序

真的！我已经 30 年未写 Shell 脚本了？！？现在仔细想想，我想应该有吧，虽然一开始只是作些简单的工作（早期的 UNIX Shell，在 Bourne Shell 之前，是极为原始的，因此要写个实用的脚本是很难的事，幸好那段日子并不长）。

近几年来，Shell 一直被忽略，是一个不受重视的脚本语言。Shell 虽然是 UNIX 的第一个脚本语言，但它仍是相当优秀的。它结合了延展性与效率，持续保有独具的特色，并不断地被改良，使它们多年来一直能与那些花招很多的脚本语言保持抗衡。GUI 是比命令行 Shell 更流行的用户界面，但脚本语言时常都是这些花哨的屏幕图形界面最强有力的支柱，并一直称职地扮演这个角色。

Shell 需依赖其他程序才能完成大部分的工作，这或许是它的缺陷，但它不容置疑的长处是：简洁的脚本语言标记方式，而且比 C（还有其他语言）所编写的程序执行更快、更有效率。它使用通用的、一般用途的数据表示方式，文本行，在一个大的（且可扩展的）工具集中，让脚本语言能够搭配工具程序，产生无穷的组合。用户可以得到比那些独占性软件包更灵活、功能更强大的工具。Shell 的早期成功即以此法强化 UNIX 的开发哲学，构建一套专门性、单一目的工具，并将它们整合在一起做更多的事。该原则接着鼓励了 Shell 的改良，允许用这种方式完成更多的工作。

Shell 脚本还有一个超越 C 程序的优势，同样也优于其他脚本语言的地方，可用一般方式轻松地读取与修改。即便不是C的程序设计人员，也能像现今许多系统管理人员一样，很快就能接受 Shell 脚本。如此种种，让 Shell 脚本成为延展用户环境与定制化软件包的重要一环。

的确，它其实有一种“周而复始”的特性，在我看过这么多软件项目之后。项目将简单的 Shell 脚本置于关键位置，让用户容易地从他们的角度来定制软件。然而，也因为这

些项目的 Shell 脚本与周围的 C 程序码相比较，要更容易解决问题，所以不断产生更复杂的脚本。最后，它们终于复杂到让用户很难轻易地处理（我们在C News项目里的部分脚本就拥有著名的 Shell 压力测试，完全未考虑用户的立场），且必须提供新的脚本集合，供用户进行定制……

长久以来，一直都没有编写 Shell 脚本相关的好书出现。UNIX 程序设计环境方面的书籍偶有触及这方面议题，但通常只是简短带过，作为它众多主题的一部分，有些写得不错的书也很久没有更新了。好的参考文件应该是针对各种不同的 Shell 讨论，但必须是贴近新手的实战手册，涵盖工具程序与 Shell，以循序渐近的方式介绍，告诉我们如何得到更好的结果与输出，还要注意到实例面，像是可读性议题。最好它还讨论各式 Shell 的异同，而不是好像世上只有一个 Shell 存在一样。

这本书就是这样的，甚至做到比上面说的还多。至少，它是第一本且最好的一本、内容最新的、以最轻松的方式介绍 UNIX 脚本语言的书。以实用的范例进行解说，让工具充分发挥自己的效能。它包括了标准 UNIX 工具，让用户有个好的开始（对于觉得看手册页有点难的用户来说，这会是个相当不错的参考教材）。我最高兴的是看到将 `awk` 列入取材范围，这是相当有用且不容忽视的工具程序，适于整合其他工具及简洁地完成小型程序设计的工作。

我建议所有正在编写 Shell 脚本或管理 UNIX 系统的人都要读这本书。我在这本书上学到很多，我想你也会。

—— Henry Spencer
SP Systems

前言

刚开始使用UNIX（注 1）的用户与程序员突然面对各式各样的程序时，都会有很多疑问，例如“它们的功能是什么”，还有“我怎么使用它们”。

本书可以回答你这些问题。告诉你如何结合 UNIX 工具，将其与标准的Shell 相结合完成工作。Shell脚本的编写是门艺术，需要的不只是Shell语言的相关知识，还要你对各个独立的UNIX程序有基本认识：为什么会有这些工具，要怎么单纯地使用它们，怎么将它们与其他程序结合应用。

为什么需要学习如何编写 Shell 命令？因为大部分情况下，中型到大型的问题都能拆成较小的部分，这些小部分也多半都能找到现成的UNIX工具处理。用心编写的好用Shell脚本常常能够比C或C++语言编写的程序更快地解决相同的问题。也可以让Shell脚本提供可移植性，也就是说，可以跨越UNIX与POSIX兼容的系统，有时仅需略作修改，甚至不必修改，即可使用。

谈到UNIX程序时，我们使用工具（tool）这个字。以UNIX工具箱（toolbox）的做法解决问题，长久以来以“软件工具（Software Tools）”哲学（注 2）为人所熟知。

瑞士军刀是很多人口袋里的好帮手。它有刀刃、螺丝起子、开罐器、牙签等工具。功能更齐备的，还有其他像拔塞钻、放大镜等工具。瑞士军刀能派上用场的时候很多，虽然用它来修削和进行简单雕刻很不错，但你绝不会拿它来盖狗屋或制作鸟类喂食器。相反，

注 1：整本书里，我们都使用UNIX这个字，指的不单单是商用的UNIX系统原始版本，像Solaris、Mac OS X，与HP-UX，还包括可自由取得的类似系统，例如GNU/Linux与各种BSD系统：BSD/OS、NetBSD、FreeBSD，与OpenBSD。

注 2：此法因《Software Tools》（Addison-Wesley）这本书而广受欢迎。

做这类工作时你会寻求更专门的工具，例如铁槌、锯子、夹钳或刨刀等。同理，当你在解决程序化问题时，使用专门的软件工具会比较好。

这是给谁看的书

这本书是写给那些在UNIX环境下发现必须写些Shell脚本，以利于工作进行的计算机用户与软件开发人员。例如，你可能是正在念计算科学的学生，手上有学校给你的第一个UNIX系统账号，你想知道在UNIX下更多的东西，例如你的Windows个人计算机无法处理的那些工作（这种情况下，你通常得写几个脚本来定制个人环境）。或者，你可能是个系统管理新手，需要为公司或学校写几个专用程序（可能是处理事件日志文件，账号、账单管理之类的事情）。你也可能是Mac OS的开发老手，但转到崭新的Mac OS X的世界，它的安装程序是以Shell脚本写成。不管你来自哪里，如果你想学Shell脚本，这本书就是写给你的。在这本书里你能学到：

软件工具设计概念与原则

一些好的软件工具设计与实例上的实践规则。我们会解释这些原则，还会在这本书里贯彻执行。

UNIX工具是什么

UNIX的核心工具组会在我们编写Shell脚本时不断地重复使用。我们会介绍Shell与正则表达式的基本概念，并在解决特定问题时展现各种核心工具的用法。除了介绍工具能做什么之外，我们还会告诉你，为什么要使这个工具，为什么它有这些特殊选项。

《Learning UNIX》这本书是在介绍UNIX系统，让你从对UNIX毫无经验成长为会基本操作的用户。《UNIX in a Nutshell》这本书则是广泛地介绍UNIX工具包，对于使用时机与特定工具用法的介绍很少。我们的目的就在弥补这两本书之间的鸿沟：如何灵活运用这些UNIX提供的工具包，让工作更顺畅，更有效率，也更从容（我们的期望）。

如何结合所有工具，完成工作

编写Shell脚本时，其实会是“整体的功能比各部分加起来的总和还强大”。Shell的使用就像整合个别工具的黏着剂，让你只要花点心思，就能得到惊人的效果。

标准工具几个常见的扩展

如果你已经是GNU/Linux或BSD系统的用户，很可能你的工具还有其他额外的、好用的功能或选项。这部分我们也会介绍。

不可或缺的非标准工具

有些程序，在大部分传统的UNIX系统里并非“标准的”，但我们又不能没有它。我们会在适当的地方介绍它们，也会提供使用时机的相关信息。

对长期使用UNIX的开发人员与管理者来说，软件工具的设计原则一直没有什么改变。因此，推广的书籍虽然还算堪用，但已经20年未更新了，甚至更久！UNIX系统在这些书写成之后，有了许多变动。因此，我们觉得是更新这些想法的时候了，我们利用这些工具的现行版本、在现行系统下展示范例。下面是我们将要强调的部分：

- 所有的呈现是以POSIX为基础。POSIX为一系列描述可移植操作系统环境的标准正式名称的缩写。POSIX标准是开发人员的挑战，他们必须兼顾程序与Shell脚本在不同厂商所提供的各种平台上的可移植性。我们将在最新的POSIX标准下展现Shell语言、各个工具程序及其选项。

  该标准的官方名称为IEEE Std. 1003.1-2001（注3），它包括数个可选用的部分，最重要的一个就是X/Open System Interface（XSI）规格。这些功能文件详细描述了UNIX系统长久以来的行为模式。我们会介绍现行标准与早期1992标准间的差异，也会提供与XSI相关的特点。要了解UNIX相关标准，*http://www.UNIX.org/*（注4）是一个很好的起点。

  Single UNIX Specification的官方网站为*http://www.UNIX.org/version3/*。该标准可在线访问，不过得先到*http://www.UNIX.org/version3/online.html*注册。

  有时，该标准会将特殊行为模式保留为未定义（unspecified）。这是为了能让厂商以扩展的方式支持该行为，例如：额外的功能与标准本身未做成文件的部分。

- 除了告诉你如何执行特定程序外，我们还会强调这些程序存在的原因，及它们能解决什么问题。了解为什么会有这样的程序，有助于你进一步了解它的使用时机与方式。

- 大部分的程序都提供了相当多的选项组合。但通常只有一小部分是日常工作用得上的。这类程序，我们会让你知道它的哪些选项较方便好用。事实上，我们无法遍及每个程序的所有选项，未提及的部分，你可以通过程序的使用手册或其他参考书籍，例如《UNIX in a Nutshell》（O'Reilly）与《Linux in a NutShell》（O'Reilly），来了解它。

在你看完这本书之后，你不仅能了解UNIX工具集，还能吸收到UNIX的中心思想与软件工具设计的原则。

注3：该标准的2004版在本书内文底定后才发表。对学习Shell脚本而言，2001与2004间的差异并不重要。

注4：有关IEEE Std. 1003.1-2001的常见技术性问答（FAQ）文件，你或许可以在*http://www.opengroup.org/austin/papers/posix_faq.html*找到。与标准有关的后台知识，则在*http://www.opengroup.org/austin/papers/backgrounder.html*中。

你应该先有什么基础

我们认为读者应该了解以下背景知识：

- 如何登录 UNIX 系统
- 如何在命令行上执行程序
- 如何做一个简单的命令管道，与使用简单的输出 / 入重定向，例如 < 与 >
- 如何以 & 将程序放到后台执行
- 如何建立与编辑文件
- 如何使用 chmod，将脚本设为可执行权限

再者，如果你想试着操作本书范例，在你的终端机（或终端机模拟器）下达命令时，我们建议你使用 POSIX 兼容的 Shell，例如 ksh93 的最新版本，或 bash 的近期版本。请特别注意：商用 UNIX 系统上的 /bin/sh 可能并非完全兼容于 POSIX。

ksh93、bash 与 zsh 的下载网址请见第 14 章。

各章介绍

我们建议你依序阅读本书，因为每个章节都与前面章节息息相关。我们在此逐章介绍如下。

第 1 章，背景知识

此处提供简短的 UNIX 历史沿革。特别是贝尔实验室的运算环境，也就是 UNIX 开发的地方，激发了许多软件工具设计上的哲学。该章还会介绍这些原则，并在本书中贯彻执行。

第 2 章，入门

该章从编译语言与脚本语言间的取舍开始讨论。之后再介绍两个相当简单但很实用的 Shell 脚本程序。涵盖范围包括了命令、选项、参数、Shell 变量、echo 与 printf 的输出、基本输入 / 输出重定向、命令查找、从脚本里访问参数以及执行跟踪。最后则以国际化与本地化结束；这是在今日“地球村”环境下渐受重视的议题。

第 3 章，查找与替换

这里会介绍以正则表达式进行文字查找（或比对）。我们还会说明修改与提取文字的操作。这些都是最基本的 Shell 脚本编写的操作。

第 4 章，文本处理工具

该章介绍的是一些文字处理的软件工具，这些在 Shell 脚本编写时，都会一再地使用。其中最重要的两个就是 sort 与 uniq，在重组与降低数据量上，它们扮演很重要的角色。本章还会带你看看如何重新编排段落、计算文字单位、显示文件以及取出文件的前几行、后几行数据。

第 5 章，管道的神奇魔力

该章以几个小型脚本为例，展示结合简单的 UNIX 工具程序能够产生更强大、更灵活的工具。本章的内容采取 cookbook（问题描述与解决方案）的形式，它们共同的部分在于所有的解决方案都组合自线性的管道（pipelines）。

第 6 章，变量、判断、重复动作

这章介绍 Shell 语言里不可或缺的部分。包含了 Shell 变量与算法、退出状态的重要概念、如何判断，以及 Shell 循环的处理。最后以 Shell 的函数作结束。

第 7 章，输入 / 输出、文件与命令执行

该章为 Shell 描述的另一章，也是结尾，重点放在输入 / 输出、Shell 所执行的各种替换、加引号、命令行执行顺序，以及 Shell 内置命令上。

第 8 章，产生脚本

我们在这里会示范如何结合 UNIX 的工具以处理更复杂的文本处理工作。本章的程序比第 5 章的还大，但仍是几分钟便能消化掉。甚至它们所完成的工作，如果使用传统的程序语言，例如 C、C++ 或 Java™ 来做，会很困难。

第 9 章，awk 的惊人表现

该章介绍的是 awk 语言必备的组成部分。awk 是一套功能强大且自给自足的语言。而 awk 程序更可用来与其他软件工具箱里的其他程序相结合，以执行简单的数据提取、处理与格式编排工作。

第 10 章，文件处理

该章介绍了处理文件的几个主要工具。包括列出文件、产生临时文件，以及利用指定标准寻找文件的 find 命令。另外还有两个与磁盘空间有关的重要命令，以及比较文件间异同的几个程序。

第 11 章，扩展实例：合并用户数据库

将所有东西串起来，解决既有趣又难易适中的挑战性工作。

第 12 章，拼写检查

该章利用拼写检查的问题，展现如何以数种方式解决它。这里展现了原始的 UNIX Shell 脚本管道以及两个小型的脚本：ispell 与 aspell 命令，可自由下载，它们更适用于批处理的拼写检查工作。我们以 awk 写了一个大小适当的拼写检查程序，充分展现使用该语言的简单利落。

第 13 章，进程

该章将重点从文本处理的领域转到工作（job）与系统管理上。我们介绍了几个用于管理进程的必备工具，还有 `sleep` 命令，这在脚本需要等待某些事发生时很有用，另外则是其他一些用于延迟的标准工具，或修正日期时间命令的处理。最重要的是，该章也包括了 `trap` 命令，它可以让 Shell 脚本控制 UNIX 的信号。

第 14 章，Shell 可移植性议题与扩展

这里介绍的是一些更有用的扩展，可使用于 `ksh` 与 `bash` 之下，而非 POSIX。一般情况下，你都能安心地将这些扩展套用在你的脚本里。该章还会带你看几个“gotchas”，这是等待粗心大意的Shell脚本编写者跳入的陷阱。内容包括了在编写脚本时该注意的事项，还有在执行时可能出现的矛盾。除此之外，还包括有`ksh`与`bash`的下载与安装。该章最后会探讨各种不同的 Shell 实现间，Shell 初始化与终结的差异。

第 15 章，安全的 Shell 脚本：起点

该章会粗略介绍编写 Shell 脚本时的安全性议题。

附录 A，编写手册页

该附录讲的是如何编写手册页。这个必备的技术，在传统的UNIX的书籍里常被忽略。

附录 B，文件与文件系统

这个附录会介绍 UNIX 的字节数据流文件系统模型，并与较复杂的文件系统对照，然后解释其简洁的好处。

附录 C，重要的 UNIX 命令

该附录提供了许多UNIX命令的列表。建议你了解这些命令，它们可以增强你的能力。

参考书目

这是 UNIX 的 Shell 脚本编写的其他资料来源。

本书惯例

我们假设你已经知道，输入 Shell 命令时，最后会按下 Enter。Enter 在某些键盘上被表示为 Return。

提到 Ctrl-X 时，X 指的是任意字母，是在你按住 Ctrl（或 Ctl，或 Control）之后，接着按下的键。虽然我们这里用的是大写，不过你在按这个字母的时候无须按住 Shift 键。

其他特殊字符有换行符号（同于 Ctrl-J）、Backspace（同于 Ctrl-H）、Esc、Tab，与 Del（有时被标示为 Delete 或 Rubout）。

本书使用下列字体惯例：

斜体（*Italic*）

用在电子邮件地址、Internet URL、使用手册的引用。还用以表示参数，表示你在使用时，可以将它置换为你要的实际值，以及为范例提供说明性文字。

等宽字体（Constant Width）

提及UNIX文件名、扩展与内置命令、命令选项时，我们就会用到这个字体。除此之外，变量名称、Shell关键字、选项、函数、文件名结尾、范例中显示文件内容或命令的输出，以及当命令行或输入范例存在于正规文字中时，我们也会以等宽字体表示。简而言之，任何与计算机使用相关的，我们都用这个字体。

粗体等宽字体（**Constant Width Bold**）

这种字体用以区分比对文字中的正则表达式与Shell通配字符样式。在范例中，显示用户与Shell间的互动，我们也会使用这个字体，所有用户应键入的，我们都以粗体等宽字体显示，像这样：

```
$ pwd                              用户输入这个
/home/tolstoy/novels/w+p           系统显示这个
$
```

斜体等宽字体（*Constant width italic*）

这个字体是用在范例与内文中，需置换为正确值的命令行参数上。例如：

```
$ cd directory
```

注意：表示诀窍、建议或一般注意事项。

警告：表示警告与提醒。

参照UNIX User's Manual时，我们会使用标准形式：*name*(N)，*name*为命令名称，而N为section编号（通常是1），也就是寻找信息的地方。例如*grep*(1)即`grep`的Section 1的手册页。参考文件我们使用man page，或直接简写为manpage。

UNIX系统调用与C程序库，我们都这么写：`open()`、`printf()`。你可以使用`man`命令，查看这两者的manpage：

```
$ man open          查看open(2)的manpage
$ man printf        查看printf(3)的manpage
```

当我们要介绍一个程序时，就使用下面的方式，显示在正文附近，说明该工具程序与它的重要选项、语法与用途。

范例

语法

whizprog [*options* ...] [*arguments* ...]

