通用线程：**Awk** 实例，第 **1**部分

一种名称很奇特的优秀语言介绍

Daniel Robbins
总裁兼 CEO, Gentoo Technologies, Inc.
2000 年 12 月

Awk是一种非常好的语言，同时有一个非常奇怪的名称。在本系列（共三篇文章）的第一篇文章中，DanielRobbins 将使您迅速掌握 awk编程技巧。随着本系列的进展，将讨论更高级的主题，最后将演示一个真正的高级awk 演示程序。

捍卫 **awk**

在本系列文章中，我将使您成为精通 awk 的编码人员。我承认，awk 并没有一个非常好听且又非常“时髦”的名字。awk 的 GNU 版本（叫作 gawk）听起来非常怪异。那些不熟悉这种语言的人可能听说过 "awk"， 并可能认为它是一组落伍且过时的混乱代码。它甚至会使最博学的 UNIX 权威陷于错乱的边缘（使他不断地发出 "kill -9!" 命令，就象使用咖啡机一样）。

的确，awk 没有一个动听的名字。但它是一种很棒的语言。awk 适合于文本处理和报表生成，它还有许多精心设计的特性，允许进行需要特殊技巧程序设计。与某些语言不同，awk 的语法较为常见。它借鉴了某些语言的一些精华部分，如 C 语言、python 和 bash（虽然在技术上，awk 比 python 和 bash 早创建）。awk 是那种一旦学会了就会成为您战略编码库的主要部分的语言。

第一个 **awk**

让我们继续，开始使用 awk，以了解其工作原理。在命令行中输入以下命令：

```
$ awk '{ print }' /etc/passwd
```

您将会见到 /etc/passwd 文件的内容出现在眼前。现在，解释 awk 做了些什么。调用 awk 时，我们指定 /etc/passwd 作为输入文件。执行 awk 时，它依次对 /etc/passwd 中的每一行执行 print 命令。所有输出都发送到 stdout，所得到的结果与与执行 catting /etc/passwd完全相同。

现在，解释 `{ print }` 代码块。在 awk 中，花括号用于将几块代码组合到一起，这一点类似于 C 语言。在代码块中只有一条 print 命令。在 awk 中，如果只出现 print 命令，那么将打印当前行的全部内容。

这里是另一个 awk 示例，它的作用与上例完全相同：

```
$ awk '{ print $0 }' /etc/passwd
```

在 awk 中， `$0` 变量表示整个当前行，所以 `print` 和 `print $0` 的作用完全一样。

如果您愿意，可以创建一个 awk 程序，让它输出与输入数据完全无关的数据。以下是一个示例：

```
$ awk '{ print "" }' /etc/passwd
```

只要将 "" 字符串传递给 print 命令，它就会打印空白行。如果测试该脚本，将会发现对于 /etc/passwd 文件中的每一行，awk 都输出一个空白行。再次说明， awk 对输入文件中的每一行都执行这个脚本。以下是另一个示例：

```
$ awk '{ print "hiya" }' /etc/passwd
```

运行这个脚本将在您的屏幕上写满 hiya。:)

多个字段

awk 非常善于处理分成多个逻辑字段的文本，而且让您可以毫不费力地引用 awk 脚本中每个独立的字段。以下脚本将打印出您的系统上所有用户帐户的列表：

```
$ awk -F":" '{ print $1 }' /etc/passwd
```

上例中，在调用 awk 时，使用 -F 选项来指定 ":" 作为字段分隔符。awk 处理 `print $1` 命令时，它会打印出在输入文件中每一行中出现的第一个字段。以下是另一个示例：

```
$ awk -F":" '{ print $1 $3 }' /etc/passwd
```

以下是该脚本输出的摘录：

```
halt7
operator11
root0
shutdown6
sync5
bin1
....etc.
```

如您所见，awk 打印出 /etc/passwd 文件的第一和第三个字段，它们正好分别是用户名和用户标识字段。现在，当脚本运行时，它并不理想 -- 在两个输出字段之间没有空格！如果习惯于使用 bash 或 python 进行编程，那么您会指望 `print $1 $3` 命令在两个字段之间插入空格。然而，当两个字符串在 awk 程序中彼此相邻时，awk 会连接它们但不在它们之间添加空格。

以下命令会在这两个字段中插入空格：

```
$ awk -F":" '{ print $1 " " $3 }' /etc/passwd
```

以这种方式调用 print 时，它将连接 $1 、" " 和 $3 ，创建可读的输出。当然，如果需要的话，我们还可以插入一些文本标签：

```
$ awk -F":" '{ print "username: " $1 "\t\tuid:" $3" }' /etc/passwd
```

这将产生以下输出：

```
username: halt          uid:7
username: operator      uid:11
username: root          uid:0
username: shutdown      uid:6
username: sync          uid:5
username: bin           uid:1
....etc.
```

外部脚本

将脚本作为命令行自变量传递给 awk 对于小的单行程序来说是非常简单的，而对于多行程序，它就比较复杂。您肯定想要在外部文件中撰写脚本。然后可以向 awk 传递 -f 选项，以向它提供此脚本文件：

```
$ awk -f myscript.awk myfile.in
```

将脚本放入文本文件还可以让您使用附加 awk 功能。例如，这个多行脚本与前面的单行脚本的作用相同，它们都打印出 /etc/passwd 中每一行的第一个字段：

```
BEGIN {
    FS=":"
}

{ print $1 }
```

这两个方法的差别在于如何设置字段分隔符。在这个脚本中，字段分隔符在代码自身中指定（通过设置 FS 变量），而在前一个示例中，通过在命令行上向 awk 传递 -F":" 选项来设置 FS。通常，最好在脚本自身中设置字段分隔符，只是因为这表示您可以少输入一个命令行自变量。我们将在本文的后面详细讨论 FS 变量。

BEGIN 和 **END** 块

通常，对于每个输入行，awk 都会执行每个脚本代码块一次。然而，在许多编程情况中，可能需要在 awk 开始处理输入文件中的文本之 前 执行初始化代码。对于这种情况，awk 允许您定义一个 BEGIN 块。我们在前一个示例中使用了 BEGIN 块。因为 awk 在开始处理输入文件之前会执行 BEGIN 块，因此它是初始化 FS（字段分隔符）变量、打印页眉或初始化其它在程

