版权信息

书名：bash shell脚本编程经典实例（第2版）

作者：[美] 卡尔 • 阿尔宾　JP • 沃森

译者：门佳

ISBN：978-7-115-55378-2

版权声明

O'Reilly Media, Inc. 介绍

O'Reilly 以“分享创新知识、改变世界”为己任。40 多年来我们一直向企业、个人提供成功所必需之技能及思想，激励他们创新并做得更好。

O'Reilly 业务的核心是独特的专家及创新者网络，众多专家及创新者通过我们分享知识。我们的在线学习（Online Learning）平台提供独家的直播培训、图书及视频，使客户更容易获取业务成功所需的专业知识。几十年来 O'Reilly 图书一直被视为学习开创未来之技术的权威资料。我们每年举办的诸多会议是活跃的技术聚会场所，来自各领域的专业人士在此建立联系，讨论最佳实践并发现可能影响技术行业未来的新趋势。

我们的客户渴望做出推动世界前进的创新之举，我们希望能助他们一臂之力。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——*Wired*

“O'Reilly 凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”

——*Business 2.0*

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的领域，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，那就走小路。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

前言

所有的现代操作系统至少都配备了一种 shell，有的还不止一种，其中一些 shell 是面向命令行的，比如本书中讨论的 shell。另外一些 shell 是图形化的，比如 Windows Explorer 或 Macintosh Finder。有些用户也只是在启动常用程序时才跟 shell 打打交道，然后直到退出系统都不会再和 shell 有什么交集。但是大部分用户会在 shell 身上花费大量的时间。你对 shell 了解得越深入，你的工作成效也就越显著。

不管你是系统管理员、程序员还是最终用户，肯定会有那么些时候，一个简单（可能也没那么简单）的 shell 脚本就能助你一臂之力，或者提高某个重要任务的一致性和可重复性。哪怕是利用别名改变或缩短常用命令的名称，其效果也不容小觑。随后我们会详谈这一点。

和任何一种通用编程语言一样，在 shell 中执行特定任务的方法是多种多样的。有时**最佳**方法只有一种，但大多数情况下，等效的解决方法至少有两三种。具体选择哪种方法取决于你的个人风格、创造力以及对不同命令和技术的熟悉程度。无论是身为作者的我们，还是作为读者的你，这一点都毋庸置疑。在大多数例子中，我们只选择一种方法并将其实现。在少数例子中，我们可能会挑选某种特定方法并解释为什么这是最佳选择。偶尔我们还会同时展示多种效果相同的解决方法，这样你就可以从中选择最符合自己需求和环境的那种。

有时需要面对这样的选择：在编写某些代码时，是该采用精巧的写法，还是可读性高的写法？我们坚持选择后者，因为经验告诉我们，不管你现在觉得这段高明的代码有多么清晰、多么一目了然，过个一年半载，经手两位数的项目之后，你肯定会抓耳挠腮地自问当时到底是怎么想的。相信我们：编写明晰的代码并书写文档。日后你会感谢自己的（还有我们）。

目标读者

本书适用于所有的 Unix 用户和 Linux 用户，以及那些可能在一天内接触多种系统的系统管理员。有了这本书，你就可以创建脚本，用更短的时间，更轻松、更稳定地完成更多的工作。

任何人都可以？没错。自动化重复任务、进行简单的替换、定制更为友好和熟悉的操作环境，这些章节会让新用户受益良多。中级用户和管理员可以找到常见任务和难题的全新解决方法。高级用户则可以得到一个能在关键时刻派上用场的技能包，无须再劳心记忆每处语法细节。

目标读者包括：

- 对 shell 知之甚少，但是不满足于只用鼠标指指点点的 Unix 或 Linux 新用户；
- 想快速得到 shell 脚本编程问题答案的 Unix 或 Linux 老用户及系统管理员；
- 在 Unix 或 Linux（甚至是 Windows）环境下工作，希望提高自身工作效率的程序员；
- 需要尽快上手工作的 Unix 或 Linux 系统管理员“菜鸟”或是从 Windows 环境转行过来的新人；
- 希望拥有一个更为强大的脚本编程环境的 Windows 老用户和系统管理员。

本书会简要地讲述初级及中级的 shell 编程知识，更深入的内容请参阅 Cameron Newham 所著的 *Learning bash shell*（第 3 版）以及 Nelson H. F. Beebe 和 Arnold Robbins 合著的 *Classic Shell Scripting*。我们的目标是提供常见问题的对策，重点关注实现方法，而不是理论。希望本书能够帮助你节省思考答案或记忆语法的时间。说白了，这正是我们决定创作此书的原因：这也是我们想要的，一本能够帮助获得灵感，需要时有现成实例可供参考的图书。这样我们就不用再记忆 shell、Perl、C 等语言之间的细微差异了。

本书假设你能够访问 Unix 或 Linux 系统（参见 1.14~1.18 节，或者 15.4 节）并熟悉如何登录、输入基本命令以及使用文本编辑器。大多数实例并不要求你是 root 用户，不过有少数实例，尤其涉及安装 bash 时，需要你是 root 用户。

关于本书

本书讲述的是 bash（GNU Bourne Again Shell），它是 shell 家族的一员，这个大家族中包括了最初的 Bourne shell（sh）、Korn shell（ksh）和 public domain Korn shell（pdksh）。虽然它们以及其他一些 shell（如 dash 和 zsh）并没有重点讨论，但大部分脚本在这些 shell 中运行得还不错。

你既可以逐页阅读，也可以选择性翻阅。最重要的是，我们希望在你不知道如何上手或需要提示时，能够从中轻松地找到正确（或者接近正确）的答案，为你省时省力。

Unix 哲学的很大一部分就是构建各种只专注于做好一件事的简单工具，然后根据需要组合这些工具。这种组合过程经常是通过 shell 脚本实现的，因为这些称为**管道**的命令要么很长，要么难以记忆和输入。在适合的情况下，我们会在 shell 脚本的背景下讲述大量此类工具的用法，将其作为各个部分的黏合剂，以达成最终的目标。

本书第 1 版使用的写作工具是 OpenOffice.org Writer，根据情况运行在 Linux 或 Windows 主机上，书稿全都用 Subversion 保存（参见附录 D）。开放文档格式的性质为写作过程的诸多重要方面带来了便利，例如交叉参考和代码提取（参见 13.18 节）。源文档随后转换成了用于出版的 DocBook 格式。

本书（第 2 版）转向了 O'Reilly Atlas 系统上的 Asciidoc 和 Git，效果非常好。十分感谢 O'Reilly 的产品与工具部门所提供的帮助。

GNU软件

本书中讨论的 bash 以及其他很多软件是 GNU 项目的一部分。GNU（发音为 guh-noo，类似于 canoe）是“GNU's Not Unix”的递归式缩写，该项目可以追溯到 1984 年，其目标是开发出一款自由的类 Unix 操作系统。

这里不讲太多细节，常说的 Linux 其实指的是包含各种支持软件的内核。这个内核的周围还有各种 GNU 工具，根据你使用的发行版，可

能还包括其他种类的软件。不过，Linux 内核本身并不是 GNU 软件。

GNU 项目认为 Linux 实际上应该称作“GNU/Linux”，这么说并不是没有道理，因而有些发行版（尤其是 Debian）也采用了这种叫法。因此，GNU 的目标照理说算是达到了，尽管结果算不上纯粹的 GNU。

GNU 项目贡献了大量卓越的软件，bash 就是其中的明星之一。书中讲到的每种工具差不多都有 GNU 版本，尽管 GNU 工具的特性更为丰富，（通常）也更友好，但有时在用法上存在少量差异。15.3 节会谈到这个问题，但 20 世纪 80 年代和 90 年代的商业 Unix 厂商对这些差异也负有很大责任。

有关 GNU、Unix 和 Linux 方方面面的讨论已经够多了（像本书这种厚度的著作就有好几本），不过我们觉得这样的简述也算差不多了。更多详情可以参阅 GNU 网站。

关于代码示例

当在本书中展示 shell 脚本的可执行代码时，我们通常会将其显示在一个缩进区域内，如下所示：

```
$ ls
a.out cong.txt def.conf file.txt more.txt zebra.list
$
```

第一个字符一般是美元符号（$），表示在 bash shell 提示符处输入命令。（记住，这个提示符是可以修改的，如 16.2 节中所述，这意味着你自己用的提示符可能和这里看到的大相径庭。）提示符由 shell 输出，行中余下部分由你输入。类似地，例子中的最后一行通常也是个提示符（还是 $），表明命令执行完毕，控制权已经返回给 shell。

英镑符号或井字符（#）就要复杂些了。在包括 bash shell 脚本在内的很多 Unix 或 Linux 文件中，开头的 # 表示注释，我们在一些代码示例中也采用了这种用法。但如果作为 bash 命令提示符的结尾符号（而非 $），# 则表示当前的用户身份是 root。我们只有一个示例是以 root 身份执行的，应该不会造成什么困惑，但理解其中的含义还是很重要的。

如果示例中没有提示字符串，那么表明当前看到的是 shell 脚本的内容。对于一些比较长的示例，我们还会给脚本加上行号，但这些行号可不属于脚本。

我们偶尔也会以会话日志或命令序列的形式展示示例。有时候，为了让你看到示例或操作结果中的脚本或数据文件，我们会对一个或多个文件使用 cat 命令，如下所示：

```
$ cat data_file
static header line1
static header line2
1 foo
2 bar
3 baz
```

很多长脚本和函数也可以从网络下载，具体参见“使用示例代码”一节。只要条件允许，本书示例会选择使用 !/usr/bin/env bash，这种写法的可移植性要比你在 Linux 或 Mac 中见到的 !/bin/bash 更好。详细内容参见 15.1 节。

另外，你也许会注意到不少代码示例中会有类似以下的内容：

```
# cookbook filename: snippet_name
```

这表示你当前所阅读的代码可以从本书的 GitHub 仓库中下载。代码都在形如 ./chXX/snippet_name 的文件中，其中的 chXX 指明了具体是哪一章，snippet_name 是文件名。

无谓的cat

一些 Unix 用户非常乐于指出其他用户代码中的低效之处。这种具有建设性的批评多是委婉提出的，对方也能欣然接受。

最常见的可能就是“无谓的 cat”（useless use of cat award），如果有人写成 cat file | grep foo，而不是简单地使用 grep foo file，那可就算是“中奖”了。在这种情况下，cat 不仅没有必要，而且还会引发系统开销，因为它是在子进程中运行的。另一种常见用法是 cat file | tr '[A-Z]' '[a-z]'，而非 tr

'[A-Z]' '[a-z]' < file。有时 cat 甚至会导致脚本运行失败（参见 19.8 节）。

但是……（你也知道下面该说什么了吧？）看似没必要的 cat 有时其实是刻意为之。它可以在演示管道的某部分时占据一个位置，随后用其他命令替换掉（甚至可能是 cat -n）。也可以将文件名靠近代码左侧放置，相较于躲在书页最右侧的 < 之后的文件名，这更能清晰地吸引用户注意力。

我们赞赏高效，也认可这是要努力达成的目标，但其重要性已经不可与过去相提并论了。这绝不是在提倡漫不经心的态度，也不是鼓励代码膨胀，我们只是说这并不会立马就把 CPU 拖慢了。要是你喜欢 cat，那就用吧。

关于Perl

除了几个确实有必要的示例，我们刻意在解决方案中尽可能地避免使用 Perl。无论是深度还是广度，本书中所讲到的 Perl 都远不及其他资料。相较于 shell 脚本，Perl 解决方案的代码量通常要多出不少，开销也大得多。shell 脚本和 Perl 脚本之间泾渭分明，而本书可是一本关于 shell 脚本编程的专著。

shell 脚本可以说是 Unix 程序之间的黏合剂，而 Perl 将 Unix 外部程序的大量功能融入了语言本身。这提高了 Perl 的效率，也改善了某些方面的可移植性，但所付出的代价就是用法上的差异以及难以高效地运行仍旧需要的外部程序。

究竟选择哪一种工具，这往往与熟悉程度有关。关键是完成工作，工具的选择是放在其次的。我们会向你展示 bash 及相关工具的多种用法。当需要搞定手头的工作时，你得挑选用什么工具。

更多资源

- *Perl Cookbook, 2nd Edition*，Nathan Torkington 和 Tom Christiansen 合著，O'Reilly 出版
- *Programming Perl, 4th Edition*，Larry Wall 等著，O'Reilly 出版

- *Perl Best Practices*，Damian Conway 著，O'Reilly 出版
- *Mastering Regular Expressions, 3rd Edition*，Jeffrey E. F. Friedl 著，O'Reilly 出版
- *Learning the bash Shell, 3rd Edition*，Cameron Newham 著，O'Reilly 出版
- *Classic Shell Scripting*，Nelson H. F. Beebe 和 Arnold Robbins 合著，O'Reilly 出版
- 本书网址链接请到图灵社区本书页面查看。

排版约定

本书使用了下列排版约定。

- **黑体**

  表示新术语或重点强调的内容。

- 等宽字体（`constant width`）

  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键词等。

- 加粗等宽字体（**`constant width bold`**）

  表示应该由用户输入的命令或其他文本。

- 斜体等宽字体（*`constant width italic`*）

  表示应该由用户输入的值或根据上下文确定的值替换的文本。

该图标表示一般注记。

该图标表示提示或建议。

该图标表示警告或警示。

使用示例代码

书中示例的源代码请到图灵社区本书主页“随书下载”处下载。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无须联系我们获得许可。比如，用本书的几个代码片段写一个程序就无须获得许可，销售或分发 O’Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无须获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如，“*bash Cookbook, 2nd Edition*, by Carl Albing and JP Vossen (O’Reilly). Copyright 2018 Carl Albing and JP Vossen, 978-1-491-97533-6”。

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

O’Reilly在线学习平台（O’Reilly Online Learning）

近 40 年来，O’Reilly Media 致力于提供技术和商业培训、知识和卓越见解，来帮助众多公司取得成功。

我们拥有独一无二的专家和革新者组成的庞大网络，他们通过图书、文章、会议和我们的在线学习平台分享他们的知识和经验。O’Reilly 的在线学习平台允许你按需访问现场培训课程、深入的学习路径、交互式编程环境，以及 O’Reilly 和 200 多家其他出版商提供的大量文本和视频资源。有关的更多信息，请访问 http://oreilly.com。

联系我们

与本书有关的评论和问题，请发给出版社。

美国：

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）

奥莱利技术咨询（北京）有限公司

请访问 http://oreilly.com，到本书页面查看相关勘误。[1]

[1]本书中文版勘误请到图灵社区本书主页查看和提交。——编者注

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

感谢 GNU 软件基金会和 bash 的作者 Brain Fox。感谢 Chet Ramey 从早期的 bash 1.14 版开始一直到 20 世纪 90 年代中期所做的维护和改进工作。还要感谢 Chet 解答我们提出的问题并审阅了本书的初稿。

另外要特别感谢 Cameron Newham 为本书第 1 版提供了一些材料，其中部分内容保留在了第 2 版中。我们强烈推荐 Cameron Newham 和 Bill Rosenblatt 合著的 *Learning the bash Shell*（O'Reilly 出版）。

审稿人

十分感谢我们的审稿人！他们提出了宝贵的反馈和建议，有时候还给出了替代方案，指出我们忽略的问题，大大地提高了本书的质量。书中遗留的任何错误或疏忽，责任全在我们。准确细致的观察便是他们智慧的体现之一：“这句话读起来让人不明所以！”

第 1 版：Yves Eynard、Chet Ramey、William Shotts、Ryan Waldron、Michael Wang。

第 2 版：Chet Ramey、Robert Day、Arnold Robbins。

O'Reilly

感谢 O'Reilly 的整个团队，没有他们，这本书断然难以成形，即便是亮相于世，质量也肯定不会像现在这么优秀。

第 1 版：我们的编辑 Mike Loukides、Derek Di Matteo、Laurel Ruma。

第 2 版：我们的编辑 Jeff Bleiel、Rachel Head、Kristen Brown、James Fraleigh、Ellen Troutman-Zaig、Rebecca Demarest，还有 Matthew Hacker 和其他得力人员。由于种种原因，第 2 版需要付出大量艰辛的工作。感谢大家。你们为项目所做的奉献以及对细节的关注给我们留下了深刻的印象。

作者的话

卡尔

写书从来不是一个人的事，尽管也有孤军奋战的时候。感谢 JP 多年来与我合作完成本书的第 1 版和第 2 版。我们在个人才能和时间安排上的互补，使得本书的质量好过我一个人写作。还要感谢 JP 在系统管理方面付出的大量努力，这为我们提供了部分基础设施。感谢 Mike 认真听取我们关于本书的建议，感谢他在我们陷入困境时鞭策我们继续前进，感谢他在我们挠头抓狂时约束我们。非常感激 Mike 稳定扎实的指导和技术上的传道授业。我的妻子 Cynthia 和孩子们（现在已经是小大人了！）在本书写作过程中始终耐心地给予我支持、鼓励、动力，以及工作时间和空间。由衷地感谢他们！

但是，比摆在眼前的写作任务更为重要的是背景知识和准备工作。我要特别感谢 Ralph Bjork 博士，他在绝大多数人没听说过 Unix 之前就带我踏上了 Unix 之路。他的眼界、远见和指引给我带来的是超出预期的成长。

本书献给我的父母 Hank 和 Betty，他们尽力给予我一切美好——生命、信仰、爱、良好的教育、归属感，以及每个人都愿意传递给孩子们的那些优秀而健康的事物。我对他们感激不尽。

JP

感谢 Cameron 所著的 *Learning the bash Shell*，我自己从中受益良多，在开始写作本书之前，它一直是我的主要参考并贡献了很多有用的材料。感谢卡尔所做的一切。如果没有他，本书的写作肯定是事倍功半。感谢 Mike 启动这个写作项目并不断向前推进，同时还拉来了卡尔。还要感谢 Mike 和卡尔对我的生活和时间管理方面的问题所展现出来的耐心。

本书献给我的父亲，他肯定会乐开花。老人家总是告诉我，只有两个决定是重要的：你做什么和你娶谁。这两个决定我都已经完成了，我觉得自己做得还不错。因此，本书也献给 Karen，在这个比预想还要漫长的写作过程中，她给予了我难以置信的支持、耐心和理解，没有她，哪怕是计算机也了无生趣。最后，感谢 Kate 和 Sam，他们在我前面提到的生活管理问题上帮了大忙。

更多信息

扫描下方二维码，即可获取电子书相关信息及读者群通道入口。

第 1 章　bash入门

什么是 shell ？为什么要关注 shell ？

所有**近代**（指从 1970 年左右开始）的计算机操作系统都采用了某种形式的用户界面，借此指定需要操作系统执行的命令。但是在很多操作系统中，命令行界面是内建的，是同计算机交互的唯一方式。操作系统的命令行界面允许你执行命令，可也就如此而已了。毕竟，除了这个，还有什么别的能做的吗？

将 shell（系统中允许用户输入命令的部分）与其他部分相分离的概念通过 Unix 操作系统流行开来，这里的其他部分包括：输入/输出系统、调度程序、内存管理以及操作系统负责为用户打理的其他事务方面（还有多数用户懒得操心的那些）。shell 无非是另一个程序罢了，它的工作就是为用户执行其他程序。

但这只是变革的开始。shell 不过是在 Unix 上运行的一个程序，如果你不喜欢标准 shell，可以自己编写一个。因此，在 Unix 头个十年的尾声，至少出现了两个竞争对手：Bourne shell（sh，这是最初的 Thompson shell 的接替版本）和 C shell（csh）。到 Unix 第二个十年末，又涌现出了更多选择：Korn shell（ksh）和 bash shell 的首个版本。在 Unix 第三个十年结束时，各种 shell 估计有一打了。

你大概不会坐着喃喃自语：“我今天该用 csh，还是 bash 或 ksh 呢？”不管 Linux（或者 BSD、macOS、Solaris、HP/UX）提供的是哪种标准 shell，你八成都会乐于接受。但是将 shell 与操作系统自身分离使得软件开发人员（例如 bash 之父 Brian Fox 以及 bash 当前的开发者和维护者 Chet Ramey）更容易编写出更好的 shell，你也可以在不改动操作系统的前提下编写出新的 shell。添加新 shell 更是容易得多，用不着说服操作系统厂商将 shell 构建入系统；你要做的就是将 shell 打包，使其能够像其他程序一样安装就行了。

你可能会觉得 shell 无非就是一个接受命令并执行的程序嘛，这也有点太大惊小怪了吧。你想得没错：**一个只能允许你输入命令的 shell**

的确没多大意思。但是有两个因素推动了 Unix shell 的演变：用户便利性和编程。由此产生的现代 shell 所能做的可就远不止处理命令了。

现代 shell 用起来非常趁手。例如，它们能记得你输入过的命令，便于你随后重新使用。你还能编辑这些历史命令。另外，你可以定义自己的命令缩写、快捷方式以及其他功能。对于有经验的用户，输入命令（例如配合简写、快捷方式、命令补全）要比在炫目的窗口化界面中拖曳高效得多。

除了提供这些便利，shell 还是可编程的。很多命令序列需要一遍又一遍地输入。但凡同一件事做了两遍，你就应该自问：“难道就不能写个程序帮我搞定吗？”当然可以。shell 也是一种编程语言，特别设计用于处理计算机系统的命令。如果你想从 WAV 文件中生成上千个 MP3 文件或压缩所有的系统日志文件，都可以写一个 shell 程序（或者说 **shell 脚本**）来实现。只要发现在重复做某项任务，就应该尝试用 shell 脚本实现任务自动化。Perl、Python、Ruby 这些脚本语言的功能更为强大，但 Unix shell（无论你使用的是哪种 shell）可以作为一个很好的学习起点。毕竟你已经知道如何输入命令，干吗不更进一步呢？

1.1 为什么是bash

为什么本书要介绍 bash，而不是别的 shell ？因为 bash 无处不在。它未必是最时髦的，可以说也不是最炫或最厉害的（就算不是，也差不多了），更不是唯一一个作为开源软件发布的 shell，但 bash 的大名尽人皆知。

究其原因，和历史发展有关。第一批 shell 算得上相当不错的编程工具，但不便于日常使用。C shell 中加入了大量的用户便利功能（例如，可以重复用户输入过的命令），但作为编程语言，它又太诡异了。接下来的 ksh（20 世纪 80 年代初期）也没少为用户提供方便，同时还改善了编程语言，看上去已经走在了被广泛接纳的康庄大道上。但 ksh 并非开源软件，作为一款专有软件，它很难被纳入 Linux 这种自由操作系统。（ksh 在 2000 年修改了其许可证，后来又在 2005 年再次改动。）

到了 20 世纪 80 年代后期，Unix 社区认为标准化是一种不错的做法，同时成立了 POSIX 工作组（由 IEEE 负责组织）。POSIX 实现了 Unix 库和实用工具的标准化，其中也包括 shell。标准 shell 主要基于 1988 年版的 ksh，另外还包括部分 C shell 特性以及一些用于弥补不足的新功能。GNU 项目旨在创建一款彻头彻尾的 POSIX 系统，这个系统自然需要一个 POSIX shell，bash 最初正是该项目的一部分。

bash 提供了 shell 程序员所需要的编程特性和命令行用户喜爱的各种便利。它一开始打算作为 ksh 的替代品，但随着自由软件运动变得愈发重要，Linux 越来越流行，bash 的风头很快就盖过了 ksh。

bash 也因此成了我们已知的所有 Linux 发行版以及 macOS（包括早期的 OS X 版本）的默认 shell（Linux 发行版的数量有数百种之多，可能会有一些没有默认使用 bash）。包括 BSD Unix 和 Solaris 在内的其他 Unix 操作系统也可以使用 bash。如果碰到操作系统中没有 bash 的罕见情况，安装起来也很容易。通过 Cygwin 以及新的 Linux 子系统（Ubuntu），bash 甚至可用于 Windows。bash 既是一门强大的编程语言，也是一种优秀的用户界面。它让你在获得复杂编程特性的同时，能够保持键盘输入的便捷性。

学习 bash 准没错。最常见的默认 shell 就是古老的 Bourne shell 和 bash，后者基本上兼容前者。任何现代的主流 Unix 或类 Unix 操作系统肯定安装了其一。但正如前面所说，万一没有的话，你也可以自己动手安装。当然，还有其他种类的 shell。秉承自由软件的精神，所有 shell 的创建者及维护人员都彼此分享观点。要是读过 bash 的变动日志，你就会发现很多地方提到，为了符合其他 shell 的行为而引入或调整了某项特性。不过多数用户并不在意。有什么功能，他们就用什么功能，并乐在其中。要是感兴趣，你可以研究一下其他 shell。不错的选择还有很多，也许你会找到更喜欢的，尽管可能不如 bash 这样普及。

1.2 bash shell

bash 是一种 shell，也就是命令解释器。bash（或者说任何 shell）的主要目的是让用户可以同计算机操作系统交互，以便完成想做的任

务。这通常涉及运行程序，因此 shell 会接受你输入的命令，判断要用到的程序，然后执行命令来启动程序。你还会碰到一些需要执行一系列操作的任务，这些操作要么是重复性的，要么非常复杂。shell 编程（通常称为 shell 脚本编程）允许你对此类任务进行自动化，以实现易用性、可靠性以及可重现性。

如果你刚接触 bash，我们在本书开始部分为你准备了一些基础知识。如果你一直在使用 Unix 或 Linux，大概对 bash 也不会陌生，不过也许你还不知道自己用的就是它。bash 其实就是一种用于执行命令的语言，你输入的命令（如 ls、cd、grep、cat）从某种意义上来说就是 bash 命令。这些命令有些是 bash 内建的，有些是独立的程序。就目前而言，这种差异并不重要。

接下来我们介绍 bash 的几个上手实例。多数系统预装了 bash，只有少数没有。即便是系统自带 bash，也应该知道如何获取及安装 bash，每次新版本的发布都会带来新的特性。

如果你用的正是 bash，对其也比较熟悉，那么可以直接跳到第 2 章阅读。估计你不会按部就班地阅读本书，要是翻到中间部分，你会发现一些展示了 bash 真正威力的实例。不过，首先还是先来学习基础知识。

1.3 提示符揭秘

1.3.1 问题

你肯定很想知道屏幕上的所有符号都代表什么含义。

1.3.2 解决方案

所有的命令行 shell 都采用某种形式的提示符来告诉用户 shell 已经准备好接受输入了。提示符的具体形式取决于很多因素，其中包括操作系统的类型和版本、shell 的类型和版本、发行版以及其他人所做的配置。在 Bourne 系列的 shell 中，提示符结尾的 $ 通常表明你是以普通用户身份登录的，而结尾的 # 表明你是 root。root 账户是系统的管理员，等同于 Windows 系统中的 System 账户（该账

户的权限比 Administrator 还要高）。在典型的 Unix 或 Linux 系统中，root 无所不能，可以执行任何操作。

默认提示符经常还会显示你当前所处的目录路径，不过多是缩写形式，~表示所在的是主目录。有些默认提示符也会加上用户名和你所登录的主机名。要是还不清楚，当你同时用不同的用户名登入 5 台主机时，就会明白了。

下面是一个典型的 Linux 命令行提示符，它表明用户 jp 所登录的主机名为 adams，当前位于用户主目录中。结尾的 $ 说明 jp 是一个普通用户，并非 root：

```
jp@adams:~$
```

当切换到 /tmp 目录之后，提示符如下所示。注意，表示 /home/jp 的 ~ 已经变成了 /tmp：

```
jp@adams:~$
```

1.3.3 讨论

在和命令行打交道时，shell 提示符会是你见得最多的东西。定制提示符的方法有很多，按照你自己的喜好来即可。目前只需要知道如何解读它就够了。当然了，你所用的默认提示符可能和本书并不一样，不过你现在应该能琢磨出其含义。

在有些 Unix 或 Linux 系统中，可以使用 su 和 sudo 命令分享 root 的权限。如果系统中运行了某种强制性访问控制（mandatory access control，MAC）系统，如 NSA 的 SELinux，root 甚至有可能不再是全能的了。

1.3.4 参考

- 1.4 节
- 14.19 节
- 16.2 节
- 17.15 节

1.4 显示当前位置

1.4.1 问题

你现在不确定自己所处的目录，默认的提示符也帮不上忙。

1.4.2 解决方案

使用内建命令 pwd，或设置一个更有帮助的提示符（参见 16.2 节）。例如：

```
bash-4.3$ pwd
/tmp

bash-4.3$ export PS1='[\u@\h \w]$ '
[jp@solaris8 /tmp]$
```

1.4.3 讨论

pwd 是 print working directory（打印工作目录）的缩写，该命令接受两个选项。-L 显示当前的逻辑路径，这也是默认选项。-P 显示当前的物理路径，如果跟随符号链接，结果可能和逻辑路径不同。与此类似，cd 命令也提供了 -P 和 -L 选项：

```
bash-4.3$ pwd
/tmp/dir2

bash-4.3$ pwd -L
/tmp/dir2

bash-4.3$ pwd -P
/tmp/dir1
```

1.4.4 参考

- 16.2 节

1.5 查找并运行命令

1.5.1 问题

你需要在 bash 下查找并运行特定的命令。

1.5.2 解决方案

可以试试 type、which、apropos、locate、slocate、find 和 ls 命令。

1.5.3 讨论

bash 会在环境变量 PATH 中保留一个用于查找命令的目录列表。内建命令 type 会在环境（包括别名、关键字、函数、内建命令、$PATH 中的目录以及命令散列表）中搜索匹配其参数的可执行文件并显示匹配结果的类型和位置。该命令有多个选项，其中值得注意的是 -a，它会打印出所有的匹配结果，而不是只找出第一个匹配。which 命令与 type 类似，但它只搜索 $PATH（以及 csh 别名）。在不同的系统中，which 的形式各异（通常在 BSD 中是一个 csh 脚本，但在 Linux 中是一个二进制文件），不过一般有 -a 选项。如果已知命令名，想知道其确切位置或想看看其是否存在于系统中，可以借助这两个命令。例如：

```
$ type which
which is hashed (/usr/bin/which)

$ type ls
ls is aliased to `ls -F -h'

$ type -a ls
ls is aliased to `ls -F -h'
ls is /bin/ls

$ which which
/usr/bin/which
```

几乎所有的命令都自带某种形式的用法帮助。通常采用的是称为**手册页**（manpage）的在线文档，其中 man 是 manual（手册）的简写。可以使用 man 命令访问这些手册页，man ls 会显示 ls 命令的相关文档。很多程序还有内建的帮助机制，通过 -h 或 --help 这样的“帮助”选项就能使用。尤其是在其他操作系统中的一些程序，如果不提供命令参数，就会直接显示帮助信息。部分 Unix 命令也会这么做，不过大多数没有这种功能。这是因为 Unix 命令还要组合起来形成**管道**，后面我们会讲到。但如果你不知道或忘记了命令名，该怎么办呢？apropos 命令可以根据所提供的正则表达式参数搜索手册页名称及描述。在你忘记所需要的命令名时，该命令尤其管用。它和 man -k 的效果一样：

```
$ apropos music
cms (4) - Creative Music System device driver

$ man -k music
cms (4) - Creative Music System device driver
```

locate 和 slocate 通过查询系统数据库文件（通常由调度程序 cron 运行的作业负责编译和更新）来查找文件或命令，几乎立刻就能得到结果。实际数据库文件的位置、索引内容、检查频率都因系统而异。具体细节可以查阅系统的手册页。slocate（secure locate）存储了权限信息（除文件名和路径之外），以免列出用户没有权限访问的程序。在多数 Linux 系统中，locate 是指向 slocate 的符号链接；在其他系统中，两者可能是不同的程序，也可能根本就没有 slocate。以下就是一个示例：

```
$ locate apropos
/usr/bin/apropos
/usr/share/man/de/man1/apropos.1.gz
/usr/share/man/es/man1/apropos.1.gz
/usr/share/man/it/man1/apropos.1.gz
/usr/share/man/ja/man1/apropos.1.gz
/usr/share/man/man1/apropos.1.gz
```

有关 find 命令的细节，参见第 9 章。

最后同样重要的是 ls 命令。记住，如果你想要执行的命令位于当前目录，则必须在命令前加上 ./，这是出于安全方面的考虑，因为当

前目录通常并不在 $PATH 中（参见 14.3 节和 14.10 节）。

1.5.4 参考

- help type
- man which
- man apropos
- man locate
- man slocate
- man find
- man ls
- 第 9 章
- 4.1 节
- 14.3 节
- 14.10 节

1.6 获取文件的相关信息

1.6.1 问题

你需要文件的更多相关信息，例如类型、属主、是否可执行、有多少硬链接，以及最后一次访问或更改的时间。

1.6.2 解决方案

使用 ls、stat、file 或 find 命令：

```
$ touch /tmp/sample_file

$ ls /tmp/sample_file
/tmp/sample_file

$ ls -l /tmp/sample_file
-rw-r--r-- 1 jp         jp            0 Dec 18 15:03
/tmp/sample_file
$ stat /tmp/sample_file
File: "/tmp/sample_file"
```

```
Size: 0           Blocks: 0          IO Block: 4096   Regular File
Device: 303h/771d Inode:  2310201    Links: 1
Access: (0644/-rw-r--r--) Uid: (  501/      jp)   Gid: ( 501/
jp)
Access: Sun Dec 18 15:03:35 2005
Modify: Sun Dec 18 15:03:35 2005
Change: Sun Dec 18 15:03:42 2005

$ file /tmp/sample_file
/tmp/sample_file: empty

$ file -b /tmp/sample_file
empty

$ echo '#!/bin/bash -' > /tmp/sample_file

$ file /tmp/sample_file
/tmp/sample_file: Bourne-Again shell script text executable

$ file -b /tmp/sample_file
Bourne-Again shell script text executable
```

有关 find 命令的更多细节，参见第 9 章。

1.6.3 讨论

ls 命令只显示文件名，-l 选项可以提供每个文件更详细的信息。ls 的选项很多，可以查询手册页了解其所支持的选项，其中有用的选项包括以下几个。

-a

不隐藏以 .（点号）开头的文件。

-A

和 -a 相似，但不显示两个常见的目录 . 和 ..，因为每个目录中都有这两项。

-F

文件名结尾以下列类型标识符之一显示文件类型。

斜线（/）表示该文件是目录，星号（*）表示该文件是可执行文件，@ 表示符号链接，等号（=）表示套接字，竖线（|）表示 FIFO（first in, first out）缓冲。

-l

使用长列表格式。

-L

显示链接目标文件的信息，而非符号链接本身。

-Q

引用名（quote name）（GNU 扩展，仅部分系统支持）。

-r

逆序排列。

-R

递归显示子目录。

-S

按照文件大小排序。

-1

使用短格式，每行只显示一个文件。

stat、file 和 find 命令都拥有众多控制输出格式的选项，你可以查询手册页了解它们所支持的选项。例如，下列这些选项可以生成类似于 ls -l 的输出：

```
$ ls -l /tmp/sample_file
-rw-r--r--  1 jp         jp                14 Dec 18 15:04
/tmp/sample_file
```

```
$ stat -c'%A %h %U %G %s %y %n' /tmp/sample_file
-rw-r--r-- 1 jp jp 14 Sun Dec 18 15:04:12 2005 /tmp/sample_file

$ find /tmp/ -name sample_file -printf '%m %n %u %g %t %p'
644 1 jp jp Sun Dec 18 15:04:12 2005 /tmp/sample_file
```

注意，并非所有操作系统及其各版本都有这些工具。例如，Solaris 就默认不包含 stat。

另外值得指出的是，目录其实就是一种操作系统会特别处理的文件，因此本节展示的这些命令也完全能够应用于目录，只不过有时可能需要修改命令才能得到想要的结果。例如，可以用 ls -d 列出目录本身的信息（不加该选项的 ls 列出的是目录的**内容**）。

1.6.4 参考

- man ls
- man stat
- man file
- man find
- 第 9 章

1.7 显示当前目录下的所有隐藏（点号）文件

1.7.1 问题

你只想查看目录下的隐藏（点号）文件，然后编辑其中一个已经忘记名字的文件，或者删除一些废弃文件。ls -a 可以显示出包括隐藏文件在内的所有文件，但这种输出往往包含了过多干扰项，而 ls -a .* 的结果也比你想象或需要的更多。

1.7.2 解决方案

使用 ls -d 配合筛选条件。例如：

```
ls -d .*
ls -d .b*
ls -d .[!.]*
ls -d .*/
```

因为每个正常的目录中都包含 . 和 ..，所以就没必要再显示了。你可以用 ls -A 列出目录中除这两个文件外的其余所有文件。对于可以利用通配符（也就是模式）列出文件的命令，下面给出了能够将 . 和 .. 排除在外的通配符写法：

```
$ grep -l 'PATH' ~/.[!.]*
/home/jp/.bash_history
/home/jp/.bash_profile
$
```

1.7.3 讨论

鉴于 shell 处理文件通配符的方式，.* 的行为并不符合你的预期。**文件名扩展**或**通配符匹配**（globbing）[1] 的工作方式如下：任何包含字符 *、? 或 [的字符串均被视为**模式**，会被依字母排序的、匹配该模式的文件名列表所替换。* 可以匹配包括空串在内的任意字符串，? 可以匹配任意单个字符。[] 内出现的字符形成了一个字符列表或范围，可以匹配其中的任意字符。还有其他各种扩展模式匹配操作符，我们就不在此展开细说了（参见 A.15 节和 A.16 节）。因此，*.txt 表示以 .txt 结尾的所有文件，*txt 表示以 txt 结尾的所有文件（模式中没有点号）。f?o 可以匹配 foo 或 fao，但无法匹配 fooo。有鉴于此，你可能认为 .* 会匹配以点号开头的所有文件。

[1]这里特别说明一下 globbing 和 wildcard 的区别：globbing 是对 wildcard 进行扩展的过程。在贝尔实验室诞生的 Unix 中，有一个名为 glob（global 的简写）的独立程序（/etc/glob）。早期 Unix 版本（第 1~6 版，1969—1975 年）的命令解释器（也就是 shell）都要依赖于该程序扩展命令中未被引用的 wildcard，然后将扩展后的结果提供给命令执行。因此，本书正文将 globbing 译为“通配符匹配”，将 wildcard 译为“通配符”。另外，正文中的“扩展模式匹配”是指，如果使用内建命令 shopt 启用了 shell 选项 extglob，那么 shell 就能够识别一些扩展模式匹配操作符，如 ?(pattern-list)、*(pattern-list)、+(pattern-list) 等。——译者注

问题在于 .* 还可以匹配目录 . 和 ..（存在于每个目录中），它们会和其他以点号开头的文件名一同显示出来。如果 ls 的参数是目录名，那么除了目录名之外，它还会列出该目录中的内容。因此，当你使用 ls .* 时，得到的可不仅仅是当前目录下的点号文件，还包括当前目录（.）下的所有文件和子目录、父目录（..）下的所有文件和子目录、当前目录下以点号开头的子目录名及其内容。这种结果可以说非常杂乱，而且通常也用不着这么多输出。

你可以用同样的 ls 命令实验一下，看看加上 -d 选项和没有 -d 选项的区别，然后试试 echo .*。echo 命令只会简单地显示出扩展 .* 后的结果，这些结果就是 ls 命令的参数。

接着还可以试试 echo .[!.]*。.[!.]* 是文件名扩展模式，其中 [] 指定了要匹配的字符列表，但是开头的 ! 对该列表做了求反操作。因此我们要找的是一个点号，然后是**除点号外的**任意字符，接着是任意数量的任意字符。^ 也可以实现相同的求反效果，但 ! 是 POSIX 标准中规定的，可移植性更好。

ls 命令中的一种特殊情况也能帮上忙。如果指定了 -d 选项，同时文件名模式以斜线结尾，那么 ls 命令只显示匹配模式的**目录**，匹配的文件名一概不显示。例如：

```
$ ls -d .v*
.vim  .viminfo  .vimrc
$ ls -d .v*/
.vim
$
```

第一条命令显示出以 .v 开头的 3 个文件名，如果其中有目录，也不会列出其内容（因为使用了 -d 选项）。第二条命令在模式结尾加上了斜线（.v*/），因此 ls 命令只显示匹配该模式的目录，在这种情况下，符合条件的只有目录 .vim。

如果你在命令 ls -d .v*/ 的输出中看到了两条斜线：

```
$ ls -d .v*/
.vim//
$
```

这可能是因为你执行的是加入了 -F 选项的 ls 别名。在命令名前面使用反斜线就可以避开所有的别名：

```
$ \ls -d .v*/
.vim/
$
```

有些文件名很难匹配。.[!.]* 会漏掉名为 ..foo 的文件。你可以加上形如 .??* 的模式来匹配以点号开头且长度至少为 3 个字符的文件名，但是 ls -d .[!.]* .??* 会将同时匹配这两种模式的文件名显示两遍。但如果只使用 .??*，又会漏掉像 .a 这样的文件：

```
$ touch ..foo .a .normal_dot_file normal_file

$ ls -a
. .. ..foo .a .normal_dot_file normal_file

$ ls -d .??*
..foo .normal_dot_file

$ ls -d .[!.]*
.a .normal_dot_file

$ ls -d .[!.]* .??* | sort -u
..foo
.a
.normal_dot_file
```

具体用哪一种取决于你的需求和环境，没有什么通用的解决方案。

如果 ls 命令损坏或不知何故无法使用，那么可以用 echo * 作为一种应急替代。这种方法可行的原因是，shell 会将 * 扩展为当前目录下的所有文件，其结果和 ls 差不多。

1.7.4 参考

- man ls
- `GNU Core Utilities FAQ 中的问题 18
- Unix FAQs 中的 2.11 节
- A.14 节
- A.15 节

1.8 使用shell引用

1.8.1 问题

你需要了解一些使用命令行引用的经验法则。

1.8.2 解决方案

将字符串放进单引号中，除非字符串中包含需要 shell 进行插值的元素。

1.8.3 讨论

shell 会对非引用文本以及双引号中的文本执行扩展和替换操作。思考以下示例：

```
$ echo A coffee is $5?!
A coffee is ?!

$ echo "A coffee is $5?!"
-bash: !": event not found

$ echo 'A coffee is $5?!'
A coffee is $5?!
```

第一个例子要对 $5 进行变量扩展，但因为该变量并不存在，所以扩展结果为空。第二个例子要遵循同样的规则，但我们没有看到任何命令输出，这是因为对 ！进行历史替换时失败了，原因是其不匹配任何历史记录。第三个例子的结果符合我们的预期。

要想将 shell 扩展与字符串字面量混合在一起，可以利用 shell 转义字符或修改引用方式。惊叹号是一种特殊情况，放在其之前的反斜线转义字符不会被删除。可以通过单引号或尾部空格解决这个问题，如下所示：

```
$ echo 'A coffee is $5 for' "$USER" '?!'
A coffee is $5 for jp ?!
```

```
$ echo "A coffee is \$5 for $USER?\!"
A coffee is $5 for jp?\!

$ echo "A coffee is \$5 for $USER?! "
A coffee is $5 for jp?!
```

另外，你无法在一对单引号中再嵌入另一个单引号，使用反斜线也不行，因为单引号内不会执行任何插值操作。解决方法有两种：使用双引号以及转义字符，或者在单引号对之外转义单引号。

```
# 我们会得到一个示意继续输入的提示符，因为目前的引号并不对称
$ echo '$USER won't pay $5 for coffee.'
> ^C

# 错误
$ echo "$USER won't pay $5 for coffee."
jp won't pay for coffee.

# 有效
$ echo "$USER won't pay \$5 for coffee."
jp won't pay $5 for coffee.

# 同样有效
$ echo 'I won'\''t pay $5 for coffee.'
I won't pay $5 for coffee.
```

1.8.4 参考

- 有关 shell 变量和 $VAR 语法的更多信息，参见第 5 章
- 有关 ! 和历史命令的更多信息，参见第 18 章

1.9 使用或替换内建命令与外部命令

1.9.1 问题

你想用自己编写的函数或外部命令替换内建命令，还想知道脚本究竟执行的是哪种命令（例如，是 /bin/echo 还是内建的 echo）。又或者你创建了一个新命令，但可能和已有的外部或内建命令冲突。

1.9.2 解决方案

用 type 和 which 命令查看指定命令是否存在，并确定其是内建命令还是外部命令：

```
$ type cd
cd is a shell builtin

$ type awk
awk is /usr/bin/awk

$ which cd
/usr/bin/which: no cd in
(/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/ \
local/sbin:/usr/bin/X11:/usr/X11R6/bin:/root/bin)

$ which awk
/usr/bin/awk
```

1.9.3 讨论

顾名思义，内建命令就是内建在 shell 自身中的命令，外部命令则是由 shell 运行的外部文件。外部文件可以是二进制文件，也可以是 shell 脚本。理解内外部命令之间的差异很重要。首先，当你使用特定 shell 的某个版本时，内建命令总是可用的，但外部程序就未必已经安装在系统中了。其次，如果你自己的某个程序和内建命令同名，结果会让人很困惑，因为内建命令总是会优先执行（参见 19.4 节）。可以用 enable 命令来关闭或启用内建命令，不过强烈建议不要这么做，除非你有十足把握知道自己在做什么。enable -a 可以列出所有的内建命令及其当前状态（启用或禁用）。

内建命令的问题在于你无法使用 -h 或 --help 选项获得使用帮助，即便有手册页，通常也只是简单地指向 bash 手册页而已。这时候就用得上内建命令 help 了。help 会显示 shell 内建命令的帮助信息。

```
help: help [-dms] [pattern ...]
    Display information about builtin commands.

    Displays brief summaries of builtin commands.  If PATTERN is
    specified, gives detailed help on all commands matching
```

```
PATTERN,
    otherwise the list of help topics is printed.

    Options:
      -d        output short description for each topic
      -m        display usage in pseudo-manpage format
      -s        output only a short usage synopsis for each topic
matching
     PATTERN

    Arguments:
      PATTERN   Pattern specifying a help topic

    Exit Status:
    Returns success unless PATTERN is not found or an invalid
option is given.
```

需要重新定义某个内建命令时，可以用 builtin 命令来避免出现循环现象。例如，我们可以定义一个 shell 函数（参见 10.4 节）来改变 cd 命令的工作方式。

```
cd () {
    builtin cd "$@"
    echo "$OLDPWD --> $PWD"
}
```

要想避开优先级更高的函数或内建命令，而强制使用外部命令，可以利用 enable -n 关闭 shell 内建命令或用 command 忽略 shell 函数。例如，输入 enable -n test，然后执行 test，此时运行的就不再是内建命令，而是 $PATH 中的 test。也可以用 command ls 来执行原生的 ls 命令，而非可能存在的自定义 ls 函数。

1.9.4 参考

- man which
- help help
- help builtin
- help command
- help enable
- help type

- 10.4 节
- 19.4 节
- A.6 节

1.10 确定是否处于交互模式

1.10.1 问题

你手边有一些代码，希望仅在处于（或不处于）交互模式时运行。

1.10.2 解决方案

使用例 1-1 中的 case 语句。

例 1-1 ch01/interactive

```
#!/usr/bin/env bash
# 文件名: interactive

case "$-" in
    *i*) # 在交互式shell中运行的代码位于此处
     ;;
    *) # 在非交互式shell中运行的代码位于此处
     ;;
esac
```

1.10.3 讨论

变量 $- 中保存了一个字符串，其中列出了当前所有的 shell 选项。如果 shell 处于交互模式，则其中会包含 i。

你也可以采用如下代码（同样有效，但更推荐使用例 1-1 中的方法）：

```
if [ -n "$PS1" ]; then
    echo This shell is interactive
else
```

```
    echo This shell is not interactive
fi
```

1.10.4 参考

- help case
- help set
- 有关 case 语句的更多讲解，参见 6.14 节

1.11 将bash安装为默认shell

1.11.1 问题

你所使用的 BSD 系统、Solaris 或其他 Unix 变体并没有将 bash 作为默认 shell，每次都需要手动启动 bash，你现在希望将 bash 设置为默认 shell。

1.11.2 解决方案

先确定已经安装了 bash。在命令行中输入 bash --version。如果获得了版本输出，则说明 bash 已经安装好了：

```
$ bash --version
GNU bash, version 3.00.16(1)-release (i386-pc-solaris2.10)
Copyright (C) 2004 Free Software Foundation, Inc.
$
```

如果没有看到版本号，可能是系统路径中缺少了相应的目录。在一些系统中，chsh -l 或 cat /etc/shells 可以给出一份可用的 shell 清单。否则，可以询问系统管理员 bash 的安装位置，或者是否可以安装 bash。

在 Linux 系统中，chsh -l 会提供一份可用的 shell 清单；而在 BSD 中，会打开编辑器并允许修改设置。在 macOS 中，-l 并非 chsh 的有效选项，执行 chsh 会打开编辑器并允许修改设置，chpass -s *shell* 可以修改用户所用的 shell。

如果已经安装了 bash，可以用 chsh -s 命令修改默认 shell，例如 chsh -s /bin/bash。如果修改失败，可以再试试 chsh、passwd -e、passwd -l、chpass 或者 usermod -s /usr/bin/ bash。要是还不行，那就得咨询系统管理员了，可能需要管理员编辑 /etc/passwd 文件。在多数系统中，/etc/passwd 包含如下内容：

```
cam:pK1Z9BCJbzCrBNrkjRUdUiTtFOh/:501:100:Cameron
Newham:/home/cam:/bin/bash
cc:kfDKDjfkeDJKJySFgJFWErrElpe/:502:100:Cheshire
Cat:/home/cc:/bin/bash
```

作为 root，你可以编辑密码文件每行的最后一个字段，将其修改成想要选用的 shell 的完整路径。如果系统中有 vipw 命令，则应该使用该命令，确保密码文件的一致性。

> 有些系统不允许将未出现在 /etc/shells 中的 shell 作为登录 shell。如果 bash 不在该文件中，那就只能让系统管理员将其添加进去了。

1.11.3 讨论

有些操作系统（尤其是各种 BSD Unix）通常会将 bash 放在 /usr 分区。在这种系统中更改 root shell 时可得三思。如果系统出现引导故障，而你又需要在 /usr 分区挂载前进行处理，那就碰上真正的麻烦了：root 账户没有 shell 可用。因此，最好不要改动 root 账户的默认 shell。至于普通用户的默认 shell，完全可以改成 bash。除非绝对必要，否则使用 root 账户不是什么好做法，这一点不用说你也明白。尽可能使用普通账户。配合 sudo 这类命令，你应该极少会用到 root shell。

如果上面提到的这些方法都无效，还可以使用 exec 将当前的登录 shell 替换成 bash，但胆小的用户还是别这么做了。具体参见 bash FAQ 中的“A7) How can I make bash my login shell?”一节。

1.11.4 参考

- man chsh
- man passwd
- man chpass
- /etc/shells
- bash FAQ，“A7) How can I make bash my login shell?”一节
- 1.12 节
- 14.13 节
- 14.19 节

1.12 持续更新bash

1.12.1 问题

这显然算不上是一个标准的实例，但没人能忽略这个话题，因此无论如何我们都要说说。你得用安全补丁让 bash 和整个系统保持最新状态。

1.12.2 解决方案

保持整个系统处于最新状态超出了本书的范围，这得咨询你的系统管理员并查看相关文档。

如何让 bash 保持最新状态取决于一开始获取 bash 的方式。理想情况下，bash 通常是作为系统的一部分，随着系统一块更新。如果你用的是已经不被支持的古董系统，可就未必如此了。要是你使用了打包系统，并且原始仓库仍有人维护，那么应该可以从仓库［如 EPEL（Extra Packages for Enterprise Linux）或 PPA（Ubuntu Personal Package Archive）］中获取更新。

如果是通过源代码安装的，那就只能由你负责更新源代码并重新构建了。

1.12.3 讨论

大家都清楚为什么要保持最新状态，但我们还是要引用一个众所周知的原因：CVE-2014-6271，它还有一个更为人熟知的名字：shellshock 漏洞。

1.12.4 参考

- Fedora Project Wiki 中的 EPEL 页面
- Personal Package Archives for Ubuntu
- Wikipedia 中的 Shellshock (software bug) 页面
- 1.13 节
- 1.14 节
- 1.15 节
- 1.16 节
- 1.17 节

1.13 获取Linux版的bash

1.13.1 问题

你想要获取 Linux 系统下的 bash，或确认已安装的 bash 是否为最新版。

1.13.2 解决方案

几乎所有的现代 Linux 发行版都包含了 bash。可以用发行版自带的打包工具来确认安装的是否为最新版。要想升级或安装应用程序，你必须是 root 身份、使用 sudo，或知道 root 密码。

有些 Linux 发行版（尤其是 Debian 家族）用 Debian Almquist shell 或 dash 作为 /bin/sh[2]，因为这两种 shell 更为小巧，而且运行速度比 bash 更快。对于那些假定 /bin/sh 指向 bash 的脚本，这种切换会造成很严重的混乱，因为凡是用到 bash 特性的地方全都会失效。更多细节参见 15.3 节。

[2]在这类发行版中，/bin/sh 是指向 Debian Almquist shell 或 dash 的符号链接。——译者注

对于 Debian 及其衍生出的系统（如 Ubuntu 和 Linux Mint），可以使用图形用户界面工具或命令行工具（如 apt-get、aptitude、apt）来确定 bash 的安装情况：

```
apt-get update && apt-get install bash bash-completion bash-doc
```

对于包括 Fedora、Community OS（CentOS）、Red Hat Enterprise Linux（RHEL）在内的 Red Hat 发行版，可以使用图形用户界面的 Add/Remove Application 工具。如果只有命令行，则可以使用下列命令：

```
yum update bash
```

对于 SUSE，可以使用 YaST 的图形用户界面或终端版本。你也可以使用命令行工具 rpm。

1.13.3 讨论

我们不可能涵盖所有的 Linux 发行版，甚至连主流发行版都很难全部讲到，因为它们的发展速度实在是太快了。好在这方面的多数发展是为了提高易用性，搞明白如何在所选用的发行版上安装软件应该不是难事。

如果用的是 LiveCD，鉴于其所采用的只读介质，软件更新和安装基本上不可能成功。如果这种发行版安装到了硬盘上，那么应该可以更新。

要是不确定特定 Linux 发行版中安装的是哪个版本的 bash，可以到 DistroWatch 网站上搜索该发行版并查询相关的软件包表格。例如，DistroWatch 网站中显示了表 1-1 中的内容。

表1-1：Linux Mint中的bash版本

软件包	18 sarah	17.3 rosa	16 petra	15 olivia	14 nadia	13 maya	12 lisa	11 katya	10 julia	……

软件包	18 sarah	17.3 rosa	16 petra	15 olivia	14 nadia	13 maya	12 lisa	11 katya	10 julia	……
bash (4.4)	4.3	4.3	4.2	4.2	4.2	4.2	4.2	4.2	4.1	……

1.13.4 参考

- Debian 说明文档
- Wikipedia 中的 Almquist shell 页面和 Ubuntu Wiki 中的 DashAsBinSh 页面
- Fedora Project Wiki 中的 EPEL 页面
- SuSE 文档
- OpenSuSE 文档
- 1.11 节
- 1.12 节

1.14 获取xBSD版的bash

1.14.1 问题

你想要获取 FreeBSD、NetBSD、OpenBSD 系统下的 bash，或者想要确认已安装的 bash 是否为最新版。

1.14.2 解决方案

根据 Chet Ramey 所维护的 bash 页面：

> bash-4.3 已经包含在 FreeBSD ports collection、OpenBSD packages collection 以及 NetBSD packages collection 中。

要想知道是否已经安装 bash，可以检查 /etc/shells 文件。要想安装或升级 bash，可以使用 `pkg_add` 命令。如果你是 BSD 的老用

户，可能更喜欢用 ports collection，但这里不打算涉及这方面的内容。

要是不确定特定 BSD 发行版中安装的是哪个版本的 bash，可以到 DistroWatch 网站上搜索该发行版并查询相关的软件包表格。

对于 FreeBSD，使用如下命令：

```
pkg_add -vr bash
```

对于 NetBSD，浏览 Application Software for NetBSD，找到符合相应发行版本和架构的最新 bash 软件包，然后使用命令：

```
pkg_add -vu ftp://ftp.netbsd.org/pub/NetBSD/packages/pkgsrc-
2005Q3/NetBSD-2.0/ \
i386/All/bash-3.0pl16nb3.tgz
```

对于 OpenBSD，可以使用 pkg_add -vr 命令。你可能需要将 FTP 路径调整为适应相应的发行版本和架构。另外，也许存在静态编译版本。例如：

```
pkg_add -vr
ftp://ftp.openbsd.org/pub/OpenBSD/3.8/packages/i386/bash-
3.0.16p1.tgz
```

1.14.3 讨论

FreeBSD 和 OpenBSD 将 bash 安装在 /usr/local/bin/bash 中，而 NetBSD 将 bash 安装在 /usr/pkg/bin/bash 中。

1.14.4 参考

- 1.11 节
- 1.12 节
- 15.4 节

1.15 获取macOS版的bash

1.15.1 问题

你想要获取 macOS 系统下的 bash，或确认已安装的 bash 为最新版。

1.15.2 解决方案

根据 Chet Ramey 所维护的 bash 页面：

> Mac OS X（现在称作 macOS）的当前版本（从 Jaguar/Mac OS X 10.2 开始）使用 bash-3.2 作为 /bin/sh。也可以在很多网站上找到 bash-4.3 的 OS X 预编译包，但源代码包通常要更新一些。Darwin 版（MacOS X 的核心）的 bash 可以从 MacPorts、Homebrew、Fink 处获得。

1.15.3 讨论

也可以从源代码构建更新版本的 bash，但仅推荐有经验的用户这么做（参见附录 E）。

1.15.4 参考

- Homebrew 网站
- Fink 项目网站
- 1.12 节
- 附录 E

1.16 获取Unix版的bash

1.16.1 问题

你想要获取 Unix 系统下的 bash，或确认已安装的 bash 是否为最新版。

1.16.2 解决方案

如果在系统或软件包仓库中都找不到，那么可以在 Chet Ramey 所维护的 bash 页面下载二进制文件，或从源代码构建（参见附录 E）。

1.16.3 讨论

根据 Chet Ramey 维护的 bash 页面：

> OpenPKG 项目使得 bash-4.3 的源代码 RPM 可作为当前发布的核心部分，用于各种 Unix 和 Linux 系统。
>
> Solaris 2.x 和 Solaris 7/8/9/10/11 的用户可以从 Unixpackages（订阅）或 OpenCSW 站点获得 bash-3.4 的预编译版本。Oracle 在 Solaris 10 中包含了 bash-3.2，在 Solaris 11 中包含了 bash-4.1。2016 年 9 月，以 OpenIndiana 发行的 Solaris/Illumos 包含了 bash-4.3。
>
> AIX 用户可以从 Groupe Bull 获得 bash-4.3 的预编译版本以及更早的版本，也可以从 Michael Perzl 的网站获得 bash-4.3 的源代码和二进制文件。IBM 使得 bash-4.2 和 bash-4.3 可以作为 GNU/Linux 应用程序的 AIX 工具箱的一部分，从而用于 AIX 5L、AIX 6.1、AIX 7.1。它们采用的都是 RPM 格式。你也可以从那儿获取用于 AIX 的 RPM。
>
> HP-UX 用户可以从 HP-UX 软件移植与存档中心（Software Porting and Archive Center）处获得 bash-4.3 的二进制文件与源代码。

1.16.4 参考

- OpenPKG 项目网站
- Solaris
  - UNIX packages 网站
  - OpenCSW 网站
  - Oracle 网站中的 Oracle Solaris 10 页面
  - Oracle 网站中的 Oracle Solaris 11 页面

    - OpenIndiana 网站
- AIX
    - Bull Freeware 网站
    - Michael Perzl 的网站
    - IBM 网站中的 IT Infrastructure 页面
- HP-UX 网站 Home>Categories>Shells 页面
- 1.11 节
- 1.12 节
- 附录 E

1.17 获取Windows版的bash

1.17.1 问题

你想要获取 Windows 系统下的 bash，或确认已安装的 bash 是否为最新版。

1.17.2 解决方案

使用 Cygwin 或 Windows 上的 Ubuntu，也可以使用虚拟机。又或者干脆不用 bash。

下载 Cygwin 并运行。按照提示选择要安装的包，bash 位于 shell 分类，默认是选中状态。安装好 Cygwin 后，你还得做一番配置。具体可参见用户手册。

对于 Windows 上的 Ubuntu，你需要使用 2016 年夏季版或更新版本的 Windows 10，然后按照安装说明操作，具体方法参见 1.17.3 节。

要想使用虚拟机，参见 15.4 节。

最后，尽管我们讨厌这么说，但也许正确的解决方案是使用 PowerShell 这样的原生工具。

1.17.3 讨论

01. Cygwin

Cygwin 在 Windows 下提供了一种具备 Linux 观感的类 Linux 环境。

以下描述取自 Cygwin 的官方站点。

Cygwin 是：

- 一个数量庞大的 GNU 及开源工具合集，在 Windows 上提供了与 Linux 发行版类似的功能；
- 一个提供了大量 POSIX API 功能的 DLL（cygwin1.dll）。

Cygwin 不是：

- 在 Windows 上运行原生 Linux 应用程序的方法。如果想要这么做，必须从源代码重新构建应用程序；
- 一种让原生 Windows 应用程序能够知晓 UNIX® 功能（如信号、伪终端等）的神奇方法。再次强调，如果想要利用 Cygwin 的功能，必须从源代码重新构建应用程序。

Cygwin DLL 可以在从 Windows Vista 起的所有 X86 32 位和 64 位的 Windows 商业发行版上工作。

> Cygwin 2.5.2 是支持 Windows XP 和 Server 2003 的最后一个版本。

Cygwin 是在 Windows 上运行的一种真正的类 Unix 环境。它的确是一款绝佳的工具，但有时似乎有点杀鸡用牛刀了。

02. Windows上的Ubuntu

在 Windows 上运行 Ubuntu 颇有意思，但除了其中包含 bash 外，其他方面与本书主题无关，因此我们就不细谈了。详细信息可参见 1.17.4 节中列出的参考资料。

简单来说：

- 启用 Developer Mode。
  - 搜索“Windows Features”。
  - 选择“Turn Windows features on or off”，启用“Windows Subsystem for Linux”。
    - 这可能需要重启！！！开玩笑的吧？！
- 打开 Command Prompt 并输入 bash。
  - 从 Windows 商店下载 Windows Subsystem for Linux。

03. 使用PowerShell或其他原生工具

面对可脚本化命令行工具的威力和灵活性，微软给出的答案就是 PowerShell，而且用其替代了 command.com 和 cmd.exe 批处理文件。除了作为 Windows 原生的 shell 脚本编程语言，PowerShell 已经超出了本书的讨论范围，因此不再多讲。

虽然与 Unix/Linux 工具比起来相形见绌，但旧式的 Windows shell 脚本语言其实比很多人所知的更为强大。它们也许适合于那些使用其他解决方案显得大材小用的简单任务。

如果想要一款功能强大、基于字符且拥有更为一致的 DOS/Windows 界面特色的图形用户界面命令行 shell，不妨看看 JP Software 网站。本书作者无一隶属这家公司，但其中一位作者长期以来对该公司的产品都很满意。

1.17.4 参考

- Cygwin 站点
- Windows 上的 Ubuntu：
  - Windows Subsystem for Linux documentation
  - “Microsoft and Canonical Partner to Bring Ubuntu to Windows 10”，Steven Vaughan-Nichols 撰写
  - “Ubuntu on Windows—The Ubuntu Userspace for Windows Developers”，Dustin Kirkland 撰写
  - “Developers Can Run Bash Shell and User-Mode Ubuntu Linux Binaries on Windows 10”，Scott Hanselman 撰写

    - “Announcing Windows 10 Insider Preview Build 14316”，Gabe Aul 撰写
    - “alwsl Project Lets You Install Arch Linux in the Windows Subsystem for Linux”，Marius Nestor 撰写
    - “How to Install and Use the Linux Bash Shell on Windows 10”，Chris Hoffman 撰写
- 维基百科（PowerShell）
- JP Software 网站
- 1.12 节
- 1.18 节
- 15.4 节

1.18 不获取bash的情况下使用bash

1.18.1 问题

你想要尝试一下 shell 或 shell 脚本，但又没有时间或资源来搭建或购买系统。

又或者现在你想读一则富有禅宗意味的实例。

1.18.2 解决方案

从 Polarhome 网站获得一个近乎免费的 shell 账户，只需要象征性地支付点一次性费用即可，或者更换其他厂商。

几乎所有的 Linux 和 BSD 发行版都有 LiveCD 或 LiveDVD 镜像，基本上都可以用作 LiveUSB，你可以下载并用其引导系统来做实验。如果你在考虑切换操作系统，这也是个不错的主意，可以借此验证所有的硬件是否都被支持，工作是否正常。棘手的地方可能是如何设置系统的 BIOS 或 UEFI，以便其能从 CD/DVD 或 USB 引导。将 ISO“烧制”到 U 盘在以前一直是个难题，不过如今针对特定的发行版已经有很多现成的工具，网站上也有详细的操作说明。

另外，也可以采用虚拟化解决方案，参见 15.4 节。

1.18.3 讨论

Polarhome 提供了多项免费服务以及基本免费的 shell 账户。根据其网站上的描述：

> 为推广具备 shell 支持的操作系统以及互联网服务，Polarhome 站点提供了非商业化的教育性支持，为所有的可用系统（目前包括各种 Linux 版本、MacOS X、OpenVMS、Solaris、OpenIndiana、AIX、QNX、IRIX、HP-UX、Tru64、SCO OpenServer、UnixWare、FreeBSD、OpenBSD、NetBSD、Dragon - Fly/BSD、MirBSD、Ultrix、Minix、GNU Hurd、Syllable 以及 OPENSTEP）提供了 shell 账户、开发环境、邮件及其他在线服务。

1.18.4 参考

- 免费 shell 账户清单
- Polarhome 网站
- 15.4 节

1.19 更多的bash文档

1.19.1 问题

你想要了解更多有关 bash 的信息，却不知从何开始。

1.19.2 解决方案

是的，本书就是一个很好的起点！由 O'Reilly 出版的与 bash 和 shell 编程相关的其他图书分别为由 Cameron Newham 所著的 *Learning the bash Shell, 3rd Edition* 以及由 Nelson H. F. Beebe 和 Arnold Robbins 合著的 *Classic Shell Scripting*。

遗憾的是，并非所有官方 bash 文档和支持文件都可以轻松地在线访问。你可以在 GNU 项目网站上找到 Bash Reference Manual，但其他

资料就不太容易获取了。本书的配套网站帮你完成了所有的相关工作，其中提供了官方的 bash 参考文件以及其他有用的在线资料，减少了你的麻烦。需要时可以从中查阅。

01. 官方文档

查看 bash FAQ 的官方文档，尤其要注意“H2) What kind of bash documentation is there?”一节。另外，强烈推荐官方的参考指南，详情如下。

Chet Ramey 的 bash 页面包含了大量非常有用的信息。另外，Chet（当前的 bash 维护者）还维护了下列内容。

README

bash 的自述文件。

NEWS

该文件简要列出了当前版本与上个版本之间值得注意的变动。

CHANGES

一份完整的 bash 变动历史。

INSTALL

安装方法。

NOTES

特定平台的配置与操作事项。

COMPAT

bash3 和 bash1 之间的一系列兼容性问题。

搜索最新的 bash 源代码和文档。

即便使用的是预先打包好的二进制文件，我们也强烈建议你下载源代码和文档（附录 B 中列出了所包含的示例与源代码的索引）。下面简要列出了包含在源代码存档文件（tarball）中 ./doc 目录下的文档。

FAQ

一组有关 bash 的常见问题及答案。

INTRO

bash 的简介。

article.ms

Chet 为 *The Linux Journal* 撰写的一篇文章。

bash.1

bash 的手册页。

bashbug.1

bashbug 的手册页。

builtins.1

取自 bash.1 的内建命令手册页。

bashref.texi

Bash Reference Manual。

bashref.info

makeinfo 处理过的 Bash Reference Manual。

rbash.1

受限 bash shell 的手册页。

readline.3

readline 手册页。

.ps 文件是上述文件的 PostScript 版本。.html 文件是手册页和参考手册的 HTML 版本。.0 文件是格式化过的手册页。.txt 文件是 groff -Tascii 的输出。

在文档的存档文件中，你可以找到以下文件的格式化版本。

bash-doc-4.4：

bash.0

bash 手册页（还包括 .pdf、.ps、.html）。

bashbug.0

bashbug 手册页。

bashref

GNU Bash Reference Manual（还包括 .pdf、.ps、.html、.dvi）。

builtins.0

内建命令手册页。

rbash.0

受限 bash shell 的手册页。

02. 其他文档

- Mendel Cooper 的“Advanced Bash-Scripting Guide”

- “Writing Shell Scripts”
- Mike G 的“BASH Programming - Introduction HOW-TO”
- Machtelt Garrels 的“Bash Guide for Beginners”
- Giles Orr 的“Bash Prompt HOWTO”
- 颇有点年头，但仍然有价值：“UNIX shell differences and how to change your shell”
- Apple 的“Shell Scripting Primer”

1.19.3 参考

- 附录 B
- Cameron Newham 所著的 *Learning the bash Shell, 3rd Edition*（O'Reilly 出版）
- Nelson H. F. Beebe 与 Arnold Robbins 合著的 *Classic Shell Scripting*（O'Reilly 出版）
- Bash Reference Manual
- 本书官网

第 2 章　标准输出

如果无法产生输出，那么软件也就没什么价值了，但长久以来，I/O 一直是难缠的计算领域之一。如果你是元老级用户，应该不会忘记当初运行一个程序的大部分功夫耗费在了设置程序的输入和输出上。如今有些问题已经不存在了，例如，再也不用让操作员将磁带挂载到磁带驱动器上（至少我们见过的桌面系统和笔记本计算机不用了）。但仍有很多麻烦摆在我们眼前。

问题之一就是有太多类型的输出。向屏幕写入不同于向文件写入，至少看起来肯定不一样。向文件写入也不同于向磁带或闪存写入。如果你想让一个程序的输出直接进入另一个程序，该怎么做呢？软件开发人员是否要针对各种输出设备编写代码，甚至包括尚未发明的设备？这显然是件麻烦事。用户是不是也得知道如何将想要运行的程序连接到不同种类的设备？这也不是什么好主意。

Unix 操作系统背后最重要的概念之一就是一切皆文件（有序的字节序列）。操作系统负责实现这套魔法。无论你要写入的目标是磁盘文件、终端、磁带设备、记忆棒，还是其他东西，程序只需要知道如何写入文件就够了，剩下的事情由操作系统搞定。这种方法很大程度上简化了前面提到的问题。

下一个问题很简单：写到哪个文件？程序怎么知道是该写入代表终端窗口的文件、磁盘文件还是其他种类的文件？不难，这种事情留给 shell 就行了。

在运行程序时，还需要将它连接到输出文件和输入文件（下一章会详细介绍）。这项任务是少不了的，但 shell 令其变得异常简单。考虑下面这条简单的命令：

```
dosomething < inputfile > outputfile
```

该命令从 *inputfile* 中读取输入，然后将输出发送给 *outputfile*。如果省略 `> outputfile`，则会输出到终端窗口。如果省略 `<inputfile`，那么该程序会从键盘接受输入。程序确实不

知道自己输出到了哪里，也不知道自己从哪里得到的输入。你可以利用 bash 的重定向功能将输出发送到想要的任何地方（也可以是另一个程序）。

这仅仅是个开始而已。本章将探讨产生输出的各种方法，以及 shell 如何将输出发送到其他位置。

2.1 输出到终端/终端窗口

2.1.1 问题

你想要用 shell 命令产生一些简单的输出。

2.1.2 解决方案

使用内建命令 echo。命令行中的所有参数都会打印到屏幕上。例如：

```
echo Please wait.
```

输出：

```
Please wait.
```

结果和在 bash 提示符（字符 $）后输入该命令相同：

```
$ echo Please wait.
Please wait.
$
```

2.1.3 讨论

echo 是最简单的 bash 命令之一。该命令可以将参数输出到屏幕上。但是有几点需要记住。首先，shell 负责解析 echo 的命令行参数（shell 对其他命令也是如此）。这意味着，在将参数交给 echo

前，shell 会完成所有的替换、通配符匹配等操作。其次，在解析参数时，参数之间的空白字符会被忽略。例如：

```
$ echo this    was     very    widely    spaced
this was very widely spaced
$
```

shell 对参数间的空白字符没有太多限制，这通常是一种不错的特性。但对于 echo 来说，就有点烦人了。（2.2 节将讨论如何保留输出中的空白字符，13.15 节将讨论如何修剪数据中的空白字符。）

2.1.4 参考

- help echo
- help printf
- 2.2 节
- 2.3 节
- 13.15 节
- 15.6 节
- 19.1 节
- A.11 节
- A.12 节

2.2 保留输出中的空白字符

2.2.1 问题

你想要保留输出中的空白字符。

2.2.2 解决方案

将字符放入引号中。在上一节的示例中加入引号就可以保留空白字符：

```
$ echo "this was    very    widely    spaced"
this    was    very    widely    spaced
```

```
$
```

或者：

```
$ echo 'this  was  very   widely   spaced'
this    was   very  widely  spaced
$
```

2.2.3 讨论

引号中的单词组成了 echo 命令的单个参数。该参数是一个字符串，shell 不会干涉字符串的内容。实际上可以用单引号（''）明确告诉 shell 不要干涉字符串。如果使用的是双引号（""），那么 shell 还是会执行一些替换操作（变量扩展、算术扩展、波浪号扩展以及命令替换），但上述示例不涉及这些操作，所以输出没有什么变化。如果不确定，那就用单引号。

2.2.4 参考

- help echo
- help printf
- 第 5 章
- 2.3 节
- 15.6 节
- 19.11 节
- A.11 节

2.3 在输出中加入更多格式控制

2.3.1 问题

你希望能够更多地控制输出的格式和位置。

2.3.2 解决方案

使用内建命令 printf。例如：

```
$ printf '%s = %d\n' Lines $LINES
Lines = 24
$
```

或者：

```
$ printf '%-10.10s = %4.2f\n' 'Gigahertz' 1.92735
Gigahertz   = 1.93
$
```

2.3.3 讨论

内建命令 printf 的行为和 C 语言中的同名库函数相似，其中第一个参数是格式控制字符串，之后的参数都根据格式规范（%）进行格式化。

% 和格式类型（本例为 s 或 f）之间的数字提供了额外的格式化细节。对于浮点类型（f），第一个数字（指示符 4.2 中的 4）是整个字段的宽度。第二个数字（2）是应该在小数点右侧打印出的数位量。注意，结果会按照四舍五入处理。

对于字符串，第一个数字是字段的最大宽度，第二个数字是要输出的字符数量。根据需要，字符串会被截断（长于 *max*）或用空白填充（不足 *min*）。如果指示符 *max* 和 *min* 相同，那么就可以确保字符串按照该长度输出。指示符左侧的负号表示字符串向左对齐（在字段宽度内）。如果不使用负号，则字符串向右对齐，因此：

```
$ printf '%10.10s = %4.2f\n' 'Gigahertz' 1.92735
  Gigahertz = 1.93
$
```

可以引用字符串参数，也可以不引用。如果需要保留字符串中嵌入的空白字符（本例的字符串中只有一个单词，不存在空格），或者需要转义字符串中的某些特殊字符（本例也未出现这些特殊字符），可以加上引号 [1]。最好养成将传给 printf 的字符串都放进引号中的习惯，这样一来，需要用到引号时就不会遗漏了。

[1]如果想要转义特殊字符，则需要使用双引号。——译者注

2.3.4 参考

- help printf
- Cameron Newham 所著的 *Learning the bash Shell, 3rd Edition* 的第 171 页，或者任何一本 C 语言参考书中的 printf 函数部分
- 15.6 节
- 19.11 节
- A.12 节

2.4 消除输出中的换行符

2.4.1 问题

你希望输出中不包含 echo 默认生成的换行符。

2.4.2 解决方案

使用 printf，做法很简单，去掉格式化字符串末尾的 \n 即可。

```
$ printf "%s %s" next prompt
next prompt$
```

如果是 echo，则使用 -n 选项：

```
$ echo -n prompt
prompt$
```

2.4.3 讨论

因为 printf 的格式字符串（第一个参数）末尾并没有换行符，所以命令行提示符（$）出现在了 printf 的输出之后。该特性在

shell 脚本中用处更大，你可能希望在形成一整行前由多条语句逐部分输出，或者在读取输入前显示用户提示。

换作 echo 命令（参见 15.6 节），消除换行符的方法有两种。首先，-n 选项能够抑制输出行尾的换行符。另外，echo 命令还可以处理多种具有特殊含义的转义序列（如表示换行符的 \n），这些转移序列与 C 语言字符串中的类似。要想使用它们，调用 echo 命令时必须加上 -e 选项。其中一种转义序列是 \c，它并不会输出什么字符，而是禁止在行尾输出换行符。因此，下面是第三种解决方案：

```
$ echo -e 'hi\c'
hi$
```

因为 printf 所提供的格式化功能既强大又灵活，而且该命令还是内建的，调用开销非常小（不像在其他 shell 或老版 bash 中，printf 是一个独立的可执行文件），所以本书中的很多示例都采用了 printf。

2.4.4 参考

- help echo
- help printf
- 第 3 章，尤其是 3.5 节
- 2.3 节
- 15.6 节
- 19.11 节
- A.11 节
- A.12 节

2.5 保存命令输出

2.5.1 问题

你想把命令输出保存在文件中。

2.5.2 解决方案

用 > 符号告诉 shell 将输出重定向至文件。例如：

```
$ echo fill it up
fill it up
$ echo fill it up > file.txt
$
```

我们来查看一下文件 file.txt 的内容，看看其中是否包含了命令的输出：

```
$ cat file.txt
fill it up
$
```

2.5.3 讨论

示例第一部分的第一行中出现的 echo 命令包含了 3 个要输出的参数。第二行用 > 将这些输出保存到文件 file.txt 中，这就是看不到 echo 输出的原因。

示例第二部分用 cat 命令显示文件内容。我们可以看出，文件中包含的正是 echo 本该输出的内容。

cat 命令得名自一个较长的单词 concatenation（拼接）。该命令会将出现在命令行上的文件的输出拼接在一起，如果你输入 cat file1 filetwo anotherfile morefiles，那么这些文件的内容会逐个发送到终端窗口。如果一个大文件被分成了两半，你也可以用 cat 将其恢复原样（也就是将两部分拼接起来），这只需将输出保存到另一个文件中：

```
cat first.half second.half > whole.file
```

所以说，前面那个简单的命令 cat file.txt 所做的事其实不过是拼接出了一个文件并将结果显示在屏幕上。尽管 cat 的能耐不小，但在本例中的主要用途就是在屏幕上显示文件内容。

2.5.4 参考

- man cat
- 17.23 节

2.6 将输出保存到其他文件

2.6.1 问题

你想要用重定向将输出保存到当前目录之外的其他位置。

2.6.2 解决方案

重定向输出时加上路径：

```
echo some more data > /tmp/echo.out
```

或者：

```
echo some more data > ../../over.here
```

2.6.3 讨论

出现在重定向符号（>）后的文件名其实就是路径名。如果没有任何限定部分，那么文件就会放置在当前目录中。

如果文件名以斜线（/）起始，那就是**绝对**路径名，此时文件会被放置在文件系统层次结构（**目录树**）中以根目录起始的指定位置（假设所有的中间目录都存在且你有权限进入）。我们使用了 /tmp，这是一个几乎所有 Unix 系统都存在且普遍可用的临时目录。在本例中，shell 会在 /tmp 目录中创建名为 echo.out 的文件。

在第二个例子中，我们使用了**相对**路径名 ../../over.here，其中的 .. 是一个指向父目录的特殊目录，存在于每个目录中。因此，每次引用 .. 就会在文件系统目录树中向上（往根的方向，并非惯常意义中

的沿着树“向上”）移动一级。这里的重点是，我们可以按照自己的意愿，将输出重定向到当前目录之外的某个文件中。

2.6.4 参考

- Cameron Newham 所著的 *Learning the bash Shell, 3rd Edition* 中的第 7~10 页，其中介绍了文件、目录和点号（. 和 ..）。

2.7 保存ls命令的输出

2.7.1 问题

你尝试用重定向保存 ls 命令的输出，但查看结果文件时发现格式和预想的不同。

2.7.2 解决方案

重定向输出时，使用 ls 命令的 -C 选项。

下面用 ls 命令显示目录内容：

```
$ ls
a.out cong.txt def.conf  file.txt  more.txt  zebra.list
$
```

但是，当用 > 将输出重定向到文件并显示该文件内容时，看到的是如下结果：

```
$ ls > /tmp/save.out
$ cat /tmp/save.out
a.out
cong.txt
def.conf
file.txt
more.txt
zebra.list
$
```

这次，我们加上 -C 选项：

```
$ ls -C > /tmp/save.out
$ cat /tmp/save.out
a.out cong.txt def.conf file.txt more.txt zebra.list
$
```

不进行重定向时，也可以使用 ls 的 -1 选项，得到的输出如下所示：

```
$ ls -1
a.out
Cong.txt
def.conf
file.txt
more.txt
zebra.list
$
```

这和最初使用重定向时的输出结果一样。

2.7.3 讨论

当你觉得自己已经理解重定向，试着将它用于简单的 ls 命令时，结果却不尽正确。这是怎么回事？

shell 的重定向功能意在对所有程序保持透明，因此程序无须编写特定代码来让自身的输出能够被重定向。当你用 > 将输出发送到别处时，shell 负责完成相关工作。但是，我们可以在程序中加入代码，判断输出何时被发往终端（参见 man isatty）。然后，程序就能够针对不同情况进行处理，这也正是 ls 所做的。

ls 的作者认为，如果要向屏幕输出，那么用户可能希望按列输出（-C 选项），毕竟屏幕空间有限，寸土寸金。但作者同时也认为，如果重定向到文件，那么用户会希望一行显示一个文件名（-1 选项），这样一来，更容易实现一些有意思的事情（其他处理）。

2.7.4 参考

- man ls
- man isatty
- 2.6 节

2.8 将输出和错误消息发送到不同文件

2.8.1 问题

你希望获得程序的输出，但不想输出被出现的错误消息弄乱。你要保存的错误消息混杂在程序输出中，不容易找出。

2.8.2 解决方案

将输出和错误消息重定向到不同文件：

```
myprogram 1> messages.out 2> message.err
```

或者采用更常见的方法：

```
myprogram > messages.out 2> message.err
```

2.8.3 讨论

这个示例展示了 shell 创建的两个输出文件。第一个是 messages.out，假想程序 myprogram 的所有输出都会重定向到该文件。myprogram 的所有错误消息都会重定向到 message.err。

在 1> 和 2> 中，数字表示文件 **描述符**。1 代表标准输出（STDOUT），2 代表标准错误（STDERR）。起始编号 0 对应标准输入（STDIN）。如果不指定数字，则假定为 STDOUT。有关文件描述符的更多信息及 STDOUT 和 STDERR 之间的区别，参见 2.19 节。

2.8.4 参考

- 2.6 节

- 2.13 节
- 2.19 节

2.9 将输出和错误消息发送到同一文件

2.9.1 问题

利用重定向，我们可以将输出或错误消息保存到单独的文件中，但如何将两者送往同一文件呢？

2.9.2 解决方案

用 shell 语法将标准错误消息重定向到和标准输出相同的地方。

首选：

```
both >& outfile
```

或者：

```
both &> outfile
```

又或者老式且略烦琐（但可移植性更好）的写法：

```
both > outfile 2>&1
```

其中，both 是准备向 STDERR 和 STDOUT 生成输出的（假想）程序。

2.9.3 讨论

&> 和 >& 只是将 STDOUT 和 STDERR 发送到相同地方（这正是我们想做的）的便捷写法。

在第三个示例中，1 用作重定向的目标，但是 >& 将 1 解释为文件描述符。实际上，2>&1 是一个实体（其中不允许出现空格），表示标准错误（2）会被重定向（>）到随后的文件描述符（&）1。2>& 必须作为整体出现，不能夹杂空格；否则，2 就成了另一个参数，而 & 代表与其表面完全不同的含义（与在后台运行命令有关）。

不妨认为所有的重定向操作符都带有一个前导数字（如 2>），而 >（标准输出的文件描述符）的默认数字为 1。

尽管可读性会略差，你也可以按照其他顺序进行重定向，将标准输出的指向重定向到与标准错误相同的地方（实际上，使用管道就必须这么做，参见 2.15 节）：

```
both 2> outfile 1>&2
```

1 代表标准输出，2 代表标准错误。我们本可以将最后一个重定向写作 >&2，因为对于 > 而言，1 是默认的，但是我们发现重定向文件描述符时，明明白白地写出数字，可读性会更好。

> 注意输出文件的内容顺序。有时候，错误消息在文件中出现的时间可能早于在屏幕上出现的时间。这与标准错误的无缓冲性质有关，当写入文件而不是屏幕时，这种现象会更加明显。

2.9.4 参考

- 2.6 节
- 2.13 节

2.10 追加输出

2.10.1 问题

每次重定向输出，都会产生一个全新的输出文件。如果想要两次（或三次、四次……）重定向输出，同时又不想破坏之前的输出，该怎么办呢？

2.10.2 解决方案

在 bash 的重定向符号中，双大于号（>>）表示**追加输出**：

```
$ ls > /tmp/ls.out
$ cd ../elsewhere
$ ls >> /tmp/ls.out
$ cd ../anotherdir
$ ls >> /tmp/ls.out
$
```

2.10.3 讨论

如果存在同名文件，第一行中的重定向会将其截断，并将 `ls` 命令的输出保存在这个已被清空的文件中。

后两次调用 `ls` 时使用了双大于号（>>），表示向输出文件中追加内容，而不是覆盖其原有内容。

如果想要同时重定向错误消息（STDERR），可以将 STDERR 的重定向放在后面，如下所示：

```
ls >> /tmp/ls.out 2>&1
```

在 bash 4 中，你可以将这两个重定向合二为一：

```
ls &>> /tmp/ls.out
```

该命令会重定向 STDERR 和 STDOUT，并将两者追加到指定文件中。记住，& 符号必须先出现，且这 3 个字符之间不能有空格。

2.10.4 参考

- 2.6 节
- 2.13 节

2.11 仅使用文件的起始或结尾部分

2.11.1 问题

你只需要显示或使用文件的起始或结尾部分。

2.11.2 解决方案

使用 `head` 或 `tail` 命令。默认情况下，`head` 输出指定文件的前 10 行，`tail` 输出最后 10 行。如果指定多个文件，则依次输出各文件中相应的行。可以通过 *`-number`* 选项（如 `-5`）修改默认行数。`tail` 还有 `-f` 和 `-F` 选项，这两个选项能够跟踪文件末尾的写入，2.12 节还会介绍另一个有意思的加号（+）选项。

2.11.3 讨论

`head`、`tail`、`cat`、`grep`、`sort`、`cut` 和 `uniq` 都属于最常用的 Unix 文本处理工具。如果尚不熟悉这些工具的用法，那么你很快就会好奇自己是如何在没有它们的情况下撑过来的。

2.11.4 参考

- 2.12 节
- 7.1 节
- 8.1 节
- 8.4 节
- 8.5 节
- 17.23 节

2.12 跳过文件标题

2.12.1 问题

文件中有标题行，你希望将其跳过，只处理数据。

2.12.2 解决方案

使用带有特殊选项的 tail 命令。例如，跳过文件的第一行：

```
$ tail -n +2 lines
Line 2
Line 3
Line 4
Line 5
$
```

2.12.3 讨论

tail 命令的选项 -n *number*（或者 -*number*）可以指定相对于文件末尾的行偏移。因此，tail -n 10 *file* 会显示 *file* 的最后 10 行，这也是不指定任何选项时的默认处理方式。如果数字以加号（+）开头，则表示相对于文件起始的行偏移。tain -n +1 *file* 会显示整个文件，tain -n +2 *file* 则是跳过第一行，剩下的以此类推。

2.12.4 参考

- man tail
- 13.12 节

2.13 丢弃输出

2.13.1 问题

你有时不想将输出保存到文件中，实际上，有时甚至不想看到输出。

2.13.2 解决方案

将输出重定向到 /dev/null，如下所示：

```
find / -name myfile -print 2> /dev/null
```

或者：

```
noisy > /dev/null 2>&1
```

2.13.3 讨论

你可以将不想要的输出重定向到文件，然后再将其删除。但还有一个更简单的方法。Unix 和 Linux 系统都存在一个特殊设备，该设备并非真实的硬件，而仅仅是一个**位桶**（bit bucket），我们可以将不需要的数据都扔进去。它就是 /dev/null，非常适用于此类场景。写入其中的数据会被直接丢弃，不会占用磁盘空间。重定向很容易做到这一点。

在第一个示例中，只有发往标准错误的输出被丢弃了。在第二个示例中，标准输出和标准错误均被丢弃。

在极少数情况下，/dev 可能会位于只读文件系统（如某些信息安全装置）中，此时你可以采用第一种方法：将输出重定向到文件，然后再删除该文件。

2.13.4 参考

- 2.6 节

2.14 保存或分组多个命令的输出

2.14.1 问题

你想用重定向获得输出，但是在一行中输入了多个命令：

```
pwd; ls; cd ../elsewhere; pwd; ls > /tmp/all.out
```

末尾的重定向仅应用于最后那个 `ls` 命令。其他命令的输出依旧出现在屏幕上（并未被重定向）。

2.14.2 解决方案

使用花括号（{}）将这些命令组合在一起，然后将重定向应用于分组中所有命令的输出。例如：

```
{ pwd; ls; cd ../elsewhere; pwd; ls; } > /tmp/all.out
```

这里有两个不易察觉的细微之处。花括号实际上是保留字，因此两侧必须有空白字符。另外，闭合花括号之前的拖尾分号也是不能少的。

或者，你也可以用括号（()）告诉 bash 在子 shell 中运行这些命令，然后重定向整个子 shell 的输出。例如：

```
(pwd; ls; cd ../elsewhere; pwd; ls) > /tmp/all.out
```

2.14.3 讨论

尽管这两种解决方案看起来差不多，但存在两处重要的区别。第一处是语法上的，第二处是语义上的。从语法上来看，花括号两侧需要有空白字符，命令列表中的最后一个命令必须以分号结尾。如果使用括号，那就不要求这些了。更大的区别在于语义方面，也就是这些构件所代表的含义。花括号只是一种组合多个命令的方式而已，更像是重定向的便捷写法，这样我们就不用单独重定向各个命令了。而出现在括号中的命令是在 shell 的另一个实例中运行，也就是当前 shell 的子 shell。

子 shell 几乎复刻了当前 shell 的环境，包括 $PATH 在内的变量都是一模一样的，但对陷阱的处理有所不同（有关陷阱的更多信息，参见 10.6 节）。采用子 shell 方法的重大不同在于：因为 cd 命令是在子 shell 中执行的，所以退出子 shell 后，父 shell 的当前目录仍保持原样，shell 变量也不会发生变化。

如果使用花括号分组命令，最后你会位于一个全新的目录中（这个示例为 ../elsewhere）。你所做的其他改动（如变量赋值）也会应用于当前 shell。虽然两种方法会获得相同的输出，但最终所在的目录位置大不相同。

花括号有一种值得注意的用法，即能够形成更简洁的分支语句块（参见 6.2 节）。你可以缩短下列语句：

```
if [ $result = 1 ]; then
    echo "Result is 1; excellent."
    exit 0
else
    echo "Uh-oh, ummm, RUN AWAY! "
    exit 120
fi
```

精简成：

```
[ $result = 1 ] \
  && { echo "Result is 1; excellent." ; exit 0; } \
  || { echo "Uh-oh, ummm, RUN AWAY! " ; exit 120; }
```

采用哪种写法取决于你的个人风格以及对可读性的理解，但我们推荐第一种形式，这种写法更清晰，适合更大范围的受众。

2.14.4 参考

- 6.2 节
- 10.6 节
- 15.11 节
- 19.5 节
- 19.8 节
- A.6 节，从中了解 $BASH_SUBSHELL

2.15 将输出作为输入，连接两个程序

2.15.1 问题

你希望从一个程序中获得输出，再将其作为另一个程序的输入。

2.15.2 解决方案

可以将第一个程序的输出重定向到一个临时文件，然后将该文件作为另一个程序的输入。例如：

```
$ cat one.file another.file > /tmp/cat.out
$ sort < /tmp/cat.out
...
$ rm /tmp/cat.out
$
```

也可以将这些操作合为一步，利用管道符号（|）将输出直接发送到下一个程序。例如：

```
cat one.file another.file | sort
```

还可以用多个管道将一系列命令连接在一起：

```
cat my* | tr 'a-z' 'A-Z' | sort | uniq | awk -f transform.awk | wc
```

2.15.3 讨论

使用管道符号意味着不用再创建临时文件，事后再将其删除。

像 sort 这种程序，既能从标准输入中获取输入（通过 < 符号进行重定向），也能从作为参数的文件中获取输入。因此，可以按以下方式操作：

```
sort /tmp/cat.out
```

不用再将输入重定向到 sort：

```
sort < /tmp/cat.out
```

这种行为（如果提供了文件名参数，就使用文件作为输入；否则，使用标准输入）是 Unix/Linux 的典型特征，也是一种应该遵循的实用模型，以便多个命令可以通过管道机制彼此相连。这种程序称为**过滤器**，如果照此方式编写程序和 shell 脚本，则更有助于你个人以及你的同事。

尽情感叹简洁有力的管道机制吧。你甚至可以将管道视为一种初级的并行处理机制。你可以让两个命令（程序）并行运行并共享数据：一个的输出作为另一个的输入。二者不必按顺序运行（一个结束，另一个接着开始），只要第一个命令产生可用数据，第二个命令立刻就可以开始处理。

但要注意，按照这种方式运行的多个命令（通过管道相连）分别在多个独立的进程中运行。尽管这一细微之处经常被忽视，有时所带来的影响却不容小视。19.8 节会对此展开讨论。

另外思考命令 `svn -v log | less`。如果 `less` 在 Subversion 结束数据发送前退出，就会出现错误 `svn: Write error: Broken pipe`。尽管看起来不怎么美观，但也没什么坏处。通过管道将大量数据传给 `less` 等命令时，总会出现这种事：一旦发现了要找的东西，就会想要退出，哪怕是管道中还有更多数据。

2.15.4 参考

- 3.1 节
- 19.8 节

2.16 将输出作为输入，同时保留其副本

2.16.1 问题

你想要调试一个比较长的管道化 I/O 序列，例如：

```
cat my* | tr 'a-z' 'A-Z' | uniq | awk -f transform.awk | wc
```

在不中断管道的情况下，该如何查看 `uniq` 和 `awk` 之间发生了什么？

2.16.2 解决方案

解决方案是使用水管工在维修管线时会用到的 T 形接头。对于 bash 来说，这意味着用 tee 命令将输出分成两个一模一样的流，一个写入文件，另一个写入标准输出，以便继续沿着管道发送数据。

对于上述示例，我们在 uniq 和 awk 之间插入 tee 命令：

```
... uniq | tee /tmp/x.x | awk -f transform.awk ...
```

2.16.3 讨论

tee 命令将输出写入其参数所指定的文件，同时还将相同的输出写入标准输出。在上述示例中，它将输出同时发送到 /tmp/x.x 和 awk，后者通过管道与 tee 的输出连接在一起。

不用关心示例中的这条命令的各个部分都是做什么的，我们只是演示 tee 命令在命令序列中的用法。

降低点儿难度，来看一个更简单的命令行。假设你想要将某个长期运行的命令的输出保存起来，以备随后参考，同时还想在屏幕上看到这些输出。思考下面的命令：

```
find / -name '*.c' -print | less
```

该命令会查找出大量的 C 程序源代码文件，可能需要好几屏的输出。使用 more 或 less 可以一次只查看部分输出，但命令结束后就无法回看之前的输出，除非再次运行命令。当然，你可以将输出保存到文件：

```
find / -name '*.c' -print > /tmp/all.my.sources
```

但这样的话，只有等到命令结束后才能看到文件内容。（好了，我们知道还有 tail -f，但现在说这个就跑题了。）tee 命令可以用来代替重定向标准输出的做法：

```
find / -name '*.c' -print | tee /tmp/all.my.sources
```

在这个例子中，因为 tee 的输出并未重定向到别处，因此会显示在屏幕上。但是输出的副本也会发送给指定的文件（如 cat /tmp/all.my.sources），以备后用。

还要注意，在这几个例子中，我们压根没有重定向过标准错误。这意味着错误（如来自 find 命令）会显示在屏幕上，但不会出现在 tee 指定的文件中。我们可以在 find 命令中加入 2>&1：

```
find / -name '*.c' -print 2>&1 | tee /tmp/all.my.sources
```

这样就可以将错误输出一并存入 tee 指定的文件。错误输出与普通输出之间未必泾渭分明，但的确能被捕获到。

2.16.4 参考

- man tee
- 18.5 节
- 19.13 节

2.17 以输出为参数连接两个程序

2.17.1 问题

如果想要接入管道的程序并不适用于管道，该怎么办？例如，你可以用 rm 命令删除文件，将待删除的文件指定为命令参数即可：

```
rm my.java your.c their.*
```

但是，rm 并不会从标准输入中读取参数，因此不能按以下方式这么做：

```
find . -name '*.c' | rm
```

rm 只能以命令行参数的形式获取文件名，那该如何将先前运行过的命令（如 echo 或 ls）的输出放入命令行呢？

2.17.2 解决方案

使用 bash 的命令替换特性：

```
rm $(find . -name '*.class')
```

也可以使用 xargs 命令，参见 15.13 中的讨论。

2.17.3 讨论

出现在 $() 中的命令是在子 shell 中运行的。$() 会被替换成所包含命令的输出。换行符会将输出变成多个命令行参数 [2]，尽管这种行为往往很有用，但有时也会产生出人意料的结果。

[2]详见 Bash Reference Manual 的 3.5.4 节。

早期的 shell 语法没有使用 $()，而是将命令放进反引号（``）。在语法上，$() 优于老式的 ``，因为前者更利于嵌套，自然也更容易阅读。但你可能更多看到的是 ``，尤其是在有点年代的 shell 脚本中，或是从早先的 Bourne shell/C shell 成长起来的人们那里。

在这个例子中，find 命令的输出（通常是一系列文件名）成了 rm 命令的参数。

> 做这种操作时要格外小心，因为 rm 可是不讲情面的。如果 find 找到的文件比预想的多，rm 也会将其一并删除，无可挽回。这可不是 Windows，你无法从回收站中恢复已删除文件。可以用 rm -i 降低风险，选项 -i 会提醒你确认每次删除。对于小批量文件来说，这么做没毛病，但如果文件数量众多，那整个过程可就冗长不堪了。
>
> 在 bash 中，更安全地使用该机制的一种方法是先运行其中的命令，看看结果是否符合预期，如果没问题，再将该命令放入 $()。例如：

```
$ find . -name '*.class'
First.class
```

```
Other.class
$ rm $(find . -name '*.class')
$
```

后文会介绍如何用 !! 来确保更加万无一失，而不是重新输入 find 命令（参见 18.2 节）。

2.17.4 参考

- 9.8 节
- 18.2 节
- 15.13 节

2.18 在一行中多次重定向

2.18.1 问题

你想将输出重定向到多个地方。

2.18.2 解决方案

用带有文件描述符编号的重定向打开所有要使用的文件。例如：

```
divert 3> file.three 4> file.four 5> file.five 6> else.where
```

其中，divert 可能是包含各种命令的 shell 脚本，你希望将其输出发送到不同地方。例如，divert 可能包含 echo option $OPTSTR >&5。也就是说，shell 脚本 divert 可以将其输出定向到不同的描述符，调用程序（invoking program）可以将描述符指向不同的目标。

与此类似，如果 divert 是一个 C 程序的可执行文件，则无须任何 open() 调用就可以向文件描述符 3、4、5、6 写入。

2.18.3 讨论

2.8 节介绍过，每个文件描述符都由一个数字（从 0 开始）代表：标准输入是 0，标准输出是 1，标准错误是 2。如果不指定数字，则假定为 1。这意味着可以用略显啰唆的写法 1>（而不是简单的 >）跟上文件名来重定向标准输出，不过其实没这个必要，便捷写法 > 就挺好。可以在 shell 中打开任意数量的文件描述符，令其指向各种文件，这样一来，随后在命令行上调用的程序就不用再费事了，直接就可以使用这些已打开的文件描述符。

我们不建议这么做，因为这种技术不仅脆弱，而且具有不必要的复杂性，不过倒是挺有意思的。

2.18.4 参考

- 2.6 节
- 2.8 节
- 2.13 节

2.19 重定向不起作用时保存输出

2.19.1 问题

你使用了 >，但是有些（或全部）输出仍旧出现在屏幕上。

例如，编译器产生了下列错误消息：

```
$ gcc bad.c
bad.c: In function `main':
bad.c:3: error: `bad' undeclared (first use in this function)
bad.c:3: error: (Each undeclared identifier is reported only once
bad.c:3: error: for each function it appears in.)
bad.c:3: error: parse error before "c"
$
```

因为想要获取这些消息，所以你尝试重定向输出：

```
$ gcc bad.c > save.it
bad.c: In function `main':
bad.c:3: error: `bad' undeclared (first use in this function)
```

```
bad.c:3: error: (Each undeclared identifier is reported only once
bad.c:3: error: for each function it appears in.)
bad.c:3: error: parse error before "c"
$
```

但是，看起来重定向好像没起什么作用。实际上，如果检查重定向文件，你会发现这个文件是空的（长度为 0）：

```
$ ls -l save.it
-rw-r--r-- 1 albing users 0 2005-11-13 15:30 save.it
$ cat save.it
$
```

2.19.2 解决方案

重定向错误输出，如下所示：

```
gcc bad.c 2> save.it
```

现在，save.it 的内容就是你先前看到的错误信息。

2.19.3 讨论

这是怎么回事？Unix 和 Linux 中的每个进程通常一开始都有 3 个已打开的文件描述符：一个用于**输入**（标准输入 STDIN），一个用于**输出**（标准输出 STDOUT），一个用于**错误消息**（标准错误 STDERR）。至于是否遵循这种约定，将错误消息写入标准错误，将正常输出写入标准输出，那真的就得看程序员了，因此，无法确保所有的错误消息都会进入标准错误。但大多数历史较久的实用工具很好地贯彻了这种做法。这就是为什么编译器消息无法使用简单的 > 进行重定向。它重定向的是标准输出，而不是标准错误。

前一节提过，每个文件描述符都由一个数字（从 0 开始）表示。标准输入是 0，标准输出是 1，标准错误是 2。这意味着可以用略显啰唆的写法来重定向标准输出：1>（而不是简单的 >）跟上文件名，但其实没这个必要，便捷写法 > 就够了。要想重定向标准错误，可以使用 2>。

标准输出和标准错误之间有一处重要不同：前者是**缓冲式**的（buffered），后者是**非缓冲式**的（unbuffered）。非缓冲意味着每个字符都是单独写入，不会被收集在一起，然后再批量写入。也就是说，你可以立刻看到错误消息，发生故障时丢失此类消息的可能性较小，但由此带来的就是效率问题。这并不是说标准输出就不可靠，但在发生错误的情况下（例如，某个程序出乎意料地“挂掉”了），缓冲式输出未必能在程序停止执行前将错误消息呈现在屏幕上。这就是标准错误采用非缓冲式的原因：确保能够及时写入消息。相比之下，对于标准输出，只有缓冲区满了（或文件被关闭），才会写入输出。对于频繁用到的输出，这么做效率更高，但要报告错误时，效率并不是最重要的。

如果想要同时保存并查看输出，该怎么办？2.16 节中讨论过的 tee 命令此时就能派上用场了。

```
gcc bad.c 2>&1 | tee save.it
```

该命令将标准错误重定向到标准输出，然后通过管道将两者传给 tee 命令。tee 会将其输入写入文件（save.it）和自己的标准输出，因为 tee 并没有重定向标准输出，所以你也可以在屏幕上看到经由管道传来的输入信息。

这是重定向的一种特例，因为重定向的顺序通常很重要。比较以下两个命令：

```
somecmd >my.file 2>&1
somecmd 2>&1 >my.file
```

在第一个命令中，标准输出被重定向到文件 my.file，然后标准错误被重定向到和标准输出相同的地方。所有的输出都会出现在 my.file 中。

第二个命令可就不是这样了，其中标准错误被重定向到标准输出（此时标准输出指向的是屏幕），然后，标准输出被重定向到 my.file。因此，只有标准输出消息会出现在文件中，而错误消息仍旧会出现在屏幕上。

但是，这种顺序不适用于管道。你不能将第二个重定向放在管道符号后面，因为管道之后出现的是下一个命令。因此，按照以下方式写入时：

```
somecmd 2>&1 | othercmd
```

bash 对此做了特殊处理，它可以识别出标准输出连接到了管道 [3]。因此，当你写出 2>&1 时，bash 会假定你希望将标准错误也连入管道，尽管 2>&1 的正常处理方式并非如此。

[3]命令行中的管道会先于重定向进行处理。详见 Bash Reference Manual 的 3.2.2 节。——译者注

这种做法（包括一般的管道语法）带来的另一个结果是，我们无法只将标准错误（而非标准输出）传入其他命令，除非事先交换文件描述符（参见 2.20 节）。

bash 4.*x* 版本中的一种便捷语法可以同时将标准输出和标准错误连入管道。要想将 somecmd 的两种输出流全都传给 othercmd，可以使用|&：

```
somecmd |& othercmd
```

2.19.4 参考

- 2.17 节
- 2.20 节

2.20 交换STDERR和STDOUT

2.20.1 问题

你需要交换 STDERR 和 STDOUT，这样就可以将 STDOUT 发送到日志文件，然后用 tee 命令将 STDERR 发送到屏幕和另一个文件。但是，管道仅适用于 STDOUT。

2.20.2 解决方案

在进入管道前用第三个文件描述符交换 STDERR 和 STDOUT：

```
./myscript 3>&1 1>stdout.logfile 2>&3- | tee -a stderr.logfile
```

2.20.3 讨论

每次重定向文件描述符时，就会将打开的描述符复制到另一个描述符。这样我们就得到了一种交换描述符的方法，这跟在程序中交换两个值差不多：借助一个临时的第三方位置。将 A 复制到 C，将 B 复制到 A，再将 C 复制到 B，这样就交换了 A 和 B 的值。对于文件描述符，操作如下：

```
./myscript 3>&1 1>&2 2>&3
```

将语法 3>&1 读作“使文件描述符 3 指向与标准输出文件描述符 1 相同的值”。这里是将文件描述符 1（STDOUT）复制到文件描述符 3，也就是我们的临时存放位置。然后再将文件描述符 2（STDERR）复制到 STDOUT，最后将文件描述符 3 复制到 STDERR。这样的结果就实现了交换 STDERR 和 STDOUT 文件描述符。[4]

[4]更准确地说，是交换了 STDERR 和 STDOUT 文件描述符的值。——译者注

目前一切还算不错。现在我们再略加改动。一旦复制了 STDOUT（复制到文件描述符 3），就可以放心将 STDOUT 重定向到用来捕获脚本或其他程序输出的日志文件。然后，我们再将保存在临时位置（文件描述符 3）的文件描述符复制到 STDERR。添加管道也没有问题，因为管道连接到（最初的）STDOUT。这样就得到了先前看到的解决方案：

```
./myscript 3>&1 1>stdout.logfile 2>&3- | tee -a stderr.logfile
```

注意 2>&3- 中结尾的 -。这么做是为了操作完成后关闭文件描述符 3。如此一来，程序就没有额外打开的文件描述符了。我们做到了自己的事情自己整理。

另外还用到了 tee 的 -a 选项，该选项表示追加，而不是替换。

2.20.4 参考

- Rob Flickenger 所著的 *Linux Server Hacks*（O'Reilly 出版），Hack #5,“n>&m: Swap STDOUT and STDERR”
- 2.19 节
- 10.1 节

2.21 避免意外覆盖文件

2.21.1 问题

你害怕错误地删除文件内容。输错文件名，发现将输出重定向到了原本打算保存的文件，这种事情太常见了。

2.21.2 解决方案

告诉 shell 更加谨慎：

```
set -o noclobber
```

如果你觉得完全用不着这么小心翼翼，可以关闭该选项：

```
set +o noclobber
```

2.21.3 讨论

noclobber 选项告诉 bash 在重定向输出时不要覆盖任何现有文件。如果重定向输出的文件尚不存在，一切照常进行，由 bash 创建该文件并将其打开以供输出之用。如果文件已经存在，则产生错误消息。

我们来实际演练一下。先关闭该选项，这只是为了让 shell 处于已知状态，不管特定系统先前是怎么配置的：

```
$ set +o noclobber                                    ❶
$ echo something > my.file                            ❷
$ echo some more > my.file
$ set -o noclobber                                    ❸
$ echo something > my.file
bash: my.file: cannot overwrite existing file
$ echo some more >> my.file1                          ❹
$
```

❶ 第一次，我们将输出重定向到 my.file，bash 会为我们创建该文件。

❷ 第二次，我们又将输出重定向到 my.file，bash 覆盖了该文件（它将文件截断为 0 字节并从头写入）。

❸ 设置 noclobber 选项，再次试图重定向到该文件时，我们得到了错误消息。

❹ 就像示例中显示的，我们可以（用 >>）向该文件中追加内容。

注意！noclobber 选项仅针对 shell 重定向输出时的文件覆盖行为。它并不能阻止其他程序覆盖文件（参见 14.13 节）：

```
$ echo useless data > some.file
$ echo important data > other.file
$ set -o noclobber
$ cp some.file other.file
$
```

这里并不会出现错误。现有文件被 cp 命令覆盖了。noclobber 选项在这种情况下不起作用。

如果你敲代码时又快又好，这看起来似乎不是一个重要的选项，但在其他实例中，我们会探究由正则表达式生成或以变量传递的文件名，它们均能用于输出重定向。在这种情况下，noclobber 选项就是一个重要的安全特性了，可以避免不必要的副作用（无论是一时糊涂还是有意为之）。

2.21.4 参考

- 2.22 节
- 14.13 节

2.22 有意覆盖文件

2.22.1 问题

你乐于设置 noclobber 选项，但在重定向输出时，偶尔也想覆盖文件。能不能有那么一次可以有意忽略 bash 的好心？

2.22.2 解决方案

使用 >| 重定向输出。即便是设置了 noclobber，bash 也会忽略该选项，并覆盖文件。

思考以下示例：

```
$ echo something > my.file
$ set -o noclobber
$ echo some more >| my.file                          ❶
$ cat my.file
some more
$ echo once again > my.file                          ❷
bash: my.file: cannot overwrite existing file
$
```

- 注意，第二次 echo 并不会产生错误。
- 但在第三次 echo 中，我们并未用到 |，仅仅使用了 > 符号。shell 发出了警告，也并未覆盖现有文件。

2.22.3 讨论

noclobber 的使用并不会代替文件权限。不管有没有使用 >|，如果没有目录的写权限，那么就无法创建文件。与此类似，不管有没有使用 >|，你必须拥有文件的写权限才能覆盖现有文件。

为什么要用 | ？据 Chet 所说，“POSIX 指定了语法 >|，这还是从 ksh88 中选出来的。我也说不清楚 Korn[5] 为什么会选用这种写法。csh 用的是 >!”。为了帮助记忆，你可以将其视为一种强调。它的用法在英语中带有祈使语气，符合要求 bash 在必要时“无论如何”都要覆盖文件的意味。vi 和 ex 编辑器在其 write 命令中也用 ! 表达了相同含义（:w! *filename*）。如果没有 !，覆盖已有文件时，编辑器会发出抱怨。要是加上 !，就相当于告诉编辑器“做就行了！”。

[5]David Korn 是 ksh 的作者。——译者注

2.22.4 参考

- 2.21 节
- 14.13 节

第 3 章　标准输入

无论是要交给程序处理的数据，还是控制脚本的简单命令，都少不了输入和输出。程序要做的第一件事就是处理如同一阴一阳的“输入与输出”。

3.1　从文件获取输入

3.1.1　问题

你希望 shell 命令从文件中读取数据。

3.1.2　解决方案

用代表输入重定向的符号 < 从文件中读取数据：

```
wc < my.file
```

3.1.3　讨论

就像 > 可以将输出发送至文件，< 则可以从文件中获取输入。之所以选择这种形状的操作符号，原因在于它们可以从视觉上提示重定向的方向。你看出来了吗？（注意“箭头”）

很多 shell 命令可以接受一个或多个文件名作为参数，但如果没有给出文件名，命令就会从标准输入读取。使用这种命令时，可以采用 *command filename* 或者 *command < filename*，这两种形式的结果没什么区别。在这个例子中，wc 是这样，换作 cat 或其他命令，也是如此。

这看起来也许并不起眼，如果你之前接触过 DOS 命令行，一定也不会陌生，但这是 shell 脚本编程的一项重要特性（DOS 命令行也借鉴了），对 shell 的功能性和简单性必不可少。

3.1.4 参考

- 2.6 节

3.2 将数据与脚本存放在一起

3.2.1 问题

你需要获得脚本输入，但又不想用单独的文件。

3.2.2 解决方案

使用 <<（here-document）从命令行而非文件重定向输入文本。如果放在 shell 脚本中，则脚本文件可以同时包含数据与代码。

以下是名为 ext 的 shell 脚本示例：

```
$ cat ext
#
# 下面是here-document
#
grep $1 <<EOF
mike x.123
joe  x.234
sue  x.555
pete x.818
sara x.822
bill x.919
EOF
$
```

这个脚本可以用于简单的电话号码搜索：

```
$ ext bill
bill x.919
$
```

或者：

```
$ ext 555
sue x.555
$
```

3.2.3 讨论

grep 命令查找第一个参数是否在指定文件中出现，如果没有指定文件，那么它会在标准输入中查找。

grep 的典型用法如下所示：

```
grep somestring file.txt
```

或者：

```
grep myvar *.c
```

在脚本 ext 中，通过将待搜索的字符串称为 shell 脚本的参数（$1），我们实现了 grep 的参数化。通常我们认为 grep 就是在不同文件中搜索固定的字符串，但这里我们打算在同一批数据中搜索不同的内容。

可以将电话号码保存在文件中，比如 phonenumbers.txt，然后在调用 grep 命令时指定文件名：

```
grep $1 phonenumbers.txt
```

但这样就需要两个文件（脚本文件和数据文件），随之而来的问题是：把这两个文件放哪里？如何确保两者集中在一起？

因此，我们不再提供一个或多个用于搜索的文件名，而是设置 here-document，告诉 shell 将标准输入重定向（临时）到此处。

<< 语法表示我们想创建一个临时输入源，EOF 是一个任意的字符串（你想用什么都行），用作临时输入的终止符。它并不属于输入的一部分，只是作为标记告诉输入在哪里结束。正常的 shell 脚本（如果有的话）会在该标记之后继续。

你可以给 grep 命令加入 -i 选项，以示搜索时不区分大小写。这样便可以使用 grep -i $1<<EOF同时搜索“Bill”或“bill”。

3.2.4 参考

- man grep
- 3.3 节
- 3.4 节

3.3 避免here-document中的怪异行为

3.3.1 问题

here-document 在使用时可能会出现一些怪异的行为。你想用上一节介绍的方法来保存一份简单的捐赠人列表，因此创建了一个名为 donors 的文件，如下所示：

```
$ cat donors
#
# 简单地查找慷慨的捐赠人
#
grep $1 <<EOF
# 捐赠人及其捐赠额
pete $100
joe  $200
sam  $ 25
bill $ 9
EOF
$
```

但是运行时出现了奇怪的输出：

```
$ ./donors bill
pete bill00
bill $  9
$ ./donors pete
pete pete00
$
```

3.3.2 解决方案

通过转义结尾标记中的任意或所有字符，关闭 here-document 内部的 shell 特性：

```
grep $1 <<'EOF'
pete $100
joe  $200
sam  $ 25
bill $ 9
EOF
```

3.3.3 讨论

尽管其中存在非常微妙的区别，但也可以将 <<EOF 替换成 <<\EOF 或 <<'EOF'，甚至是 <<E\OF，都没问题。尽管这并不是最优雅的语法，但足以告诉 bash 你希望区别处理 here-document 中的内容。

正常情况下（除非使用了转义语法），bash 手册页中是这样说的：“……here-document 的每一行都要执行参数扩展、命令替换以及算术扩展”。

因此，最初的 donors 脚本中所发生的事情是捐赠额被当作 shell 变量了。例如，$100 被视为 shell 变量 $1，随后跟着两个 0。这就是为什么我们在搜索“pete”时，得到的是 pete00；搜索“bill”时，得到的是 bill00。

如果我们转义了 EOF 的部分或全部字符，那么 bash 就知道不用执行扩展，这样就符合我们的预期行为了。

```
$ ./donors pete
pete $100
```

当然了，你可能**确实想要** bash 对 here-document 内部执行扩展操作，这种做法在某些情况下自有用处，不过眼前这个例子并非如此。我们发现，除非你真的要扩展数据，否则坚持转义终止符（就像

<<'EOF' 或 <<\EOF）是一种不错的做法，这样能够避免产生出乎意料的结果。

> 在结尾处的 EOF 标记中出现的拖尾空白字符（哪怕只是一个空格）会导致无法将其识别为结束标记。bash 会吞掉脚本的剩余部分，将其也视为输入并继续查找 EOF。所以，一定要确保 EOF 之后没有额外的空白字符（尤其是空格或制表符）。

3.3.4 参考

- 3.2 节
- 3.4 节

3.4 缩进here-document

3.4.1 问题

here-document 固然不错，但是它会弄乱 shell 脚本的格式。为了提高可读性，你希望能对其作缩进处理。

3.4.2 解决方案

使用 <<-，然后就可以在每行的开头用制表符（仅限制表符！）缩进 shell 脚本中的 here-document 部分：

```
$ cat myscript.sh
...
    grep $1 <<-'EOF'
        lots of data
        can go here
        it's indented with tabs
        to match the script's indenting
        but the leading tabs are
        discarded when read
        EOF
    ls
...
$
```

3.4.3 讨论

<< 之后的连字符（-）足以告诉 bash 忽略前导制表符。**但这仅适用于制表符**，其他空白字符是无法忽略的。注意，对于 EOF 或者选用的其他标记，这一点尤为重要。如果出现了空格，就无法将 EOF 识别为结束标记，here-document 的数据部分就会一直延续到文件结尾（吞掉剩余的脚本）。出于安全起见，你可以始终左对齐 EOF（或其他标记）。

> 就像结尾处的 EOF 标记中出现的任何拖尾空白字符都会导致其无法被识别为结束标记一样（参见 3.3 节中的警告部分），使用除制表符外的前导字符也会造成同样的后果。如果你的脚本使用空格或混合空格和制表符来进行缩进，可别在 here-document 中这么做。要么使用制表符，要么什么都不用。另外，小心有些文本编辑器会自动将制表符替换成空格。

3.4.4 参考

- 3.2 节
- 3.3 节

3.5 获取用户输入

3.5.1 问题

你需要获取用户输入。

3.5.2 解决方案

使用 read 语句：

```
read
```

或者：

```
read -p "answer me this " ANSWER
```

或者：

```
read -t 3 -p "answer quickly: " ANSWER
```

又或者：

```
read PRE MID POST
```

3.5.3 讨论

不带参数的 read 语句会读取用户输入并将其保存在 shell 变量 REPLY 中，这是 read 的最简形式。

如果希望 bash 在读取用户输入前先输出提示信息，可以使用 -p 选项。-p 之后的单词就是提示信息，如果想提供多个单词，可以将其引用起来。记住，要在提示信息结尾处加上标点符号或空格，因为光标会停在那里等待输入。

-t 选项可以设置超时值。指定秒数达到后，不管用户是否输入，read 语句都会返回。我们的示例同时用到了 -t 和 -p 选项，但你也可以单独使用 -t 选项。从 bash 4 开始，你甚至可以将超时值指定为小数，如 .25 或 3.5。如果读取超时，则退出状态码（$?）将大于 128。

如果你在 read 语句中提供了多个变量名，那么 read 会将输入解析为多个单词，依次赋给这些变量。如果用户输入的单词数较少，多出的变量就会被设为空（null）。如果用户输入的单词数多于变量数，则多出的单词会全部赋给最后那个变量。

3.5.4 参考

- help read，从中了解内建命令 read 的更多选项
- 3.8 节
- 6.11 节

- 13.6 节
- 14.12 节

3.6 获取yes或no

3.6.1 问题

你需要从用户处获得一个简单的 yes 或 no 作为输入，同时希望尽可能地对用户友好，尤其是不用区分大小写，如果用户直接按回车键，还能提供默认值。

3.6.2 解决方案

如果操作比较简单，可以使用例 3-1 中的独立函数。

例 3-1 ch03/func_choose

```
# 实例文件：func_choose

# 由用户选择并根据答案执行相应代码
# 调用方式：choose <default (y or n)> <prompt> <yes action> <no
action>
# 例如：choose "y" \
#       "Do you want to play a game?" \
#       /usr/games/GlobalThermonuclearWar \
#       'printf "%b" "See you later Professor Falkin.\n"' >&2
# 返回：无
function choose {

    local default="$1"
    local prompt="$2"
    local choice_yes="$3"
    local choice_no="$4"
    local answer

    read -p "$prompt" answer
    [ -z "$answer" ] && answer="$default"

    case "$answer" in
        [yY1] ) eval "$choice_yes"
            # error check
```

```
            ;;
        [nN0] ) eval "$choice_no"
            # error check
            ;;
        *     ) printf "%b" "Unexpected answer '$answer'!" >&2 ;;
    esac
} # 函数choose结束
```

如果操作比较复杂，可以使用例 3-2 中的函数，并在主代码中处理函数返回值。

例 3-2 ch03/func_choice.1

```
# 实例文件：func_choice.1

# 由用户选择并返回标准答案。默认值的处理方式以及接下来怎么做
# 取决于主代码中choice函数之后的if/then分支
# 调用方式：choice <prompt>
# 例如：choice "Do you want to play a game?"
# 返回：全局变量CHOICE
function choice {

    CHOICE=''
    local prompt="$*"
    local answer

    read -p "$prompt" answer
    case "$answer" in
        [yY1] ) CHOICE='y';;
        [nN0] ) CHOICE='n';;
        *     ) CHOICE="$answer";;
    esac
} # 函数choice结束
```

如果我们返回“0”表示 no，返回“1”表示 yes，表达式 if choice .. ; then 的用法则值得注意。我们将此作为练习留给你来实践。

例 3-3 中的代码调用了 choice 函数，提示并核实软件包的日期。假设已经设置 $THISPACKAGE，代码显示该日期，并要求用户核实。如果用户输入 y、Y，或者直接按下回车键，就接受所显示的日期。如果用户输入新的日期，代码则再次循环，继续核实日期（11.7 节中给出了该问题的另一种处理方法）。

例 3-3 ch03/func_choice.2

```
# 实例文件：func_choice.2

CHOICE=''
until [ "$CHOICE" = "y" ]; do
    printf "%b" "This package's date is $THISPACKAGE\n" >&2
    choice "Is that correct? [Y/,<New date>]: "
    if [ -z "$CHOICE" ]; then
        CHOICE='y'
    elif [ "$CHOICE" != "y" ]; then
        printf "%b" "Overriding $THISPACKAGE with $CHOICE\n"
        THISPACKAGE=$CHOICE
    fi
done

# 在此构建软件包
```

接下来看看处理 yes 或 no 的其他方法。仔细阅读提示信息，注意默认操作。在两种情况下，用户都可以直接按回车键，脚本将采用程序员安排好的默认处理方式：

```
# 如果用户输入'n'（不区分大小写）之外的任何内容，则会看到错误日志
choice "Do you want to look at the error logfile? [Y/n]: "
if [ "$CHOICE" != "n" ]; then
    less error.log
fi

# 如果用户输入'y'（不区分大小写）之外的任何内容，则看不到消息日志
choice "Do you want to look at the message logfile? [y/N]: "
if [ "$CHOICE" = "y" ]; then
    less message.log
fi
```

最后，例 3-4 中的函数会要求用户输入（有可能不存在）。

例 3-4 ch03/func_choice.3

```
# 实例文件：func_choice.3

choice "Enter your favorite color, if you have one: "
if [ -n "$CHOICE" ]; then
    printf "%b" "You chose: $CHOICE\n"
else
```

```
    printf "%b" "You do not have a favorite color.\n"
fi
```

3.6.3 讨论

在脚本编程中，要求用户做出选择往往是必不可少的。要想获取任意输入，参见 3.5 节。要想选择列表选项，参见 3.7 节。

如果用户可能做出的选择及相应处理代码非常直观，那么第一个独立函数更易于使用，但可能不够灵活。第二个函数在主代码中做了更多处理，因而灵活性更好。

注意，我们将用户提示发送到了 STDERR，这样一来，就算重定向出现在 STDOUT 的主脚本输出也不会弄乱提示信息。

3.6.4 参考

- 3.5 节
- 3.7 节
- 11.7 节

3.7 选择选项列表

3.7.1 问题

你需要为用户提供选项列表，但不希望用户输入不必要的内容。

3.7.2 解决方案

使用 bash 内建的 `select` 来生成菜单，随后用户可以输入选项数字进行选择（参见例 3-5）。

例 3-5 ch03/select_dir

```
# 实例文件：select_dir

```

```
directorylist="Finished $(for i in /*;do [ -d "$i" ] && echo $i;
done)"

PS3='Directory to process? ' # 设置有帮助的选择提示
until [ "$directory" == "Finished" ]; do

    printf "%b" "\a\n\nSelect a directory to process:\n" >&2
    select directory in $directorylist; do

        # 用户输入的数字被保存在$REPLY中，
        # 但是select返回的是用户选中的选项值
        if [ "$directory" == "Finished" ]; then
            echo "Finished processing directories."
            break
        elif [ -n "$directory" ]; then
            echo "You chose number $REPLY, processing
$directory..."
            # 在此进行相关处理
            break
        else
            echo "Invalid selection!"
        fi # 结束选项处理

    done # 结束目录选择处理
done # 如果用户选中Finished选项，则结束循环
```

3.7.3 讨论

select 语句能够轻松地在 STDERR 上为用户生成编号列表，以便用户从中做出选择。虽然按下 Ctrl-D 可以结束 select，空输入会再次输出菜单，但别忘了提供“退出”或“结束”选项。

用户所输入的选项编号保存在 $REPLY 中，选项值保存在 select 语句指定的变量中。

3.7.4 参考

- help select
- help read
- 3.6 节

3.8 提示输入密码

3.8.1 问题

你需要提示用户输入密码，但不希望在屏幕上回显出密码内容。

3.8.2 解决方案

用 read 命令读取用户输入，但要加上一个特殊选项来关闭回显：

```
read -s -p "password: " PASSWD
printf "%b" "\n"
```

3.8.3 讨论

-s 选项告诉 read 命令不要回显输入的字符（s 代表 silent），-p 选项指明下一个参数是提示信息，会在读取用户输入之前显示。

从用户那里读取到的输入行保存在变量 $PASSWD 中。

在 read 之后，我们用 printf 输出了一个换行符。这里的 printf 不能少，因为 read -s 会关闭字符回显。如果禁止了回显功能，当用户按下回车键时，就不会回显换行符，后续输出就会和提示信息出现在同一行。输出换行符会将光标带到下一行，这也正是我们想要的效果。将代码全都写在同一行也许更方便，这样可以避免干扰代码逻辑（另外还能避免将其复制到别处时出现遗漏）：

```
read -s -p "password: " PASSWD ; printf "%b" "\n"
```

如果要将密码读入环境变量，那就要小心了，因为密码在内存中是以明文形式存放的，有可能通过核心转储或 /proc/core（如果你所用的操作系统提供了 /proc/）访问到。在多进程环境中也是如此，其他进程也有可能读取到密码。可能的话，最好使用 SSH 证书。无论任何情况，明智的做法是假定系统中的 root 和其他可能的用户都能接触到密码并对其进行相应的处理。

有些陈旧的脚本也许会用 `stty -echo` 来禁止输入密码时的屏幕回显。这么做的问题在于如果脚本意外终止，那么回显仍会处于关闭状态。有经验的用户知道输入 `stty sane` 将其恢复，但这并非人人都懂。如果你的确需要使用这种方法，就设置好陷阱，以便在脚本终止时重新启用回显。具体参见 10.6 节。

3.8.4 参考

- `help read`
- 10.6 节
- 14.14 节
- 14.20 节
- 14.21 节
- 19.9 节

第 4 章　执行命令

bash（或者任何一种 shell）的主要目的是允许用户与计算机操作系统打交道，以便完成各种任务。这往往涉及执行程序，shell 因此要获取输入的命令，从中确定要运行的程序，然后将其启动。

我们来看看启动程序的基本机制，并探究 bash 提供的一些特性，这些特性可用于在前台或后台以串行或并行形式启动程序，指明程序是否成功等。

4.1　运行程序

4.1.1　问题

你需要在 Linux 或 Unix 系统上执行某个命令。

4.1.2　解决方案

使用 bash 并在提示符下输入命令名。

```
$ someprog
```

4.1.3　讨论

这看起来相当简单，某种程度上也的确如此，但是你根本无从知晓其中大量的幕后处理。bash 的基本操作就是载入并运行程序，理解这一点很重要。余下的都是些准备工作。确实，还有 shell 变量、循环控制语句、if/then/else 分支以及各种控制输入和输出的方法，但这些只不过是为程序运行锦上添花而已。

那么从哪里运行程序呢？

bash 使用名为 $PATH 的 shell 变量来定位可执行文件。$PATH 变量包含了一个目录列表。各个目录之间以冒号（:）分隔。bash 在这些目录中查找命令行上指定的可执行文件。目录的顺序很重要：bash 按照变量中所列出的目录顺序依次查找，选择所找到的第一个同名的可执行文件。

```
$ echo $PATH
/bin:/usr/bin:/usr/local/bin:.
$
```

这里显示出 $PATH 变量中包含了 4 个目录，其中最后一个目录就是一个点号（称为**点号目录**，或者干脆就叫**点号**），代表 Linux 或 Unix 文件系统的当前目录，也就是你当前所处的目录。例如，如果你将文件从别处复制到点号目录（如 cp /other/place/file .），其实就是将该文件复制到当前目录。将点号目录也包含在 $PATH 变量中是为了告诉 bash，不仅要在其他目录中查找命令，同时也别漏了当前目录（.）。

很多人觉得将点号放进 $PATH 是一种很大的安全风险：别有用心的人会欺骗你执行同名命令的恶意版本（如 ls）。如果将点号放在目录列表的最前面，那么他人的 ls 版本的优先级就高于正常的 ls，你可能不知不觉间就执行了前者。不信？来试试。

```
$ bash
$ cd
$ touch ls
$ chmod 755 ls
$ PATH=".:$PATH"
$ ls
$
```

主目录下的 ls 好像突然不工作了。没有看到任何输出。如果用 cd 切换到其他位置（如 cd /tmp），ls 又一切正常了。原因何在？这是因为主目录下有一个名为 ls 的空文件，你执行的其实就是这个 ls（什么都不做，本来就是个空文件），而不是位于 /bin 下正常的 ls 命令。我们演示这个例子时启动的是 bash 的新副本，退出该子 shell 就能摆脱这种混乱的局面，不过你可能想先删除这个伪造的 ls 命令。

```
$ cd
$ rm ls
$ exit
$
```

如果进入陌生的目录，同时又将 $PATH 设置为先搜索点号目录，是存在潜在危险的，现在你明白了吧？

如果将点号作为 $PATH 变量中的最后一个目录，至少不会那么容易被骗。当然，要是压根就不将它列入目录列表，那肯定更安全，而且你仍然可以执行当前目录中的命令，在前面加上点号和斜线即可，如下所示：

```
./myscript
```

究竟用哪种方法，就看你自己了。

绝不要将点号目录或可写目录放进 root 的 $PATH 变量中。有关该话题的更多信息，参见 14.9 节和 14.10 节。

在调用脚本前，别忘记给脚本文件设置执行权限。

```
chmod +x myscript
```

只需要设置一次权限，以后就可以像其他命令一样调用该脚本了。

一些 bash 用户的常见做法是创建个人 bin 目录，这类似于保存可执行文件的系统目录 /bin 和 /usr/bin。你可以将自己喜欢的 shell 脚本和其他定制或私有命令放入个人的 bin 目录中（如果创建于主目录，则路径为~/bin）。然后将该目录加入 $PATH，甚至可以放在目录列表的最前面（PATH=~/bin:$PATH）。这样一来，你既可以拥有自己偏好的定制工具，也不存在误执行陌生人命令的安全风险。

