sed & awk

第二版

RHEL饭醉团伙

无码高清版

sed 与 awk

修订第三版

Dale Dougherty & Arnold Robbins 著

张旭东 杨作梅 田丽华 等译

O'REILLY®

机械工业出版社 China Machine Press

HZ BOOKS

运维技术书籍免费下载 https://www.linuxprobe.com

免责声明

1. 本人整理此书完全来来自于网络，本人只复制和粘贴。此书是本人学习练习打字后结果，和本人的政治和法律立场无关。

2. 本人整理此书无意对版权所有者或者任何一名读者、组织、机构、国家、星球、星系造成任何伤害，如果任何读者、组织、机构、国家、星球、星系对本人整理的书有不良反应，导致恶心、呕吐、气喘，胸闷，头晕、发热、火山暴发、地震等症状，请迅速关闭屏幕，拨打 120 求救。有心脏病、高血压等疾病的任何个人、组织、机构、国家、星球、星系，请谨慎选择是否阅读本书，一切因阅读本书造成的后果，本人概不负责。

3. 因为本人不知道此书是否合法，请勿随意复制、转贴、抄录，拍摄本书内容，更不要邀请朋友一起阅读本书，看过此书后，必须立即忘记，网管如果认为此书内容非法，请立即删除。

4. 本人所打文字如果涉及违反任何星球任何国家任何地区的任何法律、法规、版权、规章、制度、惯例、习俗等，恳请阅读人员及时予以删除。如因删除不及时产生了任何无论正面的负面的前面的后面上面的下面的影响本人均不承担责任。如要跨时空跨星球跨国跨省跨市跨县跨乡追捕请勿将本人列入名单之内。

感谢以下朋友参与复制粘贴：

PingSun	QQ：466109309	（整理完准备睡觉时天亮了）
linux--root#	QQ：422394243	（每天熬到 3、4 点的超人）
mickey.li	QQ：740424619	（表现的很积极的参与）
cuttlefish	QQ：389740256	（出差还不忘整理书）
飞机片	QQ：50288401	（半夜 4 点搞定发的邮件）
曾经	QQ：526230665	（差一点耽搁了培训）
小青椒	QQ：88602273	（考试前一天还在帮整理）
睡 zZ	QQ：363991837	（整理的很细心）
lzt.wxhkk	QQ：287569564	（很热心半夜帮人家整理）

如发现书中的错误可以联系本人：
邮箱：lulin2008@qq.com;
QQ：　304583943（蓝血攻击者）
QQ 交流群：50192241（rhel 饭醉团伙）

本书主要参考文献：sed 与 awk（第二版）/(美)多尔蒂，（美）罗宾斯 著
张旭乐等译　　北京：机械工业出版社，2003.6

运维技术书籍免费下载　https://www.linuxprobe.com

目录

前言..... 6

第一章 强大的编辑工具.....16
解决有趣的问题.....16
字符流编辑器.....18
模式匹配的程序设计语言.....19
掌握 sed 和 awk 的 4 个障碍.....22

第二章 了解基本操作.....23
awk 起源于 sed 和 grep 而不是 ed.....24
命令行的语法.....28
使用 sed.....29
使用 awd.....33
同时使用 sed 和 awk.....37

第三章 了解正则表达式语法.....40
表达式.....41
成排的字符.....43
使用喜欢的元字符.....67

第四章 编写 sed 脚本.....68
在脚本中应用命令.....68
寻址上的全局透视.....71
测试并保存输出.....73
sed 脚本的 4 种类型.....76
开始 Promised Land.....89

第五章 基本 sed 命令.....90
sed 命令的语法.....90
注释.....91
替换.....92
删除.....99
追加 插入和更改.....99

运维技术书籍免费下载 https://www.linuxprobe.com

列表.....102
转换.....105
打印.....106
打印行号.....107
下一步.....108
读和写文件.....109
退出.....117

第六章　高级 sed 命令.....119
多行模式空间.....119
学习案例.....128
包含那一行.....131
高级的流控制命令.....139
加入一个短语.....145

第七章　编写 awk 脚本.....148
遵守规则.....149
Hello,World.....149
awk 程序设计模型.....150
记录和字段.....153
表达式.....157
系统变量.....162
关系操作符和布尔操作符.....167
格式打印.....174
向脚本传递参数.....177
信息的检索.....179

第八章　条件、循环和数组.....183
条件语句.....183
循环.....198
影响流控制的其他语句.....191
数组.....192
首字母缩写词处理器.....204
作为系统变量的数组.....213

第九章　函数.....214
算术函数.....214
字符串函数.....219
自定义函数.....228

运维技术书籍免费下载　https://www.linuxprobe.com

第十章 “底部抽屉”.....254
getline 函数.....254
close()函数.....259
system()函数.....260
基于菜单的命令生成器.....262
直接向文件和管道输出.....266
生成柱状报告.....270
调试.....273
约束.....278
使用#!语法调用 awk.....279

第十一章 awk 的系列产品.....282
原始的 awk.....282
可免费使用的 awk.....285
商业版的 awk.....300
后记.....304

第十二章 综合应用.....305
一个交互式拼写检查器.....305
生成格式化索引.....327
Masterindex 程序的其他细节.....361

第十三章 脚本的汇总.....368
uutot.awk----uucp 的统计报告.....368
phonebill----跟踪电话的使用情况.....372
combile----抽取多部分用 uuencoded 编码技术处理的二进制代码.....375
mailavg----检查邮箱的大小.....377
adj----调整文本文件的行.....378
readsource----将程序源文件格式化为 troff 格式.....386
gent----获得 termcap 条目.....393
plpr----行式打印的预处理器.....395
transpose----实现矩阵的转置.....399
m1----简单的宏处理器.....400

附录一 sed 的快速参考.....409
附录二 awk 的快速参考.....414
附录三 第十二章的补充.....428

运维技术书籍免费下载 https://www.linuxprobe.com

前言

本书介绍了一组名字奇特的 UNIX 实用工具:sed 和 awk。这组实用工具有很多共同的特征，譬如正则表达式在模式匹配在 sed 和 awk 的使用中是很重要的部分，因此本书详尽地解释了 UNIX 正则表达式的语法。一般情况下，从 grep 到 sed 和 awk 的学习过程是很自然的，所以本书涵盖了上述 3 个程序，而重点集中在 sed 和 awk。

sed 和 awk 是一般用户，程序员和系统管理员们处理文本文件的有力工具。sed 的名字来源于其功能，它是一个字符流编辑器（stream editor），可以很好地完成对多个文件的一系列编辑工作。Awk 的名字来源于它的开发人 Aho，Weinberger 和 Kernighan，这是一程序设计语言，非常适合结构化数据的处理和格式化报表的生成。本书强调了 awk 和 POSIX 定义。另外，在讨论的 3 个可以免费获得的版本和 2 个商业版本以前，本书还简要地描述了 awk 的最初版本，所有的这些版本都实现了 awk 的 POSIX 定义。

本书的重点是编写 sed 和 awk 脚本来快速解决用户各种各样的问题。大多数脚本都可以称为“快速定位”。另外，我们还会涉及到一些需要更仔细地设计和开发，能够解决较大问题的脚本。

本书内容

第一章 “强大的编辑工具”，是对 sed 和 awk 的特选和功能的概括性描述。

第二章 “了解基本操作”，论述了 sed 和 awk 的基本操作，并展示了从 sed 到 awk 的功能方面的进步。二者共有相似的命令行语法，以脚本的形式接受用户指令。

第三章 “了解正则表达式语法”，非常详细地描述了 UNIX 正则表达式语法。通常，新用户会对这些用于模式匹配的奇怪表达式感到无所适从。掌握正则表达式语法是很重要的，这可以从 sed 和 awk 中得到更多的东西。本章中模式匹配的例子主要依赖于 grep 和 egrep。

第四章 “编写 sed 脚本”，从本章开始，用 3 章的篇幅对 sed 进行介绍。本章介绍了那些只使用几个 sed 命令编写简单的 sed 脚本的期末要素。还给出了一个可以简化 sed 脚本调用的脚本。

第五章 “基本 sed 命令”和第六章 “高级 sed 命令”，将 sed 命令分成基本的和高级的命令。基本命令类似于手工编辑命令，而高级命令则介绍简单的编程功能。高级命令包含对保留空间（一个预留的临时缓冲区）的处理命令。

第七章“编写 sed 脚本”，从本章开始，用 5 章的篇幅对 awk 进行介绍。本章

运维技术书籍免费下载 https://www.linuxprobe.com

介绍了这个脚本化语言的主要特征。介绍了许多脚本，其中包含修改 ls 命令输出结果的脚本。

第八章“条件，循环和数组”，描述了如何使用普通的程序设计结构，例如条件，循环和数组。

第九章 “函数”，描述了如何使用 awk 的内置函数以及如何编写用户定义的函数。

第十章 “底部抽屉”，概述了一组不同性质的 awk 主题。其中包括：如何从 awk 脚本中执行 UNIX 命令，如何将输出定向到文件的管道。另外，本章还提供了几个调试 awk 脚本方面的建议。

第十一章 “awk 系列产品”，描述了 awk 最初的 V7 版本，当前的贝尔实验室的版本，来自自由软件联盟的 GNU awk （gawk），以及 Michael Brennan 编写的 mawk 等。后面三者都有可以自由获取的源代码。本章还描述了两个商业实现，MKS awk 和 Thomson Automation awk（tawk），以及将类似 awk 的功能带到 Visual Basic 环境的 VSAwk。

第十二章 “综合应用”，给出了两个较长，更复杂的 awk 脚本，它们共同印证了这种语言的几乎所有特征。第一个脚本是交互式拼写检查程序。第二个脚本则处理和格式化一本书的索引或一套书的主索引。

第十三章 “脚本的汇总”，给出了用户提供的许多脚本，展示了编写 sed 和 awk 脚本的不同的风格和技术。

附录一 “sed 的快速参考”，是描述的命令和命令行选项的快速参考。
附录二 “awk 的快速参考”，是 awk 的命令行选项和它脚本语言完整描述的快速参考。
附录三 ”第十二章的补充“，给出了第十二章描述的 spellcheck.awk 脚本和 masterindex shell 脚本的完整清单。

Sed 和 awk 的实用性

sed 和 awk 是 Version 7 UNIX（也称 V7 或第 7 版）的一部分。从那里起它们就成为标准发布的一部分。Sed 自从被提出以来就没改动过。

自由软件联盟 GNU 项目的 sed 版本是可以自由获取的（从技术上讲虽然没有放在公共域中）。GNU sed 的源代码可以通过匿名的 FTP（注 1）从 ftp.gnu.ai.mit.edu 上得到。它存在于文件/pub/gnu/sed-2.05.tar.gz 中。这是利用 gzip 程序压缩的 tar 文件，gzip 的源代码可以在相同的目录下得到。全球有许多站点对主 GNU 发布站点的文件做了“镜像”；如果你知道离你最近

运维技术书籍免费下载 https://www.linuxprobe.com

的站点，就可以从那得到这些文件。注意要使用“binary”或”image”模式传送这些文件。

1985 年，awk 的作者对 awk 做了补充，添加了许多有用的特征。可惜的是，几年以来这个版本一直只存在于 AT&T 系统中。从 Release 3.1 开始它成为 UNIX SystemV 的一部分。新的 awk 名为 nawk，旧版本仍然保留原来的名字。SystemV Release 4 系统也是这样。

对商业系统（例如来自 Hewlett-Packard,sun,IBM,Digital 和其他的系统）来说，命名情况变得理复杂了。所有这些系统都有一些旧的和新的 awk 版本，但是每个厂商为程序的命名都不同。有的为 oawk 和 awk，有的为 awk 和 nawk。

注 1： 如果不能访问 Internet 并且还希望得到 GNU sed 副本，请联系 Free software Foundation,inc .，59 Temple Place,Suite 330,Boston,MA 02111-1307 U.S.A。电话号码是 1-617-542-5942，传真号是 1-617-542-2652。

我们能够提供的最好的建议就是检查本地文档（注 2）。在本书中，使用术语 awk 来描述 POSIX awk，特殊的实现则通过名字来引用，例如“gawk”或“Bell Labs awk”。

第十一章讨论了 3 个可自由获取的 awk（包括从什么地方可以得到它们）以及几个商业版本。

注意 自从本书的第一版以来，awk 语言已经被标准化为 POSIX Command Language 和 utilities Standard (P1003.2) 的一部分。所有现代的 awk 实现都向上与 POSIX 兼容。P1003.2 标准混合了来自新的 awk 和 gawk 的特征。在本书中，可以认为，对 POSIX awk 的一个实现是对的东西对另一个实现也是对的，除非注明是特别的版本。

DOS 版本

gawk，mawk 和 GNU sed 已经移植到 DOS 系统。在主 GNU 发布站点上有这些程序的 DOS 版本的文件。另外，gawk 已经移植到 OS/2,VMS 和 Atari 与 Amiga 微型计算机系统中，移植到其他系统（Macintosh, Windows）的工作也正在进行。

egrep，sed 和 awk 可以作为 MKS 工具包（Mortice Kern Systems,Inc.,Ontario,Canada）的一部分用于基于 MS—DOS 的机器。它们的 awk 实现支持 POSIX awk 的特征。

MKS 工具包还包括 Korn shell,这意味着为 UNIX 系统上的 Bourne shell 编写的许多 shell 脚本都可以在 PC 上运行。而 MKS 工具包的大多数用户可能已经发现了这些 UNIX 中的工具，我们希望这些程序的好处对那些没有大胆地进入 UNIX 的 PC 用户来说也是显而易见的。

运维技术书籍免费下载 https://www.linuxprobe.com

Thompson Automation Software(注 3)有一个用于 UNIX ，DOS 和 Microsoft Windows 的 awk 编译器，这个版本很有意思，它拥有 awk 编写的 awk 的一些扩展，还包括一个用 awk 编写的 awk 的调试程序。

有时我们也使用 PC，因为 Ventura Publisher 是一个非常大的格式化软件包。我们

注 2 ： 纯化论者将新的 awk 简单地称为“awk”，这个新的 awk 打算取代最初的 awk。可是，自发布以来几乎已经 10 年了，却还是没有取代。

注 3: 5616 SW Jefferson,Portland,OR 97221 U.S.A,在美国电话为 1-800-944-0139.，在其他地方电话为 1-503-224-1639。

我喜欢它的原因之一是可以连续使用 vi 创建和编辑文本文件，并使用 sed 编写用于编辑工作的脚本。我们曾使用 sed 编写转换程序，从而将 troff 宏转换成 Ventura 样式表标签。我们还利用它 在批处理方式下插入标签。这可以省去必须手工为文件中的重复元素加标签的麻烦。

Sed 和 awk 对于编写处理不同的文件格式的转换程序也非常有用。

Sed 和 awk 的其他信息源

长时间以来，这些实用工具的主要信息源是包含在 UNIX Programmer's Guide 第 2 卷中的两篇文章。文章 awk——A Pattern Scanning and Processing Language（1978 年 9 月 1 日）是由 awk 的 3 个作者编写的。在这 10 页中，它提供了一个简要的指南并且讨论了几个设计和实现的问题。文章 SED——A Non-Interactive Text Editor（1978 年 8 月 15 日）由 Lee E.McMahon 编写。它是一个参考，给出了每个功能的完整描述，并且包含一些很有用的示例（使用 Coleridg 的 Xanadu 作为示例输入）。

在商业书籍中，sed 和 awk 的最重要的处理出现在由 Brian W.Kernighan 和 Rob Pike 合著《The UNIX Programming Enviroment》(Prentice-Hall,1984)中。标题为“Filters”的章节不公解释了这些程序如何工作，而且还展示了它们如何一起来构建有用的应用程序。

awk 的作者合著了一本描述其增强版本的书：《The AWK Programming Language》(Addison-Wesley,1988)。它包含了许多完整的例子并且论述了可以应用 awk 的广泛领域。它遵从《The UNIX Programming Enviroment》的风格，有时对于新用户来说太难了。书中示例程序的源代码可以在 netlib.bell-labs.com 的 netlib/research/awkbookcode 目录下找到。

信息与技术 POSIX （Portable Operating System Interface，可移植的操作

运维技术书籍免费下载 https://www.linuxprobe.com

系统接口）的 IEEE 标准第 2 部分；Shell 和 Utility(标准 1003.2-1992)（注 4）描述了 sed 和 awk（注 5）。这是针对基于 sed 和 awk 编写的可移植 shell 程序所能提供的功能特

注 4: 据说那快了 3 倍！

注 5: 这个标准不能在线获取。它可以向 IEEE 定购，在美国和加拿大的电话为 1-800-678-IEEE（4333），在其他地方的电话为 1-908- 981-0060，或者在 Web 浏览器中浏览 http://www.ieee.org/。费 用为 U.S.$228，包括标准 1003.2-在 1994——Amendment 1 for Batch Environment 。 IEEE 会员和/或 IEEE 协会享受折扣。

征的“官方”描述。因为 awk 本身就是一种程序设计语言，因此它同样是可移植 awk 程序的官方描述。

1996 年，自由软件联盟出版了由 Arnold Robbins 编著的《The GNU Awk User's Guide》。这是 gawk 的文档文件，相比 Aho，Kernighan 和 weinberger 的书，它采用了更多实例教学的方式。它有两个完整的章节完全是示例，并且涵盖了 POSIX awk。该书还由 SSC 以书名《Effective AWK Programming》出版，而且该书的 Texinfo 源自于 gawk 的发布。

当前 GNU 版的 sed 所存在的最大不足就缺乏相应的文档，甚至连一页帮助页（manpage）都没有。

在对 UNIX 的大多数一般性介绍中，对一大串实用工具的介绍时都会提到 sed 和 awk。这些书中，Henry McGilton 和 Rachel Morgan 的《Introducing the UNIX System》还提供了基本编辑技巧的最佳处理，包括所有 UNIX 文本编辑器的使用。

由本书的原作者和 Tim O'Reilly 合著的《UNIX Text Processing》(Hayden Books,1987)一书完整地概述了 sed 和 awk（虽然没有介绍 awk 的新版本）。那本书的读者会在本书中发现一些重复的部分，但是从总体上讲这里采用了不同的方法，但在本书中我们将 sed 和 awk 区别对待，在假设只有高级用户才会使用 awk 工具的情况下，这里我们尽量给出与这两者彼此相关的程序。这些不同的工具，可以独立使用也可以相互配合，为文本处理提供令人兴奋的强大功能。

最后，在 1995 年 Usenet 新闻组 comp.lang.awk 形成了。如果你在前面提到的书籍中没有找到自己需要了解的知识，你可以在新闻组张贴问题，这是一个可能获得他人帮助的好机会。

这个新闻组会定期张贴一篇 “常见问题解答（FAQ）”的文章。除了回答有关 awk 的问题以外，FAQ 还列出了许多站点，从那些站点可以获得用于不同系统的

运维技术书籍免费下载 https://www.linuxprobe.com

不同的 awk 版本的二进制程序。你可以通过 FTP 从主机 rtfm.mit.edu 的 /pub/usenet/com.lang.awk/faq 文件中检索到 FAQ。

示例程序

本书中的示例程序最初是在运行 A/UX 2.0 (UNIX System V Release 2)的 MacHci 和运行 SunOS4.0 的 SparcStation 1 上编写和测试的。要求 POSIX awk 的程序使用 gawk 3.0.0 和来自 Bell Labs FTP 站点的 Bell Labs awk 的 August 1994 版本进行了重新测试（参看第十一章有关 FTP 的详细内容）。Sed 程序用 SunOS 4.1.3 sed 和 GNU sed 2.05 进行了重新测试。

获取示例源代码

可以通过从 O'Reilly & Associates 的 Internet 服务器上获得本书中程序的源代码。本书的示例程序可以用多种电子方式获得：FTP，Ftpmail，BITFTP 和 UUCP。最先列出的是最便宜，最快速和最容易的方式。如果你从上至下读取，第一个为你工作的可能是最好的。如果你直接从 Internet 相连就使用 FTP。如果你没有连接到 Internet 上，但是可以向 Internet 站点（包括 CompuServe 用户）发送和接收电子邮件，那就使用 Ftpmail。如果你能够通过 BITNET 发送电子邮件刚使用 BITFTP。如果上面的方式都不能工作就使用 UUCP。

FTP

为了使用 FTP，需要一台可以直接访问 Internet 的机器。以下是一段示例，黑体字是你应该键入的：

```
$ ftp ftp.oreilly.com
Connected to ftp.oreilly.com.
220 FTP server (Version 6.21 Tue Mar 10 22:09:55 EST 1992) ready.
Name (ftp.oreilly.com:yourname): anonymous
331 Guest login ok, send domain style e-mail address as password.
Password: yourname@domain.name (Use your user name and host here)
230 Guest login ok, access restrictions apply.
ftp> cd /published/oreilly/nutshell/sedawk_2
250 CWD command successful.
ftp> binary (Very important! You must specify binary transfer
for compressed files.)
200 Type set to I.
ftp> get progs.tar.gz
200 PORT command successful.
150 Opening BINARY mode data connection for progs.tar.gz.
226 Transfer complete.
ftp> quit
221 Goodbye.
```


运维技术书籍免费下载 https://www.linuxprobe.com

这个文件是gzip压缩的tar档案文件，通过键入下面的语句从档案文件中提取这些文件：

```
$ gzcat progs.tar.gz | tar xvf -
```

System V系统要求使用下面的tar命令：

```
$ gzcat progs.tar.gz | tar xof -
```

如果gzcat在你的系统上不可用，就分别使用gunzip和tar命令：

```
$ gunzip progs.tar.gz
$ tar xvf progs.tar
```

Ftpmail

Ftpmail 是一个任何可以向 Internet 发送或接收邮件的人都可以访问的邮件服务器。
下面介绍如何操作。要向 ftpmail@online.oreilly.com 发送邮件，在消息体中，给出想要运行的 FTP 命令。服务器将运行匿名的 FTP 并将这些文件邮寄给你。为了得到完整的帮助文件，可以发送一封没有主体并且主体中只有一个单词“help”的邮件。下面给出的是一个邮件对话示例，这可以使你得到这个例子。该命令发送给你所选目录中的一个文件列表和请求的示例文件。如果这个列表中有你感兴趣的示例的较新版本，那么这个列表是很有用的。

```
$ mail ftpmail@online.oreilly.com
Subject:
reply-to yourname@domain.name (Wher e you want files mailed)
open
cd /published/oreilly/nutshell/sedawk_2
dir
mode binary
uuencode
get progs.tar.gz
quit
.
```

出现在“quit”后面的邮件结尾处的签名是可接受的。

BITFTP

BITFTP是BITNET用户的邮件服务器。你可以向它发送请求文件的电子邮件，而

运维技术书籍免费下载 https://www.linuxprobe.com

它通过电子邮件为你送回这些文件。BITFTP目前只为那些直接在BITNET，向BITFTP@PUCC发送邮件要包含你的ftp命令。要得到完整的帮助文件，将HELP做为消息体发送。
下面是发送给BITFTP的消息体：

```
FTP ftp.oreilly.com NETDATA
USER anonymous
PASS yourname@yourhost.edu Put your Internet email address here
(not your BITNET address)
CD /published/oreilly/nutshell/sedawk_2
DIR
BINARY
GET progs.tar.gz
QUIT
```

一旦你得到了想要的文件，按照FTP中的指示将档案文件中的文件解压到相应的目录中。因为你可能不在UNIX系统上，所以你需要为系统获得uudecode,gunzip,atob和tar的版本。VMS，DOS和Mac版本是可用的。

UUCP

UUCP事实上是所有UNIX系统上的标准，而且适用于IBM兼容的PC和Apple Macintosh。这个示例可以由UUCP通过调制解调器从UUNET中获得，UUNET根据连接时间收费。如果你或你的公司有一个UUNET的账号，那么你就有一个在某处通过UUCP连接到UUNET的系统。找到这个系统并且键入：

```
uucp uunet\!~/published/oreilly/nutshell/sedawk_2/progs.tar.gz
yourhost\!~/yourname/
```

如果你使用Bourne样式的shell（sh,ksh,bash,zsh,pdksh）代替csh，那么可以省略反斜杠。一段时间以后（一天或更多的时间）这个文件应该出现在目录/usr/spool/uucppublic/yourname中。如果你没有帐号，但却想要一个以便 你能够得到电子邮件，通过电话703-206-5400和UUNET联系。
/published/oreilly/ls-lR.Z中包含当前所有文件的名称和长度，用它作为一个简要的测试文件是个不错的主意。一旦你得到了目标文件，按照FTP中的指示从档案文件中将这些文件解压缩。

排版约定

本书使用下列排版约定：

粗体 （BOLD）
用于语句，函数，标识符和程序名。

斜体 （Italic）

运维技术书籍免费下载 https://www.linuxprobe.com

用于出现在段落中的文件和目录名以及数据类型；当引入新的术语和概念时，用于强调。

等宽字体（Constant width）

在示例中用于表示文件的内容或来自命令的输出。

等宽字体粗体（Constant Bold）

在示例中用来表示用户应逐字键入的命令行和选项（例如，rm foo表示正确在键入“rm foo”， 就和它出现在文本或示例中的形式一样）。

“” 在解释性的文本中用来标识代码段。系统消息和符号也用引号括起来。

$ 是UNIX Bourne shell或Korn shell提示符。

[] 在程序语法的描述中包围可选的元素（括号本身不用键入，除非另外提到）。

... 代表文本（通常是计算机输出），这是为了清楚或节省空间而省略的文本。

口 标识一个字面空格，这个符号用于示例和文本中以可见的形式表示空格。

• 标识一个字面TAB字符。这个符号用于示例和文本中，以可见的形式表示制表符。

符号CTRL－X或^X标识控制字符的使用。它表示当键入字符“X”的同时按住“CONTROL”键。我们还指示了其他类似的键（例如，RETURN表示回车）。所有的命令行示例后面都跟有RETURN，除非另外说明。

关于第二版

自从1990年本书第一次出版以来，它就成为O'Reilly & Associates “”坚果”系列中最基础的一本书。在编著这本书后发生了3件重要的事情。第一件事是sed的POSIX标准的发布，而且更重要的是awk的POSIX标准的发布。第二件事（可能应归于第一件事）是一些awk版本或其他新的awk版本在所有的现代UNIX系统（商业系统和可免费获取的类似UNIX系统，例如NetBSD,FreeBSD和Linux等 ）上的广泛使用。第三件事是GNU sed和3个awk版本的源代码可用性，而不仅仅是gawk。

因为这些和其他原因，O'Reilly & Associates认为这本书需要修订。的目的是保持这本书特色的完整性，围绕POSIX awk重新定位本书的awk部分，更正错误并使本书跟上时代。

我要对O'Reilly & Associates的Gigi Estabrook, Chris Reilly 和lenny Muellner所提供的帮助表示感谢，对Marc Vauclair（第一版的法语翻译)提供的许多有益的建议，以及John Dzubera的对于第一版的译注表示感谢。Michael Brennan ,Henry Specer 和 Ozan Yigit 充当了这个版本的技术审校，感谢他们的付出。特别要感谢Ozan Yigit 强制我一丝不苟地进行测试。Thompson

运维技术书籍免费下载 https://www.linuxprobe.com

Automation Software的Pat Thompson对本书的修订恰如其分地提供了tawk的评估拷贝。Videosoft的Richard Montgomery为我提供了关于VSAwk的信息。

下面这些人提供了第十三章中的脚本：Jon L.Bentley,Tom Christiansen,Geoff Clare,Roger A.Cornelius,Rahul Dhesi,Nick Holloway, Norman Joseph,Wes Morgan,Tom Van Raalte 和martin Weitzel。他们的贡献是公认的。

还要感谢O'Reilly & Associates的职员。Nicole Gipson Arigo是产品编辑和项目经理。David Sewell是技术编辑，而Chairemarie Fisher O'Leary 负责本书的校对。Jane Ellin和Sheryl Avruch执行质量控制检查。Seth maislin编写索引。Erik Ray,Ellen Siever和Lenny Muellner利用工具创建本书。Chris Reillery调整插图。Nancy Priest和Mary Jane Walsh设计中版式，Edie Freedman设计封面。

我的姻亲，西雅图的Marshall和Elaine Hartbolz特别值得感谢，因为他们是这一个星期以来带着我们的孩子进行野营生活，从而让我在更新的重要阶段取得重大的进步。

Ar nold Robbins

第一版的致谢

说这本书是期待已久的并不过分。1987 年春夏，我在UNIX/World上发表了 3 篇有关awk的文章，声称这些文章来自即将出版的“坚果”系统，即《sed & awk 》。然而说这些话说得太早了。我曾向Tim O'Reilly 提议我将改编这些文章并整理成一本书，并且计划在我的儿子Benjamin出生之后就可以在家里开始工作。我认为可以在几个月的时间之内完成它。然而，我的儿子都 3 岁了，我才完成第一份草稿。Cathy Brennan客户服务代表自从UNIX/World文章发表以来，就一直耐心地处理对这本书的需求自信。Cathy说她甚至让人电话订购本书，并且发誓说这本书已经出版了，因为他们知道已经有人读过它了。我应感谢她和她的职员以及那些由于我而一直在等待的读者。

感谢Tim O'Reilly创办了一个伟大的公司，在那里人们可以很容易地陶醉在有大量有趣的工程中。作为一个编辑，他促使我这本书，但是不允许不经过他的修改而完成它。照例，他的建议促使我努力完善这本书。

感谢O'Reilly & Associates的所有作者和产品编辑，他们介绍了可以利用sed和awk解决的有趣的问题。感谢Ellie Cutler,他是本书的产品编辑同时编写了索引。感谢Lenny Muellner允许我在整本书中提到他。还要感谢Sue Willing和Donna Woonteiler，是由于他们的努力使这本书进入印刷过程。感谢Chris Reilly,他制作了插图。感谢第十三章中的脚本的个人贡献者。还要感谢 Kevin C.Castner, Tim Irvin, Mark Schalz, Alex Humez, Glenn Saito, Geoff

运维技术书籍免费下载 https://www.linuxprobe.com

Hagel, Tony Hurson,
Jerry Peek, Mike Tiller, 和 Lenny Muellner
，他们给我发邮件指出排印错误和其他错误。最后，最值得感谢的是Nancy和Katie,Ben 和gleda.

建议与评论

本书的内容都经过测试，尽管我们做了最大的努力，但错误和疏忽仍然是在所难免的。如果你发现有什么错误，或是针对将来的版本有什么建议，请通过下面的地址告诉我们：

美国：
O’Reilly & Associates, Inc.
101 Morris Street
Sebastopol, CA 95472
1-800-998-9938 (in the U.S. or Canada)
1-707-829-0515 (international or local)
1-707-829-0104 (FAX)
中国：
10080 北京市海淀区知春路 49 号希格玛公寓B座 809 室
奥莱理软件（北京）有限公司

运维技术书籍免费下载 https://www.linuxprobe.com

本章内容

- 解决有趣的问题
- 字符流编辑器
- 匹配的程序设计语言
- 掌握 sed 和 awk 的四个障碍

第一章

强大的编辑工具

我的妻子不让我买动力锯，她怕我用时会发生意外。所以我依靠手锯进行类似搭架子这样的周末工程。然而，如果我从事木工工作，那么我就必须使用动力锯。动力锯的速度和效率对多产是很重要的。[D.D.]

对于创建和修改文本文件的人来说，sed 和 awk 是编辑的动力工具。这些工作中所做的大部分事情都可以用文本编辑器交互式地完成。然而，使用 sed 和 awk 在达到同样结果的情况下可以节省大量重复性工作。

sed 和 awk 是特殊的，它们需要花时间来学习，但是它们的功能将带给你事半功倍的效果，尤其是当文本编辑是你的职业的正常组成部分时。

解决有趣的问题

学习 sed 和 awk 的主要动机是因为，在解决文本编辑的一般问题时它们很有用（注 1）。有些人（包括我自己）对某个问题是否乐于去解决，取决于该问题是否单调乏味。如果让我选择使用 vi 还是 sed 来对大量文件进行一系列重复性编辑工作，我选择 sed，因为对我来说，它可以使这些问题更加有趣。我总结了一个解决方案以取代

注 1： 我猜想本节标题体现了中国古代的咒语“你们生活在有趣的时代”和 Tim O’Rei 曾经对我说过的话——如果人们发现一个问题有趣就会去解决它。[D.D.]

重复一系列按键。除此之外，一旦完成了任务，我会为自己的聪明而自豪。好像有了一点魔法一样，省去了一些枯燥的劳动。

最初，使用 sed 和 awk 来完成一项任务看起来像是需要很长的时间。在尝试几次之后，你可能还会觉得用手工完成任务还更容易些。耐心点！你不但需要学习如何使用 sed 和 awk，而且还要了解在什么情况下使用它们更好。你越精通，解决问题就越快而且与也越广。

运维技术书籍免费下载 https://www.linuxprobe.com

你还有机会发现解决特俗问题的一般方法。你会发现，考虑某种问题的方式如果与一类问题相关，这样就能够设计出在其他情况下可以重用的解决方案。

我们给出了一个示例（没有给出任何程序代码）。在我们写的书中有一本实用了交叉引用命名模式，即引用由格式化软件（sqtroff）定义和处理。在文本文件中，对有关错误处理一章的引用可能是按如下方式编码的

*[CHerrorhand]

“CHerrorhand”是引用的名字，“*[” 和“]”是区分引用和其他文本的调用序列。在中心文件中，文档中用于交叉引用的名字被定义为 sqtroff 字符串。例如，“CHerrorhand”被定义为“Chapter 16,Error Handling”(像这样使用符号交叉引用模式而不使用显式引用的优点是，如果添加、删除或重新排序文章，只需要编辑中心文件就可以反映新的组织结构)。当用格式化软件处理文档时，引用呗正确的重新解析和扩展。

我们面临的问题时必须使用同样的文件来创建本书的在线版本。因为不能使用 sqtroff 格式化软件，所以需要利用一些方式来扩展文件中的交叉引用。话句话说，就是不希望文件中包含“*[CHerrorhand]”，而想要“CHerrorhand”引用的内容。

解决这个问题有 3 种可能的方式：

1. 使用文本编辑器查找所有的引用并用适当的文字串代替它们。
2. 使用 sed 进行编辑工作。这类似于手工编辑工作，只是比较快。
3. 使用 awk 编写一个程序：（a）用来读取中心文件，并生成引用名及其定义的列表；（b）读取文档查找引用调用序列；（c）查寻列表中引用的名字并用其定义代替它。

第一种方法明显的很耗时（而且没有趣！）。第二种方法（使用 sed）的优点是它可以创建一个工具来完成这项工作。例如，编写寻找“*[CHerrorhand]”并用“Chapter 16,Error Handling”来代替它的 sed 脚本相当简单。同样的脚本可用于修改文档的每个文件。缺点是这种替换是硬编码的，也就是说，对于每个交叉引用，都需要编写一个条命令来进行替换。第三种方法（使用 awk）构建一个工具，这个工具对遵循这种语法的任何交叉引用都起作用。这个脚本也可以对于其他书中的交叉引用进行扩展，它将大大节省你编写特殊替换列表的时间。它是这三种方法中最常用的解决方案，并且被设计为最可能重要的工具。

运维技术书籍免费下载 https://www.linuxprobe.com

解决问题中的部分工作是要知道构建哪种工具。有时 sed 脚本可能是一个比较好的选择，因为问题本身可能无法用 awk 进行处理，或者可能需要非常冗长复杂的 awk 脚本才能解决。必须要记住 sed 和 awk 对什么样的应用是最合适的。

字符流编辑器

sed 是一个“作交互式的”面向字符流的编辑器。它和许多 UNIX 程序一样，被认为是面向字符流的，这是因为输入流通过程序并将输出直接送到标准输出端。（例如，vi 不是面向字符流的，大多数 DOS 应用程序也不是）。输入一般来自文件，但是也可以直接来自键盘（注 2）。输出在默认情况下输出到终端屏幕上，但是也可以输出到文件中，sed 通过解释脚本来工作，该脚本指定了将要执行的动作。

sed 提供的功能好像是交互式文本编辑的自然延伸。例如，它提供的查找和替换程序可以被全局的应用于单个文件或一组文件。在处理一个特定文件中之出现一次的术语时，你可能不常用 sed，但你会发现利用它处理很多文件中的一系列修改是很有用的。只要想到在大约 100 多个文件中，处理 20 个不同的编辑工作可以在几分钟之内完成，你就会知道 sed 的强大了。

使用 sed 和编写简单的 shell 脚本（或 DOS 中的批处理文件）类似，可以指定一系列顺序执行的动作。其中的大多数动作可以在 vi 的内部用手工来完成：替换文本、删除行、插入新文本等等。sed 的优点是可以在一个地方指定所有的编辑指令，然后

注 2： 然而，这么做并不特别有用。

通过文件传递一次来执行它们。你不必进入每个文件来做每个修改。sed 还可以有效地用于编辑非常庞大的文件，而这个文件利用交互式编辑进行处理将会很慢。

在创建和维护文档的过程中也经常会用到 sed，尤其当文档由独立的章节组成，而且每个章节有存在不同的文件中时。通常，在评审返回文档的草稿之后，有许多修改可以应用于所有的文件。例如，在软件文档设计过程中，软件的名字或它的组件可能会改变，你必须找到他们并进行修改。使用 sed 进行修改将是一个非常简单的过程。

sed 能够用来保持整个文档的一致性。你可以查找到某个特定术语的所有不同

运维技术书籍免费下载 https://www.linuxprobe.com

的使用方式并且使它们变得一致。在通过 troff 进行格式化之前，可以使用 sed 插入特殊的排版代码或符号。例如，利用 sed 可以用 ASCII 字符代码取代前后的双引号（“弯引号”而不是“直引号”）。

sed 还可以作为编辑过滤器使用。换句话说，你可以处理输入文件并将输出结果发送到另外一个程序。例如，你可以使用 sed 分析纯文本本件并在将输出送到 troff 进行格式化之前插入 troff 宏。它允许你动态（on the fly）编辑（也许是临时的）。

作者或出版商可以用 sed 编写大量的转换程序，例如，将 Scribe 或 TBX 文件中的格式化代码转换成 troff，或者转换 PC 字处理文件，例如 WordStar。稍后，我们将看到一个用在 Venturna Publisher 中的将 troff 宏转换成样式表标签的 sed 脚本（也许利用 sed 可以将用某种语言的语法编写的程序，转换成具有另一种语言的语法的程序）。当 Sun 公司首次产生 Xview 时，它们发布了一些将 SunView 程序转换成 Xview 的转换程序，这些程序主要由 sed 脚本组成，可以转换不同函数的名字。

Sed 拥有用于构建更复杂的脚本的几个基本的程序设计结构。它在每次多于一行的处理能力方面有限制。

除了最简单的 sed 脚本外，几乎所有的都是从“shell 实现”中调用的，shell 实现是一种 shell 脚本，它调用 sed 而且还包含 sed 执行的命令。shell 实现是一个命名和执行单字命令的简单方式。使用这个命令的用户甚至不需要知道 sed 正在被使用。一个 shell 包装器的例子如 phrase 脚本（在本书后面我们将看到这个脚本）。它允许你匹配包含两行的模式，寻址 grep 的特殊限制。

使用 sed 的小结：

1. 在一个或多个文件上自动实现编辑操作。
2. 简化对多个文件执行相同的编辑处理工作。
3. 编写转换程序。

模式匹配的程序设计语言

将 awk 称为程序设计语言会吓跑一些人。如果你是其中之一，可以把 awk 看做是解决问题的一种不同方法,即利用这种方法可以对计算机所要做的事情增加更多的控制。

sed 很容易被堪称是与交互式编辑相反的程序。sed 程序与如何手动地应用编辑

运维技术书籍免费下载 https://www.linuxprobe.com

是很相近的。Sed 限制你利用在文本编辑器中使用的方法。Awk 为处理文件提供了更一般的计算模型。

awk 程序的典型示例是将数据转换成格式化的报表。这些数据可能是由 UNIX 程序（例如 uucp）产生的日志文件，而且报表以一种对系统管理员有用的格式化将数据汇总。另一个例子是由独立的数据项和数据检索程序组成的数据处理应用程序。数据项是以结构化方式记录数据的过程。数据检索式从文件中提取数据并生成报告的过程。

所有这些操作的关键是数据拥有某种结构，我们可以利用衣柜来类比解释它。衣柜由多个抽屉组成，每个抽屉都放人一些特定的内容：一个抽屉里是短袜，另一个抽屉式内衣，第三个抽屉式运动衫。有事抽屉里面可以分隔成几个间隔并将不同种类的东西放在一起。这些就是决定了东西放在哪里的结构，当你挑选要送去洗的东西，以及当你要穿衣服时，就可以在他们所在的地方找到。Awk 允许你在编写放进和拿出东西的程序中使用文本文件结构。

因为，当数据拥有某种结构时就能最好地体现 awk 的好处。文本文件的结构或松或紧。包含主要的节和次要的节的一章也是一种结构。我们将看到一个脚本，用于提取章节标题并将它们编号以生成一个大纲，由制表符分隔的列项目所组成的表是高度结构化的。可以使用 awk 脚本对数据的列重新排序，甚至可以将列变成行以及将行变成列。

和 sed 脚本一样，awk 脚本一般是利用 shell 包装器来调用。Shell 包装器是一个 shell 脚本，它通常包含调用 awk 的命令行以及 awk 解释的脚本。简单的一行 awk 脚本可以从命令行输入。

下面是 awk 能够完成的一些功能：

- 将文本文件看做由记录和字段组成的文本数据库。
- 使用变量操作数据库。
- 使用算术和字符串操作符。
- 使用普通的程序设计结构，例如循环和条件。
- 生成格式化报告。
- 定义函数。
- 从脚本中执行 UNIX 命令。
- 处理 UNIX 命令的结果。
- 更加巧妙的处理命令行的参数。
- 更容易地处理多个输入流。

运维技术书籍免费下载 https://www.linuxprobe.com

由于这些特征，用户可以根据 awk 所具有的能力和适用范围来处理由 shell 脚本执行的各种任务。在本书中，你将看到一个基于菜单的命令生成器，一个交互式拼写检查器和一个索引处理程序的示例，所有这些示例都是使用了上面提到的功能。

Awk 的功能将文本编辑的思想扩展到计算，使它有可能执行各种数据处理任务，包括分析、提取和数据报告。实际上，这是 awk 的最普遍的用法，但是也有许多不常使用的应用：awk 曾被用来编写 Lisp 解释程序，甚至是编译程序！

掌握 sed 和 awk 的四个障碍

有许多介绍 UNIX 的书都可以让你了解 sed 和 awk。本书的目的是帮组你以最快和最容易地方式更进一步的掌握 sed 和 awk。

在掌握 sed 和 awk 的过程中存在四个障碍，你必须学会：

1. 如何使用 sed 和 awk。清楚这个障碍相对容易，因为 sed 和 awk 的工作方式类似，都基于行编辑器 ed。第二章“了解基本操作”概述了使用 sed 和 awk 的机制。
2. 应用 UNIX 正则表达式语法。使用 UNIX 正则表达式语法进行模式匹配是 sed 和 awk 共有的特征，对许多其他的 UNIX 程序一样。这是一个较苦难的障碍，原因有两个：语法是神秘的，虽然许多人都有使用正则表达式的经验，但是很少有人能完全掌握语法。对语法的使用越熟悉，那么对 sed 和 awk 的使用也越容易，这就是我们为什么要花大量时间在第三章“了解正则表达式语法”中讨论正则表达式的原因。
3. 如何与 shell 进行交互。当与 sed 和 awk 本身不直接相关时，管理与 shell 命令的交互式一个困难的问题，因为 shell 与两个程序共享大量特殊的字符。如果可以的话，通过在独立的文件中放置脚本来避免这种问题。如果不行的话，就为脚本使用兼容的 Bourne shell（引用规则更加直观），并使用单引号包含你的脚本。如果将 csh 作为你的交互式 shell，记住要用反斜线（”\!”）转义感叹号。没有其他的方式能让 csh 别管感叹号（注 3）。
4. 脚本编写的技巧。这是最困难的，这就像跨栏竞赛中的高栏。因此，本书大部分都介绍脚本编写。使用 sed，你必须了解一组单字母的命令。使用 awk，你必须了解程序设计语言的语句。要掌握脚本编写的技巧，不仅必须注意大量的示例，而且必须亲自尝试编写脚本。

如果你正进行跨栏竞赛，只是跃过跨栏并不能赢得比赛，而需要的是快速跃过它们才能赢得比赛。在编写脚本中，学习脚本化命令集或语言只是简单地“跳过了跨栏”，要让你的脚本处理问题更有趣，就要获得在竞赛中跑的更快的能力。

注 3： 可以设置 histchars 变量。参见 csh 帮助页。

运维技术书籍免费下载 https://www.linuxprobe.com

第二章

了解基本操作

本章内容
- awk 起源于 sed 和 grep 而不是 ed
- 命令行的语法
- 使用 sed
- 使用 awk
- 同时使用 sed 和 awk

如果你正要开学习 sed 和 awk,最好从了解他们的共同点入手:
- 它们都使用相似的语法来调用
- 它们都是面向字符流的，都是从文本文件中一次一行的读取输入，并将输出直接送到标准输出端。
- 它们都使用正则表达式进行模式匹配。
- 它们都允许用户在脚本中指定指令。

它们有如此多的共同点，原因之一是它们都起拳于相同的行编辑器--ed。在这一章中，我们首先对 ed 做简短介绍，再介绍 sed 和 awk 是如何一步步形成可编程的编辑器的。

Sed 和 awk 的区别在于它们控制所做的工作时所用的指令不同。这是一个主要的区别，而且这影响了这些程序最适于处理的任务类型。

本章将介绍 sed 和 awk 的命令行语法和脚本的基本结构。还使用邮件列表提供了一个指南，这将会为你编写脚本提供一定的感性认识。在集中理解 sed 和 awk 之前，先阅读它们的脚本是很有价值的。

awk 起源于 sed 和 grep 而不是 ed

可以将 awk 的起源追溯到 sed 和 grep，并且经由这两个程序追溯到 ed(最初的 unix 行编辑器)

如果使用行为编辑器，那么理解 sed 和 awk 的行定位就会更容易，如果使用过 vi（全屏幕的编辑器），那么你一定熟悉由底层的行编辑器 ex(它依次是 ed 中特征的扩展集).

我们来看一些使用行编辑器 ed 的基本操作。不要担心，这只是帮助你了解 sed 和 awk 的练习，而不是想让你相信行编辑器的奇妙。这个练习中出现的 ed 命令和稍后要学到的 sed 命令相同。你可以自由地使用 ed 做实验，以便对它如何工作有一个了解（如果你已经很熟悉 ed，可以直接跳到下一节）。

使用行编辑器，每次可以处理一行。要知道处于文件那一行是很重要的。当使用 ed 打开文件时，它显示了文件中的字符个数并定位在最后一行。

```
$ ed tes
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
339
```

没有提示符。如果输入了 ed 不理解命令，它将打印一个问号作为错误消息。可以输入打印命令 p 来显示当前的行。

```
p
label on the first box.
```

默认情况下，一个命令只影响当前的行。要进行一项编辑工作，首先要移至想要编辑的行，然后应用相应该的命令。要移到某一行，就要指定它的地址(address).一个地址可以由一个行号，一个指示文件中特定位置的符号或一个正则表达式组成。通过输入行号 1 可以转到第一行，然后输入删除命令来删除那一行。

```
1
You might think of a regular expression
D
```

输入“1”使第一行成为当前行，并在屏幕上显示它。Ed 中删除命令是 d,上例中是删除当前行。与移到某行然后再对它进行编辑不同的是，可以将标识命令对某一行或某些行的地址，放在编辑命令的前面作为编辑命令的前缀。例如，如果输入“1d”，那么第一行会被删除。

还可以把一个正则表达式作为一个地址。为了删除包含单词“regular”的行，可以使用下面的命令：

```
/regular/d
```

其中的斜杠界定的对象是正则表达式，“regular”是想要匹配的字符串。这个命令删除包含“regular”的第一行并且使跟在它后面的这一行为成当前行。

注意： **确信你已经理解了使用删除命令来删除整个行。它不只是删除那一行上的单词“regular”。**

要删除包含这个正则表达式的所有行，可以在命令前面加上字母 g，表示该命令是一个全局命令。

```
g/regular/d
```

全局命令使匹配正则表达式的所有行成为特定命令的对象。

迄今为止你只接触到了删除文件。替代文件（用文本中的一部分取代另一部分）更有趣。ed 中的替换命令 s 是：

```
[address]s /pattern/replacement/flag
```

pattern 是一个正则表达式，度用 replacement 替代当中行中与这个正则表达

运维技术书籍免费下载 https://www.linuxprobe.com

式匹配的字符串。例如，下面的命令中“complex"取代当前行上第一次出现的"regular"。

```
s/regular/complex/
```

由于没有指定地址，所以它只影响当前行上的第一次出现。如果在当前行上没有找到“regular”则出现一个错误。为了寻找同一行上多次出现，必须指定 g 作为标志：

```
s/regular/complex/g
```

这个命令改变了当前行上的所有的出现。必须指定地址从而使该命令不只是对当前行操作。下面的替换命令指定了一个地址：

```
/regular/s/regular/complex/g
```

这个命令影响文件中与这个地址匹配的第一行。记住，第一个“regular”是一个地址，第二个是匹配替换命令的模式。要将它应用于所有的行，必须使用全局命令，既在地址前放置 g：

```
g/regular/s/regular/complex/g
```

现在，这个替换应用于所有的地方，即所有行上的所有出现。

注意： **注意“g”的不同含义。开始处的“g”是全局命令，意味着对所有与地址匹配的行进行改变。结尾处的“g”是一个标志、意味着改变一行上的每个出现，不只是第一个。**

地址和模式不必相同。例如：

```
g/regular expression/s/regular/complex/g
```

表示在包含字符串“regular expression”的任意行上，用“complex”代替“regular”。如果地址和模式相同，那么可以通过指定两个连续的定界符（//）来告诉 ed.

```
g/regular/s//complex/g
```

在这个例子中，“regular”被指定为“地址”、同时应用相应的地址匹配替换模式。这些命令中有大量的内容需要掌握、然而看上去我们对这些命令的介绍却只是一带而过，不要担心，稍后还要讲到这些命令。

类似的 unix 实用工具 grep 来源于 ed 中的下面的全局命令：

运维技术书籍免费下载 https://www.linuxprobe.com

```
g/re/p
```

它表示“全局正则表达式打印”。Grep 是从 ed 中提取并可用做外部程序的行编辑命令，它只是执行一个编辑命令的“硬连接（hard-wired）”。将正则表达式作为命令行上的一个参数并将它用做要打印行的地址。如下例所示，寻找匹配“box”
的行：

```
$ grep ’box’ test
You are given a series of boxes, the first one labeled "A",
label on the first box.
```

它打印匹配正则表达式的所有的行。

ed 的一个更有趣的特征是脚本化编辑工作的能力，将编辑命令放在独立的文件中并将它们作为行编辑器的输入。例如，如果将一系列放到名为 ed-script 的文件中，下面的命令将执行这个脚本：

```
ed test < ed-script
```

这个特征使 ed 成为可编程的编辑器。也就是说，你可以脚本化任何手动执行的操作。

sed 是作为特殊目的的编辑器而创建的，用于专门执行脚本;与 ed 不同，它不能交互使用。Sed 与 ed 的主要区别在于它是面向字符流的。默认情况下，到 sed 的所有输入都会经过相应的处理，并转为标准输出。输入文件本身不发生改变。哪果确实想改变输入文件，一盘使用 shell 机制进行输入重定向，当你对所做的编辑工作满意时，用修改后的版本代替最初的文件。

ed 不是面向字符流的，并且文件本身会发生改变。Ed 脚本必须包含保存文件并退出编辑器的命令。它不产生到达屏幕的输出，但由特殊命令生成的东西除外。

Sed 的字符流定位对如何应用寻址有重要影响。在 ed 中没有指定地址的命令只影响当前行。Sed 遍历文件，每次一行，这样每一行都成为当前行，而且每一行都应用这个命令，结果是 sed 对文件中的每一行应用了没有地址的命令。

看一下下面的替换命令：

```
s/regular/complex/
```

如果在 ed 中交互式地输入这个命令，则用“complex”取代当前行上第一次出现的“regular”。在 ed 脚本中，如果这是脚本中的第一个命令，那么它就只应用于文件的最后一行（ed 的默认当前行）。然而，在 sed 脚本中，相同的命

运维技术书籍免费下载 https://www.linuxprobe.com

令应用于所有的行。也就是说，sed 命令是隐式的全局命令。在 sed 中，上一个示例的命令和 ed 中如下所示的全局命令结果相同。

```
g/regular/s//complex/
```

注意： 理解 ed 中的当前行寻址与 sed 中全局行寻址之间的区别是很重要的。在 ed 中，使用寻址扩大受命令影响的行数;在 sed 中，使用寻址限制受命令影响的行数。

sed 还具有一些支持编写脚本的额外命令。在第六章“高级 sed 命令”中可以看到一部分这样的命令。

awk 是作为可编程的编辑器而开发的，同 sed 一样，它也是面向字符流的，并且解释编辑命令的脚本。Awk 与 sed 不同的地址是它废弃了行编辑器的命令集。它提供了仿效 C 语言的程序设计语言，例如，print 语句取代了 p 命令;但是延续了寻址的概念，例如：

```
/regular/ { print }
```

用于打印匹配“regular”的那些行。大括号（{}）用于包围应用于同一地址的一个或多个语句。

在脚本中使用程序设计语言的优点是，它提供了更多的方式来控制可编程的编辑器所做的事情。Awk 提供了表达式、条件语句、循环和其他程序设计结构。

Awk 最独特的特征之一是它分析或拆分每个输入行，并生成可用于脚本处理的独立的单词（一个编辑器，例如 vi,也识别单词，允许一个单词一个单词地移动，或者使一个单词成为操作对象，但是这些特征只能在交互方式下使用）。

虽然 awk 是作为可编程的编辑器设计的，但是用户全发现，awk 脚本也能完成许多其他任务。awk 的作者永远不会想到它被用于编写庞大的程序。但是，当认识到 awk 能这样使用以后，作者们修订了这种语言，创建了 nawk，为编写庞大的程序和解决多方面的程序设计问题提供了更多的支持。这个新的版本（有较小的改进）目前已经由 posix 组织作为标准。

命令行的语法

可以用大至相同的方式调用 sed 和 awk。命令行讲法是：

```
command [options] script filename
```

几乎和所有的 unix 程序一样，sed 和 awk 都可以从标准输入中取得输入并将输出发送到标准输出。如果指定文件名 filename，输入就取自那个文件。输出包含处理后的信息。标准输出是指屏幕，而且一般来自这些程序的输出都输出到

运维技术书籍免费下载 https://www.linuxprobe.com

那里。输出也可被送到一个文件。例如 shell 中的 I/O 重定向，但是不允许送到向程序提供输入的同一个文件。

每个命令的 options 是不同的。我们将在以后讨论这些选项(sed 命令行选项的完整列表可以在附录一“sed 的快速参考”中找到，awk 命令选项的完整列表可以在附录二“awk 的快速参考”中找到）。

scipt 指定了要执行的指令。如果在命令行上 script，假如它包含有可以由 shell 解释的空格或任意字符（例如$和*），那么它必须用单引号括起。

sed 和 awk 一个共同的选项是-f 选项，这个选项允许你指定脚本文件的名字。附着脚本大小的增长，将它放置在文件中比较方便。因此，可以按如下方式调用 sed:

```
sed -f scriptfile inputfile
```

图 2-1 展示了 sed 和 awk 的基本操作。每个程序每次从输入文件中读取一个输入行，生成该输入行的备份，并且对该备份执行脚本中指定的指令。因此，对输入行所做的改动不会影响真正的输入文件。

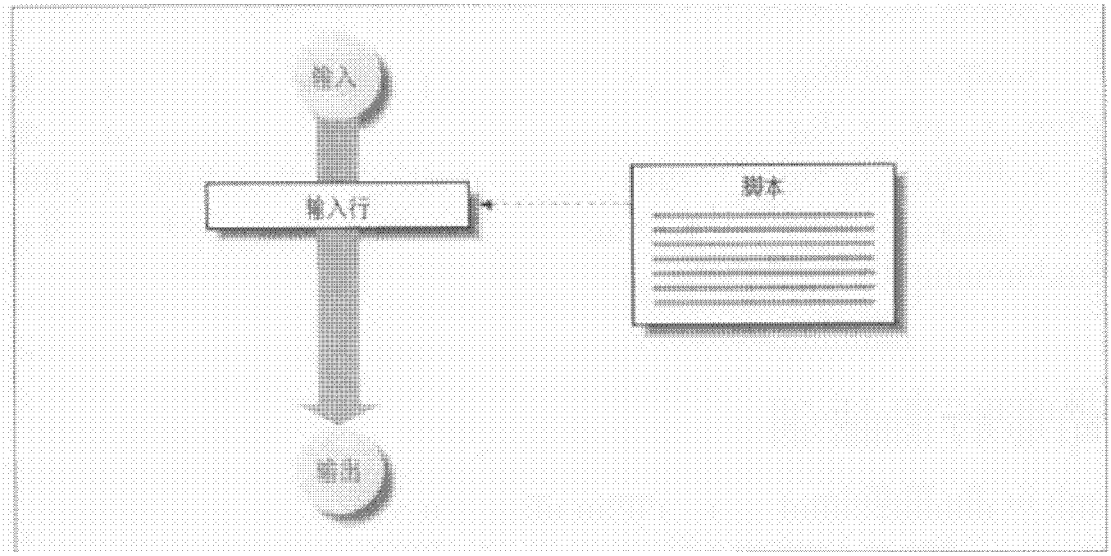

图 2-1：sed 和 awk 的工作方式

脚本化

脚本就是告诉程序做什么的地方。至少要包含一行指令。短的脚本可以在命令行上指定，长的脚本通常放在容易被修改和测式的文件中。在编写脚本时，要记住指令执行的顺序以及每个指令如何改变输入行。

在 sed 和 awk 中，每个指令都包括两个部分：模式和过程。模式是由斜杠（/）分隔的正则表达式。过程指定一个或多个将被执行的动作。

当读取输入的每行时，程序读取脚本中的第一个指令并检测当前行的模式。如果没有匹配，这个过程被忽略并读取下一个指令。如果有一个不匹配，那么执

运维技术书籍免费下载 https://www.linuxprobe.com

行过程中指定的一个或多个动作。读取所有的指令，而不只是读取与输入行匹配的第一条指令。

当所有可用指令被解释并应用于单个行后，sed 输出该行并循环处理每个输入行。另一方面，awk 不自动输出行，脚本中的指令控制 awk 最终所做的事情。

在 sed 和 awk 中过程的内容有很大不同。在 sed 中，过程由类似于行编辑器中使用的那些编辑命令组成。大部分命令由单个字母组成。

在 awk 中，过程由程序设计语句和函数组成。过程必须用大括号括起。

在下面的一节中，我们将看到几个处理邮件列表的脚本示例。

邮件列表的示例

在接下来的一节中，示例中使用了样本的文件，名为 list。它包含了名字和地址的列表。如下图所示：

```
$ cat list
John Daggett, 341 King Road, Plymouth MA
Alice Ford, 22 East Broadway, Richmond VA
Orville Thomas, 11345 Oak Bridge Road, Tulsa OK
Terry Kalkas, 402 Lans Road, Beaver Falls PA
Eric Adams, 20 Post Road, Sudbury MA
Hubert Sims, 328A Brook Road, Roanoke VA
Amy Wilde, 334 Bayshore Pkwy, Mountain View CA
Sal Carpenter, 73 6th Street, Boston MA
```

如果愿意，可以在你的系统上创建这个文件或者创建一个类似的文件。因为本章中多数示例都很短并且是可交互的，你可以从键盘上输入它们并确认最终结果。

使用 sed

调用 sed 有两种方法：在命令行上指定编辑指令，或者将它们放到一个文件中并提供这个文件的名字。

指定简单的指令

可以在命令行上指定简单的编辑命令。

```
sed [-e] ’instruction’ file
```

只有在命令行上给出多个指令时才需要用-e 选项。它告诉 sed 将一下参数解释为指令。当只有一个指令时，sed 可以自己做决定。看一些示例。

```
$ sed ’s/MA/Massachusetts/’ list
John Daggett, 341 King Road, Plymouth Massachusetts
Alice Ford, 22 East Broadway, Richmond VA
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
Orville Thomas, 11345 Oak Bridge Road, Tulsa OK
Terry Kalkas, 402 Lans Road, Beaver Falls PA
Eric Adams, 20 Post Road, Sudbury Massachusetts
Hubert Sims, 328A Brook Road, Roanoke VA
Amy Wilde, 334 Bayshore Pkwy, Mountain View CA
Sal Carpenter, 73 6th Street, Boston Massachusetts
```

该指令只影响了 3 行，但显示出了所有行。

并不是在任何情况下都需要将指令用单引号包围起来，但是你应该养成这个习惯。使用单引号可以阻止 shell 解释编辑指令中的特殊字符或空格（shell 使用空格决定提交给程序独立参数，特殊的 shell 字符在调用之前被展开）。

例如，以上例子可以不使用单引号，但下一个例子就需要使用单引号，因为替换命令中包含空格：

```
$ sed ' s/ MA/, Massachusetts/'  list
John Daggett, 341 King Road, Plymouth, Massachusetts
Alice Ford, 22 East Broadway, Richmond VA
Orville Thomas, 11345 Oak Bridge Road, Tulsa OK
Terry Kalkas, 402 Lans Road, Beaver Falls PA
Eric Adams, 20 Post Road, Sudbury, Massachusetts
Hubert Sims, 328A Brook Road, Roanoke VA
Amy Wilde, 334 Bayshore Pkwy, Mountain View CA
Sal Carpenter, 73 6th Street, Boston, Massachusetts
```

为了在城市（city)和（state）之间放置逗号，指令用一个逗号和一个空格取代两字母缩写词前面的空格。

有 3 种方式可以指定命令行上的多重指令：

1. 用分号分隔指令。
   sed ' s/ MA/, Massachusetts/; s/ PA/, Pennsylvania/' list
2. 在每个指令前放置-e.
   sed -e ' s/ MA/, Massachusetts/' -e ' s/ PA/, Pennsylvania/' list
3. 使用 Bourne shell 的分行指令功能（注 1）。在输入单引号后按 return 键，就会出现多行输入的提示符（>).

```
$ sed '
> s/ MA/, Massachusetts/
> s/ PA/, Pennsylvania/
> s/ CA/, California/'  list
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
John Daggett, 341 King Road, Plymouth, Massachusetts
Alice Ford, 22 East Broadway, Richmond VA
Orville Thomas, 11345 Oak Bridge Road, Tulsa OK
Terry Kalkas, 402 Lans Road, Beaver Falls, Pennsylvania
Eric Adams, 20 Post Road, Sudbury, Massachusetts
Hubert Sims, 328A Brook Road, Roanoke VA
Amy Wilde, 334 Bayshore Pkwy, Mountain View, California
Sal Carpenter, 73 6th Street, Boston, Massachusetts
```

这种技术在 C shell 中不能使用。C shell 中采用的方法是，在每个指令的结尾使用分号，并且通过用反斜杠作为每一行的结束，从而可以输入跨越多行的命令(或者通过输入 sh 临时进行 Bourne shell，然后用心上方法键入命令。

注 1： 目前，有许多与 Bourne shell 兼容的 shell，能够像这里描述的那样工作， 如 ksh, bash, pdksh 和 zsh 等。

失误的命令

Sed 命令的语法可以被细化，在使用时很容易犯错误或者忽略掉需要的元素。注意输入以下不完整的语法将发生的情况：

```
$ sed -e ' s/MA/Massachusetts' list
sed: command garbled: s/MA/Massachusetts
```

Sed 通常会显示任何它不能执行的行，但是它不会告诉你使用这个命令所发生的错误是什么（注 2）。在上述情况下，标记搜索和替换命令末尾的斜杠丢失了。

以下 GNU sed 给出了更清楚的提示：

```
$ gsed -e ' s/MA/Massachusetts' list
gsed: Unterminated 's' command
```

脚本文件

在命令行上输入较长的编辑脚本是不实际的。这就是通常最好创建包含编辑指令的脚本文件的原因。编辑脚本只是一系列要依次的简单的 sed 命令。这种形式使用-f 选项来指定命令行上的脚本文件的名字，格式如下：

```
sed -f scriptfile file
```

将想要执行的所有编辑命令都放置在文件中。我们按照惯例创建临时脚本文件 sedscr。

```
$ cat sedscr
s/ MA/, Massachusetts/
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
s/ PA/, Pennsylvania/
s/ CA/, California/
s/ VA/, Virginia/
s/ OK/, Oklahoma/
```

注 2: 一些厂商在这方面似乎已经做了改进。
例如:在 sunos 4.1.X 上，sed 可以报告“sed:Ending delimiter missing on substitution :s/MA/Massachusetts”.

下面的命令读取 sedscr 中的所有替换命令，并将这些命令应用于输入文件 list 中的每一行:

```
$ sed -f sedscr list
John Daggett, 341 King Road, Plymouth, Massachusetts
Alice Ford, 22 East Broadway, Richmond, Virginia
Orville Thomas, 11345 Oak Bridge Road, Tulsa, Oklahoma
Terry Kalkas, 402 Lans Road, Beaver Falls, Pennsylvania
Eric Adams, 20 Post Road, Sudbury, Massachusetts
Hubert Sims, 328A Brook Road, Roanoke, Virginia
Amy Wilde, 334 Bayshore Pkwy, Mountain View, California
Sal Carpenter, 73 6th Street, Boston, Massachusetts
```

再说一次，显示在屏幕上的结果是临时的，输入文件中没有发生改变。

如果 sed 脚本能再次使用，那么应该重新命名这个脚本并保存它。有用的脚本可以保存在个人的或系统共享库中。

保存输出

只有将 sed 的输出重定向到另一个程序中，才能够捕获文件中的输出。要完成这项工作需要在一个文件名后面指定一个 shell 的 I/O 重定向符号。例如:

```
$ sed -f sedscr list > newlist
```

不要将输出重定向到正在编辑的文件中，否则就会使它变成乱码（“>”重定向操作符在 shell 做任何其他事情之前截取文件）。如果想用输出文件取代输入文件，那么可以采用 mv 命令并将它做为单独的步骤来处理。但是首先要确保编辑的脚本是正确的！

在第四章“编写 sed 脚本”中将看到名为 runsed 的 shell 脚本，它能够自动的创建临时文件并使用 mv 改写原始文件。

阻止输入行的自动显示

运维技术书籍免费下载 https://www.linuxprobe.com

sed 的默认操作是输出每个输入行。-n 选项可以阻止自动输出。当指定该选项时，每个要生成输出的指令都必须包含打印命令 p。请看下面的示例。

```
$ sed -n -e ' s/MA/Massachusetts/p' list
John Daggett, 341 King Road, Plymouth Massachusetts
Eric Adams, 20 Post Road, Sudbury Massachusetts
Sal Carpenter, 73 6th Street, Boston Massachusetts
```

将这一输出与本节中的第一个示例进行比较。这里只会显示受命令影响的行。

混合选项（POSIX）

通过合并命令行上的-e 和-f 选项可以构建一个脚本。该脚本是所有命令按命令的给出顺序组合起来的。在 sed 的 unix 版本中似乎支持这项功能，但是这种特征在帮助页中没有明确给出。POSIX 标准明确地要求这种特征。

选项总结

表 2-1 汇总了 sed 的命令行选项。

表 2-1：sed 的命令行选项

选项	描述
-e	编辑随后的指令
-f	跟随脚本中的文件名
-n	阻止输入行的自动输出

使用 awk

与 sed 相似，awk 为每个输入行执行一套指令。可以在命令行上指定指令或创建脚本文件。

运行 awk

命令行的语法是：

```
awk ' instructions' files
```

每次从一个或多个文件中读入一行或从标准输入中读入一行。指令必须包含在单引号中，从而与 shell 区别开(指令几乎总是包含大括号和/或美元符号，shell 将它们解释为特殊符号)。可以用与 sed 相同的方式输入多重命令行：用分号分隔命令或使用 Borune shell 的多行输入功能。

awk 程序通常被放置在可以对它们进行测式和修改的文件中。用脚本文件调用 awk 的语法如下：

运维技术书籍免费下载 https://www.linuxprobe.com

```
awk -f script files
```

-f 选项的工作方式与在 sed 中相同。

尽管 awk 指令与 sed 指令的结构相同，都由模式和过程两部分组成，但过程本身有很大的不同。awk 看上去不像编辑器而更像一种程序设计语言。语句和函数取代了使用一个或两二个字符组成的命令序列。例如，使用 print 语句打印表达式的值或打印当前输入行的内容。

在通常情况下，awk 将每个输入行解释为一条记录而将一行上的每个单词（由空格或制表符分隔）解释为每一个字段(可以改变这些默认设置)。一个或多个连续的空格或制表符看做一个定界符。awk 允许在模式或过程中引用这些字段。$0 代表整个记录。我们来看几个示例，使用样本输入文件 list.

第一个示例包含单个指令，用于打印输入文件中每行的第一个字段。

```
$ awk ' { print $1 }'  list
John
Alice
Orville
Terry
Eric
Hubert
Amy
Sal
```

“$1”表示每个输入行上的第一个字段的值。因为这里没有指定模式，所以打印语句应用于所有的行。下一个示例指定了“/MA/”模式，但是其中没有过程。这个默认操作是打印匹配这种模式的每一行。

```
$ awk '/MA/'  list
John Daggett, 341 King Road, Plymouth MA
Eric Adams, 20 Post Road, Sudbury MA
Sal Carpenter, 73 6th Street, Boston MA
```

本例打印了 3 行。正如在第一章所提到的那样，awk 程序更像一种查询语言，从文件中提取有用的信息。可以认为惟上模式指定了一种条件，用于选择要包括在报表中的记录，也就是这些记录必须包含字符串“MA”。现在还可以指定记录的哪些部分要包括在报表中。下一个示例使用一条 print 语句限制只输出每条记录的第一个字段。

```
$ awk '/MA/ { print $1 }'  list
John
Eric
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
Sal
```

如果尝试着大声音阅读这一句话：print the first word of each line containing the sting “MA”，这将有助于我们理解上面的指令。可以把它说成“word（单词）”，这是因为在默认情况下，awk 使用空格或制表作为字段分隔符将输入分隔成字段。

在下一示例中，使用-F 选项将字段分隔符改变为逗号。它使我们能够检索 3 个字段中的任一个：全称、街道地址或城市和州。

```
$ awk -F, ' /MA/ { print $1 }'  list
John Daggett
Eric Adams
Sal Carpenter
```

不要改变字段分隔符的-F 选项与指定脚本文件名的-f 选项弄混。

下一个示例将每个字段单独打印在这一行上。多重命令由分号隔开。

```
$ awk -F, ' { print $1; print $2; print $3 }'  list
John Daggett
341 King Road
Plymouth MA
Alice Ford
22 East Broadway
Richmond VA
Orville Thomas
11345 Oak Bridge Road
Tulsa OK
Terry Kalkas
402 Lans Road
Beaver Falls PA
Eric Adams
20 Post Road
Sudbury MA
Hubert Sims
328A Brook Road
Roanoke VA
Amy Wilde
334 Bayshore Pkwy
Mountain View CA
Sal Carpenter
73 6th Street
Boston MA
```


运维技术书籍免费下载 https://www.linuxprobe.com

前面的示例使用 sed 改变了输入数据的内容。使用 awk 重新排列数据。在上面的 awk 示例中，注意如何将前导空白看做是第二个和第三个字段的一部分。

出错信息

当遇到程序中的问题时，awk 的每个实现都会给出不同的出错信息。因此，这里不引用特殊版本的消息;当出现问题时它是显而易见的。下面任何一种情况都会产生消息：

- 没有用大括号（{}）将过程括起来
- 没有用单引号（‘’）将指令括起来
- 没有用斜杠（//）中将正则表达式括起来

选项总结

表 2-2 汇总了 awk 命令行的选项同。

表 2-2：awk 的命令行选项

-f	跟随脚本的文本名
-F	改变字段分隔符
-v	跟随 var=value

在命令行上指定的参数中的-v 选项将在第七章“编写 awk 脚本”中讨论。

同时使用 sed 和 awk

在 unix 中，管道可用于将一个程序的输出作为输入传递给另一个程序。参看几个综合使用 sed 和 awk 来产生报表的示例。用州的全名代替邮政编码的 sed 脚本通常已经足够，它可被再次用做名为 nameState 的脚本文件：

```
$ cat nameState
s/ CA/, California/
s/ MA/, Massachusetts/
s/ OK/, Oklahoma/
s/ PA/, Pennsylvania/
s/ VA/, Virginia/
```

当然，要处理所有的州，而不只是这 5 个，并且如果是对文档而不是邮件列表执行这些命令，则要确保不会生成不必要的替换。

该程序(使用输入文件 list)的输出，与我们已经看到的相同。在下一个示例中，由 nameState 产生的输出被输送到一个 awk 程序中，这个 awk 程序用于从每条记录中提取州的名字。

运维技术书籍免费下载 https://www.linuxprobe.com

```
$ sed -f nameState list | awk -F, ' { print $4 }'
Massachusetts
Virginia
Oklahoma
Pennsylvania
Massachusetts
Virginia
California
Massachusetts
```

以上 awk 程序处理由 sed 脚本产生的输出。记住前面的 sed 脚本用逗号和州的全称代替缩写。实际上，它将包含城市和州的第三个字段拆分成二个字段。“$4”表示第 4 个字段。

这里所做的事情可以完全由 sed 完成，但可能会有更多的困难和更少的通用性。而且，因为 awk 允许替换匹配的字符串，所以可以完全用 awk 脚本得到这个结果。

本程序的结果不是非常有用，可以将它传递给 sort |uniq -c ，它将州名按字母表排序，同量给出每个州名出现的资料。

现在，我们要做一些更有趣的事情。按州的名字排序并列出州的名字，以及住在那个州的人的名字。下面的示例展示了 byState 程序：

```
#! /bin/sh
awk -F, ' {
print $4 ", " $0
}'  $* |
sort |
awk -F, '
$1 == LastState {
print "\t" $2
}
$1 != LastState {
LastState = $1
print $1
print "\t" $2
}'
```

这个 shell 脚本有 3 个部分。程序中调用 awk 以产生 sort 程序的输入，然后再次调用 awk 测式排好序的输入，并确定当前记录中的州的名字，是否与前一个记录中的名字相同。我们来看这个脚本的执行：

运维技术书籍免费下载 https://www.linuxprobe.com

```
$ sed -f nameState list | byState
California
        Amy Wilde
Massachusetts
        Eric Adams
        John Daggett
        Sal Carpenter
Oklahoma
        Orville Thomas
Pennsylvania
        Terry Kalkas
Virginia
        Alice Ford
        Hubert Sims
```

这些名字被按州排序。这是使用 awk 从结构化数据中生成报表的典型示例。

为了检查 byState 程序是如何工作的，让我们分别看看每个部分。它被设计为从 nameState 程序读取输入并期待“$4”成为州的名字。查看由程序的第一行产生的输出：

```
$ sed -f nameState list | awk -F, ' { print $4 ", " $0 }'
Massachusetts, John Daggett, 341 King Road, Plymouth,
Massachusetts
Virginia, Alice Ford, 22 East Broadway, Richmond, Virginia
Oklahoma, Orville Thomas, 11345 Oak Bridge Road, Tulsa, Oklahoma
Pennsylvania, Terry Kalkas, 402 Lans Road, Beaver Falls,
Pennsylvania
Massachusetts, Eric Adams, 20 Post Road, Sudbury, Massachusetts
Virginia, Hubert Sims, 328A Brook Road, Roanoke, Virginia
California, Amy Wilde, 334 Bayshore Pkwy, Mountain View,
California
Massachusetts, Sal Carpenter, 73 6th Street, Boston,
Massachusetts
```

默认情况下，sort 程序按字母顺序排列行，从左到右查看字符。为了按州(而不是名字)对记录进行排序，我们将州作为排序的关键字插入到记录的开始处。现在，sort 程序可以工作了（注意使用 sort 实用工具可以避免在 awk 内部编写排序程序）。

第二次调用 awk 时执行程序设计任务。脚本查看每条记录的第一个字段以决定它是否与前一条记录相同。如果不相同，则同时打印州的名字的人的名字。如果相同，则只打印人的名字。

运维技术书籍免费下载 https://www.linuxprobe.com

```
$1 == LastState {
print "\t" $2
}
$1 != LastState {
LastState = $1
print $1
print "\t" $2
}'
```

这里还有几个重要的事情，包括给一个变量赋值，测式每个输入行的第一个字段来看它是否包含一个变量字符串，并且打印制符来调整输出数据的对齐。注意在使用某个变量之前不必对它先赋值(因为 awk 将变量初始化为空字符串)。这是较小的脚本，但是在第十二章“综合应用”中的更大的索引程序中，可以看到用于比较索引条目的类似的程序。然而，目前可以先不用过多的关心有关每条语句做什么事情。这里的目的是给你一个 sed 和 awk 能做什么的概述。

本章概述了有关 sed 的 awk 的基本操作。介绍了重要的命令行选项以及脚本的使用。下一章将介绍正则表达式，即两个程序用来在输入中进行模式匹配的某些内容。

运维技术书籍免费下载 https://www.linuxprobe.com

本章内容

- 表达式
- 成排的字符
- 使用喜欢的元字符

第三章 了解正则表达式语法

当小孩努力理解惯用语表达的含义时，例如“Someone let the cat out of the bag（有人把猫从口袋里放了出来）”，你可能帮他解释为这是一种表达意思的方式（表达式）、而不表示它的字面的含义。

即使在计算机术语中，表达式也不是按字面意思被解释。它是某些需要被计算的东西。一个表达式描述一种结果。

在本章中，我们将介绍正则表达式的语法。正则表达式描述了模式或特殊字符序列，尽管没有必要指定一个精确的序列。

虽然正则表达式是 UNIX 的一个基本部分，但并不是每个人都完全理解其语法。事实上，下面的这种表达式是非常难理解的：

^□□*.*

该表达式使用元字符（metacharacter）（也做通配符）或特殊的符号，匹配一个具有一个或多个前导空格的行（在实例中使用方框“□”是为了使空格可见）。

如果你用过基于宏的 UNIX 文本编辑器，那么可能对正则表达式语法比较熟习。grep、sed 和 awk 都使用正则表达式。然而，这 3 个程序并不能完全使用正则表达式语法中的所有元字符。基本的元字符集是由 ed 行编辑器引入的，而且在 grep 中可用。Sed 使用相同的元字符集。后来引入了名为 egrep 的程序，它提供了一个扩展的元字符集。awk 基本上使用与 egrep 相同的元字符集。

为了理解正则表达式，必须了解由不同的元字符执行的功能。而且还必须参见一些组合使用元字符进行工作的示列。这就是我们学习本章的方法。介绍每个元字符并提供大量的示列，大部分使用 grep 和他的堂兄弟 egrep 来演示实际的应用。

如果你已经了解了正则表达式语法，可以根据自己的意愿跳过这一章。正则表达式中的元字符的完整清单可以在表 3-1、附录一“sed 的快速参考”和附录一

运维技术书籍免费下载 https://www.linuxprobe.com

“awk 的快速参考”中找到。有兴趣的读者可以参考 O’Reilly 出版并由 Jeffrey E.F.Friedl 编著的《Mastering Regular Expression》，该书详细介绍了正则表达式的结构和用法。

表达式

你可能熟悉一个计算器解释的表达式。请看下面的算术表达式：

2+4

“2 加 4”出几个常数或字面值和一个操作符组成。计算器程序必须能够识别，例如，“2”是数字常数而加号表示一个操作符，而不能解释为“+”字符。

表达式告诉计算机如何产生结果。尽管我们真正想要的就是“2+4”的结果，但我们不能简单地告诉计算机返回 6。我们指示计算机计算表达式并返回值。

表达式可以比“2+4”更复杂，事实上，它由多个简单的表达式组成，例如：

2+3*4

计算器通常从左到右计算表达式。然而，某些操作符比其他操作符的优先级高，也就是，它们将被首先执行。因此，上面的表达式的结果为 14 而不是 20，因为乘法的优先级高于加法。将简单的表达式放入圆括号中可以改变优先级，因此“（2+3）*4”或“2 加 3 的和的 4 倍”的结果为 20。圆括号是指示计算器改变表达式计算顺序的符号。

相反，一个正则表达式描述了一种模式或字符序列。字符串连接是每个正则表达式的基本操作。也就是 一个模式匹配相邻的一系列字符。请看下面的正则表达式：

ABE

每个字面字符都是一个正则表达式，它只匹配那个单独的字符。这个表达式描述了“B 跟着 A，E 跟着 B”或简单称为“字符串 ABE”。术语“字符串”意味着每个字符都与它前面的字符相连接。不一定要将正则表达式描述为由字符序列组成（初学者倾向于将其考虑成由较高级的单元组成，例如由单词而不是独立的字符组成）。正则表达式区分大小写，因此“A”不匹配“a”（注 1）。

接受正则表达式的程序（例如 grep）必须首先解析正则表达式的语法来产生一个模式。然后它们一行一行地读取输入来尝试匹配该模式。输入行是一个字符串、而且要看字符串与模式是否匹配，程序将字符串的第一个字符与模式的第一个字符进行比较。如果匹配，就比较第二个字符。无论何时只要匹配失败，就返回并从字符串中这个字符后面的字符重新开始匹配。图 3-1 说明了这个过程，在输入行上尝试匹配模式“abe”。

运维技术书籍免费下载 https://www.linuxprobe.com

正则表达式不只限于文字字符。例如，元字符句点（.）可以作为“通配符”匹配任何单个字符。你可以认为这个通配符与 Scrabble 中的空白类似，可以表示任意字母。因此，我们可以指定正则表达式“A.E”而且它将和“ACE”、”ABE”和“ALE”都匹配。句点与“A”后面的位置上的任何字符匹配。

元字符*（星号）用于与它前面的正则表达式的零个或多个出现匹配，该表达式通常是一个字符。你也许对*作为一个 shell 元字符更熟习，在那里它表示“零或多个字符”。但是这与它在正则表达式中的含义不同。星号元字符本身不匹配任何字符、它用于修改它前面的内容。正则表达式.*匹配任意数目的字符，而在 shell 中，*本身就具有这种含义（例如，在 shell 中，ls*表示列出当前目录中的所有的文件）。正则表达式“A.*E”匹配任何与“A.E”匹配的字符串，但是它还匹配在“A”和“E”之间具有任意数目的字符；例如，“AIR-PLANE”、“A FINE”、“AFFABLE”或“A LONG WAY HOME”。注意“任意数目的字符”可以是零个字符！

如果理解了正则表达式中“.”和“*“之间的区别，那么就已经了解了元字符的两个基本类型，那些能够被看做单个字符的元字符和那些被看做如何修改前面的字符的元字符。

注 1： 某些其他程序为使用正则表达式提供了不区分大小写的选项，但是 sed 和 awk 区分大小写。

图 3-1：解释正则表达式

运维技术书籍免费下载 https://www.linuxprobe.com

很明显使用元字符可以扩展或限制可能的匹配。你可以更多地控制匹配什么和不匹配什么。

成排的字符

我们已经看过了表达式中的两个基本元素：

1. 以一个字面值或变量表示的值。
2. 一个操作符。

正则表达式是由这些相同的元素组成的。任意字符（表 3-1 中的元字符除外）都被解释为只匹配它本身的字面值。

表 3-1：元字符汇总

特殊字符	用途
.	匹配除换行符以外的任意单个字符。在 awk 中，句点也能匹配换行符。
*	匹配任意一个（包括零个）在它面前的字符（包括由正则表达式指定的字符）
[...]	匹配方括号中的字符类中的任意一个。如果方括号中第一个字符为脱字符号（^），则表示否定匹配，即匹配除了换行符和类中列出的那些字符以外的所有字符。在 awk 中，也匹配换行符。连字符（-）用于表示字符的范围。如果类中的第一个字符为右方括号（]）则表示它是类的成员。所有其他的元字符在被指定为类中的成员时都会失去它们原来的含义
^	如果作为正则表达式的第一个字符，则表示匹配行的开始。在 awk 中匹配字符串的开始，即使字符串包含嵌入的换行符。
$	如果作为正则表达式的最后一个字符，则表示匹配行的结尾。在 awk 中匹配字符串的结尾，即使字符串包含嵌入的换行符。
\{n,m\}	匹配它前面某个范围内单个字符出现的次数（包括由正则表达式指定的字符）。\{n\}将匹配 n 次出现，\{n,\}至少匹配 n 次出现，而且\{n,m\}匹配 n 和 m 之间的任意次出现。（只有 sed 和 grep 的一些非常旧的版本中不能使用）。
\	转义随后的特殊字符。

运维技术书籍免费下载 https://www.linuxprobe.com

扩展的元字符（egrep 和 awk）

特殊字符	用途
+	匹配前面的正则表达式的一次或多次出现
?	匹配前面的正则表达式的零次或一次出现
|	指定可以匹配其前面的或后面的正则表达式（替代方案）
()	对正则表达式分组
{n,m}	匹配它前面某个范围内单个字符出现的次数（包括由正则表达式指定的字符）。{n} 表示匹配 n 次出现，{n,} 至少匹配 n 次出现，{n,m} 匹配 n 和 m 之间的任意次出现。（用于 POSIX 的 egrep 和 POSIX awk 而不是传统的 egrep 或 awk。）

a 大多数 awk 实现仍然不支持这种表示法。

元字符在正则表达式中有特殊的含义，与+和*在算术表达式中具有特殊含义的方式相同。有些元字符（+? ()|）只有作为由程序（例如 egrep 和 awk）使用的扩展集的一部分时才可用。在一下几节中我们将介绍每个元字符的用法，首先从反斜杠开始介绍。

普遍存在的反斜杠

元字符反斜杠（\）将元字符转换成普通字符（和将普通字符转换成元字符）。它强制将任意元字符解释为普通文字，以便匹配该字符本身。例如，句点（.）是元字符，如果想匹配句点，那么就需要用反斜杠对其进行转义。下面的正则表达式匹配由 3 个空格跟随的句点。

\.□□□

反斜杠常用于匹配以句点开始的 troff 请求或宏。

\.nf

还可以用反斜杠转义反斜杠。例如，troff 中的字体改变请求是“\f”。为了搜索包含这个请求的行，可以使用下面的正则表达式：

\\f

另外，sed 使用反斜杠将一组普通字符被解释为元字符，如图 3-2 所示。

在“\n”结构中的 n 表示从 1 到 9 之间的一个数字，在第五章“基本的 sed 命

运维技术书籍免费下载 https://www.linuxprobe.com

令”中将解释它的用法。

\(\) \{ \} \n

图 3-2：转义 sed 中的元字符

通配符

通配符元字符或者句点（.）被认为是与变量等价的。变量表示算术表达式中的任意值。在正则表达式中，句点（.）是代表除换行符以外的任意字符的通配符（在 awk 中，句点甚至可以匹配嵌入式换行符）。

假定我们正在描述一个字符序列，使用通配符元字符可以指定任何字符都可以填充的一个位置。

例如，如果要索索包含 Intel 系统微处理器的讨论文件，使用下面的正则表达式：

```
80.86
```

将匹配包含序列“80286，“、“80386，”或“80486.”（注 2）的行。为了匹配小数点或句点，必须用反斜杠转义点。

只匹配模式开头或结尾处的任何字符没有什么用。因此，通配符字符通常放在字面字符或其他元字符的前面或后面。例如，下面编写的正则表达式搜索 chapter 的序列：

```
chapter.
```

它搜索“‘chapter’后面跟有任意字符的字符串”。在某些搜索中，这个表达式有可能与固定的字符串模式“chapter”具有相同的匹配。请看下面的示列：

```
$ grep ’ Chapter.’  sample
you will find several examples in Chapter 9.
"Quote me ’ Chapter and Verse’," she said.
Chapter Ten
```

该示列所搜与“chapter”相匹配的字符串，而使用“chapter”也将匹配同样的行。然而，有一种不同的情况----如果“chapter”出现在行尾。因为通配符不匹配换行符，所以“chapter”不匹配那一行，而固定字符串模式“chapter”则匹配那一行。

注 2： pentium 系列微处理器打破了我们简单的模式匹配实验，破坏了其趣味性。更不必说最初的 8086 了。

运维技术书籍免费下载 https://www.linuxprobe.com

编写正则表达式

为了所有实用化的用途，你可以通过使用程序产生正确的结果。然而，并不意味着程序总是如你所愿的那样正确地工作。多数情况下，如果程序不能产生想要的输出，可以断定真正的问题（排除输入或语法错误）在于如何描述想要的东西。

换句话说，应该考虑纠正问题的地方是描述想要的结果的表达式。表达式不完整或者公式表示得不正确。例如，如果程序计算机下面的表达式：

```
PAY = WEEKLY_SALARY * 52
```

并知道这些变量的值，它将计算出正确的结果。但是有人会反对，因为公式没有说明销售人员，他也得到了一份佣金。所以为了描述这种情况，这个表达式需要重新用公式表示为：

```
PAY = WEEKLY_SALARY * 52 + COMMISSION
```

也许你会认为，编写第一个表达式的人没有完全理解问题所涉及的范围，因此不能很好地进行描述。知道如何详细的进行描述是很重要的。如果请求某人为你拿一本书，而且 如果摆在面前有很多书，那么就需要更加明确地描述你想要的书（或者满足与不确定的选择过程）。

这同样适用于正则表达式。程序（例如 grep）简单而且容易使用。理解正则表达式的元素也不难。正则表达式允许编写简单的或复杂的模式描述。而使编写正则表达式很难（和有趣）的因素是应用的复杂性：模式出现在各种不同的情况和上下文中。复杂性是语言本身所固有的，就像你并不总能通过查询字典来获得某个语义的正确理解一样。

编写正则表达式的过程涉及 3 个步骤：

1. 知道要匹配的内容以及它如何出现在文本中。
2. 编写一个模式来描述要匹配的内容。
3. 测试模式来查看它匹配的内容。

这个过程实质上与程序员开发程序的过程相似。步骤 1 可以当作规范，它反映理解要解决的问题以及如何解决它。步骤 2 类似于编写程序代码，而步骤 3 相当于运行程序并根据规范测试它。步骤 2 和步骤 3 需重复进行，知道程序令人满意为止。

对匹配描述进行测试可以确保这个描述和所期待的一样。它通常揭示一些令人惊奇的事。仔细检查测试的结果、比较输出和输入，这可以大大地提高对正则表达式的理解。可以按下面的方式解析模式匹配的结果：

Hits （击中）

运维技术书籍免费下载 https://www.linuxprobe.com

这是我想要匹配的行。

Misses （未击中）

这是我不想匹配的行。

Omissions （遗漏）

这是我不能匹配但想要匹配的行。

False alarms （假警报）

这是我不想匹配的但却匹配了的行。

尝试完善模式的描述也可以从对立段解决；可以试着通过限制可能的匹配来排错假警报，通常扩展可能的匹配来试着捕获遗漏。

当你必须使用固定的字符串描述模式时，困难尤其明显。从固定字符串的模式中删除每个字符都会增加可能的匹配数量。例如，当搜索字符串“what”时，还决定匹配“What”，那么同时匹配“What”和“what”的惟一的固定字符串模式是“hat”，即两者共用的最长的字符串。显然，搜索“hat”将产生不想要的匹配。给固定字符串模式添加每个字符都可能减少匹配数量。字符串“them”通常比字符串“the”产生的匹配更少。

在扩大和缩小匹配范围方面，在模式中使用元字符可以提供更大的灵活性。与字面值或其他元字符组合使用可以扩展匹配的范围，同时也减少了不想匹配的范围。

字符类

字符类是对通配符概念的改进。我们可以列出要匹配的字符，而不是匹配特殊位置的任意字符。使用方括号元字符（[]）将字符列表括起来，其中每个字符占据一个位置。

字符类在处理大写和小写字母时非常有用。例如，如果“what”可能以首字母大写或小写的形式出现，则可以指定：

```
[Ww]hat
```

这个正则表达式可以匹配“what”或“What”。它匹配包含这 4 个字符的字符串的任意行，第一个字符是“W”或“w”。因此，它可以匹配“Whatever”或“somewhat”。

如果一个文件中包含结构化的标题宏，例如：.H1，.H2，.H3,等等，那么可以用下面的正则表达式提取这些行中的任意一行：

```
\.H[12345]
```

这一模式匹配包含 3 个字符的字符串，最后一个字符是从 1 到 5 的任意数字。

UNIX shell 使用相同的语法，因此，可以使用字符类在 UNIX 命令中制定文件

运维技术书籍免费下载 https://www.linuxprobe.com

名。例如，为了从一组由章组成的文件中提取标题可能要输入：

```
$ grep ’ \.H[123]’  ch0[12]
ch01:.H1 "Contents of Distribution Tape"
ch01:.H1 "Installing the Software"
ch01:.H1 "Configuring the System"
ch01:.H2 "Specifying Input Devices"
ch01:.H3 "Using the Touch Screen"
ch01:.H3 "Using the Mouse"
ch01:.H2 "Specifying Printers"
ch02:.H1 "Getting Started"
ch02:.H2 "A Quick Tour"
.
.
```

注意必须用引号引住其中的模式，以便把它传递给grep而不只是由shell解释。由grep产生的输出为每个要打印的行都标识了文件名。作为字符类的另一个实例，假设想要指定不同的标点符合来结束句子：

```
.[!?;:,".] .
```

这个表达式匹配“任意后面有一个感叹号、问号、分号、冒号、逗号、引号或句点，随后是两个空格和任意一个字符的字符“。它可以用于寻找在一个句子的结束和另一个句子的开头之间有两个空格的地方（当一行中有这种情况时）。注意：在这个表达式中有3个点，第一个和最后一个点是通配符元字符，但是第二个点解释为字面意义。在方括号中、标准的元字符会失去它们的含义。因此，方括号中的点表示一个句点。表3-2列出了方括号中具有特殊含义的字符。

表3-2：字符类中的特殊字符

字符	功能
\	转义任意特殊字符（只用于awk中）
-	当它不在第一个或最后一个位置时，表示一个范围。
^	仅当在第一个位置时表示反转匹配

反斜杠只在awk中是特殊的，因此可以使用字符类“[a\]1]”以匹配一个a、一个]或一个1。

字符的范围

L连字符（-）用于指定一个字符范围。例如，所有大写英文字面（注3）的范围可以指定为：

运维技术书籍免费下载 https://www.linuxprobe.com

[A-Z]

一个数字的数字范围可以指定为：

[0-9]

该字符类有助于解决匹配文章引用的问题。请看下面的正则表达式：

[cC]hapter [1-9]

它匹配字符串“chapter”或“chapter”且其后面跟有空格，然后是从1到9的任意单个数字，下面的每一行都匹配这种模式：

you will find the information in chapter 9
and chapter 12.
Chapter 4 contains a summary at the end.

