
```
[me@linuxbox ~]$ ps2pdf ~/Desktop/foo.ps ~/Desktop/ls.pdf
```

ps2pdf 程序是 ghostscript 软件包的一个小成员，该程序在多数支持打印的 Linux 系统上都有安装。

注意

Linux 系统通常包含许多进行文件格式转换的命令行程序，它们的命名方式通常为 format2format。尝试使用命令行 ls /usr/bin/*[[[:alpha:]]]2[[[:alpha:]]]*输出程序名列表，另外作为比较，可以尝试搜索名为 formattoformat 的程序。

作为 groff 的最后一个演示实例，重新启用“老朋友”*distros.txt* 文件。此次，使用表格格式化工具 *tbl* 对 *distros.txt* 文件中的 Linux 发行版本列表进行排版。要完成这样的操作，需要使用较早的 *sed* 脚本向 groff 将要处理的文本流中增加标记语言。

首先，需要对 *sed* 脚本做一些修改，比如增加 *tbl* 命令需要的一些必要项。使用文本编辑器，将 *distros.sed* 脚本改为如下内容。

```
# sed script to produce Linux distributions report

1 i\
.TS\
center box;\
cb s s\
cb cb cb\
1 n c.\
Linux Distributions Report\
=\
Name Version Released\
-
s/\([0-9]\{2\}\)\(\([0-9]\{2\}\)\)\(\([0-9]\{4\}\)\)$/\3-\1-\2/
$ a\
.TE
```

请注意，该脚本文件若要正常工作，必须确保其中的“*Name Version Released*”词组之间是以 tab 制表符而不是空格分开。此外，将输出结果另存为 *distros-tbl.sed* 文件。*tbl* 使用.TS 和.TE 请求作为表格的开头和末尾，跟在.TS 请求后面的行定义了表格的全局属性。就本例而言，其定义了页面内容水平居中并用文本框包围这样的属性。定义文本的其余内容则描述了每个表格中行的布局。现在，我们使用新的 *sed* 脚本处理 groff 的格式化结果，便会得到下面的内容。

```
[me@linuxbox ~]$ sort -k 1,1 -k 2n distros.txt | sed -f distros-tbl.sed | groff
-t -T ascii 2>/dev/null
+-----+
| Linux Distributions Report |
+-----+
| Name      Version     Released   |
+-----+
| Fedora    5          2006-03-20 |
```

Fedora	6	2006-10-24
Fedora	7	2007-05-31
Fedora	8	2007-11-08
Fedora	9	2008-05-13
Fedora	10	2008-11-25
SUSE	10.1	2006-05-11
SUSE	10.2	2006-12-07
SUSE	10.3	2007-10-04
SUSE	11.0	2008-06-19
Ubuntu	6.06	2006-06-01
Ubuntu	6.10	2006-10-26
Ubuntu	7.04	2007-04-19
Ubuntu	7.10	2007-10-18
Ubuntu	8.04	2008-04-24
Ubuntu	8.10	2008-10-30

groff 增加-t 选项表示先用tbl 预处理文本流。同样，增加-T 选项则输出 ASCII 格式的数据，否则会输出默认的 PostScript 格式。

如果显示终端或是打字机类型的打印机能力有限，那么 ASCII 码的输出格式其实是所能期望的最好的显示格式了。但是，如果指定了 PostScript 格式的输出，并用图形化方式查看，便会得到令人更满意的输出效果（见图 21-2）。

```
[me@linuxbox ~]$ sort -k 1,1 -k 2n distros.txt | sed -f distros-tbl.sed | groff
-t > ~/Desktop/foo.ps
```

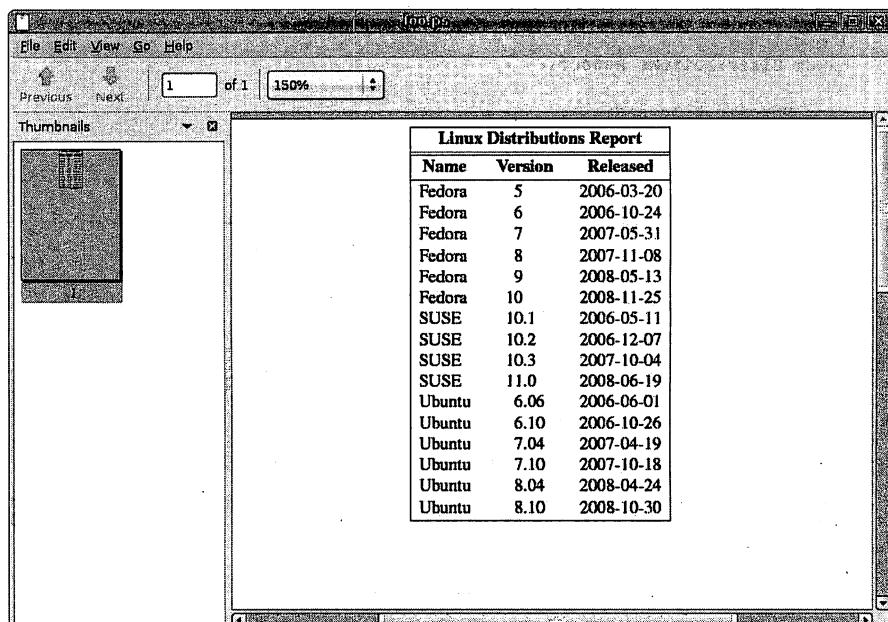


图 21-2 完成格式转换后的输出表格

21.3 本章结尾语

既然文本对于类 UNIX 操作系统中的字符是如此重要，那么有许多用于操控和格式化文本的工具就显得很有必要。我们已经知道，确实有很多这样的文本工具！简单的格式化工具，如 fmt 和 pr 在一些生成短文档的脚本中用途很大，而像 groff 这类复杂工具则可以用来排版书籍。当然，我们可能永远也不会用命令行工具来编辑技术文档（尽管也有许多人这么做！），但是知道可以用命令行完成也不失为一件好事。



第 22 章

打 印

前面几章讲述了文本的操纵，现在是时候讨论如何将文本输出到纸张上了，本章将会介绍一些用于打印文件以及控制打印机操作的命令行工具。诚然，本书并不会讨论如何配置打印参数，因为这取决于不同的发行版本，而且通常操作系统安装时就已自动设置完成。请留意，本章的实例练习中会需要一个有效的打印机配置文件。

本章会讨论如下命令。

- pr: 转换文本文件，从而进行打印操作。
- lpr: 打印文件。
- lp: 打印文件 (System V)。
- a2ps: 格式化文件，以在 PostScript 打印机上打印。
- lpstat: 显示打印状态信息。
- lpq: 显示打印机队列状态。

- `lprm`: 取消打印任务。
- `cancel`: 取消打印任务 (System V)。

22.1 打印操作简史

在充分理解类 UNIX 操作系统的打印特性之前，我们必须首先了解一些打印操作的发展史。类 UNIX 系统的打印操作可以追溯到操作系统的起源。那个时期，打印机及其使用方法都与现在有很大区别。

22.1.1 灰暗时期的打印

与计算机类似，打印机在前 PC 时代也具有庞大、昂贵和集中化的特点。20 世纪 80 年代，计算机用户都在离计算机一定距离的终端上进行操作，打印机也是置于计算机附近而处于计算机操作员的监控下工作。

UNIX 早期，打印机贵而集中化，多个用户共用一台打印机是司空见惯的事。为了区分某个任务属于哪一个具体用户，一张显示用户名的标题页 (*banner page*) 通常会在每个打印任务的开头打印出来，然后计算机辅助人员会把当天的打印任务装到一个手推车中，然后将它们配送给各个用户。

22.1.2 基于字符的打印机

与今天的打印机技术相比，20 世纪 80 年代的打印机技术在两个方面有很大的不同。第一，20 世纪 80 年代的打印机基本上都是击打式打印机。击打式打印机采取机械机制通过色带撞击页面，从而在页面上形成字符印迹。菊花轮打印和点阵打印是当时非常受欢迎的两种技术。

第二，也是更重要的一点，早期打印机使用的是设备本身所固有的字符集。例如，菊花轮打印机只能打印那些已经塑造成菊花轮花瓣形状的字符，这使得打印机就像高速打字机一样。而多数打字机打字的时候使用的是等宽 (固定宽度) 字体，这意味着每个字符都有相同的宽度。并且打印机在纸张的固定位置进行打印，打印区域包含固定数量的字符。多数打印机水平方向上每英寸打印 10 个字符 (10CPI)，垂直方向上每英寸打印 6 行 (6LPI) 内容。在这样的机制下，一张美式信纸可容纳 85 字符宽、66 行高的内容。考虑到每个边界将留有少许页边距，于是每行最多可打印 80 个字符，这就解释了为什么终端显示 (以及终端仿真器) 只有 80 个字符宽。它提供了一种以等宽字体表现的 WYSIWYG (所见即所得) 的输出视觉效果。

需要打印的字符数据以简单的字节流的形式发送给上述类打字机打印设备。例如，在打印字母 a 时，ASCII 码 97 便被发送出去。另外，低编号的 ASCII 控制码则用于移动打印机的送纸箱和纸张，以及取代回车、换行、换页符等。控制代码的使用，可以获得一定的字体效果。比如粗体可以通过先打印一个字符，然后退格再打印一次的方法来实现。如下例，我们使用 nroff 浏览 man 手册页，并且使用 cat -A 选项查看粗体效果。

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | nroff -man | cat -A | head
LS(1)                               User Commands                         LS(1)
$                                          
$                                          
$                                          
N^HNA^HAM^HME^HE$
      ls - list directory contents$
$                                          
S^HSY^HYN^HNO^HOP^HPS^HSI^HIS^HS$
  1^Hls^Hs[_^HO_^HP_^HT_^HI_^HO_^HN]...[_^HF_^HI_^HL_^HE]...$
```

其中^AH（表示 Ctrl-H 快捷键）字符是用来创建粗体效果的后退格。类似地，我们还可以用退格/下划线以产生下划线效果。

22.1.3 图形化打印机

GUI 的发展促进了打印机技术的重大变革。由于计算机日渐趋向于图形化显示，打印技术也从基于字符走向图形化。低成本的激光打印机的出现为这一转变提供了可能。激光打印机可以在页面的任何可打印区域打印一个个小点，而不是打印单个固定字符，如此便可以按比例打印文字（类似于排版机），甚至是照片以及高质量的图表。

然而，基于字符的打印机制向图形化机制的转变仍存在一个艰巨的技术性挑战。原因是，使用基于字符的打印机时，填充一页纸所需要字节数可以用如下方法计算（假定一页有 60 行，每一行包含 80 个字符）： $60 \times 80 = 4,800$ 字节。

相比较而言，一个每英寸 300 个点 (300 DPI) 的激光打印机（假设每页包含 8×10 英寸打印区域）则需要 $(8 \times 300) \times (10 \times 300) \div 8 = 900,000$ 个字节来填充一页纸。

多数慢速 PC 网络根本无法处理激光打印机打印一整页纸所需要的这近 1MB 的数据，所以很明显，我们需要一个更高端的发明。

页面描述语言 (PDL) 解决了这一问题，页面描述语言是一种描述页面内容的编程语言。它的描述方式可以简单表达为“在这个地方，使用 10 个点的无衬线字体 (Helvetica,)，打印 a 字符；在这个地方……”，直到该页所有内容被

描述结束。第一个主流页面描述语言是 Adobe 系统的 PostScript，如今仍被广泛使用。PostScript 几乎是为排版或是其他的图表图像印刷而量身定做的一套完整的编程语言。它除了自带的 35 种标准的高质量字体外，还允许用户在运行时定义额外字体。对 PostScript 的支持已内嵌在打印机中，这解决了数据传送的问题。虽然 PostScript 语言相比于基于字符的打印机使用的简单比特流显得冗长，但却比打印整页所需要的字符容量要小很多。

PostScript 打印机以 PostScript 程序作为输入。该打印机自带处理器和存储器（这些使得打印机成为比其连接的计算机功能更强的计算机），它可以运行 PostScript 解释器，该解释器读入 PostScript 程序并将其解释结果送至打印机内部的存储器，如此便形成了向纸张传送的位（点）模式。这个将程序转变为大型位模式（称为点阵图）的过程统称为光栅图形处理器，或 RIP。

随着时间的推移，计算机和网络的速度也越来越快，这使得可以让计算机来执行 RIP 过程而非让打印机执行，这使得高质量打印机的价格下降了很多。

如今的许多打印机仍然支持字符流输入，但是那些价钱比较低的打印机仍然不可以。它们主要依赖于主计算机的 RIP 提供比特流以进行点打印。当然，现今也还是有一些 PostScript 打印机的。

22.2 Linux 方式的打印

现代 Linux 系统采取两组软件来执行和管理打印操作：第一种是 CUPS（通用 UNIX 打印系统），提供打印驱动程序以及打印任务管理程序；另一种是 Ghostscript，这是一个 Postscript 编译器，充当 RIP 的作用。

CUPS 通过创建、维护打印队列进行打印机管理。正如前面在介绍打印机历史时介绍的，UNIX 打印的设计初衷是管理多用户共享的集中化打印机。由于打印机的处理速度比驱动它们的计算机慢，所以打印系统需要一种协调多个打印任务的机制，从而使所有事情能井然有序地进行。CUPS 能够识别不同类型的数据（在合理范围内），且能将文件转换为可打印的形式。

22.3 准备打印文件

作为命令行用户，我们多数还是倾向于打印文本文件，虽然其他的数据形式也可以打印。

22.3.1 pr——将文本文件转换为打印文件

前面章节略微介绍了 pr 的相关知识，本节将讨论其与打印操作联合使用时用到的一些参数选项。在前面打印机简史这一节中，我们介绍了基于字符的打印机使用等宽字体使得每页有固定数目的行并且每行有固定的字符数。pr 则用于调整文本以适应特定纸张的大小，并且可自主选择页眉和页边距。下表 22-1 总结了较常使用的一些选项。

表 22-1 常见的 pr 选项

选项	功能描述
+first[:last]	输出一个从 first 开始以 last 结束的页范围
-columns	将页的内容分成指定的 columns 列
-a	默认情况下，多列输出是垂直列出的。通过增加-a (across) 选项，内容便是水平列出
-d	隔行打印输出
-D format	用 format 格式来格式化页眉的显示日期。可以查看日期命令的 man 手册页以了解格式字符串的描述
-f	使用换页符而不是回车符作为页与页之间的分隔符
-h header	在页眉的中间部分，使用 header 代替正在处理的文件名
-l length	将页的长度设为 length。默认是 66 行（按每英寸有 6 行的美式字母算）
-n	对行进行编号
-o offset	创建一个有 offset 字符宽的左页边距
-w width	设定页面宽度为 width，默认是 72 个字符

pr 通常用于管道传输的过滤器。下面例子生成了一个/usr/bin 文件夹下的目录列表，并用 pr 将其格式化为分页的、三列的输出。

```
[me@linuxbox ~]$ ls /usr/bin | pr -3 -w 65 | head
2012-02-18 14:00
[               apturl          bsd-write
411toppm      ar              bsh
a2p           arecord         btcflash
a2ps          arecordmidi    bug-buddy
a2ps-lpr-wrapper ark            buildhash
                                         Page 1
```

22.4 向打印机发送打印任务

CUPS 打印组件支持两种打印方法，它们都是类 UNIX 系统过去使用过的。

一种方法叫做 Berkeley 或 LPD (UNIX 的 Berkeley 软件发行版本中使用的), 运用的是 lpr 命令; 另外一种方法叫做 SysV(来源于 UNIX 的 System V 版本), 运用的是 lp 命令。这两个命令大致做着相同的事情, 用户可依据个人喜好选择使用。

22.4.1 lpr——打印文件 (Berkeley 类型)

lpr 命令可以将文件传送至打印机, 同时由于其支持标准输入, 所以也可用于管道传输。例如, 要打印上述的多列目录列表, 我们可以进行如下操作。