4.1.4 参考

- 第 16 章，其中包含了更多有关自定义环境的内容
- 1.5 节
- 14.9 节

- 14.10 节
- 16.11 节
- 19.1 节

4.2 依次执行多个命令

4.2.1 问题

你需要执行多个命令，但其中有些命令的运行时间较长，你又不想在输入下一个命令前等着前面的命令执行完毕。

4.2.2 解决方案

这个问题有 3 种解决方案，第一种非常平淡无奇：不停输入就行了。Linux 或 Unix 系统足够先进，能够在处理先前命令的同时允许你继续输入，因此将命令一个接一个敲进去即可。

另一种解决方案也挺简单：将这些命令写入文件，然后让 bash 执行该文件（简单的 shell 脚本）中的命令。假设我们想执行 3 个命令：long、medium、short，这些命令名反映了各自的执行时间。我们需要按顺序执行这些命令，但又不想在输入其他命令前等着 long 脚本运行完毕。此时可以用 shell 脚本（也就是批处理文件）来完成。以下是初级做法：

```
$ cat > simple.script
long
medium
short
^D                          # Ctrl-D，屏幕上不可见
$ bash ./simple.script
```

第三种是依次执行每个命令，这无疑也是最好的解决方案。如果你想运行各个程序，不管之前的程序是否成功运行，只需要用分号将其隔开：

```
long ; medium ; short
```

如果只想在上一个程序成功运行的情况下运行下一个程序，并且所有的程序都正确设置了退出码，那么可以用 `&&` 将其隔开：

```
long && medium && short
```

4.2.3 讨论

`cat` 示例是向文件输入文本的一种非常初级的方法：我们将命令输出重定向到文件 `simple.script`（有关输出重定向的更多内容，参见第 2 章）。最好还是使用一款真正的编辑器，但这不太方便在示例中展示。从现在开始，如果需要展现脚本，我们要么直接给出代码，不特地在命令行下输入；要么用 `cat` *`filename`* 这样的命令将原本出现在屏幕上的内容（而不是重定向输出到文件）放在示例中。

第三种解决方案的重点在于可以在 bash 命令行中放入多个命令。在第一种解决方案中，第一个命令退出后才会执行第二个命令，第二个命令退出后才会执行第三个命令，以此类推。在第二种解决方案中，只有第一个命令成功执行后第二个命令才会执行，第二个命令成功执行后第三个命令才会执行，以此类推，命令行中出现的命令均是如此。

4.2.4 参考

- 第 2 章中有关输出重定向的内容

4.3 同时执行多个命令

4.3.1 问题

你需要执行 3 个命令，但这些命令相互独立，不必等待前一个命令运行结束。

4.3.2 解决方案

你可以在命令末尾添加一个 & 符号，在后台运行该命令。这样一来，就能够连续快速地执行这 3 个命令，如下所示：

```
$ long &
[1] 4592
$ medium &
[2] 4593
$ short
$
```

或者，更好的做法是在单个命令行中完成所有操作。

```
$ long & medium & short
[1] 4592
[2] 4593
$
```

4.3.3 讨论

在“后台”（Linux 其实没有这么个地方）运行某个命令时，真正的意思是我们断开了键盘输入与命令之间的联系，shell 在显示命令行提示符并接受更多命令输入前不会再等着该命令完成。命令输出（除非我们采取明确的操作来改变这种行为）仍然会出现在屏幕上，因此，示例中 3 个命令的输出会在屏幕上交错出现。

方括号内有点奇怪的数字是作业号，随后是在后台启动的命令的进程 ID。在我们的示例中，作业 1（进程 4592）是 long 命令，作业 2（进程 4593）是 medium。

short 之后并未出现 & 符号，因此该命令不会在后台运行，bash 会等待其运行完毕才显示命令行提示符（$）。

作业号或进程 ID 可用于对作业实施有限的控制。例如，我们可以用 kill %1（因为作业号为 1）或者指定进程 ID（kill 4592）来“杀死”long 作业，二者能够实现相同的结果。

你也可以用作业号重新连接到后台作业。例如，可以用 fg %1 将 long 作业放回前台。如果后台只有一个作业在运行，甚至都不用指定作业号，只使用 fg 即可。

如果执行某个命令，但发觉其完成时间比预想的更长，那么可以用 Ctrl-Z 暂停该命令，返回到提示符下。接着输入 bg 来恢复作业，并在后台继续运行。这么做的效果相当于事前在命令尾部加上 & 符号。

4.4 了解命令是否成功运行

4.4.1 问题

你需要知道命令是否成功运行。

4.4.2 解决方案

如果编写命令或 shell 脚本的程序员遵循既定约定，那么 shell 变量 $? 会在命令失败时被设置为非 0 值。

```
$ somecommand
# it works...
$ echo $?
0
$ badcommand
# it fails...
$ echo $?
1
$
```

4.4.3 讨论

shell 变量 $? 中保存着命令的退出状态，其取值范围为 0~255。在编写 shell 脚本时，良好的做法是：如果一切正常，脚本退出时就返回 0；如果运行过程中出错，则返回非 0 值。我们推荐只使用 0~127 作为返回值，因为 shell 用 128+N 代表**被信号 N“杀死”**。另外，如果使用的值大于 255 或小于 0，则会出现值回绕。可以用 exit 语句（如 exit 1 或 exit 0）返回退出状态。但要注意，读取命令退出状态的机会只有一次。

```
$ badcommand
# 运行失败……
$ echo $?
1
$ echo $?
0
$
```

为什么第二个 echo 的结果是 0 ？它报告的其实是第一个 echo 命令的退出状态。第一次输入 echo $? 时，结果是 1，这是 badcommand 的返回值。但是该 echo 命令本身是成功执行的，因此最新的退出状态就是 0（表示成功）。因为检查退出状态的机会只有一次，所以很多 shell 脚本会立即将退出状态保存到其他 shell 变量中。

```
$ badcommand
# 运行失败……
$ STAT=$?
$ echo $STAT
1
$ echo $STAT
1
$
```

我们将退出状态值保存在变量 $STAT 中，以便随后检查。

尽管这个示例是在命令行中展示的，但是 $? 这种变量的真正用途是在脚本编写方面。如果在屏幕上观看命令运行，通常可以看出该命令是否正常。但在脚本中，命令运行时也许无人值守。

bash 的一大特点是，脚本语言与在终端窗口提示符下键入的命令完全相同。在编写脚本时，这使得检查语法和逻辑要容易得多。

退出状态多用于脚本和 if 语句中，根据所执行命令的成功与否采取不同的处理方式。下面这个示例比较简单，但我们后续还会讨论这个话题。

```
somecommand
...
if (( $? )) ; then echo failed ; else echo OK; fi
```

(( )) 对算术表达式进行求值；参见 6.1 节和 6.2 节。

我们不推荐使用负数。shell 可以接受负数，也不会报错，但结果并不如你所愿。

```
$ bash -c 'exit -2' ; echo $?
254

$ bash -c 'exit -200' ; echo $?
56
```

4.4.4 参考

- 4.5 节
- 4.8 节
- 6.1 节
- 6.2 节

4.5 仅当一个命令运行成功后才执行下一个命令

4.5.1 问题

你需要运行一些命令，但其中部分命令需要在其他命令运行成功的情况下执行。例如，你想要切换到临时目录（使用 cd 命令），然后删除其中的所有文件。但如果 cd 命令失败（例如，没有权限进入该目录或者敲错了目录名），则不能删除任何文件。

4.5.2 解决方案

可以用 cd 命令的退出状态（$?）配合 if 语句来实现仅当 cd 命令成功时执行 rm 命令。

```
cd mytmp
if (( $? == 0 )); then rm * ; fi
```

以下写法更好，它更清晰地展现并诠释了我们的意图。

```
if cd mytmp; then rm * ; fi
```

4.5.3 讨论

如果手动敲入命令，显然用不着这么做。因为可以看到 cd 命令出现的任何错误，所以也就不会再用 rm 命令。但是写脚本就是另一回事了，对于示例中这样的脚本，进行测试是非常有必要的，这可以确保不会意外地将所在目录中的文件全部删除。

假设在错误的目录中运行这个脚本，该目录下并没有子目录 mytmp，那么 cd 命令会失败，当前目录仍保持不变。如果少了 if 语句（检查 cd 命令是否失败），就会继续执行脚本中的下一条语句 rm *，删除当前目录内的所有文件。要这么看的话，if 语句还真是不能少。

那么变量 $? 中的值是怎么来的？这个值是命令的退出码（参见 4.4 节）。C 语言程序员会将其作为提供给 exit() 函数的参数值，例如，exit(4); 会返回 4。对于 shell 而言，退出码 0 代表成功，非 0 则代表失败。

如果正在编写 bash 脚本，那么要确保明确设置了返回值，以便 $? 得以正确设置。如果不这样做，$? 的值则是最后执行的那个命令的返回值，但这未必是你想要的结果。

4.5.4 参考

- 4.4 节
- 4.6 节

4.6 减少if语句的数量

4.6.1 问题

作为一名尽职尽责的程序员，你牢记着上节所讲的内容。你将相关的概念应用于最近编写的 shell 脚本，却发现其中充斥着检查各个命令返回值的 if 语句，导致脚本难以阅读。有没有什么别的办法呢？

4.6.2 解决方案

在 bash 中使用 && 运算符，根据条件执行命令。

```
cd mytmp && rm *
```

4.6.3 讨论

用 && 分隔两个命令，以此告诉 bash 先执行第一个命令，如果该命令成功（退出状态为 0），再执行第二个命令。这非常类似于用 if 语句检查第一个命令的退出状态，从而判断是否执行第二个命令。

```
cd mytmp
if (( $? == 0 )); then rm * ; fi
```

&& 语法会让人想起 C 语言中的逻辑 AND 运算符。如果了解逻辑运算（以及 C 语言），那么你应该记得在评估逻辑表达式 A AND B 时，如果（子）表达式 A 和（子）表达式 B 均为真，则整个表达式才为真。如果其中任何一个为假，则整个表达式为假。C 语言利用了这个事实，当编写 if (A && B) { ... } 之类的表达式时，它会先评估表达式 A。如果为假，那么它甚至都不会再费事去评估 B，因为整个表达式的结果（假）已经由此（A 为假）确定了。

这跟 bash 有什么关系？如果第一个命令（&& 的左侧）的退出状态不为 0（失败），那么它就不会再评估第二个表达式，也就是说压根不会执行另一个命令。

要想彻底检查错误，但又不想到处出现 if 语句，可以设置 -e 标记，这样的话，只要脚本中有任何命令（排除在 while 循环和 if 语句中，因为其本身就要用到退出状态）出现错误（退出状态为非 0），bash 就会退出。

```
set -e
cd mytmp
rm *
```

设置了 -e 标记之后，shell 会在命令失败时退出。如果本例中的 cd 命令失败，脚本则直接退出，不再执行 rm * 命令。但不推荐在交互式 shell 中这么做，因为如果 shell 退出，那么终端窗口也会随之消失。

4.6.4 参考

- 4.8 节讲解了 || 语法，其形式与 && 类似，但意义大不相同。

4.7 无人值守下运行耗时作业

4.7.1 问题

你在后台运行了一项作业，然后退出 shell，出去喝了杯咖啡。回来检查时发现作业已经停止运行，而且还未完成。实际上，该作业压根就没有完成多少。你退出 shell 时，它似乎也跟着退出了。

4.7.2 解决方案

如果想在后台运行作业并在该作业完成前退出 shell，那就需要对作业使用 nohup。

```
$ nohup long &
nohup: appending output to `nohup.out'
$
```

4.7.3 讨论

将作业置入后台时（通过 4.3 节中介绍过的 &），它仍旧是 bash shell 的子进程。如果退出 shell 的某个实例，bash 就会向其所有子进程发送 hangup 信号。这就是作业运行不了多久的原因。只要

退出 bash，后台作业就会被“杀死”。（嗨，你要离开了。它是怎么知道的呢？）

nohup 命令只是设置子进程忽略 hangup 信号。你仍可以用 kill 命令“杀死”作业，因为 kill 发送的是 SIGTERM 信号，而非 SIGHUP 信号。但有了 nohup，作业就不会在退出 bash 时被无意间“杀死”。

nohup 给出的那句关于追加输出的消息只是为了提高自身的实用性。因为你有可能发出 nohup 命令后就退出 shell，输出信息也就无处可去了，也就是说，终端中的 bash 会话已经结束，作业无法再向 STDOUT 写入。更重要的是，向不存在的位置写入信息会产生错误。因此，nohup 会替你重定向输出，将其追加（不是覆盖，而是添加到文件现有内容的末尾）到当前目录下的 nohup.out 文件中。你也可以明确地在命令行上指定将输出重定向到其他地方，nohup 足够聪明，能够发现你已经另有安排，也就不会再使用 nohup.out 了。

4.7.4 参考

- 第 2 章，考虑到你可能会对后台作业执行输出重定向，其中涵盖了各种与输出重定向相关的实例。
- 4.3 节
- 10.1 节
- 17.4 节

4.8 出现故障时显示错误消息

4.8.1 问题

shell 脚本需要详细报告故障。你想在命令出现问题时看到错误消息，但 if 语句更多地只是转移语句的执行流程。

4.8.2 解决方案

一些 shell 程序员的惯用做法是配合使用 || 和命令来输出调试 / 错误消息，如下所示：

```
cmd || printf "%b" "cmd failed. You're on your own\n"
```

4.8.3 讨论

在 4.6 节中，&& 告诉 bash 如果第一个表达式为假，则不再评估第二个表达式，与此类似，|| 告诉 bash 如果第一个表达式为真（成功），则不再评估第二个表达式。和 && 一样，|| 可以追溯到逻辑运算和 C 语言，如果 A OR B 中的表达式 A 为真，则最终结果就为真，无须评估表达式 B。在 bash 中，如果第一个表达式返回 0（成功），则跳过剩下的表达式，继续往下执行。仅当第一个表达式返回非 0 值（命令出错）时，才必须评估第二个表达式中的命令。[1]

[1]shell 使用 0 代表真，非 0 代表假，这正好和其他编程语言相反。在 shell 中判断逻辑表达式的结果时，一定要注意这点。——译者注

警告！千万别被下面这条语句骗了。

```
cmd || printf "%b" "FAILED.\n" ; exit 1
```

exit 无论如何都不会执行！|| 仅作用于前两个命令。如果只想在 cmd 出错时执行 exit，那么要将其与 printf 分组到一起，以便两者被视为一个单元。语法如下所示：

```
cmd || { printf "%b" "FAILED.\n" ; exit 1 ; }
```

注意，最后一个命令必须以分号结尾，闭合花括号与其中的内容之间要用空白字符分隔。[2] 具体的讨论参见 2.14 节。

[2]这么做的原因详见 Bash Reference Manual 的 3.2.4.3 节。——译者注

4.8.4 参考

- 2.14 节
- 4.6 节

4.9 执行变量中的命令

4.9.1 问题

你希望在脚本中根据情况执行不同的命令。这该怎么实现呢？

4.9.2 解决方案

该问题有很多种解决方案，这正是脚本的用武之地。后续章节会讨论可以搞定此问题的各种编程构件，如 `if/then/else`、`case` 语句等。但这里要给出一种略有不同的方法，它揭示了 bash 自身的一些东西。我们不仅可以将变量内容（详见第 5 章）用于参数，还可以用于命令本身。

```
FN=/tmp/x.x
PROG=echo
$PROG $FN
PROG=cat
$PROG $FN
```

4.9.3 讨论

我们可以将命令名保存在变量（`$PROG`）中，然后在需要命令名的地方引用该变量，bash 会使用变量（`$PROG`）的值作为要执行的命令。bash 解析命令行，用相应的值替换其中的变量，最后将替换后的结果作为最终命令，就像是我们一字不差地敲入那样。

> 注意你所使用的变量名。有些程序（如 InfoZip）使用环境变量（如 `$ZIP` 和 `$UNZIP`）向自身传递设置，如果你做了类似 `ZIP=/usr/bin/zip` 的操作，少不了要抓耳挠腮上几天，搞不明白为什么该程序可以在命令行上正常运行，偏偏在脚本中就不行了。相信我们，这都是从惨痛教训中总结出的经验。另外，好好读读手册。

4.9.4 参考

- 第 11 章
- 14.3 节
- 16.21 节
- 16.22 节
- 附录 C，其中描述了在命令行上执行的各种替换操作。在讨论这个话题前，你应该再多读几章。

4.10 执行目录中的所有脚本

4.10.1 问题

你想要执行一系列脚本，但是脚本清单并不固定，新脚本不停地加入其中，可你也不想总是修改清单。

4.10.2 解决方案

将要执行的脚本全都放进一个目录，让 bash 逐一执行。不用保存脚本清单，用该目录的内容作为清单即可。以下脚本会执行特定目录中的所有可执行文件。

```
for SCRIPT in /path/to/scripts/dir/*
do
    if [ -f "$SCRIPT" -a -x "$SCRIPT" ]
    then
        $SCRIPT
    fi
done
```

4.10.3 讨论

第 6 章将详细讨论 `for` 循环和 `if` 语句，这里先小试牛刀。变量 `$SCRIPT` 会依次获得通配符 `*` 匹配到的各个文件名，该通配符能够匹配指定目录下的所有内容（文件名以点号开头的隐藏文件除外）。

如果匹配到的是文件（由 `-f` 测试）且具有执行权限（由 `-x` 测试），那么 shell 会尝试执行此脚本。

在这个简单示例中，我们无法为执行的脚本指定参数。该脚本也许能够很好地满足你的个人需求，但算不上稳健。有些人可能认为这很危险，但我们希望你能从中先了解一下脚本编程的功用。

4.10.4 参考

- 第 6 章，其中提供了有关 `for` 循环和 `if` 语句的更多信息。

第 5 章　脚本编程基础：shell 变量

bash shell 编程和其他编程语言差不多，同样包含变量（存放字符串和数值的容器，可以进行修改、比较、传递）。在引用 bash 变量时，可以使用一些非常特殊的运算符。bash 还拥有内建变量，这些变量可以提供有关脚本中其他变量的重要信息。本章将介绍 bash 变量和一些特殊的变量引用机制，展示如何将其运用于你自己的脚本。

bash 脚本中的变量名称通常采用全大写，但这并非强制性的，只是一种常见做法而已。变量不用事先声明，直接使用就行了。变量基本上都是字符串类型，不过有些运算符能够将变量内容视为数字。变量的实际用法如下所示。

```
# 使用shell变量的普通脚本
# （不过好歹也注释过了）
MYVAR="something"
echo $MYVAR
# 写法类似，但没有引号
MY_2ND=anotherone
echo $MY_2ND
# 这里需要使用引号：
MYOTHER="more stuff to echo"
echo $MYOTHER
```

bash 变量的语法有两处要点，但可能不那么一目了然。首先，赋值语法 *name=value* 看起来相当直观，但 = 两侧不能有任何空白字符。

我们来想想为什么是这样。记住，shell 的主要目的是执行命令：你在命令行上指定要执行的命令。命令名之后的任何单词都会作为参数传给该命令。例如，当你敲入：

```
ls filename
```

其中，单词 ls 是命令名，*filename* 是这个示例中第一个，也是唯一一个参数。

这有什么关系？考虑一下，如果允许 = 两侧出现空白字符，那么变量赋值就会变成下面这样：

```
MYVAR = something
```

看出来没有，此时 shell 很难区分出到底是要调用命令（如本例中的 ls）还是要给变量赋值。对于能够以 = 为参数的命令（如 test）更是如此。所以，还是让事情简单点吧：变量赋值时，shell 不允许在 = 两侧出现空白字符。该规定的另一方面也值得注意：不要在文件名中使用 =，对于 shell 脚本尤为如此（尽管可行，但不推荐）。

其次需要注意的是，引用变量时要使用 $ 符号。给变量赋值时不需要在变量名前加 $，但获取变量值时需要。（出现在表达式 $(( )) 中的变量是个例外。）用编译器的行话来说，赋值和检索值在语法上的差异就是变量的左值（L-value）与右值（R-value）之间的差异（赋值运算符的**左侧**和**右侧**）。

原因很简单，就是为了消除歧义。思考下列情况：

```
MYVAR=something
echo MYVAR is now MYVAR
```

正如这个示例试图指出的，你能分辨出哪个是字符串 MYVAR，哪个是变量 MYVAR 吗？使用引号？如果将字符串放进引号，那么一切只会更乱，你得将包括命令在内的所有非变量名全部加上引号！没人想像下面这样敲命令：

```
"ls" "-l" "/usr/bin/xmms"
```

（对于那些想试试看的用户来说，没错，这么写确实也管用。）因此，不用将一切都加上引号，使用右值语法来引用变量就够了。要想获得与某个变量名相关联的值，在变量名前加上 $ 符号即可。

```
MYVAR=something
echo MYVAR is now $MYVAR
```

只需记住，bash 中的一切都是字符串，所以需要用 $ 来表明变量引用。另外，有可能还得在变量名两边加上花括号，原因参见 5.4 节。

5.1 记录脚本

5.1.1 问题

详细讨论 shell 脚本或变量前，我们还得说说如何记录（documenting）脚本。毕竟，你得能看明白自己的脚本，即便是在编写完的几个月后。

5.1.2 解决方案

用注释记录脚本。# 代表注释的开始。该行上随后的所有字符都会被 shell 忽略。

```
#
# 这是一行注释
#
# 多用注释
# 注释是你的好朋友
```

5.1.3 讨论

有些人将 shell 语法、正则表达式，以及 shell 脚本编程的其他部分描述为**只写（write-only）**语法，以此暗示很多 shell 脚本中难以理解的错综复杂之处。

要想让你的 shell 脚本避开这种陷阱，最好的方法就是该用注释的地方就用注释（另一种方法是使用有意义的变量名）。在奇怪的语法或紧凑的表达式前加上注释的确有用。

```
# 用空格替换分号
NEWPATH=${PATH/;/ }
#
# 交换分号两侧的文本
sed -e 's/^\(.*\);\(.*\)$/\2;\1/' < $FILE
```

甚至可以在命令提示符下用交互式 shell 输入注释。该功能可以关闭，但默认处于启用状态。在有些情况下，可能用得上交互式注释。

5.1.4 参考

- 表 5-1
- A.8 节，其中介绍了如何启用或关闭交互式注释

5.2 在shell脚本中嵌入文档

5.2.1 问题

你想用一种简单方法为脚本提供格式化过的终端用户文档（如手册页或 HTML 页面）。你希望能将代码和文档标记放在同一个文件中，以此简化更新、分发以及版本控制。

5.2.2 解决方案

使用内建命令 :（空命令）和 here-document 在脚本中嵌入文档，如例 5-1 所示。

例 5-1 ch05/embedded_documentation

```
#!/usr/bin/env bash
# 实例文件：embedded_documentation

echo 'Shell script code goes here'

# 使用:和here-document嵌入文档
: <<'END_OF_DOCS'

Embedded documentation such as Perl's Plain Old Documentation
```

```
(POD),
or even plain text here.

Any accurate documentation is better than none at all.

Sample documentation in Perl's Plain Old Documentation (POD)
format adapted from
CODE/ch07/Ch07.001_Best_Ex7.1 and 7.2 in the Perl Best Practices
example tarball
"PBP_code.tar.gz".

=head1 NAME

MY~PROGRAM--One-line description here

=head1 SYNOPSIS

  MY~PROGRAM [OPTIONS] <file>

=head1 OPTIONS

  -h = This usage.
  -v = Be verbose.
  -V = Show version, copyright, and license information.


=head1 DESCRIPTION

A full description of the application and its features.
May include numerous subsections (i.e., =head2, =head3, etc.)

[...]

=head1 LICENSE AND COPYRIGHT

=cut

END_OF_DOCS
```

然后用下列命令提取并使用 POD 文档。

```
# 自动分页，以便在屏幕上阅读
$ perldoc myscript

# 只提取usage小节
$ pod2usage myscript

# 创建HTML版
$ pod2html myscript > myscript.html
```

```
# 创建手册页
$ pod2man myscript > myscript.1
```

5.2.3 讨论

任何纯文本文档或标记都可以像这样使用，要么散布在脚本内，要么集中放在脚本末尾，后一种做法会更好些。考虑到安装了 bash 的计算机系统可能也安装了 Perl，POD 格式也许是不错的选择。Perl 通常包含 pod2* 程序，可用于将 POD 转换成 HTML、LaTeX、手册页、文本以及用法文件。

Damian Conway 所著的 *Perl Best Practices*（O'Reilly 出版）一书中有一些优秀的库模块和应用程序文档模板，可以轻松地转换包括纯文本在内的任何文档格式。参见本书示例归档文件中的 CODE/ch07/Ch07.001_Best_Ex7.1 和 CODE/ch07/Ch07.001_Best_Ex7.2。

要想将所有的嵌入式文档放在脚本最后，还可以在文档起始位置前加上 `exit 0`。这样就会直接退出脚本，不再强制 shell 解析每一行代码来查找 here-document 的结尾，运行速度会快一点。不过你得小心，别覆盖了上一个运行失败的命令的退出码，因此可以考虑使用 `set -e`。如果你的脚本中散布着代码和嵌入式文档，那就别这样做了。

5.2.4 参考

- 4.6 节，其中讲解了 `set -e`
- http://examples.oreilly.com/perlbp/PBP_code.tar.gz

5.3 提高脚本可读性

5.3.1 问题

你希望尽可能提高脚本的可读性，以便于理解和日后维护。

5.3.2 解决方案

遵循以下最佳实践。

- 按照 5.1 节和 5.2 节中的做法记录脚本。
- 缩进并明智地使用垂直空白字符。
- 使用有意义的变量名。
- 使用函数（10.4 节）并指定有意义的函数名。
- 在少于 76 个字符左右的有意义的位置处断行。
- 将最有意义的部分放在左边。

5.3.3 讨论

记录意图，而不是代码中的繁枝末节。如果遵循余下的要点，则代码应该会相当清晰。写代码时，编制提醒、提供数据布局示例或标题，记下脑子里想到的方方面面的细节。如果实现过程比较微妙或难懂，代码本身也要加以说明。

我们推荐缩进时使用 4 个空格作为一级，不要用制表符，尤其是别混用制表符和空格。这么做的原因有很多，通常是出于个人喜好或公司标准的缘故。无论你的编辑器（比例字体除外）或打印机如何设置，4 个空格始终就是 4 个空格 [1]。这段缩进距离大到足以让你浏览脚本时一目了然，小到可以在不超出屏幕或打印页面右边界的情况下设置行内多级缩进。如果有续行，我们还建议再额外加上 2 个空格（或者根据需要调整数量），以便代码的清晰性达到最佳。

> [1]部分编辑器会自动将制表符替换成空格，有的设置 1 个制表符等于 4 个空格，有的设置 1 个制表符等于 8 个空格。——译者注

可以用垂直空白字符创建功能相似的代码块。当然，对函数也是如此。

使用有意义的变量名和函数名。唯一能接受 `$i` 或 `$x` 这种写法的时候就是在 `for` 循环中。你也许觉得简短神秘的名称能节省眼前的时间、少敲键盘，但我们敢保证，当不得不修正或改动脚本时，你绝对会在某个地方搭进去 10 倍或 100 倍的时间。

对于比较长的行，选择在 76 个字符左右的位置处断行。没错，我们知道大多数屏幕（更准确地说，应该是终端程序）能处理的字符个数远超于此，但是 80 个字符宽度的纸张和屏幕仍是默认设置，而且在代码右侧留些空白绝对没什么坏处。不断往右边拖滚动条或看到屏幕上那些丑陋的折行，绝不是一件令人愉悦的事情，还容易让人分神。可别这么做。

但这条规则也有例外。在某些情况下，（可能通过 SSH）将生成的行发往别处时，断行会得不偿失。但大部分情况下，这么做是值得的。

要断行的话，试着将最有意义的部分放在左边：我们是从左往右阅读脚本代码的，这样续行才会更显眼。查找续行时也更容易识别。你觉得以下哪种写法更清晰？

```
# 不错
[ -n "$results" ] \
  && echo "Got a good result in $results" \
  || echo 'Got an empty result, something is wrong'

# 也挺好
[ -n "$results" ] && echo "Got a good result in $results" \
                  || echo 'Got an empty result, something is
wrong'

# 尚可，但不理想
[ -n "$results" ] && echo "Got a good result in $results" \
  || echo 'Got an empty result, something is wrong'

# 不好
[ -n "$results" ] && echo "Got a good result in $results" || \
echo 'Got an empty result, something is wrong'

# 不好（行尾的\是可选的，但出于清晰性的考虑，此处建议使用）
[ -n "$results" ] && \
  echo "Got a good result in $results" || \
  echo 'Got an empty result, something is wrong'
```

5.3.4 参考

- 5.1 节
- 5.2 节
- 10.4 节

5.4 将变量名与周围的文本分开

5.4.1 问题

你需要输出变量以及其他文本。引用变量要用到 $，但是该怎么区分变量名与紧随其后的其他文本呢？例如，你想要用 shell 变量作为文件名的一部分。

```
for FN in 1 2 3 4 5
do
    somescript /tmp/rep$FNport.txt
done
```

shell 会怎么理解这段代码？它会认为变量名从 $ 开始，到点号结束。换句话说，它将 $FNport 视为变量名，而非我们想要的 $FN。

5.4.2 解决方案

使用完整的变量引用语法，不仅要包括 $，还要在变量名周围加上花括号。

```
somescript /tmp/rep${FN}port.txt
```

5.4.3 讨论

因为 shell 变量名中只能包含字母、数字以及下划线，所以很多时候并不需要使用花括号。任何空白字符或标点符号（下划线除外）都足以提示变量名的结束位置。但只要有疑问，就应该用花括号。实际上，有人认为坚持使用花括号是一种好习惯，这样就不用考虑什么时候该用，什么时候不用，还能在整个脚本中保持写法一致。也有人觉得这样需要敲入的可用可不用的字符太多了，不但不好输入，还会让代码看起来很繁杂。说到底，这还是个人喜好问题。

5.4.4 参考

- 1.8 节

5.5 导出变量

5.5.1 问题

你在某个脚本中定义了一个变量，但在调用其他脚本时，该脚本并不知道这个变量的存在。

5.5.2 解决方案

将希望传给其他脚本的变量导出。

```
export MYVAR
export NAME=value
```

5.5.3 讨论

有时候，两个脚本互不知道对方的变量是件好事。如果在一个脚本的 for 循环中调用了另一个 shell 脚本，那么你肯定不希望这个脚本将 for 循环弄得乱七八糟（这种事不大可能发生，因为该脚本几乎肯定是在子 shell 中运行，这里只是举例说明）。

但有时你的确想在脚本之间传递信息。在这种情况下，你可以导出变量，使变量的值能够传递给脚本所调用的其他程序。

要想查看所有已导出的变量，敲入命令 env（或者内建命令 export -p）就能列出各个变量及其值。当脚本运行时，这些变量都可供使用，其中很多是 bash 启动脚本已经设置好的（有关 bash 的配置和定制，参见第 16 章）。

你可以在 export 后面跟上变量赋值，不过这种写法不适用于比较老的 shell 版本。也可以在 export 后面跟上要导出的变量名。尽管只要把 export 语句放在待导出的值之前就行，但脚本程序员通常会将这类语句（如变量声明）一块放在脚本的起始位置。

导出之后，就可以随意给变量赋值，不用重复导出。因此，有时你会看到下列语句。

```
export FNAME
export SIZE
export MAX
...
MAX=2048
SIZE=64
FNAME=/tmp/scratch
```

还有这样的语句。

```
export FNAME=/tmp/scratch
export SIZE=64
export MAX=2048
...
FNAME=/tmp/scratch2
...
FNAME=/tmp/stillexported
```

注意，导出的变量实际上是**按值调用的**。在被调用脚本中修改导出变量的值并不会改变调用脚本中该变量的值。

这就产生了一个问题：“如何将被调用脚本修改过的值传回来？”答案是：做不到。

你只能设计脚本时避开这种需求。有什么办法能够应对这种限制吗？

方法之一是让被调用的脚本自己输出修改过的值，然后调用脚本读取该输出。例如，某个脚本导出了变量 `$VAL`，然后调用了另一个脚本，后者修改了 `$VAL`。要想在调用脚本中得到 `$VAL` 的新值，就得将修改后的值写入标准输出，然后获取并重新赋值。

```
VAL=$(anotherscript)
```

（有关 `$()` 语法的更多讲解，参见 10.5 节。）你甚至可以修改多个值并依次将其写入标准输出。然后调用脚本会用 `read` 命令逐行将其读入对应的变量。但是，这要求被调用脚本不要向标准输出中写入其他输出（至少别在此之前或期间），而且脚本之间要设置非常密切的依赖关系（从维护角度来看，这可不是什么好事）。

5.5.4 参考

- help export
- 第 16 章，其中介绍了 bash 的配置和自定义
- 5.6 节
- 10.5 节
- 19.5 节

5.6 查看所有的变量值

5.6.1 问题

如何查看哪些变量已经导出，对应的值是什么？徒手用 echo 命令逐个检查？你怎么知道哪些变量已经导出了？

5.6.2 解决方案

用 set 命令查看当前 shell 中的所有变量值以及函数定义。

用 env（或 export -p）命令查看那些被导出的、可用于子 shell 的变量。

在 bash 4 或更高版本中，也可以使用 declare -p 命令。

5.6.3 讨论

不加任何参数的 set 命令会以 *name=value* 的格式（在标准输出上）生成当前定义的所有 shell 变量及其值的列表。env 命令与之类似。不管执行哪个命令，你都会看到一个相当长的变量列表，其中大多数变量你可能都不认识。这些变量是作为 shell 启动进程的一部分而创建的。

env 生成的列表是 set 生成的列表的子集，因为并非所有变量都被导出了。

如果对某些变量或值感兴趣，但又不想生成完整的列表，通过管道将结果传给 grep 命令即可。例如：

```
set | grep MY
```

以上代码只显示名称或值中包含字符串 MY 的变量。

较新的 declare -p 命令在输出中按照声明和初始化变量的形式显示变量名及其值。以下是输出片段。

```
$ declare -p
...
declare -i MYCOUNT="5"
declare -x MYENV="10.5.1.2"
declare -r MYFIXED="unchangeable"
declare -a MYRA=([0]="5" [1]="10" [2]="15")
...
$
```

declare 语句形式的输出可以在 shell 脚本中作为源代码，重新创建这些变量并为其赋值。各个选项（-i、-x、-r、-a）分别指明了变量为整数类型、已经导出、只读、数组类型。

5.6.4 参考

- help set
- help export
- help declare
- man env
- 第 16 章，其中介绍了 bash 的配置和自定义
- 附录 A，其中列出了全部的 shell 内建变量

5.7 在shell脚本中使用参数

5.7.1 问题

你希望用户能在调用脚本时指定参数。可以要求用户设置一个 shell 变量，但这种做法似乎不够灵活。另外还需要向其他脚本传递数据。这可以通过环境变量实现，但会使两个脚本之间的联系过于紧密。

5.7.2 解决方案

使用命令行参数。在命令行上，出现在脚本名之后的任意单词都可以在脚本中作为编号变量（numbered variable）被访问。假设有下列脚本 simplest.sh。

```
# 一个简单的shell脚本
echo $1
```

该脚本会显示在命令行上被调用时所指定的第一个参数。我们来看一种实际用法。

```
$ cat simplest.sh
# 一个简单的shell脚本
echo ${1}
$ ./simplest.sh you see what I mean
you
$ ./simplest.sh one more time
one
$
```

5.7.3 讨论

其他参数的可用形式分别为 ${2}、${3}、${4}、${5} 等。单个数位的数字用不着花括号，除非要区分变量名与其后出现的文本。典型的脚本只用到少部分参数，但如果涉及 ${10}，那就得使用花括号了，否则 shell 会将 $10 理解为 ${1} 后面紧跟着字符串 0，如下所示。

```
$ cat tricky.sh
echo $1 $10 ${10}
$ ./tricky.sh I II III IV V VI VII VIII IX X XI
I I0 X
$
```

第 10 个参数的值是 X，但如果在脚本中写成 $10，那么你在 echo 语句中得到的会是第一个参数 $1，后面紧跟着一个字符串 0。

5.7.4 参考

- 5.4 节

5.8 遍历传入脚本的参数

5.8.1 问题

你想对指定的一系列参数执行某些操作。编写 shell 脚本，对单个参数进行处理不是什么问题，只需要用 $1 引用这个参数即可。但如果面对的是一大批文件呢？你可能想这样调用脚本。

```
./actall *.txt
```

shell 会进行模式匹配，生成匹配 *.txt 模式（以 .txt 结尾的文件名）的文件名列表。

5.8.2 解决方案

特殊的 shell 变量 $* 能够引用所有的参数，可以将其用于 for 循环，如例 5-2 所示。

例 5-2 ch05/chmod_all.1

```
#!/usr/bin/env bash
# 实例文件：chmod_all.1
#
# 批量修改文件权限
#
for FN in $*
do
    echo changing $FN
    chmod 0750 $FN
done
```

5.8.3 讨论

变量 $FN 是我们自己挑选的；使用别的变量名也没有任何问题。$* 引用的是命令行上出现的所有参数。假如用户输入：

```
./actall abc.txt another.txt allmynotes.txt
```

调用该脚本时，$1 等于 abc.txt、$2 等于 another.txt、$3 等于 allmynotes.txt，而 $* 等于整个参数列表。换句话说，shell 替换 for 语句中的 $* 后，脚本就变成了如下这样：

```
for FN in abc.txt another.txt allmynotes.txt
do
  echo changing $FN
  chmod 0750 $FN
done
```

for 循环从列表中获取第一个值，并将其赋给变量 $FN，然后执行 do 和 done 之间的语句。列表中的其他值会重复执行该过程。

别急，还没完！如果文件名中不包含空格，该脚本则万事大吉，但有时难免碰上带有空格的文件名。阅读接下来两节的内容，看看如何改进这个脚本。

5.8.4 参考

- help for
- 5.9 节
- 5.10 节
- 6.12 节

5.9 处理包含空格的参数

5.9.1 问题

你编写了一个可以接受文件名作为参数的脚本，看起来一切正常，但有一次脚本出现了问题，结果发现是因为文件名中带有空格。

5.9.2 解决方案

你得仔细将所有可能包含文件名的命令参数全部加上引号。引用变量时，将其放入双引号中。

5.9.3 讨论

真得多谢谢苹果公司了！为了向用户示好，这家公司的设计师普及了一个概念：在文件名中，空格是有效的字符。因此，用户可以将自己的文件命名为 My Report 和 Our Dept Data，而不是只能用那种又丑又没什么可读性的 MyReport 和 Our_Dept_Data。（还有人能明白这种老派文件名的意思吗？）好吧，接下来就该 shell 难受了，原先空格是基本的单词分隔符，文件名始终是一个单词，可如今未必如此了。

那我们该如何应对呢？

在 shell 脚本中，曾经简单的写作 `ls -l $1` 的地方，现在最好给参数加上引号，改写成 `ls -l "$1"`。否则，如果参数包含空格，那么会被 shell 解析成两个单词，`$1` 中只会包含部分文件名。我们来看一个**运行失败**的脚本示例。

```
$ cat simpls.sh
# 一个简单的shell脚本
ls -l ${1}
$
$ ./simple.sh Oh the Waste
ls: Oh: No such file or directory
$
```

如果调用脚本时没有将文件名放进引号，那么 bash 会看到 3 个参数并将 `$1` 替换成第 1 个参数（Oh）。`ls` 命令运行时只有一个参数 Oh，结果就是无法找到该文件。

接下来，我们在调用脚本时给文件名加上引号。

```
$ ./simpls.sh "Oh the Waste"
ls: Oh: No such file or directory
ls: the: No such file or directory
ls: Waste: No such file or directory
$
```

还是不行。bash 得到了一个包含 3 个单词的文件名，并将 ls 命令中的 $1 替换成了该文件名。到目前一切都还好。但是，我们并没有将脚本中的变量引用放入引号，因此 ls 将文件名中的各个单词视为单独的参数（作为单独的文件名）。结果还是无法找到这些文件。

我们将变量引用放进引号。

```
$ cat quoted.sh
# note the quotes
ls -l "${1}"
$
$ ./quoted.sh "Oh the Waste"
-rw-r--r--  1 smith users 28470 2007-01-11 19:22 Oh the Waste
$
```

如果将变量引用写成 "${1}"，它就会被视为一个单词（单个文件名），这样 ls 就只得到了一个参数（文件名），任务顺利完成。

5.9.4 参考

- 第 19 章，其中讲解了一些常见的错误
- 1.8 节，其中介绍了一些有关 shell 引用的技巧
- 附录 C，其中介绍了有关命令行处理的更多信息

5.10 处理包含空格的参数列表

5.10.1 问题

按照上节中的建议，你给变量加上引号，但是仍然出现了错误。脚本和 5.8 节中的类似，如果文件名中带有空格，就会报错。

```
for FN in $*
do
    chmod 0750 "$FN"
done
```

5.10.2 解决方案

报错的原因与 for 循环中使用的 $* 有关。在这个示例中，我们需要用到另一个不同但相关的 shell 变量 $@。如果该变量出现在引号中，则会得到一个命令行参数列表，其中每个参数都会被单独引用起来。修改后的 shell 脚本如例 5-3 所示。

例 5-3 ch05/chmod_all.2

```
#!/usr/bin/env bash
# 实例文件：chmod_all.2
#
# 在文件名包含空格时选择更好的引号添加方式，批量修改文件权限
#
for FN in "$@"
do
    chmod 0750 "$FN"
done
```

5.10.3 讨论

变量 $* 会扩展成 shell 脚本的参数列表。如果按照以下方式调用脚本：

```
myscript these are args
```

那么 $* 引用的就是这 3 个参数：these are args。当其用于 for 循环中时：

```
for FN in $*
```

第一次循环将第一个单词（these）赋给 $FN，第二次循环将第二个单词（are）赋给 $FN，以此类推。

如果参数是文件名，而且是以模式匹配的方式出现在命令行上，按以下方式调用脚本时：

```
myscript *.mp3
```

shell 会匹配到当前目录下以 .mp3 结尾的所有文件并将其传给脚本。思考以下示例，其中有 3 个 MP3 文件，名称分别为：

```
vocals.mp3
cool music.mp3
tophit.mp3
```

第二首 MP3 的文件名中的 cool 和 music 之间有一个空格。按照以下方式调用脚本时：

```
myscript *.mp3
```

实际上得到的是：

```
myscript vocals.mp3 cool music.mp3 tophit.mp3
```

如果脚本中包含以下这行：

```
for FN in $*
```

该行会扩展为：

```
for FN in vocals.mp3 cool music.mp3 tophit.mp3
```

列表中共有 4 个单词。第二首 MP3 的文件名中的第 5 个字符是空格（cool music.mp3），这导致 shell 将此文件名视为两个单词（cool 和 music.mp3），因此，在 for 循环的第二次迭代中，$FN 的值就是 cool。第三次迭代中的 $FN 的值是 music.mp3，但因为这两个文件名都不存在，所以出现了文件无法找到的错误信息。

将 $* 引用起来似乎行得通，但是：

```
for FN in "$*"
```

会扩展成：

```
for FN in "vocals.mp3 cool music.mp3 tophit.mp3"
```

结果就是 $FN 只得到了一个值（整个参数列表）。你会看到如下错误信息。

```
chmod: cannot access 'vocals.mp3 cool music.mp3 tophit.mp3': No
such file or
directory
```

因此，你得改用 shell 变量 $@ 并将其放入引号。如果不加引号，$* 和 $& 没什么两样。但当两者出现在引号中时，bash 就会区别对待了。就像先前看到的那样，"$*" 得到的是整个参数列表。而 "$@" 得到的可不是一个字符串，而是与各个参数对应的带有引号的字符串列表。

在使用 MP3 文件名的示例中：

```
for FN in "$@"
```

会扩展为：

```
for FN in "vocals.mp3" "cool music.mp3" "tophit.mp3"
```

你可以看到，现在第二首 MP3 的文件名加上了引号，其中的空格也因此成了文件名的一部分，不再被视为单词分隔符。

在第二次循环中，$FN 的值是包含空格的 cool music.mp3。因此，使用 $FN 时要小心。可能你也想将其放入引号，这样一来，其中的空格就不再作为分隔符，而是整个字符串的一部分。也就是说，应该使用 "$FN"。

```
chmod 0750 "$FN"
```

难道不应该始终在 for 循环中使用 "$@" 吗？好吧，敲入这几个字符实在是太麻烦了，对于一些只为求快的脚本来说，如果你知道文件名中没有空格，沿用老派的 $* 语法基本没什么大碍。对于那些更稳健的脚本而言，安全起见，建议使用 "$@"。本书可能会交替使用这两种语法，尽管我们更清楚二者的区别，但旧习难改，而且我们当中有人“绝不会”在文件名中使用空格！

5.10.4 参考

- 5.8 节
- 5.9 节
- 5.12 节
- 6.12 节

5.11 统计参数数量

5.11.1 问题

你想知道调用脚本时使用了多少个参数。

5.11.2 解决方案

使用 shell 内建变量 $#。例 5-4 展示了一个严格要求 3 个参数的脚本。

例 5-4 ch05/check_arg_count

```
#!/usr/bin/env bash
# 实例文件：check_arg_count
#
# 检查正确的参数数量：
# 使用下列语法或者：if [ $# -lt 3 ]
if (( $# < 3 ))
then
    printf "%b" "Error. Not enough arguments.\n" >&2
    printf "%b" "usage: myscript file1 op file2\n" >&2
    exit 1
elif (( $# > 3 ))
then
    printf "%b" "Error. Too many arguments.\n" >&2
    printf "%b" "usage: myscript file1 op file2\n" >&2
    exit 2
else
    printf "%b" "Argument count correct. Proceeding...\n"
fi
```

以下分别是参数过多和参数数量正好时的运行情况。

```
$ ./myscript myfile is copied into yourfile
Error. Too many arguments.
usage: myscript file1 op file2
$ ./myscript myfile copy yourfile
Argument count correct. Proceeding...
```

5.11.3 讨论

在开头的注释（在脚本中坚持这么做总是件好事）之后，我们用 `if` 测试所提供的参数数量（保存在 `$#` 中）是否大于 3。如果答案是肯定的，则输出一条错误信息，提醒用户正确的脚本用法，然后退出。

错误信息会被重定向到标准错误。这种做法符合标准错误的本意：作为所有错误信息的通道。

该脚本还会根据检测到的错误返回不同的值。尽管这里没什么意义，但对于可能会被其他脚本调用的脚本而言，还是有用处的，这样就拥有了一种程序化的方法，不仅能够检测故障（非 0 的退出值），还可以区分不同的错误类型。

注意，别仅仅因为 `${#}`、`${#VAR}`、`${VAR#alt}` 都在花括号里用到了 `#`，就把三者搞混了。第一种语法可以获得参数的数量，第二种语法可以获得变量 `VAR` 所保存值的长度，最后一种语法会执行某种替换操作。

5.11.4 参考

- 4.4 节
- 5.1 节
- 5.12 节
- 5.18 节
- 6.12 节

5.12 丢弃参数

5.12.1 问题

所有的正式脚本可能都要有两种参数：修改脚本行为的选项以及要处理的真正参数。你需要用一种方法在处理完选项后将其丢弃。

例如，现在有如下脚本。

```
for FN in "$@"
do
    echo changing $FN
    chmod 0750 "$FN"
done
```

脚本内容非常简单，它会显示正在处理的文件名，然后修改文件权限。但有时你希望脚本静悄悄地工作，不要显示文件名。如何在保留 for 循环的同时添加一个能够关闭文件名显示的选项呢？

5.12.2 解决方案

用 shift 删除处理过的参数，如例 5-5 所示。

例 5-5 ch05/use_up_option

```
#!/usr/bin/env bash
# 实例文件：use_up_option
#
# 使用并丢弃一个选项
#
# 解析可选参数
VERBOSE=0
if [[ $1 = -v ]]
then
    VERBOSE=1
    shift
fi
#
# 实际工作如下
#
for FN in "$@"
do
    if (( VERBOSE == 1 ))
    then
        echo changing $FN
    fi
    chmod 0750 "$FN"
done
```

5.12.3 讨论

我们添加了标记变量 $VERBOSE，借此了解是否应该输出所处理的文件名。可是一旦 shell 脚本发现 -v 选项并设置好标记，我们就用不着参数列表中的 -v 了。shift 语句告诉 bash 将命令行参数挪动一个位置，使 $2 变成 $1、$3 变成 $2，以此类推，这样就丢弃了第一个参数（$1）。

如此一来，当 for 循环启动时，参数列表（$@）中就再也没有 -v，剩下的是紧随其后的那些参数。

这种解析参数的方法在处理单个选项时完全没问题，但要想应对多个选项，那就得多花点心思了。按照惯例，shell 脚本的选项应该不区分位置，也就是说，myscript -a -p 应该等同于 myscript -p -a。而且，稳健的脚本还应该能处理重复选项，要么忽略，要么报错。更全面细致的参数解析，参见 13.1 节，其中讨论了 bash 的内建命令 getopts。

5.12.4 参考

- help shift
- 5.8 节
- 5.11 节
- 5.12 节
- 6.15 节
- 13.1 节
- 13.2 节

5.13 获取默认值

5.13.1 问题

有一个可以接受命令行参数的 shell 脚本。你希望能够提供默认值，这样就不用每次都让用户输入那些频繁用到的值了。

5.13.2 解决方案

用 `${:-}` 语法引用参数并提供默认值。

```
FILEDIR=${1:-/tmp}
```

5.13.3 讨论

在引用 shell 变量时，有多种特殊运算符可用。`:-` 运算符的意思是，如果指定参数（这里是 `$1`）不存在或为空，则将运算符之后的内容（本例为 `/tmp`）作为值。否则，使用已经设置好的参数值。该运算符可用于任何 shell 变量，并不局限于位置参数（`$1`、`$2`、`$3` 等），但后者是最常用到的。

当然，你也可以用更多的代码来实现：用 `if` 语句检查变量是否为空或不存在（我们将这个作为练习留给你），但在 shell 脚本中，此类处理司空见惯，`:-` 运算符可谓是一种颇受欢迎的便捷写法。

5.13.4 参考

- bash 手册页中有关参数替换的部分
- Cameron Newham 所著的 *Learning the bash Shell, 3rd Edition*（O'Reilly 出版），91~92 页
- Nelson H. F. Beebe 与 Arnold Robbins 合著的 *Classic Shell Scripting*（O'Reilly 出版），113~114 页
- 5.14 节
- 表 5-1

5.14 设置默认值

5.14.1 问题

你的脚本依赖于某些常用（如 `$USER`）或业务特定的环境变量。要想构建一个稳健的 shell 脚本，就得确保这些变量都有合理的默认值。那么该如何确保呢？

5.14.2 解决方案

首次引用 shell 变量时，如果该变量没有值，则使用赋值运算符为其赋值。

```
cd ${HOME:=/tmp}
```

5.14.3 讨论

示例中所引用的 $HOME 会返回其当前值，除非它为空或者压根就没设置。对于后两种情况（为空或没有设置），返回 /tmp，该值还会被赋给 $HOME，随后再引用 $HOME 的话，返回的就是这个新值。

我们来看一下实际应用。

```
$ echo ${HOME:=/tmp}
/home/uid002
$ unset HOME # 这么做通常并非明智之举
$ echo ${HOME:=/tmp}
/tmp
$ echo $HOME
/tmp
$ cd ; pwd
/tmp
$
```

一旦删除该变量，它就不再具有任何值。然后，用 := 运算符作为变量引用的一部分时，$HOME 被替换成了新值（/tmp）。接下来引用 $HOME 返回的都是这个新值。

记住，赋值运算符有一个重要的例外：不能对位置参数（如 $1 或 $*）赋值。在这种情况下，可以使用 :-（如 ${1:-*default*}），该表达式只返回值，但不进行赋值。

顺便说一下，注意 ${VAR:=*value*} 和 ${VAR:-*value*} 在形式上的差异，也许可以帮助你记忆这两种让人抓狂的符号。:= 执行赋值操作，同时返回运算符右侧的值。:- 只做了前者一半的工作：返回值，但不赋值。因此，它的符号也只有等号的一半（一个横杠，而不是两个）。

5.14.4 参考

- 5.13 节
- 表 5-1

5.15 使用空值作为有效的默认值

5.15.1 问题

你需要设置默认值，但希望空字符串也能作为有效值。你只想在变量不存在时才替换默认值。

${:=} 运算符在两种情况下会使用新值：一种是指定的 shell 变量不存在（或被明确删除）；另一种是变量存在，但值为空，就像 HOME="" 或 HOME=$OTHER（$OTHER 没有值）这样。

5.15.2 解决方案

shell 能够区分这两种情况，忽略冒号（:）表示只希望在变量不存在时才进行替换。如果写成不带冒号的 ${HOME=/tmp}，那么赋值操作仅会在该变量不存在的情况下（从未设置或已明确删除）发生。

5.15.3 讨论

我们再用 $HOME 变量演示一次，但这次不使用冒号。

```
$ echo ${HOME=/tmp}       # 不会被替换
/home/uid002
$ HOME=""                 # 通常并非明智之举
$ echo ${HOME=/tmp}       # 不会被替换

$ unset HOME              # 通常并非明智之举
$ echo ${HOME=/tmp}       # 会被替换
/tmp
$ echo $HOME
/tmp
$
```

这个示例将 $HOME 设置为空字符串，虽然为空，但毕竟也是有效值，因此 = 运算符并不会替换 $HOME 的内容。但如果删除该变量，则会发生替换。要想允许出现空字符串，使用不带冒号的 = 即可。但大部分时候是使用 :=，因为无论你是否有意，空值基本上没什么用处。

5.15.4 参考

- 5.13 节
- 5.14 节
- 表 5-1

5.16 不只使用字符串常量作为默认值

5.16.1 问题

你不想只使用字符串常量作为变量默认值。

5.16.2 解决方案

能出现在 shell 变量引用右侧的内容并不局限于字符串常量。例如：

```
cd ${BASE:="$(pwd)"}
```

5.16.3 讨论

如上所示，用于替换的值并不一定非得是字符串常量。它可以是更为复杂的 shell 表达式的结果，其中包括在子 shell 中运行的命令（如上述示例所示）。如果 $BASE 不存在，那么 shell 会运行内建命令 pwd（以获得当前目录），并使用其返回的字符串。

那么，我们可以在该运算符（以及其他类似运算符）右侧做些什么？bash 手册页对于出现在运算符右侧的内容是这么表述的：“……要经过波浪号扩展、参数扩展、命令替换以及算术扩展”。

具体含义如下。

- **参数扩展**意味着可以使用其他变量，如 `${BASE:=${HOME}}`。
- **波浪号扩展**意味着可以使用 `~bob` 这样的表达式，它会扩展成用户 bob 的主目录。可以通过 `${BASE:=~uid17}` 将默认值设置为用户 uid17 的主目录，但注意不要给 `~uid17` 加引号，因为波浪号扩展不会在引号中执行。
- 该示例用到的就是**命令替换**，它将命令的输出结果作为变量的值，其语法为 `$(cmds)`。
- **算术扩展**意味着可以用 `$(( ))` 语法执行整数算术运算。例如：

```
echo ${BASE:=/home/uid$((ID+1))}
```

5.16.4 参考

- 表 5-1
- 2.15 节
- 5.13 节
- 6.1 节

5.17 对不存在的参数输出错误消息

5.17.1 问题

这些能够设置默认值的便捷写法都挺酷，但有时你需要强制用户指定某个值，否则就无法继续往下进行。用户有可能会遗漏某个参数，因为他们确实不知道该怎样调用脚本。你希望能给用户点提示，省得他们自己瞎猜。相较于堆砌成堆的 `if` 语句，有没有更简洁的方法来检查各个参数？

5.17.2 解决方案

引用参数时使用 `${:?}` 语法，如例 5-6 所示。如果指定参数不存在或为空，那么 bash 会输出错误消息并退出。

例 5-6　ch05/check_unset_parms

```
#!/usr/bin/env bash
# 实例文件：check_unset_parms
#
USAGE="usage: myscript scratchdir sourcefile conversion"
FILEDIR=${1:?"Error. You must supply a scratch directory."}
FILESRC=${2:?"Error. You must supply a source file."}
CVTTYPE=${3:?"Error. ${USAGE}"}
```

如果执行脚本时没有指定足够的参数，则会出现下列结果。

```
$ ./myscript /tmp /dev/null
./myscript: line 7: 3: Error. usage: myscript scratchdir
sourcefile conversion
$
```

5.17.3　讨论

bash 会测试各个参数，如果参数不存在或为空，则输出错误信息并退出。

$3 所对应的错误消息中使用了另一个 shell 变量。你甚至可以在其中执行其他命令。

```
CVTTYPE=${3:?"Error. $USAGE. $(rm $SCRATCHFILE)"}
```

如果 $3 不存在，则错误消息中会包含短语 "Error."、变量 $USAGE 的值以及 rm $SCRATCHFILE 命令的输出。我们先说到这里。你可以让自己的 shell 脚本紧凑得吓人，没错，我们说的就是吓人（awfully）。最好还是多加些空白字符，让代码逻辑更加清晰可读。

```
if [ -z "$3" ]
then
    echo "Error. $USAGE"
    rm $SCRATCHFILE
fi
```

另一方面的考虑：${:?} 生成的错误信息包含 shell 脚本文件名和行号。例如，以下是例 5-6 产生的运行结果片段。

```
$ ./check_unset_parms
./check_unset_parms: line 5: 1: Error. You must supply a scratch
directory.

$ ./check_unset_parms somedir
/tmp/check_unset_parms: line 6: 2: Error. You must supply a source
file.

$ ./check_unset_parms somedir somefile
./check_unset_parms: line 7: 3: Error. usage: myscript scratchdir
sourcefile \
conversion
```

因为无法控制错误消息的这部分内容，而且这看起来好像是 shell 脚本自身出现的错误，再加上可读性的问题，该技术在商业级的 shell 脚本中并不多见。（不过确实有助于调试。）

要想所有变量都具备这种行为，同时又不想逐个改动，可以使用 set -u 命令，“在变量替换时，将不存在的变量视为一种错误”。

```
$ echo "$foo"                          ❶
$ set -u                              ❷
$ echo "$foo"
bash: foo: unbound variable           ❸
$ echo $?   # 退出码
1
$ set +u                              ❹
$ echo "$foo"                         ❺
$ echo $?   # 退出码
0
$
```

❶ 一开始是正常行为

❷ 启用 nounset（-u）

❸ 现在看到了错误信息和故障退出码

❹ 再次关闭 nounset（-u）

❺ 恢复正常行为

5.17.4 参考

- 5.13 节
- 5.14 节
- 5.16 节

5.18 修改部分字符串

5.18.1 问题

你想要重命名一批文件。这些文件的名称基本上正确，就是后缀不对。

5.18.2 解决方案

使用 bash 的参数扩展特性来删除匹配模式的文本，如例 5-7 所示。

例 5-7 ch05/suf?xer

```
#!/usr/bin/env bash
# 实例文件: suffixer
#
# 将文件结尾的.bad修改为.bash

for FN in *.bad
do
    mv "${FN}" "${FN%bad}bash"
done
```

5.18.3 讨论

for 循环迭代当前目录中所有以 .bad 结尾的文件名。变量 $FN 每次都会获得其中一个文件名。在该循环中，mv 命令负责重命名文件（将旧名称**转变成**新名称）。为了避免文件名中包含空格，我们要将其放入引号。

重命名的关键之处是，所引用的 $FN 包含自动删除结尾字符串 bad 的操作。${} 对该引用做了界定，以便与之相邻的字符串 bash 正好追加到已经过删除的文件名结尾。

以下是分解步骤。

```
NOBAD="${FN%bad}"
NEWNAME="${NOBAD}bash"
mv "${FN}" "${NEWNAME}"
```

这样你就能从中看出删除无用后缀、创建新文件名、重命名文件的各个步骤。只要习惯了那些特殊运算符，将这些步骤合并成一行也不算糟糕。

因为我们不仅从变量中删除了子串，还将 bad 替换为 bash，所以也可以将替换运算符 / 用于变量引用。类似于那些用 / 进行搜索替换的编辑器命令（vi 和 sed 中就有），我们也可以按以下方式这么写：

```
# 没有锚定好位置，可别这么做
mv "${FN}" "${FN/bad/bash}"
```

（和编辑器命令不一样的是，我们没有在结尾使用 /，右侧的花括号起到了相同的作用。）

但这种写法有一个问题：没有锚定替换操作的位置，因此可以在变量中的任意位置执行。例如，名为 subaddon.bad 的文件经过替换后就会变成 subashdon.bad，这可不是我们想要的结果。如果将第一个 / 换成 //，则会替换变量中所有指定的内容，最后就成了 subashdon.bash，这也并非我们所愿。最好按以下方式这么写：

```
# 添加"."来"锚定"替换模式的位置。这种写法更好，但也不是万无一失
mv "${FN}" "${FN/.bad/.bash}"
```

解决方案中给出的 `${FN%bad}bash` 已经锚定了位置，只删除字符串尾部的文本，在本例中，这正是我们想要的结果。

对变量中的字符串值执行各种操作的运算符有很多种。表 5-1 总结了这些运算符。

表5-1：字符串操作运算符

${...}内	操作
name:*number*:*number*	从字符串 *name* 的索引位置 *number* 开始，返回长度为 *number* 的子串
#*name*	返回字符串长度
name#*pattern*	从字符串起始位置开始，删除匹配 pattern 的最短子串
name##*pattern*	从字符串起始位置开始，删除匹配 pattern 的最长子串
name%*pattern*	从字符串结束位置开始，删除匹配 pattern 的最短子串
name%%*pattern*	从字符串结束位置开始，删除匹配 pattern 的最长子串
name/*pattern*/*string*	替换字符串中第一次出现的 pattern
name//*pattern*/*string*	替换字符串中出现的所有 pattern

你可以挨个试试。用起来都非常方便。

5.18.4 参考

- man rename

- 12.5 节
- 13.10 节
- 13.15 节
- 5.19 节

5.19 获得某个数的绝对值

5.19.1 问题

变量中的数值可能是负数，也可能是零或正数。你想得到它的大小（也就是绝对值），但 bash 似乎没有求绝对值的功能。

5.19.2 解决方案

利用字符串操作。

```
${MYVAR#-}
```

5.19.3 讨论

这是一种简单的字符串操作。在本例中，# 从字符串起始位置开始搜索负号（-）。如果找到，则将其删除；如果没有找到，就保留原值。不管是哪种情况，最后得到的都是不包含负号的绝对值。

也可以使用 `if/then/else` 按照数学方法来实现。

```
# 何必费事呢?
if (( MYVAR < 0 ))
then
    let MYVAR=MYVAR*-1
fi
```

但就像注释中所说，何必呢？字符串操作简洁有力。但考虑到可读性，也可以加上注释。

```
MYVAR=${MYVAR#-}          # ABS(MYVAR)
```

5.19.4 参考

- 5.18 节
- 表 5-1

5.20 用bash实现basename

5.20.1 问题

basename 命令可以做到你想要的，但是能够在不调用外部命令的情况下得到相同的结果吗？bash 的字符串操作能不能做到？

5.20.2 解决方案

是的，bash 能够从 shell 变量中剥离路径中的目录，只留下最后一部分（文件名）。当你想按以下方式写时：

```
FILE=$(basename $FULLPATHTOFILE)
```

其实只需要写成：

```
FILE=${FULLPATHTOFILE##*/}
```

5.20.3 讨论

这两种写法的不同之处在于花括号。第一种写法使用的是括号（旧式写法是 ``），它会生成一个子 shell 来执行参数值为 $FULLPATHTOFILE 的命令 basename。第二种写法使用的花括号只是 shell 变量替换语法的一部分，并不会生成子 shell，也不会执行命令。它从字符串起始位置开始（因为 #）查找并删除匹配模式 */ 的最长子串（因为 ##）。* 匹配任意数量的字符，/ 仅代表其字面含义。对于路径 /usr/local/bin/mycmd，指定模式会匹配（并删除）/usr/local/bin/，将剩下的 mycmd 赋给变量 $FILE。

因为 basename 命令会忽略路径结尾的 /，所以 $(basename /usr/local/bin/) 会返回 bin，而我们用 bash 所实现的版本则返回空串（因为模式 */ 所能匹配的最长内容就是整个字符串）。为了与 basename 兼容，第一件事就是删除路径结尾的 /。

真正的 basename 命令还可以将要删除的后缀作为第二个可接受的参数。bash 也可以做到，只是需要额外多加一步。所以，对于：

```
FILE=$(basename $MYIMAGEFILE .jpg)
```

更完整的替代方案是：

```
FILE=${MYIMAGEFILE%/}   # 删除结尾的/
FILE=${FILE##*/}        # 删除结尾/之前的所有字符
FILE=${FILE%.jpg}       # 如果存在，删除后缀.jpg
```

5.20.4 参考

- man basename
- 5.18 节
- 5.21 节
- 表 5-1

5.21 用bash实现dirname

5.21.1 问题

dirname 命令的作用和你预想的一样，但就像 basename，它也是一个单独的可执行文件，需要在子 shell 中运行。能不能用字符串操作来代替？

5.21.2 解决方案

当然能。用字符串操作删除文件名（路径中的最后一部分），尽可能多地保留路径中的目录部分。

```
DIR=${MYPATHTOFILE%/*}
```

5.21.3 讨论

如果变量中保存的是 /usr/local/bin/mycmd，我们希望得到的是 /usr/local/bin，同时丢弃最后一部分（文件名）。由于路径中的各个部分是由斜线分隔的，从右侧开始（因为 %）删除匹配模式“斜线之后跟着任意数量字符（/*）”的最短子串（因为只有一个 %，而不是两个）即可。

这个示例演示了如何对 shell 变量进行字符串操作。其中并未给出完整的、可兼容的 dirname 命令替代方案，尤其是针对一些路径名以斜线结尾的边缘情况。

5.21.4 参考

- man dirname，参见其他选项以及细微的差异
- 5.18 节
- 5.20 节
- 表 5-1

5.22 选取CSV的替换值

5.22.1 问题

你想制作一个由逗号分隔的值列表，但不希望开头或结尾处出现逗号。

5.22.2 解决方案

如果在循环内部用 LIST="${LIST},${NEWVAL}" 构建该列表，那么第一次循环（此时 LIST 为空）过后会得到一个前导逗号。你可以对 LIST 进行特殊的初始化处理，以便它在进入循环前就先得到第一个值，但如果觉得这种方法不实用，或是为了避免重复代码（用于得到新值），你可以改用 bash 中的 ${:+} 语法。

```
LIST="${LIST}${LIST:+,}${NEWVAL}"
```

如果 {LIST} 为空或不存在，那么 $LIST 的两个表达式不会产生任何内容。这就意味着，第一次循环过后，LIST 中保存的只有 NEWVAL 的值。如果 LIST 不为空，则第二个表达式 ${LIST:+,} 会被替换为逗号，将先前的值与新值分隔开来。

下面的代码片段用于读取并构建 CSV 列表。

```
#
# 一次读取一个值，构建以逗号分隔的值列表
#
while read NEWVAL
do
    LIST="${LIST}${LIST:+,}${NEWVAL}"
done
echo $LIST
```

5.22.3 参考

- 5.18 节
- 表 5-1

5.23 使用数组变量

5.23.1 问题

到目前为止，你已经见识了不少使用变量的脚本，但是 bash 能不能处理数组呢？

5.23.2 解决方案

当然能。bash 有专门的一维数组语法。

5.23.3 讨论

如果编写脚本时已经知道具体的值，则初始化数组就很容易了。格式如下：

```
MYRA=(first second third home)
```

括号内列表的每个单词都对应着一个数组元素。你可以像下面这样引用各个元素：

```
echo runners on ${MYRA[0]} and ${MYRA[2]}
```

输出结果如下：

```
runners on first and third
```

如果只写 $MYRA，那么只会得到第一个数组元素，相当于 ${MYRA[0]}。

5.23.4 参考

- Cameron Newham 所著的 *Learning the bash Shell, 3rd Edition*（O'Reilly 出版），157~161 页，其中讲述了有关数组的更多信息
- 7.15 节
- 13.4 节

5.24 转换大小写

5.24.1 问题

数码相机里全是一堆大写字母命名的文件，如 IMG0001.JPG。你想将文件名改成小写，但又不愿意挨个重新输入。

5.24.2 解决方案

bash 4.0 中的几个运算符可以在引用变量名时转换其大小写。如果变量 $FN 中包含一个需要转换成小写的文件名（字符串），那么 ${FN,,} 会返回全部是小写形式的字符串。与此类似，${FN^^} 会返回全部是大写形式的字符串。甚至还有 ${FN~~}，它可以切换大小写，将所有的小写字母转换成大写，大写字母转换成小写。

以下的 for 循环会将所有 .JPG 类型的文件名更改成小写字母。

```
for FN in *.JPG
do
    mv "$FN" "${FN,,}"
done
```

或者写成单行脚本：

```
for FN in *.JPG; do mv "$FN" "${FN,,}" ; done
```

bash 4 或更高版本中还有另外一种方法：你可以将变量声明成始终为小写字母的类型。赋给该变量的任何文本都会转换成小写。要想在 for 循环中使用这种方法来重命名文件，使用简单的赋值即可，无须任何字符串操作。

```
declare -l lcfn      # 变量内容会转换成小写字母
for FN in *.JPG
do
    lcfn="$FN"
    mv "$FN" "$lcfn"
done
```

还有一些类似的变量声明，它们可以更改所有字母或首字母的大小写。以下是一个简单的演示程序，其中展示了这类声明的用法。

```
declare -u UP     # 全部大写
declare -l dn     # 全部小写
```

```
declare -c Ca    # 仅首字母大写

while read TXT
do
    UP="${TXT}"
    dn="${TXT}"
    Ca="${TXT}"

    echo $TXT  $UP  $dn  $Ca
done
```

如果变量是通过 -c 声明的，即便字符串中包含多个单词，也只有首字母大写。你可以尝试运行并观察其如何工作。

5.24.3 参考

- man rename
- 5.25 节

5.25 转换为驼峰命名法

5.25.1 问题

你希望字符串中每个单词的首字符都是大写，而不仅限于首字母。

5.25.2 解决方案

结合使用数组与大小写转换运算符。

```
while read TXT
do
    RA=($TXT)          # 一定得是($，不要写成$(
    echo ${RA[@]^}
done
```

5.25.3 讨论

`$TXT` 周围的括号令其被视为数组初始化。文本中分隔单词的空白字符划分了数组元素。`[@]` 语法一次性引用数组中的所有元素，`^` 操作符将每个元素的首字符转换为大写。

5.25.4 参考

- 5.24 节

第 6 章　shell逻辑与算术

相较于最初的 Bourne shell，现代 bash 版本的最大改进之一体现在算术方面。早期的 shell 版本没有内建的算术功能，哪怕是给变量加1，也得调用单独的程序来完成。尽管算术机制如此不堪，但很多任务中仍旧少不了 shell 的身影，从某种程度上来说，这也反衬出 shell 以往的强大功用（现在依然如此）。过了一段时间以后，我们明显发现要有一种直观的语法，用以实现自动化重复性任务所需要的简单计数功能。正是早期 Bourne shell 在此项上的缺失成就了 csh，当时后者在 shell 编程中引入了类似于 C 语言的语法，其中就包括数字类型变量。好吧，过去是过去，现在是现在。如果有一阵子没了解过 bash shell 的算术功能，如今你会感到非常惊讶。

除了算术，还有程序员都很熟悉的控制结构。有用于条件判断的 if/then/else 结构，以及 while 循环和 for 循环，你也会在其中发现一些 bash 的奇特之处。case 语句的字符串模式匹配能力令其具备了异常强大的功能，另外，select 用起来也挺古怪。讨论过这些特性之后，我们将利用它们构建两个简单的命令行计算器，以此为本章作结。

6.1　在shell脚本中执行算术操作

6.1.1　问题

你需要在 shell 脚本中执行一些简单的算术操作。

6.1.2　解决方案

用 $(( )) 或 let 进行整数运算。例如：

```
COUNT=$((COUNT + 5 + MAX * 2))
let COUNT+='5+MAX*2'
```

6.1.3 讨论

只要都是整数运算，就可以在 $(( )) 的算术表达式内使用所有的标准（与 C 语言类似）运算符。这里还有一个额外的运算符：可以用 ** 进行幂运算，如 MAX=$((2**8))，结果为 256。

$(( )) 表达式内不需要使用空格，不过在运算符和操作数两边加上空格也无妨（但 ** 必须写在一起）。但是 = 两边绝不能出现空格，这和 bash 变量赋值的规则一样。另外，确保给 let 的表达式加上引号，因为 let 语句是 bash 内建的，其参数要经过单词扩展。

不要在赋值号（=）两边放置空格！如果你按以下方式写：

```
COUNT = $((COUNT+5))     # 可不像你想的那样!
```

那么，bash 会尝试运行一个名为 COUNT 的程序，其第一个参数为 =，第二个参数为 $COUNT 与 5 之和。记住，别在赋值号两边加空格！

另一个怪异之处是，通常出现在 shell 变量前表示取值的 $ 符号（如 $COUNT 或 $MAX）在双括号内部是不需要的。例如，我们可以写：

```
$((COUNT + 5 + MAX * 2))
```

在上述示例中，shell 变量前并没有 $ 符号，实际上，外部的 $ 应用于整个表达式。但如果用到了位置参数（如 $2），那么 $ 还是少不了的，因为只有这样才能区分位置参数与数字常量（如 2）。以下是一个示例。

```
COUNT=$((COUNT + $2 + OFFSET))
```

用 bash 内建的 let 语句对 shell 变量执行整数运算时，也涉及类似的问题。let 用到的算术运算符和 $(( )) 一样。

```
let COUNT=COUNT+5
```

其中有一些花哨的赋值运算符可供使用，如下所示（它将完成与前一行相同的操作）。

```
let COUNT+=5
```

（C/C++ 和 Java 程序员应该不会陌生。）这个示例给变量 COUNT 的现有值加 5，而且不需要重复书写变量名。

表 6-1 列出了这些特殊的赋值运算符。

表6-1：bash中的赋值运算符

运算符	赋值操作	用法	含义
=	简单的赋值	a=b	a=b
*=	乘法	a*=b	a=(a*b)
/=	除法	a/=b	a=(a/b)
%=	求余	a%=b	a=(a%b)
+=	加法	a+=b	a=(a+b)
-=	减法	a-=b	a=(a-b)
<<=	按位左移	a<<=b	a=(a<<b)
>>=	按位右移	a>>=b	a=(a>>b)
&=	按位“与”	a&=b	a=(a&b)

运算符	赋值操作	用法	含义
^=	按位“异或”	a^=b	a=(a^b)
|=	按位“或”	a|=b	a=(a|b)

这些赋值操作符也可以在 $(( )) 内使用。最外层的赋值仍旧是普通的 shell 变量赋值。

也可以用逗号运算符形成级联赋值。

```
echo $(( X+=5 , Y*=3 ))
```

该表达式执行两次赋值操作，然后由 echo 显示出第二个子表达式的结果（因为逗号运算符返回其第二个操作数的值）。如果不想显示结果，更常见的做法是使用 let 语句。

```
let   X+=5 Y*=3
```

这里不需要逗号运算符，因为 let 语句中的每个单词本身就代表单独的算术表达式。

let 语句和 $(( )) 语法的另一处重要区别在于两者处理空白字符（空格字符）的方式不同。对 let 语句来说，要么添加引号，要么赋值运算符（=）和其他运算符两边不能出现空格。必须将运算符和操作数放在一起形成一个单词。以下两种写法都没问题。

```
let i=2+2
let "i = 2 + 2"
```

$(( )) 语法就宽松多了，它允许各种空白字符出现在双括号内。这种写法不易出错，代码的可读性也要好得多，是我们执行 bash 整数运算时的首选方式。偶尔用到 += 赋值或 ++ 运算符，抑或是怀念早期的 BASIC 编程（该语言也有 LET 语句）时，也会存在例外。

记住，这是整数运算，不是浮点运算。别对 2/3 这样的表达式抱太大希望，它在整数运算中的计算结果为 0。除法是整数除法，它会舍去任何小数结果。

6.1.4 参考

- help let
- bash 手册页

6.2 条件分支

6.2.1 问题

你想要检查参数数量是否正确并执行相应的操作。这得用到条件分支。

6.2.2 解决方案

bash 中的 if 语句表面上和其他语言中的差不多。

```
if [ $# -lt 3 ]
then
    printf "%b" "Error. Not enough arguments.\n"
    printf "%b" "usage: myscript file1 op file2\n"
    exit 1
fi
```

或者：

```
if (( $# < 3 ))
then
    printf "%b" "Error. Not enough arguments.\n"
    printf "%b" "usage: myscript file1 op file2\n"
    exit 1
fi
```

以下是一个带有 elif（bash 中的 else-if）和 else 子句的完整 if 语句。

```
if (( $# < 3 ))
then
    printf "%b" "Error. Not enough arguments.\n"
    printf "%b" "usage: myscript file1 op file2\n"
    exit 1
elif (( $# > 3 ))
then
    printf "%b" "Error. Too many arguments.\n"
    printf "%b" "usage: myscript file1 op file2\n"
    exit 2
else
    printf "%b" "Argument count correct. Proceeding...\n"
fi
```

甚至可以写成：

```
[ $result = 1 ] \
  && { echo "Result is 1; excellent." ; exit 0;   } \
  || { echo "Uh-oh, ummm, RUN AWAY! " ; exit 120; }
```

（有关最后一个示例的讨论，参见 2.14 节。）

6.2.3 讨论

我们有两件事需要讨论：if 语句的基本结构和 if 表达式的不同语法（括号或方括号，运算符或选项）。前者也许有助于解释后者。按照 bash 手册页中的描述，if 语句的一般形式如下所示。

```
if list; then list; [ elif list; then list; ] ... [ else list; ]
fi
```

[和] 用于划分语句中的可选部分（例如，有些 if 语句中就没有 else 子句）。我们先来看看不带任何可选部分的 if 语句。

最简单的 if 语句形式如下所示。

```
if list; then list; fi
```

在 bash 中，和换行符一样，分号的作用也是结束某个语句。我们可以用分号将解决方案部分中的示例塞进更少的行中，但使用换行符的可读性更好。

then *list* 的存在看起来是有意义的，其中的语句在 if 条件为真的情况下执行（我们也可以从其他编程语言中猜测出来）。但是，if *list* 算是怎么回事？难道不应该是 if *expression* 吗？

没错，但这是 shell，一个命令处理器。它的主要任务就是执行命令。因此，if 后面的 *list* 就是放置命令列表的地方。你可能会问，决定分支走向（then 子句或 else 子句）的是什么呢？答案是 *list* 中最后一个命令的返回值。（你可能还记得，返回值可以从变量 $? 中获取。）

我们通过一个有点奇怪的示例来说明这一点。

```
$ cat trythis.sh
if ls; pwd; cd $1;
then
    echo success
else
    echo failed
fi
pwd

$ bash ./trythis.sh /tmp
...
$ bash ./trythis.sh /nonexistent
...
$
```

在这个奇怪的脚本中，shell 会在选择分支前执行 3 个命令（ls、pwd、cd），其中 cd 命令的参数是调用该脚本时所提供的第一个命令行参数。如果没有提供参数，那就只执行 cd，返回到主目录中。

结果会是怎样？你可以自己试试。最终是显示“success”还是“failed”，取决于 cd 命令是否执行成功。在示例中，cd 是 if 语句命令列表中的最后一个命令。如果 cd 执行失败，就转到 else 子句；但如果执行成功，则选择 then 子句。

如果命令书写正确，同时执行过程中没有出现错误，那么返回值为 0。如果有错误（如参数有问题、I/O 错误、找不到指定文件），则返回非 0 值（不同的错误种类通常对应不同的值）。

这就是为什么 shell 脚本编写人员和 C（以及其他语言）程序员要确保退出脚本和程序时返回有意义的值，这一点很重要。有些 `if` 语句可能依赖于此。

那么我们该如何从这个奇怪的 `if` 构造中找出与真实 `if` 语句（经常在程序中看到的那种）相似的地方？这则实例开头部分的那些示例是怎么回事？毕竟它们看起来并不像是语句列表。

我们来看看以下这段代码。

```
if test $# -lt 3
then
    echo try again.
fi
```

就算不是命令列表，但有没有从中看出起码类似于单个 shell 命令（内建命令 `test` 接受参数并比较参数值）的东西？如果比较结果为真，那么 `test` 命令会返回 0，否则返回 1。你可以自己动手在命令行上尝试一下这个命令，然后用 `echo $?` 查看返回值。

我们给出的第一个示例中开头的 `if [ $# -lt 3 ]` 看起来很像 `test` 命令。这是因为 `[` 其实只是相同命令的不同名称而已。（出于可读性和美观方面的考虑，调用 `[` 时还要求将 `]` 作为最后一个参数。）因此，对于该语法，`if` 语句中的表达式其实就是一个只包含单个命令（`test` 命令）的列表。

> 在早期的 Unix 中，`test` 是一个独立的可执行文件，`[` 只是指向该文件的链接。现在两者仍以可执行文件的形式存在，但 bash 也将它们实现为内建命令。[1]

[1]两者的可执行文件分别位于 /bin/test 和 /bin/[。将其实现为内建命令是为了提高执行效率。——译者注

那么 `if (( $# < 3 ))` 又是什么意思？双括号是**复合命令**的一种。因为它会对其中的算术表达式求值，所以能在 `if` 语句中派上用场。这是一处比较新的 bash 改进 [2]，专门用于有 `if` 语句的场合。

[2]仅适用于 bash 2.0 之后的版本。——译者注

可用于 `if` 语句的这两种语法之间的重要区别在于测试的表达方式及其能够测试的对象种类。双括号仅限于算术表达式，方括号还可以测试文件特性，但后者的算术测试语法远不如前者方便，尤其是用括号将表达式划分成若干子表达式时（在方括号中，需要给括号加上引号或将其转义）[3]。

[3]如 `[ \( 3 -gt 2 \) -o \( 4 -le 1 \) ]`。这是因为括号在 shell 中属于元字符。——译者注

6.2.4 参考

- `help if`
- `help test`
- `man test`
- 2.14 节
- 4.4 节
- 6.3 节
- 6.5 节
- 15.11 节

6.3 测试文件特性

6.3.1 问题

为了提高脚本的稳健性，你希望在读取输入文件前先检查该文件是否存在；另外，还想在写入输出文件前确认其是否具备写权限，在用 `cd` 切换目录前看看到底有没有这个目录。这些该如何在 bash 脚本中实现呢？

6.3.2 解决方案

在 if 语句的 test 命令部分使用各种文件特性测试。可以使用类似于例 6-1 的脚本来解决你的问题。

例 6-1 ch06/check?e

```
#!/usr/bin/env bash
# 实例文件：checkfile
#
DIRPLACE=/tmp
INFILE=/home/yucca/amazing.data
OUTFILE=/home/yucca/more.results

if [ -d "$DIRPLACE" ]
then
    cd $DIRPLACE
    if [ -e "$INFILE" ]
    then
        if [ -w "$OUTFILE" ]
        then
            doscience < "$INFILE" >> "$OUTFILE"
        else
            echo "cannot write to $OUTFILE"
        fi
    else
        echo "cannot read from $INFILE"
    fi
else
    echo "cannot cd into $DIRPLACE"
fi
```

6.3.3 讨论

我们将各种文件名引用全都放入了引号，以防路径名中包含空格。这个示例并不存在这种情况，但如果修改脚本，使用其他路径名，那可就不一定了。

在测试另外两种条件前，我们先测试并执行了 cd 命令。在这个示例中，测试顺序并不重要，但如果 $INFILE 或 $OUTFILE 中保存的是相对路径（不是以根目录开始，即 / 开头的路径），同样的测试

也许在 cd 之前为真，之后为假，反过来也有可能。因此，我们要在使用文件之前测试。

示例用双大于号运算符（>>）将输出追加到结果文件中，而不是用新内容替换原有文件内容。

可以用 -a（读作“and”）运算符将多个测试组合成一个更长的 if 语句，但如果测试失败，你也看不到特别有帮助的错误消息，因为根本就不知道是哪部分测试没有通过。

我们也可以测试一些别的文件特性，其中有 3 个特性要用到双目运算符（接受两个文件名）。

FILE1 -nt *FILE2*

是否更新（检查文件的修改时间）。现有文件要比不存在的文件“新”。

FILE1 -ot *FILE2*

是否更旧。同样，不存在的文件要比现有文件“旧”。

FILE1 -ef *FILE2*

具有相同设备和 inode 编号（即便由不同链接所指向，也视为相同的文件）。

表 6-2 展示了与文件相关的其他测试（更完整的列表参见 A.9 节）。表中全都是单目运算符，其形式为 *option filename*，例如，if [-e myfile]。

表6-2：检查文件特性的单目运算符

运算符	描述
-b	块设备文件（如 /dev/hda1）

运算符	描述
-c	字符设备文件（如 /dev/tty）
-d	目录文件
-e	文件存在
-f	普通文件
-g	文件设置了 set-group-ID（setgid）位
-h	符号链接文件（等同于 -L）
-G	有效组 ID（effective group ID）拥有的文件
-k	文件设置了粘滞位
-L	符号链接文件（等同于 -h）
-N	文件自上次读取后被修改过
-O	有效用户 ID（effective user ID）拥有的文件
-p	具名管道文件
-r	可读文件
-s	文件大小不为空

运算符	描述
-S	套接字文件
-u	文件设置了 set-user-ID（setuid）位
-w	可写文件
-x	可执行文件

6.3.4 参考

- 2.10 节
- 4.6 节
- A.9 节

6.4 测试多个特性

6.4.1 问题

该怎么测试多个特性？必须嵌套 if 语句吗？

6.4.2 解决方案

使用 -a（逻辑与）和 -o（逻辑或）运算符将多个测试条件组合成一个表达式。例如：

```
if [ -r $FILE -a -w $FILE ]
```

该 if 语句会测试指定文件是否可读且可写。

6.4.3 讨论

因为所有的文件测试条件都隐含了该文件存在的测试，所以测试文件可读性时不用测试文件是否存在。如果文件不存在，自然也就不可读。

这些逻辑运算符（-a 表示 AND，-o 表示 OR）可用于所有的测试条件，并不局限于文件测试。

同一个语句中可以出现多个 AND/OR。你可能要用括号来获得正确的优先级，比如 a and (b or c)，但一定要记得在括号前加上反斜杠或将括号放进引号，以消除其特殊含义。不过，可别让整个表达式都出现在引号中，这会令其作为整体被当成是对空字符串的测试（参见 6.5 节）。以下这个测试更为复杂，其中的括号已经过正确转义。

```
if [ -r "$FN" -a \( -f "$FN" -o -p "$FN" \) ]
```

不要认为这些子表达式的求值顺序和 Java 或 C 中的顺序一样。在后两种语言中，如果 AND 表达式的第一部分为假（或者 OR 表达式的第一部分为真），则不会再对第二部分求值（我们称之为**表达式短路**）。但是，因为 shell 会在准备对表达式求值时对语句进行好几趟处理（如执行参数替换等），所以由逻辑运算符连接在一起的两部分可能会被部分求值。尽管在这个简单示例中不会有什么问题，但碰上更复杂的情况，那可就未必了。例如：

```
if [ -z "$V1" -o -z "${V2:=YIKES}" ]
```

如果 $V1 为空，那就完全不用再对 if 语句中的第二个条件（检查 $V2 是否为空）求值了，但 $V2 的值也许已经被修改了（对 $V2 进行变量替换的副作用）。变量替换发生在 -z 测试之前。是不是被搞晕了？别去依赖条件表达式短路就行了。如果的确需要这种行为，可以将 if 语句拆成两个嵌套 if 语句，或者使用 && 和 ||。

6.4.4 参考

- 6.5 节
- 附录 C，可参见有关命令行处理的更多信息

6.5 测试字符串特性

6.5.1 问题

你希望在使用字符串前先检查一下它们的值。这些字符串可以是用户输入、读入的文件或传入脚本的环境变量。如何用 bash 脚本实现呢？

6.5.2 解决方案

你可以在 `if` 语句中使用单方括号形式的 `test` 命令进行一些简单的测试，其中包括检查变量是否包含文本以及两个变量中的字符串是否相同。

6.5.3 讨论

我们来看看例 6-2。

例 6-2 ch06/checkstr

```
#!/usr/bin/env bash
# 实例文件：checkstr
#
# if语句测试变量中的字符串长度是否为0
#
# 使用命令行参数
VAR="$1"
#
# if [ "$VAR" ]这种形式通常也管用，但并不是一种好的写法，加上-n会更清晰
if [ -n "$VAR" ]
then
    echo has text
else
    echo zero length
fi
#
if [ -z "$VAR" ]
then
    echo zero length
else
```

```
    echo has text
fi
```

我们特意使用了“长度是否为 0”这个短语。长度为 0 的变量有两种：设置为空串的变量和不存在的变量。示例中的测试并不区分这两种情况。它只关心变量中是否有字符存在。

重要的是要将 $VAR 放进引号，否则测试会被一些怪异的用户输入干扰。如果 $VAR 的值是 x -a 7 -lt 5 且没有使用引号，那么下列语句：

```
if [ -z $VAR ]
```

就会变成（在变量扩展之后）：

```
if [ -z x -a 7 -lt 5 ]
```

这是一种更为复杂的测试语法，完全有效，但结果可就不是你想要的了（也就是说，它测试的并非是字符串长度是否为 0）。

6.5.4 参考

- 6.7 节
- 6.8 节
- 14.2 节
- A.9 节

6.6 测试等量关系

6.6.1 问题

你想要检查两个 shell 变量是否相等，但是存在两种测试运算符：-eq 和 =（或 ==）。该用哪个呢？

6.6.2 解决方案

你需要的比较类型决定了该用哪种运算符。如果是进行数值比较，可以使用 -eq 运算符；如果是进行字符串比较，则使用 =（或 ==）运算符。

6.6.3 讨论

例 6-3 是一个演示了这两种情况的简单脚本。

例 6-3 ch06/strvsnum

```
#!/usr/bin/env bash
# 实例文件：strvsnum
#
# 老生常谈的字符串与数值比较
#
VAR1=" 05 "
VAR2="5"
printf "%s" "do they -eq as equal? "
if [ "$VAR1" -eq "$VAR2" ]
then
    echo YES
else
    echo NO
fi

printf "%s" "do they = as equal? "
if [ "$VAR1" = "$VAR2" ]
then
    echo YES
else
    echo NO
fi
```

如果运行该脚本，我们会得到：

```
$ bash strvsnum
do they -eq as equal? YES
do they = as equal? NO
$
```

尽管两个变量的数值相等（5），但从字符角度来看，前导字符 0 和空白字符意味着这两个字符串并不相同。

= 和 == 都可以使用，但 = 符合 POSIX 标准，可移植性更好。

如果认识到 -eq 运算符类似于 FORTRAN 中的 .eq. 运算符，也许有助于你记忆采用哪种比较。（FORTRAN 是一种用于科学计算的语言，非常倚重数值处理。）实际上，bash 还有其他一些数值比较运算符，都类似于古老的 FORTRAN 运算符。表 6-3 已经将其全部列出，这些缩写非常好记，很容易就能猜出含义。

表6-3：bash的比较运算符

数值比较运算符	字符串比较运算符	含义
-lt	<	小于
-le	<=	小于或等于
-gt	>	大于
-ge	>=	大于或等于
-eq	=，==	等于
-ne	!=	不等于

另一种记忆方法是“逆反法”：将看似字符串一样的比较运算符（采用的是字符写法，如 -eq）用于数值比较，将看似比较数值的运算符（如数学中用到的 +<=+）用于字符串比较。

这和 Perl 正好相反，在后者中，eq、ne 等属于字符串比较运算符，而 ==、!= 等属于数值比较运算符。

可能最好的办法就是始终在 (( )) 语法中进行数值测试，在 [[ ]] 语法中进行字符串比较。这样你就可以在比较时一直使用数学形式的

符号了。

6.6.4 参考

- 6.7 节
- 6.8 节
- 14.12 节
- A.9 节

6.7 用模式匹配进行测试

6.7.1 问题

你不想对字符串进行字面匹配，而是想看看它是否符合某种模式。例如，想知道是否存在 JPEG 类型的文件。

6.7.2 解决方案

在 `if` 语句中使用复合命令 `[[ ]]`，以便在等量运算符右侧启用 shell 风格的模式匹配：

```
if [[ "${MYFILENAME}" == *.jpg ]]
```

6.7.3 讨论

`[[ ]]` 语法不同于 `test` 命令的老形式 `[`，它是一种较新的 bash 机制（2.01 版左右才出现）。能够在 `[ ]` 中使用的运算符也可用于 `[[ ]]`，但在后者中，等号是一种更为强大的字符串比较运算符。就像我们一样，你也可以使用 `=` 或 `==`，两者在语义上相同。我们偏好 `==`（尤其是进行模式匹配时），这样更加醒目，但模式匹配功能可不是因此而有的，它源于复合命令 `[[ ]]`。

标准模式匹配包括 `*`（匹配任意数量的字符）、`?`（匹配单个字符）以及 `[ ]`（匹配字符列表中的任意一个）。注意，其写法类似于

shell 文件通配符，但和正则表达式可不是一回事。

如果进行模式匹配，就别把模式放入引号。否则，就只能匹配到以星号为首的字符串 [4]。

[4]也就是说，按照模式字符串的字面意义进行匹配。——译者注

通过启用某些 bash 选项，可以使用一些更为强大的模式匹配功能。我们来扩展一下之前的例子，要求其查找以 .jpg 或 .jpeg 结尾的文件名。可以这么做：

```
shopt -s extglob
if [[ "$FN" == *.@(jpg|jpeg) ]]
then
    # 余下的处理
```

shopt -s 命令可以打开 shell 选项。extglob 选项涉及扩展模式匹配（或通配符匹配）。借助扩展模式匹配，我们可以使用多个模式，彼此之间用 | 字符分隔并通过括号分组。括号之前的 @ 字符表示仅匹配括号中的模式一次。表 6-4 列出了所有的扩展模式匹配。

表6-4：扩展模式匹配

扩展模式	含义
@(...)	仅匹配 1 次
*(...)	匹配 0 次或多次
+(...)	匹配 1 次或多次
?(...)	匹配 0 次或 1 次
!(...)	匹配除此之外的任何模式

匹配区分大小写，不过可以用 shopt -s nocasematch（适用于 bash 3.1 以上版本）改变这种行为。该选项会影响到 case 和 [[命令。

6.7.4 参考

- 14.2 节
- 16.9 节
- A.8 节
- A.14 节
- A.15 节

6.8 用正则表达式测试

6.8.1 问题

有时候，即便是 extglob 选项所启用的扩展模式匹配也不够用。你真正需要的是正则表达式。假设你将一张古典音乐 CD 翻录到了目录中，浏览该目录会发现下列文件名：

```
$ ls
Ludwig Van Beethoven - 01 - Allegro.ogg
Ludwig Van Beethoven - 02 - Adagio un poco mosso.ogg
Ludwig Van Beethoven - 03 - Rondo - Allegro.ogg
Ludwig Van Beethoven - 04 - "Coriolan" Overture, Op. 62.ogg
Ludwig Van Beethoven - 05 - "Leonore" Overture, No. 2 Op. 72.ogg
$
```

你想写一个脚本，把这些文件名改得简单些，比如只保留曲目编号。该怎么实现呢？

6.8.2 解决方案

使用 =~ 运算符进行正则表达式匹配。只要能够匹配到某个字符串，就可以在 shell 数组变量 $BASH_REMATCH 中找到模式中的各个部分所匹配到的内容。例 6-4 是脚本中负责模式匹配的部分。

例 6-4　ch06/trackmatch

```
#!/usr/bin/env bash
# 实例文件：trackmatch
#
for CDTRACK in *
do
    if [[ "$CDTRACK" =~ "([[:alpha:][:blank:]]*)- ([[:digit:]]*) -
(.*)$" ]]
    then
        echo Track ${BASH_REMATCH[2]} is ${BASH_REMATCH[3]}
        mv "$CDTRACK" "Track${BASH_REMATCH[2]}"
    fi
done
```

这要求使用 bash 3.0 或更高版本，老版中没有 =~ 运算符。另外，bash 3.2 统一了条件运算符 == 和 =~ 中的模式处理，但同时有一处细微的引用 bug，后来在 3.2 的补丁 #3 中得以纠正。如果此处给出的解决方案无效，可能是你使用的 bash 3.2 没有安装这个补丁，需要升级到更高版本。你也可以使用另一个可读性较差的写法来避开这个 bug：删除正则表达式两边的引号，逐个转义其中每个括号和空格字符。但这么一来，整个正则表达式很快就会变得丑陋不堪。

```
if [[ "$CDTRACK" =~ \([[:alpha:][:blank:]]*\)-\ \
> \([[:digit:]]*\)\ -\ \(.*\)\$ ]]
```

6.8.3　讨论

如果熟悉 sed、awk 以及老版本 shell 中的正则表达式，你也许会注意到它们与示例中的新写法有少许不同。最明显的是 [:alpha:] 这样的字符类以及用于分组的括号不需要转义，我们并没有像在 sed 中那样写成 \(。这里 \(表示字面意义上的括号。

bash 内建的数组变量 $BASH_REMATCH 的各个元素由括号中的每个子表达式产生。第 0 个元素（${BASH_REMATCH[0]}）是正则表达式所匹配的整个字符串。子表达式可以通过 ${BASH_REMATCH[1]}、${BASH_REMATCH[2]} 等形式引用。只要将正则表达式写成这样，就会生成变量 $BASH_REMATCH。由于其他 bash 函数也许会使用正则表达式进行匹配，因此你可能需要尽快

将该变量另存起来，以备后用。这个示例立刻在 if 的 then 子句中使用了该值，所以就不用再将其保存到别处了。

因为实在是难以解读，所以正则表达式往往被描述为**只写**表达式。我们通过分步的方法来展示如何从头构建一个完整的正则表达式。示例涉及的文件名的一般结构如下所示：

```
Ludwig Van Beethoven - 04 - "Coriolan" Overture, Op. 62.ogg
```

其中包括：作曲人姓名、曲目编号、曲目名称以及结尾的 .ogg（文件保存为 Ogg Vorbis 格式，这种格式体积更小，保真度更高）。

表达式左边是一个开括号（或者左括号）。这表示第一个子表达式开始。我们要在其中匹配文件名的第一部分，也就是作曲人姓名，这部分以粗体标出：

```
([[:alpha:][:blank:]]*)- ([[:digit:]]*) - (.*)$
```

作曲人姓名由任意数量的字母和空白字符组成。我们用方括号将组成姓名的各种字符划分在一起。其中没有使用 [a-zA-Z] 的写法，而是采用了 POSIX 字符组 [:alpha:] 和 [:blank:]，并将它们放入方括号。随后的星号表示 0 次或多次重复。右括号关闭了第一个子表达式，接下来是一个字面连字符和空白字符。

第二个子表达式（标记为粗体）会尝试匹配曲目编号。

```
([[:alpha:][:blank:]]*)- ([[:digit:]]*) - (.*)$
```

这个子表达式以另一个左括号起始。曲目编号为整数，由多个数位组成（字符类 [:digit:]），我们将其写入另一对方括号并在后面加上一个星号，也就是 [[:digit:]]*，以此表示方括号中的内容（数位）重复出现 0 次或多次。接着是字面空白字符、连字符、空白字符。

最后一个子表达式匹配包括曲目名、文件扩展名在内的所有剩余内容。

```
([[:alpha:][:blank:]]*)- ([[:digit:]]*) - (.*)$
```

括号中的 .* 是一种常见的正则表达式，并不陌生，它表示任意字符（.）出现任意多次（*）。我们用美元符号结束整个表达式，以匹配字符串结尾。匹配过程区分大小写，但是可以用 shopt -s nocasematch（bash 3.1 版本以上可用）修改这种行为。该选项会影响到 case 和 [[命令。

6.8.4 参考

- 有关正则表达式库的细节，详见 man regex（Linux、Solaris、HP-UX）或者 man re_format（BSD、Mac）
- Jeffrey E. F. Friedl 所著的 *Mastering Regular Expressions, 3rd Edition*（O'Reilly 出版）
- 7.7 节
- 7.8 节
- 19.15 节

6.9 用重定向改变脚本行为

6.9.1 问题

正常情况下，无论输入是来自键盘或文件，还是输出到屏幕或文件，你都希望脚本能保持相同的行为。但如果偶尔想对此进行区分，该如何在脚本中实现呢？

6.9.2 解决方案

可以在 if 语句中用 test -t 0 对两种行为做出选择。0 是标准输入的文件描述符；测试标准输出重定向则使用 1。如果文件描述符连接到终端，测试结果就为真；否则，测试结果则为假（例如，重定向到文件或通过管道连接到另一个程序）。

6.9.3 讨论

在这么做之前，一定要三思。bash 脚本编程中的大量功能和灵活性都源于脚本能够通过管道彼此相连这一事实。当输入或输出被重定向时，要确保你有足够充分的理由让脚本做出这么奇怪的处理。

6.9.4 参考

- 2.18 节
- 2.19 节
- 2.20 节
- 10.1 节
- 15.9 节
- 15.12 节
- A.10 节

6.10 循环一段时间

6.10.1 问题

你希望在符合条件的情况下重复执行某些操作。

6.10.2 解决方案

对于算术条件，使用 `while` 循环：

```
while (( COUNT < MAX ))
do
    some stuff
    let COUNT++
done
```

对于文件系统相关的条件：

```
while [ -z "$LOCKFILE" ]
do
    some things
done
```

对于读取输入：

```
while read lineoftext
do
    process $lineoftext
done
```

6.10.3 讨论

第一个 while 语句中的双括号界定了算术表达式，这很像 shell 变量赋值中用到的 $(( ))（参见 6.1 节）。双括号内出现的变量名表示取值。也就是说，不需要写成 $VAR，直接在括号中使用 VAR 就行了。

while [-z"$LOCKFILE"] 中的方括号和 if 语句中的一样，等同于使用 test 命令。

最后的示例 while read lineoftext 没有用到任何括号、方括号或花括号。在 bash 中，while 语句的语法是这么定义的：该语句的条件是一系列要执行的命令（就像 if 语句），最后一个命令的退出状态决定了条件是真还是假。退出状态为 0，表示真；否则，表示假。

如果顺利读取，read 返回 0；如果读到文件末尾，则返回 1。这意味着，对于前者，while 视其为真；对于后者，while 视其为假并停止循环。然后接着执行 done 之后的语句。

“当命令返回 0 时继续循环”这种逻辑似乎有点颠倒了，因为大多数类 C 语言采用的是与此相反的逻辑，也就是“非 0 时循环”。但在 shell 中，为 0 的返回值表示一切顺利；而非 0 的返回值则表示错误退出。

(( )) 也是如此。shell 会对其中的表达式求值，如果结果为非 0，那么 (( )) 就返回 0；如果结果为 0，则返回 1。这意味着我们可以像 Java 或 C 程序员那些书写表达式，但 while 语句沿用的仍旧是 bash 那一套，视 0 为真。

实际上，这意味着我们可以编写一个无限循环：

```
while (( 1 )); do
    ...
done
```

对 C 程序员而言，这种写法“感觉是对的”。但要记住，while 语句查找的是为 0 的返回值，这里它确实可以得到，因为对于为真（也就是非 0）的结果，((1)) 返回 0。

结束 while 循环的相关讨论之前，我们再来看看 while read 这个示例，它目前从标准输入（键盘）中读取输入信息，要想令其从文件中读取，该如何修改代码呢？

通常有 3 种实现方法。第一种完全不需要改动代码。只需要在调用脚本时将标准输入重定向到文件。

```
myscript < file.name
```

要是不想把这个工作留给调用脚本的用户呢？如果知道待处理的文件名，或者该文件要作为脚本的命令行参数，那就可以保持 while 循环不变，将其输入重定向到该文件。

```
while read lineoftext
do
    process that line
done < file.input
```

第三种方法是用 cat 命令将文件输出到标准输出，然后再将标准输出连接到 while 语句的标准输入。

```
cat file.input |
while read lineoftext
do
    process that line
done
```

因为管道的存在，cat 命令和 while 循环（包括其中的 *process that line* 部分）各自在单独的进程中运行。这意

味着，如果你采用了这种方法，while 循环内部的命令不会影响到循环外部的其他部分。例如，在 while 循环中设置的变量在循环结束后也就失去了所设置的值。但如果你采用的是 while read ... done < file.input，就不会出现这种情况，因为这种方法并不涉及管道。

6.10.4 参考

- 6.1 节
- 6.2 节
- 6.3 节
- 6.4 节
- 6.5 节
- 6.6 节
- 6.7 节
- 6.8 节
- 6.11 节
- 19.8 节

6.11 在循环中使用**read**

6.11.1 问题

你正在使用 Subversion 版本控制系统，其对应的可执行文件是 svn。（这个例子对于 CVS 来说也是大同小异。）检查目录子树状态，查看是否有文件被改动时，你会看到如下显示：

```
$ svn status bcb
M      bcb/amin.c
?      bcb/dmin.c
?      bcb/mdiv.tmp
A      bcb/optrn.c
M      bcb/optson.c
?      bcb/prtbout.4161
?      bcb/rideaslist.odt
?      bcb/x.maxc
$
```

以 ? 起始的行代表 Subversion 尚不知晓的文件，通常是草稿文件或文件的临时副本。以 A 起始的行代表新添加的文件，以 M 起始的行代表自上次提交后又发生改动的文件。

要想清理该目录，最好是删除所有的草稿文件。

6.11.2 解决方案

while 循环的一种常见用法是读取文件和之前命令的输出。尝试下列命令：

```
svn status mysrc | grep '^?' | cut -c8- |
  while read FN; do echo "$FN"; rm -rf "$FN"; done
```

或者：

```
svn status mysrc |
while read TAG FN
do
    if [[ $TAG == \? ]]
    then
        echo $FN
        rm -rf "$FN"
    fi
done
```

6.11.3 讨论

这两个脚本的效果相同：删除 svn 报告其中带有 ? 标记的文件。同样的解决方案也适用于其他版本控制系统。

第一种方法是用多个子程序来实现（在如今处理器以 GHz 为单位进行计算的时代，这算不上什么大事），在典型的终端窗口中够写满一行。先用 grep 选择以 ? 起始（由 ^ 表示）的行。将 ^? 放进单引号中（'^?'）以避免 bash 解释其中具有特殊含义的字符。接着用 cut 提取从第 8 列开始（一直到行尾）的字符串。

如果没有输入，read 会返回非 0 值并停止循环。在此之前，它每次都会将读取到的文本行（其实也就是要删除的文件名）赋给变量

$FN。-rf 选项是以防待删除的未知文件是目录或只读文件。如果不希望删除文件时表现得如此极端，可以去掉这些选项。

第二个脚本更有 shell 的范儿，既不用 grep 进行搜索（使用的是 if 语句），也没用 cut 剪切（使用的是 read 语句）。我们还像在脚本文件中那样对其做了格式化。如果是在命令提示符下输入这些代码，你可以省略缩进，但作为书中的示例，可读性远比少敲几个键更重要。

第二个脚本中的 read 将内容读入两个变量，看清楚，这次可不是一个。我们也要让 bash 将每行分成两部分：为首的字符和文件名。read 语句将输入行解析成若干单词（和 shell 命令行中的类似）。输入行的第一个单词赋给 read 语句中的第一个变量，第二个单词赋给第二个变量，以此类推。最后一个变量获得输入行中剩余的所有内容，即便其中不止一个单词。在这个例子中，$TAG 得到的第一个单词是字符（M、A 或 ?）；空白字符用于分隔单词。变量 $FN 得到了行中剩余部分，也就是文件名，这一点在文件名中包含空白字符时尤为重要。（我们想要的可不是文件名中的第一个单词。）脚本接下来会删除相应的文件并进入下一次循环。

6.11.4 参考

- 附录 D

6.12 循环若干次

6.12.1 问题

你需要循环够一定次数。可以使用 while 循环，在计数时进行测试，不过编程语言中的 for 循环正是针对这种情况设计的。那么，如何在 bash 中实现呢？

6.12.2 解决方案

使用 for 循环语法的一种特例，看起来和 C 语言中的差不多，但使用的是双括号。

```
for (( i=0 ; i < 10 ; i++ )) ; do echo $i ; done
```

6.12.3 讨论

在早期的 shell 版本中，for 循环只能按照固定的列表项进行迭代。和文件名之类的打交道时，shell 脚本是面向单词的，就此而言，这算得上是一个不错的创新。但如果需要计数，用户会发现自己可能写出了如下代码。

```
for i in 1 2 3 4 5 6 7 8 9 10
do
    echo $i
done
```

看起来还行，尤其是循环次数不多时。可是说实话，换成 500 次循环可就不好使了。（没错，你可以写成 5×10 的嵌套循环，不过还是不要这样做了吧！）你真正需要的是能够计数的 for 循环。

bash 2.04 版开始引入一种 for 循环的变体，语法与 C 语言类似。其一般形式如下所示。

```
for (( expr1 ; expr2 ; expr3 )) ; do list ; done
```

双括号表明这是算术表达式，在其中引用变量时，不用加 $（但 $1 等位置参数除外），只要是 bash 中出现双括号的地方，均是如此。该表达式是整数表达式，可以使用包括逗号（用于在一个表达式中放入多个操作）在内的大量运算符。

```
for (( i=0, j=0 ; i+j < 10 ; i++, j++ ))
do
    echo $((i*j))
done
```

for 循环先初始化了两个变量（$i 和 $j），然后在第二个更复杂的子表达式中对 $i 和 $j 求和，接着判断是否小于 10。第三个子

表达式再次用逗号运算符累加这两个变量。

6.12.4 参考

- 6.1 节
- 6.13 节
- 17.24 节

6.13 在循环中使用浮点值

6.13.1 问题

带有算术表达式的 `for` 循环只能执行整数运算。如果是浮点值，该怎么办呢？

6.13.2 解决方案

如果系统提供了 `seq` 命令，则可以用它来生成浮点值。

```
for fp in $(seq 1.0 .01 1.1)
do
    echo $fp; other stuff too
done
```

或者：

```
seq 1.0 .01 1.1 |
while read fp
do
    echo $fp; other stuff too
done
```

6.13.3 讨论

`seq` 命令会生成一系列浮点值，每行一个。该命令的参数依次是起始值、增量、结束值。如果习惯了 C 语言中的 `for` 循环或学过 BASIC

中的循环（如 FOR I=4 TO 10 STEP 2），你可能会觉得这种次序不符合直觉。seq 使用中间那个参数作为增量。

在第一个例子中，$() 在子 shell 中执行命令，返回结果中的换行符会被空白字符替换，因此，就 for 循环而言，每个值都是字符串。

在第二个例子中，seq 作为单独的命令运行，通过管道将其输出传入 while 循环，后者读取其中的每一行并做相应处理。如果数值序列特别长，这种方法更可取，因为 seq 命令是与 while 并行运行的。在第一个例子的 for 循环中，seq 必须先运行完毕，然后用全部的输出结果替换掉 $()。对于冗长的数值序列来说，这种方法的时间开销和内存开销都不小。

6.13.4 参考

- 2.17 节
- 6.12 节
- 17.24 节

6.14 多路分支

6.14.1 问题

你需要进行一系列比较，但使用 if/then/else 显得又长又啰唆。有没有更简单的方法？

6.14.2 解决方案

可以用 case 语句实现多路分支。

```
case $FN in
    *.gif) gif2png $FN
        ;;
    *.png) pngOK $FN
        ;;
```

```
    *.jpg) jpg2gif $FN
        ;;
    *.tif | *.TIFF) tif2jpg $FN
        ;;
    *) printf "File not supported: %s" $FN
        ;;
esac
```

与其等价的 if/then/else 形式如下所示。

```
if [[ $FN == *.gif ]]
then
    gif2png $FN
elif [[ $FN == *.png ]]
then
    pngOK $FN
elif [[ $FN == *.jpg ]]
then
    jpg2gif $FN
elif [[ $FN == *.tif || $FN == *.TIFF ]]
then
    tif2jpg $FN
else
    printf "File not supported: %s" $FN
fi
```

6.14.3 讨论

case 语句会扩展 case 与 in 之间的单词（包括参数替换）。然后依次尝试匹配各个模式。这可是 shell 了不得的一项特性。它不是简单地比对值，而是进行字符串模式匹配（尽管并非正则表达式）。示例中用到的模式并不复杂：*.gif 匹配以普通字符[5] .gif 结尾的任意字符串（由 * 表示）。

[5]这里所谓的“普通字符”（literal character）是指仅代表字面含义的字符，与之对应的是“元字符”（metacharacter），这类字符具有某种特殊含义。——译者注

代表逻辑 OR 的 | 用于分隔不同模式，无论匹配到其中哪个模式，都会执行相同的处理。在我们给出的例子中，如果 $FN 以 .tif 或 .TIFF 结尾，则匹配模式并执行（假想的） tif2jpg 命令。

case 中并没有 else 或 default 关键词来指示未匹配到任何模式时该执行哪些语句。它使用 * 作为最后的模式，因为该模式什么都能匹配。将其放在最后以作为默认分支，匹配之前尚未被匹配到的内容。

双分号（;;）用于结束与某个模式关联的一组语句。从 bash 4 开始，还可以用另外两种写法结束一组语句。;;& 表示，即便找到匹配，依然尝试匹配下一个模式，如果发现该模式也匹配，则执行与之关联的语句。;& 表示执行流程会“直落而下”，不管模式是否匹配，都执行下一组语句。下面这个多少有些无聊的例子演示了上述特性的用法。

```
# 使用其他结束标记

case $FN in
    *.gif) gif2png $FN
           ;;&          # 继续向下查找匹配
    *.png) pngOK $FN
           ;;&          # 继续向下查找匹配
    *.jpg) jpg2gif $FN
           ;;&          # 继续向下查找匹配
    *.tif) tif2jpg $FN
           ;&           # 直落而下
    *.* ) echo "two.words"
           ;;
    * ) echo "oneword"
esac
```

只要 $FN 匹配前 4 个模式中的任意一个，bash 都会执行（假想的）转换命令，并继续向下查找匹配。它会发现第 5 个模式也能够匹配，因此还会显示短语 two.words。

同 C/C++ 和 Java 程序员谈一个题外话：bash 中的 case 类似于 switch 语句，每个模式对应一处分支。但要注意，switch/case 中使用的是 shell 变量（通常包含字符串值），分支中使用的是模式（不仅仅是常量值）。模式以右括号（可不是冒号）作结。C/C++ 和 Java 的 switch 语句中的 break 等同于 bash 中的双分号，default 关键词等同于 bash 中的 * 模式。

匹配过程区分大小写，但可以用 shopt -s nocasematch（bash 3.1 以上版本可用）改变此行为。该选项会影响到 case 和 [[。

我们用 esac（反过来拼写“c-a-s-e”，这种写法源于 Algol 68）来结束 case 语句。

6.14.4 参考

- help case
- help shopt
- 6.2 节

6.15 解析命令行参数

6.15.1 问题

你想编写一个简单的脚本，在屏幕上打印出一行连字符，但同时希望将该脚本参数化，以便指定不同的行长度以及除连字符之外的其他字符。其语法类似如下代码。

```
dashes              # 打印出72个连字符
dashes 50           # 打印出50个连字符
dashes -c = 50      # 打印出50个等号
dashes -c x         # 打印出72个字符x
```

有什么易行的方法可以解析这些简单参数？

6.15.2 解决方案

对于正式的脚本编程，应该使用 bash 内建的 getopts。但这里只是想向你展示 case 语句的实践用法，因此，对于这种简单的情况，我们选择用 case 来解析参数。

例 6-5 展示了该脚本的开头部分（完整版参见 12.1 节）。

例 6-5 ch06/dashes

```
#!/usr/bin/env bash
# 实例文件: dashes
#
# dashes - 打印出一行连字符
#
# 选项: #指定字符数量（默认为72个）
# -c X 使用字符X，不再使用默认的连字符
#

LEN=72
CHAR='-'
while (( $# > 0 ))
do
    case $1 in
        [0-9]*) LEN=$1
        ;;
        -c) shift;
               CHAR=${1:--}
        ;;
        *) printf 'usage: %s [-c X] [#]\n' ${0##*/} >&2
            exit 2
        ;;
    esac
    shift
done
#
# 未完……
```

6.15.3 讨论

默认字符数量（72）和默认字符（-）是在脚本起始部分（位于几行注释之后）设置的。while 允许脚本解析多个参数。只要参数数量（$#）不为 0，循环就会持续进行下去。

case 语句匹配 3 个模式。第一个模式 [0-9]* 能够匹配任意数位以及随后任意数量的字符。可以用更复杂的模式仅允许纯数字的出现，但我们假定任何以数位（digit）开头的参数都是数字（number）。如果事实并非如此（例如，用户输入的是 1T4），那么脚本尝试使用 $LEN 时就会出错。我们暂时可以容忍这种情况。

第二个模式是普通的 -c。这其实算不上模式，只是严格的字面匹配。在此分支中，我们用内建命令 shift 丢弃这个参数（因为已经知道这个参数是什么了）并获取下一个参数（现在已变成第一个参

数，因此用 $1 来引用它），同时将用户指定的新字符保存起来。在引用 $1 时，我们使用了 :-（${1:-x}），以便在未指定新字符时设置默认字符。这样一来，如果用户只输入 -c，却未指定新字符，则使用紧随 :- 之后的默认字符。要是写成 ${1:-x}，则默认字符为 x。脚本中出现的是 ${1:--}（注意，有两个减号），因此默认字符就是（第二个）减号。

第三个模式 * 可以匹配任意数量的字符，之前未匹配的参数都能够在此得以匹配。可以将其放在 case 语句的最后一个分支中来负责收尾工作，提醒用户发生了错误（出现了未知参数）；打印提示信息，告知用户正确的用法。

如果刚开始接触 bash，可能需要向你解释一下 printf 输出的错误信息。该语句分为 4 部分。第一部分很简单，就是命令名 printf。第二部分是 printf 要用到的格式化字符串（参见 2.3 节和附录 A）。为了避免 shell 解释其中的内容，我们在字符串周围加上了单引号。最后一部分（>&2）告诉 shell 将 printf 的输出重定向到标准错误。因为属于错误信息，所以这种做法没毛病。不少脚本作者并没有将此放在心上，经常忽略错误信息的重定向。我们认为坚持将错误消息重定向到标准错误是一个很好的习惯。

第三部分对 $0 进行字符串操作。这是一种惯用写法，用于将调用命令时的前导路径部分剔除。如果我们只使用 $0，会发生什么情况呢？以下是调用同一脚本时的两种错误方法。注意错误信息。

```
$ dashes -g
usage: dashes [-c X] [#]

$ /usr/local/bin/dashes -g
usage: /usr/local/bin/dashes [-c X] [#]
```

在第二次调用时，我们使用的是完整路径，于是其也出现在了错误信息中。有人会觉得这样很烦。因此，我们剔除了 $0 中的部分内容，只留下脚本的基本名称（类似于使用 basename 命令）。以后无论以何种方式调用脚本，错误信息看起来都是一模一样的。

```
$ dashes -g
usage: dashes [-c X] [#]
```

```
$ /usr/local/bin/dashes -g
usage: dashes [-c X] [#]
```

相较于硬编码脚本名称或原封不动地使用 $0，这种方法肯定要多花点时间，但脚本的可移植性会更好，就算重命名脚本，也用不着再修改代码。如果你更喜欢在子 shell 中使用 basename 命令，同样值得一试，多出的那点运行时间不值一提。输出错误信息后，就退出脚本。

我们用 esac 结束了 case 语句，然后通过 shift 丢弃刚刚在 case 语句中匹配到的参数。要是不这么做的话，就会深陷在 while 循环之中，一遍又一遍地解析同一个参数。shift 会使第 2 个参数（$2）成为第 1 个参数（$1），第 3 个参数成为第 2 个参数，以此类推，同时还会将 $# 减 1。经过几次循环后，$# 最终会变成 0（表示此时已经没有任何参数了），循环随之终止。

这里并没有展示连字符（或者其他字符）的实际输出效果，因为我们希望将重点放在 case 语句及其相关处理上。你可以在 12.1 节中找到完整的脚本以及用于输出用法信息的函数。

6.15.4 参考

- help case
- help getopts
- 2.3 节
- 5.8 节
- 5.11 节
- 5.12 节
- 5.20 节
- 6.15 节
- 12.1 节
- 13.1 节
- 13.2 节
- A.12 节

6.16 创建简单的菜单

6.16.1 问题

你有一个简单的 SQL 脚本，希望能够针对不同的数据库运行，以便将其重置用于各种测试。可以直接在命令行上提供数据库名，但你想要拥有更多的交互性。如何编写一个可以从数据库名称列表中进行选择的 shell 脚本呢？

6.16.2 解决方案

用 select 语句创建简单的字符型屏幕菜单，如例 6-6 所示。

例 6-6 ch06/dbinit.1

```
#!/usr/bin/env bash
# 实例文件：dbinit.1
#
DBLIST=$(sh ./listdb | tail -n +2)
select DB in $DBLIST
do
    echo Initializing database: $DB
    mysql -u user -p $DB <myinit.sql
done
```

暂时先忽略 $DBLIST 是如何得到值的，知道该值是一个单词列表即可（就像 ls 命令的输出）。select 语句会显示出这些单词，每个单词前都有一个数字，然后由用户做出选择。用户只需要输入菜单项数字，对应的单词就会保存在 select 之后指定的变量（本例中是变量 DB）中。

以下是运行该脚本时的输出。

```
$ ./dbinit
1) testDB
2) simpleInventory
3) masterInventory
4) otherDB
#? 2
Initializing database: simpleInventory
#?
$
```

6.16.3 讨论

如果用户输入“2”，则单词 simpleInventory 会保存在变量 DB 中。如果着实需要用户所选的菜单项编号，你可以在变量 $REPLY 中找到。本例中的这个值是 2。

select 语句其实就是一个循环。当用户输入选择时，执行循环体（位于 do 和 done 之间），然后再次提示用户输入下一次选择。

菜单不会每次都重新显示，除非用户什么都不选，直接按下回车键。因此，要想再看菜单，按回车键就行了。

此外，菜单也不会对 in 之后的变量重新求值，也就是说，一旦 select 语句运行，就无法改动菜单了。就算你在循环内修改了 $DBLIST，菜单也不会出现变化。

如果碰到文件结束符，则停止循环，在交互式用法中，这意味着用户按下了 Ctrl-D。（如果通过管道将一系列选择传给 select，那么输入结束时，循环也会随之结束）。

尽管 select 确实会考虑 $COLUMNS 的值，但并没有什么办法能控制菜单的格式。要是你打算使用 select，那也只能坦然接受了。不过，你可以用变量 $PS3 来修改 select 的提示符，详见 6.17 节和 16.12 节。

6.16.4 参考

- 3.7 节
- 6.17 节
- 16.2 节
- 16.12 节

6.17 修改简单菜单的提示符

6.17.1 问题

你不喜欢 select 所生成菜单的提示符。该怎么修改？

6.17.2 解决方案

bash 环境变量 $PS3 就是 select 所使用的提示符。将其设置成别的值就可以得到一个新的提示符。

6.17.3 讨论

这是第三种 bash 提示符。第一种提示符（$PS1）出现在大多数命令之前。（我们给出的例子用 $ 作为提示符，但它可以加入用户 ID 或目录名来变得更为复杂。）如果命令行需要续行，那么就得用到第二种提示符（$PS2）了。

第三种提示符 $PS3 出现在 select 循环中。在执行 select 语句前设置该变量，就可以得到你想要的提示符。甚至还可以在循环内部对其做出改动。

例 6-7 中的脚本和上一节中的类似，但是它会统计处理了多少次有效选择。

例 6-7 ch06/dbinit.2

```
#!/usr/bin/env bash
# 实例文件：dbinit.2
#
DBLIST=$(sh ./listdb | tail -n +2)

PS3="0 inits >"

select DB in $DBLIST
do
    if [ $DB ]
    then
        echo Initializing database: $DB

        PS3="$((++i)) inits> "

        mysql -u user -p $DB <myinit.sql
    fi
done
```

我们额外加入了空行，这样做是为了让 $PS3 的设置部分更加显眼。if 语句确保我们只统计用户的有效选择。这种检查可以用在该例的上一个版本中，不过当时我们想要示例保持简单化。

6.17.4 参考

- 3.7 节
- 6.17 节
- 16.2 节
- 16.12 节

6.18 创建简单的RPN计算器

6.18.1 问题

你也许只用口算就能将二进制转换成十进制、八进制、十六进制，不过同样的法子对简单的算术运算似乎无能为力，而且计算器还总是在需要的时候不见踪影。这该怎么办？

6.18.2 解决方案

像例 6-8 中那样，用 shell 算术和 RPN[6] 创建一个计算器。

[6] RPN（Reverse Polish notation，逆波兰表示法或逆波兰记法）是一种由波兰数学家扬·武卡谢维奇于 1920 年引入的数学表达式方式。在逆波兰记法中，所有运算符都置于操作数的后面，因此也被称为后缀表示法。逆波兰记法不需要括号来标识运算符的优先级。

例 6-8 ch06/rpncalc

```
#!/usr/bin/env bash
# 实例文件：rpncalc
#
# 简单的RPN命令行（整数）计算器
#
# 获取用户提供的参数并计算
# 参数形式为：a b运算符
# 允许使用*x*代替*
```

```
#
# 检查参数数量：
if [ \( $# -lt 3 \) -o \( $(($# % 2)) -eq 0 \) ]
then
    echo "usage: calc number number op [ number op ] ..."
    echo "use x or '*' for multiplication"
    exit 1
fi

ANS=$(($1 ${3//x/*} $2))
shift 3
while [ $# -gt 0 ]
do
    ANS=$((ANS ${2//x/*} $1))
    shift 2
done
echo $ANS
```

6.18.3 讨论

RPN（或者后缀）风格的表示法将操作数（数字）放在前面，然后是运算符。如果使用 RPN，那么表达式应该写成 5 4 +，而不是 5 + 4。要是想将结果再乘以 2，将 2 * 写在后面即可，因此整个表达式就是 5 4 + 2 *，这种写法在解析表达式时实在是再好不过了，因为可以从左向右处理，压根不需要括号。任何运算的结果都会变成下一个子表达式的第一个操作数。

在我们这个简单的 bash 计算器中，允许用小写字母 x 代替乘号，因为 * 对于 shell 而言具有特殊含义。但如果你写成 '*' 或 *，也没问题。

如何检查参数错误？如果参数数量少于 3 个（我们需要 2 个操作数和 1 个运算符，如 6 3 /），那么就认为出错了。参数可以多于 3 个，但总数始终应该是奇数（一开始是 3 个参数，然后是下一个操作数和运算符，共计 2 个，以此类推，始终要再添加 2 个参数。有效的参数数量应该是 3、5、7、9……）。我们用下列语句检查参数数量，比较结果是否为 0。

```
$(($# % 2)) -eq 0
```

$(( )) 用来执行 shell 算术运算。我们用 % 运算符（**求余运算符**）检查 $#（其中保存着参数数量）能否被 2 整除（通过 -eq 0 判断）。

$(( )) 只能对其中的表达式执行整数运算。

现在我们知道了参数数量没问题，可以计算结果了。

```
ANS=$(($1 ${3//x/*} $2))
```

该语句将 * 替换成字母 x 并计算出最终结果。在命令行上调用脚本时，给出的是 RPN 表达式，但是 shell 算术运算所采用的语法还是正常的（中缀）写法。因此，必须交换一下参数位置才能在 $(( )) 中求值。暂时忽略用 * 替换 x 这部分，前面的语句就变成了：

```
ANS=$(($1 $3 $2))
```

这里只是将运算符移到了两个操作数之间。在进行算术求值之前，bash 会替换掉这几个参数，因此，如果 $1 是 5、$2 是 4、$3 是 +，那么参数替换后得到的就是：

```
ANS=$((5 + 4))
```

我们将求值结果 9 赋给变量 $ANS。处理完这 3 个参数后，用 shift 3 将其丢弃并获得新的参数，然后继续进行后续处理。由于之前已经确认了有奇数个参数，因此如果还有参数要处理，至少是 2 个（只有 1 个的话，参数总量就是偶数了，因为 3+1=4）。

接下来进入 while 循环，每次处理 2 个参数。上一步的结果作为第一个操作数，下一个参数（执行完 shift 之后，现在变成了 $1）作为第二个操作数，我们将保存在 $2 中的运算符放置在两个操作数之间，然后像先前那样对表达式求值。一旦处理完所有参数，$ANS 中保存的就是最终结果。

最后再说一下替换操作。我们用 ${2} 引用第二个参数。虽然通常用不着 {}，直接写成 $2 即可，但这里是为了让 bash 对该参数执行额外的操作。我们将其写成 ${2//x/*} 是想在返回 $2 的值之前

先用 *（在第 3 个 / 后指明）替换掉（//）x。多加一个变量可以将这一步拆成两小步。

```
OP=${2//x/*}
ANS=$((ANS OP $1))
```

如果是刚接触 bash 的这些特性，这个额外变量有助于你加深理解，一旦熟悉了这种惯用写法，你会发现自己不由自主地就将它们写成了一行（尽管可读性不太好）。

你是不是好奇我们对表达式求值时为什么不写成 $ANS 和 $OP ？在 $(( )) 中，除了位置参数（如 $1、$2），变量名前面并不需要加 $。位置参数需要 $ 是为了和普通数字（如 1、2）区分开来。

6.18.4 参考

- 第 5 章
- 6.1 节
- 6.19 节

6.19 创建命令行计算器

6.19.1 问题

你需要的不仅仅是整数算术，而且压根也不喜欢 RPN 表示法。有没有别的方法可以实现命令行计算器？

6.19.2 解决方案

可以用 awk 内建的浮点算术表达式创建简单的命令行计算器，如例 6-9 所示。

例 6-9 ch06/func_calc

```
# 实例文件：func_calc
```

```
# 简单的命令行计算器
function calc {
    # 仅适用于整数! --> echo The answer is: $(( $* ))
    # 浮点数
    awk "BEGIN {print \"The answer is: \" $* }";
} # 函数calc定义完毕
```

6.19.3 讨论

你可能想跳过 awk 命令，并尝试用 echo The answer is: $$$*$$。如果是整数，这么做没问题，但如果是浮点数运算，结果会被截断。

因为别名（参见 10.7 节）不允许使用参数，所以我们使用了函数。

你可能会想将这个函数添加到全局配置文件 /etc/bashrc 或本地配置文件~/.bashrc。

运算符和你期望中的一样，与 C 语言中的也无异。

```
$ calc 2 + 3 + 4
The answer is: 9

$ calc 2 + 3 + 4.5
The answer is: 9.5
```

小心 shell 的元字符。例如：

```
$ calc (2+2-3)*4
-bash: syntax error near unexpected token `2+2-3'
```

你需要将括号转义，消除其特殊含义。可以将表达式放进单引号，或在任何特殊字符（对 shell 而言）之前加上斜线。例如：

```
$ calc '(2+2-3)*4'
The answer is: 4

$ calc \(2+2-3\)\*4
The answer is: 4
```

```
$ calc '(2+2-3)*4.5'
The answer is: 4.5
```

我们还得转义 *，因为它是文件名通配符，对 bash 具有特殊含义。如果你想在运算符周围加上空白字符，则更是如此，因为 * 会匹配当前目录下的所有文件名，bash 会在命令行上用这些文件名替换 *，这肯定不是你想要的结果。

6.19.4 参考

- man awk
- bash（1）手册页中的“ARITHMETIC EVALUATION”部分
- 6.18 节
- 10.7 节
- 16.8 节

第 7 章　中级shell工具

是时候扩充你的工具库了。本章中的实例用到了一些不属于 shell 的实用工具，其实用性令人难以想象要是没了它们，该如何使用 shell。

Unix（以及 Linux）的首要理念之一就是小（规模有限）程序能够相互结合来获得惊人的结果。我们并不需要一个无所不能的程序，相反可以使用多个程序，每个程序做好一件事即可。

