**GNU Awk**

GAWK： awk 完全参考手册

GNU awk 4.1 版用户指南

2015年4月

献给我的父母，谢谢他们给我的爱，以及为我树立起的榜样。

献给我的妻子，Miriam，他使我有一个完整的人生。谢谢你让我一起建设你的生活。

献给我的孩子们，Chana，Rivka，Nachum，以及 Malka，因为你们而使得我们的生活丰富多彩。

# 第三版前言

我跟 Arnold Robbins 是好朋友。我们由于 AWK 因缘际会而结识。这是好几年前的事了。当时我正开始一份新工作，并注意到在公司角落里放着一台没用的 Unix 计算机。没有人知道怎么使用它，我也不会。但是，几天后，它跑起来了，并且我就是 root，也是唯一的用户。从那天起，我就开始从一个统计分析师向一个 Unix 程序员转变了。

我经常往返于图书馆与书店，搜索跟 Unix 相关的图书。有一次找到一本灰色的 AWK 相关的书，即 a.k.a. Alfred V. Aho， Brian W. Kernighan， 与 Peter J. Weinberger 写的 *The AWK Programming Language (Addison-Wesley， 1988)*。***awk*** 简洁的编程规范——就是在输入中查找某种模式的字串并执行相应动作——经常能够将复杂的或者冗长的操作化为几行代码。我非常兴奋，并着手开始利用 AWK 进行编程。

可是，我计算机上的 ***awk*** 是在灰色书里所描述的一个受限的语言版本。而我发现我的计算机中的就是“旧 ***awk***”，而书中讲述的是“新 ***awk***”。我知道这是典型情况；旧版本不肯让位，也不愿意放弃自己的名字。如果一个系统有一个“新 ***awk***”，就会毫无例外地叫做 nawk，并且有几个系统有这样的软件。取得 nawk 的方式是通过 ***prep.ai.mit.edu*** 的 ftp 下载源代码。gawk 是一个由 David Trueman and Arnold 所写的新版本的 ***awk***，并可以在 GNUL General Public License 的许可下取得。

（顺便说一句，现在找“新 ***awk***”很容易了，GNU/Linux 就带有，你也可以在大多数系统中下载二进制版本或者源代码；我老婆就在他的 VMS BOX 中使用 gawk。）

我的 Unix 系统因闲置而被使用起来，当然也还不会连接到网络中。因此，很明显由于gawk 的存在，以及 Unix 社会社区的一般特点，我需要一个“新 ***awk***”，因此，我就写了一个，我把它叫做 ***mawk***。在我完成前，我已经知道了 gawk，但是已经太迟了。因此，我还是把 ***mawk*** 发送到了 comp.source 的新闻组里。

几天后，我收到了来自 Arnold 的邮件，里面他介绍了一下他自己。他建议我分享 ***mawk*** 的设计与算法，并且提交一个 POSIX 标准草案，这样我就能够更新 ***mawk*** 来支持 *AWK Programming Language* 发布后的语言扩展。

袒白讲，如果我们的角色互换，我可能就不会如此开放，我们也就可能从不会见面。我很高兴我们见了一面。他是一个 AWK 专家中的专家，也是确实是个好人。Arnold 为 Free Software Foundation 贡献了很多的才能与时间。

这本书是 gawk 的参考书，但是核心是一个关于 AWK 编程的书，适用于非常广的读者。编程规范参考的是 1987 年贝尔实验室的定义，并于1992年成为 POSIX 工具标准的语言规范。

另一方面，对于 AWK 程序新手，能够学到实用的编程方法，而这也正是 AWK 的基本方法：数据驱动控制流，利用正则表达式进行匹配，以及关联数组。那些正在尝鲜的人可以通过特殊的 /inet 文件来使用 gawk 的网络协议接口。

这本书中的例子会清楚地表明，使用 AWK 编写的程序要比C写成的程序更小更快。结果，可能利用 AWK 来编写算法并快速运行，以此来及时发现问题。经常的情况下，解释型的性能就足够了，因此 AWK 的原型就直接转换成了产品。

新的 pgawk（profiling gawk）会产生代码执行的采样数据。我最近实验了一个算法，即对 n 行输入，显示出约Cn2这样的性能，而理论预测下为约 Cnlogn。查看了 awkprof.out 程序几分钟后，就在程序的一行代码中发现了问题所在。这样，pgawk 又成为我程序员工具箱中的一个好工具。

Arnold 淫浸在 AWK 中有十几年，并且开发了 gawk，也即本书的主题。如果你使用 AWK，并且想知道 how，那么就读这本书吧。

Michael Brennan

Author of mawk

March 2001

# 第四版前言

有些东西一直没有变化。13年前，我写下“如果你使用 AWK 或者想学习它如何工作，那么就读这本书。”这句话当时是真理，而现在依然是真理。

学习如何使用编程语言并仅仅是掌握语法这么简单。需要在理解的基础上利用语言提供的特性来解决现实中的编程问题。这本书的焦点就是用许多的例子来展示如何使用 AWK。

而有些东西是有变化的。我们的计算机要比之前快得多，并且有更多的内存。结果是高级的语言导致了速度与存储的低效使用。因为性能而先在 AWK 中写原型然后用 C 重写，这种情况很少发生，因为更多的情况下，原型已经足够快了。

当然，有些操作更适合于用 C 或者 C++ 来实现，对于 gawk 4.1 以及后面的版本，你不需要选择到底是用 AWK 还是用 C/C++ 来实现。你可以用 AWK 来实现大部分的程序。而用 C/C++ 来写的程序，你完全可以用 C 或者 C++ 来写成动态插件，然后使用 gawk 进行装载合并。第十六章描述了所有的技术细节，其中有许多的例子，希望让你学习到里里外外的一切。

我很享受用 AWK 来进行编程，并且在用心（重）读这本书。我想你也会这样。

Michael Brennan

Author of mawk

October 2014

# 序言

当工作于文本文件的时候，有几类工作不断重复出现。你可能想抽取指定的行并丢弃其他部分。或者你想更改特定模式的文本，而其他部分的不变。这样的工作使用awk来完成通常很简单。awk工具解释特定的编程语言，使得它处理数据重组的工作非常在行。

GNU 版本的 awk 称为 gawk，使用恰当的参数或者环境变量，它完全与 POSIX 标准的，由 Brian Kemighan 维护的 Unix 版本的 awk 语言兼容。这就是说，正确编写的 awk 程序，也可以在 gawk 下运行。因此，在大多情况下，我们不区别两者的实现。

通过 awk 我们可以达到如下目的：

* 管理小的个人数据库
* 产生报表
* 完善数据
* 产生索引并执行其他文件准备工作
* 实验算法，这样可以回头用其他的语言实现。

除此之外，gawk 还提供如下设施，使我们能够：

* 抽取数据的位或者块用于处理
* 排序数据
* 测评并调试 awk 程序
* 用 C/C++ 编写函数以扩展语言功能

本书会向你们传授awk语言，并告诉你们如何来有效地使用它。你有可能已经熟悉基本的系统命令，如 cat、ls[[1]](#footnote-2)等，也熟悉一些基本的 shell 工具，如输入输出重定向以及管道。

awk 语言的实现可以在许多不同的计算机环境中获取。本书除了描述 awk 语言的总体，也会描述awk 的特殊实现，即 gawk（表示 GNU awk）。gawk 在很多 Unix 系统下运行，包括从 Intel 架构的 PC 机，到大型机系统。gawk 也被移植到了 Mac OS 平台，Windows 平台（所有版本）以及 OS/2 的 PC 机，还有 OpenVMS [[2]](#footnote-3)系统上。

# 译序

这是我第一次翻译这么长的书。

当我看到 GAWK 简单的语言与强大的功能时，我于是想到网上查找一些中文的资料，以快速学习一下这个工具，这样可以方便我在处理 Linux 字串时能够使用到它。

但是我很失望，找了很久，能够找到的资料基本上是支离破碎的，并没有完整全面的中文手册。我于是求其次地找了一个英文文档，其原名为《GAWK: Effective AWK Programming》，由 Arnold D. Robbins，即 gawk 当前的主要维护者之一编写。初看之下，里面的内容浅显易懂，正好符合我自己的需求。因为这个原因，我萌发了将这个手册翻译为中文的想法。但是我拿不准我是否有这样的坚持，能够将这个手册全部翻译出来。

我长久都没有动手，主要是因为没有这个魄力下定决心来做这件事情，再一个对于自己的语言能力也不太自信。一年后，这个想法一直在大脑里挥之不去。于是心里盘算，英文文档大概有500多页，如果我每天翻译两页，则一年后，怎么着也能够把这个文档翻译出来。并且，如果能够翻译到四页，则所需要的时间立即减少一半。翻译一本英文书，除去能力外，还得看自己是不是能够放弃翻译时的功利想法，觉得这么做是否值得。并在这种情况下，是否能够坚持一个较长而无味的过程，最后将文档翻译出来。最后，我还是下定决心，试一试。

于是我第一步是将 PDF 格式的英文文档转换成 Word 文档（我用其他的工具转换过，但是结果不是很好。因为原来的 PDF 文档中有很多特别的格式，在 Word 中很难控制。所以一不做，二不休，干脆把文字复制出来，再调整其格式与原来的 PDF 相符。我不知道这么做是不是很蠢，并有违程序员懒的美名！）。光这个事情就耗了我近一个星期的时间，2016 年春节前把这个底本做好了，并且开始了翻译。

实在的，前面的翻译速度是有一点慢，但是总体来说，还是按照原先的设计速度进行的。就算在过年的几天里，几乎每天都有进展。而且，翻译的难度不是很大，最考验人的是每天要拿出一定的时间来做这个事情。因为一旦停下来，就会为停下来找理由。后面就会有越来越多的理由来为延迟开脱。所以，自从翻译开始，我几乎没有中断每天做一点翻译。在翻译最后的几个附录中，我还差点要放弃翻译下去，因为觉得那些东西翻译没有什么用。但是为了保证翻译的完整性，同时也为把这个事情做“圆满”，我还是坚持把这个事情做下来了。最后只剩下 GPL 与 FDL 没有翻译。这两个协议比较其他的部分有难度一些。关键在于要翻译出带点协议性质，语言精准而且严谨，确实对我而言，能力还是不够的。而且这两个协议，我想在互联网上应该已经有了，所以也就不再翻译了。

翻译到后面的时候，自己确实感觉比原来快了，可能是因为熟练了的原因。剩下的整理也花掉了我的一些时间。通过翻译这个事情，确实使自己有如下收获：

1. 至少对 Word 文档的操作有一点点提升。

2. 考验并提升了一下自己的能力。

3. 再遇到长的英文文档，可能就没那么怵了。

4. 翻译能力是否有大的提高，这个不好说，但是应该有一点吧。

这里的文字，几乎都是一个字一个字码起来，而且也是第一次翻译，里面肯定还是有很多错误的地方，也有一些词不达意的地方，对于汉语的掌握有欠火候，也会使得翻译出来的行文没有那么符合中文习惯，或者不够优美。这些真还得要各位海涵，如果大家觉得这个翻译对大家有一点作用，我就觉得这些付出也值得了。如果读者更慷慨一点，能够告诉我哪里有问题，或者有一些改进的建议那就更好了。如果有这方面的改进，请发邮件到 yeyouqun@163.com。

我希望我的一点小努力，也为中国的软件事业做了一点点贡献。

在最后，如果要有什么感谢的，首先得要感谢自由软件运动的发起者们。如果没有他们，就不会有自由软件社区，也就不会有 GNU/Linux，也就没有 GAWK 这些优秀的工具。其次还要感谢原文的作者。别人也是自由软件的拥趸，如果没有他们的先期付出，也就没有这个文档的存在。目前还只有我一个人在翻译这个文档，如果可以，我想小激动地感谢一下自己。我希望以后感谢列表能够变得更长。希望更多的人参与到完善它的过程里来。

对了，我想感谢一下我儿子。好多次我打开电脑开启一天的翻译工作的时候，他经常会在键盘上随意敲击，使得我还得花掉一些时间来撤销他所做的“工作”。而且还要嚷着要听 《ABCD 歌》什么的，还得陪他听他所说的一万分钟那么久。

不自量力的叶佑群

2016年 4 月

**目 录**

[第三版前言 III](#_Toc448583435)

[第四版前言 V](#_Toc448583436)

[序言 VI](#_Toc448583437)

[译序 VII](#_Toc448583438)

[*awk*与 gawk 的历史 11](#_Toc448583439)

[玫瑰别名 12](#_Toc448583440)

[使用本书 12](#_Toc448583441)

[排版约定 14](#_Toc448583442)

[暗角 15](#_Toc448583443)

[GNU 项目及本书 15](#_Toc448583444)

[如何贡献 16](#_Toc448583445)

[致谢 17](#_Toc448583446)

[第一部分：*awk*语言 19](#_Toc448583447)

[第一章 开始*awk* 20](#_Toc448583448)

[1.1 如何执行*awk*程序 20](#_Toc448583449)

[1.1.1 一次性 awk 程序 20](#_Toc448583450)

[1.1.2 不带输入文件运行 awk 21](#_Toc448583451)

[1.1.3 运行长程序 22](#_Toc448583452)

[1.1.4 可执行的 awk 程序 22](#_Toc448583453)

[1.1.5 *awk*程序中的注释 23](#_Toc448583454)

[1.1.6 Shell引号问题 24](#_Toc448583455)

[1.1.6.1 MS-Windows批处理文件中的引号问题 26](#_Toc448583456)

[1.2 例子中使用的数据文件 27](#_Toc448583457)

[1.3 一些简单例子 28](#_Toc448583458)

[1.4 两规则的例子 29](#_Toc448583459)

[1.5 复杂一点的例子 30](#_Toc448583460)

[1.6 *awk*语句与行 31](#_Toc448583461)

[1.7 *awk*的其他特性 33](#_Toc448583462)

[1.8 何时使用*awk* 33](#_Toc448583463)

[1.9 总结 34](#_Toc448583464)

[第二章 运行*awk*与 gawk 35](#_Toc448583465)

[2.1 调用*awk* 35](#_Toc448583466)

[2.2 命令行参数 35](#_Toc448583467)

[2.3 其他命令行参数 41](#_Toc448583468)

[2.4 命名标准输入 42](#_Toc448583469)

[2.5 gawk 使用的环境变量 43](#_Toc448583470)

[2.5.1 环境变量 AWKPATH 43](#_Toc448583471)

[2.5.2 环境变量 AWPLIBPATH 44](#_Toc448583472)

[2.5.3 其他的环境变量 44](#_Toc448583473)

[2.6 gawk 的退出状态 46](#_Toc448583474)

[2.7 在你的程序中包含其他的文件 46](#_Toc448583475)

[2.8 在你的程序装入动态扩展 48](#_Toc448583476)

[2.9 已经取消的选项以及特性 49](#_Toc448583477)

[2.10 未记载的选项与特性 49](#_Toc448583478)

[2.11 总结 49](#_Toc448583479)

[第三章 正则表达式 50](#_Toc448583480)

[3.1 如何使用正则表达式 50](#_Toc448583481)

[3.2 转义序列 51](#_Toc448583482)

[3.3 正则表达式操作符 53](#_Toc448583483)

[3.4 使用方括号表达式 56](#_Toc448583484)

[3.5 匹配多少？ 57](#_Toc448583485)

[3.6 使用动态正则表达式 58](#_Toc448583486)

[3.7 特定于 gawk 的正则表达式操作符 59](#_Toc448583487)

[3.8 匹配时的大小写 60](#_Toc448583488)

[3.9 总结 61](#_Toc448583489)

[第四章 读取输入文件 63](#_Toc448583490)

[4.1 输入如何拆分为记录 63](#_Toc448583491)

[4.1.1 标准 awk 的记录拆分 63](#_Toc448583492)

[4.1.2 使用 gawk 分割记录 65](#_Toc448583493)

[4.2 检查域 67](#_Toc448583494)

[4.3 非常数域编号 68](#_Toc448583495)

[4.4 更改域内容 69](#_Toc448583496)

[4.5 指定记录如何进行分隔 71](#_Toc448583497)

[4.5.1 标准的域分隔符空白符 72](#_Toc448583498)

[4.5.2 使用正则表达式分隔域 73](#_Toc448583499)

[4.5.3 各字符皆为分隔符 74](#_Toc448583500)

[4.5.4 从命令行中设置 FS 74](#_Toc448583501)

[4.5.5 一行即一域 76](#_Toc448583502)

[4.5.6 域分割总结 76](#_Toc448583503)

[4.6 读取固定宽度数据 77](#_Toc448583504)

[4.7 以内容定义域 79](#_Toc448583505)

[4.8 多行记录 81](#_Toc448583506)

[4.9 用 getline 输入数据 83](#_Toc448583507)

[4.9.1 不带参数调用 getline 83](#_Toc448583508)

[4.9.2 将 getline 赋值给变量 84](#_Toc448583509)

[4.9.3 使用 getline 读取文件 85](#_Toc448583510)

[4.9.4 从文件中 getline 到变量 86](#_Toc448583511)

[4.9.5 从管道中使用 getline 86](#_Toc448583512)

[4.9.6 从管道中getline到变量 88](#_Toc448583513)

[4.9.7 从并程中使用 getline 88](#_Toc448583514)

[4.9.8 使用并程 getline 到变量 89](#_Toc448583515)

[4.9.9 回顾 *getline* 89](#_Toc448583516)

[4.9.10 getline 变体总结 90](#_Toc448583517)

[4.10 带超时的输入读取 90](#_Toc448583518)

[4.11 命令行中的目录 91](#_Toc448583519)

[4.12 总结 92](#_Toc448583520)

[4.13练习 93](#_Toc448583521)

[第五章 打印输出 94](#_Toc448583522)

[5.1 print 语句 94](#_Toc448583523)

[5.2 print 语句例子 95](#_Toc448583524)

[5.3 输出分隔符 96](#_Toc448583525)

[5.4 控制 print 的数字输出 97](#_Toc448583526)

[5.5 使用 printf 语句美化打印 97](#_Toc448583527)

[5.5.1 printf 介绍 98](#_Toc448583528)

[5.5.2 格式控制字母 98](#_Toc448583529)

[5.5.3 printf 的修饰符 99](#_Toc448583530)

[5.5.4 *printf* 范例 102](#_Toc448583531)

[5.6 重定向*print* 与 *printf* 的输出 103](#_Toc448583532)

[5.7 标准的预打开的特殊文件数据流 105](#_Toc448583533)

[5.8 gawk 中的特殊文件名 106](#_Toc448583534)

[5.8.1 在gawk访问其他已开文件 106](#_Toc448583535)

[5.8.2 用于网络的特殊文件 107](#_Toc448583536)

[5.8.3 特殊文件名说明 107](#_Toc448583537)

[5.9 关闭输入输出重定向 108](#_Toc448583538)

[5.10 总结 110](#_Toc448583539)

[5.11练习 111](#_Toc448583540)

[第六章 表达式 112](#_Toc448583541)

[6.1 常量、变量与转换 112](#_Toc448583542)

[6.1.1 常量表达式 112](#_Toc448583543)

[6.1.1.1 数值与字串常量 112](#_Toc448583544)

[6.1.1.2 八进制与十六进制数值 113](#_Toc448583545)

[6.1.1.3 正则表达式常量 114](#_Toc448583546)

[6.1.2 使用正则表达式常量 114](#_Toc448583547)

[6.1.3 变量 116](#_Toc448583548)

[6.1.3.1 在程序中使用变量 116](#_Toc448583549)

[6.1.3.2 在命令行中进行变量赋值 116](#_Toc448583550)

[6.1.4 在字串与数值之间进行转换 117](#_Toc448583551)

[6.1.4.1 awk 如何转换字串与数值 117](#_Toc448583552)

[6.1.4.2 语言设置会影响转换 118](#_Toc448583553)

[6.2 操作符：对值进行处理 119](#_Toc448583554)

[6.2.1 算术操作符 119](#_Toc448583555)

[6.2.2 字串连接 121](#_Toc448583556)

[6.2.3 赋值表达式 122](#_Toc448583557)

[6.2.4 递增与递减操作符 125](#_Toc448583558)

[6.3 真值与条件 126](#_Toc448583559)

[6.3.1 *awk* 的真与假 126](#_Toc448583560)

[6.3.2 变量类型与比较表式式 127](#_Toc448583561)

[6.3.2.1 字串类型与数值类型 127](#_Toc448583562)

[6.3.2.2 比较操作符 128](#_Toc448583563)

[6.3.2.3 POSIX 中的字串比较规则 130](#_Toc448583564)

[6.3.3 布林表达式 131](#_Toc448583565)

[6.3.4 条件表达式 132](#_Toc448583566)

[6.4 函数调用 133](#_Toc448583567)

[6.5 操作符优先级（操作符的嵌套） 135](#_Toc448583568)

[6.6 哪里造成差异 136](#_Toc448583569)

[6.7 总结 136](#_Toc448583570)

[第七章 模式、动作与变量 138](#_Toc448583571)

[7.1 模式元素 138](#_Toc448583572)

[7.1.1 以正则表达式为模式 139](#_Toc448583573)

[7.1.2 以表达式为模式 139](#_Toc448583574)

[7.1.3 用模式指定记录范围 140](#_Toc448583575)

[7.1.4 BEGIN 与 END 特殊模式 141](#_Toc448583576)

[7.1.4.1 启动与清理动作 141](#_Toc448583577)

[7.1.4. BEGIN 与 END 规则中的输入输出2 142](#_Toc448583578)

[7.1.5 BEGINFILE 与 ENDFILE 特殊模式 143](#_Toc448583579)

[7.1.7 空模式 144](#_Toc448583580)

[7.2 在程序中使用 Shell 变量 144](#_Toc448583581)

[7.3 动作 145](#_Toc448583582)

[7.4 动作中的控制语句 146](#_Toc448583583)

[7.4.1 if-else 语句 146](#_Toc448583584)

[7.4.2 while 语句 147](#_Toc448583585)

[7.4.3 do-while 语句 148](#_Toc448583586)

[7.4.4 for 语句 148](#_Toc448583587)

[7.4.5 switch 语句 150](#_Toc448583588)

[7.4.6 break 语句 151](#_Toc448583589)

[7.4.7 continue 语句 152](#_Toc448583590)

[7.4.8 next 语句 153](#_Toc448583591)

[7.4.9 nextfile 语句 154](#_Toc448583592)

[7.4.10 exit 语句 154](#_Toc448583593)

[7.5 预定义变量 155](#_Toc448583594)

[7.5.1 控制 *awk* 的内置变量 156](#_Toc448583595)

[7.5.2 传递信息的内置变量 158](#_Toc448583596)

[7.5.3 使用 ARGC 与 ARGV 163](#_Toc448583597)

[7.6 总结 165](#_Toc448583598)

[第八章 *awk* 数组 166](#_Toc448583599)

[8.1 数组基础 166](#_Toc448583600)

[8.1.1 数组介绍 166](#_Toc448583601)

[8.1.2 指向数组元素 168](#_Toc448583602)

[8.1.3 给数组元素赋值 169](#_Toc448583603)

[8.1.4 基本数组实例 169](#_Toc448583604)

[8.1.5 遍历数组元素 170](#_Toc448583605)

[8.1.6 在 gawk 中使用预定义的数组遍历顺序 171](#_Toc448583606)

[8.2 用数值作为数组下标 174](#_Toc448583607)

[8.3 用未初始化的数值作为数组下标 174](#_Toc448583608)

[8.4 delete 语句 175](#_Toc448583609)

[8.5 多维数组 176](#_Toc448583610)

[8.5.1 遍历多维数组 178](#_Toc448583611)

[8.6 数组之数组 178](#_Toc448583612)

[8.7 总结 180](#_Toc448583613)

[第九章 函数 182](#_Toc448583614)

[9.1 内置函数 182](#_Toc448583615)

[9.1.1 调用内置函数 182](#_Toc448583616)

[9.1.2 数值函数 183](#_Toc448583617)

[9.1.3 字串操作函数 184](#_Toc448583618)

[9.1.3.1 sub()、gsub() 与 gensub() 函数中的 ‘\’与 ‘&’ 192](#_Toc448583619)

[9.1.4 输入输出函数 194](#_Toc448583620)

[9.1.5 时间函数 197](#_Toc448583621)

[9.1.6 位操作函数 201](#_Toc448583622)

[9.1.7 获取类型信息 203](#_Toc448583623)

[9.1.8 字串翻译函数 203](#_Toc448583624)

[9.2 用户自定义函数 204](#_Toc448583625)

[9.2.1 函数定义语法 204](#_Toc448583626)

[9.2.2 函数定义范例 206](#_Toc448583627)

[9.2.3 调用用户自定义函数 207](#_Toc448583628)

[9.2.3.1 如何调用函数 208](#_Toc448583629)

[9.2.3.2 控制变量作用域 208](#_Toc448583630)

[9.2.3.3 函数变量中的传参与传引用 210](#_Toc448583631)

[9.2.4 return 语句 212](#_Toc448583632)

[9.2.5 函数与其在变量类型上的影响 213](#_Toc448583633)

[9.3 间接函数调用 213](#_Toc448583634)

[9.4 总结 218](#_Toc448583635)

[第十章 awk 函数库 221](#_Toc448583636)

[10.1 命名函数库的全局变量 222](#_Toc448583637)

[10.2 通用编程 223](#_Toc448583638)

[10.2.1 将数值转换成字串 223](#_Toc448583639)

[10.2.2 断言 224](#_Toc448583640)

[10.2.3 数值舍入舍出 226](#_Toc448583641)

[10.2.4 Cliff 随机数生成器 227](#_Toc448583642)

[10.2.5 在字符与数值之间变换 227](#_Toc448583643)

[10.2.6 将数组合并成字串 228](#_Toc448583644)

[10.2.7 管理时间 229](#_Toc448583645)

[10.2.8 一次读取整个文件 231](#_Toc448583646)

[10.2.9 将字串用引号引起并传递给 Shell 232](#_Toc448583647)

[10.3 数据文件管理 232](#_Toc448583648)

[10.3.1 注意数据文件的边界 233](#_Toc448583649)

[10.3.2 重新读取当前文件 234](#_Toc448583650)

[10.3.3 检查可读的数据文件 235](#_Toc448583651)

[10.3.4 检查长度为零的文件 235](#_Toc448583652)

[10.3.5 将赋值当成是文件名 236](#_Toc448583653)

[10.4 处理命令行选项 237](#_Toc448583654)

[10.5 读取用户数据库 242](#_Toc448583655)

[10.6 读取组数据库 246](#_Toc448583656)

[10.7 遍历数组之数组 249](#_Toc448583657)

[10.8 总结 251](#_Toc448583658)

[10.9练习 252](#_Toc448583659)

[第十一章 实用的 awk 程序 253](#_Toc448583660)

[11.1 运行例子程序 253](#_Toc448583661)

[11.2 为快乐与评测重新发明轮子 253](#_Toc448583662)

[11.2.1 剪切域与列 254](#_Toc448583663)

[11.2.2 用正则表达式在文件中搜索 258](#_Toc448583664)

[11.2.3 打印用户信息 262](#_Toc448583665)

[11.2.4 将大文件分片 263](#_Toc448583666)

[11.2.5 将输出复制到多个文件中 265](#_Toc448583667)

[11.2.6 打印去重文本行 267](#_Toc448583668)

[11.2.7 统计 270](#_Toc448583669)

[11.3 awk 程序的大集合 272](#_Toc448583670)

[11.3.1 查找文档中的重复字 272](#_Toc448583671)

[11.3.2 报警时钟程序 273](#_Toc448583672)

[11.3.3 转换字符 275](#_Toc448583673)

[11.3.4 打印邮件标题 277](#_Toc448583674)

[11.3.5 生成字使用统计 279](#_Toc448583675)

[11.3.6 从未排序的文本中去重 281](#_Toc448583676)

[11.3.7 从 Texinfo 源文件中提取程序 281](#_Toc448583677)

[11.3.8 一个简单的流编辑器 285](#_Toc448583678)

[11.3.9 使用库函数的简单方式 286](#_Toc448583679)

[11.3.10 从字典中查找单词 292](#_Toc448583680)

[11.3.11 来些新鲜的 293](#_Toc448583681)

[11.4 总结 293](#_Toc448583682)

[11.5练习 294](#_Toc448583683)

[第十二章 gawk 的高级特性 297](#_Toc448583684)

[12.1 允许非十进制数数据输入 297](#_Toc448583685)

[12.2 控制数组排序与遍历 298](#_Toc448583686)

[12.2.1 控制数据遍历 298](#_Toc448583687)

[12.2.2 用 gawk 对数组进行值与键的排序 302](#_Toc448583688)

[12.3 与其他进程进行双向通信 304](#_Toc448583689)

[12.4 利用 gawk 进行网络通信 306](#_Toc448583690)

[12.5 测评你的 awk 程序 307](#_Toc448583691)

[12.6总结 310](#_Toc448583692)

[第十三章 gawk 的国际化 311](#_Toc448583693)

[13.1 国际化与本地化 311](#_Toc448583694)

[13.2 GNU 的 gettext 311](#_Toc448583695)

[13.3 awk 程序的国际化 313](#_Toc448583696)

[13.4 翻译 awk 程序 315](#_Toc448583697)

[13.4.1 提取已标志字串 315](#_Toc448583698)

[13.4.2 重排 printf 参数 315](#_Toc448583699)

[13.4.3 awk 的移植性问题 316](#_Toc448583700)

[13.5 简单的国际化例子 317](#_Toc448583701)

[13.6 gawk 可以说你的语言 319](#_Toc448583702)

[13.7总结 319](#_Toc448583703)

[第十四章 调试 awk 程序 320](#_Toc448583704)

[14.1 gawk 调试器介绍 320](#_Toc448583705)

[14.1.1 通用的调试技术 320](#_Toc448583706)

[14.1.2 调试概念 320](#_Toc448583707)

[14.1.3 awk 调试 321](#_Toc448583708)

[14.2 gawk 调试例子 321](#_Toc448583709)

[14.2.1 如何开启调试器 322](#_Toc448583710)

[14.2.2 查找 Bug 322](#_Toc448583711)

[14.3 主要的调试器命令 325](#_Toc448583712)

[14.3.1 控制断点 326](#_Toc448583713)

[14.3.2 控制执行 327](#_Toc448583714)

[14.3.3 查看与更改数据 329](#_Toc448583715)

[14.3.4 程序栈 330](#_Toc448583716)

[14.3.5 获取程序与调试器状态信息 331](#_Toc448583717)

[14.3.6 其他命令 332](#_Toc448583718)

[14.4 支持 334](#_Toc448583719)

[14.5 限制 334](#_Toc448583720)

[14.6 总结 335](#_Toc448583721)

[第十五章 算术运算与gawk 中的任意精度算术运算 336](#_Toc448583722)

[15.1 计算机算术计算说明 336](#_Toc448583723)

[15.2 要了解的其他知识 337](#_Toc448583724)

[15.3 gawk 的任意精度计算 338](#_Toc448583725)

[15.4 浮点算法：买者自慎！ 339](#_Toc448583726)

[15.4.1 浮点计算是不精确的 339](#_Toc448583727)

[15.4.1.1 很多的数不能完全表示 340](#_Toc448583728)

[15.4.1.2 值比较时要小心 340](#_Toc448583729)

[15.4.1.3 错误累积 341](#_Toc448583730)

[15.4.2 获取你需要的精度 341](#_Toc448583731)

[15.4.3 增加几位精度然后再舍入转出 342](#_Toc448583732)

[15.4.4 设置精度 343](#_Toc448583733)

[15.4.5 设置舍入转出模式 344](#_Toc448583734)

[15.5 gawk 中的任意精度整数计算 345](#_Toc448583735)

[15.6 标准与既存实践 346](#_Toc448583736)

[15.7总结 348](#_Toc448583737)

[第十六章 编写 gawk 扩展 349](#_Toc448583738)

[16.1 介绍 349](#_Toc448583739)

[16.2 扩展许可 349](#_Toc448583740)

[16.3 在高层级中是它是如何工作的 350](#_Toc448583741)

[16.4 API 描述 351](#_Toc448583742)

[16.4.1 介绍 351](#_Toc448583743)

[16.4.2 通用目的的数据类型 353](#_Toc448583744)

[16.4.3 内存分配函数与相应的简化宏 356](#_Toc448583745)

[16.4.4 构造函数 357](#_Toc448583746)

[16.4.5 注册函数 357](#_Toc448583747)

[16.4.5.1 注册扩展函数 357](#_Toc448583748)

[16.4.5.2 注册退出函数 358](#_Toc448583749)

[16.4.5.3 注册扩展版本字串 359](#_Toc448583750)

[16.4.5.4 定制输入分析器 359](#_Toc448583751)

[16.4.5.5 定制输出包装器 362](#_Toc448583752)

[16.4.5.6 定制双路处理器 364](#_Toc448583753)

[16.4.6 打印信息 365](#_Toc448583754)

[16.4.7 更新 ERRNO 366](#_Toc448583755)

[16.4.8 请求值 366](#_Toc448583756)

[16.4.9 访问与更新参数 367](#_Toc448583757)

[16.4.10 符号表访问 367](#_Toc448583758)

[16.4.10.1 通过名字访问与更新变量 367](#_Toc448583759)

[16.4.10.2 通过 Cookie 访问与更新变量 368](#_Toc448583760)

[16.4.10.3 创建并使用缓存值 370](#_Toc448583761)

[16.4.11 数组操作 371](#_Toc448583762)

[16.4.11.1 数组数据类型 371](#_Toc448583763)

[16.4.11.2 数组函数 372](#_Toc448583764)

[16.4.11.3 遍历数组的所有元素 374](#_Toc448583765)

[16.4.11.4 如何生成与发布数组 377](#_Toc448583766)

[16.4.12 API 变量 379](#_Toc448583767)

[16.4.12.1 API 版本常量与变量 379](#_Toc448583768)

[16.4.12.2 信息变量 380](#_Toc448583769)

[16.4.13 样板代码 380](#_Toc448583770)

[16.5 gawk 如何找到扩展 382](#_Toc448583771)

[16.6 例子：一些文件函数 382](#_Toc448583772)

[16.6.1 使用 *chdir()*  与 *stat()* 382](#_Toc448583773)

[16.6.2 *chdir()* 与 *stat()* 的C 代码 384](#_Toc448583774)

[16.6.3 整合扩展 390](#_Toc448583775)

[16.7 gawk 发行版本中的例子扩展 391](#_Toc448583776)

[16.7.1 与文件相关的函数 391](#_Toc448583777)

[16.7.2 针对 *fnmatch()*的接口 394](#_Toc448583778)

[16.7.3 针对*fork()，wait()* 与 *waitpid()* 的接口 395](#_Toc448583779)

[16.7.4 启用文件就地编辑 395](#_Toc448583780)

[16.7.5 字符与数值：ord()与chr() 396](#_Toc448583781)

[16.7.6 读取目录 397](#_Toc448583782)

[16.7.7 倒转输出 397](#_Toc448583783)

[16.7.8 双路 I/O 例子 398](#_Toc448583784)

[16.7.9 输出与恢复数组 398](#_Toc448583785)

[16.7.10 读取整个文件 399](#_Toc448583786)

[16.7.11 扩展的时间函数 399](#_Toc448583787)

[16.7.12 API 测试 400](#_Toc448583788)

[16.8 gawkextlib 项目 400](#_Toc448583789)

[16.9总结 401](#_Toc448583790)

[16.10练习 402](#_Toc448583791)

[附录 A awk 语言的演化 404](#_Toc448583792)

[A.1在 V7 与 SVR3.1 之间的变化 404](#_Toc448583793)

[A.2 在SVR3.1 与 SVR4 之间的变化 405](#_Toc448583794)

[A.3 在 SVR4 与 POSIX awk 之间的变化 405](#_Toc448583795)

[A.4 Brian Kernighan awk 中的扩展 406](#_Toc448583796)

[A.5 不在 POSIX awk 中的 gawk 扩展 406](#_Toc448583797)

[A.6 gawk 特性历史 408](#_Toc448583798)

[A.7 通用扩展汇总 414](#_Toc448583799)

[A.8正则表达式范围与本地语言设置：又臭又长陈年旧事 414](#_Toc448583800)

[A.9 gawk 的主要贡献者 416](#_Toc448583801)

[A.10总结 418](#_Toc448583802)

[附录 B 安装 gawk 419](#_Toc448583803)

[B. 1gawk 发布版 419](#_Toc448583804)

[B.1.1 取得 gawk 发布版 419](#_Toc448583805)

[B.1.2 提取发布版 419](#_Toc448583806)

[B.1.3 gawk 发布版的内容 420](#_Toc448583807)

[B.2 在 Unix-like 的系统上编译与安装 gawk 423](#_Toc448583808)

[B.2.1 为 Unix-like 的系统编译 gawk 423](#_Toc448583809)

[B.2.2其他配置选项 424](#_Toc448583810)

[B.2.3配置过程 424](#_Toc448583811)

[B.3安装到其他操作系统上 425](#_Toc448583812)

[B.3.1安装到 PC 操作系统 425](#_Toc448583813)

[B.3.1.1为 PC 系统安装一个编译版 425](#_Toc448583814)

[B.3.1.2为 PC 操作系统编译 gawk 426](#_Toc448583815)

[B.3.1.3在PC操作系统上测试gawk 427](#_Toc448583816)

[B.3.1.4在PC 操作系统上使用 gawk 428](#_Toc448583817)

[B.3.1.5在Cygwin 中使用 gawk 429](#_Toc448583818)

[B.3.1.6在MSYS 环境中使用 gawk 430](#_Toc448583819)

[B.3.2在 Vax/VMS 以及 OpenVMS 系统上编译安装 gawk 430](#_Toc448583820)

[B.3.2.1在VMS 上编译 gawk 430](#_Toc448583821)

[B.3.2.2在 VMS 编译 gawk 动态扩展 431](#_Toc448583822)

[B.3.2.3在VMS 上安装gawk 432](#_Toc448583823)

[B.3.2.4在 VMS 上运行 gawk 433](#_Toc448583824)

[B.3.2.5 VMS 的 GNV 项目 434](#_Toc448583825)

[B.3.2.6有一些 VMS 系统有旧版本的 gawk 434](#_Toc448583826)

[B.4报告问题与BUG 434](#_Toc448583827)

[B.5其他可自由获取的 awk 实现 435](#_Toc448583828)

[B.6总结 437](#_Toc448583829)

[附录C 实现提示 439](#_Toc448583830)

[C.1后向兼容与调试 439](#_Toc448583831)

[C.2为 gawk 添加功能 439](#_Toc448583832)

[C.2.1访问 gawk 的 Git 仓库 439](#_Toc448583833)

[C.2.2添加新特性 440](#_Toc448583834)

[C.2.3将 gawk 移植到新系统中 441](#_Toc448583835)

[C.2.4为什么自动生成的文件要放在 Git 中 442](#_Toc448583836)

[C.3后续可能的扩展 444](#_Toc448583837)

[C.4实现的一些限制 445](#_Toc448583838)

[C.5扩展 API 设计 445](#_Toc448583839)

[C.5.1旧机制的问题 446](#_Toc448583840)

[C.5.2新机制的目标 446](#_Toc448583841)

[C.5.3其他的设计决策 447](#_Toc448583842)

[C.5.4未来的增长空间 448](#_Toc448583843)

[C.6对旧扩展的兼容性 448](#_Toc448583844)

[C.7总结 449](#_Toc448583845)

[附录D 基本编程概念 450](#_Toc448583846)

[D.1程序做的是什么 450](#_Toc448583847)

[D.2计算机中数据的值 451](#_Toc448583848)

[名词解释 453](#_Toc448583849)

[GNU 通用公共许可 465](#_Toc448583850)

[GNU 自由文档许可 466](#_Toc448583851)

# *awk*与 gawk 的历史

|  |
| --- |
| 编程语言菜单  1份egrep，1 份snobol[[3]](#footnote-4)  2 份ed，3 份C  将所有部分通过lex 与yacc 混合在一起，并且辅以少量文档，并发布。8年后，添加多一份egrep，以及两份的C。文档完备然后发布。 |

Awk的名字来源于最初的设计者：Alfred V. Aho， Peter J. Weinberger 和 Brian W. Kernighan。它的第一个版本写于 1977 年的AT&T 贝尔实验室。1985 年，新版本语言通过引入用户自定义的函数、多输入流、正则表达式，而使得其功能更加强大。新版本的gawk在 Unix System V Release 3.1（1987年）的时候已经广泛可用。而在 System V Release 4(1989年）中的版本添加了一些新的特性，并清理掉了一些语言的“暗角”行为。在 POSIX 命令语言以及工具标准中，对 ***awk***规范进行进一步的澄清。 gawk 的设计者与最初的 ***awk***设计者对于 POSIX 规范都有相应的反馈。

Paul Rubin在 1986 编写了 gawk，Jay Fenlason 根据Richard Stallman的建议完善了它。John Woods也贡献了部分代码。1988年跟1989年，David Trueman在我的帮助下完全重构了 gawk，以解决与更新的 ***awk***兼容的问题。大概是1994年，我成为了主要的维护者。现在的版本主要着眼于 Bug 修复、性能提升，标准化，有时也添加一些新的特性。

1997 年5月， J¨urgen Kahrs觉得有必要用 awk 访问网络，在我的一些帮助下， gawk 添加进了这样的特性。也就是那个时候，我也用 gawk 写了一堆的 TCP/IP 代码（作为 gawk 发布版本一部分，有一个独立的文档可用）。他的代码最终在 gawk 3.1 版本的时候成为主流 gawk 发布版本的一部分。

John Haque重写了 gawk 的内部实现，以提供***awk***级的调试器，这个版本在 2011 年的 4.0 版本可用。

请查看，***在第 416 页***， 以了解对 gawk 作出过重要贡献的人。

# 玫瑰别名

***awk***语言已经进化了多年，在 *，****在第 404 页***，有完全的列表。在本书中所涉及的我们可以称为“new ***awk***”。类似的，***awk***的原始版本称为“old ***awk***”。

在现在多数的系统中，你运行***awk***工具你都有一些版本的新***awk***。[[4]](#footnote-5)如果你的系统中的标准***awk***是旧的版本，你用下面的代码测试，就可以得到这样的结果：

**$ *awk* 1 /dev/null**

**errorawk: syntax error near line 1**

**errorawk: bailing out near line 1**

在这个例子中，你应该能查找到一新版本的***awk***，或者直接安装 gawk 即可。

在整本书中，如果所涉及的所有语言特性，都在一个完整实现了 POSIX 标准的***awk***中找到，我们为了简化，就使用***awk***来指代。当提到的特性特定于 GNU 版本，我们则使用 gawk指代。

# 使用本书

***awk***这个词，即指代特定的程序，也表示你要用于告诉这个程序所做事情的语言。当需要特别小心时，我们称之为“***awk***语言”，以及“***awk***工具”。本书既说明如何用***awk***语言编写程序，也说明如何来运行***awk***工具。“***awk***程序”术语则指用***awk***语言编写的程序。

本书主要涉及 POSIX 标准所定义的***awk***特性。也包括在 gawk 中实现了的那些特性。这么处理，在于试图描述gawk 与其他***awk***实现版本[[5]](#footnote-6)之间的差异。最后，会提及那些在 gawk 中实现了，但不在***awk*** POSIX 标准中定义的特性。

本书既可以作为教程，又可以作为参考。如果你是个新手，你可以跳过那些复杂的部分。你也可以忽略那些交叉参考；他们主要用于专家级用户，同时作为信息参考以及用于本书的 HTML 版本。

在全书中还分布有一些边侧栏。它们主要对相关的点进行更完整全面的解释，但是第一次读本书时可能不太有兴趣涉猎。所有在边侧栏的内容会在索引中出现。

在多数据情况下，我们使用的例子都是完整的***awk***程序。一些进阶部分的小节，仅显示***awk***程序的一部分，以此来说明正在进行描述的概念。

本书主要目标人群主要集中在那些还没接触过***awk***的人，书中相当多的信息对于专家也非常有用。对于***awk*** POSIX 标准的部分更是如此。包括在*，****在第 221 页***， 以及*，****在第 253 页***，中的例子应该会相当有趣。

本书分为如下这些部分：

* 第一部分对***awk***语言以及 gawk 程序进行详细描述。通过从最基础的开始，不断详解***awk***的所有特性。它们包含如下章节：
* *，* ***在第 20 页***，包括一些使用***awk***的基本内容。
* *，* ***在第 35 页***，描述如何运行 gawk、命令行参数的意义，以及它如何来查找***awk***程序源代码。
* *，* ***在第 50 页***，总体介绍正则表达式，尤其是那些 POSIX ***awk***以及 gawk 所支持的内容。
* *，* ***在第 63 页***，描述 ***awk*** 如何来读取数据。本章中介绍记录(Records）与域（Fields）的概念，也包括 ***getline*** 命令。I/O 重定向是首先要描述的部分，而网络 I/O 则简单涉及。
* *，* ***在第 94 页***，介绍在 ***awk*** 程序中如何使用 print 与 ***printf*** 产生输出。
* *，* ***在第 112 页***，介绍表达式，这些表达式是构建程序代码块，并完成目标的基础。
* *，* ***在第 138 页***，描述如何为匹配记录编写模式，即当某个记录匹配时所执行的动作。同时还包括 ***awk*** 与 gawk 所使用的预定义变量。
* *，* ***在第 166 页***，包括 ***awk*** 中的唯一的数组结构，关联数组。删除数组元素以及整个数组，同时也涉及gawk中的数组的排序。在本章还包括 gawk 如何来提供数组之数组的实现方式。
* *，* ***在第 182 页***，描述 ***awk*** 与 gawk 所提供的内建函数，以及用户自定义函数。讨论 gawk 如何来间接调用函数。
* 第二部分展示如何使用***awk***与 gawk 来解决实际问题。这里有非常多的代码供读者阅读与学习。这个部分包括如下各章：
* *，****在第 221 页***，展示大量的可以通过 ***awk*** 程序来调用的函数。
* *，* ***在第 253 页***，有许多的 ***awk*** 程序，通过阅读这两章，可以让你看到 ***awk*** 程序如何来解决真正的问题。
* 第三部分关注特定于 gawk 的特性，它们包括如下各章：
* *，****在第 297 页***，描述各种高级特性。其中包括如来控制数组遍历的顺序、与另一个进程之间的双向通信、执行 TCP/IP 通信、测评你的 ***awk*** 程序。
* *，* ***在第 311 页***，描述如何将程序消息在运行时翻译成不同语言的特性。
* *，****在第 320 页***，描述 gawk 调试器。
* *，****在第 336 页***，描述高级的数学计算工具。
* *，****在第 349 页***，描述如何通过编写C与C++ 扩展为gawk添加新变量与函数。
* 第四部分包括一些附录，名词解释，以及本书代码所使用的两个许可。它们包含如下附录：
* *，****在第 404 页***，描述 ***awk*** 语言从第一个版本到现在的演化过程。同时也描述 gawk 在不同的时间内所引入的特性。
* *，****在第 419 页***，描述如何获取 gawk，如何在 POSIX 兼容的系统中编译，以及如何在非 POSIX 系统中编译与使用它。介绍如何来报告 gawk Bug，以及介绍可以自由获取 ***awk*** 的地方。
* *，****在第 439 页***，描述如何禁用 gawk 扩展，以及如何向 gawk 贡献新代码，还包括 gawk 未来可能的开发方向。
* *，****在第 450 页***，为那些初识计算编程的人提供一些粗略的背景资料。
* *，****在第 453 页***，定义大多数的本书中所用到的术语的意义，如果你看到你不熟悉的，可以参考这里。
* *，****在第 465 页***，与 *，****在第 466 页***，提供本书中源代码以及本书所使用到的两个许可证协议。

# 排版约定

本书是通过 TextInfo[[6]](#footnote-7) 完成的。这是一种文档格式化语言。单一的 TexInfo 源文件，即可用于产生在线版本的文档，也可以产生打印输出的文档。因为这个原因，其排版约定与你所读其他的书可能有一些差异。

你在命令行上输入的例子，由通用的Shell 主次提示符，***‘$’***与***‘>’***引导。命令行的输出由字符“-|”引导。这代表了命令行的典型输出。错误信息以及其他的命令行标准错误输出则由“error”引导，例如：

**$ echo hi on stdout**

**-|hi on stdout**

**$ echo hello on stderr 1>&2**

**errorhello on stderr**

在文本中，所有与程序相关的东西，如命令名、变量和函数名、字串、数据、正则表达式常量都以这样的字体出现。代码段也以相同的字体呈现，并由“如下”来括起来。由用户或者程序员替换的部分也用这个字体。选项是如–f 这样的参数。文件名由/path/to/ourfile 这样的串提示。如果有什么东西需要特别强调，如某个点上需要强调，也这么处理。第一个出现的术语，同一个句子中的先前出现的术语，包括其定义也用相同的字体。

你从键盘上键入的字符也用这样的字体呈现。对于一些特别的“控制字符”，这些字符是需要你同时键入 CONTROL 键以及另一个键来输入。比如Ctrl-d 需要首先按下并保持 CONTROL 键，然后敲击d 键，然后再释放两个键。

为了简单处理，在整本书中，我们把Brian Kernighan’s版本的awk 称为“BWK awk”（查看*，****在第 435 页***，查看这个版本与其他版本的信息）。

# 暗角

*暗角总是存在的——无论你多么仔细，总是有一些更小的，黑暗的地方。*

*—Brian Kernighan*

flash.jpg在没有做 POSIX标准（以及没写 *GAWK: Effective AWK Programming*）之前，***awk***特性的文档少得可怜，甚至是没有文档。对于这样的特性（经常称为“暗角”）会通过手电的图形在页边上给予提示，正如这个页所示的。它们也出现在索引中的“暗角”区。

但是，正如开放性引述所提示的，在定义上对暗角的覆盖是不完全的。

对于***awk***标准语言的扩展，很多***awk***实现都支持，他们用“(c.e)”进行标示，并在索引中的“通用扩展”以及“扩展，通用”中进行列出。

# GNU 项目及本书

自由软件基金会是一个非营利性组织，其专注于开发与发布可自由分发的软件。它由Richard M. Stallman——即最初的 Emacs 编辑器的作者——创建。GNU 的 Emacs 软件也是目前用得最广泛的 Emacs 版本。

GNU[[7]](#footnote-8)项目是自由软件基金会的项目，其目前依然还在持续地为创建一个完全可自由发布，与POSIX兼容的计算环境而努力。自由软件基金会通过“通用公共请可”（GPL）来保证其下的软件的源代码对于终端用户总是可用的。GPL 的复本已经包含在本书中（*，****在第 465 页***）。GPL 也应用于用 C 写成的 gawk。要查找更多的关于自由软件基金会以及GNU项目的在线信息，你可以查看 GNU 项目的主页。本书也可以从 GNU 站点上阅读。

Shell，编辑器（Emacs），高度可移植的 C、C++ 与 Object-C 编译器，符号调试器以及大量的小工具（如 gawk），都已经完成并可自由获取。GNU 操作系统内核（HURD）也已经发布，只是还处于开发的早期阶段。

在 GNU 操作系统完全开发好前，你应该考虑使用 GNU/Linux，一个可自由发布的，Unix-like 的操作系统，用于 Intel、Power 这样的构架，Sun Sparc，IBM S/390 等其他系统。[[8]](#footnote-9)很多的 GNU/Linux 发行版本都可以通过互联网下载。

你正在阅读的这本书也是免费的——最起码的。这里面的信息对任何一个人都是免费的。也包括与 gawk 相关的，对于机器可读的源代码。（花一点时间阅读一下在*，****在第 466 页***，中的自由文档许可证协议。）

此书已经经过了多个版本。Paul Rubin写了第一份 GAWK Manual，大概有40页。Diane Close 与 Richard Stallman丰富了它，产生了一个大概90页的版本，它只针对最初的，“旧”版本的***awk***。

# 如何贡献

作为 GNU ***awk***的维护者，我曾经想着我应该能够管理这一大堆的公开可得的***awk***程序，我甚至招募过贡献者。从互联网可以得到我想要的东西，能够帮助 gawk 的发布版保持在一个可管理的大小。

最原始的材料，如本书，还可以从ftp://ftp.freefriends.org/arnold/Awkstuff获取，为了把这个事情做得更为开放广泛，我还申请了***awk***.info域名。

但是，我发现我没有时间完全地来管理这些分布的代码：文档已经不再增长，而域名已经几年前就不能使用了。

2008年的晚些时候，有志愿者设立了***awk***相关的网站——http://***awk***.info——工作做得相当棒。

如果你写过一个有味的awk程序，或者写了 gawk 扩展并且希望与其他人分享，请看http://***awk***.info/?contribute来了解如何将它发布到网站上。

# 致谢

最初的 The GAWK Manual 的感谢如下：

很多的人需要感谢，因为他们帮助产生了这个手册。Jay Fenlason 提供了许多的想法与例子。Richard Mlynarik 与 Robert Chassell 给了文档非常有帮助的点评。John W写了*A Supplemental Document for AWK* 文档。UC San Diego 化学部的 Pierce 指出几个问题，即与 ***awk*** 实现有关，也与手册有关，不然这些问题会逃过我们的眼睛。

我要感谢Richard M. Stallman，感谢他对美好世界的愿景，并有勇气成立 FSF 并开启 GNU 项目。

本书早期版本的感谢如下：

下面的人（按字母顺序）为本书的不同版本提供了很好的点评：Rick Adams， Dr. Nelson H.F. Beebe， Karl Berry， Dr. Michael Brennan， Rich Burridge， Claire Cloutier， Diane Close， Scott Deifik， Christopher (“Topher”) Eliot， Jeffrey Friedl， Dr. Darrel Hankerson， Michal Jaegermann， Dr. Richard J. LeBlanc， Michael Lijewski， Pat Rankin， Miriam Robbins， Mary Sheehan， 和 Chuck Toporek。

Robert J. Chassell在使用 Texinfo 上提供了非常有价值的建议。它也应该得到特别感谢，因为他说服我不要用How to Gawk Politely这样的书名。Karl Berry在 Texinfo 中的 TEX 部分给予了相当大的帮助。我也要感谢Marshall 与 Seattle 的Elaine Hartholz ，还有 Detroit 的 Bert and Rita Schreiber 博士，他们牺牲了大量的假期时间，安安静解地呆在他们的家里，使得我能够写作神速，也包括在 gawk 上的工作。

SSC的Phil Hughes借给我他的笔记本 GNU/Linux 系统，不是一次，而是两次，他以这样的一种方式使我在离开家的时候做了大量的工作。

David Trueman应该得到特别的致谢。他自告奋勇地承担起了优化 gawk 的工作，因此 gawk 才会执行得如此之好，没有 Bug。尽管他现在不再介入 gawk 的工作，但是与他在这个项目上一起工作相当愉快。

GNITS邮件列表中的那些无畏的成员们，以及最值得尊敬的Ulrich Drepper为设计国际化特性提供了无价的帮助与反馈。

Dr. Nelson Beebe， Andreas Buening， Dr. Manuel Collado， Antonio Colombo， Stephen Davies， Scott Deifik， Akim Demaille， Darrel Hankerson， Michal Jaegermann， J¨urgen Kahrs，Stepan Kasal， John Malmberg， Dave Pitts， Chet Ramey， Pat Rankin， Andrew Schorr， Corinna Vinschen， 跟 Eli Zaretskii（按字母顺序）组成了目前的 gawk “霹雳移植性小组”。如果没有他们的艰苦工作与帮助，gawk 不会今天有如此稳定，可移植的程序。与这样的一群好人小组一起工作曾经是也将一直是件快乐的事情。

许多人贡献了相当多的代码与文档，他们可以在*，****在第 416 页，***中有完整的列表。

谢谢Michael Brennan写的序言。

谢谢Patrice Dumas新写的makeinfo程序，也谢谢Karl Berry持续工作使用 Texinfo 这样的标志语言如此牛B。

Robert P.J. Day， Michael Brennan， 还有 Brian Kernighan非常好心地作为了本书 2015版本的审核者，他们的工作使用让本书质量更优。

我还要感谢Brian Kernighan，因为在测试与调试 gawk 期间提供的无价帮助，也为他在澄清无数的语言疑点上的的持续帮助与建议。

Brian是一流程序员与技术作者，我要（又一次）感谢他，因为他持续的友谊与为我树立起来的近30年的模范作用。有这样的人作为评论人真是大幸。而他一直以来都如此谦卑。

我还得感谢我的内人，Miriam，因为她的耐心，在这个项目的多个版本之间作为第一读者，同时还与我分享电脑。还要感谢我的老爸老妈，因为他们的爱与生而育我的艰辛。最后，我也要把我的感谢献给 G-d[[9]](#footnote-10)，是他给我了许多机会，礼物，而这些礼物使我能够更好地利用这些机会。

Arnold Robbins

Nof Ayalon

Israel

February 2015

# 第一部分：*awk*语言

# 第一章 开始*awk*

Awk的基本功能就是查找具某种模式的文本行（或者其他单位的文本）。当匹配某个模式后，***awk***会对该行执行指定的动作。Awk然后继续输入行，直到文件输入处理完毕。

Awk中的程序与其他语言的程序不同，因为***awk***程序是数据驱动的（即你描述你要处理的数据，然后告诉程序找到相应数据后要执行的工作）。多数其他语言都是过程式的；你需要在非常细的细节上描述程序每一行要做什么。当与过程式语言打交道时，一般很难清楚地描述你的程序所要处理的数据。正是由于这个原因，***awk***程序非常易读易写。

当执行awk程序，你需要指定一个程序告知awk接下来要做什么。这个程序包括一系统的规则（有可能也会包含函数，一些你现在可以忽略的高级特性；查看*，****在第 204 页***)，每个规则指定了要查找的模式以及一个在查找到模式时要执行的动作。

在语法上，规则包括一个模式，后面带着动作。动作是包含在花括号中的，与模式分离。每一行一般都是新的不同的规则。因此，***awk***程序看起来像这样：

**pattern { action }**

**pattern { action }**

**...**

## 1.1 如何执行*awk*程序

有几种方式执行***awk***程序。如果程序很短，最好的方式直接包含在程序的执行命令行中，如下：

**awk 'program' input-file1 input-file2 ...**

当程序比较长，更好的方式是将其放在一个文件中，然后通过命令行执行：

**awk -f program-file input-file1 input-file2 ...**

在这节中，我们使用两种方式，但是每种方式都有一些变化。

### 1.1.1 一次性 awk 程序

一旦你熟悉了awk，你就可以在使用时立即输入一些简单的程序。你可以把程序作为awk命令行的第一个参数，像这样：

**awk 'program' input-file1 input-file2 ...**

这个程序包括一系列的模式与动作，正如早些时候所描述的那样。

这种命令格式告诉Shell，或者命令行解释器，开启***awk***，并用其中的程序来处理输入文件。程序是由单引号引起来的，因此Shell 不会将***awk***中的特殊字符解析成Shell 的字符。单引号也使用Shell 将所有的程序都作为单独的***awk***程序，可以让程序多于一行。

这样的方式用来执行短的或者中等大小的***awk***程序是很方便的，因为它避免了要用一个单独的文件来写***awk***程序。自包含的Shell 脚本更加可靠，这样就不会跟其他的文件弄混淆。

在本章的后面部分，即*，****在第 28 页***，我们会学一些那些自包含的程序很短的例子。

### 1.1.2 不带输入文件运行 awk

你也可以运行***awk***程序，不带输入文件，如果你敲入下面的命令行：

**awk 'program'**

***awk***会使程序导向到标准输入，也就是说，你在键盘上输入的一切都会接受为输入数据。这会一直持续，直到你输入Ctrl-d 来表示***‘end-of-file’***。（在非POSIX 操作系统中，***‘end-of-file’***有可能不同，如在OS/2 中则是Ctrl-z）

如下面的例子，程序打印一小段建议（来自由Douglas Adams 的 *The Hitchhiker’s Guide to the Galaxy*），使你消除计算机程序复杂性的忧虑：

**$ awk ’BEGIN { print "Don\47t Panic!" }’**

**-| Don’t Panic!**

***awk***在读入任何输入之前，先执行与BEGIN 相关的语句。如果程序中没有了其他的代码了，像这个例子中的情况一样，***awk***就直接结束，而不是试图读取自己不知道如何处理的输入。***‘\47’***是一个小技巧（后面会解释）来将单引号引入程序，这样就不用使用Shell 丑陋的引文伎俩。

提示：如果你使用Bash，你需要先执行命令set+H，然后再执行这个交互式的命令，以禁用C shell 式的命令历史，在这种方式下***‘!’***会被当作特殊字符。我们建议你将这个命令放在你用户的启动文件中。[[10]](#footnote-11)

下面是一个简单模拟cat 工具的***awk***程序；它复制你输入的字符，然后输出到标准输出上（为什么这个可以做得到等会解释）：

**$ awk '{ print }'**

**Now is the time for all good men**

**-| Now is the time for all good men**

**to come to the aid of their country.**

**-| to come to the aid of their country.**

**Four score and seven years ago， ...**

**-| Four score and seven years ago， ...**

**What， me worry?**

**-| What， me worry?**

**Ctrl-d**

### 1.1.3 运行长程序

有时，***awk***程序非常长，在这种情况下，更方便的方法是将程序放在一个独立的文件中。为了告诉***awk***执行某个文件中的程序，你敲入：

**awk -f source-file input-file1 input-file2 ...**

-f 参数指示***awk***工具从***source-file***中读取程序（查看*，****在第 35 页***）。任何的文件名都可以作为***source-file***。如你可以输入下面命令行：

**BEGIN { print "Don’t Panic!" }**

到文件***advice***中，然后执行下面的命令行：

**awk -f advice**

下面的代码做一样的事情：

**awk 'BEGIN { print "Don\47t Panic!" }'**

这个已经早解释过（查看*，****在第 21 页***）。注意，当你用–f 参数指定文件名时，你不需要把文件名引起来，因为大多数文件名都不会含Shell 特殊字符。在文件advice 中，***awk***程序并没有把文件内容给引起来。引号只在命令行上输入程序才需要。（当然，把程序代码放入文件中，使得我们可以直接使用引号（‘’），而不需要像***‘\47’***这样的进行输入。

如果你想清楚地标识这样的***awk***程序，你可以给文件加上扩展名 .***awk***。这不会影响awk 程序的执行，但确实使“家务事”容易了许多。

### 1.1.4 可执行的 awk 程序

如果你学过***awk***，你可能就会想用***‘#!’*** 这样的机制写一个自包含的***awk***脚本。在很多系统[[11]](#footnote-12)上你都可以这么干。例如，你可以把***advice***文件写成这样：

**#! /bin/awk -f**

**BEGIN { print "Don’t Panic!" }**

把这个文件设置成可执行权限（用 chmod 工具），在Shell中简单地输入***‘advice’***，系统就会将awk执行成像你输入了***‘awk –f advice’***一样：

**$ chmod +x advice**

**$ advice**

**-| Don’t Panic!**

（我们假设你已经将你的当前目录放在了你的Shell 的搜索路径变量下【一般为 $PATH】。如果没有这么做，你可以敲入***‘./advice ’***就可以了。）

自包含的awk 脚本在用户调用脚本但不需要知道程序是用***awk***实现时非常有用。

|  |
| --- |
| 理解***‘#!’***  Awk 是一个解释型语言。这表示，***awk***工具把你的程序读入后，会根据你程序中的指令来处理你的数据。（这不同于C这样的编译型语言，首先你需要将它们编译成机器代码，这些代码可以直接被你系统的处理器执行。）***awk***工具因此被称为解释器（interpreter）。多数的现代语言都是解释执行的。  以***‘#!’***开始的行列出了要执行的解释器文件的完整路径，包括一个单一的可选命令行参数要传递给对应的解释器。操作系统然后用给定的参数执行解释器以及所有的参数列表执行被执行的程序。第一个参数即是***awk***程序的完整路径。其他的参数，要么是***awk***的参数，要么是数据文件，或者二者兼有。（注意，在很多的系统上，***awk***会在 /usr/bin 目录下找到，而不是在 /bin 目录下。）  有的系统会将解释器的长度限制在 32 个字符。但这种情况用软连接就可以解决。  在***‘#!’***行，你在***awk***的路径后面最好不好放超一个参数。这不会有作用。操作系统会将那一行剩下的内容当作一个参数，并传递给***awk***。这会导致一些令人疑惑的行为——多数像是***awk***使用方法的一些诊断信息。  最后，ARGV[0] 参数的值（查看see *，****在第 155 页***）会根据系统的同而不同。有的系统会把***‘awk’***作为它的值，而有的会将***awk***的完整路径赋值给它（如 /bin/***awk***）。有一些会将脚本的文件名（***‘advice’***）赋给它。所以，不要依赖于 ARGV[0] 的值来提供给你的脚本使用。 |

### 1.1.5 *awk*程序中的注释

注释，就是程序中给人读的文本；它不是可执行程序的一部份。注释可以解释程序是做什么的，以及如何做的。几乎所有的程序语言都会提供注释，因为程序如果没有它们是非常于难理解的。

在***awk***语言中，注释以***‘#’***开头然后持续到一行的结束。***‘#’***不一定要是一行的开始字符。***awk***会忽略数字符号***‘#’***后面的内容。如我们已经在 advice 中输入了下面的内容：

**# This program prints a nice， friendly message. It helps**

**# keep novice users from being afraid of the computer.**

**BEGIN { print "Don’t Panic!" }**

你可以将注释也带进命令行中的“用后即扔”***awk***程序中，但是一般来讲，这个用处不大；注释的目的是帮你或者其他人后面阅读时理解程序。

注意：如*，****在第 20 页***，所提到的，你可以将一个中等大小的程序用单引号引起来。这样就使得你的Shell脚本是自包含的了。你这么做的时候，你不要将撇号（’）放到你的注释中（或者你的程序的任何位置）。Shell 会将闭引号当做整个程序的结束。结果就是，Shell 会打印出引号不匹配的错误，而如果***awk***实际是执行了，也会打印出语法错误等奇怪的信息。如下面的例子中：

**$ awk 'BEGIN { print "hello" } # let's be cute'**

**>**

Shell首先看到前面两个引号匹配，然后在后面的注释又看到了一个开始的引号。因此就有第二个提示，并等待输入。对于Unix ***awk***，闭引号串会产生这样的结果：

**$ *awk*  '{ print "hello" } # let's be cute’**

**>’**

**error*awk*: can’t open file be**

**error source line number 1**

在撇号的前面加一个转义符（\）也没有作用，因为转义符在单引号中也不是一个特殊的字符。后面一节中我们会解释Shell的引号规则。

### 1.1.6 Shell引号问题

对于短到中等大小的***awk***程序而言，最方便的方式是将程序在命令行中输入。这个通过将整个程序包含在一对单引号当中即可。无论在交互式的输入程序，还是在一个大的Shell 脚本中输入都是可以的：

**awk 'program text' input-file1 input-file2 ...**

一旦你通过Shell 来运行，对于Shell 的引号规则有所了解总是好的。下面的规则只应用于POSIX 兼容的Bourne-式的Shell 中（如Bash，GNU Bourne-Again Shell）。如果你用的是C Shell，那就需要你自处了。

在深入规则之前，我们先介绍本书中通书所使用的概念，它们就是null，或者empty 串。

null string 就是在字符数据中没有值。换句话说，就是空的。在***awk***程序中写成这样：””。在Shell中，它可以写成单或者双引号的形式：”” 或者’’。虽说null string 中没有字符，但是确实是存在的。例如下面的指令：

**$ echo ""**

在这里，echo 工具收到单一的参数，就算参数中没有字符在其中。在本书的后续部分，我们使用术语 null string 以及 empty string 来表示这样的串。现在在引号规则中：

* 被引起来的项可以与没有被引起来项连接，也可以与已经引起来的项进行连接。Shell 会将它们全部转换成命令的参数。
* 在引起来的串中，在任何单个字符前的反斜杠（’\’）都会被Shell 移除，而将被引起来的字符传递给命令。
* 单引号保护在开与闭引号之间的所有字符。Shell 不会对串中的东西做任何解析，并原样地传递给命令。不可能在一个单引号中嵌入单引号。参考，***在第 23 页***，中的例子，看看如果你这么做会有什么结果。
* 双引号保护开引号与闭引号之间的大多数据字串。Shell 只做最小的变量与命令的替换。不同的Shell 对于双引号的处理是不一样的。

由于在双引号中，特定的字符是会被Shell处理的，因此他们在串中一定要进行转义。要注意这些符号***‘$’***，***‘‘’***， ***‘\’***， 和***‘"’***。这些符号如果要原样地传递给程序中，你就需要在串中用反斜杠（***‘\’***）进行转义。（前导的反斜杠（’\’）会先被过滤掉。）你可以参考之前的例子，在这里*，****在第 21 页***：

**awk 'BEGIN { print "Don\47t Panic!" }'**

可以写成这样的形式：

**$ awk "BEGIN { print \"Don’t Panic!\" }"**

**-| Don’t Panic!**

* 注意双引号中的单引号。空串（Null string）如果出现在非空串中时，会被移除，而显式的空串会被保留。例如：可以将域分隔符FS 指定为空串（null string），可以这么来：

**awk -F "" 'program' files # correct**

而不要这么来：

**awk -F"" 'program' files # wrong!**

在第二个例子中，***awk***会尝试使用程序的文本作为FS 的值，第一个文件名会做为程序的文本！这会导致在最好的情况下是语法错误，而最坏的情况下会导致令人费解的行为。

混用单双引号是比较麻烦的。你需要用一些Shell 的伎俩，如：

**$ awk ’BEGIN { print "Here is a single quote <’"’"’>" }’**

**-| Here is a single quote <’>**

这个程序中包括三个连接的引串。第一个与第三个都是单引号串，而第二个是双引号串。这个其实可以简化为：

**$ awk ’BEGIN { print "Here is a single quote <’\’’>" }’**

**-| Here is a single quote <’>**

这个得要你自己看哪个更好读。

另一个使用双引号的选择，就是转换内嵌的，***awk***层的双引号。

**$ awk "BEGIN { print \"Here is a single quote <’>\" }"**

**-| Here is a single quote <’>**

这种选择会非常痛苦，因为双引号，反斜杠（’\’）以及美元符号（$）在高级的awk程序中是很普遍的。

第三个选择是使用转义的八进制序列（查看see *，****在第 51 页*** 查看单双引号字符的对应码），如下：

**$ awk 'BEGIN { print "Here is a single quote <\47>" }'**

**-| Here is a single quote <’>**

**$ awk 'BEGIN { print "Here is a double quote <\42>" }'**

**-| Here is a double quote <">**

这看起来就漂亮多了，但是你需要注释好所转义的东西是什么。第四个选择就是使用命令行式的变量赋值，如下：

**$ awk -v sq="’" 'BEGIN { print "Here is a single quote <" sq ">" }'**

**-| Here is a single quote <’>**

（在这里，两个串常量以及sq 的值会被连接起来成为一个单一的串，然后用print 打印出来。）

如果你确实在程序中即需要单引号也需要双引号，你最好是将它们移到一个单独的文件中，在文件里，Shell 不会解析你的程序所要做的事情。

#### 1.1.6.1 MS-Windows批处理文件中的引号问题

尽管本书中一般只针对POSIX 系统以及POSIX Shell，下面的问题也值得为用户提出来。

微软Windows 系统中的Shell 用双引号来引字串。因此，比较难或者不可能在命令行脚本中通过转义的方式来包含双引号。多谢Jeroen Brink 提供，下面的例子，呈现文件中的每一行都并用双引号引起来：

**gawk "{ print \"\042\" $0 \"\042\" }" file**

## 1.2 例子中使用的数据文件

本书中所使用到的例子，主要从两个例子数据文件中输入数据。第一个是***mail-list***，里面展示了一系列的人名，同时还有他们的邮件地址，以及相关的个人信息。第二个文件为***inventory-shipped***，包含了每月的发货情况。两个文件中，每一行都称为一条记录。

在邮件列表中，每个记录包括了人名，他/她的电话号码，他/她的邮件地址，以及他/她与作者本人的关系。列用行来进行分隔。最后一列中的‘A’ 表示相识的人。‘F’ 表示相应的人是朋友关系，‘R’ 表示相应的人是亲属关系。

**Amelia 555-5553 amelia.zodiacusque@gmail.com F**

**Anthony 555-3412 anthony.asserturo@hotmail.com A**

**Becky 555-7685 becky.algebrarum@gmail.com A**

**Bill 555-1675 bill.drowning@hotmail.com A**

**Broderick 555-0542 broderick.aliquotiens@yahoo.com R**

**Camilla 555-2912 camilla.infusarum@skynet.be R**

**Fabius 555-1234 fabius.undevicesimus@ucb.edu F**

**Julie 555-6699 julie.perscrutabor@skeeve.com F**

**Martin 555-6480 martin.codicibus@hotmail.com A**

**Samuel 555-3430 samuel.lanceolis@shu.edu A**

**Jean-Paul 555-2127 jeanpaul.campanorum@nyu.edu R**

发货清单中的数据表示了在一年中的发货信息。每个记录都包括一个月绿色集装箱的发货数，红箱盒装的发货数统计，橙色的袋装的发货数统计，以及蓝色包装的发货数统计。这里面有16条记录，包括了了上一年最后12个月的数据，以及本年的前4个月的数据。用空行分隔了年份：

**Jan 13 25 15 115**

**Feb 15 32 24 226**

**Mar 15 24 34 228**

**Apr 31 52 63 420**

**May 16 34 29 208**

**Jun 31 42 75 492**

**Jul 24 34 67 436**

**Aug 15 34 47 316**

**Sep 13 55 37 277**

**Oct 29 54 68 525**

**Nov 20 87 82 577**

**Dec 17 35 61 401**

**Jan 21 36 64 620**

**Feb 26 58 80 652**

**Mar 24 75 70 495**

**Apr 21 70 74 514**

上面的文件包含在 gawk 发布版本，位于目录awklib/eg/data 中。

## 1.3 一些简单例子

下面的命令行执行了一个很简单的***awk***程序，就是查找邮件列表文件中的字串‘***li***’（一组字符称为一个串，这个是基于英语中的术语串的用法，如“一串珍珠”或者“火车中的一串汽车”）：

**awk’/li/ { print $0 }’ mail-list**

如果行中包括了***‘li’***，他们就会被打印出来，因为***‘print $0’***的意思就是打印当前行。

你应该注意到***awk***程序中在串***‘li’***两边的斜杠（‘/’）。这个斜杠指示了***‘li’***是要搜索的东西。这种模式称为正则表达式，这个知识点会在后面涉及（查看*，* ***在第 50 页***）。这个模式允许匹配单词的部分。在***awk***程序两边由单引号引起来，因此不会解析其中的任何字符。

下面是打印输出。

**$ *awk*’/li/ { print $0 }’ mail-list**

**-| Amelia 555-5553 amelia.zodiacusque@gmail.com F**

**-| Broderick 555-0542 broderick.aliquotiens@yahoo.com R**

**-| Julie 555-6699 julie.perscrutabor@skeeve.com F**

**-| Samuel 555-3430 samuel.lanceolis@shu.edu A**

在***awk***的规则中，即可以省略模式，也可以省略相关的动作，但不能两者都省略。如果模式省略了，那么动作的执行就会针对所有的输入行。如果动作被省略了，那么默认的运行就是打印所有的匹配行。

因此，我们不管例子中的动作（即***print***语句以及花括号），结果也是一样的：***awk***会打印所有匹配了***‘li’***字串的行。如果省略了***print***语句，而保留花括号，则动作会什么也不做（即不会有行被输出）。

实际的***awk***程序大多只有一两行长。下面收集了一些有用而短小的程序，以便于你开始使用***awk***。其中有些程序包括的特性目前还没有涉及（对于程序的描述可以提示你接下来将做什么，但不需要你先读完本书后成为专家后再来阅读程序！）大多数的例子都使用名为data 的数据文件。这仅是个占位符；如果你自己使用这些程序，替换成你自己的数据文件就好了。为了后续参考：一般来讲要完成同样一件事，***awk***可以有不止一种方法。在某个时候，你可能想回过头来看这些例子，看是否能够想出更多方法来做这里展示的相同的事情。

* 打印大于80个字符的行：

**awk 'length($0) > 80' data**

这里唯一的规则就是有一个关系表达式作为模式，但是没有动作。因此它用的是默认动作，就是打印记录（输入行）。

* 打印最长的行的长度：

**awk '{ if (length($0) > max) max = length($0) }**

**END { print max }' data**

这里与END 关联的代码，会在所有的行读入后执行；它与BEGIN刚好相反。

* 打印数据中最长的行的长度：

**expand data | awk’{ if (x < length($0)) x = length($0) }**

**END { print "maximum line length is " x }’**

这个例子与前面的有一些不同：输入是通过expand 工具，将TABs 转换成空格。因此长度的比较实际上是到最右边的列数，而不是每一行的输入字符数。

* 打印至少有一个域的行：

**awk’NF > 0’ data**

这个是最好的删除空白行的方法（即是，创建一个与旧文件一样的文件，但是移除了其中的空白行）。

* 打印从 0 到 100 （包含）的7个随机数：

**awk ’BEGIN { for (i = 1; i <= 7; i++)**

**print int(101 \* rand()) }’**

* 打印多个文件中总共的字节数：

**ls -l files | awk ’{ x += $5 }**

**END { print "total bytes: " x }’**

* 打印多个文件所使用的 KB 总数：

**ls -l files | awk ’{ x += $5 }**

**END { print "total K-bytes:"， x / 1024 }’**

* 打印所有用户的登录名的排序列表：

**awk -F: ’{ print $1 }’ /etc/passwd | sort**

* 打印文件的行数：

**awk 'END { print NR }' data**

* 打印文件中的偶数行：

**awk’NR % 2 == 0’ data**

如果你使用表达式***‘NR % 2 == 1’***，则程序将打印出奇数行。

## 1.4 两规则的例子

***awk***工具从文件中一次读取一行。对于每一行，***awk***都是尝试与每个模式进行匹配。如果有几个模式与其匹配，则几个动作都会按照匹配的顺序被执行。如果没有模式匹配上，则不会有动作会执行。

当处理了匹配行的所有规则后（也许也没有），***awk***会读取下一行。（但是请看，***在第 153 页***，与，***在第 154 页***，中的相关知识。）这个过程一直持续到文件读取完毕。例如，下面的***awk***程序包含了两个规则：

**/12/ { print $0 }**

**/21/ { print $0 }**

第一个规则用字符串‘12’ 作为模式，以及‘print $0’ 作为动作。第二个规则用‘21’ 作为模式，也用‘print $0’ 作为动作。每个规则的动作都用自己的花括号包含起来。

这个程序打印每一行包含有‘12’或者‘21’的行。如果有一行包含了两个串，则会打印两次，每个规则一次。

如果我们同时对我们的两个数据文件，即 mail-list 与 inventory-shipped 执行这个程序会有什么结果：

**$ awk '/12/ { print $0 }**

**> /21/ { print $0 }' mail-list inventory-shipped**

**-| Anthony 555-3412 anthony.asserturo@hotmail.com A**

**-| Camilla 555-2912 camilla.infusarum@skynet.be R**

**-| Fabius 555-1234 fabius.undevicesimus@ucb.edu F**

**-| Jean-Paul 555-2127 jeanpaul.campanorum@nyu.edu R**

**-| Jean-Paul 555-2127 jeanpaul.campanorum@nyu.edu R**

**-| Jan 21 36 64 620**

**-| Apr 21 70 74 514**

留意在mail-list 文件中的***‘Jean-Paul’***开头的行，它被打印了两次，一个规则一次。

## 1.5 复杂一点的例子

目前我们已经掌握一些基本的技能，现在我们再看一下***awk***典型地还能做些什么事情。这个例子展示了***awk***如何被用来做汇总，选择以及重新安排另一个工具的输出。所用到的特性还没有涉及，因此你不要担心不理解所有的细节。

**ls -l | awk ’$6 == "Nov" { sum += $5 }**

**END { print sum }’**

这个命令打印当前目录下，最后的修改时间为十一月份的所有文件的总共的字节数。‘ls –l’是这整体命令的一部分，用于显示一个目录下的文件，包括文件的大小以及最后修改时间。它的输出类似于这样：

**-rw-r--r-- 1 arnold user 1933 Nov 7 13:05 Makefile**

**-rw-r--r-- 1 arnold user 10809 Nov 7 13:03 awk.h**

**-rw-r--r-- 1 arnold user 983 Apr 13 12:14 awk.tab.h**

**-rw-r--r-- 1 arnold user 31869 Jun 15 12:20 awkgram.y**

**-rw-r--r-- 1 arnold user 22414 Nov 7 13:03 awk1.c**

**-rw-r--r-- 1 arnold user 37455 Nov 7 13:03 awk2.c**

**-rw-r--r-- 1 arnold user 27511 Dec 9 13:07 awk3.c**

**-rw-r--r-- 1 arnold user 7989 Nov 7 13:03 awk4.c**

第一个域包含了读写权限，第二个域包含了文件的硬连接[[12]](#footnote-13)个数，第三个域显示了文件的属主信息。第四个域显示的文件的用户信息。第五个域显示了文件的字节大小。第六、七、八个域分别包括文件最后修改时间的月，日与时间。最后，第九个域包含了文件名。

在***awk***程序中‘$6 == “Nov”’ 就是用来判断‘ls –l’ 输出的第六个域是否为‘Nov’ 的表达式。如果每次的输入行的第六个域匹配字串‘Nov’，***awk***就会执行‘sum+=$5’。这个会加上第五个域（文件大小）到变量sum 中。结果就是，当***awk***完成了所有行的输入时，sum 就包含了匹配模式的所有文件的大小。（这可以工作是因为***awk***的变量会默认初始化为 0。）

最后一行处理后，END 规则就会执行，并打印出sum 的值，在这个例子中为 80600。

这些个高级的***awk***技术会在后面的章节中涉及（查看*，****在第 145 页***）。在你学习到这些高级技术前，你只需要知道***awk***如何解析你的输入并显示你的输出。通过操作各个域并使用print 语句，你就可以产生一些相当有用的，令人印象深刻的报表。

## 1.6 *awk*语句与行

通常情况下，***awk***程序的每一行都是一个独立的语句或者一个独立的规则，类似于：

**awk ’/12/ { print $0 }**

**/21/ { print $0 }’ mail-list inventory-shipped**

但是gawk 会忽略下面符号后面的任何新行。

**， { ? : || && do else**

任何一个新行都会认为是一个语句[[13]](#footnote-14)的结束。为了识别一行的后续字符（续行），（‘\’）反斜杠必须是行的结束字符，就算在串的或者正则表达式的中间。如：

**awk ’/This regular expression is too long， so continue it\**

**on the next line/ { print $1 }’**

我们一般不在我们的程序代码中利用反斜杠来续行。gawk 对于一行的长度没有限制，因此反斜杠续行并非是必要的。这仅是使得程序更有可读性。因为这个原因，也为清晰起见，我们尽量保证书中代码的语句的短小。如果你的代码是写在单独的文件而不是命令行中，那么反斜杠就最有用了。同时你也需要知道不同的发布版本的***awk***对于反斜杠在何处使用是有特殊地方的。因此，为了能最大化你的代码的可移植性，最好不要把你的代码行在字串或者正则表达式中间断行分开。

注意：反斜杠续行不会在C Shell 中有效。它只在独立的awk程序文件，或者一次性的程序中有用，前提是你使用的是POSIX 兼容的Shell，例如Unix Bourne Shell 或者Bash。但是C Shell 的表现不同！你需要在行中使用两个反斜杠，然后再写入新行。在使用C Shell 时你也需要留意awk程序中的每一个新行都需要反斜杠转义。作为说明如下：

**% awk ’BEGIN { \**

**? print \\**

**? "hello， world" \**

**? }’**

**-| hello， world**

***‘$’*** and ***‘>’***.

这里面，‘%’ 与‘？’ 都是C Shell 的主、次提示符，与标准的Shell ‘$’ 与‘>’ 类似。

与前面一个例子相比，看看POSIX 兼容的Shell 如何来完成：

**$ awk ’BEGIN {**

**> print \**

**>"hello， world"**

**> }’**

**-| hello， world**

***awk***是一个面向行的语言。每一个规则的动作都与模式在同一行。为了将它们分隔在不同的行，你必须使用反斜杠来进行续行；没有他选项。

还有一个事情要记住，不要将反斜杠与注释混用。***awk***只要看到‘#’ 开始一个注释，就会忽略一行中的后续部分。例如：

**$ gawk ’BEGIN { print "dont panic" # a friendly \**

**> BEGIN rule**

**> }’**

**error gawk: cmd. line:2: BEGIN rule**

**error gawk: cmd. line:2: ^ syntax error**

在这个例子中，看起来反斜杠会将注释续到下一行。但是，反斜杠-新行这样的组合从来不会被注意到，因为它在注释中被藏起来了。因此，BEGIN 会被视作一个语法错误。

当***awk***的一个规则中的语句很短，你想把它们放在同一行，这个可通过语句之间用分号（‘；’）进行分隔。这也同样适应于规则本身。因此，在这节中展示的程序也可以写在这样：

**/12/ { print $0 } ; /21/ { print $0 }**

提示：需要声明的一点是，用一个分号分隔同一行中的规则并不是awk语言的首创。为了与在一个动作的语句处理保持一致，这才加入到awk中的。

## 1.7 *awk*的其他特性

***awk***语言提供了非常多的预定义的，或者内建的变量，可让你的程序从***awk***获取信息。也有些变量集是让你的***awk***程序控制***awk***如何处理你的数据。

另外，***awk***也提供了很多的内建函数，来做一些通用性的计算与字串相关的操作。gawk 提供了内建的函数用来处理时间戳，执行位操作，以及运行时的字串翻译（国际化），确定变量的类型以及数据排序。

当我们开发***awk***语言程序时，我们会介绍多数的变量与函数，他们会在，***在第 155 页***，以及，***在第 182 页***，有系统的描述。

## 1.8 何时使用*awk*

目前，我们已经了解了一些***awk***可以做的事情，你可能有疑问***awk***对你有什么用处。通过使用其他工具程序，更高级的模式，域分隔符，算术语句以及其他的选择方案，你可以产生更为复杂的输出。***awk***对于从大量的原始数据产生报表的场景中非常用有，如从其他的工具，如 ls 的输出中汇总信息。（查看 ，***在第 30 页***）

用***awk***语言写的程序一般会比较其他的语言所写的程序要短。这使得***awk***程序非常容易组合与使用。***awk***常常在“用后即丢”的场景中快速实现你的代码。由于***awk***是解释型的，这就避免了在编辑-编译-测试-调试这样的程序开发流程的编译流程，而这个流程通常都比较长。

也有非常复杂***awk***程序，包括一个完整的可重定位的为8位微处理写的汇编器（查看*，****在第 453 页***，取得详细信息），以及为专用的Prolog 计算机写的微代码的汇编器。起初的***awk***的能力没有办法完成这么复杂的任务，但是现代版本的***awk***则具备了这样的能力。

如果你发现你可能需要写好几百行的程序代码，你或许应该考虑使用其他的编程语言，Shell 就对字串处理与匹配就很在行；另外，它也可以让你充分使用系统中的强大工具。Python 在高层级上的语言编程方便性与使用系统工具之间提供了很好的平衡。[[14]](#footnote-15)

## 1.9 总结

* ***awk***程序由模式-动作对组成。
* 没有模式的动作也会运行。对于没有模式的默认动作就是‘{print $0 }’。
* 要么使用***awk*** ‘program’ file 或者***awk*** –f program-file files 来运行***awk***。
* 你可以使用特殊的‘#！’ 字串放在***awk***程序的第一行来直接执行。
* ***awk***程序的注释以‘#’ 开始，并持续到同一行的结束。
* 当在写大的脚本的一部份时，awk时候要注意引号问题（或者MS-Windows 的批处理文件）。
* 可以使用反斜杠来进行源代码续行。也可以在逗号‘，’，开大括号‘{’，问号‘？’，冒号‘：’，‘||’，‘&&’，‘do’，与‘else’后面自动续行。

# 第二章 运行*awk*与 gawk

本章的内容包括如何来运行awk，介绍POSIX 标准的与gawk 特定的命令行选项，以及awk 与gawk 不带参数运行时的情形。后面再涉及gawk 如何来搜索文件，在读取其他文件的同时从标准输入读取数据，gawk 的环境变量，退出状态，使用包含文件，以及那些过时的，未文档化的选项与特性。

本章中的许多特性与选项在后面的章节中会更详细地说明。可以跳过本章中那些你现在不感兴趣的内容。

## 2.1 调用*awk*

有两种方式来运行awk——带显式程序的方式或者是一到多个的程序文件方式。下面是两种运行方式的模式；在 [] 中包括内容是可选。

**awk [options] -f progfile [--] file . . .**

**awk [options] [--] 'program' file . . .**

除支持传统的单字母的POSIX 选项外，gawk 也支持GNU 长格式的选项。也可以用空程序的方式调用awk：

**awk ’’ datafile1 datafile2**

但是，这样做基本没什么用。当程序内容为空时，awk 会安静地退出。如果在命令行中进行了指定了--lint 参数，gawk会警告说，程序为空。

## 2.2 命令行参数

选项以横杠开始，后面接一个字符。GNU 风格的长选项，则是两个横杠后面带关键字。关键字可以是缩写的，只要缩写使得选项是可识别的唯一标识即可。如果选项带参数，要么是关键字后面直接跟一个等于号（=）加上参数的值，要么是关键字与参数的值通过空格分隔。如果特定的选项有多个值，那么只有最后一个值有效。

gawk 的每一个长选项都有一个对应的POSIX风格的短选项。长短选项在所有的上下文中都可以互相替换的。下面的列表列出了POSIX 标准中规定的选项：

***-F fs***

***--field-separator fs***

设置FS 变量的值为fs（查看***，在第 71 页***）。

***-f source-file***

***--file source-file***

从文件中读取awk 程序，而不是第一个非选项的参数。这个选项可以指定多次；awk 的程序可以由每一个指定的源程序文件连接而成。

***-v var=val***

***--assign var=val***

在执行程序之前，设置变量值为val。这个变量在BEGIN 规则中可用（查看***，在第 41 页***）。

-v 选项只可以设置一个变量；但是可以使用多次，如此来设置另一个变量的值：‘awk -v foo=1 -v bar=2 ...’。

注意：使用–v 选项设置内置变量的值可能导致令人惊讶的结果。awk 在需要的时候，可以重新设置这些变量的值，而忽略你指定的任何初始值。

***-W gawk-opt***

选择一个实现有关的选项。这是为特定实现而提供的一种POSIX 转换。这个选项也有一个GNU 风格的长选项。注意长选项是可以缩写的，只要保证缩写是唯一的即可。特定于gawk 的选项的完整列表下面会列出。

***--***

指示命令行选项的结束。后续的参数不会被识别为选项，即使它们以‘-’开始。

对于–- 的解释是参照POSIX 的参数分析习惯。如果你有一个文件名以‘-’开始，这个参数则非常有用，或者在Shell 脚本里，文件名会由用户来指定。在把参数传递给awk 程序上，这个参数也非常有用。查看，***在第 237 页***。

下面的选项是特定于gawk 的选项：

***-b***

***--characters-as-bytes***

指定gawk 将输入的数据按照单字节的方式对待。另外，所有的由 print 或者 printf 的输出也以单字节的方式对待。

正常情况下，gawk 会按照POSIX 的标准，按照当前语言设置来处理输入数据（查看***，在第 136 页***）。这常常导致将多字节字符转换为宽字符（内部处理了），当输入的数据不是有效的多字节字符时，这可能导致一些问题或者疑惑。这个选项简单地告诉gawk，“别碰我的数据！”

***-c***

***--traditional***

指定兼容模式，在这种模式下awk 的GNU扩展会被禁用，这样gawk 的表现就跟BWK awk 一致。查看，***在第 406 页***，其中汇总了所有的扩展。也可以参考，***在第 439 页***。

***-C***

***--copyright***

打印简单版本的通用许可证协议（General Public License，GPL）然后退出。

***-d[file]***

***--dump-variables[=file]***

打印经过排序的全局变量，以及它们的类型，以及最终值到file中。如果没有提供file参数，就把这个列表打印到一个awkvars 的文件中，并输出在当前目录。如果提供了file参数，在–d 参数与file之间不能有空格。

查看全局变量列表，是一种查找程序中输入错误的一种好方法。如果你的程序很大，而函数很多时，可以使用这个选项，来确定你的函数是否有这种情况，即本来使用本地变量却错误地使用了全局变量。（这个在将变量名命令为i，j 等等名字的情况下尤其容易出错。）

***-D[file]***

***--debug[=file]***

启用awk程序调试（查看，***在第 320 页***）。默认情况下，调试器从键盘中交互式地读入命令。可选参数file 可以让你指定在文件中的命令行列表，而不是交互地输入。如果你提供了file 参数，则在–D 与file 之间不能有空格。

***-e program-text***

***--source program-text***

在***program-text***中指定源代码。这个选项可以让你混用通过文件与通过命令行来提供源代码的方式。这个选项在你有自己的库函数要使用，但是你又通过命令行来转入程序的情况下非常有用。（查看***，在第 43 页***）。

***-E file***

***--exec file***

与–f 参数类似，awk 从文件中读取程序。与–f 相比，有两个不同：

* 这个选项会结束选项的处理；任何在命令行上后续的其他参数都将直接传递给awk 程序。
* 命令变量赋值形式***‘var=value’***也将禁止。

这个选项在写WWW CGI 程序时通过URL 传统参数时非常有用。通过这个选项可以防止一些有害的（或者其他的）用户将选项，赋值以及awk 源代码（通过 –e 选项）传递给CGI 应用。[[15]](#footnote-16)这个选项需要与 ‘#!’脚本一起使用（查看 ***，在第 22 页***），例如这样：

**#! /usr/local/bin/gawk -E**

**awk program here ...**

***-g***

***--gen-pot***

分析源程序中的已经做了翻译标志所有字串常量，并产生一个GNU gettext 可移植的对象模板文件到标准输出上。查看*，* ***在第 311 页***，来取得这个选项的相关信息。

***-h***

***--help***

打印“使用方式”信息，汇总gawk 接收的长、短风格的选项然后退出。

***-i source-file***

***--include source-file***

从***source-file***读取awk 源库。这个选项与在你的程序中的***@include***等同。也非常类似于–f 选项，但是有两个重要的不同。首先，当–i 指定时，如果源程序之前已经装载过了，则不会再被装载，而当使用–f 时，gawk 总是会装载源文件。第二，由于这个选项目的是用于库类型的代码，gawk 不会将这些文件识别成主程序的连续输入。因此，在处理了–i 参数后，gawk 还是希望通过命令行的–f 参数来取得主程序源代码。

***-l ext***

***--load ext***

载入一个名为ext 的动态扩展。扩展存储为系统的共享动态库。这个选项使用 AWKPATH 变量来搜索库。正确的库后缀会根据系统默认提供，因此你不需要指定扩展名。扩展的初始化函数为***dl\_load()***。另外一个选项就是通过在程序中通过***@load***关键字来装载共享库。这个高级特性的详细说明在*，****在第 349 页***。

***-L[value]***

***--lint[=value]***

对于可疑的或者不可移植到共他awk 实现的结果提出告警。如果提供了value，则与–L 参数之间不可以有空格。一些警告会在第一次读入程序时就会给出。而其他的一些会在运行时给出。当可选参数的值是 ‘fatal’，lint[[16]](#footnote-17)警告就成了致命错误。这可能有点夸张，但是这确实能够刺激开发者开发出更加“干净”的awk 程序。如果可选参数的值为***‘invalid’***，则只有在确实是无效的情况下才会发出告警。（这个还没有完全实现）

有些告警只打印一次，就算这样的可疑结果在你的awk 程序中出现多次。因此，当去除了由–-lint 所报告的问题时，你应该要仔细检查你的程序中所有不恰当的结构。如果你的程序不长，做这个工作也不会成为什么负担。

***-M***

***--bignum***

强制对于数据计算使用任意精度。这个选项如果在gawk 编译时没有使用GNU MPFR 以及MP 库时，不会起作用。（查看*，****在第 349 页***）

***-n***

***--non-decimal-data***

启用自动的对输入数据的八进制与十六进制值的自动解释（查看***，在第 297 页***）。

注意：这个选项有可能严重破坏旧的程序，因此要小心使用。同时也提示，这个参数可能在gawk 的后面的版本会被取消。

***-N***

***--use-lc-numeric***

当分析数值输入数据时，强制使用当前的语言设置(current locale)的十进制小数点字符。（查看***，在第 136 页***）

***-o[file]***

***--pretty-print[=file]***

启用awk 程序的简单打印。默认情况下，输出程序被创建在一个文件名为***awkprof.out***文件中（查看，***在第 307 页***）。可选参数file 可以让你可以指定一个不同的文件用于输出。如果你提供了file 参数，则–o 选项与file 之间不能有空格。

提示：由于gawk 演化的方式，有这个参数时，你的程序依然会执行。这个会在下一个主要版本发生变化，因此gawk 仅简化打印程序，而不是执行它。

***-O***

***--optimize***

启用某些程序内部表达式的优化。在此时，这个仅仅包括一些简单的常量折叠。[[17]](#footnote-18)

***-p[file]***

***--profile[=file]***

启用awk 程序采样分析（查看，***在第 307 页***）。默认情况下，采样分析会生成文件名为 awkprof.out。可选参数file 可以让你指定一个不同的文件名。如果你指定了file 参数，则在–p 选项与file 参数之间不能有空格。采样分析结果中在左边包括程序中的每条语句的执行次数，以及每个函数的函数调用次数。

***-P***

***--posix***

严格按POSIX 模式来。这个禁用所有的gawk 扩展（就像--traditional一样），并禁用所有不属于POSIX 的扩展，查看，***在第 414 页***，了解gawk 中通过这个参数被禁用的所有扩展。同时，也附加如下限制：

* 当FS 等于单个空格时，新行不会像空格一样用来进行域分隔。查看，***在第 67 页***。
* 在‘？’ 或者‘：’ 后不能断行（查看***，在第 132 页***）。
* 指定‘-Ft’ 时，不能将FS 的值设置为单一的TAB 字符（查看***，在第 71 页***）。
* 语言设置中的十进制小数点字符用来分析输入数据（查看，***在第 136 页***）。

如果你即提供了--traditional 与--posix 参数在命令行中，--posix 会优先执行。gawk 则会输出一个警告。

***-r***

***--re-interval***

在 regexps 中允许区间表达式（查看，***在第 53 页***）。现在这个是gawk 的默认行为。这个参数保留下来，即为了身后的兼容性，也为了与–-traditional 组合使用。

***-S***

***--sandbox***

禁用 system () 函数，用 getline 进行输入重定向，用 print， printf 以及动态扩展来进行输出重定向。当你要执行来源有有疑问的awk 源代码时，为了确保脚本不会访问你的系统（而不是通过指定输入数据文件），这个参数尤其有用。

***-t***

***--lint-old***

当相应的结构在 Version 7 Unix起初始的awk 版本中不可用时，发出告警。（查看，***在第 404 页***）

***-V***

***--version***

打印这个本gawk 的版本信息。这个可以让你确定你的gawk 版本是不是与Free Software Foundation 发布的最新版本匹配。在Bug 报告中也非常有用。（查看***，在第 434 页***）

只要提供了程序文本，如果有选项由于警告而被标志为无效，忽略它就可以了。

在兼容模式里，有个特殊的地方，如果果 fs 的值通过 -F 参数设置为‘t’，那么 FS 就会被设置成 TAB 字符（“\t”）。这种情况只有在使用 --traditional 时才会为真，而不是 --posix（查看，***在第 71 页***）。

-f 参数可以在命令行中使用多次。awk 将从多个文件中读取程序，就好像这些文件已经合并成了一个大的文件。这种方式在创建 awk 函数库时非常有用。这些函数只需要写一次，然后从某个标准的位置取用，而不是在每个单独的文件中包含。-i 参数在这个方面的考虑也类似。（正如，***在第 204 页，***中所提到的，这些函数名必须是唯一的。）

标准的 awk 中，就算程序是通过键盘输入的，通过 -f /dev/tty 完成，库函数也还是可以使用的。输入程序后，键入 Ctrl+d （文件结束符）来结束。（你也可以使用 -f - 来从标准输入中读取程序源代码，只是这个时候，你就不能同时使用标准输入作为数据的输入源。

由于标准 awk 中混用源文件与命令行 awk 程序，显得有一些笨拙，所以 gawk 提供了一个 -e 选项。这并不需要你抢占标准输入来输入源代码，而是让你很容易地混合使用命令行与库源代码（查看***，在第 43 页***）。与 -f 一样，-e 与 -i 选项也可以在命令中多次使用。

如果没有 -f 或者 -e 选项指定，gawk 会使用命令行中的第一个非选项参数作为程序的源代码。

如果环境变量POSIXLY\_CORRECT存在，gawk 就会严格地按照 POSIX 模式执行，就如你提供了 --posix 参数一样。许多的 GNU 程序都查找这个环境变量来禁用与 POSIX 标准有冲突的扩展，但是 gawk 的行为有一些不同：它禁用所有的扩展，就算某个特性与 POSIX 没有冲突，然后按照严格的POSIX 标准模式执行。如果指定了 --lint 选项，gawk 会因为POSIXLY\_CORRECT而开启 POSIX 模式，然后会输出警告信息，说 POSIX 模式已经启用。一般来说，你可能会在你的 Shell 的启动脚本中设置这个变量。对于与 Bourne Shell 兼容的 Shell 来说（如 Bash），你可以将下面的行加入到用户主目录的 .profile 文件中：

**POSIXLY\_CORRECT=true**

**export POSIXLY\_CORRECT**

对于与C shell 兼容的 shell，[[18]](#footnote-19)你可以将下面的行加入到用户主目录的 .login 文件中：

**setenv POSIXLY\_CORRECT true**

对于每日的工作环境，设置POSIXLY\_CORRECT变量并不值得建议，但是对于测试你的程序在其他环境中的可移植性则有相当作用。

## 2.3 其他命令行参数

对于命令行中的额外参数，一般都会被当作输入文件来对待，并依次处理。但是具有 var=value（将 value 赋值给变量 var） 形式的参数，却并不作为一个输入文件指定。（查看 .***在第 116 页***）在下面的例子中，count=1 是一个变量赋值，而不是文件名：

***awk* -f program.*awk* file1 count=1 file2**

有的命令行参数都可以通过ARGV 数组被 awk 程序得到（查看***，在第 155 页***）。命令行参数与程序文本（如果有）都在 ARGV 中忽略掉了。其他的参数，包括变量赋值，都包含在了这个数组中。随着对 ARGV 参数的处理，gawk 会设置 ARGIND 的值为 ARGV 数组的索引，以记录当前的元素位置。

通过更改 ARGC 与 ARGV 的值，你可以控制 awk 处理输入文件的方式；这个内容会在，***在第 163 页***，会有更详细的描述。

对文件名参数与赋值变量参数的区分是在 awk 在准备打开下一个输入文件时处理的。在那个执行点，awk 会检查文件名，看它是一个文件还是一个变量赋值；如果是，awk 会设置那个变量而不是打开文件进行读操作。

因此，在处理完所有之前的文件后，变量就会被设置成相应的值。特别的，以这种方式对变量进行的赋值，在 BEGIN 规则（查看***，在第 141 页***）中是不可访问到的，因为这个规则在 awk 开始扫描参数列表之前就已经执行了。

在命令行上所给的赋值变量会按照转义的方式进行处理（查看***，在第 51 页***）。

在一些非常早期的 awk 实现中，当变量赋值参数出现在任何文件名之前，则赋值操作会发生在 BEGIN 规则执行前。awk 的行为因此就不一致了；一些命令行赋值会在 BEGIN 规则中就可以使用，而其他的则不可以。非常不幸的是，一些应用程序却依赖于这样的“特性”。当 awk 变得更一致的时候，-v 选项就用来协调应用依赖于这种旧行为的代码。

变量赋值特性会在设置 RS，OFS，ORS 变量的值时非常有作用，这个变量用来在扫描数据文件时控制输入输出格式。同时，如果 awk 要多次扫描数据文件时，用来控制状态也非常有用。例如：

**awk’pass == 1 { pass 1 stuff }**

**pass == 2 { pass 2 stuff }’ pass=1 mydata pass=2 mydata**

有了变量赋值特性后，用 -F 参数来设置 FS 的值，严格来说就不那么必要了。这里保留下来只是为了兼容历史版本。

## 2.4 命名标准输入

很多情况下，你希望同时从标准输入与文件中输入数据，你希望读一个文件，再从标准输入的管道中读入数据，然后再读另一个文件。

将标准输入命名的方式，对所有版本的 awk，都是使用一个单独的减号，或者说是横杠 ‘-’，例如：

**some\_command | awk -f myprog.awk file1 - file2**

这个例子中，awk 首先读取文件 file1，然后从***some\_command***的输出中读入数据，最后从 file2 中读取数据。

当你使用 getline 来读取文件时，你也可以用“-”来命名标准输入（查看***，在第 85 页***）。

另外，gawk 也可以让你在命令行或者程序文件中指定特殊的文件名 /dev/stdin。一些版本的 awk 也支持这种方式，但这并不是标准。（有些操作系统在文件系统中提供文件 /dev/stdin；而 gawk 总是自己来处理这个文件。）

## 2.5 gawk 使用的环境变量

有很多的环境变量会影响到 gawk 的行为。

### 2.5.1 环境变量 AWKPATH

在大多的 awk 实现中，你可以指定为每个程序文件指定确定的路径名，除非文件就在当前目录。但是，如果通过 -f 或者 -i 选项所提供的文件名中没有包括目录分隔符‘/’，gawk 则会在一系列的目录中进行搜索（称为搜索路径），一个接一个地查找是否有指定的文件名。

这些搜索路径包括一系列的由冒号[[19]](#footnote-20)分隔的路径名。gawk 从 AWKPATH 环境变量中取得它的搜索路径。如果不存在这个变量，或者这个变量为空，gawk 会使用默认的路径（后面说明）。

搜索路径特性用来创建通用的 awk 函数库非常有用。库文件可以放在一个标准的默认目录中，然后在命令行中指定一个短的文件名。不然的话，你需要输入每个文件的完整路径。

通过使用 -i 或者 -f 选项，你的命令行 awk 程序可以利用 awk 的库文件（查看*，****在第 221 页***）。如果是在兼容模式中，路径搜索并没有实现，因此，而对于--traditional 与 --posix模式，搜索路径不可用。查看.***在第 35 页***。

如果源代码文件没有在初始的路径中找到，则路径搜索会在文件名后面加一个 .awk 后缀名后再执行搜索。

gawk 的路径搜索机制与 Shell 的机制有点类似。（可以查看 Bourne-Again Shell 手册）它将空的路径项视为当前路径。（一个空项由一个冒号开始，或者以冒号结束的路径，或者两个紧接着的冒号表示[‘::’]。）

提示：要将当前路径包括在路径中，要么在项目中放一个点号 . 或者写一个空的路径项。

gawk 之前的几个不同版本会显示地在当前目录进行搜索，不管是在搜索路径前还是后。在版本 4.1.2 这个版本中，则不再这么处理；如果你希望搜索当前目录，你必须包括一个点 . 作为一个独立的项，要么在搜索路径中加入一个空项。

AWKPATH 的默认路么是：***‘.:/usr/local/share/awk’***。[[20]](#footnote-21)由于 . 包括在了开始处，gawk 会首先搜索当前目录，然后再搜索/usr/local/share/awk。在实践中，这意味着你很少需要更改 AWKPATH 的值。

gawk 将其使用的搜索路径的值放在ENVIRON["AWKPATH"]元素中。这提供了可以让你在程序中访问搜索路径的机制。

尽管你可以在程序中更改ENVIRON["AWKPATH"]的值，但这不会影响正在执行的程序的行为。原因如下：AWKPATH是用来作为搜索程序文件的环境变量。一旦你的程序运行起来，所有的文件都已经搜索到了，gawk 就不再需要 AWKPATH 变量了。

### 2.5.2 环境变量 AWPLIBPATH

AWKLIBPATH环境变量与 AWKPATH 变量类似，但是是由来搜索可装载的扩展（存储为系统的共享库），而不是用来搜索源代码文件，这些扩展由 -l 选项进行指定。如果扩展没有找到，会在搜索完成后加上恰当的共享库后缀后再次搜索。例如，在 GNU/Linux 系统中，后缀是‘.so’。搜索路径也可以用于用***@load***关键词进行的扩展装载（查看***，在第 48 页***）。

如果 AWKLIBPATH 在环境变量中不存在，或者为空，gawk 会使用默认的路径，一般是‘***/usr/local/lib/gawk***’，但这也会由于 gawk 构建的方式而有不同。

gawk 会将搜索路径的值存放于 ENVIRON["AWKLIBPATH"]中。这样，在你的程序中可以得到实际的搜索路径。

### 2.5.3 其他的环境变量

还有很多的环境变量会影响gawk的行为，但是它们都比较特殊。下面的变量都是一般的用户使用：

***GAWK\_MSEC\_SLEEP***

指定连接重试的间隔，以毫秒为单位。在不支持 usleep 调用的系统上，这个值为向上舍入为整数秒。

***GAWK\_READ\_TIMEOUT***

指定 gawk 在返回出错前，等待输入的毫秒数。查看***，在第 90 页***。

***GAWK\_SOCK\_RETRIES***

控制 gawk 重试双向 TCP/IP （Socket）连接的次数。查看***，在第 306 页***。

***POSIXLY\_CORRECT***

导致 gawk 切换到 POSIX 兼容模式，并禁用所有的传统的与 GNU 扩展。查看***，在第 35 页***。

下面的环境变量由 gawk 的开发者用于测试与调测目的。它们会改变。这些变量是：

***AWKBUFSIZE***

这个变量只影响在 POSIX 兼容系统上的 gawk。如果果值是 exact，gawk 会使用输入文件的大小作为内存缓存的大小，来用进行 I/O 的分配。其他情况下的值则是一个整数值，gawk 使用这个值进行缓存的分配。（当这个变量没有设置，gawk 使用文件大小与默认块大小两者中的小的数值，这一般来说者是文件系统的 I/O 块大小。）

***AWK\_HASH***

如果这个变量存在，值为‘gst’，gawk 会切换使用来自于 GNU Smalltalk 的哈希函数来管理数组。这个函数比标准的函数要稍快。

***AWKREADFUNC***

如果这个变量存在，gawk 会切换成一次一行地读取文件，而不是整块地读。这个变量的存在是为了调试在 non-POSIX 操作系统上文件系统的问题，在这个系统上的 I/O 是以记录来进行读取而不是以块的方式。

***GAWK\_MSG\_SRC***

如果这个变量存在，gawk 会在产生的警告信息与致命信息中加入文件名与行号。它的目的是帮助分离消息源，因为有可能有多个地方会产生相同的警告或者错误信息。

***GAWK\_NO\_DFA***

如果这个变量存在，gawk 不会使用 DFA regexp 匹配器[[21]](#footnote-22)来进行“它匹配么？”类型的匹配测试。这会导致 gawk 更慢。它的目的是帮助分隔两个 gawk 内部使用的 regexp 匹配器。（本来应该是没有什么差异的，但是偶尔有时理论与实际相互不太合拍。）

***GAWK\_NO\_PP\_RUN***

注意：这个变量在下个版本发布时不会继续存在。

当 gawk 指定 --pretty-print 选项时，如果存在这个变量，它将不会执行程序。

***GAWK\_STACKSIZE***

指定 gawk 内部求值栈的增长的大小，如果需要增长的话。

***INT\_CHAIN\_MAX***

指定gawk 管理的由整数进行索引的数组中的哈希链的元素的最大数值。

***STR\_CHAIN\_MAX***

指定gawk 管理的由字串进行索引的数组中的哈希链的元素的最大数值。

***TIDYMEM***

如果这个变量存在，gawk 则使用 GNU C 库中的 mtrace () 库调用来跟踪可能的内存泄露。

## 2.6 gawk 的退出状态

如果使用一个数值来与退出语句一起使用（查看 ***，在第 154 页***），gawk 则用这个数值做为它的退出状态数值。

否则，如果在执行中没有发生错误，gawk 则用 C 的常量EXIT\_SUCCESS退出，这一般为 0。

如果出现了错误，gawk 则使用 C 常量EXIT\_FAILURE退出，这一般是 1。

如果 gawk 由于致命错误而退出，则退出状态为 2。在 non-POSIX 系统上，这个值可能会被影射为EXIT\_FAILURE。

## 2.7 在你的程序中包含其他的文件

本节中所描述的特性都是gawk 所独有的。

***@include***关键字可以用来读取 awk 外部的源文件。这赋予了你将大的awk 源程序拆分为更小的，可管理的代码块的能力，也可以让你复用通用的awk 源代码。这就是说，你可以将实现特定任务的 awk 函数组织成外部文件。通过***@include***关键字与 AWKPATH 环境变量搭配使用，这些文件可以像函数库一样被使用。注意，源文件也可以通过–i 选项包含进来。

让我们看个例子。我们启动两个小的（没啥用的）awk 脚本，名字为 test1 与test2。这里是 test1 的脚本：

**BEGIN {**

**print "This is script test1."**

**}**

这里是 test2 的：

**@include "test1"**

**BEGIN {**

**print "This is script test2."**

**}**

用gawk 执行test2 产生了下面的结果：

**$ gawk -f test2**

**-| This is script test1.**

**-| This is script test2.**

gawk 执行 test2 脚本，而这脚本通过***@include***关键字包含了 test1。因此，包含外部的awk 源代码，你可以只使用***@include***，后面加上要包含的文件名，并用双引号引起来。

提示：要记住，这是个语言结构，并且文件名不能是个串变量，而只能是一个在双引号中的代表其自身的字串常量。

文件包含有可能是内嵌的；例如，如果有第三个脚本，名字为 test3：

**@include "test2"**

**BEGIN {**

**print "This is script test3."**

**}**

用gawk 执行脚本 test3 则会产生下面的结果：

**$ gawk -f test3**

**-| This is script test1.**

**-| This is script test2.**

**-| This is script test3.**

文件名也可以是一个目录名。如：

**@include "../io\_funcs"**

或者：

**@include "/usr/awklib/network"**

上面两个格式都是正确的。AWKPATH 环境变量在使用***@include***时会产生巨大好处。在***@include***使用 AWKPATH 环境变量的规则，在命令行搜索中也是一样适应的。（查看，***在第 43 页***。）

用这来构建gawk 的函数库非常有用。如果你有一个大的，非常有用的通用awk 函数脚本，你就可以将他们拆分为一个个库文件，然后将这些文件放在一个特定目录。然后你可以将这些“库”包含进来。你可以使用文件的全路径，也可以通过设置 AWKPATH 环境变量，然后使用***@include***，并仅使用全路径名的文件名部分即可。当然，你也可以将库文件放在不止一个目录下；更复杂的工作环境是这样的，你用多个目录来组织需要包含的文件。

由于可以通过多个 -f选项来指定个文件，***@include***这样的机制并不严格需要。但是，***@include***关键字可以帮你构建自包含的gawk 程序，这样可以避免写复杂，多余的命令行。特别是，***@include***在写CGI 脚本来运行网络页面时非常有用。

如在，***在第 43 页***，所提到的，当前的目录总是包含在文件AWKPATH 搜索路径的第一个位置，这也同样应用于***@include***机制。

## 2.8 在你的程序装入动态扩展

这一节描述的也是gawk 的特性。

***@load***关键字，可以用来读取外部的awk 扩展（存储为系统共享库）。这个机制可以让你使用编译过的代码来实现非常高的执行性能，并且/或者使用使用一些awk 语言所不支持的能力。AWKLIBPATH 变量来用进行扩展的搜索。使用***@load***机制完全与使用命令行选项 -l 等同。

如果扩展一开始没有在 AWKLIBPATH 中找到，会通过给文件名附加上默认动态库扩展名再搜索一次。例如，在GNU/Linux 系统中，这个后缀名是‘.so’：

**$ gawk ’@load "ordchr"; BEGIN {print chr(65)}’**

**-| A**

上面的代码与下面的例子等同：

**$ gawk -lordchr ’BEGIN {print chr(65)}’**

**-| A**

对于命令行的使用，-l 选项更加方便，但是***@load***用来在你的awk 源代码中嵌入必须的扩展也非常有用。

***,在第 349 页***，详细解释了如何（用C 或者C++）来写扩展，这些扩展可以用***@load***来装载，也可以用 -l 选项来装载。在这章里也述及了 ordchr 扩展。

## 2.9 已经取消的选项以及特性

本节所涉及的特性以及/或者命令行选项，来自之前的gawk 发布版本。这些特性，命令行选项要么已经不再可用，要么是虽然是还支持，但是已经标志为放弃状态(意味着它们在下个版本中将不再可用)。

进程相关的特殊文件/dev/pid， /dev/ppid， /dev/pgrpid， 与 /dev/user已经在 gawk 3.1 版本取消了，但是还可以使用。到 4.0 版本的时候，它们将不再被 gawk 识别。（而是使用PROCINFO，查看，***在第 158 页***。）

## 2.10 未记载的选项与特性

*使用源代码，祝好运！*

*—Obi-Wan*

本节特意留空。

## 2.11 总结

* 使用‘awk 'programe' files’或者‘awk -f program-file files’来运行 awk。
* 对所有版本的 awk，有三个标准的选项，即 -f， -F 与 -v。gawk 支持这几个选项，也支持其他许多的选项，以及 GNU 风格的长选项。
* 非选项的命令行参数通常会被当做文件名对待，除非它他有‘var=value’这样的形式，这种形式下，会在 awk 处理输入文件时执行变量赋值语句。
* 所有的非选项命令行参数，不包括程序文本，都会存放于ARGV数组中，调整 ARGC 与 ARGV 会影响 awk 处理输入的方式。
* 在命令行中，你可以使用一个单一的减号（‘-’）来指定标准输入。gawk 也可以让你使用特殊文件 /dev/stdin。
* gawk 会关注一系列的环境变量，AWKPATH， AWKLIBPATH与 POSIXLY\_CORRECT是最重要的几个。
* gawk 的退出状态向调用它的程序传递了相应的信息。在程序代码中使用退出语句来设置 awk 程序的退出状态。
* gawk 可以将其他的 awk 源代码包含在到你的程序中，这是通过@include语句以及/或者 -i 与 -f 命令行参数来实现的。
* 通过***@load***语句与/或者 -l 选项，你可以将使用 C 或者 C++ 编译的其他附加功能模块（扩展）引入你的程序。（这个高级特性会在*，****在第 297 页，*** 中进行详细说明。）

# 第三章 正则表达式

正则表达式，也称为‘regexp’，是描述字串集的方法。由于正则表达式是 awk 程序最基础的部分，它们的格式与使用有必要用单独的一章来进行解析。

在斜杠对（‘/’）中包括的正则表达式，在 awk 中称为模式，用以匹配那些属于这个字串集的输入记录。最简单的正则表达式就是一串字符序列，数字或者二者兼而有之。这样的正则表达式匹配任意具有这个字符序列的字串。所以正则表达式‘foo’匹配任意包含‘foo’的字串。这样，模式 /foo/ 匹配任意的输入记录，只要这个记录在任意位置包含这三个连续的字符序列‘foo’好可。其他类型的正则表达式可以让你指定更复杂的字串类。

在本章的开始的例子，都是比较简单的。当我们更深入地解释正则表达式如何工作时，就会写一些更复杂的例子。

## 3.1 如何使用正则表达式

正则表达式通过两个斜杠包含起来进行使用。然后正则表达式就会对记录的整个文本进行测试。（正常情况下，只需要按顺序匹配文本的一部份就能够成功测试匹配与否。）例如，下面只需要记录的任意位置出现‘li’串，就打印它的第二个域：

**$ awk’/li/ { print $2 }’ mail-list**

**-| 555-5553**

**-| 555-0542**

**-| 555-6699**

**-| 555-3430**

正则表达式也可以用于匹配表达式。这些表达式中你可以指定一个要匹配的串；而不需要整个的输入记录。有两个操作符来执行匹配，一个是‘~’，一个是‘!~’。这些表达式可以用作模式部分，也可以用在 if 语句，while 语句，for 语句，以及 do 语句。（查看 ，***在第 146 页***。）例如，下面的例子中，如果 exp（存有字串）匹配 regexp 则会为真：

**exp ~ /regexp/**

下面这个例子中，匹配，或者选择所有的输入记录中的第一个域中某个地方存在大字的字母‘J’：

**$ awk’$1 ~ /J/’ inventory-shipped**

**-| Jan 13 25 15 115**

**-| Jun 31 42 75 492**

**-| Jul 24 34 67 436**

**-| Jan 21 36 64 620**

下面执行一样的功能：

**awk’{ if ($1 ~ /J/) print }’ inventory-shipped**

下个例子中，如果表达式 exp （存有字串）不匹配 regexp 则为真：

**exp !~ /regexp/**

下面的例子中匹配或者选择所有的，第一个域不包含大写字母‘J’的记录：

**$ awk’$1 !~ /J/’ inventory-shipped**

**-| Feb 15 32 24 226**

**-| Mar 15 24 34 228**

**-| Apr 31 52 63 420**

**-| May 16 34 29 208**

**...**

当 regexp 被斜杠包含时，如 /foo/ 我们称 regexp 为常量，就类似于 5.27 是一个数值常量以及“foo”是一个字串常量一样。

## 3.2 转义序列

有些字符不能够原样地在字串常量（“foo”）或者正则表达式常量（/foo/）中。相反，它们需要放在转义序列中，在这个序列中，字符会用一个反斜杠开始。其中的一个用法就是在一个双引号字串常量中包含进一个双引号。由于光双引号会结束一个串，在字串中，你必须使用‘\"’来表示一个实际的双引号。例如：

**$ awk’BEGIN { print "He said \"hi!\" to her." }’**

**-| He said "hi!" to her.**

反斜杠字符自己本身也是一个不能以通常方式包含的字符；你必须输入‘\\’来将反斜杠包含进字串或者正则表达式中。因此，如果字串的内容为‘"’与‘\’，则必须写成 "\"\\"。

其他的转义字符是一些如 TAB 或者换行符这样的不可打印字符。没什么东西可以阻止你输入大多数的不可打印字符到字串常量或者正则表达式常量中，但那看起来奇丑无比。

下面的列表列出了所有的在 awk 中使用的转义的字符以及其对应的字符。除非另外提示，所有这些转义字符即可用于字串常量也用于正则表达式：

\\ 字符反斜杠‘\’。

\a “alert”字符，Ctrl+g，ASCII 代码 7（BEL，响铃）。（这通常是制造一些可以听得到的噪声。）

\b 退格，Ctrl + h，ASCII 代码8（BS）。

\f 换页，Ctrl + l，ASCII 代码 12（FF）。

\n 换行，Ctrl + j，ASCII 代码 10（LF）。

\r 回车，Ctrl + m，ASCII 代码13（CR）。

\t 水平跳格，Ctrl+i，ASCII 代码9（HT）。

\v 垂直跳格，Ctrl+k，ASCII 代码11（VT）。

\nnn 八进制值 nnn，nnn 表示 1 到 3位的‘0’到‘7’之间的数字。如，ASCII ESC 字符的代码是‘\033’。

\xhh... 十六进制值 hh，hh 表示十六进制数字（‘0’到‘9’，以及‘A’到‘F’或者‘a’到‘f’）序列。类似于 ISO C 中的结构，转换序列一直延续到第一个非十六进制的数字为止。(c.e.)但是，使用两个以上的十六进制的数字会产生未定义的结果。（‘\x’转义序列在 POSIX awk 中不被支持。）

注意：下一个 gawk 的主要版本会有改变，会使用到‘\x’后面跟两个最大的十六进制数字的情况。

\/ 字面斜杠字符（在正则表达式的常量中使用）。当你要在一个正则表达式中加入有反斜杠的表达式常量时使用（如***/.\*:\/home\/[[:alnum:]]+:.\*/***；***[[:alnum:]]’***符号会在，***在第 56 页***讨论。）由于正则表达式通过斜杠来分隔，所以如果斜杠是其中的一部分，你就需要进行转义，以通知 awk 继续处理正则表达式后面的部分。

\" 字面双引号（仅在字串常量中使用）。这个序列当你要在一个字串常量中包含双引号时使用（例如"He said \"hi!\" to her."）。由于双引号用来分隔字串，所以当在字串中有双引号时，你需要进行转义，以通知 awk 继续处理字串后面的部分。

在 gawk 的正则表达式中，有许多的以反斜杠开始两字符序列有关特殊的意义。查看，***在第 59 页***。

在正则表达式中，在任意字符前的反斜杠，不在前面的列表中，也没有在，***在第 59 页***，中，其下一个字符则会按字面的处理。就算它正常情况下是正则表达式的操作符。例如/a\+b/匹配3个字符的串***‘a+b’***。

为了完全可移植，不要使用任何没有在前面列表列出的字符或者在不是正则表达式操作符字符前使用反斜杠。

|  |
| --- |
| 常规字符前的反斜杠  如果你在字串常量中的某个未在前面的列表中列出的字符前使用反斜杠，POSIX awk 有意地使这种情况为未定义。有两个选择：  把反斜杠去掉  这正是 BWK awk 与 gawk 两者所做的。例如“a\qc”与“aqc”一样。（因为这是很容易引入，又容易错过的 Bug，gawk 会给予警告。）考虑***‘FS = "[ \t]+\|[ \t]+"’***使用坚线‘|’以及周边的空白作为域分隔符，这时，需要在字串中使用两个反斜杠。  保留反斜杠  有一些 awk 实现会这么做。在这种实现中，输入“a\qc”将与输入“a\\qc”相同。 |

总结一下：

* 对于字串常量与正则表达式常量，前面列表中的转义序列都是会首先处理。处理的时间非常靠前，一读完你的程序，就处理了。
* 常量正则表达式或者动态正则表达式（查看***，在第 58 页***），gawk 都会处理列在，***在第 59 页***，中的操作符。
* 在其他的符号前使用反斜杠，都表示把这个符号按字面处理。

|  |
| --- |
| 转义元字符  假设你使用八进制或者十六进制转义来提交正则表达式的元字符（查看，***见下方***），awk是否将这些字符做为字面字符还是一个正则表达式操作符？  由于历史原因，这样的字符是按照字面字符来处理的。然而，POSIX 标准却表示它们应该被当成真正的元字符，gawk 则是这么处理的。在兼容模式下（查看，***在第 35 页***），gawk 在正则表达式常中会将这些转义的八进制或者十六进制的字符按字面处理。因此 /a\52b/ 与 /a\\*b/ 等同。 |

## 3.3 正则表达式操作符

你可以在正则表达式中与特定字符进行组合，称为正则表达式操作符或者元字符，来给正常表达式加入更加强大的能力与多样性。

在前节，***在第 51 页***，中所描述的转义字符，在正则表达式中也是有效的。他它们通过反斜杠‘\’引入，然后在处理正则表达式第一步，被识别成或者转换成相应的真实字符。

下面是元字符列表。所有的那些未列在此的，非转义序列，就是代表他们自己。

\ 这个取消了某个字符的在匹配时的特殊含义，如‘\$’就是匹配字符‘$’。

^ 这里匹配字串的开头。‘^@chapter’匹配字串开头的‘@chapter’，如可以用来识别 Texinfo 中章的开头。‘^’也被称为锚定符，因为它就是将模式设定成匹配字串的开头。要了解‘^’并不是匹配内嵌在字串中行头（即‘\n’字符正右边的点）。下面的例子中条件表达式不为真：

if ("line1\nLINE 2" ~ /^L/) ...