序中以后会引用的全局变量的极佳位置。

awk 还提供了另一个特殊块，叫作 END 块。awk 在处理了输入文件中的所有行之后执行这个块。通常，END 块用于执行最终计算或打印应该出现在输出流结尾的摘要信息。

规则表达式和块

awk 允许使用规则表达式，根据规则表达式是否匹配当前行来选择执行独立代码块。以下示例脚本只输出包含字符序列 `foo` 的那些行：

```
/foo/ { print }
```

当然，可以使用更复杂的规则表达式。以下脚本将只打印包含浮点数的行：

```
/[0-9]+\.[0-9]*/ { print }
```

表达式和块

还有许多其它方法可以选择执行代码块。我们可以将任意一种布尔表达式放在一个代码块之前，以控制何时执行某特定块。仅当对前面的布尔表达式求值为真时，awk 才执行代码块。以下示例脚本输出将输出其第一个字段等于 `fred` 的所有行中的第三个字段。如果当前行的第一个字段不等于 `fred`，awk 将继续处理文件而不对当前行执行 `print` 语句：

```
$1 == "fred" { print $3 }
```

awk 提供了完整的比较运算符集合，包括 "==="、"<"、">"、"<="、">=" 和 "!="。另外，awk 还提供了 "~" 和 "!~" 运算符，它们分别表示“匹配”和“不匹配”。它们的用法是在运算符左边指定变量，在右边指定规则表达式。如果某一行的第五个字段包含字符序列 `root`，那么以下示例将只打印这一行中的第三个字段：

```
$5 ~ /root/ { print $3 }
```

条件语句

awk 还提供了非常好的类似于 C 语言的 `if` 语句。如果您愿意，可以使用 `if` 语句重写前一个脚本：

```
{
    if ( $5 ~ /root/ ) {
        print $3
    }
}
```

这两个脚本的功能完全一样。第一个示例中，布尔表达式放在代码块外面。而在第二个示例中，将对每一个输入行执行代码块，而且我们使用 `if` 语句来选择执行 print 命令。这两个方法都可以使用，可以选择最适合脚本其它部分的一种方法。

以下是更复杂的 awk `if` 语句示例。可以看到，尽管使用了复杂、嵌套的条件语句， `if` 语句看上去仍与相应的 C 语言 `if` 语句一样：

```
{
    if ( $1 == "foo" ) {
        if ( $2 == "foo" ) {
            print "uno"
        } else {
            print "one"
        }
    } else if ($1 == "bar" ) {
        print "two"
    } else {
        print "three"
    }
}
```

使用 `if` 语句还可以将代码：

```
! /matchme/ { print $1 $3 $4 }
```

转换成：

```
{
    if ( $0 !~ /matchme/ ) {
        print $1 $3 $4
    }
}
```

这两个脚本都只输出 不 包含 `matchme` 字符序列的那些行。此外，还可以选择最适合您的代码的方法。它们的功能完全相同。

awk 还允许使用布尔运算符 "||"（逻辑与）和 "&&"（逻辑或），以便创建更复杂的布尔表达式：

```
( $1 == "foo" ) && ( $2 == "bar" ) { print }
```

这个示例只打印第一个字段等于 `foo` 且 第二个字段等于 `bar` 的那些行。

数值变量！

至今，我们不是打印字符串、整行就是特定字段。然而，awk 还允许我们执行整数和浮点运算。通过使用数学表达式，可以很方便地编写计算文件中空白行数量的脚本。以下就是这样一个脚本：

```
BEGIN   { x=0 }
/^$/    { x=x+1 }
END     { print "I found " x " blank lines. :)" }
```

在 BEGIN 块中，将整数变量 x 初始化成零。然后，awk 每次遇到空白行时，awk 将执行 x=x+1 语句，递增 x 。处理完所有行之后，执行 END 块，awk 将打印出最终摘要，指出它找到的空白行数量。

字符串化变量

awk 的优点之一就是“简单和字符串化”。我认为 awk 变量“字符串化”是因为所有 awk 变量在内部都是按字符串形式存储的。同时，awk 变量是“简单的”，因为可以对它执行数学操作，且只要变量包含有效数字字符串，awk 会自动处理字符串到数字的转换步骤。要理解我的观点，请研究以下这个示例：

```
x="1.01"
# We just set x to contain the *string* "1.01"
x=x+1
# We just added one to a *string*
print x
# Incidentally, these are comments :)
```

awk 将输出：

```
2.01
```

有趣吧！虽然将字符串值 1.01 赋值给变量 x ，我们仍然可以对它加一。但在 bash 和 python 中却不能这样做。首先，bash 不支持浮点运算。而且，如果 bash 有“字符串化”变量，它们并不“简单”；要执行任何数学操作，bash 要求我们将数字放到丑陋的 $( )) 结构中。如果使用 python，则必须在对 1.01 字符串执行任何数学运算之前，将它转换成浮点值。虽然这并不困难，但它仍是附加的步骤。如果使用 awk，它是全自动的，而那会使我们的代码又好又整洁。如果想要对每个输入行的第一个字段乘方并加一，可以使用以下脚本：

```
{ print ($1^2)+1 }
```

如果做一个小实验，就可以发现如果某个特定变量不包含有效数字，awk 在对数学表达式求值时会将该变量当作数字零处理。

众多运算符

awk 的另一个优点是它有完整的数学运算符集合。除了标准的加、减、乘、除，awk 还允许使用前面演示过的指数运算符 "^"、模（余数）运算符 "%" 和其它许多从 C 语言中借入的易于使用的赋值操作符。

这些运算符包括前后加减（ i++ 、 --foo ）、加／减／乘／除赋值运算符（ a+=3 、 b*=2 、 c/=2.2 、 d-=6.2 ）。不仅如此 -- 我们还有易于使用的模／指数赋值运算符（ a^=2 、 b%=4 ）。

字段分隔符

awk 有它自己的特殊变量集合。其中一些允许调整 awk 的运行方式，而其它变量可以被读取以收集关于输入的有用信息。我们已经接触过这些特殊变量中的一个，FS。前面已经提到过，这个变量让您可以设置 awk 要查找的字段之间的字符序列。我们使用 /etc/passwd 作为输入时，将 FS 设置成 ":"。当这样做有问题时，我们还可以更灵活地使用 FS。

