AWK教程

极客学院出版

前言

本教程将会带你学习 GNU/Linux 上最卓越的文件处理工具 AWK。 与其它 GNU/Linux 工具一样，AWK 非常强大，而且只用到十分简单的编程语言。 它仅仅需要几行代码就能够完成复杂的文本处理工作。这个简单强大的工具也使得 GNU/Linux 变得有意思多了。

适用群体

如果你是软件开发人员，或者系统管理人员，或者 GNU/Linux爱好者，那么这份教程就是为你量身定做的。

预备知识

你需要提前对 GNU/Linux 以及 shell 脚本编程有一个基本的了解。

目录

1

概述

AWK 是一种解释执行的编程语言。它非常的强大，被设计用来专门处理文本数据。AWK 的名称是由它们设计者的名字缩写而来 —— Afred Aho, Peter Weinberger 与 Brian Kernighan。

由 GNU/Linux 发布的 AWK 版本通常被称之为 GNU AWK，由自由软件基金（ Free Software Foundation, FSF ）负责开发维护的。 目前总共有如下几种不同的 AWK 版本。

- AWK——这个版本是 AWK 最原初的版本，它由 AT&T 实验室开发。
- NAWK ——NAWK(New AWK)是 AWK 的改进增强版本。
- GAWK—— GAWK 即 GNU AWK，所有的 GNU/Linux 发行版都包括 GAWK，且 GAWK 完全兼容 AWK 与 NAWK。

部分 AWK 的典型应用场景

AWK 可以做非常多的工作。 下面只是其中的一小部分：

- 文本处理，
- 生成格式化的文本报告，
- 进行算术运算，
- 字符串操作，以及其它更多。

2

运行环境

这一章节将会讲解如何在你的 GNU/Linux 系统中如何搭建 AWK 的运行环境

使用包管理器安装 AWK

一般情况下，绝大多数 GNU/Linux 发行版中都默认安装了 AWK。 使用 which 命令可以判断你当前的系统上是否安装了 AWK。

如果没有安装，在Debian GNU/Linux 系统中你可以使用 apt 包管理工具安装 AWK。如下所示：

```
[jeryy]$ sudo apt-get update
[jeryy]$ sudo apt-get install gawk
```

同样地，在基于RPM的 GNU/Linux 的系统上安装 AWK 时，你可以使用 yum 包管理工具安装：

```
[root]# yum install gawk
```

安装过后，使用命令行命令确认 AWK 已成安装成功：

```
[jerry]$ which awk
```

执行上面的代码，如果得到如下的结果，则说明你已经成功安装 awk：

```
/usr/bin/awk
```

由源码安装

因为 GNU AWK 是 GNU 项目的一部分，所以它的源代码也是可以自由下载的。在前面我们已经看到了如何使用包管理器安装 AWK。接下来，我们将讲解如何通过源代码安装 AWK。

下面的安装方法适用于所有的 GNU/Linux 软件，同时也适用于大部分其它自由应用程序。下面是安装的步骤：

step 1——从可信的源下载源代码。可以在命令行使用 wget 命令下载。

```
[jerry]$ wget http://ftp.gnu.org/gnu/gawk/gawk-4.1.1.tar.xz
```

step 2——解压并提取下载的源代码。

```
[jerry]$ tar xvf gawk-4.1.1.tar.xz
```

step 3——切换至解压后的目录并运行 configure 命令

```
[jerry]$ ./configure
```

step 4——configure 命令成功执行后会生成一个 Makefile 文件。 接下来使用 make 命令编译源代码。

```
[jerry]$ make
```

