

Bash Guide for Beginners 中文版

Machtelt Garrels

Xalasy.com

<tille wants no spam _at_ xalasy dot com>

Version 1.7 Last updated 20050905 Edition

翻译 by

Wei Wang

<wangwei0609 at gmail dot com>

Version 1.7 Last updated 20060105 Edition

介绍.....	2
1. 本指南诞生原因	2
2. 谁应该阅读此书	2
3. 新版本和取得方法.....	2
4. 历史修订	2
5. 贡献.....	2
6. 反馈.....	2
7. 版权信息	2
8. 你需要什么?	2
9. 文档约定	2
10. 本文档结构.....	2
第一章 Bash 和 Bash 脚本	2
1.1. 普通 shell 程序.....	2
1.1.1. shell 的作用	2
1.1.2. shell 类型.....	2
1.2. Bash 的优势	2
1.2.1. Bash 是 GNU Shell	2
1.2.2. Bash 独有的特性	2
1.3. 执行命令	2
1.3.1. 概要	2
1.3.2. Shell 内部命令	2
1.3.3. 从脚本执行程序.....	2
1.4. 建立块.....	2
1.4.1. Shell 建立块.....	2
1.5. 开发优良脚本.....	2
1.5.1. 优良脚本的要素	2
1.5.2. 结构	2
1.5.3. 术语	2
1.5.4. 有序且有逻辑性.....	2
1.5.5. 一个 Bash 脚本的例子: mysystem.sh.....	2
1.5.6. 初始化脚本例子.....	2
1.6. 总结	2
1.7. 练习	2
第二章 编写和调试脚本.....	2
2.1. 建立并且运行一个脚本	2
2.1.1. 编写与命名	2
2.1.2. Script1.sh	2
2.1.3. 执行脚本.....	2
2.2. 脚本基础	2
2.2.1. 哪个 Shell 来执行脚本	2
2.2.2. 加入注释.....	2
2.3. 调试 Bash 脚本	2
2.3.1. 调试整个脚本	2
2.3.2. 调试部分脚本	2

2.4. 总结	2
2.5. 练习	2
第三章 Bash 环境	2
3.1. Shell 初始化文件	2
3.1.1. 跨系统配置文件	2
3.1.2. 单独用户配置文件	2
3.1.3. 改变 shell 配置文件	2
3.2. 变量	2
3.2.1. 变量的类型	2
3.2.2. 建立变量	2
3.2.3. 导出变量	2
3.2.4. 保留变量	2
3.2.5. 特殊参数	2
3.2.6. 脚本通过变量循环	2
3.3. 引用字符	2
3.3.1. 为何?	2
3.3.2. 转义字符	2
3.3.3. 单引用	2
3.3.4. 双引用	2
3.3.5. ANSI-C 引用	2
3.3.6. Locales..... 46	2
3.4. Shell 扩展	2
3.4.1. 概要	2
3.4.2. 大括号表达式	2
3.4.3. 波浪表达式	2
3.4.4. Shell 参数和变量扩展	2
3.4.5. 命令替换	2
3.4.6. 算术表达式	2
3.4.7. 替换的处理	2
3.4.8. Word splitting	2
3.4.9. 文件名扩展	2
3.5. 别名	2
3.5.1. 什么是别名	2
3.5.2. 建立和消除别名	2
3.6. 更多 Bash 选项	2
3.6.1. 显示选项	2
3.6.2. 改变选项	2
3.7. 总结	2
3.8. 练习	2
第四章 正则表达式	2
4.1. 正则表达式	2
4.1.1. 什么是正则表达式	2
4.1.2. 正则表达式 metacharacters	2
4.1.3. Basic versus 扩展正则表达式	2

4.2. 使用 Grep 的例子	2
4.2.1. 什么是 Grep?.....	2
4.2.2. Grep 与正则表达式	2
4.3. 模式匹配使用 Bash 特性	2
4.3.1. 字符范围	2
4.3.2. 字符 classes	2
4.4. 总结	2
4.5. 练习	2
第五章 GNU SED 流编辑器	2
5.1. 介绍	2
5.1.1. 什么是 sed?.....	2
5.1.2. sed 命令	2
5.2. 交互编辑	2
5.2.1. 打印包含 pattern 的行	2
5.2.2. 删除包含 pattern 的输入行	2
5.2.3. 行的范围	2
5.2.4. 用 sed 查找替换	2
5.3. 非交互编辑	2
5.3.1. 从文件读取 sed 命令	2
5.3.2. 写输出文件	2
5.4. 总结	2
5.5. 练习	2
第六章 GNU AWK 编程语言	2
6.1. gawk 上路	2
6.1.1. 什么是 gawk?	2
6.1.2. Gawk 命令	2
6.2. 打印程序	2
6.2.1. 打印选择的域	2
6.2.2. 格式化块	2
6.2.3. 打印命令和正则表达式	2
6.2.4. 特殊的 pattern	2
6.2.5. Gawk 脚本	2
6.3. Gawk 变量	2
6.3.1. 输入块的分隔符	2
6.3.2. 输出的分隔符	2
6.3.3. 记录的数量	2
6.3.4. 用户定义变量	2
6.3.5. 更多例子	2
6.3.6. printf 程序	2
6.4. 总结	2
6.5. 练习	2
第七章 条件语句	2
7.1. 介绍 if	2
7.1.1. 概要	2

7.1.2.if 的简单应用	2
7.2.更多 if 的高级使用方法	2
7.2.1.if/then/else 结构	2
7.2.2. if/then/elif/else 结构	2
7.2.3.if 嵌套语句	2
7.2.4.布尔操作	2
7.2.5. 使用 exit 语句和 if	2
7.3.使用 case 语句	2
7.3.1.简单的条件	2
7.4.总结	2
7.5.练习	2
第八章 编写交互脚本	2
8.1.显示用户消息	2
8.1.1. 交互与否	2
8.1.2. 使用内建 echo 命令	2
8.2. 捕获用户输入	2
8.2.1. 使用内建 read 命令	2
8.2.2. 提示用户输入	2
8.2.3. 重定向和文件描述符	2
8.2.4. 文件输入和输出	2
8.3. 总结	2
8.4. 练习	2
第九章 重复性任务	2
9.1. for 循环	2
9.1.1. 如何工作	2
9.1.2. 例子	2
9.2. while 循环	2
9.2.1. What is it?.....108	2
9.2.2. 例子	2
9.3. until 循环	2
9.3.1. What is it?.....111	2
9.3.2. 例子	2
9.4. I/O 重定向和循环	2
9.4.1. 输入重定向	2
9.4.2. 输出重定向	2
9.5. Break 和 continue	2
9.5.1.内建 break	2
9.5.2.内建 continue.....	2
9.5.3. 例子	2
9.6. Making menus with the select built-in.....115	2
9.6.1. 概要.....115	2
9.6.2. 子菜单	2
9.7.内建 shift.....	2
9.7.1. What does it do?.....117	2

9.7.2. 例子	2
9.8. 总结	2
9.9. 练习	2
第十章 深入变量	2
10.1. 变量种类	2
10.1.1. 概要 assignment of values.....120	2
10.1.2. 使用内建 declare	2
10.1.3. 常量	2
10.2. 数组变量.....122	2
10.2.1. 创建数组	2
10.2.2. Dereferencing the 变量 in an array.....122	2
10.2.3. 删除数组变量	2
10.2.4. 数组例子	2
10.3. Operations on 变量	2
10.3.1. Arithmetic on 变量.....126	2
10.3.2. 变量的长度	2
10.3.3. 变量的转变	2
10.4. 总结	2
10.5. 练习	2
第十一章 函数	2
11.1. 介绍	2
11.1.1. 什么是函数?	2
11.1.2. 函数语法	2
11.1.3. 函数中参数的位置	2
11.1.4. 显示函数	2
11.2. 脚本中函数的例子	2
11.2.1. 循环利用	2
11.2.2. 设置路径	2
11.2.3. 远程备份	2
11.3. 总结	2
11.4. 练习	2
第十二章 捕捉信号	2
12.1. 信号	2
12.1.1. 介绍	2
12.1.2. kill 信号的使用	2
12.2. 陷阱	2
12.2.1. 概要	2
12.2.2. Bash 怎样解释陷阱	2
12.2.3. 更多例子	2
12.3. 总结	2
12.4. 练习	2
附录 A Shell 特性	2
A.1. 常用特性	2
附录 B GNU 自由文档守则	2

介绍

1. 本指南诞生原因

许多读者感到现有的 HOWTO 过于简短和不完整，同时那本 ABS 对于参考来说又过于庞大是撰写本文的主要原因，没有出现在介于两个极端之间教材。我写本指南的主要由于缺乏免费的基本课程，尽管它们非常应该存在。

这是一本实用的指南，并不十分的严肃，尝试用实际的东西来代替那些理论的例子。我分部分来写因为我对那些知道自己在谈论什么人写的脱离实际的和过分单纯的例子并不感到兴奋，展示一些比较酷的 bash 的特性，但是离开上下文环境后你无法在实际环境中使用。你可以完成本书后再回过头去阅读那些能帮助你在现实世界生存的例子和练习。

以我自己是一名 UNIX/Linux 用户，系统管理员和培训人员的经验来说，我知道人们可以又几年中天天与他们的系统打交道，而不用知道任务自动控制的那些细小知识。因此他们经常认为 UNIX 并不友好，更坏的是，他们产生了 UNIX 速度缓慢而又老掉牙的想法。这个成了另外一个本指南想加以纠正的问题。

2. 谁应该阅读此书

每个在 UNIX 或者类 UNIX 系统上工作且想使生活变得简单的人以及系统管理员，能从阅读本书获益。已经掌握通过命令行使系统运转的读者将会学到通过了解 shell 脚本使日常任务的执行变得安逸。系统管理员依靠大量的 shell 脚本，日常任务通常使用一些简单脚本自动运行。本文档充满例子将会鼓励你写出你自己的脚本而且鼓励你改进现有的脚本。

先决条件，不包含在本教程

你应该是一个有经验的 UNIX 或者 Linux 用户，熟悉基本命令和 man 帮助文档
能够使用文本编辑器
理解系统启动和关闭进程，初始化和初始化脚本
能建立用户和组，以及密码
拥有相应权限，能进入不同模式
理解设备，分区，挂载/卸载文件系统的名字约定
在系统中安装/删除软件

3. 新版本和取得方法

最新版本可以在 <http://tille.xalasy.com/training/bash> 找到，你也可以在找到相同的版本 <http://tldp.org/LDP/Bash-Beginners-Guide/html/index.html>

Fultus.com 已经有印刷版本。

图 1.封面



4. 历史修订

Revision 1.7 2005-09-05 Revised by: MG

Corrected typos in chapter 3, 6 and 7, incorporated user remarks, added a note in chap7.

Revision 1.6 2005-03-01 Revised by: MG

Minor debugging, added more keywords, info about new Bash 3.0, took out blank image.

Revision 1.5 2004-12-06 Revised by: MG

Changes because of new domain, minor corrections.

Revision 1.4 2004-10-18 Revised by: MG

Debugging, added a couple of notes in chap9, replaced screenshots with screen sections.

Corrected some

typos.

Revision 1.3 2004-07-09 Revised by: MG

Added tracer image 1x1 pixel <http://tulle.xalasys.com/images/blank-bash.png>, added textobjects for all

pictures, fixed wrong links in index, made signal list more clear.

Revision 1.2 2004-06-15 Revised by: MG

Added index, more markup in screen sections.

Revision 1.1 2004-05-22 Revised by: MG

Last read-through before going to press, added more examples, checked summaries, exercises, cleaned up

introduction.

Revision 1.0 2004-04-27 Revised by: TM

Initial release for LDP; more exercises, more markup, less errors and abuse; added glossary.

Revision 1.0-beta 2003-04-20 Revised by: MG

Pre-release

5. 贡献

感谢所有帮助（或者尝试帮助）过我的所有朋友，还有我的丈夫；你的鼓励让这个艰难工作变成

了可能。感谢所有提交错误报告，例子，注释的人们。

- Hans Bol, one of the groupies
- Mike Sim, remarks on style
- Dan Richter, for array examples
- Gerg Ferguson, for ideas on the title
- Mendel Leo Cooper, for making room
- #linux.be, for keeping my feet on the ground
- Frank Wang, for his detailed remarks on all the things I did wrong ;-)

特别感谢对本书作完整查阅和拼写语法错的 Tabatha Marshall。我们创造了一个伟大的团队，我睡觉的时候她工作。当然了，反之亦然。

6. 反馈

任何信息，错误等请写 mail 到 <tille wants no spam _at_ xalasy dot com>

7. 版权信息

8. 你需要什么？

9. 文档约定

10. 本文档结构

第一章： Bash 基础

第二章： 脚本基础

第三章： Bash 环境

第四章： 正则表达式

第五章： Sed

第六章： Awk

第七章： 条件语句

第八章： 交互脚本

第九章： 重复执行命令

第一章 Bash 和 Bash 脚本

在这个介绍章节中，我们

讨论一些常用的 Shell

指出 GNU Bash 的优势和特性

描述 shell 的块建立

讨论 Bash 初始化文件

观察 Shell 怎么执行命令

察看一些简单的脚本

1.1. 普通 shell 程序

1.1.1. shell 的作用

UNIX 的 shell 程序解释用户的命令，不管是用户直接输入的或者从一个称作 Shell 脚本或者 Shell 程序文件读入。Shell 脚本是解释型的，而不是编译型的。Shell 从脚本行每行读取命令并在系统中搜索这些命令(see Section 1.2)，当编译器把一个程序转化为可供机器读取的可执行文件形式时，那么它就可以被用在 shell 脚本当中。

除了向内核传输命令之外，shell 的主要任务是提供一个可单独配置的使用 shell 资源配置文件的用户环境。

1.1.2. shell 类型

就像人们知道有不同的语言和方言一样，你的 UNIX 系统通常提供多种 shell 类型：

sh 或称 Bourne Shell：最早的 shell 并且仍然在 UNIX 系统和 UNIX 相关系统中使用。它是基本的 shell，是一个特性不多的小程序。虽然不是一个标准的 shell，但是在每个和 UNIX 程序兼容的 Linux 系统上仍然存在。

bash 或称 Bourne Again shell：标准的 GNU shell，直观而又灵活。或许是初学者的最明智选择同时对高级和专业用户来说也是一个强有力的工具。在 Linux 上，**bash** 是普通用户的标准 shell。这个 shell 因此称为 Bourne shell 的超集，一套附件和插件。意味着 **bash** 和 **sh** 是兼容的：在 **sh** 中可以工作的命令，在 **bash** 中也能工作，反之则不然。本书所有的例子均使用 **Bash**

csh 或称 C shell：语法了类似于 C 语言，某些时候程序员会使用

tcsh 或称 Turbo C shell： a superset of the common 普通 C shell 的一个超集，加强了的用户友好度和速度。

ksh 或称 the Korn shell：某些时候被有 UNIX 背景的人所赏识。Bourne shell 的一个超集，标准配置对初学者来说就是一场恶梦。

文件 `/etc/shells` 给出了 Linux 系统上所有的已知 shell

```
mia:~> cat /etc/shells
/bin/bash
/bin/sh
```

```
/bin/tcsh
```

```
/bin/csh
```

你默认的 shell 设置在文件 `/etc/passwd` 中，象下面这行对用户 mia 的设置

```
mia:L2NOFqdlPrHwE:504:504:Mia Maya:/home/mia:/bin/bash
```

要从一个 shell 转换到另外一个，只要在活动的终端里输入新 shell 的名字。系统在 PATH 设置的目录里面寻找，既然一个 shell 就是一个可执行的文件，当前的 shell 激活它使它运行起来，新的提示符出现，因为每个 shell 都有自己典型的外观：

```
mia:~> tcsh
```

```
[mia@post21 ~]$
```

1.2. Bash 的优势

1.2.1. Bash 是 GNU Shell

GNU 计划(GNU's Not UNIX)提供工具给类 UNIX 系统，象遵守 UNIX 标准的免费系统管理软件。

Bash 是兼容 sh 的 shell 且从 Korn shell (ksh) and C shell (csh)整合了一些有用的特性。它遵循 IEEE POSIX P1003.2/ISO 9945.2 Shell 和工具标准。提供了基于 sh 的编程和交互的功能改进。其中包括命令行编辑，无限制的历史命令，作业控制，shell 函数和别名，无大小现实的索引数组，和以 2 到 64 为基础的整数算法。Bash 可以无修改地运行多数 sh 脚本。

和其他的 GNU 项目一样，bash 主动开始保留，保护和促进使用，学习，拷贝，修改和再发布软件的自由。普遍认为这样的情况激励了创造力。这也是 bash 程序可以而许多其他 shell 无法提供的额外特性。

1.2.2. Bash 独有的特性

1.2.2.1. 改进

除了单字符 shell 命令行选项可以通常使用内建命令 `set` 来配置外，你还可以使用几种多字符选项。我们会在本章和后继章节中一系列的更流行的选项中留下深刻印象；完整的列表可以在 Bash 的信息页面中找到，Bash 特性->调用 Bash。

1.2.2.2. Bash 启动文件

启动文件是当 Bash 启动时候读取并且执行的脚本。下面的部分描述了启动 shell，和读取启动文件的不同途径。

1.2.2.2.1. 以交互登陆 shell 调用，或者使用 `--login`

交互意味着你可以输入命令。Shell 没有运行因为一个脚本被激活了。一个登陆 shell 就是在系统验证完了你输入的用户名和密码后得到 shell。

读取的文件：

```
/etc/profile
```

```
~/.bash_profile, ~/.bash_login 或者 ~/.profile: 读取第一个存在的可读的文件
```

`~/ .bash_logout` 退出登陆

错误消息将会显示如果配置文件存在但是不能读取。一个文件不存在，Bash 将搜索下一个。

1.2.2.2.2. 以一个交互非登陆 shell 调用

一个非登陆 shell 就是不需要进行系统的认证。比如，通过一个图标打开一个终端，或者一个菜单项目，那样就是非登陆 shell。

读取的文件：

`~/ .bashrc`

此文件通常指向 `~/ .bash_profile`：

```
if [ -f ~/ .bashrc ]; then . ~/ .bashrc; fi
```

参阅第七章获取 if 结构的详细信息

1.2.2.2.3. 非交互调用

所有脚本使用非交互 shell。他们是被编制出来完成特定任务且不能完成其他工作。

读取的文件：

由环境变量 `BASH_ENV` 定义

搜索此文件时候不使用环境变量 `PATH`，所以若是你想使用它，最好给出全部路径名和文件名来。

1.2.2.2.4. 以 sh 命令调用

Bash 尝试 sh 的相似行为同时也遵循 POSIX 标准。

读取的文件：

`/etc/profile`

`~/ .profile`

当以交互方式调用时，环境变量 `ENV` 能指出额外的启动信息。

1.2.2.2.5. POSIX 模式

本选项在使用内建的 `set` 命令：

```
set -o posix
```

或者再以 `-posix` 选项调用 `bash` 程序时候被激活。Bash 会试着尝试尽可能遵循 POSIX 的 shell 标准。同样可以设置 `POSIXLY_CORRECT` 变量来达到目的。

读取的文件：

在环境变量 `ENV` 中定义。

1.2.2.2.6. 远程调用

调用 rshd 读取的文件:

~/bashrc

禁止使用 r 系列工具



要注意使用类似 rlogin,telnet,rsh,rcp 工具的危险。由于他们在网络上传输数据是未经过加密的所以他们本质上是不安全的。如果你需要远程执行和文件传输之类的工具,推荐使用 SSH,从 <http://www.openssh.org> 下载,不同的客户端程序已经出现非 UNIX 系统上,请察看本地的软件镜像。

1.2.2.2.7. 当 UID (用户 ID) 不等于 EUID 时候调用

此种方式无起始文件读取。

1.2.2.3. 交互 shell

1.2.2.3.1. 什么是交互 shell

交互 shell 通常在用户终端读取并且输入: 输入输出都连接到终端。Bash 启动交互行为当 bash 以无选项方式调用,除了选项以字符串读取或者 shell

1.2.2.3.2. 这个 shell 是交互的吗?

