已经是公共版权了(版权已经过期了).
15
16echo
17
18for index in 1 2 3 4 5      # 5行.
19do
20    printf "%s\n" "${Line[index]}"
21done
22
23for index in 1 2            # 出处为2行.
24do
25    printf "%s\n" "${Attrib[index]}"
26done
27
28echo
29
30exit 0
31
32# 练习:
33# -----

```

```
34 # 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，  
35 # 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26-3. 多种数组操作

```
1#!/bin/bash  
2# array-ops.sh: 更多有趣的数组用法。  
3  
4  
5array=( zero one two three four five )  
6# 数组元素 0 1 2 3 4 5  
7  
8echo ${array[0]}      # 0  
9echo ${array:0}        # 0  
10                         # 第一个元素的参数扩展，  
11                         #+ 从位置0(#0)开始(即第一个字符).  
12echo ${array:1}        # ero  
13                         # 第一个元素的参数扩展，  
14                         #+ 从位置1(#1)开始(即第2个字符).  
15  
16echo "-----"  
17  
18echo ${#array[0]}      # 4  
19                         # 第一个数组元素的长度。  
20echo ${#array}          # 4  
21                         # 第一个数组元素的长度。  
22                         # (另一种表示形式)  
23  
24echo ${#array[1]}      # 3  
25                         # 第二个数组元素的长度。  
26                         # Bash中的数组是从0开始索引的。  
27  
28echo ${#array[*]}      # 6  
29                         # 数组中的元素个数。  
30echo ${#array[@]}      # 6  
31                         # 数组中的元素个数。  
32  
33echo "-----"  
34  
35array2=( [0]="first element" [1]="second element" [3]="fourth element" )  
36  
37echo ${array2[0]}        # 第一个元素  
38echo ${array2[1]}        # 第二个元素  
39echo ${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  
43  
44 exit 0
```

大部分标准字符串操作都可以用于数组中。

例子26-4. 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# 本脚本由Michael Zick所编写。  
4# 授权使用在本书中。  
5  
6# 一般来说，任何类似于${name ... }(这种形式)的字符串操作  
7#+ 都能够应用于数组中的所有字符串元素，  
8#+ 比如说${name[@] ... }或${name[*] ... }这两种形式。  
9  
10  
11arrayZ=( one two three four five five )  
12  
13echo  
14  
15# 提取尾部的子串  
16echo ${arrayZ[@]:0}      # one two three four five five  
17                      # 所有元素。  
18  
19echo ${arrayZ[@]:1}      # two three four five five  
20                      # element[0]后边的所有元素。  
21  
22echo ${arrayZ[@]:1:2}    # two three  
23                      # 只提取element[0]后边的两个元素。  
24  
25echo "-----"  
26  
27# 子串删除  
28# 从字符串的开头删除最短的匹配，  
29#+ 匹配的子串也可以是正则表达式。  
30  
31echo ${arrayZ[@]##f*r}   # one two three five five  
32                      # 匹配将应用于数组的所有元素。  
33                      # 匹配到了"four"，并且将它删除。  
34  
35# 从字符串的开头删除最长的匹配  
36echo ${arrayZ[@]##t*e}   # one two four five five  
37                      # 匹配将应用于数组的所有元素。
```

```
38 # 匹配到了"three", 并且将它删除.
39
40 # 从字符串的结尾删除最短的匹配
41 echo ${arrayZ[@]%-*e} # one two t four five five
42 # 匹配将应用于数组的所有元素.
43 # 匹配到了"hree", 并且将它删除.
44
45 # 从字符串的结尾删除最长的匹配
46 echo ${arrayZ[@]%%t*e} # one two four five five
47 # 匹配将应用于数组的所有元素.
48 # 匹配到了"three", 并且将它删除.
49
50 echo "-----"
51
52 # 子串替换
53
54 # 第一个匹配到的子串将会被替换
55 echo ${arrayZ[@]/fiv/XYZ} # one two three four XYZe XYZe
56 # 匹配将应用于数组的所有元素.
57
58 # 所有匹配到的子串都会被替换
59 echo ${arrayZ[@]//iv/YY} # one two three four fYYe fYYe
60 # 匹配将应用于数组的所有元素.
61
62 # 删除所有的匹配子串
63 # 如果没有指定替换字符串的话, 那就意味着, 删除,
64 echo ${arrayZ[@]//fi/} # one two three four ve ve
65 # 匹配将应用于数组的所有元素.
66
67 # 替换字符串前端子串
68 echo ${arrayZ[@]#/fi/XY} # one two three four XYve XYve
69 # 匹配将应用于数组的所有元素.
70
71 # 替换字符串后端子串
72 echo ${arrayZ[@]%/ve/ZZ} # one two three four fiZZ fiZZ
73 # 匹配将应用于数组的所有元素.
74
75 echo ${arrayZ[@]%/o/XX} # one twXX three four five five
76 # 为什么?
77
78 echo "-----"
79
80
81 # 在将处理后的结果发送到awk(或者其他处理工具)之前 --
82 # 回忆一下:
```

```

83 # $( ...) 是命令替换.
84 # 函数作为子进程运行.
85 # 函数结果输出到stdout.
86 # 用read来读取函数的stdout.
87 # 使用name[@]表示法指定了一个"for-each"操作.
88
89 newstr() {
90     echo -n "!!!"
91 }
92
93 echo ${arrayZ[@]//%/$(newstr)}
94 # on!!! two thre!!! four fiv!!! fiv!!!
95 # Q.E.D: 替换动作实际上是一个'赋值'.
96
97 # 使用"For-Each"形式的
98 echo ${arrayZ[@]//*/$(newstr optional_arguments)}
99 # 现在, 如果Bash只将匹配到的子串作为$0
100 #+ 传递给将被调用的函数 . .
101
102 echo
103
104 exit 0

```

命令替换可以构造数组的独立元素. (译者注: 换句话说, 就是命令替换也能够给数组赋值.)

例子26-5. 将脚本的内容赋值给数组

```

1#!/bin/bash
2# script-array.sh: 将这个脚本的内容赋值给数组.
3# 这个脚本的灵感来自于Chris Martin的e-mail(感谢!).
4
5script_contents=( $(cat "$0") ) # 将这个脚本的内容($0)
6                                #+ 赋值给数组.
7
8for element in $($(( ${#script_contents[@]} - 1)))
9do
10    # ${#script_contents[@]}
11    # 表示数组元素的个数.
12    #
13    # 一个小问题:
14    # 为什么必须使用seq 0?
15    # 用seq 1来试一下.
16    echo -n "${script_contents[$element]}"
17    # 在同一行上显示脚本中每个域的内容.
18    echo -n " -- "
19done
20echo

```

```
21
22 exit 0
23
24 # 练习：
25 # -----
26 # 修改这个脚本,
27 #+ 让这个脚本能够按照它原本的格式输出,
28 #+ 连同空白, 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26-6. 一些数组专用的小道具

```
1#!/bin/bash
2
3declare -a colors
4# 脚本中所有的后续命令都会把
5#+ 变量"colors"看作数组.
6
7echo "Enter your favorite colors (separated from each other by a space)."
8
9read -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
13echo
14
15element_count=${#colors[@]}
16# 提取数组元素个数的特殊语法.
17# 用element_count=${#colors[*]}也一样.
18#
19# "@"变量允许在引用中存在单词分割(word splitting)
20#+ (依靠空白字符来分隔变量).
21#
22# 这就好像"${@}"和"${*}"
23#+ 在位置参数中的所表现出来的行为一样.
24
25index=0
26
27while [ "$index" -lt "$element_count" ]
28do    # 列出数组中的所有元素.
29    echo ${colors[$index]}
30    let "index = $index + 1"
31    # 或:
32    #     index+=1
33    # 如果你运行的Bash版本是3.1以后的话, 才支持这种语法.
```

```

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 # (感谢, S.C.)
44
45 echo
46
47 # 再次列出数组中的所有元素, 不过这次的做法更优雅.
48 echo ${colors[@]}          # 用echo ${colors[*]}也行.
49
50 echo
51
52 # "unset"命令即可以删除数组数据, 也可以删除整个数组.
53 unset colors[1]           # 删除数组的第2个元素.
54                                     # 作用等效于 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[*] 或
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
63 exit 0

```

正如我们在前面例子中所看到的, \${array_name[@]}或\${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\${#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26-7. 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对这个例子所作的扩展.
6
7
8# 空数组与包含有空元素的数组, 这两个概念不同.
9
10array0=( first second third )

```

```

11 array1=( '') # "array1"包含一个空元素.
12 array2=( ) # 没有元素 . . . "array2"为空.
13
14 echo
15 ListArray()
16 {
17 echo
18 echo "Elements in array0: ${array0[@]}"
19 echo "Elements in array1: ${array1[@]}"
20 echo "Elements in array2: ${array2[@]}"
21 echo
22 echo "Length of first element in array0 = ${#array0}"
23 echo "Length of first element in array1 = ${#array1}"
24 echo "Length of first element in array2 = ${#array2}"
25 echo
26 echo "Number of elements in array0 = ${#array0[*]}" # 3
27 echo "Number of elements in array1 = ${#array1[*]}" # 1 (惊奇!)
28 echo "Number of elements in array2 = ${#array2[*]}" # 0
29 }
30
31 # =====
32
33 ListArray
34
35 # 尝试扩展这些数组.
36
37 # 添加一个元素到这个数组.
38 array0=( "${array0[@]}" "new1" )
39 array1=( "${array1[@]}" "new1" )
40 array2=( "${array2[@]}" "new1" )
41
42 ListArray
43
44 # 或
45 array0[$#array0[*]]="new2"
46 array1[$#array1[*]]="new2"
47 array2[$#array2[*]]="new2"
48
49 ListArray
50
51 # 如果你按照上边的方法对数组进行扩展的话; 数组比较象, 栈,
52 # 上边的操作就是, 压栈,
53 # 栈, 高, 为:
54 height=${#array2[@]}
55 echo

```

```

56 echo "Stack height for array2 = $height"
57
58 # ,出栈,就是:
59 unset array2[$#array2[@]-1]    # 数组从0开始索引,
60 height=${#array2[@]}           #+ 这意味着第一个数组下标为0.
61 echo
62 echo "POP"
63 echo "New stack height for array2 = $height"
64
65 ListArray
66
67 # 只列出数组array0的第二个和第三个元素.
68 from=1             # 从0开始索引.
69 to=2               #
70 array3=( ${array0[@]:1:2} )
71 echo
72 echo "Elements in array3: ${array3[@]}"
73
74 # 处理方式就像是字符串(字符数组).
75 # 试试其他的"字符串"形式.
76
77 # 替换:
78 array4=( ${array0[@]/second/2nd} )
79 echo
80 echo "Elements in array4: ${array4[@]}"
81
82 # 替换掉所有匹配通配符的字符串.
83 array5=( ${array0[@]//new?/old} )
84 echo
85 echo "Elements in array5: ${array5[@]}"
86
87 # 当你开始觉得对此有把握的时候 . . .
88 array6=( ${array0[@]##*new} )
89 echo # 这个可能会让你感到惊奇.
90 echo "Elements in array6: ${array6[@]}"
91
92 array7=( ${array0[@]#new1} )
93 echo # 数组array6之后就没有惊奇了.
94 echo "Elements in array7: ${array7[@]}"
95
96 # 看起来非常像 . . .
97 array8=( ${array0[@]/new1/} )
98 echo
99 echo "Elements in array8: ${array8[@]}"
100

```

```

101 # 所以，让我们怎么形容呢？
102
103 # 对数组var[@]中的每个元素
104 #+ 进行连续的字符串操作。
105 # 因此：如果结果是长度为0的字符串，
106 #+ Bash支持字符串向量操作，
107 #+ 元素会在结果赋值中消失不见。
108
109 # 一个问题，这些字符串是强引用还是弱引用？
110
111 zap='new*'
112 array9=( ${array0[@]//$zap} )
113 echo
114 echo "Elements in array9: ${array9[@]}"
115
116 # 当你还在考虑，你身在Kansas州何处时 . .
117 array10=( ${array0[@]#$zap} )
118 echo
119 echo "Elements in array10: ${array10[@]}"
120
121 # 比较array7和array10.
122 # 比较array8和array9.
123
124 # 答案：必须是弱引用。
125
126 exit 0

```

`${array_name[@]}` 和 `${array_name[*]}` 的关系非常类似于 `$@` 和 `$*`。这种数组用法用处非常广泛。

```

1 # 复制一个数组。
2 array2=( "${array1[@]}" )
3 # 或
4 array2="${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] 没赋值
12 array1[2]=2
13 array2=( "${array1[@]}" )           # 拷贝它？
14
15 echo ${array2[0]}      # 0
16 echo ${array2[2]}      # (null)，应该是2

```

```

17 # -----
18
19
20
21 # 添加一个元素到数组.
22 array=( "${array[@]}" "new element" )
23 # 或
24 array[$#]=new element
25
26 # 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 lename=sample_file
2 #!/bin/bash
3
4 filename=sample_file
5
6
7 #         cat sample_file
8 #
9 #         1 a b c
10 #        2 d e fg
11
12
13 declare -a array1
14
15 array1=( `cat "$filename"` )           # 将$filename的内容
16 #       List file to stdout          #+ 加载到数组array1.
17 #
18 # array1=( `cat "$filename" | tr '\n' ' '` )
19 #           把文件中的换行替换为空格.
20 # 其实这么做是没必要的, Not necessary because Bash does word splitting,
21 #+ 因为Bash在做单词分割(word splitting)的时候, 将会把换行转换为空格.
22
23 echo ${array1[@]}           # 打印数组.
24 #                         1 a b c 2 d e fg
25 #
26 # 文件中每个被空白符分隔的"单词"
27 #+ 都被保存到数组的一个元素中.
28
29 element_count=${#array1[*]}
30 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26-8. 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
14 #+ 出现在本书中的一个例子.
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 # 在2.04, 2.05a和2.05b版本的Bash中,
23 #+ 这两种方法都会丢弃数组中的"空引用"(null值)元素.
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他方法
29 #+ 也能很好的完成任务.
30
31 declare -a bigOne=( /dev/* )
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
39
40 echo
41 echo '-- testing: =({array[@]} ) --'
42 times
43 declare -a bigTwo=( ${bigOne[@]} )
```

```

44 #           ^
45 times
46
47 echo
48 echo '- - testing: ${array[@]} - -'
49 times
50 declare -a bigThree=${bigOne[@]}
51 # 这次没用括号.
52 times
53
54 # 正如Stephane Chazelas所指出的，通过比较，
55 #+ 可以了解到第二种格式的赋值比第三或第四种形式更快.
56 #
57 # William Park解释：
58 #+ 数组bigTwo是作为一个单个字符串被赋值的，
59 #+ 而数组bigThree，则是一个元素一个元素进行的赋值.
60 # 所以，实质上是：
61 #           bigTwo=( [0]="...." )
62 #           bigThree=( [0]="..." [1]="..." [2]="..." ... )
63
64
65 # 在本书的例子中，我还是会继续使用第一种形式，
66 #+ 因为我认为这种形式更有利于将问题阐述清楚.
67
68 # 在我所使用的例子中，在其中复用的部分，
69 #+ 还是使用了第二种形式，那是因为这种形式更快.
70
71 # MSZ：很抱歉早先的疏忽(译者：应是指本书的老版本).
72
73
74 # 注意事项：
75 # -----
76 # 31行和43行的"declare -a"语句其实不是必需的，
77 #+ 因为Array=( ...)形式
78 #+ 只能用于数组赋值.
79 # 然而，如果省略这些声明的话，
80 #+ 会导致脚本后边的相关操作变慢.
81 # 试一下，看看发生了什么.
82
83 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26-9. 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 # 已通过作者授权，可以在本书中使用 .
6
7 # 如何"通过名字传值&通过名字返回"(译者注：这里可以理解为C中的"数组指针"，
8 # 或C++中的"数组引用")
9 #+ 或者"建立自己的赋值语句".
10
11
12 CpArray_Mac() {
13
14 # 建立赋值命令
15
16     echo -n 'eval '
17     echo -n "$2"                      # 目的参数名
18     echo -n '=( ${'
19     echo -n "$1"                      # 原参数名
20     echo -n '[@]}' )
21
22 # 上边这些语句会构成一条命令 .
23 # 这仅仅是形式上的问题 .
24 }
25
26 declare -f CopyArray           # 函数"指针"
27 CopyArray=CpArray_Mac          # 构造语句
28
29 Hype()
30 {
31
32 # 需要连接的数组名为$1 .
33 # (把这个数组与字符串"Really Rocks"结合起来，形成一个新数组 .)
34 # 并将结果从数组$2中返回 .
35
36     local -a TMP
37     local -a hype=( Really Rocks )
38
39     $($CopyArray $1 TMP)
40     TMP=( ${TMP[@]} ${hype[@]} )
41     $($CopyArray TMP $2)
42 }
43
44 declare -a before=( Advanced Bash Scripting )
45 declare -a after
46
47 echo "Array Before = ${before[@]}"
```

```

48
49 Hype before after
50
51 echo "Array After = ${after[@]}"
52
53 # 连接的太多了?
54
55 echo "What ${after[@]:3:2}?"
56
57 declare -a modest=( ${after[@]:2:1} ${after[@]:3:2} )
58 # ----- 子串提取 -----
59
60 echo "Array Modest = ${modest[@]}"
61
62 # 'before'发生了什么变化么?
63
64 echo "Array Before = ${before[@]}"
65
66 exit 0

```

例子26-10. 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
11 # 数组操作是Bash特有的属性.
12 # 传统的UNIX /bin/sh缺乏类似的功能.
13
14
15 # 将这个脚本的输出通过管道传递给'more',
16 #+ 这么做的目的是防止输出的内容超过终端能够显示的范围.
17
18
19 # 依次使用下标.
20 declare -a array1=( zero1 one1 two1 )
21 # 数组中存在空缺的元素([1]未定义).
22 declare -a array2=( [0]=zero2 [2]=two2 [3]=three2 )
23
24 echo

```

```

25 echo '- Confirm that the array is really subscript sparse. -'
26 echo "Number of elements: 4"          # 仅仅为了演示, 所以就写死了.
27 for (( i = 0 ; i < 4 ; i++ ))
28 do
29     echo "Element [$i]: ${array2[$i]}"
30 done
31 # 也可以参考一个更通用的例子, basics-reviewed.bash.
32
33
34 declare -a dest
35
36 # 将两个数组合并(附加)到第3个数组.
37 echo
38 echo 'Conditions: Unquoted, default IFS, All-Elements-Of operator'
39 echo '- Undefined elements not present, subscripts not maintained. -'
40 # # 那些未定义的元素不会出现; 组合时会丢弃这些元素.
41
42 dest=( ${array1[@]} ${array2[@]} )
43 # dest=${array1[@]}${array2[@]}      # 令人奇怪的结果, 或许是个bug.
44
45 # 现在, 打印结果.
46 echo
47 echo '-- Testing Array Append --'
48 cnt=${#dest[@]}
49
50 echo "Number of elements: $cnt"
51 for (( i = 0 ; i < cnt ; i++ ))
52 do
53     echo "Element [$i]: ${dest[$i]}"
54 done
55
56 # 将数组赋值给一个数组中的元素(两次).
57 dest[0]=${array1[@]}
58 dest[1]=${array2[@]}
59
60 # 打印结果.
61 echo
62 echo '-- Testing modified array --'
63 cnt=${#dest[@]}
64
65 echo "Number of elements: $cnt"
66 for (( i = 0 ; i < cnt ; i++ ))
67 do
68     echo "Element [$i]: ${dest[$i]}"
69 done

```

```

70
71 # 检查第二个元素的修改状况.
72 echo
73 echo '- - Reassign and list second element - -'
74
75 declare -a subArray=${dest[1]}
76 cnt=${#subArray[@]}
77
78 echo "Number of elements: $cnt"
79 for (( i = 0 ; i < cnt ; i++ ))
80 do
81     echo "Element [$i]: ${subArray[$i]}"
82 done
83
84 # 如果你使用'=${ ... }'形式
85 #+ 将一个数组赋值到另一个数组的一个元素中,
86 #+ 那么这个数组的所有元素都会被转换为一个字符串,
87 #+ 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).
88
89 # 如果原来数组中的所有元素都不包含空白符 . .
90 # 如果原来的数组下标都是连续的 . .
91 # 那么我们就可以将原来的数组进行恢复.
92
93 # 从修改过的第二个元素中, 将原来的数组恢复出来.
94 echo
95 echo '- - Listing restored element - -'
96
97 declare -a subArray=( ${dest[1]} )
98 cnt=${#subArray[@]}
99
100 echo "Number of elements: $cnt"
101 for (( i = 0 ; i < cnt ; i++ ))
102 do
103     echo "Element [$i]: ${subArray[$i]}"
104 done
105 echo '- - Do not depend on this behavior. - -'
106 echo '- - This behavior is subject to change - -'
107 echo '- - in versions of Bash newer than version 2.05b - -'
108
109 # MSZ: 抱歉, 之前混淆了一些要点(译者注: 指的是本书以前的版本).
110
111 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26-11. 一位老朋友：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最底部。
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 {
17     # 交换数组中的两个元素。
18     local temp=${Countries[$1]} # 临时保存
19                         #+ 要交换的那个元素。
20     Countries[$1]="${Countries[$2]}"
21     Countries[$2]=$temp
22
23     return
24 }
25
26 declare -a Countries # 声明数组，
27                         #+ 此处是可选的，因为数组在下面被初始化。
28
29 # 我们是否可以使用转义符(\)
30 #+ 来将数组元素的值放在不同的行上？
31 # 可以。
32
33 Countries=(Netherlands Ukraine Zaire Turkey Russia Yemen Syria \
34 Brazil Argentina Nicaragua Japan Mexico Venezuela Greece England \
35 Israel Peru Canada Oman Denmark Wales France Kenya \
36 Xanadu Qatar Liechtenstein Hungary)
37
38 # "Xanadu"虚拟出来的世外桃源。
39 #+
40
41
42 clear                  # 开始之前的清屏动作。
43
```

```

44 echo "0: ${Countries[*]}" # 从索引0开始列出整个数组.
45
46 number_of_elements=${#Countries[@]}
47 let "comparisons = $number_of_elements - 1"
48
49 count=1 # 传递数字.
50
51 while [ "$comparisons" -gt 0 ]           # 开始外部循环
52 do
53
54     index=0 # 在每轮循环开始之前, 重置索引.
55
56     while [ "$index" -lt "$comparisons" ] # 开始内部循环
57     do
58         if [ ${Countries[$index]} > ${Countries['expr $index + 1']} ]
59             # 如果原来的排序次序不对...
60             # 回想一下, 在单括号中,
61             #+ >是ASCII码的比较操作符.
62
63         # if [[ ${Countries[$index]} > ${Countries['expr $index + 1']} ]]
64         #+ 这样也行.
65         then
66             exchange $index 'expr $index + 1' # 交换.
67         fi
68         let "index += 1" # 或者, index+=1 在Bash 3.1之后的版本才能这么用.
69     done # 内部循环结束
70
71 # -----
72 # Paulo Marcel Coelho Aragao建议我们可以使用更简单的for循环.
73 #
74 # for (( last = $number_of_elements - 1 ; last > 1 ; last-- ))
75 # do
76 #     for (( i = 0 ; i < last ; i++ ))
77 #         do
78 #             [[ "${Countries[$i]}" > "${Countries[$((i+1))]}" ]] \
79 #                 && exchange $i $((i+1))
80 #         done
81 #     done
82 # -----
83
84
85 let "comparisons -= 1" # 因为最"重"的元素到了底部,
86                         #+ 所以每轮我们可以少做一次比较.
87
88 echo

```

```

89 echo "$count: ${Countries[@]}"\# 每轮结束后，都打印一次数组.
90 echo
91 let "count += 1"\# 增加传递计数.
92
93 done\# 外部循环结束
94\# 至此，全部完成.
95
96 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
7AnArray=( $(ls --inode --ignore-backups --almost-all \
8    --directory --full-time --color=none --time=status \
9    --sort=time -l ${PWD} ) ) # 命令及选项.
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
13SubArray=( ${AnArray[@]:11:1} ${AnArray[@]:6:5} )
14# 这个数组有六个元素:
15#+ SubArray=( [0]=${AnArray[11]} [1]=${AnArray[6]} [2]=${AnArray[7]} \
16#     [3]=${AnArray[8]} [4]=${AnArray[9]} [5]=${AnArray[10]} )
17#
18# Bash数组是字符串(char *)类型
19#+ 的(循环)链表.
20# 因此，这不是真正意义上的嵌套数组，
21#+ 只不过功能很相似而已.
22
23echo "Current directory and date of last status change:"
24echo "${SubArray[@]}"
25
26exit 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26-12. 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
10 ARRAY1=(
11     VAR1_1=value11
12     VAR1_2=value12
13     VAR1_3=value13
14 )
15
16 ARRAY2=(
17     VARIABLE="test"
18     STRING="VAR1=value1 VAR2=value2 VAR3=value3"
19     ARRAY21=${ARRAY1[*]}
20 ) # 将ARRAY1嵌套到这个数组中。
21
22 function print () {
23     OLD_IFS="$IFS"
24     IFS=$'\n'      # 这么做是为了每行
25             #+ 只打印一个数组元素。
26     TEST1="ARRAY2[*]"
27     local ${!TEST1} # 删除这一行，看看会发生什么？
28     # 间接引用。
29     # 这使得$TEST1
30     #+ 只能够在函数内被访问。
31
32
33     # 让我们看看还能干点什么。
34     echo
35     echo "\$TEST1 = $TEST1"      # 仅仅是变量名字。
36     echo; echo
37     echo "{$TEST1} = ${!TEST1}" # 变量内容。
38             # 这就是
39             #+ 间接引用的作用。
40     echo
41     echo "-----"; echo
42     echo
43
44
45     # 打印变量
46     echo "Variable VARIABLE: $VARIABLE"
47
48     # 打印一个字符串元素
49     IFS="$OLD_IFS"
50     TEST2="STRING[*]"

```

```

51     local ${!TEST2}      # 间接引用(同上).
52     echo "String element VAR2: $VAR2 from STRING"
53
54     # 打印一个数组元素
55     TEST2="ARRAY21[*]"
56     local ${!TEST2}      # 间接引用(同上).
57     echo "Array element VAR1_1: $VAR1_1 from ARRAY21"
58 }
59
60 print
61 echo
62
63 exit 0
64
65 # 脚本作者注,
66 #+ "你可以很容易的将其扩展成一个能创建hash的Bash脚本."
67 # (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26-13. 复杂的数组应用: 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
10LOWER_LIMIT=1      # 从1开始.
11UPPER_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
14PRIME=1
15NON_PRIME=0
16
17let SPLIT=UPPER_LIMIT/2
18# 优化:
19# 只需要测试中间到最大的值(为什么?).
20# (译者注: 这个变量在脚本正文并没有被使用, 仅仅在107行之后的优化部分才使用.)
21
22declare -a Primes
23# Primes[] 是个数组.