FS 值并没有被限制为单一字符；可以通过指定任意长度的字符模式，将它设置成规则表达式。如果正在处理由一个或多个 tab 分隔的字段，您可能希望按以下方式设置 FS：

```
FS="\t+"
```

以上示例中，我们使用特殊 "+" 规则表达式字符，它表示“一个或多个前一字符”。

如果字段由空格分隔（一个或多个空格或 tab），您可能想要将 FS 设置成以下规则表达式：

```
FS="[[:space:]+]"
```

这个赋值表达式也有问题，它并非必要。为什么？因为缺省情况下，FS 设置成单一空格字符，awk 将这解释成表示“一个或多个空格或 tab”。在这个特殊示例中，缺省 FS 设置恰恰是您最想要的！

复杂的规则表达式也不成问题。即使您的记录由单词 "foo" 分隔，后面跟着三个数字，以下规则表达式仍允许对数据进行正确的分析：

```
FS="foo[0-9][0-9][0-9]"
```

字段数量

接着我们要讨论的两个变量通常并不是需要赋值的，而是用来读取以获取关于输入的有用信息。第一个是 NF 变量，也叫做“字段数量”变量。awk 会自动将该变量设置成当前记录中的字段数量。可以使用 NF 变量来只显示某些输入行：

```
NF == 3 { print "this particular record has three fields: " $0 }
```

当然，也可以在条件语句中使用 NF 变量，如下：

```
{
    if ( NF > 2 ) {
        print $1 " " $2 ":" $3
    }
}
```

记录号

记录号 (NR) 是另一个方便的变量。它始终包含当前记录的编号（awk 将第一个记录算作记录号 1）。迄今为止，我们已经处理了每一行包含一个记录的输入文件。对于这些情况，NR 还会告诉您当前行号。然而，当我们在本系列以后部分中开始处理多行记录时，就不会再有这种情况，所以要注意！可以象使用 NF 变量一样使用 NR 来只打印某些输入行：

```
(NR < 10 ) || (NR > 100) { print "We are on record number 1-9 or 101+" }
```

另一个示例：

```
{
    #skip header
    if ( NR > 10 ) {
        print "ok, now for the real information!"
    }
}
```

awk 提供了适合各种用途的附加变量。我们将在以后的文章中讨论这些变量。

现在已经到了初次探索 awk 的尾声。随着本系列的开展，我将演示更高级的 awk 功能，我们将用一个真实的 awk 应用程序作为本系列的结尾。同时，如果急于学习更多知识，请参考以下列出的参考资料。

参考资料

- 您可以参阅本文在 developerWorks 全球站点上的 英文原文.
- 如果想看好的老式书籍，O'Reilly 的 sed & awk, 2ndEdition是极佳选择。
- 请参考 comp.lang.awkFAQ 。它还包含许多附加 awk 链接。
- Patrick Hartigan 的 awk tutorial 还包括了实用的 awk 脚本。
- Thompson's TAWKCompiler 将 awk 脚本编译成快速二进制可执行文件。可用版本有 Windows 版、OS/2 版、DOS 版和 UNIX 版。
- The GNUAwk User's Guide可用于在线参考。

关于作者

Daniel Robbins 居住在新墨西哥州的 Albuquerque。他是 Gentoo Technologies, Inc. 的总裁兼 CEO， **Gentoo Linux**（用于 PC 的高级 Linux）和 **Portage** 系统（Linux 的下一代移植系统）的创始人。他还是 Macmillan 书籍 *Caldera OpenLinux Unleashed*、 *SuSE Linux Unleashed* 和 *Samba Unleashed* 的合作者。Daniel 自二年级起就与计算机某些领域结下不解之缘，那时他首先接触的是 Logo 程序语言，并沉溺于 Pac-Man 游戏中。这也许就是他至今仍担任 **SONY Electronic Publishing/Psygnosis** 的首席图形设计师的原因所在。Daniel 喜欢与妻子 Mary 和新出生的女儿 Hadassah 一起共度时光。可通过 drobbins@gentoo.org 与 Daniel 联系。

对本文的评价

您对这篇文章的看法如何？

○ 太差！ (1) ○ 需提高 (2) ○ 一般；尚可 (3) ○ 好文章 (4) ○ 真棒！(5)

建议？

通用线程：**Awk** 实例，第 **2**部分

记录、循环和数组

Daniel Robbins
总裁兼 CEO, Gentoo Technologies, Inc.
2001 年 1 月

在这篇 awk简介的续集中，Daniel Robbins 继续探索awk（一种很棒但有怪异名称的语言）。Daniel将演示如何处理多行记录、使用循环结构，以及创建并使用awk数组。阅读完本文后，您将精通许多 awk的功能，而且可以编写您自己的功能强大的 awk 脚本。

多行记录

awk 是一种用于读取和处理结构化数据（如系统的 /etc/passwd 文件）的极佳工具。/etc/passwd 是 UNIX 用户数据库，并且是用冒号定界的文本文件，它包含许多重要信息，包括所有现有用户帐户和用户标识，以及其它信息。在我的 前一篇文章 中，我演示了 awk 如何轻松地分析这个文件。我们只须将 FS（字段分隔符）变量设置成 ":"。

正确设置了 FS 变量之后，就可以将 awk 配置成分析几乎任何类型的结构化数据，只要这些数据是每行一个记录。然而，如果要分析占据多行的记录，仅仅依靠设置 FS 是不够的。在这些情况下，我们还需要修改 RS 记录分隔符变量。RS 变量告诉 awk 当前记录什么时候结束，新记录什么时候开始。

譬如，让我们讨论一下如何完成处理“联邦证人保护计划”所涉及人员的地址列表的任务：

```
Jimmy the Weasel
100 Pleasant Drive
San Francisco, CA 12345

Big Tony
200 Incognito Ave.
Suburbia, WA 67890
```

理论上，我们希望 awk 将每 3 行看作是一个独立的记录，而不是三个独立的记录。如果 awk 将地址的第一行看作是第一个字段 ($1)，街道地址看作是第二个字段 ($2)，城市、州和邮政编码看作是第三个字段 $3，那么这个代码就会变得很简单。以下就是我们想要得到的代码：

```
BEGIN {
    FS="\n"
    RS=""
}
```