```
[me@linuxbox ~]$ ls /usr/bin | pr -3 | lpr
```

该报告将被送至系统默认的打印机。如果想将文件发送至不同的打印机, 可以使用-P 选项, 示例如下。

```
lpr -P printer_name
```

此处 `printer_name` 是指目标打印机的名称。可以用下面的命令行查看系统打印机列表。

```
[me@linuxbox ~]$ lpstat -a
```

注意

许多 Linux 发行版本可以定义一个“虚拟打印机”, 以输出 PDF 格式的文件, 而不是在真正的物理打印机上打印, 这一特性可以方便地用于实践打印命令。检查打印机设置程序可以判断系统是否支持这一设置。对于某些发行版本, 你可能会需要安装额外的软件包 (比如像 cups-pdf 软件包) 以支持这一特性。

表 22-2 列出了 lpr 的常用选项。

表 22-2 常用的 lpr 选项

选项	功能描述
<code>-# number</code>	打印 <code>number</code> 份副本
<code>-p</code>	每一页都将包括日期、时间、工作名称和页码的页眉用阴影打印出来。这种所谓的“优质打印”选项可以用于打印文本文件
<code>-P printer</code>	指定用于打印输出的打印机名。如果未指定打印机, 那就是用系统默认的打印机
<code>-r</code>	打印结束后删除文件。此选项适用于那些产生临时打印输出文件的程序

22.4.2 lp——打印文件 (System V 类型)

与 lpr 类似，lp 命令既支持文件输入也支持标准输入。它与 lpr 的不同之处在于它有一个不同（稍微复杂点）的参数选项设置。表 22-3 列出了常用的选项。

表 22-3 常见的 lp 选项

选项	说明
-d printer	设置目标打印机为 printer。如果未指定-d 选项，将会使用系统默认的打印机
-n number	打印 number 份副本
-o landscape	将输出设置为横向
-o fitplot	根据页面大小缩放文件。这在打印诸如 JPEG 文件时非常有用
-o scaling=number	设定文件缩放比例为 number。如果该值为 100，则正好填充一页纸；如果该值小于 100，那么一页纸将填不满；如果该值大于 100，则打印内容将打印在多个页面上
-o cpi=number	设置每英寸输出字符数位 number。默认是 10
-o lpi=number	设定每英寸输出指定 number 的行，默认是 6
-o page-bottom=points	
-o page-left=points	设置页边距。其值以点的形式表示，点是排版测量单位，每英寸有 72 个点
-o page-right=points	
-o page-top=points	
-P pages	指定页列表。页的表达形式可以为用逗号隔开的页列表或是以“—”表示的页范围，如 1,3,5,7-10

让我们重新生成目录清单，此次打印的规格为 12CPI（字符/英寸）和 8LPI（行/英寸），并且左边距为 0.5 英寸。请注意，考虑到需要适应新的页面大小，所以必须调整 pr 的参数选项。

```
[me@linuxbox ~]$ ls /usr/bin | pr -4 -w 90 -l 88 | lp -o page-left=36 -o cpi=12 -o lpi=8
```

该管道使用了比默认规格更小的字体从而得到了一个 4 列列表。每行增加的字符数使得一页纸中能够容纳更多列。

22.4.3 另外一个参数选项：a2ps

a2ps 是一个非常有趣的命令，从其命令名中可以看出它是一个格式转换程序，但其功能远不止这些。该名称原义是指将 ASCII 格式转化为 PostScript 格式，并且用于为 PostScript 打印机上的打印任务准备文本文件。然而，随着时间的推

移，该程序的功能不断扩大，至今已可以将任何格式的文件转化为 PostScript 格式了。尽管表面上看起来它是一个格式转换程序，但其实实际是一个打印程序。它会将默认输出而非标准输出送至系统默认的打印机。该程序默认使用的是“优质打印机”模式，也就是说会自动优化输出内容的可视性。我们可以用该程序在桌面上创建一个 PostScript 文件。

```
[me@linuxbox ~]$ ls /usr/bin | pr -3 -t | a2ps -o ~/Desktop/ls.ps -L 66
[stdin (plain): 11 pages on 6 sheets]
[Total: 11 pages on 6 sheets] saved into the file '/home/me/Desktop/ls.ps'
```

该命令行用 pr 过滤了该文本流，使用-t 选项（省略了页眉页脚），然后使用 a2ps 指定了一个每页 66 行 (-L 选项) 的输出文件 (-o 选项) 以匹配 pr 的分页输出。如果我们选用合适的文件浏览器查看其输出结果，可以看到下图 22-1 的内容。

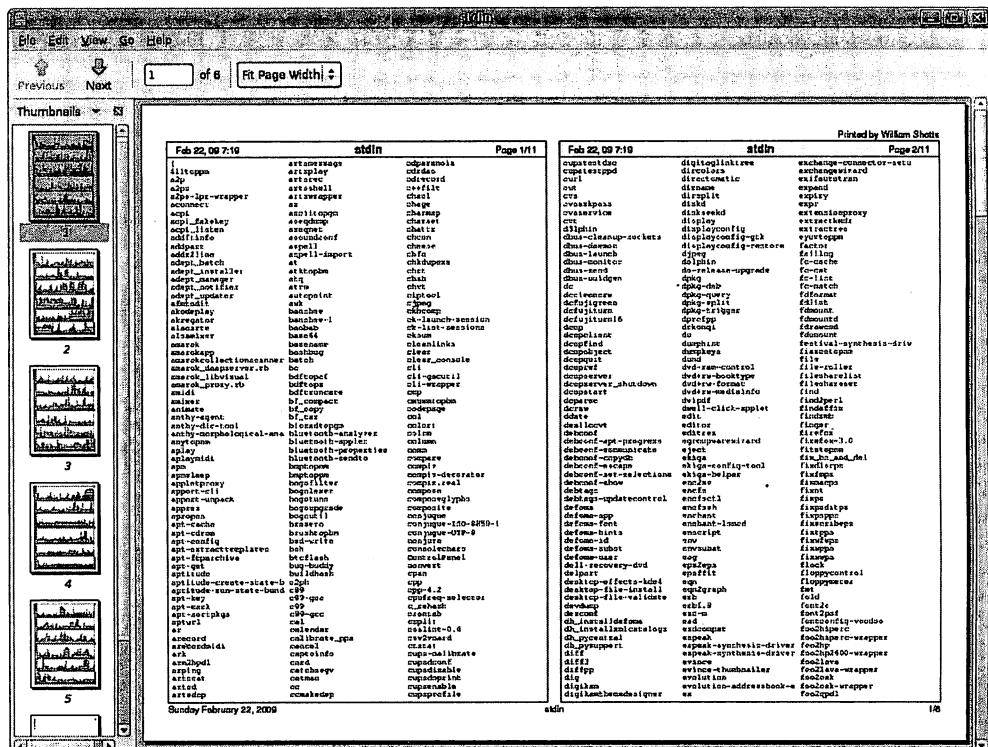


图 22-1 a2ps 命令的输出内容

可以看到，默认的输出布局是“双面” (“two up”) 格式，也就是两页纸的内容会在一张纸的正反面打印。a2ps 会自动应用美观的页眉和页脚。

a2ps 有许多选项，表 22-4 总结了一些。

表 22-4 a2ps 选项

选项	功能描述
--center-title text	将中心页标题设为 text
--columns number	将页分成 n 列。默认值是 2
--footer text	设置页脚内容为 text
--guess	给出输入文件的类型。因为 a2ps 试图转换以及格式化所有类型的数据，该选项可以用于在给定了具体文件的情况下，告诉 a2ps 下一步做什么
--left-footer text	将左页脚内容设为 text
--left-title text	将左页标题设置为 text
--line-numbers=interval	每 interval 的间隔对输出文件进行编号
--list=defaults	显示默认设置
--list=topic	显示 topic 的设置，这里的 topic 可以是下面的一种：委托（用于转换数据的外部程序）、编码、属性、变量、媒介（纸张大小及其类似的设置）、ppd（Postscript 打印机描述）、打印机、序言（位于正常输出前面的代码段）、样式表、用户选项
--pages range	打印 range 范围内的内容
--right-footer text	设置右页脚内容为 text
--right-title text	设置右页标题内容为 text
--rows number	将页面分成 number 行。默认值是 1
-B	无页眉
-b text	设置页眉内容为 text
-f size	使用 size 点的字体
-l number	设置每行有 number 个字符。该选项和下面的-L 选项可以用于诸如 pr 等的其他命令的分页处理，以适应打印纸张的大小
-L number	设置每页有 number 行
-M name	使用名为 name 的纸张大小，例如 A4
-n number	打印 n 份副本
-o file	将输出内容送至 file。如果指定 file 是 1，则是标准输出
-P printer	使用 printer 打印机，如果并未指定任何打印机，那么就使用系统的默认打印机
-R	纵向排版
-r	横向排版
-T number	设置制表符 tab 作为每一个数字字符的结尾
-u text	采用底图（水印）技术打印文本纸张

该表仅仅是一个简单的概括，a2ps 还有很多其他选项。

注意

人们仍在积极开发 a2ps 程序。就我使用 a2ps 的经历而言，不同 Linux 发行版本中 a2ps 有着不同的表现。在 CentOS 4 系统上，默认输出为标准输出。在 CentOS 4 和 Fedora 10 系统上，尽管程序已经设置默认使用信封大小的纸张，但其默认输出仍为 A4 大小的纸张。诚然，这些问题可以通过详细地定义所需要的参数选项来解决。在 Ubuntu 8.04 系统上，a2ps 则表现得如前面所记录的一样。

还要注意的是，有另外一种将文本转化为 PostScript 的工具叫做 *enscript*。它可以进行许多与 a2ps 同样的格式化和打印操作，不同的是它只支持文本输入。

22.5 监测和控制打印任务

与 UNIX 打印系统类似，CUPS 也是以处理来自多用户的多项打印任务而设计的。每个打印机都有一个打印队列，打印任务排列其中等待被送至打印机打印。CUPS 提供了几个用于管理打印机状态和打印队列命令行的程序。与 lpr 和 lp 程序类似，这些管理程序都是以 Berkeley 和 System V 打印系统相应的程序为原型的。

22.5.1 lpstat——显示打印系统状态

lpstat 程序可以确定系统中打印机的名称和可用性。例如，假定一台打印系统，既有物理打印设备（叫做 *printer*），也有 PDF 虚拟打印机（叫作 PDF），那么便可以用下面的命令行检查它们各自的状态。

```
[me@linuxbox ~]$ lpstat -a
PDF accepting requests since Mon 05 Dec 2011 03:05:59 PM EST
printer accepting requests since Tue 21 Feb 2012 08:43:22 AM EST
```

更进一步，我们可以用下面的方式得到更具体的打印系统配置描述。

```
[me@linuxbox ~]$ lpstat -s
system default destination: printer
device for PDF: cups-pdf:/
device for printer:ipp://print-server:631/printers/printer
```

本例中，*printer* 是系统默认的打印机，同时它也是一个利用网络打印协议 (ipp://) 与一个叫做 *print-server* 的系统连接的网络打印机。

表 22-5 中描述了 *lpstat* 的常用选项。

表 22-5 常用的 lpstat 选项

选项	功能描述
-a [printer...]	显示 printer 打印机的打印队列状态。请注意，该状态显示的是打印队列接受打印任务的能力，而不是物理打印机的状态。如果未指定打印机，所有的打印队列都会显示出来
-d	显示系统默认的打印机名
-p [printer...]	显示指定的 printer 状态。如果未指定打印机，所有的打印机都会显示出来
-r	显示打印服务器的状态
-s	显示状态汇总报告
-t	显示一个完整的状态报告

22.5.2 lpq——显示打印队列状态

我们可以使用 lpq 程序查看一个打印队列的状态，该程序可以查看打印机队列状态及其所包含的打印任务。如下为系统默认的打印机 printer 的一个空队列。

```
[me@linuxbox ~]$ lpq
printer is ready
no entries
```

如果事先并未指定打印机（使用-P 选项），系统便会显示默认的打印机。如果向打印机发送打印任务，然后查看打印队列，便会看到如下列表。

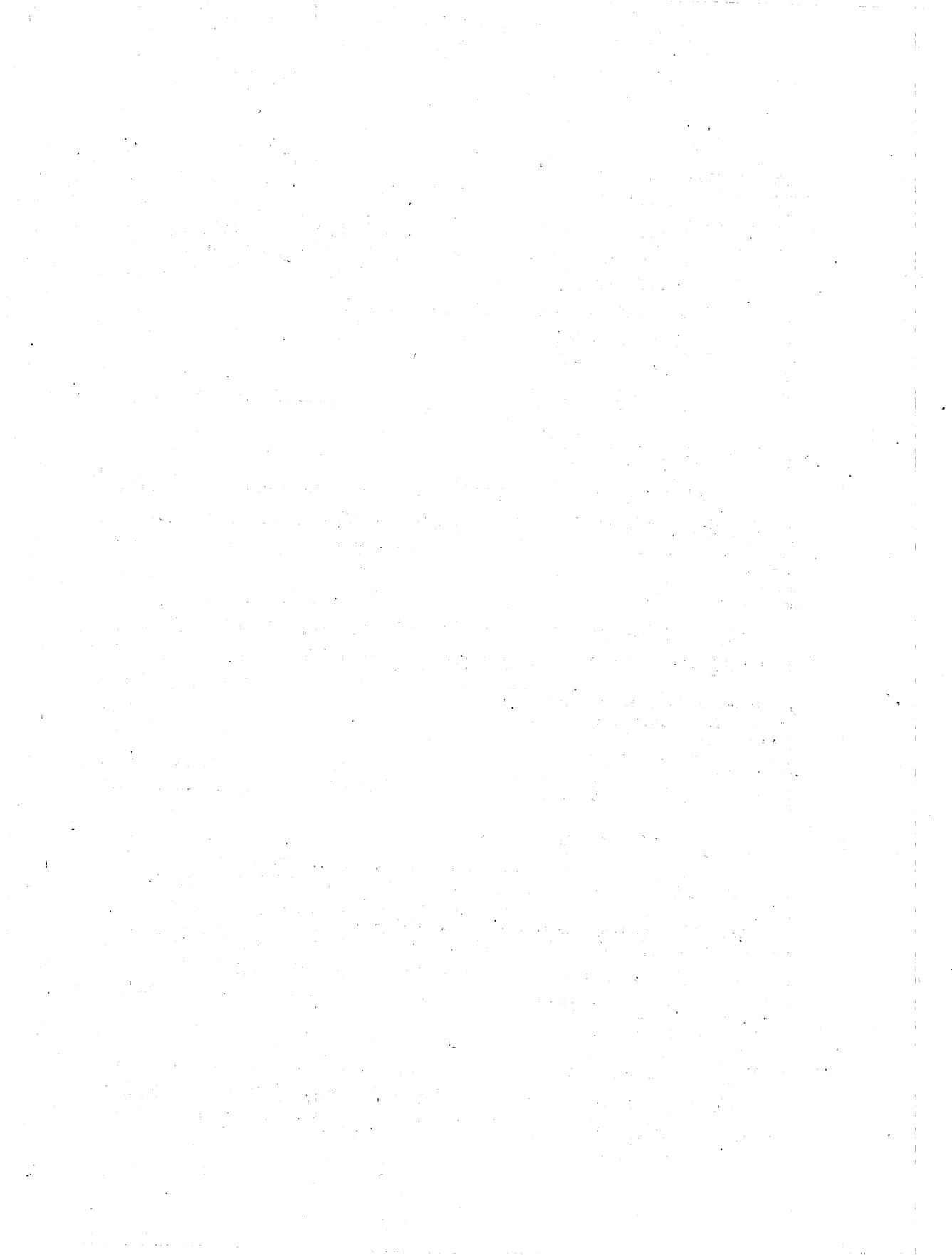
```
[me@linuxbox ~]$ ls *.txt | pr -3 | lp
request id is printer-603 (1 file(s))
[me@linuxbox ~]$ lpq
printer is ready and printing
Rank   Owner   Job   File(s)           Total Size
active  me      603   (stdin)          1024 bytes
```

22.5.3 lprm 与 cancel——删除打印任务

CUPS 提供了两个用于终止打印任务并将它们从打印队列中移除的程序。其中一个就是 Berkeley 类型的 lprm，另外一个是 System V 的 cancel。它们在所支持的参数选项上有稍许不同，但基本上实现的是同样的功能。以上面例子中使用的打印任务为例，将该打印任务终止并移除，可以使用如下命令行。

