Unix awk 使用手册

作者：莫名　　发表时间：2002/01/27 01:39pm

什么是 awk?

你可能对 UNIX 比较熟悉，但你可能对 awk 很陌生，这一点也不奇怪，的确，与其优秀的功能相比，awk 还远没达到它应有的知名度。awk 是什么？与其它大多数 UNIX 命令不同的是，从名字上看，我们不可能知道 awk 的功能：它既不是具有独立意义的英文单词，也不是几个相关单词的缩写。事实上，awk 是三个人名的缩写，他们是：Aho、(Peter)Weinberg 和(Brain)Kernighan。正是这三个人创造了 awk---一个优秀的样式扫描与处理工具。

AWK 的功能是什么？与 sed 和 grep 很相似，awk 是一种样式扫描与处理工具。但其功能却大大强于 sed 和 grep。awk 提供了极其强大的功能：它几乎可以完成 grep 和 sed 所能完成的全部工作，同时，它还可以进行样式装入、流控制、数学运算符、进程控制语句甚至于内置的变量和函数。它具备了一个完整的语言所应具有的几乎所有精美特性。实际上，awk 的确拥有自己的语言：awk 程序设计语言，awk 的三位创建者已将它正式定义为：样式扫描和处理语言。

为什么使用 awk?

即使如此，你也许仍然会问，我为什么要使用 awk?

使用 awk 的第一个理由是基于文本的样式扫描和处理是我们经常做的工作，awk 所做的工作有些象数据库，但与数据库不同的是，它处理的是文本文件，这些文件没有专门的存储格式，普通的人们就能编辑、阅读、理解和处理它们。而数据库文件往往具有特殊的存储格式，这使得它们必须用数据库处理程序来处理它们。既然这种类似于数据库的处理工作我们经常会遇到，我们就应当找到处理它们的简便易行的方法，UNIX 有很多这方面的工具，例如 sed 、grep、sort 以及 find 等等，awk 是其中十分优秀的一种。

使用 awk 的第二个理由是 awk 是一个简单的工具，当然这是相对于其强大的功能来说的。的确，UNIX 有许多优秀的工具，例如 UNIX 天然的开发工具 C 语言及其延续 C++就非常的优秀。但相对于它们来说，awk 完成同样的功能要方便和简捷得多。这首先是因为 awk 提供了适应多种需要的解决方案：从解决简单问题的 awk 命令行到复杂而精巧的 awk 程序设计语言，这样做的好处是，你可以不必用复杂的方法去解决本来很简单的问题。例如，你可以用一个命令行解决简单的问题，而 C 不行，即使一个再简单的程序，C 语言也必须经过编写、编译的全过程。其次，awk 本身是解释执行的，这就使得 awk 程序不必经过编译的过程，同时，这也使得它与 shell script 程序能够很好的契合。最后，awk 本身较 C 语言简单，虽然 awk 吸收了 C 语言很多优秀的成分，熟悉 C 语言会对学习 awk 有很大的帮助，但 awk 本身不须要会使用 C 语言——一种功能强大但需要大量时间学习才能掌握其技巧的开发工具。

使用 awk 的第三个理由是 awk 是一个容易获得的工具。与 C 和 C++语言不同，awk 只有一个文件(/bin/awk)，而且几乎每个版本的 UNIX 都提供各自版本的 awk，你完全不必费心去想如何获得 awk。但 C 语言却不是这样，虽然 C 语言是 UNIX 天然的开发工具，但这个开发工具却是单独发行的，换言之，你必须为你的 UNIX 版本的 C 语言开发工具单独付费（当然使用 D 版者除外），获得并安装它，然后你才可以使用它。

基于以上理由，再加上 awk 强大的功能，我们有理由说，如果你要处理与文本样式扫描相关的工
作，awk 应该是你的第一选择。在这里有一个可遵循的一般原则：如果你用普通的 shell

工具或 shell script 有困难的话，试试 awk,如果 awk 仍不能解决问题，则便用 C 语言，如果 C 语言仍然失败，则移至 C++。

awk 的调用方式

前面曾经说过，awk 提供了适应多种需要的不同解决方案，它们是：

一、awk 命令行，你可以象使用普通 UNIX 命令一样使用 awk，在命令行中你也可以使用 awk 程序设计语言，虽然 awk 支持多行的录入，但是录入长长的命令行并保证其正确无误却是一件令人头疼的事，因此，这种方法一般只用于解决简单的问题。当然，你也可以在 shell script 程序中引用 awk 命令行甚至 awk 程序脚本。