在上面这段代码中，将 FS 设置成 "\n" 告诉 awk 每个字段都占据一行。通过将 RS 设置成 ""，还会告诉 awk 每个地址记录都由空白行分隔。一旦 awk 知道是如何格式化输入的，它就可以为我们执行所有分析工作，脚本的其余部分很简单。让我们研究一个完整的脚本，它将分析这个地址列表，并将每个记录打印在一行上，用逗号分隔每个字段。

address.awk

```
BEGIN {
    FS="\n"
    RS=""
}

{
    print $1 ", " $2 ", " $3

}
```

如果这个脚本保存为 address.awk，地址数据存储在文件 address.txt 中，可以通过输入 "awk -f address.awk address.txt" 来执行这个脚本。此代码将产生以下输出：

```
Jimmy the Weasel, 100 Pleasant Drive, San Francisco, CA 12345
Big Tony, 200 Incognito Ave., Suburbia, WA 67890
```

OFS 和 ORS

在 address.awk 的 print 语句中，可以看到 awk 会连接（合并）一行中彼此相邻的字符串。我们使用此功能在同一行上的三个字段之间插入一个逗号和空格 (", ")。这个方法虽然有用，但比较难看。与其在字段间插入 ", " 字符串，倒不如让通过设置一个特殊 awk 变量 OFS，让 awk 完成这件事。请参考下面这个代码片断。

```
print "Hello", "there", "Jim!"
```

这行代码中的逗号并不是实际文字字符串的一部分。事实上，它们告诉 awk "Hello"、"there" 和 "Jim!" 是单独的字段，并且应该在每个字符串之间打印 OFS 变量。缺省情况下，awk 产生以下输出：

```
Hello there Jim!
```

这是缺省情况下的输出结果，OFS 被设置成 " "，单个空格。不过，我们可以方便地重新定义 OFS，这样 awk 将插入我们中意的字段分隔符。以下是原始 address.awk 程序的修订版，它使用 OFS 来输出那些中间的 ", " 字符串：

address.awk 的修订版

```
BEGIN {
    FS="\n"
    RS=""
    OFS=", "
}

{
    print $1, $2, $3

}
```

awk 还有一个特殊变量 ORS，全称是“输出记录分隔符”。通过设置缺省为换行 ("\n") 的 OFS，我们可以控制在 print 语句结尾自动打印的字符。缺省 ORS 值会使 awk 在新行中输出每个新的 print 语句。如果想使输出的间隔翻倍，可以将 ORS 设置成 "\n\n"。或者，如果想要用单个空格分隔记录（而不换行），将 ORS 设置成 ""。

将多行转换成用 **tab** 分隔的格式

假设我们编写了一个脚本，它将地址列表转换成每个记录一行，且用 tab 定界的格式，以便导入电子表格。使用稍加修改的 address.awk 之后，就可以清楚地看到这个程序只适合于三行的地址。如果 awk 遇到以下地址，将丢掉第四行，并且不打印该行：

```
Cousin Vinnie
Vinnie's Auto Shop
300 City Alley
Sosueme, OR 76543
```

要处理这种情况，代码最好考虑每个字段的记录数量，并依次打印每个记录。现在，代码只打印地址的前三个字段。以下就是我们想要的一些代码：

适合具有任意多字段的地址的 **address.awk** 版本

```
BEGIN {
    FS="\n"
    RS=""
    ORS=""
}

{
        x=1
        while ( x<NF ) {
                print $x "\t"
                x++
        }
        print $NF "\n"
}
```

首先，将字段分隔符 FS 设置成 "\n"，将记录分隔符 RS 设置成 ""，这样 awk 可以象以前一样正确分析多行地址。然后，将输出记录分隔符 ORS 设置成 ""，它将使 print 语句在每个调用结尾 不 输出新行。这意味着如果希望任何文本从新的一行开始，那么需要明确写入 `print "\n"` 。

在主代码块中，创建了一个变量 x 来存储正在处理的当前字段的编号。起初，它被设置成 1。然后，我们使用 while 循环（一种 awk 循环结构，等同于 C 语言中的 while 循环），对于所有记录（最后一个记录除外）重复打印记录和 tab 字符。最后，打印最后一个记录和换行；此外，由于将 ORS 设置成 ""，print 将不输出换行。程序输出如下，这正是我们所期望的：

我们想要的输出。不算漂亮，但用 **tab** 定界，以便于导入电子表格

```
Jimmy the Weasel        100 Pleasant Drive      San Francisco, CA 12345
Big Tony        200 Incognito Ave.      Suburbia, WA 67890
Cousin Vinnie   Vinnie's Auto Shop      300 City Alley  Sosueme, OR 76543
```

循环结构

我们已经看到了 awk 的 while 循环结构，它等同于相应的 C 语言 while 循环。awk 还有 "do...while" 循环，它在代码块结尾处对条件求值，而不象标准 while 循环那样在开始处求值。它类似于其它语言中的 "repeat...until" 循环。以下是一个示例：

do...while 示例

```
{
    count=1
    do {
        print "I get printed at least once no matter what"
    } while ( count != 1 )
}
```

与一般的 while 循环不同，由于在代码块之后对条件求值，"do...while" 循环永远都至少执行一次。换句话说，当第一次遇到普通 while 循环时，如果条件为假，将永远不执行该循环。

for 循环

awk 允许创建 for 循环，它就象 while 循环，也等同于 C 语言的 for 循环：

```
for ( initial assignment; comparison; increment ) {
    code block
}
```

以下是一个简短示例：

```
for ( x = 1; x <= 4; x++ ) {
    print "iteration",x
}
```

此段代码将打印：

```
iteration 1
iteration 2
iteration 3
iteration 4
```

break 和 continue

此外，如同 C 语言一样，awk 提供了 break 和 continue 语句。使用这些语句可以更好地控制 awk 的循环结构。以下是迫切需要 break 语句的代码片断：

while 死循环

```
while (1) {
    print "forever and ever..."
}
```

while 死循环 1 永远代表是真，这个 while 循环将永远运行下去。以下是一个只执行十次的循环：

break 语句示例

```
x=1
while(1) {
    print "iteration",x
    if ( x == 10 ) {
        break
    }
    x++
}
```

这里，break 语句用于“逃出”最深层的循环。"break" 使循环立即终止，并继续执行循环代码块后面的语句。

continue 语句补充了 break，其作用如下：

```
x=1
while (1) {
    if ( x == 4 ) {
        x++
        continue
    }
    print "iteration",x
    if ( x > 20 ) {
        break
    }
    x++
}
```