说明如何执行这里指出的whizprog命令。

用途

说明此程序存在的目的。

主要选项

列出此程序每天要用到的重要选项。

行为模式

概括程序所做的事。

警告

所有要注意的事全在这。

程序码范例

本书并非只是解释命令与程序的功能，还提供了完整的Shell命令与程序的设计范例，便于用户或程序员直接应用于日常工作上。我们非常鼓励你修改与强化这些范例。

本书所提供的程序代码，都公开在GNU General Public License（GPL）条款下，允许程序复制、再利用与编修。请参阅范例包括的COPYING文件，了解许可权的实际条款。

代码可从原书网站获得：*http://www.oreilly.com/catalog/shellsrptg/index.html*。

我们会感谢你在使用程序码范例时注明出处，但这并非必要。这通常包括标题、作者、出版商与ISBN。例如：《Classic Shell Scripting》，作者Arnold Robbins与Nelson H.F. Beebe。版权所有2005 O'Reilly Media, Inc.，ISBN为0-596-00595-4。

Windows系统下的UNIX工具程序

很多的程序员初次接触UNIX系统后，再回到个人计算机的世界，常会希望也有一个像UNIX那样好用的环境（特别是在面对难以处理的MS-DOS命令行时），所以会有UNIX Shell式的界面出现在一些小型计算机的操作系统上，也不是什么奇怪的事。

近几年，我们不只看过Shell仿制品，还看过整个完整的UNIX环境的仿制品。其中两个就是使用bash与ksh93，其他则是提供自有的Shell重新实现（reimplementation）。本节将依次（以字母顺序）予以介绍，并附上联络方式与Internet下载信息。

Cygwin

Cygnus Consulting（现为Red Hat）建立了cygwin环境。首先出现的是提供UNIX系统调用模拟器的共享程序库cgywin.dll，该公司释出了许多GNU工具程序，供各种不同的Microsoft Windows版本使用。模拟器还包括了Berkeley socket API的TCP/IP网络。在Windows/NT、Windows 2000与Windows XP下，功能性最佳，不过在Windows 95/98/ME下也是可以运行的。

cygwin环境使用自己的Shell、自己的C编译器GCC，以及搭配其UNIX工具集里的GNU工具程序。设计精良的mount命令提供了将Windows C:\path的标记方式对应到UNIX文件名。

*http://www.cygwin.com/*是你了解cygwin项目的起点。第一步就是下载它的安装程序，执行时，你可以选择要安装哪些额外包。整个安装过程都是在Internet上进行，没有官方的cygwin CD，至少项目的维护人员并没有提供。

DJGPP

DJGPP程序组提供了MS-DOS环境下所使用的32位GNU工具程序。以下内容摘自其网页：

> DJGPP为执行MS-DOS的Intel 80386（及更高级的）个人计算机提供了完整的32位C/C++开发系统。其中包含许多GNU开发工具程序。这些开发的工具必须在80386或更高级的计算机上执行产生程序。大部分情况下，其所产生的程序可做商业用途而无须授权或版税。

其名称一开始是来自D.J. Delorie，D.J. Delorie将GNU C++编译器g++移植到MS-DOS，而g++一开始的名称为GPP。之后逐渐茁壮成长，成为MS-DOS下完整的UNIX环境不可或缺的要素，并带有GNU工具，以bash作为其Shell。不同于cygwin或UWIN的是：不需要使用任何的Windows版本，只要有完整的32位处理器与MS-DOS即可（当然，你也可以在Windows的MS-DOS窗口下使用DJGPP）。官方网站为*http://www.delorie.com/djgpp/*。

MKS Toolkit

个人计算机世界里的UNIX环境有许多都是以Mortice Kern Systems的MKS Toolkit建立的：

MKS Canada – Corporate Headquarters
410 Albert Street
Waterloo, ON
Canada N2L 3V3
1-519-884-2251
1-519-884-8861 (FAX)
1-800-265-2797 (Sales)
http://www.mks.com/

MKS Toolkit版本非常多，根据你的开发环境和开发人员的数量来决定要用哪一个版本。其中包含了与POSIX兼容的Shell，拥有1988 Korn Shell里的所有功能，超过300种的工具组，例如`awk`、`perl`、`vi`、`make`等。MKS程序库支持超过1500个UNIX API，使它更为完整，更易于应用在Windows环境下。

AT&T UWIN

UWIN包是David Korn与同事为了在Microsoft Windows下使用UNIX环境而产生的项目。其架构类似先前讨论的`cygwin`。共享程序库`posix.dll`提供了UNIX系统调用API的模拟器。其系统调用模拟器相当完整。其中一个有趣的创新就是将Windows的登录改为可在`/reg`文件系统下访问的方式。UWIN环境依赖原始的Microsoft Visual C/C++编译器，不过仍可自行下载GNU开发工具用于UWIN下。

*http://www.research.att.com/sw/tools/uwin/*是此项目的网页，说明可供下载的有哪些，并附上二进制文件的下载链接，还有与UWIN的使用许可权相关的信息。除此之外，另有UWIN的各类报告、额外的好用软件及其他类似包的链接。

UWIN包最大的优势为：它的Shell是一个真正的`ksh93`，因此在与UNIX的`ksh93`版本的兼容性上不会有问题。

联系我们

请将关于本书的意见和问题通过以下地址提供给出版商：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

100035 北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室
奥莱利技术咨询（北京）有限公司

O'Reily公司是世界性的计算机信息出版公司。我们永远乐意听到读者对出版物的意见，包括如何让本书可以更好的建议，指正本书的错误或是读者建议本书往后改版时应该再加入的其他主题。以下是英文原书的联络数据：

http://oreilly.com/catalog/9780596005955/index.html

如果想要发表关于本书的评论和技术问题，请发邮件至：

bookquestions@oreilly.com
info@mail.oreilly.com.cn

关于图书、会议、资源中心和O'Reilly 网络的更多信息，请查看我们的站点：

http://www.oreilly.com
http://www.oreilly.com.cn

致谢

我们俩要感谢对方的付出。虽然我们未曾见面，但协同作业的工作一直进行得很好。除了对彼此献上最热忱的感激，还要谢谢我们挚爱的妻子，谢谢她们在这段期间的贡献、耐心、爱，以及在此书编写期间的支持。

bash的维护人员 Chet Ramey 回答了许许多多与POSIX Shell相关的细微问题。AT&T Research的 Glenn Fowler 与 David Korn，与 GNU Project的 Jim Meyering，也提供不少解答。我们以字母顺序排列下面要致谢的人：Keith Bostic、George Coulouris、Mary Ann Horton、Bill Joy、Rob Pike、Hugh Redelmeier、Dennis Ritchie，他们为UNIX的历史问题解惑。由于O'Reilly Media的Nat Torkington、Allison Randall、Tatiana Diaz的指导，让本书由概念逐渐成型。感激O'Reilly的Robert Romano将我们原始的ASCII草图与pic略图作成图片。也谢谢Angela Howard为本书制作全面性索引，造福读者。

以下仍以字母顺序：谢谢Geoff Collyer、Robert Day、Leroy Eide、John Halleck、Mark Lucking与Henry Spencer为本书初稿再作技术上的检阅，及 Sean Burke审阅第 2 版草稿。谢谢他们提供无价且非常有帮助的回馈。

UNIX Guru的UNIX Guru（精神导师）Henry Spencer，谢谢你为本书作的序。

感谢University of Utah的Electrical and Computer Engineering、Mathematics，与Physics等系所，以及 Center for High-Performance Computing，提供UNIX 系统供访问；还有IBM 与 Hewlett-Packard提供访客访问，让我们取得编写本书所需的软件以供检测，谢谢，谢谢他们。

—— Arnold Robbins
—— Nelson H.F. Beebe

第 1 章

背景知识

本章将简述 UNIX 系统的发展史。了解 UNIX 在何处开发、如何开发，以及它的设计动机。这有助于用户善加利用 UNIX 所提供的工具。此外，本章将介绍软件工具的设计原则。

1.1 UNIX 简史

或许你对于 UNIX 的发展史已有些了解，并且已经有很多介绍 UNIX 完整发展历史的资料。这里，我们只想让你知道：UNIX 是在何种环境下诞生的，以及它如何影响软件工具的设计。

UNIX 最初是由贝尔电话实验室（Bell Telephone Laboratories，注 1）的计算机科学研究中心（Computing Sciences Research Center）开发的。第一版诞生于 1970 年 —— 也就是在贝尔实验室（Bell Labs）退出 Multics 项目不久。在 UNIX 广受欢迎的功能中，有许多便是来自 Multics 操作系统。其中最著名的有：将设备视为文件，以及特意不将命令解释器（command interpreter）或 Shell 整合到操作系统中。更完整的历史信息可在 *http://www.bell-labs.com/history/unix* 找到。

由于 UNIX 是在面向研究的环境下开发的，因而没有必须生产或销售成品的盈利压力。这使其具有下列优势：

- 系统由用户自行开发。他们使用这套系统来解决每天所遇到的计算问题。

注 1：该名称至今已变更数次。本书从这里开始都以口语式名称“贝尔实验室”(Bell Labs) 称呼它。

- 研究人员可以不受拘束地进行实验，必要时也可任意变换程序。由于用户群不大，若程序有必要整个重写，多半也不会太难。由于用户即为开发人员，发现问题时便能随即修正，有地方需要加强时，也可以马上就做。

  UNIX 已历经数个版本，各个版本以字母 V 加上数字作为简称，如 V6、V7，等等。（正式名称则是遵循发行的使用手册的修订次数编号来命名，例如 First Edition、Second Edition，等等。这两种名称的对应其实很直接：V6 = Sixth Edition、V7 = Seventh Edition。和大多数有经验的 UNIX 程序员一样，这两种命名方式我们都会用到）。影响最深远的 UNIX 系统是 1979 年所发行的第 7 版（Seventh Edition），但是在最初的几年，它仅应用于学术教育机构领域。值得一提的是：第 7 版的系统同时提出了 `awk` 与 Bourne Shell，这二者是 POSIX Shell 的基础，同时，第一本讨论 UNIX 的书也在此诞生。

- 贝尔实验室的研究人员都是计算机科学家。他们所设计的系统不单单是自己使用，还要分享给同事 —— 这些人一样也是计算机科学家。因此，衍生出“务实”（no nonsense）的设计模式：程序会执行你所赋予的任务，但不会跟你对话，也不会问一堆“你确定吗？”之类的问题。

- 除了精益求精，在设计与问题解决上，他们也不断地追求“优雅”（elegance）。关于“优雅”有一个贴切的定义：简单就是力量（power cloaked in simplicity，注 2）。贝尔实验室自由的环境，所造就的不仅是一个可用的系统，也是一个优雅的系统。

当然，自由同样也带来了一些缺点。当 UNIX 流传至开发环境以外的地方，这些问题也逐一浮现：

- 工具程序之间存在许多不一致的地方。例如，同样的选项字母，在不同程序之间有完全不一样的定义；或是相同的工作却需要指定不同的选项字母。此外，正则表达式的语法在不同程序之间用法类似，却又不完全一致，易产生混淆 —— 这种情况其实可以避免。（直至正则表达式的重要性受到认可，其模式匹配机制才得以收录在标准程序库中。）

- 诸多工具程序具有缺陷，例如输入行（input lines）的长度，或是可打开的文件个数，等等。（现行的系统多半已经修正这些缺陷。）

- 有时程序并未经过彻底测试，这使得它们在执行的时候一不小心就会遭到破坏。这

注 2： 我最初是在 20 世纪 80 年代从 Dan Forsyth 口中听到这个定义的。

可能会导致核心转储（core dumps，译注 1），令用户不知所措。幸好，现行的UNIX系统极少会面临这样的问题。

- 系统的文档尽管大致上内容完备，但通常极为简单。使得用户在学习时很难找到所需要的信息（注 3）。

本书之所以将重点放在文本（而非二进制）数据的处理与运用上，是由于UNIX早期的发展都源自于对文本处理的强烈需求，不过除此之外还有另外的重要理由（马上会讨论到）。事实上，贝尔实验室专利部门（Bell Labs Patent Department）在UNIX系统上所使用的第一套产品，就是进行文本处理和编排工作的。

最初的UNIX机器（Digital Equipment Corporation PDP-11s）不能运行大型程序。要完成复杂的工作，得先将它分割成更小的工作，再用程序来完成这些更小的工作。某些常见的工作（从数据行中取出某些字段、替换文本，等等）也常见于许多大型项目，最后就成了标准工具。人们认为这种自然而生的结果是件好事：由于缺乏大型的解决空间，因而产生了更小、更简单、更专用的程序。

许多人在UNIX的使用上采用半独立的工作方式，重复套用彼此间的程序。由于版本之间的差异，而且不需要标准化，导致许多日常工具程序的发展日渐分歧。举例来说，`grep`在某系统里使用`-i`来表示“查找时忽略大小写”，但在另一个系统中，却使用`-y`来代表同样的事！无独有偶，这种怪事也发生在许多工具程序上。还有，一些常用的小程序可能会取相同的名字，针对某个UNIX版本所编写的Shell程序，不经修改可能无法在另一个版本上执行。

最后，对常用标准工具组与选项的需求终于明朗化，POSIX标准即为最后的结果。现行标准IEEE Std. 1003.1-2004包含了C的库层级的主题，还有Shell语言与系统工具及其选项。

好消息是，在这些标准上所做的努力有了回报。现在的商用UNIX系统，以及可免费使

译注 1：在UNIX系统中，常将“主内存”（main memory）称为核心（core），因为在使用半导体作为内存材料之前，便是使用核心（core）。而核心映像（core image）就是“进程”（process）执行当时的内存内容。当进程发生错误或收到“信号”（signal）而终止执行时，系统会将核心映像写入一个文件，以作为调试之用，这就是所谓的核心转储（core dump）。

注 3：系统文档分成两个部分：参考手册与使用手册。后者是系统各功能的教学手册。虽然把整份文件读完就可能学会UNIX——事实上有许多人（包括作者）真的是这么做，不过现今的系统，已不再附上打印好的文件。

用的同类型产品，例如GNU/Linux与BSD衍生系统，都兼容POSIX。这样一来，学习UNIX变得更容易，编写可移植的Shell脚本也成为可能（详见第14章）。

值得注意的是：POSIX并非UNIX标准化的唯一成果，POSIX之外仍有其他标准。例如，欧洲计算机制造商协会自行发起了一套名为X/Open的标准。其中最受欢迎的是1988年首度出现的XPG4（X/Open Portability Guide，Fourth Edition）。另外还有XPG5，其更广为人知的名称为UNIX 98标准，或Single UNIX Specification。XPG5很大程度上把POSIX纳入为一个子集，同样深具影响力（注4）。

XPG标准在措辞上可能不够严谨，但其内容却较为广泛，其目标是将现存于UNIX系统上实际用到的各种功能正式生成文档。（POSIX的目的在于建立一套正式的标准，让从头开始的实践者有指导方针可以套用 —— 即便是在非UNIX的平台上。因此，许多UNIX系统上常见的功能，一开始就排除在POSIX标准之外）。2001 POSIX标准由于纳入了X/Open System Interface Extension（XSI）而有了双重身份，也叫做XPG6，这是它首度正式扩张POSIX版图。此文档的特色在于：让系统不只兼容POSIX，也兼容于XSI。所以，当你在编写工具或应用程序时，必须参考的正式文件只有一份（就叫做Single UNIX Standard）。

本书自始至终都把重点放在根据POSIX标准所定义的Shell语言与UNIX工具程序。重点部分也会加入XSI定义的说明，因为你很可能会用得到。

1.2 软件工具的原则

随着时间的流逝，人们开发出了一套设计与编写软件工具的原则。在本书用来解决问题的程序中，你将会看到这些原则的应用示例。好的软件工具应该具备下列特点：

一次做好一件事

在很多方面，这都是最重要的原则。若程序只做一件事，那么无论是设计、编写、调试、维护，以及生成文件都会容易得多。举例来说，对于用来查找文件中是否有符合样式的`grep`程序，不应该指望用它来执行算术运算。

这个原则的结果，自然就是会不断产生出更小、更专用于特定功能的程序，就像专业木匠的工具箱里，永远会有一堆专为特定用途所设计的工具。

处理文本行，不要处理二进制数据

文本行是UNIX的通用格式。当你在编写自己的工具程序时便会发现，内含文本行的数据文件很好处理，你可以用任何唾手可得的文本编辑器来编辑它，也可以让这

注4： X/Open的出版物列表可参见*http://www.opengroup.org/publications/catalog/*。

些数据在网络与各种机器架构之间传输。使用文本文件更有助于任何自定义工具与现存的 UNIX 程序之间的结合。

使用正则表达式

正则表达式（regular expression）是很强的文本处理机制。了解它的运作模式并加以使用，可适度简化编写命令脚本（script）的工作。

此外，虽然正则表达式多年来在工具与UNIX 版本上不断在变化，但 POSIX 标准仅提供两种正则表达式。你可以利用标准的库程序进行模式匹配的工作。这样就可以编写出专用的工具程序，用于与`grep`一致的正则表达式（POSIX称之为基本型正则表达式，Basic Regular Expressions，BRE），或是用于与`egrep`一致的正则表达式（POSIX称之为扩展型正则表达式，Extended Regular Expressions，ERE）。

默认使用标准输入 / 输出

在未明确指定文件名的情况下，程序默认会从它的标准输入读取数据，将数据写到它的标准输出，至于错误信息则会传送到标准错误输出（这部分将于第2章讨论）。以这样的方式来编写程序，可以轻松地让它们成为数据过滤器（filter），例如，组成部分的规模越大，越需要复杂的管道（pipeline）或脚本来处理。

避免喋喋不休

软件工具的执行过程不该像在“聊天”（chatty）。不要将“开始处理”（`starting processing`）、“即将完成”（`almost done`）或是“处理完成”（`finished processing`）这类信息放进程序的标准输出（至少这不该是默认状态）。

当你有意将一些工具串成一条管道时，例如：

```
tool_1 < datafile | tool_2 | tool_3 | tool_4 > resultfile
```

若每个工具都会产生“正处理中”（yes I'm working）这样的信息并送往管道，那么别指望执行结果会像预期的一样。此外，若每个工具都将自己的信息传送至标准错误输出，那么整个屏幕画面就会布满一堆无用的过程信息。在工具程序的世界里，没有消息就是好消息。

这个原则其实还有另外一个含义。一般来说，UNIX 工具程序一向遵循“你叫它做什么，你就会得到什么”的设计哲学。它们不会问“你确定吗？”（are you sure?）这种问题，当用户键入 `rm somefile`，UNIX 的设计人员会认为用户知道自己在做什么，然后毫无疑问地 `rm` 删除掉要删除的文件（注 5）。

注 5：如果你真觉得这样不好，`rm` 的 `-i` 选项可强制 `rm` 给你提示以做确认，这么一来，当你要求删除可疑文件时，`rm` 便会提示确认它。一直以来，应该“永远不要提示”还是应该“永远得到提示”是个争议的话题，值得用户深思。

输出格式必须与可接受的输入格式一致

专业的工具程序认为遵循某种格式的输入数据，例如标题行之后接着数据行，或在行上使用某种字段分隔符等，所产生的输出也应遵循与输入一致的规则。这么做的好处是，容易将一个程序的执行结果交给另一个程序处理。

举例来说，netpbm程序集（注 5）是用来处理以 Portable BitMap（PBM）格式保存的图像文件（注 6）。这些文件内含 bitmapped 图像，并使用定义明确的格式加以绘制。每个读取PBM 文件的工具程序，都会先以某种格式来处理文件内的图像，然后再以 PBM 的格式写回文件。这么一来，便可以组合简单的管道来执行复杂的图像处理，例如先缩放影像后，再旋转方向，最后再把颜色调淡。

让工具去做困难的部分

虽然 UNIX 程序并非完全符合你的需求，但是现有的工具或许已经可以为你完成 90% 的工作。接下来，若有需要，你可以编写一个功能特定的小型程序来完成剩下的工作。与每次都从头开始来解决各个问题相比，这已经让你省去许多工作了。

构建特定工具前，先想想

如前所述，若现存系统里就是没有需要的程序，可以花点时间构建满足所需的工具。然而，动手编写一个能够解决问题的程序前，请先停下来想几分钟。你所要做的事，是否有其他人也需要做？这个特殊的工作是否有可能是某个一般问题的一个特例？如果是的话，请针对一般问题来编写程序。当然，这么做的时侯，无论是在程序的设计或编写上，都应该遵循前面所提到的几项原则。

1.3 小结

UNIX 原为贝尔实验室的计算机科学家所开发的产品。由于没有盈利上的压力，再加上 PDP-11 小型计算机的能力有限，因而程序都以小型、优雅为圭臬。也因为没有盈利上的压力，系统之间并非完全一致，学习上也不太容易。

随着 UNIX 持续地流行，各种版本陆续开发出来（尤其是衍生自 System V 和 BSD 的版本），Shell 脚本层次的可移植性也日益困难。幸好，POSIX 标准成熟后，几乎所有商用 UNIX 系统与免费的 UNIX 版本都兼容 POSIX。

注 6： 这套程序并非 UNIX 工具集的标准配备，不过 GNU/Linux 与 BSD 系统上通常都会安装。其网站位于 *http://netpbm.sourceforge.net/*。可按照 Sourceforge 项目网页的指示，下载源代码。

注 7： 有三种格式。若你的系统里有安装 netpbm，可参阅 *pnm*(5)手册页。

之所以会在这里指出软件工具的设计原则，主要是为了提供开发与使用UNIX工具集的指导方针。让软件工具的设计原则成为思考习惯，将有助于编写简洁的Shell程序和正确使用UNIX 工具。

第2章

入门

当需要计算机帮你做些什么时，最好用对工具。你不会用文字编辑器来做支票簿的核对，也不会用计算器来写策划方案。同理，当你需要程序语言协助完成工作时，不同的程序语言用于不同的需求。

Shell脚本最常用于系统管理工作，或是用于结合现有的程序以完成小型的、特定的工作。一旦你找出完成工作的方法，可以把用到的命令串在一起，放进一个独立的程序或脚本（script）里，此后只要直接执行该程序便能完成工作。此外，如果你写的程序很有用，其他人可以利用该程序当作一个黑盒（black box）来使用，它是一个可以完成工作的程序，但我们不必知道它是如何完成的。

本章中，我们会先对脚本编程（scripting）语言和编译型（compiled）语言做个简单的比较，再从如何编写简单的Shell脚本开始介绍起。

2.1 脚本编程语言与编译型语言的差异

许多中型、大型的程序都是用编译型语言写成，例如Fortran、Ada、Pascal、C、C++或Java。这类程序只要从源代码（source code）转换成目标代码（object code），便能直接通过计算机来执行（注1）。

编译型语言的好处是高效，缺点则是：它们多半运作于底层，所处理的是字节、整数、浮点数或是其他机器层级的对象。例如，在C++里，就很难进行“将一个目录里所有的文件复制到另一个目录中”之类的简单操作。