1.3. 执行命令

1.3.1. 概要

Bash 会确定执行的程序的类型。普通程序是那些已经为你的系统编译好的系统命令。当这样的程序

运行的时候, Bash 创建一个自身的拷贝因此一个新的进程就被创建。子进程和父进程拥有一样的环境,唯一区别只有进程号。这个过程被称作内核空间创建进程 (Forking)。

在内核空间创建进程后,子进程的地址空间被新进程数据覆盖。此步骤通过 exec 系统调用完成。

fork-and-exec 机制因此把旧的命令转化成新的。这种机制用来创建所有的 UNIX 进程,因此对 Linux 操作系统也起作用,甚至是第一个进程,进程号是 1 的 init,也是在称为 bootstrapping 的启动过程中被 fork 的。

1.3.2. Shell 内部命令

内建的命令包含在 shell 本身里面。当内建的命令的名字被用作一个简单命令的第一个词时, shell 直接执行那个命令, 而不创建新的进程。

内建的命令是实现

bash 支持 3 种内建的命令:

1.3.3. 从脚本执行程序

当程序被一个脚本来执行, bash 会使用 fork 创建一个新的 bash 进程。这个子 shell 一次从 shell 脚本中读取一行。每行的命令如果直接来自键盘的话会被读取, 翻译和执行

当子 shell 处理脚本中的每一行时, 父进程等待子进程结束。当运行完 shell 脚本中每一行时, 子 shell 就结束。父 shell 苏醒并且显示一个新的提示符。

1.4. 建立块

1.4.1. Shell 建立块

1.4.1.1. Shell 语法

1.4.1.2. Shell 命令

一个简单的 shell 命令例如 `touch file1 file2 file3` 由命令本身, 参数以及空格构成。

更复杂的 shell 命令由简单的命令以多种方式组织在一起: 例如管道把一个命令的输出变成另外一个命令的输入, 循环或者条件结构, 或者其他的组织方式。请看一些例子:

```
ls | more
```

```
gunzip file.tar.gz | tar xvf -
```

1.4.1.3. Shell 函数

Shell 函数是一种为一组之后执行的命令命名一个名字。他们执行起来就像“普通”命令。当一个 Shell 函数的名字被一个简单命令使用后, 和函数名字相关联的命令就会执行。

Shell 函数会在当前 shell 下执行, 不会有新的进程创建来打断它们。

1.4.1.4. Shell 参数

参数是储存值得实体。可以一个名字, 一个数字或者一个特别的值。从 shell 的目的来看, 变量是一个存储名字的参数。一个变量拥有一个值, 零或者更多的属性。变量是用 shell 内建命令 `declare` 来建立的。

如果没有赋予任何值，变量就被赋予空字符串。变量只能使用内建命令 `unset` 来移除。在 3.2 节将讨论对变量赋值，第十章将会讨论高级使用方法。

1.4.1.5. Shell 表达式

Shell 表达式是在每一行分成几部分后执行的。以下是表达式的类型

大括号表达式

波浪号表达式

参数和变量表达式

命令替换表达式

算术表达式

字分离表达式

文件名表达式

在 3.4 节我们将详细讨论这些表达式

1.4.1.6. 重定向

在一个命令执行之前，它可能借助 `shell` 翻译的一个特殊的符号重定向它的输入输出。重定向也可以用来为现有的 `shell` 执行环境打开和关闭文件。

1.4.1.7. 执行命令

当执行一个命令的时候，

1.4.1.8. Shell 脚本

调用 `bash` 时，当一个包含 `shell` 命令的文件被用作第一个非选项参数时候（不带 `-c` 或者 `-s`，这两个参数将会创建一个非交互的 `shell`）。这个 `shell` 首先搜寻脚本文件所在的当前目录，如果没有找到则对环境变量 `PATH` 进行查找。

1.5. 开发优良脚本

1.5.1. 优良脚本的要素

本指南只要是关于上面的 `shell` 构成部分，脚本。在继续之前我们应该考虑一些东西：

1. 一个脚本应该无错运行

2. 它应该完成他要完成的任务
3. 程序的逻辑结构定义清晰而且明显
4. 一个脚本不做不必要的工作
5. 脚本可以重用

1.5.2. 结构

1.5.3. 术语

1.5.4. 有序且有逻辑性

1.5.5. 一个 Bash 脚本的例子: `mssystem.sh`

1.5.6. 初始化脚本例子

1.6. 总结

Bash 是 GNU shell, 兼容 sh 以及其他 shell 里的许多有用的特性。当 shell 启动的时候, 它读取它自己的配置文件。最重要的几个如下所示:

`/etc/profile`

`~/.bash_profile`

`~/.bashrc`

Bash 在交互模式下, 遵循 POSIX 标准的时候和限制模式下的行为会有所不同。shell 命令可以分为 3 个种类: shell 的功能命令, shell 内建命令和你系统里某个目录里的命令。Bash 支持额外的 sh 不包含的内建命令。Shell 脚本把那些命令组成 shell 语法命令。脚本读取一行执行一行且应该有逻辑结构。

1.7. 练习

第二章 编写和调试脚本

经过学习本章节，你将能够：

- 写简单的脚本
- 定义 shell 类型来运行脚本
- 在脚本中加入注释
- 改变脚本的权限
- 执行并且调试脚本

2.1. 建立并且运行一个脚本

2.1.1. 编写与命名

一个 shell 脚本是你重复使用的命令序列。这个序列通常是通过在命令行输入整个脚本的命令来执行的。或者，你可以使用 cron 工具来使脚本自动执行任务。另外脚本也可以用于 UNIX 的启动和关闭过程中定义在 init 脚本中的守护程序和服务的操作。

要建立一个 shell 脚本，在你的编辑器中打开一个新的空白文件。任何文本编辑器就可以：vim, emacs, gedit, dtpad 都可以。可能你会选择一个高级的编辑器比如 vim 或者 emacs，因为它们可以设置来高亮语法来帮助初学者避免很多经常犯的错误，比如忘记括号和冒号。

把 UNIX 命令放在一个新的空白文件里，就象在命令行输入命令。就像先前的讨论（参阅 1.3 节），命令可以作为 shell 函数，shell 组成部分，UNIX 命令和其他脚本。

为你的脚本起一个合理的名字来提醒自己脚本的作用。确定你的脚本的名字和已经存在的命令是没有冲突的。为了保证不会引起混乱，脚本名字经常以.sh 结尾。

尽管如此，还是有可能你选择的脚本名字已经和系统中原来的是相同的。使用 which, whereis 和一些其他命令来查找程序的文件的信息：

```
which -a script_name
```

```
whereis script_name
```

```
locate script_name
```

2.1.2. Script1.sh

在本例子中我们使用 bash 内建命令 echo 在此任务运行建立输出之前来通知用户将发生什么。强烈建议告知用户脚本的用途，因为本脚本什么事都不干防止用户变得过于紧张。我们会在第八章继续如何通知用户。

```

#!/bin/bash

clear

echo "The script starts now."

echo "Hi, $USER!"
echo

echo "I will now fetch you a list of connected users:"
echo
w
echo

echo "I'm setting two variables now."
COLOUR="black"
VALUE="9"
echo "This is a string: $COLOUR"
echo "And this is a number: $VALUE"
echo

echo "I'm giving you back your prompt now."
echo
~
~
~

```

正好自己写一下脚本。建立一个目录 `~/scripts` 来存放你的脚本将会是个好主意。把此目录添加到 `PATH` 变量中：

```
export PATH="$PATH:~/scripts"
```

如果你才开始使用 `Bash`，最好使用一个给不同结构以不同颜色的文本编辑器。`vim`，`gvim`，`(x)emacs`，`kwwrite` 和许多其他支持语法高亮的编辑器；详细请察看编辑器的文档。

不同的提示符

本教程的提示符的改变都依赖作者的心情。相对于标准的 `$` 提示符他更接近之实际情况。唯一的约定就是，`root` 提示符以 `#` 结尾。

2.1.3. 执行脚本

为了能使当前用户运行脚本，它应该有可执行权限。在设置权限时，检查你是否得到你想要得权限。完成后脚本就可以象其他命令一样运行。

```

willy:~/scripts> chmod u+x script1.sh
willy:~/scripts> ls -l script1.sh
-rwxrw-r-- 1 willy willy 456 Dec 24 17:11 script1.sh
willy:~> script1.sh

```

```
The script starts now.
Hi, willy!
I will now fetch you a list of connected users:
3:38pm up 18 days, 5:37, 4 users, load average: 0.12, 0.22, 0.15
USER TTY FROM LOGIN@ IDLE JCPU PCPU WHAT
root tty2 - Sat 2pm 4:25m 0.24s 0.05s -bash
willy :0 - Sat 2pm ? 0.00s ? -
willy pts/3 - Sat 2pm 3:33m 36.39s 36.39s BitchX willy ir
willy pts/2 - Sat 2pm 3:33m 0.13s 0.06s /usr/bin/screen
I'm setting two variables now.
This is a string: black
And this is a number: 9
I'm giving you back your prompt now.
willy:~/scripts> echo $COLOUR
willy:~/scripts> echo $VALUE
willy:~/scripts>
```

这是执行脚本最普通的方法。在子 shell 中一般都是这么执行脚本。建立在子 shell 中的变量，函数和别名等只有子 shell 使用。当那个 shell 退出，父 shell 重新得到控制的时候，所有的东西都被清空，脚本对 shell 状态所作的改变全部清除。

如果你没有把脚本的目录放到 PATH 里面，当前目录也不在 PATH 变量中，那么你可以这样来执行脚本：

```
./script_name.sh
```

脚本也可以被明确地指定 shell 执行，不过通常我们只有在想得到特殊的行为时候才这样做，比如脚本是否能在另外一个 shell 里面工作或者打印调试的信息。

```
rbash script_name.sh
sh script_name.sh
bash script_name.sh
```

特定的 shell 会成为当前 shell 子 shell 来运行脚本。当你想以特殊的选项或者以脚本没有指定的特殊条件来启动脚本时候可以这么做。

如果你想在当前脚本执行脚本而不想启动一个新的 shell，你可以使用 source：

```
source script_name.sh
```

 source =.

Bash 的内建命令 source 和 sh 的命令是相同的。这里脚本不需要可执行权限。命令在当前 shell 力执行，所以任何对环境的改变，将在脚本结束时同样起作用。

```
willy:~/scripts> source script1.sh
```

```
--output omitted--
willy:~/scripts> echo $VALUE
9
willy:~/scripts>
```

2.2. 脚本基础

2.2.1. 哪个 Shell 来执行脚本

当在子 shell 运行脚本时，你应该定义哪个 shell 来运行脚本，你编写的脚本的 shell 类型可能不是你系统默认的，所以用错误的 shell 来运行你输入的命令可能最终出错。

第一行决定了启动的 shell，第一行的开始 2 个字符应该是 #!，然后紧跟解释后面命令的 shell 的路径。空白行也被认为是一行，所以不要让你的脚本以空白行开始。

出于本教程的考虑，所有的脚本都这样开头：

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

先前提到过，这样表明 bash 可以在 /bin 里面找到。

2.2.2. 加入注释

你应该知道事实上你不会阅读你自己脚本的唯一的一个人。很多用户和系统管理员运行别人编写的脚本。如果他们想知道你是如何做到的，注释能很好的提醒读者。

注释也同样让你自己更方便。你一定阅读了很多帮助页面通过脚本中的一些命令来得到特定的结果。如果不对脚本加上注释，几个星期或者几个月后你需要更改你的脚本，你会忘记脚本做了些什么事，你怎么做的和为什么要做。

把 script1.sh 例子拷贝到 commented-script1.sh，编辑注释来反映脚本的作用。在 # 后面的行被忽略而且只能在打开脚本时才能看到。

```
#!/bin/bash
# This script clears the terminal, displays a greeting and gives information
# about currently connected users. The two example variables are set and displayed.
clear # clear terminal window
echo "The script starts now."
echo "Hi, $USER!" # dollar sign is used to get content of variable
echo
echo "I will now fetch you a list of connected users:"
echo
w # show who is logged on and
echo # what they are doing
echo "I'm setting two variables now."
```

```
COLOUR="black" # set a local shell variable
VALUE="9" # set a local shell variable
echo "This is a string: $COLOUR" # display content of variable
echo "And this is a number: $VALUE" # display content of variable
echo
echo "I'm giving you back your prompt now."
echo
```

在一个良好的脚本中，第一行经常注明要完成的任务。然后为了明确每一大块命令将被加上注释。作为一个例子，Linux 的初始脚本，在你系统里的 `init.d` 目录下，为了能让每个使用 Linux 的人读取和更改它，通常都作了良好的注释。

2.3. 调试 Bash 脚本

2.3.1. 调试整个脚本

当一些事情不能按照计划进行，你需要确定到底是什么导致了脚本运行失败。Bash 提供了大量的调试特性。最通常的做法是使用 `-x` 选项来启动子 shell，这将让整个脚本在调试模式下进行。每个命令和他附加参数的信息会在执行之前被展开并且送到标准输出打印。

以下是脚本 `commented-script1.sh` 在调试模式下运行。再次注意在脚本的输出中注释是不可见的。

```
willy:~/scripts> bash -x script1.sh
+ clear
+ echo 'The script starts now.'
The script starts now.
+ echo 'Hi, willy!'
Hi, willy!
+ echo
+ echo 'I will now fetch you a list of connected users:'
I will now fetch you a list of connected users:
+ echo
+ w
4:50pm up 18 days, 6:49, 4 users, load average: 0.58, 0.62, 0.40
USER TTY FROM LOGIN@ IDLE JCPU PCPU WHAT
root tty2 - Sat 2pm 5:36m 0.24s 0.05s -bash
willy :0 - Sat 2pm ? 0.00s ? -
willy pts/3 - Sat 2pm 43:13 36.82s 36.82s BitchX willy ir
willy pts/2 - Sat 2pm 43:13 0.13s 0.06s /usr/bin/screen
+ echo
+ echo 'I\'m setting two variables now.'
I'm setting two variables now.
+ COLOUR=black
+ VALUE=9
+ echo 'This is a string: '
```

```

This is a string:
+ echo 'And this is a number: '
And this is a number:
+ echo
+ echo 'I'\''m giving you back your prompt now.'
I'm giving you back your prompt now.
+ echo
    
```



未来 bash 的特性

现在有一个全新的 Bash 调式工具出现在 SourceForge。然而现在，你需要一个已经打过补丁的 Bash-2.05。新的特性可能会在 bash-3.0 中出现。

2.3.2. 调试部分脚本

使用 bash 内建命令 `set` 可以让那些确定没有错误的部分以正常模式运行，而只对有错误的部分显示其 `debug` 信息。比如我们不确定在 `commented-script1.sh` 里面 `w` 命令会做些什么，那么我们可以把它象这样包含起来：

```

set -x # activate debugging from here
w
set +x # stop debugging from here
    
```

输出看来就像这样：

```

willy: ~/scripts> script1.sh
The script starts now.
Hi, willy!
I will now fetch you a list of connected users:
+ w
5:00pm up 18 days, 7:00, 4 users, load average: 0.79, 0.39, 0.33
USER TTY FROM LOGIN@ IDLE JCPU PCPU WHAT
root tty2 ? Sat 2pm 5:47m 0.24s 0.05s ?bash
willy :0 ? Sat 2pm ? 0.00s ? ?
willy pts/3 ? Sat 2pm 54:02 36.88s 36.88s BitchX willyke
willy pts/2 ? Sat 2pm 54:02 0.13s 0.06s /usr/bin/screen
+ set +x
I'm setting two variables now.
This is a string:
And this is a number:
I'm giving you back your prompt now.
willy: ~/scripts>
    
```

你可以在同样的脚本里多次打开关闭调试模式。

下表给出了其他有用的 bash 选项概貌

Table 2–1. Overview of set debugging options

Short notation	Long notation	Result
set -f	set -o noglob	Disable file name generation using metacharacters (globbing).
set -v	set -o verbose	Prints shell input lines as they are read.
set -x	set -o xtrace	Print command traces before executing command.

横线用来激活一个 shell 选项，再使用一次将解除。不要搞错了。

以下例子，我们在命令行里面证明这些选项。

```
willy:~/scripts> set -v
willy:~/scripts> ls
ls
commented?scripts.sh script1.sh
willy:~/scripts> set +v
set +v
willy:~/scripts> ls *
commented?scripts.sh script1.sh
willy:~/scripts> set -f
willy:~/scripts> ls *
ls: *: No such file or directory
willy:~/scripts> touch *
willy:~/scripts> ls
* commented?scripts.sh script1.sh
willy:~/scripts> rm *
willy:~/scripts> ls
commented?scripts.sh script1.sh
```

或者，这些模式也可以在脚本里面指定，只需在第一行 shell 的声明中加入需要的选项。选项可以叠加，和通常 UNIX 命令一样：

```
#!/bin/bash -xv
```

每当你找到脚本中的错误部分，你可以在每个你不确定的命令之前增加 echo 语句，这样你就会明确的看到哪里或者为什么脚本没有正常工作。在 commented-script1.sh 例子中，可以这么做，依然假设显示用户这部分出了问题：

```
echo "debug message: now attempting to start w command"; w
```

在更高级的脚本中，echo 可以插入到在不同的阶段显示变量表，因此可以检查到错误：

```
echo "Variable VARNAME is now set to $VARNAME."
```

2.4. 总结

shell 脚本是一个放在可执行文本文件中的一组可以重用的命令。任何文本编辑器都可以用来编

写脚本。

脚本以`#!`开头，后面紧跟执行脚本的 `shell` 的路径。在脚本中加入注释是为了将来的参考，同样也使得其他人能理解脚本。解释多肯定比少要好。

使用选项可以用来调试脚本。`shell` 选项可以用来部分或者扫描整个脚本。善用 `echo` 你某些位置也是一种常用的排错技术。

2.5. 练习

第三章 Bash 环境

本章我们讨论几种能影响 Bash 环境的不同方法

编辑 shell 初始化文件

使用变量

使用不同引用风格

实现算术计算

指派别名

使用扩充和替换

3.1. Shell 初始化文件

3.1.1. 跨系统配置文件

3.1.1.1. /etc/profile

当用 `--login` 选项或者以 `sh` 来调用交互模式时，Bash 读取 `/etc/profile` 的指令。通常是一些设置 shell 变量 `PATH`, `USER`, `MAIL`, `HOSTNAME` 和 `HISTSIZE`。

再某些系统上，`umask` 的值在 `/etc/profile` 中配置；其他的一些系统中这个文件包含了指向了其他配置文件的指针：

`/etc/inputrc`，可以配置命令行响铃风格的跨系统的行读取初始化文件。

`/etc/profile.d` 目录，包含了配置特别程序的跨系统行为的文件。

你想应用到所有用户环境的所有设置都必须在这个文件中，这个文件看上去像这样：

```
# /etc/profile
# System wide environment and startup programs, for login setup
PATH=$PATH:/usr/X11R6/bin
# No core files by default
ulimit -S -c 0 > /dev/null 2>&1
USER=`id -un`
LOGNAME=$USER
MAIL="/var/spool/mail/$USER"
HOSTNAME=`/bin/hostname`
HISTSIZE=1000
# Keyboard, bell, display style: the readline config file:
if [ -z "$INPUTRC" -a ! -f "$HOME/.inputrc" ]; then
INPUTRC=/etc/inputrc
fi
PS1="\u@\h \w"
export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE INPUTRC PS1
# Source initialization files for specific programs (ls, vim, less, ...)
for i in /etc/profile.d/*.sh ; do
```

```

if [ -r "$i" ]; then
. $i
fi
done

# Settings for program initialization
source /etc/java.conf
export NPX_PLUGIN_PATH="$JRE_HOME/plugin/ns4plugin/:/usr/lib/netscape/plugins"
PAGER="/usr/bin/less"
unset i

```

3.1.2. 单独用户配置文件

我没有这些文件?!