```

```
24
25
26 initialize ()
27 {
28 # 初始化数组 .
29
30 i=$LOWER_LIMIT
31 until [ "$i" -gt "$UPPER_LIMIT" ]
32 do
33     Primes[i]=$PRIME
34     let "i += 1"
35 done
36 # 假定所有数组成员都是需要检查的(素数)
37 #+ 直到检查完成 .
38 }
39
40 print_primes ()
41 {
42 # 打印出所有数组Primes[]中被标记为素数的元素 .
43
44 i=$LOWER_LIMIT
45
46 until [ "$i" -gt "$UPPER_LIMIT" ]
47 do
48
49     if [ "${Primes[i]}" -eq "$PRIME" ]
50     then
51         printf "%8d" $i
52         # 每个数字打印前先打印8个空格 , 在偶数列才打印 .
53     fi
54
55     let "i += 1"
56
57 done
58
59 }
60
61 sift () # 查出非素数 .
62 {
63
64 let i=$LOWER_LIMIT+1
65 # 我们都知道1是素数 , 所以我们从2开始 .
66 # (译者注: 从2开始并不是由于1是素数 , 而是因为要去掉以后每个数的倍数 ,
67 # 感谢网友KevinChen.)
68 until [ "$i" -gt "$UPPER_LIMIT" ]
```

```
69 do
70
71 if [ "${Primes[i]}" -eq "$PRIME" ]
72 # 不要处理已经过滤过的数字(被标识为非素数).
73 then
74
75 t=$i
76
77 while [ "$t" -le "$UPPER_LIMIT" ]
78 do
79     let "t += $i "
80     Primes[t]=$NON_PRIME
81     # 标识为非素数.
82 done
83
84 fi
85
86 let "i += 1"
87 done
88
89
90 }
91
92
93 # =====
94 # main ()
95 # 继续调用函数.
96 initialize
97 sift
98 print_primes
99 # 这里就是被称为结构化编程的东西.
100 # =====
101
102 echo
103
104 exit 0
105
106
107
108 # ----- #
109 # 因为前面的'exit'语句, 所以后边的代码不会运行.
110
111 # 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,
112 #+ 这个版本可以运行的快一些.
113
```

```

114 # 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).
115
116 UPPER_LIMIT=$1          # 来自于命令行.
117 let SPLIT=UPPER_LIMIT/2    # 从中间值到最大值.
118
119 Primes=( '' $(seq $UPPER_LIMIT) )
120
121 i=1
122 until ((( i += 1 ) > SPLIT ))  # 仅需要从中间值检查.
123 do
124     if [[ -n ${Primes[i]} ]]
125     then
126         t=$i
127         until ((( t += i ) > UPPER_LIMIT ))
128         do
129             Primes[t]=
130             done
131         fi
132     done
133 echo ${Primes[*]}
134
135 exit 0

```

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-16](#)，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26-14. 模拟一个下推堆栈

```

1#!/bin/bash
2# stack.sh: 模拟下推堆栈
3
4# 类似于CPU栈，下推堆栈依次保存数据项，
5#+ 但是取数据时，却反序进行，后进先出。
6
7BP=100          # 栈数组的基址指针.
8                  # 从元素100开始.
9
10SP=$BP          # 栈指针.
11                  # 将其初始化为栈"基址"(栈底).
12
13Data=          # 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
17declare -a stack

```

```
18
19
20 push()           # 压栈.
21 {
22 if [ -z "$1" ]   # 没有可压入的数据项?
23 then
24     return
25 fi
26
27 let "SP -= 1"    # 更新栈指针.
28 stack[$SP]="$1"
29
30 return
31 }
32
33 pop()           # 从栈中弹出数据项.
34 {
35 Data=          # 清空保存数据项的中间变量.
36
37 if [ "$SP" -eq "$BP" ]  # 栈空?
38 then
39     return
40 fi
41             # 这使得SP不会超过100,
42             #+ 例如, 这可以防止堆栈失控.
43
44 Data=${stack[$SP]}
45 let "SP += 1"    # 更新栈指针.
46 return
47
48 status_report()  # 打印当前状态.
49 {
50 echo -----
51 echo "REPORT"
52 echo "Stack Pointer = $SP"
53 echo "Just popped \\"$Data"\ off the stack."
54 echo -----
55 echo
56 }
57
58
59 # =====
60 # 现在, 来点乐子.
61
62 echo
```

```
63
64 # 看你是否能从空栈里弹出数据项来.
65 pop
66 status_report
67
68 echo
69
70 push garbage
71 pop
72 status_report      # 压入garbage, 弹出garbage.
73
74 value1=23; push $value1
75 value2=skidoo; push $value2
76 value3=FINAL; push $value3
77
78 pop          # FINAL
79 status_report
80 pop          # skidoo
81 status_report
82 pop          # 23
83 status_report # 后进, 先出!
84
85 # 注意: 栈指针在压栈时减,
86 #+ 在弹出时加.
87
88 echo
89
90 exit 0
91
92 # =====
93
94
95 # 练习:
96 # -----
97
98 # 1) 修改"push()"函数,
99 # + 使其调用一次就能够压入多个数据项.
100
101 # 2) 修改"pop()"函数,
102 # + 使其调用一次就能弹出多个数据项.
103
104 # 3) 给那些有临界操作的函数添加出错检查.
105 #      说不明白一些, 就是让这些函数返回错误码,
106 #      + 返回的错误码依赖于操作是否成功完成,
107 #      + 如果没有成功完成, 那么就需要启动合适的处理动作.
```

```
108  
109 # 4) 以这个脚本为基础,  
110 # + 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26-15. 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# 序列的头20项，如下所示：  
10# 1 1 2 3 3 4 5 5 6 6 6 8 8 8 10 9 10 11 11 12  
11  
12# 请参考相关书籍，Hofstadter的，"_Goedel, Escher,  
13# Bach: An Eternal Golden Braid_"，第137页。  
14  
15  
16LIMIT=100      # 需要计算的数列长度。  
17LINEWIDTH=20    # 每行打印的个数。  
18  
19Q[1]=1          # 数列的头两项都为1。  
20Q[2]=1  
21  
22echo  
23echo "Q-series [$LIMIT terms]:"  
24echo -n "${Q[1]} "           # 输出数列头两项。  
25echo -n "${Q[2]} "  
26  
27for ((n=3; n <= $LIMIT; n++)) # C风格的循环条件。  
28do  # Q[n] = Q[n - Q[n-1]] + Q[n - Q[n-2]]  当n>2时  
29# 需要将表达式拆开，分步计算，  
30#+ 因为Bash不能够很好的处理复杂数组的算术运算。  
31  
32let "n1 = $n - 1"          # n-1  
33let "n2 = $n - 2"          # n-2  
34  
35t0='expr $n - ${Q[n1]}'  # n - Q[n-1]  
36t1='expr $n - ${Q[n2]}'  # n - Q[n-2]  
37
```

```

38 T0=${Q[t0]}          # Q[n - Q[n-1]]
39 T1=${Q[t1]}          # Q[n - Q[n-2]]
40
41 Q[n]=`expr $T0 + $T1`    # Q[n - Q[n-1]] + Q[n - Q[n-2]]
42 echo -n "${Q[n]} "
43
44 if [ `expr $n % $LINEWIDTH` -eq 0 ]      # 格式化输出.
45 then   #      ^ 取模操作
46   echo # 把每行都拆为20个数字的小块.
47 fi
48
49 done
50
51 echo
52
53 exit 0
54
55 # 这是Q-series的一个迭代实现.
56 # 更直接明了的实现是使用递归, 请读者作为练习完成.
57 # 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.

```

=====

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26-16. 模拟一个二维数组, 并使他倾斜

```

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
7Rows=5
8Columns=5
9# 5 X 5 的数组.
10
11declare -a alpha      # char alpha [Rows] [Columns];
12                      # 没必要声明. 为什么?
13
14load_alpha ()
15{
16local rc=0
17local index
18
19for 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
20do    # 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][$column]
26 let "rc += 1"
27 done
28
29 # 更简单的方法:
30 #+ declare -a alpha=( A B C D E F G H I J K L M N O P Q R S T U V W X Y )
31 #+ 但是如果写的话, 就缺乏二维数组的"风味"了.
32 }
33
34 print_alpha ()
35 {
36 local row=0
37 local index
38
39 echo
40
41 while [ "$row" -lt "$Rows" ]    # 以"行序为主"进行打印:
42 do
43                         #+ 行号不变(外层循环),
44                         #+ 列号进行增长. (译者注: 就是按行打印)
45     local column=0
46
47     echo -n "          "           # 按照行方向打印"正方形"数组.
48
49     while [ "$column" -lt "$Columns" ]
50     do
51         let "index = $row * $Rows + $column"
52         echo -n "${alpha[index]} "  # alpha[$row][$column]
53         let "column += 1"
54     done
55
56     let "row += 1"
57     echo
58
59 done
60
61 # 更简单的等价写法:
62 #     echo ${alpha[*]} | xargs -n $Columns
63
64 echo
65 }
66 filter ()      # 过滤掉负的数组下标.

```

```
67 {
68
69 echo -n " " # 产生倾斜.
70         # 解释一下, 这是怎么做到的.
71
72 if [[ "$1" -ge 0 && "$1" -lt "$Rows" && "$2" -ge 0 && "$2" -lt "$Columns" ]]
73 then
74     let "index = $1 * $Rows + $2"
75     # 现在, 按照旋转方向进行打印.
76     echo -n "${alpha[index]}"
77     #           alpha[$row] [$column]
78 fi
79
80 }
81
82
83
84
85 rotate () # 将数组旋转45度 --
86 {          #+ 从左下角进行"平衡".
87 local row
88 local column
89
90 for (( row = Rows; row > -Rows; row-- ))
91 do        # 反向步进数组, 为什么?
92
93 for (( column = 0; column < Columns; column++ ))
94 do
95
96     if [ "$row" -ge 0 ]
97 then
98     let "t1 = $column - $row"
99     let "t2 = $column"
100 else
101     let "t1 = $column"
102     let "t2 = $column + $row"
103 fi
104
105 filter $t1 $t2 # 将负的数组下标过滤出来.
106             # 如果你不做这一步, 将会怎样?
107 done
108
109 echo; echo
110
111 done
```

```
112
113 # 数组旋转的灵感来源于Herbert Mayer所著的
114 #+ "Advanced C Programming on the IBM PC"的例子(第143-146页)
115 #+ (参见参考书目).
116 # 由此可见, C语言能够做到的好多事情,
117 #+ 用shell脚本一样能够做到.
118
119 }
120
121
122 #----- 现在, 让我们开始吧. -----
123 load_alpha    # 加载数组.
124 print_alpha   # 打印数组.
125 rotate        # 逆时钟旋转45度打印.
126 #-----
127
128 exit 0
129
130 # 这是有点做作, 不是那么优雅.
131
132 # 练习:
133 # -----
134 # 1) 重新实现数组加载和打印函数,
135 #     让其更直观, 可读性更强.
136 #
137 # 2) 详细地描述旋转函数的原理.
138 #     提示: 思考一下倒序索引数组的实现.
139 #
140 # 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.
141 #     比如6 X 4的数组.
142 #     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](#).

=====

还有两个使用脚本的更有趣的例子, 请参考:

[例子14-3](#)和[例子A-23](#)

第27章 /dev和/proc

本章目录

1. [/dev](#)
 2. [/proc](#)
-

Linux或者UNIX机器典型地都带有/dev和/proc目录, 用于特殊目的.

1 /dev

/dev目录包含物理设备的条目, 这些设备可能会以硬件的形式出现, 也可能不会. [1] 包含有挂载文件系统的硬驱动器分区, 在/dev目录中都有对应的条目, 就像df命令所展示的那样.

```
1 bash$ df
2 Filesystem      1k-blocks      Used Available Use%
3 Mounted on
4 /dev/hda6        495876     222748    247527  48% /
5 /dev/hda1        50755       3887     44248   9% /boot
6 /dev/hda8        367013     13262    334803   4% /home
7 /dev/hda5        1714416    1123624   503704  70% /usr
```

在其他方面, /dev目录也包含环回设备, 比如/dev/loop0. 一个环回设备就是一种机制, 可以让一般文件访问起来就像块设备那样. [2] 这使得我们可以挂载一个完整的文件系统, 这个文件系统是在一个大文件中所创建的. 参考[例子13-8](#)和[例子13-7](#).

/dev中还有少量的伪设备用于其它特殊目的, 比如[/dev/null](#), [/dev/zero](#), [/dev/urandom](#), [/dev/sda1](#), [/dev/udp](#), 和[/dev/tcp](#).

举个例子:

为了[挂载\(mount\)](#)一个USB闪存驱动器, 将下边一行附加到/etc/fstab中. [3]

```
1 /dev/sda1      /mnt/flashdrive  auto      noauto,user,noatime  0 0
```

(也请参考[例子A-24](#).)

当在/dev/tcp/host/port伪设备文件上执行一个命令的时候, Bash会打开一个TCP连接, 也就是打开相关的socket. [4]

从nist.gov上获取时间:

```
1 bash$ cat </dev/tcp/time.nist.gov/13
2 53082 04-03-18 04:26:54 68 0 0 502.3 UTC(NIST) *
```

[Mark贡献了上面的例子.]

下载一个URL:

```
1 bash$ exec 5<>/dev/tcp/www.net.cn/80
2 bash$ echo -e "GET / HTTP/1.0\n" >&5
3 bash$ cat <&5
```

[感谢, Mark和Mihai Maties.]

例子27-1. 利用/dev/tcp来检修故障

```
1#!/bin/bash
2# dev-tcp.sh: 利用/dev/tcp重定向来检查Internet连接.
3
4# 本脚本由Troy Engel编写.
5# 经过授权在本书中使用.
6
7TCP_HOST=www.dns-diy.com # 一个已知的对垃圾邮件友好的ISP.
8TCP_PORT=80               # 端口80是http.
9
```

```

10 # 尝试连接. (有些像'ping' . . .)
11 echo "HEAD / HTTP/1.0" >/dev/tcp/${TCP_HOST}/${TCP_PORT}
12 MYEXIT=$?
13
14 : <<EXPLANATION
15 If bash was compiled with --enable-net-redirections, it has the capability of
16 using a special character device for both TCP and UDP redirections. These
17 redirections are used identically as STDIN/STDOUT/STDERR. The device entries
18 are 30,36 for /dev/tcp:
19
20 mknod /dev/tcp c 30 36
21
22 >From the bash reference:
23 /dev/tcp/host/port
24     If host is a valid hostname or Internet address, and port is an integer
25 port number or service name, Bash attempts to open a TCP connection to the
26 corresponding socket.
27 EXPLANATION
28
29
30 if [ "$MYEXIT" = "0" ]; then
31     echo "Connection successful. Exit code: $MYEXIT"
32 else
33     echo "Connection unsuccessful. Exit code: $MYEXIT"
34 fi
35
36 exit $MYEXIT

```

译者注: 由于上边例子的输出大部分都是英文, 所以译者补充一下脚本输出的译文.

如果bash以--enable-net-redirections选项来编译, 那么它就拥有了使用一个特殊字符设备的能力, 这个特殊字符设备用于TCP和UDP重定向.

这种重定向的能力就像STDIN/STDOUT/STDERR一样被使用. 该设备/dev/tcp的主次设备号是30, 36:

```
1 mknod /dev/tcp c 30 36
```

>摘自bash参考手册:

/dev/tcp/host/port

如果host是一个有效的主机名或Internet地址, 并且port是一个整数端口号或者是服务器名称, Bash将会打开一个TCP连接, 到相应的socket上.

注意事项

1. /dev目录中的条目为各种物理设备和虚拟设备提供挂载点. 这些条目占用非常少的硬盘空间. 某些设备, 比如/dev/null, /dev/zero, 和/dev/urandom都是虚拟的. 它们都不是真实的物理设备, 它们仅仅存在于软件中.

2. 块设备都是以块为单位进行读写的, 与之相对应的字符设备都是以字符为单位进行访问的.
典型的块设备比如硬盘和CD ROM驱动器. 典型的字符设备例如键盘.

3. 当然, 挂载点/mnt/flashdrive必须存在. 如果不存在, 请使用root用户来执行mkdir /mnt/flashdrive.

为了能够真正的挂载驱动器, 请使用下面的命令: mount /mnt/flashdrive

对于现在比较新的Linux发行版来说, 都会自动把闪存驱动器设备挂载到/media目录上.

4. socket是一个通讯节点, 这个通讯节点与一个特殊的I/O端口相关联. 它允许数据传输, 可以在同一台机器的不同硬件设备间传输, 可以在同一个网络中的不同主机之间传输, 可以在不同网络的不同主机间传输, 当然, 也可以在Internet上的不同地区之间的不同主机之间传输.

2 /proc

/proc目录实际上是一个伪文件系统。/proc目录中的文件用来映射当前运行的系统，内核进程以及与它们相关地状态与统计信息。

```
1 bash$ cat /proc/devices
2 Character devices:
3   1 mem
4   2 pty
5   3 ttyp
6   4 ttys
7   5 cuu
8   7 vcs
9   10 misc
10  14 sound
11  29 fb
12  36 netlink
13  128 ptm
14  136 pts
15  162 raw
16  254 pcmcia
17
18 Block devices:
19   1 ramdisk
20   2 fd
21   3 ide0
22   9 md
23
24
25
26 bash$ cat /proc/interrupts
27          CPU0
28  0:      84505      XT-PIC  timer
29  1:      3375       XT-PIC  keyboard
30  2:          0       XT-PIC  cascade
31  5:          1       XT-PIC  soundblaster
32  8:          1       XT-PIC  rtc
33 12:      4231       XT-PIC  PS/2 Mouse
34 14:     109373      XT-PIC  ide0
35 NMI:          0
36 ERR:          0
37
38
39
40 bash$ cat /proc/partitions
```

```

41 major minor #blocks name rio rmerge rsect ruse wio wmerge wsect wuse running use aveq
42 3 0 3007872 hda 4472 22260 114520 94240 3551 18703 50384 549710 0 111550 644030
43 3 1 52416 hda1 27 395 844 960 4 2 14 180 0 800 1140
44 3 2 1 hda2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
45 3 4 165280 hda4 10 0 20 210 0 0 0 0 0 210 210
46 ...
47
48
49
50 bash$ cat /proc/loadavg
51 0.13 0.42 0.27 2/44 1119
52
53
54
55 bash$ cat /proc/apm
56 1.16 1.2 0x03 0x01 0xff 0x80 -1% -1 ?

```

Shell脚本可以从/proc目录下的某些特定文件中提取数据. [1]

```

1 FS=iso                      # 内核是否支持ISO文件系统?
2
3 grep $FS /proc/filesystems   # iso9660

```

```

1 kernel_version=$( awk '{ print $3 }' /proc/version )

```

```

1 CPU=$( awk '/model name/ {print $4}' < /proc/cpuinfo )
2
3 if [ $CPU = Pentium ]
4 then
5     run_some_commands
6     ...
7 else
8     run_different_commands
9     ...
10 fi

```

```

1 devfile="/proc/bus/usb/devices"
2 USB1="Spd=12"
3 USB2="Spd=480"
4
5
6 bus_speed=$(grep Spd $devfile | awk '{print $9}')
7
8 if [ "$bus_speed" = "$USB1" ]
9 then
10    echo "USB 1.1 port found."
11    # 可以在这里添加操作USB 1.1的相关动作.

```

```
12 fi
```

/proc目录下包含有许多以不同数字命名的子目录。这些作为子目录名字的数字，代表的是当前正在运行进程的[进程ID](#)。在这些以数字命名的子目录中，每一个子目录都有许多文件用来保存对应进程的可用信息。文件stat和status保存着进程运行时的各项统计信息，文件cmdline保存着进程被调用时的命令行参数，而文件exe是一个符号链接，这个符号链接指向这个运行进程的完整路径。还有许多类似这样的文件，如果从脚本的视角来看它们的话，这些文件都非常的有意思。

例子27-2. 找出与给定PID相关联的进程

```
1 #!/bin/bash
2 # pid-identifier.sh: 给出与指定pid相关联进程的完整路径名。
3
4 ARGNO=1 # 期望的参数个数。
5 E_WRONGARGS=65
6 E_BADPID=66
7 E_NOSUCHPROCESS=67
8 E_NOPERMISSION=68
9 PROCFILE=exe
10
11 if [ $# -ne $ARGNO ]
12 then
13     echo "Usage: `basename $0` PID-number" >&2 # Error message
14     >stderr(错误信息重定向到标准错误)。
15     exit $E_WRONGARGS
16 fi
17
18 pidno=$( ps ax | grep $1 | awk '{ print $1 }' | grep $1 )
19 # 从"ps"命令的输出中搜索带有pid的行，pid位置在第一列#1，由awk过滤出来。
20 # 然后再次确认这就是我们所要找的进程，而不是由这个脚本调用所产生的进程。
21 # 最后的"grep $1"就是用来过滤掉这种可能性。
22 #
23 # pidno=$( ps ax | awk '{ print $1 }' | grep $1 )
24 # 这么写就可以了，这一点由Teemu Huovila指出。
25
26 if [ -z "$pidno" ] # 如果经过所有的过滤之后，得到的结果是一个长度
27 then                 # 为0的字符串，那就说明这个pid没有相应的进程在运行。
28     echo "No such process running."
29     exit $E_NOSUCHPROCESS
30 fi
31
32 # 也可以这么写：
33 # if ! ps $1 > /dev/null 2>&1
34 # then                  # 没有与给定pid相匹配的进程在运行。
35 #     echo "No such process running."
36 #     exit $E_NOSUCHPROCESS
37 # fi
```