注1：　这种说法在Java上并不完全正确，不过已相当接近我们所说的情况了。

脚本编程语言通常是解释型(interpreted)的。这类程序的执行,是由解释器(interpreter)读入程序代码,并将其转换成内部的形式,再执行(注2)。请注意,解释器本身是一般的编译型程序。

2.2 为什么要使用Shell脚本

使用脚本编程语言的好处是,它们多半运行在比编译型语言还高的层级,能够轻易处理文件与目录之类的对象。缺点是:它们的效率通常不如编译型语言。不过权衡之下,通常使用脚本编程还是值得的:花一个小时写成的简单脚本,同样的功能用C或C++来编写实现,可能需要两天,而且一般来说,脚本执行的速度已经够快了,快到足以让人忽略它性能上的问题。脚本编程语言的例子有awk、Perl、Python、Ruby与Shell。

因为Shell似乎是各UNIX系统之间通用的功能,并且经过了POSIX的标准化。因此,Shell脚本只要“用心写”一次,即可应用到很多系统上。因此,之所以要使用Shell脚本是基于:

简单性
: Shell是一个高级语言;通过它,你可以简洁地表达复杂的操作。

可移植性
: 使用POSIX所定义的功能,可以做到脚本无须修改就可在不同的系统上执行。

开发容易
: 可以在短时间内完成一个功能强大又好用的脚本。

2.3 一个简单的脚本

让我们从简单的脚本开始。假设你想知道,现在系统上有多少人登录。who命令可以告诉你现在系统有谁登录:

```
$ who
george     pts/2        Dec 31 16:39    (valley-forge.example.com)
betsy      pts/3        Dec 27 11:07    (flags-r-us.example.com)
benjamin   dtlocal      Dec 27 17:55    (kites.example.com)
jhancock   pts/5        Dec 27 17:55    (:32)
camus      pts/6        Dec 31 16:22
tolstoy    pts/14       Jan  2 06:42
```

注2: 尽管*http://foldoc.doc.ic.ac.uk/foldoc/foldoc.cgi?Ousterhout's+dichotomy*试图为编译型与脚本编程语言的差异下定义,但是人们对此一直很难达成共识。

在大型的、多用户的系统上，所列出来的列表可能很长，在你能够计算用户个数之前，列表早已滚动出屏幕画面，因此每次做这件事的时候，都会让你觉得很麻烦。这正是进行自动化的好时机。计算用户总数的方法尚未提到。对此，我们可以利用 wc（字数计算）程序，它可以算出行数（line）、字数（word）与字符数（character）。在此例中，我们用的是 wc -l，也就是只算出行数：

```
$ who | wc -l                计算用户个数
      6
```

|（管道）符号可以在两程序之间建立管道（pipeline）：who 的输出，成了 wc 的输入，wc 所列出的结果就是已登录用户的个数。

下一步则是将此管道转变成一个独立的命令。方法是把这条命令输入一个一般的文件中，然后使用 chmod 为该文件设置执行的权限，如下所示：

```
$ cat > nusers               建立文件，使用 cat 复制终端的输入
who | wc -l                  程序的内容
^D                           Ctrl-D 表示 end-of-file
$ chmod +x nusers            让文件拥有执行的权限
$ ./nusers                   执行测试
      6                      输出我们要的结果
```

这展现了小型 Shell 脚本的典型开发周期：首先，直接在命令行（command line）上测试。然后，一旦找到能够完成工作的适当语法，再将它们放进一个独立的脚本里，并为该脚本设置执行的权限。之后，就能直接使用该脚本。

2.4 自给自足的脚本：位于第一行的 #!

当 Shell 执行一个程序时，会要求 UNIX 内核启动一个新的进程（process），以便在该进程里执行所指定的程序。内核知道如何为编译型程序做这件事。我们的 nusers Shell 脚本并非编译型程序；当 Shell 要求内核执行它时，内核将无法做这件事，并回应“not executable format file”（不是可执行的格式文件）错误信息。Shell 收到此错误信息时，就会说“啊哈，这不是编译型程序，那么一定是 Shell 脚本”，接着会启动一个新的 /bin/sh（标准 Shell）副本来执行该程序。

当系统只有一个 Shell 时，“退回到 /bin/sh”的机制非常方便。但现行的 UNIX 系统都会拥有好几个 Shell，因此需要通过一种方式，告知 UNIX 内核应该以哪个 Shell 来执行所指定的 Shell 脚本。事实上，这么做有助于执行机制的通用化，让用户得以直接引用任何的程序语言解释器，而非只是一个命令 Shell。方法是，通过脚本文件中特殊的第一行来设置：在第一行的开头处使用 #! 这两个字符。

当一个文件中开头的两个字符是#!时，内核会扫描该行其余的部分，看是否存在可用来执行程序的解释器的完整路径。(中间如果出现任何空白符号都会略过。)此外，内核还会扫描是否有一个选项要传递给解释器。内核会以被指定的选项来引用解释器，再搭配命令行的其他部分。举例来说，假设有一个csh脚本（注3），名为/usr/ucb/whizprog，它的第一行如下所示：

```
#! /bin/csh -f
```

再者，如果Shell的查找路径（后面会介绍）里有/usr/ucb，当用户键入whizprog -q /dev/tty01这条命令，内核解释#!这行后，便会以如下的方式来引用csh：

```
/bin/csh -f /usr/ucb/whizprog -q /dev/tty01
```

这样的机制让我们得以轻松地引用任何的解释器。例如我们可以这样引用独立的awk程序：

```
#! /bin/awk -f
此处是awk程序
```

Shell脚本通常一开始都是#! /bin/sh。如果你的/bin/sh并不符合POSIX标准，请将这个路径改为符合POSIX标准的Shell。下面是几个初级的陷阱（gotchas），请特别留意：

- 当今的系统，对#!这一行的长度限制从63到1024个字符（character）都有。请尽量不要超过64个字符。(表2-1列出了各系统的长度限制。)
- 在某些系统上，命令行部分（也就是要传递给解释器执行的命令）包含了命令的完整路径名称。不过有些系统却不是这样；命令行的部分会原封不动地传给程序。因此，脚本是否具可移植性取决于是否有完整的路径名称。
- 别在选项（option）之后放置任何空白，因为空白也会跟着选项一起传递给被引用的程序。
- 你需要知道解释器的完整路径名称。这可以用来规避可移植性问题，因为不同的厂商可能将同样的东西放在不同的地方（例如/bin/awk和/usr/bin/awk）。
- 一些较旧的系统上，内核不具备解释#!的能力，有些Shell会自行处理，这些Shell对于#!与紧随其后的解释器名称之间是否可以有空白，可能有不同的解释。

注3：/bin/csh是C Shell的命令解释器，由加州大学伯克利分校所开发。本书不讨论C Shell程序设计的原因很多，其中最重要一的点是：就脚本的编写来说，大多数人认为它不是个好用的Shell，另一个原因则是它并未被POSIX标准化。

表 2-1 列出了各 UNIX 系统对于 #! 行的长度限制（这些都是通过经验法则得知的）。结果出乎意料：有一半以上的数字都不是二的次方。

表 2-1：各系统对 #! 行的长度限制

平台	操作系统版本	最大长度
Apple Power Mac	Mac Darwin 7.2 (Mac OS 10.3.2)	512
Compaq/DEC Alpha	OSF/1 4.0	1024
Compaq/DEC/HP Alpha	OSF/1 5.1	1000
GNU/Linux 注	Red Hat 6, 7, 8, 9; Fedora 1	127
HP PA-RISC and Itanium-2	HP-UX 10, 11	127
IBM RS/6000	AIX 4.2	255
Intel x86	FreeBSD 4.4	64
Intel x86	FreeBSD 4.9, 5.0, 5.1	128
Intel x86	NetBSD 1.6	63
Intel x86	OpenBSD 3.2	63
SGI MIPS	IRIX 6.5	255
Sun SPARC, x86	Solaris 7, 8, 9, 10	1023

注：所有架构。

POSIX 标准对 #! 的行为模式保留未定义（unspecified）状态。此状态是“只要一直保持 POSIX 兼容性，这是一个扩展功能”的标准说法。

本书接下来的所有脚本开头都会有 #! 行。下面是修订过的 nusers 程序：

```
$ cat nusers                    显示文件内容
#! /bin/sh -                    神奇的 #! 行

who | wc -l                     所要执行的命令
```

选项 —— 表示没有 Shell 选项；这是基于安全上的考虑，可避免某种程度的欺骗式攻击（spoofing attack）。

2.5 Shell 的基本元素

本节要介绍的是，适用于所有 Shell 脚本的基本元素。通过以交互的方式使用 Shell，你会慢慢熟悉的。

2.5.1 命令与参数

Shell最基本的工作就是执行命令。以互动的方式来使用Shell很容易了解这一点：每键入一道命令，Shell就会执行。像这样：

```
$ cd work ; ls -l whizprog.c
-rw-r--r--    1 tolstoy   devel       30252 Jul  9 22:52 whizprog.c
$ make
...
```

以上的例子展现了UNIX命令行的原理。首先，格式很简单，以空白（Space键或Tab键）隔开命令行中各个组成部分。

其次，命令名称是命令行的第一个项目。通常后面会跟着选项（option），任何额外的参数（argument）都会放在选项之后。如下的语法是不可能出现的：

```
COMMAND=CD,ARG=WORK
COMMAND=LISTFILES,MODE=LONG,ARG=WHIZPROG.C
```

这类语法多半出现在正在设计UNIX时的传统大型系统上。UNIX Shell的自由格式语法在当时是一大革新，大大增强了Shell脚本的可读性。

第三，选项的开头是一个破折号（或减号），后面接着一个字母。选项是可有可无的，有可能需要加上参数（例如`cc -o whizprog whizprog.c`）。不需要参数的选项可以合并：例如，`ls -lt whizprog.c`比`ls -l -t whizprog.c`更方便（后者当然也可以，只是得多些录入）。

长选项的使用越来越普遍，特别是标准工具的GNU版本，以及在X Window System（X11）下使用的程序。例如：

```
$ cd whizprog-1.1
$ patch --verbose --backup -p1 < /tmp/whizprog-1.1-1.2-patch
```

长选项的开头是一个破折号还是两个（如上所示），视程序而定。（`< /tmp/whizprog-1.1-1.2-patch`是一个I/O重定向。它会使得`patch`从`/tmp/whizprog-1.1-1.2-patch`文件而不是从键盘读取输入。I/O重定向也是重要的基本概念之一，本章稍后会谈到）。

以两个破折号（`--`）来表示选项结尾的用法，源自System V，不过已被纳入POSIX标准。自此之后命令行上看起来像选项的任何项目，都将一视同仁地当成参数处理（例如，视为文件名）。

最后要说的是，分号（`;`）可用来分隔同一行里的多条命令。Shell会依次执行这些命令。

如果你使用的是 & 符号而不是分号，则 Shell 将在后台执行其前面的命令，这意味着，Shell 不用等到该命令完成，就可以继续执行下一个命令。

Shell 识别三种基本命令：内建命令、Shell 函数以及外部命令：

- 内建命令就是由 Shell 本身所执行的命令。有些命令是由于其必要性才内建的，例如 cd 用来改变目录，read 会将来自用户（或文件）的输入数据传给 Shell 变量。另一种内建命令的存在则是为了效率，其中最典型的就是 test 命令（稍后在 6.2.4 节会谈到），编写脚本时会经常用到它。另外还有 I/O 命令，例如 echo 与 printf。
- Shell 函数是功能健全的一系列程序代码，以 Shell 语言写成，它们可以像命令那样引用。稍后会在 6.5 节讨论这个部分。此处，我们只需要知道，它们可以引用，就像一般的命令那样。
- 外部命令就是由 Shell 的副本（新的进程）所执行的命令，基本的过程如下：
  a. 建立一个新的进程。此进程即为 Shell 的一个副本。
  b. 在新的进程里，在 PATH 变量内所列出的目录中，寻找特定的命令。/bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin 为 PATH 变量典型的默认值。当命令名称含有斜杠（/）符号时，将略过路径查找步骤。
  c. 在新的进程里，以所找到的新程序取代执行中的 Shell 程序并执行。
  d. 程序完成后，最初的 Shell 会接着从终端读取的下一条命令，或执行脚本里的下一条命令。如图 2-1 所示。

以上只是基本程序。当然，Shell 可以做的事很多，例如变量与通配字符的展开、命令与算术的替换等。接下来，本书会一一探讨这些话题。

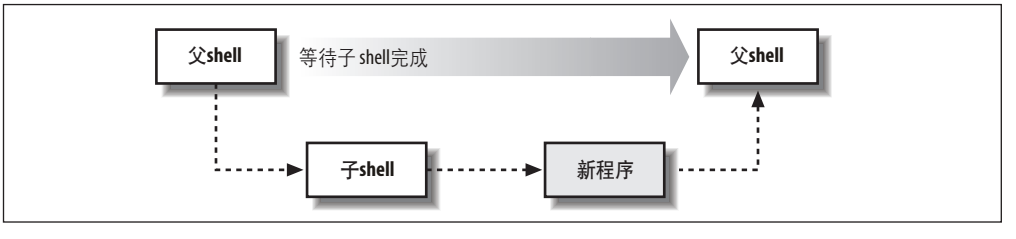

图 2-1：程序执行

2.5.2 变量

变量（variable）就是为某个信息片段所起的名字，例如 first_name 或 driver_lic_no。所有程序语言都会有变量，Shell 也不例外。每个变量都有一个值（value），这是由你分

配给变量的内容或信息。在 Shell 的世界里，变量值可以是（而且通常是）空值，也就是不含任何字符。这是合理的，也是常见的、好用的特性。空值就是 null，本书接下来的部分将会经常用到这个术语。

Shell变量名称的开头是一个字母或下划线符号，后面可以接着任意长度的字母、数字或下划线符号。变量名称的字符长度并无限制。Shell变量可用来保存字符串值，所能保存的字符数同样没有限制。Bourne Shell是少数几个早期的UNIX程序里，遵循不限制（no arbitrary limit）设计原则的程序之一。例如：

```
$ myvar=this_is_a_long_string_that_does_not_mean_much    分配变量值
$ echo $myvar                                            打印变量值
this_is_a_long_string_that_does_not_mean_much
```

变量赋值的方式为：先写变量名称，紧接着 = 字符，最后是新值，中间完全没有任何空格。当你想取出 Shell 变量的值时，需于变量名称前面加上 $ 字符。当所赋予的值内含空格时，请加上引号：

```
first=isaac  middle=bashevis  last=singer   单行可进行多次赋值
fullname="isaac bashevis singer"            值中包含空格时使用引号
oldname=$fullname                           此处不需要引号
```

如上例所示，当变量作为第二个变量的新值时，不需要使用双引号（参见 7.7 节），但是使用双引号也没关系。不过，当你将几个变量连接起来时，就需要使用引号了：

```
fullname="$first $middle $last"             这里需要双引号
```

2.5.3 简单的 echo 输出

这里要看的是echo命令如何显示myvar变量的值，这是很可能会在命令行里使用到的情况。echo的任务就是产生输出，可用来提示用户，或是用来产生数据供进一步处理。

原始的echo 命令只会将参数打印到标准输出，参数之间以一个空格隔开，并以换行符号（newline）结尾。

```
$ echo Now is the time for all good men
Now is the time for all good men
$ echo to come to the aid of their country.
to come to the aid of their country.
```

不过，随着时间的流逝，有各种版本的echo 开发出来。BSD 版本的 echo 看到第一个参数为 -n 时，会省略结尾的换行符号。例如（下划线符号表示终端画面的光标）：

```
$ echo -n "Enter your name: "          显示提示
Enter your name: _                    键入数据
```

echo

语法

echo [*string* ...]

用途

产生 Shell 脚本的输出。

主要选项

无。

行为模式

echo 将各个参数打印到标准输出，参数之间以一个空格隔开，并以换行符号结束。它会解释每个字符串里的转义序列（escape sequences）。转义序列可用来表示特殊字符，以及控制其行为模式。

警告

UNIX 各版本间互不相同的行为模式使得 echo 的可移植性变得很困难，不过它仍是最简单的一种输出方式。

许多版本都支持 -n 选项。如果有支持，echo 的输出会省略最后的换行符号。这适合用来打印提示字符串。不过，目前 echo 符合 POSIX 标准的版本并未包含此选项。

System V 版本的 echo 会解释参数里特殊的转义序列（稍后会说明）。例如，\c 用来指示 echo 不要打印最后的换行符号：

```
$ echo "Enter your name: \c"          显示提示
Enter your name: _                    键入数据
```

转义序列可用来表示程序中难以键入或难以看见的字符。echo 遇到转义序列时，会打印相应的字符。有效的转义序列如表 2-2 所示。

表 2-2：echo 的转义序列

序列	说明
\a	警示字符，通常是 ASCII 的 BEL 字符
\b	退格（Backspace）
\c	输出中忽略最后的换行字符（Newline）。这个参数之后的任何字符，包括接下来的参数，都会被忽略掉（不打印）
\f	清除屏幕（Formfeed）
\n	换行（Newline）

表 2-2：echo 的转义序列（续）

序列	说明
\r	回车（Carriage return）
\t	水平制表符（Horizontal tab）
\v	垂直制表符（Vertical tab）
\\	反斜杠字符
\0*ddd*	将字符表示成 1 到 3 位的八进制数值

实际编写Shell脚本的时候，\a序列通常用来引起用户的注意；\0ddd序列最有用的地方，就是通过送出终端转义序列进行（非常）原始的光标操作，但是不建议这么做。

由于很多系统默认以BSD的行为模式来执行echo，所以本书只会使用它的最简单形式。比较复杂的输出，我们会使用printf。

2.5.4 华丽的 printf 输出

由于echo有版本上的差异，所以导致UNIX版本间可移植性的头疼问题。在POSIX标准化的首次讨论中，与会成员无法在如何标准化echo上达到共识，便提出折衷方案。虽然echo是POSIX标准的一部分，但该标准却未说明当第一个参数是-n或有任何参数包含转义序列的行为模式。取而代之的是，将其行为模式保留为实现时定义(implementation-defined)；也就是说，各厂商必须提供说明文件，描述其echo版本的做法（注4）。事实上，只要是使用最简单的形式，其echo的可移植性不会有问题。相对来看，Ninth Edition Research UNIX 系统上所采用的printf命令，比echo更灵活，却也更复杂。

printf命令模仿C程序库（library）里的printf()库程序（library routine）。它几乎复制了该函数所有的功能（见*printf(3)*的在线说明文档），如果你曾使用C、C++、awk、Perl、Python或Tcl写过程序，对它的基本概念应该不陌生。当然，它在Shell层级的版本上，会有些差异。

如同echo命令，printf命令可以输出简单的字符串：

```
printf "Hello, world\n"
```

你应该可以马上发现，最大的不同在于：printf不像echo那样会自动提供一个换行符号。你必须显式地将换行符号指定成\n。printf命令的完整语法分为两部分：

```
printf format-string [arguments ...]
```

注4：值得玩味的是，现行版本的标准中，说明echo在本质上等同于System V版本，后者会处理其参数中的转义序列，但不处理-n。

第一部分是一个字符串，用来描述输出的排列方式，最好为此字符串加上引号。此字符串包含了按字面显示的字符（characters to be printed literally）以及格式声明（format specifications），后者是特殊的占位符（placeholders），用来描述如何显示相应的参数（argument）。

第二部分是与格式声明相对应的参数列表（argument list），例如一系列的字符串或变量值。（如果参数的个数比格式声明还多，则 `printf` 会循环且依次地地使用格式字符串里的格式声明，直到处理完参数）。格式声明分成两部分：百分比符号（`%`）和指示符（specifier）。最常用的格式指示符（format specifier）有两个，`%s` 用于字符串，而 `%d` 用于十进制整数。

格式字符串中，一般字符会按字面显示。转义序列则像 `echo` 那样，解释后再输出成相应的字符。格式声明以 `%` 符号开头，并以定义的字母集中的一个来结束，用来控制相应参数的输出。例如，`%s` 用于字符串的输出：

```
$ printf "The first program always prints '%s, %s!'\n" Hello world
The first program always prints 'Hello, world!'
```

`printf` 的所有详细说明见 7.4 节。

2.5.5 基本的 I/O 重定向

标准输入 / 输出（standard I/O，注 5）可能是软件设计原则里最重要的概念了。这个概念就是：程序应该有数据的来源端、数据的目的端（数据要去的地方）以及报告问题的地方，它们分别被称为标准输入（standard input）、标准输出（standard output）以及标准错误输出（standard error）。程序不必知道也不用关心它的输入与输出背后是什么设备：是磁盘上的文件、终端、磁带机、网络连接或是另一个执行中的程序！当程序启动时，可以预期的是，标准输出入都已打开，且已准备好供其使用。

许多 UNIX 程序都遵循这一设计原则。默认的情况下，它们会读取标准输入、写入标准输出，并将错误信息传递到标准错误输出。这类程序常叫做过滤器（filter），你马上就会知道这么叫的原因。默认的标准输入、标准输出以及标准错误输出都是终端，这点可通过 `cat` 程序得知：

```
$ cat                                   未指定任何参数，读取标准输入，写入标准输出
now is the time                         由用户键入
now is the time                         由 cat 返回
for all good men
for all good men
to come to the aid of their country
```