step 5——你可以运行测试工具保证 build 是干净的。 这一步是可选的。

```
[jerry]$ make check
```

step 6——最后一步，安装 AWK。 安装前请确认你有超级用户的权限。

```
[jerry]$ sudo make install
```

通过以上六个步骤，你就成功地编译并安装了 AWK。 你可以通过如下的命令来确认 awk 安装成功：

```
[jerry]$ which awk
```

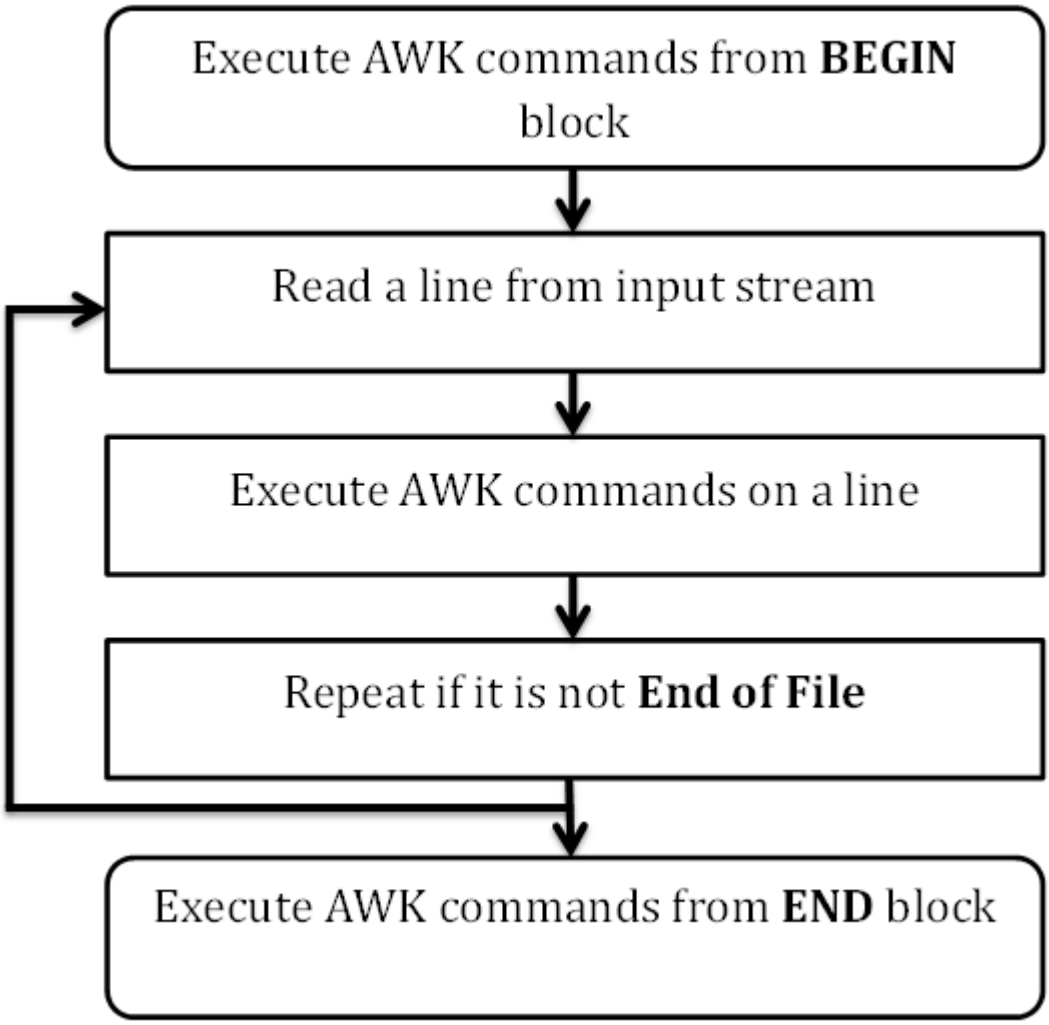
执行上面的命令，你将会得到如下的结果：

```
/usr/bin/awk
```

3

工作流程

这一章节中，我们将解释 AWK 是如何工作的。 要想成为 AWK 专家，你必须得了解其内部工作的原理。 AWK 执行的流程非常简单：读(Read)、执 行 （ Execute)与重复(Repeat)。下面的流程图描述出了 AWK 的工作流程：



读 (Read)

AWK 从输入流（ 文件、管道或者标准输入 ）中读入一行然后将其存入内存中。

执行(Execute)

对于每一行输入，所有的 AWK 命令按顺执行。 默认情况下，AWK 命令是针对于每一行输入，但是我们可以将其限制在指定的模式中。

重复（Repeate）

一直重复上述两个过程直到文件结束。

程序的结构

我们已经见过 AWK 程序的工作流程。 现在让我们一起来学习 AWK 程序的结构。

开始块（BEGIN block）

开始块的语法格式如下所示：

```
BEGIN {awk-commands}
```

顾名思义，开始块就是在程序启动的时候执行的代码部分，并且它在整个过程中只执行一次。 一般情况下，我们在开始块中初始化一些变量。BEGIN 是 AWK 的关键字，因此它必须是大写的。 不过，请注意，开始块部分是可选的，你的程序可以没有开始块部分。

主体块（Body Block）

主体部分的语法要求如下：

```
/pattern/ {awk-commands}
```

对于每一个输入的行都会执行一次主体部分的命令。默认情况下，对于输入的每一行，AWK 都会很执行命令。但是，我们可以将其限定在指定的模式中。 注意，在主体块部分没有关键字存在。

结束块（END Block）

下面是结束块的语法格式：

```
END {awk-commands}
```

结束块是在程序结束时执行的代码。 END 也是 AWK 的关键字，它也必须大写。 与开始块相似，结束块也是可选的。

示例

先创建一个名为 marks.txt 的文件。其中包括序列号、学生名字、课程名称与所得分数。

```
1)  Amit    Physics   80
2)  Rahul   Maths     90
3)  Shyam   Biology   87
4)  Kedar   English   85
5)  Hari    History   89
```

接下来，我们将使用 AWK 脚本来显示输出文件中的内容，同时输出表头信息。

```
[jerry]$ awk 'BEGIN{printf "Sr No\tName\tSub\tMarks\n"} {print}' marks.txt
```

执行上面的代码后，将会输出如下的结果：

```
Sr No Name    Sub       Marks
1)  Amit    Physics   80
2)  Rahul   Maths     90
3)  Shyam   Biology   87
4)  Kedar   English   85
5)  Hari    History   89
```

程序启动时，AWK 在开始块中输出表头信息。在主体块中，AWK 每读入一行就将读入的内容输出至标准输出流中，一直到整个文件被全部读入为止。

4

基本语法

AWK 使用起来非常方便。我们可以直接通过命令行的方式为 AWK 程序提供 AWK 命令，也可以使用包括 AWK 命令的脚本文件。这篇教程将使用合适的例子分别介绍这两种使用 AWK 的方法:

AWK 命令行

如下所示，在命令行中，我们可以使用如下的格式调用 AWK 命令，其中 AWK 命令由单引号括起来:

```
awk [options] file ...
```

例子

假设我们有一个名为marks.txt的文件需要处理，文件中的内容如下:

```
1)  Amit    Physics   80
2)  Rahul   Maths     90
3)  Shyam   Biology   87
4)  Kedar   English   85
5)  Hari    History   89
```

我们可以按如下方式使用 AWK 命令输出整个文件中的内容:

```
[jerry]$ awk '{print}' marks.txt
```

执行上面的命令可以得到如下的结果:

```
1)  Amit    Physics   80
2)  Rahul   Maths     90
3)  Shyam   Biology   87
4)  Kedar   English   85
5)  Hari    History   89
```