```

38
39 # 为了简化整个过程，可以使用"pidof".
40
41
42 if [ ! -r "/proc/$1/$PROCFILE" ] # 检查读权限.
43 then
44     echo "Process $1 running, but..."
45     echo "Can't get read permission on /proc/$1/$PROCFILE."
46     exit $E_NOPERMISSION # 一般用户不能访问/proc目录下的某些文件.
47 fi
48
49 # 最后两个测试可以使用下面的语句来代替：
50 #     if ! kill -0 $1 > /dev/null 2>&1 # '0'不是一个信号，but
51 #                                     # 但是这么做，可以测试一下是否
52 #                                     # 可以向该进程发送信号.
53 #     then echo "PID doesn't exist or you're not its owner" >&2
54 #     exit $E_BADPID
55 # fi
56
57
58
59 exe_file=$( ls -l /proc/$1 | grep "exe" | awk '{ print $11 }' )
60 # 或      exe_file=$( ls -l /proc/$1/exe | awk '{print $11}' )
61 #
62 # /proc/pid-number/exe是一个符号链接，
63 # 指向这个调用进程的完整路径名.
64
65 if [ -e "$exe_file" ] # 如果/proc/pid-number/exe存在...
66 then                  # 那么相应的进程就存在.
67     echo "Process #$1 invoked by $exe_file."
68 else
69     echo "No such process running."
70 fi
71
72
73 # 这个精心制作的脚本，*几乎*能够被下边这一行所替代：
74 # ps ax | grep $1 | awk '{ print $5 }'
75 # 但是，这样并不会工作...
76 # 因为'ps'输出的第5列是进程的argv[0](译者注：这是命令行第一个参数，
77 # 即调用时程序用的程序路径本身.)
78 # 而不是可执行文件的路径.
79 #
80 # 然而，下边这两种方法都能正确地完成这个任务.
81 #     find /proc/$1/exe -printf '%l\n'
82 #     lsof -aFn -p $1 -d txt | sed -ne 's/^n//p'
```

```
83  
84 # 附加注释，是Stephane Chazelas添加的.  
85  
86 exit 0
```

例子27-3. 网络连接状态

```
1#!/bin/bash  
2  
3 PROCNAME=pppd          # ppp守护进程  
4 PROCFILENAME=status    # 在这里寻找信息.  
5 NOTCONNECTED=65  
6 INTERVAL=2              # 每2秒刷新一次.  
7  
8 pidno=$( ps ax | grep -v "ps ax" | grep -v grep | grep $PROCNAME | \  
9 awk '{ print $1 }' )  
10 # 找出'pppd'所对应的进程号，即'ppp守护进程'.  
11 # 必须过滤掉由搜索本身所产生的进程行.  
12 #  
13 # 然而，就像Oleg Philon所指出的那样，  
14 #+ 如果使用"pidof"的话，就会非常简单.  
15 # pidno=$( pidof $PROCNAME )  
16 #  
17 # 从经验中总结出来的忠告：  
18 #+ 当命令序列变得非常复杂的时候，一定要找到更简洁的方法.  
19  
20  
21 if [ -z "$pidno" ]      # 如果没有找到匹配的pid，那么就说明相应的进程没运行.  
22 then  
23   echo "Not connected."  
24   exit $NOTCONNECTED  
25 else  
26   echo "Connected."; echo  
27 fi  
28  
29 while [ true ]          # 死循环，这里可以改进一下.  
30 do  
31  
32   if [ ! -e "/proc/$pidno/$PROCFILENAME" ]  
33   # 进程运行时，相应的"status"文件就会存在.  
34   then  
35     echo "Disconnected."  
36     exit $NOTCONNECTED  
37   fi  
38  
39 netstat -s | grep "packets received"  # 获得一些连接统计.
```

```
40 netstat -s | grep "packets delivered"
41
42
43 sleep $INTERVAL
44 echo; echo
45
46 done
47
48 exit 0
49
50 # 如果你想停止这个脚本, 只能使用Control-C.
51
52 # 练习:
53 # -----
54 # 改进这个脚本, 使它可以按"q"键退出.
55 # 提高这个脚本的用户友好性.
```

►: 一般来说, 在/proc目录中, 进行写文件操作是非常危险的, 因为这么做可能会破坏文件系统, 甚至于摧毁整个机器.

注意事项

1. 某些系统命令也可做类似的事情, 比如[procinfo](#), [free](#), [vmstat](#), [lsdev](#), 和[uptime](#).

第28章 Zero与Null

/dev/zero与/dev/null

使用/dev/null

可以把/dev/null想象为一个”黑洞”. 它非常接近于一个只写文件. 所有写入它的内容都会永远丢失. 而如果想从它那读取内容, 则什么也读不到. 但是, 对于命令行和脚本来说, /dev/null却非常的有用.

禁用stdout.

```
1 cat $filename >/dev/null
2 # 文件的内容不会输出到stdout.
```

禁用stderr (来自于[例子12-3](#)).

```
1 rm $badname 2>/dev/null
2 # 错误消息[stderr]就这么被丢到太平洋去了.
```

禁用stdout和stderr.

```
1 cat $filename 2>/dev/null >/dev/null
2 # 如果"$filename"不存在, 将不会有错误消息输出.
3 # 如果"$filename"存在, 文件内容不会输出到stdout.
4 # 因此, 上边的代码根本不会产生任何输出.
5 #
6 # 如果你只想测试一下命令的返回码, 而不想要任何输出时,
7 #+ 这么做就非常有用了.
8 #
9 # cat $filename &>/dev/null
10 # 也可以, 由Baris Cicek指出.
```

删除一个文件的内容, 但是保留文件本身, 并且保留所有的文件访问权限(来自于[例子2-1](#)和[例子2-3](#)):

```
1 cat /dev/null > /var/log/messages
2 # : > /var/log/messages 具有同样的效果,
3 # 但是不会产生新进程.(译者注: 因为是内建的)
4
5 cat /dev/null > /var/log/wtmp
```

自动清空日志文件的内容(特别适用于处理那些由商业站点发送的, 令人厌恶的”cookie”):

例子28-1. 隐藏令人厌恶的cookie

```
1 if [ -f ~/.netscape/cookies ] # 如果存在, 就删除.
2 then
3   rm -f ~/.netscape/cookies
4 fi
5
```

```
6 ln -s /dev/null ~/.netscape/cookies  
7 # 现在所有的cookie都被扔到黑洞里去了，这样就不会保存在我们的磁盘中了。
```

使用/dev/zero

类似于/dev/null, /dev/zero也是一个伪文件, 但事实上它会产生一个null流(二进制的0流, 而不是ASCII类型). 如果你想把其他命令的输出写入它的话, 那么写入的内容会消失, 而且如果你想从/dev/zero中读取一连串null的话, 也非常的困难, 虽然可以使用od或者一个16进制编辑器来达到这个目的. /dev/zero的主要用途就是用来创建一个指定长度, 并且初始化为空的文件, 这种文件一般都用作临时交换文件.

例子28-2. 使用/dev/zero来建立一个交换文件

```
1#!/bin/bash  
2# 创建一个交换文件.  
3  
4ROOT_UID=0          # Root用户的$UID为0.  
5E_WRONG_USER=65    # 不是root?  
6  
7FILE=/swap  
8BLOCKSIZE=1024  
9MINBLOCKS=40  
10SUCCESS=0  
11  
12  
13# 这个脚本必须以root身份来运行.  
14if [ "$UID" -ne "$ROOT_UID" ]  
15then  
16    echo; echo "You must be root to run this script."; echo  
17    exit $E_WRONG_USER  
18fi  
19  
20  
21blocks=${1:-$MINBLOCKS}          # 如果没在命令行上指定,  
22                                #+ 默认设置为40块.  
23# 上边这句等价于下面这个命令块.  
24# -----  
25# if [ -n "$1" ]  
26# then  
27#     blocks=$1  
28# else  
29#     blocks=$MINBLOCKS  
30# fi  
31# -----  
32  
33  
34if [ "$blocks" -lt $MINBLOCKS ]
```

```

35 then
36     blocks=$MINBLOCKS          # 至少要有40块.
37 fi
38
39
40 echo "Creating swap file of size $blocks blocks (KB)."
41 dd if=/dev/zero of=$FILE bs=$BLOCKSIZE count=$blocks # 用零填充文件.
42
43 mkswap $FILE $blocks        # 将其指定为交换文件(译者注: 或称为交换分区).
44 swapon $FILE                # 激活交换文件.
45
46 echo "Swap file created and activated."
47
48 exit $SUCCESS

```

/dev/zero还有其他的应用场合, 比如当你出于特殊目的, 需要”用0填充”一个指定大小的文件时, 就可以使用它. 举个例子, 比如要将一个文件系统挂载到环回设备(loopback device)上(请参考[例子13-8](#)), 或者想”安全”的删除一个文件(请参考[例子12-55](#)).

例子28-3. 创建一个ramdisk

```

1#!/bin/bash
2# ramdisk.sh
3
4# 一个"ramdisk"就是系统RAM内存中的一部分,
5#+ 只不过它被当作文件系统来操作.
6# 它的优点是访问速度非常快(读/写时间快).
7# 缺点: 易失性, 当机器重启或关机时, 会丢失数据.
8#+           而且会减少系统可用的RAM.
9#
10# 那么ramdisk有什么用呢?
11# 保存一个大数据集, 比如保存表格或字典.
12#+ 这样的话, 可以增加查询速度, 因为访问内存比访问硬盘快得多.
13
14
15E_NON_ROOT_USER=70      # 必须以root身份来运行.
16ROOTUSER_NAME=root
17
18MOUNTPT=/mnt/ramdisk
19SIZE=2000                # 2K个块(可以进行适当的修改)
20BLOCKSIZE=1024            # 每块的大小为1K(1024字节)
21DEVICE=/dev/ram0          # 第一个ram设备
22
23username='id -nu'
24if [ "$username" != "$ROOTUSER_NAME" ]
25then
26    echo "Must be root to run \\"`basename $0`\\"."

```

```

27 exit $E_NON_ROOT_USER
28 fi
29
30 if [ ! -d "$MOUNTPT" ]    # 测试挂载点是否已经存在,
31 then                      #+ 如果做了这个判断的话, 当脚本运行多次的时候,
32   mkdir $MOUNTPT           #+ 就不会报错了. (译者注: 主要是为了避免多次创建目录.)
33 fi
34
35 dd if=/dev/zero of=$DEVICE count=$SIZE bs=$BLOCKSIZE # 把RAM设备的内容用0填充.
36                                         # 为什么必须这么做?
37 mke2fs $DEVICE          # 在RAM上创建一个ext2文件系统.
38 mount $DEVICE $MOUNTPT  # 挂载上.
39 chmod 777 $MOUNTPT      # 使一般用户也可以访问这个ramdisk.
40                         # 然而, 只能使用root身份来卸载它.
41
42 echo "\"$MOUNTPT\" now available for use."
43 # 现在ramdisk就可以访问了, 即使是普通用户也可以访问.
44
45 # 小心, ramdisk存在易失性,
46 #+ 如果重启或关机的话, 保存的内容就会消失.
47 # 所以, 还是要将你想保存的文件, 保存到常规磁盘目录下.
48
49 # 重启之后, 运行这个脚本, 将会再次建立一个ramdisk.
50 # 如果你仅仅重新加载/mnt/ramdisk, 而没有运行其他步骤的话, 那就不会正常工作.
51
52 # 如果对这个脚本进行适当的改进, 就可以将其放入/etc/rc.d/rc.local中,
53 #+ 这样, 在系统启动的时候就会自动建立一个ramdisk.
54 # 这么做非常适合于那些对速度要求很高的数据库服务器.
55
56 exit 0

```

最后值得一提的是, ELF二进制文件需要使用/dev/zero.

第29章 调试

首先, 调试要比编写代码困难得多,
因此, 如果你尽可能聪明的编写代码,
你就不会在调试的时候花费很多精力.

— Brian Kernighan

Bash并不包含调试器, 甚至都没有包含任何用于调试目的的命令和结构. [1] 脚本中的语法错误, 或者拼写错误只会产生模糊的错误信息, 当你调试一些非功能性脚本的时候, 这些错误信息通常都不会提供有意义的帮助.

```
1
2 1#!/bin/bash
3 # ex74.sh
4
5 # 这是一个错误脚本.
6 # 哪里出了错?
7
8 7 a=37
9
10 9 if [$a -gt 27 ]
11 then
12 echo $a
13 fi
14
15 14 exit 0
```

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29-2. 缺少关键字

```
1
2 1#!/bin/bash
3 # missing-keyword.sh: 这个脚本会产生什么错误?
4
5 4 for a in 1 2 3
6 do
7   echo "$a"
8 # done      # 第7行上的关键字done'被注释掉了.
9
10 9 exit 0
```

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29-3. test24, 另一个错误脚本

```
1
2 1#!/bin/bash
3
4 # 这个脚本的目的是删除当前目录下的某些文件,
5 #+ 这些文件特指那些文件名包含空格的文件.
6 # 但是不能如我们所愿的那样工作.
7 # 为什么?
8
9
10 badname='ls | grep '
11
12 # 试试这个:
13 # echo "$badname"
14
15 rm "$badname"
16
17 exit 0
```

为了找出[例子29-3](#)中的错误, 我们可以把echo "\$badname"行的注释符去掉. echo出来的信息能够帮助你判断脚本是否按你期望的方式运行.

在这个特定的例子里, rm "\$badname"之所以没有给出期望的结果, 是因为\$badname不应该被引用起来. 加上引号会保证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 # 删除文件名中包含空格的文件, 下面这才是正确的方法.
2 rm *\ *
3 rm *" "*
4 rm *' '*
5 # 感谢. S.C.
```

总结一下这个脚本的症状,

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).
3. 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).

如果想调试不工作的脚本, 有如下工具可用:

-
1. `echo`语句可以放在脚本中存在疑问的位置上, 来观察变量的值, 也可以了解脚本后续的动作.

最好只在调试的时候才使用`echo`语句.

```
1 1 ##### debecho (debug-echo), 由Stefano Falsetto编写 #####
2 2 ##### 只有在DEBUG变量被赋值的情况下, 才会打印传递进来的参数. #####
3 3 debecho () {
4 4     if [ ! -z "$DEBUG" ]; then
5 5         echo "$1" >&2
6 6         #           ^^^ 打印到stderr
7 7     fi
8 8 }
9 9
10 10 DEBUG=on
11 11 Whatever=whatnot
12 12 debecho $Whatever    # whatnot
13 13
14 14 DEBUG=
15 15 Whatever=notwhat
16 16 debecho $Whatever    # (这里就不会打印.)
```

2. 使用过滤器`tee`来检查临界点上的进程或数据流.

3. 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. 这等价于把`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. 这等价于把`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.

4. 使用”assert”(断言)函数在脚本的临界点上测试变量或条件. (这是从C语言中引入的.)

例子29-4. 使用”assert”来测试条件

```
1 1#!/bin/bash
2 2 # assert.sh
3 3
4 4 assert ()          # 如果条件为false,
5 5 {                  #+ 那么就打印错误信息并退出脚本.
6 6     E_PARAM_ERR=98
7 7     E_ASSERT_FAILED=99
8 8
9 9
10 10    if [ -z "$2" ]      # 传递进来的参数个数不够.
```

```

11 11 then
12 12     return $E_PARAM_ERR # 什么都不做就return.
13 13 fi
14 14
15 15 lineno=$2
16 16
17 17 if [ ! $1 ]
18 18 then
19 19     echo "Assertion failed: \"$1\""
20 20     echo "File \"$0\", line $lineno"
21 21     exit $E_ASSERT_FAILED
22 22 # else
23 23 # 返回
24 24 # 然后继续执行脚本余下的代码.
25 25 fi
26 26 }
27 27
28 28
29 29 a=5
30 30 b=4
31 31 condition="$a -lt $b"      # 产生错误信息并退出脚本.
32 32                         # 尝试把这个"条件"放到其他的地方,
33 33                         #+ 然后看看发生了什么.
34 34
35 35 assert "$condition" $LINENO
36 36 # 只有在"assert"成功时, 脚本余下的代码才会继续执行.
37 37
38 38
39 39 # 这里放置的是其他的一些命令.
40 40 #
41 41 echo "This statement echoes only if the \"assert\" does not fail."
42 42 #
43 43 # 这里也放置其他一些命令.
44 44
45 45 exit 0

```

5. 使用变量\$LINENO和内建命令caller.
6. 捕获exit. 脚本中的exit命令会触发一个信号0, 这个信号终止进程, 也就是终止脚本本身.
[2] 捕获exit在某些情况下很有用, 比如说强制"打印"变量值. trap命令必须放在脚本中第一个命令的位置上.

捕获信号

trap

可以在收到一个信号的时候指定一个处理动作; 在调试的时候, 这一点也非常有用.

►: A signal就是发往进程的一个简单消息, 这个消息即可以由内核发出, 也可以由另一个进程发出, 发送这个消息的目的是为了通知目的进程采取一些指定动作(通常都是终止动作). 比如说, 按下**Control-C**, 就会发送一个用户中断(即INT信号)到运行中的进程.

```
1 trap '' 2
2 # 忽略中断2(Control-C), 没有指定处理动作.
3
4 trap 'echo "Control-C disabled."' 2
5 # 当Control-C按下时, 显示一行信息.
```

例子29-5. 捕获exit

```
1#!/bin/bash
2# 使用trap来捕捉变量值.
3
4 trap 'echo Variable Listing --- a = $a b = $b' EXIT
5 # EXIT是脚本中exit命令所产生信号的名字.
6 #
7 # "trap"所指定的命令并不会马上执行,
8 #+ 只有接收到合适的信号, 这些命令才会执行.
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
16 b=36
17
18 exit 0
19 # 注意, 即使注释掉上面的这行'exit'命令, 也不会产生什么不同的结果,
20 #+ 这是因为所有命令都执行完毕后, 不管怎么样, 脚本都会退出的.
```

例子29-6. Control-C之后, 清除垃圾

```
1#!/bin/bash
2# logon.sh: 一个检查你是否在线的脚本, 这个脚本实现的很简陋.
3
4 umask 177 # 确保temp文件并不是所有用户都有权限访问.
5
6
7 TRUE=1
8 LOGFILE=/var/log/messages
9 # 注意: $LOGFILE必须是可读的
10 #+ (使用root身份来执行, chmod 644 /var/log/messages).
11 TEMPFILE=temp.$$
12 # 使用脚本的进程ID, 来创建一个"唯一"的临时文件名.
13 # 也可以使用'mktemp'.
```

```
14 # 比如:
15 # TEMPFILE='mktemp temp.XXXXXX'
16 KEYWORD=address
17 # 登陆时, 把"remote IP address xxx.xxx.xxx.xxx"
18 # 添加到/var/log/messages中.
19 ONLINE=22
20 USER_INTERRUPT=13
21 CHECK_LINES=100
22 # 日志文件有多少行需要检查.
23
24 trap 'rm -f $TEMPFILE; exit $USER_INTERRUPT' TERM INT
25 # 如果脚本被control-c中断的话, 那么就清除掉临时文件.
26
27 echo
28
29 while [ $TRUE ] #死循环.
30 do
31 tail -$CHECK_LINES $LOGFILE> $TEMPFILE
32 # 将系统日志文件的最后100行保存到临时文件中.
33 # 这么做很有必要, 因为新版本的内核在登陆的时候, 会产生许多登陆信息.
34 search='grep $KEYWORD $TEMPFILE'
35 # 检查是否存在"IP address"片断,
36 #+ 它用来提示, 这是一次成功的网络登陆.
37
38 if [ ! -z "$search" ] # 必须使用引号, 因为变量可能会包含一些空白符.
39 then
40     echo "On-line"
41     rm -f $TEMPFILE # 清除临时文件.
42     exit $ONLINE
43 else
44     echo -n "."
45         # echo的-n选项不会产生换行符.
46         #+ 这样你就可以在一行上连续打印.
47 fi
48 sleep 1
49 done
50
51
52 # 注意: 如果你将变量KEYWORD的值改为"Exit",
53 #+ 当在线时, 这个脚本就可以被用来检查
54 #+ 意外的掉线情况.
55
56 # 练习: 按照上面"注意"中所说的那样来修改这个脚本,
57 # 让它表现的更好.
58
```

```

59 59 exit 0
60 60
61 61
62 # Nick Drage建议使用另一种方法：
63
64 while true
65 do ifconfig ppp0 | grep UP 1> /dev/null && echo "connected" && exit 0
66 echo -n "."    # 不停的打印(.....), 用来提示用户等待, 直到连接上位置.
67 sleep 2
68 done
69
70 # 问题: 使用Control-C来终止这个进程可能是不够的.
71 #+      (可能还会继续打印"...")
72 # 练习: 修复这个问题.
73
74
75
76 # Stephane Chazelas提出了另一种方法:
77
78 CHECK_INTERVAL=1
79
80 while ! tail -1 "$LOGFILE" | grep -q "$KEYWORD"
81 do echo -n .
82     sleep $CHECK_INTERVAL
83 done
84 echo "On-line"
85
86 # 练习: 讨论一下这几种不同方法
87 # 各自的优缺点.

```

如果使用trap命令的DEBUG参数, 那么当脚本中每个命令执行完毕后, 都会执行指定的动作. 比方说, 你可以跟踪某个变量的值.

例子29-7. 跟踪一个变量

```

1#!/bin/bash
2
3 trap 'echo "VARIABLE-TRACE> \$variable = \"\$variable\\"" DEBUG
4 # 当每个命令执行之后, 就会打印出$variable的值.
5
6 variable=29
7
8 echo "Just initialized \"\$variable\" to $variable."
9
10 let "variable *= 3"
11 echo "Just multiplied \"\$variable\" by 3."
12

```

```

13 13 exit $?
14
15 15 # "trap 'command1 . . . command2 . . .' DEBUG" 结构更适合于
16 16 #+ 使用在复杂脚本的上下文中,
17 17 #+ 如果在这种情况下大量使用"echo $variable"语句的话,
18 18 #+ 将会非常笨拙, 而且很耗时.
19
20 20 # 感谢, Stephane Chazelas指出这点.
21
22
23 23 脚本的输出:
24
25 25 VARIABLE-TRACE> $variable = ""
26 26 VARIABLE-TRACE> $variable = "29"
27 27 Just initialized "$variable" to 29.
28 28 VARIABLE-TRACE> $variable = "29"
29 29 VARIABLE-TRACE> $variable = "87"
30 30 Just multiplied "$variable" by 3.
31 31 VARIABLE-TRACE> $variable = "87"

```

当然, 除了调试之外, trap命令还有其他的用途.

