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。

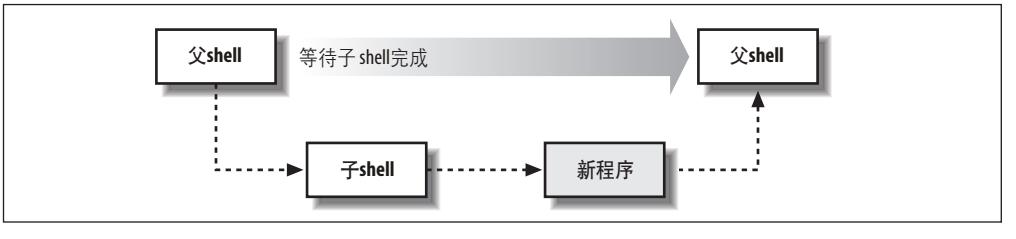

图 2-1：程序执行

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。