高级Bash脚本编程指南

一本深入学习shell脚本艺术的书籍

Mendel Cooper

<thegrendel@theriver.com>

春敏 杨 -

毅 黄 -

3.9.1

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bzip2-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!

目录

序

1. 原书作者致中国读者(英文)
2. 原书作者致中国读者(译文)
3. 黄毅
4. 杨春敏

第一部分. 热身

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

    2.1. 调用一个脚本
    2.2. 初步的练习

第二部分. 基本

3. 特殊字符
4. 变量和参数的介绍

    4.1. 变量替换
    4.2. 变量赋值
    4.3. Bash变量是不区分类型的

4.4. 特殊的变量类型

5. 引用

5.1. 引用变量
5.2. 转义

6. 退出和退出状态码
7. 条件判断

7.1. 条件测试结构
7.2. 文件测试操作符
7.3. 其他比较操作符
7.4. 嵌套的if/then条件测试
7.5. 检测你对测试知识的掌握情况

8. 操作符与相关主题

8.1. 操作符
8.2. 数字常量

第三部分. 进阶

9. 变量重游

9.1. 内部变量
9.2. 操作字符串
9.3. 参数替换
9.4. 指定变量的类型: 使用declare或者typeset
9.5. 变量的间接引用
9.6. $RANDOM: 产生随机整数
9.7. 双圆括号结构

10. 循环与分支

10.1. 循环
10.2. 嵌套循环
10.3. 循环控制
10.4. 测试与分支(case与select结构)

11. 内部命令与内建命令

11.1. 作业控制命令

12. 外部过滤器, 程序和命令

12.1. 基本命令
12.2. 复杂命令
12.3. 时间/日期 命令
12.4. 文本处理命令
12.5. 文件与归档命令
12.6. 通讯命令
12.7. 终端控制命令
12.8. 数学计算命令
12.9. 混杂命令

13. 系统与管理命令

13.1. 分析一个系统脚本

14. 命令替换
15. 算术扩展
16. I/O重定向

16.1. 使用exec
16.2. 代码块重定向
16.3. 重定向的应用

17. Here Document

17.1. Here String

18. 休息片刻

第四部分. 高级主题

19. 正则表达式

19.1. 一份简要的正则表达式介绍
19.2. 通配(globbing)

20. 子shell
21. 受限shell
22. 进程替换
23. 函数

23.1. 复杂函数和函数复杂性
23.2. 局部变量
23.3. 不使用局部变量的递归

24. 别名
25. 列表结构
26. 数组
27. /dev和/proc

27.1. /dev
27.2. /proc

28. Zero与Null
29. 调试
30. 选项
31. 陷阱
32. 脚本编程风格

32.1. 非官方的Shell脚本编写风格

33. 杂项

33.1. 交互与非交互式的交互与非交互式的shell和脚本
33.2. Shell包装
33.3. 测试和比较: 一种可选的方法
33.4. 递归
33.5. 将脚本"彩色化"
33.6. 优化
33.7. 各种小技巧
33.8. 安全问题
33.9. 可移植性问题
33.10. Windows下的shell脚本

34. Bash, 版本2与版本3

34.1. Bash, 版本2
34.2. Bash, 版本3

35. 后记

35.1. 作者后记

35.2. 关于作者
35.3. 译者后记

35.3.1. 杨春敏
35.3.2. 黄毅

35.4. 在哪里可以获得帮助
35.5. 用来制作这本书的工具

35.5.1. 硬件
35.5.2. 软件与排版软件

35.6. 致谢
35.7. 译者致谢

参考文献
A. 捐献的脚本
B. 参考卡片
C. 一个学习Sed和Awk的小手册

C.1. Sed
C.2. Awk

D. 带有特殊含义的退出码
E. I/O和I/O重定向的详细介绍
F. 命令行选项

F.1. 标准命令行选项
F.2. Bash命令行选项

G. 重要的文件
H. 重要的系统目录
I. 本地化
J. 历史命令
K. 一个简单的.bashrc文件
L. 将DOS批处理文件转换为Shell脚本
M. 练习

M.1. 分析脚本
M.2. 编写脚本

N. 修订记录
O. 翻译版修订记录
P. 镜像站点
Q. To Do列表
R. 版权

表格清单

11-1. 作业标识符
30-1. Bash选项
33-1. 转义序列中颜色与数值的对应
B-1. 特殊的shell变量
B-2. 测试操作: 二元比较
B-3. 文件类型的测试操作
B-4. 参数替换和扩展
B-5. 字符串操作
B-6. 一些结构的汇总
C-1. 基本sed操作
C-2. sed操作符举例
D-1. "保留的"退出码
L-1. 批处理文件关键字 / 变量 / 操作符, 和等价的shell符号
L-2. DOS命令与UNIX的等价命令

N-1. 修订历史
0-1. 翻译版修订历史

例子清单
2-1. 清除: 清除/var/log下的log文件
2-2. 清除:一个改良的清除脚本
2-3. 清除: 一个增强的和广义的删除logfile的脚本
3-1. 代码块和I/O重定向
3-2. 将一个代码块的结果保存到文件
3-3. 在后台运行一个循环
3-4. 备份最后一天所有修改的文件
4-1. 变量赋值和替换
4-2. 简单的变量赋值
4-3. 简单和复杂, 两种类型的变量赋值
4-4. 整型还是字符串?
4-5. 位置参数
4-6. wh, whois节点名字查询
4-7. 使用shift命令
5-1. echo出一些诡异变量
5-2. 转义符
6-1. 退出/退出状态码
6-2. 反转一个条件的用法!
7-1. 什么是真?
7-2. test, /usr/bin/test, [ ], 和/usr/bin/[都是等价命令
7-3. 算术测试需要使用(( ))
7-4. 测试那些断掉的链接文件
7-5. 算术比较与字符串比较
7-6. 检查字符串是否为*null*
7-7. zmore
8-1. 最大公约数
8-2. 使用算术操作符
8-3. 使用&&和||进行混合条件测试
8-4. 数字常量表示法
9-1. $IFS与空白字符
9-2. 定时输入
9-3. 再来一个, 定时输入
9-4. 定时read
9-5. 我是root么?
9-6. arglist: 通过$*和$@列出所有的参数
9-7. $*和$@的不一致的行为
9-8. 当$IFS为空时的$*和$@
9-9. 下划线变量
9-10. 在一个文本文件的段落之间插入空行
9-11. 转换图片文件格式, 同时更改文件名
9-12. 将音频流文件转换为*ogg*各式的文件
9-13. 模拟*getopt*
9-14. 提取字符串的另一种方法
9-15. 使用参数替换和错误消息
9-16. 参数替换和"usage"消息(译者注: 通常就是帮助信息)
9-17. 变量长度
9-18. 参数替换中的模式匹配
9-19. 修改文件扩展名:
9-20. 使用模式匹配来解析任意字符串
9-21. 对字符串的前缀和后缀使用匹配模式
9-22. 使用declare来指定变量的类型
9-23. 间接引用
9-24. 传递一个间接引用给*awk*
9-25. 产生随机整数
9-26. 从一幅扑克牌中取出一张随机的牌
9-27. 两个指定值之间的随机数

9-28. 用随机数来摇单个骰子
9-29. 重新分配随机数种子
9-30. 使用awk来产生伪随机数
9-31. C语言风格的变量操作
10-1. 一个简单的for循环
10-2. 每个[list]元素中都带有两个参数的for循环
10-3. *文件信息*: 对包含在变量中的文件列表进行操作
10-4. 在for循环中操作文件
10-5. 在for循环中省略**in [list]**部分
10-6. 使用命令替换来产生for循环的[list]
10-7. 对于二进制文件的grep替换
10-8. 列出系统上的所有用户
10-9. 在目录的所有文件中查找源字串
10-10. 列出目录中所有的符号链接
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
10-12. 一个C风格的for循环
10-13. 在batch mode中使用efax
10-14. 简单的while循环
10-15. 另一个while循环
10-16. 多条件的while循环
10-17. C风格的while循环
10-18. until循环
10-19. 嵌套循环
10-20. break和continue命令在循环中的效果
10-21. 多层循环的退出
10-22. 多层循环的continue
10-23. 在实际的任务中使用"continue N"
10-24. 使用case
10-25. 使用case来创建菜单
10-26. 使用命令替换来产生case变量
10-27. 简单的字符串匹配
10-28. 检查输入字符是否为字母
10-29. 使用select来创建菜单
10-30. 使用函数中的select结构来创建菜单
11-1. 一个fork出多个自身实例的脚本
11-2. 使用printf的例子
11-3. 使用read来进行变量分配
11-4. 当使用一个不带变量参数的read命令时, 将会发生什么?
11-5. read命令的多行输入
11-6. 检测方向键
11-7. 通过文件重定向来使用read命令
11-8. 管道输出到read中的问题
11-9. 修改当前工作目录
11-10. 使用"let"命令来做算术运算.
11-11. 展示eval命令的效果
11-12. 强制登出(log-off)
11-13. 另一个"rot13"版本
11-14. 在Perl脚本中使用eval命令来强制变量替换
11-15. 使用set命令来改变脚本的位置参数
11-16. 反转位置参数
11-17. 重新分配位置参数
11-18. "Unsett"一个变量
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
11-20. 使用getopts命令来来读取传递给脚本的选项/参数
11-21. "includ"一个数据文件
11-22. 一个(没什么用的)source自身的脚本
11-23. exec命令的效果
11-24. 一个exec自身的脚本
11-25. 在继续处理之前, 等待一个进程的结束
11-26. 一个结束自身的脚本程序

12-1. 使用ls命令来创建一个烧录CDR的内容列表
12-2. 到底是Hello还是Good-bye
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.
12-4. 通过文件的*inode*号来删除文件
12-5. Logfile: 使用xargs来监控系统log
12-6. 把当前目录下的文件拷贝到另一个文件中
12-7. 通过名字kill进程
12-8. 使用xargs分析单词出现的频率
12-9. 使用expr
12-10. 使用date命令
12-11. 分析单词出现的频率
12-12. 哪个文件是脚本?
12-13. 产生10-进制随机数
12-14. 使用tail命令来监控系统log
12-15. 在脚本中模拟"grep"的行为
12-16. 在1913年的韦氏词典中查找定义
12-17. 检查列表中单词的正确性
12-18. 转换大写: 把一个文件的内容全部转换为大写.
12-19. 转换小写: 将当前目录下的所有文全部转换为小写.
12-20. Du: DOS到UNIX文本文件的转换.
12-21. rot13: rot13, 弱智加密.
12-22. 产生"Crypto-Quote"游戏(译者: 一种文字游戏)
12-23. 格式化文件列表.
12-24. 使用column来格式化目录列表
12-25. nl: 一个自己计算行号的脚本.
12-26. manview: 查看格式化的man页
12-27. 使用cpio来拷贝一个目录树
12-28. 解包一个*rpm*归档文件
12-29. 从C文件中去掉注释
12-30. 浏览**/usr/X11R6/bin**
12-31. 一个"改进过"的*strings*命令
12-32. 在一个脚本中使用cmp命令来比较两个文件.
12-33. basename和dirname
12-34. 检查文件完整性
12-35. Uudecode编码后的文件
12-36. 查找滥用的链接来报告垃圾邮件发送者
12-37. 分析一个垃圾邮件域
12-38. 获得一份股票报价
12-39. 更新FC4(Fedora 4)
12-40. 使用ssh
12-41. 一个mail自身的脚本
12-42. 按月偿还贷款
12-43. 数制转换
12-44. 使用"here document"来调用bc
12-45. 计算圆周率
12-46. 将10进制数字转换为16进制数字
12-47. 因子分解
12-48. 计算直角三角形的斜边
12-49. 使用seq命令来产生循环参数
12-50. 字母统计
12-51. 使用getopt来分析命令行选项
12-52. 一个拷贝自身的脚本
12-53. 练习dd
12-54. 记录按键
12-55. 安全的删除一个文件
12-56. 文件名产生器
12-57. 将长度单位-米, 转化为英里
12-58. 使用m4
13-1. 设置一个新密码
13-2. 设置一个擦除字符

13-3. 保密密码：关闭终端对于密码的echo
13-4. 按键检测
13-5. 扫描远程机器上的*identd*服务进程
13-6. 使用pidof命令帮忙kill一个进程
13-7. 检查一个CD镜像
13-8. 在一个文件中创建文件系统
13-9. 添加一个新的硬盘驱动器
13-10. 用umask将输出文件隐藏起来
13-11. killall，来自于/etc/rc.d/init.d
14-1. 愚蠢的脚本策略
14-2. 将一个循环输出的内容设置到变量中
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，但是还是原来的子母集合)
16-1. 使用exec重定向stdin
16-2. 使用exec来重定向stdout
16-3. 使用exec在同一个脚本中重定向stdin和stdout
16-4. 避免子shell
16-5. *while*循环的重定向
16-6. 重定向*while*循环的另一种形式
16-7. 重定向*until*循环
16-8. 重定向*for*循环
16-9. 重定向*for*循环(stdin和stdout都进行重定向)
16-10. 重定向*if/then*测试结构
16-11. 用于上面例子的"names.data"数据文件
16-12. 事件纪录
17-1. 广播：将消息发送给每个登陆的用户
17-2. 虚拟文件：创建一个2行的虚拟文件
17-3. 使用cat的多行消息
17-4. 带有抑制tab功能的多行消息
17-5. 使用参数替换的here document
17-6. 上传一个文件对到"Sunsite"的incoming目录
17-7. 关闭参数替换
17-8. 生成另外一个脚本的脚本
17-9. Here document与函数
17-10. "匿名"的here Document
17-11. 注释掉一段代码块
17-12. 一个自文档化(self-documenting)的脚本
17-13. 在一个文件的开头添加文本
17-14. 分析一个邮箱
20-1. 子shell中的变量作用域
20-2. 列出用户的配置文件
20-3. 在子shell中进行并行处理
21-1. 在受限模式下运行脚本
23-1. 简单函数
23-2. 带参数的函数
23-3. 函数与传递给脚本的命令行参数
23-4. 将一个间接引用传递给函数
23-5. 对一个传递给函数的参数进行解除引用的操作
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
23-7. 取两个数中的最大值
23-8. 将阿拉伯数字转化为罗马数字
23-9. 测试函数最大的返回值
23-10. 比较两个大整数
23-11. 从username中取得用户的真名
23-12. 局部变量的可见范围
23-13. 使用局部变量的递归
23-14. 汉诺塔
24-1. 用在脚本中的别名
24-2. unalias：设置与删除别名
25-1. 使用"与列表"来测试命令行参数

25-2. 使用"与列表"来测试命令行参数的另一个例子
25-3. 将"或列表"和"与列表"结合使用
26-1. 简单的数组使用
26-2. 格式化一首诗
26-3. 多种数组操作
26-4. 用于数组的字符串操作
26-5. 将脚本的内容赋值给数组
26-6. 一些数组专用的小道具
26-7. 空数组与包含空元素的数组
26-8. 初始化数组
26-9. 拷贝和连接数组
26-10. 关于串联数组的更多信息
26-11. 一位老朋友: *冒泡排序*
26-12. 嵌套数组与间接引用
26-13. 复杂的数组应用: *埃拉托色尼素数筛子*
26-14. 模拟一个下推堆栈
26-15. 复杂的数组应用: *探索一个神秘的数学序列*
26-16. 模拟一个二维数组, 并使他倾斜
27-1. 利用/dev/tcp来检修故障
27-2. 找出与给定PID相关联的进程
27-3. 网络连接状态
28-1. 隐藏令人厌恶的cookie
28-2. 使用/dev/zero来建立一个交换文件
28-3. 创建一个ramdisk
29-1. 一个错误脚本
29-2. 缺少关键字
29-3. test24, 另一个错误脚本
29-4. 使用"assert"来测试条件
29-5. 捕获exit
29-6. Control-C之后, 清除垃圾
29-7. 跟踪一个变量
29-8. 运行多进程(在对称多处理器(SMP box)的机器上)
31-1. 数字比较与字符串比较并不相同
31-2. 子shell缺陷
31-3. 将echo的输出通过管道传递给read命令
33-1. shell包装
33-2. 稍微复杂一些的shell包装
33-3. 一个通用的shell包装, 用来写日志文件
33-4. 包装awd脚本的shell包装
33-5. 另一个包装awd脚本的shell包装
33-6. 将Perl嵌入到Bash脚本中
33-7. 将Bash和Perl脚本写到同一个文件中
33-8. 递归调用自身的(没用的)脚本
33-9. 递归调用自身的(有用的)脚本
33-10. 另一个递归调用自身的(有用的)脚本
33-11. 一个"彩色的"地址数据库
33-12. 画一个盒子
33-13. 显示彩色文本
33-14. "赛马"游戏
33-15. 返回值小技巧
33-16. 返回多个值的技巧
33-17. 传递数组到函数, 从函数中返回数组
33-18. anagram游戏
33-19. 从shell脚本中调用窗口部件
34-1. 字符串扩展
34-2. 间接变量引用 - 新方法
34-3. 使用间接变量引用的简单数据库应用
34-4. 使用数组和其他的小技巧来处理4人随机打牌
A-1. mailformat: 格式化一个e-mail消息
A-2. rn: 一个非常简单的文件重命名工具

A-3. blank-rename: 重命名包含空白的文件名
A-4. encryptedpw: 使用一个本地加密口令, 上传到一个ftp服务器.
A-5. copy-cd: 拷贝一个数据CD
A-6. Collatz序列
A-7. days-between: 计算两个日期之间天数差
A-8. 构造一个"字典"
A-9. Soundex转换
A-10. "Game of Life"
A-11. "Game of Life"的数据文件
A-12. behead: 去掉信件与新消息的头
A-13. ftpget: 通过ftp下载文件
A-14. password: 产生随机的8个字符的密码
A-15. fifo: 使用命名管道来做每日的备份
A-16. 使用模操作符来产生素数
A-17. tree: 显示目录树
A-18. string functions: C风格的字符串函数
A-19. 目录信息
A-20. 面向对象数据库
A-21. hash函数库
A-22. 使用hash函数来给文本上色
A-23. 深入hash函数
A-24. 挂载USB keychain型的存储设备
A-25. 保存weblog
A-26. 保护字符串的字面含义
A-27. 不保护字符串的字面含义
A-28. 鉴定是否是垃圾邮件服务器
A-29. 垃圾邮件服务器猎手
A-30. 使得wget更易用
A-31. 一个"podcasting"(译者: 指的是在互联网上发布音视频文件, 并允许用户订阅并自动接收的方法)脚本
A-32. 基础回顾
A-33. 一个扩展的cd命令
C-1. 计算字符出现次数
K-1. .bashrc文件样本
L-1. VIEWDATA.BAT: DOS批处理文件
L-2. viewdata.sh: 转换自VIEWDATA.BAT的shell脚本
Q-1. 打印服务器环境

下一页
序

12.5. 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:

1. -c 创建(一个新的归档文件)
2. -x 解压文件(从存在的归档文件中)
3. --delete 删除文件(从存在的归档文件中)

   这个选项不能用于磁带类型设备.
4. -r 将文件添加到现存的归档文件的尾部
5. -A 将*tar*文件添加到现存的归档文件的尾部
6. -t 列出现存的归档文件中包含的内容
7. -u 更新归档文件
8. -d 使用指定的文件系统，比较归档文件
9. -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
10. -j 用bzip2压缩归档文件

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.

```
```

```
bash$ rpm -qf /bin/ls
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.

```
bash$ rpm -qa
redhat-logos-1.1.3-1
 glibc-2.2.4-13
 cracklib-2.7-12
 dosfstools-2.7-1
 gdbm-1.8.0-10
 ksymoops-2.4.1-1
 mktemp-1.5-11
 perl-5.6.0-17
 reiserfs-utils-3.x.0j-2
 ...


bash$ rpm -qa docbook-utils
docbook-utils-0.6.9-2


bash$ rpm -qa docbook | grep docbook
docbook-dtd31-sgml-1.0-10
 docbook-style-dsssl-1.64-3
 docbook-dtd30-sgml-1.0-10
 docbook-dtd40-sgml-1.0-11
 docbook-utils-pdf-0.6.9-2
 docbook-dtd41-sgml-1.0-10
 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
10 ARGS=2
11 E_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.

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.

```
bash$ file test.tar.gz
test.tar.gz: gzip compressed data, deflated, last modified: Sun Sep 16
13:34:51 2001, os: Unix

bash file -z test.tar.gz
test.tar.gz: GNU tar archive (gzip compressed data, deflated, 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
 4 E_NOARGS=0
 5 E_ARGERROR=66
 6 E_WRONG_FILE_TYPE=67
 7
 8 if [ $# -eq "$E_NOARGS" ]
 9 then
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:
```

```
#--------
sed '
/^\/\*/d
/.*\*\//d
' $1
#--------
# 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.


#  如果注释和代码在同一行上, 上边的脚本就不行了.
#+ 所以需要添加一些代码来处理这种情况.
#  这是一个很重要的练习.

#  当然, 上边的代码也会删除带有"*/"的非注释行 --
#+ 这也不是一个令人满意的结果.

exit 0


# ------------------------------------------------------------------
# 下边的代码不会执行, 因为上边已经'exit 0'了.

# Stephane Chazelas建议使用下边的方法:

usage() {
  echo "Usage: `basename $0` C-program-file" >&2
  exit 1
}

WEIRD=`echo -n -e '\377'`    # 或者WEIRD=$'\377'
[[ $# -eq 1 ]] || usage
case `file "$1"` in
  *"C program text"*) sed -e "s%/\*%${WEIRD}%g;s%\*/%${WEIRD}%g" "$1" \
      | tr '\377\n' '\n\377' \
      | sed -ne 'p;n' \
      | tr -d '\n' | tr '\377' '\n';;
  *) usage;;
esac

#  如果是下列的这些情况, 还是很糟糕:
#  printf("/*");
#  或者
#  /*  /* buggy embedded comment */
#
#  为了处理上边所有这些特殊情况(字符串中的注释, 含有 \", \\" ...
#+ 的字符串中的注释)唯一的方法还是写一个C分析器
#+ (或许可以使用lex或者yacc?).

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, 这个命令就非常有用了.

```
$bash which rm
/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.

```
$bash whereis rm
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.

```
$bash whatis whatis
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
  5 DIRECTORY="/usr/X11R6/bin"
  6 # 也试试 "/bin", "/usr/bin", "/usr/local/bin", 等等.
  7
  8 for file in $DIRECTORY/*
  9 do
 10   whatis `basename $file`   # 将会echo出这个2进制文件的信息.
 11 done
 12
 13 exit 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*.

```
bash$ vdir
total 10
 -rw-r--r--    1 bozo  bozo      4034 Jul 18 22:04 data1.xrolo
 -rw-r--r--    1 bozo  bozo      4602 May 25 13:58 data1.xrolo.bak
 -rw-r--r--    1 bozo  bozo       877 Dec 17  2000 employment.xrolo

bash ls -l
total 10
 -rw-r--r--    1 bozo  bozo      4034 Jul 18 22:04 data1.xrolo
 -rw-r--r--    1 bozo  bozo      4602 May 25 13:58 data1.xrolo.bak
 -rw-r--r--    1 bozo  bozo       877 Dec 17  2000 employment.xrolo
```

locate, slocate

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).

```
$bash locate hickson
/usr/lib/xephem/catalogs/hickson.edb
```

readlink

显示符号链接所指向的文件.

```
bash$ readlink /usr/bin/awk
../../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
```

```
ARGS=1
E_BADARGS=65
E_NOFILE=66

if [ $# -ne $ARGS ]
then
  echo "Usage: `basename $0` filename"
  exit $E_BADARGS
fi

if [ ! -f "$1" ]                        # 检查文件是否存在.
then
    echo "File \"$1\" does not exist."
    exit $E_NOFILE
fi
# ============================================================


MINSTRLEN=3                             #  最小的字符串长度.
WORDFILE=/usr/share/dict/linux.words    #  字典文件.
                                        #  也可以指定一个不同的
                                        #+ 单词列表文件,
                                        #+ 但这种文件必须是以每个单词一行的方式
进行保存.


wlist=`strings "$1" | tr A-Z a-z | tr '[:space:]' Z | \
tr -cs '[:alpha:]' Z | tr -s '\173-\377' Z | tr Z ' '`

#  将'strings'命令的输出通过管道传递到多个'tr'命令中.
#  "tr A-Z a-z" 全部转换为小写字符.
#  "tr '[:space:]'" 转换空白字符为多个Z.
#  "tr -cs '[:alpha:]' Z" 将非字母表字符转换为多个Z,
#+ 然后去除多个连续的Z.
#  "tr -s '\173-\377' Z" 把所有z后边的字符都转换为Z.
#+ 并且去除多余重复的Z. (注意173(123 ascii "{")和377(255 ascii 最后一个字符)都
是8进制)
#+ 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
#+ 就全都转换为字符Z了.
#  最后"tr Z ' '" 将把所有的Z都转换为空格,
#+ 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.

#  ************************************************************
#  注意, 我们使用管道来将多个'tr'的输出传递到下一个'tr'时
#+ 每次都使用了不同的参数.
#  ************************************************************


for word in $wlist                      # 重要:
                                        # $wlist 这里不能使用双引号.
                                        # "$wlist" 不能正常工作.
                                        # 为什么不行?
do

  strlen=${#word}                       # 字符串长度.
  if [ "$strlen" -lt "$MINSTRLEN" ]     # 跳过短的字符串.
  then
    continue
  fi

  grep -Fw $word "$WORDFILE"            #  只匹配整个单词.
#      ^^^                              #  "固定字符串" 和
                                        #+ "整个单词" 选项.

done


exit $?
```

Comparison

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).

```
bash$ diff -r ~/notes1 ~/notes2
Only in /home/bozo/notes1: file02
 Only in /home/bozo/notes1: file03
 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.

```
bash$ diff3 file-1 file-2 file-3
====
 1:1c
   This is line 1 of "file-1".
 2:1c
   This is line 1 of "file-2".
 3:1c
   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.

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)

Utilities

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, csplit

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).

```
bash$ cksum /boot/vmlinuz
1670054224 804083 /boot/vmlinuz

bash$ echo -n "Top Secret" | cksum
3391003827 10



bash$ md5sum /boot/vmlinuz
0f43eccea8f09e0a0b2b5cf1dcf333ba  /boot/vmlinuz

bash$ echo -n "Top Secret" | md5sum
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
 5 E_DIR_NOMATCH=70
 6 E_BAD_DBFILE=71
 7
 8 dbfile=File_record.md5
 9 # 存储记录的文件名(数据库文件).
10
11
12 set_up_database ()
13 {
14   echo ""$directory"" > "$dbfile"
15   # 把目录名写到文件的第一行.
16   md5sum "$directory"/* >> "$dbfile"
17   # 在文件中附上md5 checksum和filename.
18 }
19
20 check_database ()
21 {
22   local n=0
23   local filename
24   local checksum
25
26   # ------------------------------------------- #
27   #  这个文件检查其实是不必要的,
28   #+ 但是能更安全一些.
29
30   if [ ! -r "$dbfile" ]
31   then
32     echo "Unable to read checksum database file!"
33     exit $E_BAD_DBFILE
```

```
 34   fi
 35   # ------------------------------------------- #
 36
 37   while read record[n]
 38   do
 39
 40     directory_checked="${record[0]}"
 41     if [ "$directory_checked" != "$directory" ]
 42     then
 43       echo "Directories do not match up!"
 44       # 换个目录试一下.
 45       exit $E_DIR_NOMATCH
 46     fi
 47
 48     if [ "$n" -gt 0 ]   # 不是目录名.
 49     then
 50       filename[n]=$( echo ${record[$n]} | awk '{ print $2 }' )
 51       #  md5sum向后写记录,
 52       #+ 先写checksum, 然后写filename.
 53       checksum[n]=$( md5sum "${filename[n]}" )
 54
 55
 56       if [ "${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$PWD中运行这个脚本时候,
 65          #+ 篡改这个checksum数
 66          #+ 据库文件将不会被检测出来.
 67          #  练习: 修正这个问题.
 68     then
 69           echo "${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                    #+ 那么就使用当前的工作目录.
 89   directory="$1"
 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   fi
102 # ------------------------------------------------------------------- #
103
104 check_database          # 调用主要处理函数.
105
106 echo
107
108 #  你可能想把这个脚本的输出重定向到文件中,
109 #+ 尤其在这个目录中有很多文件的时候.
110
111 exit 0
112
113 #  如果要对数量非常多的文件做完整性检查,
114 #+ 可以考虑一下"Tripwire"包,
115 #+ http://sourceforge.net/projects/tripwire/.
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.

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?

```
bash$ md5sum testfile
e181e2c8720c60522c4c4c981108e367  testfile


bash$ sha1sum testfile
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.

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 #!/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, mmencode

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cruft (请参考例子 A-4).

Miscellaneous

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.

```
bash$ mktemp
/tmp/tmp.zzsvql3154
```

```
 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.XXXXXXXXXX".
 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*.

序

目录

1. 原书作者致中国读者(英文)
2. 原书作者致中国读者(译文)
3. 黄毅
4. 杨春敏

1. 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

2. 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

3. 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了翻译，多谢你的理解，你是一个很棒的女朋友!

4. 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"在一起来建立应用. 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.

目录

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

1. 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.
- 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的经典的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中. 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.

注意事项

[1] 这些将在内建章节被引用, 这些都是shell的内部特征.

[2] *ksh88*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.

2. 带着一个Sha-Bang出发(Sha-Bang指的是#!)

Shell程序设计是1950年的光盘机 . . .
Larry Wall

目录
2.1. 调用一个脚本
2.2. 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
9 LOG_DIR=/var/log
10 # 如果使用变量,当然比把代码写死的好.
11 cd $LOG_DIR
12
13 cat /dev/null > messages
14 cat /dev/null > wtmp
15
16
17 echo "Logs cleaned up."
18
19 exit # 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
12
13 LOG_DIR=/var/log
14 ROOT_UID=0     # $UID为0的时候,用户才具有root用户的权限
15 LINES=50      # 默认的保存行数
16 E_XCD=66      # 不能修改目录?
17 E_NOTROOT=67  # 非root用户将以error退出
18
19
20 # 当然要使用root用户来运行.
21 if [ "$UID" -ne "$ROOT_UID" ]
22 then
23   echo "Must be root to run this script."
24   exit $E_NOTROOT
25 fi
26
27 if [ -n "$1" ]
28 # 测试是否有命令行参数(非空).
29 then
30   lines=$1
31 else
32   lines=$LINES # 默认,如果不在命令行中指定.
33 fi
34
35
36 #  Stephane Chazelas 建议使用下边
37 #+ 的更好方法来检测命令行参数.
38 #+ 但对于这章来说还是有点超前.
39 #
40 #    E_WRONGARGS=65  # 非数值参数(错误的参数格式)
41 #
42 #    case "$1" in
43 #    ""       ) lines=50;;
44 #    *[!0-9]*) echo "Usage: `basename $0` file-to-cleanup"; 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 $E_XCD;
65 # }
66
67
68
69
70 tail -$lines messages > mesg.temp # 保存log file消息的最后部分.
71 mv mesg.temp messages             # 变为新的log目录.
72
73
74 # cat /dev/null > messages
75 #* 不再需要了,使用上边的方法更安全.
76
77 cat /dev/null > wtmp  #  ': > wtmp' 和 '> wtmp'具有相同的作用
78 echo "Logs cleaned up."
79
80 exit 0
81 #  退出之前返回0,
82 #+ 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 -- 通常是"Command not found" -- 这将是你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. . . 当然这个文件消失不见了.
 5
 6 WHATEVER=65
 7
 8 echo "This line will never print (betcha!)."
 9
10 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 *I*nterface(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shell来开启一个脚本的话, 比如*tcsh*, 那么你就*必须*需要#!了.

2.1. 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.

2.2. 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.

第二部分. 基本

目录
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

3. 特殊字符

用在脚本和其他地方的特殊字符

#

注释. 行首以#(#!是个例外)开头是注释.

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

注释也可以放在于本行命令的后边.

```
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#.

;

命令分隔符[分号, 即;]. 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.

;;

终止case选项[双分号, 即;;].

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

.

"点"命令[句点，即.]. 等价于source命令(参见 例子 11-21). 这是一个bash的内建命令.

.

"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.

```
bash$ touch .hidden-file
bash$ ls -l
total 10
 -rw-r--r--    1 bozo      4034 Jul 18 22:04 data1.addressbook
 -rw-r--r--    1 bozo      4602 May 25 13:58 data1.addressbook.bak
 -rw-r--r--    1 bozo       877 Dec 17  2000 employment.addressbook


bash$ ls -al
total 14
 drwxrwxr-x    2 bozo  bozo      1024 Aug 29 20:54 ./
 drwx------   52 bozo  bozo      3072 Aug 29 20:51 ../
 -rw-r--r--    1 bozo  bozo      4034 Jul 18 22:04 data1.addressbook
 -rw-r--r--    1 bozo  bozo      4602 May 25 13:58
data1.addressbook.bak
 -rw-r--r--    1 bozo  bozo       877 Dec 17  2000
employment.addressbook
 -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.

```
bash$ pwd
/home/bozo/projects

bash$ cd .
bash$ pwd
/home/bozo/projects

bash$ cd ..
bash$ pwd
/home/bozo/
```

点经常会出现在文件移动命令的目的参数(目录)的位置上.

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

.

"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.

"

部分引用[双引号，即"]. "*STRING*"将会阻止(解释)*STRING*中大部分特殊的字符. 参见 5.

'

全引用[单引号，即']. '*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 5.

,

逗号操作符. 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"
```

\

转义符[反斜线，即\]. 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 5来深入地了解转义符的详细解释.

/

文件名路径分隔符[斜线, 即/]. 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.

`

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.

:

空命令[冒号, 即:]. 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":"的话, 将会给出一个错误,
3 #                         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.

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

!

取反操作符[叹号, 即!]. !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.

*

通配符[星号, 即*]. *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.

```
bash$ echo *
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.

*

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.

?

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. .

?

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.

$

变量替换(引用变量的内容).

```
  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*.

$

行结束符. 在正则表达式中, "$"表示行结束符.

${}

参数替换.

$*, $@

位置参数.

$?

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.

$$

进程ID变量. 这个$$ 变量 保存了它所在脚本的*进程 ID* [1]

()

命令组.

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

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
5 cp file22.{txt,backup}
6 # 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.

在大括号中, 不允许有空白, *除非*这个空白被引用或转义.

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

file1 : A file1 : B file1 : C file2 : A file2 : B file2 : C
```

{}

代码块[大括号, 即{}]. 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.

```
bash$ { local a;
                a=123; }
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, 影响了外边的
变量)
  4
  5 # 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. 代码块和I/O重定向

```
  1 #!/bin/bash
  2 # 从/etc/fstab中读行.
  3
  4 File=/etc/fstab
  5
  6 {
  7 read line1
  8 read line2
  9 } < $File
 10
 11 echo "First line in $File is:"
 12 echo "$line1"
 13 echo
 14 echo "Second line in $File is:"
 15 echo "$line2"
 16
 17 exit 0
 18
 19 # 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
  9 SUCCESS=0
 10 E_NOARGS=65
 11
 12 if [ -z "$1" ]
 13 then
 14   echo "Usage: `basename $0` rpm-file"
 15   exit $E_NOARGS
 16 fi
 17
 18 {
 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
36 echo "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
```

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.

";"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.

[ ]

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.

[[ ]]

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.

[ ]

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.

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

[ ]

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.

(( ))

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.

> &> >& >> < <>

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.

<<

用在here document中的重定向.

<<<

用在here string中的重定向.

<, >

ASCII comparison.

```
  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 .

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

|

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.

```
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.

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

管道中的每个进程的stdout比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.

```
bash$ sleep 10 &
[1] 850
[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
  4 for i in 1 2 3 4 5 6 7 8 9 10            # 第一个循环.
  5 do
  6   echo -n "$i "
  7 done & # 在后台运行这个循环.
  8        # 在第2个循环之后, 将在某些时候执行.
  9
 10 echo   # 这个'echo'某些时候将不会显示.
 11
 12 for i in 11 12 13 14 15 16 17 18 19 20   # 第二个循环.
 13 do
 14   echo -n "$i "
 15 done
 16
 17 echo   # 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?)
33
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.
```

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. 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sucess).

-

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
```

-

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>, 走出了部分修改]
 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 #                              :'x'          , 'p', 'v', 'f'       .
```

```
                                    注意       是一个命令                    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 xpvf -)
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" 没能够正常的处理"bunzip2",
 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.

```
bash$ echo "whatever" | cat -
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.

```
bash$ file
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.

```
bash$ file -
abc
standard input:              ASCII text



bash$ file -
#!/bin/bash
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:

```
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 #!/bin/bash
 2
 3 #  在一个"tarball"中(经过tar和gzip处理过的文件)
 4 #+ 备份最后24小时当前目录下d所有修改的文件.
 5
 6 BACKUPFILE=backup-$(date +%m-%d-%Y)
 7 #                  在备份文件中嵌入时间.
 8 #                  Thanks, Joshua Tschida, for the idea.
 9 archive=${1:-$BACKUPFILE}
10 #  如果在命令行中没有指定备份文件的文件名,
11 #+ 那么将默认使用"backup-MM-DD-YYYY.tar.gz".
12
```

```
 13 tar cvf - `find . -mtime -1 -type f -print` > $archive.tar
 14 gzip $archive.tar
 15 echo "Directory $PWD backed up in archive file \"$archive.tar.gz\"."
 16
 17
 18 #  Stephane Chazelas指出上边代码,
 19 #+ 如果在发现太多的文件的时候, 或者是如果文件
 20 #+ 名包括空格的时候, 将执行失败.
 21
 22 # Stephane Chazelas建议使用下边的两种代码之一:
 23 # --------------------------------------------------------------------
--
 24 #   find . -mtime -1 -type f -print0 | xargs -0 tar rvf
"$archive.tar"
 25 #       使用gnu版本的"find".
 26
 27
 28 #   find . -mtime -1 -type f -exec tar rvf "$archive.tar" '{}' \;
 29 #         对于其他风格的UNIX便于移植, 但是比较慢.
 30 # --------------------------------------------------------------------
--
 31
 32
 33 exit 0
```

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.
```

\-

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.

在另一种上下文环境中, +也是一种正则表达式操作.

\+

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.

%

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.

~

home目录[波浪号, 即~]. 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.

```
bash$ echo ~bozo
/home/bozo

bash$ echo ~
/home/bozo

bash$ echo ~/
/home/bozo/

bash$ echo ~:
/home/bozo:

bash$ echo ~nonexistent-user
~nonexistent-user
```

~+

当前工作目录. 相当于$PWD内部变量.

~-

先前的工作目录. 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
 4 read -n 1 -s -p $'Control-M leaves cursor at beginning of this
line. Press Enter. \x0d'
 5                                        # 当然, '0d'就是二进制的回车.
 6 echo >&2   #  '-s'参数使得任何输入都不将回显出来.
 7            #+ 所以, 明确的重起一行是必要的.
 8
 9 read -n 1 -s -p $'Control-J leaves cursor on next line. \x0a'
10            #  '0a' 等价于Control-J, 换行.
11 echo >&2
12
13 ###
14
15 read -n 1 -s -p $'And Control-K\x0bgoes straight down.'
16 echo >&2   #  Control-K 是垂直制表符.
17
18 # 关于垂直制表符效果的一个更好的例子见下边:
19
20 var=$'\x0aThis is the bottom line\x0bThis is the top line\x0a'
21 echo "$var"
22 #  这句与上边的例子使用的是同样的办法, 然而:
23 echo "$var" | col
24 #  这将造成垂直制表符右边的部分比左边部分高.
25 #  这也解释了为什么我们要在行首和行尾加上一个换行符 --
26 #+ 这样可以避免屏幕显示混乱.
27
28 # Lee Maschmeyer的解释:
29 # -------------------------
30 #  在这里[第一个垂直制表符的例子中]  .  .  .
31 #+ 这个垂直制表符使得还没回车就直接打印下来.
32 #  这只能在那些不能"后退"的设备中才行,
33 #+ 比如说Linux的console.
34 #  垂直制表符的真正意义是向上移, 而不是向下.
35 #  它可以用来让打印机打印上标.
36 #  col工具可以模拟垂直制表符的正确行为.
```

```
37
38 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"  # 不能工作.
5
6 # 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.

---

4. 变量和参数的介绍

目录
4.1. 变量替换
4.2. 变量赋值
4.3. Bash变量是不区分类型的
4.4. 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, 而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.

4.1. 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*, 即这边变量所包含的数据.

```
bash$ variable=23


bash$ echo variable
variable

bash$ echo $variable
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exporte, 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Section 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, If there is whitespace embedded within a
variable,
 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
```

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.

4.2. 变量赋值

=

赋值操作(*前后都不能有空白*)

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

```
#!/bin/bash
# "裸体"变量

echo

# 变量什么时候是"裸体"的，比如前边少了$的时候?
# 当它被赋值的时候，而不是被引用的时候.

# 赋值
a=879
echo "The value of \"a\" is $a."

# 使用'let'赋值
let a=16+5
echo "The value of \"a\" is now $a."

echo

# 在'for'循环中(事实上，这是一种伪赋值):
echo -n "Values of \"a\" in the loop are: "
for a in 7 8 9 11
do
  echo -n "$a "
done

echo
echo

# 使用'read'命令进行赋值(这也是一种赋值的类型):
echo -n "Enter \"a\" "
read a
echo "The value of \"a\" is now $a."

echo

exit 0
```

例子 4-3. 简单和复杂，两种类型的变量赋值

```
#!/bin/bash

a=23              # 简单的赋值
echo $a
b=$a
echo $b

# 现在让我们来点小变化(命令替换).

a=`echo Hello!`   # 把'echo'命令的结果传给变量'a'
echo $a
#  注意，如果在一个#+的命令替换结构中包含一个(!)的话,
#+ 那么在命令行下将无法工作.
#+ 因为这触发了Bash的"历史机制."
#  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.

```

```
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)
```

4.3. 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?

```
#!/bin/bash
# int-or-string.sh: 整型还是字符串?

a=2334                        # 整型.
let "a += 1"
echo "a = $a "                # a = 2335
echo                          # 还是整型.


b=${a/23/BB}                  # 将"23"替换成"BB".
                              # 这将把变量b从整型变为字符串.
echo "b = $b"                 # b = BB35
declare -i b                  # 即使使用declare命令也不会对此有任何帮助.
echo "b = $b"                 # b = BB35

let "b += 1"                  # BB35 + 1 =
echo "b = $b"                 # b = 1
echo

c=BB34
echo "c = $c"                 # c = BB34
d=${c/BB/23}                  # 将"BB"替换成"23".
                              # 这使得变量$d变为一个整形.
echo "d = $d"                 # d = 2334
let "d += 1"                  # 2334 + 1 =
echo "d = $d"                 # d = 2335
echo

# null变量会如何呢?
e=""
echo "e = $e"                 # e =
let "e += 1"                  # 算术操作允许一个null变量?
echo "e = $e"                 # e = 1
echo                          # null变量将被转换成一个整型变量.

# 如果没有声明变量会怎样?
echo "f = $f"                 # f =
let "f += 1"                  # 算术操作能通过么?
echo "f = $f"                 # f = 1
echo                          # 未声明的变量将转换成一个整型变量.


# 所以说Bash中的变量都是不区分类型的.

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.

4.4. 特殊的变量类型

局部变量

这种变量只有在代码块或者函数中(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).

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.

```
bash$ eval "`seq 10000 | sed -e 's/.*/export
var&=ZZZZZZZZZZZZZ/'`"

bash$ du
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
  5 MINPARAMS=10
  6
  7 echo
  8
  9 echo "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.

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
 8 until [ -z "$1" ]  # 直到所有的位置参数都被存取完...
 9 do
10   echo -n "$1 "
```

```
11   shift
12 done
13
14 echo                 # 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.

5. 引用

目录
5.1. 引用变量
5.2. 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中具有的特殊意义, 而不是字符的字面意思, 比如通配符 -- *.)

```
bash$ ls -l [Vv]*
-rw-rw-r--    1 bozo  bozo       324 Apr  2 15:05 VIEWDATA.BAT
 -rw-rw-r--    1 bozo  bozo       507 May  4 14:25 vartrace.sh
 -rw-rw-r--    1 bozo  bozo       539 Apr 14 17:11 viewdata.sh

bash$ ls -l '[Vv]*'
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.

```
bash$ grep '[Ff]irst' *.txt
file1.txt:This is the first line of file1.txt.
 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".

```
bash$ echo $(ls -l)
total 8 -rw-rw-r-- 1 bozo bozo 130 Aug 21 12:57 t222.sh -rw-rw-r-- 1 bozo bozo 78
Aug 21 12:57 t71.sh


bash$ echo "$(ls -l)"
total 8
 -rw-rw-r--  1 bozo bozo 130 Aug 21 12:57 t222.sh
 -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.

5.1. 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.

```
variable1="a variable containing five words"
COMMAND This is $variable1    # 用下面7个参数执行COMMAND命令:
# "This" "is" "a" "variable" "containing" "five" "words"

COMMAND "This is $variable1"  # 用下面1个参数执行COMMAND命令:
# "This is a variable containing five words"


variable2=""    # Empty.

COMMAND $variable2 $variable2 $variable2        # COMMAND将不带参数执行.
COMMAND "$variable2" "$variable2" "$variable2"  # COMMAND将以3个空参数来执行.
COMMAND "$variable2 $variable2 $variable2"      # COMMAND将以1个参数来执行(2空格).

# 感谢, Stephane Chazelas.
```

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

```
#!/bin/bash
# weirdvars.sh: echo出一些诡异变量.

var="'(]\\{}\$\""
echo $var        # '(]\{}$"
echo "$var"      # '(]\{}$"     和上一句没什么区别.Doesn't make a difference.

echo

IFS='\'
echo $var        # '(] {}$"     \ 字符被空白符替换了, 为什么?
echo "$var"      # '(]\{}$"

# 这个例子由Stephane Chazelas提供.

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.

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).

```
echo "Why can't I write 's between single quotes"

echo

# 一种绕弯的方法.
echo 'Why can'\''t I write '"'"'s between single quotes'
#    |-------|  |----------|   |------------------------|
# 三个被单引号引用的字符串, 在这三个字符串之间有一个用转义符转义的单引号, 和一个用双引号括起来的单引号.

# 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.

```
bash$ echo hello\!
hello!


bash$ echo "hello\!"
hello\!




bash$ echo -e x\ty
xty


bash$ echo -e "x\ty"
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.

5.2. 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.

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).
```

```
 21
 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, $quote and this lies outside
the quotes."
 40
 41 echo
 42
 43 # 变量中的连续的ASCII字符.
 44 triple_underline=$'\137\137\137'   # 137是八进制的'_'.
 45 echo "$triple_underline UNDERLINE $triple_underline"
 46
 47 echo
 48
 49 ABC=$'\101\102\103\010'           # 101, 102, 103是八进制码的A, B, C.
 50 echo $ABC
 51
 52 echo; echo
 53
 54 escape=$'\033'                    # 033 是八进制码的esc.
 55 echo "\"escape\" echoes as $escape"
 56 #                                    没有变量被输出.
 57
 58 echo; echo
 59
 60 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)

```
  1 echo "\$variable01"  # 结果是$variable01
```

\\

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(译者注: 提示你
命令不全, 在添加一个"就好了.
  6            # 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(""), 或者是否出现在命令替换或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 /usr/bin/emacs-20.7"
 2 # 列出的文件都作为命令的参数.
 3
 4 # 加两个文件到参数列表中, 列出所有的文件信息.
 5 ls -l /usr/X11R6/bin/xsetroot /sbin/dump $file_list
 6
 7 echo "------------------------------------------------------------------------"
 8
 9 # 如果我们将上边的两个空个转义了会产生什么效果?
10 ls -l /usr/X11R6/bin/xsetroot\ /sbin/dump\ $file_list
11 # 错误: 因为前3个路径被合并成一个参数传递给了'ls -l'
12 #       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.)
```

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.
```

6. 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).

!, 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).
15
16 # 感谢, Stephane Chazelas和Kristopher Newsome.
```

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.

7. 条件判断

目录
7.1. 条件测试结构
7.2. 文件测试操作符
7.3. 其他比较操作符
7.4. 嵌套的if/then条件测试
7.5. 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.

7.1. 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?

```
#!/bin/bash

#  小技巧:
#  如果你不能够确定一个特定的条件该如何进行判断,
#+ 那么就使用if-test结构.

echo

echo "Testing \"0\""
if [ 0 ]      # zero
then
  echo "0 is true."
else
  echo "0 is false."
fi            # 0 为真.

echo

echo "Testing \"1\""
if [ 1 ]      # one
then
  echo "1 is true."
else
  echo "1 is false."
fi            # 1 为真.

echo

echo "Testing \"-1\""
if [ -1 ]     # 负1
then
  echo "-1 is true."
else
  echo "-1 is false."
fi            # -1 为真.

echo

echo "Testing \"NULL\""
if [ ]        # NULL (空状态)
then
  echo "NULL is true."
else
  echo "NULL is false."
fi            # NULL 为假.

echo

echo "Testing \"xyz\""
if [ xyz ]    # 字符串
then
  echo "Random string is true."
else
  echo "Random string is false."
fi            # 随便的一串字符为真.

echo

echo "Testing \"\$xyz\""
if [ $xyz ]   # 判断$xyz是否为null, 但是...
              # 这只是一个未初始化的变量.
then
  echo "Uninitialized variable is true."
else
  echo "Uninitialized variable is false."
fi            # 未定义的初始化为假.

echo

echo "Testing \"-n \$xyz\""
if [ -n "$xyz" ]            # 更加正规的条件检查.
then
  echo "Uninitialized variable is true."
else
  echo "Uninitialized variable is false."
fi            # 未初始化的变量为假.

echo


xyz=          # 初始化了, 但是赋null值.

echo "Testing \"-n \$xyz\""
if [ -n "$xyz" ]
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 #   如果我们测试以下为初始化的变量"$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.

```
bash$ type test
test is a shell builtin
bash$ type '['
[ is a shell builtin
bash$ type '[['
[[ is a shell keyword
bash$ type ']]'
]] is a shell keyword
bash$ type ']'
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 #                              # 注意:
38 #                                在版本3.x的Bash中, 这个bug已经被修正了.
39 then
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.

```
file=/etc/passwd

if [[ -e $file ]]
then
  echo "Password file exists."
fi
```

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 [[ ]]).

```
dir=/home/bozo

if cd "$dir" 2>/dev/null; then   # "2>/dev/null" 会隐藏错误信息.
  echo "Now in $dir."
else
  echo "Can't change to $dir."
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.

```
var1=20
var2=22
[ "$var1" -ne "$var2" ] && echo "$var1 is not equal to $var2"

home=/home/bozo
[ -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(( ))

```
#!/bin/bash
# 算术测试.

# (( ... ))结构可以用来计算并测试算术表达式的结果.
# 退出状态将会与[ ... ]结构完全相反!

(( 0 ))
echo "Exit status of \"(( 0 ))\" is $?."         # 1

(( 1 ))
echo "Exit status of \"(( 1 ))\" is $?."         # 0

(( 5 > 4 ))                                      # 真
echo "Exit status of \"(( 5 > 4 ))\" is $?."     # 0

(( 5 > 9 ))                                      # 假
echo "Exit status of \"(( 5 > 9 ))\" is $?."     # 1

(( 5 - 5 ))                                      # 0
echo "Exit status of \"(( 5 - 5 ))\" is $?."     # 1

(( 5 / 4 ))                                      # 除法也可以.
echo "Exit status of \"(( 5 / 4 ))\" is $?."     # 0

(( 1 / 2 ))                                      # 除法的计算结果 < 1.
echo "Exit status of \"(( 1 / 2 ))\" is $?."     # 截取之后的结果为 0.
                                                 # 1

(( 1 / 0 )) 2>/dev/null                          # 除数为0，非法计算.
#           ^^^^^^^^^^
echo "Exit status of \"(( 1 / 0 ))\" is $?."     # 1

# "2>/dev/null"起了什么作用?
# 如果这句被删除会怎样?
# 尝试删除这句，然后在运行这个脚本.

exit 0
```

7.2. 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

文件大小不为零

-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.

```
                -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*.

```
               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 # 已经争得作者的授权引用在本书中.
 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.

7.3. 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

-eq

等于

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

-ne

不等于

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

-gt

大于

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

-ge

大于等于

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

-lt

小于

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

-le

小于等于

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

<

小于(在双括号中使用)

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

<=

小于等于(在双括号中使用)

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

>

大于(在双括号中使用)

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

>=

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 [ $a == z* ]      # 文件扩展匹配(file globbing)和单词分割有效.
  5 [ "$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.
  6
  7 # 感谢, Stephane Chazelas
```

!=

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

```
#!/bin/bash

a=4
b=5

#  这里的"a"和"b"既可以被认为是整型也可被认为是字符串.
#  这里在算术比较与字符串比较之间是容易让人产生混淆,
#+ 因为Bash变量并不是强类型的.

#  Bash允许对于变量进行整形操作与比较操作.
#+ 但前提是变量中只能包含数字字符.
#  不管怎么样, 还是要小心.

echo

if [ "$a" -ne "$b" ]
then
  echo "$a is not equal to $b"
  echo "(arithmetic comparison)"
fi

echo

if [ "$a" != "$b" ]
then
  echo "$a is not equal to $b."
  echo "(string comparison)"
  #     "4"  != "5"
  # ASCII 52 != ASCII 53
fi

# 在这个特定的例子中, "-ne"和"!="都可以.

echo

exit 0
```

例子 7-6. 检查字符串是否为*null*

```
#!/bin/bash
#  str-test.sh: 检查null字符串和未引用的字符串,
#+ but not strings and sealing wax, not to mention cabbages and kings . . .
#+ 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)

# 使用   if [ ... ]


# 如果字符串并没有被初始化, 那么它里面的值未定义.
# 这种状态被称为"null" (注意这与零值不同).

if [ -n $string1 ]    # $string1 没有被声明和初始化.
then
  echo "String \"string1\" is not null."
else
  echo "String \"string1\" is null."
fi
# 错误的结果.
# 显示$string1为非null, 虽然这个变量并没有被初始化.


echo


# 让我们再试一下.

if [ -n "$string1" ]  # 这次$string1被引号扩起来了.
then
  echo "String \"string1\" is not null."
else
  echo "String \"string1\" is null."
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, 一个是空的"$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 NOTGZIP=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 $NOTGZIP
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).

7.4. 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.

7.5. 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 ]; then
11             netscape /usr/share/doc/HTML/index.html &
12      fi
13 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.

8. 操作符与相关主题

目录
8.1. 操作符
8.2. 数字常量

8.1. 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.
```

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*)

```
bash$ expr 5 % 3
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

```
#!/bin/bash
# gcd.sh: 最大公约数
#         使用Euclid的算法

#  两个整数的"最大公约数" (gcd),
#+ 就是两个整数所能够同时整除的最大的数.

#  Euclid算法采用连续除法.
#  在每一次循环中,
#+ 被除数 <---  除数
#+ 除数 <---  余数
#+ 直到 余数 = 0.
#+ 在最后一次循环中, gcd = 被除数.
#
#  关于Euclid算法的更精彩的讨论, 可以到
#+ Jim Loy的站点, http://www.jimloy.com/number/euclids.htm.


# ------------------------------------------------------
# 参数检查
ARGS=2
E_BADARGS=65

if [ $# -ne "$ARGS" ]
then
  echo "Usage: `basename $0` first-number second-number"
  exit $E_BADARGS
fi
# ------------------------------------------------------


gcd ()
{

  dividend=$1                         #  随意赋值.
  divisor=$2                          #+ 在这里, 哪个值给的大都没关系.
                                      #  为什么没关系?

  remainder=1                         #  如果在循环中使用了未初始化的变量,
                                      #+ 那么在第一次循环中,
                                      #+ 它将会产生一个错误消息.

  until [ "$remainder" -eq 0 ]
  do
    let "remainder = $dividend % $divisor"
    dividend=$divisor                 # 现在使用两个最小的数来重复.
    divisor=$remainder
  done                                # Euclid的算法

}                                     # Last $dividend is the gcd.


gcd $1 $2

echo; echo "GCD of $1 and $2 = $dividend"; echo


# Exercise :
# --------
#  检查传递进来的命令行参数来确保它们都是整数.
#+ 如果不是整数, 那就给出一个适当的错误消息并退出脚本.

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在 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
 4 n=1; echo -n "$n "
 5
 6 let "n = $n + 1"   # let "n = n + 1"  也可以.
 7 echo -n "$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)

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

在2.05b版本之后, Bash开始支持64位整型了.

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.

```
a=1.5

let "b = $a + 1.3"  # 错误.
# t2.sh: let: b = 1.5 + 1.3: 表达式的语法错误(错误标志为".5 + 1.3")

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.

位操作符

<<

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)

```
if [ $condition1 ] && [ $condition2 ]
# 与 if [ $condition1 -a $condition2 ] 相同
# 如果condition1和condition2都为true, 那结果就为true.

if [[ $condition1 && $condition2 ]]    # 也可以.
# 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)

```
if [ $condition1 ] || [ $condition2 ]
# 与 if [ $condition1 -o $condition2 ] 相同
# 如果condition1或condition2中的一个为true, 那么结果就为true.

if [[ $condition1 || $condition2 ]]    # 也可以.
# 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

```
#!/bin/bash

a=24
b=47

if [ "$a" -eq 24 ] && [ "$b" -eq 47 ]
then
  echo "Test #1 succeeds."
else
  echo "Test #1 fails."
fi

# ERROR:   if [ "$a" -eq 24 && "$b" -eq 47 ]
#+          尝试运行' [ "$a" -eq 24 '
#+          因为没找到匹配的']'所以失败了.
#
#  注意:  if [[ $a -eq 24 && $b -eq 24 ]]  也能正常运行.
#  双中括号的if-test结构要比
#+ 单中括号的if-test结构更加灵活.
#    (在第17行"&&"与第6行的"&&"具有不同的含义.)
#    感谢, Stephane Chazelas, 指出这点.


if [ "$a" -eq 98 ] || [ "$b" -eq 47 ]
then
  echo "Test #2 succeeds."
else
  echo "Test #2 fails."
fi

```

```
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.

```
bash$ echo $(( 1 && 2 )) $((3 && 0)) $((4 || 0)) $((0 || 0))
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*负作用*), 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.

8.2. 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

```
#!/bin/bash
# numbers.sh: 几种不同数制的数字表示法.

# 10进制: 默认情况
let "dec = 32"
echo "decimal number = $dec"                # 32
# 这没什么特别的.


# 8进制: 以'0'(零)开头
let "oct = 032"
echo "octal number = $oct"                  # 26
# 表达式结果是用10进制表示的.
# ---------------------------

# 16进制: 以'0x'或者'0X'开头的数字
let "hex = 0x32"
echo "hexadecimal number = $hex"            # 50
# 表达式结果是用10进制表示的.

# 其他进制: BASE#NUMBER
# BASE的范围在2到64之间.
# 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.


let "bin = 2#111100111001101"
echo "binary number = $bin"                 # 31181

let "b32 = 32#77"
echo "base-32 number = $b32"                # 231

let "b64 = 64#@_"
echo "base-64 number = $b64"                # 4031
# 这个表示法只能工作于受限的ASCII字符范围(2 - 64).
# 10个数字 + 26个小写字母 + 26个大写字符 + @ + _


echo

echo $((36#zz)) $((2#10101010)) $((16#AF16)) $((53#1aA))
                                            # 1295 170 44822 3375


#  重要的注意事项:
#  ---------------
#  使用一个超出给定进制的数字的话,
#+ 将会引起一个错误.

let "bad_oct = 081"
# (部分的) 错误消息输出:
#  bad_oct = 081: value too great for base (error token is "081")
#               Octal numbers use only digits in the range 0 - 7.

exit 0       # 感谢, Rich Bartell 和 Stephane Chazelas的指正.
```

第三部分. 进阶

目录

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

9. 变量重游

目录

9.1. 内部变量
9.2. 操作字符串

9.2.1. 使用awk来处理字符串
9.2.2. 更深入的讨论

9.3. 参数替换
9.4. 指定变量的类型：使用declare或者typeset
9.5. 变量的间接引用
9.6. $RANDOM：产生随机整数
9.7. 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.

9.1. 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

```
bash$ echo $BASH
/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

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

```
tcsh% echo $BASH_VERSION
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.

$EUID并不一定与$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.

```
root# echo $GROUPS
0


root# echo ${GROUPS[1]}
1


root# echo ${GROUPS[5]}
6
```

$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.

```
bash$ echo $HOSTTYPE
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.

```
bash$ echo $IFS | cat -vte
$
(Show tabs and display "$" at end-of-line.)


bash$ bash -c 'set w x y z; IFS=":-;"; echo "$*"'
w:x:y:z
(从字符串中读取命令, 并分配参数给位置参数.)
```

$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 # $IFS 处理空白与处理其他字符不同.
  3
  4 output_args_one_per_line()
  5 {
  6   for arg
  7   do echo "[$arg]"
  8   done
  9 }
 10
 11 echo; echo "IFS=\" \""
 12 echo "-------"
 13
 14 IFS=" "
 15 var=" a  b c   "
 16 output_args_one_per_line $var  #
output_args_one_per_line `echo " a  b c   "`
 17 #
 18 # [a]
 19 # [b]
 20 # [c]
 21
 22
 23 echo; echo "IFS=:"
 24 echo "-----"
 25
 26 IFS=:
 27 var=":a::b:c:::"                # 与上边一样, 但是用" "替换
了":".
 28 output_args_one_per_line $var
 29 #
 30 # []
 31 # [a]
 32 # []
 33 # [b]
 34 # [c]
 35 # []
 36 # []
 37 # []
 38
 39 # 同样的事情也会发生在awk的"FS"域中.
 40
 41 # 感谢, Stephane Chazelas.
 42
 43 echo
 44
 45 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.

$IGNOREEOF

忽略EOF: 告诉shell在log out之前要忽略多少文件结束符(control-D).

$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.

```
bash$ echo $MACHTYPE
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

```
bash$ echo $OSTYPE
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).

```
bash$ echo $PATH
/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.

```
bash$ echo $PIPESTATUS
```

```
0

bash$ ls -al | bogus_command
bash: bogus_command: command not found
bash$ echo $PIPESTATUS
141

bash$ ls -al | bogus_command
bash: bogus_command: command not found
bash$ echo $?
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.

$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).

```
tcsh% bash

bash$ who | grep nobody | sort
bash$ echo ${PIPESTATUS[*]}
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.

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

bash$ $ ls | bogus_command | wc
bash: bogus_command: command not found
 0       0       0

bash$ echo ${PIPESTATUS[@]}
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.

$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.

```
bash$ $ ls | bogus_command | wc
bash: bogus_command: command not found
 0       0       0

bash$ echo ${PIPESTATUS[@]}
0 127 0

bash$ echo ${PIPESTATUS[@]}
0
```

$PPID

进程的$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
 6 echo
 7 echo -n "What is your favorite vegetable? "
 8 read
 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.

```
bash$ echo $SHELLOPTS
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.
```

```
fi

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

还有更加复杂的办法可以在脚本中实现定时输入. 一种办法就是建立一个定式循环, 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

```
#!/bin/bash
# timed-input.sh

# TMOUT=3    在新一些的Bash版本上也能运行的很好.


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

PrintAnswer()
{
  if [ "$answer" = TIMEOUT ]
  then
    echo $answer
  else       # 别和上边的例子弄混了.
    echo "Your favorite veggie is $answer"
    kill $!  # 不再需要后台运行的TimerOn函数了, kill了吧.
             # $! 变量是上一个在后台运行的作业的PID.
  fi

}



TimerOn()
{
  sleep $TIMELIMIT && kill -s 14 $$ &
  # 等待3秒, 然后给脚本发送一个信号.
}

Int14Vector()
{
  answer="TIMEOUT"
  PrintAnswer
  exit 14
}

trap Int14Vector 14   # 定时中断(14)会暗中给定时间限制.

echo "What is your favorite vegetable "
TimerOn
read answer
PrintAnswer


#  无可否认, 这是一个定时输入的复杂实现,
#+ 然而"read"命令的"-t"选项可以简化这个任务.
#  参考后边的"t-out.sh".

#  如果你需要一个真正优雅的写法...
#+ 建议你使用C或C++来重写这个应用,
#+ 你可以使用合适的函数库, 比如'alarm'和'setitimer'来完成这个任务.

exit 0
```

另一种选择是使用stty.

例子 9-3. 再来一个, 定时输入

```
#!/bin/bash
# timeout.sh

#  由Stephane Chazelas所编写,
#+ 本书作者做了一些修改.

INTERVAL=5                 # 超时间隔

```

```
 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.

```
tcsh% echo $LOGNAME
bozo
tcsh% echo $SHELL
/bin/tcsh
tcsh% echo $TERM
rxvt

bash$ echo $LOGNAME
bozo
bash$ echo $SHELL
/bin/tcsh
bash$ echo $TERM
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

```
#!/bin/bash
# arglist.sh
# 多使用几个参数来调用这个脚本, 比如"one two three".

E_BADARGS=65

if [ ! -n "$1" ]
then
  echo "Usage: `basename $0` argument1 argument2 etc."
  exit $E_BADARGS
fi

echo

index=1          # 起始计数.

echo "Listing args with \"\$*\":"
for arg in "$*"  # 如果"$*"不被""引用,那么将不能正常地工作.
do
  echo "Arg #$index = $arg"
  let "index+=1"
done             # $* 将所有的参数看成一个单词.
echo "Entire arg list seen as single word."

echo

index=1          # 重置计数(译者注: 从1开始).
                 # 如果你写这句会发生什么?

echo "Listing args with \"\$@\":"
for arg in "$@"
do
  echo "Arg #$index = $arg"
  let "index+=1"
done             # $@ 把每个参数都看成是单独的单词.
echo "Arg list seen as separate words."

echo

index=1          # 重置计数(译者注: 从1开始).

echo "Listing args with \$* (unquoted):"
for arg in $*
do
  echo "Arg #$index = $arg"
  let "index+=1"
done             # 未引用的$*将会把参数看成单独的单词.
echo "Arg list seen as separate words."

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.

```
#!/bin/bash
# 使用 ./scriptname 1 2 3 4 5 来调用这个脚本

echo "$@"     # 1 2 3 4 5
shift
echo "$@"     # 2 3 4 5
shift
echo "$@"     # 3 4 5

# 每次"shift"都会丢弃$1.
# "$@" 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

```
#!/bin/bash

#  内部Bash变量"$*"和"$@"的古怪行为,
#+ 都依赖于它们是否被双引号引用起来.
#  单词拆分与换行的不一致的处理.


set -- "First one" "second" "third:one" "" "Fifth: :one"
# 设置这个脚本的参数, $1, $2, 等等.

echo

echo 'IFS unchanged, using "$*"'
c=0
for i in "$*"                 # 引用起来
do echo "$((c+=1)): [$i]"    # 这行在下边每个例子中都一样.
                              # 打印参数.
done
echo ---

echo 'IFS unchanged, using $*'
c=0
for i in $*                   # 未引用
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS unchanged, using "$@"'
c=0
for i in "$@"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS unchanged, using $@'
c=0
for i in $@
do echo "$((c+=1)): [$i]"
done
echo ---

IFS=:
echo 'IFS=":", using "$*"'
c=0
for i in "$*"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", using $*'
c=0
for i in $*
do echo "$((c+=1)): [$i]"
done
echo ---

var=$*
echo 'IFS=":", using "$var" (var=$*)'
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", using $var (var=$*)'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done
echo ---

var="$*"
echo 'IFS=":", using $var (var="$*")'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", using "$var" (var="$*")'
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
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.

例子 9-8. 当**$IFS**为空时的**$***和**$@**

```
 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
 6 mecho ()       # 打印位置参数.
 7 {
 8 echo "$1,$2,$3";
 9 }
10
11
12 IFS=""         # 设置了, 但值为空.
13 set a b c      # 位置参数.
14
15 mecho "$*"     # 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.

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 & { sleep ${TIMEOUT}; eval 'kill -9 $!' &>
/dev/null; }
 2 # 强制结束一个出错程序.
 3 # 很有用, 比如用在init脚本中.
 4
 5 # 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
 3 echo $_                # /bin/bash
 4                        # 只是调用/bin/bash来运行这个脚本.
 5
 6 du >/dev/null          # 这么做命令行上将没有输出.
 7 echo $_                # du
 8
 9 ls -al >/dev/null      # 这么做命令行上将没有输出.
10 echo $_                # -al  (这是最后的参数)
11
12 :
13 echo $_                # :
```

$?

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.)

9.2. 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

${#string}

expr length $string

expr "$string" : '.*'

```
1 stringZ=abcABC123ABCabc
2
3 echo ${#stringZ}                 # 15
4 echo `expr length $stringZ`      # 15
5 echo `expr "$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: $0 <FILENAME
 6
 7 MINLEN=45        # 可能需要修改这个值.
 8 #  假定行的长度小于$MINLEN所指定的长度的时候
 9 #+ 才认为此段结束.
10
11 while read line  # 提供和输入文件一样多的行...
12 do
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)后面添加一个空行.
18   fi
19 done
20
21 exit 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.

```
stringZ=abcABC123ABCabc
echo `expr index "$stringZ" C12`            # 6
                                            # C 字符的位置.

echo `expr index "$stringZ" 1c`             # 3
# '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.

```
stringZ=abcABC123ABCabc
#       0123456789.....
#       0-based indexing.

echo ${stringZ:0}                           # abcABC123ABCabc
echo ${stringZ:1}                           # bcABC123ABCabc
echo ${stringZ:7}                           # 23ABCabc

echo ${stringZ:7:3}                         # 23A
                                            # 提取子串长度为3.



# 能不能从字符串的右边(也就是结尾)部分开始提取子串?

echo ${stringZ:-4}                          # abcABC123ABCabc
# 默认是提取整个字符串, 就象${parameter:-default}一样.
# 然而 . . .

echo ${stringZ:(-4)}                        # Cabc
echo ${stringZ: -4}                         # Cabc
# 这样, 它就可以工作了.
# 使用圆括号或者添加一个空格可以"转义"这个位置参数.

# 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.

```
echo ${*:2}           # 打印出第2个和后边所有的位置参数.
echo ${@:2}           # 同上.

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.

```
stringZ=abcABC123ABCabc
#       123456789......
#       以1开始计算.

echo `expr substr $stringZ 1 2`             # ab
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

```
```

```
#!/bin/bash
#  cvt.sh:
#  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.

#  使用"netpbm"包中的"macptopbm"程序进行转换,
#+ 这个程序主要是由Brian Henderson(bryanh@giraffe-data.com)来维护的.
#  Netpbm绝大多数Linux发行版的标准套件.

OPERATION=macptopbm
SUFFIX=pbm          # 新的文件名后缀.

if [ -n "$1" ]
then
  directory=$1      # 如果目录名作为参数传递给脚本...
else
  directory=$PWD    # 否则使用当前的工作目录.
fi

#  假定目标目录中的所有文件都是MacPaint格式的图像文件,
#+ 并且都是以".mac"作为文件名后缀.

for file in $directory/*    # 文件名匹配(filename globbing).
do
  filename=${file%.*c}      #  去掉文件名的".mac"后缀
                            #+ ('.*c' 将会匹配
                          #+ '.'和'c'之间任意字符串).
  $OPERATION $file > "$filename.$SUFFIX"
                            # 把结果重定向到新的文件中.
  rm -f $file               # 转换后删除原始文件.
  echo "$filename.$SUFFIX"  # 从stdout输出转换后文件的文件名.
done

exit 0

# 练习:
# -----
#  就像它现在的样子, 这个脚本把当前
#+ 目录下的所有文件都转换了.
#  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*各式的文件

```
#!/bin/bash
# ra2ogg.sh: 将音频流文件(*.ra)转换为ogg格式的文件.

# 使用"mplayer"媒体播放器程序:
#      http://www.mplayerhq.hu/homepage
#      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.
# 需要使用"ogg"库和"oggenc":
#      http://www.xiph.org/


OFILEPREF=${1%%ra}      # 去掉"ra"后缀.
OFILESUFF=wav           # wav文件的后缀.
OUTFILE="$OFILEPREF""$OFILESUFF"
E_NOARGS=65

if [ -z "$1" ]          # 必须要指定一个需要转换的文件名.
then
  echo "Usage: `basename $0` [filename]"
  exit $E_NOARGS
fi


################################################################################
mplayer "$1" -ao pcm:file=$OUTFILE
oggenc "$OUTFILE"  # oggenc编码后会自动加上正确的文件扩展名.

################################################################################

rm "$OUTFILE"       # 删除中介的*.wav文件.
                    # 如果你想保留这个文件的话, 可以把上边这行注释掉.

exit $?

#  注意:
#  ----
#  在网站上, 简单的在*.ram流音频文件上单击的话,
#+ 一般都只会下载真正音频流文件(就是*.ra文件)的URL.
#  你可以使用"wget"或者一些类似的工具
#+       *.ra       .
```

```
      来下载     文件本身
39
40
41 #  练习:
42 #  -----
43 #  像上面所看到的, 这个脚本只能够转换*.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
 7 getopt_simple()
 8 {
 9     echo "getopt_simple()"
10     echo "Parameters are '$*'"
11     until [ -z "$1" ]
12     do
13       echo "Processing parameter of: '$1'"
14       if [ ${1:0:1} = '/' ]
15       then
16           tmp=${1:1}               # 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*$replacement*来替换第一个匹配的*$substring*.

${string//substring/replacement}

使用*$replacement*来替换所有匹配的*$substring*.

```
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*$substring*匹配*$string*的*开头部分*, 那么就用*$replacement*来替换*$substring*.

${string/%substring/replacement}

如果*$substring*匹配*$string*的*结尾部分*, 那么就用*$replacement*来替换*$substring*.

```
1 stringZ=abcABC123ABCabc
2
3 echo ${stringZ/#abc/XYZ}           # XYZABC123ABCabc
4                                    # 用'XYZ'替换开头的'abc'.
5
6 echo ${stringZ/%abc/XYZ}           # abcABC123ABCXYZ
7                                    # 用'XYZ'替换结尾的'abc'.
```

9.2.1. 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
```

9.2.2. 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Section 9.3和expr命令列表的相关章节. 相关脚本的例子, 参见:

1. 例子 12-9
2. 例子 9-17
3. 例子 9-18
4. 例子 9-19
5. 例子 9-21

注意事项

[1] 这适用于命令行参数或函数参数.

9.3. 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.

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

```
echo ${username-`whoami`}
# 如果变量$username还没有被声明, 那么就echoe出`whoami`的结果(译者注: 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.

```
#!/bin/bash
# param-sub.sh

#  一个变量是否被声明或设置,
#+ 将会影响这个变量是否使用默认值,
#+ 即使这个变量值为空(null).

username0=
echo "username0 has been declared, but is set to null."
echo "username0 = ${username0-`whoami`}"
# 不会有输出.

echo

echo username1 has not been declared.
echo "username1 = ${username1-`whoami`}"
# 将会输出默认值.

username2=
echo "username2 has been declared, but is set to null."
echo "username2 = ${username2:-`whoami`}"
#                           ^
# 会输出, 因为:-会比-多一个条件测试.
# 可以与上边的例子比较一下.


#

# 再来一个:

variable=
# 变量已经被声明, 但是设为空值.

echo "${variable-0}"    # (没有输出)
echo "${variable:-1}"   # 1
#               ^

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

```
#!/bin/bash

#  检查一些系统环境变量.
#  这是一种可以做一些预防性保护措施的好习惯.
#  比如, 如果$USER(用户在控制台上中的名字)没有被设置的话,
#+ 那么系统就会不认你.

: ${HOSTNAME?} ${USER?} ${HOME?} ${MAIL?}
  echo
  echo "Name of the machine is $HOSTNAME."
  echo "You are $USER."
  echo "Your home directory is $HOME."
  echo "Your mail INBOX is located in $MAIL."
  echo
  echo "If you are reading this message,"
  echo "critical environmental variables have been set."
  echo
  echo

# ------------------------------------------------------

#  ${variablename?}结构
#+ 也能够检查脚本中变量的设置情况.

ThisVariable=Value-of-ThisVariable
#  注意, 顺便提一下,
#+ 这个字符串变量可能会被设置一些非法字符.
: ${ThisVariable?}
echo "Value of ThisVariable is $ThisVariable".
echo
echo


: ${ZZXy23AB?"ZZXy23AB has not been set."}
#  如果变量ZZXy23AB没有被设置的话,
#+ 那么这个脚本会打印一个错误信息, 然后结束.

# 你可以自己指定错误消息.
# : ${variablename?"ERROR MESSAGE"}


# 等价于:    dummy_variable=${ZZXy23AB?}
#             dummy_variable=${ZZXy23AB?"ZXy23AB has not been set."}
#
#             echo ${ZZXy23AB?} >/dev/null

#  使用命令"set -u"来比较这些检查变量是否被设置的方法.
#


echo "You will not see this message, because script already terminated."

HERE=0
exit $HERE   # 不会在这里退出.

# 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)

```
#!/bin/bash
# usage-message.sh

: ${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
  9 echo "These two lines echo only if command-line parameter given."
 10 echo "command line parameter = \"$1\""
 11
 12 exit 0  # 如果提供了命令行参数, 那么脚本就会在这里退出.
 13
 14 # 分别检查有命令行参数时和没有命令行参数时, 脚本的退出状态.
 15 # 如果有命令行参数, 那么"$?"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
  4 E_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.
```

下边是Manfred Schwarb给出的一个更加详细的例子:

```
  1 strip_leading_zero2 () # 去掉开头可能存在的0(也可能有多个0), 因为如果不取掉
    ,
```

```
的话
  2 {                           # Bash就会把这个值当作8进制的值来解释.
  3   shopt -s extglob          # 打开扩展的通配(globbing).
  4   local val=${1##+(0)} # 使用局部变量, 匹配最长连续的一个或多个0.
  5   shopt -u extglob          # 关闭扩展的通配(globbing).
  6   _strip_leading_zero2=${val:-0}
  7                             # 如果输入为0, 那么返回0来代替"".
  8 }
```

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                               # 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. parameter substitution operators.
  5
  6 var1=abcd12345abc6789
  7 pattern1=a*c  # *(通配符)匹配a - c之间的任意字符.
  8
  9 echo
 10 echo "var1 = $var1"           # abcd12345abc6789
 11 echo "var1 = ${var1}"         # abcd12345abc6789
 12                               # (另一种形式)
 13 echo "Number of characters in ${var1} = ${#var1}"
 14 echo
 15
 16 echo "pattern1 = $pattern1"   # a*c  (匹配'a'到'c'之间的任意字符)
 17 echo "--------------"
 18 echo '${var1#$pattern1}  =' "${var1#$pattern1}"    #         d12345abc6789
 19 # 最短的可能匹配, 去掉abcd12345abc6789的前3个字符.
 20 #                      |-|                  ^^^^^
 21 echo '${var1##$pattern1} =' "${var1##$pattern1}"   #                   6789
 22 # 最长的可能匹配, 去掉abcd12345abc6789的前12个字符
 23 #                      |----------|       ^^^^^^
 24
 25 echo; echo; echo
 26
 27 pattern2=b*9             # 匹配'b'到'9'之间的任意字符
 28 echo "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:

```
#!/bin/bash
# rfe.sh: 修改文件扩展名.
#
# 用法:                rfe old_extension new_extension
#
# 示例:
# 将指定目录中所有的*.gif文件都重命名为*.jpg,
# 用法:                rfe gif jpg


E_BADARGS=65

case $# in
  0|1)              # 竖线"|"在这里表示"或"操作.
  echo "Usage: `basename $0` old_file_suffix new_file_suffix"
  exit $E_BADARGS   # 如果只有0个或1个参数的话, 那么就退出脚本.
  ;;
esac


for filename in *.$1
# 以第一个参数为扩展名的全部文件的列表.
do
  mv $filename ${filename%$1}$2
  #  把筛选出来的文件的扩展名去掉, 因为筛选出来的文件的扩展名都是第一个参数,
  #+ 然后把第2个参数作为扩展名, 附加到这些文件的后边.
done

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

```
#!/bin/bash

var1=abcd-1234-defg
echo "var1 = $var1"

t=${var1#*-*}
echo "var1 (with everything, up to and including first - stripped
out) = $t"
#  t=${var1#*-}  也一样,
#+ 因为#匹配最短的字符串,
#+ 同时*匹配任意前缀, 包括空字符串.
# (感谢, Stephane Chazelas, 指出这点.)

t=${var1##*-*}
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, 那么       `basename $path_name`
就不能正常工作了,
32 #+ 但是上边的表达式可以.
33 # (感谢, S.C.)
34
35 t=${path_name%/*.*}
36 # 与 t=`dirname $path_name`  效果相同.
37 echo "path_name, stripped of suffixes = $t"
38 # 在某些情况下将失效, 比如 "../", "/foo////", # "foo/", "/".
39 #  删除后缀, 尤其是在basename没有后缀的情况下,
40 #+ 但是dirname可以, 不过这同时也使问题复杂化了.
41 # (感谢, S.C.)
42
43 echo
44
45 t=${path_name:11}
46 echo "$path_name, with first 11 chars stripped off = $t"
47 t=${path_name:11:5}
48 echo "$path_name, with first 11 chars stripped off, length 5 = $t"
49
50 echo
51
52 t=${path_name/bozo/clown}
53 echo "$path_name with \"bozo\" replaced  by \"clown\" = $t"
54 t=${path_name/today/}
55 echo "$path_name with \"today\" deleted = $t"
56 t=${path_name//o/O}
57 echo "$path_name with all o's capitalized = $t"
58 t=${path_name//o/}
59 echo "$path_name with all o's deleted = $t"
60
61 exit 0
```

`${var/#Pattern/Replacement}`

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").

9.4. 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
  3 func1 ()
  4 {
  5 echo This is a function.
  6 }
  7
  8 declare -f        # 列出前面定义的所有函数.
  9
 10 echo
 11
 12 declare -i var1   # var1是个整型变量.
 13 var1=2367
 14 echo "var1 declared as $var1"
 15 var1=var1+1       # 整型变量的声明并不需要使用'let'命令.
 16 echo "var1 incremented by 1 is $var1."
 17 # 尝试修改一个已经声明为整型变量的值.
 18 echo "Attempting to change var1 to floating point value, 2367.1."
 19 var1=2367.1       # 产生错误信息, 并且变量并没有被修改.
 20 echo "var1 is still $var1"
 21
 22 echo
 23
 24 declare -r var2=13.36         # 'declare'允许设置变量的属性,
 25                               #+ 同时给变量赋值.
 26 echo "var2 declared as $var2" # 试图修改只读变量的值.
 27 var2=13.37                    # 产生错误消息, 并且从脚本退出.
 28
 29 echo "var2 is still $var2"    # 将不会执行到这行.
 30
 31 exit 0                        # 脚本也不会从此处退出.
```

使用*declare*内建命令可以限制变量的*作用域*.

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

9.5. 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? 这确实是可以做到的, 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

```
#!/bin/bash
# ind-ref.sh: 间接变量引用.
# 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)

a=letter_of_alphabet   # 变量"a"的值是另一个变量的名字.
letter_of_alphabet=z

echo

# 直接引用.
echo "a = $a"          # a = letter_of_alphabet

# 间接引用.
eval a=\$$a
echo "Now a = $a"      # 现在 a = z

echo


# 现在, 让我们试试修改第二个引用的值.

t=table_cell_3
table_cell_3=24
echo "\"table_cell_3\" = $table_cell_3"                 # "table_cell_3" = 24
echo -n "dereferenced \"t\" = "; eval echo \$$t     # 解引用 "t" = 24
# 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?).
#          eval t=\$$t; echo "\"t\" = $t"

echo

t=table_cell_3
NEW_VAL=387
table_cell_3=$NEW_VAL
echo "Changing value of \"table_cell_3\" to $NEW_VAL."
echo "\"table_cell_3\" now $table_cell_3"
echo -n "dereferenced \"t\" now "; eval echo \$$t
# "eval" 带有两个参数 "echo" 和 "\$$t" (与$table_cell_3等价)

echo

# (感谢, Stephane Chazelas, 澄清了上边语句的行为.)


# 另一个方法是使用${!t}符号, 见"Bash, 版本2"小节的讨论.
# 也请参考 ex78.sh.

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.

```
#!/bin/bash


# ------------------------------------------------------------
# 这部分内容可以用单独文件通过使用"source"命令来单独加载.
isdnMyProviderRemoteNet=172.16.0.100
isdnYourProviderRemoteNet=10.0.0.10
```

```
isdnOnlineService="MyProvider"
# ------------------------------------------------------
remoteNet=$(eval "echo \$$(echo isdn${isdnOnlineService}RemoteNet)")
remoteNet=$(eval "echo \$$(echo isdnMyProviderRemoteNet)")
remoteNet=$(eval "echo \$isdnMyProviderRemoteNet")
remoteNet=$(eval "echo $isdnMyProviderRemoteNet")

echo "$remoteNet"    # 172.16.0.100

# ============================================================

#  能够做得更好.

#  注意下面的脚本, 给出了变量getSparc,
#+ 但是没有变量getIa64:

chkMirrorArchs () {
  arch="$1";
  if [ "$(eval "echo \${$(echo get$(echo -ne $arch |
       sed 's/^\(.\).*/\1/g' | tr 'a-z' 'A-Z'; echo $arch |
       sed 's/^.\(.*\)/\1/g')):-false}")" = true ]
  then
     return 0;
  else
     return 1;
  fi;
}

getSparc="true"
unset getIa64
chkMirrorArchs sparc
echo $?        # 0
               # True

chkMirrorArchs Ia64
echo $?        # 1
               # False

# 注意:
# -----
# 变量名中由替换命令产生的部分被准确地生成了.
# chkMirrorArchs函数的参数全都是小写字母.
# 新产生的变量名由两部分组成: "get"和"Sparc" . . .
# (译者注: 此处是将chkMirrorArchs函数参数的第一个字母转为大写, 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***

```
#!/bin/bash

#  这是"求文件中指定列的总和"脚本的另一个版本,
#+ 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)的所有数字的和.
#  这个脚本使用了间接引用.

ARGS=2
E_WRONGARGS=65

if [ $# -ne "$ARGS" ] # 检查命令行参数的个数是否合适.
then
   echo "Usage: `basename $0` filename column-number"
   exit $E_WRONGARGS
fi

filename=$1
column_number=$2

#===== 在这一行上边的部分, 与原始脚本是相同的 =====#


# 多行的awk脚本的调用方法为: awk ' ..... '


# awk脚本开始.
# ---------------------------------------------
awk "

{ total += \$${column_number} # 间接引用
}
END {
     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
```

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.