注3： 实际上当处理非ASCII字符集和/或非英语时，这会非常混乱。POSIX标准解决了这个问题，后面会看到新的POSIX特征。

根据这个任务，本例中的第二行可以看做是假警报，可以在“[1-9]”之后添加空格来避免匹配两个数字。也可以指定不在那个位置匹配的字符类，正如我们在下一节要看到的那样。可以同时指定多重范围，也可以混合使用字面字符和字符范围：

[0-9a-z?,.;:’”]

这个表达式将匹配“任意单个字符，可以是数字、小写字母、问号、逗号、句点、分号、冒号、单引号和引号”。记住每个字符类都匹配单个字符。如果指定多个类，可以描述多个连续的字符，例如：

[a-zA-Z][.?!]

这个表达式匹配“任意后面跟有句点、问号或感叹号的小写或大些字面”。

如果闭括号（]）是作为类中的第一个字符出现（或者是脱字符后的第一个字符，参见下一节），那么它就被解释为类的一个成员。如果连字符在一个类中是第一个或最后一个字符，则失去其特殊含义。因此，为了匹配算术操作符，我们在下面的示例中将连字符（-）放在第一位：

[-+*/]

在awk中，还可以使用反斜杠转义在范围中出现的连字符或闭方括号，但是语法

运维技术书籍免费下载 https://www.linuxprobe.com

更杂乱了。

尝试用正则表达式匹配日期是一个有趣的问题。下面是两种可能的格式：

```
MM-DD-YY
MM/DD/YY
```

下面的正则表达式指示每个字符位置可能的数值范围：

```
[0-1][0-9][-/][0-3][0-9][-/][0-9][0-9]
```

“-”或“/”都可能是定界符。在第一个位置放置连字符确保它在字符类中解释为字面意义，即作为一个连字符，而不是指示一个范围（注4）。

排除字符类

通常，字符类包括在哪个位置想要匹配的所有的字符。在类中作为第一个字符的脱字符（^）将类中的所有字符由排除在被匹配之外。相反，除换行符（注5）以外的没有列在方括号中的任意字符都将匹配。下面的模式将匹配任意非数字字符：

```
[^0-9]
```

它匹配字面表中所有的大写和小写字面以及所有特殊字符，例如标点符合。

排除特殊字符有时比显示地列出想要匹配的所有字符更方便。例如，如果想要匹配任意辅音，可以简单地排除元音：

```
[^aeiou]
```

该表达式匹配任意辅音，大写的任意元音，任意标点符合或特殊的字符。

请看下面的正则表达式：

```
\.DS "[^1]"
```

该表达式匹配字符串“.DS”其后依次跟随一个空格、一个双引号、一个非数字和一个双引号（注6）。这样设置是为了避免匹配下面的行：

```
.DS "1"
```

而匹配这样的行，例如：

```
.DS "I"
.DS "2"
```


运维技术书籍免费下载 https://www.linuxprobe.com

这种语法还可以用来限制匹配的范围，正如随后要看到的那样。

注4： 注意这个表达式匹配以指定定界符分隔的日期表达式，即使是“15/32/78”这样不可能的日期。
注5： 在awk中，换行符也可以被匹配。
注6： 当在命令行结尾处键入这种模式时，务必用单引号包围它。脱字符^对最初的Bourne shell是特殊的。

POSIX字符类补充

POSIX标准对正则表达式字符和操作符的含义进行了形式化。这种标准定义了两类正则表达式：基本的正则表达式（BRE），grep和sed使用这种正则表达式;扩展的正则表达式，egrep和awk使用这种正则表达式。

为了适应非英文的环境，POSIX标准增强了匹配不在英文字母表中的字符的字符类的功能。例如，法文è是一个字母字符，但是使用典型的字符类[a-z]不匹配它。该标准提供了附加的字母序列，当匹配和整理（排序）字符串数据时，这些字符序列应该被作为单个单元看待。

POSIX还改变了常用的术语，我们一直称为“字符类”的东西在POSIX标准中称为“括号表达式”。在括号表达式中，除了字面字符（例如a,!等等）以外，还可以由其他标记。如下：

- 字符类。由[:和:]包围的关键字组成的POSIX字符类。关键字描述了不同的字符类，例如，文字字符，控制字符等等（参见表3-3）。
- 整理符合。整理符合是多字符的序列。表示这些字符应该被看做是一个单元，它由[,和.]包围的字符组成。
- 等价类。等价类列出了应该看做是等价的字符集，例如e和è。它由地区化的字符元素（由[=和=]包围）组成。

所有的这3中结果都必须出现在括号表达式的方括号中。例如[[:alpha:]!]匹配任意单个字母字符或感叹号，[[.ch.]]匹配整理元素ch，但不只匹配字母c或字母h。在法语地区中，[[=e=]]可以匹配任意e、è或é。表3-3列出了类及其匹配字符。

表3-3 POSIX字符类

类	匹配字符
[:alnum:]	可打印的字符（包括空白字符）
[:alpha:]	字母字符
[:blank:]	空格和制表符

运维技术书籍免费下载 https://www.linuxprobe.com

[:cntrl:]	控制字符
[:digit:]	数字字符
[:graph:]	可打印的和可见的（非空格）字符
[:lower:]	小写字符
[:print:]	可打印的字符（包括空白字符）
[:punct:]	标点符号字符
[:space:]	空白字符
[:upper:]	大写字符
[:xdigit:]	十六进制数字

当厂商完全实现了POSIX标准时，这些特征逐渐向sed和awk的商业版接近。GNU awk和GNU sed支持字符类符号，但不支持另外两个括号符号。可以检查本地系统文档来查看它们是否可用。

因为这些特征还不能被广泛地应用，本书的脚本不依赖它们，而且我们要继续使用术语“字符类”来表示方括号中的字符表。

重复出现的字符

星号（*）元字符表示它前面的正则表达式可以出现零次或多次。也就是说，如果它修改了单个字符，那么该字符可以在那里也可以不在那里，并且如果它在那里，那可能会不止出现一个。可以使用星号元字符匹配出现在引号中的单词。

```
□"*hypertext"*□
```

不管单词“hypertext”是否出现在引号中都会被匹配。

而且，如果由星号修饰的字面字符确实存在，那么有可能出现多次。例如，我们来看一系列数字：

```
1
5
10
50
100
```


运维技术书籍免费下载 https://www.linuxprobe.com

500
1000
5000

正则表达式

[15]0*

将匹配所有的行，而正则表达式

[15]00*

匹配除前两行以外的所有的行。第一个0是字面值，但是第二个由星号修饰，意味着它可能出现也可能不出现。常用类似的方法匹配一个或多个（而不是灵个或多个）空格，可以使用下面的表达式来完成：

□□^

当星号元字符前面由句点元字符时，biao是匹配任意数目的字符。这可用于标识两个固定的字符串之间的字符的跨度。如果想要匹配引号中的任意字符串，可以指定：

".*"

它匹配该行上的第一个引导和最后一个引导之间的所有字符以及引号。使用“.*“进行匹配的范围总是最大的。目前它似乎并不重要，但是一旦学习替换被匹配的字符串时这就很重要了。

作为另一个例子，一对尖括号时标记语言中用来包围格式化指令的普通符号，标记语言如SGML、HTML和Ventura Publisher。

通过指定下面的表达式可以打印带有标记的所有的行：

$ grep ’<.*>’ sample

当星号用于修饰字符类时，则可以匹配类中的任意数目的字符。例如下面5行的示例文件：

I can do it
I cannot do it
I can not do it
I can’t do it
I cant do it

运维技术书籍免费下载 https://www.linuxprobe.com

如果我们想要匹配以上语句中的否定语句，但不匹配肯定语句，可以使用下面的正则表达式：

```
can[ no’ ]*t
```

星号使得类中的任意字符以任意顺序匹配，并且匹配任意多次的出现。如下所示：

```
$ grep "can[ no’ ]*t" sample
| cannot do it
| can not do it
| can’ t do it
| cant do it
```

有4个成功和1个失败（肯定语句）。注意如果正则表达式试图匹配字符串“can”和“t”之间的任意个字符，如下例所示：

```
can.*t
```

它将匹配所有的行。

技术术语“closure（闭合）”有匹配“零次或多次”的能力。egrep和awk使用的元字符扩展集提供了几个非常有用的closure的变化。加号（+）匹配前面的正则表达式的一次或多次出现。前面匹配一个或多个空格的示例可以简化为：

```
□*
```

元字符加号可以被认为是“至少一个”的前导字符。事实上，它和许多人使用的“*”号相对应。

问好（？）匹配零次或一次出现。例如，在前面的示例中，我们使用正则表达式匹配“80286”、“80386”和“80486”。如果我们还想匹配字符串“8086”，可以用egrep或awk编写正则表达式：

```
80[234]?86
```

它匹配“80”后面跟有一个“2，”，一个“3，”，一个“4，”或者没有字符，然后跟字符串“86.”。不要混淆表达式中的？和shell中的？通配符。Shell中的？表示单个字符，等效于正则表达式中的“.”。

单词是什么？第一部分

也许你已经发现，有时匹配完整的单词很难。例如，如果想匹配模式“book”，搜索会命中包含单词“book”和“books”的行，而且还有单词“bookish”、“handbook”和“booky”。很显然可以在“book”前后使用空

运维技术书籍免费下载 https://www.linuxprobe.com

格来限制匹配情况。

□book□

然而，这个表达式只匹配单词“book”，它会丢到它的复数形式“books”。为了匹配单数或复数单词，可能要使用星号元字符：

□books*□

这样就可以匹配“book”或“books”。然而，如果单词后面有句点、逗号、问好或引号时就不会匹配“book”。

当将星号和通配符元字符（.）结合起来使用时，可以匹配任意字符的零次或多次出现。在前面的示例中，可以像下面这样编写比较完整的正则表达式：

□book.*□

这个表达式匹配字符串“book”，其后面跟有任意个字符或没有字符，最后跟着空格。下面是将要匹配的几行：

```
Here are the books that you requested
Yes, it is a good book for children
It is amazing to think that it was called a "harmful book" when
once you get to the end of the book, you can’t believe
```

（注意只有第二行可以和固定字符串“□book□”匹配。）表达式“□book.*□”匹配包含类似与单词“booky”、“bookworm”和“bookish”的行。通过使用不同的修饰符可以排除这些匹配中的两种。问号(?)是元字符的扩展集的一部分，匹配前面表达式的0次或1次出现。因此，表达式：

□book.?□

将与“book，”、“books，”和“booky”匹配，而不与“bookish”和“bookworm”匹配。为了排除类似“booky”这样的单词，我们可以使用字符类来指定想要匹配的位置的所有字符。而且，因为元字符问号在sed中不可用，我们必须求助于字符类，这些在后面将会看到。

尝试用正则表达式包括一切并不实际，尤其是在使用grep时，有时最好使表达式保持简单并允许遗漏。然而，当在sed中使用正则表达式进行替换时，就需要注意的是使用的表达式应该完整。在本章“单词是什么？第二部分”中我们将会看到用于搜索单词更全面的正则表达式。

定位元字符

有两个元元字符用于指定字符串出现在行首或行末的上下文。脱字符（^）元字

运维技术书籍免费下载 https://www.linuxprobe.com

符是指示开始的单字符正则表达式。美元符号（$）元字符是指示行结尾的单字符的正则表达式。这些通常称为“定位符”，因为它们将匹配限定在特定位置。例如，可以使用以下表达式打印以制表符开始的行：

```
^•
```

(•表示可见的制表符，它实际上是不可见的。)没有^元字符，这个表达式将打印包含制表符的任意行。

通常，使用vi输入要由troff处理的文本，并且不想让空格出现在行的结尾。如果想找到（并删除）它们，下面的正则表达式可以匹配在结尾处有一个或多个空格的行：

```
□□*$
```

Troff请求和宏必须在行的开始处输入。它们是两个字符的字符串，前面带有一个句点。如果请求或宏有一个参数，那么它通常后面跟有一个空格。用于搜索这样的请求的正则表达式是：

```
^\...□
```

这个表达式匹配“行首有一个句点，随后跟有两个字符的字符串，然后是一个空格的行”。

可以使用两个连续的定位元字符来匹配空行，即：

```
^$
```

可以使用这种模式计算文件中的空行数，在grep中使用计数选项-c：

```
$grep -c '^$' ch04
5
```

如果想使用sed来删除空行，那么这个正则表达式很有用。下面的正则表达式可用于匹配空行，即使其中包含空格：

```
^□*$
```

同样，可以使用以下表达式匹配整个行：

```
^.*$
```

这可能是你想使用sed来处理的事情。

运维技术书籍免费下载 https://www.linuxprobe.com

在sed（和grep）中，只有当“^”和“$”分别出现在正则表达式的开始或结尾时才是特殊的。因此“^abc”意味着“匹配只处于行的开始处的字母a、b和c”，而“ab^c”意味着“匹配处于行的任意位置的a、b、字面^，然后是c”。这对于“$”同样使用。

在awk中则不同，“^”和“$”总是特殊的，即使它们可能使编写的正则表达式不匹配任何东西。可以说，在awk中，当想要匹配字面“^”或“$”时，不管它处于正则表达式的什么位置都应该用反斜杠对其进行转义。

短语

模式匹配的程序（例如grep）不能匹配跨两行的字符串。由于所有特殊的目的，要保证匹配短语是困难的。记住，文本文件基本上是无结构的，而且换行位置是随即的。如果要寻找单词的任意序列，它们也许出现在一行上，但也许被分成两行。

```
Almond Joy
Almond$
^Joy
```

这并不完美，因为第二个正则表达式匹配行结尾处的“Almond”，而不管下一行是否以“Joy”开始。第三个正则表达式也存在同样的问题。

稍后将在介绍sed时，介绍如何匹配多行模式，并且介绍一个shell脚本，它与sed结合使用，提供了一种通用的方式来解决该问题。

字符的跨度

这个元字符指示了一个不确定的长度，这允许你指定重复出现的字符。考虑下面的表达式：

```
11*0
```

它将匹配下面的每一行：

```
10
110
111110
11111111111111111111111111110
```

这些元字符使正则表达式具有了伸缩性。

现在我们来看一对用于指定跨度并决定跨度长度的元字符。可以指定一个字母字符或正则表达式出现的最小或最大次数。

运维技术书籍免费下载 https://www.linuxprobe.com

在grep和sed中使用\{和\}(注7)。POSIX egrep和POSIX awk使用｛和｝。在任何情况下，大括号包围一个或两个参数。

\｛n,m\｝

n和m是0到255之间的整数。如果只指定\{n\}本身，那么将精确匹配前面的字符或正则表达式的n次出现。如果指定\{n,m\}，那么就匹配出现的次数为n和m之间的任意数。（注8）

注7： 非常旧的版本可能没有它们； Caveat emptor。
注8： 注意“？”等价于“\{0,1\}”，“*”等价于“\｛0，\｝”，“+”等价于“\｛1，\｝”，没有等价于“\｛1\｝”的修饰符。

例如，下面的表达式将匹配“1001”，“10001”和“100001”、但是不匹配“101”或“1000001”：

10\{2,4\}1

这对元字符对于匹配固定长度字段中的数据非常有用，数据可能是从数据库中提前的，它也用于匹配格式化数据，例如电话号码，U.S社会保险号，库存零件ID等等。例如，社会保险号的格式为：3个数字，一个连字符，再跟2个数字，一个连字符，然后是4个数字。可以描述为一下模式：

[0-9]\{3\}-[0-9]\{2\}-[0-9]\{4\}

类似地，北美地区的电话号码可以用下面的正则表达式描述：

[0-9]\{3\}-[0-9]\{4\}

如果使用POSIX之前的awk，大括号就不可用，只能简单地重复适当次数的字符类：

[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]

选择性操作

竖线（|）元字符是元字符扩展集的一部分，用于指定正则表达式的联合。如果某行匹配其中的一个正则表达式，那么它就匹配该模式。例如，正则表达式：

UNIX|LINUX

将匹配包含字符串“UNIX”或字符串“LINUX”的行。可以指定更多的选择，例如：

UNIX|LINUX|NETBSD

运维技术书籍免费下载 https://www.linuxprobe.com

表示使用egrep打印匹配这3种模式中任意一种的行。

在sed中，没有联合元字符，可以分别指定每种模式。在下一节中，我们将考虑分钟的操作，我们将看到关于这个元字符的其他示列。

分组操作

圆括号（）用于对正则表达式进行分组并设置优先级。它们是元字符扩展集的一部分。假设在文本文件中将公司的名称为“BigOne”或“BigOne Computer”，使用表达式：

```
BigOne( Computer)?
```

将匹配字符串“BigOne”本身或后面跟有一个字符串“□Computer”的形式。同样，有些术语有时会用全拼，有时会用缩写、则可以使用：

```
$ egrep "Lab(oratorie)?s" mail.list
Bell Laboratories, Lucent Technologies
Bell Labs
```

可以使用竖线和圆括号来对选择性操作进行分组。在下面的示列中，我们使用它来指定与单词“company”的单数或复数匹配。

```
compan(y|ies)
```

要注意，在大多数sed和grep的版本中不能对加圆括号的一组字符应用数量词，但是在egrep和awk的所有版本中都是可以的。

单词是什么？第二部分

我们来使用我们所讨论的新的元字符，来重新构建搜索单个单词的正则表达式。首先我们试着为grep编写一个如下的正则表达式，用它来搜索单词：

```
□book.*□
```

这个表达式非常简单，它匹配的模式为：一个空格后面是字符串“book”，再后面是任意数量的字符和一个空格。然而，它不能匹配所有可能的出现并且它还匹配了几个错误单词。

下面的测试文件包含许多“book”。我们添加一个符号，它不属于文件的部分，用它来指示输入行是“击中”（>）并且包括在输出中，还是“未击中”（<）。这里包括了尽可能多的不同示列。

```
$ cat bookwords
> This file tests for book in various places, such as
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
> book at the beginning of a line or
> at the end of a line book
> as well as the plural books and
< handbooks. Here are some
< phrases that use the word in different ways:
> "book of the year award"
> to look for a line with the word "book"
> A GREAT book!
> A great book? No.
> told them about (the books) until it
> Here are the books that you requested
> Yes, it is a good book for children
> amazing that it was called a "harmful book" when
> once you get to the end of the book, you can't believe
< A well-written regular expression should
< avoid matching unrelated words,
< such as booky (is that a word?)
< and bookish and
< bookworm and so on.
```

当我们搜索单词“book”的出现时，应该有13行匹配，7行不匹配。首先，我们对示列文件执行以前的正则表达式并检查结果。

```
$ grep ' book.* ' bookwords
This file tests for book in various places, such as
as well as the plural books and
A great book? No.
told them about (the books) until it
Here are the books that you requested
Yes, it is a good book for children
amazing that it was called a "harmful book" when
once you get to the end of the book, you can't believe
such as booky (is that a word?)
and bookish and
```

它只打印我们想要匹配的13行中的8行，并打印我们不想匹配的行中的2行。这个表达式匹配包含单词“booky”和“bookish”的行。它忽略了行开始和结尾处的“book”。当涉及某种标点符号时它忽略“book”。

为了进一步限制搜索，我们必须使用字符类。一般地，可以结束单词的字符列表是标点符号，例如：

```
? . , ! ; : '
```


运维技术书籍免费下载 https://www.linuxprobe.com

另外，引号、圆括号、大括号和方括号可以包围一个单词出现在单词的左侧或右侧：

```
" () {} []
```

你还必须调整单词的复数或所有格形式。

因此，应该有两个不同的字符类：单词之前和单词之后。记住我们必须做的就是列出方括号中的类的成员。在单词前面使用：

```
["[{(]
```

在单次后面使用：

```
[]})"?!.,;:' s]
```

要注意，在类中第一个位置放置闭方括号，表示它是类的成员而不是关闭括号。将以上两个类放在一起，我们得到下面的表达式：

```
□["[{(]*book[]})"?!.,;:' s]*□
```

对初学者显示这些内容，会使他们因为绝望而放弃！但是在了解了有关的原理之后，就不仅可以理解这个表达式，而且能很容易地重构它。让我们看看它在示列文件中是如何运行的（我们使用双引号引住单引号字符，然后在嵌入的双引号的前面放置一个反斜杠）：

```
  $ grep " [\"[{(]*book[]})\"?!.,;:' s]* " bookwords
This file tests for book in various places, such as
as well as the plural books and
A great book? No.
told them about (the books) until it
Here are the books that you requested
Yes, it is a good book for children
amazing that it was called a "harmful book" when
once you get to the end of the book, you can't believe
```

我们排除了不想要的行，但是有4行我们没有得到。让我们来检查这4行：

```
  book at the beginning of a line or
at the end of a line book
"book of the year award"
A GREAT book!
```

所有这些都是由在行的开始处或结尾处出现的字符串所导致的错误。因为在行

运维技术书籍免费下载 https://www.linuxprobe.com

的开始处或结尾处没有空格。所以模式不被匹配。可以使用定位元字符^和$。因为要匹配一个空格或行的开始或结尾，可以使用egrep并指定的“或”元字符，同时用圆括号分隔。例如，为了匹配行的开始或一个空格，可以编写下面的表达式：

```
(^| )
```

（因为|和（）是元字符的扩展集的一部分，所以如果使用的是sed，则必须编写不同的表达式来处理每一种情况。）

下面是修订过的正则表达式：

```
(^| )["[{(]*book[]})"?\!.,;:' s]*( |$)
```

现在我们来看看它如何工作：

```
  $ egrep "(^| )[\"[{(]*book[]})\"?\!.,;:' s]*( |$)" bookwords
This file tests for book in various places, such as
book at the beginning of a line or
at the end of a line book
as well as the plural books and
"book of the year award"
to look for a line with the word "book"
A GREAT book!
A great book? No.
told them about (the books) until it
Here are the books that you requested
Yes, it is a good book for children
amazing that it was called a "harmful book" when
once you get to the end of the book, you can't believe
```

这确实是一个复杂的正则表达式，然而，可以将它分成几个部分。这个表达式也许不匹配每个单一的实例，但是可以很容易对它进行改写来处理其他情况。

你也可以创建一个简单的shell脚本，用命令行参数取代“book”。惟一的问题是单词的复数并不总是“s”。采用手工修改的方法，可以通过给单词后面的字符类添加“e”来处理“es”复数，它在许多情况下都可以工作。

需要进一步注意的是，ex和vi文件编辑器使用特殊的元字符\<和\>，分别匹配单词开始处和结尾处的字符串。当它们作为一对使用时，它们只和那些是完整单词的字符串匹配（对于这些操作符，单词是指非空格且两侧为空格的字符串，或者指在行的开始或结尾处的字符串）。如果这些元字符可用于所有的正则表达式，那么这些元字符可以得到广泛的应用，因为匹配单词是一种非常普

运维技术书籍免费下载 https://www.linuxprobe.com

遍的操作（注9）。

在这里替换

当使用grep时，只要能匹配就行，至于如何匹配很少会有问题。然而，当想要进行替换时，就必须考虑匹配的范围。那么，行上的什么字符是实际上匹配的呢？

在本节中，我们将看几个演示匹配范围的示列。然后我们将使用类似grep的一个程序，但它还允许指定替换字符串。最后，我们将看几个用于描述替换字符串的元字符。

匹配的范围

让我们来看下面的正则表达式：

```
A*Z
```

这个表达式匹配“零次或多次出现的A，同时A后面跟字符Z”。它产生的结果与仅仅指定“Z”是相同的。字符“A”可能在那里也可能不在那里，事实上，字符“Z”是惟一匹配的字符。下面是一个两行的示列文件：

```
All of us, including Zippy, our dog
Some of us, including Zippy, our dog
```

如果我们尝试匹配前面的正则表达式，那么就会打印两行。有缺的是，这两种情况下的真正的匹配都作用于“Z”而且只有“Z”。我们可以使用gres命令（参见后面“生成单个替换的程序”）演示匹配的范围。

```
$ gres "A*Z" "OO" test
All of us, including OOippy, our dog
Some of us, including OOippy, our dog
```

注9： GNU程序，例如awk的GUN版本，sed和grep也都支持\<和\>。

生成单个替换的程序

MKS工具包（又Mortice Kern Systems公司开发的用于DOS的一组UNIX使用工具）包含一个被成为gres（全局的正则表达式替换）的非常有用的程序。就像grep一样，它在文件中搜索一个模式，然而，它允许要匹配的字符串指定一个替换。这个程序事实上是sed的简化版本、并且和sed一样，它打印所有的行而不管是否对其执行了一个替换。它不对文件本身进行替换。如果想保存更改结果，必须将程序的输出重定向到一个文件。

gres不是标准UNIX的一部分，但它是非常好的工具。它可以使用简单的shell脚本来创建，并通过调用sed完成相应的功能。

运维技术书籍免费下载 https://www.linuxprobe.com

```
#!/bin/sh
$ cat gres
if [ $# -lt 3 ]
then
echo Usage: gres pattern replacement file >&2
exit 1
fi
pattern=$1
replacement=$2
if [ -f $3 ]
then
file=$3
else
echo $3 is not a file. >&2
exit 1
fi
A="‘echo | tr ’ \012’ ’ \001’ ‘" # （注10）
sed -e "s$A$pattern$A$replacement$A" $file
```

本章的其余部分将使用gres来演示替换元字符的用法。记住，应用于gres的任何操作都可以应用于sed。这里我们用两个零（00）来替换被正则表达式“A.*Z”匹配的字符串。

```
$ gres "A.*Z" "00" sample
00ippy, our dog
00iggy
00elda
```

注10 ： echo|tr...行虽然较复杂，但是生成用作sed替换命令的分隔符Control-A很方便。这样做大大地减少了分隔符出现在模式中或替换文本中的机会。

我们期望第一行上的匹配范围扩展为从“A”到“Z”，而不仅是“Z”真正被匹配。如果我们稍微改动正则表达式，结果就会很明显：

A.*Z

“.*”可以被解释为出现零次或多次的任意字符，这意味着可以找到“出现任意次的字符”，包括什么也没有的情况。整个表达式表示A和Z之间有任意数目的字符；“A”是模式中的开始字符，“Z”是最后一个字符；在它们之间可以有任意多的字符或没有字符。在有以上两行的文件上运行grep会产生一行的输出。我们在所匹配的内容下面添加一行脱字符（^）来标记所匹配的内容。

运维技术书籍免费下载 https://www.linuxprobe.com

```
All of us, including Zippy, our dog
^^^^^^^^^^^^^^^^^^^^^^^
```

匹配的范围从“A”到“Z”。相同的正则表达式还匹配下面的行：

```
I heard it on radio station WVAZ 1060.
                                ^^
```

字符串“A.*Z”匹配A后面跟有任意个字符（包括零个字符）再跟有Z的模式。现在，让我们看看类似的一组包含“A”和“Z”的多次出现的示列行。

```
All of us, including Zippy, our dog
All of us, including Zippy and Ziggy
All of us, including Zippy and Ziggy and Zelda
```

正则表达式“A.*Z”在每种情况下都匹配可能为最长的范围。

```
All of us, including Zippy, our dog
^^^^^^^^^^^^^^^^^^^^^^^
All of us, including Zippy and Ziggy
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
All of us, including Zippy and Ziggy and Zelda
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

这样，当我们想要匹配最短的范围时会出问题。

限制范围

前面我们说过，正则表达式尝试匹配最长的字符串，并且这可能会引起意想不到的问题。例如，查看以下正则表达式，它与引号中的任意个字符匹配：

```
".*"
```

让我们来看有两个用引号包围的参数的troff宏，如下所示：

```
.Se "Appendix" "Full Program Listings"
```

要匹配第一个参数，可以用下面的正则表达式来描述匹配的模式：

```
\.Se ".*"
```

然而，因为模式中的第二个引号与该行上的最后一个引号匹配，所以它结束匹配整个行。如果知道参数的个数，那么可以对每个进行说明：

```
\.Se ".*" ".*"
```


运维技术书籍免费下载 https://www.linuxprobe.com

虽然它可以像所期望的那样工作，但是每行也许不会有相同数目的参数，省略你只想要第一个参数。这里有一个匹配两个引号之间最短范围的正则表达式：

```
"[^ "]*"
```

它匹配引号并且后面跟有任意个字符的情况，但它不匹配引号后面跟有引号的情况：

```
$ gres ' "[^ "]*"' ' 00' sampleLine
.Se 00 "Full Program Listings"
```

现在我们来看看在两列数字之间使用句点字符（.）作为引导符的几行：

```
1........5
5........10
10.......20
100......200
```

这里匹配引导字符的困难是它们的数量是可变的。假如想用单个制表符取代所有的引导符。则可以按下面的形式编写一个匹配行的正则表达式：

```
[0-9][0-9]*\.\.*[0-9][0-9]*
```

这个表达式也许会出乎意料地匹配下面的行：

```
see Section 2.3
```

为了限制匹配，你可以指定所有行共用句点的最小数目：

```
[0-9][0-9]*\.\{5,\}[0-9][0-9]*
```

这个表达式使用sed中可用的大括号来匹配“一个数字后面至少跟有5个句点，然后又跟有一个数字”的情况。为了查看执行过程，我们将给出一个sed命令，用一个连字符代替引导符中的句点。然而，我们没有介绍sed的替换元字符的语法——\（和\）用于保存正则表达式的一部分，而\1和\2用于回调保存的部分。因此，这个命令看上去相当复杂（确实如此!）,但它可以完成工作。

```
$ sed ' s/\([0-9][0-9]*\)\.\{5,\}\([0-9][0-9]*\)/\1-\2/' sample
1-5
5-10
10-20
100-200
```

可以编写一个类似的表达式，匹配一个或多个前导制表符或两列数据之间的制

运维技术书籍免费下载 https://www.linuxprobe.com

表符。可以改变列的顺序和使用另一个定界符取代制表符。你可以使用sed或gres通过自己编写简单的和复杂的替换进行检验。

使用喜欢的元字符

表3-4列出了正则表达式的有趣的示列，其中许多已经在本章进行了介绍：

表3-4：有用的正则表达式

项目	正则表达式
州的邮政缩写	□[A-Z][A-Z] □
城市、州	^ .*, □[A-Z][A-Z]
城市、州、邮编（POSIX egrep）	^ .*, □[A-Z][A-Z] [0-9]{5}(-[0-9]{4})?
月、日、年	[A-Z][a-z]\{3,9\}□[0-9]\{1,2\},□[0-9]\{4\}
美国社会保险号	[0-9]\{3\}-[0-9]\{2\}-[0-9]\{4\}
北美地区电话	[0-9]\{3\}-[0-9]\{4\}
格式化的美元数额	\$[□0-9]*\.[0-9][0-9]
troff嵌入的字体请求	\\f[(BIRP]C*[BW]*
troff请求	^ \.[a-z]\{2\}
troff宏	^ \.[A-Z12].
带有参数的troff宏	^ \.[A-Z12].□”.*”
HTML嵌入的代码	<[^ >]*>
Ventura Publisher Style Codes	^ @.*□=□.*
匹配空行	^ $
匹配整行	^ .*$
匹配一个或多个空格	□□*

运维技术书籍免费下载 https://www.linuxprobe.com

本章内容

- 在脚本中应用命令
- 寻址上的全局透视
- 测式并保存输出
- sed 脚本的 4 种类型
- 开始 PromiSed Land

第四章

编写 sed 脚本

为了使用 sed，首先要编写一个含有一系列编辑操作的脚本，然后在某个输入文件上运行它。使用 sed 可以将类似于 vi 编辑器中手动的操作过程提取出来，并转换成一个非手动的过程，即通过执行一个脚本来实现。

手动进行编辑工作时，可以通过输入一个编辑命令并观察立即出现的结果来检查因果关系。通常有一个“undo”命令用于撤销一个命令的影响，并将文本文件返回到它的前一个状态。一旦学会了交互式文本编辑器，就可以采用安全的并可控制的方式，来体验一次一步进行改变的感觉。

大多数不熟习 sed 的人都觉得，编写执行一系列编辑动作的脚本，比手动做一些改动更冒险。这种担心的原因是自动化任务会发生一些不可逆转的事情。学习 sed 的目标就是很好地理解它从而可以预测执行结果。换句话说，你将逐渐理解编辑的脚本与得到的输出之间的因果关系。

这就要求采用可控制的，有秩序的方式来使用 sed。在编写脚本时，应遵循以下这些步骤：

1. 在着手做之前要弄清楚想做什么。
2. 明确地描述处理的过程。
3. 在提交最终的改变之前反复测试这个过程。

这些步骤只是第三章“了解正则表达式语法”中介绍编写正则表达式的过程的重述。它们描述了编写任意种类的程序的方法论。检测脚本是否工作的最好的方式，是使用不同的输入样本进行测试并观察结果。

稍微实践一下，你就会信赖你的 sed 脚本，它将会做你想让它做的工作（这有点类似于个人的时间管理，学会将某些任务委托给别人去做。你可以将小的任务委托给别人，如果成功了，再将大的任务委托给它们）。

因此，本章可以使你自由地编写处理编辑工作的脚本。这就要了解 sed 工作的 3 个基本的原理：

- 脚本中的所有编辑命令都将依次应用与每个输入行。

运维技术书籍免费下载 https://www.linuxprobe.com

- 命令应用于所有的行（全局的），除非寻址限制了受编辑命令影响的行。
- 原始的输入文件未被改变，编辑命令修改了原始行的备份并且此备份被发送到标准输出。

介绍了这些基本原理之后，我们将看一下不同的 sed 应用的 4 种类型的脚本。这些脚本提供了你将要编写的脚本的基本模型。虽然在 sed 中可以使用许多命令变量，但本章中的脚本将有针对性地使用几个命令。虽然如此，你仍可能会对用这么少的命令能处理这么多事情而感到惊奇（第五章“基本 sed 命令”和第六章“高级 sed 命令”分别给出了基本的和高级的 sed 命令）。这种想法是在探讨脚本中可使用的所有命令之前，首先了解脚本如何工作以及如何使用脚本。

在脚本中应用命令

将一系列编辑组合进一个脚本中会出现意想不到的结果。你可能考虑不到一个编辑操作会对另一个编辑操作产生什么影响。新用户一般会认为，sed 在应用下一个编辑命令之前，先将一个单独的编辑命令应用与输入的所有的行。但事实正好相反。Sed 首先将整个编辑脚本应用于第一个输入行，然后再读取第二个输入行并对其应用整个脚本。引文 sed 总是处理原是行的最新形式，所以生成的任何编辑工作都会改变后续命令应用的行。Sed 不会保留最初的行。这意味着与原始输入行匹配的模式可能不再与经过编辑操作之后的行匹配。

我们将看到使用替换命令的示列。假设有人快速编写了下面的脚本，来将“pig”换成“cow”并将“cow”换成“horse”。

```
s/pig/cow/g
s/cow/horse/g
```

你认为会发生什么呢？用一个样本文件试一下。在了解了 sed 是如何工作的之后，我们将讨论所发生的事情。

模式空间

sed 维护一种模式空间，即一个工作区或临时缓冲区，当应用编辑命令时将在那里存储单个输入行（注 1）。图 4-1 展示了进行模式空间转换的一个两行的脚本。它将“The Unix system”改变为“The UNIX Operating System”。

运维技术书籍免费下载 https://www.linuxprobe.com

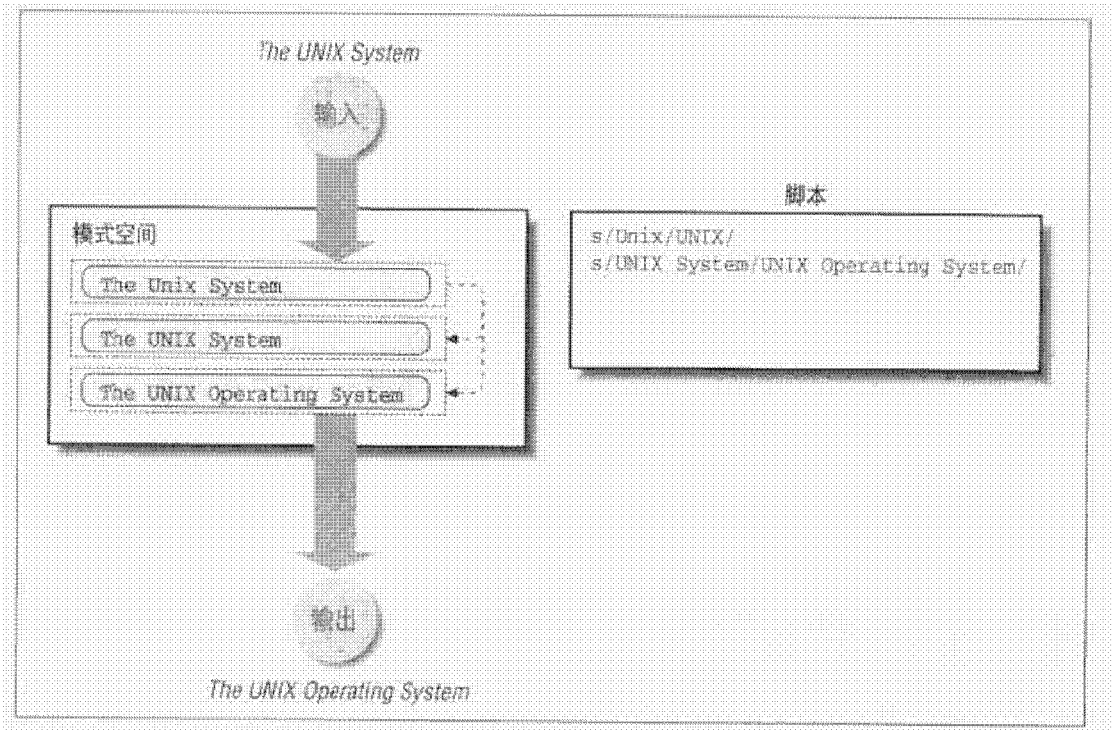

图 4-1： 脚本中的命令改变了模式模式空间的内容

注 1： 一次一行的设计的一个优点是 sed 在读取非常庞大的文件时不会出现问题。屏幕编辑程序必须将整个文件（或者它的一些庞大的部分）读入内存，这将会产生内存溢出或者在处理庞大的文件时速度非常慢。

初始时，模式空间包含有单个输入行的备份。在图 4-1 中是“The Unix System”行。脚本中正常的流程是在这一行上执行每个命令知道脚本达到末尾。脚本中的第一个命令应用于这一行，将“Unix”换成“UNIX”。然后应用第二个命令，将“UNIX System”换成“UNIX Operating system”(注 2)。注意第二个替换命令的模式不匹配最初的输入行，它匹配模式空间中发生了变化的当前行。

当应用了所有的指令后，当前行被输出并且输入的下一行被读入模式空间。然后脚本中的所有的命令应用与新读入的行。

结果是，任何 sed 命令都可以为应用下一个命令改变模式空间的内容。漠视空间的内容是动态的，而且并不总是匹配最初的输入行。这就是本章开始处的示列脚本中的问题。正如所希望的那样，死一个命令将“pig”换成“cow”。然而，当第二个命令在统一行上将“cow”换成“horse”时，它还改变了有“pig”换成“cow”。所以，在输入文件中包含 pig 和 cow，而输出文件中只包含 horse！

这个错误只是脚本中命令的顺序的问题，反转命令的顺序——在将“pig”换成“cow”之前将“cow”换成“horse”——这就是技巧。

运维技术书籍免费下载 https://www.linuxprobe.com

```
s/cow/horse/g
s/pig/cow/g
```

一些 sed 命令改变了整个标本的流程，正如我们在后面的章节中将会看到的那样。例如，N 命令将另一个读入读入模式空间但没有删除当前的行，所以可以测试跨越多行的模式。其他一些命令告诉 sed，在到达脚本的底部之前退出或者转到带标记的命令。Sed 还维护了称为保持空间（hold space）的另一个临时缓冲区。可以将模式空间的内容复制到保持空间并在以后检索它们。使用保持空间的命令将在第六章讨论。

注 2： 是的，我们可在第一步就将“Unix System”换成”UNIX Operating system”，然而，输入文件也许包含“UNIX System”和“Unix System”两种情况，因此通过将“Unix”换成“UNIX”，从而再将它们换成“UNIX Operating System”之前可以使两种情况保持一致。

寻址上的全局透视

要注意的第一件事是 sed 将命令应用于每个输入行。与 ed、ex 或 vi 不同，sed 是隐式全局的。下面的替换命令将每个“CA”都换成“California”。

```
s/CA/California/g
```

如果在 vi 中从 ex 命令提示符后输入相同的命令，那么它将只替换当前行中的所有出现。在 sed 中，好像每一行都会成为一次当前行，因此命令可以应用于每一行。行地址用于提供操作（或限制）的上下文环境（简而言之，在 vi 中除非告诉它对哪一行操作，否则它什么事情也不会做；在 sed 将处理每一行，除非你告诉它不要做）。例如，通过向前的替换命令提供地址“Sebastopol”，我们可以限制只对包含“Sebastopol”的行才将“CA”替换为“California”

```
/Sebastopol/s/CA/California/g
```

由“Sebastopol,CA”组成的输入行将匹配这个地址，并且应用替换命令将它替换为“Sebastopol,California”。由“San Francisco, CA”组成的行不会被匹配，而且不会应用替换命令。

sed 命令可以指定零个、一个或两个地址。每个地址都是一个描述模式、行号或者行寻址符号的正则表达式。

- 如果没有指定地址，那么命令将应用于每一行。
- 如果只有一个地址，那么命令应用于与这个地址匹配的任意行。
- 如果指定了由逗号分隔的两个地址，那么命令应用于匹配第一个地址的第一行和它后面的行，直到匹配第二个地址的行（包括此行）。
- 如果地址后面跟有感叹号（！），那么命令就应该用于不匹配该地址的所有的行。

运维技术书籍免费下载 https://www.linuxprobe.com

为了解释寻址是如何工作的，让我们先来看看使用删除命令 d 的示列。一个只由 d 命令组成并且没有地址的脚本不会产生输出，因为它删除了所有的行：

```
d
```

当行号作为一个地址提供时，命令只影响那一行。例如，下面的示列只删除第一行：

```
1d
```

行号指由 sed 维护的内部行数。该计数器不会因为多个输入文件而重置。因此，不管指定多少个输入文件，在输入流中也只有一行 1.

同样，输入流也只有一个最后的行。可以使用寻址符号$指定。下面的示列删除输入的最后一行：

```
$d
```

$符号不应该和正则表达式中使用的$相混淆，在这里表示行的结束。

当正则表达式作为地址提供时，这个命令只影响与这个模式匹配的行。正则表达式必须封闭在斜杠（/）中。下面的删除命令：

```
/^$/d
```

只删除空行。所有其他行都不会改变。

如果提供两个地址，那么就指定了命令执行的行范围。下例展示了如何删除由一对宏包围的所有行，在这种情况下，.TS 和.TE 标记了 tb1 输入：

```
/^\.TS/,/^\.TE/d
```

它删除了从第一种模式匹配的行开始，到由第二种模式匹配的行（包括此行在内）为止的所有行。这个范围之外的行不受影响。下面的命令删除了文件中从行 50 到最后一行的所有行：

```
50,$d
```

可以混合使用行地址和模式地址：

```
1,/^$/d
```

这个示列删除了从第一行直到第一个空行的所有的行，例如，可用它来删除保

运维技术书籍免费下载 https://www.linuxprobe.com

存在文件中的 Internet 邮件消息中的邮件头。

你可以把第一个地址看做是启用动作，并把第二个地址看做是禁用动作。Sed 没有把法先行决定第二个地址是否会匹配。一旦匹配了第一个地址，这个动作就将应用与这些行。于是命令应用于“所有”随后的行直到第二个地址被匹配。在上例中，如果文件不包含空行，那么将删除所有的行。

跟在地址后面的感叹号会反转匹配的意义。例如，下面的脚本将删除了在 tb1 输入块中的那些行以外的所有行：

```
/^\.TS/,/^\.TE/!d
```

实际上，这个脚本从源文件中提取 tb1 输入。

分组命令

sed 使用大括号（{ }）将一个地址嵌套在另一个地址中，或者在相同的地址上应用多个命令。如果想指定行的范围，然后在这个范围内指定另一个地址，则可以嵌套地址。例如，为了只删除 tb1 输入块中的空行，使用下面的命令：

```
/^\.TS/,/^\.TE/{
/^$/d
}
```

左大括号必须在行末，而且右大括号本身必须单独占一行。要确保在大括号之后没有空格。

可以使用大括号将编辑命令括起来对某个范围的行应用多个命令，如下所示：

```
/^\.TS/,/^\.TE/{
/^$/d
s/^\.ps 10/.ps 8/
s/^\.vs 12/.vs 10/
}
```

这个示列不仅删除了 tb1 输入块中的空行，而且它还使用替换命令 s，改变了几个 troff 请求。这些命令只应用于.TS/.TE 块中的行。

测试并保存输出

在前面关于模式空间的讨论中，可以看到 sed：

1. 生成输入行的备份。
2. 修改模式空间中的备份。
3. 将备份输出到标准输出。

运维技术书籍免费下载 https://www.linuxprobe.com

这些意味着 sed 有其内置的安全措施，所以会改变原始的文件。因此，下面的命令行：

```
$ sed -f sedscr testfile
```

不会在 testfile 中做改动。它将所有的行送往标准输出（一般是屏幕）——包括被修改的行和没有被修改的行。如果想要保存这些输出，就必须将它们收集到一个新文件中。

```
$ sed -f sedscr testfile > newfile
```

其中，重定向符号“>”将来自 sed 的输出直接送往文件 newfile 中。不要将来自命令的输出重定向到输入文件，否则会改写输入文件，甚至可能在 sed 处理这个文件之前发生，并破坏你的数据。

将输出重定向到文件的一个重要的理由是要检验结果。可以检查 newfile 的内容并于 testfile 进行比较。如果想要很系统地检查结果（应该这样），可以使用 diff 程序指出两个文件之间的区别。

```
$ diff testfile newfile
```

这个命令将在 testfile 特有的行钱买你显示“<”，并在 newfile 特有的行前面显示“>”。当检验了结果后，要生成原始文件的一个备份，然后使用 mv 命令来用新文件改写原始文件。在丢弃原始文件之前要确保编辑的脚本可以正确地工作。

由于要如此频繁地重复这些步骤，因此将它们放入一个 shell 脚本是非常有用的。但我们不能更深入地讨论 shell 脚本的工作情况，对这些脚本的理解和使用非常简单。编写 shell 脚本需要使用文本编辑器，向文件中输入一个或多个命令行，并保存文件然后使用 chmod 命令生成可执行的文件。文件的名字就是命令的名字，并且可以在系统提示符下输入它。如果不熟悉 shell 脚本，可以按照本书出现的 shell 脚本生成你自己的替换。

下面两个 shell 脚本对于测试 sed 脚本以及对文件中进行永久性改动非常有用。当需要在多个文件上运行同一个脚本时，它们尤其有用。

tested

shell 脚本 tested 自动将 sed 的输出保存在临时文件中。它期望在当前目录中找打脚本文件 sedscr，并将其中的指令应用与在命令行指定的输入文件。这些输出被放置在一个临时文件中。

```
for x
do
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
sed -f sedscr $x > tmp.$x
done
```

文件的名字必须在命令行上指定。结果，这个 shell 脚本将输出保存到以“tmp”为前缀的临时文件中。可以检查临时文件以决定编辑工作是否正确。如果结果是对的，则可以使用 mv 命令来用临时文件改写原始文件。也可以将 diff 命令插入到 shell 脚本（在 sed 命令之后添加 diff$x tmp.$x）。

如果你发现脚本没有产生所期望的结果，记住最简单的“修复（fix）”通常是完善编辑脚本然后在原始输入文件上再次运行它。不要编写一个新脚本来“取消（undo）”或者改进临时文件中生成的改动。

runsed

这个 shell 脚本 runsed 是为了对输入文件实现永久性的改变。换句话说，如果想让输入文件与输出文件相同时可以使用它。与 testsed 类似，它创建一个临时文件，但是它随后会执行下一步：用这个文件改写原始文件。

```
#! /bin/sh
 for x
do
   echo "editing $x: \c"
   if test "$x" = sedscr; then
    echo "not editing sedscript!"
  elif test -s $x; then
  sed -f sedscr $x > /tmp/$x$$
  if test -s /tmp/$x$$
  then
    if cmp -s $x /tmp/$x$$
    then
        echo "file not changed: \c"
    else
        mv $x $x.bak # save original, just in case
        cp /tmp/$x$$ $x
    fi
    echo "done"
   else
      echo "Sed produced an empty file\c"
      echo " - check your sedscript."
  fi
  rm -f /tmp/$x$$
  else
    echo "original file is empty."
  fi
done
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
echo "all done"
```

要使用 runsed，在想要进行编辑操作的目录下，创建一个名为 sedser 的 sed 脚本，在命令行上写上要编辑的文件名。Shell 元字符可用于指定一组文件。

```
$ runsed ch0?
```

runsed 简单地对命名文件使用 sed -f sedscr，一次一个，并且将输出重定向到一个临时文件中。然后 runsed 测试这个临时文件，以确认在覆盖原始文件之前产生了输出。

这个 shell 脚本的内容（第 9 行）本质上与 testsed 相同。附加的行用于测试未成功的运行，例如，当没有产生输出时。它比较两个文件看看是否真的进行了改动，或者看看在改写原始文件之前是否产生了空的输出文件。

然而，runsed 无法避免产生未完成的编辑脚本。在用 runsed 生成永久性的改变之前，应该首先使用 testsed 检验这些改变。

sed 脚本的 4 中类型

在本节中，我们将会看到 4 种类型的脚本，每类脚本都说明了一种典型的 sed 应用。

对同一文件的多重编辑

sed 脚本的第一种类型示范了在一个文件中进行一系列编程工作。我们使用的第一个示列是将由字处理程序创建的文件转换为用于 troff 的编码文件。

本书的以为作者曾经为计算机公司编写了一个项目，这里指 BigOne Computer。这个文档必须包括“Horsefeathers Software”产品数名。这个公司许诺产品说明是在线的并且他们将发送它。可惜的是，当文件到达时，它包含行式打印机的格式化输出，这事他们可以提供的惟一方式。那个文件的一部分（在 horsefeathers 文件中保存的用于测试的部分）如下所示：

```
HORSEFEATHERS SOFTWARE PRODUCT BULLETIN
DESCRIPTION
+ ____________
BigOne Computer offers three software packages from the suite
of Horsefeathers software products -- Horsefeathers Business
BASIC, BASIC Librarian, and LIDO. These software products can
fill your requirements for powerful, sophisticated,
general-purpose business software providing you with a base for
software customization or development.
Horsefeathers BASIC is BASIC optimized for use on the BigOne
machine with UNIX or MS-DOS operating systems. BASIC Librarian
is a full screen program editor, which also provides the ability
```


运维技术书籍免费下载 https://www.linuxprobe.com

注意，文本已经通过在单词之间添加空格调整过了。还添加了空格来产生左边界。

当开始用 sed 处理问题时，如果对想要做的事情设计一个备忘录，我们将会把它做的很好。当我们开始编码时，我们编写一个完成单一功能的单个命令的脚本。并测试它的作用，然后添加另一个命令，重复这个循环过程知道完成所有显然要做的事情（因为列表不总是很完整的，所以称“所有显然的”，而且这个实现与测试的循环常会向列表添加其他的项目）。

这样的工作似乎是非常乏味的过程，而且实际上有许多脚本在一次编写完成后才开始测试而且效果很好，然而，向初学者高难度推荐逐步编写的技术，是因为将每个命令隔离开来可以很容易地看出哪些功能实现了，哪些还没有。如果同时尝试几个命令，则在问题出现时需要按和创建命令相反的过程来结束；也就是说，一个一个地删除命令直到找到问题为止。

下面是 Horsefeathers Software 产品说明显然需要做的一个编辑工作的列表：

1. 用段落宏（.LP）取代所有的空行。
2. 删除每行的所有的前导空格。
3. 删除打印机下划线的行，即以“+”开始的行。
4. 删除添加在两个单词之间的多个空格。

第一个编辑操作要求匹配空行。但是，在查看输入文件内容时，空行是否有前导空格并不明显。当清楚空格后，它们没有前导空格，所以空行可以采用模式“^$”匹配（如果行上有空格，那么模式写成“^[]*$*”）。因此，第一个编辑操作可以用以下方法非常简单地实现：

```
s/^$/.LP/
```

它用“.LP”取代每个空行。注意在替换命令的替代部分不必转义字母句点。我们可以在 sedscr 文件中放置这个命令，并且使用下面的命令进行测试：

```
$ sed -f sedscr horsefeathers
HORSEFEATHERS SOFTWARE PRODUCT BULLETIN
.LP
DESCRIPTION
+ ___________
.LP
BigOne Computer offers three software packages from the suite
of Horsefeathers software products -- Horsefeathers Business
Four Types of sed Scripts 65
18 May
BASIC, BASIC Librarian, and LIDO. These software products can
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
    fill your requirements for powerful, sophisticated,
    general-purpose business software providing you with a base for
    software customization or development.
    .LP
    Horsefeathers BASIC is BASIC optimized for use on the BigOne
    machine with UNIX or MS-DOS operating systems. BASIC Librarian
    is a full screen program editor, which also provides the
ability
```

哪些行被改变过是很明显的（将文件的一部分单独隔离出来进行测试常常很有用。如果这一部分足够小，以使用屏幕范围，同时又足以覆盖你想要进行改变的不同的示列，那么它可以工作得最好。当这个测试文件的所有编辑工作完成之后，你就可以将这部分移植回源文件，并开始第二次测试）。

我们要做的第二个编辑操作是删除以“+”开始并且包含行式打印机下划线的行。我们可以使用删除命令 d 只删除这一行。在编写匹配这一行的模式中，我们可以有许多选择。下面的每一种形式都可以匹配这一行：

```
/^+/
/^+□/
/^+□□*/
/^+□□*_*/
```

可以看出，每个正则表达式一次匹配的字符越来越多。只有通过测试才能决定需要多复杂的表达式来匹配特定的行而不是其他的行。在正则表达式中定义的模式越长，就越容易使它不会产生不想要的匹配，对于这个脚本，我们选择第三种表达式：

```
/^+□□*/d
```

这个命令删除以加号开始并且后面跟有至少一个空格的任意行。模式中指定两个空格，但是第二个可能个个由“*”修饰，意味着第二个空格可以有也可以没有。

将这个命令添加到 sed 脚本并进行测试，但是因为它只影响一行，所以我们省略了结果的显示并且继续。下一个编辑操作需要删除在行开始位置填充的空格。匹配序列的模式非常类似于前一个命令的地址。

```
s/^□□*//
```

这个命令删除在行的开头发现的任意空格序列。替换命令的替换部分为空，这意味着删除了被匹配的字符串。

我们将这个命令添加到脚本中并测试它。

运维技术书籍免费下载 https://www.linuxprobe.com

```
$ sed -f sedscr horsefeathers
HORSEFEATHERS SOFTWARE PRODUCT BULLETIN
.LP
DESCRIPTION
.LP
BigOne Computer offers three software packages from the suite
of Horsefeathers software products -- Horsefeathers Business
BASIC, BASIC Librarian, and LIDO. These software products can
fill your requirements for powerful, sophisticated,
general-purpose business software providing you with a base for
software customization or development.
.LP
Horsefeathers BASIC is BASIC optimized for use on the BigOne
machine with UNIX or MS-DOS operating systems. BASIC Librarian
is a full screen program editor, which also provides the ability
```

下一个编辑操作试图处理为了对齐而添加的额外空格。我们可以编写一个替换命令匹配任意连续空格字符串并用一个空格取代它。

```
s/□□*/□/g
```

在命令的结尾添加全局标志以便取代所有的出现（不只是第一个），要注意，和前面的正则表达式一样，我们不能指定有多少空格 ——可能只有一个或多个。有可能是 2 个，3 个或 4 个连续的空格。不管有多少，都将它们缩减为一个（注 3）。

注 3： 这个命令还匹配单个空格。但是因为替换内容也是一个空格，因此有一种情况实际上“没有工作”。

我们来测试这个新的脚本：

```
$ sed -f sedscr horsefeathers
HORSEFEATHERS SOFTWARE PRODUCT BULLETIN
.LP
 DESCRIPTION
.LP
BigOne Computer offers three software packages from the suite
of Horsefeathers software products -- Horsefeathers Business
BASIC, BASIC Librarian, and LIDO. These software products can
fill your requirements for powerful, sophisticated,
general-purpose business software providing you with a base for
software customization or development.
.LP
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
Horsefeathers BASIC is BASIC optimized for use on the BigOne
machine with UNIX or MS-DOS operating systems. BASIC Librarian
is a full screen program editor, which also provides the ability
```

该脚本就像所建议的那样工作，将两个或多个空格缩减为一个。但是，进一步查看会发现该脚本删除了跟在句点后的两个空格序列，而在这里本应该有两个空格。

可以完善替换命令以便它不会替换句点后的空格。问题是句点后也可能有 3 个空格，则需要将它们缩减为 2 个空格。最好的方式似乎是编写独立的命令来处理这种句点后面跟有空格的特殊情况。

```
s/\.□□*/.□□/g
```

这个命令用后面跟有两个空格的句点，替换了后面跟有任意多个空格的句点。应该注意，前面的这个命令将多个空格缩减为一个，所以在句点后面只能找到一个空格。（注 4）尽管如此，这种模式不管句点后面有多少个空格都会工作，只要至少有一个空格就可以了（例如，这种形式不会影响出现在文档中的文件名 text.ext）。这个命令被放置在脚本的结尾处并进行测试：

```
$ sed -f sedscr horsefeathers
HORSEFEATHERS SOFTWARE PRODUCT BULLETIN
.LP
DESCRIPTION
.LP
BigOne Computer offers three software packages from the suite
of Horsefeathers software products -- Horsefeathers Business
BASIC, BASIC Librarian, and LIDO. These software products can
fill your requirements for powerful, sophisticated,
general-purpose business software providing you with a base for
software customization or development.
.LP
Horsefeathers BASIC is BASIC optimized for use on the BigOne
machine with UNIX or MS-DOS operating systems. BASIC Librarian
is a full screen program editor, which also provides the ability
```

注 4： 这个命令因此可以简化为：

```
s/\.□/.□□/g
```

它能正常工作。下面是完整的脚本：

```
s/^$/.LP/
/^+ */d
s/^ *//
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
s/ */ /g
s/\. */. /g
```

正如我们前面所说的那样，下一个阶段将使用 testsed 在完整的文件（hf.product.bulletin）上测试脚本并彻底地检查结果。当对这个结果满意时，可以使用 runsed 生成永久性的改变：

```
$ runsed hf.product.bulletin
done
```

通过执行 runsed，我们改写了原始文件。

在离开这个脚本之前，需要指出的是，尽管这个脚本是为处理特殊的文件而编写的，但是脚本中的每个命令都可能会再次用到，尽管可能不会再使用整个脚本。换句话说，你可以很好的编写其他的脚本，来删除空行或检查跟在句点后的两个空格。要认识到在其他的情况下有多少命令可以重用，这样可以缩减开发和测试信脚本的时间。就像歌手学唱一首歌并把它添加到他或她的歌单中一样。

改变一组文件

sed 的最常见的用法是对一组文件进行一系列搜索和替换编辑操作。通常这些脚本并不少见也没有趣，它们只是一个将单词或短语变成另一种形式的替换命令的列表。当然，这样的脚本不需要有趣，只要它们有用并能节省手动工作就行。

本节中我们看到的例子是转换脚本，被用来修改 UNIX 文档集中的各种“机器专用的”术语。一个人使用文档集来生成一个需要被改变的事情的列表。另一个人根据列表来创建下面的替换清单。

```
s/ON switch/START switch/g
s/ON button/START switch/g
s/STANDBY switch/STOP switch/g
s/STANDBY button/STOP switch/g
s/STANDBY/STOP/g
s/[cC]abinet [Ll]ight/control panel light/g
s/core system diskettes/core system tape/g
s/TERM=542[05] /TERM=PT200 /g
s/Teletype 542[05]/BigOne PT200/g
s/542[05] terminal/PT200 terminal/g
s/Documentation Road Map/Documentation Directory/g
s/Owner\/Operator Guide/Installation and Operation Guide/g
s/AT&T 3B20 [cC]omputer/BigOne XL Computer/g
s/AT&T 3B2 [cC]omputer/BigOne XL Computer/g
s/3B2 [cC]omputer/BigOne XL Computer/g
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
s/3B2/BigOne XL Computer/g
```

这个脚本非常直观。好处并不在脚本本身，而是在于 sed 能将这个脚本应用于组成文档集的数百个文件。一旦这个脚本通过了测试，就可以使用 runsed 来一次处理这里所有的文件。

这样的脚本可以节省大量的时间，但是它还是有可能产生导致浪费大量时间的错误。有时人们编写脚本并在数百个文件中的一到两个上测试它，并得出脚本能很好工作的结论。测试每个文件是不切实际的，因此选择有代表性且包含异常的测试文件是非常重要的。记住，文本之间有很大的不同，所以不能认为一种特殊的情况为真，所有的情况就都为真。

使用 grep 来检查大量输入是非常有帮助。例如，如果要确定“core system diskettes”如何出现在文档中，则可以在各处查找（grep）它并注意清单。为了更加彻底，还可以查找“core”、“core system”、“system diskettes”和“diskettes”来寻找在多行上的出现（也可以使用第六章中的 phrase 脚本，寻找连续行上的多个单词）。检查输入是了解脚本必须要做什么的最好方式。

在某些方面，编写脚本就像设计一个假设，给定事实的某种集合。通过增加测试数据来试着验证假设的合法性。如果打算在多个文件上运行该脚本，使用 testsed 首先在较小的示列上测试它，然后在许多文件上运行这个脚本。接着比较临时文件和原始文件看看假设是否正确。脚本也许会有错误，需要对它进行修改。花费在测试上的时间越多（实际上它是非常有趣的工作），那么在解决由拙劣的脚本导致的问题上花费时间就越少。

提取文件的内容

sed 应用程序的一种典型的用法是从文件中提取相关的材料。sed 这一功能类似与 grep，而且它具有在输出之前修改输入的又一优点。这种类型的脚本是 shell 脚本很好的候选。

下面有两个示列：从宏包中提取宏定义并且显示文档的提纲。

提取宏定义

Troff 宏在宏包中进行定义，它通常是存放在某个目录（例如 /usr/lib/macros）下的一个文件中。Troff 宏定义总是以字符串“.de”开始，后面跟有可选的空格或者是由一个或两个字母构成的宏的名字。宏定义在以两个句点（..）开始的行处结束。本节显示的脚本是从一个宏包中提取特殊的宏定义（它可以节省你用编辑器定位并打开文件和搜索想要检查的行的时间）。

设计这个脚本的第一步是编写提取指定宏的部分，本例指-mm 包中的（注 5）BL（Bulleted List）宏。

运维技术书籍免费下载 https://www.linuxprobe.com

```
$ sed -n "/^\.deBL/,/^\.\.$/p" /usr/lib/macros/mmt
.deBL
.if\\n(.$<1 .)L \\n(Pin 0 1n 0 \\*(BU
.if\\n(.$=1 .LB 0\\$1 0 1 0 \\*(BU
.if\\n(.$>1 \{.ie !\w^G\\$1^G .)L \\n(Pin 0 1n 0 \\*(BU 0 1
.el.LB 0\\$1 0 1 0 \\*(BU 0 1 \}
..
```

可以用-n 选项来调用 sed 从而阻止它打印整个文件。使用这个选项，sed 只打印通过打印命令显示指定的要打印的行。sed 脚本包含两个地址：第一个匹配宏定义的开始“.deBL”，第二个匹配它的终端“..”（自成一行）。注意，这两个模式中出现的句点用反斜杠转义。

这两个地址指定了打印命令 p 的行范围。这就是这种搜索脚本与 grep（不能匹配行的范围）的功能区别。

通过将该命令放置在 shell 脚本中可以使它更通用。创建 shell 脚本的一个明显的优点是它保存键入的内容。另一个优点是 shell 脚本可以被设计为更一般的用法。例如，可以允许用户从命令行提供信息。在这种情况下，在 sed 脚本中可不用硬编码指定宏的名字，而是使用命令行参数来提供它。在 shell 脚本中可以使用位置符号来指定命令行上的每个参数：第一个参数是$1,第二个参数是$2，以此类推。下面是 getmac 脚本：

```
#! /bin/sh
# getmac -- print mm macro definition for $1
sed -n "/^\.de$1/,/^\.\.$/p" /usr/lib/macros/mmt
```

注 5： 我们碰巧知道-mm 宏在“-.de”命令之后没有空格。

这个 shell 脚本的第一行强制使用 Bourne shell 进行脚本的解释，其中使用了在现代 UNIX 系统上都可以使用的“#!”可执行解释程序机制。第二行是描述脚本的名字和目的的注释。第三行上的 sed 命令，除了“BL”被“$1”取代以外（“$1”是表示第一个命令行参数的变量），其余的与上一个示列相同。注意，括住 sed 脚本的双引号是必须的。如果使用单引号，则 shell 不对“$1”进行解释。

脚本 getmac 可以按下面所示的方式执行：

```
$ getmac BL
```

这里的“BL”是第一个命令行参数。它和前面的示列产生同样的输出。

这个脚本可以处理任意个宏包。下面的 getmac 版本允许用户将宏包的名字指定

运维技术书籍免费下载 https://www.linuxprobe.com

为第二个命令行参数。

```
#! /bin/sh
# getmac - read macro definition for $1 from package $2
file=/usr/lib/macros/mmt
mac="$1"
case $2 in
-ms) file="/work/macros/current/tmac.s";;
-mm) file="/usr/lib/macros/mmt";;
-man) file="/usr/lib/macros/an";;
esac
sed -n "/^\.de *$mac/,/^\.\.$/p" $file
```

这里的新内容是一个 case 语句，它用于测试$2 的值并随后给变量 file 赋一个值。注意，首先给 file 赋了一个默认的值、所以如果用户没有指明宏包，那么就会搜索-mm 宏包。而且，为了清楚和易读、$1 的值被赋给变量 mac。

在创建这个脚本的过程中，我们在宏定义第一行中发现了不同宏包之间的区别。-ms 宏在“.de”和宏的名字之间有一个空格，而-mm 和-man 没有。幸运的是，可以通过修改模式来适应这两种情况。

```
/^\.de *$mac/
```

在“.de”的后面指定了一个空格，后面跟一个星号，这意味着这个空格是可选的。

这个脚本在标准输出上打印了结果，但是它可以被很容易地重定向到一个文件，在这里它可以成为重新定义宏的基础。

生成提纲

我们的下一个示列不仅提取了信息，而且对它进行修改从而使它更易阅读。我们创建了名为 do.outline 的 shell 脚本，这个脚本使用 sed 生成了文档的大纲视图。它处理包含被编码的标题部分的行，例如：

```
.Ah "Shell Programming"
```

我们使用的这个宏包中有一个章标题宏，其名称为“Se”，分层标题命名为“Ah”，“Bh”和“Ch”。在-mm 宏包中，这些宏可以是“H”、“H1”、“H2”、“H3”等等。可以使脚本适用于任何标识文档结构的宏或标志。Do.outline 脚本的目的是通过打印缩进格式的标题从而使结构更加突出。

do.outline 的结果显示如下：

```
$ do.outline ch13/sect1
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
CHAPTER 13 Let the Computer Do the Dirty Work
   A. Shell Programming
      B. Stored Commands
      B. Passing Arguments to Shell Scripts
      B. Conditional Execution
      B. Discarding Used Arguments
      B. Repetitive Execution
      B. Setting Default Values
      B. What We’ve Accomplished
```

它将结果打印到标准输出上（当然，在文件内没有发生任何改变）。

让我们来看如何将这个脚本放置在一起。这个脚本需要匹配以下面的宏开始的行：

- 章标题（.Se）
- 节标题（.Ah）
- 子节标题（.Bh）

我们需要在那些行上进行替换，用文本标记（例如，A、B）取代宏并添加适当数量的空格（使用制表符）来缩进每个标题。（记住，“•”表示一个制表符。）

下面是基本的脚本：

```
sed -n ’
s/^\.Se /CHAPTER /p
s/^\.Ah /•A. /p
s/^\.Bh /••B. /p’  $*
```

do.outline 对在命令行（“$*”）上指定的所有的文件起作用。-n 选项抑制了程序的默认输出。这个 sed 脚本包含 3 个替换命令，用字母替换代码并缩进每一行。每个替换命令都可以用 p 标志来修饰，这个标志表示该行应该打印。

当测试这个脚本时，产生如下结果：

```
CHAPTER "13" "Let the Computer Do the Dirty Work"
  A. "Shell Programming"
  B. "Stored Commands"
  B. "Passing Arguments to Shell Scripts"
```

宏参数中的引号被同时传递进来。我们可以编写一个替换命令来删除这些引号。

运维技术书籍免费下载 https://www.linuxprobe.com

```
s/ "//g
```

指定全局标志 g 来捕获一行上的所有出现是必要的。然而，关键是将这个命令放在脚本中的什么位置。如果我们将它放在脚本的结尾，那么它将在输出行之后删除引号。我们必须将它放在脚本的起始位置并针对所有的行进行同样的修改，不管它们之后是否在脚本中输出。

```
sed -n '
s/ "//g
s/^\.Se /CHAPTER /p
s/^\.Ah /•A. /p
s/^\.Bh /••B. /p'  $*
```

这个脚本当前产生的结果在前面已显示过。

可以修改这个脚本以搜索任意种类的编码格式。例如，下面是 LATEX 文件的粗略版本。

```
sed -n '
s/[{}]//g
s/\\section/•A. /p
s/\\subsection/••B. /p'  $*
```

编辑工作转移

让我们看看将 sed 作为真正的流编辑器的一个应用，在管道中进行编辑操作，这些编辑操作永远不会被写回到文件中。

在一些类似打字机的设备上（包括 CRT），一个长破折号被作为一对连字符（-）来键入。在排版过程中，它被作为一个长破折号（—）打印。Troff 为长破折号提供了一个特殊的字符名、但是键入“\(em”很不方便

下面的命令将两个连续的破折号转换为一个长破折号。

```
s/--/\\(em/g
```

我们在替换字符串中用两个反斜杠来表示\(em ，因为反斜杠在 sed 中有特殊的含义。

也许在很多情况下我们不想应用这个替换命令。如果有人用连字符绘制水平线将会怎样？我们可以重定义这个命令来排除含有 3 个或多个连续的连字符的行。为了完成这个任务，我们使用!地址修饰符：

```
/---/!s/--/\\(em/g
```


运维技术书籍免费下载 https://www.linuxprobe.com

可能要花一点时间来理解这个语法。所不同的是使用模式地址限制受替换命令影响的行，并且使用！反转模式匹配的意义。简单地说，“如果找到含有 3 个连续的连字符的行，不要应用这个编辑操作。”在所有的其他行上，应用替换命令。

可以在脚本中使用这个命令自动插入长破折号。要实现这一功能，可以将 sed 用做 troff 文件的预处理程序。文件将用 sed 处理，然后输送到 troff。

```
sed ’ /---/!s/--/\\(em/g’  file | troff
```

换句话说，sed 改变了这个输入文件并且将输出直接传递到 troff，而不用创建一个中间文件。编辑在运行中进行，而且不会影响输入文件。你也许想知道为什么不再原始文件中直接进行永久性修改？一个很简单的原因是这样做没有必要——输入一直和用户兼容的内容保持一致，但是 troff 能产生看上去版面质量最好的输出。而且，因为它内嵌在庞大的 shell 脚本中，所以连字符到长破折号的转换对用户来说是不可见、并且在格式化进程中没有额外的步骤。

我们使用名为 format 的 shell 脚本，使用 sed 来达到这个目的。下面是这个 shell 脚本的内容：

```
#! /bin/sh
eqn= pic= col=
files= options= roff="ditroff -Tps"
sed="| sed ’ /---/!s/--/\\(em/g’ "
while [ $# -gt 0 ]
do
  case $1 in
   -E) eqn="| eqn";;
 -P) pic="| pic";;
 -N) roff="nroff" col="| col" sed= ;;
 -*) options="$options $1";;
 *) if [ -f $1 ]
 then files="$files $1"
 else echo "format: $1: file not found"; exit 1
 fi;;
  esac
shift
done
 eval "cat $files $sed | tbl $eqn $pic | $roff $options $col |
lp"
```

这个脚本对一些变量（以美元符号为前缀）进行赋值和求值，这些变量构成了这个格式化和打印文档的命令行（注意我们为 nroff 设置了-N 选项，所以它可以将 sed 变量设置为空字符串，因为如果使用 troff，我们指向生成这种改

运维技术书籍免费下载 https://www.linuxprobe.com

变，即使 nroff 可以理解这个特殊的字符\(em,进行这种改变也不会对输出有实际的影响）。

当对一个文档排版时，将连字符换成长破折号不是惟一要做的“美化的”工作。例如，多数键盘不允许键入左引号和右引号（“和”与“和”相对）。在 troff 中，可以通过键入两个连续的重音符或“反引号（‘‘）”来表示左引号。键入两个连续的单引号(’’)来标识右引号。我们可以使用 sed 将每个双引号字符换成一个单个的左引号或右引号（取决于上下文），也就是，当排版时，产生合适的“双引号”。

这事一个相当困难的编辑工作，因为有许多涉及到标点符号、空格和制表符的各种情况。脚本可能看上去如下所示：

```
s/^"/ ‘ ‘/
s/"$/’ ’ /
s/"? /’ ’ ? /g
s/"?$/’ ’ ?/g
s/ "/  ‘ ‘/g
s/" /’ ’  /g
s/•"/• ‘ ‘/g
s/"•/’ ’ •/g
s/")/’ ’ )/g
s/"]/’ ’ ]/g
s/("/( ‘ ‘/g
s/\["/\[ ‘ ‘/g
s/";/’ ’ ;/g
s/":/’ ’ :/g
s/,"/,’ ’ /g
s/",/’ ’ ,/g
s/\."/.\\\&’ ’ /g
s/"\./’ ’ .\\\&/g
s/\\(em\\^"/\\(em ‘ ‘/g
s/"\\(em/’ ’ \\(em/g
s/\\(em"/\\(em ‘ ‘/g
s/@DQ@/"/g
```

第一个替换命令寻找一行开始处的引号并把它换成一个左引号。第二个命令寻找行结尾处的引号并把它换成一个右引号。剩下的命令寻找不同上下文中的引号，例如，标点符号、空格、制表符或长破折号前后的引号。最后一个命令允许我们在需要的时候为 troff 提供真正的双引号（“）。我们将这些命令与将连字符转换成破折号的命令放置在“cleanup”脚本中，并且在使用 troff 来格式化和打印文档的管道中调用它。

运维技术书籍免费下载 https://www.linuxprobe.com

开始 PromiSed Land

目前，你已经看过了 4 种不同类型的 sed 脚本，以及它们如何被嵌入到 shell 脚本中以创建易于使用的应用。当使用 sed 工作时，你会开发越来越多的用于创建和测试 sed 脚本的方法。你将逐渐依赖这些方法，并且对你的脚本在做什么和为什么这么做充满信息。

下面是几个提示：

1. 了解你的输入！在使用 grep 设计脚本之前，自信检查输入文件。
2. 购买之前要采样。从测试文件中出现的小示列开始。在示列上运行你的脚本并且确信脚本能正常工作，记住，确保脚本在你不想让它工作的地方不能工作同样重要。然后增加示列的规模，试着增加输入的复杂性。
3. 做之前要仔细考虑。仔细地测试你添加到脚本中的每个命令。比较输出和输入文件来看看发生了什么变化。亲自证明你的脚本是完整的。确定在输入文件正确的前提下，你的脚本确定可以正确地工作，而不仅仅是你认为可以。
4. 要使用！尝试完成你用 sed 脚本可以完成的事，但不必 100%完成这个工作。如果遇到困难的情况，检查并查看它们发生的频繁程度。有时手动来完成剩下的几个编辑工作比较好。

随着你的经验的增长，在以上列表中添加你自己的“脚本化提示”。当使用 awk 工作时，你会发现这些提示同样很有用。

运维技术书籍免费下载 https://www.linuxprobe.com

第五章

基本 sed 命令

本章内容

- sed 命令语法
- 注释
- 替换
- 删除
- 追加、插入和更换
- 列表
- 转换
- 打印
- 打印行号
- 下一步
- 读和写文件
- 退出

Sed 命令集合由 25 个命令组成。本章我们介绍 4 个新的编辑命令：d（删除），a（追加），i（插入）和 c（更改）。我们还要看一下改变脚本中流程控制（例如，决定下一步执行哪个命令）的方式。

sed 命令的语法

在看单个命令以前，需要回顾一下关于所有 sed 命令的两点语法。在上一个章中，我们介绍了其大部分内容。

行地址对于任何命令都是可选的。它可以使一个模式，被描述为由斜杠、行号或行寻址符号括住的正则表达式。大多数 sed 命令能接受由逗号分隔的两个地址，这两个地址，这两个地址用来标识行的范围。这些命令的语法格式为：

```
[address]command
```

有一些命令只接受单个行地址。它们不能应用于某个范围的行。它们的语法格式为：

```
[line-address]command
```

记住命令还可以用大括号进行分组以使其作用于同一个地址：

```
address {

command1

command2

command3
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
}
```

第一个命令可以喝左大括号放置在同一行，但是右大括号必须自己单独处于一行。每个命令都可以有自己的地址并允许有多层分组。而且，就像命令在大括号内的缩进方式一样，允许在行的开始处插入空格和制表符。

当 sed 不理解一个命令时，它打印出消息“Command garbled(命令不清)”。在命令后添加空格会产生一个小的语法错误，这个不允许的，命令的结束必须在行的结尾处。

这个约束的依据是来自一个文档中“无记载”特征提供的：如果命令之间用一个分号分隔，那么可以将多个 sed 命令放在同一行（注 1）。下面的示例在语句构成上是正确的：

```
n;d
```

然而，在 n 命令后面放置一个空格会导致语法错误，而在 d 命令前面放置一个空格是可以的。

不提倡在同一行放置多个命令，因为即使将这些命令写在各自的行上，sed 脚本也是很难阅读的（注意，更改、插入和追加命令必须在多行上指定，不能在同一行上指定）。

注释

可以使用注释描述脚本的作用，来为脚本编写文档。从本章开始，完整的脚本示例都以一个注释行开始。注释行可以作为脚本的第一行出现。在 sed 的 System V 版本中，注释只允许出现在第一行。在一些版本中，包括 SunOS 4.1x 环境下运行的 sed 以及 GNU sed，可以在脚本的任何地方放置注释，甚至是跟在命令行的后面。本书的示例更多地遵循 System V sed 的限制，将注释限制在脚本的第一行。然而，使用注释作为脚本文档往往非常有效，如果你的 sed 版本允许则应该使用它。

注释 1： 令人惊讶的是，用分号分隔命令的用法不在 POSIX 标准中。

注释行的第一个字必须是“#”号。注释行的语法如下：

```
#[n]
```

下列显示了一个脚本的第一行：

```
# wstar.sed: convert WordStar files
```

如果有必要，可以用反斜杠来结束前面的行使得注释可以继续多行（注释 2）。为了前后一致，必须用#开始继续行，以便使行的目的显而易见。

运维技术书籍免费下载 https://www.linuxprobe.com

如果跟在#后面的下一个字符是 n，那么脚本不会自动产生输出。这和指令命令行选项-n 是等价的。跟在 n 后面的其余的内容被看做是注释。在 POSIX 标准中，采用这种方式的#n 必须是文件的前两个字符。

替换

我们已经讨论了替换命令的许多用法。下面是它的详细的用法：

[address]s/pattern/replacement/flags

这里修饰替换的标志 flags 是：

n 1 到 512 之间的一个数字，表示对文本模式中指定模式第 n 次出现的情况进行替换

g 对模式空间的所有出现的情况进行全局更改。而没有 g 是通常只有第一次出现的情况被取代。

p 打印模式空间的内容。

W file

将模式空间的内容写到文件 file 中。

注 2： 使用 GNU sed（版本 2.05）时不起作用。

替换命令应用于与 address 匹配的行。如果没有指定地址，那么就应用于 Pattern 匹配的所有行。如果正则表达式作为地址来提供，并且没有指定模式，那么替换命令匹配由地址匹配的内容。当替换命令是应用于同一个地址上的多个命令之一时，这可能会非常有用。可以参看本章后面的“检验参考页”一节中的示例。

和地址不同的是，地址需要一个作为定界符的斜杠（/），而正则表达式可以用任意字符来分隔，只有换行符除外。因此，如果模式包含斜杠，那么可以选择另一个字符作为定界符，例如感叹号。

s!/usr/mail!/usr2/mail!

注意，定界符出现了 3 次而且在 replacement 之后是必需的。不管使用哪种定界符，如果它出现在正则表达式中，或者在替换文本中，那么就用反斜杠来转义它。

从前，计算机用固定长度的记录来存储文本。一行在出现许多字符（一般为 80 个）之后结束，然后开始下一行。数据中没有显式的字符来标记一行的结束和下一行的开始，每一行都有相同的（固定的）数量字符。现在的系统比较灵活，它们使用特殊的字符（称为换行符（newline））标记行的结束。这样就允许行行的长度为任意（注 3）。

运维技术书籍免费下载 https://www.linuxprobe.com

因为在内部存储时换行符只是一个字符，所以正则表达式可以使用“\n”来匹配嵌入的换行符。正如将在下一个章所看到的那样，在模式空间中当另一行扩展到当前行的特殊情况下，就会出现这样情况（参见第二章“了解基本操作“中关于行寻址的讨论和第三章“了解正则表达式语法“中有关正则表达式语法的讨论）。

Replacement 是一个字符串，用来替换与正则表达式匹配的内容（参见第三章中的“匹配的范围“一节）。在 replacement 部分，只用下列字符有特殊含义：

&	用正则表达式匹配的内容进行替换。
\n	匹配第 n 个字串（n 是一个数字），这个字串以前在 pattern 中用“\(“和 “\)“指定。
\	当在替换部分包含“与“符号（&），反斜杠（\）和替换命令的定界符时可用\转义它们。另外，它用于转义换行符并创建多行 replacement 字符串。

注 3： 或多或少。许多 UNIX 程序对它们处理的行的长度都有内部限制。但大多数 GNU 程序没有这样的限制。

因此，除了正则表达式中的元字符以外，sed 的替换部分也有元字符。参见下一节“替换元字符“中使用它们的示例。

Flag 可以组合使用，只要有意义。例如，gp 表示对行进行全局替换并打印这一行。迄今为止，全局标志是最常用的。没有它，替换只能在行的第一次出现的位置执行。打印标志和写标志与打印命令和写命令（本章后面会进行讨论）的功能相同，但有一个重要的区别。这些操作是随替换的成功而发生的。换句话说，如果进行了替换，那么这个行被打印或写到文件中。因为默认的动作时处理所有的行，不管是否执行了任何动作，当取消默认的输出时（-n 选项）通常使用打印和写标志。另外，如果脚本包含匹配同一行的多个替换命令，那么那一行的多个备份就会被打印或写到文件中。

数字表示很少使用，在这种情况下，正则表达式在一行上重复匹配，而只需要对其中某个位置的匹配进行替换。例如，某输入行也许包含 tb1 输入，也许包含多个制表位。假设每行有 3 个制表符，并且要用“>“替换第二个制表位，则可以使用下面的替换命令完成该功能：

```
s/•/>/2
```

“•” 表示一个真正的制表符，而制表符在屏幕上是不可见的。如果输入是一行的文件，如下所示：

```
Column1•Column2•Column3•Column4
```


运维技术书籍免费下载 https://www.linuxprobe.com

对这个文件运行以上脚本所产生的输出如下：

```
Column1•Column2>Column3•Column4
```

注意，如果没有数字标识，则替换命令只替换第一个制表符（因此“1“可以被看作是默认的数字标志）。

替换元字符

替换元字符是反斜杠（\）、“与”符号（&）和\n。反斜杠一般用于转义其他的元字符，但是他在替换字符串中也用于包含换行符。

我们可以对前面的示例做一些改动，用换行符取代每行上的第二个制表符。

