AWK 教程

> 原文： https://zetcode.com/lang/awk/

这是 AWK 教程。 它涵盖了 AWK 工具的基础知识。

AWK

AWK 是一种模式扫描和处理语言。 AWK 包含一组针对文本数据流要采取的措施。 AWK 广泛使用正则表达式。 它是大多数类 Unix 操作系统的标准函数。

AWK 于 1977 年在贝尔实验室创立。它的名字取自其作者的姓氏-Alfred Aho， Peter Weinberger 和 Brian Kernighan。

AWK 程序

AWK 程序由一系列模式操作语句和可选的函数定义组成。 它处理文本文件。 AWK 是一种面向行的语言。 它将文件划分为称为记录的行。 每行被分解为字段的序列。 这些字段由特殊变量访问：`$1`读取第一个字段，`$2`读取第二个字段，依此类推。 `$0`变量引用整个记录。

AWK 程序的结构具有以下形式：

```
pattern { action }
```

模式是对每个记录执行的测试。 如果满足条件，则执行操作。 模式或动作都可以省略，但不能两者都省略。 默认模式匹配每行，默认操作是打印记录。

```
awk -f program-file [file-list]
awk program [file-list]
```

AWK 程序可以通过两种基本方式运行：a) 从单独的文件中读取程序； 程序的名称紧随`-f`选项，b) 程序在命令行中用引号引起来。

AWK 单线

AWK 单线性是从命令行运行的简单单发程序。 让我们有以下文本文件：

```
$ cat mywords
brown
tree
craftsmanship
book
beautiful
```

```
existence
ministerial
computer
town
```

我们要打印mywords文件中包含的所有超过五个字符的单词。

```
$ awk 'length($1) > 5 {print}' mywords
craftsmanship
beautiful
existence
ministerial
computer
```

AWK 程序位于两个单引号字符之间。 首先是模式； 我们指定记录的长度大于五。 length()函数返回字符串的长度。 $1变量引用记录的第一个字段； 在我们的情况下，每条记录只有一个字段。 动作放置在大括号之间。

```
$ awk 'length($1) > 5' mywords
craftsmanship
beautiful
existence
ministerial
computer
```

正如我们前面所指定的，该动作可以省略。 在这种情况下，将执行默认操作-打印整个记录。

正则表达式通常应用于 AWK 字段。 ~是正则表达式匹配运算符。 它检查字符串是否与提供的正则表达式匹配。

```
$ awk '$1 ~ /^[b,c]/ {print $1}' mywords
brown
craftsmanship
book
beautiful
computer
```

在此程序中，我们打印所有以b或c字符开头的单词。 正则表达式位于两个斜杠字符之间。

AWK 提供重要的内置变量。 例如，NR是一个内置变量，指向正在处理的当前行。

```
$ awk 'NR % 2 == 0 {print}' mywords
tree
book
existence
computer
```

上面的程序每隔mywords文件打印一次记录。 模除NR变量，我们得到一条偶数行。

假设我们要打印文件的行号。

```
$  awk '{print NR, $0}' mywords
1 brown
2 tree
3 craftsmanship
4 book
5 beautiful
6 existence
7 ministerial
8 computer
9 town
```

同样，我们使用NR变量。 我们跳过该模式，因此，每一行都执行该操作。 $0变量引用整个记录。

对于以下示例，我们具有此 C 源文件。

```
$ cat source.c
1  #include <stdio.h>
2
3  int main(void) {
4
5      char *countries[5] = { "Germany", "Slovakia", "Poland",
6              "China", "Hungary" };
7
8      size_t len = sizeof(countries) / sizeof(*countries);
9
10     for (size_t i=0; i < len; i++) {
11
12         printf("%s\n", countries[i]);
13     }
14 }
```

碰巧我们复制了一些数据，包括行号。 我们的任务是从文本中删除数字。

```
$ awk '{print substr($0, 4)}' source.c
#include <stdio.h>

int main(void) {

    char *countries[5] = { "Germany", "Slovakia", "Poland",
            "China", "Hungary" };

    size_t len = sizeof(countries) / sizeof(*countries);

    for (size_t i=0; i < len; i++) {

        printf("%s\n", countries[i]);
    }
}
```