9.6. $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

```
#!/bin/bash

# 每次调用$RANDOM都会返回不同的随机整数.
# 一般范围为: 0 - 32767 (有符号的16-bit整数).

MAXCOUNT=10
count=1

echo
echo "$MAXCOUNT random numbers:"
echo "-----------------"
while [ "$count" -le $MAXCOUNT ]      # 产生10 ($MAXCOUNT)个随机整数.
do
  number=$RANDOM
  echo $number
  let "count += 1"  # 增加计数.
done
echo "-----------------"

# 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.(译者注: 事实上, 这不是一个非常好的办法. 理由见man 3 rand)
# 取模操作会返回除法的余数.

RANGE=500

echo

number=$RANDOM
let "number %= $RANGE"
#            ^^
echo "Random number less than $RANGE  ---  $number"

echo



#  如果你需要产生一个大于某个下限的随机整数.
#+ 那么建立一个test循环来丢弃所有小于此下限值的整数.

FLOOR=200

number=0   #初始化
while [ "$number" -le $FLOOR ]
do
  number=$RANDOM
done
echo "Random number greater than $FLOOR ---  $number"
echo


   # 让我们对上边的循环尝试一个小改动, 如下:
   #        let "number = $RANDOM + $FLOOR"
   # 这将不再需要那个while循环, 并且能够运行的更快.
   # 但是, 这可能会产生一个问题, 思考一下是什么问题?


# 结合上边两个例子, 来在指定的上下限之间来产生随机数.
number=0   #initialize
while [ "$number" -le $FLOOR ]
do
  number=$RANDOM
  let "number %= $RANGE"  # 让$number依比例落在$RANGE的范围内.
done
echo "Random number between $FLOOR and $RANGE ---  $number"
echo


```

```
 67
 68 # 产生二元值, 就是, "true" 或 "false" 两个值.
 69 BINARY=2
 70 T=1
 71 number=$RANDOM
 72
 73 let "number %= $BINARY"
 74 #  注意 let "number >>= 14"     将会给出一个更好的随机分配. #(译者注: 正如man页中提到的, 更高位的随机分布更加平均)
 75 #+ (右移14位将把所有的位全部清空, 除了第15位, 因为有符号, 第16位是符号位). #取模操作使用低位来产生随机数会相对不平均)
 76 if [ "$number" -eq $T ]
 77 then
 78   echo "TRUE"
 79 else
 80   echo "FALSE"
 81 fi
 82
 83 echo
 84
 85
 86 # 抛骰子.
 87 SPOTS=6   # 模6给出的范围是0 - 5.
 88           # 加1会得到期望的范围1 - 6.
 89           # 感谢, Paul Marcel Coelho Aragao, 对此进行的简化.
 90 die1=0
 91 die2=0
 92 # 是否让SPOTS=7会比加1更好呢? 解释行或者不行的原因?
 93
 94 # 每次抛骰子, 都会给出均等的机会.
 95
 96     let "die1 = $RANDOM % $SPOTS +1" # 抛第一次.
 97     let "die2 = $RANDOM % $SPOTS +1" # 抛第二次.
 98     #  上边的算术操作中, 哪个具有更高的优先级呢 --
 99     #+ 模(%) 还是加法操作(+)?
100
101
102 let "throw = $die1 + $die2"
103 echo "Throw of the dice = $throw"
104 echo
105
106
107 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
  9 Suites="Clubs
 10 Diamonds
 11 Hearts
 12 Spades"
 13
 14 Denominations="2
 15 3
 16 4
 17 5
 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
```

```
echo -n "${denomination[$((RANDOM%num_denominations))]} of "
echo ${suite[$((RANDOM%num_suites))]}


# $bozo sh pick-cards.sh
# Jack of Clubs


# 感谢, "jipe," 指出$RANDOM的这个用法.
exit 0
```

---

*Jipe*展示了一套技巧来在一个指定范围内产生随机数.

```
#  在6 到 30之间产生随机数.
   rnumber=$((RANDOM%25+6))

#  还是在6 - 30之间产生随机数,
#+ 但是这个数还必须能够被3整除.
   rnumber=$(((RANDOM%30/3+1)*3))

#  注意, 并不是在所有情况下都能正确运行.
#  如果$RANDOM返回0, 那么就会失败.

#  Frank Wang 建议用下边的方法:
   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书.

```
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

```
#!/bin/bash
# random-between.sh
# 产生两个指定值之间的随机数.
# 由Bill Gradwohl编写, 本书作者做了一些修改.
# 脚本作者允许在这里使用.


randomBetween() {
   #  在$min和$max之间,
   #+ 产生一个正的或负的随机数.
   #+ 并且可以被$divisibleBy所整除.
   #  给出一个合理的随机分配的返回值.
   #
   #  Bill Gradwohl - Oct 1, 2003

   syntax() {
   # 在函数中内嵌函数
      echo
      echo    "Syntax: randomBetween [min] [max] [multiple]"
      echo
      echo    "Expects up to 3 passed parameters, but all are completely
optional."
      echo    "min is the minimum value"
      echo    "max is the maximum value"
      echo    "multiple specifies that the answer must be a multiple of this
value."
      echo    "    i.e. answer must be evenly divisible by this number."
      echo
      echo    "If any value is missing, defaults area supplied as: 0 32767 1"
      echo    "Successful completion returns 0, unsuccessful completion returns"
      echo    "function syntax and 1."
      echo    "The answer is returned in the global variable randomBetweenAnswer"
      echo    "Negative values for any passed parameter are handled correctly."
   }

   local min=${1:-0}
   local max=${2:-32767}
   local divisibleBy=${3:-1}
   # 默认值分配, 用来处理没有参数传递进来的情况.

   local x
   local spread

   # 确认divisibleBy是正值.
   [ ${divisibleBy} -lt 0 ] && divisibleBy=$((0-divisibleBy))

   # 完整性检查.
```

```
   if [ $# -gt 3 -o ${divisibleBy} -eq 0 -o  ${min} -eq ${max} ]; then
      syntax
      return 1
   fi

   # 查看min和max是否颠倒了.
   if [ ${min} -gt ${max} ]; then
      # 交换它们.
      x=${min}
      min=${max}
      max=${x}
   fi

   #  如果min自己并不能够被$divisibleBy所整除,
   #+ 那么就调整max的值, 使其能够被$divisibleBy所整除, 前提是不能放大范围.
   if [ $((min/divisibleBy*divisibleBy)) -ne ${min} ]; then
      if [ ${min} -lt 0 ]; then
         min=$((min/divisibleBy*divisibleBy))
      else
         min=$((((min/divisibleBy)+1)*divisibleBy))
      fi
   fi

   #  如果min自己并不能够被$divisibleBy所整除,
   #+ 那么就调整max的值, 使其能够被$divisibleBy所整除, 前提是不能放大范围.
   if [ $((max/divisibleBy*divisibleBy)) -ne ${max} ]; then
      if [ ${max} -lt 0 ]; then
         max=$((((max/divisibleBy)-1)*divisibleBy))
      else
         max=$((max/divisibleBy*divisibleBy))
      fi
   fi

   #  ----------------------------------------------------------------------
   #  现在, 来做点真正的工作.

   #  注意, 为了得到对于端点来说合适的分配,
   #+ 随机值的范围不得不落在
   #+ 0 和 abs(max-min)+divisibleBy 之间, 而不是 abs(max-min)+1.

   #  对于端点来说,
   #+ 这个少量的增加将会产生合适的分配.

   #  如果修改这个公式, 使用 abs(max-min)+1 来代替 abs(max-min)+divisibleBy的话,
   #+ 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
   #+ 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,
   #+ 因为你才加1而已, 这比正常的公式下所产生的几率要小的多(正常为加divisibleBy).
   #  ----------------------------------------------------------------------

   spread=$((max-min))
   [ ${spread} -lt 0 ] && spread=$((0-spread))
   let spread+=divisibleBy
   randomBetweenAnswer=$(((RANDOM%spread)/divisibleBy*divisibleBy+min))

   return 0

   #  然而, Paulo Marcel Coelho Aragao 指出
   #+ 当 $max 和 $min 不能够被$divisibleBy所整除时,
   #+ 这个公式将会失败.
   #
   #  他建议使用如下公式:
   #    rnumber = $(((RANDOM%(max-min+1)+min)/divisibleBy*divisibleBy))

}

# 让我们测试一下这个函数.
min=-14
max=20
divisibleBy=3


#  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,
#+ 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, 那么我们一定会得到至少一次出现机会.

declare -a answer
minimum=${min}
maximum=${max}
   if [ $((minimum/divisibleBy*divisibleBy)) -ne ${minimum} ]; then
      if [ ${minimum} -lt 0 ]; then
         minimum=$((minimum/divisibleBy*divisibleBy))
      else
         minimum=$((((minimum/divisibleBy)+1)*divisibleBy))
      fi
   fi

```

```
131
132 132     #  如果max本身并不能够被$divisibleBy整除,
133 133     #+ 那么就调整max的值, 使其能够被$divisibleBy整除, 前提是不能放大范围.
134
135     if [ $((maximum/divisibleBy*divisibleBy)) -ne ${maximum} ]; then
136        if [ ${maximum} -lt 0 ]; then
137           maximum=$((((maximum/divisibleBy)-1)*divisibleBy))
138        else
139           maximum=$((maximum/divisibleBy*divisibleBy))
140        fi
141     fi
142
143
144 #  我们需要产生一个下标全部为正的数组.
145 #+ 所以我们需要一个displacement,
146 #+ 这样就可以保证结果都为正.
147
148 displacement=$((0-minimum))
149 for ((i=${minimum}; i<=${maximum}; i+=divisibleBy)); do
150    answer[i+displacement]=0
151 done
152
153
154 # 现在, 让我们循环足够多的次数, 来得到我们想要的答案.
155 loopIt=1000    #  脚本作者建议循环 100000 次,
156                #+ 但是这需要的时间太长了.
157
158 for ((i=0; i<${loopIt}; ++i)); do
159
160    #  注意, 我们在这里调用randomBetweenAnswer函数时, 估计将min和max颠倒顺序.
161    #+ 这是为了测试在这种情况下, 此函数是否还能正确的运行.
162
163    randomBetween ${max} ${min} ${divisibleBy}
164
165    # 如果答案不是我们所期望的, 就报错.
166    [ ${randomBetweenAnswer} -lt ${min} -o ${randomBetweenAnswer} -gt ${max} ] &&
echo MIN or MAX error - ${randomBetweenAnswer}!
167    [ $((randomBetweenAnswer%${divisibleBy})) -ne 0 ] && echo DIVISIBLE BY error -
${randomBetweenAnswer}!
168
169    # 将统计值保存到answer中.
170
answer[randomBetweenAnswer+displacement]=$((answer[randomBetweenAnswer+displacement]+1))
171 done
172
173
174
175 # 让我们来察看一下结果.
176
177 for ((i=${minimum}; i<=${maximum}; i+=divisibleBy)); do
178    [ ${answer[i+displacement]} -eq 0 ] && echo "We never got an answer of $i." ||
echo "${i} occurred ${answer[i+displacement]} times."
179 done
180
181
182 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
 4 RANDOM=$$        # 使用脚本的进程ID来作为随机数的种子.
 5
 6 PIPS=6           # 一个骰子有6个面.
 7 MAXTHROWS=600    # 如果你没别的事做, 可以增加这个数值.
 8 throw=0          # 抛骰子的次数.
 9
10 ones=0           #  必须把所有的count都初始化为0,
11 twos=0           #+ 因为未初始化的变量为null, 不是0.
12 threes=0
13 fours=0
14 fives=0
15 sixes=0
16
17 print_result ()
18 {
```

```
19 echo
20 echo "ones =   $ones"
21 echo "twos =   $twos"
22 echo "threes = $threes"
23 echo "fours =  $fours"
24 echo "fives =  $fives"
25 echo "sixes =  $sixes"
26 echo
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
 5 AWKSCRIPT=' { 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
10 echo -n "Random number between 0 and 1 = "
11
12 echo | awk "$AWKSCRIPT"
13 # 如果你省去'echo', 会怎样?
14
15 exit 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.

9.7. 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

```
#!/bin/bash
# 使用((...))结构操作一个变量, C语言风格的变量操作.


echo

(( a = 23 ))  # C语言风格的变量赋值, "="两边允许有空格.
echo "a (initial value) = $a"

(( a++ ))     # C语言风格的后置自加.
echo "a (after a++) = $a"

(( a-- ))     # C语言风格的后置自减.
echo "a (after a--) = $a"


(( ++a ))     # C语言风格的前置自加.
echo "a (after ++a) = $a"

(( --a ))     # C语言风格的前置自减.
echo "a (after --a) = $a"

echo

########################################################
#  注意: 就像在C语言中一样, 前置或后置自减操作
#+ 会产生一些不同的副作用.

n=1; let --n && echo "True" || echo "False"  # False
n=1; let n-- && echo "True" || echo "False"  # True

#  感谢, Jeroen Domburg.
########################################################

echo

(( t = a<45?7:11 ))   # C语言风格的三元操作.
echo "If a < 45, then t = 7, else t = 11."
echo "t = $t "        # Yes!

echo


# ------------
# 复活节彩蛋!
# ------------
#  Chet Ramey显然偷偷摸摸的将一些未公开的C语言风格的结构
#+ 引入到了Bash中 (事实上是从ksh中引入的, 这更接近些).
#  在Bash的文档中, Ramey将((...))称为shell算术运算,
#+ 但是它所能做的远远不止于此.
#  不好意思, Chet, 现在秘密被公开了.

# 你也可以参考一些 "for" 和 "while" 循环中使用((...))结构的例子.

# 这些只能够在Bash 2.04或更高版本的Bash上才能运行.

exit 0
```

参见例子 10-12.

10. 循环与分支

目录
10.1. 循环
10.2. 嵌套循环
10.3. 循环控制
10.4. 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.

10.1. 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循环

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.

```
for arg in [list]
do
兼ommand(s)...
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
  4 for planet in Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune
Pluto
  5 do
  6   echo $planet  # 每个行星都被单独打印在一行上.
  7 done
  8
  9 echo
 10
 11 for planet in "Mercury Venus Earth Mars Jupiter Saturn Uranus
Neptune Pluto"
 12 # 所有的行星名称都打印在同一行上.
 13 # 整个'list'都被双引号封成了一个变量.
 14 do
 15   echo $planet
 16 done
 17
 18 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
  4 echo
  5
  6 for 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.
  9 do
 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.
18 done
19
20 echo; echo
21
22 for file in [jx]*
23 do
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\"".
26 done
27
28 echo
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.

例子 10-6. 使用命令替换来产生for循环的[list]

```
 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
 4 PASSWORD_FILE=/etc/passwd
 5 n=1           # User number
 6
 7 for 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
 5 directory=/usr/bin/
 6 fstring="Free Software Foundation"  # 查看哪个文件中包含FSF.
 7
```

```
 8 for file in $( find $directory -type f -name '*' | sort )
 9 do
10   strings -f $file | grep "$fstring" | sed -e "s%$directory%%"
11   #  在"sed"表达式中,
12   #+ 我们必须替换掉正常的替换分隔符"/",
13   #+ 因为"/"碰巧是我们需要过滤的字符串之一.
14   #  如果不用"%"代替"/"作为分隔符,那么这个操作将失败,并给出一个错误消息.(试一
试).
15 done
16
17 exit 0
18
19 #  练习 (很简单):
20 #  ---------------
21 #  转换这个脚本, 用命令行参数
22 #+ 代替内部用的$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 # ARGS=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 $directory -type l )用""引用起来的话,
32 #+ 那么将会把一个带有空白部分的文件名拆分成以空白分隔的两部分(文件名允许有空白).
33 #  即使这里只会取出每个参数的第一个域.
34
35 exit 0
36
37
38 # Jean Helou建议采用下边的方法:
39
40 echo "symbolic links in directory \"$directory\""
41 # 当前IFS的备份. 要小心使用这个值.
42 OLDIFS=$IFS
43 IFS=:
44
45 for file in $(find $directory -type l -printf "%p$IFS")
46 do     #                              ^^^^^^^^^^^^^^^^
47        echo "$file"
48        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
 4 OUTFILE=symlinks.list                                  # 保存符号链接文件名的文件
 5
 6 directory=${1-`pwd`}
 7 #  如果没有其他特殊的指定,
 8 #+ 默认为当前工作目录.
 9
10
11 echo "symbolic links in directory \"$directory\"" > "$OUTFILE"
12 echo "----------------------------" >> "$OUTFILE"
13
14 for file in "$( find $directory -type l )"    # -type l = 符号链接
15 do
16   echo "$file"
17 done | sort >> "$OUTFILE"                     # 循环的stdout
18 #            ^^^^^^^^^^^^^                     重定向到一个文件中.
19
20 exit 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
 4 echo
 5
 6 # 标准语法.
 7 for a in 1 2 3 4 5 6 7 8 9 10
 8 do
 9   echo -n "$a "
10 done
11
12 echo; echo
13
14 # +=========================================+
15
16 # 现在, 让我们用C风格语法来做相同的事情.
17
18 LIMIT=10
19
20 for ((a=1; a <= LIMIT ; a++))  # 双圆括号, 并且"LIMIT"变量前面没有"$".
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
```

```
 4 EXPECTED_ARGS=2
 5 E_BADARGS=65
 6
 7 if [ $# -ne $EXPECTED_ARGS ]
 8 # 检查命令行参数的个数是否正确.
 9 then
10    echo "Usage: `basename $0` phone# text-file"
11    exit $E_BADARGS
12 fi
13
14
15 if [ ! -f "$2" ]
16 then
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.

```
while [condition]
do
command...
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
 3 var0=0
 4 LIMIT=10
 5
 6 while [ "$var0" -lt "$LIMIT" ]
 7 do
 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
 3 echo
 4                                  # 等价于:
 5 while [ "$var1" != "end" ]       # while test "$var1" != "end"
 6 do
 7   echo "Input variable #1 (end to exit) "
 8   read var1                     # 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
13 done
14
15 exit 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
 3 var1=unset
 4 previous=$var1
 5
 6 while 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.
12 do
13 echo "Input variable #1 (end to exit) "
14   read var1
15   echo "variable #1 = $var1"
16 done
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
 4 LIMIT=10
 5 a=1
 6
 7 while [ "$a" -le $LIMIT ]
 8 do
 9   echo -n "$a "
10   let "a+=1"
11 done            # 到目前为止都没有什么令人惊奇的地方.
12
13 echo; 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
22 while (( a <= LIMIT ))   # 双圆括号, 变量前边没有"$".
23 do
24   echo -n "$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.
```

```
28 done
29
30 echo
31
32 # 现在, C程序员可以在Bash中找到回家的感觉了吧.
33
34 exit 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).

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

10.2. 嵌套循环

*嵌套循环*就是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

```
#!/bin/bash
# nested-loop.sh: 嵌套的"for"循环.

outer=1             # 设置外部循环计数.

# 开始外部循环.
for a in 1 2 3 4 5
do
  echo "Pass $outer in outer loop."
  echo "---------------------"
  inner=1           # 重置内部循环计数.

  # ===========================================
  # 开始内部循环.
  for b in 1 2 3 4 5
  do
    echo "Pass $inner in inner loop."
    let "inner+=1"  # 增加内部循环计数.
  done
  # 内部循环结束.
  # ===========================================

  let "outer+=1"    # 增加外部循环的计数.
  echo              # 每次外部循环之间的间隔.
done
# 外部循环结束.

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.

10.3. 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

```
#!/bin/bash

LIMIT=19  # 上限

echo
echo "Printing Numbers 1 through 20 (but not 3 and 11)."

a=0

while [ $a -le "$LIMIT" ]
do
 a=$(($a+1))

 if [ "$a" -eq 3 ] || [ "$a" -eq 11 ]  # 除了3和11.
 then
   continue      # 跳过本次循环剩余的语句.
 fi

 echo -n "$a "   # 在$a等于3和11的时候, 这句将不会执行.
done

# 练习:
# 为什么循环会打印出20?

echo; echo

echo Printing Numbers 1 through 20, but something happens after 2.

##############################################################

# 同样的循环, 但是用'break'来代替'continue'.

a=0

while [ "$a" -le "$LIMIT" ]
do
 a=$(($a+1))

 if [ "$a" -gt 2 ]
 then
   break  # 将会跳出整个循环.
 fi

 echo -n "$a "
done

echo; echo; echo

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
  6 for outerloop in 1 2 3 4 5
  7 do
  8   echo -n "Group $outerloop:   "
  9
 10   # ----------------------------------------------------------
 11   for innerloop in 1 2 3 4 5
 12   do
 13     echo -n "$innerloop "
 14
 15     if [ "$innerloop" -eq 3 ]
 16     then
 17       break  # 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
 24 done
 25
 26 echo
 27
 28 exit 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, 因此我很想避免这个问题].
 48 #+ 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.
 49 #+ 当然, 还有一种办法, 也可以不使用`continue 2',
 50 #+ 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,
 51 #+ (而我们应该马上来找到下一个要运行的任务)
 52 #+ (在演示的情况中, 检查新任务前我们终止或睡眠了很长一段时间).
 53 #+
```

---

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.

10.4. 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.

```
case "$variable" in

  $condition1" )
  TT CLASS="REPLACEABLE" >command...
  ;

  $condition2" )
  TT CLASS="REPLACEABLE" >command...
  ;

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
 4 echo; echo "Hit a key, then hit return."
 5 read Keypress
 6
 7 case "$Keypress" in
 8   [[:lower:]]   ) echo "Lowercase letter";;
 9   [[:upper:]]   ) echo "Uppercase letter";;
10   [0-9]         ) echo "Digit";;
11   *             ) echo "Punctuation, whitespace, or other";;
12 esac      #  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
 5 clear # 清屏.
 6
 7 echo "           Contact List"
 8 echo "           ------- ----"
 9 echo "Choose one of the following persons:"
10 echo
11 echo "[E]vans, Roland"
12 echo "[J]ones, Mildred"
13 echo "[S]mith, Julie"
14 echo "[Z]ane, Morris"
15 echo
16
17 read person
18
19 case "$person" in
20 # 注意, 变量是被""引用的.
21
22   "E" | "e" )
23   # 接受大写或者小写输入.
24   echo
25   echo "Roland Evans"
26   echo "4321 Floppy Dr."
27   echo "Hardscrabble, CO 80753"
28   echo "(303) 734-9874"
29   echo "(303) 734-9892 fax"
30   echo "revans@zzy.net"
31   echo "Business partner & old friend"
32   ;;
33 # 注意, 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
 4 while [ $# -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
 4 case $( 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
```

```
c=three
d=two


match_string $a     # 参数个数错误.
echo $?              # 91

match_string $a $b  # 不匹配
echo $?              # 90

match_string $b $d  # 匹配
echo $?              # 0


exit 0
```

例子 10-28. 检查输入字符是否为字母

```
#!/bin/bash
# isalpha.sh: 使用"case"结构来过滤字符串.

SUCCESS=0
FAILURE=-1

isalpha ()  # 检查输入的 *第一个字符* 是不是字母表上的字符.
{
if [ -z "$1" ]                 # 没有参数传进来?
then
  return $FAILURE
fi

case "$1" in
[a-zA-Z]*) return $SUCCESS;;  # 以一个字母开头?
*        ) return $FAILURE;;
esac
}             # 同C语言的"isalpha ()"函数比较一下.


isalpha2 ()   # 测试 *整个字符串* 是否都是字母表上的字符.
{
  [ $# -eq 1 ] || return $FAILURE

  case $1 in
  *[!a-zA-Z]*|"") return $FAILURE;;
                *) return $SUCCESS;;
  esac
}

isdigit ()    # 测试 *整个字符串* 是否都是数字.
{             # 换句话说, 就是测试一下是否是整数变量.
  [ $# -eq 1 ] || return $FAILURE

  case $1 in
  *[!0-9]*|"") return $FAILURE;;
             *) return $SUCCESS;;
  esac
}



check_var ()  # 测试isalpha().
{
if isalpha "$@"
then
  echo "\"$*\" begins with an alpha character."
  if isalpha2 "$@"
  then        # 不需要测试第一个字符是否是non-alpha.
    echo "\"$*\" contains only alpha characters."
  else
    echo "\"$*\" contains at least one non-alpha character."
  fi
else
  echo "\"$*\" begins with a non-alpha character."
              # 如果没有参数传递进来, 也是"non-alpha".
fi

echo

}

digit_check ()  # 测试isdigit().
{
```

```
 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=H3llo
 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 #  暗示: 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.

```
select variable [in list]
do
  TT CLASS="REPLACEABLE" >command...
燾reak
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
  3 PS3='Choose your favorite vegetable: ' # 设置提示符字串.
  4
  5 echo
  6
  7 select vegetable in "beans" "carrots" "potatoes" "onions"
"rutabagas"
  8 do
  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?
 14 done
 15
 16 exit 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
```

参考例子 34-3.

---

11. 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，一部分原因是因为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 #+ 是否所有产生出来的进程都会被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
3 echo "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) 一般都会将 \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.

```
bash$ ls -l /usr/share/apps/kjezz/sounds
-rw-r--r--    1 root     root         1407 Nov  7  2000 reflect.au
 -rw-r--r--    1 root     root          362 Nov  7  2000 seconds.au



bash$ echo `ls -l /usr/share/apps/kjezz/sounds`
total 40 -rw-r--r-- 1 root root 716 Nov 7 2000 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.

```
bash$ type -a echo
echo is a shell builtin
 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.

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
  4 echo -n "Enter the value of variable 'var1': "
  5 # -n 选项, 阻止换行.
  6
  7 read var1
  8 # 注意: 在var1前面没有'$', 因为变量正在被设置.
  9
 10 echo "var1 = $var1"
 11
 12
 13 echo
 14
 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' (separated
by a space or tab): "
 17 read var2 var3
 18 echo "var2 = $var2      var3 = $var3"
 19 # 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).
 20
 21 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
  4 echo
  5
  6 # ------------------------- #
  7 echo -n "Enter a value: "
  8 read var
  9 echo "\"var\" = "$var""
 10 # 到这里为止, 都与期望的一样.
 11 # ------------------------- #
 12
 13 echo
 14
 15 # ------------------------------------------------------------------
-- #
```

```
 16 echo -n "Enter another value: "
 17 read            #  没有变量分配给'read'命令, 所以...
 18                  #+ 输入将分配给默认变量, $REPLY.
 19 var="$REPLY"
 20 echo "\"var\" = "$var""
 21 # 这部分代码和上边的代码等价.
 22 # ------------------------------------------------------------------------
-- #
 23
 24 echo
 25
 26 exit 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
  3 echo
  4
  5 echo "Enter a string terminated by a \\, then press <ENTER>."
  6 echo "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.
 11
```

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, 可能还是需要按回车. (译者注: 因为一般按键是一个
字符)
 20 read -n3 key                         # 读取3个字符.
 21
 22 echo -n "$key" | grep "$arrowup"   # 检查输入字符是否匹配.
 23 if [ "$?" -eq $SUCCESS ]
 24 then
 25   echo "Up-arrow key pressed."
 26   exit $SUCCESS
 27 fi
 28
 29 echo -n "$key" | grep "$arrowdown"
 30 if [ "$?" -eq $SUCCESS ]
 31 then
 32   echo "Down-arrow key pressed."
 33   exit $SUCCESS
 34 fi
 35
 36 echo -n "$key" | grep "$arrowrt"
 37 if [ "$?" -eq $SUCCESS ]
 38 then
 39   echo "Right-arrow key pressed."
 40   exit $SUCCESS
 41 fi
 42
 43 echo -n "$key" | grep "$arrowleft"
 44 if [ "$?" -eq $SUCCESS ]
 45 then
 46   echo "Left-arrow key pressed."
 47   exit $SUCCESS
 48 fi
 49
 50 echo -n "$key" | grep "$insert"
 51 if [ "$?" -eq $SUCCESS ]
 52 then
 53   echo "\"Insert\" key pressed."
 54   exit $SUCCESS
 55 fi
 56
 57 echo -n "$key" | grep "$delete"
 58 if [ "$?" -eq $SUCCESS ]
 59 then
 60   echo "\"Delete\" key pressed."
 61   exit $SUCCESS
 62 fi
 63
 64
 65 echo " Some other key pressed."
 66
 67 exit $OTHER
 68
 69 #  练习:
 70 #  -----
 71 #  1) 使用'case'结构来代替'if'结构,
 72 #+     这样可以简化这个脚本.
 73 #  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
 3 read var1 <data-file
 4 echo "var1 = $var1"
 5 # var1将会把"data-file"的第一行的全部内容都为它的值.
 6
 7 read var2 var3 <data-file
 8 echo "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Bjœrn 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 #           Bjon Eriksson       .
```

```
      这个例子是由                所编写的
 4
 5 last="(null)"
 6 cat $0 |
 7 while read line
 8 do
 9     echo "{$line}"
10     last=$line
11 done
12 printf "\nAll done, last:$last\n"
13
14 exit 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
33 All 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. .

```
bash$ cd //
bash$ pwd
//
```

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
 3 echo
 4
 5 let a=11             # 与 'a=11' 相同
 6 let 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.)
 8 echo "11 + 5 = $a"  # 16
 9
10 let "a <<= 3"        # 等价于 let "a = a << 3"
11 echo "\"\$a\" (=16) left-shifted 3 places = $a"
12                      # 128
13
14 let "a /= 4"         # 等价于 let "a = a / 4"
15 echo "128 / 4 = $a" # 32
16
17 let "a -= 5"         # 等价于 let "a = a - 5"
18 echo "32 - 5 = $a"  # 27
19
20 let "a *=  10"       # 等价于 let "a = a * 10"
21 echo "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). 任何被包含在表达示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了.

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

例子 11-11. 展示eval命令的效果

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

```
#!/bin/bash
# 结束ppp进程来强制登出log-off.

# 本脚本应该以root用户的身份来运行.

killppp="eval kill -9 `ps ax | awk '/ppp/ { print $1 }'`"
#                       -------- ppp的进程ID -------

$killppp                    # 这个变量现在成为了一个命令.


# 下边的命令必须以root用户的身份来运行.

chmod 666 /dev/ttyS3       # 恢复读写权限,否则什么?
#  因为在ppp上执行一个SIGKILL将会修改串口的权限,
#+ 我们把权限恢复到之前的状态.

rm /var/lock/LCK..ttyS3    # 删除串口琐文件.为什么?

exit 0

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

例子 11-13. 另一个"rot13"版本

```
#!/bin/bash
# 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).
# 与脚本"rot13.sh" 比较一下.

setvar_rot_13()               # "rot13" 函数
{
  local varname=$1 varvalue=$2
  eval $varname='$(echo "$varvalue" | tr a-z n-za-m)'
}


setvar_rot_13 var "foobar"    # 将 "foobar" 传递到 rot13函数中.
echo $var                      # sbbone

setvar_rot_13 var "$var"      # 传递 "sbbone" 到rot13函数中.
                              # 又变成了原始值.
echo $var                      # foobar

# 这个例子是Segebart Chazelas编写的.
# 作者又修改了一下.

exit 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

```
In the Perl script "test.pl":
        ...
        my $WEBROOT = <WEBROOT_PATH>;
        ...

To force variable substitution try:
        $export WEBROOT_PATH=/usr/local/webroot
        $sed 's/<WEBROOT_PATH>/$WEBROOT_PATH/' < test.pl > out

But this just gives:
        my $WEBROOT = $WEBROOT_PATH;

However:
        $export WEBROOT_PATH=/usr/local/webroot
        $eval sed 's%\<WEBROOT_PATH\>%$WEBROOT_PATH%' < test.pl > out
#        ====

```

```
That works fine, and gives the expected substitution:
        my $WEBROOT = /usr/local/webroot;


### Paulo Marcel Coelho Aragao校正了这个原始例子.
```

---

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，如果命令中包含有`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

```
#!/bin/bash

# script "set-test"

# 使用3个命令行参数来调用这个脚本,
# 比如, "./set-test one two three".

echo
echo "Positional parameters before  set \`uname -a\` :"
echo "Command-line argument #1 = $1"
echo "Command-line argument #2 = $2"
echo "Command-line argument #3 = $3"


set `uname -a` # 把`uname -a`的命令输出设置
               # 为新的位置参数.

echo $_        # unknown(译者注: 这要看你的uname -a输出了,这句打印出的就是输出的最后一个单词.)
# 在脚本中设置标志.

echo "Positional parameters after  set \`uname -a\` :"
# $1, $2, $3, 等等. 这些位置参数将被重新初始化为`uname -a`的结果
echo "Field #1 of 'uname -a' = $1"
echo "Field #2 of 'uname -a' = $2"
echo "Field #3 of 'uname -a' = $3"
echo ---
echo $_        # ---
echo

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

```
#!/bin/bash
# revposparams.sh: 反转位置参数.
# 本脚本由Dan Jacobson所编写, 本书作者做了一些格式上的修正.


set a\ b c d\ e;
#     ^      ^     转义的空格
#       ^ ^        未转义的空格
OIFS=$IFS; IFS=:;
#          ^   保存旧的IFS, 然后设置新的IFS.

echo

until [ $# -eq 0 ]
do          #      步进位置参数.
  echo "### k0 = "$k""     # 步进之前
  k=$1:$k;  #      将每个位置参数都附加在循环变量的后边.
#     ^
```

```
 19   echo "### k = "$k""       # 步进之后
 20   echo
 21   shift;
 22 done
 23
 24 set $k  #  设置一个新的位置参数.
 25 echo -
 26 echo $# #  察看位置参数的个数.
 27 echo -
 28 echo
 29
 30 for i   #  省略 "in list" 结构,
 31         #+ 为位置参数设置变量 -- i --.
 32 do
 33   echo $i  # 显示新的位置参数.
 34 done
 35
 36 IFS=$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 #+ 才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.

```
bash$ set
AUTHORCOPY=/home/bozo/posts
 BASH=/bin/bash
 BASH_VERSION=$'2.05.8(1)-release'
 ...
 XAUTHORITY=/home/bozo/.Xauthority
 _=/etc/bashrc
 variable22=abc
 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*unsets*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.

```
bash$ unset PATH

bash$ echo $PATH


bash$
```

例子 11-18. "Unsett"一个变量

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

```
10 E_WRONGARGS=65
11
12 if [ $# -ne "$ARGS" ] # 检查命令行参数的个数.
13 then
14    echo "Usage: `basename $0` filename column-number"
15    exit $E_WRONGARGS
16 fi
17
18 filename=$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.

```
bash$ export var=(a b); echo ${var[0]}
(a b)


bash$ var=(a b); export var; echo ${var[0]}
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强力的工具. 这个命令与外部命令getopt, 还有*C*语言中的库函数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.

```
while getopts ":abcde:fg" Option
# 开始的声明.
# a, b, c, d, e, f, 和 g 被认为是选项(标志).
# 'e' 选项后边的 : 提示这个选项需要带一个参数.
# 译者注:       解释一下 'a' 前边的那个 : 的作用.
#                    如果选项'e'不带参数进行调用的话, 会产生一个错误信息.
#                    这个开头的 : 就是用来屏蔽掉这个错误信息的,
#                    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.
do
  case $Option in
    a ) # 对选项'a'作些操作.
    b ) # 对选项'b'作些操作.
    ...
    e)  # 对选项'e'作些操作, 同时处理一下$OPTARG,
        # 这个变量里边将保存传递给选项"e"的参数.
    ...
    g ) # 对选项'g'作些操作.
  esac
done
shift $(($OPTIND - 1))
# 将参数指针向下移动.

# 所有这些远没有它看起来的那么复杂.<嘿嘿>.

```

例子 11-20. 使用getopts命令来来读取传递给脚本的选项/参数

```
#!/bin/bash
# 练习 getopts 和 OPTIND
# 在Bill Gradwohl的建议下, 这个脚本于 10/09/03 被修改.


# 在这里我们将学习如何使用 'getopts' 来处理脚本的命令行参数.
# 参数被作为"选项"(标志)来解析, 并且对选项分配参数.

# 试一下, 使用如下方法来调用这个脚本
# 'scriptname -mn'
# 'scriptname -oq qOption' (qOption 可以是任意的哪怕有些诡异字符的字符串.)
# 'scriptname -qXXX -r'
#
# 'scriptname -qr'     - 意外的结果, "r" 将被看成是选项 "q" 的参数.
# 'scriptname -q -r'   - 意外的结果, 同上.
# 'scriptname -mnop -mnop'   - 意外的结果
# (OPTIND在选项刚传递进来的地方是不可靠的).
# (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.
#  比如:       -mnop 在mno处理的位置OPTION都为1, 而到p的处理就变成2,
#                    -m -n -o 在m的时候OPTION为2, 而n为3, o为4,
#                    也就是说它总指向下一个位置).
#
#  如果选项需要一个参数的话("flag:"), 那么它将获取
#+ 命令行上紧挨在它后边的任何字符.

NO_ARGS=0
E_OPTERROR=65

if [ $# -eq "$NO_ARGS" ]  # 不带命令行参数就调用脚本?
then
  echo "Usage: `basename $0` options (-mnopqrs)"
  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.
fi
# 用法: scriptname -options
# 注意: 必须使用破折号 (-)


while getopts ":mnopq:rs" Option
do
  case $Option in
    m     ) echo "Scenario #1: option -m-   [OPTIND=${OPTIND}]";;
    n | o ) echo "Scenario #2: option -$Option-   [OPTIND=${OPTIND}]";;
    p     ) echo "Scenario #3: option -p-   [OPTIND=${OPTIND}]";;
```

```
    q     ) echo "Scenario #4: option -q-\
 with argument \"$OPTARG\"   [OPTIND=${OPTIND}]";;
    #  注意, 选项'q'必须分配一个参数,
    #+ 否则, 默认将失败.
    r | s ) echo "Scenario #5: option -$Option-";;
    *     ) echo "Unimplemented option chosen.";;   # 默认情况的处理
  esac
done

shift $(($OPTIND - 1))
#  (译者注: shift命令是可以带参数的, 参数就是移动的个数)
#  将参数指针减1, 这样它将指向下一个参数.
#  $1 现在引用的是命令行上的第一个非选项参数,
#+ 如果有一个这样的参数存在的话.

exit 0

#   就像 Bill Gradwohl 所描述的,
#  "getopts机制允许指定一个参数,
#+ 但是scriptname -mnop -mnop就是一种比较特殊的情况,
#+ 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. sourc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

```
#!/bin/bash

. data-file    # 加载一个数据文件.
# 与"source data-file"效果相同, 但是更具可移植性.

#  文件"data-file"必须存在于当前工作目录,
#+ 因为这个文件是使用'basename'来引用的.

# 现在, 引用这个文件中的一些数据.

echo "variable1 (from data-file) = $variable1"
echo "variable3 (from data-file) = $variable3"

let "sum = $variable2 + $variable4"
echo "Sum of variable2 + variable4 (from data-file) = $sum"
echo "message1 (from data-file) is \"$message1\""
# 注意:                                 将双引号转义

print_message This is the message-print function in the data-file.


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.

```
# 这是需要被脚本加载的数据文件.
# 这种文件可以包含变量, 函数, 等等.
# 在脚本中可以通过'source'或者'.'命令来加载.

# 让我们初始化一些变量.

variable1=22
variable2=474
variable3=5
variable4=97

message1="Hello, how are you?"
message2="Enough for now. Goodbye."

print_message ()
{
# 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 <zzz-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
 3 function1 ()
 4 {
 5   # 在 function1 () 内部.
 6   caller 0   # 显示调用者信息.
 7 }
 8
 9 function1    # 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
16 caller 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.

```
bash$ type '['
[ is a shell builtin
bash$ type -a '['
[ is a shell builtin
[ 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*readline* [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.

```
bash$ help exit
exit: exit [n]
    Exit the shell with a status of N.  If N is omitted, the exit
status
    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)

11.1. 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.

```
bash$ sleep 100 &
[1] 1384

bash $ jobs
[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
 3 ROOT_UID=0   # 只有$UID为0的用户才拥有root权限.
 4 E_NOTROOT=65
 5 E_NOPARAMS=66
 6
 7 if [ "$UID" -ne "$ROOT_UID" ]
 8 then
 9   echo "Must be root to run this script."
10   # "Run along kid, it's past your bedtime."
11   exit $E_NOTROOT
12 fi
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."
```

```
bash$ ./test.sh
Done.
 [bozo@localhost test-scripts]$ total 1
 -rwxr-xr-x    1 bozo     bozo           34 Oct 11 15:09
test.sh
 —
```

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
```

```
bash$ ./test.sh
Done.
 [bozo@localhost test-scripts]$ total 1
 -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:

```
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
20 #              结束信号
```

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 1   别名
2 2   关键字
3 3   函数
4 4   内建命令
5 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)
%+	"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.

---

12. 外部过滤器，程序和命令

目录
12.1. 基本命令
12.2. 复杂命令
12.3. 时间/日期 命令
12.4. 文本处理命令
12.5. 文件与归档命令
12.6. 通讯命令
12.7. 终端控制命令
12.8. 数学计算命令
12.9. 混杂命令

标准的 UNIX 命令使得 shell 脚本更加灵活.通过简单的编程结构把shell指令和系统命令结 合起来,这才是脚本能力的所在.

12.1. 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>>), 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.

```
bash$ cat file1.txt
This is line 1.
 This is line 2.


bash$ tac file1.txt
This is line 2.
 This is line 1.


bash$ rev file1.txt
.1 enil si sihT
 .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).

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.

```
bash$ mv source_directory target_directory

bash$ ls -lF target_directory
total 1
 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).

rm将无法删除以破折号开头的文件.

```
bash$ rm -badname
rm: invalid option -- b
 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./ .

```
bash$ rm ./-badname
```

另一种解决的方法是在文件名前边加上" -- ".

```
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

```
root# chattr +i file1.txt


root# rm file1.txt
```

```
rm: remove write-protected regular file `file1.txt'? y
 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`.

如果之前`newfile`已经存在的话, 将会产生一个错误信息.

使用链接中的哪种类型?

就像John Macdonald解释的那样:

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, 不管你用文件的哪个名字对文件内容进行修改, 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. 当你工作在更高层次的时候, 才会发生软硬链接的不同. 硬链接的优点是, 原始文件与链接文件之间是相互独立的 -- 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, 硬链接的文件讲还是原来文件的内容. 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, 软链接将再也找不到原来文件的内容了. 而软链接的优点是它可以跨越文件系统(因为它只不过是文件名的一个引用, 而并不是真正的数据). 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.

12.2. 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shll传递到find命令中的字符不会被解释为其他的特殊字符).

```
bash$ find ~/ -name '*.txt'
/home/bozo/.kde/share/apps/karm/karmdata.txt
 /home/bozo/misc/irmeyc.txt
 /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 '[0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-9]*[.][0-
9][0-9]*' {} \;
  2
  3 # 在 /etc 目录中的文件找到所所有包含 IP 地址(xxx.xxx.xxx.xxx) 的文件.
  4 # 可能会查找到一些多余的匹配. 我们如何去掉它们呢?
  5
  6 # 或许可以使用如下方法:
  7
  8 find /etc -type f -exec cat '{}' \; | tr -c '.[:digit:]' '\n' \
  9 | grep '^[^.][^.]*\.[^.][^.]*\.[^.][^.]*\.[^.][^.]*$'
 10 #
 11 #  [:digit:] 是一种字符类.
 12 #+ 关于字符类的介绍请参考 POSIX 1003.2 标准化文档.
 13
 14 # 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.(这些特殊字符指的是不应该出现在文件
名中的字符)
```

```
  4
  5 for filename in *
  6 do
  7   badname=`echo "$filename" | sed -n /[\+\{\;\"\\\=\?
~\(\)\<\>\&\*\|\$]/p`
  8 # badname=`echo "$filename" | sed -n '/[+{;"\=?~()<>&*|$]/p'`  这句也
行.
  9 # 删除文件名包含这些字符的文件:     + { ; " \ = ? ~ ( ) < > & * | $
 10 #
 11   rm $badname 2>/dev/null
 12 #             ^^^^^^^^^^^ 错误消息将被抛弃.
 13 done
 14
 15 # 现在, 处理文件名中以任何方式包含空白的文件.
 16 find . -name "* *" -exec rm -f {} \;
 17 # "find"命令匹配到的目录名将替换到"{}"的位置.
 18 # '\'是为了保证';'被正确的转义, 并且放到命令的结尾.
 19
 20 exit 0
 21
 22 #-----------------------------------------------------------------------
--
 23 # 这行下边的命令将不会运行, 因为有 "exit" 命令.
 24
 25 # 下边这句可以用来替换上边的脚本:
 26 find . -name '*[+{;"\\=?~()<>&*|$ ]*' -exec rm -f '{}' \;
 27 # (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.

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.

```
bash$ ls -l
total 0
 -rw-rw-r--    1 bozo  bozo         0 Jan 29 23:58 file1
 -rw-rw-r--    1 bozo  bozo         0 Jan 29 23:58 file2



bash$ ls -l | xargs
total 0 -rw-rw-r-- 1 bozo bozo 0 Jan 29 23:58 file1 -rw-rw-r-- 1 bozo
bozo 0 Jan 29 23:58 file2



bash$ find ~/mail -type f | xargs grep "Linux"
./misc:User-Agent: slrn/0.9.8.1 (Linux)
 ./sent-mail-jul-2005: hosted by the Linux Documentation Project.
 ./sent-mail-jul-2005: (Linux Documentation Project Site, rtf version)
 ./sent-mail-jul-2005: Subject: Criticism of Bozo's Windows/Linux
article
 ./sent-mail-jul-2005: while mentioning that the Linux ext2/ext3
filesystem
 . . .
```

`ls | xargs -p -l gzip` 使用gzips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 -print0`grep -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/messagesGenerates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
>>logfile
 29
 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.
```

As in find, a curly bracket pair serves as a placeholder for replacement text.

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
  7 E_NOARGS=65
  8
  9 if [ -z "$1" ]   # 如果没有参数传递进来那就退出.
 10 then
 11   echo "Usage: `basename $0` directory-to-copy-to"
 12   exit $E_NOARGS
 13 fi
 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
 15 E_BADARGS=66
 16
 17 if test -z "$1"  # 没有参数传递进来?
 18 then
 19   echo "Usage: `basename $0` Process(es)_to_kill"
 20   exit $E_BADARGS
 21 fi
 22
 23
 24 PROCESS_NAME="$1"
```

```
 25 ps 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.
  9 ARGS=1
 10 E_BADARGS=65
 11 E_NOFILE=66
 12
 13 if [ $# -ne "$ARGS" ]
 14 # 纠正传递到脚本中的参数个数?
 15 then
 16   echo "Usage: `basename $0` filename"
 17   exit $E_BADARGS
 18 fi
 19
 20 if [ ! -f "$1" ]       # 检查文件是否存在.
 21 then
 22   echo "File \"$1\" does not exist."
 23   exit $E_NOFILE
 24 fi
 25
 26
 27
 28 ########################################################################
 29 cat "$1" | xargs -n1 | \
 30 #  列出文件, 每行一个单词.
 31 tr A-Z a-z | \
 32 #  将字符转换为小写.
 33 sed -e 's/\.//g'  -e 's/\,//g' -e 's/ /\
 34 /g' | \
 35 #  过滤掉句号和逗号,
 36 #+ 并且将单词间的空格修改为换行,
 37 sort | uniq -c | sort -nr
 38 #  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.
 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

```
#!/bin/bash

# 展示一些使用'expr'的例子
# ========================

echo

# 算术 操作
# ---- ----

echo "Arithmetic Operators"
echo
a=`expr 5 + 3`
echo "5 + 3 = $a"

a=`expr $a + 1`
echo
echo "a + 1 = $a"
echo "(incrementing a variable)"

a=`expr 5 % 3`
# 取模操作
echo
echo "5 mod 3 = $a"

echo
echo

# 逻辑 操作
# ---- ----

#  true返回1, false返回0,
#+ 而Bash的使用惯例则相反.

echo "Logical Operators"
echo

x=24
y=25
b=`expr $x = $y`          # 测试相等.
echo "b = $b"             # 0  ( $x -ne $y )
echo

a=3
b=`expr $a \> 10`
echo 'b=`expr $a \> 10`, therefore...'
echo "If a > 10, b = 0 (false)"
echo "b = $b"             # 0  ( 3 ! -gt 10 )
echo

b=`expr $a \< 10`
echo "If a < 10, b = 1 (true)"
echo "b = $b"             # 1  ( 3 -lt 10 )
echo
# 注意转义操作.

b=`expr $a \<= 3`
echo "If a <= 3, b = 1 (true)"
echo "b = $b"             # 1  ( 3 -le 3 )
# 也有 "\>=" 操作 (大于等于).


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

[1] 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.

12.3. 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).

```
bash$ date
Fri Mar 29 21:07:39 MST 2002



bash$ date -u
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.

```
bash$ zdump EST
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:

```
0.00user 0.01system 0:00.05elapsed 16%CPU (0avgtext+0avgdata
0maxresident)k
 0inputs+0outputs (149major+27minor)pagefaults 0swaps
```

请参考前边章节所讲的一个类似的命令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, 这就非常有用了.

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Ctl-D结束.

你可以使用-f选项或者使用(<)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.

```
bash$ at 2:30 am Friday < at-jobs.list
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微秒.
```

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.

12.4. 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.

```
cat list-1 list-2 list-3 | sort | uniq > final.list
# 将3个文件连接起来,
# 将它们排序,
# 删除其中重复的行,
# 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.

```
bash$ cat testfile
This line occurs only once.
 This line occurs twice.
 This line occurs twice.
 This line occurs three times.
 This line occurs three times.
 This line occurs three times.


bash$ uniq -c testfile
      1 This line occurs only once.
       2 This line occurs twice.
       3 This line occurs three times.


bash$ sort testfile | uniq -c | sort -nr
      3 This line occurs three times.
       2 This line occurs twice.
       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

```
#!/bin/bash
# wf.sh: 分析文本文件中词汇出现的频率.
# "wf2.sh"脚本是一个效率更高的版本.


# 从命令行中检查输入的文件.
ARGS=1
E_BADARGS=65
E_NOFILE=66

if [ $# -ne "$ARGS" ]  # 检验传递到脚本中参数的个数.
then
  echo "Usage: `basename $0` filename"
  exit $E_BADARGS
fi

if [ ! -f "$1" ]       # 检查传入的文件是否存在.
then
  echo "File \"$1\" does not exist."
```

```
  exit $E_NOFILE
fi



############################################################
# main ()
sed -e 's/\.//g'  -e 's/\,//g' -e 's/ /\
/g' "$1" | tr 'A-Z' 'a-z' | sort | uniq -c | sort -nr
#                            ========================
#                               检查单词出现的频率

#  过滤掉句号和逗号,
#+ 并且把单词间的空格转化为换行,
#+ 然后转化为小写,
#+ 最后统计单词出现的频率并按频率排序.

#  Arun Giridhar建议将上边的代码修改为:
#  . . . | sort | uniq -c | sort +1 [-f] | sort +0 -nr
#  这句添加了第2个排序主键, 所以
#+ 这个与上边等价的例子将按照字母顺序进行排序.
#  就像他所解释的:
#  "这是一个有效的根排序, 首先对频率最少的
#+ 列进行排序
#+ (单词或者字符串, 忽略大小写)
#+ 然后对频率最高的列进行排序."
#
#  像Frank Wang所解释的那样, 上边的代码等价于:
#+       . . . | sort | uniq -c | sort +0 -nr
#+ 用下边这行也行:
#+       . . . | sort | uniq -c | sort -k1nr -k
############################################################

exit 0

# 练习:
# -----
# 1) 使用'sed'命令来过滤其他的标点符号,
#+   比如分号.
# 2) 修改这个脚本, 添加能够过滤多个空格或者
#    空白的能力.
```

```
bash$ cat testfile
This line occurs only once.
 This line occurs twice.
 This line occurs twice.
 This line occurs three times.
 This line occurs three times.
 This line occurs three times.


bash$ ./wf.sh testfile
      6 this
       6 occurs
       6 line
       3 times
       3 three
       2 twice
       1 only
       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:

```
bash$ grep '^Subject:' read-messages | cut -c10-80
Re: Linux suitable for mission-critical apps?
 MAKE MILLIONS WORKING AT HOME!!!
 Spam complaint
 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.
```

`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).

```
bash$ cut -d'
 ' -f3,7,19 testfile
This is line 3 of testfile.
 This is line 7 of testfile.
 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
```

```
bash$ join 1.data 2.data
File: 1.data 2.data

 100 Shoes $40.00
 200 Laces $1.00
 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?

```
#!/bin/bash
# script-detector.sh: 在一个目录中检查所有的脚本文件.

TESTCHARS=2    # 测试前两个字符.
SHABANG='#!'   # 脚本都是以"#!"开头的.

for file in *  # 遍历当前目录下的所有文件.
do
  if [[ `head -c$TESTCHARS "$file"` = "$SHABANG" ]]
  #      head -c2                    #!
  #  '-c' 选项将从文件头输出指定个数的字符,
  #+ 而不是默认的行数.
  then
    echo "File \"$file\" is a script."
  else
    echo "File \"$file\" is *not* a script."
  fi
done

exit 0

#  练习:
#  -----
#  1) 修改这个脚本,
#+    让它可以指定扫描的路径.
#+    (而不是只搜索当前目录).
#
#  2) 以这个脚本目前的状况, 它不能正确识别出
#+    Perl, awk, 和其他一些脚本语言的脚本文件.
#     修正这个问题.
```

例子 12-13. 产生10-进制随机数

```
#!/bin/bash
# rnd.sh: 输出一个10进制随机数

# 由Stephane Chazelas所编写的这个脚本.

head -c4 /dev/urandom | od -N4 -tu4 | sed -ne '1s/.* //p'


# ============================================================== #

# 分析
# ----

# head:
# -c4 选项将取得前4个字节.

# od:
# -N4 选项将限制输出为4个字节.
# -tu4 选项将使用无符号10进制格式来输出.

# sed:
# -n 选项, 使用"s"命令与"p"标志组合的方式,
# 将会只输出匹配的行.



# 本脚本作者解释'sed'命令的行为如下.

# head -c4 /dev/urandom | od -N4 -tu4 | sed -ne '1s/.* //p'
# ---------------------------------> |

# 假设一直处理到"sed"命令时的输出--> |
# 为 0000000 1198195154\n

#  sed命令开始读取字串: 0000000 1198195154\n.
#  这里它发现一个换行符,
#+ 所以sed准备处理第一行 (0000000 1198195154).
#  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:

#   range     action
#   1         s/.* //p

#  因为行号在range中, 所以sed开始执行action:
#+ 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
#  ("0000000 "), 用空字符串(//)将这个匹配到的字串替换掉, 如果成功, 那就打印出结果
#  ("p"在这里是"s"命令的标志, 这与单独的"p"命令是不同的).

#  sed命令现在开始继续读取输入. (注意在继续之前,
#+ continuing, 如果没使用-n选项的话, sed命令将再次
#+ 将这行打印一遍).

```

```
52 # 现在, sed命令读取剩余的字符串, 并且找到文件的结尾.
53 # sed命令开始处理第2行(这行也被标记为'$'
54 # 因为这已经是最后一行).
55 # 所以这行没被匹配到<range>中, 这样sed命令就结束了.
56
57 #  这个sed命令的简短的解释是:
58 #  "在第一行中删除第一个空格左边全部的字符,
59 #+ 然后打印出来."
60
61 # 一个更好的来达到这个目的的方法是:
62 #           sed -e 's/.* //;q'
63
64 # 这里, <range>和<action>分别是(也可以写成
65 #           sed -e 's/.* //' -e q):
66
67 #   range                     action
68 #   nothing (matches line)    s/.* //
69 #   nothing (matches line)    q (quit)
70
71 #  这里, sed命令只会读取第一行的输入.
72 #  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),
73 #+ 因为没使用"-n"选项.
74
75 # ============================================================================ #
76
77 # 也可以使用如下一个更简单的语句来代替:
78 #           head -c4 /dev/urandom| od -An -tu4
79
80 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
 3 filename=sys.log
 4
 5 cat /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
10 tail /var/log/messages > $filename
11 # /var/log/messages 必须具有全局的可读权限才行.
12
13 echo "$filename contains tail end of system log."
14
15 exit 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:

```
1 var=$(head -$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.

```
bash$ grep '[rst]ystem.$' osinfo.txt
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.

```
bash$ ps ax | grep clock
765 tty1     S      0:00 xclock
 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.

```
bash$ grep -n Linux osinfo.txt
2:This is a file containing information about Linux.
 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\000\n\000e\000\000\nf' | grep -cz .     # 3
 10 printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -cz '$'   # 5
 11 printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -cz '^'   # 5
 12 #
 13 printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -c '$'    # 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.

```
bash$ grep Linux osinfo.txt misc.txt
osinfo.txt:This is a file containing information about Linux.
 osinfo.txt:The GPL governs the distribution of the Linux operating system.
 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.

```
bash$ grep Linux osinfo.txt /dev/null
osinfo.txt:This is a file containing information about Linux.
 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.

```
bash$ grep file tstfile
# 文件名: tstfile
 This is a sample file.
 This is an ordinary text file.
 This file does not contain any unusual text.
 This file is not unusual.

bash$ grep file tstfile | grep text
This is an ordinary text file.
 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.

```
bash $ egrep 'matches|Matches' file.txt
Line 1 matches.
 Line 3 Matches.
 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.
```

```
sed '$d' | sed '$d'
#  删除输出的最后两行
#+ (空行和下一部分的第一行).
# ------------------------------------------------------------

exit 0

# 练习:
# -----
# 1)  修改这个脚本, 让它具备能够处理任何字符形式的输入
#   + (大写, 小写, 或大小写混合), 然后将其转换为
#   + 能够处理的统一形式.
#
# 2)  将这个脚本转化为一个GUI应用,
#   + 使用一些比如像"gdialog"的东西 .  .  .
#     这样的话, 脚本将不再从命令行中
#   + 取得这些参数.
#
# 3)  修改这个脚本让它具备能够分析另外一个
#   + 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

```
#!/bin/bash
# lookup: 对指定数据文件中的每个单词都做一遍字典查询.

file=words.data   # 指定的要搜索的数据文件.

echo

while [ "$word" != end ]  # 数据文件中最后一个单词.
do
  read word       # 从数据文件中读, 因为在循环的后边重定向了.
  look $word > /dev/null  # 不想将字典文件中的行显示出来.
  lookup=$?       # 'look'命令的退出状态.

  if [ "$lookup" -eq 0 ]
  then
    echo "\"$word\" is valid."
  else
    echo "\"$word\" is invalid."
  fi

done <"$file"     # 将stdin重定向到$file, 所以"reads"来自于$file.

echo

exit 0

# ------------------------------------------------------------------
# 下边的代码行将不会执行, 因为上边已经有"exit"命令了.


# Stephane Chazelas建议使用下边更简洁的方法:

while read word && [[ $word != end ]]
do if look "$word" > /dev/null
   then echo "\"$word\" is valid."
   else echo "\"$word\" is invalid."
   fi
done <"$file"

exit 0
```

---

sed, awk

这个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":

```
bash $ wc /usr/share/doc/sed-4.1.2/README
13  70  447 README
[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 $#)
8
9 # 感谢, S.C.
```

wc命令来统计所有以 d - h 开头的文件的大小.

```
bash$ wc [d-h]* | grep total | awk '{print $3}'
71832
```

使用wc命令来查看指定文件中包含"Linux"的行一共有多少.

```
bash$ grep Linux abs-book.sgml | wc -l
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.

```
echo "abcdef"                   # abcdef
echo "abcdef" | tr -d b-d       # aef


tr -d 0-9 <filename
# 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.

```
bash$ echo "XXXXX" | tr --squeeze-repeats 'X'
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.

```
bash$ echo "acfdeb123" | tr -c b-d +
+c+d+b++++
```

注意tr命令支持POSIX字符类. [1]

```
bash$ echo "abcd2ef1" | tr '[:alpha:]' -
----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.

```
#!/bin/bash
# 把一个文件的内容全部转换为大写.

E_BADARGS=65

if [ -z "$1" ]  # 检查命令行参数.
then
  echo "Usage: `basename $0` filename"
  exit $E_BADARGS
fi

tr a-z A-Z <"$1"

# 与上边的作用相同, 但是使用了POSIX字符集标记方法:
#       tr '[:lower:]' '[:upper:]' <"$1"
# 感谢, S.C.

exit 0

#  练习:
#  重写这个脚本, 通过选项可以控制脚本或者
#+ 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.

```
#!/bin/bash
#
#  将当前目录下的所有文全部转换为小写.
#
#  灵感来自于John Dubois的脚本,
#+ Chet Ramey将其转换为Bash脚本,
#+ 然后被本书作者精简了一下.


for filename in *                 # 遍历当前目录下的所有文件.
do
   fname=`basename $filename`
   n=`echo $fname | tr A-Z a-z`   # 将名字修改为小写.
   if [ "$fname" != "$n" ]        # 只对那些文件名不是小写的文件进行重命名.
   then
     mv $fname $n
   fi
done

exit $?


# 下边的代码将不会被执行, 因为上边的"exit".
#-----------------------------------------#
# 删除上边的内容, 来运行下边的内容.

```

```
# 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.
# Stephane Chazelas因此建议使用下边的方法:


for filename in *     # 不必非得使用basename命令,
                      # 因为"*"不会返回任何包含"/"的文件.
do n=`echo "$filename/" | tr '[:upper:]' '[:lower:]'`
#                           POSIX 字符集标记法.
#                     添加的斜线是为了在文件名结尾换行不会被
#                     命令替换删掉.
   # 变量替换:
   n=${n%/}           # 从文件名中将上边添加在结尾的斜线删除掉.
   [[ $filename == $n ]] || mv "$filename" "$n"
                      # 检查文件名是否已经是小写.
done

exit $?
```

例子 12-20. Du: DOS到UNIX文本文件的转换.

```
#!/bin/bash
# Du.sh: DOS到UNIX文本文件的转换.

E_WRONGARGS=65

if [ -z "$1" ]
then
  echo "Usage: `basename $0` filename-to-convert"
  exit $E_WRONGARGS
fi

NEWFILENAME=$1.unx

CR='\015'  # 回车.
           # 015是8进制的ASCII码的回车.
           # DOS中文本文件的行结束符是CR-LF.
           # UNIX中文本文件的行结束符只是LF.

tr -d $CR < $1 > $NEWFILENAME
# 删除回车并且写到新文件中.

echo "Original DOS text file is \"$1\"."
echo "Converted UNIX text file is \"$NEWFILENAME\"."

exit 0

# 练习:
# -----
# 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.

```
#!/bin/bash
# rot13.sh: 典型的rot13算法,
#           使用这种方法加密至少可以愚弄一下3岁小孩.

# 用法: ./rot13.sh filename
# 或     ./rot13.sh <filename
# 或     ./rot13.sh and supply keyboard input (stdin)

cat "$@" | tr 'a-zA-Z' 'n-za-mN-ZA-M'   # "a"变为"n", "b"变为"o", 等等.
#  'cat "$@"'结构
#+ 允许从stdin或者从文件中获得输入.

exit 0
```

例子 12-22. 产生"Crypto-Quote"游戏(译者: 一种文字游戏)

```
#!/bin/bash
# crypto-quote.sh: 加密

#  使用单码替换(单一字母替换法)来进行加密.
#  这个脚本的结果与"Crypto Quote"游戏
#+ 的行为很相似.


key=ETAOINSHRDLUBCFGJMQPVWZYXK
# "key"不过是一个乱序的字母表.
# 修改"key"就会修改加密的结果.

```

```
# 'cat "$@"' 结构既可以从stdin获得输入, 也可以从文件中获得输入.
# 如果使用stdin, 那么要想结束输入就使用 Control-D.
# 否则就要在命令行上指定文件名.

cat "$@" | tr "a-z" "A-Z" | tr "A-Z" "$key"
#          |    转化为大写    |     加密
# 小写, 大写, 或混合大小写, 都可以正常工作.
# 但是传递进来的非字母字符将不会起任何变化.


# 用下边的语句试试这个脚本:
# "Nothing so needs reforming as other people's habits."
# --Mark Twain
#
# 输出为:
# "CFPHRCS QF CIIOQ MINFMBRCS EQ FPHIM GIFGUI'Q HETRPQ."
# --BEML PZERC

# 解密:
# cat "$@" | tr "$key" "A-Z"


#  这个简单的密码可以轻易的被一个12岁的小孩
#+ 用铅笔和纸破解.

exit 0

#  练习:
#  -----
#  修改这个脚本, 让它可以用命令行参数
#+ 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".

例子 12-23. 格式化文件列表.

```
#!/bin/bash

WIDTH=40                        # 设为40列宽.

b=`ls /usr/local/bin`           # 取得文件列表...

echo $b | fmt -w $WIDTH

# 也可以使用如下方法, 作用是相同的.
#    echo $b | fold - -s -w $WIDTH

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.

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 | cut -b 3- | tr -d \\n
>> "$BOOKLIST"
  6 }
  7
  8 #  来自于Peter Knowles的"booklistgen.sh"脚本
  9 #+ 目的是把文件转换为Sony Librie格式.
 10 #  (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, enscript 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
  8 E_WRONGARGS=65
  9
 10 if [ -z "$1" ]
 11 then
 12   echo "Usage: `basename $0` filename"
 13   exit $E_WRONGARGS
 14 fi
 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.

12.6. 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.

```
bash$ host surfacemail.com
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.

```
bash$ ipcalc -h 202.92.42.236
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.

```
bash$ nslookup -sil 66.97.104.180
nslookup kuhleersparnis.ch
 Server:         135.116.137.2
 Address:        135.116.137.2#53

 Non-authoritative answer:
 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.

```
bash$ dig -x 81.9.6.2
;; Got answer:
 ;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 11649
 ;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 0

 ;; QUESTION SECTION:
 ;2.6.9.81.in-addr.arpa.          IN      PTR

 ;; AUTHORITY SECTION:
 6.9.81.in-addr.arpa.    3600    IN      SOA     ns.eltel.net.
noc.eltel.net.
 2002031705 900 600 86400 3600

 ;; Query time: 537 msec
```

```
;; SERVER: 135.116.137.2#53(135.116.137.2)
;; WHEN: Wed Jun 26 08:35:24 2002
;; 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.
 6 ARGCOUNT=1
 7 E_WRONGARGS=65
 8 if [ $# -ne "$ARGCOUNT" ]
 9 then
10   echo "Usage: `basename $0` domain-name"
11   exit $E_WRONGARGS
12 fi
13
14
15 dig +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
```

```
#
# 并将其作为这个脚本的唯一参数.
#(另: 你的Inet连接应该保证连接好)
#
# 这样, 在上边两个实例中调用这个脚本:
#       is-spammer.sh spammer.biz


# Whitespace == :Space:Tab:Line Feed:Carriage Return:
WSP_IFS=$'\x20'$'\x09'$'\x0A'$'\x0D'

# No Whitespace == Line Feed:Carriage Return
No_WSP=$'\x0A'$'\x0D'

# 域分隔符为点分10进制ip地址
ADR_IFS=${No_WSP}'.'

# 取得dns文本资源记录.
# get_txt <error_code> <list_query>
get_txt() {

    # 分析在"."中分配的$1.
    local -a dns
    IFS=$ADR_IFS
    dns=( $1 )
    IFS=$WSP_IFS
    if [ "${dns[0]}" == '127' ]
    then
        # 查看此处是否有原因.
        echo $(dig +short $2 -t txt)
    fi
}

# 取得dns地址资源纪录.
# chk_adr <rev_dns> <list_server>
chk_adr() {
    local reply
    local server
    local reason

    server=${1}${2}
    reply=$( dig +short ${server} )

    # 假设应答可能是一个错误码 . . .
    if [ ${#reply} -gt 6 ]
    then
        reason=$(get_txt ${reply} ${server} )
        reason=${reason:-${reply}}
    fi
    echo ${reason:-' not blacklisted.'}
}

# 需要从名字中取得 IP 地址.
echo 'Get address of: '$1
ip_adr=$(dig +short $1)
dns_reply=${ip_adr:-' no answer '}
echo ' Found address: '${dns_reply}

# 一个可用的应答至少是4个数字加上3个点.
if [ ${#ip_adr} -gt 6 ]
then
    echo
    declare query

    # 通过点中的分配进行分析.
    declare -a dns
    IFS=$ADR_IFS
    dns=( ${ip_adr} )
    IFS=$WSP_IFS

    # 用8进制表示法将dns查询循序记录起来.
    rev_dns="${dns[3]}"'.'"${dns[2]}"'.'"${dns[1]}"'.'"${dns[0]}"'.'

# 查看: http://www.spamhaus.org (传统地址, 维护的很好)
    echo -n 'spamhaus.org says: '
    echo $(chk_adr ${rev_dns} 'sbl-xbl.spamhaus.org')

# 查看: http://ordb.org (开放转发Open mail relay)
    echo -n '   ordb.org  says: '
    echo $(chk_adr ${rev_dns} 'relays.ordb.org')

# 查看: http://www.spamcop.net/ (你可以在这里报告spammer)
    echo -n ' spamcop.net says: '
    echo $(chk_adr ${rev_dns} 'bl.spamcop.net')

```

```
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.

```
bash$ traceroute 81.9.6.2
traceroute to 81.9.6.2 (81.9.6.2), 30 hops max, 38 byte packets
 1  tc43.xjbnnbrb.com (136.30.178.8)  191.303 ms  179.400 ms  179.767
ms
 2  or0.xjbnnbrb.com (136.30.178.1)  179.536 ms  179.534 ms  169.685 ms
 3  192.168.11.101 (192.168.11.101)  189.471 ms  189.556 ms *
 ...
```

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.

```
bash$ ping localhost
PING localhost.localdomain (127.0.0.1) from 127.0.0.1 : 56(84) bytes of
data.
 64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=0 ttl=255
time=709 usec
 64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=1 ttl=255
time=286 usec

 --- localhost.localdomain ping statistics ---
 2 packets transmitted, 2 packets received, 0% packet loss
 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.

```
bash$ finger
Login  Name            Tty      Idle  Login Time   Office     Office
Phone
 bozo   Bozo Bozeman   tty1        8  Jun 25 16:59
 bozo   Bozo Bozeman   ttyp0          Jun 25 16:59
 bozo   Bozo Bozeman   ttyp1          Jun 25 17:07



bash$ finger bozo
Login: bozo                             Name: Bozo Bozeman
 Directory: /home/bozo                   Shell: /bin/bash
 Office: 2355 Clown St., 543-1234
 On since Fri Aug 31 20:13 (MST) on tty1    1 hour 38 minutes idle
 On since Fri Aug 31 20:13 (MST) on pts/0   12 seconds idle
 On since Fri Aug 31 20:13 (MST) on pts/1
 On since Fri Aug 31 20:31 (MST) on pts/2   1 hour 16 minutes idle
 No mail.
 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.

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.

```
wget -p http://www.xyz23.com/file01.html
#  -p或--page-requisite选项将会使得wget取得所有在显示指定页时
#+ 所需要的文件. (译者: 比如内嵌图片和样式表等.)

wget -r ftp://ftp.xyz24.net/~bozo/project_files/ -O $SAVEFILE
#  -r选项将会递归的从指定站点
#+ 上下载所有连接.
```

例子 12-38. 获得一份股票报价

```
#!/bin/bash
# quote-fetch.sh: 下载一份股票报价.


E_NOPARAMS=66

if [ -z "$1" ]  #必须指定需要获取的股票(代号).
  then echo "Usage: `basename $0` stock-symbol"
  exit $E_NOPARAMS
fi

stock_symbol=$1

file_suffix=.html
# 获得一个HTML文件, 所以要正确命名它.
URL='http://finance.yahoo.com/q?s='
# Yahoo金融板块, 后缀是股票查询.

# -----------------------------------------------------------
wget -O ${stock_symbol}${file_suffix} "${URL}${stock_symbol}"
# -----------------------------------------------------------


# 在http://search.yahoo.com上查询相关材料:
# -----------------------------------------------------------
# URL="http://search.yahoo.com/search?fr=ush-news&p=${query}"
# wget -O "$savefilename" "${URL}"
# -----------------------------------------------------------
# 保存相关URL的列表.

exit $?

# 练习:
# -----
#
# 1) 添加一个测试来验证用户是否在线.
#    (暗示: 对"ppp"或"connect"来分析'ps -ax'的输出.
#
# 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,
#+   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.

```
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, 但是这个包更新一些,
152                 pre_date=$cur_date                       #+ 那么就更新
最新的.
153                 pre_file=$cur_file
154             fi
155             done
156             echo $pre_file >> $TMP                       #  TMP现在包
含所有
157                                                          #+ 提炼过的列
表.
158             # echo "subshell=$BASH_SUBSHELL"
159
160     }       # 这里的大括号是为了让最后这句"echo $pre_file >> $TMP"
161             # 也能与整个循环一起放到同一个子shell ( 1 )中.
162
163     RET=$?  # 取得管道命令的返回状态.
164
165     [ "$RET" -ne 0 ] && {
166         echo "List retrieving failed with code $RET"
167         exit $E_RETURN
168     }
169
170     echo "done"; echo
171 }
172
173 # 真正的rsync下载部分.
174 get_file () {
175
176     echo "Downloading..."
177     /bin/nice /usr/bin/rsync \
178         $OPTS \
179         --filter "merge,+/ $TMP" \
180         --exclude '*'  \
181         $URL $DEST     \
182         | /usr/bin/tee $LOG
183
184     RET=$?
185
186         #  --filter merge,+/ 对于这个目的来说, 这句是至关重要的.
187         #  + 修饰语意为着包含, / 意味着绝对路径.
188         #  然后$TMP中排过序的列表将会包含升序的路径名,
189         #+ 并从"简化的流程"(shortcutting the circuit)中阻止下边的 --
exclude '*'.
190
191     echo "Done"
192
193     rm -f $PID_FILE 2>/dev/null
194
195     return $RET
196 }
197
198 # -------
199 # Main
200 init
201 check_pid
202 set_range
203 get_list
204 get_file
205 RET=$?
206 # -------
207
208 if [ "$RET" -eq 0 ]; then
```

```
209     /usr/bin/logger -t ${0##*/} "Fedora update mirrored
successfully."
210 else
211     /usr/bin/logger -t ${0##*/} "Fedora update mirrored with failure
code: $RET"
212 fi
213
214 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
```

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.

Mail

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 $0`\" has mailed itself to
you." "$adr"
16 #
================================================================================
17
18 # ---------------------------------------------
19 #  来自self-mailing脚本的一份祝福.
20 #  一个喜欢恶搞的家伙运行了这个脚本,
21 #+ 这导致了他自己收到了这份mail.
22 #  显然的, 有些人确实没什么事好做,
```

```
23 #+ 就只能浪费他们自己的时间玩了.
24 # ------------------------------------------
25
26 echo "At `date`, script \"`basename $0`\" mailed to "$adr"."
27
28 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.

12.7. 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.

```
bash$ tput longname
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.

```
bash$ infocmp
#       通过infocmp显示出来, 内容都来自于文件:
 /usr/share/terminfo/r/rxvt
 rxvt|rxvt terminal emulator (X Window System),
         am, bce, eo, km, mir, msgr, xenl, xon,
         colors#8, cols#80, it#8, lines#24, pairs#64,
         acsc=``aaffggjjkkllmmnnooppqqrrssttuuvvwwxxyyzz{{||}}~~,
         bel=^G, blink=\E[5m, bold=\E[1m,
         civis=\E[?25l,
         clear=\E[H\E[2J, cnorm=\E[?25h, cr=^M,
         ...
```

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.

12.8. 数学计算命令

"操作数字"

factor

将一个正数分解为多个素数.

```
bash$ factor 27417
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.

```
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
  9 echo
 10 echo "Given the principal, interest rate, and term of a mortgage,"
 11 echo "calculate the monthly payment."
 12
 13 bottom=1.0
 14
 15 echo
 16 echo -n "Enter principal (no commas) "
 17 read principal
 18 echo -n "Enter interest rate (percent) "  # 如果是12%，那就键入"12",
而不是".12".
 19 read interest_r
 20 echo -n "Enter term (months) "
 21 read term
 22
 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; ($interest_rate^$term-1)/($interest_rate-
1)" | bc)
 69
 70 #  因为 . . .
 71 #  在循环后边的算法
 72 #+ 事实上是一个等比数列的求和公式.
 73 #  求和公式是 e0(1-q^n)/(1-q),
 74 #+ e0是第一个元素, q=e(n+1)/e(n),
 75 #+ n是元素数量.
 76 # ------------------------------------------------------------------
---------
 77
 78
 79  # let "payment = $top/$bottom"
 80  payment=$(echo "scale=2; $top/$bottom" | bc)
 81  # 使用2位有效数字来表示美元和美分.
 82
 83  echo
 84  echo "monthly payment = \$$payment"  # 在总和的前边显示美元符号.
 85  echo
 86
 87
 88  exit 0
 89
 90
 91  # 练习:
 92  #   1) 处理输入允许本金总数中的逗号.
 93  #   2) 处理输入允许按照百分号和小数点的形式输入利率.
 94  #   3) 如果你真正想好好编写这个脚本,
 95  #      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 ##########################################################################
```

```
# 描述
#
# 修改纪录
# 21-03-95 stv      fixed error occuring with 0xb as input (0.2)
################################################################################

# ==> 在本书中使用这个脚本通过了原作者的授权.
# ==> 注释是本书作者添加的.

NOARGS=65
PN=`basename "$0"`                              # 程序名
VER=`echo '$Revision: 1.2 $' | cut -d' ' -f2`   # ==> VER=1.2

Usage () {
    echo "$PN - print number to different bases, $VER (stv '95)
usage: $PN [number ...]

If no number is given, the numbers are read from standard input.
A number may be
    binary (base 2)         starting with 0b (i.e. 0b1100)
    octal (base 8)          starting with 0  (i.e. 014)
    hexadecimal (base 16)   starting with 0x (i.e. 0xc)
    decimal                 otherwise (i.e. 12)" >&2
    exit $NOARGS
}   # ==> 打印出用法信息的函数.

Msg () {
    for i   # ==> 省略[list].
    do echo "$PN: $i" >&2
    done
}

Fatal () { Msg "$@"; exit 66; }

PrintBases () {
    # 决定数字的数制
    for i      # ==> 省略[list]...
    do         # ==> 所以是对命令行参数进行操作.
    case "$i" in
        0b*)                ibase=2;;       # 2进制
        0x*|[a-f]*|[A-F]*)  ibase=16;;      # 16进制
        0*)                 ibase=8;;       # 8进制
        [1-9]*)             ibase=10;;      # 10进制
        *)
            Msg "illegal number $i - ignored"
            continue;;
    esac

    # 去掉前缀, 将16进制数字转换为大写(bc命令需要这么做)
    number=`echo "$i" | sed -e 's:^0[bBxX]::' | tr '[a-f]' '[A-F]'`
    # ==> 使用":" 作为sed分隔符, 而不使用"/".

    # 将数字转换为10进制
    dec=`echo "ibase=$ibase; $number" | bc`  # ==> 'bc'是个计算工具.
    case "$dec" in
        [0-9]*)     ;;                      # 数字没问题
        *)          continue;;              # 错误: 忽略
    esac

    # 在一行上打印所有转换后的数字.
    # ==> 'here document'提供命令列表给'bc'.
    echo `bc <<!
        obase=16; "hex="; $dec
        obase=10; "dec="; $dec
        obase=8;  "oct="; $dec
        obase=2;  "bin="; $dec
!
    ` | sed -e 's: :        :g'

    done
}

while [ $# -gt 0 ]
# ==>  这里必须使用一个"while循环",
# ==>+ 因为所有的case都可能退出循环或者
# ==>+ 结束脚本.
# ==> (感谢, Paulo Marcel Coelho Aragao.)
do
    case "$1" in
    --)     shift; break;;
    -h)     Usage;;                 # ==> 帮助信息.
    -*)     Usage;;
         *)     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
 6 var1=`bc << EOF
 7 18.33 * 19.78
 8 EOF
 9 `
10 echo $var1       # 362.56
11
12
13 #  使用$( ... )这种标记法也可以.
14 v1=23.53
15 v2=17.881
16 v3=83.501
17 v4=171.63
18
19 var2=$(bc << EOF
20 scale = 4
21 a = ( $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.
```

```
hypotenuse ()    # 计算直角三角形的斜边.
{
hyp=$(bc -l << EOF
scale = 9
sqrt ( $1 * $1 + $2 * $2 )
EOF
)
# 不幸的是, 不能从bash函数中返回浮点值.
}

hypotenuse 3.68 7.31
echo "hypotenuse = $hyp"    # 8.184039344


exit 0
```

例子 12-45. 计算圆周率

```
#!/bin/bash
# cannon.sh: Approximating PI by firing cannonballs.

# 这事实上是一个"Monte Carlo"蒙特卡洛模拟的非常简单的实例:
#+ 蒙特卡洛模拟是一种由现实事件抽象出来的数学模型,
#+ 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.

#  想象有一个完美的正方形土地, 边长为10000个单位.
#  在这块土地的中间有一个完美的圆形湖,
#+ 这个湖的直径是10000个单位.
#  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.
#  (可以把这个湖想象成为这个正方形的内接圆.)
#
#  我们将使用老式的大炮和铁炮弹
#+ 向这块正方形的土地上开炮.
#  所有的炮弹都会击中这块正方形土地的某个地方.
#+ 或者是打到湖上, 或者是打到4个角的土地上.
#  因为这个湖占据了这个区域大部分地方,
#+ 所以大部分的炮弹都会"扑通"一声落到水里.
#  而只有很少的炮弹会"砰"的一声落到4个
#+ 角的土地上.
#
#  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,
#+ 那么落到水里的炮弹与打出炮弹总数的比率,
#+ 大概非常接近于PI/4.
#
#  原因是所有的炮弹事实上都
#+ 打在了这个土地的右上角,
#+ 也就是, 笛卡尔坐标系的第一象限.
#  (之前的解释只是一个简化.)
#
#  理论上来说, 如果打出的炮弹越多, 就越接近这个数字.
#  然而, 对于shell 脚本来说一定会做些让步的,
#+ 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.
#  当然就会降低精度.


DIMENSION=10000  # 这块土地的边长.
                 # 这也是所产生随机整数的上限.

MAXSHOTS=1000    # 开炮次数.
                 # 10000或更多次的话, 效果应该更好, 但有点太浪费时间了.
PMULTIPLIER=4.0  # 接近于PI的比例因子.

get_random ()
{
SEED=$(head -1 /dev/urandom | od -N 1 | awk '{ print $2 }')
RANDOM=$SEED                                    #  来自于"seeding-
random.sh"
                                                #+ 的例子脚本.
let "rnum = $RANDOM % $DIMENSION"               #  范围小于10000.
echo $rnum
}

distance=        # 声明全局变量.
hypotenuse ()    # 从"alt-bc.sh"例子来的,
{                # 计算直角三角形的斜边的函数.
distance=$(bc -l << EOF
scale = 0
sqrt ( $1 * $1 + $2 * $2 )
EOF
)
#  设置 "scale" 为 0 , 好让结果四舍五入为整数值,
#+                                .
```

```
       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
  4 E_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"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.

```
bash$ echo "16i[q]sa[ln0=aln100%Pln100/snlbx]sbA0D68736142snlbxq" | dc"
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=13sidvsr[dli%0=1lrli2+dsi!>.]ds.xd1<2"
| dc
 25 # ---------------------------------------------------------------------
----------------
 26 # 上边这行代码是Michel Charpentier编写的<charpov@cs.unh.edu>.
 27 # 在此使用经过授权(感谢).
 28
```

```
29  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
 5 ARGS=2                 # 需要将2个直角边作为参数传递进来.
 6 E_BADARGS=65           # 错误的参数值.
 7
 8 if [ $# -ne "$ARGS" ] # 测试传递到脚本中的参数值.
 9 then
10   echo "Usage: `basename $0` side_1 side_2"
11   exit $E_BADARGS
12 fi
13
14
15 AWKSCRIPT=' { 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.
20 echo -n "Hypotenuse of $1 and $2 = "
21 echo $1 $2 | awk "$AWKSCRIPT"
22
23 exit 0
```

12.9. 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.

```
bash$ seq 5
1
 2
 3
 4
 5



bash$ seq -s : 5
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
 4 echo
 5
 6 for a in `seq 80`  # or   for a in $( seq 80 )
 7 # 与 for a in 1 2 3 4 5 ... 80   相同(少敲了好多字!).
 8 # 也可以使用'jot'(如果系统上有的话).
 9 do
10   echo -n "$a "
11 done       # 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
15 echo; echo
16
17
18 COUNT=80  # 当然, 'seq'也可以使用一个可替换的参数.
19
20 for a in `seq $COUNT`  # 或者   for a in $( seq $COUNT )
21 do
22   echo -n "$a "
23 done       # 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
17 E_OPTERR=65
18
19 if [ "$#" -eq 0 ]
20 then   # 脚本需要至少一个命令行参数.
21   echo "Usage $0 -[options a,b,c]"
22   exit $E_OPTERR
23 fi
24
25 set -- `getopt "abcd:" "$@"`
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
29 while [ ! -z "$1" ]
30 do
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!).

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:

```
bash$ yes $BASH_VERSION
3.00.16(1)-release
 3.00.16(1)-release
 3.00.16(1)-release
 3.00.16(1)-release
 3.00.16(1)-release
 . . .
```

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.

```
bash$ printenv | grep HOME
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了.

```
                    (重定向)
                    |----> 到文件
```

```
  ==========================|====================
  命令    --->    命令    ---> |tee ---> 命令 ---> ---> 管道的输出
  ================================================
```

```
  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
  6 file_subscript=copy
  7
  8 dd if=$0 of=$0.$file_subscript 2>/dev/null
  9 # 阻止dd产生的消息:              ^^^^^^^^^^^
 10
 11 exit $?
```

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
  7 input_file=$0    # 脚本自身.
  8 output_file=log.txt
  9 n=3
 10 p=5
 11
 12 dd if=$input_file of=$output_file bs=1 skip=$((n-1)) count=$((p-
n+1)) 2> /dev/null
 13 # 从脚本中把位置n到p的字符提取出来.
 14
 15 # ------------------------------------------------------
 16
 17 echo -n "hello world" | dd cbs=1 conv=unblock 2> /dev/null
```

```
18 # 垂直地echo "hello world".
19
20 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 $E_NOT_FOUND
 30 fi
 31
 32 echo; echo -n "Are you absolutely sure you want to blot out
\"$file\" (y/n)? "
 33 read answer
 34 case "$answer" in
 35 [nN]) echo "Changed your mind, huh?"
 36       exit $E_CHANGED_MIND
 37       ;;
 38 *)    echo "Blotting out file \"$file\".";;
 39 esac
 40
 41
 42 flength=$(ls -l "$file" | awk '{print $5}')  # 5是文件长度.
 43 pass_count=1
 44
 45 chmod u+w "$file"   # 允许覆盖/删除这个文件.
 46
 47 echo
 48
 49 while [ "$pass_count" -le "$PASSES" ]
 50 do
 51   echo "Pass #$pass_count"
 52   sync          # 刷新buffers.
 53   dd if=/dev/urandom of=$file bs=$BLOCKSIZE count=$flength
 54                 # 使用随机字节进行填充.
 55   sync          # 再次刷新buffer.
 56   dd if=/dev/zero of=$file bs=$BLOCKSIZE count=$flength
 57                 # 用0填充.
 58   sync          # 再次刷新buffer.
 59   let "pass_count += 1"
 60   echo
 61 done
 62
 63
 64 rm -f $file    # 最后, 删除这个已经被破坏得不成样子的文件.
 65 sync           # 最后一次刷新buffer.
 66
 67 echo "File \"$file\" blotted out and deleted."; echo
 68
 69
 70 exit 0
 71
 72 #  这是一种真正安全的删除文件的办法,
 73 #+ 但是效率比较低, 运行比较慢.
 74 #  GNU文件工具包中的"shred"命令,
 75 #+ 也可以完成相同的工作, 不过更有效率.
 76
 77 #  使用普通的方法是不可能重新恢复这个文件了.
 78 #  然而 . . .
 79 #+ 这个简单的例子是不能够抵抗
 80 #+ 那些经验丰富并且正规的分析.
 81
 82 #  这个脚本可能不会很好的运行在日志文件系统上(JFS).
 83 #  练习 (很难): 像它做的那样修正这个问题.
 84
 85
 86
 87 #  Tom Vier的文件删除包可以更加彻底的删除文件,
 88 #+ 比这个例子厉害的多.
 89 #     http://www.ibiblio.org/pub/Linux/utils/file/wipe-2.0.0.tar.bz2
 90
 91 #  如果想对安全删除文件这一论题进行深入的分析,
 92 #+ 可以参见Peter Gutmann的网页,
 93 #+      "Secure Deletion of Data From Magnetic and Solid-State
Memory".
 94 #       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).

```
bash$ objdump -d /bin/ls
/bin/ls:     file format elf32-i386

 Disassembly of section .init:

 080490bc <.init>:
  80490bc:       55                      push   %ebp
  80490bd:       89 e5                   mov    %esp,%ebp
  . . .
```

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". 这个命令还可以用来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 $2}')
 8   # 除了真正需要转换的部分保留下来外,其他的部分都去掉.
 9   echo "$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
 5 string=abcdA01
 6 echo "len($string)" | m4                           # 7
 7 echo "substr($string,4)" | m4                      # A01
 8 echo "regexp($string,[0-1][0-1],\&Z)" | m4         # 01Z
 9
10 # 算术操作
11 echo "incr(22)" | m4                               # 23
12 echo "eval(99 / 3)" | m4                           # 33
13
14 exit 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"*so*und e*x*change"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 Journal*1997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.