```
s/•/\
/2
```

注意，在反斜杠后面不允许有空格。这个脚本产生下面的结果：

```
Column1•Column2
Column3•Column4
```

另一个示例来自于将 troff 文件转换成 Ventura Publisher 的 ASCII 输入格式。它将下面的 troff 行：

```
.Ah "Major Heading"
```

转换成类似的 Ventura Publisher 行：

```
@A HEAD = Major Heading
```

这个问题中的难点是这一行需要前后都有空行。这是一个编写多行替换字符串的问题。

```
/^\.Ah/{
s/\.Ah */\
\
@A HEAD = /
s/"//g
s/$/\
/
}
```


运维技术书籍免费下载 https://www.linuxprobe.com

第一个替换命令用两个换行符和“@A HEAD =”取代“.Ah”，在行结尾处有必要用反斜杠转义换行符。第二个替换删除了引号。最后一个命令匹配模式空间中的行的结尾（不是嵌入的换行符），并在它后面添加一个换行符。

在下一个例子中，反斜杠用来转义“与”符号，它作为普通字符出现在替换部分。

```
s/ORA/O’Reilly \& Associates, Inc./g
```

很容易忘记作为普通字符出现在替换部分的“与”符号。如果在这个例子中没有对它进行转义，那么输出的结果为“O’Reilly ORA Associates, Inc.”。

作为元字符，“与”符号（&）表示模式匹配的范围，不是被匹配的行。可以使用“与”符号匹配一个单词并且用 troff 请求来包围它。下面的示例用点数请求包围一个单词：

```
s/UNIX/\\s-2&\\s0/g
```

因为反斜杠也是替换字符串中的元字符，所以需要用两个反斜杠来输出一个反斜杠。替换字符串中的“&”表示“UNIX”。如果输入行为：

```
on the UNIX Operating System.
```

那么替换命令将产生：

```
on the \s-2UNIX\s0 Operating System.
```

当正则表达式匹配单词的变化时，“与”符号特别有用。它允许指定一个可变的替换字符串，该字符串相当于匹配的内容与实际内容匹配的字符创。例如，假设要用圆括号括住文档中对已编号部分的任意交叉引用。换句话说，任意诸如“See Section 1.4”或“See Section 12.9”的引用都应该出现在圆括号中，如“（See Section 12.9）”。正则表达式可以匹配数字的不同组合，所以在替换字符串中可以使用“&”并括起所匹配的内容。

```
s/See Section [1-9][0-9]*\.[1-9][0-9]*/(&)/
```

“与”符号用于在替换字符创中引用整个匹配内容。

现在，我们来看一种元字符，它用于选择被匹配的字符串的任意独立部分，并且在替换字符创中回调它。在 sed 中转义的圆括号括住正则表达式的任意部分

运维技术书籍免费下载 https://www.linuxprobe.com

并且保存它以备回调。一行最多允许“保存”9 次。“\n”用于回调被保存的匹配部分，n 是从 1 到 9 的数字，用于饮用特殊“保存的”备用字符串。

例如,当节号出现在交叉引用中时要表示为用粗体，可以编写下面的替换:

```
s/\(See Section \)\([1-9][0-9]*\.[1-9][0-9]*\)/\1\\fB\2\\fP/
```

指定了两队转义的圆括号。第一对捕获“See Section ”（因为它是固定的字符串，它可以直接在替换字符串中简单的被重新键入）。第二对捕获节号。在替换字符串中用”\1“回调第一个被保存的子串，用”\2“回调第二个被保存的子串，\2 用粗体字请求包围。

我们可以使用类似的技术匹配行的部分内容并交换它们。例如，假如在一行上有用冒号分隔的两个部分。我们可以匹配每个部分，把它们放置在转义的圆括号内并在替换过程交换它们。

```
$ cat test1
first:second
one:two
$ sed  ’ s/\(.*\):\(.*\)/\2:\1/’  test1
second:first
two:one
```

重要的是可以按任意顺序回调保存的子串，并且可以多次调用，正如在下一个示例中将要看到的那样。

校正索引条目

稍后，在本书的 awk 部分中，我们将给出一个格式化索引的程序，例如本书的索引。创建索引的第一个步是在文档文件中放置索引代码。我们使用命名为.XX 的索引宏，它采用单个参数，即索引条目。样本索引条目如下:

```
.XX "sed, substitution command"
```

每个索引条目占用一行。当运行一个索引时，会得到一个带有页码的索引条目的集合，该集合接着被排序并放入到一个列表中。编辑器常常会发现列表中需要进行改正的错误和不一致之处。简而言之，必须查找到索引太欧姆驻留的文件，然后进行校正，这是一件痛苦的事情，特别是当有大量的条目需要被修正的时候。

sed 在对一组文件进行这些编辑工作时可以提供很大的帮助。可以简单的在 sed 脚本中创建一个编辑列表，然后在所有的文件中运行该脚本。关键是替换命令

运维技术书籍免费下载 https://www.linuxprobe.com

需要将地址限制在以“.XX“开始的行。脚本本身不能改动正文。

假如我们要将上面的索引条目换成“sed,substitute command“。可以使用下面的命令来完成：

```
/^\.XX /s/sed, substitution command/sed, substitute command/
```

以上地址匹配所有以“.XX”开始的行，并且只有在那些行上才可以进行替换。你也许会想，为什么不指定一个较短的正则表达式呢？例如：

```
/^\.XX /s/substitution/substitute/
```

答案很简单，其他条目中“substitution”的使用可能是正确的，我们不想对这些条目改变。

我们可以进一步创建一个 shell 脚本，用于建立一个待编辑的索引条目列表，为下面一系列的 sed 替换命令做好准确。

```
#! /bin/sh
# index.edit -- compile list of index entries for editing.
grep "^\.XX" $* | sort -u |
sed '
s/^\.XX \(.*\)$/\/^\\.XX \/s\/\1\/\1\//'
```

sell 脚本 index.edit 使用 grep 从命令行上指定的任意数量的文件中，提取包含索引条目的所有行。它将列表递给 sort,sort 使用-u 选项来排序和删除重复的条目。然后这个列表被送到 sed，其中，这一行的 sed 脚本则构建了一个替换命令。

我们现在来进一步查看以上的 sed 脚本。下面就是其中的正则表达式：

```
^\.XX \(.*\)$
```

它匹配整个行，保存索引条目以备回调。下面是替换字符串：

```
\/^\\.XX \/s\/\1\/\1\/
```

它产生以地址开头的替换命令：地址开始为斜杠，然后是两个反斜杠——输出一个反斜杠以保护跟在后面的“.XX”中的句点，然后出现一个空格，接着是另一个斜杠以结束地址。接下来我们输出后面跟有斜杠的“s”，然后回调被保存的部分用来作为正则表达式。这后面跟着一个斜杠并且我们再次调用保存的子串并将它作为替换字符串。最后用一个斜杠结束这个命令。

运维技术书籍免费下载 https://www.linuxprobe.com

当 index.edit 脚本在文件上运行时，创建类似下面的一个清单：

```
$ index.edit ch05
/^\.XX /s/"append command(a)"/"append command(a)"/
/^\.XX /s/"change command"/"change command"/
/^\.XX /s/"change command(c)"/"change command(c)"/
/^\.XX /s/"commands:sed, summary of"/"commands:sed, summary of"/
/^\.XX /s/"delete command(d)"/"delete command(d)"/
/^\.XX /s/"insert command(i)"/"insert command(i)"/
/^\.XX /s/"line numbers:printing"/"line numbers:printing"/
/^\.XX /s/"list command(l)"/"list command(l)"/
```

这个输出可以被捕获到一个文件中。然后可以删除不需要改变的条目，而且可以通过编辑替换字符串来完成修改。这样，可以把这个文件作为 sed 脚本来纠正所有文档中的索引条目。

当处理有许多条目的厚书时，也许你可以要再次使用 grep 从 index.edit 的输出中提取特殊的条目，并把它们纳入要编辑的文件中。这样可以避免费力地处理无数的条目。

这个程序有个小缺点。它应该在索引中的普通文字中查找元字符，并且在正则表达式中保护它们。例如，如果索引条目包含一个星号，它将不会按星号解释，而是作为一个元字符解释。为了进行有效的改变需要使用几个高级的命令，指所以我们暂且将改进脚本的工作搁置，直到下一章中再继续进行相关的探讨。

删除

前面已经展示了删除命令（d）的示例。它采用一个地址，如果行匹配这个地址就删除模式空间的内容。

删除命令还是一个可以改变脚本中的控制流的命令。这是因为一旦执行这个命令，那么在“空的”模式空间（注 4）中就不会再有命令执行。删除命令会导致读取新输入行，而编辑脚本则从头开始新的一轮（这一行为和 next 命令的行为相同，本章后面将介绍 next 命令）。

重要的是：如果某行匹配这个地址，那么就删除整个行，而不只是删除行中匹配的部分（要删除行的一部分，可以使用替换命令并指定一个空的替换）。上一章展示了删除空行的命令：

```
/^$/d
```


运维技术书籍免费下载 https://www.linuxprobe.com

注 4： UNIX 文件写道：“不允许在被删除的行上进一步操作”。R.I.P.

删除命令的另一个用处是去除某些 troff 请求、例如添加空格，分页和填充模式的打开和关闭：

```
/^\.sp/d
/^\.bp/d
/^\.nf/d
/^\.fi/d
```

这些命令删除一个完整的行。例如，第一个命令删除行“.sp 1”或“.sp.03v”。

删除命令可以用于删除一个范围内的行。在上一章中，有一个通过删除宏.TS 和.TE 之间的行来删除文件中的所有表格的示例。还有一个删除命令（D）用于删除多行模式空间的一部分。这个高级命令将在下一章介绍。

追加、插入和更改

追加（a）、插入（i）和更改（c）命令提供了通常在交互式编辑器（例如 vi）中所选的编辑功能。你会奇怪地发现、可以使用这些相同的命令在非交互编辑器中“输入”文本。这些命令的语法在 sed 中不常用，因为它们必须在多行上来指定。语法如下：

```
追加[line-address]a\
      text
插入[line-address]i\
      text
更改[address]c\
      text
```

插入命令将所提供的文本放置在模式空间的当前行之前。追加命令将文本放置在当前行之后。更改命令用所提供的文本取代模式空间的内容。

这些命令中的每一个都要求后面跟一个反斜杠用于转义第一个行尾。text 必须从下一行开始。要输入多行文本，每个连续的行都必须用反斜杠结束，最后一行例外。例如，下面的插入命令在匹配“<Larry’s Address>”行的地方插入两行文本：

```
/<Larry’s Address>/i\
4700 Cross Court\
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
French Lick, IN
```

而且，如果文本包含一个字面的反斜杠，要再添加一个反斜杠来转义它（注 5）。

追加命令和插入命令只应用于单个行地址，而不是一个范围内的行。然而，更改命令可以处理一个范围内的行。在这种情况下，它用一个文本备份取代所有被寻址的行。换句话说，它删除这个范围中的所有行，但是提供的文本只被输出一次。例如，当下面的脚本在包含邮件消息的文件上运行时：

```
/^From /,/^$/c\
<Mail Header Removed>
```

删除整个邮件消息头并用行“<Mail Header Removed>”取代它。注意，当更改命令作为一组命令之一被封闭在大括号中并作用于一个范围内的行时，它将具有相反的功能。例如，下面的脚本：

```
/^From /,/^$/{
s/^From //p
c\
<Mail Header Removed>
}
```

将对这个范围内的每行输出“<Mail Header Removed>”。所以，当前面的示例输出文本一次时，这个示例会输出 10 次，如果在这个范围内有 10 行的话。

更改命令清除模式空间，它在模式空间中与删除命令有同样的效果。脚本中在更改命令之后的其他命令没有被提供。

插入命令和追加命令不影响模式空间的内容。提供的文本将不匹配脚本中后续命令中的任何地址，那些命令也不影响该文本。不管什么更改改变了模式空间，所提供的文本仍然会正确地输出。当默认的输出受到抑制时也是这样——所提供的文本将被输出，即使模式空间不是那样的。而且，所提供的文本不影响 sed 的内部行计数器。

注 5： 最初的 UNIX 文件提到，在提供的文本中的任何前导制表符或空格在输出上都不会出现。这是旧版本中的情况，例如 SunOS 4.1.x 和 /usr/ucb/sed on Solaris。System V 和 GNU sed 不会删除前导空白。如果它们在你的系统上消失，解决方法是在行的开始处（第一个制表符或空格之前）放置反斜杠。反斜杠不输出。

下面来看插入命令的示例。假设我们想为一个特殊文档包含的所有文件提供一

运维技术书籍免费下载 https://www.linuxprobe.com

个局部的宏文件。另外，我们想定义一个页眉字符串来将文档标识为草稿。下面的脚本在文件的第一行之前插入两行新行：

```
1i\
.so macros\
.ds CH First Draft
```

在 sed 执行这个命令之后，模式空间不会更改。其中的新文本在当前行的前面输出。后续命令不能成功地匹配“macros”或“First Draft”。

对上例进行修改可以实现在文件的结尾处添加行的追加命令：

```
$a\
End of file
```

$是行寻址符号，用于匹配文件的最后一行。提供的文本在前行之后输出，所以它成为输出中的最后一行。注意，即使只输出一行，提供的文本也必须自成一行并且与追加命令放在同一行。

下一个示例展示了在同一个脚本中使用插入命令和追加命令。其任务是在初始化列表的宏之前添加几个 troff 请求，而且在关闭列表的宏之后也添加几个。

```
/^\.Ls/i\
.in 5n\
.sp .3
/^\.Le/a\
.in 0\
.sp .3
```

插入命令在.Ls 宏之前放置两行，而追加命令在.Le 宏之后放置两行。

插入命令可以用来在当前行之前放置一个空行，或者附加命令用来在当前行之后旋转一个空行，方法是让命令后面的行为空。

更改命令可以用来在当前行之前放置一个空行，或者附加命令用来在当前行之后放置一个空行，方法是让命令后面的行为空。

更改命令用所提供的文本取代模式空间的内容。实际上，它删除行并且在该位置所提供的文本。当想要匹配行并且整个取代它时可以使用这个命令。我们来看一个示例，文件包含许多的显式 troff 空格请求（具有不同数量的空格）的情况。请看下面系列：

```
.sp 1.5
.sp
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
.sp 1
.sp 1.5v
.sp .3v
.sp 3
```

如果你想要将所有的参数都换成“.5”，那么使用更改命令比尝试匹配所有单独的参数，并进行正确的替换可能更容易。

```
/^\.sp/c\
.sp .5
```

这个命令允许我们忽略参数并取代它们，而不管它们是什么。

列表

列表命令（l）用于显示模式空间的内容，将非打印的字符显示为两个数字的 ASCII 代码。其功能类似于 vi 中的列表命令（:l）。可以使用该命令来检测输入中的“不可见”字符（注 6）。

```
$ cat test/spchar
Here is a string of special characters: ^A ^B
^M ^G
$ sed -n -e "l" test/spchar
Here is a string of special characters: \01 \02
\15 \07
$ # test with GNU sed too
$ gsed -n -e "l" test/spchar
Here is a string of special characters: \01 \02
\r \a
```

因为列表命令产生立即的输出，所以我们抑制了默认的输出，否则将得到行的重复复制。

注 6： GNU sed 显示某些字符，例如，回车符，使用的是 ANSI C 转义序列，而不是八进制。也许，对于熟悉 C 语言（或者 awk,正如我们在本书后面将看到的那样）的人来说领会这些比较容易。

在 sed 中不能用于 ASCII 值匹配字符（也不能匹配八进制数值）（注 7）。相反，在 vi 中必须找到一个组合键来产生它。使用 CTRL-V 引用该字符。例如，可以匹配一个 ESC 字符（^[）。请看下面的脚本：

```
# 列出列并用“Escape”替代“^|”：
]
s/^[/Escape/
```


运维技术书籍免费下载 https://www.linuxprobe.com

下面是一个一行的测试文件：

```
The Great ^[ is a movie starring Steve McQueen.
```

运行以上脚本产生下列输出：

```
The Great \33 is a movie starring Steve McQueen.
The Great Escape is a movie starring Steve McQueen.
```

GNU sed 产生的是这些：

```
The Great \1b is a movie starring Steve McQueen.
The Great Escape is a movie starring Steve McQueen.
```

在 vi 中通过键入 CTRL-V 然后按 ESC 键，产生字符^[。

从 nroff 文件中去除不可打印的字符

UNIX 格式化程序 nroff 产生行式打印机和 CRP 显示的输出。为了达到粗体化的特殊效果，它输出后面跟有退格键字符，然后再输出同样的字符。用文本编辑器观看它的一个示例如下：

```
N^HN^HN^HNA^HA^HA^HAM^HM^HM^HME^HE^HE^HE
```

它使单词“NAME”成为粗体。对每个字符进行 3 次重叠输出。同样，加下划线可以通过输出下划线、退格键和要加下划线的字符来完成。下面的示例是为了给单词“file”加下划线添加了一些符号：

```
_^Hf_^Hi_^Hl_^He
```

注 7： 然而，可以在 awk 中完成这些。

有时去除这些打印的“特殊效果”是有必要的，比如你可以将输出作为一个源文件。下面的行会删除用于加粗和加下划线的序列：

```
s/.^H//g
```

它可以删除退格符前面的任意字符以及这个退格符。对于加下划线的情况，“.”匹配下划线；对于加粗的情况，它匹配加粗的字符。因为它被重复调用，多次出现的重复序列被去除，每个序列只保留一个字符。注意，在 vi 中通过按下 CTRL-V 和 CTRL-H 输入“^H”。

一个应用实例就是去除老一代 System V UNIX 系统中由 nroff 所产生的 man 页中的格式信息（注 8）。如果你想用文本编辑器访问这个格式化的页面，那么

运维技术书籍免费下载 https://www.linuxprobe.com

你会想要一个整洁的版本（在许多方面，这个问题与上一章中转换字处理文件方面的问题类似）。一个带有格式的 man 面将以下面所示的样于保存到一个文件中：

```
^[9 who(1) who(1)
^[9 N^HN^HN^HNA^HA^HA^HAM^HM^HM^HME^HE^HE^HE
who - who is on the system?
S^HS^HS^HSY^HY^HY^HYN^HN^HN^HNO^HO^HO^HOP^HP^HP^HPS^HS^HS^HSI^HI
who [-a] [-b] [-d] [-H] [-l] [-p] [-q] [-r] [-s] [-t] [-T]
[-u] [_^Hf_^Hi_^Hl_^He]
who am i
who am l
D^HD^HD^HDE^HE^HE^HES^HS^HS^HSC^HC^HC^HCR^HR^HR^HRI^HI^HI^HIP^HP
who can list the user’ s name, terminal line, login time,
elapsed time since activity occurred on the line, and the
...
```

除了去掉加粗的和加下划线的序列以外，还有产生换页或各种其他的打印函数的转义序列。在以上格式化的帮助面的顶端可以看到序列“^[9”。这个转义序列可以被简单地删除：

```
a/^[9//g
```

再次强调，在 vi 中可以通过按下 CTRL-V 并随后按下 ESC 键来输入 ESC 字符。数字 9 是字面值。这里还有形成左边界和缩进的前导空格。进一步检查，在标题（例）
如“NAME”）前面有一些前导空格而每行前面在一个制表符。而且，在一些制表符意想不到地出现在文本中，这与 nroff 如何优化 CRT 屏幕上的显示有关。

注 8： 过去许多 System V UNIX 厂商只提供预先格式化的帮助面。这就允许 man 命令快速地显示信息，而不是格式化它，但是帮助面上缺少 troff 源使得很难修复文件错误。幸运的是，大多数现在的 UNIX 系统厂商都提供了参考手册的源代码

为了取消左边界和不想要的制表符，我们在上面的两个命令后添加两个命令：

```
#  sedman - -对 nroff 格式化的帮助面进行反格式化
s/.^H//g
s/^[9//g
s/^[ •]*//g
s/•/ /g
```

第三个命令寻找行开始处的任意数目的制表符或空格（制表位用“●”表示，

运维技术书籍免费下载 https://www.linuxprobe.com

空格用“□”表示）。最后一个命令寻找制表符并用一个空格取代它。在样本 msn 帮助页输出上运行这个脚本会产生下面的文件：

```
who(1) who(1)
NAME
who - who is on the system?
SYNOPSIS
who [-a] [-b] [-d] [-H] [-l] [-p] [-q] [-r] [-s] [-t] [-T]
[-u] [file]
who am i
who am I
DESCRIPTION
who can list the user’s name, terminal line, login time,
elapsed time since activity occurred on the line, and the
...
```

这个脚本不清除由分布引起的不必要的空行。我们在下一单会看到这种处理，这要求多行操作。

转换

转换命令是特有的，不仅因为它在所有的 sed 命令中拥有最小的肋记符。这个命令按位置将字符串 abc 中的每个字符，都转换成字符串 xyz 中的等价字符（注 9）。它的语法如下：

```
[address]y/abc/xyz/
```

注 9 这个命令在 UNIX tr 命令之后被模式化，被用于转换字符。这本身是一个非常有用的命令，参阅你的本地文档的详细资料。无疑、如果 t 还没有被使用（已被 test 命令使用。参见第六章“高级 sed 命令”。），那么 sed 的 y 命令就应该命名为 t。

替换根据字符的位置来进行。因此，它没有“词”的概念。这样， 在该行上的任何地方的“a”都被换成了“x”，而不管它后面是否跟有“b”。这个命令的一个可能的用处是用大写字母替换对应的小写字母，

```
y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
```

这个命令影响整个模式空间的所有内容。如果想在输入行上转换单个单词，那么通过使用保持空间可以完成。参见第六章有关如何使用保持空间的详介绍（大致过程是：输出更改单的那一行之前的所有行，删除这些行，将单词后面的行复制到保持空间，转换这个单词，然后将保持空间的内容追加到模式空间）。

打印

运维技术书籍免费下载 https://www.linuxprobe.com

打印命令（p）输出模式空间的内容。它既不清除模式空间也不改变脚本中的控制流。然而，它频繁地用在改变流控制的命令（d,N,b）之前。除非抑制（-n）默认的输出，否则打印命令将输出行的重复复制。当抑制默认的输出或者当通过程序的流控制来避免到达脚本的底部时，可能会使用它。

下面我们看一个如何使用打印命令来进行调试的脚本。它用于显示在发生任意改变之前行是什么样的。

```
#n 在行改变之前和之后打印行
/^\.Ah/{
p
s/ "//g
s/^\.Ah //p
}
```

注意，打印标志被提供给替换命令。替换命令的打印标志不同于打印命令，因为它是以成功的替换为条件的。

下面是运行上面脚本的一个例子：

```
$ sed -f sed.debug ch05
.Ah "Comment"
Comment
.Ah "Substitution"
Substitution
.Ah "Delete"
Delete
.Ah "Append, Insert and Change"
Append, Insert and Change
.Ah "List"
List
```

每个受影响的行都被打印了两次。
我们将在下一章看到打印命令的其他示例。在下一章还会看到多行打印命令（P）。

打印行号

跟在地址后面的等号（=）打印被匹配的行的行号。除非抑制行的自动输出，行号和行本身将被打印。它的语法如下：

```
[line-address]=
```

这个命令不能对一个范围内的行进行操作。

运维技术书籍免费下载 https://www.linuxprobe.com

程序员也许用该命令打印源文件中的某些行。例如，下面的脚本打印行号和行本身，这些行都包含后面跟有字符串“if”的制表符。脚本如下：

```
#n 打印具有 if 语句的行号和行
/   if/{
=
p
}
```

注意，#n 抑制行的默认输出。现在，我们看看它如何在一个样本程序 random.c 上工作的：

```
$ sed -f sedscr.= random.c
192
if( rand_type == TYPE_0 ) {
234
if( rand_type == TYPE_0 ) state[ -1 ] = rand_type;
236
if( n < BREAK_1 ) {
252
if( n < BREAK_3 ) {
274
if( rand_type == TYPE_0 ) state[ -1 ] = rand_type;
303
if( rand_type == TYPE_0 ) state[ -1 ] = rand_type;
```

在寻找由编译器报告的问题时，行号是非常有用的，编译器通常列出行号。

下一步

下一步（next）命令（n）输出模式空间的内容，然后读取输入的下 一行，而不用返回到脚本的顶端。它的语法如下：

```
[address]n
```

next 命令改变了正常的流控制，直到到达脚本的底部才会输出模式空间的内容，它总是在读入新行之后从脚本的顶端开始。实际上，next 命令导致输入的下一行取代横穿空间中的当前行。脚本中的后续命令应用于替换后的行，而不是当前行。如果没有抑制默认输出，那么在替换发生之前会打印当前行。

下面我们来看 next 命令的示例，在这个例子中，当空行跟随一个匹配模式的行时，则删除该空行。在这种情况中，假设作者已经在节标题宏（.H1）之后插入一个空行。我们想要删除这个窄而不是删除文件中所有的空行。下面是示例文

运维技术书籍免费下载 https://www.linuxprobe.com

件：

```
.H1 "On Egypt"

Napoleon, pointing to the Pyramids, said to his troops:
"Soldiers, forty centuries have their eyes upon you."
```

下面的脚本删除其中的空行：

```
/^\.H1/{
n
/^$/d
}
```

可以按下面的方式阅读这个脚本：“匹配任何以字符串‘.H1’开始的行，然后打印那一行并读入下一行。如果那一行为空，则删除它”。大括号用于在同一个地址应用多个命令。

在较长的脚本中，必须记住出现在 next 命令之前的命令不会应用于新的输入行，而且出现在后面的命令不应用于旧的输入行。
第六章将介绍有关 n 命令的其他示例，以及该命令的多行形式。

读和写文件

读（r）和写（w）命令用于直接处理文件。这两个命令都只有一个参数，即文件名。语法如下：

```
[line-address]r file
[address]w file
```

读命令将由 file 指定的文件确定的行之后的内容读入模式空间。它不能对一个范围内的行进行操作。写命令将模式空间的内容写到 file 中。

在命令和文件名之前必须有一个空格（空格后到换行符前的每个字符都被当做文件名.因此，前导的和嵌入的空格也是文件名的一部分）。如果文件不存在，读命令也不会报错。如果写命令中指定的文件不存在，将创建一个文件；如果文件已经存在，那么写命令将在每次调用脚本时改写它。如果一个脚本中有多个指令写到同一个文件中，那么每个写命令都将内容追加到这个文件中。而且，每个脚本最多只能打开 10 个文件。

读命令对于将一个文件的内容插入到另一个文件的特定位置是很有用的。例如，假设有一组文件并且每个文件都应以相同的一个或两个段落的语句结束。使用 sed 脚本可以在必要时分别单独对结束部分进行维护，例如，当将文件发

运维技术书籍免费下载 https://www.linuxprobe.com

送到打印机时：

```
sed ' $r closing' $* | pr | lp
```

$是指定文件最后一行的寻址符号。文件 closing 的内容放置在模式空间的内容之后并且和它一起输出。这个示例没指定路径名，假设文件和命令在同一个目录下。一个更通用的命令应该使用完整的路径名。

你也许想要测试读命令的几个方面。我们看看下面的命令：

```
/^<Company-list>/r company.list
```

也就是，当 sed 匹配以字符串“<Company-list>”开始的行时，它将文件 company.list 的内容附加在被匹配的行的末尾。后面的命令不会影响从这个文件中读取的行。例
如，不能对读入到文件跌公司列表进行任何改变。然而，寻址初始行的命令将会起作用。第二个命令可以跟着前一个命令：

```
/^<Company-list>/d
```

删除初始行。所以如果输入文件如下所示：

```
For service, contact any of the following companies:
<Company-list>
Thank you.
```

运行这个包含两行的脚本会产生：

```
For service, contact any of the following companies:
Allied
Mayflower
United
Thank you.
```

使用-n 选项或#n 脚本语法可以取消自动输出，阻止模式空间的初始行被输出，但是读命令的结果仍然转到标准输出。

现在我们来看写命令的例子。其中之一是从一个文件中提取信息并将它放置在自己的文件中。例如，假设有一个按字母顺序列出的销售人员名字的文件。对于每个人这个清单都指明了他被分配到 4 个区域中的哪个区域。下面是一个示例：

```
Adams, Henrietta Northeast
Banks, Freda South
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
Dennis, Jim Midwest
Garvey, Bill Northeast
Jeffries, Jane West
Madison, Sylvia Midwest
Sommes, Tom South
```

当然，为包含 7 行的文件编写一个脚本是很可笑的。然而这样一个脚本能够处理任意多的可以放在一起的名字，而且是可以重用的。

如果我们想要的是提取特定区域的名字，可以很容易地使用 grep 来完成。使用 sed 的优点是我们可以一步就将文件分成 4 个独立的文件。下面的 4 行脚本可以做这项工作：

```
/Northeast$/w region.northeast
/South$/w region.south
/Midwest$/w region.midwest
/West$/w region.west
```

所有被分配到 Northeast 地区销售员的名字，都被放置在文件 region.northeast 中。

写命令在被调用时就写出模式空间的内容，而不是等到到达脚本的结尾时才进行写操作。在上一个例子中，我们也许想在写到文件之前删除地区的名字。对于每一种情况，我们都可以像对 Northeast 地区那样来处理：

```
/Northeast$/{
s///
w region.northeast
}
```

替换命令匹配与地址相同的模式并删除它。写命令有许多不同的应用，例如，可以在脚本中使用它来生成同一源文件的几个自定义版本。

检查参考页

像许多程序一样，sed 脚本通常一开始都很小，并且写和读都很简单。在测试脚本时，可能会发现不适用于一般规则的特殊情况。为了解决这些问题，可以给脚本增加行，生成更长、更复杂并且更完整的脚本。虽然花费在细化脚本上的时间抵消了不用手动编辑而节省下来的时间，但至少在这段时间内，你的头脑被自己的这个似乎熟悉的想法占据：“看！计算机完成的”。

运维技术书籍免费下载 https://www.linuxprobe.com

在准备命令页的格式化复制中，我们会遇到这位的问题，该命令页是作者作为文本文件键入的，它没有包含任何格式化信息。虽然这个文件没有格式化代码，但使用了一致的标题来标识命令页的格式。示例文件如下所示：

```
**************************************************************
NAME: DBclose - closes a database
SYNTAX:
void DBclose(fdesc)
DBFILE *fdesc;
USAGE:
fdesc - pointer to database file descriptor
DESC:
DBclose( ) closes a file when given its database file descriptor.
Your pending writes to that file will be completed before the
file is closed. All of your update locks are removed.
*fdesc becomes invalid.
Other users are not affected when you call DBclose(). Their
update
locks and pending writes are not changed.
Note that there is no default file as there is in BASIC.
*fdesc must specify an open file.
DBclose( ) is analogous to the CLOSE statement in BASIC.
RETURNS:
There is no return value
**************************************************************
```

现在的任务是使用我们开发的引用首部宏为激光打印机格式化以上文档。因为这些命令页戊大约有 40 页，所以看完它们并手动添加代码完全是个苦差使。而且，因为有这么多页，即使作者在输入它们一般都能保持一致性，但在命令和命令之间也会有很多的差别，这就需要检查许多遍。

我们将检查构建这个 sed 脚本的过程。在某种意义上，就是仔细查看示例输入文件的每一行决定是否在那一行上进行编辑。然后查看文件的其他部分来查找类似的出现情况。我们要尽量找到能标记这些需要编辑的行或行范围的特殊模式。

例如，通过查看第一行，我们知道需要排除包含分隔每个命令的一排星号。我们指定以星号开始和结束的任意行的地址，并寻找两个星号之间的零个或多个星号。正则表达式将星号用作字面值和元字符。

运维技术书籍免费下载 https://www.linuxprobe.com

```
/^\*\**\*$/d
```

这个命令删除在文件中任何地方出现的整个星号行。我们可以看到用空行分隔段落，但是用段落宏取代每个空行会引起其他问题。在许多情况下，因为宏中已经提供了空格，所以可以删除空行。在这种情况下，我们可以在全局基础上推迟删除或取代空行直到完成了对特殊情况的处理。例如，一些空行分隔了带标记的节，我们可以使用它们定义一个行范围的结果位置。然后，将脚本的最后一个操作设计为删除不想要的空行。

制表符是一个类似的问题。制表符用于缩进语法行，在某些情况下位于标签（如
NAME）的冒号之后。我们的第一个想法是用 8 个空格来取代制表符从而删除所有的制表符，但是有一些我们想要保留的制表符，例如语法行中的那些。所以我们只删除特殊情况下的制表符，包括在行的开始处的制表符和在冒号后面的制表符。

```
/^•/s///
/:•/s//:/
```

我们遇到的下一行中包含有命令的名字和一个描述。

```
NAME: DBclose - closes a database
```

我们需要用宏.Rh 0 取代它。它的语法为：

```
.Rh 0 “command” “description”
```

我们在行的开始处插入这个宏，删除连字符并且用引号包围参数。

```
/NAME:/ {
s//.Rh 0 "/
s/ - /" "/
s/$/ "/
}
```

下面我们超前一些，来看看脚本将这些示例进行什么操作：

```
.Rh 0 "DBclose" "closes a database"
```

我们检查的下一部分以“SYNTAX”开始。这里要做和的是插入.Rh 宏和一些用于缩进、字体更改以及非填充和非调整的 troff 请求（缩进是需要的，因为我们在行的开始处删除了制表位）。这些请求必须在语法行的前后加入，将相应的功能打开和关闭。为了做到这一点，我们定义一个在两模式之间的行范围地址，这两模式标签和空行。然后，我们使用更改命令，用一系列格式化请求取

运维技术书籍免费下载 https://www.linuxprobe.com

代标签和空行。

```
/SYNTAX:/,/^$/ {
/SYNTAX:/c\
.Rh Syntax\
.in +5n\
.ft B\
.nf\
.na
/^$/c\
.in -5n\
.ft R\
.fi\
.ad b
}
```

在更改命令之后，每个输入行都以一个反斜杠结束，最后一行除外。更改从的一个副作用是当前行从模式空间中被删除了。

下面是 USAGE 部分，由一个或多个变量的相应描述组成。在这里我们将每个描述格式化成具有缩进斜体字标签的段落。首先，输出.Rh 宏，然后我们搜索由制表符和连字符分隔成两个部分的行。采用反斜杠一圆括号的形式保存每个部分，并且在替换期间回调。

```
/USAGE:/,/^$/ {
/USAGE:/c\
.Rh Usage
/\(.*\)•- \(.*\)/s//.IP "\\fI\1\\fR" 15n\
\2./
}
```

这是一个展示正则表达式的能力的很好的示例。再一次，让我们提前预览示例的输出。

```
.Rh Usage
.IP "\fIfdesc\fR" 15n
pointer to database file descriptor.
```

我们遇到的下一部分是描述。这一部分使用的空行来分隔段落。在指定这一部分的地址时，我们使用下一个标签“RETURNS”。

```
/DESC:/,/RETURNS/ {
/DESC:/i\
.LP
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
s/DESC: *$/.Rh Description/
s/^$/.LP/
}
```

我们要做的第一件事是插入段落宏，因为前面的 USAGE 部分由缩进的段落组成（在 USAGE 部分我们可能也使用了-mm 包中的变量列表宏，假如这样的话，我们应在这里插入.LE）。这只能做一次，这就是它成为“DESC”标签的关键原因。然后我们对样本文件测试 sed 脚本的这一部分时，它不起作用，这是因为在 DESC 标签后面有一个空格。我们修改正则表达式来寻找标签后面的零个或多个空格。虽然这对于样本文件是可行的，但是当我们使用比较大的样本时还有其他问题。作者对标签“SEDC”的使用不一致致。通常，它自己占据一行；有时，它位于第二段的起始处。所以我们必须增加另外的模式来处理这种情况，用于搜索后面跟有一个空格和一个或多个字符的标签。

```
s/DESC: *$/.Rh Description/
s/DESC: \(.*\)/.Rh Description\
\\1/
```

在第二种情况下，引用首部宏被输出并且后面跟有一个换行符。

下一部分（标记的“RETURNS”）采用与 SYNTAX 部分相同的方式处理。

我们更改了很少的内容，用“Return Value”取代标签“RETURNS”，并且相应地增加以下替换：

```
s/There is no return value\.*/None./
```

我们做的最后一件事是删除剩下的空行。

```
/^$/d
```

我们的脚本被插入到一个文件中，文件命名为 refsed.下面是完整的脚本：

```
# refsed - 给参考页添加格式化编码
/^\*\**\*$/d
/^•/s///
/:•/s//:/
/NAME:/ {
s//.Rh 0 "/
s/ - /" "/
s/$/"/
}
/SYNTAX:/,/^$/ {
/SYNTAX:/c\
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
.Rh Syntax\
.in +5n\
.ft B\
.nf\
.na
/^$/c\
.in -5n\
.ft R\
.fi\
.ad b
}
/USAGE:/,/^$/ {
/USAGE:/c\
.Rh Usage
/\(.*\)•- \(.*\)/s//.IP "\\fI\1\\fR" 15n\
\2./
}
/DESC:/,/RETURNS/ {
/DESC:/i\
.LP
s/DESC: *$/.Rh Description/
s/DESC: \(.*\)/.Rh Description\
\1/
s/^$/.LP/
}
/RETURNS:/,/^$/ {
/RETURNS:/c\
.Rh "Return Value"
 s/There is no return value\.*/None./
 }
 /^$/d
```

正如我们已经说明的那样，不应该让 sed 改写原始文件。最好是将 sed 的输出重定向到另一个文件或使它转到屏幕上。如果 sed 脚本不能正确地工作，那么你会发现，更改脚本并在原始文件上重新运行它，比编写一个新的脚本来纠正上一次运行导致的问题要容易。

```
$ sed -f refsed refpage
.Rh 0 "DBclose" "closes a database"
.Rh Syntax
.in +5n
.ft B
.nf
.na
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
    void DBclose(fdesc)
    DBFILE *fdesc;
    .in -5n
    .ft R
    .fi
    .ad b
    .Rh Usage
    .IP "\fIfdesc\fR" 15n
    pointer to database file descriptor.
    .LP
    .Rh Description
    DBclose( ) closes a file when given its database file descriptor.
    Your pending writes to that file will be completed before the
    file is closed. All of your update locks are removed.
    *fdesc becomes invalid.
    .LP
    Other users are not effected when you call DBclose(). Their
update
    locks and pending writes are not changed.
    .LP
    Note that there is no default file as there is in BASIC.
    *fdesc must specify an open file.
    .LP
    DBclose( ) is analogous to the CLOSE statement in BASIC.
    .LP
    .Rh "Return Value"
    None.
```

退出

退出命令（q）会使 sed 停止读取新的输入行（并停止将它们发送到输出）。它的语法为：

```
    [line-address]q
```

它只适用于单行的地址。一旦找到和 address 匹配的行，那么脚本就结束（注 10）。例如，下面一行命令使用退出命令从文件中打印前 100 行：

```
    $ sed ’ 100q’  test
    ...
```

它打印每一行，直到它到达行 100 并且退出。在这点上，这个命令的功能与 UNIX head 命令类似。

quit 的另一个可能的用法是在从文件中提取了想要的内容后退出脚本。例如，在类似 getmac（在第四章“编写 sed 脚本”中介绍）的应用中，在 sed 已经找

运维技术书籍免费下载 https://www.linuxprobe.com

到它寻找的东西之后继续扫描庞大的文件是相当低效的。
因此，我们可以按照下面的方式在 getmac shell 脚本中修订这个 sed 脚本：

```
sed -n "
/^\.de *$mac/,/^\.\.$/{
p
/^\.\.$/q
}" $file
```

以上命令组阻止了行：

```
/^\.\.$/q
```

被执行，直到 sed 找到了要寻找的宏的结尾（这一行本身在第一个宏定义结束的地方终止脚本）。sed 程序当场退出，并且不再继续遍历文件的剩余部分寻找其他可能的匹配。

注 10： 你需要小心，在将编辑操作写回到原始文件的任何程序中不要使用 q 命令，在执行 q 命令之后，就不会再产生输出。在想要编辑文件的前一部分并保留剩余部分不改变的情况下，不要全用 q 命令。在这种情况下使用 q 是初学者常犯的危险错误。

因为这个宏定义文件不长，而且脚本本身也不复杂，所以从脚本节省的时间可以忽略。然而，对于非常庞大的文件或者一个复杂的、多行脚本，它们可能只需要这个文件的很小的一部分，这种类型的脚本对时间节省意义重大。

如果比较下面的两个 shell 脚本，就会发现第一个脚本比第二个要运行得更好。下面这个简单的 shell 程序打印文件的前 10 行，然后退出：

```
for file
do
   sed 10q $file
done
```

下一个示例也打印前 10 行，它采用打印命令并抑制默认的输出：

```
for file
do
    sed -n 1,10p $file
done
```

如果你还没有这么做，那么在进入下一章高级命令之前应该使用本章出现的命令进行练习。

运维技术书籍免费下载 https://www.linuxprobe.com

第六章

高级 sed 命令

本章内容：

- 多行模式空间
- 学习案例
- 包含那一行
- 高级的流控制命令
- 加入一个短语

本章中，我们介绍剩下的 sed 命令。这些命令需要用更大的决心来掌握，而且从标准的文档中学习比学习任何基本的命令都困难。一旦你理解了这里所给出的命令，那么就可以认为自己是真正的 sed 的主人了。

高级命令分成了 3 个组：

2. 处理了多行模式空间(N、D、P)。
3. 采用保持空间来保存模式空间的内容并使它可用于后续的命令（H、h、G、g、x）。
4. 编写使用分支和条件指令的脚本来更改控制流（:、b、t）。

本章的高级脚本都做一件共同的事，那就是它们改变了执行或控制的流程顺序。通常，一行被读入模式空间并且用脚本中的每个命令（一个接一个地）应用于那一行。当达到脚本的底部时，输出这一行并且清空模式空间。然后新行被读入模式空间，并且控制被转移回脚本的顶端。这是 sed 脚本中正常的控制流。

本章中的脚本由于各种原因中断或暂停了正常的控制流。它们也许想阻止脚本中的命令被执行，某些特定的情况除外，或者阻止模式空间的内容被清除。更改控制流会使脚本更加难以阅读和理解。事实上，写脚本比读脚本更容易。当你编写很难的脚本时，测式它来看看命令如何工作对你会有好处。

我们建议你测式本章中的脚本，并且通过增加或删除命令来理解脚本是如何工作的。亲自验证结果比只是简单地阅读能更好的理解脚本。

多行模式空间

在前面正则表达式的讨论中，我们强调模式匹配是面向行的。像 grep 这样的程序尝试在单个输入行上匹配一个模式。这就使它很难匹配一个在一行的结尾处开始。并在下一行的开始处结束的短语。其他一些模式只有当在多行上重复时

运维技术书籍免费下载 https://www.linuxprobe.com

才有意义。

sed 能查看模式空间的多个行。这就是允许匹配模式扩展到多行上。在本节中，我们将来看一下创建多行模式空间并处理它的内容的命令。这里的 3 个多行命令（N、D、P）对应于上一章出现的小写字母的基本命令（n、d、p）。例如，删除命令（D）是删除命令（d）的多行形式。区别是：d 删除模式空间的内容，D 只删除多行模式空间的第一行。

追加下一行

多行 Next（N）命令通过读取新的输入行，并将它添加到模式空间的现有内容之后来创建多行模式空间。模式空间最初的内容和新的输入行之间用换行符分隔。在模式空间中嵌入的换行符可以利用转义序列“\n”来匹配。在多行模式空间中，元字符“^”匹配空间中的第一个字条，而不匹配换行符后面的字符。同样，“$”只匹配模式空间中最后的换行符，而不匹配任何嵌入的换行符。在执行 next 命令之后，控制将被传递给脚本中的后续命令。

Next 命令与 next 命令不同，next 输出模式空间的内容，然后读取新的输入行。next 命令不创建多行模式空间。

第一个示例是，我们假设想要将“Owner and Operator Guide”换成“Installation Guide”，但是我们发现它出现在文件中的两行上，“Operator”和“Guide”被分开了。

例如，下面示例文本的几行：

```
Consult Section 3.1 in the Owner and Operator
Guide for a description of the tape drives
available on your system.
```

下面的脚本寻找行结尾处的“Operator”，读取下一个输入行，然后进行替换。

```
/Operator$/{
N
s/Owner and Operator\nGuide/Installation Guide/
}
```

在这个例子中，我们知道行在哪里被拆分成两行，并且知道在哪里指定嵌入的换行符。当这个脚本在样本文件上运行中，它产生两行输出，其中的一行合并了第一行和第二行，在这显示就太长了，发生这种情况的原因是替换命令匹配了嵌入的换行符，但是没有取代它。可惜的是，我们不能用使用“\n”在替换字符串中插入换行符。你必须使用反斜杠转义换行符，如下所示：

```
s/Owner and Operator\nGuide /Installation Guide\
/
```


运维技术书籍免费下载 https://www.linuxprobe.com

这个命令在“Installation Guide”之后重新插入换行符。匹配跟在“Guide”后面的空格也是有必要的，这样新行就不会以空格开始。现在，我们来显示输出：

```
Consult Section 3.1 in the Installation Guide
for a description of the tape drives
available on your system.
```

记住，不一定要替换换行符，但是如果不替换它将会生成一些很长的行。

如果“Owner and Operator Guide”在不同的位置分成多该如何？你可以修改正则表达式来寻找单词之间的空格或换行符，如下所示：

```
/Owner/{
N
s/Owner *\n*and *\n*Operator *\n*Guide/Installation Guide/
}
```

其中的星号表示空格或换行符是可选的。但这似乎是一项艰难的工作，实际上有更加通用的方式。我们还修改了地址以匹配“Owner”，即模式中的第一个单词而不是最后一个。我们可以将换行符读入模式空间，然后使用替换命令删除嵌入的换行符，无论它在哪儿。

```
s/Owner and Operator Guide/Installation Guide/
/Owner/{
N
s/ *\n/ /
s/Owner and Operator Guide */Installation Guide\
/
}
```

第一行匹配单独占据一行的“Owner and Operator Guide”（参阅示例后面有关为什么这是必要的讨论）。如果匹配了字符串“Owner”，则将下一行读入模式空间，并且用空格取代嵌入的换行符。然后我们试着匹配整个模式并且执行后面跟有换行符的替换。这个脚本将匹配“Owner and Operator Guide"，不管它被分成怎样的两行。下面是扩展的测式文件：

```
Consult Section 3.1 in the Owner and Operator
Guide for a description of the tape drives
available on your system.
Look in the Owner and Operator Guide shipped with your system.
Two manuals are provided including the Owner and
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
Operator Guide and the User Guide.
The Owner and Operator Guide is shipped with your system.
```

在样本文件上运行上面的脚本会产生如下结果：

```
$ sed -f sedscr sample
Consult Section 3.1 in the Installation Guide
for a description of the tape drives
available on your system.
Look in the Installation Guide shipped with your system.
Two manuals are provided including the Installation Guide
and the User Guide.
The Installation Guide is shipped with your system.
```

在这个脚本中，有两个匹配模式的替换命令似乎是多余的。第一个替换命令匹配在一行上找到模式的情况，第二个替换命令匹配已有两行被读入模式空间之后的模式。为什么第一个命令是必要的？我们可以通过从脚本中删除此命令，然后同样在样本文件上运行它来得到答案：

```
$ sed -f sedscr2 sample
Consult Section 3.1 in the Installation Guide
for a description of the tape drives
available on your system.
Look in the Installation Guide
shipped with your system.
Two manuals are provided including the Installation Guide
and the User Guide.
```

你看出其中有两个问题了吗？最明显的问题是最后一行不打印。最后一行匹配“Owner",并且当执行 N 时，没有另外的输入行被读取，所以 sed 退出（立刻退出，甚至没有输出这一行）。为了修正这个问题，按如下方式使用 Next 命令应该是安全的：

```
$!N
```

它排除了对最后一行（$)执行 Next 命令。在以上的脚本中，通过匹配最后一行上的“Owner and Operator Guide”，我们避免了匹配“Owner”和应用 N 命令。然而，如果单词“Owner”出现在最后一行，我们会遇到同样的问题，除非使用“$!N”语法：

```
Look in the Owner and Operator Guide shipped with your system.
```

在上面展示的输出中，跟在“shipped with your system.”后的空行丢失了。原因是这一行匹配“Owner”而且将下一行（空行）追加到模式空间。替换命令删除嵌入的换行符，并且这个空行受到影响消失了（如果该行不为空，那么换

运维技术书籍免费下载 https://www.linuxprobe.com

行符仍然被删除，但是文件将与“shipped with your system.”出现在同一行）。最好的解决方案就是当这个模式能在一行上匹配时避免读取下一行。这就是第一条指令尝试匹配所有出现在一行上的字符串的原因。

转换 Interleaf 文件

FrameMaker 和 Interleaf 给出了 WYSIWYG(所见即所得）技术发布包。它们两个都有能力读取和保存与普通二进制文件格式相对应的 ASCII 编码格式的文档内容。在这个例子中，我们将 Interleaf 文件转换成 trooff;然而，同一类型的脚本可以应用于将 troff 编码的文件转换成 Interleaf 格式。这同样也适用于 FrameMaker.它们两个都在文件中放置编码标签，由尖括号括起来。

在这个例子中，我们的转换证明了更改命令对多行模式空间的影响。在 Interleaf 文件中，“<para>”标记一个段落。标签的前后是空行，请看样本文件：

```
<para>
This is a test paragraph in Interleaf style ASCII. Another line
in a paragraph. Yet another.
<Figure Begin>
v.1111111111111111111111100000000000000000000111111111111111000000
100001000100100010001000000100000000000000000000000000000000000000
000000
<Figure End>
<para>
More lines of text to be found after the figure.
These lines should print.
```

这个文件还包含一个位图图形，被打印成一系列 1 和 0。为了将这个文件转换成 troff 宏，我们必须用宏（.LP)取代“<para>”代码，然而，还要做一些处理，因为需要删除跟在代码后面的空行。有几种方式可以完成该功能，但是我们将使用 Next 命令创建多行模式空间，它由“<para>”和空行组成，然后使用更改命令用段落宏取代模式空间的内容。下面是完成这些工作的脚本的一部分：

```
/<para>/{
N
c\
.LP
}
```

这个地址匹配具有段落标签的行，Next 命令将下一行(应该是空行）追加到模式空间中。这里使用 Next 命令（N)而不是 next 命令（n),是因为我们不想输出模式空间的内容。更改命令改写模式空间以前的内容(后面有换行符的“<para>”，即使在模式空间包含多行的时候。

运维技术书籍免费下载 https://www.linuxprobe.com

在这个转换脚本中，我们将提取位图图形数据，并把它写到一个独立的文件中。在它的位置上，我们插入标记文件中图形的图形宏。

```
/<Figure Begin>/,/<Figure End>/{
        w fig.interleaf
        /<Figure End>/i\
.FG\
<insert figure here>\
.FE
         d
}
```

这个过程匹配“<Figure Begin>”和“<Figure End>”之间的行，并且把它们写到文件 fig.intetleaf 中。每次指令被匹配时,删除命令都将被执行，删除已经被写到文件中的行。当“<Figure End>”被匹配时，将一对宏插入到输出中图形所在的位置。注意，后续的删除命令不会影响插入命令的文件输出，然而，它从模式空间中删除“<Figure End>”。

下面是完整的脚本：

```
/<para>/{
       N
       c\
.LP
}
/<Figure Begin>/,/<Figure End>/{
    w fig.interleaf
    /<Figure End>/i\
.FG\
<insert figure here>\
.FE
    d
}
/^ $/d
```

第三个指令只删除不必要的空行（注意，这个指令可以被认为是删除跟在“<para>”标签后面的空行;但是你并不总是想删除所有的空行，并且我们想要演示跨越多行模式空间的更改命令）。

在测式文件上运行这个脚本产生的结果为：

```
$ sed -f sed.interleaf test.interleaf
.LP
This is a test paragraph in Interleaf style ASCII. Another line
in a paragraph. Yet another.
.FG
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
<insert figure here>
.FE
.LP
More lines of text to be found after the figure.
These lines should print.
```

多行删除

删除命令 (d)删除模式空间的内容并导致读入新的输入行，从而在脚本的顶端重新使用编辑方法。删除命令 (D)稍微有些不同：它删除模式空间中直到第一个嵌入的换行符的这部分内容。它不会导致读入新的输入行，相反，它返回到脚本的顶端，将这些指令应用于模式空间剩余的内容，我们可以编写一个实现查找一系列空行并输出单个空行的脚本，以看看它们之间的区别。下面的语句使用了删除命令(d)：

```
# 将多个空行减少到一行，利用 d 命令的版本
/^ $/{
N
/^ \n$/d
}
```

当遇到一个空行时，下一行就追加到模式空间中。然后试着匹配嵌入的换行符。注意定位元字符^和$分别匹配模式空间的开始处和结束处。下面是测式文件：

```
This line is followed by 1 blank line.

This line is followed by 2 blank lines.


This line is followed by 3 blank lines.



This line is followed by 4 blank lines.



This is the end.
```

在测式文件上运行这个脚本会产生下列结果：

```
$ sed -f sed.blank test.blank
This line is followed by 1 blank line.
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
This line is followed by 2 blank lines.
This line is followed by 3 blank lines.

This line is followed by 4 blank lines.
This is the end.
```

当有偶数个空行时，所有的空行都会被删除。仅当有奇数个空行时，有一行被保留下来。这是因为删除命令清除的是整个模式空间。一旦遇到第一个空行，就读入下一行，并且两行都被删除。如果遇到第三个空行，并且下一行不为空，那么删除命令就不会被执行，因此空行被输出。如果使用多行 Delete 命令（是 D 不是 d),就能得到我们想要的结果：

```
$ sed -f sed.blank test.blank
This line is followed by 1 blank line.

This line is followed by 2 blank lines.

This line is followed by 3 blank lines.

This line is followed by 4 blank lines.

This is the end.
```

多行 Delete 命令完成工作的原因是，当遇到两个空行时，Delete 命令只删除两个空行中的第一个。下一次遍历该脚本时，这个空行将导致另一行被读入模式空间。如果那行不为空，那么两行都输出，因此确保了输出一个空行。换句话说，当模式空间中有两个空行时、只有第一个空行被删除。当一个空行后面跟有文本时，模式空间可以正常输出。

多行打印

多行打印（Print）命令与小写字母的 print 命令稍有不同。该命令输出多行模式空间的第一部分，直到第一个嵌入的换行符为止。在执行完脚本的最后一个命令之后，模式空间的内容自动输出（-n 选项或#n 抑制这个默认的动作）。因此，当默认的输出被抑制或者脚本中的控制流更改，以至不能到达脚本的底部时，需要使用打印命令（P 或 p）.Print 命令经常出现在 Next 命令之后和 Delete 命令之前。这 3 个命令能建立一个输入/输出循环，用来维护两行的模式空间，但是一次只输出一行。这个循环的目的是只输出模式空间的第一行，然后返回到脚本的顶端将所有的命令应用于模式空间的第二行。没有这个循环，当执行脚本中的最后一个命令时，模式空间中的这两行都将被输出。创建多行模式空间以匹配第一行结尾处的“UNIX”和第二行开始处的“System”。如果发现“UNIX System”跨越两行，那么我们将它变成“UNIX Operating System”。建立这个循环以返回到脚本的顶端，并寻找第二行结尾处的

运维技术书籍免费下载 https://www.linuxprobe.com

“UNIX”。

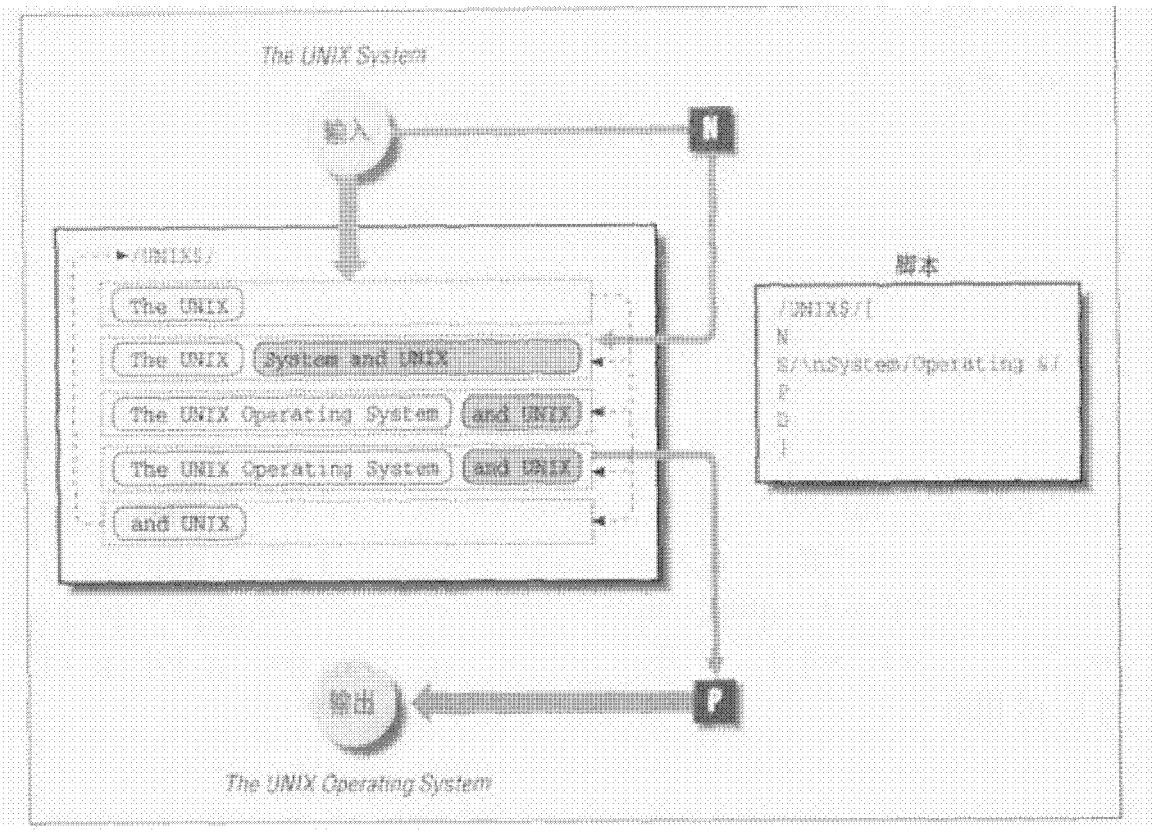

图 6-1：Next、Print 和 Delete 命令用于建立一个输入/输出循环

Next 命令将一个新的输入行追加到模式空间的当前行。在替换命令应用于多行模式空间之后，模式空间的第一部分被 Print 命令输出，然后被 Delete 命令删除。这意味着当前被输出并且新的行成为当前行。Delete 命令阻止脚本到达底部，这将输出两行并清除模式空间的内容。Delete 命令让我们保护了模式空间的第二部分，并将控制转移动脚本的顶端，在顶端所有的编辑命令都可以被应用于一行。这些命令中有一个是 Next 命令，它将另一个新行读入模式空间。

下面的脚本实现了同样的循环：

```
/UNIX$/{
N
/\nSystem/{
s// Operating &/
P
D
}
}
```

替换命令匹配“\nSystem”，并且用“Operating\nSystem”取代它。保留换行是很重要的，否则模式空间中就有只一行。注意 Print 和 Delete 命令的顺序。

运维技术书籍免费下载 https://www.linuxprobe.com

下面是测式文件：

```
Here are examples of the UNIX
System.  Where UNIX
System appears, it should be the UNIX
Operating System.
```

在测式文件上运行这个脚本会产生：

```
$ sed -f sed.Print test.Print
Here are examples of the UNIX Operating
System.  Where UNIX Operating
System appears, it should be the UNIX
Operating System.
```

输入/输出循环让我们匹配在第二行结束处出现的 UNIX。如果正常输出两行的模式空间，那么它就会丢失。

如果你还不清楚 P 和 D 命令之间的关系，在下一个例子中我们会再次提到。你还可以通过删除上面脚本中两个命令中的其中一个来进行实验，或者试试使用它们对应的小写字母命令。

学习案例

由于字形改变，Lenny(我们的职员)在将 Scribe 文档转换成 troff 宏包方面遇到了困难，他遇到的问题很有趣，这与他要完成的任务无关。

将文本设置成粗体的 Scribex 约定是：

```
@f1(put this in bold)
```

该字体更改命令可以嵌入到行中，并且可以 从一行开始并在随后的一行结束。它在一行上也可以出现多次。下面是它的几种不同用法：

```
$ cat test
I want to see @f1(what will happen) if we put the
font change commands @f1(on a set of lines). If I understand
things (correctly), the @f1(third) line causes problems. (No?).
Is this really the case, or is it (maybe) just something else?

Let’s test having two on a line @f1(here) and @f1(there) as
well as one that begins on one line and ends @f1(somewhere
on another line). What if @f1(it is here) on the line?
Another @f1(one).
```


运维技术书籍免费下载 https://www.linuxprobe.com

这个样本文件展示了字体更改命令出现的不同环境。所写的脚本必须能匹配当“@f1(anything)”出现在一行，或者在一行上出现多次，或者扩展到多行的情况。

进行单个匹配的最简单的方式是：

```
s/@f1(\(.*\))/\\fB\1\\fR/g
```

该正则表达式匹配“@f1(.*)”并使用\（和\）保存圆括号中的任意内容。在替换部分，保存的匹配部分用“\1”回调.

在 sed 脚本中插入该命令，并在样本文件上运行它。

```
$ sed -f sed.len test
I want to see \fBwhat will happen\fR if we put the
font change commands \fBon a set of lines\fR. If I understand
things (correctly), the \fBthird) line causes problems. (No?\fR.
Is this really the case, or is it (maybe) just something else?

Let’s test having two on a line \fBhere) and @f1(there\fR as
well as one that begins on one line and ends @f1(somewhere
on another line). What if \fBit is here\fR on the line?
Another \fBone\fR.
```

替换命令在前两行可以正确地工作。但它在第三行失效，在第二段的第一行也失效，在这些行上多次出现了字体更改命令。

因为正则表达式总是进行可能最长的匹配。“.*”匹配从“@f1(”开始到这一行上的最后一行右圆括号的所有的字符。换句话说。由“.*”标识的跨度在它找到的最后一个右圆括号时结束，而不是第一个右圆括号。

我们可以通过将正则表达式“.*”修改为除“）”以外的零次或多次出现的任意字符来解决这个问题，即：

```
[^)]*
```

在字符类中，脱字符（^）反转了操作的含义，所以它匹配除方括号中指定的那些字符以外的所有的字符。下面是修订后的命令：

```
s/@f1(\([^)]*\))/\\fB\1\\fR/g
```

现在，我们就有了一个命令，这一命令可以处理一行中出现一个或多个字体更改命令的情况了。

运维技术书籍免费下载 https://www.linuxprobe.com

```
I want to see \fBwhat will happen\fR if we put the
font change commands \fBon a set of lines\fR. If I understand
things (correctly), the \fBthird\fR line causes problems. (No?).
Is this really the case, or is it (maybe) just something else?
Let’s test having two on a line \fBhere\fR and \fBthere\fR as
well as one that begins on one line and ends @f1(somewhere
on another line). What if \fBit is here\fR on the line?
Another \fBone\fR.
```

这个命令可以处理除第二段中扩展到两行的情况以外的所有情况。在解决这个问题以前，看看 Lenny 的第一个解决方案以及它为什么失败是很有趣的。下面是 Lenny 的第一个脚本：

```
/@f1(/,/)/{
        s/@f1(/\\fB/g
        s/)/\\fR/g
}
```

他试图通过指定行范围来匹配跨越在多行上的出现来处理这个问题。下面是在测式文件上运行这个脚本的结果：

```
$ sed -f sed.len test.len
I want to see \fBwhat will happen\fR if we put the
font change commands \fBon a set of lines\fR. If I understand
things (correctly, the \fBthird) line causes problems. (No?\fR.
Is this really the case, or is it (maybe) just something else?

Let’s test having two on a line \fBhere) and (there\fR as
well as one that begins on one line and ends \fBsomewhere
on another line\fR. What if \fBit is here\fR on the line?
Another \fBone\fR.
```

匹配包含“）”的行会得到行上只包含圆括号的不必要的匹配。匹配多行上的模式的解决方案是创建多行模式空间，如果我们匹配“@f1(”并且没有找到右圆括号的话，那么就需要将另一行读入（N）缓冲区并试着生成与第一种情况相同的匹配（\n 表示换行符）。

```
s/@f1(\([^)]*\))/\\fB\1\\fR/g
/@f1(.*/{
        N
        s/@f1(\(.*\n[^)]*\))/\\fB\1\\fR/g
 }
```

可以对它进行测式，得出：

运维技术书籍免费下载 https://www.linuxprobe.com

```
$ sed -f sednew test
I want to see \fBwhat will happen\fR if we put the
font change commands \fBon a set of lines\fR. If I understand
things (correctly), the \fBthird\fR line causes problems. (No?).
Is this really the case, or is it (maybe) just something else?

Let’s test having two on a line \fBhere\fR and \fBthere\fR as
well as one that begins on one line and ends \fBsomewhere
on another line\fR. What if @f1(it is here) on the line?
Another \fBone\fR.
```

正如所看到的那样，我们捕捉了除了倒数第二个字体更改命令之外的所有匹配。N 命令将第二行读入模式空间。脚本匹配跨越两行的模式，然后从模式空间输出这两行。第二行会怎么样呢？它需要一个机会让脚本中从上至下的所有命令应用到它。现在，也许你明白了我们为什么需要像上一节所讨论的那样建立一个输入/输出循环。我们在脚本中增加多行 Print 命令和多行 Delete 命令。

```
# Scribe font change script.
s/@f1(\([^)]*\))/\\fB\1\\fR/g
/@f1(.*/{
        N
        s/@f1(\(.*\n[^)]*\))/\\fB\1\\fR/g
        P
        D
}
```

这可以解释为：一旦进行跨越两行的替换，就打印第一行并且把它从模式空间删除。第二部分保留在模式空间中，将控制转移到脚本的顶端，这时检查是否在该行上还有其他的“@f1(”。

修订的脚本匹配样本文件中的所有的出现。然而，它并不完美。所以我们将再次听取 Lenny 的意见。

包含那一行

模式空间是容纳当前输入行的缓冲区。还有一个称为保持空间（hold space）的顶留（set-aside）缓冲区。模式空间的内容可以复制到保持空间，而且保持空间的内容也可以复制到模式空间。有一组命令用于在保持空间和模式空间之间移动数据。保持空间用于临时存储。单独的命令不能寻址保持空间或者更改它的内容。

保持空间最常的用途是，当改变模式空间中的原始内容时，用于保留当前输入行的副本。影响模式空间的命令有：

运维技术书籍免费下载 https://www.linuxprobe.com

命令	缩写	功能
Hold	h 或 H	将模式空间的内容复制或追加到保持空间
Get	g 或 G	将保持空间的内容复制或追加到模式空间
Exchange	x	交换保持空间和模式空间的内容

这些命令中的每一条都可以利用一个地址来指定一行或行范围。Hole(h,H)命令将数据移至保持空间、而 get（g.G）命令将保持空间的数据移回到模式空间。同一命令的小写字母和大写字母之间的差别是，小字字母命令改写目的缓存区的内容，而大写字母命令追加缓存区的现有内容 。

Hold 命令用模式空间的内容取代保持空间的内容。get 命令用保持空间的内容取代模式空间的内容。

Hole 命令在保持空间的内容之后放置一个换行符，且后面跟随模式空间的内容（即使保持空间是空的，换行符也被追加到保持空间中）。Get 命令模式空间的内容之后放置一个换行符，且后面跟随保持空间的内容。

交换命令交换两个缓存区的内容，对两个缓存区没有副作用。

我们使用较通俗的示例来解释在保持空间放入行，并在稍后检索它们的情况。我们将编写一个脚本来反转部分行。我们将使用一个数字列表作为样本文件：

```
1
2
11
22
111
222
```

这里的目的是颠倒以 1 开始的行和以 2 开始的行的顺序。下面展示了如何使用保持空间：我们将第一行复制到保持空间（它一直在那），这时清除模式空间。然后 sed 将第二行读入模式空间，并且将保持空间的行追加到模式空间的结尾。请看下面的脚本：

```
# 反转 flip
/1/{
h
d
}
/2/{
G
}
```


运维技术书籍免费下载 https://www.linuxprobe.com

匹配“1”的任何行都被复制到保持空间并且从模式空间删除。控制转移到脚本的顶端并且不打印那一行。当读取下一行时，它匹配模式“2”且将已经复制到保持空间的行追加到模式空间之后。然后两行都被打印出来。换句话说，我们保存这两行中的第一行并且直到匹配第二行时才输出它

下面是在样本文件上运行这个脚本的结果：

```
$ sed -f sed.flip test.flip
2
1
22
11
222
111
```

在 hold 命令后面跟 delete 命令是一种常见的搭配。没有 delete 命令，控制将一直进行到脚本的底部，并且模式空间的内容将被输出。如果脚本中使用 next(n)命令而不是 delete 命令，那么模式空间的内容也会被输出。可以通过完全删除 delete 命令，或者在它的位置上放置 next 命令，来用这个脚本做实验。还可以看看如果使用 g 代替 G 会出现什么情况。

虽然这个对脚本对于演示这一目的是有用的，但要注意这个脚本的逻辑性是很差的，如果一行匹配第一个指令并且下一行匹配第二个指令失败，那么第一行就不会被输出。这是一个漏洞，因为它使行消失。

大写转换

上一章我们介绍了转换命令（y），并且描述了在一行上如何将小写字母转换为大写字母。因为 y 命令作用于模式空间的所有内容，所以对行的一部分进行逐字转换有点繁琐。尽管这有点令人费解，但也是可能的，正如下面的示例所演示那样。

当编写程序设计指南时，我们发现语句的名字输入不一致，它们应该都为大写字母，但是有一些是小写字母另一些的首字母却是大写的。虽然任务很简单，就是将语句的名字改成大写，但是这里有将近 100 条语句。编写如何之多的显式替换似乎是作件冗长乏味的工作。

```
s/find the Match statement/find the MATCH statement/g
```

转换命令可以进行小写字母到大写字母的转换，但它将转换应用于整个行。使用保持空间可以实现以上任务，因为可以用保持空间来存储输入行的备份而将语句名独立出来，并在模式空间进行转换。先来看看脚本：

```
# capitalize statement names
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
/the .* statement/{
h
s/.*the \(.*\) statement.*/\1/
y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
G
s/\(.*\)\n\(.*the \).*\( statement.*\)/\2\1\3/
}
```

地址将过程限制在匹配“the .* statement”的行上。让我们看看每个命令都做了些什么：

h hold 命令将当前输入行复制到保持空间。使用样本行“find the Match statrment”，我们来显示模式空间和保持空间的内容。在应用 h 命令之后，模式空间和保持空间的内容是完全相同的。

```
Pattern Space:  find the Match statement
Hold Space:     find the Match statement
s/.*the \(.*\) statement.*/\1/
```

这个替换命令从行中提取语句的名字，并且用它来取代整个行。

```
Pattern Space:  Match
Hold Space:     find the Match statement
```

y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/

这个转换命令将每个小写字母转换成大写字母

```
Pattern Space:  MATCH
Hold Space:     find the Match statement
```

G Get 命令将保存在保持空间中的行追加到模式空间。

```
Pattern Space:  MATCH\nfind the Match statement
Hold Space:     find the Match statement
```

s/\(.*\)\n\(.*the \).*\(statement.*\)/\2\1\3/

这个替换命令匹配模式空间的 3 个不同的部分：1）嵌入的换行符之前的所有的字符。2）从嵌入的换行符开始直到后面跟有一个空格的“the”，且包括 the 在内的所有字符.3)以空格并且后面跟有“statement”开始直到模式空间结尾处的所有的字符。当语句的名字出现在原始行中时，被匹配而不被保存。这个命令人替换部分回调被保存的部分，并按不同的顺序重新组合它们，在“the”和“statement”之间放置大写的命令的名字。

```
Pattern Space:  find the MATCH statement
Hold Space:     find the Match statement
```


运维技术书籍免费下载 https://www.linuxprobe.com

让我们来看看试运行。下面是样本文件：

```
find the Match statement
Consult the Get statement.
Using the Read statement to retrieve data
```

在这个样本文件上运行以上脚本结果如下：

```
find the MATCH statement
Consult the GET statement.
using the READ statement to retrieve data
```

从这个脚本可以看到，灵活地使用保持空间对于隔离和操作输入行的某部分内容很有用。

纠正索引条目（第二部分）

上一章介绍了名为 index.edit 的 shell 脚本。这个脚本从一个或多个文件中提取索引条目，并自动生成由每个索引条目的替换命令组成的 sed 脚本。我们提到的这个脚本有一个小失误，它不能找出在索引条目中心字面值出现的正则表达式的元字符，如下所示：

```
.XX "asterisk (*) metacharacter"
```

处理这个条目之后，原始的 index.edit 生成下列替换命令：

```
/^\.XX /s/asterisk (*) metacharacter/asterisk (*) metacharacter/
```

虽然它“知道“转义“.XX”前面的句点，但是它不能保护元字符“*”。问题是模式“（*）”不匹配“（*）”，因此应用替换命令将失败。解决方面是修改 index.edit，使它能寻找元字符并且转义它们。还有一种手法：在替换字符串中识别一组不同的元字符。

我们必须维护索引条目的两份拷贝。编辑的第一份备份用于转义正则表达式的元字符，然后再用于相应的模式。编辑的第二份备份转义对替换字符串来说特殊的元字符。在我们编辑第一份备份的时候，保持空间保存第二份备份。然后交换这两份备份，并编辑第二份备份。下面是脚本：

```
#! /bin/sh
# index.edit - 编辑索引项目的列表
#以便于编号匹配元字符的新版本
grep "^\.XX" $* | sort -u |
sed '
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
h
s/[][\\*.]/\\&/g
x
s/[\\&]/\\&/g
s/^\.XX //
s/$/\//
x
s/^\\\.XX \(.*\)$/\/^\\.XX \/s\/\1\//
G
s/\n//'
```

hold 命令将当前索引条目的备份放入保持空间。随后的替换命令寻找下面元字符中的任意一个：“]”，“[”， “\”， “*” 或 “.”。这一正则表达式非常有趣：1）如果右方括号是字符类中的第一个字符，那么它照字面意义解释，而不是类的右定界符;2)在指定元字符中，只有反斜杠在字符类中有特殊含义而且必须被转义。并且，不需要转义元字符“^”和“$”,因为仅当它们分别位于正则表达式的第一个或最后一个位置时才有的特殊含义，对于给定索引条目的结构，这是不可能的。转义元字符之后，交换命令交换模式空间和保持空间的内容。

从行的新备份开始 ，替换命令为替换字符串添加反斜杠来转义反斜杠或“与”符号。然后用另一个替换命令从行中删除“.XX”,随后的一条替换命令在这行末端追加一个反斜杠，产生了一个如下替换字符串。

```
"asterisk (*) metacharacter"/
```

再一次使用交换命令交换模式空间和保持空间的内容。对于模式空间的第一份备份，我们需要准备模式地址和替换模式。接下来的替换命令保存了索引条目，并且用替换命令的语法的第一部分取代这行：

```
\/^\\.XX \/s\/\1\//
```

使用样本条目，模式空间将产生下列内容：

```
/^\.XX /s/"asterisk (\*) metacharacter"/
```

然后 Get 命令获取保持空间的替换字符串，并且把它追加到模式空间。因为 Get 还插入一个换行符，所以有必要用替换命令删除它。最终输出如下的行：

```
/^\.XX /s/"asterisk (\*) metacharacter"/"asterisk (*)
metacharacter"/
```


运维技术书籍免费下载 https://www.linuxprobe.com

构建文本块

保持空间可用于在输出行块之前收集它们。一些 troff 请求和宏是面向块的，这命令中必须包围文件块。通常在块开始处的代码用启用格式，而大块结尾处的代码用禁用格式。HTML 编码的文档还包含许多面向块的结构。例如，“ ”和“ ”分别用于开始和结束一个段落。在下一个示例中，我们将看到在纯文本文件中放置 HTML 风格的段落标签。例如，输入文件包含由可变长度的行组成的段落。段落之间都有一个空行。因此，脚本必须将空行之前的所有行收集到保持空间。检索保持空间的内容并且用段落标签包围这些内容。

下面是脚本：

```
/^$/!{
H
d
}
/^$/{
x
s/^\n/<p>/
s/$/<\/p>/
G
}
```

在样本文件上运行这个脚本会产生：

```
<p>My wife won’t let me buy a power saw. She is afraid of an
accident if I use one.
So I rely on a hand saw for a variety of weekend projects like
building shelves.
However, if I made my living as a carpenter, I would
have to use a power
saw.  The speed and efficiency provided by power tools
would be essential to being productive.</p>

<p>For people who create and modify text files,
sed and awk are power tools for editing.</p>

<p>Most of the things that you can do with these programs
can be done interactively with a text editor. However,
using these programs can save many hours of repetitive
work in achieving the same result.</p>
```

这个脚本有两个基本部分，相应有各自的地址。如果输入行不是空行则做一件事，如果它是空行则做另一件事。如果输入行不是空行，它就被追加到保持空间（用 H），然后从模式空间中删除。删除命令阻止行的输出并且清除模式空

运维技术书籍免费下载 https://www.linuxprobe.com

间。控制转移回脚本的顶端并读取一个新行。总的思想是不输出任何文本行，而将它们收集到保持空间。

如果输入行是空行，则处理保持空间的内容。为了说明第二个过程，我们使用上一个样本文件的第二段并显示所发生的事情。在读怪一个空行之后，模式空间和保持空间包含下列内容：

```
Pattern Space:  ^$
Hold Space:     \nFor people who create and
modify text files, \nsed and
awk are power tools for
editing.
```

模式空间的空行表示为“^$”,即匹配它的正则表达式。保持空间的嵌入的换行符表示为“\n”.注意，Hold 命令在保持空间插入换行符，然后将当前行追加到保持空间。即使保持空间为空，Hold 命令也将在模式空间的内容之前放置换行符。

交换命令（x）交换保持空间和模式空间的内容。空行被保存在保持空间，所以我们可以在过程结束时检索它（也可以用其它方式插入换行符）。

```
Pattern Space:  \nFor people who create and
                 modify text files, \nsed and
                 awk are power tools for
                 editing.
 Hold Space:     ^$
```

接下来进行两个替换：在模式空间的开始位置放“<p>”，并且在结束位置放置“</P>”。第一个替换命令匹配“^\n”,因为以上的 Hold 命令使用换行符位于行的开始位置。第二个替换命令匹配模式空间的结尾（“$”不匹配任何嵌入的换行符而只匹配末端换行符）。

```
Pattern Space:  <p>For people who create and
                modify text files, \nsed and
                awk are power tools for
                editing.</p>
Hold Space:     ^$
```

注意，嵌入的换行符保存在模式空间中。最后一个命令 G 将保持空间的空行追加到模式空间。一旦到达脚本的底部，sed 就输出在保持空间中收集，并在模式空间中编码的段落。

该脚本说明了如何收集输入并将它保存到保持空间中，直到匹配另一个模式为止。注意脚本中的流控制是很重要的。脚本中的第一个过程并没有到达底部，

运维技术书籍免费下载 https://www.linuxprobe.com

因为我们还不想要任何输出。第二个过程到达底部，在开始收集下一段落的行之前清除模式空间和保持空间。

这个脚本还说明了如何使用寻址来建立互斥的地址，即行必须匹配互斥的地址中的一个或另一个地址。还可以设置地址来处理输入中的各种例外，并且因此改善脚本的可靠性。例如，在上一个脚本中，如果输入文件的最后一行不为空会出现什么情况？从最后一个空行以来收集到的行都不会被输出。处理这种情况有多种方式，但是较聪明的方式是构造一个空行，随后处理空行的过程会对其进行匹配处理。换句话说，如果最后一行包含一行文件，那么我们将把它复制到保持空间并用替换令清除模式空间的内容。将当前行替换成空行，使它匹配用来输出已经收集在保持空间中的内容的过程。下面就是处理过程：

```
${
/^$/!{
     H
     s/.*//
     }
 }
```

这个过程必须旋转在脚本中并在前面给出的两个过程之前。寻址符号“$”只匹配文件中的最后一行。这个过程测式不为空的行。如果行为空，那么不做任何处理。如果当前行不为空，那么就把它追加到保持空间。这就是在另一个过程中匹配非空行所做的事情。然后使用替换命令在模式空间创建一个空行。

一旦退出这个过程，在模式空间中会有一个空行。它匹配后续过程中空行并添加
HTML 分段标签，同时输出段落。

高级的流控制命令

我们已经看过了几个改变 sed 的正常流控制的示例。在本书中，我们将会看到两个用于控制执行脚本的哪一部分以及何时执行的命令。分支（b）和测式（t）命令将脚本中的控制转移到包含特殊标签的行。如果没有指定标签，则将控制转移到脚本的结尾处。分支命令用于无条件转移，测式命令用于有条件转移，它们只有当替换命令改变当前行时才会执行。

标签是任意不多于 7 个字符的序列(注 1)。标签本身占据一行并以冒号开始：

```
:mylabel
```

在冒号和标签之间不允许有空格。行结尾处的空格将被认为是标签的一部分。当在分支命令或测试命令中指定标签时，在命令和标签之间允许有空格：

```
b mylabel
```


运维技术书籍免费下载 https://www.linuxprobe.com

注意：不要在标签后面插入空格。

注 1： POSIX 标准表明具体实现工具允许有其较长的标签，GNUsed 允许标签为任意长度

分支

Branch 命令用于在脚本中将控制友转移到另一行。

```
[address]b[label]
```

Label 是可选的，如果没有给出 label,控制就被转到脚本的结尾处。如果有 label 就继续执行标签后面的行。

在第四章“编写 shell 脚本”中，介绍了将引号和连字符自身的排版形式转换成对应的排版脚本。如果想要避免改变某些行，那和可以使用分支命令跳过脚本这一部分。例如，计算机生成的由.ES 和.EE 宏标记的示例中的文本不应该被改变。因此，我们将以前的脚本改写成：

```
/^\.ES/,/^\.EE/b
s/^"/ ‘ ‘/
s/ "$/’ ’ /
s/ "? /’ ’ ? /g
.
.
.
s/\\(em\\^"/\\(em ‘ ‘/g
s/ " \\(em/’ ’ \\(em/g
s/\\(em"/\\(em ‘ ‘/g
s/@DQ@/"/g
```

因为没有提供标签，所以分支命令跳过后面所有的命令转到脚本的结尾。

分支命令可用于将一组命令作为一个过程来执行，这个过程可以从这个脚本的主体中重复调用。和上面的情况一样，它也可用于避免执行某个基于一个模式匹配的过程。

通过使用！并组合一组命令可以实现类似的效果。在应用中对分支命令使用！的优点是，可以更容易地指定要避免的多个情况。！符号可以应用于单个命令，或者应用于紧随其后的包围在大括号中的一组命令。另一方面，分支命令赋予了你几乎不受限制的脚本跳转控制能力。

例如，如果我们使用了多个宏包，那么除了.ES 和.EE 之外，还有可能有其他宏

运维技术书籍免费下载 https://www.linuxprobe.com

所定义的范围与我们所定义的相重合，这就是我们需要全力避免的。因此我们可以写出下例：

```
/^\.ES/,/^\.EE/b
/^\.PS/,/^\.PE/b
/^\.G1/,/^\.G2/b
```

为了得到 sed 脚本中更好的流控制类型的想法，我们来看看一些既简单又抽象的示例。第一个例子表明了如何使用分支命令创建循环。一旦读取一个输入行，command1 和 command2 就会应用于那一行；然后，如果模式空间的内容与指定的模式匹配，那么控制就被转移到跟在标签“top”后面的行、这意味着 command1 和 command2 将再次执行。

```
:top
command1
command2
/pattern/b top
command3
```

只有当模式不匹配时才执行 command3。所以这 3 个命令都会执行，尽管前两个命令可以多次执行。

在下一个例子中，首先执行 command1。如果模式匹配、控制权就会转移支跟在标签“end”后面的行，这意味着跳过了 command2.

```
 command1
/pattern/b end
command2
:end
command3
```

在所有的情况下，都会执行 command1 和 command3。
现在，我们来看看如何指定执行 command2 或 command3 中的一个，但不是两都都执行。在下一个脚本中，有两个分支命令。

```
command1
/pattern/b dothree
command2
b
:dothree
command3
```

第一个分支命令在模式匹配时将控制转移到 command3。如果模式不匹配，则执行 command2。跟在 command2 后面的分支命令将控制转移支脚本的结尾处，绕

运维技术书籍免费下载 https://www.linuxprobe.com

过了 command3。第一个分支命令来执行的条件是匹配模式，第二个命令执行没有条件。我们在看过 test 命令之后就会看到一个“真实的世界”的示例。

测式命令

如果在当前匹配地址的行上进行了成功的替换，那么 test 命令就会转到标签（或者脚本的结尾）处。因此，它隐含了一个条件分支。Test 命令语法如下：

```
[address]t[label]
```

如果没有给出标签 label，控制被转移支脚本的结尾处。如果提供了标签 label，那么就会继续执行标签后面的行。

我们看看来自 Tim O’Reilly 的一个示例。他试图在计算产生命令参考页顶端的宏中的参数的基本上，生成自动的索引条目。如果有 3 个引用的参数、那么需要做一些与只有两个或一个引用参数不同的事情。这个任务尝试连续地（3、2、1）匹配每一种情况并且当成功地进行替换时、避免进行进一步的替换。下面是 Tim 的脚本：

```
/\.Rh 0/{
s/ " \(.*\)" "\(.*\)" "\(.*\)"/ " \1" "\2" "\3"/
t
s/ " \(.*\)" "\(.*\)"/ " \1" "\2"/
t
s/ " \(.*\)"/ " \1"/
}
```

一旦一个替换被执行、其中的 test 命令就使控制到达脚本的末尾。如果在.Rh 行上有 3 个参数，那么第一个替换命令之后的 test 的命令为真，sed 继续进行下一个输入行。如果少于 3 个参数，就不会进行替换，test 命令为假，并且将试着执行下一个替换命令。这个过程将一直重复到所有的可能性都用完为止。

Test 命令提供的功能类似于 C 程序设计语言，或 shell 程序设计语言中的 case 语句的功能。即测试每种情况并且当一种情况为真时，退出结构。

如果上面的脚本是一个更大的脚本中的一部分，我们可以使用标签（可以形象地命名为“break”）直接转移到分组命令的末尾，然后继续执行其他一些命令。

```
/\.Rh 0/{
s/ " \(.*\)" "\(.*\)" "\(.*\)"/ " \1" "\2" "\3"/
t break
.
.
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
.
}
:break
mor e commands
```

下一节给出了 test 命令和标签的用法的一个完整的示例。

另一种情况

还记得 Lenny 吗？他提出了将 Scribe 文档转换成 troff。我们向他发送了下面的脚本：

```
# 改变 Scribe 字体的脚本.
s/@f1(\([^)]*\))/\\fB\1\\fR/g
/@f1(.*/{
N
s/@f1(\(.*\n[^)]*\))/\\fB\1\\fR/g
P
D
}
```

他使用这个脚本后回复了如下邮件：

```
Thank you so much! You’ve not only fixed the script but shown me
where I was confused about the way it works. I can repair the
conversion script so that it works with what you’ve done, but to
be
optimal it should do two more things that I can’t seem to get
working
at all—maybe it’s hopeless and I should be content with what’s
there.

First, I’d like to reduce multiple blank lines down to one.
Second, I’d like to make sed match the pattern over more than
two
(say, even only three) lines.

Thanks again.