AWK 程序文件

接下来讲解的是另外一种提供 AWK 命令的方式——通过脚本文件提供:

```
awk [option] -f file ....
```

首先，创建一个文本文件 command.awk，在文件中输入如下 AWK 命令:

```
{print}
```

现在，我们可以调用 AWK 从文本文件中读入命令并执行。这里，我们实现了与上面例子相同的效果:

```
[jerry]$ awk -f command.awk marks.txt
```

执行上面的命令可以得到如下的结果：

```
1)  Amit    Physics   80
2)  Rahul   Maths     90
3)  Shyam   Biology   87
4)  Kedar   English   85
5)  Hari    History   89
```

AWK 标准选项

在命令行环境下，AWK 支持如下的标准选项：

-v 选项

这个选项可以为变量赋值。它允许在程序执行之前为变量赋值。下面是一个 -v 选项使用的示例程序：

```
[jerry]$ awk -v name=Jerry 'BEGIN{printf "Name = %s\n", name}'
```

执行上面的命令可以得到如下的结果：

```
Name = Jerry
```

--dump-variables[=file] 选项

此选项会将全局变量及相应值按序输出到指定文件中。默认的输出文件名是 awkvars.out。

```
[jerry]$ awk --dump-variables ''
[jerry]$ cat awkvars.out
```

执行上面的命令可以得到如下的结果：

```
ARGC: 1
ARGIND: 0
ARGV: array, 1 elements
BINMODE: 0
CONVFMT: "%.6g"
ERRNO: ""
FIELDWIDTHS: ""
FILENAME: ""
FNR: 0
FPAT: "[^[:space:]]+"
```

```
FS: " "
IGNORECASE: 0
LINT: 0
NF: 0
NR: 0
OFMT: "%.6g"
OFS: " "
ORS: "\n"
RLENGTH: 0
RS: "\n"
RSTART: 0
RT: ""
SUBSEP: "\034"
TEXTDOMAIN: "messages"
```

--help 选项

此选项将帮助消息转出到标准输出中。

```
[jerry]$ awk --help
```

执行上面的命令可以得到如下的结果：

```
Usage: awk [POSIX or GNU style options] -f progfile [--] file ...
Usage: awk [POSIX or GNU style options] [--] 'program' file ...
POSIX options:          GNU long options: (standard)
    -f progfile         --file=progfile
    -F fs               --field-separator=fs
    -v var=val          --assign=var=val
Short options:          GNU long options: (extensions)
    -b                  --characters-as-bytes
    -c                  --traditional
    -C                  --copyright
    -d[file]            --dump-variables[=file]
    -e 'program-text'   --source='program-text'
    -E file             --exec=file
    -g                  --gen-pot
    -h                  --help
    -L [fatal]          --lint[=fatal]
    -n                  --non-decimal-data
    -N                  --use-lc-numeric
    -O                  --optimize
    -p[file]            --profile[=file]
    -P                  --posix
    -r                  --re-interval
```

```
-S          --sandbox
-t          --lint-old
-V          --version
```

--lint[=fatal] 选项

这个选项用于检查程序的可移植情况以及代码中的可疑部分。如果提供了参数 fatal，AWK 会将所有的警告信息当作错误信息处理。下面这个简单的示例说明了 lint 选项的用法：

```
[jerry]$ awk --lint '' /bin/ls
```

执行上面的命令可以得到如下的结果：

```
awk: cmd. line:1: warning: empty program text on command line
awk: cmd. line:1: warning: source file does not end in newline
awk: warning: no program text at all!
```

--posix 选项

这个选项会打开严格 POSIX 兼容性审查。 如此，所有共同的以及 GAWK 特定的扩展将被设置为无效。

--profile[=file] 选项

这个选项会将程序文件以一种很优美的方式输出（译注：用于格式化 awk 脚本文件）。默认输出文件是 awkprof.out。示例如下：

```
[jerry]$ awk --profile 'BEGIN{printf"---|Header|--\n"} {print} END{printf"---|Footer|---\n"}' marks.txt > /dev/null
[jerry]$ cat awkprof.out
```

执行上面的命令可以得到如下的结果：

```
\# gawk profile, created Sun Oct 26 19:50:48 2014

    # BEGIN block(s)

    BEGIN {
        printf "---|Header|--\n"
    }

    # Rule(s)
```

```
{
   print $0
}

# END block(s)

END {
   printf "---|Footer|---\n"
}
```

--traditional 选项

此选项用于禁止 GAWK 相关的扩展。

--version 选项

此选项显示 AWK 程序的版本信息。

```
[jerry]$ awk --version
```

上面的代码执行后，将产生下面的输出结果(译注：与具体的 AWK 版本相关)：

```
GNU Awk 4.0.1
Copyright (C) 1989, 1991-2012 Free Software Foundation.
```

5

基本示例

本章节中，我们将用几个示例来讲解几个有用的 AWK 命令。 假设我们经有一个文件文件 marks.txt 等待处理，它所包含的内容如下：

```
1)  Amit    Physics   80
2)  Rahul   Maths     90
3)  Shyam   Biology   87
4)  Kedar   English   85
5)  Hari    History   89
```

打印列或域

我们可以使用 AWK 命令仅输出输入文件中某些特定的列的内容。示例如下：

```
[jerry]$ awk '{print $3 "\t" $4}' marks.txt
```

执行上面的命令可以得到如下结果：

```
Physics   80
Maths     90
Biology   87
English   85
History   89
```

在 marks.txt 文件中，第三列包含课程名字，第四列包含在该课程的得分。我们使用 AWK 输出命令只输出了这两列的内容。上面例子中，$3与$4代表输入记录中的第三列与第四列的内容。

输出所有行

默认情况下，如果某行与模式串匹配，AWK 会将整行输出：

```
[jerry]$ awk '/a/ {print $0}' marks.txt
```

执行上面的命令可以得到如下结果：

```
2)  Rahul   Maths     90
3)  Shyam   Biology   87
4)  Kedar   English   85
5)  Hari    History   89
```

