Linux Awk用法总结

2017-12-09 Xiaoh's Blog 马哥Linux运维

目录

> f. split
> g. sprintf
> h. substr
> i. tolower
> j. toupper
> 13. I/O处理函数
> a. getline
> b. close
> c. system

> *作为另一篇关于 Awk的文章 的姐妹篇，这篇文章也是简述了Awk的使用方法。更多的内容实则是从以前的 博客 摘抄过来的。记载在这里，方便学习。*

Awk是什么

Awk、sed与grep，俗称Linux下的三剑客，它们之前有很多相似点，但是同样也各有各的特色，相似的地方是它们都可以匹配文本，其中sed和awk还可以用于文本编辑，而grep则不具备这个功用。sed是一种非交互式且面向字符流的编辑器（a “non-interactive” stream-oriented editor），而awk则是一门模式匹配的编程语言，因为它的主要功能是用于匹配文本并处理，同时它有一些编程语言才有的语法，例如函数、分支循环语句、变量等等，当然比起我们常见的编程语言，Awk相对比较简单。

使用Awk，我们可以做以下事情：

- 将文本文件视为由字段和记录组成的文本数据库；
- 在操作文本数据库的过程中能够使用变量；
- 能够使用数学运算和字符串操作；
- 能够使用常见的编程结构，例如条件分支与循环；
- 能够格式化输出；
- 能够自定义函数；
- 能够在awk脚本中执行UNIX命令；
- 能够处理UNIX命令的输出结果；

装备以上功能，awk能够做得事情非常多。但千里之行，始于足下，我们首先从最基本的命令行语法开始，一步一步得走入awk的编程世界。

命令行语法

同sed一样，awk的命令行语法也有两种形式：

```
awk [-F ERE] [-v assignment] ... program [argument ...]
awk [-F ERE] -f progfile ...  [-v assignment] ...[argument ...]
```

这里的program类似sed中的script，因为我们一直强调awk是一门编程语言，所以将awk的脚本视为一段代码。而awk的脚本同样可以写到一个文件中，并通过-f参数指定，这一点和sed是一样的。program一般多个pattern和action序列组成，当读入的记录匹配pattern时，才会执行相应的action命令。这里有一点要注意，在第一种形式中，除去命令行选项外，program参数一定要位于第一个位置。

Awk的输入被解析成多个记录（Record），默认情况下，记录的分隔符是，因此可以认为一行就是一个记录，记录的分隔符可以通过内置变量RS更改。当记录匹配某个pattern时，才会执行后续的action命令。

而每个记录由进一步地被分隔成多个字段（Field），默认情况下字段的分隔符是空白符，例如空格、制表符等等，也可以通过-F ERE选项或者内置变量FS更改。在awk中，可以通过$1，$2...来访问对应位置的字段，同时$0存放整个记录，这一点有点类似shell下的命令行位置参数。关于这些内容，我们会在下面详细介绍，这里你只要知道有这些东西就好。

标准的awk命令行参数主要由以下三个：

- -F ERE：定义字段分隔符，该选项的值可以是扩展的正则表达式（ERE）；
- -f progfile：指定awk脚本，可以同时指定多个脚本，它们会按照在命令行中出现的顺序连接在一起；
- -v assignment：定义awk变量，形式同awk中的变量赋值，即name=value，赋值发生在awk处理文本之前；

为了便于理解，这里举几个简单的例子。通过-F参数设置冒号:为分隔符，并打印各个字段：

```
echo "1:2:3" | awk -F: '{print $1 " and " $2 " and " $3}'
1 and 2 and 3
```

在awk的脚本中访问通过-v选项设置的变量：

```
echo | awk -v a=1 'BEGIN {print a}'
1
```

从上面可以看到，通过-v选项设置的变量在BEGIN的位置就可以访问了。BEGIN是一个特殊的pattern，它在awk处理输入之前就会执行，可以认为是一个初始化语句，与此对应的还有END。