13. 系统与管理命令

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.

```
bash$ groups
bozita cdrom cdwriter audio xgrp

bash$ echo $GROUPS
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]

```
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.

```
bash$ id
```

```
uid=501(bozo) gid=501(bozo)
groups=501(bozo),22(cdrom),80(cdwriter),81(audio)

bash$ echo $UID
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.

```
bash$ who
bozo  tty1      Apr 27 17:45
 bozo  pts/0    Apr 27 17:46
 bozo  pts/1    Apr 27 17:47
 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.

```
bash$ who -m
localhost.localdomain!bozo  pts/2    Apr 27 17:49
```

whoami与who -m很相似，但是只列出用户名.

```
bash$ whoami
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.

```
bash$ w | grep startx
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.

```
bash$ logname
bozo

bash$ whoami
bozo
```

然而...

```
bash$ su
Password: ......

bash# whoami
root
bash# logname
bozo
```

logname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(*s*ubstitute *u*ser)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 $E_WRONG_USER
17 else
18   echo
19   echo "You should know better than to run this script, root."
20   echo "Even root users get the blues... "
21   echo
22 fi
23
24
25 username=bozo
26 NEWPASSWORD=security_violation
27
28 # 检查bozo是否在这里.
29 grep -q "$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.

```
bash$ ac
        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:

```
bash$ last reboot
reboot   system boot  2.6.9-1.667      Fri Feb  4 18:18
(00:02)
 reboot   system boot  2.6.9-1.667      Fri Feb  4 15:20
(01:27)
 reboot   system boot  2.6.9-1.667      Fri Feb  4 12:56
(00:49)
 reboot   system boot  2.6.9-1.667      Thu Feb  3 21:08
(02:17)
 . . .

 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.

```
bash$ tty
/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
  4 echo -n "What is your name? "
  5 read name                          #  试试用退格键
  6                                    #+ 来删除输入的字符.
  7                                    #  有什么问题?
  8 echo "Your name is $name."
  9
 10 stty erase '#'                     #  将"hashmark"(#)设置为退格字符.
 11 echo -n "What is your name? "
 12 read name                          #  使用#来删除最后键入的字符.
 13 echo "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. (译者: 可以使用stty
erase '^?'进行恢复)
 16
 17 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
  4 echo
  5 echo -n "Enter password "
  6 read passwd
```

```
  7 echo "password is $passwd"
  8 echo -n "If someone had been looking over your shoulder, "
  9 echo "your password would have been compromised."
 10
 11 echo && echo  # 在一个"与列表"中产生两个换行.
 12
 13
 14 stty -echo    # 关闭屏幕的echo.
 15
 16 echo -n "Enter password again "
 17 read passwd
 18 echo
 19 echo "password is $passwd"
 20 echo
 21
 22 stty echo     # 恢复屏幕的echo.
 23
 24 exit 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
  4 echo
  5
  6 old_tty_settings=$(stty -g)   # 保存老的设置(为什么?).
  7 stty -icanon
  8 Keypress=$(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.

终端与模式terminals and modes

一般情况下, 一个终端都是工作在*canonical*(标准)模式下. 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. 终端上的一个本地缓存保存了这些按键. 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. 这就意味着在终端内部存在一个基本的行编辑器.

```
bash$ stty -a
speed 9600 baud; rows 36; columns 96; line = 0;
 intr = ^C; quit = ^\; erase = ^H; kill = ^U; eof = ^D; eol =
<undef>; eol2 = <undef>;
 start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W; lnext =
^V; flush = ^O;
 ...
 isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -
echoprt
```

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.

```
bash$ cat > filexxx
wha<ctl-W>I<ctl-H>foo bar<ctl-U>hello world<ENTER>
<ctl-D>
```

```
bash$ cat filexxx
hello world
bash$ wc -c < filexxx
12
```

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.

在non-canonical(("raw")模式下, 每次按键(包括特殊定义的按键, 比如ctl-H)都将会立即发送一个字符到控制进程中.

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, 因为Bash的行编辑器更好. 比如, 当你在Bash提示符下敲ctl-A的时候, 终端将不会显示^A, 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.

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.

```
bash$ setterm -cursor off
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.

```
bash$ tset -r
Terminal type is xterm-xfree86.
 Kill is control-U (^U).
 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).

```
bash$ wall System going down for maintenance in 5 minutes!
Broadcast message from bozo (pts/1) Sun Jul  8 13:53:27 2001...

 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.

```
bash$ uname -a
Linux localhost.localdomain 2.2.15-2.5.0 #1 Sat Feb 5 00:13:43 EST 2000
i686 unknown

bash$ uname -s
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.

```
bash$ arch
i686

bash$ uname -m
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.

```
bash$ lastlog
root           tty1                      Fri Dec  7 18:43:21 -0700 2001
 bin                                      **Never logged in**
 daemon                                   **Never logged in**
 ...
 bozo            tty1                     Sat Dec  8 21:14:29 -0700
2001



bash$ lastlog | grep root
root           tty1                      Fri Dec  7 18:43:21 -0700 2001
```

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.

```
bash$ lsof
COMMAND    PID    USER   FD   TYPE     DEVICE    SIZE     NODE NAME
 init         1    root  mem    REG        3,5   30748    30303
```

```
/sbin/init
 init          1    root   mem    REG        3,5    73120     8069
/lib/ld-2.1.3.so
 init          1    root   mem    REG        3,5   931668     8075
/lib/libc-2.1.3.so
 cardmgr     213    root   mem    REG        3,5    36956    30357
/sbin/cardmgr
 ...
```

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.

```
bash$ strace df
execve("/bin/df", ["df"], [/* 45 vars */]) = 0
 uname({sys="Linux", node="bozo.localdomain", ...}) = 0
 brk(0)                                  = 0x804f5e4

 ...
```

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.

```
bash$ ltrace df
__libc_start_main(0x804a910, 1, 0xbfb589a4, 0x804fb70, 0x804fb68
<unfinished ...>:
 setlocale(6, "")                                  = "en_US.UTF-8"
bindtextdomain("coreutils", "/usr/share/locale") = "/usr/share/locale"
textdomain("coreutils")                           = "coreutils"
__cxa_atexit(0x804b650, 0, 0, 0x8052bf0, 0xbfb58908) = 0
getenv("DF_BLOCK_SIZE")                           = NULL

 ...
```

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.

```
bash$ nc localhost.localdomain 25
220 localhost.localdomain ESMTP Sendmail 8.13.1/8.13.1; 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. 噢, 他会被气死的.
  3 ## 参数: target port [port port port ...]
  4 ## 标准输出和标准输入被混到一块.
  5 ##
  6 ##  优点: 运行起来比ident-scan慢, 这样使远程机器inetd进程更不易注意而不会产生警告,
  7 ##+ 并且只有很少的知名端口会被指定.
  8 ##  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,
  9 ##+ 并且当远程服务监听在很高的端口时无法工作.
 10 # 脚本作者: Hobbit <hobbit@avian.org>
 11 # 已征得作者同意在ABS指南中使用.
 12
 13 # ------------------------------------------------------
 14 E_BADARGS=65        # 至少需要两个参数.
 15 TWO_WINKS=2         # 睡眠多长时间.
 16 THREE_WINKS=3
 17 IDPORT=113          # indent协议的认证端口.
 18 RAND1=999
 19 RAND2=31337
 20 TIMEOUT0=9
 21 TIMEOUT1=8
 22 TIMEOUT2=4
 23 # ------------------------------------------------------
 24
 25 case "${2}" in
 26   "" ) echo "Need HOST and at least one PORT." ; exit $E_BADARGS ;;
 27 esac
 28
 29 # 测试目标主机看是否运行了identd守护进程.
 30 nc -z -w $TIMEOUT0 "$1" $IDPORT || { echo "Oops, $1 isn't running identd." ; exit 0 ; }
 31 #  -z 选项扫描监听进程.
 32 #     -w $TIMEOUT = 尝试连接多长时间.
 33
 34 # 产生一个随机的本地起点源端口.
 35 RP=`expr $$ % $RAND1 + $RAND2`
 36
 37 TRG="$1"
 38 shift
 39
 40 while test "$1" ; do
 41   nc -v -w $TIMEOUT1 -p ${RP} "$TRG" ${1} < /dev/null > /dev/null &
 42   PROC=$!
 43   sleep $THREE_WINKS
 44   echo "${1},${RP}" | nc -w $TIMEOUT2 -r "$TRG" $IDPORT 2>&1
 45   sleep $TWO_WINKS
 46
 47 # 这看上去是不是像个残疾的脚本或是其他类似的东西... ?
 48 # ABS作者评注 : "这不是真的那么糟糕,
 49 #+                事实上, 做得非常聪明."
 50
 51   kill -HUP $PROC
 52   RP=`expr ${RP} + 1`
 53   shift
 54 done
 55
 56 exit $?
 57
 58 #  注意事项:
 59 #  ---------
 60
 61 #  试着把第30行去掉,
 62 #+ 然后以"localhost.localdomain 25"为参数来运行脚本.
 63
 64 #  关于Hobbit写的更多'nc'例子脚本,
 65 #+ 可以在以下文档中找到:
 66 #+ /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.

```
```

```
bash$ free
                total       used       free     shared    buffers
cached
   Mem:          30504      28624       1880      15820       1608
16376
   -/+ buffers/cache:      10640      19864
   Swap:         68540       3128      65412
```

打印出未使用的RAM内存:

```
bash$ free | grep Mem | awk '{ print $4 }'
1880
```

procinfo

从/proc pseudo-filesystem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.

```
bash$ procinfo | grep Bootup
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.

```
bash$ lsdev
Device            DMA   IRQ  I/O Ports
 ------------------------------------------------
 cascade             4     2
 dma                           0080-008f
 dma1                          0000-001f
 dma2                          00c0-00df
 fpu                           00f0-00ff
 ide0                       14  01f0-01f7 03f6-03f6
 ...
```

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.

```
bash$ du -ach
1.0k    ./wi.sh
 1.0k    ./tst.sh
 1.0k    ./random.file
 6.0k    .
 6.0k    total
```

df

使用列表的形式显示文件系统的使用状况.

```
bash$ df
Filesystem           1k-blocks      Used Available Use% Mounted on
 /dev/hda5               273262     92607    166547  36% /
 /dev/hda8               222525    123951     87085  59% /home
 /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.

```
bash$ dmesg | grep hda
Kernel command line: ro root=/dev/hda2
 hda: IBM-DLGA-23080, ATA DISK drive
 hda: 6015744 sectors (3080 MB) w/96KiB Cache, CHS=746/128/63
 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(*stat*istic).

```
```

```
bash$ stat test.cru
  File: "test.cru"
   Size: 49970        Allocated Blocks: 100          Filetype: Regular
File
   Mode: (0664/-rw-rw-r--)         Uid: (  501/ bozo)  Gid: (  501/
bozo)
 Device:  3,8   Inode: 18185     Links: 1
 Access: Sat Jun  2 16:40:24 2001
 Modify: Sat Jun  2 16:40:24 2001
 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.

```
bash$ stat nonexistent-file
nonexistent-file: No such file or directory
```

vmstat

显示虚拟内存的统计信息.

```
bash$ vmstat
   procs                      memory    swap          io system
cpu
 r   b   w   swpd   free   buff  cache   si   so    bi    bo   in    cs
us  sy id
 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(27)中的信息. 请参考例子 27-3.

netstat -r等价于route命令.

```
bash$ netstat
Active Internet connections (w/o servers)
 Proto Recv-Q Send-Q Local Address           Foreign Address
State
 Active UNIX domain sockets (w/o servers)
 Proto RefCnt Flags       Type       State         I-Node Path
 unix  11     [ ]         DGRAM                    906    /dev/log
 unix  3      [ ]         STREAM     CONNECTED     4514   /tmp/.X11-
unix/X0
 unix  3      [ ]         STREAM     CONNECTED     4513
 . . .
```

uptime

显示系统运行的时间, 还有其他的一些统计信息.

```
bash$ uptime
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.

```
bash$ hostname
localhost.localdomain

bash$ echo $HOSTNAME
localhost.localdomain
```

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.

```
bash$ hostid
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.

```
bash$ cat /etc/hosts
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的概要. 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.

```
bash$ sar
Linux 2.4.9 (brooks.seringas.fr)        09/26/03

10:30:00          CPU     %user     %nice   %system   %iowait     %idle
10:40:00          all      2.21     10.90     65.48      0.00
21.41
10:50:00          all      3.36      0.00     72.36      0.00
24.28
11:00:00          all      1.12      0.00     80.77      0.00
18.11
Average:          all      2.23      3.63     72.87      0.00
21.27

14:32:30          LINUX RESTART

15:00:00          CPU     %user     %nice   %system   %iowait     %idle
15:10:00          all      8.59      2.40     17.47      0.00
71.54
15:20:00          all      4.07      1.00     11.95      0.00
82.98
15:30:00          all      0.79      2.94      7.56      0.00
88.71
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.

```
bash$ readelf -h /bin/bash
ELF Header:
   Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
   Class:                             ELF32
   Data:                              2's complement, little endian
   Version:                           1 (current)
   OS/ABI:                            UNIX - System V
   ABI Version:                       0
   Type:                              EXEC (Executable file)
   . . .
```

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.

```
bash$ size /bin/bash
   text    data     bss     dec     hex filename
 495971   22496   17392  535859   82d33 /bin/bash
```

系统日志类

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(*P*rocess *S*tatistics): 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).

```
bash$  ps ax | grep sendmail
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.

```
bash$ ps a | grep mingetty
2212 tty2     Ss+    0:00 /sbin/mingetty tty2
 2213 tty3     Ss+    0:00 /sbin/mingetty tty3
 2214 tty4     Ss+    0:00 /sbin/mingetty tty4
 2215 tty5     Ss+    0:00 /sbin/mingetty tty5
 2216 tty6     Ss+    0:00 /sbin/mingetty tty6
 4849 pts/2    S+     0:00 grep mingetty


bash$ pgrep mingetty
2212 mingetty
```

```
 2213 mingetty
 2214 mingetty
 2215 mingetty
 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.

```
bash$ top -b
  8:30pm  up 3 min,  3 users,  load average: 0.49, 0.32, 0.13
 45 processes: 44 sleeping, 1 running, 0 zombie, 0 stopped
 CPU states: 13.6% user,  7.3% system,  0.0% nice, 78.9% idle
 Mem:    78396K av,   65468K used,   12928K free,       0K shrd,
2352K buff
 Swap:  157208K av,       0K used,  157208K free
37244K cached

   PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
   848 bozo      17   0   996  996   800 R     5.6  1.2   0:00 top
     1 root       8   0   512  512   444 S     0.0  0.6   0:04 init
     2 root       9   0     0    0     0 SW    0.0  0.0   0:00 keventd
   ...
```

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.

```
bash$ pidof xclock
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=xxxyyyzzz  # 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

或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.

```
bash$ fuser -u /usr/bin/vim
/usr/bin/vim:         3207e(bozo)



bash$ fuser -u /dev/null
/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.

```
bash$ umount /mnt/usbdrive
umount: /mnt/usbdrive: device is busy



bash$ fuser -um /dev/usbdrive
/mnt/usbdrive:        1772c(bozo)

bash$ kill -9 1772
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.

```
root# nmap localhost.localdomain
PORT     STATE SERVICE
 25/tcp   open  smtp



root# fuser -un tcp 25
25/tcp:               2095(root)

root# ps ax | grep 2095 | grep -v grep
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), 还是正处于重起状态(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.

```
root# /sbin/service iptables stop
Flushing firewall rules:                                   [  OK  ]
 Setting chains to policy ACCEPT: filter                    [  OK  ]
 Unloading iptables modules:                                [  OK  ]
```

网络类

ifconfig

网络的*接口配置*和调试工具.

```
bash$ ifconfig -a
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:10 errors:0 dropped:0 overruns:0 frame:0
          TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          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.

```
bash$ route
Destination     Gateway         Genmask         Flags   MSS Window
irtt Iface
 pm3-67.bozosisp *               255.255.255.255 UH        40 0
0 ppp0
 127.0.0.0       *               255.0.0.0       U         40 0
0 lo
 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.

```
bash$ chkconfig --list
atd             0:off   1:off   2:off   3:on    4:on    5:on    6:off
 rwhod           0:off   1:off   2:off   3:off   4:off   5:off   6:off
 ...
```

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.

```
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.

```
  1 umount /mnt/cdrom
  2 # 现在你可以按下弹出键(指的是cdrom或软盘驱动器上的弹出按键), 并安全的弹出光盘.
```

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. 但是如果在带有软躯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
 8 ROOT_UID=0      # 这个脚本必须以root身份运行.
 9 E_NOTROOT=67    # 非root用户将会产生这个错误.
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 # 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.

tune2fs

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.

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.

```
root# dumpe2fs /dev/hda7 | grep 'ount count'
dumpe2fs 1.19, 13-Jul-2000 for EXT2 FS 0.5b, 95/08/09
 Mount count:              6
 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.

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!

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.

```
root# lsusb
Bus 001 Device 001: ID 0000:0000
 Device Descriptor:
   bLength                18
   bDescriptorType         1
   bcdUSB               1.00
   bDeviceClass            9 Hub
   bDeviceSubClass         0
   bDeviceProtocol         0
   bMaxPacketSize0         8
   idVendor           0x0000
   idProduct          0x0000

   . . .
```

lspci

显示*pci*总线及其设备.

```
bash$ lspci
00:00.0 Host bridge: Intel Corporation 82845 845 (Brookdale) Chipset
Host Bridge (rev 04)
 00:01.0 PCI bridge: Intel Corporation 82845 845 (Brookdale) Chipset
AGP Bridge (rev 04)
 00:1d.0 USB Controller: Intel Corporation 82801CA/CAM USB (Hub #1)
(rev 02)
 00:1d.1 USB Controller: Intel Corporation 82801CA/CAM USB (Hub #2)
(rev 02)
 00:1d.2 USB Controller: Intel Corporation 82801CA/CAM USB (Hub #3)
(rev 02)
 00:1e.0 PCI bridge: Intel Corporation 82801 Mobile PCI Bridge (rev 42)

   . . .
```

mkbootdisk

创建启动软盘，启动盘可以唤醒系统，比如当MBR(master boot record主启动记录)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了.

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.

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). MAKEDEV工具事实上具

有mknod的全部功能, 而且更容易使用.

MAKEDEV

创建设备文件的工具. 必须在/dev目录下, 并且以root身份使用.

```
root# ./MAKEDEV
```

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.

```
bash$ lsmod
Module                  Size  Used by
 autofs                  9456   2 (autoclean)
 opl3                   11376   0
 serial_cs               5456   0 (unused)
 sb                     34752   0
 uart401                 6384   0 [sb]
 sound                  58368   0 [opl3 sb uart401]
 soundlow                 464   0 [sound]
 soundcore               2800   6 [sb sound]
 ds                      6448   2 [serial_cs]
 i82365                 22928   2
 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.

```
bash$ modinfo hid
filename:    /lib/modules/2.4.20-6/kernel/drivers/usb/hid.o
 description: "USB HID support drivers"
 author:      "Andreas Gal, Vojtech Pavlik <vojtech@suse.cz>"
 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 var1=value1 var2=value2 commandXXX
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 #! /usr/bin/env perl
2
3 print "This Perl script will run,\n";
4 print "even when I don't know where to find Perl.\n";
5
6 # 在不知道perl程序路径的时候,
7 # 这么写有利于跨平台移植.
8 # 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.

```
bash$ ldd /bin/ls
libc.so.6 => /lib/libc.so.6 (0x4000c000)
/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.

13.1. 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**

```
#!/bin/sh

# -->  本书作者所做的注释全部以"# -->"开头.

# --> 这是由Miquel van Smoorenburg所编写的
# --> 'rc'脚本包的一部分, <miquels@drinkel.nl.mugnet.org>.

# --> 这个特殊的脚本看起来是Red Hat/FC专用的,
# --> (在其它的发行版中可能不会出现).

#  停止所有正在运行的不必要的服务
#+ (不会有多少, 所以这是个合理性检查)

for i in /var/lock/subsys/*; do
        # --> 标准的for/in循环, 但是由于"do"在同一行上,
        # --> 所以必须添加";".
        # 检查脚本是否在那里.
        [ ! -f $i ] && continue
        # --> 这是一种使用"与列表"的聪明方法, 等价于:
        # --> if [ ! -f "$i" ]; then continue

        # 取得子系统的名字.
        subsys=${i#/var/lock/subsys/}
        # --> 匹配变量名, 在这里就是文件名.
        # --> 与subsys=`basename $i`完全等价.

        # -->  从锁定文件名中获得
        # -->+ (如果那里有锁定文件的话,
        # -->+ 那就证明进程正在运行).
        # -->  参考一下上边所讲的"锁定文件"的内容.


        # 终止子系统.
        if [ -f /etc/rc.d/init.d/$subsys.init ]; then
           /etc/rc.d/init.d/$subsys.init stop
        else
           /etc/rc.d/init.d/$subsys stop
        # -->  挂起运行的作业和幽灵进程.
        # -->  注意"stop"只是一个位置参数,
        # -->+ 并不是shell内建命令.
        fi
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. 看看你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.

14. 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.

```
script_name=`basename $0`
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. .

```
rm `cat filename`    # "filename"包含了需要被删除的文件列表.
#
# S. C. 指出，这种使用方法可能会产生"参数列表太长"的错误.
# 更好的方法是                xargs rm -- < filename
# ( -- 同时涵盖了某些特殊情况，这种特殊情况就是，以"-"开头的文件名会产生不良结果.)

textfile_listing=`ls *.txt`
# 变量中包含了当前工作目录下所有的*.txt文件.
echo $textfile_listing

textfile_listing2=$(ls *.txt)   # 这是命令替换的另一种形式.
echo $textfile_listing2
# 同样的结果.

# 如果将文件列表放入到一个字符串中的话，
# 可能会混入一个新行.
#
# 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.
#       shopt -s nullglob     # 如果不匹配，那就不进行文件名扩展.
#       textfile_listing=( *.txt )
#
# 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).

```
COMMAND `echo a b`      # 两个参数: a and b

COMMAND "`echo a b`"    # 1个参数: "a b"

COMMAND `echo`          # 无参数

COMMAND "`echo`"        # 一个空参数


# 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.

```
# cd "`pwd`"  # 这句总能正常运行.
# 然而...

mkdir 'dir with trailing newline
'

cd 'dir with trailing newline
'

cd "`pwd`"  # 错误消息:
# bash: cd: /tmp/file with trailing newline: No such file or directory

cd "$PWD"   # 运行良好.




old_tty_setting=$(stty -g)   # 保存旧的终端设置.
```

```
 20 echo "Hit a key "
 21 stty -icanon -echo            # 对终端禁用"canonical"模式,
 22                                # 这样的话, 也会禁用了*本地*的echo.
 23 key=$(dd bs=1 count=1 2> /dev/null)   #  使用'dd'命令来取得一个按键.
 24 stty "$old_tty_setting"       # 恢复旧的设置.
 25 echo "You hit ${#key} key."   # ${#variable} = number of characters in
$variable
 26 #
 27 # 除了回车, 你随便敲任何按键都会输出"You hit 1 key."
 28 # 如果敲回车, 那么输出就是"You hit 0 key."
 29 # 新行已经被命令替换吃掉了.
 30
 31 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>/dev/null|grep -E
"^I.*Cls=03.*Prot=02"`
 26     kbdoutput=`cat /proc/bus/usb/devices 2>/dev/null|grep -E
"^I.*Cls=03.*Prot=01"`
 27 ...
 28 fi
```

不要将一个*长*文本文件的全部内容设置到变量中, 除非你有一个非常好的原因非这么做不可, 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

```
#!/bin/bash
# stupid-script-tricks.sh: 朋友, 别在家试这个脚本.
# 来自于"Stupid Script Tricks," 卷I.


dangerous_variable=`cat /boot/vmlinuz`    # 这是压缩过的Linux内核自身.

echo "string-length of \$dangerous_variable = ${#dangerous_variable}"
# 这个字符串变量的长度是$dangerous_variable = 794151
# (不要使用as 'wc -c /boot/vmlinuz'来计算长度.)

# echo "$dangerous_variable"
# 千万别尝试这么做! 这样将挂起这个脚本.


#  脚本作者已经意识到将二进制文件设置到
#+ 变量中一点作用都没有.

exit 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

```
#!/bin/bash
# csubloop.sh: 将循环输出的内容设置到变量中.

variable1=`for i in 1 2 3 4 5
do
  echo -n "$i"                  #  对于在这里的命令替换来说
done`                           #+ 这个'echo'命令是非常关键的.

echo "variable1 = $variable1"   # variable1 = 12345


i=0
variable2=`while [ "$i" -lt 10 ]
do
  echo -n "$i"                  # 再来一个, 'echo'是必需的.
  let "i += 1"                  # 递增.
done`

echo "variable2 = $variable2"   # variable2 = 0123456789

#  这就证明了在一个变量的声明中
#+ 嵌入一个循环是可行的.

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.)

```
#include <stdio.h>

/*  "Hello, world." C program  */

int main()
{
  printf( "Hello, world." );
  return (0);
}
```

```
bash$ gcc -o hello hello.c
```

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

```
bash$ sh hello.sh
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.

```
bash$ echo `echo \\`

bash$ echo $(echo \\)
\
```

$(...)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, 但是还是原来的子母集合)

```
  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"yawl"文字表软件包中的一部分.
  7 #  http://ibiblio.org/pub/Linux/libs/yawl-0.3.2.tar.gz
  8 #  http://personal.riverusers.com/~thegrendel/yawl-0.3.2.tar.gz
  9
 10 E_NOARGS=66
 11 E_BADARG=67
 12 MINLEN=7
 13
 14 if [ -z "$1" ]
 15 then
 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}'\` `
```

15. 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

16. I/O重定向

目录
16.1. 使用exec
16.2. 代码块重定向
16.3. 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    1>>filename
 29       # 重定向并追加stdout到文件"filename".
 30    2>filename
 31       # 重定向stderr到文件"filename".
 32    2>>filename
 33       # 重定向并追加stderr到文件"filename".
 34    &>filename
 35       # 将stdout和stderr都重定向到文件"filename".
 36
 37    M>N
 38      # "M"是一个文件描述符, 如果没有明确指定的话默认为1.
 39      # "N"是一个文件名.
 40      # 文件描述符"M"被重定向到文件"N".
 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"是另一个文件描述符.
 44
 45
#==============================================================================
 46
 47       # 重定向stdout, 一次一行.
 48       LOGFILE=script.log
 49
 50       echo "This statement is sent to the log file, \"$LOGFILE\"." 1>$LOGFILE
 51       echo "This statement is appended to \"$LOGFILE\"." 1>>$LOGFILE
```

```
 52       echo "This statement is also appended to \"$LOGFILE\"." 1>>$LOGFILE
 53       echo "This statement is echoed to stdout, and will not appear in
\"$LOGFILE\"."
 54       # 每行过后, 这些重定向命令会自动"reset".
 55
 56
 57
 58       # 重定向stderr, 一次一行.
 59       ERRORFILE=script.errors
 60
 61       bad_command1 2>$ERRORFILE       #  Error message sent to $ERRORFILE.
 62       bad_command2 2>>$ERRORFILE      #  Error message appended to $ERRORFILE.
 63       bad_command3                    #  Error message echoed to stderr,
 64                                        #+ and does not appear in $ERRORFILE.
 65       # 每行过后, 这些重定向命令也会自动"reset".
 66
#=============================================================================
 67
 68
 69
 70    2>&1
 71       # 重定向stderr到stdout.
 72       # 将错误消息的输出, 发送到与标准输出所指向的地方.
 73
 74    i>&j
 75       # 重定向文件描述符i到j.
 76       # 指向i文件的所有输出都发送到j.
 77
 78    >&j
 79       # 默认的, 重定向文件描述符1(stdout)到j.
 80       # 所有传递到stdout的输出都送到j中去.
 81
 82    0< FILENAME
 83     < FILENAME
 84       # 从文件中接受输入.
 85       # 与">"是成对命令, 并且通常都是结合使用.
 86       #
 87       # grep search-word <filename
 88
 89
 90    [j]<>filename
 91       # 为了读写"filename", 把文件"filename"打开, 并且将文件描述符"j"分配给它.
 92       # 如果文件"filename"不存在, 那么就创建它.
 93       # 如果文件描述符"j"没指定, 那默认是fd 0, stdin.
 94       #
 95       # 这种应用通常是为了写到一个文件中指定的地方.
 96       echo 1234567890 > File    # 写字符串到"File".
 97       exec 3<> File             # 打开"File"并且将fd 3分配给它.
 98       read -n 4 <&3             # 只读取4个字符.
 99       echo -n . >&3             # 写一个小数点.
100       exec 3>&-                 # 关闭fd 3.
101       cat File                  # ==> 1234.67890
102       # 随机访问.
103
104
105
106    |
107       # 管道.
108       # 通用目的处理和命令链工具.
109       # 与">", 很相似, 但是实际上更通用.
110       # 对于想将命令, 脚本, 文件和程序串连起来的时候很有用.
111       cat *.txt | sort | uniq > result-file
112       # 对所有.txt文件的输出进行排序, 并且删除重复行.
113       # 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. 也可以的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.

16.1. 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**

```
#!/bin/bash
# 使用'exec'重定向stdin.


exec 6<&0          # 将文件描述符#6与stdin链接起来.
                   # 保存stdin.

exec < data-file   # stdin被文件"data-file"所代替.

read a1            # 读取文件"data-file"的第一行.
read a2            # 读取文件"data-file"的第二行.

echo
echo "Following lines read from file."
echo "-------------------------------"
echo $a1
echo $a2

echo; echo; echo

exec 0<&6 6<&-
#  现在将stdin从fd #6中恢复, 因为刚才我们把stdin重定向到#6了,
#+ 然后关闭fd #6 ( 6<&- ), 好让这个描述符继续被其他进程所使用.
#
# <&6 6<&-     这么做也可以.

echo -n "Enter data  "
read b1  # 现在"read"已经恢复正常了, 就是能够正常的从stdin中读取.
echo "Input read from stdin."
echo "----------------------"
echo "b1 = $b1"

echo

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**

```
#!/bin/bash
# reassign-stdout.sh

LOGFILE=logfile.txt

exec 6>&1           # 将fd #6与stdout链接起来.
                    # 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stout.
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.
```

16.2. 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), 最好让重定向循环运行在当前工作区内,
 7 #+ 这样的话, 需要提前进行文件描述符重定向,
 8 #+ 因为变量如果在(子shell上运行的)循环中被修改的话, 循环结束后并不会保存修改后的值.
```

```
 9
10
11 if [ -z "$1" ]
12 then
13   Filename=names.data      # 如果没有指定文件名则使用默认值.
14 else
15   Filename=$1
16 fi
17
18
19 exec 3<&0                  # 将stdin保存到文件描述符3.
20 exec 0<"$Filename"         # 重定向标准输入.
21
22 count=0
23 echo
24
25
26 while [ "$name" != Smith ]
27 do
28   read name                # 从stdin(现在已经是$Filename了)中读取.
29   echo $name
30   let "count += 1"
31 done                       #  从文件$Filename中循环读取
32                            #+ 因为文件(译者注: 指默认文件, 在本节最后)有20行.
33
34 #  这个脚本原先在"while"循环的结尾还有一句:
35 #+       done <"$Filename"
36 #  练习:
37 #  为什么不需要这句了?
38
39
40 exec 0<&3                  # 恢复保存的stdin.
41 exec 3<&-                  # 关闭临时文件描述符3.
42
43 echo; echo "$count names read"; echo
44
45 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
 4 if [ -z "$1" ]
 5 then
 6   Filename=names.data          # 如果没有指定文件名那就使用默认值.
 7 else
 8   Filename=$1
 9 fi
10
11 # while [ "$name" != Smith ]
12 until [ "$name" = Smith ]     # 把!=改为=.
13 do
14   read name                   # 从$Filename中读取, 而不是从stdin中读取.
15   echo $name
16 done <"$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
21 exit 0
```

例子 16-8. 重定向*for*循环

```
 1 #!/bin/bash
 2
 3 if [ -z "$1" ]
 4 then
 5   Filename=names.data           # 如果没有指定文件名就使用默认值.
 6 else
 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...
```

```
#+ 当然, 你得足够聪明, 才能看得出来.
#
# 更简洁的写法是      line_count=$(wc -l < "$Filename")


for name in `seq $line_count`  # "seq"打印出数字序列.
# while [ "$name" != Smith ]   --   比"while"循环更复杂   --
do
  read name                    # 从$Filename中, 而非从stdin中读取.
  echo $name
  if [ "$name" = Smith ]       # 因为用for循环, 所以需要这个多余测试.
  then
    break
  fi
done <"$Filename"              # 重定向stdin到文件$Filename.
#    ^^^^^^^^^^^^

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)

```
#!/bin/bash

if [ -z "$1" ]
then
  Filename=names.data          # 如果没有指定文件名, 则使用默认值.
else
  Filename=$1
fi

Savefile=$Filename.new         # 保存最终结果的文件名.
FinalName=Jonah                # 终止"read"时的名称.

line_count=`wc $Filename | awk '{ print $1 }'`  # 目标文件的行数.


for name in `seq $line_count`
do
  read name
  echo "$name"
  if [ "$name" = "$FinalName" ]
  then
    break
  fi
done < "$Filename" > "$Savefile"     # 重定向stdin到文件$Filename,
#    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^    并且将它保存到备份文件中.

exit 0
```

例子 16-10. 重定向*if/then*测试结构

```
#!/bin/bash

if [ -z "$1" ]
then
  Filename=names.data    # 如果文件名没有指定, 使用默认值.
else
  Filename=$1
fi

TRUE=1

if [ "$TRUE" ]           # if true    和   if :   都可以.
then
 read name
 echo $name
fi <"$Filename"
#  ^^^^^^^^^^^^

# 只读取了文件的第一行.
# 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.

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.

16.3. 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

```
#!/bin/bash
# logevents.sh, 由Stephane Chazelas所编写.

# 把事件记录在一个文件中.
# 必须以root身份运行 (这样才有权限访问/var/log).

ROOT_UID=0     # 只有$UID值为0的用户才具有root权限.
E_NOTROOT=67   # 非root用户的退出错误.


if [ "$UID" -ne "$ROOT_UID" ]
then
  echo "Must be root to run this script."
  exit $E_NOTROOT
fi


FD_DEBUG1=3
FD_DEBUG2=4
FD_DEBUG3=5

# 去掉下边两行注释中的一行, 来激活脚本.
# LOG_EVENTS=1
# LOG_VARS=1


log()  # 把时间和日期写入日志文件.
{
echo "$(date)  $*" >&7      # 这会把日期*附加*到文件中.
                            # 参考下边的代码.
}



case $LOG_LEVEL in
 1) exec 3>&2          4> /dev/null 5> /dev/null;;
 2) exec 3>&2          4>&2         5> /dev/null;;
 3) exec 3>&2          4>&2         5>&2;;
 *) exec 3> /dev/null 4> /dev/null 5> /dev/null;;
esac

FD_LOGVARS=6
if [[ $LOG_VARS ]]
then exec 6>> /var/log/vars.log
else exec 6> /dev/null               # 丢弃输出.
fi

FD_LOGEVENTS=7
if [[ $LOG_EVENTS ]]
then
  # then exec 7 >(exec gawk '{print strftime(), $0}' >> /var/log/event.log)
  # 上面这行不能在2.04版本的Bash上运行.
  exec 7>> /var/log/event.log        # 附加到"event.log".
  log                                    # 记录日期与时间.
else exec 7> /dev/null                   # 丢弃输出.
fi

echo "DEBUG3: beginning" >&${FD_DEBUG3}

ls -l >&5 2>&4                        # command1 >&5 2>&4

echo "Done"                                 # command2

echo "sending mail" >&${FD_LOGEVENTS}    # 将字符串"sending mail"写到文件描述符#7.


```

```
67 exit 0
```

17. 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
6 E_BADARGS=65
7
```

```
 8 if [ -z "$1" ]
 9 then
10   echo "Usage: `basename $0` filename"
11   exit $E_BADARGS
12 fi
13
14 TARGETFILE=$1
15
16 # 在文件中插入两行, 然后保存.
17 #--------Begin here document-----------#
18 vi $TARGETFILE <<x23LimitStringx23
19 i
20 This is line 1 of the example file.
21 This is line 2 of the example file.
22 ^[
23 ZZ
24 x23LimitStringx23
25 #----------End here document-----------#
26
27 #  注意上边^[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, (译者注: Bram Moolenaar是vim作者)
31 #+ 因为可能会存在终端相互影响的问题.
32
33 exit 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
 7 cat <<End-of-message
 8 -------------------------------------
 9 This is line 1 of the message.
10 This is line 2 of the message.
11 This is line 3 of the message.
12 This is line 4 of the message.
13 This is the last line of the message.
14 -------------------------------------
15 End-of-message
16
17 #  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
22 exit 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 (<<-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 docutment来说,
 5 #+ <<-可以抑制文档体前边的tab,
 6 #+ 而*不*是空格.
 7
 8 cat <<-ENDOFMESSAGE
 9     This is line 1 of the message.
10     This is line 2 of the message.
11     This is line 3 of the message.
12     This is line 4 of the message.
13     This is the last line of the message.
14 ENDOFMESSAGE
15 # 脚本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
23 exit 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
```

```
# 注意上边的空行也打印输出,
# 而上边那行"注释"当然也会打印到输出.
# (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)
exit 0
```

这是一个非常有用的脚本, 其中使用了包含参数替换的here document.

例子 17-6. 上传一个文件对到"Sunsite"的incoming目录

```
#!/bin/bash
# upload.sh

#  上传这一对文件(Filename.lsm, Filename.tar.gz)
#+ 到Sunsite/UNC (ibiblio.org)的incoming目录.
#  Filename.tar.gz是自身的tar包.
#  Filename.lsm是描述文件.
#  Sunsite需要"lsm"文件, 否则就拒绝上传.


E_ARGERROR=65

if [ -z "$1" ]
then
  echo "Usage: `basename $0` Filename-to-upload"
  exit $E_ARGERROR
fi


Filename=`basename $1`            # 从文件名中去掉目录字符串.

Server="ibiblio.org"
Directory="/incoming/Linux"
#  在这里也不一定非得将上边的参数写死在这个脚本中,
#+ 可以使用命令行参数的方法来替换.

Password="your.e-mail.address"    # 可以修改成相匹配的密码.

ftp -n $Server <<End-Of-Session
# -n选项禁用自动登录.

user anonymous "$Password"
binary
bell                              # 在每个文件传输后, 响铃.
cd $Directory
put "$Filename.lsm"
put "$Filename.tar.gz"
bye
End-Of-Session

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

```
#!/bin/bash
#  一个使用'cat'的here document, 但是禁用了参数替换.

NAME="John Doe"
RESPONDENT="the author of this fine script"

cat <<'Endofmessage'

Hello, there, $NAME.
Greetings to you, $NAME, from $RESPONDENT.

Endofmessage

#  如果"limit string"被引用或转义的话, 那么就禁用了参数替换.
#  下边的两种方式具有相同的效果.
#  cat <<"Endofmessage"
#  cat <<\Endofmessage

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

```
#!/bin/bash
# generate-script.sh
# 这个脚本的诞生基于Albert Reiner的一个主意.

OUTFILE=generated.sh          # 所产生文件的名字.


# ------------------------------------------------------------
# 'Here document包含了需要产生的脚本的代码.
(
cat <<'EOF'
#!/bin/bash

echo "This is a generated shell script."
#  Note that since we are inside a subshell,
#+ we can't access variables in the "outside" script.

echo "Generated file will be named: $OUTFILE"
#  Above line will not work as normally expected
#+ because parameter expansion has been disabled.
#  Instead, the result is literal output.

a=7
b=3

let "c = $a * $b"
echo "c = $c"

exit 0
EOF
) > $OUTFILE
# ------------------------------------------------------------

#  将'limit string'引用起来将会阻止上边
#+ here document消息体中的变量扩展.
#  这会使得输出文件中的内容保持here document消息体中的原文.

if [ -f "$OUTFILE" ]
then
  chmod 755 $OUTFILE
  # 让所产生的文件具有可执行权限.
else
  echo "Problem in creating file: \"$OUTFILE\""
fi

#  这个方法也可以用来产生
#+ C程序代码, Perl程序代码, Python程序代码, makefile,
#+ 和其他的一些类似的代码.
#  (译者注: 中间一段没译的注释将会被here document打印出来)
exit 0
```

也可以将here document的输出保存到变量中.

```
variable=$(cat <<SETVAR
This variable
runs over multiple lines.
SETVAR)

echo "$variable"
```

A here document can supply input to a function in the same script.

例子 17-9. Here document与函数

```
#!/bin/bash
# here-function.sh

GetPersonalData ()
{
  read firstname
```

```
  read lastname
  read address
  read city
  read state
  read zipcode
} # 这个函数看起来就是一个交互函数, 但是...


# 给上边的函数提供输入.
GetPersonalData <<RECORD001
Bozo
Bozeman
2726 Nondescript Dr.
Baltimore
MD
21226
RECORD001


echo
echo "$firstname $lastname"
echo "$address"
echo "$city, $state $zipcode"
echo

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

```
#!/bin/bash

: <<TESTVARIABLES
${HOSTNAME?}${USER?}${MAIL?}  # 如果其中某个变量没被设置, 那么就打印错误信息.
TESTVARIABLES

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

```
#!/bin/bash
# commentblock.sh

: <<COMMENTBLOCK
echo "This line will not echo."
This is a comment line missing the "#" prefix.
This is another comment line missing the "#" prefix.

&*@!!++=
The above line will cause no error message,
because the Bash interpreter will ignore it.
COMMENTBLOCK

echo "Exit value of above \"COMMENTBLOCK\" is $?."   # 0
# 这里将不会显示任何错误.


#  上边的这种技术当然也可以用来注释掉
#+ 一段正在使用的代码, 如果你有某些特定调试要求的话.
#  这比在每行前边都敲入"#"来得方便的多,
#+ 而且如果你想恢复的话, 还得将添加上的"#"删除掉.

: <<DEBUGXXX
for file in *
do
 cat "$file"
done
DEBUGXXX

```

```
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
 5 DOC_REQUEST=70
 6
 7 if [ "$1" = "-h"  -o "$1" = "--help" ]     # 请求帮助.
 8 then
 9   echo; echo "Usage: $0 [directory-name]"; echo
10   sed --silent -e '/DOCUMENTATIONXX$/,/^DOCUMENTATIONXX$/p' "$0" |
11   sed -e '/DOCUMENTATIONXX$/d'; exit $DOC_REQUEST; fi
12
13
14 : <<DOCUMENTATIONXX
15 List the statistics of a specified directory in tabular format.
16 ---------------------------------------------------------------
17 The command line parameter gives the directory to be listed.
18 If no directory specified or directory specified cannot be read,
19 then list the current working directory.
20
21 DOCUMENTATIONXX
22
23 if [ -z "$1" -o ! -r "$1" ]
24 then
25   directory=.
26 else
27   directory="$1"
28 fi
29
30 echo "Listing of "$directory":"; echo
31 (printf "PERMISSIONS LINKS OWNER GROUP SIZE MONTH DAY HH:MM PROG-NAME\n" \
32 ; ls -l "$directory" | sed 1d) | column -t
33
34 exit 0
```

使用cat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.

```
bash$ bash -c 'lsof -a -p $$ -d0' << EOF
> EOF
lsof    1213 bozo    0r   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.

17.1. 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.

```
String="This is a string of words."

read -r -a Words <<< "$String"
#  "read"命令的-a选项
#+ 将会把结果值按顺序的分配给数组中的每一项.

echo "First word in String is:    ${Words[0]}"   # This
echo "Second word in String is:   ${Words[1]}"   # is
echo "Third word in String is:    ${Words[2]}"   # a
echo "Fourth word in String is:   ${Words[3]}"   # string
echo "Fifth word in String is:    ${Words[4]}"   # of
echo "Sixth word in String is:    ${Words[5]}"   # words.
echo "Seventh word in String is:  ${Words[6]}"   # (null)
                                                  # $String的结尾.

# 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

```
#!/bin/bash
# prepend.sh: 在文件的开头添加文本.
#
#  Kenny Stauffer所捐助的脚本例子,
#+ 本文作者对这个脚本进行了少量修改.


E_NOSUCHFILE=65

read -p "File: " file   #  'read'命令的-p参数用来显示提示符.
if [ ! -e "$file" ]
then   # 如果这个文件不存在, 那就进来.
  echo "File $file not found."
  exit $E_NOSUCHFILE
fi

read -p "Title: " title
cat - $file <<<$title > $file.new

echo "Modified file is $file.new"

exit 0

# 下边是'man bash'中的一段:
# Here String
#   here document的一种变形, 形式如下:
#
#           <<<word
#
#   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

```
#!/bin/bash
#  由Francisco Lobo所提供的脚本,
#+ 本文作者进行了少量修改和注释.
#  并且经过授权, 可以使用在本书中.(感谢你!)

# 这个脚本不能运行于比Bash version 3.0更低的版本中.


E_MISSING_ARG=67
if [ -z "$1" ]
then
```

```
 12   echo "Usage: $0 mailbox-file"
 13   exit $E_MISSING_ARG
 14 fi
 15
 16 mbox_grep()  # 分析邮箱文件.
 17 {
 18     declare -i body=0 match=0
 19     declare -a date sender
 20     declare mail header value
 21
 22
 23     while IFS= read -r mail
 24 #         ^^^^                          重新设置$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           #                           ^^^  "here string"
 43
 44           case "$header" in
 45           [Ff][Rr][Oo][Mm] ) [[ $value =~ "$2" ]] && (( match++ )) ;;
 46           # 匹配"From"行.
 47           [Dd][Aa][Tt][Ee] ) read -r -a date <<< "$value" ;;
 48           #                                   ^^^
 49           # 匹配"Date"行.
 50           [Rr][Ee][Cc][Ee][Ii][Vv][Ee][Dd] ) read -r -a sender <<< "$value" ;;
 51           #                                                   ^^^
 52           # 匹配IP地址(可能被欺骗).
 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.

---

18. 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

19. 正则表达式

. . . 与软件开发相关的一些需要思想的行为，主要取决于你对问题的深刻见解.

Stowe Boyd

目录
19.1. 一份简要的正则表达式介绍
19.2. 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.

19.1. 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．具个例子，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.

```
bash$ cat textfile
This is line 1, of which there is only one instance.
 This is the only instance of line 2.
 This is line 3, another line.
 This is line 4.


bash$ grep 'the' textfile
This is line 1, of which there is only one instance.
 This is the only instance of line 2.
 This is line 3, another line.


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

```
bash$ grep "1133*" tstfile
Run   grep "1133*"  on this file.                  # 匹配.
 This line contains the number 113.                 # 匹配.
 This line contains the number 1133.                # 匹配.
 This line contains the number 113312.              # 匹配.
 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, 仅仅是个技巧而以.

  "[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).

  ```
  bash$ echo 2222 | gawk --re-interval '/2{3}/'
  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.

- 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.

```
bash$ egrep 're(a|e)d' misc.txt
People who read seem to be better informed than those who do not.
 The clarinet produces sound by the vibration of its reed.
```

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:]

  这是另外一种, 用于指定匹配字符范围的方法.

- [: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([[ ]])引起来.

  ```
  bash$ grep [[:digit:]] test.file
  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.

```
bash$ ls -l ?[[:digit:]][[:digit:]]?
-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.

```
 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
```

19.2. 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.

```
bash$ ls -l
total 2
 -rw-rw-r--    1 bozo  bozo          0 Aug  6 18:42 a.1
 -rw-rw-r--    1 bozo  bozo          0 Aug  6 18:42 b.1
 -rw-rw-r--    1 bozo  bozo          0 Aug  6 18:42 c.1
 -rw-rw-r--    1 bozo  bozo        466 Aug  6 17:48 t2.sh
 -rw-rw-r--    1 bozo  bozo        758 Jul 30 09:02 test1.txt

bash$ ls -l t?.sh
-rw-rw-r--    1 bozo  bozo        466 Aug  6 17:48 t2.sh

bash$ ls -l [ab]*
-rw-rw-r--    1 bozo  bozo          0 Aug  6 18:42 a.1
 -rw-rw-r--    1 bozo  bozo          0 Aug  6 18:42 b.1

bash$ ls -l [a-c]*
-rw-rw-r--    1 bozo  bozo          0 Aug  6 18:42 a.1
 -rw-rw-r--    1 bozo  bozo          0 Aug  6 18:42 b.1
 -rw-rw-r--    1 bozo  bozo          0 Aug  6 18:42 c.1

bash$ ls -l [^ab]*
-rw-rw-r--    1 bozo  bozo          0 Aug  6 18:42 c.1
 -rw-rw-r--    1 bozo  bozo        466 Aug  6 17:48 t2.sh
 -rw-rw-r--    1 bozo  bozo        758 Jul 30 09:02 test1.txt

bash$ ls -l {b*,c*,*est*}
-rw-rw-r--    1 bozo  bozo          0 Aug  6 18:42 b.1
 -rw-rw-r--    1 bozo  bozo          0 Aug  6 18:42 c.1
 -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.

```
bash$ echo *
a.1 b.1 c.1 t2.sh test1.txt

bash$ echo t*
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.

```
~/[.]bashrc     #   不能扩展成~/.bashrc
~/?bashrc       #   也不能扩展.
                #   通配(globbing)中所使用的通配符
                #+  和匹配字符串"不能"扩展"点".

~/.[b]ashrc     #   可以扩展成~/.bashrc
~/.ba?hrc       #   也可以扩展.
~/.bashr*       #   也可以扩展.

# 可以设置"dotglob"选项, 把这个特性关闭.

# 感谢, S.C.
```

20. 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

```
#!/bin/bash
# subshell.sh

echo

echo "Subshell level OUTSIDE subshell = $BASH_SUBSHELL"
# Bash, 版本3, 添加了这个新的            $BASH_SUBSHELL 变量.
echo

outer_variable=Outer

(
echo "Subshell level INSIDE subshell = $BASH_SUBSHELL"
inner_variable=Inner

echo "From subshell, \"inner_variable\" = $inner_variable"
echo "From subshell, \"outer\" = $outer_variable"
)

echo
echo "Subshell level OUTSIDE subshell = $BASH_SUBSHELL"
echo

if [ -z "$inner_variable" ]
then
  echo "inner_variable undefined in main body of shell"
else
  echo "inner_variable defined in main body of shell"
fi

echo "From main body of shell, \"inner_variable\" = $inner_variable"
#  $inner_variable将被作为未初始化的变量，被显示出来,
#+ 这是因为变量是在子shell里定义的"局部变量".
#  还有补救的办法么?

echo

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

```
#!/bin/bash
# allprofs.sh: 打印所有用户的配置文件

# 由Heiner Steven编写, 并由本书作者进行了修改.

FILE=.bashrc  #  在原始脚本中, File containing user profile,
              #+ 包含用户profile的是文件".profile".

for home in `awk -F: '{print $6}' /etc/passwd`
do
  [ -d "$home" ] || continue    # 如果没有home目录, 跳出本次循环.
  [ -r "$home" ] || continue    # 如果home目录没有读权限, 跳出本次循环.
  (cd $home; [ -e $FILE ] && less $FILE)
done

#  当脚本结束时, 不必使用'cd'命令返回原来的目录.
#+ 因为'cd $home'是在子shell中发生的, 不影响父shell.

exit 0
```

子shell可用于为一组命令设置一个"独立的临时环境".

```
COMMAND1
COMMAND2
COMMAND3
(
  IFS=:
  PATH=/bin
  unset TERMINFO
  set -C
  shift 5
  COMMAND4
  COMMAND5
  exit 3 # 只是从子shell退出.
)
# 父shell不受任何影响, 并且父shell的环境也没有被更改.
COMMAND6
COMMAND7
```

子shell的另一个应用, 是可以用来检测一个变量是否被定义.

```
if (set -u; : $variable) 2> /dev/null
then
  echo "Variable is set."
fi     #  变量已经在当前脚本中被设置,
       #+ 或者是一个Bash的内建变量,
       #+ 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).

# 也可以写成             [[ ${variable-x} != x || ${variable-y} != y ]]
# 或                       [[ ${variable-x} != x$variable ]]
# 或                       [[ ${variable+x} = x ]]
# 或                       [[ ${variable-x} != x ]]
```

子shell还可以用来检测一个加锁的文件:

```
if (set -C; : > lock_file) 2> /dev/null
then
  :   # lock_file不存在, 还没有用户运行这个脚本
else
  echo "Another user is already running that script."
exit 65
fi

#  这段程序由Stephane Chazelas所编写,
#+ 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

```
    (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; }
```

21. 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

```
#!/bin/bash

#  脚本开头以"#!/bin/bash -r"来调用,
#+ 会使整个脚本在受限模式下运行.

echo

echo "Changing directory."
cd /usr/local
echo "Now in `pwd`"
echo "Coming back home."
cd
echo "Now in `pwd`"
echo

# 非受限的模式下, 所有操作都正常.

set -r
# set --restricted    也具有相同的效果.
echo "==> Now in restricted mode. <=="

echo
echo

echo "Attempting directory change in restricted mode."
cd ..
echo "Still in `pwd`"

echo
echo

echo "\$SHELL = $SHELL"
echo "Attempting to change shell in restricted mode."
SHELL="/bin/ash"
echo
echo "\$SHELL= $SHELL"

echo
echo

```

```
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
```

---

22. 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"<"或">"与圆括号之间是*没有*空格的. 如果加了空格, 会产生错误.

```
bash$ echo >(true)
/dev/fd/63

bash$ echo <(true)
/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.

```
bash$ comm <(ls -l) <(ls -al)
total 12
-rw-rw-r--    1 bozo bozo       78 Mar 10 12:58 File0
-rw-rw-r--    1 bozo bozo       42 Mar 10 12:58 File2
-rw-rw-r--    1 bozo bozo      103 Mar 10 12:58 t2.sh
        total 20
        drwxrwxrwx    2 bozo bozo     4096 Mar 10 18:10 .
        drwx------   72 bozo bozo     4096 Mar 10 17:58 ..
        -rw-rw-r--    1 bozo bozo       78 Mar 10 12:58 File0
        -rw-rw-r--    1 bozo bozo       42 Mar 10 12:58 File2
        -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.
```

```
#
# 这种效果可以被模拟出来.
#
bzip2 -c < pipe > file.tar.bz2&
tar cf pipe $directory_name
rm pipe
#        或
exec 3>&1
tar cf /dev/fd/4 $directory_name 4>&1 >&3 3>&- | bzip2 -c > file.tar.bz2 3>&-
exec 3>&-


# 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.

```
# 摘自SuSE发行版中的代码片断:

while read  des what mask iface; do
# 这里省略了一些命令...
done < <(route -n)


# 为了测试它, 我们让它做点事.
while read  des what mask iface; do
  echo $des $what $mask $iface
done < <(route -n)

# 输出:
# Kernel IP routing table
# Destination Gateway Genmask Flags Metric Ref Use Iface
# 127.0.0.0 0.0.0.0 255.0.0.0 U 0 0 0 lo



# 就像Stephane Chazelas所给出的那样, 一个更容易理解的等价代码是:
route -n |
  while read des what mask iface; do   # 管道的输出被赋值给了变量.
    echo $des $what $mask $iface
  done  #  这将产生出与上边相同的输出.
        #  然而, Ulrich Gayer指出 . . .
        #+ 这个简单的等价版本在while循环中使用了一个子shell,
        #+ 因此当管道结束后, 变量就消失了.



#  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,
#+ 如下所示.

(
route -n | while read x; do ((y++)); done
echo $y # $y 仍然没有被声明或设置

while read x; do ((y++)); done < <(route -n)
echo $y # $y 的值为route -n的输出行数.
)

# 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)
(
: | x=x
# 看上去是启动了一个子shell
: | ( x=x )
# 但
x=x < <(:)
# 其实不是
)

# 当你要解析csv或类似东西的时侯, 这非常有用.
# 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.

23. 函数

目录
23.1. 复杂函数和函数复杂性
23.2. 局部变量
23.3. 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*.

例子 23-1. 简单函数

```
#!/bin/bash

JUST_A_SECOND=1

funky ()
{ # 这是一个最简单的函数.
  echo "This is a funky function."
  echo "Now exiting funky function."
} # 函数必须在调用前声明.


fun ()
{ # 一个稍微复杂一些的函数.
  i=0
  REPEATS=30

  echo
  echo "And now the fun really begins."
  echo

  sleep $JUST_A_SECOND    # 嘿，暂停一秒!
  while [ $i -lt $REPEATS ]
  do
    echo "----------FUNCTIONS---------->"
    echo "<------------ARE-------------"
    echo "<------------FUN------------>"
    echo
    let "i+=1"
  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.
```

23.1. 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
 4 DEFAULT=default                                  # 默认参数值.
 5
 6 func2 () {
 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 \"$1\".-"
12    fi
13
14    variable=${1-$DEFAULT}                        #  这里的参数替换
15    echo "variable = $variable"                   #+ 表示什么?
16                                                  #  ----------------------------
17                                                  #  为了区分没有参数的情况,
18                                                  #+ 和只有一个null参数的情况.
19
20    if [ "$2" ]
21    then
22      echo "-Parameter #2 is \"$2\".-"
23    fi
24
25    return 0
26 }
27
28 echo
29
30 echo "Nothing passed."
31 func2                            # 不带参数调用
32 echo
33
34
35 echo "Zero-length parameter passed."
36 func2 ""                         # 使用0长度的参数进行调用
37 echo
38
39 echo "Null parameter passed."
40 func2 "$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

```
#!/bin/bash
# func-cmdlinearg.sh
#  调用这个脚本, 并且带一个命令行参数.
#+ 类似于 $0 arg1.


func ()
{
echo "$1"
}

echo "First call to function: no arg passed."
echo "See if command-line arg is seen."
func
# 不行! 命令行参数不可见.

echo "============================================================="
echo
echo "Second call to function: command-line arg passed explicitly."
func $1
# 现在可见了!

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

```
#!/bin/bash
# ind-func.sh: 将一个间接引用传递给函数.

echo_var ()
{
echo "$1"
}

message=Hello
Hello=Goodbye

echo_var "$message"         # Hello
# 现在, 让我们传递一个间接引用给函数.
echo_var "${!message}"      # Goodbye

echo "-------------"

# 如果我们改变"hello"变量的值会发生什么?
Hello="Hello, again!"
echo_var "$message"         # Hello
echo_var "${!message}"      # Hello, again!

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

```
#!/bin/bash
# dereference.sh
# 对一个传递给函数的参数进行解除引用的操作.
# 此脚本由Bruce W. Clare所编写.

dereference ()
{
     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
 5  ITERATIONS=3  # 取得输入的次数.
 6  icount=1
 7
 8  my_read () {
 9    #  用my_read varname这种形式来调用,
10    #+ 将之前用括号括起的值作为默认值输出,
11    #+ 然后要求输入一个新值.
12
13    local local_var
14
15    echo -n "Enter a value "
16    eval 'echo -n "[$'$1'] "'   #  之前的值.
17  # eval echo -n "[\$$1] "      #  更容易理解,
18                                #+ 但会丢失用户在尾部输入的空格.
19    read local_var
20    [ -n "$local_var" ] && eval $1=\$local_var
21
22    # "与列表": 如果"local_var"的测试结果为true, 则把变量"$1"的值赋给它.
23  }
24
25  echo
26
27  while [ "$icount" -le "$ITERATIONS" ]
28  do
29    my_read var
30    echo "Entry #$icount = $var"
31    let "icount += 1"
32    echo
33  done
34
35
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
```

```
# max.sh: 取两个整数中的最大值.

E_PARAM_ERR=-198    # 如果传给函数的参数少于2个时, 就返回这个值.
EQUAL=-199          # 如果两个整数相等时, 返回这个值.
#  任意超出范围的
#+ 参数值都可能传递到函数中.

max2 ()             # 返回两个整数中的最大值.
{                   # 注意: 参与比较的数必须小于257.
if [ -z "$2" ]
then
  return $E_PARAM_ERR
fi

if [ "$1" -eq "$2" ]
then
  return $EQUAL
else
  if [ "$1" -gt "$2" ]
  then
    return $1
  else
    return $2
  fi
fi
}

max2 33 34
return_val=$?

if [ "$return_val" -eq $E_PARAM_ERR ]
then
  echo "Need to pass two parameters to the function."
elif [ "$return_val" -eq $EQUAL ]
  then
    echo "The two numbers are equal."
else
    echo "The larger of the two numbers is $return_val."
fi


exit 0

#  练习(简单):
#  -----------
#  把这个脚本转化为交互式脚本,
#+ 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.

```
count_lines_in_etc_passwd()
{
  [[ -r /etc/passwd ]] && REPLY=$(echo $(wc -l < /etc/passwd))
  #  如果/etc/passwd是可读的, 那么就把REPLY设置为文件的行数.
  #  这样就可以同时返回参数值与状态信息.
  #  'echo'看上去没什么用, 可是 . . .
  #+ 它的作用是删除输出中的多余空白字符.
}

if count_lines_in_etc_passwd
then
  echo "There are $REPLY lines in /etc/passwd."
else
  echo "Cannot count lines in /etc/passwd."
fi

# 感谢, S.C.
```

例子 23-8. 将阿拉伯数字转化为罗马数字

```
#!/bin/bash

# 将阿拉伯数字转化为罗马数字
# 范围: 0 - 200
# 比较粗糙, 但可以正常工作.

# 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 $num 90 LXXXX
52 num=$?
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 {
```

```
  return $1
}

return_test 27          # o.k.
echo $?                 # 返回27.

return_test 255         # 依然是o.k.
echo $?                 # 返回255.

return_test 257         # 错误!
echo $?                 # 返回1 (对应各种错误的返回码).

# ======================================================
return_test -151896     # 能返回一个大负数么?
echo $?                 # 能否返回-151896?
                        # 显然不行! 只返回了168.
#  Bash 2.05b以前的版本
#+ 允许返回大负数.
#  Bash的新版本(2.05b之后)修正了这个漏洞.
#  这可能会影响以前所编写的脚本.
#  一定要小心!
# ======================================================

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.

```
Return_Val=   # 用于保存函数特大返回值的全局变量.

alt_return_test ()
{
  fvar=$1
  Return_Val=$fvar
  return   # 返回0 (成功).
}

alt_return_test 1
echo $?                                 # 0
echo "return value = $Return_Val"       # 1

alt_return_test 256
echo "return value = $Return_Val"       # 256

alt_return_test 257
echo "return value = $Return_Val"       # 257

alt_return_test 25701
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

```
#!/bin/bash
# max2.sh: 取两个大整数中的最大值.

#  这是前一个例子"max.sh"的修改版,
#+ 这个版本可以比较两个大整数.

EQUAL=0             # 如果两个值相等, 那就返回这个值.
E_PARAM_ERR=-99999  # 没有足够多的参数传递给函数.
#           ^^^^^^    任意超出范围的参数都可以传递进来.

max2 ()             # "返回"两个整数中最大的那个.
{
if [ -z "$2" ]
then
  echo $E_PARAM_ERR
  return
fi

if [ "$1" -eq "$2" ]
then
  echo $EQUAL
  return
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.

23.2. 局部变量

怎样使一个变量变成"局部"变量?

局部变量

如果变量用*local*来声明，那么它就只能够在该变量被声明的代码块中可见. 这个代码块就是局部"范围". 在一个函数中，一个*局部变量*只有在函数代码块中才有意义.

例子 23-12. 局部变量的可见范围

```
#!/bin/bash
# 函数内部的局部变量与全局变量.

func ()
{
  local loc_var=23       # 声明为局部变量.
  echo                   # 使用'local'内建命令.
  echo "\"loc_var\" in function = $loc_var"
  global_var=999         # 没有声明为局部变量.
                         # 默认为全局变量.
  echo "\"global_var\" in function = $global_var"
}

func

# 现在, 来看看局部变量"loc_var"在函数外部是否可见.

echo
echo "\"loc_var\" outside function = $loc_var"
                                     # $loc_var outside function =
                                     # 不行, $loc_var不是全局可见的.
echo "\"global_var\" outside function = $global_var"
                                     # 在函数外部$global_var = 999
                                     # $global_var是全局可见的.
echo

exit 0
#  与C语言相比, 在函数内声明的Bash变量
#+ 除非它被明确声明为local时, 它才是局部的.
```

在函数被调用之前，*所有*在函数中声明的变量，在函数体外都是不可见的，当然也包括那些被明确声明为*local*的变量.

```
#!/bin/bash

func ()
{
global_var=37    #  在函数被调用之前,
                 #+ 变量只在函数体内可见.
}                #  函数结束

echo "global_var = $global_var"  # global_var =
                                 #  函数"func"还没被调用,
                                 #+ 所以$global_var还不能被访问.

func
echo "global_var = $global_var"  # global_var = 37
                                 # 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

```
#!/bin/bash

#                阶乘
#                ----


# bash允许递归吗?
# 嗯, 允许, 但是...
# 他太慢了, 所以恐怕你难以忍受.


MAX_ARG=5
E_WRONG_ARGS=65
E_RANGE_ERR=66


if [ -z "$1" ]
then
  echo "Usage: `basename $0` number"
  exit $E_WRONG_ARGS
fi

if [ "$1" -gt $MAX_ARG ]
then
  echo "Out of range (5 is maximum)."
  #  现在让我们来了解一些实际情况.
  #  如果你想计算比这个更大的范围的阶乘,
  #+ 应该用真正的编程语言来重写它.
  exit $E_RANGE_ERR
fi

fact ()
{
  local number=$1
  #  变量"number"必须声明为局部变量,
  #+ 否则不能正常工作.
  if [ "$number" -eq 0 ]
  then
    factorial=1    # 0的阶乘为1.
  else
    let "decrnum = number - 1"
    fact $decrnum  # 递归的函数调用(就是函数调用自己).
    let "factorial = $number * $?"
  fi

  return $factorial
}

fact $1
echo "Factorial of $1 is $?."

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.

```
#!/bin/bash

#  警告: 运行这个脚本可能使你的系统失去响应!
#  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.

recursive_function ()
{
echo "$1"      # 让这个函数做点事, 以便于加速段错误.
(( $1 < $2 )) && recursive_function $(( $1 + 1 )) $2;
#  当第一个参数比第二个参数少时,
#+ 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.
```

23.3. 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

```
#! /bin/bash
#
# 汉诺塔
# Bash script
# Copyright (C) 2000 Amit Singh. All Rights Reserved.
# http://hanoi.kernelthread.com
#
# 在bash version 2.05b.0(13)-release下通过测试
#
#  经过脚本原作者同意
#+ 可以使用在"Advanced Bash Scripting Guide"中.
#  本书作者对此脚本做了少许修改.

#==========================================================================#
#  汉诺塔是由Edouard Lucas提出的数学谜题,
#+ 他是19世纪的法国数学家.
#
#  有三个直立的柱子竖在地面上.
#  第一个柱子上有一组盘子套在上面.
#  这些盘子是平的, 中间有孔,
#+ 可以套在柱子上面.
#  这些盘子的直径不同, 它们从下到上,
#+ 按照尺寸递减的顺序摆放.
#  也就是说, 最小的在最上边, 最大的在最下面.
#
#  现在的任务是要把这组盘子
#+ 从一个柱子上全部搬到另一个柱子上.
#  你每次只能将一个盘子从一个柱子移动到另一个柱子上.
#  你也可以把盘子从其他的柱子上移回到原来的柱子上.
#  你只能把小的盘子放到大的盘子上,
#+ 反过来就*不*行.
#  切记, 这是规则, 绝对不能把大盘子放到小盘子的上面.
#
#  如果盘子的数量比较少, 那么移不了几次就能完成.
#+ 但是随着盘子数量的增加,
#+ 移动次数几乎成倍的增长,
#+ 而且移动的"策略"也会变得越来越复杂.
#
#  想了解更多信息的话, 请访问http://hanoi.kernelthread.com.
#
#
#              ...                     ...                     ...
#             | |                     | |                     | |
#            _|_|_                    | |                     | |
#           |_____|                   | |                     | |
#          |_______|                  | |                     | |
#         |_________|                 | |                     | |
#        |___________|                | |                     | |
#       |_____________|               | |                     | |
#        |             |              | |                     | |
# .--------------------------------------------------------------------.
# |*********************************************************************|
#           #1                      #2                      #3
#
#==========================================================================#


E_NOPARAM=66  # 没有参数传给脚本.
E_BADPARAM=67 # 传给脚本的盘子个数不符合要求.
Moves=        # 保存移动次数的全局变量.
              # 这里修改了原来的脚本.

dohanoi() {   # 递归函数.
    case $1 in
    0)
        ;;
    *)
        dohanoi "$(($1-1))" $2 $4 $3
        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  *)
 90      echo "usage: $0 N"
 91      echo "       Where \"N\" is the number of disks."
 92      exit $E_NOPARAM;
 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)
```

前一页 局部变量
首页 上一级
下一页 别名

24. 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='echo "\"Alias Jesse James\" was a 1959 comedy starring Bob
Hope."'
 10 Jesse_James
 11
 12 echo; echo; echo;
 13
 14 alias ll="ls -l"
 15 # 可以使用单引号(')或双引号(")来定义一个别名.
 16
 17 echo "Trying aliased \"ll\":"
 18 ll /usr/X11R6/bin/mk*   #* 别名工作了.
 19
 20 echo
 21
 22 directory=/usr/X11R6/bin/
 23 prefix=mk*  # 看一下通配符会不会引起麻烦.
 24 echo "Variables \"directory\" + \"prefix\" = $directory$prefix"
 25 echo
 26
 27 alias lll="ls -l $directory$prefix"
 28
 29 echo "Trying aliased \"lll\":"
 30 lll         # 详细列出/usr/X11R6/bin目录下所有以mk开头的文件.
 31 # 别名能处理连接变量 -- 包括通配符 -- o.k.
 32
 33
 34
 35
 36 TRUE=1
 37
 38 echo
 39
 40 if [ TRUE ]
 41 then
 42   alias rr="ls -l"
 43   echo "Trying aliased \"rr\" within if/then statement:"
 44   rr /usr/X11R6/bin/mk*   #* 产生错误信息!
 45   # 别名不能在混合结构中使用.
 46   echo "However, previously expanded alias still recognized:"
 47   ll /usr/X11R6/bin/mk*
 48 fi
 49
 50 echo
 51
 52 count=0
 53 while [ $count -lt 3 ]
```

```
54 do
55   alias rrr="ls -l"
56   echo "Trying aliased \"rrr\" within \"while\" loop:"
57   rrr /usr/X11R6/bin/mk*   #* 这里, 别名也不会扩展.
58                            #  alias.sh: line 57: rrr: command not found
59   let count+=1
60 done
61
62 echo; echo
63
64 alias xyz='cat $0'    # 脚本打印自身内容.
65                       # 注意是单引号(强引用).
66 xyz
67 #  虽然Bash文档建议, 它不能正常运行,
68 #+ 不过它看起来是可以运行的.
69 #
70 #  然而, 就像Steve Jacobson所指出的那样,
71 #+ 参数"$0"立即扩展成了这个别名的声明.
72
73 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
```

```
bash$ ./unalias.sh
total 6
drwxrwxr-x    2 bozo     bozo         3072 Feb  6 14:04 .
drwxr-xr-x   40 bozo     bozo         2048 Feb  6 14:04 ..
-rwxr-xr-x    1 bozo     bozo          199 Feb  6 14:04 unalias.sh

./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.

---

25. 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
  4 if [ ! -z "$1" ] && echo "Argument #1 = $1" && [ ! -z "$2" ] &&
echo "Argument #2 = $2"
  5 then
  6   echo "At least 2 arguments passed to script."
  7   # 所有连接起来的命令都返回true.
  8 else
  9   echo "Less than 2 arguments passed to script."
 10   # 整个命令列表中至少有一个命令返回false.
 11 fi
 12 # 注意, "if [ ! -z $1 ]"也可以, 但它是有所假定的等价物.
 13 #   if [ -n $1 ] 这个不行.
 14 #     然而, 如果加了引用就行了.
 15 #  if [ -n "$1" ] 这样就行了.
 16 #     小心!
 17 # 最好将你要测试的变量引用起来, 这么做是非常好的习惯.
 18
 19
 20 # 下面这段代码与上面代码是等价的, 不过下面这段代码使用的是"纯粹"的if/then结构.
 21 if [ ! -z "$1" ]
 22 then
 23   echo "Argument #1 = $1"
 24 fi
 25 if [ ! -z "$2" ]
 26 then
 27   echo "Argument #2 = $2"
 28   echo "At least 2 arguments passed to script."
 29 else
 30   echo "Less than 2 arguments passed to script."
 31 fi
 32 # 这么写的话, 行数太多了, 没有"与列表"来的精简.
 33
 34
 35 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
  3 ARGS=1         # 期望的参数个数.
  4 E_BADARGS=65   # 如果传递的参数个数不正确, 那么给出这个退出码.
  5
  6 test $# -ne $ARGS && echo "Usage: `basename $0` $ARGS argument(s)"
&& exit $E_BADARGS
  7 #  如果"条件1"测试为true (表示传递给脚本的参数个数不对),
  8 #+ 则余下的命令会被执行, 并且脚本将结束运行.
  9
 10 #                    false     ,                .
```

```
      只有当上面的测试条件为      的时候  这行代码才会被执行
11 echo "Correct number of arguments passed to this script."
12
13 exit 0
14
15 # 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.

```
 1 arg1=$@       # 不管怎样, 将$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

```
1 command-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
 6 E_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, 这个脚本由Miquel van Smoorenburg所编
写.
 2 #+==> 用于展示"与"/"或"列表的使用.
 3 # ==> "箭头"注释是由本书作者添加的.
 4
 5 [ -x /usr/bin/clear ] && /usr/bin/clear
 6   # ==> 如果/usr/bin/clear存在, 那么就调用它.
 7   # ==> 在调用一个命令前, 检查一下它是否存在.
 8   #+==> 这样可以避免错误信息, 和其他愚蠢的结果.
 9
10   # ==> . . .
11
12 # 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...
13 for i in /etc/rc1.d/S[0-9][0-9]* ; do
14         # 检查一下脚本是否在那里.
15         [ -x "$i" ] || continue
16   # ==> 如果在$PWD中没发现相应的文件,
17   #+==> 则会使用"continue"跳过本次循环.
18
```

```
        # 不接受备份文件, 也不接受由rpm产生的文件.
        case "$1" in
                *.rpmsave|*.rpmorig|*.rpmnew|*~|*.orig)
                        continue;;
        esac
        [ "$i" = "/etc/rc1.d/S00single" ] && continue
  # ==> 设置脚本名, 但现在还不执行它.
        $i start
done

  # ==> . . .
```

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.

```
false && true || echo false         # false

# 与下面的结果相同
( false && true ) || echo false     # false
# But *not*
false && ( true || echo false )     # (没有输出)

#  注意, 以从做到右的顺序进行分组与求值,
#+ 这是因为逻辑操作符"&&"和"||"具有相同的优先级.

#  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.

#  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.

---

26. 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), 可以使用*大括号*, 访问形式为`${variable[xx]}`.

例子 26-1. 简单的数组使用

```
#!/bin/bash


area[11]=23
area[13]=37
area[51]=UFOs

#  数组成员不一定非得是相邻或连续的.

#  数组的部分成员可以不被初始化.
#  数组中允许空缺元素.
#  实际上, 保存着稀疏数据的数组("稀疏数组")
#+ 在电子表格处理软件中是非常有用的.


echo -n "area[11] = "
echo ${area[11]}    #  需要{大括号}.

echo -n "area[13] = "
echo ${area[13]}

echo "Contents of area[51] are ${area[51]}."

# 没被初始化的数组成员打印为空值(null变量).
echo -n "area[43] = "
echo ${area[43]}
echo "(area[43] unassigned)"

echo

# 两个数组元素的和被赋值给另一个数组元素
area[5]=`expr ${area[11]} + ${area[13]}`
echo "area[5] = area[11] + area[13]"
echo -n "area[5] = "
echo ${area[5]}

area[6]=`expr ${area[11]} + ${area[51]}`
echo "area[6] = area[11] + area[51]"
echo -n "area[6] = "
echo ${area[6]}
# 这里会失败, 是因为不允许整数与字符串相加.

echo; echo; echo

# --------------------------------------------------------------------
# 另一个数组, "area2".
# 另一种给数组变量赋值的方法...
# array_name=( XXX YYY ZZZ ... )

area2=( zero one two three four )

echo -n "area2[0] = "
echo ${area2[0]}
# 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]).

echo -n "area2[1] = "
echo ${area2[1]}    # [1]是数组的第2个元素.
# --------------------------------------------------------------------

echo; echo; echo

# ----------------------------------------------
# 第3个数组, "area3".
#   3                   ...
```

```
    第 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).
 5 Line[1]="I do not know which to prefer,"
 6 Line[2]="The beauty of inflections"
 7 Line[3]="Or the beauty of innuendoes,"
 8 Line[4]="The blackbird whistling"
 9 Line[5]="Or just after."
10
11 # 出处.
12 Attrib[1]=" Wallace Stevens"
13 Attrib[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
16 echo
17
18 for index in 1 2 3 4 5    # 5行.
19 do
20   printf "     %s\n" "${Line[index]}"
21 done
22
23 for index in 1 2          # 出处为2行.
24 do
25   printf "          %s\n" "${Attrib[index]}"
26 done
27
28 echo
29
30 exit 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, 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
 5 array=( zero one two three four five )
```

```
 6 # 数组元素 0   1   2    3     4    5
 7
 8 echo ${array[0]}        #  0
 9 echo ${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).
12 echo ${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
16 echo "--------------"
17
18 echo ${#array[0]}       #  4
19                         #  第一个数组元素的长度.
20 echo ${#array}          #  4
21                         #  第一个数组元素的长度.
22                         #  (另一种表示形式)
23
24 echo ${#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
28 echo ${#array[*]}       #  6
29                         #  数组中的元素个数.
30 echo ${#array[@]}       #  6
31                         #  数组中的元素个数.
32
33 echo "--------------"
34
35 array2=( [0]="first element" [1]="second element" [3]="fourth element" )
36
37 echo ${array2[0]}       # 第一个元素
38 echo ${array2[1]}       # 第二个元素
39 echo ${array2[2]}       #
40                         # 因为并没有被初始化, 所以此值为null.
41 echo ${array2[3]}       # 第四个元素
42
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
11 arrayZ=( one two three four five five )
12
13 echo
14
15 # 提取尾部的子串
16 echo ${arrayZ[@]:0}      # one two three four five five
17                          # 所有元素.
18
19 echo ${arrayZ[@]:1}      # two three four five five
20                          # element[0]后边的所有元素.
21
22 echo ${arrayZ[@]:1:2}    # two three
23                          # 只提取element[0]后边的两个元素.
24
25 echo "-----------------------"
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
31 echo ${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
36 echo ${arrayZ[@]##t*e}   # one two four five five
```

```
                            # 匹配将应用于数组的所有元素.
                            # 匹配到了"three", 并且将它删除.

# 从字符串的结尾删除最短的匹配
echo ${arrayZ[@]%h*e}   # one two t four five five
                            # 匹配将应用于数组的所有元素.
                            # 匹配到了"hree", 并且将它删除.

# 从字符串的结尾删除最长的匹配
echo ${arrayZ[@]%%t*e}  # one two four five five
                            # 匹配将应用于数组的所有元素.
                            # 匹配到了"three", 并且将它删除.

echo "-----------------------"

# 子串替换

# 第一个匹配到的子串将会被替换
echo ${arrayZ[@]/fiv/XYZ}   # one two three four XYZe XYZe
                             # 匹配将应用于数组的所有元素.

# 所有匹配到的子串都会被替换
echo ${arrayZ[@]//iv/YY}    # one two three four fYYe fYYe
                             # 匹配将应用于数组的所有元素.

# 删除所有的匹配子串
# 如果没有指定替换字符串的话, 那就意味着'删除'
echo ${arrayZ[@]//fi/}      # one two three four ve ve
                             # 匹配将应用于数组的所有元素.

# 替换字符串前端子串
echo ${arrayZ[@]/#fi/XY}    # one two three four XYve XYve
                             # 匹配将应用于数组的所有元素.

# 替换字符串后端子串
echo ${arrayZ[@]/%ve/ZZ}    # one two three four fiZZ fiZZ
                             # 匹配将应用于数组的所有元素.

echo ${arrayZ[@]/%o/XX}     # one twXX three four five five
                             # 为什么?

echo "-----------------------"


# 在将处理后的结果发送到awk(或者其他的处理工具)之前 --
# 回忆一下:
#   $( ... )是命令替换.
#   函数作为子进程运行.
#   函数结果输出到stdout.
#   用read来读取函数的stdout.
#   使用name[@]表示法指定了一个"for-each"操作.

newstr() {
    echo -n "!!!"
}

echo ${arrayZ[@]/%e/$(newstr)}
# on!!! two thre!!! four fiv!!! fiv!!!
# Q.E.D: 替换动作实际上是一个'赋值'.

#  使用"For-Each"形式的
echo ${arrayZ[@]//*/$(newstr optional_arguments)}
#  现在, 如果Bash只将匹配到的子串作为$0
#+ 传递给将被调用的函数 . . .

echo

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

```
#!/bin/bash
# script-array.sh: 将这个脚本的内容赋值给数组.
# 这个脚本的灵感来自于Chris Martin的e-mail(感谢!).

script_contents=( $(cat "$0") )  #  将这个脚本的内容($0)
                                  #+ 赋值给数组.

```

```
 8 for element in $(seq 0 $((${#script_contents[@]} - 1)))
 9   do                  #  ${#script_contents[@]}
10                       #+ 表示数组元素的个数.
11                       #
12                       #  一个小问题:
13                       #  为什么必须使用seq 0?
14                       #  用seq 1来试一下.
15   echo -n "${script_contents[$element]}"
16                       # 在同一行上显示脚本中每个域的内容.
17   echo -n " -- "      # 使用 " -- " 作为域分割符.
18 done
19
20 echo
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
 3 declare -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
 7 echo "Enter your favorite colors (separated from each other by a space)."
 8
 9 read -a colors     # 至少需要键入3种颜色, 以便于后边的演示.
10 #  'read'命令的特殊选项,
11 #+ 允许给数组元素赋值.
12
13 echo
14
15 element_count=${#colors[@]}
16 # 提取数组元素个数的特殊语法.
17 #     用element_count=${#colors[*]}也一样.
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
25 index=0
26
27 while [ "$index" -lt "$element_count" ]
28 do    # 列出数组中的所有元素.
29   echo ${colors[$index]}
30   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
10 array0=( first second third )
11 array1=( '' )    # "array1"包含一个空元素.
12 array2=( )       # 没有元素 . . . "array2"为空.
13
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$@ and $*. 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
```

```
# ------------------------------------------


# 添加一个元素到数组.
array=( "${array[@]}" "new element" )
# 或
array[${#array[*]}]="new element"

# 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.

```
#!/bin/bash

filename=sample_file

#            cat sample_file
#
#            1 a b c
#            2 d e fg


declare -a array1

array1=( `cat "$filename"`)                 #  将$filename的内容
#         List file to stdout                #+ 加载到数组array1.
#
#  array1=( `cat "$filename" | tr '\n' ' '`)
#                                 把文件中的换行替换为空格.
#  其实这么做是没必要的, Not necessary because Bash does word splitting,
#+ 因为Bash在做单词分割(word splitting)的时候, 将会把换行转换为空格.

echo ${array1[@]}             # 打印数组.
#                               1 a b c 2 d e fg
#
#  文件中每个被空白符分隔的"单词"
#+ 都被保存到数组的一个元素中.

element_count=${#array1[*]}
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

```
#! /bin/bash
# array-assign.bash

#  数组操作是Bash所特有的,
#+ 所以才使用".bash"作为脚本扩展名.

# Copyright (c) Michael S. Zick, 2003, All rights reserved.
# License: Unrestricted reuse in any form, for any purpose.
# Version: $ID$
#
# 说明与注释由William Park所添加.

#  基于Stephane Chazelas所提供的
#+ 出现在本书中的一个例子.

# 'times'命令的输出格式:
# User CPU <space> System CPU
# User CPU of dead children <space> System CPU of dead children

#  Bash有两种方法,
#+ 可以将一个数组的所有元素都赋值给一个新的数组变量.
#  在2.04, 2.05a和2.05b版本的Bash中,
#+ 这两种方法都会丢弃数组中的"空引用"(null值)元素.
#  另一种给数组赋值的方法将会被添加到新版本的Bash中,
#+ 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.

#  可以使用内部命令来构造一个大数组,
#+ 当然, 构造一个包含上千元素数组的其他方法
#+ 也能很好的完成任务.