二、使用-f 选项调用 awk 程序。awk 允许将一段 awk 程序写入一个文本文件，然后在 awk 命令行中用-f 选项调用并执行这段程序。具体的方法我们将在后面的 awk 语法中讲到。

三、利用命令解释器调用 awk 程序：利用 UNIX 支持的命令解释器功能，我们可以将一段 awk 程序写入文本文件，然后在它的第一行加上:

#!/bin/awk -f

并赋予这个文本文件以执行的权限。这样做之后，你就可以在命令行中用类似于下面这样的方式调用并执行这段 awk 程序了。

$awk 脚本文本名 待处理文件

awk 的语法：

与其它 UNIX 命令一样，awk 拥有自己的语法：

awk [-F re] [parameter...] ['prog'] [-f progfile][in_file...]

参数说明：

-F re:允许 awk 更改其字段分隔符。

parameter: 该参数帮助为不同的变量赋值。

'prog': awk 的程序语句段。这个语句段必须用单拓号：'和'括起，以防被 shell 解释。这个程序语句段的标准形式为：

'pattern {action}'

其中 pattern 参数可以是 egrep 正则表达式中的任何一个，它可以使用语法/re/再加上一些样式匹配技巧构成。与 sed 类似，你也可以使用","分开两样式以选择某个范围。关于匹配的细节，你可以参考附录，如果仍不懂的话，找本 UNIX 书学学 grep 和 sed（本人是在学习 ed 时掌握匹配技术的）。action 参数总是被大括号包围，它由一系列 awk 语句组成，各语句之间用";"分隔。awk 解释它们，并在 pattern 给定的样式匹配的记录上执行其操作。与 shell 类似，你也可以使用“ # ”作为注释符，它使“ # ”到行尾的内容成为注释，在解释执行时，它们将被忽略。你可以省略 pattern 和 action 之一，但不能两者同时省略，当省略 pattern 时没有样式匹配，表示对所有行（记录）均执行操作，省略 action 时执行缺省的操作——在标准输出上显示。

-f progfile:允许 awk 调用并执行 progfile 指定有程序文件。progfile 是一个文本文件，他必须符合 awk 的语法。

in_file:awk 的输入文件，awk 允许对多个输入文件进行处理。值得注意的是 awk 不修改输入文件。如果未指定输入文件，awk 将接受标准输入，并将结果显示在标准输出上。

awk 支持输入输出重定向。

awk 的记录、字段与内置变量：

前面说过，awk 处理的工作与数据库的处理方式有相同之处，其相同处之一就是 awk 支持对记录和字段的处理，其中对字段的处理是 grep 和 sed 不能实现的，这也是 awk 优于二者的原因之一。在 awk 中，缺省的情况下总是将文本文件中的一行视为一个记录，而将一行中的某一部分作为记录中的一个字段。为了操作这些不同的字段，awk 借用 shell 的方法，用$1,$2,$3...这样的方式来顺序地表示行（记录）中的不同字段。特殊地，awk 用$0 表示整个行（记录）。不同的字段之间是用称作分隔符的字符分隔开的。系统默认的分隔符是空格。awk 允许在命令行中用-F re 的形式来改变这个分隔符。事实上，awk 用一个内置的变量 FS 来记忆这个分隔符。awk 中有好几个这样的内置变量，例如，记录分隔符变量 RS、当前工作的记录数 NR 等等，本文后面的附表列出了全部的内置变量。这些内置的变量可以在 awk 程序中引用或修改，例如，你可以利用 NR 变量在模式匹配中指定工作范围，也可以通过修改记录分隔符 RS 让一个特殊字符而不是换行符作为记录的分隔符。

例:显示文本文件 myfile 中第七行到第十五行中以字符%分隔的第一字段，第三字段和第七字段：

awk -F % 'NR==7,NR==15 {printf $1 $3 $7}'

awk 的内置函数

awk 之所以成为一种优秀的程序设计语言的原因之一是它吸收了某些优秀的程序设计语言（例如 C）语言的许多优点。这些优点之一就是内置函数的使用，awk 定义并支持了一系列的内置函数，由于这些函数的使用，使得 awk 提供的功能更为完善和强大，例如，awk 使用了一系列的字符串处理内置函数(这些函数看起来与 C 语言的字符串处理函数相似，其使用方式与 C 语言中的函数也相差无几)，正是由于这些内置函数的使用，使 awk 处理字符串的功能更加强大。本文后面的附录中列有一般的 awk 所提供的内置函数，这些内置函数也许与你的 awk 版本有些出入，因此，在使用之前，最好参考一下你的系统中的联机帮助。