好像还没介绍如何指定处理的文件，是不是最后的argument就是指定的文件？在看我这本书之前，我也是这样认为的，但是实际上arguemnt有两种形式，它们分别是输入文件（file）和变量赋值（assignment）。

awk可以同时指定多个输入文件，如果输入文件的文件名为'-'，表示从标准输入读取内容。

变量赋值类似-v选项，它的形式为name=value。awk中的变量名同一般的编程语言无太多区别，但是不能同awk的保留关键字重名，可以查看awk的man手册查询哪些是保留关键字。而变量值只有两种形式：字符串和数值。变量赋值必须位于脚本参数的后面，与文件名参数无先后顺序的要求，但是位于不同位置的赋值它的执行时机是不同的。

我们用实际的例子来解释这个区别，假设有两个文件：a和b，它们的内容分别如下所示：

```
cat a
file a
cat b
file b
```

为了说明赋值操作发生的时机，我们在BEGIN，正常处理，END三个地方都打印变量的值。

第一种情况： 变量赋值位于所有文件名参数之前

```
awk 'BEGIN {print "BEGIN: " var} {print "PROCESS: " var} END {print "END: " var }' var=1 a
BEGIN:
PROCESS: 1
END: 1
```

结果：赋值操作发生在正常处理之前，BEGIN动作之后。

第二种情况：变量赋值位于所有文件名之后：

```
awk 'BEGIN {print "BEGIN: " var} {print "PROCESS: " var} END {print "END: " var }' a var=1
BEGIN:
PROCESS:
END: 1
```

结果：赋值操作发生在正常处理之后，END动作之前。

第三种情况：变量赋值位于文件名之间：

```
awk 'BEGIN {print "BEGIN: " var} {print "PROCESS: " var} END {print "END: " var }' a var=1 b
BEGIN:
PROCESS:
PROCESS: 1
END: 1
```

结果：赋值操作发生在处理前面的文件之后，并且位于处理后面的文件之前；

总结如下：

- 如果变量赋值在第一个文件参数之前，在BEGIN动作之后执行，影响到正常处理和END动作；
- 如果变量赋值在最后一个文件参数之后，在END动作之前执行，仅影响END动作；
- 如果文件参数不存在，情况同1所述；
- 如果变量赋值位于多个文件参数之间，在变量赋值前面的文件被处理后执行，影响到后续文件的处理和END动作；

所以变量赋值一定要考虑清楚用途，否则比较容易出错，不过一般情况下也不会用到变量赋值。

自然地大家会将变量赋值与-v assignment选项进行比较，赋值的形式是一致的，但是-v选项的执行时机比变量赋值要早：

```
echo 1 | awk -v var=a 'BEGIN {print "BEGIN: " var}'
BEGIN: a
```

可见，-v选项的赋值操作在BEGIN动作之前就执行了。

变量赋值一定要小心不要与保留关键字重名，否则会报错：

```
echo 1 | awk -v BEGIN=1 'BEGIN {print "BEGIN: " BEGIN}'
awk: fatal: cannot use gawk builtin `BEGIN' as variable name
```

记录（Record）与字段（Field)

对于数据库来说，一个数据库表是由多条记录组成的，每一行表示一条记录（Record）。每条记录由多列组成，每一列表示一个字段（Field)。Awk将一个文本文件视为一个文本数据库，因此它也有记录和字段的概念。默认情况下，记录的分隔符是回车，字段的分隔符是空白符，所以文本文件的每一行表示一个记录，而每一行中的内容被空白分隔成多个字段。利用字段和记录，awk就可以非常灵活地处理文件的内容。

可以通过-F选项来修改默认的字段分隔符，例如/etc/passwd的每一行都是由冒号分隔成多个字段的，所以这里就需要将分隔符设置成冒号：

```
awk -F: '{print $1}' /etc/passwd | head -3
root
bin
daemon
```