$ 这个与‘^’类似，但是匹配的是串尾。如‘p$’匹配以‘p’结尾的串。‘$’是个锚定符，但是不匹配内嵌在字串中行尾（在‘\n’按行符正前的位置）。下面的例子的条件返回非真。

if ("line1\nLINE 2" ~ /1$/) ...

. (period) 匹配单个字符，包括换行符。如‘.P’匹配任意字符后面跟‘P’的任意字串。通过使用连接符，我们可以构造这样的正则表达式，如‘U.A’， 来匹配任意的三字符序列，其以‘U’开头，以‘A’结尾。在严格的 POSIX 模式（查看，***在第 35 页***）下，‘.’不匹配 nul（空）字符，该字符所有的位都被置成了0。否则，nul 就只是另一个字符。其他版本的 awk 有可能可以匹配 nul 字符。

[. . .] 这东西叫方括号表达式。[[22]](#footnote-23)它要匹配的是包含在方括号中的任一字符。例如，“[MVX]”匹配字符‘M’，‘V’或者‘X’中的一个。可以在方括号中放入什么字符的完整讨论在，***在第 56 页***。

[^. . .] 这个写方括号表达相反。在字符‘[’后面的第一个字符必须是‘^’。这里匹配除了方括号中列出的那些字符。例如，‘[^awk]’，匹配不是‘a’，‘w’，‘k’的字符。

| 这个是选择操作符，用来指定其他选择。‘|’在所有的正则表达式操作符中具有最低的优先级。例如，‘^P|[aeiiouy]’要么匹配 ^P，要么匹配[aeiouy]。就是说，匹配任意以P开头的字串，或者包含（在字串中的任一位置）英语元音字母的字串。

选择操作会在正则表达式的两边取中最大可能性的一边。

(. . .) 圆括号用来对正则表达式进行组合，如算法中的类似。他们可以将具有选择操作符‘|’的正则表达式组合起来。例如‘@(samp|code)\{[^}]+\}’能够匹配‘@code{foo}’与‘@samp{bar}’。（这里 Texinfo 的格式控制序列。‘+’在本列表中后面解释。）

\* 这个符号表达前面的正则表达式可以重复多次来找匹配。例如‘ph\*’中将‘\*’符号放在前面的‘h’后面，来查找一个‘p’后面跟跟随任意多个‘h’的匹配串。这也会匹配仅有‘p’，而后面没有‘h’的情况。在理解‘\*’如何工作上面，有两个微妙的地方要注意。第一，‘\*’只应用在其前面的单独正则表达式组件（例如，在‘ph\*’中，只应用要‘h’中）。要使‘\*’应用在一个更大的子表达式中，则使用括号：‘(ph)\*’匹配‘ph’，‘phph’，‘phphph’等等。第二，‘\*’会查找尽可能多的重复。如果要匹配的文本为‘phhhhhhhhhhhhhhooey’，则‘ph\*’会匹配所有的‘h’。

+ 本符号类似于‘\*’，只是其前面的表达式至少要匹配一次。这就是说，‘wh+y’会匹配‘why’，以及‘whhy’，但是不会匹配‘wy’，而‘wh\*y’则会匹配所有的三个。

? 这个符号类似于‘\*’，只是其前面的表达式只能匹配要么一次，要么不匹配。例如，‘fe?d’匹配‘fed’与‘fd’，而不是其他的什么。

{n}

{n，}

{n，m} 在花括号中与其中的一个或者两个数值表示一种区间表达式。如果只有一个数值在花括号中，则前面的表达式重复 n 次。如果有两个数值，并以逗号分隔，则前面的表达式重复 n 到 m 次。如果只有一个数值，后面跟一个逗号，则前面的表达式至少要重复 n 次。

**wh{3}y 匹配‘whhhy’， 但不匹配‘why’ or ‘whhhhy’。**

**wh{3，5}y 只匹配‘whhhy’， ‘whhhhy’， 或者‘whhhhhy’。**

**wh{2，}y 匹配‘whhy’， ‘whhhy’， 等等。**

区间表达式在 awk 上历来不受支持。后面成为 POSIX 标准的一部分，使得 awk 与 egrep 之间相互保持一致。

最开始时，由于旧的程序可能在正则表达式常量中使用‘{’与‘}’，gawk 因此在正则表达式中不匹配区间表达式。

但是从 4.0 开始，gawk 开始默认匹配区间表达式。这是因为与 POSIX 标准兼容，相比于与旧的程序兼容，前者更为重要。

对于在正则表达式中使用‘{’与‘}’的程序，好的实践是将他们用反斜杠进行转义。这样的话，你的正则表达式就是有效的，并在任何其他的版本中可用。[[23]](#footnote-24)

最后，当‘{’与‘}’出现在正则表达式常量中，但不是以一种可以被解释为区间表达式的方式（如 /q{a}/），那么他们就只表示自己。

在正则表达式中，‘\*’，‘+’以及‘?’操作符，包括花括号‘{’与‘}’，具有最高的优先级，然后是连接操作符，最后才是‘|’，在数学上，括号可以改变操作符的组合方式。

在 POSIX awk 中与 gawk 中，‘\*’，‘+’以及‘?’操作符，如果在正则表达式中，它们前面没有任何字符了，那么它就代表它们自己。如 /+/ 匹配一个字面加号。但是，很多其他的 awk 版本会把这样的用法视为语法错误。

如果 gawk 在兼容模式下，查看（，***在第 35 页***），区间表达式在正则表达式中是不可用的。

## 3.4 使用方括号表达式

如前面所提到的，方括号表达式匹配在括号中列出的任意字符。

在方括号表达式内，一个范围表达式由两个用连字符分隔的字符组成。它匹配以系统的本地字符集排序，并处于这两个字符间的符号。如‘[0-9]’与‘[0123456789]’等同。（查看，***在第 414 页***，对 POSIX 标准与gawk变化过程的解释，这主要是有一些历史兴趣所在。）

为了包括这些字符***‘\’***， ***‘]’***， ***‘-’***， 或者***‘^’***在方括号表达式中，将在它们之前使用‘\’，如：

**[d\]]**

会匹配‘d’或者‘]’，另外，如果你将‘]’就放在开方括号‘[’的后面，则闭方括号会以其字面字符对待并进行匹配。

对于在方括号中的‘\’表达式对待上，与其他的 awk 实现是兼容的，也是 POSIX 标准所要求的。在 awk 中的正则表达式，是 POSIX EREs（扩展正则表达式，Extended Regular Expressions）规范的一个超集。POSIX 的扩展正则表达式基于传统的 egrep 工具所接受的正则表达式。

字符类是在 POSIX 标准中引入的一个特性。字符类是一种特殊的符号，用其来表述具有特定属性的一类字符的列表，但是实际的字符由于国家的不同，以及/或者字符集的不同而不同。如，对于字母类的符号，在美国与法国之间就不一样。

正则表达式的字符类，只有在一个方括号表达式中的方括号内才会有效。字符类由‘[:’，一个代表类的关键字，以及‘:]’组成，表 3.1 列出来由 POSIX 标准定义的字符类。

类 含义

[:alnum:] 字母数字字符

[:alpha:] 字母字符

[:blank:] 空格与 TAB 字符

[:cntrl:] 控制字符

[:digit:] 数字字符

[:graph:] 可见可打印的字符（空格是可打印的，但是是不可见的，而‘a’则兼而有之）

[:lower:] 小写字母字符

[:print:] 可打印字符（那些不是控制字符的字符）

[:punct:] 标点字符（那些不是字母，数字，控制以及空格字符的字符）

[:space:] 空格字符（如空格，TAB，以及回档，等）

[:upper:] 字母的大写字符

[:xdigit:] 十六进制的数字的字符

表 3.1 POSIX 字符类

在 POSIX 标准前，如果你写了这样的表达式 /[A-Za-z0-9]/来匹配字母数字字符。如果你的字符集有另一套字母字符，这个就不会匹配它们。通过 POSIX 的字符类，你可写成这样/[[:alnum:]]/来匹配你的字符集中的字母与数字。

有一些工具的正则表达式提供了一种非标准的***‘[:ascii:]’***字符类，awk 则没有提供。但是，你可以这样***‘[\x00-\x7F]’***来模拟这种结构。

这会匹配数值上为 0 到 127 之间的，定义在 ASCII 字符集中的字符。利用其反的字符列表（***‘[^\x00-\x7F]’***）可以匹配任何个不在 ASCII 中的单字节字符。

有两个额外的序列可以出现在方括号表达式中，这些用来应用到非ASCII字符集中，这些字符集有的单符号（称为集合元素）可以由多个字符来表示。它们也可以有几个字符是等同的，以进行校对，排序等目的。（如在法语中，普通字母‘e’有一个重音字母‘`e’是等同的。）这些序列是：

集合符号

在‘[.’与‘.]’之间的，多字符集合元素。如，如果‘ch’是一个集合元素，这样***‘[[.ch.]]’***就是可以匹配这样的集合元素的正则表达式。而‘[ch]’则是匹配‘c’或者‘h’的正则表达式。

等同类

特定于字符集的用于命名等同字符列表。其名字由‘[=’与‘=]’包含。如名字 e 可以用来表示所有的 ***‘e’***， ***‘^e’***， ***‘′e’***， 与***‘`e’***。在这种情况下，***‘[[=e=]]’***就是用来匹配***‘e’***， ***‘^e’***， ***‘′e’***， 或者***‘`e’***的正则表达式。

这个特性在那些非英语的字符集中非常有用。

注意：gawk 当前所使用的一些正则表达式匹配库，只能识别 POSIX 字符类；它们无法识别集合符号或者等同类。

## 3.5 匹配多少？

考虑下面的例子：

**echo aaaabcd | awk’{ sub(/a+/， "<A>"); print }’**

在这个例子中，我们使用 sub() 函数来改变输入记录。（sub () 用第二个参数替换，任意匹配第一个参数的文本内容，查看，***在第 184 页***，这里，正则表达式 /a+/ 表示一个或者多个‘a’字符，并将其替换为文件‘<A>’。

输入中包括4个‘a’字符。awk（以及 POSIX）正则表达式总是匹配最左边最长的输入字符序列。因此，所有的4个‘a’字符都将被‘<A>’替换。

**$ echo aaaabcd | awk’{ sub(/a+/， "<A>"); print }’**

**-|<A>bcd**

对于简单的匹配/不匹配的测试，这就并不太重要，但是当执行文本匹配与替换时，如 match()， sub()， gsub() 与 gensub() 等函数所执行的，这个特点就非常重要了。理解这个原则对于基本正则表达式分隔的记录也非常重要（查看，***在第 63 页，***也可查看，***在第 71 页***）。

## 3.6 使用动态正则表达式

在‘~’或者‘!=’操作符的右手边，不一定需要一个正则表达式常量（例如用斜杠分隔这些的字串）。也可以是任意的表达式。在必要的情况下，表达式会被求值，或者转换为字串；这个字串即被用于当做正则表达式。以这种方式计算的表达式称为动态正则表达式或者计算型正则表达式。

**BEGIN { digits\_regexp = "[[:digit:]]+" }**

**$0 ~ digits\_regexp { print }**

上面的例子中，将 digits\_regexp 设置成描述一个或者多个数字的正则表达式，并测试输入记录是否匹配这个表达式。

当使用‘~’与‘!=’操作符时，要了解使用这样的用斜杠表式的正则表达式字串常量与用双引号括起来的正则表达式之间的不同。如果你准备使用字串常量，你就必须得理解字串实际上会被扫描两遍；第一次是 awk 读取你的程序的时候，第二次是当要执行与操作符左边的字串与右边的模式符进行匹配的时候。这对于所有求值为字串的表达式都是如此（如前例中的***digits\_regexp***），而并不止是字串常量才如此。

字串会被扫描两次会有什么不同呢？答案是与转义序列有关，尤其是反斜杠。为了将反斜杠加入到正则表达式中的字串，你需要输入两个反斜杠。

例如，/\\*/ 是字符‘\*’的正则表达式。只需要一个反斜杠。如果你要在字串在实现相同的事情，你就得输入“\\\*”。第一个反斜杠转义第二个反斜杠，因此实际的字串才包含两个字符，即‘\’与‘\*’。

假设你既使用正则表达式常量与字串常量来描述正则表达式，你该如何来选择使用？答案是正则表达式常量，有几个原因：

* 字串常量写起来更复杂，读起来也更难。使用正则表达式常量使你的程序少出错。不理解两种常量之间的差异常常是错误之源。
* 用正则表大式常量更加高效。awk 会注意到你提供正则表达式，会将它存成能够更加高效地进行模式匹配的形式。当使用字串常量，awk则首先需要将其转换成内部形式然后才执行模式匹配。
* 使用正则表达式常量是一种更好的形式，能够更容易看清你要进行的正则表达式匹配。

|  |
| --- |
| 在动态正则表达式中的方括号表达式中使用 \n  一些旧版本的 awk 不能够在动态正则表达式中的方括号表达式里使用换行符‘\n’。  **$ *awk* ’$0 ~ "[ \t\n]"’**  **error*awk*: newline in character class [**  **error ]...**  **error source line number 1**  **error context is**  **error $0 ~ "[ >>> \t\n]"<<<**  但是在正则表达式常量中却没有问题：  **$ *awk* ’$0 ~ /[ \t\n]/’**  **here is a sample line**  **-| here is a sample line**  **Ctrl-d**  gawk 没有这个问题，并且在实践中也很少出现，但是在后面的使用中，则值得注意一下。 |

## 3.7 特定于 gawk 的正则表达式操作符

GNU 软件在处理正则表达式时提供了许多额外的正则操作符。在本节中列出的操作符是 gawk 特有的，他们在其他的 awk 中是没有的。这些操作符多数是用来处理字（单词）匹配。对于我们的目的而言，一个单词就是一个或者多个字母、数字或者下划线（‘\_’）的连续序列：

\s 匹配任意的空白字符，你可看成是***‘[[:space:]]’***缩写。

\S 匹配任意非代白字会，可以看成是***‘[^[:space:]]’***的缩写。

\w 匹配任意组成单词的字符，也就是匹配任意的字母，数字或者下划线。可以看成为***‘[[:alnum:]\_]’***的缩写。

\W 匹配任意的非组成单词的字符，可以看成是***‘[^[:alnum:]\_]’***的缩写。

\< 匹配单词开始的空串，如/\<away/匹配‘away’，但是不匹配‘stowaway’。

\> 匹配单词后的空串，如 /stow\>/ 匹配‘stow’，但是不匹配‘stowaway’。

\y 匹配单词两边的空串（如单词的边界）。如***‘\yballs?\y’***匹配***‘ball’***或者***‘balls’，***作为单独的单词。

\B 匹配单词中任意两个组成单词字符间的空串，例如 /\Brat\B/匹配‘crate’，但是不匹配‘dirty rat’。‘\B’的反面实际就是‘\y’。

还有还两个操作符用在缓存中，在 Emacs 里，缓存很自然的就是一个 Emacs 缓存。在其他的 GNU 程序中，包括 gawk，可以认为整个要匹配的字串就是一个缓存。这两个操作符是：

\***‘*** 匹配缓存（字串）的开始处的空串。

***\’*** 匹配缓存（字串）的结束处的窗口。

由于‘^’与‘$’总是会匹配字串的开头与结尾，这个操作符没有给 awk 带来什么新的功能。提供这些操作符只是为了与其他的 GNU 软件保持兼容性。

在其他的 GNU 软件中，单词边界操作符是‘\b’。但是这个与 awk 语言定义‘\b’是一个退格有冲突，因此 gawk 使用了一个不同的字母。一个可选择的方法就是在 GNU 操作符中必须输入双反斜杠，但是这个用来起令人疑惑。当前的方法就是在使用‘\y’，这样 GNU 中的‘\b’在两个恶棍中就出现得少一些。

有几个命令行参数（查看，***在第 35 页***）可以控制 gawk 如何来解释正则表达式中的字符：

No options 在默认情况下，gawk 会提供所有的 POSIX 正则表达式的设施，以及前面所描述的那些正则表达式操作符。

--posix 只匹配 POSIX 正则表达式，GNU 操作符没有特殊含义（如‘\w’就是匹配字面的‘w’）。区间表达式也可用。

--traditional 匹配传统的 awk 正则表达式。GNU 操作符没有特殊含义，并且区间表达式也不可用。因为 BWK awk 支持它们，POSIX 的字符类（***‘[[:alnum:]]’***等）可用。由八进制与十六进制描述的转义字符序列会按字面处理，就算它们以正则表达式的元字符出现。

--re-interval 如果指定了--traditional，则可以在正则表达式中使用区间表达式。否则区间表达式是默认可用的。

## 3.8 匹配时的大小写

大小写在正则表达式中非常重要，无论是匹配正常字符（如非元字符）还是在方括号表达式中。因此‘w’只匹配小写的‘w’，而不是大写的‘W’。

不区分大小写的最简单的方法就是使用方括号表达式——如***‘[Ww]’***。但是要经常使用的时候就有点烦了，也使得正则表达式更难读。有可以有两个其他方法可以选择。

第一种方法，就是在要进行不区分大小写匹配的地方，用内建的函数 tolower() 或者 toupper() （当然还没讨论过；可以查看，***在第 184 页***）将其转换成同一种大小写。例如：

**tolower($1) ~ /foo/ { ... }**

将第一个域转换成小写，然后对其进行匹配。这个在 POSIX 兼容的 awk 中是可以使用的。

另外一个方法，只在 gawk 中使用，就是设置变量IGNORECASE为非零值（查看，***在第 155 页***）。当 IGNORECASE 为非零时，所有的正则表达式以字串操作都不区分大小写。

动态地改变 IGNORECASE 的值，就可以控制程序运行时的大小写处理方式。大小写默认情况下是区分的，因为 IGNORECASE （像其他大多数变量一样）被初始化为0：

**x = "aB"**

**if (x ~ /ab/) ... # this test will fail**

**IGNORECASE = 1**

**if (x ~ /ab/) ... # now it will succeed**

一般情况下，你不能通过 IGNORECASE 来使特定的规则是大小写不区分的，而其他的规则则是区分的。因为没有直接的方法来设置IGNORECASE以针对特定的规则。[[24]](#footnote-25)如果要这么做，要么使用方括号表达式或者使用 tolower() 方法，但是你可以对IGNORECASE所做的事情，就是动态地为所有规则开关大小写区分功能。

IGNORECASE也可以在命令行上设置，或者在 BEGIN 规则中设置（查看，***在第 41 页***，也可以参考，***在第 141 页***）。在命令行上设置IGNORECASE使得程序不用更改就可以进行不区分大小写的匹配。

在多字节字符的区域中，大小写字符是等同的，则是根据它们在字符集中的宽字符值来进行测试的。否则字符就是按照ISO-8859-1 (ISO Latin-1)字符集进行测试的。这个字符集是传统 128 ASCII 字符的超集，它也提供了许多的字符可用于欧洲语言。[[25]](#footnote-26)

如果 gawk 是在兼容模式中，则IGNORECASE的值不会有影响（查看，***在第 35 页***）。大小写在兼容模式下都是区分的。

## 3.9 总结

* 正则表达式描述了要被匹配的字串集。在 awk 中，正则表达式常量写成这样：/.../。
* 正则表达式常量可以在模式与条件表达式中单独使用，或者利用‘~’与‘!~’操作符使其成为匹配表达式的一部分。
* 转义序列可以让你输入不可打印字符，也可以让你将正则表达式的元字符作为字面字符在表达式中进行匹配。
* 正则表达式操作符可以提供组合，选择以及重复功能。
* 方括号表达式使你方便地指定一个字符集合，在正则表达式中的某个点进行匹配。在方括号表达式中。POSIX 字符类可以指定一组区域字符无关的字符。
* 正则表达式会匹配最左最长的字串。这对于你需要了解匹配的区域范围时是有意义的，像文件替换，以及记录分隔符是正则表达式这种情况。
* 匹配表达式可以使用动态的正则表达式（如字串值按照正则表达式对待）。
* gawk 的IGNORECASE变量可以让你控制正则表达式匹配时的大小区分情况。在其他版本的 awk 中，使用 tolower() 或者 toupper ()。

# 第四章 读取输入文件

在典型的 awk 程序中， awk 要么从标准输入（默认情况下，就是键盘，但是常常是其他程序命令的管道输入），要么从你在 awk 命令行上指定的文件中读取数据。如果你指定了输入文件， awk 会按顺序从中读取，并在下一个文件之前处理其中的数据。当前正在处理的文件可以从参定义的变量 FILENAME 中取得（查看，***在第 155 页***）。

输入单位称为记录，并利用你提供的规则一次处理一个记录，每个记录一行。每个记录会被自动力地拆分成几个块，称为域。这样程序就能够更好地处理记录的不同部分。

在极少情况下，你可能需要使用 getline 命令来读取数据。getline 非常有用，一是其可以从大量的文件中显示地输入，二是在读取文件时，这些文件不需要在命令地中指定（查看，***在第 83 页***）。

## 4.1 输入如何拆分为记录

awk 将输入拆分为记录与域，然后输入到你的程序中。同时，awk 记录从当前输入文件中已经读取的记录数。这个值存储在预定义的变量 FNR 中，在处理下一个文件时，它会被重置为 0。另一个预定义的变量称为 NR，来用记录从所有文件中已经读取输入记录的总数。它从0开始，但是不会自动重设为0。

### 4.1.1 标准 awk 的记录拆分

记录通过被称为记录分隔符的字符分隔。默认的分隔符是换行符（‘\n’）。这就是为什么记录默认都是一行一行的。要使用不同的字符来进行记录分隔，简单地给预定义的变量 RS 赋值即可。

与其他的变量类似，RS 的值可以在 awk 程序中通过操作符‘=’来改变（查看，***在第 122 页***）。新的记录分隔符可以用引号引起来，即指示为一个字串常量。一般地，这么做的正确时间是在程序执行的开始，并在记录未输入之前，因此就可以从第一条记录开始进行处理。要这么做，可以使用特殊的 BEGIN 模式（查看***，在第 141 页***）。例如：

**awk’BEGIN { RS = "u" }**

**{ print $0 }’ mail-list**

在读取任何输入之前，将 RS 的值改成‘u’。新的值是一个字串，它的第一个字符是字母‘u’；其结果就是，记录都被字母‘u’进行分隔。然后输入文件被读取，awk 程序的第二个规则（没有模式的动作）打印每一个记录。由于每条记录打印时，都添加了一个换行符在输入的末尾，所以，awk 程序就是将每个字母‘u’换成了换行符。下面是执行程序在使用 mail-list作为输入文件时的输出：

**$ awk ’BEGIN { RS = "u" }**

**>{ print $0 }’ mail-list**

**-| Amelia 555-5553 amelia.zodiac**

**-| sq**

**-| e@gmail.com F**

**-| Anthony 555-3412 anthony.assert**

**-| ro@hotmail.com A**

**-| Becky 555-7685 becky.algebrar**

**-| m@gmail.com A**

**-| Bill 555-1675 bill.drowning@hotmail.com A**

**-| Broderick 555-0542 broderick.aliq**

**-| otiens@yahoo.com R**

**-| Camilla 555-2912 camilla.inf**

**-| sar**

**-| m@skynet.be R**

**-| Fabi**

**-| s 555-1234 fabi**

**-| s.**

**-| ndevicesim**

**-| s@**

**-| cb.ed**

**-| F**

**-| J**

**-| lie 555-6699 j**

**-| lie.perscr**

**-| tabor@skeeve.com F**

**-| Martin 555-6480 martin.codicib**

**-| s@hotmail.com A**

**-| Sam**

**-| el 555-3430 sam**

**-| el.lanceolis@sh**

**-| .ed**

**-| A**

**-| Jean-Pa**

**-| l 555-2127 jeanpa**

**-| l.campanor**

**-| m@ny**

**-| .ed**

**-| R**

**-|**

注意名字为‘Bill’的那一项没有被拆分。在初始的数据文件中（查看***，在第 27 页***），那一行是这样的：

**Bill 555-1675 bill.drowning@hotmail.com A**

里面不包含‘u’，因此没有理由要分隔记录，不像其他的部分，它们每个都有一个或者多个‘u’出现。实际上，这个记录会被当前一个记录的一部分；输出时分隔他们的换行符是原来数据文件中的换行符，而不是 awk 后来打印记录时加上的。

另一个方法就是利用变量赋值特性，在命令行上改变记录分隔符（查看，***在第 41 页***）：

**awk’{ print $0 }’ RS="u" mail-list**

这里将在处理 mail-list 之前将 RS 设置成‘u’。

利用类似于‘u’这样的字母字符作为记录的分隔符有可能产生奇怪的结果。使用特别一点的字符，如‘/’更可能在大多数例子中产生正确的结果，但是这也不是打包票的事情。终极方法就是：要了解你的数据。

使用正则表达式作为记录的分隔符，当 gawk 完全与 POSIX 兼容时，有一个特殊的情况会出现（查看，***在第 35 页***）。下面（极端情况）的管程会打印令人惊讶的‘1’：

**$ echo | gawk --posix ’BEGIN { RS = "a" } ; { print NF }’**

**-| 1**

这里有一个域，只包含一个换行符。内建变量 NF 的值就是当前记录的域的数量。（在正常情况下，gawk 会将换行符当做空格处理，会打印***‘0’***作为结果。awk 大多数其他版本都是这么表现的。）

当输入文件结束后，就会结束当前的输入记录，就算文件的最后一个字符不在 RS 里。

空串（即没有任何字符的串）对于 RS 的值来说，有着特殊的含义。即表示记录之间是通过一个或者多个空行来分隔的，而不是其他的什么字符。查看，***在第 81 页***，来获取更多的细节。

如果你在awk程序中中途改变 RS 的值，新的值将用来做后续记录的分隔符，但是当前的记录会继续按原来的方式处理。

当记录尾已经确定好了，gawk 设置变量 RT 的值为输入文件中匹配 RS 的文本。

### 4.1.2 使用 gawk 分割记录

当使用 gawk 时，RS 的值并不局限于单个字符的字串，它也可以是任何的正则表达式（查看 ***, 在第 50 页***）。(c.e.)一般来说，每个记录都在下个匹配正则表达式的文件串地方结束；而下一个记录则开始于该匹配文本的地方。这个一般的原则在通常的情况下是么工作的，此时 RS 只包括一个换行符：记录结束于下一个匹配串的开始处（输入中的下一个换行符），然后下一个记录就开始于刚才匹配串的后面（即下一行的第一个字符处）。由于换行符匹配 RS，所以其并不是任何记录的一部份。

当 RS 只是单一的一个字符时，RT包含相同的单一字符。但是，当RS 是一个正则表达式时，RT包含匹配正则表达式的实际的输入文本。

如果输入文件结束时也没有文本匹配 RS，gawk 将 RT 设置成空串。

下面的例子解释了这两个特性。例子中将 RS 设置成一个正则表达式，表达式要么匹配换行符，要么是一个或者多个大写字母的序列，其前导并且/或者其末尾带有可选的空白符。

**$ echo record 1 AAAA record 2 BBBB record 3 |**

**> gawk ’BEGIN { RS = "\n|( \*[[:upper:]]+ \*)" }**

**> { print "Record ="， $0，"and RT = [" RT "]" }’**

**-|Record = record 1 and RT = [ AAAA ]**

**-|Record = record 2 and RT = [ BBBB ]**

**-|Record = record 3 and RT = [**

**-|]**

方括号勾勒出 RT 的内容，让你可看到了前面与后面的空白。RT 的终值是一个换行符。可以查看 ，***在第 285 页***，来看看更有用的以正则表达式作为 RS 与 RT 的例子。

如果你将 RS 设置在可以有可选结束文本的正则表达式，如***‘RS ="abc(XYZ)?"’***， 由于实现上的限制，gawk 有可能匹配正则表达式的前部分，而不会匹配后面的可选部分，尤其是可选的结尾匹配很长的情况。gawk 会试图避免这个问题，但是目前，并不保证绝不会发生这个情况（即不匹配结尾可选部分———译者）。

提示：要记住，在 awk 中，‘^’与‘$’锚定元字符匹配字串的开始与结束，而不是行的开始与结束。这样的结果就是，如果有这样的表达式‘RS = "^[[:upper:]]"’可以只匹配文件的开头。这是因为 gawk 将输入文件看作是一个包含换行符的长串。因此最好的避免在 RS 的值中含有锚定符。

将 RS 设置成一个正则表达式，以及 RT 变量是 gawk 的扩展；它们在兼容模式下是不可用的（查看 ）。在兼容模式下，只有 RS 值的第一个字符会用来确定记录的结束。

|  |
| --- |
| RS = "\0" 是不可移植的  有那么几次，你可能想将整个文件作为一个单独的记录。唯一能够实现这个功能的就是将 RS 的值设置成不会在输入文件中出现的字符。一般来说，程序比较难做到在任意的输入文件中都是正确的。  对于文本文件来说，你可能认为 nul 字符，即字符的所有位都是0，可以用在这种情况下：  **BEGIN { RS = "\0" } # whole file becomes one record?**  gawk 实际上接受这样的输入，并且使用 nul 字符作为记录的分隔符。这对于特定的特殊文件是可以的，例如在 GNU/Linux 系统上的文件 /proc/environ ，实际上 nul 字符实际上是记录分隔符。可是，对于大多数的 awk 实现来说，这是不可移植的。  大多数的其他 awk 实现[[26]](#footnote-27)内部将字串存成 C 风格的字串。而 C 风格的字串使用 nul 做为字串的结束符，效果就是***‘RS ="\0"’***与***‘RS = ""’***是一样的。  新近的 mawk 的版本也可以使用 nul 字符作为记录的分隔符。但是，这是个特殊情况：mawk 不允许在字串中嵌入 nul 字符。（这可能在 mawk 的后续版本中会改变。）  查看，***在第 231 页***，来查看一些有趣的方法来读取整个文件。如果你使用 gawk，查看，***在第 399 页***，来选择其他的选项。 |

## 4.2 检查域

当 awk 读取输入记录时，记录是自动地被 awk 工具分析或者分隔成称为域的块，默认情况下，域是通过空白进行分隔的，像一行中的单词一样。

awk 中的空白符表示一个或者多个空格、TABs，或者换行符所组成的串；[[27]](#footnote-28)其他的字符如回档符，垂直跳格等）被其他语言认为是空白符的这些符号都不被 awk 识别为空白符。

域的用处是为了让你可以很方便地指定这些记录的块。你不必一定要使用他们，你可操作整个记录，但是域可以使得 awk 程序变得非常强大。

你使用美元符号（‘$’）来指定awk程序中的域，符号后面跟的是域的编号。因此，$1 表示是第一个域，$2 是第二个等等。（不像 Unix Shell 中，域的编号只被限制用单个数字，例如，$127 是记录中的第 127 个域。假设下面的行是输入行：

**This seems like a pretty nice example.**

这里第一个域，或者 $1 是‘This’，第二个域，或者说 $2 是‘seems’等等。注意最后一个域，即 $7 是‘example.’。因为在‘e’与‘.’之间没有空格，因此点号被认为是第七个域的一部份。

NF 是预定义变量，用来记录当前记录的域数量。awk 在每次读取记录的时候，会自动更新 NF 的值。无论记录中有多少个域，其最后一个域都可以用 $NF 来表示。因此在上面的例子中，$NF 与 $7 相同，即‘example.’。如果你指定域超过了最后一个域（如上面的只有7个域的例子中指定 $8），你会得到一个空串（如果用作数值操作中，则你会得到0。）

$0 看起来像用来表示第0个域，但这是个特别情况，它代表的是整个输入记录。当你不使用单个的域的时候，你可以使用这个变量。这里有更多的例子：

**$ awk ’$1 ~ /li/ { print $0 }’ mail-list**

**-| Amelia 555-5553 amelia.zodiacusque@gmail.com F**

**-| Julie 555-6699 julie.perscrutabor@skeeve.com F**

在这个例子中，会打印文件 mail-list 中的每一条记录，其第1个域中包含‘li’字串。

作为比较，下面的例子中，会在整个记录中查找‘li’字串，然后打印匹配的记录中的第一个与最后一个域：

**$ awk ’/li/ { print $1， $NF }’ mail-list**

**-| Amelia F**

**-| Broderick R**

**-| Julie F**

**-| Samuel A**

## 4.3 非常数域编号

域编号不一定要是一个常数。awk 程序中的任意表达式都可以用在‘$’的后面来指定一个域。表达式的值指定了一个域编号。如果果是一个字串，而不是一个数值，则将会被转换成一个数值。看下面例子：

**awk’{ print $NR }’**

回忆一下，NR 是到目前为止所读取的记录数：当在第一个记录是是为1，第二时为2等等。在这个例子中，会打印第一个记录的第一个域，第二个记录的第二个域等等。对于20个记录，第20个域的值会被打印；最可能的情况是，记录少于20个域，因此这会打印一个空行。这里有另外一个例子，来用另一个表达式来计算域域编号：

**awk’{ print $(2\*2) }’ mail-list**

awk 会对‘（2\*2）’表达式进行求值，并用其值作为域的编号，并打印该域。‘\*’代表乘法操作，会在‘$’之前完成；当在域编号表达式中有二元操作符[[28]](#footnote-29)存在时，这都是必要的。这个例子中，打印 mail-list 文件中每一行中关系的类型（第4个域）。（所有的 awk 操作符都按照优先级降序列出在，***在第 135 页***）

如果域编号最终计算为0，你就会得到整个记录。因此，***‘$(2-2)’***与 $0 的值是一样的。负的域编号是无效的；这么做一般会导致程序的结束。（POSIX 标准没有定义在使用负的域编号是会产生什么情况。gawk 会注意到这样的使用方式，并结束整个程序。其他的 awk 实现处理方式可能不同。）

如在，***在第 67 页***，中所提到的，awk 将当前记录的域数目存储在内建的变量 NF 中（查看，***在第 155 页***）。因此， $NF 表达式具有一个特别的特性——就是会直接对 NF 求值，并将这个值作为域编号。

## 4.4 更改域内容

awk 所看到的域内容，是可以在 awk 程序中更改的，这会更改 awk 能看到当前输入记录的结果。（这个输入实际上还没有被处理；awk 不会修改输入文件。）考虑下面的例子以及它的输出：

**$ awk’{ nboxes = $3 ; $3 = $3 - 10**

**> print nboxes， $3 }’ inventory-shipped**

**-| 25 15**

**-| 32 22**

**-| 24 14**

**...**

上面的程序，首先保存第三个域的原始的值到变量 nboxes 中。‘-’号表示相减，因此程序重新对第三个域，即 $3 进行赋值为第三个域的原始值减10：‘$3 - 10’。（查看，***在第 119 页***）然后打印第三个域的旧值与新值（库房中有人出错了，导致在盘点红盒子时导致了不一致。）

要使这个能够工作，则 $3 的文件必须是一个数值；而字符串也必须转换成数值以进行算法计算。而减法所得的结果也会转化成字串，并成为第三个域的值。查看，***在第 117 页***。

当某个域的值被更改了后（如 awk 所见的），输入记录的文本就会被重新计算，以将新的域包含在旧的域所在的位置。换句话说，$0 会变更以反映变化的域。因此，这个程序会打印输入文件的复本，同时，每行中的第二个域会减去10。

**$ awk ’{ $2 = $2 - 10; print $0 }’ inventory-shipped**

**-| Jan 3 25 15 115**

**-| Feb 5 32 24 226**

**-| Mar 5 24 34 228**

**...**

给域范围之外的域赋值也是可以的，如：

**$ awk ’{ $6 = ($5 + $4 + $3 + $2)**

**> print $6 }’ inventory-shipped**

**-| 168**

**-| 297**

**-| 301**

**...**

我们在上面创建了 $6，它的值反映了域 $2，$3，$4 与 $5 的和。‘+’号代码加操作。对于文件 inventory-shipped 来说，$6 表示了特定月份中，已经发送的包裹的总数。

创建一个新的域，会改变 awk 的针对当前的输入记录内部复本，也就是 $0 的值。因此，如果你在添加了一个域后执行 ***‘print $0’***，导致输出结果也会包括新的域，同时相应数量的域分隔符会插入到其本身与前一个已经存在的域之间。

这样的重新计算即受 NF 的影响，也会影响 NF（域数量；查看***，在第 67 页***）。如果，NF 的值被设置成你创建的域的最大值。$0 的确切格式也同样受一个未讨论的特性的影响：输出域分隔符，OFS，该分隔符用来分隔域（查看，***在第 96 页***）。

但是，仅仅参考一个域范围外的域，并不会改变 $0 的值，也不会改变 NF 的值。参考范围之外的域仅会产生个空串，如：

**if ($(NF+1) != "")**

**print "can’t happen"**

**else**

**print "everything is normal"**

上面的例子应当打印***‘everything is normal’***， 因为 NF + 1 一定会超过域范围。（查看，***在第 146 页***，来获取关于 awk 的 if-else 语句的更多信息。查看，***在第 127 页***，来获取‘!=’操作符的更多信息。）

知道这样的事实也非常重要：对一个已经存在的域赋值会改变 $0 的值，但是不会改变 NF 的值，就算你给这个域赋给一个空串。例如：

**$ echo a b c d | awk’{ OFS = ":"; $2 = ""**

**> print $0; print NF }’**

**-| a::c:d**

**-| 4**

相应的域还是存在的，只是具有一个空值，在‘a’与‘c’之间用冒号进行了分隔。下面的例子显示，如果你创建一个新的域，会发生什么样的情况：

**$ echo a b c d | awk’{ OFS = ":"; $2 = ""; $6 = "new"**

**> print $0; print NF }’**

**-| a::c:d::new**

**-| 6**

中间的域，$5 会以空值创建（由第二对紧临的冒号指示），NF 的值则更新为6。

减少 NF 的值，将会将新值后面的域全部丢弃，并重新计算 $0 的值。下面是这样的例子：

**$ echo a b c d e f | awk ’{ print "NF ="， NF;**

**> NF = 3; print $0 }’**

**-| NF = 6**

**-| a b c**

注意：一些版本的 awk 当 NF 没有递减时则不会重建 $0。

最终，有机会在合适的时机强制 awk 利用当前的域与 OFS 值重新整个记录。为了实现这个目的，使用下面看起来没用的赋值语句：

**$1 = $1 # force record to be reconstituted**

**print $0 # or whatever else with $0**

这强制 awk 重建记录。如这里所示，添加相应的注释是有帮助的。

这是 $0 与记录之间关系的生动体现。任何对 $0 的赋值都会导致 awk 利用当前的 FS 值对记录进行重新分析。这也同样使用于内建的函数对于 $0 的更改，例如 sub() 以及 gsub()（查看，***在第 184 页***）。

|  |
| --- |
| 理解 $0  记住 $0 是整个记录这一点非常重要，知道他是通过输入获取的，这一点也同样重要。它包含任何的前导与后缀的空白符，也就是分隔域的空白符（或者其他的分隔字符）。  通过简单地设置 FS 或者 OFS 的值来改变域分隔符，并期望简单的***‘print’***或者***‘print $0’***打印的时新记录的值，是个常见的错误。  但是这不会有作用，因为对于更改记录本身什么也没有做。相反的，你必须强制记录要进行重建，典型的情况就是利用如***‘$1 = $1’***这样的语句来触发。 |

## 4.5 指定记录如何进行分隔

域分隔符要么是一个单个字符，或者是一个正则表达式，以控制 awk 将输入记录分隔成域。awk 扫描输入记录的字符序列以匹配分隔符；域则是匹配之间的文本。在下面的例子中，我们使用子弹符(***●***)个表示输出之间的空间。如果域分隔符是‘oo’然后下面的行：

**moo goo gai pan**

将被分成三个域：***‘m’***， ***‘●g’***， and ***‘●gai●pan’***。注意第二个与第三个域中的前导空间。域分隔符由预定义的变量 FS 表示。Shell 程序员要注意：awk 不使用由 POSIX 兼容 Shell(如果 Unix Bourne shell， sh， 或者 Bash)使用的名字 IFS。

FS 的值可以在 awk 的程序当中使用赋值操作符来进行改变，‘=’（查看，***在第 122 页***），一般来说，进行赋值的正确时机一般是在执行的开始，并在任何记录还没有被处理之前，因此第一条记录会通过正确的分隔符进行处理。为了实现这样的赋值，你可以使用 BEGIN 模式（查看，***在第 141 页***）。例如，下面的例子将 FS 的值设置成字串“，”：

**awk ’BEGIN { FS = "，" } ; { print $2 }’**

如果输入的行为：

**John Q. Smith， 29 Oak St.， Walamazoo， MI 42139**

这个 awk 程序会提取并打印字串***‘●29●Oak●St.’***。

有时，输入数据中包含有分隔符，因此没有按你所预期的方式对或者分隔。如，在我们刚才使用的例子中，人名中有可能有称谓或者后缀，如：

**John Q. Smith， LXIX， 29 Oak St.， Walamazoo， MI 42139**

同样的程序，会提取***‘●LXIX’***而不是***‘●29●Oak●St.’***。如果你预期其打印的是地址，你就会感觉到很奇怪。终极方法是要选择你的数据外观并仔细选择分隔符以避免这样的问题。（如果果数据的格式并不好处理，也可以先将它用另一个程序先输出来进行查看。）

### 4.5.1 标准的域分隔符空白符

通常的情况下，域是通过空白符序列进行分隔的（空格，TABs，或者换行符），而不是通过空格。一行中的两上连续字符不会分隔出一个空域。域分隔符 FS 的默认值是包含一个空格的字串，即“”。如果 awk 将这个值按照通常的方法解释，每个空格都会分隔出一个域出来。因此一行中的连续两个空格会在他们之间产生一个空域。没有产生这种情况，是因为 FS 值中的单个空格是一种特殊情况——其用来指定域分隔的默认方式。

如果 FS 是其他的任意单个字符，如“，”，则每一个这样的字符出现，都会分隔出两个域出来。两个连续的出现，就会分隔出一个空域。如果字会出现在开头或者结尾，也会产生一个空域。空格是唯一一个不按这种规则处理的字符。

### 4.5.2 使用正则表达式分隔域

前面的章节讨论了使用单个字符或者简单串作为 FS 值的情况。更通常的情况，FS 的值可以包含任意的正则表达式。在这种情况下，每一次匹配正则表达式就会分隔域。如下面的赋值：

**FS = "， \t"**

每一个输入行中，匹配逗号后面跟着一个与一个TAB字符的文本区，都将成为一个分隔符。

在一个不是太普通的正则表达式例子中，会尝试用单个空格来分隔域，与单个的逗号一样的方式。FS 可以被设置成“[ ]”（左方括号，空格，右方括号）。这个正则表达式只匹配单个空格而不是其它的（查看*，* ***在第 50 页***）。

***‘FS = " "’*** (单空格) 与***‘FS = "[ \t\n]+"’***（匹配一个或者多个空格，TABs 与换行符）之间存在一个重大的差异。对于 FS 的两个值，域都会被一系列的空格、TABs、与换行符分隔。但是，当 FS 的值是“”时，awk 首先将记录的前后空白符都去掉，然后才去决定域所在的区域。例如下面的管程会打印‘b’：

**$ echo ' a b c d ' | awk’{ print $2 }’**

**-| b**

但是这个管程会打印‘a’（注意一下在每个字母前后两额外空格）：

**$ echo ' a b c d ' | awk’BEGIN { FS = "[ \t\n]+" }**

**> { print $2 }’**

**-| a**

在这个例子中，第一个域是 null，或者称为空。

略去记录的前后空白符在 $0 是如果何来重新计算也会有作用。如，考察下面的管程：

**$ echo ' a b c d' | awk’{ print; $2 = $2; print }’**

**-| a b c d**

**-|a b c d**

第一个 print 语句打印出被读时的内容，并且前头的空白是保留了的。对于 $2 域的赋值会导致 $0 的重建，并且将 $1 到 $NF 的记录合并在一起，相互之间用 OFS 的值进行分隔（默认情况下也是空格）。因为前导的空白在查找 $1 时已经被忽略了，所以它不再是新的 $0 的一部份。最终，最后的一个 print 语句打印的是新的 $0。

使用正则表达式来分隔记录时，有一个额外的细节之处要了解。这在 POSIX 标准在没有确定，也在没其他地方说明，到底‘^’在域分隔时表示什么。是只匹配整个记录的开始？还是域分隔出的新串？因此不同的版本的 awk 对于这个问题的回答也是不一样的。你不要依赖于你的程序中的特定行为表现。

作为提示信息，BWK awk 允许‘^’只匹配记录的开头。gawk 也这么处理。如：

**$ echo 'xxAA xxBxx C' |**

**> gawk -F ’(^x+)|( +)’’{ for (i = 1; i <= NF; i++)**

**>printf "-->%s<--\n"， $i }’**

**-| --><--**

**-| -->AA<--**

**-| -->xxBxx<--**

**-| -->C<--**

### 4.5.3 各字符皆为分隔符

有那么几次，你可能想测试一下单独地每个字符作为分隔符的情况。这种情况下，在gawk中你只需要简单地将空串（“”）赋值给 FS 即可。（c.e.)在这个例子中，记录中的每一个单独的字符都将成为一个分隔符。如：

**$ echo a b | gawk ’BEGIN { FS = "" }**

**> {**

**> for (i = 1; i <= NF; i = i + 1)**

**> print "Field"， i， "is"， $i**

**> }’**

**-| Field 1 is a**

**-| Field 2 is**

**-| Field 3 is b**

传统上，FS等于“”的行为并不是确定的，在这个例子中，Unix awk 的大多数版本只是简单地将整个记录当做一个域。在兼容模式下（查看，***在第 35 页***），如果 FS 是一个空串，那么 gawk 也会按照上面的方式进行处理。

### 4.5.4 从命令行中设置 FS

FS 可以在命令行中通过 -F 选项进行设置，如：

**awk -F， 'program' input-files**

设置 FS 的值为‘，’，注意选项使用的是一个大写的‘F’，而不是小写的‘f’。后者‘-f’选项指定了包含 awk 程序的文件。

用来 -F 参数的值按照命令行上的赋值参数的方式一样处理。任何在域分隔字符中的特殊字符都必须进行适当地转义。如使用一个‘\’用作一个域分隔符，你需要这样来输入：

**# same as FS = "\\"**

**awk -F\\\\ ’...’ files ...**

由于 ‘\’ 要在 shell 中用引号引起来，awk 会看到 ‘-F\\’。之后 awk 会处理 ‘\\’ 转义字符，（查看，***在第 51 页***），最后，产生一个单独的 ‘\’ 用作域分隔符。

有一个特殊的例子，在兼容模式下（查看，***在第 35 页***），如果 -F 的参数是‘t’，则 FS 会被设置成 TAB 符号。如果你在 Shell 中输入‘-F\t’，没有进行任何转义，则‘\’会被删除掉，所以， awk 会了解到你是真的想用 TABs 来分隔记录，而不是‘t’。如果你确实是要用‘t’来分隔记录，则在命令行中使用***‘-v FS="t"’***或者***‘-F"[t]"’***。如果不是在兼容模式下，使用***‘-F ’***\t’’来指定 TABs 来做域分隔符。

下面的例子中，我们利用一个名为 edu.awk 的程序文件，其中包括了 /edu/ 模式，以及动作‘print $1’：

**/edu/ { print $1 }**

我们将 FS 的值设置成‘-’字符，并把程序运行在文件 mail-list 上。下面的命令打印出那些工作在大学或者在上大学的人的名字列表，以及他们电话号码的前三个数字：

**$ awk -F- -f edu.awk mail-list**

**-| Fabius 555**

**-| Samuel 555**

**-| Jean**

注意第三行输出，第三行内容在原文件是这样的：

**Jean-Paul 555-2127 jeanpaul.campanorum@nyu.edu R**

‘-’ 是人名的一部份，但是却被用来作了域分隔符，而不是按照预想的那样用电话号码中的 ‘-’ 来做分隔符。这说明了，你需要仔细地选择你的域分隔符。

也许，大多数使用单字符作为域分隔的情况下出现在处理 Unix 系统的密码文件。在许多的 Unix 系统中，每个用户在系统的密码文件中都有一个项，一行一用户。这些行中的信息是用冒号分隔的。第一个域是用户的登录名，第二项是用户已经加密过的密码，或者是影子密码（影子密码就是在第二个域中用 x 代替的密码。）密码文件的项看起来可能如下：

**arnold:x:2076:10:Arnold Robbins:/home/arnold:/bin/bash**

下面的程序搜索系统的密码文件并搜打印那些用户的全名没有存在的项：

**awk -F: ’$5 == ""’ /etc/passwd**

### 4.5.5 一行即一域

有时候，将整行当成一个域非常有用，这只要简单地将 FS 设置成‘\n’（换行符）[[29]](#footnote-30)即可，这也是可移植的。

**awk -F’\n’ 'program' files ...**

当你这么做时，$1 与 $0 一样。

|  |
| --- |
| 改变 FS 不会影响域  根据 POSIX 标准，awk 的行为起来就就像记录在被读取的时候就已经分隔了。这意思就是说，如果你在记录读取后再更改 FS 的值，域的值（如他们如何被分割）应该反映的是旧的 FS 的情况，而不是新的。  但是，许多的旧的awk 实现不是这么处理的。相反，这些实现是当要使用域时，才开始进行分割。域使用当前的 FS 值来进行分割，这个行为很验证进行识别。下面的例子解释了两种方法的差异：  **sed 1q /etc/passwd | *awk*’{ FS = ":" ; print $1 }’**  一般都打印：  **root**  这是在一个不正确的 awk 实现中的行为，而 gawk 则会打印文件的第一行的整行，如：  **root:x:0:0:Root:/:**  （sed [[30]](#footnote-31)命令打印文件 /etc/passwd 的第一行。） |

### 4.5.6 域分割总结

当你给 FS 赋值为一个字串常量的时候，要记住这个非常重要，其与 awk 正常的字符处理是不一样的。例如：Unix awk 与 gawk，这样的赋值***‘FS = "\.."’***会将“..”赋值给 FS（反斜杠会被去掉）。这个赋值创建了一个正则表达式表示“域由两个字符进行分割。”如果你确实想用一个点号（.）后面跟一个字符来分割域，使用***‘FS = "\\.."’***。

下面是基于 FS 的值的（‘==’表示等于‘’）域分割的总结：

FS == " " 域由连续的空格进行分割。记录的前后的空白会被忽略，这是默认分割方式。

FS == 任意单个字符

域由单个字符进行分割。多个连续的字符会分割出空域，头、尾的出现也一样。字符甚至可以是正则表达式的元字符，并且也不需要进行转义。

FS == ***regexp***

域由匹配‘regexp’的正则表达式字符串进行分割。前后匹配会分割出空字符。

FS == "" 记录中的每个字符都是分隔符。（这是一个通用扩展；但是没有由 POSIX 标准定义。）

|  |
| --- |
| FS 与 IGNORECASE  当域分割符 FS 的值是正则表达式时，IGNORECASE 变量会影响到分割。当 FS 是一个单独的字符时，则没有影响，即使字符是一个字母。因此下面的代码：  **FS = "c"**  **IGNORECASE = 1**  **$0 = "aCa"**  **print $1**  输出为‘cCa’。如果你确实想用字母字符来分割域而忽略大小写。就使用正则表达式，并由它来为你处理这样的问题（如，***‘FS = "[c]"’***）。在这个例子，IGNORECASE 会产生效果。 |

## 4.6 读取固定宽度数据

本单讨论 gawk 一个高级特性。如果你是 awk 新手，你可以第一次阅读时跳过。

gawk 提供了处理固定宽度域的功能，而不需要特别的域分割符。例如，这种数据可能由旧的 Fortran 程序处理数据时出现，或者是程序的输出不参与将其输出作为输入的程序。

作为后者的一个例子就是表，它的每一列通过利用可变的数字或者空格来对齐，空的域则全是空格。很明显的，awk 的一般域分割是基于 FS 的值，不会在这样的例子中工作得好。尽管一个可移植的 awk 程序可以使用一系列的 substr() 调用来处理 $0（查看），在处理有很多域，并且数据量巨大时，这是一很蹩脚也低效的方法。

将输入记录分割成固定宽度的域，通过指定一个包含空格分割的数字到内建变量FIELDWIDTHS来实现。每一个数字表示了域的宽度，包含了域之间的列。如果你想忽略域之间的列，则可以指定分离的域宽度，则后面的处理会忽略掉它。指定一个负的域宽是一个致命错误。下面的数据是 Unix 工具 w 的输出。这个用来演示FIELDWIDTHS的使用非常有用：

**10:06pm up 1 days， 14:04， 23 users**

**User tty login idle JCPU PCPU what**

**hzuo ttyV0 8:58pm 9 5 vi p24.tex**

**hzang ttyV3 6:37pm 50 -csh**

**eklye ttyV5 9:53pm 7 1 em thes.tex**

**dportein ttyV6 8:17pm 1:47 -csh**

**gierd ttyD3 10:00pm 1 elm**

**dave ttyD4 9:47pm 4 4 w**

**brent ttyp0 26Jun91 4:46 26:46 4:41 bash**

**dave ttyq4 26Jun91 15days 46 46 wnewmail**

下面的程序使用上面的输入，将空闲时间转换为秒数，并打印输出头两个域并计算空闲时间：

**BEGIN { FIELDWIDTHS = "9 6 10 6 7 7 35" }**

**NR > 2 {**

**idle = $4**

**sub(/^ +/， ""， idle) # strip leading spaces**

**if (idle == "")**

**idle = 0**

**if (idle ~ /:/) {**

**split(idle， t， ":")**

**idle = t[1] \* 60 + t[2]**

**}**

**if (idle ~ /days/)**

**idle \*= 24 \* 60 \* 60**

**print $1， $2， idle**

**}**

提示：前厕所程序使用了一些 awk 我们目前还没有讨论的特性。

执行上面的程序，产生了如下输出：

**hzuo ttyV0 0**

**hzang ttyV3 50**

**eklye ttyV5 0**

**dportein ttyV6 107**

**gierd ttyD3 1**

**dave ttyD4 0**

**brent ttyp0 286**

**dave ttyq4 1296000**

另一个固定宽度输入数据例子（可能更复杂）是来自于投票站。在美国的一些地区，投票者会将他们的选择在计算机卡上打上孔。然后处理这些卡，从而统计特定的候选者或者特定的问题。由于投票者可以不选择某些问题，因此卡上的某些列可能是空的。处理这类数据的 awk 程序可以使用 FIELDWIDTHS特性来简化数据的读取。（当然，让 gawk 选择在选举卡阅读器上的系统是另一个问题！）

向 FS 赋值会让 gawk 使用 FS 来进行域分割。使用 ***‘FS = FS’***来完成这样的功能，而不需要知道当前的 FS 值。为了了解当前正在使用的域分割策略，可以使用 PROCINFO["FS"] （查看，***在第 158 页***）。如果是常规的分割策略，则返回的值是“FS”，如果是固定宽度的分割策略则返回的是“FIELDWIDTHS”：

**if (PROCINFO["FS"] == "FS")**

**regular field splitting ...**

**else if (PROCINFO["FS"] == "FIELDWIDTHS")**

**fixed-width field splitting ...**

**else**

**content-based field splitting ... (*see next section*)**

当要写一个需要临时改变 FS 或者 FIELDWIDTHS 策略的函数时，这个功能则非常有用。这个函数读一些记录后，然后恢复初始配置（查看，***在第 242 页***，有这种函数的例子）。

## 4.7 以内容定义域

本节中，讨论一个 gawk 的高级特性。如果你是一 awk 新手，第一次阅读时，你可以路过这一节。

通常情况下，当使用 FS 策略时，gawk 将域定义为记录的一部份，它们出现在每个域分割符之间。换言之，FS 定义了域不是什么，而不是域是什么。但是，有那么一些时候，你确实需要按照域是什么来定义它们，而不是它们不是什么来定义。

最臭的例子就是所谓的“逗号分隔数据”（CSV）。许多的表单程序，都可以将他们的数据输出为文本文件人，这些记录都以换行符结束，同时通过逗号分隔。如果只有逗号来分割数据，这个就不是问题。问题是，当其中的某个域中有逗号时。在这样的状况下，大多数据的程序都将域用双引号引起来。[[31]](#footnote-32)所以，我们可能有如下这样的数据：

**Robbins，Arnold，"1234 A Pretty Street， NE"，MyTown，MyState，12345-6789，USA**

FPAT 变量为这样的问题提供了解决方案。FPAT 的值应该是一个正则表达式字串。这个表达式描述了每个域的内容。

在上面所列的 CSV 数据中，每个域要么是“一切都不是逗号，”或者“一个双引号，一切不是双引号，一个闭双引号。”如果写这样的一个正则表达式常量（查看*，* ***在第 50 页***），我们有这样的正则表达式/([^，]+)|("[^"]+")/。

**FPAT = "([^，]+)|(\"[^\"]+\")"**

是骡子是马，拉出来溜溜，这是来分析上面数据的简单程序：

**BEGIN {**

**FPAT = "([^，]+)|(\"[^\"]+\")"**

**}**

**{**

**print "NF = "， NF**

**for (i = 1; i <= NF; i++) {**

**printf("$%d = <%s>\n"， i， $i)**

**}**

**}**

当运行时，我们可以得到下面的结果：

**$ gawk -f simple-csv.awk addresses.csv**

**NF = 7**

**$1 = <Robbins>**

**$2 = <Arnold>**

**$3 = <"1234 A Pretty Street， NE">**

**$4 = <MyTown>**

**$5 = <MyState>**

**$6 = <12345-6789>**

**$7 = <USA>**

注意含在 $3 里的逗号。

这种处理 CSV 数据类型的直接提升是在双引号出现时，会自动移除它们，如像这样的：

**if (substr($i， 1， 1) == "\"") {**

**len = length($i)**

**$i = substr($i， 2， len - 2) # Get text within the two quotes**

**}**

与 FS 一起，IGNORECAASE 变量会影响 FPAT 类型的域分割策略（查看，***在第 156 页***）。

给 FPAT 赋值，会覆盖 FS 与 FIELDWIDTHS 策略。与 FIELDWIDTHS 中的 PROCINFO["FS"] 的值类似，如果采用的是基于内容的域分割策略，其值将是“FPAT”。

提示：有一些程序输出的 CSV 数据会包含换行符在双引号之间。gawk 没办法处理这个问题。就算是符号正规的 CSV 规范的数据，也无能为力。FPAT 机制为大多数情形提供了一种优雅的解决方法，gawk 的开发者对此也非常满意。

正如所写的，FPAT 所使用的正则表达式，要示每个域包含至少一个字符。直接更改（更改第一个‘+’为‘\*’）允许域可为空：

**FPAT = "([^，]\*)|(\"[^\"]+\")"**

最后，patsplit() 函数也实现相同的功能，用来对常规的字符进行分割（查看，***在第 184 页***）。

总结一下，gawk 提供三种独立的方法来将输入记录分割成记录。这些机制通过三个变量来实现，即FS， FIELDWIDTHS， 或者 FPAT，谁最后被赋值，则采用相应的对应的机制。

## 4.8 多行记录

在一些数据中，单行无法方便地包含所有的信息在一个项中。在这样的情况下，你可以使用多行记录。实现这个方式的第一步是选择你的数据格式。

其中一个技巧是使用一个不常用的字符或者字串来分割数据。例如，你可以使用换页符（在 awk 中写成‘\f’，如C）来分割他们，使得每个记录是文件中的一个页。接着，将变量 RS 的值设置成‘\f’（只包含一个换页符的字串）。任意其他的字符也是可以用的，只要它们不是记录数据的一部分。

另一个技巧是用一个空行来分隔记录，然后通过一个特殊的值，即一个空串赋值给 RS，以提示记录是通过一个或者多个空行来进行分隔的。当 RS 被设置成空串，每条记录就是结束于第一个空行。下一个记录直到后续遇到的第一个非空行开始。不管在行中出现多少空行，它们都实际分隔一个记录。（空行必须是完全空的，包含空格的也不算。）

你可以通过将串“\n\n+”赋值给 RS，可以取得与‘RS=""’一样的效果。这个正则表达式匹配记录结尾一个到多个换行符。另外，正则表达式总是匹配最长的可能序列（查看***在第 57 页***）。因此，下一个记录开始于后续的第一个非空行，不管一行中出现多少的空行，他们都会认为是一个记录分隔符。

但是，‘RS=""’与‘RS="\n\n+"’两者之间有一个重要的差别。第一个情形下，输入数据的前导空行会被忽略，并且如果文件的最后一条记录后面没有额外的一个空行，最后的换行符会被从记录中删除。但是在第二种情况下，这个特殊处理过程不会执行。

现在，输入已经被分割成记录，第二步是将记录分割成域。一种方法是用通常的方式是将记录分割成域。在这种特殊情况下，这是一种默认情况。当 RS 被设置成了空串，FS 被设置成单个字符，域分隔符总是换行符。同时也包含由 FS 产生的任意分割结果。[[32]](#footnote-33)

这个特殊的例外的原始动机可能是为了提供在默认场景下有用的行为（例如 FS 等于“”时）。当你确实不想用换行符来分隔域时，这个特性可能就会有问题，因为没有方法可以避免它。但是，你可以通过 split() 函数来手工分解记录来修复这个问题（查看，***在第 184 页***）。如果你有一个单字符的域分隔符，你可以用另外一种方式来解决这个特性带来的问题，只需要将 FS 设置成这个单字符的正则表达式。例如，如果域分隔符是一个百号，要使用‘FS=“[%]”’，而不要使用‘FS=“%”’.

flash.jpg对域进行分割的另外一个方法是将每个域放在一个行中：为了实现这个功能，只需要将变量 FS 设置成字串“\n”。（这个单字符分隔符匹配一个换行符。）一个实际的以这种方式进行组织的数据文件可能就是邮件列表，在这个列表中，空行用来分隔项目。考虑一个名字为 addresses 的邮件列表，看起来像下面这样：

**Jane Doe**

**123 Main Street**

**Anywhere， SE 12345-6789**

**John Smith**

**456 Tree-lined Avenue**

**Smallville， MW 98765-4321**

**...**

下面则是一个处理该文件的简单的程序：

**# addrs.awk --- simple mailing list program**

**# Records are separated by blank lines.**

**# Each line is one field.**

**BEGIN { RS = "" ; FS = "\n" }**

**{**

**print "Name is:"， $1**

**print "Address is:"， $2**

**print "City and State are:"， $3**

**print ""**

**}**

执行后，产生如下的输出：

**$ awk -f addrs.awk addresses**

**-| Name is: Jane Doe**

**-| Address is: 123 Main Street**

**-| City and State are: Anywhere， SE 12345-6789**

**-|**

**-| Name is: John Smith**

**-| Address is: 456 Tree-lined Avenue**

**-| City and State are: Smallville， MW 98765-4321**

**-|**

**...**

查看，***在第 277 页***，查看一个更实际的程序来处理邮件列表。下面的列表总结了记录基于 RS 的值的记录分割方式：

RS == "\n" 记录通过换来符来进行分割（“\n”）。在效果上，数据文件中的每一行都是一个记录，包括空行。这是默认行为。

RS == 任意单独字符

记录通过字符来进行分割。多个连续的字符会分割出空记录。

RS == "" 记录通过一连串的空行进行分割。当 FS 是一个单独的字符时，则换行符总会表现为域分隔符，不管 FS 有什么样的值。文件的头尾的换行符会被忽略掉。

RS == ***regexp*** 记录通过匹配 regexp 正则表达式的字符串来进行分割。文件头尾的空行会分割出空的记录。（这是 gawk 的扩展；在 POSIX 中没有对这种情形进行定义。）

如果不是在兼容模式下（查看，***在第 35 页***），gawk 设置 RT 的值为输入数据中匹配由 RS 指定的值的文本。但是，如果输入文件结束输入而没有匹配 RS，则 gawk 会将 RT 的值设置成空串。

## 4.9 用 getline 输入数据

到目前为止，我们都是从 awk 的主要输入流中获取数据，要么是标准输入（通常是你的键盘，有时是其他程序的输出）或者通过命令行指定的文件。awk 语言有一个特殊的内建命令 getline 可以用来显式地在你的控制下读取输入。

getline 命令有几种不同的使用方式，并且初学者最好不要使用。下面的 getline 使用例子以及相应的解释包括了一些目前还没有涉及的知识。因此，当你阅读了第一、二部分剩下的部分时，再回过头来学习 getline，则你会更了解 awk 是如何进行工作的。

getline 命令查看到一个记录时则返回 1，如果遇到文件结束时则返回 0。如果在读取数据时发生了错误，如文件不能被打开，则 getline 返回 -1。在这个情况下，gawk 将变量 ERRNO 的值设置成所遇到的的错误的描述字串。

在下面的例子中，command 代表 shell 命令的字串值。

提示：当 --sandbox 指定时（查看，***在第 35 页***），从文件，管道或者并程[[33]](#footnote-34)中读取行数据是被禁止的。

### 4.9.1 不带参数调用 getline

getline 命令可以不带参数来从当前文件中读取输入。在这种情形下，就是读取下一个输入记录并将其分割为域。当你已经完成当前记录的处理，但是对于下一个记录想先做一些处理的情况下非常有用。例如：

**# Remove text between /\* and \*/， inclusive**

**{**

**if ((i = index($0， "/\*")) != 0) {**

**out = substr($0， 1， i - 1) # leading part of the string**

**rest = substr($0， i + 2) # ... \*/ ...**

**j = index(rest， "\*/") # is \*/ in trailing part?**

**if (j > 0) {**

**rest = substr(rest， j + 2) # remove comment**

**} else {**

**while (j == 0) {**

**# get more text**

**if (getline<= 0) {**

**print("unexpected EOF or error:"， ERRNO) >"/dev/stderr"**

**exit**

**}**

**# build up the line using string concatenation**

**rest = rest $0**

**j = index(rest， "\*/") # is \*/ in trailing part?**

**if (j != 0) {**

**rest = substr(rest， j + 2)**

**break**

**}**

**}**

**}**

**# build up the output line using string concatenation**

**$0 = out rest**

**}**

**print $0**

**}**

这个 awk 程序从输入中删除 C 风格的注释（‘/\* ... \*/’）。其中用到的一些特性我们还没有讲到，包括字串合并（查看，***在第 121 页***），以及 index()， substr () 内建函数等（查看，***在第 184 页***）。通过替换‘print $0’为其他的语句，你可以执行更为复杂的从输入中删除注释的处理，例如搜索正则表达式等。（这个程序有一个很微小的细节——就是如果一个注释在本行结束，同时又开启另一个注释。）这种使用 getline 的方式，会设置 $0 的 NF，NR，FNR，RT 变量。

提示：$0 的新值用于测试接所有接下来的模式。$0 在触发执行 getline 调用的规则的最初始就会丢失。另一方面，next 语句读取一个新的记录，并且立即从程序中的第一个规则开始处理它。查看，***在第 153 页***。

### 4.9.2 将 getline 赋值给变量

你也可以使用‘getline var’将从 awk 的输入的下一个记录赋值给一个变量。这没有做任何针对数据的处理。例如，假设下一行是一个注释或者一个特殊字串，你想读取它的同时，不触发任何规则。这种形式的 getline 调用可以让你读取行并且将其存在一个变量中，因此，主循环中***read-a-line-and-check-each-rule（读取一行并检查每个规则）***的 awk 循环就看不到这个记录了。下面的例子交换输入中的两行：

**{**

**if ((getline tmp) > 0) {**

**print tmp**

**print $0**

**} else**

**print $0**

**}**

例子中使用下面的输入：

**wan**

**tew**

**free**

**phore**

并产生如下的结果：

**tew**

**wan**

**phore**

**free**

这么使用 getline 命令仅设置变量 NR，FNR 以及 RT（当然还有变量 var）。记录不会被分割成域，因此域的值（包括 $0）以及 NF 的值都不会改变。

### 4.9.3 使用 getline 读取文件

使用 ‘getline < file’ 来从 file 中读取记录。在这里，file 是一个求值为字串的表达式，并用其指定文件名。‘<file’ 称为重定向，因为它将输入定向为另一个不同的位置。例如，下面的程序当它遇到从当前输入文件输入的记录中的第一个域的值为10时，则从文件 secondary.input 中读取输入记录：

**{**

**if ($1 == 10) {**

**getline<"secondary.input"**

**print**

**} else**

**print**

**}**

由于主输入流没有被使用，变量 NR 以及 FNR 的值是不会改变的。但是，它所读取的记录会被正常地分割成域，因此 $0 以及其他域的值是会改变的，因此导致 NF 与 RT 的值是会改变的。

根据 POSIX 标准，‘getline < expression’如果 expression 中包含未括起来的除‘$’的操作符时，则会产生歧义。例如***‘getline< dir "/" file’***就是有歧义的，因为连接操作符（还未讨论到，查看，***在第 121 页***）是没有被括起来的。如果你的程序要能够移植到所有的 awk 实现中，你应该这么写***‘getline< (dir "/" file)’***。

### 4.9.4 从文件中 getline 到变量

使用***‘getline var < file’***来从文件 file 中读取输入，并且将其存放在变量 var 中。如前所示，文件是一个求值为字串的变量，结果中包括要被读取的文件。

在这个版本的 getline 中，没有一个预定义的变量会改变，记录也不会被分割成域。唯一改变的就是 var[[34]](#footnote-35)变量。例如，下面的程序复制所有的输入文件到输出，除了如果记录内容为***‘@include filename’***。这样的记录会被文件 filename 替换：

**{**

**if (NF == 2 && $1 == "@include") {**

**while ((getline line < $2) > 0)**

**print line**

**close($2)**

**} else**

**print**

**}**

注意这里的另外的输入文件的名字是如何构建到程序中的；它是直接从数据中来的，通过***@include***行的第2个域来指定的。

close() 函数的调用是为了保证如果***@include***在输入文件中出现两次，文件的内容也会被指定两次。查看，***在第 108 页***。

这个程序的不足就是不能处理内嵌***@include***的情况（例如，***@include***语句包含在被包含的文件中），如一个真正的宏处理器所执行的功能。在，***在第 286 页***， 查看处理内嵌***@include***语句的程序。

### 4.9.5 从管道中使用 getline

*全知全能太遥远，还不是关注细节更有用。*

*—Brian Kernighan*

命令的输入也可以被通过使用‘command|getline’来以管道输出给 getline。在这种情形中，command 是一个 Shell 命令，同时，它的输出会被以管道形式输出给 awk 来作为输入。这种形式的 getline 从管道中一次读取一个记录。如，下面的程序复制它的输入到输出中，排除那些以‘***@exclude***’开头的行，这些行会被替换为将行的剩下的内容当做Shell命令执行时产生的输出：

**{**

**if ($1 == "@execute") {**

**tmp = substr($0， 10) # Remove "@execute"**

**while ((tmp | getline) > 0)**

**print**

**close(tmp)**

**} else**

**print**

**}**

close() 函数保证两个相同的‘***@execute***’行，命令会执行相应次数。假设有下面的输入：

**foo**

**bar**

**baz**

**@execute who**

**bletch**

程序可能产生如下的输出：

**foo**

**bar**

**baz**

**arnold ttyv0 Jul 13 14:22**

**miriam ttyp0 Jul 13 14:23 (murphy:0)**

**bill ttyp1 Jul 13 14:23 (murphy:0)**

**bletch**

注意这个程序会执行命令 who 然后打印出结果。（如果你测试这个程序，你可能会得到不同的结果，这依赖于你所登入的系统。）

这样的 getline 形式会将记录分割成域，并设置 NF 的值，并重新计算 $0 的值。NR 与 FNR 的值不会改变，RT 的值则会被设置。

根据 POSIX 标准，***‘expression | getline’***表达式中，如果 expression 包含了未被括起来的除 $ 之外的操作符时，则会是有歧义的。例如，***‘"echo ""date" | getline’***由于连接操作符没有被括起来，因此这个语句是有歧义的。如果乐想你的程序可以有移植到所有的 awk 实现里，你应该这么写：***‘("echo ""date") | getline’***。

提示：不幸的是，gawk 在对待像这样的结构‘"echo ""date" | getline’时是不一致的。大多数的版本，包括目前的版本，都将其看成‘("echo ""date") | getline’。（这也是 BWK awk 的行为）有一些版本则会将其看成‘"echo " ("date" | getline)’。（这是 mawk 的行为）简单地说。显式地使用括号，就不用担心这些问题。

### 4.9.6 从管道中getline到变量

当使用***‘command | getline var’***这种形式，即 command 命令的输出通过管道发送给 getline，并将其存储在变量 var 中。例如，下面的程序利用 date 工具，读取当前的日期与时间到变量 current\_time 中，然后打印它：

**BEGIN {**

**"date" | getline current\_time**

**close("date")**

**print "Report printed on " current\_time**

**}**

在这个版本的 getline 中，没有任何预定义的变量会改变，记录也不会分割成域，但是 RT 会被设定。

### 4.9.7 从并程中使用 getline

从管道中 getline 读取输入是单向操作。通过***‘command | getline’***启动的命令 command 只会将数据发送到你的 awk 程序。

有时候，你可能会想发送数据到另一个程序用于处理，并将结果返回来。gawk 可以启动一个并程，同时允许双向通信。这是通过‘|&’操作符来实现的。典型的情况中，你首先将数据写到并程中，然后将结果读取回来，如下面所示：

**print "some query" |&"db\_server"**

**"db\_server" |&*getline***

上面的例子中将请求发送到 db\_server 中然后读取结果。

NR 与 FNR 的值不会变更，因为主输入流没有被使用。但是，记录会按正常方式分割成域，因此会改变 $0 的值，其他域的值，以及 NF 跟 RT 的值。

并程是一个高级特性。在这里讨论是因为这跟 getline 有关。查看 ，***在第 304 页***，这里并程会更讨论得更深入一些。

### 4.9.8 使用并程 getline 到变量

当你使用***‘command |&getline var’***，并程 command 的输出会通过一个双向管斗管道发送到 getline 中，并保存在变量 var 中。

在这个版本的 getline 中，没有任何预定义的变量会改变，记录也不会分割成域。唯一变化的是变量 var，但是 RT 的值会被设定。

### 4.9.9 回顾 *getline*

这下面有几点关于 getline 的内容需要留意：

* 当 getline 改变了 $0 与 NF 的值时，awk 不会自动跳到程序的开始并测试新的记录与模式。但是新的记录会匹配任意后后续规则。
* 一些非常老的 awk 实现限制了 awk 程序可以打开的管线的数量就是1个。在 gawk 中，没有这样的限制。你可以打开你想要的在操作系统限制下的管线（以及并程）数量。
* 如果你在 BEGIN 规则中使用 getline 而没有重定向符，则会出现一个有趣的副作用。由于一个未重定向的 getline 会从命令行中的数据文件中读取数据，第一个 getline 命令会导致 awk 设置 FILENAME 的值。正常情况下。FILENAME 不会在 BEGIN 规则内有值，由于你还没有开始处理命令行数据文件。（查看，***在第 141 页***与，***在第 158 页***。）
* 在 getline 中使用 FILENAME（***‘getline< FILENAME’***）有可能导致困惑。awk 利用当前文件打开一个独立的输入流。但是不通过使用变量，$0 与 NF 的值也还会改变。如果你这么做，有可能是偶然，你应该重新考虑你到底想要什么结果。
* ，***在第 90 页***，提供了 getline 不同变体及在相应情况下受影响的变量的总结表。也值得注意，那些变体不使用重定向，如果导致了 awk 重新读取一个新的输入文件，也可以导致 FILENAME 的更新。
* 如果正在被赋值的变量是一个带副效应的表达式，则不同的 awk 版本会有不同的表现。这依赖于怎样碰到“end-of-file”。这是一个例子，感谢Duncan Moore：

**BEGIN {**

**system("echo 1 > f")**

**while ((getline a[++c] <"f") > 0) { }**

**print c**

**}**

这里，副效应是‘++c’。如果文件结束时，c 是否是在 a 的元素被赋值之前进行递增么？

gawk 将 getline 按照函数调用来对待，并在尝试读取 f 之前对表达式***‘a[++c]’***进行求值。但是，在 awk 的某些版本中，当它知道有字串值要进行赋值时，才会对表达式求值。

### 4.9.10 getline 变体总结

表 4.1 总结了 getline 的8个变体，并列举了在每种变体下会被设置的预定义变量，以及变体是否是 gawk 的标准扩展。提示：对每个变体，gawk 都会设置预定义变量 RT。

|  |  |  |
| --- | --- | --- |
| Variant | Effect | awk / gawk |
| getline | Sets $0， NF， FNR， NR， and RT | awk |
| getline var | Sets var， FNR， NR， and RT | awk |
| getline< file | Sets $0， NF， and RT | awk |
| getline var < file | Sets var and RT | awk |
| command | getline | Sets $0， NF， and RT | awk |
| command | getline var | Sets var and RT | awk |
| command |&getline | Sets $0， NF， and RT | gawk |
| command |&getline var | Sets var and RT | gawk |

表4.1：getline 变体及其设置的变量

## 4.10 带超时的输入读取

本节讨论的特性是 gawk 所独有的。

在读取键盘上、管道、双向通信，包括TCP/IP 套接字中读取输入的时候，你可以指定一个为毫秒单位超时时间。这可以在每个输入，每个命令或者每个连接的基础上来指定。所有这些都可以指定 PROCINFO 数组中的某个元素的值来实现（查看，***在第 158 页***）：

**PROCINFO["input\_name"， "READ\_TIMEOUT"] = timeout in milliseconds**

如果设置了这样的时间，gawk 超时后，如果没有在指定的超时时间内读取到数据，则会返回一个失败状态。例如，TCP 客户端可以在指定时间后，决定是放弃接收从服务器返回的响应。

**Service = "/inet/tcp/0/localhost/daytime"**

**PROCINFO[Service， "READ\_TIMEOUT"] = 100**

**if ((Service |&getline) > 0)**

**print $0**

**else if (ERRNO != "")**

**print ERRNO**

下面则是从 user 那里[[35]](#footnote-36)进行交互式读写，并且不等待超过5秒：

**PROCINFO["/dev/stdin"， "READ\_TIMEOUT"] = 5000**

**while ((getline<"/dev/stdin") > 0)**

**print $0**

gawk 会在等待指定的超时时期后，如果没有数据输入，则会结束读操作，并返回失败，同时设置 ERRNO 的值为恰当的值。负值或者0的超时时间，与不指定超时时间一样。

超时时间也可以设置在在显式的循环中来读取键盘输入，通过读取这些输入来对模式进行匹配，如：

**$ gawk ’BEGIN { PROCINFO["-"， "READ\_TIMEOUT"] = 5000 }**

**> { print "You entered: " $0 }’**

**gawk**

**-| You entered: gawk**

在这个情形下，如果在5秒之内反应失败，会导致如下错误信息：

**error gawk: cmd. line:2: (FILENAME=- FNR=1) fatal: error reading input file *‘-’*: Connection timed out**

超时或者变更超时可以在任意时间，并在下次从输入设备中读取时产生效果。在下面的例子中，我们从1秒开始，并不断按1/10秒的时间递减，直到有输入到达：

**PROCINFO[Service， "READ\_TIMEOUT"] = 1000**

**while ((Service |&getline) > 0) {**

**print $0**

**PROCINFO[Service， "READ\_TIMEOUT"] -= 100**

**}**

提示：你不可以假设读操作在读取了第10个记录后会阻塞。有可能 gawk 第一次读取时就读取了多于一个的记录并缓存起来。因为这个原因，在上面的例子中，改变超时值不是特别有用。

如果 PROCINFO 元素没有设置，并且GAWK\_READ\_TIMEOUT环境变量存在，gawk 会使用这个变量的值来初始化超时值。唯一地使用环境变量来指定超时值有这样的一个缺陷，就是不能够按照每命令或者每个连接地进行控制。

gawk 会认为超时事伯是一个错误。就算后续对于输入设备的读取是成功的。这是一个限制，这也表示你不可以用这个来复用两个或者更多的输入源。

赋值一个超时值，可以避免读操作一直被阻塞。但是要记住，还有其他的方法来等待输入设置准备就绪。一个网络客户端在可以读取数据之前可能需要等待很长的时间来建立连接，或者要很长的时间来打开 FIFO 特殊文件以进行读取，这时会一直阻塞，直到其他的进程打开这个特殊文件并进行写。

## 4.11 命令行中的目录

根据 POSIX 标准，在 awk 命令行中指定的文件名必须是文本文件；如果不是则会是一个致命错误。大多数 awk 版本将命令行上指定目录看成是一个致命错误。

在默认情况下，gawk 在命令行上指定目录会产生一个告警，但是会忽略它。这样的知，可能很容易使用右 awk 程序中使用 Shell 通配符：

**$ gawk -f whizprog.awk \* Directories could kill this program**

如果指定了 --posix 或者 --traditional 选项，gawk 则会将在命令行上指定目录视为致命错误。

查看 ，***在第 397 页***，有一个方式来在 awk 程序将目录视为有用的数据。

## 4.12 总结

* 根据 RS 的值，输入会被分割成记录，可能如下：

|  |  |  |
| --- | --- | --- |
| Value of RS | Records are split on . . . | awk / gawk |
| Any single character | That character | awk |
| The empty string ("") | Runs of two or more newlines | awk |
| A regexp | Text that matches the regexp | gawk |

* FNR 表示已经从当前输入文件中读取了多少记录，NR则表则从所有的文件总读取了多少记录。
* gawk 设置 RT 的值为通过 RS 匹配的文本。
* 将输入分割成记录后，awk 会进一步将记录分割成不同的域，称为 $1，$2 等等。$0 是整个记录，NF 表示有多少个域。默认的分割域的方式是在空白符之间的字符。
* 域可以通过变量来进行参考，如 $NF。域也可以被赋值，这会导致 $0 再次被使用时，会被重新计算。对大于 NF 的值的域进行赋值可以创建新的域，并重建记录，通过 OFS 来分割域。增大 NF 的值也产生相同的结果。减少 NF 的值，会丢弃超过 NF 之外的域，并重建记录。
* 域分割比记录分割要复杂一些：

|  |  |  |
| --- | --- | --- |
| Field separator value | Fields are split . . . | ***awk*** / gawk |
| FS == "" | On runs of whitespace | ***awk*** |
| FS == any single character | On that character | ***awk*** |
| FS == ***regexp*** | On text matching the ***regexp*** | ***awk*** |
| FS == "" | Such that each individual character is a separate field | gawk |
| FIELDWIDTHS == list of columns | Based on character position | gawk |
| FPAT == ***regexp*** | On the text surrounding text matching the ***regexp*** | gawk |

* 使用‘FS="\n"’会使整个记录成为一个域（假设用换行符来分隔记录）。
* FS 可以通过命令行选项 -F 进行设置，也可以通过命令行中的变量赋值方式进行设置。
* Use PROCINFO["FS"] to see how fields are being split.
* 通过 PROCINFO["FS"] 来查看正在被用于进行域分割的方式。
* 使用 getline 及其不同的格式来从默认的输入流之外的文件，管管或者并程等地方读取记录。
* Use PROCINFO[file， "READ\_TIMEOUT"] to cause reads to time out for file.
* 使用PROCINFO[file， "READ\_TIMEOUT"]来对从文件读进行超时设置。
* 标准的 awk 中，在命令行中指定目录是一个致命错误；gawk 在非 POSIX 模式中会忽略它们。

## 4.13练习

1. 使用 FIELDWIDTHS 变量（查看，***在第 77 页***），写一个程序来读取选举数据，这些数据中每个记录代表一个投票者的选票。构造一个方式来定义列如何来与每个投票项进行关联，并打印总的票数，包括每个项的弃权票数。
2. 在，***在第 83 页***中，有一个程序来从输入中移除 C 风格的注释（/\*...\*/)。那个程序在一个行中结束一个注释，同时开启另一个注释时工作不正确。可以做一些很小的变更就可以来修复它，如何来做？

# 第五章 打印输出

大多数编程行为就是打印或者说是输出一些或者全部的输入。使用 print 语句进行简单的输出，利用 printf 语句进行格式化输出。print 语句并不仅限于何时计算哪个值要进行打印。但是有两个例外，你无法指定它们如何来打印它们——多少列，是否使用幂方式等等。（对于这些例外，查看，***在第 96 页***，以及，***在第 97 页***）对于规范的打印，你需要 printf 语句（查看，***在第 97 页***）。

除了基本的与格式化的打印，本章还会讨论一 I/O 重定向到文件与管道的相关问题，介绍一些 gawk 内部使用的特殊的文件名，还讨论 close() 这个内建函数。

## 5.1 print 语句

利用 print 进行简单的，标准化的输出。只需要指定要打印的字串以及数字，并用逗号分隔它们。它们输出时用简单的空格进行分隔，并在最后附加上换行符。语句看起来如下：

**print item1， item2， ...**

整个的项目列表可以用括号括起来，但是可选的。如果项目表达式使用了‘>’这样的关系操作符，则括号是必须的；否则会产生令人困惑的输出重定向（查看，***在第 103 页***）。

要打印的项，可以是常量字串，数值，当前记录的域（如$1），变量，以及任意的 awk 表达式。数值会被转换成字串，然后再打印。

简单的 print 语句，同时没有带任何打印项，等同于‘print $0’：即打印整个的当前记录。要打印一个空行，则使用‘print ""’。要打字固定的文本片，使用字串常量，如“Don't Panic”这样的项。如果你忘记使用双引号字符，则文件会被当成 awk 表达式，这样有可能会产生一个错误。要记住，在任意两个要打印的项目之间空格是要打印的。

注意 print 语句是一个语句，而不是一个表达式——你不能使用在模式-动作语句中的模式部分。

## 5.2 print 语句例子

每一个 print 语句至少会产生一行输出。但是不限于只输出一行。如是项目中的值是一个字串，并包含一个换行符，则换行符会与剩下的串一起输出。一个单一的 print 语句可以按这种方式首先任意行数。下面是一个打印内嵌了换行符串的例子：

**$ awk’BEGIN { print "line one\nline two\nline three" }’**

**-| line one**

**-| line two**

**-| line three**

下一个例子，会运行在文件 inventory-shipped 上，并打印每个输入记录的前两个域，它们之间有空格：

**$ awk’{ print $1， $2 }’ inventory-shipped**

**-|Jan 13**

**-|Feb 15**

**-|Mar 15**

**...**

常见的使用 print 语句的错误是省略了项目之间的逗号。这常会导致在输出中所有的项目挤在了一些，相互之间没有空格。原因在于在 awk 中，并列两个字串表示的是合并它们。这是相同的程序，但是没有逗号的情况：

**$ awk’{ print $1 $2 }’ inventory-shipped**

**-|Jan13**

**-|Feb15**

**-|Mar15**

**...**

对于不熟悉 inventory-shipped 文件的读者，上面的两个例子的输出没什么意义。开头的行可能更清楚。让我们在我们的月（$1）表以及已经发送的蓝包（$2）中添加一些头部信息：

**awk’BEGIN { print "Month Crates"**

**print "----- ------" }**

**{ print $1， $2 }’ inventory-shipped**

当执行时，程序打印如下：

**Month Crates**

**----- ------**

**Jan 13**

**Feb 15**

**Mar 15**

**...**

然而，唯一的问题是头部信息与表数据并没有对齐。我们可以在其中打印一些空格来修复这个问题：

**awk’BEGIN { print "Month Crates"**

**print "----- ------" }**

**{ print $1， ""， $2 }’ inventory-shipped**

如果要修复的列有很多，这样的修复方式会非常繁琐。计算两三个列是很简单的，但是超过这个数的时候，所耗的时间就很多了。这就是为什么要使用 printf 语句原因（查看 ，***在第 97 页***）；它的其中一个特别功能就是对齐数据列。

提示：你可以对 print 或者 printf 进行续行，只需要简单地在任意的逗号后面放置一个换行符即可。（查看，***在第 31 页***）。

## 5.3 输出分隔符

如前所述，print 语句包含了由逗号分隔的项目列表。在输出中，项目一般是用空格进行分隔。但是也不一定非要这样，空格只是默认情况。任意的字串都可以用来当成是输出分隔会，只需要设定预定义的变量 OFS。这个变量初始值就是串“”（一个空格）。

整个的 print 语句输出称为一个输出记录（output record）。每一个 print 语句输出一个输出记录，然后输出一个输出记录分隔符（或者 ORS）。ORS 的初始值为串“\n”（一个换行符）。因此，每一个 print 语句产生一个单独的行。

为了改变输出域以及记录的分隔方式，给变量 OFS 以及 ORS 赋值即可。通常做这件事的地方就是 BEGIN 规则里（查看），因此，在任意的输入被处理之前就可以就绪。也可以在命令行中的对其进行赋值，需要放在输入文件之前，或者使用 -v 命令行选项（查看）。下面的例子打印记录的第一与第二个域，并用分号分隔，并在每个换行符的后面增加一个空行：

**$ awk’BEGIN { OFS = ";"; ORS = "\n\n" }**

**> { print $1， $2 }’ mail-list**

**-|Amelia;555-5553**

**-|**

**-|Anthony;555-3412**

**-|**

**-|Becky;555-7685**

**-|**

**-|Bill;555-1675**

**-|**

**-|Broderick;555-0542**

**-|**

**-|Camilla;555-2912**

**-|**

**-|Fabius;555-1234**

**-|**

**-|Julie;555-6699**

**-|**

**-|Martin;555-6480**

**-|**

**-|Samuel;555-3430**

**-|**

**-|Jean-Paul;555-2127**

**-|**

如果 ORS 的值中不包含换行符，程序的输出就会挤在一行中。

## 5.4 控制 print 的数字输出

当用 print 语句打印数值时，awk 内部会将每一个数值转换成一个字符串，然后打印它。awk 使用 sprintf 函数来做这样的转换（查看，***在第 184 页***）。目前，只需要知道 sprintf 接受一个格式规范（字串）用以告诉如何来对数字或者字串如何进行格式化，并且数字有很多种方式来进行格式化。不同的格式规范会在，***在第 98 页***中讨论得更加详细。

预定义的变量 OFMT 包含了 print 时想将数值转换成字串，使用 sprintf 时的格式规范。OFMT 的默认的值是 “%.6g”。print 打印数字的格式可能通过给 OFMT 赋予不同的格式规范来改变，如下面的例子所示：

**$ awk’BEGIN {**

**> OFMT = "%.0f" # print numbers as integers (rounds)**

**> print 17.23， 17.54 }’**

**-| 17 18**

根据 POSIX 标准，如果 OFMT 包含的不是浮点转换规范的其他任意数值，会产生未定义的行为。

## 5.5 使用 printf 语句美化打印

为了比 print 提供更加精准地控制输出格式，可以使用 printf。printf 可以指定每个项目的宽度，以及一系列的数值输出格式（如使用的是什么基数，是否打印幂指数形式，是否打印符号，以及在小数点后要打印多个位等）。

### 5.5.1 printf 介绍

一个简单的 printf 语句是这样的：

**printf format， item1， item2， ...**

与 print 相似，整个的参数列表可以使用括号括起来。这里也是一样，如果项目表达式中包含了‘>’关系表达操作符，则括号是必须的，否则会导致一个输出重定向而使用困惑（查看*，****在第 103 页***）。

在 print 与 printf 之间的不同，就是 format 参数。这是一个会被当成字串识别的表达式，它指定了每个参数的输出格式。它称为格式字串。

格式字串与 ISO C 库中的函数 printf () 非常类似。大多的格式文本都输出来说都是字面的。在这些文本之间分散着格式指定符。每个项目一个。每个格式指定符都定义了下一个输出的项目在相应位置的格式。

printf 不会自动地在输出中附加一个换行符。它只输出格式串所定义的串。因此，如果需要换行符，你需要在格式中加入。输出分隔符变量 OFS 以及 ORS 在 printf 语句中没有作用。例如：

**$ awk’BEGIN {**

**> ORS = "\nOUCH!\n"; OFS = "+"**

**> msg = "Don\47t Panic!"**

**> printf "%s\n"， msg**

**> }’**

**-|Don’t Panic!**

例子中，无论是 ‘+’ 还是‘OUCH’都没有出现在输出信息中。

### 5.5.2 格式控制字母

格式指定符以字符‘%’开始，并以格式控制字母结束——整个格式指定符告诉 printf 语句如何来输出项目。格式字母指定了要打印的值的类型。格式指定符的其余部分则由可选的修饰符组成，这些修饰符控制如何来打印这些值，如域宽度。下面是格式控制字母：

%c 将数字打印成一个字符，所在‘printf “%c”，65’ 输出字母‘A’。如果是一个字串，则输出字串的第一个字符。

提示：POSIX标准说是打印字中的第一个字符。在多字节语言中，gawk 会尝试将字串的首字节转换成一个有效的宽字节，然后打印那个字符的多字节编码。相似地，当打印一个数值时，gawk 允许数值的值在一个宽字符所允许的值范围内。如果转换成多字节编码出错，gawk 会使用数值的低8位来打成字符打印。

其他版本的awk 一般严格限制为打印字串的第一个字节，或者数值的范围在一个字节的所表示的范围内（0-255）。

%d， %i 打印一个十进制整数。两个控制字母是等同的。（‘%i’ 规范主要是为了兼容ISO C。）

%e， %E 将数值打印成科学计算方式（幂指数），例如：

***printf “%4.3e\n”， 1950***

会打印‘1.950e+03’，带有4个有效位，其中3个在小数点的后面。（‘4.3’表示两个修饰符，会在下个小节中进行讨论。）‘%E’ 表示在输出中使用‘E’ 而不是‘e’ 。

%f 打印一个浮点数，例如：

***printf*** "%4.3f"， 1950

会打印‘1950.000’，带有4个有效位，3个在小数点后面。（‘4.3’表示两个修饰符，会在下个小节中进行讨论。）

在支持IEEE 754 浮点格式的系统上，代表负无穷会格式化为‘-inf’，或者‘-infinity’，正无穷则为‘inf’ 或者‘infinity’。对于“不是数值”这种特殊情况，则格式化为‘-nan’ 或者‘ nan’（查看，***在第 337 页***）。

%F 类似于‘%f’，只是无穷大，以及“不是数值” 会使用大写字母输出。‘%F’ 是一个ISO C 的一个扩展，不是所有的系统都支持。对于不支持的系统，gawk 会使用‘%f’ 来代替。

%g， %G 以科学方式或者浮点数方式打印浮数值，看哪个使用更少的字符，如果结果是以科学计数方式，‘%G’ 则使用‘E’ 而不是使用 ‘e’。

%o 打印一个无符号八进制整数（查看，***在第 113 页***）。

%s 打印一个字串。

%u 打印一个无符号整数。（使用这个格式要注意，因此在awk 中的所有数值都是浮点数，这个只是为了与ISO C 保持兼容。）

%x， %X 打印一个无符号的十六进制整数，‘%X’ 使用字母‘A’ 到‘F’ 而不是‘a’ 到‘f’（查看，***在第 113 页***）。

%% 打印‘%’。这个不会消耗后面的参数，并且忽略掉所有的修饰符。

提示：当使用整数的控制字母来打印其值超过 C 整数的最大的范围时，gawk 会切换到‘%g’ 这个格式指定符。如果在命令行中使用了 --lint 选项（查看），gawk 会生成一个告警。其他版本的awk 有可能打印一个无效的值，或者产生完全不同的其他结果。flash.jpg

### 5.5.3 printf 的修饰符

一个格式指示符功能包括一些修饰符以控制项目的值如何来进行打印，以及其需要多少的空间。修饰符处于‘%’与格式控制字母之间。我们使用子弹符“***●“***来表示输出中的空格。下面是可能的修饰符，以出现的顺序排序：

N$ 一个整数后面带一个‘$’ 是一个位置指定符。一般情况下，格式字串中的格式指定符是按顺序应用到后面的参数的。通过位置指定符，格式指定符则应用到指定的参数中，而不是原来本应该应用在列表中的参数。位置指定符从0开始计算。因此：

**printf "%s %s\n"， "don’t"， "panic"**

**printf "%2$s %1$s\n"， "panic"， "don’t"**

打印两次最有名的友好信息。

初看之下，这个特性没有什么特别的用处。事实上，这是gawk 的一个扩展，主要用于在运行时进行信息翻译。查看， ***在第 315 页***， 当中所描述的为什么以及如何来使用位置指定符。目前，我们可以先忽略它。

- (Minus) 一个减号 ‘-’，用在宽度指定符之前（看本列表之后），告诉对指定的宽度内进行左对齐。一般情况下，在指定的宽度内，参数是右对齐。因此：

**printf "%-4s"， "foo"**

打印***‘foo●’***。

space 对于数值转换，表示正值之前缀一个空格，或者负值之间带一个减号‘-’。

+ 加号，用在宽度指定符前（看本列表之后），告诉在数值转换上总是使用符号，就算数值格式是正数。‘+’ 号重载了空格指定符。

# 对于特定的控制字母使用“候选格式”。对于‘%o’ 则提供一个前导的0。对于‘%x’ 与‘%X’，则使用‘0x’ 或者‘0X’ 在非零的结果之前。对于‘%e’，‘%E’，‘%f’，‘%F’，结果中总是包含小数点。对于‘%g’ 与‘%G’，后面的零不会从结果中移除。

0 前导的 ‘0’ （零）是一个标志指示符，指定输出中使用0来填充而不是使用空格。这只用在数值格式中。这个标志也应用在如果宽度指定符中指定的宽度大于要打印的数值的宽度的情况。

’ 单引号或者撇号是ISO C 的POSIX 扩展。它表示的是浮点数值中的整数部分，或者整数的全部，应该包含有一个百分位分隔符。这个只在支持这样的格式的语言中有用。例如：

**$ cat thousands.*awk* *Show source program***

**-| BEGIN { *printf* "%’d\n"， 1234567 }**

**$ LC\_ALL=C gawk -f thousands.*awk***

**-| 1234567 *Results in "C" locale***

**$ LC\_ALL=en\_US.UTF-8 gawk -f thousands.*awk***

**-| 1，234，567 *Results in US English UTF locale***

要获取关于语言设置与国际化问题的更多信息，请查看，***在第 136 页***。

提示：‘'’标志是一个非常好的特性，但是使用起来会导致一些事情复杂起来：在命令行程序中用起来非常困难。要了解引号技巧的一些信息，请看 ，***在第 24 页***。

width 这是一个数值指定符，指定一个域所需要的最小宽度。在‘%’与格式控制字母之间插入一个数值，强制域的宽度扩展到指定的大小。默认的情况下会在左边的位置在空闲位置以空格填充。例如：

**printf "%4s"， "foo"**

会打印***‘●foo’***。

宽度的值是最小宽度，不是最大。如果值所需要的宽度更大，会扩展到必要的宽度。因此下面：

**printf "%4s"， "foobar"**

会打印***‘foobar’***。

如果在宽度的面前加一个减号（‘-’），则输出中的填充会在右边，而不是在左边。

.prec 一个点后面跟一个整数常量指定了打印时所使用的精度。其含义根据控制字母不同。

**%d， %i， %o， %u， %x， %X**

要打印的最小数字个数。

**%e， %E， %f， %F**

在小数点后面的数值个数。

**%g， %G**

有效数字位的最大个数。

**%s**

从要打印的字串中打印的最大字符数。

因此下面的代码：

**printf "%.4s"， "foobar"**

会打印‘foob’。

C 库中的 printf 的动态宽度与 prec 能力也被支持。不需要显式地在格式字串中指定一个宽度与/或者 prec 的值，它们也可以在参数列表中进行处理，如：

**w = 5**

**p = 3**

**s = "abcdefg"**

**printf "%\*.\*s\n"， w， p， s**

与下面的完全一致：

**s = "abcdefg"**

**printf "%5.3s\n"， s**

两个程序都输出***‘●●abc’***。awk 的期的版本不支持这样的能力。如果你必需使用这样的版本，你可以通过（字串）连接的方式来构建格式格式字串来模拟这个特性：

**w = 5**

**p = 3**

**s = "abcdefg"**

**printf "%" w "." p "s\n"， s**

在实践中这个代码不是很好读，但是确实可以工作。

C 程序员还可以在 printf 的格式字串中使用一些额外的修饰符（***‘h’***， ***‘j’***， ***‘l’***， ***‘L’***， ***‘t’***， 与***‘z’***）。这些在 awk 中都是无效的。大多数 awk 实现都是默默地忽略它们。如果 --lint 在命令行中指定了（查看），gawk 会产生告警。如果指定了 --posix ，则会产生一个致命错误。

### 5.5.4 *printf* 范例

下面是一个简单的如何来使用 printf 实现对齐表格的例子：

**awk’{ printf "%-10s %s\n"， $1， $2 }’ mail-list**

这个命令打印文件 mail-list 中的人名（$1）为10个字符的串，并且左对齐。同时也打印电话号码（$2）在同一行的后面。这会产生一个两行对齐的名字与电话的列表，如：

**$ awk’{ printf "%-10s %s\n"， $1， $2 }’ mail-list**

**-| Amelia 555-5553**

**-| Anthony 555-3412**

**-| Becky 555-7685**

**-| Bill 555-1675**

**-| Broderick 555-0542**

**-| Camilla 555-2912**

**-| Fabius 555-1234**

**-| Julie 555-6699**

**-| Martin 555-6480**

**-| Samuel 555-3430**

**-| Jean-Paul 555-2127**

在这例子中，电话号码要以字串的方式打印，因为号码是以横杠进行分隔的。如果将电话号码打印成数值，则产生的输出只有三个数字：‘555’。这就有点蛋疼了。

没有必要指定电话号码的长度，因为它们是一行中的最后一项。不需要在它们的后面填上空格。

如果在列的上面打印一些头部，看起来就更加漂亮了。这可以通过 BEGIN 规则来实现（查看，***在第 141 页***），这样的话，头部就只会在 awk 程序开始打印一次：

**awk’BEGIN { print "Name Number"**

**print "---- ------" }**

**{ printf "%-10s %s\n"， $1， $2 }’ mail-list**

在前面的例子中混合了 print 与 printf 语句。只使用 printf 也可以产生相同的结果：

**awk’BEGIN { printf "%-10s %s\n"， "Name"， "Number"**

**printf "%-10s %s\n"， "----"， "------" }**

**{ printf "%-10s %s\n"， $1， $2 }’ mail-list**

在打印头部时，使用与列一样的格式指定符，可以保证头部也列一样地对齐。

其中的格式指定串被使用了三个，因此可以将其存储在一个变量中，如：

**awk’BEGIN { format = "%-10s %s\n"**

**printf format， "Name"， "Number"**

**printf format， "----"， "------" }**

**{ printf format， $1， $2 }’ mail-list**

## 5.6 重定向*print* 与 *printf* 的输出

到目前，从 print 与 printf 的输出都打印在标准输出上，一般就是屏幕。print 与 printf 都可以将其输出定向到其他的地方，这就称为重定向。

提示：当在命令行中指定了 --sandbox （查看），将输出定向到文件，管道与并程会被禁用。

重定向出现在 print 与 printf 语句之后。awk 中的重定向如 Shell 脚本中的重定向一样，区别只是它们是写在 awk 程序中。

有4种重定向方式：定向到文件，定向附加到文件的末尾；定向到另一个命令的管道中，与定向到一个并程。我们将其展示在 print 语句中，但也可以应用在 printf 语句：

print items > output-file

这个类型的定向会将项目输出到一个名字为 output-file 的文件中。文件名 output-file 可以是任意表达式。它的值会被转换成字串并用作文件名（查看*，* ***在第 112 页***）。

当使用这种重定向时，output-file 在第一个输出写入前会将文件清空。后续对同一个文件的写不再进行内容擦除，只会附加。（这与你在 Shell 中使用重定向不同。）如果 output-file 不存在，则创建它。例如下面是一个 awk 程序，其会将人名列表写到名为***name-list***的文件中，并将电话号码写入到另一个名为 phone-list 的文件中：

**$ awk’{ print $2 >"phone-list"**

**> print $1 >"name-list" }’ mail-list**

**$ cat phone-list**

**-| 555-5553**

**-| 555-3412**

**...**

**$ cat name-list**

**-| Amelia**

**-| Anthony**

**...**

输出的文件中，每行包含一个名字或者一个电话号码。

print items >> output-file

这个类型的重定向将项目输出到一个已经存在的名为 output-file 的文件中。这个用法与单个的‘>’的重定向有不同时，output-file 文件中的旧内容（如果有）不会被清空。awk 的输出会附加到文件中。如果文件不存在，则创建它。

print items | command

通过管道可以将输出定向到一个另一个程序而不是文件。这个方式的重定向会向 command 打开一个管道，然后将项目的值通过为执行的 command 创建的管道写到咖一个程序中。

重定向参数 command 实际是是一个 awk 表达式。它的值会被转换成要执行的 Shell 命令的字串。例如，下面的代码会产生两个文件，一个是未排序的人名，一个按字母倒序对人名进行排列：

**awk’{ print $1 >"names.unsorted"**

**command = "sort -r > names.sorted"**

**print $1 | command }’ mail-list**

未排序的列表按照常规的方式进行重定向，而排序的列表则通过管道向 sort 工具输出。下个例子利用重定向将消息通过邮件列表系统邮寄出去。这个功能在一个周期执行的系统维护 awk 脚本中遇到问题时，非常有用：

**report = "mail bug-system"**

**print("Awk script failed:"， $0) | report**

**print("at record number"， FNR， "of"， FILENAME) | report**

**close(report)**

close() 函数在这里调用，是因为在将所有的输出写到管道后关闭它是一个好的编程实践。查看，***在第 108 页***，来获取更多的信息。

这个例子也说明了利用变量来表示文件或者命令——没有必要总是用字串常量来表示它们。使用变量一般来说是个好主意，因为（如果你要用相同的名字或者命令）awk 需要每次都要写相同的字串。

print items |& command

这个重定向将项目打印到 command 的输入中。这个方式与单个‘|’重定向的不同是，这里的 command 输出可以用 getline 进行读取。因此，command 是一个并程，即与 awk 一起执行，但是只作为 awk 程序的补充。

这个特性是 gawk 的扩展，在 POSIX 类型的 awk 是不可用的。查看，***在第 88 页***，来获取简单的说明，查看，***在第 304 页***，来获取更全面的讨论。

使用‘>’， ‘>>’，‘|’或者‘|&’ 请示系统打开一个文件、管道或者并程，只要你所指定的文件或者命令还没有被你的程序写入，又或者它已经在最后一次写入后关闭了。

使用‘>’ 重定向先打印到文件，然后用‘>>’ 为后续的输出是一个常见的错误：

**# clear the file**

**print "Don’t panic">"guide.txt"**

**...**

**# append**

**print "Avoid improbability generators">>"guide.txt"**

在Shell 中确实是这么重定向的。但是在awk 中，却没有必要。在这咱情况下，程序应该在所有的 print 语句上使用‘>’，因为输出文件只会打开一次。（在混合使用‘>’ 与‘>>’ 时，也是打开一次。但是，混用操作符是一种糟糕的编程习惯，会让你程序的读取困惑。）

如早前所述（查看，***在第 89 页***），许多的awk 实现限制了awk 程序可以打开的管线数量为一个！gawk则没有这个限制。gawk 可以打开的管线的数量只能所在的操作系统的限制。

|  |
| --- |
| 管道输出到sh  一种使用重定向的强大方式是构造一个命令行，然后将用管道输出到Shell，sh 中。例如，你有一堆的文件是从其他系统中取来的，并且文件名都以大写存储，而你想把他们全部改成小写的。下面的程序即简单也很高效：  **{ printf("mv %s %s\n"， $0， tolower($0)) | "sh" }**  **END { close("sh") }**  tolower() 函数返回参数字串中所有的大写字母被转换为小写的串（查看，***在第 184 页***）。程序构建了一系列的命令行，并使用mv 工具对文件进行改名，然后将这些命令列表发送给Shell 执行。  查看，***在第 232 页***，了解可以辅助产生可以发送到Shell 的命令行函数。 |

## 5.7 标准的预打开的特殊文件数据流

一般地，正在执行的程序有三个输入输出游戏已经可以用来进行读写。这就是众所周知的标准输入，标准输出以及标准错误输出。这些已经打开的流（以及任意已经打开的文件或者管道）技术上可以由所谓的文件描述符来表示。

默认情况下，这些流是你的键盘与你的屏幕，但是他们常常通过‘<’，‘<<’，‘>’，‘>>’，‘>&’ 与‘|’ 操作符，在Shell 中被重定向。标准错误一股用于写错误信息，有两个分离流（标准输出与标准错误输出）的原因是它们可以被单独地进行重定向。

在awk 的传统实现中，在awk 程序中将错误信息写到标准错误输出的唯一方式如下：

**print "Serious error detected!" | "cat 1>&2"**

通过打开一个到Shell 命令的管线，而该命令可以访问从awk 继承而来的标准错误流。这样写没那么优雅，而且也需要开启一个独立的进程。因此，人们在awk 程序不常这么干，而是通过下面的方式来输出信息到屏幕：

**print "Serious error detected!">"/dev/tty"**

（/dev/tty 是一个由操作系统提供的特殊文件，该文件与你的键盘与屏幕连接。它表示的是“终端”[[36]](#footnote-37)（Terminal），在现在系统中是键盘与屏幕，而不是串行控制台。）这一般有相同的效果，但是并不总是如此：尽管标准错误输出一般就是屏幕，但是它可以被重定向。当发生这种情况时，写到屏幕就不正确了。事实上，如果awk 是一个后台任务，它根本主没有终端。这样，打开 /dev/tty 就会失败。

gawk，GWK awk 以及mawk 提供了访问三个标准流的特殊文件名。如果gawk 在重定向输入输出时，文件名匹配这其中之一，它就直接使用这个名字所所代表的文件描述符。这此特殊文件名在所有的 gawk 已经***移植***到的操作系统中都有效，并不仅仅是POSIX 兼容的系统：

/dev/stdin

标准输入（文件描述符为0）。

/dev/stdout

标准输出（文件描述符为 1）。

/dev/stderr

标准错误输出（文件描述符 2）。

复用这些工具，恰当的将错误信息输出的方式就如下：

**print "Serious error detected!">"/dev/stderr"**

注意在文件上使用的双引号。如其他的重定向，其值必须是字串。常见的错误就是省略了引号，没有引号会产生令人疑惑的结果。

gawk在POSIX 兼容模式下，并不将这些文句特殊对待。但是，因为BWK awk 支持它们，gawk 也支持它们，就算使用了 --traditional 选项也是一样（查看，***在第 35 页***）。

## 5.8 gawk 中的特殊文件名

除了标准输入，标准输出与标准错误输出，gawk 还提供了访问任意已经打开文件描述符的方法。另外，还保留了用于TCP/IP 网络的特殊文件名。

### 5.8.1 在gawk访问其他已开文件

除了***/dev/stdin， /dev/stdout***， 与***/dev/stderr***这些早前提过的特殊文件外，gawk 还提供了访问已经打开的文件的语法：

/dev/fd/N

与文件描述符N 关联的文件。该文件必须由启动awk 程序执行的程序（一般是Shell）打开的文件。除非在调用gawk的Shell 中多花一些功夫，不然只会有描述符 0，1 与 2 是打开的。

文件名 /dev/stdin， /dev/stdout， 与 /dev/stderr实际上就是/dev/fd/0， /dev/fd/1， 与 /dev/fd/2的别名。但是，这些名字更有说明性。

注意，对于文件描述符号大于2以上的文件形式“/dev/fd/N” 执行 close() 函数，才会真正地执行关闭（close）操作。

### 5.8.2 用于网络的特殊文件

gawk程序可以打开双向通信的TCP/IP 连接，正如一个客户端与一个服务端一样。这可以通过下面形式的特殊文件来实现：

**/net-type/protocol/local-port/remote-host/remote-port**

net-type 是***‘inet’***， ***‘inet4’***， 或者***‘inet6’***之一。protocol 是‘tcp’ 或者 ‘udp’ 之一，其他的参数则代表了要启动一个网络连接所需要的其他信息。这些文件名与‘|&’ 一起使用用来进行并程通信（查看，***在第 304 页***）。这是一个高级特性，为了完整性在这里先说明一下。完整的讨论要到，***在第 306 页***。

### 5.8.3 特殊文件名说明

如果要使用 gawk 提供的一些特殊文件名，则下面有一些要记住的点：

* 对于三个预打开的标准文件的识别只有在 POSIX 模式下才会被禁止。
* 对于其他特殊文件名的识别，只有 gawk 处于兼容模式时才会被禁止（要么使用 --traditional，要么是使用 --posix 选项，查看 ，***在第 35 页***）。
* gawk 总是会解释这些特殊文件名。例如，使用 ‘/dev/fd/4’ 为输出，实际上写入的是描述符为4的文件，而不是使用 dup 复制4 描述符的新文件。大多数情况下，这没有什么问题。但是，不要关闭任意与文件描述符 0，1，2 相关的文件，这个相当重要。这么做的话，会导致不可预知的行为。

## 5.9 关闭输入输出重定向

在 awk 程序执行期间，如果同样的文件名或者同一个 Shell 命令被 getline 使用过多于一次（查看 ），则文件（或者命令的执行）只在第一次发生。从那以后，从中读取第一个输入记录。通过 getline 再次读取同一个文件或者命令时，第二条记录从其中获取，以此类推。

相似地，当文件或者管道被打开用于输出时，awk 会记录文件名或者与其相关的命令，后续对于同一个文件或者命令写入，会被附加到前一个写操作之后。真到 awk 退出后，文件或者管道才会被关闭。

这意思是说，为了再次从头开始读取同一个文件，或者重新执行一个Shell 命令，你需要一些特别步骤（而不是从同一个命令中读取更多的输出）。close() 函数就是用来执行这个步骤：

**close(filename)**

或者：

**close(command)**

filename 与 command 参数可以是任意的表达式。它的值必须完全匹配用以打开的文件或者启动的命令行的字串（空格与其他“无关”的字符也包括在内）。例如，如果你像这样打开一个管道：

**"sort -r names" | getline foo**

然后你必须这样来关闭它：

**close("sort -r names")**

只要函数调用执行过，下一个从那个文件或者管道进行 getline 时，或者下一个 print 与 printf 到那个文件或者管道，会被重新打开或者重新执行命令。由于你用来关闭的文件或者管线必须完全匹配你用来打开的文件或者执行的命令，所以用一个变量来存储文件名或者命令是一个好的编程实践。上面的例子因此可以变成这样：

**sortcom = "sort -r names"**

**sortcom | getline foo**

**...**

**close(sortcom)**

这可以帮助你避免awk 程序中那些很难查找的输出错误。这也有一些理由为什么要关闭输出文件：

* 为了写一个文件，之后不久，在同一个程序中，从其中读取。在写完之后要关闭它，然后再使用 getline 进行读取。
* 为了在 awk 程序中，连续地写大量的文件。如果文件没有被关闭，awk 有可能最终会超过系统对于同一个进程中所能打开的文件数的限制。因此最好是在写完每个文件时关闭它。
* 为了完成一个命令。当输出重定向是通过管道进行的。从这个管道读取的命令会持续尝试从管道中读取，只要管道是打开的。通常情况下，相关的命令直到管道关闭的情况下才能继续干自己的事情。例如，如果输出被重定向到了邮件程序，消息直到管道被关闭了才实际地发送完。
* 为了执行第二次，利用相同的参数，执行相同的程序。这与比第一次执行输入更多的数据不是一件事情！例如，假设一种程序会将输出通过管道输出给邮件程序。如果输出了几行重定向到了这个管道而没有关闭，则它们会生成一个包括几行数据的简单的消息（邮件）。另一方面，如果程序每输出一行就关闭一次管道，则每一行都成一个一独立的消息（邮件）。

如果你使了比系统所允许你打开的文件数更多的文件，gawk 会尝试复用你的数据文件中已经打开的文件。gawk 这样的能力依赖于你的操作系统所提供的工具，因此并不一定总是会有用。所以，无论对于是好的实践也好，还是好的可移植性建议也好，当处理完一个文件时，就应该通过 close() 将它们关闭。事实上，如果你使用了大量的管道，在使用完后进行关闭是很重要的。例如，考虑下面的代码：

**{**

**...**

**command = ("grep " $1 " /some/file | my\_prog -q " $3)**

**while ((command | getline) > 0) {**

**process output of command**

**}**

**# need close(command) here**

**}**

在这个例子中创建了基于每个记录的管线，如注释中所述的，没有调用 close() 函数，则 awk 会创建一个子进程来执行命令，直到最终由于请求更多的管线而耗尽文件描述符。

尽管每个命令都已经执行完成（通过 getline 的 end-of-file 的返回值指示），子进程却没有结束，[[37]](#footnote-38)更重要的是，与管道关联的描述符真到 awk 退出后，才会调用 close() 进行关闭释放。

close() 如果接收的不是一个文件、管道或者并程这样的打开用于重定向的参数，则默默地什么也不干。在这种情况下，它的返回值是负值，表示错误。另外，gawk 会设置 ERRNO 的值为指示对应的错误。

也要注意到 ***‘close(FILENAME)’*** ，针对从命令行中读取文件名这样的隐式循环没有什么“魔法”效应。更有可能的是，对于文件的关闭，在重定向中不会再打开，因此 awk 会什么也不做，而是返回一个负值。

当使用 ‘|&’ 操作符与并程进行通信时，关闭双向通信中的一端偶尔的情况下会非常有用。这只需要在调用 close() 时提供第二个参数就可以完成。如其他地方调用 close()类似，第一个参数是命令名，或者是用于开启并程的特殊文件。而第二个参数是一个字串，其值是 “to” 或者是 “from”，大小写无关。这是一个高级特性，会到 ，***在第 304 页***，进行详细的***讨论***，并提供更多的实际例子。

|  |
| --- |
| 使用 close() 的返回值  在许多 Unix awk 的老版本中，close() 函数实际上是一条语句，尝试利用 close() 的返回值是一个语法***错误***：  **command = "..."**  **command | *getline* info**  **retval = close(command) # syntax error in many Unix awks**  gawk 将 close() 当成一个函数。如果其参数表示从未打开的东西（文件、管道、并程），则返回值-1，或者是系统问题导致文件或者进程的关闭，则也会返回-1。在这个情况下，gawk 将预定义的变量 ERRNO 的值设定为描述问题的字串。  在 gawk 中，当关闭管道或者并程时（输入或输出），其返回值就是命令的退出状态。[[38]](#footnote-39)否则，它就是从系统的 close() ***或者*** C 函数 close() 的关闭输入输出文件时的返回值。如果成功，则值返回为0，失败下返回-1。  POSIX 标准非常模糊地说，close() 在成功的时候返回0，而失败时返回非零值。一般情况下，不同的实现在关闭管道时会报告不同的结果，因此，返回值是不可移植的。在 POSIX 模式下（查看 ，***在第 35 页***），gawk 在关闭管道时，只返回0。 |

## 5.10 总结

* print 打印逗号分隔的表达式。每个表达式在输出时都由 OFS 的值进行分隔，并以 ORS 的值结束。OFMT 提供了对数值 print 输出时的格式转换。
* printf 语句提供了非常好的输出控制粒度，通过格式控制字母来输出不同的数据类型。同时，用不同的标志来修改格式控制字母的行为。
* print 与 printf 的输出都可以重定向到文件、管道与并程。
* gawk 提供了特殊的文件名用于访问标准输入、标准输出与标准错误输出，以及进行网络通信。
* 使用 close() 来关闭打开的文件、管道以并并程的重定向。对于并程来说，可以通过关闭某个方向的通信。

## 5.11练习

1. 重写下面的程序：

**awk ’BEGIN { print "Month Crates"**

**print "----- ------" }**

**{ print $1， ""， $2 }’ inventory-shipped**

它来自于 ，***在第 96 页***，并在输出中使用 OFS 变量。

1. 使用 printf 语句来对齐头部与数据表，例子来自由 ，***在第 94 页***，inventory-shipped。
2. 在下面的代码中，如果你没有用双引号在重定向输出时会发生什么：