注 5：此处的 Standard I/O 请不要与 C 程序库的 standard I/O 程序库混淆了，后者的接口定义于 <stdio.h>，不过此程序库的工作一样是提供类似的概念给 C 程序使用。

```
to come to the aid of their country
^D                                  Ctrl-D，文件结尾
```

你可能想要知道，是谁替执行中的程序初始化标准输入、输出及错误输出的呢？毕竟，总应该有人来替执行中的程序打开这些文件，甚至是让用户在登录后能够看到交互的Shell界面。

答案就是在你登录时，UNIX便将默认的标准输入、输出及错误输出安排成你的终端。I/O重定向就是你通过与终端交互，或是在Shell脚本里设置，重新安排从哪里输入或输出到哪里。

2.5.5.1 重定向与管道

Shell提供了数种语法标记，可用来改变默认I/O的来源端与目的端。此处会先介绍基本用法，稍后再提供完整的说明。让我们由浅入深地依次介绍如下：

以< 改变标准输入

program < *file*可将*program*的标准输入修改为*file*：

```
tr -d '\r' < dos-file.txt ...
```

以> 改变标准输出

program > *file*可将*program*的标准输出修改为*file*：

```
tr -d '\r' < dos-file.txt > UNIX-file.txt
```

这条命令会先以tr将dos-file.txt里的ASCII carriage-return（回车）删除，再将转换完成的数据输出到UNIX-file.txt。dos-file.txt里的原始数据不会有变化。（tr命令在第5章有完整的说明。）

>重定向符（redirector）在目的文件不存在时，会新建一个。然而，如果目的文件已存在，它就会被覆盖掉；原本的数据都会丢失。

以>> 附加到文件

program >> *file*可将*program*的标准输出附加到*file*的结尾处。

如同>，如果目的文件不存在，>>重定向符便会新建一个。然而，如果目的文件存在，它不会直接覆盖掉文件，而是将程序所产生的数据附加到文件结尾处：

```
for f in dos-file*.txt
do
    tr -d '\r' < $f >> big-UNIX-file.txt
done
```

（for循环的介绍详见6.4节。）

以| 建立管道

program1 | *program2*可将*program1*的标准输出修改为*program2*的标准输入。

虽然 < 与 > 可将输入与输出连接到文件，不过管道（pipeline）可以把两个以上执行中的程序衔接在一起。第一个程序的标准输出可以变成第二个程序的标准输入。这么做的好处是，管道可以使得执行速度比使用临时文件的程序快上十倍。本书中有相当多篇幅都是在讨论如何将各类工具串在一起，置入越来越复杂且功能越来越强大的管道中。例如：

```
tr -d '\r' < dos-file.txt | sort > UNIX-file.txt
```

这条管道会先删除输入文件内的回车字符，在完成数据的排序之后，将结果输出到目的文件。

tr

语法

tr [*options*] *source-char-list replace-char-list*

用途

转换字符。例如，将大写字符转换成小写。选项可让你指定所要删除的字符，以及将一串重复出现的字符浓缩成一个。

常用选项

-c

取source-char-list的反义。tr要转换的字符，变成未列在source-char-list中的字符。此选项通常与-d或-s配合使用。

-C

与-c相似，但所处理的是字符（可能是包含多个字节的宽字符），而非二进制的字节值。参考“警告”的说明。

-d

自标准输入删除 *source-char-list*里所列的字符，而不是转换它们。

-s

浓缩重复的字符。如果标准输入中连续重复出现 *source-char-list*里所列的字符，则将其浓缩成一个。

行为模式

如同过滤器：自标准输入读取字符，再将结果写到标准输出。任何输入字符只要出现在*source-char-list*中，就会置换成*replace-char-list*里相应的字符。POSIX风格的字符与等效的字符集也适用，而且tr还支持replace-char-list中重复字符的标记法。相关细节请参考 *tr*(1)的在线说明文档。

警告

根据POSIX标准的定义，-c处理的是二进制字节值，而-C处理的是现行locale所定义的字符。直到2005年初，仍有许多系统不支持-C选项。

使用UNIX工具程序时，不妨将数据想象成水管里的水。未经处理的水，将流向净水厂，经过各类滤器的处理，最后产生适合人类饮用的水。

同样，编写脚本时，你通常已有某种输入格式定义下的原始数据，而需要处理这些数据后产生结果。（处理一词表示很多意思，例如排序、加和与平均、格式化以便于打印，等等。）从最原始的数据开始，然后构造一条管道，一步一步地，管道中的每个阶段都会让数据更接近要的结果。

如果你是UNIX新手，可以把<与>想象成数据的漏斗（funnels）——数据会从大的一端进入，由小的一端出来。

注意： 构造管道时，应该试着让每个阶段的数据量变得更少。换句话说，如果你有两个要完成的步骤与先后次序无关，你可以把会让数据量变少的那一个步骤放在管道的前面。这么做可以提升脚本的整体性能，因为UNIX只需要在两个程序间移动少的数据量，每个程序要做的事也比较少。

例如，使用`sort`排序之前，先以`grep`找出相关的行；这样可以让`sort`少做些事。

2.5.5.2 特殊文件：/dev/null与/dev/tty

UNIX系统提供了两个对Shell编程特别有用的特殊文件。第一个文件`/dev/null`，就是大家所熟知的位桶（bit bucket）。传送到此文件的数据都会被系统丢掉。也就是说，当程序将数据写到此文件时，会认为它已成功完成写入数据的操作，但实际上什么事都没做。如果你需要的是命令的退出状态（见6.2节），而非它的输出，此功能会很有用。例如，测试一个文件是否包含某个模式（pattern）：

```
if grep pattern myfile > /dev/null
then
    ...     找到模式时
else
    ...     找不到模式时
fi
```

相对地，读取`/dev/null`则会立即返回文件结束符号（end-of-file）。读取`/dev/null`的操作很少会出现在Shell程序里，不过了解这个文件的行为模式还是非常重要的。

另一个特殊文件为`/dev/tty`。当程序打开此文件时，UNIX会自动将它重定向到一个终端［一个实体的控制台（console）或串行端口（serial port），也可能是一个通过网络与窗口登录的伪终端（pseudoterminal）］再与程序结合。这在程序必须读取人工输入时（例如密码）特别有用。此外，用它来产生错误信息也很方便，只是比较少人这么做：

```
printf "Enter new password: "     提示输入
stty -echo                        关闭自动打印输入字符的功能
read pass < /dev/tty              读取密码
printf "Enter again: "            提示再输入一次
read pass2 < /dev/tty             再读取一次以确认
stty echo                         别忘了打开自动打印输入字符的功能
...
```

stty（set tty）命令用来控制终端（或窗口，注 6）的各种设置。-echo 选项用来关闭自动打印每个输入字符的功能；stty echo 用来恢复该功能。

2.5.6 基本命令查找

之前，我们曾提及 Shell 会沿着查找路径 $PATH 来寻找命令。$PATH 是一个以冒号分隔的目录列表，你可以在列表所指定的目录下找到所要执行的命令。所找到的命令可能是编译后的可执行文件，也可能是 Shell 脚本；从用户的角度来看，两者并无不同。

默认路径（default path）因系统而异，不过至少包含 /bin 与 /usr/bin，或许还包含存放X Windows程序的/usr/X11R6/bin，以及供本地系统管理人员安装程序的/usr/local/bin。例如：

```
$ echo $PATH
/bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin
```

名称为 bin 的目录用来保存可执行文件，bin 是 binary 的缩写。你也可以直接把 bin 解释成相应的英文字义 —— 存储东西的容器；这里所存储的是可执行的程序。

如果你要编写自己的脚本，最好准备自己的 bin 目录来存放它们，并且让 Shell 能够自动找到它们。这不难，只要建立自己的 bin 目录，并将它加入 $PATH 中的列表即可：

```
$ cd                              切换到 home 目 录
$ mkdir bin                       建立个人 bin 目录
$ mv nusers bin                   将我们的脚本置入该目录
$ PATH=$PATH:$HOME/bin            将个人的 bin 目录附加到 PATH
$ nusers                          试试看
      6                           Shell 有找到并执行它
```

要让修改永久生效，在 .profile 文件中把你的 bin 目录加入 $PATH，而每次登录时 Shell 都将读取 .profile 文件，例如：

```
PATH=$PATH:$HOME/bin
```

注 6：stty 可能是现有的 UNIX 命令中，最怪异且最复杂的一个。相关细节可参考 *stty*(1)的 manpage 或是《UNIX in a Nutshell》这本书。

$PATH 里的空项目（empty component）表示当前目录（current directory）。空项目位于路径值中间时，可以用两个连续的冒号来表示。如果将冒号直接置于最前端或尾端，可以分别表示查找时最先查找或最后查找当前目录：

```
PATH=:/bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin      先找当前目录
PATH=/bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin:      最后找当前目录
PATH=/bin:/usr/bin:/usr/X11R6/bin::/usr/local/bin      当前目录居中
```

如果你希望将当前目录纳入查找路径（search path），更好的做法是在 $PATH 中使用点号（dot）；这可以让阅读程序的人更清楚程序在做什么。

测试过程中，我们发现同一个系统有两个版本并未正确支持 $PATH 结尾的空项目，因此空项目在可移植性上有点问题。

注意： 一般来说，你根本就不应该在查找路径中放进当前目录，因为这会有安全上的问题（进一步的信息请参考第15章）。之所以会提到空项目，只是为了让你了解路径查找的运作模式。

2.6 访问 Shell 脚本的参数

所谓的位置参数（positional parameters）指的也就是Shell脚本的命令行参数（command-line arguments）。在 Shell 函数里，它们同时也可以是函数的参数。各参数都由整数来命名。基于历史的原因，当它超过 9，就应该用大括号把数字框起来：

```
echo first arg is $1
echo tenth arg is ${10}
```

此外，通过特殊变量，我们还可以取得参数的总数，以及一次取得所有参数。相关细节参见 6.1.2.2 节。

假设你想知道某个用户正使用的终端是什么，你当然可以直接使用 who 命令，然后在输出中自己慢慢找。这么做很麻烦又容易出错 —— 特别是当系统的用户很多的时候。你想做的只不过是在 who 的输出中找到那位用户，这个时候你可以用 grep 命令来进行查找操作，它会列出与第一个参数（所指定的模式）匹配的每一行。假设你要找的是用户 betsy：

```
$ who | grep betsy                          betsy 在哪?
betsy      pts/3        Dec 27 11:07    (flags-r-us.example.com)
```

知道如何寻找特定的用户后，我们可以将命令放进脚本里，这段脚本的第一个参数就是我们要找的用户名称：

```
$ cat > finduser                          建立新文件
#! /bin/sh

# finduser --- 察看第一个参数所指定的用户是否登录

who | grep $1
^D                                        以 End-of-file 结尾

$ chmod +x finduser                       设置执行权限

$ ./finduser betsy                        测试：寻找 betsy
betsy      pts/3        Dec 27 11:07     (flags-r-us.example.com)

$ ./finduser benjamin                     再找找好友 Ben
benjamin   dtlocal      Dec 27 17:55     (kites.example.com)

$ mv finduser $HOME/bin                   将这个文件存进自己的 bin 目录
```

以 `# finduser...` 开头的这一行是一个注释（comment）。Shell 会忽略由 # 开头的每一行。（相信你也已经发现：当 Shell 读取脚本时，前面所提及的 `#!` 行也同样扮演注释的角色。）为你的程序加上注释绝对不会错。这样可以帮助其他人或是自己在一年以后还能够了解你在做什么以及为什么要这么做。等到我们觉得程序能够运行无误时，就可以把它移到个人的 `bin` 目录。

这个程序还没有达到完美。要是我们没给任何参数，会发生什么事？

```
$ finduser
Usage: grep [OPTION]... PATTERN [FILE]...
Try 'grep --help' for more information.
```

我们将在 6.2.4 节看到，如何测试命令行参数数目，以及在参数数目不符时，如何采取适当的操作。

2.7 简单的执行跟踪

程序是人写的，难免会出错。想知道你的程序正在做什么，有个好方法，就是把执行跟踪（execution tracing）的功能打开。这会使得 Shell 显示每个被执行到的命令，并在前面加上“+”：一个加号后面跟着一个空格。（你可以通过给 Shell 变量 `PS4` 赋一个新值以改变打印方式。）

例如：

```
$ sh -x nusers                打开执行跟踪功能
+ who                         被跟踪的命令
+ wc -l
      7                       实际的输出
```

你可以在脚本里，用 `set -x` 命令将执行跟踪的功能打开，然后再用 `set +x` 命令关闭它。这个功能对复杂的脚本比较有用，不过这里只用简单的程序来做说明：

```
$ cat > trace1.sh                    建立脚本
#! /bin/sh

set -x                               打开跟踪功能
echo 1st echo                        做些事

set +x                               关闭跟踪功能
echo 2nd echo                        再做些事
^D                                   以 end-of-file 结尾

$ chmod +x trace1.sh                 设置执行权限

$ ./trace1.sh                        执行
+ echo 1st echo                      被跟踪的第一行
1st echo                             命令的输出
+ set +x                             被跟踪的下一行
2nd echo                             下一个命令的输出
```

执行时，`set -x` 不会被跟踪，因为跟踪功能是在这条命令执行后才打开的。同理，`set +x` 会被跟踪，因为跟踪功能是在这条命令执行后才关闭的。最后的 `echo` 命令不会被跟踪，因为此时跟踪功能已经关闭。

2.8 国际化与本地化

编写软件给全世界的人使用，是一项艰难的挑战。整个工作通常可以分成两个部分：国际化（internationalization，缩写为 i18n，因为这个单字在头尾之间包含了 18 个字母），以及本地化（localization，缩写为 l10n，理由同前）。

当国际化作为设计软件的过程时，软件无须再修改或重新编译程序代码，就可以给特定的用户群使用。至少这表示，你必须将“所要显示的任何信息”包含在特定的程序库调用里，执行期间由此“程序库调用”负责在消息目录（message catalog）中找到适当的译文。一般来说，消息的译文就放在软件附带的文本文件中，再通过 `gencat` 或 `msgfmt` 编译成紧凑的二进制文件，以利快速查询。编译后的信息文件会被安装到特定的系统目录树中，例如 GNU 的 `/usr/share/locale` 与 `/usr/local/share/locale`，或商用 UNIX 系统的 `/usr/lib/nls` 或 `/usr/lib/locale`。详情可见 setlocale(3)、catgets(3C) 与 gettext(3C) 等手册页面（manual pages）。

当本地化作为设计软件的过程时，目的是让特定的用户群得以使用软件。在本地化的过程可能需要翻译软件文件和软件所输出的所有文字，可能还必须修改程序输出中的货币、日期、数字、时间、单位换算等格式。文字所使用的字符集（character set）可能也得

变动（除非使用通用的Unicode字符集），并且使用不同的字体。对某些语言来说，书写方向（writing direction）也可能需要变动。

UNIX的世界中，ISO程序语言标准与POSIX对此类问题的处理都提供了有限度的支持，不过要做的事还很多，而且各种UNIX版本之间差异极大。对用户而言，用来控制让哪种语言或文化环境生效的功能就叫做 *locale*，你可以通过如表 2-3 所示的一个或多个环境变量（environment variable）来设置它。

表 2-3：各种 Locale 环境变量

名称	说明
LANG	未设置任何 LC_xxx 变量时所使用的默认值
LC_ALL	用来覆盖掉所有其他 LC_xxx 变量的值
LC_COLLATE	使用所指定地区的排序规则
LC_CTYPE	使用所指定地区的字符集（字母、数字、标点符号等）
LC_MESSAGES	使用所指定地区的响应与信息；仅 POSIX 适用
LC_MONETARY	使用所指定地区的货币格式
LC_NUMERIC	使用所指定地区的数字格式
LC_TIME	使用所指定地区的日期与时间格式

一般来说，你可以用 LC_ALL 来强制设置单一 locale；而 LANG 则是用来设置 locale 的默认值。大多数时候，应避免为任何的 LC_xxx 变量赋值。举例来说，当你使用 sort 命令时，可能会出现要你正确设置 LC_COLLATE 的信息，因为这个设置可能会跟 LC_CTYPE 的设置相冲突，也可能在 LC_ALL 已设置的情况下完全被忽略。

ISO C 与C++ 标准只定义了 C 这个标准的 locale 名称：用来选择传统的面向 ASCII 的行为模式。POSIX 标准则另外定义了 POSIX 这个 locale 名称，其功能等同于 C。

除 C 与 POSIX 外，locale 名称并未标准化。不过，有很多厂商采用类似但不一致的名称。locale 名称带有语言和地域的意义，有时甚至会加上一个内码集（codeset）与一个修饰符（modifier）。一般来说，它会被表示成 ISO 639 语言代码（language code，注 7）的两个小写字母、一个下划线符号与 ISO 3166-1 国家代码（country code，注 8）的两个大写字母，最后可能还会加上一个点号、字符集编码、@ 符号与修饰词（modifier word）。语文名称有时也会用上。你可以像下面这样列出系统认得哪些 locale 名称：

注 7： 见 *http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt*。

注 8： 见 *http://userpage.chemie.fu-berlin.de/diverse/doc/ISO_3166.html*。

```
$ locale -a                                    列出所有 locale 名称
...
français
fr_BE
fr_BE@euro
fr_BE.iso88591
fr_BE.iso885915@euro
fr_BE.utf8
fr_BE.utf8@euro
fr_CA
fr_CA.iso88591
fr_CA.utf8
...
french
...
```

查询特定locale变量相关细节的方法如下：为执行环境指定locale（放在命令前面）并以-ck选项与一个LC_xxx变量来执行locale命令。下面的例子是在Sun Solaris系统下，以Danish（丹麦文）locale来查询日期时间格式所得到结果：

```
$ LC_ALL=da locale -ck LC_TIME          取得 Danish 的日期时间格式
 LC_TIME
d_t_fmt="%a %d %b %Y %T %Z"
d_fmt="%d-%m-%y"
t_fmt="%T"
t_fmt_ampm="%I:%M:%S %p"
am_pm="AM";"PM"
day="søn dag";"mandag";"tirsdag";"onsdag";"torsdag";"fredag";"lørdag"
abday="søn ";"man";"tir";"ons";"tor";"fre";"lør"
mon="januar";"februar";"marts";"april";"maj";"juni";"juli";"august"; \
    "september";"oktober";"november";"december"
abmon="jan";"feb";"mar";"apr";"maj";"jun";"jul";"aug";"sep";"okt"; \
      "nov";"dec"
era=""
era_d_fmt=""
era_d_t_fmt=""
era_t_fmt=""
alt_digits=""
```

能够使用的locale相当多。一份调查了约20种UNIX版本的报告发现，BSD与Mac OS X系统完全不支持locale（没有locale命令可用），甚至在某些系统上也只支持5种，不过新近发布的GNU/Linux版本则几乎可以支持500种。locale的支持在安装时或许可以由系统管理者自行决定，所以即便是相同的操作系统，安装在两个类似的机器上，对locale的支持可能有所不同。我们发现，在某些系统上，要提供locale的支持，可能需要用到约300 MB（注9）的文件系统。

注9： MB = megabyte，约1百万字节，一个字节传统上有8位，不过更大或更小的尺寸都有人用过。通常M意即2的20次方，也就是1 048 576。

有些GNU包已完成国际化，并在本地化支持上加入了许多locale。例如，以Italian（意大利文）locale来说，GNU的ls命令已提供如下的辅助说明：

```
$ LC_ALL=it_IT ls --help                取得GNU ls的Italian辅助说明
Uso: ls [OPZIONE]... [FILE]...
Elenca informazioni sui FILE (predefinito: la directory corrente).
Ordina alfabeticamente le voci se non è usato uno di -cftuSUX oppure --sort.
""
Mandatory arguments to long options are mandatory for short options too.
  -a, --all                  non nasconde le voci che iniziano con .
  -A, --almost-all           non elenca le voci implicite . e ..
      --author               stampa l'autore di ogni file
  -b, --escape               stampa escape ottali per i caratteri non grafici
      --block-size=DIMENS    usa blocchi lunghi DIMENS byte
...
```

注意，没有译文的地方（输出结果的第5行）会回到原本的语言：英文。程序名称及选项名称没有翻译，因为这么做会破坏软件的可移植性。

目前大多数系统均已对国际化与本地化提供些许支持，让Shell程序员得以处理这方面的问题。我们所写的Shell脚本常受到locale的影响，尤其是排序规则（collation order），以及正则表达式（regular expression）的“方括号表示式”（bracket-expression）里的字符范围。不过，当我们在3.2.1节讨论到字符集（character class）、排序符号（collating symbol）与等价字符集（equivalence class）的时候，你会发现，在大多数UNIX系统下，很难从locale文件与工具来判定“字符集与等价字符集”实际上包含了哪些字符，以及有哪些排序符号可用。这也反映出，在目前的系统上，locale的支持仍未成熟。