上面的示例中，我们搜索模式串 a，每次成功匹配后都会执行主体块中的命令。如果没有主体块——默认的动作是输出记录（行）。因此上面的效果也可以使用下面简略方式实现,它们会得到相同的结果：

```
[jerry]$ awk '/a/' marks.txt
```

通过匹配模式串输出列

前面我们已经看到了，当模式串匹配成功后， AWK 默认会输出整个记录。不过，我们可以让 AWK 只输出特定的域（列）的内容。 例如，下面的这个例子中当模式串匹配成功后只会输出第三列与第四列的内容:

```
[jerry]$ awk '/a/ {print $3 "\t" $4}' marks.txt
```

执行上面的命令可以得到如下结果:

```
Maths     90
Biology   87
English   85
History   89
```

以任意顺序输出列

我们能以任意顺序输出各列吗？当然可以！ 下面的例子中我们将在第四列后输出第三列的内容:

```
[jerry]$ awk '/a/ {print $4 "\t" $3}' marks.txt
```

执行上面的命令可以得到如下结果:

```
90    Maths
87    Biology
85    English
89    History
```

计数匹配次数并输出

让我们尝试一个更有意思的例子，在这个例子中我们会统计模式串成功匹配的次数，并将该结果打印出来:

```
[jerry]$ awk '/a/{++cnt} END {print "Count = ", cnt}' marks.txt
```

执行上面的命令可以得到如下结果:

```
Count = 4
```

上面这个例子中，每次成功的匹配我们都会增加计数器的值，并在结束块中将该计数器的值输出。 请注意，与其它编程语言不一样的地方在于， AWK 在使用一个变量前不需要特意地声明这个变量。

输出字符数多于 18 的行

这个例子中我们只输出那些字符数超过 18 的记录：

```
[jerry]$ awk 'length($0) > 18' marks.txt
```

执行上面的命令可以得到如下结果：

```
3)  Shyam   Biology   87
4)  Kedar   English   85
```

AWK 提供了内置的 length 函数。该函数返回字符串的长度。变量 $0 表示整行，缺失的主体块会执行默认动作，例如，打印输出。 因此，如果一行中字符数超过 18， 则比较的结果为真，该行则被输出。

6

内置变量

AWK 提供了一些内置变量。 它们在你写 AWK 脚本的时候起着很重要的作用。 这一章节中将会展示如何使用这些内置变量。

标准 AWK 变量

下面将介绍标准 AWK 变量：

ARGC

ARGC 表示在命令行提供的参数的个数。

```
[jerry]$ awk 'BEGIN {print "Arguments =", ARGC}' One Two Three Four
```

执行上面的命令可以得到如下的结果：

```
Arguments = 5
```

程序哪儿出毛病了吗？为什么只输入四个参数而 AWK 却显示输入的参数个数的五呢？ 看完下面这个例子，你就会明白的。

ARGV

这个变量表示存储命令行输入参数的数组。数组的有效索引是从 0 到 ARGC−1。

```
[jerry]$ awk 'BEGIN { for (i = 0; i < ARGC − 1; ++i)
    { printf "ARGV[%d] = %s\n", i, ARGV[i] }
            }' one two three four
```

执行上面的命令可以得到如下的结果：

```
ARGV[0] = awk
ARGV[1] = one
ARGV[2] = two
ARGV[3] = three
```

CONVFMT

此变量表示数据转换为字符串的格式，其默认值为 %.6g。

```
[jerry]$ awk 'BEGIN { print "Conversion Format =", CONVFMT }'
```

执行上面的命令可以得到如下的结果：

```
Conversion Format = %.6g
```

ENVIRON

此变量是与环境变量相关的关联数组变量。

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

执行上面的命令可以得到如下的结果：

```
jerry
```

可以使用 GNU/Linux 系统中的 env 命令查询其它环境变量的名字。

FILENAME

此变量表示当前文件名称。

```
[jerry]$ awk 'END {print FILENAME}' marks.txt
```

执行上面的命令可以得到如下的结果：

```
marks.txt
```

值得注意的是在开始块中FILENAME是未定义的。

FS

此变量表示输入的数据域之间的分隔符，其默认值是空格。 你可以使用 -F 命令行选项改变它的默认值。

```
[jerry]$ awk 'BEGIN {print "FS = " FS}' | cat -vte
```

执行上面的命令可以得到如下的结果：

```
FS =  $
```

NF

此变量表示当前输入记录中域的数量。例如，下面这个例子只输出超过两个域的行：

```
[jerry]$ echo -e "One Two\nOne Two Three\nOne Two Three Four" | awk 'NF > 2'
```

执行上面的命令可以得到如下的结果：

```
One Two Three
One Two Three Four
```

NR

此变量表示当前记录的数量。（译注：该变量类似一个计数器，统计记录的数量）。下面例子会输出读入的前三行（NR<3）。

```
[jerry]$ echo -e "One Two\nOne Two Three\nOne Two Three Four" | awk 'NR < 3'
```

执行上面的命令可以得到如下的结果：

```
One Two
One Two Three
```

FNR

该变量与 NR 类似，不过它是相对于当前文件而言的。此变量在处理多个文件输入时有重要的作用。每当从新的文件中读入时 FNR 都会被重新设置为 0。

OFMT

此变量表示数值输出的格式，它的默认值为 %.6g。

```
[jerry]$ awk 'BEGIN {print "OFMT = " OFMT}'
```

执行上面的命令可以得到如下的结果：

```
OFMT = %.6g
```

OFS

此变量表示输出域之间的分割符，其默认为空格。

```
[jerry]$ awk 'BEGIN {print "OFS = " OFS}' | cat -vte
```