```
[me@linuxbox ~]$ cancel 603
[me@linuxbox ~]$ lpq
printer is ready
no entries
```

这两个命令都提供了一些选项，用于移除属于一个特定用户、特定打印机以及多个任务号的所有打印任务，它们各自的 man 手册页都详细介绍了其用法。



第 23 章

编译程序

本章将介绍如何通过源代码生成可执行程序。开放源代码是 Linux 自由开源的必要因素，整个 Linux 系统的开发依赖于开发人员之间的自由交流。对于多数桌面系统用户来说，编译已经是一门失传的艺术。编译技术虽然曾经非常普遍，但是如今，版本发行商却维护着很大的预编译二进制库，以便用户下载使用。在本书编写之时，Debian 系统的预编译库（所有 Linux 发行版中最大的库之一）包含了近 23000 个软件包。

那么，为什么要编译软件呢？有如下两个原因。

- **可用性：**尽管有些发行版已经包含了版本库中的一些预编译程序，但并不会包含用户所有可能需要的应用程序。这种情况下，用户获取所需要软件的唯一方式就是编译源代码。
- **及时性：**虽然有些发行版本专注于一些前沿的程序版本，但是多数并不会。这就意味着想要获取最新版本的程序，编译是必不可少的。

编译软件源代码是一项非常复杂并有技术性的任务，这远远超出了多数用户

的能力范围。然而，仍有许多非常简单的编译任务，只需要几个步骤即可完成。编译的复杂程度完全取决于所要安装的软件包。本章会介绍一个非常简单的例子，从而让大家对编译过程有一个大致了解，并为那些准备深入学习的人打下基础。

本章会介绍一个新的命令。

- make—维护程序的工具。

23.1 什么是编译

简单来说，编译就是一个将源代码（由程序员编写的人类可读的程序描述）翻译成计算机处理器能识别的语言的过程。

计算机处理器（或 CPU）在一个非常基础的层次上工作，只能运行称之为机器语言的程序。而机器语言其实就是一些数值代码，它描述的都是一些非常小的操作，比如“增加某个字节”、“指向内存中某个位置”或者“复制某个字节”等，并且每一个这样的指令都是以二进制（0 和 1）的形式表示的。最早的计算机程序就是用这样的数值代码编写，这也许可以解释为什么写这些数值代码的程序员吸很多烟，喝大量咖啡，戴着厚重的眼镜。

汇编语言的出现解决了这一问题，因为它用诸如 CPY（复制）和 MOV（转移）等更容易的助记符取代了那些数值代码，汇编语言编写的程序会由汇编程序（assembler）处理成机器语言。如今，汇编语言仍然用于某些专门的编程任务，诸如设备驱动和嵌入式系统等。

之后出现了高级编程语言，被称为高级语言是因为它们可以让程序员少关注些处理器的操作细节而把更多的精力集中在解决手头的问题上。早期（20 世纪 50 年代开发的）的高级语言包括 FORTRAN（专为科学技术任务设计）和 COBOL（专为商务应用设计），两者至今仍然在某些专门领域发挥着作用。

虽然现在流行的编程语言有很多，但有两个占据着主导地位，C 和 C++便是多数现代系统采用的编程语言。下面的内容中，我们编译了一个 C 语言程序作为例子进行讲解。

高级语言编写的程序通过编译器转换为机器语言。有些编译器则将高级语言程序转换为汇编语言，然后再使用一个汇编程序（assembler）将其转换为机器语言。

经常与编译一起使用的步骤就是链接。程序执行着许多共同的任务。例如，打

开一个文件，许多程序都会需要进行此操作。如果每个程序都采用自己的方式实现该功能的话便是一种浪费。编写一个用于打开文件的单个程序，并允许其他程序共享它，反而更有意义。提供这种通用任务支持功能的便是库，库中包含了多个例程，每一个实现的都是许多程序能够共享的通用任务。在/lib 和/usr/lib 目录中，我们可以发现很多这样的程序。链接器（linker）程序可以实现编译器的输出与编译程序所需要库之间的链接。该操作的最后结果就是生成一个供使用的可执行文件。

23.2 是不是所有的程序都需要编译

答案是否定的。我们都知道，有些程序，如 shell 脚本，可以直接运行而不需要编译，这些文件都是用脚本或解释型语言编写的。这些语言在近些年来越来越受欢迎，其中就有 Perl、Python、PHP、Ruby 以及其他多种语言等。

脚本语言由一个称为解释器的特殊程序来执行，解释器负责输入程序文件并执行其所包含的所有指令。通常来讲，解释型程序要比编译后的程序执行起来慢。这是因为在解释型程序中，每条源代码指令在执行时都要重新翻译一次该源代码指令。然而在编译后的程序中，每条源代码指令只翻译一次，并且该翻译结果将永久地记录到最后的可执行文件中。

那为什么解释型语言如此受欢迎？其实对于许多日常的编程工作，解释型程序的执行速度也是足够的，而其真正的优点在于开发解释型程序要比开发编译程序简单而迅速得多。程序开发总是经历着这样一个重复的循环——编码、编译和测试。随着程序规模的逐渐扩大，编译时间也逐渐变长。解释型程序则省略了编译这一过程，因此加速了程序的开发。

23.3 编译一个 C 程序

现在我们可以编译程序了。然而，在执行编译操作前，需要一些工具，诸如编译器、链接器以及 make 等。gcc（GNU C 编译器）是 Linux 环境中通用的 C 编译器，最初是由 Richard Stallman 编写的。多数 Linux 发行版本并不会默认安装 gcc，我们可以用下面的命令行查看系统是否安装了该编译器。

```
[me@linuxbox ~]$ which gcc  
/usr/bin/gcc
```

该例子的结果表明已安装了此编译器。

注意

你所使用的 Linux 版本也许提供了一个用于软件开发的元数据包（一个软件包集）。如果这样，你可以直接安装该软件包便可在系统上编译程序。但如果 你的系统并未提供该元数据包，则只能自己安装 gcc 和 make 软件包。当然，针对多数发行版本，安装这两个软件包已经足够执行接下来的例程。

23.3.1 获取源代码

从一个叫做 *diction* 的 GNU 项目中选择一个程序进行编译练习，这个方便短小的程序一般用于检查文本文件的质量和写作风格，它非常小并且很容易生成。

依据惯例，我们首先创建一个 *src* 目录用于存放源代码，然后使用 *ftp* 下载源代码至该目录。

```
[me@linuxbox ~]$ mkdir src
[me@linuxbox ~]$ cd src
[me@linuxbox src]$ ftp ftp.gnu.org
Connected to ftp.gnu.org.
220 GNU FTP server ready.
Name (ftp.gnu.org:me): anonymous
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd gnu/diction
250 Directory successfully changed.
ftp> ls
200 PORT command successful. Consider using PASV.
150 Here comes the directory listing.
-rw-r--r-- 1 1003 65534 68940 Aug 28 1998 diction-0.7.tar.gz
-rw-r--r-- 1 1003 65534 90957 Mar 04 2002 diction-1.02.tar.gz
-rw-r--r-- 1 1003 65534 141062 Sep 17 2007 diction-1.11.tar.gz
226 Directory send OK.
ftp> get diction-1.11.tar.gz
local: diction-1.11.tar.gz remote: diction-1.11.tar.gz
200 PORT command successful. Consider using PASV.
150 Opening BINARY mode data connection for diction-1.11.tar.gz (141062
bytes).
226 File send OK.
141062 bytes received in 0.16 secs (847.4 kB/s)
ftp> bye
221 Goodbye.
[me@linuxbox src]$ ls
diction-1.11.tar.gz
```

注意

由于编译源代码时我们便是源代码的维护者，所以我们将其存放于`~/src` 目录中。发行版本自行安装的源代码一般安装于`/usr/src` 目录下，而面向多用户使用的源代码则通常安装在`/usr/local/src` 目录中。

我们可以看到，源代码通常以一个压缩的 tar 文件的形式存在，有时被称为 *tarball*，该文件包含了源代码树，即构成该源代码的目录及文件的组织框架。连接到 FTP 站点后，我们便可以查看可用的 tar 文件列表并挑选其中最新的版本进行下载。在 ftp 中使用 get 命令，即可将文件从 FTP 服务器复制到本地主机。

下载了该 tar 文件后，就必须对其进行解压缩，这是通过 tar 程序来完成的：

```
[me@linuxbox src]$ tar xzf diction-1.11.tar.gz
[me@linuxbox src]$ ls
diction-1.11      diction-1.11.tar.gz
```

注意

该 diction 程序与所有其他 GNU 项目的软件一样，都遵循一定的源代码打包标准，Linux 系统中许多其他可用的源代码同样遵循这样的标准。该标准规定，当源代码的 tar 文件解压缩后，会创建一个包含源代码树的目录，并且该目录以 project-x.xx 的格式命名，此格式包含了该项目的名称及其版本号。这一机制有助于实现不同版本之间同一程序的安装。然而，在解压之前先检查源代码树的布局也是很有必要的，因为有些项目并不会创建目录而是将所有文件直接送至当前目录，这可能会给原先井然有序的 src 目录造成混乱。为了避免这样的事情发生，我们可以使用下面的命令行检查 tar 文件的内容。

```
tar tzvf tarfile | head
```

23.3.2 检查源代码树

解压 tar 文件会产生一个新的目录——diction-1.11。该目录包含了源文件树，具体内容如下。

```
[me@linuxbox src]$ cd diction-1.11
[me@linuxbox diction-1.11]$ ls
config.guess    diction.c        getopt.c       n1
config.h.in     diction.pot      getopt.h       n1.po
config.sub      diction.spec    getopt_int.h  README
configure       diction.spec.in INSTALL       sentence.c
configure.in    diction.texi.in  install-sh    sentence.h
COPYING         en              Makefile.in   style.1.in
de              en_GB           misc.c       style.c
de.po          en_GB.po       misc.h       test
diction.1.in    getopt1.c      NEWS
```

此解压缩包中包含了许多文件。与其他许多程序相似，GNU 项目的程序都有提供如 README、INSTALL、NEWS 和 COPYING 等这些文档文件。这些文件包含的是程序的描述、安装步骤说明及其许可条款。在着手生成程序前，

认真阅读 README 和 INSTALL 文档是很有必要的。

此目录下的其他有趣文件便是这些以.c 和.h 结尾的文件。

```
[me@linuxbox diction-1.11]$ ls *.c
diction.c getopt1.c getopt.c misc.c sentence.c style.c
[me@linuxbox diction-1.11]$ ls *.h
getopt.h getopt_int.h misc.h sentence.h
```

软件包提供的两个 C 程序 (style 和 diction) 便是以.c 结尾，并且它们都被分成了多个模块。如今，这种将一些大的程序分成较小的、易于管理的小程序片的做法已是司空见惯。这些源代码文件都是普通的文本文件，可以用 less 查看。

```
[me@linuxbox diction-1.11]$ less diction.c
```

.h 文件是大家熟知的头文件。这些文件，同样也是普通的文本文件。头文件中包含了对源代码文件或库中的例程的描述。编译器在链接这些例程模块时，必须给它提供一个其所用到的所有模块的描述。因此，在 diction.c 的开头，可以看到如下文本行。

```
#include "getopt.h"
```

该文本行会指示编译器在读取 diction.c 中的源代码内容时先读取文件 getopt.h 中的内容，进而读取 getopt.c 中的内容。getopt.c 文件包含的是由 style 和 diction 程序所共享的例程模块。

在 getopt.h 的 include 语句上面，还可以看到一些其他 include 语句，如下所示。

```
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

这些都是用来引用头文件，但是它们引用的是那些不在当前源目录下的头文件。它们由系统提供，为每个程序的编译提供支持。如果查看/usr/include 目录，便会看到如下内容。

```
[me@linuxbox diction-1.11]$ ls /usr/include
```

该目录下的头文件在安装编译器时便已安装。

23.3.3 生成程序

大多数程序都是使用一个简单的两行命令来生成的。

```
./configure
make
```

`configure` 程序其实是源代码树下的一个 shell 脚本，它的任务就是分析生成环境。多数源代码都设计成可移植的。也就是说，源代码可以在多种类型的 UNIX 系统上生成，只是源代码在生成时可能需要经过细微的调整以适应各系统之间的不同。`configure` 同样会检查系统是否已经安装了必要的外部工具和组件。

接下来运行 `configure`。由于 `configure` 并不是存放于 shell 通常期望程序所在的目录下，所以必须显式告知 shell 有关 `configure` 的位置，我们可在命令前添加“`./`”目录符来实现这一目的。该符号表示 `configure` 程序位于在当前的工作目录下。

```
[me@linuxbox diction-1.11]$ ./configure
```

`configure` 会在检查以及配置 build 程序的同时输出许多相关信息。运行结束后，会输出如下类似内容。

```
checking libintl.h presence... yes
checking for libintl.h... yes
checking for library containing gettext... none required
configure: creating ./config.status
config.status: creating Makefile
config.status: creating diction.1
config.status: creating diction.texi
config.status: creating diction.spec
config.status: creating style.1
config.status: creating test/rundiction
config.status: creating config.h
[me@linuxbox diction-1.11]$
```

需要重点注意的是，这里并没有错误信息。如果有的话，该 `configure` 操作将以失败告终，并且不会生成可执行程序，直到所有的错误都被纠正过来。

可以看到，`configure` 在源目录中创建了几个新文件，其中最重要的就是 `Makefile`。`Makefile` 是指导 `make` 命令如何生成可执行程序的配置文件，如果没有该文件，`make` 便无法运行。`Makefile` 也是一个普通的文本文件，所以我们可以用 `less` 查看其内容。

```
[me@linuxbox diction-1.11]$ less Makefile
```

`make` 程序的作用其实就是输入 `makefile`（通常叫做 `Makefile`），该文件描述了生成最后可执行程序时的各部件之间的联系及依赖关系。

`makefile` 的第一部分内容定义了一些变量，这些变量在 `makefile` 的后面部分

将被替换掉。示例如下。

CC=	gcc
-----	-----

该行将 C 编译器定义为 gcc。然后在 makefile 的后面内容中，我们可以看到如下使用 CC 的例子。

diction:	diction.o sentence.o misc.o getopt.o getopt1.o \$(CC) -o \$@ \$(LDFLAGS) diction.o sentence.o misc.o \ getopt.o getopt1.o \$(LIBS)
----------	--