GNU *gettext*包（注10）或许可用来支持Shell脚本的国际化与本地化。这个高级主题不在本书的探讨范围，不过相关细节可以在gettext.info在线手册中的“Preparing Shell Scripts for Internationalization”一节找到。

支持locale的系统很多，但缺乏标准的locale名称，因此locale对Shell脚本的可移植性帮助不大，最多只是将LC_ALL设置为C，强制采用传统的locale。在本书中，当遇到locale的设置可能会产生非预期结果时，我们就会这么做。

2.9 小结

该选编译型语言还是脚本编程语言，通常视应用程序的需求而定。脚本编程语言多半用

注10： 见*ftp://ftp.gnu.org/gnu/gettext/*。megabyte的简易算法就是把它想成大概一本书的字数（300页×60行/页×60字符/行=1 080 000字符）。

于比编译型语言高级的情况，当你对性能的要求不高，希望尽快开发出程序并以较高级的方式工作时，也就是使用脚本编程语言的好时机。

Shell是UNIX系统中最重要、也是广为使用的脚本语言。因为它的无所不在，而且遵循POSIX标准，这使得写出来的Shell程序多半能够在各厂商的系统下运行。由于Shell函数是一个高级的功能，所有Shell程序其实相当实用，用户只要花一点力气就能做很多事情。

所有的Shell脚本都应该以#!为第一行；这一机制可让你的脚本更有灵活性，你可以选择使用Shell或其他语言来编写脚本。

Shell是一个完整的程序语言。目前，我们已经说明过基本的命令、选项、参数与变量，以及`echo`与`printf`的基本输出。我们也大致介绍了基本的I/O重定向符：<、>、>>以及|。

Shell会在`$PATH`变量所列举的各个目录中寻找命令。`$PATH`常会包含个人的`bin`目录（用来存储你个人的程序与脚本），你可以在`.profile`文件中将该目录列入到`PATH`里。

我们还看过了取得命令行参数的基本方式，以及简易的执行跟踪。

本章最后讨论的是国际化与本地化。在世界各地的人们对运算需求越来越大的时候，该主题在计算机系统上也日益重要了。对Shell脚本而言，尽管这方面的支持仍然有限，不过Shell程序员还是应该了解locale对他们的程序代码所造成的影响。

第3章

查找与替换

我们在1.2节里曾提及UNIX程序员偏好处理文本的行与列。文本型数据比二进制数据更具灵活性，且UNIX系统也提供许多工具，让用户可以轻松地剪贴文本。

在本章中，我们要讨论的是编写Shell脚本时经常用到的两个基本操作：文本查找（searching）——寻找含有特定文本的行，文本替换（substitution）——更换找到的文本。

虽然你可以使用简单的固定文本字符串完成很多工作，但是正则表达式（regular expression）能提供功能更强大的标记法，以单个表达式匹配各种实际的文本段。本章会介绍两种由不同的UNIX程序所提供的正则表达式风格，然后再进一步介绍提取文本与重新编排文本的几个重要工具。

3.1 查找文本

以`grep`程序查找文本（以UNIX的专业术语来说，是匹配文本（matching text））是相当方便的。在POSIX系统上，`grep`可以在两种正则表达式风格中选择一种，或是执行简单的字符串匹配。

传统上，有三种程序，可以用来查找整个文本文件：

`grep`
: 最早的文本匹配程序。使用POSIX定义的基本正则表达式（Basic Regular Expression，BRE），本章稍后会提到这部分。

`egrep`
: 扩展式`grep`（Extended `grep`）。这个程序使用扩展正则表达式（Extended Regular Expression，ERE）——这是一套功能更强大的正则表达式，使用它的代价就是会

耗掉更多的运算资源。在早期出现的PDP-11的机器上，这点事关重大，不过以现在的系统而言，在性能影响上几乎没有太大的差别。

`fgrep`

快速`grep`（Fast `grep`）。这个版本匹配固定字符串而非正则表达式，它使用优化的算法，能更有效地匹配固定字符串。最初的版本，也是唯一可以并行（in parallel）地匹配多个字符串的版本；也就是说，`grep`与`egrep`只能匹配单个正则表达式；而`fgrep`使用不同的算法，却能匹配多个字符串，有效地测试每个输入行里，是否有匹配的查找字符串。

1992 POSIX标准将这三个改版整合成一个`grep`程序，它的行为是通过不同的选项加以控制。POSIX版本可以匹配多个模式 —— 不管是BRE还是ERE。`fgrep`与`egrep`两者还是可用，只是标记为不推荐使用（deprecated），即它们有可能在往后的标准里删除。果然，在2001 POSIX标准里，就只纳入合并后的`grep`命令。不过实际上，`egrep`与`fgrep`在所有UNIX与类UNIX的系统上都还是可用的。

3.1.1 简单的grep

`grep`最简单的用法就是使用固定字符串：

```
$ who                                                  有谁登录了
tolstoy  tty1         Feb 26 10:53
tolstoy  pts/0        Feb 29 10:59
tolstoy  pts/1        Feb 29 10:59
tolstoy  pts/2        Feb 29 11:00
tolstoy  pts/3        Feb 29 11:00
tolstoy  pts/4        Feb 29 11:00
austen   pts/5        Feb 29 15:39 (mansfield-park.example.com)
austen   pts/6        Feb 29 15:39 (mansfield-park.example.com)
$ who | grep -F austen                          austen登录于何处
austen   pts/5        Feb 29 15:39 (mansfield-park.example.com)
austen   pts/6        Feb 29 15:39 (mansfield-park.example.com)
```

范例中使用`-F`选项，以查找固定字符串**`austen`**。事实上，只要匹配的模式里未含有正则表达式的meta字符（metacharacter），则`grep`默认行为模式就等同于使用了`-F`：

```
$ who | grep austen                                    不具 -F，但结果一样
austen   pts/5        Feb 29 15:39 (mansfield-park.example.com)
austen   pts/6        Feb 29 15:39 (mansfield-park.example.com)
```

3.2 正则表达式

本节提供有关正则表达式构造与匹配方式的概述。特别会提及POSIX BRE与ERE构造，因为它们想要将大部分UNIX工具里的两种正则表达式基本风格（flavors）加以正式化。

grep

语法

grep [*options ...*] *pattern-spec* [*files ...*]

用途

显示匹配一个或多个模式的文本行。时常会作为管道（pipeline）的第一步，以便对匹配的数据作进一步处理。

主要选项

-E

使用扩展正则表达式进行匹配。grep -E可取代传统的egrep。

-F

使用固定字符串进行匹配。grep -F可取代传统的fgrep命令。

-e *pat-list*

通常，第一个非选项的参数会指定要匹配的模式。你也可以提供多个模式，只要将它们放在引号里并以换行字符分隔它们。模式以减号开头时，grep会混淆，而将它视为选项。这就是-e选项派上用场的时候，它可以指定其参数为模式——即使它以减号开头。

-f *pat-file*

从*pat-file*文件读取模式作匹配。

-i

模式匹配时忽略字母大小写差异。

-l

列出匹配模式的文件名称，而不是打印匹配的行。

-q

静默地。如果模式匹配匹配，则grep会成功地离开，而不将匹配的行写入标准输出；否则即是不成功。（我们尚未讨论成功/不成功；可参考6.2节）。

-s

不显示错误信息。通常与-q并用。

-v

显示不匹配模式的行。

行为模式

读取命令行上指名的每个文件。发现匹配查找模式的行时，将它显示来。当指明多个文件时，grep会在每一行前面加上文件名与一个冒号。默认使用BRE。

警告

你可以使用多个-e与-f选项，建立要查找的模式列表。

我们期望你在阅读这本书前，已经接触过正则表达式与文本匹配，并已有些了解。如果是这样，下面的段落将澄清如何使用正则表达式完成具有可移植性的Shell脚本。

若你完全没接触过正则表达式，那么这里提到的东西对你来说可能太简略了，你应该先去看看介绍性的资料，例如《Learning the UNIX Operating System》（O'Reilly）或是《sed & awk》（O'Reilly）。因为正则表达式是UNIX工具使用和构建模型上的基础，花些时间学习如何使用它们并且好好利用它们，你会不断地从各个层面得到充分的回报。

如果你使用正则表达式处理文本已有多年经验，可能会觉得这里所介绍的内容略嫌粗略。在这种情况下，我们会建议你浏览了第一部分POSIX BRE与ERE的表格式概括之后，就直接跳到下一节，然后找一些比较深入的资料来阅读，例如《Mastering Regular Expressions》（O'Reilly）。

3.2.1 什么是正则表达式

正则表达式是一种表示方式，让你可以查找匹配特定准则的文本，例如，“以字母a开头”。此表示法让你可以写一个表达式，选定或匹配多个数据字符串。

除了传统的UNIX正则表达式表示法之外，POSIX正则表达式还可以做到：

- 编写正则表达式，它表示特定于locale的字符序列顺序和等价字符。
- 编写正则表达式，而不必关心系统底层的字符集是什么。

很多的UNIX工具程序沿用某一种正则表达式形式来强化本身的功能。这里列举一部分例子：

- 用来寻找匹配文本行的`grep`工具族：`grep`与`egrep`，以及非标准但很好用的`agrep`工具（注1）。
- 用来改变输入流的`sed`流编辑器（stream editor），本章稍后将会介绍。
- 字符串处理程序语言，例如`awk`、Icon、Perl、Python、Ruby、Tcl等。
- 文件查看程序（有时称为分页程序，pagers），例如`more`、`page`，与`pg`，都常出现在商用UNIX系统上，另外还有广受欢迎的`less`分页程序（注2）。

注1：1992年原始的UNIX版本是在*ftp://ftp.cs.arizona.edu/agrep/agrep-2.04.tar.Z*。Windows版本则在*http://www.tgries.de/agrep/337/agrep337.zip*。`agrep`不同于我们在本书中介绍的大部分可自由下载的软件，它并不能随意地用于任何目的；你可以参考程序所附的许可文件。

注2：与`more`对应的双关语。见*ftp://ftp.gnu.org/gnu/less/*。

- 文本编辑器，例如历史悠久的ed行编辑器、标准的vi屏幕编辑器，还有一些插件（add-on）编辑器，例如emacs、jed、jove、vile、vim等。

正因为正则表达式对于UNIX的使用是这么的重要，所以花些时间把它们弄熟绝对不会错，越早开始就能掌握得越好。

从根本上来看，正则表达式是由两个基本组成部分所建立：一般字符与特殊字符。一般字符指的是任何没有特殊意义的字符，正如下表中所定义的。在某些情况下，特殊字符也可以视为一般字符。特殊字符常称为元字符（metacharacter），本章接下来的部分都会以 meta 字符表示。表 3-1 为 POSIX BRE 与 ERE 的 meta 字符列表。

表 3-1：POSIX BRE 与 ERE 的 meta 字符

字符	BRE/ERE	模式含义
\	两者都可	通常用以关闭后续字符的特殊意义。有时则是相反地打开后续字符的特殊意义，例如 \(...\)与\{...\}。
.	两者都可	匹配任何单个的字符，但 NUL 除外。独立程序也可以不允许匹配换行字符。
*	两者都可	匹配在它之前的任何数目（或没有）的单个字符。以ERE而言，此前置字符可以是正则表达式，例如：因为.（点号）表示任一字符，所以 .* 代表“匹配任一字符的任意长度”。以 BRE 来说，* 若置于正则表达式的第一个字符，不具任何特殊意义。
^	两者都可	匹配紧接着的正则表达式，在行或字符串的起始处。BRE：仅在正则表达式的开头处具此特殊含义，ERE：置于任何位置都具特殊含义。
$	两者都可	匹配前面的正则表达式，在字符串或行结尾处。BRE：仅在正则表达式结尾处具特殊含义。ERE：置于任何位置都具特殊含义。
[...]	两者都可	方括号表达式（bracket expression），匹配方括号内的任一字符。连字符（-）指的是连续字符的范围（注意：范围会因 locale 而有所不同，因此不具可移植性）。^ 符号置于方括号里第一个字符则有反向含义：指的是匹配不在列表内（方括号内）的任何字符。作为首字符的一个连字符或是结束方括号（]），则被视为列表的一部分。所有其他的 meta 字符也为列表的一部分（也就是：根据其字面上的意义）。方括号表达式里可能会含有排序符号（collating symbol）、等价字符集（equivalence class），以及字符集（character class）（文后将有介绍）。

表 3-1：POSIX BRE 与 ERE 的 meta 字符（续）

字符	BRE/ERE	模式含义
\{*n*,*m*\}	BRE	区间表达式（interval expression），匹配在它前面的单个字符重现（occurrences）的次数区间。\{*n*\}指的是重现 *n* 次；\{*n*,\}则为至少重现（occurrences）*n* 次，而\{*n*,*m*\}为重现 *n* 至 *m* 次。*n* 与 *m* 的值必须介于 0 至 RE_DUP_MAX（含）之间，后者最小值为 255。
\(\)	BRE	将\(与\)间的模式存储在特殊的“保留空间（holding space）”。最多可以将 9 个独立的子模式（subpattern）存储在单个模式中。匹配于子模式的文本，可通过转义序列（escape sequences）\1 至 \9，被重复使用在相同模式里。例如**\(ab\).*\1**，指的是匹配于 ab 组合的两次重现，中间可存在任何数目的字符。
n	BRE	重复在\(与\)方括号内第 *n* 个子模式至此点的模式。*n* 为 1 至 9 的数字，1 为由左开始。
{*n*,*m*}	ERE	与先前提及 BRE 的\{*n*,*m*\}一样，只不过方括号前没有反斜杠。
+	ERE	匹配前面正则表达式的一个或多个实例。
?	ERE	匹配前面正则表达式的零个或一个实例。
|	ERE	匹配于|符号前或后的正则表达式。
( )	ERE	匹配于方括号括起来的正则表达式群。

表 3-2 列举了一些简单的范例。

表 3-2：简单的正则表达式匹配范例

表达式	匹配
tolstoy	位于一行上任何位置的 7 个字母：tolstoy
^tolstoy	7 个字母 tolstoy，出现在一行的开头
tolstoy$	7 个字母 tolstoy，出现在一行的结尾
^tolstoy$	正好包括 tolstoy 这 7 个字母的一行，没有其他的任何字符
[Tt]olstoy	在一行上的任意位居中，含有 Tolstoy 或是 tolstoy
tol.toy	在一行上的任意位居中，含有 tol 这 3 个字母，加上任何一个字符，再接着 toy 这 3 个字母
tol.*toy	在一行上的任意位居中，含有 tol 这 3 个字母，加上任意的 0 或多个字符，再继续 toy 这 3 个字母（例如，toltoy、tolstoy、tolWHOtoy 等）

3.2.1.1 POSIX 方括号表达式

为配合非英语的环境，POSIX 标准强化其字符集范围的能力（例如，**[a-z]**），以匹配

非英文字母字符。举例来说，法文的è是字母字符，但以传统字符集**[a-z]**匹配并无该字符。此外，该标准也提供字符序列功能，可用以在匹配及排序字符串数据时，将序列里的字符视为一个独立单位（例如，将 locale 中 ch 这两个字符视为一个单位，在匹配与排序时也应这样看待）。越来越广为使用的Unicode字符集标准，进一步地增加了在简单范围内使用它的复杂度，使得它们对于现代应用程序而言更加不适用。

POSIX也在一般术语上作了些变动，我们早先看到的范围表达式在UNIX里通常称为字符集(character class)，在POSIX的标准下，现在叫做方括号表达式(bracket expression)。在方括号表达式里，除了字面上的字符（例如 z、;等等）之外，另有额外的组成部分，包括：

字符集（Character class）
: 以**[:**与**:]**将关键字组合括起来的 POSIX 字符集。关键字描述各种不同的字符集，例如英文字母字符、控制字符等，见表 3-3。

排序符号（Collating symbol）
: 排序符号指的是将多字符序列视为一个单位。它使用**[.**与**.]**将字符组合括起来。排序符号在系统所使用的特定 locale 上各有其定义。

等价字符集（Equivalence class）
: 等价字符集列出的是应视为等值的一组字符，例如 e 与 è。它由取自于locale的名字元素组成，以[= 与 =]括住。

这三种构造都必须使用方括号表达式。例如**[[:alpha:]!]**匹配任一英文字母字符或惊叹号 (!)；而**[[.ch.]]**则匹配于 ch（排序元素），但字母 c 或 h 则不是。在法文 French 的 locale 里，**[[=e=]]**可能匹配于 e、è、ë、ê 或 é。接下来会有字符集、排序符号，以及等价字符集的详细说明。

表 3-3 描述 POSIX 字符集。

表 3-3：POSIX 字符集

类别	匹配字符	类别	匹配字符
[:alnum:]	数字字符	[:lower:]	小写字母字符
[:alpha:]	字母字符	[:print:]	可显示的字符
[:blank:]	空格（space）与定位（tab）字符	[:punct:]	标点符号字符
[:cntrl:]	控制字符	[:space:]	空白（whitespace）字符
[:digit:]	数字字符	[:upper:]	大写字母字符
[:graph:]	非空格（nonspace）字符	[:xdigit:]	十六进制数字

BRE 与 ERE 共享一些常见的特性，不过仍有些重要差异。我们会从 BRE 的说明开始，再介绍 ERE 附加的 meta 字符，最后针对使用相同（或类似）meta 字符但拥有不同语义（或含义）的情况进行说明。

3.2.2 基本正则表达式

BRE是由多个组成部分所构建，一开始提供数种匹配单个字符的方式，而后又结合额外的 meta 字符，进行多字符匹配。

3.2.2.1 匹配单个字符

最先开始是匹配单个字符。可采用集中方式做到：以一般字符、以转义的meta字符、以 .（点号）meta 字符，或是用方括号表达式：

- 一般字符指的是未列于表 3-1 的字符，包括所有文字和数字字符、绝大多数的空白（whitespace）字符以及标点符号字符。因此，正则表达式 **a**，匹配于字符 a。我们可以说，一般字符所表示的就是它们自己，且这种用法应是最直接且易于理解的。所以，**shell** 匹配于 shell；**WoRd** 匹配于 WoRd，但不匹配于 word。
- 若 meta 字符不能表示它们自己，那当我们需要让 meta 字符表示它们自己的时候，该怎么办？答案是转义它。在前面放一个反斜杠来做到这一点。因此，***** 匹配于字面上的 *，**** 匹配于字面上的反斜杠，还有 **\[** 匹配于左方括号（若将反斜杠放在一般字符前，则 POSIX 标准保留此行为模式为未定义状态。不过通常这种情况下反斜杠会被忽略，只是很少人会这么做）。
- .（点号）字符意即“任一字符”。因此，**a.c** 匹配于 abc、aac 以及 aqc。单个点号用以表示自己的情况很少，它多半与其他 meta 字符搭配使用，这一结合允许匹配多个字符，这部分稍后会提及。
- 最后一种匹配单个字符的方式是使用方括号表达式（bracket expression）。最简单的方括号表达式是直接将字符列表放在方括号里，例如，**[aeiouy]** 表示的就是所有小写元音字母。举例来说，**c[aeiouy]t** 匹配于 cat、cot 以及 cut（还有 cet、cit，与 cyt），但不匹配于 cbt。

  在方括号表达式里，^ 放在字首表示是取反（complement）的意思；也就是说，不在方括号列表里的任意字符。所以 **[^aeiouy]** 指的就是小写元音字符以外的任何字符，例如：大写元音字母、所有辅音字母、数字、标点符号等。

将所有要匹配的字母全列出来是一件无聊又麻烦的事。例如 **[0123456789]** 指所有数字，或 **[0123456789abcdefABCDEF]** 表示所有十六进制数字。因此，方括号表达式可以包括字符的范围。像前面提到的两个例子，就可以分别以 **[0-9]** 与 **[0-9a-fA-F]** 表示。

警告： 一开始，范围表示法匹配字符时，是根据它们在机器中字符集内的数值。因此字符集的不同（ASCII v.s EBCDIC），会使得表示法无法百分之百地具有可移植性，但实际上问题不大，因为几乎所有的 UNIX 系统都是使用 ACSII。

以 POSIX 的 locale 来说，某些地方可能会有问题。范围使用的是各个字符在 locale 排序序列里所定义的位置，与机器字符集里的数值不相关。因此，范围表示法仅在程序运行在 locale 设置为“POSIX”之下，才具可移植性。前面所提及的 POSIX 字符集表示法，提供一种可移植方式表示概念，例如“所有数字”，或是“所有字母字符”，因此在方括号表达式内的范围不建议用在新程序里。

在前面的 3.2.1 节里，我们曾简短地介绍 POSIX 的排序符号（collating symbol）、等价字符集（equivalence class）以及字符集（character class）。这些是方括号表达式最后出现的组成部分。接下来我们就要说明它们的构造方式。