默认情况下这些文件可能在你的主目录中，需要的话也可以建立他们。

3.1.2.1. ~/.bash_profile

这是单独为用户配置环境的首选的配置文件。在这个文件中，用户可以增加额外的配置选项或者改变默认设置：

```

franky~> cat .bash_profile
#####
# #
# .bash_profile file #
# #
# Executed from the bash shell when you log in. #
# #
#####
source ~/.bashrc
source ~/.bash_login
case "$OS" in
IRIX)
stty sane dec
stty erase
;;
# SunOS)
# stty erase
# ;;
*)
stty sane
;;
esac

```

这个用户配置了登陆到不同操作系统的退格字符。除此之外，用户的 `.bashrc` 和 `.bash_login` 也被读取。

3.1.2.2. ~/.bash_login

这个文件包含了只有在你登陆进系统的才执行的特殊的设置。在这个例子中，我用它来配置 `umask` 的值来显示一个当前连接的用户列表。该用户也得到了当前月的日历。

```
#####
# #
# Bash_login file #
# #
# commands to perform from the bash shell at login time #
# (sourced from .bash_profile) #
# #
#####
# file protection
umask 002 # all to me, read to group and others
# miscellaneous
w
cal `date +%m` `date +%Y``
```

在没有 `~/.bash_profile` 的情况下，这个文件就被读取。

3.1.2.3. ~/.profile

在没有 `~/.bash_profile` 和 `~/.bash_login` 的情况下，`~/.profile` 就被读取。他能保存一些可以被别的 shell 访问的配置。注意其他的 shell 可能不能识别 Bash 的语法。

3.1.2.4. ~/.bashrc

如今，更加普遍的是使用一个非登陆 shell，比如使用 X 终端窗口登陆进图形模式的时候。打开一个这样的窗口之后，用户不需要提供用户名和密码；无需认证。此时 Bash 会搜索 `~/.bashrc`，所以也指向登陆时读取得文件，同时也意味着你不需要在多个文件中输入相同的设置。

在这个用户的 `.bashrc` 里，在读取了跨系统的 `/etc/bashrc` 之后定义了一些别名和为特定的程序使用的变量。

```
franky ~-> cat .bashrc
# /home/franky/.bashrc
# Source global definitions
if [ -f /etc/bashrc ]; then
. /etc/bashrc
fi
# shell options
set -o noclobber
# my shell variables
```

```

export PS1="\[\033[1;44m\]\u \w\[\033[0m\] "
export PATH="$PATH:~/bin:~/scripts"
# my aliases
alias cdrecord='cdrecord -dev 0,0,0 -speed=8'
alias ss='ssh octarine'
alias ll='ls -la'
# mozilla fix
MOZILLA_FIVE_HOME=/usr/lib/mozilla
LD_LIBRARY_PATH=/usr/lib/mozilla:/usr/lib/mozilla/plugins
MOZ_DIST_BIN=/usr/lib/mozilla
MOZ_PROGRAM=/usr/lib/mozilla/mozilla-bin
export MOZILLA_FIVE_HOME LD_LIBRARY_PATH MOZ_DIST_BIN MOZ_PROGRAM
# font fix
alias xt='xterm -bg black -fg white &'
# BitchX settings
export IRCNAME="frnk"
# THE END
franky ~>

```

更多的例子能在 `bash` 的包内找到。记住例子可能需要修改才能在你的环境内工作。

别名将在 3.5.节讨论。

3.1.2.5. ~/.bash_logout

这个文件包含了登出系统时候的特别指令。在这个例子中，终端窗口在登出的时候被清空。在关闭时留下一个干净的窗口对远程连接来说非常有用。

```

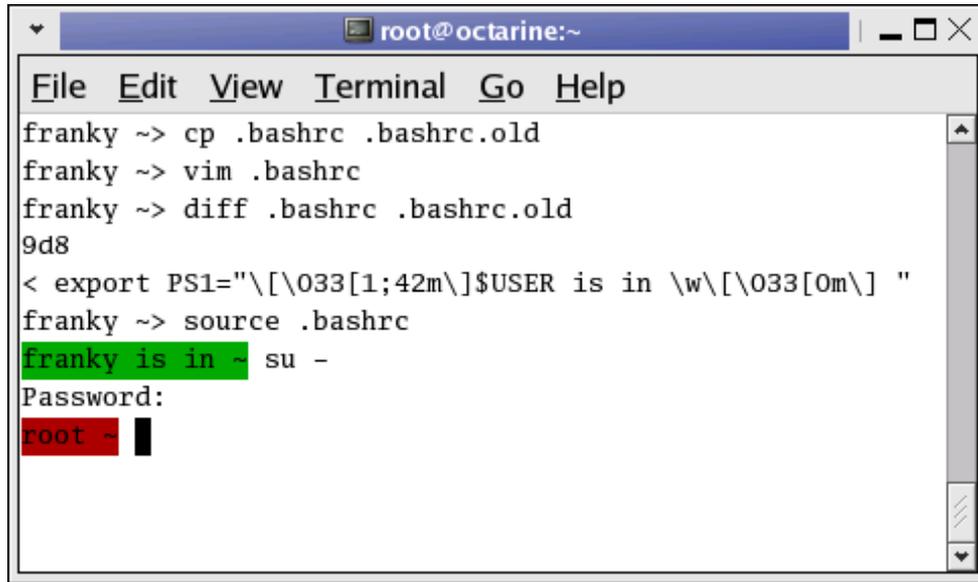
franky ~> cat .bash_logout
#####
# #
# Bash_logout file #
# #
# commands to perform from the bash shell at logout time #
# #
#####
clear
franky ~>

```

3.1.3. 改变 shell 配置文件

当对上述文件进行任何修改的时候，用户可以重新连接到系统，或者 `source` 这个改变的文件来生效。这样解释脚本的话，修改就应用到现在的 shell。

图 3-1.不同用户的不同提示



多数 shell 脚本在一个私有环境中执行：除非变量是父脚本 `export` 出来的，不然他们不会被子进程继承下来。

3.2. 变量

3.2.1. 变量的类型

正如上面的例子，shell 变量约定俗成地用大写表示。Bash 保留两种类型的变量列表：

3.2.1.1. 全局变量

全局变量或者环境变量存在于所有的 shell 里面。`env` 和 `printenv` 命令能够通常用于显示环境变量。这些程序在 `sh-utils` 包内。

下面是一个典型的输出：

```
franky ~> printenv
CC=gcc
CDPATH=.:~/usr/local:/usr:/
CFLAGS=-O2 -fomit-frame-pointer
COLORTERM=gnome-terminal
CXXFLAGS=-O2 -fomit-frame-pointer
DISPLAY=:0
DOMAIN=hq.xalasys.com
e=
TOR=vi
FCEDIT=vi
```

```

FIGIGNORE=.o:~
G_BROKEN_FILENAMES=1
GDK_USE_XFT=1
GDMSESSION=Default
GNOME_DESKTOP_SESSION_ID=Default
GTK_RC_FILES=/etc/gtk/gtkrc:/nethome/franky/.gtkrc-1.2-gnome2
GWMCOLOR=darkgreen
GWMTERM=xterm
HISTFILESIZE=5000
history_control=ignoredups
HISTSIZ=2000
HOME=/nethome/franky
HOSTNAME=octarine.hq.xalays.com
INPUTRC=/etc/inputrc
IRCNAME=franky
JAVA_HOME=/usr/java/j2sdk1.4.0
LANG=en_US
LDFLAGS=-s
LD_LIBRARY_PATH=/usr/lib/mozilla:/usr/lib/mozilla/plugins
LESSCHARSET=latin1
LESS=-edfMQ
LESSOPEN=|/usr/bin/lesspipe.sh %s
LEX=flex
LOCAL_MACHINE=octarine
LOGNAME=franky
LS_COLORS=no=00:fi=00:di=01;34:ln=01;36:pi=40;33:so=01;35:bd=40;33;01:cd=40;33;01:or=01;05;37;41:MACHINES=octarine
MAILCHECK=60
MAIL=/var/mail/franky
MANPATH=/usr/man:/usr/share/man:/usr/local/man:/usr/X11R6/man
MEAN_MACHINES=octarine
MOZ_DIST_BIN=/usr/lib/mozilla
MOZILLA_FIVE_HOME=/usr/lib/mozilla
MOZ_PROGRAM=/usr/lib/mozilla/mozilla-bin
MTOOLS_FAT_COMPATIBILITY=1
MYMALLOC=0
NNTPPORT=119
NNTPSERVER=news
NPX_PLUGIN_PATH=/plugin/ns4plugin:/usr/lib/netscape/plugins
OLDPWD=/nethome/franky
OS=Linux
PAGER=less
PATH=/nethome/franky/bin.Linux:/nethome/franky/bin:/usr/local/bin:/usr/local/sbin:/usr/X11R6/bin:/PS1=\[\033[1;44m\]franky is in \w\[\033[0m\]

```

```

PS2=More input>
PWD=/nethome/franky
SESSION_MANAGER=local/octarine.hq.xalaysys.com:/tmp/.ICE-unix/22106
SHELL=/bin/bash
SHELL_LOGIN=--login
SHLVL=2
SSH_AGENT_PID=22161
SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
SSH_AUTH_SOCK=/tmp/ssh-XXmhQ4fC/agent.22106
START_WM=twm
TERM=xterm
TYPE=type
USERNAME=franky
USER=franky
_=/usr/bin/printenv
VISUAL=vi
WINDOWID=20971661
XAPPLRESDIR=/nethome/franky/app-defaults
XAUTHORITY=/nethome/franky/.Xauthority
XENVIRONMENT=/nethome/franky/.Xdefaults
XFILESEARCHPATH=/usr/X11R6/lib/X11/%L/%T/%N%C%S:/usr/X11R6/lib/X11/%l/%T/%N%C%S:/usr/X11R6/lib/X11/%XKEYSYMDB=/usr/X11R6/lib/X11/XKeysymDB
XMODIFIERS=@im=none
XTERMINID=
XWINHOME=/usr/X11R6
X=X11R6
YACC=bison -y

```

3.2.1.2.本地变量

本地变量只存在于当前 **shell**。使用内建的不带选项的 **set** 命令将显示所有变量的列表（包括环境变量）和函数。输出会根据当前的设置排列而且以可以重用的方式显示。

以下是在退出了同样被 **set** 命令显示的函数之后，比较 **printenv** 和 **set** 的输出的文件。

```

franky ~> diff set.sorted printenv.sorted | grep "<" | awk '{ print $2 }'
BASE=/nethome/franky/.Shell/hq.xalaysys.com/octarine.aliases
BASH=/bin/bash
BASH_VERSINFO=([0]="2"
BASH_VERSION='2.05b.0(1)-release'
COLUMNS=80
DIRSTACK=()
DO_FORTUNE=
EUID=504
GROUPS=()

```

```

HERE=/home/franky
HISTFILE=/nethome/franky/.bash_history
HOSTTYPE=i686
IFS=$'
'
LINES=24
MACHTYPE=i686-pc-linux-gnu
OPTERR=1
OPTIND=1
OSTYPE=linux-gnu
PIPESTATUS=( [0]="0" )
PPID=10099
PS4='+
'
PWD_REAL='pwd
'
SHELLOPTS=braceexpand:emacs:hashall:histexpand:history:interactive-comments:monitor
THERE=/home/franky
UID=504

```

awk

GNU awk 变成语言将会在第六章讲述

3.2.1.3.按内容分变量

除了把变量分为本地和全局变量，我们也可以根据变量所含内容的类型来分。这样，变量就有四种类型：

字符串变量

整数变量

常量

数组变量

我们将在第十章来讨论这些类型，我们会用整数和字符串来操纵变量。

3.2.2. 建立变量

默认情况下，变量是区分大小写的。某些时候给于一个本地变量小写字母是约定俗成的。然而，你可以自由的使用这些名字或者混合大小写。变量也可以包含数字，但是数字开头的名字是不允许的：

```

prompt> export lnumber=1
bash: export: `lnumber=1': not a valid identifier

```

在 shell 中设置一个变量，使用：

```
VARNAME="value"
```

在等号周围放置空格会造成错误。在对变量赋值的时候把内容字符串用引号引起来是一个良好的习惯：这样会降低你出错的机会。

一些实用大小写，数字和空格的例子：

```
franky ~> MYVAR1="2"
franky ~> echo $MYVAR1
2
franky ~> first_name="Franky"
franky ~> echo $first_name
Franky
franky ~> full_name="Franky M. Singh"
franky ~> echo $full_name
Franky M. Singh
franky ~> MYVAR-2="2"
bash: MYVAR-2=2: command not found
franky ~> MYVAR1 ="2"
bash: MYVAR1: command not found
franky ~> MYVAR1= "2"
bash: 2: command not found
franky ~> unset MYVAR1 first_name full_name
franky ~> echo $MYVAR1 $first_name $full_name
<--no output-->
franky ~>
```

3.2.3. 导出变量

一个变量的建立就像上面的例子那样仅仅存在于当前 shell。他是本地变量：当前 shell 的子进程不会意识到这个的存在。为了把变量传递到子 shell，我们需要使用内建的 `export` 命令把他们 `export` 出来。被 `export` 出来的变量就像环境变量一样，设置和 `export` 变量通常用下面一步来完成：

```
export VARNAME="value"
```

一个子 shell 能够改变从父 shell 变量继承过来的变量，但是在子 shell 所作的改变对父 shell 也没有影响。下面的例子来证明这个：

```
franky ~> full_name="Franky M. Singh"
franky ~> bash
franky ~> echo $full_name
franky ~> exit
franky ~> export full_name
franky ~> bash
franky ~> echo $full_name
```

```
Franky M. Singh
franky ~-> export full_name="Charles the Great"
franky ~-> echo $full_name
Charles the Great
franky ~-> exit
franky ~-> echo $full_name
Franky M. Singh
franky ~->
```

当第一次尝试在子 shell 里面读取 `full_name` 的值时，它并不存在（`echo` 显示了一个空字符串）。子 shell 退出，然后 `full_name` 在父 shell 里面 `export`，一个变量在赋值后仍然可以被 `export`。然后一个新的子 shell 开始运行，从父 shell 那里 `export` 出来的变量是可见的。这个变量被修改来存放其他名字，但是在父 shell 中放置变量的值还是一样的。

3.2.4. 保留变量

3.2.5. 特殊参数

3.2.6. 脚本通过变量循环

除了使脚本变得更加易读，变量也会使你更加快速的在另外一个环境中应用一个脚本或者其他的。考虑下面的例子，一个非常简单的脚本把 `franky` 的主目录备份到远程服务器上。

首先，如果每次需要的时候你都手动命名文件和目录很可能造成错误。另外，假设 `franky` 向把这个脚本给 `carol`，在 `carol` 能使用脚本备份她的主目录之前需要做一些编辑。同样如果 `franky` 想使用这个脚本来备份其他目录。为了简单的再循环，使所有的文件，目录，用户名，服务器名字等变量。因此，你只需要编辑一次值，而不需要在整个脚本中检查哪里需要出现参数。以下是一个例子：

oooo

大目录和低带宽

以上是个大家都理解了的例子，使用一个小目录和一个在相同子网的主机。根据你的带宽，目录的大小和远程服务器的，使用这种机制来备份可能会花费很多时间。对更大的目录和更低的带宽，使用 `rsync` 来保证目录之间保持同步。

3.3. 引用字符

3.3.1. 为何？。。

3.3.2. 转义字符

3.3.3. 单引用

3.3.4. 双引用

3.3.5. ANSI-C 引用

3.3.6. Locales.....46

3.4. Shell 扩展

3.4.1. 概要

3.4.2. 大括号表达式

3.4.3. 波浪表达式

3.4.4. Shell 参数和变量扩展

3.4.5. 命令替换

3.4.6. 算术表达式

3.4.7. 替换的处理

3.4.8. Word splitting

3.4.9. 文件名扩展

3.5. 别名

3.5.1. 什么是别名

3.5.2. 建立和消除别名

3.6. 更多 Bash 选项

3.6.1. 显示选项

3.6.2. 改变选项

3.7. 总结

3.8. 练习

第四章 正则表达式

本章我们讨论:

使用正则表达式

正则表达式统配符

文件和输出中的找寻模板

Bash 中字符的范围和分类

4.1. 正则表达式

4.1.1. 什么是正则表达式

4.1.2. 正则表达式 metacharacters

4.1.3. Basic versus 扩展正则表达式

4.2. 使用 Grep 的例子

4.2.1. 什么是 Grep?