该例中包含了一个替代操作，在运行时\$(CC)便被替换为 gcc。

大多数 makefile 文件都有很多行，这些行定义了目标文件（在本例中是 diction 可执行文件），也定义了目标文件所依赖的一些文件，剩下的行则描述的是那些将原文件生成目标文件的命令。就本例而言，我们可以看到可执行文件 diction 依赖于文件 diction.o、sentence.o、misc.o、getopt.o 以及 getopt1.o 的存在。继续查看 makefile 内容，我们可以看到这些目标文件的定义。

diction.o:	diction.c config.h getopt.h misc.h sentence.h
getopt.o:	getopt.c getopt.h getopt_int.h
getopt1.o:	getopt1.c getopt.h getopt_int.h
misc.o:	misc.c config.h misc.h
sentence.o:	sentence.c config.h misc.h sentence.h
style.o:	style.c config.h getopt.h misc.h sentence.h

然而，我们并没有看到为它们指定的任何命令。其实，它们是由文件前面的一个总体目标行生成的，该目标行描述了用来将所有.c 文件编译为.o 文件的命令。

.c.o:	\$(CC) -c \$(CPPFLAGS) \$(CFLAGS) \$<
-------	---------------------------------------

该命令行看起来确实复杂，那为什么不简单地列出编译的所有步骤然后照着步骤做呢？答案即将揭晓。揭晓前，我们先运行 make，来生成我们的程序。

[me@linuxbox diction-1.11]\$ make

make 程序运行时，会使用 Makefile 文件中的内容指导其操作，其间会产生许多信息。

运行结束后，我们会看到所有的目标文件都出现在了目录中。

[me@linuxbox diction-1.11]\$ ls				
config.guess	de.po	en	install-sh	sentence.c
config.h	diction	en_GB	Makefile	sentence.h
config.h.in	diction.1	en_GB.mo	Makefile.in	sentence.o
config.log	diction.1.in	en_GB.po	misc.c	style

config.status	diction.c	getopt1.c	misc.h	style.i
config.sub	diction.o	getopt1.o	misc.o	style.i.in
configure	diction.pot	getopt.c	NEWS	style.c
configure.in	diction.spec	getopt.h	nl	style.o
COPYING	diction.spec.in	getopt_int.h	nl.mo	test
de	diction.texi	getopt.o	nl.po	
de.mo	diction.texi.in	INSTALL	README	

在这些文件中，我们也看到了 `diction` 和 `style` 程序，这是我们最初希望生成的。是不是应该庆祝一番，因为我们刚刚成功利用源代码编译了第一个可执行程序！

好吧，抑制住好奇心，再次运行 `make` 命令。

```
[me@linuxbox diction-1.11]$ make
make: Nothing to be done for 'all'.
```

仅仅出现了这样一个奇怪的信息。到底怎么回事？为什么它没有再次生成该程序呢？其实，这便是 `make` 的神奇之处。`make` 并不是简单盲目地重新生成所有东西，它只会生成那些需要生成的文件。在所有的目标文件已经存在的情况下，`make` 会判定源文件没有任何改动，也就不会进行任何操作。当然，可以删除其中某个目标文件，然后再次运行 `make` 以观察 `make` 的执行情况。

```
[me@linuxbox diction-1.11]$ rm getopt.o
[me@linuxbox diction-1.11]$ make
```

可以看到，`make` 重新生成了 `getopt.o`，并重新链接 `diction` 和 `style` 程序，因为它们都依赖于被删除的文件。这一特性同样指出了 `make` 的另一个用法——可以维护目标文件的更新。`make` 坚持一个原则，就是目标文件要比依赖文件新。这个特性有着重要的意义。因为程序员会经常更新部分源代码，然后使用 `make` 生成一个新版本。`make` 能够确保所有需要基于刚更新的代码而生成的程序都会生成。如果使用 `touch` 命令更新上例中某个源代码文件，就会得到下面的结果。

```
[me@linuxbox diction-1.11]$ ls -l diction getopt.c
-rwxr-xr-x 1 me      me      37164 2009-03-05 06:14 diction
-rw-r--r-- 1 me      me      33125 2007-03-30 17:45 getopt.c
[me@linuxbox diction-1.11]$ touch getopt.c
[me@linuxbox diction-1.11]$ ls -l diction getopt.c
-rwxr-xr-x 1 me      me      37164 2009-03-05 06:14 diction
-rw-r--r-- 1 me      me      33125 2009-03-05 06:23 getopt.c
[me@linuxbox diction-1.11]$ make
```

`make` 运行后，可以看到目标文件已经比依赖文件新了。

```
[me@linuxbox diction-1.11]$ ls -l diction getopt.c
-rwxr-xr-x 1 me      me      37164 2009-03-05 06:24 diction
-rw-r--r-- 1 me      me      33125 2009-03-05 06:23 getopt.c
```

make 能够智能地仅生成需要执行 building 操作的目标文件，这一能力让程序员获益不少。其所带来的时间节省效益，虽然对于一些小的项目而言并不是那么明显，但是对于大项目来说确是非常重要。要知道，Linux 内核（一个不断进行修改和完善的程序）可是包含几百万行的代码。

23.3.4 安装程序

打包好的源代码一般包含一个特殊的 make 目标程序，它便是 install。该目标程序将会在系统目录下安装最后生成的可执行程序。通常，会安装在目录 /usr/local/bin 下，该目录是本地主机上生成软件的常用安装目录。然而，对于普通用户，该目录通常是不可写的，所以必须转换成超级用户才可以运行安装。

```
[me@linuxbox diction-1.11]$ sudo make install
```

安装结束后，就可以查看该程序是否可以运行。

```
[me@linuxbox diction-1.11]$ which diction
/usr/local/bin/diction
[me@linuxbox diction-1.11]$ man diction
```

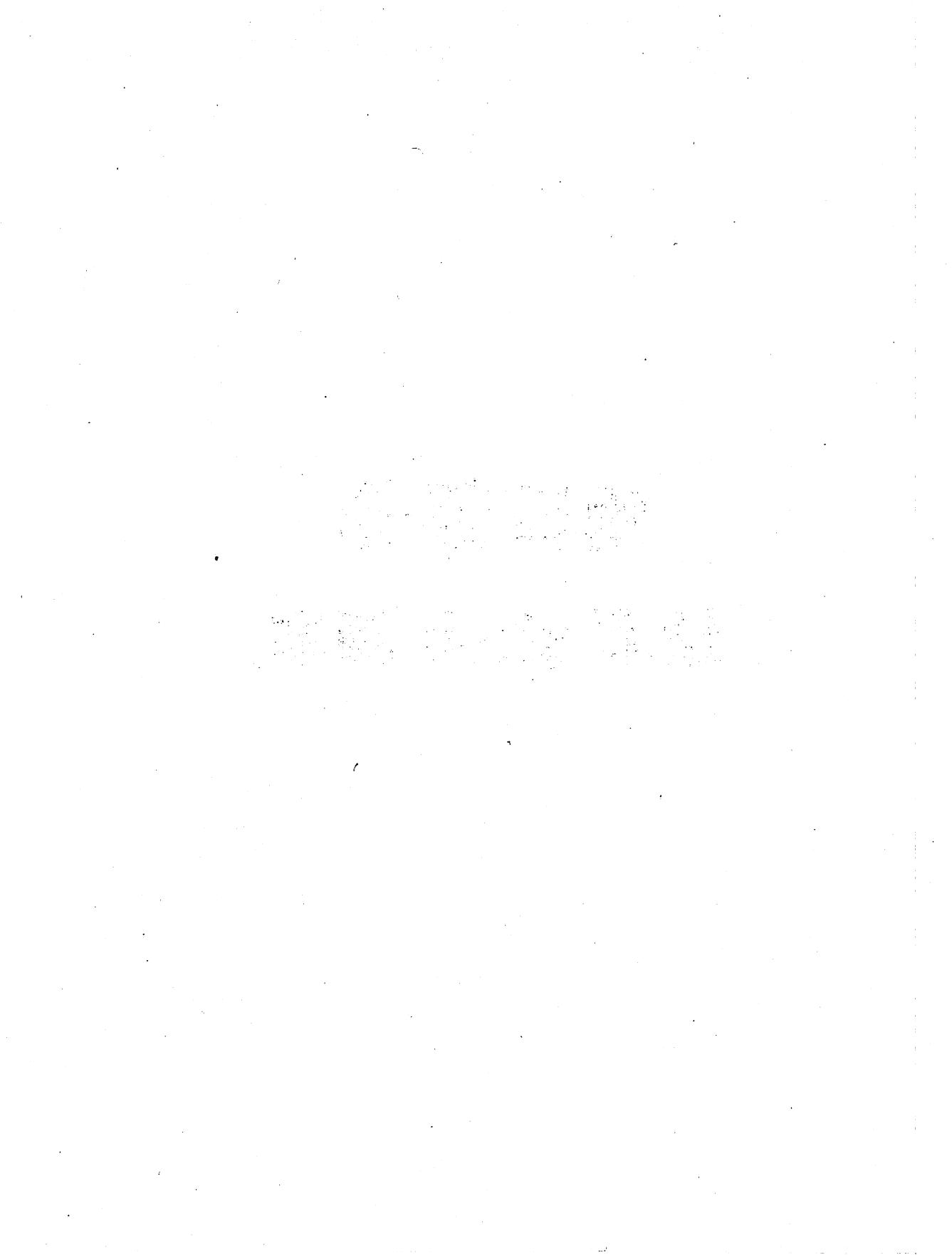
在安装完之后，便可以使用该应用该程序了。

23.4 本章结尾语

本章中我们介绍了三个简单的命令，它们是./configure、make 和 make install，它们可以用于建立多个源代码软件包。同样，我们还介绍了 make 在软件维护的过程中所扮演的重要角色。make 程序并不局限于源代码编译，它还可以用于所有需要维护“目标/依赖”间关系的任务。

第四部分

编写 shell 脚本



第 24 章

编写第一个 shell 脚本

在之前的章节中，我们已经学习了一系列的命令行工具。虽然这些工具可以解决很多计算问题，但是我们在使用它们时只能在命令行中一个一个手动输入。如果可以让 shell 完成更多工作，岂不是更好？当然可以。通过自行设计，将命令行组合成程序的方式，shell 就可以独立完成一系列复杂的任务。我们可以通过编写 shell 脚本方式来实现。

24.1 什么是 shell 脚本

最简单的解释是，shell 脚本是一个包含一系列命令的文件。shell 读取这个文件，然后执行这些命令，就好像这些命令是直接输入到命令行中一样。

shell 很独特，因为它既是一个强大的命令行接口，也是一个脚本语言解释器。我们将会看到，大多数能够在命令行中完成的工作都可以在脚本中完成，反之亦然。

我们已经讲解了许多 shell 特性，但我们关注的是哪些经常直接在命令行中使用的特性。shell 还提供了一些通常（但不总是）在编写程序时才使用的特性。

24.2 怎样写 shell 脚本

为了成功创建和运行一个 shell 脚本，我们需要做三件事情。

1. 编写脚本。shell 脚本是普通的文本文件。所以我们需要一个文本编辑器来编辑它。最好的文本编辑器可以提供“语法高亮”功能，从而能够看到脚本元素彩色代码视图。“语法高亮”可以定位一些常见的错误。vim、gedit、kate 和许多其他的文本编辑器都是编写 shell 脚本的不错选择。
2. 使脚本可执行。系统相当严格，它不会将任何老式的文本文件当做程序。这样做有充足的理由！所以我们需要将脚本文件的权限设置为允许执行。
3. 将脚本放置在 shell 能够发现的位置。当没有显式指定路径名时，shell 会自动地寻找某些目录，来查找可执行文件。为了最大程度的方便，我们会将脚本放置在这些目录下。

24.2.1 脚本文件的格式

为了保持编程的传统，我们将创建一个“hello world”的程序，演示一个非常简单的脚本。启动文本编辑器并且输入以下脚本。

```
#!/bin/bash

# This is our first script.

echo 'Hello World!'
```

这个脚本的最后一行看起来非常熟悉，仅仅是一个 echo 命令加上一个字符串参数。第二行也很熟悉，看起来很像在很多配置文件中用到的注释行。就 shell 脚本中的注释来说，它们可以放置在一行的最后，如下所示。

```
echo 'Hello World!' # This is a comment too
```

文本行中，在“#”符号后面的所有内容会被忽略：

很很多命令一样，它也是在命令行中工作：

```
[me@linuxbox ~]$ echo 'Hello World!' # This is a comment too
Hello World!
```

尽管命令行中的注释没有用，但是他们也能起作用。

脚本的第一行看起来有点神奇。由于它以符号“#”开头，看起来像是注释，但是它应该具有一定的意义，所以它不仅仅是注释。实际上，这个“#!”字符序列是一种特殊的结构，称之为 shebang。shebang 用来告知操作系统，执行后面的脚本应该使用的解释器的名字。每一个 shell 脚本都应该将其作为第一行。

将这个脚本保存为 `hello_world`。

24.2.2 可执行权限

下一步要做的事情是让脚本可执行。使用 `chmod` 命令可以轻松做到：

```
[me@linuxbox ~]$ ls -l hello_world
-rw-r--r-- 1 me me 63 2012-03-07 10:10 hello_world
[me@linuxbox ~]$ chmod 755 hello_world
[me@linuxbox ~]$ ls -l hello_world
-rwxr-xr-x 1 me me 63 2012-03-07 10:10 hello_world
```

对于脚本，有两种常见的权限设置：权限为 755 的脚本，每个人都可以执行；而权限为 700 的脚本，则只有脚本所有人才能执行。注意，为了能够执行脚本，它必须是可读的。

24.2.3 脚本文件的位置

设置完权限之后，现在来执行脚本：

```
[me@linuxbox ~]$ ./hello_world
Hello World!
```

为了使脚本运行，我们必须显式指定脚本文件的路径。如果不这样做，我得到下面的结果：

```
[me@linuxbox ~]$ hello_world
bash: hello_world: command not found
```

为何会这样呢？是什么让脚本有别于程序呢？结果证明，什么都没有。脚本本身没有问题，问题在于脚本的位置。在第 11 章，我们讨论了 PATH 环境变量，以及它对系统搜索可执行程序方面的影响。如果没有显式指定路径，则系统在查找可一个执行程序时，需要搜索一系列目录。这就是当我们在命令行中输入 `ls` 时，系统知道要执行 `/bin/ls` 的原因。`/bin` 目录是系统会自动搜索的一个目录。目录列表存放在名为 `PATH` 的环境变量中。这个 `PATH` 变量包含一个由冒号分隔开的待搜索目录的列表。我们可以查看 `PATH` 的内容：

```
[me@linuxbox ~]$ echo $PATH
```

```
/home/me/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr
/games
```

我们在这里看到了目录列表。如果脚本位于该列表中的任何一个目录中，问题就解决了。注意列表中的第一个目录/home/me/bin。大多数Linux发行版会配置PATH变量，使其包含用户主目录中的bin目录，允许用户执行他们自己的程序。如果我们创建了bin目录，并且将脚本放置在这个目录内，该脚本应该就会向其他程序那样开始运行。

```
[me@linuxbox ~]$ mkdir bin
[me@linuxbox ~]$ mv hello_world bin
[me@linuxbox ~]$ hello_world
Hello World!
```