这也适用于 bash。虽然 bash 的体积越来越大，特性越来越丰富，但它仍不打算大包大揽。虽然有时候单凭 bash 费点事也能搞定，但换作其他命令的话，任务会完成得更加轻松。

ls 命令就是一个例子。不用 ls 也能查看当前目录的内容。输入 echo * 就能显示出其中的文件名。或者还可以更炫一些：使用 bash 的 printf 命令，再加上一些格式化操作等。但这并非 shell 的真正目的，有人已经提供了列表程序（ls）来处理各种文件系统信息。

不只靠 bash 提供更多的文件系统列表特性，在避免额外的特性蔓延（feature creep）[1] 压力的同时获得一定程度的独立性，也许这才是更重要的。

[1] “特性蔓延”指在电子设备或软件上添加过多功能的倾向，这种做法并没有令原来的产品变得更好，而是愈加复杂难用，含贬义。——译者注

设计理念已经说得够多了，接下来让我们回归实践。

本章要介绍 3 个最实用的文本相关工具：grep、sed、awk。

grep 可以搜索字符串，sed 提供了在管道中编辑文本的方法，而 awk……嗯，awk 自成一派，它是 Perl 的前身，形式多变，根据其具体用法，呈现出的样子可能大相径庭。

这 3 个以及下一章要介绍的另一些实用工具是大多数 shell 脚本的重要组成部分，我们大部分时间要和它们打交道。如果你的 shell 脚本需要一个待处理文件的列表，可以通过 find 或 grep 来提供，也许还可以用 sed/awk 来解析输入或者格式化 shell 脚本在某个阶段的输出。

换句话说，如果要编写脚本来处理现实世界的问题，那么就得用现实世界中的 bash 用户和程序员所使用的各种工具。

7.1 在文件中查找字符串

7.1.1 问题

你需要在一个或多个文件中查找所有出现过的某个字符串。

7.1.2 解决方案

grep 命令可以搜索文件，查找指定的字符串。

```
$ grep printf *.c
both.c:    printf("Std Out message.\n", argv[0], argc-1);
both.c:    fprintf(stderr, "Std Error message.\n", argv[0], argc-
1);
good.c:    printf("%s: %d args.\n", argv[0], argc-1);
somio.c:         // we'll use printf to tell us what we
somio.c:         printf("open: fd=%d\n", iod[i]);
$
```

在这个例子中，我们搜索的文件全都位于当前目录下。因此，我们只使用了简单的 shell 模式 *.c 来匹配以 .c 结束的文件，并没有在文件名前再添加路径。

但并非所有待搜索的文件都老老实实地待在当前目录下。但因为 shell 并不在意你输入多少路径名，所以我们也可以这么写：

```
grep printf ../lib/*.c ../server/*.c ../cmd/*.c */*.c
```

7.1.3 讨论

如果待搜索的文件不止一个，grep 会在输出前加上文件名以及冒号，然后是该文件中包含 grep 搜索内容的文本。

搜索会匹配指定的字符串，所以含有“fprintf”的行会被返回，因为“fprintf”包含“printf”。

grep 的第一个（非选项）参数可以是一个简单的字符串，正如本例中所示，也可以是更复杂的正则表达式（regexp）。正则表达式不同于 shell 的模式匹配，尽管两者有时看起来差不多。模式匹配的功能无比强大，你可能会发现自己对其的依赖已经到了开始将“grep”作为动词的程度，只想随时随地都能用到它，比如“希望能在我的书桌上 grep 到你想要的论文”。

你可以用命令行选项控制 grep 的输出形式。如果不想看到特定文件名，指定 -h 选项即可。

```
$ grep -h printf *.c
  printf("Std Out message.\n", argv[0], argc-1);
  fprintf(stderr, "Std Error message.\n", argv[0], argc-1);
  printf("%s: %d args.\n", argv[0], argc-1);
      // we'll use printf to tell us what we
      printf("open: fd=%d\n", iod[i]);
$
```

如果不想看到文件中包含指定字符串的那些行，只想知道匹配到了几次，可以使用 -c 选项。

```
$ grep -c printf *.c
both.c:2
good.c:1
somio.c:2
$
```

常见的错误是忘记指定 grep 的输入，例如 grep *myvar*。这种情况下，grep 会认为你要从 STDIN 提供输入，而你以为它会读取文件，于是 grep 就干等着，无所事事。（其实它是在等

待着你从键盘输入。）当你要搜索大量数据，预计会花费很长时间时，很难分辨出这种尴尬的情况。

7.1.4 参考

- man grep
- man regex（Linux、Solaris、HP-UX）或 man re_format（BSD、Mac），参见有关正则表达式库的详情
- Jeffrey E. F. Friedl 所著的 *Mastering Regular Expressions, 3rd Edition*（O'Reilly 出版）
- Nelson H. F. Beebe 与 Arnold Robbins 合著的 *Classic Shell Scripting*（O'Reilly）中的 3.1 节及 3.2 节
- 第 9 章和 find 命令，了解更多高级搜索功能
- 9.5 节

7.2 只显示包含搜索结果的文件名

7.2.1 问题

你需要找出包含特定字符串的文件，但是不想看到其所在的文本行，只用输出文件名即可。

7.2.2 解决方案

用 grep 的 -l 选项仅显示文件名即可。

```
$ grep -l printf *.c
both.c
good.c
somio.c
$
```

7.2.3 讨论

如果在一个文件中找到了多次匹配，grep 仍然只输出该文件名一次。如果没有找到匹配，则什么都不输出。

由于这些文件包含了你要查找的字符串，如果想据此构建一个待处理文件的列表，选项 -h 就能派上用场了。将 grep 命令放进 $()，然后就可以在命令行上使用这些文件名了。

例如，要想删除包含字符串“This file is obsolete”的文件，可以使用以下这样的 shell 命令组合。

```
rm -i $(grep -l 'This file is obsolete' * )
```

我们给 rm 加上了 -i 选项，以便在删除每个文件前都先询问你。考虑到该命令组合的威力，这显然是一种更安全的处理方式。

bash 对 * 进行扩展，以匹配当前目录（但并不会深入子目录）下的所有文件并将其作为 grep 的参数。grep 生成包含指定字符串的文件列表。最后，rm 会删除该列表中的所有文件。

7.2.4 参考

- man grep
- man rm
- man regex（Linux、Solaris、HP-UX）或 man re_format（BSD、Mac），参见有关正则表达式库的详情
- Jeffrey E. F. Friedl 所著的 *Mastering Regular Expressions, 3rd Edition*（O'Reilly 出版）
- 2.15 节
- 9.5 节

7.3 了解搜索是否成功

7.3.1 问题

你想要知道某个文件是否包含指定的字符串，而且只关心“有”还是“没有”。

7.3.2 解决方案

使用 grep 的 -q（quiet）选项。或者，考虑到最大的可移植性，简单将输出重定向到 /dev/null 丢弃。不管用哪种方法，你想知道的答案就在 bash 的返回状态变量 $? 中，因此可以像下面这样将其用于 if 语句：

```
$ if grep -q findme bigdata.file ; then echo yes ; else echo nope
; fi
nope
$
```

7.3.3 讨论

在 shell 脚本中，通常并不需要输出搜索结果；你只是想知道有没有找到匹配，以便脚本选择相应的分支。

和大多数 Unix/Linux 命令一样，返回值 0 表示命令成功执行完毕。在这个例子中，成功的意思就是在指定文件（这里只搜索了一个文件）至少找到了一处匹配。返回值保存在 shell 变量 $? 中，随后可以在 if 语句中用到。

如果我们在 grep -q 之后列出多个文件名，那么 grep 会在找到第一处匹配后就停止搜索。它并不会搜索所有文件，因为你只是想知道有没有找到匹配。要是真想搜遍所有文件，（为什么这么做？）那就别用 -q 选项了。

```
$ if grep findme bigdata.file > /dev/null ; then echo yes ; else
echo nope ; fi
nope
$
```

/dev/null 是一个特殊设备（位桶），重定向到此处的输出全都会被丢弃。

如果你希望自己编写的脚本能够适用于 Unix/Linux 系统上的各种 grep 版本，即便某些版本不支持 -q 选项，/dev/null 同样能够发挥作用。

7.3.4 参考

- man grep
- man regex（Linux、Solaris、HP-UX）或 man re_format（BSD、Mac），参见有关正则表达式库的详情
- Jeffrey E. F. Friedl 所著的 *Mastering Regular Expressions, 3rd Edition*（O'Reilly 出版）
- 9.5 节

7.4 不区分大小写搜索

7.4.1 问题

你想要在日志文件中不区分大小写地搜索字符串（如“error”），以匹配该字符串的所有出现。

7.4.2 解决方案

用 grep 的 -i 选项忽略大小写。

```
grep -i error logfile.msgs
```

7.4.3 讨论

不区分大小写的搜索能够找出包含“ERROR”、“error”、“Error”的日志消息，“ErrOR”和“eRrOr”这样的也不例外。该选项在查找大小写混合的单词时尤其管用，包括可能在句首或电子邮件地址中大写的那些单词。

7.4.4 参考

- man grep
- man regex（Linux、Solaris、HP-UX）或 man re_format（BSD、Mac），参见有关正则表达式库的详情
- Jeffrey E. F. Friedl 所著的 *Mastering Regular Expressions, 3rd Edition*（O'Reilly 出版）
- 9.5 节

7.5 在管道中进行搜索

7.5.1 问题

你要搜索的文本并不在文件中，而是在某个命令或命令管道的输出中。

7.5.2 解决方案

将输出导入 grep 即可。

```
some pipeline | of commands | grep
```

7.5.3 讨论

如果没有提供文件名，grep 会从标准输入读取。大多数经过良好设计可用于 shell 脚本编程的实用工具会这么做。这也是它们能够作为 shell 脚本的构建块（building block）发挥作用的原因之一。

如果还想用 grep 搜索上一个命令中的错误信息，那么要确保在管道之前将该命令的错误输出重定向到标准输出。

```
gcc bigbadcode.c 2>&1 | grep -i error
```

此命令尝试编译一些假想的代码。将输出通过管道（|）传给 grep 之前，我们先重定向命令的标准错误到标准输出（2>&1），然后由 grep 搜索字符串 error，不区分其大小写（-i）。

对 grep 的结果再执行 grep 也不是不可能的事。为什么要这么做？这是为了进一步缩小搜索范围。假设你想要找出 Bob Johnson 的电子邮件地址。

```
$ grep -i johnson mail/*
... too much output to think about; there are lots of Johnsons in
the world ...
$ !! | grep -i robert
grep -i johnson mail/* | grep -i robert
... more manageable output ...
$ !! | grep -i "the bluesman"
grep -i johnson mail/* | grep -i robert | grep -i "the bluesman"
Robert M. Johnson, The Bluesman <rmj@noplace.org>
```

你可以将第一个 grep 命令再输入一遍，但这个示例还展示了历史运算符 !! 的威力（参见 18.2 节）。!! 可以重复上一个命令，无须重新输入。然后就像示例那样在 !! 之后添加新的命令。shell 会显示所执行的命令，因此你会看到 !! 替换后的结果。

你可以按照这种方法简单快速地构建一个比较长的 grep 管道，查看中间步骤的结果，并决定如何用其他 grep 表达式改善搜索过程。也可以用单个 grep 配合精巧的正则表达式来完成相同任务，不过我们发现以增量方式构建管道要来得更加容易。

7.5.4 参考

- man grep
- man regex（Linux、Solaris、HP-UX）或 man re_format（BSD、Mac），参见有关正则表达式库的详情
- Jeffrey E. F. Friedl 所著的 *Mastering Regular Expressions, 3rd Edition*（O'Reilly 出版）
- 2.15 节
- 9.5 节
- 18.2 节

7.6 缩减搜索结果

7.6.1 问题

搜索返回的结果不符合预期，其中包括许多并不需要的内容。

7.6.2 解决方案

将结果通过管道传给 grep -v 并用表达式描述出你不想看到的内容。

假设你想在日志文件中找出整个 12 月的日志消息。你知道日志文件用字母缩写 Dec 代表 12 月，但不敢肯定总是如此，为了确保找出所有的日志消息，输入下列命令：

```
grep -i dec logfile
```

得到的结果却如下所示：

```
...
error on Jan 01: not a decimal number
error on Feb 13: base converted to Decimal
warning on Mar 22: using only decimal numbers
error on Dec 16 : the actual message you wanted
error on Jan 01: not a decimal number
...
```

一种快而糙的解决方案是，将第一次得到的结果通过管道传给另一个 grep，由后者过滤掉所有的“decimal”。

```
grep -i dec logfile | grep -vi decimal
```

将多个 grep 串联在一起（因为前所未见、出乎意料的匹配会不断出现），逐步过滤搜索结果，直至满意，这种做法并不鲜见。

```
grep -i dec logfile | grep -vi decimal | grep -vi decimate
```

7.6.3 讨论

在这种解决方案中，“糙”是指它可能会漏掉一些你本想留下的 12 月的日志消息：如果消息中含有单词“decimal”，那么就会被 grep -v 过滤。

如果谨慎使用，-v 选项非常方便，你只需要记住该排除什么就行了。

对于这个例子，更好的解决方案是使用威力更为强大的正则表达式来匹配月份和日期，查找“Dec”以及紧随其后的一个空格和两个数字。

```
grep 'Dec [0-9][0-9]' logfile
```

不过，这种写法往往行不通，因为 syslog 会用一个空格填充单数字日期。要想解决这个问题，可以在第一个字符集中加上一个空格。

```
grep 'Dec [0-9 ][0-9]' logfile
```

由于存在内嵌空格，同时为了避免 shell 解释方括号，我们在方括号周围加上了单引号。在任何可能会对 shell 造成困惑的地方使用单引号是一个好习惯。我们可以使用反斜线转义空格，如下所示：

```
grep Dec\ [0-9\ ][0-9] logfile
```

但是这种写法不太容易看出待搜索的字符串在哪里结束，文件名从哪里开始。

7.6.4 参考

- man grep
- man regex（Linux、Solaris、HP-UX）或 man re_format（BSD、Mac），参见有关正则表达式库的详情
- Jeffrey E. F. Friedl 所著的 *Mastering Regular Expressions, 3rd Edition*（O'Reilly 出版）
- 9.5 节

7.7 搜索更复杂的模式

grep 中的正则表达式提供了更为强大的模式匹配功能，能够满足大部分需求。

正则表达式描述了待匹配字符串的模式。字母字符（或者对于 shell 没有特殊含义的其他字符）只匹配自身。“A”匹配 A，“B”匹配 B，这没什么好说的。另一个重要的规则是按位置组合字母，如 AB匹配“AB”。这看起来也是显而易见的。但是，正则表达式还定义了其他一些特殊字符，它们既可以单独使用，也可以与其他字符结合，从而形成更为复杂的模式。

第一个特殊字符是点号（.），它可以匹配任意单个字符。因此，.... 可以匹配任意 4 个字符；A. 匹配“A”以及紧随其后的任意单个字符；.A. 匹配任意单个字符，然后是“A”，接着是任意单个字符（未必和匹配到的第一个字符相同）。

星号（*）匹配上一个字符的 0 次或多次出现，因此，A* 匹配 0 个或多个“A”字符，.* 匹配 0 个或多个任意字符（如“abcdefg”、“aaaabc”、“sdfgf ;lkjhj”，甚至是空行）。

那么 ..* 是什么意思？它匹配任意单个字符以及紧随其后的 0 个或多个任意字符（也就是一个或多个字符，但不能是空行）。

说到文本行，脱字符 ^ 匹配文本行的行首位置，而美元符号 $ 匹配文本行的行尾位置，因此 ^$ 匹配空行（行首紧接行尾，两者之间什么都没有）。

如果想要匹配字面上的点号、脱字符、美元符号或者其他特殊字符，该怎么办呢？在特殊字符前加上反斜线（\）即可。ion. 匹配字母“ion”以及紧随其后的任意单个字符，但是 ion\. 则匹配“ion”以及点号（例如，在句末或者其他以尾部点号出现的地方）。

方括号中的一组字符（如 [abc]）匹配其中任意某个字符（如“a”、“b”或“c”）。如果方括号内的第一个字符是脱字符，则匹配的是**不在该字符组中的**任意字符。

举例来说，[AaEeIiOoUu] 匹配任意的元音字母，[^AaEeIiOoUu] 则匹配元音字母以外的任意字符。后一种情况不同于匹配辅音字母，

因为 [^AaEeIiOoUu] 也能匹配标点符号和其他既非元音也非辅音的特殊字符。

另一个要介绍的是被称为“区间表达式”的重复机制，写作 \{*n*,*m*\}，其中，*n* 是重复的最小次数，*m* 是重复的最大次数。如果写作 \{*n*\}，则表示“只重复 *n* 次”；写作 \{*n*,\}，则表示“至少重复 *n* 次”。

例如，正则表达式 A\{5\} 匹配连续 5 个“A”，而 A\{5,\} 至少匹配连续 5 个“A”。

参考

- man grep
- 7.8 节

7.8 搜索SSN

7.8.1 问题

你需要一个能够匹配社会保险号（social security number，SSN）的正则表达式。

7.8.2 解决方案

美国的社会保险号由 9 位数字组成，通常划分为 3 部分：先是 3 位数字，然后是 2 位数字，最后是 4 位数字（如 123-45-6789）。有时候也可以不使用连字符，因此要在正则表达式中令连字符成为可选字符。

```
grep '[0-9]\{3\}-\{0,1\}[0-9]\{2\}-\{0,1\}[0-9]\{4\}' datafile
```

你应该能根据需要修改这个正则表达式以适应其他国家的要求，或者参考 7.8.4 节给出的参考图书。

7.8.3 讨论

这种正则表达式经常被戏称为**只写**表达式，意思是很难或压根无法阅读。这里我们将其一一分解来帮助你理解。一般而言，只要是涉及正则表达式的 bash 脚本，一定要在正则表达式附近加上注释，解释清楚要匹配的内容。

在正则表达式中加入一些额外的空格能够改善可读性，让视觉效果更加轻松，但这同时也会改变正则表达式的含义，变成了需要在这些位置上匹配空格字符。暂时忽略这一点，先在其中插入一些空格，方便阅读。

```
[0-9]\{3\} -\{0,1\} [0-9]\{2\} -\{0,1\} [0-9]\{4\}
```

第一部分匹配 3 个任意数字。第二部分匹配 0 次或 1 次连字符。第三部分匹配 2 个任意数字。第四部分匹配 0 次或 1 次连字符。最后一部分匹配 4 个任意数字。

7.8.4 参考

- man regex（Linux、Solaris、HP-UX）或 man re_format（BSD、Mac），参见有关正则表达式库的详情
- Nelson H. F. Beebe 与 Arnold Robbins 合著的 *Classic Shell Scripting*（O'Reilly）中的 3.2 节，参见有关正则表达式及其相关应用工具的详情
- Jeffrey E. F. Friedl 所著的 *Mastering Regular Expressions, 3rd Edition*（O'Reilly 出版）
- Jan Goyvaerts 与 Steven Levithan 合著的 *Regular Expressions Cookbook, 2nd Edition*（O'Reilly 出版）
- 9.5 节

7.9 搜索压缩文件

7.9.1 问题

你需要搜索压缩文件。是不是非得先解压缩？

7.9.2 解决方案

如果系统中提供了 zgrep、zcat 或 gzcat，那就不必了。

zgrep 就是能够理解各类压缩文件和未压缩文件（具体类型视系统而定）的 grep。通常来说，在 Linux 系统中搜索 syslog 消息时就得用上它，因为日志轮替机制不会压缩当前日志文件（这样才不会影响正常使用），但会用 gzip 压缩归档日志。

```
zgrep 'search term' /var/log/messages*
```

zcat 就是能够理解各类压缩文件和未压缩文件（具体类型视系统而定）的 cat。它比 zgrep 理解的格式还要多，而且很多系统已经默认安装了。zcat 还可用于恢复受损的压缩文件，因为它会尽可能输出所有内容，而不像 gunzip 或其他工具那样出错。

```
zcat /var/log/messages.1.gz
```

gzcat 类似于 zcat，二者的区别在于商业软件与自由软件之分，以及向后兼容性。

7.9.3 讨论

经过配置后，实用工具 less 也可以显示各种压缩文件内容，相当方便。参见 8.15 节。

7.9.4 参考

- 8.6 节
- 8.7 节
- 8.15 节

7.10 保留部分输出

7.10.1 问题

你需要用某种方法保留部分输出，丢弃其余输出。

7.10.2 解决方案

以下代码会打印出所有输入行的第一个字段。

```
awk '{print $1}' myinput.file
```

字段之间以空白字符分隔。实用工具 awk 从命令行上指定的文件中读取数据，如果没有指定文件，则从标准输入读取。因此，你也可以将输入重定向到文件，如下所示：

```
awk '{print $1}' < myinput.file
```

或者通过管道传入：

```
cat myinput.file | awk '{print $1}'
```

7.10.3 讨论

awk 的用法多变。最简单的用法就是从输入中打印出所选的一个或多个字段。

字段之间以空白字符分隔（也可以用 -F 选项指定分隔字符），编号从 1 开始。字段 $0 代表整个输入行。

awk 是一门完备的编程语言。awk 脚本可以变得极为复杂。这个实例仅仅是个开始而已。

7.10.4 参考

- 8.4 节
- 13.13 节
- man awk

- comp.lang.awk FAQ
- Arnold Robbins 所著的 *Effective awk Programming, 4th Edition*（O'Reilly 出版）
- Arnold Robbins 与 Dale Dougherty 合著的 *sed & awk, 2nd Edition*（O'Reilly 出版）

7.11 仅保留部分输入行

7.11.1 问题

你只想保留部分输入行，例如第一个和最后一个字段。举例来说，你希望 `ls` 只列出文件名和权限，不需要 `ls -l` 所提供的其他信息。可惜的是，`ls` 并没有相应的选项能够按照这种方式限制输出。

7.11.2 解决方案

可以通过管道将 `ls` 的输出传给 `awk`，并从中挑选出你需要的字段。

```
$ ls -l | awk '{print $1, $NF}'
total 151130
-rw-r--r-- add.1
drwxr-xr-x art
drwxr-xr-x bin
-rw-r--r-- BuddyIcon.png
drwxr-xr-x CDs
drwxr-xr-x downloads
drwxr-sr-x eclipse
...
$
```

7.11.3 讨论

思考 `ls -l` 命令的输出。其形式如下所示：

```
drwxr-xr-x 2 username group        176 2006-10-28 20:09 bin
```

对于 awk 而言，解析这种输出易如反掌（在 awk 中，默认的字段分隔符为空白字符）。

在输出文件名时，我们用了点小技巧。在 awk 中，各种字段是用美元符号和字段编号来引用的（如 $1、$2、$3），而且 awk 还有一个内建变量 NF，其中保存着当前行中的字段总数，$NF 总是引用最后一个字段。（例如，ls 的输出行共有 8 个字段，因此变量 NF 的值就是 8，$NF 指向的就是输入行中的第 8 个字段，在这个例子中就是文件名）。

记住，读取 awk 变量时不需要使用 $（这一点和 bash 变量不同）。NF 本身就是一个有效的变量引用。在其之前加上 $ 就将其含义从“当前行的字段总数”改成了“当前行的最后一个字段”。

7.11.4 参考

- man awk
- comp.lang.awk FAQ
- Arnold Robbins 所著的 *Effective awk Programming, 4th Edition*（O'Reilly 出版）
- Arnold Robbins 与 Dale Dougherty 合著的 *sed & awk, 2nd Edition*（O'Reilly 出版）

7.12 颠倒每行的单词

7.12.1 问题

你想按照逆序输出输入行中的单词。

7.12.2 解决方案

```
$ awk '{
>     for (i=NF; i>=0; i--) {
>         printf "%s ", $i;
>     }
```

```
>     printf "\n"
> }' <filename>
```

字符 > 不用你输入，shell 会输出该字符来提醒你还没有敲完命令（shell 在查找能配对的单引号）。由于 awk 程序位于单引号中，因此 bash shell 允许我们输入多行代码，同时使用 > 作为辅助提示符，直到我们给出与先前匹配的结束单引号。考虑到可读性，我们在程序中加入了空白字符，不过也完全可以写成一行。

```
$ awk '{for (i=NF; i>=0; i--) {printf "%s ", $i;} printf "\n"
}'<filename>
```

7.12.3 讨论

awk 语言的 for 循环语法和 C 语言中的非常相似。前者甚至还支持用 printf 进行格式化输出，这自然也是从 C 语言（以及 bash）中学来的。我们用 for 循环从最后一个字段开始倒着处理到第一个字段，同时输出每个字段的内容。我们特意没有在第一个 printf 处加上 \n，这是因为想将多个字段输出在同一行中。for 循环结束后，输出换行符来结束输出行。

对于 $i 的引用，awk 与 bash 的含义大相径庭。在 bash 中，$i 表示要获取保存在变量 i 中的值。但在 awk 中，和大多数编程语言一样，要想引用变量 i 的值，写作 i 即可。那么 $i 在 awk 中是什么意思呢？变量 i 的值被解析成一个数字，然后“美元符号 - 数字”的表达式会被理解为引用某个输入字段（或单词），即第 i 个字段。因此，随着 i 从最后一个字段倒计到第一个字段，for 循环也就以逆序的方式输出了各个字段的内容。

7.12.4 参考

- man printf(1)
- man awk
- comp.lang.awk FAQ
- Arnold Robbins 所著的 *Effective awk Programming, 4th Edition*（O'Reilly 出版）

- Arnold Robbins 与 Dale Dougherty 合著的 *sed & awk, 2nd Edition*（O'Reilly 出版）
- A.12 节

7.13 汇总数字列表

7.13.1 问题

你需要汇总数字列表，其中有些数字并未出现在行中。

7.13.2 解决方案

用 awk 先过滤出待汇总的字段，然后再做汇总。这里我们要对 ls -l 命令输出的文件大小进行汇总。

```
ls -l | awk '{sum += $5}; END {print sum}'
```

7.13.3 讨论

我们要汇总 ls -l 输出的第 5 个字段。ls -l 的输出如下所示：

```
-rw-r--r-- 1 albing users 267 2005-09-26 21:26 lilmax
```

各个字段分别为：权限、链接、所有者、所属组、大小（以字节为单位）、最后一次修改日期、最后一次修改时间，以及文件名。我们只对文件大小感兴趣，因此在 awk 程序中用 $5 来引用该字段。

我们在花括号（{}）里放置了两段 awk 代码，注意，awk 程序中可以有多个代码段（或代码块）。前有关键词 END 的代码块仅在程序其他部分完成后运行一次。同样，你可以在代码块前添加 BEGIN 关键词并提供读取输入前运行的代码。BEGIN 代码块可用于初始化变量，这里我们用它来初始化 sum，不过 awk 能够确保变量内容一开始为空。

如果查看 ls -l 命令的输出，你会注意到第一行是文件总量信息[2]，与其他行的预期格式不一致。

[2]该行给出了目录下的文件总量，其形式为“total *N*”（*N* 代表具体数量）。——译者注

我们有两种处理方法。第一种方法是，假装该行不存在，之前的解决方案用到的就是这个方法。因为用不着的这行并没有第 5 个字段，所以 $5 为空，汇总结果不会受到影响。

更负责的做法，即第二种方法是跳过这一行。将 ls -l 命令的输出传给 awk 之前，先使用 grep。

```
ls -l | grep -v '^total' | awk '{sum += $5}; END {print sum}'
```

或者在 awk 中完成类似操作。

```
ls -l | awk '/^total/{next} {sum += $5}; END {print sum}'
```

^total 是一个正则表达式，意思是“位于行首的字母 t-o-t-a-l”（开头的 ^ 将搜索锚定在行首）。对于匹配该正则表达式的输入行，执行与之关联的代码块。第二个代码块（sum）前面什么都没有，这意味着 awk 会对所有输入行执行该代码块（不管是否匹配正则表达式）。

针对“total”进行特殊处理的目的是想要将其从汇总过程中排除。因此，在 ^total 对应的代码块中，我们使用了 next 命令，它会停止处理该输入行，并从下一行输入重新开始。因为下一行并不是以“total”起始，所以 awk 将对其执行第二个代码块。我们也可以用 getline 代替 next 命令。getline 并不会重新匹配开头的模式，而是继续往后匹配。注意，在 awk 编程中，代码块的顺序很重要。

7.13.4 参考

- man awk
- comp.lang.awk FAQ

- Arnold Robbins 所著的 *Effective awk Programming, 4th Edition*（O'Reilly 出版）
- Arnold Robbins 与 Dale Dougherty 合著的 *sed & awk, 2nd Edition*（O'Reilly 出版）

7.14 用awk统计字符串出现次数

7.14.1 问题

你需要统计多个不同字符串出现的次数，其中包括一些事先不知道具体是什么的字符串。也就是说，你要统计的并不是已经确定好的一组字符串的出现次数。相反，你会在数据中碰到一些未知字符串并对其进行统计。

7.14.2 解决方案

可以用 awk 的关联数组（其他语言中也称为**散列**或**字典**）来搞定。

在这个例子中，我们要统计系统中各个用户都拥有多少文件。`ls -l` 输出的第三个字段就是用户名，因此使用该字段（`$3`）作为数组索引，并增加数组元素（参见例 7-1）。

例 7-1 ch07/asar.awk

```
#!/usr/bin/awk -f
# 实例文件：asar.awk
# Awk关联数组
# 用法：ls -lR /usr/local | asar.awk

NF > 7 {
    user[$3]++
}

END {
    for (i in user) {
        printf "%s owns %d files\n", i, user[i]
    }
}
```

这里我们调用 awk 的方式有点不一样。由于这个 awk 脚本有些复杂，我们将它放在了单独的文件中。-f 选项告诉 awk 从文件中读取脚本，不过也可以在脚本中使用 shebang 行 [3]#!/usr/bin/awk。

[3]shebang 这个词其实是两个字符名称（sharp-bang）的简写。在 Unix 的行话中，用 sharp 或 hash（有时候是 mesh）来称呼字符“#”，用 bang 来称呼惊叹号“!”，因而 shebang 合起来就代表了这两个字符。——译者注

```
$ ls -lR /usr/local | awk -f asar.awk
bin owns 68 files
albing owns 1801 files
root owns 13755 files
man owns 11491 files
$
```

7.14.3 讨论

我们用条件 NF > 7 来限制执行部分 awk 脚本，以此过滤不包含文件名的那些行，此类行出现在 ls -lR 的输出中是为了提高可读性，其中包括用于分隔不同目录的空行以及每个子目录中的文件总量信息。这种行包含的字段（单词）不多。左花括号前的 NF > 7 并没有出现在斜线中，这表明它并非正则表达式。它其实就是一个条件表达式，和 if 语句中使用的差不多，用于判断条件的真假。NF 是一个特殊的内建变量，其中保存着当前输入行的字段总数。因此，如果某个输入行的字段（单词）数超过了 7 个，则由花括号中的语句进行处理。

最重要的是下面这行。

```
user[$3]++
```

这里用户名（如 bin）被用作数组索引。之所以称为**关联数组**，是因为使用散列表（或类似机制）将每个独一的字符串与数值关联了起来。这些事情已经由 awk 在幕后帮你做好了；字符串比较或查找之类的都不用你操心。

建立好关联数组之后，似乎不那么容易获得数组元素的值了。为此，awk 提供了一种特殊形式的 for 循环。与数字形式的 for(i=0;

i<max; i++) 不同，它有专用于关联数组的语法。

```
for (i in user)
```

在该语句中，变量 i 连续从数组 user 的索引中获得值（没有什么特定顺序）。在示例中，这意味着 i 在每次循环中都会得到一个值（bin、albing、root、man）。如果以前还没见识过关联数组，希望这次能让你大开眼界，印象深刻。关联数组是 awk（以及 Perl）的一个强大特性。

7.14.4 参考

- man awk
- comp.lang.awk FAQ
- Arnold Robbins 所著的 *Effective awk Programming, 4th Edition*（O'Reilly 出版）
- Arnold Robbins 与 Dale Dougherty 合著的 *sed & awk, 2nd Edition*（O'Reilly 出版）
- 7.15 节

7.15 用bash统计字符串出现次数

7.15.1 问题

你需要统计多个字符串出现的次数，其中包括一些事先不知道具体是什么的字符串。也就是说，你要统计的并不是已经确定好的一组字符串的出现次数。相反，你会在数据中碰到一些未知字符串并需要对其进行统计。

7.15.2 解决方案

如果使用的是 bash 4.0 或更高版本，那么可以用关联数组（其他语言中也称为**散列**或**字典**）来搞定。

在这个例子中，我们要统计系统中的各个用户都拥有多少文件。ls -l 输出的第 3 个字段就是用户名，因此我们用该字段（$3）作为数组索引，并增加数组元素（参见例 7-2）。

例 7-2 ch07/cnt_owner.sh

```
# 实例文件：cnt_owner
# 使用bash统计文件所有者
# 将"ls -l"的输出通过管道传入该脚本

declare -A AACOUNT
while read  -a LSL
do
    # 只考虑包含7个及以上字段的行
    if (( ${#LSL[*]} > 7 ))          # 数组大小
    then
        NDX=${LSL[3]}                # 字符串赋值
        (( AACOUNT[${NDX}] += 1 ))   # 算术递增
    fi
done

for VALS in "${!AACOUNT[@]}"         # 各个元素的索引
do
    echo $VALS "owns" ${AACOUNT[$VALS]} "files"
done
```

我们可以像下面这样调用该脚本并获得结果。

```
$ ls -lR /usr/local | bash cnt_owner.sh
bin owns 68 files
root owns 13755 files
man owns 11491 files
albing owns 1801 files
$
```

7.15.3 讨论

read -a LSL 每次读取一行，将其中的各个单词（由空白字符分隔）分别赋给数组 LSL 的各元素。通过检查数组大小，我们可以确定读入了多少个单词，以此过滤不包含文件名的那些行。它们属于 ls -lR 输出的一部分，通常是为了提高可读性，因为其中包含分隔不同目录的空行和各个子目录文件总量的统计。对于我们的脚本而

言，这都是些无用信息，好在这种行并不像其他行那样包含太多的字段（单词），不难分辨。

只有那些至少有 7 个单词的行，我们才从中提取第 3 个单词（文件所有者）并将其作为关联数组的索引。像 LSL 这样的标准数组，数组元素通过整数索引来引用。对于关联数组，索引可以是字符串。

为了输出结果，我们需要遍历数组索引。${AACOUNT[@]} 会生成数组所有元素值的列表，如果加上惊叹号变成 ${!AACOUNT[@]}，则得到的是该数组所有索引的列表。

注意，输出并不遵循特定的顺序（这与散列算法的实现有关）。要想按照用户名或文件数量排序，可以通过管道将结果传给 sort 命令。

7.15.4 参考

- 7.14 节
- 7.16 节

7.16 用便捷的直方图展示数据

7.16.1 问题

你需要为一些数据生成快捷的屏幕直方图。

7.16.2 解决方案

使用 7.14 节讨论过的 awk 关联数组（参见例 7-3）。

例 7-3 ch07/hist.awk

```
#!/usr/bin/awk -f
# 实例文件：hist.awk
# 用Awk生成直方图
# 用法：ls -lR /usr/local | hist.awk

function max(arr, big)
```

```
{
    big = 0;
    for (i in user)
    {
        if (user[i] > big) { big=user[i];}
    }
    return big
}

NF > 7 {
    user[$3]++
}
END {
    # 进行缩放
    maxm = max(user);
    for (i in user) {
        #printf "%s owns %d files\n", i, user[i]
        scaled = 60 * user[i] / maxm ;
        printf "%-10.10s [%8d]:", i, user[i]
        for (i=0; i<scaled; i++) {
            printf "#";
        }
        printf "\n";
    }
}
```

用 7.14 节中的输入来运行该脚本，可以得到：

```
$ ls -lR /usr/local | awk -f hist.awk
bin        [      68]:#
albing     [    1801]:#######
root       [
13755]:##################################################
man        [   11491]:############################################
$
```

7.16.3 讨论

我们可以将 max 对应的那部分代码放进 END 代码块的开头，但这里是想告诉你也可以在 awk 中定义函数。这里用到了一个比较复杂的 printf 语句。字符串格式 %-10.10s 会将字符串左对齐并填充至 10 个字符宽，同时限制长度为 10 个字符，多出的字符会被全部截断。整数格式 %8d 确保输出的整数宽度为 8 个字符。通过使用相

同的屏幕空间（无论用户名的长度或整数大小是多少），可以确保每个直方图的起始位置相同。

与 awk 中的所有算术运算一样，缩放计算使用浮点数完成，除非调用内建的 int() 函数来明确截断结果。我们没有这么做，这意味着 for 循环至少会执行一次，这样一来，最少量的数据也会显示一个 # 标记。

for(i in user) 循环返回的数据并不遵循特定顺序，可能是基于底层散列表的某些便利排序（convenient ordering）。要想按照文件数量或用户名排序显示直方图，那就得加入一些排序处理。一种方法是将脚本拆成两部分，将第一部分的输出发送给 sort 命令，然后通过管道将排序后的结果传给第二部分，以输出直方图。

7.16.4 参考

- man awk
- comp.lang.awk FAQ
- Arnold Robbins 所著的 *Effective awk Programming, 4th Edition*（O'Reilly 出版）
- Arnold Robbins 与 Dale Dougherty 合著的 *sed & awk, 2nd Edition*（O'Reilly 出版）
- 7.14 节
- 7.17 节
- 8.1 节

7.17 用bash轻松实现直方图

7.17.1 问题

你想在不借助外部程序的情况下用 bash 计算并绘制直方图。有没有可能？

7.17.2 解决方案

当然没问题，这多亏了关联数组。bash 4.0 及更高版本都可使用。可以将 7.15 节中的代码作为基础，唯一的不同在于输出部分。首先，遍历所有的数组元素值来找出其中的最大值，然后以此对直方图输出进行缩放以适应页面大小。

```
BIG=0
for VALS in "${!UCOUNT[@]}"
do
    if (( UCOUNT[$VALS] > BIG )) ; then BIG=${UCOUNT[$VALS]} ; fi
done
```

有了最大值（保存在 BIG 中）后，为每个数组元素输出一行。

```
#
# 输出直方图
#
for VALS in "${!UCOUNT[@]}"
do
    printf "%-9.9s [%7d]:" $VALS ${UCOUNT[$VALS]}
    # 缩放至最大值（BIG）；注：整数
    SCALED=$(( ( (59 * UCOUNT[$VALS]) / BIG) +1 ))
    for ((i=0; i<SCALED; i++)) {
        printf "#"
    }
    printf "\n"
done
```

7.17.3 讨论

与 7.5 节中一样，${!UCOUNT[@]} 至关重要。可以通过它得到数组的索引值列表（这个示例为数组 UCOUNT）。for 循环每次获得一个值，并用它作为数组索引来获得该用户的文件数量。

我们将直方图缩放至 59 个字符宽度，然后再多加 1 个字符宽度，这是为了令任何非 0 值 [4] 都能在直方图上至少生成一个标记。这在 awk 版本（参见 7.16 节）中不是什么问题，因为 awk 使用的是浮点运算，但 bash 版本使用的是整数运算，执行除法后，过小的值会变成 0。

[4]意思是，哪怕用户只有一个文件，也能在直方图上生成标记。——译者注

7.17.4 参考

- 7.15 节
- 7.16 节

7.18 显示匹配短语之后的文本段落

7.18.1 问题

你要在文档中搜索某个短语，并希望找到该短语后还能显示之后的文本段落。

7.18.2 解决方案

假设在一个简单的文本文件中，**段落**就是空行之间的所有文本，因此空行的出现就意味着段落的结束。有鉴于此，我们来看看下面这个短小精悍的 awk 脚本。

```
$ cat para.awk
/keyphrase/ { flag=1 }
flag == 1 { print }
/^$/ { flag=0 }

$ awk -f para.awk < searchthis.txt
```

7.18.3 讨论

这里只有 3 个简单的代码块。当输入行匹配正则表达式（这里是单词“keyphrase”）时，调用第一个代码块。只要“keyphrase”出现在输入行内的任何位置，就算匹配，然后执行该代码块。第一个代码块只干一件事：设置标志变量。

对于所有的输入行，第二个代码块都会被调用，因为其左花括号之前没有正则表达式。即便是匹配“keyphrase”的输入行，也会调用该代码块（如果不想这样，可以在第一个代码块中使用 next 语句）。第二个代码块所做的是：仅当标志变量设置时，打印整个输入行。

第三个代码块有对应的正则表达式，如果匹配，则重置（关闭）标志变量。这里的正则表达式使用了两个带有特殊含义的字符：脱字符（^），如果用作正则表达式的第一个字符，就匹配行首位置；美元符号（$），如果用作正则表达式的最后一个字符，就匹配行尾位置。因此，正则表达式 ^$ 匹配的是一个空行，行首和行尾之间没有任何字符。

我们可以对空行使用一个稍微复杂的正则表达式，使其也能够处理只包含空白字符的行，而不只是彻头彻尾的空行。修改后的第 3 行代码如下所示：

```
/^[[:blank:]]*$/ { flag=0 }
```

Perl 程序员会喜欢本节中讨论的这种问题和解决方案，但我们是用 awk 搞定的，因为 Perl（差不多）超出了本书的讨论范围。如果了解 Perl，那么不用犹豫，你可以直接使用。如果不了解，那就全靠 awk 了。

7.18.4 参考

- man awk
- comp.lang.awk FAQ
- Arnold Robbins 所著的 *Effective awk Programming, 4th Edition*（O'Reilly 出版）
- Arnold Robbins 与 Dale Dougherty 合著的 *sed & awk, 2nd Edition*（O'Reilly 出版）

第 8 章　中级shell工具（续）

本章介绍了一些更为实用的工具，它们不属于 shell，但大量的 shell 脚本中都能发现其身影，你确实应该有所了解。

排序是一项相当常见的任务，也有助于提高可读性，因此有必要知晓 sort 命令。同样，tr 命令可以将一个字符转换或映射为另一个字符，甚至只是删除字符。

这里有一个共同点：这些实用工具不仅可以作为独立命令，还可以作为命令管道中的**过滤器**。它们通常接受一个或多个文件名作为参数，但如果没有指定任何文件名，则从标准输入读取。此外，还会向标准输出写入。这种组合使其易于连接管道中的其他命令，如 *something* | sort | *even more*。

这使得这些工具格外实用，同时避免了大量临时文件的堆砌及其带来的混乱。

8.1　输出排序

8.1.1　问题

你想对输出进行排序，但又不想（再次）为程序或 shell 脚本编写一个自定义排序函数。有没有现成可用的工具？

8.1.2　解决方案

使用 sort 实用工具。通过在命令行上指定文件名，你可以对一个或多个文件进行排序。

```
sort file1.txt file2.txt myotherfile.xyz
```

如果不指定文件名，sort 则会从标准输入读取，这样一来，你就可以通过管道将前一个命令的输出传给 sort。

```
somecommands | sort
```

8.1.3 讨论

对输出进行排序能带来不少便利，如果还不用为每个程序编写排序代码，那就更省事了。shell 的管道功能允许你将 sort 连接到任意程序的标准输出。

sort 有不少选项，最值得记住的有 3 个选项，以下是其中两个。

```
sort -r
```

该选项可以按照逆序进行排序（最后一个变成第一个，第一个变成最后一个）。另外还有：

```
sort -f
```

该选项可以“混用”大小写字母，即不区分大小写。同样的效果也可以用 GNU 的长格式选项来实现。

```
sort --ignore-case
```

先留一个悬念，8.2 节再介绍第三个选项。

8.1.4 参考

- man sort
- 8.2 节

8.2 数字排序

8.2.1 问题

对数值进行排序时，你发现结果有问题。

```
$ sort somedata
2
200
21
250
$
```

8.2.2 解决方案

你得用 -n 选项告诉 sort 将数据作为数值排序。

```
$ sort -n somedata
2
21
200
250
$
```

8.2.3 讨论

第一个排序结果并没有问题（尽管怪异），因为它是对数据按照字母排序的（也就是说，21 排在 200 的后面，因为在字母排序中，1 位于 0 之后）。当然，你需要的可能是按照数值排序，因此得使用 -n 选项。

如果配合 uniq -c 使用，sort -rn 可以非常方便地给出一个按照降序排列的使用频率列表。例如，我们来展示一下系统中最常用的 shell。

```
$ cut -d':' -f7 /etc/passwd | sort | uniq -c | sort -rn
     20 /bin/sh
     10 /bin/false
      2 /bin/bash
      1 /bin/sync
$
```

cut -d':' -f7 /etc/passwd 用于提取 /etc/passwd 文件中各个用户所使用的 shell。然后，必须先进行一次排序，这样 uniq 才

能正常工作。uniq -c 统计连续出现的重复行，这正是我们需要预排序的原因。sort -rn 按照数值进行逆向排序，最常用的 shell 出现在最前面。

如果不需要统计出现次数，只是想得到一个不重复的列表（也就是说，想通过 sort 去掉重复内容），那么可以使用 sort 命令的 -u 选项（无须使用 uniq 命令）。因此，要想找出系统中都有哪些 shell，可以使用下列命令。

```
cut -d':' -f7 /etc/passwd | sort -u
```

8.2.4 参考

- man sort
- man uniq
- man cut

8.3 IP地址排序

8.3.1 问题

你想对数字 IP 地址列表进行排序，但希望按最后部分的数字或在逻辑上按整个地址进行排序。

8.3.2 解决方案

按照最后一部分数字排序（旧语法）。

```
$ sort -t. -n +3.0 ipaddr.list
10.0.0.2
192.168.0.2
192.168.0.4
10.0.0.5
192.168.0.12
10.0.0.20
$
```

按照整个 IP 地址排序（POSIX 语法）。

```
$ sort -t . -k 1,1n -k 2,2n -k 3,3n -k 4,4n ipaddr.list
10.0.0.2
10.0.0.5
10.0.0.20
192.168.0.2
192.168.0.4
192.168.0.12
$
```

8.3.3 讨论

我们知道这些数据都是数值，因此使用了 -n 选项。-t 选项用于指定字段之间的分隔符（这里的分隔符是点号），这样就可以指定先按照哪个字段排序了。在第一个例子中，我们在左边找到第 3 个字段（从 0 开始计数），从该字段的第 1 个字符（也是从 0 开始计数）开始排序，写作 +3.0。

在第二个例子中，我们没有再沿用传统的（已经过时）+pos1 -pos2 格式，而是采用了新的 POSIX 语法规范。与旧格式不同，在 POSIX 语法中，字段是从 1 开始计数的。

```
sort -t . -k 1,1n -k 2,2n -k 3,3n -k 4,4n ipaddr.list
```

哇，真难看。以下是旧格式，也没好到哪去。

```
sort -t. +0n -1 +1n -2 +2n -3 +3n -4
```

两者都用 -t. 来定义字符分隔符，但指定排序字段时就大不一样了。在 POSIX 语法中，-k 1,1n 的意思是“排序起止范围从第一个字段（1）开头至（,）第一个字段（1）末尾，按照数值排序（n）”。只要搞明白这部分的写法，剩下的就很简单了。如果使用多个排序字段，重要的是一定要告诉 sort 字段的结束位置。默认是直到行尾才结束，但这通常并不符合我们的要求，要是你不明白这一点，着实会摸不着头脑。

sort 使用的排序方法受到语言环境设置的影响。如果得到的结果与预期不符，你可以检查一下这方面的设置。

排序结果在不同系统上会出现不一致的情况，这取决于 sort 命令是否默认使用了**稳定**排序。如果排序字段内容相同，那么稳定排序会在排序结果中保留其原始顺序。Linux 和 Solaris 默认没有使用稳定排序，而 NetBSD 使用了稳定排序。-S 选项在 NetBSD 上用于关闭稳定排序，但在其他 sort 版本中，该选项用于设置缓冲区大小。

假设我们有一个普通文件：

```
10.0.0.4         # mainframe
192.168.0.12     # speedy
10.0.0.20        # lanyard
192.168.0.4      # office
10.0.0.2         # sluggish
192.168.0.2      # laptop
```

如果在 Linux 或 Solaris 系统上执行下列命令：

```
sort -t. -k4n ipaddr.list
```

或者在 NetBSD 系统上执行下列命令：

```
sort -t. -S -k4n ipaddr.list
```

我们得到排序后的结果如表 8-1 中第一列所示。去掉 NetBSD 系统中的 sort 命令的 -S 选项后，结果如表中第二列所示。

表8-1：Linux/Solaris/NetBSD系统上的排序结果比对

Linux/Solaris（默认）和NetBSD（-s选项）	NetBSD的稳定排序（默认）
192.168.0.2 # laptop	10.0.0.2 # sluggish
10.0.0.2 # sluggish	192.168.0.2 # laptop

Linux/Solaris（默认）和NetBSD（-s选项）	NetBSD的稳定排序（默认）
192.168.0.4 # office	10.0.0.4 # mainframe
10.0.0.4 # mainframe	192.168.0.4 # office
192.168.0.12 # speedy	192.168.0.12 # speedy
10.0.0.20 # lanyard	10.0.0.20 # lanyard

在输入文件 ipaddr.list 中，以 192.168 开头的地址最先出现，接着是 10 开头的地址，如果出现平局（tie），也就是两个排序字段的值一样，那么稳定排序会将 192.168 开头的地址放在排序结果的前面。在表 8-1 中，我们可以看到 laptop 和 sluggish 就是这种情况，因为两者的第四个字段都是 2；mainframe 和 office 也是同理，两者的第四个字段都是 4。在 Linux 的默认排序（以及指定了 -S 选项的 NetBSD）中，无法确保排序结果是稳定的。

要想追求简单实用，可以按照 IP 地址列表中的文本排序。这一次我们选用 # 字符作为分隔符，并按照字母顺序对第二个字段排序，得到的结果如下所示。

```
$ sort -t'#' -k2 ipaddr.list
10.0.0.20       # lanyard
192.168.0.2     # laptop
10.0.0.5        # mainframe
192.168.0.4     # office
10.0.0.2        # sluggish
192.168.0.12    # speedy
$
```

排序范围从第二个字段开始，直到行尾。因为每行仅有一个分隔符（#），所以不必指定范围的结束位置，不过你也可以写成 -k2,2。

8.3.4 参考

- man sort
- bash tarball 中提供的 ./functions/inetaddr（附录 B）

8.4 提取部分输出

8.4.1 问题

你想要根据列的位置获取固定宽度或分列数据中的一部分。

8.4.2 解决方案

用带有 -c 选项的 cut 命令提取特定列。[1]

[1]注意，本节中所列举的 ps 命令的例子仅适用于某些系统，如 CentOS-4、Fedora Core 5、Ubuntu，但由于列的位置不同，在 Red Hat 8、NetBSD、Solaris、masOS 中会出现错乱。

```
$ ps -l | cut -c12-15
PID
5391
7285
7286
$
```

或者：

```
$ ps -elf | cut -c58-
(不再显示输出)
$
```

8.4.3 讨论

使用 cut 命令时，我们需要指定保留每行的哪一部分。在第一个例子中，我们保留第 12 列（从 1 开始计数）到第 15 列（包含在

内）。在第二个例子中，我们指定从第 58 列开始，但没有指定结束位置，因此 cut 会一直提取到行尾。

到目前为止，我们看过的大部分数据处理是基于**字段**，字段之间由**分隔符**隔开。cut 命令也可以这么做，但它是 bash 中少数也能够轻松处理分列数据的工具之一（通过 -c 选项）。

除了列，cut 也可以输出字段，不过相较于其他工具（如 awk），限制要更多一些。字段之间的默认分隔符是制表符，你也可以用 -d 选项指定其他分隔符。以下是一个处理字段的 cut 命令。

```
cut -d'#' -f2 < ipaddr.list
```

与其等价的 awk 命令如下所示。

```
awk -F'#' '{print $2}' < ipaddr.list
```

你甚至还可以用多个 cut 来处理不一致的分隔符。使用带有正则表达式的 awk 可能效果更好，不过有时一连串快而糙的 cut 更容易理解和键入。

接下来我们将展示如何提取一对中括号之间的字段。注意，第一个 cut 使用左中括号作为分隔符（-d'['），并提取第 2 个字段（-f2，字段从 1 开始计数）。在此基础上，第二个 cut 使用右中括号作为分隔符（-d']'），并提取第 1 个字段。

```
$ cat delimited_data
Line [l1].
Line [l2].
Line [l3].

$ cut -d'[' -f2 delimited_data | cut -d']' -f1
l1
l2
l3
$
```

8.4.4 参考

- man cut
- man awk

8.5 删除重复行

8.5.1 问题

选择或排序完数据后，你发现结果中出现了很多重复行。你想要去除这些重复内容。

8.5.2 解决方案

你有两种选择。如果只对输出进行排序，可以使用 sort 命令的 -u 选项。

```
somesequence | sort -u
```

如果不用 sort，则只需要通过管道将输出传给 uniq。但有一个前提：输出是有序的，相同的行集中在一起。

```
somesequence | uniq > myfile
```

8.5.3 讨论

因为 uniq 要求数据已经事先完成排序，所以我们更可能只使用带有 -u 选项的 sort，除非还需要统计重复行的数量（-c，参见 8.2 节）或只查看重复行（-d），此时才轮到 uniq 上阵。

> 可别不小心把重要的文件给误覆盖了。uniq 命令的参数有点怪。大多数 Unix/Linux 命令可以在命令行上接受多个输入文件，但 uniq 不然。事实上，uniq 的第一个（非选项）参数被视为（唯一的）输入文件，第二个参数（如果指定了）被视为**输出**文件。因此，如果你在命令行上指定了两个文件名，那么第二个文件会在毫无预警的情况下被覆盖。

8.5.4 参考

- man sort
- man uniq
- 8.2 节

8.6 压缩文件

8.6.1 问题

你需要压缩几个文件，但又不确定最佳的压缩方法。

8.6.2 解决方案

首先，你得知道在传统 Unix 中，归档（或者称作合并）和文件压缩是两种操作，各自使用的工具也不同，而在 DOS 和 Windows 世界中，两者通常是一回事，一种工具就能完成。使用 tar（tape archive，磁带归档）命令合并多个文件或目录，然后再用 compress、gzip 或 bzip2 工具进行压缩，得到的就是“tarball”，其文件名称类似于 tarball.tar.Z、tarball.tar.gz、tarball.tgz 或 tarball.tar.bz2。话虽如此，包括 zip 在内的很多其他工具也是支持的。

为了使用正确的格式，需要搞清楚数据将用于何处。如果纯粹只是压缩一些文件自用，怎么方便怎么来就行。如果其他人也要使用你的数据，那就要考虑对方所用的平台以及习惯。

Unix 的传统 tarball 是 tarball.tar.Z，但如今 gzip 用的更多，xz 和 bzip2（压缩率比 gzip 更好）也正在逐渐普及。但还有一个工具问题。有些版本的 tar 允许在创建归档的同时用指定的压缩工具自动压缩。有些版本则不行。

Unix 或 Linux 普遍接受的格式是 tarball.tar.gz，其创建方式如下所示：

```
$ tar cf tarball_name.tar directory_of_files
$ gzip tarball_name.tar
$
```

如果使用的是 GNU tar，那么 -Z 可用于压缩（该选项已经过时）、-z 用于指定 gzip（最稳妥）、-j 用于指定 bzip2（压缩率最高）。别忘了使用适合的文件名，tar 可不会帮你自动生成。例如：

```
tar czf tarball_name.tgz directory_of_files
```

虽然 tar 和 gzip 可用于很多平台，但如果涉及 Windows，最好还是使用 zip，因为它几乎是通用的。

```
zip -r zipfile_name directory_of_files
```

在 Unix 以及其他几乎所有平台上，zip 和 unzip 都是由 InfoZip 软件包提供的，但可能并未默认安装。与多数其他 Unix 工具不同，这些工具只执行命令本身的话，输出的是一些辅助的用法信息。另外注意，-l 选项可以将 Unix 行尾转换成 DOS 行尾，-ll 则执行相反操作。

8.6.3 讨论

这里要讨论的压缩算法和工具太多了，其中还包括 ar、arc、arj、bin、bz2、cab、jar、cpio、deb、hqx、lha、lzh、rar、rpm、uue、zoo。

使用 tar 时，我们**强烈**建议采用相对路径来保存所有文件。如果使用绝对路径，可能会误覆盖系统中的其他文件。不使用任何路径的话，提取出的文件会七零八散地出现在用户当前目录下（参见 8.8 节）。推荐用法是采用当前正在处理的数据名称和可能的版本号。表 8-2 展示了一些例子。

表8-2：tar实用工具文件命名的好坏示例

好	坏
./myapp_1.0.1	myapp.c myapp.h myapp.man
./bintools	/usr/local/bin

值得一提的是，Red Hat Package Manager（RPM，Red Hat 软件包管理器）文件其实就是带有头部信息的 CPIO 文件。你可以用名为 rpm2cpio 的 shell 或 Perl 脚本将头部剥离，提取其中的文件。

```
rpm2cpio some.rpm | cpio -i
```

Debian 的 .deb 文件其实就是 ar 归档，其中包含经过 gzip 或 bzip 压缩的 tar 归档。可以用标准工具 ar、gunzip 或 bunzip2 将其提取出来。

很多 Windows 工具（如 WinZip、PKZIP、FilZip、7-Zip）能处理这里提到的多数或全部格式，甚至包括 tarball 和 RPM。

8.6.4 参考

- man tar
- man gzip
- man bzip2
- man compress
- man zip
- man rpm
- man ar
- man dpkg
- Info-ZIP 主页
- RPM Guide（“Manipulating Package Files with rpm2cpio”一节）

- 维基百科（Deb_(file_format)）
- RPM Package Manager 主页
- 维基百科（RPM Package Manager）
- 7.9 节
- 8.7 节
- 8.8 节
- 17.3 节

8.7 解压文件

8.7.1 问题

你需要解压一个或多个扩展名为
.tar、.tar.gz、.gz、.tgz、.Z、.zip 的文件。

8.7.2 解决方案

先搞清楚要处理的文件，然后选择正确的工具。表 8-3 列出了常见的扩展名以及能够处理该类文件的程序。此时 `file` 命令可以派上用场，它能告诉你文件是哪种类型，哪怕是文件名有误。

表8-3：常见的文件扩展名和压缩工具

文件扩展名	命令
.tar	`tar tf`（列出文件内容），`tar xf`（提取文件内容）
.tar.gz，.tgz	GNU tar：`tar tzf`（列出文件内容），`tar xzf`（提取文件内容） 否则：`gunzip` *`file`* `&& tar xf` *`file`*
.tar.bz2	GNU tar：`tar tjf`（列出文件内容），`tar xjf`（提取文件内容） 否则：`gunzip2` *`file`* `&& tar xf` *`file`*

文件扩展名	命令
.tar.Z	GNU tar：tar tZf（列出文件内容），tar xZf（提取文件内容） 否则：uncompress *file* && tar xf *file*
.zip	unzip（通常没有默认安装）

你也不妨试试 file 命令。

```
$ file what_is_this.*
what_is_this.1: GNU tar archive
what_is_this.2: gzip compressed data, from Unix

$ gunzip what_is_this.2
gunzip: what_is_this.2: unknown suffix -- ignored

$ mv what_is_this.2 what_is_this.2.gz

$ gunzip what_is_this.2.gz

$ file what_is_this.2
what_is_this.2: GNU tar archive
```

8.7.3 讨论

如果文件扩展名不在表 8-3 中，而且 file 命令也无济于事，但你确信该文件属于某种归档，那不妨在网上搜一下。

8.7.4 参考

- 7.9 节
- 8.6 节

8.8 检查tar归档文件中不重复的目录

8.8.1 问题

你想提取归档中的文件，但希望提前知道这些文件会写入哪些目录。可以通过 tar -t 查看 tar 归档文件所包含的内容列表，不过这样可能会产生大量输出，容易造成遗漏。

8.8.2 解决方案

使用 awk 脚本从 tar 归档的内容列表中解析出目录名称，然后用 sort -u 得到不重复的目录名称：

```
tar tf some.tar | awk -F/ '{print $1}' | sort -u
```

8.8.3 讨论

t 选项可以为 f 选项指定的 tar 归档文件生成内容列表。awk 命令通过 -F/ 将字段分隔符指定为非默认的斜线。因此，print $1 会输出路径名中的第一个目录名。

最后，对所有目录名排序并仅输出不重复的那些目录。

如果某行输出包含单个点号，那么部分文件就会被提取到当前目录，因此要确保位于所需要的目录中。

与此类似，如果归档文件中的文件名全都是相对路径，起始部分没有 ./，那么会在当前目录下生成一系列文件。

如果输出中出现了空行，这意味着有些文件采用了绝对路径（路径起始部分为 /）。再次提醒，提取这种归档有可能会误覆盖别的文件。

有些版本的 tar（如 GNU tar）会默认或有选择地剥除路径中的前导 /。这种方式创建的 tarball 要安全得多，但你可别指望所有的 tar 都会这么做。

8.8.4 参考

- man tar
- man awk
- 8.1 节
- 8.2 节
- 8.3 节

8.9 转换字符

8.9.1 问题

你需要将文本中出现的某个字符全部转换成另一个字符。

8.9.2 解决方案

可以使用 tr 命令。例如：

```
tr ';' ',' <be.fore >af.ter
```

8.9.3 讨论

就其最简单的形式而言，tr 命令用第二个参数的第 1 个（仅此）字符替换第一个参数的第 1 个（仅此）字符。

在上述示例中，我们将输入重定向为文件 be.fore，将输出重定向至文件 af.ter，将输入文件中所有的分号替换成了逗号。

为什么要在分号和逗号两边使用单引号呢？这是因为分号在 bash 中有特殊含义，如果不用引号，那么 bash 会将命令拆成两个，这肯定要出错。逗号倒是没有特殊含义，将它放入引号纯粹出于习惯，免得万一忘记什么。坚持使用引号会更安全，这样就永远不会漏写了。

如果使用多个待转换的字符作为第一个参数，要转换成的字符作为第二个参数，那么 tr 命令就会一次性执行多个字符转换。记住，这种转换是一对一进行的。例如：

```
tr ';:.!?' ',' <other.punct >commas.all
```

该命令会将分号、冒号、句号、惊叹号、问号全部转换成逗号。因为第二个参数比第一个参数短，所以不断重复第二个参数的最后一个字符（这个例子中就只有一个）来匹配第一个参数的长度，这样一来，字符就能在转换时一一对应了。

sed 命令也能完成这种转换，只不过语法有点麻烦。tr 命令没有那么厉害，因为它无法使用正则表达式，但 tr 有一些特别的字符区间语法，非常有用，8.10 节将对其进行介绍。

8.9.4 参考

- man tr
- 8.10 节

8.10 将大写字母转换为小写字母

8.10.1 问题

你需要消除文本流中的字母大小写差异。

8.10.2 解决方案

可以使用 tr 命令并指定字符区间，将所有的大写字母（A-Z）转换成小写字母（a-z）：

```
tr 'A-Z' 'a-z' <be.fore >af.ter
```

tr 还有一种特别的语法，可用于指定大小写字母转换的区间：

```
tr '[:upper:]' '[:lower:]' <be.fore >af.ter
```

有些版本的 tr 支持当前语言环境的对照顺序（collating sequence），A-Z 不一定总是当前语言环境中的大写字母集合。尽可

能使用 [:lower:] 和 [:upper:] 来避开这种问题，不过如此一来，就无法使用子范围（如 N-Z 和 a-m）了。

8.10.3 讨论

虽然 tr 不支持正则表达式，但支持字符区间。确保两个参数包含相同数量的字符即可。如果第二个参数长度较短，那么其最后一个字符会不断重复，直到匹配第一个参数的长度。如果第一个参数较短，第二个参数则会被截断，以匹配第一个参数的长度。

以下使用简单的替换加密算法对一段文本消息进行了简单的编码，这种算法将每个字符偏移 13 个位置（ROT13）。ROT13 有一个值得注意的地方：相同的过程既可以用于文本加密，也可以用于文本解密。

```
$ cat /tmp/joke
Q: Why did the chicken cross the road?
A: To get to the other side.

$ tr 'A-Za-z' 'N-ZA-Mn-za-m' < /tmp/joke
D: Jul qvq gur puvpxra pebff gur ebnq?
N: Gb trg gb gur bgure fvqr.

$ tr 'A-Za-z' 'N-ZA-Mn-za-m' < /tmp/joke | tr 'A-Za-z' 'N-ZA-Mn-
za-m'
Q: Why did the chicken cross the road?
A: To get to the other side.
```

8.10.4 参考

- man tr
- 维基百科（Rot13）
- 8.9 节

8.11 将DOS文件转换为Linux格式

8.11.1 问题

你需要将 DOS 格式的文本文件转换为 Linux 格式。在 DOS 中，每行文本由一对字符作结：回车符和换行符。在 Linux 中，每行文本仅由单个换行符作结。该如何删除多出的那个 DOS 字符呢？

8.11.2 解决方案

可以用 tr 命令的 -d 选项删除指定字符。例如，要想删除所有的回车符（\r），可以使用下列命令。

```
tr -d '\r' <file.dos >file.txt
```

这会删除文件中的所有 \r 字符，而不仅仅是行尾的。典型的文本文件中极少会在行内出现该字符，不过也并非不可能。如果对此有所顾忌，可以考虑 dos2unix 和 unix2dos 程序。

8.11.3 讨论

tr 能够识别包括 \r（回车符）和 \n（换行符）在内的一些特殊转义序列。表 8-4 列出了各种转义序列。

表8-4：tr能够识别的转义序列

转义序列	含义
\ooo	以八进制值描述的字符（1~3 位八进制数字）
\\	反斜线（转义反斜线本身）
\a	铃声，ASCII BEL 字符（因为“b”已经用作退格符）
\b	退格符
\f	换页符

转义序列	含义
\n	换行符
\r	回车符
\t	制表符（有时候也叫作“水平”制表符）
\v	垂直制表符

8.11.4 参考

- man tr

8.12 删除智能引号

8.12.1 问题

你想从 MS Word 的文档中提取简单的 ASCII 文本，但将其另存为文本时，仍然有一些奇怪的字符。

8.12.2 解决方案

将那些奇怪的字符转换成简单的 ASCII 字符。

```
tr '\221\222\223\224\226\227' '\047\047""--' <odd.txt >plain.txt
```

8.12.3 讨论

这种“智能引号”[2] 出自 Windows-1251 字符集，也可能会出现在保存为文本的电子邮件消息中。

[2]从形态上看，引号有两种：直引号（straight quote，也称为 dumb quote）和弯引号（curly quote，也称为 smart quote）。智能引号的设计目的是让用户在视觉上区分左引号和右引号，以免忘记关闭引号。——译者注

要想清理这种文本，得用到 tr 命令，将值为 221 和 222（八进制）的弯单引号转换成简单的单引号 [3]。这里我们用八进制值（047）来指定后者，这样更容易处理，因为 shell 引用字符串时要用到单引号。223 和 224（八进制）分别是左右弯双引号。可以将直双引号直接放入第二个参数，因为外部的单引号能够避免 shell 对双引号做出解释。226 和 227（八进制）是破折号，它们会被转换成连字符（技术上并不需要第二个参数中的第二个连字符，因为 tr 会重复最后一个字符，使之匹配第一个参数的长度，不过最好还是写清楚）。

[3]也就是直引号。——译者注

8.12.4 参考

- man tr
- 维基百科（Quotation mark，其中所包含的有关引号和相关字符集的问题可能比你想象中的还多）

8.13 统计文件的行数、单词数或字符数

8.13.1 问题

你想知道指定文件有多少行、多少个单词以及多少个字符。

8.13.2 解决方案

在命令替换中使用 wc（word count，单词统计）命令。

wc 的正常输出如下所示：

```
$ wc data_file
       5     15         60 data_file

# 只显示行数
$ wc -l data_file
       5 data_file

# 只显示单词数
$ wc -w data_file
      15 data_file

# 只显示字符数（通常等同于字节数）
$ wc -c data_file
      60 data_file

# 注意文件大小为60B
$ ls -l data_file
-rw-r--r--  1 jp  users   60B Dec    6 03:18 data_file
```

你可能想按照以下方式来做：

```
data_file_lines=$(wc -l "$data_file")
```

这不会如你所愿，因为得到的是 "5 data_file" 这样的值。你可能还会看到：

```
data_file_lines=$(cat "$data_file" | wc -l)
```

丢掉**毫无必要的** cat 吧，使用以下命令来避免文件名问题。

```
data_file_lines=$(wc -l < "$data_file")
```

8.13.3 讨论

如果你使用的 wc 版本支持语言环境，在某些字符集中，文件的字符数量并不等于字节数量。

8.13.4 参考

- man wc

- 15.7 节

8.14 重新编排段落

8.14.1 问题

有些文本行要么太长，要么太短，你希望重新编排来提高可读性。

8.14.2 解决方案

使用 fmt 命令。

```
fmt mangled_text
```

也可以选择指定行的目标长度和最大长度。

```
fmt 55 60 mangled_text
```

8.14.3 讨论

fmt 的棘手之处是，它期望使用空行来分隔标题和段落。如果输入文件中没有这些空行，那么 fmt 就无法区分不同的段落和同一段落中额外的换行符，结果就是你会得到一个行长度正确的大段落。

pr 命令也可用于文本格式化。

8.14.4 参考

- man fmt
- man pr

8.15 你不知道的less

8.15.1 问题

“less 可不止如此（less is more）！”[4] 你希望能够更充分地利用分页程序 less 的特性。

[4]这里的 less is more 是一个双关语。原话是由著名的现代主义建筑大师 Ludwig Mies van der Rohe 于 1928 年提出的，意为“少即是多”。本节引用这句话旨在表达，“less 命令还有更多可讲的东西”。——译者注

8.15.2 解决方案

阅读 less 的手册页，使用 $LESS 变量以及~/.lessfilter 和 ~/.lesspipe 文件。

less 可以从 $LESS 变量中获取选项，因此无须再创建包含偏好选项的别名，只需将选项放入该变量即可。长短选项都能接受，在命令行上指定的选项会覆盖变量中的选项。我们建议在 $LESS 变量中使用长选项，因为这种选项形式更易于阅读。例如：

```
export LESS="--LONG-PROMPT --LINE-NUMBERS --ignore-case --QUIET"
```

这只是个开始而已。less 能利用**输入预处理器（input preprocessor）**进行扩展，此处理器可以是程序或脚本，负责对 less 要显示的文件进行预处理。这是通过对环境变量 $LESSOPEN 和 $LESSCLOSE 做相应设置来实现的。

虽然可以自己动手，但时间能省则省（阅读完随后的讨论部分），不妨试试 Wolfgang Friebel 的 lesspipe.sh。该脚本在单独运行时会设置和导出环境变量 $LESSOPEN。

```
$ ./lesspipe.sh
LESSOPEN="|./lesspipe.sh %s"
export LESSOPEN
$
```

你只需将其放入 eval 语句，例如 eval $(/path/to/lesspipe.sh) 或 eval

'/path/to/lesspipe.sh' 中，然后像平常那样使用 less 就行了。1.82 版支持的部分格式包括：

> gzip、compress、bzip2、zip、rar、tar、nroff、ar 归档、pdf、ps、dvi、共享库、可执行文件、目录、RPM、Microsoft Word、OASIS（OpenDocument、Openoffice、Libreoffice）格式、Debian、MP3 文件、图像格式（png、gif、jpeg、tiff……)、utf-16 文本、iso 镜像、可通过 /dev/xxx 访问的可移动介质文件系统。

有一点要注意，这些格式需要各种外部工具，没有的话，lesspipe.sh 示例中的有些特性是无法使用的。根据你当前可用的工具，该软件包中的 ./configure（或 make）脚本还可以生成适用于系统的过滤器版本。

8.15.3 讨论

less 的独特之处在于，在我们碰到过的所有单一测试系统中，它都是已经默认安装好的 GNU 工具。这一点甚至连 bash 都不敢保证。抛开版本差异不谈，less 在所有系统上的表现都一模一样。这实在是很了不起，值得大书特书。

但是，对于 lesspipe* 和 $LESSOPEN 过滤器而言，可就不是这么回事了。除了 8.15.2 节中列出的那些，我们还发现了功能各异的其他版本。

- Red Hat 包含 /usr/bin/lesspipe.sh，但不能像 eval '/path/to/less pipe.sh' 这样使用。
- Debian 包含 /usr/bin/lesspipe，但不能用于 eval，另外还通过~/.lessfilter 文件支持额外的过滤器。
- SUSE Linux 包含 /usr/bin/lessopen.sh，但不能用于 eval。
- FreeBSD 包含一个不起眼的 /usr/bin/lesspipe.sh（不能用于 eval，不支持 .Z、.gz、.bz2）。
- Solaris、HP-UX、其他版本的 BSD 和 macOS 默认什么都没有。

要想知道自己的系统中是否有这些文件，可以试验一下。Debian 系统包含 lesspipe，但并未使用（因为没有定义 $LESSOPEN）。

```
$ type lesspipe.sh; type lesspipe; set | grep LESS
-bash3: type: lesspipe.sh: not found
lesspipe is /usr/bin/lesspipe
$
```

Ubuntu 系统包含并使用了 Debian lesspipe。

```
$ type lesspipe.sh; type lesspipe; set | grep LESS
-bash: type: lesspipe.sh: not found
lesspipe is hashed (/usr/bin/lesspipe)
LESSCLOSE='/usr/bin/lesspipe %s %s'
LESSOPEN='| /usr/bin/lesspipe %s'
$
```

我们推荐你下载、配置并使用 Wolfgang Friebel 的 lesspipe.sh，因为其功能最为全面。另外还建议你阅读 less 的手册页，其内容非常有意思。

8.15.4 参考

- man less
- man lesspipe
- man lesspipe.sh
- less 主页
- lesspipe 主页

第 9 章　查找文件：find、locate、slocate

对你而言，在整个文件系统中搜索文件有多简单？

如果是刚创建了几个文件，恐怕没有比记住这些文件的名字和保存位置更容易的事了。随着文件的增加，你创建了子目录（GUI 中称作**文件夹**）来划分文件。很快，子目录又有了子目录，此时再想记住文件的确切位置就没那么容易了。当然了，现在硬盘越来越大，很容易只顾埋头创建文件，从来不作删除处理（旧文件对有些人也没什么用）。

不过，怎么才能找到上周刚编辑过的文件？或者是保存在某个子目录中的附件（当时看来放在那里还是挺合适的）？要是想将散落在文件系统中的 MP3 文件汇集起来，又该怎么做呢？

图形界面中提供了各种功能来帮助用户搜索文件，用起来都还不错，但如何将 GUI 工具的搜索结果作为输入传给其他命令呢？

bash 和 GNU 工具可以助你一臂之力。它们提供了一些非常强大的搜索功能，允许按照文件名、文件创建或修改日期，甚至是文件内容进行搜索。搜索结果可以发送到标准输出，供其他命令或脚本使用。

别惊讶了，本章内容正是你需要的。

9.1　查找所有的MP3文件

9.1.1　问题

文件系统中到处都是 MP3 音频文件。你想将它们集中到一个位置，组织好后再复制到音乐播放器中。

9.1.2 解决方案

find 命令可以找出符合要求的所有文件并执行命令，将其移动到指定位置。例如：

```
find . -name '*.mp3' -print -exec mv '{}' ~/songs \;
```

9.1.3 讨论

find 命令的语法和其他 Unix 命令不同，其选项并不是那种典型的连字符加上单字母，后面再跟上若干参数。find 命令的选项看起来像是简短的单词 [1]，依照逻辑顺序出现，并描述要查找哪些文件以及如何处理找到的文件（如果存在的话）。这种像单词一样的选项通常称为**谓词**（predicate）。

> [1]准确地说，它们其实是 find 命令的表达式选项（expression option）、测试条件（test）或操作（action）。参见 man find。——译者注

find 命令的第一个参数是待搜索的目录。典型用法是用点号（.）代表当前目录，不过你也可以提供一个目录列表，甚至通过指定根目录（/）来搜索整个文件系统（只要权限允许）。

示例中的第一个选项（谓词 -name）指定了要搜索的文件模式。其语法和 bash 的模式匹配语法差不多，因此 *.mp3 能够匹配所有以“.mp3”结尾的文件名。匹配该模式的文件被认为返回的是真（true），接着将其交给下一个谓词进行处理。

不妨这样想：find 会遍历文件系统，将找到的文件名交给谓词测试。如果谓词返回真，就通过。如果返回假，则不再继续往下进行，会接着处理下一个文件名。

谓词 -print 很简单。它总是返回真，同时会将文件名打印到标准输出，因此，能在谓词序列中通过测试而到达这一步的文件都会输出其名称。

-exec 就有点怪异了。到达这一步的文件名都会变成接下来要执行的命令的一部分。剩下一直到 \; 的这部分就是命令，其中的 {} 会被

替换成已查找到的文件名。因此，在上面的例子中，如果 find 在 ./music/jazz 子目录中找到名为 mhsr.mp3 的文件，那么要执行的命令就会是：

```
mv ./music/jazz/mhsr.mp3 ~/songs
```

所有匹配指定模式的文件都会执行命令。如果找到的文件数量众多，那么命令的执行次数自然也不会少。有时这对系统提出了很高的要求，更好的办法是用 find 查找文件并将文件名输出到数据库，然后将多个参数集中在一起，这样就不用执行那么多次命令了。（但随着计算机的速度越来越快，这个问题已经越来越不是问题。甚至可以说，你的双核或四核处理器正适合干这事。）

9.1.4 参考

- man find
- 1.5 节
- 1.6 节
- 9.2 节

9.2 处理文件名中的怪异字符

9.2.1 问题

你按照 9.1 节中的做法使用 find 命令，却没有得到想要的结果，因为很多文件名中有怪异的字符。

9.2.2 解决方案

首先，要理解 Unix 用户口中的怪异（odd）字符是指“除小写字母和数字之外的任何字符”。因此，大写字母、空格、标点符号、声调字符统统都属于怪异字符，但是你会发现很多歌曲名中的怪异字符远不止这些。

根据字符的怪异程度以及你的系统、工具和目标，可能将替换字符串放进引号就足够了（也就是在 {} 两边加上引号，即 '{}'）。你总得先测试一下命令吧？

如果不行，尝试使用 find 的 -print0 选项和 xargs 的 -0 选项。-print0 告诉 find 不要用空白字符，而是改用空字符（null character）（\0）作为输出的文件名之间的分隔符。-0 告诉 xargs 输入分隔符是空字符。两者配合的效果不错，但并非所有系统都支持这两个命令。

xargs 命令从标准输入中接收以空白字符分隔（指定 -0 时除外）的文件名，然后对尽可能多的文件（略微少于系统的 ARG_MAX 值，参见 15.13 节）执行指定命令。由于调用其他命令会带来不小的开销，因此使用 xargs 可以显著提升操作速度，因为它能够尽量减少命令的调用次数，而不是每个文件都调用。

因此，可以重写 9.1 节中的解决方案来处理怪异字符。

```
find . -name '*.mp3' -print0 | xargs -i -0 mv '{}' ~/songs
```

以下是一个类似示例，它演示了在定位并复制文件时，如何用 xargs 解决路径或文件名中的空格。

```
locate P1100087.JPG PC220010.JPG PA310075.JPG PA310076.JPG | xargs
-i cp '{}' .
```

9.2.3 讨论

这个解决方案存在两个问题。首先，并非所有的 xargs 版本都支持 -i 选项，另外，-i 选项无法实现参数分组，自然也就无法提升执行速度。问题是，mv 命令需要将目标目录作为最后一个参数，而传统的 xargs 只是简单地获取输入，然后将其附加到指定命令的尾部，直到达到上限 [2] 或处理完输入。这样一来，mv 命令肯定报错。有些 xargs 版本提供了 -i 选项，默认使用 {} 作为替换字符串（类似于 find），但加入 -i 选项会导致对每个参数都要执行一次命令。相较于 find 的 -exec，唯一的优势就是能够处理怪异字符。

[2]也就是系统的 ARG_MAX 值。——译者注

xargs 最有效的场景是与 find 以及那种只需要参数列表的命令（如 chmod）配合使用。在处理大量文件名时，你可以看到 xargs 所带来的速度优化。例如：

```
find some_directory -type f -print0 | xargs -0 chmod 0644
```

9.2.4 参考

- man find
- man xargs
- 9.1 节
- 15.13 节

9.3 提升已找到文件的处理速度

9.3.1 问题

你按照 9.1 节中的做法使用 find 命令，但由于找到的文件数量庞大，结果花费了不少时间来处理。你希望能够提高处理速度。

9.3.2 解决方案

参考 9.2 节中有关 xargs 的讨论内容。

9.3.3 参考

- 9.1 节
- 9.2 节

9.4 跟随符号链接查找文件

9.4.1 问题

你用 find 命令查找 .mp3 文件，结果却空空如也，所有符号链接指向的文件都被漏掉了。find 无法跟随符号链接吗？

9.4.2 解决方案

使用 -L 选项 [3]。我们对 9.2 节中的示例加以改动：

[3]也可以换用 -follow。但根据 man find 中的叙述，-follow 的用法已经过时，推荐使用 -L。——译者注

```
find -L . -name '*.mp3' -print0 | xargs -i -0 mv '{}' ~/songs
```

9.4.3 讨论

有时候，你并不想跨界到符号链接所指向的其他文件系统中。因此，默认情况下，find 命令不会跟随符号链接。如果确实需要，可以在 find 之后、目录列表之前加入 -L 选项。

9.4.4 讨论

- man find

9.5 查找文件时不区分大小写

9.5.1 问题

有些 MP3 文件的扩展名是 .MP3，而不是 .mp3。查找时该如何兼顾两者？

9.5.2 解决方案

用 -iname 谓词（如果使用的 find 版本支持）执行不区分大小写的搜索。例如：

```
find . -follow -iname '*.mp3' -print0 | xargs -i -0 mv '{}'
~/songs
```

9.5.3 讨论

有时你在意文件名的大小写，有时则不在意。对于后者，可以使用 -iname 选项，也就是说，在这种情况下，.mp3 和 .MP3 都表明该文件可能是 MP3 文件。（之所以说“可能”，是因为在类 Unix 系统中，你可以随意命名文件，并不强制使用某种扩展名。）

与 Microsoft Windows 兼容的文件系统打交道时，最容易出现大小写问题，尤其是那些陈旧的或“最低标准的”（lowest common denominator）文件系统。数码相机中的文件名类似于 PICT001.JPG，数字随照片递增。如果尝试：

```
find . -name '*.jpg' -print
```

这个命令找不到太多照片。你也可以试试：

```
find . -name '*.[Jj][Pp][Gg]' -print
```

该模式能够匹配方括号中的任一字母，但就是不太好输入，尤其要匹配的模式比较长时。在实践中，-iname 用起来更加简单。要注意的是，并非所有的 find 版本都支持 -iname。如果你所在的系统不支持，那就只能依靠 -name 配合难懂的模式匹配了，或者安装 GNU 版本的 find。

9.5.4 参考

- man find

9.6 按日期查找文件

9.6.1 问题

几个月前，有人给你发了一张 JPEG 图片，你接收后就保存了起来，但现在记不清放哪了。怎样才能找到这张图片呢？

9.6.2 解决方案

使用 find 命令的 -mtime 谓词来检查文件的最后修改日期。例如：

```
find . -name '*.jpg' -mtime +90 -print
```

9.6.3 讨论

-mtime 谓词接受一个参数，用于指定要搜索的时间段。90 代表 90 天。在数字前使用加号（+90）表明要搜索的文件是在 **90 天前**修改的。使用减号（-90）表明文件是在 **90 天以内**修改的。如果既没减号，也没加号，则表明正好就是 90 天。

find 还可以使用逻辑运算符 AND、OR、NOT，如果知道文件修改时间至少在一周（7 天）前，但不超过 14 天，那么就可以像下面这样将两个谓词结合起来。

```
find . -mtime +7 -a -mtime -14 -print
```

还有更复杂的，还可以用 OR 和 AND，甚至是 NOT 来结合多个谓词。

```
find . -mtime +14 -name '*.text' -o \( -mtime -14 -name '*.txt' \)
-print
```

该命令会输出以 .text 结尾、修改时间在 14 天前的文件，或者修改时间在 14 天以内、以 .txt 结尾的那些文件。

为了获得正确的优先级，可能还得使用括号。连续出现的两个谓词，其效果类似于逻辑 AND，优先级高于 OR（在 find 中与在大多数编程语言中一样）。要想消除歧义，多用括号就行了。

括号在 bash 中有着特殊含义，需要对其进行转义，要么写作 \(和 \)，要么放入单引号，写作 '(' 和 ')'。可别将整个表达式都放进单引号，这会把 find 命令搞晕的。它希望每个谓词都是独立的单词。

9.6.4 参考

- man find

9.7 按类型查找文件

9.7.1 问题

你正在查找名称中带有单词“java”的目录。先尝试了以下命令。

```
find . -name '*java*' -print
```

找到的文件太多了，其中还包括文件系统中所有的 Java 源代码文件。

9.7.2 解决方案

使用 -type 谓词只选择目录。

```
find . -type d -name '*java*' -print
```

9.7.3 讨论

我们将 -type d 放在前面，然后是 -name '*java*'。两者的顺序并不影响最终结果，但将 -type d 放在谓词列表的最前面能略微提高搜索效率：对于碰到的每个文件，先测试其是否为目录，如果是，才测试名称是否符合模式。目录的数量比文件要少一些。因此，这种测试顺序使得大部分文件不用再进一步比较名称了。这有什么大不了的吗？随着处理器的速度越来越快，这算不上什么大事。但随着

硬盘容量越来越大，这又变得愈发重要。能测试的文件类型不止目录一种。表 9-1 列出了代表文件类型的单字符。

表9-1：用于**find**命令的**-type**谓词的文件类型字符

字符	含义
b	块设备文件
c	字符设备文件
d	目录
p	管道（或 fifo）
f	普通文件
l	符号链接
s	套接字
D	门（door）（仅限于 Solaris）

9.7.4 参考

- man find

9.8 按大小查找文件

9.8.1 问题

你想清理一下硬盘，为了事半功倍，打算先查找个头最大的文件，决定是否保留。但是该如何找到最大的文件呢？

9.8.2 解决方案

使用 find 命令的 -size 谓词来查找满足大小要求的文件。例如：

```
find . -size +3000k -print
```

9.8.3 讨论

和 -mtime 的数值参数一样，-size 谓词的数值参数前也可以添加减号、加号，或者什么都不写，分别表示小于、大于、等于该数值参数。以上示例要找大于指定大小的文件。

另外，文件大小还包含了单位 k（千字节）。如果单位是 c，则表示字节（或字符）。如果使用 b 作为单位或者不写任何单位，则表示块。（一个块为 512 字节，历史上是 Unix 系统中的通用单位。）这里我们要找的是大于 3MB 的文件。

如果你想删除这些文件，而且所使用的 find 版本也支持，那么 -delete 操作要比使用 rm 或 xargs rm 容易得多。

9.8.4 参考

- man find
- man du
- 9.2.2 节

9.9 按内容查找文件

9.9.1 问题

你之前写了一份重要的信件，并将其保存为以 .txt 为扩展名的文本文件，但现在想不起文件名的其余部分了。除此之外，唯一记得的就

是信件内容中用到过单词“portend”。那么该如何查找已知部分内容的文件呢？

9.9.2 解决方案

如果文件就在当前目录下，可以使用简单的 grep 命令。

```
grep -i portend *.txt
```

使用 -i 选项的话，grep 会忽略大小写。该命令可能尚不足以找到你想要的信件，但可以作为第一步。当然，如果认为信件可能在众多子目录中，可以尝试用下列命令查找当前目录下所有子目录中的所有文本文件。

```
grep -i portend */*.txt
```

说实话，搜索范围仍然不够全面。

如果还没找到，我们换用一个更完备的解决方案：find 命令。使用其 -exec 选项对满足谓词的文件执行命令。你可以按下列方式使用 grep 或其他实用工具。

```
find . -name '*.txt' -exec grep -Hi portend '{}' \;
```

9.9.3 讨论

我们用 -name '*.txt' 帮助减少搜索对象。任何此类测试都会有所帮助，因为要对找到的每个文件单独执行命令会带来时间和 CPU 处理成本。如果对文件日期（如 -mdate -5）有粗略印象，也可以将其加入 find 命令。

执行命令时，文件名会出现在 '{}' 所在的位置。\; 表示命令到此结束，以免还有更多谓词。花括号和分号都需要转义，因此我们分别用了单引号和反斜线。使用哪种转义方式并不重要，只要能避免 bash 误会即可。

在有些系统中，-H 选项会输出包含搜索内容的文件名。一般情况下，使用 grep 时，命令行上只指定一个文件名，因此 grep 也就不用显示该文件名，只输出匹配的文本行即可。因为现在要搜索多个文件，所以得知道对应的文件名。

如果你使用的 grep 版本不支持 -H 选项，只将 /dev/null 作为其中一个文件名在命令行上给出就行了。如果打开多个文件，grep 命令就会输出包含查找文本的文件名。

9.9.4 参考

- man find

9.10 快速查找现有文件及其内容

9.10.1 问题

你想查找文件，但又不想等待冗长的 find 结束，或是需要查找包含特定内容的文件。

9.10.2 解决方案

如果你的系统中有 locate、slocate、Beagle、Spotlight，或者其他索引器，那就万事俱备了。没有的话，那就先来了解一下。

1.5 节中讨论过，locate 和 slocate 会查询系统的数据库文件（通常由 cron 作业负责编译和更新），几乎瞬间就能找到文件或命令。数据库文件的实际位置、会对什么对象进行索引、索引的频率都是视系统而定的。具体细节参见系统的手册页。以下是一个示例：

```
$ locate apropos
/usr/bin/apropos
/usr/share/man/de/man1/apropos.1.gz
/usr/share/man/es/man1/apropos.1.gz
/usr/share/man/it/man1/apropos.1.gz
/usr/share/man/ja/man1/apropos.1.gz
/usr/share/man/man1/apropos.1.gz
```

locate 和 slocate 并不索引文件内容，9.9 节中给出了相关的解决方案。

采用 GUI 的大多数现代操作系统配备了本地搜索工具，这些工具用索引器爬取、解析、索引用户个人空间（Unix 或 Linux 系统的用户主目录）中所有文件的**名称和内容**（通常还包含电子邮件），几乎可以瞬间搜到相关信息。此类工具的可配置性很好，均采用了图形化界面，而且针对单个用户运行。

9.10.3 讨论

slocate 保存了权限信息（以及文件名和路径），因此不会列出用户无权访问的文件。在多数 Linux 系统中，locate 只不过是指向 slocate 的符号链接而已；其他系统可能有单独的程序，或者干脆就没有 slocate。这两者都是命令行工具，或多或少地爬取和索引整个文件系统，但仅输出文件名和位置。

9.10.4 参考

- man locate
- man slocate
- 1.5 节
- 9.9 节

9.11 在可能的位置上查找文件

9.11.1 问题

你需要执行、读入（source）[4] 或读取某个文件，但该文件位于 $PATH 之外的其他位置。

[4] source 本身是 bash shell 的内建命令，能够在当前 shell 的上下文中读取并执行指定文件中的命令。source 一词有时候也作为动词使用，目前尚无较好的统一译法，本书中选择将其译为“读入”。——译者注

9.11.2 解决方案

如果打算读入的某个文件位于 $PATH 所列路径之中，直接读入就行了。如果设置了 shell 选项 sourcepath（默认启用），bash 的内建命令 source（该命令还有另一个 POSIX 名称 .，虽然够简短且易于输入，但确实容易漏看）会搜索 $PATH。

```
source myfile
```

如果想要仅在某个文件位于 $PATH 所列路径之中且可执行时才执行该文件，同时 bash 版本为 2.05b 或更高，可以用 type -P 来搜索 $PATH。和 which 命令不同，type -P 只会在找到文件时才产生输出，这使其更易于在下列情况中使用：

```
LS=$(type -P ls)
[ -x "$LS" ] && $LS

# --或者--

LS=$(type -P ls)
if [ -x "$LS" ]; then
    : commands involving $LS here
fi
```

如果需要查找可能包括 $PATH 在内的多个位置，可以使用 for 循环。要想搜索 $PATH 中的各个部分，使用变量替换运算符 ${*variable*//*pattern*/*replacement*} 将所有的 : 分隔符替换成空格，使之成为一个个独立的单词，然后用 for 迭代单词列表。具体做法如下所示。

```
for path in ${PATH//:/ }; do
    [ -x "$path/ls" ] && $path/ls
done

# --或者--
for path in ${PATH//:/ } /opt/foo/bin /opt/bar/bin; do
    [ -x "$path/ls" ] && $path/ls
done
```

如果文件未在 $PATH 所列出的路径中，而是在其他位置，甚至连名字都可能不一样，那么可以在 for 语句中列出所有的完整路径。

```
for file in /usr/local/bin/inputrc /etc/inputrc ~/.inputrc; do
    [ -f "$file" ] && bind -f "$file" && break  # 使用找到的第一个文件
done
```

根据需要执行其他测试。例如，你可能想在登录时使用 screen（如果系统中有该命令）。

```
for path in ${PATH//:/ }; do
    if [ -x "$path/screen" ]; then
        # 如果screen(1)存在且可执行：
        for file in /opt/bin/settings/run_screen ~/settings/run_screen; do
            [ -x "$file" ] && $file && break # 执行找到的第一个可执行文件
        done
    fi
done
```

有关该代码片段的更多细节，参见 16.22 节。

9.11.3 讨论

用 for 语句迭代所有的可能位置看起来有些杀鸡用牛刀了，但其实这种用法非常灵活，允许你搜索所需要的任何位置、进行任何测试，并对找到的文件执行任何操作。通过将 $PATH 中的每个 : 替换成空格，可以形成 for 语句所要求的以空白字符作为分隔的单词列表（任何以空白字符分隔的列表都管用）。根据需要灵活运用该技术，能够帮你编写出一些对文件位置适应性颇高的 shell 脚本，不仅非常灵活，而且具备可移植性。

也许你想通过设置 $IFS=':' 来直接解析 $PATH，而不是将其拆解到 $path 中。这样做当然可以，但涉及一些额外工作，而且也不够灵活。

你可能也想要像下面这样做：

```
[ -n "$(which myfile)" ] && bind -f $(which myfile)
```

这里的问题不在于文件存在时，而在于文件不存在时。which 命令在不同系统上行为各异。Red Hat 的 which 命令其实是一个别名，设置了各种命令行选项，如果其参数也是别名，则可以提供与该参数相关的细节信息；指定的参数不存在时，则返回消息，表明未查找到（Debian 或 FreeBSD 上的 which 不会这么做）。要是在 NetBSD 上尝试上面这行，你会得到 no myfile in /sbin /usr/sbin /bin /usr/bin /usr/pkg/sbin /usr/pkg/bin /usr/X11R6/ bin /usr/ local/sbin /usr/local/bin，这不会是你想看到的。

这种情况下，command 命令也值得留意。该命令出现的时间早于 type -P，有时也许能派上用场。

Red Hat Enterprise Linux 4.*x* 的行为与以下类似。

```
$ alias which
alias which='alias | /usr/bin/which --tty-only --read-alias --
show-dot --show-tilde'

$ which rd
alias rd='rmdir'
        /bin/rmdir

$ which ls
alias ls='ls --color=auto -F -h'
        /bin/ls

$ which cat
/bin/cat

$ which cattt
/usr/bin/which: no cattt in
(/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/usr/
X11R6/bin:/home/jp/bin)

$ command -v rd
alias rd='rmdir'

$ command -v ls
alias ls='ls --color=auto -F -h'
```

```
$ command -v cat
/bin/cat
```

Debian 和 FreeBSD（但不包括 NetBSD 或 OpenBSD）的行为与以下类似。

```
$ alias which
-bash3: alias: which: not found

$ which rd

$ which ls
/bin/ls

$ which cat
/bin/cat

$ which cattt

$ command -v rd
-bash: command: rd: not found

$ command -v ls
/bin/ls

$ command -v cat
/bin/cat

$ command -v ll
alias ll='ls -l'
```

9.11.4 参考

- help type
- man which
- help source
- man source
- 16.22 节
- 17.4 节

第 10 章　脚本编程的附加特性

很多脚本属于作者自用的一次性脚本，内容也不复杂，寥寥数行而已，可能也就是一个循环。但有些脚本责任重大，会被各色用户频繁使用。此类脚本往往需要利用各种 shell 特性，以便更好地实现代码的共享和重用。这种高级脚本编程技术可以用于很多脚本，在一些较大的系统脚本（如很多 Linux 系统中的 /etc/init.d 脚本）中也会经常看到其身影。即便不是系统管理员，你也能领会并运用本章介绍的技巧和技术。等到你编写大型脚本的时候，就会发现物有所值了。

10.1　脚本“守护进程化”

10.1.1　问题

有时候，你希望脚本能像**守护进程**（daemon）那样在后台永不休止地运行。要想正确地做到这一点，需要将脚本与其控制 TTY（controlling TTY，也就是用于启动守护进程的终端会话）断开。简单的在命令后面放上一个 & 是不够的。如果通过 SSH（或类似的）会话在远程系统启动守护进程脚本，你会发现当注销时，SSH 会话并不会终止，窗口直到脚本结束才会关闭（真正的守护进程可不会这样）。

10.1.2　解决方案

用下列方式调用脚本，将其置于后台运行，同时自己也可以注销。

```
nohup mydaemonscript 0<&-1>/dev/null 2>&1 &
```

或者：

```
nohup mydaemonscript >>/var/log/myadmin.log 2>&1 <&-  &
```

10.1.3 讨论

你需要关闭控制 TTY（终端），它通过 3 种方式与你的（或任何）作业相连：标准输入（STDIN）、标准输出（STDOUT）和标准错误（STDERR）。关闭 STDOUT 和 STDERR 的方法是将其指向其他文件，通常是日志文件，以便随后检索输出信息；或者指向 /dev/null 来丢弃所有输出。我们用重定向运算符 > 来实现。

那 STDIN 呢？最干脆的方法是关闭其文件描述符。实现该操作的 bash 语法和重定向差不多，但用的是连字符作为文件名（0<&- 或 <&-）。

我们用到了 nohup 命令，这样一来，脚本就不会在注销时被挂起（hangup）信号中断了。

第一个示例中的 3 个重定向都明确使用了文件描述符编号（0、1、2）。对于 STDIN 和 STDOUT，这是可选的，因此第二个示例中就不再写明了。另外，相较于起始部分，我们还将输入重定向放在了第二个命令的末尾，因为这里次序并不重要。（但重定向 STDERR 时，次序就变得重要起来了，而且文件描述符编号也要写出。）

10.1.4 参考

- 第 2~3 章，参见有关输入和输出重定向的内容

10.2 代码重用

10.2.1 问题

你想将一组 shell 变量赋值语句共用于所编写的一批脚本。你尝试过将这些配置信息放入单独的脚本，但在其他脚本中运行此脚本时，变量值无法保留下来。这是因为配置脚本运行在另一个 shell 中，当该 shell 退出时，这些变量值也一并消失了。有什么办法能在当前 shell 中运行配置脚本吗？

10.2.2 解决方案

用 bash shell 的 source 命令或 POSIX 的点号（.）读取配置文件的内容。就像在当前脚本中一样处理文件中的各行。

以下是一些配置数据示例。

```
$ cat myprefs.cfg
SCRATCH_DIR=/var/tmp
IMG_FMT=png
SND_FMT=ogg
$
```

这不过就是由 3 个赋值语句组成的简单脚本。以下是另一个用到这些值的脚本。

```
# 使用用户的偏好配置
source $HOME/myprefs.cfg
cd ${SCRATCH_DIR:-/tmp}
echo You prefer $IMG_FMT image files
echo You prefer $SND_FMT sound files
```

10.2.3 讨论

打算使用配置文件的脚本可以通过 source 命令读入该文件。也可以用点号（.）代替 source。在键盘上敲入点号既快又省事，但在脚本或截图中不太容易注意到这个字符。

```
. $HOME/myprefs.cfg
```

放心，你绝对不是第一个漏看了点号，觉得直接就能运行该脚本的那个人。

读入（sourcing）是 bash 脚本编程中一种既强大又危险的特性。有了它，就可以创建配置文件并将其共享给多个脚本。利用这种机制，修改配置文件时，只用改动一次就够了。

配置文件的内容并不局限于简单的变量赋值。只要是有效的 shell 命令都可以，因为读入配置文件时，无非就是从其他地方获取输入，命

令还是由 bash shell 处理的。不管被读入的文件包含什么样的 shell 命令（例如，循环或调用其他命令），全都属于合法的 shell 输入，将作为脚本的一部分运行。

下面是一个修改过的配置文件。

```
$ cat myprefs.cfg
SCRATCH_DIR=/var/tmp
IMG_FMT=$(cat $HOME/myimage.pref)
if [ -e /media/mp3 ]
then
        SND_FMT=mp3
else
        SND_FMT=ogg
fi
echo config file loaded
$
```

该文件中并非想象那样充斥着各种配置变量。它还可以执行其他命令（如 cat）、使用 if 语句进行条件判断，甚至还能输出消息。读入文件时一定要当心，它就像一扇敞开的大门，直通你的脚本。

最好的办法是定义 bash 函数（参见 10.4 节）。这些函数能够以通用函数库的形式在所有读入了函数定义的脚本之间共享。

10.2.4 参考

- bash 手册页，参见有关 readline 的更多内容
- 10.3 节
- 10.4 节

10.3 在脚本中使用配置文件

10.3.1 问题

你想在脚本中使用外部配置文件。

10.3.2 解决方案

你可以编写大量代码来解析某些特殊的配置文件格式。不过，拜托了，别这么干。将配置文件制作成 shell 脚本，然后使用 10.2 节中的方法就行了。

10.3.3 讨论

这只不过是 source 命令的一种特定应用而已。值得一提的是，你可能得花点心思，想想如何将所有的配置项转换成合法的 bash 语句。特别是，你可以利用布尔标志和可选变量（参见第 5 章和 15.11 节）。

```
# 在配置文件中
VERBOSE=0                          # 0或者''表示关闭，1表示启用
SSH_USER='jbagadonutz@'            # 注意结尾的@，如果将该配置项的值设置为''，
                                   # 则表示使用当前用户

# 在脚本中
[ "$VERBOSE" ] || echo "Verbose msg from $0 goes to STDERR" >&2
[...]
ssh $SSH_USER$REMOTE_HOST [...]
```

当然了，依靠用户正确书写配置文件有一定风险，因此，不要指望用户阅读注释并在结尾处添加 @，我们还是自己在脚本中完成吧。

```
# 如果已设置$SSH_USER且结尾处没有@，那么将其添加上：
[ -n "$SSH_USER" -a "$SSH_USER" = "${SSH_USER%@}" ] &&
SSH_USER="$SSH_USER@"
```

或者只用：

```
ssh ${SSH_USER:+${SSH_USER}@}${REMOTE_HOST} [...]
```

两种写法的效果相同。bash 变量运算符 :+ 的操作如下：如果 $SSH_USER 不为空，则返回 :+ 右侧的值（这里指定的是该变量自身以及额外的 @）；否则，什么都不返回。

10.3.4 参考

- 第 5 章
- 10.2 节
- 15.11 节

10.4 定义函数

10.4.1 问题

你想在 shell 脚本的几处地方为用户显示用法信息（描述正确的命令语法），但又不愿意重复同样的 echo 语句。有办法只写一次代码，然后多次引用吗？如果可以将用法信息作为单独的脚本，那就可以在原始脚本的任意位置调用了，但这样的话，本来一个脚本就能搞定的事，现在则需要两个。而且，将一个脚本的用法信息作为另一个脚本的输出看起来怪怪的。有没有更好的解决方法？

10.4.2 解决方案

你需要的是 bash 函数。在脚本的起始部分加入下列内容：

```
function usage ()
{
    printf "usage: %s [ -a | - b ] file1 ... filen\n" ${0##*/} >
&2
}
```

随后，可以在脚本中按以下方式写代码：

```
if [ $# -lt 1]
then
    usage
fi
```

10.4.3 讨论

定义函数的方法不止一种（[function] *name* [()] { *compound-command* } [*redirections*]），以下方法都可以。

```
function usage ()
{
    printf "usage: %s [ -a | - b ] file1 ... filen\n" ${0##*/} >
&2
}

function usage {
    printf "usage: %s [ -a | - b ] file1 ... filen\n" ${0##*/} >
&2
}

usage ()
{
    printf "usage: %s [ -a | - b ] file1 ... filen\n" ${0##*/} >
&2
}
```

保留字 function 或结尾的 () 必须出现。如果使用了 function，那么 () 就是可选的。我们喜欢使用 function，因为既清晰，可读性又好，还便于 grep 搜索。例如，grep^'function' *script* 可以列出 script 文件中的所有函数。

函数定义应该放在 shell 脚本的开头，或者至少位于函数调用之前。从某种意义上来说，函数定义就是另一种 bash 语句。但只要得以执行，就定义了函数。如果在定义之前调用函数，则会产生“command not found”错误。这就是为什么我们要始终将函数定义写在脚本中的其他命令之前。

例子中的函数基本上没干什么事，就只有一个 printf 语句而已。但由于我们将用法信息放在了函数中，以后如果增加了新选项，就不用修改散落在脚本各处的各个语句，只修改这个函数就够了。

除了格式字符串之外，printf 唯一的参数就是 $0，其中包含的是所调用的 shell 脚本的完整名称，经过 ## 运算符修改后，可以得到路径名的最后一部分。其效果类似于使用 $(basename $0)。

由于用法信息属于错误信息，我们将 printf 语句的输出重定向到了标准错误。也可以将重定向放在函数定义之外，如此一来，函数的所有输出都会被重定向。如果有多个输出语句，这种做法很方便。

```
function usage ()
{
    printf "usage: %s [ -a | - b ] file1 ... filen\n" ${0##*/}
    printf "example: %s -b *.jpg \n" ${0##*/}
    printf "or else: %s -a myfile.txt yourfile.txt \n" ${0##*/}
} > &2
```

10.4.4 参考

- 5.20 节
- 7.1 节
- 16.15 节
- 16.16 节
- 19.14 节

10.5 使用函数：参数和返回值

10.5.1 问题

你想使用函数，同时需要将一些值传入该函数。怎么传递参数呢？又该如何将值返回？

10.5.2 解决方案

你不用像在某些编程语言中那样在参数两边加上括号。直接将参数放在函数名右边，彼此之间用空白字符分隔，就像调用 shell 脚本或命令那样。如果有必要，记得给参数加上引号！

```
# 定义函数：
function max ()
{ ... }
#
# 调用函数：
#
max   128   $SIM
max  $VAR  $CNT
```

从函数返回值的方法有两种。一种方法是向函数内部的变量赋值，如例 10-1 所示。这些变量都是全局变量，除非函数内明确使用 local 声明。

例 10-1 ch10/func_max.1

```
# 实例文件：func_max.1

# 定义函数：
function max ()
{
    local HIDN
    if [ $1 -gt $2 ]
    then
        BIGR=$1
    else
        BIGR=$2
    fi
    HIDN=5
}
```

例如：

```
# 调用函数：
max 128 $SIM
# 使用结果：
echo $BIGR
```

另一种方法是使用 echo 或 printf 将输出发送到标准输出，如例 10-2 所示。

例 10-2 ch10/func_max.2

```
# 实例文件：func_max.2

# 定义函数：
function max ()
{
    if [ $1 -gt $2 ]
    then
        echo $1
    else
        echo $2
```

```
    fi
}
```

然后，你必须在 $() 内调用函数，获取其输出并使用，否则就白白显示在屏幕上了。例如：

```
# 调用函数：
BIGR=$(max 128 $SIM)
# 使用结果：
echo $BIGR
```

10.5.3 讨论

在调用函数时放置参数就和调用 shell 脚本一样。参数无非就是另一种命令行单词罢了。

与命令行参数一样，在函数中引用参数也是使用 $1、$2 等。但是，$0 保持不变，其中包含的还是所调用脚本的名称。一旦从函数中返回，$1、$2 等就恢复原状，仍旧引用脚本参数。

$FUNCNAME 数组也值得关注。$FUNCNAME 本身引用的是该数组的第 0 个元素，其中包含的是当前执行函数的名称。换句话说，$FUNCNAME 对于函数来说就像 $0 对于脚本一样，只是前者不包含路径信息。数组的其余元素形成了一个调用栈（call stack），最底部或者最后那个元素 [1] 就是“main”。$FUNCNAME 仅存在于函数执行期间。

[1]索引值最高的元素。——译者注

我们创建 $HIDN 这个没什么用的变量只是为了表明它是函数的局部变量。虽然可以在函数内为其赋值，但该值无法在脚本的其他地方使用。函数的局部变量仅在函数被调用时存在，函数返回后就会被销毁。

通过设置变量的方式从函数返回值更为高效，而且可以处理大量数据（设置多个变量），但这种方法也有缺点。它要求函数和脚本其余部分就传递信息的变量名达成一致。这种耦合性存在维护方面的问题。别的方法（通过输出返回值）确实可以降低耦合，但同时也限制了其

实用性：由于脚本不得不花很多精力来解析函数的结果，因而限制了返回的数据量。那该采用哪种方法呢？和大量的工程实践一样，这同样需要根据具体需求进行权衡。

10.5.4 参考

- 1.8 节
- 16.5 节

10.6 中断陷阱

10.6.1 问题

你正在编写一个能够捕获信号并做出相应处理的脚本。

10.6.2 解决方案

使用实用工具 `trap` 来设置信号处理程序。首先，用 `trap -l`（或者 `kill -l`）列出你想捕获的信号。输出结果因系统而异。

```
# NetBSD
$ trap -l
 1) SIGHUP    2)   SIGINT      3) SIGQUIT    4) SIGILL
 5) SIGTRAP   6)   SIGABRT     7) SIGEMT     8) SIGFPE
 9) SIGKILL   10)  SIGBUS     11) SIGSEGV   12) SIGSYS
 13) SIGPIPE  14)  SIGALRM    15) SIGTERM   16) SIGURG
 17) SIGSTOP  18)  SIGTSTP    19) SIGCONT   20) SIGCHLD
 21) SIGTTIN  22)  SIGTTOU    23) SIGIO     24) SIGXCPU
 25) SIGXFSZ  26)  SIGVTALRM  27) SIGPROF   28) SIGWINCH
 29) SIGINFO  30)  SIGUSR1    31) SIGUSR2   32) SIGPWR
$
# Linux（为了适合页面排版，对输出做了折行处理）
$ trap -l
 1) SIGHUP        2) SIGINT        3) SIGQUIT       4) SIGILL
 5) SIGTRAP       6) SIGABRT       7) SIGBUS        8) SIGFPE
 9) SIGKILL      10) SIGUSR1      11) SIGSEGV      12) SIGUSR2
13) SIGPIPE      14) SIGALRM      15) SIGTERM      16) SIGSTKFLT
17) SIGCHLD      18) SIGCONT      19) SIGSTOP      20) SIGTSTP
21) SIGTTIN      22) SIGTTOU      23) SIGURG       24) SIGXCPU
25) SIGXFSZ      26) SIGVTALRM    27) SIGPROF      28) SIGWINCH
```

```
29) SIGIO        30) SIGPWR       31) SIGSYS       34) SIGRTMIN
35) SIGRTMIN+1   36) SIGRTMIN+2   37) SIGRTMIN+3   38) SIGRTMIN+4
39) SIGRTMIN+5   40) SIGRTMIN+6   41) SIGRTMIN+7   42) SIGRTMIN+8
43) SIGRTMIN+9   44) SIGRTMIN+10  45) SIGRTMIN+11  46) SIGRTMIN+12
47) SIGRTMIN+13  48) SIGRTMIN+14  49) SIGRTMIN+15  50) SIGRTMAX-14
51) SIGRTMAX-13  52) SIGRTMAX-12  53) SIGRTMAX-11  54) SIGRTMAX-10
55) SIGRTMAX-9   56) SIGRTMAX-8   57) SIGRTMAX-7   58) SIGRTMAX-6
59) SIGRTMAX-5   60) SIGRTMAX-4   61) SIGRTMAX-3   62) SIGRTMAX-2
63) SIGRTMAX-1   64) SIGRTMAX
$
```

接下来，设置要捕获的信号和信号处理程序。注意，如果命令被信号 *signal number* 所终止，则脚本的退出状态码为 128+*signal number*。下面是一个简单的例子，其中只关心是否捕获到信号，并不在意是什么信号。如果使用 `trap '' ABRT EXIT HUP INT QUIT TERM`，脚本将很难被“杀死”，因为这些信号都会被忽略。

```
$ cat hard_to_kill
#!/bin/bash
trap ' echo "You got me! $?" ' ABRT EXIT HUP INT QUIT TERM
trap ' echo "Later... $?"; exit ' USR1
sleep 120

$ ./hard_to_kill
^CYou got me! 130
You got me! 130

$ ./hard_to_kill &
[1] 26354

$ kill -USR1 %1
User defined signal 1
Later... 158
You got me! 0
[1]+ Done                       ./hard_to_kill

$ ./hard_to_kill &
[1] 28180

$ kill %1
You got me! 0
[1]+ Terminated                 ./hard_to_kill
```

例 10-3 更有意思。

例 10-3 ch10/hard_to_kill

```
#!/usr/bin/env bash
# 实例文件：hard_to_kill

function trapped {
    if [ "$1" = "USR1" ]; then
        echo "Got me with a $1 trap!"
        exit
    else
        echo "Received $1 trap--neener, neener"
    fi
}

trap "trapped ABRT" ABRT
trap "trapped EXIT" EXIT
trap "trapped HUP"  HUP
trap "trapped INT"  INT
trap "trapped KILL" KILL    # 没有效果
trap "trapped QUIT" QUIT
trap "trapped TERM" TERM
trap "trapped USR1" USR1    # 特殊情况

# 空循环，什么都不做，不引入"第三方"捕获行为，
# 效果就像使用了sleep一样
while (( 1 )); do
    :  # :是一个空命令
done
```

这里调用该脚本，然后尝试将其“杀死”：

```
$ ./hard_to_kill
^CReceived INT trap--neener, neener
^CReceived INT trap--neener, neener
^CReceived INT trap--neener, neener
^Z
[1]+ Stopped                 ./hard_to_kill

$ kill -TERM %1

[1]+ Stopped                 ./hard_to_kill
Received TERM trap--neener, neener

$ jobs
[1]+ Stopped                 ./hard_to_kill

$ bg
[1]+ ./hard_to_kill &

$ jobs
```

```
[1]+ Running                  ./hard_to_kill &

$ kill -TERM %1
Received TERM trap--neener, neener

$ kill -HUP %1
Received HUP trap--neener, neener

$ kill -USR1 %1
Got me with a USR1 trap!
Received EXIT trap--neener, neener

[1]+ Done                     ./hard_to_kill
```

10.6.3 讨论

首先得说明一下，其实你是无法捕获 -SIGKILL（-9）的。该信号可以立刻“杀死”进程，压根没机会捕获。因此，也许我们的例子并不是真的难以被“杀死”。但要记住，该信号使得脚本或程序无法随时完成清理工作或者有条不紊地终止。这往往不是什么好事，所以除非没有选择，否则尽力避免使用 `kill -KILL`。

`trap` 的用法如下所示。

```
trap [-lp] [arg] [signal [signal]]
```

第一个非选项参数是接收到信号时要执行的代码。如先前的示例所示，代码可以是独立的（self-contained），也可以调用函数。对于大多数一般应用，调用一个或多个错误处理函数可能是最好的方法，这种形式可以很好地完成清理工作，然后有条不紊地终止。如果该参数为空串，则忽略指定的信号。如果该参数为 - 或不指定，那么后续列出的一个或多个信号会被重置为默认值。`-l` 选项会列出信号名称，如 10.6.2 节所示，而 `-p` 选项会打印出当前捕获的信号及其处理程序。

如果使用了多个信号处理程序，我们建议你花点时间按照字母顺序排列信号名称，因为这样更易于阅读和后续查找。

之前提到过，如果命令是被信号 *signal number* 终止的，则脚本的退出状态码为 128+*signal number*。

有 3 个用于特殊目的的伪信号[2]。DEBUG 信号类似于 EXIT，但它是在每个命令执行前产生的，可作为调试之用。从函数中返回或用 source（.）运行完脚本后，RETURN 信号会被触发。如果命令执行失败，ERR 信号会被触发。更多具体细节和注意事项，请查阅 Bash Reference Manual，尤其是如何用 declare 或 set -o functrace 选项处理函数。

[2]之所以称为“伪信号”，是因为这 3 个信号是由 shell 产生的，而其他信号是由操作系统产生的。——译者注

一些 POSIX 差异会对 trap 产生影响。如 Bash Reference Manual 中所述：“使用命令行选项 --posix 启动 bash 或者在 bash 运行时执行 'set -o posix'，会使得 bash 在默认行为存在差异的地方做出改变，以符合 POSIX 的规定，从而更严格地遵循 POSIX 标准。”尤其是，这会导致 kill 和 trap 不显示信号名称前的 SIG，同时 kill -l 的输出也会有所不同。另外，trap 在处理参数时会更严格，重置信号时会要求在信号名称前加上 -。这也就是说，需要写成 trap -USR1，而不是 trap USR1。不管有没有必要，我们都建议你加上 -，这样能更清晰地表达意图。

10.6.4 参考

- help trap
- 1.19 节
- 10.1 节
- 14.11 节
- 17.7 节

10.7 用别名重新定义命令

10.7.1 问题

你想略微更改一下命令的定义，这样就可以总是使用该命令的特定选项（例如，ls 命令的 -a 选项或 rm 命令的 -i 选项）。

10.7.2 解决方案

（仅）对于交互式 shell，可以使用 bash 的**别名**特性。alias 命令足够聪明，即使写成下面这样，也不会陷入死循环。

```
alias ls='ls -a'
```

事实上，如果只输入 alias，不使用其他参数，你可以看到当前 bash 会话中已经定义好的别名列表。有些发行版可能已经预先设置好了一些别名。

10.7.3 讨论

别名机制就是一种直截了当的文本替换。它发生在命令行处理的早期阶段，因此在别名之后还会出现其他替换操作。例如，要想将字母“h”定义为可列出用户主目录内容的命令，可以使用以下方式：

```
alias h='ls $HOME'
```

或者：

```
alias h='ls ~'
```

第一种写法中的单引号至关重要，这意味着定义别名时并不会对变量 $HOME 求值。只有执行命令时才会进行（字符串）替换并对变量 $HOME 求值。因此可以说，如果更改了 $HOME 值，别名也会随之变动。

如果改用双引号，变量会被立刻求值，定义别名时使用的就是 $HOME 的值。输入不带参数的 alias 就可以从所有的别名定义中看到类似于下列的结果。

```
...
alias h='ls /home/youracct'
...
```

如果对别名不满意，不想再用，使用 unalias 以及想删除的别名即可。例如：

```
\unalias h
```

如果场面难以收拾，可以用 unalias -a 删除当前 shell 会话中的所有别名定义。为什么要在上面的命令前添加反斜线前缀？反斜线能够禁止任何命令的别名扩展，这是标准的最佳安全实践，目的就是避免不安好心的人给 unalias 也设置别名（可能会设置成“:”），从而令其丧失作用。

```
$ alias unalias=':'

$ alias unalias
alias unalias=':'

$ unalias unalias

$ alias unalias
alias unalias=':'

$ \unalias unalias

$ alias unalias
bash: alias: unalias: not found
```

定义别名时不允许使用参数。例如，你不能写成：

```
alias='mkdir $1 && cd $1'
```

$1 和 $HOME 的区别在于：$HOME 是在定义别名时指定的，而你希望 $1 在运行时传入。不好意思，这无法实现。改用函数吧。

10.7.4 参考

- 附录 C，参见有关命令行处理的更多内容
- 10.4 节
- 10.5 节
- 14.4 节
- 16.16 节

10.8 避开别名和函数

10.8.1 问题

你创建的别名或函数覆盖了真正的命令，但现在想执行的是后者。

10.8.2 解决方案

使用 bash shell 的 builtin 命令来忽略 shell 函数和别名，执行实际的内建命令。

使用 command 命令来忽略 shell 函数和别名，并执行实际的外部命令。

如果只是想避开别名扩展，但仍允许执行函数，在命令前加上 \ 前缀即可。

使用 type 命令（以及 -a 选项）来了解 shell 如何解释指定的名称。

我们来看几个示例。

```
$ alias echo='echo ~~~'

$ echo test
~~~ test

$ \echo test
test

$ builtin echo test
test

$ type echo
echo is aliased to `echo ~~~'

$ unalias echo

$ type echo
echo is a shell builtin
```

```
$ type -a echo
echo is a shell builtin
echo is /bin/echo

$ echo test
test
```

我们来接着讨论这个函数定义。

```
function cd ()
{
    if [[ $1 = "..." ]]
    then
        builtin cd ../..
    else
        builtin cd "$1"
    fi
}
```

10.8.3 讨论

alias 命令足够聪明，不会在碰到类似于 alias ls='ls-a' 或 alias echo='echo ~~~' 这种写法时陷入死循环，因此，第一个例子不需要对别名定义的右侧部分做什么特殊处理就可以引用实际的 echo 命令。

如果已经将 echo 定义为别名，type 命令不仅会告诉我们它是别名，还会显示该别名的定义。与此类似，如果是函数定义，则显示实际的函数代码。type -a *some_command*（只要不同时使用 -p）[3] 会显示所有包含 *some_command* 的地方（别名、内建命令、函数、外部命令）。

[3]如果同时使用 -p 选项与 -a 选项，type 不会输出别名和函数。——译者注

最后一个例子用函数覆盖了内建命令 cd，以便添加一项简单的便捷功能。我们希望函数能够理解 cd ... 代表向上退回两级目录，也就是 cd ../..（参见 16.15 节）。其他所有参数仍按照正常方式对待。该函数只是简单地尝试匹配 ...，然后将其替换成实际含义。但在函数内部（或外部），如何调用实际的 cd 命令来切换目录呢？builtin 命令告诉 bash 将其后的命令视为 shell 的内建命令，不

使用任何别名或函数。我们在函数中用到了该命令，但任何时候都可以用它来明明白白地引用实际命令，避开同名的函数。

如果与函数同名的是可执行文件（如 ls），而非内建命令，那么你可以改用可执行文件的完整路径（如 /bin/ls）来避开别名以及函数定义。如果不知道完整路径，那就在命令前加上关键字 command，这样 bash 也会忽略同名的别名和函数定义，使用实际命令。但要注意，查找命令时仍会用到 $PATH 变量。如果因为 $PATH 设置有误而导致无法执行 ls，在这种情况下，使用 command 也无济于事。

10.8.4 参考

- help builtin
- help command
- help type
- 14.4 节
- 16.15 节

10.9 计算已过去的时间

10.9.1 问题

你想要显示脚本或脚本中某个操作花费了多长时间。

10.9.2 解决方案

使用内建命令 time 或 bash 变量 $SECONDS。

10.9.3 讨论

time 以多种方式报告某个进程或管道所花费的时间。

```
$ time sleep 4

real    0m4.029s
```

```
user    0m0.000s
sys     0m0.000s

$ time sha256sum /bin/* &> /dev/null

real    0m1.252s
user    0m0.072s
sys     0m0.028s
```

你可以在脚本中对各个命令或函数计时，但无法对整个脚本计时。显然可以在调用脚本时或者对 cron 作业使用 time，但要注意，将其添加到 cron 会一直产生输出，因而总是会收到 cron 发来的邮件。

如果这种方法看起来有些大材小用，或者你只是想知道整个脚本用了多长时间，那么不妨使用变量 $SECONDS。根据 Bash Reference Manual 的说法：

> [$SECONDS] 会扩展成 shell 启动以来的时长（秒数）。如果为其赋值，那么随后在扩展该变量时，得到的值就是先前的赋值与赋值操作以后的时长（秒数）之和。

例如：

```
$ cat seconds
started="$SECONDS"
sleep 4
echo "Run-time = $(($SECONDS - $started)) seconds..."

$ bash seconds
Run-time = 4 seconds...