Lenny
```

第一个请求是将一系列空行缩减为一个，这已经在本章介绍过了。下面的 4 行执行这个功能：

```
/^$/{
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
N
/^\n$/D
}
```

我们主要来看看如何完成第二个请求。前面的字体更改脚本创建了一个两行的模式空间，尝试生成跨越那些行的匹配，然后输出第一行。第二行成为模式空间中的第一行，控制被传递到脚本的顶端，接着读入另一行。

我们可以使用标签建立读取多行的循环，并且使匹配跨越多行的模式成为可能。下面的脚本设置了两个标签：位于脚本顶端的 begin 和位于脚本底部的 again。改进的脚本如下：

```
# 改变 Scribe 字体的脚本。新改进的脚本.
:begin
/@f1(\([^)]*\))/{
s//\\fB\1\\fR/g
b begin
}
/@f1(.*/{
N
s/@f1(\([^)]*\n[^)]*\))/\\fB\1\\fR/g
t again
b begin
}
:again
P
D
```

我们来进一步查看这个脚本，它有 3 部分。从跟在：begin 后面的行开始，第一部分尝试匹配完全在一行上找到的字体更改语法。在进行替换之后，分友命令将控制转移到标签 begin。换句话说，一旦我们执行了匹配。我们希望返回到顶端并寻找其它可能的匹配，包括已经被应用的指令（在一行上可能有多次出现）。

第二部分试着匹配多行上的模式。Next 命令构建了多行模式空间。替换命令尝试用模式匹配嵌入的换行符。如果匹配成功，test 命令将控制转移到 again 标签后面的行。如果没有进行替换，控制就被转移动跟在标签 begin 后面的行，所以我们就可以读入一一行。当我们匹配了字体更改请求的开始序列但还没有找到结束序列时，这个循环就生效了。Sed 将回到循环开始处并将行追加到模式空间直接到找到匹配为止。

第三部分是跟在标签 again 后面的过程，模式空间的第一行在输出后被删除。与这个脚本以前的版本相同。连续地处理多个行。控制永远不会被转移到脚本的底部。而是被 Delete 命令重定向到相同的顶端。

运维技术书籍免费下载 https://www.linuxprobe.com

加入一个短语

我们已经介绍了 sed 的所有的高级结构，现在来看一个名为 phrase 的 shell 脚本，这个脚本用到了几乎所有的 sed 的高级结构。这个脚本是通用的，类似于 grep 程序，用于寻找一系列出现在两行上的多个单词。

与 grep 一样，该程序的一个必不可少的部分是只打印匹配模式的行。你也许认为可以用-n 选项来抑制默认的行输出。然而，这个 sed 脚本不寻常之处是它创建了一个输入/输出循环，用来控制行何时输出或不输出。

脚本的逻辑是先在一行上寻找模式，如果匹配就打印这一行，如果没有找到匹配的内容，就将另一行读入模式空间(正如前面的多行脚本中那样)。然后，我们将两行的模式空间复制到保持空间以保护其内容。现在，以前被读入模式空间的新行本身可能匹配搜索模式，所以我们想要的下一个匹配只在第二行上。一旦确定在第一行或第二行上都没有找到模式，我们就删除这两行之间的替换行符并跨行寻找模式。

这个脚本被设计为可以接受命令行中的参数。第一个参数是搜索模式。所有其它的命令行参数都被解释为文件名。在分析它之前我们先来看看整个脚本：

```
#! /bin/sh
# phrase -- search for words across lines
# $1 = search string; remaining args = filenames
search=$1
shift
for file
do
sed ’
/’ "$search"’ /b
N
h
s/.*\n//
/’ "$search"’ /b
g
s/ *\n/ /
/’ "$search"’ /{
g
b
}
g
D’ $file
done
```


运维技术书籍免费下载 https://www.linuxprobe.com

名为 search 的 shell 变量被指定为命令行上的第一个参数，它表示搜索模式。这个脚本展示了将 shell 变量传递给脚本的另一种方法。这里我们用一对双引号把变量引用括起来，然后再用单引号括住它。注意脚本本身被包围在单引号中，这可以防止对 shell 特殊的字符被解释。在单引号对（注 2）中的双引号序列确保被包围的参数，首先被 shell 求值，然后再由 sed 对 sed 脚本进行处理（注 3）。

该 sed 脚本在 3 个不同的点尝试匹配搜索字符串，每一个都标有用于查找搜索模式的地址。脚本的第一行寻找在一行上出现搜索模式的行：

```
/' "$search"' /b
```

如果搜索模式匹配这一行，那么分支命令（没有标签）就将控制转移到脚本的底部并打印该行。使用 sed 的正常控制流，下一个输入行被读入模式空间，并且控制返回到脚本的顶端。每次尝试匹配模式时，都可以用相同的方式使用分支命令。

如果一个输入行不匹配这个模式，则开始下一个过程来创建多行模式空间。新行本身可以匹配这一搜索字符串。为什么这一步是必要的也许并不明显----为什么不直接寻找任何跨越两行的模式？原因是如果模式实际上在第二行被匹配，那 么我们仍然输出这两行。换句话说，用户会看到被匹配的行前面的行，因而会被它搞糊涂。这里的方法是，如果第二行匹配模式，则只输出第二行。

```
N
h
s/.*\n//
/' "$search"' /b
```

然而，如果发现一个匹配，我们不想打印整个模式空间，只想打印它的第二部分。那么当第二行匹配时，采用上面的结构将打印两行。

你也许想在尝试匹配模式之前，使用 Delete 命令删除模式空间中的第一行。Delete 命令的副作用是会更改流控制，重新从脚本顶端开始执行（Delete 命令可以大胆地使用，但是会改变脚本的逻辑）。
所以，我们尝试匹配第二行的模式，并且如果不成功，那么尝试跨越两行进行匹配：

```
g
s/ *\n/ /
/' "$search"' /{
g
b
}
```


运维技术书籍免费下载 https://www.linuxprobe.com

这里的 get 命令从保持空间获取原始的两行的一个备份，并改写模式空间中处理过的行。替换命令用一个空格取代嵌入的换行符和它前面的任意空格。然后我们试着匹配这种模式。如果匹配成功、我们不想打印模式空间的内容，只是从保持空间（保护了换行符）中得到副本并打印它。因此，在转移支脚本的结尾之前，get 命令从保持空间获取备份。

只有当模式不匹配时才执行脚本的最后一部分。

```
g
D
```

这里的 get 命令从操持空间获取保持换行符的副本。Delete 命令删除模式空间中的第一行并且将控制转移回脚本的顶端。我们只删除模式空间的第一部分，而不是清空它，因为在读取另一个输入行之后，有可以要进行跨两行的匹配。

下面是程序在样本文件上运行的结果 ：

```
$ phrase "the procedure is followed" sect3
If a pattern is followed by a \f(CW!\fP, then the procedure
is followed for all lines that do not match the pattern.
so that the procedure is followed only if there is no match
```

正如我们在开始时提到的那样，编定 sed 脚本是进行程序设计的好的开始。在接下来的章节中，我们要介绍 awk 程序设计语言。你将会看到许多和 sed 类似的内容，这会使你感到宽慰，但是你也会看到更多用于编写有用的程序的结构。当你开始用 sed 完成更复杂的任务时，所编写的脚本会变得令人很难理解，awk 的优点之一是，它更容易处理复杂问题，并且一旦你学会了 awk 的基础知识，awk 脚本就很容易编写和理解。

运维技术书籍免费下载 https://www.linuxprobe.com

本章内容

- 遵守规则
- Hello,world
- awk 程序设计模型
- 模式匹配
- 记录和字段
- 表达式
- 系统变量
- 关系操作符和布尔操作符
- 格式化打印
- 向脚本传递参数
- 信息的检索

编写 awk 脚本

在序言中曾提到，本书所描述的是 POSIX awk，也就是说，遵循 POSIX 标准的 awk 语言。在深入介绍细节之前，我们先介绍一下他的一些历史。

早期的 awk 是一个很好的小型的语言。大约在 1978 年，他首次出现在 UNIX 的第七版中，随着他的发展和流行，人们开始使用它来编写重要的程序。

在 1985 年，awk 的原作者发现，awk 被应用于比他们原来预期的更多的程序中，于是他们决定改进这种语言（参阅第十一章“awk 的系列产品”，介绍原始的 awk，以及于新版本相比较它不具备的所有功能）。这个新版本在 1987 年发布，这一版本现在一直应用的 SunOS4.1.x 操作系统上。

在 1989 年，awk 用于 System VRelease 4，并在一些小的方面做了更新（注 1）。这个版本成为 POSIX 标准中 awk 功能列表的基本组成，POSIX 解释了许多 awk 的功能凭增加了 CONVFMT 变量（将在本章后面部分介绍）。

注 1： 添加了 -v 选项以及 tolower()和 toupper()函数，删除了 srand()和 printf。这些细节将在本章和以后的章节进行详细介绍

当你在阅读余下的章节是，请你牢记术语 awk 是指 POSIX awk，而不是任何特殊的工具，不管是贝尔实验式的原始版本，还是任何的其他将在第十一章介绍的版本。但是，在某些方面不同的版本在性能上具有重要的区别，这些将在介绍的主要部分指出。

运维技术书籍免费下载 https://www.linuxprobe.com

遵守规则

要编写一个 awk 程序，就必须熟悉这个工具的规则。这些规则做了清楚的规定，可以在附录二“awk 的快速参考”中找到，而不在本章中讨论。本章的目标不是向你介绍这些规则，二是延时如何使用这些规则。用这种方式，你将会熟悉这种语言的许多特征，而且通过看例子了解脚本如何工作。有些人喜欢先阅读规则，这种方式与使用手册来学习应用程序，或通过浏览语法规则来学习语言是一样的，不是一种简单的工作。然而，熟练掌握规则是开始规范地使用 awk 的基础。但是使用 awk 越多，其规则就越快变成根深蒂固的习惯。你应反复试验来学习他们（花费很长的时间来查找一个简单的语法错误），例如漏掉一个空格或括号，这些错误对长期记忆具有神奇的效果。因此学习编写脚本的最好方法是从编写开始。当编写脚本取得一定的进步是，阅读附录二中的规则（并且反复的读它们）,或者阅读 awk 的手册，或者阅读《The AWK Programming Language》这本书无疑都将是有益的。你可以在以后做这些工作。现在让我们开始吧。

Hello,World

通过演示“Hello,World”这个程序来介绍一种程序设计语言已经成为一种管理。通过延时这个程序在 awk 中如何工作将证明 awk 是如何的不寻常。实际上，有必要演示几种打印“Hello,World”的不同方法

在第一个例子中，我们创建了一个文件，命名为 *test*,它只包含一个句子。这个例子是一个包含 print 语句的脚本：

```
$ echo ’ this line of data is ignored’ > test
$ awk ’ { print "Hello, world" }’ test
Hello, world
```

这个脚本只有一条包含在大括号中语句。这个操作对每个输入行执行 print 语句。在本里中，test 文件只包含一行，因此，print 操作只执行一次。注意这个输入行将被读入但没有被输出。

现在让我们看另外一个例子，这里使用一个包含“Hello,World.”行的文件。

```
$ cat test2
Hello, world
$ awk ’ { print }’ test2
Hello, world
```

在这个例子中，“Hello,World.”出现在输入文件中，得到了相同的结果。因为其中的 print 语句没有参数，只简单的输出每个输入行。如果文件中有其他的输入行，他们同样可以被输出。

运维技术书籍免费下载 https://www.linuxprobe.com

这两个例子都说明了 awk 是输入驱动的。也就是说，除非有可以在其上操作的输入行，否则将什么也不能做。当调用 awk 程序是，它将读入所提供的脚本，并检查其中的指令的语法。然后 awk 将对每个输入行执行脚本中的指令。因此，如果没有来自文件中的输入行，以上的 print 语句将不做任何事情。

为了验证这一点，可以输入第一个例子中的命令行，但忽略文件名。你将发现由于 awk 期望得到来自键盘的输入，所以它将一直等待知道对它的提供了输入：按几次 RETURN 键，并用 EOF（大多系统使用 CTRL-D）作为输入的结束标志。每次按 RETURN 键，打印“Hello,World.”的操作将被执行。还有另外一种方法来输出“Hello,World”信息，并且 awk 不需要等待输入。这个方法的实现和 BEGIN 模式是相关的，BEGIN 模式用于指定的第一个输入行读入之前要执行动作。

```
$ awk ’ BEGIN { print "Hello, world" }’
Hello, world
```

Awk 打印这个消息，然后退出程序。如果一个程序只有一个 BEGIN 模式，并且没有其他的语句，awk 将不处理任何输入文件。

awk 程序设计模型

理解 awk 提供给程序员的基本模型是很重要的。学习 awk 比学习其他程序设计语言更容易的部分原因，是由于 awk 为程序员提供了定义得当且有用的模型。

awk 程序是由所谓的主输入（main input）循环组成的。一个循环是一个例程，它将一直重复执行知道有一些存在的条件终止它。你不必写这个循环，他是现成的，它作为一个框架存在，在这个框架中你编写的代码能够执行。在 awk 中的主要输入循环是一个例程，它可以从文件中读取一行冰似的进程可以访问它。你所编写的处理操作的代码假设有一个可用的输入行。在其他的程序设计语言中，你必须建立一个主输入循环并将它作为程序的一个组成部分。它必须打开一个输入文件并一次读入一行。在其他的程序设计语言中，你必须建立一个主输入循环并将它作为程序的一个组成部分。它必须打开一个输入文件并一次读入一行。这并不需要许多共组，但它说明了基本的 awk 简化操作可以使得编程更容易。

主输入循环执行的次数和输入的行数相同。就像在“Hello,World.”例子中所看到的，这种循环仅当有一个输入时才执行。当没有其他输入行读入是循环将终止。

awk 允许你编写两个特殊的例程，他们在任何输入被读取前和所有输入都被读取后执行。他们是与 BEGIN 和 END 规则相关的过程。换句话说，在主输入循环执行前和主输入循环钟之后你可以做一些处理。BEGIN 和 END 过程是可选的。

运维技术书籍免费下载 https://www.linuxprobe.com

你可以把 awk 脚本看做由 3 个主要部分组成：处理输入前将做的处理，处理输入过程中将做的处理，处理输入完成后做的处理。图 7-1 解释了在 awk 脚本的控制流中这些部分之间的关系。

对于这 3 个组成部分，主输入循环或称为“处理过程中将做的处理”是主要的处理部分。在主输入循环中，指令被写成一系列的模式/动作过程。模式是用于测试输入行的规则，以确定动作是否将应用于这些输入行。我们将看到的操作可能很复杂，它由语句、函数和表达式组成。

要记住的主要事情是每个模式/操作过程位于主输入循环中个，且负责读取输入行。所编写的过程将应用于每个输入行，而且一次一行。

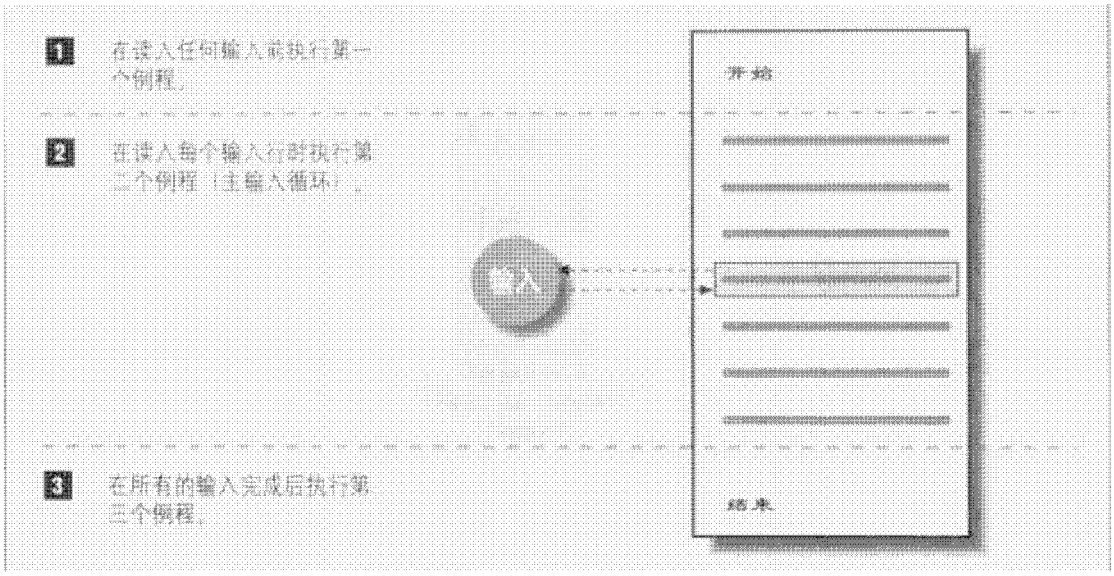

图 7-1：awk 脚本中的流程和控制

模式匹配

“Hello,World.”程序没有演示出模式匹配规则的作用。在本节中，我们将看到许多小的，甚至没有什么意义的例子，但他们能够验证 awk 程序脚本的主要特点。

当 awk 读入一行是时，它试图匹配脚本中的每个模式匹配规则。只有与一个特定的模式相匹配的输入行才能成为操作对象。如果没有指定操作，于模式相匹配的输入行将被打印出来（执行打印语句是一个默认操作）。参阅下面的脚本：

```
/^$/ { print "This is a blank line." }
```

该脚本表示：如果输入行为空，那么打印“This is a blankline”。这里的模式为一个正则表达式，用来表示一个空行。这个处理和我们以前所见到的一样，只包含一条 print 语句。

运维技术书籍免费下载 https://www.linuxprobe.com

如果我们将这个脚本放在一个称为 awkscr 的文件中，并使用名为 test 的输入文件，这个文件中包含 3 个空行，然后使下面的命令执行这个脚本：

```
$ awk -f awkscr test
This is a blank line.
This is a blank line.
This is a blank line.
```

(从这里开始，我们将假定脚本放在一个单独的文件中，并使用-f 命令行选项来调用。)这个结果告诉我们在 test 中有 3 个空行，脚本忽略了非空的行。

给以上的脚本加入几个新的规则。现在的脚本要对输入进行分析，并将他们归类为整数、字符串或空行。

#测试一下是整数、字符串还是空行。

```
/[0-9]+/ { print "That is an integer" }
/[A-Za-z]+/ { print "This is a string" }
/^$/ { print "This is a blank line." }
```

总的思想是，如果一个输入行能够和任何一个模式匹配，那么就执行相关的 print 语句。元字符+是正则表达式元字符扩展急中的一部分，它表示“一个或更多”。因此，包含一个或多个数字序列的行将被看作是一个整数。一下是一个使用标准输入的运行示例：

```
$ awk -f awkscr
4
That is an integer
t
This is a string
4T
That is an integer
This is a string
RETURN
This is a blank line.
44
That is an integer
CTRL-D
$
```

注意，输入“4T”被标识为既是整数又是字符串。一行可以匹配一条或多条规则。你可以编写一个更严格的规则以防止一行与多条规则相匹配。也可以编写操作来跳过脚本中的其他部分。

运维技术书籍免费下载 https://www.linuxprobe.com

我们通过这一章来解释模式匹配规则的应用。

程序脚本的注释

在写脚本时添加注释是一个好的习惯。注释以字符“#”开始，以换行符结束。和 sed 不同，awk 允许在程序的任何地方添加注释。

注意：如果以命令行的方式提供 awk 程序，而不是将它写入一个文件中，那么在程序的任何地方都不能用单引号，否则 shell 将对它进行解释而导致错误。当我们开始编写叫本事，我们将用注释来对脚本进行描述：

```
# blank.awk -- Print message for each blank line.
/^$/ { print "This is a blank line." }
```

注释指出了脚本的名字是 blank.awk，并简洁的描述了脚本的功能。对于较长的脚本，注释可以用来描述输入文件的预期的结构。例如，在下一节，我们将学习编写脚本来读取包含姓名和电话号码的文件。这个程序的介绍性注释如下：

```
#blodklist.awk  - 打印表格中的名字和地址。
#字段：名字、公司、街道、城市、州和邮编、电话
```

将这些信息嵌入在程序脚本中是很有用的，因此除非输入文件的结构和所编写的程序脚本中的结构一直，否则将无法工作。

记录和字段

awk 假设它的输入是有结构的，而不只是一串无规则的字符。在最简单的情况下，它将每个输入行作为一条记录，而将由空格或制表符分隔的单词作为字段（用来分隔字段的字符被称为分隔符）。下面的 names 文件中的记录有 3 个字段，由一个空格或制表符进行分隔。

```
John Robinson 666-555-1111
```

连续的两个或多个空格和/或制表符被作为一个分隔符。

字段和引用的分离

awk 允许使用字段操作符$来指定字段。在该操作符后面跟着一个数字或变量，

运维技术书籍免费下载 https://www.linuxprobe.com

用于标识字段的位置。”$1”表示第一个字段，”$2”表示第二个字段等等。“$0”表示整个输入记录。下面的例子显示了第一个字段是姓，第二个字段是名字，后面是电话号码。

```
$ awk ’ { print $2, $1, $3 }’  names
Robinson John 666-555-1111
```

$1 表示名字，$2 表示姓，而$3 表示电话号码。Print 语句中分隔每个参数的逗号使得输入的各值之间有一个空格（随后，我们将讨论输出字段分隔符（OFS），它的值中输出的逗号默认为空格）。在这个例子中，一个输入行形成包含 3 个字段的记录：在名字和姓之间有一个空格，在姓和电话号码之间有一个制表符。如果想将姓和名字结合起来作为一个字段，可以通过显式地指定字段分隔符是的只识别制表符。这样，awk 将只识别该记录中的两个字段。

可以用任何计算值为整数的表达式来表示一个字段，而不只是用数字和变量。

```
$ echo a b c d | awk ’ BEGIN { one = 1; two = 2 }
> { print $(one + two) }’
C
```

可以在命令行中使用-F 选项改变字段的分隔符。它后面跟着（或者紧跟，或者有空白）分隔符。下面的例子将字段分隔符修改为制表符。

```
$ awk -F"\t" ’ { print $2 }’  names
666-555-1111
```

“\t”是表示一个实际的制表符的转义序列（在下面讨论），它应由单引号或双引号包围着。

下面两个地址记录中的字段是由逗号分隔的。

```
John Robinson,Koren Inc.,978 4th Ave.,Boston,MA 01760,696-0987
Phyllis Chapman,GVE Corp.,34 Sea Drive,Amesbury,MA 01881,879-0900
Awk程序可以用块格式打印姓名和地址
# blocklist.awk  -  用块格式打印姓名和地址
# 输入文件—名字、公司、街道、城市、州和邮编、电话
        { print "" # output blank line
        print $1 # name
        print $2 # company
        print $3 # street
        print $4, $5 # city, state zip
}
```

第一个 print 语句指定一个空串（””）（记住 print 本身输出当前行）。这种安排使得在报告中的记录由空格隔开。我们可以执行这个脚本并使用下面的

运维技术书籍免费下载 https://www.linuxprobe.com

命令指定字段分隔符为逗号：

```
awk -F, -f blocklist.awk names
```

产生结果如下：

```
John Robinson
Koren Inc.
978 4th Ave.
Boston MA 01760
Phyllis Chapman
GVE Corp.
34 Sea Drive
Amesbury MA 01881
```

在脚本中指定域分隔符是一个好的习惯并且是非常方便的。可以通过定义系统变量 FS 来改变字段分隔符。因为这个必须在读取第一个舒服行之前执行，所以必须在由 BEGIN 规则控制的操作中指定这个变量。

```
 BEGIN { FS = "," }
```

我们现在在程序中使用它来打印出姓名和电话号码。

```
# phonelist.awk - 打印姓名和电话号码
#输入文件-名字、公司、街道、城市、州和邮编、电话
BEGIN { FS = "," } #用逗号分割字段
{ print $1 ", " $6 }
```

注意，我们在脚本中使用空行来改善可读性。在 print 语句的两个输出字段之间插入逗号和一个空格。这个程序脚本可以通过以下命令行来执行：

```
$ awk -f phonelist.awk names
John Robinson, 696-0987
Phyllis Chapman, 879-0900
```

这些给了你一个关于如何使用 awk，来处理可识别的结构化数据的基本概念。这个程序脚本用来输出所有的输入行，但我们可以编写匹配规则来修改这个操作符使得只打印出特定的名字或地址。因此，如果我们有一个长的名字列表，我们可以仅选择居住在特定州的人名。我们可以编写为：

```
/MA/ { print $1 ", " $6 }
```

这里的 MA 于马萨诸塞州邮政局的缩写相匹配。然而，MA 也可能于一个公司的名字或其他地方的名字相匹配，其中在这些公司的名字或其他地方的名字中包

运维技术书籍免费下载 https://www.linuxprobe.com

含字母“MA”.我们可以测试匹配指定的字段。使用（~）操作符可以测试一个字段的正则表达式：

```
$5 ~ /MA/ { print $1 ", " $6 }
```

可以使用组合符号（!~）来反转这个规则的意义。

```
$5 !~ /MA/ { print $1 ", " $6 }
```

这个规则将与所有其第五个字段不包含“MA”的记录相匹配一个更有挑战性的模式匹配是仅与长途电话号码相匹配。下面的正则表达式查找一个区域代码。

```
$6 ~ /1?(-|□)?\(?[0-9]+\)?(□|-)?[0-9]+-[0-9]+/
```

这个规则和下列的形式相匹配：

```
707-724-0000
(707) 724-0000
(707)724-0000
1-707-724-0000
1 707-724-0000
1(707)724-0000
```

这个正则表达式可以分段进行解释。“1？”表示出现零个或一个 1；“（-|□）？”表示在随后的位置上查找一个连字符或一个空格，或什么也没有；“\（？”表示查找零个或一个左括号；反斜杠能够防止将“（”解释为用于分组的元字符；“[0-9]+”表示查找一个或多个数字；注意我们采用了简便的方法，仅指定一到多位数字，而不是精确地指定 3 位数字。在随后的位置，我们查找一个可选的右括号，接着查找一个空格或一个连字符，或什么也没有。然后用”[0~9]+”查找一到多位数字，随后跟一个连字符，最后跟一到多位数字。

字段的划分：完整的问题

可以使用 3 个完全不同的方法使 awk 分隔字段。第一个方法是用空白字符来分隔字段。要实现这种方法，可将 FS 设置为一个空格。在这种情况下，记录的前导空白字符和结尾空白字符（空格和/或制表符）将被忽略。并且字段空格和/或制表位来分隔。因为 FS 的默认值为一个空格，所以这也是通常情况下 awk 将记录划分为字段的方法。

第二个方法是使用其他单个字符来分隔字段。例如，awk 程序经常使用“：”作为分隔符来访问 UNIX /etc/passwd 文件。当 FS 表示任何单个字符时，在这个字符出现的任何地方都将分隔出另外一个字段。如果出现两个连续的分隔符，在它们之间的字段值为空串。

运维技术书籍免费下载 https://www.linuxprobe.com

最后一种方法是，如果你设置了不止一个字符作为字段分隔符，它将被作为一个正则表达式来解释。也就是说，字段分隔符将是与正则表达式匹配的“最左边最长的非空的不重叠的”子串（注 2）。（“null 字符串”是一个技术术语，我们将它定义为“空的字符串”。）你可以发现以下各种不同表示之间的区别：

```
FS = "\t"
```

这个命令行将每个制表符作为一个字段分隔符，而：

```
FS = "\t+"
```

这个命令行表示用一个或多个制表符来分隔字段。使用第一种定义，下面的一行可以分为 3 个字段：

```
abc\t\tdef
```

而使用第二种定义，只能分隔为两个字段。可以使用一个正则表达式指定几个字符作为分隔符：

```
FS = "[’ :\t]"
```

在括号中的任何 3 个字符之一都可以被解释为字段分隔符。

注 2： 《The AWK Programming Language》[Aho],第 60 页。

表达式

可以使用表达式来存储、操作和检索数据，这些操作与 sed 中的有很大的区别，但这是大多数程序设计语言所具有的共同特性。

一个表达式通过计算返回一个值。表达式由数字和字符串常量、变量、操作符、函数和正则表达式组成。我们在第二章“了解基本操作”中详细介绍了正则表达式，并在附录二对它们做了总结。函数将在第九章“函数”中完整地介绍，在这一部分，我们将学习由常量、变量和操作符组成的表达式。

常量有两种类型：字符串型或数字型（“red”或 1）。字符串在表达式中必须用引号括起来。在字符串中可以使用在表 7-1 中列出的转义顺序。

表 7-1：转义序列

序列	描述
\a	报警字符，通常是 ASCII BEL 字符
\b	退格键

运维技术书籍免费下载 https://www.linuxprobe.com

\f	走纸符
\n	换行符
\r	回车
\t	水平制表符
\v	垂直制表符
\ddd	将字符表示为 1 到 3 位八进制
\xbex	将字符表示为十六进制值[a]
\c	任何需要字面表示的字符 c（例如，“for”）[b]

a. POSIX 没有提供“\x”，但它通常是可用的。
b. 和 ANSIC 一样，当你在没有列在这个表中的任意字符当前放置一个反斜杠时，POSIX 保留这些字符为未定义。在大多数版本的 awk 中，你就会直接得到那个字符。

变量是引用值的标识符。定义变量只需要为它定义一个名字并将数据赋给它即可。变量名只能由字母、数字和下划线组成。而且不能以数字开头。变量名的大小写很重要：Salary 和 salary 是两个不同的变量，变量不必进行说明，你不必告诉 awk 什么类型的数据存储在一个变量中。每个变量有一个字符串型值和数字型值，awk 能够根据表达式的前后关系来选择合适的值（不包含数字的字符串值为 0）。变量不必初始化。awk 自动将它们初始化为空字符串，如果作为数字，它的值为 0.下面的表达式表示将一个值赋给 x：

```
x = 1
```

x 是变量的名字、=是一个赋值操作符，1 是一个数字常量。

下面的表达式表示将字符串“Hello”赋给 z：

```
z = "Hello"
```

空格是字符串连接操作符，表达式：

```
z = "Hello" "World"
```

将两个字符串连接在一起，并将结果“HelloWorld”赋给变量 z。

美元符号（$）是引用字段操作符。下面的表达式表示把当前输入记录的第一个字段的值赋予变量 w：

运维技术书籍免费下载 https://www.linuxprobe.com

```
w = $1
```

多种操作符可以用在表达式中。表 7-2 中列出了算术操作符。

表 7-2：算术操作符

操作符	描述
+	加
-	减
*	乘
/	除
%	取模
^	取幂
**	取幂[a]

a. 这是一个普通扩展式，不再 POSIX 标准中，在系统文档中也通常没有。因此，其使用是不可移植的。

一旦变量被赋予了一个值，那么就可以用这个变量名来引用这个值。下面的表达式表示将变量 x 的值和 1 相加并将结果赋给变量 y：

```
y = x + 1
```

即计算 x 的值，使它加 1，并将结果赋给变量 y。语句：

```
print y
```

打印 y 的值。如果下面的一系列语句将出现在脚本中：

```
x = 1
y = x + 1
print y
```

那么 y 的值为 2。

我们可以将这 3 个语句减少为两个：

```
x = 1
print x + 1
```


运维技术书籍免费下载 https://www.linuxprobe.com

注意，print 语句后面的 x 的值却仍为 1。我们没有变量 x 的值，我们只是将它和 1 相加并打印结果。换句话说，如果第三个语句是 print x，那么将输出 1。实际上，如果我们想将 x 的值增加，我们可以用赋值操作符+=。这个操作符组合了两个操作符：它将 1 和 x 的值相加并将结果赋给 x。表 7-3 列出了 awk 表达式中的赋值操作符。

表 7-3：赋值操作符

操作符	定义
++	变量加 1
--	变量减 1
+=	将加的结果赋给变量
-=	将减的结果赋给变量
*=	将乘的结果赋给变量
/+	将除的结果赋给变量
%=	将取模的结果赋给变量
^=	将取幂的结果赋给变量
**=	将取幂的结果赋给变量.a

a. 和**一样，这是一个普通扩展式，它也是不可移植的。

下面的例子用于计算一个文件中空行的目录。

```
# 统计空行数
/^$/ {
print x += 1
}
```

虽然这里没有为变量 x 赋初值。但在遇到第一个空行之前它的值一直为 0。表达式 x+=1 在每次遇到空行时进行求值并将 x 的值增加 1.print 语句打印表达式返回的值。因为我们在遇到每个空行时都执行 print 语句，所以我们得到了空行数的一个连续值。

表达式可以表示为不同形式，有些和其他相比更简洁。表达式 x+=1 比等价的表达式：x=x+1 更简洁。但这两个都没有下面这个表达式简洁：

运维技术书籍免费下载 https://www.linuxprobe.com

```
++x
```

“++”是递增操作符（“_”是递减操作符）。表达式每计算一次变量的值就增加 1.递增和递减操作符可以出现在操作数的任何一边，与前缀或后缀操作符一样。位置不同可以得到不同的计算结果。

++x *在返回结果前递增x的值（前缀）*
x++ 在返回结果后递增 x 的值（后缀）

例如，如果将以上例子写为：

```
/^$/ {
print x++
}
```

当遇到第一个空行时，表达式返回的值为“0“，遇到第二个空行时返回值为“1”，依此类推。如果将递增操作符放置与 x 的前面，当表达式第一次计算后，返回的值为“1”。

下面我们在上例中使用递增表达式。另外，在每次遇到空行时不要再打印空行的数值，而是计算所有空行的值后才打印空行的总数。在 END 模式中放置 print 语句，当读完最后一个空行后打印 x 的值。

```
# 统计空行数
/^$/ {
++x
}
END {
print x
}
```

我们使用一个包含 3 个空行的样本文件来测试这个表达式。

```
$ awk -f awkscr test
3
```

程序输出了空行的数目。

计算学生的平均成绩

让我们来看另一个例子，其中先对一系列学生的成绩进行相加，然后计算其平均值。下面是输入文件的具体数据：

```
john 85 92 78 94 88
andrea 89 90 75 90 86
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
jasper 84 88 80 92 84
```

在学生的姓名后面有 5 个成绩。下面的脚本将给出每个学生的平均成绩：

```
# 求5个成绩的平均值
{ total = $2 + $3 + $4 + $5 + $6
avg = total / 5
print $1, avg }
```

该脚本将第二到第六个字段相加得到 5 个成绩的总和。将 total 的值除以 5 并结果赋给变量 avg（“/”是除非操作符）。Print 语句打印学生的姓名和平均成绩。注意我们可以省略 avg 赋值而将计算平均成绩作为 print 语句的一部分，如下：

```
print $1, total / 5
```

这个程序使我们了解了在 awk 中编写程序是如此简单。awk 将输入解析成字段和记录。你不用去读单独的字符和声明数据类型。awk 将自动替你做这些工作。

在样本数据上运行以上脚本，计算出学生的平均成绩如下：

```
$ awk -f grades.awk grades
john 87.4
andrea 86
jasper 85.6
```

系统变量

awk 中有许多系统变量或内置变量。awk 有两种类型的系统变量。第一种类型定义的变量默认值可以改变，例如默认的字段和记录分隔符。第二种类型定义的变量的值可用于报告或数据处理中。例如当前记录中字段的数量，当前记录的数量等。这些可以由 awk 自动更新，例如，当前记录的编号和输入文件名。

有一组默认值会影响对记录和字段的输入和输出的识别。系统变量 FS 定义字段分隔符。它的默认值为一个空格，这将提示 awk 可以用若干个空格和/或制表符来分隔字段。FS 可以被设置为任何单独的字符或一个正则表达式，前面，我们将分隔符改变为逗号，为的是读取一个名字和地址的列表。

和 FS 等效的输出是 OFS，它的默认值为一个空格。我们将看到一个例子来简单地重新定义 OFS。

awk 将变量 NF 定义为当前输入记录的字段个数。改变 NF 的值会有副作用。当 $0（字段）和 NF 被改变时将产生令人费解的相互作用，尤其是当 NF 减小时（注 3）。增加 NF 值会创建新的（空的）字段，并重新建立$0，字段由 OFS 的

运维技术书籍免费下载 https://www.linuxprobe.com

值来分隔。在 NF 减小的情况下，gawk 和 mawk 重新建立记录，超过新的 NF 值的字段被设置为一个空字符。Bell Labs awk 没有改变$0.

awk 还定义了记录分隔符 RS 为一个换行符。RS 有一点例外，它是 awk 仅仅注意它的值的首字符的惟一变量。

和 RS 输出等价的是 ORS，它的默认值也是一个换行符。在系一部分“处理多行记录“中，我们将解释如何改变记录分隔符的默认值。awk 设置变量 NF 为当前输入记录的编号。它可以用来给列表中的记录编号。变量 FILENAME 中包含了当前输入文件的名称。当应用多个输入文件时，变量 FNR 被用来表示与当前输入文件相关的当前记录的代码。

通常情况下，因为希望在读入第一个输入行之前设置字段和记录分隔符的值。所以可以在 BEGIN 过程中定义它们。然而，也可以在脚本的任何位置重定义它们的值，在 POSIX awk 中为 FS 赋值不影响当前的输入行，它仅影响下一个输入行。

注 3：很不幸，POSIX 标准在这一点上没有提供应有的帮助。

注意： 在 1996 年 6 月以前的 Bell Labs awk 版本中，UNIX 的 awk 版本在这点上没有遵守 POSIX 的标准，在这些版本中，如果还没有引用一个单独的字段，并将字段分隔符设置为不同的值，那么当前输入行将使用 FS 的新值分隔字段，因此，你应该测试你的 awk 的性能，如果可能，将 awk 升级为新的版本。

最后，POSIX 增加了一个新的变量 CONVFMT，它用来控制数字到字符串的转换。例如：

```
str = (5.5 + 3.2) " is a nice value"
```

这里的数字表达式 5.5+3.2（结果是 8.7）的值，必须在它被用于字符串的连接之前转换为一个字符串。CONVFMT 控制这种转换，它的默认值为“%.6g“，这是一个用于浮点型数据的 printf 风格的格式说明。例如，将 CONVFMT 改变为“%d”，将使所有的数字作为整数转变为字符串。在 POSIX 标准之前，awk 使用 OFMT 来实现这个功能。OFMT 可以做相同的工作，但是控制执行 print 语句时进行数据的转换。POSIX 委员会想将输出转换的任务从简单的字符串转换中独立出来。注意，证书转换为字符串时总是作为整数看待，而不管 CONVFMT 和 OFMT 的值是什么。

现在我们来看一些例子，它们以变量 NR 开头。将前面计算平均成绩的脚本中的 print 语句修改为：

```
print NR ".", $1, avg
```


运维技术书籍免费下载 https://www.linuxprobe.com

运行修改过的脚本输出如下结果：

```
1. john 87.4
2. andrea 86
3. jasper 85.6
```

当读入最后一行后，NF 的值是读入的输入记录的个数。它可用于 END 过程中来产生总结报告。下面是 phonelist.awk 脚本修改过的版本。

```
# phonelist.awk - 打印名字和电话号码。
# 输入文件——名字、公司、街道、城市、州和邮编、电话
BEGIN { FS = ", *" } # 用逗号分隔字段
{ print $1 ", " $6 }
END { print ""
print NR, "records processed." }
```

这个程序修改了默认的字段分隔符，并使用 NR 打印记录的总数。注意，这个程序使用了一个正则表达式来表示 NF 的值。程序执行的输出结果如下：

```
John Robinson, 696-0987
Phyllis Chapman, 879-0900
2 records processed.
```

当在 print 语句中用逗号分隔参数时，将产生输出字段分隔符（OFS）。你或许会对下面的表达式中逗号将起什么作用产生疑问：

```
print NR ".", $1, avg
```

默认情况下，逗号将在输出中产生一个空格（OFS 的默认值）。例如，你可以使用 BEGIN 过程将 OFS 重定义为制表符。那么前面的 print 语句将产生如下输出：

```
1. john 87.4
2. andrea 86
3. jasper 85.6
```

如果输入字段由制表符分隔，并且希望产生相同的输出时，这种方法特别有用。OFS 可以重定义为一系列字符，例如逗号后面跟一个空格。

另一个常用的系统变量是 NF，它的值被设置为当前记录的字段个数。就像我们将在下一部分看到的那样，可以用 NF 来测试一个记录的字段个数是否与所期望的相同。也可以用 NF 来引用每个记录的最后一个字段。使用“$”字段操作符和 NF 可以实现该引用。如果有 6 个字段，那么“$NF”与“$6”一样。假定有一下名字列表：

运维技术书籍免费下载 https://www.linuxprobe.com

```
John Kennedy
Lyndon B. Johnson
Richard Milhouse Nixon
Gerald R. Ford
Jimmy Carter
Ronald Reagan
George Bush
Bill Clinton
```

可以看出每个人的姓在记录中的字段号是不同的。这时可以用“$NF”打印每个总统的姓（注 4）。

注 4：这个方案被 Martin Van Buren 打破了；幸运的是，我们只列出了最近的美国总统。

以上介绍了基本的系统变量，也是最常用的系统变量，其他更多的系统变量列举在附录二中，下面我们将介绍在本章随后的内容中需要的新的系统变量。

处理多行记录

我们所有的例子中用到的输入文件其记录都是由单独一行组成的。在这一部分，我们将演示如何读入一个记录，而记录中的每个字段都由单独一行组成。

前面我们了解了处理姓名和地址的文件的例子。让我们假设相同的数据保存在块格式的文件中。不是将所有的信息放置在一行，而是将任命放在一行，在下一行放置公司名、一次类推。下面是一个记录样本：

```
John Robinson
Koren Inc.
978 Commonwealth Ave.
Boston
MA 01760
696-0987
```

这个记录有 6 个字段，记录之间用空行分隔。

为了处理这种包括多行数据的记录，我们可以将字段分隔符定义为换行符，换行符用“\n”来表示，并将记录分隔符设置为空字符串，它代表一个空行。

```
BEGIN { FS = "\n"; RS = "" }
```

我们可以使用下面的脚本来打印第一个和最后一个字段：

```
# block.awk - 打印第一个和最后一个字段
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
# $1 = name; $NF = phone number
BEGIN { FS = "\n"; RS = "" }
{ print $1, $NF }
```

例子的运行结果如下：

```
$ awk -f block.awk phones.block
John Robinson 696-0987
Phyllis Chapman 879-0900
Jeffrey Willis 914-636-0000
Alice Gold (707) 724-0000
Bill Gold 1-707-724-0000
```

这两个字段输出在同一行是因为默认的输出字段分隔符（OFS）仍然是一个空格。如果希望将这些字段输出在不同的行上，可以将 OFS 的值改变为一个换行符来实现。这样，你可能希望用空格将记录分隔开，因此必须将输出记录分隔符 ORS 设置为两个换行符。

```
OFS = "\n"; ORS = "\n\n"
```

支票簿的结算

这事一个简单的应用，用于处理支票登记条目。虽然不一定是最简单的结算支票簿的方法，但使用 awk 来完成某事的速度会快得令人惊奇。

假设已经输入了一个如下的文件：

```
1000
125 Market -125.45
126 Hardware Store -34.95
127 Video Store -7.45
128 Book Store -14.32
129 Gasoline -16.10
```

在第一行列出了初始的余额。其他的每一行提供了单个支票的信息：支票号，使用支票的场所和支票金额。这 3 个字段由制表符分隔，账户的数据用负数表示。于是，存款就可以用证书表示。

这个脚本的核心任务是必须得到初始的余额，并从余额中减去每个支票的金额。我们可以为每个支票提供详细的行来与支票登记内容比较。最后打印出最终的余额。代码如下：

```
# checkbook.awk
BEGIN { FS = "\t" }
#1 期望第一条记录为初始余额。
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
NR == 1 { print "Beginning Balance: \t" $1
balance = $1
next # 取得下一条记录并结束
}
#2 应用与每个支票记录，将余额与数量相加。
{ print $1, $2, $3
print balance += $3 # 支票数额有负数
}
```

运行这段程序得到的结果如下：

```
$ awk -f checkbook.awk checkbook.test
Beginning Balance: 1000
125 Market -125.45
874.55
126 Hardware Store -34.95
839.6
127 Video Store -7.45
832.15
128 Book Store -14.32
817.83
129 Gasoline -16.10
801.73
```

这个报告很难阅读，但在后面我们将学习使用 printf 语句来格式化报告的格式。最重要的是该脚本执行了我们所期望做的事情。注意，在 awk 中编写这样的脚本仅需要几分钟。如果用类似于 C 的语言来编写这样的程序将需要更长时间，其中一个原因是，你必须写更多行的代码，而且可能需要进行更低层次的编程。可以用许多精心的设计来改善这个程序，而且精炼一个程序将需要更多的时间。而使用 awk 可以很容易地将基本功能独立出来并加以实现。

关系操作符和布尔操作符

关系操作符和布尔操作符用于在两个表达式之间进行比较。表 7-4 列出了关系操作符。

表 7-4：关系操作符

操作符	描述
<	小于
>	大于
<=	小于或等于
>=	大于或等于

运维技术书籍免费下载 https://www.linuxprobe.com

==	相等的
!=	不等的
~	匹配
!-	不匹配

关系表达式可用在模式中来控制特殊的操作。例如，如果我们想限定要处理的记录包含 5 个字段，则可以用下面的表达式：

```
NF == 5
```

这个关系表达式将 NF（每个输入记录的字段数）的值和 5 想比较。如果结果为真，那么就进行相应的处理，否则不进行处理。

注意： 关系操作符“==（相等）”和赋值操作符“=（等于）”是不同的。用“=”代替“==”来检测相等性是一个普通的错误。

我们可以在试图打印数据库 phonelist 的记录之前用一个关系表达式来检测。

```
NF == 6 { print $1, $6 }
```

只有具有 6 个字段的记录才能被打印。

和“==”相反的是“！=（不相等的）”。同样地，可以比较一个表达式是否大于（>）或小于(<)，或大于等于（>=），或小于等于（<=）另一个表达式。如下的表达式：

```
NR > 1
```

检测当前记录号是否大于 1，在下章我们将看到，关系表达式经常用在 if 语句中，通过计算来决定是否执行特殊的操作。

正则表达式经常用斜杠包围。这经常被作为正则表达式常量，正如“Hello”是一个字符串常量一样。我们已经看到很多这样例子：

```
/^$/ { print "This is a blank line." }
```

然而，也常常不局限于正则表达式常量。当使用关系操作符~(匹配)或！~（不匹配）时，右边的表达式可以是 awk 中的任意表达式；awk 将它作为一个字符串并用来指定一个正则表达式（注 5）。我们曾学习了一个使用~操作符的例子，用于电话号码数据库中模式匹配规则。

运维技术书籍免费下载 https://www.linuxprobe.com

```
$5 ~ /MA/ { print $1 ", " $6 }
```

这个语句是将第五个字段的值与正则表达式“MA”比较。

注 5： 当调用 match()、split()、sub()个 gsub()函数时，也可以使用字符串代替正则表达式常量。

因为所有的表达式都与~和！~一起使用，因此正则表达式可以用变量来提供。例如，在脚本 phonelist 中，我们可以用 state 来代替“/MA/”，并编写一个过程来定义 state 的值。

```
$5 ~ state { print $1 ", " $6 }
```

这使得程序代码更加通用，因为在脚本执行过程中可以动态改变模式。例如，我们可以从命令行参数得到 state 的值，在本章的后面我们将讨论如何将命令行参数传递给脚本。

使用布尔操作符可以将一系列的比较组合起来。表 7-5 列出了布尔操作符。

表 7-5：布尔操作符

操作符	定义
||	逻辑或
&&	逻辑与
!	逻辑非

给定两个或多个表达式，只有当给定的表达式之一的值为真（非零或非空）时，使用操作符||的等个表达式的值才为真。而只有当&&操作符连接的两个表达式的值都为真时结果才为真。

下面的表达式：

```
NF == 6 && NR > 1
```

表示字段的数量必须等于 6 并且记录的编号必须大于 1.

&&比||的优先级别高。你能说出下面的表达式的计算结果吗？

```
NR > 1 && NF >= 2 || $1 ~ /\t/
```

下面的例子用圆括号表明了基于优先规则那个表达式将首先被求值。

```
(NR > 1 && NF >= 2) || $1 ~ /\t/
```


运维技术书籍免费下载 https://www.linuxprobe.com

换句话说，圆括号中的两个表达式必须都为真或圆括号右边的表达式为真。可以用圆括号来改变优先规则，例如，下面的例子规定两个条件必须都为真。

```
NR > 1 && (NF >= 2 || $1 ~ /\t/)
```

第一个条件必须为真，而且另外两个条件中必须有一个为真。

无法一个表达式的值为真或为假，操作符！都对其值取反。

```
! (NR > 1 && NF > 3)
```

如果圆括号中的表达式的值为假，那么上面的表达式的结果就为真。这个操作符与 awk 的 in 操作符结合起来非常有用，可用来判断某个下标否在数组中（在后面我们将会看到）。当然它还有其他的用途。

获取文件的信息

现在我们来学习处理 UNIX 命令 ls 的输出的几个脚本。下面的执行命令 ls -l 得到的一个长列表样本（注 6）：

```
$ ls -l
-rw-rw-rw- 1 dale project 6041 Jan 1 12:31 com.tmp
-rwxrwxrwx 1 dale project 1778 Jan 1 11:55 combine.idx
-rw-rw-rw- 1 dale project 1446 Feb 15 22:32 dang
-rwxrwxrwx 1 dale project 1202 Jan 2 23:06 format.idx
```

这个列表是一个报告，其中的数据按行和列显示。每个文件信息显示在单独的一行上。文件列表由 9 个列组成。文件的操作权限出现在第一列，文件的字节数显示在第五列，文件名显示在最后一列。在列之间由一个或多个空格来分隔，我们可以将每个列看做是一个字段。

在第一个例子中，我们将这个命令的输出结果导入一个 awk 脚本中，该脚本打印出该文件列表中选定的字段。为了完成这项操作，我们将创建一个 shell 脚本使得能够将数据输送到用户。因此，shell 程序的结构是：

```
ls -l $* | awk ' script'
```

shell 使用$*变量来扩展通过命令行传递的所有变量（这里可以使用$1 来传递第一个变量，但是传递所有的变量将具有更大的灵活性）。这些参数可能是文件名、目录或 ls 命令的附加选项。如果没有指定参数，“$*”将为空并且显示当前目录。因此 ls 命令的输出可以传给 awk，即使没有给出文件名也能自动读取标准的输入。

注 6： 注意在 Berkeley 4.3BSD 派生的 UNIX 系统上，例如 Ultrix 或

运维技术书籍免费下载 https://www.linuxprobe.com

SunOS4.1x,ls -l 产生一个 8 列的报告；使用 ls -lg 得到与这里显示的相同的报告格式。

我们希望 awk 脚本能够打印文件的大小和名字。即打印第五个字段（$5）和第九个字段（$9）。

```
ls -l $* | awk ' {
print $5, "\t", $9
}'
```

如果将上面的代码保存在文件 fls 中并运行该文件，则可以将 fls 作为一个命令来输入：

```
$ fls
6041 com.tmp
1778 combine.id
1446 dang
1202 format.idx
$ fls com*
6041 com.tmp
1778 combine.idx
```

以上程序所做的工作就是读入一个长列表并将其减少为两个字段。现在我们来为所产生的报告，增加一些新的功能以产生一些信息，这是 ls -l 列表所没有提供的。我们将每个文件的大小相加，得到列表中所有文件的总字节数。我们还能够跟踪文件的数量并计算出总数。增加这些功能包括两部分。首先累计每个输入行。我们创建变量 sum 来累加列表中文件的大小，用变量 filenum 来累加列表中文件的数量。

```
{
sum += $5
++filenum
print $5, "\t", $9
}
```

第一个表达式使用赋值操作符+=。它的功能是将第五个字段的值加到变量 sum 的当前值上。第二个表达式递增变量 filenum 的值。该变量作为一个计数器，每次表达式计算一次，计数器加 1.

我们所编写的操作将应用于所有的输入行。当 awk 读入所有的输入行后，所产生的总数必须打印出来，因此，我们编写了一个由 END 规则控制的操作：

```
END { print "Total: ", sum, "bytes (" filenum " files)" }
```


运维技术书籍免费下载 https://www.linuxprobe.com

我们还可以使用 BEGIN 规则给报告增加列标题。

```
BEGIN { print "BYTES", "\t", "FILE" }
```

现在，我们将这些程序代码防御文件 filenum 中并作为一个单词命令运行。

```
$ filesum c*
BYTES FILE
882 ch01
1771 ch03
1987 ch04
6041 com.tmp
1778 combine.idx
Total: 12459 bytes (5 files
```

这个命令的优点是能够确定一个目录或某个文件组中文件的大小。

在显示基本机制之后，还需要注意几个问题。第一个问题出现在当使用 ls -l 命令显示整个目录时，列表中包含一个指定目录中块的总数的行。在前面例子（所有以“c”开头的文件）中的部分列表中没有包含这一行。但是如果整个目录都列出来，那么在输出中将会包含下面的一行：

```
total 555
```

我们对块总数并不感兴趣，因为程序显示了文件的总的大小。现在 filesum 没有打印这一行，但是它读入了这一行并使计数器 filenum 得到了递增。

这个脚本还有一个问题，也就是它如何处理子目录。参见下面 ls -l 执行结果中的一行：

```
drwxrwxrwx 3 dale project 960 Feb 1 15:47 sed
```

第一列（文件的操作权限）中的首字符“d”表示该文件是子目录。文件的大小（960B）并不表示子目录下的文件的大小，因此，经常错误地把这个数加到文件总的大小上。因此，指出这是一个目录可能是有用的。

如果想列出子目录中的文件，可以在命令行中提供-R（递归）选项。该选项将传递给 ls 命令。然而，当它识别每个目录时，列表有一些区别。例如，要识别子目录 old，ls -lR 将产生一个空行，气候跟随：

```
./old:
```

我们前面的脚本忽略了这行及其前面的空行，但是仍然递增文件计数器。幸运的是，我们可以设计规则来处理这类情况。我们来看下面修改后的、加了注释

运维技术书籍免费下载 https://www.linuxprobe.com

的脚本：

```
ls -l $* | awk '
# filesum: 列出文件和总的字节数
# 输入: 由命令“ls -l“生成的长列表
#1输出列的标题
BEGIN { print "BYTES", "\t", "FILE" }
#2 测试9个字段, 文件以"-"开始
NF == 9 && /^-/ {
sum += $5 # 累计文件大小
++filenum # 系统文件个数
print $5, "\t", $9 # 打印大小和文件名
}
#3 测试9个字段, 目录由 "d"开始
NF == 9 && /^d/ {
print "<dir>", "\t", $9 # print <dir> and name
}
#4 测试 ls -lR line ./dir:
$1 ~ /^\..*:$/ {
print "\t" $0 #打印用制表符处理的行
}
#5 所有工作已完
END {
# 打印所有文件总的大小和文件数目
print "Total: ", sum, "bytes (" filenum " files)"
}'
```

这里对规则以及与之相关的操作设置了编号，以便进行讨论。由 ls -l 产生的列表包含文件的 9 个字段。Awk 在系统变量 NF 中给出了一个记录中的字段的个数。因此，规则 2 和规则 3 测试 NF 是否等于 9.这可以使我们避免与奇数空行或表示块总量的行匹配。因为我们希望对目录和文件做不同的处理，所以我们使用另一个模式来匹配行的第一个字符。在规则 2 中我们测试行的第一个位置的字符“-“，这表示一个文件。如果匹配则递增与之相关的文件计数器并累计文件的大小。在规则 3 中我们测试一个目录，用”d”作为第一个字符来表示。相关的操作是在文件大小的位置上打印”<dir>”。规则 2 和规则 3 是符合表达式，用&&操作符将两个模式结合起来。两个模式都必须匹配表达式的值才为真。

规则 4 用于测试由 ls -lR 产生的列表的特殊形式(“./old;”)。我们可以使用正则表达式或关系表达式来编写一些匹配这一行的模式。

NF == 1 *如果域的个数等于1 ...*

/^\..*:$/ *如果行以句点开头，其后跟有任意数量的字符并以冒号结束...*

运维技术书籍免费下载 https://www.linuxprobe.com

```
$1 ~ /^\..*:$/            如果域上匹配正则表达式...
```

我们采用后面的表达式，因为它似乎是最具有针对性的。它使用匹配操作符（~）来测试第一个字段是否匹配一个正则表达式。相关的操作只由一个 print 语句组成。

规则 5 是 END 模式，它的操作只被执行一次，用于打印出文件总的尺寸和数量。

程序 filesum 演示了 awk 中的许多基本的结构。更重要的是，它提供了如何编写程序的一个很好的思路（尽管由于排版和草率的想法产生的语法错误被忽略了）。如果想修改以上程序，可以为目录增加一个计数器，或简历一个规则来处理符号连接。

格式化打印

到现在为止，我们编写的许多脚本可以很好地实现对数据的操作，但没有对输出进行适当的格式化。这是因为基本的 print 语句所能做的工作有限。因为 awk 的大多数功能是产生报告，因此以整齐的样式产生格式化报告是很重要的。程序 filesum 可以很好地处理数据，但它的报告缺乏整齐的格式。

awk 提供的 printf 可以代替 print 语句，printf 是借用了 C 程序设计语言。Printf 语句和 print 语句一样可以打印一个简单的字符串。

```
awk ' BEGIN { printf ("Hello, world\n") }'
```

首先可以看出，printf 和 print 的主要区别是 printf 没有提供自动换行功能。必须明确地为它指定“\n”。

Printf 语句的完整语法由两部分组成：

printf (*for mat-expression* [, *ar guments*])

其中的圆括号是可选的。第一部分是一个用来描述格式的表达式，通常以引号括起的字符串常量的形式提供。第二部分是一个参数列表，例如变量名列表，它和格式说明相对应。在格式说明前面有一个百分号（%），而格式说明符号为表 7-6 列出的字符之一。两个主要的格式说明符是 s 和 d，s 表示字符串，d 表示十进制整数（注 7）。

表 7-6：用在 printf 的格式说明符

字符	定义
c	ASCII 字符
d	十进制整数

运维技术书籍免费下载 https://www.linuxprobe.com

i	十进制整数（在 POSIX 中增加的）
e	浮点格式（[-]d.precisione[+-]dd）
E	浮点格式（[-]d.precisionE[+-]dd）
f	浮点格式（[-]ddd.precision）
g	e 或 f 的转换形式，长度最短，末尾的 0 被去掉
G	E 或 f 的转换形式，长度最短，末尾的 0 被去掉
O	无符号的八进制
s	字符串
u	无符号的十进制
x	无符号的十六进制，用 a-f 表示 10-15
X	无符号的十六进制，用 A-F 表示 10-15
%	字面字符%

下例在程序 filenum 的规则 2 中用 Printf 产生一个输出。它输出不同的两个字段上的字符串和十进制值：

```
printf("%d\t%s\n", $5, $9)
```

该语句输出$5 的值，后面是制表符\t 和$9、然后输出一个换行符（\n）（注 8）。对每个格式说明必须提供一个相应的参数。

注 7： printf 进行舍入的方式在附录二中讨论。

注 8： 将这条语句与 filesum 程序中打印标题行的语句进行比较。Print 语句自动提供换行符（ORS 的值）；当使用 printf 时，你必须提供换行符，它永远不会自动产生。

Printf 语句可以规定输出域的宽度和对齐方式。一个格式表达式由 3 个可选的修饰符组成，跟在“%”后面，并出现在格式说明符之前。

```
%-width.precision format-specifier
```

描述输出字段宽度的 width 是一个数值。当指定域宽度时，这个域的内容默认为向右对齐。必须指定“-”来设置左对齐。因此，“%-20s”输出的是向左对

运维技术书籍免费下载 https://www.linuxprobe.com

齐的一个域长度为 20 个字符的字符串，如果字符串少于 20 个字符，那么这个域将用空格来填满。在下面的例子中，输出一个“|”来指示输出域的真实长度。第一个例子是右对齐的文本：

```
printf("|%10s|\n", "hello")
```

结果是：

```
|  hello|
```

下一个例子是左对齐的文本：

```
printf("|%-10s|\n", "hello")
```

结果是：

```
|hello  |
```

precision 修饰符用于十进制或浮点数，用于控制小数点右边的数字位数。对于字符串型值，它用于控制要打印的字符的最大数量。

注意，数值的默认 precision 值为“%.6g”。

可以根据 print 或 printf 的参数列表中的值，动态地指定宽度 width 和精度 precision。通过用星号实际的值来实现这个功能：

```
printf("%*.*g\n", 5, 3, myvar);
```

在这个例子中，宽度是 5，精度为 3，要打印的值来自 myvar。

Print 语句输出数值的默认精度可以通过设置系统变量 OFMT 来改变。例如，如果使用 awk 打印报告，其中包含美元（$）数值，可以将 OFMT 设置为“%.2f”。

使用格式表达式的完整语法可以解决 filesum 中的各个字段和标题的对齐问题。我们在文件名前输出文件大小的一个原因，就是以这种顺序输出字段对齐的可能性更大，在很大程度上它们可以自己对齐。Printf 提供给我们的解决方法能够固定输出域的宽度，因此，每个域在相同的列开始。

我们来重新调整 filesum 报告的输出域。我们希望得到最小的域宽度使得第二个域在相同的位置开始。域宽度应放置在%和转换说明符之间。“%-15s”规定域的宽度为 15 个字符并且字符左对齐。“%10d”中没有连字符，是右对齐的，这就是我们所希望的十进制值的表示。

运维技术书籍免费下载 https://www.linuxprobe.com

```
printf("%-15s\t%10d\n", $9, $5) # 打印文件名和大小
```

这将产生一个报告，其中的数据按列对齐并且数字是右对齐的。看一下 printf 语句在 END 中是如何被应用的：

```
printf("Total: %d bytes (%d files)\n", sum, filenum)
```

在 BEGIN 规则中的标题也被适当地改变了。通过使用 printf 语句。filesum 产生如下的结果：

```
$ filesum g*
FILE BYTES
g 23
gawk 2237
gawk.mail 1171
gawk.test 74
gawkro 264
gfilesum 610
grades 64
grades.awk 231
grepscript 6
Total: 4680 bytes (9 files)
```

向脚本传递参数

在 awk 中，一个容易引起混乱的地方就是想脚本传递参数。参数将值赋给一个变量，这个变量可以在 awk 脚本中访问。这个变量可以在命令行上设置，放在脚本的后面，文件名前面。

```
awk ’ script’  var=value inputfile
```

每一项都必须作为单一的参数来解释。因此，在等号的两边不允许出现空格。也可以用这个方法传递多个参数。例如，如果想在命令行定义变量 high 和 low，可以用下面的代码调用 awk：

```
$ awk -f scriptfile high=100 low=60 datafile
```

在脚本中，这两个变量可以作为 awk 的任何变量来访问。如果要将这以脚本写入一个 shell 脚本的实现中，则可以以数值的形式传递 shell 的命令行参数（shell 按位置提供了命令行参数变量：$1 表示第一个参数，$2 表示第二个参数，依此类推）（注 9）。例如，参阅前面命令的 shell 脚本：

```
awk -f scriptfile "high=$1" "low=$2" datafile
```

如果这个 shell 脚本被命名为 awket，可以如下调用它：

运维技术书籍免费下载 https://www.linuxprobe.com

```
$ awket 100 60
```

“100”对应于$1，其值将赋给变量 high。

另外，环境变量或命令的输出结果也可以作为变量的值来传递。这里有两个例子：

```
awk ’ { ... }’  directory=$cwd file1 ...
awk ’ { ... }’  directory= ‘pwd ‘ file1 ...
```

“$cwd”返回变量 cwd 的值，即当前的工作路径（仅是 csh）。第二个例子使用反引号来执行 pwd 命令，并将它的结果赋予变量 directory（这事非常方便的）。

也可以使用命令行参数定义系统变量，像在下面的例子一样：

```
$ awk ’ { print NR, $0 }’  OFS=’ . ’  names
1. Tom 656-5789
2. Dale 653-2133
3. Mary 543-1122
4. Joe 543-221
```

输出字段分隔符被重定义为句点跟一个空格。

注 9：注意！不要将 shell 中的参数同 awk 中的字段变量混淆。

命令行参数的一个重要限制是它们在 BEGIN 过程中是不可用的。也就是说，直到首行输入完成以后它们才可用。为什么？这是一个容易混乱的部分。从命令行传递的参数就好像文件名一样被处理。赋值操作知道这个变量（如果它是一个文件名）被求值时才进行。

参阅下面的脚本，该脚本将变量 n 设置为一个命令行参数。

```
awk ’ BEGIN { print n }
{
if (n == 1) print "Reading the first file"
if (n == 2) print "Reading the second file"
}’  n=1 test n=2 test2
```

这里有 4 个命令行参数：“n=1”、“test”、“n=2”和“test2”。如果你现在还记着 BEGIN 过程即“在处理输入之前所要做的”，你将会理解为什么在 BEGIN 过程中的参数 n 返回值为空，因此 print 语句将打印一个空行。如果第一个参数是一个文件而不是一个变量赋值。该文件会知道 BEGIN 过程执行后才

运维技术书籍免费下载 https://www.linuxprobe.com

被打开。

第一个参数为变量 n 赋初值 1，第二个参数提供了文件名。因此，对于 test 中的每一行，条件“n==1”都为真。在读完 test 中的所有行之后，计算第三个参数，并将 n 赋值为 2。最后，第四个参数提供了第二个文件名。这时在主过程中的条件“n==2”为真。

以这种方法对参数求值的后果是不能用 BEGIN 过程测试或检验命令行提供的参数。只有当输入一行后它们才能够使用。要了解这种局限性，可以通过编写规则“NR==1”并使用它的过程来检验参数的赋值。另一个方法是在调用 awk 之前在 shell 脚本中测试命令行参数。

POSIX awk 提供了一个解决这个问题的方法，即在任何输入被读入前定义参数。用-v 选项（注 10）指定要在执行 BEGIN 过程之前得到变量赋值（也就是，在读入第一个输入行之前）。-v 选项必须在一个命令行脚本前说明。例如：下列命令使用-v 选项为多行记录设置记录分隔符。

```
$ awk -F"\n" -v RS="" ’ { print }’  phones.block
```

注 10：-v 选项并不是 nawk 原始版本（它仍然在 SunOS4.1x 系统和一些 System V Release3.x 系统上使用）的一部分，它是 1989 年在 Bell Lab 的 Brian Kernighan 增加的，GNU awk 的作者和 MKS awk 的作者一致同意在 BEGIN 块中可用的命令行上设置变量的方式，现在，它是 awk 的 POSIX 规范的一部分。

每个传递给程序的变量赋值都需要一个不同的-v 选项。

和 C 程序语言类似，awk 也提供系统变量 ARGC 和 ARGV。因为这需要了解数组，我们将在第八章“条件、循环和数组”中讨论这些特点。

信息的检索

awk 程序可以用于检索数据库中的信息，数据库实际上是各种类型的文本文件。文本文件的结构化越好，对其处理就越容易工作，尽管这个结果不过是由独立的单词组成的行。

下面这个首字母缩写词列表是一个简单的数据库。

```
$ cat acronyms
BASIC Beginner’s All-Purpose Symbolic Instruction Code
CICS Customer Information Control System
COBOL Common Business Oriented Language
DBMS Data Base Management System
GIGO Garbage In, Garbage Out
GIRL Generalized Information Retrieval Language
```


运维技术书籍免费下载 https://www.linuxprobe.com

制表符被作为字段分隔符。我们将看到一个程序，它将首字母缩写词作为输入并选择数据库中对应的行作为输出。（在下一章中，我们将看到另外两个使用首字母缩写词数据库的程序。一个程序是读取首字母缩写词列表并在另一个文件中找到这些首字母缩写词出现的位置。另一个程序是定位这些首字母缩写词在文本文件中的第一次出现的位置并插入相应的首字母缩写词的描述。）

我们编写的 shell 脚本命名为 acro。它从命令行中获取第一个参数（首字母缩写词的名字）并将它传递给 awk 脚本，acro 脚本如下：

```
$ cat acro
#! /bin/sh
# 将 shell’的 $1 赋给 awk 的 search 变量
awk ’ $1 == search’  search=$1 acronyms
```

在 shell 命令行中的第一个参数（$1）被赋给变量 search，这个变量作为参数传递给 awk 程序。传递给 awk 程序的参数在脚本之后说明。（这闲的有些混乱，因为在 awk 程序中$1 代表每个输入行的第一个字段，而在 shell 脚本中$1 代表命令行提供的第一个参数。）

下面的例子演示了如何用这个程序在列表中找到特殊的首字母缩写词。

```
$ acro CICS
CICS Customer Information Control System
```

注意，我们将参数作为字符串来检测（$1==search）。我们也可以将其写成一个正则表达式匹配（$1~search）。

查找小故障

如果它出现了一个可以用 awk 解决的问题，就会向我们发出一跳信息。下面是 Emmett Hogan: 写的原始信件：

```
     I have been trying to rewrite a sed/tr/fgrep script that we use
quite
      a bit here in Perl, but have thus far been unsuccessful...hence
this
      posting. Having never written anything in perl, and not wishing
to
       wait for the Nutshell Perl Book, I figured I’d tap the
       knowledge of this
      group.
      Basically, we have several files which have the format:
      item info line 1
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
      info line 2
      .
      .
      .
      info line n
    Where each info line refers to the item and is indented by
either
    spaces or tabs. Each item "block" is separated by a blank line.
    What I need to do, is to be able to type:
    info glitch filename
     Where info is the name of the perl script, glitch is what I
     want to
    find out about, and filename is the name of the file with the
    information in it. The catch is that I need it to print the
entire
    "block" if it finds glitch anywhere in the file, i.e.:
    machine Sun 3/75
          8 meg memory
          Prone to memory glitches
              more info
              more info
    would get printed if you looked for "glitch" along with any
other
    "blocks" which contained the word glitch.
    Currently we are using the following script:
    #!/bin/csh -f
    #
     sed ' /^ /\!s/^/@/'  $2 | tr ' \012@'  ' @\012'  | fgrep -i $1
     | tr ' @'  ' \012'
    Which is in a word....SLOW.
     I am sure Perl can do it faster, better, etc...but I cannot
     figure it out.
    Any, and all, help is greatly appreciated.
    Thanks in advance,
    Emmett
-----------------------------------------------------------------------
    Emmett Hogan Computer Science Lab, SRI International
```

这个问题可以用 awk 来解决。你也许在阅读更多的信息之前，试图自己解决这个问题。这个解决方法基于 awk 的多记录功能，需要将查找的字符串作为命令行参数进行传递。

下面是使用 awk 编写的脚本 info（注 11）。

运维技术书籍免费下载 https://www.linuxprobe.com

```
awk ' BEGIN { FS = "\n"; RS = "" }
$0 ~ search { print $0 }'  search=$1 $2
```

给出一个有多个条目的测试文件，测试 info 看它是否能找到单词“glitch”。

```
$ info glitch glitch.test
machine Sun 3/75
8 meg memory
Prone to memory glitches
more info
more info
```

下一章中我们将介绍条件、循环结构和数组。

注 11： 记住你需要能提供 POSIX 语义的 awk 来完成这项工作。它可以是 awk、nawk 乃至其他的东西！请检查你的本地系统文档。

运维技术书籍免费下载 https://www.linuxprobe.com

第八章 条件、循环和数组

本章内容

- 条件语句
- 循环
- 影响流控制的其他语句
- 数组
- 首字母缩写词处理器
- 作为系统变量的数组

这一章包含了一些基本的编程结构。它覆盖了 awk 程序设计语言中的所有控制结构。它还包括数组，即一种可以存储一系列值的变量。如果这是你第一次接触这些结构。你会认识到甚至 sed 也提供了条件和循环功能。在 swk 中，这些功能更普遍而且语法用起来更简单。实际上，awk 中的条件和循环结构的语法借鉴于 C 程序设计语言。因此，通过学习去 awk 和本章的结构，你也同样在学习 C 语言。

条件语句

条件语句用于在执行操作之前做一个测试。在前面的章节中，我们看到了模式匹配规则的一些示例。模式匹配规则本质上就是影响住输入循环的条件表达式。在这一部分，我们主要就在 action 中所使用的条件语句进行探讨。
条件语句以 if 开头，并计算放在圆括号中的表达式。语法是：

```
if ( expression )
   action1
[else
 action2]
```

如果条件表达式 expression 的值为真（非零或非空），就执行 action1。当存在 else 语句时，如果条件表达式的值为假（零或空），则执行 action2。一个条件表达式可能包含算术运算符、关系操作符、或布尔操作符，这些都在第七章编写 awk 脚本中讨论过。也许最简单的条件表达式是测试一个变量是否是一个非零值。

```
if ( x ) print x
```

如果 X 是零，print 语句将不执行。如果 X 是一个非零值，将打印 X 的值。也可以测试 X 是否等于另一个值：

```
if ( x == y ) print
```

注意，“==”是关系操作符而“==”是赋值操作符。我们还可以用模式匹配操

运维技术书籍免费下载 https://www.linuxprobe.com

作符“~”来测试 X 是否与一个模式匹配：

```
if ( x ~ /[yY](es)?/ ) print x
```

以下是几个补充的语法要点：

. 如果操作时由多个语句组成的，要用一对大括号将操作括起来。

```
if ( expression ) {
    statement1
    statement2
}
```

awk 对大括号和语句的位置没有特殊的要求（和 sed 不同）。左大括号放在条件表达式后面，可以与条件表达式位于一行也可以在下一行。第一条语句可以紧跟左大括号或从下一行开始，右大括号放在最好一条语句的后面，可以与最后一条语句位于同一行也可以在下一行。在大括号的前后允许有空格或制表符。
虽然没有要求语句缩进书写，但这样可以改善可读性。

. 右大括号和 else 后面的换行时可选的。

```
if ( expression ) action1
[else action2]
```

. 如果在 action1 后面加一个分号表示结束，action1 后面的换行也是可选的。

```
if ( expression ) action1; [else action2]
```

. 如果在同一行上用分号分隔多个语句，同样需要使用大括号。

在前面的章节中，我们曾看见过一段脚本用于计算学生平均成绩。我们可以用一个条件来判断某个学生是否及格。
假设平均分为 65 或更高为及格。我们可以编写如下的条件：

```
if ( avg >= 65 )
        grade = "Pass"
else
        grade = "Fail"
```

赋给 grade 的值取决于表达式“avg >=65”的计算结果是真还是假。

可以用多个条件语句来测试多个条件中的某个是否为真。例如，也许学生的成

运维技术书籍免费下载 https://www.linuxprobe.com

绩要用字母
分级表示，而不是用及格或不及格来表示。以下用一个条件结构根据学生的平均分来指定
一个字母成绩：

```
if (avg >= 90) grade = "A"
else if (avg >= 80) grade = "B"
else if (avg >= 70) grade = "C"
else if (avg >= 60) grade = "D"
else grade = "F"
```

应该注意到的一个重要事情是，这种连续条件只有当一个条件表达式计算结果为真时才停止求值，这时将跳过其他的条件。如果没有一个条件表达式的计算结果为真，将执行最后的 else 部分，运行默认操作，在这种情况下为 grade 赋值“F”。

条件操作符

Awk 中提供的条件操作符可以在 C 语言中找到，它的形式为：

expr ? *action1* : *action2*

前面的简单 if/else 条件可以用条件操作符改写成：

```
grade = (avg >= 65) ? "Pass" : "Fail"
```

这种形式更简洁而且适合于上面所示的简单的条件。? : 操作符可以嵌套使用。这样做将导致程序不易阅读。为了清晰，建议将条件用圆括号括起来，如上例所示。

循环

循环是一种用于重复执行一个或多个操作的结构。在 awk 中循环结构可以用 while、do 或 For 语句来指定。

While 循环

While 循环的语法是：

while (*condition*)

运维技术书籍免费下载 https://www.linuxprobe.com

```
        action
```

右圆括号后面的换行时可选的。条件表达式在循环的顶部进行计算，如果为真，就执行循环体 action 部分。如果表达式不为真，则不执行循环体。通常情况下，条件表达式的值为真并执行循环体，在循环体中改变某一值，直到最后条件表达式的值为假并推出循环。例如，如果希望执行一个循环体 4 次，可以编写下面的循环语句：

```
i = 1
while ( i <= 4 ) {
        print $i
        ++i
}
```

就像 if 语句一样，当循环体由多个语句构成时必须将它们放在大括号中。注意每个语句的作用。第一个语句是为 i 赋初值。表达式“i <= 4”用于比较 i 和 4 的值以决定是否执行循环体。这里的循环体由两个语句组成，一个语句简单地打印字段$i 的值，另一个语句递增 i 的值。i 是一个计数器变量，用来跟踪循环次数。如果我们没有递增计数器或条件一直不为假（例如 i>0），那么这个 action 将被不停地重复执行。

Do 循环

Do 循环是 while 循环的一个变型。Do 循环的语法为：

```
do
        action
while (condition)
```

Do 后面的换行时可选的。如果 action 后面使用分号，则换行也是可选的。这个结构的主要部分是 action 后面的条件表达式 condition。因此，循环体至少执行一次。参见下面的 Do 循环。

```
BEGIN {
        do {
                ++x
                print x
            } while ( x <= 4 )
    }
```

在这个例子中，X 的值在循环体中被自动递增操作符赋值，循环体首先执行一次，然后条件表达式被求值。在前面的 while 循环的例子中，i 的初值在循环的前面赋值，首先求条件表达式的值，然后执行一次循环体。请注意我们在运

运维技术书籍免费下载 https://www.linuxprobe.com

行上面的例子时 X 的值：

```
$ awk -f do.awk
1
2
3
4
5
```

在条件表达式被计算之前，X 被递增为 1（这是由于 awk 中的变量实际上都被初始化为 0）。这个循环体被执行了 5 次而不是 4 次。当 X 的值为 4 时条件表达式为真，循环体又被执行，X 递增到 5 并打印它的值，这时条件表达式的值为假并退出循环。将操作符“<=”改为“<”，即小于，循环体将运行 4 次。

请记住 do 循环和 while 循环之间的区别，do 循环至少执行一次循环体。在循环的底部都可以决定是否再次执行。

例如，我们来看一段循环访问记录中字段的程序，该程序将尽可能多地访问字段直到他们的总数达到 100 为止。我们使用 do 循环的原因是因为可以至少访问一个字段。我们将访问的字段值加到 total 上。当总数 total 超过 100 时就不再访问其他的字段。只有当第一个字段的值小于 100 时才能访问第二个，将它们的值再加到 total 上，当总数 total 超过 100 时退出循环。如果它仍然小于 100，将再次执行循环。

```
{
        total = i = 0
        do {
        ++i
        total += $i
        } while ( total <= 100 )
        print i, ":", total
}
```

程序的第一行初始化两个变量：total 和 i。循环递增 i 的值并用字段操作符引用一个特定的字段。循环每执行一次，将引用一个不同的字段。当第一次执行循环时，字段引用得到一个字段的值并将该值赋给变量 total。循环末尾的条件表达式用于判断 total 的值是否超过 100. 如果是，将结束循环。然后将变量 i 的值，即所访问的字段的个数，以及 total 打印出来（这个脚本假设每个记录的字段总和至少是 100，否则，将不得不检测 i 的值使得它不超过记录中的个数。我们将在下面一部分 for 循环中构造这样的测试）。

下面是包含一系列数据的测试文件：

```
$ cat test.do
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
45 25 60 20
10 105 50 40
33 5 9 67
108 3 5 4
```

对该测试文件中运行以上脚本得到下列结果：

```
$ awk -f do.awk test.do
3 : 130
2 : 115
4 : 114
1 : 108
```

对每个记录，仅访问字段的总和大于 100 的。

For 循环

For 语句是同 while 循环一样，能够得到相同结果的一个更紧凑的语法形式。尽管它看起来比较困难，但其语法使用更简单并能保证提供一个循环所需要的所有元素。For 循环的语法为：

```
for ( set_counter ; test_counter ; increment_counter )
      action
```

右圆括号后面的换行时可选的。For 循环由 3 个表达式组成：

set_counter
设置计数器变量的初值。

Test_counter
描述在循环开始时要测试的条件。

Increatment_counter
每次在循环的底部递增计数器，且恰好在重新测试 test_counter 之前。

下面来看使用 for 循环打印输入行的每一个字段。

```
for ( i = 1; i <= NF; i++ )
        print $i
```

和前面的例子中一样，i 是一个变量，使用字段操作符引用一个字段。系统变量 NF 是当前输入巨鹿中的字段的数量。这里用它来确定是否 i 已经到达了行的最后一个字段。变量 NF 的值是循环重复的最大次数。在循环体中执行 print 语

运维技术书籍免费下载 https://www.linuxprobe.com

句，打印本行的每个字段。使用这种结构的脚本可以打印一行中的每个单词，然后可以使用 sort|uniq -c 统计每个文件中单词的分布。

也可以编写一个循环按从最后一个字段到第一个字段的次序来打印。

```
for ( i = NF; i >= 1; i-- )
        print $i
```

循环每运行一次计数器递增 1。可以用这个循环反转字段的位置。

我们前面见到的 grades，awk 脚本是用来计算 5 哥成绩的平均值。可以将这段脚本改写为可以计算任意个数的平均值以使它更通用。也就是说，如果要用这个脚本来处理一个学年的成绩，那么需要处理的成绩的数量将增加。这里不用修改脚本来适应特别的字段的个数，而是编写一个通用的脚本来循环读取已有任意多的字段。这段程序的早期版本使用两个语句来计算 5 个成绩的平均值：

```
total = $2 + $3 + $4 + $5 + $6
avg = total / 5
```

我们将它修改为用一个 for 循环来合计记录的每个字段。

```
total = 0
for (i = 2; i <= NF; ++i)
        total += $i
avg = total / (NF - 1)
```

我们每次都初始化变量 total，因为我们不想将一个记录的值同下一个记录的值累计在一起。在 for 循环的开始将计数器 1 初始化为 2，因为第一个数字字段是 2。每循环一次将当前字段值累加到 total 中。当访问最后一个字段后（1 的值大于 NF 时），结束循环并计算平均值。例如，如果一个记录由 4 哥字段组成，循环第一次将$2 的值赋于 total。在循环结构的底部，i 递增 1，并与 NF 比较（NF 的值为 4），表达式的值为真同时将$3 的值累加到 Total 中。

注意，我们用总数除以字段总数的个数减 1，以将学生姓名字段排除。“NF-1”两边的圆括号是需要的，否则，由于除法操作符的优先级高于减法，所以将用 NF 除 total 然后再减去 1，而不是先计算 NF 减去 1。

求阶乘

一个数据的阶乘是由该数累乘小于这个数的数得到的结果。4 的阶乘是 4*3*2*1，或 24.5 的阶乘是 4 的阶乘的 5 倍或 5*24，或 120。求一个给定数的阶乘可以用下面的循环计算：

运维技术书籍免费下载 https://www.linuxprobe.com

```
    fact = number
    for (x = number - 1 ; x > 1; x--)
            fact *= x
```

这里的 number 是我们要计算阶乘 fact 的那个数。假设 number 的值为 5，执行第一次循环时 X 的值为 4，计算“5*4”并将结果赋值给 fact。下一次执行循环时，X 的值为 3，将其与 20 相乘并将结果赋值给 fact。循环一直重复到 X 的值为 1 时结束。

将上面的程序插入到一个独立的脚本中，提示用户输入一个数字并打印这个数字的阶乘。

```
    awk ' # factorial: return factorial of user-supplied number
    BEGIN {
              # prompt user; use printf, not print, to avoid the
newline
              printf("Enter number: ")
    }
    # check that user enters a number
    $1 ~ /^[0-9]+$/ {
              # assign value of $1 to number & fact
              number = $1
              if (number == 0)
                      fact = 1
              else
                      fact = number
              # loop to multiply fact*x until x = 1
              for (x = number - 1; x > 1; x--)
                      fact *= x
              printf("The factorial of %d is %g\n", number, fact)
              # exit -- saves user from typing CRTL-D.
              exit
     }
     # if not a number, prompt again.
     { printf(" \nInvalid entry. Enter a number: ")
     }'  -
```

这是一个有趣的例子，在住输入循环中提示输入并读取标准输入端的回答。

BEGIN

规则用于提示用户输入一个数据。因为我们已经规定输入内容不是来自一个文件而是来自标准输入，所以系统输出提示后，将暂停以等待用户输入数据。第一个规则是测试用户是否已经输入了数据，如果没有用第二个规则提示用户重新输入一个数据。我们设计了一个循环，用于从标准输入从夫读入数据，知道输入一个合法的数据为止。参见一部分中的 lookup 程序，该程序是另一个构造

运维技术书籍免费下载 https://www.linuxprobe.com

输入循环的例子。

下面的例子中显示了 factorial 程序是如何工作的：

```
$ factorial
Enter number: 5
The factorial of 5 is 120
```

注意，结果用“%g”转换说明来格式化 printf 语句。这使得用浮点数表示法可以表示很大的数。看下面的例子：

```
$ factorial
Enter number: 33
The factorial of 33 is 8.68332e+36
```

影响流控制的其他语句

你可以用 if、while、for 和 do 语句来改变程序的正常控制流，在本节中，我们将看到另外几个也能够影响控制流的语句。

在一个循环中有两个语句可以影响控制流，break 和 continue。Break 语句顾名思义就是推出循环，这样将不再继续执行循环。Continue 语句在到达循环底部之前终止当前的循环，并从循环的顶部开始一个新的循环。

考虑以下程序段将得到的结果：

```
for ( x = 1; x <= NF; ++x )
        if ( y == $x ) {
                print x, $x
                break
        }
Print
```

这个例子通过对当前记录的字段的循环访问，来打印字段的标号和它的值。但是（由于一些原因），我们希望避免打印第三个字段。使用条件语句检测计数器变量，当它等于 3 时，执行 continue 语句。Continue 语句将控制返回到循环的顶部，重复递增计数器变量。这避免了本次循环执行 print 语句。可以通过改写条件，即当 X 不等于 3 时执行 print 得到同样的结果。这里要说明的是，可以使用 continue 语句避免在特定的循环中达到循环的底部。

有两个语句能影响住输入循环，next 和 exit。Next 语句能够导致读入下一个输入行，并返回到脚本的底部（注 1）。这可以避免对当前输入行执行其他的操作过程。Next 语句的典型应用是可以连续从文件读取内容，忽略脚本的其他的操作直到文件被读完。系统变量 FILRNAME 提供了当前文件的名字。因此，可

运维技术书籍免费下载 https://www.linuxprobe.com

以如下编写模式：

```
FILENAME == "acronyms" {
        action
        next
}
{ print }
```

这使得对文件 acronyms 的每行都执行 action 指定的操作。当完成操作后，输入下一个新行。只有当从不同的文件输入内容时控制才执行 print 语句。

Exit 语句使主输入循环退出并将控制移到 END 规则，如果 END 存在的话。如果没有定义 END 规则，或在 END 中应用 exit 语句，则终止脚本的执行。我们在前面的程序 factorial 中，使用 exit 语句实现当输入一行后退出程序。

Exit 语句可以使用一个表达式作为参数，该表达式将作为 awk 的退出状态返回。如果没有提供表达式，那么将返回 0，如果为 exit 语句设置一个初值，然后在 END 中再次调用没有参数的 exit，则使用第一个值，例如：

```
awk ' {
        ...
        exit 5
}
END { exit }'
```

这里，awk 的退出状态是 5。

在后面的章节中你将看到应用这些控制流语句的一些例子。

注 1 ： 一些 awk 不允许你在用户定义的函数中用 next 语句，Caveat emptor。

数组

数组是可以用来存储一组数据的变量。通常这些数据之间具有某种关系。数组中的每一个元素通过它们在数组中的下标来访问。每个下标都用方括号括起来。下面的语句表示为数组中的一个元素赋值。

```
array[subscript] = value
```

在 awk 中不必指明数组的大小，只需要为数组指定标识符。向数组元素赋值是最容易完成的。例如，下面的例子中为数组 flavor 的一个元素指定了一个字符串”cherry”。

```
flavor[1] = "cherry"
```


运维技术书籍免费下载 https://www.linuxprobe.com

这个数组元素的下标是“1”。下面的语句将打印字符串“Cherry”。

```
print flavor[1]
```

可以用循环向数组中写入或取出元素。例如，如果数组 flavor 有 5 个元素，可以编写以下循环来打印每个元素：

```
flavor_count = 5
for (x = 1; x <= flavor_count; ++x)
        print flavor[x]
```

在 awk 中，数组的一个应用方式是存储每个记录的值，用记录的编号作为数组的下标。假设我们想跟踪计算出的每个学生的平均成绩，并且计算出一个班的平均成绩。每读入一个记录我们将做下面的工作：

```
student_avg[NR] = avg
```

系统变量 NR 作为数组的下标是因为对于每个记录它是递增的。当读入第一个记录时，avg 的值被放置到 student_avg[1]；对于第二个记录，它的值被放置到 student_avg[2]；以此类推。当我们将所有的记录读入后，将在数组 flavor 中生成一个平均值列表。在ＥＮＤ规则中，我们可以使用一个循环来讲所有这些平均成绩和全班的平均成绩比较，计算出大于或等于平均分的学生人数以及低于平均分的学生人数。

```
END {
        for ( x = 1; x <= NR; x++ )
              class_avg_total += student_avg[x]

        class_average = class_avg_total / NR

        for ( x = 1; x <= NR; x++ )
              if (student_avg[x] >= class_average)
                    ++above_average
              else
                    ++below_average
        print "Class Average: ", class_average
        print "At or Above Average: ", above_average
        print "Below Average: ", below_average
}
```

以上程序设置了两个循环来访问数组元素。第一个循环是将所有的平均分相加以便除以学生记录数。第二个循环检索每个学生的平均成绩，以便和全班的平均成绩相比较。如果大于或等于全班平均成绩，则递增变量ａｂｏｖｅ＿ａｖ

运维技术书籍免费下载 https://www.linuxprobe.com

ｅｒａｇｅ，否则递增变量ｂｅｌｏｗ＿ａｖｅｒａｇｅ。

关联数组

在ａｗｋ中，所有的数组都是关联数组。关联数组的独特之处在于它的下标可以是一个字符或一个数值。

在大多数的编程语言中，数组的下标都是唯一的数字。在这些语言中，数组时存储数据的一系列存储单元。数组的下表来自数在数组中存储的次序。没有必要跟踪数组的下标。例如，数组中的第一个元素的下标是“１”或是数组的第一个位置。

关联数组在数组的下标和元素之间建立了一种“关联”。对于数组中的每个元素都有两个相关的值：元素的下标和元素的值。这些元素不像传统的数组那样按一定的顺序存储。尽管在ａｗｋ中的数组的下标也可以是数据型的，但是这些下标的意义和其他编程语言中所表示的意义不同—它们不一定代表数的位置。然而，对于数值型下标，也能够顺序访问数值中的所有元素，就像我们在前面的例子中所完成的一样。你可以创建一个循环来递增计数器并按顺序访问数组元素。

有时，数值型和字符型下标之间的差别是很重要的。例如，如果用“０４”作为数组中一个元素的下标，就不能用“４”作为下标来定位这个元素。在后面的章节中，你将看到在示例程序ｄａｔａ＿ｍｏｎｔｈ中如何让处理这个问题。

关联数组是ａｗｋ中一个独特的特征，它的一个强大功能就是可以使用字符串作为一个数据的下标。例如，可以使用一个单词作为下标来查找他的定义。如果你知道这个单词，你就可以检索到它的定义。

例如，可以使用下面的赋值语句来将输入行的第一个字段作为第二个字段的下标：