如果PATH环境变量不包括这个bin目录，我们也可以轻松进行添加，方法是在.bashrc文件中增加下面这行：

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

修改完毕之后，它会在每一个新的终端会话中生效。为了将这一修改应用到当前的终端会话，则必须让shell重新读取.bashrc文件。读取的方式如下。

```
[me@linuxbox ~]$ . .bashrc
```

这个“.”命令和source命令相同，是shell内置命令，用来读取一个指定的shell命令文件，并将其看做是像从键盘中输入的一样。

注意

在Ubuntu系统中，如果存在~/bin目录，则当执行用户的.bashrc文件时，Ubuntu系统会自动将~/bin目录添加到PATH变量中。所以，在Ubuntu操作系统中，如果创建了~/bin目录然后退出并再重新登录，一切也会正常运行。

24.2.4 脚本的理想位置

~/bin目录是一个存放个人使用脚本的理想位置。如果我们编写了一个系统上所有用户都可以使用的脚本，则该脚本的传统位置是/usr/local/bin。系统管理员使用的脚本通常放置在/usr/local/sbin。在大多数情况下，本地支持的软件，无论是脚本或者是编译好的程序，应该放置在/usr/local目录下，而不是/bin或是/usr/bin目录下。这些目录都是由Linux文件系统层次结构标准指定的，只能包含由Linxu发行商所提供的和维护的文件。

24.3 更多的格式诀窍

之所以严肃认真地编写脚本，其中一个目的是为维护提供便利。容易维护的脚本可以被它的作者或其他人员进行修改，以适应变化的要求。而让脚本易于阅读和理解是一种方便维护的方法。

24.3.1 长选项名

我们学习的很多命令都有短选项名和长选项名。例如，ls 命令有很多选项，它们既可以用长选项名表示，也可以用短选项名表示。例如：

```
[me@linuxbox ~]$ ls -ad
```

和

```
[me@linuxbox ~]$ ls --all --directory
```

这两个命令一样。为了减少输入，当在命令行中输入选项时，短选项更可取。但是在编写脚本时，长选项名可以提高可读性。

24.3.2 缩进和行连接

当使用长选项命令时，将命令扩展为好几行，可以提高命令的可读性。在第 17 章中，我们查看了一个特别长的 find 命令示例。

```
[me@linuxbox ~]$ find playground \( -type f -not -perm 0600 -exec chmod 0600 {} ';' \) -or \( -type d -not -perm 0700 -exec chmod 0700 {} ';' \)
```

乍一看，该命令有点难以理解。在脚本中，如果以如下方式编写，该命令就比较容易理解了。

```
find playground \
  \( \
    -type f \
    -not -perm 0600 \
    -exec chmod 0600 {} ';' \
  \) \
  -or \
  \( \
    -type d \
    -not -perm 0700 \
    -exec chmod 0700 {} ';' \
  \)
```

通过行连接符（反斜杠-回车符序列）和缩进，读者可以清楚地理解这个复杂命令的逻辑。该技术在命令行中也奏效，但是很少使用，原因就是在输入和编辑时会相当麻烦。脚本和命令行的一个区别是，脚本可以使用制表符来实现缩进，但在命令行中，Tab 键用来激活自动补齐功能。

为编写脚本而配置 vim

vim 文本编辑器有很多的配置项。有几个常用的选项为编写脚本提供了方便。

“:syntax on” 用来打开“语法高亮”。打开这个选项后，在查看 shell 时，不同的 shell 语法元素会以不同的颜色显示。这对于识别某些编程错误很有帮助。它看起来也很酷。注意，为了使用这种功能，必须安装 vim 文本编辑器的完整版，并且编辑的文件必须含有 shebang 来标识这是一个 shell 脚本文件。如果无法设置“:syntax on”，可以试试“:set syntax=sh”。

“:set health” 用来将搜索的结果高亮显示。比如我们查找单词“echo”，在开启该选项后，则所有的“echo”单词都会高亮显示。

“:set tabstop=4” 用来设置 Tab 键造成的空格长度。默认值是 8 列。将这个值设置成 4，会让长文本行更容易适应屏幕。

“:set autoindent” 用来开启自动缩进特性。这个选项会让 vim 对新的一行的缩进程度和上一行保持一致。对很多编程结构来说，这就加快了输入速度。要停止缩进，则可以按下 Ctrl-D。

通过把这些命令（不需要前面的冒号字符）添加到`./vimrc`文件，则这些改变会永久生效。

24.5 本章结尾语

本章与脚本有关，我们介绍了如何编写脚本，以及如何轻松地在系统中执行脚本。我们还介绍了如何使用各种格式技术来提升脚本的可读性（以及可维护性）。在随后的章节中，易于维护将作为编写良好脚本的核心原则多次出现。

第 25 章

启动一个项目

从本章开始，我们将开始构建一个程序。该项目的目的是查看如何使用各种 shell 特性来创建程序，而且更为重要的是，如何创建好的程序。

我们将要编写的程序是一个报告生成器，它会显示系统的各种统计数据和状态。并以 HTML 格式来产生该报告。这样，我们就可以使用 Web 浏览器进行查看。

程序一般由一系列阶段组成，每一个阶段都会增加一些特性和功能。该程序的第一个阶段是创建一个非常小的 HTML 页面，它不包含任何系统信息（后面会添加这些信息）。

25.1 第一阶段：最小的文档

我们需要知道的第一件事就是一个结构良好的 HTML 文档的格式，如下所示。

```
<HTML>
  <HEAD>
    <TITLE>Page Title</TITLE>
```

```

</HEAD>
<BODY>
    Page body.
</BODY>
</HTML>

```

如果将这些内容输入到文本编辑器，并将该文件保存为 `foo.html`，就可以在火狐浏览器中输入“`file:///home/username/foo.html`”这个 URL 地址查看该文件。

程序的第一个阶段是将这个 HTML 文件输出到标准输出。我们可以编写一个程序，来很容易地完成该任务。启动文本编辑器，然后创建一个名为“`~/bin/sys_info_page`”的新文件。

```
[me@linuxbox ~]$ vim ~/bin/sys_info_page
```

随后输入以下的程序。

```

#!/bin/bash

# Program to output a system information page

echo "<HTML>"
echo "  <HEAD>"
echo "    <TITLE>Page Title</TITLE>"
echo "  </HEAD>"
echo "  <BODY>"
echo "    Page body."
echo "  </BODY>"
echo "</HTML>"

```

我们第一次要尝试解决的这个程序包含了一个“shebang”、一条“注释”（总是一个比较好的习惯）和一系列 `echo` 命令。每一个 `echo` 命令用来显示一行。保存该文件之后，就其成为可执行文件，并尝试去运行它。

```
[me@linuxbox ~]$ chmod 755 ~/bin/sys_info_page
[me@linuxbox ~]$ sys_info_page
```

该程序在运行时，我们就可以看到这个 HTML 文档的文本显示在屏幕上，这是因为脚本中的 `echo` 命令将其输出发送到了标准输出。再一次运行该程序，并且将该程序的输出重新定向到 `sys_info_page.html` 文件，这样就可以用 Web 浏览器查看结果了。

```
[me@linuxbox ~]$ sys_info_page > sys_info_page.html
[me@linuxbox ~]$ firefox sys_info_page.html
```

目前为止，一切顺利。

在编写程序时，最好尽力保持程序的清晰明了。当一个程序易于阅读和理解时，维护起来也会很容易，而且通过减少输入量，也会程序更容易编写。该

程序的当前版本虽然工作良好，但是可以更简洁一些。实际上，我们可以将所有的 echo 命令整合为一个命令，这样就可以更容易地将更多的行添加到程序的输出中。现在，将程序修改为如下所示：

```
#!/bin/bash

# Program to output a system information page

echo "<HTML>
<HEAD>
    <TITLE>Page Title</TITLE>
</HEAD>
<BODY>
    Page body.
</BODY>
</HTML>"
```

一个带引号的字符串可以包含换行符，因此也就可以包含多个文本行。shell 将持续读取文本，直到读取到下一个引号为止。它在命令行中也是这样工作的：

```
[me@linuxbox ~]$ echo "<HTML>
>     <HEAD>
>         <TITLE>Page Title</TITLE>
>     </HEAD>
>     <BODY>
>         Page body.
>     </BODY>
> </HTML>"
```

每行开头的“>”字符是包含在 PS2 shell 变量中的 shell 提示符。每当我们在 shell 中输入多行语句时就会出现。现在来看，这个功能现在还不是很明显，但当我们介绍多行编程语句时，它会相当方便。

25.2 第二阶段：加入一点数据

现在该程序可以生成一个最小的文档，我们在这个文档中加入一些数据。为了实现这个目标，需要做以下改动。

```
#!/bin/bash

# Program to output a system information page

echo "<HTML>
<HEAD>
    <TITLE>System Information Report</TITLE>
</HEAD>
<BODY>
    <H1>System Information Report</H1>
</BODY>
</HTML>"
```

我们加入了一个页面标题，并为报告正文部分添加了一个标题。

25.3 变量和常量

但是，脚本现在还有一个问题。请注意字符串“System Information Report”是如何被重复的。对于我们这个小脚本来说，这不是问题。但是，如果我们的脚本很长，而且多处都存在该字符串，现在我们想将标题修改为其他内容，则需要在多个地方进行修改，这样工作量就大了。我们是否可以修改脚本，使得字符串只出现一次而不是多次呢？而且这也会让脚本在日后的维护更加简单。为此，可以这样做：

```
#!/bin/bash

# Program to output a system information page

title="System Information Report"

echo "<HTML>
  <HEAD>
    <TITLE>$title</TITLE>
  </HEAD>
  <BODY>
    <H1>$title</H1>
  </BODY>
</HTML>"
```

通过创建一个叫名为 title 的变量，并且将其赋值为“System Information Report”，就可以利用参数扩展能，将该字符串放置到多个地方了。

25.3.1 创建变量和常量

如何创建变量呢？这很简单，我们刚刚使用过。当 shell 遇到一个变量时，会自动创建这个变量。这点与大多数程序中，使用一个变量时必须先声明或定义有所不同。在这个方面，shell 非常的宽松，但也会导致一些问题。例如，请看以下命令行中出现的场景。

```
[me@linuxbox ~]$ foo="yes"
[me@linuxbox ~]$ echo $foo
yes
[me@linuxbox ~]$ echo $fool
[me@linuxbox ~]$
```

首先我们为变量 foo 赋值 yes，再使用 echo 命令显示 foo 的值。然后我们显示一个由拼写错误造成的变量 fool 的值，而得到了一个空值。这是因为 shell 在

遇到 `fool` 变量时，会轻松地创建这个变量，并且为这个 `fool` 变量赋了一个默认的空值。从这点可以看出，我们必须对拼写有足够的关注。同进，也要理解在这个示例中，究竟发生了什么。我们前面学到，shell 会执行扩展，所以命令

```
[me@linuxbox ~]$ echo $foo
```

会经历参数扩展，而且结果是：

```
[me@linuxbox ~]$ echo yes
```

然而命令

```
[me@linuxbox ~]$ echo $fool
```

则扩展为

```
[me@linuxbox ~]$ echo
```

这个为空的变量在扩展后为空。这对需要使用参数的命令说来，将会引起混乱。例如：

```
[me@linuxbox ~]$ foo=foo.txt
[me@linuxbox ~]$ fool=fool.txt
[me@linuxbox ~]$ cp $foo $fool
cp: missing destination file operand after 'foo.txt'
Try 'cp --help' for more information.
```

我们为变量 `foo` 和 `fool` 赋值，然后执行 `cp` 命令，但将第二个参数名拼错了。在参数扩展后，这个 `cp` 命令仅接受了一个参数，但是它需要的是两个。

变量的命名规则如下所示。

- 变量名称应由字母、数字和下划线组成。
- 变量名称的第一个字符必须是字母或者下划线。
- 变量名称中不允许空格和标点。

变量意味着值会发生变化，并且在大多数应用中，变量就是这样使用的。在前面的应用中，变量 `title` 其实是作为常量使用的。常量其实就是一个有名称和确定值的变量。区别就是，常量的值不会发生变化。在一个执行几何计算的应用中，我们可以定义 `PI` 为一个常量，并且将它的值赋为 `3.1415`，从而无需在程序中使用数值 `3.1415`。对于 shell 来说，变量和常量没有区别，这种划分更多的是为了编程者的方便。较为普遍的约定是，我们使用大写字母表示常量，使用小写字母表示变量。我们将前面的 shell 按照这个约定进行修改。

```
#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"

echo "<HTML>
    <HEAD>
        <TITLE>$TITLE</TITLE>
    </HEAD>
    <BODY>
        <H1>$TITLE</H1>
    </BODY>
</HTML>"
```

利用这个机会，我们为 shell 中的变量 HOSTNAME 赋值。这个 HOSTNAME 是机器在网络上的名称。

注意

实际上，shell 还提供了强制常量不发生变化的方法，这是通过使用带有-r 选项(只读)的 declare 内置命令实现的。我们在这里使用这种方式为变量 TITLE 赋值：

```
declare -r TITLE=“Page Title”
```

shell 将阻止一切后续的向 TITLE 的赋值。这个特性很少使用，但在很早的脚本中曾经存在。

25.3.2 为变量和常量赋值

这是我们开始真正使用扩展的地方。我们已经看到，变量用以下方式来赋值：

```
variable=value
```

其中，variable 是变量的名称，value 是变量的值。和大多数的编程语言不同，shell 并不关心赋给变量的值数值类型，它会将其都当成字符串。通过使用带-i 选项的 declare 命令，可以强制 shell 将变量限制为整型数值。但是，这如同将变量设置成只读模式一样，我们很少这样做。

注意，在赋值时，变量名、等号和值之间不能含有空格。那么，变量的值中可以包括什么呢？它包含的是可以扩展为字符串的任意值：

a=z	#将字符串 “z” 赋值给变量 a
b=“a string”	#嵌入的空格必须用引号括起来
c=“a string and \$b”	#可以被扩展到赋值语句中的其他扩展，比如变量
d=\$(ls-l foo.txt)	#命令的结果
e=\$((5*7))	#算术扩展
f=“\t\ta string\n”	#转义序列，比如制表符和换行符

可以在一行中给多个变量赋值。

```
a=5 b="a string"
```

在扩展期间，变量名称可以用花括号“{}”括起来。当变量名因为周围的上下文而变得不明确时，这就会很有帮助了。在这里，我们使用变量将一个文件的名字由 myfile 改为 myfile1：

```
[me@linuxbox ~]$ filename="myfile"
[me@linuxbox ~]$ touch $filename
[me@linuxbox ~]$ mv $filename ${filename}1
mv: missing destination file operand after 'myfile'
Try 'mv --help' for more information.
```

因为 shell 将 mv 命令的第二个参数当成了一个新的变量，所以这样做没有成功。该问题可以用以下方法解决。

```
[me@linuxbox ~]$ mv $filename ${filename}1
```

用花括号括起来后，shell 将不会把跟在后面的“1”认为是变量名称的一部分。

我们现在添加一些数据到我们的报告中，比如报告创建的日期、时间，以及报告创建者的用户名。

```
#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

echo "<HTML>
  <HEAD>
    <TITLE>$TITLE</TITLE>
  </HEAD>
  <BODY>
    <H1>$TITLE</H1>
    <P>$TIME_STAMP</P>
  </BODY>
</HTML>"
```