```

```
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", 或C++中的"数组引用")
 8 #+ 或者"建立自己的赋值语句".
 9
10
11 CpArray_Mac() {
12
13 # 建立赋值命令
14
15     echo -n 'eval '
16     echo -n "$2"                    # 目的参数名
17     echo -n '=( ${'
18     echo -n "$1"                    # 原参数名
19     echo -n '[@]} )'
20
21 # 上边这些语句会构成一条命令.
22 # 这仅仅是形式上的问题.
```

```
}

declare -f CopyArray                  # 函数"指针"
CopyArray=CpArray_Mac                 # 构造语句

Hype()
{

# 需要连接的数组名为$1.
# (把这个数组与字符串"Really Rocks"结合起来, 形成一个新数组.)
# 并将结果从数组$2中返回.

    local -a TMP
    local -a hype=( Really Rocks )

    $($CopyArray $1 TMP)
    TMP=( ${TMP[@]} ${hype[@]} )
    $($CopyArray TMP $2)
}

declare -a before=( Advanced Bash Scripting )
declare -a after

echo "Array Before = ${before[@]}"

Hype before after

echo "Array After = ${after[@]}"

# 连接的太多了?

echo "What ${after[@]:3:2}?"

declare -a modest=( ${after[@]:2:1} ${after[@]:3:2} )
#                     ----       子串提取       ----

echo "Array Modest = ${modest[@]}"

# 'before'发生了什么变化么?

echo "Array Before = ${before[@]}"

exit 0
```

例子 26-10. 关于串联数组的更多信息

```
#! /bin/bash
# array-append.bash

# Copyright (c) Michael S. Zick, 2003, All rights reserved.
# License: Unrestricted reuse in any form, for any purpose.
# Version: $ID$
#
# 在格式上, 由M.C做了一些修改.


# 数组操作是Bash特有的属性.
# 传统的UNIX /bin/sh缺乏类似的功能.


#  将这个脚本的输出通过管道传递给'more',
#+ 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.


# 依次使用下标.
declare -a array1=( zero1 one1 two1 )
# 数组中存在空缺的元素([1]未定义).
declare -a array2=( [0]=zero2 [2]=two2 [3]=three2 )

echo
echo '- Confirm that the array is really subscript sparse. -'
echo "Number of elements: 4"        # 仅仅为了演示, 所以就写死了.
for (( i = 0 ; i < 4 ; i++ ))
do
    echo "Element [$i]: ${array2[$i]}"
done
# 也可以参考一个更通用的例子, basics-reviewed.bash.


declare -a dest

#            (   )   3     .
```

```
      将两个数组合并 附加 到第 个数组
echo
echo 'Conditions: Unquoted, default IFS, All-Elements-Of operator'
echo '- Undefined elements not present, subscripts not maintained. -'
# # 那些未定义的元素不会出现; 组合时会丢弃这些元素.

dest=( ${array1[@]} ${array2[@]} )
# dest=${array1[@]}${array2[@]}      # 令人奇怪的结果, 或许是个bug.

# 现在, 打印结果.
echo
echo '- - Testing Array Append - -'
cnt=${#dest[@]}

echo "Number of elements: $cnt"
for (( i = 0 ; i < cnt ; i++ ))
do
    echo "Element [$i]: ${dest[$i]}"
done

# 将数组赋值给一个数组中的元素(两次).
dest[0]=${array1[@]}
dest[1]=${array2[@]}

# 打印结果.
echo
echo '- - Testing modified array - -'
cnt=${#dest[@]}

echo "Number of elements: $cnt"
for (( i = 0 ; i < cnt ; i++ ))
do
    echo "Element [$i]: ${dest[$i]}"
done

# 检查第二个元素的修改状况.
echo
echo '- - Reassign and list second element - -'

declare -a subArray=${dest[1]}
cnt=${#subArray[@]}

echo "Number of elements: $cnt"
for (( i = 0 ; i < cnt ; i++ ))
do
    echo "Element [$i]: ${subArray[$i]}"
done

#  如果你使用'=${ ... }'形式
#+ 将一个数组赋值到另一个数组的一个元素中,
#+ 那么这个数组的所有元素都会被转换为一个字符串,
#+ 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).

# 如果原来数组中的所有元素都不包含空白符 . . .
# 如果原来的数组下标都是连续的 . . .
# 那么我们就可以将原来的数组进行恢复.

# 从修改过的第二个元素中, 将原来的数组恢复出来.
echo
echo '- - Listing restored element - -'

declare -a subArray=( ${dest[1]} )
cnt=${#subArray[@]}

echo "Number of elements: $cnt"
for (( i = 0 ; i < cnt ; i++ ))
do
    echo "Element [$i]: ${subArray[$i]}"
done
echo '- - Do not depend on this behavior. - -'
echo '- - This behavior is subject to change - -'
echo '- - in versions of Bash newer than version 2.05b - -'

# 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).

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*

```
#!/bin/bash
# bubble.sh: 一种排序方式, 冒泡排序.

# 回忆一下冒泡排序的算法. 我们在这里要实现它...

#  依靠连续的比较数组元素进行排序,
#+ 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.
#  当第一轮比较结束之后, 最"重"的元素就会被移动到最底部.
#  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.
#  依此类推.
#  这意味着每轮比较不需要比较之前已经"沉淀"好的数据.
#  因此你会注意到后边的数据在打印的时候会快一些.


exchange()
{
  # 交换数组中的两个元素.
  local temp=${Countries[$1]} #  临时保存
                              #+ 要交换的那个元素.
  Countries[$1]=${Countries[$2]}
  Countries[$2]=$temp

  return
}

declare -a Countries  #  声明数组,
                      #+ 此处是可选的, 因为数组在下面被初始化.

#  我们是否可以使用转义符(\)
#+ 来将数组元素的值放在不同的行上?
#  可以.

Countries=(Netherlands Ukraine Zaire Turkey Russia Yemen Syria \
Brazil Argentina Nicaragua Japan Mexico Venezuela Greece England \
Israel Peru Canada Oman Denmark Wales France Kenya \
Xanadu Qatar Liechtenstein Hungary)

# "Xanadu"虚拟出来的世外桃源.
#+


clear                       # 开始之前的清屏动作.

echo "0: ${Countries[*]}"  # 从索引0开始列出整个数组.

number_of_elements=${#Countries[@]}
let "comparisons = $number_of_elements - 1"

count=1 # 传递数字.

while [ "$comparisons" -gt 0 ]          # 开始外部循环
do

  index=0  # 在每轮循环开始之前, 重置索引.

  while [ "$index" -lt "$comparisons" ] # 开始内部循环
  do
    if [ ${Countries[$index]} \> ${Countries[`expr $index + 1`]} ]
    #  如果原来的排序次序不对...
    #  回想一下, 在单括号中,
    #+ \>是ASCII码的比较操作符.

    #  if [[ ${Countries[$index]} > ${Countries[`expr $index + 1`]} ]]
    #+ 这样也行.
    then
      exchange $index `expr $index + 1`  # 交换.
    fi
    let "index += 1"  # 或者,   index+=1   在Bash 3.1之后的版本才能这么用.
  done # 内部循环结束

# ----------------------------------------------------------------------
# Paulo Marcel Coelho Aragao建议我们可以使用更简单的for循环.
#
# for (( last = $number_of_elements - 1 ; last > 1 ; last-- ))
# do
#     for (( i = 0 ; i < last ; i++ ))
#     do
#         [[ "${Countries[$i]}" > "${Countries[$((i+1))]}" ]] \
#             && exchange $i $((i+1))
#     done
# done
```

```
# ------------------------------------------------------------------------

let "comparisons -= 1" #  因为最"重"的元素到了底部,
                       #+ 所以每轮我们可以少做一次比较.

echo
echo "$count: ${Countries[@]}"  # 每轮结束后, 都打印一次数组.
echo
let "count += 1"                # 增加传递计数.

done                            # 外部循环结束
                                # 至此, 全部完成.

exit 0
```

--

我们可以在数组中嵌套数组么?

```
#!/bin/bash
# "嵌套"数组.

#  Michael Zick提供了这个用例,
#+ William Park做了一些修正和说明.

AnArray=( $(ls --inode --ignore-backups --almost-all \
    --directory --full-time --color=none --time=status \
    --sort=time -l ${PWD} ) )  # 命令及选项.

# 空格是有意义的 . . . 并且不要在上边用引号引用任何东西.

SubArray=( ${AnArray[@]:11:1}  ${AnArray[@]:6:5} )
#  这个数组有六个元素:
#+     SubArray=( [0]=${AnArray[11]} [1]=${AnArray[6]} [2]=${AnArray[7]}
#      [3]=${AnArray[8]} [4]=${AnArray[9]} [5]=${AnArray[10]} )
#
#  Bash数组是字符串(char *)类型
#+ 的(循环)链表.
#  因此, 这不是真正意义上的嵌套数组,
#+ 只不过功能很相似而已.

echo "Current directory and date of last status change:"
echo "${SubArray[@]}"

exit 0
```

--

如果将"嵌套数组"与间接引用组合起来使用的话, 将会产生一些非常有趣的用法.

例子 26-12. 嵌套数组与间接引用

```
#!/bin/bash
# embedded-arrays.sh
# 嵌套数组和间接引用.

# 本脚本由Dennis Leeuw编写.
# 经过授权, 在本书中使用.
# 本书作者做了少许修改.


ARRAY1=(
        VAR1_1=value11
        VAR1_2=value12
        VAR1_3=value13
)

ARRAY2=(
        VARIABLE="test"
        STRING="VAR1=value1 VAR2=value2 VAR3=value3"
        ARRAY21=${ARRAY1[*]}
)       # 将ARRAY1嵌套到这个数组中.

function print () {
        OLD_IFS="$IFS"
        IFS=$'\n'       #  这么做是为了每行
                        #+ 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
10  LOWER_LIMIT=1        # 从1开始.
11  UPPER_LIMIT=1000     # 到1000.
12  # (如果你时间很多的话 . . . 你可以将这个数值调的很高.)
13
14  PRIME=1
15  NON_PRIME=0
16
17  let SPLIT=UPPER_LIMIT/2
18  # 优化:
19  # 只需要测试中间到最大的值(为什么?).
20  # (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
22  declare -a Primes
23  # Primes[]是个数组.
24
25
26  initialize ()
27  {
28  # 初始化数组.
29
```

```
i=$LOWER_LIMIT
until [ "$i" -gt "$UPPER_LIMIT" ]
do
  Primes[i]=$PRIME
  let "i += 1"
done
#  假定所有数组成员都是需要检查的(素数)
#+ 直到检查完成.
}

print_primes ()
{
# 打印出所有数组Primes[]中被标记为素数的元素.

i=$LOWER_LIMIT

until [ "$i" -gt "$UPPER_LIMIT" ]
do

  if [ "${Primes[i]}" -eq "$PRIME" ]
  then
    printf "%8d" $i
    # 每个数字打印前先打印8个空格, 在偶数列才打印.
  fi

  let "i += 1"

done

}

sift () # 查出非素数.
{

let i=$LOWER_LIMIT+1
# 我们都知道1是素数, 所以我们从2开始.
# (译者注: 从2开始并不是由于1是素数, 而是因为要去掉以后每个数的倍数, 感谢网友KevinChen.)
until [ "$i" -gt "$UPPER_LIMIT" ]
do

if [ "${Primes[i]}" -eq "$PRIME" ]
# 不要处理已经过滤过的数字(被标识为非素数).
then

  t=$i

  while [ "$t" -le "$UPPER_LIMIT" ]
  do
    let "t += $i "
    Primes[t]=$NON_PRIME
    # 标识为非素数.
  done

fi

  let "i += 1"
done


}


# ==============================================
# main ()
# 继续调用函数.
initialize
sift
print_primes
# 这里就是被称为结构化编程的东西.
# ==============================================

echo

exit 0



# ------------------------------------------------------------ #
# 因为前面的'exit'语句, 所以后边的代码不会运行.

#  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,
#+ 这个版本可以运行的快一些.

# 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 UPPER_LIMIT=$1                    # 来自于命令行.
116 let SPLIT=UPPER_LIMIT/2           # 从中间值到最大值.
117
118 Primes=( '' $(seq $UPPER_LIMIT) )
119
120 i=1
121 until (( ( i += 1 ) > SPLIT ))  # 仅需要从中间值检查.
122 do
123   if [[ -n $Primes[i] ]]
124   then
125     t=$i
126     until (( ( t += i ) > UPPER_LIMIT ))
127     do
128       Primes[t]=
129     done
130   fi
131 done
132 echo ${Primes[*]}
133
134 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
  7 BP=100             #  栈数组的基址指针.
  8                     #  从元素100开始.
  9
 10 SP=$BP             #  栈指针.
 11                     #  将其初始化为栈"基址"(栈底).
 12
 13 Data=               #  当前栈的数据内容.
 14                     #  必须定义为全局变量,
 15                     #+ 因为函数所能够返回的整数存在范围限制.
 16
 17 declare -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 "$BP" ]    # 栈空?
 38 then
 39   return
 40 fi                         #  这使得SP不会超过100,
 41                             #+ 例如, 这可以防止堆栈失控.
 42
 43 Data=${stack[$SP]}
 44 let "SP += 1"              # 更新栈指针.
 45 return
 46 }
 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, Bach: An Eternal Golden Braid_",
13 #+ 第137页.
14
15
16 LIMIT=100      # 需要计算的数列长度.
17 LINEWIDTH=20   # 每行打印的个数.
18
19 Q[1]=1         # 数列的头两项都为1.
20 Q[2]=1
21
22 echo
23 echo "Q-series [$LIMIT terms]:"
24 echo -n "${Q[1]} "              # 输出数列头两项.
25 echo -n "${Q[2]} "
26
27 for ((n=3; n <= $LIMIT; n++))  # C风格的循环条件.
28 do   # 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
32   let "n1 = $n - 1"         # n-1
33   let "n2 = $n - 2"         # n-2
34
35   t0=`expr $n - ${Q[n1]}`   # n - Q[n-1]
36   t1=`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
 7 Rows=5
 8 Columns=5
 9 # 5 X 5 的数组.
10
11 declare -a alpha     # char alpha [Rows] [Columns];
12                       # 没必要声明. 为什么?
13
14 load_alpha ()
15 {
16 local rc=0
17 local index
18
19 for i in A B C D E F G H I J K L M N O P Q R S T U V W X Y
20 do     # 你可以随你的心意, 使用任意符号.
21   local row=`expr $rc / $Columns`
22   local column=`expr $rc % $Rows`
23   let "index = $row * $Rows + $column"
24   alpha[$index]=$i
25 # alpha[$row][$column]
26   let "rc += 1"
```

```
 27 done
 28
 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                             #+ 行号不变(外层循环),
 43                                #+ 列号进行增长. (译者注: 就是按行打印)
 44   local column=0
 45
 46   echo -n "       "            #  按照行方向打印"正方形"数组.
 47
 48   while [ "$column" -lt "$Columns" ]
 49   do
 50     let "index = $row * $Rows + $column"
 51     echo -n "${alpha[index]} "  # alpha[$row][$column]
 52     let "column += 1"
 53   done
 54
 55   let "row += 1"
 56   echo
 57
 58 done
 59
 60 # 更简单的等价写法为:
 61 #     echo ${alpha[*]} | xargs -n $Columns
 62
 63 echo
 64 }
 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

---

27. /dev和/proc

目录
27.1. /dev
27.2. /proc

Linux或者UNIX机器典型地都带有/dev和/proc目录，用于特殊目的.

27.1. /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.

```
bash$ df
Filesystem           1k-blocks      Used Available Use%
 Mounted on
 /dev/hda6                495876    222748    247527  48% /
 /dev/hda1                 50755      3887     44248   9% /boot
 /dev/hda8                367013     13262    334803   4% /home
 /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:

```
bash$ cat </dev/tcp/time.nist.gov/13
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:

```
bash$ exec 5<>/dev/tcp/www.net.cn/80
bash$ echo -e "GET / HTTP/1.0\n" >&5
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
 7 TCP_HOST=www.dns-diy.com    # 一个已知的对垃圾邮件友好的ISP.
 8 TCP_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 [ "X$MYEXIT" = "X0" ]; 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.

27.2. /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*以及与它们相关的状态与统计信息.

```
bash$ cat /proc/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 ttyS
  5 cua
  7 vcs
 10 misc
 14 sound
 29 fb
 36 netlink
128 ptm
136 pts
162 raw
254 pcmcia

Block devices:
  1 ramdisk
  2 fd
  3 ide0
  9 md


bash$ cat /proc/interrupts
           CPU0
  0:      84505          XT-PIC  timer
  1:       3375          XT-PIC  keyboard
  2:          0          XT-PIC  cascade
  5:          1          XT-PIC  soundblaster
  8:          1          XT-PIC  rtc
 12:       4231          XT-PIC  PS/2 Mouse
 14:     109373          XT-PIC  ide0
NMI:          0
ERR:          0


bash$ cat /proc/partitions
major minor  #blocks  name     rio rmerge rsect ruse wio wmerge wsect wuse running
use aveq

   3     0    3007872 hda 4472 22260 114520 94240 3551 18703 50384 549710 0 111550
644030
   3     1      52416 hda1 27 395 844 960 4 2 14 180 0 800 1140
   3     2          1 hda2 0 0 0 0 0 0 0 0 0 0 0
   3     4     165280 hda4 10 0 20 210 0 0 0 0 0 210 210
   ...


bash$ cat /proc/loadavg
0.13 0.42 0.27 2/44 1119


bash$ cat /proc/apm
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 )
```

```
CPU=$( awk '/model name/ {print $4}' < /proc/cpuinfo )

if [ $CPU = Pentium ]
then
  run_some_commands
  ...
else
  run_different_commands
  ...
fi
```

```
devfile="/proc/bus/usb/devices"
USB1="Spd=12"
USB2="Spd=480"


bus_speed=$(grep Spd $devfile | awk '{print $9}')

if [ "$bus_speed" = "$USB1" ]
then
  echo "USB 1.1 port found."
  # 可以在这里添加操作USB 1.1的相关动作.
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

```
#!/bin/bash
# pid-identifier.sh: 给出与指定pid相关联进程的完整路径名.

ARGNO=1  # 期望的参数个数.
E_WRONGARGS=65
E_BADPID=66
E_NOSUCHPROCESS=67
E_NOPERMISSION=68
PROCFILE=exe

if [ $# -ne $ARGNO ]
then
  echo "Usage: `basename $0` PID-number" >&2  # Error message >stderr(错误信息重定向到标准错误).
  exit $E_WRONGARGS
fi

pidno=$( ps ax | grep $1 | awk '{ print $1 }' | grep $1 )
# 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.
# 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.
# 最后的"grep $1"就是用来过滤掉这种可能性.
#
#    pidno=$( ps ax | awk '{ print $1 }' | grep $1 )
#    这么写就可以了, 这一点由Teemu Huovila指出.

if [ -z "$pidno" ]  # 如果经过所有的过滤之后, 得到的结果是一个长度为0的字符串,
then                # 那就说明这个pid没有相应的进程在运行.
  echo "No such process running."
  exit $E_NOSUCHPROCESS
fi

# 也可以这么写:
#   if ! ps $1 > /dev/null 2>&1
#   then                # 没有与给定pid相匹配的进程在运行.
#     echo "No such process running."
#     exit $E_NOSUCHPROCESS
#    fi

# 为了简化整个过程, 可以使用"pidof".


if [ ! -r "/proc/$1/$PROCFILE" ]  # 检查读权限.
then
  echo "Process $1 running, but..."
  echo "Can't get read permission on /proc/$1/$PROCFILE."
  exit $E_NOPERMISSION  # 一般用户不能访问/proc目录下的某些文件.
fi
```

```
47
48 # 最后两个测试可以使用下面的语句来代替:
49 #    if ! kill -0 $1 > /dev/null 2>&1 # '0'不是一个信号, but
50                                       # 但是这么做, 可以测试一下是否
51                                       # 可以向该进程发送信号.
52 #    then echo "PID doesn't exist or you're not its owner" >&2
53 #    exit $E_BADPID
54 #    fi
55
56
57
58 exe_file=$( ls -l /proc/$1 | grep "exe" | awk '{ print $11 }' )
59 # 或       exe_file=$( ls -l /proc/$1/exe | awk '{print $11}' )
60 #
61 # /proc/pid-number/exe是一个符号链接,
62 # 指向这个调用进程的完整路径名.
63
64 if [ -e "$exe_file" ]  # 如果/proc/pid-number/exe存在...
65 then                   # 那么相应的进程就存在.
66   echo "Process #$1 invoked by $exe_file."
67 else
68   echo "No such process running."
69 fi
70
71
72 # 这个精心制作的脚本, *几乎*能够被下边这一行所替代:
73 # ps ax | grep $1 | awk '{ print $5 }'
74 # 但是, 这样并不会工作...
75 # 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, 即调用时程序用的程序路径本
身.)
76 # 而不是可执行文件的路径.
77 #
78 # 然而, 下边这两种方法都能正确地完成这个任务.
79 #       find /proc/$1/exe -printf '%l\n'
80 #       lsof -aFn -p $1 -d txt | sed -ne 's/^n//p'
81
82 # 附加注释, 是Stephane Chazelas添加的.
83
84 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 | awk '{ print
$1 }' )
  9 # 找出'pppd'所对应的进程号, 即'ppp守护进程'.
 10 # 必须过滤掉由搜索本身所产生的进程行.
 11 #
 12 #  然而, 就像Oleg Philon所指出的那样,
 13 #+ 如果使用"pidof"的话, 就会非常简单.
 14 #  pidno=$( pidof $PROCNAME )
 15 #
 16 #  从经验中总结出来的忠告:
 17 #+ 当命令序列变得非常复杂的时候, 一定要找到更简洁的方法.
 18
 19
 20 if [ -z "$pidno" ]   # 如果没有找到匹配的pid, 那么就说明相应的进程没运行.
 21 then
 22   echo "Not connected."
 23   exit $NOTCONNECTED
 24 else
 25   echo "Connected."; echo
 26 fi
 27
 28 while [ true ]       # 死循环, 这里可以改进一下.
 29 do
 30
 31   if [ ! -e "/proc/$pidno/$PROCFILENAME" ]
 32   # 进程运行时, 相应的"status"文件就会存在.
 33   then
 34     echo "Disconnected."
 35     exit $NOTCONNECTED
 36   fi
 37
 38 netstat -s | grep "packets received"  # 获得一些连接统计.
 39 netstat -s | grep "packets delivered"
```

```
40
41
42   sleep $INTERVAL
43   echo; echo
44
45 done
46
47 exit 0
48
49 # 如果你想停止这个脚本, 只能使用Control-C.
50
51 #     练习:
52 #     -----
53 #     改进这个脚本, 使它可以按"q"键退出.
54 #     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.

28. 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, 但是不会产生新进程.(译者注: 因为是内建的)
3
4 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
 4 ROOT_UID=0          # Root用户的$UID为0.
 5 E_WRONG_USER=65     # 不是root?
 6
 7 FILE=/swap
 8 BLOCKSIZE=1024
 9 MINBLOCKS=40
10 SUCCESS=0
11
12
13 # 这个脚本必须以root身份来运行.
14 if [ "$UID" -ne "$ROOT_UID" ]
15 then
16   echo; echo "You must be root to run this script."; echo
17   exit $E_WRONG_USER
18 fi
19
20
21 blocks=${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
34 if [ "$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, 会丢失数组.
 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
15 E_NON_ROOT_USER=70               # 必须以root身份来运行.
16 ROOTUSER_NAME=root
17
18 MOUNTPT=/mnt/ramdisk
19 SIZE=2000                        # 2K个块(可以进行适当的修改)
20 BLOCKSIZE=1024                   # 每块的大小为1K(1024字节)
21 DEVICE=/dev/ram0                 # 第一个ram设备
22
23 username=`id -nu`
```

```
 24 if [ "$username" != "$ROOTUSER_NAME" ]
 25 then
 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.

---

29. 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.

例子 29-1. 一个错误脚本

```
 1 #!/bin/bash
 2 # ex74.sh
 3
 4 # 这是一个错误脚本.
 5 # 哪里出了错?
 6
 7 a=37
 8
 9 if [$a -gt 27 ]
10 then
11   echo $a
12 fi
13
14 exit 0
```

脚本的输出:

```
./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 #!/bin/bash
 2 # missing-keyword.sh: 这个脚本会产生什么错误?
 3
 4 for a in 1 2 3
 5 do
 6   echo "$a"
 7 # done     # 第7行上的关键字done'被注释掉了.
 8
 9 exit 0
```

脚本的输出:

```
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 #!/bin/bash
 2
```

```
 3 #   这个脚本的目的是删除当前目录下的某些文件,
 4 #+  这些文件特指那些文件名包含空格的文件.
 5 #   但是不能如我们所愿的那样工作.
 6 #   为什么?
 7
 8
 9 badname=`ls | grep ' '`
10
11 # 试试这个:
12 # echo "$badname"
13
14 rm "$badname"
15
16 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 ### debecho (debug-echo), 由Stefano Falsetto编写 ###
   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 debecho () {
   4   if [ ! -z "$DEBUG" ]; then
   5      echo "$1" >&2
   6      #          ^^^ 打印到stderr
   7   fi
   8 }
   9
  10 DEBUG=on
  11 Whatever=whatnot
  12 debecho $Whatever   # whatnot
  13
  14 DEBUG=
  15 Whatever=notwhat
  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 #!/bin/bash
 2 # assert.sh
 3
 4 assert ()                 #  如果条件为false,
 5 {                         #+ 那么就打印错误信息并退出脚本.
 6   E_PARAM_ERR=98
 7   E_ASSERT_FAILED=99
 8
 9
10   if [ -z "$2" ]          # 传递进来的参数个数不够.
11   then
12     return $E_PARAM_ERR   # 什么都不做就return.
13   fi
14
15   lineno=$2
16
17   if [ ! $1 ]
18   then
19     echo "Assertion failed:  \"$1\""
20     echo "File \"$0\", line $lineno"
21     exit $E_ASSERT_FAILED
22   # else
23   #   返回
24   #   然后继续执行脚本余下的代码.
25   fi
26 }
27
28
29 a=5
30 b=4
31 condition="$a -lt $b"      # 产生错误信息并退出脚本.
32                            #  尝试把这个"条件"放到其他的地方,
33                            #+ 然后看看发生了什么.
34
35 assert "$condition" $LINENO
36 # 只有在"assert"成功时, 脚本余下的代码才会继续执行.
37
38
39 # 这里放置的是其他的一些命令.
40 # ...
41 echo "This statement echoes only if the \"assert\" does not fail."
42 # ...
43 # 这里也放置其他一些命令.
44
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.

```
trap '' 2
# 忽略中断2(Control-C), 没有指定处理动作.

trap 'echo "Control-C disabled."' 2
# 当Control-C按下时, 显示一行信息.
```

例子 29-5. 捕获exit

```
#!/bin/bash
# 使用trap来捕捉变量值.

trap 'echo Variable Listing --- a = $a  b = $b' EXIT
#  EXIT是脚本中exit命令所产生信号的名字.
#
#  "trap"所指定的命令并不会马上执行,
#+ 只有接收到合适的信号, 这些命令才会执行.

echo "This prints before the \"trap\" --"
echo "even though the script sees the \"trap\" first."
echo

a=39

b=36

exit 0
#  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,
#+ 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

```
#!/bin/bash
# logon.sh: 一个检查你是否在线的脚本, 这个脚本实现的很简陋.

umask 177  # 确保temp文件并不是所有用户都有权限访问.


TRUE=1
LOGFILE=/var/log/messages
#  注意: $LOGFILE必须是可读的
#+ (使用root身份来执行, chmod 644 /var/log/messages).
TEMPFILE=temp.$$
#  使用脚本的进程ID, 来创建一个"唯一"的临时文件名.
#      也可以使用'mktemp'.
#      比如:
#      TEMPFILE=`mktemp temp.XXXXXX`
KEYWORD=address
#  登陆时, 把"remote IP address xxx.xxx.xxx.xxx"
#                       添加到/var/log/messages中.
ONLINE=22
USER_INTERRUPT=13
CHECK_LINES=100
#  日志文件有多少行需要检查.

trap 'rm -f $TEMPFILE; exit $USER_INTERRUPT' TERM INT
#  如果脚本被control-c中途中断的话, 那么就清除掉临时文件.

echo

while [ $TRUE ]  #死循环.
do
  tail -$CHECK_LINES $LOGFILE> $TEMPFILE
  #  将系统日志文件的最后100行保存到临时文件中.
  #  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.
  search=`grep $KEYWORD $TEMPFILE`
  #  检查是否存在"IP address"片断,
  #+ 它用来提示, 这是一次成功的网络登陆.

  if [ ! -z "$search" ] #  必须使用引号, 因为变量可能会包含一些空白符.
  then
     echo "On-line"
     rm -f $TEMPFILE    #  清除临时文件.
     exit $ONLINE
  else
     echo -n "."        #  echo的-n选项不会产生换行符.
                        #+ 这样你就可以在一行上连续打印.
  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 exit 0
60
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 exit $?
14
15 #  "trap 'command1 . . . command2 . . .' DEBUG"结构更适合于
16 #+ 使用在复杂脚本的上下文中,
17 #+ 如果在这种情况下大量使用"echo $variable"语句的话,
18 #+ 将会非常笨拙, 而且很耗时.
19
20 # 感谢, Stephane Chazelas指出这点.
21
22
23 脚本的输出:
24
25 VARIABLE-TRACE> $variable = ""
26 VARIABLE-TRACE> $variable = "29"
27 Just initialized "$variable" to 29.
28 VARIABLE-TRACE> $variable = "29"
29 VARIABLE-TRACE> $variable = "87"
30 Just multiplied "$variable" by 3.
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)

```
#!/bin/bash
# parent.sh
# 在多处理器(SMP box)的机器里运行多进程.
# 作者: Tedman Eng

#  我们下面要介绍两个脚本, 这是第一个,
#+ 这两个脚本都要放到当前工作目录下.



LIMIT=$1          # 想要启动的进程总数
NUMPROC=4         # 并发的线程数量(forks?)
PROCID=1          # 启动的进程ID
echo "My PID is $$"

function start_thread() {
        if [ $PROCID -le $LIMIT ] ; then
                ./child.sh $PROCID&
                let "PROCID++"
        else
           echo "Limit reached."
           wait
           exit
        fi
}

while [ "$NUMPROC" -gt 0 ]; do
        start_thread;
        let "NUMPROC--"
done


while true
do

trap "start_thread" SIGRTMIN

done

exit 0



# ======== 下面是第二个脚本 ========


#!/bin/bash
# child.sh
# 在SMP box上运行多进程.
# 这个脚本会被parent.sh调用.
# 作者: Tedman Eng

temp=$RANDOM
index=$1
shift
let "temp %= 5"
let "temp += 4"
echo "Starting $index  Time:$temp" "$@"
sleep ${temp}
echo "Ending $index"
kill -s SIGRTMIN $PPID

exit 0


# ======================== 脚本作者注 ======================== #
#  这个脚本并不是一点bug都没有.
#  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,
#+ 有些并发线程消失了!
#  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.
#  一旦接收到捕捉的信号, 那么在下一次捕捉到来之前,
#+ 会有一段短暂的时间来执行信号处理程序,
#+ 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, 因此失去一个产生子进程的机会.

#  毫无疑问的, 肯定有人能够找出产生这个bug的原因,
```

```
#+ 并且在将来的某个时候. . . 修正它.


# ============================================================================ #



# ----------------------------------------------------------------------------#



######################################################################
# 下面的脚本是由Vernia Damiano原创.
# 不幸的是, 它并不能正常工作.
######################################################################

#!/bin/bash

#  要想调用这个脚本, 至少需要一个整形参数
#+ (并发的进程数).
#  所有的其他参数都传递给要启动的进程.


INDICE=8        # 想要启动的进程数目
TEMPO=5         # 每个进程最大的睡眠时间
E_BADARGS=65    # 如果没有参数传到脚本中, 那么就返回这个错误码.

if [ $# -eq 0 ] # 检查一下, 至少要传递一个参数给脚本.
then
  echo "Usage: `basename $0` number_of_processes [passed params]"
  exit $E_BADARGS
fi

NUMPROC=$1              # 并发进程数
shift
PARAMETRI=( "$@" )      # 每个进程的参数

function avvia() {
         local temp
         local index
         temp=$RANDOM
         index=$1
         shift
         let "temp %= $TEMPO"
         let "temp += 1"
         echo "Starting $index Time:$temp" "$@"
         sleep ${temp}
         echo "Ending $index"
         kill -s SIGRTMIN $$
}

function parti() {
         if [ $INDICE -gt 0 ] ; then
              avvia $INDICE "${PARAMETRI[@]}" &
                let "INDICE--"
         else
                trap : SIGRTMIN
         fi
}

trap parti SIGRTMIN

while [ "$NUMPROC" -gt 0 ]; do
         parti;
         let "NUMPROC--"
done

wait
trap - SIGRTMIN

exit $?

: <<SCRIPT_AUTHOR_COMMENTS
我需要使用指定的选项来运行一个程序,
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.
所以我想(我也会)运行指定数目个进程,
并且每个进程终止之后, 都要启动一个新进程.

"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,
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,
并且使用了"trap"指令.
  --Vernia Damiano
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     trap '' 2  # 信号2就是Control-C, 现在被禁用了.
2     command
3     command
4     command
5     trap 2     # 重新恢复Control-C
6
```

Bash3.0之后增加了如下这些特殊变量用于调试.

1. $BASH_ARGC
2. $BASH_ARGV
3. $BASH_COMMAND
4. $BASH_EXECUTION_STRING
5. $BASH_LINENO
6. $BASH_SOURCE
7. $BASH_SUBSHELL

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.

30. 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.

```
#!/bin/bash

set -o verbose
# 打印出所有执行前的命令.

```

```
#!/bin/bash

set -v
# 与上边的例子具有相同的效果.

```

如果你想在脚本中*禁用*某个选项, 可以使用set +o option-name或set +option-abbrev.

```
#!/bin/bash

set -o verbose
# 激活命令回显.
command
...
command

set +o verbose
# 禁用命令回显.
command
# 没有命令回显了.


set -v
# 激活命令回显.
command
...
command

set +v
# 禁用命令回显.
command

exit 0

```

还有另一种可以在脚本中启用选项的方法, 那就是在脚本头部, *#!*的后边直接指定选项.

```
#!/bin/bash -x
#
# 下边是脚本的主要内容.

```

也可以从命令行中打开脚本的选项. 某些不能与set命令一起用的选项就可以使用这种方法来打开. *-i*就是其中之一, 这个选项用来强制脚本以交互的方式运行.

```
bash -v script-name

bash -o verbose script-name
```

下表列出了一些有用的选项. 它们都可以使用缩写的形式来指定(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	noclobber	防止重定向时覆盖文件(可能会被>|覆盖)
-D	(none)	列出用双引号引用起来的，以$为前缀的字符串，但是不执行脚本中的命令
-a	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.

31. 陷阱

Turandot: Gli enigmi sono tre, la morte una!

Caleph: No, no! Gli enigmi sono tre, una la vita!

Puccini

- 将保留字或特殊字符声明为变量名.

```
case=value0        # 引发错误.
23skidoo=value1    # 也会引发错误.
# 以数字开头的变量名是被shell保留使用的.
# 试试_23skidoo=value1. 以下划线开头的变量就没问题.

# 然而 . . .    如果只用一个下划线作为变量名就不行.
_=25
echo $_            # $_是一个特殊变量, 代表最后一个命令的最后一个参数.

xyz((!*=value2     # 引起严重的错误.
# 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).

```
var-1=23
# Use 'var_1' instead.

function-whatever ()    # 错误
# 使用'function_whatever ()'来代替.


# Bash3.0之后, 标点不能出现在函数名中.
function.whatever ()    # 错误
# 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.

```
do_something ()
{
  echo "This function does something with \"$1\"."
}

do_something=do_something

do_something do_something

# 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.

```
var1 = 23   # 'var1=23'才是正确的.
# 对于上边这一行来说, Bash会把"var1"当作命令来执行,
# "="和"23"会被看作"命令""var1"的参数.

let c = $a - $b   # 'let c=$a-$b'或'let "c = $a - $b"'才是正确的.

if [ $a -le 5]    # if [ $a -le 5 ]   是正确的.
# if [ "$a" -le 5 ]   这么写更好.
# [[ $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.

```
{ ls -l; df; echo "Done." }
# bash: syntax error: unexpected end of file

{ 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
 3 echo "uninitialized_var = $uninitialized_var"
 4 # uninitialized_var =
```

- 混淆测试符号=和*-eq*. 请记住, =用于比较字符变量, 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
 4 echo
 5 number=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
10 while [ "$number" < 5 ]    # 错! 应该是:  while [ "$number" -lt 5 ]
11 do
12   echo -n "$number "
13   let "number += 1"
14 done
15 #  如果你企图运行这个错误的脚本, 那么就会得到一个错误消息:
16 #+ bad-op.sh: line 10: 5: No such file or directory
17 #  在单中括号结构([ ])中, "<"必须被转义.
18 #+ 即便如此, 比较两个整数还是错误的.
19
20
21 echo "----------------------"
22
23
24 while [ "$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机器上却不一定会这么做.

- 使用Bash未文档化的特征, 将是一种危险的举动. 本书之前的几个版本就依赖一个这种"特征", 下面说明一下这个"特征", 虽然exit或return所能返回的最大正值为255, 但是并没有限制我们使用负整数. 不幸的是, Bash 2.05b之后的版本, 这个漏洞消失了. 请参考例子 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
```

```
bash$ echo $WHATEVER

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
 6 a=aaa
 7 b=bbb
 8 c=ccc
 9
10 echo "one two three" | read a b c
```

```
 11 # 尝试重新给变量a, b, 和c赋值.
 12
 13 echo
 14 echo "a = $a"  # a = aaa
 15 echo "b = $b"  # b = bbb
 16 echo "c = $c"  # c = ccc
 17 # 重新赋值失败.
 18
 19 # ------------------------------
 20
 21 # 试试下边这种方法.
 22
 23 var=`echo "one two three"`
 24 set -- $var
 25 a=$1; b=$2; c=$3
 26
 27 echo "-------"
 28 echo "a = $a"  # a = one
 29 echo "b = $b"  # b = two
 30 echo "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

注意事项

[1] 给脚本设置*suid*权限是没用的.

32. 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*.

32.1. 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.

```
PASS="$PASS${MATRIX:$(($RANDOM%${#MATRIX})):1}"
# 去年你写下这段代码的时候，你非常了解这段代码的含义，但现在它对你来说完全是个谜.
# (摘自Antek Sawicki的"pw.sh"脚本.)
```

给脚本和函数加上描述性的头信息.

```
#!/bin/bash

#***********************************************#
#                    xyz.sh                     #
#            written by Bozo Bozeman            #
#                 July 05, 2001                 #
#                                               #
#            Clean up project files.            #
#***********************************************#

E_BADDIR=65                         # 没有这个目录.
projectdir=/home/bozo/projects      # 想要清除的目录.

# ------------------------------------------------------------ #
# cleanup_pfiles ()                                            #
# 删除指定目录中的所有文件.                                    #
# Parameter: $target_directory                                 #
# 返回值: 0表示成功，失败返回$E_BADDIR.                        #
# ------------------------------------------------------------ #
cleanup_pfiles ()
{
  if [ ! -d "$1" ]  # Test if target directory exists.
  then
    echo "$1 is not a directory."
    return $E_BADDIR
  fi

  rm -f "$1"/*
  return 0   # Success.
}

cleanup_pfiles $projectdir

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.

```
if [ -f /var/log/messages ]
then
  ...
fi
# 一年以后，你决定修改这个脚本，让它来检查/var/log/syslog.
# 到时候你就必须一行一行的手动修改这个脚本,
# 并且寄希望于没有遗漏的地方.

# 更好的办法是:
LOGFILE=/var/log/messages   # 只需要改动一行就行了.
if [ -f "$LOGFILE" ]
then
  ...
fi
```

- 给变量和函数起一些有意义的名字.

```
fl=`ls -al $dirname`                  # 含义模糊.
file_listing=`ls -al $dirname`        # 更好的名字.


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.
while [ "$index" -le "$MAXVAL" ]
...


```

```
E_NOTFOUND=75                                 #  错误码使用大写,
                                              #+ 并且命名的时候用"E_"作为前缀.
if [ ! -e "$filename" ]
then
  echo "File $filename not found."
  exit $E_NOTFOUND
fi


MAIL_DIRECTORY=/var/spool/mail/bozo   # 环境变量名使用大写.
export MAIL_DIRECTORY


GetAnswer ()                                  # 函数名采用大小写混合的方式.
{
  prompt=$1
  echo -n $prompt
  read answer
  return $answer
}

GetAnswer "What is your favorite number? "
favorite_number=$?
echo $favorite_number


_uservariable=23                              # 语法上可以这么起名, 但是不推荐.
# 用户定义的变量名最好不要以下划线开头.
# 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.

```
E_WRONG_ARGS=65
...
...
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.

```
-a      全部: 返回全部信息(包括隐藏的文件信息).
-b      摘要: 缩减版本, 通常用于其它版本. 通常用于其它脚本.
-c      拷贝, 连接, 等等.
-d      日常的: 使用全天的信息,
        而不仅仅是特定用户或特定实例的信息.
-e      扩展/详细描述: (通常不包括隐藏文件信息).
-h      帮助: 详细的使用方法, 附加信息, 讨论, 帮助.
        也请参考-V.
-l      打印出脚本的输出记录.
-m      手册: 显示基本命令的man页.
-n      数字: 仅使用数字数据.
-r      递归: 这个目录中所有的文件(也包含所有子目录).
-s      安装&文件维护: 这个脚本的配置文件.
-u      用法: 列出脚本的调用方法.
-v      详细信息: 只读输出, 或多或少的会做一些格式化.
-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.

```
COMMAND
if [ $? -eq 0 ]
...
# 多余, 而且不好理解.

if COMMAND
...
# 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.

33.1. 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.

```
#!/bin/bash
MY_PROMPT='$ '
while :
do
  echo -n "$MY_PROMPT"
  read line
  eval "$line"
  done

exit 0

# 这个例子脚本, 还有上面那么多的解释
# 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.)

```
if [ -z $PS1 ] # 没有提示符?
then
  # 非交互式
  ...
else
  # 交互式
  ...
fi
```

另一种办法, 脚本可以测试一下标志$-中是否存在选项"i".

```
case $- in
*i*)    # 交互式shell
;;
*)      # 非交互式shell
;;
# ("UNIX F.A.Q."的惯例, 1993)
```

使用#!/bin/bash -i头, 或者使用-i选项, 可以强制脚本运行在交互模式下. 注意, 这么做可能会让脚本产生古怪的行为, 有时候即使在没有错误的情况下, 也可能会显示错误信息.

33.2. 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

```
#!/bin/bash

# 这个简单的脚本可以把文件中所有的空行删除.
# 没做参数检查.
#
# 你或许想添加如下代码:
#
# E_NOARGS=65
# if [ -z "$1" ]
# then
#  echo "Usage: `basename $0` target-file"
#  exit $E_NOARGS
# fi


# 这个脚本调用起来的效果,
# 等价于从命令行上调用:
# sed -e '/^$/d' filename.

sed -e /^$/d "$1"
#  '-e'意味着后边跟的是"编辑"命令. (在这里是可选的).
#  '^'匹配行首, '$'匹配行尾.
#  这条语句用来匹配行首与行尾之间什么都没有的行,
#+ 即空白行.
#  'd'为删除命令.

#  将命令行参数引用起来,
#+ 就意味着可以在文件名中使用空白字符或者特殊字符.

#  注意, 这个脚本其实并不能真正的修改目标文件.
#  如果你想保存修改, 可以将它的输出重定向.

exit 0
```

例子 33-2. 稍微复杂一些的shell包装

```
#!/bin/bash

#  "替换", 这个脚本的用途:
#+ 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),
#+ 比如, "subst Smith Jones letter.txt".

ARGS=3          # 这个脚本需要3个参数.
E_BADARGS=65    # 传递给脚本的参数个数不对.

if [ $# -ne "$ARGS" ]
# 测试脚本的参数个数(这是个好办法).
then
  echo "Usage: `basename $0` old-pattern new-pattern filename"
  exit $E_BADARGS
fi

```

```
old_pattern=$1
new_pattern=$2

if [ -f "$3" ]
then
    file_name=$3
else
    echo "File \"$3\" does not exist."
    exit $E_BADARGS
fi


#  下面是实现功能的代码.

# ------------------------------------------------
sed -e "s/$old_pattern/$new_pattern/g" $file_name
# ------------------------------------------------

#  's'在sed中是替换命令,
#+ /pattern/表示匹配模式.
#  "g", 即全局标志, 用来自动替换掉每行中
#+ 出现的全部$old_pattern模式, 而不仅仅替换掉第一个匹配.
#  如果想深入了解, 可以参考'sed'命令的相关书籍.

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

```
#!/bin/bash
#  通用的shell包装,
#+ 执行一个操作, 然后把所作的操作写入到日志文件中.

# 需要设置如下两个变量.
OPERATION=
#         可以是一个复杂的命令链,
#+        比如awk脚本或者一个管道 . . .
LOGFILE=
#         命令行参数, 不管怎么样, 操作一般都需要参数. (译者注: 这行解释的是下面的OPTIONS变量, 不是LOGFILE.)


OPTIONS="$@"


# 记录下来.
echo "`date` + `whoami` + $OPERATION "$@"" >> $LOGFILE
# 现在, 执行操作.
exec $OPERATION "$@"

# 必须在操作执行之前, 记录到日志文件中.
# 为什么?
```

例子 33-4. 包装awd脚本的shell包装

```
#!/bin/bash
# pr-ascii.sh: 打印ASCII码的字符表.

START=33   # 可打印的ASCII字符的范围(十进制).
END=125

echo " Decimal   Hex     Character"   # 表头.
echo " -------   ---     ---------"

for ((i=START; i<=END; i++))
do
  echo $i | awk '{printf("  %3d       %2x          %c\n", $1, $1, $1)}'
# 在这种上下文中, 不会运行Bash内建的printf命令:
#     printf "%c" "$i"
done

exit 0


#  十进制   16进制     字符
#  -------  ------   ---------
#    33       21         !
#    34       22         "
#    35       23         #
```

```
#    36       24         $
#
#    . . .
#
#   122       7a         z
#   123       7b         {
#   124       7c         |
#   125       7d         }


#  将脚本的输出重定向到一个文件中,
#+ 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

```
#!/bin/bash

# 给目标文件添加(由数字组成的)指定的一列.

ARGS=2
E_WRONGARGS=65

if [ $# -ne "$ARGS" ] # 检查命令行参数个数是否正确.
then
   echo "Usage: `basename $0` filename column-number"
   exit $E_WRONGARGS
fi

filename=$1
column_number=$2

#  将shell变量传递给脚本的awk部分, 需要一点小技巧.
#  一种办法是, 将awk脚本中的Bash脚本变量,
#+ 强引用起来.
#      $'$BASH_SCRIPT_VAR'
#       ^                ^
#  在下面的内嵌awd脚本中, 就会这么做.
#  请参考awk的相关文档来了解更多的细节.

# 多行awk脚本的调用格式为:  awk ' ..... '


# 开始awk脚本.
# ------------------------------
awk '

{ total += $'"${column_number}"'
}
END {
     print total
}

' "$filename"
# ------------------------------
# 结束awk脚本.


#   将shell变量传递给内嵌awk脚本可能是不安全的,
#+  所以Stephane Chazelas提出了下边这种替代方法:
#   ---------------------------------------
#   awk -v column_number="$column_number" '
#   { total += $column_number
#   }
#   END {
#       print total
#   }' "$filename"
#   ---------------------------------------


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

```
#!/bin/bash

# Shell命令可以放到Perl脚本的前面.
echo "This precedes the embedded Perl script within \"$0\"."
echo "============================================================="

perl -e 'print "This is an embedded Perl script.\n";'
# 类似于sed, Perl也可以使用"-e"选项.

echo "============================================================="
echo "However, the script may also contain shell and system commands."

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

```
#!/bin/bash
# bashandperl.sh

echo "Greetings from the Bash part of the script."
# 这里可以放置更多的Bash命令.

exit 0
# 脚本的Bash部分结束.

# ====================================================

#!/usr/bin/perl
# 脚本的这部分必须使用-x选项来调用.

print "Greetings from the Perl part of the script.\n";
# 这里可以放置更多的Perl命令.

# 脚本的Perl部分结束.
```

```
bash$ bash bashandperl.sh
Greetings from the Bash part of the script.


bash$ perl -x bashandperl.sh
Greetings from the Perl part of the script.
```

注意事项

[1] 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.

33.3. 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 a=8
  2
  3 # 下面所有的比较都是等价的.
  4 test "$a" -lt 16 && echo "yes, $a < 16"          # "与列表"
  5 /bin/test "$a" -lt 16 && echo "yes, $a < 16"
  6 [ "$a" -lt 16 ] && echo "yes, $a < 16"
  7 [[ $a -lt 16 ]] && echo "yes, $a < 16"           # 在[[ ]]和(( ))结构中,
  8 (( a < 16 )) && echo "yes, $a < 16"              # 不用将变量引用起来.
  9
 10 city="New York"
 11 # 同样地, 下面所有的比较都是等价的.
 12 test "$city" \< Paris && echo "Yes, Paris is greater than $city"  # 按照ASCII顺序
比较.
 13 /bin/test "$city" \< Paris && echo "Yes, Paris is greater than $city"
 14 [ "$city" \< Paris ] && echo "Yes, Paris is greater than $city"
 15 [[ $city < Paris ]] && echo "Yes, Paris is greater than $city"    # 不需要引
用$city.
 16
 17 # 感谢, S.C.
```

33.4. 递归

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

```
#!/bin/bash
# recurse.sh

#   脚本能否递归地调用自己?
#   是的, 但这有什么实际的用处吗?
#   (看下面的.)

RANGE=10
MAXVAL=9

i=$RANDOM
let "i %= $RANGE"  # 在0到$RANGE - 1之间, 产生一个随机数.

if [ "$i" -lt "$MAXVAL" ]
then
  echo "i = $i"
  ./$0              #   脚本递归地产生自己的一个新实例, 并调用.
fi                  #   每个子脚本都做同样的事情, until
                    #+ 直到产生的变量$i等于$MAXVAL为止.

#   如果使用"while"循环来代替"if/then"测试结构的话, 会产生问题.
#   解释一下为什么.

exit 0

# 注意:
# -----
# 脚本想要正常的工作, 就必须具备可执行权限.
# 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.
# 解释一下原因.
```

例子 33-9. 递归调用自身的(有用的)脚本

```
#!/bin/bash
# pb.sh: 电话本

# 由Rick Boivie编写, 已经得到作者授权, 可以在本书中使用.
# 本书作者做了一些修改.

MINARGS=1      #   脚本至少需要一个参数.
DATAFILE=./phonebook
               #   当前目录下,
               #+ 必须有一个名字为"phonebook"的数据文件.
PROGNAME=$0
E_NOARGS=70    #   未传递参数错误.

if [ $# -lt $MINARGS ]; then
      echo "Usage: "$PROGNAME" data"
      exit $E_NOARGS
fi


if [ $# -eq $MINARGS ]; then
      grep $1 "$DATAFILE"
      # 如果文件$DATAFILE不存在, 'grep'就会打印一个错误信息.
else
      ( shift; "$PROGNAME" $* ) | grep $1
      # 脚本递归调用自身.
fi

exit 0         #   脚本在此退出.
               #   因此, 在这句之后,
                     #+ 即使不加"#"号, 也可以添加注释和数据.

```

```
# ------------------------------------------------------------------------
"phonebook"数据文件的例子:

John Doe        1555 Main St., Baltimore, MD 21228          (410) 222-3333
Mary Moe        9899 Jones Blvd., Warren, NH 03787          (603) 898-3232
Richard Roe     856 E. 7th St., New York, NY 10009          (212) 333-4567
Sam Roe         956 E. 8th St., New York, NY 10009          (212) 444-5678
Zoe Zenobia     4481 N. Baker St., San Francisco, SF 94338  (415) 501-1631
# ------------------------------------------------------------------------

$bash pb.sh Roe
Richard Roe     856 E. 7th St., New York, NY 10009          (212) 333-4567
Sam Roe         956 E. 8th St., New York, NY 10009          (212) 444-5678

$bash pb.sh Roe Sam
Sam Roe         956 E. 8th St., New York, NY 10009          (212) 444-5678

#  如果给脚本传递的参数超过了一个,
#+ 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

```
#!/bin/bash
# usrmnt.sh, 由Anthony Richardson编写,
# 经过作者授权, 可以在本书中使用.

# 用法:        usrmnt.sh
# 描述: 挂载设备, 调用这个脚本的用户必须属于
#               /etc/sudoers文件中的MNTUSERS组.

# ------------------------------------------------------------
#  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.
#  只有拥有合适权限的用户才能使用

#   usermount /dev/fd0 /mnt/floppy

#  来代替

#   sudo usermount /dev/fd0 /mnt/floppy

#  我使用相同的技术来处理我所有的sudo脚本,
#+ 因为我觉得它很方便.
# ------------------------------------------------------------

#  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
#+ (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. 传递用户的真实id和组id . . .

if [ -z "$SUDO_COMMAND" ]
then
   mntusr=$(id -u) grpusr=$(id -g) sudo $0 $*
   exit 0
fi

# 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), 那么我们就会运行到这里.
/bin/mount $* -o uid=$mntusr,gid=$grpusr

exit 0

# 附注(脚本作者添加的):
# ------------------------------------------------

# 1) Linux允许在/etc/fstab文件中使用"users"选项,
#    以便于任何用户都可以挂载可移动设备.
#    但是, 在服务器上,
#    我希望只有一小部分用户可以访问可移动设备.
#    我发现使用sudo可以给我更多的控制空间.

# 2) 我还发现, 通过使用组,
#    我能够更容易的完成这个任务.

# 3) 这个方法可以将root访问mount命令的权利,
#    赋予任何具有合适权限的用户,
#    所以一定要小心那些被你赋予访问权限的用户.
#    你可以开发出类似于mntfloppy, mntcdrom,
#    和mntsamba脚本, 将访问类型分类,
#    然后你就可以使用上面所讲的这种技术,
#    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.

33.5. 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

```
#!/bin/bash
# ex30a.sh: 脚本ex30.sh的"彩色"版本.
#             没被加工处理过的地址数据库


clear                                          # 清屏.

echo -n "           "
echo -e '\E[37;44m'"\033[1mContact List\033[0m"
                                               # 在蓝色背景下的白色.
echo; echo
echo -e "\033[1mChoose one of the following persons:\033[0m"
                                               # 粗体
tput sgr0
echo "(Enter only the first letter of name.)"
echo
echo -en '\E[47;34m'"\033[1mE\033[0m"    # 蓝色
tput sgr0                                      # 将颜色重置为"常规".
echo "vans, Roland"                            # "[E]vans, Roland"
echo -en '\E[47;35m'"\033[1mJ\033[0m"    # 红紫色
tput sgr0
echo "ones, Mildred"
echo -en '\E[47;32m'"\033[1mS\033[0m"    # 绿色
tput sgr0
echo "mith, Julie"
echo -en '\E[47;31m'"\033[1mZ\033[0m"    # 红色
tput sgr0
echo "ane, Morris"
echo

read person

case "$person" in
# 注意, 变量被引用起来了.

  "E" | "e" )
  # 大小写的输入都能接受.
  echo
  echo "Roland Evans"
  echo "4321 Floppy Dr."
  echo "Hardscrabble, CO 80753"
  echo "(303) 734-9874"
  echo "(303) 734-9892 fax"
  echo "revans@zzy.net"
  echo "Business partner & old friend"
  ;;

  "J" | "j" )
  echo
  echo "Mildred Jones"
  echo "249 E. 7th St., Apt. 19"
  echo "New York, NY 10009"
  echo "(212) 533-2814"
  echo "(212) 533-9972 fax"
  echo "milliej@loisaida.com"
  echo "Girlfriend"
  echo "Birthday: Feb. 11"
  ;;

# 稍后为Smith & Zane添加信息.

          * )
   # 默认选项.
   # 空输入(直接按回车)也会在这被匹配.
   echo
```

```
   echo "Not yet in database."
  ;;

esac

tput sgr0                                     # 将颜色重置为"常规".

echo

exit 0
```

例子 33-12. 画一个盒子

```
#!/bin/bash
# Draw-box.sh: 使用ASCII字符画一个盒子.

# 由Stefano Palmeri编写, 本书作者做了少量修改.
# 经过授权, 可以在本书中使用.


##########################################################################
###  draw_box函数注释  ###

#  "draw_box"函数可以让用户
#+ 在终端上画一个盒子.
#
#  用法: draw_box ROW COLUMN HEIGHT WIDTH [COLOR]
#  ROW和COLUMN用来定位你想要
#+ 画的盒子的左上角.
#  ROW和COLUMN必须大于0,
#+ 并且要小于当前终端的尺寸.
#  HEIGHT是盒子的行数, 并且必须 >0 .
#  HEIGHT + ROW 必须 <= 终端的高度.
#  WIDTH是盒子的列数, 必须 >0 .
#  WIDTH + COLUMN 必须 <= 终端的宽度.
#
# 例如: 如果你的终端尺寸为20x80,
#  draw_box 2 3 10 45 是合法的
#  draw_box 2 3 19 45 的HEIGHT是错的 (19+2 > 20)
#  draw_box 2 3 18 78 的WIDTH是错的 (78+3 > 80)
#
#  COLOR是盒子边框的颜色.
#  这是第5个参数, 并且是可选的.
#  0=黑 1=红 2=绿 3=棕褐 4=蓝 5=紫 6=青 7=白.
#  如果你传递给函数的参数错误,
#+ 它将会退出, 并返回65,
#+ 不会有消息打印到stderr上.
#
#  开始画盒子之前, 会清屏.
#  函数内不包含清屏命令.
#  这样就允许用户画多个盒子, 甚至可以叠加多个盒子.

###  draw_box函数注释结束  ###
##########################################################################

draw_box(){

#==============#
HORZ="-"
VERT="|"
CORNER_CHAR="+"

MINARGS=4
E_BADARGS=65
#==============#


if [ $# -lt "$MINARGS" ]; then                  # 如果参数小于4, 退出.
    exit $E_BADARGS
fi

# 找出参数中非数字的字符.
# 还有其他更好的方法么(留给读者作为练习?).
if echo $@ | tr -d [:blank:] | tr -d [:digit:] | grep . &> /dev/null; then
   exit $E_BADARGS
fi

BOX_HEIGHT=`expr $3 - 1`    #  必须-1, 因为边角的"+"是
BOX_WIDTH=`expr $4 - 1`     #+ 高和宽共有的部分.
T_ROWS=`tput lines`         #  定义当前终端的
T_COLS=`tput cols`          #+ 长和宽的尺寸.
```

```
if [ $1 -lt 1 ] || [ $1 -gt $T_ROWS ]; then     #  开始检查参数
   exit $E_BADARGS                              #+ 是否正确.
fi
if [ $2 -lt 1 ] || [ $2 -gt $T_COLS ]; then
   exit $E_BADARGS
fi
if [ `expr $1 + $BOX_HEIGHT + 1` -gt $T_ROWS ]; then
   exit $E_BADARGS
fi
if [ `expr $2 + $BOX_WIDTH + 1` -gt $T_COLS ]; then
   exit $E_BADARGS
fi
if [ $3 -lt 1 ] || [ $4 -lt 1 ]; then
   exit $E_BADARGS
fi                                      # 参数检查结束.

plot_char(){                            # 函数内的函数.
   echo -e "\E[${1};${2}H"$3
}

echo -ne "\E[3${5}m"                    # 如果定义了, 就设置盒子边框的颜色.

# 开始画盒子

count=1                                          #  使用plot_char函数
for (( r=$1; count<=$BOX_HEIGHT; r++)); do       #+ 画垂直线.
  plot_char $r $2 $VERT
  let count=count+1
done

count=1
c=`expr $2 + $BOX_WIDTH`
for (( r=$1; count<=$BOX_HEIGHT; r++)); do
  plot_char $r $c $VERT
  let count=count+1
done

count=1                                          #  使用plot_char函数
for (( c=$2; count<=$BOX_WIDTH; c++)); do        #+ 画水平线.
  plot_char $1 $c $HORZ
  let count=count+1
done

count=1
r=`expr $1 + $BOX_HEIGHT`
for (( c=$2; count<=$BOX_WIDTH; c++)); do
  plot_char $r $c $HORZ
  let count=count+1
done

plot_char $1 $2 $CORNER_CHAR                     # 画盒子的角.
plot_char $1 `expr $2 + $BOX_WIDTH` +
plot_char `expr $1 + $BOX_HEIGHT` $2 +
plot_char `expr $1 + $BOX_HEIGHT` `expr $2 + $BOX_WIDTH` +

echo -ne "\E[0m"              #  恢复原来的颜色.

P_ROWS=`expr $T_ROWS - 1`     #  在终端的底部打印提示符.

echo -e "\E[${P_ROWS};1H"
}


# 现在, 让我们开始画盒子吧.
clear                         # 清屏.
R=2       # 行
C=3       # 列
H=10      # 高
W=45      # 宽
col=1     # 颜色(红)
draw_box $R $C $H $W $col   # 画盒子.

exit 0

# 练习:
# -----
# 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, \033[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.

```
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).

```
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.

```
bash$ echo -e '\E[34;47mThis prints in blue.'; tput sgr0

bash$ echo -e '\E[33;44m'"yellow text on blue background"; tput sgr0

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.

```
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

```
#!/bin/bash
# color-echo.sh: 使用颜色来显示文本消息.

# 可以按照你自己的目的来修改这个脚本.
# 这比将颜色数值写死更容易.
```

```
black='\E[30;47m'
red='\E[31;47m'
green='\E[32;47m'
yellow='\E[33;47m'
blue='\E[34;47m'
magenta='\E[35;47m'
cyan='\E[36;47m'
white='\E[37;47m'


alias Reset="tput sgr0"       #  不用清屏,
                              #+ 将文本属性重置为正常情况.


cecho ()                      # Color-echo.
                              # 参数$1 = 要显示的信息
                              # 参数$2 = 颜色
{
local default_msg="No message passed."
                              # 其实并不一定非的是局部变量.

message=${1:-$default_msg}    # 默认为default_msg.
color=${2:-$black}            # 如果没有指定, 默认为黑色.

  echo -e "$color"
  echo "$message"
  Reset                       # 重置文本属性.

  return
}


# 现在, 让我们试一下.
# ------------------------------------------------------
cecho "Feeling blue..." $blue
cecho "Magenta looks more like purple." $magenta
cecho "Green with envy." $green
cecho "Seeing red?" $red
cecho "Cyan, more familiarly known as aqua." $cyan
cecho "No color passed (defaults to black)."
       # 缺少$color参数.
cecho "\"Empty\" color passed (defaults to black)." ""
       # 空的$color参数.
cecho
       # 缺少$message和$color参数.
cecho "" ""
       # 空的$message和$color参数.
# ------------------------------------------------------

echo

exit 0

# 练习:
# -----
# 1) 为'cecho ()'函数添加"粗体"属性.
# 2) 为彩色背景添加选项.
```

例子 33-14. "赛马"游戏

```
#!/bin/bash
# horserace.sh: 一个非常简单的模拟赛马的游戏.
# 作者: Stefano Palmeri
# 经过授权可以在本书中使用.

##########################################################################
#   脚本目的:
#   使用转义序列和终端颜色进行游戏.
#
#   练习:
#   编辑脚本, 使它运行起来更具随机性,
#+  建立一个假的赌场 . . .
#   嗯 . . . 嗯 . . . 这种开场让我联想起一部电影 . . .
#
#   脚本将会给每匹马分配一个随机障碍.
#   按照马的障碍来计算几率,
#+  并且使用一种欧洲(?)的风格表达出来.
#   比如: 几率(odds)=3.75的话, 那就意味着如果你押$1,
#+  你就会赢得$3.75.
#
#             GNU/Linux                ,
```

```
   此脚本已经在          操作系统上测试过
#+ 测试终端有xterm, rxvt, 和konsole.
#  测试机器上安装的是AMD 900 MHz处理器,
#+ 平均比赛时间为75秒.
#  如果使用更快的机器, 那么比赛用时会更少.
#  所以, 如果你想增加比赛的悬念, 可以重置变量USLEEP_ARG.
#
#  本脚本由Stefano Palmeri编写.
######################################################################

E_RUNERR=65

# 检查一下md5sum和bc是否已经被安装.
if ! which bc &> /dev/null; then
   echo bc is not installed.
   echo "Can\'t run . . . "
   exit $E_RUNERR
fi
if ! which md5sum &> /dev/null; then
   echo md5sum is not installed.
   echo "Can\'t run . . . "
   exit $E_RUNERR
fi

#  设置下面的变量将会降低脚本的运行速度.
#  它会作为参数, 传递给usleep命令(man usleep),
#+ 并且单位是微秒(500000微秒 = 半秒).
USLEEP_ARG=0

#  如果脚本被Ctl-C中断, 那就清除临时目录,
#+ 恢复终端光标和终端颜色.
trap 'echo -en "\E[?25h"; echo -en "\E[0m"; stty echo;\
tput cup 20 0; rm -fr  $HORSE_RACE_TMP_DIR'  TERM EXIT
#  请参考与调试相关的章节, 可以获得'trap'命令的详细用法.

# 为脚本设置一个唯一的(实际上不是绝对唯一)临时目录名.
HORSE_RACE_TMP_DIR=$HOME/.horserace-`date +%s`-`head -c10 /dev/urandom | md5sum |
head -c30`

# 创建临时目录, 并移动到该目录下.
mkdir $HORSE_RACE_TMP_DIR
cd $HORSE_RACE_TMP_DIR


#  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$3.
#  例如: "move_and_echo 5 10 linux"等价与"tput cup 4 9; echo linux",
#+ 但是使用一个命令代替了两个命令.
#  注意: "tput cup"定义0 0位置, 为终端左上角,
#+ 而echo定义1 1位置, 为终端左上角.
move_and_echo() {
          echo -ne "\E[${1};${2}H""$3"
}

# 此函数用来产生1-9之间的伪随机数.
random_1_9 () {
                head -c10 /dev/urandom | md5sum | tr -d [a-z] | tr -d 0 | cut -c1
}

#  在画马的时候, 这两个函数用来模拟"移动".
draw_horse_one() {
               echo -n " "//$MOVE_HORSE//
}
draw_horse_two(){
              echo -n " "\\\\$MOVE_HORSE\\\\
}


# 定义当前终端尺寸.
N_COLS=`tput cols`
N_LINES=`tput lines`

# 至少需要一个20(行) X 80(列)的终端. 检查一下.
if [ $N_COLS -lt 80 ] || [ $N_LINES -lt 20 ]; then
   echo "`basename $0` needs a 80-cols X 20-lines terminal."
   echo "Your terminal is ${N_COLS}-cols X ${N_LINES}-lines."
   exit $E_RUNERR
fi


# 开始画赛场.

# 需要一个80字符的字符串. 见下面.
BLANK80=`seq -s "" 100 | head -c80`

clear
```

```
# 将前景色与背景色设为白.
echo -ne '\E[37;47m'

# 将光标移动到终端的左上角.
tput cup 0 0

# 画6条白线.
for n in `seq 5`; do
      echo $BLANK80         # 用这个80字符的字符串将终端变为彩色的.
done

# 将前景色设为黑色.
echo -ne '\E[30m'

move_and_echo 3 1 "START  1"
move_and_echo 3 75 FINISH
move_and_echo 1 5 "|"
move_and_echo 1 80 "|"
move_and_echo 2 5 "|"
move_and_echo 2 80 "|"
move_and_echo 4 5 "|  2"
move_and_echo 4 80 "|"
move_and_echo 5 5 "V  3"
move_and_echo 5 80 "V"

# 将前景色设置为红色.
echo -ne '\E[31m'

# 一些ASCII的艺术效果.
move_and_echo 1 8 "..@@@..@@@@@...@@@@@.@...@..@@@@..."
move_and_echo 2 8 ".@...@...@.......@...@...@.@......."
move_and_echo 3 8 ".@@@@@...@.......@...@@@@@.@@@@...."
move_and_echo 4 8 ".@...@...@.......@...@...@.@......."
move_and_echo 5 8 ".@...@...@.......@...@...@..@@@@..."
move_and_echo 1 43 "@@@@...@@@...@@@@..@@@@..@@@@."
move_and_echo 2 43 "@...@.@...@.@.....@.....@....."
move_and_echo 3 43 "@@@@..@@@@@.@.....@@@@...@@@.."
move_and_echo 4 43 "@..@..@...@.@.....@.........@."
move_and_echo 5 43 "@...@.@...@..@@@@..@@@@.@@@@.."


# 将前景色和背景色设为绿色.
echo -ne '\E[32;42m'

# 画11行绿线.
tput cup 5 0
for n in `seq 11`; do
      echo $BLANK80
done

# 将前景色设为黑色.
echo -ne '\E[30m'
tput cup 5 0

# 画栅栏.
echo "++++++++++++++++++++++++++++++++++++++++\
++++++++++++++++++++++++++++++++++++++++"

tput cup 15 0
echo "++++++++++++++++++++++++++++++++++++++++\
++++++++++++++++++++++++++++++++++++++++"

# 将前景色和背景色设回白色.
echo -ne '\E[37;47m'

# 画3条白线.
for n in `seq 3`; do
      echo $BLANK80
done

# 将前景色设为黑色.
echo -ne '\E[30m'

# 创建9个文件, 用来保存障碍物.
for n in `seq 10 7 68`; do
      touch $n
done

# 将脚本所要画的"马"设置为第一种类型.
HORSE_TYPE=2

#  为每匹"马"创建位置文件和几率文件.
#+ 在这些文件中, 保存马的当前位置,
#+ 类型和几率.
```

```
for HN in `seq 9`; do
      touch horse_${HN}_position
      touch odds_${HN}
      echo \-1 > horse_${HN}_position
      echo $HORSE_TYPE >>  horse_${HN}_position
      # 给马定义随机障碍物.
       HANDICAP=`random_1_9`
      # 检查函数random_1_9是否返回一个有效值.
      while ! echo $HANDICAP | grep [1-9] &> /dev/null; do
                 HANDICAP=`random_1_9`
      done
      # 给马定义最后一个障碍物的位置.
      LHP=`expr $HANDICAP \* 7 + 3`
      for FILE in `seq 10 7 $LHP`; do
            echo $HN >> $FILE
      done

      # 计算几率.
      case $HANDICAP in
              1) ODDS=`echo $HANDICAP \* 0.25 + 1.25 | bc`
                                echo $ODDS > odds_${HN}
              ;;
              2 | 3) ODDS=`echo $HANDICAP \* 0.40 + 1.25 | bc`
                                   echo $ODDS > odds_${HN}
              ;;
              4 | 5 | 6) ODDS=`echo $HANDICAP \* 0.55 + 1.25 | bc`
                                        echo $ODDS > odds_${HN}
              ;;
              7 | 8) ODDS=`echo $HANDICAP \* 0.75 + 1.25 | bc`
                                   echo $ODDS > odds_${HN}
              ;;
              9) ODDS=`echo $HANDICAP \* 0.90 + 1.25 | bc`
                                 echo $ODDS > odds_${HN}
      esac


done


# 打印几率.
print_odds() {
tput cup 6 0
echo -ne '\E[30;42m'
for HN in `seq 9`; do
      echo "#$HN odds->" `cat odds_${HN}`
done
}

# 在起跑线上把马画出来.
draw_horses() {
tput cup 6 0
echo -ne '\E[30;42m'
for HN in `seq 9`; do
      echo /\\$HN/\\"                                    "
done
}

print_odds

echo -ne '\E[47m'
# 等待按下回车键, 按下之后就开始比赛.
# 转义序列'\E[?25l'禁用光标.
tput cup 17 0
echo -e '\E[?25l'Press [enter] key to start the race...
read -s

#  禁用了终端的常规echo功能.
#  这么做用来避免在比赛中,
#+ 按键所导致的"花"屏.
stty -echo

# ----------------------------------------------------------
# 开始比赛.

draw_horses
echo -ne '\E[37;47m'
move_and_echo 18 1 $BLANK80
echo -ne '\E[30m'
move_and_echo 18 1 Starting...
sleep 1

# 设置终点线的列号.
WINNING_POS=74

# 定义比赛开始的时间.
```

```
START_TIME=`date +%s`

# 下面的"while"结构需要使用COL变量.
COL=0

while [ $COL -lt $WINNING_POS ]; do

           MOVE_HORSE=0

           # 检查random_1_9函数是否返回了有效值.
           while ! echo $MOVE_HORSE | grep [1-9] &> /dev/null; do
                 MOVE_HORSE=`random_1_9`
           done

           # 定义"随机抽取的马"的原来类型和位置.
           HORSE_TYPE=`cat  horse_${MOVE_HORSE}_position | tail -1`
           COL=$(expr `cat  horse_${MOVE_HORSE}_position | head -1`)

           ADD_POS=1
           # 判断当前位置是否存在障碍物.
           if seq 10 7 68 | grep -w $COL &> /dev/null; then
                 if grep -w $MOVE_HORSE $COL &> /dev/null; then
                       ADD_POS=0
                       grep -v -w  $MOVE_HORSE $COL > ${COL}_new
                       rm -f $COL
                       mv -f ${COL}_new $COL
                       else ADD_POS=1
                 fi
           else ADD_POS=1
           fi
           COL=`expr $COL + $ADD_POS`
           echo $COL >  horse_${MOVE_HORSE}_position  # 保存新位置.

          # 选择要画出来的马的类型.
           case $HORSE_TYPE in
                 1) HORSE_TYPE=2; DRAW_HORSE=draw_horse_two
                 ;;
                 2) HORSE_TYPE=1; DRAW_HORSE=draw_horse_one
           esac
           echo $HORSE_TYPE >>  horse_${MOVE_HORSE}_position # 保存当前类型.

           # 将前景色设为黑, 背景色设为绿.
           echo -ne '\E[30;42m'

           # 将光标移动到马的新位置.
           tput cup `expr $MOVE_HORSE + 5`  `cat  horse_${MOVE_HORSE}_position | head
-1`

           # 画马.
           $DRAW_HORSE
            usleep $USLEEP_ARG

            # 当所有的马都越过第15行之后, 再次打印几率.
            touch fieldline15
            if [ $COL = 15 ]; then
              echo $MOVE_HORSE >> fieldline15
            fi
            if [ `wc -l fieldline15 | cut -f1 -d " "` = 9 ]; then
                print_odds
                : > fieldline15
            fi

           # 取得领头的马.
           HIGHEST_POS=`cat *position | sort -n | tail -1`

           # 将背景色设为白.
           echo -ne '\E[47m'
           tput cup 17 0
           echo -n Current leader: `grep -w $HIGHEST_POS *position | cut -c7`"
"

done

# 定义比赛结束的时间.
FINISH_TIME=`date +%s`

# 将背景色设置为绿色, 并且开启闪烁文本的功能.
echo -ne '\E[30;42m'
echo -en '\E[5m'

# 让获胜的马闪烁.
tput cup `expr $MOVE_HORSE + 5` `cat  horse_${MOVE_HORSE}_position | head -1`
$DRAW_HORSE

# 禁用闪烁文本.
```

```
echo -en '\E[25m'

# 将前景色和背景色设置为白色.
echo -ne '\E[37;47m'
move_and_echo 18 1 $BLANK80

# 将前景色设置为黑色.
echo -ne '\E[30m'

# 让获胜的马闪烁.
tput cup 17 0
echo -e "\E[5mWINNER: $MOVE_HORSE\E[25m""  Odds: `cat odds_${MOVE_HORSE}`"\
"  Race time: `expr $FINISH_TIME - $START_TIME` secs"

# 恢复光标, 恢复原来的颜色.
echo -en "\E[?25h"
echo -en "\E[0m"

# 恢复打印功能.
stty echo

# 删除掉和赛马有关的临时文件.
rm -rf $HORSE_RACE_TMP_DIR

tput cup 19 0

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.

33.6. 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办法. 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.

33.7. 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要跟踪记录的脚本中. 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.

```
# 添加(>>)下面的代码, 到你想跟踪记录的脚本末尾.

whoami>> $SAVE_FILE     # 记录调用脚本的用户.
echo $0>> $SAVE_FILE    # 脚本名.
date>> $SAVE_FILE       # 记录日期和时间.
echo>> $SAVE_FILE       # 空行作为分隔符.

#  当然, 我们应该在~/.bashrc中定义并导出变量SAVE_FILE.
#+ (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?

```
file=data.txt
title="***This is the title line of data text file***"

echo $title | cat - $file >$file.new
# "cat -" 将stdout连接到$file.
#  最后的结果就是生成了一新文件,
#+ 并且成功的将$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.

```
# 脚本库
# ------ -------

# 注:
# 这里没有"#!".
# 也没有"真正需要执行的代码".


# 有用的变量定义

ROOT_UID=0              # root用户的$UID为0.
E_NOTROOT=101           # 非root用户的出错代码.
MAXRETVAL=255           # 函数最大的返回值(正值).
SUCCESS=0
FAILURE=-1



# Functions

Usage ()                # "Usage:"信息. (译者注: 即帮助信息)
{
  if [ -z "$1" ]        # 没有参数传递进来.
  then
    msg=filename
  else
    msg=$@
  fi

  echo "Usage: `basename $0` "$msg""
```

```
}


Check_if_root ()        # 检查运行脚本的用户是否为root.
{                       # 摘自"ex39.sh".
  if [ "$UID" -ne "$ROOT_UID" ]
  then
    echo "Must be root to run this script."
    exit $E_NOTROOT
  fi
}


CreateTempfileName ()  # 创建"唯一"的临时文件.
{                      # 摘自"ex51.sh".
  prefix=temp
  suffix=`eval date +%s`
  Tempfilename=$prefix.$suffix
}


isalpha2 ()            # 测试*整个字符串*是否都是由字母组成的.
{                      # 摘自"isalpha.sh".
  [ $# -eq 1 ] || return $FAILURE

  case $1 in
  *[!a-zA-Z]*|"") return $FAILURE;;
  *) return $SUCCESS;;
  esac                 # 感谢, S.C.
}


abs ()                              # 绝对值.
{                                   # 注意: 最大的返回值 = 255.
  E_ARGERR=-999999

  if [ -z "$1" ]                    # 需要传递参数.
  then
    return $E_ARGERR                # 返回错误.
  fi

  if [ "$1" -ge 0 ]                 # 如果是非负值,
  then                              #
    absval=$1                       # 那就是绝对值本身.
  else                              # 否则,
    let "absval = (( 0 - $1 ))"     # 改变符号.
  fi

  return $absval
}


tolower ()              #  将传递进来的参数字符串
{                       #+ 转换为小写.

  if [ -z "$1" ]        #  如果没有参数传递进来.
  then                  #+ 打印错误消息
    echo "(null)"       #+ (C风格的void指针错误消息)
    return              #+ 并且从函数中返回.
  fi

  echo "$@" | tr A-Z a-z
  # 转换所有传递进来的参数($@).

  return


# 使用命令替换, 将函数的输出赋值给变量.
# 举例:
#    oldvar="A seT of miXed-caSe LEtTerS"
#    newvar=`tolower "$oldvar"`
#    echo "$newvar"     # 一串混合大小写的字符全部转换为小写
#
# 练习: 重写这个函数,
#            将传递进来的参数全部转换为大写[容易].
}
```

- 使用特殊目的注释头来增加脚本的条理性和可读性.

```
## 表示注意.
rm -rf *.zzy    ##  "rm"命令的"-rf"选项非常的危险.
                ##+ 尤其对通配符, 就更危险.

#+ 表示继续上一行.
#  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
 3 COMMENT_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
 7 if [ $COMMENT_BLOCK ]; then
 8
 9 Comment block --
10 ================================
11 This is a comment line.
12 This is another comment line.
13 This is yet another comment line.
14 ================================
15
16 echo "This will not echo."
17
18 Comment blocks are error-free! Whee!
19
20 fi
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="$@"                # 能够接受多个参数.
  5
  6   firstchar=${string0:0:1}    # 首字母.
  7   string1=${string0:1}        # 余下的字符.
  8
  9   FirstChar=`echo "$firstchar" | tr a-z A-Z`
 10                               # 将首字母转换为大写.
 11
 12   echo "$FirstChar$string1"   # 输出到stdout.
 13
 14 }
 15
 16 newstring=`capitalize_ichar "every sentence should start with a
capital letter."`
 17 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"yawl"文字表软件包中的一部分.
18 #  http://ibiblio.org/pub/Linux/libs/yawl-0.3.2.tar.gz
19 #  http://personal.riverusers.com/~thegrendel/yawl-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 isotheral
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 #+        $command_test     $CMD                ,
```

```
      因为变量              只有在      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                        #   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.
```

- 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 $
```

33.8. 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骇客的攻击和破坏.

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骇客来说, 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.

33.9. 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，反过来看，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.

33.10. 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. Cygnus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.

34. Bash, 版本2与版本3

目录
34.1. Bash, 版本2
34.2. Bash, 版本3

34.1. 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.

```
bash$ echo $BASH_VERSION
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

```
#!/bin/bash

# 字符串扩展.
# Bash版本2中引入的特性.

#  $'xxx'格式的字符串
#+ 具备解释里面标准转义字符的能力.

echo $'Ringing bell 3 times \a \a \a'
     # 可能在某些终端中, 只会响一次铃.
echo $'Three form feeds \f \f \f'
echo $'10 newlines \n\n\n\n\n\n\n\n\n\n'
echo $'\102\141\163\150'    # Bash
                            # 8进制的等价字符.

exit 0
```

例子 34-2. 间接变量引用 - 新方法

```
#!/bin/bash

# 间接变量引用.
# 这种方法比较像C++中的引用特性.


a=letter_of_alphabet
letter_of_alphabet=z

echo "a = $a"           # 直接引用.

echo "Now a = ${!a}"    # 间接引用.
# ${!variable}表示法比老式的"eval var1=\$$var2"表示法高级的多.

echo

t=table_cell_3
table_cell_3=24
echo "t = ${!t}"                        # t = 24
table_cell_3=387
echo "Value of t changed to ${!t}"    # 387

#  在引用数组成员或者引用表的时候, 这种方法非常有用,
#+ 还可以用来模拟多维数组.
#  如果有能够索引的选项(类似于指针的算术运算)
#+ 就更好了. 可惜.

exit 0
```

例子 34-3. 使用间接变量引用的简单数据库应用

```
#!/bin/bash
# resistor-inventory.sh
# 使用间接变量引用的简单数据库应用.

# ============================================================== #
# 数据

B1723_value=470                                          # 欧姆
B1723_powerdissip=.25                                    # 瓦特
B1723_colorcode="yellow-violet-brown"                    # 颜色
B1723_loc=173                                            # 位置
B1723_inventory=78                                       # 数量

B1724_value=1000
B1724_powerdissip=.25
B1724_colorcode="brown-black-red"
B1724_loc=24N
B1724_inventory=243

B1725_value=10000
B1725_powerdissip=.25
B1725_colorcode="brown-black-orange"
B1725_loc=24N
B1725_inventory=89

# ============================================================== #


echo

PS3='Enter catalog number: '

echo

select catalog_number in "B1723" "B1724" "B1725"
do
  Inv=${catalog_number}_inventory
  Val=${catalog_number}_value
  Pdissip=${catalog_number}_powerdissip
  Loc=${catalog_number}_loc
  Ccode=${catalog_number}_colorcode

  echo
  echo "Catalog number $catalog_number:"
  echo "There are ${!Inv} of [${!Val} ohm / ${!Pdissip} watt] resistors in
stock."
  echo "These are located in bin # ${!Loc}."
  echo "Their color code is \"${!Ccode}\"."

  break
done

echo; echo

# 练习:
# -----
# 1) 重写脚本, 使其从外部文件读取数据.
# 2) 重写脚本,
#+   用数组来代替间接变量引用,
#    因为使用数组更简单, 更易懂.


# 注:
# ---
#  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.
#+ 因为它太依赖于工作环境和机器的运算能力.
#  更好的办法还是使用支持数据结构的本地语言,
#+ 比如C++或者Java(或者甚至可以是Perl).

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

```
#!/bin/bash

# 纸牌:
# 处理4人打牌.

UNPICKED=0
PICKED=1

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 $Card
71 # 使用整洁的列形式来打印每张牌.
72 }
73
74 seed_random ()  # 种子随机数产生器.
75 {               # 如果不这么做, 会发生什么?
76 seed=`eval date +%s`
77 let "seed %= 32766"
78 RANDOM=$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:
140 # 简化并理顺脚本逻辑.
```

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.

34.2. 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.

```
#!/bin/bash

for i in {1..10}
#  比下面的方式更简单，更直接
#+ for i in $(seq 10)
do
  echo -n "$i "
done

echo

# 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.

```
#!/bin/bash

Array=(element-zero element-one element-two element-three)

echo ${Array[0]}    # 元素0
                    # 数组的第一个元素.

echo ${!Array[@]}   # 0 1 2 3
                    # 数组的全部索引.

for i in ${!Array[@]}
do
  echo ${Array[i]} # 元素0
                   # 元素1
                   # 元素2
                   # 元素3
                   #
                   # 数组的全部元素.
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.)

```
#!/bin/bash

variable="This is a fine mess."

echo "$variable"

if [[ "$variable" =~ "T*fin*es*" ]]
# 在[[ 双中括号 ]]中使用=~操作符进行正则匹配.
then
  echo "match found"
      # match found
fi
```

或者，更有用的用法:

```
#!/bin/bash

input=$1


if [[ "$input" =~ "[1-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]"
]]
# NNN-NN-NNNN
# 每个N都是一个数字.
# 但是，第一个数字不能为0.
then
```

```
11   echo "Social Security number."
12   # 处理SSN.
13 else
14   echo "Not a Social Security number!"
15   # 或者, 要求正确的输入.
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
```

35. 后记

目录

35.1. 作者后记
35.2. 关于作者
35.3. 译者后记
35.3.1. 杨春敏
35.3.2. 黄毅
35.4. 在哪里可以获得帮助
35.5. 用来制作这本书的工具
35.5.1. 硬件
35.5.2. 软件与排版软件
35.6. 致谢
35.7. 译者致谢

35.1. 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.

35.2. 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yawl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.

35.3. 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！

35.4. 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.

35.5. 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<g>?

35.5.2. 软件与排版软件

i. Bram Moolenaar的强大的SGML软件, vim文本编辑器.

ii. OpenJade, 使用DSSSL翻译引擎, 来将SGML文档转换为其他格式的工具.

iii. Norman Walsh的DSSSL样式单.

iv. *DocBook, The Definitive Guide*(译者注: 这本书被亲切称为TDG), 这本书由Norman Walsh和Leonard 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TLDP的DSSSL基础上做了最大努力的修改.

35.6. 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鰊 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.

35.7. 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.

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 Burtch, *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, 1-156592-225-5.

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.

*

Aeleen 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 scrips.

"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.

Appendix A. 捐献的脚本

这些脚本展示了一些有趣的shell编程技术，但是它们并不适合放入本文档的文本讲解中. 不过它们还是非常有用，运行和分析它们都是很有意思的事.

译者：这里留给那些有能力而且有多余时间的读者来详读，个人认为翻译这些注释有点画蛇添足.

例子 A-1. mailformat: 格式化一个e-mail消息

```
#!/bin/bash
# mail-format.sh (ver. 1.1): Format e-mail messages.

# Gets rid of carets, tabs, and also folds excessively long lines.

# =================================================================
#                 Standard Check for Script Argument(s)
ARGS=1
E_BADARGS=65
E_NOFILE=66

if [ $# -ne $ARGS ]  # Correct number of arguments passed to script?
then
  echo "Usage: `basename $0` filename"
  exit $E_BADARGS
fi

if [ -f "$1" ]       # Check if file exists.
then
    file_name=$1
else
    echo "File \"$1\" does not exist."
    exit $E_NOFILE
fi
# =================================================================

MAXWIDTH=70          # Width to fold excessively long lines to.

# --------------------------------
# A variable can hold a sed script.
sedscript='s/^>//
s/^  *>//
s/^  *//
s/          *//'
# --------------------------------

#  Delete carets and tabs at beginning of lines,
#+ then fold lines to $MAXWIDTH characters.
sed "$sedscript" $1 | fold -s --width=$MAXWIDTH
                      #  -s option to "fold"
                      #+ breaks lines at whitespace, if possible.


#  This script was inspired by an article in a well-known trade journal
#+ extolling a 164K MS Windows utility with similar functionality.
#
#  An nice set of text processing utilities and an efficient
#+ scripting language provide an alternative to bloated executables.

exit 0
```

例子 A-2. rn: 一个非常简单的文件重命名工具

这个脚本是例子 12-19的一个修改版.

```
#! /bin/bash
#
# Very simpleminded filename "rename" utility (based on "lowercase.sh").
#
#  The "ren" utility, by Vladimir Lanin (lanin@csd2.nyu.edu),
#+ does a much better job of this.


ARGS=2
E_BADARGS=65
ONE=1                     # For getting singular/plural right (see below).

if [ $# -ne "$ARGS" ]
then
  echo "Usage: `basename $0` old-pattern new-pattern"
  # As in "rn gif jpg", which renames all gif files in working directory to jpg.
  exit $E_BADARGS
fi

number=0                  # Keeps track of how many files actually renamed.