我们使用substr()函数。 它从给定的字符串打印一个子字符串。 我们在每行上应用该函数，跳过前三个字符。 换句话说，我们从第四个字符开始打印每个记录直到结束。

开始和结束模式

BEGIN和END是在读取所有记录之前和之后执行的特殊模式。 这两个关键字后跟大括号，我们在其中指定要执行的语句。

我们有以下两个文件：

```
$ cat mywords;
brown
tree
craftsmanship
book
beautiful
existence
ministerial
computer
town
$ cat mywords2;
pleasant
curly
storm
hering
immune
```

我们想知道这两行中的行数。

```
$ awk 'END {print NR}' mywords mywords2
14
```

我们将两个文件传递给 AWK 程序。 AWK 按顺序处理在命令行上收到的文件名。 关键字END之后的块在程序结尾处执行； 我们打印NR变量，该变量保存最后处理的行的行号。

```
$ awk 'BEGIN {srand()} {lines[NR] = $0} END { r=int(rand()*NR + 1); print
lines[r]}' mywords
tree
```

上面的程序从mywords文件中打印一条随机行。 srand()函数为随机数生成器提供种子。 该函数仅需执行一次。 在程序的主要部分，我们将当前记录存储到lines数组中。 最后，我们计算 1 到NR之间的随机数，并打印从数组结构中随机选择的行。

匹配函数

match()是内置的字符串操作函数。 它测试给定的字符串是否包含正则表达式模式。 第一个参数是字符串，第二个参数是正则表达式模式。 它类似于~运算符。

```
$ awk 'match($0, /^[c,b]/)' mywords
brown
craftsmanship
book
beautiful
computer
```

程序将打印以c或b开头的行。 正则表达式位于两个斜杠字符之间。

match()函数设置RSTART变量； 它是匹配模式开始的索引。

```
$ awk 'match($0, /i/) {print $0 " has i character at " RSTART}' mywords
craftsmanship has i character at 12
beautiful has i character at 6
existence has i character at 3
ministerial has i character at 2
```

程序将打印出包含i字符的单词。 此外，它还会打印字符的首次出现。

AWK 内置变量

AWK 有几个内置变量。 它们在运行程序时由 AWK 设置。 我们已经看到了NR, $0和RSTART变量。

```
$ awk 'BEGIN { print ARGC, ARGV[0], ARGV[1]}' mywords
2 awk mywords
```

该程序将打印 AWK 程序的参数数量和前两个参数。 ARGC是命令行参数的数量； 在我们的案例中，有两个论点，包括 AWK 本身。 ARGV是命令行参数数组。 数组的索引从 0 到ARGC-1。

FS是输入字段分隔符，默认为空格。 NF是当前输入记录中的字段数。

对于以下程序，我们使用此文件：

```
$ cat values
2, 53, 4, 16, 4, 23, 2, 7, 88
4, 5, 16, 42, 3, 7, 8, 39, 21
23, 43, 67, 12, 11, 33, 3, 6
```

我们有三行用逗号分隔的值。

stats.awk

```
BEGIN {

    FS=","
    max = 0
    min = 10**10
    sum = 0
    avg = 0
}

{
    for (i=1; i<=NF; i++) {

        sum += $i

        if (max < $i) {
            max = $i
        }

        if (min > $i) {
            min = $i
        }

        printf("%d ",  $i)
    }
}
```

```
END {

    avg = sum / NF
    printf("\n")
    printf("Min: %d, Max: %d, Sum: %d, Average: %d\n", min, max, sum, avg)
}
```

程序将从提供的值中统计基本统计信息。

```
FS=","
```

文件中的值用逗号分隔； 因此，我们将FS变量设置为逗号字符。

```
max = 0
min = 10**10
sum = 0
avg = 0
```

我们定义最大值，最小值，总和和平均值的默认值。 AWK 变量是动态的； 它们的值可以是浮点数或字符串，或两者兼有，这取决于它们的使用方式。

```
{
    for (i=1; i<=NF; i++) {

        sum += $i

        if (max < $i) {
            max = $i
        }

        if (min > $i) {
            min = $i
        }

        printf("%d ",  $i)
    }
}
```