**BEGIN { print "Serious error detected!"> /dev/stderr }**

# 第六章 表达式

表达式是 awk 模式与动作的基本构造块。表达试求值后可用于打印，测试或者传递给函数。另外，表达式还可以通过赋值操作符将一个新值赋给一个变量。

表达式本身可以成为一个模式，或者动作语句。多数其他类型的语句包含一个或者多个表达式，来指定要操作的数据。如在其他语言中一样，awk 中的表达式可以包含变量、数组参考、常量以及函数调用，还可以是这些内容与不同的操作符的各种组合。

## 6.1 常量、变量与转换

表达式由值与处理这些值的操作组成。本节描述在表达式中使用的基本元素对象。

### 6.1.1 常量表达式

最简单的表达式类型就是常量，它们总是有不变的值。有三个常量：数值常量、字串常量与表达式常量。

如果你所需要的值不需要变化，那么你可以在适当的上下文中使用它们。数值常量又有不同的形式，但是在内部，它们都以相同的方式进行存储。

#### 6.1.1.1 数值与字串常量

数值常量表示数值。此数值可以是整数，十进制小数或者是科学计数形式（幂指数）的形式。[[39]](#footnote-40)下面的例子只所有形式的常量的值都是一样的：

**105**

**1.05e+2**

**1050e-1**

字串常量由双引号括起来的字符序列组成。如：

**"parrot"**

表示字串内容为 ‘parrot’。在 gawk 中的字串可以有任意长度，可以包括任何可能的 8比特的 ASCII 字符，包括 ASCII 的 nul字符（字符编码为0）。其他的 awk 实现有可能在处理一些字符编码时存在一些困难。

#### 6.1.1.2 八进制与十六进制数值

在 awk 中，所有的数值都小十进制数（以10为进制）。很多的其他编程变速器允许你指定其他进制的数，一般为八进制（以8为进制）和十六进制（以16为进制）。在八进制中，数值为 0，1，2，4，5，6，7，10，11，12 等等。正如十进制中的‘11’，是1乘10，然后加1，因此八进制中的‘11’，表示是1乘8，然后加1。在十进制中它等于9。对于十六进制，有16个数字位。由于我们日常的十进制数值系统中只有10个数字位（‘0’-‘9’），因此，‘a’ 到 ‘f’ 这6个字符用来表示剩下的部分。（大小写通常是无关的，十六进制的 ‘a’ 与 ‘A’ 有相同的值。）因此，十六进制中的 ‘11’ 表示的是 1 乘 16，然后加1，在进制中则是17。

通过 ‘11’ 这个普通字串，你无法知道它是什么进制的。因此，在C，C++以及其他的从C派生出来的语言中。有一个特殊的符号用来指明进制。八进制以 ‘0’ 开头，而十六进制则以 ‘0x’ 或者 ‘0X’ 开头。

11 十进制值的11

011 八进制的11，十进制的9

0x11 十六进制的11，十进制的17

这个例子体现了其中的不同：

**$ gawk ’BEGIN { printf "%d， %d， %d\n"， 011， 11， 0x11 }’**

**-| 9， 11， 17**

当在你的程序中，所处理的数据不能方便地进行表示成字串或者正则表达式时，使用这些的八进制或者十六进制常数则非常有用，例如各种不同的二进制数据。

gawk 可以让你在程序文件中使用八进制或者十六进制数值。但是，这样的数值在输入时会区别对待，默认这么做会导致旧的程序无法正常执行。（如果你确实要这么做，使用 --non-decimal-data 这个命令行选项，查看 ，***在第 297 页***）如果你有八进制或者十六进制的数据，你可以使用 strtonum() （查看 ，***在第 184 页***）函数来将这些数据转换为数值。大多数时候，你在使用内建的位操作函数时，你都会使用八进制或者十六进制数据。可以查看 ，***在第 201 页***，来获取更多的信息。

不像早期的 C 实现，‘8’ 与 ‘9’ 不是有效的八进制常数。例如，gawk 会将 ‘018’ 当成是十进制的18：

**$ gawk ’BEGIN { print "021 is"， 021 ; print 018 }’**

**-| 021 is 17**

**-| 18**

八进制与十六进制源代码常量是 gawk 的扩展。如果 gawk 处于兼容模式（查看 ，***在第 35 页***），则它们是不可用的。

|  |
| --- |
| 常量的进制不会影响它的值  只要数值常量被转换成了数值的内部表示，gawk 就不再记住常量的原始格式，gawk 总是会使用内部的值。这在将数值转换成字串时有一个特殊的结果：  **$ gawk ’BEGIN { printf "0x11 is <%s>\n"， 0x11 }’**  **-| 0x11 is <17>** |

#### 6.1.1.3 正则表达式常量

正则表达式常量是一个包含在一对斜杠中的正则表达式描述。例如 ***/^beginning and end$/***。awk 程序中使用的大多数正则表达式都是常量，但是 ‘~’ 与 ‘!~’ 匹配操作符也可以匹配计算型，即动态型正则表大式（典型的情况下，它们就是包含正则表达式的普通的字串或者变量，但是是更复杂的表达式而已）。

### 6.1.2 使用正则表达式常量

当在 ‘~’ 或者 ‘!~’ 操作的右手边时，正则表达式仅权表示要匹配的正则表达式。但是正则表达式常量（例如 /foo/）可以当成简单的表达式来使用。当正则表达式常量单独出现时，与它在模式中出现时有相同的含义（如 ***‘($0 ~ /foo/)’***）查看 *，****在第 139 页***。这表示下面的两个代码片段：

**if ($0 ~ /barfly/ || $0 ~ /camelot/)**

**print "found"**

与：

**if (/barfly/ || /camelot/)**

**print "found"**

是完全等同的。这个规则的一个非常奇怪的结果就是下面的布林表达式是有效的，但是并不是作者所认为的那样执行：

**# Note that /foo/ is on the left of the ~**

**if (/foo/ ~ $1) print "found foo"**

这个代码“很明显”地是要测试 $1 是不是匹配正则表达式 /foo/。但是实际上，表达式 ***‘/foo/ ~ $1’*** 表示的是 ***‘($0 ~ /foo/) ~ $1’***。换句话说，首先匹配输入记录与正则表达式 /foo/，所得到的结果要么是0，要么是1。这就要看匹配的是成功还是失败。匹配的结果然后再与记录的第一个域进行匹配。由于你不可能是想做这样的测试，gawk 会针对程序中这样的结构发出一个告警。这个规则的另一个结果就是如下的匹配语句：

**matches = /foo/**

是要么将0，要么是将1赋值给变量 matches，内容依赖于当前的输入记录。

正则表达式也可以用于 ***gensub()***， ***sub()***与 ***gsub()*** 函数的第一个参数，或者是 match() 函数的第二个参数，以及 split() 与 patsplit() 函数的第三个参数（查看 ， ***在第 184 页***）。awk 的现代版本，包括 gawk，允许 split() 的第三个参数是一个正则表达式常量，但是旧一点的版本不允许。

由于一些内置的函数接受正则表达式会它的参数，因此将其作为用户自定义函数（查看 ， ***在第 204 页***）的参数时，使用正则表达式会带来困惑。如：

**function mysub(pat， repl， str， global)**

**{**

**if (global)**

**gsub(pat， repl， str)**

**else**

**sub(pat， repl， str)**

**return str**

**}**

**{**

**...**

**text = "hi! hi yourself!"**

**mysub(/hi/， "howdy"， text， 1)**

**...**

**}**

在这个例子中，程序员想将正则表达式常量传递给用户自定义函数 mysub()，该函数又将其传给 sub() 或者 gsub()。但是真正发生的事情是，pat 参数会被赋值为1或者0，这就要看 $0 是否匹配 /hi/。gawk 会针对正则表达式常旦传递给用户自定义函数这种情况发出一个警告，因此将一个真值通过这种方式来传递有可能不是有意的。

### 6.1.3 变量

变量是在你程序的某个点存储值，以备程序的其他部分后续使用的方式。它们可以在程序文本的整个部分被操作，也可以在 awk 的命令行中被赋值。

#### 6.1.3.1 在程序中使用变量

变量可让你给值赋予一个名字，然后通过这个名字来使用它。变量在多个例子中都使用到了。变量的名字必须是一个字母、数字或者下划线的序列，并且不可以以数字开头。在这里，字母是任何的 52 个大小与小字的英文字母。其他的在非英语语言设置中定义的字符不可用于变量句。大小写在变量名中是区分的，a 与 A 是不同的变量。

变量本身就是一个有效的表达式：因为它代表了变量的当前值。变量通过赋值操作符、递增操作符与递减操作符可以赋予一个新值（查看 ， ***在第 122 页***）。另外， sub() 与 gsub() 函数可以改变变量的值，而 match()，split() 与 patsplit() 函数可以改变数组参数的值（查看 ，***在第 184 页***）。

少数几个变量有一些内置的含义，如 FS（域分隔符）以及 NF（当前记录的域数目）。查看 ，***在第 155 页***，可获取预定义的变量列表。这些预定义的变量可以像其他变量一样被使用与赋值，但是它们的值也可以被 awk 自动使用与更改。所有的预定义的变量的名字都是大写的。

awk 的变量可以被赋值为数值与字串。某个变量所持有值的类型在程序的生命期中是可以改变的。默认情况下，没有初始化的变量的值是一个空串，如果转换为数值则为0.没有必要显式地在 awk 中初始化变量，这些是在 C 语言中以及其他传统语言中才需要做的。

#### 6.1.3.2 在命令行中进行变量赋值

awk 变量可以在命令行上通过赋值操作符来设置（查看 *，****在第 41 页***）。这样的赋值有如下形式：

**variable=text**

通过这样的方式，变量要么在 awk 程序运行开始或者在输入文件之间进行设置。当 -v 选项先于赋值，如下面的形式：

**-v variable=text**

则变量将在程序的开始，甚至在 BEGIN 规则执行之间就设置了。-v 选项以及他的赋值必须在所有的文件名参数以及程序代码之前。（查看 ，***在第 35 页***，来获取关于 -v 选项的更多信息。）否则，变量赋值的执行决定于它在输入文件中的位置——要先处理先于该选项之前的文件。如：

**awk ’{ print $n }’ n=4 inventory-shipped n=2 mail-list**

会打印所有输入记录的域数值 n。在第一个文件读取之前，命令行参数设置变量 n 的值为4。这会打印来自 ***inventory-shipped*** 文件行中的第4个域。当处理完第一个文件后，但是在开始第2个文件之前，n 被设置成2，因此会打印来自 ***mail-list*** 文件的行的第2个域：

**$ awk’{ print $n }’ n=4 inventory-shipped n=2 mail-list**

**-| 15**

**-| 24**

**...**

**-| 555-5553**

**-| 555-3412**

**...**

命令行参数还可以在 ARGV 数组中获取，并利用 awk 程序进行显示检查（查看 *，****在第 163 页***）。awk 会处理命令行赋值参数的值中转义序列（查看 *，****在第 51 页***）。

### 6.1.4 在字串与数值之间进行转换

“数值到字串”以及“字串到数值”的转换一般都是直接的。但是要了解几个细微的地方，通过这小节来说明 awk 中这些重要的细节。

#### 6.1.4.1 awk 如何转换字串与数值

如果 awk 程序上下文要求，字串会转换成数值，数值也可以转换成字串。例如，在表达式 ***'foo + bar’*** 中如果 foo 或者 bar 如果是一个串，它就会转在执行加操作之前转换成数值。如果数值出现在字串的连接操作里，它们就会转换成字串。考虑如下的代码：

**two = 2; three = 3**

**print (two three) + 4**

这个代码会打印（数值）值27。two 与 three 的数值值被黑铁成字串并连接在一起。结果字串被转换成数值23，用于与4进行相加。

由于某种原因，如果你需要强制数值被转换成字串，则将这个数值与空串（""）连接。强制一个字串要转换成数值，则半这个字串加上0即可。字串转换成数值是通过将字串的数字形式的前缀串解释当成数值：“2.5” 会被转换成 2.5，“1e3”会被转换成 1000，“25fix”会被转换25。无法转换成数值的字串则会转换成0。将数值转换成字串上的准确方式是由 awk 的预定义变量 CONVFMT 进行控制的（查看 *，****在第 155 页***）。数值转换成字串是使用 sprintf() 函数并将 CONVFMT 当做模式指定符（查看 *，****在第 184 页***）。

CONVFMT 的值默认是 “%.6g”，表示的是创建的值最多有6个有效数字位。对于一些应用来说，你可以将其更改得更加精确。在大多数现在机器上，17位的数值通常足够将捕获浮点数的值了。[[40]](#footnote-41)

如果你将 CONVFM 的值设置成一个无法告知 sprintf() 如何来以一种有用的方式格式化浮点数的字串，则会产生奇怪的结果。例如，如果你忘记了格式指定符中的 ‘%’， awk 会将所有数值转换成一样的字串常量。

作为一个特殊例子，如果数值是一个整数，转换成字串的结果总是一个整数，无论 CONVFMT 的值是什么。如下面的代码片段：

**CONVFMT = "%2.2f"**

**a = 12**

**b = a ""**

b 的值为 “12”，而不是 “12.00”。

#### 6.1.4.2 语言设置会影响转换

当要在字串与数值之间进行转换时，什么地方会对此产生影响？本地字符设置与语言，即 locale，会影响数值的格式。特别是，对于 awk 程序，它还会影响十进制小数点字符以及百分位字符。“C” 语言设置或者大多数的英语语言，使用点符号（‘.’）当做为十进制数小数点，并且没有百分位。但是，许多（如果不是大多数）的欧洲以及非英语语言区域使用逗号（‘，’）作为十进制小数点字符。欧洲语言通常在需要的时候使用空格或者点来作为百分位。

POSIX 标准规定，当读取 awk 程序源代码，以及在读取命令行赋值变量时，总是使用点号作为十进制小数点（查看 *，****在第 41 页***）。但是，当解释输入数据时、进行 print 或者 printf 输出，以及进行数值与字串转换时，则使用本地的十进制小数点。在所有的情况下，源代码中的数值，以及输入数据不可以有百分位分隔符。下面的代码在 GNU/Linux 系统解释了这些差异：

**$ export POSIXLY\_CORRECT=1 Force POSIX behavior**

**$ gawk ’BEGIN { printf "%g\n"， 3.1415927 }’**

**-| 3.14159**

**$ LC\_ALL=en\_DK.utf-8 gawk ’BEGIN { printf "%g\n"， 3.1415927 }’**

**-| 3，14159**

**$ echo 4，321 | gawk ’{ print $1 + 1 }’**

**-| 5**

**$ echo 4，321 | LC\_ALL=en\_DK.utf-8 gawk ’{ print $1 + 1 }’**

**-| 5，321**

en\_DK. utf-8 是丹麦英语语言，在这个设置中，逗号是作为小数点使用的。在正常的 “C” 语言设置中，gawk 将 ‘4，321’ 当成是4，而在丹麦语言设置中，则被完全当成了一个数值，其包含分数为 4.321。

一些早期的 gawk 版本完全按照标准进行编译。但是很多的非英语语言的用户抱怨这样的行为，因为他们用点表示十进制里的小数点，所以，默认的行为又恢复到使用点来做为小数点符号。你可以使用 --use-lc-numeric （查看 ，***在第 35 页***）选项来强制 gawk 使用本地语言设置中的十进制小数点。（gawk 在 POSIX 模式中也使用本地语言设置中的十进制小数点，不管是通过 --posix 选项或者是 POSIXLY\_CORRECT 环境变量来指定。）

表 6.1 说明了什么时候会使用本地语言设置中的十进制小数点符号，什么时候使用点作为十进制小数点符号。还有一些特性还没有涉及到。

|  |  |  |
| --- | --- | --- |
| Feature | Default | --posix or --use-lc-numeric |
| %’g | Use locale | Use locale |
| %g | Use period | Use locale |
| Input | Use period | Use locale |
| strtonum() | Use period | Use locale |

表 6.1：本地语言设置中的十进制小数点与圆点

最后，现在的正式标准以及 IEEE 标准的浮点数表示法对于 gawk 转换一些特定的字串为数值的方式，都有一些不常见但是非常重要的影响。相关的细节呈现在 *，****在第 346 页*** 。

## 6.2 操作符：对值进行处理

这一节介绍利用常量与变量值的操作符。

### 6.2.1 算术操作符

awk 语言使用常用的算术操作符来对表达式求值。所有的这些算术表达式都具有正常的优先级，并按你所期望的方式工作。

下面的例子使用一个文件名为 grades 的文件，其中包含了学生名字与每个学生的三个测试成绩（班级确实小）：

**Pat 100 97 58**

**Sandy 84 72 93**

**Chris 72 92 89**

下面的程序读取文件 grades 并打印平均成绩：

**$ awk ’{ sum = $2 + $3 + $4 ; avg = sum / 3**

**> print $1， avg }’ grades**

**-| Pat 85**

**-| Sandy 83**

**-| Chris 84.3333**

下面的列表列出了 awk 中的算术操作符，并按优先级降序排序：

x ^ y

x \*\* y 幂乘，表示 x 的 y 次幂。‘2^3’ 的值为8，字符序列 ‘\*\*’ 与 ‘^’ 等同。（c.e.）

- x 取负。

+ x 一元加，表达式会被转换成数值。

x \* y 乘。

x / y 除。由于在 awk 中的所有数值都是浮点数，结果不会舍入输出为整数——‘3/4’的值为0.75。（这通过容易出错，尤其对 C 程序员，会忘记 awk 中的数值都是浮点数，看起来是整数的除，产生一个实数，而不是整数。）

x % y 求余，进一步的讨论会在下面。

x + y 加。

x - y 减。

一元加以减都有相同的优先级，乘法操作符都有相同优先级，加与减具有相同的优先级。

当计算 ‘x%y’的余数时，商会向零舍入舍出，并被y乘。这个结果再被从x减去。本操作有时候会称为‘trunc-mode’ 操作，下面的关系总是会成立：

**b \* int(a / b) + (a % b) == a**

这种定义下，如果在 ‘x%y’ 在 x 为负值时，其也为负值，这可能不是预期的结果：

**-17 % 8 = -1**

在其他的 awk 实现中，余数的符号可能跟机器有关。

提示：POSIX 标准中只指定了 ‘^’ 作为幂乘。为了最大的可移植性，不要使用 ‘\*\*’ 操作符。

### 6.2.2 字串连接

*在那个时候看起来确实是个好主意。*

—Brian Kernighan

只有一个字串操作符：连接。实际上并没有特定的操作符来表示它。相反，连接操作是通过将表达式写成两两相邻。例如：

**$ awk ’{ print "Field number one: " $1 }’ mail-list**

**-| Field number one: Amelia**

**-| Field number one: Anthony**

**...**

在字串常量 ‘:’ 后面没有空格，整行输出在一起。比如：

**$ awk ’{ print "Field number one:" $1 }’ mail-list**

**-| Field number one:Amelia**

**-| Field number one:Anthony**

**...**

由于字串连接没有显式的操作符，因此通常需要保证这发生成合适的时间，这是通过将要连接的项用括号包起来。例如，你可能想要下面的代码来连接起文件与名字：

**file = "file"**

**name = "name"**

**print "something meaningful"> file name**

这在 Unix awk [[41]](#footnote-42)中会产生一个语法错误。应该如下使用：

**print "something meaningful"> (file name)**

在字串连接两边，括号在除了 ‘=’ 右边的所有情况下都应该使用。在特别注意在字串连接中所使用的表达式。特别是，表达式的求值顺序用来进行字串连接，在 awk 语言中是没有定义的。如下面的例子：

**BEGIN {**

**a = "don’t"**

**print (a "" (a = "panic"))**

**}**

是否是第二个赋值操作发生在 a 的值返回之前还是之后这是不确定的。结果可以是 ***‘***don’t panic'， 或者是 ***‘***panic panic***’***。

在与其他的操作符混合时，连接操作的优先级通常是反直觉的。如下面的例子：

**$ awk ’BEGIN { print -12 " " -24 }’**

**-| -12-24**

很明显的是要连接 -12，一个空格与 -24。但是空格消失到哪里去了呢？答案是在操作符的优先组合与 awk 的自动转换规则里。为了得到所想要的结果，像下面这样写：

**$ awk ’BEGIN { print -12 " " (-24) }’**

**-| -12 -24**

这强制 awk 将 ‘-24’ 中的 ‘-’ 当成是一元操作符。否则，它会这么来分析：

**-12 (" " - 24)**

**→-12 (0 - 24)**

**→ -12 (-24)**

**→ -12-24**

如早先所提，当连接操作与其他操作符混合时，用括号括起来。否则，你无法确认你会得到什么结果。

### 6.2.3 赋值表达式

赋值表达式是将值存入某个变量的表达式。例如，我们将变量 z 的值赋为 1：

**z = 1**

当这个表达式执行后，变量 z 的值就为1。不管 z 的旧值是什么，都将丢弃。

赋值也可以存储字串值。例如，下面将值“this food is good”存储到变量 message 中：

**thing = "food"**

**predicate = "good"**

**message = "this " thing " is " predicate**

这个例子也呈现了字串连接操作。 ‘=’ 符号称为赋值操作符。这也是最简单的操作符，因为其右边的操作数的值会原封不动地存下来。大多数操作符（加，连接等）除非要计算值，否则都没有实际效果。如果值没有被使用，也没有理由使用这个操作符。赋值操作符则不同：因为它确实产生一个值，但是就算你忽略它，赋值仍然使自己走完变量的变化过程。我们称这个效应为副作用。

赋值左边的操作数不需要是一个变量（查看 ， ***在第 116 页***），其也可以是一个域（查看 ， ***在第 69 页***） 或者一个数组元素（查看 *，* ***在第 112 页***）。所有这些都称为值，这意味着它们可以出现在赋值操作符左边。右边的操作数可以是表达式，其会产生一个新的值来赋值存储在指定的变量、域或者数组元素。（这样的值称为右值（rvalues）。）

要注意的是，变量没有固定的类型。变量的类型就是其最后被赋值的数据的类型。在下面的程序片段中，变量 foo 先具有数值类型，然后具有字串类型：

**foo = 1**

**print foo**

**foo = "bar"**

**print foo**

当第二个赋值给了 foo 一个字串值，而这之前其具有数值类型就丢弃了。

字串值中，如果不是一个以一个数值开头的话，其对应的数值值为0。在执行了下面的代码后，foo 的值为 5：

**foo = "a string"**

**foo = foo + 5**

提示：将变量作为一个数值，然后作为字串有时候会产生困惑，而且也是一个不好的编程风格。前面的两个例子只是用来说明 awk 如何工作，而不是告诉大家这么来写程序的！

赋值是一个表达式，因此它有值——一个与其被赋予的相同的值。因此 ‘z=1’ 是一个具有值为1的表达式。这个特性的一个结果就是你可以将多个赋值写在一起，如：

**x = y = z = 5**

这个例子中，5这个值 被赋给了三个变量（x，y 与 z）。这是因为 ‘z=5’ 的值为5，然后再存于y中，因此 ‘y=z=5’ 的值为5，最后存到 x 中。

赋值可以被用于任何可称为表达式的地方。例如，这么写 ‘x!=(y=1)’ 是有效的形式，其将 y 的值设成1，然后测试 x 是否等于1。但是这样的风格会导致程序难以阅读，这样的赋值嵌套应该要避免，除非是在一次性的程序里。

除了 ‘=’ 外，还有几个其他的操作符用于执行与变量旧值的算术运算。例如，操作符 ‘+=’ 将右边的值加到变量的旧值以生成新值。所以，下面的赋值会在 foo 值的基础上加上5。

**foo += 5**

这与下面的代码是等同的：

**foo = foo + 5**

用哪一种形式，都可以清晰地表示你程序的意图。

有几个种情况使用 ‘+=’ （或者任意的赋值表操作符）与简单地将左边的操作数作为右边的表达式的重复是不一样的，如：

**# Thanks to Pat Rankin for this example**

**BEGIN {**

**foo[rand()] += 5**

**for (x in foo)**

**print x， foo[x]**

**bar[rand()] = bar[rand()] + 5**

**for (x in bar)**

**print x， bar[x]**

**}**

bar 这个下标一般在实际中都是不相同的，因为 rand() 每次调用时都产生不同的值。（数组与 rand() 函数都还没有讨论到，查看 *，* ***在第 112 页***，与 ，***在第 183 页***，来获取更多的信息。）这个例子说明了赋值操作符的一个重要事实：表达式的左边只会求值一次。

但是对于表达式的求值顺序，是先左边还是右边则是依赖于实现的。如下面：

**i = 1**

**a[i += 2] = i + 1**

a[3] 的值可以是 2， 也可以是4。

表 6.2 列出了所有的算术赋值操作符，在每种情况下，右边的操作数是一个值会被转换为值的表达式。

|  |  |
| --- | --- |
| Operator  操作符 | Effect  效果 |
| ***lvalue*** += increment | Add increment to the value of ***lvalue***.  将 increment 添加到 lvalue 的值上。 |
| ***lvalue*** -= decrement | Subtract decrement from the value of ***lvalue***.  在 lvalue 的值上减去 decrement。 |
| ***lvalue*** \*= coefficient | Multiply the value of ***lvalue*** by coefficient.  将 lvalue 的值乘上一个 coefficient。 |
| ***lvalue*** /= divisor | Divide the value of ***lvalue*** by divisor.  将 lvalue 的值除上一个 divisor。 |
| ***lvalue*** %= modulus | Set ***lvalue*** to its remainder by modulus.  设置 lvalue 的值与其值与 modulus 求模。 |
| ***lvalue*** ^= power | Raise ***lvalue*** to the power power.  将 lvalue 的值进行幂乘 power。 |
| ***lvalue*** \*\*= power | Raise ***lvalue*** to the power power. (c.e.)  将 lvalue 的值进行幂乘 power。(c.e.) |

表 6.2：算法赋值操作符

提示： POSIX 中只指定了 ‘^=’。为了移植性，不要使用 ‘\*\*=’ 操作符。

|  |
| --- |
| ‘/=’ 与正则表达式之间的二义性  在 /= 赋值操作符与正则表达式常量（哪个记录的第一个字符是‘=’）之间有一个语法上的二义性。这在一些商业性的 awk 版本里更显眼。例如：  **$ *awk* /==/ /dev/null**  **error*awk*: syntax error at source line 1**  **error context is**  **error>>> /= <<<**  **error*awk*: bailing out at source line 1**  解决方法是：  **awk’/[=]=/’ /dev/null**  gawk 没有这个问题，BWK awk 与 mawk 也没有。 |

### 6.2.4 递增与递减操作符

递增与递减操作符对变量的值进行递增与递减1。赋值操作符可以做相同的事情，因此递增操作符并没有给 awk 带来额外的功能，但是它们对于一些常见的操作的缩写非常方便。

用于加一的操作符写在 ‘++’。它可以用在递增的变量的前面与后面。在变量 v 前面递增写成 ‘++v’，这会对 v 值进行加一——新的值也是一个表达式。（赋值表达式 ‘v+=1’ 与其赞同。）在变量后面递增为后缀递增。这与前面的形式一样，不同的是递增表达式的值是变量的旧值。因此，如果 foo 的值 为4，则表达式 ‘foo++’ 的值为4，但是变量的值会变成 5。换句话说，操作符返回变量的旧值，但是作为副作用会对其递增。

后缀递增 ‘foo++’ 几乎与 ‘（foo+=1)-1’相同。但是又不完全相同，因为在 awk 中所有的数值都是浮点数——在浮点数中，‘foo+1-1’不一定与 foo 相同。但是这个差异非常小，只要你要处理的的值非常小（小于1012）。

域与数组元素也可以像变量一样递增。（当你想执行一个域参考，又对变量进行递增时，使用 ‘$(i++)’ 。括号是必须的，因为域操作符 ‘$’ 先于递增操作符。）

递增操作符 ‘--’ 与 ‘++’ 类型，只是其用于减而不是用于加。与 ‘++’ 一样，其用可用在左传的前面进行前缀递减，在后面进行后缀递减。下面是对于递增与递减操作符表达式的汇总：

++***lvalue*** 给 lvalue 递增1，并返回新值作为表达式的值。

***lvalue***++ 给 lvalue 递增1，并返回旧值作为表达式的值。

--***lvalue*** 给 lvalue 递减1，并返回新值作为表达式的值。（这个与表达式 ‘++lvalue’一样，只是用于减，而不是用于加。）

***lvalue***-- 给 lvalue 递减1，并返回旧值作为表达式的值。（这个与表达式 ‘++lvalue’一样，只是用于减，而不是用于加。）

|  |
| --- |
| 操作符求值顺序  *医生，当我这么干的时候就会受伤！*  *那么你就别这么干啊！*  —Groucho Marx  当我们像下面这样做，会发生什么：  **b = 6**  **print b += b++**  或者一些奇怪的：  **b = 6**  **b += ++b + b++**  **print b**  换句话说，当执行不同的后缀操作符（‘b++’）时，会有哪些副作用产生？产生的副作用是由实现来定义的，即依赖于 awk 的特定版本。第一个例子的结果可能是12或者13，对于第二个例子则可能是 22 或者是 23。  一句话，这么干是不建议的并且其移植性完全不可依赖。你要在你的程序中避免这样的代码。 |

## 6.3 真值与条件

在特定的上下文中，表达式的值会用来作为 “真值”，如，用来决定接下来程序应该执行什么。这一章节中会描述 awk 如何定义 ‘true’ 与 ‘false’ 以及值如何进行比较。

### 6.3.1 *awk* 的真与假

许多的编程语言对于 “true” 与 “false” 概念都有特殊的表示方式。这样的语言通常使用特殊的常量 true 与 false，以及对应的大写。但是 awk 却不同。它从 C 中借鉴了 true 与 false 的概念。在awk 中，所有非0的数值与非空串都是 true。而其他的值（0 或者空串，“”）都是 false。下面的程序会打印 ***‘A strange truth value’*** 三次：

**BEGIN {**

**if (3.1415927)**

**print "A strange truth value"**

**if ("Four Score And Seven Years Ago")**

**print "A strange truth value"**

**if (j = 57)**

**print "A strange truth value"**

**}**

这个“非0非空”的规则有一个令人惊讶的结果：字串 “0” 为真，因为它非空。

### 6.3.2 变量类型与比较表式式

*指南都是确定的，但是现实常常不精确。*

—Douglas Adams， The Hitchhiker’s Guide to the Galaxy

与其他的语言不同， awk 中的变量没有固定的类型。相反，它们即可以是数值，也可以是字串，这主要依赖于所赋给它的值来确定。我们现在来看看变量是如何来确定类型的，以及 awk 如何来比较这些变量。

#### 6.3.2.1 字串类型与数值类型

POSIX 标准中引入了数值式字串的概念，简单地说就是看起来像是一个数值的字串——如 “+2”。这个概念也用来确定变量的类型。变量的类型是非常重要的，因为两变量的类型决定了它们之间如何进行比较。变量类型的确定依赖下面的规则：

* 数值常量或者是数值操作结果具有数值属性。
* 字串常量或者字串操作结果具有字串属性。
* 域，getline 输出，FILENAME， ARGV 元素，ENVIRON 元素以及由 match()，split() 与patsplit() 生成的数组的元素，如果是数值字串，则具有 strnum 属性。否则就具有字串属性。未初始化的变量也具有 strnum 属性。
* 属性通过赋值操作符进行传递，但是使用则不会改变。最后的规则尤其重要。在下面的程序中，a 具有数值类型，就算之后它被用于字串操作中：

**BEGIN {**

**a = 12.345**

**b = a " is a cute number"**

**print b**

**}**

当两个操作数进行比较时，要么是进行字串比较，要么是进行数值比较。这依赖于操作数的属性，如下面的对称阵列：

|  |  |
| --- | --- |
|  | STRING NUMERIC STRNUM |
| STRING  NUMERIC  STRNUM | string string string  string numeric numeric  string numeric numeric |

基本的思想就是，用户的输入看起来像数值——也仅在用户输入中——就会被当成数值对待，就算它实际上是由字符组成的字串。所以，如这样的字串常量“+3.14”，当它出现在程序原代码中，是一个字串——就算它是一个数值——并且在比较目的中也不会被当成数值。

简单地说，当操作数是一个“纯粹”的字串，如字串常量，则就会执行字串比较。否则就会执行数值比较。

对此还要强调一下：所有的用户输入都由字符组成，因此首先它是字串类型，输出的字串看起来像是数值的就会赋给 strnum 属性。所以，下面这个6字符的输出串“ +3.14” 会具有 strnum 属性。相反地，下面的8字符串“ +3.14”出现在程序文本中则组成了一个字串常量。下面的例子中，当比较两个不同的常量时为真，因此打印 ‘1’，否则打印 ‘0’：

**$ echo ’ +3.14’ | awk ’{ print($0 == " +3.14") }’ True**

**-| 1**

**$ echo ’ +3.14’ | awk ’{ print($0 == "+3.14") }’ False**

**-| 0**

**$ echo ’ +3.14’ | awk ’{ print($0 == "3.14") }’ False**

**-| 0**

**$ echo ’ +3.14’ | awk ’{ print($0 == 3.14) }’ True**

**-| 1**

**$ echo ’ +3.14’ | awk ’{ print($1 == " +3.14") }’ False**

**-| 0**

**$ echo ’ +3.14’ | awk ’{ print($1 == "+3.14") }’ True**

**-| 1**

**$ echo ’ +3.14’ | awk ’{ print($1 == "3.14") }’ False**

**-| 0**

**$ echo ’ +3.14’ | awk ’{ print($1 == 3.14) }’ True**

**-| 1**

#### 6.3.2.2 比较操作符

比较表达式在数值或者字串中进行比较，比如说相等性。它们通常使用关系操作符来表现，这些操作是 C 中对应的操作的超集。表 6.3 中有说明。

表达式 结果

x < y 如果 x 小于 y 则为真

x <= y 如果 x 小于或者等于 y 则为真

x > y 如果 x 大于 y 则为真

x >= y 如果 x 大于或者等于 y 则为真

x == y 如果 x 等于 y 则为真

x != y 如果 x 不等于 y 则为真

x ~ y 如果字串 x 与 y 所代表的正则表达式匹配则为真

x !~ y 如果字串 x 与 y 表示的正则表达式不匹配则为真

subscript in array

如果数组 array 在下存在下标 subscript 所指明的元素则为真

表 6.3：关系操作符

比较表达式的结果为 true，则值为1，反之则为0。当不同的类型之间进行比较时，数值类型的操作符会按照 CONVFMT （查看 ， ***在第 117 页***）的格式转换为字串。

字串的比较每个串的第一个字符，然后第二个字符，依次进行比较。因此，“10” 小于 “9”。如果两个串中，一个串是另一个串的前缀，短的串小于长的串。所以，“abc” 小于 “abcd”。

很容易不小将 “==” 操作符少打一个 “=” 符号。结果还是一个有效的 awk 代码，但是程序执行的可就不是你的意图了：

**if (a = b) # oops! should be a == b**

**...**

**else**

**...**

除非 b 碰巧为0或者是空串，否则 if 的测试部分总会是成功的（为 true）。因此操作符看起来这么像，此类的错误会非常难发现。

下面的表达式列表说明了 awk 执行的比较类型，以及比较的结果：

1.5 <= 2.0 数值比较（true）

"abc">= "xyz" 字串比较（false）

1.5 != " +2" 字串比较（true）

"1e2"<"3" 字串比较（true）

a = 2; b = " 2"

a == b 字串比较（true）

a = 2; b = " +2"

a == b 字串比较（false1）

在下面的例子中：

**$ echo 1e2 3 | awk ’{ print ($1 < $2) ? "true" : "false" }’**

**-| false**

结果为 'false'，因为 $1 与 $2 都是用户输入。它们都是数值字串——因此两个都有 strnum 属性，所以执行的是数值比较。比较规则的目的以及数值字串的使用是试图产生“最少惊讶”的行为，同时还会“做正确的事”。

字串比较与正则表达式比较是非常不同的，如：

**x == "foo"**

具有值 1，或者 true，如果变量 x 确实为 ‘foo’。相反：

**x ~ /foo/**

则当 x 包含 ‘foo’ 时为1，如 "Oh， what a fool am I!"（天啊，我真是蠢得要死！）

***‘~’*** 与 ***‘!~’***右边的操作数可以是正则表达式常量（/.../），也可以是一个普通的表达式。在后面的情况下，表达式的值会当成字串，并作为动态正则表达式（查看 ，***在第 50 页***，与， ***在第 58 页***）。

带斜杠的正则表达式常量其本身也是一个表达式。/regexp/ 是下面比较操作表达式的缩写形式：

**$0 ~ /regexp/**

/foo/ 不是 ‘$0 ~ /foo/’ 的缩写的一个特殊地方就是它在操作符 ‘~’ 或者 ‘!~’ 的右边。查看 ，***在第 114 页***，在此处会有更详细的讨论。

#### 6.3.2.3 POSIX 中的字串比较规则

POSIX 标准规定，字串的比较是按照本地语言设置的分类顺序进行的。这个顺序字符是按照本语言设置经过排序的（更多的讨论可查看 ， ***在第 136 页***）。这个顺序完全不同于进行直接的字符对字符的比较所得到的结果。[[42]](#footnote-43)

由于这样的行为与既有的实践有巨大的差异，gawk 只在 POSIX 模式下（查看 ）实现了。下面的例子解释了在 en\_US.UTF-8 语言设置下的差异：

**$ gawk ’BEGIN { printf("ABC < abc = %s\n"，**

**> ("ABC"<"abc" ? "TRUE" : "FALSE")) }’**

**-| ABC < abc = TRUE**

**$ gawk --posix ’BEGIN { printf("ABC < abc = %s\n"，**

**> ("ABC"<"abc" ? "TRUE" : "FALSE")) }’**

**-| ABC < abc = FALSE**

### 6.3.3 布林表达式

布林表达式式由比较表达式或者是匹配表达式，利用操作 “or”（‘||’），“and”（‘&&’），还有“not”（‘!’）组合而成，并利用括号来控制表达式的嵌入。布林表达式的真值通过计算组成该表达式的直值组合而得。布林表达式也称为逻辑表达式。两者是一样的。

布林表达式可以用在所有的比较与匹配表达式可以使用的地方。可以用在 if，while，do 与 for 语言中（查看 ***，在第 146 页***）。当布林表达式的结果要存在变量中或者在算法表达式中使用，则其值为数值类型（真则为1，否则为0）

其次，每一个布林表达式都是有效的模式，因此你可以将其用在模式中，以控制规则的执行。布林操作符有如下这些：

***boolean1 && boolean2***

如果 ***boolean1*** 与 ***boolean2*** 都为 true，则返回 true。例如，下面的语句，如果输入记录中包含 ‘edu’ 与 ‘li’ 则打印它：

**if ($0 ~ /edu/ && $0 ~ /li/) print**

子表达式 boolean2 只有在 boolean1 求值为 true 时，才会进行求值。如果 boolean2 表达式包含有副作用的表达式，则会产生不同的结果。在 ***‘$0 ~ /foo/ && ($2 == bar++)’*** 例子中，变量 bar 在记录中不包含子串 ‘foo’ 时，不会进行递增。

***boolean1 || boolean2***

***boolean1*** 或者 ***boolean2*** 之一为 true 是返回 true。例如，下面的语句中，如果输入中包含 ‘edu’ 或者 ‘li’，则打印这些记录：

**if ($0 ~ /edu/ || $0 ~ /li/) print**

子表达式 boolean2 只有在 boolean1 的值为 false 才会进行求值。因此如果 boolean2 表达式有副作用，则会带来不同的结果。（所以，这里的测试并不是真正地测试记录中是否包含 ‘edu’ 与 ‘li’ 二者——只要匹配 ‘edu’，整个测试就成功。）

***! boolean***  如果 boolean 为 false，则为 true。下面的代码，当 HOME 环境变量不存在这种不正常事牛时，则打印 ‘no home’：

**BEGIN { if (! ("HOME" in ENVIRON))**

**print "no home!" }**

（in 操作符在 ， ***在第 168 页***，这里描述）

***‘&&’*** 与 ***‘||’*** 操作符因为它们的执行方式被称为短路操作符。对于完整表达式的求值，如果其结果只需要由其部分求值即可得到，则我们称这个表达式 ‘被短路了’ 。

如果语句由 ‘&&’ 或者 ‘||’ 结束，可以简单地在其后进行换行就可以进行续行。但是在其前面，如果你不加上一个反斜杠，都不能进行换行（查看 ， ***在第 31 页***）。

使用 ‘!’ 操作符的表达式的值要么是1，要么是0，则要视其应用所在的表达式的值。‘!’ 操作符在将标志变量的值从 false 转换为 true，再转回来的情况下很有用。例如，下面的程序就是一种打印特殊括号之间的行的方式：

**$1 == "START" { interested = ! interested; next }**

**interested { print }**

**$1 == "END" { interested = ! interested; next }**

变量 interested 与共他所有的 awk 变量一样，一开始被初始化为 0，其表示 false。当行发现了第一个域为 ‘START’ 的行后，interested 变量的值则利用用 ‘!’ 变为 true。下一个规则中，如果 interested 为 true，则打印这些行。当发现第一个域为 ‘END’的行后，则 interested 变量又会转回 false。[[43]](#footnote-44)

更通常的是，‘!’ 操作符是用在 if 与 while 语句中，在这些语句中，写否定逻辑含义更加明确：

**if (! some condition || some other condition) {**

**... do whatever processing ...**

**}**

提示：next 语句将在 ***，在第 153 页***，讨论。next 告诉 awk 跳过剩下的规则，并读取下一个记录，并从最开始来处理各个规则。写在这里的原因是避免打印括起文本的 ‘START’ 与 ‘END’ 行。

### 6.3.4 条件表达式

条件表达式是一种拥有三个操作数的特殊的表达式。该表达式可以让你使用一个表达式的值，即可以从其他两个表达式中选择一个。awk 中的条件表达式与 C语言中的表达式一样，如下所示：

**selector ? if-true-exp : if-false-exp**

这里有三个表达式。第一个表达是选择器，它总是首先被计算。如果结果为 ‘true’ （非零或者非空），则 if-true-exp 则接着被计算，其值将成为整个表达式的值。否则，if-false-exp 被计算，其值成为整个表达式的值。例如，下面的表达式产生 x 的绝对值：

**x >= 0 ? x : -x**

条件表达式的每一次计算，***if-true-exp*** 与 ***if-false-exp*** 只会有一个被使用，另外一个就被忽略了。如果表达式中有副作用，这就变得相当重要。例如，下面的条件表达式提取数组 a 或者 b 的元素 i，并递增 i：

**x == y ? a[i++] : b[i++]**

这保证了 i 只会被递增一次，当表达式被计算时，两个递增表达式只会执行一个。查看 *，* ***在第 166 页***，来获取更多的关于数组的信息。

作为一个小的 gawk 扩展，使用 ‘?:’ 的语句可以在两个符号中的任意字符处进行换行。但是，在这两个字符前进行换行，如果不加反斜杠是不行的（查看 ， ***在第 31 页***）。如何使用了 --posix 参数（查看 ， ***在第 35 页***），这个扩展也不被禁用了。

## 6.4 函数调用

函数是一个特殊计算过程的名字。这样你可以通过名字在程序的任何位置来请求执行。例如，函数 sqrt() 计算数值的平方根。

awk 中有固定的内建函数集合，它们可以在任意的 awk 程序中使用。sqrt() 就是其中之一。查看 ，***在第 182 页***，内建函数的列表以及其说明。另外，你也可以定义只有你的程序使用的函数。查看 ，***在第 204 页***，来获取自定义函数的方法。gawk 也可以让你使用 C 或者 C++ 来编写函数，并在你的程序中调用（查看 *，****在第 349 页***）。

函数的使用带有一个函数调用表达式，其包括函数名，后面直接跟着在括号中的参数列表。参数是为调用函数计算过程而提供的原始资料的表达式。当多于一个参数时，它们则通过逗号分隔。如果没有参数，则只要在函数名后写 ‘()’ 即可。下面的例子展现了带参与不带参的函数调用。

**sqrt(x^2 + y^2) one argument**

**atan2(y， x) two arguments**

**rand() no arguments**

注意：不要在函数名与开括号之间输出任何空格！用户自定义的函数名看起来就像是变量名——因此空格会导致表达式看起来是变量与括号中的表达式的连接操作。而对于内建的函数，在括号之前的空格是可选的，但是最好不要形成插入空格的习惯，以避免在调用用户自定义函数的时候犯错。

每个函数都有特定的参数。例如，sqrt() 函数必须用单参数调用，即要计算平方根的数值：

**sqrt(argument)**

一些内建的函数有一个或者多个的可选参数。如果没有提供那些参数，则函数会使用合适的默认值。查看 ，***在第 182 页***，来获取更多细节。如果在调用用户自定义的函数时省略了参数，则这些参数将会当成本变量（在定义函数时的括号中的变量——译者注）。这些本地变量当被使用时，如果需要的是一个字串，则其为空串，如果需要的数值，则其值为 0（查看 ， ***在第 204 页***）。

作为一个高级特性，gawk 提供了间接调用函数的方式，这是一种在运行时进行函数选择的方式，而不是在你写程序的时候就确定要调用的函数。我们后面再对这个进行讨论，可以查看 ， ***在第 213 页***。

如其他的表达式一样，函数调用也会生成一个值，通常称为返回值，这个值是根据函数的参数进行计算所得到的值。在这个例子中，‘sqrt(argument)’的返回值就是 argument 的平方根。下面的程序一行行地读取数值，然扣打印它们的平方根：

**$ awk ’{ print "The square root of"， $1， "is"， sqrt($1) }’**

**1**

**-| The square root of 1 is 1**

**3**

**-| The square root of 3 is 1.73205**

**5**

**-| The square root of 5 is 2.23607**

**Ctrl-d**

函数调用有副作用，例如将值赋给一个指定的变量或者执行 I/O 操作。这个程序展现了 match() 函数（查看 ， ***在第 184 页***）如何来更改变量 RSTART 与 RLENGTH：

**{**

**if (match($1， $2))**

**print RSTART， RLENGTH**

**else**

**print "no match"**

**}**

这是运行例子：

**$ awk -f matchit.awk**

**aaccdd c+**

**-| 3 2**

**foo bar**

**-| no match**

**abcdefg e**

**-| 5 1**

## 6.5 操作符优先级（操作符的嵌套）

操作符的优先级决定了不同的操作符紧接着出现在一个表达式中时如何结合。例如，‘\*’ 比 ‘+’ 有更高的优先级，所以 ‘a+b\*c’ 表示先将 b 与 c 相乘，然后再与 a 相加（如 ‘a+(b\*c)’）

正常的操作优先级可以通过使用括号来进行重新排序。一想到优先级规则就像是说，括号是该放在哪个位置。事实上，当操作符的组合不常出现时，使用括号总是很明智的做法。就算是很有经验的操作员，也偶尔忘记确切的规则，因此会犯错误。显式地使用括号则可以避免这样的错误。

当有相同优先级的操作符出现在一起时，先对最左边的操作符进行结合。对于赋值、条件以及幂乘它们则是以相反的顺序进行结合的。所以 ‘a-b+c’ 结合成 ‘(a-b)+c’， 而 ‘a=b=c’ 则是结合成 ‘a=(b=c)’。

一般情况下，前缀一元操作符对于优先级并不重要，因为只有一个解释它们的方式：内层的优先。所以，‘$++i’ 表示 ‘$(++i)’，而 ‘++$x’ 则表示 ‘++($x)’。但是，当另一个操作符跟着操作数后面，则一元操作符的优先级就重要起来了。‘$x^2’ 表示 ‘($x)^2’，但是 ‘-x^2’ 则为 ‘-(x^2)’，因为 ‘-’ 的优先级低于 ‘^’，而 ‘$’ 则有较高的优先级。另外，操作符也不能以一种违反优先级规则的方式进行组合，例如 ‘$$0++--’ 并不是一个有效的表达式，因为第一个 ‘$’ 有比 ‘++’ 更高的优先级，为了避免产生问题，表达式可以写成 ‘$($0++)--’。

下面的列表列出了 awk 的操作符，并按优先级的降序排序：

(. . .) 结合

$ 域引用

++ -- 递增，递减

^ \*\* 幂乘。这个操作符的结合由右往左。

+ - ! 一元加，一元减，以及逻辑 ‘not’。

\* / % 乘，除，取余。

+ - 加，减。

String concatenation

没有针对字串连接的特别符号。操作数简单地一个接一个输入（查看 ， ***在第 121 页***）。

< <= == != > >= >> | |&

关系与重定向。关系操作符与重定向有相同的优先级。如 ‘>’ 这样的字符，即为关系操作符，也可以作为重定向符。上下文环境用来区别使用的是哪种含义。

注意在 print 与 printf 语句中的 I/O 重定向，是语句级的，而不是表达式级的。重定向不会产生表达式，不可以成为别一个操作符的操作数。结果就是，不用括号而在一个低优先级的操作符附近使用重定向操作符是没有意义的。这样的组合（如 ‘printf foo > a ? b : c’ 会导致语法错误。正确地写法应该是 ‘printf foo > (a ? b : c)’。

~ !~ 匹配与不匹配

in 数组成员判定

&& 逻辑 ‘and’。

|| 逻辑 ‘or’。

?: 条件，这个操作符的结合是从右到左。

= += -= \*= /= %= ^= \*\*=

赋值操作符，这些操作符结合由右至左。

提示： ‘|&’、‘\*\*’ 与 ‘\*\*=’ 都没有由 POSIX 指定，为了移植性，请不要使用它们。

## 6.6 哪里造成差异

现代操作系统都有本地语言的概念：即告诉系统本地字符集与语言的一种方式。ISO C 标准定义了一种默认的 “C” 语言设置，这是一种很多 C 程序员所使用的一种典型环境。

以前，语言设置用来影响正则表达式的匹配，但是现在事情不再是这样的（查看 ， ***在第 414 页***）。

本地语言设置也会影响记录的分割。对于正常的 ‘RS="\n"’ 这种情况，语言设置大多是不相关的。对于其他单字符记录分隔符，在环境变量中设置 ‘LC\_ALL=C’ 将使你在读取记录时有更好的性能。否则，gawk 得调用几个函数，针对每一个输出字符进行调用，以查找记录的分隔符。

语言设置也会影响日期与时间的格式（查看 ， ***在第 197 页***）。例如，对于 2015 年 9月4号的一般缩写方式，在美国是“9/4/15”。但在欧洲的许多国家，则是 “4.9.15”。

根据 POSIX 标准，字串比较也会受到语言设置的影响（与正则表达式类型）。具体的细节在 ， ***在第 130 页***。

最后，语言设置也会影响 gawk 在分析输入数据时所使用的十进制小数点符号的值。关于这一点的细节则在 ， ***在第 117 页***。

## 6.7 总结

* 表达式是程序中基本的计算元素。它们由常量、变量、函数调用以及用操作符组合起来的各种值组成。
* awk 提供三种常量：数值常量，字串常量以及正则表达式常量。gawk 可以让你以八进制以及十六进制的方式来指定数值常量（相关的基数为8与16），也可以以十进制方式进行指定（基数为10）。在特定的上下文中，单独的正则表达式，如 /foo/ 与 ‘$0 ~ /foo/’ 有一样的含义。
* 变量在计算的过程中持有值。有一些内置变量来提供 awk 程序的相关信息，还有一些变量用来控制 awk 的行为。
* 在 awk 需要的时候，数值会自动地转换为字串，字串也会转换为数值。数值转换的方式就好像通过以 CONVFMT 格式来调用 sprintf 进行格式化一样。语言设置可以影响转换。
* awk 提供了常用的算术操作符（加、减、乘、除，取模），还有一元的加与减。同时也提供比较操作符，布林操作符，以及数组元素成员测试操作符和正则表达式匹配操作符。字串连接通过将通过简单地将表达式顺序排放来完成，而不需要显式的操作符。三元操作符则提供了在表达式中进行 ‘if-else’ 的测试。
* 赋值操作符为通用的算术操作提供了方便的缩略。
* 在 awk 中，如果非零或者非空，则被认为是 true，否则为 false。
* 变量的类型依赖于赋值，并且在生命期中可以变化。类型决定了它在比较中的行为（是以字串方式还是以数值方式进行）。
* 函数调用的返回值 可以做为一个更大的表达式的一部分。用于在调用函数时向函数传递的参数的表达式会在调用之前进行求值。awk 即提供了内置的函数，也提供用户自定义的函数，相关的细节在 *，* ***在第 182 页***。
* 操作符的优先级指定了操作执行的顺序，除非使用了括号来更改。awk 的操作符优先级与 C 中的优先级兼容。
* 语言设置可以影响 awk 程序输出数据的格式，偶尔也影响作为数据输入的格式。

# 第七章 模式、动作与变量

如我们已经看到的，每一个 awk 语句都由与模式及与其关联的动作组成。本章描述如何来创建模式与动作，包括你在动作中可以做的操作的类型，以及 awk 的预定义变量。

“模式-动作”规则以及可在动作中使用的语句组成了 awk 编程的核心。在一定意义上，到此为止所涉及的内容是建立程序的基础。目前可以弄些有用的时候了。

## 7.1 模式元素

在 awk 中的模式控制着规则的执行，当模式匹配当前的输入记录时，就执行它的规则。下面总结了 awk 模式的类型：

***/regular expression/***

正则表达式。当输入记录的文本适配于正则表达式则匹配。（查看 *，* ***在第 50 页***）

***expression***

单独的表达式。当这个表达式的值为非零（如果是数值的话）或者非空（如果是字串的话）则匹配。（查看 ， ***在第 139 页***）

***begpat***， ***endpat***

一对由逗号分隔的模式，指定了记录的范围。范围包括匹配 begpat 的开始的记录以及最后一个匹配 endpat 的记录（查看 ， ***在第 140 页***）。

***BEGIN***

***END*** 指定 awk 程序开始与清理行为的特殊模式。（查看 ， ***在第 141 页***）

***BEGINFILE***

***ENDFILE*** 针对每个文件为基础的在开始与清理中执行的动作的特殊模式。（查看 ， ***在第 143 页***）

***empty*** 空模式匹配每一条记录。（查看 ， ***在第 144 页***）

### 7.1.1 以正则表达式为模式

正则表达式是本书中出现的第一种模式。此类模式只是简单的处于模式规则中的正则表达工常量。它表示 ***‘$0 ~/pattern/’***。当输出记录匹配正则表达式是，则规则匹配。如：

**/foo|bar|baz/ { buzzwords++ }**

**END { print buzzwords， "buzzwords seen" }**

### 7.1.2 以表达式为模式

awk 表达是有效的 awk 模式。如果表达式的值是非零（如果结果为数值）或者非空（如果结果为字串）就表示模式是匹配的。表达式每次匹配输入记录的时候都需要重新计算一次。如果表达式使用的是域 ，如 $1，则它的值直接依赖于新的输入记录文本，不然，它就只依赖于 awk 程序到目前为止已经执行的结果。

使用在 ，***在第 127 页***，描述的比较操作符的比较表达式，是很常用的模式类型。正则表达式匹配与不匹配也是非常常用的表达式。***‘~’*** 与 ***‘!~’*** 操作符的左边的操作数是一个字串。右手的操作数要么是一个常量的正则表达式，要么是一个动态的正则表达式（查看 *，* ***在第 50 页***）。下面的例子中打印其第一个域为 ‘li’ 的所有记录的第二个域：

**$ awk ’$1 == "li" { print $2 }’ mail-list**

（这里没有输出，因为没有人的名字为 ‘li’）。与下面的正则表达式匹配相比，则会匹配任意的记录中第一个域中包含 ‘li’ 的记录：

**$ awk’$1 ~ /li/ { print $2 }’ mail-list**

**-| 555-5553**

**-| 555-6699**

正则表达式常量是表达式模式的一种特列。表达式 /li/ 中，如果 ‘li’ 出现在当前的记录中，则其值为1。因此，作为模式， /li/ 匹配任意的包含 ‘li’ 的记录。

布林表达式也是常用的模式。模式是否匹配输入记录，则依赖于其子表达式是否匹配。例如，下面的命令打印 mail-list 文件中，即包含 ‘edu’ 也包含 ‘li’ 的记录：

**$ awk ’/edu/ && /li/’ mail-list**

**-| Samuel 555-3430 samuel.lanceolis@shu.edu A**

下面的命令打印 mail-list 文件中包含 ‘edu’ 或者 ‘li’ （或者包含二者）的所有的记录。

**$ awk ’/edu/ || /li/’ mail-list**

**-| Amelia 555-5553 amelia.zodiacusque@gmail.com F**

**-| Broderick 555-0542 broderick.aliquotiens@yahoo.com R**

**-| Fabius 555-1234 fabius.undevicesimus@ucb.edu F**

**-| Julie 555-6699 julie.perscrutabor@skeeve.com F**

**-| Samuel 555-3430 samuel.lanceolis@shu.edu A**

**-| Jean-Paul 555-2127 jeanpaul.campanorum@nyu.edu R**

下面的命令打印 mail-list 文件中不包含串 ‘li’ 的所有记录：

**$ awk’! /li/’ mail-list**

**-| Anthony 555-3412 anthony.asserturo\hotmail.com A**

**-| Becky 555-7685 becky.algebrarum@gmail.com A**

**-| Bill 555-1675 bill.drowning\hotmail.com A**

**-| Camilla 555-2912 camilla.infusarum@skynet.be R**

**-| Fabius 555-1234 fabius.undevicesimus@ucb.edu F**

**-| Martin 555-6480 martin.codicibus\hotmail.com A**

**-| Jean-Paul 555-2127 jeanpaul.campanorum@nyu.edu R**

在模式中布林表达式操作符的子表达式可以是正则表达式常量、比较、或者任意的 awk 表达式。范围模式不是表达式，因此它们不可以出现在布林表达式中。而特殊模式 BEGIN，END，BEGINFILE 与 ENDFILE，它们不会匹配任何的输入记录，也不是表达式，所以也不能出现在布林模式中。

可以出现在模式中的不同的操作符优先级，在这里 ， ***在第 135 页***，进行讨论。

### 7.1.3 用模式指定记录范围

范围模式由两个逗号分隔的模式组成，以这样的方式 ***‘begpat， endpat’*** 出现。这个用来对连续的记录进行匹配。第一个模式 begpat 控制范围从何处开始，而 endpat 则控制在何处结束。例如下面的代码：

**awk ’$1 == "on"， $1 == "off"’ myfile**

打印 myfile 中 ‘on’ 与 ‘off’ 中的所有记录（包含）。

范围模式通过将输入记录与 begpat 开始。当记录匹配 begpat 时，范围就会被打开，同时范围模式也会匹配这个记录。只要范围模式保持打开状态，其就会匹配所读取的每个记录。同时，输入记录也会与 endpat 进行匹配，当匹配成功时，范围模式就会在下一条记录之前关闭。然后，范围模式又回头检查 begpat 与每条记录的匹配情况。

打开范围模式的记录与关闭范围模式的记录都匹配范围模式。如果你不想处理这些记录，你可以在规则中通过 if 语句来区分它们，并关注你所感兴趣的记录上。

也有可能在同一条记录上打开与关闭范围模式。如果记录满足两个条件，则动作就针对那条记录执行。例如，如果在两个相同的标志间有文本（如 ‘%’ 符号），且每一行包含一个这样的标志，则忽略这两个符号。第一种尝试组合范围模式的方式是描述分隔符，然后跟着 next 语句（还没有讨论到，请看 ， ***在第 153 页***）。这使得 awk 会跳过处理当前的记录而开始下一个输入记录。程序看起来如下：

**/^%$/，/^%$/ { next }**

**{ print }**

这个程序不会成功，因为范围模式在包含 ‘%’ 的同一行开启并关闭。为了完成这个任务，应该使用一个标志的方式来写程序：

**/^%$/ { skip = ! skip; next }**

**skip == 1 { next } # skip lines with ‘skip’ set**

在范围模式中，逗号（‘，’）在所有的操作符中具有最低的优先级（它将在最后进行求值）。所以，下面的程序试图用另一种更简单的测试来生成范围模式：

**echo Yes | awk ’/1/，/2/ || /Yes/’**

这个程序的意图本来是 ***‘(/1/，/2/) || /Yes/’***。但是 awk 将其解释成了 ***‘/1/， (/2/ || /Yes/)’***。没有办法改变或者解决这个问题，范围模式不可以与其他的模式进行组合。

**$ echo Yes | gawk ’(/1/，/2/) || /Yes/’**

**error gawk: cmd. line:1: (/1/，/2/) || /Yes/**

**error gawk: cmd. line:1: ^ syntax error**

一个有趣的点是，POSIX 允许你在范围模式的逗号后面进行换行，虽然这看起来不那么好看。

### 7.1.4 BEGIN 与 END 特殊模式

到目前为止所描述的模式都用来匹配输入记录。BEGIN 与 END 特殊模式则不同。它们为 awk 启动与清理操作。BEGIN 与 END 规则必须有动作。这些规则没有默认的动作因为在运行时没有当前记录。BEGIN 与 END 规则常常被那些老的 awk 程序员称为“BEGIN 与 END 块”。

#### 7.1.4.1 启动与清理动作

BEGIN 规则只执行一次，并且是在读取记录之前执行。相应地，END 规则也只执行一次，并且在所有的记录已经读取完成之后执行。例如：

**$ awk ’**

**> BEGIN { print "Analysis of \"li\"" }**

**> /li/ { ++n }**

**> END { print "\"li\" appears in"， n， "records." }’ mail-list**

**-| Analysis of "li"**

**-| "li" appears in 4 records.**

此程序查找在输入文件 mail-list 中，包含字串 ‘li’ 串的记录。BEGIN 规则打印报告的标题。没有必须通过 BEGIN 规则来初始化变量 n 为零，因为 awk 会自动处理（查看 ，***在第 116 页***）。第二个规则是当输入记录中包含 ‘li’ 时，就递增变量 n。END 规则在程序执行完成后，打印变量 n 的值。

BEGIN 与 END 特殊规则不可以用在范围规则中，也不可以与布林操作符一起使用（实际上，它们不可以与任何操作符一起使用）。一个 awk 程序可以有多个 BEGIN 与/或 END 规则。它们会按照出现的顺序被执行：所有的 BEGIN 规则在启动时执行，而所有的 END 规则在结束时执行。BEGIN 与 END 规则可以与其他的规则交叉混合。这样的特性是在 awk 的 1987 年的版本被加入，并加入到 POSIX 标准中。最开始（1978）的 awk 版本要求 BEGIN 规则要放在程序的开始处，END 规则则需要放在程序的结尾处，并且两者都只能出现一次。目前不再需要这样，但是按照这样的模板来写程序却是个好习惯，因为这样可以使程序组织得更好，并且增强可读性。

多个 BEGIN 与 END 规则在写库函数的时候非常有用，因为每个库文件都可以有它们自己的 BEGIN 与/或 END 规则来处理初始化与/或清理工作。在命令行中指定的库函数的名字，可以控制与其相关的 BEGIN 与 END 规则执行的顺序。因此，你要仔细地处理库文件中的这些规则，使得它们的执行顺序不会带来影响。

查看 ，***在第 35 页***，来获取更多的使用库函数的相关信息。查看 *，* ***在第 182 页***，来查看各种有用的库函数。

如果 awk 只有 BEGIN 规则而没有其他的规则，则程序在执行了 BEGIN 规则之后就退出了。[[44]](#footnote-45)但是，如果 END 规则存在，则输入会被读取，就算程序中没有其他的规则存在。因为如果在 END 规则中检查 FNR 与 NR 变量，因此这么做就是很有必要的。

#### 7.1.4.2 BEGIN 与 END 规则中的输入输出

在 BEGIN 与 END 规则中进行 I/O 操作时，要注意几点（其中的某些还比较微妙）。首先要关注的是在 BEGIN 中 $0 的值。由于 BEGIN 的执行在任何的输入之前，所以也就没有输入记录，也没有域。对于 $0 以及域的访问都将产生空串或者0值，这视上下文的情况。要使 $0 中有实值，可以以不带变量的形式执行 getline（查看 ***，在第 83 页***）。另一个方式就是给 $0 赋值。

第二点则跟第一点有点类似，但是是在另一个方向上。传统上，大部份是由于实现的问题，$0 与 NF 在 END 规则中都是未定义的。POSIX 标准指出 NF 在 END 规则中是可用的。它包含的是最后一行记录中的域数目。很大可能是由于疏忽，在标准中却没有说也要保留 $0 的值，尽管在逻辑上大家都认为是该保留下来的。事实上，所有的 BMW awk，mawk，以及 gawk 都在 END 规则中保留了 $0 的值。但是也要注意，一些其他的实现，与一些旧版本的 UNIX awk 版本却没有这么处理。

第三点，则是前面两点的延续。在 BEGIN 与 END 规则中的 print 的含义总是一样的：‘print $0’。如果 $0 的值是空串，则打印一个空的记录。很多使用 awk 很长时间的程序员在 BEGIN 与 END 规则中使用这种光头 print 来表示 ‘print ""’，这依赖于 $0 的值为空串。通常来说，在 BEGIN 规则中可以实现这样的意图，但是至少在 gawk 中，在 END 规则中这么写是一个坏主意。同时这么写也是一种很蹩脚的风格，因为如果确实需要输出一个空串，你就在显式地进行打印就可以了。

最后，next 与 nextfile 语句在 BEGIN 规则中是不允许使用的，因为隐式的 read-a-record-and-match-against-the-rules 循环都还没有开始呢。相似的，这样的语句在 END 规则中也是无效的，因为输入已经完成。（查看 ，***在第 153 页*** ，与 ，***在第 154 页***。）

### 7.1.5 BEGINFILE 与 ENDFILE 特殊模式

本节中描述的是 gawk 的特性。

有两种特殊的规则，即 BEGINFILE 与 ENDFILE，可以让你在命令行指定的文件处理过程中打入“勾子”。与 BEGIN 跟 END 类似（查看前章），程序中的所有的 BEGINFILE 规则都会安按照 gawk 读取的顺序合并起来，而所有 ENDFILE 规则也会被合并。

BEGINFILE 规则的执行体会在 gawk 读取文件的第一个记录前执行。FILENAME 将被设置成当前文件的名字，FNR 会被设置成0。

BEGINFILE 为你提供了完成两个任务的一个机会，这些任务如果没有这个机制是不可能实现的：

* 你可以测试文件是否可读。正常情况下，如果命令行上指定的文件无法打开读取是一个致使错误。但是，你可以绕过这样的致使错误，并移动到命令行中的下一个文件。你可以检查 ERRNO 的值不是一个空值，如果非空，就表示gawk没办法打开一个文件。此时，你的程序可以执行 nextfile 语句（查看 ，***在第 154 页***）。这使得 gawk 可以完整地跳过这个文件。否则，gawk 遇到致命错误时会退出。
* 如果你编写了修改记录处理方式的扩展（通过插入一个“输入分析器”，查看 ***，在第 359 页***），你就可以在这个点上，在处理文件之前调用它们。（这是一个非常高级的特性，当前只被 gawkextlib 项目使用。）

ENDFILE 规则会在 gawk 处理完一个文件的输入后执行。对于最后一个输入文件，它将会在调用 END 之前进行调用。ENDFILE 就算是针对空文件也会执行。

正常的情况下，当错误出现在正常的输入处理循环中的读取时，错误就会是致命的，如果提供了 ENDFILE 规则，这样的错误就不再是致命的，只是 ERRNO 被设置而已。这使得能够在 awk 程序一级捕获到处理 I/O 的错误。

next 语句（查看 ，***在第 153 页***）不准使用在 BEGINFILE 或者 ENDFILE 规则中。而 nextfile 只可以使用在 BEGINFILE 规则中，而不能使用在 ENDFILE 规则中。

getline 语句（查看 ，***在第 83 页***）在两个规则中都被限制使用：只有重定向形式的 getline 可以在其中使用。

BEGINFILE 与 ENDFILE 是 gawk 的扩展。在多数的 awk 实现中，或者 gawk 的兼容模式下（查看 ，***在第 35 页***），它们没有特殊含义。

### 7.1.7 空模式

一个空模式（如不存在模式）被视为匹配所有的输入记录。例如，下面的程序：

**awk ’{ print $1 }’ mail-list**

打印每一个记录的第一个域。

## 7.2 在程序中使用 Shell 变量

awk 程序常用于一个用 Shell 写成的大程序中的一个组件。例如，使用一个 Shell 变量来存放 awk 程序要搜索的模式。有两个方式来取得 Shell 变量的值，并插入到 awk 程序中。

一种通用的方式是使用 Shell 引号来在脚本中替代变量的值，由此来插入到程序中。例如，下面的程序：

**printf "Enter search pattern: "**

**read pattern**

**awk "/$pattern/ "’{ nmatches++ }**

**END { print nmatches， "found" }’ /path/to/data**

awk 程序将引号文本的片段视作为连接操作来形成程序。第一部分就被双引号引起来，这可以使得用 Shell 变量来替换引起来模式。第二部分是单引号的部分。

通过引号方式的变量替换虽然可以工作，但是有可能导致潜在的混乱。这需要对 Shell 的引号规则有比较好的理解（查看 ，***在第 24 页***），并且当读取程序，匹配引号的时候会比较麻烦。

更好的方法是使用 awk 的变量赋值特性（查看 ，***在第 116 页***）来将 Shell 变量的值赋给 awk 变量。然后使用动态正则表达式来匹配模式（查看 ，***在第 58 页***）。下面的代码显示使用这个技巧来重写上面的功能：

**printf "Enter search pattern: "**

**read pattern**

**awk -v pat="$pattern" ’$0 ~ pat { nmatches++ }**

**END { print nmatches， "found" }’ /path/to/data**

现在， awk 程序只需要单引号的串。赋值操作 ***‘-v pat="$pattern"’*** 还是需要双引号的，以防止在 $ ***pattern*** 的值中出现空格。awk 变量 pat 也可以使用名字 pattern，但是看起来就令人比较困惑得多。使用变量的方式也提供了更多的灵活性，因为变量可以在程序中的任何地方使用——用于打印、作为数组下标，或者其他的用途——并且不需要在使用的地方使用双引号。

## 7.3 动作

awk 程序或者脚本由一连串的规则与函数定义交叉形成。（函数会在后面论及。查看 ，***在第 204 页***）规则中包含了模式与动作，两者之一可以省略，但不可以都省略。动作的目的是告知 awk 在某个模式被找到后如何执行。因此，概括起来，awk 程序看起来都像这样：

**[pattern] { action }**

**pattern [{ action }]**

**. . .**

**function name(args) { ... }**

**. . .**

动作由一个或者多个 awk 语句组成，并且由花括号包起来（‘{}’）。每条语句都指定了一个要处理的过程。语句由换行符或者分号分隔。动作两边的花括号是必须使用的，就算只有一个语句，甚至没有语句也一样。但是，如果你整个地省略了动作，也要省略花括号。省略动作与 ‘***{ print $0 }***’ 等同。

**/foo/ { } match foo， do nothing — empty action**

**/foo/ match foo， print the record — omitted action**

awk 支持下面的语句类型：

***表达式***

调用函数或者赋值给变量（查看 *，* ***在第 182 页***）。这类型语句的执行就是简单地求值。如果表达式有副作用，这就非常有用（查看 ，***在第 122 页***）。

***控制语句***

指定 awk 程序的控制流。awk 语言可以让你写 C 风格的结构（if，for，while，以及 do），也有一些自己特殊的结构（查看 ***，见下方***）。

***复合语句***

包括在花括号中一个或者多个语句。复合语句用于在 if，while，do 或者 for 的语句体中，将几个语句顺序地放在一起。

***输入语句***

使用 getline 命令（查看 ，***在第 83 页***）。在 awk 中也提供 next 语句（查看 ，***在第 153 页***） 与 nextfile 语句（查看 ***，在第 154 页***）。

***输出语句***

如 print 与 printf，查看 *，* ***在第 94 页***。

***删除语句***

由于删除数组元素。查看 ，***在第 175 页***。

## 7.4 动作中的控制语句

如 if，while 等等的控制语句，用来控制 awk 程序的执行流。大多数的 awk 控制语句都是使用与 C 类似的模式。

所有的控制语句都以一个特殊关键词开始，如 if 与 while 等，以将它们与一般的表达式进行区分。很多的控制语句还包含其他的语句。例如，if 语句包含其他的语句，可能执行也可能不执行。被包含的语句称为相应语句的语句体。为了在语句体中包含多于一个语句，则将他们包含在花括号中，将通过换行或者分号来分隔它们。

### 7.4.1 if-else 语句

if-else 语句是 awk 的“决策-执行”语句，看起来如下：

**if (condition) then-body [else else-body]**

condition 是一个表达式，用来控制后面的语句的执行。如果 condition 为 true，则执行 then-body 的部分，否则执行 else-body 的部分。这个语句的 else 部分是可选的。condition 的值如果为0或者为空串，则被认为是 false，其他情况则为 true。参考如下：

**if (x % 2 == 0)**

**print "x is even"**

**else**

**print "x is odd"**

在这个例子中，如果表达式 ***‘x % 2 == 0’*** 为真（比如， x 的值是偶数，可被2整除），则第一个 print 语句被执行，否则，就执行第二个 print 语句。如果 else 关键词与 then-body 出现在一行中，并且 then-body 不是一个复合语句（没有被花括号包含），则必须要用分号将 then-boyd 的部分与 else 部分进行分隔。如，上面的例子代码可以写在这样：

**if (x % 2 == 0) print "x is even"; else**

**print "x is odd"**

如果没有 ‘;’ ，则 awk 无法解释上面的语句，因此会生成一个语法错误。在实际的程序中不要这么写，因此程序的人类阅读者有可能不会看到 else，因为它没出现在它自己所应在行的第一位置。

### 7.4.2 while 语句

在程序中，循环是程序的一部份，并且可被连续两次或者多次地执行。while 语句是 awk 中最简单的循环语句。只要条件为 true，就会一直重复执行。例如：

**while (condition)**

**body**

body 语句称为循环体，而 condition 是一个表达式，用来控制循环的执行。while 语句所做的第一件事情就是测试 condition 的值。如果为 true，则执行语句 body。完成后，condition 会被再次测试，如果依然为 true，则 body 会再次被执行。这个过程一真持续到 condition 不再为 true。如果 condition 一开始就为 false，而循环体就根本不会执行，awk 会继续执行循环后厕所语句。下面的例子打印每个记录的头三个域，一个一行：

**awk’**

**{**

**i = 1**

**while (i <= 3) {**

**print $i**

**i++**

**}**

**}’ inventory-shipped**

这个循环的循环休是一个包含在花括号中的复合语句，包含两个语句。循环按下面的方式执行：首先，i 的值被设置为1。然后 while 语句会测试 i 的值是否为小于等于3。当 i 为 1 时，这肯定为 true，因此第 i 个域就打印出来。然后执行 ‘i++’，递增 i 的值，并重复循环。当 i 为4 时，结束循环。

在条件与循环体之间的换行不是必须的，但是，使用换行会使得程序更加清晰，除非循环体是一个复合语句或者是非常简单。开花操号后面的换行开启复合语句，也不是必须的，但是程序不这么处理就比较难阅读。

### 7.4.3 do-while 语句

do 循环是 while 循环的变体。 do 循环中先会执行循环体一次，如果条件为 true，则会重复执行。它看起来如下：

**do**

**body**

**while (condition)**

就算是 condition 一开始为 false，循环体也会至少执行一次（并且只执行一次，除非执行体内将 condition 变成 true）。与 while 语句比较起来：

**while (condition)**

**body**

这个语句中，如果 condition 一开始为 false，则一次也不会被执行。下面是 do 语句的一个例子：

**{**

**i = 1**

**do {**

**print $0**

**i++**

**} while (i <= 10)**

**}**

这个程序打印记录10次。但是，这并不是一个非常理想的例子，因此在这种情况下， 普通的 while 循环就可以完成了。这个反映了一种实际使用经验：只有很少的情况下，才需要真正使用 do 语句。

### 7.4.4 for 语句

for 语句可以更方便地统计循环的迭代次数。for 语句的一般格式如下：

**for (initialization; condition; increment)**

**body**

***initialization***， ***condition*** 与 ***increment*** 的部分可以是任意的 awk 表达式，而 body 表示任意的 awk 语句。

for 语句以执行 initialization 开始，然后，只要 condition 为 true，则会重复执行 body 与 increment。典型的情况下， initialization 将某个变量设置成 0 或者 1，increment 将其增加 1，然后 condition 将其与所要迭代的次数进行比较。例如：

**awk ’**

**{**

**for (i = 1; i <= 3; i++)**

**print $i**

**}’ inventory-shipped**

这个例子中，会打印每个记录的前三个域，并且每个域一行。在 initialization 部分，如果不使用多个赋值语句，如 ***‘x = y = 0’***，不可能设置多于一个变量的值。这只有在所有的初始值都是一样的才有意义。（但是不可能在 for 循环前，用几个赋值语句来初始化其他变量的值。）

在 increment 部分也是一样。对其他的变量的递增，需要在循环结束时用独立的语句来进行递增。C 类型的复合表达式，使用 C 的逗号操作符，在这样的上下文中很有用，但是在 awk 中不被支持。

在多数情况下， increment 是一个递增表达式，就如前面的例子中的一样。但是却并不是必须的，它也可以是任意的表达式。例如，下面的语句打印 1 到 100 之间的 2 次幂的值：

**for (i = 1; i <= 100; i \*= 2)**

**print i**

如果在 for 后面括号中的表达式什么也不需要做，则可以省略它们。如 ***‘for (; x > 0;)’*** 与 ***‘while (x > 0)’*** 是等同的。如果 condition 被省略了，则其将被视作 true，这样可以有效地产生一个无限循环（不会终止的循环）。

在多数情况下，for 循环是 while 循环的一个简写方式，如下面所示：

**initialization**

**while (condition) {**

**body**

**increment**

**}**

唯一的例外是在循环中使用 continue 语句时（查看 ， ***在第 152 页***）。将 for 循环改为 while 循环，也能够改变循环内的 continue 语句的结果。

awk 语言除了 while 还要加上一个 for 循环，主要是因为 for 循环通常有更少的敲击输入，同时也是一种很自然的过程。统计迭代次数在循环中是一种很普遍的事情。将这种统计作为循环的一部份要比在循环内部来做这些事情要更容易。

这是另一种版本的 for 循环，for 迭代数组的所有下标：

**for (i in array)**

**do something with array[i]**

查看 ，***在第 170 页***，来获取更多的关于 for 循环的信息。

### 7.4.5 switch 语句

本节描述的是一个 gawk 专有的特性。如果 gawk 是在兼容模式（查看 ***，在第 35 页***）下，则这个特性就不可用。

switch 语句允许对表达式求值，并针对匹配的值来执行不同的语句。Case 语句用来顺序地检查所定义的匹配。如果没有合适的 case，如果提供了 default 段，则 default 段会被执行。

每个 case 中包含一个常量，可以是数值，字串或者正则表达式。switch 表达式被求值后，会按顺序对 case 中的学常量进行比较。常量的类型决定了比较的类型：数值的，还是字串的。一个正则表达式常量执行的是针对初始表达式的值的正则表达式匹配。一般地， switch 语句看起来如下：

**switch (expression) {**

**case value or regular expression:**

**case-body**

**default:**

**default-body**

**}**

switch 的控制流与 C 中的执行一致。只要与给定的 case 匹配了，case 语句体中的语句就会被执行，直到遇到 break，continue，next ，nextfile，或者是 exit，或者是 switch 语句本身的结束。

**while ((c = getopt(ARGC， ARGV， "aksx")) != -1) {**

**switch (c) {**

**case "a":**

**# report size of all files**

**all\_files = TRUE;**

**break**

**case "k":**

**BLOCK\_SIZE = 1024 # 1K block size**

**break**

**case "s":**

**# do sums only**

**sum\_only = TRUE**

**break**

**case "x":**

**# don’t cross filesystems**

**fts\_flags = or(fts\_flags， FTS\_XDEV)**

**break**

**case "?":**

**default:**

**usage()**

**break**

**}**

**}**

要注意的是，如果这里没有指定任何语句，则会停止执行匹配的 case 语句，则执行流程会落入下一个 case 中，直到执行中止。在这个例子中，“?” 的 case 中，会落入到 default case 中执行，并在其中调用 usage() 函数。其中所调用的 getopt () 函数在这里 ， ***在第 237 页***，描述。）

### 7.4.6 break 语句

break 语句跳出最包含该语句的最内层的 for，while，do 循环。下面的例子查找任意整数的最小因数，并判断是否为质数：

**# find smallest divisor of num**

**{**

**num = $1**

**for (divisor = 2; divisor \* divisor <= num; divisor++) {**

**if (num % divisor == 0)**

**break**

**}**

**if (num % divisor == 0)**

**printf "Smallest divisor of %d is %d\n"， num， divisor**

**else**

**printf "%d is prime\n"， num**

**}**

当第一个 if 语句中的余数是 0 时，awk 就直接跳出循环。这表示 awk 会执行直接跟在循环后面的语句。（这与 exit 非常不同，exit 会停止跟个 awk 程序，查看 ， ***在第 154 页*** 。）

下面的程序解释了在 for 或者 while 循环语句中的 condition，如何用 if 中的 break 来代替：

**# find smallest divisor of num**

**{**

**num = $1**

**for (divisor = 2; ; divisor++) {**

**if (num % divisor == 0) {**

**printf "Smallest divisor of %d is %d\n"， num， divisor**

**break**

**}**

**if (divisor \* divisor > num) {**

**printf "%d is prime\n"， num**

**break**

**}**

**}**

**}**

break 语句也用来中断 switch 中的语句。这在 ， ***在第 150 页***，进行了描述。

在循环体或者 switch 外面使用 break 没有意义。但是，虽然没有在文档中记录下来，awk 的历史版本中将循环外的 break 语句，当成是一个 next 语句（查看 ， ***在第 153 页***）。近来的 BWK awk 版本不再允许这样的用法，gawk 也不允许。

### 7.4.7 continue 语句

与 break 类型，continue 语句只要 for，while，以及 do 循环中使用，它路过循环体后面的部分，使得下一个循环立即开始。与 break 相比，它只是跳出循环。

for 循环中的 continue 语句，引导 awk 路过循环体后面的部分，并执行 for 语句中的递增表达式（increment-expression）。下面的程序解释了这样的情形：

**BEGIN {**

**for (x = 0; x <= 20; x++) {**

**if (x == 5)**

**continue**

**printf "%d "， x**

**}**

**print ""**

**}**

这个程序打印 0 到 20 之间的数，除了 5 外，因为在这个数值时 printf 被路过了。因为 ***‘x++’*** 并没有被路过，因此，x 的值不会卡在 5 上。下面的 while 循环与前面的 for 循环相比：

**BEGIN {**

**x = 0**

**while (x <= 20) {**

**if (x == 5)**

**continue**

**printf "%d "， x**

**x++**

**}**

**print ""**

**}**

这个程序中，只要 x 达到5 就会成为一个死循环，因此递增 （‘x++’）语句永远不会执行。

与 switch 语句对应，continue 语句并没有特别的含义，在循环体之外也没有什么特别的含义。awk 的历史版本将循环体之外的 continue 与循环体之外的的 break 一样对待：就像它是一个 next 语句一样（查看 ， ***见下方***）。BWK awk 的后面的版本不再这么处理，gawk 也不这么做。

### 7.4.8 next 语句

next 语句强制 awk 立即停止当前记录的处理并处理下一条记录。这表示针对当前的记录不再进一步进行规则的匹配，当前规则的余下部分也不被执行。

与 getline 函数（查看 ， ***在第 83 页***）进行对比。getline 也会导致 awk 立即读取下一条记录，但是并没有改变控制流（即，当前的动作会在新的记录上执行）。

在最高级上，awk 程序是这样的一个循环，即读取一条记录，然后与每一个模式进行匹配。如果你把这样的一个循环看成是一个 for 语句，其循环体包含了这些规则，然后，next 语句则与 continue 语句类似。它会跳过这个隐式循环的后面部分，然后执行递增（也就是读取下一条记录）。

例如，假设一个 awk 程序处理的记录只有4个域，并且当输入不规范时时不会导致失败。为了避免后面程序的复杂性，用下面的方式，在开头处这一个这样的“排除”代码：

**NF != 4 {**

**printf("%s:%d: skipped: NF != 4\n"， FILENAME， FNR) >"/dev/stderr"**

**next**

**}**

由于有一 next 语句，程序后面的规则就不会再看到那些不规范的记录。错误的信息会被重定向到标准错误输出流上，正如错误信息应该这么被处处理一样。要取得更多的信息，请查看 ， ***在第 106 页***。

如果 next 语句导致了输入文件的结束，则在 END 规则里的代码就会被执行，查看 ， ***在第 141 页***。

next 语句不可以使用在 BEGINFILE 与 ENDFILE 规则中，查看 ， ***在第 143 页***。

根据 POSIX 标准，如果 next 语句用在 BEGIN 与 END 规则中，其行为是未定义的。gawk 将其视为语法错误。尽管 POSIX 没有禁止，但大多数的 awk 实现不准在函数体内（查看 ， ***在第 204 页***）使用 next 语句。与共他的处的 next 语句一样，在函数内的 next 语句也是读取下一条记录然后从程序的第一个规则开始处理。

### 7.4.9 nextfile 语句

nextfile 与 next 语句类似。但是不是放弃处理当前记录，nextfile 则是通知 awk 停止处理当前的数据文件。

随着 nextfile 的执行，FILENAME 也会更新以反映命令行列表中的下一个数据文件名。FNR 则被重置为 0，处理流程回到程序中的第一个规则中。如果 nextfile 语句导致了输入的结束，则在 END 规则中的代码就会被执行。一个例外就是这个 nextfile 是在执行 END 规则中被调用的，在这种情况下，会导致程序退出。查看 ， ***在第 141 页***。

当有很多的数据文件要处理，但是不需要处理每个文件中的每个记录的情况下非常有用。如果没有 nextfile，为了要处理下一个数据文件，程序就得断续输入那些不需要的记录。nextfile 语句以一种更为高效的方式来完成这样的目标。

在 gawk 中，nextfile 的执行会导致一些其他的事情发生：如果 gawk 当前没有执行 END 或者 ENDFILE 规则，则会导致所有 ENDFILE 被执行，ARGIND 会被递增，所有的 BEGINFILE 会被执行。（ARGIND 还没有介绍。查看 ， ***在第 155 页***。）

对于 gawk，在 BEGINFILE 规则中跳过一个会导致 gawk 因为致命错误而退出的文件也非常有用。在这种情况下， ENDFILE 不会被执行。

尽管有可能看到用 ***‘close(FILENAME)’*** 来完成如 nextfile 一样的功能，但是事实却不是这样的。close() 保留用来关闭打开用于重定向的文件、管道以及并程。它与主过程没有关系，它们不处理 awk ARGV 中的文件。

提示：许多年了，nextfile 一直是一个扩展。在2012年9月，它报授受并包含在 POSIX 标准中，查看 Austin Group 的网站。

BWK awk 与 mawk 的最新版本也支持 nextfile。但是，它们不允许在函数体（查看 ， ***在第 204 页***）中使用 nextfile 语句。gawk 则允许。在函数体内的 nextfile 会读取下一个记录，然后从程序的第一条规则开始处理，这与其他处的 nextfile 语句一样。

### 7.4.10 exit 语句

exit 语句会导致 awk 立即停止执行当前的规则，并停止处理输入，所有剩下的输入都将被忽略。exit 的写法如下：

**exit [return code]**

当在 BEGIN 规则中执行 exit 语句时，程序会立即停止一切处理。也没有输入记录被读取，如果提供了 END 规则，会作为执行 exit 语句的一部份，而被执行（查看 ， ***在第 141 页***）。如果在 END 规则中使用 exit，会导致程序立即停止。

exit 语句如果不是 BEGIN 或者 END 规则一部份，则会停止任何后续的针对当前记录的自动规则的执行，并不再读取剩下的输入记录，然后如果有 END 规则，则执行。gawk 也会跳过 ENDFILE 规则，因此不会执行。

在这样的情况下，如果你不想 END 规则被执行，可以将某个变量的值设置成非零值，然后在 END 规则中的 exit 语句前，判断这个变量的值。查看 ，***在第 224 页***，来参考这样的具体例子。

如果给 exit 语句提供了参数，它的值将作为 awk 进程的退出状态码。如果没有提供，awk 则会以“成功”的方式退出。在第一次调用 exit 时提供了参数，而第二次从 END 规则中调用，但是没有提供参数的情况下，awk 会使用之前提供的退出值。查看 ，***在第 46 页***，来取得更多信息。

例如，如果一个错误条件出现，很难或者无法处理。一般的作法是程序会以一个非零状态退出。awk 程序通过带非零值的 exit 语句来处理，如下面的例子所示：

**BEGIN {**

**if (("date" | getline date\_now) <= 0) {**

**print "Can’t get system date">"/dev/stderr"**

**exit 1**

**}**

**print "current date is"， date\_now**

**close("date")**

**}**

提示：为了完全的可移植性，exit 的值应该在 0 于是 126 （含）之间。负值与 127 或者更大的值，在不同的平台之间可能不会产生一致的结果。

## 7.5 预定义变量

大多数的 awk 变量是我们自己使用，如果他们没有被赋值，则不会改变，并且如果你不检查它们，对你也没有什么影响。但是，有一些的 awk 变量有一些特殊的内置含义。awk 会自动检查它们当中的一些值，因此使你可以告诉 awk 执行一些特定的事情。有一些是被 awk 自动设置的，所以它们为你的程序带来一些 awk 执行时的内部信息。

本节记录了所有的 gawk 预定义的变量，它们中的大多数也记录在了描述它所应用的场景中的章节中。

### 7.5.1 控制 *awk* 的内置变量

下面是按字母顺序列出的变量，你可更改它们的值来控制 awk 所执行的特定事情。

这些变量如果是 gawk 特有的，则会被标志一个井号（‘#’）。这些变量是 gawk 的扩展。在一些其他的 awk 实现中，或者在 gawk 的兼容模式（查看 ， ***在第 35 页***）下，它们没有特殊含义。（如果变量有什么例外，则会在变量的说明中列出。）

BINMODE # 在非 POSIX 系统上，这个变量指定了所有I/O所使用的二进制模式。数值1，2，3分别指定为输入文件，输出文件与所有文件相应地要使用二进制I/O模式。小于0的值，则当成0对待，而大于3的值，则会被当为3对待。相应的，字串“r”与“w”分别指定输入输出 要使用二进制I/O。字串“rw”或者“wr”表示所有的文件使用二进制I/O。其他的字串值则被当成“rw”，但是 gawk 会产生一告警。BINMODE 在 ，***在第 428 页***，这里有更详细的说明，mawk（查看 ***，在第 435 页***）也支持这个变量，但是只使用数值。

CONVFMT 用于控制数值转换成字串的格式（查看 ， ***在第 117 页***）。在效果上，它的工作原理是将其作为 sprintf() 函数的第一个参数（查看 ， ***在第 184 页***）。它的默认值是“%.6g”。CONVFMT 由 POSIX 标准引入。

FIELDWIDTHS #

一个由空格分隔的列，由以告诉 gawk 如何用固定的列边界来分割输入。给 FIELDWIDTHS 赋值，会覆盖用 FS 与 FPAT 进行域分割的方式。查看 ，***在第 77 页***，来获取更多信息。

FPAT # 告诉 gawk 域的分割是由匹配正则表达式（一个字串）的文本进行的。对 PFAT 的赋值会覆盖 FS 与 FIELDWIDTHS 的域分割方式。查看 ，***在第 79 页***，来获取更多的信息。

FS 输入的域分隔符（查看 ， ***在第 71 页***）。它的值是一个单独的字符或者是一个多字符的正则表达式，该表达式匹配输入记录中域之间的分隔内容。如果它的值为空串（“”），则输入记录中的每个字符都是分隔符。（这个行为是 gawk 扩展。POSIX awk 没有指定当 FS 的值为窗口时的行为。因此，其他版本的 awk 也有可能将 “” 特殊对待。）

它的默认值是 “ ”，即只包含一个空格的字串。这是一个特殊的例外，这个值表示所有空格，TABs 与/或换行序列，都是一个分隔符。[[45]](#footnote-46)同时，在记录前后的空格，TABs 与换行符也被忽略掉。

你也可以在命令行中用 -F 来指定 FS 的值：

***awk* -F， 'program' input-files**

如果 gawk 使用了 FIELDWIDTHS 或者 FPAT 用来进行域分割，对 FS 的赋值会导致 gawk 退回到常规的，基于 FS 的域分割方式。一个简单的方法就是就是使用 ‘***FS = FS***’，并加上一些解释型注释。

IGNORECASE #

如果 IGNORECASE 是一个非零值或者非空串，则字串的比较与所有的正则表达式匹配都是大小写无关的。这应用于用 ***‘~’*** 与 ***‘!~’*** 进行的正则匹配， 也用于 ***gensub()***， ***gsub()***， ***index()***， ***match()***， ***patsplit()***，***split()***， 与 ***sub()*** 等函数，还有以 RS 进行分割的记录，用 FS 与 PFAT 进行分割的域中。但是，IGNORECASE 的值不会影响用下标索引数组元素，也不会影响当使用半字符的分割符时的域分割。查看 ， ***在第 60 页***。

LINT # 当这个变量为 true（非零或者非空），gawk 表现得就像在命令行参数中指定了 --lint 参数一样效果（查看 ， ***在第 35 页***）。当它的值为“fatal”时，lint 告警会成为致命（fatal）错误。如果值是“invalid”时，则只有真的是无效的事情的告警才会生成。（这还没有完全实现。）任意的其他 true 值都打印非致命告警。给 LINT 变量赋值为 false，会关闭 lint 告警。

此变量是一个 gawk 扩展。在其他的 awk 实现中则并不特殊。与其他的特殊变量不同的是，改变 LINT 变量不会影响 lint 告警的产生，就算 gawk 是处于兼容模式。虽然 --lint 与 --traditional 选项单独地控制 gawk 行为的不同方面，但是程序执行中的 lint 告警的控制是与 awk 被执行的方式是无关的。

OFMT 控制 print 语句中，数字到字串的转换方式的字串（查看 ， ***在第 117 页***）。它的工作方式是将其作为传递给 sprintf() 函数的第一个参数（查看 ， ***在第 184 页***）。它的默认值是 “%.6g”。早期的 awk 版本使用 OFMT 来指定将数值转换为字串的格式的一般表达式，但是这个现在由 CONVFMT 来完成。

OFS 输出域分隔符（查看 ， ***在第 96 页***）。它被输出在通过 print 语句打印的域之间。它的默认值是“ ”，即一个空格的字串。

ORS 输出记录的分隔符。它会被输出在每个 print 语句的后面。其默认值是 “\n”，即换行符。（查看 .***在第 96 页***）

PREC # 任意精度浮点数的工作精度，默认为53位（查看 ，***在第 343 页***）。

ROUNDMODE #

用于任意精度数值算术计算的舍入舍出模式，默认为 “N”（即 IEEE 754 标准中的 roundTiesToEven，查看 ，***在第 344 页***）。

RS 输入记录的分隔符。它的默认值是包含一个换行符的字串，表示输入的记录由一行的文本组成。它的值也可以是空串，在这种情况下，记录由于一连串的空行分隔。如果它是一个正则表达式，则记录是由匹配正则表达式的文本进行分割。（查看 ， ***在第 63 页***）RS 可以使用正则表达式的能力是 gawk 的扩展。在多数的其他 awk 实现中，或者 gawk 是处于兼容模式下（查看 ， ***在第 35 页***），只有 RS 中的第一个字符被用来进行分割记录。

SUBSEP 下标分隔符。它的默认值是 “\034”，并用来作为多维数组下标索引的一部分。因此，表达式 ***‘foo["A"， "B"]’*** 实际上是访问 foo["A\034B"]（查看 ， ***在第 176 页***）。

TEXTDOMAIN #

用于在 awk 层中的程序的国际化。它设置默认的文本域，其用来在源文本中用来标识字串常量，也用于 ***dcgettext()***， ***dcngettext()***， 与 ***bindtextdomain()*** 函数（查看 ***， 在第 182 页***）。它的默认值为 “messages”。

### 7.5.2 传递信息的内置变量

下面的变量，以字母顺序排列，awk 在特定的情况下会自动设置它的值，以向程序传递信息。

如果是 gawk 特有的变量，则被用井号标识（“#”）。这些变量是 gawk 的扩展。对于其他的 awk 实现，或者在兼容模式下的 gawk（查看 ， ***在第 35 页***），则它们没有特别含义：

ARGC， ARGV

命令行中的参数被存在了名为 ARGV 的数组中，并可以被 awk 程序使用。ARGC 是命令行中参数的个数。查看 ， ***在第 41 页***。与大多数的 awk 数组不同，ARGV 的下标从0到ARGC - 1。在下面的例子中：

**$ *awk* ’BEGIN {**

**> for (i = 0; i < ARGC; i++)**

**> print ARGV[i]**

**> }’ inventory-shipped mail-list**

**-|*awk***

**-| inventory-shipped**

**-| mail-list**

ARGV[0] 包含 ‘awk’，ARGV[1] 包含 ***‘inventory-shipped’***，ARGV[2] 包含 ***‘mail-list’***。ARGC 的值为3，比 ARGV 数组最后一个元素的下标大1，因为元素是从0开始编号的。ARGC 与 ARGV 的名字，与其下标为从0到 ARGC - 1 都是从 C 语言中访问命令行参数的方法继承而来。

ARGV[0] 的值会因为系统不同可能不同。同时，你应该注意程序文本与 awk 的命令行选项都没有包含在 ARGV 中。查看 ，***在第 163 页***，来获取使用这些 awk 变量的信息。

ARGIND # 当前正在被处理的文件在 ARGV 中的索引。每次 gawk 打开一个文件进行处理，它就会设置 ARGIND 的值文件名在 ARGV 中的索引。当 gawk 处理输入文件时，***‘FILENAME == ARGV[ARGIND]’*** 的值总是为 true。

这个变量在处理文件时很有用，它可以让你知道数据文件中已经处理了多少，以及用来区分在命令行中两个连续的同名文件。

当你在 awk 程序中改变 ARGIND 变量的值时，gawk 会在打开下一个文件时，自动设置它为新的值。

ENVIRON 一个包含环境变量以其值的关联数组。数组的索引是环境变量名，元素则是特定环境变量的值。例如 ENVIRON["HOME"] 的值可以是 “/home/arnold”。改变这个数组不会影响传递到任意的通过 system() 函数或者重定向启动的程序的环境变量。（在后续的 gawk 版本，有可能会有影响。）

有一些操作系统没有环境变量。在这样的系统中，ENVIRON 数组为空（除了 ENVIRON["AWKPATH"] 与 ENVIRON["AWKLIBPATH"]，查看 ，***在第 43 页***，与 ， ***在第 44 页***）。

ERRNO # 如果在为 getline 进行重定向、读取时，或者进行 close() 操作中，发生了系统错误，则 ERRNO 会包含一个描述错误的字串。

另外，gawk 会在每一个命令行中的输入文件的操作前清空 ERRNO。这样可以在 BEGINFILE 模式中检查文件的可读性（查看 ， ***在第 143 页***）。

其他情况，ERRNO 的作用就类似于 C 变量 errno。除了则才所提到的几个种情况，gawk 绝不会清除它（即将其设置为0，或者“”）。因此，只有在 I/O 操作返回一个失败的值的情况下，你才可寄希望于它的值有意义，例如 getline 返回 -1。当然，你也可以在执行 I/O 操作之前自己清理它。

FILENAME 当前输入文件的名字。如果在命令行中没有输入文件时，awk 会从标准输入中读取数据，同时FILENAME 的值会设置成 “-”。FILENAME 每次读取新文件的时候都会改变（查看 ， ***在第 63 页***）。在 BEGIN 规则内，FILENAME 的值为 “”，因为这时还没有输入文件被处理。[[46]](#footnote-47)但是，在 BEGIN 规则中使用 getline （查看 ， ***在第 83 页***）可以使 FILENAME 有一个值。

FNR 当前文件的当前记录数。awk 每读一个新记录的时候，会递增 FNR（查看 ， ***在第 63 页***）。awk 每次开始读取一个新文件的时候，会将 FNR 设置为0。

NF 当前输入记录的域数值。 NF 每读一个新记录，并且域已经生成的时候，就会被重置，或者是 $0 改变也会重置（查看 ， ***在第 67 页***）。

与在本节中描述的其他大多数变量不同，给 NF 赋值对 awk 的内部工作有一些潜在的效果。具体地说，给 NF 赋值可能创建新的域，或者是从当前的记录中删除域。查看 ， ***在第 69 页***。

FUNCTAB # 一个数组，其索引以及对应的值全部都是内建的，用户定义的，以及程序中扩展的函数。

提示：在 FUNCTAB 上尝试用 delete 语句会导致致使错误。而对其元素进行赋值，也会导致命令错误。

NR 从程序执行开始到现在为止，awk 已经处理的记录的个数（查看 ， ***在第 63 页***）。awk 每读一个新记录就会对其进行递增。

PROCINFO #

这个数组的元素为运行中的程序的信息提供了访问手段。下面的元素（按字母顺序）保证可用：

PROCINFO["egid"]

系统调用 getegid() 的值。

PROCINFO["euid"]

系统调用 geteuid() 的值。

PROCINFO["FS"]

如果域分割方式是 FS，则其值为 “FS”，如果是 FIELDWIDTHS 分割方式起作用，则其值为 “FIELDWIDTHS”，如果是 PFAT 方式进行分割，则其值为 “FPAT”。

PROCINFO["identifiers"]

一个子数组，由在 awk 程序文件中所使用的标识符名字进行索引。一个标识符简单地说就是变量的名字（变量是一个标量或者是一个数组），也可以是内建函数，用户自定义函数，或者是函数扩展。对每一个标识符，元素的值可能是下面其中之一：

"array"

标识符是一个数组。

"builtin"

标识符是一个内建函数。

"extension"

标识符是一个函数扩展，并通过 @load 或者 -l 进行装载。

"scalar"

标识符是一个标量。

"untyped"

标识符是无类型的（可用于标量或者是数组，gawk 还不知道而已）。

"user"

标识符是一个用户自定义的函数。其值显示了 gawk 完成程序的分析后，对所了解到的标识符的情况，它们的值在程序的运行过程中不能更新。

PROCINFO["gid"]

系统调用 getgid() 的值。

PROCINFO["pgrpid"]

当前进程的组 ID。

PROCINFO["pid"]

当前进程的进程 ID。

PROCINFO["ppid"]

当前进程的父进程的进程 ID。

PROCINFO["sorted\_in"]

如果这个元素在 PROCINFO 中存在，则它的值控制针对数组索引在 ***‘for (indx in array)’*** 循环中的处理顺序。这是一个高级特性，因此我们后再详细讨论。查看 ， ***在第 170 页***。

PROCINFO["strftime"]

strftime() 时间函数的默认格式串。对这个值进行赋值可以改变默认串，查看 ， ***在第 197 页***。

PROCINFO["uid"]

系统调用 getuid() 的值。

PROCINFO["version"]

gawk 的版本号。

下面在数组中的额外元素，如果你的 gawk 版本支持任意精度的算术运算（查看 ***，在第 336 页***），则它们会在数组中存在，并提供关于 MPFR 以及 GMP 库的相关信息。

PROCINFO["mpfr\_version"]

GNU MPFR 库的版本。

PROCINFO["gmp\_version"]

GNU MP 库的版本。

PROCINFO["prec\_max"]

MPFR 支持的最大精度。

PROCINFO["prec\_min"]

MPFR 所要求的最小精度。

下面的附加元素，如果你的 gawk 版本支持动态装载扩展函数（查看 ***，在第 349 页***），则会在数组中存在，并向你提供这些扩展 API 的版本信息：

PROCINFO["api\_major"]

扩展 API 的主版本。

PROCINFO["api\_minor"]

扩展API 的次版本。

在某些系统中，有可能存在 “group1” 到值为 N 的 “groupN”的元素存在。N 是进程所有的补充组的数值。使用 in 操作符来测试这些元素（查看 ***在第 168 页***）。

PROCINFO 数组还有下面的用途：

* 用来提供从任意打开文件、管道或者并程中读取时的超时时间。查看 ，***在第 90 页***，来取得更多信息。
* 可以用来使并程之间通过 pseduo-ttys 上进行双向通信，而不是通过双向的管道，这个在 ， ***在第 304 页***，进行更详细的讨论。

RLENGTH ***match()*** 函数匹配的子串的长度（查看 ***在第 184 页***）。RLENGTH 在调用 match() 函数时被设置。它的值是匹配串的长度，如果没有匹配找到，则值为 -1。

RSTART 调用 match() 函数所匹配的子串的开始字符的下标（查看 ， ***在第 184 页***）。 RSTART 在调用 match() 函数的时候被设置。它的值是匹配串的在串中的位置，如果没有匹配串，则其值为0。

RT # 输入文件中匹配由 RS，即记录分隔符，表示的串。每次读取一个记录时，则其值会被设置。

SYMTAB # 在程序中由名字进行索引的所有全局定义的变量与数组的数组。SYMTAB 使得 gawk 的符号表可由 awk 程序员看到。它在 gawk 分析程序的时候建立，并在程序开始执行前完成。

数组可以用来间接地读取与写入一个数量的值：

**foo = 5**

**SYMTAB["foo"] = 4**

**print foo # prints 4**

isarray() 函数（查看 ， ***在第 203 页***）可以用来测试 SYMTAB 中的元素是否为数组。类似地，你也不可以对 SYMTAB 中的元素使用 delete 语句。

你也可以使用没有定义的标识符作为 SYMTAB 的下标：

**SYMTAB["xxx"] = 5**

**print SYMTAB["xxx"]**

下如下情况，其运行与预期一致：SYMTAB 就是一个普通的数组。唯一的不同就是你不可以删除 SYMTAB["xxx"]（delete SYMTAB["xxx"]）。

SYMTAB 比初看起来要有趣得多。Andrew Schorr 指出来，可以用其作为 awk 的数组指针。看看他的例子：

**# Indirect multiply of any variable by amount， return result**

**function multiply(variable， amount)**

**{**

**return SYMTAB[variable] \*= amount**

**}**

提示：为了避免严重的长时间旅行悖论，[[47]](#footnote-48)不管是 FUNCTAB 还是 SYMTAB 都不能作为 SYMTAB 数组中的元素。

|  |
| --- |
| 改变 NR 与 FNR  每次读取一个记录时， awk 就是递增 NR 与 FNR 的值，而不是直接将它们设置成已读记录的绝对值。这意味着，程序可以改变这些变量的值，并且在每个记录间都将被递增。下面的例子可以说明：  **$ echo ’1**  **> 2**  **> 3**  **> 4’ | *awk* ’NR == 2 { NR = 17 }**  **> { print NR }’**  **-| 1**  **-| 17**  **-| 18**  **-| 19**  在 FNR 被加入到 awk 语言之前（查看 ， ***在第 404 页***），很多的 awk 程序使用这个特性来跟踪文件中的记录数，这只需要在 FILENAME 变化时，将 NR 的值设置成 0 即可。 |

### 7.5.3 使用 ARGC 与 ARGV

在 ，***在第 158 页***，中有提供下面的程序来描述包含在 ARGC 与 ARGV 中的信息：

**$ awk ’BEGIN {**

**> for (i = 0; i < ARGC; i++)**

**> print ARGV[i]**

**> }’ inventory-shipped mail-list**

**-|awk**

**-| inventory-shipped**

**-| mail-list**

在这个例子中，ARGV[0] 包含 ***‘awk’***， ARGV[1] 包含 ***‘inventory-shipped’***， 而 ARGV[2] 包含 ***‘mail-list’***。注意 awk 程序并没有在 ARGV 中。其他的命令行选项，以及它们的参数都没有进入到这个数组中。这也包括通过 -v 参数（查看 ， ***在第 35 页***）进行变量赋值的部分。在命令行中的正常的变量赋值被当成参数对待，并且会在 ARGV 中出现。有下面的名为 ***showargs.awk*** 的文件：

**BEGIN {**

**printf "A=%d， B=%d\n"， A， B**

**for (i = 0; i < ARGC; i++)**

**printf "\tARGV[%d] = %s\n"， i， ARGV[i]**

**}**

**END { printf "A=%d， B=%d\n"， A， B }**

运行程序会产生如下输出：

**$ awk -v A=1 -f showargs.awk B=2 /dev/null**

**-| A=1， B=0**

**-| ARGV[0] = awk**

**-| ARGV[1] = B=2**

**-| ARGV[2] = /dev/null**

**-| A=1， B=2**

程序可以改变 ARGC 与 ARGV 的元素值。当每次 awk 达到文件尾时，它会使用 ARGV 的下一个元素作为输入文件。通过在那个位置存入一个不同的串，程序就可以改变要读取的文件。使用 “-” 来表示标准输入。存储额外的元素并递增 ARGC 的值，可以处理额外的文件。

如果 ARGC 的值被递减了，则会从将输入文件从列表的尾部删除。通过在其他地方记录 ARGC 的旧值，程序可以将取消的参数当做其他东西来对待，而不是文件名。

为了从列表的中间将文件排除，可以将空串存到 ARGV 中来代替文件名。作用一个特殊的特性，awk 会忽略那些用空串替换掉的文件。另一个选择就是用 delete 语句来移除 ARGV 中的元素（查看 ， ***在第 175 页***）。

所有这些动作一般都要在 BEGIN 规则中完成，也就是在程序实际处理输入前。查看 ，***在第 263 页***，以及 ，***在第 265 页***，来获取从 ARGV 中删除元素的各种方法。

为了取得输入 awk 程序中的选项，用 -- 来结束 awk 的选项，然后再提供awk 程序的选项，如下面的方式：

**awk -f myprog.awk -- -v -q file1 file2 ...**

下面的处理 ARGV 的代码片段，就是用来检查，移除前面所提到的命令行选项：

**BEGIN {**

**for (i = 1; i < ARGC; i++) {**

**if (ARGV[i] == "-v")**

**verbose = 1**

**else if (ARGV[i] == "-q")**

**debug = 1**

**else if (ARGV[i] ~ /^-./) {**

**e = sprintf("%s: unrecognized option -- %c"，**

**ARGV[0]， substr(ARGV[i]， 2， 1))**

**print e >"/dev/stderr"**

**} else**

**break**

**delete ARGV[i]**

**}**

**}**

在 gawk 中用 -- 来结束 awk 的选项不是必须的，除非指定了 --posix，gawk 会默默地将不能够识别的选项放到 ARGV 中以备 awk 程序来处理。一看到不认识的选项，gawk 会停止查找其他的可能可以识别的选项。前面的命令行在 gawk 中可以如下：

**gawk -f myprog.awk -q -v file1 file2 ...**

由于 -q 不是一个有效的 gawk 选项，它以及后面的 -v 选项都会被传递给 awk 程序。（查看 ，***在第 237 页***，来查看分析命令选项的库函数。）

当设计你的程序的时候，你应该选择那些不与 gawk 冲突的选项，因为它将后续的要传递给你的程序的选项之前先处理掉自己可以接受的选项。使用以 ‘#!’ 并使用 -E 参数也许有帮助（查看 ，***在第 22 页***，以及 ，***在第 35 页***）。

## 7.6 总结

* 模式-行为 对组成了 awk 程序的基本元素。模式要是么一个普通的表达式，范围表达式，或者是正则表达式常量，或者是特殊关键字 BEGIN，END，BEGINFILE，ENDFILE，或者为空。动作在当前记录匹配模式后执行。空的（省略）的模式匹配所有的记录。
* 从 BEGIN 与 END 模式中进行 I/O 有一定的限制，对于 BEGINFILE 以及 ENDFILE 也是如此，而且限制更多。后两者可以让你在处理文件之间打入勾子，因此可你使用从可以导致致使错误的文件中恢复回来（例如文件不能被打开）。
* Shell 变量可以通过仔细地使用 Shell 引号来用在 awk 程序中。用 awk 的 -v 选项来传递 Shell 变量到 awk 变量会更加容易。
* 动作由花括号包含的语句组成 。语句由表达式，控制语句，复合语句，输入输出语句以及删除文句组成。
* awk 的控制语句是 if-else，while，for 和 do-while。gawk 添加了 switch 语句。for 语句有两个用处：一个是执行一般的循环，另一个是用来迭代数组。
* break 与 continue 可以让你早点或者开启下一个循环替代（或者跳出 switch）。
* next 与 nextfile 可以让你读取下一个记录并从你程序的开头执行，或者路过下一个输入文件并重新开始。
* exit 语句用来结束你的程序。当从动作中执行时（或者在函数体内执行时），它将控制流传递到 END 语句。从 END 语句体中执行时，则会直接退出。你可以传递一个可选的数值参数来作为 awk 的退出状态。
* 一些预定义的函数可以控制 awk，主要是用于 I/O。其他的变量用来向你的程序传递信息。
* ARGC 与 ARGV 可以让你的程序使用命令行参数。从 BEGIN 规则中操作它们，可以使你控制 awk 如何来处理数据文件。

# 第八章 *awk* 数组

数据是被称为元素的值的表。数组的元素都由索引来区分，索引即可以是数值也可以是字串。

本章描述 awk 中的数组如保来工作，如何使用元素，如何来扫描数组成的每个元素，以及如可来删除数组元素。同时，也描述 awk 如何来模拟多维数组，以及在使用数组时的一些不太明显的地方。本章还会讨论 gawk 中用于对数组排序的工具，然后以 gawk 支持真正的数组之数组的能力为结尾。

## 8.1 数组基础

本节描述一些基础：一次处理数组中的一个元素，以及遍历数组中的所有元素。

### 8.1.1 数组介绍

*对一个关联数组进行线性扫描就好像用一个已经装弹的 Uzi 枪将人敲死。（喻大才小用——译者）*

*—Larry Wall*

awk 语言提供了一维数组用以存储一组相关的串或者数字。每一个 awk 数组都必须有一个名字。数组名字与变量名有一样的语法，任意有效的变量名也都是一个有效的数组名。但是在同一个程序中，一名不能两用（同时作为数组又作为变量）。

在 awk 中的数组表面上与在其他的编程语言中的数组相似，但是它们却有着本质的不同。在 awk 中，没有必要指定数组的大小就可以使用。而且，任意的数字或者字串，不仅仅是连续的整数，都可以用来作数组的索引。

其他大多数语言，数组必须经声明才可用，包括指定在数组中包含什么样元素与组件。在这样的语言中，声明[[48]](#footnote-49)会导致内存的连续块分配用来存储这些元素。通常情况下，数组中的索引必须是一个非负整数。例如，索引 0 指定了数组中的第一个元素，实际上就是存储在内存块中开始的元素，等等。没有可能为数组添加更多的元素，因为它只包含声明时所给定的元素个数。（一些其他语言可以使用任意的起始与结束索引，如 ***‘15 .. 27’***，但是数组的大小一经声明后，就是固定的。）

一个4元素的连续数组看起来如图 8.1 所示，元素的值假设为8，“foo”，“”跟30。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 8 | foo | "" | 30 | Value |
| 0 | 1 | 2 | 3 | Index |

图 8.1：连续数组

其中仅值会被存储起来，索引则是在值的顺序中隐含的。这里，8的索引为0，因为8出现在0的位置，其之前没有元素存在。

awk 的数组也是不同的——它们都是关联性的。这就是说，每个数组都是“成对值”集合——下面是索引与其关联的数组元素值：

**Index Value**

**3 30**

**1 "foo"**

**0 8**

**2 ""**

成对值以随意的顺序出现，因为他们与顺序无关。[[49]](#footnote-50)

关联数组的一个好处就是新的成对值可以在任意时间进行添加。例如，假如第10个元素被加到数组中，而其值为 “number ten”。则结果为：

**Index Value**

**10 "number ten"**

**3 30**

**1 "foo"**

**0 8**

**2 ""**

现在数组是稀疏的，即有些索引是缺失的。它有元素 0-3 与10，但是没有元素 4，5，6，6，7，8 与 9。

关联数组的另一个结果就是索引不必是非负的整数。任何数值，甚至是字串都可以作为索引。例如，下面的数组将英语单词翻译为法语：

**Index Value**

**"dog" "chien"**

**"cat" "chat"**

**"one" "un"**

**1 "un"**

在这里，我们决定将数值1翻译为拼写与数值形式——这个例子说明了一个数组即可以用数字也可以用字串作为索引。（事实上，数组下标总是字串。这里面，数值用于数组下标如何工作，有一些微妙的地方，这会在 ， ***在第 174 页***，进行更详细的说明。）在这里，数值 1 没有双引号引起来，因为 awk 会自动地将其转换为字串。

IGNORECASE 的值不会影响在数组进行下标操作时结果。用来存储与返回数组元素必须使用相同的字串。当 awk 创建一个数组（比如在 split() 内建函数中），那个数组的索引就是从1开始的连续整数。（查看 ， ***在第 184 页***）

awk 的数组是非常高效的——访问元素的时间与数组中的元素数量无关。

### 8.1.2 指向数组元素

使用数组的主要方式就是使用它的元素。数组参考是一个如下面这样的表达式：

**array[index-expression]**

在这里，array 是数组的名字。表达式 index-expression 是数组中所需要元素的索引。

数组参考的值是数组指定元素的当前值。例如 foo[4.3] 是一个表达式参考，是指数组在索引 ‘4.3’ 的元素。

对未存在数组数组元素的参考，会产生一个 “” 值，即空串。这也包括了那些还没有被赋值的元素，以及已经删除的元素（查看 ***，在第 175 页***）。

提示：对不存在元素的参考会自动地创建这个数组元素，并将窗口作为它的值。（在某种情况下，这却是不幸的，因为这会浪费 awk 中的内存。）

awk 新手程序员常常犯通过检查其值是否为空来检查元素存不存在的错误：

**# Check if "foo" exists in a: Incorrect!**

**if (a["foo"] != "") ...**

这是不正确的，有两个原因：第一，如果之前不存在 a["foo"]，则会创建它。第二，（看起来不寻常）将数组元素的值设置成空串是有效的。

为了确定特定索引的元素是否存在数组中，使用下面的表达式：

**indx in array**

这个表达式测试特定的索引 indx 是否存在，并且不会产生如果元素不存在会创建这个元素的副作用。如果 array[indx] 存在，则表达式的具有值1（true），否则为0（false）。（我们在这里使用的是 indx，是因为 index 是内建函数的名字。）例如，这个语句测试数组 frequencies 是不存在索引 ‘2’：

**if (2 in frequencies)**

**print "Subscript 2 is present."**

注意，这不是测试数组 frequencies 是否包含值为 2 的元素。除了扫描所有的元素，否则的话没有其他的办法。同时，这也不会创建 frequencies[2]，而下面的代码则会（是不正确的）：

**if (frequencies[2] != "")**

**print "Subscript 2 is present."**

### 8.1.3 给数组元素赋值

数组元素可以像 awk 变量一样被赋值：

**array[index-expression] = value**

array 是数组的名字。表达式 ***index-expression*** 是 array 要被赋值的元素的索引。表达式的值是赋给数组元素的值。

### 8.1.4 基本数组实例

下面的程序提取一系列的行，每一行以数字开始，并按行号的顺序打印它们。行号第一次读取的时候不是顺序的——相反，它们是打乱的。程序对行进行排序，使数组用行号作为下标。程序然后按照行号的顺序打印它们。这是个非常简单的程序，但是在遇到重复的行号，空格或者不是以数值开始的行的时候会导致困惑：

**{**

**if ($1 > max)**

**max = $1**

**arr[$1] = $0**

**}**

**END {**

**for (x = 1; x <= max; x++)**

**print arr[x]**

**}**

第一个规则跟踪当前已经看到的最大的行号，同时也将每一行存储在数组 arr 中，并以行号作为索引。每二个规则则是打印所有的行。当程序执行时，有下面的输入：

**5 I am the Five man**

**2 Who are you? The new number two!**

**4 . . . And four on the floor**

**1 Who is number one?**

**3 I three you.**

输出则是：

**1 Who is number one?**

**2 Who are you? The new number two!**

**3 I three you.**

**4 . . . And four on the floor**

**5 I am the Five man**

如果行号是重复的，已经存在的行的最后一行将覆盖其他的。在行号间的空隙可以通过在程序的 END 规则中做一个简单的增强就可以处理，如下：

**END {**

**for (x = 1; x <= max; x++)**

**if (x in arr)**

**print arr[x]**

**}**

### 8.1.5 遍历数组元素

在使用数组的程序中，常常需要使用循环来执行针对数组中每个元素的操作。在其他的语言中，数组是连续的，并且过些限制为非负整数，处理起来就非常的简单：所有有效的索引都可以为最低索引到最大的索引。这个技巧在 awk 中并不可用，因此任意的数字与字串都可以作为数组的索引。因此，awk 有一个特殊的 for 语句来扫描数组：

**for (var in array)**

**body**

这个循环为程序中使用过的数组中的每一个索引执行一次 body，并将 var 变量设置成索引值。

下面的程序使用这种形式的 for 语句。第一个规则扫描输入记录，并且注意出现在输入中的单词（至少出现一次），通过用其中一个单词作为索引来并将值 1 存到存储数组中。第二个规则扫描已经使用过的元素来查找所有的在输入中出现的不同的单词。然后打印其长度大于10个字符的单词，同时也打印此类单词出现的次数。查看 ，***在第 184 页***，来获取内建函数 length() 的更多信息。

**# Record a 1 for each word that is used at least once**

**{**

**for (i = 1; i <= NF; i++)**

**used[$i] = 1**

**}**

**# Find number of distinct words more than 10 characters long**

**END {**

**for (x in used) {**

**if (length(x) > 10) {**

**++num\_long\_words**

**print x**

**}**

**}**

**print num\_long\_words， "words longer than 10 characters"**

**}**

查看 ，***在第 279 页***，来获取更多的更多更详细的此类例子。

通过这个语句访问数组中的元素的顺序则是由 awk 内部对数组元素的安排确定的，并且在标准的 awk 中是无法控制与改变的。如果新元素被在循环体内的语句加入到数组中，前面的规则可会导致问题的产生，即在 for 循环中什么时候指向它们是不可预测的。相似的，改变循环体内的 var 会导致奇怪的结果。因此最好没做这样的事情。

作为信息点，gawk 开始循环之前，挂起被迭代的元素的列表，并不可以改变它。但是不是所有的 awk 版本都这么做。考虑下面的程序，名为 loopcheck.awk：

**BEGIN {**

**a["here"] = "here"**

**a["is"] = "is"**

**a["a"] = "a"**

**a["loop"] = "loop"**

**for (i in a) {**

**j++**

**a[j] = j**

**print i**

**}**

**}**

这是用 gawk （与 mawk）执行时的结果：

**$ gawk -f loopcheck.awk**

**-| here**

**-| loop**

**-| a**

**-| is**

而下面是 BWK awk 的执行结果：

**$ nawk -f loopcheck.awk**

**-| loop**

**-| here**

**-| is**

**-| a**

**-| 1**

### 8.1.6 在 gawk 中使用预定义的数组遍历顺序

本小节描述的是 gawk 特有的特性。

在默认情况下，当 for 在遍历数组时，其顺序是未定义的，即是说，awk 的实现决定了在遍历数组时的顺序。这个顺序通常是基于数组的内部实现，并在不同版本的 awk 有差异。

尽管你通常做的都是一些简单的事情，如“通过以升序的方式比较索引来遍历数组”，或者是“通过以降序的方式比较值来遍历数组”。gawk 通过两种机制让你来执行这样的控制：