例子29-8. 运行多进程(在对称多处理器(SMP box)的机器上)

```

1 1#!/bin/bash
2 2 # parent.sh
3 3 # 在多处理器(SMP box)的机器里运行多进程.
4 4 # 作者: Tedman Eng
5
6 6 # 我们下面要介绍两个脚本, 这是第一个,
7 7 #+ 这两个脚本都要放到当前工作目录下.
8
9
10 10
11 11
12 12 LIMIT=$1          # 想要启动的进程总数
13 13 NUMPROC=4         # 并发的线程数量(forks?)
14 14 PROCID=1           # 启动的进程ID
15 15 echo "My PID is $$"
16
17 17 function start_thread() {
18     if [ $PROCID -le $LIMIT ] ; then
19         ./child.sh $PROCID&
20         let "PROCID++"
21     else
22         echo "Limit reached."
23         wait

```

```
24         exit
25     fi
26 }
27
28 while [ "$NUMPROC" -gt 0 ]; do
29     start_thread;
30     let "NUMPROC--"
31 done
32
33
34 while true
35 do
36
37 trap "start_thread" SIGRTMIN
38
39 done
40
41 exit 0
42
43
44
45 # ===== 下面是第二个脚本 =====
46
47
48 #!/bin/bash
49 # child.sh
50 # 在SMP box上运行多进程.
51 # 这个脚本会被parent.sh调用.
52 # 作者: Tedman Eng
53
54 temp=$RANDOM
55 index=$1
56 shift
57 let "temp %= 5"
58 let "temp += 4"
59 echo "Starting $index Time:$temp" "$@"
60 sleep ${temp}
61 echo "Ending $index"
62 kill -s SIGRTMIN $PPID
63
64 exit 0
65
66
67 # ===== 脚本作者注 ===== #
68 # 这个脚本并不是一点bug都没有.
```

```
69 # 我使用limit = 500来运行这个脚本，但是在过了开头的一两百个循环后，  
70 #+ 有些并发线程消失了！  
71 # 还不能确定这是否是由捕捉信号的冲突引起的，或者是其他什么原因。  
72 # 一旦接收到捕捉的信号，那么在下一次捕捉到来之前，  
73 #+ 会有一段短暂的时间来执行信号处理程序，  
74 #+ 在信号处理程序处理的过程中，很有可能会丢失一个想要捕捉的信号，  
75 #+ 因此失去一个产生子进程的机会。  
76  
77 # 毫无疑问的，肯定有人能够找出产生这个bug的原因，  
78 #+ 并且在将来的某个时候... 修正它。  
79  
80  
81 # ====== #  
82  
83  
84  
85 # ----- #  
86  
87  
88  
89 #####  
90 # 下面的脚本是由Vernia Damiano原创。  
91 # 不幸的是，它并不能正常工作。  
92 #####  
93  
94 #!/bin/bash  
95  
96 # 要想调用这个脚本，至少需要一个整形参数  
97 #+ (并发的进程数)。  
98 # 所有的其他参数都传递给要启动的进程。  
99  
100  
101  
102 INDICE=8      # 想要启动的进程数目  
103 TEMPO=5       # 每个进程最大的睡眠时间  
104 E_BADARGS=65  # 如果没有参数传到脚本中，那么就返回这个错误码。  
105  
106 if [ $# -eq 0 ] # 检查一下，至少要传递一个参数给脚本。  
107 then  
108   echo "Usage: `basename $0` number_of_processes [passed params]"  
109   exit $E_BADARGS  
110 fi  
111  
112 NUMPROC=$1      # 并发进程数  
113 shift
```

```

114 113 PARAMETRI=( "$@" )      # 每个进程的参数
115
116 114
116 115 function avvia() {
117 116     local temp
118 117     local index
119 118     temp=$RANDOM
120 119     index=$1
121 120     shift
122 121     let "temp %= $TEMPO"
122 122     let "temp += 1"
123 123     echo "Starting $index Time:$temp" "$@"
124 124     sleep ${temp}
125 125     echo "Ending $index"
126 126     kill -s SIGRTMIN @@
127 127 }
128
128
130 129 function parti() {
131 130     if [ $INDICE -gt 0 ] ; then
132 131         avvia $INDICE "${PARAMETRI[@]}" &
133 132         let "INDICE--"
134 133     else
135 134         trap : SIGRTMIN
136 135     fi
136 }
137
137
139 138 trap parti SIGRTMIN
140
140
141 140 while [ "$NUMPROC" -gt 0 ]; do
141 141     parti;
142 142     let "NUMPROC--"
143
143 done
144
144
145 145 wait
146 146 trap - SIGRTMIN
147
147
148 148 exit $?
149
149
150 150 : <<SCRIPT_AUTHOR_COMMENTS
151 我需要使用指定的选项来运行一个程序,
152 并且能够接受不同的文件, 而且要运行在一个多处理器(SMP)的机器上.
153 所以我想(我也会)运行指定数目个进程,
154 并且每个进程终止之后, 都要启动一个新进程.
155
155
156 156 "wait"命令并没有提供什么帮助, 因为它需要等待一个指定的后台进程,
157 157 或者等待*全部*的后台进程. 所以我编写了[这个]bash脚本程序来完成这个工作,

```

```
159 158 并且使用了"trap"指令.  
160 159 --Vernia Damiano  
161 160 SCRIPT_AUTHOR_COMMENTS
```

trap ” SIGNAL(两个引号之间为空)在剩余的脚本中禁用了SIGNAL信号的动作. trap SIGNAL则会恢复处理SIGNAL的动作. 当你想保护脚本的临界部分不受意外的中断骚扰, 那么上面讲的这种办法就非常有用了.

```
1 1      trap '' 2 # 信号2就是Control-C, 现在被禁用了.  
2 2      command  
3 3      command  
4 4      command  
5 5      trap 2      # 重新恢复Control-C  
6 6
```

```
1  
2 Bash3.0(chapter.34 section.2)之后增加了如下这些特殊变量用于调试.  
3  
4 $BASH_ARGC  
5  
6 $BASH_ARGV  
7  
8 $BASH_COMMAND  
9  
10 $BASH_EXECUTION_STRING  
11  
12 $BASH_LINENO  
13  
14 $BASH_SOURCE  
15  
16 $BASH_SUBSHELL (chapter.9 section.1)
```

注意事项

1. 事实上, Rocky Bernstein的Bash debugger 填补了这项空白.
2. 根据惯例, 信号0被指定为exit.

第30章 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 1#!/bin/bash
2 2
3 3    set -o verbose
4 4    # 打印出所有执行前的命令.
5 5
```

```
1 1#!/bin/bash
2 2
3 3    set -v
4 4    # 与上边的例子具有相同的效果.
5 5
```

►: 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 1#!/bin/bash
2 2
3 3    set -o verbose
4 4    # 激活命令回显.
5 5    command
6 6    ...
7 7    command
8 8
9 9    set +o verbose
10 10 # 禁用命令回显.
11 11    command
12 12 # 没有命令回显了.
13 13
14 14
15 15    set -v
16 16 # 激活命令回显.
17 17    command
18 18    ...
19 19    command
20 20
21 21    set +v
22 22 # 禁用命令回显.
23 23    command
24 24
```

```
25      exit 0  
26
```

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.  
4
```

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形式来指定(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30-1. Bash选项

缩写	名称	作用
-C	(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
-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	allexport	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
-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)
-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。
-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21)。
-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。

第31章 陷阱

Turandot: Gli enigmi sono tre, la morte una!
Caleph: No, no! Gli enigmi sono tre, una la vita!
— Puccini

(boolhead注：上面句子的来历可以参见WIKI: [In questa reggia](#)
貌似是这样的:P：

Turandot: The riddles are three, Death is one!
Caleph: No, no! the riddles are three, Life is one!
Turandot: 不解之谜有三，死亡是其一！
Caleph: 不，不！不解之谜有三，生命是其一！)

- 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就没问题.  
5  
6 # 然而 . . .  如果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  
10 xyz((!*=value2  # 引起严重的错误.  
11 # 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 # Use 'var_1' instead.  
3  
4 function-whatever ()  # 错误  
5 # 使用'function_whatever ()'来代替.  
6  
7  
8 # Bash3.0之后，标点不能出现在函数名中.  
9 function.whatever ()  # 错误  
10 # 使用'functionWhatever ()'来代替.
```

- 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，但是会让人混淆。

```

- 不合时宜的使用空白字符. 与其它编程语言相比, Bash非常讲究空白字符的使用.

```

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
3echo "uninitialized_var = $uninitialized_var"
4# uninitialized_var =

```

- 混淆测试符号=和-eq. 请记住, =用于比较字符串, 而-eq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-eq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 [ "$a" = 273 ]
11 then
12   echo "Comparison works."
13 else
14   echo "Comparison does not work."
15 fi    # Comparison does not work.
16
17 # 与 a=" 273" 和 a="0273" 相同 .
18
19
20 # 类似的, 如果对非整数值使用 "-eq" 的话, 就会产生问题 .
21
22 if [ "$a" -eq 273.0 ]
23 then
24   echo "a = $a"
25 fi # 因为产生了错误消息, 所以退出 .
26 # test.sh: [: 273.0: integer expression expected

```

- 误用了[字符串比较](#)操作符.

例子31-1. 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
4echo
5number=1
6
7# 下面的 "while 循环" 有两个错误:
8#+ 一个比较明显, 而另一个比较隐蔽 .
9
10while [ "$number" < 5 ]      # 错! 应该是: while [ "$number" -lt 5 ]
11do
12  echo -n "$number "
13  let "number += 1"
14done
15# 如果你企图运行这个错误的脚本, 那么就会得到一个错误消息:
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, 比较两个整数还是错误的 .
19
20
21echo -----
22

```

```

23
24 while [ "$number" < 5 ]      # 1 2 3 4
25 do
26   echo -n "$number "          # 看起来好像可以工作，但是 . .
27   let "number += 1"           #+ 事实上是比较ASCII码,
28 done                          #+ 而不是整数比较.
29
30 echo; echo -----
31
32 # 这么做会产生问题. 比如:
33
34 lesser=5
35 greater=105
36
37 if [ "$greater" < "$lesser" ]
38 then
39   echo "$greater is less than $lesser"
40 fi                                # 105 is less than 5
41 # 事实上，在字符串比较中(按照ASCII码的顺序)
42 #+ "105"小于"5".
43
44 echo
45
46 exit 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](#), 和[例子9-6](#).
- 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就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 # 试图将command1的错误输出重定向到一个管道中...
2 #     ...不会工作.
3
4 command1 2>& - | command2 # 也没效果.
5
6 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23-9](#).
- 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.

- 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
4
5 -----
6 bash$ echo $WHATEVER
7 bash$
```

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.

例子31-2. 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 # 会修改全局变量么?
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" # 未设置.
28 echo "outer_variable outside subshell = $outer_variable" # 未修改.
29 echo
```

```
30
31 exit 0
32
33 # 如果你打开第19和第20行的注释会怎样?
34 # 会产生不同的结果么? (译者注: 小提示, 第18行的, 导出, 都加上引号了.)
```

- 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11-17](#)一样).

例子31-3. 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
6a=aaa
7b=bbb
8c=ccc
9
10echo "one two three" | read a b c
11# 尝试重新给变量a, b, 和c赋值.
12
13echo
14echo "a = $a" # a = aaa
15echo "b = $b" # b = bbb
16echo "c = $c" # c = ccc
17# 重新赋值失败.
18
19# -----
20
21# 尝试下边这种方法.
22
23var='echo "one two three"'
24set -- $var
25a=$1; b=$2; c=$3
26
27echo "-----"
28echo "a = $a" # a = one
29echo "b = $b" # b = two
30echo "c = $c" # c = three
31# 重新赋值成功.
32
33# -----
34
35# 也请注意, echo到'read'的值只会在子shell中起作用.
```

```

36 # 所以, 变量的值*只*会在子shell中被修改.
37
38 a=aaa          # 重新开始.
39 b=bbb
40 c=ccc
41
42 echo; echo
43 echo "one two three" | ( read a b c;
44 echo "Inside subshell: "; echo "a = $a"; echo "b = $b"; echo "c = $c" )
45 # a = one
46 # b = two
47 # c = three
48 echo "-----"
49 echo "Outside subshell: "
50 echo "a = $a"  # a = aaa
51 echo "b = $b"  # b = bbb
52 echo "c = $c"  # c = ccc
53 echo
54
55 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 $HOME -type f -atime +30 -size 100k) # 这里没使用管道.
32 do
33     echo "$f is over 100KB and has not been accessed in over 30 days"
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.

```

-
- 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. [1]
 - 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不能正确的处理[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(或Apple OSX)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](#).

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

第32章 脚本编程风格

编写脚本时，最好养成系统化和结构化的风格。即使你在“空闲时”，“在信封后边顺便做一下草稿”也是非常有益处的，所以，在你坐下来编写代码之前，最好花几分钟的时间来规划和组织一下你的想法。

这里所描述的是一些风格上的指导原则。但是请注意，这节文档并不是想成为一个官方的Shell脚本编写风格。

1 非官方的Shell脚本编写风格

- 习惯性的注释你的代码. 这可以让别人更容易看懂(或者感激)你的代码(译者注: 犯错时, 别人也会靠注释找到你), 而且也更便于维护.

```
1 PASS="$PASS${MATRIX:$((RANDOM%${#MATRIX}))}:1"
2 # 去年你写下这段代码的时候, 你非常了解这段代码的含义,
3 # 但现在它对你来说完全是个谜.
4 # (摘自 Antek Sawicki 的 "pw.sh" 脚本.)
```

给脚本和函数加上描述性的头信息.

```
1 1#!/bin/bash
2
3 #####
4 # xyz.sh #
5 # written by Bozo Bozeman #
6 # July 05, 2001 #
7 #
8 # Clean up project files. #
9 #####
10
11 E_BADDIR=65 # 没有这个目录.
12 projectdir=/home/bozo/projects # 想要清除的目录.
13
14 # -----
15 # cleanup_pfiles () #
16 # 删除指定目录中的所有文件. #
17 # Parameter: $target_directory #
18 # 返回值: 0表示成功, 失败返回$E_BADDIR. #
19 # -----
20 cleanup_pfiles ()
21 {
22     if [ ! -d "$1" ] # Test if target directory exists.
23     then
24         echo "$1 is not a directory."
25         return $E_BADDIR
26     fi
27
28     rm -f "$1"/*
29     return 0 # Success.
30 }
31
32 cleanup_pfiles $projectdir
33
```

34 | 34 exit 0

在脚本开头添加任何注释之前，一定要确保#!/bin/bash放在脚本第一行的开头。

- 避免使用”魔法数字”，[1] 也就是，避免”写死的”字符常量。可以使用有意义的变量名来代替。这使得脚本更易于理解，并且允许在不破坏应用的情况下进行修改和更新。

```
1 if [ -f /var/log/messages ]
2 then
3 ...
4 fi
5 # 一年以后，你决定修改这个脚本，让它来检查/var/log/syslog.
6 # 到时候你就必须一行一行的手动修改这个脚本，
7 # 并且寄希望于没有遗漏的地方。
8
9 # 更好的办法是：
10 LOGFILE=/var/log/messages # 只需要改动一行就行了。
11 if [ -f "$LOGFILE" ]
12 then
13 ...
14 fi
```

- 给变量和函数起一些有意义的名字。

```
1 fl='ls -al $dirname'                      # 含义模糊。
2 file_listing='ls -al $dirname'              # 更好的名字。
3
4
5 MAXVAL=10      # 使用变量来代替脚本常量，并且在脚本中都是用这个变量。
6 while [ "$index" -le "$MAXVAL" ]
7 ...
8
9
10 E_NOTFOUND=75                            # 错误码使用大写，
11                               #+ 并且命名的时候用"E_"作为前缀。
12 if [ ! -e "$filename" ]
13 then
14   echo "File $filename not found."
15   exit $E_NOTFOUND
16 fi
17
18
19 MAIL_DIRECTORY=/var/spool/mail/bozo    # 环境变量名使用大写。
20 export MAIL_DIRECTORY
21
22
23 GetAnswer ()                           # 函数名采用大小写混合的方式。
```

```

24 24 {
25   prompt=$1
26   echo -n $prompt
27   read answer
28   return $answer
29 }
30
31 31 GetAnswer "What is your favorite number? "
32 favorite_number=$?
33 echo $favorite_number
34
35
36 36 _uservariable=23          # 语法上可以这么起名，但是不推荐。
37 # 用户定义的变量名最好不要以下划线开头。
38 # 因为以下划线开头的变量，一般都保留，作为系统变量。

```

- [退出码](#)最好也采用具有系统性的或有意义的命名方式.

```

1 1 E_WRONG_ARGS=65
2   2 ...
3   3 ...
4   4 exit $E_WRONG_ARGS

```

也请参考[Appendix D](#).

最后，我们建议采用/usr/include/sysexits.h中的定义作为退出码，虽然这些定义主要用于C/C++编程语言。

- 在脚本调用中使用标准化的参数标志. 最后，我们建议使用下面的参数集.

1	1 -a	全部： 返回全部信息(包括隐藏的文件信息).
2	2 -b	摘要： 缩减版本，通常用于其它版本. 通常用于其它脚本.
3	3 -c	拷贝，连接，等等.
4	4 -d	日常的： 使用全天的信息，
5	5	而不仅仅是特定用户或特定实例的信息.
6	6 -e	扩展/详细描述： (通常不包括隐藏文件信息).
7	7 -h	帮助： 详细的使用方法，附加信息，讨论，帮助.
8	8	也请参考-V.
9	9 -l	打印出脚本的输出记录.
10	10 -m	手册： 显示基本命令的man页.
11	11 -n	数字： 仅使用数字数据.
12	12 -r	递归： 这个目录中所有的文件(也包含所有子目录).
13	13 -s	安装&文件维护： 这个脚本的配置文件.
14	14 -u	用法： 列出脚本的调用方法.
15	15 -v	详细信息： 只读输出，或多或少的会做一些格式化.
16	16 -V	版本/许可/版权Copy(right left)/捐助(邮件列表).

也请参考[Section F.1](#).

-
- 将一个复杂脚本分割成一些简单的模块. 使用合适的函数来实现模块的功能. 请参考[例子34-4](#).
 - 如果有更简单的结构可以使用的话, 就不要使用复杂的结构.

```
1
2 1 COMMAND
3 2 if [ $? -eq 0 ]
4 3 ...
5 4 # 多余, 而且不好理解.
6 5
7 6 if COMMAND
8 7 ...
9 8 # 更简练(可能会损失一些可读性).
```

... 当我阅读UNIX中Bourne shell (/bin/sh)部分的源代码时.
我被震惊了,
有多少简单的算法被恶心的编码风格弄得令人看不懂,
并且因此变得没用.
我问我自己,
”有人会对这种代码感到骄傲和自豪么?”
— Landon Noll

注意事项

1. 在这种上下文中所说的”魔法数字”与用来指明文件类型的[魔法数字](#), 在含义上完全不同.

第33章 杂项

本章目录

1. 交互与非交互式的交互与非交互式的shell和脚本
 2. Shell包装
 3. 测试和比较：一种可选的方法
 4. 递归
 5. 将脚本“彩色化”
 6. 优化
 7. 各种小技巧
 8. 安全问题
 9. 可移植性问题
 10. Windows下的shell脚本
-

1 交互与非交互式的交互与非交互式的shell和脚本

交互式的shell会在tty上从用户输入中读取命令. 另一方面, 这样的shell能在启动时读取启动文件, 显示一个提示符, 并默认激活作业控制. 也就是说, 用户可以与shell交互.

shell所运行的脚本通常都是非交互的shell. 但是脚本仍然可以访问它的tty. 甚至可以在脚本中模拟一个交互式的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 # 这个例子脚本, 还有上面那么多的解释
13 # 都是由Stephane Chazelas提供的(再次感谢).
```

让我们考虑一个需要用户输入的交互式脚本, 这种脚本通常都要使用read语句(请参考[例子11-3](#)). 但是”现实的情况”肯定要比这复杂的多. 就目前的情况来看, 交互式脚本通常都绑定在一个tty设备上, 换句话说, 用户都是在控制终端或xterm上来调用脚本的.

初始化脚本和启动脚本都是非交互式的, 因为它们都不需要人为干预, 都是自动运行的. 许多管理脚本和系统维护脚本也同样是非交互式的. 对于那些不需要经常变化的, 重复性的任务, 应该交给非交互式的脚本来自动完成.

非交互式的脚本可以在后台运行, 但是如果交互式脚本在后台运行的话, 就会被挂起, 因为它们在等待永远不会到来的输入. 如果想解决后台运行交互式脚本的问题, 可以使用带有expect命令的脚本, 或者在脚本中嵌入[here document](#)来提供交互式脚本所需要的输入. 最简单的办法其实就是将一个文件重定向给read命令, 来提供它所需要的输入(read variable <file>). 通过使用上述方法, 就可以编写出通用目的脚本, 这种脚本即可以运行在交互模式下, 也可以运行在非交互模式下.

如果脚本需要测试一下自己是否运行在交互式shell中, 那么一个简单的办法就是察看是否存在提示符(prompt)变量, 也就是察看一下变量\$PS1是否被设置. (如果脚本需要用户输入, 那么脚本就需要显示提示符.)

```
1 if [ -z $PS1 ] # 没有提示符?
2 then
3     # 非交互式
4     ...
5 else
6     # 交互式
7     ...
8 fi
```

另一种办法，脚本可以测试一下标志\$-中是否存在选项”i”。

```
1 case $- in
2 *i*)    # 交互式shell
3 ;;
4 *)      # 非交互式shell
5 ;;
6 # ("UNIX F.A.Q."的惯例， 1993)
```

2 Shell包装

”包装”脚本指的是内嵌系统命令或工具的脚本，并且这种脚本保留了传递给命令的一系列参数。[\[1\]](#) 因为包装脚本中包含了许多带有参数的命令，使它能够完成特定的目的，所以这样就大大简化了命令行的输入。这对于sed和awk命令特别有用。

sed或awk脚本通常都是在命令行中被调用的，使用的形式一般为sed -e 'commands' 或awk 'commands'。将这样的脚本(译者注：指的是包装了sed和awk的脚本)嵌入到Bash脚本中将会使调用更加简单，并且还可以”重复利用”。也可以将sed与awk的功能结合起来使用，比如，可以将一系列sed命令的输出通过管道传递给awk。还可以保存为可执行文件，这样你就可以重复的调用它了，如果功能不满足，你还可以修改它，这么做可以让省去每次都在命令行上输入命令的麻烦。

例子33-1. shell包装

```
1 #!/bin/bash
2
3 # 这个简单的脚本可以把文件中所有的空行删除。
4 # 没做参数检查。
5 #
6 # 你或许想添加如下代码：
7 #
8 # E_NOARGS=65
9 # if [ -z "$1" ]
10 # then
11 #   echo "Usage: `basename $0` target-file"
12 #   exit $E_NOARGS
13 # fi
14
15
16 # 这个脚本调用起来的效果，
17 # 等价于从命令行上调用：
18 # sed -e '/^$/d' filename.
19
20 sed -e '/^$/d "$1"
21 # '-e'意味着后边跟的是"编辑"命令。(在这里是可选的)。
22 # '^'匹配行首，'$'匹配行尾。
23 # 这条语句用来匹配行首与行尾之间什么都没有的行，
24 #+ 即空白行。
25 # 'd'为删除命令。
26
27 # 将命令行参数引用起来，
28 #+ 就意味着可以在文件名中使用空白字符或者特殊字符。
29
30 # 注意，这个脚本其实并不能真正的修改目标文件。
31 # 如果你想保存修改，可以将它的输出重定向。
32
```

```
33 exit 0
```

例子33-2. 稍微复杂一些的shell包装

```
1 #!/bin/bash
2
3 # "替换", 这个脚本的用途:
4 #+ 将一个文件中的某个字符串(或匹配模式), 替换为另一个字符串(或匹配模式),
5 #+ 比如, "subst Smith Jones letter.txt".
6
7 ARGS=3          # 这个脚本需要3个参数.
8 E_BADARGS=65    # 传递给脚本的参数个数不对.
9
10 if [ $# -ne "$ARGS" ]
11 # 测试脚本的参数个数(这是个好办法).
12 then
13     echo "Usage: `basename $0` old-pattern new-pattern filename"
14     exit $E_BADARGS
15 fi
16
17 old_pattern=$1
18 new_pattern=$2
19
20 if [ -f "$3" ]
21 then
22     file_name=$3
23 else
24     echo "File \"\$3\" does not exist."
25     exit $E_BADARGS
26 fi
27
28
29 # 下面是实现功能的代码.
30
31 # -----
32 sed -e "s/$old_pattern/$new_pattern/g" $file_name
33 # -----
34
35 # 's'在sed中是替换命令,
36 #+ /pattern/表示匹配模式.
37 # "g", 即全局标志, 用来自动替换掉每行中
38 #+ 出现的全部$old_pattern模式, 而不仅仅替换掉第一个匹配.
39 # 如果想深入了解, 可以参考'sed'命令的相关书籍.
40
41 exit 0      # 成功调用脚本, 将会返回0.
```

例子33-3. 一个通用的shell包装, 用来写日志文件

```
1 #!/bin/bash
2 # 通用的shell包装,
3 #+ 执行一个操作, 然后把所作的操作写入到日志文件中.
4
5 # 需要设置如下两个变量.
6 OPERATION=
7 # 可以是一个复杂的命令链,
8 #+ 比如awk脚本或者一个管道 . .
9 LOGFILE=
10 # 命令行参数, 不管怎么样, 操作一般都需要参数.
11 # (译者注: 这行解释的是下面的OPTIONS变量, 不是LOGFILE.)
12
13
14 OPTIONS="$@"
15
16
17 # 记录下来.
18 echo "'date' + 'whoami' + $OPERATION \"$@\" >> $LOGFILE"
19 # 现在, 执行操作.
20 exec $OPERATION "$@"
21
22 # 必须在操作执行之前, 记录到日志文件中.
23 # 为什么?
```

例子33-4. 包装awd脚本的shell包装

```
1 #!/bin/bash
2 # pr-ascii.sh: 打印ASCII码的字符表.
3
4 START=33    # 可打印的ASCII字符的范围(十进制).
5 END=125
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, $1)}'
13 # 在这种上下文中, 不会运行Bash内建的printf命令:
14 #     printf "%c" "$i"
15 done
16
17 exit 0
18
19
```

```

20 # 十进制 16进制 字符
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 # 将脚本的输出重定向到一个文件中,
36 #+ 或者通过管道传递给"more": sh pr-asc.sh | more

```

例子33-5. 另一个包装awd脚本的shell包装

```

1#!/bin/bash
2
3# 给目标文件添加(由数字组成的)指定的一列.
4
5ARGS=2
6E_WRONGARGS=65
7
8if [ $# -ne "$ARGS" ] # 检查命令行参数个数是否正确.
9then
10  echo "Usage: `basename $0` filename column-number"
11  exit $E_WRONGARGS
12fi
13
14filename=$1
15column_number=$2
16
17# 将shell变量传递给脚本的awk部分, 需要一点小技巧.
18# 一种办法是, 将awk脚本中的Bash脚本变量,
19#+ 强引用起来.
20# '$$BASH_SCRIPT_VAR'
21#
22# 在下面的内嵌awd脚本中, 就会这么做.
23# 请参考awk的相关文档来了解更多的细节.
24
25# 多行awk脚本的调用格式为: awk '.....'
26

```

```

27
28 # 开始awk脚本.
29 # -----
30 awk '
31
32 { total += $'${column_number}' ,
33 }
34 END {
35     print total
36 }
37
38 , "$filename"
39 # -----
40 # 结束awk脚本.
41
42
43 # 将shell变量传递给内嵌awk脚本可能是不安全的,
44 #+ 所以Stephane Chazelas提出了下边这种替代方法:
45 # -----
46 # awk -v column_number="$column_number" ,
47 # { total += $column_number
48 #
49 # END {
50 #     print total
51 # }' "$filename"
52 # -----
53
54
55 exit 0

```

如果那些脚本需要的是一个全功能(多合一)的工具, 一把瑞士军刀, 那么只能使用Perl了. Perl兼顾sed和awk的能力, 并且包含了C的很大的一个子集, 用于引导. 它是模块化的, 并且包含从面向对象编程到厨房水槽的所有功能(译者注: 就是表示Perl无所不能). 小段的Perl脚本可以内嵌到shell脚本中, 以至于有人声称Perl可以完全代替shell脚本(不过本文作者对此持怀疑态度).