这里通过$1引用第一人字段，类似地$2表示第二个字段，$3表示第三个字段…. $0则表示整个记录。内置变量NF记录着字段的个数，所以$NF表示最后一个字段：

```
awk -F: '{print $NF}' /etc/passwd | head -3
/bin/bash
/bin/false
/bin/false
```

当然，$(NF-1)表示倒数第二个。

内置变量FS也可以用于更改字段分隔符，它记录着当前的字段分隔符：

```
awk -F: '{print FS}' /etc/passwd | head -1
:
awk -v FS=: '{print $1}' /etc/passwd | head -1
root
```

记录的分隔符可以通过内置变量RS更改：

```
awk -v RS=: '{print $0}' /etc/passwd | head -1
root
```

如果将RS设置成空，行为有就一点怪异了，它会将连续不为空行的所有行（一个段落）当作一个记录，而且强制回车为字段分隔符：

```
cat awk_man.txt
The awk utility shall execute programs written in the awk programming language,
which is specialized for textual data manipulation. An awk program is a sequence
of patterns and corresponding actions.  When  input  is  read  that matches a
pattern, the action associated with that pattern is carried out.

Input shall be interpreted as a sequence of records. By default, a record is a line,
less its terminating <newline>, but this can be changed by using the RS built-in
variable. Each record of input shall be matched in turn against each pattern in the
program. For each pattern matched, the associated action shall be executed.

awk 'BEGIN {RS="";FS=":"} {print "First line: " $1}' awk_man.txt
First line: The awk utility shall execute programs written in the awk programming language,
First line: Input shall be interpreted as a sequence of records. By default, a record is a line
```

这里，我们将变量赋值放到BEGIN动作中执行，因为BEGIN动作是在文件处理之前执行的，专门用于放初始化的语句。FS的赋值在这里是无效的，awk依然使用回车符来分隔字段。

脚本（Script）组成

命令行中的program部分，可以称为awk代码,也可以称为awk脚本。一段awk脚本是由多个’pattern { action }‘序列组成的。action是一个或者多个语句，它在输入行匹配pattern的时候被执行。如果pattern为空，表明这个action会在每一行处理时都会被执行。下面的例子简单地打印文件的每一行，这里不带任何参数的print语句打印的是整个记录，类似’print $0‘：

```
echo -e 'line1\nline2' | awk '{print}'
line1
line2
```

除了pattern { action }，还可以在脚本中定义自定义的函数，函数定义格式如下所示：

```
function name(parameter list) { statements }
```

函数的参数列表用逗号分隔，参数默认是局部变量，无法在函数之外访问，而在函数中定义的变量为全局变量，可以在函数之外访问，如：

```
echo line1 | awk '
function t(a) {
    b=a;
    print a;
}
{
    print b;
    t("kodango.me");
    print b;
}'
kodango.me
kodango.me
```

Awk脚本中的语句使用空行或者分号分隔，使用分号可以放在同一行，不过有时候会影响可读性，尤其是分支或循环结构中，很容易出错。

如果Awk中的一个语句太长，要分成多行，可以在行为使用反斜杠''：

```
cat test.awk
function t(a)
{
        b=a
        print "This is a very long line, so use backslash to escape the newline then we will print the
                variable a: a=" a
}
{ print b; t("kodango.me"); print b;}
echo 1 | awk -f test.awk

This is a very long line, so use backslash to escape the newline then we will print the variable a:
a=kodango.me
kodango.me
```

这里我们将脚本写到文件中，并通过-f参数来指定。但是，在一些特殊符号之后，是可以直接换行的，例如”, { &&		”。

模式（Pattern）

模式是awk中比较重要的一部分，它有以下几种情况：

- /regular expression/： 扩展的正则表达式（Extended Regular Expression）， 关于ERE可以参考 这篇文章；
- relational expression： 关系表达式，例如大于、小于、等于，关系表达式结果为true表示匹配；
- BEGIN： 特殊的模式，在第一个记录处理之前被执行，常用于初始化语句的执行；
- END： 特殊的模式，在最后一个记录处理之前被执行，常用于输出汇总信息；
- pattern, pattern：模式对，匹配两者之间的所有记录，类似sed的地址对；