* 设置 PROCINFO["sorted\_in"] 的值为预定义值之一。后面会讲到。
* 设置 PROCINFO["sorted\_in"] 为用户自定义的函数用来比较数组中的元素。这个高级特性在后面的章节 ， ***在第 298 页***，中进行描述，下面是可获取的 PROCINFO["sorted\_in"] 值：

"***@unsorted***"

数组元素以任意顺序处理，这是默认的 awk 行为。

***"@ind\_str\_asc"***

以字串比较索引升序方式，这是最基本的排序。（在内部，数组索引总是字串，因此 ***‘a[2\*5] = 1’*** 中的索引是 “10”，而不是数字的 10。）

***"@ind\_num\_asc"***

以索引升序的方式，但是强制索引以数值对待。对于不是数值的索引，就当成是 0 的方式进行定位。

***"@val\_type\_asc"***

以元素值的升序方式排序（而不是索引）。排序是通过给元素所赋值的类型进行的（查看 ， ***在第 127 页***）。所有的数值都在字串之前，后面再接子数组。（子数组还没有讨论到，查看 ， ***在第 178 页***）。

***"@val\_str\_asc"***

按元素值的升序排序（不是按索引）。标量值与字串进行比较，如果有子数组，则最后与其比较。

***"@val\_num\_asc"***

按元素值的升序排序（不是按索引）。标量值与字串进行比较，如果有子数组，则最后与其比较。当数值相同时，用字串比较来确定顺序。这保证了在不同版本的 C qsort() 函数[[50]](#footnote-51)间的一致结果，在内部它被 gawk 用来执行排序。

***"@ind\_str\_desc"***

与 "@ind\_str\_asc" 相似，但是字串顺序是从高到低，即降序。

***"@ind\_num\_desc"***

与 "@ind\_num\_asc" 相似，但是数值索引是从高到低，即降序。

***"@val\_type\_desc"***

与 "@val\_type\_asc" 相似，但是基于元素值的类型则是从高到低。如果有子数组，则最先比较子数组。

***"@val\_str\_desc"***

与 "@ val\_str\_asc " 相似，但是基于元素值的字串则是从高到低。如果有子数组，则最先比较子数组。

***"@val\_num\_desc"***

与 "@ val\_num\_asc " 相似，但是基于元素值的数值则是从高到低。如果有子数组，则最先比较子数组。

数组的遍历顺序是在循环开始执行之前就确定的。在循环体内改变 PROCINFO["sorted\_in"] 的值不会影响循环。例如：

**$ gawk ’**

**> BEGIN {**

**> a[4] = 4**

**> a[3] = 3**

**> for (i in a)**

**> print i， a[i]**

**> }’**

**-| 4 4**

**-| 3 3**

**$ gawk ’**

**> BEGIN {**

**> PROCINFO["sorted\_in"] = "@ind\_str\_asc"**

**> a[4] = 4**

**> a[3] = 3**

**> for (i in a)**

**> print i， a[i]**

**> }’**

**-| 3 3**

**-| 4 4**

当通过元素值的方式来排序，如果某个值为子数组，则其被认为比任意的字串与数组都大，而不管子数组中包含的内容，而所有的子数组都当成相同来对待。而它们之间的相对顺序则由它们的索引字串来决定 。

下面是一些在进行排序数组遍历时要记住的点：

* PROCINFO["sorted\_in"] 的值是全局的，它影响所有的数组遍历循环。如果你要在你的代码中改变它，则你需要确定它是否存在，同时要保存与恢复它的值：

**...**

**if ("sorted\_in" in PROCINFO) {**

**save\_sorted = PROCINFO["sorted\_in"]**

**PROCINFO["sorted\_in"] = "@val\_str\_desc" # or whatever**

**}**

**...**

**if (save\_sorted)**

**PROCINFO["sorted\_in"] = save\_sorted**

* 如早前所提到的，默认的数组遍历由 “@unsorted”表示。你也可以给 PROCINFO["sorted\_in"] 赋一个空串来获得这样的默认行为，或者只需要用 delete 语句将 “sorted\_in” 从 PROCINFO 中删除。（delete 语句还没有讨论到，请查看 ， ***在第 175 页***。）

另外，gawk 提供了内建的用于数组排序的函数，查看 ， ***在第 302 页***。

## 8.2 用数值作为数组下标

要记住数组的一个重要点就是数组的下标总是字串。当数值用于下标，它会在用于下标前被转换为字串（查看 ， ***在第 117 页***）。这意味着，预定义变量 CONVFMT 会影响你的程序访问数组的元素。例如：

**xyz = 12.153**

**data[xyz] = 1**

**CONVFMT = "%2.2f"**

**if (xyz in data)**

**printf "%s is in data\n"， xyz**

**else**

**printf "%s is not in data\n"， xyz**

程序会打印 ***‘12.15 is not in data’***。第一条语句给了 xyz 一个数值值。用字串“12.153”赋给 data[xyz] 的下标（使用默认的 CONVFMT 转换值 “%.6g”）。因此，数组元素 data["12.153"] 被赋值为1。程序然后改变 CONVFMT 的值。测试语句 ***‘(xyz in data)’*** 产生一个 xyz 的新字串值——这次则是 “12.15”——因为 CONVFMT 的值只允许有两个有效位。这个测试会失败，因为 “12.15” 与 “12.153”是不同的。

根据转换规则（查看 ， ***在第 117 页***），整数总是会转换为整数的字串，不管 CONVFMT 的值是什么。因此下面的例子可以工作：

**for (i = 1; i <= maxsub; i++)**

**do something with array[i]**

“整数总是转换为整数字串”规则在数组索引中有另一个结果。八进制与十六进制常量（查看 ， ***在第 113 页***）在内部会转换为数值，它们的原型形式会被丢弃。这就是说，array[17]， array[021]， 与 array[0x11]都是指向同一个元素！

与 awk 许多的事情一样，大部份时间下，事情都与你所期望它所应该是那样的方式工作。但是对于实际规则具有精确知识是非常有用的，因为它们有时会让你的程序表现出一些很微妙的结果。

## 8.3 用未初始化的数值作为数组下标

假如有必要写一个程序来将输入的数据逆序打印。一个合理的尝试来这么做（带上一些测试数据）可能如下：

**$ echo ’line 1**

**> line 2**

**> line 3’ | awk’{ l[lines] = $0; ++lines }**

**> END {**

**> for (i = lines - 1; i >= 0; i--)**

**> print l[i]**

**> }’**

**-| line 3**

**-| line 2**

不幸的是，第一行的数据不会出现在输出中！初看起来，我们认为这个程序应该可以工作。变量 lines 是未初始化的，而一个未初始化的变量具有值为 0。因此，awk 应该会打印 l[0] 的值。

这里的问题是 awk 数组中的下标总是串。未初始化的变量被用作串时，其值为 “”，而不是0。所以， ***‘line 1’*** 最终被存在 l[""] 中。下面的版本则可以正确工作：

**{ l[lines++] = $0 }**

**END {**

**for (i = lines - 1; i >= 0; i--)**

**print l[i]**

**}**

在这里，‘++’ 强制 lines 成为一数值，所在使得其“旧值”（因为是未初始化的变量，因此就无所谓旧值，这里是为表述方便。——译者）为数值0。然后其被转换为“0”作为数组的下标。

就算不是特别常用，空串（“”）也是一个有效的数组下标。gawk 会对使用空串作为下标发出警告，但这只在命令行参数中指定了 --lint 参数（查看 ， ***在第 35 页***）。

## 8.4 delete 语句

为了删除数组的一个独立元素，可以使用 delete 语句：

**delete array[index-expression]**

一旦数组中的一个元素被删除，那个元素的值就不再可用。就好像那个元素从来没有被使用到或者也从来没有被赋值。下面的例子从数组中删除一个元素：

**for (i in frequencies)**

**delete frequencies[i]**

这个例子删除数组 ***frequencies*** 的所有元素。只要一个元素被删除了，后续的 for 语句来扫描数组时就不会再报告被删除的元素，并且用 in 操作符来检查被删除的元素的是否存在会返回 0（即 fallse）：

**delete foo[4]**

**if (4 in foo)**

**print "This will never be printed"**

要注意的是，删除一个元素给元素赋一个空值（空串，“”）是两码事。如：

**foo[4] = ""**

**if (4 in foo)**

**print "This is printed， even though foo[4] is empty"**

删除一个不存在的元素并不是一个错误。但是，如果在命令行中指定了 --lint 参数（查看 ， ***在第 35 页***），当删除数组中不存在的元素的时候，gawk 会产生一个告警信息。

通过在 delete 语句后面的数组中，不写下标时，可以用一个语句就可删除数组的所有元素，如下：

**delete array**

使用这种版本的 delete 语句大概要比使用等同的一次删除一个元素的循环要快大概3倍。

这种类型的 delete 语句也被 BWK awk 与 mawk 所支持，很多其他版本的实现也支持。

提示：很多年以来，使用 delete 语句不带下标是一个通用的扩展。在 2012年的9月，它被接受并包含到 POSIX 标准中。查看 Austin Group 网站。

下面的语句提供了一种可移植但是不是特别明显的方式来将数组清除：[[51]](#footnote-52)

**split(""， array)**

split() 函数（查看 ***，在第 184 页***）会首先清除目标数组。这个调用请求分割一个空串。因为没有数据用来分割，函数只是简单地清除数组然后返回。

注意：删除数组中的所有元素不会改变它的类型，你不可清除一个数组，然后使用数组的名字来作为一个标量名（比如一个常规的变量）。例如，下面的例子不会工作：

**a[1] = 3**

**delete a**

**a = 3**

## 8.5 多维数组

多维数组是这样的数组，其元素的标识是通过一连串的索引而不是单个索引来标识的。例如，两维数组需要两个索引。一般的方式（包括 awk 在内的许多语言）要参考到两维数 grid 中的元素，则用 grid[x，y]。

在 awk 中的多维数组 是通过将索引连接成字串来实现的。awk 将索引转换为字串（查看 ， ***在第 117 页***）然后将它他连接在一起，并在它们之间存放分隔符。这样可以创建一个单一 的字串来描述分离索引的值。合并后的字串用作普通的一维数组的一个索引。分隔符则使用内建变量 SUBSEP 的值。

例如，假设我们对表达式 ***‘foo[5，12] = "value"’*** 求值，并且 SUBSEP 的值为 “@” 的时候。数值 5 与 12 会被转换成字串并且用 “@” 连接它们，从而产生 “5@12”，所以，数组元素foo["5@12"] 的值被设置成 “value”。

一量元素被存储，awk 就不再记得它是存成单个的索引还是用一串索引。***‘foo[5，12]’*** 与 ***‘foo[5*** SUBSEP ***12]’*** 总是等同的。

SUBSEP 的默认值是 “\034”，这包含了非打印字符，并不会出现在 awk 程序中，或者不会出现在输入数据中。使用一个不太可能的字符的用处来自于这样的事实，即索引值中包含一个串匹配 SUBSEP 可以导致合并后的字串的含义不明。如 SUBSEP 是 “@”，则 ***‘foo["a@b"， "c"]’*** 与 ***‘foo["a"， "b@c"]’*** 则没办法区分开来，因为两者都会存成 ***‘foo["a@b@c"]’***。

要测试一个特定的索引序列是否存在于多维数组中，使用与一维数组相同的 in 操作符。将整个序列写在气括号中，并用逗号分隔，并做为 in 的左操作数：

**if ((subscript1， subscript2， ...) in array)**

**...**

下面是一个将输入作为一个域的二维数组的例子，然后将数组顺时针转90度（注下），再打印结果。假设所有的行都相相同数目的元素：

**{**

**if (max\_nf < NF)**

**max\_nf = NF**

**max\_nr = NR**

**for (x = 1; x <= NF; x++)**

**vector[x， NR] = $x**

**}**

**END {**

**for (x = 1; x <= max\_nf; x++) {**

**for (y = max\_nr; y >= 1; --y)**

**printf("%s "， vector[x， y])**

**printf("\n")**

**}**

**}**

如果输入如下：

**1 2 3 4 5 6**

**2 3 4 5 6 1**

**3 4 5 6 1 2**

**4 5 6 1 2 3**

则程序会产生下面的输出：

**4 3 2 1**

**5 4 3 2**

**6 5 4 3**

**1 6 5 4**

**2 1 6 5**

**3 2 1 6**

### 8.5.1 遍历多维数组

没有特殊的 for 语句用来进行多维数组的遍历。也不会有这么一个，因为事实上， awk 根本就没有二维数组或者元素。只有一个多维方式来访问数组。

但是，如果你的程序有这样的数组总是像多维数组一样来访问，你可以用 split() 函数（查看 ***，在第 184 页***）来获得 for 语句一样的扫描效果（查看 ，***在第 170 页***）。它是以下面的方式来工作的：

**for (combined in array) {**

**split(combined， separate， SUBSEP)**

**...**

**}**

这里将变量 combined 为数组中每个索引的连接组合，然后用 SUBSEP 的值，将它分割为独立的索引。单独的索引然后就可以独立地成为数组的元素。

因此，如果前面的值存在 array[1，"foo"] 里，则会有索引为 “1\034foo” 的元素存在数组中。（回忆一下 SUBSEP 的默认值是 “\034”。）早早晚晚，for 语句会找到索引，并将变量 combined 设置成 “1\034foo”来进行迭代。然后 split() 函数会像下面这样来调用：

**split("1\034foo"， separate， "\034")**

结果会将 separate[1] 设置成 “1”，separate[2] 为 “foo”。太神奇了，原来的分离的索引序列又被恢复了。

## 8.6 数组之数组

gawk 超越了标准的 awk 的多维数组的访问，并提供了真正的数组之数组。子数组的元素通过它们在方括号中的下标来取得，就是主数组元素一样。例如，下面创建一个两元素的子数组，索引为主数组 a 的 1：

**a[1][1] = 1**

**a[1][2] = 2**

这里模拟了一个真正的两维数组。每一个子数组元素包含另一个子数组作为值，进而也可以包含其他的数组。通过这样的方式，你可以创建三维或者多维的数组。索引可以是 awk 所任意表达式，包括用逗号分隔的标量（例如常规的 awk 模拟多维的下标）。因此，下面代码在 gawk 中是有效的：

**a[1][3][1， "name"] = "barney"**

每一个子数组与主数组可以有不同的长度。实际上，数组的元素，或者它的子数组不必要都包含相同类型。这意味着，主数组与它的子数组在结构上，可以是非矩形的，即是锯齿形的。你可以给主数组 a 索引为 4 的元素赋值一个标量，就算是 a[1] 本身是一个数组而不是一个标量：

**a[4] = "An element in a jagged array"**

维度，行，列这样的术语用在这样的数组中是没有含义的，但是我们使用“维度”隐含，如果要访问一个已经存在的元素，所需要的索引数的最大值。任意元素的类型，一旦被赋值，就不能通过赋值一个不同类型的值来改变它的类型。你必须先删除当前的元素，这样使 gawk 可以有效地丢弃在那个索引下的元素：

**delete a[4]**

**a[4][5][6][7] = "An element in a four-dimensional array"**

这里删除索引为 4 的标量值，然后插入一个包含一个标量的三层的内嵌子数组。你也可以删除整个的子数组的子数组的了数组：

**delete a[4][5]**

**a[4][5] = "An element in subarray a[4]"**

但是要记住，你不可以删除主数组，然后将其用作一个标量。

内置函数中使用数组作为参数时，函数也可以使用子数组。例如，下面的代码段使用 length() （查看 ，***在第 184 页***）来确定主数组与其子数组的元素个数：

**print length(a)， length(a[1])， length(a[1][3])**

对一主数组 a 的输出结果如下：

**2， 3， 1**

***‘subscript in array’*** 表达式（查看 ，***在第 168 页***）工作原理与正规的 awk 数组以及数组之数组的方式相似。例如测试语句 ***‘1 in a’***， ***‘3 in a[1]’***， 与 ***‘(1， "name") in a[1][3]’*** 对于数组 a 都会求值为 1（true）。

***‘for (item in array)’*** 语句（查看 ，***在第 170 页***）可以嵌套，用来扫描一个数组之数组的所有元素，只要它在结构上是矩形的。为了打印两维数组之数组的内容（标量）（在数组中，它的第一层本身是一个数组，但不必是有相同的长度），你可以使用下面的代码：

**for (i in array)**

**for (j in array[i])**

**print array[i][j]**

isarray() 函数 （查看 ， ***在第 203 页***）可以让你测试数组的元素是否为数组：

**for (i in array) {**

**if (isarray(array[i]) {**

**for (j in array[i]) {**

**print array[i][j]**

**}**

**}**

**else**

**print array[i]**

**}**

如果一个事先知道是一个锯齿形的数组，你通常要通过控制语句来解决遍历问题。例如，下面的代码打印主数组 a 的元素：

**for (i in a) {**

**for (j in a[i]) {**

**if (j == 3) {**

**for (k in a[i][j])**

**print a[i][j][k]**

**} else**

**print a[i][j]**

**}**

**}**

查看 ，***在第 249 页***，来获取用户自定义的函数，其功能就是用来遍历任意维度的数组之数组。

回忆一下参考一个未初始化的数组元素会产生一个 “” 值，即空串。这有一个非常重要的含义，即当你想使用一个子数组作为参数向函数传递，如下面的代码所呈现的：

**$ gawk ’BEGIN { split("a b c d"， b[1]); print b[1][1] }’**

**error gawk: cmd. line:1: fatal: split: second argument is not an array**

要解决这样的问题，首先要通过创建一个任意的索引，以强制 b[1] 成是一个数组：

**$ gawk ’BEGIN { b[1][1] = ""; split("a b c d"， b[1]); print b[1][1] }’**

**-| a**

## 8.7 总结

* 标准的 awk 提供一维关联数组（通过字串进行索引的数组）。所有的数组都是关联的，数值索引会被自动转换为字串。
* 数组元素通过 array[indx] 进行参考。参考一个不存在的元素，则会创建这个元素。
* 正确的查看是否给定索引的元素是否存在数组中的方式是使用 in 操作符：

***‘indx in array’***.

* 使用 ***‘for (indx in array) ...’*** 来描述数组中的每个元素。在环境体中， indx 会轮流地设置为元素的索引值。
* ***用 ‘for (indx in array)’*** 来遍历数组的顺序在 POSIX awk 中是未定义的，并且不同的版本实现是不同的。gawk 使用通过设置预定义变量 PROCINFO["sorted\_in"] 的值来控制遍历的顺序。
* 使用 ***‘delete array[indx]’*** 来删除单个元素。要删除数组中的所有元素，则使用 ***‘delete array’***。后者的特性多年来都是一个通用扩展，但是现在已经标准化了，但是可能不是所有的被商业版本的 awk 都支持。
* 标准的 awk 通过用逗号来分隔不同的下标来模拟多维数组。这些下标值会被合并成一个单独的串，并且使用 SUBSEP 的值来进行分离。通过这种方式创建的下标是没有保存的，所以改变 SUBSEP 的值会导致未预期的结果。你可以使用 ***‘(sub1， sub2， ...)in array’*** 来查看一个多维数组的相应下标是否存在数组中。
* gawk 提供了真正的数组之数组。在这样的数组中，你可以在方括号中使用独立的集合来表示每一维。例如 data[row][col]。数组元素即可以是标量（数值或者字串），也可以是其他的数组类型。
* 使用 isarray() 内置函数来确定一个数组元素是不是子数组。

# 第九章 函数

本章描述 awk 的内置函数，它们被分成三类：数值类，字串类与 I/O 类。gawk 还提供额外的一些函数用来表示时间的值、或者进行位操作，数组排序，提供类型信息，国际化与本地化程序。

除了内置函数，awk 也提供写新函数的能力，这样可以在程序的后面进行调用。本章的下半部份描述了用户自定义函数。最后，我们探讨间接函数调用，一种 gawk 才有的扩展，这样可以让你在运行时来确定最终来调用什么样的函数。

## 9.1 内置函数

内置函数在你的 awk 程序总是可用的。本章定义了所有的 awk 中的函数，它们中的一些在其他的章节中已经被提到过，在这里进行汇总，你可以方便地参考。

### 9.1.1 调用内置函数

为了调用 awk 的内置函数，只需要输入函数的名字，后面跟着放置在括号内的参数列表即可。例如，***‘atan2(y + z， 1)’*** 为调用函数 atan2()，并带有两个参数。

内建函数中函数名与开括号之间的空格会被忽略，但是这不是一个好的编程习惯，应该要避免在它们之间输入空格。用户自定义的函数不允许在这个位置有空格，因此一个简单的编程习惯就可以避免这样的错误——就是不要在函数名后面插入空格。

每个内置函数都接收特定数目的参数。在某些情况下，参数可以省略。对于省略参数的默认值函数与函数间是不同的，并且会单独进行说明。一些 awk 实现中，有一些额外会被忽略。但是，在 gawk 中，给内置函数一个额外的参数是一个致使错误。

当函数被调用时，生成函数实际参数的表达式会在调用执行前进行求值。例如，下面的代码片段中：

**i = 4**

**j = sqrt(i++)**

变量 i 会在调用 sqrt() 之前递增为值 5，但是调用时会用值 4 作为实际的参数。作为函数参数的表达式的求值的顺序是未定义的。所以，要避免写从左到右，或者从右到左进行求值的程序。如：

**i = 5**

**j = atan2(++i， i \*= 2)**

如果求值的顺序是从左到右，则 i 会为 6，然后成为 12。则会用 6 与 12 两个值调用 atan2()。但是如果顺序为从右到左，i 首先为 10，然后为 11，则 atan2 会用 11，与 10 来进行调用。

### 9.1.2 数值函数

下面的列表描述了所有的用于数值处理的内置函数。可选的参数用方括号（[]）包含起来：

***atan2(y， x)***

返回 y/x 弧度的反正切值。你可以使用 ‘pi = atan2(0， -1)’ 来求得 π 的值。

***cos(x)*** 返回 x 的余弦值，x 为弧度。

***exp(x)*** 返回 x 的自然幂（e ^ x）或者 x 的值超过范围则报错。x 的范围依赖于你的机器的浮点数表示。

***int(x)*** 返回最接近于 x 的整数，其值在 x 与 0 之间，并向 0 进行截取。如 int(3) 为 3， int(3.9) 为 3，int(-3.9) 为 -3， int(-3) 也为 -3。

***log(x)*** 如果 x 为正数，返回 x 的自然对数，否则在 IEEE 754 的系统上返回 NaN（“not a number”）。此外，如果 x 为负数，gawk 还会打印一个警告信息。

***rand()*** 返回一个随机数。 rand() 的值一致地分布在 0 与 1 之间，值可以为 0，但不会等于 1。[[52]](#footnote-53)

通常情况下，你需要的是随机的整数。下面是一个用户自定义的函数，此函数可以用来产生小于 n 的随机非负整数：

**function randint(n)**

**{**

**return int(n \* rand())**

**}**

乘法产生一个随机数大于等于0，但小于 n。使用 int() 因此可以使得结果成为 0 到 n - 1 之间的整数。

下面的例子使用一个相似的函数来产生在 1 到 n 之间的随机整数。这个程序会为每个输入记录打印一个新的随机数：

**# Function to roll a simulated die.**

**function roll(n) { return 1 + int(rand() \* n) }**

**# Roll 3 six-sided dice and**

**# print total number of points.**

**{**

**printf("%d points\n"， roll(6) + roll(6) + roll(6))**

**}**

注意：在大多数的 awk 实现中，包括 gawk，每次运行 awk 时，rand() 都从相同的开始数，或者种子起产生随机数。[[53]](#footnote-54)所以，程序每次运行都产生相同的结果。在 awk 一次运行中产生的数值是随机的，但是每次运行时产生的随机数却是可预测的。这用一调试时非常有用，但是如果你想程序每次使用的时候做不同的事情，你必须改变 seed 为每次运行都不相同的值。要这么做，就调用 srand() 函数。

***sin(x)***  返回弧度 x 的正弦值。

***sqrt(x)*** 返回 x 的正平方根。gawk 在 x 为负值时会打印一个告警。sqrt(4) 的值是 2。

s***rand([x])***

设置开始点，或者种子，用来产生相对于 x 的随机数。每一个种子都会产生特定序列的随机数。[[54]](#footnote-55)所以，如果种子每次都设置成相同的值，则随机数每次都会产生一样。

注意：不同的 awk 实现使用不同的内部随机数产生器。不要希望 awk 程序被不同版本的 awk 执行时会产生相同的随机数序列。

如果在 srand() 函数中省略了 x，则会使用当前的日期与当天的时间来作为种子。这种方式下取得的随机数是真正不可预测的。

srand() 返回的值是前一个种子的值。这可以使得很容易来跟踪种子，以便你需要产生一致的随机数序列。

POSIX 没有指定初始的种子值，因此在不同的 awk 实现中是不同的。

### 9.1.3 字串操作函数

本节我们来讨论一下查找或者更改一个或者一些串的文本。

gawk 可以理解本本语言设置（查看 ， ***在第 136 页***）并且对字串的处理都是根据字符来的，而不是字节。这个区别在理解本地语言设置中，一个字符需要多个字节来表示的时候非常重要。所以，像 length() 函数会返回字串中的字符数，而不是用来表示这些字符的字节数。相似地， index() 是返回字符的索引，而不是字节的索引。

注意：有很多的函数会处理字串的索引。对于这此函数，字串的第一个字符的索引为1。这个与 C 以及其派生的语言不同，在这些语言中，第一个字符的索引位置为0。你需要记住这个区别，以正确处理索引的计算，尤其是你已经习惯了 C 的情况。

下面的列表中，可选的参数放在方括号中（[]）。有几个函数用来处理字串替换，完整的讨论会在 sub() 函数的说明中给出，并且以此为结束，因为列表是按字母顺序排序的。

那些特定于 gawk 的函数用井号字符标示。它们在兼容模式中不可用（查看 ， ***在第 35 页***）：

***asort(source [， dest [， how ] ]) #***

***asorti(source [， dest [， how ] ]) #***

这两个函数的行为是类似的，因此在一起描述。

提示：下面的描述忽略了第三个参数，how，因为需要理解目前我们还没有讨论到的特性。所以，这里讨论有意简化了。（我们后面会提供所有的细节，查看 ，***在第 302 页***，来获取所有的信息。）

两个函数都返回数组 source 中的元素的个数。对于 asort()，gawk 对 source 的值进行排序，并替换经过排序的值的索引为由 1 开始连续整数。如果可选的数组 dest 存在，则 source 的值会复制到 dest 中。dest 则也是经过排序的，而 source 的索引则保持不变。

当进行字串比较时， IGNORECASE 会影响排序（查看 ， ***在第 302 页***）。如果 source 数组包含了子数组（查看 ， ***在第 178 页***），它们会在所有的标量值的后面。子数组不会再进行递归排序。例如，a 的内容如下：

**a["last"] = "de"**

**a["first"] = "sac"**

**a["middle"] = "cul"**

调用 asort()：

**asort(a)**

a 内容的结果如下：

**a[1] = "cul"**

**a[2] = "de"**

**a[3] = "sac"**

asorti() 函数与 asort() 类型，但是只对索引排序，而不是针对值进行排序。所以，在前面的例子中，a 用同样的索引与值，调用 ***‘asorti(a)’*** 则产生如下结果：

**a[1] = "first"**

**a[2] = "last"**

**a[3] = "middle"**

***gensub(regexp， replacement， how [， target]) #***

搜索匹配正则表达式 regexp 的目标字串，如果 how 是以 ‘g’ 或者 ‘G’ 开如的字串，则用 replacement 替换所有匹配的 regexp。否则 how 将会当面成一个指示哪一个要进行替换的数值。如果没有提供 target 参数，则使用 $0。函数返回经过修改的字串，原始的字串则不会变化。

gensub() 是一个通用的替换函数。它的目的是提供比标准的 sub() 与 gsub() 函数多一个特性。

gensub() 提供一个额外的特性是 sub() 与 gsub() 所没有的，即：在 replacement 文本中，指定一个 regexp 组件的能力。这个通过在 regexp 中使用括号来标志组件，然后通过在 replacement 中用 ‘\N’ 来指定，这里 N 是一个从 0 到 9 的数字。如：

**$ gawk ’**

**> BEGIN {**

**> a = "abc def"**

**> b = gensub(/(.+) (.+)/， "\\2 \\1"， "g"， a)**

**> print b**

**> }’**

**-| def abc**

用 sub() 函数，你必须指定两个反斜杠来在字串中包含一个。在 replacement 的文本中，‘\0’ 的序列表示整个匹配的文本，如字符 ‘&’ 所表示的一样。

下面的例子显示了你可以使用第三个参数来控制哪个匹配的 regexp 可以被替换：

**$ echo a b c a b c |**

**> gawk ’{ print gensub(/a/， "AA"， 2) }’**

**a a b c AA b c**

在这个例子中，$0 是默认的 target 字串。gensub() 返回新的串作为它的结果，这个结果直接传递给 print 用来打印。

如果 how 这个参数是一个不以 ‘g’ 或者 ‘G’ 开头的串，或者是一个小于等于 0 的数值，则只有一个替换会被执行。如果 how 是 0，gawk 会产生一个告警。

如果 regexp 没有匹配 target，则 gensub() 的返回值就是原来未经更改的 target 的值。

***gsub(regexp， replacement [， target])***

搜索所有的最左最长的的，无重叠的匹配子串，并且 replacement 进行替换。***gsub()*** 中的 ‘g’ 表示 ‘global’，其表示替换所有出现的字串。例如：

**{ gsub(/Britain/， "United Kingdom"); print }**

会用 ***‘United Kingdom’*** 替换输入记录所有的 ***‘Britain’*** 字串。

gsub() 函数返回执行了替换的个数。如果要搜索及变更（target）的变量被省略了，就是使用整个输入记录 $0。如在 sub() 中，字串 ‘&’ 与 ‘\’ 是特殊字符，并且第三个参数必须是可赋值的。

***index(in， find)***

在 in 这个字串中搜索第一次出现的 find 串，并且返回在字串 in 中出现的最开始的位置。考虑下面的例子：

**$ awk’BEGIN { print index("peanut"， "an") }’**

**-| 3**

如果 find 没有被找到，index 会返回0。对于 BWK awk 与 gawk，使用正则表达式常量来作为 find 参数会是一个致命错误。其他的实则允许，他们仅是简单地将正则表达式常量当成与 ***‘$0 ~ /regexp/’*** 一样的表达式。

***length([string])***

返回字串的字符数。如果 string 是一个数值，则表示这个数值串的数字位数被返回。例如 length("abcde") 的值为5。比较 length(15 \* 35) 其值则为 3。在这个例子中，15 · 35 = 525，而 525 转换成字串 “525”，这里面有三个字符。

如果没有提供参数，length() 则返回 $0 的长度。

提示：在旧版本的 awk 中，length() 函数可以不带括号进行调用。这么做被视为很不好的实践，尽管 2008 版本的 POSIX 标准还是显式地允许它，但那也是为了支持历史实践。为了程序的最大可移植性，在调用时总是使用括号。

如果调用 length() 时使用的是一个还没有被使用的变量，gawk 会强制变量为一个标量。其他的 awk 实现会将变量保留为一个无类型的变量。如：

**$ gawk ’BEGIN { print length(x) ; x[1] = 1 }’**

**-| 0**

**error gawk: fatal: attempt to use scalar *‘x’* as array**

**$ nawk ’BEGIN { print length(x) ; x[1] = 1 }’**

**-| 0**

如果在命令行中指定了 --lint 参数，gawk 会对这使用产生一个告警。

gawk 与其他的几个 awk 实现，当参数是一个数组时， length() 返回的是数组中的元素个数。（c.e.）初看起来，这没什么作用，因为数组的索引并不保证是从 1 到数组中的元素数目。如果在命令行中使用了 --lint 选项（查看 ， ***在第 35 页***），gawk 会警告说，传递数组参数并不是可移植的。如果指定了 --posix 选项，则使用数组作为参数会是一个致命错误（查看 ***， 在第 166 页***）。

***match(string， regexp [， array])***

搜索匹配正则表达式 regexp 的最长最左边的子串，并返回子串开始的字符位置（索引，如果子串是字串的开始，则返回1）。如果没有找到匹配，则返回 0。

regexp 参数即可以是正则表达式常量（/.../），也可以是一个字串常量（“...”）。如果是后者，字串被当成是正则表达式用来进行匹配。查看 ，***在第 58 页***，可以了解到两种方式之间不同的讨论，以及如何来写出正确的程序的一些提示。

两个参数的顺序，与大多数的其他用正则表达式处理字串函数的相反，比如 sub()，gsub()。记住 match() 函数的参数顺序与 ‘~’ 操作符是一样的：

**‘string ~ regexp’.**

match() 函数会设置预定义的变量 RSTART 为索引，同时也设置预定义的变量 RLENGTH 为匹配子串的长度。如果没有找到匹配，RSTART 的值被设置成 0，而RLENGTH 的值为 -1。

如：

**{**

**if ($1 == "FIND")**

**regex = $2**

**else {**

**where = match($0， regex)**

**if (where != 0)**

**print "Match of"， regex， "found at"， where， "in"， $0**

**}**

**}**

这个程序查找那些匹配存储在变量 regex 中的正则表达式的行。这个正则表达式可以被更改。如果行中的第一个单词是 ‘FIND’，则 regex 会被改成那一行的第二个单词。所以，假如有下面的数据：

**FIND ru+n**

**My program runs**

**but not very quickly**

**FIND Melvin**

**JF+KM**

**This line is property of Reality Engineering Co.**

**Melvin was here.**

则 awk 会打印：

**Match of ru+n found at 12 in My program runs**

**Match of Melvin found at 1 in Melvin was here.**

如果提供了 array 参数，它首先会被清除，然后数组的第0个元素会被设置为匹配正则表达式 regexp 的整个字串。如果 regexp 包含了括号，数组的整数索引元素会被设置成匹配相应匹配括号子表达式的字串部分。例如：

**$ echo foooobazbarrrrr |**

**> gawk ’{ match($0， /(fo+).+(bar\*)/， arr)**

**> print arr[1]， arr[2] }’**

**-| foooo barrrrr**

另外，多维下标也可用来提供 start 索引与每个匹配子表达式的长度 length 索引：

**$ echo foooobazbarrrrr |**

**> gawk ’{ match($0， /(fo+).+(bar\*)/， arr)**

**> print arr[1]， arr[2]**

**> print arr[1， "start"]， arr[1， "length"]**

**> print arr[2， "start"]， arr[2， "length"]**

**> }’**

**-| foooo barrrrr**

**-| 1 5**

**-| 9 7**

有可能没有为每一个括号子表达式提供 start 以及 index 的下标，因为有可能并不是所有都匹配，所以，它们应该用 in 操作符来进行测试（查看 ， ***在第 168 页***）。

match() 的 array 参数是 gawk 扩展。在兼容模式中（查看 ， ***在第 35 页***），使用第三个参数是一个致使错误。

***patsplit(string， array [， fieldpat [， seps ] ]) #***

将 string 用 filedpat 定义的分隔符分割成片，并存储在 array 中，并将字串中分隔符存储在 seps 中。第一块片存储在 array[1] 中，第二片存储在 array[2] 中，依此类推。第三个参数 fieldpat 是一个正则表达式描述在 string 中的域（就好像 PFAT 是一个描述输入记录域的正则表达式）。它即可以是一个正则表达式常量，也可以是一个字串。如果省略了 fieldpat，则会使用 PFAT 的值。patsplit() 返回创建元素的个数。sep[i] 是在 array[i] 与 array[i+1] 之间的分隔符。所有前导的分隔符都存储在 seps[0] 中。

patsplit() 函数将 string 分割为片，其方式是与用 PFAT 将输入行分割成域的方式相似（查看 ， ***在第 79 页***）。

在分割 string 之前，patsplit() 会删除在 array 与 seps 中已经存在的元素。

***split(string， array [， fieldsep [， seps ] ])***

将 string 用 filedpat 定义的分隔符分割成片，并存储在 array 中，并将字串中分隔符存储在 seps 中。第一块片存储在 array[1] 中，第二片存储在 array[2] 中，依此类推。第三个参数 fieldsep 的字串值是一个用来描述在何片分割 string 的正则表达式（正如 FS 可以为正则表达式来描述在何处来分割输入记录一样）。如果 fieldsep 被省略了，则使用 FS 的值。split() 返回被创建的元素个数。seps 是 gawk 扩展，sep[i] 是在 array[i] 与 array[i+1] 之间的分隔符。如果 fieldsep 是一个空格，则前导的空格会全部存储 seps[0] 中，而后缀的空格会存储在 seps[n] 中，其中 n 是 split() 返回的值（即 array 数组中的元素个数）。

split 函数将字串分割成片的方式与将输入行分割成域的方式类似。比如：

**split("cul-de-sac"， a， "-"， seps)**

将字串 "cul-de-sac" 用 ‘-’ 分割成三个域。它将数组 a 的内容设置成如下：

**a[1] = "cul"**

**a[2] = "de"**

**a[3] = "sac"**

同时设置数组 seps 的值为：

**seps[1] = "-"**

**seps[2] = "-"**

调用 split() 的返回值为 3。

随着输入的域分割，当 fieldsep 的值为 “ ” 时，前导的后缀的空格会在赋值给数组元素的值中被忽略，而不是在 seps 中，数组则是由一连串的空格分隔。同时，随着输入记录的分割，如果 fieldsep 是一个空串，在字串中的每一个字符都将分割到数组的元素中。(c.e.）

但是要注意，RS 的值对 split() 函数的工作没有影响。就算是 ***‘RS = ""’*** 会导致换行符成为域分割符，这也不会影响 split() 分割字串的方式。

现代的 awk 实现版本，包括 gawk，允许第三个参数为正则表达式常量（/.../），也可以是一字串。POSIX 标准允许这样的参数。查看 ，***在第 58 页***，可了解使用字串常量与正则表达式之间的常量，并获取一些如何正确地写程序的提示。

在分割 string 前，split() 会删除在数组 array 与 seps 中已经存在的元素。

如果 string 为空，则数组就不会有元素。（所以这是一个可移植的以一个语句来删除整个数组的方法。查看 ， ***在第 175 页***）

如果 string 没有匹配 fieldsep（但 string 不是空），则 array 只会有一个元素。元素的值就是原来的字串。

在 POSIX 模式下（查看 ， ***在第 35 页***），第四个参数是不允许的。

***sprintf(format， expression1， ...)***

返回 printf 采用相同的参数时的输出（但是不输出）的字串。（查看 ， ***在第 97 页***）。例如：

**pival = sprintf("pi = %.2f (approx.)"， 22/7)**

将字串　***‘pi = 3.14 (approx.)’*** 　赋值给变量 pival。

***strtonum(str) #***

检查 str 并返回它的数值值。如果 str 以 ‘0’ 开头，则 ***strtonum()*** 会假设 str 是一个八进制数。如果 str 是以 ‘0x’ 开头，或者 ‘0X’ 开头，***strtonum()*** 会假设 str 是一个十六进制数。例如：

**$ echo 0x11 |**

**> gawk ’{ printf "%d\n"， strtonum($1) }’**

**-| 17**

使用 ***strtonum()*** 函数与给字串添加一个零不同，自动强制性地将字串转换为数值只会在十进制数下有效，对于八进制与十六进制都是无效的。[[55]](#footnote-56)

同时也要注意 ***strtonum()*** 会使用当前的本地语言设置中的用于识别数值的十进制位（查看 ， ***在第 136 页***）。

***sub(regexp， replacement [， target])***

在被当成字串的 target 中搜索匹配正则表达式 regexp 的最左最长的匹配子串。并用 replacement 来规则匹配的子串。被修改过的字串成为 target 的新值。返回替换的次数（0 或者 1）。

regexp 参数即可以是一个正则表达式常量（/.../），也可以是一个字串常量（“...”）。对于后者，字串会被当面成一个正则表达式来进行匹配。查看 ，***在第 58 页***，以了解这两种形式之间的差异，以及了解一些如何正确地写这样的程序的提示。

这个函数非常特别，是因为 target 并不是简单地使用来进行计算出一个值，也不像其他的表达式一样——它必须是一个变量，域或者一个数组元素，因此 sub() 可以将修改的值存在那个地方。如果这个参数被省略了，则默认的值会使用 $0。[[56]](#footnote-57)例如：

**str = "water， water， everywhere"**

**sub(/at/， "ith"， str)**

通过将出现 ‘at’ 的地方替换成 ‘ith’ 来将 str 设置成 ***‘wither， water， everywhere’***。

如果特殊字符 ‘&’ 出现在 replacement 中，它就表示匹配 regexp 的精确子串。（如果 regexp 可以匹配 1 到多个串，则精确的匹配串可能不同。）例如：

**{ sub(/candidate/， "& and his wife"); print }**

会将输入的行中出现的第一个 ‘candidate’ 替换成 ‘candidate and his wife’。下面是另外一个例子：

**$ awk’BEGIN {**

**> str = "daabaaa"**

**> sub(/a+/， "C&C"， str)**

**> print str**

**> }’**

**-| dCaaCbaaa**

这里显示了 ‘&’ 可以表示一个非常量的字串，也说明了正则表达式匹配的最左最长规则的含义（查看 ***在第 57 页***）。

这个特殊字符 ‘&’ 的效果可以被关闭，只需要在字串中这个字符前加一个反斜杠即可。通常，为了插入一个反斜杠到字串中，你必须写两个反斜杠。因此在字串常量中要写 ‘\\’ 来在 replacement 中生成一个字面的 ‘&’ 字符。例如，下面的代码展示了如何来行中将第一个 ‘|’ 替换成 ‘&’：

**{ sub(/\|/， "\\&"); print }**

如上面所提到的，sub() 的第三个参数必须是一个变量，域或者数组元素。某些版本的 awk 允许第三个参数是一个表达式，这个表达式不是个左传。在这种情况下，sub() 依然会搜索模式并返回 0 或者 1，但是替换的结果会被丢弃（如果有的话），而不会发生替换。这种版本的 awk 接受下面这样的表达式：

**sub(/USA/， "United States"， "the USA and Canada")**

为了历史的兼容，gawk 会授受这样的错误代码。但是使用任何不可更改的对象作为第三个参数，会导致致命错误，你的程序也没办法执行下去。

最后，如果 regexp 不是一个正则表达式常量，则它会被转换为一个字串，然后它的值会被当成正则表达式来进行匹配。

***substr(string， start [， length ])***

返回字中 string 的从 start 开始的，***length 字符长度的***子串。字串的第一个字符编号为 1。例如 substr("washington"， 5， 3) 返回 "ing"。

如果没有提供 length 参数，substr() 则返回 string 从 start 字符数开始的全部后面的字串。例如，substr("washington"， 5) 返回 "ington"。如果 length 大于 string 后面剩下的字符数，则也会返回整个后缀的字串，这其中包括 start 处的字符。

如果 start 的值小于 1，substr() 会将其当为 1。（POSIX 没有指定这种情况下如何处理：BWK awk 会如上这么处理，gawk 也这么处理。）如果 start 大于 string 中的字符数，substr() 返回空串。相似的，如果 length 提供了，但是小于等于 0，也返回空串。

由 substr() 返回的串不可以被赋值。所以，试图改变 string 的一部分是一个错误，如下面的例子中所示的：

**string = "abcdef"**

**# try to get "abCDEf"， won’t work**

**substr(string， 3， 3) = "CDE"**

将 substr() 的返回作为 sub() 或者 gsub() 的第三个参数，也是错误的：

**gsub(/xyz/， "pdq"， substr($0， 5， 20)) # WRONG**

（有一些商业版本的 awk 会把 substr() 的返回作为可赋值的，但是这么做是不可移植的。）

如果你想替换 string 的一部分，将 substr() 与字串连接一起使用，如下面的方式：

**string = "abcdef"**

**...**

**string = substr(string， 1， 2) "CDE" substr(string， 6)**

***tolower(string)***

返回 string 的复本，但是将所有的大写字母用它对应的小写字母进行替换。非字母字符则不变化。例如 tolower("MiXeD cAsE 123") 返回 "mixed case123"。

***toupper(string)***

返回 string 的复本，但是将所有的小写字母用它对应的大写字母进行替换。非字母字符则不变化。例如 tolower("MiXeD cAsE 123") 返回 " MIXED CASE123"。

|  |
| --- |
| 匹配空串  在 awk 中，‘\*’ 操作符可以匹配空串。这对于 sub()，gsub() 与 gensub() 函数尤其重要。如：  **$ echo abc | awk ’{ gsub(/m\*/， "X"); print }’**  **-| XaXbXcX**  尽管这有一些意义，但看起来还是让人有点惊讶。 |

#### 9.1.3.1 sub()、gsub() 与 gensub() 函数中的 ‘\’与 ‘&’

注意：本小节据报说是会造成头痛，有可能第一次阅读时想跳过。

当使用 sub()，gsub() 或者 gensub()，并且试图获取字面的反斜杠与星号，并放在 replacement 文本中，你需要记住，到底有几级转义要处理。

首先，有一个词汇层，就是当 awk 读取你的程序，并建立一个要执行的内部复本的时候。然后一个是运行时层，就是当 awk 实际扫描 replacement 串用于确定所产生的结果的时候。

在两个层上，awk 会搜索已经定义好的可以放在反斜杠之后的字符集。在词汇层，它会搜索在 .***在第 51 页***，中列出的转义序列。所以，对每一个 awk 在运行层处理的 ‘\’，你必须在词汇层输入两个反斜杠。当跟在 ‘\’ 之后的字符不是一个有效的转换序列，GWK awk 与 gawk 两者都是简单地移除 ‘\’ 并将下一个字符放在字串中。因此，如 “a\qb” 会被当成 “aqb”。

在运行层，不同的函数在处理 ‘\’ 与 ‘&’ 序列时是不同的。这种情况很（蛋疼）地有点复杂。由于历史原因，sub() 与 gsub() 函数都将两字符 ‘\&’ 序列特殊对待，这个序列在产生的文本中被替换为 ‘&’。任何在 replacement 中的其他的不在 ‘&’ 之前的 ‘\’ 都原样地传递。这个将在表 9.1 中解释。

|  |  |  |
| --- | --- | --- |
| 输入 | ***sub()*** 所见 | ***sub()*** 所产生 |
| \& | & | 匹配的字串 |
| \\& | \& | 字符 ‘&’ |
| \\\& | \& | 字符 ‘&’ |
| \\\\& | \\& | 字符 ‘\&’ |
| \\\\\& | \\& | 字符 ‘\&’ |
| \\\\\\& | \\\& | 字符 ‘\\&’ |
| \\q | \q | 字符 ‘\q’ |

表 9.1：由于历史原因 sub() 与 gsub() 处理转义序列情况

上表显示了词汇层的处理，在里面，一个奇数个的反斜杠在运行时，就变成了偶数，这与 sub() 的运行时的处理一样。（为了简单起见，后面的表只显示在词汇层输入偶数的反斜杠的情形。）

历史方式的问题是，没有办法取得后面跟着匹配字串的字面的 ‘\’。

POSIX 标准的几个版本都尝试来修复这个问题，但是都不成功。细节在此无关。

在某个时候，gawk 的维护者提交了建议文本，形成一个修订版本的标准，以逆转到与最初存在的实践更加接近的规则上。提交的规则存在特殊情形，以有可能产生在匹配的文本前的 ‘\’。这个在表 9.2 中显示。

|  |  |  |
| --- | --- | --- |
| 输入 | ***sub()*** 所见 | ***sub()*** 产生 |
| \\\\\\& | \\\& | 字符 ‘\&’ |
| \\\\& | \\& | 字符 ‘\’ ， 后面跟着匹配文本 |
| \\& | \& | 字符 ‘&’ |
| \\q | \q | 字符 ‘\q’ |
| \\\\ | \\ | \\ |

表 9.2: gawk 针对 sub() 与反斜杠的规则

总的来说，在运行时，现在有三个特殊的字符序列 （***‘\\\&’***， ***‘\\&’***， and ***‘\&’***），由于历史原因只会存在一个。但是在历史情况下， ‘\’ 并不是这三个序列的一部份也并不特殊，并以字符本身输出。

gawk 3.0 与 3.1 会针对 sub() 与 gsub() 来匹配这些规则。POSIX 标准的修改要比期望的时间长得多。另外，gawk 的维护者的建议在标准化的过程中丢失了。最后的规则看起来更简单。结果除了一个情形外都类似。

POSIX 规则描述说，在 replacement 中的 ‘\&’ 产生一个 ‘&’，***‘\\’*** 产生一个字面 ‘\’，‘\’ 后面再跟其他的字符都没有特殊含义，‘\’ 会被直接放到输出中。这些规则在后面的表 9.3。

|  |  |  |
| --- | --- | --- |
| 输入 | ***sub()*** 所见 | sub() 生成 |
| \\\\\\& | \\\& | 字符 ‘\&’ |
| \\\\& | \\& | 字符 ‘\’， 后面跟匹配文本 |
| \\& | \& | 字符 ‘&’ |
| \\q | \q | 字符 ‘\q’ |
| \\\\ | \\ | \ |

表 9.3: POSIX 针对 ***sub()*** 与 ***gsub()*** 的规则

唯一可注意的不同的情形是最后一个：***‘\\\\’*** 被看成 ‘\\’ 并产生 ‘\’，而不是 ‘\\’。

从 3.1.4 版本开始，如果指定了 --posix 选项，gawk 按照 POSIX 的标准来（查看 ， ***在第 35 页***）。否则，它还是会使用所建议的方案，因为那是多年来的行为。

当发布了 4.0.0 版本后，gawk 维护者将 POSIX 规则为默认规则，打破了10几年来后向兼容性的价值。[[57]](#footnote-58)无须多说，这个是非常糟糕的主意，当 4.0.1 版本时，gawk 恢复了它的历史行为，当指定了 --posix 选项时，则只遵守 POSIX 规则。

gensub() 的规则则相当简单。在运行层，无论什么时候 gawk 看到一个 ‘\’，如果后面跟着一个数字，则匹配相应括号的子表达式的文件来产生输出。否则，不管 ‘\’ 后面跟着什么字符，它都出现在产生的文本中，而 ‘\’ 则不出现。如表9.4所示。

|  |  |  |
| --- | --- | --- |
| 输入 | ***gensub()*** 所见 | ***gensub()*** 产生 |
| & | & | 匹配文本 |
| \\& | \& | 字符 ‘&’ |
| \\\\ | \\ | 字符 ‘\’ |
| \\\\& | \\& | 字符 ‘\’，后面跟着匹配文本 |
| \\\\\\& | \\\& | 字符 ‘\&’ |
| \\q | \q | 字符 ‘q’ |

表 9.4: ***gensub()*** 的转义字符

由于词汇层与运行层处理的复杂性，以及 sub() 与 gsub() 的特殊情形，我们建议使用 gawk 与 gensub() 来执行替换操作。

### 9.1.4 输入输出函数

下面的函数与 I/O 相关。可选择的参数放在方括号中（[]）：

***close(filename [， how])***

关闭用于输入或者输出的文件 filename。另外，参数也可以是一个用来创建并程或者用于重定向输出，或者从管道中输入的 Shell 命令。如果参数是这个，则会关闭相应的并程与管道。查看 ，***在第 108 页***，获取更多信息。

当关闭一个并程时，有时先关闭两路管道中的一端然后再关闭另一端非常有用。这通过向 close 传递第二个参数来完成。这第二个参数（how）要是下面两个串中的一个，即 “to” 或者 “from”，以表示关闭管道的哪一端。字串的大小写是无关的。查看 ，***在第 304 页***，在里面更详细地讨论了这个特性，并且还举了一个例子。

注意 close() 的第二个参数是 gawk 扩展，在兼容模式下是不可用的（查看 ， ***在第 35 页***）。

***fflush([filename])***

将与 filename 相关的缓存输出刷出，filename 要么是一个打开的用于写的文件，或者用于重定向输出的管道，或者并程。很多的工具程序都会缓存它们的输出（即它们会将写到磁盘的文件或者屏幕的信息保存，并等待足够的时间以值得将这些数据输出到输出设备中）。这比起每次准备好一点数据就写要高效得多。但是，有时很有必要强制程序要刷出它的缓存（即将信息写到它的目的地，就是缓存还没有满）。这就是 fflush() 函数的目的——gawk 也会缓存它的输出，而 fflush() 函数强制 gawk 刷出它的缓存。

Brian Kernighan 在 1992 年 4 月将 fflush() 加入到他的 awk 中。二十年过去了，它现在成为了一个通过扩展。2012 年 12 月，它被接受包含到 POSIX 标准中。可以查看 Austin Group 网站来了解。

POSIX 标准化了 fflush 下面几个方面：如果没有参数，或者参数是空串（“”），则 awk 刷出所有已经打开的文件与管道的缓存。

提示：在 4.0.2 之前的版本，如果没有参数，gawk 会只刷出标准输出，而为空串时，才会刷出所有的输出文件与管道。为了与 Brian Kernighan 的 awk 兼容，则对其作了更改，并希望在 POSIX 中对这个特性的标准化可以更加容易（实际上也确实是这样）。

对于 gawk，你可以使用 ‘fflush("/dev/stdout")’ 来只刷出标准输出。

如果缓存被成功刷出，则 fflush() 返回 0，否则，它会返回一个非零值。（gawk 返回 -1。）在所有的缓存都刷出的情况下，当只有所有的缓存都成功刷出后才会返回 0。否则会返回 -1，并对有问题的文件名发送告警。

如果你刷出的文件或者管道是打开用于读取的（如 getline），或者 filename 不是打开的文件，管道或者并程，gawk 也会发出一个告警。这种情况下，fflush() 也返回 -1。

|  |
| --- |
| 交互式与非交互式缓存  多说一点，如果你的程序是交互式的（就是与坐在键盘前的用户进行通信），[[58]](#footnote-59)缓存问题有时会非常让人疑惑。  交互式的程序一般都是行缓存它们的输出（即它们一行行地写输出）。非交互式的程序直到缓存满后才会输出，而这可能包含很多行输出。下面是其差异的例子：  **$ *awk* ’{ print $1 + $2 }’**  **1 1**  **-| 2**  **2 3**  **-| 5**  **Ctrl-d**  输出的每一行是立即打印的。与下面这个例子的行为进行比较：  **$ *awk* ’{ print $1 + $2 }’ | cat**  **1 1**  **2 3**  **Ctrl-d**  **-| 2**  **-| 5**  在这里，直到键入 Ctrl-d 后才有输出打印，因此它将所有的缓存，然后一次性发送到管道输出到 cat。 |

***system(command)***

执行操作系统命令 command 并返回到 awk 程序中。同时返回命令的退出状态。例如，如果下面的代码片段放在你的 awk 程序中：

END {

system("date | mail -s 'awk run done' root")

}

当 awk 程序完成处理输入，并且开启输入结束处理流程时，会向系统管理员发送邮件。

注意重定向 print 或者 printf 到管道一般来就就够来完成你的工作。如果你需要运行许多命令，则简单地将它们通过管线打印到 Shell 更高效：

while (more stuff to do)

print command | "/bin/sh"

close("/bin/sh")

但是，如果你的 awk 程序是交互式的，system() 用于大型的自包含的程序就非常有用，比如 Shell 或者编辑器。一些操作系统不能实现 system 函数。system() 在不其不支持的情况下生成一个致命错误。

注意：当指定了 --sandbox 选项，system() 函数会被禁用（查看 ，***在第 35 页***）。

|  |
| --- |
| 通过 system() 来控制输出缓存  fflush() 函数提供了显式的控制输出单独文件或者管道缓存的方式。但是，对于许多的旧的 awk 实现却是不可移植的。一个可用的方法就是通过调用 system() 函数，并给其提供一个空串作为其参数来刷出输出缓存：  **system("") # flush output**  gawk 将这种使用 system() 函数的方式作为一种特殊方式来对待，并且聪明地不用空命令来运行一个 Shell（或者其他的命令解释器）。因此，在 gawk 中，这种范式不仅很有用，同时也非常高效。尽管这个方法也会在其他的 awk 实现中可工作，但是却不是必须要避免开启一个不必要 Shell。（其他的实现可能只刷出与标准输出关键的缓存，而必是所有的缓存输出。）  如果你考虑一下一个程序员的预期，system() 会刷出所有的挂起的输出看起来就有意思了。下面的程序：  **BEGIN {**  **print "first print"**  **system("echo system echo")**  **print "second print"**  **}**  必须打印：  **first print**  **system echo**  **second print**  而不是：  **system echo**  **first print**  **second print**  如果 awk 不在调用 system() 前刷出它的缓存，你就可以会看到后面（不希望看到的）输出。 |

### 9.1.5 时间函数

awk 程序常常用来处理带时间戳信息的日志文件，以提示特定的日志记录是何时被写入的。许多的程序将日志的时间信息记录成 time() 系统调用返回的格式，即是从特定时刻开始的秒数。在 POSIX 兼容的系统上，它是从 1970-01-01 00:00:00 UTC 时间开始以来的秒数，没有包含润秒。[[59]](#footnote-60)所有知名的 POSIX 兼容的系统支持从 0 到 232-1 之间的时间戳，这个时间可以表示到 2038-01-19 03:14:07 UTC 的时间。很多的系统支持更宽的时间戳范围，包括负的时间戳，以表示 epoch 之间的时间。

为了更容易处理这样的带时间戳的日志文件并且产生有用的报表，gawk 提供了下面的函数来处理时间戳。它们都是 gawk 的扩展，并没有在 POSIX 标准中指定。[[60]](#footnote-61)但是，mawk 的新近的版本（查看 ，***在第 435 页***）也支持这些函数。可选的参数被包含在方括号中（[]）：

***mktime(datespec)***

将 datespec 转换为与 systime() 返回的相同格式的时间戳。它与在 ISO C 中同名的函数相似。datespec 参数是一个 "YYYY MM DD HH MM SS [DST]" 格式的字串。字串包含6个或者7个数字，以相应地表示年的全的世纪年，从1到12的月，1到31的日，一天中从0 到23的小时，0到59分钟，0 到60秒，[[61]](#footnote-62)以及一个可选的夏令时标志。

这些数字的值不是必须在所指定的范围内，例如 -1 小时表时午夜前的 1 小时。初始的格林日历的0为始，0年表示1年之前，而 -1 年表示在 0 年前。时间也可以假设为本地时间。如果夏令时标志为正，则时间假设为夏令时时间，如果为0，则时间假设为标准时间，如果为负（默认值），mktime() 会试图确定特定的时间是否是夏令时时间。如果 datespec 没有包含足够的元素或者结果的时间超出了范围，mktime() 返回 -1。

***strftime([format [， timestamp [， utc-flag] ] ])***

根据字串格式的内容，将指定的时间戳进行格式化，并返回结果。这与 ISO C 中同名的函数的功能类似。如果提供了 utc-flag 参数，并且是非0，或者非空，则值会格式化为 UTC（统一协调时间，之前为 GMT，格林尼治标准时间）。否则，值将被格式化为本地时区时间。timestamp 的格式与 systime() 返回的值的格式一样。如果没有指定 timestamp 参数，gawk 则使用当天的当时时间作为 timestamp。如果没有格式参数，strftime() 则使用 PROCINFO["strftime"] 的值作为格式字串（查看 ，***在第 155 页***）。默认的字串值为 “%a %b %e %H:%M:%S %Z %Y”。这个格式字串产生的输出与 date 工具产生的输出一样。你可以给 PROCINFO["strftime"] 赋一个新值，以改变默认格式，查看下面的列表来获取格式指示符。

***systime()***

返回从系统 epoch 开始当前时间的秒数。在 POSIX 系统中，这是从 1970-01-01 00:00:00 UTC 开始的秒数，不包含润秒。在其他系统中可能是不同的值。

systime() 函数可以让你比较日志文件的时间戳与当前的时间。特别是，很容易确定相应的记录是多久之前记录的。它也可以让你用从 epoch 开始格式的时间来产生日志记录。

mktime() 函数可以让你将文本形式的日期与时间的表示转换为一个时间戳。因此很容易来作日期与时间的前后比较，尤其是当处理的日期与时间数据来自于外部，比如一个日志文件。

strftime() 函数可以让你很容易将一个时间戳转换成一个人可读的信息。这有点像 printf 函数（查看 ，***在第 184 页***）的特点一样，在这个函数中将非格式化的范围字符一字不变地加入到返回串中。将日期与时间值替换为格式串中所指定的格式规范。

strftime() 使用的是 1999 ISO C 的标准，[[62]](#footnote-63)可以支持下面的日期格式规范：

%a 本地语言设置中缩写的星期名。

%A 本地语言设置中的完整的星期名。

%b 本地语言设置中缩写的月份名。

%B 本地语言设置中完整的月份名。

%c 本地语言设置中 “恰当的” 日期与时间表示。（这在 “C” 语言设置中为 ‘%A %B %d %T%Y’）

%C 当前年的世纪部分。这是年除以 100，并向下整数进行截取得到的数值。

%d 月中的10进制日期，（01-31）。

%D 与指定 ‘%m/%d/%y’ 相同。

%e 月中的日期，如果只有一位，则进行空格补全。

%F 与指定 ‘%Y-%m-%d’ 等同。这是 ISO 8601 的格式。

%g ISO 8601 标准中年与 100 的模的周数值，格式为十进制数（00-99）。例如， 2012年的1月1日， 是 2011 年的 53 周。因此，年在 ISO 8601 中的周数在 2011 年，就算年已经是 2012年了。相似的，2012 年 12 月 31 日是 2013 年的第 1 周。所以，年的 ISO 周是 2013 年，就算年还是 2012 年。

%G ISO 周数的完整年，为十进制数。

%h 与 ‘%b’ 等同。

%H 十进制的小时（24 小时制，00-23）。

%I 十进制的小时（12 小时制，00-12）。

%j 一年中的天数的十进制数（001-366）。

%m 月份的十进制数（01-12）。

%M 分钟的十进制数（00-59）。

%n 换行符（ASCII LF）。

%p 与本地语言设置中 AM/PM 的等同，与 12 小时间制关联的表示法。

%r 本地语言设置中 12 小时制的时钟时间。（这在 “C” 语言设置中为 ‘%I:%M:%S %p’。）

%R 与 ‘%H:%M’ 等同。

%S 十进制的秒数（00-60）。

%t TAB 字符。

%T 与 ‘%H:%M:%S’ 等同。

%u 十进制的星期数（1-7）。星期一为1。

%U 一年中的十进制的周数（0-53，第一个星期日作为第一周的第一天）。

%V 一年中的十进制的周数（01-53，第一个星期一作为第一周的第一天）。确定周数的方法由 ISO 8601 中所指定的一样。（即：如果包含1月1日星期中有在新的一年中有4天，或者更多的天数，则就是算成一周，否则前一年就是53周，下一周就算成第一周。）

%w 十进制的星期几（0-6）。星期天是 0 天。

%W 一年的周数（00-53，第一个星期一作为第一周的第一天）。

%x 本地语言设置中 “恰当的” 日期表示。（对于 “C” 的语言设置则为 ‘%A %B %d %Y’。）

%X 本地语言设置中 “恰当的” 时间表示。（对于 “C” 的语言设置则为 ‘%T’。）

%y 年与 100 的模的十进制数（00-99）。

%Y 年的十进制全写（如 2015）。

%z 时区差，用 ‘+HHMM’ 格式表示（即格式需要产生 RFC822/RFC 1036 的日期头）。

%Z 时区名或者是其缩写，如果时区不能确定则没有字符。

%Ec %EC %Ex %EX %Ey %EY %Od %Oe %OH

%OI %Om %OM %OS %Ou %OU %OV %Ow %OW %Oy

在规范中只使用第二个字符的“替代表示”（‘%c’， ‘%C’，等等）。[[63]](#footnote-64)（这个设施是与 POSIX 的 date 工具兼容的。）

%% 字面的 ‘%’。

如果转换指定符不是上面所列之一，行为就是未定义的。[[64]](#footnote-65)

对于没有完全与标准兼容的系统，gawk 提供一个来自于 GNU C 库的 strftime() 的一个复本。它支持上面所列的所有格式指定符。如果那个版本用于编译 gawk（查看 ***，在第 419 页***），则下面的额外的格式指定符也可用：

%k 24 小时制的十进制小时（0-23）。一个位的数值会用空格进行填补。

%l 12 小时制的十进制小时（1-12）。一个位的数值会用空格进行填补。

%s 十进制的时间戳，从 epoch 开始的秒数。

另外，替代的表示可以被识别到，但是只使用它们的常规表示。

下面的例子是 POSIX date 工具的 awk 实现。正常情况下， date 工具以一种大家都了解的格式打印当前的日期与时间。但是，如果你提供了前面带 ‘+’ 的参数，date 则会复杂非格式指定符到标准输出中，并且按照串中所格式指定符来解释当前的时间。如：

**$ date ’+Today is %A， %B %d， %Y.’**

**-| Today is Monday， September 22， 2014.**

下面是 gawk 版本的 date 工具。它有一个壳为 “wrapper” 来处理 -u 选项，这个选项要求 date 运行时假设时区是 UTC：

**#! /bin/sh**

**#**

**# date --- approximate the POSIX 'date' command**

**case $1 in**

**-u) TZ=UTC0 # use UTC**

**export TZ**

**shift ;;**

**esac**

**gawk ’BEGIN {**

**format = PROCINFO["strftime"]**

**exitval = 0**

**if (ARGC > 2)**

**exitval = 1**

**else if (ARGC == 2) {**

**format = ARGV[1]**

**if (format ~ /^\+/)**

**format = substr(format， 2) # remove leading +**

**}**

**print strftime(format)**

**exit exitval**

**}’ "$@"**

### 9.1.6 位操作函数

*我可以向你解释，但是没办法帮你理解。*

*—Anonymous*

很多的语言提供了在两个整数之间执行位操作的能力。换句话说，操作在两个在操作数上的连续位对进行操作。三个常用的操作为按位 AND，OR 与 XOR。操作在下表中描述。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 操作数 | AND | | OR | | XOR | |
| 0 | 1 | 0 | 1 | 0 | 1 |
| 0  1 | 0  0 | 0  1 | 0  1 | 1  1 | 0  1 | 1  0 |

Table 9.5: 位操作

如你所见， AND 操作的结果是只有当两个位都为1 时才为1。OR 的操作结果是当两个位有一个为1则为1。XOR 的操作结果是有一个位1则为1，而不是两个为1。下一个参数是“补”（取反），1 的补为 0，0 的被为 1。所以，这个操作 “翻转” 所给值的所有位。

最后，还有两个常用的操作将位从左或者从右进行平移。如，如果你有值串 ‘***10111001***’，然后你将它向右平移3位，最终得到的值为 ‘***00010111***’。[[65]](#footnote-66)如果你又用同样的值 ‘***10111001’***’ 进行左移三位，则得到 ‘***11001000***’。下面的列表中描述了 gawk 中内建的实现位操作的函数。可选的参数包含在方括号中（[]）：

***and(v1， v2 [， ...])***

返回参数的 AND 结果。至少有两个参数。

***compl(val)***

返回 val 的位“补”结果。

***lshift(val， count)***

返回 val 左移 count 位的值。

***or(v1， v2 [， ...])***

返回参数的 OR 结果。至少有两个参数。

***rshift(val， count)***

返回 val 右移 count 位的值。

***xor(v1， v2 [， ...])***

返回参数的 XOR 结果。至少有两个参数。

对所有这些函数，产生双精度的浮点数会被转换为最宽的 C 的无符号整数，然后再执行位操作。如果结果不能被 C 的 double 精确表示，其前导的非零位会一位位地移除，直到可以被精确表示。结果再转换加 double 类型。（如果你不理解这个转换图，也不要担心。）

这里是一个用户自定义的函数（查看 ， ***在第 204 页***），其中解释了这些函数的使用：

**# bits2str --- turn a byte into readable ones and zeros**

**function bits2str(bits， data， mask)**

**{**

**if (bits == 0)**

**return "0"**

**mask = 1**

**for (; bits != 0; bits = rshift(bits， 1))**

**data = (and(bits， mask) ? "1" : "0") data**

**while ((length(data) % 8) != 0)**

**data = "0" data**

**return data**

**}**

**BEGIN {**

**printf "123 = %s\n"， bits2str(123)**

**printf "0123 = %s\n"， bits2str(0123)**

**printf "0x99 = %s\n"， bits2str(0x99)**

**comp = compl(0x99)**

**printf "compl(0x99) = %#x = %s\n"， comp， bits2str(comp)**

**shift = lshift(0x99， 2)**

**printf "lshift(0x99， 2) = %#x = %s\n"， shift， bits2str(shift)**

**shift = rshift(0x99， 2)**

**printf "rshift(0x99， 2) = %#x = %s\n"， shift， bits2str(shift)**

**}**

当执行时，产生如下输出：

**$ gawk -f testbits.awk**

**-| 123 = 01111011**

**-| 0123 = 01010011**

**-| 0x99 = 10011001**

**-| compl(0x99) = 0xffffff66 = 11111111111111111111111101100110**

**-| lshift(0x99， 2) = 0x264 = 0000001001100100**

**-| rshift(0x99， 2) = 0x26 = 00100110**

**bits2*str()*** 函数将一个二进制数转换为字符串。给 mask 初始化为 1 来创建一个这样的二进制数据，其最右边的位被设置为 1。使用这个 mask，函数重复地检查最右边的位。将 mask 与值进行 AND 操作，可以显示出最右边的位是 1 还是0。如果是 1，就会在字串的前面连接一个 “1”，否则连接一个 “0”。值然后向右称 1 位，直到没有更多的 1 存在。

如果初始值为 0，它会简单地返回 “0”。否则最后，会将值进行填补以产生 8 位数量的整倍数。在现代的计算机中这是一种典型的处理。在 BEGIN 中的主代码规则显示了十进制与八进制数据的同一个值的不同（查看 ），然后展示了 ***compl()***， ***lshift()***， 与 ***rshift()*** 函数的结果。

### 9.1.7 获取类型信息

gawk 提供了一个函数来让你可以区分数组与标量变量。这在遍历数组之数组的每一个元素的时候是非常有用的（查看 ， ***在第 178 页***）。

***isarray(x)***

如果 x 是一个数组，则返回 true，否则返回 false。

isarray() 主要用在两种环境中。第一种是当遍历多维数组时，你可以测试某个元素是否是数组。第二个场景是在用户自定义函数中（还未讨论到，查看 ， ***在第 204 页***），用来测试参数是否为数组。

提示：在全局范围内使用 isarray() 来测试变量并没有含义。因为你就是写程序的人，你当然知道你的变量是不是数组，实际上，由于 gawk 的工具方式，如果你变量的名字在 isarray() 之前还没有用到，gawk 会将其转换成一个标量。

### 9.1.8 字串翻译函数

gawk 提供了一些设施用来国际化 awk 程序。这些包含了下面列表中描述的函数。这些的描述只是目的性的。查看 ，***， 在第 311 页***，来查看全部的说明。可选的参数放在方括号中（[]）：

***bindtextdomain(directory [， domain])***

设置目录，当不会或者不能定位到“标准”的位置时，gawk 会在这个目录里查找信息翻译文件（如在测试期间）。它返回绑定域的目录。

默认的域是 TEXTDOMAIN 的值。如果目录是一个空串（“”），则 ***bindtextdomain()*** 返回给定域的当前绑定。

***dcgettext(string [， domain [， category] ])***

返回在文本域 domain 中，对于本地语言设置 category 的串的翻译串。默认的域值是当前 TEXTDOMAIN 的值。category 的默认值是 “LC\_MESSAGE”。

***dcngettext(string1， string2， number [， domain [， category] ])***

返回在分类 category 的文本域 domain 中，用于进行 string1 与 string2 翻译的数目的复数形式。string1 是某个消息的英语单一变体，string2 同一个消息的英语复数形式。domain 的默认值就是 TEXTDOMAIN 的当前值。category 的默认值是 “LC\_MESSAGE”。

## 9.2 用户自定义函数

复杂的 awk 程序常常通过定义自己的函数来进行简化。用户自定义函数也可以像内置函数一样调用（查看 ， ***在第 133 页***），但是定义他们则是你的自由（即告诉 awk 你所要做的事情）。

### 9.2.1 函数定义语法

*说 awk 本地变量的定义语法惊人的糟糕是很公正的。*

*—Brian Kernighan*

函数的定义可以出现在 awk 程序规则之间的任何地方。所以，awk 程序的一般形式是被扩展为包含一系列的规则与用户自定义函数定义。没有必须将函数定义放在所有使用函数的地方之间。这是因为 awk 读取程序是在开始执行任何部份之间。

某个函数名为 name 的定义看起来如下：

**function name([parameter-list])**

**{**

**body-of-function**

**}**

在这里，name 是要定义函数的名字。有一个有效的函数名就像一个有效的变量名一关：一系列的字母、数字以及下划线，同时不以数字开头的序列。因此，只有 52 个大写跟小写的英文字母可以用于函数名。

parameter-list 是一个可选的函数参数，以及本地变量名的列表，通过逗号分隔。当函数被调用的时候，参数名用于在调用时传递的参数值。

函数不可以用两个相同名字的参数，也不可以与函数的名字同名。

注意：根据 POSIX 标准，函数参数不可以与特殊的预定义变量同名（查看 ， ***在第 155 页***），函数参数也不可以与另一个函数名同名。

并不是所有的 awk 版本都强制这些限制。gawk 总是强制第一条限制。在指定 --posix 选项时（查看 ， ***在第 35 页***），它也强制第二个限制。

如果被参考的地方需要一个字串值的时候，本变量表现像一个空串，如果参考到的地方需要一个数值的时候，就表现为 0。这与常规的变量在还没有被赋值时的表现一样。（要更多地理解本地变量，查看 ， ***在第 213 页***）

body-of-function 由 awk 的语句组成。这是定义的最重要的部分，因此它定义了函数实际要做的事情。参数名存在给了函数体与参数进行对话的方式，本地变量的存在可以使函数体来保存临时值。

参数名与本地变量名没有语法上的区别。相反，调用函数时提供的参数个数决定了有多少个参数变量。因此，如果给了三个参数值，parameter-list 中的头三个名字就是参数，而剩下的就是本地变量。

规则是这样的，如果参数的个数与所有的函数的所有调用都不一样，在 parameter-list 中的一些名字有时是参数，而在其他时间则是本地变量。另一个思考这个问题的方式是，被省略的参数的默认值为空串。通常情况下，当你写一个函数的时候，你知道你要使用的名字数，以基作为参数，而有相应的个数用来作本地变量。将一些额外的空格放在参数与本地变量之间可以很方便地来加以区分，这可以作为如何来使用函数的文档。

在函数体的执行过程中，参数与本地变量会被隐藏，或者遮盖掉，程序后面所有使用到的同名的变量。被遮盖掉的变量在函数定义中是不可访问的，因此没有方法在它们的名字被用于参数或者本地变量时进行命名。所以其他的在 awk 程序中使用的变量可以在函数体中被参考，或者设置。参数与本地变量只持续到函数体执行完成。只要函数体一完成，你就再一次可以访问那些在函数执行时被遮盖掉的变量。

函数体可以包含调用函数的表达式。他们甚至可以调用函数本身，直接地或者通过另一个函数来调用。当发生了这样的事情时，我们称函数是递归的。函数调用其本身的动作称为递归。

所有的内建函数返回一个值给它他的调用者。用户自定义函数也可以这么做，只需要使用 return 语句，它在 ， ***在第 212 页***，进行详细的描述。本节中后续的很多例子都使用 return 语句。

在很多的 awk 实现中，包括 gawk，关键词 function 可以简写为 func。（c.e.）。但是 POSIX 只指定了使用关键词 function。实际上有一些实践上的隐义。如果 gawk 是 POSIX 兼容模式（查看 ， ***在第 35 页***），则下面的语句不会定义一个函数：

**func *foo()* { a = sqrt($1) ; print a }**

相反，它定义了这样的一个规则：对每一个记录，连接变量 func 的值与函数 ‘foo’ 的返回值。如果结果字串是非空的，动作就会被执行。这可能不是所想要的。（awk 接收这样的输入为语法有效，因此函数可以在它们在 awk 程序中定义之前被调用。[[66]](#footnote-67)）

为了保证你的 awk 程序是可移植的，在定义函数时，总是使用关键字 function。

### 9.2.2 函数定义范例

下面是一个用户自定义函数的例子，函数名字叫 myprint()，该函数取得一个数值，并以指定的格式打印它：

**function myprint(num)**

**{**

**printf "%6.3g\n"， num**

**}**

为了解清楚，下面是一个使用 myprint() 函数的 awk 规则：

**$3 > 0 { myprint($3) }**

此程序用我们特定的格式打印所有输入中第三个域是正数的域。因此，当有下面的输入时：

**1.2 3.4 5.6 7.8**

**9.10 11.12 -13.14 15.16**

**17.18 19.20 21.22 23.24**

用我们的程序格式，则会打印：

**5.6**

**21.2**

下面这个函数删除数组中的所有元素（回忆一下额外的空格来标识本地变量列表）：

**function delarray(a， i)**

**{**

**for (i in a)**

**delete a[i]**

**}**

当与数组一起执行时，常常需要删除数组中的所有元素，并开始一个新的元素列表（查看 ， ***在第 175 页***）。为了不在每个要清除数组的地方都要重复地写这样的循环，你的程序就只调用 delarray()。（这保证了移植性。 ‘delete array’ 的使用用来删除整个数组是最新近[[67]](#footnote-68)才加入到 POSIX 标准中的。）

下面是一个递归函数的例子。它取一个字串作为输出参数，并返回逆序字串。递归函数必须有一个测试条件来停止递归。在这种情况下，在输入串为空时就结束递归：

**function rev(str)**

**{**

**if (str == "")**

**return ""**

**return (rev(substr(str， 2)) substr(str， 1， 1))**

**}**

如果这个函数在一个名字为 rev.awk 的文件中，我们可以这么来测试：

**$ echo "Don’t Panic!" |**

**> gawk -e ’{ print rev($0) }’ -f rev.awk**

**-| !cinaP t‘noD**

C 的 ctime() 函数输入一个时间戳，并返回一个用常用的形式格式化的字串。下面的例子使用内置函数 strftime()（查看 ， ***在第 197 页***）来创建一个 awk 版本的 ctime()。

**# ctime.awk**

**#**

**# awk version of C ctime(3) function**

**function ctime(ts， format)**

**{**

**format = "%a %b %e %H:%M:%S %Z %Y"**

**if (ts == 0)**

**ts = systime() # use current time as default**

**return strftime(format， ts)**

**}**

你可能认为 ctime() 可以使用 PROCINFO["strftime"] 来作为格式串。但那会是个错误，因为 ctime() 假设返回的是标准的格式，而用户级的代码可能已经更改了 PROCINFO["strftime"]。

### 9.2.3 调用用户自定义函数

调用一个函数会导致那个函数被执行。调用调用是一个表达式，其值就是函数返回的值。

#### 9.2.3.1 如何调用函数

函数调用包含函数名，后面再跟着在括号中的参数。awk 表达式就是用把参数写在函数调用中。每次调用执行时，这些表达式会先求值，然后值为成为实际的参数。例如，下面是一个用三个参数调用 foo() 函数例子（第一个参数是一个字串连接）：

**foo(x y， "lose"， 4 \* z)**

注意：在函数名了参数列表的开括号之间的空白（空格与 TABs）是不允许的。如果你不小心写了个空白在里面，awk 可能将你的意思理解成来连接一个变量与括号表达式的值。但是，最后它会注意到你使用了一个函数名，而不是一个变量名，因此会报一个错误。

#### 9.2.3.2 控制变量作用域

不像其他的很多语言，在 awk 没有办法使一个变量只在 {...} 块中有作用，但是你可以使一个变量成为一个函数的本地变量。这么做是一个很好的实践，在任何时候在函数中使用一个变量时，都将其置成本地变量。

为了使一个变量成为函数的本地变量，只需要简单地在实际的参数（查看 ， ***在第 204 页***）的后面声明就可以。看下面的例子，变量 i 是一个全局变量，即可以被 foo() 使用，也可以被 bar() 使用：

**function bar()**

**{**

**for (i = 0; i < 3; i++)**

**print "bar’s i=" i**

**}**

**function foo(j)**

**{**

**i = j + 1**

**print "foo’s i=" i**

**bar()**

**print "foo’s i=" i**

**}**

**BEGIN {**

**i = 10**

**print "top’s i=" i**

**foo(0)**

**print "top’s i=" i**

**}**

执行这个脚本，则会产生下面的输出，因为 i 在 foo() 函数与 bar() 函数中以及在最顶层都指向同一个变量实体：

**top’s i=10**

**foo’s i=1**

**bar’s i=0**

**bar’s i=1**

**bar’s i=2**

**foo’s i=3**

**top’s i=3**

如果你想 i 成为 foo() 与 bar() 的本地变量，近下面这么做（在 i 之前的额外空格是一种编码习惯，以用来表示 i 是一个本地变量，而不是一个参数）：

**function bar( i)**

**{**

**for (i = 0; i < 3; i++)**

**print "bar’s i=" i**

**}**

**function foo(j， i)**

**{**

**i = j + 1**

**print "foo’s i=" i**

**bar()**

**print "foo’s i=" i**

**}**

**BEGIN {**

**i = 10**

**print "top’s i=" i**

**foo(0)**

**print "top’s i=" i**

**}**

执行修正过的脚本产生如下输出：

**top’s i=10**

**foo’s i=1**

**bar’s i=0**

**bar’s i=1**

**bar’s i=2**

**foo’s i=1**

**top’s i=10**

除了标量（字吕与数值）外，你也可以有本地数组。通过将某个变量名用作数组，awk 就会将其当为数组，并且是函数的本地数组。另外，递归调用会创建新的数组。考虑下面的例子：

**function some\_func(p1， a)**

**{**

**if (p1++ > 3)**

**return**

**a[p1] = p1**

**some\_func(p1)**

**printf("At level %d， index %d %s found in a\n"，**

**p1， (p1 - 1)， (p1 - 1) in a ? "is" : "is not")**

**printf("At level %d， index %d %s found in a\n"，**

**p1， p1， p1 in a ? "is" : "is not")**

**print ""**

**}**

**BEGIN {**

**some\_func(1)**

**}**

当执行时，程序产生下面的输出：

**At level 4， index 3 is not found in a**

**At level 4， index 4 is found in a**

**At level 3， index 2 is not found in a**

**At level 3， index 3 is found in a**

**At level 2， index 1 is not found in a**

**At level 2， index 2 is found in a**

#### 9.2.3.3 函数变量中的传参与传引用

在 awk 中，当你声明一个函数，没有办法显式地声明参数是通过传值还是传引用。

相反，传递规范是在函数调用时根据下面的规则来动态确定的：如果参数是一个数组变量，则是通过传引用。否则，参数就是通过传值。

当调用参数对参数进行传值，意是是说，它给了参数一个复本。调用者使用变量来作用参数的表达式，但是被调用的函数却并不知道这回事——它只知道参数具有什么样的值。例如，如果你写了下面的代码：

**foo = "bar"**

**z = myfunc(foo)**

则你不要将 myfunc() 的参数想象为 “foo 变量”。相反，则应该将参数想为字串值 “bar”。如果函数 myfunc() 改变了它本地变量的值，它对于其他的变量没有任何影响。所以，如果 myfunc() 执行：

**function myfunc(str)**

**{**

**print str**

**str = "zzz"**

**print str**

**}**

来改变它的参数变量 str，它不会更改调用者 foo 的值。在调用 myfunc() 中 foo 的角色就在它的值（“bar”）被计算完后结束了。如果 str 也在 myfunc() 之外存在，则函数体不能更改外面的值，因为它已经在 myfunc() 执行时被遮盖掉了，不能从那个地方被 看到，也不能被改变。

但是，当调用函数传递的是一个数组时，就不会执行复制。相反，数组变量本身会被函数进行直接操作。这个在术语上一般叫作引用。在函数体内对于数组的更改在函数外部也是可见的。

提示：如果你看不到你所做的事情，在函数中改变数组参数会非常危险。如：

**function changeit(array， ind， nvalue)**

**{**

**array[ind] = nvalue**

**}**

**BEGIN {**

**a[1] = 1; a[2] = 2; a[3] = 3**

**changeit(a， 2， "two")**

**printf "a[1] = %s， a[2] = %s， a[3] = %s\n"，**

**a[1]， a[2]， a[3]**

**}**

会打印 ‘a[1] = 1， a[2] = two， a[3] = 3’，因为 changeit() 会将 “two” 存储到 a 的第二个元素中。

有一些 awk 实现可以让你调用还没有定义的函数。当程序实际上试图调用时，它只在运行时报告问题。如：

**BEGIN {**

**if (0)**

**foo()**

**else**

**bar()**

**}**

**function bar() { ... }**

**# note that ‘foo’ is not defined**

因为 if 语句决不会为 true，foo() 没有定义也不会是个真正的问题。但是，一般情况下，如果程序调用一个没有定义的函数确实是个问题。

如果指定了 --lint 参数（查看 ， ***在第 35 页***），gawk 会报告调用用了未定义的函数。

如果你在用户自定义函数中使用了 next 语句或者 nextfile 语句（查看 ，***在第 153 页***，以及 ， ***在第 154 页***），一些 awk 实现会产生一个运行时错误。gawk 则没有这个限制。

### 9.2.4 return 语句

如在之前的几个例子中所见到的，用户自定义的函数体内可以包含一个 return 语句。这个语句将控制返回到 awk 程序调用的地方。也可以用于返回一个值，以在 awk 程序的后面来使用。看来起其形式如下：

**return [expression]**

expression 的部分是可选的。极有可能是因为疏忽，POSIX 没有定义如果 expression 被省略的情况下该返回什么样的值。从技术上说，这会使返回的值是未定义的，因此也是不可预测的。但是在实践中，所有的 awk 版本都是简单地返回一个空串，在需要数值的上下文则为0。

return 语句中没有 expression 被假设为每一个函数定义的完成处。所以，如果控制到达了函数体的尾部，函数技术性地返回一个不可预测的值。在实践中，它返回一个空串。awk 在你使用这样的函数返回值是不会产生告警。

有时，你想写的函数你关注的是它所做的，而不是它所返回的。这样的函数与 C 、C++，或者 Java以及 Ada 中的过程中的 void 对应。所以，不返回任何值是恰当的，只需要简单地记住不要使用这些参数的返回值即可。

下面的例子是一个返回数组元素中最大的数值的用户自定义函数：

**function maxelt(vec， i， ret)**

**{**

**for (i in vec) {**

**if (ret == "" || vec[i] > ret)**

**ret = vec[i]**

**}**

**return ret**

**}**

你用一个参数来调用 maxelt()，这个参数是一个数组名。本地变量 i 以及 ret 都不是用来作参数的，没有什么可以阻止你传递多于一个参数到 maxelt() 中，但那会产生奇怪的结果。在参数列表中， i 之前额外的空格表示 i 与 ret 都是本地变量。你在定义函数时应该使用这些的习惯。

下面的程序会使用 maxelt() 函数。它装入一个数组，然后调用 maxelt() 然后报告数组中的最大元素：

**function maxelt(vec， i， ret)**

**{**

**for (i in vec) {**

**if (ret == "" || vec[i] > ret)**

**ret = vec[i]**

**}**

**return ret**

**}**

**# Load all fields of each record into nums.**

**{**

**for(i = 1; i <= NF; i++)**

**nums[NR， i] = $i**

**}**

**END {**

**print maxelt(nums)**

**}**

如果有下面的输入：

**1 5 23 8 16**

**44 3 5 2 8 26**

**256 291 1396 2962 100**

**-6 467 998 1101**

**99385 11 0 225**

则程序报告（可预测的）99385 是数组中最大的值。

### 9.2.5 函数与其在变量类型上的影响

awk 是一种非常随意的语言。awk 有可能办法告诉你一个标识符是一个标量变量还是一个数组，直到运行时才会知道。这是一个用来说明的例子程序：

**function foo(a)**

**{**

**a[1] = 1 # parameter is an array**

**}**

**BEGIN {**

**b = 1**

**foo(b) # invalid: fatal type mismatch**

**foo(x) # x uninitialized， becomes an array dynamically**

**x = 1 # now not allowed， runtime error**

**}**

在这个例子中，第一次调用 foo() 会产生一个致命错误，所以 awk 不会报告第二个错误。如果你将这个调用注释掉，则 awk 会报第二个错误。

一般来说，这种事不是什么大问题，但是值得注意。

## 9.3 间接函数调用

这一节描述了一种高级的，特定于 gawk 的扩展。

经常的，你希望在直接运行时才确定你要调用的函数。例如，你有不同类型的记录，每一个都要有不同的处理方式。

正常情况下，你不得不用一系列的 if-else 语句来决定要调用哪一个函数。通过使用间接函数调用，你可以指定函数的名字为一个字串，然后来再调用函数。让我们来看下一例子：

假设你有一个文件，里面有你的选择的课程的测试成绩，并且你希望取得这些成绩的总合与平均值。第一个域是一个课程名。后面的域是函数调用要处理的数组，直到一个域标识 “data:” 为止，在标识后面到记录的末尾，都是一系列的测试成绩。

这里最开始的文件：

**Biology\_101 sum average data: 87.0 92.4 78.5 94.9**

**Chemistry\_305 sum average data: 75.2 98.3 94.7 88.2**

**English\_401 sum average data: 100.0 95.6 87.1 93.4**

为了处理数据，你可能开始这么写：

**{**

**class = $1**

**for (i = 2; $i != "data:"; i++) {**

**if ($i == "sum")**

**sum() # processes the whole record**

**else if ($i == "average")**

**average()**

**... # and so on**

**}**

**}**

这样风格的程序也可以，但是看起来有点别扭。通过间接函数调用，你告诉 gawk 使用函数名称变量的值作为函数来调用。

语法非常类似于常规的函数调用：一个标识符，后面跟着一个开括号，所有的参数，然后是一个闭括号，加上一个前导的 @ 符号：

**the\_func = "sum"**

**result = @the\_func() # calls the sum() function**

这是处理前所示的数据的完整程序，使用的是间接函数调用：

**# indirectcall.awk --- Demonstrate indirect function calls**

**# average --- return the average of the values in fields $first - $last**

**function average(first， last， sum， i)**

**{**

**sum = 0;**

**for (i = first; i <= last; i++)**

**sum += $i**

**return sum / (last - first + 1)**

**}**

**# sum --- return the sum of the values in fields $first - $last**

**function sum(first， last， ret， i)**

**{**

**ret = 0;**

**for (i = first; i <= last; i++)**

**ret += $i**

**return ret**

**}**

这两个函数期望是工作于域，因此，第一个与最后一个参数表示域处理的开始与结束。否则，它们执行是期望的计算但不正确：

**# For each record， print the class name and the requested statistics**

**{**

**class\_name = $1**

**gsub(/\_/， " "， class\_name) # Replace \_ with spaces**

**# find start**

**for (i = 1; i <= NF; i++) {**

**if ($i == "data:") {**

**start = i + 1**

**break**

**}**

**}**

**printf("%s:\n"， class\_name)**

**for (i = 2; $i != "data:"; i++) {**

**the\_function = $i**

**printf("\t%s: <%s>\n"， $i， @the\_function(start， NF) "")**

**}**

**print ""**

**}**

这是处理每个记录的主过程。它打印课程名（将下划线替换成空格）。然后查找实际数据的开始，并将它保存在 start 中。代码的最后一部分是循环每一个函数名（从 $2 到 “data:” 标识），调用域所表示的名字。间接调用函数将其作为 printf 函数的一个参数。（printf 的格式串使用 “%s” 作为格式指定符，所以我们可以使用返回字串的函数，当然也包括数值。注意从间接调用返回的结果与空串进行合并，是为将其强制转换为字串值。）

这是执行程序的结果：

**$ gawk -f indirectcall.awk class\_data1**

**-| Biology 101:**

**-| sum: <352.8>**

**-| average: <88.2>**

**-|**

**-| Chemistry 305:**

**-| sum: <356.4>**

**-| average: <89.1>**

**-|**

**-| English 401:**

**-| sum: <376.1>**

**-| average: <94.025>**

可以使用间接函数调用的能力比初看起来要强大得多。C 与 C++ 语言提供了“函数指针”，那也是一个在执行时选择调用函数的机制。这个能力用得最多的是 C qsort() 函数，该函数使用有名的快速排序（查看 Wikipedia 的文档来了解更多的信息）算法来对数组进行排序。为了使用这个函数，你需要指定一个用于比较的函数指针。这个机制可以使用对任意以及任何方式对数据进行排序。

我们用 gawk 也可以做同样的事情，如：

**# quicksort.awk --- Quicksort algorithm， with user-supplied**

**# comparison function**

**# quicksort --- C.A.R. Hoare’s quicksort algorithm. See Wikipedia**

**# or almost any algorithms or computer science text.**

**function quicksort(data， left， right， less\_than， i， last)**

**{**

**if (left >= right) # do nothing if array contains fewer**

**return # than two elements**

**quicksort\_swap(data， left， int((left + right) / 2))**

**last = left**

**for (i = left + 1; i <= right; i++)**

**if (@less\_than(data[i]， data[left]))**

**quicksort\_swap(data， ++last， i)**

**quicksort\_swap(data， left， last)**

**quicksort(data， left， last - 1， less\_than)**

**quicksort(data， last + 1， right， less\_than)**

**}**

**# quicksort\_swap --- helper function for quicksort， should really be inline**

**function quicksort\_swap(data， i， j， temp)**

**{**

**temp = data[i]**

**data[i] = data[j]**

**data[j] = temp**

**}**

***quicksort()*** 函数接收数据数组，以及要进行排序的开始与结束下标（left 与 right），以及执行 “less than” 比较的函数名。之后实现 quicksort 算法。

为使用这个排序函数，我们再回到之前的例子。第一件事情就是要写一个比较函数：

**# num\_lt --- do a numeric less than comparison**

**function num\_lt(left， right)**

**{**

**return ((left + 0) < (right + 0))**

**}**

**# num\_ge --- do a numeric greater than or equal to comparison**

**function num\_ge(left， right)**

**{**

**return ((left + 0) >= (right + 0))**

**}**

num\_ge() 函数在执行降序排序时需要，当执行 “less than” 的测试时，其执行的刚才是相反的（即大于或者等于），这会产生一个降序。

下面则是排序函数。这是一个参数化了 starting 与 ending 域数值与比较函数的函数。它创建有 data 数组并调用 quicksort() 函数。然后将结果作为一个字串输出：

**# do\_sort --- sort the data according to ‘compare’**

**# and return it as a string**

**function do\_sort(first， last， compare， data， i， retval)**

**{**

**delete data**

**for (i = 1; first <= last; first++) {**

**data[i] = $first**

**i++**

**}**

**quicksort(data， 1， i-1， compare)**

**retval = data[1]**

**for (i = 2; i in data; i++)**

**retval = retval "" data[i]**

**return retval**

**}**

最后，两个排序函数调用 do\_sort()，并将两个比较函数名传递进去：

**# sort --- sort the data in ascending order and return it as a string**

**function sort(first， last)**

**{**

**return do\_sort(first， last， "num\_lt")**

**}**

**# rsort --- sort the data in descending order and return it as a string**

**function rsort(first， last)**

**{**

**return do\_sort(first， last， "num\_ge")**

**}**

这是数据文件的扩展版本：

**Biology\_101 sum average sort rsort data: 87.0 92.4 78.5 94.9**

**Chemistry\_305 sum average sort rsort data: 75.2 98.3 94.7 88.2**

**English\_401 sum average sort rsort data: 100.0 95.6 87.1 93.4**

最后，这是增强过的程序的运行结果：

**$ gawk -f quicksort.awk -f indirectcall.awk class\_data2**

**-| Biology 101:**

**-| sum: <352.8>**

**-| average: <88.2>**

**-| sort: <78.5 87.0 92.4 94.9>**

**-| rsort: <94.9 92.4 87.0 78.5>**

**-|**

**-| Chemistry 305:**

**-| sum: <356.4>**

**-| average: <89.1>**

**-| sort: <75.2 88.2 94.7 98.3>**

**-| rsort: <98.3 94.7 88.2 75.2>**

**-|**

**-| English 401:**

**-| sum: <376.1>**

**-| average: <94.025>**

**-| sort: <87.1 93.4 95.6 100.0>**

**-| rsort: <100.0 95.6 93.4 87.1>**

另一个使用间接函数非常有用的地方可以在处理数组的地方找到。这在 ， ***在第 249 页***，中有描述。

要记住，在调用间接函数之前，必须在其前面放置一个 ‘@’ 符号。

从 4.1.2 版本开始，gawk 间接调用也可以用于内置函数以及扩展函数（查看 ***，在第 221 页***）。他们在使用间接调用内置函数时有一些限制，如下：

* 通过间接调用，你不可以传递正则表达式常量到一个内置函数中。[[68]](#footnote-69)这同时也适用于 sub()， gsub()， gensub()， match()， split() 与 patsplit() 函数。
* 如果调用 sub() 或者 gsub()，你只可以传递两个参数，因为这些函数通常会更改它它们的第三个参数。这表示 $0 会被改变。

gawk 会尽力地使间接函数调用非常高效。如，在下面的例子中：

**for (i = 1; i <= n; i++)**

**@the\_*func()***

gawk 实际上只查找一次要调用的函数。

## 9.4 总结

* awk 提供了内置函数，也可以让你自己定义函数。
* POSIX awk 提供了三种内置函数：数值，字串与 I/O。gawk 提供了对数组排序的函数，处理时间的函数，执行位操作的函数，确定变量类型的函数（数组还是标量），与国际化与本地化的函数。gawk 也提供了标准函数的扩展，典型的是针对额外参数的形式。
* 函数接收 0 个或者更多的参数，并返回一个值。提供参数值的表达式在函数调用前会首先全部求值。求值的顺序是未定义的。返回值可以被忽略。
* sub() 与 gsub() 函数对于反斜杠的处理有点复杂。在 gawk 的 gensub() 函数中则更直接一点，但是使用这个函数也需要注意。
* 用户定义的函数提供了很重要的能力，但是其语法确并不优雅。在函数调用时，在函数名与参数列表的开括号之间不可以有空格。也没有提供本地变量，所以习惯是添加额外的参数，并用空格可视化地将它们与参数列表分开。
* 用户自定义函数可以调用其他的用户自定义函数（以及内置函数），也可以递归地调用它们自己。函数参数会“隐藏”任何的同名的全局参数。你不可以使用保留的变量（如 ARGC）来作为自定义函数的参数名字。
* 标量值是通过传值方式传递到用户自定义函数的。数组参数则是通过传引用，函数对于数组的任何改变都会在函数返回之后可见。
* 使用 return 语言来从用户自定义函数中返回。可选的表达式成为函数的返回值。函数只可以返回标量值。
* 如果传递到用户自定义函数中的变量从来没有被使用过，那么函数如何来看待这个变量：要么是标题要么是数组。
* gawk 使用特殊语法，提供了间接函数调用。通过将某个变量的值设置成函数名，你可以在运行时确定在想说点要调用的函数。这个与 C 跟 C++ 中的函数指针等同。

**Part II:**

用 ***awk*** 来解决问题

# 第十章 awk 函数库

在 ，***在第 204 页***，描述了如何来写你自己的 awk 函数。写函数是很重要的，因为它可以让你将算法与程序任务封闭在一个单独的地方。同时，它也简化编程，使得程序开发更有可管理性，更可读。

Brian Kernighan 与 P.J. Plauger 在他们 1976 年的书，《Software Tools》[[69]](#footnote-70)中写到：

好的编程不是从一般性来学习，而是通过运用常识与编程实践，看程序如何能够写得清晰，易读，易维护与修改，可被人工程化，高效与可靠。仔细学习与模拟好的编程可以写得更好。

实际上，他们觉得这些思想是如何重要，以致于它们将这个表述放在书的封面上。由于我们强烈地认为这个表述是无此正确的，所在在这章中与 ***， 在第 253 页***，提供了大小合适的代码来让你学习，我们希望你能从中学到什么。

这一章呈现了 awk 中有很有用的函数库。很多后面的例子代码都使用这些函数。这里呈现的函数从简单到复杂进行变化。

在 ，***在第 281 页***，中提交了一个程序，你可以用来从这些例子库函数中提取源代码，以及从本书的 Texinfo 源代码中提出程序。（这已经作为 gawk 发行版本的一部份完成了。）

如果你写了一些有用的，通用目的的 awk 程序，并且想将它们贡献给 awk 社区，查看 ， ***在第 16 页***，来获取更多的信息。

这一章以及 ***， 在第 253 页***，中的程序，可以自由地使用 gawk 特有的特性。为不同的 awk 实现重写这些程序也相当地直接：

* 区别错误信息并发送到 /dev/stderr。如果你的系统中没有 /dev/stderr，或者你无法使用 gawk，则使用 ‘| "cat 1>&2"’ 而不是 ‘>"/dev/stderr"’。
* 有很多的程序中使用 nextfile （查看 ）来路过任意剩下的输入文件的输入。
* 最后，某些程序选择忽略输入中的大小写。它们是通过将 IGNORECASE 赋值为 1 来实现的。你可以得到几乎相同的效果，[[70]](#footnote-71)只要将下面的规则加到程序的开始：

**# ignore case**

**{ $0 = tolower($0) }**

同时，确定所有的正则表达式以及用在比较中的字串常量全都是小写字母。

## 10.1 命名函数库的全局变量

由于 awk 语言的演化方案，变量要么是全局的（可以在整个程序中使用），要么是本地的（只在特定的函数中使用）。没有与 C 中的静态变量类似的中间状态。

库函数常常需要有全局变量来保存在调用函数之间的状态信息——比如，getopt() 的变量 \_opti (查看 ， ***在第 237 页***）。这样的变量称为私有变量，因为在库中使用它们的函数只有一个。

当写一个库函数的时候，你应该要尝试选择你自己的私有变量的名字，这样不会与其他的库函数或者用户主程序所使用的变量冲突。如像名字 i 或者 j 就不是好的选择，因此用户程序常常使用这些类似的名字来用任自己的目的。

在本章中呈现的例子代码中，所有私有变量的名字都以下划线开始（‘\_’）。用户一般来说不会使用前导的下划线来命名变量，所以这样的惯例立即降低了变量名字偶然与用户程序冲突的机率。

另外，有几个库函数还使用相应的前缀，用来指示什么样的函数或者函数集来使用变量——如，在用户数据库例程中的 \_pw\_byname() （查看 ， ***在第 242 页***）。这样的惯例我们是推荐的，因为这更进一步降低了不小心的变量名字冲突机率。要注意，这样的惯例也可用于变量名与私有函数名。[[71]](#footnote-72)

在函数命名上还有最后一个提示，如果函数使全局变量可被主程序使用，那么惯例上需要以一个大写字母来开始变量的名字——如 getopt() 的 Opterr 与 Optind 变量（查看 ， ***在第 237 页***）。前导的大写字母表示这是一个全局变量，而变量的名字不全是大写字母则是表示变量不是 awk 预定义的变量，比如 FS。

在库中的函数的所有变量实际上不用于存储状态的变量都声明为本地变量也非常重要。[[72]](#footnote-73)如果不是这么做的，变量有可能偶然在用户程序中使用，从而导致非常难以追踪的 Bug。

**function lib\_func(x， y， l1， l2)**

**{**

**...**

**# some\_var should be local but by oversight is not**

**use variable some\_var**

**...**

**}**

在 Tcl 社区中也有相同的问题，但是使用不同的惯例，即用单一的关联数组来存储库函数或者“包”所使用的值。这更进一步地降低了实际全局使用的变量的数目。例如，在 ，***在第 242 页***，中描述的函数，已经使用数组元素 PW\_data["inited"]，PW\_data["total"]，PW\_data["count"]， 与 PW\_data["awklib"]，而不是使用 \_pw\_inited， \_pw\_awklib， \_pw\_total， 与 \_pw\_count。

这章中所呈现的惯例就是：约定俗成。但你不需要按这种方式来写你的程序——我们只是建议你这么做。

## 10.2 通用编程

这一章中呈现了在通用编程中使用到的几个函数。

### 10.2.1 将数值转换成字串

***strtonum()*** 函数（查看 ， ***在第 184 页***）是一个 gawk 扩展。下面的函数提供了 awk 中另一个版本的实现：

**# mystrtonum --- convert string to number**

**function mystrtonum(str， ret， n， i， k， c)**

**{**

**if (str ~ /^0[0-7]\*$/) {**

**# octal**

**n = length(str)**

**ret = 0**

**for (i = 1; i <= n; i++) {**

**c = substr(str， i， 1)**

**# index() returns 0 if c not in string，**

**# includes c == "0"**

**k = index("1234567"， c)**

**ret = ret \* 8 + k**

**}**

**} else if (str ~ /^0[xX][[:xdigit:]]+$/) {**

**# hexadecimal**

**str = substr(str， 3) # lop off leading 0x**

**n = length(str)**

**ret = 0**

**for (i = 1; i <= n; i++) {**

**c = substr(str， i， 1)**

**c = tolower(c)**

**# index() returns 0 if c not in string，**

**# includes c == "0"**

**k = index("123456789abcdef"， c)**

**ret = ret \* 16 + k**

**}**

**} else if (str ~ \**

**/^[-+]?([0-9]+([.][0-9]\*([Ee][0-9]+)?)?|([.][0-9]+([Ee][-+]?[0-9]+)?))$/) {**

**# decimal number， possibly floating point**

**ret = str + 0**

**} else**

**ret = "NOT-A-NUMBER"**

**return ret**

**}**

**# BEGIN { # gawk test harness**

**# a[1] = "25"**

**# a[2] = ".31"**

**# a[3] = "0123"**

**# a[4] = "0xdeadBEEF"**

**# a[5] = "123.45"**

**# a[6] = "1.e3"**

**# a[7] = "1.32"**

**# a[8] = "1.32E2"**

**#**

**# for (i = 1; i in a; i++)**

**# print a[i]， strtonum(a[i])， mystrtonum(a[i])**

**# }**

函数首先找 C 风格的八进制数（以8为底）。如果输出串匹配一个描述八进制的正则表达式，则 mystrtonum() 迭代在串中的每一个字符。它将 k 设置成当前八进制数在 “1234567” 中的索引。返回值要么是与数字相同的值，或者在不存在这个字符时返回 0。那就是真正的 ‘0’。这是安全的，因为在 if 中的正则表达式保证了只有八进制才会进行转换。

相同的逻辑也可以应用到转换十六进制的代码中，这些值以 ‘0x’ 或者 ‘0X’ 开头。使用 tolower() 可以简化查找每个十六进制位数字的计算。

最后，如果字串匹配（相当复杂的）常规十进制或者浮点数的正则表达式，则 ‘ret = str + 0’ 的计算可以让 awk 将这个值转换成一个数。

被注释掉的是测试程序，所以函数可以用 gawk 来测试，并且结果可以用来与内置的 strtonum() 函数结果来比较。

### 10.2.2 断言

当写一个很大的程序的时候，常常了解一些条件或者某集合的条件为 true 是很有用的。在执行特定计算之前，你要写一条语句你确信条件成立。这样的语句被称为断言。C 语言提供了 <assert.h> 头文件以及相应的 assert() 宏，这样程序可以用它来作断言。如果断言失败，assert() 宏会打印一些诊断信息以描述条件应该为 true，但实际上不是，然后杀死程序。在 C 语言中，像这样来使用 assert()：

**#include <assert.h>**

**int myfunc(int a， double b)**

**{**

**assert(a <= 5 && b >= 17.1);**

**...**

**}**

如果断言失败，程序打印像下面这样的信息：

**prog.c:5: assertion failed: a <= 5 && b >= 17.1**

C 语言可以使条件转换为字串，并在打印诊断信息里使用。但在 awk 中却不可能，所以 assert() 函数也需要一个测试条件的字串版本。下面则是字串函数：

**# assert --- assert that a condition is true. Otherwise， exit.**

**function assert(condition， string)**

**{**

**if (! condition) {**

**printf("%s:%d: assertion failed: %s\n"，**

**FILENAME， FNR， string) >"/dev/stderr"**

**\_assert\_exit = 1**

**exit 1**

**}**

**}**

**END {**

**if (\_assert\_exit)**

**exit 1**

**}**

assert () 函数测试条件参数。如果失败，则打印一个信息到标准输出，并使用 string 参数来描述失败的条件。然后它将变量 \_assert\_exit 设置成 1，并执行 exit 语句。exit 语句会跳到 END 规则中来执行。如果 END 规则发现 \_assert\_exit 为 true，就会立即出出。

在 END 规则中测试的目的是要保证任意的其他 END 规则不被执行。当断言失败，程序应该立即退出。如果断言通过，则 \_assert\_exit 则依然是 false，当 END 规则执行时，程序的其他的 END 规则则也会被执行。为了使这一切都正确执行，assert.awk 必须成为 awk 读取的第一个源代码文件。函数可以通过下面的方式被使用：

**function myfunc(a， b)**

**{**

**assert(a <= 5 && b >= 17.1， "a <= 5 && b >= 17.1")**

**...**

**}**

如果断言失败，你可以看到如下的信息：

**mydata:1357: assertion failed: a <= 5 && b >= 17.1**

这个版本的 assert() 有一个小问题。一个 END 规则会自动加到调用 assert() 函数的程序中。正常情况下，如果程序由 BEGIN 规则组成，输入文件与/或者标准输入不会被读取。但是，现在程序有一个 END 规则，awk 会尝试读取输入文件的数据或者标准输入的数据（查看 ， ***在第 141 页***），很有可能导致程序挂起，因引它可能会一直等待输入。

有一个很简单的解决方法：确保这这样的 BEGIN 规则总是用一个 exit 语句来结束。

### 10.2.3 数值舍入舍出

printf 与 sprintf 的工作方式（查看 ）执行的舍入舍出常常依赖于系统的 C 函数 sprintf 例程。在很多的机器上， sprintf 舍入舍出是没有偏向的，即是说，它并不总是把 0.5 向上舍入，这与很自然的预期相反。在没有偏向的舍入舍出中。.5 会偶数舍入舍出，而并不总是舍入，所以，1.5 舍入为 2，而 4.5 则舍出为 4。这就是说，如果你使用一个不舍入舍出的格式（比如 "%.0f"），你应该检查你的系统所执行的结果。下面的函数总是执行传输的舍入舍出，如果你的 awk 的 printf 执行的是没有偏向的舍入舍出，则可能很有用。

**# round.awk --- do normal rounding**

**function round(x， ival， aval， fraction)**

**{**

**ival = int(x) # integer part， int() truncates**

**# see if fractional part**

**if (ival == x) # no fraction**

**return ival # ensure no decimals**

**if (x < 0) {**

**aval = -x # absolute value**

**ival = int(aval)**

**fraction = aval - ival**

**if (fraction >= .5)**

**return int(x) - 1 # -2.5 --> -3**

**else**

**return int(x) # -2.3 --> -2**

**} else {**

**fraction = x - ival**

**if (fraction >= .5)**

**return ival + 1**

**else**

**return ival**

**}**

**}**

**# test harness**

**# { print $0， round($0) }**

### 10.2.4 Cliff 随机数生成器

Cliff 随机数产生器是一个非常简单的随机为数生成器，即“通过无结构显示来为随机性传递噪声球测试”。他很容易编程，不到 10 行的 awk 代码就可以完成：

**# cliff\_rand.awk --- generate Cliff random numbers**

**BEGIN { \_cliff\_seed = 0.1 }**

**function cliff\_rand()**

**{**

**\_cliff\_seed = (100 \* log(\_cliff\_seed)) % 1**

**if (\_cliff\_seed < 0)**

**\_cliff\_seed = - \_cliff\_seed**

**return \_cliff\_seed**

**}**

算法需要一个初始的种子值，为 0.1。每个新值都使用当前的种子的作为输出来进行计算。如果内置 rand() 函数（查看 ， ***在第 183 页***） 不够随机，你可以使用这个函数来代替。

### 10.2.5 在字符与数值之间变换

一个商业版本的 awk 实现提供了内置的函数 ord()，输入一个字符，然后返回那个字符在机器的字符集中的数值值。如果传递到 ord() 函数为字串，则会有多个字符，但是只有第一个字符被转换。这个函数相反的函数则为 chr()（来自由 Pascal 同名函数），它输入一个数值，然后返回相应的字符。这两个函数在 awk 中都可以很容易实现，因此没有真实的理由要将他们构建进行 awk 解析器：

**# ord.awk --- do ord and chr**

**# Global identifiers:**

**# \_ord\_: numerical values indexed by characters**

**# \_ord\_init: function to initialize \_ord\_**

**BEGIN { \_ord\_init() }**

**function \_ord\_init( low， high， i， t)**

**{**

**low = sprintf("%c"， 7) # BEL is ascii 7**

**if (low == "\a") { # regular ascii**

**low = 0**

**high = 127**

**} else if (sprintf("%c"， 128 + 7) == "\a") {**

**# ascii， mark parity**

**low = 128**

**high = 255**

**} else { # ebcdic(!)**

**low = 0**

**high = 255**

**}**

**for (i = low; i <= high; i++) {**

**t = sprintf("%c"， i)**

**\_ord\_[t] = i**

**}**

**}**

被 \_ord\_init() 所使用的数值需要一些解释。今天所使用的最重要的字符集就是 ASCII。[[73]](#footnote-74)尽管 8 位的字节可以表示 256 个不同的值（从 0 到 255），ASCII 只定义了从 0 到 127 之间的字符。[[74]](#footnote-75)在很久的过去，至少有一个迷你计算机厂商使用 ASCII，但是带一个校验位，即字节的最高位总是 1。这表示，在这样的系统中，字符的值是从 128 到 255。最后，很多的大型机使用 EBCDIC 字符集。这个集中使用所有的 256 个字符。在一些旧系统中也使用其他的字符集，但是没有必须为此操心了。

**function ord(str， c)**

**{**

**# only first character is of interest**

**c = substr(str， 1， 1)**

**return \_ord\_[c]**

**}**

**function chr(c)**

**{**

**# force c to be numeric by adding 0**

**return sprintf("%c"， c + 0)**

**}**

**#### test code ####**

**# BEGIN {**

**# for (;;) {**

**# printf("enter a character: ")**

**# if (getline var <= 0)**

**# break**

**# printf("ord(%s) = %d\n"， var， ord(var))**

**# }**

**# }**

这个函数的一个明显改进是将 \_ord\_init() 函数的代码移到了 BEGIN 规则中。这么做的最初目的是为了开发的方便。

### 10.2.6 将数组合并成字串

当处理字串时，可以将数组的所有字串连接为一个长的字串常常很有用。下面的 join() 函数来做这样的工作。它在后面的几个应用程序中被使用到（查看 ***， 在第 253 页***）。

好的函数设计是很重要的，这个函数应该是通用的，但是它也应该有一个合理的默认行为。这个函数合用一个数组以及要合并的开始与结束索引，作为参数。它假设数组的索引是数值的——这是合理的假充，因此数组有可能使用 split() 函数创建的（查看 ， ***在第 184 页***）：

**# join.awk --- join an array into a string**

**function join(array， start， end， sep， result， i)**

**{**

**if (sep == "")**

**sep = ""**

**else if (sep == SUBSEP) # magic value**

**sep = ""**

**result = array[start]**

**for (i = start + 1; i <= end; i++)**

**result = result sep array[i]**

**return result**

**}**

一个可选的参数是用来连接字串的分隔符。如果调用者提供了非空的值，join() 则使用它，如果没有提供，其值则为空值。在这种情况下，join() 使用一个空格来作为默认的字串分隔符。如果值与 SUBSEP 的值相同，则 join() 会在连接的字串中不使用分隔符。SUBSEP 作为一个“魔法”值来指示不要在字串组件之间使用分隔符。[[75]](#footnote-76)

### 10.2.7 管理时间

在 ，***在第 197 页***，描述的 systime() 与 strftime() 函数，提供了最小的功能需求来处理人可读形式的时间处理。尽管 strftime() 是扩展的，格式控制不容易记住，或者在读程序时意思不这么明显。

下面的函数 getlocaltime() 使用用户提供的预定义时间格式信息数组。它返回一个与 date 工具产生一样的的时间格式的字串：

**# getlocaltime.awk --- get the time of day in a usable format**

**# Returns a string in the format of output of date(1)**

**# Populates the array argument time with individual values:**

**# time["second"] -- seconds (0 - 59)**

**# time["minute"] -- minutes (0 - 59)**

**# time["hour"] -- hours (0 - 23)**

**# time["althour"] -- hours (0 - 12)**

**# time["monthday"] -- day of month (1 - 31)**

**# time["month"] -- month of year (1 - 12)**

**# time["monthname"] -- name of the month**

**# time["shortmonth"] -- short name of the month**

**# time["year"] -- year modulo 100 (0 - 99)**

**# time["fullyear"] -- full year**

**# time["weekday"] -- day of week (Sunday = 0)**

**# time["altweekday"] -- day of week (Monday = 0)**

**# time["dayname"] -- name of weekday**

**# time["shortdayname"] -- short name of weekday**

**# time["yearday"] -- day of year (0 - 365)**

**# time["timezone"] -- abbreviation of timezone name**

**# time["ampm"] -- AM or PM designation**

**# time["weeknum"] -- week number， Sunday first day**

**# time["altweeknum"] -- week number， Monday first day**

**function getlocaltime(time， ret， now， i)**

**{**

**# get time once， avoids unnecessary system calls**

**now = systime()**

**# return date(1)-style output**

**ret = strftime("%a %b %e %H:%M:%S %Z %Y"， now)**

**# clear out target array**

**delete time**

**# fill in values， force numeric values to be**

**# numeric by adding 0**

**time["second"] = strftime("%S"， now) + 0**

**time["minute"] = strftime("%M"， now) + 0**

**time["hour"] = strftime("%H"， now) + 0**

**time["althour"] = strftime("%I"， now) + 0**

**time["monthday"] = strftime("%d"， now) + 0**

**time["month"] = strftime("%m"， now) + 0**

**time["monthname"] = strftime("%B"， now)**

**time["shortmonth"] = strftime("%b"， now)**

**time["year"] = strftime("%y"， now) + 0**

**time["fullyear"] = strftime("%Y"， now) + 0**

**time["weekday"] = strftime("%w"， now) + 0**

**time["altweekday"] = strftime("%u"， now) + 0**

**time["dayname"] = strftime("%A"， now)**

**time["shortdayname"] = strftime("%a"， now)**

**time["yearday"] = strftime("%j"， now) + 0**

**time["timezone"] = strftime("%Z"， now)**

**time["ampm"] = strftime("%p"， now)**

**time["weeknum"] = strftime("%U"， now) + 0**

**time["altweeknum"] = strftime("%W"， now) + 0**

**return ret**

**}**

这些字串下标很容易使用并比 strftime() 所使用的一系列的格式要容易。在 ，***在第 273 页***，呈现的 alarm 程序，使用了这个函数。getlocaltime() 函数更通用的设计可以允许用户提供一个可选的时间戳值，而不是仅使用当前时间。

### 10.2.8 一次读取整个文件

如果将整个文件的内容都放在内存中作为一个字串经常很有用。一个直接的，但是比较幼稚的方法可能如下：

**function readfile(file， tmp， contents)**

**{**

**if ((getline tmp < file) < 0)**

**return**

**contents = tmp**

**while (getline tmp < file) > 0)**

**contents = contents RT tmp**

**close(file)**

**return contents**

**}**

这个函数一次读取一个记录，然后用文件的内容来构建本地变量 contents。这有用，但是不够高效。

下面的函数，使用由 Denis Shirokov 提的建议，来一次读取指定名字的文件内容：

**# readfile.awk --- read an entire file at once**

**function readfile(file， tmp， save\_rs)**

**{**

**save\_rs = RS**

**RS = "^$"**

**getline tmp < file**

**close(file)**

**RS = save\_rs**

**return tmp**

**}**

通过将 RS 设置成 ‘^$’，即一个一定不会匹配文件内容的正则表达式。gawk 从文件中读取内容到 tmp 中，并尝试匹配 RS。每次读后的匹配都失败，但很匹配失败很快，这样 gawk 将 tmp 的内容用文件的内容来进行填充。（查看 ，***在第 63 页***，来获取 RT 与 RS 的信息）

如果文件为空，则返回值为空串。所以，调用代码可以使用这样的代码：

**contents = readfile("/some/path")**

**if (length(contents) == 0)**

**# file was empty ...**

这里测试结果看其是否为空还是不为空，一个等同的测试为 ‘contents== ""’。

查看 ，***在第 399 页***，来查看扩展的函数，它也将整个文件的内容读取到内存中。

### 10.2.9 将字串用引号引起并传递给 Shell

Michael Brennan 提供了下面的编程模式，他经常是这么使用：

**#! /bin/sh**

**awkp= ’**

**...**

**’**

**input\_program | awk "$awkp" | /bin/sh**

比如，他的一个程序名为 flac-edit 就使用这种形式：

**$ flac-edit -song="Whoope! That’s Great" file.flac**

它产生下面的输出，这些输出通过管道输出给 Shell（/bin/sh）：

**chmod +w file.flac**

**metaflac --remove-tag=TITLE file.flac**

**LANG=en\_US.88591 metaflac --set-tag=TITLE=’Whoope! That’"’"’s Great’ file.flac**

**chmod -w file.flac**

注意Shell 需要引号。函数 shell\_quote() 就是来做这个事情的。SINGLE 是一个一字符的串 “'”，而 QSINGLE 是一个三字符的串“\"'\"”：

**# shell\_quote --- quote an argument for passing to the shell**

**function shell\_quote(s， # parameter**

**SINGLE， QSINGLE， i， X， n， ret) # locals**

**{**

**if (s == "")**

**return "\"\""**

**SINGLE = "\x27" # single quote**

**QSINGLE = "\"\x27\""**

**n = split(s， X， SINGLE)**

**ret = SINGLE X[1] SINGLE**

**for (i = 2; i <= n; i++)**

**ret = ret QSINGLE SINGLE X[i] SINGLE**

**return ret**

**}**

## 10.3 数据文件管理

本节呈现的函数用于管理命令行数据文件非常有用。

### 10.3.1 注意数据文件的边界

BEGIN 与 END 规则相应地在你的 awk 程序的开始与结束执行一次（查看 ， ***在第 141 页***）。我们（gawk 的作者们）遇到过一个用户，错误地认为 BEGIN 规则在每个文件的开始都执行，而 END 规则在每个数据文件的结束时执行。

当被通知说事情不是这样的，用户请求我们加一个新的特定模式到 gawk 中，称为 BEGIN\_FILE 与 END\_FILE，这些都则具有所期望的行为。他甚至为我们提供这么做的代码。

给 gawk 加这个特殊模式并不是必需的，awk 自己都就可以很清楚地完成这个事情，如下面的函数程序所解释的一样。它会调用两个用户自己提供的函数 ，beginfile() 与 endfile() 分别在每个数据文件的开始与结束。不但很好地用9行代码解决上面的问题，而且还可移植称，这在任何 awk 实现版本都可工作。

**# transfile.awk**

**#**

**# Give the user a hook for filename transitions**

**#**

**# The user must supply functions beginfile() and endfile()**

**# that each take the name of the file being started or**

**# finished， respectively.**

**FILENAME != \_oldfilename {**

**if (\_oldfilename != "")**

**endfile(\_oldfilename)**

**\_oldfilename = FILENAME**

**beginfile(FILENAME)**

**}**

**END { endfile(FILENAME) }**

这个文件必须在用户的主程序之前装入，因此这里所提供的规则会首先执行。

这个规则依赖于 awk 的 FILENAME 变量，这个变量在处理每个新的数据文件时自动变化。当前的文件名存储在一个私有变量， \_oldfilename 中。如果 FILENAME 与 \_oldfilename 不相同，则一个新的数据文件正在被处理，则有必须为旧文件调用 endfile() 。因为 endfile() 仅在一个文件被处理完后会调用，所以，程序首先要检查确定 \_oldfilename 不是一个空串。程序然后将当前文件名赋值给 \_oldfilename 然后为当前文件调用 beginfile() 。像 awk 的变量一样，因为 \_oldfilename 被初始化为空串，这个规则甚至在读取第一个数据文件时也是正确的。