执行上面的命令可以得到如下的结果：

```
OFS =  $
```

ORS

此变量表示输出记录（行）之间的分割符，其默认值是换行符。

```
[jerry]$ awk 'BEGIN {print "ORS = " ORS}' | cat -vte
```

执行上面的命令可以得到如下的结果：

```
ORS = $
$
```

RLENGTH

此变量表示 match 函数匹配的字符串长度。AWK 的 match 函数用于在输入的字符串中搜索指定字符串。

```
[jerry]$ awk 'BEGIN { if (match("One Two Three", "re")) { print RLENGTH } }'
```

执行上面的命令可以得到如下的结果：

```
2
```

RS

此变量表示输入记录的分割符，其默认值为换行符。

```
[jerry]$ awk 'BEGIN {print "RS = " RS}' | cat -vte
```

执行上面的命令可以得到如下的结果：

```
RS = $
$
```

RSTART

此变量表示由 match 函数匹配的字符串的第一个字符的位置。

```
[jerry]$ awk 'BEGIN { if (match("One Two Three", "Thre")) { print RSTART } }'
```

执行上面的命令可以得到如下的结果：

```
9
```

SUBSEP

此变量表示数组下标的分割行符，其默认值为 \034 。

```
[jerry]$ awk 'BEGIN { print "SUBSEP = " SUBSEP }' | cat -vte
```

执行上面的命令可以得到如下的结果：

```
SUBSEP = ^\$
```

$0

此变量表示整个输入记录。

```
[jerry]$ awk '{print $0}' marks.txt
```

执行上面的命令可以得到如下的结果：

```
1)  Amit    Physics   80
2)  Rahul   Maths     90
3)  Shyam   Biology   87
4)  Kedar   English   85
5)  Hari    History   89
```

$n

此变量表示当前输入记录的第 n 个域，这些域之间由 FS 分割。

```
[jerry]$ awk '{print $3 "\t" $4}' marks.txt
```

执行上面的命令可以得到如下的结果：

```
Physics   80
Maths     90
Biology   87
English   85
History   89
```

GNU AWK 特定的变量

下面将介绍 GNU AWK 专有的变量：

ARGIND

此变量表示当前文件中正在处理的 ARGV 数组的索引值。

```
[jerry]$ awk '{ print "ARGIND   = ", ARGIND; print "Filename = ", ARGV[ARGIND] }' junk1 junk2 junk3
```

执行上面的命令可以得到如下的结果：

```
ARGIND   = 1
Filename =  junk1
ARGIND   = 2
Filename =  junk2
ARGIND   = 3
Filename =  junk3
```

BINMODE

此变量用于在非 POSIX 系统上指定 AWK 对所有文件的 I/O 都使用二进制模式。 数值1，2 或者 3 分别指定输入文件、输出文件或所有文件。 字符串值 r 或 w 分别指定输入文件或者输出文件使用二进制 I/O模式。 字符串值 rw 或 wr 指定所有文件使用二进制 I/O模式。

ERRNO

此变量用于存储当 getline 重定向失败或者 close 函数调用失败时的失败信息。

```
[jerry]$ awk 'BEGIN { ret = getline < "junk.txt"; if (ret == -1) print "Error:", ERRNO }'
```

执行上面的命令可以得到如下的结果：

```
Error: No such file or directory
```

FIELDWIDTHS

该变量表示一个分割域之间的空格的宽度。当此变量被设置后， GAWK 将输入的域之间的宽度处理为固定宽度，而不是使用 FS 的值作为域间的分割符。

IGNORECASE

当此变量被设置后，GAWK将变得大小写不敏感。下面是一个简单的例子：

```
[jerry]$ awk 'BEGIN{IGNORECASE=1} /amit/' marks.txt
```

执行上面的命令得到如下的结果：

```
1)  Amit    Physics   80
```

LINT

此变量提供了在 GAWK 程序中动态控制 --lint 选项的一种途径。当这个变量被设置后， GAWK 会输出 lint 警告信息。如果给此变量赋予字符值 fatal，lint 的所有警告信息将会变了致命错误信息(fatal errors)输出，这和 --lint=fatal 效果一样。

```
[jerry]$ awk 'BEGIN {LINT=1; a}'
```

执行上面的命令可以得到如下的结果：

```
awk: cmd. line:1: warning: reference to uninitialized variable `a'
awk: cmd. line:1: warning: statement has no effect
```

PROCINFO

这是一个关联数组变量，它保存了进程相关的信息。比如， 真正的和有效的 UID 值，进程 ID 值等等。

```
[jerry]$ awk 'BEGIN { print PROCINFO["pid"] }'
```

执行上面的命令可以得到如下的结果：

```
4316
```

TEXTDOMAIN

此变量表示 AWK 程序当前文本域。 它主要用寻找程序中的字符串的本地翻译，用于程序的国际化。

```
[jerry]$ awk 'BEGIN { print TEXTDOMAIN }'
```

执行上面的命令可以得到如下的结果：

```
messages
```

（译注：输出 message 是由于 TEXTDOMAIN 的默认值为 messages ）上面所有的输出都是英文字符是因为本地语言环境配置为 en_IN 。

7

操作符

与其它编程语言一样，AWK 也提供了大量的操作符。这一章节中，我们将结合例子介绍 AWK 操作符的使用方法：

算术运算符

AWK 支持如下的算术运算符：

加法运算符

加法运算由符号 + 表示，它求得两个或者多个数字的和。下面是一个使用示例：

```
[jerry]$ awk 'BEGIN { a = 50; b = 20; print "(a + b) = ", (a + b) }'
```