例如查找匹配数字3的行：

```
seq 1 20 | awk '/3/ {print}'
3
13
```

相反地，可以在在正则表达式之前加上’!’表示不匹配：

```
seq 1 5 | awk '!/3/ {print}'
1
2
4
5
```

除了BEGIN和END这两个特殊的模式外，其余的模式都可以使用’&&’或者’		’运算符组合，前者表示逻辑与，后者表示逻辑或：

```
seq 1 50 | awk '/3/ && /1/ {print}'
13
31
```

前面的正则都是整行匹配，有时候仅仅需要匹配某个字符，这样我们可以用表达式$n ~ /ere/：

```
awk '$1 ~ /xi/ {print}' /etc/passwd
xingming:x:1000:1000::/home/xingming:/bin/bash
```

有时候我们只想显示特定和行，例如显示第一行：

```
seq 1 5 | awk 'NR==1 {print}'
1
```

正则表达式（Regular Expression）

正则表达式的内容介绍起来太麻烦，还是推荐同学阅读现有的文章（如 Linux/Unix工具与正则表达式的POSIX规范），里面对各个流派的正则表达式归纳地很清楚了。

表达式（Expressions）

表达式可以由常量、变量、运算符和函数组成，常数和变量的值可以为字符串和数值。

Awk中的变量有三种类型：用户定义的变量，内置变量和字段变量。其中，内置变量名都是大写的。变量并不非一定要被声明或者被初始化，未初始化的字符串变量的值为""，未初始化的数值变量的值为0。字段变量可以用$n来引用，n的取值范围为[0,NF]。n可以为一个变量，例如$NF代码最后一个字段，而$(NF-1)表示倒数第二个字段。

数组

数组是一种特殊的变量，在awk中，比较特殊地是，数组的下标可以为数字或者字符串。数组的赋值很简单，下面将value赋值给数组下标为index的元素：`array[index]=value`

可以用for..in..语法遍历数组元素，其中item是数组元素对应的下标：`for (item in array)`

当然也可以在if分支判断中使用in操作符：`if (item in array)`

一个完整的例子如下所示：

```
echo "1 2 3" | awk '{
    for (i=0;i<NF;i++)
          a[i]=i;
}

END {
    print 3 in a
        for (i in a)
              printf "%s: %s\n", i, a[i];
}'
0
0: 0
1: 1
2: 2
```

内置变量

Awk在内部维护了许多内置变量，或者称为系统变量，例如之前提到的FS、RS等等。常见的内置变量如下表所示

变量名	描述
ARGC	命令行参数的各个，即ARGV数组的长度
ARGV	存放命令行参数
CONVFMT	定义awk内部数值转换成字符串的格式，默认值为”%.6g”
OFMT	定义输出时数值转换成字符串的格式，默认值为”%.6g”
ENVIRON	存放系统环境变量的关联数组
FILENAME	当前被处理的文件名
NR	记录的总个数
FNR	当前文件中的记录的总个数
FS	字段分隔符，默认为空白
NF	每个记录中字段的个数

变量名	描述
RS	记录的分隔符，默认为回车
OFS	输出时字段的分隔符，默认为空白
ORS	输出时记录的分隔符，默认为回车
RLENGTH	被match函数匹配的子串长度
RSTART	被match函数匹配的子串位于目标字符串的起始下标

下面主要介绍几个比较难理解的内置变量：

ARGV与ARGC

ARGV与ARGC的意思比较好理解，就像C语言 `main(int argc, char **argv)`。ARGV数组的下标从0开始到ARGC-1，它存放的是命令行参数，并且排除命令行选项（例如-v/-f）以及program部分。因此事实上ARGV只是存储argument的部分，即文件名（file）以及命令行变量赋值两部分的内容。