程序也提供一个 END 规则来处理最后一个文件。因为这个 END 规则在任何其他的由主程序提供 END 规则之前，因此 endfile() 会首先被调用。再一次地， 多个 BEGIN 与 END 规则的值应该被清除。

如果同样的数据文件在一个命令中出现了两次，则 endfile() 与 beginfile() 在第一轮结束的时候不会被执行，而在第二轮开始的时候执行。下面的版本解决了这个问题：

**# ftrans.*awk* --- handle datafile transitions**

**#**

**# user supplies *beginfile()* and *endfile()* functions**

**FNR == 1 {**

**if (\_filename\_ != "")**

**endfile(\_filename\_)**

**\_filename\_ = FILENAME**

**beginfile(FILENAME)**

**}**

**END { endfile(\_filename\_) }**

在 ，***在第 270 页***，中展示了这个库函数可以用于以及如何来简化主程序的编写。

|  |
| --- |
| 那么为什么 gawk 需要 BEGINFILE 与 ENDFILE?  你可以很奇怪，如果 beginfile() 与 endfile() 函数可以做相同的事情，为什么 gawk 还要有 BEGINFILE 与 ENDFILE 模式呢？  问题很好。正常情况下，如果 awk 没法打开一个文件，这会立即导致一个致使错误。在这种情况下，对于用户自己定义的函数来讲，就没有办法来处理这个问题。因为调用函数的机制依赖于正在被打开的文件及其第一条记录。所以，BEGINFILE 的主要原因是给你一个“勾子”来捕获不能被处理的文件。要取得更多的信息，请查看 ， ***在第 143 页***。 |

### 10.3.2 重新读取当前文件

对于内置函数的另一个请求则是加入这样的一个函数使其可以重新读取当前文件。请求的用户不想在循环中使用 getline（查看 ， ***在第 83 页***）。

但是，只要你还不在 END 规则中，就非常容易安排立即关闭当前文件，然新头开始处理这个文件。由于缺少好名字，我们把它取名为 rewind()：

**# rewind.*awk* --- rewind the current file and start over**

**function rewind( i)**

**{**

**# shift remaining arguments up**

**for (i = ARGC; i > ARGIND; i--)**

**ARGV[i] = ARGV[i-1]**

**# make sure gawk knows to keep going**

**ARGC++**

**# make current file next to get done**

**ARGV[ARGIND+1] = FILENAME**

**# do it**

**nextfile**

**}**

rewind() 函数依赖于 ARGIND 变量（查看 ， ***在第 158 页***），这个变量是 gawk 特有的。同时也依赖于 nextfile 关键词（查看 ， ***在第 154 页***）。由于这个原因，你不可以在 ENDFILE 规则中调用它。（也没有必要，因此 gawk 只要 ENDFILE 一结束就开始了下一个文件！）

### 10.3.3 检查可读的数据文件

一般情况下，如果你提供 awk 一个数据文件是不可读的，它会以致使错误而结束。有一些时候，你可能想忽略这样的问题并往下走。[[76]](#footnote-77)你可以将下面的程序插入到你的 awk 程序中：

**# readable.*awk* --- library file to skip over unreadable files**

**BEGIN {**

**for (i = 1; i < ARGC; i++) {**

**if (ARGV[i] ~ /^[a-zA-Z\_][a-zA-Z0-9\_]\*=.\*/ \**

**|| ARGV[i] == "-" || ARGV[i] == "/dev/stdin")**

**continue # assignment or standard input**

**else if ((*getline* junk < ARGV[i]) < 0) # unreadable**

**delete ARGV[i]**

**else**

**close(ARGV[i])**

**}**

**}**

这个可以工作，因为 getline 不会导致致使错误。将 delete 从 ARGV 数组中移除元素以跳过这些文件（因此它不再在列表中）。查看 ， ***在第 163 页***。

由于 awk 变量名字只允许是英文字母，所以正则表达式检查有意不使用字符类，如 ‘[:alpha:]’ 与 ‘[:alnum:]’（查看 ， ***在第 56 页***）。

### 10.3.4 检查长度为零的文件

所有已知的 awk 实现都默默地跳过长度为零的文件。这是由 awk 隐式规则“读取记录并匹配规则”循环产生的结果。当 awk 尝试读取空文件记录时，它立即会接收到文件结束的指示，然后会关闭文件，并处理命令行中的下一个文件，而不会执行任务用户层的 awk 程序代码。

使用 gawk 的 ARGIND 变量（查看 ， ***在第 155 页***），有可能来检测当遇到一个空文件时则跳过。与在 ，***在第 233 页***，呈现的库文件相似，下面的库文件会调用用户必须提供的函数，名为 zerofile() 。传递的参数为文件名，以及其在 ARGV 中的位置：

**# zerofile.*awk* --- library file to process empty input files**

**BEGIN { Argind = 0 }**

**ARGIND > Argind + 1 {**

**for (Argind++; Argind < ARGIND; Argind++)**

**zerofile(ARGV[Argind]， Argind)**

**}**

**ARGIND != Argind { Argind = ARGIND }**

**END {**

**if (ARGIND > Argind)**

**for (Argind++; Argind <= ARGIND; Argind++)**

**zerofile(ARGV[Argind]， Argind)**

**}**

用户层的变量 Argind 允许 awk 程序跟踪它的 ARGV 进度。无论什么时候程序检测到了 ARGIND 大于 ‘Argind+1’，这就表示一到多个的文件名已经被跳过了。然后调用 zerofile() 的动作来处理每一个文件，并递增 Argind 变量。

‘Argind != ARGIND’ 中是简单地来使 Argind 在正常情况下更新为最新状态。最后 END 规则会捕获在命令行最后的参数中的空文件。注意在 for 循环中使用的测试条件，是 使用 ‘<=’操作符而不是 ‘<’。

### 10.3.5 将赋值当成是文件名

有时候，你不想 awk 来处理命令行的变量赋值（查看 ， ***在第 116 页***）。尤其是你有一个文件中包含 ‘=’ 符号，awk 会将这个文件名当成一个赋值，而不会处理它。

一些用户曾建议了一个额外的命令行选项来给 gawk 去禁用命令行的赋值。但是，一些库文件中的简单的程序就可以解决：

**# noassign.*awk* --- library file to avoid the need for a**

**# special option that disables command-line assignments**

**function disable\_assigns(argc， argv， i)**

**{**

**for (i = 1; i < argc; i++)**

**if (argv[i] ~ /^[a-zA-Z\_][a-zA-Z0-9\_]\*=.\*/)**

**argv[i] = ("./" argv[i])**

**}**

**BEGIN {**

**if (No\_command\_assign)**

**disable\_assigns(ARGC， ARGV)**

**}**

然后你这么来运行程序：

***awk* -v No\_command\_assign=1 -f noassign.*awk* -f yourprog.*awk* \***

函数会迭代所有的参数。它会在匹配变量赋值形式的参数前插入 ‘./’，来将参数转换为文件名。

通过给 No\_command\_assign 在调用时赋值为 true，可以让你调用时禁用命令行中的赋值。当没有设置时，它的初始值为 0（即 false），所以命令行参数就不会变化。

## 10.4 处理命令行选项

在 POSIX 兼容的系统上的大多数工具都从命令行中取得选项，用此来改变程序的行为方式。awk 是一个这样的例子（查看 ， ***在第 35 页***）。经常地，选择都会有参数（即程序命令行要正确遵守选项的所需要的数据）。例如 awk 的 -F 选项，需要一个串来作为域的分隔符。第一个出现在命令行中的 -- 或者不能 ‘-’ 开头的字串结束选项。

现在的 Unix 系统提供了 C 函数，名字为 getopt() 来处理命令行参数。程序员提供一个字串用来描述单字母的选项。如果选项需要一个参数，则它在字串后后面会跟一个冒号。getopt() 同时也传递命令行参数的数量与值，并在循环中进行调用。getopt() 为选项字母处理命令行参数。每次迭代循环，它就返回一个代表下一个它所发现的选项字母，或者为无效的选项的时候，返回 ‘?’。当返回 -1 时，表示命令行中没有要处理的选项了。

当使用 getopt() 时，没有带参数的选项可以被合并在一起。另外，需要带参数选项需要参数与要提供。参数可以直接跟在选项字母后面，或者也可以是一个分离的命令行参数。

假设有一个假想的程序，接收三个命令行参数，-a，-b 与 -c，而 -b 需要一个参数，下面所有的该当都是有效的调用程序的方式：

**prog -a -b foo -c data1 data2 data3**

**prog -ac -bfoo -- data1 data2 data3**

**prog -acbfoo data1 data2 data3**

注意到，当参数与它的选项进行组合时，参数后面的部分都被当成是选项的参数。在这个例子中， -abcfoo 表示所有的 -a，-b 与 -c 选项，并且 foo 是 -b 选项的参数。

getopt() 提供了4 个外部的变量，程序员使用命用他们：

optind 在参数值数组（argv） 中的索引，在这个数组中，可以找到第一个不是选项的命令参数。

optarg 某此选项的参数的字串值。

opterr 通常情况下， getopt() 在找到一个无效的选项的时候会打印一个错误消息。将 opterr 设置为 0 可以禁用这个特性。（应用程序可能想打印自己的错误消息。）

optopt 代表命令行选项的字母。

下面的 C 代码版本展示了 getopt() 如何来处理 awk 命令行参数的可能方式：

**int**

**main(int argc， char \*argv[])**

**{**

**...**

**/\* print our own message \*/**

**opterr = 0;**

**while ((c = getopt(argc， argv， "v:f:F:W:")) != -1) {**

**switch (c) {**

**case 'f': /\* file \*/**

**...**

**break;**

**case 'F': /\* field separator \*/**

**...**

**break;**

**case 'v': /\* variable assignment \*/**

**...**

**break;**

**case 'W': /\* extension \*/**

**...**

**break;**

**case ’?’:**

**default:**

***usage()*;**

**break;**

**}**

**}**

**...**

**}**

插一句，gawk 实际上是使用 GNU 的 getopt\_long() 函数来处理常规的选项以及 GNU 风格的长选项（查看 ， ***在第 35 页***）。

由 getopt() 提供的抽象非常有用，并且在 awk 程序中也非常方便。下面是一个 awk 版本的 getopt()。这个函数突出了 awk 中最弱的一个方面，即它在处理单个字符的时候非常差。重复地调用 substr() 在读取单个字符时是必须的（查看 ， ***在第 184 页***）。[[77]](#footnote-78)

下面一点点地来解释代码：

**# getopt.*awk* --- Do C library getopt(3) function in *awk***

**# External variables:**

**# Optind -- index in ARGV of first nonoption argument**

**# Optarg -- string value of argument to current option**

**# Opterr -- if nonzero， print our own diagnostic**

**# Optopt -- current option letter**

**# Returns:**

**# -1 at end of options**

**# "?" for unrecognized option**

**# <c> a character representing the current option**

**# Private Data:**

**# \_opti -- index in multiflag option， e.g.， -abc**

函数以注释开始，说明了它所使用的全局变量的一个列表，返回的值是什么，他们又表示什么，以及这个库函数所私有的全局变量。这样的文档对于任何程序来讲都是很重要的，尤其是对于库函数。

getopt() 函数首先检查确实是以选项的串来进行调用的（选项参数）。如果选项的长度为 0，则 getopt() 会立即返回 -1：

**function getopt(argc， argv， options， thisopt， i)**

**{**

**if (length(options) == 0) # no options given**

**return -1**

**if (argv[Optind] == "--") { # all done**

**Optind++**

**\_opti = 0**

**return -1**

**} else if (argv[Optind] !~ /^-[^:[:space:]]/) {**

**\_opti = 0**

**return -1**

**}**

下个要做的事情是要检查选项的结束。 -- 用于结束命令行选项，因为没有任何的命令行参数不是以 ‘-’ 开始。Optind 用来一个个地迭代命令行参数数组，它在调用 getopt() 的过程中一直保留它的值，因为它是一个全局变量。

在这里使用了正则表达式 /^-[^:[:space:]/，用于检查 ‘-’ 后面跟任意的不是空格与冒号的字符。如果当 命令行参数不匹配这个模式，则它就不是一个选项，然后结束处理。后续：

**if (\_opti == 0)**

**\_opti = 2**

**thisopt = substr(argv[Optind]， \_opti， 1)**

**Optopt = thisopt**

**i = index(options， thisopt)**

**if (i == 0) {**

**if (Opterr)**

***printf*("%c -- invalid option\n"， thisopt) >"/dev/stderr"**

**if (\_opti >= length(argv[Optind])) {**

**Optind++**

**\_opti = 0**

**} else**

**\_opti++**

**return "?"**

**}**

\_opti 变量用于跟踪当前命令行参数的位置（argv[Optind]）。如果有多个选项用 ‘-’ 合并在一起（如，-abx），则有必要一次返回一个。

如果 \_opti 的等于 0，则被设置为 2，这个值是字串中下一个要查找字符的位置（我们跳过 ‘-’，而它的位置是 1）。变量 thisopt 保存用 substr() 取得的字符。它被保存于 Optopt 以备主程序的使用。

如果 thisopt 不在选项串中，则它不是一个有效的选项。如果 Opterr 的值为非 0，getopt() 打印一个错误信息到标准错误输出上，这个与 C 版本的 getopt() 的信息类似。

由于选项是无效的，就有必须跳过它并处理下一个选项字符。如果 \_opti 大于或者等于当前命令行参数的个长度，则有必要移动到下一个参数。所以，Optind 被递增，\_opti 被设置为 0。否则的话，Optind 不会改变，而只递增 \_opti。

在任何情况下，由于选项是无效的，getopt() 都返回 “?”。主程序可以检查 Optopt 以确定无效的选项到底是什么。继续：

**if (substr(options， i + 1， 1) == ":") {**

**# get option argument**

**if (length(substr(argv[Optind]， \_opti + 1)) > 0)**

**Optarg = substr(argv[Optind]， \_opti + 1)**

**else**

**Optarg = argv[++Optind]**

**\_opti = 0**

**} else**

**Optarg = ""**

如果选项需要一个参数，在选项字串中的选项字母后面会跟着一个冒号。如果在当前的命令参数列表中（argv[Optind]）还有字符，则字串后面的内容会被赋值给 Optarg。否则就使用下一个命令参数（‘-xFOO' 对 ‘-x FOO’）。在任一情况下，\_opti 都被设置为 0，因为没有更多的字符剩下来检查当前的命令行参数。继续：

**if (\_opti == 0 || \_opti >= length(argv[Optind])) {**

**Optind++**

**\_opti = 0**

**} else**

**\_opti++**

**return thisopt**

**}**

最后，如果 \_opti 为 0 或者大于当前命令行参数的长度，则它表示元素在 argv 中已经处理了，所以，Optind 被递增以指向下一个 argv 中的元素。如果没有一个条件为 true，则只有 \_opti 被递增，所以下一个选项字母可以在下次调用 getopt() 时被处理。

BEGIN 规则中将 Opterr 与 Optind 都设置为 1。Opterr 被设置为 1 的原因是 getopt() 的默认行为就是在发现无效选项的时候打印诊断信息。Optind 设置为 1 是因为没有原因要去查看程序名，因为程序名存在 ARGV[0] 中：

**BEGIN {**

**Opterr = 1 # default is to diagnose**

**Optind = 1 # skip ARGV[0]**

**# test program**

**if (\_getopt\_test) {**

**while ((\_go\_c = getopt(ARGC， ARGV， "ab:cd")) != -1)**

***printf*("c = <%c>， Optarg = <%s>\n"，\_go\_c， Optarg)**

***printf*("non-option arguments:\n")**

**for (; Optind < ARGC; Optind++)**

***printf*("\tARGV[%d] = <%s>\n"，**

**Optind， ARGV[Optind])**

**}**

**}**

BEGIN 规则的剩下部分就是一个很简单的测试程序。这是两上例子，运行测试程序时的结果：

**$ *awk* -f getopt.*awk* -v \_getopt\_test=1 -- -a -cbARG bax -x**

**-| c = <a>， Optarg = <>**

**-| c = <c>， Optarg = <>**

**-| c = <b>， Optarg = <ARG>**

**-| non-option arguments:**

**-| ARGV[3] = <bax>**

**-| ARGV[4] = <-x>**

**$ *awk* -f getopt.*awk* -v \_getopt\_test=1 -- -a -x -- xyz abc**

**-| c = <a>， Optarg = <>**

**error x -- invalid option**

**-| c = <?>， Optarg = <>**

**-| non-option arguments:**

**-| ARGV[4] = <xyz>**

**-| ARGV[5] = <abc>**

在两次运行里，第一个 -- 会终结 awk 中的参数，所以它不会试图去将 -a 等等来当成自己的选项。

提示：在 getopt() 完成后，用户层代码必须清除所有的 ARGV 元素，从索引 1 到 Optind，所以，awk 不会试图去把命令行选项当成文件名来处理。

使用 ‘#!’ 并带上 -E 选项可以避免将你的程序的选项与 gawk 的选项的冲突，因为 -E 选项会导致 gawk 放弃处理后续的选项（查看 ，***在第 22 页***，以及 ， ***在第 35 页***）。

在 ***， 在第 253 页***，中的几个例子使用 getopt() 来处理它们的参数。

## 10.5 读取用户数据库

PROCINFO 数组（查看 ）提供了访问当前用户直实与有效的用户与组 ID数值的方式，并且如果可用，用户的补充的组集也可以访问。但是，因为这些都是些数值，它们没有提供非常有用的信息给到普通的用户。必须有什么样的方法来找到与用户与组 ID 数组相关的用户信息。这一节就是提供一组函数来从用户数据库中返回信息。查看 ，***在第 246 页***，可了解从组数据库中返回信息的一组函数。

POSIX 标准没有定义用户的信息存在哪。相反，它提供了 <pwd.h> 头文件以及几个 C 语言例程来获取用户的信息。主要的函数有 getpwent()，用来“get password entry”（获取密码项）。密码来自于最初的用户数据库文件，/etc/passwd，在里面存储了用户信息以及加密过的密码（与名字有相关）。

尽管 awk 程序可以简单直接地读取 /etc/passwd 文件，但是文件并没有包含系统中用户集的完整信息。[[78]](#footnote-79)为了确保你能够产生可读的以及完整的用户数据版本，有必要调用 getpwent() 来写一个小的 C 程序。getpwent() 定义为返回 struct passwd 的指针。每次调用，它都返回下一个项。当没有更多的项时，则返回 NULL，即空指针。当出现这个结果后，C 程序则要调用 endpwent() 来关闭数据库。下面是 pwcat，一个 C 程序用来 ‘cat’ 密码数据库：

**/\***

**\* pwcat.c**

**\***

**\* Generate a printable version of the password database.**

**\*/**

**#include <stdio.h>**

**#include <pwd.h>**

**int**

**main(int argc， char \*\*argv)**

**{**

**struct passwd \*p;**

**while ((p = *getpwent()*) != NULL)**

***printf*("%s:%s:%ld:%ld:%s:%s:%s\n"**

**， p->pw\_name， p->pw\_passwd**

**， (long) p->pw\_uid**

**，(long) p->pw\_gid， p->pw\_gecos， p->pw\_dir， p->pw\_shell);**

***endpwent()*;**

**return 0;**

**}**

如果你不理解 C 代码，也没有关系。pwcat 的输出就是以传统的 /etc/passwd 的冒号分隔域的格式输出用户数据库。这些域是：

Login name

用户的登录名。

Encrypted password

用户加密过的密码。这在某些系统上可能得不到。

User-ID 用户的数值 ID。（在一些系统上，这是一个 C 的 long 类型，而不是 int 型，所以，我们在所有的情况下都转换为 long。）

Group-ID 用户的组数值 ID。（内容与刚才所述的 long 与 int 类型的解释类似。）

Full name 用户的全名，以及与这个用户相关的其他信息。

Home directory

用户的登录目录（或者 “home”）（类似于 Shell 程序员中的 $HOME）。

Login shell

当用户登录时所要运行的程序。一般来说是一个 Shell 程序，如 Bash。

pwcat 的输出的几行内容例子如下：

**$ pwcat**

**-| root:x:0:1:Operator:/:/bin/sh**

**-| nobody:\*:65534:65534::/:**

**-| daemon:\*:1:1::/:**

**-| sys:\*:2:2::/:/bin/csh**

**-| bin:\*:3:3::/bin:**

**-| arnold:xyzzy:2076:10:Arnold Robbins:/home/arnold:/bin/sh**

**-| miriam:yxaay:112:10:Miriam Robbins:/home/miriam:/bin/sh**

**-| andy:abcca2:113:10:Andy Jacobs:/home/andy:/bin/sh**

**...**

根据之前的一些说明，下面是一组函数用来取得用户的信息的函数。这里有几个函数与 C 中同名的函数对应：

**# passwd.*awk* --- access password file information**

**BEGIN {**

**# tailor this to suit your system**

**\_pw\_awklib = "/usr/local/libexec/*awk*/"**

**}**

**function \_pw\_init( oldfs， oldrs， olddol0， pwcat， using\_fw， using\_fpat)**

**{**

**if (\_pw\_inited)**

**return**

**oldfs = FS**

**oldrs = RS**

**olddol0 = $0**

**using\_fw = (PROCINFO["FS"] == "FIELDWIDTHS")**

**using\_fpat = (PROCINFO["FS"] == "FPAT")**

**FS = ":"**

**RS = "\n"**

**pwcat = \_pw\_awklib "pwcat"**

**while ((pwcat | *getline*) > 0) {**

**\_pw\_byname[$1] = $0**

**\_pw\_byuid[$3] = $0**

**\_pw\_bycount[++\_pw\_total] = $0**

**}**

**close(pwcat)**

**\_pw\_count = 0**

**\_pw\_inited = 1**

**FS = oldfs**

**if (using\_fw)**

**FIELDWIDTHS = FIELDWIDTHS**

**else if (using\_fpat)**

**FPAT = FPAT**

**RS = oldrs**

**$0 = olddol0**

**}**

BEGIN 规则将私有变量设置为 pwcat 所在的目录。因为它用于辅助 awk 的库例程。我们已经将其放在 /usr/local/libexec/***awk*** 中，但是，你可能想放到你的系统中的另外一个位置。

函数 \_pw\_init() 会将用户信息的三个复本填入到三个相关的数组。数组由用户名进行（\_pw\_byname)，用户 ID（\_pw\_byuid)，以及出现的顺序（\_pw\_bycount）进行索引。变量 \_pw\_inited 为了高效而设置，因此 \_pw\_init() 只需要调用一次。

由于函数使用 getline 来从 pwcat 中读取信息，它首先将 FS，RS 与 $0 的值存起来。使用 using\_fw 变量来表示域是否是用 FIELDWIDTHS 来进行分割。这么做是有必要的，因为这些函数也可从用户程序的任何地方进行调用，并且用户有它自己的记录域的分割方式。这样，通过这样的方式使我们可以恢复正确的分割机制。测试只针对 gawk。在某此 awk 实现中，如果使用 FS 或者 FPAT，则这个值为 false。

检查是否使用 FPAT 的代码利用 using\_fpat 变量与 PROCINFO["FS"] 也是类似的情况。

函数的主体部分使用循环来读取数据库行，并将它们分割为不同的域，然后将行存储到数组中以备使用。当循环结束时，\_pw\_init() 通过关闭管线清理，并将 \_pw\_inited 设置为 1，然后恢复 FS 的值（如果必要的情况下也恢复 FIELDWIDTHS 与 FPAT），RS 的值与 $0 的值。\_pw\_count 的使用后面再解释。

getpwnam() 函数取用户名的字串为参数。如果用户在数据库中，则其返回相应的行。否则，通过指向一个不存在的元素的引用会创建一个值为空串的元素。

**function getpwnam(name)**

**{**

**\_pw\_*init()***

**return \_pw\_byname[name]**

**}**

类似地，getpwuid() 函数使用用户的 ID值作为参数。如果用户数组在数据库中，则返回相应的行，否则返回一个空串：

**function getpwuid(uid)**

**{**

**\_pw\_*init()***

**return \_pw\_byuid[uid]**

**}**

getpwent() 函数简单地遍历数据库，一次一项。它使用 \_pw\_count 来跟踪 \_pw\_bycount 数组中的位置：

**function *getpwent()***

**{**

**\_pw\_*init()***

**if (\_pw\_count < \_pw\_total)**

**return \_pw\_bycount[++\_pw\_count]**

**return ""**

**}**

endpwent() 函数重新将 \_pw\_count 设置为 0，所以后续对于 getpwent() 的调用会重头开始：

**function *endpwent()***

**{**

**\_pw\_count = 0**

**}**

在这一组例程的一致设计决策是每一个例程都调用 \_pw\_init() 来初始化数组库数组。运行另一个进程来产生用户数组库，以及扫描数据库的负担仅在用户的主程序程序调用其中的某个函数才会产生。如果这个库文件与用户程序一起装载，但是又没有调用其中的任何例程，则不会有任何的运行时负担。（另一个选择是将 \_pw\_init() 的执行体移到 BEGIN 规则中，在这里部会执行 pwcat。这简化了代码，但是会运行一些从来不会需要的额外过程。）

再者，调用 \_pw\_init() 代码也不高，因为 \_pw\_inited 变量会使程序从数据库中读取数据次数不会出于一次。如果你担心这样的测试会遍历在你的程序中，你也可以将检查 \_pw\_inited 的代码表从 \_pw\_***init()*** 移出去，并复制到所有的其他函数中。在实践中，这是没有必要的，因为大多数的 awk 程序都是 I/O 型的，这样改变会弄乱代码。

在 ，***在第 262 页***，中的 id 程序会使用到这些函数。

## 10.6 读取组数据库

在 ，***在第 242 页***，中讨论的很多问题也都可以用于组数据库中。尽管传统上以都了解的格式与文件存储，但是 POSIX 标准只提供了一组 C 库函数（<grp.h> 与 getgrent()）来访问信息。就算存在这些文件，但是也可能没有完整的信息。因此，与用户数据库一样，也有必要来写一个小的 C 程序来产生组信息。grcat 则是一个这样的 C 程序来用 “cat” 组数据库，如下：

**/\***

**\* grcat.c**

**\***

**\* Generate a printable version of the group database.**

**\*/**

**#include <stdio.h>**

**#include <grp.h>**

**intmain(int argc， char \*\*argv)**

**{**

**struct group \*g;**

**int i;**

**while ((g = *getgrent()*) != NULL) {**

***printf*("%s:%s:%ld:"， g->gr\_name， g->gr\_passwd，(long) g->gr\_gid);**

**for (i = 0; g->gr\_mem[i] != NULL; i++) {**

***printf*("%s"， g->gr\_mem[i]);**

**if (g->gr\_mem[i+1] != NULL)**

**putchar(’，’);**

**}**

**putchar(’\n’);**

**}**

***endgrent()*;**

**return 0;**

**}**

在组数据库中的一行代表一个组。域则通过冒号来分隔，表示如下信息：

Group Name

组名。

Group Password

组的加密密码。在实践中，这个域不会使用到，通过为空或者设置为 ‘\*’。

Group ID Number

组的数值 ID，将名字与数值关联在文件中必须是唯一的。（在一些系统上，它是 C 的 long 类型，而不是 int 类型。所以，我们都将其转换为 long。）

Group Member List

一个逗号分隔的用户列表。这些用户是组的成员。现代的 Unix 系统允许用户同时是多个组的成员。如果你的系统可以支持，则在 PROCINFO 中会为这些组 ID 存储元素 “group1” 到 “groupN”。（注意，PROCINFO 是一个 gawk 扩展，查看 ， ***在第 155 页***）

运行 grcat 时产生的输出可能如下：

**$ grcat**

**-| wheel:\*:0:arnold**

**-| nogroup:\*:65534:**

**-| daemon:\*:1:**

**-| kmem:\*:2:**

**-| staff:\*:10:arnold，miriam，andy**

**-| other:\*:20:**

**...**

这些函数用来从组数据库中获取信息。有几个，按照 C 库函数的方式来命名：

**# group.*awk* --- functions for dealing with the group file**

**BEGIN {**

**# Change to suit your system**

**\_gr\_awklib = "/usr/local/libexec/*awk*/"**

**}**

**function \_gr\_init( oldfs， oldrs， olddol0， grcat，using\_fw， using\_fpat， n， a， i)**

**{**

**if (\_gr\_inited)**

**return**

**oldfs = FS**

**oldrs = RS**

**olddol0 = $0**

**using\_fw = (PROCINFO["FS"] == "FIELDWIDTHS")**

**using\_fpat = (PROCINFO["FS"] == "FPAT")**

**FS = ":"**

**RS = "\n"**

**grcat = \_gr\_awklib "grcat"**

**while ((grcat | *getline*) > 0) {**

**if ($1 in \_gr\_byname)**

**\_gr\_byname[$1] = \_gr\_byname[$1] "，" $4**

**else**

**\_gr\_byname[$1] = $0**

**if ($3 in \_gr\_bygid)**

**\_gr\_bygid[$3] = \_gr\_bygid[$3] "，" $4**

**else**

**\_gr\_bygid[$3] = $0**

**n = split($4， a， "[ \t]\*，[ \t]\*")**

**for (i = 1; i <= n; i++)**

**if (a[i] in \_gr\_groupsbyuser)**

**\_gr\_groupsbyuser[a[i]] = gr\_groupsbyuser[a[i]] "" $1**

**else**

**\_gr\_groupsbyuser[a[i]] = $1**

**\_gr\_bycount[++\_gr\_count] = $0**

**}**

**close(grcat)**

**\_gr\_count = 0**

**\_gr\_inited++**

**FS = oldfs**

**if (using\_fw)**

**FIELDWIDTHS = FIELDWIDTHS**

**else if (using\_fpat)**

**FPAT = FPAT**

**RS = oldrs**

**$0 = olddol0**

**}**

BEGIN 规则将存储 grcat 所有的目录存储在私有变量中。因此这用于指出 awk 库函数的位置。我们将其放在 /usr/local/libexec/***awk*** 中。可能你会选择存在另一个地方。

这些例程遵循与用户数据库例程一样的框架（查看 ， ***在第 242 页***）。\_gr\_inited 变量用于保证数据库只被扫描一次。\_gr\_init() 函数首先存储 FS，RS 以及 $0 值，然后将 FS 与 RS 的值设置成正则扫描组信息的格式。同时也会注意是否是用 FIELDWIDTHS 或者 FPAT，后面以用来恢复恰当的域分割机制。

组信息存储在几个关联的数组中。数组由组名（\_gr\_byname)，组ID（\_gr\_bygid），以及在数据库中的位置（\_gr\_bycount）来索引。还有一个数组由用户名来索引（\_gr\_groupsbyuser），这是每个用户所属的组，每一个用空格分割。

与用户数据库不同，它有可能在一个数据库有多个记录有相同的组名。当这个组有很多的成员的时候这很正常。两个这样的项看起来像：

**tvpeople:\*:101:johnny，jay，arsenio**

**tvpeople:\*:101:david，conan，tom，joan**

因为这个，\_gr\_init() 要查看是否有相同的组名与组 ID 值。如果有，用户名只是简单地连接到前面的用户列表中。[[79]](#footnote-80)

最后，\_gr\_init() 关闭 grcat 的管线，恢复 FS（如果有必要也恢复 FIELDWIDTHS 或者 FPAT），RS 与 $0 的值，将 \_gr\_count 的值恢复为 0（后面会使用到），并使 \_gr\_inited 为非零。

getgrnam() 函数取组名为参数，如果存在相应的组，则返回。否则其所依赖的数组，会以空串创建一个元素：

**function getgrnam(group)**

**{**

**\_gr\_*init()***

**return \_gr\_byname[group]**

**}**

getgrgid() 函数类似，以数值的组 ID 为参数，来查找与这个 ID 相关的组的信息：

**function getgrgid(gid)**

**{**

**\_gr\_*init()***

**return \_gr\_bygid[gid]**

**}**

getgruser() 函数没有相应的 C 例程对应。它取用户名为参数，并返回该组中的成员用户的列表：

**function getgruser(user)**

**{**

**\_gr\_*init()***

**return \_gr\_groupsbyuser[user]**

**}**

getgrent() 函数迭代数据库，一次一个。它使用 \_gr\_count 来跟踪在列表中的位置：

**function *getgrent()***

**{**

**\_gr\_*init()***

**if (++\_gr\_count in \_gr\_bycount)**

**return \_gr\_bycount[\_gr\_count]**

**return ""**

**}**

endgrent() 函数将 \_gr\_count 设置为 0，因此 getgrent() 可以重头开始：

**function *endgrent()***

**{**

**\_gr\_count = 0**

**}**

与用户数据库列表相似，每一个函数都调用 \_gr\_init() 来初始化数组。这么做，使用这些函数被使用时导致一些额外的负担（相对于将 \_gr\_init() 的执行体移到 BEGIN 规则中）。

大多数的工作都集中于扫描数据库，并且建立多个关联数组。用户自己所调用的函数则相当简单，这依赖于 awk 的关联数组所做的工作。

在 ，***在第 262 页***，中的 id 程序会使用到这些函数。

## 10.7 遍历数组之数组

在 ，***在第 178 页***，中描述了 gawk 如何来提供数组之数组。特别是，数组的任意元素即可以是标量，也可以是另一个数组。isarray() 函数（查看 ， ***在第 203 页***）可以让你将标量与数组区分。下面的函数，walk\_array() 递归地遍历数组，打印数组的索引以及值。你可以通过数组与代表这个数组的字串名来调用这个函数：

**function walk\_array(arr， name， i)**

**{**

**for (i in arr) {**

**if (isarray(arr[i]))**

**walk\_array(arr[i]， (name "[" i "]"))**

**else**

***printf*("%s[%s] = %s\n"， name， i， arr[i])**

**}**

**}**

函数通过循环迭代数组的每一个元素来工作。如果数组的元素是一个数组，则函数会递归地调用自己，将子数组与表示当前索引的字串传递过去。否则，函数简单地打印元素的名字，索引以及它的值。这里是演示的主程序：

**BEGIN {**

**a[1] = 1**

**a[2][1] = 21**

**a[2][2] = 22**

**a[3] = 3**

**a[4][1][1] = 411**

**a[4][2] = 42**

**walk\_array(a， "a")**

**}**

当执行时，产生下面的输出：

**$ gawk -f walk\_array.*awk***

**-| a[1] = 1**

**-| a[2][1] = 21**

**-| a[2][2] = 22**

**-| a[3] = 3**

**-| a[4][1][1] = 411**

**-| a[4][2] = 42**

函数仅是简单地打印名字与每个标量的数组元素。但是，也很容易来进行通用化，只要传递一个要遍历数组的函数名就可以做到。修改过后的函数看起来如下：

**function process\_array(arr， name， process， do\_arrays， i， new\_name)**

**{**

**for (i in arr) {**

**new\_name = (name "[" i "]")**

**if (isarray(arr[i])) {**

**if (do\_arrays)**

**@process(new\_name， arr[i])**

**process\_array(arr[i]， new\_name， process， do\_arrays)**

**} else**

**@process(new\_name， arr[i])**

**}**

**}**

参数如下：

arr 相应的数组。

name 数组名（字串）。

process 要调用的函数名。

do\_arrays

如果为 true，函数可以处理为子数组的元素。

如果子数组要被处理，则在进一步迭代时进行处理。

当展开执行时，函数产生与早前的 walk\_array() 函数一样的输出结果：

**BEGIN {**

**a[1] = 1**

**a[2][1] = 21**

**a[2][2] = 22**

**a[3] = 3**

**a[4][1][1] = 411**

**a[4][2] = 42**

**process\_array(a， "a"， "do\_print"， 0)**

**}**

**function do\_print(name， element)**

**{**

***printf* "%s = %s\n"， name， element**

**}**

## 10.8 总结

* 阅读程序是学习编程的最好方法。在这章以及下一章中提供的函数以及程序就用于这样的目的。
* 当写通用目的的库函数时，要用一此些规则来命令任意的全局变量，这样它他不会与用户程序的变量冲突。
* 这里呈现的函数可以分为下面的类型：

一般问题

数组到字串的转换，测试断言，舍入舍出，随机数产生，将字符转换为数组，连接字串，方便地取得时间间信息，一次读取所有文件。

管理数据文件

提示数组文件的边界，重读当前文件，检查文件的可读性，检查0长度文件，以及将赋值当成文件名处理。

处理命令行选项

awk 版本的标准 C 函数 getopt()。

读取用户与组数据库

两个与 C 版本的库函数相同的函数集。

遍历数

两个函数用来进行任意深度的数组之数组的遍历。

## 10.9练习

1. 在 ，***在第 235 页***，中我们呈现了 ***zerofile.awk*** 程序，这个程序利用 gawk 的 ARGIND 变量。这个问题是否不依赖 ARGIND 变量就可以解决？如果可以，如何解决？
2. 作为相关的挑战，修改代码来处理这样的情形，在 ARGV 中的插入值是一个变量赋值。
3. 在 ，***在第 249 页***，中呈现了一个遍历一个多维数组并打印其元素的函数。但是，遍历与处理数组的每一个元素是一个通用的操作。通用化 walk\_array() 函数只需要加入名为 process 的额外参数。

然后，在循环中，将打印数组元素索引与值的操作，替换为以间接函数调用 process 的语法（查看 ， ***在第 213 页***），并将索引与值传递进去。

当调用 walk\_array() 时，你要传递用户自定义的函数，该函数期望接收索引与值，然后来处理相应的元素。通过打印数组来测试你的新版本的代码，你打印的格式应与最初版本的打印格式相同。

# 第十一章 实用的 awk 程序

在 ***，在第 221 页***，中呈现了读程序的代码来进行程序语言的学习的观点。这一章继续这样的主题，并通过呈现各种各样的 awk 程序以享读者。这一章有三个小节。第一节描述如可来进行这一章中提供的程序。第二小节呈现 awk 版本的几个通用的 POSIX 工具。这些程序希望你也已经熟悉了，并且也了解了这些工具的问题。通过用 awk 重新实现这些工具，你可以将焦点集中于 awk 相关的解决程序问题的方面。

第三节则是一些有趣的程序集合。这些程序解决了很多数组处理与管理相关的问题。很多的程序都很短，以用于强调 awk 用几行代码即可处理很多事情的能力。这当中的很多程序会使用在 ***，在第 221 页***，中出现过的库函数。

## 11.1 运行例子程序

要运行一个程序，典型的情况下你输入下面的代码：

***awk* -f program -- options files**

在这里，program 是 awk 程序的名字（如 cut.awk），选项则是任意的 program 命令行选项，以 ‘-’ 开头，而 files 则是实际的数据文件。

如果你的系统支持 ‘#!’ 可执行解释器机制（查看 ， ***在第 22 页***），你可以直接这么输入：

**cut.*awk* -c1-8 myfiles > results**

如果 awk 不是 gawk，你可能要这么来使用：

**cut.*awk* -- -c1-8 myfiles > results**

## 11.2 为快乐与评测重新发明轮子

本节中呈现了几个用 awk 语言实现的 POSIX 工具。重新用 awk 来发明这些程序，只是好玩，因此算法可以被清晰地表达，而且代码一般来说都非常精简与简单。因为 awk 确实为你做了很多的事情了。

也要注意到，这些程序没有必要要去替换系统中已经安装的版本。其中也没有程序被完全地实现了最近的 POSIX 标准。但这不是个问题，它们的目的是用于解释 awk 程序语言为“现实世界”的任务的编程方式。程序以字母顺序呈现。

### 11.2.1 剪切域与列

cut 这个工具从它的标准输入中选择，或者说“切分”，字符或者域并输出这些字符到它的标准输出中。域默认是通过 TABs 来进行分隔的，但是你可以提供一个命令行参数来改变域界定符（即域分隔符）。cut 的域定义没有 awk 的域那么一般化。

例如 who 的输出，下面的管线会产生有序的，唯一的登录到系统中的用户列表：

**who | cut -c1-8 | sort | uniq**

cut 的选项有：

-c list 使用 list 作为要切分的字符列表。在 list 的项可以用逗号分隔，某个范围的字符可以用横杠来分隔。list “1-8，15，22-35‘ 指定了 1 到 8 的字符，15 以及 22 到 35 之间的字符。

-f list 使用 list 作为要切分的域列表。

-d delim 使用 delim 作用域分隔符，而不是使用 TAB 字符。-s 会抑制打印那些不含有域界定符的行。

awk 的 cut 实现使用 getopt() 库函数（查看 ， ***在第 237 页***）以及 join() 库函数（查看 ， ***在第 228 页***）。

程序以描述选项的注释开始，需要的库函数以及一个 usage() 函数来打印使用信息后退出。usage() 在遇到一个无效的参数时会被调用：

**# cut.*awk* --- implement cut in *awk***

**# Options:**

**# -f list Cut fields**

**# -d c Field delimiter character**

**# -c list Cut characters**

**#**

**# -s Suppress lines without the delimiter**

**#**

**# Requires *getopt()* and *join()* library functions**

**function *usage()***

**{**

**print("usage: cut [-f list] [-d c] [-s] [files...]") >"/dev/stderr"**

**print("usage: cut [-c list] [files...]") >"/dev/stderr"**

**exit 1**

**}**

接下来就轮到 BEGIN 规则，在里面对命令行进行选项分析。它将 FS 设置为单个的 TAB 字符，因此那是 cut 的默认域分隔符。使用 getopt() 来进行循环命令行中的选项。by\_fields 与 by\_chars 中只有一个变量被设置为 true，以指示处理是通过域来完成还是通过字符来完成。当以字符来切分时，输出域分隔符被设置为空串：

**BEGIN {**

**FS = "\t" # default**

**OFS = FS**

**while ((c = getopt(ARGC， ARGV， "sf:c:d:")) != -1) {**

**if (c == "f") {**

**by\_fields = 1**

**fieldlist = Optarg**

**} else if (c == "c") {**

**by\_chars = 1**

**fieldlist = Optarg**

**OFS = ""**

**} else if (c == "d") {**

**if (length(Optarg) > 1) {**

***printf*("cut: using first character of %s" \**

**" for delimiter\n"， Optarg) >"/dev/stderr"**

**Optarg = substr(Optarg， 1， 1)**

**}**

**FS = Optarg**

**OFS = FS**

**if (FS == "") # defeat *awk* semantics**

**FS = "[ ]"**

**} else if (c == "s")**

**suppress = 1**

**else**

***usage()***

**}**

**# Clear out options**

**for (i = 1; i < Optind; i++)**

**ARGV[i] = ""**

代码必须小心处理域分界符为空格的情况。使用单独的空格（“ ”）来作为 FS 的值是不正确的——awk 会用连续的空格、TABs 与/或换行符来进行域分割，但我们是想用单独的空格来进行切分。同时也要记住当 getopt() 完成（在 ， ***在第 237 页***，中进行了描述），我们要清除在 ARGV 中 1 到 Optind 中的所有元素，这样 awk 不会将这些命令行参数当成文件进行处理。

在处理了命令行选项后，程序要验证选项是否是有用的。只有 -c 与 -f 之一被使用，而两者都需要一个域列表。然后程序调用 set\_fieldlist() 或者 ***set\_charlist()*** 来将域或者字符表进行拆分：

**if (by\_fields && by\_chars)**

***usage()***

**if (by\_fields == 0 && by\_chars == 0)**

**by\_fields = 1 # default**

**if (fieldlist == "") {**

**print "cut: needs list for -c or -f">"/dev/stderr"**

**exit 1**

**}**

**if (by\_fields)**

**set\_*fieldlist()***

**else**

**set\_*charlist()***

**}**

***set\_ fieldlist ()*** 将列表以逗号进行分割，并存储到数组中。然后对数组中的每个元素，查看其是否是一个范围，如果是，则将其分开。函数检查范围以确保第一个数值是小于第二个数值的。在列表中每一个数值都会加到 flist 数组中，这个数组只是简单地记录要打印的域。使用的是常规的域分割策略。程序使 awk 处理域分割的工作：

**function set\_fieldlist( n， m， i， j， k， f， g)**

**{**

**n = split(fieldlist， f， "，")**

**j = 1 # index in flist**

**for (i = 1; i <= n; i++) {**

**if (index(f[i]， "-") != 0) { # a range**

**m = split(f[i]， g， "-")**

**if (m != 2 || g[1] >= g[2]) {**

***printf*("cut: bad field list: %s\n"，**

**f[i]) >"/dev/stderr"**

**exit 1**

**}**

**for (k = g[1]; k <= g[2]; k++)**

**flist[j++] = k**

**} else**

**flist[j++] = f[i]**

**}**

**nfields = j - 1**

**}**

***set\_charlist()*** 函数要比 set\_fieldlist() 要复杂得多。这里的想法是利用 gawk 的 FIELDWIDTHS 变量（查看 ， ***在第 77 页***），这个变量描述了输入的常量宽度。当使用字符列表时，这正是我们所需要的。

设置 FIELDWIDTHS 的值要比简单地列表那些要打印的域要简单得多。我们要跟踪要打印的域以及要跳过的中间字符。例如，假设你想要的字符为 1 到 8，15，以及 22 到 35。可需要使用 ‘-c 1-8，15，22-35’。FIELDWIDTHS 的值应该是 “8 6 1 6 14”。这会产生 5 个域，要打印的域为 $1，$3 与 $5。中间的域则是填充符，是所需要的域之间的字符。flist 列出了要打印的域，t 则是用来跟踪完整的域列表，包括填充域：

**function set\_charlist( field， i， j， f， g， n， m， t，**

**filler， last， len)**

**{**

**field = 1 # count total fields**

**n = split(fieldlist， f， "，")**

**j = 1 # index in flist**

**for (i = 1; i <= n; i++) {**

**if (index(f[i]， "-") != 0) { # range**

**m = split(f[i]， g， "-")**

**if (m != 2 || g[1] >= g[2]) {**

***printf*("cut: bad character list: %s\n"，**

**f[i]) >"/dev/stderr"**

**exit 1**

**}**

**len = g[2] - g[1] + 1**

**if (g[1] > 1) # compute length of filler**

**filler = g[1] - last - 1**

**else**

**filler = 0**

**if (filler)**

**t[field++] = filler**

**t[field++] = len # length of field**

**last = g[2]**

**flist[j++] = field - 1**

**} else {**

**if (f[i] > 1)**

**filler = f[i] - last - 1**

**else**

**filler = 0**

**if (filler)**

**t[field++] = filler**

**t[field++] = 1**

**last = f[i]**

**flist[j++] = field - 1**

**}**

**}**

**FIELDWIDTHS = join(t， 1， field - 1)**

**nfields = j - 1**

**}**

下一个则是处理数据的规则。如果指定了 -s 选项，则 suppress 为 true。第一个 if 语句确保输入的记录中确实包含域分隔符。如果 cut 正在处理域，而 suppress 为 true，并且域分隔字符不在记录中，则这条记录会被跳过。

如果记录是有效的，则 gawk 会将记录分割成域，要么使用的是 FS 中的字符，要么是使用固定宽度的方式，以及 FIELDWIDTHS 方式。循环迭代那些要被打印的域列表。如果其包含相应数据，则打印相应的域。如果下一个域也包含数据，则在两个域之间输出分隔符：

**{**

**if (by\_fields && suppress && index($0， FS) == 0)**

**next**

**for (i = 1; i <= nfields; i++) {**

**if ($flist[i] != "") {**

***printf* "%s"， $flist[i]**

**if (i < nfields && $flist[i+1] != "")**

***printf* "%s"， OFS**

**}**

**}**

**print ""**

**}**

这个版本的 cut 依赖于 gawk 的 FIELDWIDTHS 变量来做基于字符的切分。也可以在其他的 awk 实现中使用 substr()（查看 ， ***在第 184 页***），但是那实现起来也很痛苦。FIELDWIDTHS 变量提供了一种很优雅的解决方法来处理被字符分割的输入行。

### 11.2.2 用正则表达式在文件中搜索

egrep 工具要来按模式搜索文件内容。它使用的正则表达式几乎与 awk 所使用的一样（查看 ***， 在第 50 页***）。你这么来调用：

**egrep [options] 'pattern' files . . .**

pattern 是一个正则表达式。在典型的使用情况下，正则表达式会被引起来，以避免Shell 将一些特殊字符扩展为文件名的通配符。一般情况下，egrep 打印匹配的行。如果在命令行中提供了多个文件，则每个输出行前都带上文件名，后面跟着冒号：

egrep 的选项如下：

-c 打印匹配 pattern 的行的总数，而不是匹配行本身。

-s 静默处理，不产生输出，并且用退出值来表示是否有匹配的行。

-v 对匹配测试取反。egrep 打印那些不匹配模式的行，并且如果匹配没有成功，则以成功退出。

-i 忽略 pattern 与输入数据的大小写。

-l 只打印匹配的文件名，而不是匹配的行。

-e pattern

使用 pattern 为要匹配的正则表达式。-e 选项的目的是允许模式可以以 ‘-’ 开始。

这个版本使用 getopt() 库函数（查看 ， ***在第 237 页***）以及文件转换库程序（查看 ， ***在第 233 页***）。

程序以描述性的注释开始，然后是 BEGIN 规则调用 getopt() 函数来处理命令行参数。 -i（忽略大小写）选项对于 gawk 来说尤其简单，我们只使用预定义的 IGNORECASE 变量就可以实现（查看 ， ***在第 155 页***）：

**# egrep.*awk* --- simulate egrep in *awk***

**#**

**# Options:**

**# -c count of lines**

**# -s silent - use exit value**

**# -v invert test， success if no match**

**# -i ignore case**

**# -l print filenames only**

**# -e argument is pattern**

**#**

**# Requires getopt and file transition library functions**

**BEGIN {**

**while ((c = getopt(ARGC， ARGV， "ce:svil")) != -1) {**

**if (c == "c")**

**count\_only++**

**else if (c == "s")**

**no\_print++**

**else if (c == "v")**

**invert++**

**else if (c == "i")**

**IGNORECASE = 1**

**else if (c == "l")**

**filenames\_only++**

**else if (c == "e")**

**pattern = Optarg**

**else**

***usage()***

**}**

接下来则是处理 egrep 相关的行为。如果没有使用 -e 选项提供 pattern 参数，则会使用命令行中的第一个非选项参数。awk 中到 ARGV[Optind] 中的参数会被清空，所以 awk 不会尝试将它们当成文件进行处理。如果没有指定文件，则使用标准的输入，而如果指定的多个文件，我们要确保要看到这种情况，因此文件名要出现在输出的行之前：

**if (pattern == "")**

**pattern = ARGV[Optind++]**

**for (i = 1; i < Optind; i++)**

**ARGV[i] = ""**

**if (Optind >= ARGC) {**

**ARGV[1] = "-"**

**ARGC = 2**

**} else if (ARGC - Optind > 1)**

**do\_filenames++**

**# if (IGNORECASE)**

**# pattern = tolower(pattern)**

**}**

最后两行被注释掉了，因为在 gawk 中不需要。如果你使用其他版本的 awk，则你可能需要将他们反注释回来。

如果你不是使用的 gawk，下面是要被反注释掉的行。这个规则在指定了 -i 参数时，将所有的输入行中的字符转换为小写。[[80]](#footnote-81)它们被注释掉，是因为这些代码在 gawk 中使用。

**#{**

**# if (IGNORECASE)**

**# $0 = tolower($0)**

**#}**

当每处理一个新文件的时候，beginfile() 函数在 ftrans.awk 中被调用。在这种情况下，是非常简单的，所要要做的事情就是初始化变量 fcount 为 0。fcount 用来跟踪当前的文件中有多少行匹配 pattern。名字为 junk 的参数显示 beginfile() 会使用一个参数来调用，但是我们对其值不感兴趣：

**function beginfile(junk)**

**{**

**fcount = 0**

**}**

当每一个文件被处理完后，会调用 endfile()。它只会在用户想知道有多少行匹配时才会影响输出。no\_print 需要退出状态时才为会 true。count\_only 如果要统计时才会为 true。如果打印与统计被启用，egrep 因此只打印行统计数。输出的格式必须根据要处理的文件进行处理。最后，fcount 用来进行加总，因此我们可以知道总共有多少行匹配了 pattern：

**function endfile(file)**

**{**

**if (! no\_print && count\_only) {**

**if (do\_filenames)**

**print file ":" fcount**

**else**

**print fcount**

**}**

**total += fcount**

**}**

BEGINFILE 与 ENDFILE 这两个特殊模式（查看 ， ***在第 143 页***）也可以使用，但是程序必须是用 gawk 来运行的。另外，这个例子写于 gawk 加入 BEGINFILE 与 ENDFILE 特性之前。

下面的规则处理了大部分的匹配行的工作。变量 matches 在行匹配 pattern 时为真。如果用户想要不匹配的行，则 matches 的意义就用 ‘!’ 被反过来使用。fcount 会用 matches 的值一起递增，根据匹配成功与否，其要么是 0，要么是 1。如果行不匹配，则 next 语句会调用，用于处理下一条记录。

下面执行了很多的测试，但是它们只会在我们不进行行统计时执行。首先，如果用户想要的是退出状态（no\_print 为 true），则只需要知道文件中的某行是否匹配就可以了，我们可以用 nextfile 来跳到下一个文件。相似地，如果我们仅需要打印文件名，我们则打印文件名后用 nextfile 语句来跳到下一个文件。最后，我们会打印每一行，如果需要的话，在其前面加入文件我名字与冒号：

**{**

**matches = ($0 ~ pattern)**

**if (invert)**

**matches = ! matches**

**fcount += matches # 1 or 0**

**if (! matches)**

**next**

**if (! count\_only) {**

**if (no\_print)**

**nextfile**

**if (filenames\_only) {**

**print FILENAME**

**nextfile**

**}**

**if (do\_filenames)**

**print FILENAME ":" $0**

**else**

**print**

**}**

**}**

END 规则要处理好正确的返回状态。如果没有匹配，则退出状态为 1，否则为 0：

**END {**

**exit (total == 0)**

**}**

usage() 函数用来在输入无效选项的时候打印使用信息，然后退出：

**function *usage()***

**{**

**print("Usage: egrep [-csvil] [-e pat] [files ...]") >"/dev/stderr"**

**print("\n\tegrep [-csvil] pat [files ...]") >"/dev/stderr"**

**exit 1**

**}**

### 11.2.3 打印用户信息

id 工具列出用户的真实与有效的用户 ID 值，真实与有效的组 ID 值，如果有的话，包含用户的组集。id 只有在它同于真实的ID 的情况下，才打印有效的用户 ID 与组 ID。如果可能，id 也提供了相应的前户与组名。输出可能如下：

**$ id**

**-| uid=1000(arnold) gid=1000(arnold) groups=1000(arnold)，4(adm)，7(lp)，27(sudo)**

这里的信息一部份是由 gawk 的 PROCINFO 数组来提供的（查看 ， ***在第 155 页***）。但是，id 工具会提供比单纯的数组更好的输出。

这里是 id 的一个简单的版本，用 awk 写成。它使用来自于 ***，在第 221 页***，的用户数据库函数（查看 ， ***在第 242 页***） 以及组数据库函数（查看 ， ***在第 246 页***）。

程序相当直接。所有的工作都在 BEGIN 规则中完成。用户以及组 ID 值从 PROCINFO 中取得。代码是可重复的。在用户数据库中的用于真实用户ID 的项被用 ‘:’ 分割成不同的部分。第一项是名字。相应的代码也可以用于有效的用户 ID 与组相关的值：

**# id.*awk* --- implement id in *awk***

**#**

**# Requires user and group library functions**

**# output is:**

**# uid=12(foo) euid=34(bar) gid=3(baz) \**

**# egid=5(blat) groups=9(nine)，2(two)，1(one)**

**BEGIN {**

**uid = PROCINFO["uid"]**

**euid = PROCINFO["euid"]**

**gid = PROCINFO["gid"]**

**egid = PROCINFO["egid"]**

***printf*("uid=%d"， uid)**

**pw = getpwuid(uid)**

**pr\_first\_field(pw)**

**if (euid != uid) {**

***printf*(" euid=%d"， euid)**

**pw = getpwuid(euid)**

**pr\_first\_field(pw)**

**}**

***printf*(" gid=%d"， gid)**

**pw = getgrgid(gid)**

**pr\_first\_field(pw)**

**if (egid != gid) {**

***printf*(" egid=%d"， egid)**

**pw = getgrgid(egid)**

**pr\_first\_field(pw)**

**}**

**for (i = 1; ("group" i) in PROCINFO; i++) {**

**if (i == 1)**

***printf*(" groups=")**

**group = PROCINFO["group" i]**

***printf*("%d"， group)**

**pw = getgrgid(group)**

**pr\_first\_field(pw)**

**if (("group" (i+1)) in PROCINFO)**

***printf*("，")**

**}**

**print ""**

**}**

**function pr\_first\_field(str， a)**

**{**

**if (str != "") {**

**split(str， a， ":")**

***printf*("(%s)"， a[1])**

**}**

**}**

在循环中的测试值得注意一下。任意在 PROCINFO 数组中的补充的值都有索引从 “group1” 到某个值 N 的 “groupN”（即补充组的总数）。但是，我们事先不知道有多少个这样的组。

循环从 1 开始，将值是 “group” 进行连接，然后使用 in 操作符来查看相应的值是否在数组中（查看 ， ***在第 168 页***）。最后，i 会递增越过数组中的最后一个组，然后循环退出。

循环在没有补充组的情况下也是正确的，这样第一次测试时条件就为 false，这样循环就根本没有执行过。

***pr\_first\_field()*** 函数简单地隔离一些要重复使用的代码，使得整个程序更短更清晰。特别是将检测空串的代码移进个函数中。

### 11.2.4 将大文件分片

split 工具将大的文本文件分割成更小了几个。用法如下：[[81]](#footnote-82)

**split [-count] [file] [prefix]**

默认情况下，输出文件命名为 xxa，xab 等等。每个文件有一千行，最后一个文件例外 。为了每个文件中的行数，在命令行中输入一个数值，前面带一个减号（如 ‘-500’ 使每个文件中有 500 行，而不是 1000 行）。为了改变输出文件的名字为诸如 myfileaa，myfileab 等等，则提供一个额外的参数来指定文件名的前缀。

下面是 awk 版本的 split。它使用在 ， ***在第 227 页***，中的 ord() 与 chr() 函数。

程序首先设置它的默认行为，然后测试以确保不会有太多的参数。然后依次查看每个参数。第一个参数可以一个负号，后面跟一个数值。如果果，这看起来就像是一个负数，所以使其为正数，即是行的数值。数据文件名被忽略过去，最后一个参数用于输出文件名的前缀：

**# split.*awk* --- do split in *awk***

**#**

**# Requires *ord()* and *chr()* library functions**

**# usage: split [-count] [file] [outname]**

**BEGIN {**

**outfile = "x" # default**

**count = 1000**

**if (ARGC > 4)**

***usage()***

**i = 1**

**if (i in ARGV && ARGV[i] ~ /^-[[:digit:]]+$/) {**

**count = -ARGV[i]**

**ARGV[i] = ""**

**i++**

**}**

**# test argv in case reading from stdin instead of file**

**if (i in ARGV)**

**i++ # skip datafile name**

**if (i in ARGV) {**

**outfile = ARGV[i]**

**ARGV[i] = ""**

**}**

**s1 = s2 = "a"**

**out = (outfile s1 s2)**

**}**

下一个规则做剩下的大部分的事情，tcount （temporary count）跟踪当前已经有多少行输出到了输出文件中。如果其大于 count，则要关闭当前的文件然后开启一个新的文件。s1 与 s2 跟踪当前的文件名前缀。如果他们两个都是 ‘z’，则表示文件太大了。否则，s1 移动到字母表中的下一个字母，s2 则从 ‘a’ 开始：

**{**

**if (++tcount > count) {**

**close(out)**

**if (s2 == "z") {**

**if (s1 == "z") {**

***printf*("split: %s is too large to split\n"，**

**FILENAME) >"/dev/stderr"**

**exit 1**

**}**

**s1 = chr(ord(s1) + 1)**

**s2 = "a"**

**}**

**else**

**s2 = chr(ord(s2) + 1)**

**out = (outfile s1 s2)**

**tcount = 1**

**}**

**print > out**

**}**

usage() 函数简单地打印错误并输出：

**function *usage()***

**{**

**print("usage: split [-num] [file] [outname]") >"/dev/stderr"**

**exit 1**

**}**

这个程序程序有一点滑头，他依赖于 awk 来自动关闭最后一个文件，而不是在 END 规则中来处理。同时，它也假设字母在字符集中是连续的，这在 EBCDIC 的系统中是不成立的。

### 11.2.5 将输出复制到多个文件中

tee 程序是一个“管道适配器”。tee 将他的标准输出到它的标准输出中，并将其复制到命令行中拉丁指定的文件名中。它的用法如下：

**tee [-a] file . . .**

-a 选项告诉 tee 将内容附加到命令的文件中，而不是将这些文件行清除再重新开始。

BEGIN 规则首先生成所有命令行参数的复本到名为 copy 的数组中。ARGV[0] 是不需要的，所以它不会被复制。tee 不可以直接使用 ARGV，因为 awk 会尝试将 ARGV 中的文件名当成输入的数据。

如果第一个参数是 -a，然后标志变量 append 被设置为 true，ARGV[1] 与 copy[1] 两个元素都被删除。如果 ARGC 小于 2，则表示没有提供文件名，则 tee 会打印一个使用信息然后退出。最后 ，awk 被强制从标准输入中读取数据，只要将 ARGV[1] 的值设置为 “-”，同时 ARGC 设置为 2：

**# tee.*awk* --- tee in *awk***

**#**

**# Copy standard input to all named output files.**

**# Append content if -a option is supplied.**

**#**

**BEGIN {**

**for (i = 1; i < ARGC; i++)**

**copy[i] = ARGV[i]**

**if (ARGV[1] == "-a") {**

**append = 1**

**delete ARGV[1]**

**delete copy[1]**

**ARGC--**

**}**

**if (ARGC < 2) {**

**print "usage: tee [-a] file ...">"/dev/stderr"**

**exit 1**

**}**

**ARGV[1] = "-"**

**ARGC = 2**

**}**

下面简单的规则执行所有的工作。由于没有模式，所以它会对每一个输入行都执行。规则的执行体只是简单地打印行到每个命令行上的文件，然后再打印到标准输出中：

**{**

**# moving the if outside the loop makes it run faster**

**if (append)**

**for (i in copy)**

**print >> copy[i]**

**else**

**for (i in copy)**

**print > copy[i]**

**print**

**}**

也可以这么来写循环：

**for (i in copy)**

**if (append)**

**print >> copy[i]**

**else**

**print > copy[i]**

这样更精简，但是却没有这么高效。if 会对每个记录以及每个输出文件都进行测试。通过复制循环体，if 对每个输入记录只执行一次测试。如果有 N 个输入记录，并且有 M 个输出文件，第一种方法只执行 N 个 if 语句，而第二种方法会执行 N\*M 个 if 语句。最后，END 规则通过关闭所有的输出文件来进行清理：

**END {**

**for (i in copy)**

**close(copy[i])**

**}**

### 11.2.6 打印去重文本行

uniq 工具从它的标准输入中读取经过排序的行，并且默认地移除重复的行。也就是说，它只打印唯一的行——如此名字所表示的意思。uniq 有很多的选项。其使用方法如下：

**uniq [-udc [-n]] [+n] [inputfile [outputfile]]**

uniq 的选项如下：

-d 只打印重复的（有复本）的行。

-u 只打印不重复的（唯一的）行。

-c 统计行。这个选项会覆盖 -d 与 -u。重复的与非重复的行都会被统计。

-n 在比较行前跳过 n 个域。域的定义与 awk 的默认的类似：被一连续的空格与/或者 TABs 分隔的非空白字符。

+n 在比较行前跳过 n 个字符。通过 ‘-n’ 指定的任何域会首先被跳过。

inputfile

数据从在命令行中指定的文件中读取，而不是从标准输入读取。

outputfile

将产生的输出发送到 outputfile 命名的文件中，而不是到标准输出中。

一般情况下，uniq 的行为就像 -d 与 -u 两个选项都指定的一样。

uniq 使用 getopt() 库函数（查看 ， ***在第 237 页***）以及 join() 库函数（查看 ， ***在第 228 页***）。

程序以 usage() 函数开始以及一些写在注释中的简单的选项及其含义的说明。BEGIN 规则处命令行参数与选项。它使用了一个技巧来让 getopt() 处理如 ‘-25’ 这个的选项，将这样的选项当做是字母 ‘2’ 的选项与其参数 ‘5’。如果实际上提供了两个或者多个数字（Optarg 看起来像一个数值），Optarg 会与选项数字一连接，然后结果与 0 相加来使其转换为数值。如果只有一个数值在选项中，Optarg 就不需要了。在这种情况下，Optind 必须被递减，这样 getopt() 会在下一轮中处理它。这个代码确实玩了一点花招。

如果没有提供选项，则会使用默认的行为，即打印重复与非重复的行。如果提供了输出文件，将其被赋值给 outputfile。在早期，outputfile 被初始化为标准输出，即 /dev/stdout：

**# uniq.*awk* --- do uniq in *awk***

**#**

**# Requires *getopt()* and *join()* library functions**

**function *usage()***

**{**

**print("Usage: uniq [-udc [-n]] [+n] [ in [ out ]]") >"/dev/stderr"**

**exit 1**

**}**

**# -c count lines. overrides -d and -u**

**# -d only repeated lines**

**# -u only nonrepeated lines**

**# -n skip n fields**

**# +n skip n characters， skip fields first**

**BEGIN {**

**count = 1**

**outputfile = "/dev/stdout"**

**opts = "udc0:1:2:3:4:5:6:7:8:9:"**

**while ((c = getopt(ARGC， ARGV， opts)) != -1) {**

**if (c == "u")**

**non\_repeated\_only++**

**else if (c == "d")**

**repeated\_only++**

**else if (c == "c")**

**do\_count++**

**else if (index("0123456789"， c) != 0) {**

**# *getopt()* requires args to options**

**# this messes us up for things like -5**

**if (Optarg ~ /^[[:digit:]]+$/)**

**fcount = (c Optarg) + 0**

**else {**

**fcount = c + 0**

**Optind--**

**}**

**} else**

***usage()***

**}**

**if (ARGV[Optind] ~ /^\+[[:digit:]]+$/) {**

**charcount = substr(ARGV[Optind]， 2) + 0**

**Optind++**

**}**

**for (i = 1; i < Optind; i++)**

**ARGV[i] = ""**

**if (repeated\_only == 0 && non\_repeated\_only == 0)**

**repeated\_only = non\_repeated\_only = 1**

**if (ARGC - Optind == 2) {**

**outputfile = ARGV[ARGC - 1]**

**ARGV[ARGC - 1] = ""**

**}**

**}**

下面的函数 are\_equal() 用来将当前行，$0，与前一行 last 进行比较。它处理要跳过的域与字符。如果没有指定域统计以及字符统计，are\_equal 返回 1，或者 0，这依赖于 last 与 $0 字串的简单的比较结果。

不然的话，事情会变量更加复杂。如果要跳过域，每一行都会用 split()（查看 ， ***在第 184 页***）拆分成数组，留下来的域然后再用 join() 组合成一行。被组合的行存回 clast 与 cline 中。如果没有域要跳过，clast 与 cline 则被设置成 last 与 $0。最后，如果有字符要跳过，substr() 用于清除 clast 与 cline 中前面 count 个的字符。两个字串然后进行比较，are\_equal() 返回比较的结果：

**function are\_equal( n， m， clast， cline， alast， aline)**

**{**

**if (fcount == 0 && charcount == 0)**

**return (last == $0)**

**if (fcount > 0) {**

**n = split(last， alast)**

**m = split($0， aline)**

**clast = join(alast， fcount+1， n)**

**cline = join(aline， fcount+1， m)**

**} else {**

**clast = last**

**cline = $0**

**}**

**if (charcount) {**

**clast = substr(clast， charcount + 1)**

**cline = substr(cline， charcount + 1)**

**}**

**return (clast == cline)**

**}**

下面的两个规则是程序体。第一个只用在数据的第一行。它将 last 设置为 $0，所以，后续的行才有可以比较的东西。

第二个规则执行相应的工作。变量 equal 为 1 或者 0，则依赖于 are\_equal() 的比较结果。如果 uniq 正在统计重复的行，而行是相同的，则其会递增 count 变量。否则，它打印行然后重新设置 count，因为两行已经不相同了。

如果 uniq 不在统计中，而 if 比较结果是相同的，count 则会被递增。不会打印输出行，因为要移除重复的行。否则，如果 uniq 正在统计重复的行，并且有看到了多于一行相同行，或者如果 uniq 在统计非重复的行，则只会见到一行，然后该行会被打印输出，而 count 会被重置。

**NR == 1 {**

**last = $0**

**next**

**}**

**{**

**equal = *are\_equal()***

**if (do\_count) { # overrides -d and -u**

**if (equal)**

**count++**

**else {**

***printf*("%4d %s\n"， count， last) > outputfile**

**last = $0**

**count = 1 # reset**

**}**

**next**

**}**

**if (equal)**

**count++**

**else {**

**if ((repeated\_only && count > 1) ||(non\_repeated\_only && count == 1))**

**print last > outputfile**

**last = $0**

**count = 1**

**}**

**}**

**END {**

**if (do\_count)**

***printf*("%4d %s\n"， count， last) > outputfile**

**else if ((repeated\_only && count > 1) ||(non\_repeated\_only && count == 1))**

**print last > outputfile**

**close(outputfile)**

**}**

### 11.2.7 统计

wc（word count）工具用来统计一个或者多个文件中的行，字与字符。它的用法如下：

**wc [-lwc] [files . . . ]**

如果在命令行中没有指定文件，wc 则从标准输入中读取。如果有多个文件，则它会打印所有文件的总的统计信息。wc 的选项及其意义如下：

-l 只统计行。

-w 只统计单词。一个“单词”是一个连续的非空白字符字符序列，相互之间以空格与/或者 TABs 来分割。幸运的是这也是 awk 对输入数据进行域分割的常规方式。

-c 只统计字符。

awk 中的 wc 实现特定优雅，因为 awk 已经为我们做了很多的事情了，awk 将行分割成单词（即域）然后统计它们，也统计行（即记录），它也可以很容易地告诉我们行的长度是多少。

程序使用到 getopt() 函数（查看 ， ***在第 237 页***）以及文件转换函数（查看 ， ***在第 233 页***）。

这个版本与传统的 wc 版本没有什么特别的不同：它总是按行，单词，字符这样的顺序来打印统计。传统的版本会注意到 -l，-w，-c 选项在命令行中的顺序，并以选项的顺序来打印统计。

BEGIN 规则作参数处理。变量 print\_total 在命令行中有多于一个文件的情况下为 true：

**# wc.*awk* --- count lines， words， characters**

**# Options:**

**# -l only count lines**

**# -w only count words**

**# -c only count characters**

**#**

**# Default is to count lines， words， characters**

**#**

**# Requires *getopt()* and file transition library functions**

**BEGIN {**

**# let *getopt()* print a message about**

**# invalid options. we ignore them**

**while ((c = getopt(ARGC， ARGV， "lwc")) != -1) {**

**if (c == "l")**

**do\_lines = 1**

**else if (c == "w")**

**do\_words = 1**

**else if (c == "c")**

**do\_chars = 1**

**}**

**for (i = 1; i < Optind; i++)**

**ARGV[i] = ""**

**# if no options， do all**

**if (! do\_lines && ! do\_words && ! do\_chars)**

**do\_lines = do\_words = do\_chars = 1**

**print\_total = (ARGC - i > 2)**

**}**

beginfile() 函数很简单，它仅是 lines，words 与 characters 设置为 0，并将当前的文件名存在 fname 中：

**function beginfile(file)**

**{**

**lines = words = chars = 0**

**fname = FILENAME**

**}**

endfile() 函数将当前文件的数值加到总的行，单词与字符数上。然后为刚才所读取的文件打印相应的数值。它依赖于 beginfile() 来重置后面的数据文件的数值：

**function endfile(file)**

**{**

**tlines += lines**

**twords += words**

**tchars += chars**

**if (do\_lines)**

***printf* "\t%d"， lines**

**if (do\_words)**

***printf* "\t%d"， words**

**if (do\_chars)**

***printf* "\t%d"， chars**

***printf* "\t%s\n"， fname**

**}**

有一个对每个行都执行的规则。在里面会对记录的长度加到 chars 上，再加上1。[[82]](#footnote-83)将记录的长度加 1 是因为换行符用来分割记录（即 RS 的值）但它又不是记录的一部份，因此不包含在长度里。后面，对每一行读取时 lines 会递增，而 words 会递增 NF 的值，而它的值就是本条行中“单词”的个数：

**# do per line**

**{**

**chars += length($0) + 1 # get newline**

**lines++**

**words += NF**

**}**

最后，END 规则简单地铺所有文件的总合：

**END {**

**if (print\_total) {**

**if (do\_lines)**

***printf* "\t%d"， tlines**

**if (do\_words)**

***printf* "\t%d"， twords**

**if (do\_chars)**

***printf* "\t%d"， tchars**

**print "\ttotal"**

**}**

**}**

## 11.3 awk 程序的大集合

这一节中是一个不同程序的“大集合”。我们希望你会觉得他们即有趣也很好玩。

### 11.3.1 查找文档中的重复字

在写大量的单调文章的常见错误就是会偶然输入重复的词。一般情况下你会看到似下面的东西“the the program does the following. . .”。当文件是在线的时候，重复的词常常出现在一行的尾部以及另一行的行首，因此很难进行定位。

dupword.awk 这个程序会扫描整个文件，一次一行，并查找同一个字符的的连续出现。它同时也保存一行中的最后一个单词（放在变量 prev 中），用于与下一个单词的第一个单词进行比较。

头两个语句要确保行全是小写的，所以，像 “The” 与 “the” 可以相互进行比较。下一条语句会替换非字母数字与非空格字符为空格，所以标点符号也不会影响比较。用空格替换字符这样格式控制字符不会创建一些没有意义的单词（如 Texinfo 中的 ‘@code{NF}’ 在简单地删除标点的情况下会变成 ‘codeNF’）。记录然后会被分割为域，产生了一行中的实际单词，并且保证不会有空的域。

如果在移除了所有的标点后没有域，则当前的记录会被跳过。否则程序会迭代每一个单词，并且其前一个单词进行比较：

**# dupword.*awk* --- find duplicate words in text**

**{**

**$0 = tolower($0)**

**gsub(/[^[:alnum:][:blank:]]/， "");**

**$0 = $0 # re-split**

**if (NF == 0)**

**next**

**if ($1 == prev)**

***printf*("%s:%d: duplicate %s\n"，FILENAME， FNR， $1)**

**for (i = 2; i <= NF; i++)**

**if ($i == $(i-1))**

***printf*("%s:%d: duplicate %s\n"，FILENAME， FNR， $i)**

**prev = $NF**

**}**

### 11.3.2 报警时钟程序

*没有什么像闹钟一样可以治愈失眠。*

*—Arnold Robbins*

*睡眠是给 Web 开发者的。*

*—Erik Quanstrom*

下面的程序是一个简单的“闹钟”程序。你给它一个时间以及一个可选的消息，在指定的时间上，它会打印消息到标准输出上。另外，你可以给它一个消息重复的资料，以及重复之间的延迟时间。

这个程序使用 getlocaltime() 函数（查看 ， ***在第 229 页***）。

在 BEGIN 规则中完成所有的工作。第一部分是参数检查，并设置默认值：delay，count，以及要打印的 message。如果用户提供了一个不带 ASCII BEL 字符的消息（为 ‘alert’ 字符，‘\a’），然后它会被加到 message 中。（在很多的系统中，打印 ASCII BEL 会产生一个可以听得到的报警块。所以， 当报警消除时，系统会在用户没有看电脑时来唤起其注意。）只是为了做一些改变，这个程序使用 switch 语句（查看 ， ***在第 150 页***），但是处理也可以用一系列的 if-else 语句来完成。下面是程序：

**# alarm.*awk* --- set an alarm**

**#**

**# Requires *getlocaltime()* library function**

**# usage: alarm time [ "message" [ count [ delay ] ] ]**

**BEGIN {**

**# Initial argument sanity checking**

**usage1 = "usage: alarm time [ 'message' [count [delay]]]"**

**usage2 = sprintf("\t(%s) time ::= hh:mm"， ARGV[1])**

**if (ARGC < 2) {**

**print usage1 >"/dev/stderr"**

**print usage2 >"/dev/stderr"**

**exit 1**

**}**

**switch (ARGC) {**

**case 5:**

**delay = ARGV[4] + 0**

**# fall through**

**case 4:**

**count = ARGV[3] + 0**

**# fall through**

**case 3:**

**message = ARGV[2]**

**break**

**default:**

**if (ARGV[1] !~ /[[:digit:]]?[[:digit:]]:[[:digit:]]{2}/) {**

**print usage1 >"/dev/stderr"**

**print usage2 >"/dev/stderr"**

**exit 1**

**}**

**break**

**}**

**# set defaults for once we reach the desired time**

**if (delay == 0)**

**delay = 180 # 3 minutes**

**if (count == 0)**

**count = 5**

**if (message == "")**

**message = sprintf("\aIt is now %s!\a"， ARGV[1])**

**else if (index(message， "\a") == 0)**

**message = "\a" message "\a"**

代码的下一段会将报警时间转换为小时分钟，并转换（如果有必要的话）为一个 24-小时制的时间，然后将时间转换为从午夜以来的秒数。后面它会将当前的时候转换为从午夜以来的秒数。两个时间之间的差就是要设置报警要等待的时间：

**# split up alarm time**

**split(ARGV[1]， atime， ":")**

**hour = atime[1] + 0 # force numeric**

**minute = atime[2] + 0 # force numeric**

**# get current broken down time**

**getlocaltime(now)**

**# if time given is 12-hour hours and it’s after that**

**# hour， e.g.， ‘alarm 5:30’ at 9 a.m. means 5:30 p.m.，**

**# then add 12 to real hour**

**if (hour < 12 && now["hour"] > hour)**

**hour += 12**

**# set target time in seconds since midnight**

**target = (hour \* 60 \* 60) + (minute \* 60)**

**# get current time in seconds since midnight**

**current = (now["hour"] \* 60 \* 60) + \**

**(now["minute"] \* 60) + now["second"]**

**# how long to sleep for**

**naptime = target - current**

**if (naptime <= 0) {**

**print "alarm: time is in the past!">"/dev/stderr"**

**exit 1**

**}**

最后，程序使用 system() 函数（查看 ）来调用 sleep 工具。sleep 工具会暂停给定的秒数。如果其退出时间不是 0，而程序会假设休眠中断并退出。如果 sleep 退出，退出代码为 0，则程序会在循环中打印消息，然后再通过 sleep 来迟延不管多少必要的秒时间：

**# zzzzzz..... go away if interrupted**

**if (system(sprintf("sleep %d"， naptime)) != 0)**

**exit 1**

**# time to notify!**

**command = sprintf("sleep %d"， delay)**

**for (i = 1; i <= count; i++) {**

**print message**

**# if sleep command interrupted， go away**

**if (system(command) != 0)**

**break**

**}**

**exit 0**

**}**

### 11.3.3 转换字符

系统的 tr 工具用来进行字符的一对一转换。例如，用来将大写字母转换为小写字母以便进行进一步的处理：

**generate data | tr ’A-Z’ ’a-z’ | process data ...**

tr 需要两个字符列表。[[83]](#footnote-84)当处理输入时，在第一个字符列表中的第一个字符会被第二个列表中的第一个字符替换，而第一个字符列表中的第二个字符会被第二个字符列表中的第二个字符替换，以此类推。如果在 from 列表中有比 to 列表中更多的字符，则在 to 中的最后的字符用来替换在 from 列表中的其他字符。

以前，有用户建议在 gawk 中加入字符转换函数。下面的程序可以证明用户层的函数也可以用来实现字符的转换功能。这个程序没有 tr 工具那么完整，但是可以做大部份的事情。

转换程序是在 gawk 取得对每个字串中的字符进行分割到数组中的能力很久之前写的。所以，它重复地使用 substr()，index()，与 gsub() 等内置函数（查看 ， ***在第 184 页***）。下面有两个函数，第一个函数 stranslate()，有三个参数：

from 要转换的字符列表。

to 要转换成的字符列表。

target 要执行转换的字串。

关联数组使得转换部分非常容易。t\_ar 保存 to 的字符，并用 from 的字符作为索引。然后一个简单的循环来循环 from 字符，一次一个字符。对在 from 中的每个字符，如果字符出现在 target 中，则它被替换为 to 中的相应字符。

translate() 函数调用 stranslate()，并将 $0 作为 target 参数。主程序用来自由命令行的值设置两个全局变量，FROM 与 TO，然后将 ARGV 更改为 awk 从标准输入中读取。

最后，处理规则只是简单地调用 translate() 来处理每一条记录：

**# translate.*awk* --- do tr-like stuff**

**# Bugs: does not handle things like tr A-Z a-z; it has**

**# to be spelled out. However， if ‘to’ is shorter than ‘from’，**

**# the last character in ‘to’ is used for the rest of ‘from’.**

**function stranslate(from， to， target， lf， lt， ltarget， t\_ar， i， c，**

**result)**

**{**

**lf = length(from)**

**lt = length(to)**

**ltarget = length(target)**

**for (i = 1; i <= lt; i++)**

**t\_ar[substr(from， i， 1)] = substr(to， i， 1)**

**if (lt < lf)**

**for (; i <= lf; i++)**

**t\_ar[substr(from， i， 1)] = substr(to， lt， 1)**

**for (i = 1; i <= ltarget; i++) {**

**c = substr(target， i， 1)**

**if (c in t\_ar)**

**c = t\_ar[c]**

**result = result c**

**}**

**return result**

**}**

**function translate(from， to)**

**{**

**return $0 = stranslate(from， to， $0)**

**}**

**# main program**

**BEGIN {**

**if (ARGC < 3) {**

**print "usage: translate from to">"/dev/stderr"**

**exit**

**}**

**FROM = ARGV[1]**

**TO = ARGV[2]**

**ARGC = 2**

**ARGV[1] = "-"**

**}**

**{**

**translate(FROM， TO)**

**print**

**}**

有可能在用户层的函数中来执行字符转换，但是没有这么高效，我们（即 gawk 的开发者）开始考虑添加一个内置的函数。但是，在完成这个程序之后不久，我们了解到 Brian Kernighan 已经加入了 toupper() 与 tolower() 函数到它的 gawk 中（查看 ， ***在第 184 页***）。这些函数处理处理了大量的所需要的字符转换的主要部分，因此我们只好选择来简单地将这些函数加到 gawk，并保证它们足够独立。

对这个程序的明显提升就是在 BEGIN 规则中仅设置 t\_ar 数组一次。但是，这假设 from 与 to 的列表在程序的生命期内不会改变。

另一个明显的改进，就是允许使用 tr 工具所使用的范围，如 ‘a-z’。查看 cut.awk （查看 ， ***在第 254 页***）代码来查看其中的灵感。

### 11.3.4 打印邮件标题

这里是一个“真实世界”[[84]](#footnote-85)中的程序。这个脚本读取名字与地址列表，然后产生邮件标签。标签的每一页有 20 个标签在里头，2 行 10 列。地址保证多于 5 行。每一个地方都通过一个空行与下一个地方进行分隔。

基本的想法是读取 20 个标签的数据。每个标签的每一行都存在 line 数组中。一个单独的规则来处理填充 line 数组，并在读到 20 个标签的时候打印页。

BEGIN 规则简单地将 RS 设置成空串，因此 awk 会将记录以空行分割（查看 ， ***在第 63 页***）。然后将 MAXLINES 设置为 100，因此 100 是在页中的最大行数（20 · 5 = 100）。

大多数的工作是在 printpage() 函数中完成的。标签行被顺序地存在 line 数组中。但是他们要水平地打印出来：line[1] 后面是 line[6]，line[2] 与 line[7] 相邻，等等。用两个循环来完成这个功能。外层的循环通过 i 来控制，用来迭代数据的每 10 行，这是每一行的标签。内循环，通过 j 控制，不用迭代一行中的内容。当 j 从 0 迭代到 4，则 ‘i+j’ 就是行中的第 j 行。‘i+j+5’ 则为下一行。输出最终输出可能如下：

**line 1 line 6**

**line 2 line 7**

**line 3 line 8**

**line 4 line 9**

**line 5 line 10**

**...**

printf 的 ‘%-41s’ 字串表示是左对齐数据，并在固定的宽度域上进行打印。

作为最后提示，在行 21 与行 61 中会有一个空行输出，来保持输出行是标签是对齐的。这依赖于写程序时所使用的特别的标签类型。你也需要注意，在开头与结尾都有两个空行。

END 规则用来将最后页的标签输出，因为有可能没有多个整 20 个标签存在：

**# labels.*awk* --- print mailing labels**

**# Each label is 5 lines of data that may have blank lines.**

**# The label sheets have 2 blank lines at the top and 2 at**

**# the bottom.**

**BEGIN { RS = "" ; MAXLINES = 100 }**

**function printpage( i， j)**

**{**

**if (Nlines <= 0)**

**return**

***printf* "\n\n" # header**

**for (i = 1; i <= Nlines; i += 10) {**

**if (i == 21 || i == 61)**

**print ""**

**for (j = 0; j < 5; j++) {**

**if (i + j > MAXLINES)**

**break**

***printf* " %-41s %s\n"， line[i+j]， line[i+j+5]**

**}**

**print ""**

**}**

***printf* "\n\n" # footer**

**delete line**

**}**

**# main rule**

**{**

**if (Count >= 20) {**

***printpage()***

**Count = 0**

**Nlines = 0**

**}**

**n = split($0， a， "\n")**

**for (i = 1; i <= n; i++)**

**line[++Nlines] = a[i]**

**for (; i <= 5; i++)**

**line[++Nlines] = ""**

**Count++**

**}**

**END {**

***printpage()***

**}**

### 11.3.5 生成字使用统计

当要处理大量的文本的时候，如果了解单词的使用频率是一件很有趣的事情。例如，作者可能会容易使用特定的词，在这种情况下，他或者她可能想找一些同义词来代替那些出现得太频繁的词。这一节中开发了一个这样的程序来统计单词以及它出现的频率，并以一种有用的方式输出这些信息。

乍看之下，像下面这样的程序可以完成这样的功能：

**# wordfreq-first-try.*awk* --- print list of word frequencies**

**{**

**for (i = 1; i <= NF; i++)**

**freq[$i]++**

**}**

**END {**

**for (word in freq)**

***printf* "%s\t%d\n"， word， freq[word]**

**}**

程序依赖于 awk 的默认域分割机制来将每一行分割成词并使用关联数组 freq 来存储它们，用单词来作为索引，来统计单词出现的次数，在 END 规则中来打印统计。

这个程序有几个问题会导致其处理真实文本时有可用程度：

* awk 语言会把大写与小定是有区别的。所以，“bartender” 与 “Bartender” 会当成不一样的词。这不是想要的结果，因为单词在正常的文本中的句首，会进行首字大写，而频率分析器却对首字大写不敏感。
* 单词的检测使用的是 awk 默认使用空格进行域分割的惯例。在输入中的其他字符（不包含换行符）对于 awk 来说没有特殊的含义。这表示标点符号也会成为单词的一部分。
* 输出不会有什么有用的顺序。你更有可能对最常出现的单词感兴趣，或者形成一个字母表来表示每个单词的出现频率。

第一个问题可以通过 tolower() 来移除大小定来解决。第三个问题可以用 gsub() 移除标点符号来解决。最后，我们通过使用系统的 sort 工具处理 awk 脚本的输出来解决第三个问题。这是一个新版本的程序：

**# wordfreq.*awk* --- print list of word frequencies**

**{**

**$0 = tolower($0) # remove case distinctions**

**# remove punctuation**

**gsub(/[^[:alnum:]\_[:blank:]]/， ""， $0)**

**for (i = 1; i <= NF; i++)**

**freq[$i]++**

**}**

**END {**

**for (word in freq)**

***printf* "%s\t%d\n"， word， freq[word]**

**}**

正则表达式 /[^[:alnum:]\_[:blank:]]/ 本可以写在 /[[:punct:]]/，但是下划线也要移除，但我们想保留它。

假设我们把这个程序保留为名字为 wordfreq.awk 的文件，数据在 file1 中，则下面的管程：

awk -f wordfreq.awk file1 | sort -k 2nr

会产生出现在 file1 中的单词频率的降序表。awk 程序适合于处理信息以及数据来产生单词频率表，但是不是有序的。awk 的脚本输出然后通过 sort 工具来排序，然后再打印到屏幕上。

提交给 sort 工具的选择指定 sort 使用每个输入行的第二个域（跳过第一个域），排序键被当成数值量来对待（不然 ‘15’ 会排在 ‘5’ 之前），然后排序会以降序（逆序）的方式来执行。

排序甚至可以在程序中完成，只要把 END 的动作改成如下：

**END {**

**sort = "sort -k 2nr"**

**for (word in freq)**

***printf* "%s\t%d\n"， word， freq[word] | sort**

**close(sort)**

**}**

这种方式的排序必须使用在那些在命令行一层没有真正的管道的的系统上（或者批处理文件）。查看操作系统上如何使用 sort 程序的文档来获取更多信息。

### 11.3.6 从未排序的文本中去重

uniq 程序（查看 ， ***在第 267 页***）在经过排序的数据中移除重复的行。

但是，你可能需要移除数据文件中的重复的行，而又想保留行中的顺序。这样处理的一个合适的例子就是 Shell 的 histroy 文件。history 文件保留了你已经输入过的命令的复本，在一行中重复几次相同的命令的情况并不多见。有时候，你可能想移除重复的行来压缩文件。但是还要保留最初命令行的顺序。

下面这个简单的程序可以完成这个工作。它会使用两个数组。 data 数组由每行的文本来作索引。对每一行，data[$0] 会进行递增。如果某个特定的行之前没有出现过，则 data[$0] 会是 0。在这种情况下，那一行的文本会存在 lines[count] 中。lines 中的元素都是唯一的，而 lines 的索引表示了行出现的顺序。END 规则中介简单地按顺序打印 lines：

**# histsort.*awk* --- compact a shell history file**

**# Thanks to Byron Rakitzis for the general idea**

**{**

**if (data[$0]++ == 0)**

**lines[++count] = $0**

**}**

**END {**

**for (i = 1; i <= count; i++)**

**print lines[i]**

**}**

这个程序也提供了用于产生有用信息的一个基础。如，在 END 规则中使用下面的 print 语句表示特定命令使用的频度：

**print data[lines[i]]， lines[i]**

这么做有效是因为 data[$0] 在行每出现一次都会进行递增。

### 11.3.7 从 Texinfo 源文件中提取程序

在本章与前一章（）中，都展示了大量的 awk 程序。如果你想实践这些程序，如果再次手工输出这些程序则就太繁琐了。这里，我们展示了一个程序可以从 Texinfo 的输入文件中提出部分到一个独立的文件中。

本书是用 Texinfo 写成的，这是 GNU 项目的文档格式语言。一个单独的 Texinfo 源文件可以用来产生打印的 TEX 文档，也可以产生在线文档。（Texinfo 在书 Texinfo—*The GNU Documentation Format* 有完整的文档，也可从自由软件基金会取得，也可以在网上取得。）

对于我们自己的目的，知道 Texinfo 输入文件的三件事情就够了：

* ‘at’ 符号（‘@’）在 Texinfo 中是特殊符号，就像是在 C 或者 awk 中的反斜杠一样（‘\’）。字面的 ‘@’ 在 Texinfo 源文件中由 ‘@@’ 来表示。
* 注释同 ‘@c’ 开始，或者由 ‘@comment’ 开始。文件提出程序通过使用在行首开始的特殊的注释。
* 包含 ‘@group’ 与 ‘@end group’ 命令包含例子文本不可以分页。（不幸的是，TEX 不总是足够来做正确的事情，因此我们还得给它一些帮助。）

下面的程序 extract.awk 从 Texinfo 源文件中读取，并根据特殊注释做两件事情。当看到 ‘@c system...’ 时，它执行一个命令，并从命令文本中提取控制行并将其传递给 system 函数（查看 ， ***在第 194 页***）。当看到 ‘@c filename’ 时，其后续的行将被发送到文件 filename 中，直到看到 ‘@c endfile’ 为止。在 extract.awk 规则中，要么匹配 ‘@c’ 要么匹配 ‘@comment’，只要将 ‘omment’ 成为可选的部分即可。包含 ‘***@group***’ 与 ‘***@end*** group’ 的行，只是简单地移除。extract.awk 使用到 join() 库函数（查看 ， ***在第 228 页***）。

在 *GAWK: Effective AWK Programming* 的 Texinfo 的源代码中的例子程序赵向都已经包含在 ‘file’ 与 ‘endfile’ 中的行里。gawk 发行版本使用 extract.awk 的一个复本来提取例子程序，并将他们中的多数安装到了标准的 gawk 可查找到目录中。Texinfo 文件看起来这样：

**...**

**This program has a @code{BEGIN} rule**

**that prints a nice message:**

**@example**

**@c file examples/messages.*awk***

**BEGIN @{ print "Don’t panic!" @}**

**@c end file**

**@end example**

**It also prints some final advice:**

**@example**

**@c file examples/messages.*awk***

**END @{ print "Always avoid bored archaeologists!" @}**

**@c end file**

**@end example**

**...**

extract.awk 将 IGNORECASE 设置为 1，所以大小写混合在指示符中不会有什么影响。

第一个规则处理 system() 调用，检查要执行的命令（NF 至少为 3）并且也检查命令出代码为是否为0，即表示执行 OK：

**# extract.*awk* --- extract files and run programs from Texinfo files**

**BEGIN { IGNORECASE = 1 }**

**/^@c(omment)?[ \t]+system/ {**

**if (NF < 3) {**

**e = ("extract: " FILENAME ":" FNR)**

**e = (e ": badly formed ‘system’ line")**

**print e >"/dev/stderr"**

**next**

**}**

**$1 = ""**

**$2 = ""**

**stat = system($0)**

**if (stat != 0) {**

**e = ("extract: " FILENAME ":" FNR)**

**e = (e ": warning: system returned " stat)**

**print e >"/dev/stderr"**

**}**

**}**

变量 e 用来使规则很好地匹配页。

第二个规则将数据移动到文件中。它会检查在指示符中给出的文件名。如果文件名不是当前的文件，然后当前文件就会被关闭。让当前的文件一直打开到处理下一个新文件，这样你可以使用 ‘>’ 来重定向打印内容，打开文件的管理也很简单。

for 循环处理真正的工作。它使用 getline（查看 ， ***在第 83 页***）来读取行。对于一个未期望的文件尾，它会调用 unexpected\_eof() 函数。如果行是一个 ‘endfile’ 行，则它就跳出循环。如果行是 ‘@group’ 或者 ‘***@end*** group’ 行，则它会忽略掉，然后继续处理下一行。类似地，例子中的注释也被忽略掉。

大多数的工作都在下面几行。如果行中没有 ‘@’ 符号，则程序可以直接打印它，否则每一个前导的 ‘@’ 必须被去掉。为了移除 ‘@’ 符号，使用 split() 函数（查看 ， ***在第 184 页***）行被分割成 a 数组的不同元素。‘@’ 符号被用于分隔符。a 的每一个空元素表示在原行中的两个连续的 ‘@@’ 符号。对每两个空元素（在原文件中的‘@@’），我们需要加一个 @ 符号进来。

当数组的处理已经完成，join() 则被以 SUBSEP 的值调用（查看 ， ***在第 176 页***），用以将分散的块连接成一个单独的行。该行然后被打印到输出文件中：

**/^@c(omment)?[ \t]+file/ {**

**if (NF != 3) {**

**e = ("extract: " FILENAME ":" FNR ": badly formed ‘file’ line")**

**print e >"/dev/stderr"**

**next**

**}**

**if ($3 != curfile) {**

**if (curfile != "")**

**close(curfile)**

**curfile = $3**

**}**

**for (;;) {**

**if ((*getline* line) <= 0)**

**unexpected\_*eof()***

**if (line ~ /^@c(omment)?[ \t]+endfile/)**

**break**

**else if (line ~ /^@(end[ \t]+)?group/)**

**continue**

**else if (line ~ /^@c(omment+)?[ \t]+/)**

**continue**

**if (index(line， "@") == 0) {**

**print line > curfile**

**continue**

**}**

**n = split(line， a， "@")**

**# if a[1] == ""， means leading @，**

**# don’t add one back in.**

**for (i = 2; i <= n; i++) {**

**if (a[i] == "") { # was an @@**

**a[i] = "@"**

**if (a[i+1] == "")**

**i++**

**}**

**}**

**print join(a， 1， n， SUBSEP) > curfile**

**}**

**}**

这里要注意一个重要的事情就是 ‘>’ 重定向符的使用。用 ‘>’ 输出只会打开一次文件，然后会一保持打开，后续的输出操作会附加到文件中（查看 ， ***在第 103 页***）。这使得很容易混合程序文本与针对相同源文件的解释性的文字（正如这里所做的一样）而没有任何问题。文件只在处理一个文件数据文件的时候才会被关闭，或者在输入文件输入结束时关闭。

最后，***unexpected\_eof()*** 函数打印合适的错误信息并退出。END 规则处理最后的清理工作，关闭打开的文件：

**function unexpected\_*eof()***

**{**

***printf*("extract: %s:%d: unexpected EOF or error\n"，**

**FILENAME， FNR) >"/dev/stderr"**

**exit 1**

**}**

**END {**

**if (curfile)**

**close(curfile)**

**}**

### 11.3.8 一个简单的流编辑器

sed 是一个流编辑器，它是一个读取数据流，改变它然后传递它的程序。通过它被用来对一个大的文件或者由命令的管程产生的数据流做一些全局性的更改。尽管 sed 在它那边来说是一个很复杂的程序，但是它的通常用法是在管线的中途做一些全局性的替换：

**command1 < orig.data | sed ’s/old/new/g’ | command2 > result**

如 ‘s/old/new/g’ 告诉 sed 在每一个输入行查找正则表达式 ‘old’，并全局性地替换它为 ‘new’（即在一行中所有出现）。这个与 awk 的 gsub() 函数（查看 ， ***在第 184 页***）非常类似。

下面的程序，awksed.awk 接收至少两个命令行参数：一个是要查找的模式，以及替换它的文本。任何其他的参数都当成是要处理的文件名。如果没有提供文件名，则使用标准输入：

**# awksed.*awk* --- do s/foo/bar/g using just print**

**# Thanks to Michael Brennan for the idea**

**function *usage()***

**{**

**print "usage: awksed pat repl [files...]">"/dev/stderr"**

**exit 1**

**}**

**BEGIN {**

**# validate arguments**

**if (ARGC < 3)**

***usage()***

**RS = ARGV[1]**

**ORS = ARGV[2]**

**# don’t use arguments as files**

**ARGV[1] = ARGV[2] = ""**

**}**

**# look ma， no hands!**

**{**

**if (RT == "")**

***printf* "%s"， $0**

**else**

**print**

**}**

程序依赖于 gawk 可以将 RS 赋值为正则表达式的能力，以及设置 RT 为实际的结束记录的文本（查看 ， ***在第 63 页***）。

思路就是让 RS 成为要查找的模式。gawk 会处置设置 $0 为在匹配模式之间的文本。这就是我们想要保留的，不经修改的文本。然后，设置 ORS 为替换的文件。一个简单的 print 语句输出我们想要保留的文本，后面跟着要替换的文本。

这个方法有一个不完美的地方，即如果最后一条记录不是以匹配 RS 的文本结束该怎么办。使用 print 语句无条件地打印替换文本是不正确的。但是，如果文件不是以匹配 RS 的文本结束，RT 会被设置成空串。在这种情况下，我们可以使用 printf 来打印 $0。（查看 ， ***在第 97 页***）

BEGIN 规则处理环境的设置，检查参数的正确数值，并在出问题时调用 usage() 函数。然后再设置 RS 与 ORS 为从命令行中取得的参数，并调协 ARGV[1] 与 ARGV[2] 为空串，这样它们就不会被当成文件名来处理（查看 ， ***在第 163 页***）。

usage() 函数会打印一个错误信息并退出。最后，一个简单的规则根据 RT 的值来相应地使用 print 与 printf 来打印之前确定好的方案。

### 11.3.9 使用库函数的简单方式

在 ，***在第 46 页***，中我们看到了 gawk 如何来提供内置的文件包含能力。但是，这是 gawk 的扩展。这一节提供了让标准的 awk 有文件包含能力的动机，以及如何通过 Shell 与 awk 的编程组合来实现。

使用在 awk 中的库函数用处非常大。它鼓励代码重用来写一些通用函数。程序看起来会更小更清晰。但是，写 awk 程序时使用库函数非常容易，但是运行他们时却很痛苦，因为需要多个 -f 选项。如果 gawk 不可用，则通过 AWKPATH 环境来变量来将 awk 函数放到库目录的能力也一样痛苦（查看 ， ***在第 35 页***）。如果能以下面的方式来写程序则就非常好了：

**# library functions**

**@include getopt.*awk***

**@include join.*awk***

**...**

**# main program**

**BEGIN {**

**while ((c = getopt(ARGC， ARGV， "a:b:cde")) != -1)**

**...**

**...**

**}**

下面的程序 igawk.sh 提供了这样的服务。它模拟 gawk 的搜索 AWKPATH 变量的行为，并允许嵌套的 include（即被@include 包含的文件可以再包含 @include 语句）。igawk 努力使文件只被包含一次，因此嵌套的包含不会偶然包含一个库函数两次。

igawk 应该外在表现得像 gawk。这就是说，它可以接收所有的 gawk 的命令行参数，包含通过 -f 指定多个源文件的能力以及能够混用命令行与库源文件的能力。

程序使用 POSIX Shell（sh）命令语言写成。[[85]](#footnote-86)它工作方式如下：

1. 当程序展开执行时，循环迭代参数，保存所有不代表 awk 源代码的参数以备用。
2. 对于任何不代表 awk 文件的参数，将参数放到一个可展开的 Shell 变量中。有两种情形：
   1. 对于由 -e 或者 --source 提供的字面文本。这个文本只是直接进行附加。
   2. 由 -f 参数提供的源文件名。我们使用一个小技巧，并附加 ‘***@include*** filename’ 到 Shell 变量的内容中。由于文件包含程序如 gawk 一样的工作方式，这可以在正确的点取得程序中包含的文件的文本。
3. 在 Shell 变量之上，执行 awk 程序（自然地）来展开 @include 语句。被展开的程序放放在第二个 Shell 变量中。
4. 用 gawk 运行展开的程序以及其他的用户提供的命令参数（例如数据文件名）。

这个程序扩展地使用 Shell 变量：用于存储命令行参数以及将会展开用户程序的 awk 程序的文本，用于用户的初始程序以及展开的程序。这么做可以移除一些当我们使用临时文件而导致的潜在的问题，但是代价就是使脚本带来一定的复杂性。

程序的开始部分，如果第一个参数是 ‘debug’，则打开 Shell 的跟踪。

下一个部分会循环所有的命令行参数。有几个情况非常有意思：

-- 这由于结束 igawk 的参数。其他的任意参数应该被原样地传递到用户的 awk 程序。

-W 这表示下一个选项是特定于 gawk 的。为了使参数处理更加容易，-W 被附加到剩下的参数的前面，然后继续循环。（这是一个 Sh 的编程技巧。如果你不熟悉 Sh 也不用担心。）

-v， -F 这会被存储并被传递给 gawk。

-f， --file， --file=， -Wfile=

使用 @include 语句附加到 Shell 变量程序的文件。expr 工具用于移除参数前面的的选项部分（比如 ‘--file=’）。（典型的 Sh 用法应该是使用 echo 以及sed 工具来做这样的事情。很不幸的是，有一些版本的 echo 会对它的参数列表中的转义序列进行求值，有可能会弄乱程序文本。使用 expr 可以避免这个问题。）

--source， --source=， -Wsource=

源代码被附加到程序中。

--version， -Wversion

igawk 打印它的版本号，运行 ‘gawk --version’ 来取得 gawk 的版本信息，然后退出。

如果 -f --file，-Wfile，--source 或者 -Wsource 参数一个都没有提供，则第一个非选项参数会用作 awk 的程序。如果没有其他的命令行参数，igawk 打印一个错误信息后退出。否则，第一个参数会被附加到程序。在任何情况下，在参数被处理后， Shell 变量程序包含最初 awk 程序的完全的文本。程序如下：

**#! /bin/sh**

**# igawk --- like gawk but do @include processing**

**if [ "$1" = debug ]**

**then**

**set -x**

**shift**

**fi**

**# A literal newline， so that program text is formatted correctly**

**n=’**

**’**

**# Initialize variables to empty**

**program=**

**opts=**

**while [ $# -ne 0 ] # loop over arguments**

**do**

**case $1 in**

**--) shift**

**break ;;**

**-W) shift**

**# The ${x?’message here’} construct prints a**

**# diagnostic if $x is the null string**

**set -- -W"${@?’missing operand’}"**

**continue ;;**

**-[vF]) opts="$opts $1 ’${2?’missing operand '}'"**

**shift ;;**

**-[vF]\*) opts="$opts ’$1’" ;;**

**-f) program="$program$n@include ${2?’missing operand’}"**

**shift ;;**

**-f\*) f=$(expr "$1" : ’-f\(.\*\)’)**

**program="$program$n@include $f" ;;**

**-[W-]file=\*)**

**f=$(expr "$1" : ’-.file=\(.\*\)’)**

**program="$program$n@include $f" ;;**

**-[W-]file)**

**program="$program$n@include ${2?’missing operand’}"**

**shift ;;**

**-[W-]source=\*)**

**t=$(expr "$1" : ’-.source=\(.\*\)’)**

**program="$program$n$t" ;;**

**-[W-]source)**

**program="$program$n${2?’missing operand’}"**

**shift ;;**

**-[W-]version)**

**echo igawk: version 3.0 1>&2**

**gawk --version**

**exit 0 ;;**

**-[W-]\*) opts="$opts ’$1’" ;;**

**\*) break ;;**

**esac**

**shift**

**done**

**if [ -z "$program" ]**

**then**

**program=${1?’missing program’}**

**shift**

**fi**

**# At this point， ‘program’ has the program.**

处理 @include 指令的 awk 程序被存储在 Shell 的变量 expand\_prog 中。这么做可以使得 Shell 脚本更可读。awk 程序使用 getline（查看 ， ***在第 83 页***）读取用户的程序，一次一行。输入文件名以及 @include 语句使用栈来管理。每遇到一个 @include，当前的文件名会被推到栈中，在 @include 中的文件名则成为当前文件名。每完成一个文件，则会对栈进行弹出，这样前一个文件会成为当前文件。处理是以最开始的文件作为栈中的第一个文件开始的。

pathto() 函数处理查找文件全路径的事情。它模拟 gawk 的查找 AWKPATH（查看 ， ***在第 43 页***） 环境变量的行为。如果文件名中有 ‘/’，则不会进行路径搜索。类似地，如果文件名是 “-”，则相应的字串则原样不变。否则，文件名被与路径中的每一个上进进行连接，并尝试打开生成的文件。唯一的测试文件是否可在 awk 中进行读取的方式是继续通过 getline 来尝试读取，这正是 pathto() 所做的。[[86]](#footnote-87)如果文件可读，则关闭它，把返回文件名：

**expand\_prog=’**

**function pathto(file， i， t， junk)**

**{**

**if (index(file， "/") != 0)**

**return file**

**if (file == "-")**

**return file**

**for (i = 1; i <= ndirs; i++) {**

**t = (pathlist[i] "/" file)**

**if ((*getline* junk < t) > 0) {**

**# found it**

**close(t)**

**return t**

**}**

**}**

**return ""**

**}**

主程序包含在 BEGIN 规则中。它所做的第一件事情就是设置一个 pathlist 数组给 pathto() 使用。在使用 ‘:’ 分割路径之后，空的元素用 “.” 进行替换，它表示的是当前路径：

**BEGIN {**

**path = ENVIRON["AWKPATH"]**

**ndirs = split(path， pathlist， ":")**

**for (i = 1; i <= ndirs; i++) {**

**if (pathlist[i] == "")**

**pathlist[i] = "."**

**}**

栈使用 ARGV[1] 来进行初始化，它的值为 “/dev/stdin”。后面则是主循环。输入行会一行行连续。不能 @include 开始的行则会原样打印。如果行确实是以 @include 开始的，则文件名会存储在 $2 中。pathto() 用来产生完整的路径。如果没法产生，程序则打印一个错误信息后继续。

下一个要做的事情就是检查文件是否已经被包含。已经处理的数组用已经包含的文件的全名来进行索引，它会为我们跟踪相应的信息。如果文件再次看到，会打印一个警告信息。否则新的文件名会被推到栈中并继续处理。

最后，当 getline 遇到了输入的文件尾，文件则被关闭，栈弹出。当 stackptr 小于 0 时，程序完成：

**stackptr = 0**

**input[stackptr] = ARGV[1] # ARGV[1] is first file**

**for (; stackptr >= 0; stackptr--) {**

**while ((*getline*< input[stackptr]) > 0) {**

**if (tolower($1) != "@include") {**

**print**

**continue**

**}**

**fpath = pathto($2)**

**if (fpath == "") {**

***printf*("igawk: %s:%d: cannot find %s\n"，**

**input[stackptr]， FNR， $2) >"/dev/stderr"**

**continue**

**}**

**if (! (fpath in processed)) {**

**processed[fpath] = input[stackptr]**

**input[++stackptr] = fpath # push onto stack**

**} else**

**print $2， "included in"， input[stackptr]，**

**"already included in"，**

**processed[fpath] >"/dev/stderr"**

**}**

**close(input[stackptr])**

**}**

**}’ # close quote ends ‘expand\_prog’ variable**

**processed\_program=$(gawk -- "$expand\_prog" /dev/stdin << EOF**

**$program**

**EOF**

**)**

Shell 的 ‘command << marker’ 结构称为就地文档。在 Shell 脚本中的到这个标志的所有内容都被当成是 command 的输入。Shell 处理就地文档的内容用于变量与命令的替换（也可以是其他的事情，这要看 Shell）。

Shell 的 ‘$(...)’ 这样的结构称为命令替换。在括号中的命令的输出会替换命令行。因为结果会在变量赋值中使用，它会被存成一个单独的字串，就算结果包含空格。

展开的程序被保存在变量 processed\_programe 中。它通过下面几步来完成：

* 1. 与从标准输入读取的 @include 程序一起执行 gawk （即 expand\_prog Shell 变量的值）。
  2. 标准输入是用户程序的内容，来自于 Shell 变量程序。将它的内容通过就地文档的方式灌入 gawk 中。
  3. 将这个处理的结果使用命令替换存到 Shell 变量 processed\_program 中。

最后一步使用展开的程序以及用户提供的最初的选项以及命令行参数调用 gawk：

**eval gawk $opts -- ’"$processed\_program"’’"$@"’**

eval 命令是一个 Shell 结果用来重新执行 Shell 的分析过程。这使得内容会被恰当地引起来。

igawk 的这个版本呈现了这个程序的第五个版本。有4 个关键的简化使得程序运行得更好：

* 使用 @include 使得最开始的集合 awk 程序更简单，即使是使用 -f 指定的文件名，所有的 @include 处理都可以一次完成。
* 当测试文件的可访问性，以在主程序中使用时，不会尝试存储在 pathto() 函数中使用 getline 读取的行。这使得事情得到相当的简化。
* 在 BEGIN 规则中使用 getline 在一个地方就完成所有的工作。没有必要再调用另外一个循环来处理嵌套的 @include 语句。
* 不将展开的程序存在一个临时文件中，而是将它们放在一个 Shell 变量中可以避免一些潜在的安全问题。但这也有缺点，因为脚本要依赖于 Sh 语言的更多特性，使得那些不熟悉 Sh 的人很验证于跟踪。

同时 ，这个程序也说明了将 sh 与 awk 程序混合使用是值得一试的。你可以通过这样来完成很多的事情，不用寻求更加底层的 C 或者 C++ 语言，通常在 Shell 中要比 awk 也更容易来处理特定类型的字串与参数操作。

最后 igawk 展示了给程序添加新的特性并不总是必要的，它们可以一层层叠加来实现。[[87]](#footnote-88)

### 11.3.10 从字典中查找单词

一个有趣的编程挑战就是在一个单词列表中查找单词（如在很多的 GNU/Linux 系统中的/usr/share/dict/words）。如果两个单词包含相同的字母，则一个单词是另一个单词的字谜（如 “babbling” 与 “blabbing”）。

Jon Bentley 所著的书 *Programming Pearls， Second Edition*，第二卷的问题 C，提出了一个优雅的算法。其思想是给单词的字谜一个共同的简单，并通过这些签名来对单词进行排序，然后打印它们。Bentley 博士发现发现用每个单词中出现的字母，并对它们排序来作为单词的签名.