例子33-6. 将Perl嵌入到Bash脚本中

```

1#!/bin/bash
2
3# Shell命令可以放到Perl脚本的前面.
4echo "This precedes the embedded Perl script within \"$0\"."
5echo =====
6
7perl -e 'print "This is an embedded Perl script.\n";'
8# 类似于sed, Perl也可以使用"-e"选项.
9
10echo =====

```

```
11 echo "However, the script may also contain shell and system commands."  
12  
13 exit 0
```

甚至可以将Bash脚本和Perl脚本放到同一个文件中。这依赖于如何调用这个脚本，或者执行Bash部分，或者执行Perl部分。

例子33-7. 将Bash和Perl脚本写到同一个文件中

```
1 #!/bin/bash  
2 # bashandperl.sh  
3  
4 echo "Greetings from the Bash part of the script."  
5 # 这里可以放置更多的Bash命令。  
6  
7 exit 0  
8 # 脚本的Bash部分结束。  
9  
10 # =====  
11  
12 #!/usr/bin/perl  
13 # 脚本的这部分必须使用-x选项来调用。  
14  
15 print "Greetings from the Perl part of the script.\n";  
16 # 这里可以放置更多的Perl命令。  
17  
18 # 脚本的Perl部分结束。
```

```
1  
2 bash$ bash bashandperl.sh  
3 Greetings from the Bash part of the script.  
4  
5  
6 bash$ perl -x bashandperl.sh  
7 Greetings from the Perl part of the script.
```

注意事项

- 事实上，Linux中相当一部分工具都是shell包装脚本。比如/usr/bin/pdf2ps, /usr/bin/batch, 和/usr/X11R6/bin/xmkmf。

3 测试和比较：一种可选的方法

对于测试来说，[[]]结构可能比[]结构更合适。同样地，在算术比较中，使用(())结构可能会更有用。

```
1
2 a=8
3
4 # 下面所有的比较都是等价的。
5 test "$a" -lt 16 && echo "yes, $a < 16"          # "与列表"
6 /bin/test "$a" -lt 16 && echo "yes, $a < 16"
7 [ "$a" -lt 16 ] && echo "yes, $a < 16"
8 [[ $a -lt 16 ]] && echo "yes, $a < 16"           # 在[[ ]]和(( ))结构中,
9 (( a < 16 )) && echo "yes, $a < 16"             # 不用将变量引用起来。
10
11 city="New York"
12 # 同样地，下面所有的比较都是等价的。
13
14 # 按照ASCII顺序比较。
15 test "$city" \< Paris && echo "Yes, Paris is greater than $city"
16 /bin/test "$city" \< Paris && echo "Yes, Paris is greater than $city"
17 [ "$city" \< Paris ] && echo "Yes, Paris is greater than $city"
18
19 # 不需要引用$city。
20 [[ $city < Paris ]] && echo "Yes, Paris is greater than $city"
21
22 # 感谢，S.C.
```

4 测试和比较：一种可选的方法

脚本是否可以递归调用自身？当然可以。

例子33-8. 递归调用自身的(没用的)脚本

```
1 #!/bin/bash
2 # recurse.sh
3
4 # 脚本能否递归地调用自己？
5 # 是的，但这有什么实际的用处吗？
6 # (看下面的。)
7
8 RANGE=10
9 MAXVAL=9
10
11 i=$RANDOM
12 let "i %= $RANGE" # 在0到$RANGE - 1之间，产生一个随机数。
13
14 if [ "$i" -lt "$MAXVAL" ]
15 then
16   echo "i = $i"
17   ./$0          # 脚本递归地产生自己的一个新实例，并调用。
18 fi             # 每个子脚本都做同样的事情，until
19             #+ 直到产生的变量$i等于$MAXVAL为止。
20
21 # 如果使用"while"循环来代替"if/then"测试结构的话，会产生问题。
22 # 解释一下为什么。
23
24 exit 0
25
26 # 注意：
27 # -----
28 # 脚本想要正常的工作，就必须具备可执行权限。
29 # 即使使用"sh"命令来调用它，但是没有设置正确的权限一样会导致问题。
30 # 解释一下原因。
```

例子33-9. 递归调用自身的(有用的)脚本

```
1 #!/bin/bash
2 # pb.sh: 电话本
3
4 # 由Rick Boivie编写，已经得到作者授权，可以在本书中使用。
5 # 本书作者做了一些修改。
6
7 MINARGS=1      # 脚本至少需要一个参数。
8 DATAFILE=./phonebook
```

```

9      9          # 当前目录下,
10     10         +#+ 必须有一个名字为"phonebook"的数据文件.
11    11 PROGNAME=$0
12    12 E_NOARGS=70  # 未传递参数错误.
13
14   14 if [ $# -lt $MINARGS ]; then
15     15     echo "Usage: \"$PROGNAME\" data"
16     16     exit $E_NOARGS
17   17 fi
18
19
20  20 if [ $# -eq $MINARGS ]; then
21    21     grep $1 "$DATAFILE"
22    22     # 如果文件$DATAFILE不存在, 'grep'就会打印一个错误信息.
23  23 else
24    24     ( shift; "$PROGNAME" $* ) | grep $1
25    25     # 脚本递归调用自身.
26  26 fi
27
28  28 exit 0      # 脚本在此退出.
29
30  29             #+# 因此,在这句之后,
31
32  32 # -----
33  33 "phonebook"数据文件的例子:
34
35  35 John Doe      1555 Main St., Baltimore, MD 21228      (410) 222-3333
36  36 Mary Moe       9899 Jones Blvd., Warren, NH 03787      (603) 898-3232
37  37 Richard Roe   856 E. 7th St., New York, NY 10009      (212) 333-4567
38  38 Sam Roe        956 E. 8th St., New York, NY 10009      (212) 444-5678
39  39 Zoe Zenobia   4481 N. Baker St., San Francisco, SF 94338  (415) 501-1631
40  40 # -----
41
42  42 $bash pb.sh Roe
43  43 Richard Roe   856 E. 7th St., New York, NY 10009      (212) 333-4567
44  44 Sam Roe        956 E. 8th St., New York, NY 10009      (212) 444-5678
45
46  46 $bash pb.sh Roe Sam
47  47 Sam Roe        956 E. 8th St., New York, NY 10009      (212) 444-5678
48
49  49 # 如果给脚本传递的参数超过了一个,
50  50 #+# 那这个脚本就*只*会打印包含所有参数的行.

```

例子33-10. 另一个递归调用自身的(有用的)脚本

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

```
2 # usrmnt.sh, 由Anthony Richardson编写,
3 # 经过作者授权, 可以在本书中使用.
4
5 # 用法:      usrmnt.sh
6 # 描述: 挂载设备, 调用这个脚本的用户必须属于
7 #          /etc/sudoers文件中的MNTUSERS组.
8
9 #
10 # 这是一个用户挂载设备的脚本, 脚本将会使用sudo来递归的调用自身.
11 # 只有拥有合适权限的用户才能使用
12
13 #   usermount /dev/fd0 /mnt/floppy
14
15 # 来代替
16
17 #   sudo usermount /dev/fd0 /mnt/floppy
18
19 # 我使用相同的技术来处理我所有的sudo脚本,
20 #+ 因为我觉得它很方便.
21 #
22
23 # 如果没有设置SUDO_COMMAND变量, 而且我们并没有处于sudo运行的状态下
24 #+ (译者注: 也就是说第一次运行, 还没被递归), 这样就会开始递归了.
25 # 传递用户的真实id和组id . .
26
27 if [ -z "$SUDO_COMMAND" ]
28 then
29     mntusr=$(id -u) grpusr=$(id -g) sudo $0 $*
30     exit 0
31 fi
32
33 # 如果我们处于sudo调用自身的状态中(译者注: 就是说处于递归中),
34 # 那么我们就会运行到这里.
35 /bin/mount $* -o uid=$mntusr,gid=$grpusr
36
37 exit 0
38
39 # 附注(脚本作者添加的):
40 #
41
42 # 1) Linux允许在/etc/fstab文件中使用"users"选项,
43 #    以便于任何用户都可以挂载可移动设备.
44 #    但是, 在服务器上,
45 #    我希望只有一小部分用户可以访问可移动设备.
46 #    我发现使用sudo可以给我更多的控制空间.
```

```
47 45
48 46 # 2) 我还发现, 通过使用组,
49 # 我能够更容易的完成这个任务.
50
51 49 # 3) 这个方法可以将root访问mount命令的权利,
52 # 赋予任何具有合适权限的用户,
53 # 所以一定要小心那些被你赋予访问权限的用户.
54 # 你可以开发出类似于mntfloppy, mntcdrom,
55 # 和mntsamba脚本, 将访问类型分类,
56 # 然后你就可以使用上面所讲的这种技术,
57 # 获得对mount命令更好的控制.
```

►: 过多层次的递归会耗尽脚本的栈空间, 引起段错误.

5 将脚本彩色化

ANSI [1] 定义了屏幕属性的转义序列集合, 比如说粗体文本, 前景与背景颜色. DOS批处理文件通常都使用ANSI转义码来控制颜色输出, Bash脚本也是这么做的.

例子33-11. 一个“彩色的”地址数据库

```
1
2 #!/bin/bash
3 # ex30a.sh: 脚本ex30.sh的“彩色”版本.
4 #           没被加工处理过的地址数据库
5
6
7 clear                                # 清屏.
8
9 echo -n "          "
10 echo -e '\E[37;44m"\033[1mContact List\033[0m"
11                                     # 在蓝色背景下的白色.
12 echo; echo
13 echo -e "\033[1mChoose one of the following persons:\033[0m"
14                                     # 粗体
15 tput sgr0
16 echo "(Enter only the first letter of name.)"
17 echo
18 echo -en '\E[47;34m"\033[1mE\033[0m"    # 蓝色
19 tput sgr0                                # 将颜色重置为“常规”.
20 echo "vans, Roland"                      # "[E]vans, Roland"
21 echo -en '\E[47;35m"\033[1mJ\033[0m"    # 红紫色
22 tput sgr0
23 echo "ones, Mildred"
24 echo -en '\E[47;32m"\033[1mS\033[0m"    # 绿色
25 tput sgr0
26 echo "mith, Julie"
27 echo -en '\E[47;31m"\033[1mZ\033[0m"    # 红色
28 tput sgr0
29 echo "ane, Morris"
30 echo
31
32 read person
33
34 case "$person" in
35 # 注意, 变量被引用起来了.
36
37     "E" | "e" )
38     # 大小写的输入都能接受.
39     echo
```

```

40 echo "Roland Evans"
41 echo "4321 Floppy Dr."
42 echo "Hardscrabble, CO 80753"
43 echo "(303) 734-9874"
44 echo "(303) 734-9892 fax"
45 echo "revans@zyy.net"
46 echo "Business partner & old friend"
47 ;;
48
49 "J" | "j" )
50 echo
51 echo "Mildred Jones"
52 echo "249 E. 7th St., Apt. 19"
53 echo "New York, NY 10009"
54 echo "(212) 533-2814"
55 echo "(212) 533-9972 fax"
56 echo "milliej@loisaida.com"
57 echo "Girlfriend"
58 echo "Birthday: Feb. 11"
59 ;;
60
61 # 稍后为Smith & Zane添加信息.
62
63     * )
64 # 默认选项.
65 # 空输入(直接按回车)也会在这被匹配.
66 echo
67 echo "Not yet in database."
68 ;;
69
70 esac
71
72 tput sgr0          # 将颜色重置为"常规".
73
74 echo
75
76 exit 0

```

例子33-12. 画一个盒子

```

1#!/bin/bash
2# Draw-box.sh: 使用ASCII字符画一个盒子.
3
4# 由Stefano Palmeri编写, 本书作者做了少量修改.
5# 经过授权, 可以在本书中使用.
6

```

```
7 #####
8 ##### draw_box函数注释 #####
9 #### draw_box函数注释 #####
10 #
11 # "draw_box"函数可以让用户
12 #+ 在终端上画一个盒子.
13 #
14 # 用法: draw_box ROW COLUMN HEIGHT WIDTH [COLOR]
15 # ROW和COLUMN用来定位你想要
16 #+ 画的盒子的左上角.
17 # ROW和COLUMN必须大于0,
18 #+ 并且要小于当前终端的尺寸.
19 # HEIGHT是盒子的行数, 并且必须 >0 .
20 # HEIGHT + ROW 必须 <= 终端的高度.
21 # WIDTH是盒子的列数, 必须 >0 .
22 # WIDTH + COLUMN 必须 <= 终端的宽度.
23 #
24 # 例如: 如果你的终端尺寸为20x80,
25 # draw_box 2 3 10 45 是合法的
26 # draw_box 2 3 19 45 的HEIGHT是错的 (19+2 > 20)
27 # draw_box 2 3 18 78 的WIDTH是错的 (78+3 > 80)
28 #
29 # COLOR是盒子边框的颜色.
30 # 这是第5个参数, 并且是可选的.
31 # 0=黑 1=红 2=绿 3=棕褐 4=蓝 5=紫 6=青 7=白.
32 # 如果你传递给函数的参数错误,
33 #+ 它将会退出, 并返回65,
34 #+ 不会有消息打印到stderr上.
35 #
36 # 开始画盒子之前, 会清屏.
37 # 函数内不包含清屏命令.
38 # 这样就允许用户画多个盒子, 甚至可以叠加多个盒子.
39 #
40 #### draw_box函数注释结束 #####
41 #####
42 #
43 draw_box(){
44 #
45 =====#
46 HORZ="-"
47 VERT="|"
48 CORNER_CHAR="+"
49 #
50 MINARGS=4
51 E_BADARGS=65
```

```

52 #=====
53
54
55 if [ $# -lt "$MINARGS" ]; then          # 如果参数小于4, 退出.
56     exit $E_BADARGS
57 fi
58
59 # 找出参数中非数字的字符.
60 # 还有其他更好的方法么(留给读者作为练习?).
61 if echo $@ | tr -d [:blank:] | tr -d [:digit:] | grep . &> /dev/null; then
62     exit $E_BADARGS
63 fi
64
65 BOX_HEIGHT='expr $3 - 1'    # 必须-1, 因为边角的"+"是
66 BOX_WIDTH='expr $4 - 1'      #+ 高和宽共有的部分.
67 T_ROWS='tput lines'        # 定义当前终端的
68 T_COLS='tput cols'         #+ 长和宽的尺寸.
69
70 if [ $1 -lt 1 ] || [ $1 -gt $T_ROWS ]; then    # 开始检查参数
71     exit $E_BADARGS                         #+ 是否正确.
72 fi
73 if [ $2 -lt 1 ] || [ $2 -gt $T_COLS ]; then
74     exit $E_BADARGS
75 fi
76 if [ `expr $1 + $BOX_HEIGHT + 1` -gt $T_ROWS ]; then
77     exit $E_BADARGS
78 fi
79 if [ `expr $2 + $BOX_WIDTH + 1` -gt $T_COLS ]; then
80     exit $E_BADARGS
81 fi
82 if [ $3 -lt 1 ] || [ $4 -lt 1 ]; then
83     exit $E_BADARGS
84 fi                                # 参数检查结束.
85
86 plot_char(){                      # 函数内的函数.
87     echo -e "\E[${1};${2}H"${3}
88 }
89
90 echo -ne "\E[3${5}m"                # 如果定义了, 就设置盒子边框的颜色.
91
92 # 开始画盒子
93
94 count=1                            # 使用plot_char函数
95 for (( r=$1; count<=$BOX_HEIGHT; r++)); do      #+ 画垂直线.
96     plot_char $r $2 $VERT

```

```

97 let count=count+1
98 done
99
100 count=1
101 c='expr $2 + $BOX_WIDTH'
102 for (( r=$1; count<=$BOX_HEIGHT; r++)); do
103     plot_char $r $c $VERT
104     let count=count+1
105 done
106
107 count=1                      # 使用plot_char函数
108 for (( c=$2; count<=$BOX_WIDTH; c++)); do      #+ 画水平线.
109     plot_char $1 $c $HORZ
110     let count=count+1
111 done
112
113 count=1
114 r='expr $1 + $BOX_HEIGHT'
115 for (( c=$2; count<=$BOX_WIDTH; c++)); do
116     plot_char $r $c $HORZ
117     let count=count+1
118 done
119
120 plot_char $1 $2 $CORNER_CHAR          # 画盒子的角.
121 plot_char $1 'expr $2 + $BOX_WIDTH' +
122 plot_char 'expr $1 + $BOX_HEIGHT' $2 +
123 plot_char 'expr $1 + $BOX_HEIGHT' 'expr $2 + $BOX_WIDTH' +
124
125 echo -ne "\E[Om"                  # 恢复原来的颜色.
126
127 P_ROWS='expr ${T_ROWS} - 1'      # 在终端的底部打印提示符.
128
129 echo -e "\E[$P_ROWS;1H"
130 }
131
132
133 # 现在, 让我们开始画盒子吧.
134 clear                         # 清屏.
135 R=2      # 行
136 C=3      # 列
137 H=10     # 高
138 W=45     # 宽
139 col=1    # 颜色(红)
140 draw_box $R $C $H $W $col    # 画盒子.
141

```

```
142 exit 0
143
144 # 练习:
145 # -----
146 # 添加一个选项, 用来支持可以在所画的盒子中打印文本.
```

最简单的, 也可能是最有用的ANSI转义序列是加粗文本, `\te033[1m ... \033[0m`. `\033`代表转义, “[1”打开加粗属性, 而”[0”关闭加粗属性. ”m”表示转义序列结束.

```
1 bash$ echo -e "\033[1mThis is bold text.\033[0m"
```

一种类似的转义序列用来切换下划线属性(在rxvt 和aterm上).

```
1 bash$ echo -e "\033[4mThis is underlined text.\033[0m"
```

►: echo命令的-e选项用来启用转义序列.

其他的转义序列可用于修改文本和背景色.

```
1 bash$ echo -e '\E[34;47mThis prints in blue.'; tput sgr0
2
3
4 bash$ echo -e '\E[33;44m"yellow text on blue background"; tput sgr0
5
6
7 bash$ echo -e '\E[1;33;44m"BOLD yellow text on blue background"; tput sgr0
```

►: 通常情况下, 为浅色的前景文本设置粗体属性比较好.

`tput sgr0`把终端设置恢复为原样. 如果省略这一句, 那么这个终端所有后续的输出还会是蓝色.

►: 因为`tput sgr0`在某些环境下不能恢复终端设置, `echo -ne \E[0m`可能是更好的选择.

可以在有色的背景上, 使用下面的模板, 在上面写有色的文本.

```
1 echo -e '\E[COLOR1;COLOR2mSome text goes here.'
```

”`\E[`”开始转义序列. 以分号分隔的数字”COLOR1”和”COLOR2”分别指定了前景色和背景色, 数值与色彩之间的对应, 请参考下面的表格. (数值的顺序其实没关系, 因为前景色和背景色的数值都落在互不重叠的范围中.) ”m”用来终止转义序列, 文本紧跟在”m”的后面.

也要注意, [单引号](#)将`echo -e`后面的命令序列都引用了起来.

下表的数值是在rxvt终端上运行的结果. 具体的结果可能和在其他终端上运行的结果不同.

表格33-1. 转义序列中颜色与数值的对应

颜色	前景	背景
黑	30	40
红	31	41
绿	32	42
黄	33	43
蓝	34	44
洋红	35	45
青	36	46
白	37	47

例子33-13. 显示彩色文本

```
1 #!/bin/bash
2 # color-echo.sh: 使用颜色来显示文本消息.
3
4 # 可以按照你自己的目的来修改这个脚本.
5 # 这比将颜色数值写死更容易.
6
7 black='\E[30;47m'
8 red='\E[31;47m'
9 green='\E[32;47m'
10 yellow='\E[33;47m'
11 blue='\E[34;47m'
12 magenta='\E[35;47m'
13 cyan='\E[36;47m'
14 white='\E[37;47m'
15
16
17 alias Reset="tput sgr0"      # 不用清屏,
18                      ##+ 将文本属性重置为正常情况.
19
20
21 cecho ()                  # Color-echo.
22                      ## 参数$1 = 要显示的信息
23                      ## 参数$2 = 颜色
24 {
25     local default_msg="No message passed."
26                      ## 其实并不一定非的是局部变量.
27
28     message=${1:-$default_msg}    # 默认为default_msg.
29     color=${2:-$black}          # 如果没有指定, 默认为黑色.
30
31     echo -e "$color"
32     echo "$message"
33     Reset                     # 重置文本属性.
34
35     return
36 }
37
38
39 # 现在, 让我们试一下.
40 #
41 cecho "Feeling blue..." $blue
42 cecho "Magenta looks more like purple." $magenta
43 cecho "Green with envy." $green
44 cecho "Seeing red?" $red
```

```

45 cecho "Cyan, more familiarly known as aqua." $cyan
46 cecho "No color passed (defaults to black)."
47     # 缺少$color参数.
48 cecho "\\"Empty\" color passed (defaults to black)." ""
49     # 空的$message参数.
50 cecho
51     # 缺少$message和$color参数.
52 cecho """
53     # 空的$message和$color参数.
54 # -----
55
56 echo
57
58 exit 0
59
60 # 练习:
61 # -----
62 # 1) 为'cecho ()'函数添加"粗体"属性.
63 # 2) 为彩色背景添加选项.