```

```
22
23 for filename in *$1*      #Traverse all matching files in directory.
24 do
25    if [ -f "$filename" ]  # If finds match...
26    then
27      fname=`basename $filename`            # Strip off path.
28      n=`echo $fname | sed -e "s/$1/$2/"`   # Substitute new for old in filename.
29      mv $fname $n                          # Rename.
30      let "number += 1"
31    fi
32 done
33
34 if [ "$number" -eq "$ONE" ]                # For correct grammar.
35 then
36  echo "$number file renamed."
37 else
38  echo "$number files renamed."
39 fi
40
41 exit 0
42
43
44 # Exercises:
45 # ---------
46 # What type of files will this not work on?
47 # How can this be fixed?
48 #
49 #  Rewrite this script to process all the files in a directory
50 #+ containing spaces in their names, and to rename them,
51 #+ substituting an underscore for each space.
```

例子 A-3. blank-rename: 重命名包含空白的文件名

这是上一个脚本的简化版.

```
 1 #! /bin/bash
 2 # blank-rename.sh
 3 #
 4 # Substitutes underscores for blanks in all the filenames in a directory.
 5
 6 ONE=1                      # For getting singular/plural right (see below).
 7 number=0                   # Keeps track of how many files actually renamed.
 8 FOUND=0                    # Successful return value.
 9
10 for filename in *          #Traverse all files in directory.
11 do
12      echo "$filename" | grep -q " "         #  Check whether filename
13      if [ $? -eq $FOUND ]                   #+ contains space(s).
14      then
15        fname=$filename                      # Strip off path.
16        n=`echo $fname | sed -e "s/ /_/g"`   # Substitute underscore for blank.
17        mv "$fname" "$n"                     # Do the actual renaming.
18        let "number += 1"
19      fi
20 done
21
22 if [ "$number" -eq "$ONE" ]                # For correct grammar.
23 then
24  echo "$number file renamed."
25 else
26  echo "$number files renamed."
27 fi
28
29 exit 0
```

例子 A-4. encryptedpw: 使用一个本地加密口令, 上传到一个ftp服务器.

```
 1 #!/bin/bash
 2
 3 # Example "ex72.sh" modified to use encrypted password.
 4
 5 #  Note that this is still rather insecure,
 6 #+ since the decrypted password is sent in the clear.
 7 #  Use something like "ssh" if this is a concern.
 8
 9 E_BADARGS=65
10
11 if [ -z "$1" ]
12 then
13   echo "Usage: `basename $0` filename"
14   exit $E_BADARGS
15 fi
16
17 Username=bozo           # Change to suit.
18 pword=/home/bozo/secret/password_encrypted.file
19 # File containing encrypted password.
20
21 Filename=`basename $1`  # Strips pathname out of file name.
22
23 Server="XXX"
24 Directory="YYY"         # Change above to actual server name & directory.
25
26
27 Password=`cruft <$pword`          # Decrypt password.
```

```
#  Uses the author's own "cruft" file encryption package,
#+ based on the classic "onetime pad" algorithm,
#+ and obtainable from:
#+ Primary-site:   ftp://ibiblio.org/pub/Linux/utils/file
#+                 cruft-0.2.tar.gz [16k]


ftp -n $Server <<End-Of-Session
user $Username $Password
binary
bell
cd $Directory
put $Filename
bye
End-Of-Session
# -n option to "ftp" disables auto-logon.
# Note that "bell" rings 'bell' after each file transfer.

exit 0
```

例子 A-5. copy-cd: 拷贝一个数据CD

```
#!/bin/bash
# copy-cd.sh: copying a data CD

CDROM=/dev/cdrom                            # CD ROM device
OF=/home/bozo/projects/cdimage.iso          # output file
#       /xxxx/xxxxxxx/                      Change to suit your system.
BLOCKSIZE=2048
SPEED=2                                     # May use higher speed if supported.
DEVICE=cdrom
# DEVICE="0,0"    on older versions of cdrecord.

echo; echo "Insert source CD, but do *not* mount it."
echo "Press ENTER when ready. "
read ready                                  # Wait for input, $ready not used.

echo; echo "Copying the source CD to $OF."
echo "This may take a while. Please be patient."

dd if=$CDROM of=$OF bs=$BLOCKSIZE           # Raw device copy.


echo; echo "Remove data CD."
echo "Insert blank CDR."
echo "Press ENTER when ready. "
read ready                                  # Wait for input, $ready not used.

echo "Copying $OF to CDR."

cdrecord -v -isosize speed=$SPEED dev=$DEVICE $OF
# Uses Joerg Schilling's "cdrecord" package (see its docs).
# http://www.fokus.gmd.de/nthp/employees/schilling/cdrecord.html


echo; echo "Done copying $OF to CDR on device $CDROM."

echo "Do you want to erase the image file (y/n)? "  # Probably a huge file.
read answer

case "$answer" in
[yY]) rm -f $OF
      echo "$OF erased."
      ;;
*)    echo "$OF not erased.";;
esac

echo

# Exercise:
# Change the above "case" statement to also accept "yes" and "Yes" as input.

exit 0
```

例子 A-6. Collatz序列

```
#!/bin/bash
# collatz.sh

#  The notorious "hailstone" or Collatz series.
#  --------------------------------------------
#  1) Get the integer "seed" from the command line.
#  2) NUMBER <--- seed
#  3) Print NUMBER.
#  4)  If NUMBER is even, divide by 2, or
#  5)+ if odd, multiply by 3 and add 1.
#  6) NUMBER <--- result
#  7) Loop back to step 3 (for specified number of iterations).
#
#  The theory is that every sequence,
#+ no matter how large the initial value,
#+ eventually settles down to repeating "4,2,1..." cycles,
#+ even after fluctuating through a wide range of values.
#
#  This is an instance of an "iterate",
```

```
#+ an operation that feeds its output back into the input.
#  Sometimes the result is a "chaotic" series.


MAX_ITERATIONS=200
# For large seed numbers (>32000), increase MAX_ITERATIONS.

h=${1:-$$}                      #  Seed
                                #  Use $PID as seed,
                                #+ if not specified as command-line arg.

echo
echo "C($h) --- $MAX_ITERATIONS Iterations"
echo

for ((i=1; i<=MAX_ITERATIONS; i++))
do

echo -n "$h "
#          ^^^^^
#           tab

  let "remainder = h % 2"
  if [ "$remainder" -eq 0 ]   # Even?
  then
    let "h /= 2"              # Divide by 2.
  else
    let "h = h*3 + 1"         # Multiply by 3 and add 1.
  fi


COLUMNS=10                    # Output 10 values per line.
let "line_break = i % $COLUMNS"
if [ "$line_break" -eq 0 ]
then
  echo
fi

done

echo

#  For more information on this mathematical function,
#+ see "Computers, Pattern, Chaos, and Beauty", by Pickover, p. 185 ff.,
#+ as listed in the bibliography.

exit 0
```

例子 A-7. days-between: 计算两个日期之间天数差

```
#!/bin/bash
# days-between.sh:    Number of days between two dates.
# Usage: ./days-between.sh [M]M/[D]D/YYYY [M]M/[D]D/YYYY
#
# Note: Script modified to account for changes in Bash 2.05b
#+      that closed the loophole permitting large negative
#+      integer return values.

ARGS=2                  # Two command line parameters expected.
E_PARAM_ERR=65          # Param error.

REFYR=1600              # Reference year.
CENTURY=100
DIY=365
ADJ_DIY=367             # Adjusted for leap year + fraction.
MIY=12
DIM=31
LEAPCYCLE=4

MAXRETVAL=255           #  Largest permissable
                        #+ positive return value from a function.

diff=                   # Declare global variable for date difference.
value=                  # Declare global variable for absolute value.
day=                    # Declare globals for day, month, year.
month=
year=


Param_Error ()          # Command line parameters wrong.
{
  echo "Usage: `basename $0` [M]M/[D]D/YYYY [M]M/[D]D/YYYY"
  echo "       (date must be after 1/3/1600)"
  exit $E_PARAM_ERR
}


Parse_Date ()                 # Parse date from command line params.
{
  month=${1%%/**}
  dm=${1%/**}                 # Day and month.
  day=${dm#*/}
  let "year = `basename $1`"  # Not a filename, but works just the same.
}


check_date ()                 # Checks for invalid date(s) passed.
{
```

```
 49   [ "$day" -gt "$DIM" ] || [ "$month" -gt "$MIY" ] || [ "$year" -lt "$REFYR" ] && Param_Error
 50   # Exit script on bad value(s).
 51   # Uses "or-list / and-list".
 52   #
 53   # Exercise: Implement more rigorous date checking.
 54 }
 55
 56
 57 strip_leading_zero () #  Better to strip possible leading zero(s)
 58 {                      #+ from day and/or month
 59   return ${1#0}       #+ since otherwise Bash will interpret them
 60 }                      #+ as octal values (POSIX.2, sect 2.9.2.1).
 61
 62
 63 day_index ()          # Gauss' Formula:
 64 {                     # Days from Jan. 3, 1600 to date passed as param.
 65
 66   day=$1
 67   month=$2
 68   year=$3
 69
 70   let "month = $month - 2"
 71   if [ "$month" -le 0 ]
 72   then
 73     let "month += 12"
 74     let "year -= 1"
 75   fi
 76
 77   let "year -= $REFYR"
 78   let "indexyr = $year / $CENTURY"
 79
 80
 81   let "Days = $DIY*$year + $year/$LEAPCYCLE - $indexyr + $indexyr/$LEAPCYCLE +
$ADJ_DIY*$month/$MIY + $day - $DIM"
 82   #  For an in-depth explanation of this algorithm, see
 83   #+ http://home.t-online.de/home/berndt.schwerdtfeger/cal.htm
 84
 85
 86   echo $Days
 87
 88 }
 89
 90
 91 calculate_difference ()            # Difference between to day indices.
 92 {
 93   let "diff = $1 - $2"             # Global variable.
 94 }
 95
 96
 97 abs ()                             #  Absolute value
 98 {                                  #  Uses global "value" variable.
 99   if [ "$1" -lt 0 ]                #  If negative
100   then                             #+ then
101     let "value = 0 - $1"           #+ change sign,
102   else                             #+ else
103     let "value = $1"               #+ leave it alone.
104   fi
105 }
106
107
108
109 if [ $# -ne "$ARGS" ]              # Require two command line params.
110 then
111   Param_Error
112 fi
113
114 Parse_Date $1
115 check_date $day $month $year       #  See if valid date.
116
117 strip_leading_zero $day            #  Remove any leading zeroes
118 day=$?                             #+ on day and/or month.
119 strip_leading_zero $month
120 month=$?
121
122 let "date1 = `day_index $day $month $year`"
123
124
125 Parse_Date $2
126 check_date $day $month $year
127
128 strip_leading_zero $day
129 day=$?
130 strip_leading_zero $month
131 month=$?
132
133 date2=$(day_index $day $month $year) # Command substitution.
134
135
136 calculate_difference $date1 $date2
137
138 abs $diff                            # Make sure it's positive.
139 diff=$value
140
141 echo $diff
142
143 exit 0
144 #  Compare this script with
145 #+ the implementation of Gauss' Formula in a C program at:
146 #+    http://buschencrew.hypermart.net/software/datedif
```

例子 A-8. 构造一个"字典"

```
#!/bin/bash
# makedict.sh  [make dictionary]

# Modification of /usr/sbin/mkdict script.
# Original script copyright 1993, by Alec Muffett.
#
#  This modified script included in this document in a manner
#+ consistent with the "LICENSE" document of the "Crack" package
#+ that the original script is a part of.

#  This script processes text files to produce a sorted list
#+ of words found in the files.
#  This may be useful for compiling dictionaries
#+ and for lexicographic research.


E_BADARGS=65

if [ ! -r "$1" ]                     #  Need at least one
then                                 #+ valid file argument.
  echo "Usage: $0 files-to-process"
  exit $E_BADARGS
fi


# SORT="sort"                        #  No longer necessary to define options
                                     #+ to sort. Changed from original script.

cat $* |                             # Contents of specified files to stdout.
        tr A-Z a-z |                 # Convert to lowercase.
        tr ' ' '\012' |              # New: change spaces to newlines.
#       tr -cd '\012[a-z][0-9]' |    #  Get rid of everything non-alphanumeric
                                     #+ (original script).
        tr -c '\012a-z'  '\012' |    #  Rather than deleting
                                     #+ now change non-alpha to newlines.
        sort |                       # $SORT options unnecessary now.
        uniq |                       # Remove duplicates.
        grep -v '^#' |               # Delete lines beginning with a hashmark.
        grep -v '^$'                 # Delete blank lines.

exit 0
```

例子 A-9. Soundex转换

```
#!/bin/bash
# soundex.sh: Calculate "soundex" code for names

# ======================================================
#        Soundex script
#             by
#         Mendel Cooper
#     thegrendel@theriver.com
#       23 January, 2002
#
#   Placed in the Public Domain.
#
# A slightly different version of this script appeared in
#+ Ed Schaefer's July, 2002 "Shell Corner" column
#+ in "Unix Review" on-line,
#+ http://www.unixreview.com/documents/uni1026336632258/
# ======================================================


ARGCOUNT=1                      # Need name as argument.
E_WRONGARGS=70

if [ $# -ne "$ARGCOUNT" ]
then
  echo "Usage: `basename $0` name"
  exit $E_WRONGARGS
fi


assign_value ()                 #  Assigns numerical value
{                               #+ to letters of name.

  val1=bfpv                     # 'b,f,p,v' = 1
  val2=cgjkqsxz                 # 'c,g,j,k,q,s,x,z' = 2
  val3=dt                       #  etc.
  val4=l
  val5=mn
  val6=r

# Exceptionally clever use of 'tr' follows.
# Try to figure out what is going on here.

value=$( echo "$1" \
| tr -d wh \
| tr $val1 1 | tr $val2 2 | tr $val3 3 \
| tr $val4 4 | tr $val5 5 | tr $val6 6 \
| tr -s 123456 \
| tr -d aeiouy )

# Assign letter values.
# Remove duplicate numbers, except when separated by vowels.
```

```
# Ignore vowels, except as separators, so delete them last.
# Ignore 'w' and 'h', even as separators, so delete them first.
#
# The above command substitution lays more pipe than a plumber <g>.

}


input_name="$1"
echo
echo "Name = $input_name"


# Change all characters of name input to lowercase.
# ------------------------------------------------
name=$( echo $input_name | tr A-Z a-z )
# ------------------------------------------------
# Just in case argument to script is mixed case.


# Prefix of soundex code: first letter of name.
# ----------------------------------------


char_pos=0                      # Initialize character position.
prefix0=${name:$char_pos:1}
prefix=`echo $prefix0 | tr a-z A-Z`
                                # Uppercase 1st letter of soundex.

let "char_pos += 1"             # Bump character position to 2nd letter of name.
name1=${name:$char_pos}


# ++++++++++++++++++++++++++ Exception Patch ++++++++++++++++++++++++++++++++
#  Now, we run both the input name and the name shifted one char to the right
#+ through the value-assigning function.
#  If we get the same value out, that means that the first two characters
#+ of the name have the same value assigned, and that one should cancel.
#  However, we also need to test whether the first letter of the name
#+ is a vowel or 'w' or 'h', because otherwise this would bollix things up.

char1=`echo $prefix | tr A-Z a-z`    # First letter of name, lowercased.

assign_value $name
s1=$value
assign_value $name1
s2=$value
assign_value $char1
s3=$value
s3=9$s3                               #  If first letter of name is a vowel
                                      #+ or 'w' or 'h',
                                      #+ then its "value" will be null (unset).
                                    #+ Therefore, set it to 9, an otherwise
                                    #+ unused value, which can be tested for.


if [[ "$s1" -ne "$s2" || "$s3" -eq 9 ]]
then
  suffix=$s2
else
  suffix=${s2:$char_pos}
fi
# ++++++++++++++++++++++ end Exception Patch ++++++++++++++++++++++++++++++++


padding=000                     # Use at most 3 zeroes to pad.


soun=$prefix$suffix$padding    # Pad with zeroes.

MAXLEN=4                        # Truncate to maximum of 4 chars.
soundex=${soun:0:$MAXLEN}

echo "Soundex = $soundex"

echo

#  The soundex code is a method of indexing and classifying names
#+ by grouping together the ones that sound alike.
#  The soundex code for a given name is the first letter of the name,
#+ followed by a calculated three-number code.
#  Similar sounding names should have almost the same soundex codes.

#   Examples:
#   Smith and Smythe both have a "S-530" soundex.
#   Harrison = H-625
#   Hargison = H-622
#   Harriman = H-655

#  This works out fairly well in practice, but there are numerous anomalies.
#
#
#  The U.S. Census and certain other governmental agencies use soundex,
#  as do genealogical researchers.
#
#  For more information,
#+ see the "National Archives and Records Administration home page",
#+ http://www.nara.gov/genealogy/soundex/soundex.html


# Exercise:
```

```
154 # --------
155 # Simplify the "Exception Patch" section of this script.
156
157 exit 0
```

例子 A-10. "Game of Life"

```
  1 #!/bin/bash
  2 # life.sh: "Life in the Slow Lane"
  3 # Version 2: Patched by Daniel Albers
  4 #+           to allow non-square grids as input.
  5
  6 # ##########################################################################
  7 # This is the Bash script version of John Conway's "Game of Life".      #
  8 # "Life" is a simple implementation of cellular automata.               #
  9 # --------------------------------------------------------------------- #
 10 # On a rectangular grid, let each "cell" be either "living" or "dead".  #
 11 # Designate a living cell with a dot, and a dead one with a blank space.#
 12 #  Begin with an arbitrarily drawn dot-and-blank grid,                  #
 13 #+ and let this be the starting generation, "generation 0".            #
 14 # Determine each successive generation by the following rules:          #
 15 # 1) Each cell has 8 neighbors, the adjoining cells                     #
 16 #+   left, right, top, bottom, and the 4 diagonals.                    #
 17 #                        123                                           #
 18 #                        4*5                                           #
 19 #                        678                                           #
 20 #                                                                      #
 21 # 2) A living cell with either 2 or 3 living neighbors remains alive.  #
 22 # 3) A dead cell with 3 living neighbors becomes alive (a "birth").    #
 23 SURVIVE=2                                                             #
 24 BIRTH=3                                                               #
 25 # 4) All other cases result in a dead cell for the next generation.    #
 26 # ##########################################################################
 27
 28
 29 startfile=gen0   # Read the starting generation from the file "gen0".
 30                  # Default, if no other file specified when invoking script.
 31                  #
 32 if [ -n "$1" ]   # Specify another "generation 0" file.
 33 then
 34     startfile="$1"
 35 fi
 36
 37 ##############################################
 38 #  Abort script if "startfile" not specified
 39 #+ AND
 40 #+ "gen0" not present.
 41
 42 E_NOSTARTFILE=68
 43
 44 if [ ! -e "$startfile" ]
 45 then
 46   echo "Startfile \""$startfile"\" missing!"
 47   exit $E_NOSTARTFILE
 48 fi
 49 ##############################################
 50
 51
 52 ALIVE1=.
 53 DEAD1=_
 54                  # Represent living and "dead" cells in the start-up file.
 55
 56 #  ------------------------------------------------------------ #
 57 #  This script uses a 10 x 10 grid (may be increased,
 58 #+ but a large grid will will cause very slow execution).
 59 ROWS=10
 60 COLS=10
 61 #  Change above two variables to match grid size, if necessary.
 62 #  ------------------------------------------------------------ #
 63
 64 GENERATIONS=10          #  How many generations to cycle through.
 65                         #  Adjust this upwards,
 66                         #+ if you have time on your hands.
 67
 68 NONE_ALIVE=80           #  Exit status on premature bailout,
 69                         #+ if no cells left alive.
 70 TRUE=0
 71 FALSE=1
 72 ALIVE=0
 73 DEAD=1
 74
 75 avar=                  # Global; holds current generation.
 76 generation=0           # Initialize generation count.
 77
 78 # ==================================================================
 79
 80
 81 let "cells = $ROWS * $COLS"
 82                         # How many cells.
 83
 84 declare -a initial      # Arrays containing "cells".
 85 declare -a current
 86
 87 display ()
 88 {
 89
 90 alive=0                 # How many cells "alive" at any given time.
 91                         # Initially zero.
```

```
 92
 93 declare -a arr
 94 arr=( `echo "$1"` )     # Convert passed arg to array.
 95
 96 element_count=${#arr[*]}
 97
 98 local i
 99 local rowcheck
100
101 for ((i=0; i<$element_count; i++))
102 do
103
104   # Insert newline at end of each row.
105   let "rowcheck = $i % COLS"
106   if [ "$rowcheck" -eq 0 ]
107   then
108     echo                # Newline.
109     echo -n "      "    # Indent.
110   fi
111
112   cell=${arr[i]}
113
114   if [ "$cell" = . ]
115   then
116     let "alive += 1"
117   fi
118
119   echo -n "$cell" | sed -e 's/_/ /g'
120   # Print out array and change underscores to spaces.
121 done
122
123 return
124
125 }
126
127 IsValid ()                            # Test whether cell coordinate valid.
128 {
129
130   if [ -z "$1"  -o -z "$2" ]          # Mandatory arguments missing?
131   then
132     return $FALSE
133   fi
134
135 local row
136 local lower_limit=0                   # Disallow negative coordinate.
137 local upper_limit
138 local left
139 local right
140
141 let "upper_limit = $ROWS * $COLS - 1" # Total number of cells.
142
143
144 if [ "$1" -lt "$lower_limit" -o "$1" -gt "$upper_limit" ]
145 then
146   return $FALSE                       # Out of array bounds.
147 fi
148
149 row=$2
150 let "left = $row * $COLS"             # Left limit.
151 let "right = $left + $COLS - 1"       # Right limit.
152
153 if [ "$1" -lt "$left" -o "$1" -gt "$right" ]
154 then
155   return $FALSE                       # Beyond row boundary.
156 fi
157
158 return $TRUE                          # Valid coordinate.
159
160 }
161
162
163 IsAlive ()              # Test whether cell is alive.
164                         # Takes array, cell number, state of cell as arguments.
165 {
166   GetCount "$1" $2      # Get alive cell count in neighborhood.
167   local nhbd=$?
168
169
170   if [ "$nhbd" -eq "$BIRTH" ]  # Alive in any case.
171   then
172     return $ALIVE
173   fi
174
175   if [ "$3" = "." -a "$nhbd" -eq "$SURVIVE" ]
176   then                  # Alive only if previously alive.
177     return $ALIVE
178   fi
179
180   return $DEAD          # Default.
181
182 }
183
184
185 GetCount ()             # Count live cells in passed cell's neighborhood.
186                         # Two arguments needed:
187                      # $1) variable holding array
188                      # $2) cell number
189 {
190   local cell_number=$2
191   local array
192   local top
193   local center
```

```
  local bottom
  local r
  local row
  local i
  local t_top
  local t_cen
  local t_bot
  local count=0
  local ROW_NHBD=3

  array=( `echo "$1"` )

  let "top = $cell_number - $COLS - 1"    # Set up cell neighborhood.
  let "center = $cell_number - 1"
  let "bottom = $cell_number + $COLS - 1"
  let "r = $cell_number / $COLS"

  for ((i=0; i<$ROW_NHBD; i++))           # Traverse from left to right.
  do
    let "t_top = $top + $i"
    let "t_cen = $center + $i"
    let "t_bot = $bottom + $i"


    let "row = $r"                        # Count center row of neighborhood.
    IsValid $t_cen $row                   # Valid cell position?
    if [ $? -eq "$TRUE" ]
    then
      if [ ${array[$t_cen]} = "$ALIVE1" ] # Is it alive?
      then                                # Yes?
        let "count += 1"                  # Increment count.
      fi
    fi

    let "row = $r - 1"                    # Count top row.
    IsValid $t_top $row
    if [ $? -eq "$TRUE" ]
    then
      if [ ${array[$t_top]} = "$ALIVE1" ]
      then
        let "count += 1"
      fi
    fi

    let "row = $r + 1"                    # Count bottom row.
    IsValid $t_bot $row
    if [ $? -eq "$TRUE" ]
    then
      if [ ${array[$t_bot]} = "$ALIVE1" ]
      then
        let "count += 1"
      fi
    fi

  done


  if [ ${array[$cell_number]} = "$ALIVE1" ]
  then
    let "count -= 1"        #  Make sure value of tested cell itself
  fi                        #+ is not counted.


  return $count

}

next_gen ()               # Update generation array.
{

local array
local i=0

array=( `echo "$1"` )     # Convert passed arg to array.

while [ "$i" -lt "$cells" ]
do
  IsAlive "$1" $i ${array[$i]}   # Is cell alive?
  if [ $? -eq "$ALIVE" ]
  then                           #  If alive, then
    array[$i]=.                  #+ represent the cell as a period.
  else
    array[$i]="_"                #  Otherwise underscore
   fi                            #+ (which will later be converted to space).
  let "i += 1"
done


# let "generation += 1"   # Increment generation count.
# Why was the above line commented out?


# Set variable to pass as parameter to "display" function.
avar=`echo ${array[@]}`   # Convert array back to string variable.
display "$avar"           # Display it.
echo; echo
echo "Generation $generation  -  $alive alive"

if [ "$alive" -eq 0 ]
then
  echo
  echo "Premature exit: no more cells alive!"
```

```
296   exit $NONE_ALIVE        #  No point in continuing
297 fi                        #+ if no live cells.
298
299 }
300
301
302 # ==========================================================
303
304 # main ()
305
306 # Load initial array with contents of startup file.
307 initial=( `cat "$startfile" | sed -e '/#/d' | tr -d '\n' |\
308 sed -e 's/\./\. /g' -e 's/_/_ /g'` )
309 # Delete lines containing '#' comment character.
310 # Remove linefeeds and insert space between elements.
311
312 clear          # Clear screen.
313
314 echo #         Title
315 echo "======================"
316 echo "    $GENERATIONS generations"
317 echo "        of"
318 echo "\"Life in the Slow Lane\""
319 echo "======================"
320
321
322 # -------- Display first generation. --------
323 Gen0=`echo ${initial[@]}`
324 display "$Gen0"           # Display only.
325 echo; echo
326 echo "Generation $generation  -  $alive alive"
327 # ----------------------------------------
328
329
330 let "generation += 1"     # Increment generation count.
331 echo
332
333 # ------- Display second generation. -------
334 Cur=`echo ${initial[@]}`
335 next_gen "$Cur"           # Update & display.
336 # ------------------------------------------
337
338 let "generation += 1"     # Increment generation count.
339
340 # ------ Main loop for displaying subsequent generations ------
341 while [ "$generation" -le "$GENERATIONS" ]
342 do
343   Cur="$avar"
344   next_gen "$Cur"
345   let "generation += 1"
346 done
347 # ============================================================
348
349 echo
350
351 exit 0   # END
352
353
354
355 # The grid in this script has a "boundary problem."
356 # The the top, bottom, and sides border on a void of dead cells.
357 # Exercise: Change the script to have the grid wrap around,
358 # +         so that the left and right sides will "touch,"
359 # +         as will the top and bottom.
360 #
361 # Exercise: Create a new "gen0" file to seed this script.
362 #           Use a 12 x 16 grid, instead of the original 10 x 10 one.
363 #           Make the necessary changes to the script,
364 #+          so it will run with the altered file.
365 #
366 # Exercise: Modify this script so that it can determine the grid size
367 #+          from the "gen0" file, and set any variables necessary
368 #+          for the script to run.
369 #           This would make unnecessary any changes to variables
370 #+          in the script for an altered grid size.
```

例子 A-11. "Game of Life"的数据文件

```
  1 # This is an example "generation 0" start-up file for "life.sh".
  2 # ---------------------------------------------------------------
  3 #  The "gen0" file is a 10 x 10 grid using a period (.) for live cells,
  4 #+ and an underscore (_) for dead ones. We cannot simply use spaces
  5 #+ for dead cells in this file because of a peculiarity in Bash arrays.
  6 #  [Exercise for the reader: explain this.]
  7 #
  8 # Lines beginning with a '#' are comments, and the script ignores them.
  9 __.__..___
 10 ___._._____
 11 _____.___..
 12 _._______.
 13 ____._____
 14 ..__...___
 15 _____._____
 16 ___...____
 17 __.._._.___
 18 _..___..__
```

+++

下面的两个脚本是由多伦多大学的Mark Moraes编写的. 请参考附件文件"Moraes-COPYRIGHT", 详细的指明了授权与约定.

例子 A-12. behead: 去掉信件与新消息的头

```
#! /bin/sh
# Strips off the header from a mail/News message i.e. till the first
# empty line
# Mark Moraes, University of Toronto

# ==> These comments added by author of this document.

if [ $# -eq 0 ]; then
# ==> If no command line args present, then works on file redirected to stdin.
    sed -e '1,/^$/d' -e '/^[        ]*$/d'
    # --> Delete empty lines and all lines until
    # --> first one beginning with white space.
else
# ==> If command line args present, then work on files named.
    for i do
            sed -e '1,/^$/d' -e '/^[        ]*$/d' $i
            # --> Ditto, as above.
    done
fi

# ==> Exercise: Add error checking and other options.
# ==>
# ==> Note that the small sed script repeats, except for the arg passed.
# ==> Does it make sense to embed it in a function? Why or why not?
```

例子 A-13. ftpget: 通过ftp下载文件

```
#! /bin/sh
# $Id: ftpget,v 1.2 91/05/07 21:15:43 moraes Exp $
# Script to perform batch anonymous ftp. Essentially converts a list of
# of command line arguments into input to ftp.
# ==> This script is nothing but a shell wrapper around "ftp" . . .
# Simple, and quick - written as a companion to ftplist
# -h specifies the remote host (default prep.ai.mit.edu)
# -d specifies the remote directory to cd to - you can provide a sequence
# of -d options - they will be cd'ed to in turn. If the paths are relative,
# make sure you get the sequence right. Be careful with relative paths -
# there are far too many symlinks nowadays.
# (default is the ftp login directory)
# -v turns on the verbose option of ftp, and shows all responses from the
# ftp server.
# -f remotefile[:localfile] gets the remote file into localfile
# -m pattern does an mget with the specified pattern. Remember to quote
# shell characters.
# -c does a local cd to the specified directory
# For example,
#   ftpget -h expo.lcs.mit.edu -d contrib -f xplaces.shar:xplaces.sh \
#          -d ../pub/R3/fixes -c ~/fixes -m 'fix*'
# will get xplaces.shar from ~ftp/contrib on expo.lcs.mit.edu, and put it in
# xplaces.sh in the current working directory, and get all fixes from
# ~ftp/pub/R3/fixes and put them in the ~/fixes directory.
# Obviously, the sequence of the options is important, since the equivalent
# commands are executed by ftp in corresponding order
#
# Mark Moraes <moraes@csri.toronto.edu>, Feb 1, 1989
#


# ==> These comments added by author of this document.

# PATH=/local/bin:/usr/ucb:/usr/bin:/bin
# export PATH
# ==> Above 2 lines from original script probably superfluous.

E_BADARGS=65

TMPFILE=/tmp/ftp.$$
# ==> Creates temp file, using process id of script ($$)
# ==> to construct filename.

SITE=`domainname`.toronto.edu
# ==> 'domainname' similar to 'hostname'
# ==> May rewrite this to parameterize this for general use.

usage="Usage: $0 [-h remotehost] [-d remotedirectory]... [-f remfile:localfile]... \
        [-c localdirectory] [-m filepattern] [-v]"
ftpflags="-i -n"
verbflag=
set -f              # So we can use globbing in -m
set x `getopt vh:d:c:m:f: $*`
if [ $? != 0 ]; then
    echo $usage
    exit $E_BADARGS
fi
shift
trap 'rm -f ${TMPFILE} ; exit' 0 1 2 3 15
# ==> Delete tempfile in case of abnormal exit from script.
echo "user anonymous ${USER-gnu}@${SITE} > ${TMPFILE}"
# ==> Added quotes (recommended in complex echoes).
```

```
echo binary >> ${TMPFILE}
for i in $*   # ==> Parse command line args.
do
    case $i in
    -v) verbflag=-v; echo hash >> ${TMPFILE}; shift;;
    -h) remhost=$2; shift 2;;
    -d) echo cd $2 >> ${TMPFILE};
        if [ x${verbflag} != x ]; then
            echo pwd >> ${TMPFILE};
        fi;
        shift 2;;
    -c) echo lcd $2 >> ${TMPFILE}; shift 2;;
    -m) echo mget "$2" >> ${TMPFILE}; shift 2;;
    -f) f1=`expr "$2" : "\([^:]*\).*"`; f2=`expr "$2" : "[^:]*:\(.*\)"`;
        echo get ${f1} ${f2} >> ${TMPFILE}; shift 2;;
    --) shift; break;;
    esac
        # ==> 'lcd' and 'mget' are ftp commands. See "man ftp" . . .
done
if [ $# -ne 0 ]; then
    echo $usage
    exit $E_BADARGS
        # ==> Changed from "exit 2" to conform with style standard.
fi
if [ x${verbflag} != x ]; then
    ftpflags="${ftpflags} -v"
fi
if [ x${remhost} = x ]; then
    remhost=prep.ai.mit.edu
    # ==> Change to match appropriate ftp site.
fi
echo quit >> ${TMPFILE}
# ==> All commands saved in tempfile.

ftp ${ftpflags} ${remhost} < ${TMPFILE}
# ==> Now, tempfile batch processed by ftp.

rm -f ${TMPFILE}
# ==> Finally, tempfile deleted (you may wish to copy it to a logfile).


# ==> Exercises:
# ==> ---------
# ==> 1) Add error checking.
# ==> 2) Add bells & whistles.
```

+

Antek Sawicki捐献了下面的脚本, 这个脚本非常聪明的使用了参数替换操作符, 我们在Section 9.3中讨论了参数替换操作符.

例子 A-14. password: 产生随机的8个字符的密码

```
#!/bin/bash
# May need to be invoked with  #!/bin/bash2  on older machines.
#
# Random password generator for Bash 2.x by Antek Sawicki <tenox@tenox.tc>,
# who generously gave permission to the document author to use it here.
#
# ==> Comments added by document author ==>


MATRIX="0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
# ==> Password will consist of alphanumeric characters.
LENGTH="8"
# ==> May change 'LENGTH' for longer password.


while [ "${n:=1}" -le "$LENGTH" ]
# ==> Recall that := is "default substitution" operator.
# ==> So, if 'n' has not been initialized, set it to 1.
do
    PASS="$PASS${MATRIX:$(($RANDOM%${#MATRIX})):1}"
    # ==> Very clever, but tricky.

    # ==> Starting from the innermost nesting...
    # ==> ${#MATRIX} returns length of array MATRIX.

    # ==> $RANDOM%${#MATRIX} returns random number between 1
    # ==> and [length of MATRIX] - 1.

    # ==> ${MATRIX:$(($RANDOM%${#MATRIX})):1}
    # ==> returns expansion of MATRIX at random position, by length 1.
    # ==> See {var:pos:len} parameter substitution in Chapter 9.
    # ==> and the associated examples.

    # ==> PASS=... simply pastes this result onto previous PASS (concatenation).

    # ==> To visualize this more clearly, uncomment the following line
    #                 echo "$PASS"
    # ==> to see PASS being built up,
    # ==> one character at a time, each iteration of the loop.

    let n+=1
    # ==> Increment 'n' for next pass.
done
```

```
44
45 echo "$PASS"      # ==> Or, redirect to a file, as desired.
46
47 exit 0
```

+

James R. Van Zandt捐献了这个脚本, 使用命名管道, 用他的话来说, ″引用与转义的真正练习″.

例子 A-15. fifo: 使用命名管道来做每日的备份

```
 1 #!/bin/bash
 2 # ==> Script by James R. Van Zandt, and used here with his permission.
 3
 4 # ==> Comments added by author of this document.
 5
 6
 7   HERE=`uname -n`    # ==> hostname
 8   THERE=bilbo
 9   echo "starting remote backup to $THERE at `date +%r`"
10   # ==> `date +%r` returns time in 12-hour format, i.e. "08:08:34 PM".
11
12   # make sure /pipe really is a pipe and not a plain file
13   rm -rf /pipe
14   mkfifo /pipe       # ==> Create a "named pipe", named "/pipe".
15
16   # ==> 'su xyz' runs commands as user "xyz".
17   # ==> 'ssh' invokes secure shell (remote login client).
18   su xyz -c "ssh $THERE \"cat >/home/xyz/backup/${HERE}-daily.tar.gz\" < /pipe"&
19   cd /
20   tar -czf - bin boot dev etc home info lib man root sbin share usr var >/pipe
21   # ==> Uses named pipe, /pipe, to communicate between processes:
22   # ==> 'tar/gzip' writes to /pipe and 'ssh' reads from /pipe.
23
24   # ==> The end result is this backs up the main directories, from / on down.
25
26   # ==>  What are the advantages of a "named pipe" in this situation,
27   # ==>+ as opposed to an "anonymous pipe", with |?
28   # ==>  Will an anonymous pipe even work here?
29
30
31   exit 0
```

+

Stephane Chazelas捐献了这个脚本, 用来展示如何不使用数组来产生素数

例子 A-16. 使用模操作符来产生素数

```
 1 #!/bin/bash
 2 # primes.sh: Generate prime numbers, without using arrays.
 3 # Script contributed by Stephane Chazelas.
 4
 5 #  This does *not* use the classic "Sieve of Eratosthenes" algorithm,
 6 #+ but instead uses the more intuitive method of testing each candidate number
 7 #+ for factors (divisors), using the "%" modulo operator.
 8
 9
10 LIMIT=1000                    # Primes 2 - 1000
11
12 Primes()
13 {
14  (( n = $1 + 1 ))             # Bump to next integer.
15  shift                        # Next parameter in list.
16 #  echo "_n=$n i=$i_"
17
18  if (( n == LIMIT ))
19  then echo $*
20  return
21  fi
22
23  for i; do                    # "i" gets set to "@", previous values of $n.
24 #   echo "-n=$n i=$i-"
25    (( i * i > n )) && break   # Optimization.
26    (( n % i )) && continue    # Sift out non-primes using modulo operator.
27    Primes $n $@               # Recursion inside loop.
28    return
29    done
30
31    Primes $n $@ $n            # Recursion outside loop.
32                               # Successively accumulate positional parameters.
33                               # "$@" is the accumulating list of primes.
34 }
35
36 Primes 1
37
38 exit 0
39
40 #  Uncomment lines 16 and 24 to help figure out what is going on.
41
42 #  Compare the speed of this algorithm for generating primes
43 #+ with the Sieve of Eratosthenes (ex68.sh).
```

```
44
45 #  Exercise: Rewrite this script without recursion, for faster execution.
```

+

这是Jordi Sanfeliu的*tree*脚本的升级版, 由Rick Boivie编写.

例子 A-17. tree: 显示目录树

```
 1 #!/bin/bash
 2 # tree.sh
 3
 4 #  Written by Rick Boivie.
 5 #  Used with permission.
 6 #  This is a revised and simplified version of a script
 7 #+ by Jordi Sanfeliu (and patched by Ian Kjos).
 8 #  This script replaces the earlier version used in
 9 #+ previous releases of the Advanced Bash Scripting Guide.
10
11 # ==> Comments added by the author of this document.
12
13
14 search () {
15 for dir in `echo *`
16 #  ==> `echo *` lists all the files in current working directory,
17 #+ ==> without line breaks.
18 #  ==> Similar effect to for dir in *
19 #  ==> but "dir in `echo *`" will not handle filenames with blanks.
20 do
21   if [ -d "$dir" ] ; then # ==> If it is a directory (-d)...
22   zz=0                    # ==> Temp variable, keeping track of directory level.
23   while [ $zz != $1 ]     # Keep track of inner nested loop.
24     do
25       echo -n "| "        # ==> Display vertical connector symbol,
26                           # ==> with 2 spaces & no line feed in order to indent.
27       zz=`expr $zz + 1`   # ==> Increment zz.
28     done
29
30     if [ -L "$dir" ] ; then # ==> If directory is a symbolic link...
31       echo "+---$dir" `ls -l $dir | sed 's/^.*'$dir' //'`
32       # ==> Display horiz. connector and list directory name, but...
33       # ==> delete date/time part of long listing.
34     else
35       echo "+---$dir"       # ==> Display horizontal connector symbol...
36       # ==> and print directory name.
37       numdirs=`expr $numdirs + 1` # ==> Increment directory count.
38       if cd "$dir" ; then         # ==> If can move to subdirectory...
39         search `expr $1 + 1`      # with recursion ;-)
40         # ==> Function calls itself.
41         cd ..
42       fi
43     fi
44   fi
45 done
46 }
47
48 if [ $# != 0 ] ; then
49   cd $1 # move to indicated directory.
50   #else # stay in current directory
51 fi
52
53 echo "Initial directory = `pwd`"
54 numdirs=0
55
56 search 0
57 echo "Total directories = $numdirs"
58
59 exit 0
```

经过Noah Friedman的授权, 他的*string function*脚本可以在本书中使用, 这个脚本本质上就是复制了一些C库的字符串操作函数.

例子 A-18. string functions: C风格的字符串函数

```
 1 #!/bin/bash
 2
 3 # string.bash --- bash emulation of string(3) library routines
 4 # Author: Noah Friedman <friedman@prep.ai.mit.edu>
 5 # ==>     Used with his kind permission in this document.
 6 # Created: 1992-07-01
 7 # Last modified: 1993-09-29
 8 # Public domain
 9
10 # Conversion to bash v2 syntax done by Chet Ramey
11
12 # Commentary:
13 # Code:
14
15 #:docstring strcat:
16 # Usage: strcat s1 s2
17 #
```

```
# Strcat appends the value of variable s2 to variable s1.
#
# Example:
#    a="foo"
#    b="bar"
#    strcat a b
#    echo $a
#    => foobar
#
#:end docstring:

###;;;autoload   ==> Autoloading of function commented out.
function strcat ()
{
    local s1_val s2_val

    s1_val=${!1}                        # indirect variable expansion
    s2_val=${!2}
    eval "$1"=\'"${s1_val}${s2_val}"\'
    # ==> eval $1='${s1_val}${s2_val}' avoids problems,
    # ==> if one of the variables contains a single quote.
}

#:docstring strncat:
# Usage: strncat s1 s2 $n
#
# Line strcat, but strncat appends a maximum of n characters from the value
# of variable s2.  It copies fewer if the value of variabl s2 is shorter
# than n characters.  Echoes result on stdout.
#
# Example:
#    a=foo
#    b=barbaz
#    strncat a b 3
#    echo $a
#    => foobar
#
#:end docstring:

###;;;autoload
function strncat ()
{
    local s1="$1"
    local s2="$2"
    local -i n="$3"
    local s1_val s2_val

    s1_val=${!s1}                       # ==> indirect variable expansion
    s2_val=${!s2}

    if [ ${#s2_val} -gt ${n} ]; then
       s2_val=${s2_val:0:$n}            # ==> substring extraction
    fi

    eval "$s1"=\'"${s1_val}${s2_val}"\'
    # ==> eval $1='${s1_val}${s2_val}' avoids problems,
    # ==> if one of the variables contains a single quote.
}

#:docstring strcmp:
# Usage: strcmp $s1 $s2
#
# Strcmp compares its arguments and returns an integer less than, equal to,
# or greater than zero, depending on whether string s1 is lexicographically
# less than, equal to, or greater than string s2.
#:end docstring:

###;;;autoload
function strcmp ()
{
    [ "$1" = "$2" ] && return 0

    [ "${1}" '<' "${2}" ] > /dev/null && return -1

    return 1
}

#:docstring strncmp:
# Usage: strncmp $s1 $s2 $n
#
# Like strcmp, but makes the comparison by examining a maximum of n
# characters (n less than or equal to zero yields equality).
#:end docstring:

###;;;autoload
function strncmp ()
{
    if [ -z "${3}" -o "${3}" -le "0" ]; then
       return 0
    fi

    if [ ${3} -ge ${#1} -a ${3} -ge ${#2} ]; then
       strcmp "$1" "$2"
       return $?
    else
       s1=${1:0:$3}
       s2=${2:0:$3}
       strcmp $s1 $s2
       return $?
    fi
}

```

```
#:docstring strlen:
# Usage: strlen s
#
# Strlen returns the number of characters in string literal s.
#:end docstring:

###;;;autoload
function strlen ()
{
    eval echo "\${#${1}}"
    # ==> Returns the length of the value of the variable
    # ==> whose name is passed as an argument.
}

#:docstring strspn:
# Usage: strspn $s1 $s2
#
# Strspn returns the length of the maximum initial segment of string s1,
# which consists entirely of characters from string s2.
#:end docstring:

###;;;autoload
function strspn ()
{
    # Unsetting IFS allows whitespace to be handled as normal chars.
    local IFS=
    local result="${1%%[!${2}]*}"

    echo ${#result}
}

#:docstring strcspn:
# Usage: strcspn $s1 $s2
#
# Strcspn returns the length of the maximum initial segment of string s1,
# which consists entirely of characters not from string s2.
#:end docstring:

###;;;autoload
function strcspn ()
{
    # Unsetting IFS allows whitspace to be handled as normal chars.
    local IFS=
    local result="${1%%[${2}]*}"

    echo ${#result}
}

#:docstring strstr:
# Usage: strstr s1 s2
#
# Strstr echoes a substring starting at the first occurrence of string s2 in
# string s1, or nothing if s2 does not occur in the string.  If s2 points to
# a string of zero length, strstr echoes s1.
#:end docstring:

###;;;autoload
function strstr ()
{
    # if s2 points to a string of zero length, strstr echoes s1
    [ ${#2} -eq 0 ] && { echo "$1" ; return 0; }

    # strstr echoes nothing if s2 does not occur in s1
    case "$1" in
    *$2*) ;;
    *) return 1;;
    esac

    # use the pattern matching code to strip off the match and everything
    # following it
    first=${1/$2*/}

    # then strip off the first unmatched portion of the string
    echo "${1##$first}"
}

#:docstring strtok:
# Usage: strtok s1 s2
#
# Strtok considers the string s1 to consist of a sequence of zero or more
# text tokens separated by spans of one or more characters from the
# separator string s2.  The first call (with a non-empty string s1
# specified) echoes a string consisting of the first token on stdout. The
# function keeps track of its position in the string s1 between separate
# calls, so that subsequent calls made with the first argument an empty
# string will work through the string immediately following that token.  In
# this way subsequent calls will work through the string s1 until no tokens
# remain.  The separator string s2 may be different from call to call.
# When no token remains in s1, an empty value is echoed on stdout.
#:end docstring:

###;;;autoload
function strtok ()
{
 :
}

#:docstring strtrunc:
# Usage: strtrunc $n $s1 {$s2} {$...}
#
# Used by many functions like strncmp to truncate arguments for comparison.
# Echoes the first n characters of each string s1 s2 ... on stdout.
```

```
#:end docstring:

###;;;autoload
function strtrunc ()
{
    n=$1 ; shift
    for z; do
        echo "${z:0:$n}"
    done
}

# provide string

# string.bash ends here


# ========================================================================== #
# ==> Everything below here added by the document author.

# ==> Suggested use of this script is to delete everything below here,
# ==> and "source" this file into your own scripts.

# strcat
string0=one
string1=two
echo
echo "Testing \"strcat\" function:"
echo "Original \"string0\" = $string0"
echo "\"string1\" = $string1"
strcat string0 string1
echo "New \"string0\" = $string0"
echo

# strlen
echo
echo "Testing \"strlen\" function:"
str=123456789
echo "\"str\" = $str"
echo -n "Length of \"str\" = "
strlen str
echo



# Exercise:
# --------
# Add code to test all the other string functions above.


exit 0
```

这个复杂的数组用例使用了md5sum检查和命令来编码目录信息, 此脚本由Michael Zick编写.

例子 A-19. 目录信息

```
#! /bin/bash
# directory-info.sh
# Parses and lists directory information.

# NOTE: Change lines 273 and 353 per "README" file.

# Michael Zick is the author of this script.
# Used here with his permission.

# Controls
# If overridden by command arguments, they must be in the order:
#   Arg1: "Descriptor Directory"
#   Arg2: "Exclude Paths"
#   Arg3: "Exclude Directories"
#
# Environment Settings override Defaults.
# Command arguments override Environment Settings.

# Default location for content addressed file descriptors.
MD5UCFS=${1:-${MD5UCFS:-'/tmpfs/ucfs'}}

# Directory paths never to list or enter
declare -a \
  EXCLUDE_PATHS=${2:-${EXCLUDE_PATHS:-'(/proc /dev /devfs /tmpfs)'}}

# Directories never to list or enter
declare -a \
  EXCLUDE_DIRS=${3:-${EXCLUDE_DIRS:-'(ucfs lost+found tmp wtmp)'}}

# Files never to list or enter
declare -a \
  EXCLUDE_FILES=${3:-${EXCLUDE_FILES:-'(core "Name with Spaces")'}}


# Here document used as a comment block.
: <<LSfieldsDoc
# # # # # List Filesystem Directory Information # # # # #
#
#   ListDirectory "FileGlob" "Field-Array-Name"
# or
#   ListDirectory -of "FileGlob" "Field-Array-Filename"
#   '-of' meaning 'output to filename'
```

```
# # # # #

String format description based on: ls (GNU fileutils) version 4.0.36

Produces a line (or more) formatted:
inode permissions hard-links owner group ...
32736 -rw-------    1 mszick   mszick

size    day month date hh:mm:ss year path
2756608 Sun Apr 20 08:53:06 2003 /home/mszick/core

Unless it is formatted:
inode permissions hard-links owner group ...
266705 crw-rw----    1    root  uucp

major minor day month date hh:mm:ss year path
4,  68 Sun Apr 20 09:27:33 2003 /dev/ttyS4
NOTE: that pesky comma after the major number

NOTE: the 'path' may be multiple fields:
/home/mszick/core
/proc/982/fd/0 -> /dev/null
/proc/982/fd/1 -> /home/mszick/.xsession-errors
/proc/982/fd/13 -> /tmp/tmpfZVVOCs (deleted)
/proc/982/fd/7 -> /tmp/kde-mszick/ksycoca
/proc/982/fd/8 -> socket:[11586]
/proc/982/fd/9 -> pipe:[11588]

If that isn't enough to keep your parser guessing,
either or both of the path components may be relative:
../Built-Shared -> Built-Static
../linux-2.4.20.tar.bz2 -> ../../../SRCS/linux-2.4.20.tar.bz2

The first character of the 11 (10?) character permissions field:
's' Socket
'd' Directory
'b' Block device
'c' Character device
'l' Symbolic link
NOTE: Hard links not marked - test for identical inode numbers
on identical filesystems.
All information about hard linked files are shared, except
for the names and the name's location in the directory system.
NOTE: A "Hard link" is known as a "File Alias" on some systems.
'-' An undistingushed file

Followed by three groups of letters for: User, Group, Others
Character 1: '-' Not readable; 'r' Readable
Character 2: '-' Not writable; 'w' Writable
Character 3, User and Group: Combined execute and special
'-' Not Executable, Not Special
'x' Executable, Not Special
's' Executable, Special
'S' Not Executable, Special
Character 3, Others: Combined execute and sticky (tacky?)
'-' Not Executable, Not Tacky
'x' Executable, Not Tacky
't' Executable, Tacky
'T' Not Executable, Tacky

Followed by an access indicator
Haven't tested this one, it may be the eleventh character
or it may generate another field
' ' No alternate access
'+' Alternate access
LSfieldsDoc


ListDirectory()
{
    local -a T
    local -i of=0           # Default return in variable
#   OLD_IFS=$IFS            # Using BASH default ' \t\n'

    case "$#" in
    3)      case "$1" in
            -of)    of=1 ; shift ;;
             * )    return 1 ;;
            esac ;;
    2)      : ;;            # Poor man's "continue"
    *)      return 1 ;;
    esac

    # NOTE: the (ls) command is NOT quoted (")
    T=( $(ls --inode --ignore-backups --almost-all --directory \
    --full-time --color=none --time=status --sort=none \
    --format=long $1) )

    case $of in
    # Assign T back to the array whose name was passed as $2
            0) eval $2=\( \"\$\{T\[@\]\}\" \) ;;
    # Write T into filename passed as $2
            1) echo "${T[@]}" > "$2" ;;
    esac
    return 0
   }

# # # # # Is that string a legal number? # # # # #
#
#   IsNumber "Var"
# # # # # There has to be a better way, sigh...

```

```
IsNumber()
{
    local -i int
    if [ $# -eq 0 ]
    then
            return 1
    else
            (let int=$1)  2>/dev/null
            return $?       # Exit status of the let thread
    fi
}

# # # # # Index Filesystem Directory Information # # # # #
#
#   IndexList "Field-Array-Name" "Index-Array-Name"
# or
#   IndexList -if Field-Array-Filename Index-Array-Name
#   IndexList -of Field-Array-Name Index-Array-Filename
#   IndexList -if -of Field-Array-Filename Index-Array-Filename
# # # # # #

: <<IndexListDoc
Walk an array of directory fields produced by ListDirectory

Having suppressed the line breaks in an otherwise line oriented
report, build an index to the array element which starts each line.

Each line gets two index entries, the first element of each line
(inode) and the element that holds the pathname of the file.

The first index entry pair (Line-Number==0) are informational:
Index-Array-Name[0] : Number of "Lines" indexed
Index-Array-Name[1] : "Current Line" pointer into Index-Array-Name

The following index pairs (if any) hold element indexes into
the Field-Array-Name per:
Index-Array-Name[Line-Number * 2] : The "inode" field element.
NOTE: This distance may be either +11 or +12 elements.
Index-Array-Name[(Line-Number * 2) + 1] : The "pathname" element.
NOTE: This distance may be a variable number of elements.
Next line index pair for Line-Number+1.
IndexListDoc



IndexList()
{
    local -a LIST                   # Local of listname passed
    local -a -i INDEX=( 0 0 )       # Local of index to return
    local -i Lidx Lcnt
    local -i if=0 of=0              # Default to variable names

    case "$#" in                    # Simplistic option testing
            0) return 1 ;;
            1) return 1 ;;
            2) : ;;                 # Poor man's continue
            3) case "$1" in
                    -if) if=1 ;;
                    -of) of=1 ;;
                     * ) return 1 ;;
               esac ; shift ;;
            4) if=1 ; of=1 ; shift ; shift ;;
            *) return 1
    esac

    # Make local copy of list
    case "$if" in
            0) eval LIST=\( \"\$\{$1\[@\]\}\" \) ;;
            1) LIST=( $(cat $1) ) ;;
    esac

    # Grok (grope?) the array
    Lcnt=${#LIST[@]}
    Lidx=0
    until (( Lidx >= Lcnt ))
    do
    if IsNumber ${LIST[$Lidx]}
    then
            local -i inode name
            local ft
            inode=Lidx
            local m=${LIST[$Lidx+2]}        # Hard Links field
            ft=${LIST[$Lidx+1]:0:1}         # Fast-Stat
            case $ft in
            b)      ((Lidx+=12)) ;;         # Block device
            c)      ((Lidx+=12)) ;;         # Character device
            *)      ((Lidx+=11)) ;;         # Anything else
            esac
            name=Lidx
            case $ft in
            -)      ((Lidx+=1)) ;;          # The easy one
            b)      ((Lidx+=1)) ;;          # Block device
            c)      ((Lidx+=1)) ;;          # Character device
            d)      ((Lidx+=1)) ;;          # The other easy one
            l)      ((Lidx+=3)) ;;          # At LEAST two more fields
#  A little more elegance here would handle pipes,
#+ sockets, deleted files - later.
            *)      until IsNumber ${LIST[$Lidx]} || ((Lidx >= Lcnt))
                    do
                            ((Lidx+=1))
                    done
                    ;;                      # Not required
```

```
            esac
            INDEX[${#INDEX[*]}]=$inode
            INDEX[${#INDEX[*]}]=$name
            INDEX[0]=${INDEX[0]}+1          # One more "line" found
# echo "Line: ${INDEX[0]} Type: $ft Links: $m Inode: \
# ${LIST[$inode]} Name: ${LIST[$name]}"

    else
            ((Lidx+=1))
    fi
    done
    case "$of" in
            0) eval $2=\( \"\$\{INDEX\[@\]\}\" \) ;;
            1) echo "${INDEX[@]}" > "$2" ;;
    esac
    return 0                                # What could go wrong?
}

# # # # # Content Identify File # # # # #
#
#   DigestFile Input-Array-Name Digest-Array-Name
# or
#   DigestFile -if Input-FileName Digest-Array-Name
# # # # #

# Here document used as a comment block.
: <<DigestFilesDoc

The key (no pun intended) to a Unified Content File System (UCFS)
is to distinguish the files in the system based on their content.
Distinguishing files by their name is just, so, 20th Century.

The content is distinguished by computing a checksum of that content.
This version uses the md5sum program to generate a 128 bit checksum
representative of the file's contents.
There is a chance that two files having different content might
generate the same checksum using md5sum (or any checksum).  Should
that become a problem, then the use of md5sum can be replace by a
cyrptographic signature.  But until then...

The md5sum program is documented as outputting three fields (and it
does), but when read it appears as two fields (array elements).  This
is caused by the lack of whitespace between the second and third field.
So this function gropes the md5sum output and returns:
    [0]     32 character checksum in hexidecimal (UCFS filename)
    [1]     Single character: ' ' text file, '*' binary file
    [2]     Filesystem (20th Century Style) name
    Note: That name may be the character '-' indicating STDIN read.

DigestFilesDoc



DigestFile()
{
    local if=0              # Default, variable name
    local -a T1 T2

    case "$#" in
    3)      case "$1" in
            -if)    if=1 ; shift ;;
             * )    return 1 ;;
            esac ;;
    2)      : ;;            # Poor man's "continue"
    *)      return 1 ;;
    esac

    case $if in
    0) eval T1=\( \"\$\{$1\[@\]\}\" \)
       T2=( $(echo ${T1[@]} | md5sum -) )
       ;;
    1) T2=( $(md5sum $1) )
       ;;
    esac

    case ${#T2[@]} in
    0) return 1 ;;
    1) return 1 ;;
    2) case ${T2[1]:0:1} in          # SanScrit-2.0.5
       \*) T2[${#T2[@]}]=${T2[1]:1}
           T2[1]=\*
           ;;
        *) T2[${#T2[@]}]=${T2[1]}
           T2[1]=" "
           ;;
       esac
       ;;
    3) : ;; # Assume it worked
    *) return 1 ;;
    esac

    local -i len=${#T2[0]}
    if [ $len -ne 32 ] ; then return 1 ; fi
    eval $2=\( \"\$\{T2\[@\]\}\" \)
}

# # # # # Locate File # # # # #
#
#   LocateFile [-l] FileName Location-Array-Name
# or
#   LocateFile [-l] -of FileName Location-Array-FileName
# # # # #
```

```
349
350 # A file location is Filesystem-id and inode-number
351
352 # Here document used as a comment block.
353 : <<StatFieldsDoc
354     Based on stat, version 2.2
355     stat -t and stat -lt fields
356     [0]     name
357     [1]     Total size
358             File - number of bytes
359             Symbolic link - string length of pathname
360     [2]     Number of (512 byte) blocks allocated
361     [3]     File type and Access rights (hex)
362     [4]     User ID of owner
363     [5]     Group ID of owner
364     [6]     Device number
365     [7]     Inode number
366     [8]     Number of hard links
367     [9]     Device type (if inode device) Major
368     [10]    Device type (if inode device) Minor
369     [11]    Time of last access
370             May be disabled in 'mount' with noatime
371             atime of files changed by exec, read, pipe, utime, mknod (mmap?)
372             atime of directories changed by addition/deletion of files
373     [12]    Time of last modification
374             mtime of files changed by write, truncate, utime, mknod
375             mtime of directories changed by addtition/deletion of files
376     [13]    Time of last change
377             ctime reflects time of changed inode information (owner, group
378             permissions, link count
379 -*-*- Per:
380     Return code: 0
381     Size of array: 14
382     Contents of array
383     Element 0: /home/mszick
384     Element 1: 4096
385     Element 2: 8
386     Element 3: 41e8
387     Element 4: 500
388     Element 5: 500
389     Element 6: 303
390     Element 7: 32385
391     Element 8: 22
392     Element 9: 0
393     Element 10: 0
394     Element 11: 1051221030
395     Element 12: 1051214068
396     Element 13: 1051214068
397
398     For a link in the form of linkname -> realname
399     stat -t  linkname returns the linkname (link) information
400     stat -lt linkname returns the realname information
401
402     stat -tf and stat -ltf fields
403     [0]     name
404     [1]     ID-0?           # Maybe someday, but Linux stat structure
405     [2]     ID-0?           # does not have either LABEL nor UUID
406                             # fields, currently information must come
407                             # from file-system specific utilities
408     These will be munged into:
409     [1]     UUID if possible
410     [2]     Volume Label if possible
411     Note: 'mount -l' does return the label and could return the UUID
412
413     [3]     Maximum length of filenames
414     [4]     Filesystem type
415     [5]     Total blocks in the filesystem
416     [6]     Free blocks
417     [7]     Free blocks for non-root user(s)
418     [8]     Block size of the filesystem
419     [9]     Total inodes
420     [10]    Free inodes
421
422 -*-*- Per:
423     Return code: 0
424     Size of array: 11
425     Contents of array
426     Element 0: /home/mszick
427     Element 1: 0
428     Element 2: 0
429     Element 3: 255
430     Element 4: ef53
431     Element 5: 2581445
432     Element 6: 2277180
433     Element 7: 2146050
434     Element 8: 4096
435     Element 9: 1311552
436     Element 10: 1276425
437
438 StatFieldsDoc
439
440
441 #   LocateFile [-l] FileName Location-Array-Name
442 #   LocateFile [-l] -of FileName Location-Array-FileName
443
444 LocateFile()
445 {
446     local -a LOC LOC1 LOC2
447     local lk="" of=0
448
449     case "$#" in
450     0) return 1 ;;
```

```
451     1) return 1 ;;
452     2) : ;;
453     *) while (( "$#" > 2 ))
454        do
455           case "$1" in
456            -l) lk=-1 ;;
457           -of) of=1 ;;
458             *) return 1 ;;
459           esac
460        shift
461            done ;;
462     esac
463
464 # More Sanscrit-2.0.5
465       # LOC1=( $(stat -t $lk $1) )
466       # LOC2=( $(stat -tf $lk $1) )
467       # Uncomment above two lines if system has "stat" command installed.
468     LOC=( ${LOC1[@]:0:1} ${LOC1[@]:3:11}
469           ${LOC2[@]:1:2} ${LOC2[@]:4:1} )
470
471     case "$of" in
472             0) eval $2=\( \"\$\{LOC\[@\]\}\" \) ;;
473             1) echo "${LOC[@]}" > "$2" ;;
474     esac
475     return 0
476 # Which yields (if you are lucky, and have "stat" installed)
477 # -*-*- Location Discriptor -*-*-
478 #   Return code: 0
479 #   Size of array: 15
480 #   Contents of array
481 #   Element 0: /home/mszick         20th Century name
482 #   Element 1: 41e8                 Type and Permissions
483 #   Element 2: 500                  User
484 #   Element 3: 500                  Group
485 #   Element 4: 303                  Device
486 #   Element 5: 32385                inode
487 #   Element 6: 22                   Link count
488 #   Element 7: 0                    Device Major
489 #   Element 8: 0                    Device Minor
490 #   Element 9: 1051224608           Last Access
491 #   Element 10: 1051214068          Last Modify
492 #   Element 11: 1051214068          Last Status
493 #   Element 12: 0                   UUID (to be)
494 #   Element 13: 0                   Volume Label (to be)
495 #   Element 14: ef53                Filesystem type
496 }
497
498
499
500 # And then there was some test code
501
502 ListArray() # ListArray Name
503 {
504     local -a Ta
505
506     eval Ta=\( \"\$\{$1\[@\]\}\" \)
507     echo
508     echo "-*-*- List of Array -*-*-"
509     echo "Size of array $1: ${#Ta[*]}"
510     echo "Contents of array $1:"
511     for (( i=0 ; i<${#Ta[*]} ; i++ ))
512     do
513         echo -e "\tElement $i: ${Ta[$i]}"
514     done
515     return 0
516 }
517
518 declare -a CUR_DIR
519 # For small arrays
520 ListDirectory "${PWD}" CUR_DIR
521 ListArray CUR_DIR
522
523 declare -a DIR_DIG
524 DigestFile CUR_DIR DIR_DIG
525 echo "The new \"name\" (checksum) for ${CUR_DIR[9]} is ${DIR_DIG[0]}"
526
527 declare -a DIR_ENT
528 # BIG_DIR # For really big arrays - use a temporary file in ramdisk
529 # BIG-DIR # ListDirectory -of "${CUR_DIR[11]}/*" "/tmpfs/junk2"
530 ListDirectory "${CUR_DIR[11]}/*" DIR_ENT
531
532 declare -a DIR_IDX
533 # BIG-DIR # IndexList -if "/tmpfs/junk2" DIR_IDX
534 IndexList DIR_ENT DIR_IDX
535
536 declare -a IDX_DIG
537 # BIG-DIR # DIR_ENT=( $(cat /tmpfs/junk2) )
538 # BIG-DIR # DigestFile -if /tmpfs/junk2 IDX_DIG
539 DigestFile DIR_ENT IDX_DIG
540 # Small (should) be able to parallize IndexList & DigestFile
541 # Large (should) be able to parallize IndexList & DigestFile & the assignment
542 echo "The \"name\" (checksum) for the contents of ${PWD} is ${IDX_DIG[0]}"
543
544 declare -a FILE_LOC
545 LocateFile ${PWD} FILE_LOC
546 ListArray FILE_LOC
547
548 exit 0
```

Stephane Chazelas向我们展示了如何在Bash脚本中使用面向对象的编程方法.

例子 A-20. 面向对象数据库

```
#!/bin/bash
# obj-oriented.sh: Object-oriented programming in a shell script.
# Script by Stephane Chazelas.

#  Important Note:
#  --------- ----
#  If running this script under version 3 or later of Bash,
#+ replace all periods in function names with a "legal" character,
#+ for example, an underscore.


person.new()        # Looks almost like a class declaration in C++.
{
  local obj_name=$1 name=$2 firstname=$3 birthdate=$4

  eval "$obj_name.set_name() {
          eval \"$obj_name.get_name() {
                  echo \$1
                }\"
        }"

  eval "$obj_name.set_firstname() {
          eval \"$obj_name.get_firstname() {
                  echo \$1
                }\"
        }"

  eval "$obj_name.set_birthdate() {
          eval \"$obj_name.get_birthdate() {
            echo \$1
          }\"
          eval \"$obj_name.show_birthdate() {
            echo \$(date -d \"1/1/1970 0:0:\$1 GMT\")
          }\"
          eval \"$obj_name.get_age() {
            echo \$(( (\$(date +%s) - \$1) / 3600 / 24 / 365 ))
          }\"
        }"

  $obj_name.set_name $name
  $obj_name.set_firstname $firstname
  $obj_name.set_birthdate $birthdate
}

echo

person.new self Bozeman Bozo 101272413
# Create an instance of "person.new" (actually passing args to the function).

self.get_firstname       #   Bozo
self.get_name            #   Bozeman
self.get_age             #   28
self.get_birthdate       #   101272413
self.show_birthdate      #   Sat Mar 17 20:13:33 MST 1973

echo

#  typeset -f
#+ to see the created functions (careful, it scrolls off the page).

exit 0
```

Mariusz Gniazdowski发布了一个可以在脚本中使用的hash库.

例子 A-21. hash函数库

```
# Hash:
# Hash function library
# Author: Mariusz Gniazdowski <mgniazd-at-gmail.com>
# Date: 2005-04-07

# Functions making emulating hashes in Bash a little less painful.


#    Limitations:
#  * Only global variables are supported.
#  * Each hash instance generates one global variable per value.
#  * Variable names collisions are possible
#+   if you define variable like __hash__hashname_key
#  * Keys must use chars that can be part of a Bash variable name
#+   (no dashes, periods, etc.).
#  * The hash is created as a variable:
#    ... hashname_keyname
#    So if somone will create hashes like:
#      myhash_ + mykey = myhash__mykey
#      myhash + _mykey = myhash__mykey
#    Then there will be a collision.
#    (This should not pose a major problem.)


Hash_config_varname_prefix=__hash__

```

```
 27
 28 # Emulates:  hash[key]=value
 29 #
 30 # Params:
 31 # 1 - hash
 32 # 2 - key
 33 # 3 - value
 34 function hash_set {
 35     eval "${Hash_config_varname_prefix}${1}_${2}=\"${3}\""
 36 }
 37
 38
 39 # Emulates:  value=hash[key]
 40 #
 41 # Params:
 42 # 1 - hash
 43 # 2 - key
 44 # 3 - value (name of global variable to set)
 45 function hash_get_into {
 46     eval "$3=\"\$${Hash_config_varname_prefix}${1}_${2}\""
 47 }
 48
 49
 50 # Emulates:  echo hash[key]
 51 #
 52 # Params:
 53 # 1 - hash
 54 # 2 - key
 55 # 3 - echo params (like -n, for example)
 56 function hash_echo {
 57     eval "echo $3 \"\$${Hash_config_varname_prefix}${1}_${2}\""
 58 }
 59
 60
 61 # Emulates:  hash1[key1]=hash2[key2]
 62 #
 63 # Params:
 64 # 1 - hash1
 65 # 2 - key1
 66 # 3 - hash2
 67 # 4 - key2
 68 function hash_copy {
 69     eval "${Hash_config_varname_prefix}${1}_${2}=\"\$${Hash_config_varname_prefix}${3}_${4}\""
 70 }
 71
 72
 73 # Emulates:  hash[keyN-1]=hash[key2]=...hash[key1]
 74 #
 75 # Copies first key to rest of keys.
 76 #
 77 # Params:
 78 # 1 - hash1
 79 # 2 - key1
 80 # 3 - key2
 81 # . . .
 82 # N - keyN
 83 function hash_dup {
 84     local hashName="$1" keyName="$2"
 85     shift 2
 86     until [ ${#} -le 0 ]; do
 87         eval
"${Hash_config_varname_prefix}${hashName}_${1}=\"\$${Hash_config_varname_prefix}${hashName}_${keyName}\""
 88         shift;
 89     done;
 90 }
 91
 92
 93 # Emulates:  unset hash[key]
 94 #
 95 # Params:
 96 # 1 - hash
 97 # 2 - key
 98 function hash_unset {
 99     eval "unset ${Hash_config_varname_prefix}${1}_${2}"
100 }
101
102
103 # Emulates something similar to:  ref=&hash[key]
104 #
105 # The reference is name of the variable in which value is held.
106 #
107 # Params:
108 # 1 - hash
109 # 2 - key
110 # 3 - ref - Name of global variable to set.
111 function hash_get_ref_into {
112     eval "$3=\"${Hash_config_varname_prefix}${1}_${2}\""
113 }
114
115
116 # Emulates something similar to:  echo &hash[key]
117 #
118 # That reference is name of variable in which value is held.
119 #
120 # Params:
121 # 1 - hash
122 # 2 - key
123 # 3 - echo params (like -n for example)
124 function hash_echo_ref {
125     eval "echo $3 \"${Hash_config_varname_prefix}${1}_${2}\""
126 }
127
```

```
128
129
130 # Emulates something similar to:  $$hash[key](param1, param2, ...)
131 #
132 # Params:
133 # 1 - hash
134 # 2 - key
135 # 3,4, ... - Function parameters
136 function hash_call {
137     local hash key
138     hash=$1
139     key=$2
140     shift 2
141     eval "eval \"\$${Hash_config_varname_prefix}${hash}_${key} \\\"\\\$@\\\"\""
142 }
143
144
145 # Emulates something similar to:  isset(hash[key]) or hash[key]==NULL
146 #
147 # Params:
148 # 1 - hash
149 # 2 - key
150 # Returns:
151 # 0 - there is such key
152 # 1 - there is no such key
153 function hash_is_set {
154     eval "if [[ \"\${${Hash_config_varname_prefix}${1}_${2}-a}\" = \"a\" &&
155                 \"\${${Hash_config_varname_prefix}${1}_${2}-b}\" = \"b\" ]]; then return 1;
else return 0; fi"
156 }
157
158
159 # Emulates something similar to:
160 #   foreach($hash as $key => $value) { fun($key,$value); }
161 #
162 # It is possible to write different variations of this function.
163 # Here we use a function call to make it as "generic" as possible.
164 #
165 # Params:
166 # 1 - hash
167 # 2 - function name
168 function hash_foreach {
169     local keyname oldIFS="$IFS"
170     IFS=' '
171     for i in $(eval "echo \${!${Hash_config_varname_prefix}${1}_*}"); do
172             keyname=$(eval "echo \${i##${Hash_config_varname_prefix}${1}_}")
173             eval "$2 $keyname \"\$$i\""
174     done
175     IFS="$oldIFS"
176 }
177
178 # NOTE: In lines 103 and 116, ampersand changed.
179 #       But, it doesn't matter, because these are comment lines anyhow.
```

这是个例子脚本, 这个脚本使用了前面的hash库.

例子 A-22. 使用hash函数来给文本上色

```
  1 #!/bin/bash
  2 # hash-example.sh: Colorizing text.
  3 # Author: Mariusz Gniazdowski <mgniazd-at-gmail.com>
  4
  5 . Hash.lib      # Load the library of functions.
  6
  7 hash_set colors red          "\033[0;31m"
  8 hash_set colors blue         "\033[0;34m"
  9 hash_set colors light_blue   "\033[1;34m"
 10 hash_set colors light_red    "\033[1;31m"
 11 hash_set colors cyan         "\033[0;36m"
 12 hash_set colors light_green  "\033[1;32m"
 13 hash_set colors light_gray   "\033[0;37m"
 14 hash_set colors green        "\033[0;32m"
 15 hash_set colors yellow       "\033[1;33m"
 16 hash_set colors light_purple "\033[1;35m"
 17 hash_set colors purple       "\033[0;35m"
 18 hash_set colors reset_color  "\033[0;00m"
 19
 20
 21 # $1 - keyname
 22 # $2 - value
 23 try_colors() {
 24     echo -en "$2"
 25     echo "This line is $1."
 26 }
 27 hash_foreach colors try_colors
 28 hash_echo colors reset_color -en
 29
 30 echo -e '\nLet us overwrite some colors with yellow.\n'
 31 # It's hard to read yellow text on some terminals.
 32 hash_dup colors yellow   red light_green blue green light_gray cyan
 33 hash_foreach colors try_colors
 34 hash_echo colors reset_color -en
 35
 36 echo -e '\nLet us delete them and try colors once more . . .\n'
 37
 38 for i in red light_green blue green light_gray cyan; do
 39     hash_unset colors $i
```

```
done
hash_foreach colors try_colors
hash_echo colors reset_color -en

hash_set other txt "Other examples . . ."
hash_echo other txt
hash_get_into other txt text
echo $text

hash_set other my_fun try_colors
hash_call other my_fun   purple "`hash_echo colors purple`"
hash_echo colors reset_color -en

echo; echo "Back to normal?"; echo

exit $?

#  On some terminals, the "light" colors print in bold,
#  and end up looking darker than the normal ones.
#  Why is this?
```

站在一个比较难的观点来阐明hash的结构.

例子 A-23. 深入hash函数

```
#!/bin/bash
# $Id: ha.sh,v 1.2 2005/04/21 23:24:26 oliver Exp $
# Copyright 2005 Oliver Beckstein
# Released under the GNU Public License
# Author of script granted permission for inclusion in ABS Guide.
# (Thank you!)

#-------------------------------------------------------------------
# pseudo hash based on indirect parameter expansion
# API: access through functions:
#
# create the hash:
#
#      newhash Lovers
#
# add entries (note single quotes for spaces)
#
#      addhash Lovers Tristan Isolde
#      addhash Lovers 'Romeo Montague' 'Juliet Capulet'
#
# access value by key
#
#      gethash Lovers Tristan   ---->  Isolde
#
# show all keys
#
#      keyshash Lovers         ----> 'Tristan'  'Romeo Montague'
#
#
# convention: instead of perls' foo{bar} = boing' syntax,
# use
#       '_foo_bar=boing' (two underscores, no spaces)
#
# 1) store key   in _NAME_keys[]
# 2) store value in _NAME_values[] using the same integer index
# The integer index for the last entry is _NAME_ptr
#
# NOTE: No error or sanity checks, just bare bones.


function _inihash () {
    # private function
    # call at the beginning of each procedure
    # defines: _keys _values _ptr
    #
    # usage: _inihash NAME
    local name=$1
    _keys=_${name}_keys
    _values=_${name}_values
    _ptr=_${name}_ptr
}

function newhash () {
    # usage: newhash NAME
    #        NAME should not contain spaces or '.';
    #        actually: it must be a legal name for a bash variable
    # We rely on bash automatically recognising arrays.
    local name=$1
    local _keys _values _ptr
    _inihash ${name}
    eval ${_ptr}=0
}


function addhash () {
    # usage: addhash NAME KEY 'VALUE with spaces'
    #        arguments with spaces need to be quoted with single quotes ''
    local name=$1 k="$2" v="$3"
    local _keys _values _ptr
    _inihash ${name}

    #echo "DEBUG(addhash): ${_ptr}=${!_ptr}"
```

```
 73
 74     eval let ${_ptr}=${_ptr}+1
 75     eval "$_keys[${!_ptr}]=\"${k}\""
 76     eval "$_values[${!_ptr}]=\"${v}\""
 77 }
 78
 79 function gethash () {
 80     # usage: gethash NAME KEY
 81     #        returns boing
 82     #        ERR=0 if entry found, 1 otherwise
 83     # Thats not a proper hash---we simply linearly search through the keys
 84     local name=$1 key="$2"
 85     local _keys _values _ptr
 86     local k v i found h
 87     _inihash ${name}
 88
 89     # _ptr holds the highest index in the hash
 90     found=0
 91
 92     for i in $(seq 1 ${!_ptr}); do
 93     h="\${${_keys}[${i}]}"  # safer to do it in two steps
 94     eval k=${h}             # (especially when quoting for spaces)
 95     if [ "${k}" = "${key}" ]; then found=1; break; fi
 96     done;
 97
 98     [ ${found} = 0 ] && return 1;
 99     # else: i is the index that matches the key
100     h="\${${_values}[${i}]}"
101     eval echo "${h}"
102     return 0;
103 }
104
105 function keyshash () {
106     # usage: keyshash NAME
107     # returns list of all keys defined for hash name
108     local name=$1 key="$2"
109     local _keys _values _ptr
110     local k i h
111     _inihash ${name}
112
113     # _ptr holds the highest index in the hash
114     for i in $(seq 1 ${!_ptr}); do
115     h="\${${_keys}[${i}]}"   # Safer to do it in two steps
116     eval k=${h}              # (especially when quoting for spaces)
117     echo -n "'${k}' "
118     done;
119 }
120
121
122 # ---------------------------------------------------------------------
123
124 # Now, let's test it.
125 # (Per comments at the beginning of the script.)
126 newhash Lovers
127 addhash Lovers Tristan Isolde
128 addhash Lovers 'Romeo Montague' 'Juliet Capulet'
129
130 # Output results.
131 echo
132 gethash Lovers Tristan      # Isolde
133 echo
134 keyshash Lovers             # 'Tristan' 'Romeo Montague'
135 echo; echo
136
137
138 exit 0
139
140 # Exercise: Add error checks to the functions.
```

下面这个脚本可以用来安装和挂载那些小的USB keychain"硬件设备"(译者: 就是U盘一类的东西).

例子 A-24. 挂载USB keychain型的存储设备

```
  1 #!/bin/bash
  2 # ==> usb.sh
  3 # ==> Script for mounting and installing pen/keychain USB storage devices.
  4 # ==> Runs as root at system startup (see below).
  5 # ==>
  6 # ==> Newer Linux distros (2004 or later) autodetect
  7 # ==> and install USB pen drives, and therefore don't need this script.
  8 # ==> But, it's still instructive.
  9
 10 #  This code is free software covered by GNU GPL license version 2 or above.
 11 #  Please refer to http://www.gnu.org/ for the full license text.
 12 #
 13 #  Some code lifted from usb-mount by Michael Hamilton's usb-mount (LGPL)
 14 #+ see http://users.actrix.co.nz/michael/usbmount.html
 15 #
 16 #  INSTALL
 17 #  -------
 18 #  Put this in /etc/hotplug/usb/diskonkey.
 19 #  Then look in /etc/hotplug/usb.distmap, and copy all usb-storage entries
 20 #+ into /etc/hotplug/usb.usermap, substituting "usb-storage" for "diskonkey".
 21 #  Otherwise this code is only run during the kernel module invocation/removal
 22 #+ (at least in my tests), which defeats the purpose.
 23 #
 24 #  TODO
```

```
#  ----
#  Handle more than one diskonkey device at one time (e.g. /dev/diskonkey1
#+ and /mnt/diskonkey1), etc. The biggest problem here is the handling in
#+ devlabel, which I haven't yet tried.
#
#  AUTHOR and SUPPORT
#  ------------------
#  Konstantin Riabitsev, <icon linux duke edu>.
#  Send any problem reports to my email address at the moment.
#
# ==> Comments added by ABS Guide author.



SYMLINKDEV=/dev/diskonkey
MOUNTPOINT=/mnt/diskonkey
DEVLABEL=/sbin/devlabel
DEVLABELCONFIG=/etc/sysconfig/devlabel
IAM=$0

##
# Functions lifted near-verbatim from usb-mount code.
#
function allAttachedScsiUsb {
    find /proc/scsi/ -path '/proc/scsi/usb-storage*' -type f | xargs grep -l 'Attached: Yes'
}
function scsiDevFromScsiUsb {
    echo $1 | awk -F"[-/]" '{ n=$(NF-1);  print "/dev/sd" substr("abcdefghijklmnopqrstuvwxyz",
n+1,
 1) }'
}

if [ "${ACTION}" = "add" ] && [ -f "${DEVICE}" ]; then
    ##
    # lifted from usbcam code.
    #
    if [ -f /var/run/console.lock ]; then
        CONSOLEOWNER=`cat /var/run/console.lock`
    elif [ -f /var/lock/console.lock ]; then
        CONSOLEOWNER=`cat /var/lock/console.lock`
    else
        CONSOLEOWNER=
    fi
    for procEntry in $(allAttachedScsiUsb); do
        scsiDev=$(scsiDevFromScsiUsb $procEntry)
        #  Some bug with usb-storage?
        #  Partitions are not in /proc/partitions until they are accessed
        #+ somehow.
        /sbin/fdisk -l $scsiDev >/dev/null
        ##
        #  Most devices have partitioning info, so the data would be on
        #+ /dev/sd?1. However, some stupider ones don't have any partitioning
        #+ and use the entire device for data storage. This tries to
        #+ guess semi-intelligently if we have a /dev/sd?1 and if not, then
        #+ it uses the entire device and hopes for the better.
        #
        if grep -q `basename $scsiDev`1 /proc/partitions; then
            part="$scsiDev""1"
        else
            part=$scsiDev
        fi
        ##
        #  Change ownership of the partition to the console user so they can
        #+ mount it.
        #
        if [ ! -z "$CONSOLEOWNER" ]; then
            chown $CONSOLEOWNER:disk $part
        fi
        ##
        # This checks if we already have this UUID defined with devlabel.
        # If not, it then adds the device to the list.
        #
        prodid=`$DEVLABEL printid -d $part`
        if ! grep -q $prodid $DEVLABELCONFIG; then
            # cross our fingers and hope it works
            $DEVLABEL add -d $part -s $SYMLINKDEV 2>/dev/null
        fi
        ##
        # Check if the mount point exists and create if it doesn't.
        #
        if [ ! -e $MOUNTPOINT ]; then
            mkdir -p $MOUNTPOINT
        fi
        ##
        # Take care of /etc/fstab so mounting is easy.
        #
        if ! grep -q "^$SYMLINKDEV" /etc/fstab; then
            # Add an fstab entry
            echo -e \
                "$SYMLINKDEV\t\t$MOUNTPOINT\t\tauto\tnoauto,owner,kudzu 0 0" \
                >> /etc/fstab
        fi
    done
    if [ ! -z "$REMOVER" ]; then
        ##
        # Make sure this script is triggered on device removal.
        #
        mkdir -p `dirname $REMOVER`
        ln -s $IAM $REMOVER
    fi
elif [ "${ACTION}" = "remove" ]; then
    ##
```

```
126     # If the device is mounted, unmount it cleanly.
127     #
128     if grep -q "$MOUNTPOINT" /etc/mtab; then
129         # unmount cleanly
130         umount -l $MOUNTPOINT
131     fi
132     ##
133     # Remove it from /etc/fstab if it's there.
134     #
135     if grep -q "^$SYMLINKDEV" /etc/fstab; then
136         grep -v "^$SYMLINKDEV" /etc/fstab > /etc/.fstab.new
137         mv -f /etc/.fstab.new /etc/fstab
138     fi
139 fi
140
141 exit 0
```

这个脚本对于站点管理员来说很有用：这是一个可以保存weblog的脚本.