通过下面的例子可以大概了解ARGC与ARGV的用法：

```
awk 'BEGIN {
    for (i = 0; i < ARGC; i++)
        print ARGV[i]
}' inventory-shipped BBS-list
awk
inventory-shipped
BBS-list
```

ARGV的用法不仅限于此，它是可以修改的，可以更改数组元素的值，可以增加数组元素或者删除数组元素。

更改ARGV元素的值

假设我们有a, b两个文件，它们各有一行内容：file a和file b。现在利用ARGV，我们可以做到偷梁换柱：

```
awk 'BEGIN{ARGV[1]="b"} {print}' a
file b
```

这里要注意ARGV[1]=”b”的引号不能缺少，否则ARGV[1]=b会将变量b的值赋值给ARGV[1]。

当awk处理完一个文件之后，它会从ARGV的下一个元素获取参数，如果是一个文件则继续处理，如果是一个变量赋值则执行赋值操作：

```
awk 'BEGIN{ARGV[1]="var=1"} {print var}' a b
1
```

当下一个元素为空时，则跳过不处理，这样可以避开处理某个文件：

```
awk 'BEGIN{ARGV[1]=""} {print}' a b
file b
```

上面的例子中a这个文件就被跳过了。

而当下一个元素的值为”-”时，表明从标准输入读取内容：

```
awk 'BEGIN{ARGV[1]="-"} {print}' a b
a
a    # --> 这里按下CTRL+D停止输入
file b
```

删除ARGV元素

删除ARGV元素和将元素的值赋值为空的效果是一样的，它们都会跳转对某个参数的处理：

```
awk 'BEGIN{delete ARGV[1]} {print}' a b
file b
```

删除数组元素可以用delete语句。

增加ARGV元素

我第一次看到ARGV变量的时候就在想，能不能利用ARGV变量避免提供命令行参数，就像这样: `awk 'BEGIN{ARGV[1]="a";} {print}'`

但是事实上这样不行，awk会依然从标准输入中获取内容。下面的方法倒是可以，首先增加ARGC的值，再增加ARGV元素，我到现在也没搞懂这两者的区别：

```
awk 'BEGIN{ARGC+=1;ARGV[1]="a"} {print}'
file a
```

CONVFMT与OFMT

Awk中允许数值到字符串相互转换，其中内置变量CONVFMT定义了awk内部数值到字符串转换的格式，它的默认值为”%.6g”：

```
awk 'BEGIN {
    printf "CONVFMT=%s, num=%f, str=%s\n", CONVFMT, 12.11, 12.11
}'
CONVFMT=%.6g, num=12.110000, str=12.11
```

通过更改CONVFMT，我们可以定义自己的转换格式：

```
awk 'BEGIN {
    CONVFMT="%d";
    printf "CONVFMT=%s, num=%f, str=%s\n", CONVFMT, 12.11, 12.11
}'
CONVFMT=%d, num=12.110000, str=12
```

与此对应地还有一个内置变量 *OFMT*，它与CONVFMT的作用是类似的，只不过是影响输出的时候数字转换成字符串的格式：

```
awk 'BEGIN { OFMT="%d";print 12.11 }'
12
```

ENVIRON

ENVIRON是一个存放系统环境变量的关联数组，它的下标是环境变量名称，值是相应环境变量的值。例如：

```
awk 'BEGIN { print ENVIRON["USER"] }'
xingming
```

利用环境变量也可以将值传递给awk：

```
U=hello awk 'BEGIN { print ENVIRON["U"] }'
hello
```

可以利用for..in循环遍历ENVIRON数组：

```
awk 'BEGIN {
    for (env in ENVIRON)
        printf "%s=%s\n", env, ENVIRON[env];
}'
```