```

例子33-14. “赛马”游戏

```

1#!/bin/bash
2# horserace.sh: 一个非常简单的模拟赛马的游戏.
3# 作者: Stefano Palmeri
4# 经过授权可以在本书中使用.
5
6#####
7# 脚本目的:
8# 使用转义序列和终端颜色进行游戏.
9#
10# 练习:
11# 编辑脚本, 使它运行起来更具随机性,
12#+ 建立一个假的赌场 . . .
13# 嗯 . . . 嗯 . . . 这种开场让我联想起一部电影 . . .
14#
15# 脚本将会给每匹马分配一个随机障碍.
16# 按照马的障碍来计算几率,
17#+ 并且使用一种欧洲(?)的风格表达出来.
18# 比如: 几率(odds)=3.75的话, 那就意味着如果你押$1,
19#+ 你就会赢得$3.75.
20#
21# 此脚本已经在GNU/Linux操作系统上测试过,
22#+ 测试终端有xterm, rxvt, 和konsole.
23# 测试机器上安装的是AMD 900 MHz处理器,
24#+ 平均比赛时间为75秒.

```

```
25 # 如果使用更快的机器，那么比赛用时会更少。
26 # 所以，如果你想增加比赛的悬念，可以重置变量USLEEP_ARG。
27 #
28 # 本脚本由Stefano Palmeri编写。
29 #####
30
31 E_RUNERR=65
32
33 # 检查一下md5sum和bc是否已经被安装。
34 if ! which bc &> /dev/null; then
35     echo bc is not installed.
36     echo "Can't run . . . "
37     exit $E_RUNERR
38 fi
39 if ! which md5sum &> /dev/null; then
40     echo md5sum is not installed.
41     echo "Can't run . . . "
42     exit $E_RUNERR
43 fi
44
45 # 设置下面的变量将会降低脚本的运行速度。
46 # 它会作为参数，传递给usleep命令(man usleep)，
47 #+ 并且单位是微秒(500000微秒 = 半秒)。
48 USLEEP_ARG=0
49
50 # 如果脚本被Ctl-C中断，那就清除临时目录，
51 #+ 恢复终端光标和终端颜色。
52 trap 'echo -en "\E[?25h"; echo -en "\E[0m"; stty echo; \
53 tput cup 20 0; rm -fr $HORSE_RACE_TMP_DIR' TERM EXIT
54 # 请参考与调试相关的章节，可以获得'trap'命令的详细用法。
55
56 # 为脚本设置一个唯一的(实际上不是绝对唯一)临时目录名。
57 HORSE_RACE_TMP_DIR=$HOME/.horserace-'date +%s'-'head -c10 \
58 /dev/urandom | md5sum | head -c30'
59
60 # 创建临时目录，并移动到该目录下。
61 mkdir $HORSE_RACE_TMP_DIR
62 cd $HORSE_RACE_TMP_DIR
63
64
65 # 这个函数将会把光标移动到行为$1，列为$2的位置上，然后打印$3。
66 # 例如："move_and_echo 5 10 linux"等价与"tput cup 4 9; echo linux"，
67 #+ 但是使用一个命令代替了两个命令。
68 # 注意："tput cup"定义0 0位置，为终端左上角，
69 #+ 而echo定义1 1位置，为终端左上角。
```

```
70 move_and_echo() {
71     echo -ne "\E[${1};${2}H\"$3"
72 }
73
74 # 此函数用来产生1-9之间的伪随机数.
75 random_1_9 () {
76     head -c10 /dev/urandom | md5sum | tr -d [a-z] | tr -d 0 | cut -c1
77 }
78
79 # 在画马的时候, 这两个函数用来模拟"移动".
80 draw_horse_one() {
81     echo -n "//$MOVE_HORSE//"
82 }
83 draw_horse_two(){
84     echo -n "\\\\$MOVE_HORSE\\\
85 }
86
87
88 # 定义当前终端尺寸.
89 N_COLS='tput cols'
90 N_LINES='tput lines'
91
92 # 至少需要一个20(行) X 80(列)的终端. 检查一下.
93 if [ $N_COLS -lt 80 ] || [ $N_LINES -lt 20 ]; then
94     echo "'basename $0' needs a 80-cols X 20-lines terminal."
95     echo "Your terminal is ${N_COLS}-cols X ${N_LINES}-lines."
96     exit $E_RUNERR
97 fi
98
99
100 # 开始画赛场.
101
102 # 需要一个80字符的字符串. 见下面.
103 BLANK80='seq -s "" 100 | head -c80'
104
105 clear
106
107 # 将前景色与背景色设为白.
108 echo -ne '\E[37;47m'
109
110 # 将光标移动到终端的左上角.
111 tput cup 0 0
112
113 # 画6条白线.
114 for n in `seq 5`; do
```

```

115     echo $BLANK80      # 用这个80字符的字符串将终端变为彩色的.
116 done
117
118 # 将前景色设为黑色.
119 echo -ne '\E[30m'
120
121 move_and_echo 3 1 "START 1"
122 move_and_echo 3 75 FINISH
123 move_and_echo 1 5 "|"
124 move_and_echo 1 80 "|"
125 move_and_echo 2 5 "|"
126 move_and_echo 2 80 "|"
127 move_and_echo 4 5 "| 2"
128 move_and_echo 4 80 "|"
129 move_and_echo 5 5 "V 3"
130 move_and_echo 5 80 "V"
131
132 # 将前景色设置为红色.
133 echo -ne '\E[31m'
134
135 # 一些ASCII的艺术效果.
136 move_and_echo 1 8 "...@@@..@@@@@...@@@@@.@@...@...@@@@...@"
137 move_and_echo 2 8 ".@...@...@.....@...@...@.@@.....@"
138 move_and_echo 3 8 ".@@@@...@.....@...@@@@@.@@@...@"
139 move_and_echo 4 8 ".@...@...@.....@...@...@.@@.....@"
140 move_and_echo 5 8 ".@...@...@.....@...@...@.@@...@...@"
141 move_and_echo 1 43 "@@@@...@@@...@@@...@@@...@@@...@"
142 move_and_echo 2 43 "@...@.@@...@.@@...@.@@...@.@@...@"
143 move_and_echo 3 43 "@@@@...@@@...@@...@@...@@...@@...@"
144 move_and_echo 4 43 "@...@...@.@@...@.@@...@.@@...@.@@...@"
145 move_and_echo 5 43 "@...@.@@...@...@@...@.@@...@.@@...@"
146
147
148 # 将前景色和背景色设为绿色.
149 echo -ne '\E[32;42m'
150
151 # 画11行绿线.
152 tput cup 5 0
153 for n in `seq 11`; do
154     echo $BLANK80
155 done
156
157 # 将前景色设为黑色.
158 echo -ne '\E[30m'
159 tput cup 5 0

```

```
160
161 # 画栅栏.
162 echo "+++++++\n"
163 "+++++++\n"
164
165 tput cup 15 0
166 echo "+++++++\n"
167 "+++++++\n"
168
169 # 将前景色和背景色设回白色.
170 echo -ne '\E[37;47m'
171
172 # 画3条白线.
173 for n in `seq 3`; do
174     echo $BLANK80
175 done
176
177 # 将前景色设为黑色.
178 echo -ne '\E[30m'
179
180 # 创建9个文件, 用来保存障碍物.
181 for n in `seq 10 7 68`; do
182     touch $n
183 done
184
185 # 将脚本所要画的"马"设置为第一种类型.
186 HORSE_TYPE=2
187
188 # 为每匹"马"创建位置文件和几率文件.
189 #+ 在这些文件中, 保存马的当前位置,
190 #+ 类型和几率.
191 for HN in `seq 9`; do
192     touch horse_${HN}_position
193     touch odds_${HN}
194     echo \-1 > horse_${HN}_position
195     echo $HORSE_TYPE >> horse_${HN}_position
196     # 给马定义随机障碍物.
197     HANDICAP='random_1_9'
198     # 检查函数random_1_9是否返回一个有效值.
199     while ! echo $HANDICAP | grep [1-9] &> /dev/null; do
200         HANDICAP='random_1_9'
201     done
202     # 给马定义最后一个障碍物的位置.
203     LHP='expr $HANDICAP \* 7 + 3'
204     for FILE in `seq 10 7 $LHP`; do
```

```

205         echo $HN >> $FILE
206     done
207
208     # 计算几率.
209     case $HANDICAP in
210         1) ODDS=`echo $HANDICAP \* 0.25 + 1.25 | bc`
211             echo $ODDS > odds_${HN}
212             ;;
213         2 | 3) ODDS=`echo $HANDICAP \* 0.40 + 1.25 | bc`
214             echo $ODDS > odds_${HN}
215             ;;
216         4 | 5 | 6) ODDS=`echo $HANDICAP \* 0.55 + 1.25 | bc`
217             echo $ODDS > odds_${HN}
218             ;;
219         7 | 8) ODDS=`echo $HANDICAP \* 0.75 + 1.25 | bc`
220             echo $ODDS > odds_${HN}
221             ;;
222         9) ODDS=`echo $HANDICAP \* 0.90 + 1.25 | bc`
223             echo $ODDS > odds_${HN}
224     esac
225
226
227 done
228
229
230 # 打印几率.
231 print_odds() {
232 tput cup 6 0
233 echo -ne '\E[30;42m'
234 for HN in `seq 9`; do
235     echo "#$HN odds->" `cat odds_${HN}`
236 done
237 }
238
239 # 在起跑线上把马画出来.
240 draw_horses() {
241 tput cup 6 0
242 echo -ne '\E[30;42m'
243 for HN in `seq 9`; do
244     echo /\\$HN/\\"
245 done
246 }
247
248 print_odds
249

```

```
250 echo -ne '\E[47m'
251 # 等待按下回车键，按下之后就开始比赛。
252 # 转义序列，\E[?25l，禁用光标。
253 tput cup 17 0
254 echo -e '\E[?25l'Press [enter] key to start the race...
255 read -s
256
257 # 禁用了终端的常规echo功能。
258 # 这么做用来避免在比赛中，
259 #+ 按键所导致的"花"屏。
260 stty -echo
261
262 # -----
263 # 开始比赛。
264
265 draw_horses
266 echo -ne '\E[37;47m'
267 move_and_echo 18 1 $BLANK80
268 echo -ne '\E[30m'
269 move_and_echo 18 1 Starting...
270 sleep 1
271
272 # 设置终点线的列号。
273 WINNING_POS=74
274
275 # 定义比赛开始的时间。
276 START_TIME='date +%s'
277
278 # 下面的"while"结构需要使用COL变量。
279 COL=0
280
281 while [ $COL -lt $WINNING_POS ]; do
282
283     MOVE_HORSE=0
284
285     # 检查random_1_9函数是否返回了有效值。
286     while ! echo $MOVE_HORSE | grep [1-9] &> /dev/null; do
287         MOVE_HORSE='random_1_9'
288     done
289
290     # 定义"随机抽取的马"的原来类型和位置。
291     HORSE_TYPE='cat horse_${MOVE_HORSE}_position | tail -1'
292     COL=$((expr 'cat horse_${MOVE_HORSE}_position | head -1'))
293
294     ADD_POS=1
```

```

295      # 判断当前位置是否存在障碍物.
296      if seq 10 7 68 | grep -w $COL &> /dev/null; then
297          if grep -w $MOVE_HORSE $COL &> /dev/null; then
298              ADD_POS=0
299              grep -v -w $MOVE_HORSE $COL > ${COL}_new
300              rm -f $COL
301              mv -f ${COL}_new $COL
302          else ADD_POS=1
303      fi
304  else ADD_POS=1
305  fi
306  COL='expr $COL + $ADD_POS'
307  echo $COL > horse_${MOVE_HORSE}_position # 保存新位置.
308
309  # 选择要画出来的马的类型.
310  case $HORSE_TYPE in
311      1) HORSE_TYPE=2; DRAW_HORSE=draw_horse_two
312          ;;
313      2) HORSE_TYPE=1; DRAW_HORSE=draw_horse_one
314  esac
315  echo $HORSE_TYPE >> horse_${MOVE_HORSE}_position # 保存当前类型.
316
317  # 将前景色设为黑, 背景色设为绿.
318  echo -ne '\E[30;42m'
319
320  # 将光标移动到马的新位置.
321  tput cup `expr $MOVE_HORSE + 5` 'cat \
322  horse_${MOVE_HORSE}_position | head -1'
323
324  # 画马.
325  $DRAW_HORSE
326  usleep $USLEEP_ARG
327
328  # 当所有的马都越过第15行之后, 再次打印几率.
329  touch fieldline15
330  if [ $COL = 15 ]; then
331      echo $MOVE_HORSE >> fieldline15
332  fi
333  if [ `wc -l fieldline15 | cut -f1 -d " " = 9` ]; then
334      print_odds
335      : > fieldline15
336  fi
337
338  # 取得领头的马.
339  HIGHEST_POS='cat *position | sort -n | tail -1'

```

```
340
341      # 将背景色设为白.
342      echo -ne '\E[47m'
343      tput cup 17 0
344      echo -n Current leader: `grep -w $HIGHEST_POS *position | cut -c7`"
345
346 done
347
348 # 定义比赛结束的时间.
349 FINISH_TIME='date +%s'
350
351 # 将背景色设置为绿色, 并且开启闪烁文本的功能.
352 echo -ne '\E[30;42m'
353 echo -en '\E[5m'
354
355 # 让获胜的马闪烁.
356 tput cup `expr $MOVE_HORSE + 5` `cat horse_${MOVE_HORSE}_position | head -1`$DRAW_HORSE
357
358
359 # 禁用闪烁文本.
360 echo -en '\E[25m'
361
362 # 将前景色和背景色设置为白色.
363 echo -ne '\E[37;47m'
364 move_and_echo 18 1 $BLANK80
365
366 # 将前景色设置为黑色.
367 echo -ne '\E[30m'
368
369 # 让获胜的马闪烁.
370 tput cup 17 0
371 echo -e "\E[5mWINNER: ${MOVE_HORSE}\E[25m"" Odds: `cat odds_${MOVE_HORSE}`"\"
372 " Race time: `expr $FINISH_TIME - $START_TIME` secs"
373
374 # 恢复光标, 恢复原来的颜色.
375 echo -en "\E[?25h"
376 echo -en "\E[0m"
377
378 # 恢复打印功能.
379 stty echo
380
381 # 删除掉和赛马有关的临时文件.
382 rm -rf $HORSE_RACE_TMP_DIR
383
384 tput cup 19 0
```

```
385
386 exit 0
```

也请参考[例子A-22](#).

然而, 这里有一个严重的问题. ANSI转义序列是不可移植的. 在某些终端(或控制台)上运行的好好的代码, 可能在其他终端上根本没办法运行. ”彩色”的脚本可能会在脚本作者的机器上运行的非常好, 但是在其他人的机器上就可能产生不可读的输出. 因为这个原因, 使得”彩色”脚本的用途大打折扣, 而且很有可能使得这项技术变成华而不实的小花招, 甚至成为一个”玩具”.

Moshe Jacobson的彩色工具(<http://runslinux.net/projects.html#color>) 能够非常容易的简化ANSI转义序列的使用. 这个工具使用清晰而且富有逻辑的语法代替了之前讨论的难用的结构.

Henry/teikedvl也开发了一个类似的工具(<http://scriptechocolor.sourceforge.net/>) 用来简化彩色脚本的创建.

注意事项

1. 当然, ANSI是美国国家标准组织(American National Standards Institute)的缩写. 这个令人敬畏的组织建立和维护着许多技术和工业的标准.

6 优化

大部分shell脚本在处理不太复杂的问题的时候, 使用的都是小吃店(快速但是并不优雅)的方式. 正因为这样, 所以优化脚本的速度并不是一个大问题. 考虑一下这种情况, 当脚本正在处理一个重要任务的时候, 虽然这个脚本能够处理的很好, 但是它运行的速度实在太慢. 在这种情况下, 使用编译语言重写它其实也不是一种很合适的方法. 最简单的办法其实就是重写这个脚本执行效率低下的部分. 那么, 是否这种办法可以成为处理效率低下的shell脚本的一种原则?

仔细检查脚本中循环的部分. 因为重复的操作非常耗时. 如果有可能的话, 尽量删除掉循环中比较耗时的操作.

优先使用[内建命令](#), 而不是系统命令. 这是因为内建命令执行得更快, 并且在调用时, 一般都不会产生子进程.

避免使用不必要的命令, 尤其是[管道](#)中的命令.

```
1 cat "$file" | grep "$word"  
2  
3 grep "$word" "$file"  
4  
5 # 上面的两行具有相同的效果,  
6 #+ 但是第二行运行的更快, 因为它不产生子进程.
```

[cat](#)命令看起来经常在脚本中被滥用.

使用[time](#)和[times](#)工具来了解计算所消耗的时间. 可以考虑使用C语言, 甚至是汇编语言来重写时间消耗比较大的代码部分.

尝试尽量减少文件I/O的操作. 因为Bash在处理文件方面, 显得并不是很有效率, 所以可以在脚本中考虑使用更合适的工具, 比如[awk](#)或[Perl](#).

使用结构化的思想来编写脚本, 并且按照需求将各个模块组织并紧密结合起来. 一些适用于高级语言的优化技术也可以用在脚本上, 但是有些技术, 比如, 循环展开优化(loop unrolling), 就根本用不上. 关于上面的讨论, 可以根据经验来取舍.

怎么才能很好的减少脚本的执行时间, 让我们看一个优秀的例子, [例子12-42](#).

7 各种小技巧

- 为了记录在某个(或某些)特定会话中用户脚本的运行状态, 可以将下面的代码添加到你想跟踪记录的脚本中. 添加的这段代码会将脚本名和调用次数记录到一个连续的文件中.

```
1 # 添加(>>)下面的代码, 到你想跟踪记录的脚本末尾.  
2  
3 whoami>> $SAVE_FILE      # 记录调用脚本的用户.  
4 echo $0>> $SAVE_FILE      # 脚本名.  
5 date>> $SAVE_FILE        # 记录日期和时间.  
6 echo>> $SAVE_FILE        # 空行作为分隔符.  
7  
8 # 当然, 我们应该在~/.bashrc中定义并导出变量$SAVE_FILE.  
9 #+ (看起来有点像~/.scripts-run)
```

- >>操作符可以在文件末尾添加内容. 如果你想在文件的头部添加内容怎么办, 难道要粘贴到文件头?

```
1 file=data.txt  
2 title="***This is the title line of data text file***"  
3  
4 echo $title | cat - $file >$file.new  
5 # "cat -" 将stdout连接到$file.  
6 # 最后的结果就是生成了一新文件,  
7 #+ 并且成功的将$title的内容附加到了文件的*开头*.
```

这是之前的[例子17-13](#)脚本的简化版本. 当然, [sed](#)也能做到.

- shell脚本也可以象一个内嵌到脚本的命令那样被调用, 比如Tcl或wish脚本, 甚至是[Makefile](#). 在C语言中, 它们可以作为一个外部的shell命令被system()函数调用, 比如, system("script_name");.
- 将一个内嵌sed或awk的脚本内容赋值给一个变量, 能够提高shell包装脚本的可读性. 请参考[例子A-1](#)和[例子11-19](#).
- 将你最喜欢的变量定义和函数实现都放到一个文件中. 在你需要的时候, 通过使用点(.)命令, 或者source命令, 来将这些”库文件”包含”到脚本中.

```
1 # 脚本库  
2 # -----  
3  
4 # 注:  
5 # 这里没有"#!".  
6 # 也没有"真正需要执行的代码".  
7  
8  
9 # 有用的变量定义  
10
```

```

11 ROOT_UID=0          # root用户的$UID为0.
12 E_NOTROOT=101       # 非root用户的出错代码.
13 MAXRETVAL=255       # 函数最大的返回值(正值).
14 SUCCESS=0
15 FAILURE=-1
16
17
18
19 # Functions
20
21 Usage ()           # "Usage:"信息. (译者注: 即帮助信息)
22 {
23     if [ -z "$1" ]    # 没有参数传递进来.
24     then
25         msg=$filename
26     else
27         msg=$@          # $@表示所有命令行参数
28     fi
29
30     echo "Usage: `basename $0` \"$msg\""
31 }
32
33
34 Check_if_root ()   # 检查运行脚本的用户是否为root.
35 {
36     if [ "$UID" -ne "$ROOT_UID" ]
37     then
38         echo "Must be root to run this script."
39         exit $E_NOTROOT
40     fi
41 }
42
43
44 CreateTempfileName () # 创建"唯一"的临时文件.
45 {
46     prefix=temp
47     suffix='eval date +%s'
48     Tempfilename=$prefix.$suffix
49 }
50
51
52 isalpha2 ()          # 测试*整个字符串*是否都是由字母组成的.
53 {
54     # 摘自"isalpha.sh".
55     [ $# -eq 1 ] || return $FAILURE

```

```

56 case $1 in
57 *[!a-zA-Z]*|"") return $FAILURE;;
58 *) return $SUCCESS;;
59 esac                                # 感谢, S.C.
60 }
61
62
63 abs ()                               # 绝对值.
64 {
65     E_ARGERR=-999999
66
67     if [ -z "$1" ]                      # 需要传递参数.
68     then
69         return $E_ARGERR               # 返回错误.
70     fi
71
72     if [ "$1" -ge 0 ]                  # 如果是非负值,
73     then
74         absval=$1                     # 那就是绝对值本身.
75     else
76         let "absval = (( 0 - $1 ))"  # 改变符号.
77     fi
78
79     return $absval
80 }
81
82
83 tolower ()                            # 将传递进来的参数字符串
84 {                                     #+ 转换为小写.
85
86     if [ -z "$1" ]                      # 如果没有参数传递进来.
87     then
88         echo "(null)"                  #+ (C风格的void指针错误消息)
89         return                         #+ 并且从函数中返回.
90     fi
91
92     echo "$@" | tr A-Z a-z
93     # 转换所有传递进来的参数($@).
94
95     return
96
97 # 使用命令替换, 将函数的输出赋值给变量.
98 # 举例:
99 #     oldvar="A seT of miXed-caSe LEtTerS"
100 #     newvar='tolower "$oldvar"'

```

```
101 #     echo "$newvar"      # 一串混合大小写的字符全部转换为小写
102 #
103 # 练习：重写这个函数，
104 #           将传递进来的参数全部转换为大写[容易].
105 }
```

- 使用特殊目的注释头来增加脚本的条理性和可读性.

```
1 ## 表示注意.
2 rm -rf *.zzy    ## "rm"命令的"-rf"选项非常的危险.
3                         ##+ 尤其对通配符，就更危险.
4
5 #+ 表示继续上一行.
6 # 这是多行注释的第一行,
7 #+
8 #+ 这是最后一行.
9
10 ##* 表示标注.
11
12 #o 表示列表项.
13
14 #> 表示另一种观点.
15 while [ "$var1" != "end" ]      #> while test "$var1" != "end"
```

- if-test结构有一种聪明的用法, 用来注释代码块.

```
1#!/bin/bash
2
3COMMENT_BLOCK=
4# 如果给上面的变量赋值,
5#+ 就会出现令人不快的结果.
6
7if [ $COMMENT_BLOCK ]; then
8
9Comment block --
10=====
11This is a comment line.
12This is another comment line.
13This is yet another comment line.
14=====
15
16echo "This will not echo."
17
18Comment blocks are error-free! Whee!
19
20fi
```

```
21
22 echo "No more comments, please."
23
24 exit 0
```

比较这种用法, 和使用[here document注释代码块](#)之间的区别.