25.4 here 文档

我们已经了解了两种通过 echo 命令输出文本的不同方法。此外还有称之为“here 文档”或“here 脚本”的第三种方法来输出文本。here 文档是 I/O 重定向

的另外一种形式，我们在脚本中嵌入正文本，然后将其输入到一个命令的标准输入中。工作方式如下：

```
command << token
text
token
```

其中，`command` 是接受标准输入的命令名，`token` 是用来指示嵌入文本结尾的字符串。现在修改脚本，使其使用 `here` 文档。

```
#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

cat << _EOF_
<HTML>
  <HEAD>
    <TITLE>$TITLE</TITLE>
  </HEAD>
  <BODY>
    <H1>$TITLE</H1>
    <P>$TIME_STAMP</P>
  </BODY>
</HTML>
_EOF_
```

我们的脚本不再使用 `echo`，而是使用 `cat` 和 `here` 文档。字符串 `_EOF_`（含义是“文件结尾”）被选作 `token`，并且指示嵌入文本的结尾。注意，`token` 必须在一行中单独出现，而且文本行的末尾没有空格。

那么使用 `here` 文档有什么好处呢？它和 `echo` 命令很类似，但是在默认情况下，`here` 文档内的单引号和双引号将失去它们在在 `shell` 中的特殊含义。下面是一个命令行示例。

```
[me@linuxbox ~]$ foo="some text"
[me@linuxbox ~]$ cat << _EOF_
> $foo
> "$foo"
> '$foo'
> \'$foo
> _EOF_
some text
"some text"
'some text'
$foo
```

可以看到，`shell` 没有注意到引号，而是将引号当做普通字符。这就可以在

here 文档中随意嵌入引号。这给我们的报告程序带来了方便。

here 文档可以和能够接受标准输入的任何命令一起使用。在本例中，我们使用 here 文档向 `ftp` 程序传递一系列命令，以从远程 FTP 服务器上获取一个文件。

```
#!/bin/bash

# Script to retrieve a file via FTP

FTP_SERVER=ftp.nl.debian.org
FTP_PATH=/debian/dists/lenny/main/installer-i386/current/images/cdrom
REMOTE_FILE=debian-cd_info.tar.gz

ftp -n << _EOF_
open $FTP_SERVER
user anonymous me@linuxbox
cd $FTP_PATH
hash
get $REMOTE_FILE
bye
_EOF_
ls -l $REMOTE_FILE
```

如果将重定向操作符由“`<<`”改为“`<<-`”，shell 就会忽略在 here 文档中开图的 Tab 字符。这样就能缩进 here 文档，从而提供可读性。

```
#!/bin/bash

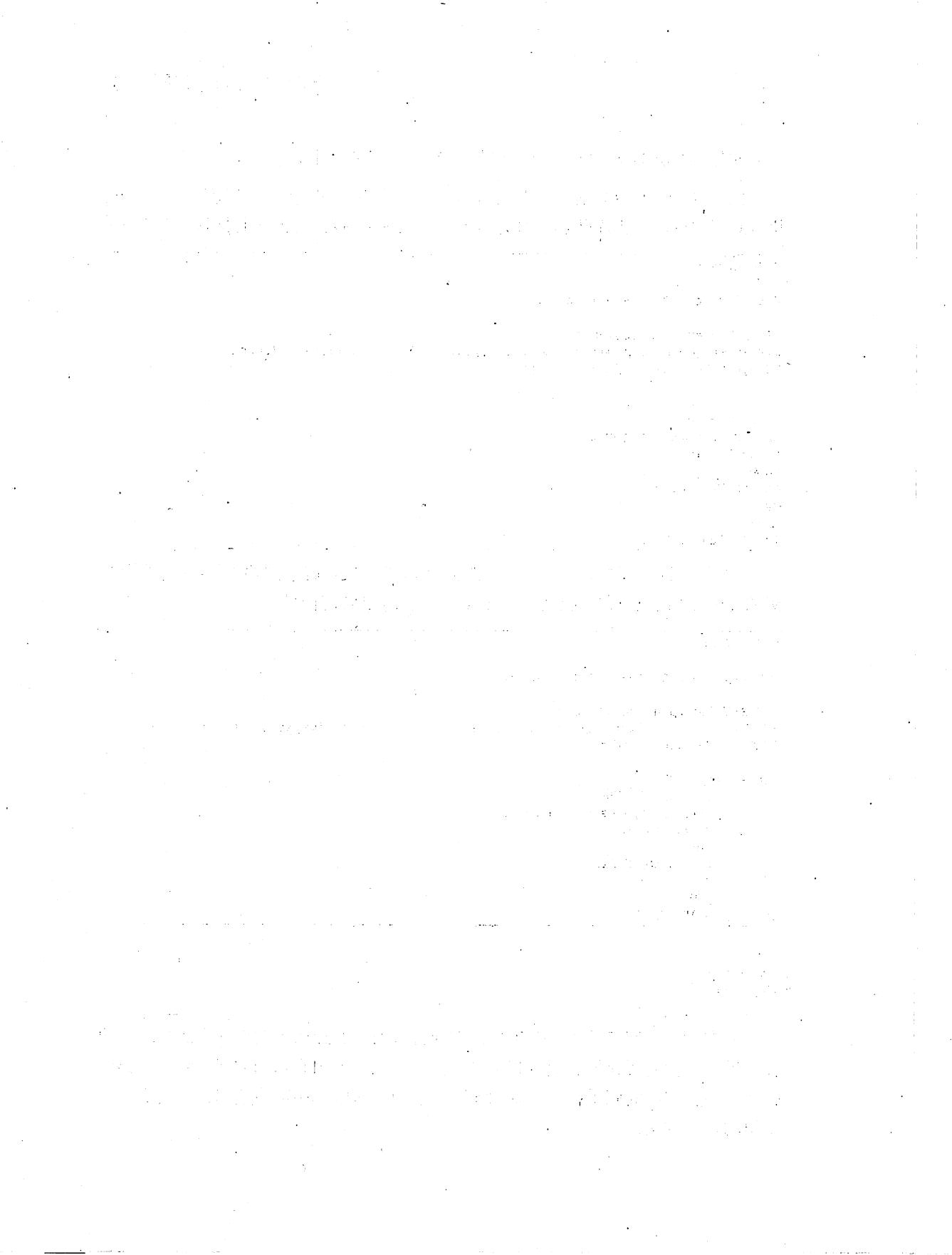
# Script to retrieve a file via FTP

FTP_SERVER=ftp.nl.debian.org
FTP_PATH=/debian/dists/lenny/main/installer-i386/current/images/cdrom
REMOTE_FILE=debian-cd_info.tar.gz

ftp -n <<- _EOF_
    open $FTP_SERVER
    user anonymous me@linuxbox
    cd $FTP_PATH
    hash
    get $REMOTE_FILE
    bye
_EOF_
ls -l $REMOTE_FILE
```

25.5 本章结尾语

在本章，我们启动了一个项目，该项目带领我们经历了构建一个成功脚本的整个过程。我们还介绍了变量和常量的概念，以及使用它们的方式。它们是第一批使用参数扩展的应用程序。我们还查看了如何从脚本中产生输出，以及嵌入文本块的各种方法。



第 26 章

自顶向下设计

随着程序越来越大，越来越复杂，设计、编码和维护都将越来越困难。所以，在设计任何大型项目时，我们最好将庞大的、复杂的任务，拆分成一系列小的、简单的任务。

想象一下，我们来描述一个常见的日常任：一个火星人去市场买食物。我们可以按照如下步骤来描述整个过程。

1. 上车。
2. 开往市场。
3. 停车。
4. 进入市场。
5. 购买食物。
6. 回到车里。

7. 开车回家。

8. 停车。

9. 进屋。

但是，这个火星人需要更多细节。我们可以将子任务“停车”进一步拆分为一系列步骤。

1. 找停车的位置。

2. 将车开入停车位。

3. 关闭发动机。

4. 拉手刹。

5. 下车。

6. 锁上车门。

“关闭发动机”子任务又可以进一步的分解成包括“熄火”和“拔出点火钥匙”等步骤。这样逐步分解，直到全部定义了“去市场”这个过程的整个步骤。

这种先确定上层步骤，然后再逐步细化这些步骤的过程，称为自顶向下设计。

通过这种设计方式，可以将大而复杂的任务分解成很多小而简单的任务。自顶向下设计是一种设计程序的常见方式，尤其适合 shell 编程。

本章我们将会使用自顶向下设计方式，进一步开发我们的报告生成器脚本。

26.1 shell 函数

我们的报告当前执行如下步骤，即可生成 HTML 文档。

1. 打开页面。

2. 打开页面标题。

3. 设置页面标题。

4. 关闭页面标题。

5. 打开页面主体。

6. 输出页面主体。

7. 输出时间戳。
8. 关闭页面主题。
9. 关闭页面。

为了下一阶段的开发，我们将在步骤 7 和步骤 8 之间额外添加一些任务，如下所示。

- **系统正常运行时间和负载。**这是自上次关机或重启之后系统的运行时间，在几个时间间隔内，当前运行在处理器上的平均任务量。
- **磁盘空间。**系统存储空间的使用情况。
- **用户空间。**每个用户所使用的存储空间。

在这些任务中，如果每个任务都有一条对应的命令，我们可以直接通过命令替换的方式，将它们添加到脚本中。

```
#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"

CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

cat << _EOF_
<HTML>
<HEAD>
    <TITLE>$TITLE</TITLE>
</HEAD>
<BODY>
    <H1>$TITLE</H1>
    <P>$TIME_STAMP</P>
    $(report_uptime)
    $(report_disk_space)
    $(report_home_space)
</BODY>
</HTML>
_EOF_
```

我们可以通过两种方式创建这些额外的命令。我们可以编写 3 个独立的脚本，并把它们放置到环境变量 PATH 所列出的目录中，或者是我们可以将脚本作为 shell 函数嵌入到程序中。前面提到，shell 函数是位于其他脚本中的迷你脚本，可以用作自主程序（autonomous program）。shell 函数有两种语法形式。第一种如下所示：

```
function name {
    commands
    return
}
```

其中 *name* 是指这个函数的名称，*commands* 是这个函数中的一系列命令。第二种看起来如下所示：

```
name () {
    commands
    return
}
```

这两种格式等价，并且可以交替使用。来看下面这个脚本，它演示了 shell 函数的使用。

```
1  #!/bin/bash
2
3  # Shell function demo
4
5  function funct {
6      echo "Step 2"
7      return
8  }
9
10 # Main program starts here
11
12 echo "Step 1"
13 funct
14 echo "Step 3"
```

当 shell 读取脚本的时候，它会跳过第 1~11 行，这些行包含的是注释和函数的定义。执行从带有 echo 命令的第 12 行开始。第 13 行调用了 shell 函数 funct，shell 会执行这个函数，而且其执行方式与执行其他命令时相同。程序控制权然后移动到第 6 行，执行第 2 个 echo 命令，随后再执行第 7 行。这个 return 命令终止函数的执行，然后将控制权还给函数调用后面的代码（第 14 行），然后执行最后一个 echo 命令。注意，为了让函数调用被识别为 shell 函数，而不是被解释为外部程序的名字，shell 函数的定义在脚本中的位置必须在它被调用的前面。

我们在脚本中添加上最小的 shell 函数定义：

```
#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

report_uptime () {
    return
}
report_disk_space () {
    return
}
report_home_space () {
```

```

        return
    }

cat << _EOF_
<HTML>
<HEAD>
    <TITLE>$TITLE</TITLE>
</HEAD>
<BODY>
    <H1>$TITLE</H1>
    <P>$TIME_STAMP</P>
    $(report_uptime)
    $(report_disk_space)
    $(report_home_space)
</BODY>
</HTML>
_EOF_

```

shell 函数的命名规则和变量相同。一个函数必须至少包含一条命令。return 命令（可选的）可以满足该要求。

26.2 局部变量

在目前我们所编写的脚本中，所有的变量（包括常量）都是全局变量。全局变量在整个程序期间会一直存在。在很多情况下，这都不错。但是有时候，它会让 shell 函数的使用变得复杂。在 shell 函数中，经常需要的是局部变量。局部变量仅仅在定义它们的 shell 函数中有效，一旦 shell 函数终止，它们就不再存在。

局部变量可以让程序员使用已经存在的变量名称，无论是脚本中的全局变量，还是其他 shell 函数中的变量，而不用考虑潜在的命名冲突。

下面是一个显示如何定义和使用局部变量的脚本示例。

```

#!/bin/bash

# local-vars: script to demonstrate local variables

foo=0 # global variable foo

funct_1 () {

    local foo # variable foo local to funct_1

    foo=1
    echo "funct_1: foo = $foo"
}

funct_2 () {
    local foo # variable foo local to funct_2

    foo=2
    echo "funct_2: foo = $foo"
}

```

```
echo "global: foo = $foo"
funct_1
echo "global: foo = $foo"
funct_2
echo "global: foo = $foo"
```

可以看到，局部变量是通过在变量名前面添加单词 local 来定义的。这样，就创建并同时定义了一个 shell 函数中的局部变量。一旦出了这个 shell 函数，这个局部变量将不再存在。当运行该脚本时，结果如下所示：

```
[me@linuxbox ~]$ local-vars
global: foo = 0
funct_1: foo = 1
global: foo = 0
funct_2: foo = 2
global: foo = 0
```

可以看到，在两个 shell 函数中对局部变量 foo 的赋值，不影响 shell 函数以外定义的变量 foo 的值。

这个特性可以让我们编写的 shell 函数相互独立，而且也独立于它们所在的脚本。这非常有用，因为它有助于阻止程序中各个部分的相互干扰。该特性也可以让我们编写出可移植的 shell 函数。也就是说，我们可以根据需要将某一个脚本中的 shell 函数剪切下来，然后粘贴到另外一个脚本中。

26.3 保持脚本的运行

在开发程序时，让程序保持可运行的状态会非常有用。通过这种方式，并经常测试，就可以在开发过程的早期检测到错误。这也会让问题的调试变得容易。例如，如果我们运行某个程序，并对它做了细微改动，然后再次运行该程序，此时发现了问题。这可能有可能是最近的改动导致的。通过添加空函数（程序员将其称为 stub），我们可以在早期验证程序的逻辑流程。当构建一个 stub 时，最好是包含一些能够为程序员提供反馈信息的东西，这些反馈信息可以显示正在执行的逻辑流程。如果现在查看脚本的输出，会发现在时间戳之后的输出中有一些空行，但是我们不确定是什么原因导致的。

```
[me@linuxbox ~]$ sys_info_page
<HTML>
  <HEAD>
    <TITLE>System Information Report For twin2</TITLE>
  </HEAD>
  <BODY>
    <H1>System Information Report For linuxbox</H1>
```

```
<P>Generated 03/19/2012 04:02:10 PM EDT, by me</P>
```

```
</BODY>
</HTML>
```

我们修改该函数，使其包含一些反馈信息。

```
report_uptime () {
    echo "Function report_uptime executed."
    return
}

report_disk_space () {
    echo "Function report_disk_space executed."
    return
}

report_home_space () {
    echo "Function report_home_space executed."
    return
}
```

然后再次运行该脚本。

```
[me@linuxbox ~]$ sys_info_page
<HTML>
    <HEAD>
        <TITLE>System Information Report For linuxbox</TITLE>
    </HEAD>
    <BODY>
        <H1>System Information Report For linuxbox</H1>
        <P>Generated 03/20/2012 05:17:26 AM EDT, by me</P>
        Function report_uptime executed.
        Function report_disk_space executed.
        Function report_home_space executed.
    </BODY>
</HTML>
```