$ time bash seconds
Run-time = 4 seconds...

real    0m4.003s
user    0m0.000s
sys     0m0.000s
```

10.9.4 参考

- help time

- 与 bash 版本对应的 Bash Reference Manual

10.10 编写包装器

10.10.1 问题

你手头有一系列相关的命令或工具，经常临阵拿来使用，你想将它们汇集到一处，以便使用和记忆。

10.10.2 解决方案

根据需要，用 `case..esac` 编写一个 shell 脚本“包装器”。

10.10.3 讨论

基本的处理方法有两种。一种是编写大量的微型 shell 脚本或别名来应对所有需求。BusyBox 就是这么做的，其中的大量工具只是指向单个二进制文件的符号链接。另一种方法和大多数版本控制工具一样，调用类似于“前缀”（prefix）的单个二进制文件，然后添加操作或命令。两种方法各有千秋，不过我们更倾向于后一种，因为你只需记得一个前缀命令就够了。

网站 Signal v. Noise 对此曾有过精彩的讨论和更为复杂的实现，值得一读。我们的实现要更简单些，用到了一些技巧，其中部分设计考虑如下：

- 易于阅读和理解
- 易于扩展
- 易于编写内建在线帮助
- 易于使用和记忆

本书是用 Asciidoc 编写的，需要记忆大量标记。以下代码（例 10-4）取自我们自己编写的实用工具。该工具能够从命令行获取输入，也可以读写 Linux 的剪切板。

例 10-4 ch10/ad

```
#!/usr/bin/env bash
# 实例文件：ad
# 用于Asciidoc的O'Reilly“图书”工具包装器

# 普通的完备性检查
❶
[ -n "$BOOK_ASC" ] || {
    echo "FATAL: must export \$BOOK_ASC to the location of
'...bcb2/head/asciidoc/'!"
    exit 1
}
\cd "$BOOK_ASC" || {
    echo "FATAL: can't cd to '$BOOK_ASC'!"
    exit 2
}

SELF="$0"    # 为了在递归中更加清晰
❷

action="$1" # 提高代码的可读性
❸
shift       # 删除参数
❹

# 如果xsel是可执行文件且没有更多的参数                                    ❺
[ -x /usr/bin/xsel -a $# -lt 1 ] && {
    # 读/写剪贴板
    text=$(xsel -b)
    function Output {
        echo -en "$*" | xsel -bi
    }
} || { # 否则……
    # 读/写STDIN/STDOUT
    text=$*
    function Output {
        echo -en "$*"
    }
}

case "$action" in


#########################################################################
#####
    # 内容/标记

    rec|recipe ) # 为新实例创建标签                                      ❻
```

```
        id="$($SELF id $text)" # 创建一个ID                    ❼
        Output "$(cat <<- EoF                                  ❽
                        [[$id]]
                        === $text

                        [[problem-$id]]
                        ==== Problem

                        [[solution-$id]]
                        ==== Solution

                        [[discussion-$id]]
                        ==== Discussion

                        [[see_also-$id]]
                        ==== See Also
                        * \`man \`
                        * item1
                        * <<xref-id-here>>
                        * URL[text]
                        EoF
            )"
    ;;

    table )                 # 为新表格创建标签                  ❾
        Output "$(cat <<- EoF
                        .A Table
                        [options="header"]
                        |=======
                        |head|h|h
                        |cell|c|c
                        |cell|c|c
                        |=======
                        EoF
            )"
    ;;

# ...
    ### 标题
    h1 )                    # 在每章的一级标题内（Asciidoc h3） ❿
        Output "=== $text"
    ;;
    h2 )                    # 在每章的二级标题内（Asciidoc h4）
        Output "==== $text"
    ;;
    h3 )                    # 在每章的三级标题内（Asciidoc h5）
        Output "===== $text"
    ;;
```

```
    ### 列表
    bul|bullet )          # 无序列表（.. = 2级，+ = 多行元素）
        Output ". $text"
    ;;
    nul|number|order* )   # 有序列表（## = 2级，+ = 多行元素）
        Output "# $text"
    ;;
    term )                # 术语
        Output "term_here::\n  $text"
⓫
    ;;
    # ...

    cleanup )             ## 清理所有xHTML/XML/PDF碎片
        rm -fv {ch??,app?}.{pdf,xml,html} book.xml docbook-xsl.css
⓬
    ;;

    * )
        \cd - > /dev/null # 返回到前一个目录
⓭
            # 另外参见Stack OverFlow网站上的如下问题及其答案：How to get
the
            # source directory of a Bash script from within the
script itself?
            # can-a-bash-script-tell-what-directory-its-stored-in
        ( echo "Usage:"
        egrep '\)[[:space:]]+# '   $SELF
        echo ''
        egrep '\)[[:space:]]+## '  $SELF ) | more
⓮
    ;;

esac
```

❶ 对要求的变量和位置进行完备性检查。

❷ 为递归设置可读性更好的名称。

❸ 为接下来要执行的命令或操作设置可读性更好的名称。

❹ 删除参数，以后不再重复使用或将其包含在输入或输出中。

❺ 如果 xsel 命令可用且可执行，同时我们也没有传递任何参数，则可以将输入和输出设置为剪切板。该脚本摇身变成了应用程序通用宏工具。无论使用什么编辑器，如果你拥有 GUI 并能读写剪贴板，则切换到终端会话时，就可以轻而易举地对文本进行复制、处理、粘贴，实在是方便极了！

❻ case..esac 中的每个语句块都包含了代码和文档，其中的分支代码部分可以采用任何有意义的顺序，但是帮助 / 用法信息可能因此有所不同。

❼ 获得输入文本并发起递归调用以得到 ID，然后输出样板标记。

❽ 注意，在 here-document 内部必须使用制表符缩进。

❾ 有时样板标记不包括任何输入文本。

❿ 有时候操作非常简单，记住需要多少个等号就行了。

⓫有时候操作就有些复杂了，还有嵌入的换行符和扩展的转义字符。

⓬可以执行你能想到的任何操作，弄清楚如何实现自动化。

⓭如果没有提供任何参数，或者提供的参数不正确，甚至包括 -h 或 --help 等，你都会看到用法信息。

⓮我们将一段代码放入子 shell ()，按照正确的顺序获得输出并将其全部传给 more 命令。两个 egrep 命令负责显示 case..esac 中与分支判断相关的行，如❻中所述，这些行包含了代码和文档，按照 # 字符的数量（1 个或 2 个）进行分组。

Mac 中用 pbcopy 和 pbpaste 代替 xsel。

用法示例：

```
$ ad
Usage:
    rec|recipe )          # 为新实例创建标签
    table )               # 为新表格创建标签
```

```
    h1 )                    # 在每章的一级标题内（Asciidoc h3）
    h2 )                    # 在每章的二级标题内（Asciidoc h4）
    h3 )                    # 在每章的三级标题内（Asciidoc h5）
    bul|bullet )            # 无序列表（.. = 2级，+ = 多行元素）
    nul|number|order* )     # 有序列表（## = 2级，+ = 多行元素）
    term )                  # 术语

    cleanup )               ## 清理所有xHTML/XML/PDF碎片
$
```

要想用 ad 为一则新实例创建标签，你需要输入标题，将其选中，打开或切换到终端窗口，输入 ad rec，再切换回编辑器，然后粘贴。说起来麻烦，但做起来又快又简单。这种脚本的优点在于“包治百病”，通常很容易扩展，也就用法提示需要自己来写。我们已经按照这种模式将脚本应用于以下方面：

- 编写本书
- 包装各种 SSH 命令来负责多组服务器的日常杂务工作
- 先于 apt 收集各种 Debian 软件包系统工具
- 自动执行各种“清理”任务，如修剪空白字符、排序，以及执行各种简单的文本操作（例如，剥离富文本格式）
- 自动化 grep 命令以搜索各种特定的文件类型和位置，获取备注和归档文档

10.10.4 参考

- Signal v. Noise（“Automating with convention: Introducing sub”一文）
- 附录 D
- 15.17 节
- Atlas Docs（“Writing in AsciiDoc”一文）

第 11 章　处理日期和时间

感觉处理日期和时间不是什么难事，但事实并非如此。不管你是编写 shell 脚本还是大型程序，记录时间（timekeeping）都是一件麻烦事：时间和日期的不同显示格式、夏令时、闰年、闰秒等。例如，假设有一份合同清单以及签订合同的日期，而你想计算所有这些合同的到期日期。这可不是件简单事：是否会受到闰年影响？夏令时会不会给合同带来问题？如何格式化输出以消除歧义？7/4/07 是指 2007 年 7 月 4 日，还是 2007 年 4 月 7 日？

日期和时间渗透在计算机领域的方方面面。你迟早会碰上它们：在系统、应用程序或事务日志中，在数据处理脚本中，在用户或管理任务中，等等。本章将帮助你尽可能干净利索地处理日期和时间。计算机特别擅长准确地记录时间，特别是当使用网络时间协议（Network time protocol，NTP）保持本机时间与国家和国际时间同步的时候。在理解不同地区之间的夏令时差异方面，计算机的表现也不错。要想在 shell 脚本中处理时间，得用到 Unix 的 date 命令（能使用该命令的 GNU 版本就更好了，这是 Linux 系统的标配）。date 能够以不同的格式显示日期，甚至还能正确地进行日期运算。

注意，gawk（awk 的 GNU 版本）的 strftime 格式化功能和 GNU date 命令一样。除了一个简单的例子，本章不打算讲述 gawk 的用法。建议你坚持使用 GNU 版本的 date 命令，不仅更易用，而且还有一个非常实用的 -d 选项。但也别忘记 gawk，以防碰到一个没有 GNU date、只有 gawk 的系统。

11.1　格式化日期显示

11.1.1　问题

你需要格式化日期或时间，方便输出。

11.1.2　解决方案

使用包含 strftime 格式规格的 date 命令。要想了解所支持的格式规格，参见附录 A.13 或者 strftime 手册页。

```
# 在脚本中设置环境变量会有所帮助：
$ STRICT_ISO_8601='%Y-%m-%dT%H:%M:%S%z' # 严格的ISO 8601格式
$ ISO_8601='%Y-%m-%d %H:%M:%S %Z'       # 和ISO 8601差不多，但可读性
更好
$ ISO_8601_1='%Y-%m-%d %T %Z'           # %T等同于%H:%M:%S
$ DATEFILE='%Y%m%d%H%M%S'               # 适合在文件名中使用

$ date "+$ISO_8601"
2006-05-08 14:36:51 CDT

$ gawk "BEGIN {print strftime(\"$ISO_8601\")}"
2006-12-07 04:38:54 EST

# 和先前的$ISO_8601一样
$ date '+%Y-%m-%d %H:%M:%S %Z'
2006-05-08 14:36:51 CDT

$ date -d '2005-11-06' "+$ISO_8601"
2005-11-06 00:00:00 CST

$ date "+Program starting at: $ISO_8601"
Program starting at: 2006-05-08 14:36:51 CDT

$ printf "%b" "Program starting at: $(date '+$ISO_8601')\n"
Program starting at: $ISO_8601

$ echo "I can rename a file like this: mv file.log file_$(date
+$DATEFILE).log"
I can rename a file like this: mv file.log file_20060508143724.log
```

11.1.3 讨论

你也许倾向于将 + 添加到环境变量中，以此简化后续命令，但有些系统的 date 命令比较挑剔 + 的存在及其位置。我们的建议是明确将它写进 date 命令。

可用的格式化选项还有不少，完整的清单可参见系统的 date 手册页或 C 语言的 strftime() 函数。

除非指定，否则假定时区为系统所定义的当地时间。GNU date 命令使用的 %z 格式是一种非标准扩展，未必能在你的系统上使用。

ISO 8601 是显示日期和时间的推荐标准，应该尽一切可能使用。相较于其他显示格式，它具备多种优势：

- 公认的标准
- 不存在歧义
- 既易于阅读，又便于程序解析（例如，使用 awk 或 cut）
- 用于成列数据或文件名时，其排序结果符合预期

尽量避免 MM/DD/YY 或 DD/MM/YY（甚至是更糟糕的 M/D/YY 或 D/M/YY）这种格式。其排序结果不尽如人意，而且存在歧义，因为根据地理位置，日或月都可能是第一部分，这也令其难于解析。同样，尽量使用 24 小时制，避免出现更多的歧义和解析问题。

11.1.4 参考

- man date
- man 3 strftime
- 维基百科（ISO 8601）
- iso.org（“ISO 8601 Date and Time Format”一文）
- A.13 节

11.2 提供默认日期

11.2.1 问题

你希望脚本能够提供一个有用的默认日期，最好还能提示用户核对。

11.2.2 解决方案

使用 GNU date 命令，将最有可能的日期分配给变量，然后允许用户更改（参见例 11-1）。

例 11-1 ch11/default_date

```
#!/usr/bin/env bash
# 实例文件：default_date

# 使用中午时间来避免脚本在午夜前后运行，如果哪个表慢了几秒，就会出现"差了一天"
的
错误 [1]
START_DATE=$(date -d 'last week Monday 12:00:00' '+%Y-%m-%d')

while [ 1 ]; do
    printf "%b" "The starting date is $START_DATE, is that
correct? (Y/new date)"
    read answer

    # 除了直接按回车、输入"Y"或"y"之外，其他内容均作为新日期来验证
    # 可以使用"[Yy]*"来允许用户输入"yes"……
    # 按照CCYY-MM-DD的格式验证新日期
    case "$answer" in
        [Yy]) break
            ;;
        [0-9][0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9])
            printf "%b" "Overriding $START_DATE with $answer\n"
            START_DATE="$answer"
            ;;

        *)   printf "%b" "Invalid date, please try again...\n"
            ;;
    esac
done

END_DATE=$(date -d "$START_DATE +7 days" '+%Y-%m-%d')

echo "START_DATE: $START_DATE"
echo "END_DATE:   $END_DATE"
```

[1] 如果是中午的话，差几秒还是同一天；要是在午夜时分差几秒，日期就要隔天了。——译者注

11.2.3 讨论

并非所有的 date 命令都支持 -d 选项，但 GNU 版本支持。我们的建议是尽一切可能获得并使用 GNU date 命令。

如果脚本是以无人值守形式或定时（如通过 cron）运行，可以省略用户验证代码。

有关如何格式化日期和时间，参见 11.1 节。

我们在脚本中使用类似的代码生成 SQL 查询。脚本在指定时间运行，针对特定日期范围创建 SQL 查询，以生成报告。

11.2.4 参考

- man date
- 11.1 节
- 11.3 节

11.3 自动生成日期范围

11.3.1 问题

你已经有了一个日期（可能来自 11.2 节），希望再自动产生另一个日期。

11.3.2 解决方案

GNU date 命令的功能非常强大，用法灵活，但 -d 选项的威力并不为人们所熟知。你的系统可能会将其记录在 getdate 名下（尝试查看 getdate 手册页）。以下是几个示例。

```
$ date '+%Y-%m-%d %H:%M:%S %z'
2005-11-05 01:03:00 -0500

$ date -d 'today' '+%Y-%m-%d %H:%M:%S %z'
2005-11-05 01:04:39 -0500

$ date -d 'yesterday' '+%Y-%m-%d %H:%M:%S %z'
2005-11-04 01:04:48 -0500

$ date -d 'tomorrow' '+%Y-%m-%d %H:%M:%S %z'
2005-11-06 01:04:55 -0500
```

```
$ date -d 'Monday' '+%Y-%m-%d %H:%M:%S %z'
2005-11-07 00:00:00 -0500

$ date -d 'this Monday' '+%Y-%m-%d %H:%M:%S %z'
2005-11-07 00:00:00 -0500

$ date -d 'last Monday' '+%Y-%m-%d %H:%M:%S %z'
2005-10-31 00:00:00 -0500

$ date -d 'next Monday' '+%Y-%m-%d %H:%M:%S %z'
2005-11-07 00:00:00 -0500

$ date -d 'last week' '+%Y-%m-%d %H:%M:%S %z'
2005-10-29 01:05:24 -0400

$ date -d 'next week' '+%Y-%m-%d %H:%M:%S %z'
2005-11-12 01:05:29 -0500

$ date -d '2 weeks' '+%Y-%m-%d %H:%M:%S %z'
2005-11-19 01:05:42 -0500

$ date -d '-2 weeks' '+%Y-%m-%d %H:%M:%S %z'
2005-10-22 01:05:47 -0400

$ date -d '2 weeks ago' '+%Y-%m-%d %H:%M:%S %z'
2005-10-22 01:06:00 -0400

$ date -d '+4 days' '+%Y-%m-%d %H:%M:%S %z'
2005-11-09 01:06:23 -0500

$ date -d '-6 days' '+%Y-%m-%d %H:%M:%S %z'
2005-10-30 01:06:30 -0400

$ date -d '2000-01-01 +12 days' '+%Y-%m-%d %H:%M:%S %z'
2000-01-13 00:00:00 -0500

$ date -d '3 months 1 day' '+%Y-%m-%d %H:%M:%S %z'
2006-02-06 01:03:00 -0500
```

11.3.3 讨论

-d 选项允许你指定特定的日期，不再使用字符串“now”，但并非所有的 date 命令都支持此选项。GNU 版本没有问题，我们建议你尽一切可能获取并使用该版本。

-d 选项不容易掌握。下列参数都能按照预期工作。

```
$ date '+%a %Y-%m-%d'
Sat 2005-11-05

$ date -d 'today' '+%a %Y-%m-%d'
Sat 2005-11-05

$ date -d 'Saturday' '+%a %Y-%m-%d'
Sat 2005-11-05

$ date -d 'last Saturday' '+%a %Y-%m-%d'
Sat 2005-10-29

$ date -d 'this Saturday' '+%a %Y-%m-%d'
Sat 2005-11-05
```

如果你是在周六执行该命令，想看看下周六的日期，但其实得到的还是今天的日期。[2]

[2]但是在译者使用的 Linxu 发行版 CentOS 7（GNU date 版本号 8.22）中并没有出现这种情况。——译者注

```
$ date -d 'next Saturday' '+%a %Y-%m-%d'
Sat 2005-11-05
$
```

另外还要注意 `this week` *`day`*，因为只要指定的那天已经过去，**本周**（this week）就会变成 **下周**（next week）。因此，如果在 2005 年 11 月 5 日周六执行下列命令，你得到的就是以下结果，这可能和你预想的并不一样。

```
$ date -d 'this week Friday' '+%a %Y-%m-%d'
Fri 2005-11-11
```

-d 选项非常有用，但一定要全面测试你的代码，并提供相应的错误检查。

如果你手边没有 GNU date 命令可用，*Unix Review* 杂志 2005 年 9 月刊的“Shell Corner: Date-Related Shell Functions”一文中给出的下列 shell 函数也许能派上用场。

pn_month

给定月份的前 x 个月和后 x 个月

end_month

给定月份的月末

pn_day

给定日期的前 x 天和后 x 天

cur_weekday

给定日期是星期几

pn_weekday

给定日期的前 x 天和后 x 天是星期几

在较新的 bash 版本中，下列函数也是可用的。

pn_day_nr

（非递归）给定日期的前 x 天和后 x 天

days_between

两个日期之间隔了几天

注意，pn_month、end_month、cur_weekday 独立于其余的函数。但是，pn_day 是在 pn_month 和 end_month 之上构建的，pn_weekday 是在 pn_day 和 cur_weekday 之上构建的。

11.3.4 参考

- man date
- man getdate
- Dr.Dobb's（“Shell Corner: Date-Related Shell Functions”一文）

- 11.2 节

11.4 将日期和时间转换为纪元秒

11.4.1 问题

你想将日期和时间转换为纪元秒，以便执行日期和时间运算。

11.4.2 解决方案

使用 GNU date 命令，配合非标准选项 -d 和标准格式 %s。

```
#"当前时间"很容易得到
$ date '+%s'
1131172934

# 其他时间需要用到非标准选项-d
$ date -d '2005-11-05 12:00:00 +0000' '+%s'
1131192000
```

纪元秒就是从纪元（1970 年 1 月 1 日午夜，也就是 1970-01-01T00:00:00）开始算起的秒数。该命令从纪元开始，加上纪元秒数，显示出你想得到的日期和时间。

11.4.3 讨论

如果你手边没有 GNU date 命令可用，这个问题就麻烦了。我们建议你尽一切可能获取并使用该版本。实在不行，也可以使用 Perl。以下给出了 3 种正确输出当前纪元秒的方法。

```
$ perl -e 'print time, qq(\n);'
1154158997

# 同上
$ perl -e 'use Time::Local; print timelocal(localtime()) .
qq(\n);'
1154158997

$ perl -e 'use POSIX qw(strftime); print strftime("%s",
```

```
localtime()) . qq(\n);'
1154159097
```

鉴于 Perl 所采用的日期 / 时间结构，用 Perl 来转换特定日期和时间要比转换“当下”（right now）更难。年份从 1900 年开始，月份（不是天数）从 0 开始（不是从 1）。函数格式为：`timelocal(`*`sec`*`, `*`min`*`, `*`hour`*`, `*`day`*`, `*`month-1`*`, `*`year-1900`*`)`。因此，要想将 2005-11-05 06:59:49 转换为纪元秒：

```
# 给定时间为本地时间
$ perl -e 'use Time::Local;
> print timelocal("49", "59", "06", "05", "10", "105") . qq(\n);'
1131191989

# 给定时间为UTC时间
$ perl -e 'use Time::Local;
> print timegm("49", "59", "06", "05", "10", "105") . qq(\n);'
1131173989
```

11.4.4 讨论

- man date
- 11.5 节
- A.13 节

11.5 将纪元秒转换为日期和时间

11.5.1 问题

你需要将纪元秒转换为人们能读懂的日期和时间。

11.5.2 解决方案

使用 GNU date 命令，配合需要的格式（参见 11.1 节）。

```
$ EPOCH='1131173989'

$ date -d "1970-01-01 UTC $EPOCH seconds" +"%Y-%m-%d %T %z"
```

```
2005-11-05 01:59:49 -0500

$ date --utc --date "1970-01-01 $EPOCH seconds" +"%Y-%m-%d %T %z"
2005-11-05 06:59:49 +0000
```

11.5.3 讨论

如果你手边没有 GNU date 命令可用，可以尝试下列 Perl 单行脚本。

```
$ EPOCH='1131173989'

$ perl -e "print scalar(gmtime($EPOCH)), qq(\n);" # UTC
Sat Nov 5 06:59:49 2005

$ perl -e "print scalar(localtime($EPOCH)), qq(\n);" # Your local
time
Sat Nov 5 01:59:49 2005

$ perl -e "use POSIX qw(strftime);
> print strftime('%Y-%m-%d %H:%M:%S',localtime($EPOCH)), qq(\n);"
2005-11-05 01:59:49
```

11.5.4 参考

- man date
- 11.1 节
- 11.4 节
- A.13 节

11.6 用Perl获得昨天或明天的日期

11.6.1 问题

你想要获得昨天或明天的日期，但系统中只有 Perl，没有 GNU date 命令。

11.6.2 解决方案

使用下列 Perl 单行脚本，调整与 time 相加或相减的秒数。

```
# 昨天的这个时候（注意是减法）
$ perl -e "use POSIX qw(strftime);
> print strftime('%Y-%m-%d', localtime(time - 86400)), qq(\n);"
2005-11-04

# 明天的这个时候（注意是加法）
$ perl -e "use POSIX qw(strftime);
> print strftime('%Y-%m-%d', localtime(time + 86400)), qq(\n);"
2005-11-06
```

11.6.3 讨论

这其实是上一个实例的特定应用，但因为经常会用到，所以值得一谈。11.7 节中有一个非常方便的表格，没准能用得上。

11.6.4 参考

- 11.2 节
- 11.3 节
- 11.4 节
- 11.5 节
- 11.7 节
- A.13 节

11.7 日期与时间运算

11.7.1 问题

你需要对日期和时间进行某些运算。

11.7.2 解决方案

如果你自己算不出结果，那就需要使用 date 命令（参见 11.3 节）将已有的日期和时间转换成纪元秒（参见 11.4 节），执行相关运

算，然后将得出的纪元秒转换成你需要的格式（参见 11.5 节）。

如果手边没有 GNU date 命令可用，*Unix Review* 杂志 2005 年 9 月刊的“Shell Corner: Date-Related Shell Functions”一文中给出的 shell 函数也许能派上用场。参见 11.3 节。

举例来说，假设你从某台机器上得到了一批日志数据，该机器的时钟慢了不少。大家现在应该都已经使用了网络时间协议，这种事情不大会发生，这里只是假设而已。

```
CORRECTION='172800'     # 两天的秒数

# 提取数据中的日期部分并保存到$bad_date
# 假设是这样：
bad_date='Jan 2 05:13:05' # syslog格式的日期

# 使用GNU date将其转换成纪元秒
bad_epoch=$(date -d "$bad_date" '+%s')

# 进行纠正
good_epoch=$(( bad_epoch + $CORRECTION ))

# 使用GNU date将纠正后的日期转换成易于阅读的格式
good_date=$(date -d "1970-01-01 UTC $good_epoch seconds")
good_date_iso=$(date -d "1970-01-01 UTC $good_epoch seconds" +'%Y-
%m-%d %T')
Date

echo "bad_date:         $bad_date"
echo "bad_epoch:        $bad_epoch"
echo "Correction:       +$CORRECTION"
echo "good_epoch:       $good_epoch"
echo "good_date:        $good_date"
echo "good_date_iso:    $good_date_iso"

# 将$good_date插回日志数据
```

小心年份！为了易于阅读，有些 Unix 命令（如 ls 和 syslog）在某些情况下会省略年份。在计算纠正量时，可能得考虑到这一点。如果数据的日期跨度比较大或时区不同，需要想办法将其分成单独的文件，并分别进行处理。

11.7.3 讨论

在处理各种日期运算时，相较于我们知道的其他格式，使用纪元秒要容易得多。你不必担心小时、天、周或年。进行一些简单的加法或减法就行了。使用纪元秒也可以避免所有关于闰年和闰秒的复杂规则，如果对一个时区进行标准化（通常是 UTC，以前称为 GMT），甚至可以避开各种时区。

表 11-1 列出了各种可能会用到的值。

表11-1：常用纪元时间值转换表

秒	分钟	小时	天
60	1		
300	5		
600	10		
3600	60	1	
18 000	300	5	
36 000	600	10	
86 400	1440	24	1
172 800	2880	48	2
604 800	10 080	168	7
1 209 600	20 160	336	14

秒	分钟	小时	天
2 592 000	43 200	720	30
31 536 000	525 600	8760	365

11.7.4 参考

- 11.3 节
- 11.4 节
- 11.5 节
- 13.13 节

11.8 处理时区、夏令时和闰年

11.8.1 问题

你需要考虑时区、夏令时、闰年或闰秒问题。

11.8.2 解决方案

没有解决方案。

11.8.3 讨论

这比听起来要难得多。还是将这项任务留给久经考验的代码吧，选择能满足要求的现成工具就行了。本章中使用 GNU date 的其他实例很可能已经能满足你的要求。如果还不行，还有其他工具能搞定，这一点基本可以肯定。例如，许多出色的 Perl 模块可以处理日期和时间。

说实话，我们不是在开玩笑。想正确地处理时区、夏令时、闰年或闰秒真的是一场噩梦。少给自己添堵，用现成的工具吧。

11.8.4 参考

- 11.1 节
- 11.3 节
- 11.4 节
- 11.5 节
- 11.7 节

11.9 用date和cron在第*N*天运行脚本

11.9.1 问题

你想在每月的第 *N* 天（如第 2 个周三）运行脚本，但大多数 cron 无法实现。

11.9.2 解决方案

在要执行的命令中加点 shell 代码。在 Linux 的 Vixie-cron crontab 中，选用以下任意一行。如果用的是其他 cron 程序，可能得根据 cron 使用的调度方式将星期几的名称转换成数字（0-6 或 1-7），并用 +%w（数字形式的星期几）代替 +%a（所在语言环境下的缩写形式的星期几）。

```
# Vixie-cron
# Min Hour DoM Mnth DoW Program
# 0-59 0-23 1-31 1-12 0-7

# Vixie-cron要求转义%，否则会报错!

# 在第1个周三的23:00运行
00 23 1-7 * Wed [ "$(date '+\%a')" == "Wed" ] && /path/to/command
args to command

# 在第2个周四的23:00运行
```

```
00 23 8-14 * Thu [ "$(date '+\%a')" == "Thu" ] && /path/to/command

# 在第3个周五的23:00运行
00 23 15-21 * Fri [ "$(date '+\%a')" == "Fri" ] &&
/path/to/command

# 在第4个周六的23:00运行
00 23 22-27 * Sat [ "$(date '+\%a')" == "Sat" ] &&
/path/to/command

# 在第5个周日的23:00运行
00 23 28-31 * Sun [ "$(date '+\%a')" == "Sun" ] &&
/path/to/command
```

注意，一周中的某一天未必总会在一个月内出现 5 次，因此，如果计划在每月的第 5 周执行某项任务，请务必确保清楚自己的需求。

也要注意另一件事，Vixie-cron 要求转义 %，否则就会得到类似“Syntax error: EOF in backquote substitution”的错误。其他版本的 cron 对此可能不做要求，最好还是检查一下相关手册页。

如果 cron 看起来好像不正常，尝试重启你的 MTA[3]（如 sendmail）。有些系统上的某些 cron 版本（如 Red Hat 上的 Vixie-cron）是和 sendmail 进程捆绑在一起的。

[3]Mail Transfer Agent，即邮件传输代理。——译者注

11.9.3 讨论

大多数版本的 cron（包括 Linux 的 Vixie-cron）不允许你在每个月的第 *N* 天安排作业。为了解决这个问题，我们将作业安排在第 *N* 天所在的**日期范围内**，然后检查日期是否正确，以此决定是否执行命令。“每月的第 2 个周三”肯定出现在这个月的 8~14 号，我们只需要在此期间每天都运行作业并检查当天是否为周三。如果是，则执行指定的命令。

表 11-2 给出了解决方案中提到的日期范围。

表11-2：每月各周的日期范围

周	日期范围
第一周	1 号至 7 号
第二周	8 号至 14 号
第三周	15 号至 21 号
第四周	22 号至 27 号
第五周（注意先前的提醒）	28 号至 31 号

我们知道这看起来也太过简单了。要是不信的话，你可以查看日历。

```
$ cal 10 2006
   October 2006
 S  M  Tu   W  Th   F   S
 1  2   3   4   5   6   7
 8  9  10  11  12  13  14
15 16  17  18  19  20  21
22 23  24  25  26  27  28
29 30  31
$
```

11.9.4 参考

- man 5 crontab
- man cal

11.10 输出带有日期的日志

11.10.1 问题

你希望输出日志或其他行时能加上日期，但同时又想避免 date 命令所带来的开销。

11.10.2 解决方案

对于 bash 4 或以上版本，可以使用 printf '%(fmt)T' 来输出日期和时间。

```
printf '%(%F %T)T; Foo Bar\n' '-1'
```

你也可以用 printf 为变量赋值，这样就可以轻松地重复使用结果了。

```
printf -v today '%(%F)T' '-1'    # 设置$today = '2014-11-15'
```

11.10.3 讨论

参数 '-1' 很重要，但并不一致！bash 手册页中是这么说的：

> 有两个特殊的参数值可以使用：-1 代表当前时间，-2 代表 shell 被调用的时间。

但是这种默认行为在 bash 4.2 和 4.3 期间发生了变化。在 4.2 版本中，空参数被视为空（null），然后返回 Unix 纪元的当地时间，这肯定不是你想要的。4.3 版本中的一处特例使得空参数被视为 '-1'。例如：

```
$ echo $BASH_VERSION
4.2.37(1)-release

$ printf '%(%F %T %Z)T; Foo Bar\n'
1969-12-31 19:00:00 EST; Foo Bar

$ printf '%(%F %T %Z)T; Foo Bar\n' '-1'
2014-11-15 15:24:26 EST; Foo Bar

$ echo $BASH_VERSION
```

```
4.3.11(1)-release

$ printf '%(%F %T %Z)T; Foo Bar\n'
2014-11-15 15:25:02 EST; Foo Bar

$ printf '%(%F %T %Z)T; Foo Bar\n' '-1'
2014-11-15 15:25:05 EST; Foo Bar
```

bash 中的 printf 是内建命令，但还有一个名为 printf 的独立二进制可执行文件，两者并不是一回事。独立的可执行文件适用于那些没有内建 printf 的 shell。因此，可别把 printf 的手册页与 bash 手册页中的 printf 部分搞混了。虽然两者之间有很大的相似性，但后者才是你需要的。

11.10.4 参考

- man date
- man strftime
- 15.15 节
- 15.17 节
- 17.18 节

第 12 章　帮助最终用户完成任务的shell脚本

目前为止，你已经见识了大量的小脚本。就规模和适用范围来说，我们的示例一直都是比较小的。现在，我们打算向你展示几个比较大（尽管不是很大）的例子。除系统管理任务外，旨在提供一些 shell 脚本的真实案例。我们期待你发现它们的价值。除此之外，通过阅读这些脚本，我们希望你能够从中学到一些东西，或者动手实践，甚至是修改代码，拿来自己使用。

12.1　输出连字符

12.1.1　问题

你想要有一个能够输出一行连字符的简单脚本。

12.1.2　解决方案

用简单的命令输出一行连字符听起来很容易，事实上也确实如此。但只要你觉得自己有了一个简单的脚本，事情就开始起变化了。要不要让连字符行的长度多样化？将连字符改成用户提供的字符怎么样？有没有见识过**特性蔓延**有多容易出现？能不能写一个简单的脚本，将这些扩展功能都考虑在内，同时又避免变得过于复杂？

思考例 12-1 中的脚本。

例 12-1　ch12/dash

```
#!/usr/bin/env bash
# 实例文件：dash
# dash - 输出一行连字符
# 参数：字符数量（默认72个）
# -c X用字符X代替连字符
```

```
#
function usagexit ( )
{
    printf "usage: %s [-c X] [#]\n" ${0##*/}  ❶
    exit 2
} >&2

LEN=72                                        ❷
CHAR='-'
while (( $# > 0 ))                            ❸
do
    case $1 in
    [0-9]*) LEN=$1;;                          ❹
    -c) shift                                 ❺
        CHAR=$1;;
    *) usagexit;;                             ❻
    esac
    shift
done

if (( LEN > 4096 ))                           ❼
then
    echo "too large" >&2
    exit 3
fi

# 构建指定长度的字符串
DASHES=""
for ((i=0; i<LEN; i++))
do
    DASHES="${DASHES}${CHAR}"
done
printf "%s\n" "$DASHES"
```

12.1.3 讨论

构建由所需数量的连字符（或其他字符）组成的字符串，然后将其输出到标准输出（STDOUT），基本任务就算完成了。这只用了最后 6 行代码。默认值在脚本的起始部分，在 `while` 循环之前就设置好了。其他行负责处理参数解析、错误检查、用户消息和注释。

你会发现，在一个稳健的最终用户脚本中，这是相当典型的场面。不到 20% 的代码干完了大部分“真正的”工作，而剩下 80% 的代码都在为脚本锦上添花，实现用户友好。

❶ 在显示此脚本的名称时，我们用带有模式（* /）的字符串操作运算符修剪掉所有的路径名前导字符。不管用户如何调用该脚本（例如 ./dashes，/home/username/bin/dashes，甚至是 ../../over/there/dashes），用法信息中仍以 dashs 出现。

❷ 用两个赋值语句设置默认值。

❸ 在这里解析参数。每处理一个参数，shift 就会减少参数数量，最终导致跳出 while 循环。

❹ 只有两个允许的参数：一个是字符长度。

❺ 另一个是 -c 选项，之后紧随着用于代替连字符的字符。

❻ 其他任何选项都在此处理：输出用法信息，提前退出脚本。

❼ 最后，注意脚本在这里强加了一个最大长度的限制，不过这完全是随意的。你会保留还是取消这种限制？

解析 -c 及其参数时得多加小心。由于没有用到更为复杂的解析技术（如使用 getopts；参见 13.1 节），我们的代码要求选项及其参数之间由空白字符分隔。（例如，在运行该脚本时，用户必须输入 -c 25，而不能是 -c25。）我们甚至压根没有检查到底有没有提供选项参数。而且，用户输入的可能不是单个字符，而是一个字符串。（能不能想一个简单的方法来限制这个问题，只提取参数的首字符如何？你需要或者想要这么做吗？为什么非得让用户指定单个字符，而不能是字符串呢？）

数值参数的解析也得用到一些比较复杂的技术。case 语句中的模式遵循的是**路径扩展**规则，而不是正则表达式。你可能会认为 case 模式 [0-9]* 表示“仅数字”，但这其实是正则表达式的含义。在 case 语句中，它表示以数字起始的任意字符串。如果没有捕捉到像 9.5 或 612more 这种错误输入，随后会导致脚本出错。这里用 if 语句配合 =~ 及其复杂的正则表达式匹配功能，应该能发挥作用。

从这个例子可以看出，简单的脚本也会牵扯到方方面面，大多数是错误检查、参数解析等问题。要是脚本只是写来自己用，这些事通常都可以含糊过去，或者干脆完全不理会。毕竟，你就是该脚本唯一的用户，很清楚适合的用法，也没打算乱用，如果脚本出错，显示错误信息时也用不着考虑是否美观。但如果要把脚本分享给别人使用，那就另当别论了，你可能得花费大量的精力来改进脚本。

12.1.4 参考

- 5.8 节
- 5.11 节
- 5.12 节
- 5.20 节
- 6.15 节
- 13.1 节

12.2 浏览相册

12.2.1 问题

你把刚从数码相机中下载的照片全放到了一个目录中，希望能方便地浏览这些照片，挑出那些好看的。

12.2.2 解决方案

编写一个能够生成一系列 HTML 页面的 shell 脚本，这样就可以在浏览器中查看照片了。我们给这个脚本起名为 mkalbum 并将其放入 ~/bin 目录中。

在命令行上，使用 cd 进入想要创建相册的目录（通常就是保存照片的地方），然后执行命令（如 ls *.jpg，也可以参见 9.5 节），生成要包含在相册内的照片列表，再将输出通过管道传给例 12-2 中的 shell 脚本 mkalbum（随后我们会解释）。你需要将要创建的相册名称（也就是脚本要创建的目录名）放在命令行上，以作为该脚本唯一的参数。看起来如下所示。

```
ls *.jpg | mkalbum rugbymatch
```

图 12-1 展示了生成的 Web 页面。

图 12-1：mkalbum 生成的 Web 页面

大标题是照片名（文件名），另外还有指向相册中第一张、最后一张、下一张和上一张照片的超链接。

例 12-2 中的 shell 脚本（mkalbum）会为相册生成一组 HTML 页面，一张照片对应一个页面。

例 12-2 ch12/mkalbum

```
#!/usr/bin/env bash                                          ❶
# 实例文件：mkalbum
# mkalbum - 制作HTML相册
# 版本号0.2
#
# 相册就是包含HTML页面的目录
# 在当前目录下创建
#
# 相册页就是显示某张照片的HTML页面，
# 其中包含标题（也就是照片的文件名），
# 另外还有指向第一张、前一张、下一张和最后一张照片的超链接
#
# ERROUT
ERROUT()                                                     ❷
{
    printf "%b" "$@"
} >&2                                                        ❸

#
# USAGE
USAGE()                                                      ❹
{
    ERROUT "usage: %s <newdir>\n" ${0##*/}                   ❺
}

# EMIT(thisph, startph, prevph, nextph, lastph)
EMIT()                                                       ❻
```

```
{
    THISPH="../$1"
    STRTPH="${2%.*}.html"
    PREVPH="${3%.*}.html"
    NEXTPH="${4%.*}.html"
    LASTPH="${5%.*}.html"
    if [ -z "$3" ]
    then
        PREVLINE='<TD> Prev </TD>'
    else
        PREVLINE='<TD> <A HREF="'$PREVPH'"> Prev </A> </TD>'
    fi
    if [ -z "$4" ]
    then
        NEXTLINE='<TD> Next </TD>'
    else
        NEXTLINE='<TD> <A HREF="'$NEXTPH'"> Next </A> </TD>'
    fi
cat <<EOF                                                    ❼
<HTML>
<HEAD><TITLE>$THISPH</TITLE></HEAD>
<BODY>
  <H2>$THISPH</H2>
<TABLE WIDTH="25%">
  <TR>
  <TD> <A HREF="$STRTPH"> First </A> </TD>
  $PREVLINE
  $NEXTLINE
  <TD> <A HREF="$LASTPH"> Last </A> </TD>
  </TR>
</TABLE>
  <IMG SRC="$THISPH" alt="$THISPH"
   BORDER="1" VSPACE="4" HSPACE="4"
   WIDTH="800" HEIGHT="600"/>
</BODY>
</HTML>
EOF
}

if (( $# != 1 ))
then
    USAGE
    exit -1
fi
ALBUM="$1"
if [ -d "${ALBUM}" ]
then
    ERROUT "Directory [%s] already exists.\n" ${ALBUM}
    USAGE
    exit -2
```

```
else
    mkdir "$ALBUM"
fi
cd "$ALBUM"

PREV=""
FIRST=""
LAST="last"

while read PHOTO
do
    # 开始制作相册
    if [ -z "${CURRENT}" ]
    then
        CURRENT="$PHOTO"
        FIRST="$PHOTO"
        continue
    fi

    PHILE=${CURRENT##*/}     # 删除路径的前导部分
    EMIT "$CURRENT" "$FIRST" "$PREV" "$PHOTO" "$LAST" >
"${PHILE%.*}.html"

    # 准备下一次迭代
    PREV="$CURRENT"
    CURRENT="$PHOTO"

done

PHILE=${CURRENT##*/}    # 删除路径的前导部分
EMIT "$CURRENT" "$FIRST" "$PREV" "" "$LAST" > "${PHILE%.*}.html"

# 生成"last"的符号链接
ln -s "${PHILE%.*}.html" ./last.html            ❽
# 生成index.html的符号链接
ln -s "${FIRST%.*}.html" ./index.html
```

12.2.3 讨论

虽然有大量免费的照片查看器，但用 bash 构建简单的相册有助于展示 shell 编程的强大威力，同时也给我们提供了一份值得讨论的干货。

❶ 最开始是一行特殊的注释，定义了用哪个可执行文件运行该脚本。接下来几行注释描述了脚本。再次强调，一定要对脚本做注释。想要

回想脚本的用途时，最松散的注释也能帮助你唤起 3 天或 13 个月前的记忆。

❷ 注释之后是函数定义。ERROUT 函数和 printf 基本上差不多（因为它就是调用 printf 而已），但是绕了个弯，前者将输出重定向到了标准错误。这就省得你还得记住重定向每个 printf 的错误消息。

❸ 通常将重定向放在命令末尾，这里放到了函数定义的结尾处，意在告诉 bash 重定向该函数产生的所有输出。

❹ USAGE 函数很方便地告知了用户如何调用脚本，不过也不是非得写成单独的函数。我们并没有将脚本名称硬编码在用法信息中，而是使用了特殊变量 $0，以免脚本以后改名。$0 是所调用脚本的名称，其中包括了用户指定的路径名。

❺ 用 ## 运算符去掉与路径无关的部分（由 */ 指定）。

❻ EMIT 函数的代码比较多。该函数用于生成相册各个页面的 HTML。相册的每页都有对应的（静态）Web 页面，其中包含指向前一张、下一张、第一张和最后一张照片的超链接。EMIT 函数可不知道那么多；它会获得要链接到的所有图片名并将其转换成页面名，除了扩展名变成 .html，两者的名称相同。例如，如果 $2 包含文件名 pict001.jpg，则 ${2%.*}.html 的结果为 pict001.html。

❼ 因为要生成的 HTML 代码太多，我们不再一个个使用 printf 语句，而是改用 cat 命令和 here-document，这样就可以在脚本中照原样（literal）逐行输入 HTML，其中出现的 shell 变量仍会被替换。cat 命令只是简单将 STDIN 复制（拼接）到 STDOUT。在脚本中，我们重定向了 STDIN，令其从后续文本行（也就是 here-document）中获取输入。不引用输入终止词（end-of-input word，写作 EOF，而非 'EOF' 或 \EOF），可以确保 bash 会继续对输入行中的变量进行替换，这样就能够根据函数的参数为各种标题和超链接使用变量。

❽ 脚本中的最后两个命令创建了符号链接，分别作为第一张和最后一张照片的快捷方式。这样一来，脚本就无须知道相册第一页和最后一页的具体名称；生成其他相册页时，分别使用硬编码名称 index.html 和 last.html 就行了。作为结束，由于最后处理的文件名就是相册中的最后一张照片，为其创建符号链接。与此类似，对于第一页（尽管早就知道这页的名称了），我们一直等到最后才一并创建符号链接，这纯粹是风格问题：将相似的操作放在一起。

有关该脚本设计的最后一点思考：我们可以传给 EMIT 函数一个文件名，让 EMIT 将自己的输出重定向到该文件，但这种重定向在逻辑上并不是 EMIT 该干的活儿（ERROUT 函数的任务才是重定向）。EMIT 的目的是创建 HTML，而发送 HTML 是另一回事。因为 bash 允许我们轻松地重定向输出，所以将其作为单独的步骤是可行的。此外，这种方法仅将其输出写入 STDOUT，调试起来会更容易。

12.2.4 参考

- w3schools
- Chuck Musciano 和 Bill Kennedy 合著的 *HTML & XHTML: The Definitive Guide, 6th Edition*（O'Reilly 出版）
- 3.2 节
- 3.3 节
- 3.4 节
- 5.13 节
- 5.14 节
- 5.18 节
- 5.23 节
- 9.5 节
- 16.11 节

12.3 填装MP3播放器

12.3.1 问题

你收集了一堆 MP3 文件，想将它们放进 MP3 播放器，但是播放器的存储空间不够用。你不想瞪着眼睛，一首接一首地往 MP3 中填装音乐，一直等到填满为止。有没有什么解决方法？

12.3.2 解决方案

可以使用例 12-3 中给出的 shell 脚本，一边向 MP3 播放器中复制文件，一边监视可用的存储空间，只要装满就退出。

例 12-3 ch12/load_mp3

```
#!/usr/bin/env bash
# 实例文件：load_mp3
# 尽可能多地向MP3播放器中填装歌曲
# 注意：这里假设MP3播放器挂载在/media/mp3上
#

#
# 确定文件大小
#
function FILESIZE ()
{
    FN=${1:-/dev/null}
    if [[ -e $FN ]]
    then
        # FZ=$(stat -c '%b' "$FN")
        set -- $(ls -s "$FN")
        FZ=$1
    fi
}

#
# 计算MP3播放器的可用存储空间
#
function FREESPACE
{
    # FREE=$(df /media/mp3 | awk '/^\/dev/ {print $4}')
    set -- $(df /media/mp3 | grep '^/dev/')
    FREE=$4
}

# 从（全局）可用存储空间中减去（给定）文件大小
function REDUCE ()
(( FREE-=${1:-0}))     # 管用，但不常见
```

```
#
# 主体部分
#
let SUM=0                                                    ❶
let COUNT=0
export FZ
export FREE
FREESPACE                                                    ❷
find . -name '*.mp3' -print | \                              ❸
( while read PATHNM                                          ❹
    do
        FILESIZE "$PATHNM"
        if ((FZ <= FREE))
        then
            echo loading $PATHNM
            cp "$PATHNM" /media/mp3
            if (( $? == 0 ))
            then
                let SUM+=FZ
                let COUNT++
                REDUCE $FZ
            else
                echo "bad copy of $PATHNM to /media/mp3"
                rm -f /media/mp3/"${PATHNM##*/}"
                # 重新计算，因为我们也不知道现在有多少可用的存储空间
                FREESPACE
            fi
            # 还要继续吗?
            if (( FREE <= 0 ))
            then
                break
            fi
        else
            echo skipping $PATHNM
        fi
    done
    printf "loaded %d songs (%d blocks)" $COUNT $SUM
    printf " onto /media/mp3 (%d blocks free)\n" $FREE
)
# 脚本结束
```

12.3.3 讨论

该脚本会将当前目录下（一直到目录树的叶子节点）的所有 MP3 文件复制到挂载在 /media/mp3 上的 MP3 播放器（或其他设备）中。开始复制前，脚本会尝试确定设备的可用存储空间，然后从中减去要复制

文件的大小，以便知道何时该退出（设备已满或者复制完所有文件）。

调用脚本的方法很简单。

```
load_mp3
```

然后就可以等着它（你也可以去接杯咖啡）复制文件，其运行时间取决于磁盘的读取速度和 MP3 内存的写入速度。

我们来看看脚本中用到的一些 bash 特性。

❶ 脚本开始是一些注释和函数定义（随后我们会讨论）。之后就是主体部分，先是初始化了几个变量，然后又导出了几个需要全局可用的变量。

❷ 复制文件之前，调用 FREESPACE 函数来确定 MP3 播放器有多少可用的存储空间。

❸ find 命令可以找出所有的 MP3 文件（其实是那些名称以“.mp3”结尾的文件）。文件名会通过管道传给下一行的 while 循环。

❹ 为什么要将 while 循环放进括号？括号意味着其中的语句是在子 shell 中运行的。但是，这里我们关注的是 while 循环和靠近脚本结尾处的 printf 语句放在一起了。由于 while 循环是在子 shell 中运行，而 find 命令将其输出通过管道传给了 while 循环，这使得在循环内部得到的计算结果无法在循环外部使用。将 while 和 printf 都放入子 shell 中，意味着两者处于相同的 shell 环境，能够共享变量。花括号也可以实现类似的效果。

> 从 bash 4.4 开始就不再需要括号了，只需要以 shell 脚本形式运行（非交互式）并设置 shell 选项 lastpipe，这可以通过将 shopt -s lastpipe 放在脚本中的 find 命令之前来实现。

我们来看看 while 循环的内部操作。

```
FILESIZE "$PATHNM"
    if ((FZ <= FREE))
    then
        echo loading $PATHNM
        cp "$PATHNM" /media/mp3
        if (( $? == 0 ))
        then
```

对于从 find 命令输出中读取到的每个文件名，使用 FILESIZE 函数（马上就会讲到）来确定文件大小。然后检查该文件是否小于 MP3 播放器当前可用的存储空间（是否有空间容纳该文件）。如果空间足够，则输出文件名，告知当前正在处理的文件，接着调用 cp 将其复制到 MP3 播放器。

记得检查复制命令是否顺利，这一点很重要。$? 中保存了上一个命令的执行结果，可以从中了解 cp 命令的执行情况。如果执行成功，将已复制文件的大小从 MP3 播放器的可用存储空间中减去。如果失败，则需要删除副本（因为文件不完整）。我们使用了 rm 命令的 -f 选项，避免删除未创建文件时出现错误信息。然后重新计算可用存储空间，以确保统计结果正确。（毕竟，复制失败也可能是因为我们估算有误、空间确实不足所致。）

在脚本的主体部分，3 个 if 语句全都在条件表达式两边使用了双括号。由于全都是数值比较，我们希望使用熟悉的运算符（如 <= 和 ==）。当然也可以使用方括号（[]），但同时需要换用 -le 和 -eq 运算符。我们在 FILESIZE 函数中用到了 if 语句的另一种形式。此时我们需要检查文件（$FN 保存着文件名）是否存在。这可以用 -e 运算符很轻松地实现，但该运算符无法在算术风格的 if 语句（使用的是括号，而非方括号）中使用。

说起算术表达式，我们来看看 REDUCE 函数的操作。

```
function REDUCE ( )
(( FREE-=${1:-0}))     # 管用，但不常见
```

大多数人编写函数时会使用花括号来分隔函数体。但 bash 中的函数体也可以是复合命令 [1]。这里选用了双括号算术表达式，我们只需要该函数做这一件事，但这种写法不常见，除非写好注释，否则可读性和可维护性方面会有问题。调用 REDUCE 时指定的值会作为第一个

（位置）参数（$1）。我们只需要将这个值从 $FREE 中减去，得到 $FREE 的新值。这就是为什么我们要使用算术表达式以及 -= 运算符。

[1]函数体可以是 {} 中的命令列表，也可以是复合命令。参见 man bash 中“Shell Function Definitions”一节以及“Compound Commands”一节。——译者注

我们来看一下 FILESIZE 函数中的两行。注释部分展示了另一种简单的实现方法，但我们想解释一种更为通用的技术，除了检查文件大小之外，它还有更值得关注的用途。注意下面这两行。

```
set -- $(ls -s "$FN")
FZ=$1
```

这里可是字短话长啊。首先，ls 命令是在子 shell（$()）中运行的。-s 选项以块为单位输出文件大小以及文件名。该命令的输出作为一系列命令行单词返回给 set 命令。set 命令的目的是解析 ls 命令所输出的单词。不少方法都可以做到这一点，但我们展示的这种技术更为实用，值得牢记在心。

set -- 获取命令行上的其余单词并将其作为新的位置参数。如果写成 set -- this is a test，那么 $1 包含 this，$3 包含 a。而 $1 和 $2 之前包含的值就全部丢失了，但在脚本中，我们将传入函数的唯一参数保存在 $FN 中。如此一来，就可以随意重用位置参数了，我们让 shell 用它们来执行解析工作。于是，$1 保存的就是文件大小，你应该也在赋值语句中看到了。（顺便说一下，在这个例子中，因为是在函数内部，所以修改的仅仅是函数的位置参数，而不是所调用脚本的位置参数。）

我们在另一个函数中又用到了同样的技术。

```
set -- $(df /media/mp3 | grep '^/dev/')
FREE=$4
```

df 命令的输出会以块为单位报告设备的可用存储空间大小。我们将输出通过管道传给 grep，因为只需要与设备信息相关的那行，所以 df 输出的标题行就不用了。只要 set 解析好参数，就可以在 $4 中得到该设备的可用存储空间了。

脚本中的注释展示了另一种解析 df 命令输出的方法。我们只需要将输出通过管道传给 awk，由后者负责解析。

```
# FREE=$(df /media/mp3 | awk '/^\/dev/ {print $4}')
```

这个版本用一对斜线之间的表达式告诉 awk 只关注以 /dev 起始的行。（其中的脱字符 ^ 将搜索锚定在行首，反斜线用于转义斜线，避免提前终止搜索表达式并将该斜线作为要查找的第一个字符。）

该用哪种方法呢？两者都涉及外部程序，一个是 grep，另一个是 awk。条条大路通罗马（bash 中如此，生活中也一样），选择权在你。根据我们的经验，先想到哪个就用哪个。

12.3.4 参考

- man df
- man grep
- man awk
- 10.4 节
- 10.5 节
- 19.8 节

12.4 刻录CD

12.4.1 问题

你的 Linux 系统中有一个满是文件的目录，你想将它刻录成 CD。需要昂贵的 CD 刻录程序？能不能通过 shell 和一些开源程序来实现？

12.4.2 解决方案

开源程序 mkisofs 和 cdrecord 可以拿来一用，此外还有一个 bash 脚本可以帮助你确保所有选项的正确性。

先将要刻录成 CD 的所有文件放进目录。例 12-4 中的脚本会接受目录作为参数，使用其中的文件制作 ISO 文件系统镜像，然后刻录 ISO 镜像。所需要的不过就是一些磁盘空间和一点时间，你也可以在 bash 脚本运行时起身溜达溜达。

> 该脚本未必能在你的系统上正常工作。我们只是将其作为 shell 脚本编程的一个示例，并非一种切实可行的 CD 刻录和备份机制。

例 12-4 ch12/cdscript

```
#!/usr/bin/env bash
# 实例文件：cdscript
# cdscript - 准备并刻录目录中的文件
#
# 用法：cdscript dir [ cddev ]
#
if (( $# < 1 || $# > 2 ))
then
    echo 'usage: cdscript dir [ cddev ]'
    exit 2
fi

# 设置默认值
SRCDIR=$1
# 你的设备可能是"ATAPI:0,0,0"或其他数字
CDDEV=${2:-"ATAPI:0,0,0"}
ISOIMAGE=/tmp/cd$$.iso                                    ❶

echo "building ISO image..."
#
# 制作ISO文件系统镜像
#
mkisofs -A "$(cat ~/.cdAnnotation)" \
  -p "$(hostname)" -V "${SRCDIR##*/}" \
  -r -o "$ISOIMAGE" $SRCDIR
STATUS=$?                                                 ❷
if (( STATUS != 0 ))
then
    echo "Error. ISO image failed."
    echo "Investigate then remove $ISOIMAGE"
    exit $STATUS
fi
```

```
echo "ISO image built; burning to cd..."
#
# 刻录CD
#
SPD=8
OPTS="-eject -v fs=64M driveropts=burnproof"
cdrecord $OPTS -speed=$SPD dev=${CDDEV} $ISOIMAGE
STATUS=$?                                                    ❸
if (( STATUS != 0 ))
then
    echo "Error. CD Burn failed."
    echo "Investigate then remove $ISOIMAGE"
    exit $STATUS
fi

rm -f $ISOIMAGE
echo "Done."
```

12.4.3 讨论

接下来我们简单看一下该脚本中一些比较奇怪的写法。

❶ 我们用 $$ 变量（包含 shell 的进程 ID）生成了一个临时文件名。只要该脚本在运行，这个 ID 值就是唯一的，因此可以用来生成绝不会重复的文件名（14.11 节中给出了一种更好的办法）。

❷ 保存 mkisofs 命令的运行状态。如果编写良好的 Unix 和 Linux 命令（以及 bash shell 脚本）运行顺利（没有出现错误），则返回 0；如果运行失败，则返回非 0 值。本来可以在下一行的 if 语句中使用 $?，但我们希望保存 mkisofs 命令的状态，万一出现故障，就将该值作为脚本的返回值。

❸ 对于 cdrecord 命令，同样保存其返回值，如果该命令失败，if 语句则为真，由 exit 语句返回故障码。

梳理清楚下面这几行估计得花点心思。

```
mkisofs -A "$(cat ~/.cdAnnotation)" \
    -p "$(hostname)" -V "${SRCDIR##*/}" \
    -r -o "$ISOIMAGE" $SRCDIR
```

这 3 行其实是一行完整的 bash 输入，只是通过行尾的换行符转义将其分成了 3 部分。千万别在 \ 之后加空格。这些不过是冰山一角而已。另外还有 3 个子 shell，它们各自的输出用于构建最终的 mkisofs 命令行。

首先调用 cat 命令来输出文件 .cdAnnotation 的内容，该文件位于调用该脚本的用户的主目录下。目的是为 -A 选项提供一个字符串，mkisofs 手册页中将其描述为“会被写入卷头部（volume header）的一个文本字符串”。与此类似，-p 选项需要另一个这样的字符串，用于指明该镜像的编制人。对于这个脚本，似乎用运行脚本的主机名作为编制人就挺方便的，因此我们在子 shell 中执行 hostname 命令（虽然用 $HOSTNAME 变量更高效）。最后，通过 -V 选项将卷名指定为待刻录文件所在的目录名。该目录在调用脚本时作为命令行参数出现，我们用 ## 运算符去除路径名中可能存在的前导部分（模式为 */），例如，/usr/local/stuff 经过处理后变成 stuff。

12.4.4 参考

- 5.20 节
- 14.11 节

12.5 比较文档

12.5.1 问题

比较两个文本文件很容易（参见 17.10 节）。那比较办公应用程序套件生成的文档呢？这种文档并不是以文本形式保存的，该如何比较呢？如果同一文档有两个版本，你想知道两个版本之间都有哪些变动（如果有的话），除了将它们打印出来逐页比较外，还有别的办法吗？

12.5.2 解决方案

首先，LibreOffice 这种办公套件在保存文档时采用的是开放文档格式（opendocument format，ODF）。只要你的文件为 ODF 格式，就可以用 shell 脚本仅比较文件内容。这里强调“内容”一词是因为文档格式方面的差异是另一回事，内容（往往）才是决定哪个版本较新的决定性因素，或者说内容对于最终用户而言更为重要。

例 12-5 中的 bash 脚本可用于比较两个保存为 ODF 格式的 LibreOffice 文档（这种文档的约定后缀名是 .odt，以此表明该文档是面向文本的，而非电子表格或演示文稿）。

例 12-5 ch12/oodiff

```
#!/usr/bin/env bash
# 实例文件：oodiff
# oodiff -- 显示两个OpenOffice/LibreOffice文件内容之间的差异
# 仅适用于.odt文件
#
function usagexit ()
{
    echo "usage: ${0##*/} file1 file2"
    echo "where both files must be .odt files"
    exit $1
} >&2                                                    ❶

# 确保作为参数的两个文件可读且文件名以.odt结尾
if (( $# != 2 ))
then
    usagexit 1
fi
if [[ $1 != *.odt || $2 != *.odt ]]
then
    usagexit 2
fi
if [[ ! -r $1 || ! -r $2 ]]
then
    usagexit 3
fi

BAS1=$(basename "$1" .odt)
BAS2=$(basename "$2" .odt)

# 将其解压缩到临时目录
PRIV1="/tmp/${BAS1}.$$_1"
PRIV2="/tmp/${BAS2}.$$_2"
```

```
# 生成绝对路径
HERE=$PWD
if [[ ${1:0:1} == '/' ]]                                    ❷
then
    FULL1="${1}"
else
    FULL1="${HERE}/${1}"
fi

# 生成绝对路径
if [[ ${2:0:1} == '/' ]]
then
    FULL2="${2}"
else
    FULL2="${HERE}/${2}"
fi

# 创建目录并检查是否创建成功
# 注意：{和}的两边必须要有空白字符
#       {}中的命令列表必须以;结尾
mkdir "$PRIV1" || { echo "Unable to mkdir '$PRIV1'" ; exit 4; }
mkdir "$PRIV2" || { echo "Unable to mkdir '$PRIV2'" ; exit 5; }

cd "$PRIV1"
unzip -q "$FULL1"
sed -e 's/>/>\                                              ❸
/g' -e 's/</\
</g' content.xml > contentwnl.xml

cd "$PRIV2"
unzip -q "$FULL2"
sed -e 's/>/>\
/g' -e 's/</\
</g' content.xml > contentwnl.xml

cd "$HERE"

diff "${PRIV1}/contentwnl.xml" "${PRIV2}/contentwnl.xml"

rm -rf "$PRIV1" "$PRIV2"
```

12.5.3 讨论

该脚本可行的原因在于 LibreOffice 文件在存储时类似于 ZIP 文件。将其解压后会生成若干个 XML 文件，正是这些文件定义了你的文档。其中一个文件包含了文档内容，也就是不带任何格式的文本段落

（但是 XML 标签会将各个文本片段及其格式关联起来）。脚本的基本思路是解压两个文件，然后用 diff 比较两者的内容，最后清理现场。

为了便于查看文件差异，需要多加一步操作。由于文件的所有内容都在 XML 中，而且其中没什么换行符，该脚本会在每个起始标签之后和结束标签（标签内容以斜线起始，例如 </ ... >）之前各插入一个换行符。虽然这会产生大量空行，但也能让 diff 关注真正的差异：文本内容。

就其中的 shell 语法而言，你已经在其他实例中见过了，但有几处语法也许仍值得讲解一下，以确保你能够理解脚本的来龙去脉。

❶ 该行将 shell 函数的所有输出全部重定向到 STDERR。这样做合情合理，因为这些属于帮助信息，并非脚本的正常输出。直接重定向整个函数定义意味着我们就不用对所有的输出行逐个重定向了。

❷ 这里出现了一个简洁的表达式 if [[${1:0:1} == '/']]，该表达式检查第一个参数是否以斜线字符开头。${1:0:1} 是 shell 变量的子串提取语法。变量 ${1} 是第一个位置参数。:0:1 语法表示从 0 偏移开始，并提取长度为 1 个字符的子串。

❸ 这几行 sed 命令可能有点不好理解，因为其中涉及将转义后的换行符作为 sed 替换字符串的一部分。替换表达式将第一处替换中的每个 > 以及第二处替换中的每个 < 更换为自身加上一个换行符。我们对内容文件执行此操作是为了分散（spread out）XML，将包含内容的各行独立出来。这样 diff 就不会显示任何 XML 标签，而是只显示内容文本。

12.5.4 参考

- 8.7 节
- 13.3 节
- 14.11 节
- 17.3 节
- 17.10 节

第 13 章　与解析相关的任务

程序员可能会对本章的任务深有感触。这里展示的实例未必就比书中的其他实例更高级，但如果你不是程序员，这些任务可能看起来挺晦涩，或者与你的日常 bash 应用没什么关系。我们不会过多解释为什么你得应对这类情景（作为程序员，你多少也知道一些）。即便对此没有什么认识，你也应该阅读本章，看看从中能够学到哪些和 bash 相关的知识。

其中部分实例涉及命令行参数解析。回忆一下，指定 shell 脚本参数的典型方式是前导减号加上单字母选项名。例如，要想脚本输出较少的信息，可以使用 -q 选项表示静默模式。选项有时会带有参数。例如，需要指定用户名的用户选项时可以使用 -u，随后跟上用户名。本章的第一个实例将会阐明这种区别。

一些 Linux 命令也接受长格式选项。以上一个短格式选项 -u 为例，命令可能还支持形如 --user=username 的长格式。我们不打算展示任何长格式选项，尽管这种选项可用于我们要介绍的某些技术。解析长格式选项的最佳方法是使用 getopt（注意，没有 s）命令。

13.1　解析shell脚本参数

13.1.1　问题

你想给脚本加入一些能够改变其行为的选项。可以直接进行解析，使用 $# 了解有多少参数，使用 ${1:0:1} 测试首个参数的第一个字符是否为减号。另外还要用到一些 if/then 或 case 逻辑来识别是哪个选项，以及选项是否带有参数。如果用户没有提供所需要的参数，或者调用脚本时将两个选项组合在了一起（如 -ab），该如何处理？也要解析这种形式的选项吗？解析 shell 脚本选项的这种需求屡见不鲜。很多脚本都带有选项。对此有没有更标准化的处理方法？

13.1.2 解决方案

使用 bash 的内建命令 getopts 来帮助解析选项。

例 13-1 主要基于 getopts 手册页中的例子演示了该命令的用法。

例 13-1 ch13/getopts_example

```
#!/usr/bin/env bash
# 实例文件：getopts_example
#
# getopts用法
#
aflag=
bflag=
while getopts 'ab:' OPTION
do
    case $OPTION in
        a) aflag=1
           ;;
        b) bflag=1
           bval="$OPTARG"
           ;;
        ?) printf "Usage: %s: [-a] [-b value] args\n" ${0##*/} >&2
           exit 2
           ;;
    esac
done
shift $(($OPTIND - 1))

if [ "$aflag" ]
then
    printf "Option -a specified\n"
fi
if [ "$bflag" ]
then
    printf 'Option -b "%s" specified\n' "$bval"
fi
printf "Remaining arguments are: %s\n" "$*"
```

13.1.3 讨论

这里支持两种选项。第一种是独立选项，这种选项比较简单。它通常代表改变命令行为的一个标志。ls 命令的 -l 选项就是一个示例。

第二种选项还需要参数。例如，mysql 命令的 -u 选项就要求提供用户名，如 mysql -u sysadmin。我们来看看 getopts 是如何解析这两种选项的。

getopts 接受两个参数。

```
getopts 'ab:' OPTION
```

第一个参数是选项字母列表。第二个参数是一个 shell 变量的名称。我们的示例仅定义了 -a 和 -b 为有效选项，因此 getopts 的第一个参数就是这两个字母和一个冒号。这个冒号是干吗的？它表明 -b 选项还需要一个参数，如 -u *username* 或 -f *filename*。冒号需要紧挨着带有参数的选项字母。举例来说，如果只有 -a 带有参数，那么我们就得写成 a:b。

在解析 shell 脚本参数（$1、$2 等）的过程中，getopts 会将第二个参数指定的变量设置成它所解析出的值。如果发现带有前导减号的字母，getopts 会将其视为选项并将该字母保存于指定变量（这个例子中是 $OPTION）。然后返回真值（也就是 0），以便 while 循环接着处理该选项。接着重复调用 getopts，不断解析选项，直到处理完毕（或者碰到双减号 --，该符号允许用户明确告知选项到此结束）。此时 getopts 返回假（非 0 值），while 循环结束。

如果在循环内部发现待处理的选项字母，我们用 case 语句来判断变量 $OPTION，要么设置标志，要么采取相应的处理。对于接受参数的选项，参数会保存在 shell 变量 $OPTARG（固定名称，与我们选用的变量 $OPTION 无关）。我们得将这个值赋给其他变量，因为随着解析过程的进行，每次调用 getopts 时，变量 $OPTARG 就会被重置。

case 语句的第三个分支是问号，这个 shell 模式匹配任意单个字符。当找到非预期选项时（这个例子中的预期选项是 'ab:'），getopts 会在指定的变量（这个例子中是 $OPTION）放入一个普通的问号。因此，我们本可以将 case 语句分支写成 \?) 或 '?') 来进行严格匹配，但使用模式 ? 来匹配任意单个字符，可以作为一种方便的默认分支写法，它既能匹配普通的问号，也能匹配其他任意字符。

在输出的用法信息中，我们对手册页中的示例脚本做了两处改动。首先，我们用 ${0##*/} 获得脚本名称，同时删除了可能包含的路径名。其次，我们将用法信息重定向到标准错误（>&2），因为这本来就是这种信息该去的地方。当遇到未知的选项或缺失参数时，所有来自 getopts 的错误消息都会写入标准错误。我们将用法信息也加入该行列。

当 while 循环结束时，接着要执行的是：

```
shift $(($OPTIND - 1))
```

其中的 shift 语句用于将 shell 脚本的位置参数（$1、$2 等）向左移动指定次数（在这个过程中，更靠左的位置参数会被丢弃）。变量 $OPTIND 是参数索引 [1]，getopts 用它来跟踪接下来要解析的参数。一旦完成解析，我们就可以用 shift 语句将已经处理过的选项全都丢弃。例如，如果有下列命令行：

[1] $OPTIND 的初始值是 1。——译者注

```
myscript -a -b alt plow harvest reap
```

解析完选项后，$OPTIND 的值应该是 4。通过将位置参数向左移动 $OPTIND - 1 次，可以丢弃所有解析过的选项，然后使用 echo $* 得到：

```
plow harvest reap
```

剩下的（非选项）参数留给脚本使用（可能用于 for 循环）。示例脚本中的最后一行是 printf，用于显示其余的所有参数。

13.1.4 参考

- help case
- help getopts
- help getopt
- 5.8 节

- 5.11 节
- 5.12 节
- 5.18 节
- 5.20 节
- 6.10 节
- 6.14 节
- 6.15 节
- 13.2 节

13.2 解析参数时使用自定义错误消息

13.2.1 问题

你正在使用 getopts 解析 shell 脚本选项，但是不喜欢它在遇到错误输入时显示的错误信息。能不能在使用 getopts 的同时编写自己的错误处理？

13.2.2 解决方案

如果只是想让 getopts 安安静静，不报告任何错误，那么在开始解析前执行赋值操作 OPTERR=0 即可。但如果希望 getopts 在不显示错误消息的情况下提供更多信息，可以在选项列表之前加上一个冒号，如例 13-2 中的脚本所示。（选项列表两边的引号是可选的。）

例 13-2 ch13/getopts_custom

```
#!/usr/bin/env bash
# 实例文件：getopts_custom
#
# 解析参数时使用自定义错误消息
#
aflag=
bflag=
# 因为不想getopts生成错误消息，
# 而是希望该脚本输出自己的错误消息，
# 所以我们在选项列表前加上':'，使得getopts保持静默
while getopts :ab: FOUND
do
```

```
    case $FOUND in
        a) aflag=1
           ;;
        b) bflag=1
           bval="$OPTARG"
            ;;
        \:) printf "argument missing from -%s option\n" $OPTARG
            printf "Usage: %s: [-a] [-b value] args\n" ${0##*/}
            exit 2
            ;;
        \?) printf "unknown option: -%s\n" $OPTARG
            printf "Usage: %s: [-a] [-b value] args\n" ${0##*/}
            exit 2
            ;;
        esac >&2
    done
shift $(($OPTIND - 1))

if [ "$aflag" ]
then
    printf "Option -a specified\n"
fi
if [ "$bflag" ]
then
    printf 'Option -b "%s" specified\n' "$bval"
fi
printf "Remaining arguments are: %s\n" "$*"
```

13.2.3 讨论

该脚本和 13.1 节中的脚本基本上差不多，更多背景知识参见 13.1.3 节。不同之处是这里的 getopts 可能会返回冒号。这种情况会出现在某个参数缺失的时候（例如，用户调用脚本时使用了 -b 选项，但没有指定对应的参数）。这个示例将选项字母存入 $OPTARG，这样就知道哪个选项丢失了参数。

与此类似，如果出现了不支持的选项（例如，用户在调用脚本时使用了 -d 选项），getopts 会返回问号作为 $FOUND 的值并将选项字母（这种情况下是 d）存入 $OPTARG，以便在错误消息中使用。

我们在冒号和问号前都加上了反斜线，以此指明两者都是普通字符，并非特殊模式或 shell 语法。虽然没必要非得对冒号这么做，但同时转义这两个符号看起来比较对称。

我们在 esac（case 语句的结尾）后面加上了 I/O 重定向，这样一来，各种 printf 命令的输出都会重定向到标准错误。这符合标准错误的用途，比起重定向每个 printf 语句，这种写法更简单。

13.2.4 参考

- help case
- help getopts
- help getopt
- 5.8 节
- 5.11 节
- 5.12 节
- 5.18 节
- 5.20 节
- 6.15 节
- 13.1 节

13.3 解析HTML

13.3.1 问题

你想将 HTML 中的字符串提取出来。例如，在一堆 HTML 中提取出 <a> 标签内形如 href="*urlstringstuff*" 的字符串。

13.3.2 解决方案

要想用 shell 快捷地解析 HTML，而且也不要求万无一失，可以尝试下列做法。

```
cat $1 | sed -e 's/>/>\
/g' | grep '<a' | while IFS='"' read a b c ; do echo $b; done
```

13.3.3 讨论

用 bash 解析 HTML 绝非易事，主要因为 bash 基本上是面向行的，而 HTML 的设计是将换行符视为空白字符。因此，我们经常会看到跨多行的 HTML 标签。

```
<a href="blah..." rel="blah..." media="blah..."
  target= "blah..." >
```

<a> 标签的写法有两种，一种要使用独立的结束标签 ，另一种则不用，其是以 /> 作为结尾的单个 <a> 标签。在起止标签之间可能存在占据了一行或数行的多个其他标签，这在解析的时候实在有些让人挠头，我们给出的简单的 bash 解析方法往往不能确保万无一失。

接下来我们将分步骤讲解解决方案。首先，将出现在一行中的多个标签拆分成一行一个标签。

```
cat file | sed -e 's/>/>\
/g'
```

没错，反斜线后面紧跟着换行符，以此将每个标签的结尾字符（>）替换成该字符本身加上一个换行符。这就使得一行只出现一个标签，另外可能还会伴随少量额外的空行。最后的 g 告诉 sed 执行**全局**搜索替换，也就是说，如果需要，可以在一行上进行多次替换。

然后将输出通过管道传给 grep 来过滤出包含 <a 标签的行。

```
cat file | sed -e 's/>/>\
/g' | grep '<a'
```

或者过滤出那些包含双引号的行。

```
cat file | sed -e 's/>/>\
/g' | grep '".*"'
```

单引号告诉 shell 将内部字符视为普通字符，不对其执行任何 shell 扩展，剩下的是一个匹配双引号的正则表达式，然后是任意多个（*）任意字符（.），接着是另一个双引号。（该正则表达式无法匹配跨行的字符串。）

要想解析出双引号中的内容，一种技巧是用 shell 的内部字段分隔符（$IFS）将双引号（"）作为分隔符。awk 及其 -F（字段分隔符）选项也可以实现类似的效果。

例如：

```
cat $1 | sed -e 's/>/>\
/g' | grep '".*"' | awk -F'"' '{ print $2}'
```

（如果只想要 <a 标签，而非所有被引用的字符串，可以使用 grep '<a'。）

要是不想使用 awk，而想使用 $IFS 的话，写法如下所示。

```
cat $1 | sed -e 's/>/>\
/g' | grep '<a' | while IFS='"' read PRE URL POST ; do echo $URL;
done
```

其中，grep 的输出通过管道传给 while 循环，后者会读取输入，将其分别保存在 3 个字段（PRE、URL 和 POST）。read 命令之前的 IFS='"' 使得我们所设置的环境变量仅对 read 命令有效，并不会影响整个脚本。因此，解析时会将引号作为输入行的单词分隔符。PRE 中保存的是第一个引号之前的内容，URL 中保存的是第一个引号之后至下一个引号之前的内容，POST 中保存的是随后的所有内容。脚本只需要显示出第二个变量 URL 中的内容即可；这也就是引号之间的所有字符。

13.3.4 参考

- man sed
- man grep

13.4 将输出解析到数组

13.4.1 问题

你想将某个程序或脚本的输出放入数组。

13.4.2 解决方案

例 13-3 演示了如何使用数组将输出解析成多个单词。

例 13-3 ch13/parseViaArray

```
#!/usr/bin/env bash
# 实例文件：parseViaArray
#
# 找出文件大小
# 使用数组将ls -l的输出解析成多个单词
LSL=$(ls -ld $1)

declare -a MYRA
MYRA=($LSL)

echo the file $1 is ${MYRA[4]} bytes.
```

13.4.3 讨论

我们的示例会获取 `ls -l` 命令的输出，并使用数组将其解析成多个单词，然后引用数组元素来获得各个单词。（记住，数组的索引是从 0 开始的，因此索引 4 指向是第 5 个元素。） `ls -l` 命令的典型输出与以下类似（在你的系统上可能会因为地区设置而有所不同）。

```
-rw-r--r-- 1 albing users 113 2006-10-10 23:33 mystuff.txt
```

如果编写脚本时值是已知的，那么初始化数组并不是什么难事。语法很简单。接下来我们先声明一个数组，然后为其赋值。

```
declare -a MYRA
MYRA=(first second third home)
```

在括号中使用变量也可以实现相同的效果。只要确保别给变量加引号就行了。`MYRA=$("$LSL")` 会将整个字符串作为第一个值，因为该字符串被视为一个整体出现在引号中。`${MYRA[0]}` 将是唯一的数组元素，其中包含整个字符串，这可不是你想要的。

我们也可以合并步骤以缩短脚本。

```
declare -a MYRA
MYRA=($(ls -ld $1))
```

如果你想知道新数组中有多少元素，查看变量 ${#MYRA[*]} 或 ${#MYRA[@]} 即可，两者都得键入不少特殊字符。

13.4.4 参考

- 5.23 节

13.5 用函数调用解析输出

13.5.1 问题

你想将程序的输出解析到各个变量中，以便在程序的其他地方使用。在遍历值时，数组是不错的选择，但如果不通过索引来单独引用各个值，可读性就不太好了。

13.5.2 解决方案

使用函数调用将输出解析成多个单词，如例 13-4 所示。

例 13-4 ch13/parseViaFunc

```
#!/usr/bin/env bash
# 实例文件：parseViaFunc
#
# 通过函数调用解析ls -l的输出
# ls -l的输出类似于以下这样：
# -rw-r--r--  1 albing users 126 Jun 10 22:50 fnsize

function lsparts ()
{
    PERMS=$1
    LCOUNT=$2
    OWNER=$3
```

```
    GROUP=$4
    SIZE=$5
    CRMONTH=$6
    CRDAY=$7
    CRTIME=$8
    FILE=$9
}

lsparts $(ls -l "$1")

echo $FILE has $LCOUNT 'link(s)' and is $SIZE bytes long.
```

运行该脚本的结果如下所示。

```
$ ./fnsize fnsize
fnsize has 1 link(s) and is 311 bytes long.
$
```

13.5.3 讨论

我们将待解析的文本放进函数调用，然后让 bash 来完成解析工作。调用函数和调用脚本的方法大同小异。bash 会将解析出的单词赋给 $1、$2 等变量。我们的函数只是将各个位置参数保存在单独的变量中。如果未被声明为局部变量，那么变量在函数内外均可使用。

我们将 ls 命令的 $1 放进了引号，以免变量中的文件名包含空格。引号可以将其视为一个整体，这样 ls 得到的就只是单个文件名，而不是一系列文件名。

我们给 links(s) 也加上了引号（'links(s)'），避免 bash 对括号做特殊处理。也可以将这行中除 echo 之外的内容放进双引号，这样就不会影响到变量替换（$FILE 等）。

根据计算机和 ls 命令呈现日期的方式，你可能需要调整字段列表或是为 ls 命令添加选项，以调整其输出。例如，ls -l --timestyle="long-iso" 生成的输出格式就略有不同，其中月份和天数采用的是 YYYY-MM-DD 格式。你得将 CRMONTH 和 CRDAY 替换成单个变量，比如 CRDATE，同时调整字段编号。

13.5.4 参考

- 10.4 节
- 10.5 节
- 13.9 节
- 17.7 节

13.6 用read语句解析文本

13.6.1 问题

用 bash 解析文本的方法有很多种。如果不想使用函数来解析，还有别的方法吗？

13.6.2 解决方案

使用例 13-5 中的 read 语句。

例 13-5 ch13/parseViaRead

```
#!/usr/bin/env bash
# 实例文件：parseViaRead
#
# 用read语句解析ls -l的输出
# ls -l的输出类似于以下这样：
# -rw-r--r--  1 albing users 126 2006-10-10 22:50 fnsize

ls -l "$1" | { read PERMS LCOUNT OWNER GROUP SIZE CRDATE CRTIME
FILE ;
                  echo $FILE has $LCOUNT 'link(s)' and is $SIZE
bytes long. ;
              }
```

13.6.3 讨论

我们将所有的解析工作都交给了 read。它会将输入拆分成**单词**，彼此之间以空白字符分隔，各个单词保存于 read 语句指定的变量中。

实际上，甚至可以通过将 bash 的 $IFS 变量设置成任何需要的字符来修改分隔符。记得完事之后恢复原状！

从 ls -l 的输出示例可以看到，我们尝试选择符合命令输出中每个单词含义的变量名。由于 FILE 是最后一个变量，因此任何多出的字段都将成为该变量内容的一部分。如果文件名中包含空格，如“Beethoven Fifth Symphony”，则 3 个单词都会出现在 $FILE 之中。

13.6.4 参考

- 2.14 节
- 19.8 节

13.7 用read将输入解析至数组

13.7.1 问题

每行输入中的单词数量各不相同，你无法将单词赋给预先确定好的变量。

13.7.2 解决方案

使用 read 命令的 -a 选项将各个单词读入数组变量。

```
read -a MYRAY
```

13.7.3 讨论

无论输入是来自用户还是管道，read 都会将其解析成单词并将各个单词放入相应的数组元素。不用专门声明数组变量，这种书写方式足以令该变量被视为数组。可以使用 bash 数组语法来引用每个元素。bash 数组的索引从 0 开始，因此，这里输入行中的第二个单词位于

${MYRAY[1]} 中。数组中有多少个元素就代表有多少个单词。示例中的数组大小为 ${#MYRAY[@]}。

13.7.4 参考

- 3.5 节
- 13.6 节

13.8 读取整个文件

13.8.1 问题

你想读入整个文件并对其解析。是不是一定得使用 for 循环一次读取一行？有没有更便捷的方法？

13.8.2 解决方案

使用 bash 的 mapfile 或 readarray 命令。这两个命令是等同的，接受相同的参数，能够将整个文件读入数组，每个数组元素对应文件中的一行。

用 readarray 还是 mapfile 属于观点问题：看你考虑的是目标（数组）还是源（数据文件）？选择更适合你的那个就行了。两者是可以互换的。

以下是 mapfile 命令的一个示例，更完整的部分在 13.8.3 节。

```
mapfile -t -s 1 -n 1500 -C showprg -c 100 BIGDATA <
/tmp/myfile.data
```

该命令会丢弃（跳过）第一行输入（-s 1），一直读取 1500 行（-n 1500）并删除每行行尾的换行符（-t）。每读取 100 行（-c 100）就调用一次用户自定义函数 showprg（以显示文件读取进度，默认是每 5000 行调用一次）。结果会放进数组 BIGDATA，每个元素对应一行。命令输入通过重定向从文件导入。

13.8.3 讨论

以下是 mapfile（你喜欢的话，也可以改用 readarray）用法示例的第一部分。这部分读取文件并显示读取进度。然后它会输出读取了多少行，也就是数组的大小。

```
# 用mapfile读取$1中的文件

# 用点号显示读取进度
function showprg ()
{
    printf "."
}

# 创建一个比较大的数据文件备用
ls -l /usr/bin > /tmp/myfile.data

# 将数据文件读入数组BIGDATA
mapfile -t -s 1 -n 1500 -C showprg -c 100 BIGDATA <
/tmp/myfile.data

# 在showprg输出的结尾加上一个换行符
echo

# 读取了多少行?
siz=${#BIGDATA[@]}
echo "size: ${siz}"
```

showprg 函数会在每次被调用时输出一个点号（不包括换行符）。这样就可以在读取大文件时显示进度。如果想的话，还可以实现一些更炫的效果。毕竟，你可以调用任何函数。

现在文件已经被读入数组，那么该怎么处理这些数据？在本例，数组中包含的是 ls 命令冗长的输出。现在我们可以一次处理一行，输出其中某些数据。

```
# 在输出时对行进行计数
for((i=0; i<siz; i++))
do
    ALINE=${BIGDATA[i]}
    if [[ ${ALINE:0:1} == 'l' ]]    # 只处理符号链接
    then
        # 输出相关的子串
```

```
        printf "%4d: %s\n" $i "${ALINE:48}"
    fi
done

rm /tmp/myfile.data      # 清理文件
```

这里脚本会查看每行的首字符，如果是 `l`，则从索引为 `48` 的字符开始输出该行。因为文件中的数据来自 `ls` 命令的“长格式”输出，所以字符 `l` 表明对应的是符号链接。（与此类似，字符 `d` 表明对应的是目录，但此处不对其做处理。）

以下输出摘自完整脚本的运行结果。第一行显示的点号出现在文件读取过程中。

```
..............
size: 1500
(other output, and then)
1307: rsh -> /etc/alternatives/rsh
1311: rtstat -> lnstat
1315: rview -> /etc/alternatives/rview
(even more output)
```

13.8.4 参考

- 13.7 节

13.9 正确书写复数形式

13.9.1 问题

你想在出现多个对象时使用名词的复数形式，但又不想代码中到处都是 `if` 语句。

13.9.2 解决方案

例 13-6 演示了一种正确书写复数形式的方法。

例 13-6 ch13/pluralize

```
#!/usr/bin/env bash
# 实例文件：pluralize
#
# 如果$2的值不等于1或-1，则在单词末尾加上s，
# 将其改为复数形式。
# 该函数只会添加s，算不上特别聪明
#
function plural ()
{
    if [ $2 -eq 1 -o $2 -eq -1 ]
    then
        echo ${1}
    else
        echo ${1}s
    fi
}

while read num name
do
    echo $num $(plural "$name" $num)
done
```

13.9.3 讨论

虽然该函数只是简单地添加 s，但足以应对不少名词了。可是它并没有对参数数量或内容做任何错误检查，要想将这个脚本用于正式的应用程序，此类检查还是少不了的。

在调用函数 plural 时，我们为变量 name 加上了引号，以防该变量中包含空格。毕竟其中的值来自 read 语句，而 read 语句中的最后一个变量会得到输入行中剩余所有的文本。以下示例就可以看出这一点。

我们将例 13-6 中的代码放进脚本 pluralize，并用下列数据运行该脚本：

```
$ cat input.file
1 hen
2 duck
3 squawking goose
4 limerick oyster
5 corpulent porpoise
```

```
$ ./pluralize < input.file
1 hen
2 ducks
3 squawking gooses
4 limerick oysters
5 corpulent porpoises
```

“gooses”并非正确的英语拼写，但这只是脚本按部就班得到的处理结果。如果更喜欢 C 语言风格的语法，你可以将 `if` 语句改成如下所示的代码。

```
if (( $2 == 1 || $2 == -1 ))
```

方括号（也就是内建命令 `test`）属于较旧的语法形式，多见于各种 bash 版本中，不过这两种写法都没问题。哪种更容易记住就用哪种。

像 `pluralize` 这样的脚本就不用留着了，但将函数 `plural` 作为规模更大的脚本项目的一部分，没准用起来挺方便的。需要报告数量时就可以使用函数 `plural`，正如例 13-6 中的 `while` 循环所示。

13.9.4 参考

- 6.11 节

13.10 一次提取一个字符

13.10.1 问题

你有一些解析工作得处理，出于某种原因，需要一次从字符串提取一个字符。

13.10.2 解决方案

变量的子串提取功能可以解决这个问题，另一个特性允许你获取字符串长度。例 13-7 演示了两者的用法。

例 13-7 ch13/onebyone

```
#!/usr/bin/env bash
# 实例文件：onebyone
#
# 从输入中一次解析一个字符

while read ALINE
do
    for ((i=0; i < ${#ALINE}; i++))
    do
        ACHAR=${ALINE:i:1}
        # 在这里执行某些操作，如echo $ACHAR
        echo $ACHAR
    done
done
```

13.10.3 讨论

read 语句从标准输入中读取输入（一次一行），然后将其保存在变量 $ALINE 中。因为 read 语句中再无其他变量，所以该变量包含了整行的内容，但不包括行首和行尾的空白字符（由 $IFS 定义），除非你写成 IFS= read 或者仅使用 read，随后引用默认的 $REPLY 变量。

for 循环依次处理变量 $ALINE 中的每个字符。${#ALINE} 可以返回 $ALINE 的长度，以此计算出循环次数。

在每次循环中，我们将 $ALINE 的第 i 个字符赋给 $ACHAR。这够简单了吧。

13.10.4 参考

- 13.1 节
- 13.4 节
- 13.5 节
- 13.6 节
- 13.7 节

13.11 清理svn源代码树

13.11.1 问题

Subversion 的 svn status 命令会显示所有被修改过的文件，但如果源代码树中还存在 scratch 文件或其他垃圾，那么 svn 也会一并将其列出。有没有什么实用方法能清理源代码树，删除那些 Subversion 未知的文件？

在使用 svn add 命令前，Subversion 对新文件一无所知。添加新的源文件或者永久删除某些文件之后，再运行该脚本。

13.11.2 解决方案

你可以用 grep 过滤 svn status 命令的输出，然后用 read 命令读取过滤后的结果，以此创建待删除的文件列表。

```
svn status src | grep '^\?' | \
  while read status filename; do echo "$filename"; rm -rf
"$filename"; done
```

13.11.3 讨论

在 svn status 的输出中，一个文件占据一行。对于已经修改过的文件，行首字符是 M；对于新添加的文件（但尚未提交），行首字符是 A；未知文件的行首字符是问号。可以用 grep 过滤出以问号起始的那些行。我们在 while 循环中使用 read 语句处理循环。并不是非得用 echo，不过能看到被删除的文件还是有帮助的，省得出现失误或错误。至少你知道哪些文件没了。在删除文件时，我们使用了 -rf 选项，以防要删除的文件是目录，不过这么做主要是图个清静。-f 选项可以消除权限之类的问题，它只会在权限许可的最大范围内删除文件。我们将文件名引用放进引号（"$fn"），以防文件名中包含空格之类的特殊字符。

13.11.4 参考

- 6.11 节
- 附录 D

13.12 用MySQL设置数据库

13.12.1 问题

你想用 MySQL 创建并初始化多个数据库。你希望这些数据库在初始化时全部用相同的 SQL 命令。每个数据库都有自己的名称，但数据库内容全都一样，至少初始化阶段是这样的。你可能得一遍又一遍地重复设置，比如，在这些数据库作为测试套件组成部分的情况下，就需要在重新运行测试时将其重置。

13.12.2 解决方案

例 13-8 中简单的 bash 脚本可以帮助你完成这项管理任务。

例 13-8 ch13/dbiniter

```
#!/usr/bin/env bash
# 实例文件：dbiniter
#
# 通过标准文件初始化数据库
# 根据需要创建数据库

DBLIST=$(mysql -e "SHOW DATABASES;" | tail -n +2)
select DB in $DBLIST "new..."
do
    if [[ $DB == "new..." ]]
    then
        printf "%b" "name for new db: "
        read DB rest
        echo creating new database $DB
        mysql -e "CREATE DATABASE IF NOT EXISTS $DB;"
    fi

    if [ -n "$DB" ]
    then
        echo Initializing database: $DB
        mysql $DB < ourInit.sql
    fi
done
```

13.12.3 讨论

添加 tail -n +2 是为了删除数据库列表的标题部分（参见 2.12 节）。

select 用于创建菜单，显示已有的数据库。我们添加了一行 "new..." 来作为额外的选择（参见 3.7 节和 6.16 节）。

当用户想创建新数据库时，我们可以发出提示并读取数据库名称，但我们在 read 语句中指定了两个变量作为错误处理的一个小措施。如果用户输入了不止一个名称，则只使用第一个并将其保存在变量 $DB 中，其余的内容放进 $rest 并忽略。（可以加入错误检查，查看 $rest 是否为空。）

不管是创建新数据库，还是从现有数据库中选择，如果 $DB 变量不为空，则调用 mysql 命令，并将作为标准初始化序列保存在文件 ourInit.sql 中的 SQL 语句传给它。

如果打算使用这种脚本，你可能还得给 mysql 命令添加一些参数，如 -u 和 -p，以提示用户名和密码。这取决于数据库及其权限的配置方式，或者 MySQL 的默认配置是否设置了 .my.cnf 文件。

我们还可以加入错误检查，查看新数据库是否成功创建；如果不成功，执行 unset DB，跳过初始化步骤。不过就像很多数学教科书中说的那样：“我们将此作为练习留给你。”

13.12.4 参考

- 2.12 节
- 3.7 节
- 6.16 节
- 14.20 节

13.13 提取数据中的特定字段

13.13.1 问题

你需要从每行输出中提取一个或多个字段。

13.13.2 解决方案

如果存在字段分隔符，使用 cut 就能很轻松地办到，哪怕字段首尾的分隔符不一样。

```
# 这个问题很简单：该NetBSD系统中都有哪些用户，
# 他们的主目录和shell都是什么？
$ cut -d':' -f1,6,7 /etc/passwd
root:/root:/bin/csh
toor:/root:/bin/sh
daemon:/:/sbin/nologin
operator:/usr/guest/operator:/sbin/nologin
bin:/:/sbin/nologin
games:/usr/games:/sbin/nologin
postfix:/var/spool/postfix:/sbin/nologin
named:/var/chroot/named:/sbin/nologin
ntpd:/var/chroot/ntpd:/sbin/nologin
sshd:/var/chroot/sshd:/sbin/nologin
smmsp:/nonexistent:/sbin/nologin
uucp:/var/spool/uucppublic:/usr/libexec/uucp/uucico
nobody:/nonexistent:/sbin/nologin
jp:/home/jp:/usr/pkg/bin/bash

# 该系统中的哪个shell用得最多？
$ cut -d':' -f7 /etc/passwd | sort | uniq -c | sort -rn
10 /sbin/nologin
2 /usr/pkg/bin/bash
1 /bin/csh
1 /bin/sh
1 /usr/libexec/uucp/uucico

# 我们来看看前两级目录
$ cut -d':' -f6 /etc/passwd | cut -d'/' -f1-3 | sort -u
/
/home/jp
/nonexistent
/root
/usr/games
/usr/guest
/var/chroot
/var/spool
```

如果分隔符包含多个空白字符，或者需要重新安排输出字段的顺序，可以使用 awk。注意，→代表输出中的制表符。默认的输出分隔符是

空格，但可以通过 $OFS 修改。

```
# 用户名、主目录和shell，
# 交换后两个字段并使用制表符作为分隔符
$ awk 'BEGIN {FS=":"; OFS="\t"; } { print $1,$7,$6; }' /etc/passwd
root → /bin/csh → /root
toor → /bin/sh → /root
daemon → /sbin/nologin → /
operator → /sbin/nologin → /usr/guest/operator
bin → /sbin/nologin → /
games → /sbin/nologin → /usr/games
postfix → /sbin/nologin → /var/spool/postfix
named → /sbin/nologin → /var/chroot/named
ntpd → /sbin/nologin → /var/chroot/ntpd
sshd → /sbin/nologin → /var/chroot/sshd
smmsp → /sbin/nologin → /nonexistent
uucp → /usr/libexec/uucp/uucico → /var/spool/uucppublic
nobody → /sbin/nologin → /nonexistent
jp → /usr/pkg/bin/bash → /home/jp

# 多个空白字符，交换第2个和第3个字段，删除第1个字段
$ grep '^# [1-9]' /etc/hosts | awk '{print $3,$2}'
10.255.255.255 10.0.0.0
172.31.255.255 172.16.0.0
192.168.255.255 192.168.0.0
```

grep -o 只显示匹配指定模式的文本。如果无法像上面那样描述分隔符，这就特别有用了。假设你需要提取文件中出现的所有 IP 地址。注意，我们使用 egrep 是鉴于其正则表达式（regex）功能，但是 -o 选项应该适用于所有 GNU grep 版本（非 GNU 版本可能无法使用，可查阅相关文档）。

```
$ cat has_ipas
This is line 1 with 1 IPA: 10.10.10.10
Line 2 has 2; they are 10.10.10.11 and 10.10.10.12.
Line three is ftp_server=10.10.10.13:21.

$ egrep -o '[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}'
has_ipas
10.10.10.10
10.10.10.11
10.10.10.12
10.10.10.13
```

13.13.3 讨论

各种可能性数不胜数，这里列举的甚至连皮毛都算不上。这正是 Unix 工具链思想的本质所在：使用大量专注于做好一件事的小工具，根据需要将其组合来解决各种问题。

另外，处理 IP 地址的正则表达式也并未经过深思熟虑，可能会匹配到包括非法 IP 地址在内的其他内容。如果你所用的 grep 支持 -P，要想取得更好的效果，可以使用 Jeffrey E. F. Friedl 所著的《精通正则表达式（第 3 版）》一书中给出的 Perl 兼容正则表达式（Perl compatible regular expression，PCRE）。

```
$ grep -oP '([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-
4]\d|25[0-5])\.
([01]?\d\
d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])' has_ipas
10.10.10.10
10.10.10.11
10.10.10.12
10.10.10.13
$
```

或者使用 Perl。

```
$ perl -ne 'while ( m/([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?
|2[0-4]\d|25[0-5])\.
([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])/g ) {
print qq($1.$2.$3.
$4\n); }' has_ipas
10.10.10.10
10.10.10.11
10.10.10.12
10.10.10.13
$
```

13.13.4 参考

- man cut
- man awk
- man grep
- Jeffrey E. F. Friedl 所著的《精通正则表达式（第 3 版）》
- 8.4 节
- 13.15 节

- 15.10 节
- 17.16 节

13.14 更新数据文件中的特定字段

13.14.1 问题

你需要提取某行（记录）中的特定部分（字段）并更新。

13.14.2 解决方案

最简单的情况是从一行中提取单个字段，然后对其进行处理。为此，你可以使用 cut 或 awk。详见 13.13 节。

在更复杂的情况中，需要就地修改数据文件的字段。如果只是简单的搜索并替换，可以使用 sed。

例如，我们想将 NetBSD 系统中所有用户的 csh 修改成 sh。

```
$ grep csh /etc/passwd
root:*:0:0:Charlie &:/root:/bin/csh

$ sed 's;/csh$;/sh;' /etc/passwd | grep '^root'
root:*:0:0:Charlie &:/root:/bin/sh
```

如果需要对字段执行算术运算或修改特定字段中的字符串，可以使用 awk。

```
$ cat data_file
Line 1 ends
Line 2 ends
Line 3 ends
Line 4 ends
Line 5 ends

$ awk '{print $1, $2+5, $3}' data_file
Line 6 ends
Line 7 ends
Line 8 ends
Line 9 ends
```

```
Line 10 ends
# 如果第2个字段中包含'3'，将其修改为'8'并做标记
$ awk '{ if ($2 == "3") print $1, $2+5, $3, "Tweaked" ; else print
$0; }' \
data_file
Line 1 ends
Line 2 ends
Line 8 ends Tweaked
Line 4 ends
Line 5 ends
```

13.14.3 讨论

你要面对的各种情况就像手里的数据一样无穷无尽，希望这些例子能为你做好充分的准备，在修改数据时事半功倍。

13.14.4 参考

- man awk
- man sed
- THE SED FAQ
- USEFUL ONE-LINE SCRIPTS FOR SED
- 11.7 节
- 13.13 节

13.15 修剪空白字符

13.15.1 问题

你需要修剪行首或行尾的空白字符。

13.15.2 解决方案

下面给出的解决方案依赖于 bash 对待 read 和 $REPLY 的特有处理方式。本节结尾处讨论了另一种解决方案。

首先，我们展示了一个包含行首和行尾空白字符的文件。注意，我们在输出中添加了 ~~ 以表示空白字符，另外用→表示制表符。

```
# 显示样例文件中的空白字符
$ while read; do echo ~~"$REPLY"~~; done < whitespace
~~ This line has leading spaces.~~
~~This line has trailing spaces. ~~
~~ This line has both leading and trailing spaces. ~~
~~ → Leading tab.~~
~~Trailing tab. → ~~
~~ → Leading and trailing tab. → ~~
~~ → Leading mixed whitespace.~~
~~Trailing mixed whitespace. →     ~~
~~ → Leading and trailing mixed whitespace. →     ~~
$
```

用 $IFS 和内建的 $REPLY 变量修剪行首和行尾的空白字符（要想知道为什么这种方法可行，参见下一节）。

```
$ while read REPLY; do echo ~~"$REPLY"~~; done < whitespace
~~This line has leading spaces.~~
~~This line has trailing spaces.~~
~~This line has both leading and trailing spaces.~~
~~Leading tab.~~
~~Trailing tab.~~
~~Leading and trailing tab.~~
~~Leading mixed whitespace.~~
~~Trailing mixed whitespace.~~
~~Leading and trailing mixed whitespace.~~
$
```

要想只修剪行首或行尾的空格，可以使用下面的简单方法。

```
# 只修剪行首空格
$ while read; do echo "~~${REPLY## }~~"; done < whitespace
~~This line has leading spaces.~~
~~This line has trailing spaces. ~~
~~This line has both leading and trailing spaces. ~~
~~ → Leading tab.~~
~~Trailing tab. ~~
~~ → Leading and trailing tab. → ~~
~~ → Leading mixed whitespace.~~
~~Trailing mixed whitespace. →     ~~
~~ → Leading and trailing mixed whitespace. →     ~~

# 只修剪行尾空格
```

```
$ while read; do echo "~~${REPLY%% }~~"; done < whitespace
~~ This line has leading spaces.~~
~~This line has trailing spaces.~~
~~ This line has both leading and trailing spaces.~~
~~ → Leading tab.~~
~~Trailing tab. ~~
~~ → Leading and trailing tab. → ~~
~~     → Leading mixed whitespace.~~
~~Trailing mixed whitespace. →     ~~
~~     → Leading and trailing mixed whitespace. →     ~~
```

只修剪行首或行尾空白字符（包括制表符）的话，那就有点复杂了。

```
# 需要先完成这一步
$ shopt -s extglob

# 只修剪行首空白字符
$ while read; do echo "~~${REPLY##+([[:space:]])}~~"; done <
whitespace
~~This line has leading spaces.~~
~~This line has trailing spaces. ~~
~~This line has both leading and trailing spaces. ~~
~~Leading tab.~~
~~Trailing tab. ~~
~~Leading and trailing tab. → ~~
~~Leading mixed whitespace.~~
~~Trailing mixed whitespace.     → ~~
~~Leading and trailing mixed whitespace.     → ~~
$
# 只修剪行尾空白字符
$ while read; do echo "~~${REPLY%%+([[:space:]])}~~"; done <
whitespace
~~ This line has leading spaces.~~
~~This line has trailing spaces.~~
~~ This line has both leading and trailing spaces.~~
~~ → Leading tab.~~
~~Trailing tab.~~
~~ → Leading and trailing tab.~~
~~     → Leading mixed whitespace.~~
~~Trailing mixed whitespace.~~
~~    → Leading and trailing mixed whitespace.~~
```

13.15.3 讨论

此时你可能正盯着这些代码，好奇我们打算怎么样将来龙去脉讲清楚。尽管的确是有些微妙之处，但解释起来并不复杂。

开始吧。第一个示例用到了 $REPLY 变量，read 会在用户未指定变量时使用该默认变量。这个设计决定是由 Chet Ramey（bash 的维护人员）做出的：“（如果）没有指定变量，就将读入的文本行保存在 $REPLY 变量之中”，现在依然如此。

```
while read; do echo ~~"$REPLY"~~; done < whitespace
```

但如果为 read 指定了一个或多个变量，那么它会使用 $IFS 中的值（默认是空格、制表符以及换行符）来解析输入。解析过程的其中一步就是修剪行首和行尾的空白字符，这正如我们所愿。

```
while read REPLY; do echo ~~"$REPLY"~~; done < whitespace
```

可以通过 ${##} 或 ${%%} 运算符（参见 6.7 节）轻松修剪行首或行尾空格（二选一）。

```
while read; do echo "~~${REPLY## }~~"; done < whitespace
while read; do echo "~~${REPLY%% }~~"; done < whitespace
```

转换制表符有点难度。如果单纯是制表符，可以使用 ${##} 或 ${%%} 运算符并通过 Ctrl-V Ctrl-I 组合键序列插入字面制表符（literan tab）。但这么做有风险，因为可能既有空格也有制表符，而且有些文本编辑器或粗心的用户可能会剔除制表符。因此，我们启用扩展通配符匹配，通过字符类来表明意图。字符类 [:space:] 不需要 extglob 就可以使用，但我们得用 +() 表明“一次或多次出现”，否则就只能修剪掉单个空格或制表符，无法顾及多个。如果只在意空格或制表符，也可以使用 [:blank:] 来代替，因为 [:space:] 还包括垂直制表符（\v）和 DOS CR（回车，\r）在内的其他字符。

```
# 要想生效，+()部分需要启用extglob
$ shopt -s extglob
...
$ while read; do echo "~~${REPLY##+([[:space:]])}~~"; done <
whitespace
...
$ while read; do echo "~~${REPLY%%+([[:space:]])}~~"; done <
whitespace
...
```

```
# 不管用
$ while read; do echo "~~${REPLY##[[:space:]]}~~"; done <
whitespace
~~This line has leading spaces.~~
~~This line has trailing spaces. ~~
~~This line has both leading and trailing spaces. ~~
~~Leading tab.~~
~~Trailing tab. ~~
~~Leading and trailing tab. ~~
~~    → Leading mixed whitespace.~~
~~Trailing mixed whitespace.     → ~~
~~    → Leading and trailing mixed whitespace.     → ~~
```

另一种方法也是利用 $IFS 进行解析，但解析出的是字段（或单词），而不是记录（或行）。

```
$ for i in $(cat white_space); do echo ~~$i~~; done
~~This~~
~~line~~
~~has~~
~~leading~~
~~white~~
~~space.~~
~~This~~
~~line~~
~~has~~
~~trailing~~
~~white~~
~~space.~~
~~This~~
~~line~~
~~has~~
~~both~~
~~leading~~
~~and~~
~~trailing~~
~~white~~
~~space.~~
$
```

最后，虽然最初的解决方案依赖于 Chet 有关 read 和 $REPLY 的设计决定，但下面这种解决方案并不需要。

```
shopt -s extglob

while IFS= read -r line; do
```

```
echo "None: ~~$line~~" # 保留所有的空白字符
echo "Ld: ~~${line##+([[:space:]])}~~" # 修剪行首空白字符
echo "Tr: ~~${line%%+([[:space:]])}~~" # 修剪行尾空白字符
line="${line##+([[:space:]])}" # 修剪行首和……
line="${line%%+([[:space:]])}" # ……行尾的空白字符
echo "All: ~~$line~~" # 显示修剪后的内容
done < whitespace
```

13.15.4 参考

- 6.7 节
- 13.6 节

13.16 压缩空白字符

13.16.1 问题

你的文件中有一些连续的空白字符（可能是用空格填充的固定长度记录），你需要将这些空白字符压缩成单个字符或分隔符。

13.16.2 解决方案

根据需要使用 `tr` 或 `awk`。

13.16.3 讨论

如果你尝试将连续的空白字符压缩成单个字符，可以使用 `tr`，但要注意，这可能会破坏格式不佳的文件。例如，如果字段之间是由多个空格分隔，但字段内部也包含空格，将多个空格压缩成单个会使得这种差异荡然无存。假设下例用空格代替 `_`。注意，→表示输出中的制表符。

```
$ cat data_file
Header1             Header2             Header3
Rec1_Field1         Rec1_Field2         Rec1_Field3
Rec2_Field1         Rec2_Field2         Rec2_Field3
Rec3_Field1         Rec3_Field2         Rec3_Field3
```

```
$ cat data_file | tr -s ' ' '\t'
Header1 → Header2 → Header3
Rec1_Field1 → Rec1_Field2 → Rec1_Field3
Rec2_Field1 → Rec2_Field2 → Rec2_Field3
Rec3_Field1 → Rec3_Field2 → Rec3_Field3
```

如果字段分隔符不止一个字符，那 tr 就没办法了，因为它是将第一个集合中的**单个字符**转换成第二个集合中**相匹配的单个字符**。[2] 你可以用 awk 来组合或转换字段分隔符。awk 的内部字段分隔符 FS 接受正则表达式，因此你能够用其分隔的字段可就太多了。另外还有一个很方便的技巧：给任意字段赋值会使得 awk 用输出字段分隔符 OFS 重组记录，因此字段 1 自己给自己赋值，然后再输出记录，其效果相当于将 FS 转换成了 OFS，而且也不用担心数据中有多少条记录。[3]

[2] tr 命令的语法为：tr [OPTION]... SET1 [SET2]。文中提到的“第一个集合”和“第二个集合”就是指 SET1 和 SET2。详见 man tr。——译者注

[3] 类似的技巧还有 $0=$0。——译者注

在这个例子中，分隔字段的是多个空格，但字段本身也包含内部空格，因此下面这种简单的处理方法并不管用。

```
awk 'BEGIN {OFS="\t"} {$1=$1; print }' data_file1
```

数据文件如下所示。

```
$ cat data_file1
Header1             Header2             Header3
Rec1 Field1         Rec1 Field2         Rec1 Field3
Rec2 Field1         Rec2 Field2         Rec2 Field3
Rec3 Field1         Rec3 Field2         Rec3 Field
$
```

接下来的示例将两个空格赋给 FS，一个制表符赋给 OFS。然后执行赋值操作（$1 = $1），因此 awk 会重建记录，从而令两个空格被一连串的制表符替换，因此我们又用 gsub 压缩制表符，然后输出。注意，→代表输出中的制表符。由于整个输出的可读性有点不太好，

我们同时给出了 16 进制格式。再回忆一下，制表符和空格的 ASCII 编码值分别是 09 和 20。

```
$ awk 'BEGIN { FS = " "; OFS = "\t" } { $1 = $1; gsub(/\t+ ?/,
"\t"); print }' \
 data_file1
Header1 → Header2 → Header3
Rec1 Field1 → Rec1 Field2 → Rec1 Field3
Rec2 Field1 → Rec2 Field2 → Rec2 Field3
Rec3 Field1 → Rec3 Field2 → Rec3 Field3

$ awk 'BEGIN { FS = " "; OFS = "\t" } { $1 = $1; gsub(/\t+ ?/,
"\t"); print }' \
 data_file1 | hexdump -C
00000000 48 65 61 64 65 72 31 09  48 65 61 64 65 72 32 09
|Header1.Header2.|
00000010 48 65 61 64 65 72 33 0a  52 65 63 31 20 46 69 65
|Header3.Rec1 Fie|
00000020 6c 64 31 09 52 65 63 31  20 46 69 65 6c 64 32 09
|ld1.Rec1 Field2.|
00000030 52 65 63 31 20 46 69 65  6c 64 33 0a 52 65 63 32 |Rec1
Field3.Rec2|
00000040 20 46 69 65 6c 64 31 09  52 65 63 32 20 46 69 65 |
Field1.Rec2 Fie|
00000050 6c 64 32 09 52 65 63 32  20 46 69 65 6c 64 33 0a
|ld2.Rec2 Field3.|
00000060 52 65 63 33 20 46 69 65  6c 64 31 09 52 65 63 33 |Rec3
Field1.Rec3|
00000070 20 46 69 65 6c 64 32 09  52 65 63 33 20 46 69 65 |
Field2.Rec3 Fie|
00000080 6c 64 0a                                       |ld.|
00000083
```

你可以用 awk 按照相同的方法修剪行首和行尾的空白字符，不过之前提到过，这么做会替换字段分隔符，除非它们已经是空格。

```
awk '{ $1 = $1; print }' white_space
```

13.16.4 参考

- Arnold Robbins 所著的 *Effective awk Programming, 4th Edition*（O'Reilly 出版）
- Arnold Robbins 和 Dale Dougherty 合著的 *sed & awk, 2nd Edition*（O'Reilly 出版）

- 13.17 节
- A.16 节
- 表 A-25

13.17 处理固定长度记录

13.17.1 问题

你需要读取并处理的数据采用的是固定长度（也称为固定宽度）格式。

13.17.2 解决方案

使用 Perl 或 gawk 2.13 及更高版本。样例文件如下所示。

```
$ cat fixed-length_file
Header1-----------Header2-------------------------Header3---------
Rec1 Field1       Rec1 Field2                     Rec1 Field3
Rec2 Field1       Rec2 Field2                     Rec2 Field3
Rec3 Field1       Rec3 Field2                     Rec3 Field3
```

你可以用 GNU 的 gawk 来处理：将 FIELDWIDTHS 设置为正确的字段宽度，根据需要设置 OFS，然后执行赋值操作，以便 gawk 用 OFS 重建记录。但是，gawk 并不会删除原始记录中用于填充各个字段的空格，因此我们得用两个 gsub 来解决这个问题：一个处理所有内部字段 [4]，另一个处理最后一个字段。最后一步就是输出了。注意，→代表输出中的制表符。由于整个输出的可读性有点不太好，我们同时给出了十六进制格式。再回忆一下，制表符和空格的 ASCII 编码值分别是 09 和 20。

[4]这里所谓的“内部字段”指的是除最后一个字段之外的那些字段。——译者注

```
$ gawk 'BEGIN { FIELDWIDTHS = "18 32 16"; OFS = "\t" }
> { $1 = $1; gsub(/ +\t/, "\t"); gsub(/ +$/, ""); print }' fixed-
length_file
Header1----------- → Header2------------------------ → Header3---
------
Rec1 Field1 → Rec1 Field2 → Rec1 Field3
```

```
Rec2 Field1 → Rec2 Field2 → Rec2 Field3
Rec3 Field1 → Rec3 Field2 → Rec3 Field3

$ gawk 'BEGIN { FIELDWIDTHS = "18 32 16"; OFS = "\t" }
> { $1 = $1; gsub(/ +\t/, "\t"); gsub(/ +$/, ""); print }' fixed-
length_file \
> | hexdump -C
00000000 48 65 61 64 65 72 31 2d 2d 2d 2d 2d 2d 2d 2d 2d |Header1-
---------|
00000010 2d 2d 09 48 65 61 64 65 72 32 2d 2d 2d 2d 2d 2d |-
-.Header2------|
00000020 2d 2d 2d 2d 2d 2d 2d 2d 2d 2d 2d 2d 2d 2d 2d 2d |--------
--------|
00000030 2d 2d 2d 09 48 65 61 64 65 72 33 2d 2d 2d 2d 2d |--
-.Header3-----|
00000040 2d 2d 2d 2d 0a 52 65 63 31 20 46 69 65 6c 64 31 |---
-.Rec1 Field1|
00000050 09 52 65 63 31 20 46 69 65 6c 64 32 09 52 65 63 |.Rec1
Field2.Rec|
00000060 31 20 46 69 65 6c 64 33 0a 52 65 63 32 20 46 69 |1
Field3.Rec2 Fi|
00000070 65 6c 64 31 09 52 65 63 32 20 46 69 65 6c 64 32
|eld1.Rec2 Field2|
00000080 09 52 65 63 32 20 46 69 65 6c 64 33 0a 52 65 63 |.Rec2
Field3.Rec|
00000090 33 20 46 69 65 6c 64 31 09 52 65 63 33 20 46 69 |3
Field1.Rec3 Fi|
000000a0 65 6c 64 32 09 52 65 63 33 20 46 69 65 6c 64 33
|eld2.Rec3 Field3|
000000b0 0a                                              |.|
000000b1
```

如果手边没有 gawk，也可以使用 Perl，其写法更直观。我们使用了 -n 选项（nonprinting），该选项在循环处理输入时不会自动输出每一行（记录），它会拆解（unpack）读入的每条记录（$_），然后使用制表符将生成的列表中的各个元素连接成一个标量（scalar）。最后，输出记录并在结尾添加换行符。

```
$ perl -ne 'print join("\t", unpack("A18 A32 A16", $_) ) . "\n";'
\
> fixed-length_file
Header1----------- → Header2------------------------ → Header3---
------
Rec1 Field1 → Rec1 Field2 → Rec1 Field3
Rec2 Field1 → Rec2 Field2 → Rec2 Field3
Rec3 Field1 → Rec3 Field2 → Rec3 Field3
$ perl -ne 'print join("\t", unpack("A18 A32 A16", $_) ) . "\n";'
```

```
\
> fixed-length_file |
> hexdump -C
00000000 48 65 61 64 65 72 31 2d 2d 2d 2d 2d 2d 2d 2d 2d |Header1-
--------|
00000010 2d 2d 09 48 65 61 64 65 72 32 2d 2d 2d 2d 2d 2d |-
-.Header2------|
00000020 2d 2d 2d 2d 2d 2d 2d 2d 2d 2d 2d 2d 2d 2d 2d 2d |--------
--------|
00000030 2d 2d 2d 09 48 65 61 64 65 72 33 2d 2d 2d 2d 2d |--
-.Header3-----|
00000040 2d 2d 2d 2d 0a 52 65 63 31 20 46 69 65 6c 64 31 |---
-.Rec1 Field1|
00000050 09 52 65 63 31 20 46 69 65 6c 64 32 09 52 65 63 |.Rec1
Field2.Rec|
00000060 31 20 46 69 65 6c 64 33 0a 52 65 63 32 20 46 69 |1
Field3.Rec2 Fi|
00000070 65 6c 64 31 09 52 65 63 32 20 46 69 65 6c 64 32
|eld1.Rec2 Field2|
00000080 09 52 65 63 32 20 46 69 65 6c 64 33 0a 52 65 63 |.Rec2
Field3.Rec|
00000090 33 20 46 69 65 6c 64 31 09 52 65 63 33 20 46 69 |3
Field1.Rec3 Fi|
000000a0 65 6c 64 32 09 52 65 63 33 20 46 69 65 6c 64 33
|eld2.Rec3 Field3|
000000b0 0a                                              |.|
000000b1
```

有关 pack 和 unpack 所使用的模板格式，参见 Perl 文档。

13.17.3 讨论

因为脑子里时刻都牢记着 textutils 工具链 [5]，任何拥有 Unix 背景的人都会自觉在输出中使用某种分隔符，所以固定长度（或称作固定宽度）的记录在 Unix 世界里极少见到。不过，这种记录在大型机（mainframe）世界倒是司空见惯，它们偶尔会从源自巨型服务器的大型应用程序中蹦出来，比如 SAP 的某些应用程序。我们之前已经见识过了，固定长度记录处理起来没有任何问题。

[5] textutils 即 GNU Text Utilities，是 GNU 操作系统中用于文本处理的一系列基本工具。fileutils、shellutils 以及 textutils 现在已经合并成 GNU Coreutils。——译者注

有一处地方要特别注意，这则实例要求每条记录都以换行符作结。但很多陈旧的大型机记录格式可不是这样，对于这种情况，可以使用 13.18 节中的方法，先在每条记录结尾添加换行符，然后再处理。

13.17.4 讨论

- man gawk
- comp.lang.awk FAQ
- 13.15 节
- 13.18 节

13.18 处理没有换行的文件

13.18.1 问题

你手头有一个比较大的文件，其中没有任何换行，现在需要处理该文件。

13.18.2 解决方案

先对文件进行预处理，在适当的位置添加换行符。例如，OpenOffice 的 ODF 文件基本上就是经过压缩的 XML 文件。可以将其解压，然后用 grep 搜索 XML 文件，我们在撰写本书时没少这么做。有关 ODF 文件的更多处理，参见 12.5 节。本例在每个结尾尖括号（>）之后插入一个换行符。这样一来，使用 grep 或其他 textutils 工具处理该文件时就容易多了。注意，我们必须紧跟着反斜线之后按下回车键，以此在 sed 脚本中嵌入一个经过转义的换行符。

```
$ wc -l content.xml
1 content.xml

$ sed -e 's/>/>\
> /g' content.xml | wc -l
1687
```

如果有不带换行符的固定长度记录，也可以这么处理，其中 48 是记录长度。

```
$ cat fixed-length
Line_1_ _aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZLine_2__
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZLine_3__
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZLine_4__
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZLine_5__
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZLine_6__
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZLine_7__
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZLine_8__
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZLine_9__
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZLine_10_
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZLine_11_
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZLine_12_
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ

$ wc -l fixed-length
  1 fixed-length

$ sed 's/.\{48\}/&\
> /g;' fixed-length
Line_1__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Line_2__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Line_3__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Line_4__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Line_5__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Line_6__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Line_7__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Line_8__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Line_9__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Line_10_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Line_11_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Line_12_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ

$ perl -pe 's/(.{48})/$1\n/g;' fixed-length
Line_1__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Line_2__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Line_3__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Line_4__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Line_5__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Line_6__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Line_7__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Line_8__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Line_9__aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Line_10_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Line_11_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
Line_12_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaZZZ
```

13.18.3 讨论

当人们以编程方式创建输出时，这种情况经常发生，尤其使用现成模块（canned module），而且还是采用 HTML 或 XML 输出时。

注意，sed 的替换功能有一种允许嵌入换行符的怪异写法。在 sed 中，替换运算符右侧（righthand side，RHS）的 & 会被左侧（lefthand side，LHS）的整个正则表达式所匹配到的字符串替换掉，第一行结尾处的 \ 用于转义换行符，否则会出现类似于“sed: -e expression #1, char 11: unterminated ‘s’ ”这样的错误。这是因为 sed 无法将 s/// 右侧的 \n 识别为元字符。

13.18.4 参考

- THE SED FAQ
- Arnold Robbins 所著的 *Effective awk Programming, 4th Edition*（O'Reilly 出版）
- Arnold Robbins 与 Dale Dougherty 合著的 *sed & awk, 2nd Edition*（O'Reilly 出版）
- 12.5 节
- 13.17 节

13.19 将数据文件转换为CSV

13.19.1 问题

你需要将数据文件转换为 CSV（comma separated value，逗号分隔值）文件。

13.19.2 解决方案

使用 awk 将数据文件转换为 CSV 格式。

```
$ awk 'BEGIN { FS="\t"; OFS="\",\"" } { gsub(/"/, "\"\""); $1 =
$1;
```

```
> printf "\"%s\"\n", $0}' tab_delimited
"Line 1","Field 2","Field 3","Field 4","Field 5 with ""internal""
double-quotes"
"Line 2","Field 2","Field 3","Field 4","Field 5 with ""internal""
double-quotes"
"Line 3","Field 2","Field 3","Field 4","Field 5 with ""internal""
double-quotes"
"Line 4","Field 2","Field 3","Field 4","Field 5 with ""internal""
double-quotes"
$
```

也可以用 Perl 实现相同的效果。

```
$ perl -naF'\t' -e 'chomp @F; s/"/""/g for @F; print
q(").join(q(","),@F)
> .qq("\n);' tab_delimited
"Line 1","Field 2","Field 3","Field 4","Field 5 with ""internal""
double-quotes"
"Line 2","Field 2","Field 3","Field 4","Field 5 with ""internal""
double-quotes"
"Line 3","Field 2","Field 3","Field 4","Field 5 with ""internal""
double-quotes"
"Line 4","Field 2","Field 3","Field 4","Field 5 with ""internal""
double-quotes"
$
```

13.19.3 讨论

首先，明确定义什么是 CSV 可不是件容易事。不存在正式的规范，各个厂商都有自己的一套实现。我们实现的版本非常简单，应该在任何地方都管用。我们给所有的字段都加上双引号（有些实现仅为字符串或包含内部逗号的字符串加引号），如果字段内部已经有双引号，我们则在其之外再加上一层引号。

为此，通过指定制表符为字段分隔符并将 OFS 设置为“,”，我们用 awk 分割输入行，这样就可以在每个字段的结尾和下一个字段的起始位置加上引号，同时在两者之间添加逗号。然后使用两个双引号替换掉所有的双引号，接着执行赋值操作，令 awk 使用指定好的 OFS 重建记录（该技巧参见 13.15 节），最后输出带有行首和行尾双引号的记录。我们必须在多处转义双引号，这导致看起来有点杂乱，否则还是挺直观易懂的。

13.19.4 参考

- comp.lang.awk FAQ
- 13.15 节
- 13.20 节

13.20 解析CSV数据文件

13.20.1 问题

需要解析一个 CSV 数据文件。

13.20.2 解决方案

不同于先前将数据文件转换为 CSV，解析 CSV 可不是一件容易事，因为 CSV 的确切定义都是个问题。

可能的解决方案如下所示。

- `sed`
- `awk`
- Perl：Jeffrey E. F. Friedl 所著的《精通正则表达式（第 3 版）》一书中给出了相应的正则表达式。另外，CPAN（Comprehensive Perl Archive Network）中也有各种专用于此的模块。
- 使用电子表格软件（LibreOffice Calc 和 Microsoft Excel 均可）打开 CSV 文件，然后将内容复制粘贴到文本编辑器中，这样应该能得到以制表符分隔的输出，然后就可以轻松处理了。

13.20.3 讨论

如 13.19 节中所言，CSV 还没有正式的规范，再加上数据的多样性，这项任务要比听起来困难得多。

13.20.4 参考

- 13.19 节

第 14 章　编写安全的shell脚本

编写安全的 shell 脚本？！你都能看得到源代码了，还怎么保护 shell 脚本的安全？

任何依赖于藏匿实现细节的系统都是在试图采用**隐晦式安全**（security by obscurity），这纯粹是一厢情愿，完全没什么安全性可言。去问问那些大型软件制造商，他们的源代码是受到严格保护的商业机密，其产品却不断地遭受漏洞攻击，这些攻击者可压根没见过软件的源代码。相比之下，OpenSSH 和 OpenBSD 的源代码是完全开放的，但它们的安全性好得很。

别指望隐晦式安全能够一劳永逸，尽管其某些形式的确可作为额外的安全层。例如，在非标准端口上侦听的守护进程能避开不少所谓的“脚本小子”。但隐晦式安全绝不能是唯一的安全层，因为迟早会有人会发现盖头下藏着的秘密。

Bruce Schneier 说过，安全是一种过程 [1]。安全不是产品，也不是物件或技术，它永无完结之日。随着技术、网络、攻击和防御的演变，安全过程也要与时俱进。那么，这对于编写安全的 shell 脚本意味着什么呢？

> [1] “The Process of Security”，2000 年 4 月发表于 *Information Security*。——译者注

安全的 shell 脚本能够可靠地做好自己的本职工作，仅此而已。它们不会被利用于获取 root 权限，不会意外地执行 rm -rf /，也不会泄露密码之类的信息。它们很稳健，就算出现故障，也是有条不紊。它们可以容忍用户的无心之错并净化用户的所有输入信息。它们会尽量保持简单，只包含清晰易读的代码和文档，不会出现模棱两可的内容。

这听起来就很像那些经过良好设计的稳健程序，难道不是吗？从一开始，安全性就应该成为优良设计过程的一部分，不能等到最后才着

手。本章将重点介绍最常见的安全漏洞和问题，并向你展示其解决方法。

多年来，安全相关的文献数量众多。如果感兴趣，Gene Spafford et al. 所著的 *Practical UNIX & Internet Security, 3rd Edition*（O'Reilly 出版）是一个不错的起点。Nelson H. F. Beebe 与 Arnold Robbins 合著的 *Classic Shell Scripting*（O'Reilly 出版）一书的第 15 章也值得一看。另外还有不少优秀的在线资源，如“A Lab engineer's check list for writing secure Unix code”。

例 14-1 中的清单涵盖了最通用的 shell 安全编程技术，将其集中在一处的优势是，需要时可以作为快速参考或复制成脚本模板。请务必阅读每种技术对应的完整实例，以便充分理解领会。

例 14-1 ch14/security_template

```
#!/usr/bin/env bash
# 实例文件：security_template

# 设置合理的/安全的路径
PATH='/usr/local/bin:/bin:/usr/bin'
# 确保导出$PATH
\export PATH

# 清除所有别名。切记：开头的\用于禁止别名扩展
\unalias -a

# 清除命令路径散列
hash -r

# 将硬性限制设置为0，关闭核心转储（core dumps）
ulimit -H -c 0 --

# 设置合理的/安全的IFS（注意，这里使用的语法仅适用于bash和ksh93，不具备可移植性！）
IFS=$' \t\n'

# 设置并使用合理的/安全的umask变量
# 注意，这不会影响已经在命令行上重定向的文件
# 022产生0755权限，077产生0700权限……
UMASK=022
umask $UMASK
```

```
until [ -n "$temp_dir" -a ! -d "$temp_dir" ]; do
    temp_dir="/tmp/meaningful_prefix.${RANDOM}${RANDOM}${RANDOM}"
done
mkdir -p -m 0700 $temp_dir \
  || (echo "FATAL: Failed to create temp dir '$temp_dir': $?";
exit 100)

# 尽全力清除临时文件
# 注意，一定要先设置好$temp_dir，千万不要修改!
cleanup="rm -rf $temp_dir"
trap "$cleanup" ABRT EXIT HUP INT QUIT
```

14.1 避开常见的安全问题

14.1.1 问题

你希望编写脚本时能够避开常见的安全问题。

14.1.2 解决方案

验证包括交互式输入以及来自配置文件和交互式用法在内的所有外部输入。尤其不要对未经全面检查的输入执行 `eval`。

使用可靠的临时文件，最好是在安全的临时目录中。

确保使用的是信得过的外部可执行文件。

14.1.3 讨论

从某种意义上来说，这则实例仅仅只是涉及了脚本编程和系统安全性的皮毛而已。但同时也涵盖了你可能会碰到的一些最常见的安全问题。

目前，数据验证（或者说缺乏数据验证）是计算机安全中的一大问题。由此引发的缓冲区溢出和数据注入攻击是迄今为止最为常见的漏洞利用方式。bash 倒是没有像 C 一样遭受该问题的困扰，但概念是相通的。在 bash 的世界中，相较于缓冲区溢出，未经验证的输入更

有可能包含 ;rm -rf / 之类的内容。话说回来，两者都不是什么好事。一定要记得验证数据！

竞争条件（race condition）是另外一个大问题，它与攻击者获得未经许可的文件写入能力紧密相关。在没有外部干扰的情况下，两个或更多的独立事件必须以正确的时间和正确的顺序发生时，**竞争条件**就会出现。这往往会使得非特权用户得以读取或写入本无权访问的文件，导致所谓的**提权**（privilege escalation），普通用户因此获得 root 权限。没有安全地使用临时文件是此类攻击中非常普遍的一个成因。要想消除这种攻击媒介，注意使用安全的临时文件，尤其是在安全的临时目录中。

另一种常见的攻击媒介是被植入木马的实用工具。就像特洛伊木马一样，这些工具可能表里不一。典型的例子就是被下过木马的 ls 命令，该命令用起来和真正的 ls 命令无异，但如果是 root 用户使用，那就不一样了。在这种情况下，它会创建一个名为 r00t 的新用户，默认密码由攻击者掌握，然后删除自身。在脚本编程方面，你能采取的最佳措施就是设置安全的 $PATH。在系统方面，包括 FCheck、Tripwire、AIDE 在内的很多相关工具能够帮助你确保系统的完整性。

14.1.4 参考

- tripwire 网站
- AIDE（Advanced Intrusion Detection Environment）

14.2 避免解释器欺骗

14.2.1 问题

你希望能避免某些 setuid root 欺骗攻击。

14.2.2 解决方案

在 shell 结尾添加一个连字符。

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

14.2.3 讨论

脚本的第一行（也称为 shebang 行）告诉内核用哪种解释器处理余下的脚本。内核还会查找指定解释器的单个选项。有些攻击者就利用了这一点，但如果传入参数，攻击就无法得逞了。详见 Unix FAQs 的 4.7 节。

但是，使用 bash 的硬编码路径可能存在移植性问题，详见 15.1 节。

14.2.4 参考

- 14.15 节
- 15.1 节

14.3 设置安全的$PATH

14.3.1 问题

你希望确保使用了安全的路径。

14.3.2 解决方案

在每个脚本的起始处妥善设置好 $PATH。

```
# 设置合理的/安全的路径
PATH='/usr/local/bin:/bin:/usr/bin'
# 确保导出$PATH
export PATH
```

也可以用 getconf 获取能够找到所有标准实用工具的路径（由 POSIX 确保）。

```
export PATH=$(getconf PATH)
```

14.3.3 讨论

第二个示例中存在两处可移植性问题。首先，$() 的可移植性不如 ''（但可读性更好）。其次，export 命令和变量赋值出现在同一行的写法在某些陈旧或怪异的 Unix 和 Linux 版本中行不通。var='foo'; export var 的可移植性优于 export var='foo'。另外注意，只需要使用 export 命令一次就可以将变量导出到子进程。

如果不使用 getconf，那么我们给出的示例对于初学者来说就是一个很好的默认路径，不过你可能得根据所处的特定环境或者需求做一些调整。也可以使用以下这个可移植性略差的版本。

```
export PATH='/usr/local/bin:/bin:/usr/bin'
```

根据安全风险和需求，你还应该考虑使用绝对路径。这往往很麻烦，并且涉及可移植性方面的问题，因为不同的操作系统会将工具放在不同的位置。使用变量可以在某种程度上缓解这类问题。如果选择这种方法，记得对变量进行排序，省得浏览未排序的变量列表时发生遗漏，从而导致相同的命令多次重复出现。参见例 14-2。

例 14-2 ch14/finding_tools

```
#!/usr/bin/env bash
# 实例文件：finding_tools

# 根据所执行的操作决定是否需要导出

# 放心，相当安全
_cp='/bin/cp'
_mv='/bin/mv'
_rm='/bin/rm'

# 下面这些有点棘手
case $(/bin/uname) in
    'Linux')
        _cut='/bin/cut'
        _nice='/bin/nice'
```

```
        # [...]
    ;;
    'SunOS')
        _cut='/usr/bin/cut'
        _nice='/usr/bin/nice'
        # [...]
    ;;
    # [...]
esac
```

这种方法的另一个优势是，很容易就能看出你的脚本究竟依赖哪些工具，你甚至可以添加一个简单的函数，以便在脚本正式执行操作前确保各种工具可用且可执行。

小心所用的变量名。有些程序（如 InfoZip）使用环境变量（如 `$ZIP` 和 `$UNZIP`）向自身传递设置，如果使用了 `ZIP='/usr/bin/zip'`，那就等着抓耳挠腮吧，苦思冥想为什么在命令行上还好好的，但到了脚本里就不行了。相信我们，这些经验来之不易。

14.3.4 参考

- 6.14 节
- 6.15 节
- 14.9 节
- 14.10 节
- 15.2 节
- 16.4 节
- 16.5 节
- 19.3 节
- A.4 节
- A.5 节

14.4 清除所有的别名

14.4.1 问题

安全起见，你需要确保环境中不存在恶意别名。

14.4.2 解决方案

用 \unalias -a 命令清除所有的已有别名。

14.4.3 讨论

如果能够哄骗 root 或其他用户执行命令，那么攻击者就可以获得本不该拥有的数据或权限。哄骗用户执行恶意程序的其中一种方法是创建一个常用程序（如 ls）的别名。

开头的 \ 能够阻止别名扩展，这一点非常重要，要是少了它，你可能会被欺骗，如下所示。

```
$ alias unalias=echo

$ alias builtin=ls

$ builtin unalias vi
ls: unalias: No such file or directory
ls: vi: No such file or directory

$ unalias -a
-a
```

正如 Chet 所说：“这是一个棘手的问题。”

因为 shell 在搜索命令时会先查找 shell 函数，然后才轮到内建命令，而且还允许在环境中导出函数，所以有必要在每个内建命令前使用 builtin，用 command 跳过函数查找，或者用 unset 删除可能涉及的所有函数。

```
_OLD=$POSIXLY_CORRECT; POSIXLY_CORRECT=1
\unset -f builtin command unset
POSIXLY_CORRECT=$_OLD ; \unset _OLD
builtin unalias command builtin unset
unset -f $(command declare -F \
 | command sed 's/^declare -f //')
```

也可以考虑禁用 expand_aliases 选项，要是这样的话，你还得对 shopt 使用 unset/unalias。

——Chet Ramey[2]

[2]Chet Ramey 目前负责维护 GNU Bash shell 和 GNU Readline。——译者注

14.4.4 参考

- 10.7 节
- 10.8 节
- 16.8 节

14.5 清除命令散列

14.5.1 问题

你需要确保命令散列没有遭到破坏。

14.5.2 解决方案

使用 `hash -r` 命令来清除命令散列中的条目。

14.5.3 讨论

在执行过程中，bash 会“记住”位于 `$PATH` 中的大部分命令的位置，以此加速随后的命令调用。

如果能够哄骗 root 或其他用户执行命令，那么攻击者就可以获得本不该拥有的数据或权限。哄骗用户执行恶意程序的另一种方法是向命令散列“投毒”，从而令用户执行错误的程序。

14.5.4 参考

- 14.9 节
- 14.10 节
- 15.2 节

- 16.4 节
- 16.5 节
- 19.3 节

14.6 防止核心转储

14.6.1 问题

你想防止脚本在出现不可恢复的错误时转储核心，因为核心转储可能包含内存中的敏感数据（如密码）。

14.6.2 解决方案

用 bash 的内建命令 ulimit 将核心文件大小设置为 0，通常可以将该命令放置在 .bashrc 文件中。

```
ulimit -H -c 0 --
```

14.6.3 讨论

核心转储的目的在于调试，其中包含进程出现故障时涉及的内存映像。这样一来，转储文件将包含该进程保存在内存中的所有内容（如用户输入的密码）。

如果不想用户对其做出改动，可以将该命令放进用户无权写入的系统级文件，如 /etc/profile 或 /etc/bashrc 中。

14.6.4 参考

- help ulimit

14.7 设置安全的$IFS

14.7.1 问题

你想要确保环境变量 $IFS 没有问题。

14.7.2 解决方案

使用下列明晰（但不兼容 POSIX）的语法在每个脚本的开头将其设置为已知的良好状态。

```
# 设置合理的/安全的IFS（注意，这里使用的语法仅适用于bash和ksh93，不具备可移植性！）
IFS=$' \t\n'
```

14.7.3 讨论

如我们所言，这种语法不具备可移植性。但是，规范的可移植语法又不可靠，因为容易被一些会修剪空白字符的编辑器给不小心剥离掉。传统的 $IFS 值是空格、制表符、换行符，它们的顺序很重要。$*（返回所有的位置参数）、特殊的 ${!prefix@} 和 ${!prefix*} 参数扩展，以及可编程命令全都使用 $IFS 的第一个值作为分隔符。

典型的写法是在第一行上留下空格和制表符（用点号和箭头表示）。

```
1 IFS=' • → ¶
2 '
```

换行符、空格、制表符不大可能被修剪，但如果修改了默认顺序，则可能会给某些程序带来意料之外的结果。

```
1 IFS=' ¶
2 • → '
```

14.7.4 参考

- 13.15 节

14.8 设置安全的umask

14.8.1 问题

你想要确保自己使用的 umask 是安全的。

14.8.2 解决方案

在每个脚本的开头使用 bash 的内建命令 umask 将掩码值设置为已知的良好状态。

```
# 设置并使用合理的/安全的umask变量
# 注意，这不会影响已经在命令行上重定向的文件
# 022产生0775权限，077产生0700权限……
UMASK=022
umask $UMASK
```

14.8.3 讨论

我们设置了 $UMASK 变量，以防需要在脚本的其他地方使用不同的掩码。你也可以跳过这一步来执行后续操作，这也没什么大不了的。