这段代码打印 "iteration 1" 到 "iteration 21"，"iteration 4" 除外。如果迭代等于 4，则增加 x 并调用 continue 语句，该语句立即使 awk 开始执行下一个循环迭代，而不执行代码块的其余部分。如同 break 一样，continue 语句适合各种 awk 迭代循环。在 for 循环主体中使用时，continue 将使循环控制变量自动增加。以下是一个等价循环：

```
for ( x=1; x<=21; x++ ) {
    if ( x == 4 ) {
        continue
    }
    print "iteration",x
}
```

在 while 循环中时，在调用 continue 之前没有必要增加 x ，因为 for 循环会自动增加 x 。

数组

如果您知道 awk 可以使用数组，您一定会感到高兴。然而，在 awk 中，数组下标通常从 1 开始，而不是 0：

```
myarray[1]="jim"
myarray[2]=456
```

awk 遇到第一个赋值语句时，它将创建 `myarray` ，并将元素 `myarray[1]` 设置成 "jim"。执行了第二个赋值语句后，数组就有两个元素了。

数组迭代
定义之后，awk 有一个便利的机制来迭代数组元素，如下所示：

```
for ( x in myarray ) {
    print myarray[x]
}
```

这段代码将打印数组 `myarray` 中的每一个元素。当对于 for 使用这种特殊的 "in" 形式时，awk 将 `myarray` 的每个现有下标依次赋值给 `x` （循环控制变量），每次赋值以后都循环一次循环代码。虽然这是一个非常方便的 awk 功能，但它有一个缺点 -- 当 awk 在数组下标之间轮转时，它不会依照任何特定的顺序。那就意味着我们不能知道以上代码的输出是：

```
jim
456
```

还是

```
456
jim
```

套用 Forrest Gump 的话来说，迭代数组内容就像一盒巧克力 -- 您永远不知道将会得到什么。因此有必要使 awk 数组“字符串化”，我们现在就来研究这个问题。

数组下标字符串化

在我的 前一篇文章 中，我演示了 awk 实际上以字符串格式来存储数字值。虽然 awk 要执行必要的转换来完成这项工作，但它却可以使用某些看起来很奇怪的代码：

```
a="1"
b="2"
c=a+b+3
```

执行了这段代码后， `c` 等于 `6` 。由于 awk 是“字符串化”的，添加字符串 "1" 和 "2" 在功能上并不比添加数字 1 和 2 难。这两种情况下，awk 都可以成功执行运算。awk 的“字符串化”性质非常可爱 -- 您可能想要知道如果使用数组的字符串下标会发生什么情况。例如，使用以下代码：

```
myarr["1"]="Mr. Whipple"
print myarr["1"]
```

可以预料，这段代码将打印 "Mr. Whipple"。但如果去掉第二个 "1" 下标中的引号，情况又会怎样呢？

```
myarr["1"]="Mr. Whipple"
print myarr[1]
```

猜想这个代码片断的结果比较难。awk 将 `myarr["1"]` 和 `myarr[1]` 看作数组的两个独立元素，还是它们是指同一个元素？答案是它们指的是同一个元素，awk 将打印 "Mr. Whipple"，如同第一个代码片断一样。虽然看上去可能有点怪，但 awk 在幕后却一直使用数组的字符串下标！

了解了这个奇怪的真相之后，我们中的一些人可能想要执行类似于以下的古怪代码：

```
myarr["name"]="Mr. Whipple"
print myarr["name"]
```

这段代码不仅不会产生错误，而且它的功能与前面的示例完全相同，也将打印 "Mr. Whipple"！可以看到，awk 并没有限制我们使用纯整数下标；如果我们愿意，可以使用字符串下标，而且不会产生任何问题。只要我们使用非整数数组下标，如 `myarr["name"]`，那么我们就在使用 关联数组 。从技术上讲，如果我们使用字符串下标，awk 的后台操作并没有什么不同（因为即便使用“整数”下标，awk 还是会将它看作是字符串）。但是，应该将它们称作 关联数组 -- 它听起来很酷，而且会给您的上司留下印象。字符串化下标是我们的小秘密。;)

数组工具

谈到数组时，awk 给予我们许多灵活性。可以使用字符串下标，而且不需要连续的数字序列下标（例如，可以定义 `myarr[1]` 和 `myarr[1000]`，但不定义其它所有元素）。虽然这些都很有用，但在某些情况下，会产生混淆。幸好，awk 提供了一些实用功能有助于使数组变得更易于管理。

首先，可以删除数组元素。如果想要删除数组 `fooarray` 的元素 `1`，输入：

```
delete fooarray[1]
```

而且，如果想要查看是否存在某个特定数组元素，可以使用特殊的 "in" 布尔运算符，如下所示：

```
if ( 1 in fooarray ) {
    print "Ayep!  It's there."
} else {
    print "Nope!  Can't find it."
}
```

下一篇

本文中，我们已经讨论了许多基础知识。下一篇中，我将演示如何使用 awk 的数学运算和字符串函数，以及如何创建您自己的函数，使您完全掌握 awk 知识。我还将指导您创建支票簿结算程序。那时，我会鼓励您编写自己的 awk 程序。请查阅以下参考资料。

参考资料

- 您可以参阅本文在 developerWorks 全球站点上的 英文原文.
- 请阅读 *developerWorks*上的 awk 实例，第 1 部分。
- 如果想看好的老式书籍，O'Reilly 的 sed & awk, 2ndEdition是极佳选择。
- 请参考 comp.lang.awkFAQ 。它还包含许多附加 awk 链接。
- Patrick Hartigan 的 awk tutorial 还包括了实用的 awk 脚本。
- Thompson's TAWKCompiler 将 awk 脚本编译成快速二进制可执行文件。可用版本有 Windows 版、OS/2 版、DOS 版和 UNIX 版。
- The GNUAwk User's Guide可用于在线参考。