作为内置函数的一个例子，我们将在这里介绍 awk 的 printf 函数，这个函数使得 awk 与 c 语言的输出相一致。实际上，awk 中有许多引用形式都是从 C 语言借用过来的。如果你熟悉 C 语言，你也许会记得其中的 printf 函数，它提供的强大格式输出功能曾经带我们许多的方便。幸运的是，我们在 awk 中又和它重逢了。awk 中 printf 几乎与 C 语言中一模一样，如果你熟悉 C 语言的话，你完全可以照 C 语言的模式使用 awk 中的 printf。因此在这里，我们只给出一个例子，如果你不熟悉的话，请随便找一本 C 语言的入门书翻翻。

例:显示文件 myfile 中的行号和第 3 字段：

$awk '{printf"%03d%s\n",NR,$1}' myfile

在命令行使用 awk

按照顺序，我们应当讲解 awk 程序设计的内容了，但在讲解之前，我们将用一些例子来对前面的知识进行回顾，这些例子都是在命令行中使用的，由此我们可以知道在命令行中使用 awk 是多么的方便。这样做的原因一方面是为下面的内容作铺垫，另一方面是介绍一些解决简单问题的方法，我们完全没有必要用复杂的方法来解决简单的问题----既然 awk 提供了较为简单的方法的话。

例：显示文本文件 mydoc 匹配（含有）字符串"sun"的所有行。

$awk '/sun/{print}' mydoc

由于显示整个记录（全行）是 awk 的缺省动作，因此可以省略 action 项。

$awk '/sun/' mydoc

例：下面是一个较为复杂的匹配的示例：

$awk '/[Ss]un/,/[Mm]oon/ {print}' myfile

它将显示第一个匹配 Sun 或 sun 的行与第一个匹配 Moon 或 moon 的行之间的行，并显示到标准输出上。

例：下面的示例显示了内置变量和内置函数 length（）的使用：

$awk 'length($0)>80 {print NR}' myfile

该命令行将显示文本 myfile 中所有超过 80 个字符的行号，在这里，用$0 表示整个记录（行），同时，内置变量 NR 不使用标志符'$'。

例：作为一个较为实际的例子，我们假设要对 UNIX 中的用户进行安全性检查，方法是考察/etc 下的 passwd 文件，检查其中的 passwd 字段（第二字段）是否为"*"，如不为"*"，则表示该用户没有设置密码，显示出这些用户名（第一字段）。我们可以用如下语句实现：

#awk -F: '$2=="" {printf("%s no password!\n",$1' /etc/passwd

在这个示例中，passwd 文件的字段分隔符是“ :”，因此，必须用-F：来更改默认的字段分隔符，这个示例中也涉及到了内置函数 printf 的使用。

awk 的变量

如同其它程序设计语言一样，awk 允许在程序语言中设置变量，事实上，提供变量的功能是程序设计语言的其本要求，不提供变量的程序设计语言本人还从未见过。

awk 提供两种变量，一种是 awk 内置的变量，这前面我们已经讲过，需要着重指出的是，与后面提到的其它变量不同的是，在 awk 程序中引用内置变量不需要使用标志符"$"（回忆一下前面讲过的 NR 的使用）。awk 提供的另一种变量是自定义变量。awk 允许用户在 awk 程序语句中定义并调用自已的变量。当然这种变量不能与内置变量及其它 awk 保留字相同，在 awk 中引用自定义变量必须在它前面加上标志符"$"。与 C 语言不同的是，awk 中不需要对变量进行初始化，awk 根据其在 awk 中第一次出现的形式和上下文确定其具体的数据类型。当变量类型不确定时，awk 默认其为字符串类型。这里有一个技巧：如果你要让你的 awk 程序知道你所使用的变量的明确类型，你应当在在程序中给它赋初值。在后面的实例中，我们将用到这一技巧。

运算与判断：

作为一种程序设计语言所应具有的特点之一，awk 支持多种运算，这些运算与 C 语言提供的基本相同：如+、-、*、/、%等等，同时，awk 也支持 C 语言中类似++、--、+=、-=、=+、=-之类的功能，这给熟悉 C 语言的使用者编写 awk 程序带来了极大的方便。作为对运算功能的一种扩展，awk 还提供了一系列内置的运算函数（如 log、sqr、cos、sin 等等）和一些用于对字符串进行操作（运算）的函数（如 length、substr 等等）。这些函数的引用大大的提高了 awk 的运算功能。