RLENGTH与RSTART

RLENGTH与RSTART都是与match函数相关的，前者表示匹配的子串长度，后者表示匹配的子串位于目标字符串的起始下标。例如：

```
awk 'BEGIN {match("hello,world", /llo/); print RSTART,RLENGTH}'
3 3
```

运算符

表达式中必然少不了运算符，awk支持的运算符可以参见man手册中的“Expressions in awk”一小节内容：

```
man awk | grep "^ *Table: Expressions in" -A 42 | sed 's/^ *//'
Table: Expressions in Decreasing Precedence in awk

Syntax              Name                    Type of Result   Associativity
( expr )            Grouping                Type of expr     N/A
$expr               Field reference         String           N/A
++ lvalue           Pre-increment           Numeric          N/A
-- lvalue           Pre-decrement           Numeric          N/A
lvalue ++           Post-increment          Numeric          N/A
lvalue --           Post-decrement          Numeric          N/A
expr ^ expr         Exponentiation          Numeric          Right
! expr              Logical not             Numeric          N/A

+ expr              Unary plus              Numeric          N/A
- expr              Unary minus             Numeric          N/A
expr * expr         Multiplication          Numeric          Left

...以下省略...
```

语句（Statement）

到目前为止，用得比较多的语句就是print，其它的还有printf、delete、break、continue、exit、next等等。这些语句与函数不同的是，它们不会使用带括号的参数，并且没有返回值。不过也有意外，比如printf就可以像函数一样的调用：

```
echo 1 | awk '{printf("%s\n", "abc")}'
abc
```

break和continue语句，大家应该比较了解，分别用于跳出循环和跳到下一个循环。

delete用于删除数组中的某个元素，这个我们在上面介绍ARGV的时候也使用过。

exit的用法顾名思义，就是退出awk的处理，然后会执行END部分的内容：

```
echo $'line1\nline2' | awk '{print;exit} END {print "exit.."}'
line1
exit..
```

next语句类似sed的n命令，它会读取下一条记录，并重新回到脚本的最开始处执行：

```
echo $'line1\nline2' | awk '{
    print "Before next.."
    print $0
    next
    print "After next.."
}'
Before next..
line1
Before next..
line2
```

从上面可以看出next后面的print语句不会执行。

print与printf语句是使用最多的，它们将内容输出到标准输出。注意在print语句中，输出的变量之间带不带逗号是有区别的：

```
echo "1 2" | awk '{print $1, $2}'
1 2
echo "1 2" | awk '{print $1 $2}'
12
```

print输出时，字段之间的分隔符可以由OFS重新定义：

```
$ echo "1 2" | awk '{OFS=";";print $1,$2}'
1;2
```

除此之外，print的输出还可以重定向到某个文件中或者某个命令：

```
print items > output-file
print items >> output-file
print items | command
```

假设有这一样一个文件，第一列是语句名称，第二列是对应的说明：

```
$ cat column.txt
statement|description
delete|delete item from an array
exit|exit from the awk process
next|read next input record and process
```

现在我们要将两列的内容分别输出到statement.txt和description.txt两个文件中：

```
$ awk -F'|' '{
    print $1 > "statement.txt";
    print $2 > "description.txt"
}' column.txt
[kodango@devops awk_temp]$ cat statement.txt
statement
delete
exit
next
[kodango@devops awk_temp]$ cat description.txt
description
delete item from an array
exit from the awk process
read next input record and process
```

下面是一个重定向到命令的例子，假设我们要对下面的文件进行排序：

```
$ cat num.list
1
3
2
9
5
```

可以通过将print的内容重定向到”sort -n”命令：

```
$ awk '{print | "sort -n"}' num.list
1
2
3
5
9
```

printf命令的用法与print类似，也可以重定向到文件或者输出，只不过printf比print多了格式化字符串的功能。printf的语法也大多数语言包括bash的printf命令类似，这里就不多介绍了。