`grep` 以行为单位搜索那些包含给出模板列表的输入文件。当在一行中找到匹配，默认把该行拷贝到默认输出，或者其他你以选项要求的任何种类的输出。

虽然 `grep` 力图做到在文字上的匹配，不考虑内存因素的话它对输入行的长度没有限制，而且它能匹配一行中的任何字符。如果输入文件的最后一个字节不是换行符的话，`grep` 会自动加上一个。既然换行符也是模板列表的分隔符，那样就没有办法来从字面上匹配换行符。

一些例子:

```
cathy ~> grep root /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin

cathy ~> grep -n root /etc/passwd
1:root:x:0:0:root:/root:/bin/bash
12:operator:x:11:0:operator:/root:/sbin/nologin

cathy ~> grep -v bash /etc/passwd | grep -v nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
news:x:9:13:news:/var/spool/news:
```

```
mailnull:x:47:47::/var/spool/mqueue:/dev/null
xfs:x:43:43:X Font Server:/etc/X11/fs:/bin/false
rpc:x:32:32:Portmapper RPC user:/:/bin/false
nscd:x:28:28:NSCD Daemon:/:/bin/false
named:x:25:25:Named:/var/named:/bin/false
squid:x:23:23::/var/spool/squid:/dev/null
ldap:x:55:55:LDAP User:/var/lib/ldap:/bin/false
apache:x:48:48:Apache:/var/www:/bin/false

cathy ~> grep -c false /etc/passwd
7
cathy ~> grep -i ps ~/.bash* | grep -v history
/home/cathy/.bashrc:PS1="\[\033[1;44m\]$USER is in \w\[\033[0m\] "
```

第一个例子，用户 **cathy** 把 `/etc/passwd` 里面包含 `root` 字符串的行显示出来。然后显示了包含所搜索字符的行号。

第三个命令她检查了哪个用户没有使用 **bash**，但是使用 **nologin shell** 的账户不显示出来。

然后他计算了哪些错误使用 `/bin/false` 作为 **shell** 的账号数量。

最后的命令显示了在她主目录中包含 `~/.bash` 的所有文件。排除了包含 **history**，以致于也排除了可能包含相同字符串的 `~/.bash_history` 文件。

现在让我们来看看 **grep** 用正则表达式还能干什么

4.2.2. Grep 与正则表达式

如果你不是在使用 Linux

我们在这些例子中使用支持扩展正则表达式的 GNU **grep**。GNU **grep** 在 Linux 系统里是默认的。如果你在有所有权的系统上工作，那么请使用 `-V` 选项来检查你在使用哪个版本的 **grep**。GNU **grep** 也可以从这里下载：<http://gnu.org/directory/>

4.2.2.1. 锚定行和字

从先前的例子中，我们只想显示那些使用字符串 `root` 开头的行：

```
cathy ~> grep ^root /etc/passwd
root:x:0:0:root:/root:/bin/bash
```

如果我们想看哪个账号什么 **shell** 都没有分配，我们搜索行结束符“`:`”：

```
cathy ~> grep :$ /etc/passwd
news:x:9:13:news:/var/spool/news:
```

要检查 **PATH** 是否在 `~/.bashrc` 中，首先选择“**export**”然后搜索以字符串“**PATH**”开始的行，这样就不会搜索到 **MANPATH** 或者其他可能的路径了：

```
cathy ~> grep export ~/.bashrc | grep '\<PATH'
export PATH="/bin:/usr/lib/mh:/lib:/usr/bin:/usr/local/bin:/usr/ucb:/usr/sbin:$PATH"
```

相似的，`\>`匹配词的结尾。

如果你想找字符串是一个单独的单词（用空格括起来的）。最好使用-w，就像在这个例子里我们显示了 root 分区的信息。

```
cathy ~> grep -w / /etc/fstab
LABEL=/ / ext3 defaults 1 1
```

如果这个选项没有使用，将会显示文件系统的表里的所有的行。

4.2.2.2. 字符族

括号表达式是一个用[]括起来的字符列表。他匹配任何在列表里的单独字符；如果列表的第一个字符是“^”，那么它就匹配所有不在列表中的文件。比如，正则表达式[0123456789]匹配任何单独的数字。

在括号表达式中，一个范围表达式有 2 个用-隔开的字符组成。它匹配在这 2 个字符之间同类的任何单独字符，包括那 2 个字符。使用本地的比较顺序和字符集。比如，再默认的 C 现场，“[a-d]”等于“[abcd]”。许多本地以字典的顺序排列字符。这样的情况下“[a-d]”通常不等于“[abcd]”，可能等于“[aAbBcCdD]”，比如。为得到括号表达式的传统的解释，你可以把环境变量 LC_ALL 设置成"C"来使用 C 现场。

最后，某几个已经命名的字符族 已经预先定义在括号表达式了。请查阅 grep 的帮助页面以得到更多关于预先定义表达式的信息。

```
cathy ~> grep [yf] /etc/group
sys:x:3:root,bin,adm
tty:x:5:
mail:x:12:mail,postfix
ftp:x:50:
nobody:x:99:
floppy:x:19:
xfs:x:43:
nfsnobody:x:65534:
postfix:x:89:
cathy ~> ls *[1-9].xml
appl.xml chap1.xml chap2.xml chap3.xml chap4.xml
```

在这个例子中，所有包含“y”或者“f”的行最先显示，后面是一个例子如何使用指定范围使用 ls。

4.2.2.3. 通配符

使用“.”来匹配单个字符。如果你想得到一个以“c”开头，“h”结尾的 5 个字符的英语字典单词，（解决猜字游戏的好办法）：

```
cathy ~> grep '\<c...h\>' /usr/share/dict/words
catch
clash
cloth
coach
couch
```

```
cough
crash
crush
```

如果你想显示.字符，使用 `-F` 选项。

要匹配多个字符，使用`*`，这个例子从系统的字典里面选择所有“c”开头和“h”结尾的单词。

```
cathy ~> grep '\<c.*h\>' /usr/share/dict/words
caliph
cash
catch
cheesecloth
cheetah
--output omitted--
```

如果你想在文件或者输出里寻找`*`，使用`-F`选项。

```
cathy ~> grep * /etc/profile
cathy ~> grep -F '*' /etc/profile
for i in /etc/profile.d/*.sh ; do
```

4.3. 模式匹配使用 Bash 特性

4.3.1. 字符范围

从 `grep` 和正则表达式出发，`shell` 里有很多无需用外部程序就可以直接使用的匹配模板。

就和你已经知道的一样，`*`和`?`匹配任何单个字符，下面分别引用这些特殊字符来匹配他们的字面值：

```
cathy ~> touch "*"
cathy ~> ls "*"
*
```

你也可以使用方括号来匹配任何 `enclosed character or range of characters`，如果字符被-分隔，一个例子

```
cathy ~> ls -ld [a-cx-z]*
drwxr-xr-x 2 cathy cathy 4096 Jul 20 2002 app-defaults/
drwxrwxr-x 4 cathy cathy 4096 May 25 2002 arabic/
drwxrwxr-x 2 cathy cathy 4096 Mar 4 18:30 bin/
drwxr-xr-x 7 cathy cathy 4096 Sep 2 2001 crossover/
drwxrwxr-x 3 cathy cathy 4096 Mar 22 2002 xml/
```

列出了 `cathy` 主目录里的所有以 `a,b,c,x,y,z` 开头的文件。

4.3.2. 字符 classes

4.4. 总结

4.5. 练习

第五章 GNU SED 流编辑器

本章结束你会了解到以下话题

什么是 sed?

sed 的交互使用

正则表达式和流编辑

在脚本中使用 sed 命令



这仅仅是一个介绍

这些说明远不够完整，不能够作为如何使用 sed 的手册。本章只含有为了在下一章展示一些更为有趣的话题，而且因为每个强力用户应该掌握一些基本的知识来使用这个编辑器完成些工作。

5.1. 介绍

5.1.1. 什么是 sed?

流编辑器是用来从文件读取文本或者从管道实现基本的变化。结果送到标准输出。sed 命令的语法不指定输出文件，但是结果可以通过使用输出重定向来写入到文件中。编辑器并不改变原来的文件。

sed 个其它编辑器比如 vi 和 ed 的区别在于它能够过滤来自管道的输入。在编辑器运行的时候你不要去干涉它；所以 sed 常常被称作批编辑器。此特性允许你在脚本中使用编辑命令，极大的方便了重复性编辑任务。当面对文件中大量的文本替换的时候，sed 将是一个极大的帮助。

5.1.2. sed 命令

sed 程序可以通过使用正则表达式来实现文本的模板替换和删除，就和使用 grep 一样，见 4.2 节。

编辑命令和 vi 编辑器的命令很相近。

5.2. 交互编辑

5.2.1. 打印包含 pattern 的行

你可以用 `grep` 做不少事情，但是你不能用它来“查找和替换”

这是我们的文本文件例子：

```
sandy ~> cat -n example
1 This is the first line of an example text.
2 It is a text with errors.
3 Lots of errors.
4 So much errors, all these errors are making me sick.
5 This is a line not containing any errors.
6 This is the last line.
sandy ~>
```

我们象让 `sed` 找到所有“errors”的行，我们使用 `p` 来得到结果。

```
sandy ~> sed '/errors/p' example
This is the first line of an example text.
It is a text with errors.
It is a text with errors.
Lots of errors.
Lots of errors.
So much errors, all these errors are making me sick.
So much errors, all these errors are making me sick.
This is a line not containing any errors.
This is the last line.
```

就像你看到的，`sed` 打印出了整个文件，但是包含搜索字符串的行被打印了 2 次。这不是我们想要的。为了只符合要求的行，使用 `-n` 选项。

```
sandy ~> sed -n '/errors/p' example
It is a text with errors.
Lots of errors.
So much errors, all these errors are making me sick.
sandy ~>
```

5.2.2. 删除包含 pattern 的输入行

我们使用同样的文本文件作为例子。现在我们只想看到那些不包含所要搜索字符串的行：

```
sandy ~> sed '/errors/d' example
This is the first line of an example text.
This is a line not containing any errors.
This is the last line.
sandy ~>
```

`d` 命令使得被排除的行不显示出来。

匹配以第一个模板开头的和第二个模板结尾的行应该写成这样：

```
sandy ~> sed -n '/^This.*errors.$/p' example
This is a line not containing any errors.
sandy ~>
```

5.2.3. 行的范围

这次我们想单独列出包含错误的行。在例子中它们是第 2-4 行。使用 `d` 命令来指定地址的范围：

```
sandy ~> sed '2,4d' example
This is the first line of an example text.
This is a line not containing any errors.
This is the last line.
sandy ~>
```

要打印出文件中以特定行开头的直到最后一行的，使用如下的命令：

```
sandy ~> sed '3,$d' example
This is the first line of an example text.
It is a text with errors.
sandy ~>
```

这个例子中只打印出例子里的头两行。

下面的命令打印出包含“`a text`”的第一次，直到下个包含“`a line`”的一行：

```
sandy ~> sed -n '/a text/,/This/p' example
It is a text with errors.
Lots of errors.
So much errors, all these errors are making me sick.
This is a line not containing any errors.
sandy ~>
```

5.2.4. 用 `sed` 查找替换

在例子中，我们会搜索和替换错误来代替只选择（或者不选择）包含需要查找字符串的行。

```
sandy ~> sed 's/erors/errors/' example
This is the first line of an example text.
It is a text with errors.
Lots of errors.
So much errors, all these errors are making me sick.
This is a line not containing any errors.
This is the last line.
sandy ~>
```

就像你看到的，这并不完全就是我们想要得：在第 4 行，只有第一个待搜索字符串被替换了，左面还有一个“`eror`”没有被替换。使用 `g` 命令来使 `sed` 检查所有的行而不在搜索到第一个匹配

字符串后就停止:

```
sandy ~> sed 's/erors/errors/g' example
This is the first line of an example text.
It is a text with errors.
Lots of errors.
So much errors, all these errors are making me sick.
This is a line not containing any errors.
This is the last line.
sandy ~>
```

要在文件的每一行开始处插入一个字符串, 比如引用:

```
sandy ~> sed 's/^/> /' example
> This is the first line of an example text.
> It is a text with errors.
> Lots of errors.
> So much errors, all these errors are making me sick.
> This is a line not containing any errors.
> This is the last line.
sandy ~>
```

在每行的尾部插入字符串:

```
sandy ~> sed 's/$/EOL/' example
This is the first line of an example text.EOL
It is a text with errors.EOL
Lots of errors.EOL
So much errors, all these errors are making me sick.EOL
This is a line not containing any errors.EOL
This is the last line.EOL
sandy ~>
```

多个查找替换命令用单独选项-e 来隔开:

```
sandy ~> sed -e 's/erors/errors/g' -e 's/last/final/g' example
This is the first line of an example text.
It is a text with errors.
Lots of errors.
So much errors, all these errors are making me sick.
This is a line not containing any errors.
This is the final line.
sandy ~>
```

记住 sed 默认把结果送到标准输出, 比如你的终端窗口。如果你想把输出保存到文件, 使用重定向:

sed option 'some/expression' file_to_process > sed_output_in_a_file

 更多例子

大量的 `sed` 例子可以在你机器的启动文件里找到，通常在 `/etc/init.d` 或者 `/etc/rc.d/init.d`。切换到包含初始化脚本的目录中然后输入如下命令。

```
grep sed*
```

5.3. 非交互编辑

5.3.1. 从文件读取 `sed` 命令

从文件读取 `sed` 命令

多个 `sed` 命令可以一起放到一个文件中，用 `-f` 选项来执行。当建立了一个这样的文件，请确保：

每行的末尾没有多余的空格。

不能使用引用

当进入文本来添加和替换的时候，除了最后一行已为的所有行都要以反斜杠结尾。

5.3.2. 写输出文件

当使用重定向符号 `>` 的时候，写输出就完成了。这是一个例子脚本，用来从纯文本文件建立非常简单的 HTML 文件。

```
sandy ~> cat script.sed
li\
<html>\
<head><title>sed generated html</title></head>\
<body bgcolor="#ffffff">\
<pre>
$a\
</pre>\
</body>\
</html>
sandy ~> cat txt2html.sh
#!/bin/bash
# This is a simple script that you can use for converting text into HTML.
# First we take out all newline characters, so that the appending only happens
# once, then we replace the newlines.
echo "converting $1..."
SCRIPT="/home/sandy/scripts/script.sed"
NAME="$1"
```

```
TEMPFILE="/var/tmp/sed.$PID.tmp"
sed "s/\n/^M/" $1 | sed -f $SCRIPT | sed "s/^M/\n/" > $TEMPFILE
mv $TEMPFILE $NAME
echo "done."
sandy ~>
```

\$1 把第一个参数送到命令中，这个例子中它是要转换的文件的名字：

```
sandy ~> cat test
line1
line2
line3
```

更多的位置参数请参见第七章。

```
sandy ~> txt2html.sh test
converting test...
done.
sandy ~> cat test
<html>
<head><title>sed generated html</title></head>
<body bgcolor="#ffffff">
<pre>
line1
line2
line3
</pre>
</body>
</html>
sandy ~>
```

这个并不是真的，这个例子，只是为了证明 **sed** 的能力，一个更好的解决这个问题的办法请参见 6.3，它使用 **awk** 的 **BEGIN** 和 **END** 结构。



Easy sed

高级编辑器，支持语法的高亮来识别 **sed** 语法。如果你忘记反斜杠之类的它将提供很大的帮助。

5.4. 总结

sed 流变及其是一个强大的命令行工具，能处理数据流：能从管道读取输入行。这使得它和非交互使用。**sed** 编辑器使用类似 **vi** 的命令且支持正则表达式。

sed 工具可以从命令行或者脚本文件读取命令。他经常用来实现查找和替换的包含特定字符串的行的工作。

5.5. 练习

第六章 GNU AWK 编程语言

本章我们会讨论

- ◆ 什么是 gawk
- ◆ 在命令行中使用 gawk 命令
- ◆ 怎么使用 gawk 来格式化文本
- ◆ gawk 怎么使用正则表达式
- ◆ 脚本中的 gawk
- ◆ gawk 和变量



把它变得更有趣味

就像 sed 一样，整本书写了多个版本的 awk。这个介绍远还不够完整，且只是为了能够理解在后面章节中的例子。要得到更多的信息，最好是从 GNU awk 的随附文档开始。。。

6.1. gawk 上路

6.1.1. 什么是 gawk?

gawk 是通常在 UNIX 系统下使用的另外一个流行的流编辑器 awk 的 GNU 版本。虽然 awk 程序常常只是一个 gawk 的连接，但是我们还是称作它为 awk。

awk 的最基本的作用是搜索含有 1 个或者多个模板的文件中的行或者其他文本块。当一行符合搜索的模板，在该行就会实现指定的动作。

在 awk 中的程序和许多其他语言中的程序是不一样的，因为 awk 程序是：你先描述你想处理的数据，然后当找到它们的时候怎么处理。许多其他的语言是“过程的”。你需要具体的描述程序该采取的措施。当使用过程化的语言时，通常更难清楚地描述你需要处理的数据。正因为如此，awk 程序经常能清爽地读写。

6.1.2. Gawk 命令

当你运行 gawk 的时候，你指定一个 awk 程序来通知 awk 该如何做。程序由几个规则组成。（也可能包含函数定义，循环，条件和其他程序结构，高级特性等一系列我们已经遗忘的东西。）每个规则指定了搜索的模板和在搜索到一个模板后执行的动作。

有几种运行 awk 的方法。如果程序很短，最方便的是在命令行运行它：

```
awk PROGRAM inputfile(s)
```

在已经完成的多个改变，通常在多个文件，很方便把 `awk` 命令放到脚本当中，读起来像这样：

```
awk -f PROGRAM?FILE inputfile(s)
```

6.2. 打印程序

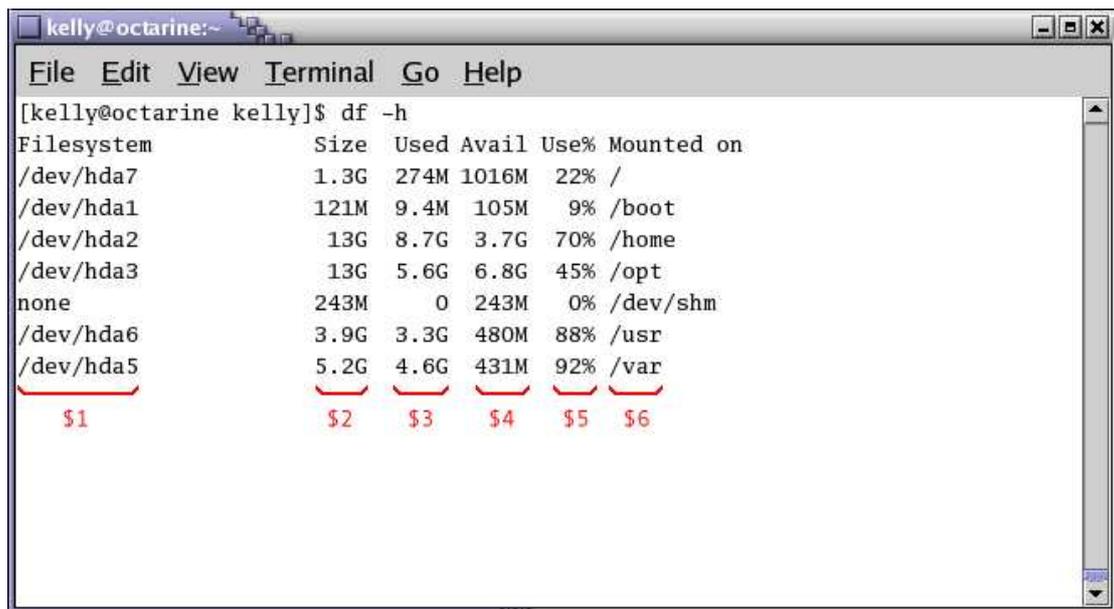
6.2.1. 打印选择的域

`awk` 中的 `print` 命令把输入文件中选择的数据输出。

当 `awk` 读取文件的一行，根据制定的域分隔符把行分开，`FS`，`awk` 的一个环境变量。它被预先定义为一个或者多个空格或者制表符。

变量 `$1$2$3..$n` 把输入行的第一第二第三直到最后一个域保存起来。变量 `$0` 把整行的值保存起来。在下面的图片描述中，我们看到 `df` 命令输出有 6 栏。