```
umask 002
```

记住，umask 是掩码，指定了要从默认的文件权限（666）和目录权限（777）中去除的位。如果对此有疑问，可以测试一下。

```
# 运行一个新shell，避免影响当前环境
/tmp$ bash

# 检查当前设置
/tmp$ touch um_current

# 检查其他设置
/tmp$ umask 000 ; touch um_000
/tmp$ umask 022 ; touch um_022
/tmp$ umask 077 ; touch um_077
```

```
/tmp$ ls -l um_*
-rw-rw-rw-   1 jp   jp  0 Jul 22 06:05 um000
-rw-r--r--   1 jp   jp  0 Jul 22 06:05 um022
-rw-------   1 jp   jp  0 Jul 22 06:05 um077
-rw-rw-r--   1 jp   jp  0 Jul 22 06:05 umcurrent

# 清理并退出子shell
/tmp$ rm um_*
/tmp$ exit
```

14.8.4 参考

- help umask
- Controlling file permissions with mask

14.9 在$PATH中查找人皆可写的目录

14.9.1 问题

你想要确定 root 用户的 $PATH 中不存在人皆可写的目录。（至于为什么要这么做，参见 14.10 节。）

14.9.2 解决方案

使用例 14-3 中的简单脚本来检查你的 $PATH。配合 su 或 sudo 检查其他用户的路径。

例 14-3 ch14/chkpath.1

```
#!/usr/bin/env bash
# 实例文件：chkpath.1
# 检查$PATH，查找人皆可写的目录或丢失的目录

exit_code=0

for dir in ${PATH//:/ }; do
    [ -L "$dir" ] && printf "%b" "symlink, "
    if [ ! -d "$dir" ]; then
        printf "%b" "missing\t\t"
```

```
            (( exit_code++ ))
    elif [ -n "$(ls -lLd $dir | grep '^d.......w. ')" ]; then
            printf "%b" "world writable\t"
            (( exit_code++ ))
    else
            printf "%b" "ok\t\t"
    fi
    printf "%b" "$dir\n"
done
exit $exit_code
```

例如：

```
# ./chkpath
ok              /usr/local/sbin
ok              /usr/local/bin
ok              /sbin
ok              /bin
ok              /usr/sbin
ok              /usr/bin
ok              /usr/X11R6/bin
ok              /root/bin
missing         /does_not_exist
world writable  /tmp
symlink, world writable /tmp/bin
symlink, ok /root/sbin
#
```

14.9.3 讨论

我们用 9.11 节中讲过的技术将 $PATH 转换成以空格分隔的列表、测试符号链接（-L），确保目录的确存在（-d）。然后获得长格式的目录列表（-l）、解引用符号链接（-L），并仅列出目录名称（-d）。最后，使用 grep 过滤出人皆可写的目录。

如你所见，我们隔开了正常的目录，而有问题的目录看起来可能有点杂乱。另外，这个脚本也没有遵循 Unix 工具“没事就不吭声”的规则，因为我们觉得这是一个不错的机会，可以看看路径中究竟都有些什么，还能在自动检查之外再浏览一遍。

如果没有在 $PATH 中发现问题，则返回退出码 0 以示成功；如果存在问题，则返回错误数量。稍作调整，如例 14-4 所示，我们可以

将文件的权限模式、所有者和组添加到输出中，这可能更有助于检查。

例 14-4 ch14/chkpath.2

```
#!/usr/bin/env bash
# 实例文件：chkpath.2
# 使用stat命令检查$PATH，查找人皆可写的目录或丢失的目录

exit_code=0

for dir in ${PATH//:/ }; do
    [ -L "$dir" ] && printf "%b" "symlink, "
    if [ ! -d "$dir" ]; then
        printf "%b" "missing\t\t\t\t"
        (( exit_code++ ))
    else
        stat=$(ls -lHd $dir | awk '{print $1, $3, $4}')
        if [ -n "$(echo $stat | grep '^d.......w. ')" ]; then
            printf "%b" "world writable\t$stat "
            (( exit_code++ ))
        else
            printf "%b" "ok\t\t$stat "
        fi
    fi
    printf "%b" "$dir\n"

done
exit $exit_code
```

例如：

```
# ./chkpath ; echo $?
ok              drwxr-xr-x root root /usr/local/sbin
ok              drwxr-xr-x root root /usr/local/bin
ok              drwxr-xr-x root root /sbin
ok              drwxr-xr-x root root /bin
ok              drwxr-xr-x root root /usr/sbin
ok              drwxr-xr-x root root /usr/bin
ok              drwxr-xr-x root root /usr/X11R6/bin
ok              drwx------ root root /root/bin
missing                              /does_not_exist
world writable  drwxrwxrwt root root /tmp
symlink, ok             drwxr-xr-x root root /root/sbin
2
#
```

14.9.4 参考

- 9.11 节
- 14.10 节
- 15.2 节
- 16.4 节
- 16.5 节
- 19.3 节

14.10 将当前目录加入$PATH

14.10.1 问题

运行脚本时必须输入 ./script（起始的点号和斜线都不能少）实在是乏味，你想将 .（或者是空目录，这意味着起始或结尾处会出现 :，要么中间出现 ::）加入 $PATH。

14.10.2 解决方案

我们不建议任何用户选择这种做法，更是强烈反对 root 这么做。如果你非做不可，要确保把 . 放在最后。root 绝不要效仿。

14.10.3 讨论

如你所知，当输入不包含路径的命令名时，shell 会搜索 $PATH 中列出的目录。不添加 . 的原因和不允许 $PATH 中出现人皆可读的目录一样（具体做法参见 14.9 节）。

假设你位于 /tmp，$PATH 中的第一项就是 .。如果输入 ls，恰好 /tmp 中也有一个同名文件（/tmp/ls），那么执行的就是该文件，而不是你原本打算执行的 /bin/ls。那又如何？嗯，这得视具体情况而定。/tmp/ls 也许（从名字上看，很可能）是一个恶意脚本，如果以 root 身份运行，在其销毁证据、删除自身之前，你根本不知道它都做了些什么。

将 . 作为 $PATH 的最后一项呢？你有没有将 mv 误输成 mc ？我们这么干过。除非系统已经安装 Midnight Commander，否则你很可能会在本打算执行 /bin/mv 时意外地执行了 ./mc，结果就和刚才描述过的一样。

压根别考虑加入点号就行了！

14.10.4 参考

- Unix FAQs 的 2.13 节
- 9.11 节
- 14.3 节
- 14.9 节
- 15.2 节
- 16.4 节
- 16.5 节
- 19.3 节

14.11 使用安全的临时文件

14.11.1 问题

你需要创建临时文件或目录，但又担心使用可预测的名称会带来安全隐患。

14.11.2 解决方案

尝试用 echo "~$TMPDIR~" 查看你所在的系统有没有提供安全的临时目录。为了在该变量未设置时能够清晰地识别出，我们加入了两个 ~。

简单易行、具备可移植性，且在通常情况下够用的解决方案是在脚本中使用行内 $RANDOM。例如：

```
# 确保设置好$TMP
[ -n "$TMP" ] || TMP='/tmp'

# 创建一个"足够用的"随机临时目录
until [ -n "$temp_dir" -a ! -d "$temp_dir" ]; do
    temp_dir="/$TMP/meaningful_prefix.${RANDOM}${RANDOM}${RANDOM}"
done
mkdir -p -m 0700 $temp_dir
  || { echo "FATAL: Failed to create temp dir '$temp_dir': $?";
exit 100 }
  # 创建一个"足够用的"随机临时文件
  until [ -n "$temp_file" -a ! -e "$temp_file" ]; do

temp_file="/$TMP/meaningful_prefix.${RANDOM}${RANDOM}${RANDOM}"
done
touch $temp_file && chmod 0600 $temp_file
  || { echo "FATAL: Failed to create temp file '$temp_file': $?";
exit 101 }
```

同时使用随机临时目录和随机文件名的效果会更好，如例 14-5 所示。

例 14-5 ch14/make_temp

```
# 实例文件：make_temp

# 确保设置好$TMP
[ -n "$TMP" ] || TMP='/tmp'

# 创建一个"足够用的"随机临时目录
until [ -n "$temp_dir" -a ! -d "$temp_dir" ]; do
    temp_dir="/$TMP/meaningful_prefix.${RANDOM}${RANDOM}${RANDOM}"

done
mkdir -p -m 0700 $temp_dir \
  || { echo "FATAL: Failed to create temp dir '$temp_dir': $?";
exit 100; }

# 在临时目录中创建一个"足够用的"随机临时文件
temp_file="$temp_dir/meaningful_prefix.${RANDOM}${RANDOM}${RANDOM}
"
touch $temp_file && chmod 0600 $temp_file \
  || { echo "FATAL: Failed to create temp file '$temp_file': $?";
exit 101; }
```

不管采用哪种方法，不要忘记设置陷阱（trap）来完成清理工作（参见例 14-6）。如前所述，必须在声明陷阱前先设置好 $temp_dir，且不得更改。如果实际情况并非如此，则重写代码逻辑以满足需求。

例 14-6 ch14/clean_temp

```
# 实例文件：clean_temp

# 尽力清理所有的临时文件
# 注意，$temp_dir必须先设置好，且不得更改
cleanup="rm -rf $temp_dir"
trap "$cleanup" ABRT EXIT HUP INT QUIT
```

> $RANDOM 在 dash 中不可用，在某些 Linux 发行版中，/bin/sh 指向的就是 dash。尤其要注意的是，Debian 和 Ubuntu 目前使用的都是 dash，因为相较于 bash，dash 的体积更小，运行速度更快，有助于加快系统引导速度。但这意味着，作为指向 bash 的符号链接 /bin/sh 现在指向的是 dash，各种 bash 独有的特性也无法使用了。

14.11.3 讨论

bash 2.0 已开始提供 $RANDOM，单使用它可能已经足够了。简单的代码比复杂的代码更有助于且更易于实现安全性。因此，相较于处理 mktemp 或 /dev/urandom 复杂的验证和错误检查，$RANDOM 也许会令代码更加安全。由于用法如此简单，你指不定还会对它倍加青睐。但是，$RANDOM 仅包含数字，mktemp 则包含数字和大小写字母，urandom 包含数字和小写字母，因此大大增加了密钥空间。

在临时目录中创建临时文件具有下列优势。

- mkdir -p -m 0700 $temp_dir 避免了 touch $temp_file && chmod 0600 $temp_file 所带来的竞争条件。
- 如果为目录设置了 0700 权限，那么在其中创建的文件对于该目录之外的非 root 攻击者是不可见的。
- 临时目录更容易确保退出时删除所有的临时文件。如果你的临时文件四处散落，清理时迟早会漏掉一两个。

- 你可以为目录中的临时文件选择有意义的名称，这样有助于开发和调试，从而提高脚本的安全性和稳健性。
- 在路径中使用有意义的前缀能让人清楚运行的是什么脚本（这种做法或好或坏，不过 ps 或 /proc 也是这么做的）。更重要的是，这能够凸显（highlight）无法完成清理工作的脚本，否则可能会造成信息泄露。

例 14-5 建议在生成的路径中使用 *meaningful_prefix*。有些人断然认为这种可预测性会影响安全。没错，部分路径的确能够被预测出来，但我们还是觉得利大于弊。如果你仍持反对意见，把有意义的前缀去掉就行了。

根据风险及安全需求，你可能也想像我们一样在随机临时目录中使用随机临时文件。这种做法未必能对安全性带来什么实质性的提高，但如果你觉得还不错，那就用吧。

前面提过 `touch $temp_file&&chmod 0600$temp_file` 存在竞争条件。解决方法之一是：

```
saved_umask=$(umask)
umask 077
touch $temp_file
umask $saved_umask
unset saved_umask
```

我们推荐采用随机临时目录和随机（或半随机）文件，因为这在整体上提供了更多好处。

如果 $RANDOM 的纯数字性质着实让你不爽，可以考虑将一些其他来源的伪不可预测数据和伪随机数据与散列函数结合起来。

```
nice_long_random_string=$( (last ; who ; netstat -a ; free ; date
\
 ; echo $RANDOM) | md5sum | cut -d' ' -f1 )
```

我们不建议使用这里给出的后备方案，因为额外的复杂性可能反而会帮倒忙。不过倒是可以借机看看什么叫没事找事。

理论上，更安全的方法是使用很多现代系统中都配备的实用工具 mktemp，选择 /dev/urandom（很多现代系统中也有），甚至是 $RANDOM 作为后备方案。问题在于 mktemp 和 /dev/urandom 未必总是可用，在实践中，兼顾可移植性的解决方法要比我们给出的复杂得多。例 14-7 展示了一种方法，但有可能的话，最好还是简单点，别搞那么复杂。

例 14-7 ch14/MakeTemp

```
# 实例文件：MakeTemp
# 该函数可以被其他脚本纳入（incorporate）或读入（source into）
#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# 尝试创建安全的临时文件或目录
# 调用方法类似于：$temp_file=$(MakeTemp <file|dir> [path/to/name-
prefix])
# 在$TEMP_NAME中返回安全的临时文件名或目录名
# For example:

#         $temp_dir=$(MakeTemp dir /tmp/$PROGRAM.foo)
#         $temp_file=$(MakeTemp file /tmp/$PROGRAM.foo)
#
function MakeTemp {
    # 确保设置好$TMP
    [ -n "$TMP" ] || TMP='/tmp'

    local type_name=$1
    local prefix=${2:-$TMP/temp} # 除非已经定义好前缀，否则使用 $TMP +
temp
    local temp_type=''
    local sanity_check=''

    case $type_name in
        file )
            temp_type=''
            ur_cmd='touch'
            #                   -f 普通文件       -r 可读
            #                   -w 可写           -O 属于自己
            sanity_check='test -f $TEMP_NAME -a -r $TEMP_NAME \
                           -a -w $TEMP_NAME -a -O $TEMP_NAME'
            ;;
        dir|directory )
            temp_type='-d'
            ur_cmd='mkdir -p -m0700'
            #                   -d 目录           -r 可读
            #                   -w 可写           -x 可搜索         -O
```

```
属于自己
            sanity_check='test -d $TEMP_NAME -a -r $TEMP_NAME \
                          -a -w $TEMP_NAME -a -x $TEMP_NAME -a -
O $TEMP_NAME'
            ;;
        * ) Error "\nBad type in $PROGRAM:MakeTemp! Needs
file|dir." 1 ;;
    esac

    # 先尝试mktemp
    TEMP_NAME=$(mktemp $temp_type ${prefix}.XXXXXXXXX)

    # 如果失败，再尝试urandom；如果还失败，则放弃
    if [ -z "$TEMP_NAME" ]; then
        TEMP_NAME="${prefix}.$(cat /dev/urandom | od -x | tr -d '
' | head -1)"
        $ur_cmd $TEMP_NAME
    fi
    # 确保的确创建了文件或目录，否则结束
    if ! eval $sanity_check; then
        Error \
        "\aFATAL ERROR: can't make temp $type_name with
'$0:MakeTemp$*'!\n" 2
    else
        echo "$TEMP_NAME"
    fi

} # 函数MakeTemp定义完毕
```

14.11.4 参考

- man mktemp
- 14.13 节
- 15.3 节
- 维基百科（Almquist shell）
- 附录 B，尤其是 ./scripts.noah/mktmp.bash

14.12 验证输入

14.12.1 问题

你需要获得输入（例如，来自用户或程序），同时还要确保安全性或数据的完整性。

14.12.2 解决方案

根据输入内容以及所要求的严格程度，验证输入的方式远不止一种。

对于简单的“要么匹配，要么不匹配”的情况，使用模式匹配即可（参见 6.6~6.8 节）。

```
[[ "$raw_input" == *.jpg ]] && echo "Got a JPEG file."
```

如果有效数据不止一种，可以使用 case 语句（参见 6.14 和 6.15 节），如例 14-8 所示。

例 14-8 ch14/validate_using_case

```
# 实例文件：validate_using_case

case $raw_input in
    *.company.com         ) # 可能是本地主机名
        ;;
    *.jpg                 ) # 可能是JPEG文件
        ;;
    *.[jJ][pP][gG]        ) # 可能是JPEG文件（不区分大小写）
        ;;
    foo | bar             ) # 输入'foo'或'bar
        ;;
    [0-9][0-9][0-9]       ) # 3位数字
        ;;
    [a-z][a-z][a-z][a-z] ) # 4个小写字母组成的单词
        ;;
    *                     ) # 以上都不是
        ;;
esac
```

如果模式匹配不够精确，而且你使用的是 3.0 以上的 bash 版本，那么可以选择使用正则表达式（参见 6.8 节）。以下示例查找文件名为 3~6 个字母数字字符，扩展名为 .jpg 的文件（区分大小写）。

```
[[ "$raw_input" =~ [[:alpha:]]{3,6}\.jpg ]] && echo "Got a JPEG
file."
```

14.12.3 讨论

要想了解规模更大、更详细的示例，参见 bash tarball 新近版本中的 examples/scripts/shprompt。注意，这个例子可是由负责 bash 维护工作的 Chet Ramey 亲自编写的。

```
# shprompt -- give a prompt and get an answer satisfying certain
criteria
#
# shprompt [-dDfFsy] prompt
#    s = prompt for string
#    f = prompt for filename
#    F = prompt for full pathname to a file or directory
#    d = prompt for a directory name
#    D = prompt for a full pathname to a directory
#    y = prompt for y or n answer
#
# Chet Ramey
# chet@ins.CWRU.Edu
```

类似的示例可参见 examples/scripts.noah/y_or_n_p.bash，该脚本由 Noah Friedman 于 1993 年编写，后经 Chet Ramey 转换为 bash 2 的语法。另外，也可以参考 ./functions/isnum2 和 ./functions/isvalidip 中的示例。

14.12.4 参考

- 3.5 节
- 3.6 节
- 3.7 节
- 3.8 节
- 6.6 节
- 6.7 节
- 6.8 节
- 6.14 节
- 6.15 节
- 11.2 节

- 13.6 节
- 13.7 节
- 附录 B

14.13 设置权限

14.13.1 问题

你希望以安全的方式设置权限。

14.13.2 解决方案

如果出于安全原因需要设置确切的权限（或者说你确定不在乎先前的权限设定，就是要对其进行改动），可以使用带有 4 位八进制数模式的 chmod。

```
chmod 0755 some_script
```

如果只是想添加或删除权限，同时还要保留原先设置的权限，可以在符号模式中使用 + 或 - 操作。

```
chmod +x some_script
```

如果试图使用类似于 chmod -R 0644 *some_directory* 的命令递归设置目录中所有文件的权限，那你会后悔的，因为你将所有的子目录都设置成了不可执行，这意味着无法访问目录中的内容，无法使用 cd 进入，也无法遍历其中。可以使用 find 和 xargs 配合 chmod 来分别设置文件和目录的权限。对于文件：

```
find some_directory -type f -print0 | xargs -0 chmod 0644
```

对于目录：

```
find some_directory -type d -print0 | xargs -0 chmod 0755
```

当然了，如果只是想设置单个目录中所有文件的权限（非递归设置），切换到该目录并设置即可。

创建目录时使用 mkdir -m *mode new_directory*，这样不仅可以用一个命令完成两项任务，还能够避免创建目录和设置权限之间可能存在的竞争条件。

14.13.3 讨论

很多人习惯使用 3 位八进制数模式，但我们喜欢将 4 位八进制数全部写出来，以明确所有的权限。在可能的情况下，我们也偏好使用八进制模式，因为它能够清晰地表达出要设置的权限。你也可以使用符号模式中的绝对赋权操作（=），但作为传统主义者，我们还是最爱老派的八进制模式。

使用符号模式的 + 或 - 操作时，确保最终的权限比较麻烦，因为这种设置方式是相对的，而非绝对的。问题是，在很多情况下，你无法使用**八进制模式**将现有权限替换了之。此时你只能指靠符号模式，在不影响现有权限的同时用 + 添加权限。具体细节请查询所在系统的 chmod，并核实结果是否符合预期。以下是几个例子。

```
$ ls -l
-rw-r--r--1 jp users 0 Dec 1 02:09 script.sh
$

# 使用八进制写法令文件对属主可读、可写、可执行
$ chmod 0700 script.sh

$ ls -l
-rwx------1 jp users 0 Dec 1 02:09 script.sh

# 使用符号写法令文件对所有人可读、可执行
$ chmod ugo+rx *.sh

$ ls -l
-rwxr-xr-x 1 jp users 0 Dec 1 02:09 script.sh
```

注意，在最后的示例中，虽然我们为所有人（ugo）添加了（+）rx 权限，但属主仍旧保留了写入（w）权限。这正是我们想要的效果，也是经常要面对的场景。你有没有发现，这种方式很容易出错，一不

小心就会造成不当的权限设置，招惹麻烦。这就是为什么我们喜欢尽可能使用绝对赋权形式的八进制模式，当然，我们也从不会忘记检查命令结果。

不管怎样，在批量调整文件权限之前，记得全面测试命令。如果你还想备份文件的权限和属主，详见 17.8 节。

14.13.4 参考

- man chmod
- man find
- man xargs
- 9.2 节
- 17.8 节

14.14 密码被泄露到进程列表

14.14.1 问题

ps 可能会将你在命令行上输入的密码一清二楚地显示出来。例如：

```
$ ./cheesy_app -u user -p password &
[1] 13301

$ ps
  PID TT STAT     TIME COMMAND
 5280 p0 S    0:00.08 -bash
 9784 p0 R+   0:00.00 ps
13301 p0 S   0:00.01 /bin/sh ./cheesy_app -u user -p password
```

14.14.2 解决方案

尽可能不要在命令行上使用密码。

14.14.3 讨论

说真的，别这么干。

如果没有根据要求输入密码，很多提供了 -p 或类似选项的应用程序会提示用户。对于交互式用法，这当然是好事，但在脚本中就未必如此了。也许你想编写一个普通的“包装器”脚本或别名，尝试在命令行上封装密码。遗憾的是，这种做法无济于事，因为命令最终还是要运行并出现在进程列表中。如果命令能从 STDIN 接受密码，则可以像下面这样做。

```
./bad_app < ~/.hidden/bad_apps_password
```

虽然还会产生其他问题，但至少不会在进程列表中显示出密码了。

如果还不行，要么换一个程序或给现有的程序打补丁，要么就凑合着用吧。

14.14.4 参考

- 3.8 节
- 14.20 节

14.15 编写setuid或setgid脚本

14.15.1 问题

你认为可以通过设置 shell 脚本的 setuid 或 setgid 位来解决碰到的问题。

14.15.2 解决方案

使用 Unix 组、文件权限或 sudo 为适合的用户授予完成任务所需要的最小权限。

启用 shell 脚本的 setuid 或 setguid 位会产生更多的问题，尤其安全方面会得不偿失。有些系统（如 Linux）甚至就没把 shell 脚本

的 setuid 位当回事，因此，除了安全风险外，创建 setuid shell 脚本反倒会产生不必要的可移植性问题。

14.15.3 讨论

setuid root 脚本尤其危险，压根就别考虑。使用 sudo 吧。

setuid 和 setgid 应用于目录的含义与应用于可执行文件的含义不同。如果在目录上设置，则会使得在其中新创建的文件或子目录归属于该目录的属主或属组。

注意，可以分别用 test -u 和 test -g 测试是否设置了 setuid 和 setgid。

```
$ mkdir suid_dir sgid_dir

$ touch suid_file sgid_file

$ ls -l
total 4
drwxr-xr-x 2 jp users 512 Dec 9 03:45 sgid_dir
-rw-r--r-- 1 jp users   0 Dec 9 03:45 sgid_file
drwxr-xr-x 2 jp users 512 Dec 9 03:45 suid_dir
-rw-r--r-- 1 jp users   0 Dec 9 03:45 suid_file

$ chmod 4755 suid_dir suid_file

$ chmod 2755 sgid_dir sgid_file

$ ls -l
total 4
drwxr-sr-x 2 jp users 512 Dec 9 03:45 sgid_dir
-rwxr-sr-x 1 jp users   0 Dec 9 03:45 sgid_file
drwsr-xr-x 2 jp users 512 Dec 9 03:45 suid_dir
-rwsr-xr-x 1 jp users   0 Dec 9 03:45 suid_file

$ [ -u suid_dir ] && echo 'Yup, suid' || echo 'Nope, not suid'
Yup, suid

$ [ -u sgid_dir ] && echo 'Yup, suid' || echo 'Nope, not suid'
Nope, not suid

$ [ -g sgid_file ] && echo 'Yup, sgid' || echo 'Nope, not sgid'
Yup, sgid
```

```
$ [ -g suid_file ] && echo 'Yup, sgid' || echo 'Nope, not sgid'
Nope, not sgid
```

14.15.4 参考

- man chmod
- 14.18 节
- 14.19 节
- 14.20 节
- 17.15 节

14.16 限制访客

本节中有关受限 shell 的内容选自 Cameron Newham 所著的 *Learning the bash Shell, 3rd Edition*（O'Reilly 出版）。

14.16.1 问题

你需要在系统中启用一些访客用户，同时限制这些用户的行为。

14.16.2 解决方案

尽可能避免使用共享账户，因为如果用户一走了之，你将无从查起，而且会给组织惹来麻烦，还得修改密码并通知其他用户。可以创建独立账户，为其赋予完成任务所需要的最小权限。考虑使用下列方法：

- chroot 囚牢（chroot jail），参见 14.17 节
- SSH，允许非交互式访问命令或资源，参见 14.21 节
- bash 的受限 shell

14.16.3 讨论

受限 shell 的作用是将用户置于移动能力和文件写入能力都受到极大限制的环境之中。通常应用于访客账户。通过将 rbash（如果编译

bash 时加入了该选项）放入用户在 /etc/passwd 文件中对应的条目，你可以将该用户的登录 shell 设置为受限 shell。

受限 shell 施加的特定限制不允许用户执行以下操作。

- 修改工作目录。cd 命令不可用。如果尝试使用，你会得到错误消息 cd:restricted from bash。
- 将输出重定向到文件。重定向操作符 >、>|、<>、>> 均不可用。
- 为环境变量 $ENV、$BASH_ENV、$SHELL、$PATH 赋值。
- 指定含有斜线（/）的命令。任何位于当前目录之外的文件都会被 shell 视为“未找到（not found）”。
- 使用内建命令 exec。
- 指定含有 / 的文件名作为内建命令 .（source）的参数。
- 启动时从 shell 环境中导入函数定义。
- 使用内建命令 enable 的 -f 和 -d 选项来添加或删除内建命令。
- 使用内建命令 command 的 -p 选项。
- 使用 set +r 关闭受限模式。

这些限制在用户的 .bash_profile 和环境文件运行之后生效。除此之外，最好将用户的 .bash_profile 和 .bashrc 文件属主更改为 root，同时将这些文件设置为只读。用户的主目录也应该是只读的。

这意味着受限 shell 用户的整个环境都是在 /etc/profile 和 .bash_profile 中设置的。因为用户既无法访问 /etc/profile，也覆盖不了 .bash_profile，所以只能让系统管理员根据需要配置环境。可以将启动文件中的最后一个命令设置为其他目录（通常是用户主目录下的某个子目录）的 cd 命令，以此多一层保护，这种做法也不错。

设置此类环境的常见方式有两种：建立一个包含安全命令的目录并使其成为 $PATH 中唯一的目录；设置一个命令选单，用户只能从中选择，除非退出 shell。

受限 shell 也抵挡不住铁了心的攻击者。你很难一厢情愿地将用户锁定在特定环境中，因为很多常用软件（如 vi 和 Emacs）

具备 shell escape 功能 [3]，有可能完全绕过受限 shell。

如果善加利用，受限 shell 会成为重要的附加安全层，但不应该将其作为唯一的安全层。

[3]有关 shell escape 的详细信息，参见 *Unix Power Tools, 2nd Edition* 一书的 30.26 节（O'Reilly 出版）。——译者注

注意，最初的 Bourne shell 有一个叫作 rsh 的受限版本，这可能会和所谓的 r-tools（rsh、rcp、rlogin 等）中的远程 shell 程序（也叫作 rsh）搞混。rsh 极不安全，基本上已经被 SSH（Secure Shell，安全 shell）替代了（我们诚挚地希望如此）。

14.16.4 参考

- 14.17 节
- 14.21 节

14.17 使用chroot囚牢

14.17.1 问题

你不得不使用一个并不信任的脚本或程序。

14.17.2 解决方案

可以考虑将其放入 chroot 囚牢。chroot 命令能够将当前进程的根目录更改为你所指定的目录，然后返回一个 shell 或调用 exec 来执行特定的命令。这就产生了将进程（程序）投入囚牢的效果，从理论上来说，该进程是无法从中逃脱到父目录的。因此，如果程序遭到损坏或者从事恶意活动，那它也只能影响到所局限于的那一小部分文件系统。如果再以权限极为有限的用户身份运行，这可以成为非常有用的额外安全层。

遗憾的是，涵盖 chroot 方方面面的细节超出了本实例的范围，可能需要单独一本书。这里我们只是想要提高你对相关功能的认识。

14.17.3 讨论

那干吗不把一切都放进 chroot 囚牢中运行呢？因为很多程序还得跟文件系统中的其他程序、文件、目录或套接字打交道。这正是使用 chroot 囚牢过程中棘手的地方。由于程序接触不到囚牢以外的世界，所需要的一切都必须存在于囚牢之内。程序越复杂，就越难以在囚牢中运行。

有些应用程序生就属于 Internet，比如 DNS（如 BIND）、Web、邮件服务器（如 Postfix），这类应用程序有可能通过难度不一的配置运行于 chroot 囚牢之中。具体细节参见发行版和具体应用程序的文档。

另一种值得留意的 chroot 应用是在系统恢复过程中。从 LiveCD 引导并挂载好根文件后，你可能需要运行如 LILO 或 GRUB 等工具，根据配置，这类工具需要认为自己的确运行在受损系统上。如果 LiveCD 和安装的系统没有太大不同，通常可以使用 chroot 将受损系统的挂载点设置为根目录并进行修复。该方法是可行的，因为所有的工具、库、配置文件以及设备文件都已经存在于囚牢之内，这已经是一个完整的（如果不算很正常的话）系统了。执行 chroot 之后，你可能得检查一下 $PATH，以便查找需要使用的命令（这就是为什么先前要提醒“如果 LiveCD 和安装的系统没有太大不同”）。

另外，不妨留意一下美国国家安全局在其安全增强型 Linux（security enhanced Linux，SELinux）中所实现的强制性访问控制（mandatory access control，MAC）。MAC 提供了一种非常细化的方式，可以在系统级别指定允许 / 禁止的操作以及系统的各个组件之间如何交互。这种细化的定义称为**安全策略**，其效果类似于囚牢：特定程序或进程只能执行策略允许的操作。

Red Hat Linux 在其企业版产品中引入了 SELinux。Novell SUSE 中的 MAC 实现称作 AppArmor，Solaris、BSD、macOS 也有类似实现。

14.17.4 参考

- man chroot
- 维基百科（SELinux Project）

- 维基百科（mandatory access control）
- Jailkit 官网

14.18 以非root用户身份运行

14.18.1 问题

你想以非 root 用户身份运行脚本，但又担心有些操作无法完成。

14.18.2 解决方案

在非 root 用户 ID（要么是你本人，要么是专门的用户）下运行脚本，并以非 root 身份交互式运行，但配置 sudo 来处理各种需要提升权限的任务。

14.18.3 讨论

无论是交互式用法还是在脚本中使用，sudo 用起来都不难。尤其注意一下 sudoers 的 NOPASSWD 选项（参见 14.19 节和 sudoers 的手册页）。

14.18.4 参考

- man sudo
- man sudoers
- 14.15 节
- 14.19 节
- 14.20 节
- 17.15 节

14.19 更安全地使用sudo

14.19.1 问题

你想使用 sudo，但又担心给过多的用户授予太多权限。

14.19.2 解决方案

很好！关注安全是应有之举。虽然用 sudo 要比不用安全太多，但 sudo 的默认设置还有很大的改进空间。

可以花点时间学习一下 sudo 本身和 /etc/sudoers 文件。尤其可以了解一下为什么大多数情况下不应该使用 ALL=(ALL) ALL ！不是说不管用，但是远谈不上安全。相较于将 root 密码告诉所有人，唯一的区别是，这里大家并不知道 root 密码，但仍然可以做 root 能做的一切。sudo 会记录下所执行的命令，但这使用 sudo bash 就能轻而易举地避开。

其次，认真思考你的需求。正如不应该使用 ALL=(ALL) ALL，你或许也不应该逐个管理用户。sudoers 允许非常精细的管理，我们强烈建议使用它。man sudoers 提供了大量材料和示例，尤其是有关防止 shell escape 的那部分内容。

第三，sudoers 文件有一个 NOPASSWD 选项，该选项允许用户执行特权操作，无须先输入用户密码，这可能正是你所期望的。这种方式能够实现需要 root 访问特权的自动化操作，同时还不会将明文密码弄得到处都是，不过显然也是有利有弊。

sudoers 允许 4 种别名：user、runas、host、command。审慎地将其作为角色或组可以大大减轻维护负担。例如，你可以为 BUILD_USERS 设置一个 User_Alias，然后用 Host_Alias 和 Cmnd_Alias 分别定义用户需要在哪些机器上执行命令以及执行哪些命令。如果设置策略，仅在一台机器上编辑 /etc/sudoers 并定期使用带有公钥认证的 scp 将其复制到所有相关机器，就可以形成一个非常安全且可用的最低特权系统。

> 当 sudo 要求输入密码时，其实索要的是你的用户账户密码，而不是 root 的。出于某些原因，人们一开始常常对此感到困惑。

14.19.3 讨论

遗憾的是，并非所有系统都默认安装了 sudo。Linux、macOS 和 OpenBSD 系统中基本都有，其他系统就不一定了。你可以查询系统文档，如果没有，自行安装即可。

> 你应该坚持用 visudo 编辑 /etc/sudoers 文件。和 vipw 一样，visudo 会锁定文件，一次只允许一个用户编辑，另外还会在替换正式文件前执行语法检查，以免用户不小心把自己锁在系统之外。

14.19.4 参考

- man sudo
- man sudoers
- man visudo
- Daniel J. Barrett、Richard Silverman、Robert G. Byrnes 合著的 *SSH, The Secure Shell: The Definitive Guide, 2nd Edition*（O'Reilly 出版）
- 14.15 节
- 14.18 节
- 14.20 节
- 17.15 节

14.20 在脚本中使用密码

14.20.1 问题

你需要将密码硬编码在脚本中。

14.20.2 解决方案

这种做法显然不是什么好主意，应该尽可能避免。可惜有时在所难免。

第一种规避的方法是，看看能否使用包含 NOPASSWD 的 sudo，这样就不用到处硬编码密码了。不过这本身也有风险，但值得一试。详见

14.19 节。

另一种方法是将 SSH 和公钥及受限命令配合使用（参见 14.21 节）。

如果没有其他解决方法，那么最好的做法就是将用户 ID 和密码放进单独的文件，该文件只能由需要的用户读取，然后必要时用 source 命令读入该文件（参见 10.3 节）。当然了，这个文件不用进行版本控制。

14.20.3 讨论

使用 SSH 安全地访问远程计算机上的数据相对比较容易（参见 14.21 节和 15.11 节）。甚至还可以使用这种方法访问同一主机上的其他数据，不过使用 sudo 的效率也许会高得多。但是，如何用 SQL 命令访问远程数据库中的数据呢？在这种情况下，你基本上做不了什么。

没错，你可能会说，用 crypt 或者其他密码散列如何？问题是，存储密码的安全方法都涉及使用单向散列（one-way hash）。密码管进不管出。也就是说，对于特定的散列，理论上是无法还原出明文密码的。但明文密码是关键，我们得用它访问数据库或别的地方。因此，安全存储就算了吧。

那剩下的就只有非安全存储了，但这可能比明文还糟糕，因为它带来了一种虚假的安全感。如果你就是喜欢，也确保不会迷信这种所谓的安全，可以动手使用 ROT13 或其他算法来混淆密码。

```
ROT13=$(echo password | tr 'A-Za-z' 'N-ZA-Mn-za-m')
```

ROT13 只能处理 ASCII 字符，你也可以使用 ROT47 来处理某些标点符号。

```
ROT47=$(echo password | tr '!-~' 'P-~!-O')
```

再强调一次也不为过，ROT13 和 ROT47 就是一种“隐晦式安全”，根本谈不上什么安全。当且仅当你（或者你的管理层）没有误认为自己“万无一失”的时候再使用这种方法，毕竟有胜于无嘛。注意，这可是有风险的。话虽如此，有时在现实中还是利大于弊。

14.20.4 参考

- 维基百科（ROT13）
- 10.3 节
- 14.15 节
- 14.18 节
- 14.19 节
- 14.21 节
- 15.11 节
- 17.15 节

14.21 使用无密码的SSH

14.21.1 问题

你需要在脚本中使用 SSH 或 `scp`，而且不想使用密码。或者你想将其用于不能有密码的 `cron` 作业。[4]

[4]感谢 Daniel Barrett、Richard Silverman、Robert Byrnes 给予我们的灵感及其在 *SSH, The Secure Shell: The Definitive Guide*（尤其是第 2 章、第 6 章和第 11 章）和 *Linux Security Cookbook* 中所做出的卓越工作，没有他们的话，这则实例肯定不会这么精彩。

SSH1（协议）和 ssh1（可执行文件）均已过时，不如 OpenSSH 和 SSH Communications Security[5] 所实现的 SSH2 协议安全。我们强烈推荐使用 OpenSSH 的 SSH2，这里不再介绍 SSH1。

[5]SSH Communications Security 是一家专注于加密和访问控制的网络安全公司。——译者注

14.21.2 解决方案

无密码的 SSH 有两种使用方式：错误的和正确的。前者是使用未经口令加密的公钥。后者是将受口令保护的公钥与 `ssh-agent` 或 `keychain` 一起使用。

假设你使用的是 OpenSSH。如果不是，请查询文档（命令和文件类似）。

首先需要创建密钥对（如果还没有的话）。无论配置了多少台机器，只用一对密钥就可以完成认证，但出于个人和工作原因，也许你会使用多对密钥。密钥对由私钥和公钥（*.pub）组成，前者应该不惜一切代价保护好，后者可以随处张贴，只要你喜欢。两者之间以复杂的数学形式相互关联，双方能够识别彼此，但无法从一个推算出另一个。

使用 `ssh-keygen`（如果使用的不是 OpenSSH，这可能是 `ssh-keygen2`）来创建密钥对。`-t` 用于指定类型，可取值请查询系统手册页。`-b` 是可选的，它指定了新密钥的位数（撰写本书之时，RSA 密钥的默认长度是 2048 位）。`-C` 可用于指定注释，如果忽略，则默认为 user@hostname。我们推荐使用 `-t rsa -b 4096 -C` *`meaningful comment`*，同时强烈反对不使用口令。`ssh-keygen` 还允许修改密钥文件的口令或注释。

```
$ ssh-keygen --help
unknown option -- -
usage: ssh-keygen [options]
Options:
  -A          Generate non-existent host keys for all key types.
  -a number   Number of KDF rounds for new key format or moduli
primality tests.
  -B          Show bubblebabble digest of key file.
  -b bits     Number of bits in the key to create.
  -C comment  Provide new comment.
  -c          Change comment in private and public key files.
  -D pkcs11   Download public key from pkcs11 token.
  -e          Export OpenSSH to foreign format key file.
  -F hostname Find hostname in known hosts file.
  -f filename Filename of the key file.
  -G file     Generate candidates for DH-GEX moduli.
  -g          Use generic DNS resource record format.
  -H          Hash names in known_hosts file.
  -h          Generate host certificate instead of a user
```

```
certificate.
  -I key_id   Key identifier to include in certificate.
  -i          Import foreign format to OpenSSH key file.
  -J number   Screen this number of moduli lines.
  -j number   Start screening moduli at specified line.
  -K checkpt  Write checkpoints to this file.
  -k          Generate a KRL file.
  -L          Print the contents of a certificate.
  -l          Show fingerprint of key file.
  -M memory   Amount of memory (MB) to use for generating DH-GEX
moduli.
  -m key_fmt  Conversion format for -e/-i (PEM|PKCS8|RFC4716).
  -N phrase   Provide new passphrase.
  -n name,... User/host principal names to include in certificate
  -O option   Specify a certificate option.
  -o          Enforce new private key format.
  -P phrase   Provide old passphrase.
  -p          Change passphrase of private key file.
  -Q          Test whether key(s) are revoked in KRL.
  -q          Quiet.
  -R hostname Remove host from known_hosts file.
  -r hostname Print DNS resource record.
  -S start    Start point (hex) for generating DH-GEX moduli.
  -s ca_key   Certify keys with CA key.
  -T file     Screen candidates for DH-GEX moduli.
  -t type     Specify type of key to create.
  -u Update   KRL rather than creating a new one.
  -V from:to  Specify certificate validity interval.
  -v          Verbose.
  -W gen      Generator to use for generating DH-GEX moduli.
  -y          Read private key file and print public key.
  -Z cipher   Specify a cipher for new private key format.
  -z serial   Specify a serial number.

$ ssh-keygen -v -t rsa -b 4096 -C 'This is my new key'
Generating public/private rsa key pair.
Enter file in which to save the key (/home/jp/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:

Your identification has been saved in /home/jp/.ssh/id_rsa.
Your public key has been saved in /home/jp/.ssh/id_rsa.pub.
The key fingerprint is:
eb:b3:0b:3a:d8:9f:d0:02:5d:99:ce:69:98:ef:f0:0c This is my new key
The key's randomart image is:
+--[ RSA 4096]----+
|                 |
|       o         |
|      +          |
|   . * .         |
```

```
|   . + = S         |
|    . +    .       |
|   oE + .          |
|  . oX +.          |
|    .o* ++         |
+-----------------+

$ $ ls -l ~/.ssh/id_rsa*
-rw------- 1 jp jp 3.3K Aug 27 15:10 /home/jp/.ssh/id_rsa
-rw-r--r-- 1 jp jp  744 Aug 27 15:10 /home/jp/.ssh/id_rsa.pub

$ fold -w75 ~/.ssh/id_rsa.pub
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAACAQCrxvIjPrLxx9VgkE0uBfdiGGZ5KC38OyTB47
7
MFyw4W7JMDnN5p7Yx8dvl91Fuc13U+RsuBBqWjNvB6hHesdWr/6D2EgoTGJDbegNNl
a+qb8jJtX
ZK1s+B9sk9SoIlT4AF5wEAMag0K4Jmv0v/xFHwVRm1BfuEQQIVP7Z8v56e7HWz/pZM
b0tM89WMg
ITyJh6cuTG1XHRmYxpOoaPBEKeDXTM0mfyAQwO2yQt6fl29RW1DH5J+jVYarsWScGe
6SKSYGQPZ
L7a3KRkbpGPRdVK2CY2P1tXQlnh9hPYqvHtAzXUMYJpSwBkNzRN3A571FBtNUxLGtP
+xHNEN7Kz
WpUsT1wv6DQw//UDSHJZShVUHMKp414y6dwmKgXTtqVWXYbB/t2EU+CuWk8OkLA2Tv
7dKUnn8tA
87D1LU3hAhr58jDEzXbIfl9yYhV2xHBxVUDf80Lv9p9ZKngRx8hkj8MoDr0J6Eql3J
hWKRqRdJy
GwKAyjCk5UQ9EH/sQ3NjhJE1Qb31o0dgE3ZKXfm8VXBZS0XTH4OHjd9RA4VCQWjEpd
R2QUgeSXW
aM94v3p6O6njKT6fFXV36S33/F/ROc1vZlcJDTpRCbpCXRNkgPtDAImBNmmweaYB0Y
m3wqHRB2I
bnw5vftDpptndB774sV2FcRxptkM8Pd/vRS35q56FSgcT6Q== This is my new
key
```

获得密钥对之后，将公钥添加到你想使用该密钥对进行连接的其他机器主目录下的~/.ssh/authorized_keys 文件中。使用 scp、cp（借助软盘或 U 盘），或者干脆在终端会话中复制粘贴，这些方法都可以。重要的是公钥必须是一整行，中间不能出现换行。虽然一个命令（如 scp id_dsa.pub remote_host:.ssh/authorized_keys）全部都能搞定，但我们不推荐这么做，哪怕你“百分之百确定”authorized_keys 并不存在。你可以使用另一个略微复杂却安全得多的命令。

```
$ ssh remote_host "echo $(cat ~/.ssh/id_rsa.pub) >>
~/.ssh/authorized_keys"
jp@remote_host's password:
```

```
$ ssh remote_host
Last login: Thu Dec 14 00:02:52 2006 from openbsd.jpsdomain.org
NetBSD 2.0.2 (GENERIC) #0: Wed Mar 23 08:53:42 UTC 2005

Welcome to NetBSD!

$ exit
logout
Connection to remote_host closed.
```

如你所见，也就是一开始的 ssh 需要输入密码，随后就不用了。这里并没有展示 ssh-agent 的使用，该命令用于缓存密钥口令，这样就无须手动输入了。

此处的命令还假设 ~/.ssh 存在于本地主机和远程主机上。如果情况并非如此，可以使用 mkdir -m 0700 -p ~/.ssh 创建。~/.ssh 目录的权限必须是 0700，否则 OpenSSH 会报错。使用 chmod 0600 ~/.ssh/authorized_keys 也不错。

另外值得注意的是，我们建立的只是一种单向关系。也就是说，无须密码就可以通过 SSH 从本地主机连接到远程主机，但反方向就不行了，因为缺少私钥和远程主机上的代理。你可以将私钥复制到各处，形成一个“纵横交错的无密码 SSH 网络”，但想修改口令时可就麻烦了，这增加了私钥保护工作的难度。如有可能，最好是配备一台安全措施良好且信得过的主机，在此之上根据需要用 ssh 连接远程主机。

SSH 代理聪明过人，用法微妙，我们甚至都觉得它机智过头了。它原本的实践用法是通过 eval 和命令替换：eval 'ssh-agent'。结果会产生两个环境变量，以便 ssh 或 scp 能找到代理并询问用户身份。这种方式很巧妙，很多地方都可以找到详细的描述。唯一的问题是和其他程序的常见用法不一样（除了 less 的某些特性，参见 8.15 节），新手或不知情的用户可能会不明所以。

如果只是运行代理，它会输出一些细节信息，看起来似乎已经开始工作了。没错，的确是在运行。但其实什么都做不了，因为必要的环境变量尚未设置。顺便提一下，-k 选项可以结束当前代理。

下面展示了一些 SSH 代理的错误用法和正确用法。

```
# 错误的代理用法
# 没有相关环境变量
$ set | grep SSH
$ ssh-agent
SSH_AUTH_SOCK=/tmp/ssh-bACKp27592/agent.27592; export
SSH_AUTH_SOCK;
SSH_AGENT_PID=24809; export SSH_AGENT_PID;
echo Agent pid 24809;

# 仍没有相关环境变量
$ set | grep SSH
# 甚至无法结束代理，因为-k需要$SSH_AGENT_PID
$ ssh-agent -k
SSH_AGENT_PID not set, cannot kill agent

# 代理是否正在运行？是的
$ ps x
  PID TT  STAT    TIME COMMAND
24809 ??  Is      0:00.01 ssh-agent
22903 p0  I       0:03.05 -bash (bash)
11303 p0  R+      0:00.00 ps -x

$ kill 24809

$ ps x
  PID TT    STAT    TIME COMMAND
22903 p0    I       0:03.06 -bash (bash)
30542 p0    R+      0:00.00 ps -x

# 正确用法
$ eval `ssh-agent`
Agent pid 21642

# 搞定，正常工作了!
$ set | grep SSH
SSH_AGENT_PID=21642
SSH_AUTH_SOCK=/tmp/ssh-ZfEsa28724/agent.28724

# 结束代理—错误的方式
$ ssh-agent -k
unset SSH_AUTH_SOCK;
unset SSH_AGENT_PID;
echo Agent pid 21642 killed;

# 进程已经结束，但没有完成清理工作
$ set | grep SSH
SSH_AGENT_PID=21642
SSH_AUTH_SOCK=/tmp/ssh-ZfEsa28724/agent.28724
```

```
# 正确的代理用法
$ eval `ssh-agent`
Agent pid 19330

$ set | grep SSH
SSH_AGENT_PID=19330
SSH_AUTH_SOCK=/tmp/ssh-fwxMfj4987/agent.4987

$ eval `ssh-agent -k`
Agent pid 19330 killed

$ set | grep SSH
$
```

挺直观的，不是吗？不。非常巧妙，非常有效，也非常微妙。但对用户就不怎么友好了。

一旦代理按预期运行，就必须用 ssh-add 命令加载身份信息。这非常简单：直接运行，根据需要选择要加载的密钥文件列表即可。ssh-add 会提示输入所需要的全部密码。这个示例没有列出任何密钥，因此它只使用主 SSH 配置文件中设置的默认值。

```
$ ssh-add
Enter passphrase for /home/jp/.ssh/id_rsa:
Identity added: /home/jp/.ssh/id_rsa (/home/jp/.ssh/id_rsa)
$
```

现在我们就可以在该 shell 会话中以交互方式使用 SSH，不需要输入密码或口令就能登录先前配置过的任何主机。那其他会话、脚本或者 cron 呢？

使用 Daniel Robbins 的 keychain 脚本，该脚本的作用如下。

> 可以充当 ssh-agent 和 ssh-add 的前端，但允许你在每个系统中轻松拥有一个能够长期运行的 ssh-agent 进程，而不是每个登录会话一个 ssh-agent。
>
> 极大地减少了输入口令的次数。有了 keychain，在本地主机每次重启时输入口令即可。

> keychain 还使得远程 cron 作业能够轻松安全地“挂接”（hook in）在长期运行的 ssh-agent 进程上，这样就允许脚本利用基于密钥的登录（key-based login）。

shell 脚本 keychain 不仅精巧、质量上乘，而且注释详尽，先前我们讨论过的那些将环境变量导到其他会话的烦琐过程，都可以通过 keychain 实现自动化并进行管理。而且还能使其为脚本和 cron 所用。但你可能会自问：先等一下，你想让我把自己所有的密钥交给 keychain，直到重启主机为止？没错，不过这并没有听起来那么糟糕。

首先，你随时都能终止 keychain，不过这也使得脚本或 cron 无法再使用它。其次，--clear 选项可以在你登录时冲洗掉已缓存的密钥。听起来怎么又倒退了？其实这么做是有道理的。keychain 的作者给出了详细的解释（首发于 IBM developerWorks）。

> 我解释过使用未经加密的私钥是一种危险的做法，因为这会使得别人盗取你的私钥，无须输入密码就能用它从任何系统登入你的远程账户。好吧，虽然 keychain 不易受到这种滥用的影响（只要坚持使用加密过的私钥），但存在一个可能会被利用的弱点，这直接与以下事实相关：keychain 使其很容易“挂接”到长期运行的 ssh-agent 进程。我在想，要是入侵者能够通过某种途径得到我的密码或口令来登录本地系统，那会怎么样？如果他们以我的身份登录，keychain 会允许入侵者立刻就能使用已解密的私钥，不费吹灰之力访问我的其他账户。
>
> 在继续之前，现在我们来客观地看待这一安全威胁。如果某些恶意用户设法以我的身份登录，keychain 的确会允许其访问我的远程账户。但即便如此，入侵者想偷走已解密的密钥也绝非易事，因为密钥在磁盘上仍是以加密形式存放的。同样，要想访问我的私钥，用户需要以我的身份登录，而不仅仅是读取个人目录中的文件。因此，滥用 ssh-agent 要比简单地窃取未加密的私钥困难得多，后者只用入侵者以某种方式访问~/.ssh 中的文件即可，是否以我的身份登录无所谓。但是，如果能够以我的身份成功登录，那么入侵者就可以使用已解密的私钥大肆破坏。因此，对于使用了 keychain 的服务器，如果你登录并不频繁或没有主动监

视其安全漏洞，则可以考虑使用 --clear 选项来提供额外的安全层。

--clear 选项允许 keychain 将账户的每次登录都视为潜在的安全漏洞，除非你能另行证明。用 --clear 选项启动 keychain 时，keychain 会在你登录后立即冲洗掉 ssh-agent 缓存的所有私钥。因此，如果你是入侵者，keychain 会提示输入口令，而不是直接允许你访问已缓存的密钥。虽然这增强了安全性，但确实让事情变得有些不便，而且与单独运行 ssh-agent（不使用 keychain）没差多少。在这里，和通常一样，人们可以选择更高的安全性或更好的便利性，但无法两者皆得。

尽管如此，使用包含 --clear 选项的 keychain 还是比单独使用 ssh-agent 更有优势。记住，如果使用 keychain --clear，那你的 cron 作业和脚本仍然能够建立无密码连接。这是因为私钥是在**登录**时被冲洗的，而不是**注销**时。从系统注销并不会产生潜在的安全漏洞，keychain 用不着此时冲洗 ssh-agent 的密钥。因此，对于那些偶尔执行安全复制任务的非常用服务器，如备份服务器、防火墙、路由器，--clear 选项是一个不错的选择。

要想在脚本或 cron 中使用 keychain 包装过的 ssh-agent，对脚本中创建的 keychain 文件使用 source 即可。keychain 也能处理 GPG 密钥。

```
[ -r ~/.ssh-agent ] && source ~/.ssh-agent \
|| { echo "keychain not runnin" >&2 ; exit 1; }
```

14.21.3 讨论

在脚本中使用 SSH 时，你不想被提示进行认证或显示过多的警告信息。-q 选项可以开启安静模式并禁止警告，而 -o 'BatchMode yes' 会阻止用户提示。显然，如果不能完成 SSH 身份验证，肯定会失败，因为根本就无法回退，提示你再输入密码。不过既然你已经读到这里了，这应该不是问题。

SSH 是一款令人称奇的工具，有太多值得一说的地方，足以再写一本和本书同等厚度的专著。我们强烈推荐由 Daniel J. Barrett、Richard Silverman、Robert G. Byrnes 合著的 *SSH, The Secure Shell: The Defnitive Guide, 2nd Edition*（O'Reilly 出版），你想知道的有关 SSH 的一切（甚至更多），尽在其中。

在 OpenSSH 和 SSH Communications Security 的 SSH2 Server 之间使用公钥是件棘手的事，你可以在 *Linux Security Cookbook*（O'Reilly 出版）一书的第 6 章中找到一些相关技巧。

keychain 的作者（以及 Gentoo 首席架构师）Daniel Robbins 在 IBM developerWorks 上发表的有关 SSH 的文章也是不错的参考资料。

如果 keychain 看起来没有工作，或者工作了一段时间后似乎又停止了，有可能是别处的其他脚本重新运行了 ssh-agent，导致操作不同步。执行下列检查，确保 PID 和套接字一致。

```
$ ps -ef | grep [s]sh-agent
jp17364  0.0  0.0 3312 1132?        S   Dec16     0:00 ssh-agent

$ cat ~/.keychain/$HOSTNAME-sh
SSH_AUTH_SOCK=/tmp/ssh-UJc17363/agent.17363; export SSH_AUTH_SOCK;
SSH_AGENT_PID=17364; export SSH_AGENT_PID;

$ set | grep SSH_A
SSH_AGENT_PID=17364
SSH_AUTH_SOCK=/tmp/ssh-UJc17363/agent.17363
```

根据所使用的操作系统，你可能需要调整 ps 命令的选项。如果 -ef 不管用，试试 -eu。

密钥指纹

各种类型的 SSH 都支持**指纹**，以便于用户密钥和主机密钥之间的比对和核实。你大概也猜到了，逐位核实形似随机的冗长数据，既乏味又容易出错，不过这都算是好的了；最坏的情况下，几乎不可能核实（如通过电话）。指纹提供了一种更简便的核实方式。

你可能也在其他应用程序中见识过指纹，尤其是 PGP/GPG 密钥。

先核实密钥是为了避免**中间人**攻击。如果 Alice 将她的密钥发送给 Bob，Bob 必须确保接收到的密钥的确来自 Alice，没有中途被Eve 截获并替换成自己的。这需要使用带外通信信道（out-of-band communications channel），例如电话。

指纹格式有两种：PGP 采用的传统十六进制格式和更易于阅读的新格式 bubblebabble。当接收到 Alice 的密钥，Bob 会给她打电话并读出她的指纹。如果一致，两人就都知道拿到了正确的密钥。

```
$ ssh-keygen -l -f ~/.ssh/id_rsa
4096 eb:b3:0b:3a:d8:9f:d0:02:5d:99:ce:69:98:ef:f0:0c  This is
my new key (RSA)

$ ssh-keygen -l -f ~/.ssh/id_rsa.pub
4096 eb:b3:0b:3a:d8:9f:d0:02:5d:99:ce:69:98:ef:f0:0c  This is
my new key (RSA)

$ ssh-keygen -B -f ~/.ssh/id_rsa
4096 xuked-dutis-hoper-berag-ducut-tycuc-salur-ruvin-kefeg-
mobyg-nyxyx
  This is my new key (RSA)

$ ssh-keygen -B -f ~/.ssh/id_rsa.pub
4096 xuked-dutis-hoper-berag-ducut-tycuc-salur-ruvin-kefeg-
mobyg-nyxyx
  This is my new key (RSA)
```

14.21.4 参考

- Daniel J. Barrett、Richard Silverman、Robert G. Byrnes 合著的 *SSH, The Secure Shell: The Defnitive Guide, 2nd Edition*（O'Reilly 出版）
- Daniel J. Barrett、Richard Silverman、Robert G. Byrnes 合著的 *Linux Security Cookbook*（O'Reilly 出版）
- Niels Ferguson 与 Bruce Schneier 合著的 *Practical Cryptography*（Wiley 出版）
- Bruce Schneier 所著的 *Applied Cryptography*（Wiley 出版）
- 8.15 节

14.22 限制SSH命令

14.22.1 问题

你想限制接入的 SSH 用户或脚本能够执行的操作。[6]

[6]同本章注 4。

14.22.2 解决方案

编辑 ~/.ssh/authorized_keys 文件，使用 SSH 的强制命令（forced command），同时有选择地禁用不必要的 SSH 特性。例如，假设你希望允许 rsync 进程，但不允许交互式用法。

首先，你需要决定究竟在远端运行什么命令。创建密钥（参见 14.21 节）并添加强制命令。编辑 ~/.ssh/authorized_keys 并在密钥前加入：

```
command="/bin/echo Command was: $SSH_ORIGINAL_COMMAND"
```

如下所示，所有内容全部出现在一行中。

```
command="/bin/echo Command was: $SSH_ORIGINAL_COMMAND" ssh-dss
AAAAB3NzaC1kc3MAAAEBANpgvvTslst2m0ZJA0ayhh1Mqa3aWwU3kfv0m9+myFZ9ve
FsxM7
IVxIjWfAlQh3jplY+Q78fMzCTiG+ZrGZYn8adZ9yg5wAC03KXm2vKt8LfTx6I+qkMR
7v15N
I7tZyhxGah5qHNehReFWLuk7JXCtRrzRvWMdsHcL2SA1Y4fJ9Y9FfVlBdE1Er+ZIuc
5xIlO
6D1HFjKjt3wjbAal+oJxwZJaupZ0Q7N47uwMslmc5ELQBRNDsaoqFRKlerZASPQ5P+
AH/+C
xa/fCGYwsogXSJJ0H5S7+QJJHFze35YZI+A1D3BIa4JBf1KvtoaFr5bMdhVAkChdAd
Mjo96
xhbdEAAAAVAJSKzCEsrUo3KAvyUO8KVD6e0B/NAAAA/3uAx2TIB/M9MmPqjeH67Mh5
Y5NaV
WuMqwebDIXuvKQQDMUU4EPjRGmS89Hl8UKAN0Cq/C1T+OGzn4zrbE06COSm3SRMP24
HyIbE
lhlWV49sfLR05Qmh9fRl1s7ZdcUrxkDkr2J6on5cMVB9M2nIl90IhRVLd5RxP01u81
yqvhv
E61ORdA6IMjzXcQ8ebuD2R733O37oGFD7e2O7DaabKKkHZIduL/zFbQkzMDK6uAMP8
ylRJN
```

```
0fUsqIhHhtc/16OT2H6nMU09MccxZTFUfqF8xIOndElP6um4jXYk5Q30i/CtU3TZyv
NeWVw
yGwDi4wg2jeVe0YHU2RhZcZpwAAAQEAv2O86701U9sIuRijp8sO4h13eZrsE5rdn6a
ul/mk
m+xAlO+WQeDXRONm9BwVSrNEmIJB74tEJL3qQTMEFoCoN9Kp00Ya7Qt8n4gZ0vcZlI
5u+cg
yd1mKaggS2SnoorsRlb2LhHpe6mXus8pUTf5QT8apgXM3TgFsLDT+3rCt40IdGCZLa
P+UDB
uNUSKfFwCru6uGoXEwxaL08Nv1wZOc19qrc0Yzp7i33m6i3a0Z9Pu+TPHqYC74QmBb
Wq8U9
DAo+7yhRIhqfdJzk3vIKSLbCxg4PbMwx2Qfh4dLk+L7wOasKnl5//W+RWBUrOlaZ1Z
P1/az
sK0Ncygno/0F1ew== This is my new key
```

现在执行命令，看看结果如何。

```
$ ssh remote_host 'ls -l /etc'
Command was: ls -l /etc
$
```

这种方法的问题是，这会破坏像 rsync 这样的程序，该程序依赖于将 STDOUT/STDIN 全部连接到自身。

```
$ rsync -avzL -e ssh remote_host:/etc .
protocol version mismatch -- is your shell clean?
(see the rsync manpage for an explanation)
rsync error: protocol incompatibility (code 2) at compat.c(64)
$
```

但可以通过修改强制命令来解决这个问题。

```
command="/bin/echo Command was: $SSH_ORIGINAL_COMMAND >>
~/ssh_command"
```

在客户端再试一次：

```
$ rsync -avzL -e ssh 192.168.99.56:/etc .
rsync: connection unexpectedly closed (0 bytes received so far)
[receiver]
rsync error: error in rsync protocol data stream (code 12) at
io.c(420)
$
```

在远端主机上：

```
$ cat ../ssh_command
Command was: rsync --server --sender -vlLogDtprz . /etc
$
```

因此，我们可以根据需要更新强制命令。

另外两件能做的事情分别是**设置源主机限制**和禁用 SSH 命令。主机限制指定了源主机的主机名或 IP 地址。禁用命令的写法也相当直观。

```
no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty
```

将所有这些合并在一起，如下所示（仍旧全部出现在很长的一行中）。

```
no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-
pty,from="local_
client",command="rsync --server --sender -vlLogDtprz . /etc" ssh-
dss
AAAAB3NzaC1kc3MAAAEBANpgvvTslst2m0ZJA0ayhh1Mqa3aWwU3kfv0m9+myFZ9ve
FsxM7
IVxIjWfAlQh3jplY+Q78fMzCTiG+ZrGZYn8adZ9yg5wAC03KXm2vKt8LfTx6I+qkMR
7v15N
I7tZyhxGah5qHNehReFWLuk7JXCtRrzRvWMdsHcL2SA1Y4fJ9Y9FfVlBdE1Er+ZIuc
5xIlO
6D1HFjKjt3wjbAal+oJxwZJaupZ0Q7N47uwMslmc5ELQBRNDsaoqFRKlerZASPQ5P+
AH/+C
xa/fCGYwsogXSJJ0H5S7+QJJHFze35YZI+A1D3BIa4JBf1KvtoaFr5bMdhVAkChdAd
Mjo96
xhbdEAAAAVAJSKzCEsrUo3KAvyUO8KVD6e0B/NAAAA/3uAx2TIB/M9MmPqjeH67Mh5
Y5NaV
WuMqwebDIXuvKQQDMUU4EPjRGmS89Hl8UKAN0Cq/C1T+OGzn4zrbE06COSm3SRMP24
HyIbE
lhlWV49sfLR05Qmh9fRl1s7ZdcUrxkDkr2J6on5cMVB9M2nIl90IhRVLd5RxP01u81
yqvhv
E61ORdA6IMjzXcQ8ebuD2R733O37oGFD7e2O7DaabKKkHZIduL/zFbQkzMDK6uAMP8
ylRJN
0fUsqIhHhtc/16OT2H6nMU09MccxZTFUfqF8xIOndElP6um4jXYk5Q30i/CtU3TZyv
NeWVw
yGwDi4wg2jeVe0YHU2RhZcZpwAAAQEAv2O86701U9sIuRijp8sO4h13eZrsE5rdn6a
ul/mk
m+xAlO+WQeDXRONm9BwVSrNEmIJB74tEJL3qQTMEFoCoN9Kp00Ya7Qt8n4gZ0vcZlI
5u+cg
yd1mKaggS2SnoorsRlb2LhHpe6mXus8pUTf5QT8apgXM3TgFsLDT+3rCt40IdGCZLa
P+UDB
uNUSKfFwCru6uGoXEwxaL08Nv1wZOc19qrc0Yzp7i33m6i3a0Z9Pu+TPHqYC74QmBb
```

```
Wq8U9
DAo+7yhRIhqfdJzk3vIKSLbCxg4PbMwx2Qfh4dLk+L7wOasKnl5//W+RWBUrOlaZ1Z
P1/az
sK0Ncygno/0F1ew== This is my new key
```

14.22.3 讨论

如果在运用 ssh 的过程中碰到问题，-v 选项能帮上大忙。ssh -v 或 ssh -v -v 起码在大部分情况下能给出一些错误线索。使用时不妨尝试一下这些选项，了解输出结果。

如果对密钥的功能感兴趣，可以深入了解 OpenSSH 的受限 shell：rssh，它支持 scp、sftp、rdist、rsync 和 cvs。

你会觉得这种限制很容易实现，但事实证明并非如此。该问题与 SSH（以及在其之前的 r 系列命令）的实际运行方式有关。这是个绝妙的想法，效果也很好，但不易实现。为简化起见，你可以将 SSH 看作是将本地的 STDOUT 连接到远端的 STDIN，同时将远端的 STDOUT 连接到本地的 STDIN，因此，诸如 scp 或 rsync 的所有操作都是将本地主机的一个个字节送往远程主机，就好像二者之间有条管道一样。但这种高度的灵活性使得 SSH 无法在允许 scp 的同时限制交互式访问。两者之间并没有什么不同。这也是你不能在 bash 配置文件中放置大量 echo 和调试语句的原因（参见 16.21 节），其输出会与字节流混杂在一起，造成灾难性后果。

那么 rssh 是如何工作的？它提供了一个包装器，用于替代 /etc/passwd 中指定的默认登录 shell（如 bash）。该包装器决定了哪些是允许的，哪些是不允许的，但又比普通的旧式 SSH 受限命令灵活得多。

14.22.4 参考

- Daniel J. Barrett、Richard Silverman、Robert G. Byrnes 合著的 *SSH, The Secure Shell: The Defnitive Guide, 2nd Edition*（O'Reilly 出版）
- Daniel J. Barrett、Richard Silverman、Robert G. Byrnes 合著的 *Linux Security Cookbook*（O'Reilly 出版）

- 14.21 节
- 16.21 节

14.23 断开非活跃会话

14.23.1 问题

你希望能够自动注销那些不活跃的用户，尤其是 root。

14.23.2 解决方案

将 /etc/bashrc 或 ~/bashrc 中的环境变量 $TMOUT 设置为结束会话之前的非活跃秒数。在交互模式中，只要出现了命令行提示符，如果用户没有在 $TMOUT 秒内输入命令，那么 bash 就会退出。

14.23.3 讨论

$TMOUT 也可用于脚本中的 read 内建命令和 select 命令。

如果不想有人改动 $TMOUT，记得在用户没有写入权限的系统级文件（如 /etc/profile 或 /etc/bashrc）将其设置为只读变量。

```
declare -r TMOUT=3600

# 或者：
readonly TMOUT=3600
```

> 由于用户能够控制自己的环境变量，即便你已经将 $TMOUT 设为只读，也别完全依靠它：用户只需要运行其他 shell，甚至是 bash 的不同实例便可轻松化解！可以将此视作对协作用户的善意提示，尤其是那些知识丰富而且随时待命的系统管理员，他们可能会（不断地）分神。

14.23.4 参考

- 16.21 节

第 15 章　高级脚本编程

长期以来，Unix 和 POSIX 一直承诺并竭力实现兼容性和可移植性。因此，高级脚本编写人员面临的最大问题之一就是编写**可移植的**脚本，也就是说，这种脚本可以在任何安装了 bash 的机器上工作。编写能够在各种平台上良好运行的脚本比我们预想的要困难得多。不同系统之间存在很多差异，例如，bash 自身未必总是安装在同一位置，根据操作系统的不同，许多常用的 Unix 命令选项或输出亦略有不同。本章将研究其中一些问题，并展示如何使用 bash 来解决。

许多其他偶有需要的事情也并不像我们希望的那样简单。因此，我们还会涵盖其他高级脚本编程任务的解决方案，例如，使用 phase 自动化执行流程、通过脚本发送电子邮件、向 syslog 记录日志、使用网络资源，以及获取输入和重定向输出的一些技巧。

虽然本章是关于高级脚本编程的，但我们还是要强调对清晰代码的需求，代码应尽可能简单并编写好文档。第一代 Unix 开发人员之一的 Brian Kernighan 说得好：

> 比起最初写的代码，调试才是难上加难，如果把十八般武艺全都花在了代码身上，等到调试的时候，十有八九也是心有余而力不足了。[1]

[1]这句话出自 *The Elements of Programming Style, 2nd Edition* 的第 2 章（Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.）。——译者注

写出精巧的脚本并不难，但这种脚本难以让人看明白。在解决问题的过程中，你越觉得自己当时干得聪明，等过上一年半载，你（更糟的情况是别人）抓耳挠腮想不明白为什么会出错的时候，就越后悔不已。如果不得不施展一些奇技妙招才能过关，那你好歹也要用注释写清楚脚本是如何工作的（参见 5.1 节）。

15.1　以可移植的方式查找bash

15.1.1 问题

你需要在多台机器上运行 bash 脚本，但 bash 未必总位于相同的位置（参见 1.14 节）。

15.1.2 解决方案

在 shebang 行中使用 /usr/bin/env 命令，也就是 `#!/usr/bin/env bash`。如果所在系统的 /usr/ bin 中没有 env，你可以请求系统管理员安装该命令并将其放到此位置，或者创建符号链接。

你也可以为 bash 自身创建符号链接，但使用 env 作为解决方案不仅规范，而且正确。

15.1.3 讨论

env 的作用是“在改动过的环境中运行程序”，但因为它会搜索待运行的命令，所以非常适合在这里使用。

你也许想改用 `!/bin/sh`。别这么做。如果脚本中用到了 bash 独有的特性，对于那些没有在 Bourne shell 模式下使用 bash 的机器（如 BSD、Solaris、Ubuntu 6.10+），这些特性将无法使用。即便现在没有涉及 bash 独有的特性，搞不好以后就把这件事给忘了。如果下定决心只使用 POSIX 特性，一定要采用 `!/bin/sh`（并且不要在 Linux 上进行开发工作，参见 15.3 节），否则视具体情况而定。

有时你可能会发现 `#!` 和 `/bin/`*whatever* 之间有一个空格。过去有些系统要求有这个空格，不过我们很久没在实践中这么写了。运行 bash 的系统几乎都不会要求使用空格，不加空格是目前最常见的写法。但为了获得最大程度的向后兼容性，可以将空格加上。

我们选择在代码较多的脚本和函数中使用 `#!/usr/bin/env bash`，这样在大多数系统中就不用再修改了。但是，因为 env 要借助 $PATH 来查找 bash，这可以说存在安全问题（参见 14.2 节），不过我们觉得也算不上什么大事。

我们尝试用 env 实现可移植性，而 shebang 行的处理在不同系统之间并不一致，这实在是讽刺。包括 Linux 在内的很多系统只允许解释器携带一个参数。因此，#!/usr/bin/env bash - 会产生错误。

```
/usr/bin/env: bash -: No such file or directory
```

原因在于解释器是 /usr/bin/env，允许出现在其后的单个参数是 bash -。其他系统（如 BSD 和 Solaris）则没有这种限制。

尾部的 - 是一种常见的安全实践（参见 14.2 节），但并非所有系统都支持这种写法，因此存在安全性和可移植性问题。

要么以牺牲可移植性为代价，使用尾部的 - 来提升些许安全性；要么为了可移植性，直接将其忽略，代价是会带来一点潜在的安全风险。

因此，我们的建议是，如果考虑到可移植性而使用 env，那么可以忽略 -；如果安全至关重要，则可以将解释器硬编码进 shebang 行并在末尾添加 -。

15.1.4 参考

- shebang 行（/usr/bin/env）的相关信息：
  - Unix FAQs 的 3.16 节
- 1.14 节
- 15.2 节
- 15.3 节
- 15.6 节

15.2 设置兼容POSIX工具的$PATH

15.2.1 问题

你所在机器上的一些工具要么上了年代，要么属于专有工具（如 Solaris），你需要设置好路径，以便能够正常执行兼容 POSIX 的工具。

15.2.2 解决方案

使用实用工具 getconf。

```
PATH=$(PATH=/bin:/usr/bin getconf PATH)
```

以下是几种系统上的一些默认路径和 POSIX 路径。

```
# Red Hat Enterprise Linux (RHEL) 4.3
$ echo $PATH
/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/hom
e/$USER/bin

$ getconf PATH
/bin:/usr/bin

# Debian Sarge
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games

$ getconf PATH
/bin:/usr/bin

# Solaris 10
$ echo $PATH
/usr/bin:

$ getconf PATH
/usr/xpg4/bin:/usr/ccs/bin:/usr/bin:/opt/SUNWspro/bin

# OpenBSD 3.7
$ echo $PATH
/home/$USER/bin:/bin:/sbin:/usr/bin:/usr/sbin:/usr/X11R6/bin:/usr/
local/bin:/usr/
local/sbin:/usr/games

$ getconf PATH
/usr/bin:/bin:/usr/sbin:/sbin:/usr/X11R6/bin:/usr/local/bin
```

15.2.3 讨论

getconf 能够显示出各种系统配置变量，你可以用它来设置默认路径。但除非 getconf 本身就是内建的，否则还需要一个最小路径来找到它，因此解决方案还包括 PATH=/bin:/usr/bin。

从理论上说，你应该使用变量 CS_PATH。但在实践中，PATH 在我们测试过的所有系统中都没有问题，而 CS_PATH 在 BSD 中反倒不行。

15.2.4 参考

- “Shell Corner: Processing Command-line Arguments with my_getopts”，*Unix Review*（2003 年 1 月刊）
- 9.11 节
- 14.3 节
- 14.9 节
- 14.10 节
- 16.4 节
- 16.5 节
- 19.3 节

15.3 开发可移植的shell脚本

15.3.1 问题

你所编写的 shell 脚本需要在多个版本的 Unix 或 POSIX 操作系统上运行。

15.3.2 解决方案

首先，使用包含 -p 选项的 command 找出 *program* 的 POSIX 版本。例如，在 Solaris 中的 /usr/xpg4 或 /usr/xpg6：

```
command -p program args
```

然后，如果可能的话，找一台年代最久或者功能最少的 Unix 机器，在该平台上开发脚本。如果不确定哪个平台的功能最少，可以使用 BSD 变体或 Solaris（版本越旧越好）。

15.3.3 讨论

command -p 使用默认路径 [2]，以确保能够找到所有符合 POSIX 标准的实用工具。如果你确定脚本只在 Linux 上运行，那就不用操心了；否则，避免在 Linux 或 Windows（例如，通过 Cygwin）上开发跨平台脚本。

[2] -p 选项使用预定义的默认搜索路径，而不是 PATH 的当前值。——译者注

在 Linux 上编写跨平台脚本存在两个问题。

01. /bin/sh 并非 Bourne shell，而是处于 POSIX 模式的 /bin/bash，除非它指向的是 /bin/dash（如 Ubuntu 6.10+）。两者都挺不错，但算不上完美，而且三者的工作方式也不完全一致，着实让人困惑不已。尤其是 echo 的行为不尽相同。
02. Linux 使用的是 GNU 工具，而不是原先的 Unix 工具。

别误会，我们热爱 Linux，天天都离不开它。但是，Linux 真的和 Unix 不是一回事：它的某些行事方式不同，而且拥有 GNU 工具。GNU 工具挺不错，问题也就在于此。这些工具所包含的大量选项和特性在其他平台上根本找不到，不管多小心翼翼，你的脚本还是会莫名其妙地出错。相反，Linux 的兼容性非常好，为别的类 Unix 系统编写的脚本基本都能在 Linux 上运行。也许表现的不算完美（例如，echo 的默认行为是显示 \n，而不是输出换行符），但通常已经够好了。

这里出现了一个让人进退两难的局面：你使用的 shell 特性越多，就越少依赖那些存不存在、能不能按照预期工作都不一定的外部程序。虽然 bash 的功能远胜于 sh，但它本身也未必总是存在。几乎所有 Unix 或类 Unix 系统都会采用某种形式的 sh，但它不一定总和你想的一样。

另一个为难的地方是，虽然 GNU 长选项在 shell 代码中可读性要好得多，但在其他系统中通常并不可用。因此，为了可移植性，还是老

老实实用 sort -t, unsorted_file>sorted_ filesort 代替 sort --field-separator=, unsorted_file > sorted_file 吧。

别泄气：在非 Linux 系统上做开发要比以前更容易了。如果你已经拥有并正在使用此类系统，这显然不是什么问题。如果尚未配备，现在动手，很容易就可以弄到。Solaris 和 BSD 都能在虚拟化环境中运行（参见 15.4 节）。

如果你使用的是运行 macOS（以前是 OS X）的 Mac，那么 bash 和 BSD 就是现成的了。如果你想要确保拥有最新版本，参见 1.15 节。

你还可以使用虚拟化环境轻松地测试脚本（参见 15.4 节）。这种解决方案的缺点是，AIX 和 HP-UX 等系统无法在 x86 架构之上运行，自然也就不能在 x86 虚拟化环境中运行。同样，如果你有这些系统，那就用；如果没有，参见 1.18 节。

> Debian 和 Ubuntu 用户应该安装 devscripts 软件包（aptitude install devscripts），它提供了 checkbashisms 脚本，可以帮助查找那些不兼容 POSIX、无法在 dash 中使用的 shell 脚本（bashisms）。其他操作系统或 Linux 发行版的用户不妨看看该脚本是否可用于自己的系统。

15.3.4 参考

- help command
- 维基百科（Almquist shell）
- 维基百科（Bash）
- comp.sys.hp.hpux FAQ
- Unix 史
- Unix 历史库
- 1.18 节
- 15.4 节
- 15.6 节
- A.11 节

15.4 用虚拟机测试脚本

15.4.1 问题

你需要开发跨平台的脚本，但缺乏适合的系统或硬件。

15.4.2 解决方案

如果目标平台运行在 x86 架构上，要么在众多的免费和商业虚拟化解决方案中选择一种，构建自己的测试虚拟机，要么在操作系统厂商的网站或者 Internet 上搜索预先构建好的虚拟机。也可以使用云供应商提供的免费（试用期间）或低成本的虚拟机。

这种解决方案的缺陷是，AIX 和 HP-UX 这种系统无法在 x86 架构上运行，因此 x86 虚拟化也就无计可施了。还是那句话，如果你有这些系统，那就用；如果没有，参见 1.18 节。

15.4.3 讨论

测试 shell 脚本通常不属于资源密集型操作，能运行 VirtualBox 或者类似虚拟化软件的一般硬件就可以了。我们特别提及 VirtualBox 是因为它不用花钱，可以在 Linux、macOS、Windows 平台上运行，网上不计其数的示例和工具（如 Vagrant）中都能看到其身影，而且灵活易用，不过肯定也有别的替代品。

配备 128 MB 内存（有时甚至更少）的最小虚拟机对于 shell 环境测试已经绰绰有余了。可以设置 NFS 共享来存储测试脚本和数据，然后通过 SSH 连接到测试系统。如果想要自己构建虚拟机，Debian 是个不错的选择；记得在安装过程中取消能取消的一切选项。

Internet 上有不少预先构建好的虚拟机，但质量和安全性参差不齐。如果你是在工作中进行测试，请务必核对公司的政策；许多公司禁止将“从 Internet 随意下载的东西”带进公司网络。另外，你所在的公司也许会构建或提供自己的虚拟机映像以供内部使用。你可能只需要一个非常小的虚拟机来测试 shell 脚本，但“小”这个词的定义在

不同的环境中差别颇大。估计你得花点心思才能找到符合需求的虚拟机。不妨先看看几个不错的地方。

- TurnKey Linux
- VMware
- OSBoxes
- KVM
- Parallels

根据需求和公司政策，你也可以从云端获取免费的或者低成本的虚拟机。有关如何从 polarhome 网站申请几乎免费的 shell 账户的详细信息，参见 1.18 节，该账户仅象征性地收取很少的一次性费用，或者考虑其他供应商。

Amazon 提供的“免费套餐”也许能派上用场，它和其他很多供应商（如 Linode、Digital Ocean）都提供了非常便宜的现用现付选项。

别忘了，也可以引导 LiveCD 或 LiveDVD，1.18 节提到过的。

最后，如果这些还不够，QEMU 仿真器的发起人 Fabrice Bellard 用 JavaScript 编写了一个 PC 仿真器，它允许你只用 Web 浏览器就可以引导虚拟机镜像。

无论做出何种选择，Internet 上的相关信息、文档、操作指南要比本节给出的多得多。这里的主要目标只是让你思考一些可能性。

> 在实践本实例中的任何操作之前，请务必先核对你所在公司的政策！

15.4.4 参考

- VirtualBox 官网
- Debian 安装网站
- Turnkey Core（bash 4.3 或更高版本）
- Vmware（商业版）
- KVM 官方网站
- polarhome 网站
- Free Shell Accounts

- QEMU 网站
- Virtual x86
- 1.14 节
- 1.18 节

15.5 使用可移植的循环

15.5.1 问题

你要用到 for 循环，但同时希望它也能在较旧的 bash 版本中正常工作。

15.5.2 解决方案

以下方法能够向后兼容到 bash 2.04+。

```
$ for ((i=0; i<10; i++)); do echo $i; done
0
1
2
3
4
5
6
7
8
9
```

15.5.3 讨论

较新的 bash 版本中还有一些更好的循环写法，但无法向后兼容。从 bash 3.0+ 开始，你可以使用语法 for {x..y}。

```
$ for i in {1..10}; do echo $i; done
1
2
3
4
5
```

```
6
7
8
9
10
```

如果系统有 seq 命令，也可以像下面这样做：

```
$ for i in $(seq 1 10); do echo $i; done
1
2
3
4
5
6
7
8
9
10
```

15.5.4 参考

- help for
- man seq
- 6.12 节
- 6.13 节
- 17.24 节

15.6 使用可移植的echo

15.6.1 问题

编写的脚本将在多个版本的 Unix 和 Linux 上运行，你需要 echo 表现出一致的行为，即便没有使用 bash。

15.6.2 解决方案

使用 printf "%b" *whatever*，或者对系统进行测试，根据需要使用 shopt -s xpg_echo 设置 xpg_echo。

如果忽略格式字符串 "%b"（如 printf *whatever*），那么 printf 会尝试解释出现在 *whatever* 中的 % 字符，这可能不是你想要的结果。"%b" 格式是对标准 printf 格式的补充，能够避免这种误解并扩展 *whatever* 中出现的转义序列。

设置 xpg_echo 的一致性较差，因为它仅适用于 bash。如果你确定只在 bash 下运行，不会涉及 sh 或其他不使用 xpg_echo 的 shell，这种方法还是行之有效的。

printf 的写法不同于 echo 语句，前者是由 POSIX 定义的，应该能在任何 POSIX shell 中保持一致性。具体来说，你得写成 printf "%b"，而不仅仅是 echo。

> 如果顺手输入的是 $b，而不是 %b，那可就要倒霉了，这时会出现一个空行，因为你指定的是空格式（null format），也就是说，除非已经定义过 $b，在这种情况下，输出结果取决于 $b 的值。不管怎样，这都是一个很难查找的 bug，毕竟 $b 和 %b 看起来的确很像。

```
$ printf "%b" "Works"
Works
$ printf "$b" "Broken"
$
```

15.6.3 讨论

某些 shell 内建的 echo 的行为和外部程序的 echo 并不一样。这一点在 Linux 中未必总能被注意到，因为 /bin/sh 指向的其实是 bash（通常是这样，在 Ubuntu 6.10+ 中则是 dash），有些 BSD 系统中也存在类似情况。两种 echo 的区别在于是否扩展转义序列。shell 内建版本倾向于不扩展，而外部版本（如 /bin/echo 和 /usr/bin/echo）倾向于扩展；但还是那句话，不同的系统之间会有变化。

典型的 Linux 系统（/bin/bash）如下所示。

```
$ type -a echo
echo is a shell builtin
echo is /bin/echo

$ builtin echo "one\ttwo\nthree"
one\ttwo\nthree\n

$ /bin/echo "one\ttwo\nthree"
one\ttwo\nthree\n

$ echo -e "one\ttwo\nthree"
one → two
three

$ /bin/echo -e "one\ttwo\nthree"
one → two
three

$ shopt -s xpg_echo

$ builtin echo "one\ttwo\nthree"
one → two
three

$ shopt -u xpg_echo

$ builtin echo "one\ttwo\nthree"
one\ttwo\nthree\n
```

典型的 BSD 系统（/bin/csh，然后是 /bin/sh）如下所示。

```
$ which echo
echo: shell builtin command.

$ echo "one\ttwo\nthree"
one\ttwo\nthree\n

$ /bin/echo "one\ttwo\nthree"
one\ttwo\nthree\n

$ echo -e "one\ttwo\nthree"
-e one\ttwo\nthree\n

$ /bin/echo -e "one\ttwo\nthree"
-e one\ttwo\nthree\n
```

```
$ printf "%b" "one\ttwo\nthree"
one → two

$ /bin/sh

$ echo "one\ttwo\nthree"
one\ttwo\nthree\n

$ echo -e "one\ttwo\nthree"
one → two
three

$ printf "%b" "one\ttwo\nthree"
one → two
three
```

Solaris 10（/bin/sh）系统如下所示。

```
$ which echo
/usr/bin/echo

$ type echo
echo is a shell builtin

$ echo "one\ttwo\nthree"
one → two
three

$ echo -e "one\ttwo\nthree"
-e one → two
three

$ printf "%b" "one\ttwo\nthree"
one → two
three
```

15.6.4 参考

- help printf
- man 1 printf
- 2.3 节
- 2.4 节
- 15.1 节
- 15.3 节
- 19.11 节

- A.12 节

15.7 仅在必要时分割输出

15.7.1 问题

你希望仅在输入超出限制时才分割输出，但 split 命令总是会创建至少一个新文件。

15.7.2 解决方案

例 15-1 演示了一种方法，该方法可以仅在输入超出限制时才将其分割为固定大小的输出。

例 15-1 ch15/func_split

```
# 实例文件：func_split
#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# 仅在超出限制时才将输入分割成固定大小的输出
# 调用方式：Split <file> <prefix> <limit option> <limit argument>
# 例如：Split $output ${output}_ --lines 100
# 选项细节参见split(1) 和wc(1)
function Split {
    local file=$1
    local prefix=$2
    local limit_type=$3
    local limit_size=$4
    local wc_option

    # 完备性检查
    if [ -z "$file" ]; then
        printf "%b" "Split: requires a file name!\n"
        return 1
    fi
    if [ -z "$prefix" ]; then
        printf "%b" "Split: requires an output file prefix!\n"
        return 1
    fi
    if [ -z "$limit_type" ]; then
        printf "%b" \
          "Split: requires a limit option (e.g. --lines), see 'man
```

```
split'!\n"
        return 1
    fi
    if [ -z "$limit_size" ]; then
        printf "%b" "Split: requires a limit size (e.g. 100), see
'man split'!\n"
        return 1
    fi
    # 将split的选项转换成wc的选项
    # 并非所有系统上的wc/split都能够支持全部的选项
    case $limit_type in
        -b|--bytes)      wc_option='-c';;
        -C|--line-bytes) wc_option='-L';;
        -l|--lines)      wc_option='-l';;
    esac

    # 如果超出了限制
    if [ "$(wc $wc_option $file | awk '{print $1}')" -gt
$limit_size ]; then
        # 进行分割
        split --verbose $limit_type $limit_size $file $prefix
    fi
} # Split函数定义完毕
```

15.7.3 讨论

根据使用的系统，某些选项（如 -C）可能不适用于 split 或 wc。

15.7.4 参考

- 8.13 节

15.8 以十六进制形式查看输出

15.8.1 问题

你需要以十六进制形式查看输出，以核实某些空白字符或不可打印字符是否符合预期。

15.8.2 解决方案

使用 hexdump 的 -C 选项获得规范的十六进制输出。

```
$ hexdump -C filename
00000000  4c 69 6e 65 20 31 0a 4c 69 6e 65 20 32 0a 0a 4c |Line
1.Line 2..L|
00000010  69 6e 65 20 34 0a 4c 69 6e 65 20 35 0a 0a       |ine
4.Line 5..|
0000001e
$
```

例如，nl 在其输出中使用了空格（ASCII 20），然后是行号，接着是制表符（ASCII 09）。

```
$ nl -ba filename | hexdump -C
00000000  20 20 20 20 20 31 09 4c  69 6e 65 20 31 0a 20 20 |
1.Line 1.  |
00000010  20 20 20 32 09 4c 69 6e  65 20 32 0a 20 20 20 20 |
2.Line 2.    |
00000020  20 33 09 0a 20 20 20 20  20 34 09 4c 69 6e 65 20 | 3..
4.Line |
00000030  34 0a 20 20 20 20 20 35  09 4c 69 6e 65 20 35 0a |4.
5.Line 5.|
00000040  20 20 20 20 20 36 09 0a                          |
6..|
00000048
$
```

15.8.3 讨论

hexdump 是一款 BSD 实用工具，很多 Linux 发行版中也能看到它的身影。其他系统（尤其是 Solaris）默认没有安装该工具。你可以使用八进制转储命令 od，但它一次只能输出一种格式，而且地址（左列）为八进制，而非十六进制。

```
$ nl -ba filename | od -x
0000000 2020 2020 3120 4c09 6e69 2065 0a31 2020
0000020 2020 3220 4c09 6e69 2065 0a32 2020 2020
0000040 3320 0a09 2020 2020 3420 4c09 6e69 2065
0000060 0a34 2020 2020 3520 4c09 6e69 2065 0a35
0000100 2020 2020 3620 0a09
0000110

$ nl -ba filename | od -tx1
0000000 20 20 20 20 20 31 09 4c 69 6e 65 20 31 0a 20 20
```

```
0000020 20 20 20 32 09 4c 69 6e 65 20 32 0a 20 20 20 20
0000040 20 33 09 0a 20 20 20 20 20 34 09 4c 69 6e 65 20
0000060 34 0a 20 20 20 20 20 35 09 4c 69 6e 65 20 35 0a
0000100 20 20 20 20 20 36 09 0a
0000110

$ nl -ba filename | od -c
0000000                      1  \t   L   i   n   e       1  \n
0000020               2  \t   L   i   n   e       2  \n
0000040       3  \t  \n                       4  \t   L   i   n
e
0000060   4  \n                       5  \t   L   i   n   e
5  \n
0000100                      6  \t  \n
0000110
```

另外还有一个简单的 Perl 脚本，没准也管用。

```
$ ./hexdump.pl filename

       /0 /1 /2 /3 /4 /5 /6 /7 /8 /9/ A /B /C /D /E /F
0123456789ABCDEF
0000 : 4C 69 6E 65 20 31 0A 4C 69 6E 65 20 32 0A 0A 4C  Line
1.Line 2..L
0010 : 69 6E 65 20 34 0A 4C 69 6E 65 20 35 0A 0A        ine 4.Line
5..
```

15.8.4 参考

- man hexdump
- man od
- A.25 节

15.9 使用bash的网络重定向

15.9.1 问题

你需要发送或接受一些非常简单的网络流量，但又没有安装 netcat 这类工具。

15.9.2 解决方案

如果你的 bash 版本是 2.04+，编译时使用了 --enable-net-redirections 选项（默认），那么只用 bash 就行了。以下示例也可用于 15.10 节。

```
$ exec 3<> /dev/tcp/checkip.dyndns.org/80
$ echo -e "GET / HTTP/1.0\n" >&3
$ cat <&3
HTTP/1.1 200 OK
Content-Type: text/html
Server: DynDNS-CheckIP/1.0
Connection: close
Cache-Control: no-cache
Pragma: no-cache
Content-Length: 105

<html><head><title>Current IP Check</title></head>
<body>Current IP Address: 72.NN.NN.225</body></html>

$ exec 3<> /dev/tcp/checkip.dyndns.org/80
$ echo -e "GET / HTTP/1.0\n" >&3
$ egrep --only-matching 'Current IP Address: [0-9.]+' <&3
Current IP Address: 72.NN.NN.225
$
```

> Debian 及其衍生发行版（如 Ubuntu）在 bash 4.0 之前都是用 --disable-net-redirections 编译的，因此本实例不适用于这些版本。

15.9.3 讨论

如 15.12 节所述，可以使用 exec 永久地重定向当前 shell 会话中的文件句柄，因此第一个命令将输入和输出设置在文件句柄 3 上。第二行向先前命令中已经定义好的 Web 服务器路径发送了一个普通的 HTTP GET 方法。注意，用户代理在 Web 服务器端会显示为“-”，这会导致产生“flagged User Agent”警告。第三个命令只是显示结果。

TCP 和 UDP 都没有问题。下面给出了一种简单的方法，可以向远程服务器发送 syslog 消息（但在生产环境中，我们还是推荐选择实用

工具 logger，它对用户友好得多，也更稳健）。

```
echo "<133>${0##*/}[$$]: Test syslog message from bash" \
  > /dev/udp/loghost.example.com/514
```

因为 UDP 是无连接的，所以这个其实要比之前的 TCP 示例更易用。<133> 是根据 RFC 3164 计算得出的 local0.notice 的 syslog **优先级**。具体细节可参见该 RFC 文档的 4.1.1 节和 logger 的手册页。$0 是程序名称，${0##*/} 则是“基本名称”，$$ 是当前程序的进程 ID。对于登录 shell，名称就是 -bash。

15.9.4 参考

- man logger
- RFC 3164
- 15.10 节
- 15.12 节
- 15.14 节
- bash 文档

15.10 查找自己的IP地址

15.10.1 问题

你需要知道自己所用机器的 IP 地址。

15.10.2 解决方案

不存在什么一劳永逸的好方法，我们得分情况考虑。

首先，可以解析 ifconfig 的输出，从中找出 IP 地址。例 15-2 中的命令要么返回第一个**非环回** IP 地址，要么什么都不返回（如果没有配置或激活接口的话）。

例 15-2 ch15/finding_ipas

```
# 实例文件：finding_ipas

# 使用awk、cut、head解析IPv4地址
$ /sbin/ifconfig -a | awk '/(cast)/ { print $2 }' | cut -d':' -f2
| head -1

# 用Perl解析IPv4，图个乐子而已
$ /sbin/ifconfig -a | perl -ne 'if ( m/^\s*inet (?:addr:)?
([\d.]+).*?cast/ )
> { print qq($1\n); exit 0; }'

# 用awk、cut、head解析IPv6地址
$ /sbin/ifconfig -a | egrep 'inet6 addr: |address: ' | cut -d':' -
f2- \
    | cut -d'/' -f1 | head -1 | tr -d ' '

# 用Perl解析IPv6，图个乐子而已
$ /sbin/ifconfig -a | perl -ne 'if
> ( m/^\s*(?:inet6)? \s*addr(?:ess)?: ([0-9A-Fa-f:]+)/ ) { print
qq($1\n);
> exit 0; }'
```

其次，可以获得主机名，然后将其反向解析为 IP 地址。这种方法往往靠不住，因为现今系统（尤其是工作站）的主机名可能不完整或不正确，也可能位于动态网络，无法完成正确的反向解析。因此存在一定风险，需要经过充分测试。

```
host $(hostname)
```

如果前面提到的程序你一个都没有，但使用的是 2.04+ 版本的 bash，而且编译时使用了选项 --enable-net-redirections（在 Debian 及其衍生发行版中，bash 4.0 之前的版本并未使用该编译选项），你也可以依靠 bash（详见 15.9 节）。

```
$ exec 3<> /dev/tcp/checkip.dyndns.org/80
$ echo -e "GET / HTTP/1.0\n" >&3
$ cat <&3
HTTP/1.1 200 OK
Content-Type: text/html
Server: DynDNS-CheckIP/1.0
Connection: close
Cache-Control: no-cache
Pragma: no-cache
Content-Length: 105
```

```
<html><head><title>Current IP Check</title></head>
<body>Current IP Address: 96.245.41.129</body></html>

$ exec 3<> /dev/tcp/checkip.dyndns.org/80
$ echo -e "GET / HTTP/1.0\n" >&3
$ egrep --only-matching 'Current IP Address: [0-9.]+' <&3
Current IP Address: 72.NN.NN.225
$
```

15.10.3 讨论

第一个解决方案中的 awk 和 Perl 代码很有意思，因为我们在这里会注意到操作系统的差异。但结果发现，我们感兴趣的那些行全都包含 Bcast 或 broadcast（要么是 inet6addr:or address:），只要得到这些行，剩下的事就是从中解析出需要的字段了。当然，由于 Linux 采用了不同的格式，这增加了难度，但我们一样能搞定。

并非所有系统使用 ifconfig 时都要求指定路径（如果你不是 root）或 -a 选项，但指定绝对没错，因此最好还是使用 /sbin/ifconfig -a。

以下是一些出自不同机器的 ifconfig 的输出示例。

```
# Linux
$ /sbin/ifconfig
eth0      Link encap:Ethernet HWaddr 00:C0:9F:0B:8F:F6
          inet addr:192.168.99.11 Bcast:192.168.99.255
Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:33073511 errors:0 dropped:0 overruns:0
frame:827
          TX packets:52865023 errors:0 dropped:0 overruns:1
carrier:7
          collisions:12922745 txqueuelen:100
          RX bytes:2224430163 (2121.3 Mb) TX bytes:51266497 (48.8
Mb)
          Interrupt:11 Base address:0xd000

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:659102 errors:0 dropped:0 overruns:0 frame:0
          TX packets:659102 errors:0 dropped:0 overruns:0
```

```
carrier:0
          collisions:0 txqueuelen:0
          RX bytes:89603190 (85.4 Mb) TX bytes:89603190 (85.4 Mb)

$ /sbin/ifconfig
eth0      Link encap:Ethernet HWaddr 00:06:29:33:4D:42
          inet addr:192.168.99.144 Bcast:192.168.99.255
Mask:255.255.255.0
          inet6 addr: fe80::206:29ff:fe33:4d42/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:1246774 errors:14 dropped:0 overruns:0
frame:14
          TX packets:1063160 errors:0 dropped:0 overruns:0
carrier:5
          collisions:65476 txqueuelen:1000
          RX bytes:731714472 (697.8 MiB) TX bytes:942695735 (899.0
MiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1 Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING MTU:16436 Metric:1
          RX packets:144664 errors:0 dropped:0 overruns:0 frame:0
          TX packets:144664 errors:0 dropped:0 overruns:0
carrier:0
          collisions:0 txqueuelen:0
          RX bytes:152181602 (145.1 MiB) TX bytes:152181602 (145.1
MiB)

sit0      Link encap:IPv6-in-IPv4
          inet6 addr: ::127.0.0.1/96 Scope:Unknown
          UP RUNNING NOARP MTU:1480 Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:101910 dropped:0 overruns:0
carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 b) TX bytes:0 (0.0 b)

# NetBSD
$ /sbin/ifconfig -a
pcn0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        address: 00:0c:29:31:eb:19
        media: Ethernet autoselect (autoselect)
        inet 192.168.99.56 netmask 0xffffff00 broadcast
192.168.99.255
        inet6 fe80::20c:29ff:fe31:eb19%pcn0 prefixlen 64 scopeid
0x1
lo0: flags=8009<UP,LOOPBACK,MULTICAST> mtu 33196
        inet 127.0.0.1 netmask 0xff000000
        inet6 ::1 prefixlen 128
```

```
        inet6 fe80::1%lo0 prefixlen 64 scopeid 0x2
ppp0: flags=8010<POINTOPOINT,MULTICAST> mtu 1500
ppp1: flags=8010<POINTOPOINT,MULTICAST> mtu 1500
sl0: flags=c010<POINTOPOINT,LINK2,MULTICAST> mtu 296
sl1: flags=c010<POINTOPOINT,LINK2,MULTICAST> mtu 296
strip0: flags=0 mtu 1100
strip1: flags=0 mtu 1100

# OpenBSD, FreeBSD
$ /sbin/ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 33224
        inet 127.0.0.1 netmask 0xff000000
        inet6 ::1 prefixlen 128
        inet6 fe80::1%lo0 prefixlen 64 scopeid 0x5
le1: flags=8863<UP,BROADCAST,NOTRAILERS,RUNNING,SIMPLEX,MULTICAST>
mtu 1500
        address: 00:0c:29:25:df:00
        inet6 fe80::20c:29ff:fe25:df00%le1 prefixlen 64 scopeid
0x1
        inet 192.168.99.193 netmask 0xffffff00 broadcast
192.168.99.255
pflog0: flags=0<> mtu 33224
pfsync0: flags=0<> mtu 2020

# Solaris
$ /sbin/ifconfig -a
lo0: flags=1000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4> mtu 8232
index 1
        inet 127.0.0.1 netmask ff000000
pcn0: flags=1004843<UP,BROADCAST,RUNNING,MULTICAST,DHCP,IPv4> mtu
1500 index 2
        inet 192.168.99.159 netmask ffffff00 broadcast
192.168.99.255

# Mac
$ /sbin/ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 16384
        inet 127.0.0.1 netmask 0xff000000
        inet6 ::1 prefixlen 128
        inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
gif0: flags=8010<POINTOPOINT,MULTICAST> mtu 1280
stf0: flags=0<> mtu 1280

en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu
1500
        inet6 fe80::20d:93ff:fe65:f720%en0 prefixlen 64 scopeid
0x4
        inet 192.168.99.155 netmask 0xffffff00 broadcast
192.168.99.255
        ether 00:0d:93:65:f7:20
```

```
        media: autoselect (100baseTX <half-duplex>) status: active
        supported media: none autoselect 10baseT/UTP <half-duplex>
10baseT/UTP <fullduplex>
10baseT/UTP <full-duplex,hw-loopback> 100baseTX <half-duplex>
100baseTX
<full-duplex> 100baseTX <full-duplex,hw-loopback>
fw0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu
2030
    lladdr 00:0d:93:ff:fe:65:f7:20
        media: autoselect <full-duplex> status: inactive
        supported media: autoselect <full-duplex>
```

15.10.4 参考

- man awk
- man curl
- man cut
- man head
- man lynx
- man perl
- man wget
- 15.9 节
- 15.12 节

15.11 从另一台机器获取输入

15.11.1 问题

你的脚本需要从另一台机器获取输入，也许是为了检查某个文件是否存在，或者检查某个进程是否在运行。

15.11.2 解决方案

使用 SSH，同时配合公钥和命令替换。首先，按照 14.21 节中描述的方法设置好无密码的 SSH。接下来，指定要通过 SSH 执行的命令，输出脚本所需要的输入。然后，简单地使用命令替换即可（参见例 15-3）。

例 15-3 ch15/command_substitution

```
#!/usr/bin/env bash
# 实例文件：command_substitution

REMOTE_HOST='host.example.com'  # 要求指定
REMOTE_FILE='/etc/passwd'       # 要求指定
SSH_USER='user@'                # 可选，如果不用，则设置为''
#SSH_ID='-i ~/.ssh/foo.id'      # 可选，如果不用，则设置为''
SSH_ID=''

result=$(
    ssh $SSH_ID $SSH_USER$REMOTE_HOST \
      "[ -r $REMOTE_FILE ] && echo 1 || echo 0"
) || { echo "SSH command failed!" >&2; exit 1; }
if [ $result = 1 ]; then
    echo "$REMOTE_FILE present on $REMOTE_HOST"
else
    echo "$REMOTE_FILE not present on $REMOTE_HOST"
fi
```

15.11.3 讨论

这里有几处操作值得留意。首先，注意 $SSH_USER 和 $SSH_ID 是如何工作的。这两个变量有值的时候就能发挥作用；如果没值，则为空，并被忽略。这就允许我们抽取出代码中的值，将其放入配置文件，将代码放入函数，或者两者兼而有之。

```
# 插入有值的变量：
ssh -i ~/.ssh/foo.id user@host.example.com [...]

# 变量没有值：
ssh host.example.com [...]
```

接下来，设置要通过 SSH 执行的命令，以便总能产生输出（0 或 1），然后检查 $result 是否为空。这是确保 SSH 命令运行的一种方法（参见 4.4 节）。如果 $result 为空，用 {} 代码块将命令划分成组，输出错误信息并退出。但因为 SSH 命令无论怎样都会有输出，所以我们必须测试 $result 的值，不能仅仅使用 if [$result];then。

如果没有使用代码块，那就只能在 SSH 命令返回为空的 $result 时发出警告，但我们始终还是会退出。回头再读读代码，搞明白为什么，因为这个地方很容易出错。与此类似，如果用 () 子 shell 代替 {} 代码块，结果只会事与愿违，原因在于 exit 1 退出的是子 shell，而不是整个脚本。即便 SSH 命令失败，脚本也会继续往下执行。只看代码的话，几乎找不出什么问题，但调试起来就棘手了。

最后的测试部分也可以写成如下所示的代码。

```
[ $result = 1 ] && echo "$REMOTE_FILE present on $REMOTE_HOST" \
                || echo "$REMOTE_FILE not present on $REMOTE_HOST"
```

至于使用哪种形式，这取决于你自己的风格以及各种情况下要执行的语句数量。在本例中，两种形式都行。

最后，我们还仔细对代码进行了格式化，以便每行不至于太长，同时仍然保持了代码的可读性并清晰地表达出我们的意图。

15.11.4 参考

- 2.14 节
- 4.4 节
- 14.21 节
- 17.20 节
- 17.21 节

15.12 在脚本运行期间重定向输出

15.12.1 问题

你想重定向整个脚本的输出，但又不想修改所有的 echo 或 printf 语句。

15.12.2 解决方案

exec 命令有一个不为众人所知的特性，该特性能够用来重定向 STDOUT 或 STDERR。

```
# 可选，保存"旧的"STDERR
exec 3>&2

# 将所有发往STDERR的输出重定向到错误日志文件
exec 2> /path/to/error_log

# 需要用到"全局"STDERR重定向的脚本代码可以写在此处

# 通过还原STDERR并关闭文件句柄3来停止重定向
exec 2>&3-
```

15.12.3 讨论

exec 通常会使用其参数所指定的命令来替换正在运行的 shell，从而破坏了原来的 shell。但如果没有指定命令，它则可以控制当前 shell 的重定向。它能做的不仅限于重定向 STDOUT 或 STDERR，但这两者是最常见的重定向目标。

15.12.4 参考

- help exec
- 15.9 节

15.13 解决“Argument list too long”错误

15.13.1 问题

在尝试执行涉及 shell 通配符扩展的操作时，碰到了“Argument list too long”错误。

15.13.2 解决方案

使用 xargs 命令（可能还要配合 find）来拆分参数列表。

对于简单的情况，使用 for 循环或 find 代替 ls。

```
$ ls /path/with/many/many/files/*e*
-/bin/bash: /bin/ls: Argument list too long
# 小demo, 两边的~仅用于演示
$ for i in ./some_files/*e*; do echo "~$i~"; done
~./some_files/A file with (parens)~
~./some_files/A file with [brackets]~
~./some_files/File with embedded
newline~
~./some_files/file with = sign~
~./some_files/file with spaces~
~./some_files/file with |~
~./some_files/file with:~
~./some_files/file with;~
~./some_files/regular_file~


$ find ./some_files -name '*e*' -exec echo ~{}~ \;
~./some_files~
~./some_files/A file with [brackets]~
~./some_files/A file with (parens)~
~./some_files/regular_file~
~./some_files/file with spaces~
~./some_files/file with = sign~
~./some_files/File with embedded
newline~
~./some_files/file with;~
~./some_files/file with:~
~./some_files/file with |~

$ for i in /path/with/many/many/files/*e*; do echo "$i"; done
[可行，但输出内容太多，不再逐一列出]


$ find /path/with/many/many/files/ -name '*e*'
[可行，但输出内容太多，不再逐一列出]
```

这个例子在配合 echo 命令时没有问题，但如果将 "$i" 传给别的程序，尤其是其他一些 shell 构造，$IFS 和另外一些解析机制可能就要发挥作用了。GNU find 和 xargs 考虑到了这一点，前者可以使用 find -print0，后者可以使用 xargs -0。（别问了，我们也不知道为什么两者还不一致，一个是 -print0，另一个是 -0）。这些选项使得 find 使用空字符（文件名中不会出现这种字符）代替

空白字符作为输出记录分隔符，而 xargs 使用空字符作为输入记录分隔符。这样一来，就能正确地解析包含怪异字符的文件名了。

```
find /path/with/many/many/files/ -name '*e*' -print0 | xargs -0
proggy
```

15.13.3 讨论

注意，bash（以及 sh）的默认行为是原封不动地返回无法匹配的模式。这意味着，如果没有文件名能够匹配指定的通配符模式，for 循环最终会将 $i 设置为 ./some_files/*e*。你可以设置 shopt -s nullglob 选项，这样一来，无法匹配的文件名模式会扩展成空串，而不是按照原样保留下来。

你可能认为先前例子中的 for 循环解决方案也会碰到和 ls 命令同样的问题，但结果并非如此。Chet Ramey 告诉我们：

> ARG_MAX 限制了 exec* 系列系统调用的总空间需求，内核以此知道它必须分配的最大缓冲区。execve 的 3 个参数是：程序名称、参数向量和环境。
>
> ls 命令失败的原因是，execve 的参数所占用的字节数量超出了 ARG_MAX 的限制。for 循环成功的原因是，所有一切都是在内部搞定的：虽然生成并保存了整个文件列表，但从未调用过 execve。

注意，find 找到的文件数量可不会少，因为它默认向下递归进入所有的子目录，而 ls 不会这么做。某些版本的 find 包含 -maxdepth 选项，能够控制向下递归查找的深度。使用 for 循环也许会更容易些。

可以用 getconf ARG_MAX 命令查看你所在系统的限制。具体的值千差万别（也可以参看 getconf LINE_MAX;）。表 15-1 列出了一些示例。

根据 GNU Core Utilities FAQ，Linux 2.6.23+ 去除了此限制，不过有可能还会产生相关报告，你所用的特定发行版内核也许仍是老样子。

表15-1：系统限制

系统	ARG_MAX限制（字节）
HP-UX 11	2 048 000
Solaris（8、9、10）	1 048 320
NetBSD 2.0.2、OpenBSD 3.7、macOS	262 144
Linux（Red Hat、Debian、Ubuntu）	131 072
FreeBSD 5.4	65 536

15.13.4 参考

- GNU Core Utilities FAQ 中的问题 19
- 9.2 节

15.14 向syslog记录脚本日志

15.14.1 问题

你希望自己的脚本能向 syslog 记录日志。

15.14.2 解决方案

用 logger、Netcat 或者 bash 内建的网络重定向特性。

大多数系统默认安装了 logger，很容易用它向本地的 syslog 服务发送消息。

```
logger -p local0.notice -t ${0##*/}[$$] test message
```

但它无法向远程主机发送 syslog 消息。如果需要这么做，可以使用 bash。

```
echo "<133>${0##*/}[$$]: Test syslog message from bash" \
  > /dev/udp/loghost.example.com/514
```

或者 Netcat：

```
echo "<133>${0##*/}[$$]: Test syslog message from Netcat" | nc -w1
-u loghost 514
```

Netcat 被称为“TCP/IP 瑞士军刀”，通常没有默认安装。它还有可能会被某些安全策略作为黑客工具而加以禁止，不过 bash 的网络重定向特性也可以非常漂亮地实现同样的效果。有关 <133>${0##*/}[$$] 部分的详细讨论，参见 15.9 节。

15.14.3 讨论

logger 和 Netcat 拥有的特性要比这里讲到的多得多。具体细节参见各自的手册页。

15.14.4 参考

- man logger
- man nc
- 15.9 节

15.15 正确地使用logger

15.15.1 问题

你想利用 logger 工具使脚本能够发送 syslog 消息，但默认情况下，logger 并没有提供足够的有用信息。

15.15.2 解决方案

像下面这样使用 logger：

```
logger -t "${0##*/}[$$]" 'Your message here'
```

15.15.3 讨论

在我们看来，要是没使用 logger 的 -t 选项，即便算不上致命错误，起码也得接受警告。t 代表“标签”（tag），按照手册页中所言，它会“使用指定的标签来标记所记录的每一行内容”。换句话说，要是没有 -t 选项，你很难知道这些日志消息都是从哪来的。

${0##*/}[$$] 标签可能看起来像乱码，但查看 syslog 日志记录时经常会看到它。该标签只不过就是脚本的基本名称和位于方括号内的进程 ID（$$）而已。比较一下使用和不使用 -t 选项的 logger。

```
$ logger -t "${0##*/}[$$]" 'Your message here'
$ tail -1 /var/log/syslog
Oct 26 12:16:01 hostname yourscript[977]: Your message here
$ logger 'Your message here'
$ tail -1 /var/log/syslog
Oct 26 12:16:01 hostname Your message here
$
```

logger 还有其他一些值得注意的选项，你可以好好阅读一下相应的手册页，但要清楚，部分选项可能会随年代、版本、发行版而异，因此得考虑你的脚本是不是要大范围运行。例如，CentOS 5/6 版本中的 logger 就缺少 Debian/Ubuntu 版本中非常实用的选项 -n。

```
-n, --server server
    使用UDP代替内建的syslog例程向指定的远程syslog服务器写入
```

15.15.4 参考

- 5.20 节
- 11.10 节
- 15.14 节
- man logger

15.16 在脚本中发送电子邮件

15.16.1 问题

你希望脚本能够发送电子邮件，同时还可以添加附件。

15.16.2 解决方案

具体的解决方案取决于你的系统是否具有兼容的邮件程序（如 mail、mailx 或者 mailto）、已安装并正在运行的邮件消息传输代理（message transfer agent，MTA）以及正确配置的电子邮件环境。遗憾的是，单凭这些也未必总是足够，因此这些解决方案必须在你的环境中进行充分的测试。

第一种方法是编写代码来生成并发送邮件消息，如下所示。

```
# 简单的邮件消息
cat email_body | \
mail -s "Message subject" recipient1@example.com
recipient2@example.com
```

或者：

```
# 只包含附件
uuencode /path/to/attachment_file attachment_name | \
mail -s "Message Subject" recipient1@example.com
recipient2@example.com
```

或者：

```
# 包含附件和邮件正文
(cat email_body ; uuencode /path/to/attachment_file
attachment_name) | \
mail -s "Message Subject" recipient1@example.com
recipient2@example.com
```

实践中不会总这么简单。一方面，虽然可能已经安装过 uuencode，但 mail 及其相关程序未必如此，而且各自的功能可能也有所不同。在某些情况下，mail 和 mailx 甚至是同一个程序，通过硬链接或软链接关联在一起。在生产环境中，为了实现可移植性，你得借助一些抽象。例如，mail 在 Linux 和 BSD 上可以正常使用，而 Solaris 则要求 mailx，因为它的 mail 不支持 -s 选项。有些 Linux 发行版（如 Debian）用的是 mailx，有些则没有。在例 15-4 中，我们根据主机名来选择邮件程序，根据你所在的环境，可能使用 uname -o 更切实际。

例 15-4 ch15/email_sample

```
# 实例文件：email_sample

# 定义了一些邮件设置。根据所在环境的要求使用case语句，
# 配合uname或hostname命令来调整设置
case $HOSTNAME in
    *.company.com     ) MAILER='mail'   ;;  # Linux和BSD
    host1.*           ) MAILER='mailx'  ;;  # Solaris、BSD以及某些
Linux
    host2.*           ) MAILER='mailto' ;;  # 方便（如果已安装）
esac
RECIPIENTS='recipient1@example.com recipient2@example.com'
SUBJECT="Data from $0"

[...]
# 使用echo、printf或here-document将邮件正文创建为文件或变量
# 根据需要，创建或修改$SUBJECT或$RECIPIENTS
[...]
( echo $email_body ; uuencode $attachment $(basename $attachment)
) \
  | $MAILER -s "$SUBJECT" "$RECIPIENTS"
```

需要指出的是，采用这种方式发送附件取决于你所使用的邮件客户端。Thunderbird（以及 Outlook）等现代客户端能够检测出经过 uuencode 编码的邮件消息并将其显示为附件。其他客户端可就未必

了。无论怎样，你都可以保存邮件并使用 uudecode 对其解码（uudecode 足够聪明，能够跳过消息部分，只处理附件），但这很麻烦。

在脚本中发送邮件的第二种方法是将任务转包给 cron 来完成。虽然 cron 的具体特性视系统而异，但有一处是相同的：cron 作业的所有输出都会邮寄给该作业的所有者或是由 MAILTO 变量定义的用户。你可以利用这点获得免费的电子邮件功能，前提是你的电子邮件基础设置一切正常。

对于通过 cron 运行的脚本（不仅如此，很多人认为这包括所有的脚本或者 Unix 工具），正确的设计方法是使其安安静静的，除非碰到警告或错误。如有必要，使用 -v 选项来允许启用更为详细的模式，但通过 cron 运行时不要采用这种形式，至少完成测试后就别这么做了。原因是：cron 会将所有的输出全都邮寄给你。如果每次运行脚本都收到来自 cron 的电子邮件，你很快就懒得再看这些邮件了。但如果脚本一直都是安安静静的（除非碰上麻烦），那么等到出现问题时你准能及时注意到，这才是理想状态。

15.16.3 讨论

注意，mailto 针对 mail 升级了多媒体和 MIME 处理能力，这样一来，发送附件时就不需要 uuencode 了，但 mailto 的使用率没有 mail 或 mailx 那么高。如果还不行，也可以使用 elm 或 mutt 代替 mail、mailx 或 mailto，不过相对于 mail*，前者很少会被默认安装。另外，这些程序的某些版本还提供了 -r 选项，可以在需要时提供返回地址。mutt 的 -a 选项可以不费吹灰之力地发送附件。

```
cat "$message_body" | mutt -s "$subject" -a "$attachment_file"
"$recipients"
```

mpack 是另一个值得一试的工具，不过极少会默认安装。可以检查系统的软件仓库或者下载其源代码。下列内容摘自 mpack 的手册页。

> mpack 程序对一个或多个 MIME 邮件消息内的命名文件进行编码。得到的邮件消息会被邮寄给一个或多个收件人、写入一个或多个命名文件，或者发布在多个新闻组。

Nelson H. F. Beebe 与 Arnold Robbins 合著的 *Classic Shell Scripting*（O'Reilly 出版）一书的第 8 章中给出了另一种处理邮件客户端名称和位置差异的方法，例 15-5 对其进行了重现。

例 15-5 ch15/email_sample_css

```
# 实例文件：email_sample_css
# 选自Classic Shell Scripting一书的第8章

for MAIL in /bin/mailx /usr/bin/mailx /usr/sbin/mailx
/usr/ucb/mailx /bin/mail \
/usr/bin/mail; do
    [ -x $MAIL ] && break
done
[ -x $MAIL ] || { echo 'Cannot find a mailer!' >&2; exit 1; }
```

uuencode 是一种将二进制数据转换为 ASCII 文本，以便在不支持二进制的链路进行传输的旧方法，Internet 和万维网出现前的大部分网络是这种链路。我们敢说目前仍有部分此类链路存在，就算你从未碰到过，但能将附件转换成现代邮件客户端可以识别的 ASCII 格式仍有用武之地。另请参见 uudecode 和 mimeencode。注意，经过 uuencode 编码的文件比其二进制版本大了约三分之一，你可能需要在编码前先压缩文件。

除了不同的前端邮件用户代理（mail user agent，MUA）程序（如 mail 和 mailx），电子邮件的问题还在于有大量的活动部件必须协同工作。垃圾邮件问题恶化了这种情况：邮件管理员不得不非常严格地封锁邮件服务器，这很容易就会影响到你的脚本。这里我们只能说请全面测试解决方案，在必要时与系统和邮件管理员联系。

你可能会在某些面向工作站的 Linux 发行版（如 Ubuntu）中碰到另一个问题，这类发行版默认没有安装或运行 MTA，因为它们默认用户会使用 Evolution 或 Thunderbird 等功能完善的 GUI 客户端。这样的话，命令行 MUA 和通过 cron 发送邮件都不管用。如果有需要，请向你所用发行版的支持小组寻求帮助。

cron 有 MTA 就够了

考虑到安全攻击面（security attack surface）、垃圾邮件以及一般可维护性，我们可以提出一个很好的论断：唯一应该运行完整MTA 的服务器就是专用邮件服务器。那如何从其他非邮件服务器节点发送邮件呢？ Debian 及其衍生发行版中安装了 nullmailer 等软件包，Red Hat 及其衍生发行版中安装了 SSMTP 等软件包。

虽然这些软件的配置和实现各不相同，但思路是相同的：“cron有MTA 就够了”。我们鼓励大家安装其中一种或其他类似的软件，因为经常可以通过 cron 消息捕获到错误和误配置信息，这实在令人惊喜。即便已经配备全套的监视方案，允许节点能够发送电子邮件也非常有用。

简单的 nullmailer 配置如下所示。

/etc/nullmailer/adminaddr

```
it@example.com
```

/etc/nullmailer/defaultdomain

```
example.com
```

可选的：/etc/nullmailer/pausetime

```
3600
```

/etc/nullmailer/remotes

```
mail.example.com smtp --port=587
```

简单的 SSMTP 配置（/etc/ssmtp/ssmtp.conf）如下所示。

```
root=it@example.com
mailhub=mail.example.com:587
```

尽管我们前面已经说过，但你并不希望所有的节点都能满世界发送电子邮件！这简直就是自找麻烦。我们假定你的防火墙已经配置妥当，加入了仅允许专用电子邮件服务器向外界发送邮件的出站规则。你还需要将规则写入日志并监视日志，那些突然开始随处发送大量电子邮件的节点肯定得仔细检查，其中必有蹊跷，可能就是被病毒感染了。而且，这并非总是那种通过定期监视 CPU 使用、磁盘空间等情况就能发现的事情。

15.16.4 参考

- man mail
- man mailx
- man mailto
- man mutt
- man uuencode
- man cron
- man 5 crontab

15.17 用阶段自动化进程

15.17.1 问题

你有一个漫长的作业或进程得实现自动化，但它可能需要手动干预并在整个过程的不同节点上重启。你可以使用 GOTO 四处跳转，但 bash 并没有此功能。

15.17.2 解决方案

使用 case 语句将脚本拆分成若干部分或**阶段（phase）**。

首先，使用例 15-6 定义的标准化方法（取自 3.6 节）从用户那里获得答案。

例 15-6 ch03/func_choice.1

```
# 实例文件：func_choice.1

# 由用户做出选择并返回标准答案。默认值的处理方式以及
# 接下来怎么做取决于主代码中choice函数之后的if/then分支
# 调用方式：choice <prompt>
# 例如：choice "Do you want to play a game?"
# 返回：全局变量CHOICE
function choice {

    CHOICE=''
    local prompt="$*"
    local answer

    read -p "$prompt" answer
    case "$answer" in
        [yY1] ) CHOICE='y';;
        [nN0] ) CHOICE='n';;
        *     ) CHOICE="$answer";;
    esac
} # 函数choice定义完毕
```

然后按照例 15-7 中给出的方法设置各个阶段。

例 15-7 ch15/using_phases

```
# 实例文件：using_phases

# 主循环
until [ "$phase" = "Finished." ]; do

    case $phase in

        phase0 )
            ThisPhase=0
            NextPhase="$(( $ThisPhase + 1 ))"
            echo '##############################################'
            echo "Phase$ThisPhase = Initialization of FooBarBaz
build"
            # 只能在新的构建周期开始时进行初始化的内容出现在此处
# ...
            echo "Phase${ThisPhase}=Ending"
            phase="phase$NextPhase"
        ;;


# ...
```

```
        phase20 )
        ThisPhase=20
            NextPhase="$(( $ThisPhase + 1 ))"
            echo '############################################'
            echo "Phase$ThisPhase = Main processing for FooBarBaz
build"


# ...


            choice "[P$ThisPhase] Do we need to stop and fix
anything? [y/N]: "
            if [ "$choice" = "y" ]; then
                echo "Re-run '$MYNAME phase${ThisPhase}' after
handling this."
                exit $ThisPhase
            fi

            echo "Phase${ThisPhase}=Ending"
            phase="phase$NextPhase"
        ;;


# ...


        * )
            echo "What the heck?!? We should never get HERE! Gonna
croak!"
            echo "Try $0 -h"
            exit 99
            phase="Finished."
        ;;
    esac
    printf "%b" "\a"    # 响铃
done
```

15.17.3 讨论

由于退出码最多只能达到 255，因此 `exit $ThisPhase` 就将阶段数量限制在了这个数字上。`exit 99` 更是对此做了进一步限制（虽然你可以很轻松进行调整）。如果你需要的阶段数量超过 254（255 为错误码），那就只能说抱歉了。要么采用其他退出码方案，要么将多个脚本串联在一起。

或许你应该设置用法或摘要例程，列出各个阶段。

```
Phase0 = Initialization of FooBarBaz build
...
Phase20 = Main processing for FooBarBaz build
...
Phase28 ...
```

你可以用类似 `grep 'Phase$ThisPhase' my_script` 的命令找出代码中的大部分文本。

你可能还想将日志写入本地平面文件、syslog 或其他机制。这种情况下，可以定义一个类似于 logmsg 的函数，在代码中的适当位置使用。该函数可以很简单。

```
function logmsg {
    # 将包含时间戳的日志消息写入屏幕和日志文件
    # 注意加上tee -a
    # printf "%b" "$(date '+%Y-%m-%d %H:%M:%S'): $*" | tee -a
$LOGFILE
    printf "%(%Y-%m-%d %H:%M:%S)T: %b\n" -1 "$*" | tee -a $LOGFILE
} # 函数logmsg定义完毕
```

这个函数使用了较新的 printf 格式，该格式支持时间和日期值。如果使用的 shell 版本有些旧（版本 4 之前），可以改用注释中给出的 printf 语句。

也许你已经注意到了，这个较长的脚本违反了我们平常所遵循的“没事就别吭声”的准则。不过它本来就是一个交互式脚本，这么做是没问题的。

15.17.4 参考

- 3.5 节
- 3.6 节
- 11.10 节
- 15.14 节

15.18 一心二用

15.18.1 问题

命令管道是单向的，每个进程都会将输出写入下一个进程的输入。两个进程能不能实现双向通信，各自将对方的输出作为自己的输入？

15.18.2 解决方案

是的！从 bash 4 开始，coproc 命令就可以实现这一目标。

在例 15-8 这个简单的示例中，我们使用 bc 程序（一种任意精度计算器语言）作为协程，从而允许 bash 将算式发送给 bc 并读取计算结果。这是一种赋予 bash 浮点计算能力的方法，尽管这里只是用它作为 coproc 命令的示例。

> 注意，要想使用协程，编译 bash 时必须加入 --enable-coprocesses 选项。这是默认选项，但有些软件包可能并没有此选项。

例 15-8 ch15/fpmath

```
# 实例文件：fpmath
# 用coproc实现浮点数运算

# 初始化协程
#   在调用fpmath之前先调用该函数
function fpinit ()
{
    coproc /usr/bin/bc

    bcin=${COPROC[1]}
    bcout=${COPROC[0]}
    echo "scale=5" >& ${bcin}
}

# 进行浮点数运算
#   先将参数发送给bc
```

```
#    然后读取计算结果
function fpmath()
{
    echo "$@" >& ${bcin}
    if read -t 0.25 -u ${bcout} responz
    then
        echo "$responz"
    fi
}
############################
# 主体部分

fpinit

while read aline
do
    answer=$(fpmath "$aline")
    if [[ -n $answer ]]
    then
        echo $answer
    fi
done
```

15.18.3 讨论

示例中定义了两个函数：fpinit 和 fpmath。fpinit 的目的在于设置协程。fpmath 的目的在于完成浮点数运算，这是通过向协程发送请求并读取结果实现的。为了演示这两个函数，我们编写了一个 while 循环，该循环会提示用户输入，然后将输入发送到协程并读取计算结果。

coproc 在当前 shell 进程之外执行命令（或一系列命令）[3]。在这个例子中，我们指定了 /usr/bin/bc（并不要求使用完整路径，shell 会搜索 $PATH 来查找指定命令）。另外，coproc 创建了两个管道，一个连接到命令的标准输出，另一个连接到命令的标准输入。这两个连接默认保存在名为 COPROC 的 shell 数组中。索引为 0 的数组元素中保存的是进程的输出文件描述符；索引为 1 的数组元素中保存的是进程的输入文件描述符。

[3]命令是在子 shell 中异步执行的。——译者注

对系统程序员而言，这看起来似乎有些落后，但是别忘了，协程的输出可以作为调用进程（calling process）（shell 脚本）的输入，反之亦然。为了使用起来更为清晰，我们将它们赋给描述其用法的变量。我们选择 $bcin 来保存用于向 bc 命令发送输入的文件描述符，$bcout 则保存用于读取输出的文件描述符。

我们在 fpmath 函数中用到了这些文件描述符。为了将算式发送给 bc 进程，输出算式的文本（如 "3.4 * 7.52"），然后将其重定向到输入文件描述符。在本例中，这意味着重定向到 bcin。为了从 bc 获取计算结果，我们用到了 read 命令，该命令的选项（-u）可以指定要从中读取的文件描述符。这里也就是 $bcout。

我们还使用了 read 命令的 -t 选项。该选项设置了一个超时值，超时后 read 命令就会返回，有可能在这期间没有读取到任何内容。这里我们使用它是因为并非所有 bc 的输入都会产生输出。（例如，"x = 5" 会将数值 5 保存在变量 x 中，却不会有任何输出。）支持 coproc 命令的新版本 bash 也同样支持小数形式的超时值。旧版本只允许使用整数。

15.18.4 参考

- man bash
- help coproc

15.19 在多个主机上执行SSH命令

15.19.1 问题

你需要通过 SSH 在多个主机上执行命令。

15.19.2 解决方案

将要执行的 SSH 命令放进 for 循环。

```
$ for host in host1 host2 host3; do echo -n "On $host, I am: " ;
> ssh $host 'whoami' ; done
On host1, I am: root
On host2, I am: jp
On host3, I am: jp
$
```

15.19.3 讨论

这看起来非常简单，没有任何问题，但是有几点需要牢记。

首先，所有底层的联网、防火墙、DNS 等相关方面必须都能正常运行。

其次，尽管并不是非得这么做，但如果使用 SSH 密钥，那么会方便得多，因此不妨阅读 14.21 节。

最后，你很快就会碰上 SSH 命令中的引用问题。例如，思考下列语句：

```
$ for host in host{1..3};
> do echo "$host:" ;
> ssh $host 'grep "$HOSTNAME" /etc/hosts' ;
> done
```

这很简单：我们将 ssh 命令放进单引号，以便本地 bash 不会对其进行插值，然后出于清晰的目的，将 grep 的参数放入双引号内，不过不是非得做这一步。但是，如果有一些变量要由本地 bash 进行插值，而另一些变量则必须由远程 bash 处理，那该怎么办呢？或者说，如果我们需要 grep 使用单引号呢？

解决该问题的方法是将 ssh 命令放入双引号，然后将远端所需要的所有变量或双引号全部转义，但这样写立刻就变难看了。