数学函数

awk中支持以下数学函数：

- atan2(y,x)：反正切函数；
- cos(x)：余弦函数；
- sin(x)：正弦函数；
- exp(x)：以自然对数e为底指数函数；
- log(x)：计算以e 为底的对数值；
- sqrt(x)：开平方函数；
- int(x)：将数值转换成整数（绝对值）；
- rand()：返回0到1的一个随机数值，不包含1；
- srand([expr])：设置随机种子，一般与rand函数配合使用，如果参数为空，默认使用当前时间为种子；

例如，我们使用rand()函数生成一个随机数值：

```
$ awk 'BEGIN {print rand(),rand();}'
0.237788 0.291066
$ awk 'BEGIN {print rand(),rand();}'
0.237788 0.291066
```

但是你会发现，每次awk执行都会生成同样的随机数，但是在一次执行过程中产生的随机数又是不同的。因为每次awk执行都使用了同样的种子，所以我们可以用srand()函数来设置种子:

```
$ awk 'BEGIN {srand();print rand(),rand();}'
0.171625 0.00692412
$ awk 'BEGIN {srand();print rand(),rand();}'
0.43269 0.782984
```

这样每次生成的随机数就不一样了。

利用rand()函数我们也可以生成1到n的整数：

```
$ awk '
    function randint(n) { return int(n*rand()); }
    BEGIN { srand(); print randint(10);
}'
3
```

字符串函数

awk中包含大多数常见的字符串操作函数。

sub

sub(ere, repl[, in])

描述：简单地说，就是将in中匹配ere的部分替换成repl，返回值是替换的次数。如果in参数省略，默认使用$0。替换的动作会直接修改变量的值。

下面是一个简单的替换的例子：

```
$ echo "hello, world" | awk '{print sub(/ello/, "i"); print}'
1
hi, world
```

在repl参数中&是一个元字符，它表示匹配的内容，例如：

```
$ awk 'BEGIN {var="kodango"; sub(/kodango/, "hello, &", var); print var}'
hello, kodango
```

gsub

gsub(ere, repl[, in])

描述：同sub()函数功能类似，只不过是gsub()是全局替换，即替换所有匹配的内容。

index

index(s, t)

描述：返回字符串t在s中出现的位置，注意这里位置是从1开始计算的，如果没有找到则返回0。

例如：

```
$ awk 'BEGIN {print index("xingming", "n")}'
2
$ awk 'BEGIN {print index("xingming", "w")}'
0
```

length

length[([s])]

描述：返回字符串的长度，如果参数s没有指定，则默认使用$0作为参数。

例如：

```
$ awk 'BEGIN {print length('xingming');}'
8
$ echo "first line" | awk '{print length();}'
10
```

match

match(s, ere)

描述： 返回字符串s匹配ere的起始位置，如果不匹配则返回0。该函数会定义RSTART和RLENGTH两个内置变量。RSTART与返回值相同，RLENGTH记录匹配子串的长度，如果不匹配则为-1。

例如：

```
awk 'BEGIN {
print match("xingming", /ngmin/);
printf "Matched at: %d, Matched substr length: %d\n", RSTART, RLENGTH;
}'
3
Matched at: 3, Matched substr length: 5
```

split

split(s, a[, fs])

描述：将字符串按照分隔符fs，分隔成多个部分，并存到数组a中。注意，存放的位置是从第1个数组元素开始的。如果fs为空，则默认使用FS分隔。函数返回值分隔的个数。

例如：

```
$ awk 'BEGIN {
    split("1;2;3;4;5", arr, ";")
    for (i in arr)
        printf "arr[%d]=%d\n", i, arr[i];
}'
arr[4]=4
arr[5]=5
arr[1]=1
arr[2]=2
arr[3]=3
```