图 6-1. `awk` 中的域



6.2.2. 格式化块

没有格式化，只使用输出分隔符的话看上去比较破，插入几个制表符和指示输出的标记会使它变得漂亮很多：

```
kelly@octarine ~/test> ls -ldh * | grep -v total | \
awk '{ print "Size is " $5 " bytes for " $9 }'
```

```
Size is 160 bytes for orig
Size is 121 bytes for script.sed
Size is 120 bytes for temp_file
Size is 126 bytes for test
```

```
Size is 120 bytes for twolines
Size is 441 bytes for txt2html.sh
kelly@octarine ~/test>
```

注意反斜杠的用法，让 **shell** 不把它翻译成分隔命令而使其在下一行继续很长的输入。虽然命令行的输入实际上是没有长度限制的，但是你的显示器不是，打印的纸当然也不是。使用反斜杠也允许拷贝和粘贴以上行道一个终端窗口。

ls 的 **-h** 选项用来支持把大文件的字节数转换成更容易读的格式。当把目录作为参数的时候长列表的输出显示目录中快的总数。这行对我们并没有什么用处，所以我们加上一个 *****。同样的原因我们同样加上 **-d** 选项，万 **一*** 对一个目录展开。

在这个例子中的反斜杠提示了一行的延长。见 3.2 节。

甚至在反序中你也能取出任何栏的数字。下面的例子证明了最苛刻的区分。

```
kelly@octarine ~> df -h | sort -rnk 5 | head -3 | \
awk '{ print "Partition " $6 "\t: " $5 " full!" }'
Partition /var : 86% full!
Partition /usr : 85% full!
Partition /home : 70% full!
kelly@octarine ~>
```

下表给出了特殊格式化字符的总揽。

oooo

引用，**\$** 和其他元字符应该使用反斜杠来进行转义。

6.2.3. 打印命令和正则表达式

用斜杠把正则表达式包含起来可以当作一个 **pattern**。然后正则表达式测试整个文本的每条记录。语法如下：

ooo

下面的例子现实了只有本地磁盘设备的信息，网络文件系统没有显示：

```
kelly is in ~> df -h | awk '/dev\/hd/ { print $6 "\t: " $5 }'
/ : 46%
/boot : 10%
/opt : 84%
/usr : 97%
/var : 73%
/.voll : 8%
kelly is in ~>
```

斜杠也需要转义，因为对于 `awk` 它们有着特殊的含义。

下面的另外一个例子是我们在 `/etc` 目录搜索以 “.conf” 结尾和 “a” 或者 “x” 开头的文件，使用扩展的正则表达式。

```
kelly is in /etc> ls -l | awk '/\<(a|x).*\.conf$/ { print $9 }'
amd.conf
antivir.conf
xcdroast.conf
xinetd.conf
kelly is in /etc>
```

这个例子说明了在正则表达式中的特殊意义。第一个表明了我们想要搜索在第一个搜索字符串之后的任何字符，第二个因为是要查找字符串的一部分所以被转义了（文件名的结束）。

6.2.4. 特殊的 pattern

为了在输出之前加上注释，使用 `BEGIN` 语句。

```
kelly is in /etc> ls -l | \
awk 'BEGIN { print "Files found:\n" } /\<[a|x).*\.conf$/ { print $9 }'
Files found:
amd.conf
antivir.conf
xcdroast.conf
xinetd.conf
kelly is in /etc>
```

加上 `END` 语句能插入文本在整个输入被处理之后。

```
kelly is in /etc> ls -l | \
awk '/\<[a|x).*\.conf$/ { print $9 } END { print \
"Can I do anything else for you, mistress?" }'
amd.conf
antivir.conf
xcdroast.conf
xinetd.conf
Can I do anything else for you, mistress?
kelly is in /etc>
```

6.2.5. Gawk 脚本

往往命令都有点长，你可能想把他们放到脚本中，来重用它们。一个 `awk` 脚本包含定义 `pattern` 和动作的 `awk` 语句。

作为一个说明，我们将建立一个报告来显示占用率最高的分区，请看 6.2.2. 节

```
kelly is in ~> cat diskrep.awk
BEGIN { print "*** WARNING WARNING WARNING ***" }
/<[8|9][0-9]%/ { print "Partition " $6 "\t: " $5 " full!" }
END { print "*** Give money for new disks URGENTLY! ***" }
kelly is in ~> df -h | awk -f diskrep.awk
*** WARNING WARNING WARNING ***
Partition /usr : 97% full!
*** Give money for new disks URGENTLY! ***
kelly is in ~>
```

awk 先打印一个开始信息，然后格式化所有包含一个 8 或者 9 开头的词的行，然后后接一个数字和百分符号，然后加入结束信息。



语法高亮

Awk 是一个编程语言。他的语法能被大多数能对其他语言比如 c,bash,HTML 进行语法高亮编辑器所识别。

6.3. Gawk 变量

既然 awk 处理输入文件，他使用几个变量。一些是可以编辑的，一些是只读的。

6.3.1. 输入块的分隔符

域分隔符，既不是一个单独的字符也不是一个普通的表达式，是控制 awk 把一个输入分割成几个域。输入纪录按分割定义进行字符顺序扫描；域就是在相符的那些文字中间的那部分。

域分隔符代表内建的变量 FS，注意 POSIX 标准的 shell 使用的变量 IFS 和他是有一些区别的。

域分隔符变量的值可以在 awk 程序中用赋值操作符=来改变。通常最好的执行时间是一开始也就是还没有处理任何输入的时候，因此第一个记录就被随合适的分隔符一起读取。要这么做，请使用特殊的 BEGIN pattern

以下的例子，我们编制了一条命令来显示系统里的所有用户及其描述：

```
kelly is in ~> awk 'BEGIN { FS=":" } { print $1 "\t" $5 }' /etc/passwd
--output omitted--
kelly Kelly Smith
franky Franky B.
eddy Eddy White
willy William Black
cathy Catherine the Great
sandy Sandy Li Wong
kelly is in ~>
```

在一个 `awk` 脚本中，他看起来像这样：

```
kelly is in ~> cat printnames.awk
BEGIN { FS=":" }
{ print $1 "\t" $5 }
kelly is in ~> awk -f printnames.awk /etc/passwd
--output omitted--
```

小心地选择输入分隔域来防止出现问题。一个例子来说明这个：说你需要输入想这样的行的形式：

"Sandy L. Wong, 64 Zoo St., Antwerp, 2000X"

你这样写了一个打印出记录中人的名字的命令行或者是脚本：

```
awk 'BEGIN { FS="," } { print $1, $2, $3 }' inputfile
```

但是可能一个人有 PhD，而且可能写成这样：

"Sandy L. Wong, PhD, 64 Zoo St., Antwerp, 2000X"

你的 `awk` 会给出错误的输出。需要的话，使用额外的一个 `awk` 或者 `sed` 来统一数据输出的格式。

默认的输入分隔符是一个或多个空格和制表符。

6.3.2. 输出的分隔符

6.3.2.1. 输出域分隔符

在输出中域通常被空格分隔。当你对 `print` 命令使用正确的语法且字段用逗号分隔时，这将很明显的：

```
kelly@octarine ~/test> cat test
record1 data1
record2 data2
kelly@octarine ~/test> awk '{ print $1 $2}' test
record1data1
record2data2
kelly@octarine ~/test> awk '{ print $1, $2}' test
record1 data1
record2 data2
kelly@octarine ~/test>
```

如果你不输入逗号，`print` 将把输出的项目全部当成一个字段，因此省略默认输出分隔符 `--OFS` 的使用。

6.3.2.2. 输出记录分隔符

整个 `print` 语句的输出叫做输出记录。每个在输出记录里的 `print` 命令的结果，输出一个叫做输出记录分隔符 `ORS` 的字符串。这个变量的默认值是 `"\n"`，一个换行符。因此，每个 `print` 语句生成一个单独的行。

要改变输出域和记录的，只要给 `OFS` 和 `ORS` 赋新的值：

```
kelly@octarine ~/test> awk 'BEGIN { OFS=";" ; ORS="\n-->\n" } \
{ print $1,$2}' test
record1;data1
-->
record2;data2
-->
kelly@octarine ~/test>
```

如果 `ORS` 的值不包含换行，那么程序中的输出就会输出成一个单独行。

6.3.3. 记录的数量

内建的 `NR` 包含了处理过的记录的数量。在读入一个新的输入行之后他会自行增加一次。你可以用它来计算记录的总数，或者在每个输出记录中：

```
kelly@octarine ~/test> cat processed.awk
BEGIN { OFS="-" ; ORS="\n--> done\n" }
{ print "Record number " NR ": \t" $1,$2 }
END { print "Number of records processed: " NR }
kelly@octarine ~/test> awk -f processed.awk test
Record number 1: record1-data1
--> done
Record number 2: record2-data2
--> done
Number of records processed: 2
--> done
kelly@octarine ~/test>
```

6.3.4. 用户定义变量

除了内建的变量之外，你也可以定义自己的变量。当 `awk` 碰到一个不存在的变量（没有事先定义的）的引用时，这个变量就被创建并且使用一个空字符串进行赋值。对于后来所有的引用，就是该变量最后被赋予的那个值。变量可以是一个字符串或者一个数字。输入域的内容也可以被赋予变量。

值可以直接用 `=` 来赋值，或者你可以使用现有变量的值和其他操作符组合。

```
kelly@octarine ~> cat revenues
```

```

20021009 20021013 consultancy BigComp 2500
20021015 20021020 training EduComp 2000
20021112 20021123 appdev SmartComp 10000
20021204 20021215 training EduComp 5000
kelly@octarine ~> cat total.awk
{ total=total + $5 }
{ print "Send bill for " $5 " dollar to " $4 }
END { print "-----\nTotal revenue: " total }
kelly@octarine ~> awk -f total.awk test
Send bill for 2500 dollar to BigComp
Send bill for 2000 dollar to EduComp
Send bill for 10000 dollar to SmartComp
Send bill for 5000 dollar to EduComp
-----
Total revenue: 19500
kelly@octarine ~>

```

类似于 C 的简写 `VAR+=value` 也是可以接受的。

6.3.5. 更多例子

当我们使用 `awk` 脚本时，5.3.2. 节的例子会变得更加容易。

```

kelly@octarine ~/html> cat make-html-from-text.awk
BEGIN { print "<html>\n<head><title>Awk-generated HTML</title></head>\n<body
bgcolor=\"#ffffff\">\{ print $0 }
END { print "</pre>\n</body>\n</html>" }

```

而且当用 `awk` 来替代 `sed` 受，命令也变得更加直截了当。

```

kelly@octarine ~/html> awk -f make-html-from-text.awk testfile > file.html

```

 在你系统上的 `awk` 例子。

我们再次回到包含你系统启动脚本的目录。输入一个和以下相似的命令来查看更多 `awk` 命令的使用方法：

```
grep awk /etc/init.d/*
```

6.3.6. printf 程序

为了更精确的控制 `print` 提供的正常输出格式，可以使用 `printf`。 `printf` 命令可以用来指明每个项目使用的域的宽度，同时也有用于数字的多种格式选择。这一切只要增加一个字符串，叫做格式字符串，控制怎样和那里来打印那些项目。

语法也和 C 语言的 `printf` 语句相似；请察看你的 C 介绍手册。gawk 信息页面包含完整的解释。

6.4. 总结

gawk 工具解释特殊目的的编程语言，处理简单的数据重新格式化的工作，只需要几行代码。他是通常 UNIX `awk` 命令的免费版本。

这个工具从输入数据读取行且能够方便的识别。`print` 是最普通的来过滤和格式化定义的域的程序。

闲置的变量可以直接声明也允许在处理输入流的同时进行简单的计算求和，统计和其他运算。变量和命令可以放到 `awk` 脚本来进行后台处理。

6.5. 练习

第七章 条件语句

本章我们会讨论在 Bash 脚本中使用条件，包含以下几个话题：

- if 语句
- 使用命令的退出状态
- 比较和测试输入和文件
- if/then/else 结构
- if/then/elif/else 结构
- 使用和测试位置参数
- 嵌套 if 语句
- 布尔表达式
- 使用 case 语句

7.1. 介绍 if

7.1.1. 概要

有时候你需要指定 shell 脚本中的依靠命令的成功与否来实施不同过程的行为。if 结构允许你来指定这样的条件。

最精简的 if 命令的语法是：

```
if 测试命令;
then 后继命令;
fi
```

测试命令执行后且它的返回状态是 0，那么后继命令就执行。返回状态是最后一个命令的退出状态，或者没有条件是真的话为 0。

测试命令经常包含数字或者字符串比较测试，但它也可以是任何成功返回状态是 0 和失败返回其他状态的命令。一元表达式经常用来检验文件的状态。如果一个主要的 FILE 参数是 /dev/fd/N 这样的形式的话，就检查文件描述符"N"。stdin, stdout 和 stderr 和他们各自的文件描述符也可以

7.1.1.1. 和 if 使用的表达式

下表包含了一个

表 7-1. 主表达式

Primary 意义

- [-a 文件] 如果文件存在为真。
- [-b 文件] 如果文件存在而且是一个块-特殊文件为真。
- [-c 文件] 为真如果文件存在而且是一个字-特殊文件。

[-d 文件] 为真 如果 文件 存在 而且 是一个 目录。

[-e 文件] 为真 如果 文件 存在。

[-f 文件] 为真 如果 文件 存在 而且 是一个 普通 文件。

[-g 文件] 为真 如果 文件 存在 而且 已经设置了他的 SGID 位。

[-h 文件] 为真 如果 文件 存在 而且 是一个 符号连接。

[-k 文件] 为真 如果 文件 存在 而且 他的粘住位已经设置。

[-p 文件] 为真 如果 文件 存在 而且 是一个 已经命名的管道 (F 如果 O)。

[-r 文件] 为真 如果 文件 存在 而且 是可读的。

[-s 文件] 为真 如果 文件 存在 而且 比零字节大。

[-t FD] 为真 如果 文件 文件描述符已经打开 而且 指向一个终端。

[-u 文件] 为真 如果 文件 存在 而且 已经设置了他的 SUID (set user ID)位。

[-w 文件] 为真 如果 文件 为真 如果 文件 存在 而且 是可写的。

[-x 文件] 为真 如果 文件 存在 而且 是可执行的。

[-O 文件] 为真 如果 文件 存在 而且 属于有效用户 ID。

[-G 文件] 为真 如果 文件 存在 而且 属于有效组 ID。

[-L 文件] 为真 如果 文件 存在 而且 是一个 符号连接。

[-N 文件] 为真 如果 文件 存在 而且 has been mod 如果 ied since it was last read。

[-S 文件] 为真 如果 文件 存在 而且 是一个 socket。

[文件 1 -nt 文件 2] 为真 如果 文件 1 has been changed more recently than 文件 2, or 如果 文件 1 存在 而且 文件 2 does not。

[文件 1 -ot 文件 2] 为真 如果 文件 1 比 文件 2 旧, 或者 文件 2 存在而且 文件 1 不存在。

[文件 1 -ef 文件 2] 为真 如果 文件 1 而且 文件 2 refer to the same device 而且 inode numbers。

[-o 选项名] 为真 如果 shell 选项 "选项名" 开启。

[-z STRING] 为真 如果 "STRING"的长度是零。

[-n STRING] 或者 [STRING] 为真 "STRING"的长度是非零值。

[STRING1 == STRING2] 如果两个字符串相等为真。 "=" may be used instead of "==" for strict POSIX compliance。

[STRING1 != STRING2] 为真 如果 两两个字符串不相等。

[STRING1 < STRING2] 为真 如果 "STRING1" sorts before "STRING2" lexicographically in the current locale。

[STRING1 > STRING2] 为真 如果 "STRING1" sorts after "STRING2" lexicographically in the current locale。

[ARG1 OP ARG2]
 "OP" 是 -eq, -ne, -lt, -le, -gt or -ge 其中一个。 These arithmetic binary operators return 为真 如果 "ARG1" is equal to, not equal to, less than, less than or equal to, greater than, or greater than or equal to "ARG2", respectively。 "ARG1" 而且 "ARG2" are integers。

表达式可以借以下操作符组合起来, listed in decreasing order of precedence:

操作符效果

[! EXPR] 如果 EXPR 为假则为真。

[(EXPR)] 返回 EXPR 的值。 这样可以用来忽略正常的操作符优先级。

[表达式 1 -a 表达式 2] 如果表达式 1 而且表达式 2 同时为真则为真。

[表达式 1 -o 表达式 2] 如果表达式 1 或者表达式 2 其中之一为真则为真。

"["内建运算条件表达式使用一系列基于参数数量的规则。更多关于这个的信息可以在 **Bash** 文档中查找。就像 **if** 使用 **fi** 来结束一样，在条件列完之后必须用"]"来结束。

7.1.1.2.后接 then 语句的命令

后继命令列出了跟在 **then** 语句后面可以使任何有效的 **UNIX** 命令，任何可执行的程序，任何可执行的 **shell** 脚本或者任何 **shell** 语句。重要地记住 **then** 和 **fi** 在 **shell** 里面被认为是分开的语句。因此，在命令行上使用的时候，他们用分号隔开。

在脚本中，**if** 语句的不同部分通常是良好分隔的。以下是一些简单的例子。

7.1.1.3.检查文件

第一个例子检查一个文件是否存在

```
anny ~> cat msgcheck.sh
#!/bin/bash
echo "This scripts checks the existence of the messages file."
echo "Checking..."
if [ -f /var/log/messages ]
then
echo "/var/log/messages exists."
fi
echo
echo "...done."
anny ~> ./msgcheck.sh
This scripts checks the existence of the messages file.
Checking...
/var/log/messages exists.
...done.
```

7.1.1.4.检查 shell 选项

加入到你的 **Bash** 配置文件中去:

```
# These lines will print a message if the noclobber option is set:
if [ -o noclobber ]
then
```

```
echo "Your files are protected against accidental overwriting using redirection."
fi
```



环境

以上的例子将在命令行输入后开始工作:

```
anny ~> if [ -o noclobber ] ; then echo ; echo "your files are protected
against overwriting." ; echo ; fi
your files are protected against overwriting.
anny ~>
```

然而,如果你使用依赖环境的测试,当你在脚本中输入相同的命令你可能得到不同的结果,因为脚本会打开一个新的预期的变量和选项可能没有自动设置的 shell。

7.1.2.if 的简单应用

7.1.2.1 测试退出状态

?变量包含了之前执行命令的退出状态 (最近完成的前台进程)

以下的例子显示了一个简单的测试:

```
anny ~> if [ $? -eq 0 ]
More input> then echo 'That was a good job!'
More input> fi
That was a good job!
anny ~>
```

以下的例子证明了测试命令可以使任何有返回和退出状态的 UNIX 命令,之后 if 再次返回零的退出状态。

```
anny ~> if ! grep $USER /etc/passwd
More input> then echo "your user account is not managed locally"; fi
your user account is not managed locally
anny > echo $?
0
anny >
```