这次可以发现，我们的三个函数事实上都执行了。

由于我们的函数框架没有问题，因此，需要更新一些函数代码。首先是 `report_uptime` 函数。

```
report_uptime () {
    cat <<- _EOF_
        <H2>System Uptime</H2>
        <PRE>$uptime</PRE>
    _EOF_
    return
}
```

该函数的代码直截了当。我们使用了一个 `here` 文档来输出一个标题，并输出 `uptime` 命令的结果。命令结果使用`<PRE>`标签围起来，其目的是保留命令的格式。`report_disk_space` 函数也类似。

```
report_disk_space () {
    cat <<- _EOF_
        <H2>Disk Space Utilization</H2>
        <PRE>$(df -h)</PRE>
    _EOF_
    return
}
```

这个函数使用了 `df -h` 命令来检查磁盘的空间。随后，我们来构建 `report_home_space` 函数。

```
report_home_space () {
    cat <<- _EOF_
        <H2>Home Space Utilization</H2>
        <PRE>$(du -sh /home/*)</PRE>
    _EOF_
    return
}
```

我们使用带`-sh` 选项的 `du` 命令来执行这个任务。这个并不是问题的完整解决方案。虽然它可以在某些操作系统中（例如 Ubuntu）运行，但是在其他系统中则无效。这是因为很多系统设置了主目录的权限，以防止被其他用户读取，这种安全措施也很合理。在这些操作系统中，只有在超级用户运行我们的脚本时，我们编写的这个 `report_home_space` 函数才会执行。一个更好的解决方案是让脚本根据用户的权限调整自己的行为。我们在下一章讨论该问题。

你的.bashrc 文件中的 shell 函数

`shell` 函数可以很好地取代别名，并且实际上也是创建个人使用的小命令的首选方法。别名非常局限于命令的种类和它们支持的 `shell` 特性，而 `shell` 特性则允许任何可以编写为脚本的东西。例如，如果我们很喜欢为脚本开发的 `report_disk_space` 这个 `shell` 函数，则可以为我们的 `.bashrc` 文件创建一个相似的名为 `ds` 的函数。

```
ds () {
    echo "Disk Space Utilization For $HOSTNAME"
    df -h
}
```

26.4 本章结尾语

在本章，我们介绍了一种常用的一种程序方式——自顶向下设计，并知道了如何使用 `shell` 函数按照要求来构建逐步细化的任务。我们还学习了如何使用局部变量时 `shell` 函数相互独立，并独立于它们所在的程序。这样，我们就可以以可移植和可重用的方式编写 `shell` 函数，并将其用到多个程序中，从而节省大量的时间。

第 27 章

流控制：IF 分支语句

上一章中，我们提出了一个问题。如何使我们的报告生成器脚本能适应运行该脚本的用户的权限？该问题的解决方案要求我们能找到一种方法，在脚本中能基于测试的结果来“改变方向”。用编程术语来说，就是我们需要程序能够分支。

让我们来考虑一个使用伪代码表示的简单逻辑示例。伪代码是计算机语言的一种模拟，为的是方便人们理解。

```
X = 5
If X = 5, then:
    Say "X equals 5."
Otherwise:
    Say "X is not equal to 5."
```

这就是一个分支的例子。根据条件，如果“X=5”，表示为“X 等于 5”；否则，则说“X 不等于 5”。

27.1 使用 if

通过 shell，我们可以对上面的逻辑进行编码，如下所示：

```
x=5

if [ $x = 5 ]; then
    echo "x equals 5."
else
    echo "x does not equal 5."
fi
```

或者可以直接在命令行中输入以上代码（略有简化）。

```
[me@linuxbox ~]$ x=5
[me@linuxbox ~]$ if [ $x = 5 ]; then echo "equals 5"; else echo "does not equal
5"; fi
equals 5
[me@linuxbox ~]$ x=0
[me@linuxbox ~]$ if [ $x = 5 ]; then echo "equals 5"; else echo "does not equal
5"; fi
does not equal 5
```

在这个例子中，我们执行了两次命令。第一次，设置变量 X 为 5，从而输出字符串“equals 5”；第二次，设置变量 X 为 0，从而输出字符串“does not equal 5”。

if 语句的语法格式如下。

```
if commands; then
    commands
[elif commands; then
    commands...]
[else
    commands]
fi
```

在这个语法格式中，“command”可以是一组命令。乍看上去可能会有些迷惑。在去除这个迷惑前，我们必须先解一下 shell 如何判断一个命令的成功与失败的。

27.2 退出状态

命令（包括我们编写的脚本和 shell 函数）在执行完毕后，会向操作系统发送一个值，称之为“退出状态”。这个值是一个 0~255 的整数，用来指示命令执行成功还是失败。按照惯例，数值 0 表示执行成功，其他的数值表示

执行失败。shell 提供了一个可以用来检测退出状态的参数。在下面的例子中可以看到。

```
[me@linuxbox ~]$ ls -d /usr/bin
/usr/bin
[me@linuxbox ~]$ echo $?
0
[me@linuxbox ~]$ ls -d /bin/usr
ls: cannot access /bin/usr: No such file or directory
[me@linuxbox ~]$ echo $?
2
```

在这个例子中，我们两次执行了 `ls` 命令。第一次，命令执行成功，如果显示参数 “`$?`” 的值，可以看到它是 0。第二次执行 `ls` 命令时，产生了一个错误，再次显示参数 “`$?`” 的值，这次则为 2，表示这个命令遇到了一个错误。有些命令使用不同的退出值来诊断错误，而许多命令在执行失败时，只是简单地退出并发送数字 1。`man` 手册中经常会包括一个标题为 “Exit Status”的段落，它描述使用的代码。数字 0 总是表示执行成功。

shell 提供了两个非常简单的内置命令，它们不做任何事情，除了以一个 0 或 1 退出状态来终止执行。“`true`” 命令总是表示执行成功，而 “`false`” 命令总是表示执行失败。

```
[me@linuxbox ~]$ true
[me@linuxbox ~]$ echo $?
0
[me@linuxbox ~]$ false
[me@linuxbox ~]$ echo $?
1
```

我们可以用这两个命令来查看 `if` 语句是如何工作的。`if` 语句真正做的事情是评估命令的成功或失败。

```
[me@linuxbox ~]$ if true; then echo "It's true."; fi
It's true.
[me@linuxbox ~]$ if false; then echo "It's true."; fi
[me@linuxbox ~]$
```

当在 `if` 后面的命令执行成功时，命令 `echo "It's true."` 会被执行，而当在 `if` 后面的命令执行失败时，该命令则不执行。如果在 `if` 后面有一系列的命令，那么则根据最后一个命令的执行结果进行评估。

```
[me@linuxbox ~]$ if false; true; then echo "It's true."; fi
It's true.
[me@linuxbox ~]$ if true; false; then echo "It's true."; fi
[me@linuxbox ~]$
```

27.3 使用 test 命令

目前为止，经常和 if 一起使用的命令是 test。test 命令会执行各种检查和比较。这个命令有两种等价的形式：

`test expression`

以及更流行的：

`[expression]`

这里 expression 是一个表达式，其结果是 true 或 false。当这个表达式为 true 时，test 命令返回一个零退出状态；当表达式为 false 时，test 命令的退出状态为 1。

27.3.1 文件表达式

表 27-1 中的表达式用来评估文件的状态。

表 27-1 测试文件的表达式

表达式	成为 true 的条件
<code>file1 -ef file2</code>	file1 和 file2 拥有相同的信息节点编号（这两个文件通过硬链接指向同一个文件）
<code>file1 -nt file2</code>	file1 比 file2 新
<code>file1 -ot file2</code>	file1 比 file2 旧
<code>-b file</code>	file 存在并且是一个块（设备）文件
<code>-c file</code>	file 存在并且是一个字符（设备）文件
<code>-d file</code>	file 存在并且是一个目录
<code>-e file</code>	file 存在
<code>-f file</code>	file 存在并且是一个普通文件
<code>-g file</code>	file 存在并且设置了组 ID
<code>-G file</code>	file 存在并且属于有效组 ID
<code>-k file</code>	file 存在并且有“粘滞位（sticky bit）”属性
<code>-L file</code>	file 存在并且是一个符号链接
<code>-O file</code>	file 存在并且属于有效用户 ID
<code>-p file</code>	file 存在并且是一个命名管道
<code>-r file</code>	file 存在并且可读（有效用户有可读权限）
<code>-s file</code>	file 存在并且其长度大于 0
<code>-S file</code>	file 存在并且是一个网络套接字

续表

表达式	成为 true 的条件
-t fd	fd 是一个定向到终端/从终端定向的文件描述符，可以用来确定标准输入/输出/错误是否被重定向
-u file	file 存在并且设置了 setuid 位
-w file	file 存在并且可写（有效用户拥有可写权限）
-x file	file 存在并且可执行（有效用户拥有执行/搜索权限）

下面的脚本可用来演示某些文件表达式。

```
#!/bin/bash

# test-file: Evaluate the status of a file

FILE=~/bashrc

if [ -e "$FILE" ]; then
    if [ -f "$FILE" ]; then
        echo "$FILE is a regular file."
    fi
    if [ -d "$FILE" ]; then
        echo "$FILE is a directory."
    fi
    if [ -r "$FILE" ]; then
        echo "$FILE is readable."
    fi
    if [ -w "$FILE" ]; then
        echo "$FILE is writable."
    fi
    if [ -x "$FILE" ]; then
        echo "$FILE is executable/searchable."
    fi
else
    echo "$FILE does not exist"
    exit 1
fi
exit
```

这个脚本会评估赋值给常量 FILE 的文件，并显示评估结果。关于该脚本，需要注意两个有趣的地方。首先，要注意\$FILE 在表达式内是怎样被引用的。尽管引号不是必需的，但是这可以防范参数为空的情况。如果\$FILE 的参数扩展产生一个空值，将导致一个错误（操作符会被解释为非空的字符串，而不是操作符）。用引号把参数括起来可以确保操作符后面总是跟随着一个字符串，即使字符串为空。其次，注意脚本末尾的 exit 命令。这个 exit 命令接受一个单独的可选参数，它将成为脚本的退出状态。当不传递参数时，退出状态默认为 0。以这种方法使用 exit 命令，当\$FILE 扩展为一个不存在的文件名时，可以允许

脚本提示失败。这个 `exit` 命令出现在脚本的最后一行。这样，当脚本执行到最后时，不管怎样，默认情况下它将以退出状态零终止。

类似地，通过在 `return` 命令中包含一个整数参数，shell 函数可以返回一个退出状态。如果要将上面的脚本转换为一个 shell 函数，从而能够在一个更大的程序中使用，可以将 `exit` 命令替换为 `return` 命令，并得到想要的行为。

```
test_file () {
    # test-file: Evaluate the status of a file

    FILE=-/.bashrc

    if [ -e "$FILE" ]; then
        if [ -f "$FILE" ]; then
            echo "$FILE is a regular file."
        fi
        if [ -d "$FILE" ]; then
            echo "$FILE is a directory."
        fi
        if [ -r "$FILE" ]; then
            echo "$FILE is readable."
        fi
        if [ -w "$FILE" ]; then
            echo "$FILE is writable."
        fi
        if [ -x "$FILE" ]; then
            echo "$FILE is executable/searchable."
        fi
    else
        echo "$FILE does not exist"
        return 1
    fi
}
```

27.3.2 字符串表达式

表 27-2 中的表达式用来测试字符串的操作。

表 27-2 测试字符串表达式

表达式	成为 true 的条件
<code>string</code>	<code>string</code> 不为空
<code>-n string</code>	<code>string</code> 的长度大于 0
<code>-z string</code>	<code>string</code> 的长度等于 0
<code>string1==string2</code>	<code>string1</code> 和 <code>string2</code> 相等。单等号和双等号都可以使用，但是双等号使用的更多
<code>string1!=string2</code>	<code>string1</code> 和 <code>string2</code> 不相等

续表

表达式	成为 true 的条件
string1>string2	在排序时，string1 在 string2 之后
string1<string2	在排序时，string1 在 string2 之前

警告

在使用 test 命令时，“>”和“<”运算符必须用引号括起来（或者是使用反斜杠进行转义）。如果不这样做，就会被 shell 解释为重定向操作符，从而造成潜在的破坏性结果。同时注意，尽管 bash 文档中已经声明，排序遵从当前语系的排列规则，但并非如此。在 bash 4.0 版本以前（包括 4.0 版本），使用的是 ASCII（POSIX）排序方式。

下面是一个合并字符串表达式的脚本。

```
#!/bin/bash

# test-string: evaluate the value of a string

ANSWER=maybe

if [ -z "$ANSWER" ]; then
    echo "There is no answer." >&2
    exit 1
fi
if [ "$ANSWER" = "yes" ]; then
    echo "The answer is YES."
elif [ "$ANSWER" = "no" ]; then
    echo "The answer is NO."
elif [ "$ANSWER" = "maybe" ]; then
    echo "The answer is MAYBE."
else
    echo "The answer is UNKNOWN."
fi
```

在这个脚本中，我们评估了常量 ANSWER。我们首先检查了这个字符串是否为空。如果是空，则终止脚本，并且把退出状态设置为 1。注意应用到 echo 命令的重定向操作。它把错误消息“*There is no answer.*”重定向到标准错误，这也是处理错误信息的“合理”方法。如果字符串非空，则评估这个字符串的值是否是“yes”、“no”或者“maybe”。我们使用 elif (else if 的缩写) 来实现上述目的。通过使用 elif，我们可以建立一个更复杂的逻辑测试。

27.3.3 整数表达式

表 27-3 中的表达式用于整数。

表 27-3 整数判断操作

表达式	成为 true 的条件
integer1 -eq integer2	integer1 和 integer2 相等
integer1 -ne integer2	integer1 和 integer2 不相等
integer1 -le integer2	integer1 小于等于 integer2
integer1 -lt integer2	integer1 小于 integer2
integer1 -ge integer2	integer1 大于等于 integer2
integer1 -gt integer2	integer1 大于 integer2

下面是一个演示这些表达式的脚本。

```
#!/bin/bash

# test-integer: evaluate the value of an integer.

INT=-5

if [ -z "$INT" ]; then
    echo "INT is empty." >&2
    exit 1
fi
if [ $INT -eq 0 ]; then
    echo "INT is zero."
else
    if [ $INT -lt 0 ]; then
        echo "INT is negative."
    else
        echo "INT is positive."
    fi
    if [ $((INT % 2)) -eq 0 ]; then
        echo "INT is even."
    else
        echo "INT is odd."
    fi
fi
```

这个脚本中最有意思的地方是，如何判断一个整数是奇数还是偶数。通过用模数 2 对数值进行求模运算，也就是将这个数值除以 2 并且返回余数，这就可以知道这个数值是奇数还是偶数了。

27.4 更现代的 test 命令版本

bash 的最近版本包括了一个符合命令，它相当于增强的 test 命令。下面是这个命令的语法。

[[expression]]

expression 是一个表达式，其结果为 true 或 false。[[]]命令和 test 命令类似（支持所有的表达式），不过增加了一个很重要的新字符串表达式。

string1=~regex

如果 string1 与扩展的正则表达式 regex 匹配，则返回 true。这就为执行数据验证这样的任务提供了许多可能性。在前面整数表达式的例子中，如果常量 INT 含有整数以外的其他值，脚本就会执行失败。脚本需要有一种方法来验证常量包含的是否是整数。可以使用[[]]和=~字符串表达式操作符，按照如下方式改进脚本：

```
#!/bin/bash

# test-integer2: evaluate the value of an integer.

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if [ $INT -eq 0 ]; then
        echo "INT is zero."
    else
        if [ $INT -lt 0 ]; then
            echo "INT is negative."
        else
            echo "INT is positive."
        fi
        if [ $((INT % 2)) -eq 0 ]; then
            echo "INT is even."
        else
            echo "INT is odd."
        fi
    fi
else
    echo "INT is not an integer." >&2
    exit 1
fi
```