这里有一个奇怪的地方是for..in..输出的数组不是按顺序输出的，如果要按顺序输出可以用常规的for循环:

```
awk 'BEGIN {
    n=split("1;2;3;4;5", arr, ";")
    for (i=1; i<=n; i++)
        printf "arr[%d]=%d\n", i, arr[i];
}'
arr[1]=1
arr[2]=2
arr[3]=3
arr[4]=4
arr[5]=5
```

sprintf

sprintf(fmt, expr, expr, …)

描述：类似printf，只不过不会将格式化后的内容输出到标准输出，而是当作返回值返回。

例如：

```
$ awk 'BEGIN {
    var=sprintf("%s=%s", "name", "value")
    print var
}'
name=value
```

substr

substr(s, m[, n])

描述：返回从位置m开始的，长度为n的子串，其中位置从1开始计算，如果未指定n或者n值大于剩余的字符个数，则子串一直到字符串末尾为止。

例如：

```
awk 'BEGIN { print substr("xiaoh.me", 2, 3); }'
iao
awk 'BEGIN { print substr("xiaoh.me", 2); }'
iaoh.me
```

tolower

tolower(s)

描述：将字符串转换成小写字符。

例如：

```
awk 'BEGIN {print tolower("XIAOH.ME");}'
xiaoh.me
```

toupper

toupper(s)

描述：将字符串转换成大写字符。

例如

```
awk 'BEGIN {print toupper('xiaoh.me')}'
XIAOH.ME
```

I/O处理函数

getline

getline的用法相对比较复杂，它有几种不同的形式。不过它的主要作用就是从输入中每次获取一行输入。

`expression | getline [var]`

这种形式将前面管道前命令输出的结果作为getline的输入，每次读取一行。如果后面跟有var，则将读取的内容保存到var变量中，否则会重新设置$0和NF。

例如，我们将上面的statement.txt文件的内容显示作为getline的输入：

```
$ awk 'BEGIN { while("cat statement.txt" | getline var) print var}'
statement
delete
exit
next
```

上面的例子中命令要用双引号，`cat statement.txt`，这一点同print/printf是一样的。

如果不加var，则直接写到$0中，注意NF值也会被更新：

```
$ awk 'BEGIN { while("cat statement.txt" | getline) print $0,NF}'
statement 1
delete 1
exit 1
next 1
```

`getline [var]`

第二种形式是直接使用getline，它会从处理的文件中读取输入。同样地，如果var没有，则会设置$0，并且这时候会更新NF, NR和FNR：

```
$ awk '{
    while (getline)
       print NF, NR, FNR, $0;
}' statement.txt
1 2 2 delete
1 3 3 exit
1 4 4 next
```

`getline [var] < expression`

第三种形式从expression中重定向输入，与第一种方法类似，这里就不加赘述了。

close

close函数可以用于关闭已经打开的文件或者管道，例如getline函数的第一种形式用到管道，我们可以用close函数把这个管道关闭，close函数的参数与管道的命令一致：

```
$ awk 'BEGIN {
    while("cat statement.txt" | getline) {
        print $0;
        close("cat statement.txt");
}}'
statement
statement
statement
statement
statement
```

但是每次读了一行后，关闭管道，然后重新打开又重新读取第一行就死循环了。所以要慎用，一般情况下也很少会用到close函数。

system

这个函数很简单，就是用于执行外部命令，例如：

```
$ awk 'BEGIN {system("uname -r");}'
3.6.2-1-ARCH
```

结束语

快速了解Awk系列的几篇文章相对比较粗糙，我是参考Awk的man手册以及《Sed & Awk》附录B总结而成的，但是应该可以让大家对awk有一个大致的了解，欢迎大家一起交流。

> **来自：Xiaoh's Blog**
>
> **作者：pmars**
>
> **链接：http://xiaoh.me/2016/03/23/awk-more/**

END

👇👇扫描二维码领取学习资料👇👇

更多Linux好文请点击【阅读原文】哦
↓↓↓

Read more