```
$ for host in host{1..3};
> do ssh $host "echo \"Local '$host' is remote '\$HOSTNAME'\"";
> done
Local 'host1' is remote 'host1'
Local 'host2' is remote 'host2'
Local 'host3' is remote 'host3'
$
```

需要指出的是，在 OpenSSH 的配置文件中，你可以实现一些令人称奇的操作，这值得花些时间来学习，但遗憾的是，这远远超出了本书的讨论范围。

另外还要指出，虽然这种技术很方便，但你最好还是学习并使用真正的配置管理系统来完成这类任务。我们特别喜欢 Ansible，不过还有很多其他选择。

15.19.4 参考

- 14.21 节
- 14.22 节
- `man ssh`
- `man ssh_config`
- Ansible 官网

第 16 章　bash的配置与自定义

你愿意在一个没法按照自己喜好调整物品的环境中工作吗？想象一下，不能调节椅子的高度，或者被迫走一大段路才能到达餐厅，而这仅仅因为某人认为这才是“正路”。人们不可能长期接受这种死板模式。然而，这却是大多数用户期望并且能够接受的计算环境。但如果你习以为常地认为用户界面缺乏灵活性、一成不变，现在是时候改改观念了。bash 允许你自定义设置，使其为你所用，而不是处处作对。

bash 提供了极其强大而灵活的环境。这种灵活性的一部分体现在自定义的程度。如果不是 Unix 的日常用户，或者已经习惯了不那么灵活的环境，那么你可能还没有意识到自己能做些什么。本章将介绍如何配置 bash 以适合个人需求和风格。如果你觉得 Unix 的 cat 命令的名字莫名其妙（大多数非 Unix 用户不会对此有异议）[1]，可以定义一个别名，给它换个名字。如果经常要用到一些命令，可以为其指定缩写，甚至让它迎合你习惯性的拼写错误（例如，用“mroe”代表 more 命令）。你可以创建自己的命令，其用法与标准 Unix 命令一模一样。也可以更改命令行提示符，在其中加入有用的信息（如当前目录）。除此之外，还可以更改 bash 的行为方式，例如，要求其不区分大小写。通过些许简单的 bash 调校（尤其对 readline 而言），由此带来的生产力提升绝对会令你既惊讶又欣喜。

[1] cat 是 concatenate 的缩写。——译者注

有关 bash 自定义及配置的更多信息，参见 Cameron Newham 所著的 *Learning the bash Shell, 3rd Edition*（O’Reilly 出版）一书的第 3 章。

16.1　bash启动选项

16.1.1　问题

你想搞清楚 bash 的各种启动选项，但是 bash --help 也没能帮上忙。

16.1.2 解决方案

除了 bash --help，还可以试试 bash -c "help set" 和 bash -c help，如果正在使用 bash shell，直接使用 help set 和 help 就行了。

16.1.3 讨论

bash 有时可以采用不同的方式设置相同的选项，这就是一个例子。你可以在启动时设置某个选项（如 bash -x），随后使用 set +x 以交互形式关闭该选项。

16.1.4 参考

- 附录 A
- 19.12 节

16.2 自定义提示符

16.2.1 问题

默认的 bash 提示符通常以 $ 结尾，但信息不够丰富，传达不出太多的东西。你希望能够自定义提示符，以展现出更多实用的信息。

16.2.2 解决方案

根据需要自定义 $PS1 和 $PS2 变量。

默认的提示符随系统而异。bash 本身会显示其主版本号和次版本号（\s-\v\$），例如，bash-3.00$。不过，你的操作系统可能有自己的默认提示符，例如，有些版本的 Fedora 使用的是 [*user*@*host*~]$ ([\u@\h\W]\$)。这里我们给出了 8 种基本提示符和 3 种花哨的提示符。

01. 基本提示符

以下这 8 种更实用的提示符适用于 bash 1.14.7 及以上版本。如果有效 UID 为 0（root 用户），结尾的 \$ 显示为 #；否则显示为 $。

- 用户名 @ 主机名、日期和时间，以及当前工作目录。

```
$ export PS1='[\u@\h \d \A] \w \$ '
[jp@freebsd Wed Dec 28 19:32] ~ $ cd /usr/local/bin/
[jp@freebsd Wed Dec 28 19:32] /usr/local/bin $
```

- 用户名 @ 长格式主机名、ISO 8601 格式的日期和时间，以及当前工作目录的基本名称（\W）。

```
$ export PS1='[\u@\H \D{%Y-%m-%d %H:%M:%S%z}] \W \$ '
[jp@freebsd.jpsdomain.org 2005-12-28 19:33:03-0500] ~ $ cd
/usr/local/
[jp@freebsd.jpsdomain.org 2005-12-28 19:33:06-0500] local $
```

- 用户名 @ 主机名、bash 版本号，以及当前工作目录（\w）。

```
$ export PS1='[\u@\h \V \w] \$ '
[jp@freebsd 3.00.16] ~ $ cd /usr/local/bin/
[jp@freebsd 3.00.16] /usr/local/bin $
```

- 换行符、用户名 @ 主机名、基本 PTY、shell 层级、历史编号、换行符，以及完整的工作目录名称（$PWD）。

```
$ export PS1='\n[\u@\h \l:$SHLVL:\!]\n$PWD\$ '

[jp@freebsd ttyp0:3:21]
/home/jp$ cd /usr/local/bin/

[jp@freebsd ttyp0:3:22]
/usr/local/bin$
```

PTY 是你所连接的伪终端（Linux 术语）的编号。当你有多个会话并且试图跟踪各个会话时，这就能派上用场了。shell 层级是你所在子 shell 的深度。首次登录时，这个值是 1，随着不断地生成子进程（如 screen），该值也会不断增加，因此，运行 screen 之后，值就变成了 2。历史编号是当前命令在命令历史记录中的编号。

- 用户名 @ 主机名、上一个命令的退出状态，以及当前工作目录。注意，只要在提示符下执行任何命令，退出状态就会被重置（因此也就失去了作用）。

```
$ export PS1='[\u@\h $? \w \$ '
[jp@freebsd 0 ~ $ cd /usr/local/bin/
[jp@freebsd 0 /usr/local/bin $ true
[jp@freebsd 0 /usr/local/bin $ false
[jp@freebsd 1 /usr/local/bin $ true
[jp@freebsd 0 /usr/local/bin $
```

- 换行符、用户名 @ 主机名，以及 shell 当前管理的作业数。如果运行了大量的后台作业，又把这事给忘了，该提示符就能发挥作用了。

```
$ export PS1='\n[\u@\h jobs:\j]\n$PWD\$ '

[jp@freebsd jobs:0]
/tmp$ ls -lar /etc > /dev/null &
[1] 96461

[jp@freebsd jobs:1]
/tmp$
[1]+ Exit 1              ls -lar /etc >/dev/null

[jp@freebsd jobs:0]
/tmp$
```

- 换行符、用户名 @ 主机名、终端、shell 层级、历史编号、作业数量、bash 版本号，以及完整的工作目录。

```
$ export PS1='\n[\u@\h t:\l l:$SHLVL h:\! j:\j v:\V]\n$PWD\$
'

[jp@freebsd t:ttyp1 l:2 h:91 j:0 v:3.00.16]
/home/jp$
```

- 换行符、用户名 @ 主机名、代表终端的 T、代表 shell 层级的 L、代表命令编号的 C，以及 ISO 8601 格式的日期和时间。

```
$ PS1='\n[\u@\h:T\l:L$SHLVL:C\!:\D{%Y-%m-
%d_%H:%M:%S_%Z}]\n$PWD\$ '

[jp@freebsd:Tttyp1:L1:C337:2006-08-13_03:47:11_EDT]
/home/jp$ cd /usr/local/bin/
```

```
[jp@freebsd:Tttyp1:L1:C338:2006-08-13_03:47:16_EDT]
/usr/local/bin$
```

这属于那种让人非爱即恨的提示符。它清晰无误地显示了谁在什么时间什么地点做了什么，非常适合在回滚缓冲区（scrollback buffer）中通过简单地复制和粘贴来记录执行某项任务的步骤，但有些人认为这种提示符的形式太杂乱，容易分散注意力。

02. 花哨的提示符

下列 3 种花哨的提示符要么使用 ANSI 转义序列来显示颜色，要么在 xterm 中设置标题栏的内容，但要注意，这些做法未必总能奏效。

系统设置、xterm 仿真，以及 SSH 和 Telnet 客户端中都存在大量让人摸不着头脑的变量，它们都可能影响到这些提示符。

另外还要注意，转义序列应该出现在 \[和 \] 的内部，以此告知 bash 其中的字符不可打印。否则，bash（从技术上来说其实是 readline）会搞不清楚行的长度，在错误的位置上折行。

- 用户名 @ 主机名、浅蓝色的当前工作目录（实体书上看不出颜色）。

```
$ export PS1='\[\033[1;34m\][\u@\h:\w]\$\[\033[0m\] '
[jp@freebsd:~]$
[jp@freebsd:~]$ cd /tmp
[jp@freebsd:/tmp]$
```

- 在 xterm 的标题栏和提示符中同时显示用户名 @ 主机名、当前工作目录。如果你使用的不是 xterm，则可能会在提示符中产生一些乱七八糟的东西。

```
$ export PS1='\[\033]0;\u@\h:\w\007\][\u@\h:\w]\$ '
[jp@ubuntu:~]$
[jp@ubuntu:~]$ cd /tmp
[jp@ubuntu:/tmp]$
```

- 同时更新颜色和 xterm。

```
$ PS1='\[\033]0;\u@\h:\w\007\]\[\033[1;34m\][\u@\h:\w]\$\
[\033[0m\] '
[jp@ubuntu:~]$
```

```
[jp@ubuntu:~]$ cd /tmp
[jp@ubuntu:/tmp]$
```

为了让你在逐个尝试时免去烦人的输入，所有这些提示符都可以在本书的 GitHub 仓库中找到（文件 ./ch16/prompts）。例 16-1 展示了该文件的内容。

例 16-1 ch16/prompts

```
# 实例文件：prompts

# 用户名@短格式主机名、日期和时间，以及当前工作目录（CWD）：
export PS1='[\u@\h \d \A] \w \$ '

# 用户名@长格式主机名、ISO 8601格式的日期和时间、当前工作目录的基本名称
（\W）：
export PS1='[\u@\H \D{%Y-%m-%d %H:%M:%S%z}] \W \$ '

# 用户名@短格式主机名、bash版本号，以及当前工作目录（\w）：
export PS1='[\u@\h \V \w] \$ '

# 换行符、用户名@短格式主机名、基本PTY、shell层级、历史编号、换行符，以
及完整的工作
目录名称（$PWD）：
export PS1='\n[\u@\h \l:$SHLVL:\!]\n$PWD\$ '

# 用户名@短格式主机名、上一个命令的退出状态，以及当前工作目录：
export PS1='[\u@\h $? \w \$ '

# 换行符、用户名@短格式主机名，以及后台作业数：
export PS1='\n[\u@\h jobs:\j]\n$PWD\$ '

# 换行符、用户名@短格式主机名、终端、shell层级、历史编号、作业数量、
bash版本号，以
及完整的工作目录：
export PS1='\n[\u@\h t:\l l:$SHLVL h:\! j:\j v:\V]\n$PWD\$ '

# 换行符、用户名@短格式主机名、代表终端的T、代表shell层级的L、代表命令编
号的C，以及
ISO 8601格式的日期和时间：
export PS1='\n[\u@\h:T\l:L$SHLVL:C\!:\D{%Y-%m-
%d_%H:%M:%S_%Z}]\n$PWD\$ '

# 用户名@短格式主机名、浅蓝色的当前工作目录：
export PS1='\[\033[1;34m\][\u@\h:\w]\$\[\033[0m\] '

# 在xterm的标题栏和提示符中同时显示用户名@短格式主机名、当前工作目录：
export PS1='\[\033]0;\u@\h:\w\007\][\u@\h:\w]\$ '
```

```
# 同时更新颜色和xterm:
export PS1='\[\033]0;\u@\h:\w\007\]\[\033[1;34m\]
[\u@\h:\w]\$\[\033[0m\] '
```

16.2.3 讨论

注意，export 命令只需要使用一次，就可以标记出要导入子进程的变量。

假设已经设置 shell 选项 promptvars（默认情况），提示字符串会被解析和扩展（通过变量扩展、命令替换以及算术扩展）、引号会被移除并显示最终结果。提示字符串分别为
$PS0、$PS1、$PS2、$PS3、$PS4。

- $PS0 仅存在于 bash 4.4 及以上版本。下一节将详细讨论这个“执行前”（preexecution）提示符。
- $PS1 是命令提示符。
- $PS2 是辅助提示符，在 bash 需要更多信息来结束命令时，它会出现。该提示符默认为 >，但你也可以改用其他字符。
- $PS3 是 select 提示符（参见 3.7 节和 6.16 节），其默认为 #?。
- $PS4 是 xtrace（调试）提示符，其默认为 +。注意，$PS4 的首个字符会根据需要重复多次，以表明当前所执行命令的嵌套级数。

```
$ export PS2='Secondary> '

$ for i in *
Secondary> do
Secondary> echo $i
Secondary> done
cheesy_app
data_file
hard_to_kill
mcd
mode

$ export PS3='Pick me: '

$ select item in 'one two three'; do echo $item; done
1) one two three
Pick me: ^C

$ export PS4='+ debugging> '

$ set -x
```

```
$ echo $( echo $( for i in *; do echo $i; done ) )
+++ debugging> for i in '*'
+++ debugging> echo cheesy_app
+++ debugging> for i in '*'
+++ debugging> echo data_file
+++ debugging> for i in '*'
+++ debugging> echo hard_to_kill
+++ debugging> for i in '*'
+++ debugging> echo mcd
+++ debugging> for i in '*'
+++ debugging> echo mode
++ debugging> echo cheesy_app data_file hard_to_kill mcd mode
+ debugging> echo cheesy_app data_file hard_to_kill mcd mode
cheesy_app data_file hard_to_kill mcd mode
```

由于仅以交互方式运行 bash 时才有用，因此 $PS1 提示符的最佳设置位置是在全局文件 /etc/bashrc 或者本地文件 ~/.bashrc 中。

从风格角度而言，我们建议将空格作为 $PS1 字符串的最后一个字符。把提示字符串与用户输入的命令隔开，更易于阅读屏幕上的内容。有鉴于此，又因为提示字符串中可能包含其他空格或特殊字符，所以最好使用双引号或单引号将其引用起来，再赋给 $PS1。

至少有 3 种简单易行的方式可以在提示符中显示当前工作目录（CWD）：\w、\W、$PWD。\W 输出**基本名称**，或者说是目录路径的最后一部分，\w 则输出整个路径。注意，当你位于主目录时，两者输出的都是 ~，而非 $HOME 的值。这让有些用户着实抓狂，因此，要想输出完整的当前工作目录，可以使用 $PWD。输出完整的当前工作目录会改变提示符的长度，在比较深的目录层次结构中，这会使得提示符折行。这又让另一些用户抓狂了。对于 bash 4 或以上版本，使用 $PROMPT_DIRTRIM 并配合 \w 或 \W 即可（不会影响 $PWD）。Bash Reference Manual 中对该变量的描述如下。

> 如果设置为大于 0 的数字，那么在扩展 \w 和 \W 提示字符串转义时，将该值作为要保留的结尾目录组件的数量……[2] 被删除的那些字符以省略号替换。

[2]举例来说，如果 PROMPT_DIRTRIM=3，那么 [daniel@localhost /media/Projects/android/ui/out/target/product/generic/system/framework]$ 会显示为 [daniel@localhost .../generic/system/framework]$。——译者注

如果能使用 $PROMPT_DIRTRIM，那就用；如果无法使用，例 16-2 中给出了一个可用于截断工作目录的函数及其用法提示。

例 16-2 ch16/func_trunc_PWD

```
# 实例文件：func_trunc_PWD

function trunc_PWD {
    # $PWD截断代码改编自The Bash Prompt HOWTO:
    # 11.10. Controlling the Size and Appearance of $PWD

    # $PWD中有多少个字符应该保留
    local pwdmaxlen=30
    # 表明目录已被截断的指示符
    local trunc_symbol='...'
    # PWD的临时变量
    local myPWD=$PWD

    # 用'~'替换$PWD中匹配$HOME的部分
    # 如果想使用完整路径，可以将下面这条语句注释掉!
    myPWD=${PWD/$HOME/~}

    if [ ${#myPWD} -gt $pwdmaxlen ]; then
        local pwdoffset=$(( ${#myPWD} - $pwdmaxlen ))
        echo "${trunc_symbol}${myPWD:$pwdoffset:$pwdmaxlen}"
    else
        echo "$myPWD"
    fi
}
```

接下来演示一下用法。

```
$ source file/containing/trunc_PWD

[jp@freebsd ttyp0:3:60]
~/this is a bunch/of really/really/really/long directories/did I
mention really/
really/long$export PS1='\n[\u@\h \l:$SHLVL:\!]\n$(trunc_PWD)\$ '

[jp@freebsd ttyp0:3:61]
...d I mention really/really/long$
```

你可以注意到，这里的提示字符串出现在单引号内，因此，$ 和其他特殊字符会按照其字面意义被对待。提示字符串在显示期间求值，其中的变量会如期被扩展。双引号也不是不能用，但这样的话，你必须对 shell 元字符进行转义，例如，使用 \$ 代替 $。

命令编号和历史编号通常不是一回事：命令的历史编号是其在历史列表中的位置，列表可能包含从历史文件中恢复的命令；而命令编号是在当前 shell 会话期间所执行的一系列命令中的位置。

另外还有一个特殊变量 $PROMPT_COMMAND，该变量（如果设置的话）会被解释为在求值并显示 $PS1 之前要执行的命令。此处以及在 $PS1 中使用命令替换的问题是，每次显示提示符时都会执行这些命令，这太频繁了。例如，你可以在提示符中嵌入 $(ls-l | wc-l) 这样的命令替换，以此获得当前工作目录内的文件数量。但对于比较陈旧或者使用率颇高的系统来说，如果位于包含大量文件的目录中，可能会导致在看到提示符并能够着手工作前出现明显的延迟。提示符最好简短明了（尽管本节中展示的部分提示符挺吓人的）。如有需要，可以定义函数或别名，别搞得你的提示符又乱又慢。

为了解决提示符中出现的 ANSI 或 xterm 转义序列在不被支持的情况下产生乱七八糟的内容，你可以在 rc 文件中加入下列操作。

```
case $TERM in
     xterm*) export \
PS1='\[\033]0;\u@\h:\w\007\]\[\033[1;34m\][\u@\h:\w]\$\[\033[0m\]' ;;
              *) export PS1='[\u@\h:\w]\$ ' ;;
esac
```

有关该话题的更多信息，参见 A.2 节。

颜色

在我们刚刚讨论的 ANSI 例子中，1;34m 代表“将字符属性设置为浅色，将字符颜色设置为蓝色”。0m 代表“清除所有属性，不设置颜色”。这些代码可参见 A.3 节。结尾的 m 表示颜色转义序列。

例 16-3 中的脚本展示了所有可能的组合。如果你的终端显示不出颜色，则表明未启用或不支持 ANSI 颜色。

例 16-3 ch16/colors

```
#!/usr/bin/env bash
# 实例文件：colors
#
# Daniel Crisman的ANSI颜色图表脚本取自
# The Bash Prompt HOWTO: 6.1. Colours
#
```

```
# 该文件向终端回显了一批颜色代码，以演示可用的颜色。
# 每行分别选择17种前景色颜色代码（默认 + 16个转义序列）中的一种，
# 然后在所有9种背景色上测试该颜色（默认 + 8个转义序列）
#

T='gYw' # 测试文本

echo -e "\n                 40m     41m     42m     43m\
     44m     45m     46m     47m";

for FGs in '    m'   ' 1m' '  30m' '1;30m' '  31m' '1;31m' '  32m' \
           '1;32m' '  33m' '1;33m' '  34m' '1;34m' '  35m' '1;35m' \
           '  36m' '1;36m' '  37m' '1;37m'; do
    FG=${FGs// /}
    echo -en " $FGs \033[$FG  $T  "
    for BG in 40m 41m 42m 43m 44m 45m 46m 47m; do
        echo -en "$EINS \033[$FG\033[$BG  $T  \033[0m";
    done
    echo;
done
echo
```

如果想用简单的方法体验一些丰富多彩的终端主题，可以试试 Bashish。

16.2.4 参考

- Bash Reference Manual
- bash tarball 中的 ./examples/scripts.noah/prompt.bash
- Bash Prompt HOWTO（tldp.org）
- 1.3 节
- 3.7 节
- 6.16 节
- 6.17 节
- 16.3 节
- 16.12 节
- 16.20 节
- 16.21 节
- 16.22 节
- A.2 节
- A.3 节

16.3 在程序运行前出现的提示符

16.3.1 问题

你想让提示符在程序运行前出现，而不只是在结束之后。这样可以方便地标记起止时间。

16.3.2 解决方案

这个解决方案仅适用于 bash 4.4 或以上版本。这些版本中引入了 $PS0 提示符。如果设置的话，该提示字符串会在命令执行前被求值并输出。

接下来我们用 $PS0 和 $PS1 来显示命令运行的起止时间。

```
PS0='                                                \t\n'
PS1='------------------------------------------------ \t\n\! \$ '
```

16.3.3 讨论

$PS0 是执行前提示符。它会在 shell 执行命令前显示。那些前导空白字符是为了将输出往右边挪动；如果觉得还不够靠右或想再往左边些，随意添加或删除空白字符即可。与此类似，$PS1 中的连字符也是为了将时间戳右移，同时在命令之间划定界限，以便更好地观察。你也可以根据需要添加更多的连字符（或改用空格）。

这两个提示符中的关键在于 \t。它会转换成时间戳。\n 代表换行符，作用是更好地格式化。如果按照上述方法设置好提示符，然后执行一个得花费些时间的命令（如 sleep 5），可以看到如下结果。

```
1037 $ echo 'sleep...' ; sleep 5; echo 'awake!'
                                                 21:36:59
sleep...
awake!
------------------------------------------------ 21:37:04
1038 $
```

如果想让 $PS0 提示符和键入的命令出现在同一行，那就在 \t 前面加上 \e[A。为了让时间戳更往右移，你可能还得添加更多空格。

要想在设置之前先测试提示符，可以利用另一个仅适用于 bash 4.4 或以上版本的特性。将提示字符串赋给某个变量，然后使用 @P 运算

符输出该变量的值。例如：

```
$ MYTRY='    \! \h \t\n'
$ echo "${MYTRY}"
    \! \h \t\n
$ echo "${MYTRY@P}"
    1015 monarch 14:07:45

$
```

如果不使用 @P，输出的则是所键入的字符；加上 @P 的话，则会按照提示字符串解释其中的特殊字符序列。如果显示效果如你所愿，就将其赋给 $PS0。

时间戳的解析度只能到秒，并不适用于精确的性能测量，但在回顾一系列命令的来龙去脉（尤其离开屏幕去拿咖啡的情况下）以及哪些命令运行时间较长时非常有用。

16.3.4 参考

- 16.2 节

16.4 永久修改$PATH

16.4.1 问题

你需要永久性修改命令路径。

16.4.2 解决方案

首先，你得找出在哪里设置路径，然后再修改。对于本地账户，可能是在 ~/.profile 或 ~/.bash_profile 中。用 grep -l PATH ~/.[^.]* 找出该文件并使用你喜欢的编辑器进行修改。接着对文件使用 source 命令，使改动立即生效。

如果你是 root 用户，则需要设置整个系统的路径，基本操作过程是一样的，但 /etc 目录中可供设置 $PATH 的文件不尽相同，具体取决于你的操作系统和版本。最有可能的文件是 /etc/profile，但也有可能是

/etc/bashrc、/etc/rc、/etc/default/login、~/.ssh/environment 以及 PAM 的 /etc/environment。

某些系统中有一个名为 /etc/profile.d 的目录，其中包含了系统启动时会运行的 shell 脚本。你可以修改该目录中已有的脚本或添加新脚本。里面的各色脚本只是组织或模块化多种设置的一种方式，这好过将它们一股脑地放进一个大文件中。

16.4.3 讨论

由于 shell 通配符扩展的性质以及 . 和 .. 目录的存在，grep -l PATH ~/.[^.]* 命令值得注意。更多细节参见 1.7 节。

$PATH 中列出的位置会带来安全隐患，尤其作为 root 用户时。如果 root 用户的路径中有一个人皆可写的目录出现在其他典型目录（/bin、/sbin）之前，那么本地用户就可以创建出让 root 执行的文件，从而对系统为所欲为。这也是当前目录（.）不应该出现在 root 用户路径中的原因。

为了避免这个问题：

- 令 root 用户的路径尽可能简短，绝不要使用相对路径
- 避免在 root 用户的路径中出现人皆可写的目录
- 考虑在 root 用户执行的 shell 脚本中明确设置路径
- 对于 root 用户执行的 shell 脚本中用到的实用工具，考虑硬编码其绝对路径
- 将用户或应用程序目录放在 $PATH 的末尾，只用于非特权用户

16.4.4 参考

- 1.7 节
- 4.1 节
- 14.3 节
- 14.9 节
- 14.10 节
- 16.5 节

16.5 临时修改$PATH

16.5.1 问题

你想在当前会话期间向 $PATH 中添加（或删除）一个目录。

16.5.2 解决方案

这个问题的解决方案不止一种。

可以使用 PATH="*newdir*:$PATH" 或 PATH="$PATH:*newdir*" 将新目录放在前面或后面，不过你得确保该目录不会在 $PATH 中重复出现。

如果需要编辑路径中间的部分，可以通过 echo 命令将路径输出到屏幕，然后使用终端的 kill 和 yank（剪切和粘贴）功能 [3] 将其复制到新行并进行编辑。你也可以从 readline 文档中添加“便于 shell 交互的宏”（[m]acros that are convenient for shell interaction）。尤其是：

[3]在命令行的 emacs 编辑模式下，kill 命令相当于“剪切”，yank 命令相当于“粘贴”。——译者注

```
# 编辑路径
"\C-xp": "PATH=${PATH}\e\C-e\C-a\ef\C-f"
# [...]
# 在当前行编辑变量
"\M-\C-v": "\C-a\C-k$\C-y\M-\C-e\C-a\C-y="
```

按下 Ctrl-X P，当前行上会显示出 $PATH 的内容，输入任何变量名称，然后按下 Meta-Ctrl-V，就会显示出该变量的内容，以供编辑。这非常方便。

对于一些简单的情况，你也可以使用例 16-4 中的函数（取自 Red Hat Linux 的 /etc/profile，略有改编）。

例 16-4 ch16/func_pathmunge

```
# 实例文件：func_pathmunge

# 改编自Red Hat Linux版本

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

```
            PATH="$PATH:$1"
        else
            PATH="$1:$PATH"
        fi
    fi
}
```

egrep 模式 (^|:)$1($|:) 用于在 $PATH 字符串中的各个可能位置查找 $1 的值。我们在自己编写的函数 func_tweak_path 中选择使用 case 语句，强制在路径的开头和结尾加上了 :[4]。因为用的是 shell 内建命令，所以这种做法在理论上速度会更快，不过 Red Hat 版本要更简洁。我们的版本也很好地演示了 if 命令能够处理**退出码**，因此第一个 if 可以使用 grep 所设置的退出码，第二个 if 则需要使用测试运算符（[]）。

[4]在 $target 两侧加上冒号，是为了保证 $target 必须是出现在冒号之间的完整路径；如果不加冒号，则有可能作为其他路径的一部分被匹配。为了处理 $target 可能会出现在 $PATH 的开头或结尾的情况，所以在 $PATH 两侧也加上了冒号（local pattern=":$1:"）。——译者注

对于需要进行大量错误检查的复杂情况，可以读入并使用例 16-5 中给出的更为通用的函数。

例 16-5 ch16/func_tweak_path

```
# 实例文件：func_tweak_path

#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++
# 在路径开头或结尾处添加目录（只要该目录尚不存在）
# 不考虑符号链接!
# 返回值：1或者设置好新的$PATH
# 调用方式：add_to_path <directory> (pre|post)
function add_to_path {
    local location=$1
    local directory=$2

    # 确保有可处理的对象
    if [ -z "$location" -o -z "$directory" ]; then
        echo "$0:$FUNCNAME: requires a location and a directory to
add" >&2
        echo "e.g. add_to_path pre /bin" >&2
        return 1
    fi

    # 确保不是相对目录
```

```
    if [ $(echo $directory | grep '^/') ]; then
        :echo "$0:$FUNCNAME: '$directory' is absolute" >&2
    else
        echo "$0:$FUNCNAME: can't add relative directory '$directory'
to \$PATH" >&2
        return 1
    fi

    # 确保要添加的目录确实存在
    if [ -d "$directory" ]; then
        : echo "$0:$FUNCNAME: directory exists" >&2
    else
        echo "$0:$FUNCNAME: '$directory' does not exist--aborting" >&2
        return 1
    fi

    # 确保要添加的目录尚不存在于$PATH之中
    if [ $(contains "$PATH" "$directory") ]; then
        echo "$0:$FUNCNAME: '$directory' already in \$PATH--aborting"
>&2
    else
        :echo "$0:$FUNCNAME: adding directory to \$PATH" >&2
    fi

    # 判断要执行的操作
    case $location in
        pre*  ) PATH="$directory:$PATH" ;;
        post* ) PATH="$PATH:$directory" ;;
        *     ) PATH="$PATH:$directory" ;;
    esac

    # 清理新路径并设置$PATH
    PATH=$(clean_path $PATH)

} # 函数add_to_path定义完毕

#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++
# 从路径中删除目录（如果存在的话）
# 返回值：设置好新的$PATH
# 调用方式：rm_from_path <directory>
function rm_from_path {
    local directory=$1

    # 从$PATH中删除所有的$directory实例
    PATH=${PATH//$directory/}

    # 清理新路径并设置$PATH
    PATH=$(clean_path $PATH)

} # 函数rm_from_path定义完毕
```

```
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++
# 删除开头/结尾或重复的':'，删除重复的目录项
# 返回值：显示"经过清理的"路径
# 调用方式：cleaned_path=$(clean_path $PATH)
function clean_path {
    local path=$1
    local newpath
    local directory

    # 确保有可处理的对象
    [ -z "$path" ] && return 1

    # 删除重复的目录项（如果存在的话）
    for directory in ${path//:/ }; do
        contains "$newpath" "$directory" &&
newpath="${newpath}:${directory}"
    done

    # 删除开头的':'分隔符
    # 删除结尾的':'分隔符
    # 删除重复的':'分隔符
    newpath=$(echo $newpath | sed 's/^:*//; s/:*$//; s/::/:/g')

    # 返回新路径
    echo $newpath

} # 函数clean_path定义完毕


#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++
# 判断路径是否包含指定的目录
# 如果目标符合模式，返回1；否则，返回0
# 调用方式：contains $PATH $dir
function contains {
    local pattern=":$1:"
    local target=$2

    # 比较时区分大小写，除非设置过nocasematch选项
    case $pattern in
        *:$target:* ) return 1;;
        *           ) return 0;;
    esac
} # 函数contains定义完毕
```

用法如下所示：

```
$ source chpath

$ echo $PATH
/bin:/usr/bin:/usr/local/bin:/usr/bin/X11:/usr/X11R6/bin:/home/jp/bin

$ add_to_path pre foo
-bash:add_to_path: can't add relative directory 'foo' to the $PATH

$ add_to_path post ~/foo
-bash:add_to_path: '/home/jp/foo' does not exist--aborting

$ add_to_path post '~/foo'
-bash:add_to_path: can't add relative directory '~/foo' to the $PATH

$ rm_from_path /home/jp/bin

$ echo $PATH
/bin:/usr/bin:/usr/local/bin:/usr/bin/X11:/usr/X11R6/bin

$ add_to_path /home/jp/bin
-bash:add_to_path: requires a location and a directory to add
e.g. add_to_path pre /bin

$ add_to_path post /home/jp/bin

$ echo $PATH
/bin:/usr/bin:/usr/local/bin:/usr/bin/X11:/usr/X11R6/bin:/home/jp/bin

$ rm_from_path /home/jp/bin

$ add_to_path pre /home/jp/bin

$ echo $PATH
/home/jp/bin:/bin:/usr/bin:/usr/local/bin:/usr/bin/X11:/usr/X11R6/bin
```

16.5.3 讨论

关于这个问题以及例 16-5 中给出的函数，有 4 个值得注意的地方。

首先，如果尝试在 shell 脚本中修改 $PATH 或其他环境变量，起不到任何效果，因为脚本在子 shell 中运行，脚本结束后，子 shell 也随之终止，修改过的环境变量全部会失效。因此，我们将函数读入当前 shell 环境并在其中运行。

其次，你可能注意到 add_to_path post ~/foo 会返回“does not exist”，而 add_to_path post'~/foo' 返回的是“can't add relative directory”。原因是，在函数运行之前，shell 会将 ~/foo

扩展成 /home/jp/foo。没考虑到 shell 扩展是一个常见错误。可以用 echo 命令查看 shell 到底将什么传给了脚本和函数。

接下来，你可能还会注意到 echo "$0:$FUNCNAME:requires a location and a directory to add" >&2 这样的代码行。$0:$FUNCNAME 可以方便地识别出错误消息究竟来自何处。$0 始终保存的是当前程序的名称（本例中为 -bash，其他情况下是脚本或程序的名称）。加上函数名便于在调试时定位错误。>&2 将输出发送到 STDERR，运行期间产生的用户反馈（尤其是警告或错误信息）都应该出现在这里。

最后，你可以认为这些函数的接口不一致，因为 add_to_path 和 remove_from_path 实际上设置了 $PATH，而 clean_path 显示的是经过清理的路径，contains 返回真或假。我们在生产环境中并不会这么做，这里只是为了让示例更加有趣，展现不同的处理方式。鉴于这些函数的用途，我们觉得这些接口也说得过去。

16.5.4 参考

- 10.5 节
- 14.3 节
- 14.9 节
- 14.10 节
- 16.4 节
- 16.22 节
- 附录 B

16.6 设置$CDPATH

16.6.1 问题

你希望能够更轻松地在不同位置的多个目录之间进行切换。

16.6.2 解决方案

恰当地设置 $CDPATH。你的常用目录可能就是那么几个，假设你要花费大量时间来处理 init 的 rc 目录：

```
/home/jp$ cd rc3.d
bash: cd: rc3.d: No such file or directory

/home/jp$ export CDPATH='.:/etc'

/home/jp$ cd rc3.d
/etc/rc3.d

/etc/rc3.d$ cd rc5.d
/etc/rc5.d

/etc/rc5.d$ cd games
bash: cd: games: No such file or directory

/etc/rc5.d$ export CDPATH='.:/etc:/usr'

/etc/rc5.d$ cd games
/usr/games

/usr/games$
```

16.6.3 讨论

根据 Bash Reference Manual，$CDPATH 是“一个以冒号为分隔符的目录列表，用作内建命令 cd 的搜索路径”。可以将它看作 cd 的 $PATH。其用法有点微妙，但着实非常方便。

如果 cd 的参数以斜线开头，则用不到 $CDPATH。如果使用了 $CDPATH，则会向 STDOUT 输出新目录的绝对路径，如上面的示例所示。

> 如果是在 POSIX 模式下运行 bash（例如，/bin/sh 或使用 --posix 选项），那可要小心了。Bash Reference Manual 中提到过：
>
> > 如果使用了 $CDPATH 中的非空目录名，或者第一个参数是 -，而且成功切换了目录，新工作目录的绝对路径会被写入标准输出。
>
> 换句话说，几乎每次使用 cd 时都会向 STDOUT 回显新路径，这并非标准行为。

$CDPATH 中常见的目录包括：

.

当前目录（可选，因为这是隐含的）

~/

主目录

..

父目录

../..

祖父目录

~/.dirlinks

隐藏目录，只包含指向其他常用目录的符号链接

因此可以得到：

```
export CDPATH='.:~/:..:../..:~/.dirlinks'
```

16.6.4 参考

- help cd
- Bash Reference Manual
- 16.15 节
- 16.22 节
- 18.1 节

16.7 当找不到命令时

16.7.1 问题

你希望能够更好地处理找不到命令的情况，或者只是输出更好的错误消息。

16.7.2 解决方案

将下列函数添加到脚本的起始部分或者 rc 文件中，后者效果更佳：

```
function command_not_found_handle ()
{
    echo "Sorry. $0: $1 not there."
    return 1
}
```

16.7.3 讨论

在 bash 4.3 及其后续版本中，如果找不到你想要运行的可执行文件，shell 会调用一个特殊的函数 command_not_found_handle，你可以根据自己的用途（重新）定义该函数。本例只是让其输出 shell 的名称以及无法找到的命令。

重要的是，函数要能返回非 0 值来表明调用命令失败。脚本的其他部分或脚本的调用者有可能依赖此信息。

有些管理员将函数 command_not_found_handle 的定义放在了系统范围的 bashrc 文件（例如，/etc/profile 等）。在函数内部，查找 /usr/lib 或 /usr/share 下的 Python 脚本 command-not-found（注意是连字符，不是下划线）。该脚本会查找刚刚调用失败的命令所对应的软件包，看看能否建议安装，以提供缺失的命令。这种做法在某些情况下的确有帮助，但如果只是简单地输错了命令，那这就是添乱了。

16.7.4 参考

- 10.4 节
- 10.5 节
- 19.14 节

16.8 缩短或修改命令名称

16.8.1 问题

你想将经常用到的那些冗长或复杂的命令变得简短些，或者重新命名难以记忆、不易输入的命令。

16.8.2 解决方案

不要手动改名或移动可执行文件，因为 Unix 和 Linux 的不少地方都依赖于某些位置上的某些文件；你应该使用别名、函数以及符号链接。

根据 Bash Reference Manual，“别名允许将简单命令的第一个单词替换成字符串。shell 维护着一系列别名，可以使用内建命令 alias 和 unalias 进行设置和删除”。这意味着你可以在一个别名中列出多个命令，从而重命名命令或创建宏；例如，alias copy='cp' 或 alias ll.='ls -ld .*'。

别名只能扩展一次，因而得以在不陷入死循环的情况下改变命令的工作方式，比如 alias ls='ls -F'。大多数情况下，别名扩展仅检查命令行的第一个单词，而且别名属于严格的文本替换，本身**无法使用参数**。也就是说，你不能定义别名 alias mkcd='mkdir $1 && cd $1'。

函数有两种用法。首先，它们可以被读入当前的交互式 shell 环境，成为常驻内存的脚本。函数占用内存少，由于已经载入内存，运行速度非常快，而且是在当前进程中执行，而非其子进程。其次，函数可以在脚本中作为子程序使用。另外，函数还可以接受参数，如例 16-6 所示（取自 6.19 节）。

例 16-6 ch06/func_calc

```
# 实例文件：func_calc

# 简单的命令行计算器
function calc {
    # 仅适用于整数! --> echo The answer is: $(( $* ))
    # 浮点数
    awk "BEGIN {print \"The answer is: \" $* }";
} # 函数calc定义完毕
```

对于个人或系统范围的用途而言，使用别名或函数来重命名或调校命令可能更好，但符号链接允许单个命令同时出现在多个位置，这一点非常实用。例如，Linux 系统基本上都是使用 /bin/bash，而其他系统使用的可能是 /usr/bin/bash、/usr/local/bin/bash 或者 /usr/pkg/bin/bash。虽然有更好的办法处理这个问题（使用 env，参见 15.1 节），但符号链接可能是一种通用的解决方式。我们不推荐使用硬链接，因为不大容易看出来它们是否就是你要找的对象，而且硬链接更容易被操作不当的编辑器等破坏。符号链接更加显眼和直观。

16.8.3 讨论

在进行别名扩展时，通常只检查命令行的第一个单词。但如果别名值的最后一个字符是空白字符，则接下来的单词也会被检查。这在实践中很少出现什么问题。

由于 bash 中的别名无法使用参数（不同于 csh），如果的确需要传递参数，你得使用函数。别名和函数都位于内存中，因此差别不大。

除非设置了 shell 选项 expand_aliases，否则别名不会在非交互式 shell 中扩展。编写脚本的最佳实践要求不使用别名，因为其他系统中未必存在同样的别名。你还需要在脚本中定义函数，或在使用前明确地将其读入（参见 19.14 节）。因此，定义函数的最佳位置是在全局的 /etc/bashrc 或本地的~/.bashrc 中。

16.8.4 参考

- 6.19 节
- 10.4 节
- 10.5 节
- 10.7 节
- 14.4 节
- 15.1 节
- 16.20 节
- 16.21 节
- 16.22 节
- 19.14 节

16.9 调整shell行为及环境

16.9.1 问题

你希望根据自己的工作方式、物理位置、语言等调整 shell 环境。

16.9.2 解决方案

参见 A.6 节、A.7 节、A.8 节。

16.9.3 讨论

有 3 种方法可以调整 shell 环境的方方面面。set 是 POSIX 标准，使用单字母选项。shopt 针对 bash 的 shell 选项。由于历史原因，再加上要与很多第三方应用程序兼容，在用的环境变量数量着实不少。如何调整以及在哪里调整让人倍感困惑。附录 A 中的表格可以帮你理清头绪，由于篇幅过多，这里就不再赘述。

16.9.4 参考

- help set
- help shopt
- Bash Reference Manual
- A.6 节
- A.7 节
- A.8 节

16.10 用.inputrc调整readline的行为

16.10.1 问题

你想要调整 bash 处理输入的方式，尤其是命令补全。例如，你希望能够忽略大小写。

16.10.2 解决方案

根据情况，编辑或创建适合的 ~/.inputrc 或 /etc/inputrc 文件。有大量的选项可供你根据喜好调整。要想 readline 在初始化时使用你所提供的文件，可以设置 $INPUTRC，例如，INPUTRC=~/.inputrc。要想重新读取文件并应用设置，或修改后进行测试，可以使用 bind -f *filename*。

我们推荐你研究一下 readline 的文件以及 bind 命令，尤其是 bind -v、bind -l、bind -s、bind -p，最后一个命令涉及的内容颇多，而且晦涩难懂。

有关 readline 的更多配置信息，参见 A.18 节。对于来自其他环境（尤其是 Windows）的用户比较有用的配置是：

```
# 这是inputrc设置中值得注意的一部分，更长的示例参见16.22节
# 要想重新读取（并应用对该文件做出的改动），可以使用：
# bind -f $SETTINGS/inputrc

# 首先，纳入/etc/inputrc中所有的系统范围绑定和变量赋值
#（如果文件不存在，则静默失败）
$include /etc/inputrc

$if Bash
  # 进行命令补全时忽略大小写
    set completion-ignore-case on
  # 在补全的目录名后添加斜线
    set mark-directories on
  # 如果补全的名称是指向目录的符号链接，在其后添加斜线
    set mark-symlinked-directories on
  # 在补全的名称后添加与ls -F相同的文件类型符号
    set visible-stats on
  # 遍历模棱两可的补全，而不是将其列出
    "\C-i": menu-complete
  # 将铃声设置为能够听到
    set bell-style audible
  # 列出可能的补全结果，而不是响铃
    set show-all-if-ambiguous on

  # 取自readline文档
  # 方便shell交互使用的一些宏
  # 编辑路径
    "\C-xp": "PATH=${PATH}\e\C-e\C-a\ef\C-f"
  # 为输入带引号的单词做准备——插入左双引号和右双引号，
  # 然后将光标移动到左双引号之后
    "\C-x\"": "\"\"\C-b"
  # 插入反斜线（在字符序列和宏中测试反斜线转义）
    "\C-x\\": "\\"
  # 引用当前或前一个单词
    "\C-xq": "\eb\"\ef\""
  # 添加绑定以刷新未绑定的行
    "\C-xr": redraw-current-line
  # 编辑当前行上的变量
    #"\M-\C-v": "\C-a\C-k$\C-y\M-\C-e\C-a\C-y="
    "\C-xe": "\C-a\C-k$\C-y\M-\C-e\C-a\C-y="
$endif
```

不妨尝试一下这些及其他配置。另外注意，$include 引入了系统配置，但要确保你能够根据需要进行修改。16.22 节中给出了可供下载的文件。

16.10.3 讨论

很多人并没有意识到 GNU readline 库具有多么出色的可定制性，更别提其强大的功能和灵活性了。虽说如此，但并不存在“一劳永逸”的方法。你应该制定适合自己需求和习惯的配置。

注意，第一次调用 readline 时会进行其正常的启动文件处理，包括查看 $INPUTRC，如果该变量未设置，则默认为 ~/.inputrc。

16.10.4 参考

- help bind
- readline 文档
- 16.21 节
- 16.22 节
- A.17 节

16.11 通过添加~/bin来存放个人工具

16.11.1 问题

你有一些要使用的个人工具，但你并非系统的 root 用户，无法将这些工具放入正常的位置（如 /bin 或 /usr/local/bin），或者出于其他原因，需要将这些工具隔离开。

16.11.2 解决方案

创建~/bin 目录，将个人工具放入其中并将该目录添加到路径：

```
PATH="$PATH:~/bin"
```

你需要在某个 shell 初始化文件（如 ~/.bashrc）中执行上述更改。有些系统已经默认将 $HOME/bin 作为非特权用户账户的最后一个目录，因此最好先检查一下。

16.11.3 讨论

作为一名绝对合格的 shell 用户（毕竟你已经买了这本书嘛），你肯定创建了不少脚本。使用完整路径名调用脚本实在不方便。通过将脚本汇集到 ~/bin 目录中，至少可以让你的脚本看起来就像普通的 Unix 程序那样。

出于安全考虑，不要将你的 bin 目录放在路径开头。这样很容易覆盖系统命令，如果是无心之举（我们也干过），这会很不方便；如果是恶意为之，那可就危险了。

16.11.4 参考

- 14.9 节
- 14.10 节
- 16.4 节
- 16.5 节
- 16.8 节
- 19.4 节

16.12 使用辅助提示符：$PS2、$PS3、$PS4

16.12.1 问题

你想知道提示符 $PS2、$PS3、$PS4 的用途。

16.12.2 解决方案

$PS2 被称为**辅助提示符**，交互式输入某个尚未结束的命令时，就会出现该提示符。其形式通常为 >，不过你也可以重新定义。例如：

```
$ export PS2='Secondary: '

$ for i in $(ls)
Secondary: do
Secondary: echo $i
Secondary: done
colors
deepdir
trunc_PWD
```

$PS3 是 select 提示符，select 语句用它来提示用户输入值。其默认形式为 #?，看起来并不怎么直观。你应该在使用 select 语句前先做修改。例如：

```
$ select i in $(ls)
Secondary: do
Secondary: echo $i
Secondary: done
1) colors
2) deepdir
3) trunc_PWD
#? 1
colors
#? ^C

$ export PS3='Choose a directory to echo: '

$ select i in $(ls); do echo $i; done
1) colors
2) deepdir
3) trunc_PWD
Choose a directory to echo: 2
deepdir
Choose a directory to echo: ^C
```

$PS4 会在跟踪输出期间显示。其首个字符会根据需要出现多次，以指明嵌套深度。该提示符默认形式为 +。例如：

```
$ cat demo
#!/usr/bin/env bash

set -o xtrace

alice=girl
echo "$alice"

ls -l $(type -path vi)

echo line 10
ech0 line 11
echo line 12

$ ./demo
+ alice=girl
+ echo girl
girl
++ type -path vi
+ ls -l /usr/bin/vi
```

```
-r-xr-xr-x 6 root wheel 285108 May 8 2005 /usr/bin/vi
+ echo line 10
line 10
+ ech0 line 11
./demo: line 11: ech0: command not found
+ echo line 12
line 12

$ export PS4='+xtrace $LINENO: '

$ ./demo
+xtrace 5: alice=girl
+xtrace 6: echo girl
girl
++xtrace 8: type -path vi
+xtrace 8: ls -l /usr/bin/vi
-r-xr-xr-x 6 root wheel 285108 May 8 2005 /usr/bin/vi
+xtrace 10: echo line 10
line 10
+xtrace 11: ech0 line 11
./demo: line 11: ech0: command not found
+xtrace 12: echo line 12
line 12
```

16.12.3 讨论

提示符 $PS4 用到了 $LINENO 变量，该变量会返回函数所在的当前行号。还要注意单引号，它会将变量扩展推迟到显示时。

16.12.4 参考

- 1.3 节
- 3.7 节
- 6.16 节
- 6.17 节
- 16.2 节
- 19.13 节

16.13 在会话间同步shell历史记录

16.13.1 问题

你同时运行了多个 bash 会话，希望能够在这些会话间共享历史记录。另外，你还想阻止最后关闭的会话破坏其他会话的历史记录。

16.13.2 解决方案

使用 history 命令手动或自动在多个会话间同步历史记录。

16.13.3 讨论

使用默认设置的话，最后一个顺利退出的 shell 会覆盖历史文件，除非将其与同时打开的其他 shell 同步，否则这些 shell 的历史记录就会被破坏。16.14 节中介绍的 shell 选项的确管用，它能够追加历史记录文件，避免出现覆盖，在多个会话间保持历史记录同步或许可以获得一些别的好处。

手动同步历史记录涉及编写别名，将当前历史记录追加到历史文件（history -a），然后重新将该文件中的新记录读入当前 shell 的历史记录（history -n）：

```
alias hs='history -a ; history -n'
```

这种方法的缺点是，如果你想同步历史记录，则必须在各个 shell 中手动执行命令。为了实现自动化，不妨利用 $PROMPT_COMMAND 变量：

```
PROMPT_COMMAND='history -a ; history -n'
```

$PROMPT_COMMAND 的值被视为要在每次显示交互式提示符 $PS1 之前执行的命令。这种方法的缺点是，每次显示 $PS1 时都会执行指定的命令。执行次数非常频繁，在负载重或者速度慢的系统中，这会显著拖慢 shell，尤其历史记录数量众多时。

16.13.4 参考

- help history
- 16.4 节

16.14 设置shell的历史选项

16.14.1 问题

你希望能够更多地控制命令行历史记录。

16.14.2 解决方案

根据需要，设置 $HIST* 变量以及相关的 shell 选项。

16.14.3 讨论

$HISTFILESIZE 变量设置了 $HISTFILE 中允许出现的行数。$HISTFILESIZE 的默认值为 500，$HISTFILE 默认为 ~/.bash_history，如果处于 POSIX 模式，则为 ~/.sh_history。增加 $HISTFILESIZE 的值也许管用，如果将其删除，$HISTFILE 的长度将不受限制。修改 $HISTFILE 并非必要，除非该变量尚未设置或者对应的文件不可写，无法将历史记录写入磁盘。$HISTSIZE 变量设置了内存中历史记录栈（history stack）所允许的行数。

$HISTIGNORE 和 $HISTCONTROL 控制着哪些内容会被保存到历史记录。$HISTIGNORE 要更灵活，因为它允许你指定模式，以决定需要保存的命令行。$HISTCONTROL 的局限性更大，仅支持下列几个关键词（其他值均会被忽略）。

ignorespace

以空格起始的命令行不会保存进历史记录。

ignoredups

与先前的历史记录项重复的命令行不会保存进历史记录。

ignoreboth

ignorespace 和 ignoredups 的简写。

erasedups

在保存当前命令行前，先删除之前所有与其匹配的历史记录项。

如果没有设置 $HISTCONTROL 或者其中不包含以上关键字，那么所有符合 $HISTIGNORE 指定模式的命令行都会保存进历史记录。多行复合命令的第二行以及后续行不参与测试，会被直接添加到历史记录中，无论 $HISTCONTROL 包含什么值。

（上述段落内容改编自 Bash Reference Manual。）

$HISTTIMEFORMAT 可用于 bash 3 及后续版本，如果该变量已设置且不为空，则指定了显示或写入历史记录时所使用的 strftime 格式字符串。如果你的 bash 版本不符，但所用的终端带有回滚缓冲区，那么在提示符中加入日期和时间戳也能帮上大忙（参见 16.2 节）。需要注意的是，bash 并不会自动在 $HISTTIMEFORMAT 之后添加空格，有些系统（如 Debian）对此作了修补：

```
$ history
    1  ls -la
    2  help history
    3  help fc
    4  history

# 难看
$ export HISTTIMEFORMAT='%Y-%m-%d_%H:%M:%S'

$ history
    1  2006-10-25_20:48:04ls -la
    2  2006-10-25_20:48:11help history
    3  2006-10-25_20:48:14help fc
    4  2006-10-25_20:48:18history
    5  2006-10-25_20:48:39export HISTTIMEFORMAT='%Y-%m-%d_%H:%M:%S'
    6  2006-10-25_20:48:41history

# 美观
$ HISTTIMEFORMAT='%Y-%m-%d_%H:%M:%S; '

$ history
    1  2006-10-25_20:48:04; ls -la
    2  2006-10-25_20:48:11; help history
    3  2006-10-25_20:48:14; help fc
    4  2006-10-25_20:48:18; history
    5  2006-10-25_20:48:39; export HISTTIMEFORMAT='%Y-%m-%d_%H:%M:%S'
    6  2006-10-25_20:48:41; history
    7  2006-10-25_20:48:47; HISTTIMEFORMAT='%Y-%m-%d_%H:%M:%S; '
    8  2006-10-25_20:48:48; history

# 需要点技巧了
$ HISTTIMEFORMAT=': %Y-%m-%d_%H:%M:%S; '

$ history
```

```
    1  : 2006-10-25_20:48:04; ls -la
    2  : 2006-10-25_20:48:11; help history
    3  : 2006-10-25_20:48:14; help fc
    4  : 2006-10-25_20:48:18; history
    5  : 2006-10-25_20:48:39; export HISTTIMEFORMAT='%Y-%m-
%d_%H:%M:%S'
    6  : 2006-10-25_20:48:41; history
    7  : 2006-10-25_20:48:47; HISTTIMEFORMAT='%Y-%m-%d_%H:%M:%S; '
    8  : 2006-10-25_20:48:48; history
```

最后一个例子用内建命令 : 和元字符 ; 将日期戳包围起来，形成了一个“什么都不干”的命令（如 : 2006-10-25_20:48:48;）。这允许你重用历史记录文件中的文字行，不必费心解析日期戳。注意，别漏了 : 之后的空格。

还有一些 shell 选项也可以配置历史记录文件处理。如果设置 histappend 选项，shell 会将历史记录追加到历史文件；否则，历史文件会被覆盖。注意，它仍然受限于 $HISTFILESIZE。如果设置 cmdhist，多行命令会保存为单行形式，同时根据需要添加分号。如果设置 lithist，则多行命令与嵌入的换行符一起被保存。

16.14.4 参考

- help history
- help fc
- 6.11 节
- 16.2 节
- 16.9 节

16.15 创建更好的cd命令

16.15.1 问题

你进入了一个层级很深的目录，希望能够通过输入 cd ... 向上跳转 4 层，而不是费力地输入 cd ../../../..。

16.15.2 解决方案

使用例 16-7 中给出的函数。

例 16-7　ch16/func_cd

```
# 实例文件：func_cd

# 允许使用cd ...向上跳转2层，使用cd ....向上跳转3层，以此类推。（就像4NT/4DOS）
# 用法：cd ...，以此类推
function cd {

    local option= length= count= cdpath= i= # 局部变量，设置好初始状态

    # 如果存在-L或-P选项，将其保存下来，然后删除
    if [ "$1" = "-P" -o "$1" = "-L" ]; then
        option="$1"
        shift
    fi

    # 我们是否使用了这种特殊的语法？确保$1不为空，
    # 然后匹配$1的前3个字符，看看是否为'...'，
    # 接着尝试替换，以确保不存在斜线；如果失败，则说明没有斜线
    if [ -n "$1" -a "${1:0:3}" = '...' -a "$1" = "${1%/*}" ]; then
        # 我们使用了特殊语法
        length=${#1}  # 假定$1只包含点号，统计点号数量
        count=2       # 'cd ..'仍代表向上跳转1层，因此忽略前2个点号

        # 只要点号尚未用完，就继续向上跳转1层
        for ((i=$count;i<=$length;i++)); do
            cdpath="${cdpath}../" # 构建cd命令用到的路径
        done

        # 执行cd命令
        builtin cd $option "$cdpath"
    elif [ -n "$1" ]; then
        # 未使用特殊语法，只是普通的cd命令
        builtin cd $option "$*"
    else
        # 未使用特殊语法，只是单独的普通cd命令，可以切换到主目录
        builtin cd $option
    fi
} # 函数cd定义完毕
```

16.15.3　讨论

cd 命令接受可选的 -L 或 -P 选项，分别表示使用逻辑路径或文件系统的物理路径。无论是哪个选项，要想重新定义 cd 命令的工作方式，必须将其考虑在内。

然后，确保 $1 不为空并匹配其前 3 个字符，测试是否为 ...。接着尝试替换，确保不存在斜线；如果失败，则说明没有斜线。这两处字符串操作均要求 bash 的版本为 2.0+。在此之后，使用可移植的 for 循环构建实际的 cd 命令，最后通过 bulitin 使用 shell 内建的 cd 命令，避免调用 cd 函数时陷入死循环。如果 -L 或 -P 选项存在，我们也会将其传入并处理。

16.15.4 参考

- help cd
- 4NT shell，本实例的灵感来源
- 15.5 节
- 16.6 节
- 16.16 节
- 16.17 节
- 16.22 节
- 18.1 节

16.16 一次性创建并切换到新目录

16.16.1 问题

你经常需要创建新目录并立即进入这些目录中执行某些操作，逐个输入命令的过程太枯燥了。

16.16.2 解决方案

在适合的配置文件（如 ~/.bashrc）中加入例 16-8 所示的函数，并将其读入 shell 环境。

例 16-8 ch16/func_mcd

```
# 实例文件：func_mcd

# 创建并切换到新目录
# 用法：mcd (<mode>) <dir>
function mcd {
    local newdir='_mcd_command_failed_'
```

```
    if [ -d "$1" ]; then          # 目录已存在，提醒用户……
        echo "$1 exists..."
        newdir="$1"
    else
        if [ -n "$2" ]; then      # 已经指定了目录的权限模式（mode）
            command mkdir -p -m $1 "$2" && newdir="$2"
        else                      # 普通的mkdir命令
            command mkdir -p "$1" && newdir="$1"
        fi
    fi
    builtin cd "$newdir"          # 直接进入
} # 函数mcd定义完毕
```

例如：

```
$ source mcd

$ pwd
/home/jp

$ mcd 0700 junk

$ pwd
/home/jp/junk

$ ls -ld .
drwx------ 2 jp users 512 Dec 6 01:03 .
```

16.16.3 讨论

该函数允许你在使用 mkdir 命令创建目录时指定可选的权限模式。如果该目录已经存在，则提醒用户，但仍切换进去。我们用 command 命令确保忽略名为 mkdir 的 shell 函数，用 builtin 命令确保只使用 shell 内建的 cd 命令。

我们还将字符串 _mcd_command_failed_ 赋给了一个局部变量，以防 mkdir 失败。如果一切顺利，该变量中保存的是正确的新目录名。如果出现问题，在试图执行 cd 命令时，它会显示出非常有帮助的消息，前提是没有大量的 _mcd_command_failed_ 目录存在。

```
$ mcd /etc/junk
mkdir: /etc/junk: Permission denied
-bash: cd: _mcd_command_failed_: No such file or directory
$
```

你可能认为，在 mkdir 失败时，我们可以用 break 或 exit 来轻松地改进代码。然而，break 仅适用于 for、while 或 until 循环，exit 实际上会退出 shell，因为被读入的函数就是在当前 shell 进程中运行的（可以使用 return，我们将此作为练习留给你来实践）。

```
command mkdir -p "$1" && newdir="$1" || exit 1 # 这会退出当前shell
command mkdir -p "$1" && newdir="$1" || break  # 这会出错
```

你也可以使用下列这种更简单的形式，但我们显然更喜欢现有解决方案中给出的更稳健的版本。

```
function mcd { mkdir "$1" && cd "$1"; }
```

16.16.4 参考

- man mkdir
- help cd
- help function
- 16.15 节
- 16.20 节
- 16.21 节
- 16.22 节

16.17 直达底部

16.17.1 问题

你要在很多又窄又深的目录结构中开展工作，所有的工作内容都位于目录的最底部，你厌倦了手动输入大量的 cd 命令。

16.17.2 解决方案

用以下别名直达目录底部。

```
alias bot='cd $(dirname $(find . | tail -n 1))'
```

16.17.3 讨论

在大型目录结构（如 /usr）中，find 命令的这种用法会花费大量时间，不推荐这么做。

取决于具体的目录结构，该方法可能并不管用，你得自己动手试试看。find . 会列出当前目录及其下层目录中的所有文件和子目录，tain -n 1 会获取最后一行输出，dirname 只提取路径部分，cd 负责切换到特定目录。你可能需要对上述命令进行调整，以便能够到达正确的位置。例如：

```
alias bot='cd $(dirname $(find . | sort -r | tail -n 5 | head -1))'
```

或者：

```
alias bot='cd $(dirname $(find . | sort -r | grep -v 'X11' | tail -n 3
\
| head -1))'
```

不断尝试调整最内部括号中的部分，尤其是 find 命令，直至获得你想要的结果。目录结构底部可能有一个密钥文件或子目录，这种情况下，下列函数也许能派上用场。

```
function bot { cd $(dirname $(find . | grep -e "$1" | head -1)); }
```

注意，别名无法使用参数，因此必须采用函数形式。我们没用 find 命令的 -name 选项，而是改用了 grep，因为后者更加灵活。根据目录结构，你可能要用 tail 代替 head。再次强调，先测试 find 命令。

16.17.4 参考

- man find
- man dirname
- man head
- man tail
- man grep
- man sort
- 16.15 节
- 16.16 节

16.18 用可装载的内建命令为bash添加新特性

本节中的内容取自 Cameron Newham 所著的 *Learning the bash Shell, 3rd Edition*（O'Reilly 出版）一书。

16.18.1 问题

你希望 bash 能够帮你完成某项任务，但是并没有适合的内建命令。考虑到效率问题，你想在 shell 中内建该功能，而不是依靠外部程序。或者你已经有用 C 语言编写好的源代码，不想或没法重写。

16.18.2 解决方案

使用 bash 2.0 版本中引入的动态可装载内建命令。bash 安装包的 ./examples/loadables/ 目录下（尤其是作为规范的 hello.c）包含了一些预先编写好的内建命令。可以将文件 Makefile 中与你所在系统相关的那些行的注释去掉，然后输入 make 来构建这些内建命令。我们选择其中的内建命令 tty 作为一般性的案例研究。

以下是 bash 4.4 版本中 ./examples/loadables/ 目录下的内建命令清单。

```
basename.c  id.c         necho.c       printenv.c   strftime.c
uname.c
cat.c       ln.c         pathchk.c     push.c       sync.c
unlink.c
dirname.c   loadables.h  perl          realpath.c   tee.c
whoami.c
finfo.c     logname.c    perl/bperl.c  rmdir.c      template.c  head.c
mkdir.c     perl/iperl.c setpgid.c     truefalse.c  hello.c
mypid.c
```

16.18.3 讨论

对于支持动态装载的系统，你可以用 C 语言编写自己的内建命令，将其编译成共享对象，随时在 shell 中通过 enable 内建命令来装载。

接下来我们将简要讨论如何在 bash 下编写及装载内建命令。该讨论假设你具备编写、编译、链接 C 语言程序的经验。

tty 模仿标准的 Unix 命令 tty，它会输出连接在标准输入上的终端名称。和其他命令一样，如果设备是 TTY，它会返回真；否则，返回假。另外，tty 还能接受 -s 选项，以指定是否采用静默模式工作（也就是什么都不输出，只返回结果）。

内建命令的 C 语言代码分为 3 个截然不同的部分：实现该内建命令功能的代码、定义帮助信息、该内建命令的描述结构，以便 bash 得以访问。

描述结构非常直观，其形式如下所示。

```
struct builtin builtin_name_struct = {
    "builtin_name",
    function_name,
    BUILTIN_ENABLED,
    help_array,
    "usage",
    0
};
```

第一行必须包含尾随的 *_struct*，以便内建命令 enable 能够找到符号名。*builtin_name* 是出现在 bash 中的内建命令名称。下一个字段 *function_name* 是实现该内建命令的 C 语言函数名称。我们一会再来探讨这个函数。BUILTIN_ENABLED 是该内建命令的初始状态，表明是否启用。这个字段应该始终设置为 BUILTIN_ENABLED。*help_array* 是一个字符串数组，对内建命令使用 help 时会被输出。*usage* 是形式更简短的帮助信息，仅包括命令名称及其选项。最后一个字段应该设置为 0。

在我们的示例中，内建命令名为 tty，对应的 C 函数为 tty_builtin，帮助数组为 tty_doc。用法字符串为 tty [-s]。最终的描述结构如下所示。

```
struct builtin tty_struct =
    "tty",
    tty_builtin,
    BUILTIN_ENABLED,
    tty_doc,
    "tty [-s]",
    0
};
```

接下来是内建命令的实现代码。

```
tty_builtin (list) WORD_LIST *list;
{
    int opt, sflag;
    char *t;

    reset_internal_getopt ( );
    sflag = 0;
    while ((opt = internal_getopt (list, "s")) != -1)
    {
      switch (opt)
      {
          case 's':
              sflag = 1;
              break;
          default:
              builtin_usage ( );
              return (EX_USAGE);
      }
    }
    list = loptend;

    t = ttyname (0);
    if (sflag == 0)
        puts (t ? t : "not a tty");
    return (t ? EXECUTION_SUCCESS : EXECUTION_FAILURE);
}
```

内建命令的实现函数始终会得到一个指向 WORD_LIST 类型列表的指针。如果内建命令不接受任何选项，则必须调用 no_options(list) 并在继续处理前先检查其返回值。如果返回值为非 0，那么函数应该立即返回值 EX_USAGE。

在处理内建命令选项时，必须坚持使用 internal_getopt，而不是标准 C 函数库中的 getopt。另外，还必须先调用 reset_internal_getopt 来重置选项处理。

选项处理以标准方式执行，除非选项不正确，这种情况下应返回 EX_USAGE。处理完选项后，剩下的所有参数都由 loptend 指向。执行完函数后，应该返回值 EXECUTION_SUCCESS 或 EXECUTION_FAILURE。

对于我们自己的内置命令 tty 来说，只需要调用标准 C 函数库例程 ttyname，如果未指定 -s 选项，则输出 TTY 的名称（如果设备并非

TTY，输出“not a tty”）。然后该函数会根据 ttyname 的调用结果返回成功或失败。

最后一个主要部分是定义帮助信息。这其实就是一个字符串数组而已，其中最后一个数组元素为 NULL。在内建命令上运行 help 时，所有字符串会逐个输出到标准输出。因此，你应该将字符串长度限制为 76 个字符或更少（80 个字符的标准显示宽度减去 4 个字符的边距）。tty 中的帮助信息如下所示。

```
char *tty_doc[] = {
  "tty writes the name of the terminal that is opened for standard",
  "input to standard output. If the `-s' option is supplied, nothing",
  "is written; the exit status determines whether or not the
standard",
  "input is connected to a tty.",
(char *)NULL
};
```

最后一件事是，在代码中添加必要的 C 头文件，包括 stdio.h 以及 bash 的头文件 config.h、builtins.h、shell.h、bashgetopt.h。

例 16-9 展示了完整的 C 程序。

例 16-9 ch16/builtin_tty.c

```
# 实例文件：builtin_tty.c

#include "config.h"
#include <stdio.h>
#include "builtins.h"
#include "shell.h"
#include "bashgetopt.h"


extern char *ttyname ( );

tty_builtin (list)
    WORD_LIST *list;
{
    int opt, sflag;
    char *t;

    reset_internal_getopt ( );
    sflag = 0;
    while ((opt = internal_getopt (list, "s")) != -1)
    {
        switch (opt)
```

```
        {
            case 's':
                sflag = 1;
                break;
            default:
                builtin_usage ( );
                return (EX_USAGE);
        }
    }
    list = loptend;

    t = ttyname (0);
    if (sflag == 0)
        puts (t ? t : "not a tty");
    return (t ? EXECUTION_SUCCESS : EXECUTION_FAILURE);
}

char *tty_doc[] = {
    "tty writes the name of the terminal that is opened for standard",
    "input to standard output. If the `-s' option is supplied,
nothing",
    "is written; the exit status determines whether or not the
standard",
    "input is connected to a tty.",
    (char *)NULL
};

struct builtin tty_struct = {
    "tty",
    tty_builtin,
    BUILTIN_ENABLED,
    tty_doc,
    "tty [-s]",
    0
};
```

现在我们需要将该程序编译并链接成一个动态共享对象。遗憾的是，不同的系统有不同的动态共享对象编译方法。

configure 脚本应该自动将正确的命令放入 Makefile。如果由于某种原因未能实现，表 16-1 列出了在一些常见系统中编译及链接 tty.c 所需要的命令。用 bash 安装包的顶层路径替换 archive。

表16-1：常见系统中编译及链接tty.c所需要的命令

系统	命令

系统	命令
SunOS 4	cc -pic -I*archive* -I*archive*/builtins -I*archive*/lib -c tty.c ld -assert pure-text -o tty tty.o
SunOS 5	cc -K pic -I*archive* -I*archive*/builtins -I*archive*/lib -c tty.c cc -dy -z text -G -i -h tty -o tty tty.o
SVR4、SVR4.2、Irix	cc -K PIC -I*archive* -I*archive*/builtins -I*archive*/lib -c tty.c ld -dy -z text -G -h tty -o tty tty.o
AIX	cc -K -I*archive* -I*archive*/builtins -I*archive*/lib -c tty.c ld -bdynamic -bnoentry -bexpall -G -o tty tty.o
Linux	cc -fPIC -I*archive* -I*archive*/builtins -I*archive*/lib -c tty.c ld -shared -o tty tty.o
NetBSD、FreeBSD	cc -fpic -I*archive* -I*archive*/builtins -I*archive*/lib -c tty.c ld -x -Bshareable -o tty tty.o

编译及链接完毕后，你应该能够得到一个名为 tty 的共享对象。输入 enable -f tty tty 就可以将其载入 bash。可以随时用 -d 选项删除已装载的内建命令，例如，enable -d tty。

可以将多个内建命令放入单个共享对象，每个内建命令的 3 个主要部分都位于相同的 C 文件即可。但是，bash 是将共享对象作为整体载入的，因此，如果你要求 bash 从包含 20 个内建命令的共享对象中装载其中一个，它会将这 20 个内建命令全部载入（但只启用一个）。最好保持较小的内建命令数量，以节省不必要的内存占用，同时将功能相似的内建命令（如 pushd、popd、dirs）进行分组，如果用户启用其中一个，其他的也会全部被载入内存，以备启用。

16.18.4 参考

- bash tarball（2.0 版本以上）中的 ./examples/loadables

16.19 改善可编程补全

本实例中的内容直接改编自 Cameron Newham 所著的 *Learning the bash Shell, 3rd Edition*（O'Reilly 出版）一书。

16.19.1 问题

你对 bash 的可编程补全钟爱有加，但希望该功能可以更多地考虑到上下文，尤其针对经常使用的那些命令。

16.19.2 解决方案

查找并安装其他可编程补全库，或者自己写一个。bash tarball 的 ./examples/complete 中也提供了一些示例。有些发行版（如 SUSE）在 /etc/profile.d/complete.bash 中有自己的版本。但毫无疑问，最大且最广为人知的第三方库出自 Ian Macdonald 之手，你可以从 GitHub 网站 scop/bash-completion 页面下载 tarball 或 RPM 格式。该库已经包含在 Debian（以及 Ubuntu 和 Mint 等衍生版本）中，而且也能在 Fedora Extras 和其他第三方仓库中找到。

> 根据 Ian 的自述："不少补全功能都假定其所调用的各种文本实用工具（如 grep、sed、awk）均为 GNU 版本。你的实际情况可能有所不同。"

在撰写本书之时，可编程补全库 bashcompletion-20060301.tar.gz 提供了 103 个模块。其中包括：

- bash alias completion
- bash export completion
- bash shell function completion
- chown(1) completion
- chgrp(1) completion
- Red Hat & Debian GNU/Linux if{up,down} completion

- cvs(1) completion
- rpm(8) completion
- chsh(1) completion
- chkconfig(8) completion
- ssh(1) completion
- GNU make(1) completion
- GNU tar(1) completion
- jar(1) completion
- Linux iptables(8) completion
- tcpdump(8) completion
- ncftp(1) bookmark completion
- Debian dpkg(8) completion
- Java completion
- PINE address-book completion
- mutt completion
- Python completion
- Perl completion
- FreeBSD package management tool completion
- mplayer(1) completion
- gpg(1) completion
- dict(1) completion
- cdrecord(1) completion
- yum(8) completion
- smartctl(8) completion
- vncviewer(1) completion
- svn completion

16.19.3 讨论

可编程补全是 bash 2.04 引入的特性。通过提供补全机制钩子，它扩展了内建的文本补全功能。这意味着你几乎可以编写出任何需要的补全功能。例如，如果能够在输入 man 命令时按下 Tab 键，并列出手册中的各节，岂不是很妙？可编程补全能做到的远不止如此。

本实例仅着眼于可编程补全的基础部分。如果想要进一步深入研究，编写自己的补全代码，可以先检查其他人开发的补全命令库，看看有没有现成或者可用的样例。要是你需要自己动手，我们只概述使用补全机制所需要的基本命令和过程。

要想以特定方式补全文本，你得先告诉 shell 按下 Tab 键时应该怎么做。这是通过 complete 命令完成的。

complete 的主要参数可以是命令名称或者其他需要进行文本补全的内容。例如，我们来看一下可以解压各种压缩文件的实用工具 gunzip。通常，如果输入：

```
gunzip [TAB][TAB]
```

可以得到一系列补全后的备选文件，其中包括各种并不适合 gunzip 处理的文件。其实你真正想要的是 gunzip 能够处理的那部分。可以用 complete 进行如下设置：

```
complete -A file -X '!*.@(Z|gz|tgz)' gunzip
```

注意，要想让 @(Z|gz|tgz) 发挥作用，需要通过 shopt -s extglob 启用扩展模式匹配。

这里我们告诉补全机制，希望能在输入 gunzip 命令时做一些特殊处理。-A 选项代表操作（action），能够接受各种参数。这个例子使用了 file 参数，以要求补全机制提供有可能补全的文件列表。接下来从中选择 gunzip 能够处理的那些文件。这一步通过 -X 选项来实现，该选项以过滤器模式为参数。应用到补全列表时，过滤器会删除与该模式匹配的所有文件名，也就是说，剩下的都是与模式不匹配的。gunzip 能解压包括扩展名 .Z、.gz、.tgz 在内的多种文件类型。我们希望能够匹配这三种扩展名之一。然后还得使用 ！将含义取反（记住，过滤器删除的是匹配指定模式的文件名）。

用 complete 安装该补全功能前，可以先通过 compgen 命令试验一下，看看效果如何。

```
compgen -A file -X '!*.@(Z|gz|tgz)'
```

这会产生一系列补全字符串（假设当前目录下有一批采用这种扩展名的文件）。compgen 有助于测试过滤器的效果，看看都能产生什么样的补全字符串。如果需要更复杂的补全，同样少不了它。随后我们还会看到另一个例子。

无论是通过读取包含该命令的脚本，还是在命令行上直接执行，一旦执行上面的 complete 命令，就可以将扩展后的补全机制与 gunzip 命令一起使用。

```
$ gunzip [TAB][TAB]
archive.tgz archive1.tgz file.Z
$ gunzip
```

你可能会看到还有其他可以实现的操作。如何为某个命令的特定选项提供可能的参数列表？例如，kill 命令可以接受进程 ID，但也能够接受以连字符（-）起始的信号名称或出现在 -n 之后的信号名称。我们可以补全 PID，但如果有连字符或 -n，那就得补全信号名称。

这要比先前那个单行的例子略复杂些。我们需要编写一些代码来区分已经输入的内容。除此之外，还得获得 PID 和信号名称。我们将相关代码放入函数，并通过补全机制调用该函数。调用函数的代码如下所示，我们称其为 _kill。

```
complete -F _kill kill
```

-F 选项告知 complete 要在执行 kill 命令的文本补全时调用函数 _kill。接下来是编写函数，如例 16-10 所示。

例 16-10 ch16/func_kill

```
# 实例文件：func_kill

_kill() {
    local cur
        local sign
    COMPREPLY=( )
    cur=${COMP_WORDS[COMP_CWORD]}

    if (($COMP_CWORD == 2)) && [[ ${COMP_WORDS[1]} == -n ]]; then
        # 返回可用的信号列表
            _signals
    elif (($COMP_CWORD == 1 )) && [[ "$cur" == -* ]]; then
        # 返回可用的信号列表
        sign="-"
        _signals
         else
        # 返回可用的PID列表
        COMPREPLY=( $( compgen -W '$( command ps axo pid | sed 1d )'
$cur ) )
```

```
    fi
}
```

除了用到一些特殊的环境变量及调用了函数 _signals（我们马上会讲到该函数），这段代码非常标准。

变量 $COMPREPLY 用于保存返回给补全机制的结果。该变量是包含一系列补全字符串的数组，其初始值为空。

局部变量 $cur 是一个用于提高代码可读性的便利变量，因为其中的值会在多处用到。这个值来自于数组 $COMP_WORDS 的某个元素。该数组保存的是当前命令行中的各个单词。$COMP_CWORD 是指向其的索引，可以通过它获得当前光标的位置。$cur 的值是当前光标所在的单词。

第一个 if 语句测试 kill 命令之后是否为 -n 选项。如果首个单词是 -n 且当前位于第二个单词，那么我们需要为补全机制提供信号名称列表。

第二个 if 语句也差不多，只不过这次关注的是补全当前单词，该单词以连字符起始，之后可以是任何内容。这个 if 语句的主体部分还是调用 _signals，但这次将变量 sign 设置为连字符。介绍 _signals 函数时，你就明白为什么要这么做了。

剩下的 else 语句块返回进程 ID 列表。它用 compgen 命令帮助创建补全字符串数组。首先，执行 ps 命令来获得 PID 列表。接着通过管道将其传给 sed，以删除第一行（也就是标题行）。然后，将结果作为参数传给 compgen 的 -W 选项，该选项接受的是一个单词列表。compgen 返回与变量 $cur 的值匹配的所有补全字符串，得到的数组会保存在 $COMREPLY。

这里 compgen 非常重要，因为不能只是将 ps 所提供的整个 PID 列表返回。也许用户已经输入了一部分 PID，正尝试将其补全。由于这部分 PID 会保存在变量 $cur 中，因此 compgen 会将结果限制为匹配或部分匹配该变量值的那些 PID。例如，如果 $cur 的值是 5，那么 compgen 仅返回以 5 起始的值，如 5、59 或 562。

终于可以讲讲 _signals 函数了（参见例 16-11）。

例 16-11 ch16/func_signals

```
# 实例文件：func_signals

_signals() {
    local i
    COMPREPLY=( $( compgen -A signal SIG${cur#-} ))

    for (( i=0; i < ${#COMPREPLY[@]}; i++ )); do
        COMPREPLY[i]=$sign${COMPREPLY[i]#SIG}
    done
}
```

虽然可以用 compgen 的 -A 信号来获得信号名称列表，但遗憾的是，这些名称所采用的形式并没有多大的用武之地，我们无法通过其直接生成信号名称数组。compgen 生成的信号名称以字母“SIG”开头，但 kill 命令不需要这 3 个字母。_signals 函数负责将信号名称数组赋给 $COMPREPLY，还可以有选择地在名称前加上一个连字符。

首先，我们用 compgen 生成信号名称列表。每个名称均以字母“SIG”开头。为了使 complete 能够在用户开始输入名称时提供正确的子集，我们在 $cur 的值前添加了“SIG”。另外，我们还借机删除了出现在该值前面的连字符，以便得以匹配。

接着，遍历数组，删除每个元素的“SIG”并根据需要添加连字符（变量 sign 的值）。

complete 和 compgen 都有许多其他选择和操作，远不止此处介绍的这些。要想进一步了解可编程补全，建议你查阅 Bash Reference Manual，还可以从 Internet 或 Bash tarball 的 ./examples/complete 中查看一些示例。

16.19.4 参考

- help complete
- help compgen
- bash tarball（2.04 版本以上）中的 ./examples/complete

16.20 正确使用初始化文件

16.20.1 问题

你想知道所有的初始化（rc[5]）文件都是怎么回事。

[5] rc 是“run commands”的缩写。——译者注

16.20.2 解决方案

以下是这些文件及其用途的备忘清单。根据具体的设置，有些文件可能并不存在于你的系统中。默认使用 bash 的系统（如 Linux）通常不会少什么文件，而默认使用其他 shell 的系统往往会缺其中一部分。

/etc/profile

用于 Bourne 和类似登录 shell 的全局登录环境文件。我们建议不要碰这个文件，除非你是系统管理员，清楚地知道自己在干什么。

/etc/bashrc（Red Hat）或 /etc/bash.bashrc（Debian）

用于交互式 bash 子 shell 的全局环境文件。我们建议不要碰这个文件，除非你是系统管理员，清楚地知道自己在干什么。

/etc/bash_completion

如果存在，这个文件几乎肯定是 Ian Macdonald 的可编程补全库的配置文件（参见 16.19 节）。建议了解一下该文件，它非常酷。

/etc/profile.d/bash_completion.sh 和 /etc/bash_completion.d

其他可能存在的特定发行版的 bash 补全文件。如今多见于 Fedora，也有可能是其他系统。

/etc/inputrc

GNU readline 的全局配置文件。如果应用范围是整个系统，建议调校该文件（假如你是系统管理员）；如果只用于个人，则建议调校 ~/.inputrc（参见 16.22 节）。此文件不会直接执行或被读入，而是通过 readline、$INPUTRC 以及 $include（或 bind -f）被读取。注意，其中也许会包含引入其他 readline 文件的 include 语句。

~/.bashrc

交互式 bash 子 shell 的个人环境文件。建议在其中放置别名、函数以及各式各样的命令行提示符。

~/.bash_profile

bash 登录 shell 的个人配置文件。建议一定要在其中读入 ~/.bashrc，剩下的就可以忽略了。

~/.bash_login

Bourne 登录 shell 的个人配置文件。仅当 ~/.bash_profile 不存在时，bash 才会使用该文件。建议忽略它。

~/.profile

Bourne 登录 shell 的个人配置文件。仅当 ~/.bash_profile 和 ~/.bash_login 均不存在时，bash 才会使用该文件。建议忽略，除非你使用的其他 shell 要用到它。

~/.bash_history

shell 命令历史记录的默认存储文件。建议你使用历史记录工具（参见 16.14 节）来处理该文件，不要尝试直接编辑。这只是一个数据文件，不会被执行或读入。

~/.bash_logout

在用户注销时执行。建议你将所有的清理例程（参见 17.7 节）都放入其中。该文件仅在干净注销时才会执行（如果会话因广域网连接掉线而中止，则不会执行）。

~/.inputrc

GNU readline 的个人定制文件。建议根据需要调校该文件（参见 16.22 节）。此文件不会直接执行或被读入，而是通过 readline、$INPUTRC 以及 $include（或 bind -f）被读取。注意，其中也许会包含引入其他 readline 文件的 include 语句。

我们意识到要遵循这份清单有点棘手；每种操作系统或发行版可能都有所不同，这完全取决于厂商如何编写这些文件。要想真正了解你的系统是如何运作的，需要将上面列出的所有文件逐个读一遍。你还可以将 echo

name_of_file_>&2 临时添加到所有会被执行或读入的文件（也就是跳过 /etc/inputrc、~/.inputrc 以及 /.bash_history）的第一行。注意，这可能会干扰到某些程序（尤其是 scp 和 rsync），它们会被 STDOUT 或 STDERR 上的额外输出搞晕，因此要在完成测试后删除这些语句。更多详细信息参见 16.21 节中的警告信息。

下面的表 16-2 仅作为指南，因为你的系统未必就是如此。（除了表 16-2 中列出的与登录相关的 rc 文件，干净地注销交互式会话时，还会用到 rc 文件~/.bash_logout。）

表16-2：Ubuntu 6.10和Fedora Core 5中与bash登录相关的rc文件

Interactive login shell	Interactive non-login shell (bash)	Noninteractive shell (script) (bash /dev/null)	Noninteractive (bash -c ':')
Ubuntu 6.10:	Ubuntu 6.10:	Ubuntu 6.10:	Ubuntu 6.10:
/etc/profile		N/A	N/A
/etc/bash.bashrc	/etc/bash.bashrc		
~/.bash_profile[a] ~/.bashrc	~/.bashrc		
/etc/bash_completion	/etc/bash_completion		
Fedora Core 5:	Fedora Core 5:	Fedora Core 5:	Fedora Core 5:
/etc/profile[b] [c]		N/A	N/A

Interactive login shell	Interactive non-login shell (bash)	Noninteractive shell (script) (bash /dev/null)	Noninteractive (bash -c ':')
/etc/profile.d/colorls.sh /etc/profile.d/glib2.sh /etc/profile.d/krb5.sh /etc/profile.d/lang.sh /etc/profile.d/less.sh /etc/profile.d/vim.sh /etc/profile.d/which-2.sh ~/.bash_profile[d] ~/.bashrc	~/.bashrc		
/etc/bashrc	/etc/bashrc		

a. 如果没有找到~/.bash_profile，则依次尝试 ~/.bash_login 或 ~/.profile。

b. 如果未设置 $INPUTRC 且 ~/.inputrc 不存在，则将 $INPUTRC 设置为 /etc/inputrc。

c. Red Hat 的 /etc/profile 也会读入 /etc/profile.d/*.sh，详见 4.10 节。

d. 如果没有找到 ~/.bash_profile，则依次尝试 ~/.bash_login 或 ~/.profile。

更多细节参见 Bash Reference Manual 中的“Bash Startup Files”一节。

16.20.3 讨论

在 Unix 或 Linux 中，其中一个棘手的问题是，当你偶尔想对整个系统进行改动时，弄清楚在哪里修改像 $PATH 或提示符这样的东西。不同的操作系统（甚至不同的版本）有不同的设置位置。下列命令基本上能帮你找出在系统的什么地方设置 $PATH。

```
grep 'PATH=' /etc/{profile,*bash*,*csh*,rc*}
```

如果行不通，那就只能用 grep 搜索 /etc 下的所有文件了。

```
find /etc -type f | xargs grep 'PATH='
```

注意，不像书中的大部分代码，上面这行最好以 root 身份运行。普通用户身份不是不能运行，虽然也能得到一部分结果，但可能会有所遗漏，而且几乎肯定会得到“Permission denied”（权限被拒绝）错误。

另一件棘手的事情是搞明白你能改动什么、在哪里进行个人设置。我们希望本章在这方面给你带来了更多的好想法。

16.20.4 参考

- man grep
- man find
- man xargs
- Bash Reference Manual 中的“Bash Startup Files”一节
- 16.4 节
- 16.14 节
- 16.19 节
- 16.21 节
- 16.22 节
- 17.7 节

16.21 创建自包含的可移植rc文件

16.21.1 问题

你在多台机器上工作，对其中有些机器拥有有限的或完全的 root 控制权，而有些则没有，你想要复制一致的 bash 环境，但同时仍然允许根据操作系统、机型或其他（如工作、家庭）标准进行自定义设置。

16.21.2 解决方案

将所有的自定义内容放进 settings 子目录下的文件中，将该目录复制或同步到 ~/ 或 /etc 等位置，并在必要时使用 include 语句和符号链接（如 ln -s ~/settings/screenrc~/.screenrc）。在自定义文件中按照逻辑考虑各种条件，如操作系统、位置等。

或许你还会选择不给文件名添加前导点号，以便更容易管理自定义文件。我们在 1.7 节中已经看到过，ls 默认不显示带有前导点号的文件，这样一来，显示目录内容时会更整洁一些。因为我们使用了专门存放配置文件的目录，所以就没必要再加点号了。注意，/etc 通常也不使用点号文件的原因也在于此。

可以参考 16.22 节中的上手样例。

16.21.3 讨论

我们来看看研究此解决方案时使用的假设和准则。首先，假设如下所示。

- 你所在的环境比较复杂，仅对其中的部分机器拥有控制权。
- 对于能够控制的机器，用一台机器导出 /opt/bin，其他机器通过 NFS 将其挂载，因此所有的配置文件都在其中。之所以选择 /opt/bin，是因为它形式简短，相较于 /usr/local/bin，它与已有目录发生冲突的可能性较小，你也可以随意使用其他合理的目录。
- 对于只有部分控制权的机器，使用 /etc 中具有系统范围的配置。
- 对于没有控制权的机器，使用~/ 中的点号文件。
- 一些设置随机器和环境（例如，家庭或工作）不同而异。

准则如下所示。

- 在操作系统和环境之间转移配置文件时，尽可能少做修改。
- 补充但不替代操作系统默认的或系统管理员提供的配置。
- 提供足够的灵活性来处理设置冲突所带来的需求（例如，工作和家庭 CVS）。

虽然可能倾向于在配置文件中加入 echo 语句，以了解发生了什么，但要小心。这样做的话，scp、rsync 以及其他类似 rsh 的程序会运行失败，产生莫名其妙的错误信息。

```
# scp
# protocol error: bad mode

# rsync
# protocol version mismatch - is your shell clean?
# (see the rsync manpage for an explanation)
# rsync error: protocol incompatibility (code 2) at compat.
# c(62)
```

ssh 能够正常工作的原因是，它实际上是交互式的，输出信息显示在屏幕上，这不会扰乱数据流。详细解释参见 14.22 节中的讨论。

为了进行调试，将下面两行代码放在 /etc/profile 或 ~/.bash_profile 的顶部附近，但记得看一下我们刚刚给出的有关扰乱数据流的警告。

```
export PS4='+xtrace $LINENO: '
set -x
```

也可以用 set -x 作为替代（或补充），将下列代码键入任意或全部的配置文件中。

```
# 例如，在~/.bash_profile中
case "$-" in
    *i*) echo "$(date '+%Y-%m-%d_%H:%M:%S_%Z') Interactive" \
               "~/.bash_profile ssh=$SSH_CONNECTION" >> ~/rc.log ;;
    *  ) echo "$(date '+%Y-%m-%d_%H:%M:%S_%Z') Noninteractive" \
               "~/.bash_profile ssh=$SSH_CONNECTION" >> ~/rc.log ;;
esac

# 在~/.bashrc中
case "$-" in
    *i*) echo "$(date '+%Y-%m-%d_%H:%M:%S_%Z') Interactive" \
               "~/.bashrc ssh=$SSH_CONNECTION" >> ~/rc.log ;;
    *  ) echo "$(date '+%Y-%m-%d_%H:%M:%S_%Z') Noninteractive" \
               "~/.bashrc ssh=$SSH_CONNECTION" >> ~/rc.log ;;
esac
```

因为不会向终端产生任何输出，所以也不会干扰到命令。在一个会话中执行 tail -f ~/rc.log 命令，在其他地方执行有麻烦的命令，以判断使用了哪个配置文件。这样就可以轻松地追溯到问题的源头。

如果要修改配置文件，我们强烈建议你打开两个会话。在其中一个会话中完成所有改动，注销该会话，接着再返回。如果出现问题导致无法重新登录，就在另一个会话中修复，然后在第一个会话再尝试登录。不要将两个会话都注销了，直到百分之百确定能够正常登录。如果所做的改动会影响到 root 用户，那么这个过程还得再重复一遍。

你确实需要注销并重新登录。读入（sourcing）修改后的文件是可行的，先前环境中残留的配置或许还能让一切暂时正常，等到干净启动（start clean）时，问题就来了。可以根据需要改动运行环境，但在准备好测试前，别更改文件；否则，你有可能会忘了这茬，要是哪个地方出了岔子，搞不好就把自己关在系统外了。

16.21.4 参考

- 1.7 节
- 14.23 节
- 16.20 节
- 16.22 节

16.22 自定义配置入门

16.22.1 问题

你想调整环境，但又不是特别确定从何处下手。

16.22.2 解决方案

下面的样例能帮助你了解可以做什么。我们遵循 16.21 节中的建议，将自定义配置分开，以便于轻松撤销，并实现系统之间的可移植性。

对于系统范围的配置文件设置，可以将例 16-12 中的内容添加到 /etc/profile。由于真正的 Bourne shell 也会使用该文件，如果你所在的是非 Linux 系统，注意不要使用任何 bash 独有的特性（例如，不要用 . 代替 source）。Linux 使用 bash 作为默认 shell，/bin/sh 和 /bin/bash 均指向 bash（在 Ubuntu 6.10+ 中，默认 shell 是 dash）。对于针对用户的设置，依次将其添加到 ~/.bash_profile、~/.bash_login、~/.profile 中的某一个即可（以先出现者为准）。

例 16-12 ch16/add_to_bash_profile

```
# 实例文件：add_to_bash_profile
# 将下列代码添加到你的~/.bash_profile

# 如果运行在bash中，进行搜索，然后读入设置
# 你也可以硬编码$SETTINGS，但这样不够灵活
if [ -n "$BASH_VERSION" ]; then
    for path in /opt/bin /etc ~ ; do
        # 先找到哪个就用哪个
        if [ -d "$path/settings" -a -r "$path/settings" -a -x
"$path/settings" ]
        then
            export SETTINGS="$path/settings"
```

```
        fi
    done
    source "$SETTINGS/bash_profile"
    #source "$SETTINGS/bashrc"      # 如果有必要
fi
```

对于系统范围的环境设置，可以将例 16-13 的内容添加到 /etc/bashrc（或 /etc/bash.bashrc）中。

例 16-13 ch16/add_to_bashrc

```
# 实例文件: add_to_bashrc
# 将此代码添加到你的~/.bashrc中

# 如果运行在bash中且尚未进行设置，
# 则进行搜索，然后读入设置
# 你也可以硬编码$SETTINGS，但这样不够灵活
if [ -n "$BASH_VERSION" ]; then
    if [ -z "$SETTINGS" ]; then
        for path in /opt/bin /etc ~ ; do
            # 先找到哪个就用哪个
            if [ -d "$path/settings" -a -r "$path/settings" -a -x
"$path/settings" ]
            then
                export SETTINGS="$path/settings"
            fi
        done
    fi
    source "$SETTINGS/bashrc"
fi
```

例 16-14 是一个 bash_profile 样例。

例 16-14 ch16/bash_profile

```
# 实例文件：bash_profile

# settings/bash_profile：设置登录shell环境
# 要想重新读入（并使得对该文件的改动生效），可以使用：
# source $SETTINGS/bash_profile

# 仅当为终端交互式bash!
[ -t 1 -a -n "$BASH_VERSION" ] || return

# 故障保护。应该在被调用时设置，但如果没有的话，
# 错误消息"not found"应该够清晰了。
# 行首的':'是为了避免变量扩展后被当作程序运行
```

```
: ${SETTINGS:='SETTINGS_variable_not_set'}

# 仅作调试之用——会破坏scp、rsync
# echo "Sourcing $SETTINGS/bash_profile..."
# export PS4='+xtrace $LINENO: '
# set -x

# 调试/日志记录——不会破坏scp、rsync
#case "$-" in
#    *i*) echo "$(date '+%Y-%m-%d_%H:%M:%S_%Z') Interactive" \
#               "$SETTINGS/bash_profile ssh=$SSH_CONNECTION" >>
~/rc.log ;;
#    *  ) echo "$(date '+%Y-%m-%d_%H:%M:%S_%Z') Noninteractive" \
#               "$SETTINGS/bash_profile ssh=$SSH_CONNECTION" >>
~/rc.log ;;
#esac

# 使用keychain shell脚本来管理ssh-agent（如果可用的话）。
# 如果没有使用的话，应该考虑将其加入
for path in $SETTINGS ${PATH//:/ }; do
    if [ -x "$path/keychain" ]; then
        # 载入默认的id_rsa和/或id_dsa密钥，根据需要添加其他密钥
        # 参见--clear --ignore-missing --noask --quiet --time-out
        $path/keychain ~/.ssh/id_?sa ~/.ssh/${USER}_?sa
        break
    fi
done


# 将交互式子shell的自定义配置也应用于登录shell。
# /etc中的系统配置文件可能已经这么做了。
# 如果没有的话，最好是在任意位置手动执行：
# source "$SETTINGS/bash_profile"
# 但为了以防万一……
# for file in /etc/bash.bashrc /etc/bashrc ~/.bashrc; do
#     [ -r "$file" ] && source $file && break  # 使用最先找到的那个文件
#done


# 执行特定站点或特定主机配置
case $HOSTNAME in
    *.company.com     ) # source $SETTINGS/company.com
                        ;;
    host1.*           ) # host1的相关配置
                        ;;
    host2.company.com ) # source .bashrc.host2
                        ;;
    drake.*           ) # echo DRAKE in bash_profile.jp!
                        ;;
esac

# 最后执行这一步。如果我们退出screen，会返回到一个完全配置过的会话。
```

```
# screen会话也会得以配置，如果不退出的话，该会话就不会膨胀

# 仅当处于交互模式且未运行screen，同时~/.use_screen也存在的情况下才执行
if [ "$PS1" -a $TERM != "screen" -a "$USING_SCREEN" != "YES" -a -f
~/.use_screen ]; \
then
    # 我们更愿意在这里使用type -P，但是该命名是在bash-2.05b版中加入的，
    # 我们对所使用的系统没有控制权，其bash版本低于2.05b。我们也无法轻易地使用
which，
    # 因为在某些系统中，该命令会产生文件找到与否的输出
    for path in ${PATH//:/ }; do
        if [ -x "$path/screen" ]; then
            # 如果screen(1)存在且可执行，则运行我们的包装程序
            [ -x "$SETTINGS/run_screen" ] && $SETTINGS/run_screen
        fi
    done
fi
```

例 16-15 是一个 bashrc 样例（我们知道这段代码确实挺长的，但最好还是读一下，了解其中的思路）。

例 16-15 ch16/bashrc

```
# 实例文件：bashrc

# settings/bash_profile：设置子shell环境
# 要想重新读入（并使得对该文件的改动生效），可以使用：
# source $SETTINGS/bashrc

# 仅当为终端交互式bash!
[ -t 1 -a -n "$BASH_VERSION" ] || return

# 故障保护。应该在被调用时设置，但如果没有的话，
# 错误消息"not found"应该够清晰了。
# 行首的':'是为了避免变量扩展后被当作程序运行
: ${SETTINGS:='SETTINGS_variable_not_set'}

# 仅作调试之用——会破坏scp、rsync
# echo "Sourcing $SETTINGS/bash_profile..."
# export PS4='+xtrace $LINENO: '
# set -x

# 调试/日志记录——不会破坏scp、rsync
# case "$-" in
#     *i*) echo "$(date '+%Y-%m-%d_%H:%M:%S_%Z') Interactive" \
#                 "$SETTINGS/bashrc ssh=$SSH_CONNECTION" >> ~/rc.log ;;
#     *  ) echo "$(date '+%Y-%m-%d_%H:%M:%S_%Z') Noninteractive" \
#                 "$SETTINGS/bashrc ssh=$SSH_CONNECTION" >> ~/rc.log ;;
#esac
```

```
# 理论上，还要读入/etc/bashrc (/etc/bash.bashrc)或~/.bashrc，
# 将所有的设置也应用于登录shell。在实践中，这些设置是否仅在
# 某些时刻有效（例如，在子shell中），则需要核实

# 读入SSH和GPG代理的keychain文件（如果存在的话）
[ -r "$HOME/.keychain/${HOSTNAME}-sh" ] \
  && source "$HOME/.keychain/${HOSTNAME}-sh"
[ -r "$HOME/.keychain/${HOSTNAME}-sh-gpg" ] \
  && source "$HOME/.keychain/${HOSTNAME}-sh-gpg"

# 设置一些更实用的提示符
# 交互式命令行提示符
# 如果确为交互式shell，仅设置其中一种提示符，因为很多人（有时也包括我们）
# 使用类似于 if [ "$PS1" ]; then的方式来测试shell是否为交互式
case "$-" in
    *i*)
        #export PS1='\n[\u@\h t:\l l:$SHLVL h:\! j:\j v:\V]\n$PWD\$ '
        #export PS1='\n[\u@\h:T\l:L$SHLVL:C\!:\D{%Y-%m-
%d_%H:%M:%S_%Z}]\n$PWD\$ '
        export PS1='\n[\u@\h:T\l:L$SHLVL:C\!:J\j:\D{%Y-%m-
%d_%H:%M:%S_%Z}]\n$PWD\$ '
        #export PS2='> ' # Secondary (i.e. continued) prompt

        #export PS3='Please make a choice: '           # select提示符
        #export PS4='+xtrace $LINENO: '                # xtrace提示符
        export PS4='+xtrace $BASH_SOURCE::$FUNCNAME-$LINENO: ' #
xtrace提示符

        # 如果是xterm，则将标题设置为user@host:dir
        case "$TERM" in
            xterm*|rxvt*)
                PROMPT_COMMAND='echo -ne
"\033]0;${USER}@${HOSTNAME}:$PWD\007"'
            ;;
        esac
    ;;
esac

# 确保处理自定义inputrc（如果存在的话）。注意不同的文件名，还有不同的处理顺序，
# 因为我们希望自己的自定义设置覆盖系统文件（如果存在的话）
for file in $SETTINGS/inputrc ~/.inputrc /etc/inputrc; do
    [ -r "$file" ] && export INPUTRC="$file" && break # 使用最先找到的那
个文件
done

# 默认没有核心转储文件
# 另请参见许多Linux系统中的/etc/security/limits.conf
ulimit -S -c 0 > /dev/null 2>&1

# 设置bash历史记录的方方面面
```

```
export HISTSIZE=5000          # 内存中的历史记录栈所能容纳的命令数量
export HISTFILESIZE=5000      # 历史文件中的命令数量
#export HISTCONTROL=ignoreboth # bash < 3，忽略历史记录中的重复命令和以空格
开头的行
export HISTCONTROL='erasedups:ignoredups:ignorespace'
export HISTIGNORE='&:[ ]*'    # bash ≥ 3，忽略历史记录中的重复命令和以空格开
头的行
#export HISTTIMEFORMAT='%Y-%m-%d_%H:%M:%S_%Z=' # bash ≥ 3，时间戳历史文件
shopt -s histappend           # 退出时追加而非覆盖历史记录
shopt -q -s cdspell           # 在交互式shell中使用cd命令时，自动纠正小的拼
写错误
shopt -q -s checkwinsize      # 更新LINES和COLUMNS的值
shopt -q -s cmdhist           # 将由多行组成的命令保存为单个历史记录项
set -o notify # （或者set -b）# 立刻提醒后台作业终止
set -o ignoreeof              # 不允许通过Ctrl-D退出shell

# 其他bash设置
PATH="$PATH:/opt/bin"
export MANWIDTH=80            # 手册页显示宽度，如果COLUMNS=80且使用了less
-N，
                              # 则设置为80以下
export LC_COLLATE='C'         # 设置为传统的C排序顺序（例如，UC优先）
export HOSTFILE='/etc/hosts'  # 使用/etc/hosts进行主机名补全
export CDPATH='.:~/:..:../..' # 类似于$PATH，但是由cd使用
# 注意，$CDPATH中的.不能少，这样cd才能在POSIX模式下正常工作
# 但这也会导致cd将新目录回显至STDOUT!
# 参见上面的cdspell!

# 如果bash补全设置位于默认位置且尚未被导入（例如，未设置
$BASH_COMPLETION_COMPAT_DIR），
# 则导入这些设置。在速度较慢的系统中，这得花上一两秒的时间，你可能未必愿意这么做，
# 即便是在补全设置已经存在的情况下（在Red Hat等不少系统中，默认并非如此）
if [ -z "$BASH_COMPLETION_COMPAT_DIR" ] && ! shopt -oq posix; then
    if [ -f /usr/share/bash-completion/bash_completion ]; then
      . /usr/share/bash-completion/bash_completion
    elif [ -f /etc/bash_completion ]; then
      . /etc/bash_completion
    fi
fi

# 使用lesspipe过滤器（如果存在的话）。这会设置$LESSOPEN变量。
# 将$PATH的分隔符:全部替换为空格，以便在列表中使用
for path in $SETTINGS /opt/bin ~/ ${PATH//:/ }; do
    # 使用'lesspipe.sh'（首选）或'lesspipe'（Debian）中找到的第一个
    [ -x "$path/lesspipe.sh" ] && eval $("$path/lesspipe.sh") && break
    [ -x "$path/lesspipe" ]    && eval $("$path/lesspipe")    && break
done

# 设置其他的less和编辑器偏好
```

```
export LESS="--LONG-PROMPT --LINE-NUMBERS --ignore-case --QUIET --no-
init"
export VISUAL='vi'  # 设置一个应该会始终有效的默认值
# 我们更愿意在这里使用type -P，但是该命名是在bash-2.05b版中加入的，
# 我们对所使用的系统没有控制权，其bash版本低于2.05b。我们也无法轻易地使用which，
# 因为在某些系统中，该命令会产生文件找到与否的输出
#for path in ${PATH//:/ }; do
#    # 如果能找到nano，就覆盖掉VISUAL
#    [ -x "$path/nano" ] \
#      && export VISUAL='nano --smooth --const --nowrap --suspend' &&
break
#done
# 参阅上面部分的注释，了解我们为什么这样写循环
for path in ${PATH//:/ }; do
    # 如果可以的话，将vi设置为vim二进制模式的别名
    [ -x "$path/vim" ] && alias vi='vim -b' && break
done
export EDITOR="$VISUAL"       # 另一种可能
export SVN_EDITOR="$VISUAL"  # Subversion
alias edit=$VISUAL            # 提供在所有系统上使用的命令

# 设置ls选项和别名
# 注意，取决于你的终端仿真器及设置，尤其是ANSI颜色，
# 这些着色效果未必能生效。不过应该也不会造成什么影响。
# 参阅上面部分的注释，了解我们为什么这样写循环
for path in ${PATH//:/ }; do
    [ -r "$path/dircolors" ] && eval "$(dircolors)" \
      && LS_OPTIONS='--color=auto' && break
done
export LS_OPTIONS="$LS_OPTIONS -F -h"
# 使用dircolors可能会造成csh脚本产生"Unknown colorls variable 'do'."错误。
# 原因在于环境变量LS_COLORS中的":do=01;35:"部分
# eval "$(dircolors)"
alias ls="ls $LS_OPTIONS"
alias ll="ls $LS_OPTIONS -l"
alias ll.="ls $LS_OPTIONS -ld"  # 用法：ll. ~/.*
alias la="ls $LS_OPTIONS -la"
alias lrt="ls $LS_OPTIONS -alrt"

# 实用别名
# 移到一个函数处：alias bot='cd $(dirname $(find . | tail -1))'
#alias clip='xsel -b'         # 将内容插入正确的"x"剪贴板
alias gc='xsel -b'            # "GetClip"，从正确的"X"剪贴板获取内容
alias pc='xsel -bi'           # "PutClip"，将内容放入正确的"X"剪贴板
alias clr='cd ~/ && clear'    # 返回$HOME并清屏
alias cls='clear'             # DOS风格的清屏
alias cal='cal -m'            # 将星期一作为每周的第一天
alias copy='cp'               # cp的DOS等价命令
#alias cp='cp -i'             # /root/.bashrc中的默认别名（Red Hat）
alias cvsst='cvs -qn update' # 获取简明的CVS状态（类似于svn st）
```

```
alias del='rm'                  # rm的DOS等价命令
alias df='df --print-type --exclude-type=tmpfs --exclude-
type=devtmpfs'
alias diff='diff -u'            # 将合并输出作为diff的默认输出格式
alias jdiff="\diff --side-by-side --ignore-case --ignore-blank-lines\
  --ignore-all-space --suppress-common-lines" # 实用的GNU diff命令
alias dir='ls'                  # ls的DOS等价命令
alias hu='history -n && history -a' # 读取新的历史记录项，追加当前的历史记录
项
alias hr='hu'                   # "History update"向后兼容于'hr'
alias inxi='inxi -c19'          # （Ubuntu）系统信息脚本
alias ipconfig='ifconfig'       # ifconfig的Windows等价命令
alias lesss='less -S'           # 不折行
alias locate='locate -i'        # locate命令不区分大小写
alias man='LANG=C man'          # 正确显示手册页
alias md='mkdir'                # mkdir的DOS等价命令
alias move='mv'                 # mv的DOS等价命令
#alias mv='mv -i'               # /root/.bashrc中的默认别名（Red Hat）
alias ntsysv='rcconf'           # Debian的rcconf与Red Hat的ntsysv颇为相近
#alias open='gnome-open'        # 使用GNOME处理程序打开文件和URL，参见下面的run
函数定义
alias pathping='mtr'            # mtr是一款网络诊断工具
alias ping='ping -c4'           # 默认只ping 4次
alias r='fc -s'                 # 重新执行上一个以指定字符串开头的命令
alias rd='rmdir'                # rmdir的DOS等价命令
alias randomwords="shuf -n102 /usr/share/dict/words \
  | perl -ne 'print qq(\u\$_);' | column"
alias ren='mv'                  # mv/rename的DOS等价命令
#alias rm='rm -i'               # /root/.bashrc中的默认别名（Red Hat）
alias reloadbind='rndc -k /etc/bind/rndc.key freeze \
  && rndc -k /etc/bind/rndc.key reload && rndc -k /etc/bind/rndc.key
thaw'
  # 编辑过db.*文件之后，重新载入动态BIND区
alias svndiff='meld'            # 挺酷的图形化界面diff，类似于TortoiseMerge
alias svnpropfix='svn propset svn:keywords "id url date"'
alias svnkey='svn propset svn:keywords "id url"'
alias svneol='svn propset svn:eol-style'  # 'native', 'LF', 'CR',
'CRLF'之一
alias svnexe='svn propset svn:executable on'
alias top10='sort | uniq -c | sort -rn | head'
alias tracert='traceroute'      # traceroute的DOS等价命令
alias vzip='unzip -lvM'         # 查看ZIP文件内容
alias wgetdir="wget --no-verbose --recursive --no-parent --no-
directories \
  --level=1"                    # 使用wget抓取整个目录
alias wgetsdir="wget --no-verbose --recursive --timestamping --no-
parent \
  --no-host-directories --reject 'index.*'"  # 抓取目录及其子目录
alias zonex='host -l'           # 提取（倾印）DNS区
```

```
# 日期/时间
alias iso8601="date '+%Y-%m-%dT%H:%M:%S%z'"  # ISO 8601时间
alias now="date       '+%F %T %Z(%z)'"       # 可读性更好的ISO 8601本地时
间
alias utc="date --utc '+%F %T %Z(%z)'"       # 可读性更好的ISO 8601 UTC

alias meminfo='free -m -l -t'   # 查看空闲内存量
alias whatpid='ps auwx | grep'  # 获取PID和进程信息
alias port='netstat -tulanp'    # 显示有网络连接的进程

# 如果该脚本存在且可执行，就创建一个别名，用于获取Web服务器返回的头部信息
for path in ${PATH//:/ }; do
    [ -x "$path/lwp-request" ] && alias httpdinfo='lwp-request -eUd'
&& break
done


# 实用函数

# 使用gnome-open打开文件或URL
function run {
    [ -r "$*" ] && {
        gnome-open "$*" >& /dev/null
    } || {
        echo "'$*' not found or not readable!"
    }
}


#  'perl -c'的Python版
function python-c {
    python -m py_compile "$1" && rm -f "${1}c"
}


# 切换到目录树的最底部
function bot {
    local dir=${1:-.}
    #\cd $(dirname $(find $dir | tail -1))
    \cd $(find . -name CVS -prune -o -type d -print | tail -1)
}


# 创建并切换到新目录
# 用法：mcd (<mode>) <dir>
function mcd {
    local newdir='_mcd_command_failed_'
    if [ -d "$1" ]; then          # 目录已存在，提醒用户……
        echo "$1 exists..."
        newdir="$1"
```

```
    else
        if [ -n "$2" ]; then      # 已经指定了目录的权限模式（mode）
            command mkdir -p -m $1 "$2" && newdir="$2"
        else                      # 普通的mkdir命令
            command mkdir -p "$1" && newdir="$1"
        fi
    fi
    builtin cd "$newdir"          # 直接进入
} # 函数mcd定义完毕


# 简单的命令行计算器
function calc {
    # 仅限整数! --> echo The answer is: $(( $* ))
    # 浮点数
    awk "BEGIN {print \"$* = \" $* }";
    #awk "BEGIN {printf \"$* = %f\", $* }";
} # 函数calc定义完毕
function addup {
    awk '{sum += $1} END {print sum}'
}


# 允许使用cd ...向上跳转2层，使用cd ....向上跳转3层，以此类推。（就像4NT/4DOS）
# 用法：cd ...，以此类推
function cd {

    local option= length= count= cdpath= i= # 局部变量，设置好初始状态

    # 如果存在-L或-P选项，将其保存下来，然后删除
    if [ "$1" = "-P" -o "$1" = "-L" ]; then
        option="$1"
        shift
    fi

    # 我们是否使用了这种特殊的语法？确保$1不为空，
    # 然后匹配$1的前3个字符，看看是否为'...'，
    # 接着尝试替换，以确保不存在斜线。如果失败，则说明没有斜线
    if [ -n "$1" -a "${1:0:3}" = '...' -a "$1" = "${1%/*}" ]; then
        # 我们使用了特殊语法
        length=${#1}  # 假定$1只包含点号，统计点号数量
        count=2       # 'cd ..'仍代表向上跳转1层，因此忽略前2个点号

        # 只要点号尚未用完，就继续向上跳转1层
        for ((i=$count;i<=$length;i++)); do
            cdpath="${cdpath}../" # 构建cd命令用到的路径
        done

        # 执行cd命令
        builtin cd $option "$cdpath"
    elif [ -n "$1" ]; then
```

```
        # 未使用特殊语法，只是普通的cd命令
        builtin cd $option "$*"
    else
        # 未使用特殊语法，只是单独的普通cd命令，可以切换到主目录
        builtin cd $option
    fi
} # 函数cd定义完毕

# 执行特定站点或特定主机配置
case $HOSTNAME in
    *.company.com     ) # source $SETTINGS/company.com
                        ;;
    host1.*           ) # host1的相关配置
                        ;;
    host2.company.com ) # source .bashrc.host2
                        ;;
    drake.*           ) # echo DRAKE in bashrc.jp!
                          export TAPE=/dev/tape
                        ;;
esac
```

例 16-16 是一个 inputrc 样例。

例 16-16 ch16/inputrc

```
# 实例文件：inputrc
# settings/inputrc: readline设置
# 要想重新读入（并使得对该文件的改动生效），可以使用：
# bind -f $SETTINGS/inputrc

# 首先，包含/etc/inputrc中所有的系统范围绑定和变量赋值。
# （如果文件不存在，则静默失败）
$include /etc/inputrc

$if Bash
  # 在进行补全时忽略大小写
    set completion-ignore-case on
  # 补全后的目录名尾部包含斜线
    set mark-directories on
  # 补全后的名称如果是指向目录的符号链接，则名称尾部包含斜线
    set mark-symlinked-directories on
  # 在补全结果的尾部添加指示符号来说明类型
    set visible-stats on
  # 遍历所有可能的补全结果
    "\C-i": menu-complete
  # 将铃声设为可闻
    set bell-style audible
  # 列出可能的补全结果而非响铃
    set show-all-if-ambiguous on
```

```
  # 取自readline文档
  # 便于shell交互操作的各种宏
  # 编辑路径
    "\C-xp": "PATH=${PATH}\e\C-e\C-a\ef\C-f"
  # 准备输入带引号的单词——插入左双引号和右双引号，
  # 然后移动到左引号之后
    "\C-x\"": "\"\"\C-b"
  # 插入反斜线（测试序列和宏中的反斜线转义）
    "\C-x\\": "\\"
  # 引用当前或前一个单词
    "\C-xq": "\eb\"\ef\""
  # 添加绑定以刷新未绑定的行
    "\C-xr": redraw-current-line
  # 编辑当前行中的变量
    #"\M-\C-v": "\C-a\C-k$\C-y\M-\C-e\C-a\C-y="
    "\C-xe": "\C-a\C-k$\C-y\M-\C-e\C-a\C-y="
$endif

# emacs模式的一些默认设置/修改
$if mode=emacs

  # 允许使用Home/End键
    "\e[1~": beginning-of-line
    "\e[4~": end-of-line

  # 允许使用Delete/Insert键
    "\e[3~": delete-char
    "\e[2~": quoted-insert

  # 将"page up"键和"page down"键映射到历史记录的开头/结尾
#    "\e[5~": beginning-of-history
#    "\e[6~": end-of-history

  # 将"page up"键和"page down"键映射为搜索历史记录
#    "\e[5~": history-search-backward
#    "\e[6~": history-search-forward

  # 更漂亮的上下箭头键搜索!
    "\e[A": history-search-backward  ## 上箭头键
    "\e[B": history-search-forward   ## 下箭头键

  # 将"Ctrl-左箭头"和"Ctrl-右箭头"映射为单词移动
  ### 下面这些存在问题，/etc/inputrc是更好的选择
#    "\e[5C": forward-word
#    "\e[5D": backward-word
#    "\e\e[C": forward-word
#    "\e\e[D": backward-word

  # 针对非RH/Debian xterm，不会影响RH/Debian xterm
    "\eOH": beginning-of-line
```

```
    "\eOF": end-of-line

  # 针对FreeBSD控制台
    "\e[H": beginning-of-line
    "\e[F": end-of-line

$endif
```

例 16-17 是一个 bash_logout 样例。

例 16-17 ch16/bash_logout

```
# 实例文件：bash_logout

# settings/bash_logout：在shell注销时执行

# 注销时清屏，避免信息泄露
# 在别处设置为退出陷阱
[ -n "$PS1" ] && clear
```

最后，例 16-18 是一个 run_screen 样例（用于 GNU screen，可能需要自行安装）。

例 16-18 ch16/run_screen

```
#!/usr/bin/env bash
# 实例文件：run_screen
# run_screen：包装脚本，旨在从“配置文件”运行，以便在登录时以友好的菜单执行screen

# 健全性检查
if [ "$TERM" == "screen" -o "$TERM" == "screen-256color" ]; then
    printf "%b" "According to \$TERM = '$TERM' we're *already* using"
\
      " screen.\nAborting...\n"
    exit 1
elif [ "$USING_SCREEN" == "YES" ]; then
    printf "%b" "According to \$USING_SCREEN = '$USING_SCREEN' we're"
      " *already* using screen.\nAborting...\n"
    exit 1
fi

# 变量$USING_SCREEN用于screen未设置$TERM=screen的这种极少数情况。
# 当screen未在TERMCAP或类似库中时就会这样，我们所用的无控制权的Solaris 9
# 主机就是如此。如果无法确定何时处于screen之中，包装脚本会陷入难堪且令人困惑的死循
环

# 生成带有Exit和New选项的列表，查看当前屏幕。
```

```
# 选择列表以空白字符分隔，我们只需要实际的screen会话，
# 所以使用perl删除空白字符，过滤会话，只显示screen -ls输出中的有用信息
available_screens="Exit New $(screen -ls \
  | perl -ne 's/\s+//g; print if s/^(\d+\..*?)(?:\(.*?\))?(\(.*?
\))$/$1$2\n/;')"

# 如果使用运行时反馈，则输出警告信息
run_time_feedback=0
[ "$run_time_feedback" == 1 ] && printf "%b" "
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++
'screen' Notes:

1) Sessions marked 'unreachable' or 'dead' should be investigated and
removed with the -wipe option if appropriate.\n\n"


# 显示选择列表
PS3='Choose a screen for this session: '
select selection in $available_screens; do
    if   [ "$selection" == "Exit" ]; then
        break
    elif [ "$selection" == "New" ]; then
        export USING_SCREEN=YES
        exec screen -c $SETTINGS/screenrc -a \
                    -S $USER.$(date '+%Y-%m-%d_%H:%M:%S%z')
        break
    elif [ "$selection" ]; then
        # 使用cut提取我们需要的部分。相较于echo，
        # 我们更愿意使用here-string [$(cut -d'(' -f1 <<< $selection)]，
        # 但后者仅适用于bash-2.05b+
        screen_to_use="$(echo $selection | cut -d'(' -f1)"
        # 老做法：exec screen -dr $screen_to_use
        # 替换做法：exec screen -x $USER/$screen_to_use
        exec screen -r $USER/$screen_to_use
        break
    else
        printf "%b" "Invalid selection.\n"
    fi
done
```

16.22.3 讨论

可以阅读代码及其注释来了解实现细节。

如果设置 `$PS1` 的时机不对或是用 `clear` 设置了陷阱，则会发生一些有意思的事情。很多人使用类似于下面的代码来测试当前 shell 是否为交互式。

```
if [ -n "$PS1" ]; then
    : Interactive code here
fi
```

如果在非交互式 shell 的情况下随意设置 $PS1，或在设置陷阱时仅用 clear 代替 ["$PS1"] && clear，那么以非交互式使用 scp 或 ssh 时会产生如下错误。

```
# 例如，来自tput
No value for $TERM and no -T specified

# 例如，来自clear
TERM environment variable not set.
```

16.22.4 参考

- 第 17~19 章
- 16.20 节
- 16.21 节
- 17.5 节
- 附录 C

第 17 章　内务及管理任务

这里的实例涵盖了使用或管理计算机时会碰到的各种任务。考虑到其他地方都不是特别适合，所以将它们安排在了本章。

17.1　批量重命名文件

17.1.1　问题

你想要重命名多个文件，但是 `mv *.foo *.bar` 并不管用。或者说，你希望能够以任意方式重命名一批文件。

17.1.2　解决方案

我们在 5.18 节中展示了一个能够修改文件扩展名的简单循环，具体细节可参见 5.18 节。以下是一个 `for` 循环的例子。

```
for FN in *.bad
do
    mv "${FN}" "${FN%bad}bash"
done
```

要是想做一些更随意的修改呢？例如，假设你正在写一本书，希望每章的文件名都遵循某种格式，但这又不符合出版社规定的格式。你可以将文件命名为 *chNN=Title=Author*.odt，然后用简单的 `for` 循环以及包含 `cut` 的命令替换进行重命名。

```
for i in *.odt; do mv "$i" "$(echo "$i" | cut -d'=' -f1,3)"; done
```

17.1.3　讨论

要坚持给文件名参数加上引号，以免其中存在空格。在该解决方案中，为了清晰地分辨都有哪些参数，我们在测试代码时使用了 `echo`

和尖括号（set -x 也可以）。一旦确定一切正常，就可以删除尖括号，并将 echo 替换成 mv。

```
# 测试
$ for i in *.odt; do echo "<$i>" "<$(echo "$i" | cut -d'=' -
f1,3)>"; done
<ch01=Beginning Shell Scripting=JP.odt><ch01=JP.odt>
<ch02=Standard Output=CA.odt><ch02=CA.odt>
<ch03=Standard Input=CA.odt><ch03=CA.odt>
<ch04=Executing Commands=CA.odt><ch04=CA.odt>
[...]

# 更多测试
$ set -x

$ for i in *.odt; do echo "<$i>" "<$(echo "$i" | cut -d'=' -
f1,3)>"; done
++xtrace 1: echo ch01=Beginning Shell Scripting=JP.odt
++xtrace 1: cut -d= -f1,3
+xtrace 535: echo '<ch01=Beginning Shell Scripting=JP.odt>'
'<ch01=JP.odt>'
<ch01=Beginning Shell Scripting=JP.odt><ch01=JP.odt>
++xtrace 1: echo ch02=Standard Output=CA.odt
++xtrace 1: cut -d= -f1,3
+xtrace 535: echo '<ch02=Standard Output=CA.odt>' '<ch02=CA.odt>'
<ch02=Standard Output=CA.odt><ch02=CA.odt>
++xtrace 1: echo ch03=Standard Input=CA.odt
++xtrace 1: cut -d= -f1,3
+xtrace 535: echo '<ch03=Standard Input=CA.odt>' '<ch03=CA.odt>'
<ch03=Standard Input=CA.odt><ch03=CA.odt>
++xtrace 1: echo ch04=Executing Commands=CA.odt
++xtrace 1: cut -d= -f1,3
+xtrace 535: echo '<ch04=Executing Commands=CA.odt>'
'<ch04=CA.odt>'
<ch04=Executing Commands=CA.odt><ch04=CA.odt>

$ set +x
+xtrace 536: set +x
```

本书有很多类似于这样的 for 循环，因为用起来着实方便。棘手的地方在于将正确的值插入 mv 或 cp 的参数。本例以 = 为分隔符，而且只关心第一个字段，因此非常容易。

要想找出需要的值，可以用 ls（或 find）命令列出正在处理的文件，并通过管道将其传入合适的工具链（通常是 cut、awk 或 sed）。bash 的参数扩展（参见 5.18 节）在这里也非常方便。

```
ls *.odt | cut -d'=' -f1
```

好在本书中有实例可以告诉你提供正确参数值的详细方法；你只需要将其插入即可。确保先用 echo 测试，注意文件名中的空格或其他怪异字符，因为它们总是会给你惹麻烦。

> 别将脚本命名为 rename。我们在主流的 Linux 发行版中至少已经发现两个 rename 命令，肯定还有不少别的。Red Hat 的 util-linux 软件包中就有一个 rename *from_string to_string file_name* 工具。Debian 及其衍生版的 Perl 软件包中包含 Larry Wall[1] 用 Perl 编写的 rename 及相关 renameutils 实用工具包。Solaris、HP-UN 和一些 BSD 发行版中还有 rename 系统调用，尽管不太容易被最终用户访问到。可以在你的系统中尝试打开 rename 手册页，看看会发现什么。

[1]Perl 之父。——译者注

17.1.4 参考

- man mv
- man rename
- help for
- 5.18 节
- 9.2 节
- 17.12 节
- 19.13 节

17.2 在Linux中使用GUN Texinfo和info

17.2.1 问题

你在访问文档时遇到了麻烦，因为 Linux 中很多 GNU 工具的文档属于 Texinfo 文档，传统的手册页仅仅是存根（stub）而已，而默认的 info 程序对用户并不友好（而且你也不想再去学习一个用途单一的程序）。

17.2.2 解决方案

通过管道将 info 命令的输出传给分页程序（如 less），不过这样会丧失 info 的链接导航特性。

```
info bash | less
```

17.2.3 讨论

info 基本上是 Emacs info 阅读器的独立版本，因此，如果你是 Emacs 的粉丝，可能还有意义。但是，通过管道将 info 的输出传给 less 是使用现已熟悉的命令来查看文件的快捷方法。

Texinfo 背后的理念很好：从单一来源产生各种输出格式。这算不上新鲜，很多其他标记语言也是这么做的，甚至 5.2 节中就谈及过一种。既然如此，干吗不用 TeX 过滤 man 的输出？可能是因为手册页遵循的是一种标准化、结构化且经过时间考验的格式，而 Texinfo 采用的形式更为自由。

如果你不喜欢 info，还有别的 Texinfo 浏览器和转换工具，如 pinfo、info2www、tkman，甚至是 info2man（先转换成 POD，再转换成手册页格式）。

17.2.4 参考

- man info
- man man
- 维基百科（Textinfo）
- 5.2 节

17.3 批量解压ZIP文件

17.3.1 问题

你想一次性解压目录下的多个 ZIP 文件，但 unzip *.zip 没有效果。

17.3.2 解决方案

将模式放入单引号即可，不同于其他大多数 Unix 命令，unzip 自己处理文件通配模式。

```
unzip '*.zip'
```

也可以用循环来解压各个文件。

```
for x in /path/to/date*/name/*.zip; do unzip "$x"; done
```

或者：

```
for x in $(ls /path/to/date*/name/*.zip 2>/dev/null); do unzip $x;
done
```

17.3.3 讨论

和很多 Unix 命令（如 gzip 和 bzip2）不同，unzip 的最后一个参数并不是任意长度的文件列表。在处理命令 unzip *.zip 时，shell 会扩展其中的通配符，因此（假设你有若干文件，文件名分别为 zipfile1.zip 到 zipfile4.zip），unzip *.zip 会扩展为 unzip zipfile1.zip zipfile2.zip zipfile3.zip zipfile4.zip。该命令尝试从 zipfile1.zip 中提取 zipfile2.zip、zipfile3.zip、zipfile4.zip。除非 zipfile1.zip 中真的包含这些文件，否则肯定失败。

解决方案中提出的第一种方法用单引号来避免 shell 扩展通配符。但是，这仅适用于只有一个通配符的情况。后两种方法的解决思路是，

在 shell 完成通配符扩展或返回 ls 命令的结果后，明确地为每个 ZIP 文件运行 unzip 命令。

之所以使用 ls，是因为 bash（以及 sh）的默认行为是原封不动地返回未匹配的模式。这意味着，如果没有文件能够匹配通配符模式，你就要尝试解压名为 /path/to/date*/name/*.zip 的文件。ls 会简单地向 STDOUT 返回空，返回 STDERR 的错误信息则会被直接丢弃。你可以用 shopt -s nullglob 来设置 shell 选项，使未能匹配的文件名模式扩展成空串，而不再返回自身。

17.3.4 参考

- man unzip
- 15.13 节

17.4 用screen恢复断开的会话

17.4.1 问题

你在 SSH 上运行了一些耗时的进程，有可能还是通过广域网运行的，如果连接断开，会丢失大量的工作成果。或者是，你上班时启动了一项漫长的工作，但现在需要回家，晚些时候才能检查工作的完成情况。这可以使用 nohup 运行进程，但等到连接恢复或返回家中时，发现无法重新连接到该进程。

17.4.2 解决方案

安装并使用 GNU screen。

screen 的用法非常简单。输入 screen 或 screen -a。-a 可以启用 screen 的所有功能，代价则是牺牲一些重绘（进而是带宽）效率。不过说实话，在使用 -a 时，我们从来没注意到有什么差别。

输入上述命令后，看起来似乎什么都没发生，但现在你已经是在一个 screen 窗口中运行了。如果一切正常，echo $SHLVL 返回的数字

应该会大于 1（有关 $SHLVL，参见 16.2 节）。要想测试的话，输入 ls -la，然后用 kill“杀死”终端（不要干净退出，因为这样会一并退出 screen）。再次登录后输入 screen -r，重新连接到 screen。如果没有恢复到断开前的状态，尝试 screen -d -r。要是还不管用，输入 ps auwx | grep [s]creen，查看 screen 是否还在运行，并在 screen 的手册页中查阅排错信息，但一般应该可以正常运行的。如果在非 Linux 系统上使用 ps 命令时碰到问题，参见 17.21 节。

启动 screen 时加入以下内容会更容易确定随后该重新连接到哪个会话。

```
screen -aS "$(whoami).$(date$$'$$ $$+$$%Y-%m-
%d$$_$$%H:%M:%S%z$$'$$)
```

参见 16.22 节中的脚本 run_screen。

要想退出 screen 及会话，需要不停地输入 exit，直到所有会话全部结束。也可以输入 Ctrl-A Ctrl-\ 或者 Ctrl-A quit 来退出 screen 本身（假定你没有修改过 Ctrl-A 的默认辅助键）。

17.4.3 讨论

以下是官方站点对 screen 的描述。

> screen 是一个全屏窗口管理器，它在多个进程（通常是交互式 shell）之间多路复用物理终端。每个虚拟终端都具备 DEC VT100 终端的功能，此外还提供了 ANSI X3.64（ISO 6429）和 ISO 2022 标准中的一些控制功能（例如，插入/删除行以及多字符集支持）。每个虚拟终端都有自己的回滚历史记录缓冲区和允许用户在窗口之间移动文本的复制粘贴机制。

这意味着你可以在一个 SSH 终端中拥有多个会话（回想一下 i286/386 上的 DeskView）。它还允许你通过 SSH 连入计算机、启动进程、断开终端后返回家中，然后重新连接，继续先前的进程。同时能够实现可用于培训、故障排除或协作的多人共享会话（参见 17.5 节）。

01. 警告

screen 在 Linux 上通常是默认安装的，但其他系统很少如此。screen 的二进制可执行文件必须以 SUID root 运行，以便能够写入适合的 /usr/dev 伪终端（PTY）。这也可能是 screen 无法正常工作的原因（要想修复，需要以 root 身份执行命令 chmod u+s /usr/bin/screen）。

另外，screen 会干扰 zmodem 等内联传输协议。较新的 screen 版本有专门针对此的配置选项，可参见其手册页。

02. 配置

bash 命令行编辑默认采用的 Emacs 模式使用 Ctrl-A 跳转到行首，而这同样也是 screen 的命令模式或辅助键 [2]，如果需要大量用到 Ctrl-A（就像我们一样），可以考虑将下列内容添加到 ~/.screenrc 文件。

```
# ~/.screenrc设置样例
# 将默认的C-a更改为C-n（用C-n发送字面^N）
escape ^Nn

# 关闭烦人的铃声
vbell off

# 接收到hangup信号时自动断开
autodetach on

# 使每个窗口中的shell成为登录shell
shell -$SHELL
```