执行上面的命令可以得到如下的结果：

```
(a + b) =  70
```

减法运算符

减法运算由符号 - 表示，它求得两个或者多个数值的差。示例如下：

```
[jerry]$ awk 'BEGIN { a = 50; b = 20; print "(a - b) = ", (a - b) }'
```

执行上面的命令可以得到如下的结果：

```
(a - b) =  30
```

乘法运算符

乘法运算由星号(*)表示，它求得两个或者多个数值的乘积。示例如下：

```
[jerry]$ awk 'BEGIN { a = 50; b = 20; print "(a * b) = ", (a * b) }'
```

执行上面的命令可以得到如下的结果：

```
(a * b) =  1000
```

除法运算符

除法运算由斜线(/) 表示，它求得两个或者两个以上数值的商。示例如下：

```
[jerry]$ awk 'BEGIN { a = 50; b = 20; print "(a / b) = ", (a / b) }'
```

执行上面的命令可以得到如下的结果：

```
(a / b) =  2.5
```

模运算符

模运算由百分（ % ）表示，它表示两个或者多个数进行模除运算得到其余数。下面是示例：

```
[jerry]$ awk 'BEGIN { a = 50; b = 20; print "(a % b) = ", (a % b) }'
```

执行上面的命令可以得到如下的结果：

```
(a % b) =  10
```

递增运算符与递减运算符

AWK 支持递增运算符与递减运算符：

前置递增运算

前置递增运算由 ++ 表示。它将操作数加 1。这个运算符将操作值增加 1，然后再返回增加后的值。下面的示例中，将操作数 a 值增加 1 后赋值给 b ， 最终 a 与 b 的值均为 11：

```
awk 'BEGIN { a = 10; b = ++a; printf "a = %d, b = %d\n", a, b }'
```

执行上面的命令可以得到如下的结果：

```
a = 11, b = 11
```

前置递减运算符

前置递减运算由 -- 表示。它的语义是将操作数减 1。这个运算符先将操作数的值减 1， 再将被减小后的值返回。下面的示例中将操作数 a 与 b 的值均设置为 9 :

```
[jerry]$ awk 'BEGIN { a = 10; b = --a; printf "a = %d, b = %d\n", a, b }'
```

执行上面的命令可以得到如下的结果：

```
a = 9, b = 9
```

后置递增运算符

后置递增运算由 ++ 表示。它同样将操作数的值加1。与前置递增运算符不同，它先将操作数的值返回，再将操作数的值加 1。下面的示例中会将操作数 a的值设置为10，b 的值设置为11。

```
[jerry]$ awk 'BEGIN { a = 10; b = a++; printf "a = %d, b = %d\n", a, b }'
```

执行上面的命令可以得到如下的结果：

```
a = 11, b = 10
```

后置递减运算符

后置递增运算符由 -- 表示。它同样将操作数的值减1。该操作符先将操作数的值返回，然后将操作数减 1。下面的示例中将操作数 a 的值设置为 9，b 的值设置为10。

```
[jerry]$ awk 'BEGIN { a = 10; b = a--; printf "a = %d, b = %d\n", a, b }'
```

执行上面的命令可以得到如下的结果：

```
a = 9, b = 10
```

赋值操作符

AWK 支持下面这些赋值操作：

简单赋值

简单赋值操作由 = 表示。示例如下：

```
[jerry]$ awk 'BEGIN { name = "Jerry"; print "My name is", name }'
```

执行上面的命令可以得到如下的结果：

```
My name is Jerry
```

加法赋值

加法赋值运算符为 +=。下面为示例：

```
[jerry]$ awk 'BEGIN { cnt=10; cnt += 10; print "Counter =", cnt }'
```

执行上面的命令可以得到如下的结果：

```
Counter = 20
```

上面的例子中，先给 cnt 变量赋值为 10。再使用加法赋值将 cnt 值增加 10。

减法赋值

减法赋值运算符为 −=。下面为示例：

```
[jerry]$ awk 'BEGIN { cnt=100; cnt −= 10; print "Counter =", cnt }'
```

执行上面的命令可以得到如下的结果：

```
Counter = 90
```

上面的例子中，先给 cnt 变量赋值为 100。再使用减法赋值运算将 cnt 值减少 10。

乘法赋值

乘法赋值运算符为 *=。下面为示例：

```
[jerry]$ awk 'BEGIN { cnt=10; cnt *= 10; print "Counter =", cnt }'
```

执行上面的命令可以得到如下的结果：

```
Counter = 100
```

上面的例子中，先给 cnt 变量赋值为 10。再使用乘法赋值运算符将 cnt 值乘以 10。

除法赋值

除法赋值运算符为 /=。下面为示例：

```
[jerry]$ awk 'BEGIN { cnt=100; cnt /= 5; print "Counter =", cnt }'
```

执行上面的命令可以得到如下的结果：

```
Counter = 20
```

上面的例子中，先将 cnt 变量赋值为 100。再使用乘法赋值运算符将 cnt 值除以 5。

模运算赋值

模运算赋值运算符为 %=。下面为示例：

```
[jerry]$ awk 'BEGIN { cnt=100; cnt %= 8; print "Counter =", cnt }'
```

执行上面的命令可以得到如下的结果：

```
Counter = 4
```

上面的例子中，先将 cnt 变量赋值为 10。再使用模运算赋值操作将 cnt 值乘以 10。

指数赋值

指数赋值运算符为 ^=。下面为示例：

```
[jerry]$ awk 'BEGIN { cnt=2; cnt ^= 4; print "Counter =", cnt }'
```

执行上面的命令可以得到如下的结果：

```
Counter = 16
```

这个例子求　cnt 的四次幂。

关系运算符

AWK 支持如下关系运算符：

等于

等于运算符为 ==。如果两个操作数相等则返回真，否则返回假。示例如下：

```
awk 'BEGIN { a = 10; b = 10; if (a == b) print "a == b" }'
```