例子 A-25. 保存weblog

```
  1 #!/bin/bash
  2 # archiveweblogs.sh v1.0
  3
  4 # Troy Engel <tengel@fluid.com>
  5 # Slightly modified by document author.
  6 # Used with permission.
  7 #
  8 #  This script will preserve the normally rotated and
  9 #+ thrown away weblogs from a default RedHat/Apache installation.
 10 #  It will save the files with a date/time stamp in the filename,
 11 #+ bzipped, to a given directory.
 12 #
 13 #  Run this from crontab nightly at an off hour,
 14 #+ as bzip2 can suck up some serious CPU on huge logs:
 15 #  0 2 * * * /opt/sbin/archiveweblogs.sh
 16
 17
 18 PROBLEM=66
 19
 20 # Set this to your backup dir.
 21 BKP_DIR=/opt/backups/weblogs
 22
 23 # Default Apache/RedHat stuff
 24 LOG_DAYS="4 3 2 1"
 25 LOG_DIR=/var/log/httpd
 26 LOG_FILES="access_log error_log"
 27
 28 # Default RedHat program locations
 29 LS=/bin/ls
 30 MV=/bin/mv
 31 ID=/usr/bin/id
 32 CUT=/bin/cut
 33 COL=/usr/bin/column
 34 BZ2=/usr/bin/bzip2
 35
 36 # Are we root?
 37 USER=`$ID -u`
 38 if [ "X$USER" != "X0" ]; then
 39   echo "PANIC: Only root can run this script!"
 40   exit $PROBLEM
 41 fi
 42
 43 # Backup dir exists/writable?
 44 if [ ! -x $BKP_DIR ]; then
 45   echo "PANIC: $BKP_DIR doesn't exist or isn't writable!"
 46   exit $PROBLEM
 47 fi
 48
 49 # Move, rename and bzip2 the logs
 50 for logday in $LOG_DAYS; do
 51   for logfile in $LOG_FILES; do
 52     MYFILE="$LOG_DIR/$logfile.$logday"
 53     if [ -w $MYFILE ]; then
 54       DTS=`$LS -lgo --time-style=+%Y%m%d $MYFILE | $COL -t | $CUT -d ' ' -f7`
 55       $MV $MYFILE $BKP_DIR/$logfile.$DTS
 56       $BZ2 $BKP_DIR/$logfile.$DTS
 57     else
 58       # Only spew an error if the file exits (ergo non-writable).
 59       if [ -f $MYFILE ]; then
 60         echo "ERROR: $MYFILE not writable. Skipping."
 61       fi
 62     fi
 63   done
 64 done
 65
 66 exit 0
```

你怎么做才能阻止shell扩展或者重新解释字符串？

例子 A-26. 保护字符串的字面含义

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

```
  2 # protect_literal.sh
  3
  4 # set -vx
  5
  6 :<<-'_Protect_Literal_String_Doc'
  7
  8     Copyright (c) Michael S. Zick, 2003; All Rights Reserved
  9     License: Unrestricted reuse in any form, for any purpose.
 10     Warranty: None
 11     Revision: $ID$
 12
 13     Documentation redirected to the Bash no-operation.
 14     Bash will '/dev/null' this block when the script is first read.
 15     (Uncomment the above set command to see this action.)
 16
 17     Remove the first (Sha-Bang) line when sourcing this as a library
 18     procedure.  Also comment out the example use code in the two
 19     places where shown.
 20
 21
 22     Usage:
 23         _protect_literal_str 'Whatever string meets your ${fancy}'
 24         Just echos the argument to standard out, hard quotes
 25         restored.
 26
 27         $(_protect_literal_str 'Whatever string meets your ${fancy}')
 28         as the right-hand-side of an assignment statement.
 29
 30     Does:
 31         As the right-hand-side of an assignment, preserves the
 32         hard quotes protecting the contents of the literal during
 33         assignment.
 34
 35     Notes:
 36         The strange names (_*) are used to avoid trampling on
 37         the user's chosen names when this is sourced as a
 38         library.
 39
 40 _Protect_Literal_String_Doc
 41
 42 # The 'for illustration' function form
 43
 44 _protect_literal_str() {
 45
 46 # Pick an un-used, non-printing character as local IFS.
 47 # Not required, but shows that we are ignoring it.
 48     local IFS=$'\x1B'               # \ESC character
 49
 50 # Enclose the All-Elements-Of in hard quotes during assignment.
 51     local tmp=$'\x27'$@$'\x27'
 52 #    local tmp=$'\''$@$'\''         # Even uglier.
 53
 54     local len=${#tmp}               # Info only.
 55     echo $tmp is $len long.         # Output AND information.
 56 }
 57
 58 # This is the short-named version.
 59 _pls() {
 60     local IFS=$'x1B'                # \ESC character (not required)
 61     echo $'\x27'$@$'\x27'           # Hard quoted parameter glob
 62 }
 63
 64 # :<<-'_Protect_Literal_String_Test'
 65 # # # Remove the above "# " to disable this code. # # #
 66
 67 # See how that looks when printed.
 68 echo
 69 echo "- - Test One - -"
 70 _protect_literal_str 'Hello $user'
 71 _protect_literal_str 'Hello "${username}"'
 72 echo
 73
 74 # Which yields:
 75 # - - Test One - -
 76 # 'Hello $user' is 13 long.
 77 # 'Hello "${username}"' is 21 long.
 78
 79 #  Looks as expected, but why all of the trouble?
 80 #  The difference is hidden inside the Bash internal order
 81 #+ of operations.
 82 #  Which shows when you use it on the RHS of an assignment.
 83
 84 # Declare an array for test values.
 85 declare -a arrayZ
 86
 87 # Assign elements with various types of quotes and escapes.
 88 arrayZ=( zero "$(_pls 'Hello ${Me}')" 'Hello ${You}' "\'Pass: ${pw}\'" )
 89
 90 # Now list that array and see what is there.
 91 echo "- - Test Two - -"
 92 for (( i=0 ; i<${#arrayZ[*]} ; i++ ))
 93 do
 94     echo  Element $i: ${arrayZ[$i]} is: ${#arrayZ[$i]} long.
 95 done
 96 echo
 97
 98 # Which yields:
 99 # - - Test Two - -
100 # Element 0: zero is: 4 long.           # Our marker element
101 # Element 1: 'Hello ${Me}' is: 13 long. # Our "$(_pls '...' )"
102 # Element 2: Hello ${You} is: 12 long.  # Quotes are missing
103 # Element 3: \'Pass: \' is: 10 long.    # ${pw} expanded to nothing
```

```
104
105 # Now make an assignment with that result.
106 declare -a array2=( ${arrayZ[@]} )
107
108 # And print what happened.
109 echo "- - Test Three - -"
110 for (( i=0 ; i<${#array2[*]} ; i++ ))
111 do
112     echo  Element $i: ${array2[$i]} is: ${#array2[$i]} long.
113 done
114 echo
115
116 # Which yields:
117 # - - Test Three - -
118 # Element 0: zero is: 4 long.            # Our marker element.
119 # Element 1: Hello ${Me} is: 11 long.    # Intended result.
120 # Element 2: Hello is: 5 long.           # ${You} expanded to nothing.
121 # Element 3: 'Pass: is: 6 long.          # Split on the whitespace.
122 # Element 4: ' is: 1 long.               # The end quote is here now.
123
124 #  Our Element 1 has had its leading and trailing hard quotes stripped.
125 #  Although not shown, leading and trailing whitespace is also stripped.
126 #  Now that the string contents are set, Bash will always, internally,
127 #+ hard quote the contents as required during its operations.
128
129 #  Why?
130 #  Considering our "$(_pls 'Hello ${Me}')" construction:
131 #  " ... " -> Expansion required, strip the quotes.
132 #  $( ... ) -> Replace with the result of..., strip this.
133 #  _pls ' ... ' -> called with literal arguments, strip the quotes.
134 #  The result returned includes hard quotes; BUT the above processing
135 #+ has already been done, so they become part of the value assigned.
136 #
137 #  Similarly, during further usage of the string variable, the ${Me}
138 #+ is part of the contents (result) and survives any operations
139 #  (Until explicitly told to evaluate the string).
140
141 #  Hint: See what happens when the hard quotes ($'\x27') are replaced
142 #+ with soft quotes ($'\x22') in the above procedures.
143 #  Interesting also is to remove the addition of any quoting.
144
145 # _Protect_Literal_String_Test
146 # # # Remove the above "# " to disable this code. # # #
147
148 exit 0
```

如果你*确实想让*shell扩展或者重新解释字符串的话, 该怎么办?

例子 A-27. 不保护字符串的字面含义

```
  1 #! /bin/bash
  2 # unprotect_literal.sh
  3
  4 # set -vx
  5
  6 :<<-'_UnProtect_Literal_String_Doc'
  7
  8     Copyright (c) Michael S. Zick, 2003; All Rights Reserved
  9     License: Unrestricted reuse in any form, for any purpose.
 10     Warranty: None
 11     Revision: $ID$
 12
 13     Documentation redirected to the Bash no-operation. Bash will
 14     '/dev/null' this block when the script is first read.
 15     (Uncomment the above set command to see this action.)
 16
 17     Remove the first (Sha-Bang) line when sourcing this as a library
 18     procedure.  Also comment out the example use code in the two
 19     places where shown.
 20
 21
 22     Usage:
 23         Complement of the "$(_pls 'Literal String')" function.
 24         (See the protect_literal.sh example.)
 25
 26         StringVar=$(_upls ProtectedSringVariable)
 27
 28     Does:
 29         When used on the right-hand-side of an assignment statement;
 30         makes the substitions embedded in the protected string.
 31
 32     Notes:
 33         The strange names (_*) are used to avoid trampling on
 34         the user's chosen names when this is sourced as a
 35         library.
 36
 37
 38 _UnProtect_Literal_String_Doc
 39
 40 _upls() {
 41     local IFS=$'x1B'                # \ESC character (not required)
 42     eval echo $@                    # Substitution on the glob.
 43 }
 44
 45 # :<<-'_UnProtect_Literal_String_Test'
 46 # # # Remove the above "# " to disable this code. # # #
 47
```

```
 48
 49 _pls() {
 50     local IFS=$'x1B'               # \ESC character (not required)
 51     echo $'\x27'$@$'\x27'          # Hard quoted parameter glob
 52 }
 53
 54 # Declare an array for test values.
 55 declare -a arrayZ
 56
 57 # Assign elements with various types of quotes and escapes.
 58 arrayZ=( zero "$(_pls 'Hello ${Me}')" 'Hello ${You}' "\'Pass: ${pw}\'" )
 59
 60 # Now make an assignment with that result.
 61 declare -a array2=( ${arrayZ[@]} )
 62
 63 # Which yielded:
 64 # - - Test Three - -
 65 # Element 0: zero is: 4 long            # Our marker element.
 66 # Element 1: Hello ${Me} is: 11 long    # Intended result.
 67 # Element 2: Hello is: 5 long           # ${You} expanded to nothing.
 68 # Element 3: 'Pass: is: 6 long          # Split on the whitespace.
 69 # Element 4: ' is: 1 long               # The end quote is here now.
 70
 71 # set -vx
 72
 73 #  Initialize 'Me' to something for the embedded ${Me} substitution.
 74 #  This needs to be done ONLY just prior to evaluating the
 75 #+ protected string.
 76 #  (This is why it was protected to begin with.)
 77
 78 Me="to the array guy."
 79
 80 # Set a string variable destination to the result.
 81 newVar=$(_upls ${array2[1]})
 82
 83 # Show what the contents are.
 84 echo $newVar
 85
 86 # Do we really need a function to do this?
 87 newerVar=$(eval echo ${array2[1]})
 88 echo $newerVar
 89
 90 #  I guess not, but the _upls function gives us a place to hang
 91 #+ the documentation on.
 92 #  This helps when we forget what a # construction like:
 93 #+ $(eval echo ... ) means.
 94
 95 # What if Me isn't set when the protected string is evaluated?
 96 unset Me
 97 newestVar=$(_upls ${array2[1]})
 98 echo $newestVar
 99
100 # Just gone, no hints, no runs, no errors.
101
102 #  Why in the world?
103 #  Setting the contents of a string variable containing character
104 #+ sequences that have a meaning in Bash is a general problem in
105 #+ script programming.
106 #
107 #  This problem is now solved in eight lines of code
108 #+ (and four pages of description).
109
110 #  Where is all this going?
111 #  Dynamic content Web pages as an array of Bash strings.
112 #  Content set per request by a Bash 'eval' command
113 #+ on the stored page template.
114 #  Not intended to replace PHP, just an interesting thing to do.
115 ###
116 #  Don't have a webserver application?
117 #  No problem, check the example directory of the Bash source;
118 #+ there is a Bash script for that also.
119
120 # _UnProtect_Literal_String_Test
121 # # # Remove the above "# " to disable this code. # # #
122
123 exit 0
```

这个强大的脚本帮助我们抓住垃圾邮件服务器.

例子 A-28. 鉴定是否是垃圾邮件服务器

```
  1 #!/bin/bash
  2
  3 # $Id: is_spammer.bash,v 1.12.2.11 2004/10/01 21:42:33 mszick Exp $
  4 # Above line is RCS info.
  5
  6 # The latest version of this script is available from http://www.morethan.org.
  7 #
  8 # Spammer-identification
  9 # by Michael S. Zick
 10 # Used in the ABS Guide with permission.
 11
 12
 13
 14 #############################################################
 15 # Documentation
 16 # See also "Quickstart" at end of script.
```

```
 17 ########################################################
 18
 19 :<<-'__is_spammer_Doc_'
 20
 21     Copyright (c) Michael S. Zick, 2004
 22     License: Unrestricted reuse in any form, for any purpose.
 23     Warranty: None -{Its a script; the user is on their own.}-
 24
 25 Impatient?
 26     Application code: goto "# # # Hunt the Spammer' program code # # #"
 27     Example output: ":<<-'_is_spammer_outputs_'"
 28     How to use: Enter script name without arguments.
 29                 Or goto "Quickstart" at end of script.
 30
 31 Provides
 32     Given a domain name or IP(v4) address as input:
 33
 34     Does an exhaustive set of queries to find the associated
 35     network resources (short of recursing into TLDs).
 36
 37     Checks the IP(v4) addresses found against Blacklist
 38     nameservers.
 39
 40     If found to be a blacklisted IP(v4) address,
 41     reports the blacklist text records.
 42     (Usually hyper-links to the specific report.)
 43
 44 Requires
 45     A working Internet connection.
 46     (Exercise: Add check and/or abort if not on-line when running script.)
 47     Bash with arrays (2.05b+).
 48
 49     The external program 'dig' --
 50     a utility program provided with the 'bind' set of programs.
 51     Specifically, the version which is part of Bind series 9.x
 52     See: http://www.isc.org
 53
 54     All usages of 'dig' are limited to wrapper functions,
 55     which may be rewritten as required.
 56     See: dig_wrappers.bash for details.
 57          ("Additional documentation" -- below)
 58
 59 Usage
 60     Script requires a single argument, which may be:
 61     1) A domain name;
 62     2) An IP(v4) address;
 63     3) A filename, with one name or address per line.
 64
 65     Script accepts an optional second argument, which may be:
 66     1) A Blacklist server name;
 67     2) A filename, with one Blacklist server name per line.
 68
 69     If the second argument is not provided, the script uses
 70     a built-in set of (free) Blacklist servers.
 71
 72     See also, the Quickstart at the end of this script (after 'exit').
 73
 74 Return Codes
 75     0 - All OK
 76     1 - Script failure
 77     2 - Something is Blacklisted
 78
 79 Optional environment variables
 80     SPAMMER_TRACE
 81         If set to a writable file,
 82         script will log an execution flow trace.
 83
 84     SPAMMER_DATA
 85         If set to a writable file, script will dump its
 86         discovered data in the form of GraphViz file.
 87         See: http://www.research.att.com/sw/tools/graphviz
 88
 89     SPAMMER_LIMIT
 90         Limits the depth of resource tracing.
 91
 92         Default is 2 levels.
 93
 94         A setting of 0 (zero) means 'unlimited' . . .
 95           Caution: script might recurse the whole Internet!
 96
 97         A limit of 1 or 2 is most useful when processing
 98         a file of domain names and addresses.
 99         A higher limit can be useful when hunting spam gangs.
100
101
102 Additional documentation
103     Download the archived set of scripts
104     explaining and illustrating the function contained within this script.
105     http://personal.riverusers.com/mszick_clf.tar.bz2
106
107
108 Study notes
109     This script uses a large number of functions.
110     Nearly all general functions have their own example script.
111     Each of the example scripts have tutorial level comments.
112
113 Scripting project
114     Add support for IP(v6) addresses.
115     IP(v6) addresses are recognized but not processed.
116
117 Advanced project
118     Add the reverse lookup detail to the discovered information.
```

```
119
120     Report the delegation chain and abuse contacts.
121
122     Modify the GraphViz file output to include the
123     newly discovered information.
124
125 __is_spammer_Doc_
126
127 ###########################################################
128
129
130
131
132 #### Special IFS settings used for string parsing. ####
133
134 # Whitespace == :Space:Tab:Line Feed:Carriage Return:
135 WSP_IFS=$'\x20'$'\x09'$'\x0A'$'\x0D'
136
137 # No Whitespace == Line Feed:Carriage Return
138 NO_WSP=$'\x0A'$'\x0D'
139
140 # Field separator for dotted decimal IP addresses
141 ADR_IFS=${NO_WSP}'.'
142
143 # Array to dotted string conversions
144 DOT_IFS='.'${WSP_IFS}
145
146 # # # Pending operations stack machine # # #
147 # This set of functions described in func_stack.bash.
148 # (See "Additional documentation" above.)
149 # # #
150
151 # Global stack of pending operations.
152 declare -f -a _pending_
153 # Global sentinel for stack runners
154 declare -i _p_ctrl_
155 # Global holder for currently executing function
156 declare -f _pend_current_
157
158 # # # Debug version only - remove for regular use # # #
159 #
160 # The function stored in _pend_hook_ is called
161 # immediately before each pending function is
162 # evaluated.  Stack clean, _pend_current_ set.
163 #
164 # This thingy demonstrated in pend_hook.bash.
165 declare -f _pend_hook_
166 # # #
167
168 # The do nothing function
169 pend_dummy() { : ; }
170
171 # Clear and initialize the function stack.
172 pend_init() {
173     unset _pending_[@]
174     pend_func pend_stop_mark
175     _pend_hook_='pend_dummy'  # Debug only.
176 }
177
178 # Discard the top function on the stack.
179 pend_pop() {
180     if [ ${#_pending_[@]} -gt 0 ]
181     then
182         local -i _top_
183         _top_=${#_pending_[@]}-1
184         unset _pending_[$_top_]
185     fi
186 }
187
188 # pend_func function_name [$(printf '%q\n' arguments)]
189 pend_func() {
190     local IFS=${NO_WSP}
191     set -f
192     _pending_[${#_pending_[@]}]=$@
193     set +f
194 }
195
196 # The function which stops the release:
197 pend_stop_mark() {
198     _p_ctrl_=0
199 }
200
201 pend_mark() {
202     pend_func pend_stop_mark
203 }
204
205 # Execute functions until 'pend_stop_mark' . . .
206 pend_release() {
207     local -i _top_              # Declare _top_ as integer.
208     _p_ctrl_=${#_pending_[@]}
209     while [ ${_p_ctrl_} -gt 0 ]
210     do
211        _top_=${#_pending_[@]}-1
212        _pend_current_=${_pending_[$_top_]}
213        unset _pending_[$_top_]
214        $_pend_hook_             # Debug only.
215        eval $_pend_current_
216     done
217 }
218
219 # Drop functions until 'pend_stop_mark' . . .
220 pend_drop() {
```

```
    local -i _top_
    local _pd_ctrl_=${#_pending_[@]}
    while [ ${_pd_ctrl_} -gt 0 ]
    do
       _top_=$_pd_ctrl_-1
       if [ "${_pending_[$_top_]}" == 'pend_stop_mark' ]
       then
           unset _pending_[$_top_]
           break
       else
           unset _pending_[$_top_]
           _pd_ctrl_=$_top_
       fi
    done
    if [ ${#_pending_[@]} -eq 0 ]
    then
        pend_func pend_stop_mark
    fi
}

#### Array editors ####

# This function described in edit_exact.bash.
# (See "Additional documentation," above.)
# edit_exact <excludes_array_name> <target_array_name>
edit_exact() {
    [ $# -eq 2 ] ||
    [ $# -eq 3 ] || return 1
    local -a _ee_Excludes
    local -a _ee_Target
    local _ee_x
    local _ee_t
    local IFS=${NO_WSP}
    set -f
    eval _ee_Excludes=\( \$\{$1\[@\]\} \)
    eval _ee_Target=\( \$\{$2\[@\]\} \)
    local _ee_len=${#_ee_Target[@]}     # Original length.
    local _ee_cnt=${#_ee_Excludes[@]}   # Exclude list length.
    [ ${_ee_len} -ne 0 ] || return 0    # Can't edit zero length.
    [ ${_ee_cnt} -ne 0 ] || return 0    # Can't edit zero length.
    for (( x = 0; x < ${_ee_cnt} ; x++ ))
    do
        _ee_x=${_ee_Excludes[$x]}
        for (( n = 0 ; n < ${_ee_len} ; n++ ))
        do
            _ee_t=${_ee_Target[$n]}
            if [ x"${_ee_t}" == x"${_ee_x}" ]
            then
                unset _ee_Target[$n]     # Discard match.
                [ $# -eq 2 ] && break    # If 2 arguments, then done.
            fi
        done
    done
    eval $2=\( \$\{_ee_Target\[@\]\} \)
    set +f
    return 0
}

# This function described in edit_by_glob.bash.
# edit_by_glob <excludes_array_name> <target_array_name>
edit_by_glob() {
    [ $# -eq 2 ] ||
    [ $# -eq 3 ] || return 1
    local -a _ebg_Excludes
    local -a _ebg_Target
    local _ebg_x
    local _ebg_t
    local IFS=${NO_WSP}
    set -f
    eval _ebg_Excludes=\( \$\{$1\[@\]\} \)
    eval _ebg_Target=\( \$\{$2\[@\]\} \)
    local _ebg_len=${#_ebg_Target[@]}
    local _ebg_cnt=${#_ebg_Excludes[@]}
    [ ${_ebg_len} -ne 0 ] || return 0
    [ ${_ebg_cnt} -ne 0 ] || return 0
    for (( x = 0; x < ${_ebg_cnt} ; x++ ))
    do
        _ebg_x=${_ebg_Excludes[$x]}
        for (( n = 0 ; n < ${_ebg_len} ; n++ ))
        do
            [ $# -eq 3 ] && _ebg_x=${_ebg_x}'*'  #  Do prefix edit
            if [ ${_ebg_Target[$n]:=} ]          #+ if defined & set.
            then
                _ebg_t=${_ebg_Target[$n]/#${_ebg_x}/}
                [ ${#_ebg_t} -eq 0 ] && unset _ebg_Target[$n]
            fi
        done
    done
    eval $2=\( \$\{_ebg_Target\[@\]\} \)
    set +f
    return 0
}

# This function described in unique_lines.bash.
# unique_lines <in_name> <out_name>
unique_lines() {
    [ $# -eq 2 ] || return 1
    local -a _ul_in
    local -a _ul_out
    local -i _ul_cnt
    local -i _ul_pos
    local _ul_tmp
```

```
    local IFS=${NO_WSP}
    set -f
    eval _ul_in=\( \$\{$1\[@\]\} \)
    _ul_cnt=${#_ul_in[@]}
    for (( _ul_pos = 0 ; _ul_pos < ${_ul_cnt} ; _ul_pos++ ))
    do
        if [ ${_ul_in[${_ul_pos}]:=} ]      # If defined & not empty
        then
            _ul_tmp=${_ul_in[${_ul_pos}]}
            _ul_out[${#_ul_out[@]}]=${_ul_tmp}
            for (( zap = _ul_pos ; zap < ${_ul_cnt} ; zap++ ))
            do
                [ ${_ul_in[${zap}]:=} ] &&
                [ 'x'${_ul_in[${zap}]} == 'x'${_ul_tmp} ] &&
                    unset _ul_in[${zap}]
            done
        fi
    done
    eval $2=\( \$\{_ul_out\[@\]\} \)
    set +f
    return 0
}

# This function described in char_convert.bash.
# to_lower <string>
to_lower() {
    [ $# -eq 1 ] || return 1
    local _tl_out
    _tl_out=${1//A/a}
    _tl_out=${_tl_out//B/b}
    _tl_out=${_tl_out//C/c}
    _tl_out=${_tl_out//D/d}
    _tl_out=${_tl_out//E/e}
    _tl_out=${_tl_out//F/f}
    _tl_out=${_tl_out//G/g}
    _tl_out=${_tl_out//H/h}
    _tl_out=${_tl_out//I/i}
    _tl_out=${_tl_out//J/j}
    _tl_out=${_tl_out//K/k}
    _tl_out=${_tl_out//L/l}
    _tl_out=${_tl_out//M/m}
    _tl_out=${_tl_out//N/n}
    _tl_out=${_tl_out//O/o}
    _tl_out=${_tl_out//P/p}
    _tl_out=${_tl_out//Q/q}
    _tl_out=${_tl_out//R/r}
    _tl_out=${_tl_out//S/s}
    _tl_out=${_tl_out//T/t}
    _tl_out=${_tl_out//U/u}
    _tl_out=${_tl_out//V/v}
    _tl_out=${_tl_out//W/w}
    _tl_out=${_tl_out//X/x}
    _tl_out=${_tl_out//Y/y}
    _tl_out=${_tl_out//Z/z}
    echo ${_tl_out}
    return 0
}

#### Application helper functions ####

# Not everybody uses dots as separators (APNIC, for example).
# This function described in to_dot.bash
# to_dot <string>
to_dot() {
    [ $# -eq 1 ] || return 1
    echo ${1//[#|@|%]/.}
    return 0
}

# This function described in is_number.bash.
# is_number <input>
is_number() {
    [ "$#" -eq 1 ]    || return 1  # is blank?
    [ x"$1" == 'x0' ] && return 0  # is zero?
    local -i tst
    let tst=$1 2>/dev/null         # else is numeric!
    return $?
}

# This function described in is_address.bash.
# is_address <input>
is_address() {
    [ $# -eq 1 ] || return 1    # Blank ==> false
    local -a _ia_input
    local IFS=${ADR_IFS}
    _ia_input=( $1 )
    if  [ ${#_ia_input[@]} -eq 4 ]  &&
        is_number ${_ia_input[0]}   &&
        is_number ${_ia_input[1]}   &&
        is_number ${_ia_input[2]}   &&
        is_number ${_ia_input[3]}   &&
        [ ${_ia_input[0]} -lt 256 ] &&
        [ ${_ia_input[1]} -lt 256 ] &&
        [ ${_ia_input[2]} -lt 256 ] &&
        [ ${_ia_input[3]} -lt 256 ]
    then
        return 0
    else
        return 1
    fi
}

```

```
# This function described in split_ip.bash.
# split_ip <IP_address> <array_name_norm> [<array_name_rev>]
split_ip() {
    [ $# -eq 3 ] ||                #  Either three
    [ $# -eq 2 ] || return 1     #+ or two arguments
    local -a _si_input
    local IFS=${ADR_IFS}
    _si_input=( $1 )
    IFS=${WSP_IFS}
    eval $2=\(\ \$\{_si_input\[@\]\}\ \)
    if [ $# -eq 3 ]
    then
        # Build query order array.
        local -a _dns_ip
        _dns_ip[0]=${_si_input[3]}
        _dns_ip[1]=${_si_input[2]}
        _dns_ip[2]=${_si_input[1]}
        _dns_ip[3]=${_si_input[0]}
        eval $3=\(\ \$\{_dns_ip\[@\]\}\ \)
    fi
    return 0
}

# This function described in dot_array.bash.
# dot_array <array_name>
dot_array() {
    [ $# -eq 1 ] || return 1     # Single argument required.
    local -a _da_input
    eval _da_input=\(\ \$\{$1\[@\]\}\ \)
    local IFS=${DOT_IFS}
    local _da_output=${_da_input[@]}
    IFS=${WSP_IFS}
    echo ${_da_output}
    return 0
}

# This function described in file_to_array.bash
# file_to_array <file_name> <line_array_name>
file_to_array() {
    [ $# -eq 2 ] || return 1  # Two arguments required.
    local IFS=${NO_WSP}
    local -a _fta_tmp_
    _fta_tmp_=( $(cat $1) )
    eval $2=\( \$\{_fta_tmp_\[@\]\} \)
    return 0
}

# Columnized print of an array of multi-field strings.
# col_print <array_name> <min_space> <tab_stop [tab_stops]>
col_print() {
    [ $# -gt 2 ] || return 0
    local -a _cp_inp
    local -a _cp_spc
    local -a _cp_line
    local _cp_min
    local _cp_mcnt
    local _cp_pos
    local _cp_cnt
    local _cp_tab
    local -i _cp
    local -i _cpf
    local _cp_fld
    # WARNING: FOLLOWING LINE NOT BLANK -- IT IS QUOTED SPACES.
    local _cp_max='                                                                    '
    set -f
    local IFS=${NO_WSP}
    eval _cp_inp=\(\ \$\{$1\[@\]\}\ \)
    [ ${#_cp_inp[@]} -gt 0 ] || return 0 # Empty is easy.
    _cp_mcnt=$2
    _cp_min=${_cp_max:1:${_cp_mcnt}}
    shift
    shift
    _cp_cnt=$#
    for (( _cp = 0 ; _cp < _cp_cnt ; _cp++ ))
    do
        _cp_spc[${#_cp_spc[@]}]="${_cp_max:2:$1}" #"
        shift
    done
    _cp_cnt=${#_cp_inp[@]}
    for (( _cp = 0 ; _cp < _cp_cnt ; _cp++ ))
    do
        _cp_pos=1
        IFS=${NO_WSP}$'\x20'
        _cp_line=( ${_cp_inp[${_cp}]} )
        IFS=${NO_WSP}
        for (( _cpf = 0 ; _cpf < ${#_cp_line[@]} ; _cpf++ ))
        do
            _cp_tab=${_cp_spc[${_cpf}]:${_cp_pos}}
            if [ ${#_cp_tab} -lt ${_cp_mcnt} ]
            then
                _cp_tab="${_cp_min}"
            fi
            echo -n "${_cp_tab}"
            (( _cp_pos = ${_cp_pos} + ${#_cp_tab} ))
            _cp_fld="${_cp_line[${_cpf}]}"
            echo -n ${_cp_fld}
            (( _cp_pos = ${_cp_pos} + ${#_cp_fld} ))
        done
        echo
    done
    set +f
    return 0
```

```
}

# # # # 'Hunt the Spammer' data flow # # # #

# Application return code
declare -i _hs_RC

# Original input, from which IP addresses are removed
# After which, domain names to check
declare -a uc_name

# Original input IP addresses are moved here
# After which, IP addresses to check
declare -a uc_address

# Names against which address expansion run
# Ready for name detail lookup
declare -a chk_name

# Addresses against which name expansion run
# Ready for address detail lookup
declare -a chk_address

#  Recursion is depth-first-by-name.
#  The expand_input_address maintains this list
#+ to prohibit looking up addresses twice during
#+ domain name recursion.
declare -a been_there_addr
been_there_addr=( '127.0.0.1' ) # Whitelist localhost

# Names which we have checked (or given up on)
declare -a known_name

# Addresses which we have checked (or given up on)
declare -a known_address

#  List of zero or more Blacklist servers to check.
#  Each 'known_address' will be checked against each server,
#+ with negative replies and failures suppressed.
declare -a list_server

# Indirection limit - set to zero == no limit
indirect=${SPAMMER_LIMIT:=2}

# # # # 'Hunt the Spammer' information output data # # # #

# Any domain name may have multiple IP addresses.
# Any IP address may have multiple domain names.
# Therefore, track unique address-name pairs.
declare -a known_pair
declare -a reverse_pair

#  In addition to the data flow variables; known_address
#+ known_name and list_server, the following are output to the
#+ external graphics interface file.

# Authority chain, parent -> SOA fields.
declare -a auth_chain

# Reference chain, parent name -> child name
declare -a ref_chain

# DNS chain - domain name -> address
declare -a name_address

# Name and service pairs - domain name -> service
declare -a name_srvc

# Name and resource pairs - domain name -> Resource Record
declare -a name_resource

# Parent and Child pairs - parent name -> child name
# This MAY NOT be the same as the ref_chain followed!
declare -a parent_child

# Address and Blacklist hit pairs - address->server
declare -a address_hits

# Dump interface file data
declare -f _dot_dump
_dot_dump=pend_dummy   # Initially a no-op

#  Data dump is enabled by setting the environment variable SPAMMER_DATA
#+ to the name of a writable file.
declare _dot_file

# Helper function for the dump-to-dot-file function
# dump_to_dot <array_name> <prefix>
dump_to_dot() {
    local -a _dda_tmp
    local -i _dda_cnt
    local _dda_form='    '${2}'%04u %s\n'
    local IFS=${NO_WSP}
    eval _dda_tmp=\(\ \$\{$1\[@\]\}\ \)
    _dda_cnt=${#_dda_tmp[@]}
    if [ ${_dda_cnt} -gt 0 ]
    then
        for (( _dda = 0 ; _dda < _dda_cnt ; _dda++ ))
        do
            printf "${_dda_form}" \
                   "${_dda}" "${_dda_tmp[${_dda}]}" >>${_dot_file}
        done
```

```
    fi
}

# Which will also set _dot_dump to this function . . .
dump_dot() {
    local -i _dd_cnt
    echo '# Data vintage: '$(date -R) >${_dot_file}
    echo '# ABS Guide: is_spammer.bash; v2, 2004-msz' >>${_dot_file}
    echo >>${_dot_file}
    echo 'digraph G {' >>${_dot_file}

    if [ ${#known_name[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Known domain name nodes' >>${_dot_file}
        _dd_cnt=${#known_name[@]}
        for (( _dd = 0 ; _dd < _dd_cnt ; _dd++ ))
        do
            printf '    N%04u [label="%s"] ;\n' \
                   "${_dd}" "${known_name[${_dd}]}" >>${_dot_file}
        done
    fi

    if [ ${#known_address[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Known address nodes' >>${_dot_file}
        _dd_cnt=${#known_address[@]}
        for (( _dd = 0 ; _dd < _dd_cnt ; _dd++ ))
        do
            printf '    A%04u [label="%s"] ;\n' \
                   "${_dd}" "${known_address[${_dd}]}" >>${_dot_file}
        done
    fi

    echo                                        >>${_dot_file}
    echo '/*'                                   >>${_dot_file}
    echo ' * Known relationships :: User conversion to'  >>${_dot_file}
    echo ' * graphic form by hand or program required.'  >>${_dot_file}
    echo ' *'                                   >>${_dot_file}

    if [ ${#auth_chain[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Authority reference edges followed and field source.'  >>${_dot_file}
        dump_to_dot auth_chain AC
    fi

    if [ ${#ref_chain[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Name reference edges followed and field source.'  >>${_dot_file}
        dump_to_dot ref_chain RC
    fi

    if [ ${#name_address[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Known name->address edges' >>${_dot_file}
        dump_to_dot name_address NA
    fi

    if [ ${#name_srvc[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Known name->service edges' >>${_dot_file}
        dump_to_dot name_srvc NS
    fi

    if [ ${#name_resource[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Known name->resource edges' >>${_dot_file}
        dump_to_dot name_resource NR
    fi

    if [ ${#parent_child[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Known parent->child edges' >>${_dot_file}
        dump_to_dot parent_child PC
    fi

    if [ ${#list_server[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Known Blacklist nodes' >>${_dot_file}
        _dd_cnt=${#list_server[@]}
        for (( _dd = 0 ; _dd < _dd_cnt ; _dd++ ))
        do
            printf '    LS%04u [label="%s"] ;\n' \
                   "${_dd}" "${list_server[${_dd}]}" >>${_dot_file}
        done
    fi

    unique_lines address_hits address_hits
    if [ ${#address_hits[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Known address->Blacklist_hit edges' >>${_dot_file}
        echo '# CAUTION: dig warnings can trigger false hits.' >>${_dot_file}
        dump_to_dot address_hits AH
```

```
    fi
    echo           >>${_dot_file}
    echo ' *'      >>${_dot_file}
    echo ' * That is a lot of relationships. Happy graphing.' >>${_dot_file}
    echo ' */'     >>${_dot_file}
    echo '}'       >>${_dot_file}
    return 0
}

# # # # 'Hunt the Spammer' execution flow # # # #

#  Execution trace is enabled by setting the
#+ environment variable SPAMMER_TRACE to the name of a writable file.
declare -a _trace_log
declare _log_file

# Function to fill the trace log
trace_logger() {
    _trace_log[${#_trace_log[@]}]=${_pend_current_}
}

# Dump trace log to file function variable.
declare -f _log_dump
_log_dump=pend_dummy   # Initially a no-op.

# Dump the trace log to a file.
dump_log() {
    local -i _dl_cnt
    _dl_cnt=${#_trace_log[@]}
    for (( _dl = 0 ; _dl < _dl_cnt ; _dl++ ))
    do
        echo ${_trace_log[${_dl}]} >> ${_log_file}
    done
    _dl_cnt=${#_pending_[@]}
    if [ ${_dl_cnt} -gt 0 ]
    then
        _dl_cnt=${_dl_cnt}-1
        echo '# # # Operations stack not empty # # #' >> ${_log_file}
        for (( _dl = ${_dl_cnt} ; _dl >= 0 ; _dl-- ))
        do
            echo ${_pending_[${_dl}]} >> ${_log_file}
        done
    fi
}

# # # Utility program 'dig' wrappers # # #
#
#  These wrappers are derived from the
#+ examples shown in dig_wrappers.bash.
#
#  The major difference is these return
#+ their results as a list in an array.
#
#  See dig_wrappers.bash for details and
#+ use that script to develop any changes.
#
# # #

# Short form answer: 'dig' parses answer.

# Forward lookup :: Name -> Address
# short_fwd <domain_name> <array_name>
short_fwd() {
    local -a _sf_reply
    local -i _sf_rc
    local -i _sf_cnt
    IFS=${NO_WSP}
echo -n '.'
# echo 'sfwd: '${1}
    _sf_reply=( $(dig +short ${1} -c in -t a 2>/dev/null) )
    _sf_rc=$?
    if [ ${_sf_rc} -ne 0 ]
    then
        _trace_log[${#_trace_log[@]}]='# # # Lookup error '${_sf_rc}' on '${1}' # # #'
# [ ${_sf_rc} -ne 9 ] && pend_drop
        return ${_sf_rc}
    else
        # Some versions of 'dig' return warnings on stdout.
        _sf_cnt=${#_sf_reply[@]}
        for (( _sf = 0 ; _sf < ${_sf_cnt} ; _sf++ ))
        do
            [ 'x'${_sf_reply[${_sf}]:0:2} == 'x;;' ] &&
                unset _sf_reply[${_sf}]
        done
        eval $2=\( \$\{_sf_reply\[@\]\} \)
    fi
    return 0
}

# Reverse lookup :: Address -> Name
# short_rev <ip_address> <array_name>
short_rev() {
    local -a _sr_reply
    local -i _sr_rc
    local -i _sr_cnt
    IFS=${NO_WSP}
echo -n '.'
# echo 'srev: '${1}
    _sr_reply=( $(dig +short -x ${1} 2>/dev/null) )
    _sr_rc=$?
    if [ ${_sr_rc} -ne 0 ]
    then
```

```
        _trace_log[${#_trace_log[@]}]='# # # Lookup error '${_sr_rc}' on '${1}' # # #'
# [ ${_sr_rc} -ne 9 ] && pend_drop
        return ${_sr_rc}
    else
        # Some versions of 'dig' return warnings on stdout.
        _sr_cnt=${#_sr_reply[@]}
        for (( _sr = 0 ; _sr < ${_sr_cnt} ; _sr++ ))
        do
            [ 'x'${_sr_reply[${_sr}]:0:2} == 'x;;' ] &&
                unset _sr_reply[${_sr}]
        done
        eval $2=\( \$\{_sr_reply\[@\]\} \)
    fi
    return 0
}

# Special format lookup used to query blacklist servers.
# short_text <ip_address> <array_name>
short_text() {
    local -a _st_reply
    local -i _st_rc
    local -i _st_cnt
    IFS=${NO_WSP}
# echo 'stxt: '${1}
    _st_reply=( $(dig +short ${1} -c in -t txt 2>/dev/null) )
    _st_rc=$?
    if [ ${_st_rc} -ne 0 ]
    then
        _trace_log[${#_trace_log[@]}]='# # # Text lookup error '${_st_rc}' on '${1}' # # #'
# [ ${_st_rc} -ne 9 ] && pend_drop
        return ${_st_rc}
    else
        # Some versions of 'dig' return warnings on stdout.
        _st_cnt=${#_st_reply[@]}
        for (( _st = 0 ; _st < ${#_st_cnt} ; _st++ ))
        do
            [ 'x'${_st_reply[${_st}]:0:2} == 'x;;' ] &&
                unset _st_reply[${_st}]
        done
        eval $2=\( \$\{_st_reply\[@\]\} \)
    fi
    return 0
}

# The long forms, a.k.a., the parse it yourself versions

# RFC 2782   Service lookups
# dig +noall +nofail +answer _ldap._tcp.openldap.org -t srv
# _<service>._<protocol>.<domain_name>
# _ldap._tcp.openldap.org. 3600   IN      SRV     0 0 389 ldap.openldap.org.
# domain TTL Class SRV Priority Weight Port Target

# Forward lookup :: Name -> poor man's zone transfer
# long_fwd <domain_name> <array_name>
long_fwd() {
    local -a _lf_reply
    local -i _lf_rc
    local -i _lf_cnt
    IFS=${NO_WSP}
echo -n ':'
# echo 'lfwd: '${1}
    _lf_reply=( $(
        dig +noall +nofail +answer +authority +additional \
            ${1} -t soa ${1} -t mx ${1} -t any 2>/dev/null) )
    _lf_rc=$?
    if [ ${_lf_rc} -ne 0 ]
    then
        _trace_log[${#_trace_log[@]}]='# # # Zone lookup error '${_lf_rc}' on '${1}' # # #'
# [ ${_lf_rc} -ne 9 ] && pend_drop
        return ${_lf_rc}
    else
        # Some versions of 'dig' return warnings on stdout.
        _lf_cnt=${#_lf_reply[@]}
        for (( _lf = 0 ; _lf < ${_lf_cnt} ; _lf++ ))
        do
            [ 'x'${_lf_reply[${_lf}]:0:2} == 'x;;' ] &&
                unset _lf_reply[${_lf}]
        done
        eval $2=\( \$\{_lf_reply\[@\]\} \)
    fi
    return 0
}
#   The reverse lookup domain name corresponding to the IPv6 address:
#       4321:0:1:2:3:4:567:89ab
#   would be (nibble, I.E: Hexdigit) reversed:
#   b.a.9.8.7.6.5.0.4.0.0.0.3.0.0.0.2.0.0.0.1.0.0.0.0.0.0.0.1.2.3.4.IP6.ARPA.

# Reverse lookup :: Address -> poor man's delegation chain
# long_rev <rev_ip_address> <array_name>
long_rev() {
    local -a _lr_reply
    local -i _lr_rc
    local -i _lr_cnt
    local _lr_dns
    _lr_dns=${1}'.in-addr.arpa.'
    IFS=${NO_WSP}
echo -n ':'
# echo 'lrev: '${1}
    _lr_reply=( $(
         dig +noall +nofail +answer +authority +additional \
             ${_lr_dns} -t soa ${_lr_dns} -t any 2>/dev/null) )
    _lr_rc=$?
```

```
    if [ ${_lr_rc} -ne 0 ]
    then
        _trace_log[${#_trace_log[@]}]='# # # Delegation lookup error '${_lr_rc}' on '${1}' # # #'
# [ ${_lr_rc} -ne 9 ] && pend_drop
        return ${_lr_rc}
    else
        # Some versions of 'dig' return warnings on stdout.
        _lr_cnt=${#_lr_reply[@]}
        for (( _lr = 0 ; _lr < ${_lr_cnt} ; _lr++ ))
        do
            [ 'x'${_lr_reply[${_lr}]:0:2} == 'x;;' ] &&
                unset _lr_reply[${_lr}]
        done
        eval $2=\( \$\{_lr_reply\[@\]\} \)
    fi
    return 0
}

# # # Application specific functions # # #

# Mung a possible name; suppresses root and TLDs.
# name_fixup <string>
name_fixup(){
    local -a _nf_tmp
    local -i _nf_end
    local _nf_str
    local IFS
    _nf_str=$(to_lower ${1})
    _nf_str=$(to_dot ${_nf_str})
    _nf_end=${#_nf_str}-1
    [ ${_nf_str:${_nf_end}} != '.' ] &&
        _nf_str=${_nf_str}'.'
    IFS=${ADR_IFS}
    _nf_tmp=( ${_nf_str} )
    IFS=${WSP_IFS}
    _nf_end=${#_nf_tmp[@]}
    case ${_nf_end} in
    0) # No dots, only dots.
        echo
        return 1
    ;;
    1) # Only a TLD.
        echo
        return 1
    ;;
    2) # Maybe okay.
       echo ${_nf_str}
       return 0
       # Needs a lookup table?
       if [ ${#_nf_tmp[1]} -eq 2 ]
       then # Country coded TLD.
           echo
           return 1
       else
           echo ${_nf_str}
           return 0
       fi
    ;;
    esac
    echo ${_nf_str}
    return 0
}

# Grope and mung original input(s).
split_input() {
    [ ${#uc_name[@]} -gt 0 ] || return 0
    local -i _si_cnt
    local -i _si_len
    local _si_str
    unique_lines uc_name uc_name
    _si_cnt=${#uc_name[@]}
    for (( _si = 0 ; _si < _si_cnt ; _si++ ))
    do
        _si_str=${uc_name[$_si]}
        if is_address ${_si_str}
        then
            uc_address[${#uc_address[@]}]=${_si_str}
            unset uc_name[$_si]
        else
            if ! uc_name[$_si]=$(name_fixup ${_si_str})
            then
                unset ucname[$_si]
            fi
        fi
    done
    uc_name=( ${uc_name[@]} )
    _si_cnt=${#uc_name[@]}
    _trace_log[${#_trace_log[@]}]='# # # Input '${_si_cnt}' unchecked name input(s). # # #'
    _si_cnt=${#uc_address[@]}
    _trace_log[${#_trace_log[@]}]='# # # Input '${_si_cnt}' unchecked address input(s). # # #'
    return 0
}

# # # Discovery functions -- recursively interlocked by external data # # #
# # # The leading 'if list is empty; return 0' in each is required. # # #

# Recursion limiter
# limit_chk() <next_level>
limit_chk() {
    local -i _lc_lmt
    # Check indirection limit.
    if [ ${indirect} -eq 0 ] || [ $# -eq 0 ]
```

```
    then
        # The 'do-forever' choice
        echo 1                # Any value will do.
        return 0              # OK to continue.
    else
        # Limiting is in effect.
        if [ ${indirect} -lt ${1} ]
        then
            echo ${1}         # Whatever.
            return 1          # Stop here.
        else
            _lc_lmt=${1}+1    # Bump the given limit.
            echo ${_lc_lmt}   # Echo it.
            return 0          # OK to continue.
        fi
    fi
}

# For each name in uc_name:
#     Move name to chk_name.
#     Add addresses to uc_address.
#     Pend expand_input_address.
#     Repeat until nothing new found.
# expand_input_name <indirection_limit>
expand_input_name() {
    [ ${#uc_name[@]} -gt 0 ] || return 0
    local -a _ein_addr
    local -a _ein_new
    local -i _ucn_cnt
    local -i _ein_cnt
    local _ein_tst
    _ucn_cnt=${#uc_name[@]}

    if  ! _ein_cnt=$(limit_chk ${1})
    then
        return 0
    fi

    for (( _ein = 0 ; _ein < _ucn_cnt ; _ein++ ))
    do
        if short_fwd ${uc_name[${_ein}]} _ein_new
        then
            for (( _ein_cnt = 0 ; _ein_cnt < ${#_ein_new[@]}; _ein_cnt++ ))
            do
                _ein_tst=${_ein_new[${_ein_cnt}]}
                if is_address ${_ein_tst}
                then
                    _ein_addr[${#_ein_addr[@]}]=${_ein_tst}
                fi
           done
        fi
    done
    unique_lines _ein_addr _ein_addr     # Scrub duplicates.
    edit_exact chk_address _ein_addr     # Scrub pending detail.
    edit_exact known_address _ein_addr   # Scrub already detailed.
    if [ ${#_ein_addr[@]} -gt 0 ]        # Anything new?
    then
        uc_address=( ${uc_address[@]} ${_ein_addr[@]} )
        pend_func expand_input_address ${1}
        _trace_log[${#_trace_log[@]}]='# # # Added '${#_ein_addr[@]}' unchecked address input(s).
# # #'
    fi
    edit_exact chk_name uc_name          # Scrub pending detail.
    edit_exact known_name uc_name        # Scrub already detailed.
    if [ ${#uc_name[@]} -gt 0 ]
    then
        chk_name=( ${chk_name[@]} ${uc_name[@]}  )
        pend_func detail_each_name ${1}
    fi
    unset uc_name[@]
    return 0
}

# For each address in uc_address:
#     Move address to chk_address.
#     Add names to uc_name.
#     Pend expand_input_name.
#     Repeat until nothing new found.
# expand_input_address <indirection_limit>
expand_input_address() {
    [ ${#uc_address[@]} -gt 0 ] || return 0
    local -a _eia_addr
    local -a _eia_name
    local -a _eia_new
    local -i _uca_cnt
    local -i _eia_cnt
    local _eia_tst
    unique_lines uc_address _eia_addr
    unset uc_address[@]
    edit_exact been_there_addr _eia_addr
    _uca_cnt=${#_eia_addr[@]}
    [ ${_uca_cnt} -gt 0 ] &&
        been_there_addr=( ${been_there_addr[@]} ${_eia_addr[@]} )

    for (( _eia = 0 ; _eia < _uca_cnt ; _eia++ ))
    do
            if short_rev ${_eia_addr[${_eia}]} _eia_new
            then
                for (( _eia_cnt = 0 ; _eia_cnt < ${#_eia_new[@]} ; _eia_cnt++ ))
                do
                    _eia_tst=${_eia_new[${_eia_cnt}]}
                    if _eia_tst=$(name_fixup ${_eia_tst})
```

```
                    then
                        _eia_name[${#_eia_name[@]}]=${_eia_tst}
                    fi
                done
            fi
    done
    unique_lines _eia_name _eia_name     # Scrub duplicates.
    edit_exact chk_name _eia_name        # Scrub pending detail.
    edit_exact known_name _eia_name      # Scrub already detailed.
    if [ ${#_eia_name[@]} -gt 0 ]        # Anything new?
    then
        uc_name=( ${uc_name[@]} ${_eia_name[@]} )
        pend_func expand_input_name ${1}
        _trace_log[${#_trace_log[@]}]='# # # Added '${#_eia_name[@]}' unchecked name input(s). #
# #'
    fi
    edit_exact chk_address _eia_addr     # Scrub pending detail.
    edit_exact known_address _eia_addr   # Scrub already detailed.
    if [ ${#_eia_addr[@]} -gt 0 ]        # Anything new?
    then
        chk_address=( ${chk_address[@]} ${_eia_addr[@]} )
        pend_func detail_each_address ${1}
    fi
    return 0
}

# The parse-it-yourself zone reply.
# The input is the chk_name list.
# detail_each_name <indirection_limit>
detail_each_name() {
    [ ${#chk_name[@]} -gt 0 ] || return 0
    local -a _den_chk       # Names to check
    local -a _den_name      # Names found here
    local -a _den_address   # Addresses found here
    local -a _den_pair      # Pairs found here
    local -a _den_rev       # Reverse pairs found here
    local -a _den_tmp       # Line being parsed
    local -a _den_auth      # SOA contact being parsed
    local -a _den_new       # The zone reply
    local -a _den_pc        # Parent-Child gets big fast
    local -a _den_ref       # So does reference chain
    local -a _den_nr        # Name-Resource can be big
    local -a _den_na        # Name-Address
    local -a _den_ns        # Name-Service
    local -a _den_achn      # Chain of Authority
    local -i _den_cnt       # Count of names to detail
    local -i _den_lmt       # Indirection limit
    local _den_who          # Named being processed
    local _den_rec          # Record type being processed
    local _den_cont         # Contact domain
    local _den_str          # Fixed up name string
    local _den_str2         # Fixed up reverse
    local IFS=${WSP_IFS}

    # Local, unique copy of names to check
    unique_lines chk_name _den_chk
    unset chk_name[@]       # Done with globals.

    # Less any names already known
    edit_exact known_name _den_chk
    _den_cnt=${#_den_chk[@]}

    # If anything left, add to known_name.
    [ ${_den_cnt} -gt 0 ] &&
        known_name=( ${known_name[@]} ${_den_chk[@]} )

    # for the list of (previously) unknown names . . .
    for (( _den = 0 ; _den < _den_cnt ; _den++ ))
    do
        _den_who=${_den_chk[${_den}]}
        if long_fwd ${_den_who} _den_new
        then
            unique_lines _den_new _den_new
            if [ ${#_den_new[@]} -eq 0 ]
            then
                _den_pair[${#_den_pair[@]}]='0.0.0.0 '${_den_who}
            fi

            # Parse each line in the reply.
            for (( _line = 0 ; _line < ${#_den_new[@]} ; _line++ ))
            do
                IFS=${NO_WSP}$'\x09'$'\x20'
                _den_tmp=( ${_den_new[${_line}]} )
                IFS=${WSP_IFS}
                # If usable record and not a warning message . . .
                if [ ${#_den_tmp[@]} -gt 4 ] && [ 'x'${_den_tmp[0]} != 'x;;' ]
                then
                    _den_rec=${_den_tmp[3]}
                    _den_nr[${#_den_nr[@]}]=${_den_who}' '${_den_rec}
                    # Begin at RFC1033 (+++)
                    case ${_den_rec} in

                         #<name>  [<ttl>]  [<class>]  SOA  <origin>  <person>
                    SOA) # Start Of Authority
                        if _den_str=$(name_fixup ${_den_tmp[0]})
                        then
                            _den_name[${#_den_name[@]}]=${_den_str}
                            _den_achn[${#_den_achn[@]}]=${_den_who}' '${_den_str}' SOA'
                            # SOA origin -- domain name of master zone record
                            if _den_str2=$(name_fixup ${_den_tmp[4]})
                            then
                                _den_name[${#_den_name[@]}]=${_den_str2}
```

```
                                    _den_achn[${#_den_achn[@]}]=${_den_who}' '${_den_str2}' SOA.O'
                                fi
                                # Responsible party e-mail address (possibly bogus).
                                # Possibility of first.last@domain.name ignored.
                                set -f
                                if _den_str2=$(name_fixup ${_den_tmp[5]})
                                then
                                    IFS=${ADR_IFS}
                                    _den_auth=( ${_den_str2} )
                                    IFS=${WSP_IFS}
                                    if [ ${#_den_auth[@]} -gt 2 ]
                                    then
                                         _den_cont=${_den_auth[1]}
                                         for (( _auth = 2 ; _auth < ${#_den_auth[@]} ; _auth++ ))
                                         do
                                           _den_cont=${_den_cont}'.'${_den_auth[${_auth}]}
                                         done
                                         _den_name[${#_den_name[@]}]=${_den_cont}'.'
                                         _den_achn[${#_den_achn[@]}]=${_den_who}' '${_den_cont}'.
SOA.C'
                                    fi
                                fi
                                set +f
                            fi
                        ;;


                        A) # IP(v4) Address Record
                            if _den_str=$(name_fixup ${_den_tmp[0]})
                            then
                                _den_name[${#_den_name[@]}]=${_den_str}
                                _den_pair[${#_den_pair[@]}]=${_den_tmp[4]}' '${_den_str}
                                _den_na[${#_den_na[@]}]=${_den_str}' '${_den_tmp[4]}
                                _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' A'
                            else
                                _den_pair[${#_den_pair[@]}]=${_den_tmp[4]}' unknown.domain'
                                _den_na[${#_den_na[@]}]='unknown.domain '${_den_tmp[4]}
                                _den_ref[${#_den_ref[@]}]=${_den_who}' unknown.domain A'
                            fi
                            _den_address[${#_den_address[@]}]=${_den_tmp[4]}
                            _den_pc[${#_den_pc[@]}]=${_den_who}' '${_den_tmp[4]}
                        ;;

                        NS) # Name Server Record
                            # Domain name being serviced (may be other than current)
                            if _den_str=$(name_fixup ${_den_tmp[0]})
                            then
                                _den_name[${#_den_name[@]}]=${_den_str}
                                _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' NS'

                                # Domain name of service provider
                                if _den_str2=$(name_fixup ${_den_tmp[4]})
                                then
                                    _den_name[${#_den_name[@]}]=${_den_str2}
                                    _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str2}' NSH'
                                    _den_ns[${#_den_ns[@]}]=${_den_str2}' NS'
                                    _den_pc[${#_den_pc[@]}]=${_den_str}' '${_den_str2}
                                fi
                            fi
                        ;;

                        MX) # Mail Server Record
                            # Domain name being serviced (wildcards not handled here)
                            if _den_str=$(name_fixup ${_den_tmp[0]})
                            then
                                _den_name[${#_den_name[@]}]=${_den_str}
                                _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' MX'
                            fi
                            # Domain name of service provider
                            if _den_str=$(name_fixup ${_den_tmp[5]})
                            then
                                _den_name[${#_den_name[@]}]=${_den_str}
                                _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' MXH'
                                _den_ns[${#_den_ns[@]}]=${_den_str}' MX'
                                _den_pc[${#_den_pc[@]}]=${_den_who}' '${_den_str}
                            fi
                        ;;

                        PTR) # Reverse address record
                             # Special name
                            if _den_str=$(name_fixup ${_den_tmp[0]})
                            then
                                _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' PTR'
                                # Host name (not a CNAME)
                                if _den_str2=$(name_fixup ${_den_tmp[4]})
                                then
                                    _den_rev[${#_den_rev[@]}]=${_den_str}' '${_den_str2}
                                    _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str2}' PTRH'
                                    _den_pc[${#_den_pc[@]}]=${_den_who}' '${_den_str}
                                fi
                            fi
                        ;;

                        AAAA) # IP(v6) Address Record
                            if _den_str=$(name_fixup ${_den_tmp[0]})
                            then
                                _den_name[${#_den_name[@]}]=${_den_str}
                                _den_pair[${#_den_pair[@]}]=${_den_tmp[4]}' '${_den_str}
                                _den_na[${#_den_na[@]}]=${_den_str}' '${_den_tmp[4]}
                                _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' AAAA'
                            else
                                _den_pair[${#_den_pair[@]}]=${_den_tmp[4]}' unknown.domain'
```

```
                            _den_na[${#_den_na[@]}]='unknown.domain '${_den_tmp[4]}
                            _den_ref[${#_den_ref[@]}]=${_den_who}' unknown.domain'
                        fi
                        # No processing for IPv6 addresses
                            _den_pc[${#_den_pc[@]}]=${_den_who}' '${_den_tmp[4]}
                    ;;

                    CNAME) # Alias name record
                           # Nickname
                        if _den_str=$(name_fixup ${_den_tmp[0]})
                        then
                            _den_name[${#_den_name[@]}]=${_den_str}
                            _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' CNAME'
                            _den_pc[${#_den_pc[@]}]=${_den_who}' '${_den_str}
                        fi
                        # Hostname
                        if _den_str=$(name_fixup ${_den_tmp[4]})
                        then
                            _den_name[${#_den_name[@]}]=${_den_str}
                            _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' CHOST'
                            _den_pc[${#_den_pc[@]}]=${_den_who}' '${_den_str}
                        fi
                    ;;
#                   TXT)
#                   ;;
                    esac
                fi
            done
        else # Lookup error == 'A' record 'unknown address'
            _den_pair[${#_den_pair[@]}]='0.0.0.0 '${_den_who}
        fi
    done

    # Control dot array growth.
    unique_lines _den_achn _den_achn      # Works best, all the same.
    edit_exact auth_chain _den_achn       # Works best, unique items.
    if [ ${#_den_achn[@]} -gt 0 ]
    then
        IFS=${NO_WSP}
        auth_chain=( ${auth_chain[@]} ${_den_achn[@]} )
        IFS=${WSP_IFS}
    fi

    unique_lines _den_ref _den_ref      # Works best, all the same.
    edit_exact ref_chain _den_ref       # Works best, unique items.
    if [ ${#_den_ref[@]} -gt 0 ]
    then
        IFS=${NO_WSP}
        ref_chain=( ${ref_chain[@]} ${_den_ref[@]} )
        IFS=${WSP_IFS}
    fi

    unique_lines _den_na _den_na
    edit_exact name_address _den_na
    if [ ${#_den_na[@]} -gt 0 ]
    then
        IFS=${NO_WSP}
        name_address=( ${name_address[@]} ${_den_na[@]} )
        IFS=${WSP_IFS}
    fi

    unique_lines _den_ns _den_ns
    edit_exact name_srvc _den_ns
    if [ ${#_den_ns[@]} -gt 0 ]
    then
        IFS=${NO_WSP}
        name_srvc=( ${name_srvc[@]} ${_den_ns[@]} )
        IFS=${WSP_IFS}
    fi

    unique_lines _den_nr _den_nr
    edit_exact name_resource _den_nr
    if [ ${#_den_nr[@]} -gt 0 ]
    then
        IFS=${NO_WSP}
        name_resource=( ${name_resource[@]} ${_den_nr[@]} )
        IFS=${WSP_IFS}
    fi

    unique_lines _den_pc _den_pc
    edit_exact parent_child _den_pc
    if [ ${#_den_pc[@]} -gt 0 ]
    then
        IFS=${NO_WSP}
        parent_child=( ${parent_child[@]} ${_den_pc[@]} )
        IFS=${WSP_IFS}
    fi

    # Update list known_pair (Address and Name).
    unique_lines _den_pair _den_pair
    edit_exact known_pair _den_pair
    if [ ${#_den_pair[@]} -gt 0 ]  # Anything new?
    then
        IFS=${NO_WSP}
        known_pair=( ${known_pair[@]} ${_den_pair[@]} )
        IFS=${WSP_IFS}
    fi

    # Update list of reverse pairs.
    unique_lines _den_rev _den_rev
    edit_exact reverse_pair _den_rev
    if [ ${#_den_rev[@]} -gt 0 ]   # Anything new?
```

```
1442     then
1443         IFS=${NO_WSP}
1444         reverse_pair=( ${reverse_pair[@]} ${_den_rev[@]} )
1445         IFS=${WSP_IFS}
1446     fi
1447
1448     # Check indirection limit -- give up if reached.
1449     if ! _den_lmt=$(limit_chk ${1})
1450     then
1451         return 0
1452     fi
1453
1454     # Execution engine is LIFO. Order of pend operations is important.
1455     # Did we define any new addresses?
1456     unique_lines _den_address _den_address    # Scrub duplicates.
1457     edit_exact known_address _den_address     # Scrub already processed.
1458     edit_exact un_address _den_address        # Scrub already waiting.
1459     if [ ${#_den_address[@]} -gt 0 ]          # Anything new?
1460     then
1461         uc_address=( ${uc_address[@]} ${_den_address[@]} )
1462         pend_func expand_input_address ${_den_lmt}
1463         _trace_log[${#_trace_log[@]}]='# # # Added '${#_den_address[@]}' unchecked address(s). # #
#'
1464     fi
1465
1466     # Did we find any new names?
1467     unique_lines _den_name _den_name          # Scrub duplicates.
1468     edit_exact known_name _den_name           # Scrub already processed.
1469     edit_exact uc_name _den_name              # Scrub already waiting.
1470     if [ ${#_den_name[@]} -gt 0 ]             # Anything new?
1471     then
1472         uc_name=( ${uc_name[@]} ${_den_name[@]} )
1473         pend_func expand_input_name ${_den_lmt}
1474         _trace_log[${#_trace_log[@]}]='# # # Added '${#_den_name[@]}' unchecked name(s). # # #'
1475     fi
1476     return 0
1477 }
1478
1479 # The parse-it-yourself delegation reply
1480 # Input is the chk_address list.
1481 # detail_each_address <indirection_limit>
1482 detail_each_address() {
1483     [ ${#chk_address[@]} -gt 0 ] || return 0
1484     unique_lines chk_address chk_address
1485     edit_exact known_address chk_address
1486     if [ ${#chk_address[@]} -gt 0 ]
1487     then
1488         known_address=( ${known_address[@]} ${chk_address[@]} )
1489         unset chk_address[@]
1490     fi
1491     return 0
1492 }
1493
1494 # # # Application specific output functions # # #
1495
1496 # Pretty print the known pairs.
1497 report_pairs() {
1498     echo
1499     echo 'Known network pairs.'
1500     col_print known_pair 2 5 30
1501
1502     if [ ${#auth_chain[@]} -gt 0 ]
1503     then
1504         echo
1505         echo 'Known chain of authority.'
1506         col_print auth_chain 2 5 30 55
1507     fi
1508
1509     if [ ${#reverse_pair[@]} -gt 0 ]
1510     then
1511         echo
1512         echo 'Known reverse pairs.'
1513         col_print reverse_pair 2 5 55
1514     fi
1515     return 0
1516 }
1517
1518 # Check an address against the list of blacklist servers.
1519 # A good place to capture for GraphViz: address->status(server(reports))
1520 # check_lists <ip_address>
1521 check_lists() {
1522     [ $# -eq 1 ] || return 1
1523     local -a _cl_fwd_addr
1524     local -a _cl_rev_addr
1525     local -a _cl_reply
1526     local -i _cl_rc
1527     local -i _ls_cnt
1528     local _cl_dns_addr
1529     local _cl_lkup
1530
1531     split_ip ${1} _cl_fwd_addr _cl_rev_addr
1532     _cl_dns_addr=$(dot_array _cl_rev_addr)'.'
1533     _ls_cnt=${#list_server[@]}
1534     echo '    Checking address '${1}
1535     for (( _cl = 0 ; _cl < _ls_cnt ; _cl++ ))
1536     do
1537         _cl_lkup=${_cl_dns_addr}${list_server[${_cl}]}
1538         if short_text ${_cl_lkup} _cl_reply
1539         then
1540             if [ ${#_cl_reply[@]} -gt 0 ]
1541             then
1542                 echo '        Records from '${list_server[${_cl}]}
```

```
                address_hits[${#address_hits[@]}]=${1}' '${list_server[${_cl}]}
                _hs_RC=2
                for (( _clr = 0 ; _clr < ${#_cl_reply[@]} ; _clr++ ))
                do
                    echo '            '${_cl_reply[${_clr}]}
                done
            fi
        fi
    done
    return 0
}

# # # The usual application glue # # #

# Who did it?
credits() {
   echo
   echo 'Advanced Bash Scripting Guide: is_spammer.bash, v2, 2004-msz'
}

# How to use it?
# (See also, "Quickstart" at end of script.)
usage() {
    cat <<-'_usage_statement_'
    The script is_spammer.bash requires either one or two arguments.

    arg 1) May be one of:
        a) A domain name
        b) An IPv4 address
        c) The name of a file with any mix of names
           and addresses, one per line.

    arg 2) May be one of:
        a) A Blacklist server domain name
        b) The name of a file with Blacklist server
           domain names, one per line.
        c) If not present, a default list of (free)
           Blacklist servers is used.
        d) If a filename of an empty, readable, file
           is given,
           Blacklist server lookup is disabled.

    All script output is written to stdout.

    Return codes: 0 -> All OK, 1 -> Script failure,
                  2 -> Something is Blacklisted.

    Requires the external program 'dig' from the 'bind-9'
    set of DNS programs.  See: http://www.isc.org

    The domain name lookup depth limit defaults to 2 levels.
    Set the environment variable SPAMMER_LIMIT to change.
    SPAMMER_LIMIT=0 means 'unlimited'

    Limit may also be set on the command line.
    If arg#1 is an integer, the limit is set to that value
    and then the above argument rules are applied.

    Setting the environment variable 'SPAMMER_DATA' to a filename
    will cause the script to write a GraphViz graphic file.

    For the development version;
    Setting the environment variable 'SPAMMER_TRACE' to a filename
    will cause the execution engine to log a function call trace.

_usage_statement_
}

# The default list of Blacklist servers:
# Many choices, see: http://www.spews.org/lists.html

declare -a default_servers
# See: http://www.spamhaus.org (Conservative, well maintained)
default_servers[0]='sbl-xbl.spamhaus.org'
# See: http://ordb.org (Open mail relays)
default_servers[1]='relays.ordb.org'
# See: http://www.spamcop.net/ (You can report spammers here)
default_servers[2]='bl.spamcop.net'
# See: http://www.spews.org (An 'early detect' system)
default_servers[3]='l2.spews.dnsbl.sorbs.net'
# See: http://www.dnsbl.us.sorbs.net/using.shtml
default_servers[4]='dnsbl.sorbs.net'
# See: http://dsbl.org/usage (Various mail relay lists)
default_servers[5]='list.dsbl.org'
default_servers[6]='multihop.dsbl.org'
default_servers[7]='unconfirmed.dsbl.org'

# User input argument #1
setup_input() {
    if [ -e ${1} ] && [ -r ${1} ]  # Name of readable file
    then
        file_to_array ${1} uc_name
        echo 'Using filename >'${1}'< as input.'
    else
        if is_address ${1}          # IP address?
        then
            uc_address=( ${1} )
            echo 'Starting with address >'${1}'<'
        else                        # Must be a name.
            uc_name=( ${1} )
            echo 'Starting with domain name >'${1}'<'
        fi
```

```
    fi
    return 0
}

# User input argument #2
setup_servers() {
    if [ -e ${1} ] && [ -r ${1} ]  # Name of a readable file
    then
        file_to_array ${1} list_server
        echo 'Using filename >'${1}'< as blacklist server list.'
    else
        list_server=( ${1} )
        echo 'Using blacklist server >'${1}'<'
    fi
    return 0
}

# User environment variable SPAMMER_TRACE
live_log_die() {
    if [ ${SPAMMER_TRACE:=} ]    # Wants trace log?
    then
        if [ ! -e ${SPAMMER_TRACE} ]
        then
            if ! touch ${SPAMMER_TRACE} 2>/dev/null
            then
                pend_func echo $(printf '%q\n' \
                'Unable to create log file >'${SPAMMER_TRACE}'<')
                pend_release
                exit 1
            fi
            _log_file=${SPAMMER_TRACE}
            _pend_hook_=trace_logger
            _log_dump=dump_log
        else
            if [ ! -w ${SPAMMER_TRACE} ]
            then
                pend_func echo $(printf '%q\n' \
                'Unable to write log file >'${SPAMMER_TRACE}'<')
                pend_release
                exit 1
            fi
            _log_file=${SPAMMER_TRACE}
            echo '' > ${_log_file}
            _pend_hook_=trace_logger
            _log_dump=dump_log
        fi
    fi
    return 0
}

# User environment variable SPAMMER_DATA
data_capture() {
    if [ ${SPAMMER_DATA:=} ]    # Wants a data dump?
    then
        if [ ! -e ${SPAMMER_DATA} ]
       then
            if ! touch ${SPAMMER_DATA} 2>/dev/null
            then
                pend_func echo $(printf '%q]n' \
                'Unable to create data output file >'${SPAMMER_DATA}'<')
                pend_release
                exit 1
            fi
            _dot_file=${SPAMMER_DATA}
            _dot_dump=dump_dot
       else
            if [ ! -w ${SPAMMER_DATA} ]
            then
                pend_func echo $(printf '%q\n' \
                'Unable to write data output file >'${SPAMMER_DATA}'<')
                pend_release
                exit 1
            fi
            _dot_file=${SPAMMER_DATA}
            _dot_dump=dump_dot
       fi
   fi
   return 0


 Grope user specified arguments.
o_user_args() {
   if [ $# -gt 0 ] && is_number $1
   then
       indirect=$1
       shift
   fi

   case $# in                     # Did user treat us well?
       1)
           if ! setup_input $1    # Needs error checking.
           then
               pend_release
               $_log_dump
               exit 1
           fi
           list_server=( ${default_servers[@]} )
           _list_cnt=${#list_server[@]}
           echo 'Using default blacklist server list.'
           echo 'Search depth limit: '${indirect}
           ;;
       2)
```

```
            if ! setup_input $1    # Needs error checking.
            then
                pend_release
                $_log_dump
                exit 1
            fi
            if ! setup_servers $2  # Needs error checking.
            then
                pend_release
                $_log_dump
                exit 1
            fi
            echo 'Search depth limit: '${indirect}
            ;;
        *)
            pend_func usage
            pend_release
            $_log_dump
            exit 1
            ;;
    esac
    return 0


 A general purpose debug tool.
 list_array <array_name>
ist_array() {
    [ $# -eq 1 ] || return 1  # One argument required.

    local -a _la_lines
    set -f
    local IFS=${NO_WSP}
    eval _la_lines=\(\ \$\{$1\[@\]\}\ \)
    echo
    echo "Element count "${#_la_lines[@]}" array "${1}
    local _ln_cnt=${#_la_lines[@]}

    for (( _i = 0; _i < ${_ln_cnt}; _i++ ))
    do
        echo 'Element '$_i' >'${_la_lines[$_i]}'<'
    done
    set +f
    return 0


 # # 'Hunt the Spammer' program code # # #
end_init                               # Ready stack engine.
end_func credits                       # Last thing to print.

 # # Deal with user # # #
ive_log_die                            # Setup debug trace log.
ata_capture                            # Setup data capture file.
cho
do_user_args $@

# # # Haven't exited yet - There is some hope # # #
# Discovery group - Execution engine is LIFO - pend
# in reverse order of execution.
_hs_RC=0                                # Hunt the Spammer return code
pend_mark
    pend_func report_pairs              # Report name-address pairs.

    # The two detail_* are mutually recursive functions.
    # They also pend expand_* functions as required.
    # These two (the last of ???) exit the recursion.
    pend_func detail_each_address       # Get all resources of addresses.
    pend_func detail_each_name          # Get all resources of names.

    #  The two expand_* are mutually recursive functions,
    #+ which pend additional detail_* functions as required.
    pend_func expand_input_address 1    # Expand input names by address.
    pend_func expand_input_name 1       # #xpand input addresses by name.

    # Start with a unique set of names and addresses.
    pend_func unique_lines uc_address uc_address
    pend_func unique_lines uc_name uc_name

    # Separate mixed input of names and addresses.
    pend_func split_input
pend_release

# # # Pairs reported -- Unique list of IP addresses found
echo
_ip_cnt=${#known_address[@]}
if [ ${#list_server[@]} -eq 0 ]
then
    echo 'Blacklist server list empty, none checked.'
else
    if [ ${_ip_cnt} -eq 0 ]
    then
        echo 'Known address list empty, none checked.'
    else
        _ip_cnt=${_ip_cnt}-1   # Start at top.
        echo 'Checking Blacklist servers.'
        for (( _ip = _ip_cnt ; _ip >= 0 ; _ip-- ))
        do
            pend_func check_lists $( printf '%q\n' ${known_address[$_ip]} )
        done
    fi
fi
pend_release
$_dot_dump                    # Graphics file dump
```

```
$_log_dump                    # Execution trace
echo


##############################
# Example output from script #
##############################
:<<-'_is_spammer_outputs_'

./is_spammer.bash 0 web4.alojamentos7.com

Starting with domain name >web4.alojamentos7.com<
Using default blacklist server list.
Search depth limit: 0
.:....::::...:::..:::.....::..::...::::....::
Known network pairs.
    66.98.208.97            web4.alojamentos7.com.
    66.98.208.97            ns1.alojamentos7.com.
    69.56.202.147           ns2.alojamentos.ws.
    66.98.208.97            alojamentos7.com.
    66.98.208.97            web.alojamentos7.com.
    69.56.202.146           ns1.alojamentos.ws.
    69.56.202.146           alojamentos.ws.
    66.235.180.113          ns1.alojamentos.org.
    66.235.181.192          ns2.alojamentos.org.
    66.235.180.113          alojamentos.org.
    66.235.180.113          web6.alojamentos.org.
    216.234.234.30          ns1.theplanet.com.
    12.96.160.115           ns2.theplanet.com.
    216.185.111.52          mail1.theplanet.com.
    69.56.141.4             spooling.theplanet.com.
    216.185.111.40          theplanet.com.
    216.185.111.40          www.theplanet.com.
    216.185.111.52          mail.theplanet.com.

Checking Blacklist servers.
    Checking address 66.98.208.97
        Records from dnsbl.sorbs.net
            "Spam Received See: http://www.dnsbl.sorbs.net/lookup.shtml?66.98.208.97"
    Checking address 69.56.202.147
    Checking address 69.56.202.146
    Checking address 66.235.180.113
    Checking address 66.235.181.192
    Checking address 216.185.111.40
    Checking address 216.234.234.30
    Checking address 12.96.160.115
    Checking address 216.185.111.52
    Checking address 69.56.141.4

Advanced Bash Scripting Guide: is_spammer.bash, v2, 2004-msz

_is_spammer_outputs_

exit ${_hs_RC}

#####################################################
#  The script ignores everything from here on down #
#+ because of the 'exit' command, just above.      #
#####################################################



Quickstart
==========

 Prerequisites

  Bash version 2.05b or 3.00 (bash --version)
  A version of Bash which supports arrays. Array
  support is included by default Bash configurations.

  'dig,' version 9.x.x (dig $HOSTNAME, see first line of output)
  A version of dig which supports the +short options.
  See: dig_wrappers.bash for details.


 Optional Prerequisites

  'named,' a local DNS caching program. Any flavor will do.
  Do twice: dig $HOSTNAME
  Check near bottom of output for: SERVER: 127.0.0.1#53
  That means you have one running.


 Optional Graphics Support

  'date,' a standard *nix thing. (date -R)

  dot Program to convert graphic description file to a
  diagram. (dot -V)
  A part of the Graph-Viz set of programs.
  See: [http://www.research.att.com/sw/tools/graphviz||GraphViz]

  'dotty,' a visual editor for graphic description files.
  Also a part of the Graph-Viz set of programs.




 Quick Start

In the same directory as the is_spammer.bash script;
```

```
Do: ./is_spammer.bash

 Usage Details

1. Blacklist server choices.

  (a) To use default, built-in list: Do nothing.

  (b) To use your own list:

    i. Create a file with a single Blacklist server
       domain name per line.

    ii. Provide that filename as the last argument to
        the script.

  (c) To use a single Blacklist server: Last argument
      to the script.

  (d) To disable Blacklist lookups:

    i. Create an empty file (touch spammer.nul)
       Your choice of filename.

    ii. Provide the filename of that empty file as the
        last argument to the script.

2. Search depth limit.

  (a) To use the default value of 2: Do nothing.

  (b) To set a different limit:
      A limit of 0 means: no limit.

    i. export SPAMMER_LIMIT=1
       or whatever limit you want.

    ii. OR provide the desired limit as the first
       argument to the script.

3. Optional execution trace log.

  (a) To use the default setting of no log output: Do nothing.

  (b) To write an execution trace log:
      export SPAMMER_TRACE=spammer.log
      or whatever filename you want.

4. Optional graphic description file.

  (a) To use the default setting of no graphic file: Do nothing.

  (b) To write a Graph-Viz graphic description file:
      export SPAMMER_DATA=spammer.dot
      or whatever filename you want.

5. Where to start the search.

  (a) Starting with a single domain name:

    i. Without a command line search limit: First
       argument to script.

    ii. With a command line search limit: Second
        argument to script.

  (b) Starting with a single IP address:

    i. Without a command line search limit: First
       argument to script.

    ii. With a command line search limit: Second
        argument to script.

  (c) Starting with (mixed) multiple name(s) and/or address(es):
      Create a file with one name or address per line.
      Your choice of filename.

    i. Without a command line search limit: Filename as
       first argument to script.

    ii. With a command line search limit: Filename as
        second argument to script.

6. What to do with the display output.

  (a) To view display output on screen: Do nothing.

  (b) To save display output to a file: Redirect stdout to a filename.

  (c) To discard display output: Redirect stdout to /dev/null.

7. Temporary end of decision making.
   press RETURN
   wait (optionally, watch the dots and colons).

8. Optionally check the return code.

  (a) Return code 0: All OK

  (b) Return code 1: Script setup failure
```

```
  (c) Return code 2: Something was blacklisted.

9. Where is my graph (diagram)?

The script does not directly produce a graph (diagram).
It only produces a graphic description file. You can
process the graphic descriptor file that was output
with the 'dot' program.

Until you edit that descriptor file, to describe the
relationships you want shown, all that you will get is
a bunch of labeled name and address nodes.

All of the script's discovered relationships are within
a comment block in the graphic descriptor file, each
with a descriptive heading.

The editing required to draw a line between a pair of
nodes from the information in the descriptor file may
be done with a text editor.

Given these lines somewhere in the descriptor file:

# Known domain name nodes

N0000 [label="guardproof.info."] ;

N0002 [label="third.guardproof.info."] ;



# Known address nodes

A0000 [label="61.141.32.197"] ;



/*

# Known name->address edges

NA0000 third.guardproof.info. 61.141.32.197



# Known parent->child edges

PC0000 guardproof.info. third.guardproof.info.

 */

Turn that into the following lines by substituting node
identifiers into the relationships:

# Known domain name nodes

N0000 [label="guardproof.info."] ;

N0002 [label="third.guardproof.info."] ;



# Known address nodes

A0000 [label="61.141.32.197"] ;



# PC0000 guardproof.info. third.guardproof.info.

N0000->N0002 ;



# NA0000 third.guardproof.info. 61.141.32.197

N0002->A0000 ;



/*

# Known name->address edges

NA0000 third.guardproof.info. 61.141.32.197



# Known parent->child edges

PC0000 guardproof.info. third.guardproof.info.

 */

Process that with the 'dot' program, and you have your
first network diagram.

In addition to the conventional graphic edges, the
descriptor file includes similar format pair-data that
describes services, zone records (sub-graphs?),
blacklisted addresses, and other things which might be
interesting to include in your graph. This additional
```

```
information could be displayed as different node
shapes, colors, line sizes, etc.

The descriptor file can also be read and edited by a
Bash script (of course). You should be able to find
most of the functions required within the
"is_spammer.bash" script.

# End Quickstart.



Additional Note
========== ====

Michael Zick points out that there is a "makeviz.bash" interactive
Web site at rediris.es. Can't give the full URL, since this is not
a publically accessible site.
```

另一个阻挡垃圾邮件的脚本.

例子 A-29. 垃圾邮件服务器猎手

```
#!/bin/bash
# whx.sh: "whois" spammer lookup
# Author: Walter Dnes
# Slight revisions (first section) by ABS Guide author.
# Used in ABS Guide with permission.

# Needs version 3.x or greater of Bash to run (because of =~ operator).
# Commented by script author and ABS Guide author.


E_BADARGS=65        # Missing command-line arg.
E_NOHOST=66         # Host not found.
E_TIMEOUT=67        # Host lookup timed out.
E_UNDEF=68          # Some other (undefined) error.
HOSTWAIT=10         # Specify up to 10 seconds for host query reply.
                    # The actual wait may be a bit longer.
OUTFILE=whois.txt   # Output file.
PORT=4321


if [ -z "$1" ]      # Check for (required) command-line arg.
then
  echo "Usage: $0 domain name or IP address"
  exit $E_BADARGS
fi


if [[ "$1" =~ "[a-zA-Z][a-zA-Z]$" ]]  # Ends in two alpha chars?
then                                  # It's a domain name && must do host lookup.
  IPADDR=$(host -W $HOSTWAIT $1 | awk '{print $4}')
                                      # Doing host lookup to get IP address.
                                  # Extract final field.
else
  IPADDR="$1"                         # Command-line arg was IP address.
fi

echo; echo "IP Address is: "$IPADDR""; echo

if [ -e "$OUTFILE" ]
then
  rm -f "$OUTFILE"
  echo "Stale output file \"$OUTFILE\" removed."; echo
fi


#  Sanity checks.
#  (This section needs more work.)
#  ==============================
if [ -z "$IPADDR" ]
# No response.
then
  echo "Host not found!"
  exit $E_NOHOST    # Bail out.
fi

if [[ "$IPADDR" =~ "^[;;]" ]]
#  ;; connection timed out; no servers could be reached
then
  echo "Host lookup timed out!"
  exit $E_TIMEOUT   # Bail out.
fi

if [[ "$IPADDR" =~ "[(NXDOMAIN)]$" ]]
#  Host xxxxxxxxx.xxx not found: 3(NXDOMAIN)
then
  echo "Host not found!"
  exit $E_NOHOST    # Bail out.
fi

if [[ "$IPADDR" =~ "[(SERVFAIL)]$" ]]
#  Host xxxxxxxxx.xxx not found: 2(SERVFAIL)
then
  echo "Host not found!"
```

```
  exit $E_NOHOST    # Bail out.
fi




# ======================== Main body of script ========================

AFRINICquery() {
#  Define the function that queries AFRINIC. Echo a notification to the
#+ screen, and then run the actual query, redirecting output to $OUTFILE.

  echo "Searching for $IPADDR in whois.afrinic.net"
  whois -h whois.afrinic.net "$IPADDR" > $OUTFILE

#  Check for presence of reference to an rwhois.
#  Warn about non-functional rwhois.infosat.net server
#+ and attempt rwhois query.
  if grep -e "^remarks: .*rwhois\.[^ ]\+" "$OUTFILE"
  then
    echo " " >> $OUTFILE
    echo "***" >> $OUTFILE
    echo "***" >> $OUTFILE
    echo "Warning: rwhois.infosat.net was not working as of 2005/02/02" >> $OUTFILE
    echo "         when this script was written." >> $OUTFILE
    echo "***" >> $OUTFILE
    echo "***" >> $OUTFILE
    echo " " >> $OUTFILE
    RWHOIS=`grep "^remarks: .*rwhois\.[^ ]\+" "$OUTFILE" | tail -n 1 |\
    sed "s/\(^.*\)\(rwhois\..*\)\(:4.*\)/\2/"`
    whois -h ${RWHOIS}:${PORT} "$IPADDR" >> $OUTFILE
  fi
}

APNICquery() {
  echo "Searching for $IPADDR in whois.apnic.net"
  whois -h whois.apnic.net "$IPADDR" > $OUTFILE

#  Just  about  every  country has its own internet registrar.
#  I don't normally bother consulting them, because the regional registry
#+ usually supplies sufficient information.
#  There are a few exceptions, where the regional registry simply
#+ refers to the national registry for direct data.
#  These are Japan and South Korea in APNIC, and Brasil in LACNIC.
#  The following if statement checks $OUTFILE (whois.txt) for the presence
#+ of "KR" (South Korea) or "JP" (Japan) in the country field.
#  If either is found, the query is re-run against the appropriate
#+ national registry.

  if grep -E "^country:[ ]+KR$" "$OUTFILE"
  then
    echo "Searching for $IPADDR in whois.krnic.net"
    whois -h whois.krnic.net "$IPADDR" >> $OUTFILE
  elif grep -E "^country:[ ]+JP$" "$OUTFILE"
  then
    echo "Searching for $IPADDR in whois.nic.ad.jp"
    whois -h whois.nic.ad.jp "$IPADDR"/e >> $OUTFILE
  fi
}

ARINquery() {
  echo "Searching for $IPADDR in whois.arin.net"
  whois -h whois.arin.net "$IPADDR" > $OUTFILE

#  Several large internet providers listed by ARIN have their own
#+ internal whois service, referred to as "rwhois".
#  A large block of IP addresses is listed with the provider
#+ under the ARIN registry.
#  To get the IP addresses of 2nd-level ISPs or other large customers,
#+ one has to refer to the rwhois server on port 4321.
#  I originally started with a bunch of "if" statements checking for
#+ the larger providers.
#  This approach is unwieldy, and there's always another rwhois server
#+ that I didn't know about.
#  A more elegant approach is to check $OUTFILE for a reference
#+ to a whois server, parse that server name out of the comment section,
#+ and re-run the query against the appropriate rwhois server.
#  The parsing looks a bit ugly, with a long continued line inside
#+ backticks.
#  But it only has to be done once, and will work as new servers are added.
#@   ABS Guide author comment: it isn't all that ugly, and is, in fact,
#@+  an instructive use of Regular Expressions.

  if grep -E "^Comment: .*rwhois.[^ ]+" "$OUTFILE"
  then
    RWHOIS=`grep -e "^Comment:.*rwhois\.[^ ]\+" "$OUTFILE" | tail -n 1 |\
    sed "s/^\(.*\)\(rwhois\.[^ ]\+\)\(.*$\)/\2/"`
    echo "Searching for $IPADDR in ${RWHOIS}"
    whois -h ${RWHOIS}:${PORT} "$IPADDR" >> $OUTFILE
  fi
}

LACNICquery() {
  echo "Searching for $IPADDR in whois.lacnic.net"
  whois -h whois.lacnic.net "$IPADDR" > $OUTFILE

#  The  following if statement checks $OUTFILE (whois.txt) for the presence of
#+ "BR" (Brasil) in the country field.
#  If it is found, the query is re-run against whois.registro.br.

  if grep -E "^country:[ ]+BR$" "$OUTFILE"
  then
```

```
    echo "Searching for $IPADDR in whois.registro.br"
    whois -h whois.registro.br "$IPADDR" >> $OUTFILE
  fi
}

RIPEquery() {
  echo "Searching for $IPADDR in whois.ripe.net"
  whois -h whois.ripe.net "$IPADDR" > $OUTFILE
}

#  Initialize a few variables.
#  * slash8 is the most significant octet
#  * slash16 consists of the two most significant octets
#  * octet2 is the second most significant octet




slash8=`echo $IPADDR | cut -d. -f 1`
  if [ -z "$slash8" ]  # Yet another sanity check.
  then
    echo "Undefined error!"
    exit $E_UNDEF
  fi
slash16=`echo $IPADDR | cut -d. -f 1-2`
#                           ^ Period specified as 'cut" delimiter.
  if [ -z "$slash16" ]
  then
    echo "Undefined error!"
    exit $E_UNDEF
  fi
octet2=`echo $slash16 | cut -d. -f 2`
  if [ -z "$octet2" ]
  then
    echo "Undefined error!"
    exit $E_UNDEF
  fi


#  Check for various odds and ends of reserved space.
#  There is no point in querying for those addresses.

if [ $slash8 == 0 ]; then
  echo $IPADDR is '"This Network"' space\; Not querying
elif [ $slash8 == 10 ]; then
  echo $IPADDR is RFC1918 space\; Not querying
elif [ $slash8 == 14 ]; then
  echo $IPADDR is '"Public Data Network"' space\; Not querying
elif [ $slash8 == 127 ]; then
  echo $IPADDR is loopback space\; Not querying
elif [ $slash16 == 169.254 ]; then
  echo $IPADDR is link-local space\; Not querying
elif [ $slash8 == 172 ] && [ $octet2 -ge 16 ] && [ $octet2 -le 31 ];then
  echo $IPADDR is RFC1918 space\; Not querying
elif [ $slash16 == 192.168 ]; then
  echo $IPADDR is RFC1918 space\; Not querying
elif [ $slash8 -ge 224 ]; then
  echo $IPADDR is either Multicast or reserved space\; Not querying
elif [ $slash8 -ge 200 ] && [ $slash8 -le 201 ]; then LACNICquery "$IPADDR"
elif [ $slash8 -ge 202 ] && [ $slash8 -le 203 ]; then APNICquery "$IPADDR"
elif [ $slash8 -ge 210 ] && [ $slash8 -le 211 ]; then APNICquery "$IPADDR"
elif [ $slash8 -ge 218 ] && [ $slash8 -le 223 ]; then APNICquery "$IPADDR"

#  If we got this far without making a decision, query ARIN.
#  If a reference is found in $OUTFILE to APNIC, AFRINIC, LACNIC, or RIPE,
#+ query the appropriate whois server.

else
  ARINquery "$IPADDR"
  if grep "whois.afrinic.net" "$OUTFILE"; then
    AFRINICquery "$IPADDR"
  elif grep -E "^OrgID:[ ]+RIPE$" "$OUTFILE"; then
    RIPEquery "$IPADDR"
  elif grep -E "^OrgID:[ ]+APNIC$" "$OUTFILE"; then
    APNICquery "$IPADDR"
  elif grep -E "^OrgID:[ ]+LACNIC$" "$OUTFILE"; then
    LACNICquery "$IPADDR"
  fi
fi

#@  ----------------------------------------------------------------
#   Try also:
#   wget http://logi.cc/nw/whois.php3?ACTION=doQuery&DOMAIN=$IPADDR
#@  ----------------------------------------------------------------

#  We've  now  finished  the querying.
#  Echo a copy of the final result to the screen.

cat $OUTFILE
# Or "less $OUTFILE" . . .


exit 0

#@  ABS Guide author comments:
#@  Nothing fancy here, but still a very useful tool for hunting spammers.
#@  Sure, the script can be cleaned up some, and it's still a bit buggy,
#@+ (exercise for reader), but all the same, it's a nice piece of coding
#@+ by Walter Dnes.
#@  Thank you!
```

"Little Monster的"之前的wget.