以下能得到同样的结果:

```
anny > grep $USER /etc/passwd
anny > if [ $? -ne 0 ] ; then echo "not a local account" ; fi
not a local account
anny >
```

7.1.2.2. 数字的比较

以下的例子是用了数值的比较:

```
anny > num=`wc -l work.txt`
anny > echo $num
201
anny > if [ "$num" -gt "150" ]
More input> then echo ; echo "you've worked hard enough for today."
More input> echo ; fi
you've worked hard enough for today.
anny >
```

这个脚本在每个星期天由 **cron** 来执行。如果星期的数量一致，他就提醒你清理垃圾。

```
#!/bin/bash
# Calculate the week number using the date command:
WEEKOFFSET=$((date +%V) % 2)
# Test if we have a remainder. If not, this is an even week so send a message.
# Else, do nothing.
if [ $WEEKOFFSET -eq "0" ]; then
echo "Sunday evening, put out the garbage cans." | mail -s "Garbage cans out"
your@your_domain.
```

7.1.2.3. 字符串比较

一个通过比较字符串来测试用户 ID 的例子:

```
if [ "$(whoami)" != 'root' ]; then
echo "You have no permission to run $0 as non-root user."
exit 1;
fi
```

使用 **Bash**，你可以缩短这样的结构。下面是以上测试的精简结构:

```
[ "$(whoami)" != 'root' ] && ( echo you are using a non-privileged account; exit 1 )
```

类似于如果测试为真就执行的“&&”表达式，“||”指定了测试为假就执行。

正则表达式也可以在比较中使用:

```
anny > gender="female"
anny > if [[ "$gender" == f* ]]
More input> then echo "Pleasure to meet you, Madame."; fi
Pleasure to meet you, Madame.
```

```
anny >
```



真正的程序员

多数程序员更喜欢使用和方括号相同作用的内建的 **test** 命令，像这样：

```
test "$(whoami)" != 'root' && (echo you are using a non-privileged account; exit 1)
```

察看 **Bash** 信息页面得到更多关于"**((EXPRESSION))**" 和 "**[[EXPRESSION]]**"结构的模块匹配信息。

7.2.更多 if 的高级使用方法

7.2.1.if/then/else 结构

7.2.1.1.虚构的例子

这是如果一个 **if** 命令测试为真，另外一个 **if** 测试是假而采取以系列行动的例子：

```
freddy scripts> gender="male"
freddy scripts> if [[ "$gender" == "f*" ]]
More input> then echo "Pleasure to meet you, Madame."
More input> else echo "How come the lady hasn't got a drink yet?"
More input> fi
How come the lady hasn't got a drink yet?
freddy scripts>
```

就像 **CONSEQUENT-COMMANDS** 跟在 **then** 语句后面一样，**ALTERNATE-CONSEQUENT-COMMANDS** 跟在 **else** 后面并可以使用任何有返回状态的 **UNIX** 风格命令。

另外一个例子，从 7.1.2.1.扩展开来

```
anny ~> su -
Password:
[root@elegance root]# if ! grep ^$USER /etc/passwd 1> /dev/null
> then echo "your user account is not managed locally"
> else echo "your account is managed from the local /etc/passwd file"
> fi
your account is managed from the local /etc/passwd file
[root@elegance root]#
```

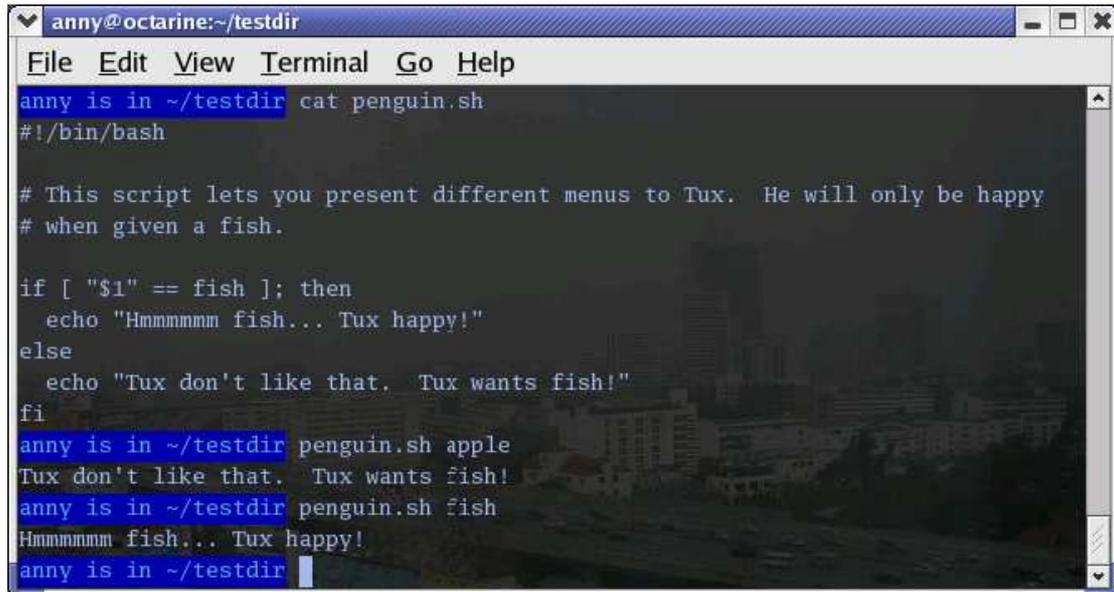
我们切换到 **root** 账号来证明 **else** 语句的效果-**root** 通常是一个本地账号虽然你自己的账号可能被一个特定的系统管理着，比如 **LDAP** 服务器。

7.2.1.2. 检查命令行参数

除了设置完参数然后运行脚本之外，通常更好的方法是通过命令行给变量设置值。

我们使用位置参数`$1, $2, ..., $N`来达到此目的。 `$#`代表了命令行的参数数量， `$0`代表了脚本的名字。

图 7-1 使用 `if` 来测试命令行参数



```
anny@octarine:~/testdir
File Edit View Terminal Go Help
anny is in ~/testdir cat penguin.sh
#!/bin/bash

# This script lets you present different menus to Tux. He will only be happy
# when given a fish.

if [ "$1" == fish ]; then
    echo "HMMMMMM fish... Tux happy!"
else
    echo "Tux don't like that. Tux wants fish!"
fi
anny is in ~/testdir penguin.sh apple
Tux don't like that. Tux wants fish!
anny is in ~/testdir penguin.sh fish
HMMMMMM fish... Tux happy!
anny is in ~/testdir
```

这里是另外一个例子，使用 2 个参数：

```
anny ~> cat weight.sh
#!/bin/bash

# This script prints a message about your weight if you give it your
# weight in kilos and hight in centimeters.

weight="$1"
height="$2"
idealweight=$((height - 110))
if [ $weight -le $idealweight ]; then
    echo "You should eat a bit more fat."
else
    echo "You should eat a bit more fruit."
fi
anny ~> bash -x weight.sh 55 169
+ weight=55
+ height=169
+ idealweight=59
+ '[' 55 -le 59 ']'
+ echo 'You should eat a bit more fat.'
You should eat a bit more fat.
```

7.2.1.3.测试参数的数量

以下的例子显示了怎么改变之前的脚本如果参数少于或者多余 2 个来打印出一条消息:

```

anny ~> cat weight.sh
#!/bin/bash
# This script prints a message about your weight if you give it your
# weight in kilos and hight in centimeters.
if [ ! $# == 2 ]; then
echo "Usage: $0 weight_in_kilos length_in_centimeters"
exit
fi
weight="$1"
height="$2"
idealweight=$((height - 110))
if [ $weight -le $idealweight ]; then
echo "You should eat a bit more fat."
else
echo "You should eat a bit more fruit."
fi
anny ~> weight.sh 70 150
You should eat a bit more fruit.
anny ~> weight.sh 70 150 33
Usage: ./weight.sh weight_in_kilos length_in_centimeters

```

第一个参数代表\$1, 第二个参数代表\$2, 以此类推, 参数数量的总数存在\$#中。

查阅 7.2.5.来得到更多打印消息的方法。

7.2.1.4.测试一个存在的文件

在许多脚本当中这个测试都成功, 因为如果你知道某些功能不工作那运行很多程序也是没有用的:

```

#!/bin/bash
# This script gives information about a file.
FILENAME="$1"
echo "Properties for $FILENAME:"
if [ -f $FILENAME ]; then
echo "Size is $(ls -lh $FILENAME | awk '{ print $5 }')"
echo "Type is $(file $FILENAME | cut -d":" -f2 -)"
echo "Inode number is $(ls -li $FILENAME | cut -d" " -f1 -)"

```

```
echo "$(df -h $FILENAME | grep -v Mounted | awk '{ print "On",$1", \
which is mounted as the",$6,"partition."}')"
else
echo "File does not exist."
fi
```

注意文件是使用变量来指向的；在这个例子中它是脚本的第一个参数。另外，当没有提供任何参数的时候，文件的存放位置通常存储在脚本开始处的变量里，他们的内容是依赖于使用的那些变量。因此，当你在脚本中改变文件的名称，你只要做一次。

7.2.2. if/then/elif/else 结构

7.2.2.1. 概要

这是 if 语句的完全形式：

```
if TEST-COMMANDS; then
CONSEQUENT-COMMANDS;
elif MORE-TEST-COMMANDS; then
MORE-CONSEQUENT-COMMANDS;
else ALTERNATE-CONSEQUENT-COMMANDS;
fi
```

TEST-COMMANDS 执行后，如果他的返回状态是零，那么就执行 CONSEQUENT-COMMANDS。如果 TEST-COMMANDS 返回一个非零值，每个 elif 依次执行，相应的 MORE-CONSEQUENT-COMMANDS 就执行，然后命令结束。

7.2.2.2. 例子

这是一个你可以把它放到 crontab 来每天执行的例子：

```
anny /etc/cron.daily> cat disktest.sh
#!/bin/bash
# This script does a very simple test for checking disk space.
space=`df -h | awk '{print $5}' | grep % | grep -v Use | sort -n | tail -1 | cut -d "%"
-f1 -`
alertvalue="80"
if [ "$space" -ge "$alertvalue" ]; then
echo "At least one of my disks is nearly full!" | mail -s "daily diskcheck" root
else
echo "Disk space normal" | mail -s "daily diskcheck" root
fi
```

7.2.3. if 嵌套语句

在 if 语句里面，你可以使用另外一个 if 语句。只要你能逻辑管理你就可以使用多层嵌套。

以下是一个测试闰年的例子：

```

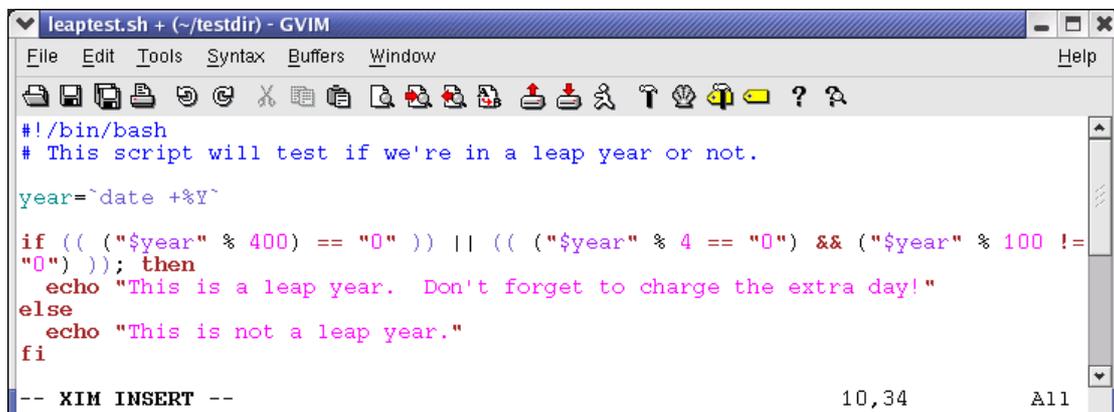
anny ~/testdir> cat testleap.sh
#!/bin/bash
# This script will test if we're in a leap year or not.
year=`date +%Y`
if [ ${year} % 400 -eq 0 ]; then
echo "This is a leap year. February has 29 days."
elif [ ${year} % 4 -eq 0 ]; then
if [ ${year} % 100 -ne 0 ]; then
echo "This is a leap year, February has 29 days."
else
echo "This is not a leap year. February has 28 days."
fi
else
echo "This is not a leap year. February has 28 days."
fi
anny ~/testdir> date
Tue Jan 14 20:37:55 CET 2003
anny ~/testdir> testleap.sh
This is not a leap year.

```

7.2.4.布尔操作

以上的脚本可以用布尔操作符"AND"(&&)和"OR"(||)来缩短。

图 7-2.使用布尔操作符的例子



```

leaptest.sh + (~/testdir) - GVIM
File Edit Tools Syntax Buffers Window Help
year=`date +%Y`
if (( ("${year}" % 400) == "0" )) || (( ("${year}" % 4 == "0" ) && ("${year}" % 100 !=
"0" ) )); then
echo "This is a leap year. Don't forget to charge the extra day!"
else
echo "This is not a leap year."
fi
-- XIM INSERT -- 10,34 All

```

我们使用双括号来测试一个数学表达式，见 3.4.6 节。和使用 let 语句是一样的。如果使用类似

`[$year%4]`,可能你会对方括号的使用感到迷惑,

在其他一些编辑器中, `gvim` 是根据文件格式来进行色彩显示的其中之一; 这些编辑器在发现代码中的错误时候非常有用。

7.2.5. 使用 `exit` 语句和 `if`

我们已经在 7.2.1.3. 节简要的看到了 `exit` 语句。他使整个脚本中止运行。最常使用于判断从用户那里清酒的输入是否正确, 比如一条语句没有成功运行或者某些其他错误发生。

`exit` 语句可以带一个可选参数。参数是一个整数, 代表存贮在 `$_` 中的返回给父进程的退出状态码。

0 参数意味着脚本成功运行完毕。程序员会用其他值来给父进程传递消息, 所以根据子进程的成功或者失败, 父进程采取不同的动作。如果没有参数给 `exit` 语句, 父 `shell` 使用 `$_` 现存值。

下面是一个和 `penguin.sh` 脚本相似的例子, 会对 `feed.sh` 传回一个退出状态:

```
anny ~/testdir> cat penguin.sh
#!/bin/bash
# This script lets you present different menus to Tux. He will only be happy
# when given a fish. We've also added a dolphin and (presumably) a camel.
if [ "$menu" == "fish" ]; then
if [ "$animal" == "penguin" ]; then
echo "HMMMMMMMM fish... Tux happy!"
elif [ "$animal" == "dolphin" ]; then
echo "Pweetpeettreetppeterdepweet!"
else
echo "*prrrrrrrrt*"
fi
else
if [ "$animal" == "penguin" ]; then
echo "Tux don't like that. Tux wants fish!"
exit 1
elif [ "$animal" == "dolphin" ]; then
echo "Pweepwishpeeterdepweet!"
exit 2
else
echo "Will you read this sign?!"
exit 3
fi
fi
```

这个脚本被下面那个调用，

```

anny ~/testdir> cat feed.sh
#!/bin/bash
# This script acts upon the exit status given by penguin.sh
export menu="$1"
export animal="$2"
feed="/nethome/anny/testdir/penguin.sh"
$feed $menu $animal
case $? in
1)
echo "Guard: You'd better give'm a fish, less they get violent..."
;;
2)
echo "Guard: It's because of people like you that they are leaving earth all the time..."
;;
3)
echo "Guard: Buy the food that the Zoo provides for the animals, you ***, how
do you think we survive?"
;;
*)
echo "Guard: Don't forget the guide!"
;;
esac
anny ~/testdir> ./feed.sh apple penguin
Tux don't like that. Tux wants fish!
Guard: You'd better give'm a fish, less they get violent...

```

就像你看到的，退出状态码可以自由选择。退出命令通常有一系列预定义码；请见程序员手册得到每个命令的更多信息。

7.3.使用 case 语句

7.3.1.简单的条件

嵌套 if 语句可能比较美观，但是只要你面临可能采取的一系列的不同的动作时，你可能会迷惑。要处理复杂条件时，使用 case 语句：

```

case EXPRESSION in CASE1) COMMAND-LIST;; CASE2) COMMAND-LIST;; ... CASEN)
COMMAND-LIST;; esac

```

每个分支是一个符合 pattern，在 COMMAND-LIST 中符合的命令就执行。“|”符号用来分割多

个 pattern,“(”操作符中断一个 pattern。每个分支加上他们的后继命令称作一个子句。每个子句必须以“;;”结尾。每个 case 语句以 esac 语句结束。

在这个例子中,我们使用 disktest.sh 脚本的分支来发送一个更有选择性的警告信息:

```

anny ~/testdir> cat disktest.sh
#!/bin/bash
# This script does a very simple test for checking disk space.
space=`df -h | awk '{print $5}' | grep % | grep -v Use | sort -n | tail -1 | cut -d "%"
-f1 -`
case $space in
[1-6]*)
Message="All is quiet."
;;
[7-8]*)
Message="Start thinking about cleaning out some stuff. There's a partition that is $space
% full."
;;
9[1-8])
Message="Better hurry with that new disk... One partition is $space % full."
;;
99)
Message="I'm drowning here! There's a partition at $space %!"
;;
*)
Message="I seem to be running with an nonexistent amount of disk space..."
;;
esac
echo $Message | mail -s "disk report `date`" anny
anny ~/testdir>
You have new mail.
anny ~/testdir> tail -16 /var/spool/mail/anny
From anny@octarine Tue Jan 14 22:10:47 2003
Return-Path: <anny@octarine>
Received: from octarine (localhost [127.0.0.1])
by octarine (8.12.5/8.12.5) with ESMTTP id h0ELALBG020414
for <anny@octarine>; Tue, 14 Jan 2003 22:10:47 +0100
Received: (from anny@localhost)
by octarine (8.12.5/8.12.5/Submit) id h0ELAltn020413
for anny; Tue, 14 Jan 2003 22:10:47 +0100
Date: Tue, 14 Jan 2003 22:10:47 +0100
From: Anny <anny@octarine>
Message-Id: <200301142110.h0ELAltn020413@octarine>
To: anny@octarine

```

```
Subject: disk report Tue Jan 14 22:10:47 CET 2003
Start thinking about cleaning out some stuff. There's a partition that is 87 % full.
anny ~/testdir>
```

当然你可以打开你的邮件程序来检查结果；这只是为了证明脚本发送一个正式的邮件。

更多使用 `case` 语句的例子可以在你系统的初始脚本目录找到。初始化脚本使用 `start` 和 `stop` 分支来启动和停止系统进程。可以在下一节找到一个更具理论性的例子。

7.4. 总结

本章我们学习了怎么在脚本中建立条件来根据命令的成功与否以执行不同的动作。动作可以使用 `if` 语句来决定。允许你进行算术和字符串比较和退出代码的测试，脚本所需的输入和文件。