[2]某些键盘上能够看到辅助键，通常位于空格键旁边，当与其他按键组合使用时，可执行特殊功能。它源于 19 世纪 60 年代的 Lisp 计算机的键盘，并在 Sun 计算机上得以沿用（键帽表面绘制有菱形）。在现代计算机中，Windows 键（Microsoft Windows 系统）或 ⌘ 键（Apple 系统）可以实现辅助键的传统功能。——译者注

17.4.4 参考

- screen manpage

- 维基百科（GNU Screen ）
- 16.2 节
- 16.22 节
- 17.5 节
- 17.6 节
- 17.9 节
- 17.20 节

17.5 共享单个bash会话

17.5.1 问题

你需要共享单个 bash 会话，以作培训或故障排除之用，但是身边围观的人太多，难以开展工作。或者你需要帮助的人身在他方，需要通过网络共享会话。

17.5.2 解决方案

在多用户模式下使用 GNU screen。以下内容假定你没有按照 17.4 节中描述的方法更改默认的辅助键 Ctrl-A。如果已经更改，则可以使用新的辅助键（如 Ctrl-N）。

作为演示方，执行下列操作。

- 输入 `screen -S` *`session_name`*（不能有空格），例如，`screen -S training`。
- 输入 Ctrl-A `addacl` *`usernames`*，列出可以访问显示器的账户（以逗号分隔，不能有空格！），例如，Ctrl-A `addacl alice,bob,carol`。注意，这允许完全的读/写访问权限。
- 根据需要，使用 Ctrl-A `chacl usernames permbits list` 命令来调整权限。
- 用 Ctrl-A `multiuser on` 启用多用户模式。

作为观众，执行下列操作。

- 用 `screen -x user/name` 连接共享屏幕；例如，`screen -x host/training`。
- 按下 Ctrl-A k 来关闭窗口，并结束会话。

17.5.3 讨论

具体细节参见 17.4 节。

对于多用户模式，/tmp/screens 必须存在且所有用户应该皆可读取及执行。

Red Hat（RHEL3）的 screen 从版本 3.9.15-8 至 4.0.1-1 都有问题，如果要用到多用户模式，不要使用这些版本。版本 4.0.2-5 或后续版本应该没问题。一旦开始使用新版本的 screen，$HOME/.screen 中的现有 screen 套接字就失效了，再也无法使用。注销所有会话，使用新版本在 /tmp/screens/S-$USER 中创建新的套接字，然后删除 $HOME/.screen 目录。

17.5.4 参考

- man screen
- 9.11 节
- 16.22 节
- 17.4 节
- 17.6 节

17.6 记录整个会话或批量作业

17.6.1 问题

你需要捕获整个会话或冗长的批量作业产生的所有输出。

17.6.2 解决方案

解决这个问题的方法有很多种，取决于你的需求和环境。

一种最简单的方法是在终端中开启向内存或磁盘记录日志的功能。问题是你所使用的终端程序可能不允许这么做，而且连接断开时，日志会丢失。

另一种简单的解决方案是修改作业，使其自身能够记录日志，或者将所有输出重定向到 tee 或文件。例如，以下做法也许可行。

```
long_noisy_job >& log_file
long_noisy_job 2>&1 | tee log_file

( long_noisy_job ) >& log_file
( long_noisy_job ) 2>&1 | tee log_file
```

这里的问题是，你可能无法修改该作业，或者作业本身要做的事情超出了这些方法的解决能力（例如，如果需要用户输入，它会在提示符出现之前一直干等着）。之所以会这样，是因为 STDOUT 属于缓冲型，当还要有更多数据到来时，提示符会在缓冲区中等待显示，但由于程序正等待输入，此时根本不会有数据进入。

第三种解决方案是使用有趣的专有工具 script，你的系统可能已经自带了。运行 script，它会将所有的输出全部记录在指定的日志文件（称为 typescript）中，如果你想记录整个会话，也完全没有问题，启动 script，然后运行作业即可。但如果只是想捕获部分会话，是没有办法让代码启动 script，运行并记录一段时间，然后再停止 script 的。script 无法脚本化，因为一旦运行，你就被置于子 shell 中了（也就是说，无法实现 script *file_to_log_to some_command_to_run* 这样的操作）。

我们给出的最终解决方案是使用终端复用器 screen。借助 screen，你可以在脚本内启用或关闭整个会话的日志记录功能。运行 screen 后，能够在脚本中执行下列操作。

```
# 设置日志文件并启用日志记录功能
screen -X logfile /path/to/logfile && screen -X log on

# 将你的命令放在这里

# 关闭日志记录功能
screen -X logfile 1      # 设置缓冲区冲刷间隔为1秒
```

```
sleep 3                    # 等待，避免文件被截断……
screen -X log off
```

17.6.3 讨论

我们建议你依次尝试这些解决方案，哪个能先满足你的需求，就用哪个。script 应该不会有什么问题，除非你有非常特殊的需求。但为了以防万一，了解 screen 的选项会带来不少便利。

17.6.4 参考

- man script
- man screen
- 17.5 节

17.7 注销时清除屏幕

17.7.1 问题

你使用或管理的一些系统在注销时不会清除屏幕，你不希望自己的工作内容露出任何蛛丝马迹，以免造成信息泄露。

17.7.2 解决方案

将 clear 命令放入 ~/.bash_logout（参见例 17-1，摘自 16.22 节）。

例 17-1 ch16/bash_logout

```
# 实例文件：bash_logout

# bash_logout：在注销时执行

# 在注销时清除屏幕，避免信息泄露（如果尚未在别处设置退出陷阱）
[ -n "$PS1" ] && clear
```

或者设置陷阱，在 shell 终止时执行 clear。

```
# 设置陷阱，在注销时清除屏幕，避免信息泄露
#（如果尚未在~/.bash_logout中设置）
trap ' [ -n "$PS1" ] && clear ' 0
```

注意，如果使用了远程连接且客户端配备了回滚缓冲区，那么你的工作内容可能仍旧会显示在屏幕上。clear 对 shell 的命令历史记录也没有效果。

17.7.3 讨论

设置陷阱来清除屏幕可能有点杀鸡用牛刀了，却能够涵盖 ~/.bash_logout 未被执行的错误场景。如果举棋不定，可以两种方法皆用，不过真要是打算这样的话，不妨再去了解一下 TEMPEST 和法拉第笼（Faraday cages）[3]。

[3]两者均是用于解决电磁信息泄露的技术措施。——译者注

如果不测试 shell 是否为交互式，某些情况下会产生下列错误。

```
# 例如，tput引发的错误
No value for $TERM and no -T specified

# 例如，clear引发的错误
TERM environment variable not set.
```

17.7.4 参考

- 维基百科（Tempest）
- 维基百科（Faraday cage）
- 16.22 节

17.8 获取用于数据恢复的文件元数据

17.8.1 问题

你想要创建文件清单以及相关文件的详细信息（例如，核实备份、重新创建目录等），以作归档之用；或是要执行大批量的 chmod -R 操作，需要准备对应的回撤计划，抑或要将 /etc/* 保存在不保留权限或所有权的版本控制系统中。

17.8.2 解决方案

配合一些 printf 格式使用 GNU find，如例 17-2 所示。

例 17-2 ch17/archive_meta-data

```
#!/usr/bin/env bash
# 实例文件：archive_meta-data

printf "%b" "Mode\tUser\tGroup\tBytes\tModified\tFileSpec\n" >
archive_file
find / \( -path /proc -o -path /mnt -o -path /tmp -o -path
/var/tmp \
  -o -path /var/cache -o -path /var/spool \) -prune \
  -o -type d -printf 'd%m\t%u\t%g\t%s\t%t\t%p/\n' \
  -o -type l -printf 'l%m\t%u\t%g\t%s\t%t\t%p -> %l\n' \
  -o         -printf '%m\t%u\t%g\t%s\t%t\t%p\n' >> archive_file
```

注意，-printf 表达式可用于 GNU 版本的 find 命令中。

17.8.3 讨论

(-path /proc -o -path...) -prune 用于去除各种无关的目录。-type d 表示查找类型为目录。printf 格式以 d 作为前缀，然后是八进制模式的文件权限、用户名、用户组等。-type l 表示查找符号链接并显示每个链接的指向。有了这些，再加上其他脚本，你就可以从较高层面确定是否有改动，或是重建损坏的所有权或权限。注意，这并不能取代 Tripwire、AIDE、Samhain 这种主打安全的程序。

17.8.4 参考

- man find
- 第 9 章

17.9 为多个文件创建索引

17.9.1 问题

你想为多个文件创建索引。

17.9.2 解决方案

使用 find 命令，配合 head、grep 或其他能够解析文件注释或摘要信息的命令。

例如，如果你所有的 shell 脚本的第二行都遵循“名称—描述”这种格式，那就可以按照以下方法创建一个不错的索引。

```
for i in $(grep -El '#![[:space:]]?/bin/sh' *); do head -2 $i |
tail -1; done
```

17.9.3 讨论

如前所述，这种技术取决于每个文件都具有能够解析出来的某种摘要信息（如注释）。然后，我们寻找一种识别文件类型（本例为 shell 脚本）的方法，并获取每个文件的第二行。

如果文件不具有易于解析的摘要信息，你可以按照以下方式尝试手动处理输出，以创建索引。

```
for dir in $(find . -type d); do head -15 $dir/*; done
```

小心二进制文件！

17.9.4 参考

- man find
- man grep
- man head

- man tail

17.10 使用diff和patch

17.10.1 问题

你总是记不起来如何使用 diff 创建之后可能要通过 patch 应用的补丁。

17.10.2 解决方案

如果要为单个文件生成一个简单的补丁，使用：

```
$ diff -u original_file modified_file > your_patch
$
```

如果要为类似目录结构中的多个文件创建补丁，使用：

```
$ cp -pR original_dirs/ modified_dirs/
$

# 在此做出改动

$ diff -Nru original_dirs/ modified_dirs/ >
your_comprehensive_patch
$
```

谨慎起见，使用 -a 选项强制 diff 将所有文件视为 ASCII 编码，并将语言和时区设置为通用默认值，如下所示。

```
$ LC_ALL=C TZ=UTC diff -aNru original_dirs/ modified_dirs/ \
  > > your_comprehensive_patch
$

$ LC_ALL=C TZ=UTC diff -aNru original_dirs/ modified_dirs/
diff -aNru original_dirs/changed_file modified_dirs/changed_file
--- original_dirs/changed_file 2006-11-23 01:04:07.000000000 +0000
+++ modified_dirs/changed_file 2006-11-23 01:04:35.000000000 +0000
@@ -1,2 +1,2 @@
 This file is common to both dirs.
```

```
-But it changes from one to the other.
+But it changes from 1 to the other.
diff -aNru original_dirs/only_in_mods modified_dirs/only_in_mods
--- original_dirs/only_in_mods 1970-01-01 00:00:00.000000000 +0000
+++ modified_dirs/only_in_mods 2006-11-23 01:05:58.000000000 +0000
@@ -0,0 +1,2 @@
+While this file is only in the modified dirs.
+It also has two lines, this is the last.
diff -aNru original_dirs/only_in_orig modified_dirs/only_in_orig
--- original_dirs/only_in_orig 2006-11-23 01:05:18.000000000 +0000
+++ modified_dirs/only_in_orig 1970-01-01 00:00:00.000000000 +0000
@@ -1,2 +0,0 @@
-This file is only in the original dirs.
-It has two lines, this is the last.
```

要想应用补丁文件，使用 cd 切换到文件目录或目录树的父目录，然后使用 patch 命令。

```
$ cd /path/to/files
$ patch -Np1 < your_patch
```

patch 的 -N 选项会忽略陈旧或已经应用过的补丁。-p *number* 会删除前导目录的 *number* 层，以此允许补丁创建人员与补丁应用人员之间存在目录结构上的差异。通常使用 -p1 即可；如果不行，尝试 -p0，然后再尝试 -p2，以此类推。要么奏效，要么出错并询问你要做什么，这种情况下，取消操作并尝试别的方法，除非你确实知道自己在做什么。

> patch 命令支持 --dry-run 选项，借用手册页中的描述，该选项“输出应用补丁后的结果，但并不会真的改动任何文件”，在实际执行 patch 命令前，值得一用。

17.10.3 讨论

diff 可以产生各种形式的输出，其中一些形式要更为实用。-u 选项产生的合并输出（unified output）通常被认为是最好的，因为与 patch 一起使用时，该形式既易于阅读又颇为稳健。它提供了变更附近的 3 行上下文，这不仅方便了阅读人员定位，而且即便要打补丁的文件与创建补丁的文件不同，patch 命令也可以正常工作。只要上下文是完整的，patch 通常就能弄清楚怎么做。使用 -c 选项的上下

文输出类似于 -u 选项的输出，但是更加冗余，也不怎么易于阅读。-e 选项生成的 ed 脚本适合于古老 ed 编辑器使用。最后，diff 的默认输出与 ed 输出类似，但上下文更容易理解一点。

```
# 合并格式（首选）
$ diff -u original_file modified_file
--- original_file       2006-11-22 19:29:07.000000000 -0500
+++ modified_file       2006-11-22 19:29:47.000000000 -0500
@@ -1,9 +1,9 @@
-This is original_file, and this line is different.
+This is modified_file, and this line is different.
 This line is the same.
 So is this one.
 And this one.
 Ditto.
-But this one is different.
+But this 1 is different.
 However, not this line.
 And this is the last same, same, same.

# 上下文格式
$ diff -c original_file modified_file
*** original_file       Wed Nov 22 19:29:07 2006
--- modified_file       Wed Nov 22 19:29:47 2006
***************
*** 1,9 ****
! This is original_file, and this line is different.
  This line is the same.
  So is this one.
  And this one.
  Ditto.
! But this one is different.
  However, not this line.
  And this is the last same, same, same.

--- 1,9 ---
! This is modified_file, and this line is different.
  This line is the same.
  So is this one.
  And this one.
  Ditto.
! But this 1 is different.
  However,


# ed格式
$ diff -e original_file modified_file
6c
```

```
But this 1 is different.
.
1c
This is modified_file, and this line is different.
.


# 正常格式
$ diff original_file modified_file
1c1
< This is original_file, and this line is different.
---
> This is modified_file, and this line is different.
6c6
< But this one is different.
---
> But this 1 is different.
```

diff 的 -r 和 -N 参数虽然简单，但功能强大。-r 依旧表示对目录结构执行递归操作，而 -N 会使 diff 认为在一个目录结构中找到的文件也会以空文件的形式存在于另一个目录结构中。从理论上讲，这可以起到按需创建或删除文件的效果；但在实践中，并不是所有系统（尤其 Solaris）都支持 -N，而且最终可能会留下零字节文件。有些版本的 patch 默认使用 -b 选项，这会遗留下大量的 .orig 文件，有些版本（尤其 Linux）没有其他版本（尤其 BSD）那么多的输出信息。不少 diff 版本（非 Solaris）也支持 -p 选项，该选项会尝试显示补丁所影响到的 C 函数。

克制住执行 diff -u prog.c.orig prog.c 这类操作的冲动。patch 也许还会创建 .orig 文件，因此可能会引起各种混乱。也别尝试 diff -u prog/prog.c new/prog/prog.c 这类操作，因为 patch 会对路径中不等量的目录数感到非常困惑。

wdiff

值得一提的还有另一个鲜为人知的工具 wdiff。wdiff 可以比较文件以检测单词的变化，这里所说的“单词”是指两侧均为空白字符的字符串。它能够处理不同的换行符并尝试使用 termcap 字符串来产生更具可读性的输出。如果逐行比较还不够细致，这个工具就能派上用场了，它类似于 Emacs 的**单词差异**（word diff）特性以及 git diff --word-diff 命令。注意，默认

情况下，很少有系统自带 wdiff。你可以从自由软件目录或系统的软件包管理器获取。以下是 wdiff 的输出示例。

```
$ wdiff original_file modified_file
This is [-original_file,-] {+modified_file,+} and this line is
different.
This line is the same.
So is this one.
And this one.
Ditto.
But this [-one-] {+1+} is different.
However, not this line.
And this is the last same, same, same.
$
```

17.10.4 参考

- man diff
- man patch
- man cmp
- xxdiff，一款不错的 GUI diff（还包括其他功能）工具

17.11 统计文件间存在多少差异

17.11.1 问题

你想知道两个文件之间存在多少处差异。

17.11.2 解决方案

统计 diff 输出中有多少个 hunk（改动过数据的区域）。

```
$ diff -C0 original_file modified_file | grep -c "^\*\*\*\*\*"
2

$ diff -C0 original_file modified_file
*** original_file       Fri Nov 24 12:48:35 2006
--- modified_file       Fri Nov 24 12:48:43 2006
***************
```

```
*** 1 ****
! This is original_file, and this line is different.
--- 1 ---
! This is modified_file, and this line is different.
***************
*** 6 ****
! But this one is different.
--- 6 ---
! But this 1 is different.
```

如果只是想知道文件是否有所差异，并不关心有多少处差异，可以使用 cmp。该命令只要找到第一处差异，就立刻退出，在面对大文件时，这种做法能够节省时间。和 diff 一样，如果文件一模一样，cmp 会一声不吭；如果存在差异，则报告第一处差异的位置。

```
$ cmp original_file modified_file
original_file modified_file differ: char 9, line 1
```

17.11.3 讨论

hunk 其实是技术术语，尽管某些地方也称其为 chunk。注意，从理论上讲，相同的文件在不同的机器或 diff 版本上得到的结果会略有不同，因为 hunk 的数量是由 diff 所使用的算法决定的。使用不同的 diff 输出格式时，得到的答案肯定也不一样，随后的示例会演示这一点。

我们发现零上下文（zero-context）的 diff 最方便用于此目的，使用 -C0 代替 -c 可以为 grep 产生较少的搜索行。diff 的合并输出倾向于将比预期更多的变更合并成一个 hunk，从而生成更少的差异报告。

```
$ diff -u original_file modified_file | grep -c "^@@"
1

$ diff -u original_file modified_file
--- original_file       2006-11-24 12:48:35.000000000 -0500
+++ modified_file       2006-11-24 12:48:43.000000000 -0500
@@ -1,8 +1,8 @@
-This is original_file, and this line is different.
+This is modified_file, and this line is different.
 This line is the same.
 So is this one.
```

```
 And this one.
 Ditto.
-But this one is different.
+But this 1 is different.
 However, not this line.
 And this is the last same, same, same.
```

使用正常的或 ed 风格的 diff 输出也没问题，只不过 grep 的搜索模式会更复杂些。虽然这个示例没有展现出来，但多行变更在普通 grep 输出中可能看起来类似于 2,3c2,3，因此，相较于使用 -C0，需要用到字符类和更多的输入。

```
$ diff -e original_file modified_file | egrep -c '^[[:digit:],]+
[[:alpha:]]+'
2

$ diff original_file modified_file | egrep -c '^[[:digit:],]+
[[:alpha:]]+'
2

$ diff original_file modified_file
1c1
< This is original_file, and this line is different.
---
> This is modified_file, and this line is different.
6c6
< But this one is different.
---
> But this 1 is different.
```

17.11.4 参考

- man diff
- man cmp
- man grep
- 维基百科（Diff）

17.12 删除或重命名名称中包含特殊字符的文件

17.12.1 问题

你需要删除或重命名的文件在创建时使用了特殊字符，这会造成 rm 或 mv 行为异常。典型的例子就是以连字符起始的文件（如 -f 或 --help），这会使得要执行的命令将这种文件名视为选项。

17.12.2 解决方案

如果文件名以连字符起始，既可以用 -- 表示命令选项到此结束，也可以使用完整路径（/tmp/-f）或相对路径（./-f）。要是文件名中还包含其他会被 shell 解释的特殊字符，如空格或星号，则使用 shell 的引用机制。如果使用了文件名补全（默认是按 Tab 键），则 shell 会自动帮你引用特殊字符。另外也可以将可能造成问题的文件名放进单引号。

```
$ ls
--help                          this is a *crazy* file name!
$ mv --help help
mv: unknown option -- -
usage: mv [-fiv] source target
       mv [-fiv] source ... directory

$ mv -- --help my_help

$ mv this\ is\ a\ \*crazy\*\ file\ name\! this_is_a_better_name
$ ls
my_help                         this_is_a_better_name
```

17.12.3 讨论

要想弄清楚 shell 扩展后执行的究竟是什么命令，可以将 echo 放在命令之前。

```
$ rm *
rm: unknown option -- -
usage: rm [-f|-i] [-dPRrvW] file ...

$ echo rm *
rm --help this is a *crazy* file name!
```

你也可以在目录中创建一个名为 -i 的文件，以免 rm * 一声不吭地就删除所有的文件。

```
$ mkdir del-test ; cd $_

$ > -i

$ touch important_file

$ ll
total 0
-rw-r--r-- 1 jp jp 0 Jun 12 22:28 -i
-rw-r--r-- 1 jp jp 0 Jun 12 22:28 important_file

$ rm *
rm: remove regular empty file 'important_file'? n
```

17.12.4 参考

- GNU Core Utilities FAQ 中的问题 11
- Unix FAQs 的 2.1 节和 2.2 节
- 1.8 节

17.13 将数据追加到文件开头

17.13.1 问题

你想将数据追加到现有文件的开头，例如，在排序后添加标题。

17.13.2 解决方案

在子 shell 中使用 cat。

```
temp_file="temp.$RANDOM$RANDOM$$"
(echo 'static header line1'; cat data_file) > $temp_file \
  && cat $temp_file > data_file
rm $temp_file
unset temp_file
```

你也可以使用流编辑器 sed。追加静态文本时要注意，反斜线转义序列在 GNU sed 中会被扩展，但在其他有些版本中则不会。另外，在有些 shell 中，行尾的反斜线可能需要写成两个。

```
# 适用于任意版本的sed，如Solaris 10 /usr/bin/sed
$ sed -e '1i\
> static header line1
> ' data_file
static header line1
1 foo
2 bar
3 baz

$ sed -e '1i\
> static header line1\
> static header line2
> ' data_file
static header line1
static header line2
1 foo
2 bar
3 baz


# GNU sed
$ sed -e '1istatic header line1\nstatic header line2' data_file
static header line1
static header line2
1 foo
2 bar
3 baz
```

追加到现有文件内容之前。

```
$ sed -e '$r data_file' header_file
Header Line1
Header Line2
1 foo
2 bar
3 baz
```

17.13.3 讨论

这看起来是一个非爱即厌的解决方案。人们要么喜欢用 cat，要么喜欢用 sed，但不会两者皆爱。cat 版本可能更快更简单；sed 版本

显然更加灵活。

你也可以将 sed 脚本保存在文件中，这样就不必写成命令行了。当然，惯常的做法是将脚本输出重定向成一个新文件，就像 sed -e '$r *data*' *header* > *new_file* 一样，但要注意，这会改变文件的 i 节点（inode），也许还会改变其他的文件属性，如权限或所有权。要想保留除 i 节点之外的其他内容不变，可以使用 -i 选项进行就地编辑（in-place editing）（如果你的 sed 版本支持）。但不要将 -i 选项与之前展示的那种在文件开头追加标题文件的方法一起使用，否则会改变标题文件！另外要注意，Perl 也有类似的 -i 选项，同样会写入新文件，但对这个例子来说，Perl 本身的工作方式与 sed 完全不同。

```
# 显示i节点
$ ls -i data_file
509951 data_file

$ sed -i -e '1istatic header line1\nstatic header line2' data_file

$ cat data_file
static header line1
static header line2
1 foo
2 bar
3 baz
# 验证i节点已经改变
$ ls -i data_file
509954 data_file
```

要想保持一切不变（或者你使用的 sed 没有 -i 选项，又或者你想使用之前介绍的文件追加方法）：

```
# 显示i节点
$ ls -i data_file
509951 data_file

# $RANDOM仅适用于bash；在其他系统中可以使用mktemp
$ temp_file=$RANDOM$RANDOM

$ sed -e '$r data_file' header_file > $temp_file

# 仅当源文件存在且不为空时才使用cat!
$ [ -s "$temp_file" ] && cat $temp_file > data
```

```
$ unset temp_file

$ cat data_file
Header Line1
Header Line2
1 foo
2 bar
3 baz
# 核实i节点并未改变
$ ls -i data_file
509951 data
```

将标题文件追加到数据文件开头是相当违反直觉的操作。如果尝试在第一行将 header_file 文件读入 data_file 文件，则会得到下列结果。

```
$ sed -e '1r header_file' data_file
1 foo
Header Line1
Header Line2
2 bar
3 baz
```

因此，我们只需要简单地将数据追加到标题文件尾部，然后将输出写入另一个文件就行了。还是那句话，别用 sed -i，否则会改变标题文件。

另一种方法是用 cat 从 STDIN 中读取 here-document 或 here-string。注意，here-string 仅在 bash 2.05b 或更高版本中可用，而且不会执行反斜线转义序列扩展，但同时也避开了所有 sed 版本存在的问题。

```
# 使用here-document
$ cat - data_file <<EoH
> Header line1
> Header line2
> EoH
Header line1
Header line2
1 foo
2 bar
3 baz
```

```
# 使用bash 2.05b或更高版本中的here-string，其中的反斜线转义序列不会被扩展
$ cat - data_file <<<'Header Line1'
Header Line1
1 foo
2 bar
3 baz
```

17.13.4 参考

- man cat
- man sed
- THE SED FAQ
- 14.11 节
- 17.14 节

17.14 就地编辑文件

17.14.1 问题

你想在不影响 i 节点或权限的情况下编辑现有文件。

17.14.2 解决方案

这比听起来要棘手，因为你平时使用的很多工具（如 sed）会写入并生成新文件（因而改变了 i 节点），尽管它们会尽力保留原文件的其他属性。

显而易见的解决方案是直接编辑文件并进行更新。但是，我们得承认，这种做法在编写脚本的情况下可能用途有限。是这样吗？

在 17.13 节中，你已经看到了 sed 会以这样或那样的方式写入并生成一个全新的文件。但 sed 的前辈可不是这么做的。这位前辈叫 ed，名字听起来不怎么样，而且它与另一位盛名在外的后辈 vi 一样，简直是无处不在。另外值得留意的是，ed 还是可脚本化的。因

此，这里又要搬出前面那个“追加标题”的例子了，不过这次改用 ed。

```
# 显示i节点
$ ls -i data_file
306189 data_file

# 使用printf "%b"来避免出现'echo -e'问题
$ printf "%b" '1i\nHeader Line1\nHeader Line2\n.\nw\nq\n' | ed -s
data_file
1 foo

$ cat data_file
Header Line1
Header Line2
1 foo
2 bar
3 baz

# 核实i节点并未改变
$ ls -i data_file
306189 data_file
```

17.14.3 讨论

当然了，你也可以像 sed 那样将 ed 脚本保存在文件中。这里看看该文件的内容可能有助于解释 ed 脚本的工作机制。

```
$ cat ed_script
1i
Header Line1
Header Line2
.
w
q

$ ed -s data_file < ed_script
1 foo

$ cat data_file
Header Line1
Header Line2
1 foo
2 bar
3 baz
```

ed 脚本中的 1i 表示跳转到第一行，然后进入插入模式，接下来的两行都是普通的字面文字。仅包含单个 . 的行表示退出插入模式，w 表示写入文件，q 表示退出。-s 表示禁止输出诊断信息，这一点在脚本中尤其有用。

ed 的一个缺点是它现存的文档不多。它自 Unix 诞生以来就已经存在，虽然在我们接触过的每个系统中都能找到其身影，如今用它的人却不多了。由于 vi（通过 ex）和 sed（至少在精神层面）都源自 ed，因此你应该能够琢磨出该怎么做。注意，在许多系统上，ex 是指向 vi 或某种变体的符号链接，而 ed 就只是 ed。

使用 sed 或其他工具将修改完的文件内容写入新的文件，然后通过 cat 再写回原文件，也能实现相同的效果。但效率显然不高。况且要做到万无一失也没有说起来那么容易，如果出于这样或那样的原因未能修改成功，结果会导致原文件被清空（参见 17.13 节中的例子）。

17.14.4 参考

- man ed
- man ex
- ls -l which ex
- THE SED FAQ
- 17.13 节

17.15 将sudo应用于一组命令

17.15.1 问题

你当前的身份是普通用户，需要一次性对多个命令使用 sudo，或者需要对除 sudo 之外的多个命令使用重定向。

17.15.2 解决方案

使用 sudo 启动一个子 shell，你可以在其中分组命令并使用管道和重定向。

```
sudo bash -c 'command1 && command2 || command3'
```

这要求你能以 root 用户身份启动子 shell。如果做不到，可以请系统管理员编写一个快速脚本（quick script），并将其加入你的 sudo 权限规范。

17.15.3 讨论

如果尝试 sudo *command1* && *command2* || *command3* 这样的命令，你会发现 *command2* 和 *command3* 是以你的身份运行的，而非 root 用户。这是因为 sudo 只能影响到其后的第一个命令，剩下的命令还是由你的 shell 来执行。也就是说，sudo 执行的命令截止到 && 为止，shell 将其视为命令之间的分隔符。

要注意 bash 的 -c 选项，该选项使得 bash 仅执行指定的命令，然后就退出。没有它的话，运行的就是一个新的交互式 root shell，这可能并不是你想要的结果。有了 -c，运行的则是非交互式 root shell，但你得有 sudo 权限才行。macOS 和某些 Linux 发行版（如 Ubuntu）实际上禁用了 root 用户，以此鼓励以普通用户身份登录，并在需要进行管理时使用 sudo（Mac 在这方面隐藏得更好）。如果使用的是这种操作系统，或者已经设置好了自己的 sudo，应该不会有什么问题。但是，如果你所处的是受限环境，本实例可能帮不上什么忙。

要想知道你能否使用 sudo，它能做什么，不能做什么，可以使用 sudo -l。几乎所有其他涉及 sudo 的操作都可能会向管理员触发一条有关你的安全消息。你可以尝试使用 sudo sudo -V | less（普通用户身份）或者 sudo -V | less（root 用户身份），获取更多有关 sudo 在系统中的编译和配置信息。

su 和sudo

通常以普通用户身份运行，仅在绝对有必要时才使用 root 权限，这始终是一种最佳实践。虽然 su 命令挺方便，但很多人认为 sudo 更胜一筹，原因如下。

- 尽管得花费更多工夫才能让sudo 正常工作（进行限制，而不是一刀切的 "ALL=(ALL)ALL"），而且使用起来略有不便，但这确实能促进更安全的工作实践。
- 你会忘记自己已经通过 su 变成了 root 用户，并不小心做出一些不当操作。
- 必须一直输入 sudo，这会提醒你更谨慎一点。
- sudo 允许在不共享 root 密码的情况下将个别命令授权给其他用户。
- su 能做的事，sudo 都能做，反过来可就未必了。

这两个命令都可以加入日志记录，而且一些技巧可以使它们非常相像。但是，二者之间仍然存在一些显著差异。其中最重要的一点是，使用 sudo 时，你需要输入自己的密码，确认身份之后才允许执行命令。因此，即便多个用户要用到 root 权限，也不用共享 root 密码。这就带来了另一处区别：sudo 可以非常明确地指定某个用户可以执行和不能执行的命令。这种限制可能很棘手，因为许多应用程序允许用户启动额外的 shell，以执行其他操作（因此，如果你能够通过 sudo 进入 vi，就可以启动一个不受限制的 root shell）。尽管如此，谨慎使用的 sudo 仍不失为一件利器。

17.15.4 参考

- man su
- man sudo
- man sudoers
- man visudo
- sudo
- 14.15 节
- 14.18 节
- 14.19 节
- 14.20 节

17.16 查找仅出现在一个文件中的行

17.16.1 问题

你手边有两个数据文件，现在需要进行对比，找出仅在其中一个文件中出现的行。

17.16.2 解决方案

对文件排序，如有必要，使用 cut 或 awk 分离出感兴趣的数据，然后根据需要使用 comm、diff、grep 或 uniq。

comm 专用于此类问题。

```
$ cat left
record_01
record_02.left only
record_03
record_05.differ
record_06
record_07
record_08
record_09
record_10

$ cat right
record_01
record_02
record_04
record_05
record_06.differ
record_07
record_08
record_09.right only
record_10

# 仅显示出现在文件left中的行
$ comm -23 left right
record_02.left only
record_03
record_05.differ
record_06
record_09

# 仅显示出现在文件right中的行
$ comm -13 left right
record_02
```

```
record_04
record_05
record_06.differ
record_09.right only

# 仅显示同时出现在两个文件中的行
$ comm -12 left right
record_01
record_07
record_08
record_10
```

diff 能够快速地展示出两个文件之间的所有差异，但其输出不是特别理想，你可能也不需要了解全部的差异。GNU diff 的 -y 和 -W 选项可以提高可读性，但你也得习惯常规输出。

```
$ diff -y -W 60 left right
record_01                       record_01
record_02.left only           |  record_02
record_03                     |  record_04
record_05.differ              |  record_05
record_06                     |  record_06.differ
record_07                       record_07
record_08                       record_08
record_09                     |  record_09.right only
record_10                       record_10

$ diff -y -W 60 --suppress-common-lines left right
record_02.left only           |  record_02
record_03                     |  record_04
record_05.differ              |  record_05
record_06                     |  record_06.differ
record_09                     |  record_09.right only

$ diff left right
2,5c2,5
< record_02.left only
< record_03
< record_05.differ
< record_06
---
> record_02
> record_04
> record_05
> record_06.differ
8c8
< record_09
```

```
---
> record_09.right only
```

有些系统（如 Solaris）可能会用 sdiff 代替 diff -y，或者有单独的命令（如 bdiff）来处理超大文件。

grep 能够显示出仅存在于其中一个文件中的那些行，你可以根据需要找出是哪个文件。但因为执行的是正则表达式匹配，所以无法处理行内的差异，除非你编辑作为参数的模式文件[4]，而且随着文件越来越大，grep 的速度也会越来越慢。

[4]在命令 grep -vf right left 中，-f 选项将参数 right 视为模式文件，从中提取模式（一行一个模式）。——译者注

以下示例显示了存在于文件 left，但不存在于文件 right 中的所有行。

```
$ grep -vf right left
record_03
record_06
record_09
```

注意，其中只有“record_03”是真正不存在于文件 right 中的，其他两行只是不一样而已。如果需要区分这种差异，得使用 diff。如果打算无视，可以使用 cut 或 awk 将用得着的部分分离到临时文件中。

uniq -u 能够显示出文件中独有的行，但不会告诉你这些行来自哪个文件（如果确实想知道，可以在先前给出的解决方案中选择一种）。uniq -d 能够显示出同时在两个文件中出现的行。

```
$ sort right left | uniq -u
record_02
record_02.left only
record_03
record_04
record_05
record_05.differ
record_06
record_06.differ
record_09
```

```
record_09.right only

$ sort right left | uniq -d
record_01
record_07
record_08
record_10
```

17.16.3 讨论

如果可用，comm 是你的最佳选择，犯不着使用 diff，大材小用了。

如果不能改动原文件，可能需要借助 sort/cut/awk 创建并处理临时文件。

17.16.4 参考

- man cmp
- man diff
- man grep
- man uniq

17.17 保留最近的*N*个对象

17.17.1 问题

你需要保留最近的 *N* 个日志文件或备份目录，同时将剩余的清理掉，不管数量有多少。

17.17.2 解决方案

创建一个有序的对象列表，将其作为参数传给函数，移动 *N* 个参数，并将剩余的返回，如例 17-3 所示。

例 17-3 ch17/func_shift_by

```
# 实例文件：func_shift_by

# 从列表头部移走指定数量的列表项，以便对余下的内容执行操作
# 调用方式：shift_by <要保留的数量> <ls或其他命令>
# 返回：返回列表中剩余的内容
#
# 例如，保留最近的10个对象，列出其余的那些
#
# 将要保留的对象放在列表头部（或者前面），这一点至关重要，
# 因为该函数要做的就是从列表头部移走（弹出）指定数量的列表项
#
# 在进行删除操作之前，记得先使用echo测试！
#
# 例如：
#       rm -rf $(shift_by $MAX_BUILD_DIRS_TO_KEEP $(ls -rd
backup.2006*))
#
function shift_by {

# 如果$1为0或者大于$#，则保持位置参数不变。
# 在这种情况下，显然是有错误！
if (( $1 == 0 || $1 > ( $# - 1 ) )); then
    echo ''
else
    # 从列表中移走指定数量（加1）的对象
    shift $(( $1 + 1 ))
    # 返回剩余的对象
    echo "$*"
fi
}
```

如果移动位置参数的次数为 0 或超过了位置参数的总数（$#），shift 则不会执行任何操作。如果用 shift 处理参数列表，然后删除剩余的参数，则会清除所有的文件。一定要检查 shift 的参数，确保其不为 0 且不大于位置参数的总数。我们编写的 shift_by 函数就做到了这一点。

例如：

```
$ source shift_by

$ touch {1..9}

$ ls ?
1 2 3 4 5 6 7 8 9
```

```
$ shift_by 3 $(ls ?)
4 5 6 7 8 9

$ shift_by 5 $(ls ?)
6 7 8 9

$ shift_by 5 $(ls -r ?)
4 3 2 1

$ shift_by 7 $(ls ?)
8 9

$ shift_by 9 $(ls ?)

# 只保留前5个参数
$ echo "rm -rf $(shift_by 5 $(ls ?))"
rm -rf 6 7 8 9

# 在生产环境中可得先测试！见下一节
$ rm -rf $(shift_by 5 $(ls ?))

$ ls ?
1 2 3 4 5
```

17.17.3 讨论

一定要充分测试返回的参数以及要对其执行的操作。例如，如果打算删除旧数据，在实际操作前，先用 `echo` 测试一下要执行的命令。另外还要确定值不为空，否则就会错误地执行 `rm -rf`。可别执行 `rm -rf / $variable` 这种命令，如果 `$variable` 为空，从根目录开始删除，要是你还是 root 用户，那更是火上浇油！

在生产环境中，应该按以下方式使用解决方案中给出的函数。

```
$files_to_nuke=$(shift_by 5 $(ls ?))
[ -n $files_to_nuke ] && rm -rf "$files_to_nuke"
```

这个实例利用了一个事实：函数内部的 `shift` 命令能够控制该函数的参数，这就使得从参数列表中移走一部分参数变得易如反掌（否则，那就只能借助于花哨的子串操作或者 `for` 循环）。我们必须移走 $n+1$ 个参数，因为第一个参数（`$1`）其实是指定要保留的对象数

量，接下来的 $2..N 才是实际的对象。可以将此过程写得更详细些：

```
function shift_by {
    shift_count=$1
    shift

    shift $shift_count

    echo "$*"
}
```

如果对象的路径过长或者要处理的对象数量过多，可能会超出系统的 ARG_MAX 限制（详见 15.13 节）。对于前一种情况，可以更改目录，使之更靠近操作对象，从而缩短路径长度，或者使用符号链接。对于后一种情况，可以使用更复杂的 for 循环。

```
objects_to_keep=5
counter=1

for file in /path/with/many/many/files/*e*; do
    if [ $counter -gt $objects_to_keep ]; then
        remainder="$remainder $file"
    fi
    (( counter++ ))
done

[ -n "$remainder" ] && echo "rm -rf $remainder"
```

进行类似操作的一种常见方法是涓滴方案（trickle-down scheme），如下所示。

```
rm -rf backup.3/
mv     backup.2/ backup.3/
mv     backup.1/ backup.2/
cp -al backup.0/ backup.1/
```

很多情况下，这种方案的效果不错，尤其是配合硬链接来保留磁盘空间的同时允许多重备份，参见 Rob Flickenger 所著的 *Linux Server Hacks*（O'Reilly 出版）一书中的 Hack #42。但如果现有对象的数量变化不定或者事先并不知晓，那该方法就不管用了。

17.17.4 参考

- help for
- help shift
- Rob Flickenger 所著的 *Linux Server Hacks*（O'Reilly 出版）中的 Hack #42
- 13.5 节
- 15.13 节
- 17.18 节

17.18 写入循环日志

17.18.1 问题

你需要生成数据文件或日志，但又不想花费太多精力来清除其中那些已经过时的部分。

17.18.2 解决方案

将数据写入一组循环文件或目录，例如，每周、每月或数月循环一次。你需要用一种方法在新的循环周期开始时清除旧数据。

17.18.3 讨论

仅当你有一些定义明确，能够作为循环的序列时（例如，一天中的小时数、每周的天数、每月的天数，或某几个月），此方案才有效。但事实证明，这种序列的涵盖面颇广。

我们先来看一个例子，按照每周天数循环的日志文件如下所示。

```
1_Mon.log
2_Tue.log
3_Wed.log
4_Thu.log
5_Fri.log
6_Sat.log
7_Sun.log
```

为了让文件能够按照人类易读的形式排序，我们使用了略有些怪异的 strftime 格式 %u_%a（没错，sort 可以根据一周中的七天来排序，但 ls 做不到）。周一的所有日志消息全部写入 1_Mon.log，以此类推，等到周日午夜，再绕回周一。

典型格式包括：

```
$ printf "%(%u_%a)T"     # 星期几
2_Tue

$ printf "%(%d)T"        # 月中的某天
06

$ printf "%(%m_%b)T"     # 月份
12_Dec
```

唯一棘手的地方是，要在开始写入本周一的数据前先清除上周一的数据。如果你有一条日志语句，每天总是第一个运行，那么该语句应该用 > 代替 >> 来截断输出文件，其他地方可以使用 >> 进行内容追加。但要小心竞争条件（race condition），务必确保它是在正确的那天（correct day）执行的第一条语句。或许更安全的做法是安排 cron 作业，在午夜前的几分钟删除明天的数据。这样就消除了竞争条件，因为你清楚上一次写入该文件的时间是一周（或者任何时期）前，不过风险还是存在的：如果 cron 作业运行失败，那就无法清除数据了。

另一种方法是每次都调用日志记录函数来删除明天的数据。这种方法虽然稳健，但效率不高，因为大多数情况下，并没有什么要删除的数据。因为总得删除“明天”，所以也将时间窗口减少到了 $N - 1$。

例如：

```
function mylog {
    local today tomorrow

    # 今天的日志
    printf -v today "%(%u_%a)T"
    echo "$*" >> $HOME/weekly_logs/$today.txt    # 例如，1_Mon

    # 清除明天的数据
```

```
    tomorrow=$(date -d 'tomorrow' '+%u_%a')
    rm -f $HOME/weekly_logs/$tomorrow.txt
}
```

注意我们是如何使用 bash 的内建命令 printf %(*strftime format*)T 和 GNU data 命令的实用选项 -d 或 --date 的 *tomorrow* 参数的。printf 的效率更高，因为 bash 知道是什么时间，

不需要子 shell 和外部程序，但它无法告诉你明天是哪一天。

以下是几个 cron 条目示例，用于只关注特定目标的脚本。

```
# 每小时检查一次
06 * * * * /home/user/report/keep-an-eye-on-it.sh          ❶

# 每周报告一次
02 00 * * Mon ln -fs "queue-report_$(date '+\%F').txt"
/home/user/report/keep-an-eye-on-it.txt                    ❷

# 重新开始新的一天（这意味着要滚动6~7天……）
03 00 * * * rm -f /home/user/report/$(date '+\%u_\%a')/*   ❸
```

❶ 每小时运行一次脚本。

❷ 创建类似于 keep-an-eye-on-it.txt → keep-an-eye-on-it_2017-10-09.txt 的符号链接，当脚本向 keep-an-eye-on-it.txt 写入时，结果实际上进入了每周归档的 keep-an-eye-on-it_2017-10-09.txt。在某些版本的 date 中，%F 是 %Y-%m-%d 的便捷写法。

❸ 在午夜前删除“明天的”目录中的内容。注意，在某些版本的 cron（如 Vixie-cron）中，必须对 % 进行转义，否则会出现类似于“Syntax error: EOF in backquote substitution”的错误。

17.18.4 参考

- help printf

- man date
- 11.10 节
- 17.19 节
- 19.10 节

17.19 循环备份

17.19.1 问题

你需要备份一些数据，但又不想花费太多精力来清除过时的备份。

17.19.2 解决方案

将数据写入一组循环文件或目录，例如，每周、每月，或者数月循环一次。你需要用一种方法在新的循环周期开始时清除旧数据。

17.19.3 讨论

我们发现 Firefox 偶尔会丧失其会话还原功能，因此编写了一个简单的脚本来备份并还原会话（参见例 17-4）。

例 17-4 ch17/ff-sessions

```
#!/usr/bin/env bash
# 实例文件：ff-sessions
# 保存/恢复Firefox会话

# cron条目：
# 45 03,15 * * * opt/bin/ff-sess.sh qsave
❶

FF_DIR="$HOME/.mozilla/firefox"
date=$(date '+%u_%a_%H')  # 例如：3_Wed_15
❷

case "$1" in
    qsave    )  # 静默保存
        cd $FF_DIR
```

```
        rm -f ff_sessions_$date.zip
❸
        zip -9qr ff_sessions_$date.zip */session*
❹
    ;;

    save    )  # 保存时附带提示信息（调用qsave）
        echo "SAVING '$FF_DIR/*/session*' data into '$date' file"
❺
        $0 qsave
❻
    ;;

    restore )
        [ -z "$2" ] && { echo "Need a date to restore from!"; exit
1; } ❼
        date="$2"
❽
        echo "Restoring session data from '$date' file"
        cd $FF_DIR
        unzip -o ff_sessions_$date.zip
❾
    ;;

    *       )
❿
        echo 'Save/Restore FF sessions'
        echo "$0 save"
        echo "$0 restore <date>"
        echo "    e.g., $0 restore 3_Wed_15"
    ;;
esac
```

❶ 按照注释中给出的形式运行 cron 条目，本例为每天运行两次，时间分别为凌晨 3:45 和下午 3:45。

❷ 和 17.18 节一样，我们在便于阅读的每周日期前加上一个数字，以便其能够正确排序，然后加上作业运行的时刻。

❸ zip 通常会将文件追加到原 ZIP 文件中，因此我们先删除现有的 ZIP 文件，以免冲突。-f（force）选项可以避免删除不存在的文件时产生错误。

❹ 我们使用 -9 来获得最大压缩比，-q 表示静默模式，-r 表示执行递归压缩，然后备份以 session 起始的 Firefox 配置目录中的所有内容。

❺ “save”参数会显示消息来表明当前正在进行的操作。

❻ 然后调用“静默”（quiet）保存。通常你只希望出错时 cron 作业才产生输出；不然每次作业运行时，你都会收到电子邮件。

❼ 我们将原本应该是多行的内容压缩成一行，虽然健全性检查很重要，但我们不想偏离这部分的重点。

❽ 为了后续代码的清晰性，我们将 $2 赋给 $date。这么做看起来可能挺蠢的，毕竟也没多少代码，却是一种应该遵循的良好实践，最好贯彻一致，别再浪费时间去想“我到底该不该？”。

❾ 用 unzip 的 -o 选项直接覆盖已有文件（如果存在的话），这样就不会再出现提示信息了。

❿ 最后，如果未提供选项或选项有错误，则输出用法帮助。

该脚本很容易扩展成按照每周、每月、每年进行备份，只需要加入更多选项或修改脚本，使其能够接受参数，而不是像现在这种硬编码的“now”，然后添加更多相应的 cron 作业。注意，在某些版本的 cron（如 Vixie-cron）中，必须对 % 进行转义，否则会出现类似于“Syntax error: EOF in backquote substitution”的错误。

17.19.4 参考

- man zip
- man unzip
- Session Restore（“Troubleshooting”）
- 17.18 节

17.20 搜索不包含grep进程自身在内的ps输出

17.20.1 问题

你想用 grep 搜索 ps 命令的输出，但不希望结果中包含 grep 进程自身。

17.20.2 解决方案

修改查找模式，设计一个正则表达式，使其不会匹配到 ps 所显示的字面文本。

```
$ ps aux | grep 'ssh'
root   366  0.0  1.2  340  1588 ?? Is   20Oct06  0:00.68
/usr/sbin/sshd
root 25358  0.0  1.9  472  2404 ?? Ss   Wed07PM  0:02.16 sshd:
root@ttyp0
jp   27579  0.0  0.4  152   540 p0 S+    3:24PM  0:00.04 grep ssh
$ ps aux | grep '[s]sh'
root   366  0.0  1.2  340  1588 ?? Is   20Oct06  0:00.68
/usr/sbin/sshd
root 25358  0.0  1.9  472  2404 ?? Ss   Wed07PM  0:02.17 sshd:
root@ttyp0
$
```

17.20.3 讨论

该方法可行的原因是 [s] 属于正则表达式字符类（character class），其中只包含单个小写字母 s，这意味着 [s]sh 能够匹配 ssh，但不会匹配 ps 所显示的字面字符串 grep [s]sh[5]。或许你还见过如下所示的解决方案（效率不高且比较烦琐）。

[5] ps aux 的输出中包含以 grep [s]sh 形式显示的 grep 进程，当在管道的另一侧使用 grep '[s]sh' 进行搜索时，正则表达式 [s]sh 无法匹配字面字符串 [s]sh，故最终输出也就不会显示 grep 进程了。——译者注

```
ps aux | grep 'ssh' | grep -v grep
```

17.20.4 参考

- man ps
- man pgrep
- man grep

17.21 确定某个进程是否正在运行

17.21.1 问题

你需要判断某个进程是否正在运行，但未必知道对应的 PID。

17.21.2 解决方案

如果不知道 PID，可以用 grep 搜索 ps 命令的输出，看看你要查找的进程是否正在运行（至于为什么要用模式 [s]sh，详见 17.20 节）。

```
ps -ef | grep -q 'bin/[s]shd' && echo 'ssh is running' || echo
'ssh not running'
```

很好，但你知道事情不会这么简单，对吧？没错。困难之处在于 ps 在不同的系统中存在很大差异。

在不知道 PID 的情况下，例 17-5 中给出的脚本可以确定进程是否正在运行。

例 17-5 ch17/is_process_running

```
# 实例文件：is_process_running

# 你敢信？！？
case `uname` in
    Linux|AIX) PS_ARGS='-ewwo pid,args'    ;;
```

```
    SunOS)      PS_ARGS='-eo pid,args'      ;;
    *BSD)       PS_ARGS='axwwo pid,args'    ;;
    Darwin)     PS_ARGS='Awwo pid,command'  ;;
esac

if ps $PS_ARGS | grep -q 'bin/[s]shd'; then
    echo 'sshd is running'
else
    echo 'sshd not running'
fi
```

如果你知道 PID（假设是从锁文件或环境变量中得知的），直接根据 PID 搜索即可（要仔细地将 PID 与其他可识别的字符串一并拿来匹配，以免出现冲突，因为你用来进行匹配的 PID 可能正好与某些随机进程的陈旧 PID 相同）。在 grep 或 ps 的 -p 选项参数中使用该 PID。

```
# Linux
$ ps -wwo pid,args -p 1394 | grep 'bin/sshd'
1394 /usr/sbin/sshd

# BSD
$ ps ww -p 366 | grep 'bin/sshd'
366 ?? Is 0:00.76 /usr/sbin/sshd
```

如果你的系统中安装了 pgrep，也可以拿来使用。pgrep 的选项众多，但我们只用 -f 搜索整个命令行，而不仅仅是进程名，-a 可以显示完整的命令行。

```
$ pgrep -fa 'bin/[s]shd' ; echo $?
1278 /usr/sbin/sshd -D
```

17.21.3 讨论

第一种解决方案的测试以及 grep 部分需要解释一下。$() 的两边要加上 ""，这样一来，只要 grep 有输出，测试结果即为真。如果因为 grep 什么都没匹配到而没有产生任何输出，则测试结果为假。你只需要确保 ps 和 grep 各司其事就行了。

遗憾的是，ps 是所有 Unix 版本中最分裂的命令之一。似乎每种 Unix 和 Linux 都有不同的 ps 选项以及不同的处理过程。我们能够

告诉你的就是对脚本所在的所有系统进行全面的测试。

只要是能用正则表达式描述的内容，都可以轻松地搜索，但要确保正则表达式足够具体，不会匹配到无关内容。这就是为什么我们要使用 bin/[s]sh，而不是简单的 [s]shd，后者还会匹配到用户连接（参见 17.20 节）。同时，/usr/sbin/[s]shd 搞不好也会出问题，因为有些疯狂的系统并没有使用这个位置。太过具体和不够具体之间往往存在一条细微的界线。例如，也许某个程序可以使用不同的配置文件运行多个实例，如果要从中找出正确的实例，就得搜索配置文件。如果你拥有足够的权限来查看其他用户的进程，那么同样的情况可能也适用于用户。

> 在低于 11.3 SRU 5 的 Solaris 版本中，ps 通过硬编码将参数长度限制为 80 个字符。如果使用的路径或命令太长，同时还需要检查配置文件名，则可能会触碰到此限制。

17.21.4 参考

- man ps
- man grep
- 17.20 节
- man pgrep
- man pidof
- man killall

17.22 为输出添加前缀或后缀

17.22.1 问题

出于某种原因，你想为特定命令的每行输出添加前缀或后缀。例如，你收集了来自多台机器的 last 统计信息，如果每行都包含主机名的话，使用 grep 或者解析数据时就容易多了。

17.22.2 解决方案

根据需要将适合的数据通过管道传入 while read 循环和 printf。例如，下列语句会打印出 $HOSTNAME，然后是一个制表符，接着是 last 命令的非空行输出。

```
last | while read i; do [[ -n "$i" ]] && printf "%b"
"$HOSTNAME\t$i\n"; done
```

也可以用 awk 为每行添加文本。

```
last | awk "BEGIN { OFS=\"\t\" } ! /^\$/ { print \"$HOSTNAME\",
\$0}"
```

或者写入新的日志文件。

```
last | while read i; do [[ -n "$i" ]] && printf "%b"
"$HOSTNAME\t$i\n"; \
    done > last_$HOSTNAME.log
```

或者：

```
last | awk "BEGIN { OFS=\"\t\" } ! /^\$/ { print \"$HOSTNAME\",
\$0}" \
    > last_$HOSTNAME.log
```

17.22.3 讨论

我们用 [[-n "$i"]] 删除 last 输出中的空行，然后用 printf 显示数据。这种方法更简单，但需要更多的步骤（last、while、read，第二种方法只用到了 last 和 awk）。根据需求，你也许会觉得其中一种方法更容易记忆、可读性更好，或者比其他方法速度更快。

这里用到的 awk 命令有一个技巧。你经常会看到 awk 命令出现在单引号内，以避免 shell 将其中的 awk 变量解释为 shell 变量。但在这个例子中，我们想让 shell 解释 $HOSTNAME，因此将命令放入了双引号。这就需要我们使用反斜线将命令中不希望由 shell 处理的部分进行转义，也就是内部的双引号、行尾锚点 $，以及包含当前行内容的 awk 变量 $0。

要想添加后缀，移动 $0 变量就行了。

```
last | while read i; do [[ -n "$i" ]] && printf "%b"
"$i\t$HOSTNAME\n"; done
```

或者使用 awk。

```
last | awk "BEGIN { OFS=\"\t\" } ! /^\$/ { print \$0,
\"$HOSTNAME\"}"
```

也可以使用 Perl。

```
last | perl -ne "print qq($HOSTNAME\t\$_) if ! /^\s*$/;"
```

或者 sed（注意，→表示字面制表符，先后按下 Ctrl-V 和 Ctrl-I 即可输入）。

```
last | sed "s/./$HOSTNAME → &/; /^$/d"
```

在 Perl 单行脚本中，我们用 qq() 代替双引号，这样就不用对那些不希望 shell 解释的部分命令进行转义了。最后一部分是正则表达式，它可以匹配空行或仅包含空白字符的行，$_ 是 Perl 用于表示当前行的惯用写法。在 sed 命令中，我们使用前缀和匹配到的字符（&）替换那些至少包含一个字符的行，然后删除所有的空行。

17.22.4 参考

- Arnold Robbins 所著的 *Effective awk Programming, 4th Edition*（O'Reilly 出版）
- Arnold Robbins 与 Dale Dougherty 合著的 *sed & awk, 2nd Edition*（O'Reilly 出版）
- 1.8 节
- 13.15 节
- 13.18 节

17.23 对行进行编号

17.23.1 问题

你需要对文本文件的各行进行编号，以作参考或示例之用。

17.23.2 解决方案

感谢 Michael Wang 贡献了纯 shell 脚本实现的解决方案，还提醒了我们 `cat -n` 的用法。注意，样例文件 lines 的末尾是一个空行。

```
$ i=0; while IFS= read -r line; do (( i++ )); echo "$i $line";
done < lines
1 Line 1
2 Line 2
3
4 Line 4
5 Line 5
6
```

`cat` 的有效用途：

```
$ cat -n lines
     1 Line 1
     2 Line 2
     3
     4 Line 4
     5 Line 5
     6

$ cat -b lines
     1 Line 1
     2 Line 2
     3 Line 4
     4 Line 5
```

17.23.3 讨论

如果只需要在屏幕上显示行号，可以使用 `less -N`。

```
$ /usr/bin/less -N filename
      1 Line 1
      2 Line 2
      3
```

```
      4 Line 4
      5 Line 5
      6
lines (END)
```

在一些过时的 Red Hat 系统的老版 less 中，行号是有毛病的。可以使用 less -V 检查你所用的版本号。已知有问题的版本号为 358+iso254（如 Red Hat 7.3 和 8.0）。378+iso254（如 RHEL3）和 382（RHEL4，Debian Sarge）这两个版本没问题。至于其他版本，我们并没有测试过。这个问题不易察觉，也许和旧的 iso256 补丁有关。你可以拿最后一行的行号与 vi 和 Perl 的例子做比较，后者没有任何问题。

也可以使用 vi（或者 vi 的只读版 view）的 :set nu! 命令。

```
$ vi filename
      1 Line 1
      2 Line 2
      3
      4 Line 4
      5 Line 5
      6
~
:set nu!
```

vi 的选项众多，启动 vi 时可以用 vi +3 -c 'set nu!' *filename* 的形式启用行编号功能并将光标置于第 3 行。要想更好地控制行号的显示方式，也可以使用 nl、awk 或者 perl。

```
$ nl lines
     1 Line 1
     2 Line 2

     3 Line 4
     4 Line 5

$ nl -ba lines
     1 Line 1
     2 Line 2
     3
     4 Line 4
     5 Line 5
     6
```

```
$ awk '{ print NR, $0 }' filename
1 Line 1
2 Line 2
3
4 Line 4
5 Line 5
6

$ perl -ne 'print qq($.\t$_);' filename
1 → Line 1
2 → Line 2
3 →
4 → Line 4
5 → Line 5
6 →
```

在 awk 中，NR 是当前输入文件的当前行号；Perl 中则是 $.，因此用它们来输出行号非常简单。注意，我们 → 用表示 Perl 输出中的制表符，而 awk 默认使用空格。

17.23.4 参考

- man cat
- man nl
- man awk
- man less
- man vi
- 8.15 节

17.24 生成序列

17.24.1 问题

你需要生成一个数字序列，可能与其他文本一起用于测试或其他目的。

17.24.2 解决方案

使用 awk，因为它的适用性最好。

```
$ awk 'END { for (i=1; i <= 5; i++) print i, "text"}' /dev/null
1 text
2 text
3 text
4 text
5 text

$ awk 'BEGIN { for (i=1; i <= 5; i+=.5) print i}' /dev/null
1
1.5
2
2.5
3
3.5
4
4.5
5
```

17.24.3 讨论

在有些系统（尤其是 Solaris）中，除非指定好文件（如 /dev/null），否则 awk 会一直处于等待状态。其他系统并不会这样，因此可以放心使用。

注意，print 语句中的变量是 i，不是 $i。如果不小心写成 $i，那么其插值结果为被处理的当前行的某个字段。由于我们什么都没处理 [6]，就算误用了 $i，也什么结果都得不到。

[6]因为指定的文件是 /dev/null。——译者注

BEGIN 和 END 模式允许在实际处理文件时执行启动处理（startup）或收尾清理（cleanup）操作。因为我们并没有处理文件，所以需要选用其中一种模式，从而让 awk 知道，即便没有正常的输入，也得做点什么。在本例中，用哪种模式都行。

GNU 实用工具 seq 正好能胜任本例所要求的任务，只不过很多系统（如 Solaris、老版的 macOS，以及 BSD）没有默认安装该工具。seq 提供了一些实用的格式化选项，而且只能够处理数值，但要注意，BSD 版本和 GNU 版本可能有所不同。

幸好在 bash 2.04 及其后续版本中，你可以在 `for` 循环中执行整数运算。

```
# 仅适用于bash 2.04及其后续版本，只能处理整数
$ for ((i=1; i<=5; i++)); do echo "$i text"; done
1 text
2 text
3 text
4 text
5 text
```

bash 3.0 还引入了 {*x*..*y*} 括号扩展，这允许我们在其中使用整数或单字符。

```
# 仅适用于bash 3.0+，只能处理整数或单字符
$ printf "%s text\n" {1..5}
1 text
2 text
3 text
4 text
5 text

$ printf "%s text\n" {a..e}
a text
b text
c text
d text
e text
```

在 bash 4.0 及其后续版本中，还可以在 {*x*..*y*} 括号扩展内添加前导数字 0。

```
# 仅适用于bash 4.0+，可以在整数前有选择地添加前导数字0
$ for num in {01..16}; do echo ssh server$num; done
ssh server01
ssh server02
ssh server03
...
ssh server14
ssh server15
ssh server16
```

17.24.4 参考

- man seq
- man awk
- comp.lang.awk FAQ

17.25 模拟DOS的pause命令

17.25.1 问题

你正在迁移 DOS/Windows 批处理文件，希望能够模拟 DOS 的 pause 命令。

17.25.2 解决方案

可以在函数中使用 read -n1 -p 命令。

```
pause ()
{
    read -n 1 -p 'Press any key when ready...'
}
```

-n 是 bash 2.04 引入的。如果必须省略 -n1（说实话，你的 bash 版本太旧了，该升级了），那么随后的提示信息需要进行修改 [7]，因为只有按 Enter 键才能结束输入。应该将命令改成：read -p '*Press the ENTER key when ready...*'。

[7]这里所说的提示信息指的是 'Press any key when ready...'。——译者注

17.25.3 讨论

读取完 *n* 个字符或一个换行符后，-n *nchars* 选项会返回。因此，-n1 会在（等待）用户按下任意键之后返回。-p 选项之后的字符串参数会出现在读取输入之前。在这个例子中，该字符串和 DOS 的 pause 命令的输出相同。

17.25.4 参考

- help read
- 1.12 节

17.26 为数值添加逗号

17.26.1 问题

你想为大数值添加千位分隔符。

17.26.2 解决方案

根据系统和配置，可以在适当的语言环境中使用 printf 的格式标志。感谢 Chet Ramey 的帮助，这是目前为止最简单的解决方案。

```
$ LC_NUMERIC=en_US.UTF-8 printf "%'d\n" 123456789
123,456,789
$ LC_NUMERIC=en_US.UTF-8 printf "%'f\n" 123456789.987
123,456,789.987000
$
```

还要感谢 Michael Wang 提供的纯 shell 脚本解决方案以及相关的讨论（参见例 17-6）。

例 17-6 ch17/func_commify

```
# 实例文件：func_commify

function commify {
    typeset text=${1}

    typeset bdot=${text%%.*}
    typeset adot=${text#${bdot}}

    typeset i commified
    (( i = ${#bdot} - 1 ))

    while (( i>=3 )) && [[ ${bdot:i-3:1} == [0-9] ]]; do
        commified=",${bdot:i-2:3}${commified}"
        (( i -= 3 ))
    done
```

```
    echo "${bdot:0:i+1}${commified}${adot}"
}
```

或者，你也可以尝试 sed FAQ 中给出的 sed 解决方案。例如：

```
sed ':a;s/\B[0-9]\{3\}\>/,&/;ta'  /path/to/file
# GNU sed
sed -e :a -e 's/\(.*[0-9]\)\([0-9]\{3\}\)/\1,\2/;ta' /path/to/file
# 其他版本的sed
```

17.26.3 讨论

这个 shell 函数的编写思路和使用纸笔的人类处理这个问题时的逻辑相同。首先，检查字符串并找到小数点（如果存在的话）。忽略小数点之后的所有内容，处理小数点前的字符串。

shell 函数将小数点前的字符串保存在 $dbot 中，小数点之后的字符串（包括小数点）保存在 $adot 中。如果没有小数点，则所有的内容都保存在 $dbot 中，$adot 为空。接下来，在小数点前的部分中，从右向左移动，当满足以下两个条件时，插入一个逗号：

- 有 4 个或更多字符；
- 逗号之前的字符是数字。

该逻辑是由函数的 while 循环实现的。

Tom Christiansen 和 Nathan Torkington 合著的 *Perl Cookbook, 2nd Edition*（O'Reilly 出版）中的 2.16 节也提供了一种解决方案，如例 17-7 所示。

例 17-7 ch17/perl_sub_commify

```
# 实例文件：perl_sub_commify

#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# 为数值添加千位分隔符
# 返回：添加过逗号的输入值
# 摘自Perl Cookbook2 2.16，第84页
sub commify {
```

```
    @_ == 1 or carp ('Sub usage: $withcomma =
commify($somenumber);');

    # 摘自Perl Cookbook1 2.17，第64页，或Perl Cookbook2 2.16，第84页
    my $text = reverse $_[0];
    $text =~ s/(\d\d\d)(?=\d)(?!\d*\.)/$1,/g;
    return scalar reverse $text;
}
```

美国以逗号作为千位分隔符，但很多国家使用的是点号。

17.26.4 参考

- sed FAQ，4.14 节
- Tom Christiansen 与 Nathan Torkington 合著的 *Perl Cookbook, 2nd Edition*（O'Reilly 出版）中的 2.16 节
- 13.19 节

第 18 章　写得少，干得快

虽然处理器速度、传输率、网速和 I/O 功能均得以改善，但仍有一个因素在限制着 bash 的众多应用——用户的输入速度。当然，脚本编写始终是我们关注的重点，但 bash 的交互式用法仍然是其应用及功用性的重要组成部分。我们讲过的许多脚本技术也可以交互式使用，但除非知道一些快捷用法，否则你会发现自己要面对大量的键盘输入。

早在 Unix 诞生之初，电传打字机每秒只能输出约 10 个字符，出色的打字员的打字速度比键盘处理速度还快。Unix 就是在这样的环境中开发出来的，只要命令能被理解，没人愿意输入不必要的内容，可能正因如此，Unix 才具有了某些简洁性。

站在历史的另一端（也就是现在），处理器速度如此之快，以至于在等待用户输入期间都是闲置的，甚至在你敲完命令之前，还能去翻看命令历史，在 $PATH 的目录列表中查找可能的命令和有效的参数。

将针对每种情况所开发出的技术结合起来，可以大大降低发出 shell 命令所需要的输入量，这可不仅仅是因为懒。相反，你可能会发现这些减少按键量的措施很有用，因为它们提高了准确性，避免了错误，也省了再恢复备份。

18.1　在任意目录之间快速移动

18.1.1　问题

你发现自己要在两个或更多目录之间频繁移动，一会切换到这里，一会切换到那里，来回跳转。这些目录之间隔得还挺远，反复输入冗长的路径让人疲惫不堪。

18.1.2　解决方案

使用内建命令 pushd 和 popd 来管理目录栈，轻松地在目录之间切换。下面是一个简单的示例。

```
$ cd /tmp/tank

$ pwd
/tmp/tank

$ pushd /var/log/cups
/var/log/cups /tmp/tank

$ pwd
/var/log/cups

$ ls
access_log error_log page_log

$ popd
/tmp/tank

$ ls
empty full

$ pushd /var/log/cups
/var/log/cups /tmp/tank

$ pushd
/tmp/tank /var/log/cups

$ pushd
/var/log/cups /tmp/tank

$ pushd
/tmp/tank /var/log/cups

$ dirs
/tmp/tank /var/log/cups
```

18.1.3 讨论

栈是一种**后进先出**的结构，这两个命令也正是这么做的。如果对一个新目录使用 pushd，那么它会将前一个目录压入栈中。当使用 popd 时，它会弹出栈顶保存的当前位置，切换到新的栈顶目录。使用这些命令更改位置时，会从左到右输出目录栈中的值，对应于栈中自顶向下的顺序。

如果使用 pushd 时没有指定目录，那么它会交换栈顶的两个目录的位置，这样就可以重复使用 pushd 命令来实现两者之间的切换。cd - 命令也能够达到相同效果。

你仍然能使用 cd 来更改位置，它会改变当前目录，也就是栈顶目录。如果不记得目录栈中都有哪些目录，可以使用内建命令 dirs 按照从左到右的顺序显示。加上 -v 选项后，显示形式更形象。

```
$ dirs -v
 0 /var/tmp
 1 ~/part/me/scratch
 2 /tmp
$
```

波浪号（~）是用户主目录的简写形式。数字可用来调整栈内目录的位置。pushd +2 会将编号为 2 的目录置为栈顶（并切换到该目录）并将其他目录下压。

```
$ pushd +2
/tmp /var/tmp ~/part/me/scratch
$ dirs -v
 0 /tmp
 1 /var/tmp
 2 ~/part/me/scratch
$
```

要想看到类似于栈的目录列表，但又不希望出现编号，可以使用 -p 选项。

```
$ dirs -p
/tmp
/var/tmp
~/part/me/scratch
$
```

多练练手，你会发现在目录间重复移动比以前更快、更容易了。

18.1.4 参考

- 1.4 节

- 14.3 节
- 16.6 节
- 16.15 节
- 16.22 节

18.2 重复上一个命令

18.2.1 问题

你刚刚输入了一个又长又麻烦的命令，其中包含了冗长的路径名和一堆复杂的参数。现在需要重新执行该命令。难道还得再输入一次？

18.2.2 解决方案

这个问题有两种解决方法。第一种方法只需要在提示符下输入两个惊叹号，然后 bash 就会显示并重复执行上一个命令。例如：

```
$ /usr/bin/somewhere/someprog -g -H -yknot -w /tmp/soforthandsoon
...
$ !!
/usr/bin/somewhere/someprog -g -H -yknot -w /tmp/soforthandsoon
...
$
```

另一种（更现代的）方法是使用箭头键。按上箭头键会回滚到执行过的上一个命令。如果找到了需要的命令，按下 Enter 键就可以（再次）执行该命令。

18.2.3 讨论

输入 !!（有时也叫作 bang bang）时会显示出命令，这样就可以知道要执行的是哪个。

> Chet 告诉我们，csh 风格的 !! 历史记录在未来的 bash 版本中可能不再默认启用了，因为他收到了多个希望将其关闭的请求。即便如此，它还是能够作为选项使用的。

18.2.4 参考

- 16.10 节
- 16.14 节
- 18.3 节

18.3 执行类似命令

18.3.1 问题

执行了又长又难输入的命令后，你得到的是一条错误信息，它告诉你命令行中间有一处小小的输入错误。难道还得整个重新输入一遍？

18.3.2 解决方案

18.2 节中讨论过的 !! 命令允许添加额外的编辑限定符。你的 sed 水平如何？在 !! 之后加上一个冒号，然后是类似于 sed 的替换表达式，如下所示。

```
$ /usr/bin/somewhere/someprog -g -H -yknot -w /tmp/soforthandsoon
Error: -H not recognized. Did you mean -A?
$ !!:s/H/A/
/usr/bin/somewhere/someprog -g -A -yknot -w /tmp/soforthandsoon
...
$
```

你也可以按照 18.2 节中描述的方法使用箭头键调出历史命令，但如果在速度缓慢的网络链路上碰到冗长的命令行，上述语法会更适合，只要你习惯这种形式。

18.3.3 讨论

要是打算使用该特性，进行替换时要小心。如果输入 !!:s/g/h/ 来尝试修改 -g 选项，结果修改的却是第一个字母 g，也就是位于命令名结尾的那个 g，那么最终执行的命令就变成了 /usr/bin/somewhere/someproh。

如果想修改命令行中出现的所有实例，则需要在 s 之前加上 g（以表示全局替换），如下所示。

```
$ /usr/bin/somewhere/someprog -g -s -yknots -w /tmp/soforthandsoon
...
$ !!:gs/s/S/
/usr/bin/Somewhere/Someprog -g -S -yknotS -w /tmp/SoforthandSoon
...
$
```

为什么 g 要写在 s 前，而不是像 sed 语法那样出现在 s 之后呢？这是因为任何出现在闭合斜线（closing slash）后的内容都会被视为要追加到命令后的新文本，要想再次执行命令时添加其他参数，这种写法很方便。

18.3.4 参考

- 16.10 节
- 16.14 节
- 18.2 节

18.4 快速替换

18.4.1 问题

你想知道有没有更简单的语法可对先前所执行的命令进行替换，然后重新执行修改后的结果。

18.4.2 解决方案

使用脱字符（^）替换机制。

```
$ /usr/bin/somewhere/someprog -g -A -yknot -w /tmp/soforthandsoon
...
$ ^-g -A^-gB^
/usr/bin/somewhere/someprog -gB -yknot -w /tmp/soforthandsoon
...
```

始终可以只使用箭头键调出历史命令，但如果在速度缓慢的网络链路上碰到冗长的命令行，上述语法会更适合，只要你习惯这种形式。

18.4.3 讨论

进行命令行内容替换时要以 ^ 起始，然后是要替换的文本，接着是另一个 ^ 及新的文本。仅当你想在命令行最后添加更多文本时，结尾的（第 3 个）^ 才有必要。

```
$ /usr/bin/somewhere/someprog -g -A -yknot
...
$ ^-g -A^-gB^ /tmp^
/usr/bin/somewhere/someprog -gB -yknot /tmp
...
```

要想删除部分内容并将其替换为空值，可以像下面这样做。

```
$ /usr/bin/somewhere/someprog -g -A -yknot /tmp
...
$ ^-g -A^^
/usr/bin/somewhere/someprog -yknot /tmp
...
$ ^knot^
/usr/bin/somewhere/someprog -gA -y /tmp
...
$
```

第一个例子使用了 3 个 ^。第二个例子只使用了 2 个 ^，因为我们想将“knot”替换为空，以换行符（Enter 键）结束本行即可。

脱字符替换法非常方便。不过很多 bash 用户认为 18.3 节中演示过的 !!:s/.../.../ 语法更易用。你怎么认为?

18.4.4 参考

- 16.10 节
- 16.14 节
- 18.3 节

18.5 参数重用

18.5.1 问题

重用上一个命令很简单，使用 !! 就行了，但你需要的未必总是整个命令。如何只重用最后一个参数呢？

18.5.2 解决方案

用 !$ 指明上一个命令中的最后一个参数。!:1 表示第一个参数，!:2 表示第二个参数，以此类推。

18.5.3 讨论

多个命令使用相同的文件名为参数是司空见惯的事情。最常见的场景之一就是程序员编辑源代码文件，然后编译、再编辑，再编译……有了 !$，事情就方便多了。

```
$ vi /some/long/path/name/you/only/type/once
...
$ gcc !$
gcc /some/long/path/name/you/only/type/once
...
$ vi !$
vi /some/long/path/name/you/only/type/once
...
$ gcc !$
gcc /some/long/path/name/you/only/type/once
...
$
```

明白其中的意思了吗？这不仅省去了大量的键盘输入，还避免了错误。如果编译时输错文件名，那编译的可就不是刚刚编辑好的源代码文件了。有了 !$，就可以始终得到刚刚用过的文件名。要是想重用的参数位于命令行内部，可以使用带编号的 !: 命令来获取。我们来看一个示例。

```
$ munge /opt/my/long/path/toa/file | more
...
```

```
$ vi !:1
vi /opt/my/long/path/toa/file
...
$
```

你可能想尝试着用 !$，但在这个例子中，得到的是 more，并非你想要再次编辑的文件名。

18.5.4 参考

- bash 手册页中的“Word Designators”部分
- 18.2 节

18.6 名称补全

18.6.1 问题

路径名有时实在太长了。bash 的运行位置位于这台计算机的……有没有省事点的办法？

18.6.2 解决方案

不确定时按 Tab 键。bash 会尝试为你补全路径名。如果没有反应，可能是因为没有匹配的路径或者匹配的路径不止一个。再按一次 Tab 键，bash 会列出所有可用的选择，然后重新显示刚才已经输入过的那部分命令，以便你继续往下进行。多输入点内容（直到没有歧义），再按 Tab 键让 bash 帮忙将名称补充完整。

18.6.3 讨论

bash 甚至聪明到能将选择范围限制在特定的文件类型。如果你输入 unzip，然后输入路径的起始部分，接着按 Tab 键，bash 会只选择以 .zip 结尾的文件名进行补全，即便还有其他文件名也匹配已输入的部分。例如：

```
$ ls
myfile.c     myfile.o     myfile.zip
$ ls -lh myfile<tab><tab>
myfile.c     myfile.o     myfile.zip
$ ls -lh myfile.z<tab>ip
-rw-r--r--     1 me mygroup 1.9M 2006-06-06 23:26 myfile.zip
$ unzip -l myfile<tab>.zip
...
$
```

最后那个 unzip 的示例需要安装 16.19 节中讨论过的 bash-completion 软件包。如果 bash 未能提供补全建议，需要确认是否已经安装该软件包。

18.6.4 参考

- 16.10 节
- 16.19 节

18.7 安全第一

18.7.1 问题

一不小心就会输错字符。[1]（不信你瞧！）即便是简单的 bash 命令，由此带来的后果也非常严重：你会移动错或删错文件。如果再加上模式匹配，结果更让人心跳，因为模式中的输入错误会导致南辕北辙的结果。小心谨慎的用户会怎么做？

[1]此处原文为“It's so easy to type the wrong character by mistrake”，其中 mistrake 就拼写错了，应该是 mistake。——译者注

18.7.2 解决方案

可以使用命令历史特性和键盘便捷方式来重复参数，无须从头输入，因此能够减少输入错误。如果要用到棘手的模式来匹配文件，先用 echo 测试一下模式能否正常匹配，然后再用 !$ 进行实际操作。例如：

```
$ ls
ab1.txt  ac1.txt  jb1.txt  wc3.txt
$ echo *1.txt
ab1.txt ac1.txt jb1.txt
$ echo [aj]?1.txt
ab1.txt ac1.txt jb1.txt
$ echo ?b1.txt
ab1.txt jb1.txt
$ rm !$
rm ?b1.txt
$
```

18.7.3 讨论

echo 是检查模式匹配结果的一种方法。一旦确信结果符合预期，就可以将模式用于实际命令。这里我们要删除有特定名称的文件，没人愿意在这种事上犯错。

另外，当你使用历史命令时，可以加入 :p 修饰符，这会使得 bash 只输出命令，但并不真的执行，这也是另一种检查历史替换是否正确的省事方法。在上一节那个例子的末尾，我们可以这样做：

```
$ echo ?b1.txt
ab1.txt jb1.txt
$ rm !$:p
rm ?b1.txt
$
```

:p 修饰符使得 bash 只输出命令，并不执行，注意，参数是 ?b1.txt，但并不会被扩展成两个文件名。该修饰符仅显示出要运行的内容，只有在运行时，shell 才会将该模式扩展为两个文件名。如果你想查看是如何扩展的，可以使用 echo 命令。

18.7.4 参考

- bash 手册页中的“Modifiers”部分，可查看用于历史命令的冒号（:）修饰符。
- C.1 节
- 18.5 节

18.8 修改多个命令

18.8.1 问题

就单个替换而言，如果需要进行的改动太过复杂或涉及多个命令行，该怎么办呢？你发现自己有时得执行 history 命令，将输出重定向到文件，然后编辑该文件，再将编辑后的命令作为脚本运行。有没有更简单的方法？

18.8.2 解决方案

fc 命令可以将最近执行的命令（或者一批命令）保存在临时文件中，调用编辑器，并允许你按照适合的方式修改其中的命令，**然后在退出编辑器时自动重新执行编辑过的命令**。

18.8.3 讨论

调用哪个编辑器？fc 会选择 shell 变量 FCEDIT（如果设置过的话）中所定义的那个。如果该变量为空，则改用更通用的 EDITOR 变量，要是也为空，那就使用 vi。

哪些命令行会出现在编辑器中？如果调用 fc 时不带参数，则只使用最后一行。你也可以用参数指定某一行：fc 1004 会使用命令历史记录中编号为 1004 的命令行，而 fc -5 会使用第 5 个最近的命令。与此类似，你也可以指定范围参数，例如，fc 1001 1005 允许编辑编号为 1001 到 1005 的命令行，fc -5 -1 允许编辑最近执行过的 5 个命令。

> 当你退出 fc 命令所调用的编辑器时，它会重新执行临时文件中剩余的所有命令，**即使你未做任何改动**。如果你改变主意，不想执行任何命令，那该怎么办？可以删除文件中的所有命令行，**保存并退出**。不要试图挂起编辑器。这会使 shell 和终端处于不稳定状态。

18.8.4 参考

- 18.3 节
- 18.4 节
- 18.5 节
- 18.7 节
- `man fc` 的更多选项，包括不调用编辑器进行多行编辑

第 19 章　窍门与陷阱：新手常见错误

人无完人。大家都会犯错，尤其学习新东西时，概莫能外。一旦搞清楚了来龙去脉，那些愚蠢问题看起来简直再明白不过了。总觉得自己每一步都严格按照要求操作，竟然还会出错，那肯定是系统有毛病，到最后发现原来是敲错了一个字母，就因为这么一个小小的字母，结果却大相径庭。有些错误在初学者当中司空见惯，几乎是预料之中的事。我们都曾纠结于脚本为什么不能运行，直到为其设置了可执行权限才知道是怎么回事，这是个如假包换的新手错误。现在我们已经是老手了，再也不会犯这种错误。什么，再也不会？好吧，基本上不会。毕竟，人无完人。

19.1　忘记设置可执行权限

19.1.1　问题

你写好了脚本，想试试效果，但运行脚本时得到了下列错误信息。

```
$ ./my.script
bash: ./my.script: Permission denied
$
```

19.1.2　解决方案

选择有两个。你可以调用 bash 并指定脚本名称为参数。

```
bash my.script
```

或者（也是更好的）为脚本设置可执行权限，这样可以直接运行该脚本。

```
chmod a+x my.script
./my.script
```

19.1.3 讨论

这两种方法都能让脚本运行起来。如果打算多次运行脚本，可能更愿意设置脚本的可执行权限。只需要设置一次，以后就能直接调用了。设置过权限后，用起来感觉更像是命令，因为不用非得再明确地调用 bash 了（当然，背后还是得调用 bash，只是用不着通过键盘输入了）。

我们用 a+x 为所有用户设置了可执行权限。没什么理由限制文件的可执行权限，除非该可执行文件位于可能被其他用户意外访问到的目录（例如，作为系统管理员的你将自己的文件放进了 /usr/bin）中。另外，如果文件对所有用户都开放了读权限，那么只要其他用户采用第一种调用形式，仍旧可以执行脚本。在八进制模式中，小心谨慎的用户通常会将 shell 脚本的权限设置为 0700（仅对所有者本人开放读 / 写 / 执行权限）；心比较大的用户通常将权限设置为 0755（对其他用户开放读 / 执行权限）。

19.1.4 参考

- man chmod
- 14.13 节
- 15.1 节
- 19.3 节

19.2 修复“No such file or directory”错误

19.2.1 问题

你已经按照 19.1 节中描述的方法设置了可执行权限，但在运行脚本时，得到了“No such file or directory”错误。

19.2.2 解决方案

明确调用 bash 运行脚本。

```
bash ./busted
```

可行的话，那就是因为权限问题或者 shebang 行中哪个地方输错字了。如果产生更多错误信息，可能是行尾字符不对。如果是在 Windows（也许是通过 Samba）中编辑的文件，或者文件是从 Windows 那里复制过来的，就会出现这种情况。

可以对存疑的脚本文件执行 file 命令，它能够告诉你是否存在行尾错误。可能会出现下列信息：

```
$ file ./busted
./busted: Bourne-Again shell script, ASCII text executable, with
CRLF line
terminators
$
```

要想修复这个问题，可以尝试使用 dos2unix（如果已经安装过的话），或是参考 8.11 节。注意，如果选用 dos2unix，那么会创建一个新文件并删除旧文件，这将改变原文件的权限，可能还会改变属主或属组，并影响到硬链接。如果不确定这是什么意思，总结成一句话就是你可能还得对文件再使用 chmod（参见 19.1 节）。

19.2.3 讨论

如果确实存在行尾错误（ASCII 编码为 10 或 0x0A 之外的字符），具体的错误信息取决于所使用的 shebang 行。以下是脚本 busted 的几个示例。

```
$ cat busted
#!/bin/bash -
echo "Hello World!"

# 正常
$ ./busted
```

```
Hello World!

# 如果文件使用的是DOS风格的行尾[1]，则会得到：
$ ./busted
: invalid option
Usage: /bin/bash [GNU long option] [option] ...
[...]

# 不同的shebang行
$ cat ./busted
#!/usr/bin/env bash
echo "Hello World!"

$ ./busted
: No such file or directory
```

[1] 以“\r\n”作为行尾的这种形式见于 DOS/Windows 系统中。——译者注

19.2.4 参考

- 8.11 节
- 14.2 节
- 15.1 节
- 19.1 节

19.3 忘记当前目录不在$PATH中

19.3.1 问题

你写好了脚本，想试试效果，甚至还记得设置可执行权限，但运行时得到了一条错误信息。

```
$ my.script
bash: my.script: command not found
$
```

19.3.2 解决方案

要么将当前目录加入 $PATH 变量（我们不推荐这种方法），要么通过脚本名称前的当前目录 ./ 来引用该脚本。

```
./my.script
```

19.3.3 讨论

初学者常犯的一个错误就是忘记在要执行的脚本前添加 ./。我们对 $PATH 变量已经有过不少讨论，在此就不赘述了，只提醒你一个可用于常用脚本的解决方案。

一种常见做法是将你经常用到的实用脚本放进主目录的 bin 子目录中，然后将其加入 $PATH 变量，这样一来，执行当中的脚本时就不用添加 ./ 了。

将自己的 bin 目录加入 $PATH 变量时，重要的是要把对 $PATH 变量的改动放在正确的启动脚本中。不要放入 .bashrc 脚本，因为每个交互式子 shell 都会调用该脚本，这意味着，只要编辑器启动 shell 或运行某些别的命令，你的路径都会被添加。$PATH 变量中用不着这么多重复的 bin 目录。

应该将改动放在 bash 相应的登录配置文件中。根据 bash 手册页所述，当你登录时，bash 会“依次查找 ~/.bash_profile、~/.bash_login 和 ~/.profile，从现有的第一个可读文件中读取并执行命令”。因此，可以在主目录中找到上述文件之一并进行编辑，如果不存在，则创建 ~/.bash_profile，将下列语句放置在文件末尾（如果充分了解配置文件的其他用途，也可以将其放在其他位置）。

```
PATH="${PATH}:$HOME/bin"
```

19.3.4 参考

- 4.1 节
- 14.3 节
- 14.9 节
- 14.10 节

- 15.2 节
- 16.4 节
- 16.5 节
- 16.11 节
- 16.20 节
- 19.1 节

19.4 将脚本命名为test

19.4.1 问题

你编写了一个 bash 脚本，想测试一下刚看到的一些有意思的内容。代码完全没问题，一个字母都没错，也没忘记设置可执行权限并将脚本放入 $PATH 所列出的目录之中，但运行脚本时没有任何反应。

19.4.2 解决方案

只要不是 test，给脚本起什么名字都行。这和 shell 的一个内建命令重名了。

19.4.3 讨论

当你想写个短平快的脚本来测试代码时，很自然就会想将文件命名为 test。然而 test 是 shell 的内建命令，这个名称属于 shell 的**保留字**。使用 type 命令就能看出来。

```
$ type test
test is a shell builtin
$
```

因为是内建命令，所以无法改变其优先级。你可以创建别名 [2]，但对于本实例的这种情况，我们强烈建议不要这么做。可以将你的脚本改名，或者调用时加上路径：./test 或 /home/path/test。

[2]别名的优先级高于内建命令。——译者注

19.4.4 参考

- 19.1 节
- 19.3 节
- A.4 节
- A.5 节

19.5 试图修改已导出的变量

19.5.1 问题

你无法让子脚本或脚本将已导出的变量回传给其父 shell 或父脚本。例如，下列脚本会设置一个值，调用另一个脚本，并在第二个脚本运行完毕后显示该值，以查看变化（如果有的话）。

```
$ cat first.sh
#
# 演示一个常见的错误
#
# 设置值：
export VAL=5
printf "VAL=%d\n" $VAL
# 调用其他脚本：
./second.sh
#
# 现在看看有什么变化（提示：没有！）
printf "%b" "back in first\n"
printf "VAL=%d\n" $VAL
$
```

第二个脚本也对变量 $VAL 做了处理。

```
$ cat second.sh
printf "%b" "in second\n"
printf "initially VAL=%d\n" $VAL
VAL=12
printf "changed so VAL=%d\n" $VAL
$
```

运行第一个脚本（该脚本会调用第二个脚本）时，你会得到下列信息。

```
$ ./first.sh
VAL=5
in second
initially VAL=5
changed so VAL=10
back in first
VAL=5
$
```

19.5.2 解决方案

有个老笑话是这样的：

病人：“医生，我做这个动作的时候会疼。”

医生：“哦，那就别做。”

我们给出的解决方案和这位医生的建议一样：**那就别做**。你应该好好设计 shell 脚本，避免这种调用切换。一种方法是明确地输出第二个脚本的结果，以便第一个脚本可以用 `$()` 操作符（或者旧写法 ''）调用第二个脚本。在第一个脚本中，将 `./second.sh` 改成 `VAL=$(./second.sh)`，第二个脚本必须使用 `echo` 将最终结果（仅此而已）输出到 STDOUT（其他消息可以重定向到 STDERR）。

```
$ cat second.sh
printf "%b" "in second\n"           >&2
printf "initially VAL=%d\n" $VAL    >&2
VAL=12
printf "changed so VAL=%d\n" $VAL   >&2
echo $VAL
$
```

19.5.3 讨论

导出的环境变量**并非**能在脚本之间共享的全局变量。这是一种单向的通信形式。所有导出的环境变量会被当作 Linux 或 Unix（子）进程（参见 `fork(2)` 手册页）调用的组成部分而一并传递。这些环境变

量无法再传回父进程。（别忘了，父进程能够衍生出大量子进程……如果能从子进程返回值，那父进程得到的到底是哪个子进程的值？）

19.5.4 参考

- man fork(2)
- 5.5 节
- 10.4 节
- 10.5 节

19.6 赋值时忘记加引号

19.6.1 问题

你在脚本中为变量赋值，但运行脚本时，shell 在变量赋值部分报告“command not found”。

```
$ cat goof1.sh
#!/bin/bash -
# 常见错误：
#  X=$Y $Z
# 不同于：
#  X="$Y $Z"
#
OPT1=-l
OPT2=-h
ALLOPT=$OPT1 $OPT2
ls $ALLOPT .

$ ./goof1.sh
goof1.sh: line 9: -h: command not found
aaa.awk cdscript.prev ifexpr.sh oldsrc xspin2.sh
$
```

19.6.2 解决方案

你需要在 $ALLOPT 赋值的右侧加上引号。脚本中原先的写法如下所示。

```
ALLOPT=$OPT1 $OPT2
```

应该改为：

```
ALLOPT="$OPT1 $OPT2"
```

19.6.3 讨论

问题的根源在于参数之间的空格。如果参数之间由斜线分隔或者压根没有空格，那就不会出现这种问题，而是形成了一个单词 [3]，因而也就只是单一赋值。

> [3]单词（word）是被 shell 视为一个处理单元（unit）的字符序列。单词中不包括未经引用的元字符（metacharacter）。元字符可以是空白字符或者下列字符之一：'|'、'&'、';'、'('、')'、'<'、'>'。——译者注

但是横插进来的空格使得 bash 将其解析成了两个单词。第一个单词是变量赋值。这种位于命令起始位置上的赋值告诉 bash 仅在该命令执行期间将变量设为指定值，而命令行上紧随其后的单词就是要执行的命令。等到下一个命令行时，变量就恢复成先前的值（如果存在的话）或者为空。

ALLOPT=$OPT1 $OPT2 中的第二个单词 $OPT2 被视为命令，也就是那个被 shell 报告“command not found”的命令。当然，$OPT2 的值也可能正好是某个可执行文件的名称（本例并非如此）。这会导致很不理想的结果。

你有没有注意到，当例子中的 ls 运行时，输出的并不是长格式，尽管我们（尝试）设置了 -l 选项。这表明 $ALLOPT 已经无效了。其先前设置的值仅适用于上一个命令（并不存在的 -h 命令）执行期间。

单独为一行的赋值语句所设置的变量在脚本的剩余部分均有效。对于出现在行首（同一行中还有要调用的命令）的赋值语句，其所设置的变量仅在该命令执行期间有效。

对 shell 变量进行赋值时，给赋值号右侧的部分加上引号通常是不错的做法。这样可以确保只形成单个赋值语句，不会再碰到本例中出现的问题。

19.6.4 参考

- 5.9 节

19.7 忘记模式匹配的结果是按字母顺序排列的

19.7.1 问题

你想在模式匹配的字符类中指定字符顺序，但结果事与愿违。

19.7.2 解决方案

bash 会按字母顺序排列模式匹配的结果。

```
$ echo x.[ba]
x.a x.b
$
```

19.7.3 讨论

就算你在方括号中将 b 放在 a 之前，模式匹配的结果依然是按字母顺序排列的。这意味着，对于下列命令：

```
mv x.[ba]
```

你认为它会扩展为：

```
mv x.b x.a
```

但实际上会扩展为：

```
mv x.a x.b
```

bash 会将模式匹配的结果按照字母顺序排序，然后放置在命令行上，这和你想要的结果恰好相反！

但是，如果你用花括号来枚举不同的值，就可以保留指定的顺序。这正符合你的要求。

```
mv x.{b,a}
```

19.8 忘记管道会产生子shell

19.8.1 问题

你有一个功能正常的脚本，其使用 `while` 循环来读取输入。

```
# 一切正常
COUNT=0
while read ALINE
do
    let COUNT++
done
echo $COUNT
```

然后你按以下方式修改了脚本，从文件中读取输入，脚本的第一个参数指定了文件名。

```
# 别这么做。结果并不如你所愿！
COUNT=0
cat $1 | while read ALINE
do
    let COUNT++
done
echo $COUNT    # $COUNT的值总是0，完全没发挥作用
```

现在脚本无法正常工作了。`$COUNT` 的值为 `0`。

19.8.2 解决方案

管道会创建子 shell。while 循环中做出的改动不会影响外层的变量，因为和管道中的其他各命令一样，while 循环也在单独的子 shell 中运行。（cat 命令也在子 shell 中运行，但不修改 shell 变量。）

解决方案之一：不要这样做（如果可以的话）。也就是说，别用管道。这个示例没必要在管道中使用 cat 将文件内容传给 while 语句，你可以用 I/O 重定向来代替。

```
# 避免|和子shell，使用"done < $1"来代替
# 现在的结果就符合预期了
COUNT=0
while read ALINE
do
    let COUNT++
done < $1       # <<<< 这是关键区别所在
echo $COUNT
```

这种简单的重新排列未必能解决问题，这种情况下，你得使用其他技术。

从 bash 4 开始，你可以一早在脚本中设置 shell 选项 lastpipe 来避免此类问题。

```
shopt -s lastpipe
```

如果还不管用，或者你使用的 bash 版本小于 4.0，参见 19.8.3 节。

19.8.3 讨论

如果在示例脚本的 while 循环中添加 echo 语句，你会看到 $COUNT 的值在不断增加，一旦退出循环，$COUNT 的值就变回了 0。bash 设置命令管道的方式决定了管道中的各个命令都在单独的 shell 中运行。因此，while 循环其实是在子 shell 中运行，而非主 shell。在 while 循环开始时，$COUNT 的值和主 shell 脚本中的相同，但由于 while 循环在子 shell 中运行，故无法将修改后的 $COUNT 的值回传给父 shell。

解决方法是将所有的额外工作全部放到 while 循环所在的子 shell 中来完成。例如：

```
COUNT=0
cat $1 | { while read ALINE
do
  let COUNT++
done
echo $COUNT ; }      # 这里的空格很重要
```

花括号的位置至关重要。我们所做的就是明确地将部分脚本放在相同的（子）shell 中运行，其中包括 while 循环和要在循环结束后完成的其他工作（这里是显示 $COUNT 的值）。因为 while 和 echo 并没有通过管道相连，所以两者均在由花括号界定的同一子 shell 中运行。$COUNT 会在 while 循环期间累加，变量值会一直保留到子 shell 结束，也就是闭合花括号的位置。

如果选用这种技术，最好将语句重新格式化一下，让花括号部分的子 shell 更显眼些。以下是重新格式化后的脚本。

```
COUNT=0
cat $1 |
{
    while read ALINE
    do
        let COUNT++
    done
    echo $COUNT
}
```

如果使用的是 bash 4，则可以完全避免这个问题。在脚本中设置 shell 选项 lastpipe 即可（有些系统管理员甚至想将其放在 /etc/profile 或相关的 .rc 文件中进行设置，这样其他用户就不用再操心了）。

```
shopt -s lastpipe
```

该选项告诉 bash 在当前 shell 环境中执行管道中的最后一个命令，而不是在单独的子 shell 中，这样一来，管道之后的 shell 脚本也可以使用其中的变量。

以下示例和前面的示例类似，只不过它没有使用 cat，而是改用 ls 作为数据源。

```
shopt -s lastpipe   # bash 4及之后版本可用
COUNT=0
ls | while read ALINE
do
  let COUNT++
done
echo $COUNT
```

试试使用 shopt 和不使用 shopt 有什么区别。

> lastpipe 仅适用于禁用作业控制的情况，这也是非交互式 shell（bash 脚本）的默认条件。如果想在交互式 shell 中使用 lastpipe，需要使用 set +m 来禁用作业控制，但这样做的话，就无法中断（^C）或挂起（^Z）正在运行的命令了，也不能再使用 fg 和 bg 命令。我们不建议这样做。

19.8.4 参考

- bash FAQ 4.14 版的问题 E4
- 10.5 节
- 19.5 节

19.9 使终端恢复正常

19.9.1 问题

中止 SSH 会话后，你发现自己看不到键盘输入的内容了。或是你不小心显示了一个二进制文件，结果搞得终端窗口中全是乱七八糟的字符。

19.9.2 解决方案

哪怕你看不到输入的内容，也可以约莫着敲入 stty sane 并按下 Enter 键来恢复正常的终端设置。你可以先多按几次 Enter 键，以确保开始输入 stty 命令时，命令行上没有残留别的内容。

如果经常要这么做，不妨考虑创建一个便于盲打的别名（参见 10.7 节）。

19.9.3 讨论

如果在提示输入密码时中止，有些老版本的 ssh 有可能会关闭终端的回显功能（显示从键盘上输入的字符，不是指 shell 的 echo 命令），导致用户看不到任何输入。根据使用的终端仿真器显示二进制文件也可能会意外地改变终端设置。不管是哪种情况，stty sane 会尝试将所有的终端设置还原成默认值，其中包括恢复回显功能，这样一来，键盘输入的内容就能出现在终端窗口中了。这也许还能消除其他终端设置中出现的奇怪现象。

你使用的终端应用可能也具有某种重置功能，不妨研究一下菜单选项和文档。你也可以尝试一下 reset 和 tset 命令，stty sane 在我们的测试中能够按照预期工作，而 reset 和 tset 的修复过程更彻底。

19.9.4 参考

- man reset
- man stty
- man tset
- 10.7 节

19.10 用空变量删除文件

19.10.1 问题

你认为某个已有变量中保存的是待删除的文件列表，可能要在脚本运行后做清理之用。但事实上，这是一个空变量，要坏事的。

19.10.2 解决方案

不要做：

```
rm -rf $files_to_delete
```

千万不要做：

```
rm -rf /$files_to_delete
```

应该这样做：

```
[ -n "$files_to_delete" ] && rm -rf $files_to_delete
```

19.10.3 讨论

第一个例子不算太糟，也就是报错而已。第二个例子糟糕透了，因为该命令会删除你的根目录。如果你的身份是普通用户（这是应该的，参见 14.18 节），也许还不至于收不住场，但如果身份是 root，那你的系统就算是彻底毁了。（是的，我们这么干过。）

解决办法很简单。首先，确保用到的变量中有值；其次，绝不要在变量前面添加 /。

19.10.4 参考

- 14.18 节
- 18.7 节

19.11 printf的怪异行为

19.11.1 问题

脚本给出的结果和预想中的不一样。思考下面这个简单的脚本及其输出。

```
$ bash oddscript
good nodes: 0
bad nodes: 6
miss nodes: 0
GOOD=6 BAD=0 MISS=0

$ cat oddscript
#!/bin/bash -
badnode=6

printf "good nodes: %d\n" $goodnode
printf "bad nodes: %d\n" $badnode
printf "miss nodes: %d\n" $missnode
printf "GOOD=%d BAD=%d MISS=%d\n" $goodnode $badnode $missnode
```

为什么 good 的数量是 6，这个数字应该是 bad 的数量啊？

19.11.2 解决方案

给变量设置初始值（例如，0）或者在 printf 语句中将变量放进引号。

19.11.3 讨论

到底怎么回事？bash 在最后一行进行了变量替换，对 $goodnode 和 $missnode 求值时，两者皆为空。因此，printf 得到的替换后的结果如下所示：

```
printf "GOOD=%d BAD=%d MISS=%d\n" 6
```

printf 尝试输出 3 个十进制值（因为有 3 个 %d 格式说明符），而现在只有一个值（6），还缺两个，故将后两者视为 0，因此得到以下结果。

```
GOOD=6 BAD=0 MISS=0
```

这真不能怪 printf，它压根就没见过另外两个参数。bash 在 printf 执行前就完成了替换。

即便将变量声明为整数类型，如下所示：

```
declare -i goodnode badnode missnode
```

这也挽救不了局面。你得实实在在地给变量赋值。

避免该问题的另一种方法是将 printf 语句中的参数放入引号。

```
printf "GOOD=%d BAD=%d MISS=%d\n" "$goodnode" "$badnode"
"$missnode"
```

这样一来，第一个参数就不会消失了，虽然只是一个占据了相应位置的空串，但 printf 因此获得了所需要的 3 个参数。

```
printf "GOOD=%d BAD=%d MISS=%d\n" "" "6" ""
```

printf 还有另一种怪异行为。我们已经见识了参数过少时的情形；如果参数过多，printf 会重复使用格式说明符，当你以为只会输出一行时，实际却得到了多行输出。

当然，这种行为也可以善加利用，如下所示：

```
$ dirs
/usr/bin /tmp ~/scratch/misc
$ printf "%s\n" $(dirs)
/usr/bin
/tmp
~/scratch/misc
$
```

这里 printf 以目录栈（dirs 命令的输出）为参数，重复使用格式说明符，一行一个地显示出其中的各个目录。

我们总结了一些最佳实践。

- 初始化变量，尤其变量值是数字并想要用于 printf 语句时。
- 如果变量可能为空，尤其要作为参数用于 printf 语句时，记得加上引号。
- 确保参数个数正确，尤其要考虑到 shell 替换后的结果。
- 显示字符串的最安全的方法是使用 printf '%s\n' "$_string_"。

19.11.4 参考

- 2.3 节
- 2.4 节
- 15.6 节
- A.12 节

19.12 测试bash脚本语法

19.12.1 问题

你正在编辑 bash 脚本，希望确保语法正确。

19.12.2 解决方案

通常用 bash 的 -n 选项测试语法，最好是在每次保存后，并且一定要在向版本控制系统提交变更前进行测试。

```
$ bash -n my_script

$ echo 'echo "Broken line' >> my_script

$ bash -n my_script
my_script: line 4: unexpected EOF while looking for matching `"'
my_script: line 5: syntax error: unexpected end of file
```

19.12.3 讨论

要想在 bash 手册页或其他参考资料中查找 -n 选项的相关信息，可能得费点周折，因为它位于内建命令 set 名下。bash --help 的 -D 选项中有所提及，但并未解释。-n 选项告诉 bash“读取命令但不执行”，在这个过程中自然能发现 bash 的语法错误。

和所有的语法检查器一样，该方法也无法找出逻辑错误或脚本所调用的其他命令的语法错误。

19.12.4 参考

- man bash
- bash --help
- bash -c "help set"
- 16.1 节

19.13 调试脚本

19.13.1 问题

你搞不清楚脚本到底为什么无法按照预期工作。

19.13.2 解决方案

在脚本顶部添加 set -x，或者在有问题的位置上使用 set -x 来启用 xtrace，随后再用 set +x 关闭 xtrace。你可能还想试试 $PS4 提示符（参见 16.2 节）。xtrace 也可以用于交互式命令行。我们怀疑例 19-1 中的脚本存在 bug。

例 19-1 ch19/buggy

```
#!/usr/bin/env bash
# 实例文件：-buggy
#

set -x

result=$1

[ $result = 1 ] \
  && { echo "Result is 1; excellent." ; exit 0;   } \
  || { echo "Uh-oh, ummm, RUN AWAY! " ; exit 120; }
```

在调用脚本前，我们先设置并导出 $PS4 提示符。在跟踪执行期间（也就是 set -x 之后），bash 会在每个命令前面输出 $PS4 的值。

```
$ export PS4='+xtrace $LINENO:'

$ echo $PS4
+xtrace $LINENO:

$ ./buggy
+xtrace 4: result=
+xtrace 6: '[' = 1 ']'
./buggy: line 6: [: =: unary operator expected
+xtrace 8: echo 'Uh-oh, ummm, RUN AWAY! '
Uh-oh, ummm, RUN AWAY!

$ ./buggy 1
+xtrace 4: result=1
+xtrace 6: '[' 1 = 1 ']'
+xtrace 7: echo 'Result is 1; excellent.'
Result is 1; excellent.

$ ./buggy 2
+xtrace 4: result=2
+xtrace 6: '[' 2 = 1 ']'
+xtrace 8: echo 'Uh-oh, ummm, RUN AWAY! '
Uh-oh, ummm, RUN AWAY!

$ /tmp/jp-test.sh 3
+xtrace 4: result=3
+xtrace 6: '[' 3 = 1 ']'
+xtrace 8: echo 'Uh-oh, ummm, RUN AWAY! '
Uh-oh, ummm, RUN AWAY!
```

19.13.3 讨论

用 - 开启功能，用 + 关闭功能，这看起来可能有些怪异，但就是这么用的。很多 Unix 工具使用 -n 这种形式来表示启用选项或标志，因此，要关闭选项的话，使用 +x 是一种自然而然的选择。

从 bash 3.0 开始，许多新变量可以更好地支持调试：$BASH_ARGC、$BASH_ARGV、$BASH_SOURCE、$BASH_LINENO、$BASH_SUBSHELL、$BASH_EXE CUTION_STRING、$BASH_COMMAND。另外还有一个新的 shell 选项 extdebug。这些是对 $LINENO 等现有 bash 变量和数组变量 $FUNCNAME 的补充。

以下内容摘自 Bash Reference Manual。

如果调用 shell 时设置了 extdebug，那么会在 shell 启动前执行调试器配置文件，这等同于 --debugger 选项。如果在调用后设置，则下列行为可供调试器使用。

- 内建命令 declare 的 -F 选项……显示了作为参数的每个函数所在的源文件名和行号。
- 如果 DEBUG 陷阱所运行的命令返回非 0 值，则跳过下一个命令。
- 如果 DEBUG 陷阱所运行的命令返回 2，并且 shell 运行于子例程（shell 函数或通过 ./source 所执行的 shell 脚本），则 shell 模拟 return 调用。
- 更新 BASH_ARGC 和 BASH_ARGV……
- 启用函数跟踪：命令替换、shell 函数以及用（ *command* ）调用的子 shell 继承 DEBUG 和 RETURN 陷阱。
- 启用错误跟踪：命令替换、shell 函数以及用（ *command* ）调用的子 shell 继承 ERR 陷阱。

xtrace 是一种非常方便的调试技术，但不能将其等同于真正的调试器。不妨了解一下 Bash Debugger Project，其中包含打过补丁的 bash 源代码，这些代码提供了更好的调试支持，改善了错误报告。除此之外，用开发人员的话来讲，该项目还包含了“迄今为止最全面的 BASH 源代码级调试器”。

19.13.4 参考

- help set
- man bash
- Cameron Newham 所著的 *Learning the bash Shell, 3rd Edition*（O'Reilly 出版）的第 9 章，其中包含了用于调试其他 shell 脚本的 shell 脚本
- 16.1 节
- 16.2 节
- 17.1 节

19.14 使用函数时避免出现“command not found”错误

19.14.1 问题

你已经习惯了 Perl 这种允许在函数实际定义前调用函数的语言。

19.14.2 解决方案

shell 脚本是按照自上而下的线性方式读取并执行的，因此必须在使用函数前先定义函数。

19.14.3 讨论

有些其他语言（如 Perl）会经历一些中间步骤，在此过程中，整个脚本将被解析为一个单元。这允许你在编写代码时将 `main()` 置于开头，将函数（或子例程）定义放在随后。相比之下，shell 脚本是被读入内存，并一次一行执行的，因此函数不能先使用再定义。

19.14.4 参考

- 10.4 节
- 10.5 节
- 附录 C

19.15 混淆了shell通配符和正则表达式

19.15.1 问题

你有时候会看到 `.*`，有时候会看到 `*`，还有时候会看到 `[a-z]*`，但它的含义并非你所想的那样。`grep` 和 `sed` 中要使用正则表达

式，bash 中的某些地方却不使用正则表达式。你始终无法将这些理得清清楚楚。

19.15.2 解决方案

放松，深呼吸。可能学得内容太多，把你给弄晕了（或者用得太少，记不起来了）。实践出真知，要坚持练习。

就 bash 本身而言，规则不难记忆。在 bash 中，正则表达式语法仅用于 =~ 运算符。其他表达式使用的均是 shell 模式匹配。

19.15.3 讨论

bash 所用的模式匹配用到了一些和正则表达式相同的符号，但意义完全不同。但 shell 脚本中经常会调用一些使用正则表达式的命令，如 grep 和 sed。

我们为此请教了 bash 源代码的现任维护者以及无所不能的 bash 大师 Chet Ramey，以确定 bash 中唯一用到正则表达式的地方是否就是 =~ 运算符。他非常友善地提供了一份使用了 shell 模式匹配的 bash 语法清单。本书的各个实例已经涵盖了其中的绝大部分。我们在此给出这份完整的清单。

执行 shell 模式匹配的操作包括：

- 文件名通配符匹配（路径名扩展）
- [[的 == 和 != 运算符
- case 语句
- $GLOBIGNORE 的处理
- $HISTIGNORE 的处理
- ${*parameter*#[#]*word*}
- ${*parameter*%[%]*word*}
- ${*parameter*/*pattern*/*string*}
- 一些可绑定的 readline 命令（glob-expand-word、glob-complete-word 等）
- complete -G 和 compgen -G

- complete -X 和 compgen -X
- 内建命令 help 的 pattern 参数

多谢，Chet ！

学习阅读 bash 手册页，多参考其中的内容，虽然篇幅有些长，但胜在准确。如果你想在线阅读 bash 手册页或其他 bash 相关的文档，可以访问本书（英文版）网站来了解最新的 bash 资讯。另外别忘了把本书常备在手边，以作参考。

19.15.4 参考

- man bash
- 5.18 节
- 6.6 节
- 6.7 节
- 6.8 节
- 13.15 节

附录 A　参考

本附录将多种表格汇集一处，以便参考，其中涵盖了取值、设置、运算符、命令、变量等方面的内容。

A.1　bash调用

以下是调用当前版本 bash 时可用的一些选项。在命令行上，多字符选项必须出现在单字符选项之前。登录 shell 通常在内部设置了选项 -i（交互式）、-s（从标准输入读取）以及 -m（启用作业控制）。

除了表 A-1 中列出的这些，所有的 set 选项都可以在命令行上使用（参见 A.7 节）。特别值得一提的是，-n 选项对于语法检查非常有用（参见 19.12 节），-x 可用于调试（参见 19.13 节）。

表A-1：bash的命令行选项

选项	含义
-c *string*	从 *string*（如果存在的话）读取命令。*string* 之后的任何参数都会被视为位置参数，以 $0 开始
-D	将所有以 $ 起始的双引号引用字符串写入标准输出。[1] 如果当前语言环境不是 C 或 POSIX，这些字符串会进行语言转换。该选项同时还会启用 -n 选项
-i	将 shell 标记为交互式 shell。忽略信号 TERM、INT、QUIT。如果作业控制有效，一并忽略 TTIN、TTOU、TSTP
-l	使 bash 成为登录 shell

选项	含义
-o *option*	接受和 set -o（参见 A.7 节）相同的参数
-O、+O *shopt-option*	*shopt-option* 是内建命令 *shopt* 能够接受的 shell 选项之一。如果指定 *shopt-option*，-O 可以设置该选项值；+O 可以取消该选项。如果未指定 *shopt-option*，则将 *shopt* 能够接受的选项名称及其取值写入标准输出。如果调用选项是 +O，所显示的输出格式还可以作为输入使用
-s	从标准输入中读取命令。如果指定了 bash 参数，则此标志优先（也就是说，该参数不会被视为脚本名，依然从标准输入读取）
-r	使 bash 成为受限 shell
-v	读取时输出 shell 的输入行
-	表示选项已结束，停止选项处理。随后出现的任何选项均视为文件名和参数。-- 等同于 -
--debugger	在 shell 启动前执行调试器的配置文件。bash 3.0 及以上版本还会启用扩展调试模式和 shell 函数跟踪功能
--dump-strings	和 -D 的行为一样
--dump-po-strings	和 -D 的行为一样，但输出的是 GNU gettext 可移植对象（.po）文件格式
--help	显示用法信息并退出
--login	使 bash 成为登录 shell。等同于 -l

选项	含义
--noediting	如果是交互式 shell，则不使用 GNU readline 库来读取命令行
--noprofile	不读取启动文件 /etc/profile 或任何个人初始化文件
--norc	如果是交互式 shell，则不读取初始化文件 ~/.bashrc；如果以 sh 调用 shell，则默认启用该选项
--posix	改变 bash 的行为，使之更符合 POSIX 标准，这种情况下，bash 的默认操作会有所不同
--rcfile *file*、--init-file *file*	如果是交互式 shell，则从 *file* 读取并执行命令，而非初始化文件 ~/.bashrc
--restricted	等同于 -r
--verbose	等同于 -v
--version	显示 bash 当前实例的版本号并退出

[1]也就是 $"..." 这种形式。——译者注

A.2 自定义提示符字符串

表 A-2 展示了可用的提示符自定义格式代码，其中，\[和 \] 在 bash 1.14 之前不可用，\a、\e、\H、\T、\@、\v、\V 在 bash

2.0 之前不可用。\A、\D、\j、\l、\r 仅可用于 bash 2.0 之后以及 bash 3.0+。

表A-2：提示符字符串格式代码

代码	含义
\a	ASCII 响铃字符（007）
\A	24 小时制（HH:MM）的当前时间
\d	采用“星期几 月份 天”格式的日期
\D{*format*}	*format* 被传给 strftime(3) 并将结果插入提示符字符串。如果 *format* 为空，则生成特定语言环境的时间表示。花括号是必需的
\e	ASCII 转义字符（033）
\H	主机名
\h	截止到第一个. 处的主机名
\j	shell 当前所管理的作业数量
\l	shell 的终端设备的基本名称
\n	回车（carriage return）和换行（line feed）
\r	回车

代码	含义
\s	shell 的名称
\T	12 小时制（HH:MM:SS）的当前时间
\t	24 小时制（HH:MM:SS）的当前时间
\@	12 小时制（a.m/p.m）的当前时间
\u	当前用户名
\v	bash 的版本号（如 2.00）
\V	bash 的发行版本号（版本号和补丁号，如 3.00.0）
\w	当前工作目录，$HOME 的缩写为波浪号（使用 $PROMPT_DIRTRIM 变量）
\W	$PWD 的基本命令，$HOME 的缩写为波浪号
\#	当前命令的命令编号
\!	当前命令的历史记录编号
\$	如果有效UID 是0，显示为 #；否则，显示为 $
nnn	八进制字符编码

代码	含义
\\	反斜线
\[	不可打印字符序列（例如，终端控制序列）的起始标记
\]	不可打印字符序列的结束标记

A.3 ANSI颜色转义序列

表 A-3 展示了 ANSI 颜色转义序列。

表A-3：ANSI颜色转义序列

代码	字符属性	FG代码	前景色	BG代码	背景色
0	重置所有属性	30	黑色	40	黑色
1	明亮	31	红色	41	红色
2	昏暗	32	绿色	42	绿色
4	下划线	33	黄色	43	黄色
5	闪烁	34	蓝色	44	蓝色
7	反显 [2]	35	洋红	45	洋红
8	隐藏	36	青色	46	青色

代码	字符属性	FG代码	前景色	BG代码	背景色
		37	白色	47	白色

[2]前景色与背景色交换。——译者注

A.4 内建命令

表 A-4 列出了 bash 当前版中的内建命令。

表A-4：内建命令

命令	总结
.	读取文件并在当前 shell 环境中执行文件内容。参见 source
:	什么都不做（只执行参数扩展）
[	对条件表达式求值。参见 test
alias	设置命令或命令行的简写
bg	将作业置入后台
bind	将按键序列与 readline 函数或宏绑定
break	退出所在的 for、select、while 或 until 循环
builtin	执行特定的 shell 内建命令

命令	总结
caller	返回活跃的子例程调用（shell 函数或使用内建命令 ./source 执行的脚本）的上下文
cd	更改工作目录
command	绕过 shell 函数查找来执行命令
compgen	生成可能的补全匹配
complete	指定如何执行补全
compopt	根据选项修改每个名称的补全方式，如果没有指定名称，则修改当前正在使用的补全
continue	直接跳到所在的 for、select、while 或 until 循环的下一次迭代
declare	声明变量并赋予属性。等同于 typeset
dirs	显示当前已记录的目录列表
disown	从作业列表中移除作业
echo	输出参数
enable	启用或禁用（使用 -n）shell 内建命令
eval	按照命令行处理流程运行指定的参数

命令	总结
exec	用指定的程序替代 shell
exit	退出 shell
export	创建环境变量
fc	修正命令（编辑命令历史记录文件）
fg	在前台终止后台作业
getopts	处理命令行参数
hash	记录特定命令的完整路径
help	显示内建命令的帮助信息
history	显示命令历史
jobs	列出所有的后台作业
kill	向进程发出信号
let	算术变量赋值
local	创建局部变量
logout	退出登录 shell

命令	总结
mapfile	从标准输入或文件描述符中将行读取到索引数组变量 array 中。参见 readarray
popd	从目录栈中删除目录
printf	依照指定的格式说明符将格式化后的参数写入标准输出
pushd	将目录压入目录栈
pwd	输出工作目录
read	从标准输入读取一行
readarray	从标准输入或文件描述符中将行读取到索引数组变量 array 中。参见 mapfile
readonly	将变量设置为只读（不能为其赋值）
return	从所在的函数或脚本中返回
set	设置选项
shift	移动命令行参数
shopt	设置 shell 选项
source	读取文件并在当前 shell 环境中执行文件内容。参见 .（点号）

命令	总结
suspend	挂起当前 shell
test	对条件表达式求值。参见 [
times	输出进程运行的累计用户时间和系统时间
trap	设置信号捕获例程
type	识别命令的来源
typeset	声明变量并赋予属性。等同于 declare
ulimit	设置/显示进程的资源限制
umask	设置/显示文件的权限掩码
unalias	删除别名定义
unset	删除变量或函数定义
wait	等待后台作业结束

A.5 bash保留字

表 A-5 列出了 bash 当前版本中的保留字。

表A-5：bash保留字

保留字	总结
!	对命令的退出状态取反（逻辑非）
[[ ]]	根据条件表达式的求值结果返回 0 或 1
(( ))	根据 bash shell 的算术规则对算术表达式进行求值
()	在子 shell 中执行命令列表
{}	在当前 shell 环境中执行命令列表
case	多路条件分支语句
do	for、select、while 或 until 循环的组成部分
done	for、select、while 或 until 循环的组成部分
elif	if 语句的组成部分
else	if 语句的组成部分
esac	case 语句的结束部分
fi	if 语句的结束部分
for	循环语句
function	定义函数

保留字	总结
if	条件语句
in	case 语句的组成部分
select	菜单生成语句
then	if 语句的组成部分
time	运行命令管道并输出执行时长。输出格式可通过 TIMEFORMAT 变量控制
until	循环语句
while	循环语句

A.6 shell内建变量

表 A-6 展示了 bash 4.4 可用的所有环境变量。表格中的“类型”一列中的字母含义如下：A= 数组、L= 以冒号分隔的列表、R= 只读、U= 取消设置会令其失去特殊意义。

注意，以 BASH_ 或 COMP 开头的变量以及 DIRSTACK、FUNCNAME、GLOBIGNORE、GROUPS、HISTIGNORE、HOSTNAME、HISTTIMEFORMAT、LANG、LC_ALL、LC_COLLATE、LC_MESSAGE、MACHTYPE、PIPESTATUS、SHELLOPTS、TIMEFORMAT 这些变量在 bash 2.0 之前不可用。BASH_ENV 取代了早期版本中的 ENV。

表A-6：shell内建环境变量

变量	类型	描述
*	R	当前脚本或函数的位置参数。如果没有使用双引号引用，每个单词会被进一步分割和扩展。[3] 如果使用了双引号引用，则返回单个字符串，其中各个参数之间以 $IFS 的第一个字符分隔（例如，"arg1 arg2 arg3"），$IFS 为空的话，就不使用分隔符
@	R	当前脚本或函数的位置参数，以双引号引用的字符串列表形式出现（例如，"arg1" "arg2" "arg3"）[4]
#	R	当前脚本或函数的参数个数
?	R	上一个命令的退出状态
-	R	调用 shell 时指定的选项
$	R	shell 的进程 ID
!	R	上一个后台进程的进程 ID
0	R	shell 或者 shell 脚本的名称
_	R	上一个命令的最后一个参数
auto_resume		控制作业管理的工作方式（取值可以是 exact、substring 等）
BASH		用于调用该 bash 实例的完整路径

变量	类型	描述
BASHOPTS	L	已启用的 shell 选项列表，以冒号分隔
BASHPID		当前 bash 进程的进程 ID
BASH_ALIASES	A	关联数组变量，其元素对应于内建命令 alias 定义的别名
BASH_ARGC	A	数组变量，其元素对应于当前 bash 执行调用栈中各帧的参数数量。位于栈顶的是当前子例程（shell 函数或用内建命令 ./source 执行的脚本）的参数数量
BASH_ARGV	A	数组变量，包含当前 bash 执行调用栈中所有的参数。最后一次子例程调用的最后一个参数位于栈顶；初始调用的第一个参数位于栈底
BASH_CMDS		关联数组变量，其元素对应于内建命令 hash 所维护的命令内部散列表
BASH_COMMAND		当前正在执行或要执行的命令，除非 shell 所执行的命令是由陷阱所导致的，这种情况下，指的则是陷入时执行的命令
BASH_ENV		调用 shell 时所使用的环境文件名
BASH_EXECUTION_STRING		调用选项 -c 的命令行参数
BASH_LINENO	A	数组变量，其元素是源文件的行号，分别对应 @var{FUNCNAME} 中的元素。$\{BASH_LINENO[$i]} 是 ${FUNCNAME[$i + 1]} 被调用时所在源文件的行号。对应的源文件名是 ${BASHSOURCE[$i + 1]}

变量	类型	描述
BASH_LOADABLES_PATH	L	以冒号分隔的目录列表，shell 会在其中查找由 enable 命令指定的动态可装载内建命令
BASH_REMATCH	AR	数组变量，由条件命令 [[的 =~ 运算符为其元素赋值。索引为 0 的元素是整个正则表达式所匹配到的字符串部分。索引为 *n* 的元素是第 *n* 个带有括号的子正则表达式所匹配到的字符串部分
BASH_SOURCE	A	数组变量，其元素是源文件名，对应 $FUNCNAME 数组变量中的元素
BASH_SUBSHELL		每生成一个子 shell，该变量值就增加 1。初始值为 0。子 shell 是父 shell 的衍生副本，并共享父 shell 的环境
BASH_VERSINFO	AR	bash 实例的版本信息。该数组的每个元素都包含版本号的一部分
BASH_VERSION		该 bash 实例的版本信息
BASH_XTRACEFD		如果设置为对应有效文件描述符的整数，bash 会将启用 set -x 时所产生的跟踪信息输出到该文件描述符
CDPATH	L	cd 命令要搜索的目录列表
CHILD_MAX		shell 能够记忆的状态为退出的子进程的最大数量
COLUMNS		select 命令用其在输出选择列表时确定终端宽度

变量	类型	描述
COMP_CWORD		指向 ${COMPWORDS} 的索引，包含当前光标所在的位置。该变量仅用于可编程补全功能所调用的 shell 函数中
COMP_LINE		当前命令行。该变量仅用于可编程补全功能所调用的 shell 函数和外部命令中
COMP_POINT		相对于当前命令起始位置的当前光标位置的索引。如果位于命令末尾，其值等于 ${#COMPLINE}。该变量仅用于可编程补全功能所调用的 shell 函数和外部命令中
COMP_WORDS	A	当前命令行中单独单词的数组，该变量仅用于可编程补全功能所调用的 shell 函数和外部命令中
COMPREPLY	A	可编程补全机制调用 shell 函数时产生的可能补全
COMPREPLY	A	数组变量，bash 从中读取可编程补全功能所调用的 shell 函数产生的可能补全
COMP_KEY		用于调用当前补全函数的按键（或按键序列的最后一个键）
COMP_TYPE		所尝试的补全类型对应的整数值，前者会导致补全函数被调用
COPROC		数组变量，其中保存着用于无名协程输出及输入的文件描述符
DIRSTACK	ARU	目录栈的当前内容

变量	类型	描述
echo_control-characters		如果设置为 on，在支持该变量的操作系统上，当从键盘产生信号时，readline 会相应地回显一个字符
editing-mode		控制默认使用哪种按键绑定集
EMACS		如果 bash 在启动时发现该变量的值以 t 开头，则假定 shell 使用的是 Emacs 缓冲并禁用行编辑
ENV		与 BASH_ENV 相似；在 POSIX 模式中调用 shell 时使用
EUID	R	当前用户的有效用户 ID
EXECIGNORE	L	以冒号分隔的 shell 模式列表，它定义了用 $PATH 搜索命令时要忽略的一系列文件名
FCEDIT		fc 命令默认的编辑器
FIGNORE	L	执行文件名补全时要忽略的名称列表
FUNCNAME	ARU	数组变量，其中包含当前在执行调用栈中的所有函数的名称。索引为 0 的元素是当前正在执行的 shell 函数的名称。最后一个元素是“主函数”。该变量仅存在于 shell 函数执行时
FUNCNEST		如果该变量设置的值大于 0，则定义了函数最大的嵌套层级

变量	类型	描述
GLOBIGNORE	L	模式列表，它定义了在路径扩展期间要忽略的文件名称
GROUPS	AR	数组变量，其中包含当前用户所属的组
histchars		指定了命令历史控制字符。通常设置为字符串 !^#
HISTCMD	U	当前命令的历史记录编号
HISTCONTROL	L	以冒号分隔的模式列表，可取值如下所示：ignorespace:（以空格开头的命令行不记入历史记录列表）、ignoredups:（和上一个历史记录条目相同的命令行不记入历史记录列表）、erasedups:（在写入当前命令行前，先删除历史记录列表中与其相同的所有条目）、ignoreboth:（同时启用 ignorespace 和 ignoredups）
HISTFILE		命令历史记录文件名称
HISTFILESIZE		历史记录文件（history file）能保存的最大命令数量
HISTIGNORE		模式列表，用于决定哪些命令行可以保留在历史记录列表中
HISTSIZE		能写入历史记录列表（history list）中的最大命令数量

变量	类型	描述
HISTTIMEFORMAT		如果设置了该变量，时间戳会写入历史记录文件，从而得以跨 shell 会话保留。变量值会作为 strftime(3) 的格式字符串，与内建命令 history 显示的历史记录条目一并输出
HOME		主（登录）目录
HOSTFILE		用于主机名补全的文件
HOSTNAME		当前主机的名称
HOSTTYPE		bash 所在的机器类型
IFS		内部字段分隔符：作为单词分隔符的一系列字符。通常设置为空格、制表符、换行符
IGNOREEOF		退出交互式 shell 前能够接收的 EOF 字符数量
INPUTRC		readline 的启动文件
LANG		用于确定那些未使用 LC_ 开头的变量专门设置的类别的语言环境
LC_ALL		覆盖 LANG 的值以及其他指定语言环境类别的 LC_ 变量
LC_COLLATE		确定为路径扩展结果排序时所使用的排序规则

变量	类型	描述
LC_CTYPE		在路径扩展和模式匹配中决定字符的含义和字符类的行为
LC_MESSAGES		确定用于转换以 $ 起始的双引号字符串 [6] 的语言环境
LC_NUMERIC		确定用于数字格式化的语言环境
LC_TIME		确定用于日期和时间格式化的语言环境
LINENO	U	脚本或函数的当前行号
LINES		select 命令用其确定输出选择列表时的列长
MACHTYPE		描述 bash 所在系统的字符串
MAIL		检查新邮件的文件名称
MAILCHECK		多久（以秒为单位）检查一次新邮件
MAILPATH	L	用于检查新邮件的文件名列表（如果 MAIL 未设置）
MAPFILE	A	数组变量，其中保存着内建命令 mapfile 所读取的文本
mark-directories		如果设置为 on，则完整的目录名会追加一条斜线

变量	类型	描述
OLDPWD		上一个工作目录
OPTARG		getopts 处理的最后一个选项的参数值
OPTERR		如果设置为 1，显示 getopts 产生的错误信息
OPTIND		getopts 处理的最后一个选项的参数索引值
OSTYPE		bash 所在的操作系统
PATH	L	命令的搜索路径
PIPESTATUS	A	数组变量，其中包含最近执行的前台管道中的各个进程的退出状态
POSIXLY_CORRECT		如果是在 bash 启动环境中设置的，shell 会在读取启动文件前进入 POSIX 模式，效果等同于调用 bash 时指定 --posix 选项。如果是在 shell 运行时设置的，bash 会启用 POSIX 模式，效果等同于执行 set -o posix 命令
PPID	R	shell 的父进程 PID
PROMPT_COMMAND		该变量的值作为显示主提示符前要执行的命令
PROMPT_DIRTRIM		如果取值大于 0，在扩展提示符字符串转义 \w 和 \W 时，则该值会作为要保留的结尾目录个数

变量	类型	描述
PS0		如果设置的话，在读入命令后、执行命令前，该字符串会在交互式 shell 中显示。仅适用于 bash 4.4 及后续版本
PS1		主命令提示字符串
PS2		续行提示字符串
PS3		select 命令提示字符串
PS4		xtrace 选项提示字符串
PPID	R	父进程的 PID
PWD		内建命令 cd 设置的当前工作目录
RANDOM	U	引用该变量时会产生一个位于 0~32 767 的伪随机整数
READLINE_LINE		readline 行缓冲区的内容，与 bind -x 一起使用
READLINE_POINT		readline 行缓冲区中的插入点位置，与 bind -x 一起使用
REPLY		select命令的用户回应；如果未指定变量，则该变量也保存 read 命令的读取结果
SECONDS	U	自启动 shell 以来的秒数

变量	类型	描述
SHELL		shell 的完整路径
SHELLOPTS	LR	已启用的 shell 选项，以冒号分隔
SHLVL		每生成一个新的 bash 实例（非子 shell），该变量值就增加 1，以表明 bash shell 的嵌套深度
TEXTDOMAIN		指明消息类别文件的位置（如果未使用 LC_MESSAGE）
TEXTDOMAINDIR		指明消息类别文件的位置（如果使用 TEXTDOMAIN）
TIMEFORMAT		对命令管道使用保留字 time 时，指定输出的格式
TMOUT		如果设置为正数，则表示 shell 会自未收到输入多少秒后自动终止
TMPDIR		如果设置，bash 会在指定的目录中创建临时文件
UID	R	当前用户的真实用户 ID（数字形式）