在部分非英语系的语言里，为了匹配需要，某些成对的字符必须视为单个字符。像这样的成对字符，当它们与语言里的单个字符比较时，都有其排序的定义方式。例如，在 Czech 与 Spanish 语系下，ch 两个字符会保持连续状态，在匹配时，会视为单个独立单位。

排序（collating）是指给予成组的项目排列顺序的操作。一个 POSIX 的排序元素由当前 locale 中的元素名称组成，并由 [. 与 .] 括起来。以刚才讨论的 ch 来说，locale 可能会用 [.ch.]（我们说“可能”是因为每一个 locale 都有自己定义的排序元素）。假定 [.ch.] 是存在的，那么正则表达式 **[ab[.ch.]de]** 则匹配于字符 a、b、d 或 e，或者是成对的 ch；而单独的 c 或 h 字符则不匹配。

等价字符集（equivalence class）用来让不同字符在匹配时视为相同字符。等价字符集将类别（class）名称以 [= 与 =] 括起来。举例来说，在 French 的 locale 下，可能有 [=e=] 这样的等价字符集，在此类别存在的情况下，正则表达式 **[a[=e=]iouy]** 就等同于所有小写英文字母元音，以及字母 è、é 等。

最后一个特殊组成部分：字符集，它表示字符的类别，例如数字、小写与大写字母、标点符号、空白（whitespace）等。这些类别名称定义于 [: 与 :] 之间。完整列表如表 3-3 所示。前 POSIX（pre-POSIX）范围表达式对于十进制与十六进制数字的表示（应该）是具有可移植性的，可使用字符集：**[[:digit:]]** 与 **[[:xdigit:]]** 达成。

注意： 排序元素、等价字符集以及字符集，都仅在方括号表达式的方括号内认可，也就是说，像 **[:alpha:]** 这样的正则表达式，匹配字符为 a、l、p、h 以及 :，表示所有英文字母的正确写法应为 **[[:alpha:]]**。

在方括号表达式中，所有其他的 meta 字符都会失去其特殊含义。所以**[*\.]**匹配于字面上的星号、反斜杠以及句点。要让]进入该集合，可以将它放在列表的最前面：**[]*\.]**，将]增加至此列表中。要让减号字符进入该集合，也请将它放到列表最前端：**[-*\.]**。若你需要右方括号与减号两者进入列表，请将右方括号放到第一个字符、减号放到最后一个字符：**[]*\.-]**。

最后，POSIX 明确陈述：NUL 字符（数值的零）不需要是可匹配的。这个字符在 C 语言里是用来指出字符串结尾，而 POSIX 标准则希望让它是直截了当的，通过正规 C 字符串的使用实现其功能。除此之外，另有其他个别的工具程序不允许使用 .（点号）meta 字符或方括号表达式来进行换行字符匹配。

3.2.2.2 后向引用

BRE 提供一种叫后向引用（backreferences）的机制，指的是“匹配于正则表达式匹配的先前的部分”。使用后向引用的步骤有两个。第一步是将子表达式包围在\(与\)里；单个模式里可包括至多 9 个子表达式，且可为嵌套结构。

下一步是在同一模式之后使用*digit*，*digit*指的是介于 1 至 9 的数字，指的是“匹配于第 *n* 个先前方括号内子表达式匹配成功的字符”。举例如下：

模式	匹配成功
\(ab\)\(cd\)[def]*\2\1	abcdcdab、abcdeeecdab、abcdddeeffcdab、...
\(why\).*\1	一行里重现两个 why
\([[:alpha:]_][[:alnum:]_]*\) = \1;	简易 C/C++ 赋值语句

后向引用在寻找重复字以及匹配引号时特别好用：

```
\(["']\).*\1        匹配以单引号或双引号括起来的字，例如 'foo' 或 "bar"
```

在这种方法下，就无须担心是单引号或是双引号先找到。

3.2.2.3 单个表达式匹配多字符

匹配多字符最简单的方法就是把它们一个接一个（连接）列出来，所以正则表达式**ab**匹配于 ab，**..**（两个点号）匹配于任意两个字符，而**[[:upper:]][[:lower:]]**则匹配于任意一个大写字符，后面接着任意一个小写字符。不过，将这些字符全列出来只有在简短的正则表达式里才好用。

虽然 .（点号）meta 字符与方括号表达式都提供了一次匹配一个字符的很好方式，但正则表达式真正强而有力的功能，其实是在修饰符（modifier）meta 字符的使用上。这类 meta 字符紧接在具有单个字符的正则表达式之后，且它们会改变正则表达式的含义。

最常用的修饰符为星号（*），表示“匹配 0 个或多个前面的单个字符”。因此，**ab*c** 表示的是“匹配 1 个 **a**、**0** 或多个 **b** 字符以及 **a c**”。这个正则表达式匹配的有 **ac**、**abc**、**abbc**、**abbbc** 等。

注意：“匹配 0 或多个”不表示“匹配其他的某一个……”，了解这一点是相当重要的。也就是说，正则表达式 **ab*c** 下，文本 aQc 是不匹配的 —— 即便是 aQc 里拥有 0 个 b 字符。相对的，以文本 ac 来说，b* 在 **ab*c** 里表述的是匹配 a 与 c 之间含有空字符串（null string）—— 意即长度为 0 的字符串（若你先前没遇过字符串长度为 0 的概念，这里可能得花点时间消化。总之，它在必要的时候会派得上用场，这在本章稍后会有所介绍）。

* 修饰符是好用的，但它没有限制，你不能用 * 表示“匹配三个字符，而不是四个字符”，而要使用一个复杂的方括号表达式，表明所需的匹配次数 —— 这也是件很麻烦的事。区间表达式（interval expressions）可以解决这类问题。就像 *，它们一样接在单个字符正则表达式后面，控制该字符连续重复几次即为匹配成功。区间表达式是将一个或两个数字，放在 \{ 与 \} 之间，有 3 种变化，如下：

\{*n*\}　前置正则表达式所得结果重现 *n* 次

\{*n*,\}　前置正则表达式所得的结果重现至少 *n* 次

\{*n*,*m*\}　前置正则表达式所得的结果重现 *n* 至 *m* 次

有了区间表达式，要表达像“重现 5 个 a”或是“重现 10 到 42 个 q”就变得很简单了，这两项分别是：**a\{5\}** 与 **q\{10,42\}**。

n 与 *m* 的值必须介于 0 至 RE_DUP_MAX（含）之间。RE_DUP_MAX 是 POSIX 定义的符号型常数，且可通过 getconf 命令取得。RE_DUP_MAX 的最小值为 255；不过部分系统允许更大值，在我们的 GNU/Linux 系统中，就遇到很大的值：

```
$ getconf RE_DUP_MAX
32767
```

3.2.2.4 文本匹配锚点

再介绍两个 meta 字符就能完成整个 BRE 的介绍了。这两个 meta 字符是脱字符号（^）与货币符号（$），它们叫做锚点（anchor），因为其用途在限制正则表达式匹配时，针对要被匹配字符串的开始或结尾处进行匹配（^ 在此处的用法与方括号表达式里的完全不同）。假定现在有一串要进行匹配的字：abcABCdefDEF，我们以表 3-4 列举匹配的范例。

表 3-4：正则表达式内锚点的范例

模式	是否匹配	匹配文本（粗体）/ 匹配失败的理由
ABC	是	居中的第 4、5 及 6 个字符：abc**ABC**defDEF
^ABC	否	限定匹配字符串的起始处
def	是	居中的第 7、8 及 9 个字符：abcABC**def**DEF
def$	否	限制匹配字符串的结尾处
[[:upper:]]\{3\}	是	居中的第 4、5 及 6 个字符：abc**ABC**defDEF
[[:upper:]]\{3\}$	是	结尾的第 10、11 及 12 个字符：abcDEFdef**DEF**
^[[:alpha:]]\{3\}	是	起始的第 1、2 及 3 个字符：**abc**ABCdefDEF

^ 与 $ 也可同时使用，这种情况是将括起来的正则表达式匹配整个字符串（或行）。有时 **^$** 这样的简易正则表达式也很好用，它可以用来匹配空的（empty）字符串或行列。例如在 grep 加上 -v 选项可以用来显示所有不匹配于模式的行，使用上面的做法，便能过滤掉文件里的空（empty）行。

例如，C 的源代码在经过处理后，变成了 #include 文件与 #define 宏时，这种用法就很有用了，因为这样一来你可以了解 C 编译器实际上看到的是什么（这是一种初级的调试法，但有时你就是要这么做）。扩展文件（expanded file）里头时常包含的空白或空行通常会比原始码更多，因此要排除空行只要：

```
$ cc -E foo.c | grep -v '^$' > foo.out      预先删除空行
```

^ 与 $ 仅在 BRE 的起始与结尾处具有特殊用途。在 BRE 下，ab^cd 里的 ^ 表示的，就是自身（^）；同样地，ef$gh 里的 $ 在这里表示的也就是字面上的货币字符。它也可能与其他正则表达式连用，例如 **\^** 与 **\$**，或是 **[$]**（注 3）。

3.2.2.5 BRE 运算符优先级

在数学表达式里，正则表达式的运算符具有某种已定义的优先级（precedence），指的是某个运算符（优先级较高）将比其他运算符先被处理。表 3-5 提供 BRE 运算符的优先级 —— 由高至低。

表 3-5：BRE 运算符优先级，由高至低

运算符	表示意义
[..] [==] [::]	用于字符排序的方括号符号
metacharacter	转义的 meta 字符

注 3：**[^]** 并非有效的正则表达式。请确认你了解是为什么。

表 3-5：BRE 运算符优先级，由高至低（续）

运算符	表示意义
[ ]	方括号表达式
\(\) *digit*	子表达式与后向引用
* \{ \}	前置单个字符重现的正则表达式
无符号（no symbol）	连续
^ $	锚点（Anchors）

3.2.3 扩展正则表达式

ERE（Extended Regular Expressions）的含义就如同其名字所示：拥有比基本正则表达式更多的功能。BRE 与 ERE 在大多数 meta 字符与功能应用上几乎是完全一致，但 ERE 里有些 meta 字符看起来与 BRE 类似，却具有完全不同的意义。

3.2.3.1 匹配单个字符

在匹配单个字符的情况下，ERE 本质上是与 BRE 是一致的。特别是像一般字符、用以转义 meta 字符的反斜杠，以及方括号表达式，这些行为模式都与先前提及的 BRE 相同。较有名的一个例外出现在 awk 里：其 \ 符号在方括号表达式内表示其他的含义。因此，如需匹配左方括号、连字符、右方括号或是反斜杠，你应该用 `[\[\-\]\\]`。这是使用上的经验法则。

3.2.3.2 后向引用不存在

ERE 里是没有后向引用的（注 4）。圆括号在 ERE 里具特殊含义，但和 BRE 里的使用又有所不同（这点稍后会介绍）。在 ERE 里，\(与 \) 匹配的是字面上的左括号与右括号。

3.2.2.3 匹配单个表达式与多个正则表达式

ERE 在匹配多字符这方面，与 BRE 有很明显的不同。不过，在 * 的处理上和 BRE 是相同的（注 5）。

注 4： 这在 grep 与 egrep 命令下有不同的影响，这并非正则表达式匹配能力的问题，只是 UNIX 的一种处理方式而已。

注 5： 有一个例外是，* 作为 ERE 的第一个字符是“未定义”的，而在 BRE 中它是指“符合字面的 *”。

区间表达式可用于ERE中，但它们是写在花括号里（{}），且不需前置反斜杠字符。因此我们先前的例子“要刚好重现5个a”以及“重现10个至42个q”，写法分别为：**a{5}**与**q{10,42}**。而**\{**与**\}**则可用以匹配字面上的花括号。当在ERE里{找不到匹配的}时，POSIX特意保留其含义为“未定义（undefined）状态”。

ERE另有两个meta字符，可更细腻地处理匹配控制：

?	匹配于0个或一个前置正则表达式
+	匹配于1个或多个前置正则表达式

你可以把?想成是“可选用的”，也就是说，匹配前置正则表达式的文本，要么出现，要么没出现。举例来说，与**ab?c**匹配的有ac与abc，就这两者！（与**ab*c**相较之下，后者匹配于中间有任意个b）。

+字符在概念上与* meta字符类似，不过前置正则表达式要匹配的文本在这里至少得出现一次。因此，**ab+c**匹配于abc、abbc、abbbc，但不匹配于ac。你当然可以把**ab+c**的正则表达式形式换成**abb*c**；无论如何，当前置正则表达式很复杂时，使用+可以少打一点字，当然也减少了打错字的几率！

3.2.3.4 交替

方括号表达式易于表示“匹配于此字符，或其他字符，或…”，但不能指定“匹配于这个序列（sequence），或其他序列（sequence），或…”。要达到后者的目的，你可以使用交替（alternation）运算符，即垂直的一条线，或称为管道字符（|）。你可以简单写好两个字符序列，再以|将其隔开。例如**read|write**匹配于read与write两者、**fast|slow**匹配于fast与slow两者。你还可以使用多个该符号：**sleep|doze|dream|nod off|slumber**匹配于5个表达式。

|字符为ERE运算符里优先级最低的。因此，左边会一路扩展到运算符的左边，一直到一个前置|的字符，或者是到另一个正则表达式的起始。同样地，|的右边也是一路扩展到运算符的右边，一直到后续的|字符，或是到整个正则表达式的结尾。这部分将在下一节探讨。

3.2.3.5 分组

你应该已经发现，在ERE里，我们已提到运算符是被应用到“前置的正则表达式”。这是因为有圆方括号（(...)）提供分组功能，让接下来的运算符可以应用。举例来说，**(why)+**匹配于一个或连续重复的多个why。

在必须用到交替（alternation）时，分组的功能就特别好用了（也是必需的）。它可以让你用以构建复杂并较灵活性的正则表达式。举例来说，**[Tt]he (CPU|computer) is** 指的就是：在 The（或 the）与 is 之间，含有 CPU 或 computer 的句子。要特别注意的一点是，圆括号在这里是 meta 字符，而非要匹配的输入文本。

将重复（repetition）运算符与交替功能结合时，分组功能也是一定用得到的。**read|write+** 指的是正好一个 read，或是一个 write 后面接着任意数个 e 字符（writee、writeee 等），比较有用的模式应该是 **(read|write)+**，它指的是：有一个或重现多个 read，或者一个或重现多个 write。

当然，**(read|write)+** 所指的字符串中间，不允许有空白。**((read|write)[[:space:]]*)+** 的正则表达式看起来虽然比较复杂，不过也比较实际些。乍看之下，这可能会搞不清楚，不过若把这些组成部分分隔开来看，其实就不难理解了。图 3-1 为图解说明。

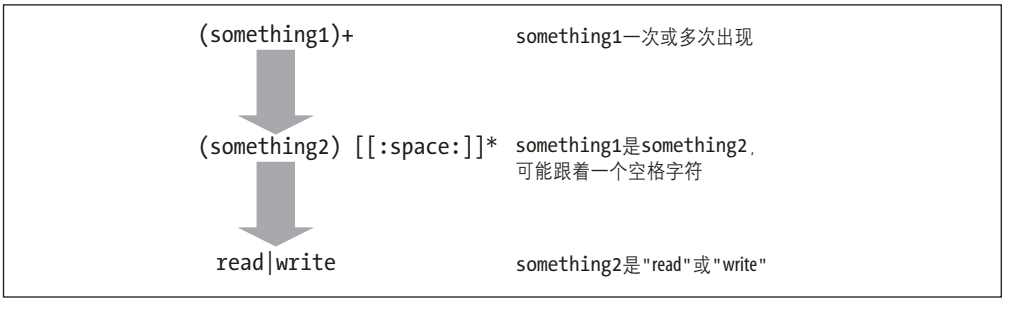

图 3-1：读取一个复杂的正则表达式

结论就是：这个单个正则表达式是用以匹配多个连续出现的 read 或是 write，且中间可能被空白字符隔开。

在 [[:space:]] 之后使用 * 是一种判断调用（judgment call）。使用一个 * 而非 +，此匹配可以取得在行（或字符串）结尾的单词。但也可能可以匹配中间完全没有空白的单词。运用正则表达式时常会需要用到这样的判断调用（judgment call）。该如何构建正则表达式，需要根据输入的数据以及这些数据的用途而定。

最后要说的是：当你将交替（alternation）操作结合 ^ 与 $ 锚点字符使用时，分组就非常好用了。由于 | 为所有运算符中优先级最低的，因此正则表达式 **^abcd|efgh$** 意思是“匹配字符串的起始处是否有 a b c d，或者字符串结尾处是否有 e f g h”，这和 **^(abcd|efgh)$** 不一样，后者表示的是“找一个正是 abcd 或正是 efgh 的字符串”。

3.2.3.6 停驻文本匹配

^ 与 $ 在这里表示的意义与BRE里的相同：将正则表达式停驻在文本字符串（或行）的起始或结尾处。不过有个明显不同的地方就是：在ERE里，^ 与 $ 永远是meta字符。所以，像 **ab^cd** 与 **ef$gh** 这样的正则表达式仍是有效的，只是无法匹配到任何东西，因为 ^ 前置了文本，与 $ 后面的文本，会让它们分别无法匹配到“字符串的开始”与“字符串结尾”。正如其他的meta字符一般，它们在方括号表达式中的确失去了它们特殊的意义。

3.2.3.7 ERE 运算符的优先级

在ERE里运算符的优先级和BRE一样。表 3-6 由高至低列出了ERE运算符的优先级。

表 3-6：ERE 运算符优先级，由高至低

运算符	含义
`[..] [= =] [: :]`	用于字符对应的方括号符号
\metacharacter	转义的meta字符
`[]`	方括号表达式
`()`	分组
`* + ? {}`	重复前置的正则表达式
无符号（no symbol）	连续字符
`^ $`	锚点（Anchors）
`|`	交替（Alternation）

3.2.4 正则表达式的扩展

很多程序提供正则表达式语法扩展。这类扩展大多采取反斜杠加一个字符，以形成新的运算符。类似POSIX BRE里 **\(...\)** 与 **\{...\}** 的反斜杠。

最常见的扩展为 \< 与 \> 运算符，分别匹配“单词（word）”的开头与结尾。单词是由字母、数字及下划线组成的。我们称这类字符为单词组成（word-constituent）。

单词的开头要么出现在行起始处，要么出现在第一个后面紧跟一个非单词组成（nonword-constituent）字符的单词组成（word-constituent）字符。同样的，单词的结尾要么出现在一行的结尾处，要么出现在一个非单词组成字符之前的最后一个单词组成字符。

实际上，单词的匹配其实相当直接易懂。正则表达式 **\<chop** 匹配于 `use chopsticks`，

但 eat a lambchop 则不匹配；同样地，**chop\>** 则匹配于第二个字符串，第一个则不匹配。需要特别注意的一点是：在 **\<chop\>** 的表达式下，两个字符串都不匹配。

虽然 POSIX 标准化的只有 ex 编辑器，但在所有商用 UNIX 系统上，ed、ex 以及 vi 编辑器都支持单词匹配，而且几乎已是标准配备。GNU/Linux 与 BSD 系统上附带的克隆程序（“clone” version）也支持单词匹配，还有 emacs、vim 与 vile 也是。除此之外，通常 grep 与 sed 也会支持，不过最好在系统里再通过手册页（manpage）确认一下。

可处理正则表达式的标准工具的GNU版本，通常还支持许多额外的运算符，如表 3-7 所示。

表 3-7：额外的 GNU 正则表达式运算符

运算符	含义
\w	匹配任何单词组成字符，等同于 **[[:alnum:]_]**
\W	匹配任何非单词组成字符，等同于 **[^[:alnum:]_]**
\<\>	匹配单词的起始与结尾，如前文所述
\b	匹配单词的起始或结尾处所找到的空字符串。这是 \< 与 \> 运算符的结合
	注意：由于 awk 使用 \b 表示后退字符，因此 GNU awk（gawk）使用 \y 表示此功能
\B	匹配两个单词组成字符之间的空字符串
\' \`	分别匹配 emacs 缓冲区的开始与结尾。GNU 程序（还有 emacs）通常将它们视为与 ^ 及 $ 同义

虽然 POSIX 明白表示了 NUL 字符无须是可匹配的，但 GNU 程序则无此限制。若 NUL 字符出现在输入数据里，则它可以通过 .meta 字符或方括号表达式来匹配。

3.2.5 程序与正则表达式

有两种不同的正则表达式风格是经年累月的历史产物。虽然egrep风格的扩展正则表达式在 UNIX 早期开发时就已经存在了，但 Ken Thompson 并不觉得有必要在 ed 编辑器里使用这样全方位的正则表达式（由于 PDP-11 的小型地址空间、扩展正则表达式的复杂度，以及实际应用时大部分的编辑工作使用基本正则表达式已足够了，这样的决定其实相当合理）。

ed 的程序代码后来成了 grep 的基础（grep 为 ed 命令中 g/*re*/p 的缩写，意即全局性匹配 *re*，并将其打印）。ed 的程序代码后来也成为初始构建 sed 的根基。