一个简单的 `if/then/fi` 测试通常先于 `shell` 脚本中的命令为了防止产生输出，因此脚本可以容易的在后台或者通过 `cron` 运行。更多复杂的条件定义通常放置在 `case` 语句。

在测试条件成功时，脚本可以使用 `exit 0` 状态来明确地通知父进程。在失败的时候，任何数字都可能返回。根据返回的代码，父进程可以采取适当的动作。

7.5. 练习

第八章 编写交互脚本

本章我们将讨论怎么通过脚本来和合用户交流

打印用户友好的消息和解释

捕捉用户的输入

提示用户输入

使用文件描述符来读取和写入到多个文件

8.1. 显示用户消息

8.1.1. 交互与否

一些脚本根本不需要来自用户的交互信息。非交互脚本的优势包括:

脚本每次都可以以预测的行为运行。

脚本可以在后台运行。

然而许多脚本,需要来自用户的输入,或者在运行的时候给用户输出信息。交互脚本的优势在于:

可以建立更加灵活的脚本。

用户可自定义脚本使得其产生不同的行为。

脚本可以在运行过程中报告状态。

当编写交互脚本的时候,不要省略注释。打印适当的信息的脚本能变的更加友好且更加容易调试。一个脚本可能做一件完美的工作,但是如果脚本不通知用户正在进行的工作,你将会得到许多来自用户的帮助请求。所以请把告诉用户等待计算完成的输出的提示信息包含进脚本。如果可能的话,尝试提醒下用户需要等待多长的时间。如果再执行某个特定任务的时候等待通常要持续很长时间,你可能会考虑把一些关于脚本输出进度的指示一起集成到脚本当中去。

当提示用户进行输入的时候,同样对输入数据的类型最好给出更多的相关信息。同样在检查参数的时候也采取同样的使用方法信息。

Bash 有 `echo` 和 `printf` 命令提供注释给用户,尽管你现在应该已经熟悉了 `echo` 的使用方法,但是我们在以下还是会讨论更多的例子。

8.1.2. 使用内建 `echo` 命令

内建命令 `echo` 输出他的参数,以空格来分隔,以换行符来结束。返回值总为 0。 `echo` 使用的一些选项:

`-e`:转义反斜杠字符。

`-n`:禁止换行。

作为添加注释的一个例子,我们将把 7.2.1.2 的 `feed.sh` 和 `penguin.sh` 改的好一点。

```
michel ~/test> cat penguin.sh
#!/bin/bash
# This script lets you present different menus to Tux. He will only be happy
# when given a fish. To make it more fun, we added a couple more animals.
if [ "$menu" == "fish" ]; then
if [ "$animal" == "penguin" ]; then
echo -e "HMMMMMMM fish... Tux happy!\n"
elif [ "$animal" == "dolphin" ]; then
echo -e "\a\a\aPweetpeettreetppeterdepweet!\a\a\a\n"
else
echo -e "*prrrrrrrt*\n"
```

```

fi
else
if [ "$animal" == "penguin" ]; then
echo -e "Tux don't like that. Tux wants fish!\n"
exit 1
elif [ "$animal" == "dolphin" ]; then
echo -e "\a\a\a\a\a\aPweepwishpeeterdepweet!\a\a\a"
exit 2
else
echo -e "Will you read this sign?! Don't feed the "$animal"s!\n"
exit 3
fi
fi
michel ~/test> cat feed.sh
#!/bin/bash
# This script acts upon the exit status given by penguin.sh
if [ "$#" != "2" ]; then
echo -e "Usage of the feed script:\t$0 food-on-menu animal-name\n"
exit 1
else
export menu="$1"
export animal="$2"
echo -e "Feeding $menu to $animal...\n"
feed="/nethome/anny/testdir/penguin.sh"
$feed $menu $animal
result="$?"
echo -e "Done feeding.\n"
case "$result" in
1)
echo -e "Guard: \"You'd better give'm a fish, less they get violent...\"\n"
;;
2)
echo -e "Guard: \"No wonder they flee our planet...\"\n"
;;
3)
echo -e "Guard: \"Buy the food that the Zoo provides at the entry, you ***\"\n"
echo -e "Guard: \"You want to poison them, do you?\"\n"
;;
*)
echo -e "Guard: \"Don't forget the guide!\"\n"
;;
esac
fi
echo "Leaving..."

```

```
echo -e "\a\a\aThanks for visiting the Zoo, hope to see you again soon!\n"
michel ~/test> feed.sh apple camel
Feeding apple to camel...
Will you read this sign?! Don't feed the camels!
Done feeding.
Guard: "Buy the food that the Zoo provides at the entry, you ***"
Guard: "You want to poison them, do you?"
Leaving...
Thanks for visiting the Zoo, hope to see you again soon!
michel ~/test> feed.sh apple
Usage of the feed script: ./feed.sh food-on-menu animal-name
```

更多关于转义字符可以参考 3.3.2 节。下表给出 echo 命令能识别的顺序总揽:

表 8-1.echo 命令使用的转义序列

序列	意义
----	----

\a	闹铃
----	----

\b	退格
----	----

\c	强制换行
----	------

\e	退出
----	----

\f	清除屏幕
----	------

\n	新行
----	----

\r	Carriage return.
----	------------------

\t	水平制表符
----	-------

\v	垂直制表符
----	-------

\\	反斜杠
----	-----

\ONNN	The eight-bit character whose value is the octal value NNN (zero to three octal digits).
-------	--

\NNN	The eight-bit character whose value is the octal value NNN (one to three octal digits).
------	---

\xHH	The eight-bit character whose value is the hexadecimal value (one or two hexadecimal digits).
------	---

要得到更多关于 printf 命令的信息以及允许你格式化输出的方法, 请参阅 Bash Info 页面。

8.2. 捕获用户输入

8.2.1. 使用内建 read 命令

8.2.2. 提示用户输入

以下的例子向你展示了使用提示来向用户解释应该输入什么。

```

michel ~/test> cat friends.sh
#!/bin/bash
# This is a program that keeps your address book up to date.
friends="/var/tmp/michel/friends"
echo "Hello, "$USER". This script will register you in Michel's friends database."
echo -n "Enter your name and press [ENTER]: "
read name
echo -n "Enter your gender and press [ENTER]: "
read -n 1 gender
echo
grep -i "$name" "$friends"
if [ $? == 0 ]; then
echo "You are already registered, quitting."
exit 1
elif [ "$gender" == "m" ]; then
echo "You are added to Michel's friends list."
exit 1
else
echo -n "How old are you? "
read age
if [ $age -lt 25 ]; then
echo -n "Which colour of hair do you have? "
read colour
echo "$name $age $colour" >> "$friends"
echo "You are added to Michel's friends list. Thank you so much!"
else
echo "You are added to Michel's friends list."
exit 1
fi
fi
michel ~/test> cp friends.sh /var/tmp; cd /var/tmp
michel ~/test> touch friends; chmod a+w friends
michel ~/test> friends.sh
Hello, michel. This script will register you in Michel's friends database.
Enter your name and press [ENTER]: michel
Enter your gender and press [ENTER] :m
You are added to Michel's friends list.
michel ~/test> cat friends

```

注意这里没有省略输出。这个脚本仅仅储存 Michel 感兴趣的信息，但是除非你已经在里面了，否则将一直提示你已经被加入了列表。

其他人现在可以执行这个脚本：

```
[anny@octarine tmp]$ friends.sh
```

```
Hello, anny. This script will register you in Michel's friends database.
Enter your name and press [ENTER]: anny
Enter your gender and press [ENTER] :f
How old are you? 22
Which colour of hair do you have? black
You are added to Michel's friends list.
```

一会之后，`friends` 列表开始开上去像这样：

```
tille 24 black
anny 22 black
katya 22 blonde
maria 21 black
--output omitted--
```

当然，这个情况并不是理想的，因为每个人都能编辑（但不是删除）`Michel` 的文件。你可以再这个脚本文件里使用特别的存取模式来解决问题，再 `Linux` 手册的介绍中见 `SUID` 和 `SGID`。

8.2.3. 重定向和文件描述符

8.2.3.1. 概要

就像你知道的在 `shell` 的基本用法中，一个命令的输入和输出可以在执行完毕前被重定向，使用一个特殊的符号-重定向操作符-由 `shell` 来解释。重定向夜可以用来为当前 `shell` 执行环境打开和关闭文件。

重定向也可以出现在一个脚本中，所以它可以从一个文件收到输入，比如，或者发送输出到一个文件。然后，用户可以回顾这个输出文件，或者可以被另外一个脚本当作输入。

文件输入输出由追踪为一个给定的进程所有打开文件的整数句柄来完成。这些数字值就是文件描述符。最为人们所知的文件来描述符是 `stdin`, `stdout` 和 `stderr`, 文件描述符的数字分别是 `0`, `1` 和 `2`。这些数字和各自的设备是保留的。`Bash` 可以也可以把网络主机的 `TCP` 或者 `UDP` 端口也认为是一个文件描述符。

下面的输出展示怎么保留文件描述符指向真实的设备：

```
michel ~> ls -l /dev/std*
lrwxrwxrwx 1 root root 17 Oct 2 07:46 /dev/stderr -> ../proc/self/fd/2
lrwxrwxrwx 1 root root 17 Oct 2 07:46 /dev/stdin -> ../proc/self/fd/0
lrwxrwxrwx 1 root root 17 Oct 2 07:46 /dev/stdout -> ../proc/self/fd/1
michel ~> ls -l /proc/self/fd/[0-2]
lrwx----- 1 michel michel 64 Jan 23 12:11 /proc/self/fd/0 -> /dev/pts/6
lrwx----- 1 michel michel 64 Jan 23 12:11 /proc/self/fd/1 -> /dev/pts/6
lrwx----- 1 michel michel 64 Jan 23 12:11 /proc/self/fd/2 -> /dev/pts/6
```

你可能想检查 `info MAKEDRV` 和 `info proc` 来得到更多关于 `/proc` 子目录和你的系统为每个运行的进程操纵文件描述符的方法的信息。

当你以命令行来运行一个脚本的时候，没有什么太多的改变，因为子 shell 进程会使用父进程相同的文件描述符。当没有这个父进程存在的话，比如你使用 cron 来运行一个脚本，标准的文件描述符是管道或者其他（临时）文件，除非使用一些形式重定向。在下面的例子中证明，展示了从例子脚本 at 的输出。

```

michel ~-> date
Fri Jan 24 11:05:50 CET 2003
michel ~-> at 1107
warning: commands will be executed using (in order)
a) $SHELL b) login shell c) /bin/sh
at> ls -l /proc/self/fd/ > /var/tmp/fdtest.at
at> <EOT>
job 10 at 2003-01-24 11:07
michel ~-> cat /var/tmp/fdtest.at
total 0
lr-x----- 1 michel michel 64 Jan 24 11:07 0 -> /var/spool/at/!0000c010959eb (deleted)
l-wx----- 1 michel michel 64 Jan 24 11:07 1 -> /var/tmp/fdtest.at
l-wx----- 1 michel michel 64 Jan 24 11:07 2 -> /var/spool/at/spool/a0000c010959eb
lr-x----- 1 michel michel 64 Jan 24 11:07 3 -> /proc/21949/fd

```

还有一个使用 cron 的

```

michel ~-> crontab -l
# DO NOT EDIT THIS FILE - edit the master and reinstall.
# (/tmp/crontab.21968 installed on Fri Jan 24 11:30:41 2003)
# (Cron version -- $Id: chap8.xml,v 1.8 2005/09/05 12:39:22 tulle Exp $)
32 11 * * * ls -l /proc/self/fd/ > /var/tmp/fdtest.cron
michel ~-> cat /var/tmp/fdtest.cron
total 0
lr-x----- 1 michel michel 64 Jan 24 11:32 0 -> pipe:[124440]
l-wx----- 1 michel michel 64 Jan 24 11:32 1 -> /var/tmp/fdtest.cron
l-wx----- 1 michel michel 64 Jan 24 11:32 2 -> pipe:[124441]
lr-x----- 1 michel michel 64 Jan 24 11:32 3 -> /proc/21974/fd

```

8.2.3.2. 错误重定向

从先前的例子中，很清楚你可以为一个脚本提供输入和输出文件（更多参阅 8.2.4.），但是一些忘记错误重定向的企图——一些之后可以仰赖的输出。同时，如果你幸运的话，错误会 mai 给你，可能的错误原因会被揭示出来。但是不幸的话，错误会导致你的脚本失败而且也不会被捕捉或者发送到任何地方，以至于你载调试的时候什么都做不了。

当重定向错误的时候，注意优先的顺序是有意义的。比如，这个命令，发生在 /var/spool

```
ls -l * 2 > /var/tmp/unaccessible-in-spool
```

将重定向 `ls` 命令的输出到文件 `/var/tmp/unaccessible-in-spool`，这个命令

```
ls -l * > /var/tmp/spoolist 2 >& 1
```

将把标准输出和标准错误都重定向到文件 `spoolist`，这个命令

```
ls -l * 2 >& 1 > /var/tmp/spoolist
```

仅仅把标准输出定向到目标文件里，因为在标准输出重定向之前标准错误已经拷贝到标准输出。

为了方便，如果确定它们将不使用，错误常常重定向到 `/dev/null`。可以在你的系统的起始脚本里找到很多例子。

Bash 允许你使用如下的结构来重定向标准输出和标准错误到名字是 `FILE` 扩展的结果的文件。

&>FILE

等价于 `>FILE 2>&1`，在先前的一系列例子中使用。也常和重定向组合输出到 `/dev/null`，比如当你只是想执行一个命令，而不管产生什么输出或者错误。

8.2.4. 文件输入和输出

8.2.4.1. 使用 `/dev/fd`

`/dev/fd` 目录包含了名为 `0`，`1`，`2` 等的入口。打开文件 `/dev/fd/N` 等价于复制文件描述符 `N`。如果你的系统提供 `/dev/stdin`，`/dev/stdout` 和 `/dev/stderr`，你会看到它们分别等于 `/dev/fd/0`，`/dev/fd/1` 和 `/dev/fd/2`。

`/dev/fd` 的主要使用价值来自于 `shell`。这种机制允许程序以和其他路径名相同的方式使用路径名参数来操纵标准输入和标准输出。如果 `/dev/fd` 在系统中不存在，你将不得不找一个办法来迂回解决这个问题。比如可以使用 `(-)` 来表明程序需要从管道读取就可以达到目的。一个例子：

```
michel ~-> filter body.txt.gz | cat header.txt - footer.txt
This text is printed at the beginning of each print job and thanks the sysadmin
for setting us up such a great printing infrastructure.
Text to be filtered.
This text is printed at the end of each print job.
```

`cat` 命令首先读取文件 `header.txt`，然后他的标准输入是 `filter` 命令的输出，以 `footer.txt` 结尾。折线作为命令行参数涉及标准输入或者标准输出是一种误解，尽管已经在许多程序中被这么认为。当指定折线作为第一个参数的时候也可能产生问题，也许他会被解释成一个先前命令的选项。使用 `/dev/fd` 来允许一致性和防止混淆。

```
michel ~-> filter body.txt | cat header.txt /dev/fd/0 footer.txt | lp
```

在这个清晰的例子中，所有的输出被附加管道通过 `lp` 送往默认的打印机。

8.2.4.2. 读取和 `exec`

8.2.4.2.1. 分配给文件以文件描述符

另外一种着眼文件描述符的方法是把他们认为是分配给文件一个数值。你可以使用文件描述符值，而不是使用文件名。内建命令 `exec` 是用来给文件分配一个文件描述符。使用

```
exec fdN>file
```

分配文件描述符 `N` 给 `file` 进行输出，

```
exec fdN<file
```

分配文件描述符 `N` 给 `file` 进行输入。在文件描述符分配给一个文件后，可以和 `shell` 的冲顶向操作符一起使用，在下面的例子中加以证明：

```
michel ~> exec 4 > result.txt
michel ~> filter body.txt | cat header.txt /dev/fd/0 footer.txt >& 4
michel ~> cat result.txt
This text is printed at the beginning of each print job and thanks the sysadmin
for setting us up such a great printing infrastructure.
Text to be filtered.
This text is printed at the end of each print job.
```



文件描述符 5

使用这个文件描述符可能导致问题，参见 [Advanced Bash-Scripting Guide 第 16 章](#)。强烈建议不要使用它。

8.2.4.2.2. 在脚本中读取

以下是一个例子向你展示怎么样在文件输入和命令行输入中进行转换：

```
michel ~/testdir> cat sysnotes.sh
#!/bin/bash
# This script makes an index of important config files, puts them together in
# a backup file and allows for adding comment for each file.
CONFIG=/var/tmp/sysconfig.out
rm "$CONFIG" 2>/dev/null
echo "Output will be saved in $CONFIG."
exec 7<&0
exec < /etc/passwd
# Read the first line of /etc/passwd
read rootpasswd
echo "Saving root account info..."
echo "Your root account info:" >> "$CONFIG"
echo $rootpasswd >> "$CONFIG"
```

```

exec 0<&7 7<&-
echo -n "Enter comment or [ENTER] for no comment: "
read comment; echo $comment >> "$CONFIG"
echo "Saving hosts information..."
# first prepare a hosts file not containing any comments
TEMP="/var/tmp/hosts.tmp"
cat /etc/hosts | grep -v "^#" > "$TEMP"
exec 7<&0
exec < "$TEMP"
read ip1 name1 alias1
read ip2 name2 alias2
echo "Your local host configuration:" >> "$CONFIG"
echo "$ip1 $name1 $alias1" >> "$CONFIG"
echo "$ip2 $name2 $alias2" >> "$CONFIG"
exec 0<&7 7<&-
echo -n "Enter comment or [ENTER] for no comment: "
read comment; echo $comment >> "$CONFIG"
rm "$TEMP"
michel ~/testdir> sysnotes.sh
Output will be saved in /var/tmp/sysconfig.out.
Saving root account info...
Enter comment or [ENTER] for no comment: hint for password: blue lagoon
Saving hosts information...
Enter comment or [ENTER] for no comment: in central DNS
michel ~/testdir> cat /var/tmp/sysconfig.out
Your root account info:
root:x:0:0:root:/root:/bin/bash
hint for password: blue lagoon
Your local host configuration:
127.0.0.1 localhost.localdomain localhost
192.168.42.1 tintagel.kingarthur.com tintagel
in central DNS

```

8.2.4.3.关闭文件描述符

既然子进程继承打开文件描述符，那么在不时用文件描述符的时候关闭它将是一个良好的习惯，使用以下语句完成：

```
exec fd<&-
```

上面的例子分配给标准输入的文件描述符 7，在每次用户需要读取真实便准输入设备-通常是键盘的时候就关闭。

下面的简单例子冲顶向标准错误到一个管道：

```

michel ~-> cat listdirs.sh
#!/bin/bash
# This script prints standard output unchanged, while standard error is
# redirected for processing by awk.
INPUTDIR="$1"
exec 6>&1
ls "$INPUTDIR"/* 2>&1 >&6 6>&- \
# Closes fd 6 for awk, but not for ls.
| awk 'BEGIN { FS=":" } { print "YOU HAVE NO ACCESS TO" $2 }' 6>&-
exec 6>&-
    
```