```
array[$1] = $2
```

使用这种技术，我们可以将首字母缩写词列表写入到ａｃｒｏ数组中。

```
acro[$1] = $2
```

数组中的每个元素都是首字母缩写词的描述，而用于检索元素的下标就是首字母缩写词本身。

下面的表达式：

运维技术书籍免费下载 https://www.linuxprobe.com

```
acro["BASIC"]
```

结果为：

```
Beginner's All-Purpose Symbolic Instruction Code
```

有一个特殊的循环语法可以访问关联数组的所有元素。它是ｆｏｒ循环的一个版本。

```
for ( variable in array )
        do something with array[variable]
```

在这个例子中，打印语句当前的下标（例如“ＢＡＳＩＣ”），随后是这个下标定位的ａｃｒｏ数组的元素。

这个语法可以应用于使用数组型下标的数组。但是，访问数组中的条目的顺序是随机的（注２），在ａｗｋ实现中这种顺序经常发生变化，仔细编写你的程序使得它不依赖于ａｗｋ的任意版本。

重要的是需要记住ａｗｋ中的所有数字下标都是字符串类型。即使使用数字作为下标，ａｗｋ将自动将它们转换为字符串。当使用整数作为下标时也不必担心，因为它们也被转换成字符串，不管这些数据时ＯＦＭＴ（ａｗｋ的原始版本和新ａｗｋ的早期版本）还是ＣＯＮＶＦＭＴ（ＰＯＳＩＸ　ａｗｋ）。但是如果使用实数作为下标，那么向字符串的转换可能会有影响。例如：

```
$ gawk ' BEGIN { data[1.23] = "3.21"; CONVFMT = "%d"
> printf "<%s>\n", data[1.23] }'
<>
```

这里，在尖括号中没有打印任何东西，因为在第二个语句中１．２３被转换为１，而ｄａｔａ［“１”］的值为空串。

> 注意：当一个数值两次调用之中ＣＯＮＶＦＭＴ发生变法时，并不是每个版本的ａｗｋ都将数值转换为字符串的，用你所用的ａｗｋ测试上面的例子确保它能正确工作。

现在让我们回到计算学生成绩的例子上。我们想公布获得“Ａ”的学生是多少，获得“Ｂ”的学生是多少等等。一旦我们明确了等级，我们就能够递增相应的等级计数器。可以为每个字母等级设置不同的变量并测试哪个可以递增。

```
if ( grade == "A" )
        ++gradeA
else if (grade == "B" )
        ++gradeB
```


运维技术书籍免费下载 https://www.linuxprobe.com

.
.
.

然而，用数组实现这个任务更容易，我们可以定义一个名为ｃｌａｓｓ＿ｇｒａｄｅ的数组，并用字母等级（从 A 到 F）作为数组的下标。

```
++class_grade[grade]
```

因此，如果等级为“A”那么 calss_grade[“A”]的值递增 1。在程序的末尾，我们在 END 规则中用特殊的 for 循环来打印这些值：

```
for (letter_grade in class_grade)
      print letter_grade ":", class_grade[letter_grade] | "sort"
```

每次通过循环遍历 letter-grade 制定数组 class_grade 的一个下标。输出被传送到 sort 中，以确保按正确的顺序输出等级（将输出传送给程序将在第十章“‘底部抽屉’”中讨论），因为这是向 grades.awk 中最后添加代码，我们可以参见完整的清单。

```
# grades.awk -- average student grades and determine
# letter grade as well as class averages.
# $1 = student name; $2 - $NF = test scores.
# set output field separator to tab.
BEGIN { OFS = "\t" }
# action applied to all input lines
{
   # add up grades
         total = 0
         for (i = 2; i <= NF; ++i)
                 total += $i
   # calculate average
         avg = total / (NF - 1)
   # assign student’s average to element of array
         student_avg[NR] = avg
   # determine letter grade
         if (avg >= 90) grade = "A"
        else if (avg >= 80) grade = "B"
        else if (avg >= 70) grade = "C"
        else if (avg >= 60) grade = "D"
        else grade = "F"
   # increment counter for letter grade array
        ++class_grade[grade]
   # print student name, average and letter grade
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
        print $1, avg, grade
    }
     # print out class statistics
     END {
      # calculate class average
        for (x = 1; x <= NR; x++)
            class_avg_total += student_avg[x]
        class_average = class_avg_total / NR
      # determine how many above/below average
        for (x = 1; x <= NR; x++)
            if (student_avg[x] >= class_average)
                ++above_average
            else
                ++below_average
      # print results
        print ""
        print "Class Average: ", class_average
        print "At or Above Average: ", above_average
        print "Below Average: ", below_average
      # print number of students per letter grade
        for (letter_grade in class_grade)
            print letter_grade ":", class_grade[letter_grade] |
    "sort"
    }
```

以下是使用样本数据的运行结果：

```
$ cat grades.test
mona 70 77 85 83 70 89
john 85 92 78 94 88 91
andrea 89 90 85 94 90 95
jasper 84 88 80 92 84 82
dunce 64 80 60 60 61 62
ellis 90 98 89 96 96 92
$ awk -f grades.awk grades.test
Mona    79      C
john    88      B
andrea  90.5    A
jasper  85      B
dunce   64.5    D
ellis   93.5    A

Class Average:          83.4167
At or Above Average:    4
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
Below Average:        2
A:    2
B:    2
C:    1
D:    1
```

测试数组中的成员资格

关键词 in 也是一个操作符，用在条件表达式中来测试一个下标是否是数组的成员。表达式为：

```
item in array
```

如果 array[item]存在则返回 1.，否则返回 0。例如，如果字符串“BASIC”是数组 acro 的下标，以下的条件表达式将为真。

```
if ( "BASIC" in acro )
      print "Found BASIC"
```

如果“BASIC”是一个可用来访问 acro 元素的下标，则以上条件返回 True。但这个语法无法告诉你“BASIC”是否是 acro 中元素的值。这个表达式和编写一个循环一样可以检查下标是否存在，但是上面的表达式更容易编写，而且执行效率更高。

词汇检查脚本

这个程序从命名为 glossary 的文件中读入一系列词汇，并将它们放到一个数组中。程序提示用户输入一个词汇术语，如果找到了它，则打印相应的术语的定义。

以下是 lookup 程序代码：

```
awk ' # lookup -- reads local glossary file and prompts user for
query

#0
BEGIN { FS = "\t"; OFS = "\t"
        # prompt user
        printf("Enter a glossary term: ")
}

#1 read local file named glossary
FILENAME == "glossary" {
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
        # load each glossary entry into an array
        entry[$1] = $2
        next
}

#2 scan for command to exit program
$0 ~ /^(quit|[qQ]|exit|[Xx])$/ { exit }

#3 process any non-empty line
$0 != "" {
        if ( $0 in entry ) {
                # it is there, print definition
                print entry[$0]
        } else
                print $0 " not found"
#4 prompt user again for another term
{
printf("Enter another glossary term (q to quit): ")
}'  glossary -
```

以上模式为模式匹配规则进行了编号以便讨论。我们将按它们在程序代码留中出现的顺序来讨论它们。规则#0 是 BEGIN 规则，它在任何输入被读取前只执行一次。它将 FS 和 OFS 设置为制表符并提示用户输入一个词汇条目。用户的响应来自标准输入，但这是在 glossary 文件之后读入的。

规则#1 测试当前的文件名（FILENAME 的值）是否是“glossary”，并且只有当从这个文件读取时才执行。这个规则将词汇条目写入一个数组：

```
entry[term] = definition
```

这里$1 是术语，而$2 是定义。在#1 末尾的 next 语句用于在程序中跳过其他规则并读入一个新行。因此，在 glossary 文件的所有条目都被读入后，才接着应用其他的规则。

一旦读完 glossary 中的所有内容，awk 将从标准输入中读取，因为在命令行中指定了“-”号，标准输入即用户的响应。规则#3 测试输入行（$0）是否非空。只要用户键入了任何内容，该规则就能匹配。其中的操作使用 in 来测试输入行是否是数组的一个下标。如果是，只简单地打印出相应的值，否则告诉用户没有发现有效的条目。

在规则#3 之后将继续执行规则#4。这个规则只是简单地提示用户输入另外一个术语。不管在规则#3 中是否处理了一个有效的条目，规则#4 都将被执行。这里的提示告诉用户如何终止程序，测试完这个规则后，awk 将寻找下一个输入行。

运维技术书籍免费下载 https://www.linuxprobe.com

如果用户在下一个输入行键入“q”来选择退出，规则#2 将被匹配。模式将搜索用户在一行上可能输入的用于退出的各种单词或单个字母。“^”和“$“很重要，规定输入行除了包含这些外没有其他的字符，否则将会匹配在词汇中出现的”q“。注意这个规则在所有规则中的位置是很重要的。他必须在规则#3 和规则#4 之前出现，因为这些规则有可能和任何东匹配，包括单词”quit“和”exit“。

让我们来看看程序是如何工作的。对于这个例子，我们将文件 acronyms 复制一份并用它作为 glossary 文件。

```
$ cp acronyms glossary
$ lookup
Enter a glossary term: GIGO
Garbage in, garbage out
Enter another glossary term (q to quit): BASIC
Beginner’s All-Purpose Symbolic Instruction Code
Enter another glossary term (q to quit): q
```

正如我们所看到的，这段程序用于不停提示用户添加条目，直到用户键入了“q“。

注意，这段程序可以很容易地修改为读取文件系统中任何位置的词汇，包括用户根目录。Shell 脚本调用 awk 可以处理命令行选项使得用户可以规定词汇文件名。你也可以读取共享的词汇文件，然后读取本地的词汇文件，通过编写独立的规则对其进行处理。

用 split（）创建数组

内置函数 split（）能够将任何字符串分解到数组的元素中。这个函数对于从字段中提取“子字段“是很有用的。函数 split（）的语法为：

n = split(*string*, *array*, *separator*)

string 是要被分解到名字为 array 的元素中的输入字符串。数组的下标从 1 开始到 n，n 即为数组中元素的个数，元素根据指定的 separator 分隔符来分解。如果没有指定分隔符，那么将使用字段分隔符（FS）。Separator 可以是一个完整的正则表达式，而不仅仅是单个字符。数组的分解与字段的分解相同，参阅第七章中的“字段的引用和分离“部分。

例如，如果记录的第一个域由人的全名组成，则可以用 split（）函数抽取人的名字和姓氏。下面的语句将第一个字段分解为赋给数组 fullname 的元素：

运维技术书籍免费下载 https://www.linuxprobe.com

```
z = split($1, fullname, " ")
```

这里将空格规定为定界符。人的名字可以如下访问：

```
fullname[1]
```

人的姓氏可以如下访问：

```
fullname[z]
```

因为 Z 包含数组中元素的个数。不管人的全名中是否包含中间名，这种方法都适用。如果 Z 是由 split（）返回的值，则可以编写一个循环来读取数组的所有元素。

```
z = split($1, array, " ")
for (i = 1; i <= z; ++i)
        print i, array[i]
```

下部分包含应用 split（）函数的其他示例。

格式转换

这一部分举了两个例子，演示了将输出从一种格式转换为另一种格式的相似方法。

当运行第十二章“综合应用“中的索引程序时，我们需要一个快速的方法将罗马数字赋值给列编号。换句话说，在索引中的第四列需要被指定为下标中的”IV“。因为需求的列编号没有超过 10 的，我们编写了一段程序脚本，输入 1 到 10 之间的数字并转换为罗马数字。

这个 Shell 程序从命令行中获取第一个参数并将它作为输入返回给 awk 程序。

```
echo $1 |
awk ' # romanum -- convert number 1-10 to roman numeral

# define numerals as list of roman numerals 1-10
BEGIN {
            # create array named numerals from list of roman
numerals
            split("I,II,III,IV,V,VI,VII,VIII,IX,X", numerals, ",")
}
# look for number between 1 and 10
$1 > 0 && $1 <= 10 {
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
            # print specified element
            print numerals[$1]
            exit
    }

   { print "invalid number"
            exit
    }'
```

这个程序定义了 10 哥罗马数字的一个列表，并使用 split（）将他们写入到名为 numerals 的数组中。因为这一操作只需要执行一次，所以写在 BEGIN 操作模块中。第二个规则测试输入行的第一个域是否包含一个处于 1 到 10 之间的数字。如果是，这个数字被作为数组 numbers 的下标，并查找相应的元素。Exit 语句用于终止这个程序，只有当没有有效条目时才执行最后一个规则。

下例说明了它的执行情况：

```
$ romanum 4
IV
```

以下程序按照相同的思路将日期格式从“mm-dd-yy“或”mm/dd/yy“转换为”月，日，年“。

```
awk '
# date-month -- convert mm/dd/yy or mm-dd-yy to month day, year

# build list of months and put in array.
BEGIN {
          # the 3-step assignment is done for printing in book
          listmonths = "January,February,March,April,May,June,"
          listmonths = listmonths "July,August,September,"
          listmonths = listmonths "October,November,December"
          split(listmonths, month, ",")
 }

 # check that there is input
 $1 != "" {

 # split on "/" the first input field into elements of array
          sizeOfArray = split($1, date, "/")

# check that only one field is returned
          if (sizeOfArray == 1)
                   # try to split on "-"
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
                    sizeOfArray = split($1, date, "-")

# must be invalid
          if (sizeOfArray == 1)
                    exit

# add 0 to number of month to coerce numeric type
          date[1] += 0

# print month day, year
          print month[date[1]], (date[2] ", 19" date[3])
}'
```

这段脚本从标准输入中读取，BEGIN 操作建立了一个名为 month 的数组，他的元素是一年中月份的名字。第二个规则用于检验输入行为非空。其相关的操作中第一个语句查找“/“分隔符并分隔出输入的第一个字段。sizeOfarray 中包含着数组中元素的个数，如果 awk 不能分解这个字符串，它将创建只有一个元素的数组。因此，我们可以测试 sizeOfarray 的值来确定有几个元素，如果不包含多个元素，那么我们假设也许是使用”-“作为分隔符。如果不能分解出多个元素，则认为输入是非法的，并退出。如果我们成功地将输入分解到数组中，那么 date[1]中包含代表月份的数据。这个值能做数组 month 的下标，即将一个数组嵌套到另一个数组中。然而，在使用 date[1]之前，我们强制地将 date[1]加上 0 以改变其类型。因为 awk 将”11 正确解释为一个数字，而带前导 0 的数字会被看作为一个字符串。因此，没有强制类型转换的“06“不能被正确地识别。由 date[1]引用的元素被作为 month 的下标。

下例是运行情况：

```
$ echo "5/11/55" | date-month
May 11, 1955
```

删除数组元素

Awk 提供了一个语句用于从数组中删除一个元素。语法是：

delete *array*[*subscript*]

这里的方括号是必需的。这个语句将删除 array 中下标为 subscript 的元素，特别地，使用 in 测试 subscript 将返回为假。这与为数组元素赋一个空值是不同的，在这种情况下 in 将一直为真。参见下一章中使用 delete 语句的脚本示例。

运维技术书籍免费下载 https://www.linuxprobe.com

首字母缩写词处理器

现在我们来看一个程序，它的功能是浏览一个文件中的首字母缩写词。每个首字母缩写词都被替换为一个详细的描述，并将首字母缩写词放在圆括号中。如果有一行是“BASIC“，我们将用描述”Beginner’s All Purpose Symbolic Instruction Code“来代替，并将首字母缩写词写在后面的圆括号中（该程序本身可能没有什么用，但是在该程序中使用的技术是很普遍的，并有许多这样的应用）。

我们可以将这个程序设计成一个过滤器，用于打印所有的行，而不管这些行是否做了改变。我们将这个程序称为 awkro。

```
awk ’ # awkro - expand acronyms
# load acronyms file into array "acro"
FILENAME == "acronyms" {
        split($0, entry, "\t")
        acro[entry[1]] = entry[2]
        next
}

# process any input line containing caps
/[A-Z][A-Z]+/ {
        # see if any field is an acronym
        for (i = 1; i <= NF; i++)
                if ( $i in acro ) {
                        # if it matches, add description
                        $i = acro[$i] " (" $i ")"
                }
}

{
         # print all lines
         print $0
 }’  acronyms $*
```

我们首先来看它的执行情况。这里使用了一个样本输入文件 sample。

```
$ cat sample
The USGCRP is a comprehensive
research effort that includes applied
as well as basic research.
The NASA program Mission to Planet Earth
represents the principal space-based component
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
    of the USGCRP and includes new initiatives
    such as EOS and Earthprobes.
```

下面是文件 acronyms：

```
    $ cat acronyms
    USGCRP   U.S. Global Change Research Program
    NASA     National Aeronautic and Space Administration
    EOS      Earth Observing System
```

现在我们在样本文件上运行以上程序。

```
    $ awkro sample
    The U.S. Global Change Research Program (USGCRP) is a
comprehensive
    research effort that includes applied
    as well as basic research.
    The National Aeronautic and Space Administration (NASA) program
    Mission to Planet Earth
    represents the principal space-based component
    of the U.S. Global Change Research Program (USGCRP) and includes
new
    initiatives
    such as Earth Observing System (EOS) and Earthprobes.
```

我们将分两部分来看这个程序，第一部分是从文件 acronyms 中读取记录。

```
    # load acronyms file into array "acro"
    FILENAME == "acronyms" {
             split($0, entry, "\t")
             acro[entry[1]] = entry[2]
             next
    }
```

将这些记录的两个字段写入到一个数组中，其中将第一个字段作为下标而将第二个字段赋给数组的一个元素。换一句话说，首字母缩写词是他自己的描述的索引。

注意，我们没有改变成分隔符，而是使用 split（）函数来创建数组 entry。然后用这个数组创建 acro.

下面是程序的第二部分：

```
    # process any input line containing caps
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
/[A-Z][A-Z]+/ {
        # see if any field is an acronym
        for (i = 1; i <= NF; i++)
                if ( $i in acro ) {
                        acronym =$i
                        # if it matches, add description
                        $i = acro[$i] " (" $i ")"
                }
}
{
        # print all lines
        print $0
 }
```

只有在一行中包含多个连续的大写字母时，才使用以上程序段中的另一部分操作进行处理，这一部分操作用循环处理记录的每个字段。这部风操作的核心部分是使用条件语句测试当前的字段（$i）是否是数组（acro）的一个下标。如果是，则用数组中的元素代替相应的字段，而将原始值放于圆括号中（字段可以被赋予新值，和一般的变量一样）。注意将首字母缩写词的描述插入记录行中会太长。参看下一章中将讨论的 lengh（）函数，这个函数可以确定一个字符串的长度，如果字符串太长则可以分隔该字符串。

现在我们来改写这个程序，使得它只在第一次出现首字母缩写词时进行替换。当发现一个首字母缩写词后不再寻找该词。这是很容易做到的，我们呢只需要从这个数组中删除这个首字母缩写词就可以了。

```
if ( $i in acro ) {
        # if it matches, add description
        $i = acro[$i] " (" $i ")"
        # only expand the acronym once
        delete acro[acronym]
}
```

还可以用其他的好方法进行修改。在运行 awkro 程序时，我们很快会发现如果在首字母缩写词后边跟一个标点符号，那么匹配就会失败。我们最初的处理方法是在 awk 中根本不处理它。而是使用两个 sed 脚本，一个用于做预处理：

```
sed ' s/\([^.,;:!][^.,;:!]*\)\([.,;:!]\)/\1 @@@\2/g'
```

另一个用于做最后处理：

```
sed ' s/ @@@\([.,;:!]\)/\1/g'
```

在运行 awk 之前调用一个 sed 脚本，可以简单地在标点符号前插入一个空格，

运维技术书籍免费下载 https://www.linuxprobe.com

是标点符号能够被作为一个单独的字段来解释。另外插入一个由无用的符号（@@@）组成的字符串，使得我们能够容易地识别和恢复标点符号（在第一个 sed 命令中的复杂的表达式，可以确保我们匹配在一行中包含多个标点符号的情况）。

这种使用 UNIX 工具箱中另一个工具的解决方案，表明了不是每一件事都需要用 awk 过程来解决。而在 awk 过程中不需要做任何事情。Awk 更有价值是因为它是包含于 UNIX 坏境中。

但是在 POSIX awk 中可以用不同的解决方案，及使用正则表达式来匹配首字母缩写词，这种方法可以使用下一章中讨论的函数 match（）和 sub（）来处理。

多维数组

Awk 支持线性数组，在这种数组中的每个元素的下标是单个下标。如果你将线性数组看成是一行数据，那么两位数组将表示数据的行和列。你可以将第三行第二列的数据元素表示为“array[3,2]”。两维和三维数组是多维数组的例子。Awk 不支持多维数组，但它为下标提供了一个语法来模拟引用多维数组。例如，你可以如下编写表达式：

```
file_array[NR, i] = $i
```

这里的每个输入记录的字段使用记录编号和字段号做下标。因此，可以如下表示：

```
file_array[2, 4]
```

这将得到第二个记录的第四个字段的值。

这个语法不能创建多维数组。它将转换为一个字符串来唯一识别线性数组中的元素。多维数组下标的分量别解释为单独的字符串（例如“2”和“4”），并使用系统变量 SUBSEP 的值来连接。下表分量的分隔符默认地被定义为“\034”，这是一个不可打印的下标实际为“2\0344”（使用 SUBSEP 将“2”和“4”连接起来）。模拟多维数组的主要后果是数组越大，访问个别的元素就越慢。然而你可以使用你自己的应用程序来测试不同的 awk 实现的时间。

以下是一个命名为 bitmap.awk 的 awk 的脚本示例，用于展示如何向多维数组写入或读出元素。这个数组表示宽度和高度为 12 个字符的位图。

```
BEGIN { FS = "," # comma-separated fields
        # assign width and height of bitmap
        WIDTH = 12
        HEIGHT = 12
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
        # loop to load entire array with "O"
        for (i = 1; i <= WIDTH; ++i)
                for (j = 1; j <= HEIGHT; ++j)
                        bitmap[i, j] = "O"
}
# read input of the form x,y.
{
         # assign "X" to that element of array
         bitmap[$1, $2] = "X"
}
# at end output multidimensional array
END {
        for (i = 1; i <= WIDTH; ++i){
                for (j = 1; j <= HEIGHT; ++j)
                        printf("%s", bitmap[i, j] )
                # after each row, print newline
                printf("\n")
         }
}
```

在读取任何输入之前，将“O”写入到bitmap数组的所有元素中，这个数组有144个元素。向这个程序的输入是一系列的坐标，每行一个：

```
$ cat bitmap.test
1,1
2,2
3,3
4,4
5,5
6,6
7,7
8,8
9,9
10,10
11,11
12,12
1,12
2,11
3,10
4,9
5,8
6,7
7,6
8,5
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
9,4
10,3
11,2
12,1
```

对每个坐标，程序使用“X”来替换该位置的数组元素值“0”。在脚本的末尾使用与写入数组同样的循环来输出数组。下面的例子从文件 bitmap.test 中读取输入：

```
$ awk -f bitmap.awk bitmap.test
X0000000000X
0X00000000X0
00X000000X00
000X0000X000
0000X00X0000
00000XX00000
00000XX00000
0000X00X0000
000X0000X000
00X000000X00
0X00000000X0
X0000000000X
```

多维数组的语法也支持测试数组的成员资格。下标必需放置在圆括号中。

```
if ((i, j) in array)
```

这可以测试下标 i，j（实际上是 SUBSEP j）是否在指定的数组中存在。

对多维数组的循环操作和一位数组相同。

```
for (item in array)
```

你必须用 split（）函数来访问单独的下标分量。即：

```
split(item, subscr, SUBSEP)
```

以上 split（）函数使用下标 item 创建数组 subscr。

注意：我们在前面的例子用嵌套循环来输出两维位图数组，因为需要维护行和列。

运维技术书籍免费下载 https://www.linuxprobe.com

作为系统变量的数组

Awk 中提供的两个系统变量，它们是数组。

ARGV

这是一个命令行参数的数组，不包括脚本本身和任何调用 awk 指定的选项，这个数组中的元素的个数可以从 ARGC 中获得。数组中第一个元素的下标是 0（和 awk 中的其他数组不同，而和 C 一致），最后一个下标是 ARGC-1。

ENVIRON

一个环境变量数组，数组中每个元素是当前环境变量的值，而其下表是环境变量的名字。

命令行参数数组

你可以编写一个循环来访问 ARGV 数组中的所有元素。

```
# argv.awk - print command-line parameters
BEGIN { for (x = 0; x < ARGC; ++x)
                print ARGV[x]
        print ARGC
}
```

这个例子也打印了 ARGC 的值，即命令行参数的个数。下面的例子显示了上例对一个命令行样本是如何处理的：

```
$ awk -f argv.awk 1234 "John Wayne" Westerns n=44 -
awk
1234
John Wayne
Westerns
n=44
-
6
```

可以看出，这个数组中有 6 个元素。第一个元素使这个命令的名称，用于调用指定的脚本。在这种情况下，最后一个参数是文件名，“-”表示标准输入，注意“-fargv.awk”没有出现在参数列表中。

在通常情况下，ARGC 的值不小于 2。如果你不希望引用程序名或文件名，你可以将计数器初始化为 1 并测试 ARGC-1 以避免访问最后一个参数（假设这里只有一个文件名）。

运维技术书籍免费下载 https://www.linuxprobe.com

注意，如果你在 Shell 脚本中调用了 awk，命令行的参数将传递给 Shell 而不是传递给 awk。你必须将 Shell 脚本的命令行参数，传递给在 Shell 脚本中的 awk 程序。例如你可以用“$*（星号）”将 Shell 脚本中的所有命令行传递给 awk。参见下面的 Shell 脚本：

```
awk '
# argv.sh - print command-line parameters
BEGIN {
        for (x = 0; x < ARGC; ++x)
                print ARGV[x]
        print ARGC
}'  $*
```

这个 Shell 脚本和第一个调用 awk 的例子所做的工作一样。

一个实际的用法是在 BEGIN 过程中，用正则表达式来测试命令行参数。在下面的例子中测试除第一个参数外的其他所有参数为整数。

```
# number.awk - test command-line parameters
BEGIN {
          for (x = 1; x < ARGC; ++x)
                  if ( ARGV[x] !~ /^[0-9]+$/ ) {
                           print ARGV[x], "is not an integer."
                           exit 1
                   }
}
```

如果参数中包含非数字的字符，程序将给出提示并退出。

在测试了某个值以后，当然可以将它赋给一个变量。例如，我们可以编写一个 BEGIN 过程，在给出提示之前检查命令行参数。参见下面的 Shell 脚本，该脚本使用了前面章节中的电话号码和地址数据库：

```
awk ' # phone - find phone number for person
# supply name of person on command line or at prompt.
BEGIN { FS = ","
        # look for parameter
        if ( ARGC > 2 ){
                name = ARGV[1]
                delete ARGV[1]
        } else {
                 # loop until we get a name
                while (! name) {
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
                    printf("Enter a name? ")
                    getline name < "-"
                }
            }
    }
    $1 ~ name {
            print $1, $NF
    }'  $* phones.data
```

我们通过测试变量 ARGC 来看参数个数是否大于 2，通过指定“$*”可以讲 Shell 命令行的所有参数传递给 awk 的命令行。如果提供了参数，我们假设第二个参数 ARGV[1]使我们所需要的，那么将它赋给变量 name。然后将这个参数从数组中删除。这是非常重要的，如果命令行上提供的这个参数不是“var=value”形式；否则，它将在随后被解释为文件名。如果提供了另外的参数，则将会被解释为可选的电话号书库的文件名。如果参数的个数小于 2，那么我们将提示输入名字，getline 函数将在第十章讨论；使用这个语法，可以从标准输入中读入下一行。

下面是这个脚本的几个运行的示例：

```
$ phone John
John Robinson 696-0987
$ phone
Enter a name? Alice
Alice Gold (707) 724-0000
$ phone Alice /usr/central/phonebase
Alice Watson (617) 555-0000
Alice Gold (707) 724-0000
```

第一个例子在命令行上提供了名字，第二个提示用户，第三个给出两个命令行参数且将第二个作为文件名（这个脚本不允许在没有给出人名的情况下提供文件名。当然，你可以设计一个测试以允许这个语法）。

因为可以再 ARGV 数组中添加和删除，因此有许多潜在的有趣的处理。例如，你可以把文件名放置在数组的末尾，这样就可以像在命令行中指定的一样被打开。同样地，你也可以从数组中删除文件名，那么它将永远无法打开。注意，如果向 ARGV 中添加元素，也必须递增 ARGC；awk 使用 ARGC 的值来得到 ARGV 中有多少元素可以处理。因此，简单地递增 ARGC 将使 awk 不能检测到 ARGV 中的最后一个元素。

在特殊的情况下，如果 ARGV 的元素的值是一个空串（“”），awk 将跳过它并继续处理下一个元素。

运维技术书籍免费下载 https://www.linuxprobe.com

环境变量数组

数组 ENVIRON 被分别添加到 gawk 和 MKS awk 中。然后被添加到 System V Rlease 4 nawk 中，现在被包含在 POSIX 标准的 awk 中。它允许你访问环境变量。下面的程序循环访问了数组 ENVIRON 的所有元素并进行打印。

```
# environ.awk - print environment variable
BEGIN {
        for (env in ENVIRON)
                print env "=" ENVIRON[env]
}
```

数组的下标是变量的名字。该脚本产生与 env 命令相同的输出（在一些系统中是 printenv）。

```
$ awk -f environ.awk
DISPLAY=scribe:0.0
FRAME=Shell 3
LOGNAME=dale
MAIL=/usr/mail/dale
PATH=:/bin:/usr/bin:/usr/ucb:/work/bin:/mac/bin:.
TERM=mac2cs
HOME=/work/dale
SHELL=/bin/csh
TZ=PST8PDT
EDITOR=/usr/bin/vi
```

可以使用变量名作为数组的下标访问任意元素：

```
ENVIRON["LOGNAME"]
```

也可以修改数组 ENVIRON 中的任意元素：

```
ENVIRON["LOGNAME"] = "Tom"
```

但是这个修改并不改变用户的真实环境（例如，当执行完 awk 时，LOGNAME 的值没有变化），同样也没有改变程序的环境，这些程序是 awk 使用 getline（）或 system（）调用的。Getline（）和 system（）将在第十章讨论。

这一章包含了许多重要的程序结构。你在以后章节的例子中将会看到对这些结构的应用。如果你对变成还不熟悉，则应该花一些时间来运行本章的示例并做些完善，而且应该自己编写一些小程序。这是最基本的，就像如何使用动词一样，你将会觉得这些结构简单易学。

运维技术书籍免费下载 https://www.linuxprobe.com

第九章 函数

本章内容：
- 算术函数
- 字符串函数
- 自定义函数

函数是一个独立的计算过程，它接受一些参数作为输入并返回一些值，awk 有许多内置函数，可分为两组：算术函数和字符串函数。Awk 也支持用户自定义函数，允许你编写自己的函数来扩展内置函数。

算术函数

有 9 个内置函数可以被归类为算术函数，它们大多数接受数值型参数并返回数值型值。表 9-1 概括了这些算术函数。

表 9-1： awk 的内置算术函数

awk 函数	描述
cos(*x*)	返回 x 的余弦（x 为弧度）
exp(*x*)	返回 e 的 x 次幂
int(*x*)	返回 x 的整数部分的值
log(*x*)	返回 x 的自然对数（以 e 为底）
sin(*x*)	返回 x 的正弦（x 为弧度）
sqr t(*x*)	返回 x 的平方根
atan2(*y*,*x*)	返回 y/x 的反正切，其值在-π 到 π 之间
rand( )	返回为随机数 r，其中 0<=r<1
srand(*x*)	建立 rand（）的新的种子数。如果没有指定种子数，就用当天的时间。返回旧的种子值。

三角函数

三角函数 sin（）和 cos（）的运行方式相同，将用弧度表示的角度作为参数并计算这个角度的正弦或余弦（将度转换为弧度，用这个数乘以 π/180）。三角函数 atan2（）有两个参数并返回这两个参数上的反正切，表达式：

运维技术书籍免费下载 https://www.linuxprobe.com

```
atan2(0, -1)
```

结果是 π。

函数 exp（）是自然指数，它是以 *e* 为底的指数。表达式：

```
exp(1)
```

返回数据 2.71828，即自然对数的底 *e*。因此 exp（x）是 *e* 的 x 次幂。

函数 log（）是函数 exp（）的反函数，即 x 的自然数，函数 sqrt（）的参数只有一个并返回这个数（正数）的平方根。

整数函数

函数 int（）将数值型数据小数点右边的数字移去得到它的舍位。参见下面的两个语句：

```
print 100/3
print int(100/3)
```

这两个语句的输出结果显示在下面：

```
33.3333
33
```

int（）函数是简单的舍位，没有使用四舍五入法则（使用 printf 格式“%.0f”实现舍入）（注 1）。

随机数的生成

函数 rand（）生成一个在 0 和 1 之间的浮点型的伪随机数。函数 srand（）为随机数发生器设置一个种子数或起点数。如果调用 srand（）时没有参数，它将用当时的时间来生成一个种子数。若有参数 x，srand（）使用 x 作为种子数。

如果没有调用 srand（）、awk 在开始执行程序之前默认为以某个常量为参数调用 srand（）、使得你的程序在每次运行时都从同一个种子数开始。这可以用于重复测试相同的操作，但是如果希望程序在不同的时间运行具有不同的操作则不合适。参见下面的程序：

```
# rand.awk - 测试随机数的生成
BEGIN {
    print rand()
    print rand()
    srand( )
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
    print rand()
    print rand()
   }
```

这里首先打印了两次 rand（）函数的结果，然后调用 srand（）函数，接着再打印两次 rand（）函数的结果。我们来看程序的运行：

```
$ awk -f rand.awk
0.513871
0.175726
0.760277
0.263863
```

产生了 4 个随即数据。现在我们来看再次运行程序将发生什么：

```
$ awk -f rand.awk
0.513871
0.175726
0.787988
0.305033
```

注 1： printf 进行舍入的方式在附录二“awk 的快速参考”中讨论。

前面两个“随机”数和上一次运行程序产生的结果一样，而后面两个数是不同的。后面两个数不同是因为我们为 rand（）提供了新的种子数。

函数 srand（）的返回值是它所使用的种子数。这可被用来跟踪随机数的序列，如果需要可以反复使用这一序列执行程序。

Pick’em

为了说明如何使用 rand（），我们将参见一个脚本，该脚本用来实现一个抽彩票游戏的“快速挑选”。这个脚本被命名为 lotto，他/她从 1 到 y 之间的一系列数据中挑选 x 个数。在命令行需要提供两个参数：挑选多少个数字（默认为 6）和数据系列中的最大值（默认为 30）。使用 x 和 y 的默认值将缠成位于 1 到 30 之间的 6 个随机数。这些数据按从小到大排序并输出，以便阅读。在阅读脚本之前，我们先运行该脚本。

```
$ lotto
Pick 6 of 30
9 13 25 28 29 30
$ lotto 7 35
Pick 7 of 35
1 6 9 16 20 22 27
```


运维技术书籍免费下载 https://www.linuxprobe.com

第一个例子使用默认值打印了从 1 到 30 之间的 6 个随机数，第二个例子从 35 个数据中打印出了 7 个随机数。

完整的 lotto 脚本是很复杂的，因此在阅读整个脚本之前，我们先来看在一系列数据中生成一个随机数的一小部分程序：

```
awk -v TOPNUM=$1 '
# pick1 - 从y个数据中挑选一个随机数
# 主例程
BEGIN {
# 用当前的时间作种子数生成随机数
    srand( )
# 取得一个随机数
    select = 1 + int(rand() * TOPNUM)
# 打印挑选的结果
    print select
}'
```

shell 脚本需要从命令行中得到一个参数，并且这个参数通过使用-v 选项按“TOPNUM=$1”的形式传递给程序，所有的操作都在 BEGIN 过程中完成。因为程序中没有其他的语句，因此在完成 BEGIN 过程后退出 awk。

主例程首先调用 srand（）函数产生随机数发生器的种子数，然后调用 rand（）函数生成一个随机数。

```
select = 1 + int(rand() * TOPNUM)
```

将这个表达式分成几部分来看有助于了解该表达式。

语句	结果
print r = rand()	0.467315
print r * TOPNUM	14.0195
print int(r * TOPNUM)	14
print 1 + int(r * TOPNUM)	15

因为 rand（）函数返回的值在 0 和 1 之间，用 TOPNUM 来乘以它得到 0 到 TOPNUM 之间的一个数。然后将这个数的小数部分去掉并加 1，最后一步操作是必要的，因为 rand（）函数可能返回 0.在这个例子中，生成的随机数是 15.你可以用这个程序打印热河一个数，例如在 1 到 100 之间的一个数。

```
$ pick1 100
83
```


运维技术书籍免费下载 https://www.linuxprobe.com

使用 lotto 脚本必须执行多次“挑选一个”才能选出多个数。实际上，我们可以设计一个循环按需要的次数来执行 rand（）函数，其中困难的一个原因是我们必须考虑重复性。换句话说，有可能重复挑选一个数，因此我们必须跟踪已经被挑选的数。

下面是脚本 lotto 的代码：

```
awk -v NUM=$1 -v TOPNUM=$2 '
# lotto - 从y个数中挑选x个随机数
# 主例程
BEGIN {
# 测试命令行参数，NUM = $1,挑选多少个数
        # TOPNUM = $2,一系列数中的最后一个
if (NUM <= 0)
        NUM = 6
if (TOPNUM <= 0)
        TOPNUM = 30
# 打印 "Pick x of y"
        printf("Pick %d of %d\n", NUM, TOPNUM)
# 利用时间和日期作为种子数，只执行一次
        srand( )
# 循环到有NUM个选择时
        for (j = 1; j <= NUM; ++j) {
            # 用循环寻找一个还没有被发现的数
            do {
                select = 1 + int(rand() * TOPNUM)
            } while (select in pick)
            pick[select] = select
        }
# 循环访问数组并打印挑选的结果
        for (j in pick)
            printf("%s ", pick[j])
        printf("\n")
}'
```

和前面的例子不同，这个程序需要两个命令行参数，表示从 y 个数据中选择 x 个数。主例程首先检查是否提供了这些数据，如果没有提供则赋默认值。

这里只有一个数组 pick，用于保存已选择的随机数。每个数确保在希望的范围内，这是因为函数 rand（）的值（一个 0 到 1 之间的值），乘以 TOPNUM 然后进行取整。这个脚本的核心部分是一个循环运行 NUM 次，并将 NUM 个元素插入属组 pick 中。

为了得到一个不重复的随机数，我们使用了一个内循环来生成选择，并测试它

运维技术书籍免费下载 https://www.linuxprobe.com

们是否在数组 pick 内（使用 in 操作符比用循环在数组中比较下标要快）。当（select in pick）条件成立时，说明已经找到相应的元素，所以当前的选择是重复的，于是丢弃这个选择。如果 select in pick 为假，那么我们将把 select 赋给数组 pick 中的一个元素。这将使将来的 in 测试为真，使 do 循环继续。

最后，程序循环访问数组 pick 并打印它的元素。这个版本的 lotto 程序有一个问题。你是否能看出重新运行它将得到什么结果：

```
$ lotto 7 35
Pick 7 of 35
5 21 9 30 29 20 2
```

对了，数据没有排序。在我们讨论完用户定义函数时将显示排序例程的代码。尽管没必要把排序代码编写成一个函数，但这样做有许多意义。一个原因是你能处理更普遍的问题，并将这种解决方案用于其他的程序。在后面，我们将编写一个函数来排序数组中的元素。

注意，pick 数组没有为排序做好准备，因为它的索引和它的值是相同的、而不是有序的数据。我们将不得不建立一个单独的数组以便用排序函数来排序：

```
# 创建用于排序的下标为数值的数组
i = 1
for (j in pick)
        sortedpick[i++] = pick[j]
```

lotto 程序在 BEGIN 块中处理每件事。没有输入被处理。然而，你可以修订这个脚本使其能从文件中读入姓名列表，并为每个名字生成一个“快速挑选”。

字符串函数

内置字符串函数比算术函数更重要且更有趣。因为 awk 实质上是被设计成字符串处理语言，它的很多功能都起源于这些函数。表 9-2 列出了 awk 中的字符串函数。

表 9-2：awk 的内置字符串函数

awk 函数	描述
gsub(r,s,t)	在字符串 t 中用字符串 s 替换和正则表达式 r 匹配的所有字符串。返回替换的个数。如果没有给出 t，默认$0
index(s,t)	返回子串 t 在字符串 s 中的位置，如果没有指定 s，则返回 0

运维技术书籍免费下载 https://www.linuxprobe.com

length(s)	返回字符串 s 的长度，当没有给出 s 时，返回$0 的长度
match(s,r)	如果正则表达式 r 在 s 中出现，则返回出现的起始位置；如果在 s 中没有发现 r，则返回 0.设置 RSTART 和 RLENGTH 的值。
split(s,a,sep)	使用字段分隔符 sep 将字符串 s 分解到数组 a 的元素中，返回元素的个数。如果没有给出 sep，则使用 FS。数组分隔和字段分隔采用同样的方式
spr intf(“fmt ”,expr)	对 expr 使用 printf 格式说明
sub(r,s,t)	在字符串 t 中用 s 替换正则表达式 r 的首次匹配。如果成功则返回 1，否则返回 0，如果没有给出 t，默认为$0
substr(s,p,n)	返回字符串 s 中从位置 p 开始最大长度为 n 的子串。如果没有给出 n，返回从 p 开始剩余的字符串。
tolower(s)	将字符串 s 中的所有大写字符转换为小写，并返回新串。
toupper(s)	将字符串 s 中的小写字符转换为大写，并返回新串。

函数 split（）在前面的章节中讨论数组时已经介绍过。

sprint（）函数使用和 printf（）相同的格式说明，在第七章“编写 awk 脚本”中讨论过。它允许你对字符串应用格式说明。Sprintf（）不是将结果打印出来，而是返回一个字符串并可以赋给一个变量。它可以对输入记录或字段进行特殊处理，例如执行字符转换。例如，下面的例子使用 sprintf（）函数将一个数字转换为一个 ASCII 字符。

```
for (i = 97; i <= 122; ++i) {
    nextletter = sprintf("%c", i)
    ...
}
```

以上循环给出的数从 97 到 122，这些数产生从 a 到 z 的 ASCII 字符。

下面将讨论 3 个基本的内置字符串函数：index（）、substr（）和 length（）

运维技术书籍免费下载 https://www.linuxprobe.com

子串

index（）和 substr（）函数都用于处理子串。给定字符串 s，函数 index（s，t）返回 t 在 s 中出现的最左边的位置。字符串的开始位置是 1（这是和 C 语言不同的，在 C 语言中字符串的开始位置是 0）。参见下面的例子：

```
pos = index("Mississippi", "is")
```

pos 的值为 2.如果没有发现子串，函数 index（）返回 0。

给定字符串 s，substr（s，p）返回从位置 p 开始的字符。下面的例子生成一个没有区号的电话号码。

```
phone = substr("707-555-1111", 5)
```

还可以提供第三个参数来表示返回字符的个数。下一个例子只返回区号：

```
area_code = substr("707-555-1111", 1, 3)
```

这两个函数可以一期使用而且经常被一起使用，想下一个例子一样，这个例子将每个输入记录的第一个词的首字母改写为大写。

```
awk ’# caps －将第一个单词的首字母改为大写
# 初始化字符串
BEGIN { upper = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        lower = "abcdefghijklmnopqrstuvwxyz"
}
# 对于每个输入行
{
# 得到第一个单词的首字母
        FIRSTCHAR = substr($1, 1, 1)
# 获取FIRSTCHAR在小写字母数组中的位置，如果为0，忽略
        if (CHAR = index(lower, FIRSTCHAR))
                # 改变 $1，用位置来检索
                # 大写字母
                $1 = substr(upper, CHAR, 1) substr($1, 2)
# 打印记录
            print $0
}’
```

这段程序创建了两个变量：upper 和 lower，分别包含大写字母和小写字母。我们在 lower 中找到的每个字母都可以在 upper 中相同的位置找到。主过程的第一个语句提取了第一个字段的第一个字母。条件语句使用 index（）函数测试以确定那个字母是否能在 lower 中找到。如果 CHAR 不是 0，那么可以用 CHAR

运维技术书籍免费下载 https://www.linuxprobe.com

从 upper 中提取大写字母。这里调用了两个 substr（）函数：第一个检索第一个字段的大写字母而第二个得到第一个字段的剩余部分，即提取从第二个字母开始剩余的所有字母。两个 substr（）函数的返回值被连接在一期并赋给$1. 像这里这样，为一个字段赋值是一种新的手法，但是这有额外的好处，即记录可以被正常输出（如果对一个变量赋值，则必须输出这个变量并输出这个记录的其他字段）。Printf 语句打印出了变化了的记录。我们来看执行情况：

```
$ caps
root user
Root user
dale
Dale
Tom
Tom
```

稍后，我们将看到如何修改这个程序来将一个字符串中的所有小写字母转换为大写字母，或者反过来转换。

字符串长度

观察在前面章节中出现的 awkro 程序，我们会注意到程序有可能产生超过 80 个字符的行。毕竟描述都非常长，使用内置函数 length（）就可以知道一个字符串中有多少个字符。例如，要计算当前输入记录的长度，我们可以使用 length（$0）（正巧，如果函数 length（）被调用时没有给出参数，它将返回$0 的长度）。

函数 length（）经常用于计算当前输入记录的长度，以决定是否需要断行。

一种处理断行的方法就是使用 length（）函数得到每个字段的长度，这样可能效率更高，通过累计这些长度，当一个新的字段使得行的总长度超多某个特定的数据时，我们就可以指定一个换行。

在第十三章“脚本的汇总”中包含一个程序，当行的宽度超过 80 列时使用函数 length（）产生断行。

替换函数

awk 提供了两个替换函数：sub（）和 gsub（）。两者之间的区别是 gsub（）可以实现输入字符串中所有位置的替换，而 sub（）函数只实现第一个位置的替换。这使 gsub（）和 sed 中用 g（全局的）标志的替换命令相同。

这两个函数都至少需要两个参数。第一个参数是一个正则表达式（用斜杠包围着），用于和一个模式匹配；而第二个参数是一个字符串，用来替换模式匹配的字符串。正则表达式可以用一个变量来给出，在这种情况下将省略斜杠。第三个可选的参数指定的字符串是将被替换的目标。如果没有第三个参数，将当

运维技术书籍免费下载 https://www.linuxprobe.com

前的输入记录（$0）作为被替换的字符串。

替换函数直接改变指定的字符串。假设函数能正常工作，你或许希望当发生替换后函数返回替换后的新串。但替换函数实际上返回替换的数量。在 sub（）运行成功时总是返回 1，在不成功时两个函数都返回 0.因此，可以通过测试这个结果来确定是否执行了替换操作。

例如，下面的例子使用 gsub（）将所有出现的“UNIX”用“POSIX”替代。

```
if (gsub(/UNIX/, "POSIX"))
    print
```

条件语句测试 gsub（）返回的值，只有发生变化时当前输入行才被打印。

和 sed 一样，如果在替换字符串中出现了一个“&”字符，它将被与正则表达式匹配的字符串代替。用“\&”将输出一个字符“&”（记住，要在字符串中加入一个反斜杠“\”，则必须输入两个反斜杠）。注意，awk 和 sed 不一样，不能“记住”前面的正则表达式，因此我们不能用语法“\\”来引用最后的正则表达式。

下面的例子用于将“UNIX”的任意出现出 troff 字体更改转义序列来代替。

```
gsub(/UNIX/, "\\fB&\\fR")
```

如果输入是“the UNIX operating system”，输出将是“the \fBUNIX\Fr operating system”。

在第四章“编写 sed 脚本”中，我们给出了下面的 sed 脚本，命名为 do.outline。

```
sed -n '
s/ "//g
s/^\.Se /Chapter /p
s/^\.Ah /•A. /p
s/^\.Bh /••B. /p'  $*
```

现在用替换函数重写上面的程序：

```
awk '
{
gsub(/ "/, "")
if (sub(/^\.Se /, "Chapter ")) print
if (sub(/^\.Ah /, "\tA. ")) print
if (sub(/^\.Bh /, "\t\tB. ")) print
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
}'  $*
```

这两个脚本完全等价的，都仅打印出那些变化了的行。在这本书的第一版中，Dale 比较了这两个脚本的运行时间，和他预期的相同，awk 脚本慢一些。在第二版中，再次确定运行时间表明使用不同的实现工具其性能不同，实际上，所有用新的 awk 版本的测试都比 sed 快！这事好现象，因为在 awk 中可以使脚本做更多的事。例如，我们可以给标题编号来代替使用字母表中的字母。下面是经过修订的 awk 脚本：

```
awk ' # do.outline -- number headings in chapter.
{
gsub(/ "/, "")
}
/^\.Se/ {
    sub(/^\.Se /, "Chapter ")
    ch = $2
    ah = 0
    bh = 0
    print
    next
}
/^\.Ah/ {
    sub(/^\.Ah /, "\t " ch "." ++ah " ")
    bh = 0
    print
    next
}
/^\.Bh/ {
    sub(/^\.Bh /, "\t\t " ch "." ah "." ++bh " ")
    print
}'  $*
```

在这个版本中，我们为每个标题编写了它们自己的模式匹配规则。这虽然不是必要的，但好像更高效，因为一旦应用了一个规则，就不需要再考虑其他规则。注意 next 语句将跳过对已经识别过的行做进一步的检测。

章编号作为“.se”宏的第一个参数被读取，也就是行的第二个字段。编号方案通过在每次做替换时递增一个变量来完成。和章一级标题相关的操作将下一级的标题计数器初始化为零。和顶层标题 “.Ah”相关的操作将第二层标题的计数器初始化为 0.显然，你可以按需要设立任意多个层。注意我们是如何将字符串和变量串连接在一起作为函数 sub（）的一个参数的。

```
$ do.outline ch02
Chapter 2 Understanding Basic Operations
```


运维技术书籍免费下载 https://www.linuxprobe.com

2.1 Awk, by Sed and Grep, out of Ed
2.2 Command-line Syntax
2.2.1 Scripting
2.2.2 Sample Mailing List
2.3 Using Sed
2.3.1 Specifying Simple Instructions
2.3.2 Script Files
2.4 Using Awk
2.5 Using Sed and Awk Together

如果你希望可以选择使用字母或数字作为标题编号，则需要维护两个程序并创建一个 shell 实现，使用一些标志来决定调用哪个程序。

大小写转换

POSIX awk 提供了两个函数，用于完成字符串中字符的大小写转换。函数为 tolower（）和 toupper（）。每个函数需要一个字符串参数，并返回该字符串的一个备份，其中所有的字符串都发生了转换（分别为大写变小写和小写变大写）。它们的使用是简单明了的：

```
$ cat test
Hello, World!
Good-bye CRUEL world!
1, 2, 3, and away we GO!
$ awk ' { printf("<%s>, <%s>\n", tolower($0), toupper($0)) }'
test
<hello, world!>, <HELLO, WORLD!>
<good-bye cruel world!>, <GOOD-BYE CRUEL WORLD!>
<1, 2, 3, and away we go!>, <1, 2, 3, AND AWAY WE GO!>
```

注意： 非字母表中的字符没有发生转换。

match（）函数

match（）函数用于确定一个正则表达式是否和指定的字符串匹配。它需要两个参数，字符串和正则表达式（这个函数容易产生混淆，因为这个函数中正则表达式在第二个位置，而在替换函数中正则表达式在第一个位置）。

match（）函数返回与正则表达式匹配的子串的开始位置，你可能会认为它和函数 index（）有紧密的联系。在下面的例子中，正则表达式和字符串“the UNIX operating system”中的所有大写字母序列匹配。

```
match("the UNIX operating system", /[A-Z]+/)
```

函数的返回值为 5，即字符串中第一个大写字母“U”的位置。

运维技术书籍免费下载 https://www.linuxprobe.com

match（）函数也设置了两个系统变量：RSTART 和 RLENGTH.RSTART 中包含

这个函数的返回值，即匹配子串的开始位置。RLENGTH 中包含匹配的字符串的字符数（而不是子串的结束位置）。当模式不匹配时，RSTART 设置为 0，而 RLENGTH 设置为-1。在前面的例子中，RSTART 的值是 5，而 RLENGTH 的值是 4（将它们相加后可以得到匹配之后的第一个字符的位置）。

让我们来参见一个简单的例子，打印出与指定的正则表达式匹配的字符串，用于说明在第三章“了解正则表达式语法”中讨论的“匹配的范围”。下面的 shell 脚本包含两个命令行参数：正则表达式（这个正则表达式必须用引号括起来）以及要查找的文件名。

```
awk ’# match - 打印匹配行的字符串
# 对于模式的行
match($0, pattern) {
        # 提取匹配中的模式的字符串
        # 用字符串在$0中的开始位置和长度
        # 打印字符串
        print substr($0, RSTART, RLENGTH)
}’  pattern="$1" $2
```

第一个命令行参数被作为 pattern 的值传递。注意，$1 是用引号括起来的、这用于保护出现在正则表达式中的任何空格。match（）函数出现在条件表达式中，用于控制 awk 脚本中惟一的一个过程的执行。如果匹配的模式不存在，那么 match（）函数返回 0，如果存在，则返回非零值（RSTART），可以将这个返回值作为一个条件来使用。如果当前记录与模式匹配，那么字符串将从$0 中提取，在 substr（）函数中使用 RSTART 和 RLENGTH 的值来指定被提取的子串的开始位置和长度。同时打印该子串。这个过程只恶化$0 中第一次出现的子串匹配。

下面是一个验证运行，给出一个正则表达式来匹配“emp”和一个空格之前的所有字符.

```
$ match "emp[^ ]*" personnel.txt
employees
employee
employee.
employment,
employer
employment
employee’s
employee
```

match 脚本对于我们进一步理解正则表达式是一个很有用的工具。

运维技术书籍免费下载 https://www.linuxprobe.com

下一个脚本使用 match（）函数来定位任意的大写字母序列并将它转换为小写。将这个程序与本章中前面的程序 caps 进行比较：

```
awk ’# lower －将大写字母转换成小写字母
# 初始化字符串
BEGIN { upper = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        lower = "abcdefghijklmnopqrstuvwxyz"
}
# 对于每个输入行
{
# 查看是否有所有大写字母的匹配
   while (match($0, /[A-Z]+/))
       # 得到每个大写字母
       for (x = RSTART; x < RSTART+RLENGTH; ++x) {
              CAP = substr($0, x, 1)
              CHAR = index(upper, CAP)
              # 将小写字母替换大写字母
              gsub(CAP, substr(lower, CHAR, 1))
       }
# 打印记录
   print $0
}’ $*
```

在这个脚本中，match（）函数出现在条件表达式中，用来确定 while 循环是否执行。通过在循环中使用这个函数，可以使循环体的执行次数和当前输入记录中模式出现的次数一样多。

这里的正则表达式用于匹配$0 中任意的大写字母序列。如果得到一个匹配，那么 for 循环将对匹配的子串中的每一个字符进行搜索，和我们在本章中前面的 caps 程序中所做的处理类似。其中的区别是我们如何使用系统变量 RSTART 和 RLENGTH。RSTART 初始化计数变量 x，它用于函数 substr（）中一次从$0 中提取一个字符，提取位置从与模式匹配的第一个字符处开始。通过将 RSTART 和 RLENGTH 相加，我们可以得到与模式匹配之后的第一个字符的位置。这就是为什么循环用“<”而不用“<=”。在最后，我们使用 gsub（）将大写字母相应的小写字母来替换（注 2）。注意，我们用 gsub（）代替 sub（），这是因为如果在一行中出现多次相同的字母，gsub（）能够进行多个替换。

```
$ cat test
Every NOW and then, a WORD I type appears in CAPS.
$ lower test
every now and then, a word i type appears in caps.
```


运维技术书籍免费下载 https://www.linuxprobe.com

注 2： 你或许会问："为什么不用 tolower { } ？"问得好。一些早期的 nawk 版本，包括用在 SunOS 4.1.x 系统上的，都没有包含 tolower（）和 toupper（），因此，了解如何自己完成转换是很有用的。

注意，可以使用"/[A-Z][A-Z]+/"来修改正则表达式，通过匹配两个或多个大写字符来避免匹配单个大写字母。这还需要改变使用 gsub（）进行小写转换的方式，因为它与一行中的单个字符匹配。

在我们讨论 sed 中的替换命令时，介绍了如何存储和调用与模式匹配的字符串的一部分，使用\（和\）将要存储的模式包围起来，并在替换模式中使用/n 来调用存储的字符串。不幸的是，在 awk 的标准替换函数中没有等价的语法。但可用 match（）函数来解决许多这样的问题。

例如，如果用 match（）函数来匹配一个字符串，你可以确定出一个字符或子串，在另一个字符串中的起始或结束位置。给出 PSTART 和 RLENGTH 的值，你就可以用 substr（）函数来提取这些字符。在下面的例子中，我们用分号将两个冒号中的第二个替换掉。我们不能用 gsub（）来做这个替换，因为"/:/"匹配第一个冒号，而"/:[^:]*:"匹配整个字符串。我们可以用 match（）匹配字符串，并提取这个串的最后一个字符。

```
# 使用match函数和substr函数用分号取代第二个冒号
if (match($1, /:[^:]*:/)) {
        before = substr($1, 1, (RSTART + RLENGTH - 2))
        after = substr($1, (RSTART + RLENGTH))
        $1 = before ";" after
}
```

放置于条件语句中的 match（）函数用于检测是否有一个匹配。如果有，则使用 substr（）函数提取第二个冒号之前的子串以及它后面的子串。然后我们将 before、": "和 after 连接起来，并赋给$1。

我们将在第十二章"综合应用"中看到函数 match（）的应用。

自定义函数

使用用户自定义函数，awk 允许程序员新手采用 C 编程语言（注 3）不同的步骤来编写程序，这就是使用自含式函数。当正确地编写了一个函数时，也就定义了一个函数组建，这个组建可以被其他的程序重复使用。随着编写的程序的大小显著增长，以及编写的程序数目的增多，使用自定义函数的有点会变得更明显。

注 3： 或者用任何其他传统高级语言编程。

函数定义可以放置在脚本中模式操作规则可以出现的任何地方。通常情况下，我们将函数定义放在脚本顶部的模式操作规则之前。函数用下面的语法定义：

运维技术书籍免费下载 https://www.linuxprobe.com

```
function name (parameter-list) {
        statements
}
```

左大括号后面的换行和右大括号前面的换行都是可选的。你也可以在包含参数列表的右圆括号后和左大括号前进行换行。

parameter-list 是用逗号分隔的变量列表，当函数被调用时吗，它被作为参数传递到函数中。函数体由一个或多个语句组成。函数中通常包含一个 return 语句，用于将控制返回到脚本中调用该函数的位置；它通常带有一个表达式来返回一个值，如下所示：

```
return expression
```

下面的例子给出了 insert（）函数的定义：

```
function insert(STRING, POS, INS) {
before_tmp = substr(STRING, 1, POS)
after_tmp = substr(STRING, POS + 1)
return before_tmp INS after_tmp
}
```

这个函数有 3 个参数，在一个字符串 STRING 的 POS（注 4）位置之后插入另一个字符串 INS。在函数体中用函数 substr（）将 STRING 分为两部分。Return 语句返回一个字符串，这个字符串是将字符串 STRING 的第一部分、INS 和 STRING 的最后一部分连接起来而得到的。函数调用可以放置在表达式可以出现的地方。因此，下面的语句：

```
print insert($1, 4, "XX")
```

注 4： 我们习惯用大写表示参数。这主要是为了使解释容易些。在实际中，这不是一个好主意，因为这很容易与系统变量相冲突。

如果$1 的值为“Hello”，这个函数返回“HellXXo”。注意当调用用户自定义函数时，在函数名和左圆括号之间可以没有空格。但这对内置函数是不适合的。

理解局部变量和全局变量的概念是很重要的。一个局部变量是函数的内部变量，不能在这个函数外面访问。全局变量正相反，可以在脚本的任何地方被访问和修改。当一个函数修改了全局变量而这个变量在其他的地方也被使用时，这可能在有潜在的破坏性副作用。因此，在一个函数内避免使用全局变量是一个好主意。

运维技术书籍免费下载 https://www.linuxprobe.com

当我们调用函数 insert（），并将$1 设置为第一个参数时，这个变量的一个副本就被传递到函数中，在那里被作为一个局部变量 STRING 来处理，在函数定义的参数列表中的所有变量都是局部的，而且它们的值在这个函数之外是不能被访问。相同地，在函数调用中的参数不会被函数本身修改。当函数 insert（）返回时，$1 的值没有改变。

然而，在函数体中定义的变量默认为全局变量。对于前面给出的函数 insert（）定义，临时变量 before_tmp 和 after_tmp 在函数外是可见的。Awk 提供了它的开发者为“粗略的”方法来声明变量为函数的局部变量，也就是在参数列表中定义这些变量。

局部的临时变量在参数列表的末尾。最基本的是在参数列表中的参数按顺序接受函数调用传递来的值。任何补充的参数，和 awk 中的普通变量一样，被初始化为空串。习惯上，局部变量和“真实的”参数用几个空格隔开。例如，下面的例子显示了如何定义带有两个局部变量的 insert（）函数。

```
function insert(STRING, POS, INS, before_tmp, after_tmp) {
                    body
}
```

如果看上去有些混乱（注 5），那么了解下面的程序是如何工作的可能会有帮助：

```
function insert(STRING, POS, INS, before_tmp) {
    before_tmp = substr(STRING, 1, POS)
    after_tmp = substr(STRING, POS + 1)
    return before_tmp INS after_tmp
}
# 主例程
{
print "Function returns", insert($1, 4, "XX")
print "The value of $1 after is:", $1
print "The value of STRING is:", STRING
print "The value of before_tmp:", before_tmp
print "The value of after_tmp:", after_tmp
}
```

注 5： 有关文档称它为语法的补充。

注意，我们在参数列表中定义 before_tmp. 在主例程中，我们调用 insert（）函数并打印它的结果。然后打印不同的变量来观察它们的值（如果有值的话）是什么。现在我们运行上面的脚本并观察它的输出结果：

```
$ echo "Hello" | awk -f insert.awk -
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
Function returns HellXXo
The value of $1 after is: Hello
The value of STRING is:
The value of before_tmp:
The value of after_tmp: o
```

函数 insert（）返回预期的“HellXXo.”。$1 的值在函数被调用前后是一样的。变量 STRING 是函数的局部变量，当在主例程中调用时它没有值。对 before_tmp 来说也是一样，因为它的名字放在函数定义的参数列表中。变量 after_tmp 没有在参数列表中指定，它有一个值是字母“o”

正如这个例子所显示的，$1 将“按值”传递给函数。这就意味着，当调用函数时产生$1 的值的一个备份，并且函数对这个备份执行操作，而不是对原来的$1 执行操作。然而，数组是“按引用”传递的，也就是函数操作的不是数组的备份而是数组本身。因此，函数对数组的任何修改在函数外部都是可见的（在 C 语言中函数的“标量”变量和数组也是有区别的）。下一节给出了对数组进行操作的一个函数的例子。

编写一个排序函数

在本章的前面部分我们给出了 lotto 程序，用来从 y 个数据中挑选 x 个随机数。这个程序没有对被选择的数据排序。在本节中，我们将为一个数组的元素建立排序函数 sort。

我们定义的函数有两个参数，数组的名字和数组中元素的个数。这个函数可以用下面的方式调用：

```
sort(sortedpick, NUM)
```

相应的函数定义列出了应用在这个函数中的两个参数和 3 个局部变量。

```
# 用升序排序数字
function sort(ARRAY, ELEMENTS, temp, i, j) {
        for (i = 2; i <= ELEMENTS; ++i) {
        for (j = i; (j-1) in ARRAY && ARRAY[j-1] > ARRAY[j]; --j)
        {
            temp = ARRAY[j]
            ARRAY[j] = ARRAY[j-1]
            ARRAY[j-1] = temp
        }
    }
    return
}
```

函数体实现了一个插入排序，这个排序算法很简单。即循环访问数组的每个元

运维技术书籍免费下载 https://www.linuxprobe.com

素并于它前面的值想比较。如果第一个元素比第二个大，则将第一个元素与第二个元素交换（注 6）。为了真正交换数据，我们用一个临时变量来存储将要被覆盖的值的一个备份。循环将不停地交换相邻的数据知道所有的数据都按顺序排列。在函数的末尾，我们用 return 语句返回到程序的调用点（注 7）。函数不必将数组返回给主例程，因为数组已经被修改，它可以直接被访问。

下面是实际的排序结果：

```
$ lotto 7 35
Pick 7 of 35
6 7 17 19 24 29 35
```

实际上，我们在本章提供的许多脚本都可以改写为函数。例如，如果我们只有 1987 年 nawk 的原始版本，我们或许就需要编写 tolower（）和 toupper（）函数。

以通过的方式编写 sort（）函数的意思是可以很容易地重用它。为了说明这一点，我们将使用前面提到的排序函数，并用它来对学生的成绩进行排序。在下面的脚本中，我们将学生的成绩读到一个数组中，并调用函数 sort（）对学生的成绩按升序排序。

```
# grade.sort.awk  -  对学生成绩进行排序的脚本
# 输入：后面跟有一系列成绩的学生姓名
# 排序函数—按升序排序数字
function sort(ARRAY, ELEMENTS, temp, i, j) {
          for (i = 2; i <= ELEMENTS; ++i)
              for (j = i; ARRAY[j-1] > ARRAY[j]; --j) {
                  temp = ARRAY[j]
                  ARRAY[j] = ARRAY[j-1]
                  ARRAY[j-1] = temp
          }
          return
}
# 主例程
{
# 通过循环将第2到第NF字段的值赋给
# 命名为grades
for (i = 2; i <= NF; ++i)
       grades[i-1] = $i
# 调用排序函数来排序元素
sort(grades, NF-1)
# 打印学生姓名
printf("%s: ", $1)
# 输出循环
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
for (j = 1; j <= NF-1; ++j)
        printf("%d ", grades[j])
printf(" \n")
}
```

注意，这里的排序例程和前面的版本相同。本例在完成排序后简单地输出它们。

```
$ awk -f grade.sort.awk grades.test
mona: 70 70 77 83 85 89
john: 78 85 88 91 92 94
andrea: 85 89 90 90 94 95
jasper: 80 82 84 84 88 92
dunce: 60 60 61 62 64 80
ellis: 89 90 92 96 96 98
```

然而，如果我们希望在删除最低分数后计算学生的平均成绩，那么可以通过删除排序后的数组中的第一个元素来实现。还可以做另一个联系，你可以编写排序函数的另一个版本，它包含第三个参数用于指示按升序或降序排序。

注 6： 我们必须首先测试 j-1 是否在数组中。确保没有超过数组边界。

注 7： return 是可选的，这和函数“在尾部退出”有相同的效果，由于函数可能要返回一个值，用 return 语句是一个好的想法。

维护函数库

你或许希望把一个有用的函数保存在一个文件中，并保存在一个重要的目录下。Awk 允许使用多个-f 选项来指定多个程序文件（注 8）。例如，我们可以将前面编写的排序函数放置在与主程序 grade，awk 不同的文件中。下面的命令指定了两个程序文件：

```
$ awk -f grade.awk -f /usr/local/share/awk/sort.awk grades.test
```

这个命令假设 grade.awk 在当前工作目录中，并且排序函数定义在目录 /usr/local/share/awk 下的文件 sorf.awk

注意： 你不能将一段脚本放在命令行，并使用-f 选项来为一个脚本指定一个文件名。

注意，保存函数文档的有关信息有助于在重用它们时理解他们如何工作。

另一个排序的例子

我们产品的编辑 Lenny 回复了另外一个需求。

运维技术书籍免费下载 https://www.linuxprobe.com

Dale:
The last section of each Xlib manpage is called "Related Commands" (that is the argument of a .SH) and it’s followed by a list of commands
(often 10 or 20) that are now in random order. It’d be more useful and professional if they were alphabetized. Currently, commands
are separated by a comma after each one except the last, which has a
period.
The question is: could awk alphabetize these lists? We’re talking about a couple of hundred manpages. Again, don’t bother if this is a
bigger job than it seems to someone who doesn’t know what’s involved.
Best to you and yours,
Lenny

注 8： 在 SunOS 4.1.x 的 nawk 中不支持多个脚本文件的并用，在 1987 年 nawk 的原始版本中也没有这个功能。它是 1989 年增加的，现在是 POSIXawk 的一部分

为了明白他的意思，下面给出了 Xlib 帮助页的一个简单的版本：

```
.SH "Name"
XSubImage — create a subimage from part of an image.
.
.
.
.SH "Related Commands"
XDestroyImage, XPutImage, XGetImage,
XCreateImage, XGetSubImage, XAddPixel,
XPutPixel, XGetPixel, ImageByteOrder.
```

可以看出，与命令相关的名字在标题行后面且出现在许多行中。而且它们没有特定的顺序。

对于命令相关的列表排序是相当简单的，因为我们已经学会了排序。这个程序的结构很有趣，因为我们必须在匹配“Related Commands”标题之后读取几行。

通过观察输入，显然相关命令的列表位于文件的最后部分。除了这些行之外，其余的行都要按原样打印。关键是要匹配从“Related Commands”标题到文件的末尾的所有的行。我们的脚本包含四个规则，它们匹配的是：

运维技术书籍免费下载 https://www.linuxprobe.com

1. “Related Commands”标题
2. 该标题后面的行
3. 所有其他行
4. 在所有行被读入后（END）

大多数“操作”在 END 过程中执行，这也是排序和输出命令列表的地方。下面是这个脚本的代码：

```
# sorter.awk - 排列相关命令的列表
# 要求在单独的文件中的sort.awk作为一个函数
BEGIN { relcmds = 0 }
#1 匹配相关的命令，将标志relateds设为1
/\.SH "Related Commands"/ {
        print
        relcmds = 1
        next
}
#2 应用于跟在"Related Commands"后面的行
(relcmds == 1) {
        commandList = commandList $0
}
#3 按原样打印所有其他的行
(relcmds == 0) { print }
#4 现在排序并输出命令的列表
END {
# 删除前导空格和最后的句点
        gsub(/, */, ",", commandList)
        gsub(/\. *$/, "", commandList)
# 将各列分解到数组中
        sizeOfArray = split(commandList, comArray, ",")
# 排序
        sort(comArray, sizeOfArray)
# 输出元素
        for (i = 1; i < sizeOfArray; i++)
                printf("%s,\n", comArray[i])
        printf("%s.\n", comArray[i])
}
```

一旦匹配“Related Commands”标题，则打印那一行并设置一个标记变量 relcmds,该变量用于指示后续的输入行是要收集的行（注 9）。第二个过程实际上将每行插入变量 commandList 中。第三个过程处理所有其他行，知识简单地打印它们。

运维技术书籍免费下载 https://www.linuxprobe.com

当读完所有的输入行之后，执行 END 过程，这时形成了完整的命令列表。在将命令分解成不同的字段之前，应先将逗号后面的所有空格去掉。接着移走最后的句点及结尾空格。最后，使用韩式 splist（）创建一个数组 comArray。我们将这个数组作为参数传递给 sort（）函数，然后打印排好序的值。

该程序产生下面的输出：

```
$ awk -f sorter.awk test
.SH "Name"
XSubImage — create a subimage from part of an image.
.SH "Related Commands"
ImageByteOrder,
XAddPixel,
XCreateImage,
XDestroyImage,
XGetImage,
XGetPixel,
XGetSubImage,
XPutImage,
XPutPixel.
```

注 9： 下一章将讨论函数 getline，它提供一个简单的控制输入行的方法。

调用一个函数来完成排序与通过编写或复制代码，来完成相同的任务相比较，其优点是函数是经过测试的模块，而且有一个标准的接口。也就是说，你知道它能工作以及它是如何工作的。当你在 awk 中编写相同的排序代码时，使用了不同的变量名，你必须浏览它以证实它和其他版本以相同的方式工作。即使将代码复制到另一个程序中，也必须做一些调整来适应新的环境。对于一个函数，你所需要知道的只是它需要什么类型的参数以及它们的调用序列。使用函数可以通过减少所要解决问题的复杂程度来降低产生错误的机会。

由于以上脚本假设 sort（）函数存在与单独的文件中，所以必须用多个-f 选项来调用它：

```
$ awk -f sort.awk -f sorter.awk test
```

这里的函数 sort（）在文件 sort.awk 中定义。

运维技术书籍免费下载 https://www.linuxprobe.com

第十章

“底部抽屉”

本章内容：
- getline 函数
- close( ) 函数
- system( ) 函数
- 基于菜单的命令生成器
- 直接向文件和管道输出
- 生成柱状报告
- 调试
- 约束
- 使用#!语法调用 **awk**

本章要证明并不是一切东西都有其特定的位置。不管怎样安排它们，有些东西看起来就是不合适。本章是这些情况的一个汇总。将它标为“高级主题”是吸引人的，就好像要解释它结构（或者还不够充分），但有些读者可能感觉在阅读它之前应该先进一步学习。我们因此将它叫做“底部抽屉”。考虑抽屉的主体结构，像内衣、袜子和其他每天都要用的东西放在抽屉的上部，而不经常用的东西，例如毛线衫等放在底部抽屉。所有的这些都是同样可以获得的，但是你必须俯身去底部抽屉找东西。总之，需要更多的努力去得到底部抽屉的东西。

本章包含以下几个主题：

- getline 函数
- system( ) 函数
- 直接向文件和管道输出
- 调试 awk 脚本

getline 函数

getline 函数用于从输入中读取另一行。getline 函数不仅能读取正常的输入数据流，而且也能处理来自文件和管道的输入。

getline 函数类似于 awk 中的 next 语句。两者都是导致下一个输入行被读取，next 语句将控制传递回脚本的顶部。getline 函数得到下一行但没有改变脚本的控制。可能的返回值为：

1 如果能够读取一行。
0 如果到了文件末尾。
-1 如果遇到错误。

运维技术书籍免费下载 https://www.linuxprobe.com

注意： 尽管 getline 函数被称为一个函数并且返回了一个值，但它的语法类似于一个语句。不能写成 getline()，它的语法不允许有圆括号。

在前面的章节中，我们使用一个手册页源文件作为例子。通常情况下，-man 宏将文本参数放置在下一行上。尽管这个宏是用来查找行的模式，但它所要处理的
实际上是下一行。例如，从帮助页中提取命令的名字。下面的例子匹配了标题“na
me”，读取下一行，并打印它的第一个字段：

```
# getline.awk  -  -测试 getline 函数
/^\.SH "?Name"?/ {
        getline # 取得下一行
         print $1 # 打印新行的 $1 值
}
```

这个模式匹配任意包含“.SH”且其后跟有“Name”的行，它们可能被包围在引号中。一旦这一行被匹配，我们使用 getline 来读取下一个输入行。当读取新行后，getline 将它赋给 $0 并将它分解成字段。同时设置系统变量 NF、NR 和 FNR。因此新行变成当前行，这时可以引用$1 并检索第一个字段。注意前面的行不再看做是变量$0。然而，如果需要，可以用 getline 读取这行并将它赋给一个变量从而避免改变$0，正如我们马上要看到的一样。

以下例子显示了上面的脚本是如何工作的，它的功能是打印在“.SH Name”后面的行的第一个字段：

```
$ awk -f getline.awk test
XSubImage
```

我们在第九章“函数”的最后演示的 sorter.awk 程序，它可以使用 getline 来读取标题“Related Commands”后面的所有行。我们可以通过在 while 循环中测试 getline 的返回值来读取若干输入行。下面的过程代替了 sorter 程序中的前面两个过程：

```
# 匹配 "Related Commands" 并收集它们
/^\.SH "?Related Commands"?/ {
        print
        while (getline > 0)
                commandList = commandList $0
}
```

当 getline 成功地读取了一行时，表达式“getline>0”将为真。当到达文件末

运维技术书籍免费下载 https://www.linuxprobe.com

尾时，getline 返回 0 并退出循环。

从文件中读取

getline 函数除了能读取正常的输入流外，还可以从一个文件或管道中读取。例如，下面的语句从文件 data 中读取了一行：

```
getline < "data"
```

尽管文件名可以通过一个变量来提供，但它通常被指定为字符串常量，这字符串必须用引号括起来。符号“<”和 shell 程序中的输入重定向符号一样，不能被解释为“小于”符号。我们可以用 while 循环从文件中读取所有的行，测试到文件结束时循环退出。下面的例子打开文件 data 并打印它的所有行：

```
while ( (getline < "data") > 0 )
        print
```

（这里加上括弧以防止混乱；“<”表示重定向，而“>”是对返回值的一个比较。）输入也可以来自标准输入。你可以在提示用户输入信息后使用 getline：

```
BEGIN { printf "Enter your name: "
         getline < "-"
         print
}
```

这一示例打印提示“Enter your name:”（使用 printf 是因为我们不想在提示后面产生一个回车符）、然后调用 getline 获取用户的响应（注 1）。用户的响应被赋给$0，并用 print 语句输出这个值。

注 1： 至少曾经有一段时间，nawk 的 SGI 版本不支持 getline 用“_”从标准输入中读取，Caveat emptor。

将输入赋给一个变量

Getline 函数允许你将输入记录赋给一个变量，变量的名字为一个参数来提供。下面的语句从输入中读取下一行并赋给变量 input：

```
getline input
```

将输入赋给一个变量不会影响当前的输入行，也就是说，对$0 没有影响。新的

运维技术书籍免费下载 https://www.linuxprobe.com

输入行没有被分解成字段，因些对变量 NF 也无影响。但它递增了记录计数器 NF 和 FNR。

上例演示了如何提示用户。这个例子也可以按下面的方式编写，将用户的响应赋给变量 name。

```
BEGIN { printf "Enter your name: "
        getline name < "-"
        print name
}
```

注意将输入数据赋给变量的语法，因为通常错误地写成：

```
name = getline # 错误
```

这里将 getline 的返回值赋给变量 name。

从管道读取输入

可以执行一个命令并将输出结果用管道输送到 getline。例如，参见下面的表达式：

```
"who am i" | getline
```

这个表达式将 who am i 命令的输出结果赋给$0。

```
dale ttyC3 Jul 18 13:37
```

这个行被分解为字段并设置了系统变量 NF。同样地，你也可以将结果赋给一个变量：

```
"who am i" | getline me
```

通过将输出结果赋给一个变量可以避免设置$0 和 NF，但输入行没有被分解为字段。

下面的脚本将一个命令的输出结果用管道输送给 getline。它使用 who am i 命令的输出结果得到用户的名字，然后在/etc/passwd 中查找这个名字，打印出那个文件的第五个字段，即用户的命名：

```
awk ' # getname - 从/etc/passwd文件中打印用户命名
BEGIN { "who am i" | getline
        name = $1
        FS = ":"
}
name ~ $1 { print $5 }
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
’ /etc/passwd
```

命令在 BEGIN 过程中执行，并为我们提供用户的名字，该名字用于在 /etc/passwd 中查找用户的条目。就像前面所解释的，who am i 输出一行，getline 将赋给$0。输出的第一个字段$1 接着被赋给 name。

字段分隔符被规定为冒号“ ： ”，这允许我们访问/etc/passwd 文件中的条目的单独的字段。注意，FS 在执行 getline 这后设置，否则将影响到对信念的输出字段的分解。

最后，主过程被设计为用来测试第一个字段是否和 name 匹配。如果匹配，将记录的第五个字段打印出来。例如，当 Dale 运行这个程序时，它打印“Dale Dougherty.”。

当一个命令的输出结果被用管道输送给 getline 且包含多个行时，getline 一次读取一行。第一次调用 getline，它将读取输出的第一行。如果再次调用 getline，它将读取第二行。要读取输出的所有行，就必须创建一个循环来执行 getline，直到不再有输出为止。例如，下面的例子使用 while 循环来读取输出的每一行并将它赋给数组 who_out 的下一个元素：

```
while ("who" | getline)
        who_out[++i] = $0
```

每次调用 getline 函数时，读取输出的下一行。然而，其中的 who 命令只执行一次。
下一个例子是一个文档中查找“@data”并用当天的日期替换它：

```
# subdate.awk - 用当天的日期替换 @date
/@date/ {
        "date +’%a., %h %d, %Y’" | getline today
       gsub(/@date/, today)
}
{ print }
```

date 命令使用它的格式化选项（注 2）来提供日期，而 getline 将它赋给变量 today。gsub()函数用当天的日期替换每个出现的“@data”。

以上脚本用于在格式信件插入日期：

```
To: Peabody
From: Sherman
Date: @date

I am writing you on @date to
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
remind you about our special offer.
```

除了包含“@date”的行，“@date”被用当天的日期替换外，其余的所有行都按原样通过：

```
$ awk -f subdate.awk subdate.test
To: Peabody
From: Sherman
Date: Sun., May 05, 1996

I am writing you on Sun., May 05, 1996 to
remind you about our special offer.
```

close()函数

close()函数用于关闭打开的文件和管道。使用它有以下几个原因：

- 每次你只能打开一定数量的管道（参见后面的“约束”一节，它描述了系统与系统之间的约束的区别）。为了在一个程序中能够打开你所希望的数量的管道，你必须用 close()函数来关闭一个你用过的管道（通常是，当 getline 返回 0 或-1 时）。它用一个语句来实现，和用于创建管道的表达式相同。下面是一个例子：

```
close("who")
```

- 关闭一个管道使得你可以运行同一个命令两次。例如，你可以用两次 date 来定是一个命令。
- 为了得到一个输出管道来完成它的工作，使用 close()可能是必要的。例如：

```
{ some processing of $0 | "sort > tmpfile" }
END {
        close("sort > tmpfile")
        while ((getline < "tmpfile") > 0) {
                do more work
        }
}
```

注 2： date 的较老版本不支持格式化选项。尤其是 SunOS 4.1x 系统上的 date 版本，在那里必须使用/usr/5bin/date。参见本地文档。

- 为了保证同时打开的文件数不超过系统的限制，关闭打开的文件是必要的。

我们将在本章稍后的“处理多个文件”一节看到关于 close()函数的例子。

运维技术书籍免费下载 https://www.linuxprobe.com

System()函数

System()函数执行一个以表达式给出的命令（注 3）。然而，它的命令没有产生可供程序处理的输出。它返回被执行的命令的退出状态。脚本等待这个命令完成任务后才继续执行。下面的例子执行了 mkdir 命令：

```
BEGIN { if (system("mkdir dale") != 0)
                print "Command Failed" }
```

这里在一个 if 语句中调用 system()函数，来测试一个非零的退出状态。运行这个程序两次，结果是一次成功和一次失败：

```
$ awk -f system.awk
$ ls dale
$ awk -f system.awk
mkdir: dale: File exists
Command Failed
```

第一次运行产生新的目录，并且 system()返回退出状态 0（成功）。第二次执行这命令，目录已经存在，因此 mkdir 失败并产生一个出错信息。“Command Failed”信息是 awk 产生的。

Berkeley UNIX 命令集中为 troff 用户提供了一个很小但很有用的命令 soelim，这样命名的原因是因为它将从一个 troff 输入文件中“消除(eliminates)”“.so”行(.so 是一个请求，包含或“指定”命名文件的内容的来源)。如果你有一个老的 System V 系统，它没有 soelim，你可以用下面的 awk 脚本来创建它：

```
/^\.so/ { gsub(/"/, "", $2)
              system("cat " $2)
              next
             }
{ print }
```

注 3： system()函数模仿标准的 C 库函数中的同名函数。

这个脚本查找在行的开始处的.so，去掉所有的引号，然后用 system()来执行 cat 命令并输出文件的内容。这些输出和文件中剩余的行合并，如下面的例子所示，它被简单地打印到标准输出。

```
$ cat soelim.test
This is a test
.so test1
This is a test
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
.so test2
This is a test.
$ awk -f soelim.awk soelim.test
This is a test
first:second
one:two
This is a test
three:four
five:six
This is a test.
```

我们没有显式地测试命令的退出状态。因些，如果指定的文件不存在，出错信息将和输出融合在一起。

```
$ awk -f soelim.awk soelim.test
This is a test
first:second
one:two
This is a test
cat: cannot open test2
This is a test.
```

我们希望测试 system()函数的返回值并为用户生成出错信息。这个程序也很简单：它没有处理包含在文件中的“.so”。考虑该如何修改这个程序的版本以处理嵌入的“.so”请求。

下面的例子是一个函数，用来提示输入一个文件名。它使用 system()函数来执行 test 命令以验证文件存在并且可读：

```
# getFilename 函数 -- 提示用户输入文件名
# 验证文件名存在，并返回绝对路径名
function getFilename( file) {
    while (! file) {
            printf "Enter a filename: "
            getline < "-" # 获取用户的响应
            file = $0
            # 检查文件存在并可读
            # 如果文件不存在，测试返回1
            if (system("test -r " file)) {
                    print file " not found"
                    file = ""
            }
}
if (file !~ /^\//) {
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
        "pwd" | getline # 得到当前目录
        close("pwd")
        file = $0 "/" file
        }
        return file
    }
```

以上函数返回了由用户指定的文件的绝对路径名。它把提示信息和验证序列放在 while 循环中，以便于用户在前一个文件非法时可以重新输入。

如果文件存在且可读，test -r 命令将返回 0，否则返回 1。一旦确定了文件名是正确的，就测试文件名是否以“/”开始，斜杠表明用户提供了一个绝对路径名。如果测试失败，我们将使用 getline 函数去获得 pwd 命令的输出结果，然后将它与文件名拼接起来。（不可否认，该脚本没有对“./”和“../”项目进行了处理，尽管测试匹配它们很容易。）注意 getline 函数的两个用法：第一个是获得用户的响应，第二个是执行 pwd 命令。

基于菜单的命令生成器

在这一部分，我们着眼于 system()函数和 getline 函数的一个常见的应用，用来建立基于菜单的命令生成器。这个程序的目标是为非熟练技术用户提供一个简单的方法，来执行长的或复杂的 UNIX 命令。菜单用于将要执行的任务的描述提示给用户，允许用户使用数字在菜单中选择要执行的任务。

这个程序被设计为一种解释程序，它从文件中读取要出现在菜单中的描述和要执行的实际命令行。这种方式可以使用多重菜单命令文件，且它们可以在不改变程序的情况下很容易地被不熟悉 awk 的用户修改。

菜单命令文件的格式是在文件中将菜单标题作为第一行。后面的行包括两个字段：第一个是要执行的动作的描述，第二个是要执行的命令行。下面是一个例子：

```
$ cat uucp_commands
UUCP Status Menu
Look at files in PUBDIR:find /var/spool/uucppublic -print
Look at recent status in LOGFILE:tail /var/spool/uucp/LOGFILE
Look for lock files:ls /var/spool/uucp/*.LCK
```

创建基于菜单的命令生成器的第一步是读取菜单命令文件。我们首先读怪文件的第一行并将它赋给变量 title。余下的行包括两个字段并被读到两个数组中，一个用于生成菜单项，另一个用于提供要执行的命令。在一个 while 循环中使用 getline 从这个文件中一次读取一行。

```
BEGIN { FS = ":"
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
if ((getline < CMDFILE) > 0)
        title = $1
else
        exit 1
while ((getline < CMDFILE) > 0) {
        # 输入数组
        ++sizeOfArray
        # 菜单项的数组
        menu[sizeOfArray] = $1
        # 与菜单项相关的命令的数组
        command[sizeOfArray] = $2
        }
...
}
```

仔细观察使用 if 语句和 while 循环测试的表达式的语法。

```
(getline < CMDFILE) > 0
```

变量 CMDFILE 是菜单命令文件的名字，它被作为一个命令行参数来传递。这里的两个符号“<”和“>”具有完全不同的功能。符号“<”被 getline 解释为输入重定向操作符，然后测试 getline 的返回值看它是否大于（“>”）0。在表达式中使用括弧的目的是为了使表达式更清楚。换句话说，首先计算“getline<CMDFILE”，然后将它的结果与 0 比较。

这个过程被放在 BEGIN 模式中。但这儿有一个问题，因为我们要将菜单文件的名字作为命令行参数来传递，因此变量 CMDFILE 在 BEGIN 模式中将得不到定义并且在 BEGIN 模式中不可用。换句话说，下面的命令是不起作用的：

```
awk script CMDFILE="uucp_commands" -
```

因为直到第一个输入行被读取后变量 CMDFILE 才有定义。
幸运的是，awk 提供了-v 选项来处理这种情况。使用-v 选项可以使变量被立即设置并在 BEGIN 模式中可用。

```
awk -v CMDFILE="uucp_commands" script -
```

如果你的 awk 版本没有-v 选项，你可以将 CMDFILE 的值作为 shell 变量来传递。创建一个 shell 程序来执行 awk，并在该脚本中定义 CMDFILE。然后修改 invoke 脚本（如下所示）中读取 CMDFILE 的行：

```
while ((getline < ' "$CMDFILE"' ) > 0 ) {
```

一旦菜单命令文件被装载后，程序必须显示菜单并提示用户。这是通过函数来

运维技术书籍免费下载 https://www.linuxprobe.com

实现的，因为我们需要在两个地方调用它：从 BEGIN 模式中调用它以向用户提供初始提示，在处理了用户的响应后再调用它以便进行另一个选择。下面是 display_menu()函数：