执行上面的命令可以得到如下的结果：

```
a == b
```

不等于

不等于运算符为 !=。如果两个操作数相等则返回假，否则返回真。示例如下：

```
[jerry]$ awk 'BEGIN { a = 10; b = 20; if (a != b) print "a != b" }'
```

执行上面的命令可以得到如下的结果：

```
a != b
```

小于

小于运算符为 <。如果左操作数小于右操作数据则返回真，否则返回假。示例如下：

```
[jerry]$ awk 'BEGIN { a = 10; b = 20; if (a < b) print "a < b" }'
```

执行上面的命令可以得到如下的结果：

```
a < b
```

小于或等于

小于等于运算符为 <=。如果左操作数小于或等于右操作数据则返回真，否则返回假。示例如下：

```
[jerry]$ awk 'BEGIN { a = 10; b = 10; if (a <= b) print "a <= b" }'
```

执行上面的命令可以得到如下的结果：

```
a <= b
```

大于

大于运算符为 >。如果左操作数大于右操作数则返回真，否则返回假。示例如下：

```
[jerry]$ awk 'BEGIN { a = 10; b = 20; if (b > a ) print "b > a" }'
```

执行上面的命令可以得到如下的结果：

```
b > a
```

大于或等于

大于等于运算符为 >=。如果左操作数大于或等于右操作数则返回真，否则返回假。示例如下：

```
[jerry]$ awk 'BEGIN { a = 10; b = 10; if (a >= b) print "a >= b" }'
```

执行上面的命令可以得到如下的结果：

```
b >= a
```

逻辑运算符

AWK 包括如下逻辑运算符：

逻辑与

逻辑与运算符为 &&。下面是逻辑与运算符的语法：

```
expr1 && expr2
```

如果 expr1 与 epxr2 均为真，则最终结果为真；否则为假。请注意，只有当 expr1 为真时才会计算 expr2 的值，若 expr1 为假则直接返回真，而不再计算 expr2 的值。下面的例子判断给定的字符串是否是十进制形式：

```
[jerry]$ awk 'BEGIN {num = 5; if (num >= 0 && num <= 7) printf "%d is in octal format\n", num }'
```

执行上面的命令可以得到如下的结果：

```
5 is in octal format
```

逻辑或

逻辑或运算符为 ||。该运算符语法如下：

```
expr1 || expr2
```

如果 expr1 与 epxr2 至少其中一个为真，则最终结果为真；二者均为假时则为假。请注意，只有当 expr1 为假时才会计算 expr2 的值，若 expr1 为真则不会再计算 expr2 的值。示例如下：

```
[jerry]$ awk 'BEGIN {ch = "\n"; if (ch == " " || ch == "\t" || ch == "\n") print "Current character is whitespace." }'
```

执行上面的命令可以得到如下的结果：

```
Current character is whitespace.
```

逻辑非

逻辑非运算为感叹号(!)。此运算符语法如下：

```
! expr1
```

逻辑非将 expr1 的真值取反。如果 expr1 为真，则返回 0。否则返回 1。下面的示例判断字符串是否为空：

```
[jerry]$ awk 'BEGIN { name = ""; if (! length(name)) print "name is empty string." }'
```

执行上面的命令可以得到如下的结果：

```
name is empty string.
```

三元运算符

我们可以使用三元运算符来实现条件表达式。下面为其语法：

```
condition expression ? statement1 : statement2
```

当条件表达式（ condition expression ）为真时，statement1 执行，否则 statement2 执行。下面的示例将返回最大数值：

```
[jerry]$ awk 'BEGIN { a = 10; b = 20; (a > b) ? max = a : max = b; print "Max =", max}'
```

执行上面的命令可以得到如下的结果：

```
Max = 20
```

一元运算符

AWK 支持如下几种一元运算符：

一元加运算

一元加运算符表示为 +。它将操作数乘以 +1。

```
[jerry]$ awk 'BEGIN { a = -10; a = +a; print "a =", a }'
```

执行上面的命令可以得到如下的结果：

```
a = -10
```

一元减运算符

一元减运算符为 - 。它表示将操作数乘以 -1。

```
[jerry]$ awk 'BEGIN { a = -10; a = -a; print "a =", a }'
```

执行上面的命令可以得到如下的结果：

```
a = 10
```

指数运算符

下面将介绍两种形式的指数运算符：

幂运算符 ^

^ 运算符对操作数执行幂运算。下面的示例求 10 的二次幂。

```
[jerry]$ awk 'BEGIN { a = 10; a = a ^ 2; print "a =", a }'
```

执行上面的命令可以得到如下的结果：

```
a = 100
```

**幂运算符 **

** 运算符对操作数执行幂运算。下面的示例求 10 的二次幂。

```
[jerry]$ awk 'BEGIN { a = 10; a = a ** 2; print "a =", a }'
```

执行上面的命令可以得到如下的结果：

```
a = 100
```

字符串连接操作符

空格 (space) 操作符可以完成两个字符串的连接操作。示例如下：

```
[jerry]$ awk 'BEGIN { str1="Hello, "; str2="World"; str3 = str1 str2; print str3 }'
```

执行上面的命令可以得到如下的结果：

```
Hello, World
```

数组成员操作符

数组成员操作符为 in。该操作符用于访问数组元素 。下面的示例用于此操作符输出数组中所有元素。

```
[jerry]$ awk 'BEGIN { arr[0] = 1; arr[1] = 2; arr[2] = 3; for (i in arr) printf "arr[%d] = %d\n", i, arr[i] }'
```