[3]这里再详细说明一下：如果没有使用双引号引用，每个位置参数会被扩展成独立的单词。这些单词会接着进行单词分割和路径名扩展。——译者注

[4]准确地说，出现在双引号（"@"）时，@ 会被扩展为以双引号引用的字符串列表形式。——译者注

[5]这里再详细说明一下 HISTFILESIZE 和 HISTSIZE 这两个环境变量的区别。首先，要区分开“历史记录列表”（history list）和“历史记录文件”（history file）。前者

位于内存中，在 bash 会话进行期间更新。后者位于硬盘，在 bash shell 中通常是 ~/.bash_history。会话结束后，历史记录列表中的内容会写入历史记录文件。如果 HISTFILESIZE=200，表示历史记录文件中最多能保存 200 个命令；如果 HISTSIZE=20，表示不管你输入多少命令，历史记录列表只记录 20 个命令，最终也只有这 20 个命令会在会话结束后写入历史记录文件。——译者注

[6]也就是 $"..." 这种形式。——译者注

A.7 set选项

set -arg 可以启用表 A-7 中的选项。除非特别说明，这些选项默认状态都是禁用。全称（full name）可作为 set -o 的参数使用。braceexpand、histexpand、history、keyword、onecmd 这些全称只能用于 bash 2.0 之后。在这些版本中，可以使用 -d 来启用散列化。

仅在启用 readline 的 show-mode-in-prompt 变量（参见表 A-22）时，才显示模式字符串变量。

表A-7：set选项

选项	全称（-o）	含义
-a	allexport	导出后续定义或修改的所有变量
-B	braceexpand	执行花括号扩展。该选项默认启用
-b	notify	立即报告后台作业的终止状态
-C	noclobber	用 >、>&、<> 阻止输出重定向覆盖已有文件
-E	errtrace	如果设置的话，ERR 上的所有陷阱都会被 shell 函数、命令替换、子 shell 环境中执行的命令所继承

选项	全称（-o）	含义
-e	errexit	如果简单命令的退出状态不为 0，则退出 shell。简单命令不属于 while、until、if、&&、|| 的一部分，也并非返回值能够被 ！取反的命令 [7]
	emacs	使用Emacs 风格的行编辑功能。这也会影响到 read -e 所用到的编辑功能
-f	noglob	禁止文件名扩展（通配符匹配）
-H	histexpand	启用 ！形式的历史替换。该选项在交互式 shell 中默认启用
	history	启用命令历史记录。该选项在交互式 shell 中默认启用
-h	hashall	在查找待执行命令时，定位并记录（散列化）这些命令。该选项默认启用
	ignoreeof	阻止读取 EOF（Ctrl-D）时退出交互式shell
-k	keyword	所有以赋值语句形式出现的参数均被置于命令的环境中，包括命令名之前的赋值语句
-m	monitor	启用作业控制。该选项在交互式 shell 中默认启用
-n	noexec	读取命令但不执行。可用于检查脚本的语法错误。该选项会被交互式 shell 忽略
-P	physical	如果设置的话，在执行会修改当前目录的命令（如 cd）时，不解析符号链接

选项	全称（-o）	含义
-p	privileged	启用特权模式
	pipefail	管道的返回值是最后一个（最右侧）以非 0 状态退出的命令的返回值，如果管道中的所有命令都顺利退出，则管道的返回值为 0。该选项默认禁用
	posix	修改默认操作与 POSIX 标准不一致的 bash 行为，使之符合标准
-T	functrace	如果设置的话，DEBUG 和 RETURN 上的所有陷阱都会被 shell 函数、命令替换、子 shell 环境中执行的命令所继承
-t	onecmd	读取并执行一个命令之后退出
-u	nounset	在执行参数扩展时，将未设置的变量或参数（不包括特殊参数 @ 或 *）视为错误
-v	verbose	在执行前显示命令行
	vi	使用 vi 风格的命令行编辑
-x	xtrace	在执行前显示（扩展后）命令行
--		如果该选项之后没有参数，则清除位置参数。否则，将位置参数依次设置为选项之后出现的参数，即便其中某些参数以 - 开头
-		表示选项结束。剩余所有参数依次赋给位置参数。关闭 -x 和 -v 选项。如果没有参数可用于设置，则位置参数保持不变

[7]简单命令是最常碰到的一类命令。这种命令其实就是空白字符分隔的一系列单词，最后由 shell 的某个控制运算符结尾。第一个单词通常指定了要执行的命令，后续单词则作为该命令的参数。——译者注

A.8 shopt选项

shopt 选项用 shopt -s *arg* 来设置，并用 shopt -u *arg* 来清除（参见表 A-8）。bash 2.0 之前的版本有环境变量来执行其中的某些设置。效果等同于 shopt -s。它们（以及对应的 shopt 选项）分别是：allow_null_glob_expansion（nullglob）、cdable_vars（cdable_vars）、command_oriented_history（cmdhist）、glob_dot_filenames（dotglob）、no_exit_on_failed_exec（execfail）。如今这些环境变量已经不存在了。

选项 extdebug、failglob、force_fignore、gnu_errfmt 在 bash 3.0 版本之前不可用。

表A-8：shopt选项

选项	含义（如果存在的话）
autocd	如果指定的命令不能执行，尝试将其作为 cd 命令的参数
cdable_vars	如果 cd 命令的参数不是目录，则将其视为变量名，该变量的值就是要切换到的目录
cdspell	如果存在适合的匹配，自动纠正 cd 命令的目录名中出现的轻微拼写错误。纠正对象包括漏写的字母、错误的字母以及字母错误。该选项仅适用于交互式 shell

选项	含义（如果存在的话）
checkhash	在散列表中找到的命令会在执行前被检查是否存在，如果不存在，则强制搜索 $PATH
checkjobs	在退出交互式 shell 前，bash 会列出所有已停止和正在运行的作业的状态
checkwinsize	bash 会在每个命令结束之后检查窗口大小，并根据需要更新 LINES 和 COLUMNS 的值
cmdhist	bash 会尝试在单个历史记录条目中保存多行命令的所有内容
compat32	在使用某些命令和运算符时，对于特定语言环境的字符串比较，bash 会将其行为更改为 3.2 版本
compat40	在使用某些命令和运算符时，对于特定语言环境的字符串比较，bash 会将其行为更改为 4.0 版本
compat41	当处于 POSIX 模式时，bash 会将双引号参数扩展中出现的单引号视为特殊字符
compat42	在处理模式替换单词扩展内的替换字符串时，bash 不会移除引号
compat43	对于某些操作，bash 将其行为更改为 4.3 版本
complete_fullquote	在进行补全时，bash 会引用文件名和目录名中的全部 shell 元字符
direxpand	在进行文件名补全时，bash 会使用单词扩展的结果替换目录名

选项	含义（如果存在的话）
dirspell	如果最初提供的目录名不存在，那么 bash 会尝试在单词补全期间更正目录名的拼写
dotglob	bash 会将以点号起始的文件名也包含在文件名扩展的结果之中
execfail	如果不能执行 exec 命令的参数，非交互式 shell 不会退出。如果 exec 失败，交互式 shell 不会退出
expand_aliases	对别名进行扩展
extdebug	启用调试器需要的行为。例如，declare 的 -F 选项显示作为参数的各个函数名所对应的源文件和行号；如果 DEBUG 陷阱运行的命令的返回不为 0，则跳过下一个命令；如果 DEBUG 陷阱运行的命令的返回值为 2，而且 shell 是在子例程中运行的，则模拟 return 调用
extglob	启用扩展模式匹配
extquote	如果设置的话，${*parameter*} 中出现的 $*string* 和 $"*string*" 会在双引号中进行扩展
failglob	路径名扩展过程中无法匹配的模式会产生扩展错误
force_fignore	在进行单词补全时忽略 $FIGNORE 变量指定的后缀单词，哪怕被忽略的这些单词是唯一可行的补全选择
globasciiranges	在进行比较时，模式匹配的方括号表达式内的区间表达式依照传统的 C 语言环境中的方式来进行处理

选项	含义（如果存在的话）
globstar	在文件名扩展中出现的模式 ** 将会匹配所有的文件以及零个或多个目录和子目录
gnu_errfmt	按照标准 GUN 错误消息格式写入 shell 错误消息
histappend	当 shell 退出时，历史列表被追加到变量 $HISTFILE 所指定的文件中，而不是覆盖该文件
histreedit	如果使用了 readline，则有机会重新编辑失败的历史记录替换
histverify	如果使用了 readline，历史记录替换的结果不会立即传给 shell 解析器，而是载入 readline 的编辑缓冲区，以允许进一步修改
hostcomplete	如果使用了 readline，对某个以 @ 开头的单词进行补全时，则会尝试执行主机名补全
huponexit	当交互式登录 shell 退出时，bash 会向所有作业发送 SIGHUP 信号
inherit_errexit	命令替换继承 errexit 选项的值，而不是在子 shell 环境中将其清除
interactive_comments	在交互式 shell 中，允许忽略以 # 开头的单词以及行中所有后续字符
lastpipe	如果未启用作业控制，在当前 shell 环境中运行管道中的最后一个命令

选项	含义（如果存在的话）
lithist	如果启用了 cmdlist 选项，则尽可能使用内嵌换行符（而非分号）将多行命令保存在历史记录中
login_shell	bash 会作为登录 shell 启动。该选项的值为只读状态
mailwarn	如果自上一次检查后用于检查邮件的文件再次被访问，则显示消息“The mail in mailfile has been read”
no_empty_cmd_completion	如果使用了 readline，尝试在空行上进行补全时不再搜索 $PATH
nocaseglob	在执行路径名补全时，bash 采用不区分大小写的方式来匹配文件名称
nocasematch	在执行特定操作时，bash 采用不区分大小写的方式来匹配模式
nullglob	这会导致未匹配到任何文件的模式被扩展成空串，而不是原封不动地保留下来
progcomp	启用可编程补全功能。该选项默认启用
promptvars	经过扩展后，提示符字符串还要进行变量扩展和参数扩展
restricted_shell	shell 以受限模式启动。该选项值不能更改
shift_verbose	如果移动个数超出最后一个位置参数，那么内建命令 shift 就会报错

选项	含义（如果存在的话）
sourcepath	内建命令 source 会使用 $PATH 的值来查找作为参数所指定的文件
xpg_echo	这会使得 echo 默认扩展反斜线转义序列

A.9 测试运算符

表 A-9 中的运算符用于 test 以及 [ ] 和 [[ ]] 构件。可以在逻辑上使用 -a（and）和 -o（or）将其组合在一起，并用转义过的括号 (\(\)) 对其分组。用于字符串比较的 < 和 > 以及 [[ ]] 构件在 bash 2.0 版本之前不可用，=~ 仅适用于 bash 3.0 及后续版本。

表A-9：测试运算符

运算符	为真的条件
-a *file*	*file* 存在，等同于 -e
-b *file*	*file* 存在且为块设备文件
-c *file*	*file* 存在且为字符设备文件
-d *file*	*file* 存在且为目录
-e *file*	*file* 存在，等同于 -a
-f *file*	*file* 存在且为普通文件

运算符	为真的条件
-g *file*	*file* 存在且设置了 *setgid* 位
-G *file*	*file* 存在且由有效组 ID 拥有
-h *file*	*file* 存在且为符号链接；等同于 -L
-k *file*	*file* 存在且设置了粘滞位
-L *file*	*file* 存在且为符号链接；等同于 -h
-n *string*	*string* 的长度不为 0
-N *file*	*file* 自上一次被读取后发生了改动
-O *file*	*file* 存在且由有效用户 ID 拥有
-p *file*	*file* 存在且为管道或具名管道（FIFO 文件）
-r *file*	*file* 存在且可读
-s *file*	*file* 存在且不为空
-S *file*	*file* 存在且为套接字
-t *fd*	文件描述符 *fd* 已打开且引用的是终端
-u *file*	*file* 存在且设置了 *setuid* 位

运算符	为真的条件
-w *file*	*file* 存在且可写
-x *file*	*file* 存在且可执行，或者 *file* 是可搜索的目录
-z *string*	*string* 的长度为 0
file1 -ef *file2*	*file1* 和 *file2* 引用的是相同的设备和 i 节点号
file1 -nt *file2*	*file1* 的修改日期晚于 *file2*，或者 *file1* 存在且 *file2* 不存在
file1 -ot *file2*	*file1* 的修改日期早于 *file2*，或者 *file1* 存在且 *file2* 不存在
string1 = *string2*	*string1* 等于 *string2*（POSIX 版本）
string1 == *string2*	*string1* 等于 *string2*
string1 != *string2*	两个字符串不相等
string1 < *string2*	按照字典排序后，*string1* 位于 *string2* 之前
string1 > *string2*	按照字典排序后，*string1* 位于 *string2* 之后

运算符	为真的条件
string1 =~ *regexp*	*string1* 匹配扩展正则表达式 *regexp*[a]
exprA -eq *exprB*	算术表达式 *exprA* 和 *exprB* 相等
exprA -ne *exprB*	算术表达式 *exprA* 和 *exprB* 不相等
exprA -lt *exprB*	*exprA* 小于 *exprB*
exprA -le *exprB*	*exprA* 小于或等于 *exprB*
exprA -gt *exprB*	*exprA* 大于 *exprB*
exprA -ge *exprB*	*exprA* 大于或等于 *exprB*
exprA -a *exprB*	*exprA* 和 *exprB* 均为真
exprA -o *exprB*	*exprA* 或 *exprB* 为真

a. 仅适用于 bash 3.0 及后续版本。或许只能在 [[]] 内部使用。

A.10 I/O重定向

表 A-10 完整地列出了所有的 I/O 重定向操作符。注意，用于指定 STDOUT 和 STDERR 重定向的格式有两种：&>*file* 和 >&*file*，其中第二种（也是本书使用的）是首选方法。

表A-10：输入/输出重定向

重定向操作符	功能
cmd1|*cmd2*	管道，将 *cmd1* 的标准输出作为 *cmd2* 的标准输入
cmd1|&*cmd2*	管道，将 *cmd1* 的标准输出和标准错误作为 *cmd2* 的标准输入（适用于 bash 4.0 及后续版本）
>*file*	将标准输出导向 *file*
<*file*	从 *file* 获取标准输入
>>*file*	将标准输出导向 *file*。如果 *file* 已存在，则执行追加操作
>|*file*	强制将标准输出导向 *file*，哪怕是设置了 *noclobber*
n>|*file*	强制将文件描述符 *n* 的标准输出导向 *file*，哪怕是设置了 *noclobber*
<>*file*	使用 *file* 作为标准输入和标准输出
&>*file*	将标准输出和标准错误导向 *file*
&>>*file*	将标准输出和标准错误导向 *file*。如果 *file* 已存在，则执行追加操作（适用于 bash 4.0 及后续版本）
n<>*file*	使用 *file* 作为文件描述符的输入和输出
<<*label*	here-document

重定向操作符	功能
<<<*word*	here-string
n>*file*	将文件描述符 *n* 的输出导向 *file*
n<*file*	从 *file* 获取文件描述符 *n* 的输入
n>>*file*	将文件描述符 *n* 的输出导向 *file*。如果 *file* 已存在，则执行追加操作
n>&	将标准输出复制到文件描述符 *n*
n<&	将标准输入复制到文件描述符 *n*
n>&*m*	使文件描述符 *n* 成为输出文件描述符 *m* 的副本
n<&*m*	使文件描述符 *n* 成为输入文件描述符 *m* 的副本
&>*file*	将标准输出和标准错误导向 *file*
<&-	关闭标准输入
>&-	关闭标准输出
n>&-	关闭文件描述符 *n* 的输出
n<&-	关闭文件描述符 *n* 的输入

重定向操作符	功能
n>&*word*	如果未指定 *n*，则使用标准输出（文件描述符 1）；如果 *word* 中的数字并未指定已打开的文件描述符以供输出，那么会产生重定向错误。特殊情况是，如果忽略 *n*，并且 *word* 没有扩展成一个或多个数字，则重定向标准输出和标准错误
n<&*word*	如果 *word* 扩展为一个或多个数字，使 *n* 代表的文件描述符成为前者所代表的文件描述符的副本；如果 *word* 中的数字并未指定已打开的文件描述符以供输入，那么会产生重定向错误。如果 *word* 求值后为 -，则关闭文件描述符 *n*；如果未指定 *n*，则使用标准输入（文件描述符 0）
n>&*digit*-	将文件描述符 *digit* 移动至文件描述符 *n*，如果未指定 *n*，则移动至标准输出（文件描述符 1）
n<&*digit*-	将文件描述符 *digit* 移动至文件描述符 *n*，如果未指定 *n*，则移动至标准输入（文件描述符 0）。然后关闭 *digit*

A.11 echo选项与转义序列

echo 接受 3 个参数（参见表 A-11）。

表A-11：echo选项

选项	功能
-e	允许解释反斜线转义序列
-E	不解释反斜线转义序列，即便在默认进行解释的系统上
-n	忽略结尾的换行符（等同于 \c 转义序列，参见表 A-12）

echo 也可以接受多个以反斜线起始的转义序列（参见表 A-12）。除了 \f，这些转义序列的行为差不多都能预测，在有些显示器上，\f 会清理屏幕，而在另一些显示器上，它会产生换行，而且在大多数打印机上会弹出纸张。

表A-12：echo转义序列

序列	显示的字符
\a	报警或 Ctrl-G（响铃）
\b	退格或 Ctrl-H
\c	阻止后续输出

序列	显示的字符
\e	转义字符（等同于 \E）
\E	转义字符
\f	馈页（form feed）或 Ctrl-L
\n	换行（不在命令结尾）或 Ctrl-J
\r	回车或 Ctrl-M
\t	制表符或 Ctrl-I
\v	垂直制表符或 Ctrl-K
\\	单反斜线
\0*nnn*	8 位字符 [8]，其值等于八进制（以 8 为基数）值 *nnn*（0~3 个八进制数位）
nnn	等同于 \0*nnn*
\x*HH*	8 位字符，其值等于十六进制（以 16 为基数）值 *HH*（1~2 个十六进制数位）
\u*HHHH*	Unicode（ISO/IEC 10646）字符，其值等于十六进制值 *HHHH*（1~4 个十六进制数位）

序列	显示的字符
\U*HHHHHHHH*	Unicode（ISO/IEC 10646）字符，其值等于十六进制值 *HHHHHHHH*（1~8 个十六进制数位）

[8]意思就是该字符占一个字节（8 位二进制）。——译者注

\n、\0 以及 \x 序列甚至更加依赖于设备，可用于光标控制、特殊图形字符等复杂 I/O。

A.12 printf

printf 命令从 bash 2.02 就存在了，该命令分为 3 部分（除命令名之外）。

```
printf [-v var] format-string [arguments]
```

-v *var* 是可选的，可以将输出赋给变量 *var*，而不再发往标准输出。

format-string 描述了格式说明。这部分最好作为字符串常量放在引号中。*arguments* 是一个列表，例如，对应于格式说明的一系列字符串或变量值。

如有必要，格式会被重用，以便处理完所有的参数。如果格式要求的参数多于现有的参数，多出的那些格式说明符就假定提供的参数是 0 值或空串（视情况而定）。

格式说明前面要加上百分号（%），说明符可以是表 A-13 中的某一个。主要的格式说明符有两种，分别是用于字符串的 %s 和用于十进制整数的 %d。

表A-13：printf格式说明符

格式	字符含义
%b	导致 printf 扩展对应参数中的反斜线转义序列，其方式和 echo -e 相同
%c	ASCII 字符（输出对应参数的第一个字符）
%d、%i	十进制（以 10 为基数）整数
%e	浮点数格式（[-]*d.precisione*[+-]*dd*）。参见表格之后有关 *precision* 含义的描述
%E	浮点数格式（[-]*d.precisionE*[+-]*dd*）
%f	浮点数格式（[-]*ddd.precision*）
%g	%e 或 %f 转换（更短者优先），同时去掉结尾的 0
%G	%E 或 %f 转换（更短者优先），同时去掉结尾的 0
%o	无符号八进制数
%q	导致 printf 输出的对应参数能够作为 shell 输入重用
%s	字符串
%u	无符号十进制数
%x	无符号十六进制数，使用 a-f 代表 10~15

格式	字符含义
%X	无符号十六进制数，使用 A-F 代表 10~15
%%	字面 %
%(*datefmt*)T	使用 datefmt 作为 strftime 的格式字符串并将结果作为 printf 输出的日期—时间字符串（适用于 bash 4.2 及后续版本）

printf 命令能够指定输出字段的宽度和对齐方式。格式表达式可以在 % 和格式说明符之间使用 3 个可选的修饰符。

```
%<flags<>>width.precision<> format-specifier>
（%<标志<>>宽度.精度<> 格式说明符>）
```

输出字段的宽度是一个数字值。如果指定字段宽度，该字段的内容默认采用右对齐。要想左对齐，必须指定 - 标志（其余的标志参见表 A-16）。因此，%-20s 输出的就是字段宽度为 20 个字符的左对齐字符串。如果字符串的宽度不足 20 个字符，那么会用空白字符填充该字段。以下示例将格式说明符放在格式字符串内的一对 | 之间，以看出输出字段的宽度。下面是一个文本右对齐的示例。

```
printf "|%10s|\n" hello@
```

结果为：

```
|        hello|
```

接下来是一个文本左对齐的示例：

```
printf "|%-10s|\n" hello
```

结果为：

```
|hello          |
```

精度修饰符用于小数或浮点数，控制着结果中的数位个数。对于字符串值来说，它控制着最多能从该字符串输出多少个字符。

你甚至可以通过 printf 参数列表中的值来动态指定宽度和精度。为此，在格式表达式中指定 *，而非字面数值。

```
$ myvar=42.123456
$ mysig=6
$ printf "|%*.*G|\n" 5 $mysig $myvar
|42.1235|
$
```

这个示例中的宽度是 5，精度是 6，要输出的值取自变量 $myvar。精度是可选的，其具体含义视控制字母而定，参见表 A-14。

表A-14：基于**printf**格式说明符的“精度”含义

格式	“精度”的含义
%d、%I、%o、%u、%x、%X	要输出的最小数位个数。如果数值的位数不足，则使用前导 0 进行填充。默认精度为 1
%e、%E	要输出的最小数位个数。如果数值的位数不足，则在小数点后填充 0。默认精度为 10。如果精度为 0，则不输出小数点
%f	小数点右侧的数位个数
%g、%G	最大有效位数
%s	输出的最大字符数

格式	“精度”的含义
%b	[POSIX shell—可能无法移植到其他版本的 printf。] 当代替 %s 使用时，%b 会在参数字符串中扩展 echo 风格的转义序列（参见表 A-15）
%q	[POSIX shell—可能无法移植到其他版本的 printf。] 当代替 %s 使用时，%q 输出的字符串参数可作为 shell 输入使用

%b 和 %q 是对 bash（以及其他 POSIX 兼容 shell）的补充，其提供实用特性的代价是，无法移植到其他 shell 和 Unix 中的 printf 命令。以下两个示例更清晰地演示了两者的功能。

%q 的 shell 引用：

```
$ printf "%q\n" "greetings to the world"
greetings\ to\ the\ world
$
```

%b 的 echo 风格转义序列：

```
$ printf "%s\n" 'hello\nworld'
hello\nworld
$ printf "%b\n" 'hello\nworld'
hello
world
$
```

表 A-15 展示了能够在 %b 格式输出的字符串中转换的转义序列。

表A-15：printf转义序列

转义序列	含义

转义序列	含义
\e	转义字符
\a	响铃字符
\b	退格字符
\f	馈页字符
\n	换行字符
\r	回车字符
\t	制表符
\v	垂直制表符
\'	单引号字符
\"	双引号字符
\\	反斜线字符
nnn	8 位字符，其 ASCII 值为 1~3 个八进制数位 *nnn*
xHH	8 位字符，其 ASCII 值为 1~2 个十六进制数位 *HH*

最后，printf 格式说明符中的字段宽度和精度之前可以加上一个或多个标志。我们已经见过用于左对齐的标志 -。其余的标志如表 A-16 所示。

表A-16：**printf**标志

字符	描述
-	将字段中格式化过的值左对齐
空格	在正值前加上空格，在负值前加上减号
+	始终为数值加上符号前缀，即便是正值
#	使用另一种形式：%o 使用 0 作为前缀；%x 和 %X 分别使用 0x 和 0X 作为前缀；%e、%E、%f 始终在结果中出现小数点；%g 和 %G 不删除末尾的 0
0	不再使用空格，改用 0 填充输出。这仅在字段宽度宽于转换后的结果宽度时出现。在 C 语言中，该标志应用于包括非数值在内的所有输出格式。对于 bash 来说，它仅应用于数值格式
'	如果格式说明包含 %i、%d、%u、%f、%F、%g 或 %G，则使用千分符进行格式化（尽管属于 POSIX 标准，但这未必会实现）

A.12.1 示例

以下这些 printf 用法示例用到了一些 shell 变量，如表 A-17 所示。

```
PI=3.141592653589
```

表A-17：**printf**示例

printf 语句	结果	注释
`printf '%f\n' $PI`	`3.141593`	注意默认的舍入
`# 并非你想的那样` `printf '%f.5\n' $PI`	`3.141593.5`	这是一个常见的错误：格式说明符应该在 `%f` 的另一侧；因为没有书写正确，`.5` 就像其他文本一样，追加到变量
`printf '%.5f\n' $PI`	`3.14159`	在小数点右侧留出 5 个空位
`printf '%+.2f\n' $PI`	`+3.14`	添加前导 `+` 号，小数点右侧只显示两个数位
`printf '[%.4s]\n' s string`	`[s] [stri]`	截取 4 个字符；只提供了 1 个字符的话，就只得到 1 个字符宽度的输出。注意，这里重复使用了格式字符串
`printf '[%4s]\n' s string`	`[ s]` `[string]`	确保使用最小 4 个字符的字段宽度，右对齐。不截取
`printf '[%-4.4s]\n' s string`	`[s ]` `[stri]`	一网打尽——最小宽度为 4，最大宽度为 4，在必要时截断，宽度如果不足 4，则左对齐（因为 - 号）

接下来还有一个无法在表格中很好呈现的示例。编写 printf 语句的传统方式是，将包括换行在内的所有格式化信息全部嵌入格式化字符串。表 A-17 中就是这么做的。我们鼓励这种做法，但你并不是非得这么做，有时不这样反而会更容易些。注意，下例中的→代表输出中的制表符。

```
$ printf "%b" "\aRing terminal bell, then tab\t then newline\nThen
line 2.\n"
```

```
Ring terminal bell, then tab → then newline
Then line 2.
```

最后，我们着实喜欢 bash 4.2 中引入的 %(*datefmt*)T，有了它，就不必创建子 shell 来调用 date 了，如下所示。

```
$ printf "%(%F)T\n" '-1'
2017-02-06

$ printf "%(%F_%T)T\n"
2017-02-06_20:31:25

$ printf "%(%F_%T%z)T: %s\n" '-1' 'Your log line here...'
2017-02-06_20:32:24-0500: Your log line here...
```

> bash 得经历几次发布后才能稳定住 %(*datefmt*)T ！有时候可以忽略 -1 参数，有时候不能，这取决于你使用的 bash 版本以及所执行的操作。为了获得最大的可移植性，不要忽略该参数。

A.12.2 参考

- The printf command [Bash Hackers Wiki]

A.13 用**strftime**格式化日期和时间

表 A-18 展示了常用的日期和时间字符串格式化选项。查询系统中 date 和 strftime(3) 的手册页，两者的选项及其含义视系统而异。

表A-18：**strftime**格式代码

格式	描述
%%	字面 %

格式	描述
%a	所在语言环境中“星期几”的名称缩写（Sun……Sat）
%A	所在语言环境中完整的“星期几”名称（Sunday……Saturday）
%B	所在语言环境中完整的月份名称（January……December）
%b	所在语言环境中缩写形式的月份名称（Jan……Dec）。参见 %h
%c	所在语言环境中默认/偏好的日期和时间描述
%C	十进制数形式的世纪（年份除以 100，然后截取为整数）
%d	十进制形式的月中某日（01……31）
%D	%m/%d/%y（MM/DD/YY）格式的日期。注意，美国使用 MM/DD/YY，其他国家使用 DD/MM/YY，因此这个格式存在歧义，应该避免。可以用 %F 代替，该格式是公认的标准，也能够很好地排序
%e	以空白填充的十进制形式的月中某日（1……31）
%F	日期格式为 %Y-%m-%d（ISO 8601 日期格式 CCYY-MM-DD，除非是完整的月份名称，就像在 HP-UX 中那样）
%g	%V 给出的星期数所对应的两位数年份（YY）
%G	%V 给出的星期数所对应的 4 位数年份（CCYY）

格式	描述
%H	十进制形式的小时数（24 小时制，00……23）
%h	所在语言环境中月份名称的缩写（Jan……Dec）。参见 %b
%I	十进制形式的小时数（12 小时制，01……12）
%j	十进制形式的天数（001……366）
%k	以空白填充的十进制形式小时数（24 小时制，00……23）
%l	以空白填充的小时数（12 小时制，01……12）
%m	十进制形式的月份（01……12）
%M	十进制形式的分钟数（00……59）
%n	字面换行符
%N	纳秒（000000000……999999999）。[GNU]
%p	所在语言环境中与“AM”或“PM”等价的描述
%P	所在语言环境中与“am”或“pm”等价的描述
%r	所在语言环境中使用AM/PM 记法的12 小时制时间描述（HH:MM:SS AM/PM）

格式	描述
%R	%H:%M（HH:MM）格式的时间
%s	从 UNIX 纪元年（世界标准时间 1970 年1 月1 日 00:00:00）起的秒数
%S	十进制形式的秒数（00……61）。取值范围是（00……61），而非（00……59），以此允许周期性出现闰秒和双闰秒
%t	字面制表符
%T	%H:%M:%S（HH:MM:SS）格式的时间
%u	十进制形式的“星期几”（1……7，以星期一作为每周的第一天）
%U	十进制形式的年中第几周（00……53，以星期日作为每周的第一天）
%v	%e-%b-%Y（D-MMM-CCYY）格式的日期。[不标准]
%V	十进制形式的年中第几周（00……53，以星期一作为每周的第一天）。根据 ISO 8601，包含 1 月 1 日的那个星期（如果至少还剩 4 天）是新年的第一周；否则就是去年的第 53 周，接下来的那个星期是新年的第一周。年份由转换说明 %G 给出
%w	十进制形式的“星期几”（0……6，以星期一作为每周的第一天）
%W	十进制形式的年中第几周（00……53，以星期一作为每周的第一天）
%x	适合所在语言环境的日期描述

格式	描述
%X	适合所在语言环境的时间描述
%y	十进制形式的年份，不包含世纪（00……99）
%Y	十进制形式的年份，包含世纪
%z	世界标准时间偏差，采用 ISO 8601 格式 [-]hhmm
%Z	时区名称

A.14 模式匹配字符

表 A-19 列出了 bash 中的模式匹配字符。本节中的内容摘自 Bash Reference Manual。

表A-19：模式匹配字符

字符	含义
*	匹配包括空串在内的任意字符串
?	匹配任意单个字符
[]	匹配其中的任意单个字符
[!] 或 [^]	匹配不在其中的任意单个字符

下面的 POSIX 字符类可以用于方括号内（详见系统 grep 或 egrep 的手册页）。

```
[[:alnum:]] [[:alpha:]] [[:ascii:]] [[:blank:]] [[:cntrl:]]
[[:digit:]]
[[:graph:]] [[:lower:]] [[:print:]] [[:punct:]] [[:space:]]
[[:upper:]]
[[:word:]]  [[:xdigit:]]
```

字符类 word 能够匹配字母、数字以及字符 _。

[=c=] 匹配所有与字符 c 拥有相同排序规则权重（由当前语言环境定义）的字符，而 [.*symbol*.] 则匹配排序规则符号 *symbol*。

这些字符类都会受到语言环境设置的影响。要想获得传统的 Unix 值，可以使用 LC_COLLATE=C 或 LC_ALL=C。

A.15 extglob扩展模式匹配运算符

表 A-20 中的运算符适用于使用 shopt -s extglob 时。匹配时区分大小写，但你可以用 shopt -s nocasematch（bash 3.1+）修改。该选项会影响到 case 和 [[命令。

表A-20：extglob扩展模式匹配运算符

分组	含义
@()	（模式）只出现 1 次
*()	（模式）出现 0 次或多次
+()	（模式）出现 1 次或多次
?()	（模式）出现 0 次或 1 次

分组	含义
!()	（模式）不出现

A.16 tr转义序列

表 A-21 列出了 tr 转义序列。

表A-21：tr转义序列

序列	含义
\ooo	编码值为八进制 ooo（1~3 个八进制数位）的字符
\\	反斜线字符（转义反斜线自身）
\a	响铃，ASCII BEL 字符（因为 b 已经被退格占用了）
\b	退格
\f	馈页
\n	换行
\r	回车
\t	制表符（有时也称为水平制表符）

序列	含义
\v	垂直制表符

A.17 readline的init文件语法

GNU readline 库为 bash 和其他一些 GNU 实用工具提供了命令行。其可配置性出奇得好，但大多数人并未意识到这一点。

表 A-22、表 A-23、表 A-24 中列出了部分可用的变量。完整的细节参见 readline 的文档。

你可以在 init 文件中使用 set 命令来更改 readline 的变量值，从而改变 readline 的运行时行为。语法很简单：

```
set variable value
```

以下示例展示了如何将默认的类似于 Emacs 风格的按键绑定更改为 vi 行编辑命令。

```
set editing-mode vi
```

在适当的情况下，变量名及其值不用区分大小写也能被识别。未识别的变量名称会被忽略。

如果布尔类型变量（可以设置为 on 或 off）的值为空、on（不区分大小写）或 1，则将其设置为 on。若是其他值，则设置为 off。

另外，启用 readline 变量 show-mode-in-prompt 时，才会显示模式字符串变量（参见表 A-7）。

表A-22：readline配置设置

变量	描述
bell-style	readline 想让终端响铃时，控制会发生什么情况。如果设置为 none，readline 不响铃。如果设置为 visible，readline 使用可视化响铃 [9]（如果可用的话）。如果设置为 audible（默认），readline 会尝试让终端响铃
bind-tty-special-chars	如果设置为 on，readline 会尝试将内核终端驱动程序特殊对待的控制键绑定到 realine 功能相同的按键。该变量默认值为 on
blink-matching-peren	如果设置为 on，则插入右括号时，readline 会尝试将光标短暂移动到左括号。默认为 off
colored-completion-prefix	如果设置为 on，在列出可能的补全结果时，readline 会使用不同的颜色显示其中的公共前缀。颜色定义取自环境变量 LS_COLORS 的值。该变量默认为 off
colored-stats	如果设置为 on，readline 会使用不同的颜色显示可能的补全结果，以指明其文件类型。颜色定义取自环境变量 LS_COLORS 的值。该变量默认为 off
comment-begin	执行 insert-comment 命令时，插入行首的字符串。该变量默认值为 #
completion-display-width	在执行补全操作时，用于显示可能的补全结果所使用的屏幕列数。如果这个值小于 0 或大于终端屏幕宽度，则忽略。如果取值为 0，则每行只显示一个补全结果。该变量默认值为 -1
completion-ignore-case	如果设置为 on，readline 以忽略大小写的方式执行文件名匹配和补全。该变量默认值为 off

变量	描述
completion-map-case	如果设置为 on 且启用了 completion-ignore-case，在执行文件名匹配和补全时，readline 会将连字符（-）和下划线（_）视为等价
completion-prefix-display-length	在显示可能的补全结果时，不改动其公共前缀的字符数量。如果这个值大于 0，超出该值的公共前缀会在显示时被替换成省略号
completion-query-items	可能的补全结果数量，用于决定什么时候询问用户是否要显示补全结果。如果数量大于该值，readline 会询问用户是不是要显示；否则，直接显示。此变量必须设置为大于或等于 0 的整数。负数意味着 readline 不用询问。默认值为 100
convert-meta	如果设置为 on，对于设置过第 8 位的字符，readline 会去除此位并在该字符前添加 Esc 字符，从而将其转换成一个带有前置辅助键的 ASCII 按键序列。该变量默认值为 off
disable-completion	如果设置为 on，readline 会禁止单词补全。补全字符会被插入行中，就好像这些字符被映射到了 self-insert。该变量默认值为 off
echo-control-characters	如果设置为 on，在支持该功能的操作系统上，readline 会回显与键盘产生的信号所对应的字符。该变量默认值为 on
editing-mode	控制默认使用哪种按键绑定。默认情况下，readline 会在 Emacs 编辑模式下启动，其击键与 Emacs 最为接近。该变量可以设置为 emacs或 vi

变量	描述
emacs-mode-string	启用 Emacs 编辑模式时，显示在主提示符最后一行前的字符串。该值会像按键绑定一样进行扩展，因此可以使用标准的辅助和控制前缀以及反斜线转义序列。\1 和 \2 分别表示非打印字符序列的起止，可用于在模式字符串中嵌入终端控制序列。该变量默认值为 @
enable-bracketed-patse	如果设置为 on，readline 会设法配置终端，允许将粘贴的内容作为单个字符串插入编辑缓冲区，而不是将每个字符按照从键盘中读入那样对待。这能够避免将粘贴的字符解释为编辑命令。该变量默认值为 off
enable-keypad	如果设置为 on， 需要用到辅助键盘（application keypad）时，readline 会尝试将其启用。某些系统要靠它来使用箭头键。该变量默认值为 off
enable-meta-key	如果设置为 on，readline 会在需要时启用终端支持的所有辅助键。在很多终端上，辅助键用于发送 8 位字符。该变量默认值为 on
expand-tilde	如果设置为 on，readline 尝试进行单词补全时会执行波浪号（~）扩展。该变量默认值为 off
history-preserve-point	如果设置为 on，对于通过 previous-history 或 next-history 检索到的每行历史记录，历史代码会将标记点（光标的当前位置）停留在行中相同的位置。该变量默认值为 off
history-size	保存在历史记录列表中的最大条目数。如果设置为 0，删除全部现有条目且不再加入任何新条目。如果设置的值小于 0，条目数量不受限，这也是默认设置。如果试图将该变量设置为非数字值，最大条目数则为 500
horizontal-scroll-mode	如果设置为 on，当编辑的文本行长度超出屏幕宽度时，那么它们会在单行内水平滚动，而不是折到新的一行。该变量默认值为 off

变量	描述
input-meta	如果设置为 on，readline 则启用 8 位字符输入（不会清除所读取字符的第 8 位），不管终端是否支持。该变量默认值为 off，其同义词是 meta-flag
isearch-terminators	该字符串能够结束增量搜索，不再将字符当作命令执行。如果未设置这个变量，则 ESC 和 C-j 会结束增量搜索
keymap	设置 readline 当前用于按键绑定命令的按键映射（keymap）。可接受的按键映射名称包括 emacs、emacs-standard、emacs-meta、emacsctlx、vi、vi-move、vi-command、vi-insert。vi 等同于vi-command; emacs 等同于 emacs-standard。该变量的默认值为 emacs。editing-mode 变量的值也会影响默认的按键映射
keyseq-timeout	在读取模棱两可的按键序列时（能够使用已有的输入形成一个完整的按键序列，或者通过接收额外的输入形成一个更长的按键序列），指定 readline 要延迟多久来等待下一个字符。如果在这期间未接收到输入，readline 将使用长度更短却完整的按键序列。readline 使用该变量值来判断当前输入源（默认为 rl_instream）是否有可用的输入。值的单位是毫秒，因此，1000 代表 readline 会等待额外的输入一秒的时间。如果设置的值小于或等于 0，或者为非数字值，那么 readline 会一直等到有按键被按下，以此决定完整的按键序列。该变量的默认值为 500
mark-directories	如果设置为 on，那么会在补全的目录名称尾部追加斜线。该变量默认值为 on
mark-modified-lines	如果设置为 on，readline 会在已修改过的历史记录条目的行首显示一个星号（*）。该变量默认值为 off
mark-symlinked-directories	如果设置为 on，指向目录的符号链接被补全后，完整名称尾部会追加斜线（依照 mark-directories 的值）。该变量默认值为 off

变量	描述
match-hidden-files	如果设置为 on，在执行文件名称补全时，readline 会匹配以点号起始的文件（隐藏文件），除非用户在要补全的文件名称开头指定了 .。该变量默认值为 on
menu-complete-display-prefix	如果设置为 on，菜单补全会先显示可能的补全结果的公共前缀（可能为空），然后循环浏览。该变量默认值为 off
output-meta	如果设置为 on，readline 会直接显示设置了第 8 位的字符，而不是将其作为辅助键前置的转义序列。该变量默认值为 off
page-completions	如果设置为 on，readline 会使用类似于 more 的内部分页工具来显示满屏的补全结果
print-completions-horizontally	如果设置为 on，readline 会将匹配的补全结果按照字母顺序水平排列显示，而不是在屏幕中垂直显示。该变量默认值为 off
revert-all-at-newline	如果设置为 on，在执行 accept-line 前，readline 会恢复对历史记录条目所做的全部改动。默认情况下，历史记录条目可能会被修改并在多个 readline 调用中保留各自的恢复列表。该变量的默认值为 off
show-all-if-ambiguous	更改补全功能的默认行为。如果设置为 on，对于存在多个补全结果的单词，立即列出所有的匹配，不再响铃。该变量默认值为 off
show-all-if-unmodified	以类似于 show-all-if-ambiguous 的方式更改补全功能的默认行为。如果设置为 on，对于存在多个补全结果且均不是部分补全（可能的匹配结果不含有公共前缀）的单词，立即列出所有的匹配，不再响铃。该变量默认值为 off

变量	描述
show-mode-in-prompt	如果设置为 on，那么会在提示符的开头添加一个用于指示编辑模式的字符：emacs、vi-command 或 vi-insert。用户可以自行设置模式字符串。该变量默认值为 off
skip-completed-text	如果设置为 on，这会更改在行中插入单个匹配时的默认补全行为。仅在单词中间执行补全时才有效。启用之后，补全结果中与单词其余部分匹配的字符会被跳过，不再插入行中，因此光标后的部分单词不会重复出现。例如，如果启用了该功能，当光标位于 Makefile 的第一个 e 后时，补全操作产生的结果是 Makefile，而非 Makefilefile（假设只有单个可能的补全结果）。该变量默认为 off
vi-cmd-mode-string	启用 vi 编辑模式且处于 vi 的命令模式时，显示在主提示符最后一行前的字符串。该值会像按键绑定一样进行扩展，因此可以使用标准的辅助和控制前缀以及反斜线转义序列。\1 和 \2 分别表示非打印字符序列的起止，可用于在模式字符串中嵌入终端控制序列。该变量默认值为(cmd)
vi-ins-mode-string	启用 vi 编辑模式且处于 vi 的插入模式时，显示在主提示符最后一行前的字符串。该值会像按键绑定一样进行扩展，因此可以使用标准的辅助和控制前缀以及反斜线转义序列。\1 和 \2 分别表示非打印字符序列的起止，可用于在模式字符串中嵌入终端控制序列。该变量默认值为(ins)
visible-stats	如果设置为 on，列出可能的补全结果时，这会在文件名称后面加上一个表明文件类型的字符。该变量默认值为 off

[9]通常表现为终端屏幕闪烁一下。——译者注

A.18 Emacs模式命令

本节内容节选自 Cameron Newham 所著的 *Learning the bash Shell, 3rd Edition*（O'Reilly 出版）一书。

表 A-23 给出了一份完整的 readline Emacs 编辑模式命令清单。

表A-23：Emacs模式命令

命令	含义
Ctrl-A	将光标移动到行首
Ctrl-B	将光标向后移动一个字符
Ctrl-D	向前删除一个字符
Ctrl-E	将光标移动到行尾
Ctrl-F	将光标向前移动一个字符
Ctrl-G	中止当前正在编辑的命令并在终端响铃
Ctrl-J	等同于 Return
Ctrl-K	从光标位置向前一直删除（“杀死”）到行尾
Ctrl-L	清屏并重新显示当前行
Ctrl-M	等同于 Return
Ctrl-N	命令历史记录中的下一行
Ctrl-O	等同于 Return，然后显示命令历史记录文件中的下一行

命令	含义
Ctrl-P	命令历史记录中的上一行
Ctrl-R	向后搜索
Ctrl-S	向前搜索
Ctrl-T	调换两个字符的位置
Ctrl-U	从光标当前位置开始，向后删除到行首
Ctrl-V	逐字输入（verbatim）接下来的字符
Ctrl-V Tab	插入制表符
Ctrl-W	删除光标之后的单词，使用空白字符作为边界
Ctrl-X/	列出当前单词可能的文件名补全结果
Ctrl-X~	列出当前单词可能的用户名补全结果
Ctrl-X$	列出当前单词可能的变量名补全结果
Ctrl-X@	列出当前单词可能的主机名补全结果
Ctrl-X!	列出当前单词可能的命令名补全结果
Ctrl-X(	开始将字符存入当前的键盘宏

命令	含义
Ctrl-X)	停止将字符存入当前的键盘宏
Ctrl-Xe	重新执行上一个定义的键盘宏
Ctrl-X Ctrl-R	读入 readline 初始化文件内容
Ctrl-X Ctrl-V	显示当前 bash 实例的版本信息
Ctrl-Y	检索（yank）最近被删除的内容
Delete	向后删除一个字符
Ctrl-[	等同于Esc（在大多数键盘中）
Esc-B	将光标向后移动一个单词
Esc-C	将光标之后的单词首字母变成大写
Esc-D	向前删除一个单词
Esc-F	将光标向前移动一个单词
Esc-L	将光标之后的单词首字母变成小写
Esc-N	非增量向前搜索
Esc-P	非增量向后搜索

命令	含义
Esc-R	恢复对当前行的所有改动
Esc-T	交换两个单词的位置
Esc-U	将光标之后的单词全部变成大写
Esc-Ctrl-E	在行中执行 shell 别名、历史记录以及单词扩展
Esc-Ctrl-H	向后删除一个单词
Esc-Ctrl-Y	将上一个命令的首个参数（通常是第二个单词）插入光标处
Esc-Delete	向后删除一个单词
Esc-^	在行中执行历史记录扩展
Esc-<	移动到历史记录文件的第一行
Esc->	移动到历史记录文件的最后一行
Esc-.	将上一个命令的最后一个单词插入光标处
Esc-_	同上
Tab	尝试对当前单词执行文件名补全
Esc-?	列出光标之前可能的文本补全

命令	含义
Esc-/	尝试对当前单词执行文件名补全
Esc-~	尝试对当前单词执行用户名补全
Esc-$	尝试对当前单词执行变量名补全
Esc-@	尝试对当前单词执行主机名补全
Esc-!	尝试对当前单词执行命令名补全
Esc-Tab	尝试补全命令历史记录中的文本
Esc-~	尝试对当前单词执行波浪号扩展
Esc-\	删除光标两边的所有空格和制表符
Esc-*	在光标前插入 Esc-= 产生的所有补全
Esc-=	列出光标前所有可能的补全结果
Esc-{	尝试执行文件名补全并将结果返回给花括号内的 shell

A.19 vi控制模式命令

本节内容节选自 Cameron Newham 所著的 *Learning the bash Shell, 3rd Edition*（O'Reilly 出版）一书。

表 A-24 给出了一份完整的 readline vi 控制模式命令清单。

表A-24：vi模式命令

命令	含义
h	光标向左移动一个字符
l	光标向右移动一个字符
w	光标向右移动一个单词
b	光标向左移动一个单词
W	将光标移动到下一个非空白单词首部
B	将光标移动到前一个非空白单词首部
e	将光标移动到当前单词末尾
E	将光标移动到当前非空白单词末尾
0	将光标移动到行首
.	重复最近一次的 a 命令插入
^	将光标移动到行内第一个非空白字符
$	将光标移动到行尾
i	在当前字符前插入文本

命令	含义
a	在当前字符后插入文本
I	在行首插入文本
A	在行尾插入文本
R	覆盖现有文本
dh	向后删除一个字符
dl	向前删除一个字符
db	向后删除一个单词
dw	向前删除一个单词
dB	向后删除一个非空白单词
dW	向前删除一个非空白单词
d$	从光标位置删除到行尾
d0	从光标位置删除到行首
D	等同于 d$（删除到行尾）
dd	等同于 0d$（删除整行）

命令	含义
C	等同于 c$（删除到行尾，进入输入模式）
cc	等同于 0c$（删除整行，进入输入模式）
x	等同于 dl（向前删除一个字符）
X	等同于 dh（向后删除一个字符）
k 或 -	光标向后移动一行
j 或 +	光标向前移动一行
G	移动到指定行
/*string*	向前搜索 *string*
?*string*	向后搜素 *string*
n	重复向前搜索
N	重复向后搜索
f *x*	将光标向右移动到 *x* 下一次出现的位置
F *x*	将光标向左移动到 *x* 上一次出现的位置
t *x*	将光标向右移动到 *x* 下一次出现的位置，然后回退一个空格

命令	含义
T *x*	将光标向左移动到 *x* 上一次出现的位置，然后前进一个空格
;	重复最近一次字符查找命令
,	在反方向上重复最近一次字符查找命令
\	执行文件名补全
*	（在命令行上）执行通配符扩展
\=	执行通配符扩展（作为输出列表）
~	反转当前字符的大小写
\	追加上一个命令的最后一个单词，进入输入模式
Ctrl-L	开始新行并在其上重新绘制当前行
#	将 #（注释字符）置于行首并将其送入历史记录

A.20 ASCII编码表

我们钟爱的不少计算机图书中都有一份 ASCII 图表（参见表 A-25）。即便是在 GUI 和 Web 服务器时代，你也会惊讶地发现自己仍需要时不时地从中查找某个字符。在使用 tr 命令或查找一些特殊的转义字符序列时，ASCII 显然仍有用武之地。

表A-25：ASCII编码表

十进制	八进制	十六进制	ASCII	十进制	八进制	十六进制	ASCII	十进制	八进制	十六进制	ASCII
0	000	00	^@	15	017	0f	^O	30	036	1e	^^
1	001	01	^A	16	020	10	^P	31	037	1f	^_
2	002	02	^B	17	021	11	^Q	32	040	20	``
3	003	03	^C	18	022	12	^R	33	041	21	!
4	004	04	^D	19	023	13	^S	34	042	22	"
5	005	05	^E	20	024	14	^T	35	043	23	#
6	006	06	^F	21	025	15	^U	36	044	24	$
7	007	07	^G	22	026	16	^V	37	045	25	%
8	010	08	^H	23	027	17	^W	38	046	26	&
9	011	09	^I	24	030	18	^X	39	047	27	'
10	012	0a	^J	25	031	19	^Y	40	050	28	(
11	013	0b	^K	26	032	1a	^Z	41	051	29	)
12	014	0c	^L	27	033	1b	^[	42	052	2a	*

十进制	八进制	十六进制	ASCII	十进制	八进制	十六进制	ASCII	十进制	八进制	十六进制	ASCII
13	015	0d	^M	28	034	1c	^\	43	053	2b	+
14	016	0e	^N	29	035	1d	^]	44	054	2c	,
45	055	2d	-	73	111	49	I	101	145	65	e
46	056	2e	.	74	112	4a	J	102	146	66	f
47	057	2f	/	75	113	4b	K	103	147	67	g
48	060	30	0	76	114	4c	L	104	150	68	h
49	061	31	1	77	115	4d	M	105	151	69	i
50	062	32	2	78	116	4e	N	106	152	6a	j
51	063	33	3	79	117	4f	O	107	153	6b	k
52	064	34	4	80	120	50	P	108	154	6c	l
53	065	35	5	81	121	51	Q	109	155	6d	m
54	066	36	6	82	122	52	R	110	156	6e	n
55	067	37	7	83	123	53	S	111	157	6f	o

十进制	八进制	十六进制	ASCII	十进制	八进制	十六进制	ASCII	十进制	八进制	十六进制	ASCII
56	070	38	8	84	124	54	T	112	160	70	p
57	071	39	9	85	125	55	U	113	161	71	q
58	072	3a	:	86	126	56	V	114	162	72	r
59	073	3b	;	87	127	57	W	115	163	73	s
60	074	3c	<	88	130	58	X	116	164	74	t
61	075	3d	=	89	131	59	Y	117	165	75	u
62	076	3e	>	90	132	5a	Z	118	166	76	v
63	077	3f	?	91	133	5b	[	119	167	77	w
64	100	40	@	92	134	5c	\	120	170	78	x
65	101	41	A	93	135	5d	]	121	171	79	y
66	102	42	B	94	136	5e	^	122	172	7a	z
67	103	43	C	95	137	5f	_	123	173	7b	{
68	104	44	D	96	140	60	`	124	174	7c	|

十进制	八进制	十六进制	ASCII	十进制	八进制	十六进制	ASCII	十进制	八进制	十六进制	ASCII
69	105	45	E	97	141	61	a	125	175	7d	}
70	106	46	F	98	142	62	b	126	176	7e	~
71	107	47	G	99	143	63	c	127	177	7f	^?
72	110	48	H	100	144	64	d				

附录 B　bash 自带的示例

bash tarball 包含了大量值得一探究竟的资料（当然了，这得等你读完本书），其中就有样例代码、示例、脚本、函数、启动文件。最简单的获取方法是访问我们提供的带有超链接的最新列表，在本书中加入该附录是为了让你有个大概印象，因为很少有人真的打开过这个 tarball 或者从源代码构建 bash。

bash文档和示例

startup-files 目录提供了许多示例，你可以将其直接放进自己的启动文件中。尤其 bash_aliases 中包含了不少有用的别名。记住，如果你要成批复制这些文件，一定得根据所用的系统做出相应的修改，因为很多路径不一样。至于如何修改文件，详见第 16 章。

functions 目录中包含了很多你可能用得上的函数定义。

basename

　　实用工具 basename，有些系统缺少该工具。

dirfuncs

　　目录操作工具。

dirname

　　实用工具 dirname，有些系统缺少该工具。

whatis

　　Tenth Edition Bourne shell 内建命令 whatis 的实现。

whence

几乎一模一样克隆了 Korn shell 的内建命令 whence。

如果用过 Korn shell，你可能会发现 kshenv 特别有用。它包含了一些常见的 Korn 功能的函数定义，如 whence、print 以及两参数的内建命令 cd。

script 目录中包含了诸多 bash 脚本示例。两个最大的脚本展示了你能够用 shell 脚本所实现的复杂操作。第一个是一款（相当有意思的）冒险游戏解释器，第二个是 C shell 解释器。其他脚本包括优先级规则、滚动文本显示、“转轮”进度显示，以及如何提示用户输入特定类型的答案。

这些脚本和函数示例不仅可以在你的环境中发挥作用，而且还提供了许多能够在本书之外学到的替代方案。我们鼓励你动手一试。

表 B-1 是 bash 4.2 的内容索引。

应自由软件基金会的要求，Chet 从最新版本的 bash 中删除了一些示例，因为对其代码来源存在一些疑问。如果感兴趣的示例不在其中，你可以在较早的版本中查找。

表B-1：bash 4.2中的文档和示例

路径	描述
./bash/ABOUT-NLS	自由翻译项目注记
./bash/AUTHORS	bash 的主要作者清单
./bash/CHANGES	版本之间的详细变更
./bash/COMPAT	bash 版本之间的不兼容之处
./bash/COPYING	GNU 通用公共许可证（各种版本）

路径	描述
./bash/INSTALL	基本安装说明
./bash/MANIFEST	bash 的主发行清单
./bash/NEWS	bash 新增功能的简短描述
./bash/NOTES	特定平台及操作注记
./bash/POSIX	bash POSIX 模式
./bash/RBASH	受限 shell
./bash/README	bash 的一般性自述文件
./bash/Y2K	Y2K 注意事项
./bash/FAQ	bash 常见问答
./bash/INTRO	bash 简介
./bash/Makefile	bash/ 文档目录的 makefile
./bash/README	bash 文档自述文件
./bash/aosa-bash	第 3 章：开源应用程序体系结构中的 Bourne-Again Shell（剪辑版）

路径	描述
./bash/aosa-bash-full	第 3 章：开源应用程序体系结构中的 Bourne-Again Shell
./bash/article	Chet 为 *The Linux Journal* 撰写的一篇文章
./bash/bash	bash 手册页
./bash/bashbug	bashbug 手册页
./bash/bashref	Bash Reference Manual
./bash/bashref_toc	老版的 Bash Reference Manual 目录
./bash/builtins	内建命令手册页，取自 bash.1
./bash/fdl	GNU 自由文档许可证
./bash/rbash	bash 受限 shell 手册页
./bash/readline	GNU `readline` 手册页
./bash/rose94	文章：“Bash, the Bourne-Again Shell”
./bash/version	bash 版本信息
./INDEX/INDEX	bash 示例索引（一部分）
./complete/bash_completion	可编程补全功能

路径	描述
./complete/bashcc-1.0.1.tar	Richard Smith 编写的 ClearCase 补全
./complete/cdfunc	用于 cd 的补全功能示例
./complete/complete	各种补全文件
./complete/complete-examples	补全示例
./complete/complete2	Ian Macdonald 编写的各种补全文件
./functions/array-stuff	各种数组函数（ashift、array_sort、reverse）
./functions/array-to-string	将数组转化成字符串
./functions/autoload	几乎与 ksh 兼容的自动加载
./functions/basename	替代 basename(1)
./functions/basename2	用于 bash/sh 的 basename(1) 和 dirname(1)，速度更快
./functions/coproc	启动、控制、终止协程
./functions/coshell	控制 shell 协程（参见 coprocess.bash）
./functions/csh-compat	C shell 兼容软件包

路径	描述
./functions/dirfuncs	*The Korn Shell* 一书中的目录操作函数
./functions/dirname	替代 dirname(1)
./functions/dirstack	*The New KornShell Command and Programming Language* 一书中的目录操作函数的另一种实现
./functions/emptydir	判断目录是否为空
./functions/exitstat	显示进程的退出状态
./functions/external	类似于 command，但强制使用外部命令
./functions/fact	一个递归阶乘函数
./functions/fstty	可以将 TERM 的变更同步到 stty(1) 和 readline 绑定的前端
./functions/func	输出参数所指定函数的定义
./functions/gethtml	从远程服务器上获取 Web 页面（相当于 bash 中的 wget(1)！）
./functions/getoptx	解析长名称选项的 getopt 函数
./functions/inetaddr	执行 Internet 地址转换（inet2hex & hex2inet）

路径	描述
./functions/inpath	如果参数位于路径中且可执行，则返回 0
./functions/isnum	测试用户输入的数字或字符值
./functions/isnum2	测试用户输入的数字，包括浮点数
./functions/isvalidip	测试用户输入的有效 IP 地址
./functions/jdate	朱利安日期（Julian date）转换函数
./functions/jj	查找正在运行的作业
./functions/keep	尝试保持某些程序在前台运行
./functions/ksh-cd	类似于 ksh 中的 cd 命令：cd [-LP] [dir [change]]
./functions/ksh-compat-test	类似 ksh 的算术测试替换
./functions/kshenv	为 bash 提供 ksh 起步环境的函数和别名
./functions/login	替代陈旧的 Bourne shell 中的内建命令 login 和 newgrp
./functions/lowercase	将文件名更改为小写字母
./functions/manpage	查找并输出手册页

路径	描述
./functions/mhfold	打印 MH 文件夹；仅在 folder(1) 不显示修改过的日期/时间时才有用
./functions/notify	作业改变状态时发出提醒
./functions/pathfuncs	与路径相关的函数（no_path、add_path、pre-path、del_path）
./functions/recurse	一个目录遍历工具
./functions/repeat2	C shell 内建命令repeat 的克隆版
./functions/repeat3	C shell 内建命令repeat 的克隆版
./functions/seq	生成从 *m* 到 *n* 的序列，*m* 默认为 1
./functions/seq2	生成从 *m* 到 *n* 的序列，*m* 默认为 1
./functions/shcat	基于 readline 的分页程序
./functions/shcat2	基于 readline 的分页程序
./functions/sort-pos-params	对位置参数进行排序
./functions/substr	模拟古老 ksh 内建命令的函数
./functions/substr2	模拟古老 ksh 内建命令的函数

路径	描述
./functions/term	以交互形式设置终端类型的 shell 函数
./functions/whatis	第 10 版 Unix sh 内建命令 whatis(1) 的实现
./functions/whence	基本上与 ksh whence(1) 兼容的命令
./functions/which	模拟 FreeBSD 中的 which(1)
./functions/xalias	将 csh 别名转换成 bash 函数
./functions/xfind	find(1) 的克隆版
./loadables/Makefile	用于可装载内建命令示例的 Makefile 示例
./loadables/Makefile.inc	用于可装载内建命令开发的 Makefile 示例
./loadables/README	自述文件
./loadables/basename	返回路径中的非目录部分
./loadables/cat	替代 cat(1)，但不包含选项——cat 原本的使用方式
./loadables/cut	替代 cut(1)
./loadables/dirname	返回路径中的目录部分

路径	描述
./loadables/finfo	输出文件信息
./loadables/getconf	POSIX.2 `getconf` 实用工具
./loadables/head	复制文件的第一部分
./loadables/hello	可装载内建命令入门示例
./loadables/id	POSIX.2 用户身份
./loadables/ln	创建链接
./loadables/loadables	所有可装载内建命令需要的包含文件
./loadables/logname	输出当前用户的登录名
./loadables/mkdir	创建目录
./loadables/mypid	添加 shell 内建命令 `$MYPID`
./loadables/necho	不解释选项或参数的 `echo`
./loadables/pathchk	检查路径名的有效性和可移植性
./loadables/print	可装载的 ksh-93 风格的内建命令 `print`
./loadables/printenv	BSD `printenv(1)` 的最小内置克隆版

路径	描述
./loadables/printf	老版的 printf
./loadables/push	还有没有人记得 TOPS-20 ？
./loadables/pushd	老版的 pushd
./loadables/realpath	规范化路径名，解析符号链接
./loadables/rmdir	删除目录
./loadables/setpgid	bash 可装载包装器，用于 setpgid 系统调用
./loadables/sleep	睡眠几分之一秒
./loadables/sprintf	老版的 sprintf
./loadables/strftime	可装载的内建 strftime(1) 接口
./loadables/sync	通过强制执行挂起的文件系统写入操作来同步磁盘
./loadables/tee	复制标准输入
./loadables/template	可装载内建命令的示例模板
./loadables/truefalse	内建命令 true 和 false
./loadables/tty	返回终端名称

路径	描述
./loadables/uname	输出系统信息
./loadables/unlink	删除目录项
./loadables/whoami	输出当前用户的用户名
./loadables/perl/Makefile	内建的 Perl 解释器的 makefile
./loadables/perl/README	演示如何将 Perl 解释器构建入 bash
./loadables/perl/bperl	内建的 Perl
./loadables/perl/iperl	Perl 解释器
./misc/aliasconv	将 csh 的别名转换成 bash 别名和函数
./misc/cshtobash	将 csh 的别名、环境变量、变量转换成 bash 中的等价对象
./misc/suncmd	SunView TERMCAP 字符串
./obashdb/PERMISSION	使用及分发许可
./obashdb/README	过时的 bash 调试器实现示例，如今可以参考 BASH with Debugger and Improved Debug Support and Error Handling

路径	描述
./obashdb/bashdb	过时的 bashdb（bash shell 调试器），如今可以参考 BASH with Debugger and Improved Debug Support and Error Handling
./scripts/adventure	bash 实现的文字冒险游戏
./scripts/bash-hexdump	bash 实现的 hexdump(1)
./scripts/bcsh	Bourne shell csh 仿真器
./scripts/cat	基于 readline 的分页程序
./scripts/center	使多行居中
./scripts/dd-ex	只使用 /bin/sh、/bin/dd、/bin/rm 的行编辑器
./scripts/fixfiles	对目录树进行递归，修复包含各种“不良”字符的文件
./scripts/hanoi	bash 实现的汉诺塔（Towers of Hanoi）
./scripts/inpath	搜索 $PATH，在其中查找与 $1 同名的文件；如果找到，则返回真
./scripts/krand	在整数范围内生成一个随机数
./scripts/line-input	GNU Bourne Again shell 的行输入例程加上终端控制原语

路径	描述
./scripts/nohup	nohup 命令的 bash 版
./scripts/precedence	测试 && 和 || 运算符的相对优先级
./scripts/randomcard	从一副牌中随机抽出一张牌
./scripts/scrollbar	显示滚动文本
./scripts/scrollbar2	显示滚动文本
./scripts/self-repro	一个会自我复制的脚本（小心！）
./scripts/showperm	将 ls(1) 输出的符号形式的权限转换为八进制形式
./scripts/shprompt	显示提示并获取满足特定条件的答案
./scripts/spin	显示一个表示进度的转轮
./scripts/timeout	给 rsh(1) 更短的超时
./scripts/timeout2	执行会超时的指定命令
./scripts/timeout3	执行会超时的指定命令
./scripts/vtree2	显示目录树，以 1k 块为单位输出磁盘占用情况

路径	描述
./scripts/vtree3	显示图形化的目录树结构
./scripts/vtree3a	显示图形化的目录树结构
./scripts/websrv	bash 实现的 Web 服务器
./scripts/xterm_title	输出 xterm 标题栏的内容
./scripts/zprintf	模拟 printf（已过时，因为它现在已经是 bash 的内建命令了）
./scripts.noah/PERMISSION	该目录中脚本的使用许可
./scripts.noah/README	自述文件
./scripts.noah/aref	伪数组（pseudoarrays）和子串索引示例
./scripts.noah/bash.sub	require.bash 所使用的库函数
./scripts.noah/bash_version	用于切分 $BASH_VERSION 的函数
./scripts.noah/meta	启用和禁止 8 位（eight-bit）readline 输入
./scripts.noah/mktmp	创建具有唯一名称的临时文件
./scripts.noah/number	一个有趣的技巧，可以将数字翻译成英文

路径	描述
./scripts.noah/prompt	一种将 $PS1 设置为预定义字符串的方法
./scripts.noah/remap_keys	bind 的前端，可以重新执行 readline 绑定
./scripts.noah/require	用于 bash 的类 Lisp require/provide 库函数
./scripts.noah/send_mail	bash 实现的 SMTP 客户端替代版
./scripts.noah/shcat	bash 实现的 cat(1) 替代版
./scripts.noah/source	source 的替代版，使用当前目录
./scripts.noah/string	string(3) 函数的 shell 脚本实现
./scripts.noah/stty	stty(1) 的前端，可以更改 readline 绑定
./scripts.noah/y_or_n_p	提示回答 yes/no/quit
./scripts.v2/PERMISSION	该目录中脚本的使用许可
./scripts.v2/README	自述文件
./scripts.v2/arc2tarz	将 arc 归档转换为压缩的 tar 归档
./scripts.v2/bashrand	具有上限和下限以及可选种子的随机数生成器

路径	描述
./scripts.v2/cal2day	将某一天转换成对应的星期几
./scripts.v2/cdhist	添加了目录栈的 cd 命令替代版
./scripts.v2/corename	告知生成什么样的核心文件
./scripts.v2/fman	更快的 man(1) 替代版
./scripts.v2/frcp	使用 ftp(1) 复制文件，但采用的是 rcp 类型的命令行语法
./scripts.v2/lowercase	将文件名更改为小写
./scripts.v2/ncp	一个更漂亮的 cp(1) 前端（包含 -i 等选项）
./scripts.v2/newext	更改一组文件的扩展名
./scripts.v2/nmv	一个更漂亮的 mv(1) 前端（包含 -i 等选项）
./scripts.v2/pages	打印文件中指定的页
./scripts.v2/pf	处理压缩文件的分页程序前端
./scripts.v2/pmtop	简化版的 top(1)，可用于SunOS 4.*x* 和 BSD/OS
./scripts.v2/ren	修改文件名中与模式匹配的部分来重命名文件

路径	描述
./scripts.v2/rename	修改与模式匹配的文件名
./scripts.v2/repeat	多次执行某个命令
./scripts.v2/shprof	bash 脚本的行分析器
./scripts.v2/untar	将 tar 归档文件（可能经过压缩）解开到目录中
./scripts.v2/uudec	对多个文件执行 uudecode
./scripts.v2/uuenc	对多个文件执行 uudecode
./scripts.v2/vtree	以可视化形式显示目录树
./scripts.v2/where	显示匹配指定模式的命令所在的位置
./startup-files/Bash_aliases	一些有用的别名（Fox）
./startup-files/Bash_profile	bash 登录 shell 的启动文件示例（Fox）
./startup-files/Bashrc	Bourne Again shell 初始化文件示例（Fox）
./startup-files/README	自述文件
./startup-files/bash-profile	bash 登录 shell 的启动文件示例(Ramey)

路径	描述
./startup-files/bashrc	Bourne Again shell 初始化文件示例（Ramey）
./startup-files/apple/README	自述文件
./startup-files/apple/aliases	masOS 系统的别名示例
./startup-files/apple/bash	用户偏好文件示例
./startup-files/apple/environment	Bourne Again shell 环境文件示例
./startup-files/apple/login	登录包装器示例
./startup-files/apple/logout	注销包装器示例
./startup-files/apple/rc	Bourne Again shell 配置文件示例
./readline/CHANGELOG	readline 变更日志
./readline/CHANGES	版本之间的变更细节
./readline/COPYING	GNU 通用公共许可证（各种版本）
./readline/INSTALL	基本安装说明

路径	描述
./readline/MANIFEST	readline 的主发行清单
./readline/NEWS	**readline** 新功能的简要描述
./readline/README	自述文件
./readline/USAGE	通过共享库链接机制正确使用 readline 的说明
./readline/Makefile	readline 库文档的 makefile
./readline/fdl	GNU 自由文档许可证
./readline/hist	readline 历史（似乎只有 RL4.3）
./readline/history	GNU History 库手册页
./readline/history_3	GNU History 库手册页
./readline/history_toc	老版的 GNU History 库
./readline/hstech	GNU History 库文档的用户界面
./readline/hsuser	GNU History 库文档的用户界面
./readline/manvers	Manuscript 版本（似乎只是 RL4.3）

路径	描述
./readline/readline	GNU readline 手册页
./readline/readline_3	readline 文档
./readline/readline_toc	老版的 GNU readline 库目录
./readline/rlman	GNU readline 库 API
./readline/rltech	GNU readline 编程
./readline/rluser	命令行编辑
./readline/rluserman	GNU readline 库用户手册
./readline/version	bash 版本信息

附录 C　命令行处理

本书已经介绍过 shell 处理输入行的各种方式，尤其是使用 read 时。我们可以将此视为 shell 命令行处理过程的一部分。本附录详细描述了命令行处理所涉及的各个步骤，以及如何通过 eval 使 bash 展开二次处理。本节内容节选自 Cameron Newham 所著的 *Learning the bash Shell, 3rd Edition*（O'Reilly 出版）一书。

C.1　命令行处理步骤

书中随处可见命令行处理的身影。我们讲过 bash 如何应对单引号（''）、双引号（""）、反斜线（\）；如何将行中的一堆字符分隔成单词，甚至通过环境变量 $IFS 来指定所使用的分隔符；如何将分隔出来的单词分配给 shell 变量（如 $1、$2……）；如何在文件或其他进程（管道）之间重定向输入和输出。要想成为真正的 shell 脚本编程专家（或者为了调试一些棘手的问题），你需要理解命令行处理过程涉及的各个步骤，尤其是这些步骤的先后顺序。

shell 从 STDIN 或脚本中读取的每一行被称为管道，因为其中包含了由零个或多个管道字符（|）分隔的一个或多个命令。图 C-1 展示了命令行处理的一系列步骤。

图 C-1：命令行处理步骤

对于读入的管道，shell 会将其分解成命令，设置管道的 I/O，然后对每个命令执行下列操作。

01. 将命令分割成由一组固定的元字符（空格、制表符、换行符、;、(、)、<、>、|、&）分隔的词法单元（token）。词法单元的类型包括单词、关键字、I/O 重定向、分号。
02. 检查每个命令的第一个词法单元是否为不带引号或反斜线的关键字。如果属于起始关键字，例如 if 或其他控制结构的开头、

function、{、(，那么该命令属于复合命令。shell 会在内部为其完成相关准备工作，读取下一个命令，并重头开始这个过程。如果关键字不是复合命令的开头（例如，then、else、do 这种属于控制结构“中间”的部分；fi、done 这种属于控制结构“结束”的部分；或者是逻辑运算符），则 shell 会提示语法错误。

03. 对照别名列表检查每个命令的第一个单词。如果有匹配，将单词替换成别名定义，然后返回第 1 步；否则，继续往下进行第 4 步。这种方式允许出现递归别名，也允许为关键字定义别名（例如，alias aslongas=while 或 alias procedure=function）。
04. 执行花括号扩展。例如，将 a{b,c} 扩展为 ab ac。
05. 如果波浪线位于单词的开头，将其替换成用户的主目录（$HOME）。例如，将 ~user 替换成 user 的主目录。
06. 对以美元符号（$）起始的表达式执行参数（变量）替换。
07. 对形如 $(*string*) 的表达式执行命令替换。
08. 对形如 $((*string*)) 的算术表达式进行求值。
09. 将命令行中经过参数替换、命令替换、算术求值得到的结果再次分割成一系列单词。这次使用 $IFS 所包含的字符作为分隔符，不再使用第 1 步中的那组元字符。
10. 对出现的 *、?、[] 执行路径扩展，也就是通配符扩展。
11. 将第一个单词作为命令，按照下列顺序查找其来源：先按照函数名，再作为内建命令，接着作为 $PATH 所包含目录中的文件。
12. 设置好 I/O 重定向和其他事宜后，执行该命令。

步骤着实不少，甚至这还不是全部！在继续往下进行之前，我们应该先用一个示例来明晰这个过程。假设要执行下列命令：

```
alias ll="ls -l"
```

再进一步假设用户 alice 的主目录（/home/alice）有一个名为 .hist537 的文件，还有一个双美元符号变量 $$，其值为 2357（记住，$$ 保存的是进程 ID，这个数字在所有运行的进程中是唯一的）。

现在我们来看看 shell 是如何处理下列命令的。

```
ll $(type -path cc) ~alice/.*$(($$%1000)
```

以下是所发生的操作：

01. ll $(type -pathcc) ~alice/.*$(($$%1000)) 将其分割成一系列单词。
02. ll 并非关键字，因此第 2 步什么都不做 [1]。
03. ls -l $(type -path cc) ~alice/.*$(($$%1000)) 将别名 ll 替换成 ls -l。然后 shell 重复第 1~3 步；第 2 步会将 ls -l 分割为 2 个单词。
04. ls-l$(type -pathcc) ~alice/.*$(($$%1000)) 什么都不做。
05. ls -l $(type -path cc) /home/alice/.*$(($$%1000)) 将~alice 扩展成 /home/alice。
06. ls -l $(type -path cc) /home/alice/.*$((2537%1000)) 将 $$ 替换成 2537。
07. ls -l /usr/bin/cc/home/alice/.*$((2537%1000)) 对 type -path cc 执行命令替换。
08. ls -l /usr/bin/cc/home/alice/.*537 对算术表达式 2537%1000 求值。
09. ls-l /usr/bin/cc/home/alice/.*537 什么都不做。
10. ls -l /usr/bin/cc/home/alice/.hist537 将通配符表达式 .*537 替换成文件名。
11. 在 /usr/bin 中找到命令 ls。
12. 执行包含选项 -l 和两个参数的 /usr/bin/ls。

[1]这里提到的第 *x* 步对应前文中相应编号的步骤。——译者注

尽管这些步骤相当直观，但只是部分而已。还有 5 种方式可以改变该流程：引用；使用 command、builtin、enable；使用高级命令 eval。

C.2 引用

你可以将引用视为跳过上述 12 个步骤中某些步骤的一种方式。

- 单引号（''）能够完全绕过包括别名在内的第 1~10 步。单引号中的所有字符全部保持不变。单引号中不能再出现单引号，就算在其之前加上反斜线（\）也没用。
- 双引号（""）能够绕过第 1~4 步以及第 9~10 步。也就是说，它会忽略管道字符、别名、波浪号替换、通配符扩展以及通过分隔符（如空白字符）将双引号中的内容分割成一系列单词。但是，双引号仍会执行参数替换、命令替换以及算术表达式求值。双引号中可以出现双引号，在其之前加上反斜线（\）即可。另外，必须使用反斜线对具有特殊含义的 $、'（古老的命令替换分隔符）、\ 进行转义。

表 C-1 给出了一个简单的示例，其中演示了引用是如何工作的。假设执行的语句是 person=hatter，用户 alice 的主目录是 /home/alice。

表C-1：单引号和双引号的用法示例

表达式	值
$pers-on	hatter
"$per-son"	hatter
\$person	$person
'$person'	$person
"'$person'"	'hatter'
~alice	/home/alice

表达式	值
"~alice"	~alice
'~alice'	~alice

如果不确定在 shell 编程的某个时刻到底是该用单引号还是双引号，最稳妥的做法就是使用单引号，除非你需要执行参数、命令或算术替换操作。

C.3 eval

我们知道引用能够跳过命令行处理过程中的某些步骤，而 eval 命令可以让你重新经历整个过程。重复进行命令行处理看似奇怪，但实则威力强大：它允许你在脚本中动态创建命令字符串，然后将其传给 shell 执行。这意味着脚本被赋予了“智慧”，可以在运行期间改变自身的行为。

eval 语句告诉 shell 获取 eval 的参数，并再次以命令行处理的所有步骤运行它们。为了帮助你理解 eval 的含义，我们从一个简单的例子开始，然后一步步实现动态构建并执行命令。

eval ls 会将字符串 ls 传给 shell 执行；shell 会输出当前目录下的文件列表。这非常简单，字符串 ls 并没有什么东西需要传经命令行处理步骤两次。但思考以下示例：

```
listpage="ls | more"
$listpage
```

shell 并不会生成分页文件列表，而是将 | 和 more 视为 ls 的参数，ls 会抱怨没有这种名称的文件存在。为什么？因为管道符号在第 6 步中才会被识别为管道，shell 求值变量是发生在查找管道符号之后。第 9 步才会解析变量扩展的结果。因此，shell 将 | 和

more 视为 ls 的参数，ls 则在当前目录中寻找名为 | 和 more 的文件！

现在考虑用 eval $listpage 替代 $listpage。进行到最后一步时，shell 会执行包含参数 ls、|、more 的 eval 命令。这使得 shell 重新回到第 1 步来处理由这些参数组成的行。第 2 步中发现了 | 并将行分割成两个命令：ls 和 more。每个命令按照正常（且简单的）方式处理。结果就是逐页输出当前目录中的文件列表。

现在你可能开始意识到 eval 的威力了。这是一种高级特性，需要具备一定的程序设计头脑才能发挥出最大的功效。这甚至还有点人工智能的意味，可以让你编写出能够“编写”并执行其他程序的程序。你未必会在日常 shell 编程中用到 eval，但它值得你花时间去理解其功用。

附录 D　修订控制

修订控制系统（revision control system）不仅可以让时光倒流，而且还能查看时间线中的各个时间点发生了什么变化。也称之为版本化（versioning）或版本控制系统（version control system），其实这个名称在技术上更为准确。你可以通过此类系统维护项目文件**仓库**，跟踪其中文件的改动及其原因。现代修订控制系统允许多个开发人员同时处理同一项目，甚至同一文件。

修订控制系统不仅对于现代软件开发工作至关重要，而且也能在许多其他领域发挥作用，例如，编写文档、跟踪系统配置（如 /etc/），甚至是写书。在撰写本书时，我们使用 Git 对其进行修订控制。第 1 版时使用的是 Subversion。

修订控制系统的一些有用特性如下。

- 几乎不会丢失工作成果，尤其是正确备份仓库的情况下。
- 有利于变更控制，鼓励写明为什么要做出变更。
- 允许不同位置的人们合作展开项目并同步他人的变更，不会因为覆盖了对方的文件或发送大量不可读的电子邮件而丢失数据。
- 允许个人更改工作地点，不会因此丢失工作成果或覆盖其他工作地点做出的变更。
- 允许你轻松地撤销更改或查看两个修订版之间到底有什么改动（二进制文件除外）。如果按照有效的日志记录实践操作，你甚至还能知道发生变更的原因。

CVS 和 Subversion 这种系统还允许某种形式的关键字扩展，使得你可以在非二进制文件中嵌入修订元数据（revision metadata）。

有许多免费版和商业版的修订控制系统，如果你正在阅读本书，则应该选用一种！如果对某种系统有所耳闻，用那种就行；如果所在公司有标准工具，照用便是。如果实在挑不出来，可以使用 Git、Bazaar 或 Mercurial。除非别无选择，否则别用 Subversion、CVS、RCS 等任何陈旧的系统。本附录将简要介绍 Git、Bazaar、Mercurial、

Subversion 的优缺点和基本用法，这些工具都可用于所有主流现代操作系统。但在此之前，我们要先提供一些背景知识。

首先，现代修订控制系统是**分布式的**，而早先的（如 Subversion 和 CVS）是**集中式的**。这种主要且根本上的差异带来了一些重要的意义。老派的集中式系统中配有中央服务器，顾名思义，它通常是由 IT 部门负责维护和备份的，这种做法很好。你得连接到服务器上才能干活，而效果未必尽如人意，因为这可要比本地磁盘访问速度慢多了，即便如此，有时候你连服务器可能都连不上，比如在旅行途中。在这种系统中，你只能检出部分仓库（repo），因此整个公司通常会有一个大仓库，而你只是检出并处理所需的部分。其次，系统还会执行我们提到过的关键字扩展，D.5 节将展示这一点。最后，**提交**即**发布**，这也许会被视为此类系统的特性或 bug，就算不是 bug，起码也是不受欢迎的。

另外，分布式系统没有中央服务器，但按照惯例，通常将某个副本指定为“可信来源”（source of truth）。你所使用的仓库是一个完整的副本，它和其他人的副本没有优劣之分。相较于只能检出部分仓库，这是一个主要变化。此类系统并不执行关键字扩展，提交和发布也不是一回事，后者需要额外的 push 步骤以及到远程仓库的网络连接。除了 push/pull 之外，其余的全都是本地操作，速度非常快。

> 使用 CVS 或 Subversion 时不需要考虑备份。如果提交了代码，IT 部分可能已经在其他地方做好了备份（提交 == 发布）。但在现代分布式系统中，通常就不是这样了。这类系统中拥有本地仓库，提交的代码仍然在本地（提交 != 发布），直到你使用 push 操作将其推送（发布）到远程仓库。如果从不推送，那就只有一份本地副本，因此务必做好备份工作！随后我们会介绍一款不错的备份工具 etckeeper。本地仓库位于 /etc/，如果不小心执行了 rm -rf /etc，随之烟消云散的还有你的仓库。

另一个关键点是 Git 已经毫无疑问地赢得了这场战争，“人人”都在使用它。不过，也不能说是每个人，如果还在阅读本书，那你可能就是个例外。但使用 Git 的人确实非常多，可以说很大一部分原因是 GitHub。那我们为什么不顺势而为？很高兴你问到这个问题。

如果你是一名从事大型 [1] 项目的专职开发人员，已经用上了 Git，那非常好。但如果你是一名轻度用户，比如是手里有一堆脚本的系统管

理员，那 Git 就没那么好了[2]。Git 对用户已经比以前友善了，但用起来还是太过复杂，我们尚未见过有什么不错的心智模型能够描述其工作方式。很多时候，你必须深入理解 Git，才能对其委以重任。这个过程并不美好。Git 由剃刀和链锯造就而成，快如疾风，威力无比，但同时又充满了危险，你会误伤到自己。例如，Git 的历史记录有很强的延展性，这是特性，而非 bug。它使用散列和日期代替了易读的修订号，尽管这么做有充分的理由，但真的很不方便。最后，Git 的“索引”也与众不同，其他常见工具都没这样做，但它确实提供了一种非常方便的技巧，你可以在其中作一番意识流更改（stream-of-consciousness change），随后使用 `git add -p` 或 `git commit -p` 在逻辑块中提交它们。我们认为 Git 是一个极为强大的工具，不适合初学者或轻度用户使用。但是……它无处不在，每个人都在使用，这也是一个强有力的证据。

> [1]我们应该说“Web 规模”，这样才符合当下的流行用语。

> [2]参见 Steve Bennett 所写的“10 Things I Hate About Git”一文。

如果你对版本控制的历史感兴趣，可以阅读 Eric Raymond 所写的“Understanding Version-Control Systems”一文，从中了解大量细节。要想见识令人瞠目结舌的应用示例和超酷的东西，可以查阅 Unix 历史库（Unix History Repository）。

如果自己打算开始使用修订控制系统，那就动手吧。但如果打算在团队中使用，你必须得先决定：

- 要使用哪种系统或产品；
- 更新、提交、标记以及分支的策略；
- 中央仓库（以及完善的备份！）的位置（如果适用的话）；
- 仓库中的项目或目录的结构（如果适用的话）。

本附录足以带你入门，但也仅限于浅尝辄止而已。要想更深入了解修订控制和各个系统的完整细节，可以参阅由 Jon Loeliger 和 Matthew McCullough 合著的 *Version Control with Git, 2nd Edition*（O’Reilly 出版）以及 C. Michael Pilato、Ben Collins-Sussman、Brian Fitzpatrick 合著的 *Version Control with Subversion, 2nd Edition*（O’Reilly 出版）。两者都对一般概念作

了很好的介绍，但由于 Subversion 的多项目性质（multiproject nature），第二本书更详细地讲述了仓库结构。这两本书还涵盖了修订控制策略。如果你所在的公司已经制订了变更控制或相关策略，那就继续沿用。如果没有，我们建议你尽早并经常提交和更新。如果是团队作战，我们强烈建议你阅读本附录中列出的一些参考图书，仔细规划出一套策略。从长远来看，这会节省大量时间。

D.1 参考

- Eric Raymond 所写的“Understanding Version-Control Systems”
- Unix 历史库
- BetterExplained 网站上的“A Visual Guide to Version Control”
- W. Curtis Preston 所著的 *Backup & Recovery*（O'Reilly 出版）
- reposurgeon，一款可在系统之间转换的工具

D.2 Git

Git 是修订控制事实上的领导者，使用它的项目和用户可能比其他系统的总和还多。如果你也选用 Git，准备好多逛逛 Google 吧。

在当时的版本控制系统厂商修改了授权 [3] 后，Git 最初是由 Linus Torvalds 亲自为 Linux 内核项目编写的，在很短时间内就力压其他同类系统。Git 的设计很大程度上受到了 Torvalds 多年在大规模全球分布式项目管理经验的影响，是硬核程序员献给硬核程序员的作品。它极其强大且灵活，但往往过于复杂。对于专业开发人员而言，其学习曲线无疑是值得的，但更多的轻度或普通用户可能会遇到困难。

[3] Linux 内核项目组于 2002 年开始启用一个专有的分布式版本控制系统 BitKeeper 来管理和维护内核代码。2005 年，开发 BitKeeper 的商业公司同 Linux 内核开源社区的合作关系结束，于是收回了免费使用 BitKeeper 的授权。——译者注

D.2.1 优点

- 极为流行，到处都有人用。
- 极其快速、强大、灵活。
- `git add -p` 和 `git commit -p` 说明了如何编写代码。
- 拥有 GitHub、GitLab 等站点。
- 历史记录颇具延展性。

D.2.2 缺点

- 你执行的操作可能会造成数据丢失！
- 除了最基本的任务，其他任务都比其他工具难于理解和使用。
- 不一致且复杂的命令行用法。
- 历史记录颇具延展性。
- 使用散列和日期代替了易读的修订号。

D.2.3 示例

这个示例并不适合于企业或多用户访问（更多信息参见 D.2.4 节）。这些只是基础操作，但是可以帮助你入门，如果需要的话，不妨从这里开始。

如果尚未安装 Git，可以使用操作系统中惯用的软件包管理器先行安装。

`git` 命令（不加选项）、`git help`、`git help` *`command`* 都能够给出有用的帮助和提示。

在你的机器上配置 Git（查看 `init` 命令创建的 ~/.gitconfig 和 .git/config）。

```
/home/jp$ git config --global user.name "JP Vossen"
/home/jp$ git config --global user.email "jp@jpsdomain.org"
/home/jp$ git config --global core.pager "less -R"
/home/jp$ git config --global color.ui true
```

如果没有像这里展示的那样设置姓名和电子邮件，你可能会收到抱怨信息，其中非常清楚地指明了该怎么做。

你也可以考虑：

```
/home/jp$ git config --global alias.co checkout
/home/jp$ git config --global alias.br branch
/home/jp$ git config --global alias.ci commit
/home/jp$ git config --global alias.st status
/home/jp$ git config --global alias.last 'log -1 HEAD'
```

在主目录中创建供个人使用的新仓库：

```
/home/jp$ git init myrepo
Initialized empty Git repository in /home/jp/myrepo/.git/
```

创建新脚本并提交：

```
/home/jp$ cd myrepo

/home/jp/myrepo$ cat << EOF > hello
> #!/bin/bash -
> echo 'Hello World!'
> EOF

/home/jp/myrepo$ chmod +x hello

/home/jp/myrepo$ git add hello

/home/jp/myrepo$ git commit -m 'Initial import of shell script'
[master (root-commit) 62cb49e] Initial import of shell script
 1 file changed, 2 insertions(+)
 create mode 100755 hello
```

git add 不同于其他工具中的 add ！在别的工具中，一旦添加了文件，对该文件的变更总是会被提交。但在 Git 中，添加意味着“将变更加入索引”。因此，如果你做了变更，将其加入，然后做出另一处变更，那么第二次变更并不在索引内，也不会被提交，除非你再添加一次，或者使用 -a 提交。这听起来很烦人，而且还是基本用途，但它是整个“索引”概念的一部分，这使得其他一些简洁操作成为可能，随后我们就会看到。

如果没有使用 -m *message* 选项，那么会弹出一个编辑器，你可以在其中创建提交日志。至于出现哪个编辑器，该如何更改，这取决于你的操作系统和发行版。要想修改编辑器，请查询相应的文档。

检查沙盒的状态：

```
/home/jp/myrepo$ git status
On branch master
nothing to commit, working directory clean
```

向修订控制中添加新脚本：

```
/home/jp/scripts$ cat << EOF > mcd
> #!/bin/bash -
> mkdir -p "$1"
> cd "$1"
> EOF

/home/jp/myrepo$ chmod +x mcd

/home/jp/myrepo$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        mcd

nothing added to commit but untracked files present (use "git add"
to track)

/home/jp/myrepo$ git add mcd

/home/jp/myrepo$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   mcd

/home/jp/myrepo$ git commit -m 'Added new script: mcd'
[master a2c254d] Added new script: mcd
 1 file changed, 3 insertions(+)
 create mode 100755 mcd
```

做出变更，然后检查差异：

```
/home/jp/myrepo$ vi hello

/home/jp/myrepo$ git diff
      1 diff --git a/hello b/hello
      2 index 353223d..f36eea4 100644
      3 --- a/hello
      4 +++ b/hello
      5 @@ -1,2 +1,2 @@
      6  #!/bin/bash -
      7 -echo 'Hello World!'
      8 +echo 'Hello Mom!'

/home/jp/myrepo$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)

        modified:   hello

no changes added to commit (use "git add" and/or "git commit -a")
```

如果执行 git diff 时屏幕上出现一堆乱七八糟的转义字符，你可以尝试设置 git config --global core.pager "less -R"。

用 -a 提交变更，避开 git add hello：

```
/home/jp/myrepo$ git commit -a -m 'Fine tuning'
[master e1f0b2f] Fine tuning
 1 file changed, 1 insertion(+), 1 deletion(-)
```

查看仓库历史或单个文件：

```
/home/jp/myrepo$ git log
      1 commit e1f0b2f8e5c489d8c9112014cf494773712786b0
      2 Author: JP Vossen <jp@jpsdomain.org>
      3 Date:   Sun Jul 3 22:56:38 2016 -0400
      4
      5     Fine tuning
      6
      7 commit a2c254d61e95eb4719746f196b66019446061d51
```

```
      8 Author: JP Vossen <jp@jpsdomain.org>
      9 Date:   Sun Jul 3 22:52:36 2016 -0400
     10
     11     Added new script: mcd
     12
     13 commit 62cb49ee962d929122051c421128fea95d571ebb
     14 Author: JP Vossen <jp@drake.jpsdomain.org>
     15 Date:   Sun Jul 3 22:44:15 2016 -0400
     16
     17     Initial import of shell script

/home/jp/myrepo$ git log hello
      1 commit e1f0b2f8e5c489d8c9112014cf494773712786b0
      2 Author: JP Vossen <jp@jpsdomain.org>
      3 Date:   Sun Jul 3 22:56:38 2016 -0400
      4
      5     Fine tuning
      6
      7 commit 62cb49ee962d929122051c421128fea95d571ebb
      8 Author: JP Vossen <jp@drake.jpsdomain.org>
      9 Date:   Sun Jul 3 22:44:15 2016 -0400
     10
     11     Initial import of shell script
```

还原到旧版。也可以用其他方法实现，这取决于你在工作目录中做了哪些别的变更，下面这种方法也许并不直观，却非常简单。

```
/home/jp/myrepo$ git checkout
62cb49ee962d929122051c421128fea95d571ebb hello

/home/jp/myrepo$ cat hello
#!/bin/bash -
echo 'Hello World!'

/home/jp/myrepo$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   hello

/home/jp/myrepo$ git diff
```

等等！我们做了变更，从 `git status` 的输出中也能看出来，但 `diff` 没有反应？这是为什么？因为它已经执行了 `git add`，变更被**暂存**或**缓存**了。

```
/home/jp/myrepo$ git diff --cached
      1 diff --git a/hello b/hello
      2 index f36eea4..353223d 100755
      3 --- a/hello
      4 +++ b/hello
      5 @@ -1,2 +1,2 @@
      6  #!/bin/bash -
      7 -echo 'Hello Mom!'
      8 +echo 'Hello World!'
```

我们之前可是警告过你的……

D.2.4 参考

- man git
- git help
- 维基百科（Git）
- Scott Chacon 与 Ben Straub 合著的 *Pro Git, 2nd Edition*（Apress 出版）
- Jon Loeliger 与 Matthew McCullough 合著的 *Version Control with Git, 2rd Edition*（O'Reilly 出版）
- Steve Bennett 所写的“10 Things I Hate About Git”
- BetterExplained 网站上的“Aha! Moments When Learning Git”
- The EasyGit wrapper
- 16.16 节

D.3 Bazaar

Bazaar 是 Canonical 公司 [4] 对 Git 的回应，可惜它已经输掉了这场战争，目前基本上处于维护模式。

> [4]Canonical Ltd. 是一家私人公司，由南非的企业家 Mark Richard Shuttleworth 创建，主要为了促进开源软件项目发展，其支持并创建的项目之一就是 Ubuntu Linux 发行版。——译者注

D.3.1 优点

- 不是 Git。
- 对用户非常友好，配有高水平的文档。
- 跨平台（Python），包括多种 GUI 工具：QBzr（Qt）、Loggerhead（web）等。
- 使用递增的整数修订编号。
- 历史记录不可变。
- 拥有 Launchpad。

D.3.2 缺点

- 不是 Git。
- 输掉了修订控制系统之战，奄奄一息。
- 没 Git 速度快，但无伤大雅。
- 没 Git 有名气。

D.3.3 示例

这个示例并不适合企业或多用户访问（更多信息参见 D.3.4 节），仅用来展示基础操作有多简单轻松。

如果尚未安装 Bazaar，可以使用操作系统中惯用的软件包管理器先行安装。

`bzr` 命令（不加选项）、`bzr help`、`bzr help` *`command`* 都能够给出有用的帮助和提示。

在主目录中创建供个人使用的新仓库：

```
/home/jp$ bzr init myrepo
Created a standalone tree (format: 2a)
```

创建新脚本并提交：

```
/home/jp$ cd myrepo

/home/jp/myrepo$ cat << EOF > hello
> #!/bin/bash -
> echo 'Hello World!'
> EOF
```

```
/home/jp/myrepo$ chmod +x hello

/home/jp/myrepo$ bzr add hello
adding hello

/home/jp/myrepo$ bzr commit -m 'Initial import of shell script'
Committing to: /home/jp/myrepo/
added hello
Committed revision 1.
```

如果没有使用 -m *message* 选项，那么会弹出一个编辑器，你可以在其中创建提交日志。至于出现哪个编辑器，该如何更改，这取决于你的操作系统和发行版。要想修改编辑器，请查询相应的文档。

检查沙盒的状态：

```
/home/jp/myrepo$ bzr status
```

向修订控制中添加新脚本：

```
/home/jp/scripts$ cat << EOF > mcd
> #!/bin/bash -
> mkdir -p "$1"
> cd "$1"
> EOF

/home/jp/myrepo$ chmod +x mcd

/home/jp/myrepo$ bzr status
unknown:
  mcd

/home/jp/myrepo$ bzr add mcd
adding mcd

/home/jp/myrepo$ bzr status
added:
  mcd

/home/jp/myrepo$ bzr commit -m 'Added new script: mcd'
Committing to: /home/jp/myrepo/
added mcd
Committed revision 2.
```

做出变更，然后检查差异：

```
/home/jp/myrepo$ vi hello

/home/jp/myrepo$ bzr diff
=== modified file 'hello'
--- hello       2016-07-04 03:26:32 +0000
+++ hello       2016-07-04 03:28:11 +0000
@@ -1,2 +1,2 @@
 #!/bin/bash -
-echo 'Hello World!'
+echo 'Hello Mom!'

/home/jp/myrepo$ bzr status
modified:
  hello
```

提交变更：

```
/home/jp/myrepo$ bzr commit -m 'Fine tuning'
Committing to: /home/jp/myrepo/
modified hello
Committed revision 3.
```

查看仓库历史或单个文件：

```
/home/jp/myrepo$ bzr log
 ------------------------------------------------------------
revno: 3
committer: JP Vossen <jp@ringo.jpsdomain.org>
branch nick: myrepo
timestamp: Sun 2016-07-03 23:28:48 -0400
message:
  Fine tuning
 ------------------------------------------------------------
revno: 2
committer: JP Vossen <jp@ringo.jpsdomain.org>
branch nick: myrepo
timestamp: Sun 2016-07-03 23:27:50 -0400
message:
  Added new script: mcd
 ------------------------------------------------------------
revno: 1
committer: JP Vossen <jp@ringo.jpsdomain.org>
branch nick: myrepo
timestamp: Sun 2016-07-03 23:26:32 -0400
message:
```

```
  Initial import of shell script

/home/jp/myrepo$ bzr log hello
 ------------------------------------------------------------
revno: 3
committer: JP Vossen <jp@ringo.jpsdomain.org>
branch nick: myrepo
timestamp: Sun 2016-07-03 23:28:48 -0400
message:
  Fine tuning
 ------------------------------------------------------------
revno: 1
committer: JP Vossen <jp@ringo.jpsdomain.org>
branch nick: myrepo
timestamp: Sun 2016-07-03 23:26:32 -0400
message:
  Initial import of shell script
```

还原到旧版：

```
/home/jp/myrepo$ bzr revert -r1 hello
 M  hello

/home/jp/myrepo$ bzr status
modified:
  hello

/home/jp/myrepo$ bzr diff
=== modified file 'hello'
--- hello       2016-07-04 03:28:48 +0000
+++ hello       2016-07-04 03:29:44 +0000
@@ -1,2 +1,2 @@
 #!/bin/bash -
-echo 'Hello Mom!'
+echo 'Hello World!'
```

D.3.4 参考

- man bzr
- bzr help
- 维基百科（Bazaar）
- Janos Gyerik 所著的 *Bazaar Version Control*（Packt 出版）
- 16.16 节

D.4 Mercurial

Mercurial 与 Git 出于相同的原因诞生于同一时间，但从未得到过太多关注。

D.4.1 优点

- 不是 Git。
- 对用户非常友好，配有高水平的文档。
- 跨平台（Python），包括多种 GUI 工具。
  - 内建了 Web 服务器（`hg serve`，然后访问 http://localhost:8000/）。
- 使用递增的整数修订编号 + 十六进制 ID。
  - 十六进制 ID 是唯一的，在所有的仓库克隆中保持一致，而整数修订编号不然。
- 操作历史不可变。
- 拥有 Atlassian。

D.4.2 缺点

- 不是 Git。
- 输给了 Git，但比 Bazaar 更活跃。
- 没 Git 速度快，但无伤大雅。
- 没 Git 有名气。

D.4.3 示例

这个示例并不适合企业或多用户访问（更多信息参见 D.4.4 节），仅用于展示基础操作有多简单轻松。

如果尚未安装 Mercurial，可以使用操作系统中惯用的软件包管理器先行安装。

`hg` 命令（不加选项）、`hg help`、`hg help` *`command`* 都能够给出有用的帮助和提示。在主目录中创建供个人使用的新仓库：

```
/home/jp$ hg init myrepo
```

创建新脚本并提交：

```
/home/jp$ cd /myrepo

/home/jp/myrepo$ cat << EOF > hello
> #!/bin/bash -
> echo 'Hello World!'
> EOF

/home/jp/myrepo$ chmod +x hello

/home/jp/myrepo$ hg add hello

/home/jp/myrepo$ hg commit -m 'Initial import of shell script'
```

如果没有使用 -m *message* 选项，那么会弹出一个编辑器，你可以在其中创建提交日志。至于出现哪个编辑器，该如何更改，这取决于你的操作系统和发行版。要想修改编辑器，请查询相应的文档。

检查沙盒的状态：

```
/home/jp/myrepo$ hg status
```

向修订控制中添加新脚本：

```
/home/jp/scripts$ cat << EOF > mcd
> #!/bin/bash -
> mkdir -p "$1"
> cd "$1"
> EOF

/home/jp/myrepo$ chmod +x mcd

/home/jp/myrepo$ hg status
? mcd

/home/jp/myrepo$ hg add mcd

/home/jp/myrepo$ hg status
A mcd
```

```
/home/jp/myrepo$ hg commit -m 'Added new script: mcd'
```

做出变更，然后检查差异：

```
/home/jp/myrepo$ vi hello

/home/jp/myrepo$ hg diff
diff -r 663ba0ec20f5 hello
--- a/hello     Sun Jul 03 23:38:54 2016 -0400
+++ b/hello     Sun Jul 03 23:39:15 2016 -0400
@@ -1,2 +1,2 @@
 #!/bin/bash -
-echo 'Hello World!'
+echo 'Hello Mom!'

/home/jp/myrepo$ hg status
M hello
```

提交变更：

```
/home/jp/myrepo$ hg commit -m 'Fine tuning'
```

查看仓库历史或单个文件：

```
/home/jp/myrepo$ hg log
changeset:   2:c88ab0cbfcda
tag:         tip
user:        JP Vossen <jp@jpsdomain.org>
date:        Sun Jul 03 23:39:38 2016 -0400
summary:     Fine tuning

changeset:   1:663ba0ec20f5
user:        JP Vossen <jp@jpsdomain.org>
date:        Sun Jul 03 23:38:54 2016 -0400
summary:     Added new script: mcd

changeset:   0:38ab693c1c72
user:        JP Vossen <jp@jpsdomain.org>
date:        Sun Jul 03 23:38:03 2016 -0400
summary:     Initial import of shell script

/home/jp/myrepo$ hg log hello
changeset:   2:c88ab0cbfcda
tag:         tip
user:        JP Vossen <jp@jpsdomain.org>
```

```
date:        Sun Jul 03 23:39:38 2016 -0400
summary:     Fine tuning

changeset:   0:38ab693c1c72
user:        JP Vossen <jp@jpsdomain.org>
date:        Sun Jul 03 23:38:03 2016 -0400
summary:     Initial import of shell script
```

还原到旧版：

```
/home/jp/myrepo$ hg revert -r 1 hello

/home/jp/myrepo$ cat hello
#!/bin/bash -
echo 'Hello World!'

/home/jp/myrepo$ hg status
M hello
```

D.4.4 参考

- man hg
- hg help
- 维基百科（Mercurial）
- 16.16 节

D.5 Subversion

根据 Subversion 网站的说法：“Subversion 项目的目标是构建一个**版本控制系统**，足以在开源社区中替代 CVS。”说的非常明白了。

D.5.1 优点

- 不是 Git。
- 比 CVS 和 RCS 更新。
- 比 CVS 更简单，也更易于理解和使用（历史包袱轻）。
- 原子提交（atomic commit）意味着整个提交操作要么成功，要么失败，并且可以轻松地将整个项目的状态作为单个修订进行跟踪。

- 易于访问远程仓库。
- 在保留历史记录的同时可以轻松地重命名文件和目录。
- 能够轻松地处理二进制文件（没有原生的 diff 支持）和其他对象（如符号链接）。
- 中央仓库 hacking [5] 得到了更多官方支持，但有点难度。

[5]在技术领域中，hack 作动词（hacking）意为创造性地解决难题，作名词（hack）则是指不符合常规但有效的手段。目前这两个意思均尚未有恰当的中文译法，故保留原词。——译者注

D.5.2 缺点

- 不是 Git。
- 技术陈旧，修订控制已经转向了分布式模型。
- 大量的依赖关系导致从头构建或安装非常复杂。尽可能使用操作系统自带的版本。

SVN 通过仓库跟踪修订，这意味着每个提交都有自己的内部 SVN 修订号。因此，个人的连续提交未必对应连续的修订版号，因为其他人提交变更（可能是其他项目）也会增加全局存储库修订号。

D.5.3 示例

这个示例并不适合企业或多用户访问（更多信息参见 D.5.4 节），仅用于展示基础操作有多简单轻松。本例还将环境变量 EDITOR 设置为 nano（export EDITOR='nano --smooth --const --nowrap --suspend'），有些用户觉得它比默认的 vi 更友好。

svn help 和 svn help help 命令非常实用。

在主目录中创建供个人使用的新仓库：

```
/home/jp$ svnadmin --fs-type=fsfs create /home/jp/svnroot
```

创建新项目并将其导入：

```
/home/jp$ cd /tmp

/tmp$ mkdir -p -m 0700 scripts/trunk scripts/tags scripts/branches

/tmp$ cd scripts/trunk

/tmp/scripts/trunk$ cat << EOF > hello
> #!/bin/sh
> echo 'Hello World!'
> EOF

/tmp/scripts/trunk$ cd ..

/tmp/scripts$ svn import /tmp/scripts
file:///home/jp/svnroot/scripts

  GNU nano 1.2.4                  File: svn-commit.tmp

Initial import of shell scripts
--This line, and those below, will be ignored--

A   .
                                        [ Wrote 4 lines ]

Adding         /tmp/scripts/trunk
Adding         /tmp/scripts/trunk/hello
Adding         /tmp/scripts/branches
Adding         /tmp/scripts/tags

Committed revision 1.
```

检出项目并更新：

```
/tmp/scripts$ cd

/home/jp$ svn checkout file:///home/jp/svnroot/scripts
A  scripts/trunk
A  scripts/trunk/hello
A  scripts/branches
A  scripts/tags
Checked out revision 1.

/home/jp$ cd scripts

/home/jp/scripts$ ls -l
total 12K
drwxr-xr-x  3 jp jp 4.0K Jul 20 01:12 branches/
drwxr-xr-x  3 jp jp 4.0K Jul 20 01:12 tags/
drwxr-xr-x  3 jp jp 4.0K Jul 20 01:12 trunk/
```

```
/home/jp/scripts$ cd trunk/

/home/jp/scripts/trunk$ ls -l
total 4.0K
-rw-r--r--  1 jp jp 30 Jul 20 01:12 hello

/home/jp/scripts/trunk$ echo "Hi Mom..." >> hello
```

检查沙盒的状态。注意 svn status 命令与本附录先前“CVS”部分中的 cvs -qn update 的相似性。

```
/home/jp/scripts/trunk$ svn info
Path: .
URL: file:///home/jp/svnroot/scripts/trunk
Repository UUID: 29eeb329-fc18-0410-967e-b075d748cc20
Revision: 1
Node Kind: directory
Schedule: normal
Last Changed Author: jp
Last Changed Rev: 1
Last Changed Date: 2006-07-20 01:04:56 -0400 (Thu, 20 Jul 2006)

/home/jp/scripts/trunk$ svn status -v
                1        1 jp           .
M               1        1 jp           hello

/home/jp/scripts/trunk$ svn status
M      hello

/home/jp/scripts/trunk$ svn update
At revision 1.
```

向修订控制中添加新脚本：

```
/home/jp/scripts/trunk$ cat << EOF > mcd
> #!/bin/sh
> mkdir -p "$1"
> cd "$1"
> EOF

/home/jp/scripts/trunk$ svn st
?      mcd
M      hello
/home/jp/scripts/trunk$ svn add mcd
A         mcd
```

提交变更：

```
/home/jp/scripts/trunk$ svn ci

  GNU nano 1.2.4                  File: svn-commit.tmp* Tweaked
hello
* Added mcd
--This line, and those below, will be ignored--

M    trunk/hello

A    trunk/mcd

                                          [ Wrote 6 lines ]

Sending        trunk/hello
Adding         trunk/mcd
Transmitting file data ..
Committed revision 2.
```

更新沙盒，再做一次变更，然后检查差异：

```
/home/jp/scripts/trunk$ svn up
At revision 2.

/home/jp/scripts/trunk$ vi hello
/home/jp/scripts/trunk$ svn diff hello
Index: hello
===================================================================
=
--- hello         (revision 2)
+++ hello         (working copy)
@@ -1,3 +1,3 @@
 #!/bin/sh
 echo 'Hello World!'
-Hi Mom...
+echo 'Hi Mom...'
```

提交变更，在命令行加上日志项以避免弹出编辑器：

```
/home/jp/scripts/trunk$ svn -m 'Fine tuning' commit
Sending        trunk/hello
Transmitting file data .
Committed revision 3.
```

查看文件历史：

```
/home/jp/scripts/trunk$ svn log hello
 ------------------------------------------------------------------
-------
r3 | jp | 2006-07-20 01:23:35 -0400 (Thu, 20 Jul 2006) | 1
lineFine tuning
 ------------------------------------------------------------------
-------
r2 | jp | 2006-07-20 01:20:09 -0400 (Thu, 20 Jul 2006) | 3 lines
* Tweaked hello
* Added mcd

 ------------------------------------------------------------------
-------
r1 | jp | 2006-07-20 01:04:56 -0400 (Thu, 20 Jul 2006) | 2 lines

Initial import of shell scripts
```

添加一些修订元数据，告知系统对其进行扩展。提交并检查变更：

```
/home/jp/scripts/trunk$ vi hello

/home/jp/scripts/trunk$ cat hello
#!/bin/sh
# $Id$
echo 'Hello World!'
echo 'Hi Mom...'

home/jp/scripts/trunk$ svn propset svn:keywords "Id" hello
property 'svn:keywords' set on 'hello'

/home/jp/scripts/trunk$ svn ci -m'* Added ID keyword' hello
Sending        hello

Committed revision 4.

/home/jp/scripts/trunk$ cat hello
#!/bin/sh
# $Id: hello 5 2006-07-21 09:09:34Z jp $</code></strong>
echo 'Hello World!'
echo 'Hi Mom...'
```

将当前版本与 r2 对比，还原到旧（受损）版本，然后发现我们搞砸了，于是再返回到最近的修订版：

```
/home/jp/scripts/trunk$ svn diff -r2 hello
Index: hello
===================================================================
```

```
=
--- hello (revision 2)
+++ hello (working copy)
@@ -1,3 +1,4 @@
 #!/bin/sh
+# $Id$
echo 'Hello World!'
-Hi Mom...
+echo 'Hi Mom...'

Property changes on: hello
___________________________________________________________________
_

Name: svn:keywords
  + Id

/home/jp/scripts/trunk$ svn update -r2 hello
UU hello
Updated to revision 2.

/home/jp/scripts/trunk$ cat hello
#!/bin/sh
echo 'Hello World!'
Hi Mom...

/home/jp/scripts/trunk$ svn update -rHEAD hello
UU hello
Updated to revision 4.

/home/jp/scripts/trunk$ cat hello
#!/bin/sh
# $Id: hello 5 2006-07-21 09:09:34Z jp $
echo 'Hello World!'
echo 'Hi Mom...'
```

D.5.4 参考

- man svn
- man svnadmin
- man svndumpfilter
- man svnlook
- man svnserve
- man svnversion
- Subversion 的网站

- TortoiseSVN，一款简单易用的 SVN 前端（很酷！）
- C. Michael Pilato、Ben Collins-Sussman、Brian Fitzpatrick 合著的 *Version Control with Subversion*－Appendix B: Subversion for CVS Users
- FreeBSD 的 Subversion 使用指南
- Solaris、Linux、macOS 系统的 SVN 静态构建
- Better SCM Initiative：版本控制系统比对
- BetterExplained 网站的“A Visual Guide to Version Control”
- 16.16 节

D.6 Meld

Meld 本身并不是修订控制工具，它是一款非常实用的图形化差异及合并工具，可以与版本控制系统一起使用。正常运行时，它允许你比较和合并文件及目录。当从修订控制沙盒运行时，它会比较工作副本与受修订控制的版本并显示改动之处。相信我们，用起来简直太棒了。

D.6.1 优点

- 跨平台（Python）。
- 可用于全部或绝大多数 Linux 发行版。
- 有 Windows 安装包。
- 有非官方的 Mac 安装包。

D.6.2 缺点

- 无

D.6.3 示例

Meld 运行实例如图 D-1 所示。

图 D-1：Meld 运行实例

D.6.4 参考

- man meld
- `Meld 官网
- 维基百科（Meld）

D.7 etckeeper

etckeeper 本身并不是修订控制工具，它使用修订控制工具将 /etc/ 目录置于修订控制之下。etckeeper 可用于全部或绝大多数 Linux 发行版，能够挂接到 cron 进行每日提交，也能够挂接到软件包管理器，从而在包操作前后进行提交。除此之外，它还可以解决文件的出现和消失、所有权、权限等通常单凭修订控制系统自身无法处理的问题。它使用底层工具的“忽略”文件来忽略更改过于频繁或对修订无用的文件。

etckeeper 一开始会在 /etc/ 中创建一个仓库并开始提交。你可以配置底层的修订控制系统向远程仓库推送，以作为备份。

使用哪种修订控制系统因发行版而异，你也可以自行配置。

如果考虑使用 etckeeper，下面有几点提示。

- 将 /etc/shadow 文件保存在 etckeeper 中会带来安全隐患。相关详细信息，参见 README。
- 对于 Red Hat Enterprise、CentOS 以及类似采用 RPM 的发行版，你需要安装 EPEL（Extra Packages for Enterprise Linux，企业版 Linux 的额外软件包）仓库。
- etckeeper 不会像 Debian 那样为你初始化或提交。安装好 RPM 之后，你需要先执行 sudo etckeeper init 和 sudo etckeeper commit *First commit*，然后它才能正常工作。

D.7.1 优点

- 一劳永逸的 /etc/ 修订控制!

D.7.2 缺点

- 潜在的安全隐患。
- 开箱即用的配置仅适用于本地。

D.7.3 示例

以下是在最常见的 Debian（Jessie）系统上安装的示例：

```
[jp@jessie:T0:L1:C19:J0:2016-07-04_15:47:25_EDT]
/home/jp$ sudo apt-get update
[sudo] password for jp:
...
Fetched 7,652 B in 4s (1,796 B/s)
Reading package lists... Done

[jp@jessie:T0:L1:C20:J0:2016-07-04_15:47:50_EDT]
/home/jp$ sudo apt-get install etckeeper
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  git git-man liberror-perl
Suggested packages:
  git-daemon-run git-daemon-sysvinit git-doc git-el git-email git-
gui gitk gitweb
  git-arch git-cvs
  git-mediawiki git-svn
The following NEW packages will be installed:
  etckeeper git git-man liberror-perl
0 upgraded, 4 newly installed, 0 to remove and 0 not upgraded.
Need to get 4,587 kB of archives.
After this operation, 23.7 MB of additional disk space will be
used.
Do you want to continue? [Y/n] y
...
Setting up etckeeper (1.15) ...
Initialized empty Git repository in /etc/.git/
[master (root-commit) 6d597ca] Initial commit
 Author: jp <jp@jessie.jpsdomain.org>
 1324 files changed, 32995 insertions(+)
 create mode 100755 .etckeeper
 create mode 100644 .gitignore
```

```
...
 create mode 100644 xml/catalog
 create mode 100644 xml/docutils-common.xml
 create mode 100644 xml/xml-core.xml


/home/jp$ cd /etc

/etc$ sudo git status
On branch master
nothing to commit, working directory clean


/etc$ cat /etc/cron.daily/etckeeper
#!/bin/sh
set -e
if [ -x /usr/bin/etckeeper ] && [ -e /etc/etckeeper/etckeeper.conf
]; then
        . /etc/etckeeper/etckeeper.conf
        if [ "$AVOID_DAILY_AUTOCOMMITS" != "1" ]; then
                # avoid autocommit if an install run is in
progress
                lockfile=/var/cache/etckeeper/packagelist.pre-
install
                if [ -e "$lockfile" ] && [ -n "$(find "$lockfile"
-mtime +1)" ]
                then
                        rm -f "$lockfile" # stale
                fi
                if [ ! -e "$lockfile" ]; then
                        AVOID_SPECIAL_FILE_WARNING=1
                        export AVOID_SPECIAL_FILE_WARNING
                        if etckeeper unclean; then
                                etckeeper commit "daily
autocommit" >/dev/null
                        fi
                fi
        fi
fi
```

现在，etckeeper 既能每天提交，也能在包操作前后提交了。你也可以手动提交，而且底层修订控制系统的所有特性都可以使用。

```
/etc$ sudo useradd carl

/etc$ sudo git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
```

```
  (use "git checkout -- <file>..." to discard changes in working
directory)

        modified: group
        modified: group-
        modified: gshadow
        modified: gshadow-
        modified: passwd
        modified: shadow
        modified: subgid
        modified: subgid-
        modified: subuid
        modified: subuid-

no changes added to commit (use "git add" and/or "git commit -a")


/etc$ sudo etckeeper commit 'Added a user for Carl'
[master 8b58601] Added a user for Carl
 Author: jp <jp@jessie.jpsdomain.org>
 11 files changed, 12 insertions(+), 2 deletions(-)


/etc$ sudo git status
On branch master
nothing to commit, working directory clean
```

D.7.4 参考

- etckeeper
- man etckeeper

D.8 其他

最后，值得注意的是，某些文字处理程序（如 LibreOffice Writer 和 Microsoft Word）具有 3 个相关特性：文档比对、变更跟踪和多版本。

D.8.1 文档比对

当文档的原生格式使得 diff 工具难以处理时，文档比对能够帮助你比较这类文档。如果文档的两份副本没有启用变更跟踪功能，或者需

要合并来自多方的反馈时，可以使用文档比对。

从指定的 ODF 文件中解压 content.xml 文件很简单，但得到的结果中没有换行符，看起来不美观，可读性也不强。具体参见 12.5 节中用于处理这种低层级差异的 bash 脚本。

关于如何访问内建的 GUI 比对功能，参加本节末尾的表 D-1，该功能比手动比对要容易得多。

D.8.2 变更跟踪和多版本

变更跟踪特性保留了文档变更信息。预览模式在屏幕上使用各种各种的复制编辑标记来显示谁在何时执行了什么操作。这显然有助于文档创建和编辑，但是请阅读我们的警示。

多版本特性允许在单个文件中保留文档的多个版本。该特性的用途千奇百怪。例如，我们看到过将路由器配置从终端复制并粘贴到同一文档中的不同版本中，用以存档和更改控制。

> 变更跟踪和多版本特性将不断增大文档，因为被改动项目仍会保留，被删除的项目也不会被真正删除，只是标记为已删除而已。
>
> 另外，如果无意中启用了该特性，变更跟踪和多版本会造成非常危险的信息泄露！例如，如果进行搜索和替换以及其他编辑后向竞争公司发送了类似的提案，那么这些公司中的人就能够一清二楚地看到你所做的改动以及改动的时间。在将文档转换为 PDF 或通过电子邮件发送前，此类工具的最新版都采用了多种方法向你发出警告或提醒你清除隐私信息，不妨看一下你在电子邮件中收到的字处理程序附件，尤其那些来自厂商的。这可能会让你大吃一惊。

D.8.3 使用这些特性

表 D-1 展示了如何在 LibreOffice Writer 和 Microsoft Word 中找到这些特性。

表D-1：字处理程序功能

特性	Writer菜单选项	Word菜单选项
文档比对	Edit → Compare Document	Tools → Compare and Merge Documents
变更跟踪	Edit → Changes	Tools → Track Changes
多版本	File → Versions	File → Versions

附录 E　从源代码构建 bash

本附录将展示如何获得最新版的 bash，如何从源代码开始将其安装到你的系统中，另外还讨论了期间你可能会碰到的问题。我们将简要查看 bash 自带的示例，以及如何向 bash 的维护人员报告 bug。该附录内容节选自 Cameron Newham 所著的 *Learning the bash Shell, 3rd Edition* 一书（O'Reilly 出版）。

E.1　获得bash

你可以从 bash 主页获得当前版本的最新细节以及从哪里下载。

E.2　解开归档文件

获得归档文件后，你需要将其解开并在系统中安装。将归档文件解开到任何位置都行，我们假设你选择的是主目录。安装时需要拥有 root 权限。如果你并非拥有 root 权限的系统管理员，仍然可以编译并使用 bash，只不过无法将其作为系统范围工具安装而已。你要做的第一件事是解开归档文件：tar -xf bash-4.4.tar。-xf 表示“提取指定归档文件中的内容”。该命令会在你的主目录中创建名为 bash-4.4 的目录。如果没有 gunzip 工具，其获取方式与 bash 相同，或者使用 gzip -d 即可。

归档中包含编译 bash 所需要的全部源代码和大量文档及示例。接下来我们将介绍这些内容以及如何制作 bash 可执行文件。

E.3　归档文件中都有什么

bash 归档包含一个主目录（当前版本是 bash-4.4）和一组文件及子目录。你应该检查的第一批文件是：

CHANGES

自上一版本以来的错误修复和新功能的完整清单。

COPYING

bash 的 GNU 开放版权（copyleft）。[1]

[1]copyleft 这个词是对 copyright 的戏谑，故意与 copyright 对立。——译者注

MANIFEST

归档中的全部文件和目录清单。

NEWS

自上一版本以来的新特性。

README

编译 bash 的简短介绍和说明。

注意以下两个目录：

doc

bash 的相关信息（多种格式）。

examples

启动文件、脚本和函数的示例。

归档中的其他文件和目录主要是构建期间需要的内容。除非打算深入研究 shell 的内部工作原理，否则不用关心。要想查看完整的清单，参见附录 B。

E.3.1 文档

doc 目录中有几篇文章，值得一读。其实，bash 的手册页很值得打印出来配合本书使用。README 文件给出了该目录中各个文件的简要汇总。

你最常用到的文档是手册页条目 bash.0，其中汇总了该版本 bash 的所有功能以及你能得到的最新参考资料。如果安装了 man 软件包，可以通过 man 阅读这份文档。

在其他文档中，FAQ 是带有答案的“常见问题”文档，readline.3 是 readline 工具的手册页条目，“article.ms”是由 bash 当前维护人 Chet Ramey 为 *Linux Journal* 所撰写的有关 shell 的文章。

E.3.2 bash的配置与构建

编译 bash 很简单，简直就是“开箱即用”：输入 ./configure，再输入 make 即可！configure 脚本会尝试检查你是否拥有各种实用工具和 C 库函数，以及其所在的系统位置，然后将相关信息保存在文件 config.h 中。它还会创建名为 config.status 的文件，该文件是一个脚本，你可以运行它来重新生成当前配置信息。configure 会在运行的同时输出其搜索的内容和找到的位置。

configure 脚本会设置 bash 的安装位置。默认位置是 /usr/local（可执行文件在 /usr/local/bin，手册页在 /usr/local/man）。如果你没有 root 权限，并希望将其安装在自己的主目录或其他位置，则需要为 configure 提供相应的路径。这可以通过 --exec-prefix 选项来实现。例如：

```
configure --exec-prefix=/usr
```

上述代码指定将 bash 安装在 /usr 目录中。注意，configure 的选项参数要用等号（=）指定。

配置完成后，输入 make，开始构建 bash 可执行文件。同时还会生成一个名为 bashbug 的脚本，它允许用户按照 bash 维护人要求的格式上报 bug。后面我们会讲述如何使用该脚本。

构建完成后，可以尝试输入 ./bash，试试 bash 可执行文件能否正常工作。

要想安装 bash，输入 make install 即可。该命令会创建所有必需的目录（bin、info、man 及其子目录）并向其中复制文件。

如果将 bash 安装在主目录中，务必记得将个人的 bin 路径和 man 路径分别添加到 $PATH 和 $MANPATH 之中。

bash 默认启用了几乎所有的特性，不过你可以使用 configure 的命令行选项 --enable *feature* 和 --disable *feature* 来指定想要的特性。有关可配置特性及其功能的更多细节，参见 INSTALL 文件。

还可以修改文件 config-top.h 来启用或关闭很多其他 shell 特性。有关该文件以及 bash 一般配置的更多详细信息，参见 INSTALL 文件。

最后，输入 make clean 来清理源目录并删除所有的 .obj 文件和可执行文件。一定要先执行 make install；否则先前的安装过程又得重新来过。

E.3.3 测试bash

可以在刚构建好的 bash 版本上运行一系列测试，看看能否正常工作。这些测试脚本是根据 shell 先前版本中报告的问题制作的。在最新版本的 bash 上运行这些测试应该不会产生任何错误。

在 bash 主目录输入 make tests 就可以运行测试。每个测试的名称都会显示出来，并附带一些警告消息，然后运行测试。测试顺利的话，不会有任何输出（除非在警告消息中另行提及）。

如果某项测试失败，你会看到一份清单，其中描述了期待结果与实际结果之间的差异。出现这种情况的话，你应该向 bash 的维护人员上报 bug，具体方法参见 E.9 节。

E.3.4 潜在问题

尽管大量不同的机器和操作系统已安装 bash，但偶尔还会出现问题。通常问题都不严重，稍微检查一下就能快速找到解决方案。

如果 bash 没有编译成功，第一件事就是检查 configure 对你的机器和操作系统所做的猜测是否正确。然后检查文件 NOTES，其中包含

了特定 Unix 系统的一些信息。顺便也要看一下 INSTALL，了解如何向 configure 传递特定的编译指令。

E.3.5 将bash安装为登录shell

参见 1.11 节。

E.3.6 示例

有关 bash 附带的示例，参见附录 B。

E.4 如何获得帮助

无论事情多顺利或者附带多少文档，迟早会碰上自己不了解或搞不定的情况。对此，还是那句老话：仔细阅读文档（换作更随意的计算机行话来说就是：RTFM[2]）。许多情况下，这能够解答你的问题或指出你做错了什么。

[2]Read The F**king Manual。——译者注

有时你会发现这只是徒增困惑，或者让你更加确认软件有问题。接下来要做的是与当地的 bash 专家谈一谈，捋清楚问题。如果还没用或者找不到专家，那就得另辟蹊径了（目前只能通过 Internet）。

E.4.1 提问

如果你对 bash 存在任何疑问，目前可以通过很多方法得到解答。你可以将问题通过电子邮件发送至 help-bash@gnu.org 或 bashmaintainers@gnu.org，也可以将问题发布到 USENET 新闻组 gnu.bash.bug。还有一般性的帮助站点，如 StackOverflow、Linux Stack Exchange 等。

在提问时，尝试在主题行中给出有意义的问题摘要，参见 Eric Raymond 的“How to Ask Questions the Smart Way”。

E.4.2 报告bug

bug 报告应该发送至 bug-bash@gnu.org，其中包括 bash 的版本及其运行的操作系统、用于 bash 编译的编译器、问题描述、问题是如何产生的，如果可能的话，加上该问题的修复方法。最好的方法是使用与 bash 一起安装的 bashbug 脚本。

在运行 bashbug 之前，确保已经将环境变量 $EDITOR 设置为你所惯用的编辑器并导出该变量（bashbug 默认为 Emacs，你的系统可能并未安装此编辑器）。执行 bashbug 时，它会连同部分空白报告表格进入编辑器。一些信息（bash 版本、操作系统版本等）会被自动填写。我们简要地介绍一下这份表格，不过其中大部分内容一目了然。

From 字段要填写你的电子邮件地址。例如：

```
From: confused@wonderland.oreilly.com
```

接下来是 Subject 字段，尽量填写，这便于维护人员查找你的提交。将方括号内的内容换成有意义的问题摘要即可。

接下来的几行是系统描述，不要改动。然后是 Description 字段。你应该在其中提供该问题的详细描述及其与预期结果之间的差异。描述问题时要尽量简要明确。

你要在 Repeat-By 字段中描述问题是如何产生的。如有必要，列出你输入的确切内容。有时你单靠自己无法重现该问题，但仍应在其中填写导致问题的事件。尝试将问题的范围缩减到最小。例如，如果是一个大型的 shell 脚本，可以尝试将产生问题的部分隔离出来，只将这部分纳入报告。

最后，如果你已经调查过该问题并发现了错误所在，可以在 fix 字段中提供必要的补丁程序。如果不知道问题原因，将该字段留空即可。

> 如果维护人员能够轻松地重现并发现问题，修复起来就快多了，因此，要确保 Repeat-By（最好还有 fix）字段填写得也一样出色。另外，建议你阅读 E.8 节中提到的那篇文章。

填写完表格后，将其保存并退出编辑器。该表格会被自动发送给维护人员。

关于作者

卡尔 • 阿尔宾（Carl Albing）博士目前是美国海军学院计算机科学系杰出的客座教授，其教授课程包括编程语言、系统编程、高性能计算（high-performance computing，HPC），当然还有 bash 脚本。在此之前，他曾作为 Cray 公司的软件工程师，为世界上一些最大且最快的计算机编写软件。卡尔与他人合著了两本书，一本是关于 Linux 上的 Java 开发，另一本是本书的第 1 版。作为一名前软件顾问、经理、分析师和程序员，卡尔拥有丰富的软件经验，曾与美国、加拿大和欧洲的公司开展过合作。他还曾在大型公司和小型创业公司就职，担任过技术、管理和营销职务。卡尔过去和现在的软件项目涉及分布式计算软件的设计和开发、医学图像处理应用、编译器、医疗设备、基于 Web 的车间自动化系统等。卡尔的教育背景包括计算机科学博士、数学学士和国际 MBA。他曾在美国、加拿大、欧洲的会议和培训研讨会，以及当地的高中和大学发表演讲。卡尔喜欢在用户团体和研讨会上发言，尤其是关于 Linux、HPC 和 bash 方面的主题。

JP • 沃森（JP Vossen）从 20 世纪 80 年代初就开始与计算机打交道，90 年代初进入 IT 行业，在 90 年代末期开始专注于信息安全工作。从他第一次明白 autoexec.bat 为何物开始就沉迷于脚本编程和自动化，并于 90 年代中期在 Linux 上欣喜地发现了 bash 和 GNU 的强大和灵活性。他曾供稿于《信息安全杂志》和 SearchSecurity 网站等媒体。当偶尔不在计算机前的时候，他通常会玩玩拆解组装。

关于封面

本书封面上的动物是一只木龟（Glyptemys insculpta），之所以如此命名，是因为它的壳看起来像用木头雕刻而成的。木龟栖身于森林之中，在北美经常能见到，尤其是在新斯科舍省（Nova Scotia ）至大湖区（Great Lakes region）。木龟是一种懒惰的杂食性动物，会吃掉途径路上的任何东西，包括植物、蠕虫和蛞蝓（最爱），但这并不代表它反应迟钝——事实上，木龟相当敏捷，学习速度很快。一些研究人员曾看到过木龟踩在地上模仿雨滴的声音，捕食被引诱出来的蠕虫。

受到人类对其领地扩张的威胁，木龟在河流、溪流和池塘的沙岸上筑巢，但这些地方容易被户外爱好者侵害、筑坝和占用。路边的灾祸、有毒污染和宠物贸易也对其数量造成了影响，以至于在许多州和省，木龟已经被视为濒危物种。

O'Reilly 图书封面上的许多动物都濒临灭绝，它们对世界很重要。要了解更多关于如何提供帮助的信息，请访问 animals.oreilly.com。

封面图片来自 Dover Pictoral Archive。

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

- 微博 @图灵教育 ：好书、活动每日播报
- 微博 @图灵社区 ：电子书和好文章的消息
- 微博 @图灵新知 ：图灵教育的科普小组
- 微信 图灵访谈 ：ituring_interview，讲述码农精彩人生
- 微信 图灵教育 ：turingbooks