在脚本的主要部分，我们遍历每一行并计算值的最大值，最小值和总和。 NF用于确定每行的值数量。

```
END {

    avg = sum / NF
    printf("\n")
    printf("Min: %d, Max: %d, Sum: %d, Average: %d\n", min, max, sum, avg)
}
```

在脚本的最后，我们计算平均值并将计算结果打印到控制台。

```
$ awk -f stats.awk values
2 53 4 16 4 23 2 7 88 4 5 16 42 3 7 8 39 21 23 43 67 12 11 33 3 6
Min: 2, Max: 88, Sum: 542, Average: 67
```

这是stats.awk程序的输出。

可以使用-F标志将FS变量指定为命令行选项。

```
$ awk -F: '{print $1, $7}' /etc/passwd | head -7
root /bin/bash
daemon /usr/sbin/nologin
bin /usr/sbin/nologin
sys /usr/sbin/nologin
sync /bin/sync
games /usr/sbin/nologin
man /usr/sbin/nologin
```

该示例从系统/etc/passwd文件中打印第一个（用户名）和第七个字段（用户的外壳程序）。 head命令仅用于打印前七行。 /etc/passwd文件中的数据用冒号分隔。 因此，冒号被赋予-F选项。

RS是输入记录分隔符，默认情况下是换行符。

```
$ echo "Jane 17#Tom 23#Mark 34" | awk 'BEGIN {RS="#"} {print $1, "is", $2,
"years old"}'
Jane is 17 years old
Tom is 23 years old
Mark is 34 years old
```

在示例中，我们用#字符分隔了相关数据。 RS用于剥离它们。 AWK 可以从echo之类的其他命令接收输入。

将变量传递给 AWK

AWK 具有-v选项，用于为变量分配值。 对于下一个程序，我们具有text文件：

```
$ cat text
The French nation, oppressed, degraded during many centuries
by the most insolent despotism, has finally awakened to a
consciousness of its rights and of the power to which its
destinies summon it.
```

mygrep.awk

```
{
    for (i=1; i<=NF; i++) {

        field = $i

        if (field ~ word) {
            c = index($0, field)
            print NR "," c, $0
            next
        }
    }
}
```

该示例模拟grep工具。 它找到提供的单词并打印其行和起始索引。 （程序仅找到单词的第一个出现。）使用-v选项将word变量传递给程序。

```
$ awk -f mygrep.awk -v word=the text
2,4 by the most insolent despotism, has finally awakened to a
3,36 consciousness of its rights and of the power to which its
```

我们在text文件中寻找了"the"字样。

管道

AWK 可以通过管道接收输入并将输出发送到其他命令。

```
$ echo -e "1 2 3 5\n2 2 3 8" | awk '{print $(NF)}'
5
8
```

在这种情况下，AWK 从echo命令接收输出。 打印最后一列的值。

```
$ awk -F: '$7 ~ /bash/ {print $1}' /etc/passwd | wc -l
3
```

在此，AWK 程序通过管道将数据发送到wc命令。 在 AWK 程序中，我们找出使用 bash 的用户。 它们的名称被传递给wc命令，该命令对其进行计数。 在我们的例子中，有三个用户使用 bash。

拼写检查

我们创建一个用于拼写检查的 AWK 程序。

spellcheck.awk

```
BEGIN {
    count = 0

    i = 0
    while (getline myword <"/usr/share/dict/words") {
        dict[i] = myword
        i++
    }
}

{
    for (i=1; i<=NF; i++) {

        field = $i

        if (match(field, /[[:punct:]]$/)) {
            field = substr(field, 0, RSTART-1)
        }

        mywords[count] = field
        count++
    }
}

END {

    for (w_i in mywords) {
        for (w_j in dict) {
            if (mywords[w_i] == dict[w_j] ||
                        tolower(mywords[w_i]) == dict[w_j]) {
                delete mywords[w_i]
            }
        }
    }

    for (w_i in mywords) {
        if (mywords[w_i] != "") {
```

```
            print mywords[w_i]
        }
    }
}
```

该脚本将提供的文本文件的单词与字典进行比较。 在标准/usr/share/dict/words路径下，我们可以找到英语词典； 每个单词在单独的行上。

```
BEGIN {
    count = 0

    i = 0
    while (getline myword <"/usr/share/dict/words") {
        dict[i] = myword
        i++
    }
}
```