下面的程序使用数组之数组来将有相同签名的单词组织在一起，以及对数组排序来以有序的方式打印单词：

**# anagram.*awk* --- An implementation of the anagram-finding algorithm**

**# from Jon Bentley’s "Programming Pearls，" 2nd edition.**

**# Addison Wesley， 2000， ISBN 0-201-65788-0.**

**# Column 2， Problem C， section 2.8， pp 18-20.**

**/’s$/ { next } # Skip possessives**

程序以一个头部开始，然后跳过在字典文件中的所有格。下一个规则则是用来建立数组结构。数组的第一维由签名来进行索引，第二维则是单词本身：

**{**

**key = word2key($1) # Build signature**

**data[key][$1] = $1 # Store word with signature**

**}**

***word2key()*** 用来创建签名，它将单词分成一个个的字符，并对字符进行排序，然后再将它全合并：

**# word2key --- split word apart into letters， sort， and join back together**

**function word2key(word， a， i， n， result)**

**{**

**n = split(word， a， "")**

**asort(a)**

**for (i = 1; i <= n; i++)**

**result = result a[i]**

**return result**

**}**

最后，END 规则用来遍历数组，然后打出出字谜列表。它会将输出系统的 sort 工具，因为不这么处理，字谜就会以随机的顺序出现：

**END {**

**sort = "sort"**

**for (key in data) {**

**# Sort words with same key**

**nwords = asorti(data[key]， words)**

**if (nwords == 1)**

**continue**

**# And print. Minor glitch: trailing space at end of each line**

**for (j = 1; j <= nwords; j++)**

***printf*("%s "， words[j]) | sort**

**print "" | sort**

**}**

**close(sort)**

**}**

程序运行时，下面是部分输出：

**$ gawk -f anagram.*awk* /usr/share/dict/words | grep ’^b’**

**...**

**babbled blabbed**

**babbler blabber brabble**

**babblers blabbers brabbles**

**babbling blabbing**

**babbly blabby**

**babel bable**

**babels beslab**

**babery yabber**

**...**

### 11.3.11 来些新鲜的

下面的程序由 Davide Brini 编写并发布在了它的网站上。它用来作为他在 Usenet 小组 comp.lang.***awk 的签名。它提供了下面的版本声明：***

*无论是否修改都允许在任何介质上，复制发布本页中公开的代码，而不需要提供正式的版本声明，只要保留本提示即可。*

这里它的程序：

**awk’BEGIN{O="~"~"~";o="=="=="==";o+=+o;x=O""O;while(X++<=x+o+o)c=c"%c";**

***printf* c，(x-O)\*(x-O)，x\*(x-o)-o，x\*(x-O)+x-O-o，+x\*(x-O)-x+o，X\*(o\*o+O)+x-O，**

**X\*(X-x)-o\*o，(x+X)\*o\*o+o，x\*(X-x)-O-O，x-O+(O+o+X+x)\*(o+O)，X\*X-X\*(x-O)-x+O，**

**O+X\*(o\*(o+O)+O)，+x+O+X\*o，x\*(x-o)，(o+X+x)\*o\*o-(x-O-O)，O+(X-x)\*(X+O)，x-O}’**

程序做的事情由你来确定。（如果你真的在理解它时绝望了，查看 Chris Johansen 的解释，该解释内嵌在本书的 Texinfo 源文件中。

## 11.4 总结

* 本章中提供的程序，延续了这样的主旨，就是要学好编程的终极方法就是去阅读程序。
* 使用 ‘#!’ 来使用 awk 程序可以直接运行而更好使用。否则就得这么来调用程序：‘***awk*** -f ...’。
* 用 awk 重新实现标准的 POSIX 程序是一个令人愉快的练习，awk 的表达的威力你以让你用很少的代码就来写出这样的程序，同时功能完备并且可用。
* awk 的一个弱点之一就是在处理单个字符的时候。使用 split() 以及空串来作为分隔符字串的能力可以极大地简化这个工作。
* 这里的例子表明了来自由第 10 章 ，***在第 221 页***，中的库函数对于很多的（小）程序的用处。
* 除了重新发明 POSIX 轮子，其他的程序可以解释很多的有趣的问题，如查找文本中的重复单词，打印邮件标签，或者查找字谜等。

## 11.5练习

1. 使用 “” 作为分隔符来调用 split() 来重写 cut.awk（查看 ， ***在第 254 页***）。
2. 在 ，***在第 258 页***，中我们提到过 ‘egrep -i’ 可以使用 tolower() 用于行与模式来模拟 awk 中没有 IGNORECASE 版本的情况。在那个里的一个脚注中，我们也提到这个解决方案有一个 Bug：被转换的行输出后，不是原来的那行。请修复这个问题。
3. POSIX 版本的 id 通过选项来控制要打印什么样的信息。修改 awk 版本（查看 ， ***在第 262 页***）来接收相同的参数并执行相同的结果。
4. split.awk 程序（查看 ， ***在第 263 页***）假设字母在字符集是连续的，但是对于 EBCDIC 的系统上却不是这样的。请修复这个问题。（提示：考虑一种不同的方法来遍历字母表，而不是使用 ord() 与 chr() 函数。）
5. 在 uniq.awk 中（查看 ，***在第 267 页***），选择哪一些来打印的逻辑表式一种状态机，即“某个设备可以处于某些稳定的状态依赖于其前面的状态以及它输入的当前值”。[[88]](#footnote-89)Brian Kernighan 则建议“状态机的另一种方式是只将输入读到数组中，然后直接使用索引。这几乎总是能够使用更简单的代码来实现，在你使用这个方式的大多数输入中，是非常快的。”请根据这样的建议来重写上面的逻辑。
6. 为什么 wc.awk （查看 ， ***在第 270 页***）程序不可以在 endfile() 中使用 FNR 的值？提示：检查 ， ***在第 233 页***，中的代码。
7. 在转换程序（查看 ）中使用标准 awk 函数操作单个的字符是很痛苦的。假设 gawk 可以用 “” 作分隔符来将字串分割为单独的字符，你会如何使用这个特定来简化程序？
8. extract.awk 程序（查看 ， ***在第 281 页***）是在 gawk 有 gensub() 函数之前写的。使用它来简化代码。

用更直接的方式来比较 awksed.awk 程序（查看 ， ***在第 285 页***）的性能：

**BEGIN {**

**pat = ARGV[1]**

**repl = ARGV[2]**

**ARGV[1] = ARGV[2] = ""**

**}**

1. 与真正的 sed 工具比起来，awksed.awk 有什么样的优缺点？
2. 在 ，***在第 286 页***，中我们提到，当在测试文件的可访问性时，不要尝试在 pathto() 函数中将 getline 读取的行进行保存，可以与主程序的一起使用时，极大地简化实现。那么这里面到底有什么问题？
3. 作为并不总是需要给程序加新功能的想法的一个额外的例子就是，假设有两个文件在搜索路径下的想法：

***default.awk***

这个文件包含默认函数集，如 getopt() 与 assert()。

***site.awk*** 这个文件包含的库函数特定于某个点或者安装版本，如本地的开发函数。用这样的一个独立文件可以使 default.awk 跟随 gawk 版本一同更新，而不需要系统的管理员添加一个本地函数的时候总是要更新它。

某个用户曾经建议 gawk 要修改成自动地在启动时读取这些文件。但是修改 igawk 来实现这个功能会更加简单。因为 igawk 可以处理嵌套的 @include 指示符，default.awk 可以只是简单地包含 @include 语句来引入所需要的库函数。请你做这些修改。

1. 修改 anagram.awk（查看 ， ***在第 292 页***）来避免使用外部的 sort 工具。

**Part III:**

用 gawk 超越标准 awk

# 第十二章 gawk 的高级特性

*写文档就要像把读它的人当成知道你住在哪的狂暴的精神变态者。*

*—Steve English， as quoted by Peter Langston*

这一章讨论 gawk 中的高级特性。他们只是互不相关的程序大麻袋中的一小点。首先，我们会看一下使 gawk 不识别输入数据，而不仅仅是在 awk 程序中的十进制点的选项。然后，再讨论 gawk 所特有的对数组进行排序的特性。接下来是双路 I/O 问题，在这之前有过简单的讨论，这里将与 TCP/IP 网络一起进行详细的讨论。最后，我们看一下 gawk 如何来进行评测 awk 程序，这样可以来对它进行性能上的调优。

其他的高级特性则在其他的章节中进行讨论：

* ***， 在第 311 页***，讨论如何来对你的 awk 程序进行国际化，这样它们可以说多国语言。
* ***，在第 320 页***，描述了 gawk 的内置的命令行调试器来进行 gawk 程序的调试。
* ***，在第 336 页***，展示了你如何来使用 gawk 来进行任意精度的算术运算。
* ***，在第 349 页***，讨论动态地给 gawk 添加新的内置函数的能力。

## 12.1 允许非十进制数数据输入

如果你用 --none-decimal-data 这个选项来运行 gawk，你可以在你的输入数据中使用非十进制数数值：

**$ echo 0123 123 0x123 |**

**> gawk --non-decimal-data ’{ *printf* "%d， %d， %d\n"， $1， $2， $3 }’**

**-| 83， 123， 291**

为了你这个特性可以工作，你需要在写程序时让 gawk 将这些数据当成是数值：

**$ echo 0123 123 0x123 | gawk ’{ print $1， $2， $3 }’**

**-| 0123 123 0x123**

print 语句将它的表达式当成是字串。尽管域在需要的时候可以表现成一个数值，但它们依然是字串，所以 print 不会尝试将它们以数值来看待。你需要向域加上一个 0 来强制它被当成数值。比如：

**$ echo 0123 123 0x123 | gawk --non-decimal-data ’**

**> { print $1， $2， $3**

**> print $1 + 0， $2 + 0， $3 + 0 }’**

**-| 0123 123 0x123**

**-| 83 123 291**

由于有十进制数之前有前导的 0 是很常见的，也由于使用这样的方式可能导致令人惊讶的结果，默认的情况是禁用的。如果你想使用，你必须显式地来指定它。

注意：不推荐使用这个选项。它会导致旧程序非常糟糕地破坏。相反，要使用 strtonum() 函数来转换你的数据（查看 ， ***在第 184 页***）。这也使得你的程序更容易写，也更容易读，结果也不那么令人惊讶。

这个选项可能在 gawk 的后续版本中消失。

## 12.2 控制数组排序与遍历

gawk 可以让你在 ‘for (indx in array)’的循环迭代数组时控制其顺序。

另外，有两个内置的函数，asrot() 与 asorti() 让你用基于数组的值与索引来对数组进行排序。还有两个函数用来在排序过程中控制元素的顺序的排序方案。

### 12.2.1 控制数据遍历

在默认情况下，在‘for (indx in array)’循环中来扫描数组的顺序是未定义的，它一般基于 awk 中对于数组的实现。

通常情况下，做为程序员的你是想来以某种特定的顺序来循环数组中的元素的。gawk 可以让你做到这个。

，***在第 171 页***，中描述了你可以赋值给某个特定预定义的变量 PROCINFO["sorted\_in"] 以来控制 gawk 在循环数组时的顺序。

另外， PROCINFO["sorted\_in"] 的值可以是一个函数名。[[89]](#footnote-90)这让你用自己定义的方案来遍历数组。数组的元素的排序由这个函数的返回值来决定。比较函数应该定义成至少有四个参数：

**function comp\_func(i1， v1， i2， v2)**

**{**

**compare elements 1 and 2 in some fashion**

**return < 0; 0; or > 0**

**}**

这里面 i1 与 i2 是索引，而 v1 与 v2 是两个正在比较的元素的值。如果正在被遍历的数组包含了子数组元素，v1 与 v2 之一或者两者都可能是数组。（查看 ，***在第 178 页***，来取得子数组的更多信息）三中可能的返回值如下所述：

***comp\_func(i1， v1， i2， v2) < 0***

在循环遍历时，索引 i1 会先于 i2。

***comp\_func(i1， v1， i2， v2) == 0***

索引 i1 与 i2 是相同的，但是其相对顺序是未定义的。

***comp\_func(i1， v1， i2， v2) > 0***

索引 i1 后在 i2 之后。

我们的第一个比较函数可以以索引的数值顺序来扫描数组：

**function cmp\_num\_idx(i1， v1， i2， v2)**

**{**

**# numerical index comparison， ascending order**

**return (i1 - i2)**

**}**

我们的第二个函数基于元素的字串值来遍历数组，而不是基于索引：

**function cmp\_str\_val(i1， v1， i2， v2)**

**{**

**# string value comparison， ascending order**

**v1 = v1 ""**

**v2 = v2 ""**

**if (v1 < v2)**

**return -1**

**return (v1 != v2)**

**}**

第三个比较函数使所有的数值，前后没空格的数字字串，会在循环时在前面：

**function cmp\_num\_str\_val(i1， v1， i2， v2， n1， n2)**

**{**

**# numbers before string value comparison， ascending order**

**n1 = v1 + 0**

**n2 = v2 + 0**

**if (n1 == v1)**

**return (n2 == v2) ? (n1 - n2) : -1**

**else if (n2 == v2)**

**return 1**

**return (v1 < v2) ? -1 : (v1 != v2)**

**}**

这是一个说明 gawk 使用前面不同函数时的不同表现时的主程序：

**BEGIN {**

**data["one"] = 10**

**data["two"] = 20**

**data[10] = "one"**

**data[100] = 100**

**data[20] = "two"**

**f[1] = "cmp\_num\_idx"**

**f[2] = "cmp\_str\_val"**

**f[3] = "cmp\_num\_str\_val"**

**for (i = 1; i <= 3; i++) {**

***printf*("Sort function: %s\n"， f[i])**

**PROCINFO["sorted\_in"] = f[i]**

**for (j in data)**

***printf*("\tdata[%s] = %s\n"， j， data[j])**

**print ""**

**}**

**}**

这是程序执行时的结果：

**$ gawk -f compdemo.*awk***

**-| Sort function: cmp\_num\_idx Sort by numeric index**

**-| data[two] = 20**

**-| data[one] = 10 Both strings are numerically zero**

**-| data[10] = one**

**-| data[20] = two**

**-| data[100] = 100**

**-|**

**-| Sort function: cmp\_str\_val Sort by element values as strings**

**-| data[one] = 10**

**-| data[100] = 100 String 100 is less than string 20**

**-| data[two] = 20**

**-| data[10] = one**

**-| data[20] = two**

**-|**

**-| Sort function: cmp\_num\_str\_val Sort all numeric values before all strings**

**-| data[one] = 10**

**-| data[two] = 20**

**-| data[100] = 100**

**-| data[10] = one**

**-| data[20] = two**

考虑对 GNU/Linux 系统的密码文件中的项按登录名来排序。下面的程序对特定域位置的比较来进行记录的排序可用于这个目的：

**# passwd-sort.*awk* --- simple program to sort by field position**

**# field position is specified by the global variable POS**

**function cmp\_field(i1， v1， i2， v2)**

**{**

**# comparison by value， as string， and ascending order**

**return v1[POS] < v2[POS] ? -1 : (v1[POS] != v2[POS])**

**}**

**{**

**for (i = 1; i <= NF; i++)**

**a[NR][i] = $i**

**}**

**END {**

**PROCINFO["sorted\_in"] = "cmp\_field"**

**if (POS < 1 || POS > NF)**

**POS = 1**

**for (i in a) {**

**for (j = 1; j <= NF; j++)**

***printf*("%s%c"， a[i][j]， j < NF ? ":" : "")**

**print ""**

**}**

**}**

密码文件中的每一项的第一个域是用户的登录名，域之间通过冒号分隔。每个记录都定义了一个子数组，每个域都是子数组中的一个元素。执行程序会产生如下输出：

**$ gawk -v POS=1 -F: -f sort.*awk* /etc/passwd**

**-| adm:x:3:4:adm:/var/adm:/sbin/nologin**

**-| apache:x:48:48:Apache:/var/www:/sbin/nologin**

**-| avahi:x:70:70:Avahi daemon:/:/sbin/nologin**

**...**

当给定一个特定的数组元素对时，比较应该总是返回相同的值。如果返回了不一致的值，则顺序是未定义的。这个行为可以用来给看起来有序的数据引入随机顺序：

**function cmp\_randomize(i1， v1， i2， v2)**

**{**

**# random order (caution: this may never terminate!)**

**return (2 - 4 \* *rand()*)**

**}**

如早前所提到的，如果两个元素比较相同，则两个元素的索引的顺序是任意的。这一般都不是什么问题，但是让两个在一起的元素以任意的顺序出现却是个问题，特别是大比较两者的值的时候。如果其他的元素加入或者从数组中移除，对相同元素间的部分排序在下次数组遍历时会改变。解决这个绑定元素的问题的方法之一是，当比较两元素值相同时，要再比较它们的索引。这么做的话，会使得循环遍历没有那高效，所以，只在必要的时候考虑使用它。下面的比较函数会一定会有确定的顺序，是基于两个元素的索引（字串）肯定不相同这样的事实：

**function cmp\_numeric(i1， v1， i2， v2)**

**{**

**# numerical value (and index) comparison， descending order**

**return (v1 != v2) ? (v2 - v1) : (i2 - i1)**

**}**

**function cmp\_string(i1， v1， i2， v2)**

**{**

**# string value (and index) comparison， descending order**

**v1 = v1 i1**

**v2 = v2 i2**

**return (v1 > v2) ? -1 : (v1 != v2)**

**}**

一个定制的比较函数常常可以简单循环遍历，而设计这样的函数的限制只受限于天空的大小。

当在进行排序时，如果使用的是字串比较，元素值之一或者两者都不会是数值，或者元素的索引会被当成字串来处理，IGNORECASE 的值（查看 ， ***在第 155 页***）控制比较是否将对应的大小写字母当成一样的字符来看待。

还要记住的一点是在子数组的情况下，元素值本身可以是数组，比较函数的结果应该使用 isarray() 函数（查看 ， ***在第 203 页***）来检测，并使用为子数组定义好的顺序。

所有基于 PROCINFO["sorted\_in"] 的排序在 POSIX 模式下都被禁用，因为 PROCINFO 数组在此种情况下并没有特殊的含义。

另一个提示就是，在遍历数组时对数组索引的排序已经有报告说会增加 awk 程序执行时 15% 到 20% 的负荷。由于这个原因，排序数组的遍历不作为默认行为。

### 12.2.2 用 gawk 对数组进行值与键的排序

在大多数的 awk 实现中，对数组排序需要写一个 sort() 函数。这对于学习不同的排序算法也是很有教育作用的，但是通过那不是程序的重点。gawk 提供了内置的 asort() 与 asorti() 函数（查看 ， ***在第 184 页***）来对数组排序。如：

**populate the array data**

**n = asort(data)**

**for (i = 1; i <= n; i++)**

**do something with data[i]**

在调用 asort() 之后，数组数据已经人 1 到某个数 n，即数组元素中的总数，进行了索引。（这个统计作为 asort() 的返回值。）data[1] ≤ data[2]≤ data[3] 依此类推。默认的比较是基于元素的类型（查看 ， ***在第 127 页***）。所有的数值类型在字串类型之前，然后是所有的子数组。

调用 asort() 函数的一个重要的副作用是数组的原始的索引不可恢复地丢失了。这并不总是所要的结果。asort() 可以授受第二个参数：

**populate the array source**

**n = asort(source， dest)**

**for (i = 1; i <= n; i++)**

**do something with dest[i]**

在这种情况下，gawk 会复制 source 数组到 dest 数组中，然后对 dest 进行排序，并导致其索引的破坏。但是， source 数组则不影响。

经常情况下，是需要对索引的值进行排序，而不是对元素的值进行排序。为了实现这个功能，则使用 asorti() 函数。这个接口与其行为与 asort() 是一样的，只是使用索引的值来进行排序，并且成为结果数组的值：

**{ source[$0] = some\_func($0) }**

**END {**

**n = asorti(source， dest)**

**for (i = 1; i <= n; i++) {**

**Work with sorted indices directly:**

**do something with dest[i]**

**...**

**Access original array via sorted indices:**

**do something with source[dest[i]]**

**}**

**}**

到目前为止都还好。现在开始进入最有趣的部分。asort() 与 asorti() 都接收第三个字串参数来控制数组元素的比较。当我们在 ，***在第 184 页***，引入 asort() 与 asorti() 时，我们忽略了第三个参数，但是现在是时候来描述这个参数如何来影响这两个函数。

基本上，第三个参数用来指定数组如何来进行排序。有两个可能。如 PROCINFO["sorted\_in"] 一样，这个参数可以是 gawk 所提供的预定义名字中的一个，或者也可以用户自定义函数的名字（查看 ， ***在第 298 页***）。

在后面的情况下，指定的函数可以以任何选择的方式来比较元素，可以只考虑索引，也可以只考虑值，或者两者都考虑。这就相当强大了。

一旦数组经过排序，asort() 会取得最终顺序的值，并使用他们填入到结果数组中，而 asorti() 取最终顺序的索引并使用它们填充到结果数组中。

提示：复制数组的索引与元素在内存中代价并不高。在内部，gawk 会保持数据的参考计数。如当 asort() 复制第一个元素到第二个时，只会有一个元素数据的复本，就算两个数组使用这个值。

因为 IGNORECASE 可以影响字串比较，IGNORECASE 的值也会影响 asort() 与 asorti() 的排序。也要注意本地语言设置中的排序则不会有影响，比较只基于字符的值。[[90]](#footnote-91)

## 12.3 与其他进程进行双向通信

把数据发向另一个单独的程序，处理后再读取结果通常都非常有用。这可以通过临时文件来做到：

**# Write the data for processing**

**tempfile = ("mydata." PROCINFO["pid"])**

**while (not done with data)**

**print data | ("subprogram >" tempfile)**

**close("subprogram >" tempfile)**

**# Read the results， remove tempfile when done**

**while ((*getline* newdata < tempfile) > 0)**

**process newdata appropriately**

**close(tempfile)**

**system("rm " tempfile)**

这可以工作，但是没那么优雅。在其他事情间，它需要程序运行一个不能被其全用户共享的目录中，比如 /tmp 就不可以，因为其他的用户有可能碰巧使用相同文件名的临时文件。[[91]](#footnote-92)

但是，使用 gawk 可以打开一个双路的管道到另一个进程中。第二个进程被称为并程，因为它与 gawk 一道运行。双路连接使用 ‘|&’操作符来建立（从 Korn shell， ksh 借来的）：[[92]](#footnote-93)

**do {**

**print data |&"subprogram"**

**"subprogram" |&*getline* results**

**} while (data left to process)**

**close("subprogram")**

第一次使用 ‘|&’ 操作符执行 I/O 操作时，gawk 会创建一个双路管线到进行另一个程序的子进程中。用 print 或者 printf 创建的输出会被写到程序的标准输入中，而程序的标准输出可以被 gawk 的程序使用 getline 来读取。与通过 ‘|’ 创建的程序一样，子程序可以是任何程序，或者程序的管线，可被 Shell 启动。

要了解的几个注意点是：

* 就目前 gawk 中的代码，并程的标准错误会输出到父进程 gawk 的标准错误输出的地方。不可以单独读取子进程的标准错误。
* I/O 缓存可能是个问题。gawk 会自动地刷出所有的输出到并程管道。但是，如果并程没有刷出它的输出，gawk 可能在执行 getline 以读取并程的输出时而挂起。这会导致所谓的死锁问题，在这种情形下，每个进程都在等待对方做些什么事情。

可以只关闭到并程的双路管道的一边，只需要为 close() 函数的提供第二个参数，即 “to” 或者 “from”（查看 ， ***在第 108 页***）。这些字串通知 gawk 关闭管道的一端，即发送数到并程的一端，或者从其读取的一端。

当使用 sort 系统工具作为并程的一部分时，上面的行为尤其重要，sort 必须在产生所有的输出之前先读取所有的输入数据。sort 程序直到 gawk 关键管道的写一端后，sort 才会接收到 end-of-file 的指示。

当你完成了数据输出到 sort 工具，你可以关闭管道的 “to”端，然后用 getline 从已经排序的数据中读取结果。例如：

**BEGIN {**

**command = "LC\_ALL=C sort"**

**n = split("abcdefghijklmnopqrstuvwxyz"， a， "")**

**for (i = n; i > 0; i--)**

**print a[i] |& command**

**close(command， "to")**

**while ((command |&*getline* line) > 0)**

**print "got"， line**

**close(command)**

**}**

这个程序将字母表的字母以逆序方式写入，一次一行，写到 sort 的双路管道中。之后关闭管道的写端，因此 sort 会接收到 end-of-file 指示。这会导致 sort 来排序数据，然后将排序后的数据写回到 gawk 程序。只要所有的数据被读取完毕，gawk 会结束并程，然后退出。

另外一个提示是，在 sort 命令的 LC\_ALL=C’赋值用来保证传统的 Unix（ASCII）排序。在这里并不是严格需要，但是知道如何来做也是很好的。

如果系统支持，你也可以使用 pseudo-ttys(ptys) 用来代替管道来做双路通信。这是通过基于单命令来完成的，只要设置 PROCINFO 数组的特殊元素的值来实现（查看 ， ***在第 158 页***），如：

**command = "sort -nr" # command， save in convenience variable**

**PROCINFO[command， "pty"] = 1 # update PROCINFO**

**print ... |& command # start two-way pipe**

**...**

使用 ptys 可以避免缓存死锁的问题，只是损失一些性能。如果你的系统没有 ptys，或者系统中所有的 ptys 都在使用，gawk 会自动地返回使用常规的管道。

## 12.4 利用 gawk 进行网络通信

EMRED:

*主机从海岸到海岸，没人可与已经关闭的主机喃喃私语，而没关的主机要么忙得死，要么挂起或死亡。*

*—Mike O‘Brien (aka Mr. Protocol)*

除了可以在同一个系统中打开一个双路到并程的管线（查看 ），也可以通过 IP 网络连接来与在另一个系统上的另一个进程之间进行双路连接。

你可以把这个当成是一个非常长的到并程的双路管线。gawk 确定你使用 TCP/IP 网络的方式是通过识别以 ‘/inet/’， ‘/inet4/’， 或者 ‘/inet6/’ 开头的特别文件名。

特殊文件名的完整语法为 /net-type/protocol/local-port/remotehost/remote-port。这些组件为：

net-type 指定要建立的互联网连接的类型。使用 ‘/inet4/’ 来强制使用 IPv4，‘/inet6/’来强制为 IPv6。普通文本 ‘/inet/’（过去是唯一的选项）用于系统默认方式，大多数可能是 IPv4。

protocol 在 IP 之上使用的协议。必须是 ‘tcp’，‘udp’之一，相应对应为 TCP 或者 UDP IP 连接。TCP 用于大多数程序。

local-port 本地 TCP 或者 UDP 端口。如果你想让系统选择端口，则指定‘0’为其端口号。如果你在写 TCP 或者 UDP 客户端则你应该这么来处理。你也可以使用知名的服务名，如‘smtp’或者‘http’，这样 gawk在使用 C 的 getaddinfo() 函数时会尝试确定预定义的端口。

remote-host

你要连接的主机的 IP 地址，或者是完整的 Internet 域名。

remote-port

连接远程主机的TCP 或者 UDP 端口。再次地，你如果不关心端口可以使用‘0’，或者知名的服务名。

提示：无法打开双路的 Socket 会导致一个致命错误返回到调用的代码。ERRNO 的值指示了错误信息（查看 ， ***在第 158 页***）。考虑下面非常简单的例子：

**BEGIN {**

**Service = "/inet/tcp/0/localhost/daytime"**

**Service |&*getline***

**print $0**

**close(Service)**

**}**

这个程序从本地系统的 TCP 时间服务器中读取当前的日期及时间。然后打印结果后关闭连接。

由于这个主题是扩展内容，用 gawk 进行 TCP/IP 编程为独立的文档。查看 TCP/IP Internetworking with gawk 文档。它是 gawk 发布版本的一部分，其中有完整的介绍与讨论，同时有更多的例子。

## 12.5 测评你的 awk 程序

你可以生成你的 awk 程序执行的跟踪信息。这只要传递 --profile 参数到 gawk 就可以。当 gawk 完成后，它会创建一个名为 awkprof.out 的测评文件。由于程序被测评，这样它要比正常的执行慢 45%。

如下面的例子所示，--profile 选项可以指定 gawk 将测评信息写入的文件的名字：

**gawk --profile=myprog.prof -f myprog.*awk* data1 data2**

在前面的例子中，gawk 将测评数据放在 myprog.prof 文件，而不是 awkprof.out 文件。这是一个简单的 awk 程序的执行信息，包含输入数据，以及用 --profile 执行 gawk 时的结果。首先，下面是 awk 程序：

**BEGIN { print "First BEGIN rule" }**

**END { print "First END rule" }**

**/foo/ {**

**print "matched /foo/， gosh"**

**for (i = 1; i <= 3; i++)**

***sing()***

**}**

**{**

**if (/foo/)**

**print "if is true"**

**else**

**print "else is true"**

**}**

**BEGIN { print "Second BEGIN rule" }**

**END { print "Second END rule" }**

**function sing( dummy)**

**{**

**print "I gotta be me!"**

**}**

下面是输入数据：

**foo**

**bar**

**baz**

**foo**

**junk**

下面是使用 gawk 测评器执行程序与数据时的产生的数据文件 awkprof.out（这个例子也说明了 awk 程序员有时很早就起来工作了）：

**# gawk profile， created Mon Sep 29 05:16:21 2014**

**# BEGIN rule(s)**

**BEGIN {**

**1 print "First BEGIN rule"**

**}**

**BEGIN {**

**1 print "Second BEGIN rule"**

**}**

**# Rule(s)**

**5 /foo/ { # 2**

**2 print "matched /foo/， gosh"**

**6 for (i = 1; i <= 3; i++) {**

**6 *sing()***

**}**

**}**

**5 {**

**5 if (/foo/) { # 2**

**2 print "if is true"**

**3 } else {**

**3 print "else is true"**

**}**

**}**

**# END rule(s)**

**END {**

**1 print "First END rule"**

**}**

**END {**

**1 print "Second END rule"**

**}**

**# Functions， listed alphabetically**

**6 function sing(dummy)**

**{**

**6 print "I gotta be me!"**

**}**

这个例子说明了测评输出的一些基本特性。它们是：

* 程序以 BEGIN 规则，模式-动作规则，ENDFILE 规则，END 规则这样的顺序来打印输出。并以字母顺序列表。多个 BEGIN 与 END 规则会有单独的标识，BEGINFILE 与 ENDFILE 也一样。
* 模式-动作规则有两个统计，第一个是规则的左边，显示的是规则的模式被测试的次数。第二个分是规则的右边在开花号中在注释里面，显示的是规则被执行的次数。两个数值的差异显示的是规则模式被测试为 false 的次数。
* 相似的，对于 if-else 语句的统计显示了条件被测试的次数。在开花括号的右边的 if 的语句体处的统计显示的是条件测试为真的次数。else 的统计显示的是测试失败的次数。
* 对于循环头的统计（如 for 或者 while）显示的是循环测试执行的次数。（由于这个原因，你不可只看在规则中的第一条语句的统计来确定规则被执行的次数。如果第一条语句是一个循环，则统计会误导你。）
* 对于用户自定义的函数，在 function 关键词后面的统计显示的是函数被调用的次数。在函数体语句语句后面的统计，则是语句被执行的次数。
* 输出布局使用的是 “K&R”带 TABs 的风格。花括号会到处使用，就算是 if，else，或者循环体只有一个语句。
* 括号则只在需要的时候使用，它由程序的结构 以及前面规则来提示。例如，‘(3 + 5) \* 4’ 表示 3 加 5，然后乘上 4 的结果。但是 ‘3 + 5 \* 4’ 没有括号，则表示 ‘3 + (5\*4) ’。
* 括号只用在 print 与 printf 参数周围，并且只当 print 与 printf 的语句后面跟着重定向符。相似的，如果重定向的目标不是标量，则它会被括起来。
* gawk 提供 BEGIN 与 END 规则，BEGINFILE 与 ENDFILE 规则，模式-动作规则与函数的前导注释。

你的程序的测评版本可能不会完全像你所写的那样。这是因为 gawk 用 “简洁打印”其程序的内部表示来生成的测评。这么做的好处是 gawk 可以产生标准的表示。不好的地方是会丢失程序的代码。同时，像这样的代码：

**/foo/**

会生成：

**/foo/ {**

**print $0**

**}**

虽然正确，但可能并不是所期望的。

当程序执行完成后会生成测评外，gawk 也可以在运行的时候生成。如果你的 awk 程序会进入一个无限循环，而你又想看执行了什么的时候，这会非常有用。为了使用这个特性，请在后台模式来执行带 --profile 选项的 gawk：

**$ gawk --profile -f myprog &**

**[1] 13992**

Shell 会打印一个任务号以及进程 ID 号，在这里是 13992。使用 kill 命令来发送 USR1 信号到 gawk：

**$ kill -USR1 13992**

如通常情况，测评的程序版本被写入到 awkprofile.out 文件中，或者写入到在 --profile 选项中指定的文件中。

作为常规的测评，如稍早显示的，测评中包含任何活动的函数的跟踪信息：

**# Function Call Stack:**

**# 3. baz**

**# 2. bar**

**# 1. foo**

**# -- main --**

你可以发送多次 USR1 信号。每发一次，测评与函数调用路径会被附加到测评文件中。

如果你使用的是 HUP 信号，而不是 USR1 信号，gawk 会产生测评与函数调用路径信息后退出。

当 gawk 在 MS-Windows 系统上运行时，它使用 INT 与 QUIT 信息来产生测评，在使用 INT 信号时，gawk 会退出。这里因为这些系统不支持 kill 命令，所以只能通过键盘产生的信号来发送给程序。INT 信号通过 Ctrl+c 或者 Ctrl+BREAK 键来产生，而 QUIT 信息则由 Ctrl+\ 键来产生。

最后，gawk 也接收其他的选项，--pretty-print。当这么调用时，gawk 会“简洁打印”程序到 awkprof.out 中，不带执行统计的方式。

提示：--pretty-print 会执行你的程序。这会在下一个主要版本中改变。

## 12.6总结

* --non-decimal-data 选项可以使 gawk 将八进制或者十六进制式的输入数据当成是八进制与十六进制数据对待。这个选项要小心使用，或者不要使用，而是要使用 strtonum()。也要注意这个选项在未来的 gawk 版本可能会消失。
* 你可以完全控制在 ‘for (indx in array)’中进行数组遍历的排序方式，这通过将预定义的 PROCINFO["sorted\_in"] 变量设置为用户自定义的函数名，而这个函数则通过对索引与值的比较来进行元素的比较。
* 类似地，你也可提供一个用户自定义的比较函数的名字来作为 asort() 或者 asorti() 函数的第三个参数来控制函数如何来对数组进行排序。或者你也可以提供一个预定义控制字串来设置 PROCINFO["sorted\_in"] 的值。
* 你可以使用 ‘|&’操作符来创建一个双路的到并程的管道。你通过 getline 来从并程中读取，并通过 print 或者 printf 来写入。使用 close() 来完全地关闭并程，或者可选地关闭双路通过中的一端。
* 通过使用特殊文件名以及 ‘|&’ 操作符，你可以使用 TCP/IP （或者 UDP/IP）来与互联网上的主机进行连接。gawk 支持 IPv4，也支持IPv6。
* 你可以生成程序的语句测评统计。这个可以让你来确定你程序的哪部分占用了较多的时间，以让你来方便地对你的程序进行调优。当进行测评进行时，你可以向程序发送 USR1 信号，这可以使 gawk 输出测评信息以及函数调用栈。
* 你也可以只“简洁打印”（pretty-print）你的程序。当前，它会运行你的程序，但是在后面的主要发行版本中会改变。

# 第十三章 gawk 的国际化

很久前，计算机的制造者们所写的程序只能使用英语。后来，硬件与软件的生产商位注意到，如果他们的系统也可以在非英语国家的语言中工作，则它他就可以卖更多的系统。结果就是，国际化与本地化程序以及软件会成为一个通用的实践。

很多年以来，提供国际化与本地化能力的程序失业地限制在用 C 或者 C++ 写的程序中。这一章会描述 gawk 底层使用的用来进行国际化的库，包含 gawk 如何来在 awk 程序代码级上实现国际化特性。在 awk 代码级实现国际化可以给软件开发者额外的灵活性——它们不再被迫使用 C 或者 C++ 来编写要实现国际化的软件。

## 13.1 国际化与本地化

国际化表示只写（编辑）一次程序，就可以不再需要对代码进行变更就可以在多种语言下使用。本地化则表示给国际化的程序提供相应的数据来在某个特定的语言中工作。最典型的是，这些术语指的是在打印错误信息，在读取反馈时使用的语言的特性，以及数值与倾向值如何来进行读取与打印的方式。

## 13.2 GNU 的 gettext

gawk 使用 GNU 的 gettext 来提供国际化特性。GNU gettext 工具关注信息：程序中打印的字串，无论是通过 printf 还是通过 sprintf()。[[93]](#footnote-94)

当使用 GNU gettext 时，每个程序都有它的文本域。这是一个唯一的名字，如‘kpilot’ 或者‘gawk’，用来标识程序。一个完整的程序有多个组件——用 C 或者 C++ 写的程序，以及使用 sh 或者 awk 写的脚本。所有的这些组件都使用相同的文本域。

为了使讨论更加具体，假设我们写一个程序名为 guide。按顺序，国际化包含下面的步骤：

1. 程序员检查所有的 guide 组件的代码，并且标注出要进行翻译的字串。例如“‘-F’: option required”是一个很好的要进行翻译的候选者。而选项名的字串表则不是（如，gawk 的 --profile 选项要保留原样，而不管是什么样的语言）。
2. 通过调用 textdomain() 函数，程序员向 gettext 库指定（“guide”）的应用文本域。
3. 从源代码中提取的消息放到一个移植的对象模板文件（guide.pot），在这个文件里列出了所有的字串以及对应的翻译串。翻译开始是空的。原始信息（一般是英语）会作为翻译查找的键。
4. 对于翻译的每中语言，guide.pot 都被复制到一个可移植的对象文件（.po）并建立起翻译然后与程序一起打包。例如，有可能有 fr.po 用来做为法语翻译。
5. 每种语言的 .po 文件都被转换为二进制消息对象（.gmo）文件。消息对象文件包含原始消息与它们的翻译的二进制格式，这样可以在运行时快速查找。
6. 当 guide 被构建并被安装，二进制的翻译文件也被安装到标准位置。
7. 对于测试与开发，可以告知 gettext 使用不是标准目录下的 .gmo 文件，这通过调用 bindtextdomain() 函数来实现。
8. 在运行时，guide 会通过调用 gettext() 来查找每一个字串。返回的串为字串的翻译串，如果没可得的话，就是原串。
9. 如果必要，也可以从另一个不属于这个程序的不同的文本域中来访问消息，而不需要在程序的默认文本域之间来回切换。

在 C（或者C++）中，标志动态翻译查找的字串是通过将字串通过 gettext() 来完成的：

***printf*("%s"， gettext("Don’t Panic!\n"));**

工具可以从源代码中将所有用 gettext() 包含的字串提取出来。

GNU gettext 的开发者，意识到一次次地输入 ‘gettext(...)’ 即痛苦，看起来也丑，因此使用宏 ‘\_’（下划线）来使得事情简化：

**/\* In the standard header file: \*/**

**#define \_(str) gettext(str)**

**/\* In the program text: \*/**

***printf*("%s"， \_("Don’t Panic!\n"));**

这产生的输入负担只是每个字串三个额外 的字符 ，而且也更容易阅读。

对于不同本地语言的信息有不同的本地语言分类。gettext 可识别的定义好的本地语言分类有：

LC\_MESSAGES

文本消息。这是 gettext 操作的默认分类，但是如果需要的话，可以显式地指定不同的类型。（几乎没有必要指定不同的分类。）

LC\_COLLATE

文本协同信息（如，不同的字符与/或者字符组在一个给定的语言中如何排序）。

LC\_CTYPE 字符类型信息（字母表，数字，大小写等等）以及字符编码。这个信息通过正则表达式的 POSIX 字符类来访问，比如 /[[:alnum:]]/（查看 ， ***在第 56 页***）。

LC\_MONETARY

货币信息，如货币符号以及符号是在数值前还是在数值后。

LC\_NUMERIC

数值信息。例如用于十进制小数点以及百分位分隔符。[[94]](#footnote-95)

LC\_TIME 时间与日期相关，如 12 或者 24 小时制，月分在日期中是打印在天的前面还是后面。本地月分的缩写等等。

LC\_ALL 之上所有的。（在 gettext 的内容中不是特别有用。）

## 13.3 awk 程序的国际化

gawk 使用下面的变量来用于国际化：

TEXTDOMAIN

这个变量表示应用的文本域。为了与 GNU gettext 的兼容性。其默认值是 “message”。

\_"your message here"

用前导的下划线标志的字串常量是在运行时用于翻译的候选串。字串常量前没有下划线的不会被翻译。

gawk 提供了下面的函数用来作国际化：

*dcgettext(string [， domain [， category]])*

在本地语言分类 category 中的 domain 文本域中返回 string 串的翻译。默认的domain 的值是 TEXTDOMAIN 变量的当前值。默认的 category 的值必须是已经“LC\_MESSAGES”。如果你为 category 提供了一个值，则它必须是前面所提的已知的本地语言分类字串名。你必需要提供文本域。如果你使用当前域则使用 TEXTDOMAIN。

注意：awk 版本的 ***dcgettext()*** 函数的参数的顺序有意地与 C 版本的中的顺序不一致。awk 版本的顺序的选择是为了简化以及允许合理的 awk 风格默认参数。

*dcngettext(string1， string2， number [， domain [， category]])*

返回 string1 与 string2 在本地语言分类 catagory 中，文本域 domain 中的翻译的复数形式。string1 是消息的英语的单数形式变体，string2 是相同消息的英语复数形式。domain 的默认值是 TEXTDOMAIN 的当前值。category 的默认值是 “LC\_MESSAGES”。

对于 ***dcgettext()*** 函数的参数的顺序说明，也适应于这个函数。

*bindtextdomain(directory [， domain ])*

改变 gettext 查找 .gmo 文件的上当，用在它们不在，或者不能放在标准位置的情况下（如，在测试期间）。返回文本域绑定的目录。

默认的域是 TEXTDOMAIN 的值。如果目录是空串（“”），***bindtextdomain()*** 则返回给定文件域 domain 的当前绑定目录。

为了在你的 awk 程序中使用这样的工具，请按照下面的步骤来执行：

1. 将 TEXTDOMAIN 变量设置为程序的文本域。最好是在 BEGIN 规则中完成（查看 ， ***在第 141 页***），或者可以通过命令行选项 -v 来完成（查看 ， ***在第 35 页***）：

**BEGIN {**

**TEXTDOMAIN = "guide"**

**...**

**}**

1. 将所有有可翻译的字串用前置下划线来进行标识（‘\_’）。它必须与字串的开引号相连。如：

**print \_"hello， world"**

**x = \_"you goofed"**

***printf*(\_"Number of users is %d\n"， nusers)**

1. 如果你动态地创建字串，你也可以通过 dcgettext() 内置函数来翻译它们：[[95]](#footnote-96)

**if (groggy)**

**message = dcgettext("%d customers disturbing me\n"， "adminprog")**

**else**

**message = dcgettext("enjoying %d customers\n"， "adminprog")**

***printf*(message， ncustomers)**

在这里，调用 dcgettext() 提供了一个不同的文本域（“adminprog”），在这里来查找消息，但是使用的是默认的“LC\_MESSAGES”分类。

前面的例子只在 ncustomers 大于 1 的情况下会工作。这个例子用 dcngettext() 可以更好地完成：

**if (groggy)**

**message = dcngettext("%d customer disturbing me\n"，**

**"%d customers disturbing me\n"， "adminprog")**

**else**

**message = dcngettext("enjoying %d customer\n"，**

**"enjoying %d customers\n"， "adminprog")**

***printf*(message， ncustomers)**

1. 在开发过程中，你可能想将 .gmo 文件放在一个私有的用于测试的目录。这可以通过 ***bindtextdomain()*** 内置函数来设置：

**BEGIN {**

**TEXTDOMAIN = "guide" # our text domain**

**if (Testing) {**

**# where to find our files**

**bindtextdomain("testdir")**

**# joe is in charge of adminprog**

**bindtextdomain("../joe/testdir"， "adminprog")**

**}**

**...**

**}**

查看 ，***在第 317 页***，来获取如何来一步步地创建并使用来自 awk 的翻译字串的例子。

## 13.4 翻译 awk 程序

只要程序中的可翻译字串已经被标志，它们就可以被提取用来创建初始的 .pot 文件。作为翻译的一部分，重排 printf 要输出的参数通常都很有帮助。

gawk 的 --gen-pot 命令行参数提出消息并在后面进行讨论。之后，也会涉及 printf 在运行时重排 printf 参数的能力。

### 13.4.1 提取已标志字串

一旦你的 awk 程序可以工作，而所有的字串都已经被标志，也设置了（可能也绑定了）文本域，就是该生成翻译的时候。首先，使用 --gen-pot 命令行选项来造成初始的 .pot 文件：

**gawk --gen-pot -f guide.*awk*> guide.pot**

当以 --gen-pot 运行时，gawk 不会执行你的程序。相反，它会按通常的方式分析它，然后打印所有的已经标志的字串到标准输出上，并且是以 GNU 的 gettext 的可移植对象文件的格式输出。同时在输出中也包含出现在 ***dcgettext()*** 函数的第一个参数中的字串常量，或者是 ***dcngettext()*** 函数中的第一、二个参数。[[96]](#footnote-97)你应该你你的 awk 程序一起发布你的 .pot 文件，翻译者最终也会使用它来为你提供翻译，你也可以将其发布。查看， ***在第 317 页***，来获取完整的为 guide 创建及测试翻译的步骤。

### 13.4.2 重排 printf 参数

用于 printf 与 sprintf() 函数的格式字串（查看 ）导致了一种特殊的翻译问题。考虑下面的代码：[[97]](#footnote-98)

***printf*(\_"String ‘%s’ has %d characters\n"，**

**string， length(string)))**

可能德语翻译成这样：

**"%d Zeichen lang ist die Zeichenkette ‘%s’\n"**

问题应该是很明显的：格式的指定与最初的是不同的！就算 gettext() 可以在运行时返回翻译后的字串，但是却没有办法改变调用 printf 参数的顺序。

为了解决这个问题，printf 格式指定符可以有另外一个可选的元素，即位置指定符。例如：

**"%2$d Zeichen lang ist die Zeichenkette ‘%1$s’\n"**

在这里，位置指定符由一个整数构成，用来表示要使用哪个参数，以及一年 $ 符号。位置是由1 开始的，不包含格式串自己。所以，在下面的例子中，‘string’是第一个参数， ‘length(string)’则是第二个参数：

**$ gawk ’BEGIN {**

**> string = "Don\47t Panic"**

**>*printf* "%2$d characters live in \"%1$s\"\n"，**

**> string， length(string)**

**> }’**

**-| 11 characters live in "Don’t Panic"**

如果提供了位置指定符，这在格式规则中要第一个出现，即在标志，域宽以及/或者精度之前。

位置指定符可以与动态域宽以及精度一起使用：

**$ gawk ’BEGIN {**

**>*printf*("%\*.\*s\n"， 10， 20， "hello")**

**>*printf*("%3$\*2$.\*1$s\n"， 20， 10， "hello")**

**> }’**

**-| hello**

**-| hello**

提示：当与位置指定符一起使用 ‘\*’时，‘\*’要先出现，然后是整数位置，然后才是 ‘$’符。这有一点反直觉。

gawk 不允许你在同样的字串中混用常规的格式指定符与位置指定符：

**$ gawk ’BEGIN { *printf* "%d %3$s\n"， 1， 2， "hi" }’**

**error gawk: cmd. line:1: fatal: must use ‘count$’ on all formats or none**

提示：有一个病态的列子中，gawk 无法识别。在这样的例子中，输出可能不是你所想要的。既使 gawk 没有检测到它们， 混用它们依然是个坏主意。

尽管位置指定符可以直接用在 awk 程序中，但它们的主要目的是用来帮助产生正确的格式串的翻译，从开始编写的语言到其他不同的语言。

### 13.4.3 awk 的移植性问题

gawk 的国际化特性有意选择对 awk 的可移植性影响最可能小，使它们可以用于其他的 awk 版本。考虑下面的程序：

**BEGIN {**

**TEXTDOMAIN = "guide"**

**if (Test\_Guide) # set with -v**

**bindtextdomain("/test/guide/messages")**

**print \_"don’t panic!"**

**}**

如上的代码，在其他的版本的 awk 中没办法工作。但是，实际它已经差不多可移植了，只需要一点点改动：

* 对 TEXTDOMAIN 的赋值没有效果，因为 TEXTDOMAIN 在其他的 awk 实现中没有特殊含义。
* 非 GNU 版本的 awk 将已经标志的变量当成是变量名为 \_ 与其他的字串的连接。[[98]](#footnote-99)典型的情况，变量 \_ 将空串当成它的值，这样最初的常量字串不变。
* 通过定义“哑”函数来代替 dcgettext()、dcngettext()与 bindtextdomain()，这样 awk 程序可以运行起来，但是只输出所有的初始语言的消息。

**function bindtextdomain(dir， domain)**

**{**

**return dir**

**}**

**function dcgettext(string， domain， category)**

**{**

**return string**

**}**

**function dcngettext(string1， string2， number， domain， category)**

**{**

**return (number == 1 ? string1 : string2)**

**}**

* 在 printf 与 snprintf 中的位置指定符是不可移植的。为了在 C 层次上支持 gettext()，很多的系统的 C 版本的 sprintf 也支持位置指定符。但是只要提供了足够的参数的调用时才会有用。很多的 awk 版本将 prinf 的格式与参数不变地传递给下层的 C 版本的 sprintf()，但是一次只传递一个格式与参数。大家可以猜测试如果使用了位置指定符的时候会发生什么。但是，由于位置指定符主要是用来做翻译格式的字串，而且因为非 GNU 的 awk 不会返回翻译后的字中，所以这在实践中不会成为一个问题。

## 13.5 简单的国际化例子

现在，让我们看看如何一步步地进行国际化与本地化一个例子 awk 程序，使用 guide.awk 作为我们的源代码：

**BEGIN {**

**TEXTDOMAIN = "guide"**

**bindtextdomain(".") # for testing**

**print \_"Don’t Panic"**

**print \_"The Answer Is"， 42**

**print "Pardon me， Zaphod who?"**

**}**

执行 ‘gawk --gen-pot’来创建 .pot 文件：

**$ gawk --gen-pot -f guide.*awk*> guide.pot**

这里会产生：

**#: guide.*awk*:4**

**msgid "Don’t Panic"**

**msgstr ""**

**#: guide.*awk*:5**

**msgid "The Answer Is"**

**msgstr ""**

这是最初的可移植对象模块文件，会被保存并在每个它要进行翻译的语言中进行重用。msgid 是最初的字串，而 msgstr 是翻译。

提示：字串没有用下划线进行标志的，不会出现在 guide.pot 文件中。

下一步，则要进行消息的翻译。这里的翻译是一个假想的英语方言，称为 “Mellow”：[[99]](#footnote-100)

**$ cp guide.pot guide-mellow.po**

**Add translations to guide-mellow.po ...**

下面是翻译：

**#: guide.*awk*:4**

**msgid "Don’t Panic"**

**msgstr "Hey man， relax!"**

**#: guide.*awk*:5**

**msgid "The Answer Is"**

**msgstr "Like， the scoop is"**

下一步是创建一个目录来存储二进制消息对象文件，然后创建 guide.mo 文件。我们假装我们的文件准备在 en\_US.UTF-8 语言设置中使用，因为我们必须使用一个 C 的gettext 例程知道的语言设置。这里显示的目录布局表示的是 GNU/Linux 系统上的 GNU gettext。其他版本的 gettext 可以使用不同的布局：

**$ mkdir en\_US.UTF-8 en\_US.UTF-8/LC\_MESSAGES**

msgfmt 工具将人可读的 .po 文件转换成机器可读的 .mo 文件。默认情况下，msgfmt 创建一个叫做 messages 文件。这个文件必须被重新命名，并被放在合适的位置（使用 -o 选项），因此 gawk 可以找到它：

**$ msgfmt guide-mellow.po -o en\_US.UTF-8/LC\_MESSAGES/guide.mo**

最后，我们运行程序来测试：

**$ gawk -f guide.*awk***

**-| Hey man， relax!**

**-| Like， the scoop is 42**

**-| Pardon me， Zaphod who?**

如果 ***dcgettext()***， ***dcngettext()***， 与 ***bindtextdomain()*** 这三个函数在文件 libintl.awk 文件中，然后我们可以这样的运行 guide.awk：

**$ gawk --posix -f guide.*awk* -f libintl.*awk***

**-| Don’t Panic**

**-| The Answer Is 42**

**-| Pardon me， Zaphod who?**

## 13.6 gawk 可以说你的语言

gawk 本身已经使用 GNU gettext 包来做了国际化。（GNU gettext 在 GNU gettext 工具里有完整的描述）在写这个的时候，GNU gettext 的最新版本是 0.19.4。

如果 gawk 的消息翻译存在，gawk 产生的信息，警告，命令错误会使用本地语言。

## 13.7总结

* 国际化表示写的程序可以使用多种语言而不需要对源代码进行改变。本地化则表示给已经国际化的语言提供必要的数据来使其他特定的语言中工作。
* gawk 使用 GNU gettext 来让你对 awk 程序进行国际化与本地化。一个程序的文本域标识了用于组合消息与其他数据的程序。
* 你将程序中的字串用下划线来进行标识以用于翻译。一旦完成了这个，字串会被提取到一个 .po 文件中。这个文件为每种语言的复制一个 .po 文件，然后 .po 文件会编译成 .gmo 文件来在运行时使用。
* 你可以在 sprintf 与 printf 中使用位置指定符，使可以在格式化串与输出中输出中重新对参数值进行排序。这对于控制字串的翻译很有用。
* 国际化特别被设计成在标准的 awk 中也可以很好处理。
* gawk 本身已经国际化，并发布了多种语言消息。

# 第十四章 调试 awk 程序

如果程序在第一次运动就表现完全那就好了，但是在现实生活中，这对于任意复杂度的程序来说都很少见。所以，大多数的程序语言都有相应的工具用来进行“调试”程序，而 awk 也不例外。

gawk 的调试器有意地参照 GNU 调试器（GDB）的命令行调试器。如果你熟悉 GDB，学习如何来使用 gawk 调试你的程序就很简单。

## 14.1 gawk 调试器介绍

本节一般性地介绍一下调试器以及开始讨论 gawk 的调试。

### 14.1.1 通用的调试技术

（如果你使用过其他语言的调试器，你可以跳过到 ，***在第 321 页***。）

当然，调试程序没办法帮你移除 Bug，因此它并不知道你或者你的用户如何将其当成是个 Bug，还是一个特性。（有时候，人类自己对于这个问题也挣扎了很久。）在这个情况下，你从这样的工具想得到什么呢？答案依赖于正在被调试的语言，但是一般来说，你至少可以得如下的结果：

* 有能力查看程序指令一条条地执行，这样你这样的程序员就有机会考虑在秒级，分钟级，或者小时级的时间上发生了什么，而不是纳秒级的时间上。在这个时间上代码总会是执行的。
* 有机会不仅是被动地发现你的程序的操作，而且能够控制它，并且尝试不同的执行路径，而不需要改动你的源代码。
* 有机会在任何的执行时间点看到数据的值，并且在执行时候改变它的值，以测试后续其如何来影响程序。（这一般都包括查看内部数据结构的能力，包含你在代码中定义的变量。）
* 有能力取得程序状态的额外 信息，甚至是内部结构。

所有这样的工作都提供大量的帮助来使用你自己的技能来理解你的程序的目标，以此来发现哪里出了错（或者，通过这样可以更好地理解你或者别人写的功能程序）。

### 14.1.2 调试概念

在深入细节之前，我们需要介绍关于调试器的几个重要概念。下面的列表定义了本章中后面使用到术语：

*Stack frame*

程序一般在它们的执行过程中调用函数。一个函数调用另一个函数，或者函数调试自己（递归）。你可以查看函数调用的链（主程序调用 A，而 A 调用 B，B 再试用 C），作为执行函数栈：当前运行的函数在栈的最顶端，当它完成时（返回），下一个则成会活动函数。这样的一个栈术语上叫做调用栈。

在调用栈上的每一个函数，系统都会保留一个数据区来包含函数的参数，本地变量以及返回值，以及其他的“登记”信息，以用来管理调用栈。这个数据区被称为栈帧。

gawk 也使用这样的模型，可以让你访问到调用栈以及每个栈帧。你可以看到调用栈，以及从栈中的哪个函数调用。用来打印调用栈的命令打印每个栈帖的信息（后面有详细内容）。

*Breakpoint*

在调试期间，你常常希望程序执行到一个特定的点，然后继续从那里一次执行一条语句。要这么做的方式是在程序中设置一个断点。一个断点是程序执行时会暂停（停止）的位置，因此你可以取得程序执行控制权。你可以添加或者移除多个断点。

*Watchpoint*

一个查看点与断点类型。不同的是断点是面向代码的，当代码执行到达后，会停止执行。一个查看点，则指定的程序当某个值改变时会停止。这会相当有用，因为有时变量接收了一个错误的值，而只查看代码很难跟踪。通过使用查看点，你可以在变量被赋值时而停下来，这样通常可以很快地查找到错误的代码。

### 14.1.3 awk 调试

调试 awk 程序有一些方面与调试其他语言写成的程序有一些不同。

首先，awk 程序通常是从一个或者多个文件中一行行地读取数据，然后使用特定的规则操作这些行，这样，根据这些规则来查看程序的程序的执行是非常有用的。正如我们见到的，每一个 awk 规则都像一个函数一样，它们有自己特定的指令块。

另外，由于 awk 设计上是一个很简练的语言，这样很容易看走眼，而不知道每一个行的 awk 程序代码里面到底发生了什么。使用更高层的 awk 命令，调试器使得我们有机会来查看单个的原始命令。

## 14.2 gawk 调试例子

为了解释 gawk 调用的使用，让我们来看一个例子的调试过程。我们利用早前提到的 POSIX uniq 命令的 awk 实现（查看 ， ***在第 267 页***）来作为我们的例子。

### 14.2.1 如何开启调试器

开始调试器几乎就跟正常运行 gawk 一样，只是你需要传递一个额外的选项 --debug，或者其对应的短选项 -D。包含程序的文件，或者任何受支持的在命令行上的代码，都作为一个或者多个 -f 选项的参数。（gawk 没有为命令行程序设计调试，只支持包含在文件的程序）在我们的例子中，我们这样来调用调试器：

**$ gawk -D -f getopt.*awk* -f join.*awk* -f uniq.*awk* -1 inputfile**

***getopt.awk*** 与 ***uniq.awk*** 两个程序都在 $AWKPATH 目录中。（GDB 或者类似调试器有经验的用户都应该注意到了这个语法与你之前所使用的语法有一些轻微的差异。在 gawk 调试器里，你将在命令行上传递给运行程序的参数给了调试器，而不是作为在调试指示符下命令行的一部分。-1 是 uniq.awk 的选项。

不像 gawk 通常所做的那样，立即对输入文件执行程序，调试器仅仅载入所有的程序源代码，在内部编译它们，然后给出提示符：

**gawk>**

从这里，你们可以发送命令到调试器。在这一点上，还没有代码被执行。

### 14.2.2 查找 Bug

假充我们在使用（一个有错误的版本） uniq.awk 中的 “域跨越”模式时有问题，看起来，当要跳过第一个域时，它没有捕获到本来相同的行，如：

***awk* is a wonderful program!**

**gawk is a wonderful program!**

这有可能会发生，如果我们将在记录中的域想成是一个以 0 开始（C 风格）编号的，所以本来应该写成下面的行：

**clast = join(alast， fcount+1， n)**

**cline = join(aline， fcount+1， m)**

我们写成了：

**clast = join(alast， fcount， n)**

**cline = join(aline， fcount， m)**

我们第一件要做的事常常是要检查出这样的问题是在程序中设置一个断点，这样我们可以查看它们如何工作，并捕获到发生的错误。在 uniq.awk 中合理的断点是在函数 are\_equal() 的开始处，在这个函数中，会将当前行与前一个行进行比较。要设置断点，使用 b（breakpoint）命令：

**gawk> b are\_equal**

**-| Breakpoint 1 set at file ‘awklib/eg/prog/uniq.*awk*’， line 63**

调试器告诉我们断点所在的文件以及行号。现在，我们输入 ‘r’ 或者 ‘run’，然后程序会执行，直到其第一次命令断点：

**gawk> r**

**-| Starting program:**

**-| Stopping in Rule ...**

**-| Breakpoint 1， are\_equal(n， m， clast， cline， alast， aline)**

**at ‘awklib/eg/prog/uniq.*awk*’:63**

**-| 63 if (fcount == 0 && charcount == 0)**

**gawk>**

现在，我们可以查看程序中哪里出了问题。首先，让我们看一下我们是怎么来到现在的位置。在提示符下，我们输入 ‘bt’（“backtrace”的缩写），则调试器会返回当前栈帧的列表：

**gawk> bt**

**-| #0 are\_equal(n， m， clast， cline， alast， aline)**

**at ‘awklib/eg/prog/uniq.*awk*’:68**

**-| #1 in *main()* at ‘awklib/eg/prog/uniq.*awk*’:88**

这告诉我们 are\_equal() 是在主程序 uniq.awk 第 88 行调用的。（这没什么大惊小怪的，因此这是在程序中唯一调用 are\_equal() 的地方，但是在更复杂的程序中，了解谁调用了函数，以及相应的参数是查找到程序问题的关键。）

现在，我们已经在 are\_equal() 中了，我们开始来查看某些变量的值。假设我们输入 ‘p n’（p 是 “print”的缩写）。我们期望看到 are\_equal() 参数 n 的值。实际是，调试器返回：

**gawk> p n**

**-| n = untyped variable**

在这个例子中，n 是一个未初始化的本地变量，因为函数调用时没有提供绘画它参数（查看 ， ***在第 133 页***）。要显示的更有用的变量应该是当前记录：

**gawk> p $0**

**-| $0 = "gawk is a wonderful program!"**

在一开始，这个可能会有一点疑惑，因为这是我们的测试输入的第二行，我们来看看 NR：

**gawk> p NR**

**-| NR = 2**

因此，我们看到 are\_equal() 只会从文件的第二个记录开始调用。当然，这是因为我们包含了这样的一个规则 ‘NR==1’：

**NR == 1 {**

**last = $0**

**next**

**}**

好了，让我们查看当前正在执行的规则：

**gawk> p last**

**-| last = "*awk* is a wonderful program!"**

目前我们所做的一切都只是说明程序正如计划的那样执行，直至 are\_equal() 为止，包含函数调用，因此问题一定出在函数内部。为了进一步调查，我们必须“一步步”地执行 are\_equal() 的代码行。我们敲入 ‘n’（表示 “next”）：

**gawk> n**

**-| 66 if (fcount > 0) {**

这告诉我们，gawk 现在已经准备执行第66行，确定这是否需要给行特殊的“域跨越”处理，正如命令行选项 -1 所示的那样。（请注意，我们是从我们之前所在的位置，第63行，跳到目前所在的位置，因为在 63 行的条件是 ‘if (fcount == 0 && charcount == 0)’，而其值为 false。）

继续执行一步，我们现在到了将当前与最后记录进行分割的地方：

**gawk> n**

**-| 67 n = split(last， alast)**

**gawk> n**

**-| 68 m = split($0， aline)**

在这一点上，我们很好奇地去查看记录被分割成了什么，我们试着去查看：

**gawk> p n m alast aline**

**-| n = 5**

**-| m = untyped variable**

**-| alast = array， 5 elements**

**-| aline = untyped variable**

（p 命令可以有多于一个参数，与 awk 的 printf 语句类似。）

但是，这里有一点失望。我们找到的是在里面有 5 个元素的 alast，m 与 aline 没有值，因为我们在行 68，但是还没有执行。这个信息已经够用了（我们现在知道没有单词被无意间丢掉了），但是如果我们想知道数组里有什么怎么办？

第一个选择是使用下标：

**gawk> p alast[0]**

**-| "0" not in array ‘alast’**

噢！！！

**gawk> p alast[1]**

**-| alast["1"] = "*awk*"**

如果有 100 个成员的数组，这就太慢了，所以，gawk 提供了一个快捷方式（对于其他语言的旧情怀就不提了）：

**gawk> p @alast**

**-| alast["1"] = "*awk*"**

**-| alast["2"] = "is"**

**-| alast["3"] = "a"**

**-| alast["4"] = "wonderful"**

**-| alast["5"] = "program!"**

目前看起来，一切 OK，让我们再执行一两步：

**gawk> n**

**-| 69 clast = join(alast， fcount， n)**

**gawk> n**

**-| 70 cline = join(aline， fcount， m)**

好了，我们已经到达了错误的地方（抱歉破坏了悬念）。在我们的大脑中是从第二个域开始来生成一个假记录来进行比较。而如果第一个记录编号为 0，这就会工作。让我们看看我们得到什么：

**gawk> p cline clast**

**-| cline = "gawk is a wonderful program!"**

**-| clast = "*awk* is a wonderful program!"**

嘿，看起来是不是很熟悉！他们就是原来的值，没有改变。思考片刻（人的大脑依然是最好的调试工具），我们知道我们少加了1！

我们退出调试器：

**gawk> q**

**-| The program is running. Exit anyway (y/n)? y**

然后进入编辑器编辑：

**clast = join(alast， fcount+1， n)**

**cline = join(aline， fcount+1， m)**

然理问题解决！

## 14.3 主要的调试器命令

gawk 调试器命令可以被分为下面的几类：

* 断点控制
* 执行控制
* 查看与更改数据
* 使用栈
* 获取信息
* 其他

每个个类型都在下面的节中进行讨论。在下面的描述中，命令，可能被缩写显示在第二个描述行。调试器命令名如果没有二义，则也可以被丢弃后面一部分。调试器有内置的自动重复前一个命令的能力，只需要输入 “回车”键。这可用于命令列表，next，nexti，step，stepi，以及 continue 等不带参数的命令。

### 14.3.1 控制断点

如早前看见，你要开启一个调试过程时，要考虑的第一件事情是设置好断点，因为程序会像没有调试器一样运行下去。用于控制断点的命令有：

***break [[filename:]n | function] ["expression"]***

***b [[filename:]n | function] ["expression"]***

如果不带任何参数，则在所选择的栈中的下一个要执行的指令处设置一个断点。参数可以是如下格式：

*n* 在当前的源文件中的 n 行设置一个断点。

*filename:n*

在源文件 filename 的 n 行设置一个断点。

*function* 在函数 function （第一条指令）入口处设置一个断点。

每一个断点被赋给了一个数值，可以用 delete 命令来从断点列表中删除。

对于断点，你还可以提供一个条件。这是一个 awk 的表达式（包在双引号中），这样调试器当到到断点时可以对其求值。如果条件为 true，则调试器停止执行，然后提示等待命令输入。否则，它会继续执行程序。

***clear [[filename:]n | function]***

如果没有参数，则在当前的栈中删除下一条要执行命令的所有断点。如果程序在断点上停止了，这会删除那个断点，这样程序在那个位置不会再停下来。参数可以有下面的形式：

*n* 在当前源文件中删除在 n 行上设置的断点。

*filename:n*

删除源文件 filename 行 n 中的断点。

*function* 删除源函数 function 的断点。

***condition n "expression"***

给已经存在的断点或者查看点加一个额外的条件。条件是一个 awk 表达式，用双引号括起来，当断点或者查看点到达时，会对其进行求值。如果条件为 true，则调试器会停止执行然后提示命令输入。否则，调试器会继续执行程序。如果条件表达式没有被指定，则已经存在的条件会被删除（即，断点或者查看点会被设置成无条件的）。

***delete [n1 n2 . . . ] [n–m]***

***d [n1 n2 . . . ] [n–m]***

删除指定的断点，或者断点范围。如果没有指定参数，则删除所有的断点。

***disable [n1 n2 . . . | n–m]***

禁用指定的断点或者断点范围。如果没有带参数，则禁用所有的断点。

***enable [del | once] [n1 n2 . . . ] [n–m]***

***e [del | once] [n1 n2 . . . ] [n–m]***

启用指定的断点或者范围内的断点。如果没有参数，则启用所有的断点。你也可以选择性地定义如何来恢复断点：

*del* 临时启用断点，然后程序在此停止时删除每个断点。

*once* 临时启用断点，然后程序在此停止时禁用每个断点。

***ignore n count***

忽略断点 n 后面命中的 count 次。

***tbreak [[filename:]n | function]***

***t [[filename:]n | function]***

设置临时断点（只启用一次）。参数与 break 一样。

### 14.3.2 控制执行

现在，你的断点已经准备好了，你可以开始运行程序，然后观察它的行为。有多个命令用来控制程序的执行，之前我们已经看过：

***commands [n]***

***silent***

***. . .***

***end***设置一个命令列表，当在断点或者查看点处停止时会被执行。n 是断点或者查看点编号。如果没有编号，则会使用最后一个。后面的命令，从下一行开始，然后在 end 命令处结束。如果命令 silent 在列表中，则通常的在断点停止处的消息以及源代码就会被打印。在列表中的所有命令如果继续执行代码（如 continue）会结束列表（一个隐式的 end），后面的命令会被忽略掉。如：

**gawk> commands**

**> silent**

**>*printf* "A silent breakpoint; i = %d\n"， i**

**> info locals**

**> set i = 10**

**> continue**

**> end**

**gawk>**

***continue [count]***

***c [count]*** 继续执行程序。如果从断点处继续，而指定了 count，则忽略所在处的断点后面的 count 次命中。直到所选择的帧结束。并打印返回值。

***next [count]***

***n*** ***[count]*** 继续执行下一行源代码，函数调用一次划过。参数 count 控制重复多少次，与 step 一样。

***nexti [count]***

***ni [count]*** 执行 1 个或者 count 个指令，并划过函数调用。

***return [value]***

取消函数调用。如果 value（一个字串或者数值）被指定了，则它会作为函数的返回值。如果是在不是最内层（当前的执行函数，即帧 0）的帧中使用，会取消所有最内层的帧，包括所在的帧，该帧的调用者会成为最内层帧。

***run***

***r*** 开始/重新开始程序的执行。当重新开始时，调试器会保留当前的断点，查看点以及命令历史，自动显示变量，以及调试器选项。

***step [count]***

***s [count]*** 继续执行直到控制流到到当前帧的源代码行的不同位置，并进入该行调用的函数中。如果参数 count 被指定了，则表示在停止前单独执行多少次，除非它遇到了断点或者查看点。

***stepi [count]***

***si [count]*** 执行 1 个（或者 count）个指令，并进入函数调用中。（为了说明“指令”的含义，查看在 ，***在第 332 页***，中 dump 下的输出。）

***until [[filename:]n | function]***

***u [[filename:]n | function]***

如果没有任何参数，执行到当前帧当前行的下一行。如果有参数，则持续执行到指定的位置，或者直到当前帧返回。

### 14.3.3 查看与更改数据

gawk 中用于查看及更改变量的命令有：

***display [var | $n]***

将变量 var 或者域 $n 添加到显示列表中。变量的值或者域的值在程序每次停止都会被显示。每个加入到列表中的列表都通过一个唯一的数字来区别：

**gawk> display x**

**-| 10: x = 1**

这显示了被赋值的项，变量名以及其及当前的值。如果显示变量指向了函数的参数，它会在执行到没有这个变量名的上下文时默默地将它从显示列表中删除。如果没有参数，display 会显示列表中的项的当前值。

***eval "awk statements"***

对 awk 语句在执行程序中的上下文中进行求值。你可以做任何的 awk 程序可以做的事情：赋值绘画变量，调用函数等等。

***eval param， . . .***

***awk statements***

***end*** 这个形式的求值类似，但是可以让你定义存在 awk 语句上下文中“本地变量”，而不是使用由程序定义的变量或者函数参数。

***print var1[， var2 . . . ]***

***p var1[， var2 . . . ]***

打印 gawk 变量或者域的值。域必须是通过常量来参考：

**gawk> print $3**

这打印输入记录中的第三个域（如果指定的域不存在，它会打印 ‘Null field’）。变量可以是数组元素，下标是常量字串。为了打印数组内容，在名字的前面加一个‘@’ 符号：

**gawk> print @a**

这会打印索引以及所有在数组 a 中相应元素的值。

***printf format [， arg . . . ]***

打印格式式的文本。格式包含转义序列，如 ‘\n’（查看 ， ***在第 51 页***）。如果不指定，不会打印换行符。

***set var=value***

将一个常量（数值或者字串）值赋值绘画一个 awk 变量或者域。字串值必须用双引号包起来（"..."）。

你也可以设置 awk 的特殊变量，像 FS，NF，NR 等等。

***watch var | $n ["expression"]***

***w var | $n ["expression"]***

将变量 var 或者域 $n 加到查看列表中。调试器会在变量或者域的值被改变时停下来。每个一个查看列表项都赋予了一个数值，这样可以用中 unwatch 命令中来它从查看列表中删除。

你可以绘画查看点附加一个条件。这是一个 awk 表达式（包在双引号中），这样当查看点被命中时，调试器会对其进行求值。如果条件为 true，则调试器会停止执行，然后提示命令输入。否则，gawk 会继续执行命令。

***undisplay [n]***

删除编号为 n 的自动显示列表项（如果没有提供参数，则删除所有的项）。

***unwatch [n]***

删除编号为 n 的查看列表中的项（如果没有提供参数，则删除所有的项）。

### 14.3.4 程序栈

无论何时你运行一个包含任意函数调用的程序，gawk 都会保持一个所有函数调用的栈，一直向上到正在执行的程序处。你可以查看你你如何到达目前所在到位置，也可以按照栈的路径来移动以来查看在你所在的被调用的函数中事件的状态。下面的命令来实现这样的功能：

***backtrace [count]***

***bt [count]***

***where [count]***

打印所有函数调用（栈帧）的路径，或者在 count > 0 时最内层 count 个栈帧。如果 count < 0，则打印最外层的 count 个栈帧。路径显示每一个函数的名字以及参数，还有文件名与行号。where 别名用于向长期使用 GDB 的用户提供栈路径，他们可能会使用这个命令。

***down [count]***

向下最内层的帧移动 count（默认为 1）个帧。然后选择并打印这个帧。

***frame [n]***

***f [n]*** 选择并打印栈帧 n。帧 0 是当前正在被执行的，或者最内层帧（函数调用），帧 1 是调用最内层的帧。最高层的帧是主程序。被打印的信息包括帧号，函数以及参数名，源文件与行号。

***up [count]*** 向上最外层移动 count（默认为 1）帧。然后选择并打印相应的帧。

### 14.3.5 获取程序与调试器状态信息

除了查看变量的值外，也常常需要来获取其他关于程序状态的信息，以及调试环境本身的信息。gawk 调试器有一个命令来提供这样的信息，非常恰当地叫做 info。info 与多个参数中的一个使用，可以告诉你想要的准备的信息：

***info what***

***i what*** 应该是下在其中之一的值：

args 列出所选择的帧的参数。

break 列出当前的所有断点。

display 列出自动显示的所有项。

frame 被选择的帧的描述。

functions 列出所有的函数定义，包括源文件名以及行号。

locals 列出被选择帧的本地变量列表。

source 打印当前源文件的名字。每次程序停止时，当前的源文件是包含当前指令的文件。当调用第一次开始时，当前的源文件是通过 -f 指定的文件。‘list filename:lineno’命令可以在任何时候用来改变当前源文件。

sources 列出所有的程序源。

variables 列出所有的全局变量。

watch 列出所有的在查看列表中的项。

其他的命令可以让你控制调试器，使其有保存调试器的状态以及从文件中运行调试器的能力，

***option [name[=value]]***

***o [name[=value]]***

如果没有参数，显示所有可用的调试器选项以及它们的当前值。‘option name’显示指定名字选项的当前值。‘option name=vale’将一个新值赋给指定的选项。可用的选项有：

*history\_size*

设置保存在历史文件 ./.gawk\_history 文件中的最大行数。默认值是 100。

*listsize* 指定列表打印的行数。默认是 15。

*outfile* 将 gawk 的输出发送到文件，调试器输出依然发送到标准输出上。空串（""）会重新将设置输出到标准输出上。

*prompt* 改变调试器的提示。默认是 ‘gawk>’。

*save*\_*history [on | off]*

保存命令历史到文件 ./.gawk\_history。默认是 on。

*save*\_*options [on | off]*

在退出时，保存当前的选项到文件 ./.gawkrc。默认是 on。选项会在启动下一个会话过程时读取回来。

*trace [on | off]*

打开或者关闭命令跟踪。默认是 off。

***save filename***

将来自于当前会话的命令保存到所给的文件中，所以他们可以使用 source 命令进行重放。

***source filename***

从文件中执行命令，任何命令的错误不会导致后续命令的结束。注释（以‘#’开头的行）可以存在命令文件中。空行会被忽略，它们不会重复最后的命令。在文件中，你不可以有多于一个的 run 命令来重新启动程序。同时，命令列表可以包含其他的源的命令，但是，gawk 调试器不会对同一个文件执行多次 source，以避免无限递归。

为了使用或者代替 source 命令，你可以使用 -D file 或者 --debug=file 命令行参数从一个文件中来非交互式地执行命令（查看 ， ***在第 35 页***）。

### 14.3.6 其他命令

有一些命令不太好归入到之前的那几种分类，他们是：

*dump [filename]*

*Dump 程序的字节码到标准输出中，或者名字为 filename 的文件中。这会打印指令的内部表示，gawk 通过执行这些指令来实现程序中的命令。这会非常有启发性，如下面的 Davide Brini 的含人无法捉摸的代码的部分 dump 内容（查看 ，* ***在第 293 页****）：*

**gawk> dump**

**-| # BEGIN**

**-|**

**-| [ 1:0xfcd340] Op\_rule : [in\_rule = BEGIN] [source\_file = brini.*awk*]**

**-| [ 1:0xfcc240] Op\_push\_i : "~" [MALLOC|STRING|STRCUR]**

**-| [ 1:0xfcc2a0] Op\_push\_i : "~" [MALLOC|STRING|STRCUR]**

**-| [ 1:0xfcc280] Op\_match :**

**-| [ 1:0xfcc1e0] Op\_store\_var : O**

**-| [ 1:0xfcc2e0] Op\_push\_i : "==" [MALLOC|STRING|STRCUR]**

**-| [ 1:0xfcc340] Op\_push\_i : "==" [MALLOC|STRING|STRCUR]**

**-| [ 1:0xfcc320] Op\_equal :**

**-| [ 1:0xfcc200] Op\_store\_var : o**

**-| [ 1:0xfcc380] Op\_push : o**

**-| [ 1:0xfcc360] Op\_plus\_i : 0 [MALLOC|NUMCUR|NUMBER]**

**-| [ 1:0xfcc220] Op\_push\_lhs : o [do\_reference = true]**

**-| [ 1:0xfcc300] Op\_assign\_plus :**

**-| [ :0xfcc2c0] Op\_pop :**

**-| [ 1:0xfcc400] Op\_push : O**

**-| [ 1:0xfcc420] Op\_push\_i : "" [MALLOC|STRING|STRCUR]**

**-| [ :0xfcc4a0] Op\_no\_op :**

**-| [ 1:0xfcc480] Op\_push : O**

**-| [ :0xfcc4c0] Op\_concat : [expr\_count = 3] [concat\_flag = 0]**

**-| [ 1:0xfcc3c0] Op\_store\_var : x**

**-| [ 1:0xfcc440] Op\_push\_lhs : X [do\_reference = true]**

**-| [ 1:0xfcc3a0] Op\_postincrement :**

**-| [ 1:0xfcc4e0] Op\_push : x**

**-| [ 1:0xfcc540] Op\_push : o**

**-| [ 1:0xfcc500] Op\_plus :**

**-| [ 1:0xfcc580] Op\_push : o**

**-| [ 1:0xfcc560] Op\_plus :**

**-| [ 1:0xfcc460] Op\_leq :**

**-| [ :0xfcc5c0] Op\_jmp\_false : [target\_jmp = 0xfcc5e0]**

**-| [ 1:0xfcc600] Op\_push\_i : "%c" [MALLOC|STRING|STRCUR]**

**-| [ :0xfcc660] Op\_no\_op :**

**-| [ 1:0xfcc520] Op\_assign\_concat : c**

**-| [ :0xfcc620] Op\_jmp : [target\_jmp = 0xfcc440]**

**-|**

**...**

**-|**

**-| [ 2:0xfcc5a0] Op\_K\_*printf* : [expr\_count = 17] [redir\_type = ""]**

**-| [ :0xfcc140] Op\_no\_op :**

**-| [ :0xfcc1c0] Op\_atexit :**

**-| [ :0xfcc640] Op\_stop :**

**-| [ :0xfcc180] Op\_no\_op :**

**-| [ :0xfcd150] Op\_after\_beginfile :**

**-| [ :0xfcc160] Op\_no\_op :**

**-| [ :0xfcc1a0] Op\_after\_endfile :**

**gawk>**

*help*

*h* 打印所有的 gawk 调试器命令列表，以及它全的简短使用说明。‘help command’打印命令 command 的信息。

*list [- | + | n | filename:n | n–m | function]*

*l [- | + | n | filename:n | n–m | function]*

打印从当源代码或者名为 filename 文件的指定行数（默认为15）。下面是可能的参数：

*- (Minus)* 打印最后打印前的行。

+ 打印最后打印行的后面的行。list 如果不带参数则与这个执行结果相同。

*n* 打印行号 n 为中心周围的行。

*n–m* 打印从 n 到 m 的行。

*filename:n* 打印在源文件 filename 行 n 为中心的行。这个命令会改变当前的源文件。

*function* 打印函数 function 开始行为中心的行。这个命令有可能改变当前的源文件。

*quit*

*q* 退出调试器。调试程序非常有意思，但是在生活中有时你还有其他的责任要尽，而有时我们找到了 Bug，则会再找下一个。如我们早前看到的，如果你正在执行一个程序，调试器会在你输入‘q’ 或者‘quit’时给你警告，以确信你真的想退出。

*trace [on | off]*

打开或者关闭连续打印将要执行的指令，以及这些指令所实现的 awk 代码。默认是关闭的。在这此指令中的大多数“opcodes”都相当地能够自我说明，使用 stepi 与 nexti 的时候，trace 为 on 会使得它们变成熟悉的朋友。（译注：就是会在接下来的显示中会显示很多字节码，看久了，就成为熟悉的朋友了。这是作者的幽默。）

## 14.4 支持

如果 gawk 使用 GNU Readline 库来编译的，则你可以使用这个库的命令完成与历史扩展特性这个优点。下面的完成类型可以使用：

Command completion

命令名字。

Source file name completion

源文件名。相关的命令是 break，clear，list，tbreak 与 until。

Argument completion

命令的非数值参数。相关的命令是 enable 与 info。

Variable name completion

全局变量句以及正在执行程序的当前上下文的函数参数。相关的命令是 display，print，set 与 watch。

## 14.5 限制

我们希望你发现 gawk 的调试器很有用并且享受使用它，但是对于任何的程序，特别是在发布的早期，依然还有有一些限制的。要注意以下几点：

* 在这个时候，调试器不会在你输入一些它不喜欢的东西之后，不会给你错误给出详细的解释。相反，它只是反馈说‘语法错误’。当你确实找到了你的错误所在，所以你确实感觉自己会是一个真正的大牛。
* 如果你一直 dump 出 ，***在第 332 页***，中的 opcodes（或者是你已经熟悉了 gawk 的内部细节），你会意识到 gawk 的很多的内部数据操作，与其他的翻译器一样是在栈中完成的。Op\_push，Op\_poo 以及像 “break and buffer”占 gawk 代码中的大多数。不幸的是，到现在为止，gawk 调试器不允许你检查栈的内容。也就是，表达式的中间结果是在栈中进行求值的，但是却不可以打印。而且，只有在程序的程序中定义的变量才可以打印。当然，要解决这个问题就是在调试阶段使用更加显式的变量，然后再转回到更加隐讳，也许是更加优化的代码。
* 没有办法查看编译的正则表达式的内部过程以查看它是否正确。作为 awk 程序员，你只希望理解/[^[:alnum:][:blank:]]/ 的意义就可以了。
* gawk 调试器设计是用在通过命令行运行的程序上的（以及其所有的参数），如 ，中所描述的。没有办法（目前）来附加或者“插入”到一个运行中的程序。这对主要是用于短程序且快速执行的语言上看起来是合理的。
* gawk 调试器只接收通过 -f 选项提供的源代码。

## 14.6 总结

* 程序很少第一次就可以正确工作。查看 Bug 的过程称为调试，帮你找到 Bug 的程序称为调试器。gawk 有内置的调试器，并且与 GNU 调试器 GDB 工作起来很类似。
* 调试器可以让你一次一个语句执行程序，以检查与改变变量与数组的值，以及许多其他的事情来让你理解你的程序实际所做的（相对于它应该所做的）。
* 与其他的调试器类似，gawk 调试器也是按照栈帧的方式工作的，并且可以你让设置断点（在代码中停止的点）与查看点（当值被改变时停止）。
* 调试器命令相当完备，提供了断点，执行，查看，改变数据等控制命令，还有处理栈，获取信息以及其他任务的命令。
* 如果 GNU Readline 库在 gawk 编译时是可用的，它可以被调试器用于命令行历史与编辑。

# 第十五章 算术运算与gawk 中的任意精度算术运算

本章介绍一些与计算如何进行算术运行相关的基本概念，并定义一些重要的术语。然后继续描述浮点数算术运算，awk 正是用其来做其所有的计算。后面还会有任意精度浮点数算术运行的讨论，这是个只在 gawk 中可用的特性。后续会再讨论任意精度的整数，及对讨论的某些点做一些小结，在这些点上 gawk 与 POSIX 标准不太一致。

提示：gawk 的很多的用户都可以完全地跳过本章。但是如果你想用 gawk 做一些科学计算，这章就非常合适。

## 15.1 计算机算术计算说明

到目前为止，我们已经处理过数字或者字串。但是，最终计算机都是以二进制数字，或者说二进制位表示一切。十进制数可以用10个值来表示：0 到 9。二进制数可以用两个值来表示，0 与 1。通过使用二进制数，计算（以及计算软件）可以表示并操作数值与字符数据。一般来说，更多的位可以用来表示特定的事，以及可以表示的更大的范围的可能数值。

现代计算机支持至少两种，通常是更多的方式来进行算术计算。每种算术计算都可以使用不同的表示（位的组织方式）来代表数值。我们感兴趣的算术类型如下：

十进制算术运算

此类型的算术运行你已经在初级学校中就已经学过了，它们使用笔与纸（或者是计算器）来运算。理论上，在十进制小数点的一边或者两边都可以用任意数值位，而结果总是精确的。

一些现代的系统可以用硬件执行十进制算术运算，但是通常需要特殊的软件库来访问这些指令。也有库用完全用软件来执行十进制算术运算。

尽管有一些用户希望 gawk 可以执行十进制运算，[[100]](#footnote-101)但是 gawk 不支持这样的能力。

整数算法运算

在学校里，整数值被指为“全部”是数值——也就是，数值中没有小数部分，如 1，42 或者 -17。整数的好处是它们完全代表值。缺点就是它们的范围是有限的。

在计算机中，整数值会分成两类：有符号与无符号的。有符号的值可以为正也可以为负，而无符号数值总是大于等于 0。

在计算机系统中，整数算术运算是精确的，但是值的可能范围是有限的。整数算术运算一般要比浮点数算术运算要快。

浮点数算术运算

浮点数值的表示，我们在学校中叫做“实”数（即这些值有小数部分，如 3.1415927）。浮点数值的优点是它可以表示比整数所能表示的数值大得多的范围。缺陷就是这此数值无法精确表示。

现代系统支持硬件实现的浮点运算，但是值是有限的。也有软件库可以使用任意精度的浮点数计算。

POSIX awk 使用双精度浮点数值，这比单精度浮点数可以保存更大范围的数值。gawk 有工具用来执行任意精度的浮点数算术运算，我们后面会进行详细地描述。

计算机处理整数与浮点数表示的不同范围的数值。整数值通常要么是 32 位的，或者是64位的值。单精度浮点数占用 32 位，而双精度浮点数占用 64 位数。浮点值总是有符号的。可能的值范围如表 15.1 所示：

|  |  |  |
| --- | --- | --- |
| 数值表示 | 最小值 | 最大值 |
| 32-bit 有符号整数 | -2，147，483，648 | 2，147，483，647 |
| 32-bit 无符号整数 | 0 | 4，294，967，295 |
| 64-bit 有符号整数 | -9，223，372，036，854，775，808 | 9，223，372，036，854，775，807 |
| 64-bit 无符号整数 | 0 | 18，446，744，073，709，551，615 |
| 单精度浮点数(大约值) | 1.175494-38 | 3.40282338 |
| 双数度浮点数(大约值) | 2.225074-308 | 1.797693308 |

表 15.1 不同的数值表示的值范围

## 15.2 要了解的其他知识

这一章的剩下部分使用大量数值术语。下面是一些非正式定义，这样可以通过这些材料了解，有助于你的工作。

数确度 浮点数计算的精确度是指其与真实值（纸笔来表示的）的接近程序。

错误 计算所以“应该是”的结果与实际的结果之间的差异。应该是尽管最小化这样的错误。

幂 值的数量级，在浮点数值的一些数值位是用来存储幂的。

Inf（无穷） 代表无穷的特殊值。与其他的数或者无穷数之间的操作产生无穷。

非数值 尝试计算时，无法得到实数结果的一个特殊值。在这样的情况下，程序要么会收到一个浮点数异常，或者取得一个 NaN 的结果。IEEE 754 标准建议系统应该返回 NaN。有一些例子如下：

sqrt(-1) 这只有在复数的范围内才是有意义的，但是在实数范围内，结果则是 NaN。

log(-8) -8 不在 log() 的域内，所以结果为 NaN。

正规化 实有效位及其实际存会方式（后面会有说明）。值会被调整使得第一位为 1，因此开始的一位是假想存在，实际并没有存储起来。这可以提供多一位的精度。

精度 用于表示浮点数值的位数。位数越多，则可以表示更多的数值。二进制与十进制数度都是大概的，可根据如下的公式：

**prec = 3.322 · dps**

在这里，prec 表示二进制数度（以位来衡量），dps（十进制位的缩写）是十进制位。

*舍入舍出模式*

如果需要，数值是如何来进行舍入舍出的。后面会提供更多的细节。

系数 浮点数由系数与 10 的幂指数相乘组成。比如，在 1.2345e67，则系数为 1.2345。

稳定性 对于稳定性，来自于维基百科的文档是说：“一个计算可证地不会扩大近似错误称为在数值上是稳定的。”

查看维基百科关于精确性与精度的文章来获取类似术语的更多信息。

在现代的系统上，浮点数硬件使用 IEEE754 标准定义的表示以及操作。IEEE754 的类型的三个标准是 32位单精度，64位双精度以及128位四精度。这个标准也指定了扩展精度的格式以允许更大精度以及更大的指数范围。（awk 只使用 64位双精度格式。）

表 15.2 列出了基本的 IEEE754 二进制格式的精度以及指数域值。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 名字 | 部位数 | 精度 | 最小指数 | 最大指数 |
| 单精度 | 32 | 24 | -126 | +127 |
| 双精度 | 64 | 53 | -1022 | +1023 |
| 四精度 | 128 | 113 | -16382 | +16383 |

表 15.2 基本的 IEEE 格式的值

提示：精度数值包含了隐含的最开始的一位，因此比所给的系数要多一位。

## 15.3 gawk 的任意精度计算

默认的情况下，gawk 使用由其所运行的系统的硬件来提供双精度浮点数值。但是，如果它编译的时候没有指定，gawk 则使用 GNU MPFR 以及 GNU MP（GMP）库来执行任意精度的数值算术运算。你可以查看这样来查看 是否支持 MPFR：

**$ gawk --version**

**-| GNU Awk 4.1.2， API: 1.1 (GNU MPFR 3.1.0-p3， GNU MP 5.0.2)**

**-| Copyright (C) 1989， 1991-2015 Free Software Foundation.**

**...**

（你可能看到与这里显示的不同的版本数。也没有问题，重要的是看 GNU MPFR 与 GNU MP 是否显示在输出中。）

另外，在 PROCINFO 数组中的一些元素可以来提供 MPFR 与 GMP 库一些信息（查看 ， ***在第 158 页***）。

MPFR 库提供了在精度与舍入舍出模式的精确控制，并给出正确的舍入舍出，可重现的，平台无关的结果。使用 -M 命令行参数，所有的浮点算术操作符以及数值函数都可以产生任意想要的由 MPFR 所提供的精度。

有两个预定义的变量，PREC 与 ROUNDMODE 提供了工作精度与舍入舍出模式的控制。精度与舍入舍出模式的设置对于之后的操作是全局的。查看 ，***在第 343 页***，以及 ，***在第 344 页***，来获取更多的信息。

## 15.4 浮点算法：买者自慎！

*数学课太难了。*

*—Teen Talk Barbie， July 1992*

这一节提供了一个更高层上视图，来审视在执行非常多的浮点数计算时所产生问题。[[101]](#footnote-102)这些讨论同样也适用于硬件与任意精度浮点数算术运算。

注意：这里的内容有意地写得非常概括。如果你需要做一些严格的计算算术运行，你首先要进行一些研究，而不仅仅依赖我们所告诉你的。

### 15.4.1 浮点计算是不精确的

二进制浮点数表示以及算术运行是不精确的。像 0.1 这样简单的值没不能精确地由二进制的浮点数来表示，浮点数有有限的精度数表示在计算等级上，或者中间的存储精度的一点点改变都会以改变结果。在任意精度浮点数算术运算时，为了使问题更糟糕， 你可以在开始计算之前设置精度，但是你没有办法保证在最终结果上的十进制的最高有效位的数值。

#### 15.4.1.1 很多的数不能完全表示

所以，在开始写任何代码之间，你要考虑一下你到底想要的是什么，以及到底会发生什么。考虑下面的两个例子中的两个数：

**x = 0.875 # 1/2 + 1/4 + 1/8**

**y = 0.425**

不像数字 y，在 x 中存储的数字可以精确地用二进制来表示，因为它可以写成有限个分数的和，而它他的分母都是 2 的幂次方。当 gawk 从程序源代码中读取浮点数时，它会自动地对于字进行舍入舍出，而不管你的机器支持的是什么样的精度。如果你尝试你用输出格式串“%.17g”打印某个变量的内容时，它不会产生与你所赋给它的一样的值：

**$ gawk ’BEGIN { x = 0.875; y = 0.425**

**>*printf*("%0.17g， %0.17g\n"， x， y) }’**

**-| 0.875， 0.42499999999999999**

通常的错误是很小的，你可能都注意不到。如果你确实需要，你可以指定输出中所需要的精度，由此来产生与输入一样的值。

#### 15.4.1.2 值比较时要小心

由于底层的表示与实际的值有一些偏离，所以比较浮点数来查看两者是否完全相等是一个不好的主意。下面是一个例子，它不会如你所预期的那样执行：

**$ gawk ’BEGIN { print (0.1 + 12.2 == 12.3) }’**

**-| 0**

比较浮点数的值以查看它们是否是在一个很小的范围内的差异才是正确想法（称为 delta（差异）或者容忍度）。你需要确定差异是多小对你来说是重要的。这样的处理代码如下：

**delta = 0.00001 # for example**

**difference = abs(a) - abs(b) # subtract the two values**

**if (difference < delta)**

**# all ok**

**else**

**# not ok**

（我们假设你有一个简单的绝对值函数，名字为 abs()，定义在你的程序的代码的某处。）

#### 15.4.1.3 错误累积

在单个浮点数计算中损失的精度，通常不需要担心。但是，如果你计算的值是一系列的浮点数操作的结果，则错误可以累积并且可以极大地影响计算本身。下面尝试用它一系列的表示来计算 π 的值：

**BEGIN {**

**x = 1.0 / sqrt(3.0)**

**n = 6**

**for (i = 1; i < 30; i++) {**

**n = n \* 2.0**

**x = (sqrt(x \* x + 1) - 1) / x**

***printf*("%.15f\n"， n \* x)**

**}**

**}**

当执行的时候，错误会传递到后面的计算中，导致在进行除0操作时，循环过早地结束：

**$ gawk -f pi.*awk***

**-| 3.215390309173475**

**-| 3.159659942097510**

**-| 3.146086215131467**

**-| 3.142714599645573**

**...**

**-| 3.224515243534819**

**-| 2.791117213058638**

**-| 0.000000000000000**

**error gawk: pi.*awk*:6: fatal: division by zero attempted**

下面是另外一个例子，说明内部表示的不精确导致一个非期望的结果：

**$ gawk ’BEGIN {**

**> for (d = 1.1; d <= 1.5; d += 0.1) # loop five times (?)**

**> i++**

**> print i**

**> }’**

**-| 4**

### 15.4.2 获取你需要的精度

任意精度的算术运算可以得到精确的结果么？这个没有很好的答案。代数的标准规则常常无法用在浮点数的算术运算中。在这些事情中，分配与关联规则都无法完整地应用，操作的等级对于你的计算非常重要。舍入舍出错误，累积精度损失，以及内部运行经常都是麻烦事。

当 gawk 使用机器的双精度来测试表达式 ‘0.1 + 12.2’ 与 ‘12.3’的相等性时，得到的结果是不相等。（查看 ***，在第 340 页***。）你可以通过提供精度来得到你想要的结果，在这个例子中则是要提到 56 位：

**$ gawk -M -v PREC=56 ’BEGIN { print (0.1 + 12.2 == 12.3) }’**

**-| 1**

如果添加多一个位不错，那么添加多一位是不是更好？下面是当你使用更大的 PREC 时所产生的结果：

**$ gawk -M -v PREC=201 ’BEGIN { print (0.1 + 12.2 == 12.3) }’**

**-| 0**

这即不是 gawk 的也不是 MPFR 库的 Bug。大家很容易忘记有限的位数用来存储一个值，通常都在进行了相应的舍入舍出后都是一个相似值。对于测试相等性，只有在所两个操作数中所有等都相同时才会成功。因为在以特定的精度与有效的舍入舍出模式进行浮点数运行后这并不一定要必须的，直接进行相等性的比较是不会成功的。相反，比较两个数以查看它们是否相互之间是在所希望的差异区间内才是正解。

在应用中，15或者更少的十进制位就够了。硬件的双精度算术运算是合适的，并且也很快速。但是你要记住，每一个浮点数操作都会导致一个新的舍入舍出错误，进而导致悲剧性的结果，这正如早先的尝试计算π的值所说明的一样。在这个情况里，额外的精度可以极大地提供计算的稳定性与精确度。

另外，你需要了解，在浮点数算术运算中，重复加并不一定等于乘。这个例子在 ***，在第 341 页***：

**$ gawk ’BEGIN {**

**> for (d = 1.1; d <= 1.5; d += 0.1) # loop five times (?)**

**> i++**

**> print i**

**> }’**

**-| 4**

通过选择任意大数值的 PREC，你可能会，也可能不会成功地取得正确的结果。在这种情况下，立即重组问题通常才是正确的方法。

### 15.4.3 增加几位精度然后再舍入转出

除了任意精度的浮点数算术运算，你所需要的常常是调整你的逻辑，或者是计算的不同等级。早先的例子中，π 计算的稳定性与精度可以通过下面的代数变形可以得到提高：

**(sqrt(x \* x + 1) - 1) / x ≡ x / (sqrt(x \* x + 1) + 1)**

通过这样的变化，在 30 个迭代后，程序汇集到 π 的结果为：

**$ gawk -f pi2.*awk***

**-| 3.215390309173473**

**-| 3.159659942097501**

**-| 3.146086215131436**

**-| 3.142714599645370**

**-| 3.141873049979825**

**...**

**-| 3.141592653589797**

**-| 3.141592653589797**

### 15.4.4 设置精度

gawk 使用全局性的工作精度 ，它不会为单个的数值跟踪精度与精确度。执行算术运算或者调用内置函数来对结果进行舍入舍出都以当的精度为准。默认的工作精度是 53 位，你可以通过修改变量 PREC 来改变。你也可以设置为预定义的大小写有关的值串，如表 15.3 所列，这些用来对 IEEE 754二进制标准进行模拟。

|  |  |
| --- | --- |
| PREC IEEE | 754 二进制格式 |
| "half" | 16 位半精度 |
| "single" | 基本的 32 位单精度 |
| "double" | 基本的 64 位双精度 |
| "quad" | 基本的 128 位四精度 |
| "oct" | 256 位八精度 |

表 15.3 PREC 的预定义精度串

下面的例子说明了改变算术操作符的精度的效果：

**$ gawk -M -v PREC=100 ’BEGIN { x = 1.0e-400; print x + 0**

**> PREC = "double"; print x + 0 }’**

**-| 1e-400**

**-| 0**

注意：要小心浮点数常量！当从程序源代码中读取浮点数常量时，gawk 会使用默认的精度（即 C 的双精度），除非通过命令行赋值来改变 PREC 的值，将其内部存为 MPFR 数值。在程序源代码通过使用 PREC 来改变精度 不会影响常量的精度。

如果需要在比默认的精度还要高的精度上表示一个浮点数，但又不能使用命令行对 PREC 进行赋值，你要么将常数指定会字串，要么只要可能，使其成为有理数， 下面的例子说明了不同的方式下打印浮点数常量时的不同：

**$ gawk -M ’BEGIN { PREC = 113; *printf*("%0.25f\n"， 0.1) }’**

**-| 0.1000000000000000055511151**

**$ gawk -M -v PREC=113 ’BEGIN { *printf*("%0.25f\n"， 0.1) }’**

**-| 0.1000000000000000000000000**

**$ gawk -M ’BEGIN { PREC = 113; *printf*("%0.25f\n"， "0.1") }’**

**-| 0.1000000000000000000000000**

**$ gawk -M ’BEGIN { PREC = 113; *printf*("%0.25f\n"， 1/10) }’**

**-| 0.1000000000000000000000000**

### 15.4.5 设置舍入转出模式

ROUNDMODE 变量提供了在程序层控制舍入舍出模式的方法。ROUNDMODE 与 IEEE 的舍入舍出的比较显示在表 15.4 中。

|  |  |  |
| --- | --- | --- |
| 舍入舍出模式 | IEEE 名 | ROUNDMODE |
| 向偶数舍入舍出到最近 | roundTiesToEven | "N" or "n" |
| 向正无穷舍入舍出 | roundTowardPositive | "U" or "u" |
| 向负无穷舍入舍出 | roundTowardNegative | "D" or "d" |
| 向零舍入舍出 | roundTowardZero | "Z" or "z" |
| 背身零舍入舍出到最近 | roundTiesToAway | "A" or "a" |

表 15.4：gawk 的舍入舍出模式

ROUNDMODE 的默认值为 “N”，这选择的是 IEEE754 的舍入舍出模式为 roundTiesToEven。在表 15.4 中，值 “A” 选择的是 roundTiesToAway。这只有顺你的 MPFR 库支持时才可用，否则设置 ROUNDMODE 为 A 是没有效果的。

默认的模式 roundTiesToEven 是使用得最多的，但是最不直观。这个方法对大多数的值的处理都是很明显的，就是将它们向上或者向下到最近的数。例如，将 1.132 舍入舍出为两位数，会产生 1.13，而对 1.157 舍入舍出产生 1.16。

但是，其值刚好是一半的时候，对其进行舍入舍出时，事情处理方式就不会像你在学校时所学的一样工作。在这种情况下，数字被舍入舍出到最近的偶数位。所以，对 0.125 舍入舍出为两位数，则会产生 0.12，但是对数值 0.6875 舍入舍出到三位则会向上舍入为 0.688。你可能在使用 printf 来格式化浮点数时已经遇到过这样的舍入舍出模式。例如：

**BEGIN {**

**x = -4.5**

**for (i = 1; i < 10; i++) {**

**x += 1.0**

***printf*("%4.1f => %2.0f\n"， x， x)**

**}**

**}**

在作者的系统上，会产生下面的输出：[[102]](#footnote-103)

**-3.5 => -4**

**-2.5=> -2**

**-1.5 => -2**

**-0.5 => 0**

**0.5 => 0**

**1.5 => 2**

**2.5 => 2**

**3.5 => 4**

**4.5 => 4**

roundTiesToEven 背后的理论是，对确定的半值舍入舍出或多或少向上或者向下是均等地分布的。这样可以使得舍入输出导致的损失可以自己补偿。这对于 IEEE754 的计算函数与操作符是默认的舍入舍出模式。

其他的舍入舍出模式很少使用。向正无穷舍入舍出（roundTowardPositive）以及向负无穷舍入舍出（roundTowardNegative）常常用于实现间隔算术，在这样的计算中，你调整舍入舍出模式来计算输出范围的上下边沿。roundTowardZero 模式可用于将浮点数转换了整数。roundTiesToAway 这个舍入输出模式将结果舍入舍出到最近的数，如果出现的绑定的情况，则选择一个较大幅度的值。

有一些数值分析师会告诉你，你的舍入舍出风格的选择会极大地影响最后的输出，并且建议你直到最后输出时才确定舍入舍出模式。相反地，你常常可以避免舍入舍出错误，需要的只是在一开始将初始精度设置比最终所需要的大得多的精度，这样累积的舍入舍出错误不会影响结果。如果你怀疑你计算的结果对于舍入舍出错误是很敏感的，需要你改变舍入舍出模式来看现输出中明显的差异来进行确认。

## 15.5 gawk 中的任意精度整数计算

当指定了 -M 选项，gawk 会使用 GMP 任意精度整数的执行所有的整数算术。在源代码中或者数据文件中任意的看起来像整数值都存成任意精度的整数。整数的大小只受限于可用的内存。例如，下面会计算 ，这个结果已经超过了一般的双精度浮点值的限制：

**$ gawk -M ’BEGIN {**

**> x = 5^4^3^2**

**> print "number of digits ="， length(x)**

**> print substr(x， 1， 20)， "..."， substr(x， length(x) - 19， 20)**

**> }’**

**-| number of digits = 183231**

**-| 62060698786608744707 ... 92256259918212890625**

如果你使用任意精度的浮点值来计算相同的值，为了正确的输出所需要的精度（使用公式 prec-3.322·dps）则会是 3.322 · 183231，或者说是 608693。

来自于一个整数与一个浮点数值的算术操作的结果，是一个与当前工作数度相同的一个浮点数。下面的程序用迭代来计算 Sylvester 序列[[103]](#footnote-104)的第八个项：

**$ gawk -M ’BEGIN {**

**> s = 2.0**

**> for (i = 1; i <= 7; i++)**

**> s = s \* (s - 1) + 1**

**> print s**

**> }’**

**-| 113423713055421845118910464**

输出不同于实际的数值，即 113，423，713，055，421，844，361，000，443，这里因为默认的精度为 53 位，不足以精确地表示清点结果。你要么提供高精（在这个例子中，100位是够了的），或者将浮点数常数‘2.0’替换为一个整数，以全部使用整数算术运行来计算正确有的输出。

有时候，gawk 必须隐式地将任意精度的整数转换为任意粒度的浮点数。这主要是因为，当一个操作或者函数需要时，MPFR 库并不总是提供相关的接口来处理任意精度的整数，或者混合模式的数字。在这样的例子中，精度会被设置成要精度转换时的最小的值，而工作精度则不用于这个目的。如果这不是你想要的，你可以用个小诡计并将整数先转换为浮点数，如：

**gawk -M ’BEGIN { n = 13; print (n + 0.0) % 2.0 }’**

你可以通过指定数值为浮点数来避免这样的问题：

**gawk -M ’BEGIN { n = 13.0; print n % 2.0 }’**

注意，对于这个特殊的例子，最好是使用下面的试：

**gawk -M ’BEGIN { n = 13; print n % 2 }’**

## 15.6 标准与既存实践

历史上，awk 会在需要的时候，将所有的看起业是非数值的字串转换为数值 0。而且，语言的最开始的定义与最初的 POSIX 标准规定，awk 只理解十进制（基数为10），而不是八进制（以8为基数）以及十六进制数（以16为基数）。

2001 的语言的变化与 2004 年的 POSIX 标准都可以解释为，awk 会可支持额外的特性，这些特性是：

* 以十六进制形成指定的浮点数的解析（如，0xDEADBEEF）。（提示：是数据值，而不是源代码常量。）
* 支持 IEEE754 特殊的浮点数值“not a number”（NaA），正无穷与负无穷。特别是，这些值的格式由 ISO1999 的 C 标准指定，它们不区别大小写，可允许依赖于实现的在‘nan’后的额外字符，也允许 ‘inf’ 或者‘infinity’。
* 第一个问题是这两个相比于历史实践都有明确的改变：
* gawk 的维护者感觉，支持十六进制的浮点数，相当的丑，从来没有被最初的实现者接受为语言的一部分。
* 完全允许字母串有一个有效的数值也与历史实践有着严重的分离。

第二个问题是 gawk 的维护者觉得，对于标准的这种解释，需要大量的“语言立法”来达到第一个目标，这甚至也不被标准的开发者预料到。也就是说，“我们看到了你怎么来的，但是我们不认为那就是你想要去的地方。”

由于发现了这样的问题，作为提供一个与早期标准版本兼容的尝试，在 2008 年的 POSIX 标准添加了一个显式的话来允许，但是不是必须，awk 支持十六进制的浮点数值以及针对 “not a number”下无穷的特殊值。

尽管 gawk 的维护者仍然觉得提供这样的特性是不被建议的，但是，在支持 IEEE 浮点数的系统上，提供某些途径来支持 NaN 与无穷值却是合理的。gawk 的解决方案如下：

* 如果有 --posix 命令行选项，则 gawk 会松手不管。字串值会被直接传递绘画系统的库函数 strtod() 函数，如果成功地返回了数值值，那么它就会被当做数值使用。[[104]](#footnote-105)根据定义，结果在不同的系统上是不可移植的。而且也有一些小的令人惊奇的行为：

**$ echo nanny | gawk --posix ’{ print $1 + 0 }’**

**-| nan**

**$ echo 0xDeadBeef | gawk --posix ’{ print $1 + 0 }’**

**-| 3735928559**

* 如果没有 --posix 选项，gawk 会将后面的字串值 ‘+inf’， ‘-inf’， ‘+nan’ 与‘-nan’作特别的解释，并产生相应的特殊的数值。前导的符号会被当成发送绘画 gawk（以及用户）的信号说，这确实是个数值。十六进制浮点数是不受支持的（除非你也使用了 --non-decimal-data 选项，但我们不建议）。例如：

**$ echo nanny | gawk ’{ print $1 + 0 }’**

**-| 0**

**$ echo +nan | gawk ’{ print $1 + 0 }’**

**-| nan**

**$ echo 0xDeadBeef | gawk ’{ print $1 + 0 }’**

**-| 0**

gawk 对这四个值不区分大小写。因此 ‘+nan’ 与 ‘+NaN’是一样的。

## 15.7总结

* 大多数的计算机算术运算是通过整数或者浮点数来完成的。标准的 awk 使用双精度浮点数数值。
* 在 1990 年代早期，Barbie 不正确地说过，“数学真难！”虽然数学不难，但是浮点数运算并不像“纸笔”数学一样，而必须仔细处理：
  + 并不是所有的数都可以被精确表示。
  + 值的比较应该使用差别，而不是直接使用 ‘==’ 与 ‘!=’。
  + 错误会累积。
  + 操作并不总是关联与分布的。
* 提供精度有用，但并不总是成能药。
* 提升精度然后舍入舍出到一个需要的数值位以产生合理的结果。
* 使用 -M（或者 --bignum）来使用 MPFR 算术。使用 PREC 来设置精度位，以及设置 ROUNDMODE 来设置 IEEE754 的舍入舍出模式。
* 通过 -M，gawk 会使用 GMP 库执行任意精度的整数算术运算。这比 MPFR 执行相同的计算会更快，在空间上更高效。
* gawk 在处理浮点数上有几个方面与 POSIX 标准不同。有必要注意。
* 总体上看，没有必要过度地怀疑浮点数运算的结果。要记住的教训是浮点数运算要比纸笔上进行运算要更复杂。为了充分利用浮点数运算的优势，你需要知道它的限制来使用它们来工作。对于大多数的浮点算术运算，如果你简单地舍入舍出来显示你的结果到正确的十进制有效位，你都可以得到想要的结果。
* 作为一般建议，避免以隐式的比实际更高精度来表示一个数据。

# 第十六章 编写 gawk 扩展

使用动态装载的库，可以用 C 或者 C++ 为 gawk 编写新的功能。这样的工具在支持 C 的 dlopen() 与 dlsym() 函数的系统上是可用的。本章会描述如何用 C 与 C++ 的代码来创建扩展。如果你不了解 C 编程，你可以跳过这一章，你可以查看随 gawk 一起发布的扩展文件（查看 ， ***在第 391 页***），也可了解 gawkextlib 项目的信息（查看 ，***在第 400 页***）。例子扩展会与 gawk 一起进行构建并安装。

提示：当使用 --sandbox 选项时，扩展会被禁用（查看 ， ***在第 35 页***）。

## 16.1 介绍

扩展（有时也称为插件）是一块外部编译的代码，gawk 可以在运行时装载来提供额外的功能，它们建立在本章中所描述的内建能力之上。

扩展非常有用，因此它们可以让你（这毫无疑问）来扩展 gawk 的功能。例如，它们可以提供访问系统调用的能力（比如 chdir() 来更改目录）以及其他的可用的 C 库例程。如大多数的软件“天空才是限制”，如果你想到某些事情，你可以通过 C 或者 C++ 代码来实现，你就可以写一个扩展来实现他。

扩展通过 C 或者 C++ 写成，并利用为此目的而向 gawk 开发者提供的 API 来实现。本章中的其他部分解释 API 工具以及如何来使用它们，并提供了一个小的扩展例子。另外，例子扩展程序也包含在了 gawk 的发布版本中，并对 gawkextlib 进行描述。查看 ，***在第 445 页***，来获取扩展机制的目标与设计的更多讨论。

## 16.2 扩展许可

每个动态的扩展必须在与 GNU GPL（查看 ***，在第 465 页***）兼容的许可下发布。

为了让扩展告知 gawk 它是有适当的许可的，扩展必须定义一个全局符号 plugin\_is\_GPL\_compatible。如果这个符号不存在，在扩展被装载时，gawk 会发送一个致命错误，并退出。

符号的声明类型为 int。它没有必要一定要在某个段中。代码仅仅是断言符号在全局范围内存在即可。像下面这样就足够了：

**int plugin\_is\_GPL\_compatible;**

## 16.3 在高层级中是它是如何工作的

gawk 与 扩展之间的通信是双向的。首先，当扩展被装载，gawk 会向它传递一个结构的指针，结构中都是函数指针。如图 16.1 所示：

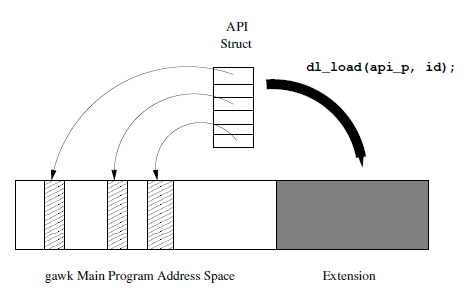


图 16.1：装载扩展

扩展通过这些指针函数来调用 gawk 中的函数，并且是在运行时调用，而无须（在连接时）访问 gawk 的符号。这些函数指针中的一个是用来“注册”新功能的函数。如图 16.2 所示：

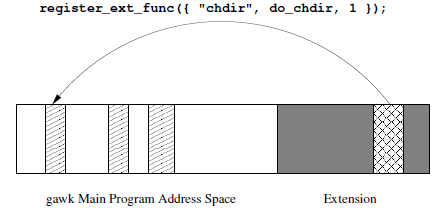


图 16.2：注意新功能

在另一个方向，扩展注册它的新功能是向 gawk 传递函数的指针，这个函数提供了新的特性（比如 do\_chdir()）。gawk 将这个函数指针与一个名字关联，然后使用定义好的调用规范来调用它。如在图 16.3 所示：

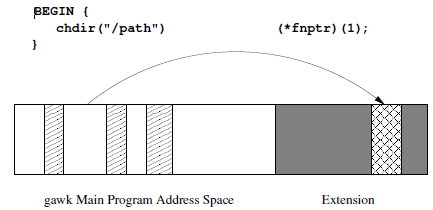


图 16.3：调用新函数

接下来 do\_xxx***()*** 函数使用 API 结构中的函数指针来实现它的功能，如更新变量或者数组，打印信息，设置 ERRNO 等等。

方便的宏可以使用通过函数指针调用看起来像常规的函数调用一样，这样扩展代码非常好读，也很好理解。

尽管这听起来有一些复杂，而结果是扩展的代码编写与阅读都非常直接。你可以查看例子 ***extensionfilefuncs.c*** （查看， ***在第 382 页***）以及 testext.c 代码来测试 APIs。

* API 提供了访问 gawk 的 do\_xxx 值的方式，以此来反应命令行参数，如 do\_lint，do\_profiling 等等（查看 ， ***在第 379 页***）。而下面提供相关信息：扩展不可以改变它们在 gawk 中的值。另外，尝试绘给它们赋值会产生一个编译时错误。
* API 也提供了主次版本号，这样扩展可以来检查装载扩展的 gawk 的版本是否支持其被编译时的所需要的功能。（版本不匹配不应该发生，但是我们所有人都知道事情怎么发生的。）查看 ，***在第 379 页***，来获取相关细节。

## 16.4 API 描述

C 或者 C++ 所写的扩展必须包含头文件 gawkapi.h，在这个文件中声明函数，并且定义了与 gawk 进行通信的数组类型。这（相当大的）节，详细地描述了这些 API。

### 16.4.1 介绍

访问 gawk 的工具是通过调用传递到你的扩展的函数指针来完成的。

API 函数指针提供了下面类型的操作：

* 分配，重新分配以及释放内存。
* 注册函数，有如下：
  + 扩展函数
  + 退出回调函数。
  + 版本字串
  + 输入分析器
  + 输出包装器。
  + 双路处理器

所有的这些都会在本章中的剩余部进行详细的讨论。

* 打印致命，告警以及“lint”警告信息。
* 更新 ERRNO，或者重置它。
* 访问参数，包括将未定义的参数转换为数组。
* 符号表访问：返回全局变量，合建或者更改全局变量。
* 创建，释放缓存值，这提供了一种高效访问多个变量的方式，可能极大地提供性能。
* 数组操作：
  + 返回，添加，删除以及修改元素。
  + 获取数组中的元素个数。
  + 创建新数组。
  + 清除数组。
  + 扁平化数组，使用以使用简单的 C 风格的循环来迭代所有的索引以及元素。

使用 API 的一些点：

* 下面的类型，宏，以及/或者函数可以在 gawkapi.h 文件参考。为了正确使用，你必须在包含 gawkapi.h 前包含相应的标准头文件。

|  |  |
| --- | --- |
| C entity | Header file |
| EOF | <***stdio.h***> |
| Values for errno | <***errno.h***> |
| FILE | <***stdio.h***> |
| NULL | <***stddef.h***> |
| ***memcpy()*** | <***string.h***> |
| ***memset()*** | <***string.h***> |
| size\_t | <***sys/types.h***> |
| struct stat | <***sys/stat.h***> |

由于考虑到可移植性，特别是那些没有完全与标准兼容的系统，你需要以正确的方式包含正确的文件。为了保证 gawkapi.h 的简单清楚，这个要求是很必要的，这样不会成为可移植性的大杂烩，而这会在部分的 gawk 源代码中看到。

* gawkapi.h 文件可以被包含多次，而不会产生不好的结果。只是这么做是个很不好的编程实践。
* 尽管 API 只使用 ISO C 90 的特性，但是有一个例外，即“构造”函数使用 inline 关键词。如果你的编译器不支持这个关键词，你要么在命令行中加入 ‘-Dinline=''’，或者使用 GNU 的 Autotools 并在你的扩展中包含一个 config.h 文件。
* 所有的在 gawk 填充的指针指向的内存，都由 gawk 管理，应该被你的扩展看成是只读的。从扩展中传递到gawk 中的字串内存，必须使用 gawk\_alloc()，gawk\_calloc() 或者 gawk\_realloc() 调用获取，并从那以后，由 gawk 进行管理。
* API 定义了几个简单的结构用来影射 awk 看到的值。值可以是double类型，或者字串，或者是数组（如多维数组，或者创建新数组时）。字串值通过指针与长度来管理，因此内嵌的 nul 字符也可以使用。

提示：从目的上讲，字串使用当前的多字节编码来维护（如 LC\_xxx 环境变量定义的）而不是使用宽字符。这个与 gawk 内部存储字串是匹配的，字符也可能是这么从文件读取或者输出到文件的。

* 当收到一个值（如参数或者全局变量，或者数组元素），扩展需要请示一个特定的类型（数值，字串，标量，值的Cookie，数组，或者“未定义”）。当请求为“未定义”时，返回的值是会有真正的底层类型的。

但是，如果请求的类型与实际的类型不匹配，读取函数会返回 “false”，并在 type 类型里填上值实际的类型，这样扩展可以打印错误的信息（比如 “scalar passed where array expected”（需要数组时传递的是标量类型））。

档可以使用函数指针来调用 API 函数，但是接品没有这么简单。为了使扩展代码看起来像常规的代码，gawkapi.h 头文件中定义了几个宏，你可以在你的代码中使用。这一节会在你使用函数时来呈现这些宏。

### 16.4.2 通用目的的数据类型

*我是真的喜欢/讨厌与联合之间的关系。*

*—Arnold Robbins*

*这是在关于联合的事情：编译器会对事进行周密安排，这样可以协调对它的爱与恨。*

*—Chet Ramey*

扩展的 API 定义了很多简单的类型以及结构来用于通用目的。另外，很多特有的数据结构会在后面的章节中进行介绍，包含使用他们的函数。

***typedef void \*awk\_ext\_id\_t;***

这个类型的值是在扩展被载入的时候从 gawk 中返回的。这个值必须被作为每个 API 的函数的第一个参数回传给 gawk。

***#define awk\_const ...***

这个宏会在编译扩展时展开成为 ‘const’，但在编译 gawk 本身时，则为空。这使得 API 数组结构中的特定域从扩展中是不可写的，而允许 gawk 在需要的时候使用它们。

***typedef enum awk\_bool {***

***awk\_false = 0，***

***awk\_true***

***} awk\_bool\_t;***

简单的布尔类型。

***typedef struct awk\_string {***

***char \*str; /\* data \*/***

***size\_t len; /\* length thereof， in chars \*/***

***} awk\_string\_t;***

这表示一个可变的字串。如果有值的话，gawk 持有指向它的内存。否则，它会自己持有其指向的内存。这些内存必须来自于 gawk\_***malloc()***， gawk\_***calloc()***，或者gawk\_***realloc()*** 函数。

如早前所提，字串使用当前的多字节编码来保存。

***typedef enum {***

***AWK\_UNDEFINED，***

***AWK\_NUMBER，***

***AWK\_STRING，***

***AWK\_ARRAY，***

***AWK\_SCALAR， /\* opaque access to a variable \*/***

***AWK\_VALUE\_COOKIE /\* for updating a previously created value \*/***

***} awk\_valtype\_t;***

这个枚举指示了数据的类型。它会在后面的结构中使用。

***typedef struct awk\_value {***

***awk\_valtype\_t val\_type;***

***union {***

***awk\_string\_t s;***

***double d;***

***awk\_array\_t a;***

***awk\_scalar\_t scl;***

***awk\_value\_cookie\_t vc;***

***} u;***

***} awk\_value\_t;***

一个 “awk 值”。val\_type 成员指示了联合体存有的值的类型，每个成员都是特定的类型。

***#define str\_value u.s***

***#define num\_value u.d***

***#define array\_cookie u.a***

***#define scalar\_cookie u.scl***

***#define value\_cookie u.vc***

使用这些宏来访问 awk\_value\_t 的域会使用代码更阅读。

***typedef void \*awk\_scalar\_t;***

标量可以被表示成一个透明类型。这些值从 gawk 中获取，然后再回传给 gawk。这在本列表下面的会以有更一般的讨论。更多的细节请参考 ， ***在第 368 页***。

***typedef void \*awk\_value\_cookie\_t;***

一个 “值 Cookie”表示一个缓存值的透明类型。这会下列表的后面进行更一般的讨论，而更多的细节请参考 ， ***在第 370 页***。

在 awk 中的标量要么是数值，要么是字串。awk\_value\_t 结构代表一个值。val\_type 成员指示联合中的具体值的类型。

表示一个数值很简单——API 使用 C double。字串则需要一点工作。因此 gawk 允许内嵌的 nul 字节在字串中。一个字串必须使用包含指针与长度一对值来表示。这个类型是 awk\_string\_t 类型。

标识符（如全局变量的名字）可以与标题值或者与数组进行关联。另外，gawk 提供了真正的数组之数类型，这样指定的数组元素也可以是一个数组。对于数组的讨论，详细内容要到 ， ***在第 371 页***。

早前所列的宏使得更容易使用联合中的元素，就好像它就是结构中域一样，这在 C 中是一个通用的编程技巧。这些代码也更容易编写与阅读，但是你有责任确保 val\_type 成员正确地反映了 awk\_value\_t 结构中的值的类型。

从概念上来说，联合中的头三个成员（number，string 与 array）是与 awk 一起使用时所有需要的类型。但是，因为 API 提供的用来访问与更改全局标量值的例程只可以使用变量名，因此会有性能损耗：gawk 必须在访问或者更改时找到这个变量。这会成为一个真正的问题，而不仅是理论上的。

所以，如果你知道你的扩展会花相当长的时候来读取/或者改变一个或者多个标量的值。你就可以为那个变量获取一个标量 Cookie 对象，[[105]](#footnote-106)然后使用这个 Cookie 来获取变量的值，或者改变变量的值。awk\_scarlar\_t 类型持有一个标量 Cookie，而 scalar\_cookie 宏提供了访问在 ***awk***\_value\_t 结构中那个类型值的方式。假设有一个标量 Cookie，gawk 可以直接返回或者修改其值，而不需要首先查找它。

***awk***\_value\_cookie\_t 类型与 value\_cookie 宏类似。如果你知道你希望使用相同的数值与字串类型来用于一个或者多个变量，你可以一次创建值，然后为它保留值 Cookie，然后在希望在设置变量的任何时候，将这个 Cookie 传递过去产。这个可以在运行的 gawk 中节约存储，并且压缩创建值的时间。

### 16.4.3 内存分配函数与相应的简化宏

API 提供了很多的内存分配函数用来进行内存分配，并可将内存传递绘画 gawk，以及一些用于方便目的的宏。这个小节将它们的函数原型呈现，扩展可以通过这种方式来使用它们：

***void \*gawk\_malloc(size\_t size);***

调用正确的 malloc() 版本来分配存储，并可以传递给 gawk。

***void \*gawk\_calloc(size\_t nmemb， size\_t size);***

调用正确的 calloc() 版本来分配内存，并可以传递给 gawk。

***void \*gawk\_realloc(void \*ptr， size\_t size);***

调用正确的 ***realloc*** () 版本来分配内存，并可以传递给 gawk。

***void gawk\_free(void \*ptr);***

调用正确的 free() 版本来释放由 gawk\_***malloc()***， gawk\_***calloc()***，或者gawk\_***realloc()*** 分配的内存。

API 要提供这些函数，是因为要编译的扩展有可能连接的是不同版本的 C 库，而不是 gawk 执行体所使用的 C 库。如果 gawk 使用的是它自己版本 free() 来释放不相同的版本 malloc() 版本分配的内存时，就可能导致未预期的行为。

有两个方便的宏可以用来使用 gawk\_***malloc()*** 与 gawk\_***realloc()*** 来进行内存分配。如果分配失败，会导致 gawk 会以致命错误退出。使用它们就像使用一个不会返回值的过程一样。

***#define emalloc(pointer， type， size， message) ...***

这个宏的参数如下：

pointer ponter 变量指向被分配的内存。

type pointer 变量的类型。这个用来创建一个调用 gawk\_malloc() 的转换。

size 要分配的字节总大小。

message 在错误信息前附加的消息。典型的是使用这个宏的函数名。

例如，你可以像这样来分配一个字串值：

***awk*\_value\_t result;**

**char \*message;**

**const char greet[] = "Don’t Panic!";**

**emalloc(message， char \*， sizeof(greet)， "myfunc");**

**strcpy(message， greet);**

**make\_malloced\_string(message， strlen(message)， & result);**

***#define erealloc(pointer， type， size， message) ...***

这个与 emalloc() 类似，但是调用的是 gawk\_realloc() 而不是 gawk\_malloc()。参数与 emalloc() 宏一样。

### 16.4.4 构造函数

API 提供了多个构造函数来创建字串或者数值，以及多个相应的宏。这个章节呈现所有函数的原型，扩展代码可以使用这些原型来调用它们：

***static inline awk\_value\_t \****

***make\_const\_string(const char \*string， size\_t length， awk\_value\_t \*result);***

这个函数在通过 result 指向的 awk\_value\_t 变量中创建一个字串值。函数需要要的 string 是一个 C 字串常量（或者其他的字串数据），并自动创建一个在 result 中的数据复本。然后返回 result。

***static inline awk\_value\_t \****

***make\_malloced\_string(const char \*string， size\_t length， awk\_value\_t \*result);***

这个函数在由 result 指向的 awk\_value\_t 变量中创建一个字串类型。函数需要的 string 是一个 ‘char\*’，指向的之前已经从 gawk\_malloc()，gawk\_calloc() 或者 gawk\_realloc() 中获取的数据。这里的想法是数值会直接传递给 gawk，将对其负责。函数返回 result。

***static inline awk\_value\_t \****

***make\_null\_string(awk\_value\_t \*result);***

这个特别函数，在一个由 result 指向的 awk\_value\_t 变量由中创建一个空串。函数返回 result。

***static inline awk\_value\_t \****

***make\_number(double num， awk\_value\_t \*result);***

这个函数简单地在由 result 指向的 awk\_value\_t 变量中创建一个数值。

### 16.4.5 注册函数

本节描述用于注册部分扩展到 gawk 中的 API 函数。

#### 16.4.5.1 注册扩展函数

扩展函数是通过下面的记录下进行描述的：

**typedef struct awk\_ext\_func {**

**const char \*name;**

**awk\_value\_t \*(\*function)(int num\_actual\_args， awk\_value\_t \*result);**

**size\_t num\_expected\_args;**

**} awk\_ext\_func\_t;**

域的含义如下：

***const char \*name;***

新功能的名字。awk 层的代码通过这个名字来调用函数。这是一个常规的 C 字串。

函数名必须符合 awk 标识符的规则。也就是，它们必须是以英文字母或者下划线开始，后面可以跟任何的字母，数字与下划线。函数名的字母的大小写是有关的。

***awk\_value\_t \*(\*function)(int num\_actual\_args， awk\_value\_t \*result);***

这是一个指向提供了扩展功能的 C 函数的指针。函数必须在 \*result 中填入数字或者字串。gawk 会接管任意字串内存的所有权。如早先所提，字串内存必须来自于 gawk\_***malloc()***， gawk\_***calloc()***， 或 gawk\_***realloc()*** 等函数之一。

num\_actual\_args 参数告知 C 函数，会有多少个实际的函数从调用的 awk 代码中传入。

函数必须返回 result 的值。这主要是为了 gawk 中调用代码使用上的方便。

***size\_t num\_expected\_args;***

这个是函数希望接受参数的个数。每一个扩展函数都可以决定如果参数不是它所希望的个数时所要做的事情。对于真正的 awk 函数，忽略额外的参数也是可以的。

一旦你有了一个表示你的扩展的函数，你就可以使用下面的 API 函数来注册：

***awk\_bool\_t add\_ext\_func(const char \*namespace， const awk\_ext\_func\_t \*func);***

这个函数在成功时返回为 true，否则返回 false。namespace 参数当前没有使用，你可以传递一个空串过去。func 指针是代表你的函数的结构指针，其结构如上所述。

#### 16.4.5.2 注册退出函数

一个退出回调函数是在 gawk 退出时要调用的函数。这样的函数会在你的扩展中有一些清理工作要做时非常有用（如关闭数据库连接以及释放其他资源）。你可以使用下面的函数来注册这样的一个函数到 gawk 中：

***void awk\_atexit(void (\*funcp)(void \*data， int exit\_status)，void \*arg0);***

参数的含义为：

funcp 一个在 gawk 退出之前要调用的函数的指针。数据参数就是 argo 的初始值。exit\_status 参数是 gawk 想传递给 exit 系统调用的状态值。

arg0 一个私有数据的指针，gawk 会保存它，并在调用 funcp 指向的函数时传递给它。

退出回调函数会先进先出（LIFO）的顺序调用，也就是以它们注册到 gawk 中的顺序相反的顺序来调用。

#### 16.4.5.3 注册扩展版本字串

你可以注意一个版本字串来表示你的扩展在 gawk 中的版本与名字，如下：

***void register\_ext\_version(const char \*version);***

将由 version 指向的版本注册到 gawk 中。要注意 gawk 不会复杂 version 字串，因此它不可以被改变。

gawk 当以 -version 选项调用时，会打印所有的已经注册了扩展的版本字串。

#### 16.4.5.4 定制输入分析器

默认的情况下，gawk 从输入中读取文本。它使用 RS 的值来查看记录的结束，并且使用 FS（或者 FIELDWIDTHS 或者 FPAT）来将其分割成域（查看 ， ***在第 63 页***）。另外，gawk 还会设置 RT 的值（查看 ， ***在第 155 页***）。

如果你想，你可以提供自己的定制的输入分析器。一个输入分析器的工作就是给 gawk 的记录处理代码返回一条记录，以及用于 RT 的数据的值与长度的指示符，如果有的话。

为了提供输入分析器，你必须首先提供两个函数（在这里 XXX 是你的扩展的前缀）：

***awk\_bool\_t XXX\_can\_take\_file(const awk\_input\_buf\_t \*iobuf);***

这个函数检查在 iobuf 中的信息（我们后面会讨论）。基于里面的信息，可以确定输入分析器是否用于这个文件。如果可以，则返回 true。否则返回 false。这个函数不应该改变 gawk 中的任何状态（比如变量的值等）。