就在 pre-V7 时期，Al Aho 创造了 egrep，Al Aho 是贝尔实验室的研究人员，他为正则表达式匹配与语言解析的研究奠定了基础。egrep 里的核心匹配程序代码，日后也被 awk 的正则表达式拿来使用。

\< 与 \> 运算符起源于滑铁卢大学的 Rob Pike、Tom Duff、Hugh Redelmeier，以及 David Tilbrook 所修改的 ed 版本（Rob Pike 是这些运算符的发明者之一）。Bill Joy 在 UCB 时，便将这两个运算符纳入 ex 与 vi 编辑器，自那时起，它就广为流传。区间表达式源起于 Programmer's Workbench UNIX（注 6），之后通过 System III 以及此后的 System V，特别将其取出用于商用 UNIX 系统上。表 3-8 列出的是各种 UNIX 程序与其使用的正则表达式。

表 3-8：UNIX 程序及其正则表达式类型

类型	grep	sed	ed	ex/vi	more	egrep	awk	lex
BRE	•	•	•	•	•			
ERE						•	•	•
\< \>	•	•	•	•	•			

lex 是一个很特别的工具，通常是用于语言处理器中的词法分析器的构建。虽然已纳入 POSIX，但我们不会在这里进一步讨论，因为它与 Shell 脚本无关。less 与 pg 虽然不是 POSIX 的一部分，但它们也支持正则表达式。有些系统会有 page 程序，它本质上和 more 是相同的，只是在每个充满屏幕的输出画面之间，会清除屏幕。

正如我们在本章开头所提到的：要（试图）解决多个 grep 的矛盾，POSIX 决定以单个 grep 程序解决。POSIX 的 grep 默认行为模式使用的是 BRE。加上 -E 选项则它使用 ERE，及加上 -F 选项，则它使用 fgrep 的固定字符串匹配算法。因此，真正地遵循 POSIX 的程序应以 grep -E ... 取代 egrep ...。不过，因为所有的 UNIX 系统确实拥有它，且可能已经有许多年了，所以我们继续在自己的脚本中使用它。

最后要注意的一点就是：通常，awk 在其扩展正则表达式里不支持区间表达式。直至 2005 年，各种不同厂商的 awk 版本也并非全面支持区间表达式。为了让程序具有可移植性，若需要在 awk 程序里匹配大方括号，应该以反斜杠转义它，或将它们括在方括号表达式里。

注 6：Programmer's Workbench（PWB）UNIX 是用在 AT&T 里以支持电信交换软件开发的变化版。它也可以用于商业用途。

3.2.6 在文本文件里进行替换

很多Shell脚本的工作都从通过grep或egrep取出所需的文本开始。正则表达式查找的最初结果，往往就成了要拿来作进一步处理的“原始数据（raw data）”。通常，文本替换（text substitution）至少需要做一件事，就是将一些字以另一些字取代，或者是删除匹配行的某个部分。

一般来说，执行文本替换的正确程序应该是sed —— 流编辑器（Stream Editor）。sed的设计就是用来以批处理的方式而不是交互的方式来编辑文件。当你知道要做好几个变更 —— 不管是对一个还是数个文件时，比较简单的方式是将这些变更部分写到一个编辑中的脚本里，再将此脚本应用到所有必须修改的文件。sed存在的目的就在这里（虽然你也可以使用ed或ex编辑脚本，但用它们来处理会比较麻烦，而且用户通常不会记得要存储原先的文件）。

我们发现，在Shell脚本里，sed主要用于一些简单的文本替换，所以我们先从它开始。接下来我们还会提供其他的后台数据，并说明sed的功能，特意不在这里提到太多细节，是因为sed所有的功能全都写在《sed & awk》（O’Reilly）这本书里了，该书已列入参考书目。

GNU sed可从*ftp://ftp.gnu.org/gnu/sed/*获取。这个版本拥有相当多有趣的扩展，且已配备使用手册，还附带软件。GNU的sed使用手册里有一些好玩的例子，还包括与众不同的程序测试工具组。可能最令人感到不可思议的是：UNIX dc任意精确度计算程序（arbitrary-precision calculator）竟是以sed所写成的！

当然绝佳的sed来源就是*http://sed.sourceforge.net/*了。这里有连接到两个sed FAQ文件的链接。第一个是*http://www.dreamwvr.com/sed-info/sed-faq.html*，第二个比较旧的FAQ则是*ftp://rtfm.mit.edu/pub/faqs/editor-faq/sed*。

3.2.7 基本用法

你可能会常在管道（pipeline）中间使用sed，以执行替换操作。做法是使用s命令 —— 要求正则表达式寻找，用替代文本（replacement text）替换匹配的文本，以及可选用的标志：

```
sed 's/:.*//' /etc/passwd |          删除第一个冒号之后的所有东西
 sort -u                             排序列表并删除重复部分
```

sed

语法

```
sed [ -n ] 'editing command' [ file ... ]
sed [ -n ] -e 'editing command' ... [ file ... ]
sed [ -n ] -f script-file ... [ file ... ]
```

用途

为了编辑它的输入流，将结果生成到标准输出，而非以交互式编辑器的方式来编辑文件。虽然sed的命令很多，能做很复杂的工作，但它最常用的还是处理输入流的文本替换，通常是作为管道的一部分。

主要选项

`-e 'editing command'`

将*editing command*使用在输入数据上。当有多个命令需应用时，就必须使用-e了。

`-f script-file`

自*script-file*中读取编辑命令。当有多个命令需要执行时，此选项相当有用。

`-n`

不是每个最后已修改结果行都正常打印，而是显示以p指定（处理过的）的行。

行为模式

读取每个输入文件的每一行，假如没有文件的话，则是标准输入。以每一行来说，sed会执行每一个应用到输入行的*editing command*。结果会写到标准输出（默认状态下，或是显示地使用p命令及-n选项）。若无-e或-f选项，则sed会把第一个参数看作是要使用的*editing command*。

在这里，/字符扮演定界符（delimiter）的角色，从而分隔正则表达式与替代文本(replacement text)。在本例中，替代文本是空的（空字符串null string），实际上会有效地删除匹配的文本。虽然/是最常用的定界符，但任何可显示的字符都能作为定界符。在处理文件名称时，通常都会以标点符号字符作为定界符（例如分号、冒号或逗点）：

```
find /home/tolstoy -type d -print   |    寻找所有目录
 sed 's;/home/tolstoy/;/home/lt/;'  |    修改名称；注意：这里使用分号作为定界符
  sed 's/^/mkdir /'                 |    插入mkdir命令
   sh -x                                 以Shell跟踪模式执行
```

上述脚本是将/home/tolstoy目录结构建立一份副本在/home/lt下（可能是为备份而

做的准备)。(find 命令在第 10 章将会介绍，在本例中它的输出是 /home/tolstoy 底下的目录名称列表：一行一个目录。)这个脚本使用了产生命令(generating commands)的手法，使命令内容成为 Shell 的输入。这是一个功能很强且常见的技巧，但却很少人这么用（注 7)。

3.2.7.1 替换细节

先前已经提过，除斜杠外还可以使用其他任意字符作为定界符；在正则表达式或替代文本里，也能转义定界符，不过这么做可能会让命令变得很难看懂：

```
sed 's/\/home\/tolstoy\//\/home\/lt\//'
```

在前面的3.2.2 节里，我们讲到POSIX 的BRE时，已说明后向引用在正则表达式里的用法。sed 了解后向引用，而且它们还能用于替代文本中，以表示“从这里开始替换成匹配第 *n* 个圆括号里子表达式的文本”：

```
$ echo /home/tolstoy/ | sed 's;\(/home\)/tolstoy/;\1/lt/;'
/home/lt/
```

sed将\1替代为匹配于正则表达式的 **/home**部分。在这里的例子中，所有的字符都表示它自己，不过，任何正则表达式都可括在 \(与 \) 之间，且后向引用最多可以用到 9 个。

有些其他字符在替代文本里也有特殊含义。我们已经提过需要使用反斜杠转义定界符的情况。当然，反斜杠字符本身也可能需要转义。最后要说明的是：& 在替代文本里表示的意思是“从此点开始替代成匹配于正则表达式的整个文本”。举例来说，假设处理 Atlanta Chamber of Commerce 这串文本，想要在广告册中修改所有对该城市的描述：

```
mv atlga.xml atlga.xml.old
sed 's/Atlanta/&, the capital of the South/' < atlga.xml.old > atlga.xml
```

(作为一个跟得上时代的人，我们在所有的地方都尽可能使用 XML，而不是昂贵的专用字处理程序)。这个脚本会存储一份原始广告小册的备份，做这类操作绝对有必要 —— 特别是还在学习如何处理正则表达式与替换（substitutions）的时候，然后再使用 sed 进行变更。

如果要在替代文本里使用 & 字符的字面意义，请使用反斜杠转义它。例如，下面的小脚本便可以转换 DocBook/XML 文件里字面上的反斜杠，将其转换为 DocBook 里对应的 \

```
sed 's/\\/\&bsol;/g'
```

注 7： 这个脚本有小瑕疵，它无法处理目录名称含有空格的情况。这个问题是可以解决的，只是要有点小技巧，这部分我们将在第 10 章介绍。

在 s 命令里以 g 结尾表示的是：全局性（global），意即以“替代文本取代正则表达式中每一个匹配的”。如果没有设置 g，sed 只会取代第一个匹配的。这里来比较看看有没有设置 g 所产生的结果：

```
$ echo Tolstoy reads well. Tolstoy writes well. > example.txt   输入样本
$ sed 's/Tolstoy/Camus/' < example.txt                         没有设置 g
Camus reads well. Tolstoy writes well.
$ sed 's/Tolstoy/Camus/g' < example.txt                        设置了 "g"
Camus reads well. Camus writes well.
```

鲜为人知的是（可以用来吓吓朋友）：你可以在结尾指定数字，指示第 *n* 个匹配出现才要被取代：

```
$ sed 's/Tolstoy/Camus/2' < example.txt    仅替代第二个匹配者
Tolstoy reads well. Camus writes well.
```

到目前为止，我们讲的都是一次替换一个。虽然可以将多个 sed 实体以管道（pipeline）串起来，但是给予 sed 多个命令是比较容易的。在命令行上，这是通过 -e 选项的方式来完成。每一个编辑命令都使用一个 -e 选项：

```
sed -e 's/foo/bar/g' -e 's/chicken/cow/g' myfile.xml > myfile2.xml
```

不过，如果你有很多要编辑的项目，这种形式就很恐怖了。所以有时，将编辑命令全放进一个脚本里，再使用 sed 搭配 -f 选项会更好：

```
$ cat fixup.sed
s/foo/bar/g
s/chicken/cow/g
s/draft animal/horse/g
...
$ sed -f fixup.sed myfile.xml > myfile2.xml
```

你也可以构建一个结合 -e 与 -f 选项的脚本；脚本为连续的所有编辑命令，依次提供所有选项。此外，POSIX 也允许使用分号将同一行里的不同命令隔开：

```
sed 's/foo/bar/g ; s/chicken/cow/g' myfile.xml > myfile2.xml
```

不过，许多商用 sed 版本还不支持此功能，所以如果你很在意可移植性的问题，请避免使用此法。

ed 与其先驱 ex 与 vi 一样，Sed 会记得在脚本里遇到的最后一个正则表达式 —— 不管它在哪。通过使用空的正则表达式，同一个正则表达式可再使用：

```
s/foo/bar/3        更换第三个 foo
s//quux/           现在更换第一个
```

你可以考虑一个 html2xhtml.sed 的简单脚本，它将 HTML 转换为 XHTML。该脚本会将标签转换成小写，然后更改
 标签为自我结束形式
：

```
s/<H1>/<h1>/g                          斜杠为定界符
s/<H2>/<h2>/g
s/<H3>/<h3>/g
s/<H4>/<h4>/g
s/<H5>/<h5>/g
s/<H6>/<h6>/g
s:</H1>:</h1>:g                        冒号为定界符，因数据内容里已有斜杠
s:</H2>:</h2>:g
s:</H3>:</h3>:g
s:</H4>:</h4>:g
s:</H5>:</h5>:g
s:</H6>:</h6>:g
s:</[Hh][Tt][Mm][LL]>:<html>:g
s:</[Hh][Tt][Mm][Ll]>:</html>:g
s:<[Bb][Rr]>:<br/>:g
...
```

像这样的脚本就可以自动执行大量的HTML转XHTML了，XHTML为标准化的、以XML为主的 HTML 版本。

3.2.8 sed 的运作

sed 的工作方式相当直接。命令行上的每个文件名会依次打开与读取。如果没有文件，则使用标准输入，文件名“-”（单个破折号）可用于表示标准输入。

sed读取每个文件，一次读一行，将读取的行放到内存的一个区域 —— 称之为模式空间（pattern space）。这就像程序语言里的变量一样：内存的一个区域在编辑命令的指示下可以修改，所有编辑上的操作都会应用到模式空间的内容。当所有操作完成后，sed会将模式空间的最后内容打印到标准输出，再回到开始处，读取另一个输入行。

这一工作过程如图3-2所示。脚本使用两条命令，将The UNIX System替代为The UNIX Operating System。

3.2.8.1 打印与否

-n 选项修改了 sed 的默认行为。当提供此选项时，sed 将不会在操作完成后打印模式空间的最后内容。反之，若在脚本里使用p，则会明白地将此行显示出来。举例来说，我们可以这样模拟 grep：

```
sed -n '/<HTML>/p' *.html          仅显示 <HTML> 这行
```

虽然这个例子很简单，但这个功能在复杂的脚本里非常好用。如果你使用一个脚本文件，可通过特殊的首行来打开此功能：

```
#n                                 关闭自动打印
/<HTML>/p                          仅打印含 <HTML> 的行
```

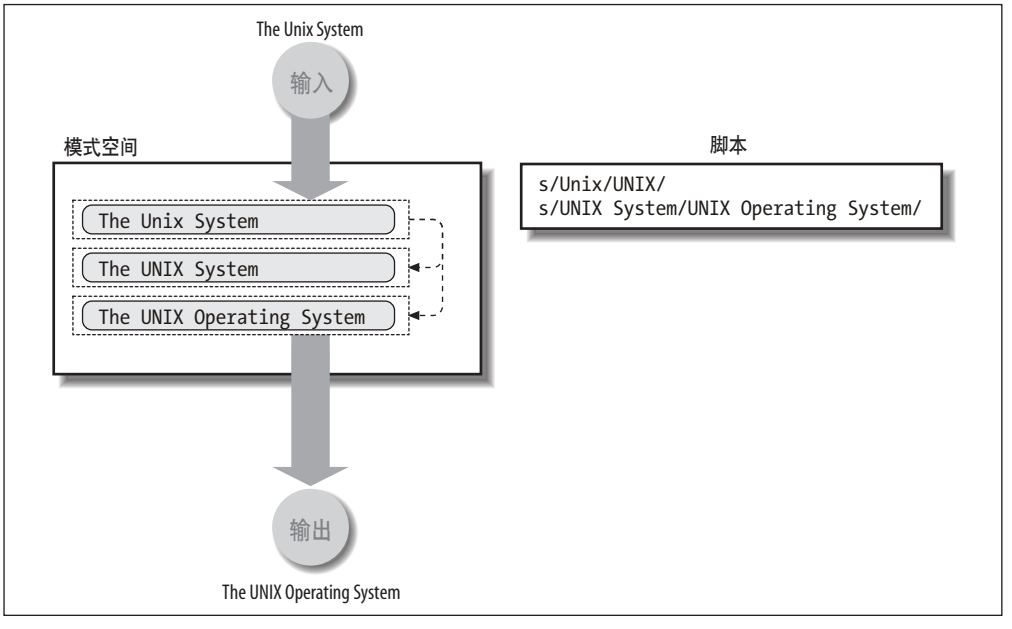

图3-2：在sed脚本中的命令改变了模式空间

在Shell中，与很多其他UNIX脚本式语言一样：#是注释的意思。sed注释必须出现在单独的行里，因为它们是语法型命令，意思是：它们是什么事也不做的命令。虽然POSIX指出，注释可以放在脚本里的任何位置，但很多旧版sed仅允许出现在首行，GNU sed则无此限制。

3.2.9 匹配特定行

如前所述，sed默认地会将每一个编辑命令（editing command）应用到每个输入行。而现在我们要告诉你的是：还可以限制一条命令要应用到哪些行，只要在命令前置一个地址（address）即可。因此，sed命令的完整形式就是：

```
address command
```

以下为不同种类的地址：

正则表达式

将一模式放置到一条命令之前，可限制命令应用于匹配模式的行。可与s命令搭配使用：

```
/oldfunc/ s/$/# XXX: migrate to newfunc/     注释部分源代码
```

s命令里的空模式指的是“使用前一个正则表达式”：

```
/Tolstoy/ s//& and Camus/g          提及两位作者
```

最终行

符号 $（就像在 ed 与 ex 里一样）指“最后一行”。下面的脚本指的是快速打印文件的最后一行：

```
sed -n '$p' "$1"                    引号里为指定显示的数据
```

对 sed 而言，“最后一行”指的是输入数据的最后一行。即便是处理多个文件，sed 也将它们视为一个长的输入流，且 $ 只应用到最后一个文件的最后一行（GNU 的 sed 具有一个选项，可使地址分开地应用到每个文件，见其说明文档）。

行编号

可以使用绝对的行编号作为地址。稍后将有介绍。

范围

可指定行的范围，仅需将地址以逗点隔开：

```
sed -n '10,42p' foo.xml             仅打印 10～42 行
sed '/foo/,/bar/ s/baz/quux/g'      仅替换范围内的行
```

第二个命令为“从含有 **foo** 的行开始，再匹配是否有 **bar** 的行，再将匹配后的结果中，有 **baz** 的全换成 quux”（像 ed、ex 这类的检阅程序，或是 vi 内的冒号命令提示模式下，都认识此语法）。

这种以逗点隔开两个正则表达式的方式称为范围表达式（range expression）。在 sed 里，总是需要使用至少两行才能表达。

否定正则表达式

有时，将命令应用于不匹配于特定模式的每一行，也是很有用的。通过将!加在正则表达式后面就能做到，如下所示：

```
/used/!s/new/used/g                 将没有 used 的每个行里所有的 new 改成 used
```

POSIX 标准指出：空白（whitespace）跟随在 ! 之后的行为是“未定义的（unspecified）”，并建议需提供完整可移植性的应用软件，不要在 ! 之后放置任何空白字符，这明显是由于某些 sed 的古董级版本仍无法识别它。

例 3-1 说明的是使用绝对的行编号作为地址的用法，这里是以 sed 展现的 head 程序简易版。

例 3-1：使用 sed 的 head 命令

```
# head --- 打印前 n 行
#
# 语法：  head N file

count=$1
sed ${count}q "$2"
```

当你引用head 10 foo.xml之后，sed会转换成sed 10q foo.xml。q命令要求sed马上离开；不再读取其他输入，或是执行任何命令。后面的7.6.1节里，我们将介绍如何让这个脚本看起来更像真正的head命令。

迄今为止，我们看到的都是sed以/字符隔开模式以便查找。在这里，我们要告诉你如何在模式内使用不同的定界符：这通过在字符前面加上一个反斜杠实现：

```
$ grep tolstoy /etc/passwd                                    显示原始行
tolstoy:x:2076:10:Leo Tolstoy:/home/tolstoy:/bin/bash
$ sed -n '\:tolstoy: s;;Tolstoy;p' /etc/passwd                改变定界符
Tolstoy:x:2076:10:Leo Tolstoy:/home/tolstoy:/bin/bash
```

本例中，以冒号隔开要查找的模式，而分号则扮演s命令的定界符角色（编辑上的操作其实不重要，我们的重点是介绍如何使用不同的定界符）。

3.2.10 有多少文本会改动

有个问题我们一直还没讨论到：有多少文本会匹配?事实上，这应该包含两个问题。第二个问题是：从哪儿开始匹配？执行简单的文本查找，例如使用grep或egrep时，则这两个问题都不重要，你只要知道是否有一行是匹配的，若有，则看看那一行是什么。至于在这个行里，是从哪儿开始匹配，或者它扩展到哪里，已经不重要了。

但如果你要使用sed执行文本替换，或者要用awk写程序，这两个问题的答案就变得非常重要了（如果你每天都在文本编辑器内工作，这也算是个重要议题，只是本书的重点不在文本编辑器）。