在BEGIN块内部，我们将字典中的单词读入dict数组。 getline命令从给定的文件名中读取一条记录； 记录存储在$0变量中。

```
{
    for (i=1; i<=NF; i++) {

        field = $i

        if (match(field, /[[:punct:]]$/)) {
            field = substr(field, 0, RSTART-1)
        }

        mywords[count] = field
        count++
    }
}
```

在程序的主要部分，我们将要进行拼写检查的文件的单词放入mywords数组。 我们会删除单词结尾处的所有标点符号（例如逗号或点）。

```
END {

    for (w_i in mywords) {
        for (w_j in dict) {
            if (mywords[w_i] == dict[w_j] ||
                        tolower(mywords[w_i]) == dict[w_j]) {
```

```
                delete mywords[w_i]
            }
        }
    }
...
}
```

我们将mywords数组中的单词与字典数组进行比较。 如果单词在词典中，则使用delete命令将其删除。 以句子开头的单词以大写字母开头； 因此，我们还利用tolower()函数检查小写字母的替换形式。

```
for (w_i in mywords) {
    if (mywords[w_i] != "") {
        print mywords[w_i]
    }
}
```

在词典中找不到其余的单词； 它们被打印到控制台。

```
$ awk -f spellcheck.awk text
consciosness
finaly
```

我们已经在文本文件上运行了该程序； 我们发现了两个拼写错误的单词。 请注意，该程序需要一些时间才能完成。

剪刀石头布

剪刀石头布是一种流行的手形游戏，其中每个玩家都用伸出的手同时形成三个形状之一。 我们在 AWK 中创建此游戏。

rock_scissors_paper.awk

```
# This program creates a rock-paper-scissors game.

BEGIN {

    srand()

    opts[1] = "rock"
    opts[2] = "paper"
    opts[3] = "scissors"

    do {
```

```
        print "1 - rock"
        print "2 - paper"
        print "3 - scissors"
        print "9 - end game"

        ret = getline < "-"

        if (ret == 0 || ret == -1) {
            exit
        }

        val = $0

        if (val == 9) {
            exit
        } else if (val != 1 && val != 2 && val != 3) {
            print "Invalid option"
            continue
        } else {
            play_game(val)
        }

    } while (1)
}

function play_game(val) {

    r = int(rand()*3) + 1

    print "I have " opts[r] " you have "  opts[val]

    if (val == r) {
        print "Tie, next throw"
        return
    }

    if (val == 1 && r == 2) {

        print "Paper covers rock, you loose"
    } else if (val == 2 && r == 1) {

        print "Paper covers rock, you win"
    } else if (val == 2 && r == 3) {

        print "Scissors cut paper, you loose"
    } else if (val == 3 && r == 2) {

        print "Scissors cut paper, you win"
    } else if (val == 3 && r == 1) {

        print "Rock blunts scissors, you loose"
    } else if (val == 1 && r == 3) {

        print "Rock blunts scissors, you win"
```

```
    }
}
```

我们在电脑上玩游戏，电脑会随机选择选项。

```
srand()
```

我们使用srand()函数为随机数生成器播种。

```
opts[1] = "rock"
opts[2] = "paper"
opts[3] = "scissors"
```

这三个选项存储在opts数组中。

```
do {

    print "1 - rock"
    print "2 - paper"
    print "3 - scissors"
    print "9 - end game"
...
```

游戏的周期由do-while循环控制。 首先，将选项打印到终端。

```
ret = getline < "-"

if (ret == 0 || ret == -1) {
    exit
}

val = $0
```

我们选择的值是使用getline命令从命令行读取的； 该值存储在val变量中。

```
if (val == 9) {
    exit
} else if (val != 1 && val != 2 && val != 3) {
```

```
    print "Invalid option"
    continue
} else {
    play_game(val)
}
```

如果选择选项 9，则退出程序。如果该值在打印的菜单选项之外，则打印错误消息，并使用continue命令开始新的循环。 如果我们正确选择了三个选项之一，则调用play_game()函数。

```
r = int(rand()*3) + 1
```

使用rand()函数从1..3中选择一个随机值。 这是计算机的选择。

```
if (val == r) {
    print "Tie, next throw"
    return
}
```