关于作者

Daniel Robbins 居住在新墨西哥州的 Albuquerque。他是 Gentoo Technologies, Inc. 的总裁兼 CEO， **Gentoo Linux**（用于 PC 的高级 Linux）和 **Portage** 系统（Linux 的下一代移植系统）的创始人。他还是 Macmillan 书籍 *Caldera OpenLinux Unleashed*、 *SuSE Linux Unleashed* 和 *Samba Unleashed* 的合作者。Daniel 自二年级起就与计算机某些领域结下不解之缘，那时他首先接触的是 Logo 程序语言，并沉溺于 Pac-Man 游戏中。这也许就是他至今仍担任 **SONY Electronic Publishing/Psygnosis** 的首席图形设计师的原因所在。Daniel 喜欢与妻子 Mary 和新出生的女儿 Hadassah 一起共度时光。可通过 drobbins@gentoo.org 与 Daniel 联系。

通用线程：**Awk** 实例，第 **3**部分

字符串函数和.....支票簿？

Daniel Robbins
总裁兼 CEO, Gentoo Technologies, Inc.
2001 年 4 月

在这篇 awk 系列的总结中，Daniel 向您介绍 awk 重要的字符串函数，以及演示了如何从头开始编写完整的支票簿结算程序。在这个过程中，您将学习如何编写自己的函数，并使用 awk 的多维数组。学完本文之后，您将掌握更多 awk 经验，可以让您创建功能更强大的脚本。

内容：

格式化输出

虽然大多数情况下 awk 的 print 语句可以完成任务，但有时我们还需要更多。在那些情况下，awk 提供了两个我们熟知的老朋友 printf() 和 sprintf()。是的，如同其它许多 awk 部件一样，这些函数等同于相应的 C 语言函数。printf() 会将格式化字符串打印到 stdout，而 sprintf() 则返回可以赋值给变量的格式化字符串。如果不熟悉 printf() 和 sprintf()，介绍 C 语言的文章可以让您迅速了解这两个基本打印函数。在 Linux 系统上，可以输入 "man 3 printf" 来查看 printf() 帮助页面。

以下是一些 awk sprintf() 和 printf() 的样本代码。可以看到，它们几乎与 C 语言完全相同。

```
x=1
b="foo"
printf("%s got a %d on the last test\n","Jim",83)
myout=("%s-%d",b,x)
print myout
```

此代码将打印：

```
Jim got a 83 on the last test
foo-1
```

字符串函数

awk 有许多字符串函数，这是件好事。在 awk 中，确实需要字符串函数，因为不能象在其它语言（如 C、C++ 和 Python）中那样将字符串看作是字符数组。例如，如果执行以下代码：

```
mystring="How are you doing today?"
print mystring[3]
```

将会接收到一个错误，如下所示：

```
awk: string.gawk:59: fatal: attempt to use scalar as array
```

噢，好吧。虽然不象 Python 的序列类型那样方便，但 awk 的字符串函数还是可以完成任务。让我们来看一下。

首先，有一个基本 length() 函数，它返回字符串的长度。以下是它的使用方法：

```
print length(mystring)
```

此代码将打印值：

```
24
```

好，继续。下一个字符串函数叫作 index，它将返回子字符串在另一个字符串中出现的位置，如果没有找到该字符串则返回 0。使用 mystring，可以按以下方法调用它：

```
print index(mystring,"you")
```

awk 会打印：

```
9
```

让我们继续讨论另外两个简单的函数，tolower() 和 toupper()。与您猜想的一样，这两个函数将返回字符串并且将所有字符分别转换成小写或大写。请注意，tolower() 和 toupper() 返回新的字符串，不会修改原来的字符串。这段代码：

```
print tolower(mystring)
print toupper(mystring)
print mystring
```

……将产生以下输出：

```
how are you doing today?
HOW ARE YOU DOING TODAY?
How are you doing today?
```

到现在为止一切不错，但我们究竟如何从字符串中选择子串，甚至单个字符？那就是使用 substr() 的原因。以下是 substr() 的调用方法：

```
mysub=substr(mystring,startpos,maxlen)
```

mystring 应该是要从中抽取子串的字符串变量或文字字符串。startpos 应该设置成起始字符位置，maxlen 应该包含要抽取的字符串的最大长度。请注意，我说的是 最大长度 ；如果 length(mystring) 比 startpos+maxlen 短，那么得到的结果就会被截断。substr() 不会修改原始字符串，而是返回子串。以下是一个示例：

```
print substr(mystring,9,3)
```

awk 将打印：

```
you
```

如果您通常用于编程的语言使用数组下标访问部分字符串（以及不使用这种语言的人），请记住 substr() 是 awk 代替方法。需要使用它来抽取单个字符和子串；因为 awk 是基于字符串的语言，所以会经常用到它。

现在，我们讨论一些更耐人寻味的函数，首先是 match()。match() 与 index() 非常相似，它与 index() 的区别在于它并不搜索子串，它搜索的是规则表达式。match() 函数将返回匹配的起始位置，如果没有找到匹配，则返回 0。此外，match() 还将设置两个变量，叫作 RSTART 和 RLENGTH。RSTART 包含返回值（第一个匹配的位置），RLENGTH 指定它占据的字符跨度（如果没有找到匹配，则返回 -1）。通过使用 RSTART、RLENGTH、substr() 和一个小循环，可以轻松地迭代字符串中的每个匹配。以下是一个 match() 调用示例：

```
print match(mystring,/you/), RSTART, RLENGTH
```

awk 将打印：

```
9 9 3
```

字符串替换

现在，我们将研究两个字符串替换函数，sub() 和 gsub()。这些函数与目前已经讨论过的函数略有不同，因为它们 确实修改原始字符串 。以下是一个模板，显示了如何调用 sub()：

```
sub(regexp,replstring,mystring)
```

调用 sub() 时，它将在 mystring 中匹配 regexp 的第一个字符序列，并且用 replstring 替换该序列。sub() 和 gsub() 用相同的自变量；唯一的区别是 sub() 将替换第一个 regexp 匹配（如果有的话），gsub() 将执行全局替换，换出字符串中的所有匹配。以下是一个 sub() 和 gsub() 调用示例：

```
sub(/o/,"O",mystring)
print mystring
mystring="How are you doing today?"
gsub(/o/,"O",mystring)
print mystring
```

必须将 mystring 复位成其初始值，因为第一个 sub() 调用直接修改了 mystring。在执行时，此代码将使 awk 输出：

```
HOw are you doing today?
HOw are yOu dOing tOday?
```