作为对条件转移指令的一部分，关系判断是每种程序设计语言都具备的功能，awk 也不例外。awk 中允许进行多种测试，如常用的==（等于）、!=（不等于）、>（大于）、<（小于）、>=（大于等于）、<=（小于等于）等等，同时，作为样式匹配，还提供了~（匹配

于）和！~（不匹配于）判断。

作为对测试的一种扩充，awk 也支持用逻辑运算符:!(非)、&&（与）、||（或）和括号（）进行多重判断，这大大增强了 awk 的功能。本文的附录中列出了 awk 所允许的运算、判断以及操作符的优先级。

awk 的流程控制

流程控制语句是任何程序设计语言都不能缺少的部分。任何好的语言都有一些执行流程控制的语句。awk 提供的完备的流程控制语句类似于 C 语言，这给我们编程带来了极大的方便。

1、BEGIN 和 END:

在 awk 中两个特别的表达式，BEGIN 和 END，这两者都可用于 pattern 中（参考前面的 awk 语法），提供 BEGIN 和 END 的作用是给程序赋予初始状态和在程序结束之后执行一些扫尾的工作。任何在 BEGIN 之后列出的操作（在{}内）将在 awk 开始扫描输入之前执行，而 END 之后列出的操作将在扫描完全部的输入之后执行。因此，通常使用 BEGIN 来显示变量和预置（初始化）变量，使用 END 来输出最终结果。

例：累计销售文件 xs 中的销售金额（假设销售金额在记录的第三字段）：

```
$awk
>'BEGIN { FS=":";print "统计销售金额";total=0}
>{print $3;total=total+$3;}
>END {printf "销售金额总计：%.2f",total}' sx
```

（注：>是 shell 提供的第二提示符，如要在 shell 程序 awk 语句和 awk 语言中换行，则需在行尾加反斜杠\） 在这里，BEGIN 预置了内部变量 FS（字段分隔符）和自定义变量 total,同时在扫描之前显示出输出行头。而 END 则在扫描完成后打印出总合计。

2、流程控制语句

awk 提供了完备的流程控制语句，其用法与 C 语言类似。下面我们一一加以说明：

2.1、if...else 语句:

格式：

```
if(表达式)
    语句 1
else
    语句 2
```

格式中"语句 1"可以是多个语句，如果你为了方便 awk 判断也方便你自已阅读，你最好将多个语句用{}括起来。awk 分枝结构允许嵌套，其格式为：

```
if(表达式 1) {if(表达式 2)
                    语句 1
                else
                   语句 2
                }
            语句 3
else {if(表达式 3)
            语句 4
            else
```

语句 5
}
语句 6

当然实际操作过程中你可能不会用到如此复杂的分枝结构，这里只是为了给出其样式罢了。

2.2、while 语句

格式为:
while(表达式)
语句

2.3、do-while 语句

格式为:

do
{ 语句 }while(条件判断语句）

2.4、for 语句

格式为：

for(初始表达式;终止条件;步长表达式) {语句}

在 awk 的 while、do-while 和 for 语句中允许使用 break,continue 语句来控制流程走向，也允许使用 exit 这样的语句来退出。break 中断当前正在执行的循环并跳到循环外执行下一条语句。continue 从当前位置跳到循环开始处执行。对于 exit 的执行有两种情况：当 exit 语句不在 END 中时，任何操作中的 exit 命令表现得如同到了文件尾，所有模式或操作执行将停止，END 模式中的操作被执行。而出现在 END 中的 exit 将导致程序终止。

例：

awk 中的自定义函数

定义和调用用户自己的函数是几乎每个高级语言都具有的功能，awk 也不例外，但原始的 awk 并不提供函数功能，只有在 nawk 或较新的 awk 版本中才可以增加函数。

函数的使用包含两部分 :函数的定义与函数调用。其中函数定义又包括要执行的代码（函数本身）和从主程序代码传递到该函数的临时调用。

awk 函数的定义方法如下：

function 函数名(参数表){ 函数体 }

在 gawk 中允许将 function 省略为 func，但其它版本的 awk 不允许。函数名必须是一个合法的标志符，参数表中可以不提供参数（但在调用函数时函数名后的一对括号仍然是不可缺少的），也可以提供一个或多个参数。与 C 语言相似，awk 的参数也是通过值来传递的。