例子 A-30. 使得wget更易用

```
#!/bin/bash
# wgetter2.bash

# Author: Little Monster [monster@monstruum.co.uk]
# ==> Used in ABS Guide with permission of script author.
# ==> This script still needs debugging and fixups (exercise for reader).
# ==> It could also use some additional editing in the comments.


#  This is wgetter2 --
#+ a Bash script to make wget a bit more friendly, and save typing.

#  Carefully crafted by Little Monster.
#  More or less complete on 02/02/2005.
#  If you think this script can be improved,
#+ email me at: monster@monstruum.co.uk
# ==> and cc: to the author of the ABS Guide, please.
#  This script is licenced under the GPL.
#  You are free to copy, alter and re-use it,
#+ but please don't try to claim you wrote it.
#  Log your changes here instead.

# ========================================================================
# changelog:

# 07/02/2005.  Fixups by Little Monster.
# 02/02/2005.  Minor additions by Little Monster.
#               (See after # ++++++++++ )
# 29/01/2005.  Minor stylistic edits and cleanups by author of ABS Guide.
#               Added exit error codes.
# 22/11/2004.  Finished initial version of second version of wgetter:
#               wgetter2 is born.
# 01/12/2004.  Changed 'runn' function so it can be run 2 ways --
#               either ask for a file name or have one input on the CL.
# 01/12/2004.  Made sensible handling of no URL's given.
# 01/12/2004.  Made loop of main options, so you don't
#               have to keep calling wgetter 2 all the time.
#               Runs as a session instead.
# 01/12/2004.  Added looping to 'runn' function.
#               Simplified and improved.
# 01/12/2004.  Added state to recursion setting.
#               Enables re-use of previous value.
# 05/12/2004.  Modified the file detection routine in the 'runn' function
#               so it's not fooled by empty values, and is cleaner.
# 01/02/2004.  Added cookie finding routine from later version (which
#               isn't ready yet), so as not to have hard-coded paths.
# ========================================================================

# Error codes for abnormal exit.
E_USAGE=67        # Usage message, then quit.
E_NO_OPTS=68      # No command-line args entered.
E_NO_URLS=69      # No URLs passed to script.
E_NO_SAVEFILE=70  # No save filename passed to script.
E_USER_EXIT=71    # User decides to quit.


#  Basic default wget command we want to use.
#  This is the place to change it, if required.
#  NB: if using a proxy, set http_proxy = yourproxy in .wgetrc.
#  Otherwise delete --proxy=on, below.
# ========================================================================
CommandA="wget -nc -c -t 5 --progress=bar --random-wait --proxy=on -r"
# ========================================================================



# ------------------------------------------------------------------------
# Set some other variables and explain them.

pattern=" -A .jpg,.JPG,.jpeg,.JPEG,.gif,.GIF,.htm,.html,.shtml,.php"
                     # wget's option to only get certain types of file.
                     # comment out if not using
today=`date +%F`     # Used for a filename.
home=$HOME           # Set HOME to an internal variable.
                     # In case some other path is used, change it here.
depthDefault=3       # Set a sensible default recursion.
Depth=$depthDefault  # Otherwise user feedback doesn't tie in properly.
RefA=""              # Set blank referring page.
Flag=""              #  Default to not saving anything,
                     #+ or whatever else might be wanted in future.
lister=""            # Used for passing a list of urls directly to wget.
Woptions=""          # Used for passing wget some options for itself.
inFile=""            # Used for the run function.
newFile=""           # Used for the run function.
savePath="$home/w-save"
Config="$home/.wgetter2rc"
                     #  This is where some variables can be stored,
                     #+ if permanently changed from within the script.
Cookie_List="$home/.cookielist"
                     # So we know where the cookies are kept . . .
cFlag=""             # Part of the cookie file selection routine.

# Define the options available. Easy to change letters here if needed.
# These are the optional options; you don't just wait to be asked.

```

```
save=s   # Save command instead of executing it.
cook=c   # Change cookie file for this session.
help=h   # Usage guide.
list=l   # Pass wget the -i option and URL list.
runn=r   # Run saved commands as an argument to the option.
inpu=i   # Run saved commands interactively.
wopt=w   # Allow to enter options to pass directly to wget.
# ------------------------------------------------------------------------


if [ -z "$1" ]; then   # Make sure we get something for wget to eat.
   echo "You must at least enter a URL or option!"
   echo "-$help for usage."
   exit $E_NO_OPTS
fi



# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# added added added added added added added added added added added added
if [ ! -e "$Config" ]; then   # See if configuration file exists.
   echo "Creating configuration file, $Config"
   echo "# This is the configuration file for wgetter2" > "$Config"
   echo "# Your customised settings will be saved in this file" >> "$Config"
else
   source $Config             # Import variables we set outside the script.
fi

if [ ! -e "$Cookie_List" ]; then
   # Set up a list of cookie files, if there isn't one.
   echo "Hunting for cookies . . ."
   find -name cookies.txt >> $Cookie_List   # Create the list of cookie files.
fi #  Isolate this in its own 'if' statement,
   #+ in case we got interrupted while searching.

if [ -z "$cFlag" ]; then # If we haven't already done this . . .
   echo                  # Make a nice space after the command prompt.
   echo "Looks like you haven't set up your source of cookies yet."
   n=0                   # Make sure the counter doesn't contain random values.
   while read; do
      Cookies[$n]=$REPLY # Put the cookie files we found into an array.
      echo "$n) ${Cookies[$n]}"  # Create a menu.
      n=$(( n + 1 ))     # Increment the counter.
   done < $Cookie_List   # Feed the read statement.
   echo "Enter the number of the cookie file you want to use."
   echo "If you won't be using cookies, just press RETURN."
   echo
   echo "I won't be asking this again. Edit $Config"
   echo "If you decide to change at a later date"
   echo "or use the -${cook} option for per session changes."
   read
   if [ ! -z $REPLY ]; then   # User didn't just press return.
      Cookie=" --load-cookies ${Cookies[$REPLY]}"
      # Set the variable here as well as in the config file.

      echo "Cookie=\" --load-cookies ${Cookies[$REPLY]}\"" >> $Config
   fi
   echo "cFlag=1" >> $Config  # So we know not to ask again.
fi

# end added section end added section end added section end added section end
# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++



# Another variable.
# This one may or may not be subject to variation.
# A bit like the small print.
CookiesON=$Cookie
# echo "cookie file is $CookiesON" # For debugging.
# echo "home is ${home}"           # For debugging. Got caught with this one!


wopts()
{
echo "Enter options to pass to wget."
echo "It is assumed you know what you're doing."
echo
echo "You can pass their arguments here too."
# That is to say, everything passed here is passed to wget.

read Wopts
# Read in the options to be passed to wget.

Woptions=" $Wopts"
# Assign to another variable.
# Just for fun, or something . . .

echo "passing options ${Wopts} to wget"
# Mainly for debugging.
# Is cute.

return
}


save_func()
{
echo "Settings will be saved."
if [ ! -d $savePath ]; then  #  See if directory exists.
```

```
197    mkdir $savePath           #  Create the directory to save things in
198                              #+ if it isn't already there.
199 fi
200
201 Flag=S
202 # Tell the final bit of code what to do.
203 # Set a flag since stuff is done in main.
204
205 return
206 }
207
208
209 usage() # Tell them how it works.
210 {
211     echo "Welcome to wgetter.  This is a front end to wget."
212     echo "It will always run wget with these options:"
213     echo "$CommandA"
214     echo "and the pattern to match: $pattern (which you can change at the top of this script)."
215     echo "It will also ask you for recursion depth, and if you want to use a referring page."
216     echo "Wgetter accepts the following options:"
217     echo ""
218     echo "-$help : Display this help."
219     echo "-$save : Save the command to a file $savePath/wget-($today) instead of running it."
220     echo "-$runn : Run saved wget commands instead of starting a new one --"
221     echo "Enter filename as argument to this option."
222     echo "-$inpu : Run saved wget commands interactively --"
223     echo "The script will ask you for the filename."
224     echo "-$cook : Change the cookies file for this session."
225     echo "-$list : Tell wget to use URL's from a list instead of from the command line."
226     echo "-$wopt : Pass any other options direct to wget."
227     echo ""
228     echo "See the wget man page for additional options you can pass to wget."
229     echo ""
230
231     exit $E_USAGE  # End here. Don't process anything else.
232 }
233
234
235
236 list_func() #  Gives the user the option to use the -i option to wget,
237             #+ and a list of URLs.
238 {
239 while [ 1 ]; do
240    echo "Enter the name of the file containing URL's (press q to change your
241 mind)."
242    read urlfile
243    if [ ! -e "$urlfile" ] && [ "$urlfile" != q ]; then
244        # Look for a file, or the quit option.
245        echo "That file does not exist!"
246    elif [ "$urlfile" = q ]; then # Check quit option.
247        echo "Not using a url list."
248        return
249    else
250       echo "using $urlfile."
251       echo "If you gave me url's on the command line, I'll use those first."
252                              # Report wget standard behaviour to the user.
253       lister=" -i $urlfile" # This is what we want to pass to wget.
254       return
255    fi
256 done
257 }
258
259
260 cookie_func() # Give the user the option to use a different cookie file.
261 {
262 while [ 1 ]; do
263    echo "Change the cookies file. Press return if you don't want to change
264 it."
265    read Cookies
266    # NB: this is not the same as Cookie, earlier.
267    # There is an 's' on the end.
268    # Bit like chocolate chips.
269    if [ -z "$Cookies" ]; then                # Escape clause for wusses.
270       return
271    elif [ ! -e "$Cookies" ]; then
272       echo "File does not exist.  Try again." # Keep em going . . .
273    else
274        CookiesON=" --load-cookies $Cookies"   # File is good -- let's use it!
275        return
276    fi
277 done
278 }
279
280
281
282 run_func()
283 {
284 if [ -z "$OPTARG" ]; then
285 # Test to see if we used the in-line option or the query one.
286    if [ ! -d "$savePath" ]; then      # In case directory doesn't exist . . .
287       echo "$savePath does not appear to exist."
288       echo "Please supply path and filename of saved wget commands:"
289       read newFile
290          until [ -f "$newFile" ]; do  # Keep going till we get something.
291             echo "Sorry, that file does not exist.  Please try again."
292             # Try really hard to get something.
293             read newFile
294          done
295
296
297 # ----------------------------------------------------------------------------
298 #         if [ -z ( grep wget ${newfile} ) ]; then
```

```
          # Assume they haven't got the right file and bail out.
#         echo "Sorry, that file does not contain wget commands.  Aborting."
#         exit
#         fi
#
# This is bogus code.
# It doesn't actually work.
# If anyone wants to fix it, feel free!
# ------------------------------------------------------------------------


      filePath="${newFile}"
   else
   echo "Save path is $savePath"
      echo "Please enter name of the file which you want to use."
      echo "You have a choice of:"
      ls $savePath                                  # Give them a choice.
      read inFile
         until [ -f "$savePath/$inFile" ]; do       # Keep going till we get something.
            if [ ! -f "${savePath}/${inFile}" ]; then # If file doesn't exist.
               echo "Sorry, that file does not exist.  Please choose from:"
               ls $savePath                         # If a mistake is made.
               read inFile
            fi
         done
      filePath="${savePath}/${inFile}"  # Make one variable . . .
   fi
else filePath="${savePath}/${OPTARG}"   # Which can be many things . . .
fi

if [ ! -f "$filePath" ]; then           # If a bogus file got through.
   echo "You did not specify a suitable file."
   echo "Run this script with the -${save} option first."
   echo "Aborting."
   exit $E_NO_SAVEFILE
fi
echo "Using: $filePath"
while read; do
    eval $REPLY
    echo "Completed: $REPLY"
done < $filePath  # Feed the actual file we are using into a 'while' loop.

exit
}



# Fish out any options we are using for the script.
# This is based on the demo in "Learning The Bash Shell" (O'Reilly).
while getopts ":$save$cook$help$list$runn:$inpu$wopt" opt
do
  case $opt in
     $save) save_func;;   #  Save some wgetter sessions for later.
     $cook) cookie_func;; #  Change cookie file.
     $help) usage;;       #  Get help.
     $list) list_func;;   #  Allow wget to use a list of URLs.
     $runn) run_func;;    #  Useful if you are calling wgetter from, for example,
                          #+ a cron script.
     $inpu) run_func;;    #  When you don't know what your files are named.
     $wopt) wopts;;       #  Pass options directly to wget.
        \?) echo "Not a valid option."
            echo "Use -${wopt} if you want to pass options directly to wget,"
            echo "or -${help} for help";;      # Catch anything else.
  esac
done
shift $((OPTIND - 1))     # Do funky magic stuff with $#.


if [ -z "$1" ] && [ -z "$lister" ]; then
                          #  We should be left with at least one URL
                          #+ on the command line, unless a list is
                       #+ being used -- catch empty CL's.
   echo "No URL's given!  You must enter them on the same line as wgetter2."
   echo "E.g.,  wgetter2 http://somesite http://anothersite."
   echo "Use $help option for more information."
   exit $E_NO_URLS        # Bail out, with appropriate error code.
fi

URLS=" $@"
# Use this so that URL list can be changed if we stay in the option loop.

while [ 1 ]; do
   # This is where we ask for the most used options.
   # (Mostly unchanged from version 1 of wgetter)
   if [ -z $curDepth ]; then
      Current=""
   else Current=" Current value is $curDepth"
   fi
       echo "How deep should I go? (integer: Default is $depthDefault.$Current)"
       read Depth   # Recursion -- how far should we go?
       inputB=""    # Reset this to blank on each pass of the loop.
       echo "Enter the name of the referring page (default is none)."
       read inputB  # Need this for some sites.

       echo "Do you want to have the output logged to the terminal"
       echo "(y/n, default is yes)?"
       read noHide  # Otherwise wget will just log it to a file.

       case $noHide in    # Now you see me, now you don't.
          y|Y ) hide="";;
          n|N ) hide=" -b";;
            * ) hide="";;
```

```
401         esac
402
403         if [ -z ${Depth} ]; then       #  User accepted either default or current depth,
404                                        #+ in which case Depth is now empty.
405            if [ -z ${curDepth} ]; then #  See if a depth was set on a previous iteration.
406               Depth="$depthDefault"    #  Set the default recursion depth if nothing
407                                        #+ else to use.
408            else Depth="$curDepth"      #  Otherwise, set the one we used before.
409            fi
410         fi
411    Recurse=" -l $Depth"                # Set how deep we want to go.
412    curDepth=$Depth                     # Remember setting for next time.
413
414         if [ ! -z $inputB ]; then
415            RefA=" --referer=$inputB"   # Option to use referring page.
416         fi
417
418    WGETTER="${CommandA}${pattern}${hide}${RefA}${Recurse}${CookiesON}${lister}${Woptions}${URLS}"
419    #  Just string the whole lot together . . .
420    #  NB: no embedded spaces.
421    #  They are in the individual elements so that if any are empty,
422    #+ we don't get an extra space.
423
424    if [ -z "${CookiesON}" ] && [ "$cFlag" = "1" ] ; then
425        echo "Warning -- can't find cookie file"
426        # This should be changed, in case the user has opted to not use cookies.
427    fi
428
429    if [ "$Flag" = "S" ]; then
430       echo "$WGETTER" >> $savePath/wget-${today}
431       #  Create a unique filename for today, or append to it if it exists.
432       echo "$inputB" >> $savePath/site-list-${today}
433       #  Make a list, so it's easy to refer back to,
434       #+ since the whole command is a bit confusing to look at.
435       echo "Command saved to the file $savePath/wget-${today}"
436            # Tell the user.
437       echo "Referring page URL saved to the file $savePath/site-list-${today}"
438            # Tell the user.
439       Saver=" with save option"
440       # Stick this somewhere, so it appears in the loop if set.
441    else
442        echo "*****************"
443        echo "*****Getting*****"
444        echo "*****************"
445        echo ""
446        echo "$WGETTER"
447        echo ""
448        echo "*****************"
449        eval "$WGETTER"
450    fi
451
452        echo ""
453        echo "Starting over$Saver."
454        echo "If you want to stop, press q."
455        echo "Otherwise, enter some URL's:"
456        # Let them go again. Tell about save option being set.
457
458        read
459        case $REPLY in                 # Need to change this to a 'trap' clause.
460           q|Q ) exit $E_USER_EXIT;;  # Exercise for the reader?
461              * ) URLS=" $REPLY";;
462        esac
463
464        echo ""
465 done
466
467
468 exit 0
```

例子 A-31. 一个"podcasting"(译者: 指的是在互联网上发布音视频文件, 并允许用户订阅并自动接收的方法)脚本

```
 1 #!/bin/bash
 2
 3 # bashpodder.sh:
 4 # By Linc 10/1/2004
 5 # Find the latest script at http://linc.homeunix.org:8080/scripts/bashpodder
 6 # Last revision 12/14/2004 - Many Contributors!
 7 # If you use this and have made improvements or have comments
 8 # drop me an email at linc dot fessenden at gmail dot com
 9 # I'd appreciate it!
10
11 # ==>  ABS Guide extra comments.
12
13 # ==>  Author of this script has kindly granted permission
14 # ==>+ for inclusion in ABS Guide.
15
16
17 # ==> ######################################################################
18 #
19 # ==> What is "podcasting"?
20
21 # ==> It's broadcasting "radio shows" over the Internet.
22 # ==> These shows can be played on iPods and other music file players.
23
24 # ==> This script makes it possible.
25 # ==> See documentation at the script author's site, above.
26
27 # ==> ######################################################################
```

```
28
29
30 # Make script crontab friendly:
31 cd $(dirname $0)
32 # ==> Change to directory where this script lives.
33
34 # datadir is the directory you want podcasts saved to:
35 datadir=$(date +%Y-%m-%d)
36 # ==> Will create a directory with the name: YYYY-MM-DD
37
38 # Check for and create datadir if necessary:
39 if test ! -d $datadir
40         then
41         mkdir $datadir
42 fi
43
44 # Delete any temp file:
45 rm -f temp.log
46
47 # Read the bp.conf file and wget any url not already in the podcast.log file:
48 while read podcast
49         do # ==> Main action follows.
50         file=$(wget -q $podcast -O - | tr '\r' '\n' | tr \' \" | sed -n
's/.*url="\([^"]*\)".*/\1/p')
51         for url in $file
52                 do
53                 echo $url >> temp.log
54                 if ! grep "$url" podcast.log > /dev/null
55                         then
56                         wget -q -P $datadir "$url"
57                 fi
58                 done
59         done < bp.conf
60
61 # Move dynamically created log file to permanent log file:
62 cat podcast.log >> temp.log
63 sort temp.log | uniq > podcast.log
64 rm temp.log
65 # Create an m3u playlist:
66 ls $datadir | grep -v m3u > $datadir/podcast.m3u
67
68
69 exit 0
70
71 ################################################
72 For a different scripting approach to Podcasting,
73 see Phil Salkie's article,
74 "Internet Radio to Podcast with Shell Tools"
75 in the September, 2005 issue of LINUX JOURNAL,
76 http://www.linuxjournal.com/article/8171
77 ################################################
```

作为本小节的结尾, 让我们回顾一下基本概念 . . . 可能还有些扩展部分.

例子 A-32. 基础回顾

```
 1 #!/bin/bash
 2 # basics-reviewed.bash
 3
 4 # File extension == *.bash == specific to Bash
 5
 6 #   Copyright (c) Michael S. Zick, 2003; All rights reserved.
 7 #   License: Use in any form, for any purpose.
 8 #   Revision: $ID$
 9 #
10 #             Edited for layout by M.C.
11 #   (author of the "Advanced Bash Scripting Guide")
12
13
14 #  This script tested under Bash versions 2.04, 2.05a and 2.05b.
15 #  It may not work with earlier versions.
16 #  This demonstration script generates one --intentional--
17 #+ "command not found" error message. See line 394.
18
19 #  The current Bash maintainer, Chet Ramey, has fixed the items noted
20 #+ for an upcoming version of Bash.
21
22
23
24         ###-------------------------------------------###
25         ###  Pipe the output of this script to 'more' ###
26         ###+ else it will scroll off the page.        ###
27         ###                                           ###
28         ###  You may also redirect its output         ###
29         ###+ to a file for examination.               ###
30         ###-------------------------------------------###
31
32
33
34 #  Most of the following points are described at length in
35 #+ the text of the foregoing "Advanced Bash Scripting Guide."
36 #  This demonstration script is mostly just a reorganized presentation.
37 #      -- msz
38
39 # Variables are not typed unless otherwise specified.
40
41 #  Variables are named. Names must contain a non-digit.
```

```
#  File descriptor names (as in, for example: 2>&1)
#+ contain ONLY digits.

# Parameters and Bash array elements are numbered.
# (Parameters are very similar to Bash arrays.)

# A variable name may be undefined (null reference).
unset VarNull

# A variable name may be defined but empty (null contents).
VarEmpty=''                           # Two, adjacent, single quotes.

# A variable name my be defined and non-empty
VarSomething='Literal'

# A variable may contain:
#   * A whole number as a signed 32-bit (or larger) integer
#   * A string
# A variable may also be an array.

#  A string may contain embedded blanks and may be treated
#+ as if it where a function name with optional arguments.

#  The names of variables and the names of functions
#+ are in different namespaces.


#  A variable may be defined as a Bash array either explicitly or
#+ implicitly by the syntax of the assignment statement.
#  Explicit:
declare -a ArrayVar



# The echo command is a built-in.
echo $VarSomething

# The printf command is a built-in.
# Translate %s as: String-Format
printf %s $VarSomething         # No linebreak specified, none output.
echo                            # Default, only linebreak output.




# The Bash parser word breaks on whitespace.
# Whitespace, or the lack of it is significant.
# (This holds true in general; there are, of course, exceptions.)



# Translate the DOLLAR_SIGN character as: Content-Of.

# Extended-Syntax way of writing Content-Of:
echo ${VarSomething}

#  The ${ ... } Extended-Syntax allows more than just the variable
#+ name to be specified.
#  In general, $VarSomething can always be written as: ${VarSomething}.

# Call this script with arguments to see the following in action.



#  Outside of double-quotes, the special characters @ and *
#+ specify identical behavior.
#  May be pronounced as: All-Elements-Of.

#  Without specification of a name, they refer to the
#+ pre-defined parameter Bash-Array.



# Glob-Pattern references
echo $*                         # All parameters to script or function
echo ${*}                       # Same

# Bash disables filename expansion for Glob-Patterns.
# Only character matching is active.


# All-Elements-Of references
echo $@                         # Same as above
echo ${@}                       # Same as above




#  Within double-quotes, the behavior of Glob-Pattern references
#+ depends on the setting of IFS (Input Field Separator).
#  Within double-quotes, All-Elements-Of references behave the same.


#  Specifying only the name of a variable holding a string refers
#+ to all elements (characters) of a string.


#  To specify an element (character) of a string,
#+ the Extended-Syntax reference notation (see below) MAY be used.


```

```
144
145
146 #  Specifying only the name of a Bash array references
147 #+ the subscript zero element,
148 #+ NOT the FIRST DEFINED nor the FIRST WITH CONTENTS element.
149
150 #  Additional qualification is needed to reference other elements,
151 #+ which means that the reference MUST be written in Extended-Syntax.
152 #  The general form is: ${name[subscript]}.
153
154 #  The string forms may also be used: ${name:subscript}
155 #+ for Bash-Arrays when referencing the subscript zero element.
156
157
158 # Bash-Arrays are implemented internally as linked lists,
159 #+ not as a fixed area of storage as in some programming languages.
160
161
162 #   Characteristics of Bash arrays (Bash-Arrays):
163 #   ---------------------------------------------
164
165 #   If not otherwise specified, Bash-Array subscripts begin with
166 #+  subscript number zero. Literally: [0]
167 #   This is called zero-based indexing.
168 ###
169 #   If not otherwise specified, Bash-Arrays are subscript packed
170 #+  (sequential subscripts without subscript gaps).
171 ###
172 #   Negative subscripts are not allowed.
173 ###
174 #   Elements of a Bash-Array need not all be of the same type.
175 ###
176 #   Elements of a Bash-Array may be undefined (null reference).
177 #       That is, a Bash-Array my be "subscript sparse."
178 ###
179 #   Elements of a Bash-Array may be defined and empty (null contents).
180 ###
181 #   Elements of a Bash-Array may contain:
182 #     * A whole number as a signed 32-bit (or larger) integer
183 #     * A string
184 #     * A string formated so that it appears to be a function name
185 #     + with optional arguments
186 ###
187 #   Defined elements of a Bash-Array may be undefined (unset).
188 #       That is, a subscript packed Bash-Array may be changed
189 #   +   into a subscript sparse Bash-Array.
190 ###
191 #   Elements may be added to a Bash-Array by defining an element
192 #+  not previously defined.
193 ###
194 # For these reasons, I have been calling them "Bash-Arrays".
195 # I'll return to the generic term "array" from now on.
196 #     -- msz
197
198
199
200
201 #  Demo time -- initialize the previously declared ArrayVar as a
202 #+ sparse array.
203 #  (The 'unset ... ' is just documentation here.)
204
205 unset ArrayVar[0]                   # Just for the record
206 ArrayVar[1]=one                     # Unquoted literal
207 ArrayVar[2]=''                      # Defined, and empty
208 unset ArrayVar[3]                   # Just for the record
209 ArrayVar[4]='four'                  # Quoted literal
210
211
212
213 # Translate the %q format as: Quoted-Respecting-IFS-Rules.
214 echo
215 echo '- - Outside of double-quotes - -'
216 ###
217 printf %q ${ArrayVar[*]}            # Glob-Pattern All-Elements-Of
218 echo
219 echo 'echo command:'${ArrayVar[*]}
220 ###
221 printf %q ${ArrayVar[@]}            # All-Elements-Of
222 echo
223 echo 'echo command:'${ArrayVar[@]}
224
225 # The use of double-quotes may be translated as: Enable-Substitution.
226
227 # There are five cases recognized for the IFS setting.
228
229 echo
230 echo '- - Within double-quotes - Default IFS of space-tab-newline - -'
231 IFS=$'\x20'$'\x09'$'\x0A'           #  These three bytes,
232                                      #+ in exactly this order.
233
234
235 printf %q "${ArrayVar[*]}"          # Glob-Pattern All-Elements-Of
236 echo
237 echo 'echo command:'"${ArrayVar[*]}"
238 ###
239 printf %q "${ArrayVar[@]}"          # All-Elements-Of
240 echo
241 echo 'echo command:'"${ArrayVar[@]}"
242
243
244 echo
245 echo '- - Within double-quotes - First character of IFS is ^ - -'
```

```
# Any printing, non-whitespace character should do the same.
IFS='^'$IFS                          # ^ + space tab newline
###
printf %q "${ArrayVar[*]}"           # Glob-Pattern All-Elements-Of
echo
echo 'echo command:'"${ArrayVar[*]}"
###
printf %q "${ArrayVar[@]}"           # All-Elements-Of
echo
echo 'echo command:'"${ArrayVar[@]}"


echo
echo '- - Within double-quotes - Without whitespace in IFS - -'
IFS='^:%!'
###
printf %q "${ArrayVar[*]}"           # Glob-Pattern All-Elements-Of
echo
echo 'echo command:'"${ArrayVar[*]}"
###
printf %q "${ArrayVar[@]}"           # All-Elements-Of
echo
echo 'echo command:'"${ArrayVar[@]}"


echo
echo '- - Within double-quotes - IFS set and empty - -'
IFS=''
###
printf %q "${ArrayVar[*]}"           # Glob-Pattern All-Elements-Of
echo
echo 'echo command:'"${ArrayVar[*]}"
###
printf %q "${ArrayVar[@]}"           # All-Elements-Of
echo
echo 'echo command:'"${ArrayVar[@]}"


echo
echo '- - Within double-quotes - IFS undefined - -'
unset IFS
###
printf %q "${ArrayVar[*]}"           # Glob-Pattern All-Elements-Of
echo
echo 'echo command:'"${ArrayVar[*]}"
###
printf %q "${ArrayVar[@]}"           # All-Elements-Of
echo
echo 'echo command:'"${ArrayVar[@]}"


# Put IFS back to the default.
# Default is exactly these three bytes.
IFS=$'\x20'$'\x09'$'\x0A'            # In exactly this order.

# Interpretation of the above outputs:
#   A Glob-Pattern is I/O; the setting of IFS matters.
###
#   An All-Elements-Of does not consider IFS settings.
###
#   Note the different output using the echo command and the
#+  quoted format operator of the printf command.


#  Recall:
#   Parameters are similar to arrays and have the similar behaviors.
###
#  The above examples demonstrate the possible variations.
#  To retain the shape of a sparse array, additional script
#+ programming is required.
###
#  The source code of Bash has a routine to output the
#+ [subscript]=value   array assignment format.
#  As of version 2.05b, that routine is not used,
#+ but that might change in future releases.



# The length of a string, measured in non-null elements (characters):
echo
echo '- - Non-quoted references - -'
echo 'Non-Null character count: '${#VarSomething}' characters.'

# test='Lit'$'\x00''eral'            # $'\x00' is a null character.
# echo ${#test}                       # See that?



#  The length of an array, measured in defined elements,
#+ including null content elements.
echo
echo 'Defined content count: '${#ArrayVar[@]}' elements.'
# That is NOT the maximum subscript (4).
# That is NOT the range of the subscripts (1 . . 4 inclusive).
# It IS the length of the linked list.
###
#  Both the maximum subscript and the range of the subscripts may
#+ be found with additional script programming.

# The length of a string, measured in non-null elements (characters):
echo
echo '- - Quoted, Glob-Pattern references - -'
```

```
echo 'Non-Null character count: '"${#VarSomething}"' characters.'

#  The length of an array, measured in defined elements,
#+ including null-content elements.
echo
echo 'Defined element count: '"${#ArrayVar[*]}"' elements.'

#  Interpretation: Substitution does not effect the ${# ... } operation.
#  Suggestion:
#  Always use the All-Elements-Of character
#+ if that is what is intended (independence from IFS).



#  Define a simple function.
#  I include an underscore in the name
#+ to make it distinctive in the examples below.
###
#  Bash separates variable names and function names
#+ in different namespaces.
#  The Mark-One eyeball isn't that advanced.
###
_simple() {
    echo -n 'SimpleFunc'$@          #  Newlines are swallowed in
}                                   #+ result returned in any case.


# The ( ... ) notation invokes a command or function.
# The $( ... ) notation is pronounced: Result-Of.


# Invoke the function _simple
echo
echo '- - Output of function _simple - -'
_simple                             # Try passing arguments.
echo
# or
(_simple)                           # Try passing arguments.
echo

echo '- Is there a variable of that name? -'
echo $_simple not defined           # No variable by that name.

# Invoke the result of function _simple (Error msg intended)

###
$(_simple)                          # Gives an error message:
#                           line 394: SimpleFunc: command not found
#                           ---------------------------------------

echo
###

#  The first word of the result of function _simple
#+ is neither a valid Bash command nor the name of a defined function.
###
# This demonstrates that the output of _simple is subject to evaluation.
###
# Interpretation:
#   A function can be used to generate in-line Bash commands.


# A simple function where the first word of result IS a bash command:
###
_print() {
    echo -n 'printf %q '$@
}

echo '- - Outputs of function _print - -'
_print parm1 parm2                  # An Output NOT A Command.
echo

$(_print parm1 parm2)               #  Executes: printf %q parm1 parm2
                                    #  See above IFS examples for the
                                    #+ various possibilities.
echo

$(_print $VarSomething)             # The predictable result.
echo



# Function variables
# ------------------

echo
echo '- - Function variables - -'
# A variable may represent a signed integer, a string or an array.
# A string may be used like a function name with optional arguments.

# set -vx                           #  Enable if desired
declare -f funcVar                  #+ in namespace of functions

funcVar=_print                      # Contains name of function.
$funcVar parm1                      # Same as _print at this point.
echo

funcVar=$(_print )                  # Contains result of function.
$funcVar                            # No input, No output.
$funcVar $VarSomething              # The predictable result.
echo

```

```
450 funcVar=$(_print $VarSomething)     #  $VarSomething replaced HERE.
451 $funcVar                            #  The expansion is part of the
452 echo                                #+ variable contents.
453
454 funcVar="$(_print $VarSomething)"   #  $VarSomething replaced HERE.
455 $funcVar                            #  The expansion is part of the
456 echo                                #+ variable contents.
457
458 #  The difference between the unquoted and the double-quoted versions
459 #+ above can be seen in the "protect_literal.sh" example.
460 #  The first case above is processed as two, unquoted, Bash-Words.
461 #  The second case above is processed as one, quoted, Bash-Word.
462
463
464
465
466 # Delayed replacement
467 # -------------------
468
469 echo
470 echo '- - Delayed replacement - -'
471 funcVar="$(_print '$VarSomething')" # No replacement, single Bash-Word.
472 eval $funcVar                       # $VarSomething replaced HERE.
473 echo
474
475 VarSomething='NewThing'
476 eval $funcVar                       # $VarSomething replaced HERE.
477 echo
478
479 # Restore the original setting trashed above.
480 VarSomething=Literal
481
482 #  There are a pair of functions demonstrated in the
483 #+ "protect_literal.sh" and "unprotect_literal.sh" examples.
484 #  These are general purpose functions for delayed replacement literals
485 #+ containing variables.
486
487
488
489
490
491 # REVIEW:
492 # ------
493
494 #  A string can be considered a Classic-Array of elements (characters).
495 #  A string operation applies to all elements (characters) of the string
496 #+ (in concept, anyway).
497 ###
498 #  The notation: ${array_name[@]} represents all elements of the
499 #+ Bash-Array: array_name.
500 ###
501 #  The Extended-Syntax string operations can be applied to all
502 #+ elements of an array.
503 ###
504 #  This may be thought of as a For-Each operation on a vector of strings.
505 ###
506 #  Parameters are similar to an array.
507 #  The initialization of a parameter array for a script
508 #+ and a parameter array for a function only differ
509 #+ in the initialization of ${0}, which never changes its setting.
510 ###
511 #  Subscript zero of the script's parameter array contains
512 #+ the name of the script.
513 ###
514 #  Subscript zero of a function's parameter array DOES NOT contain
515 #+ the name of the function.
516 #  The name of the current function is accessed by the $FUNCNAME variable.
517 ###
518 #  A quick, review list follows (quick, not short).
519
520 echo
521 echo '- - Test (but not change) - -'
522 echo '- null reference -'
523 echo -n ${VarNull-'NotSet'}' '          # NotSet
524 echo ${VarNull}                         # NewLine only
525 echo -n ${VarNull:-'NotSet'}' '         # NotSet
526 echo ${VarNull}                         # Newline only
527
528 echo '- null contents -'
529 echo -n ${VarEmpty-'Empty'}' '          # Only the space
530 echo ${VarEmpty}                        # Newline only
531 echo -n ${VarEmpty:-'Empty'}' '         # Empty
532 echo ${VarEmpty}                        # Newline only
533
534 echo '- contents -'
535 echo ${VarSomething-'Content'}          # Literal
536 echo ${VarSomething:-'Content'}         # Literal
537
538 echo '- Sparse Array -'
539 echo ${ArrayVar[@]-'not set'}
540
541 # ASCII-Art time
542 # State     Y==yes, N==no
543 #           -       :-
544 # Unset     Y       Y       ${# ... } == 0
545 # Empty     N       Y       ${# ... } == 0
546 # Contents  N       N       ${# ... } > 0
547
548 #  Either the first and/or the second part of the tests
549 #+ may be a command or a function invocation string.
550 echo
551 echo '- - Test 1 for undefined - -'
```

```
declare -i t
_decT() {
    t=$t-1
}

# Null reference, set: t == -1
t=${#VarNull}                           # Results in zero.
${VarNull- _decT }                      # Function executes, t now -1.
echo $t

# Null contents, set: t == 0
t=${#VarEmpty}                          # Results in zero.
${VarEmpty- _decT }                     # _decT function NOT executed.
echo $t

# Contents, set: t == number of non-null characters
VarSomething='_simple'                  # Set to valid function name.
t=${#VarSomething}                      # non-zero length
${VarSomething- _decT }                 # Function _simple executed.
echo $t                                 # Note the Append-To action.

# Exercise: clean up that example.
unset t
unset _decT
VarSomething=Literal

echo
echo '- - Test and Change - -'
echo '- Assignment if null reference -'
echo -n ${VarNull='NotSet'}' '          # NotSet NotSet
echo ${VarNull}
unset VarNull

echo '- Assignment if null reference -'
echo -n ${VarNull:='NotSet'}' '         # NotSet NotSet
echo ${VarNull}
unset VarNull

echo '- No assignment if null contents -'
echo -n ${VarEmpty='Empty'}' '          # Space only
echo ${VarEmpty}
VarEmpty=''

echo '- Assignment if null contents -'
echo -n ${VarEmpty:='Empty'}' '         # Empty Empty
echo ${VarEmpty}
VarEmpty=''

echo '- No change if already has contents -'
echo ${VarSomething='Content'}          # Literal
echo ${VarSomething:='Content'}         # Literal


# "Subscript sparse" Bash-Arrays
###
#  Bash-Arrays are subscript packed, beginning with
#+ subscript zero unless otherwise specified.
###
#  The initialization of ArrayVar was one way
#+ to "otherwise specify".  Here is the other way:
###
echo
declare -a ArraySparse
ArraySparse=( [1]=one [2]='' [4]='four' )
# [0]=null reference, [2]=null content, [3]=null reference

echo '- - Array-Sparse List - -'
# Within double-quotes, default IFS, Glob-Pattern

IFS=$'\x20'$'\x09'$'\x0A'
printf %q "${ArraySparse[*]}"
echo

#  Note that the output does not distinguish between "null content"
#+ and "null reference".
#  Both print as escaped whitespace.
###
#  Note also that the output does NOT contain escaped whitespace
#+ for the "null reference(s)" prior to the first defined element.
###
# This behavior of 2.04, 2.05a and 2.05b has been reported
#+ and may change in a future version of Bash.

#  To output a sparse array and maintain the [subscript]=value
#+ relationship without change requires a bit of programming.
#  One possible code fragment:
###
# local l=${#ArraySparse[@]}        # Count of defined elements
# local f=0                          # Count of found subscripts
# local i=0                          # Subscript to test
(                                    # Anonymous in-line function
    for (( l=${#ArraySparse[@]}, f = 0, i = 0 ; f < l ; i++ ))
    do
        # 'if defined then...'
        ${ArraySparse[$i]+ eval echo '\ ['$i']='${ArraySparse[$i]} ; (( f++ )) }
    done
)

# The reader coming upon the above code fragment cold
#+ might want to review "command lists" and "multiple commands on a line"
#+ in the text of the foregoing "Advanced Bash Scripting Guide."
###
```

```
#  Note:
#  The "read -a array_name" version of the "read" command
#+ begins filling array_name at subscript zero.
#  ArraySparse does not define a value at subscript zero.
###
#  The user needing to read/write a sparse array to either
#+ external storage or a communications socket must invent
#+ a read/write code pair suitable for their purpose.
###
# Exercise: clean it up.

unset ArraySparse

echo
echo '- - Conditional alternate (But not change)- -'
echo '- No alternate if null reference -'
echo -n ${VarNull+'NotSet'}' '
echo ${VarNull}
unset VarNull

echo '- No alternate if null reference -'
echo -n ${VarNull:+'NotSet'}' '
echo ${VarNull}
unset VarNull

echo '- Alternate if null contents -'
echo -n ${VarEmpty+'Empty'}' '               # Empty
echo ${VarEmpty}
VarEmpty=''

echo '- No alternate if null contents -'
echo -n ${VarEmpty:+'Empty'}' '              # Space only
echo ${VarEmpty}
VarEmpty=''

echo '- Alternate if already has contents -'

# Alternate literal
echo -n ${VarSomething+'Content'}' '         # Content Literal
echo ${VarSomething}

# Invoke function
echo -n ${VarSomething:+ $(_simple) }' '     # SimpleFunc Literal
echo ${VarSomething}
echo

echo '- - Sparse Array - -'
echo ${ArrayVar[@]+'Empty'}                  # An array of 'Empty'(ies)
echo

echo '- - Test 2 for undefined - -'

declare -i t
_incT() {
    t=$t+1
}

#  Note:
#  This is the same test used in the sparse array
#+ listing code fragment.

# Null reference, set: t == -1
t=${#VarNull}-1                      # Results in minus-one.
${VarNull+ _incT }                   # Does not execute.
echo $t' Null reference'

# Null contents, set: t == 0
t=${#VarEmpty}-1                     # Results in minus-one.
${VarEmpty+ _incT }                  # Executes.
echo $t'  Null content'

# Contents, set: t == (number of non-null characters)
t=${#VarSomething}-1                 # non-null length minus-one
${VarSomething+ _incT }              # Executes.
echo $t'  Contents'

# Exercise: clean up that example.
unset t
unset _incT

# ${name?err_msg} ${name:?err_msg}
#  These follow the same rules but always exit afterwards
#+ if an action is specified following the question mark.
#  The action following the question mark may be a literal
#+ or a function result.
###
#  ${name?} ${name:?} are test-only, the return can be tested.



# Element operations
# ------------------

echo
echo '- - Trailing sub-element selection - -'

#  Strings, Arrays and Positional parameters

#  Call this script with multiple arguments
#+ to see the parameter selections.

```

```
echo '- All -'
echo ${VarSomething:0}              # all non-null characters
echo ${ArrayVar[@]:0}               # all elements with content
echo ${@:0}                          # all parameters with content;
                                     # ignoring parameter[0]

echo
echo '- All after -'
echo ${VarSomething:1}              # all non-null after character[0]
echo ${ArrayVar[@]:1}               # all after element[0] with content
echo ${@:2}                          # all after param[1] with content

echo
echo '- Range after -'
echo ${VarSomething:4:3}            # ral
                                     # Three characters after
                                     # character[3]

echo '- Sparse array gotch -'
echo ${ArrayVar[@]:1:2}     #  four - The only element with content.
                             #  Two elements after (if that many exist).
                             #  the FIRST WITH CONTENTS
                             #+ (the FIRST WITH  CONTENTS is being
                             #+ considered as if it
                             #+ were subscript zero).
#  Executed as if Bash considers ONLY array elements with CONTENT
#  printf %q "${ArrayVar[@]:0:3}"    # Try this one

#  In versions 2.04, 2.05a and 2.05b,
#+ Bash does not handle sparse arrays as expected using this notation.
#
#  The current Bash maintainer, Chet Ramey, has corrected this
#+ for an upcoming version of Bash.


echo '- Non-sparse array -'
echo ${@:2:2}               # Two parameters following parameter[1]

# New victims for string vector examples:
stringZ=abcABC123ABCabc
arrayZ=( abcabc ABCABC 123123 ABCABC abcabc )
sparseZ=( [1]='abcabc' [3]='ABCABC' [4]='' [5]='123123' )

echo
echo ' - - Victim string - -'$stringZ'- - '
echo ' - - Victim array - -'${arrayZ[@]}'- - '
echo ' - - Sparse array - -'${sparseZ[@]}'- - '
echo ' - [0]==null ref, [2]==null ref, [4]==null content - '
echo ' - [1]=abcabc [3]=ABCABC [5]=123123 - '
echo ' - non-null-reference count: '${#sparseZ[@]}' elements'

echo
echo '- - Prefix sub-element removal - -'
echo '- - Glob-Pattern match must include the first character. - -'
echo '- - Glob-Pattern may be a literal or a function result. - -'
echo


# Function returning a simple, Literal, Glob-Pattern
_abc() {
    echo -n 'abc'
}

echo '- Shortest prefix -'
echo ${stringZ#123}                 # Unchanged (not a prefix).
echo ${stringZ#$(_abc)}             # ABC123ABCabc
echo ${arrayZ[@]#abc}               # Applied to each element.

# Fixed by Chet Ramey for an upcoming version of Bash.
# echo ${sparseZ[@]#abc}            # Version-2.05b core dumps.

# The -it would be nice- First-Subscript-Of
# echo ${#sparseZ[@]#*}             # This is NOT valid Bash.

echo
echo '- Longest prefix -'
echo ${stringZ##1*3}                # Unchanged (not a prefix)
echo ${stringZ##a*C}                # abc
echo ${arrayZ[@]##a*c}              # ABCABC 123123 ABCABC

# Fixed by Chet Ramey for an upcoming version of Bash
# echo ${sparseZ[@]##a*c}           # Version-2.05b core dumps.

echo
echo '- - Suffix sub-element removal - -'
echo '- - Glob-Pattern match must include the last character. - -'
echo '- - Glob-Pattern may be a literal or a function result. - -'
echo
echo '- Shortest suffix -'
echo ${stringZ%1*3}                 # Unchanged (not a suffix).
echo ${stringZ%$(_abc)}             # abcABC123ABC
echo ${arrayZ[@]%abc}               # Applied to each element.

# Fixed by Chet Ramey for an upcoming version of Bash.
# echo ${sparseZ[@]%abc}            # Version-2.05b core dumps.

# The -it would be nice- Last-Subscript-Of
# echo ${#sparseZ[@]%*}             # This is NOT valid Bash.

echo
echo '- Longest suffix -'
echo ${stringZ%%1*3}                # Unchanged (not a suffix)
```

```
858 echo ${stringZ%%b*c}               # a
859 echo ${arrayZ[@]%%b*c}              # a ABCABC 123123 ABCABC a
860
861 # Fixed by Chet Ramey for an upcoming version of Bash.
862 # echo ${sparseZ[@]%%b*c}           # Version-2.05b core dumps.
863
864 echo
865 echo '- - Sub-element replacement - -'
866 echo '- - Sub-element at any location in string. - -'
867 echo '- - First specification is a Glob-Pattern - -'
868 echo '- - Glob-Pattern may be a literal or Glob-Pattern function result. - -'
869 echo '- - Second specification may be a literal or function result. - -'
870 echo '- - Second specification may be unspecified. Pronounce that'
871 echo '    as: Replace-With-Nothing (Delete) - -'
872 echo
873
874
875
876 # Function returning a simple, Literal, Glob-Pattern
877 _123() {
878     echo -n '123'
879 }
880
881 echo '- Replace first occurrence -'
882 echo ${stringZ/$(_123)/999}         # Changed (123 is a component).
883 echo ${stringZ/ABC/xyz}             # xyzABC123ABCabc
884 echo ${arrayZ[@]/ABC/xyz}           # Applied to each element.
885 echo ${sparseZ[@]/ABC/xyz}          # Works as expected.
886
887 echo
888 echo '- Delete first occurrence -'
889 echo ${stringZ/$(_123)/}
890 echo ${stringZ/ABC/}
891 echo ${arrayZ[@]/ABC/}
892 echo ${sparseZ[@]/ABC/}
893
894 #  The replacement need not be a literal,
895 #+ since the result of a function invocation is allowed.
896 #  This is general to all forms of replacement.
897 echo
898 echo '- Replace first occurrence with Result-Of -'
899 echo ${stringZ/$(_123)/$(_simple)}  # Works as expected.
900 echo ${arrayZ[@]/ca/$(_simple)}     # Applied to each element.
901 echo ${sparseZ[@]/ca/$(_simple)}    # Works as expected.
902
903 echo
904 echo '- Replace all occurrences -'
905 echo ${stringZ//[b2]/X}             # X-out b's and 2's
906 echo ${stringZ//abc/xyz}            # xyzABC123ABCxyz
907 echo ${arrayZ[@]//abc/xyz}          # Applied to each element.
908 echo ${sparseZ[@]//abc/xyz}         # Works as expected.
909
910 echo
911 echo '- Delete all occurrences -'
912 echo ${stringZ//[b2]/}
913 echo ${stringZ//abc/}
914 echo ${arrayZ[@]//abc/}
915 echo ${sparseZ[@]//abc/}
916
917 echo
918 echo '- - Prefix sub-element replacement - -'
919 echo '- - Match must include the first character. - -'
920 echo
921
922 echo '- Replace prefix occurrences -'
923 echo ${stringZ/#[b2]/X}             # Unchanged (neither is a prefix).
924 echo ${stringZ/#$(_abc)/XYZ}        # XYZABC123ABCabc
925 echo ${arrayZ[@]/#abc/XYZ}          # Applied to each element.
926 echo ${sparseZ[@]/#abc/XYZ}         # Works as expected.
927
928 echo
929 echo '- Delete prefix occurrences -'
930 echo ${stringZ/#[b2]/}
931 echo ${stringZ/#$(_abc)/}
932 echo ${arrayZ[@]/#abc/}
933 echo ${sparseZ[@]/#abc/}
934
935 echo
936 echo '- - Suffix sub-element replacement - -'
937 echo '- - Match must include the last character. - -'
938 echo
939
940 echo '- Replace suffix occurrences -'
941 echo ${stringZ/%[b2]/X}             # Unchanged (neither is a suffix).
942 echo ${stringZ/%$(_abc)/XYZ}        # abcABC123ABCXYZ
943 echo ${arrayZ[@]/%abc/XYZ}          # Applied to each element.
944 echo ${sparseZ[@]/%abc/XYZ}         # Works as expected.
945
946 echo
947 echo '- Delete suffix occurrences -'
948 echo ${stringZ/%[b2]/}
949 echo ${stringZ/%$(_abc)/}
950 echo ${arrayZ[@]/%abc/}
951 echo ${sparseZ[@]/%abc/}
952
953 echo
954 echo '- - Special cases of null Glob-Pattern - -'
955 echo
956
957 echo '- Prefix all -'
958 # null substring pattern means 'prefix'
959 echo ${stringZ/#/NEW}               # NEWabcABC123ABCabc
```

```
960 echo ${arrayZ[@]/#/NEW}             # Applied to each element.
961 echo ${sparseZ[@]/#/NEW}            # Applied to null-content also.
962                                      # That seems reasonable.
963
964 echo
965 echo '- Suffix all -'
966 # null substring pattern means 'suffix'
967 echo ${stringZ/%/NEW}               # abcABC123ABCabcNEW
968 echo ${arrayZ[@]/%/NEW}             # Applied to each element.
969 echo ${sparseZ[@]/%/NEW}            # Applied to null-content also.
970                                      # That seems reasonable.
971
972 echo
973 echo '- - Special case For-Each Glob-Pattern - -'
974 echo '- - - - This is a nice-to-have dream - - - -'
975 echo
976
977 _GenFunc() {
978     echo -n ${0}                     # Illustration only.
979     # Actually, that would be an arbitrary computation.
980 }
981
982 # All occurrences, matching the AnyThing pattern.
983 # Currently //*/ does not match null-content nor null-reference.
984 # /#/ and /%/ does match null-content but not null-reference.
985 echo ${sparseZ[@]//*/$(_GenFunc)}
986
987
988 #  A possible syntax would be to make
989 #+ the parameter notation used within this construct mean:
990 #   ${1} - The full element
991 #   ${2} - The prefix, if any, to the matched sub-element
992 #   ${3} - The matched sub-element
993 #   ${4} - The suffix, if any, to the matched sub-element
994 #
995 # echo ${sparseZ[@]//*/$(_GenFunc ${3})}   # Same as ${1} here.
996 # Perhaps it will be implemented in a future version of Bash.
997
998
999 exit 0
```

例子 A-33. 一个扩展的cd命令

```
  1 ###############################################################################
  2 #
  3 #       cdll
  4 #       by Phil Braham
  5 #
  6 #       ############################################
  7 #       Latest version of this script available from
  8 #       http://freshmeat.net/projects/cd/
  9 #       ############################################
 10 #
 11 #       .cd_new
 12 #
 13 #       An enhancement of the Unix cd command
 14 #
 15 #       There are unlimited stack entries and special entries. The stack
 16 #       entries keep the last cd_maxhistory
 17 #       directories that have been used. The special entries can be assigned
 18 #       to commonly used directories.
 19 #
 20 #       The special entries may be pre-assigned by setting the environment
 21 #       variables CDSn or by using the -u or -U command.
 22 #
 23 #       The following is a suggestion for the .profile file:
 24 #
 25 #               . cdll              #  Set up the cd command
 26 #       alias cd='cd_new'           #  Replace te cd command
 27 #               cd -U               #  Upload pre-assigned entries for
 28 #                                   #+ the stact and special entries
 29 #               cd -D               #  Set non-default mode
 30 #               alias @="cd_new @"  #  Allow @ to be used to get history
 31 #
 32 #       For help type:
 33 #
 34 #               cd -h or
 35 #               cd -H
 36 #
 37 #
 38 ###############################################################################
 39 #
 40 #       Version 1.2.1
 41 #
 42 #       Written by Phil Braham - Realtime Software Pty Ltd
 43 #       (realtime@mpx.com.au)
 44 #       Please send any suggestions or enhancements to the author (also at
 45 #       phil@braham.net)
 46 #
 47 ###############################################################################
 48
 49 cd_hm ()
 50 {
 51         ${PRINTF} "%s" "cd [dir] [0-9] [@[s|h] [-g [<dir>]] [-d] [-D] [-r<n>] [dir|0-9] [-R<n>]
[<dir>|0-9]
 52     [-s<n>] [-S<n>] [-u] [-U] [-f] [-F] [-h] [-H] [-v]
 53      <dir> Go to directory
 54      0-n         Goto previous directory (0 is previous, 1 is last but 1 etc)
```

```
                n is up to max history (default is 50)
    @           List history and special entries
    @h          List history entries
    @s          List special entries
    -g [<dir>]  Go to literal name (bypass special names)
                This is to allow access to dirs called '0','1','-h' etc
    -d          Change default action - verbose. (See note)
    -D          Change default action - silent. (See note)
    -s<n>       Go to the special entry <n>*
    -S<n>       Go to the special entry <n> and replace it with the current dir*
    -r<n> [<dir>] Go to directory <dir> and then put it on special entry <n>*
    -R<n> [<dir>] Go to directory <dir> and put current dir on special entry <n>*
    -a<n>       Alternative suggested directory. See note below.
    -f [<file>] File entries to <file>.
    -u [<file>] Update entries from <file>.
                If no filename supplied then default file (${CDPath}${2:-"$CDFile"}) is used
                -F and -U are silent versions
    -v          Print version number
    -h          Help
    -H          Detailed help

    *The special entries (0 - 9) are held until log off, replaced by another entry
     or updated with the -u command

    Alternative suggested directories:
    If a directory is not found then CD will suggest any possibilities. These are
    directories starting with the same letters and if any are found they are listed
    prefixed with -a<n> where <n> is a number.
    It's possible to go to the directory by entering cd -a<n> on the command line.

    The directory for -r<n> or -R<n> may be a number. For example:
        $ cd -r3 4  Go to history entry 4 and put it on special entry 3
        $ cd -R3 4  Put current dir on the special entry 3 and go to history entry 4
        $ cd -s3    Go to special entry 3

    Note that commands R,r,S and s may be used without a number and refer to 0:
        $ cd -s     Go to special entry 0
        $ cd -S     Go to special entry 0 and make special entry 0 current dir
        $ cd -r 1   Go to history entry 1 and put it on special entry 0
        $ cd -r     Go to history entry 0 and put it on special entry 0
    "
        if ${TEST} "$CD_MODE" = "PREV"
        then
                ${PRINTF} "$cd_mnset"
        else
                ${PRINTF} "$cd_mset"
        fi
}

cd_Hm ()
{
        cd_hm
        ${PRINTF} "%s" "
        The previous directories (0-$cd_maxhistory) are stored in the
        environment variables CD[0] - CD[$cd_maxhistory]
        Similarly the special directories S0 - $cd_maxspecial are in
        the environment variable CDS[0] - CDS[$cd_maxspecial]
        and may be accessed from the command line

        The default pathname for the -f and -u commands is $CDPath
        The default filename for the -f and -u commands is $CDFile

        Set the following environment variables:
            CDL_PROMPTLEN  - Set to the length of prompt you require.
                Prompt string is set to the right characters of the
                current directory.
                If not set then prompt is left unchanged
            CDL_PROMPT_PRE - Set to the string to prefix the prompt.
                Default is:
                    non-root:  \"\\[\\e[01;34m\\]\"  (sets colour to blue).
                    root:      \"\\[\\e[01;31m\\]\"  (sets colour to red).
            CDL_PROMPT_POST    - Set to the string to suffix the prompt.
                Default is:
                    non-root:  \"\\[\\e[00m\\]$\"   (resets colour and displays $).
                    root:      \"\\[\\e[00m\\]#\"   (resets colour and displays #).
            CDPath - Set the default path for the -f & -u options.
                     Default is home directory
            CDFile - Set the default filename for the -f & -u options.
                     Default is cdfile

"
    cd_version

}

cd_version ()
{
    printf "Version: ${VERSION_MAJOR}.${VERSION_MINOR} Date: ${VERSION_DATE}\n"
}

#
# Truncate right.
#
# params:
#   p1 - string
#   p2 - length to truncate to
#
# returns string in tcd
#
cd_right_trunc ()
{
    local tlen=${2}
```

```
157     local plen=${#1}
158     local str="${1}"
159     local diff
160     local filler="<--"
161     if ${TEST} ${plen} -le ${tlen}
162     then
163         tcd="${str}"
164     else
165         let diff=${plen}-${tlen}
166         elen=3
167         if ${TEST} ${diff} -le 2
168         then
169             let elen=${diff}
170         fi
171         tlen=-${tlen}
172         let tlen=${tlen}+${elen}
173         tcd=${filler:0:elen}${str:tlen}
174     fi
175 }
176 
177 #
178 # Three versions of do history:
179 #    cd_dohistory  - packs history and specials side by side
180 #    cd_dohistoryH - Shows only hstory
181 #    cd_dohistoryS - Shows only specials
182 #
183 cd_dohistory ()
184 {
185     cd_getrc
186         ${PRINTF} "History:\n"
187     local -i count=${cd_histcount}
188     while ${TEST} ${count} -ge 0
189     do
190         cd_right_trunc "${CD[count]}" ${cd_lchar}
191             ${PRINTF} "%2d %-${cd_lchar}.${cd_lchar}s " ${count} "${tcd}"
192 
193         cd_right_trunc "${CDS[count]}" ${cd_rchar}
194             ${PRINTF} "S%d %-${cd_rchar}.${cd_rchar}s\n" ${count} "${tcd}"
195         count=${count}-1
196     done
197 }
198 
199 cd_dohistoryH ()
200 {
201     cd_getrc
202         ${PRINTF} "History:\n"
203         local -i count=${cd_maxhistory}
204         while ${TEST} ${count} -ge 0
205         do
206                 ${PRINTF} "${count} %-${cd_flchar}.${cd_flchar}s\n" ${CD[$count]}
207                 count=${count}-1
208         done
209 }
210 
211 cd_dohistoryS ()
212 {
213     cd_getrc
214         ${PRINTF} "Specials:\n"
215         local -i count=${cd_maxspecial}
216         while ${TEST} ${count} -ge 0
217         do
218                 ${PRINTF} "S${count} %-${cd_flchar}.${cd_flchar}s\n" ${CDS[$count]}
219                 count=${count}-1
220         done
221 }
222 
223 cd_getrc ()
224 {
225     cd_flchar=$(stty -a | awk -F \; '/rows/ { print $2 $3 }' | awk -F \  '{ print $4 }')
226     if ${TEST} ${cd_flchar} -ne 0
227     then
228         cd_lchar=${cd_flchar}/2-5
229         cd_rchar=${cd_flchar}/2-5
230             cd_flchar=${cd_flchar}-5
231     else
232             cd_flchar=${FLCHAR:=75}  # cd_flchar is used for for the @s & @h history
233             cd_lchar=${LCHAR:=35}
234             cd_rchar=${RCHAR:=35}
235     fi
236 }
237 
238 cd_doselection ()
239 {
240         local -i nm=0
241         cd_doflag="TRUE"
242         if ${TEST} "${CD_MODE}" = "PREV"
243         then
244                 if ${TEST} -z "$cd_npwd"
245                 then
246                         cd_npwd=0
247                 fi
248         fi
249         tm=$(echo "${cd_npwd}" | cut -b 1)
250     if ${TEST} "${tm}" = "-"
251     then
252         pm=$(echo "${cd_npwd}" | cut -b 2)
253         nm=$(echo "${cd_npwd}" | cut -d $pm -f2)
254         case "${pm}" in
255                 a) cd_npwd=${cd_sugg[$nm]} ;;
256                 s) cd_npwd="${CDS[$nm]}" ;;
257                 S) cd_npwd="${CDS[$nm]}" ; CDS[$nm]=`pwd` ;;
258                 r) cd_npwd="$2" ; cd_specDir=$nm ; cd_doselection "$1" "$2";;
```

```
259                 R) cd_npwd="$2" ; CDS[$nm]=`pwd` ; cd_doselection "$1" "$2";;
260         esac
261     fi
262
263         if ${TEST} "${cd_npwd}" != "." -a "${cd_npwd}" != ".." -a "${cd_npwd}" -le
${cd_maxhistory} >>/dev/null 2>&1
264         then
265                 cd_npwd=${CD[$cd_npwd]}
266         else
267                 case "$cd_npwd" in
268                         @)  cd_dohistory ; cd_doflag="FALSE" ;;
269                        @h) cd_dohistoryH ; cd_doflag="FALSE" ;;
270                        @s) cd_dohistoryS ; cd_doflag="FALSE" ;;
271                        -h) cd_hm ; cd_doflag="FALSE" ;;
272                        -H) cd_Hm ; cd_doflag="FALSE" ;;
273                        -f) cd_fsave "SHOW" $2 ; cd_doflag="FALSE" ;;
274                        -u) cd_upload "SHOW" $2 ; cd_doflag="FALSE" ;;
275                        -F) cd_fsave "NOSHOW" $2 ; cd_doflag="FALSE" ;;
276                        -U) cd_upload "NOSHOW" $2 ; cd_doflag="FALSE" ;;
277                        -g) cd_npwd="$2" ;;
278                        -d) cd_chdefm 1; cd_doflag="FALSE" ;;
279                        -D) cd_chdefm 0; cd_doflag="FALSE" ;;
280                        -r) cd_npwd="$2" ; cd_specDir=0 ; cd_doselection "$1" "$2";;
281                        -R) cd_npwd="$2" ; CDS[0]=`pwd` ; cd_doselection "$1" "$2";;
282                        -s) cd_npwd="${CDS[0]}" ;;
283                        -S) cd_npwd="${CDS[0]}" ; CDS[0]=`pwd` ;;
284                        -v) cd_version ; cd_doflag="FALSE";;
285                 esac
286         fi
287 }
288
289 cd_chdefm ()
290 {
291         if ${TEST} "${CD_MODE}" = "PREV"
292         then
293                 CD_MODE=""
294                 if ${TEST} $1 -eq 1
295                 then
296                         ${PRINTF} "${cd_mset}"
297                 fi
298         else
299                 CD_MODE="PREV"
300                 if ${TEST} $1 -eq 1
301                 then
302                         ${PRINTF} "${cd_mnset}"
303                 fi
304         fi
305 }
306
307 cd_fsave ()
308 {
309         local sfile=${CDPath}${2:-"$CDFile"}
310         if ${TEST} "$1" = "SHOW"
311         then
312                 ${PRINTF} "Saved to %s\n" $sfile
313         fi
314         ${RM} -f ${sfile}
315         local -i count=0
316         while ${TEST} ${count} -le ${cd_maxhistory}
317         do
318                 echo "CD[$count]=\"${CD[$count]}\"" >> ${sfile}
319                 count=${count}+1
320         done
321         count=0
322         while ${TEST} ${count} -le ${cd_maxspecial}
323         do
324                 echo "CDS[$count]=\"${CDS[$count]}\"" >> ${sfile}
325                 count=${count}+1
326         done
327 }
328
329 cd_upload ()
330 {
331         local sfile=${CDPath}${2:-"$CDFile"}
332         if ${TEST} "${1}" = "SHOW"
333         then
334                 ${PRINTF} "Loading from %s\n" ${sfile}
335         fi
336         . ${sfile}
337 }
338
339 cd_new ()
340 {
341     local -i count
342     local -i choose=0
343
344         cd_npwd="${1}"
345         cd_specDir=-1
346         cd_doselection "${1}" "${2}"
347
348         if ${TEST} ${cd_doflag} = "TRUE"
349         then
350                 if ${TEST} "${CD[0]}" != "`pwd`"
351                 then
352                         count=$cd_maxhistory
353                         while ${TEST} $count -gt 0
354                         do
355                                 CD[$count]=${CD[$count-1]}
356                                 count=${count}-1
357                         done
358                         CD[0]=`pwd`
359                 fi
```

```
                command cd "${cd_npwd}" 2>/dev/null
        if ${TEST} $? -eq 1
        then
            ${PRINTF} "Unknown dir: %s\n" "${cd_npwd}"
            local -i ftflag=0
            for i in "${cd_npwd}"*
            do
                if ${TEST} -d "${i}"
                then
                    if ${TEST} ${ftflag} -eq 0
                    then
                        ${PRINTF} "Suggest:\n"
                        ftflag=1
                fi
                    ${PRINTF} "\t-a${choose} %s\n" "$i"
                                        cd_sugg[$choose]="${i}"
                    choose=${choose}+1
        fi
            done
        fi
        fi

        if ${TEST} ${cd_specDir} -ne -1
        then
                CDS[${cd_specDir}]=`pwd`
        fi

        if ${TEST} ! -z "${CDL_PROMPTLEN}"
        then
        cd_right_trunc "${PWD}" ${CDL_PROMPTLEN}
            cd_rp=${CDL_PROMPT_PRE}${tcd}${CDL_PROMPT_POST}
                export PS1="$(echo -ne ${cd_rp})"
        fi
}
###############################################################################
#                                                                             #
#                           Initialisation here                               #
#                                                                             #
###############################################################################
#
VERSION_MAJOR="1"
VERSION_MINOR="2.1"
VERSION_DATE="24-MAY-2003"
#
alias cd=cd_new
#
# Set up commands
RM=/bin/rm
TEST=test
PRINTF=printf              # Use builtin printf

###############################################################################
#                                                                             #
# Change this to modify the default pre- and post prompt strings.             #
# These only come into effect if CDL_PROMPTLEN is set.                        #
#                                                                             #
###############################################################################
if ${TEST} ${EUID} -eq 0
then
#   CDL_PROMPT_PRE=${CDL_PROMPT_PRE:="$HOSTNAME@"}
    CDL_PROMPT_PRE=${CDL_PROMPT_PRE:="\\[\\e[01;31m\\]"}    # Root is in red
    CDL_PROMPT_POST=${CDL_PROMPT_POST:="\\[\\e[00m\\]#"}
else
    CDL_PROMPT_PRE=${CDL_PROMPT_PRE:="\\[\\e[01;34m\\]"}    # Users in blue
    CDL_PROMPT_POST=${CDL_PROMPT_POST:="\\[\\e[00m\\]$"}
fi
###############################################################################
#
# cd_maxhistory defines the max number of history entries allowed.
typeset -i cd_maxhistory=50

###############################################################################
#
# cd_maxspecial defines the number of special entries.
typeset -i cd_maxspecial=9
#
#
###############################################################################
#
# cd_histcount defines the number of entries displayed in the history command.
typeset -i cd_histcount=9
#
###############################################################################
export CDPath=${HOME}/
#  Change these to use a different                                            #
#+ default path and filename                                                 #
export CDFile=${CDFILE:=cdfile}                  # for the -u and -f commands  #
#
###############################################################################
                                                                              #
typeset -i cd_lchar cd_rchar cd_flchar
                                  #  This is the number of chars to allow for the  #
cd_flchar=${FLCHAR:=75}          #+ cd_flchar is used for for the @s & @h history #

typeset -ax CD CDS
#
cd_mset="\n\tDefault mode is now set - entering cd with no parameters has the default
action\n\tUse cd -d or -D for cd to go to previous directory with no parameters\n"
cd_mnset="\n\tNon-default mode is now set - entering cd with no parameters is the same as entering
cd 0\n\tUse cd -d or -D to change default cd action\n"

# ================================================================= #
```

```
460
461
462
463 : <<DOCUMENTATION
464
465 Written by Phil Braham. Realtime Software Pty Ltd.
466 Released under GNU license. Free to use. Please pass any modifications
467 or comments to the author Phil Braham:
468
469 realtime@mpx.com.au
470 ================================================================================
471
472 cdll is a replacement for cd and incorporates similar functionality to
473 the bash pushd and popd commands but is independent of them.
474
475 This version of cdll has been tested on Linux using Bash. It will work
476 on most Linux versions but will probably not work on other shells without
477 modification.
478
479 Introduction
480 ============
481
482 cdll allows easy moving about between directories. When changing to a new
483 directory the current one is automatically put onto a stack. By default
484 50 entries are kept, but this is configurable. Special directories can be
485 kept for easy access - by default up to 10, but this is configurable. The
486 most recent stack entries and the special entries can be easily viewed.
487
488 The directory stack and special entries can be saved to, and loaded from,
489 a file. This allows them to be set up on login, saved before logging out
490 or changed when moving project to project.
491
492 In addition, cdll provides a flexible command prompt facility that allows,
493 for example, a directory name in colour that is truncated from the left
494 if it gets too long.
495
496
497 Setting up cdll
498 ===============
499
500 Copy cdll to either your local home directory or a central directory
501 such as /usr/bin (this will require root access).
502
503 Copy the file cdfile to your home directory. It will require read and
504 write access. This a default file that contains a directory stack and
505 special entries.
506
507 To replace the cd command you must add commands to your login script.
508 The login script is one or more of:
509
510     /etc/profile
511     ~/.bash_profile
512     ~/.bash_login
513     ~/.profile
514     ~/.bashrc
515     /etc/bash.bashrc.local
516
517 To setup your login, ~/.bashrc is recommended, for global (and root) setup
518 add the commands to /etc/bash.bashrc.local
519
520 To set up on login, add the command:
521     . <dir>/cdll
522 For example if cdll is in your local home directory:
523     . ~/cdll
524 If in /usr/bin then:
525     . /usr/bin/cdll
526
527 If you want to use this instead of the buitin cd command then add:
528     alias cd='cd_new'
529 We would also recommend the following commands:
530     alias @='cd_new @'
531     cd -U
532     cd -D
533
534 If you want to use cdll's prompt facilty then add the following:
535     CDL_PROMPTLEN=nn
536 Where nn is a number described below. Initially 99 would be suitable
537 number.
538
539 Thus the script looks something like this:
540
541     ###############################################################################
542     # CD Setup
543     ###############################################################################
544     CDL_PROMPTLEN=21        # Allow a prompt length of up to 21 characters
545     . /usr/bin/cdll         # Initialise cdll
546     alias cd='cd_new'       # Replace the built in cd command
547     alias @='cd_new @'      # Allow @ at the prompt to display history
548     cd -U                   # Upload directories
549     cd -D                   # Set default action to non-posix
550     ###############################################################################
551
552 The full meaning of these commands will become clear later.
553
554 There are a couple of caveats. If another program changes the directory
555 without calling cdll, then the directory won't be put on the stack and
556 also if the prompt facility is used then this will not be updated. Two
557 programs that can do this are pushd and popd. To update the prompt and
558 stack simply enter:
559
560     cd .
561
```

```
Note that if the previous entry on the stack is the current directory
then the stack is not updated.

Usage
=====
cd [dir] [0-9] [@[s|h] [-g <dir>] [-d] [-D] [-r<n>] [dir|0-9] [-R<n>]
   [<dir>|0-9] [-s<n>] [-S<n>] [-u] [-U] [-f] [-F] [-h] [-H] [-v]

    <dir>       Go to directory
    0-n         Goto previous directory (0 is previous, 1 is last but 1, etc.)
                n is up to max history (default is 50)
    @           List history and special entries (Usually available as $ @)
    @h          List history entries
    @s          List special entries
    -g [<dir>]  Go to literal name (bypass special names)
                This is to allow access to dirs called '0','1','-h' etc
    -d          Change default action - verbose. (See note)
    -D          Change default action - silent. (See note)
    -s<n>       Go to the special entry <n>
    -S<n>       Go to the special entry <n> and replace it with the current dir
    -r<n> [<dir>] Go to directory <dir> and then put it on special entry <n>
    -R<n> [<dir>] Go to directory <dir> and put current dir on special entry <n>
    -a<n>       Alternative suggested directory. See note below.
    -f [<file>] File entries to <file>.
    -u [<file>] Update entries from <file>.
                If no filename supplied then default file (~/cdfile) is used
                -F and -U are silent versions
    -v          Print version number
    -h          Help
    -H          Detailed help



Examples
========

These examples assume non-default mode is set (that is, cd with no
parameters will go to the most recent stack directory), that aliases
have been set up for cd and @ as described above and that cd's prompt
facility is active and the prompt length is 21 characters.

    /home/phil$ @                                                    # List the entries with the @
    History:                                                         # Output of the @ command
    .....                                                            # Skipped these entries for
brevity
    1 /home/phil/ummdev               S1 /home/phil/perl             # Most recent two history
entries
    0 /home/phil/perl/eg              S0 /home/phil/umm/ummdev       # and two special entries are
shown

    /home/phil$ cd /home/phil/utils/Cdll                             # Now change directories
    /home/phil/utils/Cdll$ @                                         # Prompt reflects the
directory.
    History:                                                         # New history
    .....
    1 /home/phil/perl/eg              S1 /home/phil/perl             # History entry 0 has moved to
1
    0 /home/phil                       S0 /home/phil/umm/ummdev      # and the most recent has
entered

To go to a history entry:

    /home/phil/utils/Cdll$ cd 1                                      # Go to history entry 1.
    /home/phil/perl/eg$                                              # Current directory is now
what was 1

To go to a special entry:

    /home/phil/perl/eg$ cd -s1                                       # Go to special entry 1
    /home/phil/umm/ummdev$                                           # Current directory is S1

To go to a directory called, for example, 1:

    /home/phil$ cd -g 1                                              # -g ignores the special
meaning of 1
    /home/phil/1$

To put current directory on the special list as S1:
    cd -r1 .        #  OR
    cd -R1 .        #  These have the same effect if the directory is
                    #+ . (the current directory)

To go to a directory and add it as a special
    The directory for -r<n> or -R<n> may be a number. For example:
        $ cd -r3 4  Go to history entry 4 and put it on special entry 3
        $ cd -R3 4  Put current dir on the special entry 3 and go to
                    history entry 4
        $ cd -s3    Go to special entry 3

    Note that commands R,r,S and s may be used without a number and
    refer to 0:
        $ cd -s     Go to special entry 0
        $ cd -S     Go to special entry 0 and make special entry 0
                    current dir
        $ cd -r 1   Go to history entry 1 and put it on special entry 0
        $ cd -r     Go to history entry 0 and put it on special entry 0


    Alternative suggested directories:

    If a directory is not found, then CD will suggest any
    possibilities. These are directories starting with the same letters
```

```
656     and if any are found they are listed prefixed with -a<n>
657     where <n> is a number. It's possible to go to the directory
658     by entering cd -a<n> on the command line.
659
660         Use cd -d or -D to change default cd action. cd -H will show
661         current action.
662
663         The history entries (0-n) are stored in the environment variables
664         CD[0] - CD[n]
665         Similarly the special directories S0 - 9 are in the environment
666         variable CDS[0] - CDS[9]
667         and may be accessed from the command line, for example:
668
669             ls -l ${CDS[3]}
670             cat ${CD[8]}/file.txt
671
672         The default pathname for the -f and -u commands is ~
673         The default filename for the -f and -u commands is cdfile
674
675
676 Configuration
677 =============
678
679     The following environment variables can be set:
680
681         CDL_PROMPTLEN  - Set to the length of prompt you require.
682             Prompt string is set to the right characters of the current
683             directory. If not set, then prompt is left unchanged. Note
684             that this is the number of characters that the directory is
685             shortened to, not the total characters in the prompt.
686
687             CDL_PROMPT_PRE - Set to the string to prefix the prompt.
688                 Default is:
689                     non-root:  "\\[\\e[01;34m\\]"  (sets colour to blue).
690                     root:      "\\[\\e[01;31m\\]"  (sets colour to red).
691
692             CDL_PROMPT_POST    - Set to the string to suffix the prompt.
693                 Default is:
694                     non-root:  "\\[\\e[00m\\]$"    (resets colour and displays $).
695                     root:      "\\[\\e[00m\\]#"    (resets colour and displays #).
696
697         Note:
698             CDL_PROMPT_PRE & _POST only t
699
700         CDPath - Set the default path for the -f & -u options.
701                  Default is home directory
702         CDFile - Set the default filename for the -f & -u options.
703                  Default is cdfile
704
705
706     There are three variables defined in the file cdll which control the
707     number of entries stored or displayed. They are in the section labeled
708     'Initialisation here' towards the end of the file.
709
710         cd_maxhistory       - The number of history entries stored.
711                               Default is 50.
712         cd_maxspecial       - The number of special entries allowed.
713                               Default is 9.
714         cd_histcount        - The number of history and special entries
715                               displayed. Default is 9.
716
717     Note that cd_maxspecial should be >= cd_histcount to avoid displaying
718     special entries that can't be set.
719
720
721 Version: 1.2.1 Date: 24-MAY-2003
722
723 DOCUMENTATION
```

Appendix B. 参考卡片

下面的参考卡片提供了对于某些特定的脚本概念的一个*总结*. 之前我们已经对这里所提及的概念进行了详细的解释, 并且给出了使用的例子.

表格 B-1. 特殊的shell变量

变量	含义
$0	脚本名字
$1	位置参数 #1
$2 - $9	位置参数 #2 - #9
${10}	位置参数 #10
$#	位置参数的个数
"$*"	所有的位置参数(作为单个字符串) *
"$@"	所有的位置参数(每个都作为独立的字符串)
${#*}	传递到脚本中的命令行参数的个数
${#@}	传递到脚本中的命令行参数的个数
$?	返回值
$$	脚本的进程ID(PID)
$-	传递到脚本中的标志(使用*set*)
$_	之前命令的最后一个参数
$!	运行在后台的最后一个作业的进程ID(PID)

---

* *必须被引用起来*, 否则默认为"$@".

表格 B-2. 测试操作: 二元比较

操作	描述	-----	操作	描述
算术比较			字符串比较	
-eq	等于		=	等于
			==	等于
-ne	不等于		!=	不等于
-lt	小于		\<	小于 (ASCII) *
-le	小于等于			
-gt	大于		\>	大于 (ASCII) *
-ge	大于等于			
			-z	字符串为空
			-n	字符串不为空
算术比较	双括号((...))结构			
>				

	大于			
>=	大于等于			
<	小于			
<=	小于等于			

---

* *如果在双中括号 [[...]] 测试结构中使用的话, 那么就不需要使用转义符\ 了.*

表格 B-3. 文件类型的测试操作

操作	测试条件	____ _	操作	测试条件
-e	文件是否存在		-s	文件大小不为0
-f	是一个*标准*文件			
-d	是一个*目录*		-r	文件具有*读*权限
-h	文件是一个*符号链接*		-w	文件具有*写*权限
-L	文件是一个*符号链接*		-x	文件具有*执行*权限
-b	文件是一个*块设备*			
-c	文件是一个*字符设备*		-g	设置了*sgid*标记
-p	文件是一个*管道*		-u	设置了*suid*标记
-S	文件是一个socket		-k	设置了"粘贴位"
-t	文件与一个*终端*相关联			
-N	从这个文件最后一次被读取之后, 它被修改过		F1 -nt F2	文件F1比文件F2*新* *
-O	这个文件的宿主是你		F1 -ot F2	文件F1比文件F2*旧* *
-G	文件的*组id*与你所属的组相同		F1 -ef F2	文件F1和文件F2都是同一个文件的*硬链接* *
!	"非" (反转上边的测试结果)			

---

* 二元操作符(需要两个操作数).

表格 B-4. 参数替换和扩展

表达式	含义
${var}	变量var的值, 与$var相同
${var-DEFAULT}	如果var没有被声明, 那么就以$DEFAULT作为其值 *
${var:-DEFAULT}	如果var没有被声明, 或者其值为空, 那么就以$DEFAULT作为其值 *
${var=DEFAULT}	如果var没有被声明, 那么就以$DEFAULT作为其值 *
${var:=DEFAULT}	如果var没有被声明, 或者其值为空, 那么就以$DEFAULT作为其值 *
${var+OTHER}	如果var声明了, 那么其值就是$OTHER, 否则就为null字符串
${var:+OTHER}	如果var被设置了, 那么其值就是$OTHER, 否则就为null字符串

${var?ERR_MSG}	如果var没被声明, 那么就打印$ERR_MSG *
${var:?ERR_MSG}	如果var没被设置, 那么就打印$ERR_MSG *
${!varprefix*}	匹配之前所有以varprefix开头进行声明的变量
${!varprefix@}	匹配之前所有以varprefix开头进行声明的变量

* 当然, 如果变量var已经被设置的话, 那么其值就是$var.

表格 B-5. 字符串操作

表达式	含义
${#string}	$string的长度
${string:position}	在$string中, 从位置$position开始提取子串
${string:position:length}	在$string中, 从位置$position开始提取长度为$length的子串
${string#substring}	从变量$string的开头, 删除最短匹配$substring的子串
${string##substring}	从变量$string的开头, 删除最长匹配$substring的子串
${string%substring}	从变量$string的结尾, 删除最短匹配$substring的子串
${string%%substring}	从变量$string的结尾, 删除最长匹配$substring的子串
${string/substring/replacement}	使用$replacement, 来代替第一个匹配的$substring
${string//substring/replacement}	使用$replacement, 代替*所有*匹配的$substring
${string/#substring/replacement}	如果$string的*前缀*匹配$substring, 那么就用$replacement来代替匹配到的$substring
${string/%substring/replacement}	如果$string的*后缀*匹配$substring, 那么就用$replacement来代替匹配到的$substring
expr match "$string" '$substring'	匹配$string开头的$substring*的长度
expr "$string" : '$substring'	匹配$string开头的$substring*的长度
expr index "$string" $substring	在$string中匹配到的$substring的第一个字符出现的位置
expr substr $string $position $length	在$string中从位置$position开始提取长度为$length的子串
expr match "$string" '\($substring\)'	从$string的开头位置提取$substring*
expr "$string" : '\($substring\)'	从$string的开头位置提取$substring*
expr match "$string" '.*\($substring\)'	从$string的结尾提取$substring*
expr "$string" : '.*\($substring\)'	从$string的结尾提取$substring*

* $substring是一个*正则表达式*.

表格 B-6. 一些结构的汇总

表达式	解释
中括号	
`if [ CONDITION ]`	测试结构
`if [[ CONDITION ]]`	扩展的测试结构
`Array[1]=element1`	数组初始化
`[a-z]`	正则表达式的字符范围
大括号	
`${variable}`	参数替换
`${!variable}`	间接变量引用
`{ command1; command2; . . . commandN; }`	代码块
`{string1,string2,string3,...}`	大括号扩展
圆括号	
`( command1; command2 )`	子shell中执行的命令组
`Array=(element1 element2 element3)`	数组初始化
`result=$(COMMAND)`	在子shell中执行命令，并将结果赋值给变量
`>(COMMAND)`	进程替换
`<(COMMAND)`	进程替换
双圆括号	
`(( var = 78 ))`	整型运算
`var=$(( 20 + 5 ))`	整型运算，并将结果赋值给变量
引号	
`"$variable"`	"弱"引用
`'string'`	"强"引用
后置引用	
`result=`COMMAND``	在子shell中运行命令，并将结果赋值给变量

Appendix C. 一个学习Sed和Awk的小手册

目录
C.1. Sed
C.2. Awk

这是一份关于sed和awk文本处理工具的概要介绍. 我们在这里只讨论一些基本命令, 但是这些基本命令已经足够让我们了解如何在shell脚本中使用简单的sed和awk结构.

sed: 一个非交互的文本文件编辑器

awk: 一个面向域的模式处理语言, 使用类似C的语法

在我们讨论这两个工具的差异性之前, 我们先说一下它们的共性, 这两个工具都使用类似的调用语法, 都使用正则表达式, 默认情况下都从stdin中读取输入, 并且都输出到stdout. 它们都是行为良好的UNIX工具, 并且它们能够很好的在一起工作. 其中一个的输出可以通过管道传递给另一个, 正是由于它们组合能力, 才使得shell脚本能够具备一些Perl的特性.

注意一下这两个工具之间的一个非常重要的区别, shell脚本可以很容易的给sed传递参数, 但是传递参数给awk就比较复杂(请参考例子 33-5和例子 9-24).