- 使用 \$? 退出状态变量, 因为脚本可能需要测试一个参数是否都是数字, 以便于后边可以把它当作一个整数来处理.

```
1 #!/bin/bash
2
3 SUCCESS=0
4 E_BADINPUT=65
5
6 test "$1" -ne 0 -o "$1" -eq 0 2>/dev/null
7 # 整数要不就是0, 要不就是非0值. (译者注: 感觉像废话 . . .)
8 # 2>/dev/null禁止输出错误信息.
9
10 if [ $? -ne "$SUCCESS" ]
11 then
12     echo "Usage: `basename $0` integer-input"
13     exit $E_BADINPUT
14 fi
15
16 let "sum = $1 + 25"          # 如果$1不是整数, 就会产生错误.
17 echo "Sum = $sum"
18
19 # 任何变量都可以使用这种方法来测试, 而不仅仅适用于命令行参数.
20
21 exit 0
```

- 函数的返回值严格限制在0 - 255之间. 使用全局变量或者其他方法来代替函数返回值, 通常都很容易产生问题. 从函数中, 返回一个值到脚本主体的另一个办法是, 将这个"返回值"写入到stdout(通常都使用echo命令), 然后将其赋值给一个变量. 这种做法其实就是命令替换的一个变种.

例子33-15. 返回值小技巧

```
1 #!/bin/bash
2 # multiplication.sh
3
4 multiply ()           # 将乘数作为参数传递进来.
5 {                      # 可以接受多个参数.
6
7     local product=1
8
9     until [ -z "$1" ]      # 直到处理完所有的参数...
```

```

10 do
11     let "product *= $1"
12     shift
13 done
14
15 echo $product          # 不会echo到stdout,
16 }                      #+ 因为要把它赋值给一个变量.
17
18 mult1=15383; mult2=25211
19 val1='multiply $mult1 $mult2'
20 echo "$mult1 X $mult2 = $val1"
21                         # 387820813
22
23 mult1=25; mult2=5; mult3=20
24 val2='multiply $mult1 $mult2 $mult3'
25 echo "$mult1 X $mult2 X $mult3 = $val2"
26                         # 2500
27
28 mult1=188; mult2=37; mult3=25; mult4=47
29 val3='multiply $mult1 $mult2 $mult3 $mult4'
30 echo "$mult1 X $mult2 X $mult3 X $mult4 = $val3"
31                         # 8173300
32
33 exit 0

```

相同的技术也可以用在字符串上. 这意味着函数可以”返回”非数字的值.

```

1 capitalize_ichar ()          # 将传递进来的字符串的
2 {                            #+ 首字母转换为大写.
3
4     string0="$@"
5                           # 能够接受多个参数.
6
7     firstchar=${string0:0:1}  # 首字母.
8     string1=${string0:1}      # 余下的字符.
9
10    FirstChar='echo "$firstchar" | tr a-z A-Z'
11        # 将首字母转换为大写.
12
13    echo "$FirstChar$string1" # 输出到stdout.
14
15}
16 newstring='capitalize_ichar "every sentence should start with \
17 a capital letter."'
18 echo "$newstring"           # Every sentence should start with a capital letter.

```

使用这种办法甚至能够“返回”多个值.

例子33-16. 返回多个值的技巧

```
1 #!/bin/bash
2 # sum-product.sh
3 # 可以“返回”超过一个值的函数.
4
5 sum_and_product ()  # 计算所有传递进来的参数的总和, 与总乘积.
6 {
7     echo $(( $1 + $2 )) $(( $1 * $2 ))
8 # 将每个计算出来的结果输出到stdout, 并以空格分隔.
9 }
10
11 echo
12 echo "Enter first number "
13 read first
14
15 echo
16 echo "Enter second number "
17 read second
18 echo
19
20 retval='sum_and_product $first $second'      # 将函数的输出赋值给变量.
21 sum='echo "$retval" | awk '{print $1}''      # 赋值第一个域.
22 product='echo "$retval" | awk '{print $2}''   # 赋值第二个域.
23
24 echo "$first + $second = $sum"
25 echo "$first * $second = $product"
26 echo
27
28 exit 0
```

- 下一个技巧, 是将数组传递给函数的技术, 然后“返回”一个数组给脚本的主体.

使用命令替换将数组中的所有元素(元素之间用空格分隔)赋值给一个变量, 这样就可以将数组传递到函数中了. 我们之前提到过一种返回值的策略, 就是将要从函数中返回的内容, 用echo命令输出出来, 然后使用命令替换或者(...)操作符, 将函数的输出(也就是我们想要得返回值)保存到一个变量中. 如果我们想让函数“返回”数组, 当然也可以使用这种策略.

例子33-17. 传递数组到函数, 从函数中返回数组

```
1 #!/bin/bash
2 # array-function.sh: 将数组传递到函数中与...
3 #                   从函数中“返回”一个数组
4
5
6 Pass_Array ()
```

```

7 {
8     local passed_array    # 局部变量 .
9     passed_array=( `echo "$1"` )
10    echo "${passed_array[@]}"
11    # 列出这个新数组中的所有元素 ,
12    #+ 这个新数组是在函数内声明的, 也是在函数内赋值的 .
13 }
14
15
16 original_array=( element1 element2 element3 element4 element5 )
17
18 echo
19 echo "original_array = ${original_array[@]}"
20 #           列出原始数组的所有元素 .
21
22
23 # 下面是关于如何将数组传递给函数的技巧 .
24 # ****
25 argument='echo ${original_array[@]}'
26 # ****
27 # 将原始数组中所有的元素都用空格进行分隔 ,
28 #+ 然后合并成一个字符串, 最后赋值给一个变量 .
29 #
30 # 注意, 如果只把数组传递给函数, 那是不行的 .
31
32
33 # 下面是让数组作为"返回值"的技巧 .
34 # ****
35 returned_array=( `Pass_Array "$argument"` )
36 # ****
37 # 将函数中'echo'出来的输出赋值给数组变量 .
38
39 echo "returned_array = ${returned_array[@]}"
40
41 echo "====="
42
43 # 现在, 再试一次 ,
44 #+ 尝试一下, 在函数外面访问(列出)数组 .
45 Pass_Array "$argument"
46
47 # 函数自身可以列出数组, 但是...
48 #+ 从函数外部访问数组是被禁止的 .
49 echo "Passed array (within function) = ${passed_array[@]}"
50 # NULL值, 因为这个变量是函数内部的局部变量 .
51

```

```
52 echo  
53  
54 exit 0
```

- 如果想更加了解如何将数组传递到函数中, 请参考[例子A-10](#), 这是一个精心制作的例子.
- 利用双括号结构, 就可以让我们使用C风格的语法, 在`for`循环和`while`循环中, 设置或者增加变量. 请参考[例子10-12](#)和[例子10-17](#).
- 如果在脚本的开头设置`path`和`umask`的话, 就可以增加脚本的“可移植性” – 即使在那些被用户将`$PATH`和`umask`弄糟了的机器上, 也可以运行.

```
1 #!/bin/bash  
2 PATH=/bin:/usr/bin:/usr/local/bin ; export PATH  
3 umask 022    # 脚本创建的文件所具有的权限是755.  
4  
5 # 感谢Ian D. Allen提出这个技巧.
```

- 一项很有用的技术是, 重复地将一个过滤器的输出(通过管道)传递给这个相同的过滤器, 但是这两次使用不同的参数和选项. 尤其是`tr`和`grep`, 非常适合于这种情况.

```
1 # 摘自例子"wstrings.sh".  
2  
3 wlist='strings "$1" | tr A-Z a-z | tr '[[:space:]]' Z | \  
4 tr -cs '[[:alpha:]]' Z | tr -s '\173-\377' Z | tr Z , ,'
```

例子33-18. anagram游戏

```
1 #!/bin/bash  
2 # agram.sh: 使用anagram来玩游戏.  
3  
4 # 寻找anagram...  
5 LETTERSET=etaoinshrdlu  
6 FILTER='.....'          # 最少有多少个字母?  
7 #      1234567  
8  
9 anagram "$LETTERSET" | # 找出这个字符串中所有的anagram...  
10 grep "$FILTER" |        # 至少需要7个字符,  
11 grep '^is' |            # 以'is'开头  
12 grep -v 's$' |          # 不是复数(指英文单词的复数)  
13 grep -v 'ed$'           # 不是过去时(也指英文单词)  
14 # 可以添加许多种组合条件和过滤器.  
15  
16 # 使用"anagram"工具,  
17 #+ 这是作者的"yaw1"文字表软件包中的一部分.  
18 #  http://ibiblio.org/pub/Linux/libs/yaw1-0.3.2.tar.gz  
19 #  http://personal.riverusers.com/~thegrendel/yaw1-0.3.2.tar.gz  
20
```

```

21 exit 0                      # 代码结束.
22
23
24 bash$ sh agram.sh
25 islander
26 isolate
27 isolead
28 isothermal
29
30
31
32 # 练习:
33 #
34 # 修改这个脚本, 使其能够让LETTERSET作为命令行参数.
35 # 将第11 - 13行的过滤器参数化(比如, 可以使用变量$filter),
36 #+ 这样我们就可以根据传递的参数来指定功能.
37
38 # 可以参考脚本agram2.sh,
39 #+ 与这个例子稍微有些不同.

```

也请参考[例子27-3](#), [例子12-22](#), 和[例子A-9](#).

- 使用”匿名的here document”来注释代码块, 这样就不用在每个注释行前面都加上#了. 请参考[例子17-11](#).
- 如果一个脚本的运行依赖于某个命令, 而且这个命令没被安装到运行这个脚本的机器上, 那么在运行的时候就会产生错误. 我们可以使用`whatis`命令来避免这种可能产生的问题.

```

1 CMD=command1                  # 第一选择.
2 PlanB=command2                # 如果第一选择不存在就选用这个.
3
4 command_test=$(whatis "$CMD" | grep 'nothing appropriate')
5 # 如果在系统中没找到'command1',
6 #+ 那么'whatis'将返回"command1: nothing appropriate."
7 #
8 # 另一种更安全的做法是:
9 #     command_test=$(whereis "$CMD" | grep \/)
10 # 但是下面的测试条件应该反过来,
11 #+ 因为变量$command_test只有在$CMD存在于系统上的时候,
12 #+ 才会有内容.
13 #     (感谢, bojster.)
14
15
16 if [[ -z "$command_test" ]]  # 检查命令是否存在.
17 then
18     $CMD option1 option2      # 使用选项来调用command1.
19 else
20     # 否则,

```

```
20 $PlanB          #+# 运行command2.  
21 fi
```

- 在错误的情况下, `if-grep test` 可能不会返回期望的结果, 因为出错文本是输出到stderr上, 而不是stdout.

```
1 if ls -l nonexistent_filename | grep -q 'No such file or directory'  
2   then echo "File \"nonexistent_filename\" does not exist."  
3 fi
```

将stderr重定向到stdout上, 就可以解决这个问题.

```
1 if ls -l nonexistent_filename 2>&1 | grep -q 'No such file or directory'  
2 #  
3   then echo "File \"nonexistent_filename\" does not exist."  
4 fi  
5  
6 # 感谢, Chris Martin指出这一点.
```

将stderr重定向到stdout上, 就可以解决这个问题.

```
1 if ls -l nonexistent_filename 2>&1 | grep -q 'No such file or directory'  
2 #  
3   then echo "File \"nonexistent_filename\" does not exist."  
4 fi  
5  
6 # 感谢, Chris Martin指出这一点.
```

- `run-parts`命令可以很方便的依次运行一组命令脚本, 尤其是和`cron`或`at`组合使用的时候.
 - 如果可以在shell脚本中调用X-Windows的小工具, 那该有多好. 目前已经有一些工具包可以完成这种功能, 比如Xscript, Xmenu, 和widtools. 头两种工具包已经不再被维护了. 幸运的是, 我们还可以从[这里](#) 下载第三种工具包, widtools.
- : 要想使用widtools(widget tools)工具包, 必须先安装XForms库. 除此之外, 在典型的Linux系统上编译之前, 需要正确的编辑它的`Makefile`. 最后, 在提供的6个部件中, 有3个不能工作(事实上, 会产生段错误).

dialog工具集提供了一种从shell脚本中调用“对话框”窗口部件的方法. The 原始的dialog工具包只能工作在文本的控制台模式下, 但是后续的类似工具, 比如gdialog, Xdialog, 和kdialog都是基于X-Windows窗口部件集合的.

例子33-19. 从shell脚本中调用窗口部件

```
1#!/bin/bash  
2# dialog.sh: 使用'gdialog'窗口部件.  
3# 必须在你的系统上安装'gdialog', 才能运行这个脚本.  
4# 版本1.1 (04/05/05最后修正)  
5  
6# 这个脚本的灵感来源于下面的文章.
```

```
7 #      "Scripting for X Productivity," by Marco Fioretti,
8 #      LINUX JOURNAL, Issue 113, September 2003, pp. 86-9.
9 # 感谢你们，所有的LINUX JOURNAL好人.
10
11
12 # 在对话框窗口中的输入错误.
13 E_INPUT=65
14 # 输入窗口的显示尺寸.
15 HEIGHT=50
16 WIDTH=60
17
18 # 输出文件名(由脚本名构造).
19 OUTFILE=$0.output
20
21 # 将脚本的内容显示到文本窗口中.
22 gdialog --title "Displaying: $0" --textbox $0 $HEIGHT $WIDTH
23
24
25
26 # 现在，我们将输入保存到文件中.
27 echo -n "VARIABLE=" > $OUTFILE
28 gdialog --title "User Input" --inputbox "Enter variable, please:" \
29 $HEIGHT $WIDTH 2>> $OUTFILE
30
31
32 if [ "$?" -eq 0 ]
33 # 检查退出状态码，是一个好习惯.
34 then
35   echo "Executed \"dialog box\" without errors."
36 else
37   echo "Error(s) in \"dialog box\" execution."
38   # 或者，点"Cancel"按钮，而不是"OK".
39   rm $OUTFILE
40   exit $E_INPUT
41 fi
42
43
44
45 # 现在，我们将重新获得并显示保存的变量.
46 . $OUTFILE # 'Source'(执行)保存的文件.
47 echo "The variable input in the \"input box\" was: \"$VARIABLE\""
48
49
50 rm $OUTFILE # 清除临时文件.
51           # 某些应用可能需要保留这个文件.
```

```
52
53 exit $?
```

其他在脚本中使用窗口部件的工具, 比如Tk或wish (Tcl派生物), PerlTk(带有Tk扩展的Perl), tksh(带有Tk扩展的ksh), XForms4Perl(带有XForms扩展的Perl), Gtk-Perl(带有Gtk扩展的Perl), 或PyQt(带有Qt扩展的Python).

- 为了对复杂脚本做多次的修正, 可以使用rcs修订控制系统包.

使用这个软件包的好处之一就是可以自动升级ID头标志. rcs包中的co命令可以对特定的保留关键字作参数替换, 比如, 可以使用下面这行代码来替换掉脚本中的#\$\$Id\$,

```
1 $$Id: hello-world.sh,v 1.1 2004/10/16 02:43:05 bozo Exp $
```

8 安全问题

33.8.1. 被感染的脚本

在这里对脚本安全进行一个简短的介绍非常合适。shell脚本可能会包含蠕虫，特洛伊木马，甚至可能会中病毒。由于这些原因，永远不要用root身份来运行脚本（或者将自己不太清楚的脚本插入到/etc/rc.d里面的系统启动脚本中），除非你确定这是值得信赖的源代码，或者你已经小心的分析了这个脚本，并确定它不会产生什么危害。

Bell实验室以及其他地方的病毒研究人员，包括M. Douglas McIlroy, Tom Duff, 和Fred Cohen已经研究过了shell脚本病毒的实现。他们认为即使是初学者也可以很容易的编写脚本病毒，比如“脚本小子(script kiddie)”，就写了一个。[\[1\]](#)

这也是学习脚本编程的另一个原因。能够很好地了解脚本，就可以让的系统免受怪客的攻击和破坏。

33.8.2. 隐藏Shell脚本源代码

出于安全目的，让脚本不可读，也是有必要的。如果有软件可以将脚本转化为相应的二进制可执行文件就好了。Francisco Rosales的[shc - generic shell script compiler](#)可以出色的完成这个任务。

不幸的是，根据[发表在2005年10月的Linux Journal](#)上的一篇文章，二进制文件，至少在某些情况下，可以被恢复成原始的脚本代码。但是不管怎么说，对于那些技术不高的怪客来说，这仍然是一种保证脚本安全的有效办法。

注意事项

1. 请参考Marius van Oers的文章，[Unix Shell Scripting Malware](#)，还有列在[参考书目](#)中的Denning的书目。

9 可移植性问题

这本书主要描述的是，在GNU/Linux系统上，如何处理特定于Bash的脚本。但是使用sh和ksh的用户仍然会从这里找到很多有价值的东西。

碰巧，许多不同的shell脚本语言其实都遵循POSIX 1003.2标准。如果使用–posix选项来调用Bash，或者在脚本头插入set -o posix，那么将会使Bash与这个标准非常接近地保持一致。另一种办法就是在脚本头使用

```
1 #!/bin/sh
```

而不是

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

注意在Linux或者某些特定的UNIX上，/bin/sh其实只是一个指向/bin/bash的[链接](#)，并且使用这种方法调用脚本的话，将会禁用Bash的扩展功能。

大多数的Bash脚本都好像运行在ksh上一样，反过来，这是因为Chet Ramey一直致力于将ksh的特性移植到Bash的最新版本上。

对于商业UNIX机器来说，如果在脚本中包含了使用GNU特性的标准命令，那么这些脚本可能不会正常运行。但是在最近几年，这个问题得到了极大的改观，这是因为即使在“大块头”UNIX上，GNU工具包也非常好的替换掉了同类的私有工具。对于原始的UNIX来说，[源代码的火山喷发](#) 加剧了这种趋势。

Bash具有的某些特性是传统的Bourne shell所缺乏的。下面就是其中的一部分：

- 某些扩展的[调用选项](#)
- 使用\$()形式的[命令替换](#)
- 某些[字符串处理](#)操作符
- [进程替换](#)
- Bash特有的[内建命令](#)

请参考[Bash F.A.Q.](#)，你将会获得完整的列表。

10 Windows下的shell脚本

即使用户使用其他的操作系统来运行类UNIX的shell脚本，其实也能够从本书的大部分课程中受益。Cygwin公司的Cygwin程序包和Mortice Kern Associates的MKS工具集都能够给Windows添加处理shell脚本的能力。

这其实暗示了Windows将来的版本可能会包含处理类Bash命令行脚本的能力，不过，还是让我们拭目以待吧。

第34章 杂项

本章目录

1. [Bash, 版本2](#)
 2. [Bash, 版本3](#)
-

1 Bash, 版本2

当前比较流行的Bash版本有两个, 版本2.xx.y或版本3.xx.y, 这两个中的某一个估计就运行在你的机器上.

```
1 bash$ echo $BASH_VERSION  
2 2.05.b.0(1)-release
```

经典Bash脚本语言版本2的主要升级内容, 增加了数组变量, [1] 字符串和参数扩展, 还添加了间接变量引用的一种更好的方法, 以及其他特性.

例子34-1. 字符串扩展

```
1#!/bin/bash  
2  
3# 字符串扩展.  
4# Bash版本2中引入的特性.  
5  
6# '$'xxx'格式的字符串  
7#+ 具备解释里面标准转义字符的能力.  
8  
9echo $'Ringing bell 3 times \a \a \a'  
10    # 可能在某些终端中, 只会响一次铃.  
11echo $'Three form feeds \f \f \f'  
12echo $'10 newlines \n\n\n\n\n\n\n\n\n\n'  
13echo $'\102\141\163\150'    # Bash  
14                                # 8进制的等价字符.  
15  
16exit 0
```

例子34-2. 间接变量引用- 新方法

```
1#!/bin/bash  
2  
3# 间接变量引用.  
4# 这种方法比较像C++中的引用特性.  
5  
6  
7a=letter_of_alphabet  
8letter_of_alphabet=z  
9  
10echo "a = $a"          # 直接引用.  
11  
12echo "Now a = ${!a}"    # 间接引用.  
13# ${!variable}表示法比老式的"eval var1=\$\$var2"表示法高级的多.  
14  
15echo  
16
```

```

17 t=table_cell_3
18 table_cell_3=24
19 echo "t = ${!t}"                      # t = 24
20 table_cell_3=387
21 echo "Value of t changed to ${!t}"    # 387
22
23 # 在引用数组成员或者引用表的时候，这种方法非常有用，
24 #+ 还可以用来模拟多维数组。
25 # 如果有能够索引的选项(类似于指针的算术运算)
26 #+ 就更好了。可惜。
27
28 exit 0

```

例子34-3. 使用间接变量引用的简单数据库应用

```

1#!/bin/bash
2# resistor-inventory.sh
3# 使用间接变量引用的简单数据库应用。
4
5# ===== #
6# 数据
7
8B1723_value=470                      # 欧姆
9B1723_powerdissip=.25                  # 瓦特
10B1723_colorcode="yellow-violet-brown" # 颜色
11B1723_loc=173                         # 位置
12B1723_inventory=78                    # 数量
13
14B1724_value=1000
15B1724_powerdissip=.25
16B1724_colorcode="brown-black-red"
17B1724_loc=24N
18B1724_inventory=243
19
20B1725_value=10000
21B1725_powerdissip=.25
22B1725_colorcode="brown-black-orange"
23B1725_loc=24N
24B1725_inventory=89
25
26# ===== #
27
28
29echo
30
31PS3='Enter catalog number: '

```

```

32
33 echo
34
35 select catalog_number in "B1723" "B1724" "B1725"
36 do
37     Inv=${catalog_number}_inventory
38     Val=${catalog_number}_value
39     Pdissip=${catalog_number}_powerdissip
40     Loc=${catalog_number}_loc
41     Ccode=${catalog_number}_colorcode
42
43 echo
44 echo "Catalog number $catalog_number:"
45 echo "There are ${!Inv} of [${!Val} ohm / ${!Pdissip} watt] resistors in stock."
46 echo "These are located in bin # ${!Loc}."
47 echo "Their color code is \"${!Ccode}\"."
48
49 break
50 done
51
52 echo; echo
53
54 # 练习:
55 # -----
56 # 1) 重写脚本, 使其从外部文件读取数据.
57 # 2) 重写脚本,
58 #+ 用数组来代替间接变量引用,
59 # 因为使用数组更简单, 更易懂.
60
61
62 # 注:
63 # ---
64 # 除了最简单的数据库应用, 事实上, Shell脚本本身并不适合于数据库应用.
65 #+ 因为它太依赖于工作环境和机器的运算能力.
66 # 更好的办法还是使用支持数据结构的本地语言,
67 #+ 比如C++或者Java(或者甚至可以是Perl).
68
69 exit 0

```

例子34-4. 使用数组和其他的小技巧来处理4人随机打牌

```

1#!/bin/bash
2
3# 纸牌:
4# 处理4人打牌.
5

```

```
6 UNPICKED=0
7 PICKED=1
8
9 DUPE_CARD=99
10
11 LOWER_LIMIT=0
12 UPPER_LIMIT=51
13 CARDS_IN_SUIT=13
14 CARDS=52
15
16 declare -a Deck
17 declare -a Suits
18 declare -a Cards
19 # 使用一个3维数组来代替这3个一维数组来描述数据,
20 #+ 可以更容易实现, 而且可以增加可读性.
21 # 或许在Bash未来的版本上会支持多维数组.
22
23
24 initialize_Deck ()
25 {
26 i=$LOWER_LIMIT
27 until [ "$i" -gt $UPPER_LIMIT ]
28 do
29   Deck[i]=$UNPICKED    # 将整副"牌"的每一张都设置为无人持牌的状态.
30   let "i += 1"
31 done
32 echo
33 }
34
35 initialize_Suits ()
36 {
37 Suits[0]=C #梅花
38 Suits[1]=D #方块
39 Suits[2]=H #红心
40 Suits[3]=S #黑桃
41 }
42
43 initialize_Cards ()
44 {
45 Cards=(2 3 4 5 6 7 8 9 10 J Q K A)
46 # 另一种初始化数组的方法.
47 }
48
49 pick_a_card ()
50 {
```

```
51 card_number=$RANDOM
52 let "card_number %= $CARDS"
53 if [ "${Deck[$card_number]}" -eq $UNPICKED ]
54 then
55     Deck[$card_number]=$PICKED
56     return $card_number
57 else
58     return $DUPE_CARD
59 fi
60 }
61
62 parse_card ()
63 {
64 number=$1
65 let "suit_number = number / CARDS_IN_SUIT"
66 suit=${Suits[$suit_number]}
67 echo -n "$suit-"
68 let "card_no = number % CARDS_IN_SUIT"
69 Card=${Cards[$card_no]}
70 printf %-4s $Card
71 # 使用整洁的列形式来打印每张牌.
72 }
73
74 seed_random () # 种子随机数产生器.
75 { # 如果不这么做, 会发生什么?
76     seed='`date +%s`'
77     let "seed %= 32766"
78     RANDOM=$seed
79 # 还有其他的方法
80 #+ 能够产生种子随机数么?
81 }
82
83 deal_cards ()
84 {
85 echo
86
87 cards_picked=0
88 while [ "$cards_picked" -le $UPPER_LIMIT ]
89 do
90     pick_a_card
91     t=$?
92
93     if [ "$t" -ne $DUPE_CARD ]
94     then
95         parse_card $t
```

```
96
97     u=$cards_picked+1
98     # 将数组索引改为从1(译者注: 数组都是从0开始索引的)开始(临时的). 为什么?
99     let "u %= $CARDS_IN_SUIT"
100    if [ "$u" -eq 0 ]    # 内嵌的if/then条件测试.
101    then
102        echo
103        echo
104    fi
105    # 分手.
106
107    let "cards_picked += 1"
108    fi
109 done
110
111 echo
112
113 return 0
114 }
115
116
117 # 结构化编程:
118 # 将函数中的整个程序逻辑模块化.
119
120 =====
121 seed_random
122 initialize_Deck
123 initialize_Suits
124 initialize_Cards
125 deal_cards
126 =====
127
128 exit 0
129
130
131
132 # 练习1:
133 # 完整的注释这个脚本.
134
135 # 练习2:
136 # 添加一个例程(函数)按照花色打印出每手牌.
137 # 如果你喜欢, 可以添加任何你想要添加的代码.
138
139 # 练习3:
```

注意事项

1. Chet Ramey承诺会在Bash未来的版本中支持关联数组(一个Perl特性). 但是到了版本3, 他的承诺还没兑现.

2 Bash, 版本3

2004年7月27日, Chet Ramey发布了Bash版本3. 这一版本修复了相当多的bug, 并加入了一些新特性.