在 awk 中调用函数比较简单，其方法与 C 语言相似，但 awk 比 C 语言更为灵活，它不执行参数有效性检查。换句话说，在你调用函数时，可以列出比函数预计（函数定义中规定）的多或少的参数，多余的参数会被 awk 所忽略，而不足的参数，awk 将它们置为缺省值 0 或空字符串，具体置为何值，将取决于参数的使用方式。

awk 函数有两种返回方式：隐式返回和显式返回。当 awk 执行到函数的结尾时，它自动

地返回到调用程序，这是函数是隐式返回的。如果需要在结束之前退出函数，可以明确地使用返回语句提前退出。方法是在函数中使用形如：return 返回值 格式的语句。

例：下面的例子演示了函数的使用。在这个示例中，定义了一个名为 print_header 的函数，该函数调用了两个参数 FileName 和 PageNum，FileName 参数传给函数当前使用的文件名，PageNum 参数是当前页的页号。这个函数的功能是打印（显示）出当前文件的文件名，和当前页的页号。完成这个功能后，这个函数将返回下一页的页号。

```
nawk
>'BEGIN{pageno=1;file=FILENAME
>pageno=print_header(file，pageno)；#调用函数 print_header
>printf("当前页页号是：%d\n",pageno);
>}
>#定义函数 print_header
>function print_header(FileName,PageNum){
>printf("%s %d\n",FileName,PageNum);
>PageNum++;return PageNUm;
>}
>}' myfile
```

执行这个程序将显示如下内容：

myfile 1

当前页页号是：2

awk 高级输入输出

1.读取下一条记录：

awk 的 next 语句导致 awk 读取下一个记录并完成模式匹配，然后立即执行相应的操作。通常它用匹配的模式执行操作中的代码。next 导致这个记录的任何额外匹配模式被忽略。

2.简单地读取一条记录

awk 的 getline 语句用于简单地读取一条记录。如果用户有一个数据记录类似两个物理记录，那么 getline 将尤其有用。它完成一般字段的分离(设置字段变量$0 FNR NF NR)。如果成功则返回 1，失败则返回 0（到达文件尾）。如果需简单地读取一个文件，则可以编写以下代码：

例：示例 getline 的使用

```
{while(getline==1)
{
#process the inputted fields
}
}
```

也可以使 getline 保存输入数据在一个字段中，而不是通过使用 getline variable 的形式处理一般字段。当使用这种方式时，NF 被置成 0，FNR 和 NR 被增值。

用户也可以使用 getline<"filename"方式从一个给定的文件中输入数据，而不是从命令行所列内容输入数据。此时，getline 将完成一般字段分离（设置字段变量$0 和 NF)。如果文件不存在，返回-1,成功，返回 1,返回 0 表示失败。用户可以从给定文件中读取数据到一个变量中,也可以用 stdin(标准输入设备）或一个包含这个文件名的变量代替 filename。

值得注意的是当使用这种方式时不修改 FNR 和 NR。

另一种使用 getline 语句的方法是从 UNIX 命令接受输入，例如下面的例子:

例：示例从 UNIX 命令接受输入

```
{while("who -u"|getline)
{
#process each line from the who command
}
}
```

当然，也可以使用如下形式:

"command" | getline variable

3.关闭文件:

awk 中允许在程序中关闭一个输入或输出文件，方法是使用 awk 的 close 语句。

close("filename")

filename 可以是 getline 打开的文件（也可以是 stdin,包含文件名的变量或者 getline 使用的确切命令）。或一个输出文件（可以是 stdout，包含文件名的变量或使用管道的确切命令）。

4.输出到一个文件:

awk 中允许用如下方式将结果输出到一个文件：

```
printf("hello word!\n")>"datafile"
或
printf("hello word!\n")>>"datafile"
```

5.输出到一个命令

awk 中允许用如下方式将结果输出到一个命令：

```
printf("hello word!\n")|"sort -t',"'
```

awk 与 shell script 混合编程

因为 awk 可以作为一个 shell 命令使用，因此 awk 能与 shell 批处理程序很好的融合在一起，这给实现 awk 与 shell 程序的混合编程提供了可能。实现混合编程的关键是 awk 与 shell script 之间的对话，换言之，就是 awk 与 shell script 之间的信息交流:awk 从 shell script 中获取所需的信息（通常是变量的值）、在 awk 中执行 shell 命令行、shell script 将命令执行的结果送给 awk 处理以及 shell script 读取 awk 的执行结果等等。

1.awk 读取 Shell script 程序变量

在 awk 中我们可以通过 “ '$变量名' ” 的方式读取 shell scrpit 程序中的变量。

例：在下面的示例中，我们将读取 shell scrpit 程序中的变量 Name，该变量存放的是文本 myfile 的撰写者，awk 将打印出这个人名。

```
$cat writename
:
# @(#)
#
.
```

.
.
```
Name="张三"nawk 'BEGIN {name="'Name"';\
      printf("\t%s\t 撰写者%s\n",FILENAME,name");}\
{...}END{...}' myfile
```
.
.
.