这两个问题的答案是：正则表达式匹配可以匹配整个表达式的输入文本中最长的、最左边的子字符串。除此之外，匹配的空（null）字符串，则被认为是比完全不匹配的还长（因此，就像我们先前所解释的，正则表达式：**ab*c**匹配文本ac，而b*则成功地匹配于a与c之间的null字符串）。再者，POSIX标准指出：“完全一致的匹配，指的是自最左边开始匹配、针对每一个子模式、由左至右，必须匹配到最长的可能字符串”。（子模式指的是在ERE下圆括号里的部分。为此目的，GNU的程序通常也会在BRE里以\(...\)提供此功能）。

如果sed要替代由正则表达式匹配的文本，那么确定该正则表达式匹配的字不会太少或太多就非常重要了。这里有个简单例子：

```
$ echo Tolstoy writes well | sed 's/Tolstoy/Camus/'    使用固定字符串
Camus writes well
```

当然，sed可以使用完整的正则表达式。这里就是要告诉你，了解“从最长的最左边（longest leftmost）”规则有多的重要：

```
$ echo Tolstoy is worldly | sed 's/T.*y/Camus/'      试试正则表达式
Camus                                                结果呢 ?
```

很明显，这里只是要匹配 Tolstoy，但由于匹配会扩展到可能的最长长度的文本量，所以就一直找到 worldly 的 y 了！你需要定义的更精确些：

```
$ echo Tolstoy is worldly | sed 's/T[[:alpha:]]*y/Camus/'
Camus is worldly
```

通常，当开发的脚本是要执行大量文本剪贴和排列组合时，你会希望谨慎地测试每样东西，确认每个步骤都是你要的 —— 尤其是当你还在学习正则表达式里的微妙变化的阶段的时候。

最后，正如我们所见到的，在文本查找时有可能会匹配到 null 字符串。而在执行文本替代时，也允许你插入文本：

```
$ echo abc | sed 's/b*/1/'          替代第一个匹配成功的
1abc
$ echo abc | sed 's/b*/1/g'         替代所有匹配成功的
1a1c1
```

请留意，**b*** 是如何匹配在 abc 的前面与结尾的 null 字符串。

3.2.11 行 v.s. 字符串

了解行（line）与字符串（string）的差异是相当重要的。大部分简易程序都是处理输入数据的行，像 grep 与 egrep，以及 sed 大部分的工作（99%）都是这样。在这些情况下，不会有内嵌的换行字符出现在将要匹配的数据中，^ 与 $ 则分别表示行的开头与结尾。

然而，对可应用正则表达式的程序语言，例如 awk、Perl 以及 Python，所处理的就多半是字符串。若每个字符串表示的就是独立的一行输入，则 ^ 与 $ 仍旧可分别表示行的开头与结尾。不过这些程序语言，其实可以让你使用不同的方式来标明每条输入记录的定界符，所以有可能单独的输入“行”（记录）里会有内嵌的换行字符。这种情况下，^ 与 $ 无法匹配内嵌的换行字符；它们只用来表示字符串的开头与结尾。当你开始使用可程序化的软件工具时，这一点，请牢记在心。

3.3 字段处理

很多的应用程序，会将数据视为记录与字段的结合，以便于处理。一条记录（record）指的是相关信息的单个集合，例如以企业来说，记录可能含有顾客、供应商以及员工等数据，以学校机构来说，则可能有学生数据。而字段（field）指的就是记录的组成部分，例如姓、名或者街道地址。

3.3.1 文本文件惯例

由于UNIX鼓励使用文本型数据，因此系统上最常见的数据存储类型就是文本了，在文本文件下，一行表示一条记录。这里要介绍的是在一行内用来分隔字段的两种惯例。首先是直接使用空白（whitespace），也就是用空格键（space）或制表（tab）键：

```
$ cat myapp.data
# model     units sold     salesperson
xj11        23             jane
rj45        12             joe
cat6        65             chris
...
```

本例中，#字符起始的行表示注释，可忽略（这是一般的习惯，注释行的功能相当好用，不过软件必须可忽略这样的行才行）。各字段都以任意长度的空格（space）或制表（Tab）字符隔开。第二种惯例是使用特定的定界符来分隔字段，例如冒号：

```
$ cat myapp.data
# model:units sold:salesperson
xj11:23:jane
rj45:12:joe
cat6:65:chris
...
```

两种惯例都有其优缺点。使用空白作为分隔时，字段内容就最好不要有空白（若你使用制表字符（Tab）作分隔，字段里有空格是不会有问题的，但这么做视觉上会混淆，因为你在看文件时，很难马上分辨出它们的不同）。反过来说，若你使用显式的定界符，那么该定界符也最好不要成为数据内容。请你尽可能小心地选择定界符，让定界符出现在数据内容里的可能性降到最低或不存在。

注意：这两种方式最明显的不同，便是在处理多个连续重复的定界符之时。使用空白(whitespace)分隔时，通常多个连续出现的空格或制表字符都将看作一个定界符。然而，若使用的是特殊字符分隔，则每个定界符都隔开一个字段，例如，在myapp.data的第二个版本里使用的两个冒号字符（“::”）则会分隔出一个空的字段。

以定界符分隔字段最好的例子就是 /etc/passwd 了，在这个文件里，一行表示系统里的一个用户，每个字段都以冒号隔开。在本书中，很多地方都会以 /etc/passwd 为例，因为在系统管理工作中，很多时候都是在处理这个文件。如下是该文件的典型例子：

```
tolstoy:x:2076:10:Leo Tolstoy:/home/tolstoy:/bin/bash
```

该文件含有 7 个字段，分别是：

1. 用户名称
2. 加密后的密码（如账号为停用状态，此处为一个星号，或是若加密后的密码文件存储于另外的 /etc/shadow 里，则这里也可能是其他字符）。
3. 用户 ID 编号。
4. 用户组 ID 编号。
5. 用户的姓名，有时会另附其他相关数据（办公室号码、电话等）。
6. 根目录。
7. 登录的 Shell。

某些UNIX 工具在处理以空白界定字段的文件时，做得比较好，有些则是以定界符分隔字段比较好，更有其他的工具两种方式都能处理得当，这部分我们稍后会介绍。

3.3.2 使用 cut 选定字段

cut 命令是用来剪下文本文件里的数据，文本文件可以是字段类型或是字符类型。后一种数据类型在遇到需要从文件里剪下特定的列时，特别方便。请注意：一个制表字符在此被视为单个字符（注 8）。

举例来说，下面的命令可显示系统上每个用户的登录名称及其全名：

```
$ cut -d : -f 1,5 /etc/passwd          取出字段
root:root                              管理者账号
...
tolstoy:Leo Tolstoy                    实际用户
austen:Jane Austen
camus:Albert Camus
...
```

通过选择其他字段编号，还可以取出每个用户的根目录：

```
$ cut -d : -f 6 /etc/passwd            取出根目录
/root                                  管理账号
...
/home/tolstoy                          实际用户
/home/austen
/home/camus
...
```

通过字符列表做剪下操作有时是很方便的。例如，你只要取出命令 ls -l 的输出结果中的文件权限字段：

注 8： 这可通过 expand 与 unexpand 改变其定义。见 *expand*(1)手册页。

Cut

语法

```
cut -c list [ file ... ]
cut -f list [ -d delim ] [ file ... ]
```

用途

从输入文件中选择一或多个字段或者一组字符，配合管道（pipeline），可再做进一步处理。

主要选项

`-c list`

以字符为主，执行剪下的操作。*list*为字符编号或一段范围的列表（以逗点隔开），例如1,3,5-12,42。

`-d delim`

通过-f选项，使用*delim*作为定界符。默认的定界符为制表字符（Tab）。

`-f list`

以字段为主，作剪下的操作。*list*为字段编号或一段范围的列表（以逗点隔开）。

行为模式

剪下输入字符中指定的字段或指定的范围。若处理的是字段，则定界符隔开的即为各字段，而输出时字段也以给定的定界符隔开。若命令行没有指定文件，则读取标准输入。见正文中的范例。

警告

于POSIX系统下，cut识别多字节字符。因此，“字符（character）”与“字节（byte）”意义不同。详细内容见*cut*(1)的手册页。

有些系统对输入行的大小有所限制，尤其是含有多字节字符（multibyte characters）时，这点请特别留意。

```
$ ls -l | cut -c 1-10
total 2878
-rw-r--r--
drwxr-xr-x
-r--r--r--
-rw-r--r--
...
```

不过这种用法比使用字段的风险要大。因为你无法保证行内的每个字段长度总是一样的。一般来说，我们偏好以字段为基础来提取数据。

3.3.3 使用 join 连接字段

join命令可以将多个文件结合在一起，每个文件里的每条记录，都共享一个键值（key），键值指的是记录中的主字段，通常会是用户名称、个人姓氏、员工编号之类的数据。举例来说，有两个文件，一个列出所有业务员销售业绩，一个列出每个业务员应实现的业绩：

join

语法

join [*options* ...] *file1 file2*

用途

以共同一个键值，将已存储文件内的记录加以结合。

主要选项

-1 *field1*

-2 *field2*

标明要结合的字段。-1 *field1* 指的是从 *file1* 取出 *field1*，而 -2 *field2* 指的则为从 *file2* 取出 *field2*。字段编号自 1 开始，而非 0。

-o *file.field*

输出 *file* 文件中的 *field* 字段。一般的字段则不打印。除非使用多个 -o 选项，即可显示多个输出字段。

-t *separator*

使用 *separator* 作为输入字段分隔字符，而非使用空白。此字符也为输出的字段分隔字符。

行为模式

读取 *file1* 与 *file2*，并根据共同键值结合多笔记录。默认以空白分隔字段。输出结果则包括共同键值、来自 *file1* 的其余记录，接着 *file2* 的其余记录（指除了键值外的记录）。若 *file1* 为 -，则 join 会读取标准输入。每个文件的第一个字段是用来结合的默认键值；可以使用 -1 与 -2 更改之。默认情况下，在两个文件中未含键值的行将不打印（已有选项可改变，见 *join*(1)手册页）。

警告

-1 与 -2 选项的用法是较新的。在较旧的系统上，可能得用：-j1 *field1* 与 -j2 *field2*。

```
$ cat sales                          显示sales文件
# 业务员数据                         注释说明
# 业务员 量
joe     100
jane    200
herman  150
chris   300

$ cat quotas                         显示quotas文件
# 配额
# 业务员 配额
joe     50
jane    75
herman  80
chris   95
```

每条记录都有两个字段：业务员的名字与相对应的量。在本例中，列与列之间有多个空白，从而可以排列整齐。

为了让 join 运作得到正确结果，输入文件必须先完成排序。例 3-2 里的程序 merge-sales.sh 即为使用 join 结合两个文件。

例 3-2：merge-sales.sh

```
#! /bin/sh

# merge-sales.sh
#
# 结合配额与业务员数据

# 删除注释并排序数据文件
sed '/^#/d' quotas | sort > quotas.sorted
sed '/^#/d' sales  | sort > sales.sorted

# 以第一个键值作结合，将结果产生至标准输出
join quotas.sorted sales.sorted

# 删除缓存文件
rm quotas.sorted sales.sorted
```

首先，使用 sed 删除注释，然后再排序个别文件。排序后的缓存文件成为 join 命令的输入数据，最后删除缓存文件。这是执行后的结果：

```
$ ./merge-sales.sh
chris 95 300
herman 80 150
jane 75 200
joe 50 100
```

3.3.4 使用 awk 重新编排字段

awk 本身所提供的功能完备，已经是一个很好用的程序语言了。我们在第 9 章会好好地介绍该语言的精髓。虽然 awk 能做的事很多，但它主要的设计是要在 Shell 脚本中发挥所长：做一些简易的文本处理，例如取出字段并重新编排这一类。本节，我们将介绍 awk 的基本概念，随后你看到这样的“单命令行程序（one-liners）”就会比较了解了。

3.3.4.1 模式与操作

awk 的基本模式不同于绝大多数的程序语言。它其实比较类似于 sed：

```
awk 'program' [ file ... ]
```

awk 读取命令行上所指定的各个文件（若无，则为标准输入），一次读取一条记录（行）。再针对每一行，应用程序所指定的命令。awk 程序基本架构为：

```
pattern { action }
pattern { action }
...
```

pattern 部分几乎可以是任何表达式，但是在单命令行程序里，它通常是由斜杠括起来的 ERE。*action* 为任意的 awk 语句，但是在单命令行程序里，通常是一个直接明了的 print 语句（稍后有范例说明）。

pattern 或是 *action* 都能省略（当然，你不会两个都省略吧？）。省略 *pattern*，则会对每一条输入记录执行 *action*；省略 *action* 则等同于 { print }，将打显示整条记录（稍后将会介绍）。大部分单命令行程序为这样的形式：

```
... | awk '{ print some-stuff }' | ...
```

对每条记录来说，awk 会测试程序里的每个 *pattern*。若模式值为真（例如某条记录匹配于某正则表达式，或是一般表达式计算为真），则 awk 便执行 *action* 内的程序代码。

3.3.4.2 字段

awk 设计的重点就在字段与记录上：awk 读取输入记录（通常是一些行），然后自动将各个记录切分为字段。awk 将每条记录内的字段数目，存储到内建变量 NF。

默认以空白分隔字段 —— 例如空格与制表字符（或两者混用），像 join 那样。这通常就足够使用了，不过，其实还有其他选择：你可以将 FS 变量设置为一个不同的值，也就可以变更 awk 分隔字段的方式。如使用单个字符，则该字符出现一次，即分隔出一个字段（像 cut -d 那样）。或者，awk 特别之处就是：也可以设置它为一个完整的 ERE，这种情况下，每一个匹配在该 ERE 的文本都将视为字段分隔字符。

如需字段值，则是搭配 $ 字符。通常 $ 之后会接着一个数值常数，也可能是接着一个表达式，不过多半是使用变量名称。列举几个例子如下：

```
awk '{ print $1 }'                    打印第 1 个字段（未指定 pattern）
awk '{ print $2, $5 }'                打印第 2 与第 5 个字段（未指定 pattern）
awk '{ print $1, $NF }'               打印第 1 个与最后一个字段（未指定 pattern）
awk 'NF > 0   { print $0 }'           打印非空行（指定 pattern 与 action）
awk 'NF > 0'                          同上（未指定 action，则默认为打印）
```

比较特别的字段是编号 0：表示整条记录。

3.3.4.3 设置字段分隔字符

在一些简单程序中，你可以使用 -F 选项修改字段分隔字符。例如，显示 /etc/passwd 文件里的用户名称与全名，你可以：

```
$ awk -F: '{ print $1, $5 }' /etc/passwd      处理 /etc/passwd
root root                                     管理账号
...
tolstoy Leo Tolstoy                           实际用户
austen Jane Austen
camus Albert Camus
...
```

-F 选项会自动地设置 FS 变量。请注意，程序不必直接参照 FS 变量，也不用必须管理读取的记录并将它们分割为字段：awk 会自动完成这些事。

你可能已经发现，每个输出字段是以一个空格来分隔的 —— 即便是输入字段的分隔字符为冒号。awk 的输入、输出分隔字符用法是分开的，这点与其他工具程序不同。也就是说，必须设置 OFS 变量，改变输出字段分隔字符。方式是在命令行里使用 -v 选项，这会设置 awk 的变量。其值可以是任意的字符串。例如：

```
$ awk -F: -v 'OFS=**'  '{ print $1, $5 }' /etc/passwd     处理 /etc/passwd
root**root                                                管理者账号
...
tolstoy**Leo Tolstoy                                      实际用户
austen**Jane Austen
camus**Albert Camus
...
```

稍后就可以看到设置这些变量的其他方式。或许那些方式更易于理解，根据你的喜好而定。

3.3.4.4 打印行

就像我们已经所介绍过的：大多数时候，你只是想把选定的字段显示来，或者重新安排其顺序。简单的打印可使用print语句做到，只要提供给它需要打印的字段列表、变量或字符串即可：

```
$ awk -F: '{ print "User", $1, "is really", $5 }' /etc/passwd
User root is really root
...
User tolstoy is really Leo Tolstoy
User austen is really Jane Austen
User camus is really Albert Camus
...
```

简单明了的print语句，如果没有任何参数，则等同于print $0，即显示整条记录。

以刚才的例子来说，在混合文本与数值的情况下，多半会使用awk版本的printf语句。这和先前在2.5.4节所提及的Shell（与C）版本的printf语句相当类似，这里就不再重复。以下是把上例修改为使用printf语句的用法：

```
$ awk -F: '{ printf "User %s is really %s\n", $1, $5 }' /etc/passwd
User root is really root
...
User tolstoy is really Leo Tolstoy
User austen is really Jane Austen
User camus is really Albert Camus
...
```

awk的print语句会自动提供最后的换行字符，就像Shell层级的echo与printf那样，然而，如果使用printf语句，则用户必须要通过\n转义序列的使用自己提供它。

注意：请记得在print的参数间用逗点隔开！否则，awk将连接相邻的所有值：

```
$ awk -F: '{ print "User" $1 "is really" $5 }' /etc/passwd
Userrootis reallyroot
...
Usertolstoyis reallyLeo Tolstoy
Useraustenis reallyJane Austen
Usercamusis reallyAlbert Camus
...
```

这样将所有字符串连在一起应该不是你要的。忘了加上逗点，这是个常见又难找到的错误。

3.3.4.5 起始与清除

BEGIN与END这两个特殊的“模式”，它们提供awk程序起始（startup）与清除（cleanup）操作。常见于大型awk程序中，且通常写在个别文件里，而不是在命令行上：

```
BEGIN     {  起始操作程序代码(startup code) }

pattern1  { action1 }

pattern2  { action2 }

END       {  清除操作程序代码(cleanup code) }
```

BEGIN与END的语句块是可选用的。如需设置，习惯上（但不必须）它们应分别置于awk程序的开头与结尾处。你可以有数个BEGIN与END语句块，awk会按照它们出现在程序的顺序来执行：所有的BEGIN语句块都应该放在起始处，而所有END语句块也应放在结尾。以简单程序来看，BEGIN可用来设置变量：

```
$ awk 'BEGIN { FS = ":" ; OFS = "**" }      使用BEGIN设置变量
> { print $1, $5 }' /etc/passwd             被引用的程序继续到第二行
root**root
...
tolstoy**Leo Tolstoy                        输出，如前
austen**Jane Austen
camus**Albert Camus
...
```

警告： POSIX标准中描述了awk语言及其程序选项。POSIX awk是构建在所谓的“新awk”上，首度全球发布是在1987年的System V Release 3.1版，且在1989年的System V Release 4版中稍作修正。

但是，直到2005年底，Solaris的/bin/awk仍然还是原始的、1979年的awk V7版！在Solaris系统上，你应该使用/usr/xpg4/bin/awk，或参考第9章，使用awk自由下载版中的一个。

3.4 小结

如需从输入的数据文件中取出特定的文本行，主要的工具为grep程序。POSIX采用三种不同grep变体：grep、egrep与fgrep的功能，整合为单个版本，通过不同的选项，分别提供这三种行为模式。

虽然你可以直接查找字符串常数，但是正则表达式能提供一个更强大的方式，描述你要找的文本。大部分的字符在匹配时，表示的是自己本身，但有部分其他字符扮演的是meta字符的角色，也就是指定操作，例如“匹配0至多个的……”、“匹配正好10个的……”等。

POSIX的正则表达式有两种：基本正则表达式（BRE）以及扩展正则表达式（ERE）。哪个程序使用哪种正则表达式风格，是根据长时间的实际经验，由POSIX制定规格，简化

到只剩两种正则表达式的风格。通常，ERE比BRE功能更强大，不过不见得任何情况下都是这样。

正则表达式对于程序执行时的locale环境相当敏感；方括号表达式里的范围应避免使用，改用字符集，例如`[[:alnum:]]`较佳。另外，许多GNU 程序都有额外的 meta 字符。

`sed`是处理简单字符串替换（substitution）的主要工具。在我们的经验里，大部分的Shell脚本在使用`sed`时几乎都是用来作替换的操作，我们特意在这里不介绍`sed`所能提供的其他任务，是因为已经有《sed & awk》这本书（已列于参考书目中），它会介绍更多相关信息。

“从最左边开始，扩展至最长（longest leftmost)”，这个法则描述了匹配的文本在何处匹配以及匹配扩展到多长。在使用`sed`、`awk`或其他交互式文本编辑程序时，这个法则相当重要。除此之外，一行与一个字符串之间的差异也是核心观念。在某些程序语言里，单个字符串可能包含数行，那种情况下，`^` 与 `$` 指的分别是字符串的开头与结尾。

很多时候，在操作上可以将文本文件里的每一行视为一条单个记录，而在行内的数据则包括字段。字段可以被空白或是特殊定界符分隔，且有许多不同的UNIX 工具可处理这两种数据。`cut` 命令用以剪下选定的字符范围或字段，`join` 则是用来结合记录中具有共同键值的字段的文件。

`awk` 多半用于简单的“单命令行程序”，当你想要只显示选定的字段，或是重新安排行内的字段顺序时，就是`awk`派上用场的时候了。由于它是编程语言，即使是在简短的程序里，它也能发挥其强大的功能、灵活性与控制能力。