当然，也可以是更复杂的规则表达式。我把测试一些复杂规则表达式的任务留给您来完成。

通过介绍函数 split()，我们来汇总一下已讨论过的函数。split() 的任务是“切开”字符串，并将各部分放到使用整数下标的数组中。以下是一个 split() 调用示例：

```
numelements=split("Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec",mymonths,",")
```

调用 split() 时，第一个自变量包含要切开文字字符串或字符串变量。在第二个自变量中，应该指定 split() 将填入片段部分的数组名称。在第三个元素中，指定用于切开字符串的分隔符。split() 返回时，它将返回分割的字符串元素的数量。split() 将每一个片段赋值给下标从 1 开始的数组，因此以下代码：

```
print mymonths[1],mymonths[numelements]
```

……将打印：

```
Jan Dec
```

特殊字符串形式

简短注释 -- 调用 length()、sub() 或 gsub() 时，可以去掉最后一个自变量，这样 awk 将对 $0（整个当前行）应用函数调用。要打印文件中每一行的长度，使用以下 awk 脚本：

```
{
    print length()
}
```

财务上的趣事

几星期前，我决定用 awk 编写自己的支票簿结算程序。我决定使用简单的 tab 定界文本文件，以便于输入最近的存款和提款记录。其思路是将这个数据交给 awk 脚本，该脚本会自动合计所有金额，并告诉我余额。以下是我决定如何将所有交易记录到 "ASCII checkbook" 中：

```
23 Aug 2000 food    -   -   Y   Jimmy's Buffet      30.25
```

此文件中的每个字段都由一个或多个 tab 分隔。在日期（字段 1，$1）之后，有两个字段叫做“费用分类帐”和“收入分类帐”。以上面这行为例，输入费用时，我在费用字段中放入四个字母的别名，在收入字段中放入 "-"（空白项）。这表示这一特定项是“食品费用”。:) 以下是存款的示例：

```
23 Aug 2000 -   inco    -   Y   Boss Man        2001.00
```

在这个实例中，我在费用分类帐中放入 "-"（空白），在收入分类帐中放入 "inco"。"inco" 是一般（薪水之类）收入的别名。使用分类帐别名让我可以按类别生成收入和费用的明细分类帐。至于记录的其余部分，其它所有字段都是不需加以说明的。“是否付清？”字段（"Y" 或 "N"）记录了交易是否已过帐到我的帐户；除此之外，还有一个交易描述，和一个正的美元金额。

用于计算当前余额的算法不太难。awk 只需要依次读取每一行。如果列出了费用分类帐，但没有收入分类帐（为 "-"），那么这一项就是借方。如果列出了收入分类帐，但没有费用分类帐（为 "-"），那么这一项就是贷方。而且，如果同时列出了费用和收入分类帐，那么这个金额就是“分类帐转帐”；即，从费用分类帐减去美元金额，并将此金额添加到收入分类帐。此外，所有这些分类帐都是虚拟的，但对于跟踪收入和支出以及预算却非常有用。

代码

现在该研究代码了。我们将从第一行（BEGIN 块和函数定义）开始：

balance，第 **1** 部分

```
#!/usr/bin/env awk -f
BEGIN {
    FS="\t+"
    months="Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec"
}

function monthdigit(mymonth) {
    return (index(months,mymonth)+3)/4
}
```

首先执行 "chmod +x myscript" 命令，那么将第一行 "#!..." 添加到任何 awk 脚本将使它可以直接从 shell 中执行。其余行定义了 BEGIN 块，在 awk 开始处理支票簿文件之前将执行这个代码块。我们将 FS（字段分隔符）设置成 "\t+"，它会告诉 awk 字段由一个或多个 tab 分隔。另外，我们定义了字符串 months，下面将出现的 monthdigit() 函数将使用它。

最后三行显示了如何定义自己的 awk 。格式很简单 -- 输入 "function"，再输入名称，然后在括号中输入由逗号分隔的参数。在此之后，"{ }" 代码块包含了您希望这个函数执行的代码。所有函数都可以访问全局变量（如 months 变量）。另外，awk 提供了 "return" 语句，它允许函数返回一个值，并执行类似于 C 和其它语言中 "return" 的操作。这个特定函数将以 3 个字母字符串格式表示的月份名称转换成等价的数值。例如，以下代码：

```
print monthdigit("Mar")
```

……将打印：

```
3
```

现在，让我们讨论其它一些函数。

财务函数

以下是其它三个执行簿记的函数。我们即将见到的主代码块将调用这些函数之一，按顺序处理支票簿文件的每一行，从而将相应交易记录到 awk 数组中。有三种基本交易，贷方 (doincome)、借方 (doexpense) 和转帐 (dotransfer)。您会发现这三个函数全都接受一个自变量，叫作 mybalance。mybalance 是二维数组的一个占位符，我们将它作为自变量进行传递。目前，我们还没有处理过二维数组；但是，在下面可以看到，语法非常简单。只须用逗号分隔每一维就行了。

我们将按以下方式将信息记录到 "mybalance" 中。数组的第一维从 0 到 12，用于指定月份，0 代表全年。第二维是四个字母的分类帐，如 "food" 或 "inco"；这是我们处理的真实分类帐。因此，要查找全年食品分类帐的余额，应查看 mybalance [0,"food"]。要查找 6 月的收入，应查看 mybalance[6,"inco"]。

balance，第 **2** 部分

```
function doincome(mybalance) {
    mybalance[curmonth,$3] += amount
    mybalance[0,$3] += amount
}

function doexpense(mybalance) {
    mybalance[curmonth,$2] -= amount
    mybalance[0,$2] -= amount
}

function dotransfer(mybalance) {
    mybalance[0,$2] -= amount
    mybalance[curmonth,$2] -= amount
    mybalance[0,$3] += amount
    mybalance[curmonth,$3] += amount
}
```

调用 doincome() 或任何其它函数时，我们将交易记录到两个位置 -- mybalance[0,category] 和 mybalance[curmonth, category]，它们分别表示全年的分类帐余额和当月的分类帐余额。这让我们稍后可以轻松地生成年度或月度收入／支出明细分类帐。

如果研究这些函数，将发现在我的引用中传递了 mybalance 引用的数组。另外，我们还引用了几个全局变量：curmonth，它保存了当前记录所属的月份的数值，$2（费用分类帐），$3（收入分类帐）和金额（$7，美元金额）。调用 doincome() 和其它函数时，已经为要处理的当前记录（行）正确设置了所有这些变量。