```
function display_menu() {
        # 清屏 -- 如果清除命令不能用则注释掉该语句
        system("clear")
        # 打印标题、项目列表、退出项目和提示符
        print "\t" title
        for (i = 1; i <= sizeOfArray; ++i)
                printf "\t%d. %s\n", i, menu[i]
        printf "\t%d. Exit\n", i
        printf("Choose one: ")
}
```

我们做的第一件事就是使用 system()函数来调用一个命令以清除屏幕（在我的系统中，clear 可以完成这些工作；在其他的系统中可能用 cls 或其他命令。如果找不到这样的命令则将这一行注释掉）。然后我们打印标题和编号列表中的每项。最后一项始终是“Exit”。最后，我们提示用户选择。

这个程序将接收标准输入，这使得用户对提示的响应被作为输入的第一行。我们在这个程序中读取菜单命令文件，但它不是作为输入流的一部分。因此，这个程序的主过程就是对用户的选择做出响应并执行命令。下面是这部分程序：

```
# 处理用户对提示的响应
{
  #测试用户响应的值
  if ($1 > 0 && $1 <= sizeOfArray) {
       # 打印被执行的命令
       printf("Executing ... %s\n", command[$1])
       # 然后执行它
       system(command[$1])
       printf("<Press RETURN to continue>")
       # 在再次显示菜单前等待输入
       getline
  }
  else
       exit
  # 重新显示菜单
  display_menu( )
}
```

首先，我们检查用户的响应范围。如果响应超出了范围，我们只简单地退出程序。如果是有效的响应，那么我们从 command 数组中检索这个命令，显示它，

运维技术书籍免费下载 https://www.linuxprobe.com

并使用 system()函数执行它。用户将在屏幕上看到命令的执行结果且后面跟有信息“<Press RETURN to continue >”。显示这个信息的目的是在清除屏幕重并新显示菜单前等待用户操作的完成。Getline 函数使程序等待用户的响应，我们对这个响应不做任何事。函数 display_menu()在这个过程的末尾被调用，用来重新显示菜单并提示输入另一行。

下面是 invoke 程序的全部代码：

```
awk -v CMDFILE="uucp_commands" ' # 调用 -- 基于菜单的
                                 # 命令生成器
# CMDFILE 中的第一行是菜单的标题
# 后面的行包含: $1 - 描述;
# $2 要执行的命令
BEGIN { FS = ":"
# 处理 CMDFILE, 将项目读入菜单数组
  if ((getline < CMDFILE) > 0)
          title = $1
  else
          exit 1
  while ((getline < CMDFILE) > 0) {
        # 输入数组
        ++sizeOfArray
        # 菜单项的数组
        menu[sizeOfArray] = $1
        # 与菜单项相关的命令的数组
        command[sizeOfArray] = $2
  }
  # 调用函数显示菜单项和提示符
  display_menu( )
}
# 处理用户对提示的响应
{
  # 测试用户响应的值
  if ($1 > 0 && $1 <= sizeOfArray) {
        # 打印执行的命令
        printf("Executing ... %s\n", command[$1])
        # 然后执行它
        system(command[$1])
        printf("<Press RETURN to continue>")
        # 在再次显示菜单前等待输入
        getline
  }
  else
         exit
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
  # 重新显示菜单
  display_menu( )
}
function display_menu() {
        # 清屏 --如果不能清除，试试 "cls"
        system("clear")
        # 打印标题、项目列表、退出项目和提示符
        print "\t" title
        for (i = 1; i <= sizeOfArray; ++i)
                printf "\t%d. %s\n", i, menu[i]
        printf "\t%d. Exit\n", i
        printf("Choose one: ")
}'  -
```

当用户运行这个程序时，显示如下的输出：

```
UUCP Status Menu
        1. Look at files in PUBDIR
        2. Look at recent status in LOGFILE
        3. Look for lock files
        4. Exit
Choose one:
```

用户被提示输入菜单选项的编号。输入 1～3 之间以外的任何数字都能退出菜单。例如，如果用户输入“1”用于了解在 uucp 公共目录下的文件列表，那么在屏幕上将显示下面的结果：

```
Executing ...find /var/spool/uucppublic -print
/var/spool/uucppublic
/var/spool/uucppublic/dale
/var/spool/uucppublic/HyperBugs
<Press RETURN to continue>
```

当用户按 RETURN 键时，菜单将重新显示在屏幕上。用户可以选择“4”来退出这个程序。

这个程序实际上就是一个 shell，用以执行其他命令。任何命令序列（甚至是其他 awk 程序）可以通过修改菜单命令文件来被执行。换句话说，程序中最可能需要修改的部分被提取了出来，并通过维护一个独立的文件来完成修改。这使得菜单列表可以被非专业的用户很容易地改变和扩展。

直接向文件和管道输出

任何 print 和 printf 语句可以用输出重定向操作符“>”或“>>”直接将输出结果写入一个文件中。例如，下面的语句将当前记录写到文件 data.out 中：

运维技术书籍免费下载 https://www.linuxprobe.com

```
print > "data.out"
```

文件名可以是任何能产生合法的文件名的表达式。第一次使用重定向操作符将打开文件，随后使用重定向操作符将数据追加到文件中。“>”和“>>”之间的区别和 shell 中的重定向操作符之间的区别相同。右尖括号“>”在打开一个文件是截断它，而“>>”符号将保存文件中包含的任何东西并向文件中追加数据。

因为重定向操作符“>”和关系操作符是一样的，所以当你用表达式作为 print 命令的参数时可能会产生混淆。规定当“>”出现在任何打印语句的参数列表中时被看做是重定向操作符。要想使“>”出现在表达式的参数列表中时被看做是关系操作符，可以用圆括号将表达式或参数列表括起来。例如，下面的例子用圆括号将条件表达式括起来以确保关系表达式被正确求值：

```
print "a =", a, "b =", b, "max =", (a > b ? a : b) > "data.out"
```

条件表达式用于计算 a 是否大于 b，如果是。那么将 a 的值作为最大值打印；否则，打印 b 的值。

直接输出到一个管道

你也可以将输出直接写入一个管道，命令为：

```
print | command
```

该命令在第一次执行时打开一个管道，并将当前记录作为输入输送给命令 command。换名话说，这里的命令只执行了一次，但每执行一次 print 命令将提供另一个输入行。

下面的脚本将当前输入行的 troff 宏和请求去掉，并将该行作为 wc 的输入来计算文件中有多少单词：

```
{# words.awk - 去掉宏，然后得到单词数
sub(/^\.../,"")
print | "wc -w"
}
```

通过删除格式化代码，我们得到了更准确的单词数。

在大多数情况下，我们宁愿用 shell 脚本将 awk 命令的输出结果，用管道输送给另一个命令，而不是在 awk 脚本中来处理，例如，我们将前面的例子改写成 shell 脚本，调用 awk 并用管道输送它的结果给 wc：

运维技术书籍免费下载 https://www.linuxprobe.com

```
awk ' { # words -- 去掉宏
sub(/^\.../,"")
print
}' $* |
# 得到单词数
wc -w
```

这种方法更简单且更容易理解。然而，另一种方法的优势是可以完成相同的工作而且不用创建 shell 脚本。

注意，每次只能打开一定数量的管道。使用 close()函数关闭用过的管道。

处理多个文件

当读文件或写文件时文件被打开。每个操作系统对一个正在运行程序能够同时打开的文件的数量都有一定的限制，而且，每个 awk 实现对打开文件的数量都有内部限制，这个数字可能比系统限制要小（注 4）。为了避免打开过多的文件，awk 提供了 close()函数用于关闭打开的文件。关闭已经处理完的文件可使程序打开更多的文件。

直接向文件写入输出的一个常见的方法，是将一个大的文件分割成几个小的文件。尽管 UNIX 提供了有用的 split 和 csplit 命令，它们可以完成相同的工作，但它们不能为一个新文件指定一个有用的文件名。

注 4： **gawk 企图通过先关闭再重新打开文件，来打开比系统限制更多的文件。尽管 gawk 是“聪明的”，但关闭你使用过的文件更有效。**

类似地，也可以用 sed 来写入文件，但必须指定一个固定的文件名。在 awk 中，
你可以用变量来指定文件名并从文件的模式中挑选一个值。例如，如果$1 提供一个可以作为文件名的字符串，你可以编写一个脚本将每个记录输出到它对应的文件中：

```
print $0 > $1
```

也许需要测试文件名以确定它的长度或查找不能用在文件名中的字符。

如果不关闭文件，那么，这个程序最终将用完可打开的文件数而不得不中止。我们将要看到这个例子可以正常工作，因为它使用了 close()函数，它的运行不会超出打开文件数的限制。

下面的脚本用于将一个包含大量帮助页的大文件分割为小文件。每个帮助页以一个注册编号开始并以一个空行结束：

运维技术书籍免费下载 https://www.linuxprobe.com

```
.nr X 0
```

（尽管帮助页在大多数部分使用了-man 宏，帮助页的开始部分具有奇怪的编码，使得事情有点困难。）提供文件名的行如下所示：

```
.if \nX=0 .ds x} XDrawLine "" "Xlib - Drawing Primitives"
```

这个行的第五个字段，“XDrawLine”包含文件名，也许编写这个脚本的惟一困难是第一行不是给出文件名的行。因此，我们将这些行写入一个数组中直到找到文件名。一旦找到文件名，我们就输出这个数组，并从这一位置开始将每个输入行写入一个新的文件中。下面是 man.split 脚本：

```
# man.split -- 分隔包含x的帮助页的文件
BEGIN { file = 0; i = 0; filename = "" }

# 新帮助页的第一项是 ".nr X 0"
# 最后一行为空
/^\.nr X 0/,/^$/ {
        # 这个条件收集行直到得到一个文件名
        if (file == 0)
                line[++i] = $0
        else
                print $0 > filename
        # 匹配提供文件名的行
        if ($4 == "x}") {
                # 现在有一个文件名
                filename = $5
                file = 1
                # 将名字输出到屏幕
                print filename
                # 打印收集的所有行
                for (x = 1; x <= i; ++x){
                        print line[x] > filename
                }
                i = 0
          }
         # 关闭并清屏
        if ($0 ~ /^$/) {
                close(filename)
                filename = ""
                file = 0
                i = 0
         }
}
```


运维技术书籍免费下载 https://www.linuxprobe.com

可以看出，我们使用变量 file 作为标记来表示我们是否有一个合法的文件名，以及能否对这个文件进行写入。最初，file 为 0，并且当前的输入行被存储在一个数组中。变量 i 是一个用于数组下标的计数器。当我们遇到设置文件名的行时，将 file 设置为 1。新文件的名字打印在屏幕上，以便用户能从程序中得到脚本执行过程的一些反馈。然后我们循环访问这个数组，并将它输出到一个新文件中。不读取下一个输入行时，file 将被设置为 1 且 print 语句将把它输出到被命名的文件中。

生成柱状报告

本节描述了一个小型商务应用程序，产生美元数量的报告。这个应用程序没有介绍任何新的功能，它只是着重介绍了 awk 对数据的处理能力和报告功能（令人奇怪的是，一些人确实用 awk 编写一些小的商务应用程序）。

假设已有一个处理数据条目的脚本。该数据条目脚本有两项工作：第一项是输入客户的名字和通信地址，以便在后面创建通信地址列表；第二项是记录客户有关 7 个项目的订单，包括每项的订数及每项的单价。包含通信列表和客户订货的数据被写入不同的文件。

下面是客户订单文件中的两个客户记录的样本：

```
Charlotte Webb
P.O N61331 97 Y 045 Date: 03/14/97
#1 3 7.50
#2 3 7.50
#3 1 7.50
#4 1 7.50
#7 1 7.50
Martin S. Rossi
P.O NONE Date: 03/14/97
#1 2 7.50
#2 5 6.75
```

每个订单包含多行，并用一个空行将不同的订单分隔开。前面两行提供了客户的名字，购买订单编号和订货日期。后面的每个行用编号表示一个项目、订货数量和项目的单价。

我们来编写一个简单的程序，用价格乘以订货数量。这个脚本将忽略每个记录的前面两行。我们仅在指定为项目的地方读取相应的行，如下例所示：

```
awk ' /^#/ {
            amount = $2 * $3
            printf "%s %6.2f\n", $0, amount
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
                next
    }
    { print }'  $*
```

这里的主过程只影响匹配模式的行。它用第二字段乘以第三个字段，并将结果赋给变量 amount。Printf 格式转换符“%f”用于打印一个浮点型的数据，“6.2”指定最小输出域宽度为 6 且精度为 2。精度即十进制小数点右边的数字的个数，%f 的默认值是 6。Print 语句打印当前记录以及变量 amount 的值。如果在这个过程中打印一行，则将从标准输入中读取下一行。如果行和模式不匹配，将简单地被略过。我们来看 addem 是如何工作的：

```
$ addem orders
Charlotte Webb
P.O N61331 97 Y 045 Date: 03/14/97
#1 3 7.50 22.50
#2 3 7.50 22.50
#3 1 7.50 7.50
#4 1 7.50 7.50
#7 1 7.50 7.50

Martin S. Rossi
P.O NONE Date: 03/14/97
#1 2 7.50 15.00
#2 5 6.75 33.75
```

这个程序不必将客户的记录作为一个整体来访问，它只是简单地对独立的项目行进行处理。现在，我们来设计一个程序以读取多个记录，并累计订单信息，以便显示报告。这个报告将显示每个项目的总订单份数和总量。我们还希望产生所有订单的总份数和总数量。

该脚本将从设置字段和记录分隔符开始：

```
BEGIN { FS = "\n"; RS = "" }
```

每个记录的字段数是可变的，这依赖于订货的项目数。首先，我们检查输入记录到少有 3 个字段。然后用一个 for 循环从第三个字段开始读取所有的字段。

```
NF >= 3 {
for (i = 3; i <= NF; ++i) {
```

在数据库术语中，每个字段有一个值且每个值还能再分成子值。也就是说，如果在多行记录中一行为一个字段，则子值就是行中的单词。我们可以用 split() 函数将一个字段分割为子值。

运维技术书籍免费下载 https://www.linuxprobe.com

这个脚本的下面部分将每个字段分割为子值。$i 提供当前字段的值，它将被分割到数组 order 的元素中：

```
sv = split($i, order, " ")
if (sv == 3) {
procedure
} else
print "Incomplete Record"
} # 结束循环
```

函数返回的元素的个数被保存在变量 sv 中。可以测试是否有 3 个子值。如果没有，将执行 else 语句，在屏幕上打印出错信息。

接着我们将数组的每个独立的元素指定给一个特定的变量。这主要是为了更容易地记住每个出现所表示的意思：

```
title = order[1]
copies = order[2]
price = order[3]
```

然后我们对这些值进行一些算术运算：

```
amount = copies * price
total_vol += copies
total_amt += amount
vol[title] += copies
amt[title] += amount
```

我们对这些值进行累计，直到读完最后一条记录。END 过程用于打印报告。
以下是整个程序：

```
$ cat addemup
#! /bin/sh
# addemup -- 合计客户订单
awk ' BEGIN { FS = "\n"; RS = "" }
NF >= 3 {
        for (i = 3; i <= NF; ++i) {
                sv = split($i, order, " ")
                if (sv == 3) {
                        title = order[1]
                        copies = order[2]
                        price = order[3]
                        amount = copies * price
                        total_vol += copies
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
                                total_amt += amount
                                vol[title] += copies
                                amt[title] += amount
                        } else
                                print "Incomplete Record"
                }
    }
    END {
          printf "%5s\t%10s\t%6s\n\n", "TITLE", "COPIES SOLD",
"TOTAL"
          for (title in vol)
            printf "%5s\t%10d\t$%7.2f\n", title, vol[title],
amt[title]
          printf "%s\n", "-------------"
          printf "\t%s%4d\t$%7.2f\n", "Total ", total_vol, total_amt
    }'  $*
```

我们定义了两个下标相同的数组。我们只需要用一个 for 循环来读取这两个数组。

订单报告生成器 addemup 执行结果如下：

```
$ addemup orders
TITLE COPIES SOLD TOTAL
#1          5  $  37.50
#2          8  $  56.25
#3          1  $   7.50
#4         1  $  7.50
#7         1  $  7.50
-------------
Total     16  $  116.25
```

调试

调试比编程的任何方面都更容易令人灰心，也更必不可少。在本节中，我们将了解 awk 脚本的调试方法，并给出了当 awk 程序不能如期工作时，应如何修改 awk 程序的建议。

现代的 awk 版本提供了报告语法错误的良好功能。但是，就是具有一个好的错误检测功能，将问题孤立出来是很困难的。寻找问题根源的技术是很少的，却是必须的。不幸的是，大多数 awk 实现没有提供调试工具或扩展。

一个程序有两类问题。第一类是程序逻辑上真正的错误（bug）。也就是说，这种程序运行完成后并没有报告任何出错信息，但产生的结果却不是所希望的。

运维技术书籍免费下载 https://www.linuxprobe.com

例如，或许不生成任何输出。这种错误可能是没有用 print 语句打印计算结果引起的。程序错误是思路上的错误。

第二类错误是程序不能运行或不能完全运行。这可能是语法错误引起的，并导致 awk 向你警告所不能解释的代码。许多语法错误是由于输入错误或漏掉括号的错误。语法错误通常会产生出错信息以引导你找到问题所在。有时，程序引起 awk 失败（或“core dump”）而没有产生任何有用的出错信息（注 5）。这也可能是语法错误引起的，但有些问题可能与机器有关。我们有几个大脚本在这一台机器上运行时出现 core dump，而在另一台上运行却没有问题。例如，可能你的运行违反了 awk 对特殊实现的约束（参见本章后面“约束”一节）。

你应该清楚你想找到程序的那种类型的 bug：逻辑错误或语法错误。

制作副本

在开始调试程序前，先做一个它的副本。这是很重要的。要调试一个 awk 脚本，就不得不修改它。这些修改可能会指出你的错误，但许多修改都是没有效果的或可能产生新的错误。能将你的修改恢复是很好的。但是，恢复所做的每一个修改有些繁锁，因此我喜欢一直修改我的程序直到找到问题。当知道问题是什么时，再回到原始的程序版本并做修改。实际上是用一个副本恢复你所做的所有无意义的修改。

注 5： 这说明 awk 的实现是粗略的，“core dump”在 awk 的现代版本中很少发生。

将创建一个程序的步骤看成几个阶段也是有帮助的。将设置每个核心工作作为一个单一的阶段。一旦你完成了这些功能并且测试了它们后，在进入下一个阶段设计新的功能之前为程序做一个备份。用这个方法，当你增加新的代码产生问题时就可以恢复到前面的状态。

我们建议你使用这个方式进行，甚至可以使用源代码管理系统。例如 SCCS（源码控制系统），CRS（版本控制系统），或 CVS（并行版本控制系统，它和 RCS 是兼容的）。后面两个从 GNU FTP 镜像站点免费获得。

前像和后像

在 awk 调试中的困难是你并不是总知道在程序执行过程中发生了什么。你可以检查输入和输出，但没有办法中止程序执行并检查它的状态。因此，要想知道是程序的哪一部分引起的问题是很困难的。

运维技术书籍免费下载 https://www.linuxprobe.com

通常的问题是确定程序在什么时间或什么地方对变量进行赋值。第一个方法是用
Print 语句在程序中不同地方打印变量的值。例如，使用一个变量作为标志来确
定产生了某种特定的条件是很通用的。在程序的开始部分，标志被设置为 0。在
程序的一个或更多个点，这个标志被设置为 1。问题是找到变化到底在那里发生。
如果你想在程序的特定部分检查这个标志，在赋值的前后应用 print 语句。例如：

```
print flag, "before"
if (! $1) {
        .
        .
        .
        flag = 1
}
print flag, "after"
```

如果不能确定一个替换命令或函数产生的结果，在调用函数的前后打印相应的字
符串：

```
print $2
sub(/ *\(/, "(", $2)
print $2
```

在替换命令之前打印值为的是确信这个命令发现了你希望出现的值。前面的命令
可能已经改变了那个变量。问题将可能是输入记录的格式不是你所想像的。仔细
检查输入是调试的一个重要步骤。尤其是，可以使用 print 语句证明输入的各个
域与所期望的一样。当你发现是输入引起的问题时，可以修正输入或编写新的代
码来适应它。

查找问题的出处

一个脚本越是模块化（它可以分割成的单独的部分就越多），测试和调试程序就
越容易。编写函数的优点是可以将函数内部所做的处理隔离开，并可以在不影响

运维技术书籍免费下载 https://www.linuxprobe.com

程序其他部分的情况下测试它。你可以忽略一个总体操作并观察将发生什么。

如果一个程序有几个分支结构，你可能会发现输入行是通过一个分支传递的。测试输入到达程序的哪部分。例如，对于第十二章“综合应用”中将要介绍的 Masterindex 程序，当对它进行调试时，我们想知道包含单词“retrieving”的条目在程序的特定部分是否被处理。将下面的行插入到我们认为它在程序中应该出现的地方：

```
if ($0 ~ /retrieving/) print ">> retrieving" > "/dev/tty"
```

当运行程序时，如果它遇到字符串“retrieving”，它将打印一个信息。（“>>”是作为一对字符来用的，用来立即引起对输出的注意，“!!”也是一个很好的标志。）

有时你可能不知道是哪个 print 语句引起的问题。可以在 print 语句中插入标识符，以提示执行了哪个 print 语句。在下面的例子中，我们只简单地使用变量名作为标签来识别打印了什么：

```
if (PRIMARY)
        print (">>PRIMARY:", PRIMARY)
else
        if (SECONDARY)
                print (">>SECONDARY:", SECONDARY)
        else
                print (">>TERTIARY:", TERTIARY)
```

这项技术也被用来检查程序的某个部分究竟是否被执行。修改程序就像是重新装修一个家，在这里增加一个房间，在那里拆掉一堵墙。试图理解这些基本结构可能是困难的。你可能想知道是否每个部分都是需要的或它们确实被执行。

如果一个 awk 程序是几个程序的一个管道的一部分，甚至是其他的 awk 程序的一部分，你可以用 tee 命令将输出重定向到一个文件，同时也将输出用管道传递给下一个命令。例如，参见在第十二章中的 masterindex 程序，运行该程序的 shell 脚本如下：

```
$INDEXDIR/input.idx $FILES |
sort -bdf -t: +0 -1 +1 -2 +3 -4 +2n -3n | uniq |
$INDEXDIR/pagenums.idx | tee page.tmp |
$INDEXDIR/combine.idx |
$INDEXDIR/format.idx
```


运维技术书籍免费下载 https://www.linuxprobe.com

通过添加“tee page.tmp”，我们能够将 pagenums.idx 程序的输出结果捕获到文件 page.tmp 中。相同的输出通过管道输送到 combine.idx。

利用注释排除干扰

另一个简单的技术是注释掉一系列可能引起问题的行，看它们是否真的有问题。这里建议使用连续的两个字符符号，例如，“#%”来临时注释掉这些行。这样在后续的编辑中就能注意到它们，并做想应的处理。也便于将这些注释符号去掉并用一个编辑命令来恢复这些行，而不至于影响程序的注释。

```
#% if ( thisFails )
      print "I give up"
```

这里将条件注释掉，使得 print 语句能够无条件地被执行。

Slash and burn

当所有的方法都失败时，可以使用编辑器来删除命令或删除部分程序直到错误消失。当然，要对程序做一个备份并在临时备份上删除行。这是一个非常笨拙的技术，却是在全部放弃或重新从头开始之前一个有效的办法。有时当仅有的结果是核心程序中断时，这是惟一可以用来查找出现什么问题的方法。其思路也是相同的，也是将可能引起问题的部分孤立出来。例如删除一个函数或 for 循环，看它是否是引起问题的原因。确信删除的是完整的单元，例如，大括号之间的所有语句及相应的大括号。如果问题仍然存在（程序仍然中断），那么删除程序的另一部分。你迟早会发现引起问题的那部分代码。

可以用“slash and burn”来了解程序是如何工作的。首先，使用样本输入运行原始程序，并保存输出。从你不理解的那部分程序代码开始进行删除。然后在样本输入上运行修改过的程序并和原始的输出相比较。看发生了什么变化。

为脚本设置防御措施

各种类型的输入错误和不一致会使脚本的运行出现问题。你可能会认为用户错误不应被认为是程序问题。因此，一个好的建议就是将你的核心程序用“防御”过程包围起来，以俘获不一致的输入记录和防止程序的意外失败。例如，你可能想在处理每个记录前对它进行验证，确保存在正确的字段数或指定字段具有所希望的数据类型。

另一个和防御技术有关的方面是错误处理。换句话说就是一旦程序检测到错误时，你希望发生什么？在某些情况下可以使程序继续运行，在加外一些情况下可以使程序打印出错信息和/或使程序终止。

要认识到 awk 脚本通常只限定在固定领域，程序主要是要解决某个特定的问题

运维技术书籍免费下载 https://www.linuxprobe.com

而不是解决许多不同用户遇到的一系列问题。因为这些程序的这种特点，要求它们具有专业的质量是没有必要的。因此，编写 100%的用户层程序是没有必要的。一方面，防御程序是很耗费时间和乏味的。另一方面，作为业余的编程人员，我们可以自由地按我们希望的实现方式编写程序；作为一个专业的编程人员则必须按客户的要求来编写。简单地说，如果你是为别人编写程序，在考虑如何完成程序之前先考虑它将被用来做什么以及用户将会遇到什么问题。否则，了解脚本工作的实际情况（即使是很小的设置环境）就足够了而且有时间处理它。

约束

在任何 awk 实现中都有固定的约束。惟一的麻烦是它们的文档很少介绍它们。表 10-1 列出了在《The AWK Programming Language》中描述的约束。这些特定实现工具的约束，但对大多数系统都是一个好的参考值。

表 10-1：约束

项目	约束
每个记录中字段的个数	100
每个输入记录的字符个数	3000
每个输出记录的字符个数	3000
每个字段的字符个数	1024
每个 printf 字符串的字符个数	3000
字面字符串中的字符个数	400
字符类中的字符个数	400
打开的文件数	15
打开的管道数	1

注意：和表 10-1 中的数据不同的是，经验表明大多数 awk 允许打开的管道数大于 1。

对于数值型数据项，awk 使用双精度型，浮点型数据的长度限制由机器的结构决
定。

超过这些约束将使脚本产生无法预测的问题。在编写本书的第一版的例子中，Dale 认为他应编写一个查询程序，用于在一段中查询一个单词或一个单词序列。思路是将文档作为一系列多行记录来读取，如果记录中的字段包含了要查找的项，则打印这个记录，也就是一段。这个方法可用来查询用空行作为段的

运维技术书籍免费下载 https://www.linuxprobe.com

分界的邮件文件。这个程序对小的测试文件起作用。但是，在处理大的文件时，程序产生了中断，因为程序遇到了一个超过输入记录最大值 3000 个字符的段（实际上，文件中包含一个内置的邮件信息，其中空行以“>”为前缀）。因此，当读入多行作为一个记录时，要确保记录的长度没有超过 3000 个字符。另外，将没有特殊的出错信息来提示你当前记录的长度问题。

幸运的是 gawk 和 mawk（参阅第十一章“awk 的系列产品”）没有那么小的限制，例如，gawk 中一个记录中的字段数目最大不可超过 long 类型所能表示的范围，当然记录可以比 3000 个字符长。这些版本允许打开更多的文件和管道。

Bell Labs awk 的当前版本有两个选项：-mfN 和-mrN，用于设置字段的最大数和命令行中记录的最大长度，作为避开默认限制的应急办法。

（sed 实现也有它的限制，这些没有在文档中说明。实践表明，大多数 UNIX 版本的 sed 对替换（s）命令数的限制是 99 或 100。）

使用#!语法调用 awk

“#!”语法是从 shell 脚本中调用 awk 的可选的语法。它的优点是你可以在 shell 脚本的命令行中指定 awk 的参数和文件名。“#!”语法可以用在现代 UNIX 系统中，但在老的 System V 系统中没有发现。运用这个语法的最好方法是将下面一行作为 shell 脚本的第一行（注 6）：

```
#!/bin/awk  -f
```

“#!”后面跟的是所用的 awk 所在的路径名，然后是-f 选项。在这一行之后，你可以写 awk 脚本：

```
#!/bin/awk -f
{ print $1 }
```

注意，脚本周围不必使用引号。在第一行后面的所有行都能够像在单独的脚本文件中一样被执行。

几年以前，关于在网络上如何使用“#!”语法的讨论阐明了它是如何工作的。这个讨论是由 4.2BSD 用户发现下面 shell 脚本的执行失败而引起的：

```
#!/bin/awk
{ print $1 }
```

注 6： 注意所使用的路径名依赖于系统。

而下面的这个脚本能工作：

运维技术书籍免费下载 https://www.linuxprobe.com

```
#!/bin/sh
/bin/awk ' { print $1 }'
```

我们看到的这两个情况是由 Chris Torek 和 Guy Harris 发现的，我们将概括介绍他们的解释。第一个脚本执行失败是因为它将脚本的文件名作为第一个参数（C 中的 argv[1]）传递，而 awk 将它作为输入文件而不是脚本文件解释。因为没有给出脚本，所以 awk 产生一个语法出错信息。换句话说，如果 shell 脚本的名字是“myscript”,那么第一个脚本将按下面语句执行：

```
/bin/awk myscript
```

如果将程序改为添加-f 选项，如下所示：

```
#!/bin/awk -f
{ print $1 }
```

然后输入下面的命令：

```
$ myscript myfile
```

它将和输入下面的语句一样来执行：

```
/bin/awk -f myscript myfile
```

注意： 在“#!”行只能有一个参数。这行将由 UNIX 内核来处理，而不是由 shell 处理，因此不能包含任意 shell 构件。

“#!”语法允许你创建 shell 脚本，来将命令行参数透明地传递给 awk。换句话说，你可以通过调用 shell 脚本的命令行来给 awk 传递参数。

例如，我们通过改变 awk 样本脚本来传递参数 n:

```
{ print $1*n }
```

假设我们有一个测试文件，在它的第一个域中包含一个可以被 n 乘的数，我们可以按下面所示方式调用这个程序：

```
$ myscript n=4 myfile
```

这使我们省去了必须将“$1”作为一个 shell 变量，并在 shell 脚本中将它作为 awk 的参数赋给 n。

第十二章将要介绍的 masterindex,就是使用“#!”语法来调用 awk。如果你的

运维技术书籍免费下载 https://www.linuxprobe.com

系统不支持这个语法，可以将脚本中的“#!”删除，在整个脚本两端添加一对单引号，并用“$*”来结束脚本，这样就扩展到 shell 命令行的所有参数。

好的，我们已经清楚了底部抽屉。本章的许多内容关系到 awk 与 UNIX 操作系统的接口，调用其他实用工具，打开和关闭文件，以及使用管道。而且，我们也介绍了调试 awk 脚本的一些笨拙的技术。

我们已经涵盖了 awk 程序设计语言的所有功能。我们集中介绍了 awk 的 POSIX 规范，偶尔提了一些 awk 实用工具。在下一章将介绍不同的 awk 版本之间的区别。第十二章介绍两个大而复杂的应用程序：文档拼写检查程序和索引程序。第十三章“脚本的汇总”给出了许多用户提供的程序，提供了如何编写程序的补充例子。

运维技术书籍免费下载 https://www.linuxprobe.com

第十一章
awk 的系列产品

本章内容

- 原始的 awk
- 可免费使用的 awk
- 商业版 awk
- 后记

讨论了 POSIX awk，偶尔介绍了一些可能
本章中，我们将重点讨论可用的不同 awk 版本，即它们具有或不具有哪些功能，以及如何得到它们。
首先，我们将讨论 awk 原始的 V7 版本。awk 原始版本中缺少很多我们以及介绍过的功能，因此这一届主要介绍它没有的哪些功能。其次，我们还将介绍 3 个版本，这些版本的源代码是可以免费使用的。所有的这些已经扩展为 POSIX 标准。先从讨论 3 个版本的共同特点开始。最后，我们来讨论 awk 的 3 个商业版。

原始的 awk

在下面的每节中，我们将对原始 awk 与 POSIX awk 之间的区别做简单的描述。几年来，UNIX 销售商们已经增强了我们的原始的 awk 版本，你只需要编写一个小的测试程序来检测老的 awk 有什么功能或没有什么功能。

转义序列

在原始的 V7 awk 中只有“\t”，“\n”，“\””，和“\\”。大多数 UNIX 销售商添加了“\b”，“\r”，和“\f”中的部分或全部。

求幂

在老版本的 awk 中没有求幂（使用^、^=、**和**=操作符）计算。

C 语言的条件表达式

在 C 语言中，三元条件表达式“expr1 ? expr2 : expr3”没有在老版本的 awk 中出现。你必须求助于普通的 if-else 语句。

将变量值作为布尔模式

不能将一个变量的值作为布尔模式。

```
flag { print "..." }
```

而必须用一个比较表达式来代替。

```
flag != 0 { print "..." }
```

伪动态正则表达式

在原始的 awk 中很难动态使用模式，因为当解释脚本时它们必须是固定的。对于不能使用变量作为正则表达式这一问题，可以通过将一个

运维技术书籍免费下载 https://www.linuxprobe.com

shell 变量输入到 awk 程序中来解决。Shell 变量的值将被 awk 解释为一个常量。下面一个例子：

```
$ cat awkro2
#!/bin/sh
# 给 awk search 变量赋给 shell 的$1
search=$1
awk ' $1 ~ /' "$search" ' /'  acronyms
```

这个脚本的第一行在调用 awk 之前定义了一个变量。为了使 shell 可以识别 awk 变量，我们先用单引号，在用双引号包围它（注 1）。因此，awk 看不到 shell 变量并将它作为字符常量来处理。

注 1： 实际上，这是单引号引住的文本和双引号引住的文本，以及更多的单引号引住的文本，来产生更大的引用文本的级联。这种技巧在第六章已经使用过了。

下面是使用 Bourne shell 变量替换功能的另一个示例。使用这一功能，我们可以很容易地为变量指定默认值，如果用户没有给出命令行参数。

```
search=$1
awk ' $1 ~ /' "${search:-.*}"' /'  acronyms
```

表达式 “${search:-.*}”告诉 shell 如果 search 有定义则使用 search 的值；如果没有定义，就用“.*”的值。这里，“.*”是正则表达式语法，用于指定任意字符组成的字符串；因此，如果在命令行没有给出参数，将打印所有的记录。因为整个模式都包含在双引号中，shell 不执行通配符表达式“.*”。

控制流

在 POSIX awk 中，如果一个程序只有一个 BEGIN 过程，并没有其他的过程，awk 将在执行完该过程后退出。原始的 awk 则不同，它将执行 BEGIN 过程，然后继续处理输入，即使没有模式处理语句。可以在命令行中给出/dev/null 作为一个数据文件参数，或使用 exit 来强制退出 awk。

另外，如果有 BEGIN 和 END 过程，则必须分别出现在程序的开始处和末尾处。而且二者都只能出现一次。

字段的分隔

字段的分隔在老版本的 awk 中和现代的 awk 中其工作方式是一样的，除非你不能使用正则表达式。

数组

在原始的 awk 中没有提供从数组中删除一个元素的方法。你所能做的

运维技术书籍免费下载 https://www.linuxprobe.com

最好的工作就是将一个空字符串，赋给不需要的数组元素，并修改程序代码以忽略数组中值为空的元素。
同样，在原始的 awk 中 in 不是一个操作符，你不能用 if（item in array）来检测某项是否出现在数组中。不幸的是，你必须循环访问数组的所有元素来检测你希望的下标是否存在。

```
    for (item in array) {
      if (item == searchkey) {
            process array[item]
           break
      }
    }
```

getline 函数

在原始的 V7 awk 中没有 getline 函数。如果你的 awk 是老版本，则不能使用 getline。某些销售商有 getline 的最简单形式，它从普通的输入流中读取下一个记录，并设置$0、NF 和 NR（这里没有 FNR，参见下面的内容）。getline 的其他形式都是不可用的。

函数

原始的 awk 中只有有限的几个内置字符串函数（参见表 11-1）。
表 11-1：原始的 awk 中的内置字符串函数

awk 函数	描述
index（s，t）	返回字串 t 在字符串 s 中的位置，如果没有出现则返回 0
length（s）	返回字符串 s 的长度，当没有给出字符串时返回$0 的长度
split（a，s，sep）	使用字段分隔符 sep 将字符串 s 分解为数组 a 的元素，返回元素的个数。如果没有给出 sep，则使用 FS,数组分割和字段分割采用一样方法
sprintf（“fmt”，expr）	为 expr 使用 printf 格式规范
substr（s，p，n）	返回字符串 s 中从位置 p 开始最大长度为 n 的子串。如果 n 没有给出，返回从 p 开始剩余的部分

有些内置函数被归为算术函数。它们大多输入一个数值型参数并返回一个数值型值。

表 11-2 概括了这些算术函数。
表 11-2：原始的 awk 中的内置算术函数

运维技术书籍免费下载 https://www.linuxprobe.com

awk 函数	描述
exp（x）	返回 e 的 x 次幂
int（x）	返回 x 取整后的值
log（x）	返回 x 的自然对数（以 e 为底）
sqrt（x）	返回 x 的平方根

awk 中最大的便利就是可以定义自己的函数，但原始 awk 没有提供这一功能。

内置变量

原始的 awk 中的内置变量显示在表 11-3 中。

表 11-3：原始的 awk 中的系统变量

变量	描述
FILENAME	当前文件名
FS	字段分隔符（默认为一个空格
NF	当前记录中的字段的个数
OFMT	数字的输出格式（默认为%6g）
OFS	输出字段分隔符（默认为一个空格）
ORS	输出记录分隔符（默认为一个换行符）
RS	记录分隔符（默认为一个换行符）

OFMT 有两个作用，即负责 print 语句的格式转换和将数字转换为字符串。

可免费使用的 awk

有 3 个 awk 版本的源代码是可以免费使用的。它们是 Bell Labs awk、GUN awk、和 mawk，由 Michael Brennan 提供。这部分介绍了它们中的两个或多个版本共有的扩展功能，然后详细介绍每个版本以及和如何获得它们。

共有的扩展

本节讨论 awk 语言的扩展，这在 awk 中的两个或多个免费版本（注 2）中的可用的。

删除数组的所有元素

3 个版本都扩展了 delete 语句，使得能一次删除数组中的所有元素。语法为：

```
delete array
```

通常情况下，要从数组中删除每个元素，必须用一个如下的循环：

```
for (i in data)
    delete data[i]
```


运维技术书籍免费下载 https://www.linuxprobe.com

这对于有很多下标的数组尤其有用，这种方法比用一个循环要快得多。
即使数组中已经没有任何元素，也不能将数组名作为一个简单的变量名。一旦定义为数组就永远是一个数组。
这一扩展首先出现在 gawk 中，然后出现在 mawk 和 Bell Labs awk 中。

获得单独的字符

3 个 awk 都扩展了字段的分割和数组的分割。如果 FS 得值一个空字符串，那么输入记录的每个字符都变成为一个单独的字段。这大大地简化需要处理单独字符的工作。
类似地，如果函数 split()的第三个参数是空字符串，在原始字符串中的每个字符将成为目标数组的一个单独的元素。
如果没有这些扩展，则必须重复调用 substr()函数来获取每一个字符。

注 2： 作为 gawk 的维护人员和在这里以及在后面一节中有关 gawk 的许多扩展的描述的作者，我对这些扩展的有效性的看法也许有偏见。你应该有自己的评价。（A.R.J）

这些扩展首先出现在 mawk 中，然后出现在 gawk 和 Bell Labs awk 中。

刷新被缓存的输出

Bell Labs awk 的 1993 年版本介绍了一个 POSIX 标准中没有的新的函数，fflush()。和 close()一样，fflush()参数是一个打开的文件名或一个管道。和 close()不同的是，fflush()函数仅对输出文件和管道有效。
大多数程序对输出进行缓存，在内存块中保存将被写入文件或管道的数据，直到有足够多的数据后才将它们输送到目的地。有时，程序员能够显示地刷新出缓冲区是很有用的，也就是说，强制地传递缓冲器中所有被缓存的数据。这也是函数 fflush()的目的。
这个函数首先出现在 Bell Labs awk 中，然后出现在 gawk 和 mawk 中。

特殊文件名

对于任意版本的 awk，都能直接向 UNIX 特殊文件/dev/tty 中写入数据，这是用户终端的名字。当程序的输出直接写入一个文件时，这可以用来提示或通知用户的注意。

```
printf "Enter your name:" >"/dev/tty"
```

这将“Enter your name:” 直接输出到屏幕上，不管标准输出和标准错误被定向到哪里。

运维技术书籍免费下载 https://www.linuxprobe.com

在表 11-4 列出的是 3 个版本 awk 中都支持的特殊文件名。

表 11-4： 特殊文件名

文件名	描述
/dev/stdin	标准输入（mawk 中没有）a
/dev/stdout	标准输出
/dev/stderr	标准错误

a. Mawk 帮助页建议使用“-”表示标准输入，这是很方便的。

注意，和其他文件名一样，当特殊文件名被指定为字符串常量时必须用引号包围。特殊文件/dev/stdin、/dev/stdout 和/dev/stderr 来源于 V8 UNIX。这些特殊文件在 gawk 中首先获得认可，然后是在 mawk 和 Bell Labs awk 中获得认可。

Printer()函数

提供给用户的出错信息经常是因为漏掉输入或错误的输入。你可以简单用 print 语句来提示用户。但是，如果程序的输出被重定向到一个文件中，用户将看不到打印结果。因此，一个好的办法是明确指出将出错信息送到终端。

下面的 printerr()函数用于帮助创建一致的用户出错信息。它打印一个单词“ERRO”，后跟有一个给出的信息、记录编号和当前记录。下面的例子直接输出到/dev/tty:

```
    function printerr (message) {
                    # print message, record number and record
                    printf("ERROR:%s (%d) %s\n", message, NR,
$0) > "/dev/tty"
    }
```

如果程序的输出直接被送到终端屏幕，那么出错信息将和输出混合在一起。输出“ERROR“可以帮助用户识别出出错信息。

在 UNIX 中，出错信息的标准目的地是标准错误文件。将出错信息写入标准错误的基本原理和上面一样。要显式地指定标准错误的输出，必须像下面的例子那样使用“cat 1 > &2“语法:

```
  print "ERROR" | "cat 1>&2"
```

这个例子直接将 print 语句的输出传递给一个执行 cat 命令的管道。你也可以用 system()函数来执行一个 UNIX 命令，例如，cat 或 echo，并直接将它的输出结果送到标准错误文件。

当特殊文件/dev/stderr 能 用时，这将变得很简单:

```
    print "ERROR" > "/dev/stderr" # recent awks only
```

nextfile 语句

运维技术书籍免费下载 https://www.linuxprobe.com

nextfile 语句和 next 语句类似，但它是更高层次上的操作。当执行 nextfile 时，当前的数据文件将被放弃，操作从脚本的顶端开始，并使用下一个文件的第一个记录。当你只需要处理文件的一部分时，这是很有用的，这里不需要创建一个循环并使用 next 跳过某些记录。nextfile 语句来源于 gawk，然后被添加到 Bell Labs awk 和 mawk 中，在 mawk 的 1.4 版本中这个语句开始可用。

正则表达式的记录分隔符（gawk 和 mawk）

gawk 和 mawk 允许 RS 是一个完全的正则表达式，而不仅仅是单个字符。在这种情况下，记录被输入中与正则表达式匹配的最长文件分隔。gawk 也将 RT（记录终止符）设置为实际与 RS 匹配的输入文本。在下面给出了一个例子。

将 RS 作为正则表达式的性能首先出现在 mawk 中，并随后被加到 gawk 中。

Bell Labs awk

Bell Labs awk 是原始的 V7 awk 的直属后代，这个“新的”awk 首先应用在 System V 3.1 版本中，它的可免费使用的源代码可用通过匿名 FTP（文件传送协议）从主机 netlib.bell-labs.com 下载。位于文件/netlib/research/awk.bundle.Z 中。这是一个压缩的 shell 档案库文件。确保使用“二进制”或“图像”模式来传输这个文件。这个版本的 awk 需要一个 ANSI C 编译器。

这里还有几个不同的版本，我们将根据它们在哪一年可用来识别它们。

新 awk 的第一个版本在 1987 以后可以使用。它几乎包含了我们在前面四章中介绍的所有内容（尽管有些脚本注明了一些是不可用的）。这个版本一直在 SunOS 4.1.x 系统和 System V 第三版 UNIX 系统上使用。

在 1989 年，System V 第四版增加了一些新的内容。这个版本和 POSIX awk 之间的唯一区别是 POSIX 使用 CONVFMT 实现数据到字符串的转换，而 1989 的版本用的是 OFMT，新的功能有：

- 在命令行赋值的转义字符现在可以解释
- 添加了 tolower()和 toupper 函数
- 改进了 printf：增加了动态宽度和提高了精度，“%c”的作用被合理化
- 函数 srand()的返回值被定义为前一个种子数。（awk 书中没有陈述 srand()返回的是什么）
- 将正则表达式作为简单的表达式成为可能。例如：

```
if (/cute/ || /sweet/)
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
print "potential here!"
```

- 增加了-v 选项，允许在执行 BEGIN 过程之前在命令行设置变量
- 可以使用多个-f 选项来处理多个源文件（这首先起源于 MKS awk，然后被 gawk 采用，最后添加到 Bell Labs awk 中）
- 增加了 ENVIRON 数组（这分别由 MKS awk 和 gawk 两者独立开发而来，然后添加到 Bell Labs awk）

在 1993 年，贝尔实验室的 Brian kernighan 公开了他的 awk 的源代码。在这时，CONVFMT 变成可用，并且增加了在前面介绍的函数 fflush()。一个能锁定错误的版本在 1994 年 8 月发布。

在 1996 年 6 月， Brian Kernighan 发布了另一个版本。它不仅可以在前面给出的 FTP 站点上检索到，而且可以在通过 WWW 浏览器访问 Dr.Kernighan’s Web 主页（http://cm.bell-labs.com/who/bwk），这里将这个版本称为“一个真正的 awk”。这个版本增加了源于 gawk 和 mawk 的几个特点，这些特点在本章前面的“共有的扩展”一节介绍过。

GUN awk (gawk)

免费软件机构的 awk 的 GNU 企业版，即 gawk，实现了 POSIX awk 中的所有特征，甚至更多。它也许是可免费使用的实现工具中最流行的一个。gawk 用于 Linux 系统，以及各种其他可免费使用的类似于 UNIX 的系统中，例如 NetBSD 和 FreeBSD。

gawk 的源代码可以通过连接到主机 ftp.gun.org 上的匿名 FTP（注 3）获得。它在文件/gun/gawk/gawk-3.0.4.tar.gz 中（当你访问这个文件时可能已经出了新的版本）。它是经过 gzip 程序压缩过的 tar 文件，它的源代码在相同的目录中可以获得。世界上很多站点都从 GUN 发行站点上“镜像”了这个文件，如果有距离近的站点，你可以从那里获得这个文件。确保用“二进制”或“镜像”模式来传输这个文件。

注 3： 如果你不能使用 Internet 却希望得到一份 gawk 的副本，可以联系 Free software foundation.Inc.，59 Temple Place，Suite 330，Boston，MA 0211-1307 U.S.A.

除了从前介绍的共有的扩展外，gawk 还有一些其他的特点。下面简单介绍这些特点。

运维技术书籍免费下载 https://www.linuxprobe.com

命令行选项

gawk 有几个很有用的命令行选项。和大多数 GUN 程序一样，这个选项具有清楚地说明，并以两个短线“--”开始。

--lint 和--lint-old 使 gawk 在解析时和运行时对于不适用于或不可移植到其他 awk 版本的程序结构进行检查。--lint-old 选项用于提示在 awk 的原始版本中不适用的函数调用。他和--lint 是互相独立的，因为大多数系统中都包含新 awk 的一些版本。

--traditional 使得 GUN 的特殊扩展不可用，例如时间函数和 gensub()函数（参见下面）。使用这个选项，可以是 gawk 和 Bell Labs awk 具有相同的工作方式。

--re-interval 通过使 gawk 可以识别区间表达式（例如“/stuff{1,3}/”）而使得完全的 POSIX 正则表达式匹配有效。

--posix 使所有在 POSIX 标准中没有指定的扩展功能不可用。这个选项也打开对区间表达式的识别功能。

还有一些其他的选项对于日常的程序设计和脚本移植是不重要的，细节可以参阅 gawk 的文档。

尽管 POSIX awk 允许使用-f 选项来处理多个实例，但在命令行程序中使用库函数一简单的方法。gawk 中的—source 选项可以完成这一功能。

```
gawk --source ’ script’ -f mylibs.awk file1 file2
```

这个例子运行 script 中的程序，它可以使用来自文件 mylibs.awk 中的 awk 函数。输入数据来自 file1 和 file2。

一个 awk 程序搜索路径

gawk 允许你指定一个名为 AWKPATH 的环境变量，它定义了 awk 程序文件的搜索路径。默认路径为：/usr/local/share/awk。因此，当为一个文件名指定-f 选项时，将从当前路径开始查找两个默认路径。注意，如果文件名中包含一个“/”字符将不执行查找。

例如，如果 mylibs.awk 是/usr/local/share/awk 中的一个 awk 函数文件，而 mygrog.awk 是当前目录下的一个程序，我们可以用下面的方式运行 gawk：

```
gawk -f myprog.awk -f mylibs.awk datafile1
```

gawk 可以在适当的地方找到每个文件。这使得建立和使用 awk 库函数变得很容易。

运维技术书籍免费下载 https://www.linuxprobe.com

行的延续

gawk 允许使用“?”或“:”实现断行。也可以用一个反斜杠字符串延续到新行中。

```
$ gawk ' BEGIN { print "hello, \
> world" }'
hello, world
```

扩展的正则表达式

gawk 提供了几个补充的正则表达式操作符。这些对多数使用正则表达式的 GNU 程序是共有的。扩展操作符列于表 11-5 中。

表 11-5：gawk 的扩展表达式

特殊操作符	用途
\w	和任何可以做单词组成成分的字符匹配（字母、数字或下划线）
\W	和任何不能做单词组成成分的字符匹配
\<	和一个单词开头的空字符串匹配
\>	和一个单词末尾的空字符串匹配
\y	和一个单词开头的空字符串或末尾的空字符串匹配（单词边界）。其他 GUN 软件使用“\b”，这是已经被接受的
\B	和单词内部的空字符串匹配
\'	和在缓冲区开始处的空字符串匹配。这和在 awk 中的字符串是一样的，因此和$作用相同。他用于与 GUN Emacs 和其他 GUN 软件保持兼容

运维技术书籍免费下载 https://www.linuxprobe.com

\’	和在缓冲区末尾处的空字符串匹配。这和在 awk 中的字符串是一样的，因此和$作用相同。他用于与 GUN Emacs 和其他 GUN 软件保持兼容

你可以将“\w”作为（POSIX）符号[[:alnum:]_]的一个简写，而将“\W”作为[^[:alnum:]_]的简写。下面的表中给出了与中间 4 个操作符匹配的例子，这是从《Ef fective AWK Programming》中借用的。

表 11-6: gawk 扩展正则表达式示例

表达式	匹配的例子	不匹配的例子
\<away	away	stowaway
stow\>	stow	stowaway
\yballs?\y	ball or balls	ballroom or baseball
\Brat\B	crate	dirty rat

正则表达式记录终止符

除了 RS 是一个正则表达式外，gawk 将变量 RT（记录终止符）设置为与 RS 的值匹配的实际输入文本。

下面是来自 Micheal Brennan 的一个简单的例子，显示了 gawk 中的 RS 和 RT 变量的功能。就像我们所看到的，sed 的一个最普遍的应用是它的替代命令（s/old/new/g）。通过将 RS 设置为要匹配的模式，将 ORS 设置为替代文本，使用一个简单的 print 语句就能够原样打印替换后的文本。

```
$ cat simplesed.awk
# simplesed.awk --- do s/old/new/g using just print
# Thanks to Michael Brennan for the idea
#
# NOTE! RS and ORS must be set on the command line
{
    if (RT == " ")
        printf "%s", $0
    else
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
        print
}
```

这里有一个技巧，在文件的末尾RT将是空的，因此我们用printf语句打印这个记录（注4）。我们可以按下面方式运行这个程序。

```
$ cat simplesed.data
"This OLD house" is a great show.
I like shopping for old things at garage sales.
$ gawk -f simplesed.awk RS="old|OLD" ORS="brand new" simplesed.data
"This brand new house" is a great show.
I like shopping for brand new things at garage sales.
```

分隔字段

除了awk运行你用常规的方式将分隔成记录并将记录分隔字段外，gawk还提供的一些补充的功能。

首先，和上面提到的一样，如果FS的值为空字符串，那么输入记录的每个字符都成为一个独立的字段。

其次，特殊变量FIELDWIDTHS可以用来分隔出现在固定宽度列中的数据。这些数据可能或不可能由空白字符来分隔字段的值。

```
FIELDWIDTHS = "5 6 8 3"
```

这里的记录有4个字段：$1有5个字符的宽度，$2有6个字符的宽度等等。为FIELDWIDTHS赋一个值将导致gawk开始使用它来分隔字段。为FS指定一个值将导致gawk恢复常规的字段分隔机制。使用FS=FS将使这种恢复发生，且不必将FS的值保存到额外的变量中。

当处理固定宽度字段的数据且记录中间没有任何分隔字段的空白字符时，或者当中间字段内容全部为空白时，这个功能最有用。

注4： 参阅《Effecive AWK Programming》[Robbins]，16.2.8节，这个程序的精确版本。

运维技术书籍免费下载 https://www.linuxprobe.com

补充的特殊文件

gawk 有一些补充的特殊文件名，它们在内部解释。所有的特殊文件名列在表 11-7 中。

表 11-7：gawk 的特殊文件名

文件名	描述
/dev/stdin	标准输入
/dev/stdout	标准输出
/dev/stderr	标准错误
/dev/fd/n	用文件描述符 *n* 引用的文件名

废除的文件名	描述
/dev/pid	返回包含进程 ID 号的记录
/dev/ppid	返回包含父进程 ID 号的记录
/dev/pgrpid	返回包含进程组 ID 号的记录
/dev/user	返回包含真是有效地用户 ID 号，真实有效地组 ID，以及如果可以得到，还包括某些次要组的 ID 号的记录

最前面的 3 个文件名在以前解释过。第四个文件名用于访问任何打开文件描述符，这可能从 gawk 的父进程（经常是 shell）中继承的。你可以用文件描述符 0 表示标准输入，1 表示标准输出，2 表示标准错误。

第二组标为“废除的”特殊文件名，在 gawk 中曾经出现了一段时间，但是正被逐渐淘汰，它们将被数组 PROCINFO 替代，它的下标是想得到的项，而且它的元素值是相关的值。

例如，我们可以用 PROCINFO[“pid”]得到当前进程的 ID 号，而不是用 getline pid < “/dev/pid”。查看 gawk 的文档看 PROCINFO 是否可用，以及是否还支持这些文件名。

补充变量

gawk 还有几个补充的系统变量，它们列在表 11-8 中。

表 11-8：补充的 gawk 系统变量

运维技术书籍免费下载 https://www.linuxprobe.com

变量	描述
ARGIND	当前输入文件的 ARGV 中的引索
ERRNO	当 getline 或 close()失败时的描述错误信息
FIELDWIDTHS	用空格分隔的数据列表，用于描述输入字段的宽度
IGNORECASE	如果不为 0，模式匹配和字符串比较是不区分大小写
RT	和 RS 匹配的输入文本的值

我们已经看到过记录终止符变量 RT，现在我们对没有介绍过的变量进行介绍。

在 awk 中，所有的模式匹配和字符串比较都是区分大写的。gawk 引入了 IGNORECASE 变量，因此能将正则表达式指定为不区分大小写。从 gawk3..0 版本开始可以做不区分大小写的字符串比较。

IGNORECASE 的默认值为 0，这意味着模式匹配和字符串比较与在传统的 awk 中执行是一样。如果 IGNORECASE 被设置为一个非零值，则忽略大小写。这应用于所有可以用正则表达式的地方，包括字段分隔符 FS，记录分隔符 RS，以及所有的字符串比较。但对数组的下标不起作用。

有两个有趣的 gawk 变量。ARGIND 被 gawk 自动设置为当前输入文件名的 ARGV 中的引索。这个变量可以为你提供一个跟踪你在文件名列表中当前位置的方法。

最后，如果在为 getline 重定向时产生错误或执行 close()期间产生错误，gawk 为 ERRNO 设置一个字符串来描述这个错误。这使得当发生错误时能够提供描述错误的信息。

补充函数

gawk 有一个补充的字符串函数，以及两个处理当前日期和时间的函数，它们列在表 11-9 中。

gawk 函数	描述
gensub(*r*,*s*,*h*,*t*)	如果 *h* 以 g 或 G 开始的字符串，则对于在 *t* 中的 *r*，用 *s* 进行全局替换。否则，*h* 是一个数据：替换第 *h* 次出现的 *r*。该函数返回新值，*t* 没有发生变化。如果没有给出 *t*，默认为$0
systime()	返回用秒表示的天的当前的时间，初始时间为（00:00 am January 1，1970 UTC）
strftime(*format*,*timestamp*)	依照 *format* 格式化 *timestamp*（和 *systime*()

运维技术书籍免费下载 https://www.linuxprobe.com

	返回的形式相同）。如果没有 *timestamp*，则使用当前时间。如果也没有 *format*，用默认格式，它的输出和 data 命令的输出类似

一个通用的替换函数

gawk 的 3.0 版本引入了一个新的通用的替换函数，命令 gensub()。函数 sub()和 gsub()有如下一些问题。

- 你可以改变模式的第一次匹配或者改变模式的所有匹配。例如，没有办法仅改变模式的第三次匹配，而不改变它前面或后面的匹配。
- sub()和 gsub()都会改变实际的目标字符串，这可能不是希望的。
- 要让函数 sub()和 gsub()忽略后面跟有匹配的文本的一个字面反斜杠是不可能的，因为一个&字符的前面有反斜杠，它不能替换的（注 5）。
- 没有办法获得被匹配的文本的一部分，类似与 sed 中的\(…\)结构。

由于所有的这些原因，gawk 引入了函数 gensub()。这个函数至少有 3 个参数。第一个是要查找的正则表达式。第二个是替换字符串。第三个是一个标志，用于控制应该执行多少次替换。第四个参数（如果出现）是要改变的原始字符串。如果没有给出第四个参数，则使用当前输入记录（$0）。

这里的模式可以有包含在圆括号中的子模式。例如，可以是“/(part)(one | two | three)/”，在替换字符串中，反斜杠后面跟着一个数字表示和 n 个子模式匹配的字符串。

注 5： 在 Effecive AWK Programming》[Robbins]， 12.3 节有完整的讨论。这些细节对于软弱的核心没有作用。

```
$ echo part two | gawk ’ { print gensub(/(part) (one|two|three)/, "\\2", "g") }’
two
```

这里的标志是以 g 或 G 开始的字符串。和 sub()和 gsunb()不同，目标字符串没有改变。相反，新的字符串是 gensub()的返回值。

```
$ echo a b c a b c a b c | gawk ’ { print gensub(/a/, "AA", 2) }’
a b c AA b c a b c
```

第四个参数是改变的原始字符串。和 sub()和 gsub()不同，目标字符串没有改

运维技术书籍免费下载 https://www.linuxprobe.com

变。相反，新的字符串是 gensub()的返回值。

```
$ gawk '
BEGIN { old = "hello, world"
  new = gensub(/hello/, "goodbye", 1, old)
   printf("<%s>, <%s>\n", old, new)
}'
<hello, world>, <goodbye, world>
```

为程序员提供的时间管理

awk 程序常用于处理由各种程序产生的日志文件。通常，日志文件中的每个记录都有一个时间戳，表示这个记录是什么时间产生的。为了简明和精确，时间戳是系统调用 UNIX *time*(2)的返回结果，这是从初始时间 00:00 am January 1, 1970 UTC 开始的用秒表示的数据（这个日期经常被作为“新纪元”）。为了在 gawk 中更容易地产生和处理具有这种时间戳的日志文件记录，gawk 提供了两个函数，即 system()和 strftime()。

函数 systime()主要用于生成写入日志记录中的时间戳。例如，假设我们使用 awk 脚本来响应对 WWW 服务器的 CGI 查询。我们可以将所有的查询写入日志文件。

```
{
...
printf("%s:%s:%d\n", User, Host, systime()) >>
"/var/log/cgi/querylog"
...
}
```

由此产生的记录如下所示：

```
arnold:some.domain.com:831322007
```

使用 strftime()函数（注 6）可以方便地将时间戳转换为易读的日期。所使用格式字符串类似 sprintf()中使用的格式字符串；它由文本和格式说明组成，包括描述日期和时间的不同组成部分的格式说明。

```
$ gawk ' BEGIN { print strftime("Today is %A, %B %d, %Y") }'
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
Today is Sunday, May 05, 1996
```

可用的格式描述列表很长。可参见你本地的 *strftime*(3)帮助页，以及参见 gawk 文档可以获得全部列表内容。我们假设的 CGI 日志文件能够由下面的程序进行处理。

```
# cgiformat --- process CGI logs
# data format is user:host:timestamp
#1
BEGIN { FS = ":"; SUBSEP = "@" }
#2
{
# make data more obvious
  user = $1; host = $2; time = $3
# store first contact by this user
   if (! ((user, host) in first))
        first[user, host] = time
# count contacts
  count[user, host]++
# save last contact
   last[user, host] = time
}
#3
END {
# print the results
  for (contact in count) {
        i = strftime("%y-%m-%d %H:%M", first[contact])
        j = strftime("%y-%m-%d %H:%M", last[contact])
        printf "%s -> %d times between %s and %s\n",
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
                    contact, count[contact], i, j
        }
  }
```

注 6: 这个函数在 ANSI C 中的同名函数之后被模式化

下一步是将 FS 设置为“:”以正确地分割字段。我们使用了一个简洁的技巧并设置下标分割符为“@”。所以这里的数组使用字符串“user@host”为下标。

在第二步，我们先确定是否是第一次看到这些用户。如果是（它们没有在 first 数组中），则将它们添加到 first 数组中。并递增计数器以表示它们连接了多少次。最后将这个记录的时间戳存储到 last 数值中。每当我们看到用户的新的连接时就重新改写这个元素。这样就可以了。结果是将最后（最近的）一个连接存储到数组中。

这里的 END 过程用于格式化数据。它循环访问 count 数组，对 first 和 last 数组的时间戳进行格式化以便打印，假设有包含下面记录的日志文件。

```
$ cat /var/log/cgi/querylog
arnold:some.domain.com:831322007
mary:another.domain.org:831312546
arnold:some.domain.com:831327215
mary:another.domain.org:831346231
arnold:some.domain.com:831324598
```

下面是运行以上程序得到的结果：

```
$ gawk -f cgiformat.awk /var/log/cgi/querylog
mary@another.domain.org -> 2 times between 96-05-05 12:09 and 96-05-05 21:30
arnold@some.domain.com -> 3 times between 96-05-05 14:46 and 96-05-05 15:29
```

Michael 的 awk (mawk)

第三个可免费使用的 awk 是 mawk，是由 Michael Brennan 编写的，这个程序向上兼容 POSIX awk，并且有一些扩展。它是稳定的且运行良好。mawk 的可免费

运维技术书籍免费下载 https://www.linuxprobe.com

使用的源代码可以通过匿名的文件传送协议（FTP）从站点 ftp.whidbey.net 下载。位于目录/pub.brennan/mawk1.3.3.tar.gz 中（当你访问该站点时可能已经出了新的版本）。这是一个 gzip 程序压缩的 tar 文件。可以用“二进制”或“镜像”模式来传输这个文件。

mawk 的主要优点是它的速度和稳定性。尽管它不如 gawk 功能多，但它总比 gawk 优越（注 7）。除了 UNIX，mawk 还可以运行在 MS-DOS 下。

前面介绍的共有的扩展 mawk 中也可使用。

注 7： gawk 的优点是它有更庞大的功能集，已经被移植到更多的非 UNIX 得系统上，并且给出了更多的文档

商业版 awk

awk 还有几个商业版。本节将来看一下我们所了解的一个。

MKS awk

在加拿大安大略的滑铁卢的 Mortin Kern System（MKS）（注 8）将 awk 作为 MS-DOS/windows、OS/2、windows 95 和 Windows NT 系统中 MKS 工具包的一部分。

MKS 版本实现了 POSIX awk。它有下面的扩展：

- 函数 exp()、int()、log()、sqrt()、tolower()和 toupper()在没有给出参数时用$0 代替。
- 附加函数 ord()是可用的。这个函数的参数为一个字符串，返回参数第一个字符的数值型值。它和 Pascal 中的同名函数的功能类似。

Thompson Automation awk（tawk）

Thompson Automation SoftWare（注 9）MS-DOS/Windows、Windows 95、Windows NT 和 Solaris 提供了 awk 的一个版本（tawk）（注 10）。tawk 有几个很有趣的特点。第一，和其他版本的 awk 不同，它们作为解释器，而 tawk 是作为编译器。第二，tawk 提供了用 awk 编写的面向屏幕的调试器。调试器的源代码也包含在其中。第三，tawk 允许将编译后的 和用 C 语言编写的任意函数链接。tawk 已经在 *comp.lang.awk* 新闻组中得到热烈的好评。

运维技术书籍免费下载 https://www.linuxprobe.com

注 8： 地址是 Mortice Kern Systems, 185 Columbia Street West, Waterloo, Ontario N2L 5Z5, Canada. 北美电话是：1-800-265-2797，其他地方电话是：1-519-884-2251。URL 是 http://www.mks.com

注 9： 地址是：Thompson Automation SoftWare, 5616 SW Jefferson, Portland OR 97221 U.S.A，美国电话是：1-800-944-0139，其他各地电话是：1-503-224-1639，

注 10： Michael Brennan 在 mawk(1)的帮助页中指出“AWK 语言的实现着表明当为程序命名时缺乏想象力”。

tawk 提供一个和 POSIX awk 的操作类似的 awk 借口，用于编译和运行程序。然而，你也可以将 编译成一个独立的可执行文件。tawk 编译器实际上是将程序编译成一个压缩中间形式。当运行时，这一中间形式被链接到一个执行该程序的库，且在链接期间将其他 C 例程组合到 awk 程序中。

tawk 是 awk 全面功能的实现。除了实现 POSIX awk（基于新版的 awk）的功能外，它还用一些基本的方式扩展了这个语言，而且用于很多内置函数。

tawk 语言扩展

这部分给出了 tawk 中的新功能的列表。对它们的完整介绍超出了本书的范围。tawk 的文档详细介绍了这些功能。希望现在你对 awk 中将会出现的这些特征的价值已经比较熟悉。我们将在有关地方对比 tawk 和 gawk 中可比较的功能。

- 附加的特殊模式有：INIT、BEGINFILE 和 ENDFILE。INIT 和 BEGIN 类似，但这个过程中的操作比 BEGIN 过程中的操作先执行。BEGINFILE 和 ENDFILE 使你能够对每个文件实施启动和关闭操作。和基于 FNR==1 的规则不同，即使当文件为空这些操作也要执行。

- 可控制的正则表达式。你可以在正则表达式（“/match me/”）中添加一个标志，来告诉 tawk 如何处理正则表达式。标志 i（“/match me/i”）表示匹配时应该忽略大小写。标志 s 表示在匹配时应和最短的文本匹配，而不是最长的。

- 提供一个 abort[*expr*]语句，类似于 exit，除非 tawk 立即退出，否则将绕过任何 END 过程。如果提供了参数 *expr*，则 *expr* 将是 tawk 返回给上级程序的值。

- 真正的多维数组。传统的 awk 通过连接下标的值，模拟多维数组，下标用 SUBSEP 的值来区分，生成常规的关联数组中所希望的唯一下标。为了与这个功能兼容，tawk 提供给了真正的多维数组。

```
a[1][1] = "hello"
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
    a[1][2] = "world"
    for (i in a[1])
      print a[1][i]
```

多维数组确保下标是唯一的，而且当数组中元素的个数很多时，能够有潜力实现更大的功能。

- 数组的自动排序。当使用 for（item in array）结构循环处理数组中的每个元素时，tawk 首先将数组的下标排序，使得数组的元素按一定的顺序被访问。你可以控制是否要排序，是按字母排序还是按数值排序，是按升序还是按降序排序。虽然这种排序会引起性能下降，但比用 awk 代码，或用管道将结果送给 sort 的外部调用进行排序的开销要少。

- 函数和变量的范围控制。可以声明函数和变量在整个程序中是全局的，在“模块（源文件”内是全局的，对一个模块来讲是局部的，或对一个函数来讲是局部的。常规的 awk 只给全部变量，全局函数，以及作为局部变量的额外函数参数。整个特征很好，它使得编写 awk 函数库很容易，而且不用担心变量名与其他函数库或与用户主程序产生冲突。

- RS 可以是一个正则表达式。这和 gawk 和 mawk 类似。然而，正则表达式不能是多于一个字符的向前搜索。和 RS 匹配的文本被存储在与 gawk 中的变量 RT 类似的变量 RSM（记录分隔符匹配）中。

- 字段描述，而不是字段分隔符。变量 FPAT 可以是一个描述字段内容的正则表达式。和 FPAT 成功匹配的连续的文本成为字段的内容。

- 控制隐含文件处理循环。变量 ARGI 可以跟踪当前输入数据文件的 ARGV 中的位置。和 gawk 的 ARGIND 变量不同，为 ARGI 赋一个值就可以使 tawk 跳过输入数据文件。

- 固定长度的记录，通过给变量 RECLEN 赋值，就可以是 tawk 读取固定长度的记录。如果 RS 和 RECLE 内的字符不匹配，那么 tawk 返回一个记录，这个记录是 RECLEN 的字符长度。

- 十六进制的常量。在 tawk 的程序中可以指定 C 风格的十六进制的常量（0xDEAD 和 0xbEEF 是两个典型的十六进制数）。这有助于使用内置的位处理函数（参见后面的内容）。

好了！这是一个相当长的列表，但是这些特征对在 awk 中编写程序提供了补充的能力。

tawk 补充的内置函数

tawk 除了扩展了语言，特提供了大量补充的内置函数。以下是可用的各种类型的函数。每个类型有两个或更多的相关函数。我们将简单描述每个类型的功

运维技术书籍免费下载 https://www.linuxprobe.com

能。

- 扩展字符串函数。标准字符串函数的扩展和新的字符串函数，可用于按模式中的子模式进行匹配和替换（和 gawk 和 gensub()函数相同），而不是使用分隔符来分隔。还有补充的 printf 格式和字符串转换函数。尽管这些函数可用采用用户自定义的函数来编写，但作为内置函数能提供更强的功能。
- 位处理函数。可用用逻辑操作符 AND、OR 和 XOR 对数据（整数）按位操作。这些功能也可以用用户自定义的函数来实现，但性能较差。
- 更多的 I/O 函数。有一组函数模仿了 *stdio*(3)库中的函数。特别是在文件中进行查找以及按固定长度进行 I/O 时，功能更强。
- 目录操作函数。可以建、删除和改变路径、以及删除和重命名文件。
- 文件信息函数。可以获取文件许可权限、大小和更改时间。
- 读取目录函数。可以获得当前时间的函数，并可用各种方式格式化。这些函数没有 gawk 中的 strftime()函数灵活。
- 执行函数。可以暂停特定的时间，而使其他函数运行。tawk 的 spawn()函数允许你为新的程序环境提供数据，以及标识程序是否应该异步运行。在非 UNIX 系统上这是很有用的，因为这些系统上的命令解释器（例如 MS-DOS 的 command.com）功能非常有限。
- 锁定文件。你可以锁定和解锁文件，以及整理文件。
- 屏幕函数。你可以执行面向屏幕的输入输出，在 UNIX 上，这些函数在 *curses*(3)库上实现。
- 打包或解包二进制数据。你可以指定如何设计二进制数据结构。这些和其他新的 I/O 函数结合使用可以实现按二进制形式进行输入输出，其中的一些操作必须在 C 和 C++中处理。
- 访问内部状态。可以通过函数调用来获取或设置任何 awk 变量。
- 反问 MS-DOS 底层功能。可以使用系统中断，获取或指定内存地址。这些功能主要是针对专业人员提供高的。

通过这个列表可以清楚地看出，对于主要的编程任务，tawk 是 C 和 Perl 的一个很好的替代品。例如，屏幕函数和内部状态函数用于实现 awk 中的 tawk 调试器。

运维技术书籍免费下载 https://www.linuxprobe.com

Videsoft VSAwk

Videosoft 出售的软件 VSAwk，将 awk 风格的编程加入到 Visual Basic 环境中。VSAwk 是 Visual Basic 的一个控件，以事件驱动方式工作。和 awk 一样，VSAwk 提供了启动和清除操作，并能将输入记录分解到字段中，还能编写表达式和调用 awk 的内置函数。

VSAwk 的数据访问模式与 UNIX awk 类似，而不是语法类似。然而，有趣的人们将 awk 的概念应用于不同语言提供的环境。

后记

好了，我们已经在 awk 中将编程的输入和输出全部讨论了，包括标准的语言和不同实现工具中可用的扩展。当你使用 awk 时，你会发现它是一个简单并且令人舒服的编程语言，因为它为你完成了几乎所有的苦差事，所以你可以集中精力来解决实际的问题。

运维技术书籍免费下载 https://www.linuxprobe.com

本章内容

- 一个交互式拼写检查器
- 生成格式化索引
- Masterindex 程序的其他细节

第十二章

综合应用

本章介绍了两个复杂的应用，综合了 awk 编程语言的大多数特征。第一个程序 spellcheck 提供了一个 UNIX spell 程序的交互式接口。第二个程序 masterindex 是一个批处理程序，用于为一本或一套书生成索引，即使你对这些特殊的应用不感兴趣，也应该学习这种稍大的程序以对 awk 程序所能解决问题的范围有一个感性认识。

一个交互式拼写检查器

UNIX spell 程序对于捕获一个文档中的拼写错误做了一定的工作。然而对大多数人来说，它只做了一半的工作。它没有帮助你更正拼错的单词。首次使用 spell 的用户发现自己将拼写错误的单词记录下来，然后利用文本编辑器来修改文档。大多数熟练的用户是创建一个 sed 程序来自动进行修改。

Spellcheck 程序提供了另外一种方法：它将 spell 发现的每个单词都显示给你并询问是否修改这个单词。你可以在每次遇到这个单词时进行修改，或者可以一次将所有的拼写错误都改掉。你也可以选择添加任何单词，这些单词 spell 从本地字典文件中找到。

在介绍这个程序之前，让我们示范一下它是如何工作的。用户输入 spellcheck，即一个调用 awk 的 shell 脚本，以及文档文件的名字。

```
$ spellcheck ch00

Use local dict file? (y/n)y
```

如果在命令行指定给出字典文件，而在当前目录下有一个文件 dict，那么将询问用户是否用本地的字典。Spellcheck 用于用本地字典运行 spell 程序。

```
Running spell checker ...
```

用由 spell 找到的有拼写错误的单词列表，spellcheck 可以提醒用户修改它

运维技术书籍免费下载 https://www.linuxprobe.com

们。在显示第一个单词之前，首先显示一个可执行操作的响应列表。

```
Responses:

Change each occurrence,

Global change,

Add to Dict,

Help,

Quit

CR to ignore:

1 - Found SparcStation (C/G/A/H/Q/):a
```

Spell 找到的第一个单词是“SparcStation”。输入响应“a”（后跟一个回车键）表示将这个单词加入列表，该列表用于更新字典。第二个单词完全是拼写错误，输入响应“g”完成全局的修改。

```
2 - Found languauge (C/G/A/H/Q/):g

Globally change to:language

Globally change languauge to language? (y/n):y

> and a full description of its scripting language.

1 lines changed. Save changes? (y/n)y
```

在提示用户输入正确的拼写和确认输入后完成修改，将每个受影响的行显示出来且在前面加上一个“>”。然后在保存修改前询问用户的确认。第三个单词也被添加到字典中。

```
3 - Found nawk (C/G/A/H/Q/):a
```

第四个单词误拼“utilities”。

```
4 - Found utlitities (C/G/A/H/Q/):c
These utlitities have many things in common, including

^^^^^^^^^^
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
    Change to:utilities

    Change utlitities to utilities? (y/n):y

    Two other utlitities that are found on the UNIX system
^^^^^^^^^^

    Change utlitities to utilities? (y/n):y

    >These utilities have many things in common, including

    >Two other utilities that are found on the UNIX system

    2 lines changed. Save changes? (y/n)y
```

用户输入“c”将修改出现的每个错误。这时的响应使用户看到包含拼写错误的行并且进行修改。在用户修改完所有错误后，显示修改的行，用户将被询问是否保存修改。

因为不能确定第五个单词是否拼写错误，所以用户可以输入“c”来观察这行。

```
    5 - Found xvf (C/G/A/H/Q/):c
    tar xvf filename
    ^^^
    Change to:RETURN
```

在确认没有拼错之后，用户键入回车键以忽略这个单词。通常，spell 找出了很多单词但他们并没有拼错，因此输入回车键表示忽略这个单词。

当处理完列表中的所有单词后，或者用户在这之前退出，将提示用户保存修改的文档和字典。

```
    Save corrections in ch00 (y/n)? y

    Make changes to dictionary (y/n)? y
```

如果用户回答“n”，原始的文件和字典都保持不变。

现在我们来看 spellcheck.awk 脚本，它可以分为 4 部分：

- BEGIN 过程，处理命令行参数并执行 spell 命令建立一个单词列表。
- 主过程，一次从列表中读取一个单词并提示用户输入正确的单词。

运维技术书籍免费下载 https://www.linuxprobe.com

- END 过程，保存文件的拷贝，并覆盖原始文件。同时也将单词列表以外的单词扩充当前的字典中。
- 支持函数，调用它用于修改文件。

BEGIN 过程

Spellcheck.awk 的 BEGIN 过程很大，也有些不寻常。

```
 # spellcheck.awk  -  交互式拼写检查程序

#

# 作者: Dale Dougherty

#
# 用法: nawk -f spellcheck.awk [+dict] file

# (用 spellcheck 作为 shell 程序的名字)

# SPELLDICT = "dict"

# SPELLFILE = "file"

# BEGIN 动作执行下列任务:

# 1) 处理命令行参数

# 2) 创建临时文件名

# 3) 执行 spell 程序来创建单词列表文件

# 4) 显示用户响应的列表

BEGIN {

# 处理命令行参数

# 至少两个参数—nawk 和文件名

if (ARGC > 1) {

# 如果有两个以上的参数，那么第二个参数是 dict
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
if (ARGC > 2) {
# 测试 dict 是否用“+”指定
# 并将 ARGV[1] 赋给 SPELLDICT
if (ARGV[1] ~ /^\+.*/)
SPELLDICT = ARGV[1]
else
SPELLDICT = "+" ARGV[1]
# 将 ARGV[2] 赋给 SPELLFILE
SPELLFILE = ARGV[2]
# 删除 args 以便 awk 不会将它当作文件打开
delete ARGV[1]
delete ARGV[2]
}
# 不多于两个参数
else {
# 将 ARGV[1] 赋给 SPELLFILE
SPELLFILE = ARGV[1]
# 测试本地 dict 文件是否存在
if (! system ("test -r dict")) {
# 如果存在，询问是否应该使用它
printf ("Use local dict
getline reply < "-"
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
# 如果回答是，使用"dict"

if (reply ~ /[yY](es)?/

SPELLDICT = "+d

}

}

}

} # 如果参数大于-1 > 1，处理结束

# 如果参数不大于 1，那么打印 shell 命令用法
else {

print "Usage: spellcheck [+dict] file"

exit 1

}

# 处理命令行参数结束
# 创建临时文件名，每个都以 sp_开始

wordlist = "sp_wordlist"

spellsource = "sp_input"

spellout = "sp_out"

# 将 SPELLFILE 复制到临时输入文件

system("cp " SPELLFILE " " spellsource)

# 现在运行拼写程序，输出发送到单词列表

print "Running spell checker ..."

if (SPELLDICT)

SPELLCMD = "spell " SPELLDICT " "
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
else
SPELLCMD = "spell "
system(SPELLCMD spellsource " > " wordlist )
# 测试单词列表，看看是否发生了拼写错误的单词
if ( system("test -s " wordlist ) ) {
# 如果单词列表为空（或拼写命令失败），退出
print "No misspelled words found."
system("rm " spellsource " " wordlist)
exit
}
# 给 ARGV[1] 赋单词列表，以便 awk 读取它。
ARGV[1] = wordlist
# 显示用户响应的列表
responseList = "Responses: \n\tChange each occurrence
responseList = responseList "\n\tGlobal change,"
responseList = responseList "\n\tAdd to Dict,"
responseList = responseList "\n\tHelp,"
responseList = responseList "\n\tQuit"
responseList = responseList "\n\tCR to ignore: "
printf("%s", responseList)
} # BEGIN 过程结束
```

BEGIN 过程的第一部分处理命令行参数。它检测如果“ARGC”比 1 大，则程序继续。也就是说，除了“nawk”之外，必须给出文件名。这个文件名指定了

运维技术书籍免费下载 https://www.linuxprobe.com

spell
要分析的文件。一个可选的字典文件名可以被作为第二个参数。尽管
spellcheck
命令并不支持 spell 的任何隐含选项，但 spellcheck 程序遵循 spell 的命令行接口。如果没有给出字典，那么程序执行 test 命令以确定文件 dict 是否存在。
如果存在，提示用户确定利用这个文件作为字典文件。

一旦处理了这个参数。我们将从 ARGV 数组中删除它们，这将防止它们被解释成文件名参数。

BEGIN 过程的第二部分建立了一些临时文件，因为我们不想直接在原始文件上工
作。在这个程序的末尾，用户将选择保存或放弃在临时文件上所做的工作。临时
文件都是以“sp_”开始且在退出程序前被删除。

这个过程的第三部分执行 spell 程序并建立一个单词列表。我们测试这个文件是
否存在访问之前确保文件中包含内容。如果由于一些原因 spell 程序失败了，或
者没有发现拼错的单词，文件 wordlist 将是空的。如果这个文件存在，则将这个文件名作为 ARGV 数组的第二个元素。这是一个不常用但可行的为 awk 将要访问的输入文件提供文件名的办法。注意，当 awk 被调用时，这个文件并不存在！
在命令行中指定文件名的办法，不再存在于 ARGV 数组中。我们不利用 awk 的主输入循环来读取这个文档文件，而且利用一个 while 循环来读取文件以发现并更
正拼写错误。

BEGIN 过程的最后一部分的任务是当显示一个拼错的单词时，定义和显示用户能做出的响应的列表。这个列表在程序开始运行时显示一次，以及当用户在主菜单输入“help”时显示。将这个列表赋给一个变量，可以使我们必要时在程序的不同点访问它，以避免重复。对 responseList 的赋值可以更容易些，但所有字符串太长，在这本书中不能显示（不能将一个字符串分开两行）。

主过程

这个主过程很小，仅显示一个拼错的单词和提示用户输入适当的回答。这个过程

运维技术书籍免费下载 https://www.linuxprobe.com

对每个拼错的单词都执行。

这个过程短小的一个原因是因为中心操作（更正拼错的单词），它们由两个较大
的用户自定义函数来处理，我们将在最后一部分看到它们。

```
# 主过程，对单词列表中的每一行都执行

# 目的是显示拼写错误的单词并提示用户

# 适当的动作

{

# 将单词赋给 misspelling

misspelling = $1

response = 1

++word
# 打印拼错的单词并提示响应

while (response !~ /(^[cCgGaAhHqQ])|^$/ ) {

printf("\n%d - Found %s (C/G/A/H/Q/):", word, misspelling)

getline response < "-"

}

# 现在处理用户的响应

# CR - 回车忽略当前的单词

# 帮助

if (response ~ /[Hh](elp)?/) {

# 再次显示响应列表和提示

printf("%s", responseList)

printf("\n%d - Found %s (C/G/A/Q/):", word, misspelling)
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
getline response < "-"

}

# 退出

if (response ~ /[Qq](uit)?/) exit

# 添加到字典
if ( response ~ /[Aa](dd)?/) {

dict[++dictEntry] = misspelling

}

# 改变每次出现

if ( response ~ /[cC](hange)?/) {

# 读取正在纠正的文件的每一行
newspelling = ""; changes = ""

while( (getline < spellsource) > 0){

# 调用显示拼错单词的行的函数

#并提示用户进行每个校正

make_change($0)

# 所有的行都转到临时的输出文件

print > spellout

}

# 读完所有的行

# 结束临时输入和临时输出文件

close(spellout)

close(spellsource)
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
# 如果做了改变
if (changes){
# 显示被改变的行
for (j = 1; j <= changes; ++j)
print changedLines[j]
printf ("%d lines changed. ", changes)
# 在保存之前给出确认的函数
confirm_changes()
}
 }
# 全局改变
if ( response ~ /[gG](lobal)?/) {
# 调用提示校正的函数
# 并显示被改变的每一行
# 在保存之前询问用户确认所有的改变
make_global_change()
}
} # 主过程结束
```

Wordlist 中的每个输入行的第一个字段，包含着拼错的单词并被赋给 misspelling。我们构造了一个 while 循环，在其中我们将拼错的单词显示给用户并提示用户作出响应。仔细观察下面的正则表达式，它用于测试 response 的值：

```
while (response !~ /(^[cCgGaAhHqQ])|^$/)
```


运维技术书籍免费下载 https://www.linuxprobe.com

用户只能通过输入任意指定的字符或键入回车键（一个空行）退出的这个循环。
利用正则表达式测试用户的输入有助于编写简单灵活的程序。用户可以输入一个小写或大写字母 c 或以“c”开头的单词，例如“Change”。

主过程的余下的部分由条件语句组成，用于测试用户指定的响应并执行相应的操作。第一个响应是“help”，作用是再次显示响应列表和重新显示提示。

下一个响应是“quit”。和 quit 相关的操作是 exit，用于退出主过程并转到 END 过程。

如果用户输入“add”，拼错的单词将被加入到数组 dict 中，并将被作为一个例外添加到本地字典中。

“Change”和“Global”响应使程序真正开始工作。理解它们的区别很重要。当用户输入“c”或“change”时，将显示文档中遇到的第一个拼错的单词。然后提示用户做修改。这将在文档中每个出现错误的地方发生。当用户输入“g”或“global”时，提示用户立即修改，并且将一次修改所有的错误，且没有提示用户确认每个修改。这个工作主要由两个函数来处理，make_change()和 make_global_change(),这两个函数我们将在最后一部分看到。除了一种情况外，这些都是有效的响应。回车表示忽略拼错的单词并得到列表中的下一个单词。这是主输入循环的默认操作，因此不需要为它设置条件。

END 过程

当然，下列情况之一将进入 END 过程：
•spell 命令失败或没有发现任何拼错的单词。
•拼错单词的列表已取尽。
•用户输入“quit”作为提示的响应。

END 过程的目的是允许用户确认对文档或字典的任何永久性修改。

```
# END 过程生成永久的改变
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
# 他改写原始文件，并向字典中
# 添加单词
# 它还删除临时文件
END {
# 如果读取一条记录后到达这里没有产生
# 改变，因此退出
if (NR <= 1) exit
# 用户必须确认保存对文件的校正
while (saveAnswer !~ /([yY](es)?)|([nN]o?)/ ) {
printf "Save corrections in %s (y/n)? ", SPELLFILE
getline saveAnswer < "-"
}
# 如果答案是肯定的，那么 mv 临时输入文件为 SPELLFILE
# 保存旧的 SPELLFILE，以防万一
if (saveAnswer ~ /^[yY]/) {
system("cp " SPELLFILE " " SPELLFILE ".orig")
system("mv " spellsource " " SPELLFILE)
}
# 如果答案是否定的，那么 rm 临时输入文件
if (saveAnswer ~ /^[nN]/)
system("rm " spellsource)
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
# 如果单词已经添加到字典数组中，那么提示确定保存在当前字典中
if (dictEntry) {
printf "Make changes to dictionary (y/n)? "
getline response < "-"
if (response ~ /^[yY]/){
# 如果没有定义字典，那么使用 "dict"
if (! SPELLDICT) SPELLDICT = "dict"
# 循环使用数组并将单词添加到字典中
sub(/^\+/, "", SPELLDICT)
for ( item in dict )
print dict[item] >> SPELLDICT
close(SPELLDICT)
# 排序字典文件
system("sort " SPELLDICT "> tmp_dict")
system("mv " "tmp_dict " SPELLDICT)
}
}
# 删除单词列表
system("rm sp_wordlist")
 } # END 过程结束
```

END 过程以一个条件语句开始来测试记录的个数是否小于或等于 1。当 spell 程序没有产生单词列表或当用户看到第一个记录之后输入“quit”时将产生这种情况。如果产生这种情况，END 过程将当作没有可保存的工作而退出。接着，我

运维技术书籍免费下载 https://www.linuxprobe.com

们
创建一个 while 循环来询问用户将所做的修改保存到文档。这需要用户对提示回
答“y”或“n”。如果回答是“y”，临时输入文件将代替原始文档文件。如果回
答的是“n”，临时文件被删除。不接受其他的响应。

下一步，我们测试数组 dict 中是否有内容。它的元素是要添加到字典中的单词。
如果用户同意将它们加入到字典中，这些单词将添加到上面所定义的当前字典中，如果不同意，则添加到本地 dict 文件中。因为被 spell 读取的字典必须排序，因此将被执行一个 sort 命令对送到临时文件的输出进行排序，这个临时文件将在后面的处理中覆盖原文件。

支持函数

这里有 3 个支持函数，其中两个很大，用于完成文档的大多数修改工作。第三个
函数用于确定用户想要保存的所做的修改。

当用户想在文档中“改变每个错误”时，主过程利用一个 while 循环每次读取文
档的一行（这行成为$0）。通过调用 make_change()函数来确定这行中是否包含
拼错的单词。如果有，显示这行并提示用户输入相应单词的正确拼写。