通过应用正则表达式，我们可以限制 INT 的值只能是字符串，而且字符串必须以以减号（可选）开头，后面跟着一个或者多个数字。这个表达式同样消除了 INT 为空值的可能性。

[[]]增加的另外一个特性是==操作符支持模式匹配，就像路径名扩展那样。举例如下。

```
[me@linuxbox ~]$ FILE=foo.bar
[me@linuxbox ~]$ if [[ $FILE == foo.* ]]; then
> echo "$FILE matches pattern 'foo.*'"
> fi
foo.bar matches pattern 'foo.*'
```

这使得[[]]在评估文件和路径名的时候非常有用。

27.5 (())——为整数设计

除了[[]]的复合命令之外, bash 同样提供了(())复合命令, 它可用于操作整数。该命令支持一套完整的算术计算, 这部分内容将在第 34 章中介绍。

(())用于执行算术真值测试 (arithmetic truth test)。当算术计算的结果是非零值时, 则算术真值测试为 true。

```
[me@linuxbox ~]$ if ((1)); then echo "It is true."; fi
It is true.
[me@linuxbox ~]$ if ((0)); then echo "It is true."; fi
[me@linuxbox ~]$
```

使用(()), 我们可以简化 test-integer2 脚本, 如下所示。

```
(#!/bin/bash

# test-integer2a: evaluate the value of an integer.

INT=-5

if [[ "$INT" =~ ^-[0-9]+$ ]]; then
    if ((INT == 0)); then
        echo "INT is zero."
    else
        if ((INT < 0)); then
            echo "INT is negative."
        else
            echo "INT is positive."
        fi
        if (((INT % 2)) == 0)); then
            echo "INT is even."
        else
            echo "INT is odd."
        fi
    fi
else
    echo "INT is not an integer." >&2
    exit 1
fi
```

注意这里使用了小于号、大于号和等号用来测试相等性。在处理整数时, 这些语法看起来也更自然。另外, 由于(())复合命令只是 shell 语法的一部分, 而非普通的命令, 并且只能处理整数, 所以它能够通过名字来识别变量, 而且不需要执行扩展操作。

27.6 组合表达式

我们也可以将表达式组合起来，来创建更复杂的计算。表达式是使用逻辑运算符组合起来的。在第 17 章，我们在学习 find 命令的时候已经介绍过。与 test 和 [] 命令配套的逻辑运算符有三个，它们是 AND、OR 和 NOT。test 和 [] 使用不同的操作符来表示这三种逻辑操作，如表 27-4 所示。

表 27-4 逻辑操作符

Operation	test	[]and(())
AND	-a	&&
OR	-o	
NOT	!	!

下面是一个 AND 运算的例子。这个脚本用来检测一个整数是否属于某个范围内的值。

```
#!/bin/bash

# test-integer3: determine if an integer is within a
# specified range of values.

MIN_VAL=1
MAX_VAL=100

INT=50

if [[ "$INT" =~ ^-[0-9]+$ ]]; then
    if [[ INT -ge MIN_VAL && INT -le MAX_VAL ]]; then
        echo "$INT is within $MIN_VAL to $MAX_VAL."
    else
        echo "$INT is out of range."
    fi
else
    echo "INT is not an integer." >&2
    exit 1
fi
```

在这个脚本中，检测整数 INT 的值是否在 MIN_VAL 和 MAX_VAL 之间。这是通过一个[]运算符来执行的，该运算符中包含两个用“&&”运算符分隔来的表达式。当然我们也可以使用 test 完成该功能。

```

if [ $INT -ge $MIN_VAL -a $INT -le $MAX_VAL ]; then
    echo "$INT is within $MIN_VAL to $MAX_VAL."
else
    echo "$INT is out of range."
fi

```

“!”否定运算符对表达式的运算结果取反。如果表达式为 `false`，则返回 `true`；反之，如果表达式为 `true`，则返回 `false`。在下面的脚本中，我们将修改判断逻辑，以判断 INT 的值是否在指定的范围之外：

```

#!/bin/bash

# test-integer4: determine if an integer is outside a
# specified range of values.

MIN_VAL=1
MAX_VAL=100

INT=50

if [[ "$INT" =~ ^-[0-9]+$ ]]; then
    if [[ ! (INT -ge MIN_VAL && INT -le MAX_VAL) ]]; then
        echo "$INT is outside $MIN_VAL to $MAX_VAL."
    else
        echo "$INT is in range."
    fi
else
    echo "INT is not an integer." >&2
    exit 1
fi

```

当然我们也可以使用圆括号把表达式括起来，以进行分组。如果不使用圆括号，“！”运算符仅对第一个表达式有效，而不是对两个组合后的表达式有效。用 `test` 命令可以按照如下进行编码：

```

if [ ! \( $INT -ge $MIN_VAL -a $INT -le $MAX_VAL \) ]; then
    echo "$INT is outside $MIN_VAL to $MAX_VAL."
else
    echo "$INT is in range."
fi

```

由于 `test` 使用的所有表达式和操作符都被 shell 看做命令参数（不像`[[]]`以及`(())`），因此在 `bash` 中有特殊含义的字符，如“<”、“>”、“(” 和 “)”，必须用引号括起来或者进行转义。

`test` 和`[[]]`命令基本上完成相同的功能，那么，哪一个更好呢？`test` 更为传统（而且也是 POSIX 的一部分），而`[[]]`则是 `bash` 特定的。由于 `test` 命令的使用更为广泛，因此直到如何使用 `test` 更为重要。但是`[[]]`命令显然更有用，而且更容易编码。

可移植性是狭隘思想的心魔

如果和“真正的”的 UNIX 人员交流，就会发现他们中的大部分并不是很喜欢 Linux。他们认为 Linux 是不纯洁、不简洁的。UNIX 信徒的一个信条是 Linux 的一切应该都是可以移植的。这意味着你编写的脚本不用修改就可以直接在任何一个类 UNIX 操作系统中运行。

UNIX 人员有足够的理由去相信这些。当在 POSIX 之前，他们看到 UNIX 命令和 shell 的专有扩展对 UNIX 世界的影响之后，自然就会担心 Linux 会对他们喜欢的操作系统产生影响。

但是可移植性有一个很严重的缺陷。它阻碍了进步，它需要使用“最小公分母”技术来完成工作。在 shell 编程中，这就意味着所有的事情要与 sh(最早的 Bourne shell) 兼容。

这种缺陷是专有供应商的一个借口，它们使用该借口为自己的专有命令扩展进行辩解，也只有它们将其称之为“创新”。但它们只是为客户锁定了设备而已。

GNU 工具，比如 bash，并没有上述的限制。它通过支持标准和广泛的可用性来实现可移植性。你可以在几乎所有的操作系统中安装 bash 和其他的 GNU 工具，甚至包括 Windows，并且是免费的。所以，放心地使用 bash 的特性吧。它是真正可移植的。

27.7 控制运算符：另一种方式的分支

bash 还提供了两种可以执行分支的控制运算符。“**&&**”(AND) 和“**||**”(OR) 运算符与[[]]复合命令中的逻辑运算符类似。语法如下。

`command1 && command2`

和

`command1 || command2`

理解这两个运算符是非常重要的。对于“**&&**”运算符来说，先执行 command1，只有在 command1 执行成功时，command2 才能够执行。对于“**||**”运算符来说，先执行 command1，则只有在 command1 执行失败时，command2 才能够执行。

从实用性考虑，这意味着可以这样做：

[me@linuxbox ~]\$ mkdir temp && cd temp

这会创建一个 *temp* 目录，并且当这个创建工作执行成功后，当前的工作目录才会更改为 *temp*。只有在第一个 *mkdir* 命令执行成功后，才会尝试执行第二个命令。同样，看如下命令：

```
[me@linuxbox ~]$ [ -d temp ] || mkdir temp
```

这个命令先检测 *temp* 目录是否存在，只有当检测失败时，才会创建这个目录。这种构造类型可以轻松处理脚本中的错误，后面章节会对其详细讲解。例如，我们可以在一个脚本中这样做：

```
[ -d temp ] || exit 1
```

如果脚本需要目录 *temp*，而这个目录不存在，则这个脚本会终止，并且退出状态为 1。

27.8 本章结尾语

本章以一个问题开始。我们怎样让 *sys_info_page* 脚本检测用户是否具有读取所有主目录的权限呢？通过我们的 if 知识，我们在 *report_home_space* 函数中增加以下代码段，就可以解决这个问题。

```
report_home_space () {
    if [[ $(id -u) -eq 0 ]]; then
        cat <<- _EOF_
        <H2>Home Space Utilization (All Users)</H2>
        <PRE>$(du -sh /home/*)</PRE>
        _EOF_
    else
        cat <<- _EOF_
        <H2>Home Space Utilization ($USER)</H2>
        <PRE>$(du -sh $HOME)</PRE>
        _EOF_
    fi
    return
}
```

我们计算了 *id* 命令的输出。通过使用-u 选项，*id* 命令会输出有效用户的数字用户 ID 编号。而超级用户的编号总是 0，其他用户的编号则是一个大于 0 的数字。知道了这点以后，我们就可以创建两个不同的 *here* 文档，一个使用的是超级用户的权限，而另一个则受限于用户自己的主目录。

有关 *sys_info_page* 程序的内容暂时告一段落。不过不用担心。这些内容会再次出现。同时，当我们继续当前的工作时，还将讨论一些会用得上的主题。

第28章

读取键盘输入

迄今为止，本书所使用的脚本都缺少了一个大多数程序所常见的功能点——交互——也就是程序与用户互动的能力。尽管很多程序不需要与用户交互，但对一些程序来说，直接从用户获取输入会比较好。以前面章节中的脚本为例。

```
#!/bin/bash

# test-integer2: evaluate the value of an integer.

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if [ $INT -eq 0 ]; then
        echo "INT is zero."
    else
        if [ $INT -lt 0 ]; then
            echo "INT is negative."
        else
            echo "INT is positive."
        fi
        if [ $((INT % 2)) -eq 0 ]; then
            echo "INT is even."
        else
```

```

        echo "INT is odd."
    fi
fi
else
    echo "INT is not an integer." >&2
    exit 1
fi

```

每当用户想要改变 INT 变量的值，就必须修改脚本。若脚本能向用户请求输入，就方便多了。本章节将讲解如何向程序添加与用户交互的功能。

28.1 read——从标准输入读取输入值

内嵌命令 `read` 的作用是读取一行标准输入。此命令可用于读取键盘输入值或应用重定向读取文件中的一行。`read` 命令的语法结构如下所示。

`read [-options] [variable...]`

语法中 `options` 为表 28-1 列出的一条或多条可用的选项，而 `variable` 则是一到多个用于存放输入值的变量。若没有提供任何此类变量，则由 `shell` 变量 `REPLY` 来存储数据行。

表 28-1 `read` 选项

选项	描述
<code>-a array</code>	将输入值从索引为 0 的位置开始赋给 <code>array</code> ，本书第 35 章将介绍 <code>array</code> 的相关内容
<code>-d delimiter</code>	用字符串 <code>delimiter</code> 的第一个字符标志输入的结束，而不是新的一行的开始
<code>-e</code>	使用 <code>Readline</code> 处理输入。此命令使用户能使用命令行模式的相同方式编辑输入
<code>-n num</code>	从输入中读取 <code>num</code> 个字符，而不是一整行
<code>-p prompt</code>	使用 <code>prompt</code> 字符串提示用户进行输入
<code>-r</code>	原始模式。不能将后斜线字符翻译为转义码
<code>-s</code>	保密模式。不在屏幕显示输入的字符。此模式在输入密码和其他机密信息时很有用处
<code>-t seconds</code>	超时。在 <code>seconds</code> 秒后结束输入。若输入超时， <code>read</code> 命令返回一个非 0 的退出状态
<code>-u fd</code>	从文件说明符 <code>fd</code> 读取输入，而不是从标准输入中读取

基本上，`read` 命令将标准输入的字段值分别赋给指定的变量。若使用 `read` 命令改写之前的整数验证脚本，可能如下所示。

```

#!/bin/bash

# read-integer: evaluate the value of an integer.

echo -n "Please enter an integer -> "
read int

```

```

if [[ "$int" =~ ^-[0-9]+$ ]]; then
    if [ $int -eq 0 ]; then
        echo "$int is zero."
    else
        if [ $int -lt 0 ]; then
            echo "$int is negative."
        else
            echo "$int is positive."
        fi
        if [ $((int % 2)) -eq 0 ]; then
            echo "$int is even."
        else
            echo "$int is odd."
        fi
    fi
else
    echo "Input value is not an integer." >&2
    exit 1
fi

```

我们使用带有-n 选项（使接下来的输出在下一行显示）的 echo 命令来输出一条提示符，然后使用 read 命令给 int 变量赋值。此脚本的运行结果如下所示。

```
[me@linuxbox ~]$ read-integer
Please enter an integer -> 5
5 is positive.
5 is odd.
```

在下述脚本中，read 命令将输入值赋给多个变量。

```
#!/bin/bash

# read-multiple: read multiple values from keyboard

echo -n "Enter one or more values > "
read var1 var2 var3 var4 var5

echo "var1 = '$var1'"
echo "var2 = '$var2'"
echo "var3 = '$var3'"
echo "var4 = '$var4'"
echo "var5 = '$var5'"
```

此脚本可以给 5 个变量赋值并输入。需要注意当输入少于或多于 5 个值的时候，read 的运作方式如下。

```
[me@linuxbox ~]$ read-multiple
Enter one or more values > a b c d e
var1 = 'a'
var2 = 'b'
var3 = 'c'
var4 = 'd'
var5 = 'e'
[me@linuxbox ~]$ read-multiple
Enter one or more values > a
```

```

var1 = 'a'
var2 =
var3 =
var4 =
var5 =
[me@linuxbox ~]$ read-multiple
Enter one or more values > a b c d e f g
var1 = 'a'
var2 = 'b'
var3 = 'c'
var4 = 'd'
var5 = 'e f g'

```

若 read 命令读取的值少于预期的数目，则多余的变量值为空，而输入值的数目超出预期的结果时，最后的变量包含了所有的多余值。

如果 read 命令之后没有变量，则会为所有的输入分配一个 shell 变量：REPLY。

```

#!/bin/bash

# read-single: read multiple values into default variable

echo -n "Enter one or more values > "
read

echo "REPLY = '$REPLY'"

```

以上脚本的运行结果如下所示。

```

[me@linuxbox ~]$ read-single
Enter one or more values > a b c d
REPLY = 'a b c d'

```

28.1.1 选项

read 命令支持的选项如前面的表 28-1 所示。

使用不同的选项，read 命令可以达成不同的有趣的效果。例如，可使用-p 选项来显示提示符：

```

#!/bin/bash

# read-single: read multiple values into default variable

read -p "Enter one or more values > "

echo "REPLY = '$REPLY'"

```

使用-t 与-s 选项，可以写出读取“秘密”输入的脚本，此脚本若一定时间内没有完成输入，会造成超时。

```

#!/bin/bash

# read-secret: input a secret passphrase

```