如果两个玩家选择相同的选项，则平局。 我们从函数返回并开始新的循环。

```
if (val == 1 && r == 2) {

    print "Paper covers rock, you loose"
} else if (val == 2 && r == 1) {
...
```

我们比较所选播放器的值，并将结果打印到控制台。

```
$ awk -f rock_scissors_paper.awk
1 - rock
2 - paper
3 - scissors
9 - end game
1
I have scissors you have rock
Rock blunts scissors, you win
1 - rock
2 - paper
3 - scissors
9 - end game
3
```

```
I have paper you have scissors
Scissors cut paper, you win
1 - rock
2 - paper
3 - scissors
9 - end game
```

游戏示例。

标记关键字

在下面的示例中，我们在源文件中标记 Java 关键字。

mark_keywords.awk

```
# the program adds tags around Java keywords
# it works on keywords that are separate words

BEGIN {

    # load java keywords
    i = 0
    while (getline kwd <"javakeywords2") {
        keywords[i] = kwd
        i++
    }
}

{
    mtch = 0
    ln = ""
    space = ""

    # calculate the beginning space
    if (match($0, /[^[:space:]]/)) {
        if (RSTART > 1) {
            space = sprintf("%*s", RSTART, "")
        }
    }

    # add the space to the line
    ln = ln space

    for (i=1; i <= NF; i++) {

        field = $i

        # go through keywords
        for (w_i in keywords) {
```

```
            kwd = keywords[w_i]

            # check if a field is a keyword
            if (field == kwd) {
                mtch = 1
            }
        }

        # add tags to the line
        if (mtch == 1) {
            ln = ln  "<kwd>" field  "</kwd> "
        } else {
            ln = ln field " "
        }

        mtch = 0

    }

    print ln
}
```

该程序在它识别的每个关键字周围添加<kwd>和</kwd>标签。 这是一个基本示例； 它适用于单独单词的关键字。 它没有解决更复杂的结构。

```
# load java keywords
i = 0
while (getline kwd <"javakeywords2") {
    keywords[i] = kwd
    i++
}
```

我们从文件中加载 Java 关键字； 每个关键字在单独的行上。 关键字存储在keywords数组中。

```
# calculate the beginning space
if (match($0, /[^[:space:]]/)) {
    if (RSTART > 1) {
        space = sprintf("%*s", RSTART, "")
    }
}
```

使用正则表达式，我们计算行首的空格（如果有）。 space是一个字符串变量，等于当前行的空格宽度。 计算空间是为了保持程序缩进。

```
# add the space to the line
ln = ln space
```

将空格添加到ln变量中。 在 AWK 中，我们使用空格添加字符串。

```
for (i=1; i <= NF; i++) {

field = $i
...
}
```

我们遍历当前行的字段； 该字段存储在field变量中。

```
# go through keywords
for (w_i in keywords) {

    kwd = keywords[w_i]

    # check if a field is a keyword
    if (field == kwd) {
        mtch = 1
    }
}
```

在for循环中，我们遍历 Java 关键字，并检查字段是否为 Java 关键字。

```
# add tags to the line
if (mtch == 1) {
    ln = ln  "<kwd>" field  "</kwd> "
} else {
    ln = ln field " "
}
```

如果有关键字，我们将标签附加在关键字周围； 否则，我们只是将字段附加到该行。

```
print ln
```

构建的行将打印到控制台。

```
$ awk -f markkeywords2.awk program.java
<kwd>package</kwd> com.zetcode;

<kwd>class</kwd> Test {

    <kwd>int</kwd> x = 1;

    <kwd>public</kwd> <kwd>void</kwd> exec1() {

        System.out.println(this.x);
        System.out.println(x);
    }

    <kwd>public</kwd> <kwd>void</kwd> exec2() {

        <kwd>int</kwd> z = 5;

        System.out.println(x);
        System.out.println(z);
    }
}

<kwd>public</kwd> <kwd>class</kwd> MethodScope {

    <kwd>public</kwd> <kwd>static</kwd> <kwd>void</kwd> main(String[]
args) {

        Test ts = <kwd>new</kwd> Test();
        ts.exec1();
        ts.exec2();
    }
}
```

在小型 Java 程序上运行的示例。

这是 AWK 教程。