C.1. Sed

Sed是非交互式的行编辑器. 它即可以从stdin中接收文本输入, 也可以从文件中接收文本输入, 它对输入中的指定行进行特定的操作, 一行操作一次, 然后将结果输出到stdout, 或输出到文件中. 在shell脚本中使用的话, sed通常都是作为管道工具链中的一个处理部分来使用.

Sed会决定它需要处理那些行, 因为sed的参数就包含有*地址范围*. [1] 既可以通过行号来指定地址范围, 也可以通过模式匹配来决定地址范围. 比如, *3d*表示sed会删除输入的第3行, */windows/d*表示sed会删除掉所有匹配"windows"的输入行.

对于sed工具包的所有操作来说, 我们最关心的其实就是3个最主要的操作. 分别是printing(打印到stdout), deletion(删除), 和substitution(替换).

表格 C-1. 基本sed操作

操作符	名字	效果
[地址范围]/p	打印	打印[指定的地址范围]
[地址范围]/d	删除	删除[指定的地址范围]
s/pattern1/pattern2/	替换	将指定行中, 将第一个匹配到的pattern1, 替换为pattern2.
[地址范围]/s/pattern1/pattern2/	替换	在地址范围指定的每一行中, 将第一个匹配到的pattern1, 替换为pattern2.
[地址范围]/y/pattern1/pattern2/	transform	在地址范围指定的每一行中, 将pattern1中的每个匹配到pattern2的字符都使用pattern2的相应字符作替换. (等价于tr命令)
g	全局	在每个匹配的输入行中, 将每个模式匹配都作相应的操作. (译者注: 不只局限于第一个匹配)

除非在*替换*命令的后边明确指定选项g(*全局*), 否则的话, 替换操作只会替换掉每行上的第一个模式匹配实例.

如果在命令行或脚本中使用这个命令, sed操作可能还需要某些选项和引用.

```
sed -e '/^$/d' $filename
# -e选项, 将会使得后边的字符被看作为编辑指令.
#  (如果只给"sed"传递了单个指令, 那么"-e"是可选的.)
#  "强"引用('')将会保护指令中的RE(正则表达式)字符串,
#+ 也就是防止脚本将RE重新解释为特殊字符.
# (这会为sed命令, 保存指令的RE表达式.)
#
# 将会对文件$filename中的文本进行操作.
```

在某些特定的情况下, sed编辑命令将不会和单引号的强引用一起工作.

```
filename=file1.txt
pattern=BEGIN

  sed "/^$pattern/d" "$filename"  # 工作正常.
# sed '/^$pattern/d' "$filename"    就会出现异常的结果.
#        在这个实例中, 被强引用(' ... ')引起的
#+      "$pattern"就不会扩展为"BEGIN".
```

Sed命令的-e选项表示后续的字符串是一个指令, 或指令集. 如果后续的字符串中只有一个指令, 那么-e选项可以被省略.

```
1 sed -n '/xzy/p' $filename
2 # -n选项会让sed只打印那些匹配模式的行.
3 # 否则所有的输入行都会被打印.
4 # 这里可以省略-e选项, 因为这里只有一个编辑指令.
```

表格 C-2. sed操作符举例

表示法	效果
8d	删除输入的第8行.
/^$/d	删除所有空行.
1,/^$/d	从输入的开头一直删除到第1个空行(第一个空行也删除掉).
/Jones/p	只打印那些包含"Jones"的行(使用-n选项).
s/Windows/Linux/	在每个输入行中, 将第一个出现的"Windows"实例替换为"Linux".
s/BSOD/stability/g	在每个输入行中, 将所有"BSOD"都替换为"stability".
s/ *$//	删除掉每行结尾的所有空格.
s/00*/0/g	将所有连续出现的0都压缩成单个的0.
/GUI/d	删除掉所有包含"GUI"的行.
s/GUI//g	将所有"GUI"都删除掉, 并保持剩余部分的完整性.

在输入行中, 将一个字符串替换为空字符, 等价于删除这个字符串. 剩余部分会保持完整. 比如s/GUI//, 拿下边这句为例:

```
The most important parts of any application are its GUI and sound effects
```

结果为:

```
The most important parts of any application are its  and sound effects
```

反斜线将会强制sed替换命令延续到下一行. 类似于, 在第一行的结尾使用*换行*作为*替换字符串*.

```
1 s/^  */\
2 /g
```

这将每行开头的空格用换行来替换. 最后的结果就是将每段的缩进替换为一个空行.

地址范围后边可以加上一系列操作, 这些操作可能需要放到大括号对中, 并且需要重起一行.

```
1 /[0-9A-Za-z]/,/^$/{
2 /^$/d
3 }
```

这只会删除连续空行中的第一行. 对于单行间距的文本文件来说, 这很有用, 但是会保留段落间的空行.

将文本文件双倍行距的快速方法是`sed G filename`.

下面是一些在脚本中使用sed命令的例子:

1. 例子 33-1
2. 例子 33-2
3. 例子 12-3
4. 例子 A-2
5. 例子 12-15
6. 例子 12-24
7. 例子 A-12
8. 例子 A-17

9. 例子 12-29
10. 例子 10-9
11. 例子 12-43
12. 例子 A-1
13. 例子 12-13
14. 例子 12-11
15. 例子 A-10
16. 例子 17-12
17. 例子 12-16
18. 例子 A-29

如果想了解sed命令的更多细节，请察看*参考文献*中的这方面的参考资料.

注意事项

[1] 如果没指定地址范围，那么默认就是*所有*行.

C.2. Awk

Awk是功能完整的文本处理语言, 使用类似于C的语法. 它具有一整套操作符和能力集, 我们只在这里讲解一小部分 - 也就是在shell脚本中最有用的部分.

Awk将传递进来的每行输入都分割成*域*. 默认情况下, 一个域指的就是使用空白分隔的一个连续字符串, 不过我们可以修改属性来改变分隔符. Awk将会分析并操作每个分割域. 因为这种特性, 所以awk非常善于处理结构化的文本文件 -- 尤其是表 -- 将数据组织成统一的块, 比如说分成行和列.

强引用(单引号)和大括号用来包含shell脚本中的awk代码段.

```
echo one two | awk '{print $1}'
# one

echo one two | awk '{print $2}'
# two


awk '{print $3}' $filename
# 打印文件$filename的域#3, 到stdout.

awk '{print $1 $5 $6}' $filename
# 打印文件$filename的域#1, #5, 和#6.
```

事实上, 上边我们只讲解了awk的print命令. 我们需要在这里讲解awk的另一个特点, 变量. Awk处理变量的手段与shell脚本很相似, 虽然更复杂一些.

```
{ total += ${column_number} }
```

上边这句将*column_number*的值加上"total"的值然后再赋给*total*. 最后, 为了打印出"total", 我们需要一个END命令块, 当脚本处理完所有输入之后, 就会执行这个命令块中的内容.

```
END { print total }
```

与END对应, 还有BEGIN命令块, 在脚本处理所有输入之前, 将会执行这个命令块中的内容.

下面这个例子展示了awk如何在shell脚本中添加文本分析工具.

例子 C-1. 计算字符出现次数

```
#! /bin/sh
# letter-count2.sh: 在文本文件中计算字符的出现次数.
#
# 由nyal [nyal@voila.fr]编写.
# 授权使用.
# 本文作者重新注释.
# 版本 1.1: 经过修改可用于gawk 3.1.3.
#               (也可用于awk的早期版本.)


INIT_TAB_AWK=""
# 初始化awk脚本的参数.
count_case=0
FILE_PARSE=$1

E_PARAMERR=65

usage()
{
    echo "Usage: letter-count.sh file letters" 2>&1
    # 比如:   ./letter-count2.sh filename.txt a b c
    exit $E_PARAMERR  # 传递到脚本的参数个数不够.
}

if [ ! -f "$1" ] ; then
    echo "$1: No such file." 2>&1
    usage                   # 打印使用信息并退出.
```

```
 28 fi
 29
 30 if [ -z "$2" ] ; then
 31     echo "$2: No letters specified." 2>&1
 32     usage
 33 fi
 34
 35 shift                          # 指定的字符.
 36 for letter in `echo $@`     # for循环遍历 . . .
 37   do
 38   INIT_TAB_AWK="$INIT_TAB_AWK tab_search[${count_case}] = \"$letter\";
final_tab[${count_case}] = 0; "
 39   # 作为参数传递到下边的awk脚本中.
 40   count_case=`expr $count_case + 1`
 41 done
 42
 43 # 调试:
 44 # echo $INIT_TAB_AWK;
 45
 46 cat $FILE_PARSE |
 47 # 将目标文件通过管道传递下边的awk脚本中.
 48
 49 # ------------------------------------------------------------------------------
----
 50 # 下边是本脚本的早期版本使用的方法:
 51 # awk -v tab_search=0 -v final_tab=0 -v tab=0 -v nb_letter=0 -v chara=0 -v
chara2=0 \
 52
 53 awk \
 54 "BEGIN { $INIT_TAB_AWK } \
 55 { split(\$0, tab, \"\"); \
 56 for (chara in tab) \
 57 { for (chara2 in tab_search) \
 58 { if (tab_search[chara2] == tab[chara]) { final_tab[chara2]++ } } } } \
 59 END { for (chara in final_tab) \
 60 { print tab_search[chara] \" => \" final_tab[chara] } }"
 61 # ------------------------------------------------------------------------------
----
 62 #  不是所有的都那么复杂, 只是 . . .
 63 #+ for循环, if条件判断, 和几个指定函数而已.
 64
 65 exit $?
 66
 67 # 与脚本letter-count.sh相比较.
```

如果想再看一些在shell脚本中使用awk的简单例子, 如下:

1. 例子 11-12
2. 例子 16-8
3. 例子 12-29
4. 例子 33-5
5. 例子 9-24
6. 例子 11-19
7. 例子 27-2
8. 例子 27-3
9. 例子 10-3
10. 例子 12-55
11. 例子 9-29
12. 例子 12-4
13. 例子 9-14
14. 例子 33-16

15. 例子 10-8

16. 例子 33-4

我们在这里所要讲解的awk内容就这么多，但是事实上还有好多东西需要学. 可以参考参考文献中的内容深入学习.

Appendix D. 带有特殊含义的退出码

表格 D-1. "保留的"退出码

退出码的值	含义	例子	注释
1	通用错误	let "var1 = 1/0"	各种各样的错误都可能使用这个退出码, 比如"除0错误"
2	shell内建命令使用错误(Bash文档上有说明)		很少看到, 通常情况下退出码都为1
126	命令调用不能执行		程序或命令的权限是不可执行的
127	"command not found"		估计是$PATH不对, 或者是拼写错误
128	exit的参数错误	exit 3.14159	exit只能以整数作为参数, 范围是0 - 255(见脚注)
128+n	信号"n"的致命错误	kill -9 脚本的$PPID	$? 返回137(128 + 9)
130	用Control-C来结束脚本		Control-C是信号2的致命错误, (130 = 128 + 2, 见上边)
255*	超出范围的退出状态	exit -1	exit命令只能够接受范围是0 - 255的整数作为参数

通过上面的表, 我们了解到, 退出码1 - 2, 126 - 165, 和255 [1] 都具有特殊的含义, 因此应该避免使用用户指定的退出参数. 如果脚本使用exit 127作为退出语句, 那么可能就会在故障诊断的时候产生混淆(如何判断这是由"command not found"引起的, 还是由用户定义引起的?). 然而, 许多脚本使用exit 1作为通用的返回错误值. 因为退出码1能够表示的错误太多了, 不过这么做, 对于调试来说, 也起不到任何帮助的作用.

其实早就有人对退出状态值进行了系统的分类(请参考/usr/include/sysexits.h), 不过这个文件是为C/C++程序员准备的. 其实shell脚本也需要这样一个类似的标准. 所以本文作者呼吁限制使用用户定义的退出码, 尤其是范围64 - 113(还有0, 表示成功), 这么做, 就可以和C/C++标准保持一致. 这样我们就有了50个可用的退出码, 而且非常便于故障诊断.

本书中所有例子中的用户定义退出码都符合这个标准, 除了那些超出标准范围的例子, 比如例子 9-2.

只有在Bash或*sh*提示符下, 当shell脚本退出后, 在命令行上使用$?才会得到与上表相一致的结果. 在某些情况下, 运行C-shell或者*tcsh*可能会给出不同的值.

注意事项

[1] 超出范围的退出值可能会产生意想不到的退出码. 如果退出值比255大, 那么退出码将会取256的模. 举个例子, exit 3809的退出码将是225(3809 % 256 = 225).

Appendix E. I/O和I/O重定向的详细介绍

由Stephane Chazelas编写，本书作者修订

一个命令期望前3个文件描述符是可用的. 第一个，*fd 0*(标准输入，stdin)，用作读取. 另外两个，(*fd 1*，stdout和*fd 2*，stderr)，用来写入.

每个命令都会关联到stdin，stdout，和stderr. ls 2>&1意味着临时的将ls命令的stderr连接到shell的stdout.

按惯例，命令一般都是从fd 0(stdin)上读取输入，打印输出到fd 1(stdout)上，错误输出一般都输出到fd 2(stderr)上. 如果这3个文件描述中的某一个没打开，你可能就会遇到麻烦了:

```
bash$ cat /etc/passwd >&-
cat: standard output: Bad file descriptor
```

比如说，当xterm运行的时候，它首先会初始化自身. 在运行用户shell之前，xterm会打开终端设备(/dev/pts/<n> 或者类似的东西)三次.

这里，Bash继承了这三个文件描述符，而且每个运行在Bash上的命令(子进程)也都依次继承了它们，除非你重定向了这些命令. 重定向意味着将这些文件描述符中的某一个，重新分配到其他文件中(或者分配到一个管道中，或者是其他任何可能的东西). 文件描述符既可以被局部重分配(对于一个命令，命令组，一个子shell，一个while循环，if或case结构...)，也可以全局重分配，对于余下的shell(使用exec).

ls > /dev/null 表示将运行的ls命令的fd 1连接到/dev/null上.

```
bash$ lsof -a -p $$ -d0,1,2
COMMAND PID     USER   FD   TYPE DEVICE SIZE NODE NAME
 bash    363 bozo    0u   CHR  136,1         3 /dev/pts/1
 bash    363 bozo    1u   CHR  136,1         3 /dev/pts/1
 bash    363 bozo    2u   CHR  136,1         3 /dev/pts/1


bash$ exec 2> /dev/null
bash$ lsof -a -p $$ -d0,1,2
COMMAND PID     USER   FD   TYPE DEVICE SIZE NODE NAME
 bash    371 bozo    0u   CHR  136,1         3 /dev/pts/1
 bash    371 bozo    1u   CHR  136,1         3 /dev/pts/1
 bash    371 bozo    2w   CHR    1,3       120 /dev/null


bash$ bash -c 'lsof -a -p $$ -d0,1,2' | cat
COMMAND PID USER   FD   TYPE DEVICE SIZE NODE NAME
 lsof    379 root    0u   CHR  136,1         3 /dev/pts/1
 lsof    379 root    1w  FIFO    0,0      7118 pipe
 lsof    379 root    2u   CHR  136,1         3 /dev/pts/1


bash$ echo "$(bash -c 'lsof -a -p $$ -d0,1,2' 2>&1)"
COMMAND PID USER   FD   TYPE DEVICE SIZE NODE NAME
 lsof    426 root    0u   CHR  136,1         3 /dev/pts/1
 lsof    426 root    1w  FIFO    0,0      7520 pipe
 lsof    426 root    2w  FIFO    0,0      7520 pipe
```

这是用来展示不同类型的重定向.

练习：分析下面的脚本.

```
  1 #! /usr/bin/env bash
  2
  3 mkfifo /tmp/fifo1 /tmp/fifo2
  4 while read a; do echo "FIFO1: $a"; done < /tmp/fifo1 &
  5 exec 7> /tmp/fifo1
  6 exec 8> >(while read a; do echo "FD8: $a, to fd7"; done >&7)
  7
  8 exec 3>&1
  9 (
 10  (
 11   (
 12    while read a; do echo "FIFO2: $a"; done < /tmp/fifo2 | tee /dev/stderr | tee /dev/fd/4 | tee /dev/fd/5 | tee
/dev/fd/6 >&7 &
 13    exec 3> /tmp/fifo2
 14
 15    echo 1st, to stdout
 16    sleep 1
 17    echo 2nd, to stderr >&2
 18    sleep 1
 19    echo 3rd, to fd 3 >&3
 20    sleep 1
 21    echo 4th, to fd 4 >&4
 22    sleep 1
 23    echo 5th, to fd 5 >&5
 24    sleep 1
 25    echo 6th, through a pipe | sed 's/.*/PIPE: &, to fd 5/' >&5
 26    sleep 1
 27    echo 7th, to fd 6 >&6
 28    sleep 1
 29    echo 8th, to fd 7 >&7
 30    sleep 1
 31    echo 9th, to fd 8 >&8
 32
 33   ) 4>&1 >&3 3>&- | while read a; do echo "FD4: $a"; done 1>&3 5>&- 6>&-
 34  ) 5>&1 >&3 | while read a; do echo "FD5: $a"; done 1>&3 6>&-
 35 ) 6>&1 >&3 | while read a; do echo "FD6: $a"; done 3>&-
 36
 37 rm -f /tmp/fifo1 /tmp/fifo2
 38
 39
 40 # 对于每个命令和子shell，分别指出每个fd的指向.
 41
 42 exit 0
```

Appendix F. 命令行选项

目录
F.1. 标准命令行选项
F.2. Bash命令行选项

许多可执行文件，不管是二进制可执行文件还是脚本文件，都可以使用选项来修改它们运行时的行为. 比如：在命令行上键入command -o，那么就意味着使用选项o来调用*command*.

F.1. 标准命令行选项

随着时间的流逝，对于命令行选项标志的含义来说，已经建立起了一套比较宽松的标准. GNU工具比老式的UNIX工具更加符合这套"标准".

按惯例，UNIX命令行选项通常都包含一个破折号，后边跟一个或多个小写字母. GNU工具增加了一个双破折号，后边跟一个完整的单词或复合单词.

这两个最通用的选项是:

- `-h`

  `--help`

  帮助: 给出使用信息，然后退出.

- `-v`

  `--version`

  版本: 现实程序版本号，然后退出.

其他公用选项:

- `-a`

  `--all`

  全部: 显示*所有*参数的*全部*信息或操作.

- `-l`

  `--list`

  列表: 列出文件或参数，不采取其他动作.

- `-o`

  *输出*文件

- `-q`

  `--quiet`

  安静: 抑制`stdout`.

- `-r`

  `-R`

  `--recursive`

  递归: 递归操作(包含子目录树).

- `-v`

  `--verbose`

  冗余: 将额外的信息输出到`stdout`或`stderr`.

- `-z`

  `--compress`

压缩: 进行压缩(通常为gzip).

然而:

- 在tar和gawk中:

  `-f`

  `--file`

  文件: 跟文件名参数.

- 在cp, mv, rm中:

  `-f`

  `--force`

  强制: 目标文件的强制覆盖.

许多UNIX和Linux工具都严重的偏离了这个"标准", 所以, 按照标准来*假定*一个给定选项的行为是非常危险的. 当遇到拿不准的问题时, 一定要经常察看命令的man页.

GNU工具有一张完整的推荐选项表, 在http://www.gnu.org/prep/standards_19.html.

F.2. Bash命令行选项

*Bash*本身也带有许多命令行选项. 下面就是一些有用的选项.

- `-c`

  从这个选项后边的字符串中读取命令, 并且将参数分配到位置参数中.

  ```
  bash$ bash -c 'set a b c d; IFS="+-;"; echo "$*"'
  a+b+c+d
  ```

- `-r`

  `--restricted`

  使用受限模式运行这个shell, 或脚本.

- `--posix`

  强制Bash符合POSIX模式.

- `--version`

  显示Bash版本并退出.

- `--`

  选项的结束. 命令行上的其他东西就都是参数了, 不是选项.

Appendix G. 重要的文件

启动文件

这些文件包含别名和环境变量，正是这些别名和环境变量才使得Bash可以作为一个用户shell来运行，当系统初始化之后，这些别名和变量也可被其他的的Bash脚本调用.

`/etc/profile`

系统范围的默认值，大部分用来设置环境(所有的Bourne类型的shell，而不仅仅是Bash [1])

`/etc/bashrc`

特定于Bash的，系统范围函数与别名

`$HOME/.bash_profile`

用户定义的，环境默认设置，在每个用户的home目录下都可找到(本地副本保存在`/etc/profile`)

`$HOME/.bashrc`

用户定义的Bash初始化文件，可以在每个用户的home目录下找到(本地副本保存在`/etc/bashrc`). 只有交互式的shell和用户脚本才会读取这个文件. 请参考Appendix K，这是一个`.bashrc`文件的例子.

登出文件

`$HOME/.bash_logout`

用户定义的指令文件，在每个用户的home目录下找到. 在登出(Bash)shell的时候，这个文件中的命令就会得到执行.

注意事项

[1] 不能应用于csh，tcsh，或那些与经典Bourne shell无关的shell(也就是说那些不是派生自sh的shell).

Appendix H. 重要的系统目录

每位系统管理员或者是编写系统管理脚本的人员都应该对这些系统目录非常熟悉.

- /bin

  二进制(可执行文件). 基本的系统程序和工具(比如bash).

- /usr/bin [1]

  更多的系统二进制可执行文件.

- /usr/local/bin

  一些局部于特定机器的杂项二进制可执行文件.

- /sbin

  系统二进制可执行文件. 基本的系统管理程序和工具(比如fsck).

- /usr/sbin

  更多的系统管理程序和工具.

- /etc

  其他. 系统范围的配置脚本.

  其中比较有趣的文件是/etc/fstab(文件系统表), /etc/mtab(挂载文件系统表), 还有文件/etc/inittab.

- /etc/rc.d

  启动脚本, 适用于红帽及其派生的Linux发行版.

- /usr/share/doc

  安装包的文档.

- /usr/man

  系统范围的man页.

- /dev

  设备目录. 物理设备和虚拟设备的入口(但*不是*挂载点). 请参考 27.

- /proc

  进程目录. 包含关于运行进程和内核参数的统计信息与其他信息. 请参考 27.

- /sys

  系统范围的设备目录. 包含关于设备和设备名称的统计信息与其他信息. 这是在Linux 2.6.X内核版本上新添加的目录.

- /mnt

  挂载. 挂载硬驱动分区的目录, 比如/mnt/dos, 和物理驱动器. 在比较新的Linux发行版中, /media目录已经成为了I/O设备的首选挂载点.

- `/media`

  在比较新的Linux发行版中, I/O设备的首选挂载点, 比如CD ROM或USB flash驱动器.

- `/var`

  可变的(可修改的)系统文件. 这是一个包罗万象的"杂项"目录, 用于保存Linux/UNIX机器运行时产生的各种数据.

- `/var/log`

  系统范围的日志文件.

- `/var/spool/mail`

  用户的假脱机邮件(mail spool).

- `/lib`

  系统范围的库文件.

- `/usr/lib`

  更多系统范围的库文件.

- `/tmp`

  系统临时文件.

- `/boot`

  系统*引导*目录. 内核, 模块链接, 系统镜像, 和引导管理器都放在这.

  如果在这个目录下修改文件, 可能会导致系统不能启动.

注意事项

[1] 早期的UNIX系统一般都有两个磁盘设备, 一个是速度快但容量小的硬盘(主要包含`/`, 即根目录), 另一个磁盘容量大, 但是速度慢(主要包含`/usr`目录和其他分区). 所以, 使用频率最高的程序和工具都放到小而快的磁盘中, 也就是放到`/bin`中, 而其他的东西都放到慢磁盘上, 即`/usr/bin`中.

其他的类似的东西也是按照这种方式进行分类的, 比如`/sbin`和`/usr/sbin`, `/lib`和`/usr/lib`, 等等.

Appendix I. 本地化

本地化是Bash的一个未文档化的特征.

对于一个本地化的shell脚本来说，它的输出都会使用本地系统所定义的语言. 对于一个德国柏林的Linux用户来说，他的脚本会输出德文，而对于他在马里兰的堂兄来说，同样运行这个脚本，输出就是英文.

为了创建一个本地化的脚本，可以使用下面的模版来编写所有的用户消息(错误消息，提示符，等等.).

```
#!/bin/bash
# localized.sh
#  此脚本由Stephane Chazelas编写,
#+ Bruno Haible进行了修改, Alfredo Pironti修正了bug.

. gettext.sh

E_CDERROR=65

error()
{
  printf "$@" >&2
  exit $E_CDERROR
}

cd $var || error "`eval gettext \"Can\'t cd to \\\$var.\"`"
#  $var前面之所以需要三个反斜线(转义)
#+ "因为在变量值还没被替换之前,
#+ eval_gettext需要一个字符串."
#    -- per Bruno Haible
read -p "`gettext \"Enter the value: \"`" var
#  ...


#  ------------------------------------------------------------------
#  Alfredo Pironti注释:

#  这个脚本已经被修改,
#+ 使用"`gettext \"...\"`"语法形式替换了$"..."语法形式.
#  这么做没问题, 但是在新的localized.sh程序中,
#+ 命令"bash -D filename" and "bash --dump-po-string filename"
#+ 将不会产生输出
#+ (因为那些命令只会搜索$"..."字符串)!
#  从新文件中提取字符串的唯一方法就是使用'xgettext'程序.
#  然而, xgettext程序存在许多bug.

# 注意'xgettext'还有一个bug.
#
# shell片断:
#    gettext -s "I like Bash"
# 将会被正确的提取, 但是 . . .
#    xgettext -s "I like Bash"
# . . . 失败!
#  'xgettext'将会提取"-s"
#+ 因为这个命令仅仅会提取
#+ 'gettext'后边的第一个参数.


#  转义字符:
#
#  为了本地化一个句子, 就像
#    echo -e "Hello\tworld!"
#+ 你必须使用
#    echo -e "`gettext \"Hello\\tworld\"`"
#  `t'前边的"双转义字符"是必须的,
#+ 因为'gettext'将会搜索那些字符串(就像'Hello\tworld')
#  这是因为gettext将会读取一个字符`\')
#+ 并将输出一个字符串(就像"Bonjour\tmonde"),
#+ 所以'echo'命令将会正确的显示消息.
#
#  你可能不想使用
#    echo "`gettext -e \"Hello\tworld\"`"
#+ 因为我们上面解释的xgettext的bug.

```

```
65
66
67 # 让我们本地化下面的shell片断:
68 #     echo "-h display help and exit"
69 #
70 # 首先, 可以用:
71 #     echo "`gettext \"-h display help and exit\"`"
72 #  这样'xgettext'工作正常,
73 #+ 但是'gettext'程序将会把"-h"当作选项来读取!
74 #
75 # 一个解决方法是
76 #     echo "`gettext -- \"-h display help and exit\"`"
77 #  这样'gettext'工作正常,
78 #+ 但是'xgettext'将会提取"--", 就像上边那样.
79 #
80 # 为了获得这个本地化的字符串, 你可能使用的变通方法就是:
81 #     echo -e "`gettext \"\\0-h display help and exit\"`"
82 #  我们已经在这句的开头添加了\0 (NULL).
83 #  这样'gettext'能够正确工作, 就像'xgettext'一样.
84 #  此外, NULL字符将不会修改
85 #+ 'echo'命令的行为.
86 #  --------------------------------------------------------------------
```

```
bash$ bash -D localized.sh
"Can't cd to %s."
 "Enter the value: "
```

这将列出所有的本地化文本. (-D选项将会列出以$为前缀, 并且使用双引号引用起来的字符串, 而不会执行这个脚本.)

```
bash$ bash --dump-po-strings localized.sh
#: a:6
 msgid "Can't cd to %s."
 msgstr ""
 #: a:7
 msgid "Enter the value: "
 msgstr ""
```

Bash的--dump-po-strings选项与-D选项很相似, 但使用gettext "po"格式.

Bruno Haible指出:

以gettext-0.12.2开始, xgettext -o - localized.sh被推荐代替bash --dump-po-strings localized.sh, 因为xgettext . . .

1. 了解命令gettext和eval_gettext(而bash --dump-po-strings只认识它的$"..."语法)
2. 可以提取程序中的注释, 进而可以被翻译者读取.

这个脚本将不再被特定于Bash, 它与Bash 1.x和其他的/bin/sh实现, 都使用相同的方式工作.

现在, 为每种脚本需要被转换的语言都建立一个language.po文件, 指定*msgstr*. Alfredo Pironti给出了下面的例子:

fr.po:

```
 1 #: a:6
 2 msgid "Can't cd to $var."
 3 msgstr "Impossible de se positionner dans le repertoire $var."
 4 #: a:7
 5 msgid "Enter the value: "
 6 msgstr "Entrez la valeur : "
 7
 8 #  这个字符串和变量名被打印, 没有%s语法,
 9 #+ 与C程序很像.
10 #+ 如果程序员使用有意义的变量名,
11 #+ 那么这将会是一个非常酷的特点!
```

然后, 运行msgfmt.

```
msgfmt -o localized.sh.mo fr.po
```

将文件localized.sh.mo的结果放到/usr/local/share/locale/fr/LC_MESSAGES目录下, 并且在脚本的开头插入如下行:

```
1 TEXTDOMAINDIR=/usr/local/share/locale
```

```
2 TEXTDOMAIN=localized.sh
```

如果法文系统上的用户运行这个脚本，那么她将得到法文消息.

在老本的Bash或其他shell中，本地化需要使用-s选项的命令gettext. 在这种情况下，脚本为:

```
 1 #!/bin/bash
 2 # localized.sh
 3
 4 E_CDERROR=65
 5
 6 error() {
 7   local format=$1
 8   shift
 9   printf "$(gettext -s "$format")" "$@" >&2
10   exit $E_CDERROR
11 }
12 cd $var || error "Can't cd to %s." "$var"
13 read -p "$(gettext -s "Enter the value: ")" var
14 # ...
```

变量TEXTDOMAIN和TEXTDOMAINDIR需要被设置，并且需要export到环境变量中. 这应该在脚本中完成.

此附录由Stephane Chazelas编写，Alfredo Pironti，和Bruno Haible给出了一些建议，是GNUgettext的维护者.

Appendix J. 历史命令

Bash shell提供命令行工具用于编辑和操作用户的*命令历史*. 这其实主要就是为了方便, 节省用户的重复按键.

Bash历史命令:

1. history
2. fc

```
bash$ history
   1  mount /mnt/cdrom
    2  cd /mnt/cdrom
    3  ls
     ...
```

与Bash历史命令相关的内部变量:

1. $HISTCMD
2. $HISTCONTROL
3. $HISTIGNORE
4. $HISTFILE
5. $HISTFILESIZE
6. $HISTSIZE
7. $HISTTIMEFORMAT (Bash 3.0或后续版本)
8. !!
9. !$
10. !#
11. !N
12. !-N
13. !STRING
14. !?STRING?
15. ^STRING^string^

不幸的是, Bash历史工具在脚本中没用.

```
1 #!/bin/bash
2 # history.sh
3 # 尝试在脚本中使用'history'命令.
4
5 history
6
7 # 脚本没产生输出.
8 # 历史命令不能工作在脚本中.
```

```
bash$ ./history.sh
(no output)
```

站点Advancing in the Bash Shell给出了一份关于如何在Bash中使用历史命令的详细介绍.

Appendix K. 一个简单的.bashrc文件

~/.bashrc文件决定了交互shell的行为. 好好的了解这个文件, 将会使你更加了解Bash.

Emmanuel Rouat捐献了下边这个注释非常详细的.bashrc文件, 这个文件是为Linux系统编写的. 他希望读者能够给他一些回馈.

仔细的学习这个文件, 直到你可以自由重用其中的代码片断和函数, 并把它们用到你自己的.bashrc文件中, 甚至可以放到你的脚本中.

例子 K-1. **.bashrc**文件样本

```
#=============================================================
#
# 个人的$HOME/.bashrc文件, 基于bash-2.05a(或更高版本)
#
# 最后更新日期: 星期2 4月15 20:32:34 CEST 2003
#
# 这个文件(一般情况下)被只会被交互式shell读取.
# 这里可以定义你的别名, 函数,
# 和其他的一些交互式特征, 比如你的提示符.
#
# 这个文件(开始时)是为Solaris设计的,
# 但是基于Redhat的默认.bashrc文件
# --> 为Linux修改.
# 你在这里看到的大部分代码都是从网上找来的
# (即internet).
# 这个bashrc文件有点挤 -
# 记住, 它仅仅是个例子而已. 按照你自己的需求进行裁减.
#
#
#=============================================================

# --> 注释由HOWTO的作者添加.
# --> 然后又被ER编辑了一下 :-)

#----------------------------------------
# 如果有源代码的全局定义, 请在此处定义.
#----------------------------------------

if [ -f /etc/bashrc ]; then
        . /etc/bashrc   # --> 读取/etc/bashrc, 如果存在的话.
fi

#---------------------------------------------------------------
# $DISPLAY的自动设置 (如果还没设置的话)
# 这用于linux - 可能运行的结果不同....
# 问题是不同的终端种类对于'who am i'来说,
# 将会给出不同的答案.....
# 我还没发现一种'通用'方法
#---------------------------------------------------------------

function get_xserver ()
{
    case $TERM in
    xterm )
            XSERVER=$(who am i | awk '{print $NF}' | tr -d ')''(' )
            # Ane-Pieter Wieringa建议使用下面这种方式:
            # I_AM=$(who am i)
            # SERVER=${I_AM#*(}
            # SERVER=${SERVER%*)}

            XSERVER=${XSERVER%%:*}
        ;;
    aterm | rxvt)
    # 找出一些运行在这里的代码.....
        ;;
    esac
}

if [ -z ${DISPLAY:=""} ]; then
    get xserver
```

```
 61     if [[ -z ${XSERVER}  || ${XSERVER} == $(hostname) || ${XSERVER} == "unix" ]];
then
 62     DISPLAY=":0.0"           # 在本地主机上显示
 63     else
 64     DISPLAY=${XSERVER}:0.0  # 在远端主机上显示
 65     fi
 66 fi
 67
 68 export DISPLAY
 69
 70 #----------
 71 # 一些设置
 72 #----------
 73
 74 ulimit -S -c 0               # 不需要任何coredump
 75 set -o notify
 76 set -o noclobber
 77 set -o ignoreeof
 78 set -o nounset
 79 #set -o xtrace          # 对于调试来说非常有用
 80
 81 # 使能选项:
 82 shopt -s cdspell
 83 shopt -s cdable_vars
 84 shopt -s checkhash
 85 shopt -s checkwinsize
 86 shopt -s mailwarn
 87 shopt -s sourcepath
 88 shopt -s no_empty_cmd_completion  # 仅限于bash>=2.04
 89 shopt -s cmdhist
 90 shopt -s histappend histreedit histverify
 91 shopt -s extglob    # 对于complete命令(按情况补全)来说是必要的
 92
 93 # 禁用选项:
 94 shopt -u mailwarn
 95 unset MAILCHECK              # 当有邮件到达时, 我不希望我的shell提示我
 96
 97
 98 export TIMEFORMAT=$'\nreal %3R\tuser %3U\tsys %3S\tpcpu %P\n'
 99 export HISTIGNORE="&:bg:fg:ll:h"
100 export HOSTFILE=$HOME/.hosts        # 将远端主机的列表放入~/.hosts
101
102
103
104 #------------------------
105 # 问候, 问侯报文等等...
106 #------------------------
107
108 # 先定义一些颜色:
109 red='\e[0;31m'
110 RED='\e[1;31m'
111 blue='\e[0;34m'
112 BLUE='\e[1;34m'
113 cyan='\e[0;36m'
114 CYAN='\e[1;36m'
115 NC='\e[0m'               # 没有颜色
116 # --> 很好. 与使用"ansi.sys"的DOS效果相同.
117
118 # 在黑色背景下看起来非常好.....
119 echo -e "${CYAN}This is BASH ${RED}${BASH_VERSION%.*}${CYAN} - DISPLAY on
${RED}$DISPLAY${NC}\n"
120 date
121 if [ -x /usr/games/fortune ]; then
122     /usr/games/fortune -s     # 让我们的每天充满乐趣.... :-)
123 fi
124
125 function _exit()    # 在退出shell时运行的函数
126 {
127     echo -e "${RED}Hasta la vista, baby${NC}"
128 }
129 trap _exit EXIT
130
131 #---------------
132 # Shell提示符
133 #---------------
134
135 if [[ "${DISPLAY#$HOST}" != ":0.0" &&  "${DISPLAY}" != ":0" ]]; then
136     HILIT=${red}   # 远端主机: 提示符为红
137 else
138     HILIT=${cyan}  # 本地主机: 提示符为青色
139 fi
140
141 #  --> 下面提示符函数中\W和\w的替换实例,
142 #+ --> 用来获得完整路径名的显示.
```

```
143
144 function fastprompt()
145 {
146     unset PROMPT_COMMAND
147     case $TERM in
148         *term | rxvt )
149             PS1="${HILIT}[\h]$NC \W > \[\033]0;\${TERM} [\u@\h] \w\007\]" ;;
150     linux )
151         PS1="${HILIT}[\h]$NC \W > " ;;
152         *)
153             PS1="[\h] \W > " ;;
154     esac
155 }
156
157 function powerprompt()
158 {
159     _powerprompt()
160     {
161         LOAD=$(uptime|sed -e "s/.*: \([^,]*\).*/\1/" -e "s/ //g")
162     }
163
164     PROMPT_COMMAND=_powerprompt
165     case $TERM in
166         *term | rxvt  )
167             PS1="${HILIT}[\A \$LOAD]$NC\n[\h \#] \W > \[\033]0;\${TERM} [\u@\h]
\w\007\]" ;;
168         linux )
169             PS1="${HILIT}[\A - \$LOAD]$NC\n[\h \#] \w > " ;;
170         * )
171             PS1="[\A - \$LOAD]\n[\h \#] \w > " ;;
172     esac
173 }
174
175 powerprompt     # 这是默认提示符 - 可能比较慢
176                 # 如果很慢的话, 可以使用fastprompt来代替....
177
178 #================================================================
179 #
180 # 别名和函数
181 #
182 # 事实上, 这里定义的一些函数非常大
183 # (比如'lowercase'), 但是我的机器是512M内存, 所以 .....
184 # 如果你想让这个文件小一点,
185 # 可以将这些函数放到脚本中.
186 #
187 # 其中的许多函数来自于bash-2.04
188 # 中的例子.
189 #
190 #================================================================
191
192 #--------------------
193 # 个人的别名
194 #--------------------
195
196 alias rm='rm -i'
197 alias cp='cp -i'
198 alias mv='mv -i'
199 # -> 防止偶然的文件误操作.
200 alias mkdir='mkdir -p'
201
202 alias h='history'
203 alias j='jobs -l'
204 alias r='rlogin'
205 alias which='type -all'
206 alias ..='cd ..'
207 alias path='echo -e ${PATH//:/\\n}'
208 alias print='/usr/bin/lp -o nobanner -d $LPDEST'    # 假设LPDEST被定义
209 alias pjet='enscript -h -G -fCourier9 -d $LPDEST'   # 使用enscript的漂亮的打印
210 alias background='xv -root -quit -max -rmode 5'     # 将一张图片作为背景
211 alias du='du -kh'
212 alias df='df -kTh'
213
214 # 'ls'家族 (假定使用GNU ls)
215 alias la='ls -Al'                  # 显示隐藏文件
216 alias ls='ls -hF --color'    # 为识别的文件类型添加颜色
217 alias lx='ls -lXB'                 # 按扩展名排序
218 alias lk='ls -lSr'                 # 按尺寸排序
219 alias lc='ls -lcr'               # 按修改时间排序
220 alias lu='ls -lur'               # 按访问时间排序
221 alias lr='ls -lR'                  # 递归ls
222 alias lt='ls -ltr'                 # 按日期排序
223 alias lm='ls -al |more'            # 管道给'more'
224 alias tree='tree -Csu'                 # 'ls'的另一种好方法
225
226 # 裁减'less'
```

```
alias more='less'
export PAGER=less
export LESSCHARSET='latin1'
export LESSOPEN='|/usr/bin/lesspipe.sh %s 2>&-' # 如果lesspipe.sh存在, 就用这个
export LESS='-i -N -w  -z-4 -g -e -M -X -F -R -P%t?f%f \
:stdin .?pb%pb\%:?lbLine %lb:?bbByte %bb:-...'

# 拼写错误 - 纯粹个人喜好 :-)
alias xs='cd'
alias vf='cd'
alias moer='more'
alias moew='more'
alias kk='ll'

#----------------
# 一些有趣东西
#----------------

function xtitle ()
{
    case "$TERM" in
        *term | rxvt)
            echo -n -e "\033]0;$*\007" ;;
        *)
        ;;
    esac
}

# 别名...
alias top='xtitle Processes on $HOST && top'
alias make='xtitle Making $(basename $PWD) ; make'
alias ncftp="xtitle ncFTP ; ncftp"

# .. 和函数
function man ()
{
    for i ; do
    xtitle The $(basename $1|tr -d .[:digit:]) manual
    command man -F -a "$i"
    done
}

function ll(){ ls -l "$@"| egrep "^d" ; ls -lXB "$@" 2>&-| egrep -v "^d|total ";
}
function te()  # xemacs/gnuserv的包装器
{
    if [ "$(gnuclient -batch -eval t 2>&-)" == "t" ]; then
        gnuclient -q "$@";
    else
        ( xemacs "$@" &);
    fi
}

#----------------------------
# 与文件和字符串相关的函数:
#----------------------------

# 使用名字模式来查找文件:
function ff() { find . -type f -iname '*'$*'*' -ls ; }
# 使用pattern $1和Execute $2来查找文件:
function fe() { find . -type f -iname '*'$1'*' -exec "${2:-file}" {} \;  ; }
# 在一系列文件中找到模式, 并高亮
function fstr()
{
    OPTIND=1
    local case=""
    local usage="fstr: find string in files.
Usage: fstr [-i] \"pattern\" [\"filename pattern\"] "
    while getopts :it opt
    do
        case "$opt" in
        i) case="-i " ;;
        *) echo "$usage"; return;;
        esac
    done
    shift $(( $OPTIND - 1 ))
    if [ "$#" -lt 1 ]; then
        echo "$usage"
        return;
    fi
    local SMSO=$(tput smso)
    local RMSO=$(tput rmso)
    find . -type f -name "${2:-*}" -print0 | xargs -0 grep -sn ${case} "$1" 2>&-
| \
sed "s/$1/${SMSO}\0${RMSO}/gI" | more
```

```
}

function cuttail() # 在文件中切掉n行, 默认为10行
{
    nlines=${2:-10}
    sed -n -e :a -e "1,${nlines}!{P;N;D;};N;ba" $1
}

function lowercase()  # 将文件名转换为小写
{
    for file ; do
        filename=${file##*/}
        case "$filename" in
        */*) dirname==${file%/*} ;;
        *) dirname=.;;
        esac
        nf=$(echo $filename | tr A-Z a-z)
        newname="${dirname}/${nf}"
        if [ "$nf" != "$filename" ]; then
            mv "$file" "$newname"
            echo "lowercase: $file --> $newname"
        else
            echo "lowercase: $file not changed."
        fi
    done
}

function swap()         # 交换两个文件名
{
    local TMPFILE=tmp.$$
    mv "$1" $TMPFILE
    mv "$2" "$1"
    mv $TMPFILE "$2"
}


#-----------------------
# 进程/系统相关的函数:
#-----------------------

function my_ps() { ps $@ -u $USER -o pid,%cpu,%mem,bsdtime,command ; }
function pp() { my_ps f | awk '!/awk/ && $0~var' var=${1:-".*"} ; }

# 这个函数与linux上的'killall'基本一致
# 但是与Solaris上的却不相同
function killps()   # 按进程名进行kill
{
    local pid pname sig="-TERM"   # 默认signal
    if [ "$#" -lt 1 ] || [ "$#" -gt 2 ]; then
        echo "Usage: killps [-SIGNAL] pattern"
        return;
    fi
    if [ $# = 2 ]; then sig=$1 ; fi
    for pid in $(my_ps| awk '!/awk/ && $0~pat { print $1 }' pat=${!#} ) ; do
        pname=$(my_ps | awk '$1~var { print $5 }' var=$pid )
        if ask "Kill process $pid <$pname> with signal $sig?"
            then kill $sig $pid
        fi
    done
}

function my_ip() # 获得IP地址
{
    MY_IP=$(/sbin/ifconfig ppp0 | awk '/inet/ { print $2 } ' | sed -e s/addr://)
    MY_ISP=$(/sbin/ifconfig ppp0 | awk '/P-t-P/ { print $3 } ' | sed -e s/P-t-
P://)
}

function ii()   # 获得当前主机相关的信息
{
    echo -e "\nYou are logged on ${RED}$HOST"
    echo -e "\nAdditionnal information:$NC " ; uname -a
    echo -e "\n${RED}Users logged on:$NC " ; w -h
    echo -e "\n${RED}Current date :$NC " ; date
    echo -e "\n${RED}Machine stats :$NC " ; uptime
    echo -e "\n${RED}Memory stats :$NC " ; free
    my_ip 2>&- ;
    echo -e "\n${RED}Local IP Address :$NC" ; echo ${MY_IP:-"Not connected"}
    echo -e "\n${RED}ISP Address :$NC" ; echo ${MY_ISP:-"Not connected"}
    echo
}

# 杂项工具:

function repeat()       # 重复n次的命令
```

```
{
    local i max
    max=$1; shift;
    for ((i=1; i <= max ; i++)); do  # --> C风格的语法
        eval "$@";
    done
}

function ask()
{
    echo -n "$@" '[y/n] ' ; read ans
    case "$ans" in
        y*|Y*) return 0 ;;
        *) return 1 ;;
    esac
}

#============================================================================
#
# 按情况补全, complete命令 - BASH-2.04及其后续版本
# 大部分摘自bash 2.05文档
# 和Ian McDonalds的'Bash completion'软件
包(http://www.caliban.org/bash/index.shtml#completion)
# 某些特征可能需要使用bash-2.05a
#
#============================================================================

if [ "${BASH_VERSION%.*}" \< "2.05" ]; then
    echo "You will need to upgrade to version 2.05 for programmable completion"
    return
fi

shopt -s extglob        # 必须的
set +o nounset          # 否则某些自动补全将会失败

complete -A hostname   rsh rcp telnet rlogin r ftp ping disk
complete -A export     printenv
complete -A variable   export local readonly unset
complete -A enabled    builtin
complete -A alias      alias unalias
complete -A function   function
complete -A user       su mail finger

complete -A helptopic  help     # 通常与内建命令一样
complete -A shopt      shopt
complete -A stopped -P '%' bg
complete -A job -P '%'     fg jobs disown

complete -A directory  mkdir rmdir
complete -A directory   -o default cd

# 压缩
complete -f -o default -X '*.+(zip|ZIP)'  zip
complete -f -o default -X '!*.+(zip|ZIP)' unzip
complete -f -o default -X '*.+(z|Z)'      compress
complete -f -o default -X '!*.+(z|Z)'     uncompress
complete -f -o default -X '*.+(gz|GZ)'    gzip
complete -f -o default -X '!*.+(gz|GZ)'   gunzip
complete -f -o default -X '*.+(bz2|BZ2)'  bzip2
complete -f -o default -X '!*.+(bz2|BZ2)' bunzip2
# Postscript,pdf,dvi.....(译者: 打印格式相关)
complete -f -o default -X '!*.ps'  gs ghostview ps2pdf ps2ascii
complete -f -o default -X '!*.dvi' dvips dvipdf xdvi dviselect dvitype
complete -f -o default -X '!*.pdf' acroread pdf2ps
complete -f -o default -X '!*.+(pdf|ps)' gv
complete -f -o default -X '!*.texi*' makeinfo texi2dvi texi2html texi2pdf
complete -f -o default -X '!*.tex' tex latex slitex
complete -f -o default -X '!*.lyx' lyx
complete -f -o default -X '!*.+(htm*|HTM*)' lynx html2ps
# 多媒体
complete -f -o default -X '!*.+(jp*g|gif|xpm|png|bmp)' xv gimp
complete -f -o default -X '!*.+(mp3|MP3)' mpg123 mpg321
complete -f -o default -X '!*.+(ogg|OGG)' ogg123



complete -f -o default -X '!*.pl'  perl perl5

# 这是一个'通用的'补全函数 - 当命令具有一个所谓的"长选项"模式it works when commands have
# 的时候, 它就会工作, 比如: 'ls --all' 代替 'ls -a'

_get_longopts ()
{
    $1 --help | sed  -e '/--/!d' -e 's/.*--\([^[:space:].,]*\).*/--\1/'| \
grep ^"$2" |sort -u ;
```

```
}

_longopts_func ()
{
    case "${2:-*}" in
    -*)     ;;
    *)      return ;;
    esac

    case "$1" in
    \~*)    eval cmd="$1" ;;
    *)      cmd="$1" ;;
    esac
    COMPREPLY=( $(_get_longopts ${1} ${2} ) )
}
complete  -o default -F _longopts_func configure bash
complete  -o default -F _longopts_func wget id info a2ps ls recode


_make_targets ()
{
    local mdef makef gcmd cur prev i

    COMPREPLY=()
    cur=${COMP_WORDS[COMP_CWORD]}
    prev=${COMP_WORDS[COMP_CWORD-1]}

    # 如果之前的参数为-f, 那就返回可能的补全文件名.
    # 我们可以让它更智能一些, 并且返回匹配的
    # `makefile Makefile *.mk', 不管存在与否
    case "$prev" in
        -*f)    COMPREPLY=( $(compgen -f $cur ) ); return 0;;
    esac

    # 如果我们需要一个选项, 那就返回可能的posix选项
    case "$cur" in
        -)      COMPREPLY=(-e -f -i -k -n -p -q -r -S -s -t); return 0;;
    esac

    # 前尝试`makefile'再尝试`Makefile'
    if [ -f makefile ]; then
        mdef=makefile
    elif [ -f Makefile ]; then
        mdef=Makefile
    else
        mdef=*.mk               # 局部约定
    fi

    # 在我们扫描目标文件之前, 察看makefile文件名是否
    # 使用-f指定
    for (( i=0; i < ${#COMP_WORDS[@]}; i++ )); do
        if [[ ${COMP_WORDS[i]} == -*f ]]; then
            eval makef=${COMP_WORDS[i+1]}       # eval for tilde expansion(波浪号
扩展)
            break
        fi
    done

        [ -z "$makef" ] && makef=$mdef

    # 如果我们有特别偏爱的补全单词,
    # 那么可以限制的补全这个单词
    if [ -n "$2" ]; then gcmd='grep "^$2"' ; else gcmd=cat ; fi

    # 如果我们不想使用*.mk, 我们可以使用
    # 或者使用test -f $makef或者使用输入重定向
    COMPREPLY=( $(cat $makef 2>/dev/null | awk 'BEGIN {FS=":"} /^[^.#   ][^=]*:/
{print $1}' | tr -s ' ' '\012' | sort -u | eval $gcmd ) )
}

complete -F _make_targets -X '+($*|*.[cho])' make gmake pmake


# cvs(1) 补全
_cvs ()
{
    local cur prev
    COMPREPLY=()
    cur=${COMP_WORDS[COMP_CWORD]}
    prev=${COMP_WORDS[COMP_CWORD-1]}

    if [ $COMP_CWORD -eq 1 ] || [ "${prev:0:1}" = "-" ]; then
        COMPREPLY=( $( compgen -W 'add admin checkout commit diff \
        export history import log rdiff release remove rtag status \
        tag update' $cur ))
```

```
    else
        COMPREPLY=( $( compgen -f $cur ))
    fi
    return 0
}
complete -F _cvs cvs

_killall ()
{
    local cur prev
    COMPREPLY=()
    cur=${COMP_WORDS[COMP_CWORD]}

    # 获得进程列表(第一个sed表达式处理
    # swap out出去的进程, 第二个
    # 获得进程的basename)
    COMPREPLY=( $( /usr/bin/ps -u $USER -o comm  | \
        sed -e '1,1d' -e 's#[]\[]##g' -e 's#^.*/##'| \
        awk '{if ($0 ~ /^'$cur'/) print $0}' ))

    return 0
}

complete -F _killall killall killps


# 一个元命令补全函数, 用于sudo(8)这种命令,
# 需要先对这个命令进行补全, 然后需要按照这个命令自己的补全定义进行补全
#  - 当前并不是非常可靠(比如 mount和umount命令
# 就不能很好的工作), 但还是很有用的 - 作者, Ian McDonald, 我修改了一下.

_my_command()
{
    local cur func cline cspec

    COMPREPLY=()
    cur=${COMP_WORDS[COMP_CWORD]}

    if [ $COMP_CWORD = 1 ]; then
    COMPREPLY=( $( compgen -c $cur ) )
    elif complete -p ${COMP_WORDS[1]} &>/dev/null; then
    cspec=$( complete -p ${COMP_WORDS[1]} )
    if [ "${cspec%%-F *}" != "${cspec}" ]; then
        # complete -F <function>
        #
        # COMP_CWORD和COMP_WORDS()不是只读的,
        # 所以我们可以在传递到补全例程之前,
        # 设置它们

        # 设置当前的标志号减1
        COMP_CWORD=$(( $COMP_CWORD - 1 ))
        # 获得函数名
        func=${cspec#*-F }
        func=${func%% *}
        # 获得去掉第一个命令后的命令行
        cline="${COMP_LINE#$1 }"
        # 分离当前命令, 传递给数组
            COMP_WORDS=( $cline )
        $func $cline
    elif [ "${cspec#*-[abcdefgjkvu]}" != "" ]; then
        # complete -[abcdefgjkvu]
        #func=$( echo $cspec | sed -e 's/^.*\(-[abcdefgjkvu]\).*$/\1/' )
        func=$( echo $cspec | sed -e 's/^complete//' -e 's/[^ ]*$//' )
        COMPREPLY=( $( eval compgen $func $cur ) )
    elif [ "${cspec#*-A}" != "$cspec" ]; then
        # complete -A <type>
        func=${cspec#*-A }
    func=${func%% *}
    COMPREPLY=( $( compgen -A $func $cur ) )
    fi
    else
    COMPREPLY=( $( compgen -f $cur ) )
    fi
}


complete -o default -F _my_command nohup exec eval trace truss strace sotruss gdb
complete -o default -F _my_command command type which man nice

# 本地变量:
# mode:shell-script
# sh-shell:bash
# End:
```

Appendix L. 将DOS批处理文件转换为Shell脚本

相当多的在PC上学习脚本的程序员都在运行DOS. 事实上, 残废的DOS批处理文件语言还是可以编写出一些比较强大的脚本来的, 虽然它们一般都需要借助于外部的工具. 所以说, 某些时候, 我们还是需要将老式的DOS批处理文件转换为UNIX shell脚本. 一般来说, 做这种事情并不困难, 因为DOS批处理文件操作不过是等价的shell脚本的一个受限子集.

表格 L-1. 批处理文件关键字 / 变量 / 操作符, 和等价的shell符号

批处理文件操作符	Shell脚本等价符号	含义
%	$	命令行参数前缀
/	-	命令选项标记
\	/	目录路径分隔符
==	=	(等于)字符串比较测试
!==!	!=	(不等)字符串比较测试
|	|	管道
@	set +v	不打印当前命令
*	*	文件名"通配符"
>	>	文件重定向(覆盖)
>>	>>	文件重定向(附加)
<	<	重定向stdin
%VAR%	$VAR	环境变量
REM	#	注释
NOT	!	取反
NUL	/dev/null	"黑洞"用来阻止命令输出
ECHO	echo	打印(Bash中有更多选项)
ECHO.	echo	打印空行
ECHO OFF	set +v	不打印后续的命令
FOR %%VAR IN (LIST) DO	for var in [list]; do	"for"循环
:LABEL	没有等价物(多余)	标签
GOTO	没有等价物(使用函数)	跳转到脚本的另一个位置
PAUSE	sleep	暂停或等待一段时间
CHOICE	case or select	菜单选择
IF	if	if条件语句
IF EXIST *FILENAME*	if [-e filename]	测试文件是否存在
IF !%N==!	if [-z "$N"]	参数"N"是否存在
CALL	source命令或.(点操作符)	"include"另一个脚本
COMMAND /C	source命令或.(点操作符)	"include"另一个脚本(与CALL相同)
SET	export	设置一个环境变量
SHIFT	shift	左移命令行参数列表
SGN	-lt或-gt	(整形)符号

ERRORLEVEL	$?	退出状态
CON	stdin	"控制台"(stdin)
PRN	/dev/lp0	(一般的)打印设备
LPT1	/dev/lp0	第一个打印设备
COM1	/dev/ttyS0	第一个串口

批处理文件一般都包含DOS命令. 我们必须把它转换为UNIX的等价命令, 这样我们才能把批处理文件转换为shell脚本文件.

表格 L-2. DOS命令与UNIX的等价命令

DOS命令	UNIX等价命令	效果
ASSIGN	ln	链接文件或目录
ATTRIB	chmod	修改文件权限
CD	cd	更换目录
CHDIR	cd	更换目录
CLS	clear	清屏
COMP	diff, comm, cmp	文件比较
COPY	cp	文件拷贝
Ctl-C	Ctl-C	中断(信号)
Ctl-Z	Ctl-D	EOF(文件结束)
DEL	rm	删除文件
DELTREE	rm -rf	递归删除目录
DIR	ls -l	列出目录内容
ERASE	rm	删除文件
EXIT	exit	退出当前进程
FC	comm, cmp	文件比较
FIND	grep	在文件中查找字符串
MD	mkdir	新建目录
MKDIR	mkdir	新建目录
MORE	more	分页显示文本文件
MOVE	mv	移动文件
PATH	$PATH	可执行文件的路径
REN	mv	重命名(移动)
RENAME	mv	重命名(移动)
RD	rmdir	删除目录
RMDIR	rmdir	删除目录
SORT	sort	排序文件
TIME	date	显示系统时间
TYPE	cat	将文件输出到stdout
XCOPY	cp	(扩展的)文件拷贝

事实上, 几乎所有的UNIX和shell操作符, 还有命令都有许多的选项, 对比DOS和批处理文件来说, 它们要强大的多. 许多DOS批处理文件都需要依靠辅助工具, 比如ask.com, 这是一个比read命令差很多的类似副本.

DOS对于文件名通配符扩展支持的非常有限, 并且很不完整, 仅仅识别*和?.

将DOS批处理文件转换为sehll脚本, 通常是一件很简单的事情, 而且转换的结果通常都比原始的批处理文件好.

例子 L-1. VIEWDATA.BAT: DOS批处理文件

```
REM VIEWDATA

REM 灵感来自于例子"DOS POWERTOOLS"
REM                               PAUL SOMERSON编写


@ECHO OFF

IF !%1==! GOTO VIEWDATA
REM  如果没有命令行参数...
FIND "%1" C:\BOZO\BOOKLIST.TXT
GOTO EXIT0
REM  打印出字符串匹配的行, 然后退出.

:VIEWDATA
TYPE C:\BOZO\BOOKLIST.TXT | MORE
REM  显示整个文件, 一次一页.

:EXIT0
```

转换脚本作了一些改进.

例子 L-2. viewdata.sh: 转换自VIEWDATA.BAT的shell脚本

```
#!/bin/bash
# viewdata.sh
# 转换自VIEWDATA.BAT的shell脚本.

DATAFILE=/home/bozo/datafiles/book-collection.data
ARGNO=1

# @ECHO OFF                   这个命令在这里就不需要了.

if [ $# -lt "$ARGNO" ]    # IF !%1==! GOTO VIEWDATA
then
  less $DATAFILE          # TYPE C:\MYDIR\BOOKLIST.TXT | MORE
else
  grep "$1" $DATAFILE     # FIND "%1" C:\MYDIR\BOOKLIST.TXT
fi

exit 0                    # :EXIT0

#  跳转, 标签, 还有其他一些小手段, 在shell脚本中就不需要了.
#  我们可以说, 转换后的脚本比原始批处理文件好的多,
#+ 它更短, 看起来更整洁, 更优雅.
```

Ted Davis的Shell Scripts on the PC站点上有许多关于老式的批处理文件编程的教程, 他使用的某些独创性的技术, 和shell脚本有异曲同工之妙.

Appendix M. 练习

目录
M.1. 分析脚本
M.2. 编写脚本

M.1. 分析脚本

检查下面的脚本. 运行它, 然后解释一下这个脚本是做什么用的. 注释这个脚本, 并以更紧凑和更优雅的形式重写它.

```
#!/bin/bash

MAX=10000


  for((nr=1; nr<$MAX; nr++))
  do

    let "t1 = nr % 5"
    if [ "$t1" -ne 3 ]
    then
      continue
    fi

    let "t2 = nr % 7"
    if [ "$t2" -ne 4 ]
    then
      continue
    fi

    let "t3 = nr % 9"
    if [ "$t3" -ne 5 ]
    then
      continue
    fi

  break    # 当你注释掉这行, 会发生什么? 为什么?

  done

  echo "Number = $nr"


exit 0
```

解释一下下面脚本的作用. 事实上它只是一个参数化的命令行管道.

```
#!/bin/bash

DIRNAME=/usr/bin
FILETYPE="shell script"
LOGFILE=logfile

file "$DIRNAME"/* | fgrep "$FILETYPE" | tee $LOGFILE | wc -l

exit 0
```

一个读者发来了如下的代码片断.

```
while read LINE
do
  echo $LINE
done < `tail -f /var/log/messages`
```

他希望编写一个脚本, 用来跟踪系统日志文件(`/var/log/messages`)的更新情况. 不幸的是, 上面的这段代码会被挂起, 并且不会做任何有意义的事情. 为什么? 修复它, 让这个脚本如期望般的运行. (小提示: 不要重定向循环的`stdin`, 试试管道.)

分析例子 A-10, 然后简化它, 使其逻辑性更好. 看看可以省掉多少个变量, 尝试优化这个脚本, 并提高这个脚本的运行速度.

修改这个脚本，让它可以接受任意的ASCII文本文件作为输入，用于它初始的"产生". 这个脚本将读取最初的`$ROW*$COL`字符，并且设置元音的出现次数作为"活的"细胞. 提示：必须保证将输入文件中的空格转换为下划线.

M.2. 编写脚本

编写脚本来完成下面列出的每项任务.

译者：家庭作业，就不译了.

EASY

Home Directory Listing

Perform a recursive directory listing on the user's home directory and save the information to a file. Compress the file, have the script prompt the user to insert a floppy, then press ENTER. Finally, save the file to the floppy.

Converting for loops to while and until loops

Convert the *for loops* in 例子 10-1 to *while loops*. Hint: store the data in an array and step through the array elements.

Having already done the "heavy lifting", now convert the loops in the example to *until loops*.

Changing the line spacing of a text file

Write a script that reads each line of a target file, then writes the line back to `stdout`, but with an extra blank line following. This has the effect of *double-spacing* the file.

Include all necessary code to check whether the script gets the necessary command line argument (a filename), and whether the specified file exists.

When the script runs correctly, modify it to *triple-space* the target file.

Finally, write a script to remove all blank lines from the target file, *single-spacing* it.

Backwards Listing

Write a script that echoes itself to `stdout`, but *backwards*.

Automatically Decompressing Files

Given a list of filenames as input, this script queries each target file (parsing the output of the file command) for the type of compression used on it. Then the script automatically invokes the appropriate decompression command (gunzip, bunzip2, unzip, uncompress, or whatever). If a target file is not compressed, the script emits a warning message, but takes no other action on that particular file.

Unique System ID

Generate a "unique" 6-digit hexadecimal identifier for your computer. Do *not* use the flawed hostid command. Hint: md5sum **/etc/passwd**, then select the first 6 digits of output.

Backup

Archive as a "tarball" (`*.tar.gz` file) all the files in your home directory tree

(`/home/your-name`) that have been modified in the last 24 hours. Hint: use find.

Checking whether a process is still running

Given a process ID (*PID*) as an argument, this script will check, at user-specified intervals, whether the given process is still running. You may use the ps and sleep commands.

Primes

Print (to stdout) all prime numbers between 60000 and 63000. The output should be nicely formatted in columns (hint: use printf).

Lottery Numbers

One type of lottery involves picking five different numbers, in the range of 1 - 50. Write a script that generates five pseudorandom numbers in this range, *with no duplicates*. The script will give the option of echoing the numbers to `stdout` or saving them to a file, along with the date and time the particular number set was generated.

INTERMEDIATE

Integer or String

Write a script function that determines if an argument passed to it is an integer or a string. The function will return TRUE (0) if passed an integer, and FALSE (1) if passed a string.

Hint: What does the following expression return when `$1` is *not* an integer?

```
expr $1 + 0
```

Managing Disk Space

List, one at a time, all files larger than 100K in the `/home/username` directory tree. Give the user the option to delete or compress the file, then proceed to show the next one. Write to a logfile the names of all deleted files and the deletion times.

Removing Inactive Accounts

Inactive accounts on a network waste disk space and may become a security risk. Write an administrative script (to be invoked by *root* or the cron daemon) that checks for and deletes user accounts that have not been accessed within the last 90 days.

Enforcing Disk Quotas

Write a script for a multi-user system that checks users' disk usage. If a user surpasses the preset limit (100 MB, for example) in her `/home/username` directory, then the script will automatically send her a warning e-mail.

The script will use the du and mail commands. As an option, it will allow setting and enforcing quotas using the quota and setquota commands.

Logged in User Information

For all logged in users, show their real names and the time and date of their last login.

Hint: use who, lastlog, and parse `/etc/passwd`.

Safe Delete

Write, as a script, a "safe" delete command, `srm.sh`. Filenames passed as command-line arguments to this script are not deleted, but instead gzipped if not already compressed (use file to check), then moved to a `/home/username/trash` directory. At invocation, the script checks the "trash" directory for files older than 48 hours and deletes them.

Making Change

What is the most efficient way to make change for $1.68, using only coins in common circulations (up to 25c)? It's 6 quarters, 1 dime, a nickel, and three cents.

Given any arbitrary command line input in dollars and cents ($*.??), calculate the change, using the minimum number of coins. If your home country is not the United States, you may use your local currency units instead. The script will need to parse the command line input, then change it to multiples of the smallest monetary unit (cents or whatever). Hint: look at 例子 23-8.

Quadratic Equations

Solve a "quadratic" equation of the form *Ax^2 + Bx + C = 0*. Have a script take as arguments the coefficients, `A`, `B`, and `C`, and return the solutions to four decimal places.

Hint: pipe the coefficients to bc, using the well-known formula, *x = (-B +/- sqrt(B^2 - 4AC)) / 2A*.

Sum of Matching Numbers

Find the sum of all five-digit numbers (in the range 10000 - 99999) containing *exactly two* out of the following set of digits: { 4, 5, 6 }. These may repeat within the same number, and if so, they count once for each occurrence.

Some examples of matching numbers are 42057, 74638, and 89515.

Lucky Numbers

A "lucky number" is one whose individual digits add up to 7, in successive additions. For example, 62431 is a "lucky number" (6 + 2 + 4 + 3 + 1 = 16, 1 + 6 = 7). Find all the "lucky numbers" between 1000 and 10000.

Alphabetizing a String

Alphabetize (in ASCII order) an arbitrary string read from the command line.

Parsing

Parse `/etc/passwd`, and output its contents in nice, easy-to-read tabular form.

Logging Logins

Parse `/var/log/messages` to produce a nicely formatted file of user logins and login times. The script may need to run as root. (Hint: Search for the string "LOGIN.")

Pretty-Printing a Data File

Certain database and spreadsheet packages use save-files with *comma-separated values* (CSVs). Other applications often need to parse these files.

Given a data file with comma-separated fields, of the form:

```
1 Jones,Bill,235 S. Williams St.,Denver,CO,80221,(303) 244-7989
2 Smith,Tom,404 Polk Ave.,Los Angeles,CA,90003,(213) 879-5612
3 ...
```

Reformat the data and print it out to `stdout` in labeled, evenly-spaced columns.

Justification

Given ASCII text input either from `stdin` or a file, adjust the word spacing to right-justify each line to a user-specified line-width, then send the output to `stdout`.

Mailing List

Using the mail command, write a script that manages a simple mailing list. The script automatically e-mails the monthly company newsletter, read from a specified text file, and sends it to all the addresses on the mailing list, which the script reads from another specified file.

Generating Passwords

Generate pseudorandom 8-character passwords, using characters in the ranges [0-9], [A-Z], [a-z]. Each password must contain at least two digits.

Checking for Broken Links

Using lynx with the `-traversal` option, write a script that checks a Web site for broken links.

DIFFICULT

Testing Passwords

Write a script to check and validate passwords. The object is to flag "weak" or easily guessed password candidates.

A trial password will be input to the script as a command line parameter. To be considered acceptable, a password must meet the following minimum qualifications:

- Minimum length of 8 characters
- Must contain at least one numeric character
- Must contain at least one of the following non-alphabetic characters: @, #, $, %, &, *, +, -, =

Optional:

- Do a dictionary check on every sequence of at least four consecutive alphabetic characters in the password under test. This will eliminate passwords containing embedded "words" found in a standard dictionary.
- Enable the script to check all the passwords on your system. These may or may not reside in `/etc/passwd`.

This exercise tests mastery of Regular Expressions.

Logging File Accesses

Log all accesses to the files in `/etc` during the course of a single day. This information should include the filename, user name, and access time. If any alterations to the files take place, that should be flagged. Write this data as neatly formatted records in a logfile.

Monitoring Processes

Write a script to continually monitor all running processes and to keep track of how many child processes each parent spawns. If a process spawns more than five children, then the script sends an e-mail to the system administrator (or root) with all relevant information, including the time, PID of the parent, PIDs of the

children, etc. The script writes a report to a log file every ten minutes.

Strip Comments

Strip all comments from a shell script whose name is specified on the command line. Note that the "#! line" must not be stripped out.

HTML Conversion

Convert a given text file to HTML. This non-interactive script automatically inserts all appropriate HTML tags into a file specified as an argument.

Strip HTML Tags

Strip all HTML tags from a specified HTML file, then reformat it into lines between 60 and 75 characters in length. Reset paragraph and block spacing, as appropriate, and convert HTML tables to their approximate text equivalent.

XML Conversion

Convert an XML file to both HTML and text format.

Chasing Spammers

Write a script that analyzes a spam e-mail by doing DNS lookups on the IP addresses in the headers to identify the relay hosts as well as the originating ISP. The script will forward the unaltered spam message to the responsible ISPs. Of course, it will be necessary to filter out *your own ISP's IP address*, so you don't end up complaining about yourself.

As necessary, use the appropriate network analysis commands.

For some ideas, see 例子 12-37 and 例子 A-28.

Optional: Write a script that searches through a batch of e-mail messages and deletes the spam according to specified filters.

Creating man pages

Write a script that automates the process of creating *man pages*.

Given a text file which contains information to be formatted into a *man page*, the script will read the file, then invoke the appropriate groff commands to output the corresponding *man page* to `stdout`. The text file contains blocks of information under the standard *man page* headings, i.e., "NAME," "SYNOPSIS," "DESCRIPTION," etc.

See 例子 12-26.

Morse Code

Convert a text file to Morse code. Each character of the text file will be represented as a corresponding Morse code group of dots and dashes (underscores), separated by whitespace from the next. For example, "script" ===> "... _._. ._. .. .__. _".

Hex Dump

Do a hex(adecimal) dump on a binary file specified as an argument. The output should be in neat tabular fields, with the first field showing the address, each of the next 8 fields a 4-byte hex number, and the final field the ASCII equivalent of the previous 8 fields.

Emulating a Shift Register

Using 例子 26-14 as an inspiration, write a script that emulates a 64-bit shift register as an array. Implement functions to *load* the register, *shift left*, *shift right*, and *rotate* it. Finally, write a function that interprets the register contents as eight 8-bit ASCII characters.

Determinant

Solve a 4 x 4 determinant.

Hidden Words

Write a "word-find" puzzle generator, a script that hides 10 input words in a 10 x 10 matrix of random letters. The words may be hidden across, down, or diagonally.

Optional: Write a script that *solves* word-find puzzles. To keep this from becoming too difficult, the solution script will find only horizontal and vertical words. (Hint: Treat each row and column as a string, and search for substrings.)

Anagramming

Anagram 4-letter input. For example, the anagrams of *word* are: *do or rod row word*. You may use `/usr/share/dict/linux.words` as the reference list.

"Word Ladders"

A "word ladder" is a sequence of words, with each successive word in the sequence differing from the previous one by a single letter.

For example, to "ladder" from *mark* to *vase*:

```
1 mark --> park --> part --> past --> vast --> vase
```

Write a script that solves "word ladder" puzzles. Given a starting and an ending word, the script will list all intermediate steps in the "ladder". Note that *all* words in the sequence must be "legal."

Fog Index

The "fog index" of a passage of text estimates its reading difficulty, as a number corresponding roughly to a school grade level. For example, a passage with a fog index of 12 should be comprehensible to anyone with 12 years of schooling.

The Gunning version of the fog index uses the following algorithm.

1. Choose a section of the text at least 100 words in length.
2. Count the number of sentences (a portion of a sentence truncated by the boundary of the text section counts as one).
3. Find the average number of words per sentence.

   AVE_WDS_SEN = TOTAL_WORDS / SENTENCES
4. Count the number of "difficult" words in the segment -- those containing at least 3 syllables. Divide this quantity by total words to get the proportion of difficult words.

   PRO_DIFF_WORDS = LONG_WORDS / TOTAL_WORDS
5. The Gunning fog index is the sum of the above two quantities, multiplied by 0.4, then rounded to the nearest integer.

   G_FOG_INDEX = int (0.4 * (AVE_WDS_SEN + PRO_DIFF_WORDS))

Step 4 is by far the most difficult portion of the exercise. There exist various algorithms for estimating the syllable count of a word. A rule-of-thumb formula might consider the number of letters in a word and the vowel-consonant mix.

A strict interpretation of the Gunning fog index does not count compound words and proper nouns as "difficult" words, but this would enormously complicate the script.

Calculating PI using Buffon's Needle

The Eighteenth Century French mathematician de Buffon came up with a novel experiment. Repeatedly drop a needle of length "n" onto a wooden floor composed of long and narrow parallel boards. The cracks separating the equal-width floorboards are a fixed distance "d" apart. Keep track of the total drops and the number of times the needle intersects a crack on the floor. The ratio of these two quantities turns out to be a fractional multiple of PI.

In the spirit of 例子 12-45, write a script that runs a Monte Carlo simulation of Buffon's Needle. To simplify matters, set the needle length equal to the distance between the cracks, $n = d$.

Hint: there are actually two critical variables: the distance from the center of the needle to the crack nearest to it, and the angle of the needle to that crack. You may use bc to handle the calculations.

Playfair Cipher

Implement the Playfair (Wheatstone) Cipher in a script.

The Playfair Cipher encrypts text by substitution of "digrams" (2-letter groupings). It is traditional to use a 5 x 5 letter scrambled-alphabet *key square* for the encryption and decryption.

```
  1    C O D E S
  2    A B F G H
  3    I K L M N
  4    P Q R T U
  5    V W X Y Z
  6
  7 Each letter of the alphabet appears once, except "I" also represents
  8 "J". The arbitrarily chosen key word, "CODES" comes first, then all
  9 the rest of the alphabet, in order from left to right, skipping
letters
 10 already used.
 11
 12 To encrypt, separate the plaintext message into digrams (2-letter
 13 groups). If a group has two identical letters, delete the second, and
 14 form a new group. If there is a single letter left over at the end,
 15 insert a "null" character, typically an "X."
 16
 17 THIS IS A TOP SECRET MESSAGE
 18
 19 TH IS IS AT OP SE CR ET ME SA GE
 20
 21 For each digram, there are three possibilities.
 22 ------------------------------------------------
 23 1) Both letters will be on the same row of the key square
 24    For each letter, substitute the one immediately to the right, in
that
 25    row. If necessary, wrap around left to the beginning of the row.
 26
 27 or
 28
 29 2) Both letters will be in the same column of the key square
 30    For each letter, substitute the one immediately below it, in that
 31    row. If necessary, wrap around to the top of the column.
 32
 33 or
 34
 35 3) Both letters will form the corners of a rectangle within the key
 36    square. For each letter, substitute the one on the other corner the
 37    rectangle which lies on the same row.
 38
 39
 40 The "TH" digram falls under case #3.
```

```
 41 G H
 42 M N
 43 T U           (Rectangle with "T" and "H" at corners)
 44
 45 T --> U
 46 H --> G
 47
 48
 49 The "SE" digram falls under case #1.
 50 C O D E S     (Row containing "S" and "E")
 51
 52 S --> C  (wraps around left to beginning of row)
 53 E --> S
 54

55 ================================================================================
 56
 57 To decrypt encrypted text, reverse the above procedure under cases #1
 58 and #2 (move in opposite direction for substitution). Under case #3,
 59 just take the remaining two corners of the rectangle.
 60
 61
 62 Helen Fouche Gaines' classic work, ELEMENTARY CRYPTANALYSIS (1939),
gives a
 63 fairly detailed rundown on the Playfair Cipher and its solution
methods.
```

This script will have three main sections

I. Generating the "key square", based on a user-input keyword.

II. Encrypting a "plaintext" message.

III. Decrypting encrypted text.

The script will make extensive use of arrays and functions.

--

Please do not send the author your solutions to these exercises. There are better ways to impress him with your cleverness, such as submitting bugfixes and suggestions for improving this book.

Appendix N. 修订记录

这份文档起始于2000年春天的一份60页的HOWTO文档.
从那以后, 它经历数次升级与修订.
如果没有Linux社团的帮助, 那就不可能有这本书,
在这里尤其要感谢Linux Documentation Project的志愿者们.

表格 N-1. 修订历史

版本	日期	注释
0.1	14 Jun 2000	Initial release.
0.2	30 Oct 2000	Bugs fixed, plus much additional material and more example scripts.
0.3	12 Feb 2001	Major update.
0.4	08 Jul 2001	Complete revision and expansion of the book.
0.5	03 Sep 2001	Major update: Bugfixes, material added, sections reorganized.
1.0	14 Oct 2001	Stable release: Bugfixes, reorganization, material added.
1.1	06 Jan 2002	Bugfixes, material and scripts added.
1.2	31 Mar 2002	Bugfixes, material and scripts added.
1.3	02 Jun 2002	TANGERINE release: A few bugfixes, much more material and scripts added.
1.4	16 Jun 2002	MANGO release: A number of typos fixed, more material and scripts.
1.5	13 Jul 2002	PAPAYA release: A few bugfixes, much more material and scripts added.
1.6	29 Sep 2002	POMEGRANATE release: Bugfixes, more material, one more script.
1.7	05 Jan 2003	COCONUT release: A couple of bugfixes, more material, one more script.
1.8	10 May 2003	BREADFRUIT release: A number of bugfixes, more scripts and material.
1.9	21 Jun 2003	PERSIMMON release: Bugfixes, and more material.
2.0	24 Aug 2003	GOOSEBERRY release: Major update.
2.1	14 Sep 2003	HUCKLEBERRY release: Bugfixes, and more material.
2.2	31 Oct 2003	CRANBERRY release: Major update.
2.3	03 Jan 2004	STRAWBERRY release: Bugfixes and more material.
2.4	25 Jan 2004	MUSKMELON release: Bugfixes.
	15 Feb	

2.5	2004	STARFRUIT release: Bugfixes and more material.
2.6	15 Mar 2004	SALAL release: Minor update.
2.7	18 Apr 2004	MULBERRY release: Minor update.
2.8	11 Jul 2004	ELDERBERRY release: Minor update.
3.0	03 Oct 2004	LOGANBERRY release: Major update.
3.1	14 Nov 2004	BAYBERRY release: Bugfix update.
3.2	06 Feb 2005	BLUEBERRY release: Minor update.
3.3	20 Mar 2005	RASPBERRY release: Bugfixes, much material added.
3.4	08 May 2005	TEABERRY release: Bugfixes, stylistic revisions.
3.5	05 Jun 2005	BOXBERRY release: Bugfixes, some material added.
3.6	28 Aug 2005	POKEBERRY release: Bugfixes, some material added.
3.7	23 Oct 2005	WHORTLEBERRY release: Bugfixes, some material added.
3.8	26 Feb 2006	BLAEBERRY release: Bugfixes, some material added.
3.9	15 May 2006	SPICEBERRY release: Bugfixes, some material added.

Appendix O. 翻译版修订记录

翻译版起始于我对这本书的一份学习笔记(译者: 杨春敏), 随着笔记的增长,
我萌生了翻译它的想法, 并在linuxsir的北南南北的帮助下, 发布了出来.
与此并行的是另一译者(黄毅)也同时在翻译此书, 我们两个那时翻译进度相仿.
因此在我发布之后, 我们两个在论坛上结识, 并相约共同翻译此书.
由此, 对应于原书3.75的翻译beta版终于面世了.
最后, 由我在beta版的基础上对正文重新较稿, 并共同努力产出了对应于原书3.91的正式翻译版.

表格 O-1. 翻译版修订历史

版本	日期	注释
纯文本前10章	2006-01-08	我第一次放出的纯文本翻译版, 基于我最初的学习笔记, 对应于原书的3.75版.
纯文本beta版	2006-05-15	基于我原来的学习笔记最后完成的纯文本版. 对应于原书的3.75版.
html的beta版	2006-05-30	测试版中最完整的版本, 前2部分主要以我的纯文本版为基础, 并大量参考了黄毅的前2部分内容, 第3部分由我翻译, 第4部分由黄毅翻译. 并最终由黄毅较稿完成发布.
1.0__3.91	2007-03-04	正月15完成, 第一个翻译完整版, 与原书作者采用相同的sgml发布. 由我重新较稿并使用sgml的docbook形式重写, 黄毅修正了其中的许多问题.

Appendix P. 镜像站点

这份文档最后的更新，是以"tar包"的形式发布的，其中包括SGML源码和编译好的HTML版本，你可以从作者的主页上下载.

这份文档最主要的镜像站点就是Linux Documentation Project，这个站点也维护了相当多的指南手册和HOWTO文档.

Sunsite/Metalab/ibiblio.org也是*ABS指南*的一个镜像站点.

这份文档的另一个镜像站点是morethan.org.

Appendix Q. To Do列表

- 全面调查Bash与经典的Bourne shell之间的兼容性.
- 同上, 但需要调查的是Korn shell (ksh).
- 一个使用Bash来进行CGI编程的初级读本.

  下面是一个简单的CGI脚本, 你可以从这里开始.

例子 Q-1. 打印服务器环境

```
#!/bin/bash
# 可能需要修改你的站点位置.
# (在ISP的服务器上, Bash可能不在标准位置/bin.)
# 其他可能出现的地方: /usr/bin或/usr/local/bin
# 甚至可以不带任何路径信息来尝试使用#!.

# test-cgi.sh
# 由Michael Zick编写
# 经过授权在此使用


# 禁用文件名匹配.
set -f

# 头信息将会给浏览器需要的东西.
echo Content-type: text/plain
echo

echo CGI/1.0 test script report:
echo

echo environment settings:
set
echo

echo whereis bash?
whereis bash
echo


echo who are we?
echo ${BASH_VERSINFO[*]}
echo

echo argc is $#. argv is "$*".
echo

# CGI/1.0需要的环境变量.

echo SERVER_SOFTWARE = $SERVER_SOFTWARE
echo SERVER_NAME = $SERVER_NAME
echo GATEWAY_INTERFACE = $GATEWAY_INTERFACE
echo SERVER_PROTOCOL = $SERVER_PROTOCOL
echo SERVER_PORT = $SERVER_PORT
echo REQUEST_METHOD = $REQUEST_METHOD
echo HTTP_ACCEPT = "$HTTP_ACCEPT"
echo PATH_INFO = "$PATH_INFO"
echo PATH_TRANSLATED = "$PATH_TRANSLATED"
echo SCRIPT_NAME = "$SCRIPT_NAME"
echo QUERY_STRING = "$QUERY_STRING"
echo REMOTE_HOST = $REMOTE_HOST
echo REMOTE_ADDR = $REMOTE_ADDR
echo REMOTE_USER = $REMOTE_USER
echo AUTH_TYPE = $AUTH_TYPE
echo CONTENT_TYPE = $CONTENT_TYPE
echo CONTENT_LENGTH = $CONTENT_LENGTH

exit 0

# Here document可以给出简要的使用说明.
```

```
61 :<<-'_test_CGI_'
62
63 1) Drop this in your http://domain.name/cgi-bin directory.
64 2) Then, open http://domain.name/cgi-bin/test-cgi.sh.
65
66 _test_CGI_
```

有志愿者么?

Appendix R. 版权

The *Advanced Bash Scripting Guide* is copyright © 2000, by Mendel Cooper. The author also asserts copyright on all previous versions of this document.

This blanket copyright recognizes and protects the rights of all contributors to this document.

This document may only be distributed subject to the terms and conditions set forth in the Open Publication License (version 1.0 or later), http://www.opencontent.org/openpub/. The following license options also apply.

```
A.  Distribution of substantively modified versions of this document
    is prohibited without the explicit permission of the copyright holder.
    HOWEVER, in the event that the author or maintainer of this document
    cannot be contacted, the Linux Documentation Project will have the right
    to take over custodianship of the document and name a new maintainer,
    who would then have the right to update and modify the document.

B.  This document may NOT be distributed encrypted or with any form of
    DRM (Digital Rights Management) embedded in it. Nor may this document
    be bundled with other DRM-ed works.

C.  Distribution of the work or derivative of the work in any standard
    (paper) book form is prohibited unless prior permission is obtained from
    the copyright holder.
```

Provision A, above, explicitly prohibits *relabeling* this document. An example of relabeling is the insertion of company logos or navigation bars into the cover, title page, or the text. The author grants the following exemptions.

1. Non-profit organizations, such as the Linux Documentation Project and Sunsite.
2. "Pure-play" Linux distributors, such as Debian, Red Hat, Mandrake, SuSE, and others.

Without explicit written permission from the author, distributors and publishers (including on-line publishers) are prohibited from imposing any additional conditions, strictures, or provisions on this document or any previous version of it. As of this update, the author asserts that he has *not* entered into any contractual obligations that would alter the foregoing declarations.

Essentially, you may freely distribute this book in *unaltered* electronic form. You must obtain the author's permission to distribute a substantially modified version or derivative work. The purpose of this restriction is to preserve the artistic integrity of this document and to prevent "forking."

If you display or distribute this document or any previous version thereof under any license except the one above, then you are required to obtain the author's written permission. Failure to do so may terminate your distribution rights.

These are very liberal terms, and they should not hinder any legitimate distribution or use of this book. The author especially encourages the use of this book for classroom and instructional purposes.

Certain of the scripts contained in this document are, where noted, released into the Public Domain. These scripts are exempt from the foregoing license and copyright restrictions.

The commercial print and other rights to this book are available. Please contact the author if interested.

The author produced this book in a manner consistent with the spirit of the LDP Manifesto.

Linux is a trademark registered to Linus Torvalds.

Unix and UNIX are trademarks registered to the Open Group.

MS Windows is a trademark registered to the Microsoft Corp.

Solaris is a trademark registered to Sun, Inc.

OSX is a trademark registered to Apple, Inc.

Yahoo is a trademark registered to Yahoo, Inc.

Pentium is a trademark registered to Intel, Inc.

Scrabble is a trademark registered to Hasbro, Inc.

All other commercial trademarks mentioned in the body of this work are registered to their respective owners.

Hyun Jin Cha has done a Korean translation of version 1.0.11 of this book. Spanish, Portuguese, French, (another French), German, Italian, Russian, Czech, Chinese, and Dutch translations are also available or in progress. If you wish to translate this document into another language, please feel free to do so, subject to the terms stated above. The author wishes to be notified of such efforts.

前一页 首页
To Do列表