***awk\_bool\_t XXX\_take\_control\_of(awk\_input\_buf\_t \*iobuf);***

当 gawk 决定从输入分析器中接手文件的控制权的时候，它会调用这个函数。这个函数接着必须将特定的域填充到 ***awk***\_input\_buf\_t 结构中，并保证特定的条件为 true。函数应当返回 true。如果出现了某种错误，函数不可以再填充任何域，并且返回 false，之后，gawk 不会使用这个输入分析器。细节将在后面呈现。

你的扩展应该要将这些函数包到 ***awk***\_input\_parser\_t 中，这个结构如下：

**typedef struct *awk*\_input\_parser {**

**const char \*name; /\* name of parser \*/**

***awk*\_bool\_t (\*can\_take\_file)(const *awk*\_input\_buf\_t \*iobuf);**

***awk*\_bool\_t (\*take\_control\_of)(*awk*\_input\_buf\_t \*iobuf);**

***awk*\_const struct *awk*\_input\_parser \**awk*\_const next; /\* for gawk \*/**

**} *awk*\_input\_parser\_t;**

这些域为：

***const char \*name;***

输入分析器的名字。这是一个常规的 C 字串。

***awk\_bool\_t (\*can\_take\_file)(const awk\_input\_buf\_t \*iobuf);***

指向 XXX\_can\_take\_***file()*** 函数的指针。

***awk\_bool\_t (\*take\_control\_of)(awk\_input\_buf\_t \*iobuf);***

XXX\_take\_control\_***of()*** 函数的指针。

***awk\_const struct input\_parser \*awk\_const next;***

这个由 gawk 来使用，因此它被标志为 awk\_const，这样扩展就不能够修改它。

实现步骤：

1. 创建一个静态的 ***awk***\_input\_parser\_t 变量，并且正确地初始化它。
2. 当你的扩展被载入时，使用 register\_input\_***parser()*** API 函数注册你的输入分析器到 gawk 中。

***awk***\_input\_buf\_t 结构如下：

**typedef struct *awk*\_input {**

**const char \*name; /\* filename \*/**

**int fd; /\* file descriptor \*/**

**#define INVALID\_HANDLE (-1)**

**void \*opaque; /\* private data for input parsers \*/**

**int (\*get\_record)(char \*\*out， struct *awk*\_input \*iobuf，**

**int \*errcode， char \*\*rt\_start， size\_t \*rt\_len);**

**ssize\_t (\*read\_func)();**

**void (\*close\_func)(struct *awk*\_input \*iobuf);**

**struct stat sbuf; /\* stat buf \*/**

**} *awk*\_input\_buf\_t;**

这些域可以分成两类：一类是 XXX\_can\_take\_***file()*** 函数使用（最起码是），以及被 XXX\_take\_control\_***of() 使用，它们使用的第一组域如下：***

***const char \*name;***

文件的名字。

***int fd;*** 文件的描述符。如果 gawk 不能打开文件，fd 则会等于 INVALID\_HANDLE，否则等于文件打开的描述符。

***struct stat sbuf;***

如果文件描述符是有效的，gawk 会将这个结构通过 fstat() 系统调用来进行填充。

XXX\_can\_take\_***file()*** 函数应该检查这些域，并决定输入分析器是否可用于这个文件。决定依赖于 gawk 的状态（通过扩展以及由 awk 代码之前设置的变量的值），文件的名字，文件的描述符是否有效，在结构 stat 中的信息，以及所有这些因素的组合。

只要 XXX\_can\_take\_***file()*** 函数返回 true，gawk 就决定使用你的输入分析器，然后调用 XXX\_take\_control\_***of()***。这个函数会填充在 awk\_input\_buf\_t 结构中的 get\_recordfield 或者 read\_func 域。同时也要保证 fd 不会被设置为 INVALID\_HANDLE。下面的列表描述了可能会被 XXX\_take\_control\_***of()*** 函数使用的域：

***void \*opaque;***

这被用来存储任何的输入分析器针对这个文件所需要的状态信息。它对 gawk 是“透明”的。输入分析器并不一定要使用这个指针。

***int (\*get\_record)(char \*\*out，***

***struct awk\_input \*iobuf，***

***int \*errcode，***

***char \*\*rt\_start，***

***size\_t \*rt\_len);***

这个函数指针指向一个创建输入记录的函数。所述的函数也是输入分析器的核心。它的行为由本列表下面的文本说明。

***ssize\_t (\*read\_func)();***

这个函数指针应该指向与标准的 POSIX 函数 read() 系统调用行为相同的函数。它是 get\_record 函数指针的替代。它的行为也在本列表后面的文本中描述。

***void (\*close\_func)(struct awk\_input \*iobuf);***

这个函数指针指向一个“tear down”（结束）的函数。它应该释放所有由 XXX\_take\_control\_***of()*** 分配的资源。它也可以关闭文件。如果它这么做了，它应将 fd 域设置成 INVALID\_HANDLE。

如果 fd 在这个函数调用后还不是 INVALID\_HANDLE，gawk 会调用常规的 close() 系统调用。

是否有“teardown”函数是可选的。如果你的输入分析器不需要它，就不要设置这个域。这样，gawk 会在文件描述上调用常规的 close() 系统调用，因此这个域需要是有效的。

XXX\_get\_record***()*** 函数执行创建记录的工作。参数如下：

***char \*\*out***

这个是一个指向 char\* 变量的指针，这个指针指向记录。gawk 会复制自己的 data，所以扩展需要自己管理这个存储。

***struct awk\_input \*iobuf***

这个是用于文件的 ***awk***\_input\_buf\_t，里面的域应当用于读取数据（fd）以及如果有的话，管理私有状态（opaque）。

***int \*errcode***

如果出现了错误。\*errcode 应该被设置成 errno.h 文件中的相应代码。

***char \*\*rt\_start***

***size\_t \*rt\_len***

如果 “record terminator”有意义的话，则 \*rt\_start 应该被设置成指向用于 RT 的数据。而 \*rt\_len 应该被设置成数据的长度。否则， \*rt\_len 应该被设置为 0。gawk 会生成自己的 data 复本，所以扩展需要自己管理它们的存储。

返回值是由 \*out 指向的缓存的长度，或者在文件结束时返回 EOF，以及错误时返回错误。

必须保证 errcode 是一个有效的指针，因此就没有必要去测试它是否为 NULL 值。gawk 设置 \*errcode 为 0，所以如果没有错误的时候，也没有必要设置它。如果出现的错误，函数返回 EOF，并且设置 \*errcode 为一个大于 0 的值。在这种情况下，如果 \*errcode 不等于0，gawk 会根据 \*errcode 的值自动更新 ERRNO 变量。（一般说来，设置 “\*errcode = errno”就可以了。）

作为替代，提供一个返回输入记录的函数，你提供一个简单地读取字节的函数，gawk 会将它他分析并组合成记录。如果你这么做，数据应该以当前本地语言设置的多字节编码返回。这样的函数的行为应该与 read() 系统调用的一致，你将 ***awk***\_input\_buf\_t 结构中的 read\_func 指针填为这个函数指针。

默认情况下，gawk 会设置 read\_func 指向 read() 系统调用的指针。所以你的扩展不需要显式地这么去做。

提示：你必须选择一个或者另外一个方法：要么是一个返回记录的函数，或者一个返回原始数据的函数。特别是，如果你提供了一个函数来获取一个记录，gawk 会调用它，而绝不会调用原生的 read 函数。

gawk 在 read 目录下发布了一个例子扩展，返回某个目录下的每一个记录（查看 ， ***在第 397 页***）。你可以使用这个代码来作为指导来写你自己的输入分析器。

当写一个输入分析器时，你应该要考虑（并且用文件记录下来）它如何来与 awk 代码进行交互。你可能想它总是被调用，并且能够正确处理（如 readdir 扩展代码一样）。又或者你想它根据 awk 变量的值来确定是否被调用，如来自 gawkextlib 项目（查看 ， ***在第 400 页***）中的 XML 扩展一样。在后面的例子中，在 BEGINFILE 段中的代码可以查看 FILENAME 与 ERRNO 的值来决定是否激活输入分析器（查看 ， ***在第 143 页***）。

你通过下面的函数来注册输入分析器：

***void register\_input\_parser(awk\_input\_parser\_t \*input\_parser);***

注册通过 input\_parser 指向的输入分析器到 gawk 中。

#### 16.4.5.5 定制输出包装器

输出包装器是输入分析器的另一面。它可以使得扩展能够掌握通过 ‘>’ 或者‘>>’ 重定向I/O操作符打开文件（查看 ， ***在第 103 页***）。

输出包装器的结构与输入分析器的结构非常相似：

**typedef struct *awk*\_output\_wrapper {**

**const char \*name; /\* name of the wrapper \*/**

***awk*\_bool\_t (\*can\_take\_file)(const *awk*\_output\_buf\_t \*outbuf);**

***awk*\_bool\_t (\*take\_control\_of)(*awk*\_output\_buf\_t \*outbuf);**

***awk*\_const struct *awk*\_output\_wrapper \**awk*\_const next; /\* for gawk \*/**

**} *awk*\_output\_wrapper\_t;**

***成员如下：***

***const char \*name;***

这里输出包装器的名字。

***awk\_bool\_t (\*can\_take\_file)(const awk\_output\_buf\_t \*outbuf);***

这个指向一个函数，该函数检测由 outbuf 指向 awk\_output\_buf\_t 结构的信息。如果输出包装器想掌控输出文件，则应该返回 true，否则返回 false。它不应该改变任何 gawk 的状态（比如变量等）。

***awk\_bool\_t (\*take\_control\_of)(awk\_output\_buf\_t \*outbuf);***

由这个域所指向的函数在 gawk 决定使输出包装器控制文件的时候调用。它应该将 awk\_output\_buf\_t 结构中的成员适应地填充，如果成功则返回 true，否则返回 false。相关的说明在下面描述。

***awk\_const struct output\_wrapper \*awk\_const next;***

这个由 gawk 使用，因此它被标志为 awk\_const，这样扩展就不能够修改它。

***awk***\_output\_buf\_t 的结构如下：

**typedef struct *awk*\_output\_buf {**

**const char \*name; /\* name of output file \*/**

**const char \*mode; /\* mode argument to fopen \*/**

**FILE \*fp; /\* stdio file pointer \*/**

***awk*\_bool\_t redirected; /\* true if a wrapper is active \*/**

**void \*opaque; /\* for use by output wrapper \*/**

**size\_t (\*gawk\_fwrite)(const void \*buf， size\_t size， size\_t count，**

**FILE \*fp， void \*opaque);**

**int (\*gawk\_fflush)(FILE \*fp， void \*opaque);**

**int (\*gawk\_ferror)(FILE \*fp， void \*opaque);**

**int (\*gawk\_fclose)(FILE \*fp， void \*opaque);**

**} *awk*\_output\_buf\_t;**

这里也一样，你的施展要定义 XXX\_can\_take\_file() 与 XXX\_take\_control\_of() 这些函数，用以检测并更新 ***awk***\_output\_buf\_t 中的成员。数据成员的说明如下：

***const char \*name;***

输出文件的名字。

***const char \*mode;***

模式字串（正如 fopen() 第二个参数一样使用），通过使用这些模式来打开文件。

***FILE \*fp;*** 来自由 <stdio.h> 的 FILE 指针。gawk 会在查找输出包装器前试着打开文件。

***awk\_bool\_t redirected;***

这个域必须被 XXX\_take\_control\_***of()*** 设置为 true。

***void \*opaque;***

这个指针对于 gawk 是透明的。扩展可以使用它来存储与文件有关的私有数据的指针。

***size\_t (\*gawk\_fwrite)(const void \*buf， size\_t size， size\_t count，FILE \*fp， void \*opaque);***

***int (\*gawk\_fflush)(FILE \*fp， void \*opaque);***

***int (\*gawk\_ferror)(FILE \*fp， void \*opaque);***

***int (\*gawk\_fclose)(FILE \*fp， void \*opaque);***

这些指针应该被设置为指向与 <stdio.h> 中相应函数执行相同操作的函数。gawk 使用这些函数来处理所有输出。gawk 会将这些指针初始化指向内部的“过路”函数，这些函数中是调用常规的<stdio.h> 函数，所以扩展要根据所要做的事情定义合适函数。

XXX\_can\_take\_***file() 函数可以根据名字，模式域以及其他的状态（如 awk 变量的值）来做出相应的决定。***

当 gawk 调用 XXX\_take\_control\_***of()***函数时，这个函数应该对其他的域进行适当的填充，不包括 fp，fp 参数只是会进行常规使用。

你通过下面的函数来注册你的输出包装器：

***void register\_output\_wrapper(awk\_output\_wrapper\_t \*output\_wrapper);***

将由 output\_wrapper 指向的输出包装器注册到 gawk 中。

#### 16.4.5.6 定制双路处理器

一个双路器将输入分析器与输出包装器来处理由 ‘|&’打开的双路 I/O（查看 ）。它的使用与之前描述过的 ***awk***\_input\_parser\_t 与 ***awk***\_output\_buf\_t 相同。

两路处理器由下面的结构表示：

**typedef struct *awk*\_two\_way\_processor {**

**const char \*name; /\* name of the two-way processor \*/**

***awk*\_bool\_t (\*can\_take\_two\_way)(const char \*name);**

***awk*\_bool\_t (\*take\_control\_of)(const char \*name，**

***awk*\_input\_buf\_t \*inbuf，**

***awk*\_output\_buf\_t \*outbuf);**

***awk*\_const struct *awk*\_two\_way\_processor \**awk*\_const next; /\* for gawk \*/**

**} *awk*\_two\_way\_processor\_t;**

域如下：

***const char \*name;***

双路处理器的名字。

***awk\_bool\_t (\*can\_take\_two\_way)(const char \*name);***

由这个域指向的函数应该在控制指针文件的双路 I/O 时返回 true。它不应该改变 gawk 中的任何状态（比如变量值）。

***awk\_bool\_t (\*take\_control\_of)(const char \*name，***

***awk\_input\_buf\_t \*inbuf，***

***awk\_output\_buf\_t \*outbuf);***

由这个域指向的函数应该适当地填充由 inbuf 与 outbuf 所指向中的 awk\_input\_buf\_t 与 awk\_output\_buf\_t 的结构。这两个结构之前所描述。

***awk\_const struct two\_way\_processor \*awk\_const next;***

这个由 gawk 来使用，所以它被标志成 awk\_const，这样扩展就不可以修改它。

对于输入处理器与输出处理器器，你需要提供“yes I can take this”（是的，我可以处理这个） 与“take over for this”（我已经控制这个）类的函数，即 XXX\_can\_take\_two\_***way()*** 与 XXX\_take\_control\_***of()***。

你使用下面的函数来注册你的双路处理器：

***void register\_two\_way\_processor(awk\_two\_way\_processor\_t \*two\_way\_processor);***

将由 two\_way\_processor 指向的双路处理器注册到 gawk 中。

### 16.4.6 打印信息

你可以在你的扩展中打印如本节中所描述的不同的各类的告警信息。要注意对于这些函数，你必须提供一个扩展 ID，这个扩展 ID 是在扩展被载入时由 gawk 提供。[[106]](#footnote-107)

***void fatal(awk\_ext\_id\_t id， const char \*format， ...);***

打印一个消息，并使 gawk 立即退出。

***void warning(awk\_ext\_id\_t id， const char \*format， ...);***

打印告警信息。

***void lintwarn(awk\_ext\_id\_t id， const char \*format， ...);***

打印一个“lint 告警”。正常情况下，这与打印一个告警信息一样，但是如果 gawk 使用了 ‘--lint=fatal’ 选项，则 lint 告警会成为致命错误消息。

所有的这些函数像 C 的 printf 家庭的函数一样，在这些函数中，format 参数是一个字串，串中为字面字符以及格式代码。

### 16.4.7 更新 ERRNO

下面的函数可以让你更新 ERRNO 变量：

***void update\_ERRNO\_int(int errno\_val);***

将 ERRNO 设置为与 errno\_val 对应的错误字串。值应该定义在 <errno.h> 中。gawk 会使用 C 的 strerror() 函数将它转换为（如果可以转换）字串。

***void update\_ERRNO\_string(const char \*string);***

直接使用 string 的值来设置 ERRNO。gawk 会对这个值复制一个复本。

***void unset\_ERRNO(void);***

重置 ERRNO。

### 16.4.8 请求值

所有的这些函数都以相同的方式从 gawk 中返回值。你可以传递一个 awk\_valtype\_t 的值来指示你希望什么的样值。如果实际的值与你的请求的匹配，函数返回为 true。并且在 awk\_value\_t 中填上结果。否则函数返回 false。而 val\_type 成员会指示实际的值类型。你然后可以打印一个错误信息或者重新发起针对实际值类型的请求。这结行为在表 16.1 进行了归纳：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 实际的值类型 | | | | |
| 请求类型 |  | String | Number | Array | Undefined |
| String | String | String | False | False |
| Number | Number if can  be converted，  else false | Number | False | False |
| Array | False | False | Array | False |
| Scalar | Scalar | Scalar | False | False |
| Undefined | String | Number | Array | Undefined |
| Value Cookie | False | False | False | False |

表 16.1：返回的 API 值类型

### 16.4.9 访问与更新参数

有两个函数可以让你访问传递给你的扩展函数的参数。它们是：

***awk\_bool\_t get\_argument(size\_t count，***

***awk\_valtype\_t wanted，***

***awk\_value\_t \*result);***

用第 count 个参数来填充由 result 指向的 awk\_value\_t 结构。如果实际的类型是匹配的，则返回 true，否则返回 false。在后面的情况下，result->val\_type 指示了实际的类型（查看 表 16.1）。计数是从0开始的——第一个参数编号为0，第二个为1，等等。wanted 指示了所期望的值类型。

***awk\_bool\_t set\_argument(size\_t count， awk\_array\_t array);***

将未定义的参数转换为一个数组，这提供了通过数组引用来调用的方式。如果 count 太大，或者如果参数类型是未定义的，则返回 false。查看 ，***在第 371 页***，来查看创建数组的更多信息。

### 16.4.10 符号表访问

有两个集合的例程提供了访问全局变量的能力，以及一个集合的例程可以让你创建并释放缓存值。

#### 16.4.10.1 通过名字访问与更新变量

下面的例程提供了通过名字来访问、更新全局的 awk 层变量的能力。以编译器的术语来看，不同各类的标识符都称为符号，所以在例程中的 “sym”名字的来由。存储符号信息的数组结构称为符号表。这些函数如下：

***awk\_bool\_t sym\_lookup(const char \*name，***

***awk\_valtype\_t wanted，***

***awk\_value\_t \*result);***

用由字串 name 命令的变量的值来填充由 result 指向听 awk\_value\_t 的结构，名字为 C 字串。wanted 指示了期望的值的类型。如果实际的类型与其匹配，则返回 true，否则返回 false。在后面的情况下，result->val\_type 指示了实际的类型（查看表 16.1）。

***awk\_bool\_t sym\_update(const char \*name， awk\_value\_t \*value);***

更新由字串 name 命名的变量，这个名字为 C 字串。如果变量不存在 gawk 的符号表中，则会新增它。如果一切 OK，则返回 true，否则返回 false。

改变已经存在的变量的类型（从标量到数组，或者相反）是不允许的，这个例程也不能用于数组的更新。这个例程不可以用于更新任何的预定义变量的值（如 ARGC 或者 NF）。

扩展可以查找 gawk 的特定变量。但是，RPOCINFO 数组是个例外，扩展不可以改变这些变量的值。

注意：有可能查找 PROCINFO 失败。如果 awk 程序在运行时没有使用 PROCINFO 时会引起失败，在这个情况下，gawk 不会创建这个数组并传播它。

#### 16.4.10.2 通过 Cookie 访问与更新变量

标量 Cookie 是一个透明的句柄，可以用来访问全局变量以及数组。它是一种优化手段，以免在使用变量时去在 gawk 的符号表中去查找变量，这个早前在 ， ***在第 353 页***，已经讨论过。

下面的函数用来过处理标量 Cookies：

***awk\_bool\_t sym\_lookup\_scalar(awk\_scalar\_t cookie，***

***awk\_valtype\_t wanted，***

***awk\_value\_t \*result);***

返回标量 Cookie 的当前值，一旦你通过 sym\_lookup() 获取了一个标量的 Cookie 后，你就可以使用这个函数来高效地获取它的值。如果值没有取到，则返回 false。

***awk\_bool\_t sym\_update\_scalar(awk\_scalar\_t cookie， awk\_value\_t \*value);***

更新与标量 cookie 关联的值。如果新值即不是 AWK\_STRING，也不是 AWK\_NUMBER，则返回 false。在这里也是一样，预定的义的变量不能被更新的。

一开始这如何来使用标量 Cookie 并不是很明显，也不知道它们的 raisond’^etre 到底是什么。理论上， sym\_lookup() 以及 sym\_update() 例程就是你来处理变量时所要的所有例程。例如，你有代码要查找变量的值，对条件进行求值，然后就可以通过求值的结果变更变量的值，如：

**/\* do\_magic --- do something really great \*/**

**static *awk*\_value\_t \***

**do\_magic(int nargs， *awk*\_value\_t \*result)**

**{**

***awk*\_value\_t value;**

**if ( sym\_lookup("MAGIC\_VAR"， AWK\_NUMBER， & value)**

**&& some\_condition(value.num\_value)) {**

**value.num\_value += 42;**

**sym\_update("MAGIC\_VAR"， & value);**

**}**

**return make\_number(0.0， result);**

**}**

这个代码看起来（实际）很简单而直接，但是问题是什么呢？

考虑一下，如果 awk 层的代码通过调用 magic() 函数（通过 C 的 do\_magic() 来实现）来与你的扩展进行关联，一次一条记录来处理数以百万计的记录。变量 MAGIC\_VAR 会在每个函数的调用中都在符号表中进行查找！

符号表的查找真的是个纯的负担，如果得到一个表示这个变量的 Cookie 要更高效，并使用它在必要的时候获取与更新它的值。[[107]](#footnote-108)

所以，使用 Cookie 的方式如下。首先，将你的变量通过 sym\_update() 来安装到 gawk 的符号表中，然后通过 sym\_lookup() 获取一个该变量的 Cookie：

**static *awk*\_scalar\_t magic\_var\_cookie; /\* cookie for MAGIC\_VAR \*/**

**static void**

**my\_extension\_*init()***

**{**

***awk*\_value\_t value;**

**/\* install initial value \*/**

**sym\_update("MAGIC\_VAR"， make\_number(42.0， & value));**

**/\* get the cookie \*/**

**sym\_lookup("MAGIC\_VAR"， AWK\_SCALAR， & value);**

**/\* save the cookie \*/**

**magic\_var\_cookie = value.scalar\_cookie;**

**...**

**}**

下一步，使用这节中的函数来并通过 Cookie 来返回与更新变量的值。所以，do\_magic() 现在变成如下：

**/\* do\_magic --- do something really great \*/**

**static *awk*\_value\_t \***

**do\_magic(int nargs， *awk*\_value\_t \*result)**

**{**

***awk*\_value\_t value;**

**if ( sym\_lookup\_scalar(magic\_var\_cookie， AWK\_NUMBER， & value)**

**&& some\_condition(value.num\_value)) {**

**value.num\_value += 42;**

**sym\_update\_scalar(magic\_var\_cookie， & value);**

**}**

**...**

**return make\_number(0.0， result);**

**}**

提示：前面的代码省略了错误检查，为了是表现主题。你的扩展代码应该要更加健壮，并且仔细地检查从 API 函数中返回的值。

#### 16.4.10.3 创建并使用缓存值

本节中的全程可以让你创建与释放缓存值。像标量 Cookie 一样，理论上讲，缓存值是不需要的。你可以使用 ，中的函数创建数值与字串。你然后可以将这些值赋给使用 sym\_update() 或者 sym\_update\_scalar() 的变量。

但是，如果你记得所有的字串值的存储都来自于 gawk\_***malloc()***， gawk\_***calloc()***， 或者 gawk\_***realloc()*** 这些函数这一点，你就能够理解缓存值的出发点。如果你有 20 个变量，它们都有相同的值，你必须创建 20 个相同的字串复本。[[108]](#footnote-109)

当然，如果可能，一旦创建一个值，就可以告诉 gawk 可以为多个变量重用这个值。这正是本节中的例程所要做的事情。函数如下：

***awk\_bool\_t create\_value(awk\_value\_t \*value， awk\_value\_cookie\_t \*result);***

创建一个缓存的字串或者数值，用于后面的高效赋值。仅支持 AWK\_NUMBER 与 AWK\_STRING 类型的值。其他的类型会被拒绝。AWK\_UNDEFINED 类型也可以使用，但是这么做会导致较差的性能。

***awk\_bool\_t release\_value(awk\_value\_cookie\_t vc);***

释放与一个通过 create\_value() 获取的值 Cookie 关联的内存。

使用值 Cookie 的方式与使用标量 Cookie 的方式是一样的。在扩展初始化例程中，你这样来创建值 Cookie：

**static *awk*\_value\_cookie\_t answer\_cookie; /\* static value cookie \*/**

**static void**

**my\_extension\_*init()***

**{**

***awk*\_value\_t value;**

**char \*long\_string;**

**size\_t long\_string\_len;**

**/\* code from earlier \*/**

**...**

**/\* ... fill in long\_string and long\_string\_len ... \*/**

**make\_malloced\_string(long\_string， long\_string\_len， & value);**

**create\_value(& value， & answer\_cookie); /\* create cookie \*/**

**...**

**}**

值一旦创建，你就可以用它作为任意变量的值：

**static *awk*\_value\_t \***

**do\_magic(int nargs， *awk*\_value\_t \*result)**

**{**

***awk*\_value\_t new\_value;**

**... /\* as earlier \*/**

**value.val\_type = AWK\_VALUE\_COOKIE;**

**value.value\_cookie = answer\_cookie;**

**sym\_update("VAR1"， & value);**

**sym\_update("VAR2"， & value);**

**...**

**sym\_update("VAR100"， & value);**

**...**

**}**

通过这样的值 Cookies 可以节约大量的存储，从 VAR1 到 VAR100 都共享一样的值。

你可能会奇怪，“这样的共享不会有问题么？如果 awk 代码给 VAR1 赋了一个新值，是不是所有的其他变量都会改变？”

这是一个很好的问题。答案是不会，这不会是个问题。在内部，gawk 使用参考计数的字串。这就是说，许多的变量都可以使用共享的字串值，gawk 会跟踪它的使用。当变量值改变时，gawk 仅是简单地递减旧值的参考计数，然后更新变量让其使用新的值。

最后，作为清除操作的一部分（查看 ， ***在第 358 页***），你应该通过 release\_value() 来释放你创建的缓存值。

### 16.4.11 数组操作

在 awk 中，最重要的数据结构[[109]](#footnote-110)与数组（查看 ***， 在第 166 页***）相关。扩展需要有能力操作 awk 中的数组。API 中提供了很多的数据结构来处理数组，也提供了处理单个元素的函数，以及处理整个数组的函数。这包含了扁平化数组的能力，这样就很容易使用 C 代码来遍历数组中的每一个元素。数组数据结构很好地与值的数据结构进行了整合，使得很容易来创建真正的数组之数组（查看 ， ***在第 353 页***）。

#### 16.4.11.1 数组数据类型

与数组相关的数据类型有：

***typedef void \*awk\_array\_t;***

如果你请求一个数组变量的值，你可以取得一个 awk\_array\_t 的值。这个值对于扩展来说是透明[[110]](#footnote-111)的。这唯一地标识数组，但是只可以将其传递给 API 函数来使用，或者从 API 函数中返回。这个方式与 <stdio.h> 中函数使用 ‘FILE\*’值类似。

***typedef struct awk\_element {***

***/\* convenience linked list pointer， not used by gawk \*/***

***struct awk\_element \*next;***

***enum {***

***AWK\_ELEMENT\_DEFAULT = 0， /\* set by gawk \*/***

***AWK\_ELEMENT\_DELETE = 1 /\* set by extension \*/***

***} flags;***

***awk\_value\_t index;***

***awk\_value\_t value;***

***} awk\_element\_t;***

awk\_element\_t 是一个平的数组元素。awk 会在 awk\_flat\_array\_t 类型中生成一个此类型的数组（查看下一项的说明）。单个的元素可以被标志成删除。新的元素必须一个个地添加，一次一个。为了完成这个添加，要使用另外的 API。数据结构中的域为：

struct ***awk***\_element \*next;

这是为了方便扩展的编写者的一个指针。它允许扩展创建一个新元素的链表，这样可以在遍历列表时来加入到数组中。

enum { ... } flags;

在扩展与 gawk 之间传递信息的标志值。当前只有一个 AWK\_ELEMENT\_DELETE，如果设置这个标志，会使得 gawk 释放扁平的数组时，从原始的数组中删除这个元素。

index

value 无素的索引与值。所有被 index 与 value 指向的内存都属于 gawk。

***typedef struct awk\_flat\_array {***

***awk\_const void \*awk\_const opaque1; /\* for use by gawk \*/***

***awk\_const void \*awk\_const opaque2; /\* for use by gawk \*/***

***awk\_const size\_t count; /\* how many elements \*/***

***awk\_element\_t elements[1]; /\* will be extended \*/***

***} awk\_flat\_array\_t;***

这是一个平面数组。当扩展从 gawk 中取得其中之一时，元素数组的实际大小就是 count。opaque1 与 opaque2 指针由 gawk 使用，所以，它们被标志成 awk\_const，这样扩展就不可以修改它们。

#### 16.4.11.2 数组函数

下面的函数与单个的元素相关：

***awk\_bool\_t get\_element\_count(awk\_array\_t a\_cookie， size\_t \*count);***

为由 a\_cookie 表示的数组，放入其包含的 \*count 个元素。子数组由一个单独的元素表示。如果有错误时，返回 false。

***awk\_bool\_t get\_array\_element(awk\_array\_t a\_cookie，***

***const awk\_value\_t \*const index，***

***awk\_valtype\_t wanted，***

***awk\_value\_t \*result);***

从 a\_cookie 表示的数组中，返回元素下标 index 的值到 \*result 中。wanted 指定了你希望返回的值类型。如果 wanted 不与实际的类型匹配，或者 index 没有在数组中，则返回 false。（查看表 16.1）

index 的值可以是数值，这种情况下， gawk 会将其转换为字串。不使用整数值也是可以的，但是需要你了解这样的值如何转换为字串（查看 ， ***在第 117 页***），所以，使用整数值是最安全的。

与从扩展中传递到 gawk 中的所有的字串一样，index 的字串值必须来自于 gawk\_malloc***()***， gawk\_***calloc()***， 或者 gawk\_***realloc()***，并由 gawk\_release() 来释放内存。

***awk\_bool\_t set\_array\_element(awk\_array\_t a\_cookie，***

***const awk\_value\_t \*const index，***

***const awk\_value\_t \*const value);***

在由 a\_cookie 表示的数组中，创建或者修改一个元素，这个元素的索引由 index 给出。ARGV 与 ENVIRON 数组不可以被改变，但是 PROCINFO 数组则可以。

***awk\_bool\_t set\_array\_element\_by\_elem(awk\_array\_t a\_cookie，***

***awk\_element\_t element);***

与 set\_array\_element***()*** 相似，但是是从元素中取得索引与值。这是一个便利宏。

***awk\_bool\_t del\_array\_element(awk\_array\_t a\_cookie，***

***const awk\_value\_t\* const index);***

删除由 a\_cookie 表教学点的数组中，由 index 给出的元素。如果元素被删除，则返回 true。否则，如果元素不存在数组中，则返回 false。

下面的函数与整个数组有关：

***awk\_array\_t create\_array(void);***

创建一个可以添加元素的数组。查看 ，***在第 377 页***，中对于创建数组并添加元素的讨论。

***awk\_bool\_t clear\_array(awk\_array\_t a\_cookie);***

清除由 a\_cookie 表示的数组。如果有问题返回 false，否则返回 true。数组还是数组，但是调用这个函数后，它就没有元素了。这个与 delete 语句等同（查看 ， ***在第 175 页***）。

***awk\_bool\_t flatten\_array(awk\_array\_t a\_cookie， awk\_flat\_array\_t \*\*data);***

为由 a\_cookie 表示的数组，创建一个 awk\_flat\_array\_t 的结构并进行填充。将通过 data 传递的指针的值设置为这个结构的地址。如果成功则返回 true，否则返回 false。查看 ，***在第 374 页***，中讨论如何来扁平化数组并处理它。

***awk\_bool\_t release\_flattened\_array(awk\_array\_t a\_cookie，***

***awk\_flat\_array\_t \*data);***

当完成了数组的扁平化，释放使用这个函数的存储。你必须将最初的数组 cookie 与创建的 awk\_flat\_array\_t 数组结构的地址都传递进去。如果成功返回 true，否则返回 false。

#### 16.4.11.3 遍历数组的所有元素

扁平化数组就是创建一个数据结构来表示整个数组，其风格使得很容易使用 C 代码来遍历整个数组。在 extension/testext.c 中的一些代码就是来做这些事情，也是一个很好的例子来说明如何来使用 API。

我们一步步地来过这里面的代码。首先 gawk 脚本来驱动测试扩展：

**@load "testext"**

**BEGIN {**

**n = split("blacky rusty sophie raincloud lucky"， pets)**

***printf*("pets has %d elements\n"， length(pets))**

**ret = dump\_array\_and\_delete("pets"， "3")**

***printf*("dump\_array\_and\_delete(pets) returned %d\n"， ret)**

**if ("3" in pets)**

***printf*("dump\_array\_and\_*delete()* did NOT remove index \"3\"!\n")**

**else**

***printf*("dump\_array\_and\_*delete()* did remove index \"3\"!\n")**

**print ""**

**}**

这个代码使用 split 函数创建了一个数组（查看 ， ***在第 184 页***）然后调用 dump\_array\_and\_delete()。这个函数用于查找数组，其第一个参数是数组名，而删除的元素的索引通过第二个参数来传递。awk 代码然后打印返回的值并检查元素是否真的被删除了。下面是 dump\_array\_and\_delete() 函数的C 代码实现。为了表现上的原因做了一些修改。

第一部分是声明变量，设置默认的返回值到 result 中，并检查函数的调用是通过正确的参数个来调用的：

**static *awk*\_value\_t \***

**dump\_array\_and\_delete(int nargs， *awk*\_value\_t \*result)**

**{**

***awk*\_value\_t value， value2， value3;**

***awk*\_flat\_array\_t \*flat\_array;**

**size\_t count;**

**char \*name;**

**int i;**

**assert(result != NULL);**

**make\_number(0.0， result);**

**if (nargs != 2) {**

***printf*("dump\_array\_and\_delete: nargs not right "**

**"(%d should be 2)\n"， nargs);**

**goto out;**

**}**

函数然后一步步地执行，如下。首先返回数组的名字，其作为函数的第一个参数初传递。然后是数组本身。如果有一个操作失败，则打印错误信息并返回：

**/\* get argument named array as flat array and print it \*/**

**if (get\_argument(0， AWK\_STRING， & value)) {**

**name = value.str\_value.str;**

**if (sym\_lookup(name， AWK\_ARRAY， & value2))**

***printf*("dump\_array\_and\_delete: sym\_lookup of %s passed\n"，name);**

**else {**

***printf*("dump\_array\_and\_delete: sym\_lookup of %s failed\n"，name);**

**goto out;**

**}**

**} else {**

***printf*("dump\_array\_and\_delete: get\_argument(0) failed\n");**

**goto out;**

**}**

为了测试并确保 C 代码看代的元素个数与 awk 代码看到的元素个数是一样的，第二步是来获取数组元素的个数然后打印它：

**if (! get\_element\_count(value2.array\_cookie， & count)) {**

***printf*("dump\_array\_and\_delete: get\_element\_count failed\n");**

**goto out;**

**}**

***printf*("dump\_array\_and\_delete: incoming size is %lu\n"，(unsigned long) count);**

第三步是执行实际的数组扁平化，然后再次检查 ***awk***\_flat\_array\_t 中的数量是与刚才返回的数量是一样的：

**if (! flatten\_array(value2.array\_cookie， & flat\_array)) {**

***printf*("dump\_array\_and\_delete: could not flatten array\n");**

**goto out;**

**}**

**if (flat\_array->count != count) {**

***printf*("dump\_array\_and\_delete: flat\_array->count (%lu)"" != count (%lu)\n"，**

**(unsigned long) flat\_array->count，**

**(unsigned long) count);**

**goto out;**

**}**

第四步是返回要被删除的元素的索引，这个索引是通过第二个参数来传递的。要记住传递到 get\_***argument()*** 函数中的参数编号是从 0 开始的，因此第二个参数被编号为 1：

**if (! get\_argument(1， AWK\_STRING， & value3)) {**

***printf*("dump\_array\_and\_delete: get\_argument(1) failed\n");**

**goto out;**

**}**

第五步是执行“真正的”操作。函数循环数组中的每一个元素，打印其索引与其元素值。另外，当找到要删除元素的索引时，函数设置 AWK\_ELEMENT\_DELETE 位到元素的 flags 域中。当数组被释放时，gawk 会遍历扁平化后的数组，然后删除那些被设置了该位的元素：

**for (i = 0; i < flat\_array->count; i++) {**

***printf*("\t%s[\"%.\*s\"] = %s\n"，name，**

**(int) flat\_array->elements[i].index.str\_value.len，**

**flat\_array->elements[i].index.str\_value.str，**

**valrep2str(& flat\_array->elements[i].value));**

**if (strcmp(value3.str\_value.str，**

**flat\_array->elements[i].index.str\_value.str) == 0) {**

**flat\_array->elements[i].flags |= AWK\_ELEMENT\_DELETE;**

***printf*("dump\_array\_and\_delete: marking element \"%s\"""for deletion\n"，**

**flat\_array->elements[i].index.str\_value.str);**

**}**

**}**

第六步是释放扁平化后的数组。这会告诉 gawk 扩展已经不再使用数组，它可以删除所有已经被标志的元素。gawk 也会释放被分配的存储，因此，在调用 release\_flattened\_***array()*** 函数后，你不要再使用里面的指针（在这个代码中为 flat\_array）：

**if (! release\_flattened\_array(value2.array\_cookie， flat\_array)) {**

***printf*("dump\_array\_and\_delete: could not release flattened array\n");**

**goto out;**

**}**

最后，由于一切事情都执行良好，函数将返回值设置为成功，然后返回：

**make\_number(1.0， result);**

**out:**

**return result;**

**}**

这里是运行测试部分的输出：

**pets has 5 elements**

**dump\_array\_and\_delete: sym\_lookup of pets passed**

**dump\_array\_and\_delete: incoming size is 5**

**pets["1"] = "blacky"**

**pets["2"] = "rusty"**

**pets["3"] = "sophie"**

**dump\_array\_and\_delete: marking element "3" for deletion**

**pets["4"] = "raincloud"**

**pets["5"] = "lucky"**

**dump\_array\_and\_delete(pets) returned 1**

**dump\_array\_and\_*delete()* did remove index "3"!**

#### 16.4.11.4 如何生成与发布数组

除了通过 awk 代码来处理数组外，你也可以创建数组然后发布它们，然后 awk 代码也可以访问并处理它们。

从扩展中创建数组有两个重点：

* 你必须将新的数组创建后立即安装到 gawk 的符号表中。这么做了后，就可以发布你的数组。

相似的，如果将某个数组安装成已经存在的数组的子数组，你必须在向它添加任何元素之前将其加到父数组中。

因此，创建数组的正确方式是“自上而下”。创建一个数组，然后立即使用 sym\_update() 将回添加到符号表中，或者使用 set\_array\_element() 将其添加为一个已经存在的数组的一个元素。我们后面会有例子说明。

* 由于 gawk 的内部实现，在使用了 sym\_update() 安装了一个数组到 gawk 中后，在使用它做任何事情之前，你必须使用传给 sym\_update() 接口中的值来获取一个数组的 Cookie：

***awk*\_value\_t value;**

***awk*\_array\_t new\_array;**

**new\_array = create\_*array()*;**

**val.val\_type = AWK\_ARRAY;**

**val.array\_cookie = new\_array;**

**/\* install array in the symbol table \*/**

**sym\_update("array"， & val);**

**new\_array = val.array\_cookie; /\* YOU MUST DO THIS \*/**

如果安装的是一个子数组，你必须在调用 set\_element() 之后使用数组的值返回一个数组 Cookie。

下面的 C 代码是一个简单测试扩展，用两个常规的元素来创建一个数组，并带有一个子数组。开始的 #include 指示符以及样板式的变量声明（查看 ， ***在第 380 页***）为了简化而省略了。第一步是创建一个新的数组，然后将其安装到符号表中：

**/\* create\_new\_array --- create a named array \*/**

**static void**

**create\_new\_*array()***

**{**

***awk*\_array\_t a\_cookie;**

***awk*\_array\_t subarray;**

***awk*\_value\_t index， value;**

**a\_cookie = create\_*array()*;**

**value.val\_type = AWK\_ARRAY;**

**value.array\_cookie = a\_cookie;**

**if (! sym\_update("new\_array"， & value))**

***printf*("create\_new\_array: sym\_update(\"new\_array\") failed!\n");**

**a\_cookie = value.array\_cookie;**

注意 a\_cookies 是如何使用值中的 array\_cookie 的域来进行重置的。

第二步是安装一个常规的到 new\_array 中：

**(void) make\_const\_string("hello"， 5， & index);**

**(void) make\_const\_string("world"， 5， & value);**

**if (! set\_array\_element(a\_cookie， & index， & value)) {**

***printf*("fill\_in\_array: set\_array\_element failed\n");**

**return;**

**}**

**(void) make\_const\_string("answer"， 6， & index);**

**(void) make\_number(42.0， & value);**

**if (! set\_array\_element(a\_cookie， & index， & value)) {**

***printf*("fill\_in\_array: set\_array\_element failed\n");**

**return;**

**}**

第三步是创建了一子数组，然后安装它：

**(void) make\_const\_string("subarray"， 8， & index);**

**subarray = create\_*array()*;**

**value.val\_type = AWK\_ARRAY;**

**value.array\_cookie = subarray;**

**if (! set\_array\_element(a\_cookie， & index， & value)) {**

***printf*("fill\_in\_array: set\_array\_element failed\n");**

**return;**

**}**

**subarray = value.array\_cookie;**

最后一步是发布子数组以及它的元素：

**(void) make\_const\_string("foo"， 3， & index);**

**(void) make\_const\_string("bar"， 3， & value);**

**if (! set\_array\_element(subarray， & index， & value)) {**

***printf*("fill\_in\_array: set\_array\_element failed\n");**

**return;**

**}**

**}**

这个一个装载扩展的脚本，然后 dump 数组：

**@load "subarray"**

**function dumparray(name， array， i)**

**{**

**for (i in array)**

**if (isarray(array[i]))**

**dumparray(name "[\"" i "\"]"， array[i])**

**else**

***printf*("%s[\"%s\"] = %s\n"， name， i， array[i])**

**}**

**BEGIN {**

**dumparray("new\_array"， new\_array);**

**}**

下面是运行脚本的结果：

**$ AWKLIBPATH=$PWD ./gawk -f subarray.*awk***

**-| new\_array["subarray"]["foo"] = bar**

**-| new\_array["hello"] = world**

**-| new\_array["answer"] = 42**

（查看 ，***在第 382 页***，来获取更多的关于变量 AWKLIBPATH 的信息）。

### 16.4.12 API 变量

API 提供了两种类型的变量。第一种提供 API 版本信息（包括扩展是如何进行编译以及 gawk 如何进行编译的）。第二种是提供 gawk 如何被调用的信息。

#### 16.4.12.1 API 版本常量与变量

API 提供了“major” 与 “minor” 版本值。API 版本可以在编译的时候通过常数来得到：

***GAWK\_API\_MAJOR\_VERSION***

API 的主版本号。

***GAWK\_API\_MINOR\_VERSION***

API 的次版本号

次版本号会在向 API 添加新功能的时候进行递增。这些新功能总是添加在 API 结构的尾部。

如果数组类型改变了大小，或者成员顺序有变化，或者现有的函数改变了签名，会对主版本号递增（次版本号则重设为 0）。

有可能扩展是以某个版本的 API 进行编译的，而被另一个版本的 gawk 版本进行装载。由于这个原因，执行 gawk API 的主次版本号会包含在 API 的结构中，并且是只读的：

***api->major\_version***

执行 gawk 的主版本号。

***api->minor\_version***

执行 gawk 的次版本号。

扩展可以决定 API 是否是兼容的。典型的情况下，像下面这样的检查就可以了：

**if (api->major\_version != GAWK\_API\_MAJOR\_VERSION**

**|| api->minor\_version < GAWK\_API\_MINOR\_VERSION) {**

**fprintf(stderr， "foo\_extension: version mismatch with gawk!\n");**

**fprintf(stderr， "\tmy version (%d， %d)， gawk version (%d， %d)\n"，**

**GAWK\_API\_MAJOR\_VERSION， GAWK\_API\_MINOR\_VERSION，**

**api->major\_version， api->minor\_version);**

**exit(1);**

**}**

这样的代码被包含在模板 dl\_load\_func() 宏中，并由 gawkapi.h 文件中（在 ， ***见下方***，中进行讨论）。

#### 16.4.12.2 信息变量

API 提供了多个变量的访问，这些变量描述了 gawk 被调用时，相应的命令行选项是否启用。这些变量是：

***do\_debug*** 这个变量当使用了 --debug 选项来调用 gawk 时会为 true。

***do\_lint*** 这个变量当使用了 --lint 选项来调用 gawk 时会为 true。

***do\_mpfr*** 这个变量当使用了 --bignum 选项来调用 gawk 时会为 true。

***do\_profile***

这个变量当使用了 --profile 选项来调用 gawk 时会为 true。

***do\_sandbox***

这个变量当使用了 --sandbox 选项来调用 gawk 时会为 true。

***do\_traditional***

这个变量当使用了 --traditional 选项来调用 gawk 时会为 true。

do\_lint 的值如果 awk 代码修改了 LINT 预定义变量（查看 ， ***在第 155 页***）可能会改变。其他的在执行过程中都不会改变。

### 16.4.13 样板代码

如之前所提（查看 ， ***在第 350 页***），函数定义的呈现实际上是一个宏。为了使用这些宏，你的扩展必须使用下面的预定义的名字，提供少量的样板代码（变量以及函数）放在你的源代码的头部。所需要的样板代码也提供在 gawkapi.h 头文件的注释中：

**/\* Boilerplate code: \*/**

**int plugin\_is\_GPL\_compatible;**

**static gawk\_api\_t \*const api;**

**static *awk*\_ext\_id\_t ext\_id;**

**static const char \*ext\_version = NULL; /\* or ... = "some string" \*/**

**static *awk*\_ext\_func\_t func\_table[] = {**

**{ "name"， do\_name， 1 }，**

**/\* ... \*/**

**};**

**/\* EITHER: \*/**

**static *awk*\_bool\_t (\*init\_func)(void) = NULL;**

**/\* OR: \*/**

**static *awk*\_bool\_t**

**init\_my\_extension(void)**

**{**

**...**

**}**

**static *awk*\_bool\_t (\*init\_func)(void) = init\_my\_extension;**

**dl\_load\_func(func\_table， some\_name， "name\_space\_in\_quotes")**

这些变量与函数如下：

***int plugin\_is\_GPL\_compatible;***

这个用来声明扩展是与 GNU GPL 兼容的（查看 ***，在第 465 页***）。如果你的扩展没有这个变量，gawk 不会装载它（查看 ， ***在第 349 页***）。

***static gawk\_api\_t \*const api;***

这个全局的静态变量应该被设定指向 gawk\_api\_t 指针，gawk 会在 dl\_load() 函数中传递过来。这个变量被所有的宏使用。

***static awk\_ext\_id\_t ext\_id;***

这个全局静态变量应该设置为 awk\_ext\_id\_t 的值，这个值会通过你自己的 dl\_load() 函数传递绘画你。这个变量被所有的宏使用。

***static const char \*ext\_version = NULL; /\* or ... = "some string" \*/***

这个全局静态变量要么被设置为 NULL，或者是一个字串的指针，这个字串绘画了了扩展的名字与版本。

***static awk\_ext\_func\_t func\_table[] = { ... };***

这个一个或者多个 awk\_ext\_func\_t 结构的数组，如之所描述的（查看 ， ***在第 357 页***）。它之后可以通过循环来多次调用 add\_ext\_func()。

***static awk\_bool\_t (\*init\_func)(void) = NULL;***

或者

***static awk\_bool\_t init\_my\_extension(void) { ... }***

***static awk\_bool\_t (\*init\_func)(void) = init\_my\_extension;***

如果你需要做一些初始化的工作，你应该定义这样的函数（创建变量，打开文件等）然后定义 init\_func 指针指向你的函数。函数应该在失败时返回 awk\_false ，或者成功时返回 true。

如果你不需要初始化，定义这个指针，并将其设置为 NULL。

***dl\_load\_func(func\_table， some\_name， "name\_space\_in\_quotes")***

这个宏会扩展为 dl\_load() 函数，该函数执行所有必要的初始化。

所有变量与数组的目的是使 dl\_load() 函数（来自于 dl\_load\_func() 宏中）处理所有的标准工作。它执行下面的工作：

1. 检查 API 版本。如果扩展的主版本与 gawk 的主版本不匹配，或者扩展的次版本大于 gawk 的次版本号，它会打印错误信息后退出。
2. 装入定义在 func\_table 中的函数。如果它们中的任何一个失败，它会打印警告信息，但是继续执行。
3. 如果 init\_func 指针不为 NULL，则调用它指向的函数。如果它返回 awk\_false，打印一个警告信息。
4. 如果 ext\_version 不为 NULL，注册版本字串到 gawk 中。

## 16.5 gawk 如何找到扩展

编译好的扩展必须安装在 gawk 可以找到它们的路径下。如果 gawk 被配置并以默认的方式进行构建，来搜索扩展的目录为 /usr/local/lib/gawk。你也可以指定一个目录列表来搜索编译好的扩展。查看 ，***在第 44 页***，来获取其他信息。

## 16.6 例子：一些文件函数

*你可以到达任何你想去的地方。*

*—Buckaroo Banzai*

有两个非常有用的函数不在 awk中，他们是 chdir()（这样 awk 程序可以改变它的目录）与 stat()（这样 awk 程序可以收集文件的信息）。为了用实际的例子说明 API，这一节会将其实现为 gawk 的扩展。

### 16.6.1 使用 *chdir()* 与 *stat()*

这一节展示如何来在 awk 层上如何来使用新函数，只要他们已经整合到了 gawk 中的解释器中。使用 chdir() 非常直接。它只有一个参数，即要更改的新目录：

**@load "filefuncs"**

**...**

**newdir = "/home/arnold/funstuff"**

**ret = chdir(newdir)**

**if (ret < 0) {**

***printf*("could not change to %s: %s\n"， newdir， ERRNO) >"/dev/stderr"**

**exit 1**

**}**

**...**

如果 chdir() 失败了会返回一个负值，ERRNO（查看 ， ***在第 155 页***） 的值会被设置为指示错误的字串。

使用 stat() 就有一点复杂。C 的 stat() 函数会填充一个有大量信息的结构。在 awk 中对其进行建模的正确方法是使用关联数组来表示对应的信息：

**file = "/home/arnold/.profile"**

**ret = stat(file， fdata)**

**if (ret < 0) {**

***printf*("could not stat %s: %s\n"，**

**file， ERRNO) >"/dev/stderr"**

**exit 1**

**}**

***printf*("size of %s is %d bytes\n"， file， fdata["size"])**

stat() 函数总会清除数据数组，就算 stat() 失败情况下也如此。它会以填充下面的元素：

***"name"*** 要进行 stat() 操作的文件的名字。

***"dev"***

***"ino"*** 文件的设备与 inode 号。

***"mode"*** 文件的访问权限，为数值。这包含文件的类型以及权限。

***"nlink"*** 硬连接的个数。

***"uid"***

***"gid"*** 文件所有者的用户 ID 与组 ID。

***"size"*** 文件的字节大小。

***"blocks"*** 文件实际占有块的个数。如果文件有洞，则这个数组不会是其大小的函数。

***"atime"***

***"mtime"***

***"ctime"*** 文件的最后访问，修改及 inode 更新时间。它们都是数值的时间戳，适合用于 strftime()（查看 ， ***在第 197 页***）。

***"pmode"*** 文件的“printable mode"。这个是一个文件类型与权限的字串表示，如 ‘ls -l’输出。如，“drwxr-xr-x”。

***"type"*** 文件类型的可打印字串，值可能是下面之一：

"blockdev"

"chardev"

文件是一个块或者字符设备（“特殊文件”）。

"directory"

文件是目录。

"fifo" 文件是一个命名管道（也称为 FIFO）。

"file" 文件是一个常规文件。

"socket" 文件是文件系统中的一个 AF\_UNIX（“Unix domain”）socket。

"symlink" 文件是一个符号连接。

***"devbsize"***

文件的通过 “blocks”进行索引的基本块大小。大多数系统中，这个信息是从 <sys/param.h> 文件中的 DEV\_SIZE 或者是 BSD 系统中 <sys/stat.h> 文件中的 S\_BLKSIZE 继承下来的。对于其他的系统，预先的知识为其提供值。当没法确定值的时候，默认为 512。

还可能有几个额外的元素，其依赖于操作系统以及文件的类型。你可以在 awk 程序中使用 in 操作符来对它们进行测试（查看 ， ***在第 168 页***）：

***"blksize"***

用于文件 I/O 优先块大小。这个域在所有类 POSIX系统中的 C stat 结构中不存在。

***"linkval"***

如果文件是一个连接，这个元素就是连接文件所指向的文件名（即连接的值）。

***"rdev"***

***"major"***

***"minor"***

如果文件是一个块设备或者是一个字符设备文件，则这个值代表了设备号的数值，以及设备号中的主次设备号。

### 16.6.2 *chdir()* 与 *stat()* 的C 代码

下面是这些扩展的 C 代码。[[111]](#footnote-112)

这个文件中包含很多的标准头文件，然后再包含 gawkapi.h 头文件，这个文件文件中包了 API 的定义。后再是跟着一些必要变量声明，以使用 API 中的宏与模板代码（查看 ， ***在第 380 页***）：

**#ifdef HAVE\_CONFIG\_H**

**#include <config.h>**

**#endif**

**#include <stdio.h>**

**#include <assert.h>**

**#include <errno.h>**

**#include <stdlib.h>**

**#include <string.h>**

**#include <unistd.h>**

**#include <sys/types.h>**

**#include <sys/stat.h>**

**#include "gawkapi.h"**

**#include "gettext.h"**

**#define \_(msgid) gettext(msgid)**

**#define N\_(msgid) msgid**

**#include "gawkfts.h"**

**#include "stack.h"**

**static const gawk\_api\_t \*api; /\* for convenience macros to work \*/**

**static *awk*\_ext\_id\_t \*ext\_id;**

**static *awk*\_bool\_t init\_filefuncs(void);**

**static *awk*\_bool\_t (\*init\_func)(void) = init\_filefuncs;**

**static const char \*ext\_version = "filefuncs extension: version 1.0";**

**int plugin\_is\_GPL\_compatible;**

习惯上，对于 awk 函数 foo()，实现它的函数称为 do\_foo()。这些函数应该有两个参数。第一个是 int，通常称为 nargs，表示的是函数中实际参数的个数。第二个是 awk\_value\_t 结构的指针，一般命名为 result：

**/\* do\_chdir --- provide dynamically loaded *chdir()* function for gawk \*/**

**static *awk*\_value\_t \***

**do\_chdir(int nargs， *awk*\_value\_t \*result)**

**{**

***awk*\_value\_t newdir;**

**int ret = -1;**

**assert(result != NULL);**

**if (do\_lint && nargs != 1)**

**lintwarn(ext\_id，\_("chdir: called with incorrect number of arguments， "**

**"expecting 1"));**

newdir 变量表示要改变的新目录，这个是通过 get\_argument() 来获取的。注意第一个参数的编号为 0。

如果参数被成功返回，则函数调用 chdir() 系统调用。如果 chdir() 失败，则 ERRNO 的值会被更改：

**if (get\_argument(0， AWK\_STRING， & newdir)) {**

**ret = chdir(newdir.str\_value.str);**

**if (ret < 0)**

**update\_ERRNO\_int(errno);**

**}**

最后，函数返回要返回的值到 awk 层：

**return make\_number(ret， result);**

**}**

stat() 扩展涉及的东西更多一些。首先是一个将数值模式转换为可打印表示的函数（比如 0644 转换为 ‘-rw-r--r--’）。简单起见，这里省略了：

**/\* format\_mode --- turn a stat mode field into something readable \*/**

**static char \***

**format\_mode(unsigned long fmode)**

**{**

**...**

**}**

下一个函数用来读取软链接，也为了简化，在这里省略了：

**/\* read\_symlink --- read a symbolic link into an allocated buffer.**

**... \*/**

**static char \***

**read\_symlink(const char \*fname， size\_t bufsize， ssize\_t \*linksize)**

**{**

**...**

**}**

有两个辅助函数会简化将值输入到数组的过程，这个数组将包含 stat() 的结果：

**/\* array\_set --- set an array element \*/**

**static void**

**array\_set(*awk*\_array\_t array， const char \*sub， *awk*\_value\_t \*value)**

**{**

***awk*\_value\_t index;**

**set\_array\_element(array，**

**make\_const\_string(sub， strlen(sub)， & index)，value);**

**}**

**/\* array\_set\_numeric --- set an array element with a number \*/**

**static void**

**array\_set\_numeric(*awk*\_array\_t array， const char \*sub， double num)**

**{**

***awk*\_value\_t tmp;**

**array\_set(array， sub， make\_number(num， & tmp));**

**}**

下面的函数处理大部份的工作，即将从一个有效的 stat 结构中取值，然后将这些值填充到 awk\_array\_t 类型的结果数组中。这个工作由一个单独的函数来完成，为 gawk 提供 stat() 函数支持，也支持 fts() 扩展。 这个扩展也包含在同一个文件中，但是它的代码没有在这时显示（查看 ， ***在第 391 页***）。

这个函数的第一部分是变量声明，包含将文件类型影射到字串的表：

**/\* fill\_stat\_array --- do the work to fill an array with stat info \*/**

**static int**

**fill\_stat\_array(const char \*name， *awk*\_array\_t array， struct stat \*sbuf)**

**{**

**char \*pmode; /\* printable mode \*/**

**const char \*type = "unknown";**

***awk*\_value\_t tmp;**

**static struct ftype\_map {**

**unsigned int mask;**

**const char \*type;**

**} ftype\_map[] = {**

**{ S\_IFREG， "file" }，**

**{ S\_IFBLK， "blockdev" }，**

**{ S\_IFCHR， "chardev" }，**

**{ S\_IFDIR， "directory" }，**

**#ifdef S\_IFSOCK**

**{ S\_IFSOCK， "socket" }，**

**#endif**

**#ifdef S\_IFIFO**

**{ S\_IFIFO， "fifo" }，**

**#endif**

**#ifdef S\_IFLNK**

**{ S\_IFLNK， "symlink" }，**

**#endif**

**#ifdef S\_IFDOOR /\* Solaris weirdness \*/**

**{ S\_IFDOOR， "door" }，**

**#endif /\* S\_IFDOOR \*/**

**};**

**int j， k;**

然后目标数组也被声明了，后面的代码根据 stat 结构中的值来填充不同的元素：

**/\* empty out the array \*/**

**clear\_array(array);**

**/\* fill in the array \*/**

**array\_set(array， "name"， make\_const\_string(name， strlen(name)，& tmp));**

**array\_set\_numeric(array， "dev"， sbuf->st\_dev);**

**array\_set\_numeric(array， "ino"， sbuf->st\_ino);**

**array\_set\_numeric(array， "mode"， sbuf->st\_mode);**

**array\_set\_numeric(array， "nlink"， sbuf->st\_nlink);**

**array\_set\_numeric(array， "uid"， sbuf->st\_uid);**

**array\_set\_numeric(array， "gid"， sbuf->st\_gid);**

**array\_set\_numeric(array， "size"， sbuf->st\_size);**

**array\_set\_numeric(array， "blocks"， sbuf->st\_blocks);**

**array\_set\_numeric(array， "atime"， sbuf->st\_atime);**

**array\_set\_numeric(array， "mtime"， sbuf->st\_mtime);**

**array\_set\_numeric(array， "ctime"， sbuf->st\_ctime);**

**/\* for block and character devices， add rdev，**

**major and minor numbers \*/**

**if (S\_ISBLK(sbuf->st\_mode) || S\_ISCHR(sbuf->st\_mode)) {**

**array\_set\_numeric(array， "rdev"， sbuf->st\_rdev);**

**array\_set\_numeric(array， "major"， major(sbuf->st\_rdev));**

**array\_set\_numeric(array， "minor"， minor(sbuf->st\_rdev));**

**}**

函数的后面的部分用于向目标数组中选择性地添加一些元素，这个要依赖于文件的特定成员以及/或者文件类型。如果成功，则返回 0：

**#ifdef HAVE\_STRUCT\_STAT\_ST\_BLKSIZE**

**array\_set\_numeric(array， "blksize"， sbuf->st\_blksize);**

**#endif /\* HAVE\_STRUCT\_STAT\_ST\_BLKSIZE \*/**

**pmode = format\_mode(sbuf->st\_mode);**

**array\_set(array， "pmode"， make\_const\_string(pmode， strlen(pmode)，& tmp));**

**/\* for symbolic links， add a linkval field \*/**

**if (S\_ISLNK(sbuf->st\_mode)) {**

**char \*buf;**

**ssize\_t linksize;**

**if ((buf = read\_symlink(name， sbuf->st\_size，& linksize)) != NULL)**

**array\_set(array， "linkval"，**

**make\_malloced\_string(buf， linksize， & tmp));**

**else**

**warning(ext\_id， \_("stat: unable to read symbolic link ‘%s’")，name);**

**}**

**/\* add a type field \*/**

**type = "unknown"; /\* shouldn’t happen \*/**

**for (j = 0， k = sizeof(ftype\_map)/sizeof(ftype\_map[0]); j < k; j++) {**

**if ((sbuf->st\_mode & S\_IFMT) == ftype\_map[j].mask) {**

**type = ftype\_map[j].type;**

**break;**

**}**

**}**

**array\_set(array， "type"， make\_const\_string(type， strlen(type)， & tmp));**

**return 0;**

**}**

stat() 函数的第三个参数之前没有讨论到。这个参数是可选的。如果存在，则会导致 do\_stat() 使用 stat() 调用而不是 lstat() 系统调用。这是通过这个函数指针来完成的：statfunc。statfunc 会被初始化为 lstat()（而不是 stat()），以此来获取文件信息，以防止文件是一个软链接。

但是，如果包含了第三个参数，statfunc 为设置为 stat() 函数的指针。下面是 do\_stat() 函数，它的开始也是一些变量声明以及参数检查：

**/\* do\_stat --- provide a *stat()* function for gawk \*/**

**static *awk*\_value\_t \***

**do\_stat(int nargs， *awk*\_value\_t \*result)**

**{**

***awk*\_value\_t file\_param， array\_param;**

**char \*name;**

***awk*\_array\_t array;**

**int ret;**

**struct stat sbuf;**

**/\* default is *lstat()* \*/**

**int (\*statfunc)(const char \*path， struct stat \*sbuf) = lstat;**

**assert(result != NULL);**

**if (nargs != 2 && nargs != 3) {**

**if (do\_lint)**

**lintwarn(ext\_id，\_("stat: called with wrong number of arguments"));**

**return make\_number(-1， result);**

**}**

后面接着的是实际的工作。函数先取得参数，接下来，它会取得文件的信息。如果调用的函数（lstat()或者stat()）返回一个错误，则代码设置 ERRNO 的值然后返回：

**/\* file is first arg， array to hold results is second \*/**

**if ( ! get\_argument(0， AWK\_STRING， & file\_param)**

**|| ! get\_argument(1， AWK\_ARRAY， & array\_param)) {**

**warning(ext\_id， \_("stat: bad parameters"));**

**return make\_number(-1， result);**

**}**

**if (nargs == 3) {**

**statfunc = stat;**

**}**

**name = file\_param.str\_value.str;**

**array = array\_param.array\_cookie;**

**/\* always empty out the array \*/**

**clear\_array(array);**

**/\* stat the file; if error， set ERRNO and return \*/**

**ret = statfunc(name， & sbuf);**

**if (ret < 0) {**

**update\_ERRNO\_int(errno);**

**return make\_number(ret， result);**

**}**

其他的冗长的工作由 fill\_stat\_array() 来完成，这个已经在前面出现过。当完成时，函数返回从 fill\_stat\_array() 函数中返回的值：

**ret = fill\_stat\_array(name， array， & sbuf);**

**return make\_number(ret， result);**

**}**

最后，有必须提供一些“胶水”来将新的函数装入 gawk，filefuncs 扩展也提供了一个 fts() 函数，这个函数在这里进行了省略（查看 ， ***在第 391 页***）。由于这个原因，有一个初始化函数：

**/\* init\_filefuncs --- initialization routine \*/**

**static *awk*\_bool\_t**

**init\_filefuncs(void)**

**{**

**...**

**}**

到目前为止，就差不多完成了。我们需要一个 awk\_ext\_func\_t 结构的数组用来装载每个一个函数到 gawk 中：

**static *awk*\_ext\_func\_t func\_table[] = {**

**{ "chdir"， do\_chdir， 1 }，**

**{ "stat"， do\_stat， 2 }，**

**#ifndef \_\_MINGW32\_\_**

**{ "fts"， do\_fts， 3 }，**

**#endif**

**};**

每个指针都必须有一个例程名为 dl\_load() 来载入所有要被载入的东西。最简单的方法是使用 gawkapi.h 中的 dl\_load\_func() 宏：

**/\* define the dl\_*load()* function using the boilerplate macro \*/**

**dl\_load\_func(func\_table， filefuncs， "")**

以上即是所有代码！

### 16.6.3 整合扩展

现在代码已经写在，但是需要能够在 gawk 解释器运行的时候，将其载入。首先，代码是需要编译的。假充函数在一个名为 filefuncs.c 的文件中，而 idir 是 gawkapi.h 头文件所在的位置。下面的步骤[[112]](#footnote-113)用来创建一个 GNU/Linux 共享库：

**$ gcc -fPIC -shared -DHAVE\_CONFIG\_H -c -O -g -Iidir filefuncs.c**

**$ gcc -o filefuncs.so -shared filefuncs.o**

库一旦生成，就可以通过 @load 关键字来进行载入：

**# file testff.*awk***

**@load "filefuncs"**

**BEGIN {**

**"pwd" | *getline* curdir # save current directory**

**close("pwd")**

**chdir("/tmp")**

**system("pwd") # test it**

**chdir(curdir) # go back**

**print "Info for testff.*awk*"**

**ret = stat("testff.*awk*"， data)**

**print "ret ="， ret**

**for (i in data)**

***printf* "data[\"%s\"] = %s\n"， i， data[i]**

**print "testff.*awk* modified:"，**

**strftime("%m %d %Y %H:%M:%S"， data["mtime"])**

**print "\nInfo for JUNK"**

**ret = stat("JUNK"， data)**

**print "ret ="， ret**

**for (i in data)**

***printf* "data[\"%s\"] = %s\n"， i， data[i]**

**print "JUNK modified:"， strftime("%m %d %Y %H:%M:%S"， data["mtime"])**

**}**

AWKLIBPATH 环境变量用来告知 gawk 到哪里去查看扩展（查看 ***，在第 382 页***）。我们将其设置为当前目录并执行程序：

**$ AWKLIBPATH=$PWD gawk -f testff.*awk***

**-| /tmp**

**-| Info for testff.*awk***

**-| ret = 0**

**-| data["blksize"] = 4096**

**-| data["devbsize"] = 512**

**-| data["mtime"] = 1412004710**

**-| data["mode"] = 33204**

**-| data["type"] = file**

**-| data["dev"] = 2053**

**-| data["gid"] = 1000**

**-| data["ino"] = 10358899**

**-| data["ctime"] = 1412004710**

**-| data["blocks"] = 8**

**-| data["nlink"] = 1**

**-| data["name"] = testff.*awk***

**-| data["atime"] = 1412004716**

**-| data["pmode"] = -rw-rw-r--**

**-| data["size"] = 666**

**-| data["uid"] = 1000**

**-| testff.*awk* modified: 09 29 2014 18:31:50**

**-|**

**-| Info for JUNK**

**-| ret = -1**

**-| JUNK modified: 01 01 1970 02:00:00**

## 16.7 gawk 发行版本中的例子扩展

这一节提供了一个简单的例子扩展的概况，例子与 gawk 版本一起发布。它们中的一些就是用于生产目的的（如 filefuncs，readdir，以及 inplace 扩展）。其他的主要是一些例子代码，以示范如何来使用扩展 API。

### 16.7.1 与文件相关的函数

filefuncs 扩展提供了不同的函数，其使用方法如下：

***@load "filefuncs"***

如何来装载扩展。

***result = chdir("/some/directory")***

chdir() 函数直接是 chdir() 系统调用的钩子函数，用来更改当前目录。如果成功则返回 0，如果错误则返回负值。在后面的情况还会更新 ERRNO 的值。

***result = stat("/some/path"， statdata [， follow])***

stat() 函数提供了 stat() 系统调用的钩子。如果成功，则返回 0，如果失败则返回负值。后后还会更新 ERRNO。

默认的情况下，它使用的是 lstat() 的系统调用。但是，如果传递了第三个参数，则会使用 stat()。

所有的情况下，它都会清除 statdata 数组。当调用成功时，stat() 会在 statdata 数组中填充从文件系统中返回的值。如下：

|  |  |  |
| --- | --- | --- |
| 下标 | Stat 中的域 | 文件类型 |
| "name" | The file name | All |
| "dev" | st\_dev | All |
| "ino" | st\_ino | All |
| "mode" | st\_mode | All |
| "nlink" | st\_nlink | All |
| "uid" | st\_uid | All |
| "gid" | st\_gid | All |
| "size" | st\_size | All |
| "atime" | st\_atime | All |
| "mtime" | st\_mtime | All |
| "ctime" | st\_ctime | All |
| "rdev" | st\_rdev | Device files |
| "major" | st\_major | Device files |
| "minor" | st\_minor | Device files |
| "blksize" | st\_blksize | All |
| "pmode" | 人可读的模式值字串，如 ls 打印的情况（如 “-rwxr-xr-x”） | All |
| "linkval" | 软链接的值。 | Symbolic links |
| "type" | 文件类型字串表示，"file"， "blockdev"， "chardev"，"directory"， "socket"， "fifo"，"symlink"， "door"， 或者 "unknown" 其中之一（并不是所有的系统都支持这些类型） | All |

***flags = or(FTS\_PHYSICAL， ...)***

***result = fts(pathlist， flags， filedata)***

遍历由 pathlist 提供的文件树，并填充 filedata 数组，如后所述。flags 是几个预定义值的或，也如后所述。如果没有错误，则返回 0，错误则返回 -1。

fts() 函数提供了 C 库中的 fts() 例程的钩子，用来目录结构进行遍历。它不会在流中一次返回一个文件的数据，而是将其信息填充到一个多维数组中。这个数组中包括在遍历目录结构时遇到的每个文件与目录的的数据。

参数如下：

***pathlist*** 文件名数组。使用的是元素的值，而不使用值索引。

***flags*** 这些是下面预定的一个或者多个常量的按位或。FTS\_LOGICAL 中 FTS\_PHYSICAL 必须要提供一个，否则 fts() 会返回一个错误，并设置 ERRNO。这些标志为：

FTS\_LOGICAL

执行的是“逻辑”上的文件遍历，在这个情况下针对链接文件返回的信息是被指向文件的信息，而不是链接文件本身。这个标志与 FTS\_PHYSICAL 是互斥的。

FTS\_PHYSICAL

执行一个“物理”的遍历，其中对于软连接文件的信息是软链接文件本身的信息。这个标志与 FTS\_LOGICAL 是互斥的。

FTS\_NOCHDIR

作为性能上的优化，C 库中的 fts() 例程会改变在目录结构中遍历文件时的目录，这个标志会禁用这个优化。

FTS\_COMFOLLOW

直接跟跟随 pathlist 中命名的软连接，不管是否设置了 FTS\_LOGICAL 标志。

FTS\_SEEDOT

默认的情况下，C 库中的 fts() 例程不会返回 .（dot）以及 .. （dot-dot）目录项。这个选项会使得 dot-dot 项会被包含进来。（扩展总是会包含 dot 的项，后面会有更多的说明。）

FTS\_XDEV 在遍历过程中，不跨越不同的的挂载的文件系统。

***filedata*** filedata 数组存储结果。fts() 首先会将其清除。然后会在 filedata 中为会在 pathlist 中的每个元素创建一个元素。索引即是 pathlist 中所给的目录或者文件的名字。这个索引下的元素本身是一个数组。存在两种情况：

*路径是一个文件*

在这种情况下，数组包含两个或者三个元素：

"path" 这个文件的全路径，在 pathlist 数组中是从“根”开始。

"stat" 这个元素自己是一个数组，包含与早前讨论的 stat() 函数在自己的 statdata 参数中填充的一样的信息。如果针对文件的系统调用 stat() 函数失败，则可能不存在这个元素。

"error" 如果发什么了某种错误，数组也包含一个名为 “error”的元素，这个元素是描述错误的字串。

*路径是一个目录*

在这种情况下，数组为每个目录生成一个元素。如果相应的项是文件，则元素与刚才描述的文件类型的项一样。如果项是目录，则元素是一个（递归地）描述子目录的元素。如果 FTS\_SEEDOT 在标志中进行了设置，则也会成在名字为 “..”的元素。这个元素也会是一个数组，包含了从 stat() 调用中提供的信息。

另外，也有一个元素，其索引为 “.”。这个元素是一个包含为文件提供的两个或者三个元素的数组。它们是，“path”，“stat”，“error”。

如果没有出错，fts() 函数返回 0，否则返回 -1。

提示：fts() 扩展并不是 C 库中的函数 fts() 完全的模拟，选择提供基于关联数组的接口，是在于在 awk 中使用起来更方便。这里缺少比较函数，因为 gawk 已经提供了非常强大的数组排序功能。尽管 像 fts\_read() 这样的接口本来应该提供，但是比起创建一个多维数组来表示文件的目录结构以及其信息来，它没有这么自然。

查看 gawk 发面版本中的 test/fts.awk 文件来获取使用 fts() 扩展函数的例子。

### 16.7.2 针对 *fnmatch()*的接口

这个扩展提供了 C 库接口 fnmatch() 函数，使用方法为：

***@load "fnmatch"***

如何来装载这个扩展。

***result = fnmatch(pattern， string， flags)***

成功时，返回值为 0，如果字串不匹配这个模式则返回 FNM\_NOMATCH，如果出错，则返回不同的非零值。

关于 fnmatch() 函数的一些其他信息，fnmatch 扩展则添加了一个常数（FNM\_NOMATCH），以及一个名为 FNM 的数组标志值。

fnmatch() 的参数有：

***pattern*** 要匹配的文件的通配符。

***string*** 文件名串

***flag*** 要么0，要么是 FNM 数组中多个值的相或

标志如下：

|  |  |
| --- | --- |
| 数组元素 | ***fnmatch() 所定义的对应标志*** |
| FNM["CASEFOLD"] | FNM\_CASEFOLD |
| FNM["FILE\_NAME"] | FNM\_FILE\_NAME |
| FNM["LEADING\_DIR"] | FNM\_LEADING\_DIR |
| FNM["NOESCAPE"] | FNM\_NOESCAPE |
| FNM["PATHNAME"] | FNM\_PATHNAME |
| FNM["PERIOD"] | FNM\_PERIOD |

例子如下：

**@load "fnmatch"**

**...**

**flags = or(FNM["PERIOD"]， FNM["NOESCAPE"])**

**if (fnmatch("\*.a"， "foo.c"， flags) == FNM\_NOMATCH)**

**print "no match"**

### 16.7.3 针对*fork()，wait()* 与 *waitpid()* 的接口

fork 扩展添加了三个函数，如下：

***@load "fork"***

如何来装入扩展。

***pid = fork()***

这个函数创建一个新的进程。返回值在子进程中返回为 0，在父进程中返回的是子进程的进程 ID，或者出错的情况下为 -1。在后面的情况，ERRNO 用来指示出错的原因。在子进程中，PROCINFO["pid"] 以及 PROCINFO["ppid"] 会被更新用以反映正确的值。

***ret = waitpid(pid)***

这个函数取数值参数，即要等待的进程的 ID。返回值是 waitpid() 系统调用的值。

***ret = wait()***

这个函数等待第一个子进程死亡。返回值是 wait() 系统调用的返回值。

没有相应的 exec() 函数。

下面是例子：

**@load "fork"**

**...**

**if ((pid = *fork()*) == 0)**

**print "hello from the child"**

**else**

**print "hello from the parent"**

### 16.7.4 启用文件就地编辑

inplace 扩展模拟 GNU sed 的 -i 选项，这个选项表示对每个输入文件进行“就地”编辑。它使用打包文件 inplace.awk，其中包含了要调用的扩展的适当文件：

**# inplace --- load and invoke the inplace extension.**

**@load "inplace"**

**# Please set INPLACE\_SUFFIX to make a backup copy. For example， you may**

**# want to set INPLACE\_SUFFIX to .bak on the command line or in a BEGIN rule.**

**# N.B. We call inplace\_*end()* in the BEGINFILE and END rules so that any**

**# actions in an ENDFILE rule will be redirected as expected.**

**BEGINFILE {**

**if (\_inplace\_filename != "")**

**inplace\_end(\_inplace\_filename， INPLACE\_SUFFIX)**

**inplace\_begin(\_inplace\_filename = FILENAME， INPLACE\_SUFFIX)**

**}**

**END {**

**inplace\_end(FILENAME， INPLACE\_SUFFIX)**

**}**

对于要处理的每一个常规文件，扩展重定向输出到一个临时文件中，其与原始的文件具有相同所有权以及访问权限。在处理了相应的文件后，扩展会恢复标准输出到原来的目标。如果 INPLACE\_SUFFIX 不为空串，则初始文件将后缀联合创建一个备份文件。最后，临时文件被重新命名为初始文件。

\_inplace\_filename 变量用来跟踪当前的文件名，这样不用在处理第一个文件时调用 inplace\_end()。

如果出现的错误，扩展会生成一个致命错误，以立即结束处理，不会对原始文件造成破坏。

这是一些简单的例子：

**$ gawk -i inplace ’{ gsub(/foo/， "bar") }; { print }’ file1 file2 file3**

为了保存原文件的一个备份复本，可以这么执行：

**$ gawk -i inplace -v INPLACE\_SUFFIX=.bak ’{ gsub(/foo/， "bar") }**

**> { print }’ file1 file2 file3**

### 16.7.5 字符与数值：ord()与chr()

ordchr 扩展添加了两个函数，名字为 ord() 与 chr()，如下：

***@load "ordchr"***

这里用来进行扩展装载。

***number = ord(string)***

返回 string 中第一个字符的数值值。

***char = chr(number)***

返回第一个字母是由数值（number）表示的字串。

这些函数由 Pascal 语言的同名函数借用而来。下面是例子：

**@load "ordchr"**

**...**

***printf*("The numeric value of 'A' is %d\n"， ord("A"))**

***printf*("The string value of 65 is %s\n"， chr(65))**

### 16.7.6 读取目录

readdir 扩展添加了一个目录的输入分析器，其使用如下：

**@load "readdir"**

当使用这个扩展时，它不会跳过在命令行中的目录（或者也不会在 getline 中跳过），相反它们会被读取，每一个项作为一个记录。

记录包含三域。头两个域是 inode 号与文件名，文件名由 / 进行分隔。在目录项中包含文件类型的系统中，记录会有第三个域（这个域也由 / 进行分隔，这是一个由单个字母表示的文件的类型。这些字母以及文件的相应的类型，在表 16.2 中。

|  |  |
| --- | --- |
| 字母 | 文件类型 |
| b | 块设备 |
| c | 字符设备 |
| d | 目录 |
| f | 常规文件 |
| l | 符号连接 |
| p | 命令管道（FIFO） |
| s | Socket |
| u | 其他（未知） |

表 16.2：由 readdir 扩展返回的文件类型

在没有文件系统的系统中，第三个域总是‘u’。

提示：在 GNU/Linux 系统中，有一些文件系统不支持 d\_type 项（查看 readdir(3) 手册页），因此文件类型也总是 ‘u’。你可以使用 filefuncs 扩展来调用 stat() 来取得正确的类型信息。

下面是例子：

**@load "readdir"**

**...**

**BEGIN { FS = "/" }**

**{ print "file name is"， $2 }**

### 16.7.7 倒转输出

revoutput 扩展添加了一个简单的输出包装器，包含器会将每个输出行的字符进行倒转。这个扩展的主要目的是来演示如何写一个输出包装器，尽管它可能有点无厘头。下面是例子：

**@load "revoutput"**

**BEGIN {**

**REVOUT = 1**

**print "don’t panic">"/dev/stdout"**

**}**

这个例子中的输出为 ‘cinap t'nod’。

### 16.7.8 双路 I/O 例子

revtwoway 扩展添加了一个简单的双路处理器，用来将初 awk 程序读取回来的每一行的字母进行倒转。它的目录是演示如何来写一个双路处理串，这也许也很无厘头。下面的例子示范如何来使用：

**@load "revtwoway"**

**BEGIN {**

**cmd = "/magic/mirror"**

**print "don’t panic" |& cmd**

**cmd |&*getline* result**

**print result**

**close(cmd)**

**}**

这个程序的输出是 ‘cinap t'nod’。

### 16.7.9 输出与恢复数组

rwarray 扩展添加了两个函数，名字为 writea() 与 reada()，如下：

***@load "rwarray"***

如何来装载扩展。

***ret = writea(file， array)***

这个函数有一个字串参数，这个参数是要 dump 数组到文件的名字，而数组则为第二个参数。writea() 可以处理数组之数组。如果成功则返回 1，否则返回 0。

***ret = reada(file， array)***

reada() 是 writea() 的相反的版本，它从由第一个参数命令的文件中读取，并将其填充到作为第二个参数的数组。它首先会将数组清空。在这里也是成功时返回 1，否则返回 0。

由 reada() 创建的数组与 writea() 所写的数组在数据相同的情况下是相同的。但是，由于实现的问题，重建的数组的遍历顺序有可能不同于最开始的数组。在 awk 中数组的遍历顺序默认是未定义的，这在技术上来讲并不是什么问题。如果你要保证一个特定的遍历顺序，使用 gawk 中的数组排序特定来处理（查看 ， ***在第 298 页***）。

文件中包含的是二进制数据。所有的整数值都以网络字节序写入的。但是，双精度的浮点数是使用的本地进制数据。所以，只包含字串数据的数组理论上在系统上是以字节序的方式进行输出的，但是在另一个系统上可能是不一样的，但是也没有试过。

下面是例子：

**@load "rwarray"**

**...**

**ret = writea("arraydump.bin"， array)**

**...**

**ret = reada("arraydump.bin"， array)**

### 16.7.10 读取整个文件

readfile 扩展添加了一个函数，名字为 readfile()，以及一个输入分析器：

***@load "readfile"***

如何来使用扩展。

***result = readfile("/some/path")***

参数是要读取的文件的名字。返回值是一个字串，其中包含所请求文件的所有内容。如果错误，函数返回空串，并设置 ERRNO 的值。

***BEGIN { PROCINFO["readfile"] = 1 }***

另外，扩展添加了一个输入分析器，如果 PROCINFO["readfile"] 存在时则会被激活。当被激活时，每个文件都以整个的内容返回为 $0，RT 被设置为空串。

下面的例子：

**@load "readfile"**

**...**

**contents = readfile("/path/to/file");**

**if (contents == ""&& ERRNO != "") {**

**print("problem reading file"， ERRNO) >"/dev/stderr"**

**...**

**}**

### 16.7.11 扩展的时间函数

time 扩展添加了两个函数，名字为 gettimeofday() 以及 sleep，如下：

***@load "time"***

如何来使用扩展。

***the\_time = gettimeofday()***

返回从 1970-01-01 UTC 以来的消逝的秒数，类型为浮点数。如果时间在平台上不可得，则返回 -1 并设置 ERRNO。返回的时间应该有次秒的精度，但是实际的精度在不同的平台上是不同的。如果标准的 C gettimeofday() 系统调用在平台上是可用的，则它只是简单地返回这个函数的值。否则，如果在 MS-Windows 平台上，它返回 ***GetSystemTimeAsFileTime()*** 函数返回的值。

***result = sleep(seconds)***

尝试休眠 seconds 秒。如果 seconds 为负数，或者尝试休眠失败，则返回 -1 并且设置 ERRNO。否则在休眠了指定的时间后返回0。注意，seconds 可以是一个浮点数（非整数）。实现细节是：依赖于平台的可获得性，这个函数会尝试使用 nanosleep() 或者 select 来实现延迟。

### 16.7.12 API 测试

testext 扩展执行扩展 API 中没有在被其他例子测试到的部分。extension/testext.c 文件包含扩展的 C 代码以及 awk 测试代码，这些代码注释在执行测试的 C 注释中。测试框架提取 awk 代码来执行测试。查看源代码来获取更多的信息。

## 16.8 gawkextlib 项目

gawkextlib 项目提供了很多的 gawk 扩展，包含用于处理 XML 文件的扩展，这是由最开始的 xgawk（XML gawk）项目演化过来的。

在写本书的时候，一共有七个扩展：

* errno 扩展
* GD graphics 库扩展
* PDF 扩展
* PostgreSQL 扩展
* MPFR 库扩展（这提供了访问很多的 MPFR 函数，这些函数在 gawk 的本地 MPFR 是不被支持的）
* Redis 扩展
* XML 分析器扩展，这个扩展使用 Expat XML 分析库

你可以查看 gawkextlib 项目的代码，这些代码可以通过 Git 分布式源代码扩展系统获取。相应的命令为：

**git clone git://git.code.sf.net/p/gawkextlib/code gawkextlib-code**

为了构建 XML 扩展 ，你需要在你的系统已经安装好了 Expat XML 分析库。

另外，你必须还要有 GNU Autotools 安装在系统上（Autoconf，Automake，Libtool 以及 GNU gettext）。

一个简单的构建与测试 gawkextlib 的过程如下。首先是构建与安装 gawk：

**cd .../path/to/gawk/code**

**./configure --prefix=/tmp/newgawk *Install in /tmp/newgawk for now***

**make && make *check Build and check that all is OK***

**make install *Install gawk***

接下来，到 http://sourceforge.net/projects/gawkextlib/files 中下载 gawkextlib 以及任意的你想要构建的扩展。站点中的 README 文件说明了如何来构建代码。如果你将 gawk 安装在一个非标准的位置，你还要指定 ./configure --with-gawk=/path/to/gawk 来指定它。你可能需要使用 sudo 工具来安装 gawk 与gawkextlib，这视你所在的系统的工作方式。

如果你写了一个扩展，而希望与其他的 gawk 用户一起分离，可以考虑通过 gawkextlib 项目来进行分发。查看这个项目的网站来获取更多的信息。

## 16.9总结

* 你可以为 gawk 用 C 或者 C++ ，并利用由 gawk 开发者所定义的应用程序接口（API）来写扩展（有时也叫插件）。
* 扩展必须有一个与 GNU 通用许可证协议（GPL）兼容的许可证，而它们必须通过声明一个名为 plugin\_is\_GPL\_compatible 的变量来进行断言。
* 在 gawk 与扩展之间的通信是双向的，gawk 将一个结构传递给扩展，这个结构中包含了各种数组成员以及函数指针。而扩展可以通过 gawk 提供的函数来调用 gawk 以实现特定的任务。
* 这些工作其中之一是“注册”名字与新的 awk 层功能实现到 gawk 中，这些实现名字为 do\_XXXX()，而这个函数实现的是 awk 层的函数 XXXX()。
* API 定义在头文件 gawkapi.h 中。你的源代码中，必须在包含这个头文件之前包含很多的标准头文件。
* API 函数指针可提供下面的操作：
* 分配，重新分析以及释放内存
* 注册函数（你可以注册扩展函数，退出回调函数，版本字串，输入分析器，输出包装器以及双路处理器）
* 打印致命错误，告警以及 ‘lint’警告信息
* 更新 ERRNO 或者重置它
* 访问参数，其中包括将一个未定义的参数转换为数组
* 符号表访问（返回全局变量，创建或者更改变量）
* 创建并翻译缓存值，这会给使用多个变量值提供一种高效的方法，且获取巨大的性能回报
* 操作数组（返回，添加，删除或者修改元素，获取数组元素个数，创建新数组，清除数组，扁平化数组来使用简单的 C 风格的循环来访问其所有的索引与元素。
* API 定义了很多的标准数组类型用来表示 awk 的值，数组元素以及数组。
* API 提供了方便的函数用来构建值。也提供了内存管理函数来保证 gawk 分配的内存与由扩展分配的内存的兼容性。
* 所有的从 gawk 传递给扩展的内存必须被扩展当成是只读的。
* 所有的由扩展传递给 gawk 的内存必须通过 API 的内存分配函数来分配。gawk 会管理这些内存并在合适的时间释放它。
* API 提供了 gawk 运行时的版本信息，这样扩展可以确定其是否与载入扩展的 gawk 兼容。
* 最容易的开启一个新扩展的方式是复杂这一章中的样板代码。gawkapi.h 中的宏可以简化这个工作。
* gawk 发布版包含了很多小的，很有用的例子扩展。gawkextllib 项目包含了几个更大的扩展。如果你希望写一个扩展并将其分享给 gawk 的用户社区，gawkextlib 可以用于这样的目的。

## 16.10练习

1. 添加一个函数来实现系统调用，如 chown()，chmoe()，以及 umask() 来执行文件操作。这些函数添加到 ， ***在第 384 页***，中的扩展中。
2. 写一个输入分析器，在输入是来自于“终端”设备时则打印一个提示。你可以使用 isatty() 函数以获取输入文件是否是一个终端。（提示：这个函数调用起来通常很慢，最好只调用一次。）提示的内容应该一个可被 awk 层代码设置的变量。你可以将提示写到标准错误中。但是为了最好的结果，是打开一个在 /dev/tty 中的一个文件描述符（或者文件指针）并在其上打印提示，因为有可能标准错误可能会被重定向。

为什么标准错误比标准输出是用来输出提示的更好选择？你要替换成什么样的读取机制，是一次一条记录还是一次读取多个原始字节？

1. （难）如你何在 gawk 中来提供一个名字空间（namespace），这样不同扩展中的函数名字不会相互冲突？如果你想出了一个好方案，请与 gawk 的维护者联系并告知这个方案。
2. 写一个包装脚本来提供一个与 ‘sed -i’相似的接口，以实现在 ， ***在第 395 页***，中的“inplace”扩展。

**Part IV:**

Appendices

附录

# 附录 A awk 语言的演化

本书描述的 awk 的 GNU 版本，这个版本的 awk 是符号 POSIX 标准的。很多的老 awk 用户使用 Unix Version 7 版本的 awk 开始学习 awk 语言。（这个版本的实现是 Berkeley Unix 4.3 至Reno中的 awk 版本的基础。后续的 Berkeley Unix 版本，并且有一段时间，一些由 4.4 BSD-Lite 派生出来 Unix 版本使用不同的版本 gawk 来作为这些系统上的 awk。）这一简单地描述 awk 语言的演化过程，并对书中相应的章节做了引用 ，这样你可以看到更多的信息。

## A.1在 V7 与 SVR3.1 之间的变化

awk 语言在 Unix Version 7（1978）与 System V Release 3.1（1987）中作为通用常用工具第一个新版本之间有非常大的变化。这节对其中的变化进行归纳，并有相应的参考，这样可以获取进一步的信息：

* 需要用 ‘；’来分隔一行中的规则（查看 ， ***在第 31 页***）。
* 用户自定义函数与 return 语句（查看 ， ***在第 204 页***）。
* delete 语句（查看 ， ***在第 175 页***）。
* do-while 语句（查看 ， ***在第 148 页***）。
* 内置函数 atoan2()，cos()，sin()，rand() 与 srand()（查看 ， ***在第 183 页***）。
* 内置函数 gsub()，sub() 与 match()（查看 ， ***在第 184 页***）。
* 内置函数 close() 与 system()（查看 ， ***在第 194 页***）。
* ARGC， ARGV， FNR， RLENGTH， RSTART， 与 SUBSEP 预定义变量（查看 ， ***在第 155 页***）。
* 可赋值的 $0（查看 ， ***在第 69 页***）。
* 使用三元操作符‘?:’的条件表达式（查看 ， ***在第 132 页***）。
* 在 for 语句之外的 ‘indx in array’表达式（查看 ， ***在第 168 页***）。
* 幂操作符‘^’（查看 ， ***在第 119 页***）与其赋值操作符形式‘^=’（查看 ， ***在第 122 页***）。
* 与 C 兼容的操作符优先级，这个与一些旧的 awk 程序不同（查看 ）。
* 正则表达式可以作为 FS 的值（查看 ， ***在第 71 页***）以及作为 split() 函数（查看 ， ***在第 184 页***）的第三个参数，而不是使用 FS 的第一个字符。
* 作为 ‘~’ 与 ‘!~’ 操作符操作数的动态正则表达式（查看 ， ***在第 58 页***）。
* ‘\b’，‘\f’，与 ‘\r’转义序列（查看 ， ***在第 51 页***）。
* getline 函数（查看 ， ***在第 83 页***）输入重定向。
* 多 BEGIN 与 END 规则（查看 ， ***在第 141 页***）。
* 多维数组（查看 ， ***在第 176 页***）。

## A.2 在SVR3.1 与 SVR4 之间的变化

System V Release 4（1989）版本的 Unix awk 添加了这些特性（有一些来自于 gawk）：

* ENVIRON 数组（查看 ， ***在第 155 页***）。
* 多个 -f 选项（查看 ， ***在第 35 页***）。
* -v 选项用于在程序执行前进行变量赋值（查看 ， ***在第 35 页***）。
* -- 用于表示命令行选项的结束。
* ‘\a’，‘\v’ 与 ‘\x’ 转义字符（查看 ， ***在第 51 页***）
* 内置函数 srand() 有确定的返回值（查看 ， ***在第 183 页***）。
* toupper() 与 tolower() 内置字串函数，用于大小定转换（查看 ， ***在第 184 页***）。
* 对于 printf 中的‘%c’格式控制字串有更清晰的规范（查看 ， ***在第 98 页***）。
* 在 printf 与 sprintf() 的参数列表中，可以动态控制域的宽度与精度（“%\*.\*d”）（查看 ， ***在第 98 页***）。
* 正则表达式常量的使用，如 /foo/ 作为一个表达式，这个与匹配操作符，如‘***$0 ~ /foo/***’是一样的的（查看 ， ***在第 114 页***）。
* 在命令行变量赋值中处理转义序列（查看 ， ***在第 116 页***）。

## A.3 在 SVR4 与 POSIX awk 之间的变化

awk 命令语言与工具的 POSIX 标准引入了下面变量：

* 使用 -W 特定于实现的选项（查看 ， ***在第 35 页***）。
* 使用 CONVFMT 用于控制数值转换为字串（查看 ， ***在第 117 页***）。
* 数值字串的概念并引入与更严格的比较规则（查看 ， ***在第 127 页***）。
* 使用预定义的变量作为函数参数名被禁用（查看 ， ***在第 204 页***）。
* 很多之前没有文档的语言特性有了更完备的文档。

在 2012 年，很多之前很多年都以扩展的方式取得的特性最终被加入到 POSIX 标准中。它们是：

* fflush() 内置函数用来刷出输出（查看 ， ***在第 194 页***）。
* nextfile 语句（查看 ， ***在第 154 页***）
* 通过 ‘delete array’就可以删除整个数组（查看 ， ***在第 175 页***）。

查看 ，***在第 414 页***，来获取还没有被接受为 POSIX 标准的扩展列表。

2008 年的 POSIX 标准可以在线上找到：http://www.opengroup.org /onlinepubs/9699919799/

## A.4 Brian Kernighan awk 中的扩展

Brian Kernighan 版本的 awk 可以通过它的主页来获取（查看 ， ***在第 435 页***）。

这一节描述的是来自于他的 awk 版本的通用扩展：

* ‘’ 与 ‘’ 操作符（查看 **， *在第 119 页*** 与 ， ***在第 122 页***）。
* func 用于 function 的缩写（查看 ， ***在第 204 页***）。
* fflush() 内置函数用来刷出缓存的输出（查看 ， ***在第 194 页***）

查看 ，***在第 414 页***，来获取在它的 awk 中可以获取的扩展。

## A.5 不在 POSIX awk 中的 gawk 扩展

awk 的 GNU 实现，gawk 添加了大量的特性。它们在使用 --traditional 或者 --posix 选项的时候可以被禁用（查看 ， ***在第 35 页***）。

很多的特性已经有一些年头了，这一节对这些附加的特性进行描述，这些特性使用在现在的 gawk 版本中，但是超出 POSIX 标准。

* 额外的预定义变量：
  + ARGIND， BINMODE， ERRNO， FIELDWIDTHS， FPAT， IGNORECASE， LINT， PROCINFO， RT， 与 TEXTDOMAIN 变量（查看 ， ***在第 155 页***）。
* Special files in I/O redirections:
* I/O 重定向中的特殊文件：
  + /dev/stdin， /dev/stdout， /dev/stderr， 与 /dev/fd/N 特殊文件名（查看 ， ***在第 106 页***）。
  + /inet， /inet4， 与 ***‘/inet6’*** 等特殊文件用于使用‘|&’ TCP/IP 网络通信，来指定使用哪个版本的协议（查看 ， ***在第 306 页***）。
* 语言中的更改与/或者添加：
  + ‘\x’转义序列（查看 ， ***在第 51 页***）。
  + 完全地支持 POSIX 与 GNU 正则表达式（查看 ***， 在第 50 页***）。
  + FS 与 split() 的第三个参数可以为空串（查看 ， ***在第 74 页***）。
  + 可以设置 RS 为正则表达式（查看 ， ***在第 63 页***）。
  + 可以在 awk 程序源代码中使用八进制与十六进制（查看 ， ***在第 113 页***）。
  + 使用‘|&’操作符来与并程进行双向 I/O 通信（查看 ， ***在第 304 页***）。
  + 间接函数调用（查看 ， ***在第 213 页***）。
  + 在命令行上输入目录会导致警告产生，但会跳过（查看 ， ***在第 91 页***）。
* 新的关键字：
  + BEGINFILE 与 ENDFILE 特殊模式（查看 ， ***在第 143 页***）。
  + switch 语句（查看 ， ***在第 150 页***）。
* 对标准 awk 函数的改变：
  + close() 函数的第二个可选参数可以关闭到并程的双向管道的一端（查看 ， ***在第 304 页***）。
  + 使用 --posix 时，gsub() 与 sub() 与 POSIX 兼容。
  + length() 函数接受一个数组作为参数并返回数组中元素的个数（查看 ， ***在第 184 页***）。
  + match() 函数的第三个可选参数用于捕获在正则表达式中的文本匹配子表达式（查看 ， ***在第 184 页***）。
  + 在 print 格式中的位置指定符用来简化翻译（查看 ， ***在第 315 页***）。
  + split() 函数的额外可选的第四个参数，是用来存放域分隔文本的数组（查看 ， ***在第 184 页***）。
* gawk 中的额外函数：
  + ***gensub()***， ***patsplit()***， 与 ***strtonum() 使得文本操作更加强大（查看*** ， ***在第 184 页）。***
  + asort() 与 asorti() 函数用来对数组排序（查看 ， ***在第 298 页***）。
  + ***mktime()***， ***systime()***， 与 ***strftime() 用来处理时间戳（查看*** ， ***在第 197 页）。***
  + ***and()***， ***compl()***， ***lshift()***， ***or()***， ***rshift()***， 与 ***xor()*** 函数用来处理位操作（查看 ， ***在第 201 页***）。
  + isarray () 函数用来检查一个变量是否为数组（查看 ， ***在第 203 页***）。
  + ***bindtextdomain()***， ***dcgettext()***， 与 ***dcngettext()*** 函数用于国际化（查看 ， ***在第 313 页***）。
* 命令行中的改变与/或者添加
  + AWKPATH 环境变量用来指定 -f 命令选项的搜索路径（查看 ， ***在第 35 页***）。
  + AWKLIBPATH 环境变量用来指定 -l 命令行选项的搜索路径（查看 ， ***在第 35 页***）。
  + -b， -c， -C， -d， -D， -e， -E， -g， -h， -i， -l， -L， -M， -n， -N， -o， -O， -p， -P， -r， -S， -t， 与 -V 短选项。同时也可以使用 GNU 风格的长选项。这些选项以 -- 开始，--assign， --bignum， --characters-as-bytes，--copyright， --debug， --dump-variables， --exec， --field-separator，--file， --gen-pot， --help， --include， --lint， --lint-old， --load，--non-decimal-data， --optimize， --posix， --pretty-print， --profile，--re-interval， --sandbox， --source， --traditional， --use-lc-numeric， 与 --version 是相应的长选项（查看 ， ***在第 35 页***）。
* 支持下面的下面过时系统的被从 gawk 4.0 版本的代码与文档中移除。
  + Amiga
  + Atari
  + BeOS
  + Cray
  + MIPS RiscOS
  + MS-DOS with the Microsoft Compiler
  + MS-Windows with the Microsoft Compiler
  + NeXT
  + SunOS 3.x， Sun 386 (Road Runner)
  + Tandem (non-POSIX)
  + Prestandard VAX C compiler for VAX/VMS
  + GCC for VAX and Alpha has not been tested for a while.
* 支持下面过时系统从 gawk 4.1 版本中被移除了。
  + Ultrix

## A.6 gawk 特性历史

这一小节描述存在 gawk 但是不在 POSIX awk 中的特性，安它们加入到 gawk 中的顺序进行排序。

gawk 2.10 版本引入了下面的特性：

* AWKPATH 环境变量用于指针 -f 命令行选项指定搜索路径（查看 ， ***在第 35 页***）。
* IGNORECASE 变量及其效果（查看 ， ***在第 60 页***）。
* /dev/stdin， /dev/stdout， /dev/stderr 与 /dev/fd/N 特殊文件名（查看 ， ***在第 106 页***）。

gawk 2.12 版本引入了下面的特性：

* FIELDWIDTHS 变量与其效果（查看 ， ***在第 77 页***）。
* systime() 与 strftime() 内置函数用来获取及打印时间戳（查看 ， ***在第 197 页***）。
* 额外的命令选项（查看 ， ***在第 35 页***）：
  + -W lint 选项用于提供源代码与运行时的错误与可移植性检测。
  + -W 兼容选项用来关闭 GNU 扩展。
  + -W posix选项用来实现完全的 POSIX 兼容。

2.14 版本的 gawk 引入了下面的特性：

* nextfile 语句用来跳到下一个数据文件（查看 ， ***在第 154 页***）。

2.15 版本 gawk 引入了下面的特性：

* New variables (see ， ***在第 155 页***):
* 新的变量（查看 ， ***在第 155 页***）：
  + ARGIND 用来跟踪 ARGV 中的 FILENAME 的移动。
  + ERRNO 在 getline 返回 -1 或者 close() 失败时包含了系统的错误信息。
* /dev/pid， /dev/ppid， /dev/pgrpid， 与 /dev/user 特殊文件名，这些已经被移除。
* 可以使用一次 ‘delete array’来删除数组中的所有元素（查看 ， ***在第 175 页***）。
* 命令行选项变化 （查看 ， ***在第 35 页***）：
  + The ability to use GNU-style long-named options that start with --.
  + 可以使用 GNU 风格的以 -- 开始的长选项。
  + 通过 --source 选项来混合命令行与库文件源代码。

3.0 版本的 gawk 引入了下面的特性：

* 新的或者变化了的变量：
  + IGNORECASE 已经变化，现在可以用于字串比较以及正则表达式操作（查看 ， ***在第 60 页***）。
  + RT 其中包含了输入中匹配 RS 的文本（查看 ， ***在第 63 页***）。
* 完全支持 POSIX 与 GNU 正则表达式（查看 ***，在第 50 页***）。
* gensub() 函数可以实现更强大的文本操作（查看 ， ***在第 184 页***）。
* strftime() 函数需要一个默认的时间格式，可以不用参数就可以调用（查看 ， ***在第 197 页***）。
* FS 与 split() 函数的第三个参数可以是空串（查看 ， ***在第 74 页***）。
* RS 可以是一个正则表达式（查看 ， ***在第 63 页***）。
* nextfile 语句变成 nextfile（查看 ， ***在第 154 页***）。
* 来自于 BWK awk 的 fflush() 函数（当时在 Bell Laboratories，查看 ， ***在第 194 页***）。
* 新的命令行选项：
  + --lint-old 选项来警告在初始的 Version 7 Unix 版本中的 awk 中不支持的结构（查看 ， ***在第 404 页***）。
  + 来自于 BWK awk 的 -m 选项。（Brian 那个时候还是在 Bell Laboratories）之后从他的 awk 与 gawk 中都移除了。
  + --re-interval 选项用来提供在正则表达式中提供区间表达式（查看 ， ***在第 53 页***）。
  + -- traditional 选项添加成为 --compat 更好的名字（查看 ， ***在第 35 页***）。
* 使用 GNU Autoconf 来控制配置过程（查看 ， ***在第 423 页***）。
* 支持 Amiga。这已经被移除了。

3.1 版本的 gawk 引入了下面的特性：

* 新的变量（查看 ， ***在第 155 页***）：
  + BINMODE ，来自于非 POSIX 系统，可以用于二进制 I/O 输入输出文件（查看 ， ***在第 428 页***）。
  + LINT 用来动态地控制 lint 告警。
  + PROCINFO 是提供了进程相关信息的数组。
  + TEXTDOMAIN 用来设置程序的国际化文件域（查看 ***，在第 311 页***）。
* 可以在 awk 程序源代码中使用八进制与十六进制（查看 ， ***在第 113 页***）。
* 使用 ‘|&’来与并程进行双路 I/O（查看 ， ***在第 304 页***）。
* 使用 ‘|&’以及 /inet 特殊文件来进行 TCP/IP 网络通信（查看 ， ***在第 306 页***）。
* close() 第二个可选参数可以用于关闭到并程的双向管道的一端（查看 ， ***在第 304 页***）。
* match() 函数的第三个可选参数用于捕获正则表达式中的文本匹配子表达式（查看 ， ***在第 184 页***）。
* printf 格式中的位置指定符用于使国际化更为简单（查看 ， ***在第 315 页***）。
* 多个新的内置函数：
  + asort() 与 asorti() 函数用于排序数组（查看 ， ***在第 298 页***）。
  + ***bindtextdomain()***， ***dcgettext()*** 与 ***dcngettext()*** 函数用于国际化（查看 ， ***在第 313 页***）。
  + ***extension()*** 函数以及动态添加新的内置函数（查看 ***，在第 221 页***）。
  + mktime() 函数用于创建时间戳（查看 ， ***在第 197 页***）。
  + ***and()***， ***or()***， ***xor()***， ***compl()***， ***lshift()***， ***rshift()***， 与 ***strtonum()*** 函数（查看 ， ***在第 201 页***）。
* 支持‘next file’为两个单词的语句已经完全移除（查看 ， ***在第 154 页***）。
* 额外的命令行选项（查看 ， ***在第 35 页***）：
  + --dump-variables 选项用于打印所有的全局变量的列表。
  + --exec 选项用于 CGI 脚本。
  + --gen-po 命令选项以及前置下划线的使用来标志要被翻译的字串（查看 ， ***在第 315 页***）。
  + --non-decimal-data 选项可以输入非十进制数（查看 ， ***在第 297 页***）。
  + --profile 选项以及 pgawk （gawk 的 profiling 版本）用于产生 awk 程序的执行统计（查看 ， ***在第 307 页***）。
  + --use-lc-numeric 选项强制 gawk 使用本地语言的十进制数小数点来分析输入数据（查看 ， ***在第 117 页***）。
* 使用 GNU Automake 来辅助配置过程的标准化（查看 ， ***在第 423 页***）。
* 使用 GNU 的 gettext 来作为 gawk 自己的消息输出（查看 ， ***在第 319 页***）。
* BeOS 支持。后面被移除了。
* Tandem 支持。后面被移除了。
* Atari 移植版本不被官方支持后面整个地移除了。
* 源代码改成使用 ISO C 标准风格的函数定义。
* POSIX 兼容 sub() 与 gsub()（查看 ， ***在第 192 页***）。
* length() 函数被扩展为接受数组参数，并返回数组中元素的个数（查看 ， ***在第 184 页***）。
* strftime() 函数需要第三个参数来启用打印 UTC 时间（查看 ， ***在第 197 页***）。

gawk 4.0 版本引入了下面的特性：

* 添加的变量：
  + FPAT 可以让你指定一个正则表达式来匹配域，而不是匹配域分隔符（查看 ， ***在第 79 页***）。
  + 如果存在 PROCINFO["sorted\_in"]，则 ‘***for(iggy in foo)***’循环会在循环开始之前对索引排序。这个元素的值提供在循环遍历开始之前控制如何来对索引来进行排序（查看 ， ***在第 171 页***）。
  + PROCINFO["strftime"] 用于保存 strftime() 的默认格式（查看 ， ***在第 197 页***）。
* 移除特殊文件 /dev/pid， /dev/ppid， /dev/pgrpid 与 /dev/user。
* 通过 /inet6 特殊文件来支持 IPV6。/inet4 强制使用 IPV4，而 /inet 则使用系统默认的协议，一般是 IPV4（查看 ， ***在第 306 页***）。
* 在正则表达式中使用 ‘\s’ 与 ‘\S’转换序列（查看 ， ***在第 59 页***）。
* 区间表达式成为默认正则表达式的一部份（查看 ， ***在第 53 页***）。
* 即使使用 --traditional 情况下，也可以使用 POSIX 字符类型（查看 ， ***在第 53 页***）。
* break 与 continue 在循环外不再有效，即使使用 --traditional 选项（查看 ，***在第 151 页***，以及 ， ***在第 152 页***）。
* fflush() ，nextfile，与 ‘delete array’在使用 --posix 或者 --traditional 选项下也是允许的，因此它们已经是 POSIX 标准的一部分。
* asort() 与 asorti() 函数有可选的第三个参数，用于指定如何来排序（查看 ， ***在第 184 页***）。
* fflush() 的行为已经改成与 BWK awk 以及 POSIX 匹配，现在 ‘fflush()’与 ‘fflush("")’会刷出所有的重定向输出（查看 ， ***在第 194 页***）。
* isarray() 函数可以用于识别某个项是否为数组，这样可以来用于进行数组之数组的遍历（查看 ， ***在第 203 页***）。
* patsplit() 函数可以有与 FPAT 一样的分割能力（查看 ， ***在第 184 页***）。
* split() 函数的第四个可选参数是用于存放分割符的数组（查看 ， ***在第 184 页***）。
* 数组之数组（查看 ， ***在第 178 页***）。
* BEGINFILE 与 ENDFILE 特定规则（查看 ， ***在第 143 页***）。
* 间接函数调用（查看 ， ***在第 213 页***）。
* switch/case 默认情况下是启用的（查看 ， ***在第 150 页***）。
* 改变的命令行选项（查看 ， ***在第 35 页***）：
  + -b 与 --characters-as-bytes 选项用来防止 gawk 将输入当成多字节字串来处理。
  + 多余的 --compat，--copyleft 与 --usage 长选项已经被移除。
  + --gen-po 选项最后被改名为 --gen-pot。
  + --sandbox 选项可以禁用特定的特性。
  + 所有的长选项都需要有相应的短选项，以用于 ‘#!’脚本。
* 在命令行上输入了目录名现在会产生告警，而不是致命错误。除非使用了 --posix 或者 --traditional 选项（查看 ， ***在第 91 页***）。
* 重写了 gawk 区间表达式，引入了 dgawk 的调试器，并增强了一些性能（查看***，在第 320 页***）。
* 根据 GNU 编码标准，动态扩展必须要定义一个全局符号来指示它们是使用 GPL 兼容的协议（查看 ， ***在第 349 页***）。
* 在 POSIX 模式下，使用 strcoll()/wcscoll() 来进行字串比较（查看 ， ***在第 130 页***）。
* 用于进行原生 Socket 通信的选项被移除了，因为它从来没有被实现（查看 ， ***在第 306 页***）。
* ‘[d-h]’形式的范围会被当成在 C 语言设置中一样对待，不管使用的是什么样的正则表达式，即使使用了 --posix（查看 ， ***在第 414 页***）。
* 对下面系统的支持被移除了：
  + Atari
  + Amiga
  + BeOS
  + Cray
  + MIPS RiscOS
  + MS-DOS with Microsoft Compiler
  + MS-Windows with Microsoft Compiler
  + NeXT
  + SunOS 3.x， Sun 386 (Road Runner)
  + Tandem (non-POSIX)
  + Prestandard VAX C compiler for VAX/VMS

4.1 版本的 gawk 引入了下面的特性：

* 新加了三个变量：SYMTAB，FUNCTAB，以及 PROIFO["identifiers"]（查看 ， ***在第 158 页***）。
* gawk，pgawk 与 dgawk 被整合为一，并命名为 gawk。因为仅是命令行选项有变化。
* 改变的命令行（查看 ， ***在第 35 页***）：
  + -D 选项用来调用调试器。
  + -i 与 --include 选项用于载入 awk 库文件。
  + -l 与 --load 选项用于载入编译好的动态扩展。
  + -M 与 --bignum 选项用于启用 MPFR。
  + -o 选项用于简洁打印。
  + -p 选项用于性能统计。
  + 移除了 -R 选项被。
* 复用 MPFR 来支持高精度算术计算。（查看 ***，在第 336 页***）。
* ***and()***， ***or()*** 与 ***xor()*** 函数更改成可以接受任意数量的参数，最小为两个（查看 ， ***在第 201 页***）。
* 动态扩展接口完全全进行了重构（查看，***在第 349 页***）。
* 在 BEGINFILE 与 ENDFILE 规则中也允许重定向的 getline（查看 ， ***在第 143 页***）。
* where 命令被添加到了调试器中（查看 ， ***在第 330 页***）。

## A.7 通用扩展汇总

下面的表中归纳了由 gawk，Brian Kernighan awk 与 mawk 所支持的通用扩展。这是三个用得最多，并且可以自由获取的 awk 版本（查看 ， ***在第 435 页***）。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Feature | BWK ***awk*** | ***mawk*** | gawk | 目前标准 |
| ***‘\x’*** 转义序列 | X | X | X |  |
| FS 为空串 | X | X | X |  |
| /dev/stdin 特殊文件 | X | X | X |  |
| /dev/stdout 特殊文件 | X | X | X |  |
| /dev/stderr 特殊文件 | X | X | X |  |
| 不需要下标的数组删除 | X | X | X | X |
| ***fflush()*** 函数 | X | X | X | X |
| ***length()*** 的数组版本 | X | X | X |  |
| ***nextfile 语句*** | X | X | X | X |
| \*\* and \*\*= 操作符 | X |  | X |  |
| func 关键字 | X |  | X |  |
| BINMODE 变量 |  | X | X |  |
| RS 可为正则表达式 |  | X | X |  |
| 时间相关函数 |  | X | X |  |

## A.8正则表达式范围与本地语言设置：又臭又长陈年旧事

这一节描述的是正则表达式与本地语言设置之间的混乱的历史时期，以及其如何影响不同版本的 gawk。

最初的 Unix 的正则表达式工具所定义的字符范围（如‘[a-z]’)用于匹配第一个字符到最后一个字符之间的字符（含）。其顺序基于每个字符在机器的本地字符集的数值值。所以，在基于 ASCII 的系统上，‘[a-z]’匹配所有的小字字母，因为这些从 ‘a’ 到 ‘z’ 之间数值值是连续的。（在 EBCDIC 系统上，‘[a-z]’这个范围也包含其他的非字母字符。）

几乎所有的 Unix 基础讲义都说此类型的范围表达式都是可工作的，还特别教人们说，‘[a-z]’是用来匹配小写字母的“正确”方式，而‘[A-Z]’则是匹配大写字母的“正确”方式。而实际上，这也是真的。[[113]](#footnote-114)

1992 年 POSIX 标准引入了本地语言设置这样的点子（查看 ， ***在第 136 页***）。很多的本地语言除了包含英语表中的26个字母外，还包含其他的字母，POSIX 标准则添加了字符类（查看 ， ***在第 56 页***）来匹配除了 ASCII 字符集中的传统字母之外的字符。

但是，标准对于范围的解释也发生了变化。在 “C” 以及 “POSIX”语言配置中，范围表达式如 ***‘[a-dx-z]’*** 与 ***‘[abcdxyz]’*** 是一样的，如在 ASCII 中的一样。但是在其他的语言设置中，顺序是按照协同顺序来进行定义的。

这是什么意思呢？在很多的语言设置中，‘A’与 ‘a’ 两者都小于‘B’。也就是说，这些语言设置会将用字典顺序来对字符进行排序，那么‘***[a-dx-z]***’ 不再与 ‘***[abcdxyz]***’ 等同了。相反，他有可能与‘***[ABCXYabcdxyz]***’等同。

还需要强调的是：很多的老师都教授用 ‘[a-z]’来匹配小写字母。但是在非 ASCII 语言设置的系统上，这也会匹配所有的大写字母，不包括‘A’或者‘Z’！这导致了持续的困惑，即使到了21世纪的今天。

为了说明这个问题，下面的例子使用 sub() 函数来执行文件匹配（查看 ， ***在第 184 页***）。这里，我们的目的是移除后面的大写字母：

**$ echo something1234abc | gawk-3.1.8 ’{ sub("[A-Z]\*$"， ""); print }’**

**-| something1234a**

这个输出并不是所期望的，因为 ‘bc’是 ‘***something1234abc***’的结束串，不应该匹配 ‘***[A-Z]\****’。是本地语言设置导致了这样的结果（因此你可能在你的系统上看不这样的输出）。

其他范围也有类似的问题。如 ***‘["-/]’*** 在 ASCII 中绝对有效，但是在很多的 Unicode 语言设置中，如 en\_US.UTF-8 中就不是这样。

gawk 早期的版本使用的正则表达式代码对于语言设置是不敏感的，所以对于范围的解释是之前的做法。

当 gawk 切换到使用语言设置敏感的正则表达式匹配器，问题就出现了。特别是在 GNU/Linux 与商业 Unix 的厂商开始实现非 ASCII 语言之后，并使这些语言成为默认选项。也许 FAQ（frequently asked question） 的问题已经变成这样，“为什么‘[A-Z]’会匹配小写字母？！？”

这样的问题已经存在了近10年，如果不是更多的话。gawk 已经很好地支持标准，并处理好了用户本地语言的问题，gawk 的维护者对于解释这些问题都已经非常厌倦了。在开发 4.0 版本的过程中，他们修改了 gawk，使其总是按原来的方式对待范围，即在标准前的方式。除非吏用了 --posix 选项（查看 ， ***在第 35 页***）。[[114]](#footnote-115)

幸运的是，最 gawk 4.0 发布不久，维护者了解到 2008 的标准已经更改了范围的定义，这样 “C” 与 “POSIX”之外的语言设置，范围表达式的含义是未定义的。[[115]](#footnote-116)

由于使用了这么一个可爱的技术术语，标准给了实现者相应许可来自己选择实现范围的方式。gawk 维护者选择了使用 POSIX 标准之前的含义，而不论是默认的正则表达式匹配还是使用了 --traditional 或者 --posix 选项。在所有的情况下，gawk 都与 POSIX 标准是符合的。

## A.9 gawk 的主要贡献者

*如果感谢是应该的就得感谢。*

*—Anonymous*

这一节列出了 gawk 以及/或者本书的主要贡献者，以大概年代前后顺序列出：

* Dr. Alfred V. Aho， Dr. Peter J. Weinberger 与 Dr. Brian W. Kernighan，以及贝尔实验室中所有的设计与实现了 Unix awk 的人，从这些人中形成了 awk 的主要特性集合。
* Paul Rubin 在 1986 年完成了最初的设计与实现，并写了本书的第一稿（大概 40 页）。
* Jay Fenlason 完成了最初的实现。
* Diane Close 修订了本书的第一稿，达到了 90 多页。
* Richard Stallman 帮助完成了最开始的实现，以及本书初稿。他同时也是 FSF 与 GNU 项目的创建者。
* John Woods 贡献了 gawk 初始版本的部分代码（大部份是修复）。
* 1988 年，David Trueman 成为 gawk 的主要维护者，并使其与“新”的 awk 兼容，并极大地提高了性能。
* Conrad Kwok， Scott Garfinkle 与 Kent Williams 使用不同版本的 MSC 完成了最初的向 MS-DOS 的移植。
* Pat Rankin 提供了 VMS 的移植及其文档。
* Hal Peterson 提供了移植 gawk 到 Cray Systems 的帮助。（当前已不再支持。）
* Kai Uwe Rommel 提供了 OS/2 的最初的移植版本及其文档。
* Michal Jaegermann 提供了 Atari 系统的移植及其文档。（这个移植不再支持。）他持续提供了可移植性检查，并完成了相当多的工作，以确定 gawk 可以工作在非 32 位系统上。
* Fred Fish 提供了 Amiga 系统的移植及其文档。（随着 Fred 的去世，这也不再支持了。）
* Scott Deifik 当前使用 DJGPP 维护 MS-DOS 的移植版。
* Eli Zaretskii 当前使用 MinGW 维护 MS-Windows 的移植版。
* Juan Grigera 提供了 Window32 系统的移植。（这已经不再支持。）
* 很多年以后，Dr. Darrel Hankerson 作为不同 PC 平台的移植版本的协调者，并构建了不同 PC 操作系统上的二进制发布版本。它也维护不同 PC 平台上的文档，以保持文档的及时更新。
* Christos Zoulas 提供 extension() 内置函数用于动态添加新功能（这在 gawk 4.1 版本已经取消。）
* J¨urgen Kahrs 贡献了最初的 TCP/IP 网络代码版本及其文档，并导致了 ‘|&’操作符的加入。
* Stephen Davies 提供了 Tandem 系统的最初移植版本及其文档。（但是，这也不再支持。）它也提交了最初的整合字节码实现到 gawk 代码库中的工作。
* Matthew Woehlke 提供了 Tandem POSIX 兼容系统的增强。
* Martin Brown 提供了 BeOS 的移植版本及其文档。（已不再支持。）
* Arno Peters 做了初始工作来将 gawk 切换到 GNU Automake 以及 GNU gettext。
* Alan J. Broder 提供了最初的 asort() 函数的版本，以及 match() 函数的第三个可选参数的代码。
* Andreas Buening 更新了 gawk 的 OS/2 的移植版本。
* IBM 日本的 Isamu Hasegawa （长谷川勇）贡献了对多字节字符的支持。
* Michael Benzinger 贡献了 switch 语句的最初代码。
* Patrick T.J. McPhee 贡献了在 Windows32 环境下动态装载的代码。（不再支持。）
* Anders Wallin 帮助维护了 VMS 移植版本多年。
* Assaf Gordon 贡献了实现 --sandbox 选项的实现代码。
* John Haque 作了如下贡献：
  + 修改 gawk，将其切换为一个字节码解释器，及其调试器。
  + 添加了真正的数组之数组
  + 其他修改以支持任意精度的算术运算
  + ，***在第 336 页***，的初始文本
  + 将三个版本的 gawk 合并为一的工作，以配合 4.1 的发布
  + 增强了由整数进索引的数组的内部实现。
  + 增强了由 John 与 Pat Rankin 驱动的数组排序特性
* Panos Papadopoulos 贡献了 ***，在第 46 页***，的初始文本。
* Efraim Yawitz 贡献了***，在第 320 页***，的初始文本。
* 在 gawk 4.1 版本的扩展 API 的开发主要由 Arnold Robbins 与 Andrew Schorr 驱动，并在后面的开发团队中做了巨大的贡献。
* John Malmberg 对于 OpenVMS 移植版及其文档做了巨大的贡献与增强。
* Antonio Giovanni Colombo 重写了之前章节中的很多例子，因为那些例子很多已经严重过时，我对其感谢不尽。
* Arnold Robbins 从 1988 年以来一直工作在 gawk 上，第一个协助 David Trueman，也是 1994 年以来的最主要的维护者。