新增加的一些属性有:

- 一个新的, 更加通用的{a..z}大括号扩展操作符.

```
1 #!/bin/bash
2
3 for i in {1..10}
4 # 比下面的方式更简单, 更直接
5 #+ for i in $(seq 10)
6 do
7   echo -n "$i "
8 done
9
10 echo
11
12 # 1 2 3 4 5 6 7 8 9 10
```

- \$!array[@]操作符, 用于扩展给定数组所有元素索引.

```
1 #!/bin/bash
2
3 Array=(element-zero element-one element-two element-three)
4
5 echo ${Array[0]}      # 元素0
6                      # 数组的第一个元素.
7
8 echo ${!Array[@]}    # 0 1 2 3
9                      # 数组的全部索引.
10
11 for i in ${!Array[@]}
12 do
13   echo ${Array[i]} # 元素0
14           # 元素1
15           # 元素2
16           # 元素3
17           #
18           # 数组的全部元素.
19 done
```

- = 正则表达式匹配操作符, 在双中括号测试表达式中的应用. (Perl也有一个类似的操作符.)

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

```

2
3 variable="This is a fine mess."
4
5 echo "$variable"
6
7 if [[ "$variable" =~ "T*fin*es*" ]]
8 # 在[[ 双中括号 ]]中使用=~操作符进行正则匹配.
9 then
10   echo "match found"
11   # match found
12 fi

```

或者, 更有用的用法:

```

1 1#!/bin/bash
2
3 2
4 3 input=$1
5
6 4
7 5
8 6 if [[ "$input" =~ "[1-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9]" ]]
9 7 # NNN-NN-NNNN
10 8 # 每个N都是一个数字.
11 9 # 但是, 第一个数字不能为0.
12 10 then
13 11   echo "Social Security number."
14 12   # 处理SSN.
15 13 else
16 14   echo "Not a Social Security number!"
17 15   # 或者, 要求正确的输入.
18 16 fi

```

还有一个使用=操作符的例子, 请参考[例子A-29](#)和[例子17-14](#).

►: Bash 3.0版本的更新, 将会导致一小部分为早期Bash版本编写的脚本不能工作. 对于一些重要的早期脚本来说, 需要进行测试, 以保证它们在新版本的Bash中也可以正常工作!

如果发生确实不能正常工作的情况, 那么高级Bash脚本编程指南中的某些脚本就必须被修复(请参考[例子A-20](#)和[例子9-4](#)).

34.2.1. Bash, 版本3.1

Bash3.1版本的更新修复了一部分bug, 并且在其他方面也做了一些小的修改.

- +=操作符是新添加的, 可以放在之前只能有=赋值操作符出现的地方.

```

1 a=1
2 echo $a      # 1
3
4 a+=5      # 在Bash的早期版本中就不行, 只能运行在Bash3.1或更新的版本上.

```

```
5 echo $a      # 15
6
7 a+=Hello
8 echo $a      # 15Hello
```

在这里, `+=`是作为字符串连接操作符. 注意, 它在这种特定的上下文中所表现出来的行为, 与在`let`结构中所表现出来的行为是不同的.

```
1 a=1
2 echo $a      # 1
3
4 let a+=5      # 整数的算术运算, 而不是字符串连接.
5 echo $a      # 6
6
7 let a+=Hello  # 不会给a"添加"任何东西.
8 echo $a      # 6
```

第35章 后记

本章目录

1. [作者后记](#)
 2. [关于作者](#)
 3. [译者后记](#)
 4. [在哪里可以获得帮助](#)
 5. [用来制作这本书的工具](#)
 6. [致谢](#)
 7. [译者致谢](#)
-

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.

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, 只有一颗冲动的心, 用来写作. [1] 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.

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, 和软件包[yaw1](#)一起组成猜词表. 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.

注意事项

1. 这种事谁都可以做. 但是有些人不能...想要拿到MCSE证书.

3 译者后记

35.3.1. 杨春敏

为了学习Bash脚本知识, 我找到了本书, 并且做了详细的笔记.

因为是初学, 随着笔记的增加, 发现越来越像是翻译.

考虑到平时总在网上看其他人的免费资料, 有感而发, 才促成了本书的翻译版.

我从来没考虑过翻译书会是一件多么复杂和艰苦的事情.

但是随着翻译版的进行, 我体会了个中的酸甜苦辣, 可能我这辈子只会有这么一次翻译书的经历, 我甚至不敢说我翻译的有多好, 但是我尽力了.

希望为大家做些贡献, 退一步讲, 我自己也收获了许多.

想不到的是, 有许多朋友关心翻译版, 我在这里由衷的对你们表示感谢!

最后感谢原书作者写了这么一本好书, 才能给我这个翻译的机会.

35.3.2. 黄毅

我在www.linuxsir.org的SHELL版块任版主有些时候了, 一直没有太多心力去做些事情. 工作之后, 非常的忙碌, 为了生活的需要和自己职业的发展, 更加地力不从心, 但我想总要做些什么, 这就是翻译这本书最初的念头.

在我艰难, 断断续续的翻译着的时候, Linuxsir上, 杨春敏兄弟发了一个帖子, 他也在翻译这本书— 太好了, 我们很快就一起合作翻译, 是的, 伙计! 这比一个人傻乎乎地单打独斗强多了.

不幸的是, 我们之前的工作有一些重复, 说不幸, 那只是因为更不幸的事情还没有被发现, 更不幸的事情是, 我们是直接在HTML或是在一个文本编辑器里翻译. 这样我们无法获得可持续的升级, 并且让工作更加地艰难. 不过, 事情最终还是在大半年之后有了进展, 2006年的5月15号, 我们发布了一个beta版本, 是以文本格式发布的, 在5月30号发布了HTML版本. 随后, 翻译工作告一段落, 我甚至不愿意回忆这枯燥无味的日子, 而此时我的本职工作更加的繁忙, 我一心扑到工作中来, 杨春敏兄弟一力承担起后继的工作, 建起这本书的SGML的版本, 使可持续的维护成为可能, 较正并重新翻译了书中的全部内容, 真可谓劳苦功高, 做了大量的工作, 辛苦了!

到此, 我们认为可以发布一个正式版本, 希望大家能喜欢这个中文版.

向那些对beta版本提出修改意见的人们, 和支持此书的人们表示感谢!

特别感谢原书作者Mendel Cooper为我们贡献了这本好书!

4 在哪里可以获得帮助

如果作者不是太忙(并且心情还不错)的话, 可能会回答一些通用的脚本编程问题. [1] 但是, 如果你想询问关于你的特殊脚本的某些特定问题, 那么建议你最好将这些问题发送到[comp.os.unix.shell](#)新闻组中去.

如果你是在写作业的时候需要帮助, 那么你最好阅读这本书的相关章节, 并且查阅相关资料. 然后用你的智慧和找到的资源, 尽你最大的努力来解决这个问题. 不要浪费作者的时间, 没人同情你, 否则你再也不会得到相关的帮助.

注意事项

1. 那些来自于垃圾邮件大量滋生的顶级域名(61, 202, 211, 218, 220, 等等.)的email, 将会被垃圾邮件过滤器收集起来, 并且会删掉未读的信件. 如果你的ISP不巧就是上面中的某一个, 那么请使用Webmail账号来联系作者.

5 用来制作这本书的工具

35.5.1. 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?igc?

35.5.2. 软件与排版软件

Bram Moolenaar的强大的SGML软件, ??vim文本编辑器.

[OpenJade](#), 使用DSSSL翻译引擎, 来将SGML文档转换为其他格式的工具.

[Norman Walsh的DSSSL样式单](#).

DocBook, The Definitive Guide(译者注: 这本书被亲切称为TDG), 这本书由Norman Walsh和Leonard Muellner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.)

译者: 事实上, 译者所使用的软件环境与原书作者基本相同, 甚至编辑器都一样. 唯一一点不同就是译者使用的是cygwin下的openjade, 如果读者们有兴趣, 我总结了一个关于如何在cygwin下配置openjade环境的文档- 因为网上没找到, 所以自己花功夫研究了一下, 有机会就公布出来- 另外译者为了与原版的html版本保持格式相同, 已经在TLDL的DSSSL基础上做了最大努力的修改.

6 致谢

团体的力量才使得这本书顺利地完成。作者非常感激那些给作者提供帮助和反馈的人们，如果没有他们，这本书根本就是一个不可能完成的任务。

译者：如下是作者感谢的对象。为了保持原文作者的完整性，这里就不译了。

Philippe Martin translated the first version (0.1) of this document into DocBook/SGML. While not on the job at a small French company as a software developer, he enjoys working on GNU/Linux documentation and software, reading literature, playing music, and, for his peace of mind, making merry with friends. You may run across him somewhere in France or in the Basque Country, or you can email him at feloy@free.fr.

Philippe Martin also pointed out that positional parameters past \$9 are possible using bracket notation. (See [例子4-5](#)).

Stephane Chazelas sent a long list of corrections, additions, and [example scripts](#). More than a contributor, he had, in effect, for a while taken on the role of editor for this document. Merci beaucoup!

Paulo Marcel Coelho Aragao offered many corrections, both major and minor, and contributed quite a number of helpful suggestions.

I would like to especially thank Patrick Callahan, Mike Novak, and Pal Domokos for catching bugs, pointing out ambiguities, and for suggesting clarifications and changes. Their lively discussion of shell scripting and general documentation issues inspired me to try to make this document more readable.

I'm grateful to Jim Van Zandt for pointing out errors and omissions in version 0.2 of this document. He also contributed an instructive example script.

Many thanks to Jordi Sanfeliu for giving permission to use his fine tree script ([例子A-17](#)), and to Rick Boivie for revising it.

Likewise, thanks to Michel Charpentier for permission to use his dc factoring script ([例子12-47](#)).

Kudos to Noah Friedman for permission to use his string function script ([例子A-18](#)).

Emmanuel Rouat suggested corrections and additions on command substitution and aliases. He also contributed a very nice sample .bashrc file ([Appendix K](#)).

Heiner Steven kindly gave permission to use his base conversion script, [例子12-43](#). He also made a number of corrections and many helpful suggestions. Special thanks.

Rick Boivie contributed the delightfully recursive pb.sh script ([例子33-9](#)), revised the tree.sh script ([例子A-17](#)), and suggested performance improvements for the monthlypmt.sh script ([例子12-42](#)).

Florian Wisser enlightened me on some of the fine points of testing strings (see [例子7-6](#)), and on other matters.

Oleg Philon sent suggestions concerning cut and pidof.

Michael Zick extended the empty array example to demonstrate some surprising array properties. He also contributed the isspammer scripts ([例子12-37](#) and [例子A-28](#)).

Marc-Jano Knopp sent corrections and clarifications on DOS batch files.

Hyun Jin Cha found several typos in the document in the process of doing a Korean translation. Thanks for pointing these out.

Andreas Abraham sent in a long list of typographical errors and other corrections. Special thanks!

Others contributing scripts, making helpful suggestions, and pointing out errors were 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," Emilio Conti, Ian. D. Allen, Arun Giridhar, Dennis Leeuw, Dan Jacobson, Aurelio Marinho Jargas, Edward Scholtz, Jean Helou, Chris Martin, Lee Maschmeyer, Bruno Haible, Wilbert Berendsen, Sebastien Godard, Bj鰎n Eriksson, John MacDonald, Joshua Tschida, Troy Engel, Manfred Schwarb, Amit Singh, Bill Gradwohl, David Lombard, Jason Parker, Steve Parker, Bruce W. Clare, William Park, Vernia Damiano, Mihai Maties, Jeremy Impson, Ken Fuchs, Frank Wang, Sylvain Fourmanoit, Matthew Walker, Kenny Stauffer, Filip Moritz, Andrzej Stefanski, Daniel Albers, Stefano Palmeri, Nils Radtke, 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, Peter Knowles, Francisco Lobo, Mariusz Gniazdowski, Tedman Eng, Jochen DeSmet, Oliver Beckstein, Achmed Darwish, Andreas K黨ne, and David Lawyer (himself an author of four HOWTOs).

My gratitude to Chet Ramey and Brian Fox for writing Bash, and building into it elegant and powerful scripting capabilities.

Very special thanks to the hard-working volunteers at the Linux Documentation Project. The LDP hosts a repository of Linux knowledge and lore, and has, to a large extent, enabled the publication of this book.

Thanks and appreciation to IBM, Novell, Red Hat, the Free Software Foundation, and all the good people fighting the good fight to keep Open Source software free and open.

Thanks most of all to my wife, Anita, for her encouragement and emotional support.

7 译者致谢

这里放上所有对译本做过贡献同志们, 译者真心的感谢你们. (排名不分先后)

www.linuxsir.org站点提供了发布环境, 也是我们两个译者结识的地方.

宗耀堂 – 感谢对exit status与exit code翻译的建议.

phanrider – 感谢对于beta版本的错别字校正.

第五部分

参考文献

Those who do not understand UNIX are condemned to reinvent it, poorly.

Henry Spencer

Edited by Peter Denning, Computers Under Attack: Intruders, Worms, and Viruses, ACM Press, 1990, 0-201-53067-8.

This compendium contains a couple of articles on shell script viruses.

*

Ken Burch, [Linux Shell Scripting with Bash](#), 1st edition, Sams Publishing (Pearson), 2004, 0672326426.

Covers much of the same material as this guide. Dead tree media does have its advantages, though.

*

Dale Dougherty and Arnold Robbins, Sed and Awk, 2nd edition, O'Reilly and Associates, 1997, 1-56592-225-5.

To unfold the full power of shell scripting, you need at least a passing familiarity with sed and awk. This is the standard tutorial. It includes an excellent introduction to "regular expressions". Read this book.

*

Jeffrey Friedl, Mastering Regular Expressions, O'Reilly and Associates, 2002, 0-596-00289-0.

The best, all-around reference on [Regular Expressions](#).

*

Aleen Frisch, *Essential System Administration*, 3rd edition, O'Reilly and Associates, 2002, 0-596-00343-9.

This excellent sys admin manual has a decent introduction to shell scripting for sys administrators and does a nice job of explaining the startup and initialization scripts. The long overdue third edition of this classic has finally been released.

*

Stephen Kochan and Patrick Woods, *Unix Shell Programming*, Hayden, 1990, 067248448X.

The standard reference, though a bit dated by now.

*

Neil Matthew and Richard Stones, *Beginning Linux Programming*, Wrox Press, 1996, 1874416680.

Good in-depth coverage of various programming languages available for Linux, including a fairly strong chapter on shell scripting.

*

Herbert Mayer, *Advanced C Programming on the IBM PC*, Windcrest Books, 1989, 0830693637.

Excellent coverage of algorithms and general programming practices.

*

David Medinets, *Unix Shell Programming Tools*, McGraw-Hill, 1999, 0070397333.

Good info on shell scripting, with examples, and a short intro to Tcl and Perl.

*

Cameron Newham and Bill Rosenblatt, *Learning the Bash Shell*, 2nd edition, O'Reilly and Associates, 1998, 1-56592-347-2.

This is a valiant effort at a decent shell primer, but somewhat deficient in coverage on programming topics and lacking sufficient examples.

*

Anatole Olczak, *Bourne Shell Quick Reference Guide*, ASP, Inc., 1991, 093573922X.

A very handy pocket reference, despite lacking coverage of Bash-specific features.

*

Jerry Peek, Tim O'Reilly, and Mike Loukides, *Unix Power Tools*, 2nd edition, O'Reilly and Associates, Random House, 1997, 1-56592-260-3.

Contains a couple of sections of very informative in-depth articles on shell programming, but falls short of being a tutorial. It reproduces much of the regular expressions tutorial from the Dougherty and Robbins book, above.

*

Clifford Pickover, *Computers, Pattern, Chaos, and Beauty*, St. Martin's Press, 1990, 0-312-04123-3.

A treasure trove of ideas and recipes for computer-based exploration of mathematical oddities.

*

George Polya, *How To Solve It*, Princeton University Press, 1973, 0-691-02356-5.

The classic tutorial on problem solving methods (i.e., algorithms).

*

Chet Ramey and Brian Fox, *The GNU Bash Reference Manual*, Network Theory Ltd, 2003, 0-9541617-7-7.

This manual is the definitive reference for GNU Bash. The authors of this manual, Chet Ramey and Brian Fox, are the original developers of GNU Bash. For each copy sold the publisher donates \$1 to the Free Software Foundation.

Arnold Robbins, *Bash Reference Card*, SSC, 1998, 1-58731-010-5.

Excellent Bash pocket reference (don't leave home without it). A bargain at \$4.95, but also available for free download on-line in pdf format.

*

Arnold Robbins, *Effective Awk Programming*, Free Software Foundation / O'Reilly and Associates, 2000, 1-882114-26-4.

The absolute best awk tutorial and reference. The free electronic version of this book is part of the awk documentation, and printed copies of the latest version are available from O'Reilly and Associates.

This book has served as an inspiration for the author of this document.

*

Bill Rosenblatt, *Learning the Korn Shell*, O'Reilly and Associates, 1993, 1-56592-054-6.

This well-written book contains some excellent pointers on shell scripting.

*

Paul Sheer, *LINUX: Rute User's Tutorial and Exposition*, 1st edition, , 2002, 0-13-033351-4.

Very detailed and readable introduction to Linux system administration.

The book is available in print, or on-line.

*

Ellen Siever and the staff of O'Reilly and Associates, *Linux in a Nutshell*, 2nd edition, O'Reilly and Associates, 1999, 1-56592-585-8.

The all-around best Linux command reference, even has a Bash section.

*

Dave Taylor, *Wicked Cool Shell Scripts: 101 Scripts for Linux, Mac OS X, and Unix Systems*, 1st edition, No Starch Press, 2004, 1-59327-012-7.

Just as the title says . . .

*

The UNIX CD Bookshelf, 3rd edition, O'Reilly and Associates, 2003, 0-596-00392-7.

An array of seven UNIX books on CD ROM, including UNIX Power Tools, Sed and Awk, and Learning the Korn Shell. A complete set of all the UNIX references and tutorials you would ever need at about \$130. Buy this one, even if it means going into debt and not paying the rent.

*

The O'Reilly books on Perl. (Actually, any O'Reilly books.)

—

Fioretti, Marco, "Scripting for X Productivity," Linux Journal, Issue 113, September, 2003, pp. 86-9.

Ben Okopnik's well-written introductory Bash scripting articles in issues 53, 54, 55, 57, and 59 of the Linux Gazette, and his explanation of "The Deep, Dark Secrets of Bash" in issue 56.

Chet Ramey's bash - The GNU Shell, a two-part series published in issues 3 and 4 of the Linux Journal, July-August 1994.

Mike G's Bash-Programming-Intro HOWTO.

Richard's Unix Scripting Universe.

Chet Ramey's Bash F.A.Q.

Ed Schaefer's Shell Corner in Unix Review.

Example shell scripts at Lucc's Shell Scripts .

Example shell scripts at SHELLdorado .

Example shell scripts at Noah Friedman's script site.

Example shell scripts at zazzybob.

Steve Parker's Shell Programming Stuff.

Example shell scripts at SourceForge Snippet Library - shell scripts.

"Mini-scripts" at Unix Oneliners.

Giles Orr's Bash-Prompt HOWTO.

Very nice sed, awk, and regular expression tutorials at The UNIX Grymoire.

Eric Pement's sed resources page.

Many interesting sed scripts at the seder's grab bag.

The GNU gawk reference manual (gawk is the extended GNU version of awk available on Linux and BSD systems).

Tips and tricks at Linux Reviews.

Trent Fisher's groff tutorial.

Mark Komarinski's Printing-Usage HOWTO.

The Linux USB subsystem (helpful in writing scripts affecting USB peripherals).

There is some nice material on I/O redirection in chapter 10 of the textutils documentation at the University of Alberta site.

Rick Hohensee has written the osimpa i386 assembler entirely as Bash scripts.

Aurelio Marinho Jargas has written a Regular expression wizard. He has also written an informative book on Regular Expressions, in Portuguese.

Ben Tomkins has created the Bash Navigator directory management tool.

William Park has been working on a project to incorporate certain Awk and Python features into Bash. Among these is a gdbm interface. He has released bashdiff on Freshmeat.net. He has an article in the November, 2004 issue of the Linux Gazette on adding string functions to Bash, with a followup article in the December issue, and yet another in the January, 2005 issue.

Peter Knowles has written an elaborate Bash script that generates a book list on the Sony Librie e-book reader. This useful tool permits loading non-DRM user content on the Librie.

Rocky Bernstein is in the process of developing a "full-fledged" debugger for Bash.

Of historical interest are Colin Needham's original International Movie Database (IMDB) reader polling scripts, which nicely illustrate the use of awk for string parsing.

The excellent Bash Reference Manual, by Chet Ramey and Brian Fox, distributed as part of the "bash-2-doc" package (available as an rpm). See especially the instructive example scripts in this package.

The comp.os.unix.shell newsgroup.

The comp.os.unix.shell FAQ and its mirror site.

Assorted comp.os.unix FAQs.

The manpages for bash and bash2, date, expect, expr, find, grep, gzip, ln, patch, tar, tr, bc, xargs. The texinfo documentation on bash, dd, m4, gawk, and sed.