8.2.4.4. here 文档

经常性的，你定脚本可能调用其他程序或者脚本来请求输入。here 文档提供了一种通知 shell 从当前源读取输入直到找到仅仅包含搜索字符的行。所有

8.3. 总结

本章，我们学习了如何提供用户注释和怎样提示用户进行输入。通常使用 echo/read 组合。我们也讨论了文件如何使用文件描述符和重定向来进行输入输出，以及怎样组合起来从用户那里得到输入。

我们强调了向用户提供充足信息的重要性。就像常常别人在使用你的脚本的时候，最好提供足够的信息。Here 文档是一种 shell 结构允许为用户建立列表，保留选择。这种结构也能够用来在后台执行其他的交互式任务，而不加干涉。

8.4. 练习

第九章 重复性任务

9.1. for 循环

9.1.1. 如何工作

9.1.2. 例子

9.2. while 循环

9.2.1. What is it?.....108

9.2.2. 例子

9.3. until 循环

- 9.3.1. What is it?.....111**
- 9.3.2. 例子
- 9.4. I/O 重定向和循环**
- 9.4.1. 输入重定向
- 9.4.2. 输出重定向
- 9.5. Break 和 continue**
- 9.5.1.内建 break
- 9.5.2.内建 continue
- 9.5.3. 例子
- 9.6. Making menus with the select built-in.....115**
- 9.6.1. 概要.....115
- 9.6.2. 子菜单
- 9.7.内建 shift**
- 9.7.1. What does it do?.....117
- 9.7.2. 例子
- 9.8. 总结**
- 9.9. 练习**

第十章 深入变量

- 10.1. 变量种类**
- 10.1.1. 概要 assignment of values.....120
- 10.1.2. 使用内建 declare
- 10.1.3. 常量
- 10.2. 数组变量.....122**

10.2.1. 创建数组

10.2.2. Dereferencing the 变量 in an array.....122

10.2.3. 删除数组变量

10.2.4. 数组例子

10.3. Operations on 变量

10.3.1. Arithmetic on 变量.....126

10.3.2. 变量的长度

10.3.3. 变量的转变

10.4. 总结

10.5. 练习

第十一章 函数

本章，我们将讨论

- ◆ 什么是函数
- ◆ 从命令行建立并显示函数
- ◆ 脚本中的函数
- ◆ 向函数传递参数
- ◆ 使用函数的时机

11.1. 介绍

11.1.1. 什么是函数?

shell 函数是一种为后续操作组织命令的方法,使用单个名字来命名这些命令组或者过程。在 shell 或者是脚本当中,这个过程的名字必须是唯一的。所有用来组织函数的命令就像普通命令一样执行。当以一个简单的命令来调用函数的时候,和该函数相关的命令就被执行。函数在必须声明,然后在 shell 里执行:没有新的进程会被创建来打断这个命令。

11.1.2. 函数语法

函数使用以下 2 种形式。

```
function FUNCTION { COMMANDS; }
```

或者

```
FUNCTION () { COMMANDS; }
```

两者都定义了一个 shell 函数 FUNCTION。内建的 function 命令是可选的；然而，如果不使用，必须在函数名后加上小括号。



通常的错误

ooooo

函数体必须以分号或者新行结尾

11.1.3. 函数中参数的位置

函数就像是迷你脚本：他们可以接受参数，他们可以使用只有在函数中有效的变量（使用本地内建 shell）并且他们可以向调用者返回参数。

一个函数也有解释位置参数的机制，然而传递给函数的参数和传递给命令或者脚本的参数是不一样的。

当函数开始执行，在执行期间函数的参数变成位置参数。扩展为位置参数的数量的特别符号 # 更新来反映这个变化。位置参数 0 未改变。Bash 变量 FUNCNAME 在函数执行时被设置为函数的名字。

如果内建的 return 在函数中执行时，在调用完成后函数完成并且执行下一条命令。当函数完成后，位置参数的值和特别参数 # 被重置到函数执行前的值。如果一个数字参数被赋予 return，那个状态就返回了。一个简单的例子：

```
[lydia@cointreau ~/test] cat showparams.sh
#!/bin/bash
echo "This script demonstrates function arguments."
echo
echo "Positional parameter 1 for the script is $1."
echo
test ()
{
echo "Positional parameter 1 in the function is $1."
RETURN_VALUE=$?
echo "The exit code of this function is $RETURN_VALUE."
}
test other_param
[lydia@cointreau ~/test] ./showparams.sh parameter1
This script demonstrates function arguments.
```

```
Positional parameter 1 for the script is parameter1.
Positional parameter 1 in the function is other_param.
The exit code of this function is 0.
[lydia@cointreau ~/test]
```

注意函数的返回值或者退出代码通常储存在一个变量中，因此在以后也可以探测到。你系统的初始脚本使用一种在条件测试中探测 `RETVAL` 值的技巧，像这样：

```
if [ $RETVAL -eq 0 ]; then
<start the daemon>
```

或者像来自 `/etc/init.d/amd` 的 Bash 优化特性使用的脚本例子：

```
[ $RETVAL = 0 ] && touch /var/lock/subsys/amd
```

在 `&&` 后的命令只有在测试是真的时候才执行；这是一个简单的代替 `if/then/fi` 结构的方法。

函数的返回值通常被用作整个脚本的退出代码，你将会看到很多初始脚本用 `exit $RETVAL` 来结束。

11.1.4. 显示函数

所有被当前 shell 所识别的函数可以用 `set` 不带选项地显示出来。函数在他们被使用后保留，除非使用后进行 `unset`。 `which` 命令也可以显示函数：

```
[lydia@cointreau ~] which zless
zless is a function
zless ()
{
zcat "$@" | "$PAGER"
}
[lydia@cointreau ~] echo $PAGER
less
```

这是一种通常配置在用户 shell 资源配置文件里的函数。

函数比起别名和提供一个简单方便的方法来配合用户的环境更灵活。

以下是为了方便 DOS 用户：

```
dir ()
{
ls -F --color=auto -lF --color=always "$@" | less -r
}
```

11.2. 脚本中函数的例子

11.2.1. 循环利用

在你的系统中有大量的脚本使用函数来以结构化的方法处理一系列命令。在某些 linux 系统上，比如，你可以找到 `/etc/rc.d/init.d/functions` 定义文件，使用在所有的初始化脚本中。使用这个方法通常顺序编写一次，常见的任务比如检查进程是否运行，开始或者停止一个守护进程等等。如果某些任务还需要一次，代码就可以重新循环使用。这些文件中的 `checkpid` 函数：

```
# Check if $pid (could be plural) are running
checkpid() {
local i
for i in $* ; do
[ -d "/proc/$i" ] && return 0
done
return 1
}
```

这个函数在相同的脚本中在其他脚本的函数中被重用。守护进程，多数使用在启动一个服务进程的起始脚本中。

11.2.2. 设置路径

本节可能可以在你的 `/etc/profile` 文件中找到，`pathmunge` 函数用来定义然后为 `root` 和其他用户设置路径：

```
pathmunge () {
if ! echo $PATH | /bin/egrep -q "(^|:)$1($|:)" ; then
if [ "$2" = "after" ] ; then
PATH=$PATH:$1
else
PATH=$1:$PATH
fi
fi
}

# Path manipulation
if [ `id -u` = 0 ]; then
pathmunge /sbin
pathmunge /usr/sbin
pathmunge /usr/local/sbin
fi

pathmunge /usr/X11R6/bin after
unset pathmunge
```

这个函数把第一个参数设置为路径名。如果路径名不在当前路径中，就把它加入。传给函数的第二个参数定义了路径是加入到当前 `PATH` 之前还是之后。

普通用户只把 `/usr/X11R6/bin` 加入到他们的路径中，当 `root` 得到了一些包含系统命令的额外目录。使用完毕后，函数 `unset` 所以就不存在了。

11.2.3. 远程备份

以下的例子是用来我用来备份我的文件的。使用 SSH 密钥来启用远程连接。其中定义了 2 个函数，`buplinux` 和 `bupbash`，每个都产生一个 `a.tar` 文件，然后压缩送往远程服务器。最后，本地拷贝被删除。

星期天，只有 `bupbash` 运行。

```
#!/bin/bash
LOGFILE="/nethome/tille/log/backupsript.log"
echo "Starting backups for `date`" >> "$LOGFILE"
buplinux()
{
DIR="/nethome/tille/xml/db/linux-basics/"
TAR="Linux.tar"
BZIP="$TAR.bz2"
SERVER="rincewind"
RDIR="/var/www/intra/tille/html/training/"
cd "$DIR"
tar cf "$TAR" src/*.xml src/images/*.png src/images/*.eps
echo "Compressing $TAR..." >> "$LOGFILE"
bzip2 "$TAR"
echo "...done." >> "$LOGFILE"
echo "Copying to $SERVER..." >> "$LOGFILE"
scp "$BZIP" "$SERVER:$RDIR" > /dev/null 2>&1
echo "...done." >> "$LOGFILE"
echo -e "Done backing up Linux course:\nSource files, PNG and EPS images.\nRubbish removed."
>> "$rm "$BZIP"
}
bupbash()
{
DIR="/nethome/tille/xml/db/"
TAR="Bash.tar"
BZIP="$TAR.bz2"
FILES="bash-programming/"
SERVER="rincewind"
RDIR="/var/www/intra/tille/html/training/"
cd "$DIR"
tar cf "$TAR" "$FILES"
echo "Compressing $TAR..." >> "$LOGFILE"
bzip2 "$TAR"
echo "...done." >> "$LOGFILE"
echo "Copying to $SERVER..." >> "$LOGFILE"
scp "$BZIP" "$SERVER:$RDIR" > /dev/null 2>&1
```

```

echo "...done." >> "$LOGFILE"
echo -e "Done backing up Bash course:\n$FILES\nRubbish removed." >> "$LOGFILE"
rm "$BZIP"
}
DAY=`date +%w`
if [ "$DAY" -lt "2" ]; then
echo "It is `date +%A`, only backing up Bash course." >> "$LOGFILE"
bupbash
else
buplinux
bupbash
fi
echo -e "Remote backup `date` SUCCESS\n-----" >> "$LOGFILE"

```

这个脚本从 cron 运行,意味着没有用户交互,所以我们对 scp 的标准错误进行重定向到/dev/null.

所有的分开的步骤可以组合到下列的命令可能引起争论

```
tar c dir_to_backup/ | bzip2 | ssh server "cat > backup.tar.bz2"
```

然而,如果你对可能恢复脚本错误的中间的结果感兴趣,这不是你想要的。

表达式:

```
command &> file
```

等价于

```
command > fi
```

```
le 2>&1
```

11.3. 总结

函数提供了一种组织那些你想重复执行的命令的容易的方法。当一个函数运行时,位置参数改变为那些函数的。当函数停止时,他们重新设置回调用的程序。程序就像迷你脚本或者就是一个脚本,他们产生退出和返回的代码。

虽然这是比较短的一个章节,但是包含了对任何系统管理员来说最典型的目的--懒惰的重要的知识。

11.4. 练习

第十二章 捕捉信号

本章，我们讨论以下话题

- ◆ 现有的信号
- ◆ 信号的使用法
- ◆ trap 语句的用法
- ◆ 怎么防止用户中断你的程序

12.1. 信号

12.1.1. 介绍

12.1.1.1. 寻找信号的帮助页面

你的系统包含了一个列出所有现有信号帮助页面，但是依据你的操作系统，他可能需要用不同的方法打开。在多数 linux 系统上，将会是 `man 7 signal`。有任何怀疑，使用下面命令来定位准备的帮助页面：

```
man -k signal | grep list
```

或者

```
apropos signal | grep list
```

信号名可以用 `kill -l` 来查找

12.1.1.2. bash shell 的信号

当缺乏陷阱的时候，一个交互式 bash 脚本忽略 `SIGTERM` 和 `SIGQUIT`。`SIGINT` 被捕获并处理，在作业控制在激动状态下，`SIGTTIN`，`SIGTTOU` 和 `SIGTSTP` 也是被忽略的。当以命令替代的结果运行的命令也忽略这些信号，当键盘合成的时候。

`SIGHUP` 默认退出一个 shell，一个交互 shell 将把 `SIGHUP` 送到所有的作业，运行的或者停止的；如果你想禁用特定进程的默认行为的话，请察看 `disown` 的文档。使用内置命令 `shopt`，`huponexit` 选项来杀死所有收到 `SIGHUP` 信号的作业。

12.1.1.3. 使用 shell 来传送信号

以下信号可以使用 `bash shell` 来发送

表 12-1. 在 bash 中的控制信号

标准组合键	意义
<code>ctrl+c</code>	中断信号，向在前台运行的作业发送 <code>SIGINT</code> 。
<code>ctrl+y</code>	延迟悬挂信号，使一个试图从终端读取输入的运行中的进程停止。控制权返回

到 shell，用户可以从前台，后台或者杀死进程。延迟挂起只有在支持这种特性的操作系统才存在。

`ctrl+z` 挂起信号，向正在运行的程序发送 `SIGTSTP`，因此停止程序并且把控制权返回给 shell。

中止设置

检查你的 `stty` 设置，如果你使用“现代的”中断模拟，输出的挂起和继续通常是禁用的。标准的 `xterm` 默认支持 `ctrl+s` 和 `ctrl+q`。

12.1.2. kill 信号的使用

多数现代的 shell，包括 `bash`，有一个内建的 `kill` 函数。在 `bash` 里，信号名和数字都可以被接受为选项，参数可以使作业名或者进程号。使用 `-l` 选项使得一个退出状态可以被报告：`0` 是至少被成功发送的一个信号，有错误发生的话就是非零。

从 `/usr/bin` 使用 `kill` 命令，你的系统可能开启了额外选项，比如杀死进程

下表所列是最常用的信号：

信号名	信号量	效果
<code>SIGHUP</code>	1	挂起
<code>SIGINT</code>	2	从键盘中断
<code>SIGKILL</code>	9	杀死信号
<code>SIGTERM</code>	15	中止信号
<code>SIGSTOP</code>	17,19,23	停止进程

`SIGKILL` 和 `SIGSTOP`

`SIGKILL` 和 `SIGSTOP` 可以被捕获，阻止和忽略。当杀死一个或者一系列进程。通常先尝试最小危险的信号--`SIGTERM`。那样，关心按次序关机的程序得到设计来执行当收到类似清理关闭打开文件的 `SIGTERM` 信号。如果你向一个进程发送一个 `SIGKILL` 信号，你就把进程在关闭前进行清理工作的机会给剥夺了，可能造成意想不到的后果。

但是如果一个清理终止没有起作用，那么 `INT` 或者 `KILL` 信号可能是唯一的选择。通常，当一个进程不是使用 `Ctrl+C` 中止的话，最好使用 `kill -9` 加上进程号。

```
maud: ~> ps ?ef | grep stuck_process
maud 5607 2214 0 20:05 pts/5 00:00:02 stuck_process
```

```
maud: ~> kill ?9 5607
maud: ~> ps ?ef | grep stuck_process
maud 5614 2214 0 20:15 pts/5 00:00:00 grep stuck_process
[1]+ Killed stuck_process
```

当一个进程启动了几个实例，`killall` 可能更方便。它使用和 `kill` 同样的选项，只不过对进程的所

有实例起作用。在正式环境使用之前测试这个命令，即便它可能不能像期望的那样在某些商业用途。

12.2. 陷阱

12.2.1. 概要

可能有这样的情况，你不想使用你的脚本的用户不合时宜地通过键盘来结束进程，比如因为必须提供输入或者必须进行某些清理工作。`trap` 语句捕获到这些序列且能够被编制出来在不活这些信号时候执行一系列的命令。

`trap` 语句的语法是这样的：

```
trap [COMMANDS] [SIGNALS]
```

以上说明 `trap` 命令在捕获所列的信号的时候，可以是带或者不带 `SIG` 前缀的信号名。如果一个信号是 `0` 或者 `EXIT`，当 `shell` 出现的时候 `COMMANDS` 就被执行。如果其中一个信号是 `DEBUG`，那么 `COMMANDS` 的就在每个单独的命令后运行。信号也可以指定为 `ERR`；这样的话 `COMMANDS`

12.2.2. Bash 怎样解释陷阱

每当 `Bash` 收到一个预先设置等待命令完成的陷阱的信号，在命令结束之前，陷阱不会执行。当 `Bash` 通过内建的命令 `wait` 来等待一个异步的命令，陷阱已经发送的信号的接受会导致内建的 `wait` 立即在陷阱执行后返回一个大于 `128` 的推出状态。

12.2.3. 更多例子

12.2.3.1. 检测一个已经使用的变量

每当调试较长的脚本的时候，你可能想给于一个变量 `trace` 属性和陷阱的 `DEBUG` 信息。通常你会像这样 `VARIABLE=value` 来给变量赋值。用下面的几行来代替变量的声明可能会为你脚本的行为提供非常有价值的信息：

```
declare -t VARIABLE=value
trap "echo VARIABLE is being used here." DEBUG
# rest of the script3
```

12.2.3.2. 在退出的时候清理垃圾

`whatis` 命令依靠

```
#!/bin/bash
LOCKFILE=/var/lock/makewhatis.lock
# Previous makewhatis should execute successfully:
[ -f $LOCKFILE ] && exit 0
# Upon exit, remove lockfile.
```

```
trap "{ rm -f $LOCKFILE ; exit 255; }" EXIT
touch $LOCKFILE
makewhatis -u -w
exit 0
```

12.3. 总结

信号可以借助 `kill` 命令或者键盘快捷键来向送往你的程序。这些信号是可以被捕获,仰赖使用 `trap` 语句什么样的会产生什么样的行为。

某些程序忽略信号。但是没有程序能够忽略 `KILL` 信号。

12.4. 练习

附录 A Shell 特性

本文档对常用的 shell 特性和不同 shell 特性做一个整体的描述。

A.1. 常用特性

以下特性在每个 shell 都是标准的。注意 stop, suspend, jobs, bg 和 fg 只有在支持作业控制的系统中存在。

表 A-1. 常用 shell 特性

命令	含义
>	重定向输出
>>	追加到文件
<	重定向输入
<<	"这个" 文档 (重定向输入)
	管道输出
&	置与后台运行
;	同一行的分隔符
*	文件名通配符(多个字符)
?	文件名通配符(单个字符)
[]	匹配任何包含的字符
()	在子 shell 运行
` `	代替包含命令的输出
" "	部分引用(允许变量和命令表达式)
' '	全引用 (没有扩展)
\	引用跟随字符
\$var	对变量取值
\$\$	进程号
\$0	命令名
\$n	第 n 个参数 (n 从 0 到 9)
\$*	所有参数作为一个词
#	注释
bg	后台执行
break	从循环语句跳出
cd	更改目录
continue	继续程序循环
echo	显示输出
eval	评估参数(读取变量用于组合新命令)
exec	执行一个新的 shell
fg	前台执行
jobs	现实活动作业

kill 中止运行的作业
newgrp 更换到一个新组
shift Shift positional parameters
stop 挂起一个前台作业
suspend 挂起一个后台作业
time 对命令计时
umask 设置或列出文件权限
unset 擦除变量或者函数定义
wait 等待一个后台程序完成

附录 B GNU 自由文档守则