## A.10总结

* awk 语言已经发展了很久。第一个发布版本大约在 1978 年的 V7 Unix 系统上。在 System V Release 3.1 的 1987 年的时候，一些主要的新增功能，包含用户自定义函数已经加入到语言中。其他的改变在 1989 年的 System V Release 4 加入。从此之后，其他的次要的变量都在 POSIX 标准的框架下进行。
* Brian Kernighan 的 awk 提供了一些扩展，并被实现为其他 awk 版本的通用扩展。
* gawk 提供了大量的 POSIX awk 所没有的扩展。它们可以使用 --traditional 或者 --posix 选项来禁用。
* gawk 中的 POSIX 本地语言设置与正则表达式匹配之间的纠葛一直让人困惑了好多年。现在，gawk 实现了 “合理范围解释”，如此， ‘[a-z]’ 这样的范围形式只匹配所在机器的本地字符集中的，数值在 ‘a’ 到 ‘z’之间的字符。字符集通常是 ASCII，但是在 IBM S/390 上可能是 EBCDIC。
* 很多人多来来对 gawk 的开发做出了贡献。 我们希望本章中所提供的列表是完整的，并在值得感谢的地方给予了相应的感谢。

# 附录 B 安装 gawk

本附录提供了用于安装 gawk 到由开发者所支持的不同平台上的指导。主要的开发者支持 GNU/Linux （以及 Unix）平台，其他则由贡献决定。查看 ，***在第 434 页***，来获取维护相应移植版本的维护者的邮件。

## B. 1gawk 发布版

这一节描述如何来获取 gawk 发布版本，如何解释，并列出不同的文件以及子目录。

### B.1.1 取得 gawk 发布版

有两个方式来获取 GNU 软件：

* 从某人那里直接复制。
* 从 Internet 主机 ftp.gnu.org 中的目录 /gnu/gawk 中来取得 gawk。匿名的 ftp 与 http 访问都是受支持的。如果你有 wget 程序，你可以如下来使用命令：

**wget http://ftp.gnu.org/gnu/gawk/gawk-4.1.2.tar.gz**

GNU 软件文档在全世界都有镜像。同步更新的镜像站点可以从主 FSF 网站上获取 。尝试其中一个镜像站点，如果不太忙，你可以使用一个离你近一点的站点。

### B.1.2 提取发布版

gawk 的发布版本有内个 tar 文件并用不同的压缩程序进行压缩：zip，bzip2 以及 xz。为简单起见，后面的说明假设你使用 GNU Gzip 程序进行压缩的（gzip）。

只要有了发布片（如 gawk-4.1.2.tar.gz），则可以使用 gzip 来解压文件然后使用 tar 来提取。你可以使用下面的管线来生成 gawk 的发布文件：

**gzip -d -c gawk-4.1.2.tar.gz | tar -xvpf -**

在使用 GNU tar 的系统上，你可以让 tar 来帮你进行解压：

**tar -xvpzf gawk-4.1.2.tar.gz**

解压后的文件会在当前目录下创建一个名字为 gawk-4.1.2 的目录。

发布文件名是 gawk-V.R.P.tar.gz 形式的名字。V 代表了 gawk 的主要版本，R 表示了当前 V 版本的发布版本，P 代表了补丁等级，表示在这个发布中修复了一些将要的 Bug。当补丁等级是 2，但是当返回发布时，你会取得最高版本号，发布版以及补丁等级。（提示，补丁等级大于等于70，表示是“beta”或者是非生产用软件，除非你不介意实验，否则你不要使用这样的版本。）如果你不在 Unix 或者 GNU/Linux 系统上，你需要对获取了解压 gawk 发布版另做打算。你可以咨询一下身边的专家。

### B.1.3 gawk 发布版的内容

gawk 发布版有很多的 C 源文件，文档文件，子目录以及与配置过程相关的文件（查看 ， ***在第 423 页***），还有几个子目录与不同的非 Unix 操作系统相关：

***各种 ‘.c’，‘.y’，与 ‘.h’ 文件***

这些文件包含实现的 gawk 源代码。

***ABOUT-NLS***

文件中包含关于 GNU gettext 与翻译的相关信息。

***AUTHORS*** 一些关于 gawk 的作者信息。它的存在只是为了符合自由软件基金会的规范。

***README***

***README\_d/README.\****

描述性文件：README 用于 Unix 下的 gawk，以及其他不同的软硬件组合。

***INSTALL*** 提供了配置与安装过程的基本过程的文件。

***ChangeLog***

bugs 修复或者增强的源代码详细列表。

***ChangeLog.0***

源代码更改的旧列表。

***NEWS*** 最后一个版本以来 gawk 的更改列表。

***NEWS.0*** gawk 的旧的更改列表。

***COPYING*** GNU 通用许可证文本。

***POSIX.STD***

awk 的 POSIX 标准中定义为未定义的行为描述，或者 gawk 中那结不完全符合，或者 POSIX 本身应该描述，但是没有描述的那些事性。

***doc/awkforai.txt***

指向初始文稿，该文稿简单地描述了为什么 gawk 是一个用于人工智能（AI）的好的编程语言的原因。

***doc/bc\_notes***

gawk 的内部“字节码”的简单描述。

***doc/README.card***

***doc/ad.block***

***doc/awkcard.in***

***doc/cardfonts***

***doc/colors***

***doc/macros***

***doc/no.colors***

***doc/setter.outline***

五色 awk 参考卡的 troff 源代码。像 GNU troff（groff）这样的现代版本的 troff 用于产生彩色的版本。查看 README 文件。如果你有旧的 troff，卡是说明文档。

***doc/gawk.1***

用于描述 gawk 手册的 troff 的源代码。这个文件是为了方便 Unix 用户。

***doc/gawktexi.in***

***doc/sidebar.awk***

本书的 Texinfo 源代码文件。它应该由 doc/sidebar.awk 文件进行处理，并在使用 texi2dvi 或者 texi2pdf 产生可打印的文档之前进行，使用 makinfo 来产生 Info 或者 HTML 文件。Makefile 会处理这些处理过程，并产生使用 viatexi2dvi 或者 texi2pdf 来产生可打印的输出。

***doc/gawk.texi***

使用 sidebar.awk 处理 gawktexi.in 文件之后产生的文件。

***doc/gawk.info***

为本书产生 Info 文件。

***doc/gawkinet.texi***

使用 gawk 进行 TCP/IP 网络处理的 Texinfo 源文件。它应该由 TEX 处理（通过 texi2dvi 或者 exi2pdf）来产生可打印的文档以及用 makeinfo 来产生 Info 或者 HTML 文件。

***doc/gawkinet.info***

gawk 中进行 TCP/IP 网络处理生成的 Info 文件。

***doc/igawk.1***

在 ，***在第 286 页***，中描述的 igawak 程序手册的 troff 源代码。

***doc/Makefile.in***

在配置过程中使用的输入文件，用于产生实际的生成文档的 Makefile。

***Makefile.am***

***\*/Makefile.am***

被 GNU Automake 软件使用的文件，以生成由 Autoconf 与 configure 使用的 Makefile.in 文件。

***Makefile.in***

***aclocal.m4***

***bisonfix.awk***

***config.guess***

***configh.in***

***configure.ac***

***configure***

***custom.h***

***depcomp***

***install-sh***

***missing\_d/\****

***mkinstalldirs***

***m4/\**** 这些文件与子目录在在不同的 Unix 系统中配置及编译 gawk 时使用。它们大多数在 ，***在第 423 页***，中进行的解释，共余的则作为主框架的支持文件。

po/\* 包含信息翻译的 po 库。

***awklib/extract.awk***

***awklib/Makefile.am***

***awklib/Makefile.in***

***awklib/eg/\****

awklib 目录包含了 extract.awk 的复本（查看 ， ***在第 281 页***），这个文件用于从本书的 Texinfo 源文件中提供例子程序。同时也包含了 Makefile.in 文件， configure 使用其来生成 Makefile 文件。Makefile.am 文件被 GNU Automake 用来创建 Makefile.in 文件。来自于 ***，在第 221 页***，的库函数，以及来自于 ，***在第 286 页***，的 igawk 程序，已经含在发布版本中，并且开箱即用。在安装过程中作为一部分进行安装。本书的其他程序在 awklib 等子目录中都可以获取 。

***extension/\****

被 gawk 包含的例子扩展的源代码，手册，以及基础文件。查看 ***，在第 349 页***，来获取更多信息。

posix/\* 用于在 POSIX 兼容的系统上构建 gawk 所需要的文件。

pc/\* 用于在 MS-Windows 与 OS/2 系统上构建 gawk 所需要的文件（查看 ，***在第 425 页***，获取相关信息）。

vms/\* 用于在 Vax/VMS 与 OpenVMS 系统上构建 gawk 所需要的文件（查看 ，***在第 430 页***，获取相关信息）。

test/\* gawk 的测试包。你可以在 gawk 的顶层目录中使用‘make check’来执行 gawk 的测试包。如果 gawk 测试成功，则你可以认为移植是成功的。

## B.2 在 Unix-like 的系统上编译与安装 gawk

一般情况下，你只需要两个命令来编译与安装 gawk。但是，如果你使用的是不常用的系统，你可能需要自己来为你的系统来配置 gawk。

### B.2.1 为 Unix-like 的系统编译 gawk

在所有的现代由 Unix 派生出来的系统中，都可以使用上面的常规步骤，如 GNU/Linux，BSD 派生的系统，以及 MS-Windows 中的 Cygwin 环境。

在解压了 gawk 发布文件后 切换到 gawk-4.1.2 目录。与大多的 GNU 软件类似，你通过执行 configure 文件来在系统上对 gawk 进行配置。这个程序是一个 Bourne Shell 脚本，由 GNU 的 Autoconf 自动产生。（Autoconf 软件在 Autoconf——配置脚本自动力生成，中有全面的描述，这个可以在自由软件基金会的网站上找到。）

简单地运行下面的命令来配置 gawk：

**sh ./configure**

这会产生 Makefile 与你的系统匹配 config.h。config.h 文件描述了你的系统的各种特性。你可以编辑 Makefile 来改变 CFLAGS 变量，用以控制传递到 C 编译的命令行选项（如优化等级或者调试编译选项）。

还有一种方法，你可以添加你自己的 make 变量到命令行中，像 CC 以及 CFLAGS 等，如下：

**CC=cc CFLAGS=-g sh ./configure**

查看文件 INSTALL 来获取详细内容。

完成了 configure，并对 Makefile 做了相应修改，输入：

**make**

不久后，你就可以有一个可以运行的 gawk 版本。所有的就是这些了。为了确认gawk 是可以工作正常的，执行 ‘make check’。所有的测试都应该是成功的。如果这些步骤没有成功，或者测试失败，检查 README\_d 目录来查看是否找到已知的问题。如果在那里并没有相关的失败的描述内容，则发送一个 Bug 报告（查看 ， ***在第 434 页***）。

当然，一旦完成了构建 gawk，你可以安装它。为了安装 gawk，你还需要执行 ‘make install’，所执行的用户需要有一定的权限。如何来完成安装，不同的系统是不同的，但是在多数的系统中，你可以通过 sudo 来完成。则命令会变成 ‘sudo make install’。有可能有密码输入请求，在这之前，你应该有配置好哪些用户可以执行 sudo 命令。

### B.2.2其他配置选项

当从头编译 gawk 时，还有几个额外的选项可以用在 configure 命令行上，它们是：

--disable-extensions

禁用配置与构建 extension 目录下的例子扩展。在交叉编译的时候，这个很有用。默认的行为是动态地检查扩展是否可以配置与编译。

--disable-lint

禁用所有的 gawk 的 lint 检查。--lint 与 --lint-old 选项（查看 ）可以被接受，但是什么也不做。类似地，设置 LINT 变量（查看 ）在运行的 awk 程序中也没有任务效果。当使用 GCC 的自动力的死代码清除功能时，这个选项可以使 gawk 的可选择体在 GNU/Linux x86-64 系统上小 23KB。在其他系统与编译器上的结果有差异。使用这个选项，可以给你带来一点性能上的提升。

注意：使用这个选项可能导致测试包中的一些测试无法通过。这个选项可能在后面会被移除。

--disable-nls

禁用所有的消息翻译功能。一般来说这并不是所期望的，但是可以给你带来一点性能提升。

--with-whiny-user-strftime

强制在低效的系统上使用 C 版本的 strftime() 函数。

使用命令 ‘***./configure --help’***来查看 configure 提供的完整的选项列表。

### B.2.3配置过程

这一节只有在你了解 C 语言以及类 Unix 操作系统时才会有趣。

gawk 的代码一般会尝试尽量符合正式的标准。这就是说，gawk 会使用由 ISO C 标准以及 POSIX 操作系统接口标准的库例程。gawk 源代码需要使用 ISO C 的编译器（1990 标准）。

很多的 Unix 系统要么不支持 ISO，要么不支持 POSIX 标准。gawk 中的 missing\_d 子目录包含了那些最可以缺失的函数。

configure 生成的 config.h 文件包含了那些你编译 gawk 所在系统的特定特性。这个文件描述的三件事情是：可以包含哪些头文件，这样它们可以正确地包含，在你的 C 库中可以使用哪些标准的函数，以及其他的系统特性。如，有可能在 stat 结构中没有 st\_blksize 元素。在这个情况下，‘HAVE\_STRUCT\_STAT\_ST\_BLKSIZE’ 就会被反定义。

也有可能你的 C 编译器会对 configure 说谎。当某个库函数不可用时，不会以错误退出。为了解决这个问题，修改 custom.h 文件。使用适用于你的系统的 ‘#ifdef’ 或者 #define 所有那些应该定义而没有定义的常量，或者 #undef 那些本不应该定义而定义了的常量。custom.h 文件会被 config.h 文件自动力包含。

有可能由 Autoconf 生成的 configure 文件在你的系统上没法工作。如果有问题，configure.ac 文件作为 Autoconf 的输入文件，你以修改这个文件来产生一个新版本的 configure 文件，使其可以运行在你的系统上（查看 ，***在第 434 页***，来获取如何来报告 gawk 的配置问题）。类似的机制可以应用于 configure.ac 与/或者 custom.h 文件。

## B.3安装到其他操作系统上

这一节描述如何将 gawk 安装到其他的非 Unix 系统上。

### B.3.1安装到 PC 操作系统

本节包含的内容为在运行于 Intel 架构的机器上的 MS-DOS，任意的 MS-Windows 或者 OS/2 系统中 gawk 的安装与使用。这一节中所使用的术语“Windows32”指的是 Microsoft 的 Windows95/98/ME/NT/2000/XP/Vista/7/8。

MS-DOS 的范围（以及在其他操作系统下的 MS-DOS外壳程序）指的是不同的“DOS 扩展器”，常与 gawk 这样的程序一起使用。Microsoft Windows 3.1 与 Windows32 不同的能力导致了困惑的产生。为了对这些考量有个全面的概况，请查看发布程序中的 README\_d/README 文件。

#### B.3.1.1为 PC 系统安装一个编译版

如果你有一个维护者为 MS-DOS提供二进制版本，gawk 以及相关的支持文件会存存在 gnu 目录下，执行文件存在 gnu/bin，库存存储在 gnu/lib/awk 目录中，手册存放在 gnu/man。这是为了能够非常容易地将 gawk 安装到你的驱动程序中的 gnu 目录——但是，只要正确设置 AWKPATH，你可以安装在任何位置。如果不考虑安装目录，igawk.cmd 与 igawk.bat（在 gnu/bin 目录中）有可能需要修改。

二进制发布版本包含一个独立的文件来描述内容。有一些特别情况下，它可能包含多于一个版本的 gawk 可执行文件。

OS/2（32 位 EMS）二进制发布版本会安装在你选择的驱动的 /usr 目录下。如果你想将 gawk 安装到一个不是硬编码的 ‘c:’分区上，你可以将 UNIXROOT 设置为你安装的驱动器上（如，‘e:’）。可执行文件会出现在 /usr/bin 目录下，库会存储在 /usr/share/awk 目录下，而手册则在 /usr/man 目录下，Texinfo 文档存在 /usr/info，NLS 文件存在 /usr/share/locale 中。注意，只要正确地设置 AWKPATH，你可以安装到任何位置。

如果你已经有一个文件 /usr/info/dir 是另一个包的，不要覆盖它！相反在你提示符下输入命令（将 ‘x:’按成你的安装驱动器）：

**install-info --info-dir=x:/usr/info x:/usr/info/gawk.info**

**install-info --info-dir=x:/usr/info x:/usr/info/gawkinet.info**

二进制发布版本中可能还包含单独的文件，其中包含了一些额外的更详细的安装指导。

#### B.3.1.2为 PC 操作系统编译 gawk

可以使用来自于DJ Delorie（ 只应用于 DJGPP: MS-DOS ），MinGW（Windows32）或者 Eberhard Mattes(EMX: MS-DOS， Windows32 与 OS/2) 的 GNU 开发工具为 MS-DOS，Windows32 与 OS/2 编译 gawk。文件 README\_d/README.pc 中包含了额外的提示，pc/Makefile 包含了编译编译选项的重要信息。

为了为 MS-DOS 与 Windows32 构建 gawk，将 pc 目录下（除了 ChangeLog）的文件与 gawk 其他的源文件复制到某个目录下，然后调用 make 来执行特定的目标以构建 gawk。从 pc 目录下复制而来的 Makefile 中包含了一个配置段，其中带有相应的注释，有可能需要修改以使其可以使用你的 make 工具用于其上。

Makefile 支持很多的构建目标，用于构建不同的 MS-DOS 与 Windows32 版本。如果 make 时没有带上目标，则可用的目标会被打印出来。如，要使用 DJGPP 工具来构建 gawk，输入‘make djgpp’。（在编译时所需要的 DJGPP 工具可以在 ftp://ftp.delorie.com/pub/djgpp/current/v2gnu/ 找到）使用 MinGW 构建本地的 MS-Windows 的二进制版本的 gawk，则输入 ‘***make mingw32***’。

gawk 的 32 位 EMX 版本在 OS/2 下是“开箱即用”的。但是，我们强烈建议你用 GCC2.95.3 来进行编译。总的来说，可以通过下面的方式来对 gawk 进行编译：

**$ ./configure**

**$ make**

但是我们不建议这么做。为了要取得一个 OMF 的可执行文件，在你的 sh 提示符下，你应该使用下面的命令行：

**$ CFLAGS="-O2 -Zomf -Zmt"**

**$ export CFLAGS**

**$ LDFLAGS="-s -Zcrtdll -Zlinker /exepack:2 -Zlinker /pm:vio -Zstack 0x6000"**

**$ export LDFLAGS**

**$ RANLIB="echo"**

**$ export RANLIB**

**$ ./configure --prefix=c:/usr**

**$ make AR=emxomfar**

这是对使用 GCC 2.x 的建议。你可以使用任何其他的环境集变量（自包含的）以及编译选项。

如果使用的是 GCC 2.95，我们还建议如下：

**$ LIBS="-lgcc"**

**$ export LIBS**

如果你想的话，你还可以得到一个 a.out 可执行文件：

**$ CFLAGS="-O2 -Zmt"**

**$ export CFLAGS**

**$ LDFLAGS="-s -Zstack 0x6000"**

**$ LIBS="-lgcc"**

**$ unset RANLIB**

**$ ./configure --prefix=c:/usr**

**$ make**

提示：a.out 可执行文件的编译在 GCC3.2 下也是可用的。GCC3.2 后的版本还没有测试成功过。

‘***make install***’在 EMX 构建时会如预期的方式执行。

提示：GNU make 的古老 OS/2 移植版本无法处理这个包中的 Makefile 文件。如果你使用 make 遇到了这样的问题，尝试一下 GNU Make 3.79 或者更新的版本。你应该在 ftp://hobbes.nmsu.edu/pub/os2/ [[116]](#footnote-117)上查找最新版本。

#### B.3.1.3在PC操作系统上测试gawk

使用 make 来执行标准的测试并安装 gawk 需要一些其他的类 Unix 的工具，这些工具包括 sh，sed 以及 cp。为了要执行测试，test/\*.ok 文件必须要进行相应的更改，这样它们就有通常的 MS-DOS 风格的换行标志。另外，可以执行 make check CMP="diff -a" 来在文本模式下使用 GNU 的 diff 工具而不是 cmp 来比较结果文件。

大多数的测试在 Stewartson 的 shell 环境中都应该是成功的，这个 Shell 中带有相应的工具以及适当的 GNU 工具。但是，需要对 test/Makefile 做一些修改。我们建议你复制 pc/Makefile.tst 文件到 test/Makefile 中进行替换。相关的细节可以在 ***README\_d/README.pc 文件以及 pc/Makefile.tst*** 文件中找到。

在 OS/2 中，由于使用了 spawnl() 而不是 fork()/execl() 来开启子进程，所以 pid 测试会失败。由于所需要的多字节函数不可用，所以 mbfw1 与 mbprintf1 测试也会失败。

#### B.3.1.4在PC 操作系统上使用 gawk

在 MS-DOS 与 MS-Windows 平台下，Cygwin 与 MinGW 环境支持 ‘|&’操作符与 TCP/IP 网络（查看 ， ***在第 306 页***）。EMX (仅 OS/2)至少支持 ‘|&’ 操作符。

MS-DOS 与 MS-Windows 版本的 gawk 会搜索在 ，***在第 43 页***，中描述的程序文件。但是，是用分号（而不是冒号）来分隔 AWKPATH 变量中的元素。如果没有设置 AWKPATH 或者为空，则默认的搜索路径是 ‘***.;c:/lib/awk;c:/gnu/lib/awk***’。

对于 OS/2(32位，EMX）的搜索路径是前缀路径来确定（多为 /usr 或者 c:/usr），在 Unix 版本的情况下，这个前缀已经在 configure 配置时的一个选项中进行了指定。如果 c:/usr 是目录缀，则默认的搜索路径包含 ‘.’ 与 ‘c:/usr/share/***awk***’。另外，为了支持在 OS/2 系统中的二进制发布，在该系统上 ‘c:’分区有可能不支持长文件或者，或者根本就不存在，因此需要一个特殊的环境变量。如果 UNIXROOT 指定了一个驱动器，这个驱动器也会进行程序文件的搜索。比如，如果 UNIXROOT 被设置为 ‘e:’，则完全的默认搜索路径为 ‘***.;c:/usr/share/awk;e:/usr/share/awk***’。

如像 sh 样的 shell（相比于 MS-DOS 中的 command.com 或者 MS Windows 或者 OS/2 中的 cmd.exe）对于 awk 编译会大有用处。DJGPP 工具组件包含了一个 MS-DOS 版本的 Bash，也包含了多个用于 OS/2 中的 Shell，如 ksh。

在 MS-Windows，OS/2 以及 MS-DOS 中，gawk（以及很多的其他的文件程序）会默默地将 ‘\r\n’这样的行尾转换成成 ‘\n’，或者将输入中的 ‘\n’转换成 ‘\r\n’输出。特殊变量 BINMODE（c.e.）可以控制这样的转换，并做如下解释：

* 如果 BINMODE 为 “r”，或者 1，则二进制模式被设置在“读”操作上（即在读时不会进行转换。
* 如果 BINMODE 为 “w”或者 2，则二进制模式设计在“写”操作上（即写时不会进行转换。）
* 如果 BINMODE 为 “rw”或者“wr”或者 3，则二进制模式会设置在读写操作上。
* BINMODE=non-null-string 与 ‘BINMODE=3’一样（即，在读写时都不进行转换）。但是 gawk 会在字串不是 “rw”或者 “wr”时，会生成一个告警信息。

标准输入或者标准输出的模式只会设置一次（在读取命令行之后，但是在处理任何 awk 程序之前）。在命令行中，使用适当的 ‘***-v BINMODE=N***’的选项来设置标准输出与标准输出的 BINMODE。BINMODE 在文件或者管道打开时，并且在流处理过程中不能够改变。

BINMODE 的名字的选择与 mawk 匹配（查看 ， ***在第 435 页***）。mawk 与 gawk 处理 BINMODE 是类似的，但是 mawk 添加了一个 ‘-W BINMODE=N’ 选项以及环境变量来设置 BINMODE，RS，以及 ORS。文件 binmode[1-3].awk（一些二进制发布版本会在目录 gnu/lib/awk 下）选择与 mawk 的 ‘-W BINMODE=N’选项匹配。这即可以改变也可以取消。特别是 对于 RS 的设置带来最小的“惊奇”还在争论。如果二进制模式在读操作时设置了，则 mawk 会使用 ‘RS="\r\n"’，这对于 MS-DOS风格的行结尾的文件是合适的。

为了说明，下面的例子在写到标准输出及其他文件时设置了二进制模式，并且将 ORS 设置成“正常”的 MS-DOS风格的行尾。

**gawk -v BINMODE=2 -v ORS="\r\n" ...**

或者：

**gawk -v BINMODE=w -f binmode2.*awk* ...**

这与 mawk 中的 ‘-W BINMODE=2’有相同的结果。下面将记录的分隔符改成 “\r\n”并设置读的二进制模式，但是不影响标准输入的模式：

**gawk -v RS="\r\n" -e "BEGIN { BINMODE = 1 }" ...**

或者：

**gawk -f binmode1.*awk* ...**

如果引号恰当，第一个例子中对 RS 的设置可以移到 BEGIN 规则中。

#### B.3.1.5在Cygwin 中使用 gawk

如果你在 MS-Windows 系统中使用 Cygwin环境，则gawk可以构建完成后即可使用。这个提供了对使用 Bash，GCC，GNU Make 及其他 GNU 程序的 GNU/Linux 系统很好的模拟环境。在 Cygwin 上编译与安装与在 Unix 系统上是一样的：

**tar -xvpzf gawk-4.1.2.tar.gz**

**cd gawk-4.1.2**

**./configure**

**make && make check**

与同一个系统上的 GNU/Linux 相比，‘configure’在 Cygwin 中要花长得多的时间。但是，它一旦完成，就可以像通常那样使用 ‘make’。

#### B.3.1.6在MSYS 环境中使用 gawk

在 MS-Windows 系统中的 MSYS 环境，gawk 会自动地使用读写二进制模式。所以，没有必要设置 BINMODE 变量。

对于其他已移植到 MS-Windows 的类 Unix 组件，这可能会导致问题。因为它们可能认为 gawk 会自动转换 “\r\n”，但是实际上却没有。

### B.3.2在 Vax/VMS 以及 OpenVMS 系统上编译安装 gawk

这个子章节描述如何来编译并安装 gawk 到 VMS 中。下面使用 “VMS”这个老的术语来表示 OpenVMS。

#### B.3.2.1在VMS 上编译 gawk

为了在 VMS 下编译 gawk，有一个 DCL 命令过程会发起所有的 CC 与 LINK 命令。也有一个Makefile 与 MMS 以及 MMK 一起使用。在源代码目录下，使用：

**$ @[.vms]vmsbuild.com**

或者：

**$ MMS/DESCRIPTION=[.vms]descrip.mms gawk**

或者：

**$ MMK/DESCRIPTION=[.vms]descrip.mms gawk**

MMK 是开源免费的，几乎 是 MMS 的克隆，并能够很好地处理 ODS-5 中的大小写文件名。MMK 可以从 https: / / github . com /endlesssoftware/mmk 取得。

gawk 在使用 Compaq C V6.4 的 VAX/VMS 7.3 以及 Alpha/VMS 7.3-1 ，Alpha/VMS 7.3， Alpha/VMS 7.3-2， 与 IA64/VMS 8.3 上都已经做了相应的测试。最近的构建使用 Alpha VMS 8.3 上的 HP C V7.3，以及使用 HP C 7.3 的 Alpha 与 IA64。[[117]](#footnote-118)

查看 ，***在第 434 页***，来获取将 gawk 构建成与 GNV 产品兼容的 PCSI 包相关的信息。

#### B.3.2.2在 VMS 编译 gawk 动态扩展

被移植到 VMS 的扩展可以使用下面的命令之一来进行构建 ：

**$ MMS/DESCRIPTION=[.vms]descrip.mms extensions**

或者：

**$ MMK/DESCRIPTION=[.vms]descrip.mms extensions**

gawk 使用 AWKLIBPATH 来作为环境变量，或者是查找动态扩展的逻辑名。

动态扩展需要使用与编译 gawk 一样的编译器来编译，这样使用与 gawk 一样的方式来处理浮点数，指针大小 以及符号名。Alpha 与 Itanium 平台应该使用 IEEE 清点数。指针大小是 32 位，而符号名处理会使用 CRC 来缩短大于 32位的符号。

对于 Alpha 与 Itanium 平台：

**/name=(as\_is，short)**

**/float=ieee/ieee\_mode=denorm\_results**

对于 VAX 平台：

**/name=(as\_is，short)**

编译时间宏需要在 VMS 提供的头文件被包含前进行定义，如：

**#if (\_\_CRTL\_VER >= 70200000) && !defined (\_\_VAX)**

**#define \_LARGEFILE 1**

**#endif**

**#ifndef \_\_VAX**

**#ifdef \_\_CRTL\_VER**

**#if \_\_CRTL\_VER >= 80200000**

**#define \_USE\_STD\_STAT 1**

**#endif**

**#endif**

**#endif**

如果你写了自己的扩展运行在 VMS 上，你必须提供这些定义。config.h 文件会在 VMS 平台上构建 gawk 时为你完成这个工作。如果你使用这个文件，或者类似的其他文件，则你必须要记住在要包含 VMS 所提供的头文件之前包含它。

#### B.3.2.3在VMS 上安装gawk

要使用 gawk ，你所需要是一个“外部”命令，这是一个 DCL 符号，它的值以美元符号开始。如：

**$ GAWK :== $disk1:[gnubin]gawk**

将 gawk.exe 的实际路径替换 ***‘$disk1:[gnubin]’***。符号需要在任何的想使用 gawk 的用户的 login.com 中替换，这样每次用户登录时都会进行定义。另外 ，符号也可以用全系统的 sylogin.com 过程来替换，这样所有的用户都可以使用 gawk。

如果你的 gawk 是通过 PCSI 包来安装到 GNU$GNU: 目录树的，程序的名字则为 GNV$GNU:[bin]gnv$gawk.exe，帮助文件则为 GNV$GNU:[vms\_help]gawk.hlp。

PCSI 包也会安装 GNV$GNU:[vms\_bin]gawk\_verb.cld 文件，这个文件可以用来将将 gawk 与 awk 为 DCL 命令。

对于当前的过程，你可以使用：

**$ set command gnv$gnu:[vms\_bin]gawk\_verb.cld**

或者系统管理员可以使用 GNV$GNU:[vms\_bin]gawk\_verb.cld 来将 gawk 与 awk 加到系统级的 ***DCLTABLES’ 中。***

DCL 语法在 gawk.hlp 文件中有记录。

另外，gawk.hlp 项可以被加入到 VMS 帮助库中：

**$ LIBRARY/HELP sys$help:helplib [.vms]gawk.hlp**

（你可能想将特定于站点的帮助库进行替换，而不是标准的 VMS 库 ‘***HELPLIB***’）在载入了帮助文件后，命令：

**$ HELP GAWK**

提供了关于 gawk 实现 与 awk 程序语言的信息。

逻辑名 ‘***AWK\_LIBRARY***’可以设设成 awk 程序文件的默认位置。对于 -f 选项，如果所指定的文件吃饭去没有设备或者路径信息，gawk 会在首先在当前的目录下搜索，如果没有找到，则会在由 ‘***AWK\_LIBRARY***’ 所指定路径下搜索。如果在搜索了两个目录下，文件还没有找到，gawk 会加上‘.awk’后缀到文件名中，将再次尝试搜索。如果 ‘***AWK\_LIBRARY***’没有定义，则会使用 ‘***SYS$LIBRARY:***’的值。

#### B.3.2.4在 VMS 上运行 gawk

在 VMS 上，命令行的分析与引号规范完全不同，因此这本书中的例子，或者其他的源代码要做一些小变化。虽然小，所有的 awk 程序都应该是可以正常执行的。

这是一些简单的测试：

**$ gawk -- "BEGIN {print ""Hello， World!""}"**

**$ gawk -"W" version**

**! could also be -"W version" or "-W version"**

注意大小混用时，文本必须引起来。

gawk 的 VMS 移植版本包含了一个 DCL 风格的接口，以及原始 Shell 风格的接口（查看 help 项来获取详细信息）。双路径行分析的副作用是，如果只有一个参数（如程序中引起来的程序），则命令则会有一点模棱两可。为了解决这个问题，常规的可选的 -- 标志则用来强制使用 Unix 风格的分析方法，而不是 DCL 的分析方法。如果还有其他任何的 “横杠类型”的选项（或者多个参数，如要处理的数据文件），则就不会有二义性，-- 则可以省略掉。

退出值是一个 Unix 风格的值，当程序退出时，会被编码为 VMS 的退出值。

VMS 的严重程序位会根据退出值来进行设置。1 表示失败，VMS 会设置 ERROR 状态。2 表示致命错误，VMS 会设置 FATAL 状态。所有其他的值都表示 SUCCESS 状态。退出值的编码符合 VMS 的编码标准并使用C\_FACILITY\_NO 值 0x350000 与常量 0xA0000 加到为腾出严重程度编码位而左移了3位的数值上。

从 VMS 状态码中提取实际的 gawk 退出码，使用下面的代码：

**unix\_status = (vms\_status .and. &x7f8) / 8**

使用 exec() 的 C 程序调用 gawk 会得到最初的 Unix 风格的退出值。

用于 VMS 的旧版 gawk 会将 Unix 的退出码0当成1，失败则为 2，致命错误则为 4，其他的值则直接传递。这违反了 VMS 的退出状态编码要求。

VAX/VMS 浮点数使用无倾向的舍入舍出方案。查看 ， ***在第 226 页***。

VMS 会用 GMT 的格式报告时间，除非设置了 SYS$TIMEZONE\_RULE 或者 TZ 逻辑名字。

当搜索由 -f 指定的 awk 程序文件时，默认的搜索路径是 "SYS$DISK:[]，AWK\_LIBRARY:"。逻辑名字 AWKPATH 可以用来覆盖这个默认设置。AWKPATH 的格式是一个逗号分隔的目录列表。如果定义了这个变量，值应该被引起来，这样它会保留为独立转换，而不是 RMS 搜索列表的多转换。

#### B.3.2.5 VMS 的 GNV 项目

VMS GNV 包提供了与 POSIX 类似的构建环境，并带了很多开源工作的移植版本。gawk 可以在 GNV 的库中找到，但是是一个旧移植版本。当前，GNU 项目被重新组织了，以为每个组件提供单独的 PCSI 包。查看 https://sourceforge.net/p/gnv/wiki/InstallingGNVPackages/。gawk 的常规的构建过程会生成一个适用于 GNV 的程序。

在发行版中的 vms/gawk\_build\_steps.txt 文件记录了构建与 GNV 兼容的 VMS PCSI 包的过程。

#### B.3.2.6有一些 VMS 系统有旧版本的 gawk

有一些 VMS 的版本带有旧的 gawk 版本。为了访问它，定义如下的符号：

**$ gawk :== $sys$common:[syshlp.examples.tcpip.snmp]gawk.exe**

这个版本是 2.15.6，相当的老。我们建议你编译与使用当前的版本。

## B.4报告问题与BUG

*没有什么比一个无聊的考古学家更危险。*

*—Douglas Adams， The Hitchhiker’s Guide to the Galaxy*

如果你的 gawk 有问题，或者你认为你找到了一个 Bug，请将它报告给开发者，我们不承诺做任何事情，但是我们也是想要修复这它的。在报告 Bug 之前，确认你确实是找到了一个真的Bug。仔细重读文档，看是否你尝试要做的事。如果不是很清楚你应该做些什么，也请报告出来。这是文档中的一个 Bug。

在报告 Bug 之前，请先自己尝试修复它，将其独立成最小的 awk 程序然后输入产生问题的数据。然后将程序与数据都发给我们。对于 Unix 系统的类型，编译 gawk 的编译器版本也是很有用的，还有 gawk 所返回来的确切的结果。

确保包含你所使用的 gawk 的版本号。你可以通过下面的命令来获取 ‘***gawk --version***’。

一旦你的问题有了全面精确的描述，你就可以给 bug-gawk@gnu.org 发送邮件。

gawk 的维护者会订阅这个邮件地址，因此他们会收到你的 Bug 报告。尽管你可以直接给维护者发送邮件，但是最好还是使用专用的报告地址，因此邮件地址在 GNU 项目中是有的。所有的邮件都用英语。这是唯一被所有维护者都理解的语言。

注意：不要将 gawk 的 Bug 报告到由 comp.lang.awk 所指向的 Usenet/Internet 新闻组。gawk 的开发者偶尔才会去看这个新闻组，但是不保证我们可以看得到你提交的内容。这里所描述的步骤是官方认可的组织方式。真的。

提示：很多的 GNU/Linux 以及不同的基于 BSD 的操作系统发布版有他们自己的 bug 报告系统。如果你使用这些发布版的 Bug 报告系统，你也要发送一个复本到 bug-gawk@gnu.org。

这有两个原因。首先，尽管一些发布版本将 Bug 报告推送到 GNU 邮件列表，但是很多的都没有，所以很有可能 gawk 的维护者很有可能看不到你的报告。其次，发送到 GNU 邮件列表的邮件会被归档，将所有的关东西都放在 GNU 项目中，使得其是自说明的，而不依赖于其他的组织。

非 Bug 建议也是受欢迎的。如果你看到文档中的东西说明不清，或者不理解其中的特性，你也可以在 Bug 列表中进行提问，我们会尽可能地帮助你。

如果你在一个非 Unix 移植版本中发现 bug ，也将邮件发送到 Bug 列表 ，并抄送给维护这个移植版本的人。维护者在下面在的列表中，也存 gawk 发布版中的 README 文件里。README 文件中的信息如果与本书中的有冲突，那里的信息会更加权威。

维护不同的移植版本的人如下：

Unix and POSIX systems Arnold Robbins， arnold@skeeve.com

MS-DOS with DJGPP Scott Deifik， scottd.mail@sbcglobal.net

MS-Windows with MinGW Eli Zaretskii， eliz@gnu.org

OS/2 Andreas Buening， andreas.buening@nexgo.de

VMS John Malmberg， wb8tyw@qsl.net

z/OS (OS/390) Dave Pitts， dpitts@cozx.com

如果 Bug 在 Unix 下也是可重现的，也抄送一份报告给 bug-gawk@gnu.org 邮件列表。

## B.5其他可自由获取的 awk 实现

*如果在 awk 代码中这样来输入注释就有味了：*

*//C++ 类型的注释可用么？答案是：当然可了啦。*

*—Michael Brennan*

还有很多的可以免费获取的 awk 实现版本，这一章简单地进行说明如何来获取它们：

Unix ***awk*** Brian Kernighan 是 Unix awk 的最初设计者，它将他自己的 awk 开发出来了。你可以通过他的主页来获取，可以有几种文档格式：

Shell archive

http://www.cs.princeton.edu/~bwk/btl.mirror/***awk***.shar

Compressed tar file

http://www.cs.princeton.edu/~bwk/btl.mirror/***awk***.tar.gz

Zip file http://www.cs.princeton.edu/~bwk/btl.mirror/***awk***.zip

你也可以从 GitHub 上取得：

git clone git://github.com/onetrueawk/***awk*** bwkawk

这个命令创建了一个名为 bwkawk 的 Git 库的复本。如果你将这个参数在 git 命令里省略，则 git 库的复本会创建在名字为 awk 的目录中。

这个版本需要 ISO C（1990 标准）的编译器，来自于 GCC（GNU Compiler Collection） 的 C 编译器可以工作得很好。查看 ，***在第 414 页***，来获取这个 awk 扩展列表，这些扩展不在 POSIX awk 中。

***mawk*** Michael Brennan 单独实现了 awk，称为 mawk。这可以以 GPL （***查看 ，在第 465 页***）的方式获取，如 gawk 一样。

mawk 源代码的最初的发布地址中已经不再有 mawk，可以在这里 http://www.skeeve.com/gawk/mawk1.3.3.tar.gz 取得。

2009 年，Thomas Dickey 负责起 mawk 的维护工作。基本的信息可以在它的项目网站中可以得到。下载地址为 http://invisible-island.net/datafiles/release/***mawk***.tar.gz。

一旦取得后，用 gunzip 来解压这个文件。安装与 gawk 的相同（查看 ， ***在第 423 页***）。

查看 ，***在第 414 页***，获取不在 POSIX awk 中的 mawk 扩展列表。

awka 由于 Andrew Sumner 编写，awka 将 awk 程序转换为 C 程序，然后编译，将他们与提供了 awk 核心功能的库进行连接。它也有很多的扩展。

awk 翻译器也在 GPL 许可下发布，而库则在 LGPL 下发布。

要获取 awka，请进入 http://sourceforge.net/projects/awka。

项目看起来已经冻结起来了，从 2001 年起好像就没有新代码的变动了。

pawk 尤它大学的 Nelson H.F. Beebe 修改了 BWK awk 来提供计时与采样信息。它与 gawk 的 --profile （查看 ）是不同的，在 pawk 中使用的是基于 CPU 的采样，而不是基于行的采样。你可以在 ftp://ftp.math.utah.edu/pub/pawk/pawk-20030606.tar.gz 或者 http://www.math.utah.edu/pub/pawk/pawk-20030606.tar.gz 来获取。

BusyBox ***awk***

BusyBox 是一个 GPL 授权的程序，在单独的执行体中提供了很多应用的小版本。其目标是嵌入式系统。它包含了 POSIX awk 的完整实现。当构建时，不要使用 ‘***make install***’，因为它会覆盖 /usr/local/bin 目录下的其他复本程序。可以浏览它的主页来获取更多信息。

The OpenSolaris POSIX ***awk***

在 Solaris 系统中 /usr/xpg4/bin 与 /usr/xpg6/bin 下的 awk 多多少少与 POSIX 是兼容的。它们是基于 PC 版的 Mortice KernSystems 中的 awk 版本。我们可以在 GNU/Linux 下用一两个小时来编译它的源代码。要将它弄得更可移植（使用 GNU Autoconf 与/或 Automake）需要很多的工作，而且还没有完成，至少以我们现有的了解看来是这样的。

它的源代码过去在 OpenSolaris 站点里。但是，那个项目已经结束，网站已经关闭。幸运的是 Illumos 项目让这个实现再次可用。你可以访问 https://github.com/joyent/illumos-joyent/blob/master/usr/src/cmd/***awk***\_xpg4 来查看其中的文件。

jawk 这是用 Java 写的 awk 解释器。它声称完全是解释器，尽管因为它使用 Java 的工具来进行 I/O 以及正则表达式匹配，它所支持的语言与 POSIX awk 是不同的。可以在网页上获取更多的信息。

Libmawk 这是可嵌入的 awk 解释器，来自于mawk。要了解信息，请浏览 http://repo.hu/projects/libmawk/。

pawk 这是一个 Python 模式，它声称将 awk 类似的特性加入到 Python 中。查看 https://github.com/alecthomas/pawk 来获取更多信息（这个与早前所说的 Nelson Beebe 的 BWK awk 版本没有关系。）

QSE ***awk*** 这是一个可嵌入的 awk 解释器。更多我信息请看 http://code.google.com/p/qse/ 以及 http://***awk***.info/?tools/qse。

QTawk 这个是独立实现的，在 GPL 许可下发布的 awk。它有比标准 awk 多得多的扩展，有可能并不是 100% 与 awk 语法兼容。查看 http://www.quiktrim.org/QTawk.html 获取更多信息，其中包括手册与下载链接。

这个项目可能也已经冻结了，从大约 2008 年起，就没有新代码变更了。

Other versions

查看 Wikipedia 文档中的 “Versions and implementations”节来获取其他版本的信息。

## B.6总结

* gawk 发布版本可以从 GNU 项目的主发布站点 ftp.gnu.org 获取到。正规的构建过程是：

wget http://ftp.gnu.org/gnu/gawk/gawk-4.1.2.tar.gz

tar -xvpzf gawk-4.1.2.tar.gz

cd gawk-4.1.2

./configure && make && make check

* gawk 可以构建在非 POSIX 系统上。当前支持的系统包含使用 DJGPP，MSYS，MinGW 以及 Cygwin 的 MS-Windows，使用 EMX 的 OS/2，以及 Vax/VMS 与 OpenVMS。在相应系统上的指导也在这个附录中了。
* Bug 报告应该通过邮件发送给 bug-gawk@gnu.org。Bug 报告应该使用英语，并包含 gawk 的版本，如何进行编译的，以及一个短程序与数据文件来说明其中的问题。
* 很多的的自由获取 awk 实现，很多与 POSIX 兼容，有的则兼容得较少。

# 附录C 实现提示

这个附录中包含了主对是 gawk 的实现与维护者感兴趣的内容。里面的东西都只针对 gawk 而不是其他的实现。

## C.1后向兼容与调试

查看 ，***在第 406 页***，来获取 GNU 对于 awk 语言与程序的扩展的概况。所有的这些特性都可以通过以 --traditional 或者 --posix 选项来调用 gawk 而进行关闭。

如果 gawk 是使用 ‘***-DDEBUG***’进行编译的，则会在命令行上有额外的一个可用的命令行选项：

-Y

--parsedebug

如果程序正在被分析时，则打印出分析栈信息。

这个选项的目的仅用于很严肃的 gawk 开发者，而不用于普通用户。有可能都没有编译进你的 gawk 版本，因此它会降低执行的速度。

## C.2为 gawk 添加功能

如果你发现你想以一种特别的方式增强 gawk，那随意好了。那正是自由软件的目的，源代码是可以获取的，你也可以按你的方式来改变（查看 ***，在第 465 页***）。

这一节讨论你更改 gawk 的方式以及需要你同时记住的一些点。

### C.2.1访问 gawk 的 Git 仓库

因为 gawk 是一个免费软件，总是可以获取到它的源代码。***，在第 419 页***，中描述了如何来获取及构建 gawk 发布版本。

但是，如果你想修改 gawk 并且将这些修改发布，你可能需要使用开发版本。因此，你需要访问 gawk 的源代码库。代码是使用 Git 分布式的版本控制系统来进行维护的。如果你的系统上没有，则先需要安装它。完成后，你就可以使用下面的命令：

**git clone git://git.savannah.gnu.org/gawk.git**

这个复制 gawk 的库。如果你在防火墙后面，而不许使用 Git 的本地协议，你可以使用下面的方式来访问库：

**git clone http://git.savannah.gnu.org/r/gawk.git**

一旦你做了修改，你可以使用 ‘***git diff***’来产生一个 patch，然后发送绘画 gawk 的维护者，查看 ，***在第 434 页***，来获取如何做。

之前有一段时间有 Git-CVS 网关可以被那些没有安装的 Git 的人使用。但是，这个网关已经不再工作了，因此你最好一些更加现代的版本控制系统，如 Bazaar，它有一个 Git 插件可以用于 Git 库。

### C.2.2添加新特性

你可以自由地将新特性加入到 gawk 中。但是，如果你想将你的改变加入到 gawk 发布版中，有几个步骤可以让你有可能将它们加入到 gawk 中：

1. 在构建新的特性到 gawk 前，先考虑将其写成一个扩展（查看 **，*在第 349 页***）。如果不可能，则往下走。
2. 准备签署合适的协议。为了让 FSF 能够发布你的修改，你必须要么将这些修改发布到公共域里，并提交一个签署有相同效果的协议，或者将版本赋给 FSF。这两个做法都是很容易完成的，很多的人都已经这么做了。如果有问题，请联系我（查看 ， ***在第 434 页***），或者是 assign@gnu.org。
3. 获取最新版本，对我而言，如果这些更改相关于最新的 gawk 版本而变，我就很容易进行合并。最好的是，你的更改是相对于 Git 库的最新代码。如果你的 gawk 版本非常老，我就根本没法将它们合并了。（查看 ，***在第 419 页***，来了解如何获取最新的 gawk 版本。）
4. 遵守 GNU C 的代码标准。这个文档描述如何来写 GNU 软件。如果你还没有读过，请先读，特别是在修改 gawk 之前（GNU 代码标准可以从 GNU 项目的站点上获取。也可以获取到 Texinfo，Info，DVI 的版本。）
5. 使用 gawk 编码风格。 gawk 的 C 代码遵守 GNU 编译标准中的指导，只有一些小例外。代码使用传统的 “K&R”风格，特别在替换花括号以及 TABs 的使用上要注意。简单地说，代码规则如下：

* 使用 ANSI/ISO 风格（原型）函数来定义函数。
* 将函数放在一行的前面。
* 将函数的返回值，即使是 int 放在有参数的函数名字的上一行。
* 在控制结构与括号之间放一个空格（if，while，for，do，switch， 以及 return）。
* 在函数调用时，不要在括号之前放括号。
* 在 C 操作符的两边，以及函数调用的括号后加入空格。
* 不要使用逗号操作符来产生多种副作用，除非在 for 循环中初始化与递增的部分，以及在宏定义里。
* 使用真正的 TAB 来进行缩进，而不是空格。
* Use the “K&R” brace layout style.
* 使用 “K&R”风格的花括号布局风格。
* 在 if，while，以及 for 语句的条件中，使用与 NULL 与 ‘\0’的比较，在 switch 语句中也是一样，而不要使用光指针与字符值。
* 使用 true 与 false 来用于布尔值，NULL 符号常量是用于指针值的，而字符常量则用 ‘\0’，而不是 1 与 0。
* 对每个函数都提供一行描述性的注释。
* 不要使用 alloca() 函数用进行栈内的内存分配。它的使用会导致移植性问题，而带来的一点好处就是不用进行内存释放。要使用 malloc() 与 free()。
* 不要使用 ***‘! strcmp(a， b)’*** 类似的比较。Henry Spencer 说过，“***strcmp()*** 不是布尔值”，相反，要使用 ***‘strcmp(a， b) == 0’***。
* 如果添加了新的位标志值，要使用显式的十六进制常量（0x001， 0x002，0x004 等等），而不要使用对 1 进行连续的左移（***‘(1<<0)’***，***‘(1<<1)’***等等）。

提示：如果我要修改了你的代码来遵守 gawk 中的代码风格，是我根本就不会将你的更改整合进去的。

1. 更新文档。跟你的代码一起，请提供新的章节到本书中。如果有可能，请使用真正的 Texinfo，而不是提供一些未格式化的 ASCII 文本（尽管有文档要比没文档要好）。请遵守 *GAWK: Effective AWK Programming* 中的规则，由在 Texinfo 源文件尾部的 @bye 所提供。如果可能，请也请更新手册页。你对你的文档也需要签署相应的协议。
2. 使用统一的 diffs 来提供更改。使用 ‘***diff -u -r -N***’来比较最初的 gawk 源代码与你的代码变化。我建议使用 GNU 版本的 diff 或者最好，直接使用 ***‘git diff’*** 或者 ***‘git format-patch’。***将 diff 产生的输出发送绘画我。（查看，***在第 434 页***，来获得电子邮件信息。）

使用这样的格式，我很容易将你的更改合并到 gawk 的主版本的源代码中（使用 patch）。如果我必须要使用文本编辑器手动更改，我可能不这么做，特别是有很多的变化时。

1. 在 ChangeLog 文件中包含一项，与你的更改一起提交。这可以帮助进一步最小化我的工作，这样我就比较容易接受补丁。如果你只使用你的 diff 的这部分，那就是最简单的了。

这听起来要做很多的事情，但是请记住当你写了新代码，而我就得维护并且支持它。如果对我而言不能最小化这些额外工作，我可能就不会做。

### C.2.3将 gawk 移植到新系统中

如果你想将 gawk 移植到新的操作系统，下面是相应的步骤：

1. 遵守上一节所说的代码规范，提供 diff 等等。
2. 准备签署相应的文件。这样 FSF 可以发布你的代码，你要么将你的代码放在公共域里，要么签署一个等同于相应效果的文件，或者将版本赋给 FSF。这两个工作都是容易做的，很多人也这么做了。如果你有问题，请联系我，或者 gnu@gnu.org。
3. 当完成了一个移植，要记住你的代码要与 gawk 其他的部分一起共存。避免对系统无关的代码部分做无谓的修改。如果没有可能，请不要在你的部分里到处加 ‘***#ifdef***’。

如果改变需要一种特定系统特性，会影响太多的代码，我可能不会接受它们。在这样的情况下，你当然可以自己来发布你的更改，只要你符合 GPL 即可（查看 *，****在第 465 页***）。

1. 很多 gawk带有的文件是被其他的用人维护的。所以，你不可以更改它们，除非有很好的理由，如没有可能的情况下，但是对这些文件 的修改也要额外的仔细。这些文件是 ***dfa.c***， ***dfa.h***， ***getopt.c***， ***getopt.h***， ***getopt1.c， getopt\_int.h，gettext.h， regcomp.c， regex.c， regex.h， regex\_internal.c， regex\_internal.h，and regexec.c。***
2. 有很多文件是由 GNU Autotools 提供的（Autoconf，Automake 及 GNU gettext）。你也不可以更改它们，除非有好的理由。这些文件是 ABOUT-NLS， ***config.guess***， ***config.rpath***， ***config.sub， depcomp， INSTALL， install-sh， missing， mkinstalldirs， xalloc.h，*** and ***ylwrap***。
3. 要愿意持续维护这个移植版本。非 Unix 的操作系统是由志愿者来维护的，他们维护需要在他们的系统中的代码的维护及运行。如果没有人愿意维护这个移植版本，会成为不受支持的版本，有可能有必要从发行版本中移除掉。
4. 提供合适的 gawkmisc.??? 文件。每个移植版本都有自己的 gawkmisc.??? 文件，这些文件实现了特定于操作系统的函数。这个比在代码中用大量的 ‘#ifdef’要更加清晰。在源代码主目录中的 gawkmisc.c 包含了相应的 gawkmisc.??? 文件，这个文件来自于其子目录。记得也要更新。

每个移植版本的 gawkmisc.??? 文件有一个后缀来表示机器以及操作系统，如 ***pc/gawkmisc.pc*** 和 ***vms/gawkmisc.vms***。使用其他的后缀，而不是普通的 gawkmisc.c，使其有可能将文件从移植版本的子目录行色移到主线子目录，而不会导致偶然的对 gawkmisc.c 文件的破坏。（当前，这个只对 PC 操作系统移植才会是候总量。）

1. 提供必要的 Makefile 用于你的操作系统。你的所有代码都应该在一个单独的子目录下，名字可以与操作系统名字相同，或者是操作系统名字一部分。如果可能，将事情结构化，这样没有必须将文件从子目录中移动到主子目录下。如果不可能，则要确保使用你的文件的名字不会与主源代码目录下的文件相同。
2. 更新文档。写为本书写一个小节（或者多个小节）来描述其安装与编译步骤，这样可以在你的系统上编译与安装 gawk。

按这样的步骤，可以使其容易来将你的变化整合到 gawk 中，并使其与其他的已经存在操作系统代码和平共处。

在你提供与维护的代码中，可以自由使用你的风格以及括号布局。

### C.2.4为什么自动生成的文件要放在 Git 中

如果你查看 Git 库中的 gawk 源代码，你会注意到里面包含了由 GNU 基础工作自动力生成的文件，如 Makefile.in 由 Automake 生成，甚至是由 Autoconf 生成的 configure。

这个不同于不存储衍生文件其他自由软件项目，因为这使得库不会这么混乱，而且更容易在比较在不同提交之间的真正改变。但是，有几个原因，使得 gawk 要把所有的东西放在库中。

首先，这更容易生成完整的版本，而不依赖于其他工作的能力（有一些老的，被丢弃，甚至没办法找得到。）

一个极端例子是，如果你想编译，比方说 V7 的 awk，你会发现你不仅要用 V7 的 yacc 来处理，还需要 V7 的 lex。对于后者，在现代的 GNU/Linux 系统根本没有。[[118]](#footnote-119)

（或者说，gawk 1.2 所需要 1989 年中的 bison 什么什么的，在库中没有 awkgram.c 这个文件。有保证说我们可以找到 bison 版本？或者可以构建么？）

如果库有都有生成的文件，则很容易检查并构建。（或者，依赖于我们所回到的位置可能更容易。）

这使得我们有第二个理由（更强的）为什么把所有的文件都放在 Git 中。简单地说就是你让谁开心——是 gawk 的开发者，还是仅想取某个版本然后试试它的用户？

gawk 的维护维护者想全世界所有的对 awk 有兴趣的用户，只要克隆库，然后检出所感兴趣的分支构建即可。而不需要他们必须有正确的 autotools [[119]](#footnote-120)版本。这也是 bootstrap.sh 文件的目的。它以正确的顺序来处理不同的文件，这样：

**# The canonical incantation for building GNU software:**

**./bootstrap.sh && ./configure && make**

也可以工作。

这对于主线以及 gawk-X.Y-stable 分支来说尤其重要。

再者，gawk 的维护者也觉得这对于 gawk 的开发者也非常重要。当某人尝试检出 xgawk 的分支[[120]](#footnote-121)来构建时，却发现不能。（没有 ltmain.sh 文件，也不知道如何来生成，并且这也不是唯一的问题。）

此人会感觉精疲力尽。对于相应的分支，维护者与 Jane 这样的想构建从库中取得的 gawk-4.1-stable 用户没有什么不同。

因此，维护者觉得对于任何分支而言，上面的代码可能工作，这样不仅是重要，而且非常关键。

第三个原因是，如果没有这些文件，使用 ‘git bisect’来查找引入 bug 的提交会变成相当困难。维护者会在其他的项目中使用这个方法，这个项目需要运行自举脚本，以创建 configure 等文件，这相当痛苦。当库自己是自包含的，使用 git bisect 就相应容易了。

有什么结果与/或者采取什么行动呢？

1. 我们不介意在不同的分支下有不同的文件，这是不同的 autotools 版本的结果。
   1. 这是维护者的事来合并与处理。
   2. ***维护者们使用 ‘git diff x y > /tmp/diff1 ; gvim /tmp/diff1’来移除不感兴趣的差别以检查代码确实非常拿手。***
2. 如果所有的人都使用与维护者一样的 GNU tools 的话，肯定会有帮助，一般是最新版本的 Automake，Autoconf，以及 GNU gettext。

从源代码安装非常容易。几年来维护者都是这么做的（而且还在这么做）。只需要将 /usr/local/bin 放在 PATH 的前面，然后：

**wget http://ftp.gnu.org/gnu/package/package-x.y.z.tar.gz**

**tar -xpzvf package-x.y.z.tar.gz**

**cd package-x.y.z**

**./configure && make && make check**

**make install # as root**

上面大多数一开始是由其他的 gawk 维护者写的。有一个开发者提出的反对意见“... 从 Git 库中提取源代码的人并不都是终端用户。”

但是，事情不是这么回事。有“大牛级的 awk 用户”，他们可以构建 gawk（使用之前所列的代码），但是他们没办法用 C 写程序。所以，主要的分支 应该一真保持是可构建的。

然后建建议，应该有一个 cron 任务来每晚来创建所有代码的 tar ball 。这里的问题是，有不同分支的代码线，因此，每晚的 tar ball 不是最终解决方法，特别是库有可能几周都不会有大的变化。

幸运的是，Git 服务器可以来满足这样的需要。对于每一个名字为 branchname 的分支，使用：

**wget http://git.savannah.gnu.org/cgit/gawk.git/snapshot/gawk-branchname.tar.gz**

来返回所指定分支的快照。

## C.3后续可能的扩展

*AWK 是像 PERL 语言一样，只是要比它优美得多了。*

*—Arnold Robbins*

*嘿，怎么说的！*

*—Larry Wall*

在 Git 库中的 gawk 主线中的 TODO 文件列出来未来可能的增加。有一些与源代码相关，其他的是一些新特性。请查看这个文件来获取相关列表。如果你有兴趣跟踪列在其中的项目，请查看 ，***在第 439 页***。

## C.4实现的一些限制

下表描述了 gawk 在类 Unix 系统中的一些限制（尽管还会有变量）。其他的系统可能还有不同的限制。

|  |  |
| --- | --- |
| 项 | 限制 |
| 字符类内的字符数 | 2^(每字节的位数) |
| 输入记录的长度 | MAX\_INT |
| 输出记录的长度 | Unlimited |
| 源代码长度 | Unlimited |
| 记录中的域数 | MAX\_LONG |
| 文件重定向数 | Unlimited |
| 一个文件中的记录数 | MAX\_LONG |
| 总是输入记录数 | MAX\_LONG |
| 重定向管道数 | min(每用进程数， 打开文件数) |
| 数值 | 双精度浮点数 (如果没有使用 MPFR) |
| 域大小 | MAX\_INT |
| 字串大小 | MAX\_INT |
| printf 字串大小 | MAX\_INT |

## C.5扩展 API 设计

这一小节，记录了扩展 API 的设计，包括一些需要解决的历史问题的讨论。

用于 gawk 第一个版本的扩展在 1990 年代中期开发，并与 gawk 3.1 一起发布。15 年来基本的机制并没有什么变量，直到 2012 年。

旧扩展的机制使用使用来自 gawk 的数据类型与函数，通过“聪明黑客”手段来安装扩展函数。

gawk 包含了一些扩展，很多都很有用。但是很明显，从一开始扩展机制都只强制一边，而没有经过详细思考。

### C.5.1旧机制的问题

旧的扩展机制有下面的几个问题：

* 严重依赖于 gawk 的内部实现。结点结构[[121]](#footnote-122)一旦变化，扩展就得重新编译。而且，要真写一扩展，需要了解 gawk 内部功能的一些东西。这里面有一些记录在本书里，但是相当有限。
* 要能够从扩展调用到 gawk，需要连接工具在 Unix 派生出来的系统中是很通用的，但是在 MS-Windows 上却不是，用户在 MS-Windows 上的扩展需要静态地连接到 gawk 中，尽管 MS-Windows 也支持动态装载共享对象。
* 当 gawk 变化时，API 也会偶尔变量，并且不同的版本之间不兼容，也没有相应的计划来兼容。

尽管有这么些缺陷，xgawk 项目的开发者复制了 gawk 并且开发了几个很好的扩展。它们也增强了 gawk 中与文件包含与共享对象访问的功能。

新的 API 经过的长时间的设计，直到 2012 年，gawk 的维护者与开发者最终才开始实现。关于 xgawk 项目的信息在 ，***在第 400 页***，这里提供。

### C.5.2新机制的目标

新 API 的目标是：

* API 应该独立于 gawk 的内部实现。gawk 的内部实现的变更不对于扩展函数的实现者是不可见的。
* API 应该提供二进制兼容性，只要 gawk 的发行版本之间 API 没有变更。
* API 应该可以用 C 或者 C++ 来写，使其与 awk 层的代码有一样的“外观”，就像 awk 函数一样。这就是说，扩展应该：
  + 有能力访问函数参数。
  + 有能力将一个未定义的函数转换为数组（通过引用调用）。
  + 有能力创建，访问与更新全局变量。
  + 方便一次性访问数组的所有元素（“数组扁平化”），使其可以用 C 代码的循环就可以迭代数组中的所有的元素。
  + 有能力创建数组（包含 gawk 真正的数组之数组）。

其他的重要目标是：

* API 应该只使用 ISO C 90 的特性，这样扩展可以使用最广泛的 C 与 C++ 编译器。头文件应该包含适当的 ‘#ifdef \_\_cplusplus’ 与 ‘extern "C"’ 等代码，这样可以使用 C++ 编译器。（如果使用 C++ 时，运行时系统要足够聪明来调用任何的构造函数与析构函数，因为 gawk 是一个 C 程序。在写本书时，这还没有测试过。）
* API 机制应该不需要访问 gawk 的符号，[[122]](#footnote-123)无论是编译时还是动态连接时。这样创建的扩展也可以用在 MS-Windows 平台上。

在开发的过程中，越来越清晰的是有一些特性需要提供给扩展，后面也提供了：

* 扩展应该有能力将勾子打入 gawk 的 I/O 重向机制中。特别是 xgawk 的开发者提供了一种所谓的“开放勾子”来读取记录。在开发过程中，扩展对于输入处理、输出处理以及双路 I/O 的勾子都进了一般化。
* 扩展应该可以提供回调函数来执行 gawk 退出时的清理动作。
* 扩展应该可以提供版本字串，这样 gawk 的 --version 选项可以提供扩展的相关信息。

初看起来，避免访问 gawk 的符号是比较难实现的一个。

在设计上，很明显 Perl 与 Ruby 以及其他的语言，可以使主线的 gawk 代码成为一个库，gawk 使用小的 C main() 函数来连接这个库。

这看起来像摇着尾巴的狗，会导致构建与安装复杂化，使得在不同的系统中（或者同一个系统中的不同地方）都使用一个单独的 gawk 执行文件。

我们接受了 Pat Rankin 提出的建议。查看 ，***在第 350 页***，来获取详细信息。

### C.5.3其他的设计决策

作为任意的设计决策，扩展可以读取预定义变量与数组的值（如 ARGV 与 FS），但是不可以改变它们，除了 PROCINFO 外。原因是为了扩展函数影响 awk 程序流，使其不可按。而一个真正的 awk 函数可以做它自己想做的事情，这个则由程序员自己来判断。扩展函数 应该提供一个服务或者使 C API 可以在 awk 中可以使用。而不是将 FS ，ARGC 以及 ARGV 弄乱。

另外，很容易就可以在一个滑滑的斜坡上开头（即一旦开启刹不住——译者注）。扩展到底需要访问多少的 gawk 工具？它们是否需要 getline？是否可以调用 gsub() 或者编译正则表达式？直接调用 awk 函数如何？（这会弄得乱乱的）

为了避免这样的问题，gawk 开发者以最容易的，最基础的但是确实有用的开始。

另一个决策是，尽管 gawk 提供了像 MPFR 这样的好东西，内部使用整数索引的数组，这些特性并没有开放出来，为的是保证事情的简单，并封闭 awk 的传统的语义（不扩张其语义范围，如 gawk 是用来做字符处理的工具，而不是数学运算的工具——译者注）。（事实上，内部使用整数来索引数组如此透明，在文档中都没有记录！）

其次，在 API 中的所有函数都要检查它们的指针输入参数是不是都不是 NULL。如果是，就返回错误。（对于扩展代码来说，验证从 gawk 返回的指针是不是 NULL 也是个好主意。这样的事情不会发生，但是 gawk 开发者也是人，他们据知一直以来也会偶尔犯错误。）

随着时间的推移，API 也会演化，gawk 开发者希望这可以由用户的需要来驱动。目前，当前的 API 看起来是提供了最小的，但是功能强大的用来创建扩展的特性集合。

### C.5.4未来的增长空间

后面 API 可以两种方式被扩展：

* 在第一次装载时，gawk 传递一个 “扩展 id”到扩展中。相应的扩展然后在每个函数调用时，传递这个 id 回 gawk。这个机制可以使 gawk 来区别调用它的扩展，而它也应该知道。
* 类似的，在注册时，扩展传递一个“namespace”到 gawk 中。这个可以实现一个未来的机制，如对扩展功能进行分组，以避免名字的冲突。当然，在写这个文字的时候，对上面的东西都还没有做相应的决策。

## C.6对旧扩展的兼容性

*，****在第 349 页***，中描述 了所支持 API 以及为 gawk 写扩展的机制。这个 API 在 4.1 中引入。但是，很多年以来，gawk 提供的扩展机制都需要了解 gawk 的内部，这设计得并不好。

为了提供一个转换时期，gawk 版本 4.1 还会支持最初的扩展机制。这对于一个主要的发行版本周期是这样的。在下一个主要版本发布时，这个支持也会被丢弃，并且从源代码中移除。

简单地说，初始风格的扩展可以包含 awk.h 来进行编译。另外 ，你必须在构建时定义标识符‘GAWK’（在 Unix 风格的编译器中使用 ‘-DGAWK’）。否则，在 gawkapi.h 中的定义会与 awk.h 中的定义相冲突，你的扩展就没办法成功编译。

如前一个版本，你使用 extension() 内置函数来装载一个旧风格的扩展（这个并没有文档）。这个函数然后找到并装载包含了扩展的共享对象文件，并调用 dl\_load() C 函数。

因为 初始风格 与新风格的扩展使用不同的初始化函数（dl\_load() 与 dlload()）他们可以安全地装载在同一个目录下（可以在 AWKLIBPATH 中找到）而没有冲突。

gawk 的开发团队强烈建议你将旧的扩展进行转换，这样不得不使用新的 API，它们都在 *，****在第 349 页***，中进行了描述。

## C.7总结

* gawk 的扩展可以使用 --traditional 或者 --posix 选项被禁用。如果 gawk 被编译 时使用了 ‘***-DDEBUG***’，则 --parsedebug 选项也可以使用。
* gawk 的源代码通过公共的 Git 库进行维护。所有人都可以检出并查看代码。
* 欢迎对 gawk 作出贡献。本章中随后的步骤指出步骤可以让你的贡献更容易地与代码库进行整合。对即可用于新特性的贡献，也用于将 gawk 移植到其他的系统。
* gawk 有一些限制——一般都是由机器的架构所导致的。
* 扩展 API 的设置目标是解决前一个扩展机制的大量问题，启用 xgawk 项目所需要的特性，并提供后向的二进制兼容性。
* 之前的扩展机制在 gawk 4.1 版本还是支持的，但是会在下一个主要发布版本中移除。

# 附录D 基本编程概念

这一附录中尝试定义一些基本的概念与术语，它们会在本书中余下部分使用到。由于本书是一本关于 awk 的书，而不是计算通用编程的书，这里所包含的内容相当粗糙简单。（如果需要更多的背景，你可以参考更多入门性质的内容。）

## D.1程序做的是什么

最基本的，程序的任务是处理输入数据，然后产生输出结果。如图 D.1。

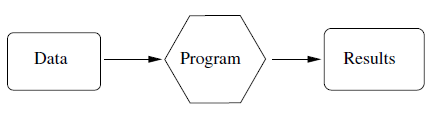


Figure D.1: General Program Flow

图 D.1：通用的程序流程

图中的“程序”可以是一个编译型程序[[123]](#footnote-124)（如 ls），或者是解释型程序。后者中，特定机器上的可执行程序，如 awk 读入你的程序，然后使用你程序中的指令来处理数据。

当写程序时，其包含如下最基本的步骤，如图 D.2：

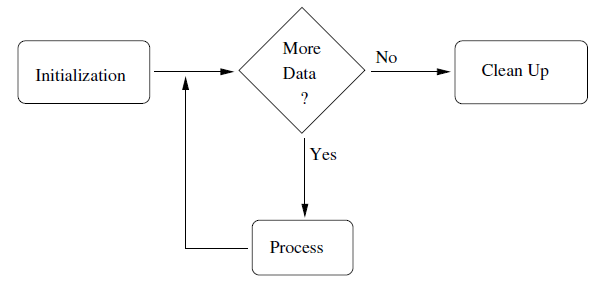


Figure D.2: Basic Program Steps

初始化

这是在程序开始处理数据之前所要做的事情，如检查参数，初始化需要使用的数据等等。这个步骤对应于 awk 的 BEGIN 规则（查看 ， ***在第 141 页***）。

就像你烤蛋糕，这可能包含将所有的不同的碗与烤箱摆放好，并确保准备好所有的原料。

处理 这里执行实际工作。你的程序数据，一次一个逻辑块，然后处理它。

在多数的编程语言中，你需要手工管理数据的读取，检查看是否一次读取了多块。awk 的模式-动作范围（查看 *，* ***在第 20 页***）帮你处理了这个机制。

在烤面包中，处理对应于实际的劳动：打蛋，和面、水及其他原料，并且将它们放到烤箱中。

Clean Up 一旦处理完了所有的数据，你退出前你还需要做一些事情。这一个步骤对应于 awk 中的 END 规则（查看 ， ***在第 141 页***）。

当蛋糕从烤箱中取出后，你还需要将他们放到塑料盒中以防别是染指，并洗干净碗及其他器皿。

算法是详细的指令集，由此来完成工作或者处理数据。这个与烤蛋糕的方法是一样的。程序实现了算法。通常，是由你来设计并实现算法。

我们在前面所说的“逻辑块”称为记录，如一个公司对员工所做的记录，或者学校对学生的记录，或者医生对病人所做的记录类似。每个记录都有很多组件，如名与姓，生日，地址等等。组件指的是记录中的域

读取数据的动作称为输入，毫无疑问，产生结果称为输出。它们一般合称为“输入/输出”，更常用的是 “I/O”的简称（你也可以看到将 “输入”与“输出”当成动词使用。）

awk 简单数据的读取，以及将记录分拆成域。你的程序的任务就是告诉 awk 如何来处理数据。你通过描述要查找数据的模式，以及当发现这些模式的记录时所做的动作来完成你所要完成的任务。这是 awk 程序数据驱动的特点，因此使用这些程序易写易读。

## D.2计算机中数据的值

在程序中，你会跟踪事件的值与信息，我们称之为变量。一个变量只是一个给定的值的名字。对于当前的输入记录以及记录的域有特殊的名字。你可以将一组有关的值组合在一起而成为数组。

数据，特别是在 awk 中，由数组，如 42 或者 3.1415927 或者字串组成。字串实质上是非数值外的一切。字串有时指字符值，因为们每存储组成字串的单个的字符。单个的变量，包含数组与字串变量都指的是标量值。一组值，如数组，就不是标量。

在 ，***在第 336 页***，中提供了数值类型（整数与浮点数）的基本介绍，以及它们在计算机中如何被使用。请参考那里的信息，其中也包含一些注意事项。

当你可能习惯于没有数没有值（如 0）的概念时，你要习惯于零长度的字符数组可能要多费点功夫。但是，确实存在这样的事情。它被称为空串。空串是字符数据中没有值。就是说，它是空的。它在 awk 程序中写成：""。

人类习惯于以十进制，如以 10 为底，在以 10 为底的情况下，数字从 0 到 9，然后进行进位到下一个数字位。（还记得学校里的 42 =4 x 10 + 2 吗）

有其他的数值底数。计算一般使用底数为 2，或者二进制，底数为 8 或者八进制，以及底数为 16 或者十六进制。在二进制中，每一个数值位表示 2 乘以所在数字列的值。每个位要么包含 0，要么包含1.因此二进制 1010 表示 (1 x 8) + (0 x 4) + (1 x 2) + (0 x 1)，或者十进制的 10。八进制与十六进制在这里有更多的讨论 ，***在第 113 页***。

在很底层的层级上，计算机都将值存成一组二进制数字，或者是位。现代计算机将这些位组成在8位一组，称为字节。高级的应用程序有时需要直接处理位操作，gawk 也提供了这些的操作函数。

程序用程序语言写成。如果没有上千，也是成百的程序语言存在。最流行的是 C 程序语言。C 语言对于 awk 语言的设计有巨大的影响。

有几个版本的 C 语言。第一个称为 “K&R” C，由 Brian Kernighan 与 Dennis Ritchie 发明，也是第一本关于 C 语言书的作者。（Dennis Ritchie 创造了这个语言，Brian Kernighan 是 awk 的作者之一。）

在 1980年代中期，人们开始努力形成一个 C 的国际标准。这个工作在 1989 年完成，产生了 C 的 ANSI 标准。这个标准在 1990 年成为 ISO 标准。1999 年，一个修订版的 ISO C 标准提出并发布。因此 ，POSIX awk 是与 ISO 1999 的 C 的兼容的。

# 名词解释

Action（动作）

一连串的附加到规则中的语名。如果规则的模式匹配于输入记录，则 awk 会执行规则的动作。动作总是包含在花括号中。（查看 ， ***在第 145 页***）

Ada

一种最初由美国国防部定义的编程语言。它设计用来强制使用一种好的软件工程实践。

Amazing ***awk*** Assembler

来自多伦多大学的 Henry Spencer 写了一个可重定目标机器的汇编器作为 sed 与 awk 的脚本。它只有几千行，包含了几个8位的微型计算机的机器描述。它应该是用另一个语言写成的很好的程序例子。你可以从这里取得：http://***awk***.info/?awk100/aaa。

Amazingly Workable Formatter (awf)

来自多伦多大学的 Henry Spencer 用 awk 与 sh 写了一个格式化器，可以接受 ‘***nroff -ms***’ 与‘***nroff -man***’格式命令的很大的子集。可以从这里获取 ：http://***awk***.info/?tools/awf

Anchor（锚）

正则表达式元字符 ‘^’ 与 ‘$’，用来强制匹配串的开始与结束。

ANSI

美国国家标准学会。这个组织产生了很多的标准，它们中有 C 与 C++ 的编译语言标准。这些标准通常也会成为国际标准。查看 “ISO”。

Argument（参数）

参数可以表示两个不同的东西。可以是一个选项或者是传递到一个从命令行中调用的命令的文件，或者是传递到程序，如在 awk 中函数中的任何东西。

在后一种情况，参数可以通过两种方式来传递。一种是将值传递给被调用的函数，如变量的一个复本可以在被调用的函数中使用，但是原始的值不可以被函数更改；或者是通过引用来传递，如一个变量的指针传递给函数，这样，函数就可以直接修改它。在 awk 中，标量都是通过值传递，而数组是通过引用传递。查看 “以值/引用传递”。

Array （数组）

同一个名字下的一组数值。很多的语言只提供顺序数组。awk 提供关联数组。

Assertion （断言）

在程序中的语句指出程序在中某点上条件为真。对于合理化程序的行为很有用。

Assignment（赋值）

一个 awk 表达式，用一塌糊涂改变 awk 变量或者数据对象的值。一个对象可被赋值则称为 lvalue。被赋的值称为 rvalue。查看 ，***在第 122 页***。

Associative Array（关联数组）

数组的索引可以是数值，也可以是字串，不仅仅是固定长度的顺序整数。

***awk*** Language（awk 语言）

awk 程序所使用的语言。

***awk*** Program（awk 程序）

awk 程序由一连续的模式与动作组成。这些动作统称为规则。对于每个输入的程序的记录，程序的规则会顺序处理。awk 程序也包含定义的函数。

***awk*** Script （awk 脚本）

awk 程序的别一个名字。

Bash

GNU 版的标准 Shell（the Bourne-Again SHel）。查看 “Bourne Shell”。

Binary

基于两个符号，其值为 0 与 1。因为电路“自然地”以2 为底进行工作（可以相像成开与关），在计算机的所有东西都以 2 为底进行计算。每一位都表示 2 的幂，被称为位。所以，以2为底的值 10101 与十进制的 10 ((1 x 16) + (1 x 4) + (1 x 1)) 相同。

由于以2为底的数值很快变得太长而难于读写，它们一般被按3位进行分组（即它们可以按八进制来读），或者按4位进行分组（即它们可以按十六进制来读）。在 C 程序没有直接的方式来插入以2为底的数。存储在寄存器中以2为底用来表示计算机中的整数，可以直接表示了计算机的处理能力。现令的大多数计算机使用 64 位来表示整数，但是过去很多使用 32 位，16 位或者8位。查看 ，***在第 113 页***。

Bit （位）

“二进制数字”的简称。在计算机内存中的所有值最终都成为二进制数字：要么是 1 要么是 0。一组位则可以有不同的解释——当成整数，浮点数，字符，另一个内存对象的地址等等。awk 可以让你处理浮点数与字串。gawk 可以让你用位操作函数来操作位（查看 ，***在第 201 页***）。

计算机一般都是以用多少位来表示整数来定义的。典型的系统有 32 位系统，但是 64 位系统也起来越流行，16位的系统会慢慢消失。

Boolean Expression（布林表达式）

以英国数数家 Boole 来命令，查看“逻辑表达式”。

Bourne Shell

在 Unix 或者类 Unix 系统上的标准 Shell（/bin/sh），最初由贝尔实验室的 Steven R. Bourne 写成。很多的 Shell （Bash， ksh， pdksh， zsh）都与 Bourne Shell 兼容。

Braces （花括号）

字符 ‘{’ 与 ‘}’。花括号用在 awk 用于区分由语句与函数体组成的动作。

Bracket Expression（方括号表达式）

在正则表达式中，包含在方括号中的表达式，用来代表属于某个字符类的字符。方括号表达式可以包含一个或者多个字符的列表，如 ‘[abc]’，或者字符范围，如 ‘[A-Z]’，或者一个用 ‘：’分割的名字，其表示一个已知的字符集，如 ‘[:digit:]’。包含在 ‘:’之间的表达式的方括号表达式形式与字符本身的表示是无关的，可以是 ASCII ，也可以ECBDIC，还可以是 Unicode 编码集，则要视计算系统的架构与本地配置。查看“正则表达式”。

Built-in Function（内置函数）

awk 语言提供了内置函数来执行各种数值、I/O相关、字串计算等操作。例子有 sqrt()（用来计算数组的平方根）与 substr()（用来取字串的子串）。gawk 提供了用于时间戳处理，位操作，数组排序，类型检查以及运行时字串翻译的函数。（查看 ，***在第 182 页***。）

Built-in Variable（内置变量）

ARGC， ARGV， CONVFMT， ENVIRON， FILENAME， FNR， FS， NF， NR， OFMT， OFS， ORS， RLENGTH， RSTART， RS， 与 SUBSEP 是对 awk 都有特殊的含义的变量。另外，ARGIND， BINMODE， ERRNO， FIELDWIDTHS， FPAT， IGNORECASE， LINT， PROCINFO， RT， 与 TEXTDOMAIN 是对于 gawk 有特殊含义的变量。对些变量的更改，会影响 awk 的运行时环境。（查看 ，***在第 155 页***。）

C

很多的 GNU 软件都使用其编写的一种系统编程语言。awk 编程语言有 C 风格的语法，本书中，我们在合适的时候也提出了 awk 与 C 之间的差异。一般来讲，gawk 尝试使其与 1990 年的 ISO C 版本相似还有有意义的。

C Shell

C Shell（csh 或者其增强版本 tcsh）是一个 Unix Shell，由 Bill Joy 在 1970 年晚些时候编写。C Shell 与其他的 Shell 之间的不同在于其交互特性以整体风格，它看起来更像 C。C Shell 没有向后兼容 Bourne Shell，所以，将其他的 Unix Shell 写的脚本转换成 C Shell 的时候要特别注意，特别是对于变量的管理。查看 “Bourne Shell”。

C++

由 C 衍生出来的一种面向对象的很流行的编程语言。

Character Class

查看“方括号表达式”。

Character List

查看“方括号表达式”。

Character Set（字符集）

由计算机系统用来表示特定国家或者地区的字符（字母，数字，标点等）的数值编码集合。目前使用最多的常用字符集是 ASCII（美国信息交换标准编码）。很多的欧洲国家使用 ASCII 的扩展集，ISO-8859-1（ISO-Latin-1）。Unicode 字符集使用得越来越普遍，并形成相应的标准，特别是在 GNU/Linux 系统上被广泛地使用。

CHEM

pic 的预处理器，用来读取分子的描述并产生 pic 输入，用于描述。它由 Brian Kernighan 与 Jon Bentley 用 awk 写成。并在此可得到：http://netlib.org/typesetting/chem。

Comparison Expression（比较表达式）

结果为 true 或者 false 的关系，如 ‘a<b’。比较表达式使用在 if，while，do 与 for 语句中，以及模式中用来从输入记录中选择记录进行处理。（查看 ，***在第 127 页***。）

Compiler（编译器）

用来将人可读的源代码翻译成机器可执行对象代码的程序。对象代码然后直接被计算机执行。查看“解释器”。

Complemented Bracket Expression（补方括号表达式）

方括号表达式的反面。表示所有的没有在给定的方括号表达式的东西。符号 ‘^’放在反方括号表达式的前面。如：‘[[^:digit]]’表示所有的非数字字符。‘[^abd]’表示不是 ‘b’，‘a’，与 ‘d’的所有字符。查看“方括号表达式”。

Compound Statement（复合语句）

连续的包含在花括号中的 awk 语句。复合语句可以内嵌。（查看 ，***在第 146 页***。）

Computed Regexps（计算型正则表达式）

查看 “动态正则表达式”。

Concatenation（连接）

将两个字串进行连接表示将它们合成一块，一个接一个，来产生一个新的串。如串‘foo’ 与串 ‘bar’进行连接，会产生串‘foobar’。（查看 ，***在第 121 页***。）

Conditional Expression（条件表达式）

使用 ‘?:’三元操作符的表达式，如 ‘expr1?expr2:expr3’。表达式 expr1 求值，如果结果为 true，则整个表达式的值为 expr2 的值，否则为 expr3 的值。在每种情况下，只会对 expr2 或者 expr3 之一进行求值。（查看 ，***在第 132 页***。）

Control Statement（控制语句）

控制语句是 awk 程序中的，如果条件为 true，则会执行一个执行给定操作或者操作集的指令。控制语句有：if，for，while，以及 do（查看 ， ***在第 146 页***）。

Cookie

一种特殊的甜点符号，由程序产生或者传递给程序的影像。

Coprocess （并程）

一种子程序，通过它可进行两路通信。

Curly Braces（曲括号）

查看“花括号”。

Dark Corner（暗角）

语言规范中常常（或者依然）不清楚的地方，从而导致非期望或者不想要的行为。这些地方在本书中通过在页边空白用手电图标标志出来，并在 “暗角”段中进行了索引。

Data Driven（数据驱动）

awk 程序的描述，表示你指定你感兴趣的数据来进行处理，以及在发现这些数据时所执行的动作。

Data Objects（数据对象）

为数值或者字符串。在需要的时候，数值被转换成字串，或者反之。（查看 ，***在第 117 页***。）

Deadlock（死锁）

两个通信的进程相互在等待对方执行某种动作的状态。

Debugger （调试器）

用来帮助开发者从程序中去掉 Bug（de-bug）的程序。

Double Precision（双数度）

数值的内部表达，这个表式可以有分数部分。双精度数可以比单精度跟踪更多的数值位，但是对他们的操作的代码也更高。这是 awk 存储数据的方式。它的类型为 C 的 double 类型。

Dynamic Regular Expression（动态正则表达式）

动态正则表达式是写在常规表达式的正则表达式。它可以是一个字串常量，如 “foo”，但是也可以是基值可以变化的表达式。（查看 ，***在第 58 页***。）

Empty String

查看“Null String”

Environment（环境）

字串的集合，其格式为‘name=val’，每个程序都可以取得它。用户通常将值放到环境中，以向不同的程序传递信息。典型的例子是环境变量 HOME 与 PATH。

Epoch

用来表示时间戳的“开始时间的”日期。大多数系统中的时间值都用从 epoch 开始的秒来表示。使用库的函数可将这些值转换为标准日期以及时间格式。Unix 与 POSIX 系统上的 epoch 是 1970-01-01 00：00：00 UTC。查看 “GMT” 与 “UTC”。

Escape Sequences（转义序列）

特殊字符序列用来描述非打印字符，如‘\n’用来表示换行行符，或者‘\033’用来表示 ASCII 中的 ESC（Escape）字符。（查看 ，***在第 51 页***。）

Extension （扩展）

额外的特性，或者成为编程语言或者工具，但未被相应语言以及工具标准所定义。gawk （也）有许多的在POSIX awk 之外的扩展。

FDL

查看 “自由文档许可”。

Field （域）

当 awk 读取一个输入记录时，它将记录分割为由空白（或者通过改变预定义的 FS 变量而将正则表达式设置成的分割符）分割的片段。这些片段被称为域。如果片段是固定长度的，你可以使用内置变量 FIELDWIDTHS 来描述它们的长度。如果你希望指定域 的内容而不是域分割符，你可以使用即定义的变量 FPAT 来处理。（查看 ，***在第 71 页***，，***在第 77 页***，与 ，***在第 79 页***。）

Flag（标志）

一个变量，其真值表示某种条件是存在还是不存在。

Floating-Point Number（浮点数）

在数学的术语中常指为“有理数”或者实数，这只是一个有小数部分的数。查看“双精度”与“单精度”。

Format （格式）

格式串控制 strftime() 与 sprintf() 函数的输出样式，同时也会控制 print 语句中的样式。另外，将数值转换为字串的转换也由预定义的变量 CONVFMT 与 OFMT 的格式串来控制。（查看 ，***在第 98 页***。）

Fortran

FORmula TRANslator 的缩写，第一个用于科学计算的编程语言。由 John Backus 创建 ，并在 1957 年就发布，目前依然还在使用。

Free Documentation License（自由文档许可）

这个文档描述了一些条款，本书也在这些条款下进行发布，也可以复制。（查看 ，***在第 466 页***。）

Free Software Foundation（自由软件基金会）

一个非盈利性组织，其致力于生产与发布自由发布的软件。由 Richard M. Stallman，即 Emacs 编辑器的最初作者创立。GNU Emacs 是目前使用最多的 Emacs 版本。

FSF See“Free Software Foundation.”

查看“自由软件基金会”。

Function （函数）

是 awk 程序的一部分，可以在程序中的任何位置进行调用，以执行某个任务。awk 有多个内置函数。用户可以定义他们自己的函数，可以放在程序中的任何位置。函数可以是递归的，即它们可以调用自己。查看 ***， 在第 182 页***。在 gawk 中，也可以在不同的程序中共享函数，在需要的程序中使用 @include 指令来包含（查看 ， ***在第 46 页***）。在 gawk 中，可以调用的函数的名字可以在运行时生成，即动态生成。gawk 扩展 API 提供了构造函数（查看 ， ***在第 357 页***）。

gawk The GNU implementation of ***awk***.

awk 的 GNU 版本。

General Public License（通用公共许可）

些文件描述的一些条款，gawk 以及它的原代码在这些条款下发布。（查看 ***，在第 465 页***）

GMT

“Greenwich Mean Time”（格林尼治标准时间）。这是 UTC 的旧术语。它是 Unix 与 POSIX 系统内部使用的时期时间。查看 “Epoch” 与 “UTC”。

GNU

“GNU's not Unix”。是一个持续运行的自由软件基金会的项目，用来创建一个完全的，自由发布的 POSIX 兼容的计算环境。

GNU/Linux

是使用 Linux Kernel 的 GNU 系统的变体，而不是自由软件基金会的 Hurd 内核。Linux 内核是一个稳定，高效，全功能的 Unix 克隆，其被移植到不各种不同的构架中。它是最流行的 PC 级的系统，但是也在其他的系统中运行。Linux 内核源代码可以在 GNU 通用公共许可的条款下获取，这可能是其最重要方面。

GPL

查看“通用公共许可”

Hexadecimal（十六进制）

以 16 为底的表示法，其数字为 0-9 以及 A-F，‘A’表示 10，‘B’表示 11，等等，到到 ‘F’ 表示 15。十六进制数组在 C 中以 ‘0x’ 开头以表示其底数。因此 0x12 为 18（(1 x 16) +2）。查看 ， ***在第 113 页***。

I/O

“Input/Output”的缩写，是将数据移入或者移出执行程序的动作。

Input Record（输入记录）

由 awk 读取的一块数据。通常，一个 awk 输入记录由一行文本构成。（查看 ， ***在第 63 页***。）

Integer（整数）

完整之数，即没有包含小数部分的数值。

Internationalization（国际化）

编写或者修改程序的过程，使其可以在多种语言下使用，而不用修改源代码。

Interpreter（解释器）

读取人可读的源代码，然后使用其内部的指令来处理数据并产生结果的程序。awk 典型的情况下（但并不总是）实现为解释器。查看“编译器”。

Interval Expression（区间表达式）

正则表达式的一个组件，可以让你重复匹配正则表达式的某个部分。区间表达式最初在 awk 程序中不可用。

ISO

标准化的国际组织。这个组织为各个方面产生国际标准，包含程序语言，如 C 与 C++。在计算时代，重要的标准诸如 C，C++ 以及 POSIX 即成为美国美标，也会同时成为 ISO 的国际标准。这本书所指的标准 C 均为 “ISO C”。查看 ISO 的站点来获取更多的组织名字及其语言无关的三字母同意词。

Java

一个由 Sun Microsystems（现在为 Oracle）开发的现代编程语言，其支持面向对象的程序方式。尽管通常通过编译为标准虚拟机指令的方式来实现（JVM），这个语言也可以编译为本地代码。

Keyword （关键词）

在 awk 语言中，关键词有特殊的含义。关键词是保留的，不可以使用的变量名。gawk 的关键词有：BEGIN， BEGINFILE， END， ENDFILE， break， case， continue， default delete， do...while， else， exit， for...in， for， function， func， if， next， ***nextfile***， switch， 与 while。

Korn Shell

Korn Shell（ksh）是由在贝尔实验室的 David Korn 在 1980 年代早期开发的。Korn Shell 与 Bourne Shell 以及很多的 C shell 特性是后向兼容的。查看 “Bourne Shell”。

Lesser General Public License（次通用公共许可）

这个文档描述了相应条款，在此条款下可以发布二进制文档或者共享对象，也可以发布相应的源代码。

LGPL

查看“次通用公共许可”。

Linux

查看 “GNU/Linux”。

Localization（本地化）

为已经国际化的程序提供必要的数据以在特定的语言中运行的过程。

Logical Expression（逻辑表达式）

使用逻辑操作符 AND，OR 与 NOT 的表达式，在 akw 中分别写在 ‘&&’，‘||’，以及 ‘！’。通常也称为布尔表达式。以开创这种数学逻辑的数学家来命令。

Lvalue（左值）

可以出现在赋值表达式左边的表达式。在很多语言中，左值（lvalue）可以是变量，也可以是数组元素。在 awk 中，域引用也可以用作左值。

Matching （匹配）

将一个字串与正则表达式进行测试的动作。如果正则表达式描述了字串的内容，则说是匹配的。

Metacharacters（元字符）

在正则表达式中并不用作为其自身的那些字符。相反，它们代表正则表达式操作，如重复，组合，选择等。

Nesting（内嵌）

内嵌指的是信息以层的方式进行组织，或者对象包含了类似的对象。在 gawk 中，@include 指令可以内嵌。算术与逻辑操作“自然的”内嵌可以通过括号来改变（查看 ， ***在第 135 页***。）

No-op

什么也不做的操作。

Null String

没有字符的串。在 awk 程序中，其由两个紧邻的引号（""）来表示。它会出现在两个紧邻的域分隔符间。

Number（数值）

一个数值数据对象。现代的 awk 实现使用双精度浮点数来表示数值。古董级的 awk 实现使用单粒度浮点数。

Octal（八进制）

以 8 为底的表示法，其数字为 0 - 7。八进制在 C 中使用前导的 ‘0’来表示其底数。因此，013 表示 13（(1 x 8) + 3）。查看 ， ***在第 113 页***。

Output Record（输出记录）

由 awk 输出的一块数据。通常 awk 输入由一到多行的文本组成。查看 ， ***在第 63 页***。

Pattern（模式）

模式告诉 awk 输入记录中哪些是规则所感兴趣的。模式是任意的条件表达式，用于对输入记录进行测试。如果条件满足，则说模式与输入记录是匹配的。一个典型的模式会将输入记录与正则表达式匹配。（查看 ， ***在第 138 页***。）

PEBKAC

是计算机使用问题最可能的来源的同义词。（Problem Exists Between Keyboard And Chair。）

Plug-in（插件）

查看“扩展”。

POSIX

指定了可移植操作系统接口（Portable Operating System interface）一系列标准名字。“IX”表示这些标准来自于 Unix。awk 用户感兴趣的主要标准是用于信息技术的 IEEE 标准，1003.1-2008 标准。2008 POSIX 标准可以在线访问：http://www.opengroup.org/onlinepubs/9699919799/

Precedence（优先级）

当操作符使用时没有显式的括号时，操作符的执行顺序。

Private（私有）

由库函数而不是主 awk 程序使用的变量与/或者函数。对这些变量与函数的命令要特别注意。（查看 ， ***在第 222 页***。）

Range (of input lines)（（输入行的）范围）

从输入文件中输入了一连串的连续行。一个模式可以指定输入行范围，使 awk 可以处理，或者它也可以指定单独的行。（查看 ， ***在第 138 页***。）

Record（记录）

查看“输入记录”与“输入记录”。

Recursion（递归）

函数直接或者间接调用它自己。如果这非常清楚，则停止，并进入下一个词条。否则参考“递归”项。（这是作用在用递归方式描述这个词条——译者注）

Redirection（重定向）

重定向表示从其他的地方而不是标准输入来执行输入，或者输出到其他地方而不是标准输出。你可以使用 ‘<’，‘|’，‘|&’操作符将输入重定向到 getline 语句。也可以使用 ‘’，‘’，‘’与‘’ 操作符将 print 与 printf 语句的输出重定向到文件或者一个系统命令。（查看 ，***在第 83 页*** 与 ， ***在第 103 页***。）

Reference Counts（参考记数）

在 gawk 中的一种内部机制，以最小化内存使用来存储字串变量的值。如果值所假设的变量的值在多于一个地方使用，只会有一个复本保留，当相应的值被另一个变量使用时，则参考计数会被递增，并且在相关的变量不再使用时，则递减它。当参考计算变为 0 时，存储变量值的内存空间会被释放掉。

Regexp

查看“正则表达式”。

Regular Expression（正则表达式）

正则表达式（“regexp”为简称）是一种模式，用来表示字串集，有可能是无限集。例如，正则表达式‘***R.\*xp***’匹配以字母 ‘R’开头，并以 ‘xp’结尾的所有字串。在 awk 中，正则表达式使用在模式以及条件表达式中。正则表达式可以包含转义序列。（查看 ***， 在第 50 页***。）

Regular Expression Constant（正则表达式常量）

正则表达式常量是一个写在斜杠中的正则表达式，如 /foo/。这个正则表达式在写 awk 程序时就已经选定，并且在执行的过程中不可以更改。（查看 ， ***在第 50 页***。）

Regular Expression Operators（正则表达式操作符）

查看“元字符”。

Rounding（舍入舍出）

对算术操作的结果进行舍入舍出会有诡异。有多于一种的舍入舍出方式，在 gawk 中，可以选择其中一种用在程序中。查看 ，***在第 344 页***。

Rule （规则）

awk 程序中的一段，用来指定如何来处理单个的输入记录。一个规则由一个模式与一个动作组成。awk 读取输入记录，然后针对每一个规则，如果输入记录满足规则的模式，则 awk 会执行规则的动作。否则，规则不会对输入记录做任何操作。

右值（Rvalue）

可以出现在赋值操作符右边的值。在 awk 中，本质上每一个表达式都有一个值，这些值就是佑值。

Scalar（标量）

单一的值，可以是一个数值或者是一个字串。常规的变量是标题，数组与函数则不是。

Search Path（搜索路径）

在 gawk 中，是用于 awk 程序搜索文件的目录列表。在 Shell 中，是用来搜索可执行文件的路径的列表。

sed

查看“流编辑器”。

Seed

一系列随机数的初始值，或者开始值。

Shell

用于 Unix 或者 POSIX 兼容系统的命令解释器。Shell 即可以用于交互式，也可以用于指处理文件或者 Shell 脚本的编程语言。

Short-Circuit（短路）

awk 逻辑操作符 ‘&&’ 与‘||’的自然特性。如果整个表达式的值只需要对操作符的左边的表达式进行求值就可以确定，而右边的表达式则不会被求值。（查看 ，***在第 131 页***。）

Side Effect（副作用）

如果一个表达式仅产生一个值则有一个其他的效果会随着发生，则就会发生副作用。赋值表达式，递增与递减表达式，还有函数调用都有副作用。（查看 ，***在第 122 页***。）

Single Precision（单精度）

数值的内部表示，其可以有小数部分。单精度的数值所跟踪的数值位要比双精度的数值少，但是对这些数值的操作按 CPU 时间也会少得多。这种类型被一些很老的 awk 版本用来存储数值值。它的类型是 C 的 float。

Space（空格）

敲击键盘上的空格键而产生的字符。

Special File（特殊文件）

文件名由 gawk 在内部进行解释，而不是被操作系统来直接处理，比如，/dev/stderr。（查看 ，***在第 106 页***。）

Statement（语句）

awk 程序中模式-动作规则中的动作部分中的表达式，或者 awk 函数中的表达式。一个语句可以是一个变量赋值，一个数组操作，或者一个循环等等。

Stream Editor（流编辑器）

从输入流中读取记录，然后一次一个进行处理的程序。这个是批处理程序的反面。在批处理文件中，它希望它的所有输入文件在开始处理之前是完整的，同时交互式的程序则要求从用户中输入数据。

String （字串）

由一串字符所组成的数据，如‘I am a string’。在 awk 中，常量字串写在双引号中，并且可能包含转义字符。（查看 ，***在第 51 页***）

Tab

由敲击解盘的 TAB 键而产生的字符。它在输出中通常展开为 8 个空格。

Text Domain（文本域）

一个标识一种应用的唯一的名字。用来在运行时将翻译的消息分组到本地语言中。

Timestamp（时间戳）

一个“从epoch以来的秒数”格式的值，用在 Unix 与 POSIX 系统中。被用在 gawk 中的函数中，如 ***mktime()***， ***strftime()***， 与 ***systime()。查看 “Epoch”，“GMT”与“UTC”。***

Unix 一种计算机操作系统，在 1970 年代最初由 AT&T 贝尔实验室开发。一开始其在全世界的大学非常流行，然后进行了商用 化，作为软件开发系统以及网络服务器系统。有很多版本的 Unix，也有几个类型相似的可自由获取的 Unix 版本的源代码（如 GNU/Linux，NetBSD，FreeBSD 以及 OpenBSD）。

UTC 已经被授受为“Universal Coordinated Time（统一协调时）”的缩写。这是 England，Greenwich 标准时间，用于时间与日期计算的参考时间。查看 “Epoch”与 “GMT”。

Variable （变量）

值的名字。在 awk 中变量可以是标量，也可是是数组。

Whitespace（空白）

一连串的空格，TAB，或者换行符出现在输入记录或者字串中。

# GNU 通用公共许可

略

# GNU 自由文档许可

略

1. 这些工具在Posix 兼容的系统上都有，在传统的基于Unix 的系统上也存在。如果你使用的是一些其他的系统，你也需要熟悉一些IO重定向及管道的概念。 [↑](#footnote-ref-2)
2. 一些已经不用的系统，在这些系统上gawk 有过移植，但是已经不再支持，并且为支持这些系统而存在的代码也已经移除。 [↑](#footnote-ref-3)
3. SNOBOL语言，一种面向字符串符号（StriNg Oriented symBOlic Language）语言。——译者 [↑](#footnote-ref-4)
4. 只有在 Solaris 的系统中依然使用旧awk 作为默认的 awk 工具。更为现代的 awk 则位于这些系统中的 /usr/xpg6/bin 目录下。 [↑](#footnote-ref-5)
5. 所有这些差异都归集在“awk 与 gawk 之间的差异”下。 [↑](#footnote-ref-6)
6. 一种文档处理软件，用一种格式即可以完成网页输出以及打印输出。这种工具主要用于编写软件手册。——译者 [↑](#footnote-ref-7)
7. GNU 表示“GNU's Not Unix”。 [↑](#footnote-ref-8)
8. 术语“GNU/Linux”在 *，****在第 422 页***，进行说明。 [↑](#footnote-ref-9)
9. 我也不知道这是什么东西。 [↑](#footnote-ref-10)
10. 如在Linux 系统中用户目录下的 .profile 等文件中。——译者注 [↑](#footnote-ref-11)
11. ‘#!’ 这样的机制可以在GNU/Linux，基于BSD 的系统以及Unix 系统上使用。 [↑](#footnote-ref-12)
12. 硬链接(hard link， 也称链接)就是一个文件的一个或多个文件名。多数文件系统可以用多个文件名与同一个文件进行链接，这些文件名可以在同一目录或不同目录。一个文件有几个文件名，就说该文件的链接数为几。这个与软链接不同，软连接是指向文件名的文件。删除所有硬连接才能删除文件。删除被软连接的指向的文件（如果只有一个硬连接）就可以删除文件本身。——译者注 [↑](#footnote-ref-13)
13. 这里的‘？’ 与‘：’ 是一个三元操作符表达式，在下面这进行了描述，*在第 127 页*。在‘？’ 与‘：’ 之后进行断行是一个小扩展，如果指定了–posix参数，则这个特性就会被禁用。可以查看， 。 [↑](#footnote-ref-14)
14. 其他的流行的工具包括 Ruby 与Perl。 [↑](#footnote-ref-15)
15. 要获取更多的细节，请参考RFC3875 的Section 4.4。也可以查看发送到gawk 的bugmailing 列表中的解释。 [↑](#footnote-ref-16)
16. lint是一种工具，用来检测程序代码的规范性，而不是源代码逻辑错误。它对于程序的检查都是静态检查。由于lint 的广泛使用，也用lint 表示对程序的静态的，语言代码的规范性等检查。——译者注。 [↑](#footnote-ref-17)
17. 常量折叠是其中一种被很多现代编译器使用的编译器优化技术。常量折叠是在编译时间简单化常量表达的一个过程。简单来说就是将常量表达式计算求值，并用求得的值来替换表达式，放入常量表。可以算作一种编译优化。——译者注 [↑](#footnote-ref-18)
18. 不建议这么做。 [↑](#footnote-ref-19)
19. 在 MS-Windows 与 MS-DOS 中则使用分号“;”。 [↑](#footnote-ref-20)
20. 你的 gawk 版本可能使用不同的目录；这依赖于 gawk 是如何被构建与安装的。实际的路径是 $(datadir) 在 gawk 配置时所产生的值。尽管如此，一般来说你也不需要担心。 [↑](#footnote-ref-21)
21. 正则引擎主要可以分为两大类：一种是DFA（确定的有穷自动机，Deterministic finite automaton），一种是NFA（非确定的有穷自动机，Non-deterministic finite automaton)。这两种引擎都有了很久的历史，当中也由这两种引擎产生了很多变体！于是POSIX的出台规避了不必要变体的继续产生。这样一来，主流的正则引擎又分为3类：一、DFA，二、传统型NFA，三、POSIX NFA。DFA 引擎在线性式状态下执行，因为它们不要求回溯（并因此它们永远不测试相同的字符两次）。DFA 引擎还可以确保匹配最长的可能的字符串。但是，因为 DFA 引擎只包含有限的状态，所以它不能匹配具有反向引用的模式；并且因为它不构造显式扩展，所以它不可以捕获子表达式。——译者注 [↑](#footnote-ref-22)
22. 在其他说法中，你可能会看到方括号表达式指的是要么是一个字符集，或者字符类型，或者是一个字符列表。 [↑](#footnote-ref-23)
23. 如果你在正则表达式操作符或者函数中使用字串常量，你需要使用两个反斜杠。 [↑](#footnote-ref-24)
24. 有经验的 C 与 C++ 程序员会注意到，通过使用这样 ‘IGNORECASE =1 && /foObAr/ { ... }’ 和 ‘IGNORECASE = 0 || /foobar/ { ... }’的方式，这种情况是可能的。但是这多少有点含糊，因此我们不建议这么干。 [↑](#footnote-ref-25)
25. 如果你了解这些，就不用担心，这仅仅意味着 gawk 可以做正确的事情。 [↑](#footnote-ref-26)
26. 至少我们所知道的那些。 [↑](#footnote-ref-27)
27. 在 POSIX awk，换行符不被认为是用来进行域分隔的空格符。 [↑](#footnote-ref-28)
28. 二元操作符，如果‘\*’用于乘法，会使用两个操作数。之所以要做这样的区分，是因为 awk 也有一元操作符（只有一个操作数）以及三元操作符（三个操作数）。 [↑](#footnote-ref-29)
29. 感谢Andrew Schorr所做的提示。 [↑](#footnote-ref-30)
30. sed 工具称为“流编辑器”。它的行为也由 POSIX 标准来定义。 [↑](#footnote-ref-31)
31. CSV 格式缺乏正式的标准定义多年，RFC4180 标准化了多数的通用实践。 [↑](#footnote-ref-32)
32. 当 FS 为空串或者正则表达式，这个特殊的特性没有应用在 RS 中。但是却可以应用在当域分隔符为默认空格时：‘FS = ""’情况中。 [↑](#footnote-ref-33)
33. 并程，指相互关联的一组执行程序。——译者注 [↑](#footnote-ref-34)
34. 这个不太是这么回事。RT 在 RS 是一个正则表达式时可能会改变。 [↑](#footnote-ref-35)
35. 这里假设标准输入就是键盘。 [↑](#footnote-ref-36)
36. 在 /dev/tty 中的tty 代表的是“Teletype“（电传打字机），是一种串行终端。 [↑](#footnote-ref-37)
37. 技术术语应该是 病态进程（morbid），已经完成的子进程称为“僵尸进程（zombie）”，通过“回收（reaping）”后，进程就会被清理。 [↑](#footnote-ref-38)
38. 这是一个全16位的由 wait() 系统调用返回的值。查看系统手册来获取如何解码这个值的相关信息。 [↑](#footnote-ref-39)
39. 所有的数值，包括整数，在内部都表示成双精度的浮点数。在多数的现代系统中，使用的都是 IEEE754 的标准格式。查看 ***,在第 311 页***，来取得更多的信息。 [↑](#footnote-ref-40)
40. 在病理学情况下需要多达 752 位数（！），但是我们认为你不需要为此担心（你平时用不到这么长的有效位）。 [↑](#footnote-ref-41)
41. 这在 BWK awk 中会发生，gawk 以及 mawk 都“执行正确”，但是别依赖于这个结果。 [↑](#footnote-ref-42)
42. 从技术上说，字串比较应该的行为应该一致，就好像字串的的比较使用的是 C strcoll() 函数一样。 [↑](#footnote-ref-43)
43. 这个程序有一个 BUG，它会打印以 ‘END’ 开头的行。看看你怎么来修复它？ [↑](#footnote-ref-44)
44. awk 的最开始的版本会读取记录，但是会忽略它们，并一直到文件尾为止。 [↑](#footnote-ref-45)
45. 在 POSIX awk 中，换行符不被当成空白。 [↑](#footnote-ref-46)
46. 有一些 Unix awk 的早期实现将 FILENAME 的值设置成“-”，就算还没有文件被处理。这个行为是不正确的，在你的程序中不能依赖于它。 [↑](#footnote-ref-47)
47. 更不用提在实现上的困难。 [↑](#footnote-ref-48)
48. 有的语言声明与定义是两个含义（如 C，或者 C++）。声明是告诉变量的使用者变量的类型，大小等属性。而定义则是分配内存过程。由于这两个通过都是在一起发生的，有时就指两者为一。——译者注 [↑](#footnote-ref-49)
49. 顺序在不同的实现中是不同的，典型的情况下是吏用哈希表来存储数组元素与它的值。 [↑](#footnote-ref-50)
50. 当两个元素比较相同时，C 的 qsort() 函数不保证它们比较之后保持比较之前的相对顺序（不稳定排序）。通过字串比较，可以保证在数值比较为相同时，可提供唯一的顺序，因此 gawk 可以在不同的环境中行为一致。 [↑](#footnote-ref-51)
51. 感谢 Michael Brennan 提出这点。 [↑](#footnote-ref-52)
52. 大家都知道，在很多 Unix 系统上的 C 版本的 rand()，产生的随机数序列非常不好。但是 awk 版本的实现一点都没有使用 C 版本的 rand()。实事上，gawk 使用 BSD 的 random 函数，这个函数产生的随机数要比 rand() 好很多。 [↑](#footnote-ref-53)
53. mawk 每次都使用不同的种子。 [↑](#footnote-ref-54)
54. 计算机产生的随机数不是真正的随机数。在技术上，它们都是伪随机的。这就是说，尽管数值序列看起来是随机的，但是事实上会一次次地产生相同的随机序列。 [↑](#footnote-ref-55)
55. 除非你使用了 --non-decimal-data 选项，但我们不建议这么使用。查看 ，*在第 313 页*，取得更多的信息。 [↑](#footnote-ref-56)
56. 注意这意味着那条记录首先会使用 OFS 来重新生成，因为有域已经被改变了，并且在替换之后，域会更新，就算操作是一个“no-op”，例如 ‘sub(/^/，"")’。 [↑](#footnote-ref-57)
57. 那个人真是幼稚，尽管这个小节中有一个提示说下个主要的版本会跟 POSIX 保持一致。 [↑](#footnote-ref-58)
58. 如果标准输出连接到一个终端设备，则程序是交互式的。在现代系统中，这表示设备是你的键盘或者屏幕。 [↑](#footnote-ref-59)
59. 查看 *,****在第 422 页***，特别是 “Epoch” 与 “UTC” 项。 [↑](#footnote-ref-60)
60. GNU 的 date 工具也可以做这里所描述的很多的事情。如果在 Shell 中处理的是简单的与时间有关的操作，则可优先使用它。 [↑](#footnote-ref-61)
61. 在偶尔的情况下，一年中有几分钟会有润秒，因此秒数可以到 60。 [↑](#footnote-ref-62)
62. 不幸的是，不是所有的系统的 strftime() 都必须支持这里所列的所有转换。 [↑](#footnote-ref-63)
63. 如果你不理解这个，也不用担心，这个设施主要用于方便实现 “国际化” 程序。其他的国际化特性在 ***， 在第 287 页***，这里描述。 [↑](#footnote-ref-64)
64. 这是因为 ISO C 将 C 版本的 strftime() 保留其行为为未定义的，gawk 就使用其所在系统的 strftime() 的版本。典型的情况下，转换指定符要么不会出现在返回的字串里，要么就以这面的方式出现。 [↑](#footnote-ref-65)
65. 这个例子显示在左边会用0进行填补。对于 gawk，这总是正确的，但是在其他语言中，有可能会用 1 进行填补。 [↑](#footnote-ref-66)
66. 这个程序实际上不会运行，因为 foo() 是未定义的。 [↑](#footnote-ref-67)
67. 2012 年的晚些时候。 [↑](#footnote-ref-68)
68. 这个在未来的版本可能会改变：要检查与你的 gawk 一起的文档来查看是否已经改变。 [↑](#footnote-ref-69)
69. 很惨的是，35 年之后，这本书中所教的经验还没有被大量的实践中的程序员所学到。 [↑](#footnote-ref-70)
70. 效果是不完全相同。变化记录的输出所有的都是小写。而 IGNORECASE 保持了输入记录的最初值。 [↑](#footnote-ref-71)
71. 尽管所有的库例程都使用这样的惯例进行了重写，但是还没有完成。为了显示我们自己的 awk 程序风格的演化，也为了这些讨论提供一些基础素材。 [↑](#footnote-ref-72)
72. gawk 的 --dump-variables 命令行选项用来验证这个非常有用。 [↑](#footnote-ref-73)
73. 这个正在改变：很多的系统使用 Unicocde。这是一种很大的字符集，ASCII 是它的一个子集。在完全支持 Unicode 的系统中，一个字符可以占到 32 位。使得这里所使用的简单测试有可能也非常消耗性能。 [↑](#footnote-ref-74)
74. ASCII 被扩展到很多的国家，并使用到 128 到 255 之间的值表示特定国家的字符。如果你的系统使用这些扩展，你可简化 \_ord\_init() 来迭代从0 到 255 的数值。 [↑](#footnote-ref-75)
75. 如果 awk 有一个赋值操作符来进行连接就很好了。缺少一个显式的连接操作符使得字串操作比实际需要的要难一些。 [↑](#footnote-ref-76)
76. BEGINFILE 特定模式（查看 ）提供了另一种机制来处理那些不可以打开的文件。但是，这里的代码提供了一种可移植的方案。 [↑](#footnote-ref-77)
77. 这个版本在 gawk 要求用 “”将字串分隔为字符的能力之前写成的。我们已经不管它了，因为用 substr() 可移植性更好。 [↑](#footnote-ref-78)
78. 用记的密码信息很多情况下都存储在网络数据库中。 [↑](#footnote-ref-79)
79. 则才提交的代码有一个很细微的问题。假设第一次时没有名字时，则这个代码会将名字前面带逗号加入。同时也没有检测是否有第4个域，$4。 [↑](#footnote-ref-80)
80. 这里同时也引入了一个隐蔽的 Bug，如果发生了匹配，我们输出被转换了的行，而不是原来的行。 [↑](#footnote-ref-81)
81. 这是传统的用法。POSIX 的用法是不同的，但是对于要程序要说明的东西是无关的。 [↑](#footnote-ref-82)
82. 由于 gawk 可以处理多字节的本地语言设置，所以这个代码统计的是字符，而不是字节。 [↑](#footnote-ref-83)
83. 在一些旧的系统上，包括 Solaris，tr 的系统版本可能需要列表要写在范围表达式，并包含在方括号中，引起来，为的是避免 Shell 会尝试对文件名扩展。但这不是个特性。 [↑](#footnote-ref-84)
84. “真实世界” 定义为 “就是实际上用来处理某些事情的程序”。 [↑](#footnote-ref-85)
85. 完整的 sh 语言的解释超出了这本书的范围。我们提供了一些最小的说明，但是如果你想更深入地理解，请阅读一个好的 Shell 编程书。 [↑](#footnote-ref-86)
86. 在一些非常老的 awk 版本中，如果文件存在但为空，‘getline junk < t’ 这样的测试能够永远循环。 [↑](#footnote-ref-87)
87. gawk 自己来处理 @include，为的是支持将 awk 程序用作 WEB CGI 脚本。 [↑](#footnote-ref-88)
88. 这个是由输入 State Machine 到 Google 返回的定义。 [↑](#footnote-ref-89)
89. 这就是为什么预定义的排序顺序都以 ‘@’字符开头，它不可以是标识符的一部份。 [↑](#footnote-ref-90)
90. 因为基于本地语言设置的比较只会出现在 POSIX 的兼容模式中，因此这确实是真的，并且由于 asort() 与 asorti() 是 gawk 的扩展，在上面的设置中，它们是不可用的。 [↑](#footnote-ref-91)
91. Michael Brenna 建议使用 rand() 来产生唯一的文件名。这是个有效的观点，但是，临时文件比起双路管道来讲还是困难些。 [↑](#footnote-ref-92)
92. 这与 Bash 与 C Shell 中的操作符非常不同。 [↑](#footnote-ref-93)
93. 对于某些操作系统，gawk 的移植不支持 GNU gettext。因此，如果你在这个系统上使用，这个特性就不可用。非常抱歉！ [↑](#footnote-ref-94)
94. 美国使用每个三个进制数之间放一个逗号，以及用圆点来作为十进制小数点，而很多的欧洲国家则相反：1，234.56 与 1.234，56。 [↑](#footnote-ref-95)
95. 感谢 Bruno Haible 提供这个例子。 [↑](#footnote-ref-96)
96. 与 GNU gettext 一起发布的 xgettext 工具可以处理 .awk 文件。 [↑](#footnote-ref-97)
97. 这个例子是从 GNU gettext 手册上借来的。 [↑](#footnote-ref-98)
98. 这是一个很好的用来存放“令人困惑的 awk”内容的目录。 [↑](#footnote-ref-99)
99. 也许，我们称之为 “Hippy”可能会更好。 [↑](#footnote-ref-100)
100. 我们不知道它们为什么希望有这个能力，从是确实是有这种期望。 [↑](#footnote-ref-101)
101. 关于浮点数算术运算，有一个非常好的论文，由 David Goldberg 所写，“What Every ComputerScientist Should Know About Floating-Point Arithmetic”，ACM Computing Surveys 23， 1 (1991-03):5-48.如果你对于细节非常感兴趣，这个值得一读，但是确实需要一定的计算机科学背景。 [↑](#footnote-ref-102)
102. 如果在你的系统上的 C 库不支持使用 IEEE754 的偶数舍入舍出规则来为 printf 舍入舍出半数的情况，有可能输出完全不同。 [↑](#footnote-ref-103)
103. Weisstein， Eric W. Sylvester 序列，来自于 MathWorld——一个Wolfram WEB 资源（http://mathworld.wolfram.com/SylvestersSequence.html）。 [↑](#footnote-ref-104)
104. 自作自受。 [↑](#footnote-ref-105)
105. 查看 the “cookie” entry in the Jargon file 来查看 Cookie 的定义，以及 the “magic cookie” entry in the Jargon file 来查看其中的例子。也可看 ，*在第 511 页*，中的 “Cookie”词条。 [↑](#footnote-ref-106)
106. 由于 API 只使用 ISO C 90 的特性，因此不能使用 ISO C 99 的可能参数宏特性来隐藏参数。多少有点遗憾。 [↑](#footnote-ref-107)
107. 相信我们，差异是可测量并且真实的。 [↑](#footnote-ref-108)
108. 数值值问题就少得多，因为只需要存储一个 C 的 double 类型。 [↑](#footnote-ref-109)
109. 好吧，也是唯一的数据结构。 [↑](#footnote-ref-110)
110. 它也是一个 “Cookie”，但是 gawk 开发得不想过多地使用这个术语。 [↑](#footnote-ref-111)
111. 为了表现，这个版本有一些小改动。到 gawk 发布版 extension/filefuncs.c 中查看完整的版本。 [↑](#footnote-ref-112)
112. 在实践中，你可能想使用 GNU Autotools（Automake，Autoconf，Libtool，以及 gettext）来配置及构建你的库。如何来做超出了本书的范围。查看 ，***在第*** *455* ***页***，来获取这些工具的地址。 [↑](#footnote-ref-113)
113. 那么生活就美妙了。 [↑](#footnote-ref-114)
114. 也因此产生了“合理范围解释报怨”（或者 RRI）。很多的 GNU 工具已经实现了这样的变化，或者不久将实现。非常感谢 Karl Berry 发明了“合理范围解释”这个词。 [↑](#footnote-ref-115)
115. 查看标准与其解释。 [↑](#footnote-ref-116)
116. 2014 年的 11 月，这个站点都还在，但是作者没有找到 GNU Make 包。 [↑](#footnote-ref-117)
117. IA 64 平台也称为 “Itanium”。 [↑](#footnote-ref-118)
118. 我们试过，但是非常痛苦。 [↑](#footnote-ref-119)
119. 有一个 GNU 程序，（以我们的观点看）很难从 Git 库中生成。比如，在作者的旧的（但是还工作）的 PowerPC Macintosh ，安装的是 Mac OS X 10.5，很有必要从 Git 库本身生成和很多软件，使其能够与最新的代码一起工作。这一点也不让人感觉愉快，特别是在一些老旧的系统中，这简直就是浪费时间。

     使用最新的 tarball 也不需要担心。维护都已经不用 .gz 与 .bz2 文件了，而只发布 .tar.xz 文件。有必要先安装 xz。 [↑](#footnote-ref-120)
120. 一个由其他开发者生成的分支（已经删除），并没有包含所生成的文件。 [↑](#footnote-ref-121)
121. gawk 中的关键的核心数组结构。 [↑](#footnote-ref-122)
122. 符号是定义在 gawk 中的变量与函数。在 gawk 外部的运行时动态装载代码中对这些符号的访问在 MS-Windows 平台上是有问题的。 [↑](#footnote-ref-123)
123. 编译型程序一般使用低阶语言来写，如 C ，C++ 或者 Ada，然后被翻译或者说是编译为计算机可直接执行的形式。 [↑](#footnote-ref-124)