2.将 shell 命令的执行结果送给 awk 处理

作为信息传送的一种方法，我们可以将一条 shell 命令的结果通过管道线（|）传递给 awk 处理：

例：示例 awk 处理 shell 命令的执行结果
```
$who -u | awk '{printf("%s 正在执行%s\n",$2,$1)}'
```
该命令将打印出注册终端正在执行的程序名。

3.shell script 程序读 awk 的执行结果

为了实现 shell script 程序读取 awk 执行的结果，我们可以采取一些特殊的方法，例如我们可以用

变量名=`awk 语句`的形式将 awk 执行的结果存放入一个 shell script 变量。当然也可以用管道线的方法将 awk 执行结果传递给 shell script 程序处理。

例：作为传送消息的机制之一，UNIX 提供了一个向其所有用户传送消息的命令 wall（意思是 write to all 写给所有用户），该命令允许向所有工作中的用户（终端）发送消息。为此，我们可以通过一段 shell 批处理程序 wall.shell 来模拟这一程序（事实上比较老的版本中 wall 就是一段 shell 批处理程序：

```
$cat wall.shell
:
# @(#) wall.shell:发送消息给每个已注册终端
#
cat >/tmp/$$
#用户录入消息文本  who -u | awk '{print $2}' | while read tty
do
cat /tmp/$$>$tty
done
```

在这个程序里，awk 接受 who -u 命令的执行结果，该命令打印出所有已注册终端的信息，其中第二个字段是已注册终端的设备名，因此用 awk 命令析出该设备名，然后用 while read tty 语句循环读出这些文件名到变量（shell script 变量）tty 中，作为信息传送的终结地址。

4.在 awk 中执行 shell 命令行----嵌入函数 system()

system()是一个不适合字符或数字类型的嵌入函数，该函数的功能是处理作为参数传递给它的字符串。
system 对这个参数的处理就是将其作为命令处理，也就是说将其当作命令行一样加以执行。这使得用户在自己的 awk 程序需要时可以灵活地执行命令或脚本。

例：下面的程序将使用 system 嵌入函数打印用户编制好的报表文件，这个文件存放在名为 myreport.txt 的文件中。为简约起见，我们只列出了其 END 部分：

.
.
.
END {close("myreport.txt");system("lp myreport.txt");}

在这个示例中，我们首先使用 close 语句关闭了文件 myreport.txt 文件，然后使用 system 嵌入函数将 myreport.txt 送入打印机打印。

写到这里，我不得不跟朋友们说再见了，实在地说，这些内容仍然是 awk 的初步知识，电脑永远是前进的科学，awk 也不例外，本篇所能做的只是在你前行的漫漫长途中铺平一段小小开端，剩下的路还得靠你自己去走。老实说，如果本文真能给你前行的路上带来些许的方便，那本人就知足了！

如对本篇有任何疑问，请 E-mail To:Chizlong@yeah.net 或到主页 http://chizling.yeah.net 中留言。

附录：

1.awk 的常规表达式元字符

\ 换码序列
^ 在字符串的开头开始匹配
$ 在字符串的结尾开始匹配
. 与任何单个字符串匹配
[**ABC**] 与[]内的任一字符匹配
[**A-Ca-c**] 与 A-C 及 a-c 范围内的字符匹配（按字母表顺序）
[^**ABC**] 与除[]内的所有字符以外的任一字符匹配
Desk|Chair 与 Desk 和 Chair 中的任一个匹配
[**ABC**][**DEF**] 关联。与 A、B、C 中的任一字符匹配，且其后要跟 D、E、F 中的任一个字符。
[**ABC**]***** 与 A、B 或 C 中任一个出现 0 次或多次的字符相匹配
[**ABC**]+ 与 A、B 或 C 中任何一个出现 1 次或多次的字符相匹配
？ 与一个空串或 A、B 或 C 在任何一个字符相匹配
（**Blue|Black**）berry 合并常规表达式，与 Blueberry 或 Blackberry 相匹配