主块

以下是主代码块，它包含了分析每一行输入数据的代码。请记住，由于正确设置了 FS，可以用 $ 1 引用第一个字段，用 $2 引用第二个字段，依次类推。调用 doincome() 和其它函数时，这些函数可以从函数内部访问 curmonth、$2、$3 和金额的当前值。请先研究代码，在代码之后可以见到我的说明。

balance，第 **3** 部分

```
{
    curmonth=monthdigit(substr($1,4,3))
    amount=$7

    #record all the categories encountered
    if ( $2 != "-" )
        globcat[$2]="yes"
    if ( $3 != "-" )
        globcat[$3]="yes"

    #tally up the transaction properly
    if ( $2 == "-" ) {
        if ( $3 == "-" ) {
            print "Error: inc and exp fields are both blank!"
            exit 1
        } else {
            #this is income
            doincome(balance)
            if ( $5 == "Y" )
                doincome(balance2)
        }
    } else if ( $3 == "-" ) {
        #this is an expense
        doexpense(balance)
        if ( $5 == "Y" )
            doexpense(balance2)
    } else {
        #this is a transfer
        dotransfer(balance)
        if ( $5 == "Y" )
            dotransfer(balance2)
    }
}
```

在主块中，前两行将 curmonth 设置成 1 到 12 之间的整数，并将金额设置成字段 7（使代码易于理解）。然后，是四行有趣的代码，它们将值写到数组 globcat 中。globcat，或称作全局分类帐数组，用于记录在文件中遇到的所有分类帐 -- "inco"、"misc"、"food"、"util" 等。例如，如果 $2 == "inco"，则将 globcat["inco"] 设置成 "yes"。稍后，我们可以使用简单的 "for (x in globcat)" 循环来迭代分类帐列表。

在接着的大约二十行中，我们分析字段 $2 和 $3，并适当记录交易。如果 $2=="-" 且 $3!="-"，表示我们有收入，因此调用 doincome()。如果是相反的情况，则调用 doexpense()；如果 $2 和 $3 都包含分类帐，则调用 dotransfer()。每次我们都将 "balance" 数组传递给这些函数，从而在这些函数中记录适当的数据。

您还会发现几行代码说“if ($5 == "Y")，那么将同一个交易记录到 *balance2* 中”。我们在这里究竟做了些什么？您将回忆起 $5 包含 "Y" 或 "N"，并记录交易是否已经过帐到帐户。由于仅当过帐了交易时我们才将交易记录到 balance2，因此 balance2 包含了真实的帐户余额，而 "balance" 包含了所有交易，不管是否已经过帐。可以使用 balance2 来验证数据项（因为它应该与当前银行帐户余额匹配），可以使用 "balance" 来确保没有透支帐户（因为它会考虑您开出的尚未兑现的所有支票）。

生成报表

主块重复处理了每一行记录之后，现在我们有了关于比较全面的、按分类帐和按月份划分的借方和贷方记录。现在，在这种情况下最合适的做法是只须定义生成报表的 END 块：

balance，第 **4** 部分

```
END {
    bal=0
    bal2=0
    for (x in globcat) {
        bal=bal+balance[0,x]
        bal2=bal2+balance2[0,x]
    }
    printf("Your available funds: %10.2f\n", bal)
    printf("Your account balance: %10.2f\n", bal2)
}
```

这个报表将打印出汇总，如下所示：

```
Your available funds:1174.22
Your account balance:2399.33
```

在 END 块中，我们使用 "for (x in globcat)" 结构来迭代每一个分类帐，根据记录在案的交易结算主要余额。实际上，我们结算两个余额，一个是可用资金，另一个是帐户余额。要执行程序并处理您在文件 "mycheckbook.txt" 中输入的财务数据，将以上所有代码放入文本文件 "balance"，执行 "chmod +x balance"，然后输入 "./balance mycheckbook.txt"。然后 balance 脚本将合计所有交易，打印出两行余额汇总。

升级

我使用这个程序的更高级版本来管理我的个人和企业财务。我的版本（由于篇幅限制不能在此涵盖）会打印出收入和费用的月度明细分类帐，包括年度总合、净收入和其它许多内容。它甚至以 HTML 格式输出数据，因此我可以在 Web 浏览器中查看它。:) 如果您认为这个程序有用，我建议您将这些特性添加到这个脚本中。不必将它配置成要 记录 任何附加信息；所需的全部信息已经在 balance 和 balance2 里面了。只要升级 END 块就万事具备了！

我希望您喜欢本系列。有关 awk 的详细信息，请参考以下列出的参考资料。

参考资料

- 您可以参阅本文在 developerWorks 全球站点上的 英文原文.
- 请阅读 Daniel 在 *developerWorks* 上发表的 awk 系列中的前几篇文章：awk 实例， 第 1 部分和 第 2 部分。
- 如果想看好的老式书籍，O'Reilly 的 sed & awk, 2ndEdition是极佳选择。
- 请参考 comp.lang.awkFAQ 。它还包含许多附加 awk 链接。
- Patrick Hartigan 的 awk tutorial 还包括了实用的 awk 脚本。
- Thompson's TAWKCompiler 将 awk 脚本编译成快速二进制可执行文件。可用版本有 Windows 版、OS/2 版、DOS 版和 UNIX 版。
- The GNUAwk User's Guide可用于在线参考。

关于作者

Daniel Robbins 居住在新墨西哥州的 Albuquerque。他是 Gentoo Technologies, Inc. 的总裁兼 CEO， **Gentoo Linux**（用于 PC 的高级 Linux）和 **Portage** 系统（Linux 的下一代移植系统）的创始人。他还是 Macmillan 书籍 *Caldera OpenLinux Unleashed*、 *SuSE Linux Unleashed* 和 *Samba Unleashed* 的合作者。Daniel 自二年级起就与计算机结下不解之缘，那时他首先接触的是 Logo 程序语言，并沉溺于 Pac-Man 游戏中。这也许就是他至今仍担任 **SONY Electronic Publishing/Psygnosis** 的首席图形设计师的原因所在。Daniel 喜欢与妻子 Mary 和新出生的女儿 Hadassah 一起共度时光。可通过 drobbins@gentoo.org 与 Daniel 联系。