```
# make_change - 提示用户纠正当前输入

#                                    行的错误拼写

#                                 调用它本身找到字符串中的其他出现

# stringToChange - 初始为 $0; 然后是 $0 中不匹配的字串

# len - 从$0 的开始处到被匹配的字符串末尾的长度

# 假设定义了拼写错误.

function make_change (stringToChange, len, # 参数

line, OKmakechange, printstring, carets) # 局部变量
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
{
# 匹配 misspelling 与 stringToChange; 不匹配则不做任何事情
if ( match(stringToChange, misspelling) ) {
# 显示被匹配的行
printstring = $0
gsub(/\t/, " ", printstring)
print printstring
carets = "^"
for (i = 1; i < RLENGTH; ++i)
carets = carets "^"
if (len)
FMT = "%" len+RSTART+RLENGTH-2 "s\n"
else
FMT = "%" RSTART+RLENGTH-1 "s\n"
printf(FMT, carets)
# 如果还没有定义，提示用户校正
if (! newspelling) {
printf "Change to:"
getline newspelling < "-"
}
# 回车则跳过
# 如果用户输入校正，确认
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
while (newspelling && ! OKmakechange) {
printf ("Change %s to %s? (y/n):", misspelling, newspelling)
getline OKmakechange < "-"
madechg = ""
# 测试响应
if (OKmakechange ~ /[yY](es)?/ ) {
# 做修改 (只在第一次遇到时)
madechg = sub(misspelling, newspelling, stringToChange)
}
else if ( OKmakechange ~ /[nN]o?/ ) {
# 提供重新更正的机会
printf "Change to:"
getline newspelling < "-"
OKmakechange = ""
}
} #while 循环结束
# 如果 len 为真, 处理$0 的子串
if (len) {
# 对它进行汇编
line = substr($0,1,len-1)
$0 = line stringToChange
}
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
else {
$0 = stringToChange
if (madechg) ++changes
}
# 将改变的行插入要显示的数组
if (madechg)
changedLines[changes] = ">" $0
# 创建子串，这样可以试图匹配其他的出现
len += RSTART + RLENGTH
part1 = substr($0, 1, len-1)
part2 = substr($0, len)
# 调用本身来看看剩余的部分中是否有错误
make_change(part2, len)
} # if 语句结束
} # 函数 make_change()结束
```

如果在当前行没有找到拼错的单词，什么都不做。如果找到了，这个函数显示包
含拼错的单词的行并询问用户是否要改正。在显示当前行的下面有一行^符号用于标识拼错的单词。

```
Two other utlitities that are found on the UNIX system
        ^^^^^^^^^^
```

当前输入行被复制到 printstring 中，因为有必要改变这行以便显示。如果这个行包含一些制表符，在这个行的副本中每个制表符都用一个空格代替。这就解决了当制表符出现时如何对齐^符号的问题（当计算一行的长度时，一个制表符作为一个单独的字符计算，但实际上它显示时占用了更多的空格，通常是 5

运维技术书籍免费下载 https://www.linuxprobe.com

到 8 个字符长）。

当显示出这行后，函数提示用户输入正确的单词。然后接着显示用户输入的内容并请求确认。如果确认是正确的单词，则调用 sub（）函数来进行修改。如果没有确认。将给用户另外一个机会输入正确的单词。

记住，sub（）函数只修改一行中第一次出现的单词。而 gsub（）函数修改一行中出现的所有单词，但我们想让用户对每次修改做出确认。因此，我们不得不提取余下的部分中拼错的单词。而且我们不得不提取下一个出现的单词，而不管第一次出现的单词是否被修改。
为了完成这些功能，将函数 make_change()设计成递归函数，它调用自己以在同一个行上查找符合条件的其他单词。换句话说，当 make_change()第一次被调用时，它查找整个$0 并提取这一行中第一个拼错的单词。然后它将这个行分为两部分——第一个部分包含到第一次出现的单词末尾的所有字符，第二部分包含这行的其他所有的字符。然后它调用自己来提取第二部分中的拼错的单词。当递归调用时，这个函数使用了两个参数。

```
make_change(part2, len)
```

第一个参数是要修改的字符串，当从主过程中调用时，它初始为$0，但以后各次都是$0 余下的部分。第二个参数的 len 或第一部分的长度，我们可以利用它来提取子串并在最后将两个部分重新组合在一起。

函数 make_change（）还用于将被改变的行收集到一个数组中。

```
# 将被改变的行放入数组中以便显示

if (madechg)

changedLines[changes] = ">" $0
```

如果函数 sub（）执行成功，变量 madechg 将得到一个值。$0（两个部分已经重新合并在一起）被赋给数组的一个元素。当读取文档中的所有行之后，主过程循环访问这个数组并显示所有被修改的行。然后调用函数 confirm_changes（）询问是否保存这些修改。它将用临时输出文件覆盖临时输入文件，保持修改了拼错单词后的完整结果。

如果一个用户决定做“全局修改”，则调用函数 make_global_change()来完成。这个函数和 make_change()函数类似，但是更简单，因为我们可以实现对每行的全局修改。

运维技术书籍免费下载 https://www.linuxprobe.com

```
# make_global_change --
#                          提示用户全局校正所有行
#                          的拼写错误
#                          没有参数
# 假设定义了拼写错误
function make_global_change( newspelling, OKmakechange, changes)
{
# 提示用户校正拼写错误的单词
printf "Globally change to:"
getline newspelling < "-"
# 回车则跳过
# 如果回答一个问答，确认
while (newspelling && ! OKmakechange) {
printf ("Globally change %s to %s? (y/n):", misspelling,
newspelling)
getline OKmakechange < "-"
# 测试响应并生成改变
if (OKmakechange ~ /[yY](es)?/ ) {
# 打开文件，读取所有的行
while( (getline < spellsource) > 0){
# 如果找到匹配，利用 gsub 进行改变
# 打印每个被改变的行
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
if ($0 ~ misspelling) {
 madechg = gsub(misspelling, newspelling)
 print ">", $0
 changes += 1 # 统计改变的行
 }
 # 将所有的行写到临时输出文件
 print > spellout
 } # 读取文件的 while 循环结束
 # 关闭临时文件
 close(spellout)
 close(spellsource)
 # 报告改变的数量
 printf ("%d lines changed. ", changes)
 # 调用在保存改变之前确认的函数
 confirm_changes()
 } # 条件语句 (OKmakechange ~-y)结束
 # 如果校正没有确认，提示新的单词
 else if ( OKmakechange ~ /[nN]o?/ ){
 printf "Globally change to:"
 getline newspelling < "-"
 OKmakechange = ""
 }
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
} # 提示用户校正的 while 循环结束

} #函数 make_global_change()结束
```

这个函数提示用户输入正确的单词。设置 while 循环用于读取文档的所有行并使用函数 gsub（）实现修改。主要区别是所有的修改一次完成，没有提示用户对它们确认。当所有的行被读取后，函数显示已经修改的行并在保存它们之前调用函数 confirm_changes（）得到用户对这些修改的确认。

函数 con□r m_changes( ) 被 make_change( ) 和 make_global_change( ) 函数调用，用来对这两个函数产生的修改进行确认。

```
# confirm_changes --

# 在保存更改之前确认

function confirm_changes( savechanges) {

# 提示确认保存更改

while (! savechanges ) {

printf ("Save changes? (y/n)")

getline savechanges < "-"

}

# 如果确认，mv 输出到输入

if (savechanges ~ /[yY](es)?/)

system("mv " spellout " " spellsource)

}
```

设计这个函数的原因是为了放置代码重复。它的目的就是当用文档文件的新版本（spellout）替换老版本（spellsource）时通知用户这些变化。

运维技术书籍免费下载 https://www.linuxprobe.com

Spellcheck 的 shell 脚本

为了使调用以上的 awk 脚本更容易，我们创建了 spellcheck 的 shell 脚本（据说可以快 3 倍）。它包含下面的程序行。

```
AWKLIB=/usr/local/awklib

nawk -f $AWKLIB/spellcheck.awk $*
```

这个程序设置了一个 shell 变量 AWKLIB，用于指定 spellcheck.awk 脚本的位置。符号“$*”将扩展跟在脚本名后的所有命令行参数。这样，这些参数就可以在 awk 中使用了。

这个拼写检查器的一个有趣的特点是这个 shell 脚本非常简短（注 1）。所有工作由 awk 编程语言完成，包括执行 10 个 UNIX 命令。我们在 awk 中利用一致的语法和相同的结构来处理。当一部分工作在 shell 中完成而另一部分工作在 awk 中完成时，可能会引起混淆。例如,你必须注意 if 条件在语法上的区别和如何引用变量。Awk 的现代版本提供了可代替的 shell 执行命令及与用户连接的功能。Spellcheck.awk 的完整列表可以在附录三“第十二章的补充”中找到。

生成格式化索引

生成一个索引的操作一般包括 3 步：

•对文档中的索引条目进行编码。
•格式化文档，用页码生成索引条目。
•处理索引条目，对它们进行分类，对知识页码不同的条目进行组合，然后准备格式化索引。

无论是使用 troff，或者其他编码批格式处理器，或者 WYSIWYG 格式处理器（如 FrameMaker）,处理起来都非常相似，尽管使用后者处理步骤允许没有那么直观。我们将描述如何用 troff 来生成和本书的索引相类似的索引。我们用下面的宏来生成索引。

宏	描述
.XX	生成普通的索引条目
.XN	创建“see”或“see also”交叉引用
.XB	创建以粗体表示的页条目，以突出时主引用
.XS	生成页码范围的开始位置
.XE	生成页码范围的终止位置

注 1： UNIX 的文字处理（Dougherty and O’Reilly, Howard W. Sams, 1987)给出一个基于 sed 的拼写检查器。它主要是基于 shell 的。将这两个版本比较是很有趣的.

运维技术书籍免费下载 https://www.linuxprobe.com

这些宏需要一个用引号括住的变量，他可以有几个形式，表示为主，次，或第三关键字。

“primary [: secondary [; tertiary]]”

利用冒号作为主关键字和次关键字的分隔符。为了支持早期的编码习惯，如果没有用冒号可以将第一个逗号解释为分隔符。分号表示存在第三个关键字。页码总是和最后一个关键字相关。

这里有一个只有主关键字的记录：

```
.XX "XView"
```

下面两个给出了次关键字：

```
.XX "XView: reserved names"

.XX "XView, packages"
```

最复杂的条目包括 3 个关键字

```
.XX "XView: objects; list"

.XX "XView: objects; hierarchy of"
```

最后，有两种类型的交叉引用：

```
.XN "error recovery: (see error handling)"

.XX "mh mailer: (see also xmh mailer)"
```

这里的“see”条目知识人们查找另一个索引条目。而“see also”也常用于当条目存在的情况下，在这个例子中，是“mh mailer”，但在另一个名字下面有编目原则的相关信息。只有“查看”项目没有相关的页码。

当文档用 troff 处理时，将产生下面的索引条目：

```
XView  42

XView: reserved names 43

XView, packages 43
```


运维技术书籍免费下载 https://www.linuxprobe.com

XView: objects; list of 43

XView: objects; hierarchy of 44

XView, packages 45

error recovery: (See error handling)

mh mailer: (see also xmh mailer) 46

这些条目被作为索引程序的输入。每个条目（除了“see”条目外）都由关键字和一个页码组成。也就是说，每个条目分成两个部分，第一部分的关键字还可以分成 3 个部分。当索引程序处理这些条目且对输出进行格式化时，“Xview”条目被按下面的方式组合：

XView, 42

objects; hierarchy of, 44;

list of, 43

packages, 43,45

reserved names, 43

要完成这些功能，索引程序必须：
•按关键字和页码对索引排序。
•合并只有页码不同的条目。
•合并有相同主关键字和/或子关键字的条目。
•寻找连续的页码并组合为一个范围。
•格式化索引以便显示在屏幕上或打印。

如果对一本书的索引条目进行处理，这就是索引程序将要做的工作。也可以建立一个主索引，即一套书的总索引。要完成这一功能，awk 脚本应该在页码后面增加一个罗马数字或缩写。每个文件包含一本书的条目而这些条目是唯一标识的。如果我们选择用罗马数字来标识卷，那么上面的条目将被修改为：

XView 42:I

XView: reserved names 43:I

XView: objects; list of 43:I

运维技术书籍免费下载 https://www.linuxprobe.com

对于多卷条目，最后生成的索引将是这样：

XView, I:42; II:55,69,75

objects; hierarchy of, I:44;

list of, I:43; II: 56

packages, I:43,45

reserved names, I:43

现在唯一重要的是，要知道作为 awk 程序输入的索引条目，是带有一个页码还是带有页码和卷标识符的。

主索引程序

因为该索引应用程序较长且较复杂，我们描述了大的程序结构。利用程序的注释来解释理解程序的每行起什么作用。

在描述了程序的每个模块之后，最后一个部分讨论剩余的几个细节。这些代码段主要用于处理编码过程中本质的与输入有关的问题。Shell 脚本 masterindex 允许用户指定一系列不同的命令行选项来定义生成什么类型的索引，并调用必须的 awk 程序来完成相应的工作。Masterindex 程序的操作被分成五个单独的程序或模块并形成一个管道。

```
input.idx | sort | pagenums.idx | combine.idx | format.idx
```

所有的这些程序除了一个外都是用 awk 编写的。要对索引条目排序，我们使用标准 UNIX 实用工具 sort 来完成。下面对每个程序的功能做了简要的概括：

input.idx

将题目的格式标准化并以此输入它们。

sort

根据关键字、卷和页码对条目进行排序。

pagenums.idx

合并有相同关键字的项、建立一个页码列表。

combine.idx

运维技术书籍免费下载 https://www.linuxprobe.com

将连续的页码组合成一个范围。

format.idx

准备格式化索引，以便显示或由 troff 处理。

我们将用单独的节来讨论每一步。

标准化输入

lpput.idx 脚本寻找不同类型的条目，并将它们标准化以便于后续的程序访问。另外，它自动使索引条目包含一个（~）符号（参阅本章后面的内容“互换两部分”）。

lpput.idx 程序的输入有两个用制表符分隔的字段组成（在前面介绍过）。这个程序产生的输出是由冒号分隔的 3 个字段。第一字段包含主关键字，第二个字段包含次关键字和第三关键字（如果给出）；第三个字段包含页码。

下面是 input.idx 程序的代码：

```
#!/work/bin/nawk -f

# ------------------------------------------------

# input.idx  -  在排序之前标准化输入

# Author:  Dale Dougherty

# Version 1.1 7/10/90

#

# 输入是"条目" 制表符 "页码"

# ------------------------------------------------

BEGIN { FS = "\t"; OFS = "" }

#1 匹配包含一个~的需要交换的条目

# $1 ~- /~~[^^-~]/  # 该正则表达式不工作，不知为什么

$1 ~~ /~~/ && $1 !--~ /~~~~/ {
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
# 将第一字段分解到名为 subfield 的数组中
n = split($1, subfield, "~")
if (n == 2) {
#  打印没有“-”的条目，然后交换
printf("%s %s::%s\n", subfield[1], subfield[2], $2)
printf("%s:%s:%s\n", subfield[2], subfield[1], $2)
}
next
}# 1 结束
#2 匹配包好两个~
$1 ~~ /~~~~/ {
# replace “~“ with “~~“
gsub(/~~~/, "~—", $1)
} # 2 结束
#3  用 "::" 匹配字面上的":"条目
$1 ~ /::/ {
# 将 "::"替换成八进制值
gsub(/::/, "\\72", $1)
}# 3 结束
#4 清楚条目
{
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
# 寻找第二个冒号，可用于代替 ";"
if (sub(/:.*:/, "&;", $1)) {
sub(/:;/, ";", $1)
}
# 如果冒号后面有内容就删除空格
sub(/: */, ":", $1)
# 如果逗号用做分隔符，将它转换成冒号.
if ( $1 !~ /:/ ) {
# 对于 also 和 see,在 "("前放置分隔符
if ($1 ~ /\([sS]ee/) {
if (sub(/, *.*\(/, ":&", $1))
sub(/:, */, ":", $1)
else
sub(/  *\(/, ":(", $1)
}
else { # 否则只查找逗号
sub(/, */, ":", $1)
}
}
else {
# 在"See"中插入分号
if ($1 ~ /:[^;]+ *\([sS]ee/)
sub(/  *\(/, ";(", $1)
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
}
}# 4 结束
#5 匹配 See Alsos 并且为最后排序做准备
$1 ~ / *\([Ss]ee +[Aa]lso/ {
# 增加"~zz" 用以排序
sub(/\([Ss]ee +[Aa]lso/, "~zz(see also", $1)
if ($1 ~ /:[^; ]+ *~zz/) {
sub(/ *~zz/, "; ~zz", $1)
 }
 # 如果没有页码
if ($2 == "") {
print $0 ":"
next
}
else {
# 输出两个条目
# 打印 See Also 条目的 w/out 页码
print $1 ":"
# 删除 See Also
sub(/ *~zz\(see also.*$/, "", $1)
sub(/;/, "", $1)
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
# 打印普通条目
if ( $1 ~ /:/ )
print $1 ":" $2
else
print $1 "::" $2
next
}
}# 5 结束
#6 处理没有页码的条目 (See 条目)
(NF == 1 || $2 == "" || $1 ~ /\([sS]ee/) {
# 如果是 "See" 条目
if ( $1 ~ /\([sS]ee/ ) {
if ( $1 ~ /:/ )
print $1 ":"
else
print $1 ":"
next
}
else {#如果是 See 条目, 则产生一个错误
printerr("No page number")
next
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
    }

    }# 6 结束
    #7 如果用冒号作为分隔符

    $1 ~~ /:/ {

    # 输出#条目：页面

    print $1 ":" $2

    next

    }# 7 结束

    #8  以主关键字匹配条目

    {

    print $1 "::" $2

    }# 8 结束

    # 支持函数

    #

    # printerr - 打印错误消息和当前记录

    # Arg: 将要显示的消息

    function printerr (message) {

    # 打印消息，记录号和记录

    printf("ERROR:%s (%d) %s\n", message, NR, $0) > "/dev/tty"

}
```

这个脚本有许多模式匹配规则用于识别不同类型的输入。注意，每个条目都可以和多个规则匹配，除非和某个规则相关的操作调用了 next 语句。

讨论这个脚本是，我们将用数字表示每个规则。规则 1 交换包含一个代字符（~）的条目并产生连个输出记录。函数 split（）生成一个数组，命名为

运维技术书籍免费下载 https://www.linuxprobe.com

subfield，其中包含组合条目的两部分。这两个部分按它们原始的顺序被打印出来，然后被交换顺序并成为第二个输出记录，在这个记录中次关键字变成主关键字。

因为我们将代字符作为特殊字符，所以我们必须提供某种方法来输入这个字符。我们利用这个约定，即将两个连续的代字符翻译成一个代字符。规则 2 用来处理这种情况，但注意规则 1 确保了它所匹配的第一代字符后面不是另一个代字符。

这个脚本中的规则 1 和规则 2 的顺序很重要。只有当过程将条目交换后，才能用“~”来代替“~~”。
规则 3 和规则 2 所做的工作类似。可以用“：：”来输出索引中的一个“：”符号。然而，因为我们把冒号作为输入分隔符将输入输送到程序，我们不允许它出现在最后输出的条目中。因此我们用八进制来表示冒号的 ASCII（format.idx 程序将反转这个替换。）

从规则 4 开始，我们试图识别不同的编码条目——为用户提供更大的灵活性。然而，为了使程序编写更容易，我们必须减少到几个基本形式。

在“基本”的语法中，主关键字和此关键字用冒号分隔。次关键字和第三关键字用分号分隔。而且这个程序也允许用冒号代替分号，作为次关键字和第三关键字定界符。如果没有将冒号指定为定界符，那么也可以用逗号作为主关键字和第二关键字定界符（这和以前利用逗号作为定界符的程序是兼容的）。函数 sub（）查找行中的第一个逗号并将它改为冒号。这个规则也用来标准化“see”和“see also”条目。对于冒号分隔的条目，规则 4 将删除冒号后面的空格。所有的这些工作都是由函数 sub（）完成的。

规则 5 处理“see also”条目。我们将字符串“~zz”附加到“see also”条目，因此通过排序它们将被排在此关键字列表的末端。Pagenums.idx 程序在管道的后端，在条目被排序后将删除“~zz”。

规则 6 匹配没有指定页码的条目。唯一没有页码的有效条目包含一个“see”引用。这个规则输出“see”条目且在末端添加“：”，用以表示第三个字段为空。所有的其他条目由函数 printerr（）生成一个出错的信息。这个函数将提示用户特殊的条目没有包含页码，故将不能包含在输出中。这是一个标准化输入的方法——不接受不能正确解释的项。但是，通知用户使他或她能更正条目是很重要的。

运维技术书籍免费下载 https://www.linuxprobe.com

规则 7 输出包含冒号定界符的条目。使用 next 可以避免到达规则 8。

最后，规则 8 匹配只包含一个主关键字的条目。换句话说，也就是没有定界符的条目。我们输出“：：”来表示第二个字段为空。

下面是 test 文件内容的一部分。我们将利用它来生成本节的例子：

```
$ cat test

XView: programs; initialization 45

XV_INIT_ARGS~macro  46

Xv_object~type  49

Xv_singlecolor~type  80

graphics: (see also server image)

graphics, XView model 83

X Window System: events 84

graphics, CANVAS_X_PAINT_WINDOW 86

X Window System, X Window ID for paint window 87

toolkit (See X Window System).

graphics: (see also server image)

Xlib, repainting canvas 88

Xlib.h~header file 89
```

当我们用 input.idx 来运行这个文件时，将产生如下结果：

```
$ input.idx test

XView:programs; initialization:45
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
XV_INIT_ARGS macro::46

macro:XV_INIT_ARGS:46

Xv_object type::49

type:Xv_object:49

Xv_singlecolor type::80

type:Xv_singlecolor:80

graphics:~zz(see also server image):

graphics:XView model:83

X Window System:events:84

graphics:CANVAS_X_PAINT_WINDOW:86

X Window System:X Window ID for paint window:87

graphics:~zz(see also server image):

Xlib:repainting canvas:88

Xlib.h header file::89

header file:Xlib.h:89
```

每个项包含 3 个用冒号分隔的字段。在样本输出中，你可以找到只有一个主关键字的条目，包含主关键字和此关键字的条目，以及那些包含主、次和第三关键字的条目。也可以找到被互换的条目、重复的条目和“see also”条目。

多卷的条目在输出中的唯一区别是每个输出条目带有包含卷标示符的第四个字段。

对条目排序

现在准备对 input.idx 产生的输出进行排序。对条目排序最简单的方法是利用标准的 UNIX sort 程序而不是编写一个自定义脚本。除了排序外，我们还希望删除所有的重复条目，这里使用 uniq 程序来完成这个工作。

运维技术书籍免费下载 https://www.linuxprobe.com

下面是使用的命令行：

```
sort -bdf -t: +0 -1 +1 -2 +3 -4 +2n -3n | uniq
```

可以看出，我们为 sort 命令设置一些选项。第一个选线为-b，表示忽略前导的空格。选项-d 表示按字典排序且忽略符号和特殊字符。选项-f 表示将小写字母和大写字母叠放在一起；换句话说，在排序中它们将被看做是同一个字符。下一个参数-t 可能是最重要的，它告诉程序在排序关键字时用冒号作为字段定界符。选项“+”表示从行开始处跳过的字段数。因此，我们用“+0”来表示排序主关键字为第一个字段。同样，用“-”表示排序关键字的末端。“-1”表示排序主关键字为第一个字段的末端，或第二个字段的开始。第二个字段是此关键字。如果第四个字段（+3）存在，表示卷号。排序的最后一个关键字是页码，这需要使用数值型排序（如果我们没有告诉 sort 关键字是数值型的，那么数字 1 后面将跟随 10，而不是 2）。注意，我们在按卷号排序后才对页码进行排序。因此，第 1 卷的所有页码都被按顺序排列，然后才排到第 2 卷的页码。最后，我们将输出结果由管道输送给 uniq 来删除相同的条目。处理来自 input.idx 的输出，命令 sort 的结果为：

```
graphics:CANVAS_X_PAINT_WINDOW:86

graphics:XView model:83

graphics:~zz(see also server image):

header file:Xlib.h:89

macro:XV_INIT_ARGS:46

toolkit:(See X Window System).:

type:Xv_object:49

type:Xv_singlecolor:80

X Window System:events:84

X Window System:X Window ID for paint window:87

Xlib:repainting canvas:88

Xlib.h header file::89
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
XView:programs; initialization:45

XV_INIT_ARGS macro::46

Xv_object type::49

Xv_singlecolor type::80
```

处理页码

程序 pagenums.idx 用于查找只有页码不同的条目，并为每个条目生成一个页码列表。这个程序的输入是用冒号分隔的 4 个字段：

PRIMARY:SECONDARY:PAGE:VOLUME

第四个域是可选的。现在，我们只考虑单本书的索引，因此没有卷号。注意现在条目已经排好序了。

这个程序的核心是将当前与前面的条目相比较并决定输出的那一个。实现比较的条件可以提取出来并表示成如下的伪代码：

```
 PRIMARY = $1

SECONDARY = $2

PAGE = $3

if (PRIMARY == prevPRIMARY)

if (SECONDARY == prevSECONDARY)

print PAGE

 else

 print PRIMARY:SECONDARY:PAGE

 else

 print PRIMARY:SECONDARY:PAGE

 prevPRIMARY = PRIMARY

 prevSECONDARY = SECONDARY
```


运维技术书籍免费下载 https://www.linuxprobe.com

我们来看这些代码如何处理一系列的条目，首先是：

```
XView::18
```

主关键字和前面的主关键字不匹配，将按下面形式输出行：

```
XView::18
```

下一个条目是：

```
XView:about:3
```

当我们将这个条目的主关键字和前面一个相比较时，它们是相同的。当比较次关键字时，它们不同，我们输出下面的记录：

```
XView:about:3
```

下一条目是：

```
XView:about:7
```

因为主关键字和次关键字都和前面的条目相匹配，我们将只输出页码。（用 printf 语句代替 print 语句避免自动换行。）这个页码被添加到前面的条目中，和下面的一样：

```
XView:about:3,7
```

下一个条目也和两个关键字都匹配：

```
XView:about:10
```

再次只输出页码，现在的条目如下所示：

```
XView:about:3,7,10
```

这样 3 个只有页码不同的条目将被合并为一个项。

在完整的脚本中添加一个附加的测试，来确定卷的标示符是否匹配。下面是完整的 pagenums.idx 脚本：

```
#!/work/bin/nawk -f

# ----------------------------------------------
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
# pagenums.idx - 收集相同条目的页面
#作者: Dale Dougherty
# Version 1.1 7/10/90
#
# 输入应该是 PRIMARY:SECONDARY:PAGE:VOLUME
# ----------------------------------------------
BEGIN { FS = ":"; OFS = ""}
# 主例程 - 应用到所有的输入行
{
# 将字段赋给变量
PRIMARY = $1
SECONDARY = $2
PAGE = $3
VOLUME = $4
# 检查 see also 条目并收集到数组中
if (SECONDARY ~ /\([Ss]ee +[Aa]lso/) {
# 创建临时副本并将 "~zz" 从副本中除去
tmpSecondary = SECONDARY
sub(/~zz\([Ss]ee +[Aa]lso */, "", tmpSecondary)
sub(/\) */, "", tmpSecondary)
# remove secondary key along with "~zz"
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
sub(/^.*~zz\([Ss]ee +[Aa]lso */, "", SECONDARY)

sub(/\) */, "", SECONDARY)

# assign to next element of seeAlsoList

seeAlsoList[++eachSeeAlso] = SECONDARY "; "

prevPrimary = PRIMARY

# assign copy to previous secondary key

prevSecondary = tmpSecondary

next

} # end test for see Also

# Conditionals to compare keys of current record to previous

# record.  If Primary and Secondary keys are the same, only

# the page number is printed.

# test to see if each PRIMARY key matches previous key

if (PRIMARY == prevPrimary) {
# test to see if each SECONDARY key matches previous key

if (SECONDARY == prevSecondary)

# test to see if VOLUME matches;

# print only VOLUME:PAGE

if (VOLUME == prevVolume)

printf (",%s", PAGE)

else {

printf ("; ")

volpage(VOLUME, PAGE)
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
}
else{
# if array of See Alsos, output them now
if (eachSeeAlso) outputSeeAlso(2)
# print PRIMARY:SECONDARY:VOLUME:PAGE
printf ("\n%s:%s:", PRIMARY, SECONDARY)
volpage(VOLUME, PAGE)
}
} # end of test for PRIMARY == prev
else { # PRIMARY != prev
# if we have an array of See Alsos, output them now
if (eachSeeAlso) outputSeeAlso(1)
if (NR != 1)
printf ("\n")
if (NF == 1){
printf ("%s:", $0)
}
else {
printf ("%s:%s:", PRIMARY, SECONDARY)
volpage(VOLUME, PAGE)
}
}
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
prevPrimary = PRIMARY

prevSecondary = SECONDARY

prevVolume = VOLUME

} # end of main routine

# at end, print newline

END {

# in case last entry has "see Also"

if (eachSeeAlso) outputSeeAlso(1)

printf("\n")

}

# outputSeeAlso function -- list elements of seeAlsoList

function outputSeeAlso(LEVEL) {

# LEVEL - indicates which key we need to output

if (LEVEL == 1)

printf ("\n%s:(See also ", prevPrimary)

else {

sub(/;.*$/, "", prevSecondary)

printf ("\n%s:%s; (See also ", prevPrimary, prevSecondary)

}
sub(/; $/, ".):", seeAlsoList[eachSeeAlso])

for (i = 1; i <= eachSeeAlso; ++i)

printf ("%s", seeAlsoList[i])
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
    eachSeeAlso = 0

    }

    # volpage function -- determine whether or not to print volume
info

    # two args: volume & page

    function volpage(v, p)

   {

    # if VOLUME is empty then print PAGE only

    if ( v == "" )

    printf ("%s", p)

    else

    # otherwise print VOLUME^PAGE

   printf ("%s^%s",v, p)

   }
```

注意，首先程序的输入应该已经按关键字排序。页码也是有顺序的，因此在输入中第七页上的条目“graphics”将出现在第十页上的这个条目的前面。同样，第一卷的条目将出现在第二卷的条目前面。因此，这个程序不需要处理排序；它只是简单地比较关键字，如果它们相同，将相应的页码加入到一个列表中。用这个方法可以减少条目。

这个脚本也可以处理“see also”条目。因为这些记录已经被排序，所以可以删除特殊的排序序列“~zz”。我们还处理了遇到连续的“see also”条目的情况。我们不希望输出：

```
    Toolkit (see also Xt) (See also XView) (See also Motif).
```

而我们希望下面所示的那样，将它们组合并放到一个列表中：

```
    Toolkit (see also Xt; XView; Motif)
```


运维技术书籍免费下载 https://www.linuxprobe.com

要完成这一功能，我们创建了名为 seeAlsoList 的数组。从 SECONDARY 开始删除圆括号，次关键字（如果存在）和“see also”，然后将它赋给 seeAlsoList 的一个元素。我们对 SECONDARY 中的次关键字做备份，并将它赋给 prevSecondary，以便和下一个条目比较。

调用函数 outputSeeAlso()来读取这个数组的所有元素并打印它们。函数 volpage()用于确定是否需要输出卷号。这两个函数在程序代码中不止一个地方被调用，因此将它们定义为函数的主要原因是为了减少重复。

下面是一个处理单本书索引的结果：

```
X Window System:Xlib:6
XFontStruct structure::317
Xlib::6
Xlib:repainting canvas:88
Xlib.h header file::89,294
Xv_Font type::310
XView::18
XView:about:3,7,10
XView:as object-oriented system:17
```

下面是一个处理主索引所产生的结果：

```
reserved names:table of:I^43
Xt:example of programming interface:I^44,65
Xt:objects; list of:I^43,58; II^40
Xt:packages:I^43,61; II^42
Xt:programs; initialization:I^45
Xt:reserved names:I^43,58
Xt:reserved prefixes:I^43,58
Xt:types:I^43,54,61
```

符号“^”作为卷号和页码列表之间的临时定界符。

合并具有相同关键字的条目

程序 pagenums.idx 减少了一些条目，这些条目除了页码外都是相同的。现在我们将处理有相同主关键字的条目。我们还希望查找连续的页码并将它们合并为一个范围。

程序 combine.idx 和 pagenums.idx 很相像，它对索引来进行另一次扫描，比较

运维技术书籍免费下载 https://www.linuxprobe.com

主关键字相同的条目。下面的伪代码抽象了这个比较（为了使问题更简单，我们将忽略第三关键字。而只显示如何比较主关键字和次关键字）。当 pagenums.idx 处理完所有条目时，则不存在主关键字和次关键字都相同的两个条目。因此，我们不必比较次关键字。

```
PRIMARY = $1
SECONDARY = $2
PAGE = $3
if (PRIMARY == prevPRIMARY)
print :SECONDARY:
else
print PRIMARY:SECONDARY
prevPRIMARY = PRIMARY
prevSECONDARY = SECONDARY
```

如果主关键字匹配，我们将只输出次关键字。例如，如果有下面的三个条目：

```
XView:18
XView:about:3, 7, 10
XView:as object-oriented system:17
```

他们将按下面的方式输出：

```
XView:18
:about:3, 7, 10
:as object-oriented system:17
```

当主关键字相同时可以将它们删除。实际代码要更复杂一些，因为包含了第三关键字。我们必须测试主关键字和次关键字来看他们是否唯一或相同，而不必测试第三关键字（我们只需知道它们存在）。

你将注意到上面的伪代码没有输出页码。这个脚本的第二个作用就是检查页码，并将连续的页码合并到一个列表。由逗号分隔的页码列表可以用 split() 函数将它们输入到一个数组中。

要检查代码是否连续，我们可以遍历数组，并将其中的每个元素与它前面的元素加一的结果相比较。

```
eachpage[j-1]+1 == eachpage[j]
```

换句话说就是，如果前面的元素加 1 后得到当前的元素，那么它们是连续的。

运维技术书籍免费下载 https://www.linuxprobe.com

前面一个元素成为这个范围中的第一个页码而当前元素成为最后一个页码。这实在 while 循环中处理的知道循环条件为假，且页码不连续。然后我们输出用连字符分隔的第一个页码和最后一个页码：

```
23-25
```

实际代码比这更复杂，因为要从一个函数中调用它并且识别卷和页码对。首先必须将卷号从页码列表中分离出来，然后通过调用函数{rageOfPage()}来处理页码列表。

下面是程序 combine.idx 的完整代码：

```
#!/work/bin/nawk -f
# ------------------------------------------------
# combine.idx -- merge keys with same PRIMARY key
# and combine consecutive page numbers
# Author: Dale Dougherty
# Version 1.1 7/10/90
#
# input should be PRIMARY:SECONDARY:PAGELIST
# ------------------------------------------------
BEGIN { FS = ":"; OFS = ""}
# main routine -- applies to all input lines
# It compares the keys and merges the duplicates.
{
    # assign first field
    PRIMARY=$1
    # split second field, getting SEC and TERT keys.
    sizeOfArray = split($2, array, ";")
    SECONDARY = array[1]
    TERTIARY = array[2]
    # test that tertiary key exists
    if (sizeOfArray > 1) {
        # tertiary key exists
        isTertiary = 1
        # two cases where ";" might turn up
        # check SEC key for list of "see also"
       if (SECONDARY ~ /\([sS]ee also/){
               SECONDARY = $2
               isTertiary = 0
       }
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
        # check TERT key for "see also"
        if (TERTIARY ~ /\([sS]ee also/){
            TERTIARY = substr($2, (index($2, ";") + 1))
        }
    }
    else # tertiary key does not exist
        isTertiary = 0
    # assign third field
    PAGELIST = $3
    # Conditional to compare primary key of this entry to that
    # of previous entry. Then compare secondary keys. This
    # determines which non-duplicate keys to output.
    if (PRIMARY == prevPrimary) {
        if (isTertiary && SECONDARY == prevSecondary)
            printf (";\n::%s", TERTIARY)
        else
            if (isTertiary)
                printf ("\n:%s; %s", SECONDARY, TERTIARY)
            else
                printf ("\n:%s", SECONDARY)
        }
    else {
        if (NR != 1)
            printf ("\n")
        if ($2 != "")
            printf ("%s:%s", PRIMARY, $2)
        else
            printf ("%s", PRIMARY)
        prevPrimary = PRIMARY
    }
    prevSecondary = SECONDARY
} # end of main procedure
# routine for "See" entries (primary key only)
NF == 1 { printf ("\n") }
# routine for all other entries
# It handles output of the page number.
NF > 1 {
    if (PAGELIST)
    # calls function numrange() to look for
    # consecutive page numbers.
        printf (":%s", numrange(PAGELIST))
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
    else
        if (! isTertiary || (TERTIARY && SECONDARY)) printf (":")
                } # end of NF > 1
        # END procedure outputs newline
END { printf ("\n") }
# Supporting Functions
# numrange -- read list of Volume^Page numbers, detach Volume
# from Page for each Volume and call rangeOfPages
# to combine consecutive page numbers in the list.
# PAGE = volumes separated by semicolons; volume and page
# separated by ^.
function numrange(PAGE, listOfPages, sizeOfArray)
{
    # Split up list by volume.
    sizeOfArray = split(PAGE, howManyVolumes,";")
    # Check to see if more than 1 volume.
        if (sizeOfArray > 1) {
            # if more than 1 volume, loop through list
            for (i = 1; i <= sizeOfArray; ++i) {
                # for each Volume^Page element, detach Volume
                # and call rangeOfPages function on Page to
                # separate page numbers and compare to find
                # consecutive numbers.
                if (split(howManyVolumes[i],volPage,"^") == 2)
                    listOfPages = volPage[1] "^"
                        rangeOfPages(volPage[2])
                    # collect output in listOfPages
                if (i == 1)
                    result = listOfPages
                else
                    result=result ";" listOfPages
            } # end for loop
        }
        else { # not more than 1 volume
            # check for single volume index with volume number
            # if so, detach volume number.
            # Both call rangeOfPages on the list of page numbers.
            if (split(PAGE,volPage,"^") == 2 )
                # if Volume^Page, detach volume and then call
             rangeOfPages
                listOfPages = volPage[1] "^"
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
            rangeOfPages(volPage[2])
            else # No volume number involved
                listOfPages = rangeOfPages(volPage[1])
            result = listOfPages
        } # end of else
        return result # Volume^Page list
} # End of numrange function
# rangeOfPages -- read list of comma-separated page numbers,
# load them into an array, and compare each one
# to the next, looking for consecutive numbers.
# PAGENUMBERS = comma-separated list of page numbers
function rangeOfPages(PAGENUMBERS, pagesAll, sizeOfArray,pages,
                listOfPages, d, p, j) {
   # close-up space on troff-generated ranges
   gsub(/ - /, ",-", PAGENUMBERS)
    # split list up into eachpage array.
    sizeOfArray = split(PAGENUMBERS, eachpage, ",")
    # if more than 1 page number
    if (sizeOfArray > 1){
        # for each page number, compare it to previous number + 1
        p = 0 # flag indicates assignment to pagesAll
        # for loop starts at 2
        for (j = 2; j-1 <= sizeOfArray; ++j) {
            # start by saving first page in sequence (firstpage)
            # and loop until we find last page (lastpage)
            firstpage = eachpage[j-1]
            d = 0 # flag indicates consecutive numbers found
            # loop while page numbers are consecutive
            while ((eachpage[j-1]+1) == eachpage[j] ||
                eachpage[j] ~ /^-/) {
                # remove "-" from troff-generated range
                    if (eachpage[j] ~ /^-/) {
                        sub(/^-/, "", eachpage[j])
                    }
                lastpage = eachpage[j]
                # increment counters
                ++d
                ++j
            } # end of while loop
            # use values of firstpage and lastpage to make range.
            if (d >= 1) {
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
                # there is a range
                pages = firstpage "-" lastpage
            }
            else # no range; only read firstpage
                pages = firstpage
            # assign range to pagesAll
                if (p == 0) {
                    pagesAll = pages
                    p = 1
                }
                else {
                    pagesAll = pagesAll "," pages
                }
        }# end of for loop
    # assign pagesAll to listOfPages
    listOfPages = pagesAll
    } # end of sizeOfArray > 1
    else # only one page
        listOfPages = PAGENUMBERS
        # add space following comma
gsub(/,/, ", ", listOfPages)
        # return changed list of page numbers
return listOfPages
} # End of rangeOfPages function
```

这个脚本包括很短的 BEGIN 和 END 过程。主例程用于比较主关键字和次关键字。这个例程的第一部分将字段赋值给变量。第二个字段包含次关键字和第三关键字，我们使用 split()分隔它们。然后我们测试第三关键字是否存在并设置标志 isTertiary 为 1 或 0。

主过程的下一部分包含用于查找相通关键字的条件表达式。和我们在讨论伪代码时对这个例程的本部分介绍的一样，具有完全相同关键字的条目已经被 pagenums.idx 删除。
这个过程中的条件根据每个关键字是否唯一来决定输出什么样得内容。如果主关键字是唯一的，那么连同条目中余下的部分一起输出。如果主关键字和前面的匹配，则比较次关键字。如果关键字是唯一的，那么连同条目中余下的部分一起输出。如果主关键字和前面的关键字匹配，而且次关键字和前面的次关键字匹配，那么第三关键字必须是唯一的。那么我们只输出第三关键字，主关键字和次关键字为空。

下面列出了不同的形式：

运维技术书籍免费下载 https://www.linuxprobe.com

```
主关键字
主关键字：次关键字
：次关键字
：次关键字：第三关键字
：第三关键字
主关键字：次关键字：第三关键字
```

主过程后面跟着两个附加程序。第一个只有在 NF 等于 1 时才执行。它处理以上列表中的第一种形式。也就是说，因为没有页码，所以必须输出换行符来完成输入。

第二个过程处理所有页码的条目，在这个过程中我们调用一个函数将页码列表拆开并寻找连续的页。它还调用函数 numrange()，该函数的主要目的是处理多卷索引，这里的页码列表如下所示：

```
I^35,55; II^200
```

这个函数调用 split()并利用一个分号作为定界符来分隔每个卷。然后我们调用 split()利用“^”作为定界符将卷号从页码列表中分离开。一旦有了这个页码列表，我们将调用第二个函数 rangeOfPages()来查找连续的页码。对于单本书的索引，如本章所给出的样本例子，函数 numrange()没有做任何事，只是调用了 rangeOfPages()函数。我们在前面讨论过函数 rangeOfPages()的主要内容，创建了 eachpage 数组并利用 while 循环访问这个数组来与前一个的元素相比较。这个函数返回页码的列表。

这个程序的样本输出如下所示：

```
Xlib:6
:repainting canvas:88
Xlib.h header file:89, 294
Xv_Font type:310
XView:18
:about:3, 7, 10
:as object-oriented system:17
:compiling programs:41
:concept of windows differs from X:25
:data types; table of:20
:example of programming interface:44
:frames and subframes:26
:generic functions:21
```


运维技术书籍免费下载 https://www.linuxprobe.com

:Generic Object:18, 24
:libraries:42
:notification:10, 35
:objects:23-24;
:: table of:20;
:: list of:43
:packages:18, 43
:programmer’ s model:17-23
:programming interface:41
:programs; initialization:45
:reserved names:43
:reserved prefixes:43
:structure of applications:41
:subwindows:28
:types:43
:window objects:25

要特别注意在“XView”下的“对象”。这是一个有多个第三关键字和次关键字的例子。他也是有连续页码范围的条目的一个例子

格式化索引

前面的脚本几乎完成了所有的处理，并给出了有序的输入纪律的列表。Format.idx 脚本或许是最简单的脚本，读取输入记录的列表被生成两个格式不同的报告，一个显示在终端屏幕上，一个被输送到 troff 利用激光打印机打印。唯一的问题时我们要输出的条目需要依据字母表的字母来分组。

用命令行参数设置变量 FMT 来确定采用哪种输出格式。

下面是 format.idx 的完整程序：

```
#!/work/bin/nawk -f
# ------------------------------------------------
# format.idx -- prepare formatted index
# Author: Dale Dougherty
# Version 1.1 7/10/90
#
# input should be PRIMARY:SECONDARY:PAGE:VOLUME
# Args: FMT = 0 (default) format for screen
# FMT = 1 output with troff macros
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
# MACDIR = pathname of index troff macro file
# ----------------------------------------------
BEGIN { FS = ":"
upper = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
lower = "abcdefghijklmnopqrstuvwxyz"
}
# Output initial macros if troff FMT
NR == 1 && FMT == 1 {
if (MACDIR)
printf (".so %s/indexmacs\n", MACDIR)
else
printf (".so indexmacs\n")
printf (".Se \"\" \"Index\"\n")
printf (".XC\n")
} # end of NR == 1
Generating a For matted Index 313
# main routine - apply to all lines
# determine which fields to output
{
# convert octal colon to "literal" colon
# make sub for each field, not $0, so that fields are not parsed
gsub(/\\72/, ":", $1)
gsub(/\\72/, ":", $2)
gsub(/\\72/, ":", $3)
# assign field to variables
PRIMARY = $1
SECONDARY = $2
TERTIARY = ""
PAGE = $3
if (NF == 2) {
SECONDARY = ""
PAGE = $2
}
# Look for empty fields to determine what to output
if (! PRIMARY) {
if (! SECONDARY) {
TERTIARY = $3
PAGE = $4
if (FMT == 1)
printf (".XF 3 \"%s", TERTIARY)
else
printf (" %s", TERTIARY)
}
else
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
if (FMT == 1)
printf (".XF 2 \"%s", SECONDARY)
else
printf (" %s", SECONDARY)
}
else { # if primary entry exists
# extract first char of primary entry
firstChar = substr($1, 1, 1)
# see if it is in lower string.
char = index(lower, firstChar)
# char is an index to lower or upper letter
if (char == 0) {
# if char not found, see if it is upper
char = index(upper, firstChar)
if (char == 0)
char = prevChar
}
# if new char, then start group for new letter of alphabet
if (char != prevChar) {
if (FMT == 1)
printf(".XF A \"%s\"\n", substr(upper, char, 1))
else
printf("\n\t\t%s\n", substr(upper, char, 1))
prevChar = char
}
# now output primary and secondary entry
if (FMT == 1)
if (SECONDARY)
printf (".XF 1 \"%s\" \"%s", PRIMARY, SECONDARY)
else
printf (".XF 1 \"%s\" \"", PRIMARY)
else
if (SECONDARY)
printf ("%s, %s", PRIMARY, SECONDARY)
else
printf ("%s", PRIMARY)
}
# if page number, call pageChg to replace "^" with ":"
# for multi-volume page lists.
if (PAGE) {
if (FMT == 1) {
# added to omit comma after bold entry
if (! SECONDARY && ! TERTIARY)
printf ("%s\"", pageChg(PAGE))
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
else
printf (", %s\"", pageChg(PAGE))
}
else
printf (", %s", pageChg(PAGE))
}
else if (FMT == 1)
printf("\"")
printf ("\n")
} # End of main routine
# Supporting function
# pageChg -- convert "^" to ":" in list of volume^page
# Arg: pagelist -- list of numbers
function pageChg(pagelist) {
gsub(/\^/, ":", pagelist)
if (FMT == 1) {
gsub(/[1-9]+\*/, "\\fB&\\P", pagelist)
gsub(/\*/, "", pagelist)
}
return pagelist
}# End of pageChg function
```

BEGIN 过程定义字段分隔符和字符串 upper 与 lower。下一个过程输出了文件名，这个文件中包含 troff 索引宏的定义。宏目录的名字可以作为命令行的第二个参数来设置。

主过程在开关将“隐含的”冒号改用字面上的冒号。注意，我们应用 gsub()函数来处理每个字段而不是处理整个行，因为后者将会引起当前行再次求值，而且字段的当前顺序将被打乱。

下一步我们将字段赋给变量，然后测试字段是否为空。如果主关键字没有定义，则检测关键字是否被定义。如果是，则将它输出。否则输出第三关键字。如果主关键字被定义，那么我们提取它的第一个字符并观察它是否是在字符串 lower 中。

```
firstChar = substr($1, 1, 1)
char = index(lower, firstChar)
```

char 变量包含字母在字符中的位置。如果这个数字大于或等于 1，那么我们也将一个索引加入到 upper 字符串中。我们比较每个签上并且当 char 与 prevChar 相同时，字母表中的当前字母保持不变。如果不同，我们首先检查 upper 字符串中的字母。如果 char 是新字母，我们输出中间的字符串，它用于标识字母表中的字母。

运维技术书籍免费下载 https://www.linuxprobe.com

然后我们观察输出的主要关键字和次关键字条目。最后，在调用函数 pageChg()函数用冒号代替在卷一页引用中的“^”后，输出页码的列表。

由 format.idx 产生的，输出到屏幕上的样本如下所示：

```
X
X Protocol, 6
X Window System, events, 84
extensibility, 9
interclient communications, 9
overview, 3
protocol, 6
role of window manager, 9
server and client relationship, 5
software hierarchy, 6
toolkits, 7
X Window ID for paint window, 87
Xlib, 6
XFontStruct structure, 317
Xlib, 6
repainting canvas, 88
Xlib.h header file, 89, 294
Xv_Font type, 310
XView, 18
about, 3, 7, 10
as object-oriented system, 17
compiling programs, 41
concept of windows differs from X, 25
data types; table of, 20
example of programming interface, 44
frames and subframes, 26
generic functions, 21
Generic Object, 18, 24
```

由 format.idx 产生的，输出到 troff 的样本如下所示：

```
.XF A "X"
.XF 1 "X Protocol" "6"
.XF 1 "X Window System" "events, 84"
.XF 2 "extensibility, 9"
.XF 2 "interclient communications, 9"
.XF 2 "overview, 3"
.XF 2 "protocol, 6"
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
.XF 2 "role of window manager, 9"
.XF 2 "server and client relationship, 5"
.XF 2 "software hierarchy, 6"
.XF 2 "toolkits, 7"
.XF 2 "X Window ID for paint window, 87"
.XF 2 "Xlib, 6"
.XF 1 "XFontStruct structure" "317"
.XF 1 "Xlib" "6"
.XF 2 "repainting canvas, 88"
.XF 1 "Xlib.h header file" "89, 294"
.XF 1 "Xv_Font type" "310"
.XF 1 "XView" "18"
.XF 2 "about, 3, 7, 10"
.XF 2 "as object-oriented system, 17"
```

这些输出必须用 troff 格式化，来产生索引的打印版本。这本书的索引最初是利用 masterindex 程序来处理的。

masterindex 的 shell 脚本

masterindex 的 shell 脚本用于将所有这些脚本组合在一起，并基于用户的命令行的相应选项来调用它们。例如，用户输入：

```
$ masterindex -s m volume1 volume2
```

表示用文件 volume1 和 volume2 创建了一个主索引，并且输出送往屏幕。

masterindex 的 shell 脚本和文档一起出现在附录三中。

masterindex 程序的其他细节

本节介绍 masterindex 程序的几个有趣的细节，这些或许不能引起你的注意。本节的目的是提取几个有趣的程序段，并显示它们是如何解决特殊问题的。

如何隐藏特殊字符

第一段程序取自 input.idx 程序，它的工作就是在对索引条目排序前将它们标准化。这个程序认为它的输入记录是由两个用制表符分隔的字段组成：索引条目和它的页码。冒号被作为这个语法的一部分来标识一个索引条目的各部分。

因为这个程序将冒号作为一个特殊字符，所以我们必须提供一个方法来通过程序传递一个字面上的冒号。要实现这一功能，我们允许编索引的人员在输入中指定两个连续的冒号。然而，我们不能简单地将这个序列转换为一个字面冒号，因为由 masterindex 调用的程序模块的剩余部分读取了用冒号分隔的 3 个字段。解决办法是使用 gsub()函数将冒号转换为它的八进制值。

运维技术书籍免费下载 https://www.linuxprobe.com

```
#< from input.idx
# convert literal colon to octal value
$1 ~ /::/ {
# substitute octal value for "::"
gsub(/::/, "\\72", $1)
```

“\\72”是冒号的八进制值（你可以浏览/uer/pub/ascii 文件中的十六进制和八进制对照表来找到这个值）。在最后的程序模块中，我们使用 gsub()函数将八进制值转换回冒号。下面来自 format.idx 的代码：

```
#< from format.idx
# convert octal colon to "literal" colon
# make sub for each field, not $0, so that fields are not parsed
gsub(/\\72/, ":", $1)
gsub(/\\72/, ":", $2)
gsub(/\\72/, ":", $3)
```

要注意的第一方面是我们为这 3 个字段分别做了替换，而不是利用一个替换命令对$0 操作。原因是输入字段是用冒号分隔的。当 awk 浏览一行时，它将行分隔为字段。如果你在脚本中的任何位置修改$0 的内容，awk 将重新对$0 求值并再次将行分解成字段。因而，如果有 3 个字段要做转换，并且替换改变了其中的一个，在$0 中添加了一个冒号，那么 awk 将辨认出 4 个字段。通过将每个字段进行替换，避免了将行又分解成字段。

互换两部分

上面我们曾讨论过用冒号分隔主关键字和次关键字的语法。对一些类型的条目，利用次关键字进行分类是有意义的。例如，我们可能有一组程序语句或用户命令，例如“sed command”。编索引的人员可能会创建两条目：一个是“sed command”，另一个是“command:sed”。为了对这种条目编码更容易，我们使用了一种编码规则，利用字符（～）来标记这种条目的两部分，使得第一和第二部分可以互换来自动创建第二个条目（注 5）。因而，编写下面的索引条目

```
.XX "sed~command"
```

产生两个条目：

```
sed command 43
command: sed 43
```

下面是互换条目的代码：

运维技术书籍免费下载 https://www.linuxprobe.com

```
#< from input.idx
# Match entries that need rotating that contain a single tilde
$1 ~ /~/ && $1 !~ /~~/ {
# split first field into array named subfield
n = split($1, subfield, "~")
if (n == 2) {
# print entry without "~" and then rotated
printf("%s %s::%s\n", subfield[1], subfield[2], $2)
printf("%s:%s:%s\n", subfield[2], subfield[1], $2)
}
next
}
```

这里的模式匹配规则与任何包含一个代字符而不是两个连续字符的条目匹配，后者表示一个字面上的代字符。这个过程利用函数 split()将第一个字段分割为两个“子字段”。这将为我们提供两个子串，一个在代字符的前面，一个在代字符的后面。输出原始的条目，然后将互换后的条目也输出，两个都用 printf 语句。

因为代字符被作为特殊字符，我们在输入中利用两个连续的代字符表示一个字面上的代字符。在对一个条目的两部分进行互换后是下面的代码。

```
#< from input.idx
# Match entries that contain two tildes
```

注 5： 这个转换索引条目的观点来自于《The AWK Programming Language》。然而，在《The AWK Programming Language》中，在有空格的地方，条目可以自动互换；用代字符来防止“加入”空格来做互换。这里不是用默认的操作来互换，我们利用不同的编码习惯，这里的代字符表示互换发生的地方。

```
$1 ~ /~~/ {
# replace ~~ with ~
gsub(/~~/, "~", $1)
}
```

和冒号不同，在整个 masterindex 程序中冒号具有特殊意义，而代字符在这个模块执行完后将没有意义，因此我们可以简单地输出一个字面上的代字符。

运维技术书籍免费下载 https://www.linuxprobe.com

寻找替换

下一段代码也是取自 input.idx。要讨论的问题是查找两个被文本分隔的冒号，并将第二个冒号改为分号。如果输入行包含

```
class: class initialize: (see also methods)
```

那么结果是：

```
class: class initialize; (see also methods)
```

将这个问题进行说明也很简单。我们希望修改第二个冒号，而不修改第一个。在 sed 中处理这个问题非常简单，因为它提供了在替换部分选择和重新调用所匹配内容的一部分功能（用\（….\）将要匹配的部分包围起来，并用\1 来重新调用第一部分）。因为在 awk 中缺少这些功能，所以必须使用其他方法。下面是一个可能的解决方法：

```
#< from input.idx
# replace 2nd colon with semicolon
if (sub(/:.*:/, "&;", $1))
sub(/:;/, ";", $1)
```

第一个替换匹配两个冒号中间的全部内容。它用所匹配的内容（&）后跟一个分号来做替换。这个替换发生在一个计算函数 sub()返回值的条件表达式中。注意，如果远的成功这个函数返回 1，而不是返回结果字符串。换句话说，如果我们完成了第一个替换，那么我们将做第二个。第二个替换用“；”替换“：；”。因为我们不能做直接的替换，我们通过构造第二个冒号为特殊形式的上下文来间接地完成替换。

报告错误的函数

程序 input.idx 的目的就是允许在编写索引条目时有变化（或有一定的不一致）。通过将这些变化减少为一种基本形式，可以更容易地编写其他程序。

另一方面是如果程序 input.idx 不能接受条目，必须向用户报告并删除这个条目，以便不影响其他的程序。程序 input.idx 有一个函数 pinterr()，可以用来产生错误报告，如下面所示：

```
function printerr (message) {
# print message, record number and record
printf("ERROR:%s (%d) %s\n", message, NR, $0) > "/dev/tty"
}
```

这个函数使得用标准方式报错误更简单。它将 message 作为参数看待，message 通常是一个描述错误的字符串。将这个消息连同记录号和记录本身一起输出。

运维技术书籍免费下载 https://www.linuxprobe.com

输出直接发送到用户的终端“/dev/tty”。这是一种很好的习惯，因为在这种情况下，程序的标准输出可能被输送给一个管道或一个文件。我们也可以将出错信息发送到标准错误文件，如下所示：

```
print "ERROR:" message " (" NR ") " $0 | "cat 1>&2"
```

这为 cat 打开了一个管道，将 cat 的标准输出重定向到标准错误文件。如果你用的是 gawk、mawk 或 Bell Labs awk，你可以这样写：

```
printf("ERROR:%s (%d) %s\n", message, NR, $0) > "/dev/stderr"
```

在这个程序中，printerr()函数可以如下调用：

```
printer(“No page number”)
```

当产生错误时，用户将看到下面的出错信息：

```
ERROR:No page number (612) geometry management:set_values_almost
```

处理 see also 条目

索引条目有一种类型是“see also”。和“see”引用一样，它为读者提供另一个条目。然而，“see also”条目也可能有一个页码。换句话，这种条目包含它自己的信息，但为读者提供在另一片的附加信息。下面是几个样本条目：

```
error procedure 34
error procedure (see also XtAppSetErrorMsgHandler) 35
error procedure (see also XtAppErrorMsg)
```

这个样本的第一个条目有页码而最后一个没有。当程序 input.idx 找到一个“see also”条目时，将检查是否有页码，如果有页码，则输出两个记录，第一个是没有页码的条目，第二个是有一个页码但没有“see also”引用。

```
#< input.idx
# if no page number
if ($2 == "") {
print $0 ":"
next
}
else {
# output two entries:
# print See Also entry w/out page number
print $1 ":"
# remove See Also
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
sub(/ *~zz\(see also.*$/, "", $1)
sub(/;/, "", $1)
# print as normal entry
if ( $1 ~ /:/ )
print $1 ":" $2
else
print $1 "::" $2
next
}
```

下一个要解决的问题是如何用正确的顺序对条目排序。程序 sort 利用我们提供的选项，将“see also”条目放在以“s”开关的次关键字下（-d 选项可以使圆括号被忽略）。为了改变排序的顺序，我们通过在条目之前添加序列“~zz”来改变排序关键字。

```
#< input.idx
# add "~zz" for sort at end
sub(/\([Ss]ee [Aa]lso/, "~zz(see also", $1)
```

这个排序函数没有解释~符号，但~符号有助于识别后来要删除的字符串。添加“~zz”确保我们将这些条目排序到按次关键字或第三关键字排序的列表末端。

pagenums.idx 脚本从“see also”条目中将排序字符串删除。然而，和我们以前所讨论的一样，我们找到了一系列具有关键字的“see also”条目建立了一个列表。因此，我们也将删除具有相同关键字的部分，并将引用本身加入到一个数组：

```
#< pagenums.idx
# remove secondary key along with "~zz"
sub(/^.*~zz\([Ss]ee +[Aa]lso */, "", SECONDARY)
sub(/\) */, "", SECONDARY)
# assign to next element of seeAlsoList
seeAlsoList[++eachSeeAlso] = SECONDARY "; "
```

有一个函数能够输出“see also”条目的列表，并用分号来分隔每一项。因此，通过 pagenums.idx 将输出如下的“see also”条目：

```
error procedure:(see also XtAppErrorMsg; XtAppSetErrorHandler.)
```

可选择的排序方法

在这个程序中，我们选择在索引条目中不支持 troff 字体和磅数请求。如果你希望支持特殊的转义序列，《The AWK Programming Language》中介绍了一个方法，对于每个记录，提取第一个字段并将它附加在这个记录上，作为排序的关键字。既然第一个字段产生了重复，可以从排序关键字中删除转义序列。一

运维技术书籍免费下载 https://www.linuxprobe.com

旦条目被排好序，就可以删除排序关键字。这个操作防止了转义序列打扰排序。

另一个方法所做的工作和我们对“see also”条目所做的类似。因为在排序中忽略了特殊字符，我们可以用 input.idx 程序转换 troff 字体更改序列，例如将“fB”改为“~~~”，将“\f1”改为“~~~~”，或任何合适的转义序列。这可以使 sort 程序处理这个序列而不影响排序（这个技术被 Steve Tallbott 应用在他的原始的索引脚本中）。

在两种情况下需要处理的惟一问题是：对于具有相同术语的两个条目，一个带有字体信息，另一个没有，当将一个同另一个比较时将被看做是不同的条目。

运维技术书籍免费下载 https://www.linuxprobe.com

第十三章

脚本的汇总

本章内容

- Untot.awk-uucp 的统计报告
- Phonebill-跟踪电话使用情况
- Combine-抽取多部分用
- uuencoded 编码技术处理的二进制代码
- Mailavg—检查邮箱的大小
- Adj-调整文本文件的行
- readsource—将程序原文件格式化为 troff 格式
- gent—获得 termcap 条目
- plpr—行式打印的预处理器
- transpose—实现矩阵的转置
- m1—简单的宏处理器

本章包含了由 Usenet 用户提供的脚本的一个汇总。每
了一个简洁的介绍。我们的注释放在方括号中[就像这
显示了出来。如果作者没有提供例子，我们将在列表
后，在称为“程序注意事项”的部分，我们简要讨论
着重说明了一下。下面是这些脚本的一个摘要。

uutot.awk	UUCP 的统计报告
phonebill	跟踪电话的使用情况
combine	抽取多部分用 uuencoded 编码技术处理的二进制代码
mailavg	检查邮箱的大小
readsource	将程序源文件格式化为 troff 格式
gent	获得 termcap 条目
plpr	行式打印的预处理器
transpose	实现矩阵的转置
m1	简单的宏处理器

uutot.awk—UUCP 的统计报告

由 Roger A. Cornelius 提供

下面是我在 nawk 环境中编写的一些东西，可以完成用 C 语言版本完成的相同的事情，这些被发送到 alt.sources，一会儿就会返回。它主要是统计了 uucp 的连接（连接时间，传输量，传输的文件数等等）。它只支持 HDB 风格的日志文件，但将显示逐站点传输的统计，或总的统计（所有站点）（和它一起工作的还有/uer/spool/uucp/SYSLOG）（注：英文原文为/usr/spool/uucp/SYSLOG。）

运维技术书籍免费下载 https://www.linuxprobe.com

我使用 shell 实现通过调用“awk -f”来运行这个程序，但这是不必要的。使用说明在程序的首部。（很抱歉没有注释。）

```
# @(#) uutot.awk - display uucp statistics - requires new awk
 # @(#) Usage:awk -f uutot.awk [site ...]
 /usr/spool/uucp/.Admin/xferstats
# Author: Roger A. Cornelius (rac@sherpa.uucp)
# dosome[]; # site names to work for - all if not set
# remote[]; # array of site names
# bytes[]; # bytes xmitted by site
# time[]; # time spent by site
# files[]; # files xmitted by site
BEGIN {
   doall = 1;
   if (ARGC > 2) {
      doall = 0;
      for (i = 1; i < ARGC-1; i++) {
         dosome[ ARGV[i] ];
         ARGV[i] = "";
      }
   }
   kbyte = 1024 # 1000 if you’re not picky
   bang = "!";
   sending = "->";
   xmitting = "->" "|" "<-";
   hdr1 = "Remote K-Bytes K-Bytes K-Bytes " \
      "Hr:Mn:Sc Hr:Mn:Sc AvCPS AvCPS # #\n";
   hdr2 = "SiteName Recv Xmit Total " \
      "Recv Xmit Recv Xmit Recv Xmit\n";
   hdr3 = "-------- --------- --------- --------- -------- " \
      "-------- ----- ----- ---- ----";
   fmt1 = "%-8.8s %9.3f %9.3f %9.3f %2d:%02d:%02.0f " \
      "%2d:%02d:%02.0f %5.0f %5.0f %4d %4d\n";
   fmt2 = "Totals %9.3f %9.3f %9.3f %2d:%02d:%02.0f " \
      "%2d:%02d:%02.0f %5.0f %5.0f %4d %4d\n";
}
{
   if ($6 !~ xmitting) # should never be
   next;
   direction = ($6 == sending ? 1 : 2)
   site = substr($1,1,index($1,bang)-1);
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
        if (site in dosome || doall) {
            remote[site];
            bytes[site,direction] += $7;
            time[site,direction] += $9;
            files[site,direction]++;
        }
    }
    END {
        print hdr1 hdr2 hdr3;
        for (k in remote) {
            rbyte += bytes[k,2]; sbyte += bytes[k,1];
            rtime += time[k,2]; stime += time[k,1];
            rfiles += files[k,2]; sfiles += files[k,1];
            printf(fmt1, k, bytes[k,2]/kbyte, bytes[k,1]/kbyte,
                (bytes[k,2]+bytes[k,1])/kbyte,
                time[k,2]/3600, (time[k,2]%3600)/60, time[k,2]%60,
                time[k,1]/3600, (time[k,1]%3600)/60, time[k,1]%60,
                bytes[k,2] && time[k,2] ? bytes[k,2]/time[k,2] : 0,
                bytes[k,1] && time[k,1] ? bytes[k,1]/time[k,1] : 0,
                files[k,2], files[k,1]);
        }
        print hdr3
        printf(fmt2, rbyte/kbyte, sbyte/kbyte, (rbyte+sbyte)/kbyte,
            rtime/3600, (rtime%3600)/60, rtime%60,
            stime/3600, (stime%3600)/60, stime%60,
            rbyte && rtime ? rbyte/rtime : 0,
            sbyte && stime ? sbyte/stime : 0,
            rfiles, sfiles);
    }
```

生成一个测试文来测试Cornelius的程序。下面是从/uer/spool/uucp/.Admin/xferstats（注：同上）提取的几行（因为在这个文件中每行都太长，不能在一页打印，我们已经将行换段，后面的方向箭头只是出于显示目的）。

```
isla!nuucp S (8/3-16:10:17) (C,126,25) [ttyi1j] ->
   1131/4.880 secs, 231 bytes/sec
isla!nuucp S (8/3-16:10:20) (C,126,26) [ttyi1j] ->
   149/0.500 secs, 298 bytes/sec
isla!sue S (8/3-16:10:49) (C,126,27) [ttyi1j] ->
   646/25.230 secs, 25 bytes/sec
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
    isla!sue S (8/3-16:10:52) (C,126,28) [ttyi1j] ->
       145/0.510 secs, 284 bytes/sec
    uunet!uisla M (8/3-16:15:50) (C,951,1) [cui1a] ->
       1191/0.660 secs, 1804 bytes/sec
    uunet!uisla M (8/3-16:15:53) (C,951,2) [cui1a] ->
       148/0.080 secs, 1850 bytes/sec
    uunet!uisla M (8/3-16:15:57) (C,951,3) [cui1a] ->
       1018/0.550 secs, 1850 bytes/sec
    uunet!uisla M (8/3-16:16:00) (C,951,4) [cui1a] ->
       160/0.070 secs, 2285 bytes/sec
    uunet!daemon M (8/3-16:16:06) (C,951,5) [cui1a] <-
       552/2.740 secs, 201 bytes/sec
    uunet!daemon M (8/3-16:16:09) (C,951,6) [cui1a] <-
       102/1.390 secs, 73 bytes/sec
```

注意：这儿有12个字段，而程序只用了第1、6、7和9字段。使用样本输入运行以上程序将产生下面的结果：

```
    $ nawk -f uutot.awk uutot.test
    Remote K-Bytes K-Bytes  K-Bytes Hr:Mn:Sc Hr:Mn:Sc AvCPS  AvCPS
#    #
    SiteName    Recv    Xmit    Total    Recv    Xmit   Recv  Xmit
Recv Xmit
    ------------------------------------------------------------------
-----------------------------------------------
    uunet      0.639   2.458    3.097  0:04:34  2:09:49    2     0
2    4
    isla       0.000   2.022    2.022  0:00:00  0:13:58    0     2
0    4
    ------------------------------------------------------------------
-----------------------------------------------
    Totals     0.639   4.480    5.119  0:04:34  2:23:47    2     1
2    8
```

uutot.awk程序的注意事项

这个nawk应用软件是一个编写清楚的akw程序的一个极好的例子。它也是一个使用awk将晦涩UNIX日志修改为有用的报告的典型例子。

虽然Cornelius没有给出注释来解释程序的逻辑结构，但从程序开始处的注释中能很清楚的了解该程序的用法。而且，他使用变量来定义搜索模式和报告的布局。这将有助于简化程序体中的条件和打印语句。同时程序中所用的变量名字

运维技术书籍免费下载 https://www.linuxprobe.com

有助于立刻识别出它们的用处。

这个程序的结构包括三个部分，和我们在第七章“编写awk脚本”中所强调的一样。它的组成中包含一个BEGIN过程，在其中定义变量；包含一个主体，在其中访问来自于日志文件的每行数据；以及一个END过程，在其中生成输出的报告。

phonebill—跟踪电话的使用情况

由Nick Holloway提供

这个程序用于计算电话费。在英国电话费是由在通话过程中所用的“单元”的数量来计算的（本地电话不是免费的）。一个“单元”的持续时间长度依赖于带宽的费用（和距离有关）和速率的费用（和一天的时间有关）。通话一开始就得对整个单元付费。

这个程序的输入有四个字段，第一个是日期（不用），第二个是“带宽/速率”用于计算一个单元的时间长度。第三个字段是通话的时间长度。它们可以是“ss”、“mm:ss”或“hh:mm:ss”格式。第四个字段是呼叫者的名字。我们使用一个秒表（旧的便宜的数字型的）、一本书和一支笔，来计算由我的awk脚本反馈的时间的账单。这只计算了打电话的费用，而没有计算固定的费用。

这个程序的目的是尽量减少呼叫者需要输入的信息量，而且可以用来将每个用户打电话的费用收集到一个报告中。这个程序还可用于当英国电话公司改变它的收费标准时，可以在顶层很容易的实现修改（这已经实现过一次）。如果增加更多的带宽收费或速率收费，相应的表可以被简单地扩展（奇妙的是相关的数组）（注：英文原文为wonders of associative arrays，此处疑为翻译不当）。这里对输入的数据没有做彻底的智能检测。用法为：

```
phonebill[file…]
```

下面是输入和输出的（短小的）样本。

输入：

```
29/05 b/p 5:35 Nick
29/05 L/c 1:00:00 Dale
01/06 L/c 30:50 Nick
```

输出：

运维技术书籍免费下载 https://www.linuxprobe.com

```
Summary for Dale:
      29/05 L/c 1:00:00 11 units
Total: 11 units @ 5.06 pence per unit = $0.56
Summary for Nick:
      29/05 b/p 5:35 19 units
   01/06 L/c 30:50 6 units
Total: 25 units @ 5.06 pence per unit = $1.26
```

下面是phonebill的程序列表：

```
#!/bin/awk -f
#-----------------------------------------------------
# Awk 脚本在电话中的使用，计算每个人的话费
# #-------------------------------------------------------
# 作者: N.Holloway (alfie@cs.warwick.ac.uk)
# 日期 : 27 January 1989
# 地点 : University of Warwick
#-----------------------------------------------------
# 条目的格式如下所示
# Date Type/Rate Length Name
#
# 格式:
# 日期 : "dd/mm" - one word
# 带宽/速率 : "bb/rr" (e.g. L/c)
# 时间长度 : "hh:mm:ss", "mm:ss", "ss"
# 名字 : "Fred" - one word (unique)
#-----------------------------------------------------
# 费用信息保存在数组“c”中
# 按“typc/rate”索引
# 而且每个单元的费用保存在变量“pence_per_unit”中
# 信息被保存在两个数组中，都用姓名做索引。第一个“summary”包含输
# 入数据的行。
# 以及单元的个数，而且“units”是每个用户所有单元的总的累加结果
#-----------------------------------------------------
BEGIN \
    {
        # --- 每个单元的费用
        pence_per_unit = 4.40  #每个单元4.4便士
        pence_per_unit *= 1.15 # 增加 15%
        # --- 对于每个单元在不同带宽/速率下的每秒费用列表
        # [ 不能将0作为值输入]
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
        c ["L/c"] = 330 ; c ["L/s"] = 85.0; c ["L/p"] = 60.0;
        c ["a/c"] = 96 ; c ["a/s"] = 34.3; c ["a/p"] = 25.7;
        c ["b1/c"]= 60.0; c ["b1/s"]= 30.0; c ["b1/p"]= 22.5;
        c ["b/c"] = 45.0; c ["b/s"] = 24.0; c ["b/p"] = 18.0;
        c ["m/c"] = 12.0; c ["m/s"] = 8.00; c ["m/p"] = 8.00;
        c ["A/c"] = 9.00; c ["A/s"] = 7.20; c ["A/p"] = 0 ;
        c ["A2/c"]= 7.60; c ["A2/s"]= 6.20; c ["A2/p"]= 0 ;
        c ["B/c"] = 6.65; c ["B/s"] = 5.45; c ["B/p"] = 0 ;
        c ["C/c"] = 5.15; c ["C/s"] = 4.35; c ["C/p"] = 3.95;
        c ["D/c"] = 3.55; c ["D/s"] = 2.90; c ["D/p"] = 0 ;
        c ["E/c"] = 3.80; c ["E/s"] = 3.05; c ["E/p"] = 0 ;
        c ["F/c"] = 2.65; c ["F/s"] = 2.25; c ["F/p"] = 0 ;
        c ["G/c"] = 2.15; c ["G/s"] = 2.15; c ["G/p"] = 2.15;
    }

    {
        spu = c [ $2 ] # 查找要负但的带宽
        if ( spu == "" || spu == 0 ) {
            summary [ $4 ] = summary [ $4 ] "\n\t" \
                sprintf ( "%4s %4s %7s ? units",\
                    $1, $2, $3 ) \
                " - Bad/Unknown Chargeband"
        } else {
            n = split ( $3, t, ":" ) # 计算以秒为单位的长度
            seconds = 0
            for ( i = 1; i <= n; i++ )
                seconds = seconds*60 + t[i]
            u = seconds / spu    # 计算总的秒数
            if ( int( u ) == u ) # 舍入到的下一个单元
                u = int( u )
            else
                u = int( u ) + 1
            units [ $4 ] += u # 在最后将出信息保存到输出中
            summary [ $4 ] = summary [ $4 ] "\n\t" \
                sprintf ( "%4s %4s %7s %3d units",\
                    $1, $2, $3, u )
        }
    }

END \
    {
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
        for ( i in units ) {                        #对每个人
            printf ( "Summary for %s:", i ) # 从汇总开始加入新行
            print summary [ i ]                     # 打印汇总的细
         节
            # 计算费用
            total = int ( units[i] * pence_per_unit + 0.5 )
            printf ( \
          "Total: %d units @ %.2f pence per unit = $%d.%02d\n\n",
\
                units [i], pence_per_unit, total/100, \
                                total%100 )
        }
    }
```

程序phonebill的注意事项：

这个程序是从一个简单的记录结构中得到综合信息并生成报告的另一个例子。

这个程序的结构也包含三个部分。BEGIN过程中定义了应用与整个程序的变量。当电话公司被通知“向上调整”他们的速率时，使用BEGIN过程可以很容易的完成程序调整。其中的变量是一个大的数组，命名为c，其中的每个元素是每个单元的秒数，使用带宽与速率之比作为这个数组的索引。

主过程读取用户日志文件的每一行。它使用第二个字段，即标识带宽/速率的字段，从数组c中得到一个值。它检查到一个正数值被返回，然后使用$3指定的时间来处理这个值。在这个通话中的所用的单元数被存储在数组units中，数组的索引是用户的名字（$4）。为每个呼叫者累计这个值。

最后，END过程打印出数组units中的值，生成每个呼叫者所用单元的报告以及通话的总费用。

combine—抽取多部分用uuencoded编码技术处理的二进制代码

由Rahul Dhesi提供

在我所编写的所有脚本中，我最引以骄傲的使这个“combine”脚本。

当我在调整*comp.binaries.ibm.pc*时，我想为用户提供一个抽取用uuencoded编码技术（将二进制数据转换成可打印文本）处理二进制代码的简单方法。我在

运维技术书籍免费下载 https://www.linuxprobe.com

每部分添加了一个BEGIN和END标题将编码部分包围起来，并向用户提供下面的脚本：

```
cat $* | sed ' /^END/,/^BEGIN/d'  | uudecode
```

这个程序将接收作为命令行参数的一个文件名（按顺序）列表。它也可以接收连在一起的文章作为标准输入。

这个脚本使用了一个很有用的方法调用cat，shell脚本熟练用户对这种方法很熟悉，但大多数用户没有有效的使用它。这种方法允许用户可以选择是提供命令行参数还是标准输入。

这个脚本调用sed将多余的头和尾剥除，第一个输入文件中的头和最后一个文件中的尾除外。最后结果将是多个输入文件中的相应的编码部分提取出来并编码，每个输入文件（参阅*comp.binaries.ibm.pc*）的格式如下：

```
headers
BEGIN
uuencoded text
END
```

我有很多其他的shell程序代码，但上面这个是最简单的且被成千上万的*comp.binaries.ibm.pc*新闻组读者证明是很有用的

程序combine的注意事项：

这个程序非常简单，但实现了很多目的。这里为那些对这个命令不理解的人做一些解释。usenet新闻组如*comp.binaries.ibm.pc*，提供了公用程序和其他类似的东西。由编译器生成的二进制目标代码，除非被“编码”，否则不能作为新闻文章来发布。程序uuencode将二进制转换为ASCII表示，这样可以很容易地发布。而且，它对新闻文章的长度有限制，大的二进制文件被分割为一系列的文章（例如，分割成三个）。Dhesi将编码的二进制分解为容易处理的块，然后添加BEGIN和END来对包含编码的二进制文本确定界限。

这些文章的读者可能会将每篇文章存储到一个文件中，Dhesi的程序可以自动组合这些文章并删除无关的信息，例如这些文章的标题以及BEGIN和END标题。他的脚本删除从第一个END开始一直到下一个BEGIN模式，并包括这个BEGIN模式之间包含的行。它将所有单独的被编码的传送信息包组合起来并将它们传给uuencode，这个程序将ASCII编码转换为二进制。

用简单的一行脚本就避免了人工编辑的大量工作是一个值得感激的事。

运维技术书籍免费下载 https://www.linuxprobe.com

Mailavg—检查邮箱的大小

由Wes Morgan提供

当调整我们的邮件系统时，我们需要为用户的邮箱每隔30天做一个“快照”。这个脚本简单的计算文件的平均大小并打印用户邮箱的数据分配。

```
#! /bin/sh
#
# 计算/usr/mail中文件的平均大小
#
# 由wes Morgan编写、morgan@engr, 1990年2月2日
ls -Fs /usr/mail | awk '
    { if(NR != 1) {
        total += $1;
        count += 1;
        size = $1 + 0;
        if(size == 0) zercount+=1;
        if(size > 0 && size <= 10) tencount+=1;
        if(size > 10 && size <= 19) teencount+=1;
        if(size > 20 && size <= 50) uptofiftycount+=1;
        if(size > 50) overfiftycount+=1;
        }
    }
    END { printf("/usr/mail has %d mailboxes using %d blocks,",
            count,total)
        printf("average is %6.2f blocks\n", total/count)
        printf("\nDistribution:\n")
        printf("Size Count\n")
        printf(" 0 %d\n",zercount)
        printf("1-10 %d\n",tencount)
        printf("11-20 %d\n",teencount)
        printf("21-50 %d\n",uptofiftycount)
        printf("Over 50 %d\n",overfiftycount)
        }'
    exit 0
```

下面是mailavg输出的一个样本：

```
$ mailavg
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
/usr/mail has 47 mailboxes using 5116 blocks,
average is 108.85 blocks
Distribution:
Size Count
0 1
1-10 13
11-20 1
21-50 5
Over 50 27
```

程序mailavg的注意事项：

这个管理程序和第七章中的filenum程序类似。他处理ls命令的输出结果。

可以将条件表达式“NR != 1”作为一个模式放到主过程的外面。当逻辑结构相同时，用这个表达式作为模式可以清楚的表示这个过程是如何被访问的，使得程序更容易理解。

在以上过程中，Morgan使用一系列条件，来收集对每个用户的邮箱大小的分配统计。

adj—调整文本文件的行

由Norman Joseph提供

（因为作者使用它的程序在发送邮件之前将邮件消息格式化，我们再把它作为实例时保持了其断行和缩进的段落格式。这个程序和**BSD fmt**程序是相似的。）

> 好，我界定接受你的提议。我相信比我更有经验的人，但是我的却有一个我很喜欢的nawk程序，我将它发送出去。
>
> 是的，现在的速度很慢。当我写电子邮件时，我经常对正文做很多修改（尤其是要发送到网站的）。因此在开始就调整好的信或邮件，在我开始增加或删除行时则看上去很混乱。所以最后我花费了很多时间对我的文档进行连接或断行，已得到一个好看的右边界。因此我对自己说，“这只是几个单调乏味的工作，可以用程序来很好的完成。”
>
> 现在，我知道可以用nroff来过滤我的文档并调整行，但是它只默认的（IMHO）处理像这个这样简单的文本。因此，抱着加深自己nawk能力的想法我编写了adj.awk。并同时用shell脚本实现了adj。

下面是nawk过滤器adj的语法：

运维技术书籍免费下载 https://www.linuxprobe.com

adj [-l/c/r/b] [-w n] [-i n] [files ...]

选项为：

-l 行左对齐，右边不规则（默认的）
-c 行居中对齐
-r 行右对齐，左边不规则
-b 左右对齐
-w n 设置行的宽度为n个字符（默认为70）
-i n 设置首行缩进为n个字符（默认为0）

因此，无论何时当我完成这封信（我用的是vi）时，我将执行命令:%!adj －w73(我希望我的行长一些)。所有的行连接或断行都由这个程序来完成，这正是所希望的。缩进和空行被保存，并且在所有的句末的标点符号后添加了两个空格。

这个程序处理了制表符，当计算行的长度时，它将制表符作为一个空格的长度来计算。

值得注意的是这个程序使用命令行参数赋值，而且具备了awk（nawk）的一些新的特点，例如内置match和split函数及其相关支持函数。

```
#! /bin/sh
#
# 调整文件的行
#
# 用法: adj [-l|c|r|b] [-w n] [-i n] [files ...]
#
# options:
# -l - 行左对齐，右边不规则（默认）
# -c - 行居中对齐
# -r - 行右对齐，左边不规则
# -b - 左右对齐
# -w n 设置行的宽为<n>字符（默认为：70）
# -i n 设置首先缩进为<n>字符（默认为：0）
#
# 注意:
#   用-w 加上 -i来设置输出行的宽度
#
# 作者:
# Norman Joseph (amanue!oglvee!norm)
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
adj=l
wid=70
ind=0
set -- `getopt lcrbw:i: $*`
if test $? != 0
then
    printf ' usage: %s [-l|c|r|b] [-w n] [-i n] [files ...]' $0
    exit 1
fi

for arg in $*
do
    case $arg in
    -l) adj=l; shift;;
    -c) adj=c; shift;;
    -r) adj=r; shift;;
    -b) adj=b; shift;;
    -w) wid=$2; shift 2;;
    -i) ind=$2; shift 2;;
    --) shift; break;;
    esac
done

exec nawk -f adj.nawk type=$adj linelen=$wid indent=$ind $*
```

下面是被shell脚本adj调用的adj.nawk脚本：

```
# 用选项调整文本的行
#
# 注意：这个nawk程序从"adj"脚本中调用
# 参见“用法和调用约定”脚本
#
#作者：
# Norman Joseph (amanue!oglvee!norm)
BEGIN {
    FS = "\n"
    blankline = "^[ \t]*$"
    startblank = "^[ \t]+[^ \t]+"
    startwords = "^[^ \t]+"
}
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
$0 ~ blankline {
    if ( type == "b" )
        putline( outline "\n" )
    else
        putline( adjust( outline, type ) "\n" )
   putline( "\n" )
   outline = ""
}

$0 ~ startblank {
    if ( outline != "" ) {
        if ( type == "b" )
            putline( outline "\n" )
        else
            putline( adjust( outline, type ) "\n" )
}

firstword = ""
i = 1
while ( substr( $0, i, 1 ) ~ "[ \t]" ) {
    firstword = firstword substr( $0, i, 1 )
    i++
}
inline = substr( $0, i )
outline = firstword

nf = split( inline, word, "[ \t]+" )

for ( i = 1; i <= nf; i++ ) {
    if ( i == 1 ) {
        testlen = length( outline word[i] )
    } else {
        testlen = length( outline " " word[i] )
            if ( match( ".!?:;", "\\" substr( outline,
                length( outline ), 1 )) )
            testlen++
    }

    if ( testlen > linelen ) {
        putline( adjust( outline, type ) "\n" )
        outline = ""
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
    }

    if ( outline == "" )
        outline = word[i]
    else if ( i == 1 )
        outline = outline word[i]
    else {
        if ( match( ".!?:;", "\\" substr( outline,
                length( outline ), 1 )) )
            outline = outline " " word[i] # 2 个空格
        else
            outline = outline " " word[i] # 1 个空格
        }
    }
}

$0 ~ startwords {
    nf = split( $0, word, "[ \t]+" )

    for ( i = 1; i <= nf; i++ ) {
        if ( outline == "" )
            testlen = length( word[i] )
        else {
            testlen = length( outline " " word[i] )
            if ( match( ".!?:;", "\\" substr( outline,
                    length( outline ), 1 )) )
                testlen++
        }

        if ( testlen > linelen ) {
            putline( adjust( outline, type ) "\n" )
            outline = ""
        }

        if ( outline == "" )
            outline = word[i]
        else {
            if ( match( ".!?:;", "\\" substr( outline,
                    length( outline ), 1 )) )
                outline = outline " " word[i] # 2 个空格
            else
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
                outline = outline " " word[i] # 1 个空格
        }
    }
}

END {
    if ( type == "b" )
        putline( outline "\n" )
    else
        putline( adjust( outline, type ) "\n" )
}

#
# -- 支持函数 --
#

function putline( line, fmt )
{
    if ( indent ) {
        fmt = "%" indent "s%s"
        printf( fmt, " ", line )
    } else
        printf( "%s", line )
    }

function adjust( line, type, fill, fmt )
{
    if ( type != "l" )
        fill = linelen - length( line )
    if ( fill > 0 ) {
        if ( type == "c" ) {
            fmt = "%" (fill+1)/2 "s%s"
            line = sprintf( fmt, " ", line )
        } else if ( type == "r" ) {
            fmt = "%" fill "s%s"
            line = sprintf( fmt, " ", line )
        } else if ( type == "b" ) {
            line = fillout( line, fill )
        }
    }
    return line
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
}

function fillout( line, need, i, newline, nextchar, blankseen )
{
    while ( need ) {
        newline = ""
        blankseen = 0
        if ( dir == 0 ) {
            for ( i = 1; i <= length( line ); i++ ) {
                nextchar = substr( line, i, 1 )
                if ( need ) {
                    if ( nextchar == " " ) {
                        if ( ! blankseen ) {
                            newline = newline " "
                            need--
                            blankseen = 1
                        }
                    } else {
                        blankseen = 0
                    }
                }
                newline = newline nextchar
            }