2.awk 算术运算符

运算符	用途
x^y	x 的 y 次幂
x**y	同上
x%y	计算 x/y 的余数（求模）
x+y	x 加 y
x-y	x 减 y
x*y	x 乘 y
x/y	x 除 y
-y	负 y(y 的开关符号);也称一目减
++y	y 加 1 后使用 y(前置加)
y++	使用 y 值后加 1（后缀加）
--y	y 减 1 后使用 y(前置减)
y--	使用后 y 减 1(后缀减)
x=y	将 y 的值赋给 x

x+=y 将 x+y 的值赋给 x
x-=y 将 x-y 的值赋给 x
x*=y 将 x*y 的值赋给 x
x/=y 将 x/y 的值赋给 x x%=y 将 x%y 的值赋给 x
x^=y 将 x^y 的值赋给 x
x**=y 将 x**y 的值赋给 x

3.awk 允许的测试：

操作符	含义
x==y	x 等于 y
x!=y	x 不等于 y
x>y	x 大于 y
x>=y	x 大于或等于 y
x<y	x 小于 y
x<=y	x 小于或等于 y?
x~re	x 匹配正则表达式 re?
x!~re	x 不匹配正则表达式 re?

4.awk 的操作符(按优先级升序排列)

= 、+=、 -=、 *= 、/= 、 %=
||
&&
> >= < <= == != ~ !~
xy (字符串连结，'x"y'变成"xy"）
+ -
* / %
++ --

5.awk 内置变量（预定义变量）

说明：表中 v 项表示第一个支持变量的工具（下同）：A=awk，N=nawk,P=POSIX awk,G=gawk

V	变量	含义	缺省值
N	ARGC	命令行参数个数	
G	ARGIND	当前被处理文件的 ARGV 标志符	
N	ARGV	命令行参数数组	
G	CONVFMT	数字转换格式	%.6g
P	ENVIRON UNIX	环境变量	
N	ERRNO UNIX	系统错误消息	
G	FIELDWIDTHS	输入字段宽度的空白分隔字符串	
A	FILENAME	当前输入文件的名字	
P	FNR	当前记录数	
A	FS	输入字段分隔符	空格
G	IGNORECASE	控制大小写敏感	0（大小写敏感）
A	NF	当前记录中的字段个数	
A	NR	已经读出的记录数	
A	OFMT	数字的输出格式	%.6g
A	OFS	输出字段分隔符	空格

A	ORS	输出的记录分隔符 新行
A	RS	输入的记录他隔符 新行
N	RSTART	被匹配函数匹配的字符串首
N	RLENGTH	被匹配函数匹配的字符串长度
N	SUBSEP	下标分隔符 "\034"

6.awk 的内置函数

V	函数	用途或返回值
N	gsub(reg,string,target)	每次常规表达式 reg 匹配时替换 target 中的 string
N	index(search,string)	返回 string 中 search 串的位置
A	length(string)	求串 string 中的字符个数
N	match(string,reg)	返回常规表达式 reg 匹配的 string 中的位置
N	printf(format,variable)	格式化输出，按 format 提供的格式输出变量 variable。
N	split(string,store,delim)	根据分界符 delim,分解 string 为 store 的数组元素
N	sprintf(format,variable)	返回一个包含基于 format 的格式化数据，variables 是要放到串中的数据
G	strftime(format,timestamp)	返回一个基于 format 的日期或者时间串，timestmp 是 systime()函数返回的时间
N	sub(reg,string,target)	第一次当常规表达式 reg 匹配，替换 target 串中的字符串
A	substr(string,position,len)	返回一个以 position 开始 len 个字符的子串
P	tolower(string)	返回 string 中对应的小写字符
P	toupper(string)	返回 string 中对应的大写字符
A	atan(x,y)	x 的余切(弧度)
N	cos(x)	x 的余弦(弧度)
A	exp(x)	e 的 x 幂
A	int(x)	x 的整数部分
A	log(x)	x 的自然对数值
N	rand()	0-1 之间的随机数
N	sin(x)	x 的正弦(弧度)
A	sqrt(x)	x 的平方根
A	srand(x)	初始化随机数发生器。如果忽略 x，则使用 system()
G	system()	返回自 1970 年 1 月 1 日以来经过的时间(按秒计算)

****

通用线程：Awk 实例

作者：donggua1　　发表时间：2002/08/29 08:24am

通用线程：Awk 实例

Daniel Robbins (drobbins@gentoo.org)

第一部分

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，Daniel Robbins 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

halt7operator11root0shutdown6sync5bin1....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:haltuid:7username:operatorid:11username:rootuid:0username: shutdown
uid:6username: sync                uid:5username: 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。

第二部分

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 Weasel100 Pleasant DriveSan Francisco, CA 12345Big Tony200 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 12345Big 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 VinnieVinnie's Auto Shop300 City AlleySosueme, 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 67890Cousin 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 1iteration 2iteration 3iteration 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..."}
```

因为 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