        } else if ( dir == 1 ) {
            for ( i = length( line ); i >= 1; i-- ) {
                nextchar = substr( line, i, 1 )
                if ( need ) {
                    if ( nextchar == " " ) {
                        if ( ! blankseen ) {
                            newline = " " newline
                            need--
                            blankseen = 1
                        }
                    } else {
                        blankseen = 0
                    }
                }
                newline = nextchar newline
            }
        }
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
            line = newline
            dir = 1 - dir
        }
        return line
    }
```

程序adj的注意事项

这个小的文本格式程序对于应用文本编辑器的用户来说是一个很好的程序。它允许你设置行的最大宽度和调整段落，因此可以用来格式化邮件消息或简单的信件。

adj的shell脚本设置所有的选项，尽管这个以在BEGIN过程中通过读取ARGV来完成。使用shell建立命令行参数对那些熟悉shell的用户来说可能更容易。

没有对adj.awk脚本代码进行注释使它读起来比其他程序更困难。BEGIN过程将三个正则表达式赋给变量：blankline，startblank和startwords。这是一个好的技术（这是用于在lex规范中德技术），因为阅读正则表达式比较困难，而通过变量名可以知道它匹配的是什么。记住，现代的awk允许你在变量中将正则表达式作为一个字符串。

这个程序包括三个主过程，它们可以用它们匹配的变量来命名。第一个是blankline，用于处理当遇到一个空行时收集的文本。第二个是startblank，用于处理一空白字符（空格或制表符）开始的行。第三个是startwords，用于处理文本行。基本过程是读取一行文本并确定在这行中要填充多少个单词，给出行的宽度，输出将要填充的那些单词并存贮将不在变量outline中的那些单词。当读取下一个输入行时，putline的内容必须在那行输出前被输出。

函数adjust()根据命令行选项设置的格式类型来调整文本。除了类型“l”（左对齐，右边不规则）外其他所有的都必须被填充。因此，这个函数要做的第一件事是通过从指定的行长度中减去当前行的长度来计算需要“填充”多少。它恰当地使用了sprintf()函数对文本定位。例如，要对文本剧中，fill的值（加一）被2整除来决定在行的每边需要填充的数量。这个数值被作为sprintf()的参数传递给变量fmt。

```
fmt = "%" (fill+1)/2 "s%s"
line = sprintf( fmt, " ", line )
```

因此，用空格来填充一个字段，这是需要填充的长度的一半。

如果文本时右对齐的，用fill的值来填充这个字段。最后，如果格式类型为“b

运维技术书籍免费下载 https://www.linuxprobe.com

（块）”，那么调用函数fillout来确定在行的哪个位置添加空格

再次仔细观察这个程序的设计，可以看出函数的用途是如何有助于理解程序的。有助于了解主过程如何用于控制程序的输入流，而子过程用于对输入进行处理。将“处理”和控制流分开有助于程序的可读性和可维护性。

在过去，我们不知道为什么字段分隔符FS在BEGIN过程中被设置为换行符。这就意味着字段和记录分隔符是相同的（也就是说，$0和$1是相同的）。函数split()用于使用制表符或空格作为分隔符将行分隔为字段。

```
nf = split( $0, word, "[ \t]+" )
```

字段分隔符能够被设置为相同的正则表达式，如下所示：

```
FS = "[ \t]+"
```

使用默认的字段分割将会更有效。

最后，使用函数match()查找标点符号是无效的，使用index()将会更好。

readsource—将程序源文件格式化为troff格式

由Martin Weitzel提供

我经常准备技术文档，尤其是为课程和辅导。在这些文档中，我经常需要打印不同的源文件（C程序，awk程序，shell脚本和makefiles）。问题是这些源文件经常随时间而改变，而打印时总希望用最新的版本。我也希望避免做排字工。

当我使用troff做文本处理时，将源文件包含在文本中是很容易的。但有些字符（尤其是当“ ”、“.”和“,”出现在一行的开始部位时）必须转义以阻止被troff解释。

我经常从源文件中摘录而不用完整的文件。同时也需要一个机制来设置分页符。是的，也许我是一个完美主义者，但我不希望看到一个C程序打印满一页，而在下一页中只有剩余的两行。因为我经常修改文档，我不能查询到一个“好的”分页符—这必须自动完成。

为了解决这些问题，我编写了一个过滤器来预处理源文件，以将其包含在troff

运维技术书籍免费下载 https://www.linuxprobe.com

文本中。我这封信发送的就是这个awk程序。（他没有给我这个程序的名字，这里就命名为readsource。）

使用makefiles整个过程将更自动化。我将源文件的预处理版本放入到troff文档中，并且根据这些预处理文件制定了格式。而这些文件又依赖于他们的原始格式，因此如果我“安排”文档来打印，预处理源文件将检查他们是否仍为当前文件；否则将通过它们的原始文件来生成新文件。

程序用注释的形式给出了一个完整描述。但是这些描述对我来说太多而对其他人则不然，我将给你一些线索。基本上，这个程序简单地监视一些字符，例如“\”被转变为“\e”，并在每行的前面写上“/&”。制表符被扩展为空格（它有一个转换），你可以在每行前面生成行的编号（可选的转换）。这些编号的格式可以用一个环境变量来设置。

如果你希望只处理文件的一部分，你可以用两个正则表达式（用另外一个开关）来选择这部分。你必须标识要包含第一行或者不包含第一行。我发现这是很实用的：如果你只想显示C程序的一个特定的函数，你只需要给出这个函数定义的第一行和下一个函数定义的第一行。如果这个源文件改变了，在两个函数之间插入新的函数和这两个函数的次序改变了，那么匹配模式将无法工作。但这更适合于一个程序中经常有小的改变。

最后一个特点是正确的得到分页符，这有点复杂。这里使用了一个技术，我称它为“你可以在这里中断”。那些点被标记为特殊的行（我在C程序中用“/*!”，在awk、nawk，shell和makefiles等中用“#!”）。这些点是如何标记的并不重要，你可以用你自己的约定，但它必须使一个正则表达式能恰好匹配这种行而不是其它的行（举例来说，如果你的源文件中的空行都可以解释为分页符，那么你就可以很简单的将这个方面贯彻下去，而所要做的仅仅是编写匹配空行的正则表达式）。

在所有标记之前，一个特殊的序列将被插入，这又是由一个环境变量给出的。通过troff，在我包含哪些预处理文本之前，我使用打开“显示”（.DS）的技术，并在任何我想接受分页符的地方插入了一个关闭（.DE）和重新打开（.DS）显示。完成这以后，troff将收集尽可能多的行以填充到当前页上。我猜测对其他文本处理器，存在相应的合适的技术。

```
#! /bin/sh
# Copyright 1990 by EDV-Beratung Martin Weitzel, D-6100 Darmstadt
#
================================================================
# PROJECT: Printing Tools
# SH-SCRIPT: Source to Troff Pre-Formatter
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
#
==================================================================
#!
# ------------------------------------------------------------------
-
# 这个程序是执行源文件的一个工具，所以它们能
# 被包含在nroff/troff-input中，当在nroff/troff-input中
# 包含任意的文件时将会在行中出现问题
# 如果有修改，以点（.）或单引号(’)，或与其
# 相关的字符，此外从嵌入的反斜杠开始。
# 当修改源文件上面的问题一个也不会出现时
# 可以处理一些其他的事
# 包括行的个数及选择有趣的部分
# ------------------------------------------------------------------
-
#!
# USAGE="$0 [-x d] [-n] [-b pat] [-e pat] [-p pat] [file ...]"
#
# 提要:
# 下面是提供的选项:
# -x d    将制表符扩展为 "d" 个空格
# -n      源行的个数(参看: NFMT)
# -b pat  从包含"pat"的行开始输出
#         其中包含这一行（默认开头）
# -e pat  在包含“pat”的行结束输出
#         其中不包括这一行（默认在结尾）
# -p pat  在包含“pat”的行的前面
#         可能有分页符（默认没有分页符）
# "pat"   是awk提供的“扩展正则表达式“
# 下面是环境中使用的变量:
# NFMT specify format for line numbers (Default: see below)
# PBRK string, to mark page breaks. (Default: see below)
#!
# 预处理:
# 普通的unix环境，包括 awk.
#
# 警告:
# "pat"在使用前没有被测式过
# （在发现问题前，已经开始处理).
# 如果使用-n选择，"NFMT"必须
# 包含一个%d格式的说明符
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
# 在 "NFMT" 和"PBRK"中，其中包含的双引号必须在前面
# 加上反斜杠作为转义符.
# 在"pat"、 "NFMT" 和 "PBRK" 中嵌入的制表符的换行符必须写为\t和\n
# 反斜杠要想被输入也必须用两个
# （后者只在一些特殊情况下是需要的，但对其他情况也没有伤害）
#
#!
# BUGS:
# 但可能被作为一个更愉快实现的原型.
# 现在处理反斜杠的代价是很昂贵的
# 而这也可以用sed 的s/\\/\\e/g’来处理。
# 但是制表符的扩展是更困难的，因为我无法想像用sed
# 如何来处理它。如果你不需要制表符的扩展，你可以修改程序
# 另外一个选择是
# 利用gsub()这些是程序的环境限制于nawk中。
#
# 如果有其他的错误，请发信给我
#!
# 作者: Martin Weitzel, D-6100 DA (martin@mwtech.UUCP)
#
# 发行: 25. Nov 1989, Version 1.00
# ----------------------------------------------------------------
-
#! CSOPT
# ----------------------------------------------------------------
-
# 测式/设置选项
# ----------------------------------------------------------------
-

xtabs=0 nfmt= bpat= epat= ppat=
for p
do
case $sk in
1) shift; sk=0; continue
esac
case $p in
-x) shift;
    case $1 in
    [1-9]|1[0-9]) xtabs=$1; sk=1;;
    *) { >&2 echo "$0: bad value for option -x: $1"; exit 1; }
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
    esac
    ;;
-n) nfmt="${NFMT:-<%03d>\•}"; shift ;;
-b) shift; bpat=$1; sk=1 ;;
-e) shift; epat=$1; sk=1 ;;
-p) shift; ppat=$1; sk=1 ;;
--) shift; break ;;
*) break
esac
done

#! MPROC
# ----------------------------------------------------------------
-
# 下面是 "真正的起作用的工作"
# ----------------------------------------------------------------
-
awk '
#.为制表符扩展，分页符和选择做准备
BEGIN {
    if (xt = ' $xtabs' ) while (length(sp) < xt) sp = sp " ";
    PBRK = "' "${PBRK-' .DE\n.DS\n' }"' "
    ' ${bpat:+'  skip = 1; ' }'
} #! 限制选择范围
{
    ' ${epat:+'  if (!skip && $0 ~ /' "$epat"' /) skip = 1; ' }'
    ' ${bpat:+'  if (skip && $0 ~ /' "$bpat"' /) skip = 0; ' }'
    if (skip) next;
}
#! 按要求处理一个输入行
{
    line = ""; ll = 0;
    for (i = 1; i <= length; i++) {
        c = substr($0, i, 1);
        if (xt && c == "\t") {
            # expand tabs
            nsp = 8 - ll % xt;
            line = line substr(sp, 1, nsp);
            ll += nsp;
        }
        else {
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
            if (c == "\\") c = "\\e";
            line = line c;
            ll++;
        }
    }
}
#! 最后打印该行
{
    ' ${ppat:+'  if ($0 ~ /' "$ppat"' /) printf("%s",
PBRK); ' }'
    ' ${nfmt:+'  printf("' "$nfmt"' ", NR) ' }'
    printf("\\ & %s\n", line);
}
,$*
```

要用实例测试它是如何工作的，我们运行readsource来提取它自己的一部分程序。

```
$ readsource -x 3 -b "process one line" -e "finally print"
readsource
\&#! 按要求处理一个输入行
\&{
\&            line = ""; ll = 0;
\&            for (i = 1; i <= length; i++) {
\&                   c = substr($0, i, 1);
\&                   if (xt && c == "\\ et") {
\&                             # 扩展制表符
\&                             nsp = 8 - ll % xt;
\&                             line = line substr(sp, 1, nsp);
\&                             ll += nsp;
\&                   }
\&                   else {
\&                             if (c == "\\e\\e") c = "\\ e\\
ee";
\&                             line = line c;
\&                             ll++;
\&                   }
\&             }
\&}
```


运维技术书籍免费下载 https://www.linuxprobe.com

程序readsource的注意事项

首先，这个程序是非常有用的，因为它能帮助我们准备本书中的列表。作者真正地延展了（旧版的）awk的局限，使用了shell变量向程序传递信息。虽然能解决问题，但很不清楚。

这个程序运行很慢。我们接受了作者的建议，改变了程序中替换制表符和反斜杠的方法。原始程序用的是逐字符的高代价的比较，使用substr()获得字符（即上面的例子提取出来的过程）。它的性能表明了在awk中一次一个字符的读取一行的代价是多么昂贵，其中一些在C中是很简单的。

在readsource上运行它将产生下面的时间（注：英文原文为Running readsource on itself produced the following times.义为用readsource处理其本身的源代码将会产生如下的时间）：

```
$ timex readsource -x 3 readsource > /dev/null
real 1.56
user 1.22
sys 0.20
```

在nawk中使用函数gsub()可以改写这个过程中对制表符和反斜杠的处理方法：

```
#! 按要求处理一个输入行
{
if ( xt && $0 ~ "\t" )
gsub(/\t/, sp)
if ($0 ~ "\\")
gsub(/\\/, "\\e")
}
```

最后一个过程需要做一些小的改变，用“$0”代替变量line（我们不用临时变量line）。nawk版本的结果是：

```
$ timex readsource.2 -x 3 readsource > /dev/null
real 0.44
user 0.10
sys 0.22
```

区别是很明显的。

最后可以通过index()查找反斜杠以提高速度：

运维技术书籍免费下载 https://www.linuxprobe.com

```
#! 按要求处理一个输入行
{
if ( xt && index($0, "\t") > 0 )
gsub(/\t/, sp)
if (index($0, "\\") > 0)
gsub(/\\/, "\\e")
}
```

gent—获得termcap条目

由Tom Christiansen提供

这是一个sed脚本，我用它来提取termcap条目。它可以处理任何和termcap类似的文件，例如，disktab。举例说明：

```
$ gent vt100
```

从termcap中抽取vt100条目，而：

```
$ gent eagle /etc/disktab
```

从disktab中得到一个eagle条目。我现在知道它可以在C或者Perl中完成，但我是在很久以前做的。它向sed脚本传递参数的方式也很有趣。我知道，它本应该被写入sh而不是csh中。

```
#!/bin/csh -f

set argc = $#argv

set noglob
set dollar = ' $'
set squeeze = 0
set noback="" nospace=""

rescan:
    if ( $argc > 0 && $argc < 3 ) then
        if ( "$1" =~ -* ) then
            if ( "-squeeze" =~ $1* ) then
                set noback=' s/\\//g'  nospace=' s/^[ ]*//'
                set squeeze = 1
                shift
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
                @ argc --
                goto rescan
            else
                echo "Bad switch: $1"
                goto usage
            endif
        endif

        set entry = "$1"
        if ( $argc == 1 ) then
            set file = /etc/termcap
        else
            set file = "$2"
        endif
    else
        usage:
        echo "usage:  `basename $0 ` [-squeeze] entry
     [termcapfile]"
        exit 1
    endif

sed -n -e \
"/^${entry}[|:]/ {\
    :x\
    /\\${dollar}/ {\
    ${noback}\
    ${nospace}\
    p\
    n\
    bx\
    }\
    ${nospace}\
    p\
    n\
    /^ / {\
    bx\
    }\
    }\
/^[^ ]*|${entry}[|:]/ {\
    :y\
    /\\${dollar}/ {\
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
${noback}\
${nospace}\
p\
n\
by\
}\
${nospace}\
p\
n\
/^ / {\
by\
}\
}" < $file
```

程序gent的注意事项

当你习惯了阅读awk脚本时，他们看起来好像除了最简单的sed脚本外，比任何其他的程序都容易理解。从上面所显示的这个小的sed程序来断定出它正在处理什么是一项艰苦的任务。

这个脚本展示了如何将shell变量传递给一个sed脚本。变量用于将可选的sed命令传递到程序中，与替换反斜杠和空格的替换命令一样。

这个脚本可以用几种方法简化。第一，用两个正则表达式来提取条目似乎是没有必要的。第一个匹配在行的开始处的条目，第二个匹配行中任何其他地方的条目。即使这两个正则表达式都是必要的，标记为x和y的循环也是相同的，我们可以使它们转移到相同的循环中。

plpr—行式打印的预处理器

由Tom Van Raalte提供

我想你可能会将下面的脚本用于办公。他是一个行式打印的预处理器，用于将输出发送到“最好的”打印机（这个shell脚本是为BSD或LINUX系统编写的，你可以在使用lpr的地方使用这个命令。它读取lpq命令的输出结果，以确定指定的打印机是否可用。如果不能用，它测试一系列的打印机，以确定其中那个是可用的或哪个是最不忙的。然后调用lpr将这个作业送给该打印机）。

```
#!/bin/sh
#
#设置临时文件
TMP=/tmp/printsum.$$
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
LASERWRITER=${LASERWRITER-ps6}
#检查看看默认的打印机是否空闲?
#
#
FREE=‘lpq -P$LASERWRITER | awk ’
{
    if ($0 == "no entries")
    {
        val=1
        print val
        exit 0
    }
    else
    {
        val=0
        print val
        exit 0
    }
}’ ‘
#响应 Free 是 $FREE
#
#如果默认打印机空闲，那么设置$FREE.并打印和退出
#
if [ $FREE -eq 1 ]
then
    SELECT=$LASERWRITER
#响应所选择的 $SELECT
    lpr -P$SELECT $*
    exit 0
fi
#响应完成退出
#
#现在，继续看看哪个打印机空闲。
#
BANK=${BANK-$LASERWRITER}
#响应 bank 是 $BANK
#
#如果 BANK 是LASERWRITER相同，那么我们没有选择。
#否则，如果有打印机空闲，就打印它
#
if [ "$BANK" = "$LASERWRITER" ]
then
    SELECT=$LASERWRITER
    lpr -P$SELECT $*
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
    exit 0
fi
#响应通过测式 bank=laserprinter
#
#现在我们检查一个空闲的打印机
#注意：当测式时，$LASERWRITER 为空闲，要对它进行重新测式
#during the check.
#
#响应现在我们检查其他的设置是否为空闲的
for i in $BANK $LASERWRITER
do
FREE=‘lpq -P$i | awk ’
{
    if ($0 == "no entries")
    {
        val=1
        print val
        exit 0
    }
    else
    {
        val=0
        print val
        exit 0
    }
}’ ‘
if [ $FREE -eq 1 ]
then
# 响应在循环中用 $i
    SELECT=$i
# 响应选择 $SELECT
# 如果 [ "$FREE" != "$LASERWRITER" ]
# 那么
# 响应“输出重定向到打印机$i”
# fi
lpr -P$SELECT $*
exit 0
fi
done
#响应为某个空闲的设备进行了检查
#
#如果执行到这里，则说明没有空闲的打印机
#所以我们向打印机打印最小的字节队列
#
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
#
for i in $BANK $LASERWRITER
do
    val= `lpq -P$i | awk '  BEGIN {
        start=0;
    }
/^Time/ {
        start=1;
        next;
    }
    (start == 1){
        test=substr($0,62,20);
        print test;
    } '  | awk '

    BEGIN {
        summ=0;
    }
    {
        summ=summ+$1;
    }

    END {
        print summ;
    }'  `

    echo "$i $val" >> $TMP
done

SELECT= `awk ' (NR==1) {
    select=$1;
    best=$2
}

($2 < best) {
    select=$1;
    best=$2
}

END {
    print select
}

'  $TMP  `
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
#echo $SELECT
#
rm $TMP
#现在在选定的打印机上打印
#如果[ $SELECT != $LASERWRITER ]
#那么
# 响应"输出重定向到打印机$i"
#fi
lpr -P$SELECT $*
trap ' rm -f $TMP; exit 99'  2 3 15
```

程序plpr的注意事项

多数情况下，我们避免使用这样的脚本，其脚本中的多数逻辑结构是用shell脚本来编写的。然而，这种最小的处理方法是awk多种应用中的一个典型方式。这里调用awk只处理那些shell脚本不能完成（或者简单完成）的事情。处理一个命令的输出并做数字比较就是这些任务的一个例子。

另外应注意，在末尾处的trap语句应在脚本的起始位置，而不是在底部。

transpose—实现矩阵的转置

由Geoff Clare提供

程序transpose的任务是对输入矩阵进行转置。我编写这个程序是因为当我看到了寄给Net的一个处理这个工作的程序，但它的效率低的可怕。我基础的这个程序在时间上相比比那个程序有了改善。如果我没有记错，那个原始版本将所有的元素单独存储，并在嵌套的循环中用printf语句打印每个元素。我立即意识到构造被转置的矩阵的行是比较快的，“像是坐飞机”。

我的脚本使用${1+"$@"}在awk命令行中给出文件名，如果没有提供文件名，那么awk将从标准输入中读取。这个方式比纯粹的$*要好，因为后者不能处理包含空白字符的文件名。

```
#! /bin/sh
# 一个矩阵的转置：假设所有的行都有相同数量的字段
# of fields
exec awk '
NR == 1 {
    n = NF
    for (i = 1; i <= NF; i++)
        row[i] = $i
    next
}
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
{
    if (NF > n)
        n = NF
    for (i = 1; i <= NF; i++)
        row[i] = row[i] " " $i
}
END {
    for (i = 1; i <= n; i++)
        print row[i]
}'  ${1+"$@"}
```

下面是测试文件：

```
1 2 3 4
5 6 7 8
9 10 11 12
```

现在我们使用这个文件运行transpose。

```
$ transpose test
1 5 9
2 6 10
3 7 11
4 8 12
```

程序transpose的注意事项

这是一个非常简单而有趣的脚本。他建立了一个名为row的数组，并将每个字段追加到这个数组的一个元素中。END过程输出了这个数组。

m1—简单的宏处理器

有Jon Bentley提供

程序m1可是在UNIX系统中找到的m4宏处理器的“小兄弟”。它首先发表在文章m1: A Mini Macro Processor，1990年6月，《Computer Language》(第七卷)，编号6，47-61页中。这个程序是因为Ozan Yigit才引起我的注意，Jon Bentley将这个程序的最新版本寄给了我，以及他的文章的初稿（我得到已发表文章的副本有些困难）。这篇论文的后记中包含一个例子程序可以在O’Reilly的FTP服务器上应用（参见前言）。我写了如下的介绍和程序的注意事项[A.R.]。

一个宏处理器将它的输入复制到它的输出，其中要完成几项工作。其任务是：

运维技术书籍免费下载 https://www.linuxprobe.com

1. 定义和扩展宏。宏包含两部分，名字和主体。在宏的名字出现的地方都用它的主体代替
2. 包含文件。在数据文件中的特殊包含命令被命名文件的内容取代。包含通常是可以嵌套的，一个包含文件可以包含另一个文件。被包含文件被宏处理。
3. 有条件的包含和排除文本。文本的不同部分可以被包含在最终的输出中，这通常根据是否定义了宏。
4. 基于宏处理器，能够出现的注释行将被从最终的输出中删除。

如果你是一个C或C++程序员，你对这些语言的内置处理器已经很熟悉。UNIX系统有一个通用的宏处理器，被称为m4。这是一个功能强大的程序，但是不太容易掌握，因为宏定义是在定义时被扩展的，而不是在运行时实现扩展的。m1比m4简单的多，很容易学习和应用。

以下是Jon在一个非常简单的宏处理器中的第一个剪切块。他所做的就是定义和扩展宏。我们将它命名为m0a。在这部分和后面的程序中，“at”符号（@）被用来区别被指示的行，并且也标识存在应该扩展的宏。

```
/^@define[ \t]/ {
    name = $2
    $1 = $2 = ""; sub(/^[ \t]+/, "")
    symtab[name] = $0
    next
}
{
    for (i in symtab)
        gsub("@" i "@", symtab[i])
    print
}
```

这个版本寻找以“@define”开始的行。关键词是$1，宏名设置为$2。行的剩余部分是宏的主体。下一个输入行使用next获取。第二个规则简单地遍历所有已定义的宏，在输入行用宏的主体对每个宏做全局替换，然后打印这行。请仔细权衡这个版本的简单性与程序运行时间。

下一版本（m0b）添加了文件包含功能：

```
function dofile(fname) {
    while (getline <fname > 0) {
        if (/^@define[ \t]/) {          # @define name 的值
            name = $2
            $1 = $2 = ""; sub(/^[ \t]+/, "")
            symtab[name] = $0
        } else if (/^@include[ \t]/)    # @include 文件名
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
            dofile($2)
        else {                          # 在行 @name@的任意位置
            for (i in symtab)
                gsub("@" i "@", symtab[i])
            print
        }
    }
    close(fname)
}
BEGIN {
    if (ARGC == 2)
        dofile(ARGV[1])
    else
        dofile("/dev/stdin")
}
```

注意，通过递归调用dofile()来处理嵌套包含文件。

介绍了以上实现方法后，下面是m1程序的全部代码。

```
#! /bin/awk -f
# 名字
#
# m1
#
# 用法
#
# awk -f m1.awk [file...]
#
# 描述
#
# M1将它的输入文件无改变地复制到输出，除非
# 被某个“宏表达式“更改了。下面的行为后续的处理定义宏:
# @comment Any text
# @@                      和@comment相同
# @define name value
# @default name value     如果没有定义名字设设置
# @include filename
# @if varname             如果 varname != 0将包含后续文本
# @unless varname         如果 varname == 0将包含后续文件
# @fi                     将@if 或@unless终止
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
# @ignore DELIM          忽略输入直到DELIM开始的行
# @stderr stuff          将结果送到标准错误文件
#
# 可以在每行的末尾，用反斜杠将定义扩展到多行，
# 因此引用了后面的换行
#
# 在输入中出现@name@的地方，在输出中
# 用相应的值代替
#
# 在行的开始处为@name被看做与 @name@相同
#
# BUGS
#
# M1比 M4少了3步. 你可能将漏掉一些
# 你所期望的内容
#
# 作者
#
# Jon L. Bentley, jlb@research.bell-labs.com
#

function error(s) {
    print "m1 error: " s | "cat 1>&2"; exit 1
}

function dofile(fname, savefile, savebuffer, newstring) {
    if (fname in activefiles)
        error("recursively reading file: " fname)
    activefiles[fname] = 1
    savefile = file; file = fname
    savebuffer = buffer; buffer = ""
    while (readline() != EOF) {
        if (index($0, "@") == 0) {
            print $0
        } else if (/^@define[ \t]/) {
            dodef()
        } else if (/^@default[ \t]/) {
            if (!($2 in symtab))
                dodef()
        } else if (/^@include[ \t]/) {
            if (NF != 2) error("bad include line")
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
                dofile(dosubs($2))
        } else if (/^@if[ \t]/) {
                if (NF != 2) error("bad if line")
              if (!($2 in symtab) || symtab[$2] == 0)
                  gobble()
        } else if (/^@unless[ \t]/) {
                if (NF != 2) error("bad unless line")
                if (($2 in symtab) && symtab[$2] != 0)
                    gobble()
        } else if (/^@fi([ \t]?|$)/) { # 可以在这里进行错误检查
        } else if (/^@stderr[ \t]?/) {
            print substr($0, 9) | "cat 1>&2"
        } else if (/^@(comment|@)[ \t]?/) {
        } else if (/^@ignore[ \t]/) { #将输入切断，直到 $2
            delim = $2
            l = length(delim)
            while (readline() != EOF)
                if (substr($0, 1, l) == delim)
                    break
        } else {
            newstring = dosubs($0)
             if ($0 == newstring || index(newstring, "@") == 0)
                print newstring
            else
                buffer = newstring "\n" buffer
        }
    }
    close(fname)
    delete activefiles[fname]
    file = savefile
    buffer = savebuffer
}

# 将下一个输入行放到全局字符串"buffer"中
# 返回"EOF" 或"" (空字符串)

function readline( i, status) {
    status = ""
        if (buffer != "") {
            i = index(buffer, "\n")
            $0 = substr(buffer, 1, i-1)
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
            buffer = substr(buffer, i+1)
        } else {
            # 没有v10的特殊情况下: 如果 (file == "/dev/stdin")
            if (getline <file <= 0)
                status = EOF
        }
        # Hack: 允许@Mname在以@结尾的行 w/o 开头
        if ($0 ~ /^@[A-Z][a-zA-Z0-9]*[ \t]*$/)
            sub(/[ \t]*$/, "@")
        return status
}

function gobble( ifdepth) {
    ifdepth = 1
    while (readline() != EOF) {
        if (/^@(if|unless)[ \t]/)
            ifdepth++
        if (/^@fi[ \t]?/ && --ifdepth <= 0)
            break
    }
}

function dosubs(s, l, r, i, m) {
    if (index(s, "@") == 0)
        return s
    l = ""    # 当前位置的左边; 准备输出
    r = s     # 当前位置的右边; 这次检验
    while ((i = index(r, "@")) != 0) {
        l = l substr(r, 1, i-1)
        r = substr(r, i+1) # 扫描当前的 @
        i = index(r, "@")
        if (i == 0) {
            l = l "@"
            break
        }
        m = substr(r, 1, i-1)
        r = substr(r, i+1)
        if (m in symtab) {
            r = symtab[m] r
        } else {
            l = l "@" m
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
                r = "@" r
            }
        }
        return l r
    }

    function dodef(fname, str, x) {
        name = $2
sub(/^[ \t]*[^ \t]+[ \t]+[^ \t]+[ \t]*/, "") # 旧 BUG: 后边的 * 是+
        str = $0
        while (str ~ /\\$/) {
            if (readline() == EOF)
                error("EOF inside definition")
            x = $0
            sub(/^[ \t]+/, "", x)
            str = substr(str, 1, length(str)-1) "\n" x
        }
        symtab[name] = str
    }

    BEGIN { EOF = "EOF"
        if (ARGC == 1)
            dofile("/dev/stdin")
        else if (ARGC >= 2) {
            for (i = 1; i < ARGC; i++)
                dofile(ARGV[i])
        } else
            error("usage: m1 [fname...]")
    }
```

程序m1的注意事项

这个程序是一个很好的模板，包含的error()函数和第十一章“awk的系列产品”中的介绍的相似，且每个任务被明确地分割为单独的函数。

主程序在底部的BEGIN过程中运行。如果没有参数，他只处理标准输入或所有在命令行上指定的文件。

高级处理发生在函数dofile()中，它每次读取一行，并决定如何处理每一行。数组activefiles跟踪每个打开的文件。变量fname标识读取数据的当前文件。当遇到“@include”指示时，dofile()只简单对新文件递归调用自己，和m0b一

运维技术书籍免费下载 https://www.linuxprobe.com

样。有趣的是，首先对被包含的文件名进行宏处理。仔细阅读这个函数，这有一些好的技巧。

函数readline()处理“回推”。扩展一个宏之后，宏处理器检测最新创建的文本中的任何附加宏名。只有当处理完所有的扩展文本并送到输出后，这个程序才从输入中获取一个新行。

函数dosubs()实际执行宏的转换。他从左向右处理行，并用宏的主体代替宏的名字。对新行的重新扫描将在较高层逻辑构件中由readline()和dofile()来管理。这个版本比用在m0程序中的方法要更有效率。

最后，函数dodef()处理宏的定义。它将$2中的宏名字保存起来，然后用sub()删除最前面的两个字段。现在$0的新值只包含（第一行）宏的主体。《Computer Language》中的内容解释了使用sub()的目的，即为了保护宏主体中的空白字符。简单的将空字符串赋值给$1和$2将重建这个记录，但所有出现空白字符的地方都被OFS（一个空行）的值代替。函数然后继续收集剩下的宏，直到遇到以“\”结尾的行为止。这是对m0的附加改进：宏的主体可以不止一行。

这个程序的剩余部分用于处理文本的条件包含或排除，这部分很简单。比较好的是这些条件可以被互相嵌套。

m1是宏处理器的一个好的开端。你或许想了解如何对它进行扩展，例如，允许在宏处理器中包含“@else”条件；通过命令行处理宏定义、“未定义”的宏，以及宏处理器通常处理的其他事情。

Jon Bentley建议可以做的其他扩展：

1. 添加“@shell DELIM shell line here”，这将读取输入行到“SELIM”，并将扩展的输出通过一个管道传递给指定的shell命令。
2. 增加命令“@longdef”和“@longend”。这些命令可以定义长主体的宏，也就是，这包含多行的宏主体，简单化dodoef()中的逻辑结构
3. 增加“@append MacName MoreText”，和troff中的“.am”一样。在troff中的这种宏将文本追加到已定义的宏中。在m1中者可以用于扩展已经定义的宏的主体。
4. 避免使用V10/dev/stdin特殊文件（注：英文原文V10与/dev/stdin之间有一空格）。贝尔实验室的UNIX系统（和一些其他的UNIX系统）包含一个特殊的文件，名为/dev/stdin，用于访问标准输入。我们正好可以用“-”来实现这个技巧。如果你用的是gawk或Bell Labs awk，这是不成问题的，它们能解释这个特殊的文件名/dev/stdin。（见第十一章）

运维技术书籍免费下载 https://www.linuxprobe.com

最后注意一点，Jon在他的两本书中--《Programming Pearls》和《More Programming Pearls- Confessions of a Coder》（两本书都被Addison-Wesley出版），均使用了awk，这些书都是很值得一读的。

运维技术书籍免费下载 https://www.linuxprobe.com

附录一
Sed 的快速参考

命令行语法

调用 sed 的语法有两种形式：

```
sed [-n][-e] 'commmand' file(s)
sed [-n] -f scriptfile file(s)
```

第一种形式允许在命令行指定一个编辑命令，用单引号括起来。第二种形式允许指定一个 scriptfile,即包含 sed 命令的文件。两种形式可以一起使用，并且它们可以被多次使用。编辑的结果是将命令和脚本文件串联起来。

下面是可识别的选项：

-n
    仅打印用 p 命令或 s 命令的 p 标记指定的行。
-e cmd
    下一个参数是编辑命令。当指定多个脚本时很有用。
-f file
    下一个参数是一个包含编辑命令的文件。

如果脚本的第一行是“#n”,sed 将按-n 指定的方式工作。

频繁使用的 sed 脚本通常是通过 shell 脚本来调用的。这对 sed 和 awk 来说是相同的，参见附录二“awk 的快速参考”中的“用 shell 实现调用 awk”一节。

sed 命令的语法

sed 命令的普通形式为：

[addr ess[,addr ess]][!]command [ar guments]

sed 将每个输入行拷贝到一个模式空间。sed 指令由地址和编辑命令组成。如果命令的地址和模式空间中的行匹配，那么这个命令将被应用于匹配行。如果一个命令没有地址，那么它被应用于每个输入行。如果一个命令改变了模式空间的内容，后续的命令地址将被应用于模式空间中的当前行，而不是原始的输入

运维技术书籍免费下载 https://www.linuxprobe.com

行。

模式寻址

地址可以是一个行号或是由斜杠包含着的一个模式（/parttern/）.模式是用正则表达式描述的。另外，\n 可以用来与模式空间（N 命令的结果）的任意换行符匹配，但模式空间底部的换行符除外。

如果没有指定模式，相应的命令将被应用于所有的行。如果只指定了一个地址，那么相应的命令将被应用于和这个地址匹配的行。如果指定了两个用逗号分隔的地址，这个命令将被应用于位于第一个和第二个地址范围之间的所有行。一些命令只接受一个地址，包括 a,i,r,q 和=。

在地址后面的！操作符使 sed 将相应的命令作用于所有与该地址不匹配的行。

大括号（{}）被 sed 应用于地址的嵌套或对同一个地址应用多个命令。

```
[/patter n/[,/patter n/]]{
command1
command2
}
```

左大括号必须在一行的末尾，而右大括号必须单独在一行。确保大括号后面没有空格。

sed 中的正则表达式元字符

下面的表列出了在第三章“了解正则表达式语法“中介绍的模式匹配元字符。

注意，空正则表达式“//”表示和前面的正则表达式一样。

表 A-1：模式匹配元字符

.	匹配除换行符以外的任意单个字符
*	匹配任意个（包括 0 个）在它前面的字符（包括由正则表达式指定的字符）
[…]	匹配广括号中的字符类中的任意一个字符。其他所有的元字符在被指定为类中的成员时都会失去它们原来的含义。如果方括号第一个字符为脱字符（^）则匹配除了换行符和类中列出的那些字符以外的所有字符。连字符(-)用天表示字符的范围。如果类中的第一个字符为右方括号(])则表示它是这个类的成员
\{n,m\}	它前面的某个范围内单个字符出现的次数（包括正则表达式指定的字符）。\{n\}将匹配 n 次出现，\{n,\}匹配 n 和 m 之间的任意次出现。（仅限于 sed 和 grep）
^	定位位于行起始位置后面的正则表达式。只有当^符号出现在正则表达式的起始位置时是特殊的。

运维技术书籍免费下载 https://www.linuxprobe.com

$	定位位于行末尾的正则表达式，只有当$符号出现在正则表达式的末尾时是特殊的
\	转义随后的特殊字符
\(\)	将包含在“\(” 和“\)”之间的模式保存到一个特殊的保持空间。用这种方法在一行中可以最多保存 9 个模式。用转义序列“\1”到“\9”可以重新使用它们。
\n	匹配用前面“\(” 和“\)”保存的第 n 个模式，这里 n 是一个从 1 到 9 的数字，前面保存的模式从行的左边开始编号。
&	当在替换字符串中使用时打印整个被匹配的文本。

sed 的命令汇总

: :label

在脚本中标记一行，用于实现由 b 或 t 的控制转移。label 最多可以包含 7 个字符。（POSIX 标准允许实现工具在需要时定义更长的标签。GUN sed 允许标签的长度任意。）

= [address]=

将所寻址的行编写到标准输出。

a [address]a\

text

在与 address 匹配的每行后面追加 text。如果 text 多于一行，必须用反斜杠将这些行前面的换行符“隐藏”起来。text 将被没有用这种方法隐藏的第一个换行符结束。text 在模式空间中不是可用的并且后续命令不能应用于它。当编辑命令的列表用完时这个命令的结果将被输送到标准输出，而不管在模式空间中的当前行发生了什么。

b [address1[,address2]]b[label]

无条件地将控制转移到脚本其它位置的：label 处。也就是说，label 后面的命令是应用于当前行的下一个命令。如果没有指定 label，控制将一直到达脚本的末端，因此不再有命令作用于当前行。

c [address1[,address2]]c\

text

用 text 替代（改变）由地址选定的行。当指定的是一个行范围时，将所有的这些行作为一个组由 text 的一个副本来替代。每个 text 行后面的换行符必须用反斜杠将其转义，但最后一行除外。实际上，模式空间的内容被删除，因此后续的命令不能应用于它（或应用于 text）。

d [address1[,address2]]d

从模式空间中删除行。因此行没有被传递到标准输出。一个新的输入行被读取，并用脚本的第一个命令来编辑。

D [address1[,address2]]D

删除由命令 N 创建的多行模式空间中的第一部分（直到嵌入的换行符），并且用脚本的第一条命令恢复编辑。如果这个命令使模式空间为空，那么将读取一个新的输入行，和执行了 d 命令。

运维技术书籍免费下载 https://www.linuxprobe.com

g [address1[,address2]]g
将保持空间（参见 h 或 H 命令）中的内容复制到模式空间。如果保持空间为空，则将换行符添加到模式空间。

G [address1[,address2]]G
将换行符后面的保持空间（参见 h 或 H 命令）内容追加到模式空间。如果保持空间为空，则将换行符添加到模式空间。

h [address1[,address2]]h
将模式空间的内容复制到保存空间，即一个特殊的临时缓冲区。保存空间的当前内容被清除。

H [address1[,address2]]H
将换行符和模式空间的内容追加到保持空间中，即使保存空间为空，这个命令也追加换行符。

i [addr ess1]i\
text
将 text 插入到每个和 address 匹配的行的前面（参见 a 详细了解 text 的细节）。

l [address1[,address2]]l
列出模式空间的内容，将不可打印的字符表示为 ASCII 码。长的行被折行。

n [address1[,address2]]n
读取下一个输入行到模式空间。当前行被送到标准输出。新行成为当前行并递增计数器。将控制转到 n 后面的命令，而不是恢复到脚本的顶部。

N [address1[,address2]]N
将下一个输入行追加到模式空间的内容之后;新添加的行与模式空间的当前内容用换行符分隔（这个命令用于实现两行的模式匹配。利用\n 来匹配嵌入的换行符，则可以实现多行模式匹配）。

p [address1[,address2]]p
打印所寻址的行。注意这将导致输出的重复，除非默认的输出用“#n”或“-n”命令行选项限制。常用于改变流控制（d,n，b）的命令之前并可能阻止当前行被输出。

P [address1[,address2]]P
打印由 n 命令创建的多行模式空间的第一部分(直到嵌入的换行符)。如果没有将 N 应用于某一行则和 p 相同。

q [address]q
当遇到 address 时退出。寻址的行首先被写到输出（如果没有限制默认输出），包括用前面的 a 或 r 命令为它追加的文本。

r [address]r file
读取 file 的内容并追加到模式空间内容的后面。必须在 r 和文件名 file 之间保存留一个空格。

s [address1[,address2]]s/patter n/replacement/[flags]
用 replacement 代替每个寻址行的 pattern。如果使用了模式地址，那么模式//表示最后指定的模式地址。可以指定下面的标志。

n 替代每个寻址的行的第 n 个/pattern/。n 是 1 到 512 之间的任意数字，并且默认值为 1.

运维技术书籍免费下载 https://www.linuxprobe.com

g 替代每个寻址的行的所有/pattrtn/，而不只是第一个。

p 如果替换成功则打印这一行。如果成功进行了多个替换，将打印这个行的多个副本。

w file 如果发生一次替换则将这行写入 file。最多可以打开 10 个不同的 file.

t [address1[,address2]]t [label]

测式在寻址的行范围内是否成功执行了替换，如果是，则转移到有 label 标志的行（参见 b 和：）。如果没有给出 label，控制将转移到脚本的底部。

w [address1[,address2]]w file

将模式空间的内容追加到 file。这个动作是在遇到命令时发生而不是在输出模式空间内容时发生。必须在 w 和这个文件名之间保留一个空格。在脚本中可以打开的最大文件数是 10.如果文件不存在，这个命令将创建一个文件。如果文件存在，则每次执行脚本时将改定其内容，多重写入命令直接将输出写入到同一个文件并追加到这个文件的末端。

x [address1[,address2]]x

交换模式空间和保持空间的内容。

y [address1[,address2]]y/abc/xyz/

按位置将字符串 abc 中的字符转换成字符串 xyz 中的相应该字符。

运维技术书籍免费下载 https://www.linuxprobe.com

附录二

awk 的快速参考

这个附录介绍关于 awk 脚本的语言的特点。

命令行语法

调用 awk 的语法有两种基本形式：

```
awk [-v var=value] [-Fr e] [--] ' patter n { action }'  var=value
datafile(s)
awk [-v var=value] [-Fr e] -f scriptfile [--] var=value datafile(s)
```

一个 awk 命令行是由命令，脚本和输入文件名组成的。输入是从命令中指定的文件中读取的。如果没有指定文件名或指定为“-”，那么将从标准输入中读取。选项-F 将字段分隔符（FS）设置为 re.

-v 选项在脚本执行前将变量 var 的值设置为 value.这个 BEGIN 过程运行前执行。（参见下面有关命令行参数的讨论）。

根据 POSIX 参数分析约定，选项“--”标记命令行选项的结束。例如，利用这个选项你可以指定以“-”开头的 datafile，否则这将和命令行选项混淆。

你可以在命令行指定一个由用单引号包围的由 pattern 和 action 组成的脚本。换句话说，你也可以将脚本写入一个单独的文件并在命令行中用-f 选项指定文件名 scriptfile.

通过在脚本后命令行上指定参数，可以将它们传递到 awk 中，这包括设置系统变量，例如 FS,OFS 和 RS。value 可以是一个文字，一个 shell 变量（$var）或一个命令的结果（‘cmd’）;如果其中包含空格或制表符则必须用引号包围起来。可以指定任意多个变量。

直到读取第一个输入行。命令行参数才能使用，因此在 BEGIN 过程中不能访问。（awk 和 nawk 中的较老的实现可以在运行 BEGIN 过程之前处理前导的命令行赋值，这种《The awk Programming Language》中所描述的是相反的，也就是，在 BEGIN 过程之后，awk 将它们作为文件名来处理。在 19891 年初，Bell

运维技术书籍免费下载 https://www.linuxprobe.com

Lab awk 修正了它们，并添加了-v 选项。现在它是 POSIX awk 的一部分。）参数按照它们出现在命令行中的顺序来求值，直到遇到 一个文件名。出现在这个文件名之后的参数在下一个文件名被识别时变为可用。

用 shell 实现调用 awk

在系统提示符下输入脚本只能练习简单的单行脚本。任何可以作为一个命令来调用并重用的脚本都可以放到 shell 脚本中。利用 shell 程序来调用 awk 可以使别人更容易使用这些脚本。

可以将调用 awk 的命令行放在一个文件中，给它一个名字以说明脚本的功能。将文件做成可执行的（利用 chmod 命令）并将它放入到一个目录中，这个目录中包含局部命令。可以在命令行输入这个 shell 脚本的名字来执行 awk 脚本。这是为了更易于使用和重用。

在现代的 unux 系统中，包括 linux，你可以用#!语法来创建自包含 awk 脚本：

```
#! /usr/bin/awk -f
script
```

awk 参数和输入文件名可以在调用 shell 脚本的命令行上指定。注意，使用的路径名取决于系统。

awk 语言概要

这部分概括了 awk 如何处理输入记录和描述组成 awk 程序的各种语法要素。

记录和字段

每个输入行都被分割为字段。默认情况下，字段定界符是一个或多个空格和/或制表符。可以使用-F 命令行选项来改变字段分隔符。同时还要设置 FS 的值。下面命令行将字段分隔符改为一个冒号：

```
awk -F: -f awkscr /etc/passwd
```

也可以将定界符赋值给系统蛮量 FS。这通常在 BEGIN 过程中完成，但也可以作为命令行的参数来传递。

```
awk -f awkscr FS=: /etc/passwd
```

每个输入行都是几个字段组成的一个记录。每个字段可以根据它在这个记录中的位置来引用。“$1”表示第一个字段中的值，“$2”表示第二个字段的值等等。“$0”表示整个记录。下面的操作打印每个输入行的第一个字段：

```
{ print $1 }
```


运维技术书籍免费下载 https://www.linuxprobe.com

默认的记录分隔符是一个换行符。下面的过程设置了 FS 和 RS，使得 awk 将直到遇到空行前的任意个行解释为一个记录，而每个行是一个单独的字段。

```
BEGIN { FS = "\n"; RS = "" }
```

注意，当 RS 被设置为一个空字符时，不管 fs 的值是什么，换行符总是用来分隔字段。这些在《The Awk Programming Languge》和《Effective AWK Programming》中讨论了更多的细节。

脚本的格式

awk 脚本包含一系列模式匹配规则和操作：

```
patter n { action }
```

action 操作由一个或多个语句组成，用于对那些怀模式匹配的输入行执行操作。如果没有指定模式，这个操作对每个行都执行。下面的例子用 print 语句打印输入文件的每个行：

```
{ print }
```

如果指定一个模式，那么默认的操作由 print 语句构成，如上所示。

也可以出现函数的定义：

```
function name (parameter list) { statements }
```

这个语法定义了函数 name，给出了在函数体中可以访问的参数列表。在参数列表中指定的变量被看做是函数内部的局部变量，其他所有的变量是全局的并可以在函数外部对它们进行访问。当调用自定义函数时，不允许在函数名和左圆括号之间有空格存在。在函数的定义中允许有空格。自定义函数在第九章“函数”中进行了介绍。

行为终止

awk 脚本中的行以一个换行符或一个分号终止。如果允许，可利用分号将多个行放在同一行，但这将降低程序的可读性。在语句之间允许存在空行。

程序控制语句（do,if,for 或 while）的范围包括下一行，在那一行列出了相关语句。如果给出了多个与控制语句相关的语句，那么必须用大括号括起来。

```
if (NF > 1) {
name = $1
total += $2
}
```


运维技术书籍免费下载 https://www.linuxprobe.com

对于使用反斜杠(\)转义符实现将一行代码写在多行上。也可以用下列字符中的任何一个来中断行。

, { && ||

在 gawk 中还是用“？”或“：”来延续一个行。字符串不能跨行（在 gawk 中除外，在 gawk 中可以在换行符的前面添加“\”来实现字符串换行）。

注解

注解以“#”开始并以换行符结束。它可以单独出现在一行上或出现在一行的最后。注释是描述性的说明，用于解释脚本有操作。注释不可能用反斜杠来延伸到下一行。

模式

一个模式可以是下面所列的任何一个：

```
/regular expression/
relational expression
BEGIN
END
patter n, patter n
```

1. 正则表达式使用元字符的扩展集并且必须用斜杠包围。对于正则表达式的详细讨论。请参阅第三章“了解正则表达式语法”。
2. 关系表达式使用关系操作符，列在本章“表达式”的后面。
3. BEGIN 模式在第一个输入行被读取之前应用，END 模式在最后一个输入行被读取之后应用。
4. 使用！可以否定匹配，也就是处理与模式不匹配的行。
5. 可以防问若干行，和在 sed 中一样：
   pattern,pattern
   除了 BEGIN 和 END 外，其他模式都可以使用下面的操作符来进行组合：
   && 逻辑与
   ||逻辑或
   nawk 的 sun 版本（sunos4.1X）不支持将正则表达式看做是一个大的布尔表达式的一部分。例如，“/cute/&&/sweet/”或“/fast/ || /quick/”是不起作用的。
   另外，C 的条件操作符?:（pattern ? pattern :pattern)可以用模式中。
6. 可以将模式放在圆括号号确保正确的求值。
7. BEGIN 和 END 模式必须和操作相联系。如果编写了多个 BEGIN 和 END 规则，它们在被应用前被合成一个规则。

运维技术书籍免费下载 https://www.linuxprobe.com

正则表达式

表 b-1 总结了在第三章中描述的正则表达式。元字符按照优先级列出。

表 b-1：正则表达式元字符

特殊字符	用法
c	和不是元字符的任何字符 C 匹配
\	转义它后面的任意元字符，包括它自己
^	将后面的正则表达式定位在字符串的开始处
$	将前面的正则表达式定位在字符串的末端
.	和任意单个字符匹配，包括换行符
[...]	和用方括号包围起来的字符类中的任何一个字符匹配、脱字符（^）作为方括号中的第一个字符表示将匹配所有列在类中的字符以外的字符，连字符(-)用于表示一个字符范围。右方括号(])在作为类中第一个字符表示是这个类的成员。当其他的元字符作为类的成员时将失去它们原来的含义。但\除外，它可以用来转义]，即使没有用在第一位。
r1|r2	允许与正则表达式 r1 或 r2 中的任何一个匹配
(r1)(r2)	用于连接正则表达式
r *	与前面的任意个（包括 0 个）正则表达式匹配
r +	与前面正则表达式的一个或多个出现匹配
r ?	与前面正则表达式的 0 个或 1 个出现匹配
(r)	用于正则表达式的分组

正则表达式也可以使用转义序列来访问特殊的字符，和在本附录后面的“转义序列”一节定义的一样。

注意。^和$对 strings 操作，它们不和嵌入到记录或字符串中的换行符匹配。在 POSIX 中，允许在一对方括号中用特殊符号匹配非英语字符。在表 B-2 中对它们进行了描述。

表 B-2：POSIX 字符列表工具

符号	功能
[.symbol.]	比较符号。比较符号是一个多字符序列，应将它看作为一个单元来处理
[=equiv=]	等价类。一个等价类列出了一组字符，这组字符应该被看做是相等的，例如“e” 和 “è”.
[:class :]	字符类。字符类关键词表示不同的字符类。例如字母字符，控制字符等等
[:alnum:]	字母数字字符
[:alpha:]	字母字符

运维技术书籍免费下载 https://www.linuxprobe.com

[:blank:]	空格和制表符
[:cntrl:]	控制字符
[:digit:]	数字字符
[:graph:]	可打印的和可见的（非空格）字符
[:lower:]	小写字符
[:print:]	可打印的字符
[:punct:]	标点符号字符
[:space:]	空白字符
[:upper:]	大写字符
[:xdigit:]	十六进制数字

注意， 这些工具（上面所写的）一直没有被广泛的应用。

表达式

一个表达式可以由常量、变量、操作符和函数组成。常量要么是字符串（字符序列）要么是数值。变量是一个符号，它引用了一个值。可心将它看做是一条信息，返回一个特定数值或字符串的值。

常量

有两种类型的常量，即字符串常量或数值常量。字符串常量必须用引号括起来，而数值常量不需要。

转义序列

表 B-3 描述了可以用在字符串和正则表达式中的转义序列。

表 B-3：转义序列

序列	描述
\a	报警字符，通常是 ASCII BEL 字符
\b	退格符
\f	走纸符
\n	换行符
\r	回车符
\t	水平制表符
\v	垂直制表符
\ddd	将字符表示为 1 到 3 位八进制值
\xhex	将字符表示为十六进制值
\c	任何需要字面表示的字符 C（例如：\" for "）b

a POSIX 不提供“\x”，但它通常是可用的。

b 和 ANSI C 一样，当你在没有列在这个表中的任意字符前放置一个反斜杠时，POSIX 保留这些字符为未定义。在大多数版本的 awk 中，你就会直接

运维技术书籍免费下载 https://www.linuxprobe.com

得到那个字符。

变量

有 3 种类型的变量：自定义变量，内置变量和字段。按照惯例，内置或系统变量的名字全部由大写的字母组成。

变量的名字不能以数字开头，而是由字母，数字和下划线组成，在变量名中字母的大小写是很重要的。

变量不需要声明或初始化。一个变量可以包含一个字符串或数值。对于未被初始化的变量将空串（“”）作为它的字符串值，将 0 作为它的数值。awk 会根据操作来解决定一个值是作为字符串还是数值来处理。

变量的赋值形式为：

```
var=expr
```

它将表达式的值赋给 var。下面的表达式将值 1 赋给变量 x。

```
x=1
```

使用变量的名字可以访问它的值：

```
{ print x }
```

以上语句打印变量 x 的值，本例可以得到 1.

参见后面的“系统变量”一节了解内置变量的信息。利用$n 可以访问字段变量，这里的 n 是 0 到 NF 之间的任意一个数，用于按位置访问字段。也可以表示为一个变量，例如$NF 表示最后一个字段，或表示为一个常量，例如$1 表示第一个字段。

数组

数组是一个可以用来存储一组值的变量。下面的语句为数组的元素赋一个值：

```
array [index] = value
```

在 awk 中，所有的数组都是关联数组。使得关联数组独特的是它的下标可以是一个字符串也可以是一个数字。

关联数组在数组的下标和元素之间建立一种“联系”。对于数组中的每个元素包含一对值：元素的下标和元素的值。关联数组的元素不像传统数组的元素那样按特定的的顺序存储。

运维技术书籍免费下载 https://www.linuxprobe.com

可以使用 for 循环来读取关联数组中的所有元素。

```
for (item in array)
```

这里数组的下标由变量 item 来指定，数组元素的值可以利用 array [item]来测式这个元素的值。也可以使用 delete 语句从数组中删除一个元素。

系统变量

awk 定义了许多特殊的变量，它们可以在程序中访问或重新设置，如表 B-4 所示（默认值列在括号中）。

表 b-4:awk 的系统变量

变量	描述
ARGC	命令行中的参数个数
ARGV	包含命令行参数的数组
CONVFMT	用于数字的字符串转换格式（%.6g）(POSIX)
ENVIRON	环境变量的关联数组
FILENAME	当前文件名
FNR	和 NR 类似，但和当前文件相关
FS	字段分隔符（一个空行）
NF	当前记录中的字段个数
NR	当前记录的个数
OFMT	数字的输出格式（%.6g）
OFS	输出字段分隔符(一个空行)
ORS	输出记录分隔符（一个换行符）
RLENGTH	和函数 match（）匹配的字符串的长度
RS	记录分隔符（一个换行符）
RSTART	和函数 match()匹配的字符串的第一个位置
SUBSEP	数组下标的分隔字符（\034）

操作符

在表 B-5 中按优先顺序列出了可在 awk 中应用的操作符。

表 B-5:操作符

操作符	描述
= += -= *= /= %= ^= **=	赋值操作符
?:	C 语言的条件表达式
||	逻辑或
&&	逻辑与
~ !~	匹配正则表达式与不匹配

运维技术书籍免费下载 https://www.linuxprobe.com

< <= > >= != ==	关系操作符
(blank)	连接符
+ -	加法，减法
* / %	乘法，除法，取模
+ - !	正，负，逻辑非
^ **	求幂
++ --	递增和递减，作为前缀或后缀
$	字段引用

注意： 这里的“**”和“**=”是共同的扩展，它们不是 POSIX awk 组成部分。

语句和函数

包含在大括号中的动作，由一个或多个语句和/或表达式组成。语句和函数之间的区别是函数将返加一个值，并且它的参数列表在圆括号中给出（在形式上不一定严格按照语法规定：printf 被看做是一个语句，而它的参数列表可以包含在圆括号中;getline 是一个函数，但它没有使用括号）。

awk 有许多预定义的算术函数和字符串函数。函数经常按下面的方式调用：

```
retur n = function(ar g1,ar g2)
```

这里的 return 是一个变量，用于保存函数的返回值（实际上，一个函数的返回值可以用在表达式的任何位置，而不仅仅只出现在赋值语句的右边）。函数的自变量是用逗号分隔的一个列表。左圆括号跟在函数名后面（对于内置函数，在函数名和括号之间允许有一个空格）。

awk 的命令汇总

下面是依字母的顺序列出的语句和函数，其中包含了 POSIX awk，gawk 或 nawk 中所有可用的语句和参数。参阅第十一章“awk 的系列产品”，介绍了扩展变量的不同实现。

atan2( ) atan2(y, x)
返回 y/x 的反正切，单位是弧度。

break 从 while、for 或 do 循环中退出。

close( ) close(filename-expr)
close(command-expr)
在大多数 awk 的实现中，你只能同时打开一定数量的文件和/或管道。因此，awk 提供了一个 close()函数，利用这

运维技术书籍免费下载 https://www.linuxprobe.com

个函数你可以关闭一个文件或一个管道。它将打开文件或管道的同一表达式作为参数。这个表达式的每个字符必须和打开文件或管道所用的表达式相同——即使空格也是有意义的。

continue　开始 while、for 或 do 循环的下一个迭代。

cos()　cos(x)
返回 x 的余弦，x 单位为弧度。

delete　delete array[element]
删除数组的元素

do　do
　　body
while (expr)
循环语句。执行在 body 中的语句并计算 expr 的值，当 expr 的值为真时，重复执行 body.

exit　exit [expr]
从脚本中退出，不再读取新行。如果有 END 规则，则执行。选项 expr 是 awk 的返回值。

exp( )　exp(x)
返回 e 的 x 次幂（e^x）

for　for (init-expr; test-expr; incr-expr) statement
C 语言风格的循环结构。init-expr 是计数器变量的初始值。test-expr 是一个关系表达式并在每次执行 statement 之前计算它的值。当 test-expr 为假时，退出循环。incr-expr 用来在每次循环中递增计数器变量。
for (item in array) statement
用于读取关联数组的特殊循环。对于数组中的每个元素，statement 都被执行;可以利用 array [item]的形式来访问元素。

getline　读取下一个输入行
getline [var] [<file]
command | getline [var]
第一种形式从 file 中读取输入，第二种形式读怪 command 的输出结果。两种形式每次都只读取一行，并且每次执行该语句得到下一个输入行。输入行被赋给$0 并分解为字段，且设置 NF、NR 和 FNR.如果指定了 var,那么结果将赋给 var 而$0 不变。因此当结果被赋给一个变量时，当时行没有变化。实际上 getline 是一个函数，当它成功读取一个记录时返回值为 1，当遇到最后一行时返回 0，当由于某些原因读取失败时返回-1.

gsub( )　gsub(r, s, t)
全局替换字符串 s 中与字符串 t 中的正则表达式 r 匹配的所有字符串。返回替换的次数。如果 t 没有给出，默认值为$0。

if　if (expr) statement1

运维技术书籍免费下载 https://www.linuxprobe.com

[else statement2]
条件语句。计算 expr 的值，如果为真，执行 statement1；如果给出了 else 字句，当 expr 为假时执行 statement2。

index( ) index(str, substr)
返回在字符串中的子串的位置（起始位置为 1）

int( ) int(x)
通过将 x 中的小数点后面的数字截断来得到 x 的整数值。

length( ) length(str)
返回字符串的长度，如果没有参数则返回$0 的长度。

log( ) log(x)
返回 x 的自然对数（以 e 为底）

match( ) match(s, r)
模式匹配函数，由正则表达式 r 给出模式，返回在字符串 s 中与 r 匹配的开始位置，如没有发现匹配则返回 0.将 RSTART 和 ELENGTH 的值分别设置为匹配的开始位置和匹配的长度。

next 读取下一个输入行并从第一个规则开始执行脚本。

print print [output-expr] [dest-expr]
求 output-expr 的值并将它直接输出到标准输出，后面跟着 ORS 的值。每个 output-expr 都被 ORS 的值分隔开。dest-expr 是一个可选表达式，直接将输出送到一个文件或管道。“>file”直接将输出送到一个文件，并覆盖它的以前内容。“>>file”将输出追加到一个文件中，保留它以前的内容。在这种情况下，如果文件不存在都将创建这个文件。“| command ”直接将输出作为一个系统命令的输入。

printf printf (for mat-expr [, expr-list]) [dest-expr]
从 C 语言中借用的一个可选的输出语句。它可以产生格式化输出。它也可以用于输出没有自动换行的数据。format-expr 是一个格式说明字符串和常量字符串（参加下一节关于格式说明符的列表）。expr-list 是一个和格式说明符对应的参数列表。参见 print 语句关于 dest-expr 的描述。

rand( ) rand( )
生成 0 到 1 之间的一个随机数。每次执行脚本时这个函数返回相同的一系列数据，除非使用 srand()函数生成随机数发生器的种子数。

retur n return [expr]
使用用户定义的终端退出函数，返回表达式的值

sin( ) sin(x)
返回 x 的正弦，x 单位为弧度。

split( ) split(str, array, sep)
这个函数利用字段分隔符将字符串分解到数组元素中。如果没有指定字段分隔符，则使用 FS 的值。数组的分割和字段的分割是相同的。

sprintf( ) sprintf (for mat-expr [, expr-list])
该函数返回根据 printf 格式说明指定的格式化的字符串。它

运维技术书籍免费下载 https://www.linuxprobe.com

格式化数据但不输出数据。format-expr 是一个格式说明字符串和常量字符串。expr-list 是一个和格式说明符对应的参数列表。

sqrt( ) sqrt(x)
返回的 x 平方根

srand( ) srand(expr)
使用 expr 为随机数发生器设置一个种子数。默认值为当天的时间。返回值为旧的种子数。

sub( ) sub(r, s, t)
替换字符串 s 中与字符串 t 中的正则表达式 r 匹配的所有字符串。如果成功则返回 1，否则返回 0。如果没有给出 t，默认值为$0.

substr( ) substr(str, beg, len)
返回字符串 str 中开始位置为 beg 的字串，后面的字符串最大长序为 len。如果没有给出长度，将得到剩余的字符串。

system( ) system(command)
该函数执行给出的 command 并返回它的状态。执行命令的状态通常表示成功或失败。0 表示命令执行成功。非零值表示某些类型的错误，不管是正的或负的，所执行的命令的有关文档将提供详细的介绍。在 awk 脚本中这个命令的输出结果是不可用的。使用“command|getline”可以将命令的输出读取到脚本中。

tolower( ) tolower(str)
将字符串 srt 中所有大写字母转换为小写字母并返回新的字符串（注 1）。

toupper( ) toupper(str)
将字符串 str 中所有小写字母转换为大写字母并返回新的字符串。

while while (expr) statement
循环结构。当 expr 为真时，执行 statement.

应用在 printf 和 sprintf 中的格式表达式

格式表达式可以在%之后有 3 个可选的修饰符，在%前面为格式说话符：

```
%-width.pr ecision for mat-specifier
```

输出字段的 width 是一个数值型的值。当你指定字段的宽度时，该字段的内容默认为右对齐。必须指定“-”来实现左对齐。因此，“%-20”输出的是一个字段宽度为 20 个字符的左对齐的字符串。如果这个字符串不足 20 个字符，则用空格来填满该字段。

precision 修饰符用于调整十进制或浮点型的值，控制小数点左边出现的数字的个数。对于字符串格式，它控制从这个字符串中打印的字符的个数。

运维技术书籍免费下载 https://www.linuxprobe.com

你可以根据 printf 或 sprintf 中的参数列表为 width 和 precision 动态赋值。要实现这一功能必须注上星号，而不是指定字面值。

```
printf("%*.*g\n", 5, 3, myvar);
```

注 1：在 nawk 的早期版本中，例如用于 sunos4.1.x 的版本，不支持 tolower() 和 toupper().但是，现在它们已经作为 awk 的 POSIX 规范的组成部分。

在这个例子中，宽度为 5，精度为 3，将要打印的值来自于 myvar。nawk 的较老版本不支持这些。

注意，数值输出的默认精度是“%.6g”。这个默认值可以通过设置系统变量 OFMT 来改变。这将影响用 print 语句输出数值的精度。例如，如果你用 awk 编写报告，其中包含美元值，则可将 OFMT 改为“%.2f”。

在表 B-6 中列出了应用于 printf 和 sprinft 语句中的格式说明符。

表 B-6：在 printf 中使用的格式说明符

字符	描述
C	ASCII 字符
d	十进制整数
i	十进制整数，已添加到 POSIX 中
e	浮点型格式([-]d.pr ecisione[+-]dd).
E	浮点型格式([-]d.pr ecisionE[+-]dd).
f	浮点型格式([-]ddd.pr ecision).
g	e 或 f 的轮换，无论哪一个最短，将末尾的零删除
G	E 或 f 的轮换，无论哪一个最短，将末尾的零删除
o	无符号的八进制值
s	字符串
x	无符号的十六进制数，用 a-f 表达 10-15
X	无符号的十六进制数，用 A-F 表达 10-15
%	字面%

通常，凡是在系统的 sprintf(3)子例程中，可用的格式说明符在 awk 中也是可用的。

printf 和 sprinft()的舍入方式通常取决于系统的 C sprintf(3)子例程。在许多机器上，sprintf 舍入是“无偏差的”，这意味着它不总是对“.5”进行进位，因此 1.5 合入得到 2。而 4.5 的舍入却为 4。因此如果你利用一个格式来进行舍入计算（例如，"%.0f"），你应该知道你的系统是如何处理的。下面的函数执行的是传统的舍入，如果你的 awk 的 printf 进行的是无编差舍入，这或许是有用的。

运维技术书籍免费下载 https://www.linuxprobe.com

```
# round --- 一般的四舍五入
#       Arnold Robbins, arnold@gnu.ai.mit.edu
#       Public Domain
function round(x,       ival, aval, fraction)
{
        ival = int(x)    # 整数部分，用 int()截断
        # 是否有小数部分
        if (ival == x)   # 没有小数部分
                return x
        if (x < 0) {
                aval = -x       # 绝对值
                ival = int(aval)
                fraction = aval - ival
                if (fraction >= .5)
                     return int(x) - 1       # -2.5 舍入为 -3
                else
                     return int(x)           # -2.3 舍入为 -2
} else {
                fraction = x - ival
                if (fraction >= .5)
                     return ival + 1
                else
                      return ival
        }
}
```


运维技术书籍免费下载 https://www.linuxprobe.com

附录三

第十二章的补充

这个附录包含了第十二章“综合应用”中介绍的程序的补充程序和文档。

程序 spellcheck.awk 的完整清单

```
# spellcheck.awk -- 交互式拼写检查器。
#
# 作者: Dale Dougherty。
#
# 用法: nawk -f spellcheck.awk [+dict] file。
# 用 spellcheck 作为 shell 程序的名字。
# SPELLDICT = "dict"
# SPELLFILE = "file"
# BEGIN 操作完成下面的任务:
#       1) 处理命令行参数
#       2) 创建临时文件名
#       3) 执行拼写程序创建单词列表文件
#       4) 显示用户响应的列表
BEGIN {
# 处理命令行参数
# 必须至少有两个参数-nawk 和 filename
      if (ARGC > 1) {
      # 如果多于两个参数，第二个参数为 dict
            if (ARGC > 2) {
            # 测试如果字典被指定为 "+"
            # 将 ARGV[1] 赋给 SPELLDICT
                  if (ARGV[1] ~ /^\+.*/)
                        SPELLDICT = ARGV[1]
                  else
                        SPELLDICT = "+" ARGV[1]
            # 将文件 ARGV[2] 赋给 SPELLFILE
                  SPELLFILE = ARGV[2]
            # 删除参数，这样 awk 将不把他们当作文件打开
                  delete ARGV[1]
                  delete ARGV[2]
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
        }
    # 不多于两个参数
        else {
        # 将文件 ARGV[1]赋给 SPELLFILE
            SPELLFILE = ARGV[1]
        # 测试本地字典是否存在
            if (! system ("test -r dict")) {
            # 如果存在，询问是否使用它
                printf ("Use local dict file? (y/n)")
                getline reply < "-"
            # 如果回答为是，使用 "dict"
                if (reply ~ /[yY](es)?/){
                    SPELLDICT = "+dict"
                }
            }
        }
    } # 处理参数> 1 的过程结束
    #如果参数个数不大于 1，那么打印 shell-命令的方法
    else {
        print "Usage: spellcheck [+dict] file"
        exit 1
    }
# 处理命令行参数的过程结束
# 创建临时文件名，每个以 sp_开头
    wordlist = "sp_wordlist"
    spellsource = "sp_input"
    spellout = "sp_out"
# 将 SPELLFILE 复制到临时输入文件
    system("cp" SPELLFILE " " spellsource)
# 现在执行拼写程序，将输出发送到 wordlist
    print "Running spell checker ..."
    if (SPELLDICT)
        SPELLCMD = "spell " SPELLDICT " "
    else
        SPELLCMD = "spell "
    system(SPELLCMD spellsource " > " wordlist )
# 测试单词列表，看是否有拼错的单词出现
    if ( system("test -s " wordlist ) ) {
    # 如果单词列表为空（或拼写命令失败），退出
        print "No misspelled words found."
        system("rm " spellsource " " wordlist)
        exit
    }
# 将单词列表文件赋给 ARGV [1] 使得 awk 能读取它
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
        ARGV[1] = wordlist
# 显示用户的响应列表
        responseList = "Responses: \n\tChange each occurrence,"
        responseList = responseList "\n\tGlobal change,"
        responseList = responseList "\n\tAdd to Dict,"
        responseList = responseList "\n\tHelp,"
        responseList = responseList "\n\tQuit"
        responseList = responseList "\n\tCR to ignore: "
        printf("%s", responseList)
} # BEGIN 过程结束
# 主过程，处理单词列表的每行。
#       目的是显示拼错的单词并提示用户进行适当的处理。

{
# 设置拼错的单词
        misspelling = $1
        response = 1
        ++word
#打印拼错的单词并提示用户响应
        while (response !~/(^ [cCgGaAhHqQ])|^ $/ ) {
                printf("\n%d - Found %s (C/G/A/H/Q/):", word, misspelling)
                getline response < "-"
        }
# 现在处理用户的响应
# CR - 回车表示忽略当前的单词
# 帮助
        if (response ~ /[Hh](elp)?/) {
        # 显示响应列表并再给出提示
                printf("%s", responseList)
                printf("\n%d - Found %s (C/G/A/Q/):", word, misspelling)
                getline response < "-"
        }
# 退出
        if (response ~ /[Qq](uit)?/) exit
# 添加到字典中
        if ( response ~ /[Aa](dd)?/) {
                dict[++dictEntry] = misspelling
        }
# 对每个出现的都进行修改
        if ( response ~ /[cC](hange)?/) {
        # 读取收集的文件的每一行
                newspelling = ""; changes = ""
                while( (getline < spellsource) > 0){
                # 调用函数显示包含拼错单词的行
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
            # 并提示用户对每个进行更正
                make_change($0)
            # 所有行被加入到临时输出文件
                print > spellout
            }
        # 所有行被读取
        # 关闭临时输入和临时输出文件
            close(spellout)
            close(spellsource)
        # 如果做了修改
            if (changes){
        # 显示被修改的行
             for (j = 1; j <= changes; ++j)
                    print changedLines[j]
              printf ("%d lines changed. ", changes)
        # 在保存前执行确认
                confirm_changes()
            }
        }
# 全局修改
        if ( response ~ /[gG](lobal)?/) {
        # 调用函数来提示更正
        # 并显示每个被修改的行。
        # 请求用户在保存前确认所有的修改。
            make_global_change()
        }
} # 主过程结束
# END 过程使所做的修改成为永久性的。
# 它覆盖原始的文件并将单词添加
# 到字典中。
# 它也删除临时文件。
END {
# 如果我们到达这时只读取了一个记录，
# 没有做修改，那么退出。
        if (NR <= 1) exit
# 用户必须对保存更正的文件给出确认
        while (saveAnswer !~ /([yY](es)?)|([nN]o?)/ ) {
            printf "Save corrections in %s (y/n)? ", SPELLFILE
            getline saveAnswer < "-"
        }
# 如果回答是，那么将临时输入文件转移到 SPELLFILE 中
# 保存旧的 SPELLFILE 文件，以防万一
        if (saveAnswer ~ /^ [yY]/) {
            system("cp " SPELLFILE " " SPELLFILE ".orig")
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
            system("mv " spellsource " " SPELLFILE)
      }
# 如果回答为不，那么删除临时输入文件
      if (saveAnswer ~ /^ [nN]/)
            system("rm " spellsource)
#如果单词已经被添加到字典中，
#那么提示用户确认保存在当前字典中
      if (dictEntry) {
            printf "Make changes to dictionary (y/n)? "
            getline response < "-"
            if (response ~ /^ [yY]/){
            # 如果没有指定字典，那么用 "dict"
                  if (! SPELLDICT) SPELLDICT = "dict"
            # 遍历数组并将单词添加到字典中
                  sub(/^ \+/, "", SPELLDICT)
                  for ( item in dict )
                        print dict[item] >> SPELLDICT
                  close(SPELLDICT)
            # 对字典文件排序
                  system("sort " SPELLDICT "> tmp_dict")
                  system("mv " "tmp_dict " SPELLDICT)
            }
      }
# 删除单词列表
      system("rm sp_wordlist")
} #  END 过程结束
# 函数的定义
# make_change -- 提示用户对当前输入行中的拼写错误
#                进行更正。调用它自己找到的字
                 符串中的其他错误
#
#     stringToChange -- 初始为 $0; 因此和$0 的子串不匹配
#     len -- 从 $0 开始到被匹配字符串的末尾的长度
# 假定已经定义了拼写错误
function make_change (stringToChange, len,   # 参数
      line, OKmakechange, printstring, carets)   # 局部的
{
# 在 stringToChange 中匹配拼错的单词; 否则什么也不做
   if ( match(stringToChange, misspelling) ) {
   # 显示匹配的行
         printstring = $0
         gsub(/\t/, " ", printstring)
      print printstring
      carets = "^ "
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
        for (i = 1; i < RLENGTH; ++i)
                carets = carets "^ "
        if (len)
                FMT = "%" len+RSTART+RLENGTH-2 "s\n"
        else
                FMT = "%" RSTART+RLENGTH-1 "s\n"
        printf(FMT, carets)
# 如果没有定义，提示用户更正
        if (! newspelling) {
                printf "Change to:"
                getline newspelling < "-"
        }
# 回车表示忽略
# 如果用户输入更正，并确认
        while (newspelling && ! OKmakechange) {
                printf ("Change %s to %s? (y/n):", misspelling,
newspelling)
                getline OKmakechange < "-"
                madechg = ""
        # 测试响应
                if (OKmakechange ~ /[yY](es)?/ ) {
                # 做修改（只在第一次遇到时）
                        madechg = sub(misspelling, newspelling,
stringToChange)
                }
                else if ( OKmakechange ~ /[nN]o?/ ) {
                        # 提供重新更正的机会
                        printf "Change to:"
                        getline newspelling < "-"
                        OKmakechange = ""
                }
        } #  while 循环结束
# 如果 len 为真，则处理 $0 的子串
        if (len) {
        # 对它进行汇编
                line = substr($0,1,len-1)
                $0 = line stringToChange
        }
        else {
                $0 = stringToChange
                if (madechg) ++changes
        }
# 将修改过的行放入到数组中以备显示
        if (madechg)
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
            changedLines[changes] = ">" $0
   # 创建子串使我们可以和其他情况相匹配
         len += RSTART + RLENGTH
         part1 = substr($0, 1, len-1)
         part2 = substr($0, len)
   # 余下的部分中，是否存在拼错的单词
         make_change(part2, len)
   } # if 结构结束
} # 函数 make_change()结束
# make_global_change --
#        提示用户对所有拼错的行
#        进行全局修改
#        没有参数
# 假定已经定义了拼写错误
function make_global_change(     newspelling, OKmakechange, changes)
{
# 提示用户对当前拼写的单词进行修改
   printf "Globally change to:"
   getline newspelling < "-"
# 回车表示忽略
# 如果有回答，确认
   while (newspelling && ! OKmakechange) {
               printf ("Globally change %s to %s? (y/n):",
misspelling,
                     newspelling)
               getline OKmakechange < "-"
         # 测试响应并做修改
               if (OKmakechange ~ /[yY](es)?/ ) {
               # 打开文件，读取所有的行
                     while( (getline < spellsource) > 0){
                     # 如果找到匹配，用
                     # gusb 做修改并打印每个被修改的行。
                           if ($0 ~ misspelling) {
                                 madechg = gsub(misspelling,
newspelling)
                                 print ">", $0
                                 changes += 1 # 计算修改的行数
                           }
                     # 将所有行写入临时输出文件
                           print > spellout
                     } # 读取文件的 while 循环结束
               # 关闭临时文件
                     close(spellout)
                     close(spellsource)
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
                # 报告修改的数量
                      printf ("%d lines changed. ", changes)
                # 在保存前请确认
                      confirm_changes()
                } #  if (OKmakechange ~ y)结束
   #如果更正没有被确认，提示输入新的单词
                else if ( OKmakechange ~ /[nN]o?/ ){
                      printf "Globally change to:"
                      getline newspelling < "-"
                      OKmakechange = ""
                }
   } #提示用户进行更正的while循环结束
} # 函数make_global_change()结束
# confirm_changes --
# 在保存修改之前确认
function confirm_changes( savechanges) {
#在保存修改之前提示用户确认
      while (! savechanges ) {
            printf ("Save changes? (y/n)")
            getline savechanges < "-"
      }
# 如果确认，用输出代替输入
      if (savechanges ~ /[yY](es)?/)
            system("mv " spellout " " spellsource)
      }
```

masterindex Shell 脚本的清单

```
#! /bin/sh
# 1.1 -- 7/9/90
MASTER=""
FILES=""
PAGE=""
FORMAT=1
INDEXDIR=/work/sedawk/awk/index
#INDEXDIR=/work/index
INDEXMACDIR=/work/macros/current
# 对所有可用的从属的模块添加检测功能
sectNumber=1
useNumber=1
while [ "$#" != "0" ]; do
   case $1 in
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
   -m*)      MASTER="TRUE";;
   [1-9])    sectNumber=$1;;
   *,*)      sectNames=$1; useNumber=0;;
   -p*)      PAGE="TRUE";;
   -s*)      FORMAT=0;;
   -*)       echo $1 " is not a valid argument";;
   *)        if [ -f $1 ]; then
               FILES="$FILES $1"
             else
               echo "$1: file not found"
             fi;;
   esac
   shift
done
if [ "$FILES" = "" ]; then
   echo "Please supply a valid filename."
   exit
fi
if [ "$MASTER" != "" ]; then
   for x in $FILES
   do
   if [ "$useNumber" != 0 ]; then
      romaNum= `$INDEXDIR/romanum $sectNumber `
      awk ' -F\t'  '
         NF == 1 { print $0 }
         NF > 1 { print $0 ":" volume }
      '  volume=$romaNum $x >>/tmp/index$$
      sectNumber= `expr $sectNumber + 1 `
   else
      awk ' -F\t'  '
         NR == 1 { split(namelist, names, ",");
                   volname = names[volume] }
         NF == 1 { print $0 }
         NF > 1 { print $0 ":" volname }
      '  volume=$sectNumber namelist=$sectNames $x >>/tmp/index$$
      sectNumber= `expr $sectNumber + 1 `
   fi
   done
   FILES="/tmp/index$$"
fi
if [ "$PAGE" != "" ]; then
   $INDEXDIR/page.idx $FILES
   exit
fi
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
$INDEXDIR/input.idx $FILES |
sort -bdf -t: +0 -1 +1 -2 +3 -4 +2n -3n | uniq |
$INDEXDIR/pagenums.idx |
$INDEXDIR/combine.idx |
$INDEXDIR/format.idx FMT=$FORMAT MACDIR=$INDEXMACDIR
if [ -s "/tmp/index$$" ]; then
   rm /tmp/index$$
fi
```

masterindex 的文档

这些文档和下面的注释是由 DALE Dougherty 提供的。

masterindex——生成单册或多卷索引的索引程序。

提要

```
masterindex [-master [volume]] [-page] [-screen] [filename..]
```

使用说明

masterindex 根据由 troff 输出的结构化索引条目生成一个格式化的索引。除非对输出进行重定向，否则结果将显示到屏幕上。

选项

-m or -master 表示你在编辑一个多卷的索引。每卷的索引条目应该在一个单独的文件中，且文件名应该依次给出。如果第一个文件不是第一卷，那么指定一个卷号作为单独的参数。这个卷号被转换为一个罗马数字，并被添加到该文件的索引条目的所有页码之前。

-p or -page 为每个页码生成了一个索引条目列表。它也可用于防止条目的硬拷贝。

-s or -screen 用于表示未格式化的索引将要显示在“屏幕”上。默认时指准备的输出中包含用于格式化的 troff 宏。

文件

work/bin/masterindex
/work/bin/page.idx
/work/bin/pagenums.idx
/work/bin/combine.idx
/work/bin/for mat.idx
/work/bin/r otate.idx
/work/bin/r omanum

运维技术书籍免费下载 https://www.linuxprobe.com

/work/macr os/current/indexmacs

参阅

注意这些程序需要“nawk”（新的 awk）：nawk（1）和 sed（1V）。

缺陷

这个新的索引程序是模块化的，它调用了一系列的小程序。这允许我们连接不同的模块来实现新的功能，并且可以更容易地孤立和修复问题。索引条目不应该包含任何 troff 字体更改。这个程序没有解决它们。大于 8 的罗马数字不能被正确的排序，因此只限于 8 本书索引（这个排序程序对罗马数字 1-10 按以下顺序进行排序：I，II,III，IV，IX，V，VI，VII，VIII，X.）

背景细节

Tim O’Reilly 推荐说 he Joy of Cooking (Jof C)索引是一个理想的索引。我彻底地检查了 JofC 索引，并打算编写一个包含它的所有特点的新的索引程序。我没有全部复制 JofC 的格式，但如果想做这是很容易做的。请你观察 JofC 索引并亲自检查它的特点。

我也努力做一些其他事情来对前面的索引程序进行改进，并为个人编写的索引提供更多的支持。

索引编码条目

本节介绍了在文档文件中索引编码条目。我们用.XX 宏来给出文件中索引条目的位置。最简单的情形是：

.XX "entry"

如果条目由主排序关键字和次排序关键字组成，那么我们可以按下面的方式编码：

.XX "primary, secondary"

逗号将两个关键字分隔开。我们还用一个.XN 宏来生成没有页码的“see”引用。它按下面的形式定义：

.XN "entry (See anotherEntry)"

这些编码形式继续按它们自己的方式起作用，而 masterindex 利用 3 个级别的关键字提供了更大的灵活性，3 个关键字为：主关键字，次关键字和第三关键字。你应该按下面的形式定义条目：

.XX "primary: secondary; tertiary"

注意，逗号没有用作定界符。用冒号将主条目和次条目分隔开，用分号将次条目和第三条目分隔开。这意味着在这个语法中逗号可以作为关键字的一个组成部分。但不必担心，你仍然可以利用逗号来分隔主关键字和次关键字（要知道在一行中如果没有发现冒号定界符，第一个逗号被转换为一个冒号）。我建议

运维技术书籍免费下载 https://www.linuxprobe.com

在新书中利用上面的语法来编码，即使你只给出了主关键字和次关键字。
另一个特点是当利用代字号（～）作为定界符时，可以将主关键字和次关键字进行转换。请看下面的条目：
.XX "cat □command"
该条目和下面的两个条目是等价的：
.XX "cat command"
.XX "command: cat"
你可以将次关键字作为主条目的一个分类（命令，属性，函数等等）。注意不要将这两个颠倒了，因为“command cat”没有太大的意义。要在一个条目中使用代字符，请输入“～～”。

我添加了一个新宏.XB,除了索引条目的页码在输出时以粗体显示，用于表示它在某个范围中是最重要的外，和其他的.XX 是一样的。这有一个例子：
.XB "cat command"

当 troff 处理这个索引条目时，它输出带有星号的页码。这就是当输出显示在屏幕上时的格式。当对 troff 格式化编码时，页码用粗体字转义序列包围。（顺便说一下，在 JofC 索引中，我注意到它们允许在罗马字体和粗体字中有相同的页码）。另外，这个页码不能组合到多个连续页码中。

JofC 索引的另外一个特点是第一个次关键字和主关键字出现在同一行。老的索引程序将次关键字放在下一行。用 JofC 方式的一个好处是当条目只包含一个次关键字时可以输出到同一行，所以更易读。因此，你应该由“行整版，清晰度”而不是由“清晰度”来排版下一行。下一个次关键字将被缩进。注意，如果主关键字作为单独的条目存在（有与它相关的页码），这个主关键字的页码引用将被输出到同一行，而第一个次条目将被输出到下一行。

再次强调，尽管三级条目的语法不同，但以下索引条目是正确的：

.XX "line justification, definition of"
它也将产生相同的结果：
.XX "line justification: definition of"
（冒号在输出中不出现。）用类似的方法可以编写下面的条目：
.XX "justification, lines, defined"
或
.XX "justification: lines, defined"

这里在“lines”和“justification”之间的逗号不是作为定界符，而是作为次关键字的一部分。

前面的例子可以写成具有三级的一个条目：
.XX "justification: lines; defined"

这里的分号将第三关键字分隔开。这个分号和关键字一起输出，并且有多个第

运维技术书籍免费下载 https://www.linuxprobe.com

三关键字将跟随在次关键字的后面。

然而主要的任务是为所有的主关键字，次关键字和第三关键字收集页码，因此可以按下面形式的输出：

```
justification 4-9
  lines 4,6; defined, 5
```

输出格式

我想做一件前面的程序没有处理的事，即生成一个没有 troff 代码的索引。masterindex 有 3 种类型的输出：troff，screen 和 page。

默认的输出是利用 troff（通过 fmt）来产生的。其中包含宏，这些宏被定义在 /work/macros/current/indexmacs.这些宏应该产生与前面相同的索引格式，它们大多数直接通过 troff 请求来完成。以下是开始的几行：

```
$ masterindex ch01
.so /work/macros/current/indexmacs
.Se "" "Index"
.XC
.XF A "A"
.XF 1 "applications, structure of 2; program
.XF 1 "attribute, WIN_CONSUME_KBD_EVENTS 13"
.XF 2 "WIN_CONSUME_PICK_EVENTS 13"
.XF 2 "WIN_NOTIFY_EVENT_PROC 13"
.XF 2 "XV_ERROR_PROC 14"
.XF 2 "XV_INIT_ARGC_PTR_ARGV 5,6"
```

顶部的两行是显而易见的。.XC 宏产生多卷输出（对小的书它将输出两卷。用参数来指定卷的宽度是不够灵活的，但这里必须这样处理）。.XF 宏的第一个参数有 3 个可能的值。一个“A”表示第二个参数是一个字母表中的字符，应该作为分隔符来输出。一个“1”表示第二个参数包含一个主目录。一个“2”表示条目以一个次条目开始。

当使用-s 参数调用时，这个程序将准备在屏幕上显示索引（或作为 ASCII 文件打印）。
这里有几行：

```
$ masterindex -s ch01
                    A
applications, structure of 2; program 1
attribute, WIN_CONSUME_KBD_EVENTS 13
  WIN_CONSUME_PICK_EVENTS 13

WIN_NOTIFY_EVENT_PROC 13
```


运维技术书籍免费下载 https://www.linuxprobe.com

```
XV_ERROR_PROC 14
XV_INIT_ARGC_PTR_ARGV 5,6
XV_INIT_ARGS 6
XV_USAGE_PROC 6
```

显然，这对快速验证索引是有用的。第三种格式也可用来验证索引。用-p 来调用，将提供索引条目的逐页列表。

```
$ masterindex -p ch01
Page 1
        structure of XView applications
        applications, structure of; program
        XView applications
        XView applications, structure of
        XView interface
        compiling XView programs
        XView, compiling programs
Page 2
        XView libraries
```

编辑一个主索引

通过指定-m 选项可以调用多卷主索引。对于特定卷的每组索引条目必须放在一个单独的文件中。

```
$ masterindex -m -s book1 book2 book3
xv_init( ) procedure II: 4; III: 5
XV_INIT_ARGC_PTR_ARGV attribute II: 5,6
XV_INIT_ARGS attribute I: 6
```

必须以连续的顺序来指定文件。如果第一个文件不是第一卷，你可以用相应的卷号作为参数。

```
$ masterindex -m 4 -s book4 book5
```


运维技术书籍免费下载 https://www.linuxprobe.com