```
jim456
还是
456jim
```

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."}
```

第三部分

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 testfoo-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 mystringmystring="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 thecategories 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 { doincome(balance) if ( $5 == "Y" )    doincome(balance2) }
  } else if ( $3 == "-" ) {doexpense(balance)
          if ( $5 == "Y" ) doexpense(balance2) }
          else {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:1174.22Your 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

·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。

Awk 编程学习笔记之一（原创）

作者：donggua1　　发表时间：2002/08/23 05:56pm

Awk 编程实例分析(学习笔记之一)

文件处理

1．有文件 gz.txt(工资)

4367422926350133100 张三 1250.00
4367422926351220178 李四 1300.00
4367422926351220546 王二 0
苏五丙 1340.00
4367422926351220178 孙六月 1390.00

…… ……

要求：按账号 19 位、姓名 8 位、工资 8 位来排列，且如姓名不足 8 位在之后补足，工资不足 8 位则在工资
之前补。同时要求去掉工资为 0 的名单，没有账号在前补 19 位空格，并输出工资总数加以核对，处理后
应如下排列：

4367422926350133100 张三　　1250.00
4367422926351220178 李四　　1300.00
　　苏五丙　1340.00
4367422926351220178 孙六月　1390.00

…… ……

awk 程序:

```
#-----------------------------------
#shgz1.sh
sblank=" "
awk '$NF!="0"{print $0}' $1 > tmp.txt                    #删除工资为 0 的人数
awk '{
if($1!~/[0-9]/){
   printf("%-19.19s%-8.8s%8.2f\n","""$kk""",$1,$2)}    #如果没有账号补上空格
else{ printf("%-19.19s%-8.8s%8.2f\n",$1,$2,$3)}}' tmp.txt > $2
awk '$NF~/[0-9]/{ sum=sum+$NF}
END{system("rm tmp.txt")printf("The sum is%16.2f!\n",sum) #输出工资总数
}' $2
```

本人刚刚开始学 awk，也会把自己的一些心得与体会发上来其中尚有许多不成熟的地方，愿得各位指正
！

Awk 编程学习笔记之二（原创）

作者：donggua1　　发表时间：2002/08/26 02:43pm

2．

有工资文件 gz2.txt 如下:

```
姓名      账号                    金额
---------------------------------
张三      4367422926350133100 1250
李四      4367422926351220178 1300
王二      43674229263512205460
苏五丙                          1340
孙六月    43674229263512201783 90
```

…… ……

这个工资文件相比以上要特殊一些，首先人名在前，而且账号与金额联在一起，19 位账号就是金额。

要求：按账号 19 位、姓名 8 位、工资 8 位来排列，且如姓名不足 8 位在之后补足，工资不足 8 位则在工资之前补。同时要求去掉前面两行及工资为 0 的名单，没有账号在前补 19 位空格，并输出工资总数加以核对，处理后应如下排列：

```
4367422926350133100 张三      1250.00
4367422926351220178 李四      1300.00
                    苏五丙    1340.00
4367422926351220178 孙六月     390.00
…… ……
```

awk 程序:

```
#shgz2.sh
cut -c1-8 $1>tmp1.txt          #用 cut 命令分别提出三个字段的内容。
cut -c9-27 $1>tmp2.txt
cut -c28-60 $1>tmp3.txt
paste -d, tmp2.txt tmp1.txt tmp3.txt|tr -d " ">tmp4.txt    #三个文件合一，并用 ”, ” 为分隔符
awk -F, '{                                  #-F, 表示分隔符为 ”, ”
if ($1~/^2/) printf("%-19.19s%-8.8s%8.2f\n",$1,$2,$3)
else printf("%-19.19s%-8.8s%8.2f\n",""$kk"",$2,$3)}'tmp4.txt > tmp5.txt
sed '/ 0.00$/d' tmp5.txt>$2      #去掉金额为 0 的行
awk '$NF~/[0-9]/{ sum=sum+$NF }
END{system("rm tmp*.txt")                                    #删除临时生成的文件
printf("The sum is%16.2f!\n", sum)                          #输出工资总数
}' $2
```

****

附注:

本例中结合了 cut,sed 与 awk 的用法!相关命令及参数可参考相关书籍如 cut -c1-8 表示提出每一行的 1 到 8
位字符.

与 awk 一样,cut 也可以按分隔符来分离字段,而且缺省的分隔符为空格,当然也可修改.cut -f1 gz2.txt 就会取出姓名这个字段.但是我们可以看到,账号与金额是分不出来的,所以也是我们用 cut -c1-8 的原因.