执行上面的命令可以得到如下的结果：

```
arr[0] = 1
arr[1] = 2
arr[2] = 3
```

正则表达式操作符

下面将介绍两种正则表达式操作符:

匹配（Match）

匹配运算符为 ~。它用于搜索包含匹配模式字符串的域。下面的示例中将输出包括 9 的行：

```
[jerry]$ awk '$0 ~ 9' marks.txt
```

执行上面的命令可以得到如下的结果：

```
2)  Rahul   Maths   90
5)  Hari    History   89
```

不匹配(Not match)

不匹配操作符为 !~。此操作符用于搜索不匹配指定字符串的域。如下示例输出不包含 9 的行：

```
[jerry]$ awk '$0 !~ 9' marks.txt
```

执行上面的命令可以得到如下的结果：

```
1)  Amit   Physics   80
3)  Shyam  Biology   87
4)  Kedar  English   85
```

8

正则表达式

AWK 可以方便高效地处理正则表达式。大量复杂的任务都可以由极其简单的正则表达式来解决。每一个精通命令行的人都知道正则表达式真正的威力所在。

这一章将着重讲解标准正则表达式的使用方法。

点（Dot）

点字符（.）可以匹配除了行结束字符的所有字符。比如下面的便子就可以匹配 fin, fun, fan 等等。

```
[jerry]$ echo -e "cat\nbat\nfun\nfin\nfan" | awk '/f.n/'
```

执行上面命令可以得到如下结果：

```
fun
fin
fan
```

行开始

行开始符(^)匹配一行的开始。下面的示例将输出所有以字符串 The 开始的行。

```
[jerry]$ echo -e "This\nThat\nThere\nTheir\nthese" | awk '/^The/'
```

执行上面的命令可以得到如下结果：

```
There
Their
```

行结束

行结束符($)匹配一行的结束。下面的例子中将输出所有以字符 n 结束的行：

```
[jerry]$ echo -e "knife\nknow\nfun\nfin\nfan\nnine" | awk '/n$/'
```

执行上面的命令可以得到如下结果：

```
fun
fin
fan
```

匹配字符集

匹配字符集用于匹配集合（由方括号表示）中的一个字符。如下例子中，匹配 Call 与 Tall 而不会匹配 Ball。

```
[jerry]$ echo -e "Call\nTall\nBall" | awk '/[CT]all/'
```

执行上面的命令可以得到如下结果：

```
fun
fin
fan
```

排除集

正则匹配时会排除集合中的字符。如下例子中只会输出 Ball。

```
[jerry]$ echo -e "Call\nTall\nBall" | awk '/[^CT]all/'
```

执行上面的命令可以得到如下结果：

```
Ball
```

或

竖线(|)允许正则表达式实现逻辑或运算. 下面例子将会输出 Ball 与 Call 。

```
[jerry]$ echo -e "Call\nTall\nBall\nSmall\nShall" | awk '/Call|Ball/'
```

执行上面的命令可以得到如下结果：

```
Call
Ball
```

最多出现一次

该符号(？)前面的字符不出现或者出现一次。如下示例匹配 Colour 与 Color。 使用？使得 u 变成了可选字符 。

```
[jerry]$ echo -e "Colour\nColor" | awk '/Colou?r/'
```

执行上面的命令可以得到如下结果：

```
Colour
Color
```

出现零次或多次

该符号(*) 允许其前的字符出现多次或者不出现。如下示例将匹配 ca，cat, catt 等等。

```
[jerry]$ echo -e "ca\ncat\ncatt" | awk '/cat*/'
```

执行上面的命令可以得到如下结果：

```
ca
cat
catt
```

出现一次或多次

该符号(+)使得其前的字符出现一次或者多次。下面的例子会匹配一个 2 或者多个连续的 2。

```
[jerry]$ echo -e "111\n22\n123\n234\n456\n222"  | awk '/2+/'
```

执行上面的命令可以得到如下结果：

```
22
123
234
222
```

分组

括号用于分组而字符 | 用于提供多种选择。如下的正则表达式会匹配所有包含 Apple Juice 或 Aplle Cake 的行。

```
[jerry]$ echo -e "Apple Juice\nApple Pie\nApple Tart\nApple Cake" | awk '/Apple (Juice|Cake)/'
```

执行上面的命令可以得到如下结果：

```
Apple Juice
Apple Cake
```

9

数组

AWK 有关联数组这种数据结构，而这种数据结构最好的一个特点就是它的索引值不需要是连续的整数值。我们既可以使用数字也可以使用字符串作为数组的索引。除此之外，关联数组也不需要提前声明其大小，因为它在运行时可以自动的增大或减小。这一章节中将会讲解 AWK 数组的使用方法。

如下为数组使用的语法格式：

```
array_name[index]=value
```

其中 array_name 是数组的名称，index 是数组索引，value 为数组中元素所赋予的值。

创建数组

为了进一步了解数组，我们先来看一下如何创建数组以及如何访问数组元素：

```
[jerry]$ awk 'BEGIN {
fruits["mango"]="yellow";
fruits["orange"]="orange"
print fruits["orange"] "\n" fruits["mango"]
}'
```

执行上面的命令可以得到如下的结果：

```
orange
yellow
```

在上面的例子中，我们定义了一个水果(fruits)数组，该数组的索引为水果名称，值为水果的颜色。可以使用如下格式访问数组元素：

```
array_name[index]
```

删除数组元素

插入元素时我们使用赋值操作符。删除数组元素时，我们则使用 delete 语句。如下所示：

```
delete array_name[index]
```

下面的例子中，数组中的 orange 元素被删除（删除命令没有输出）：

```
[jerry]$ awk 'BEGIN {
fruits["mango"]="yellow";
fruits["orange"]="orange";
delete fruits["orange"];
```

```
print fruits["orange"]
}'
```

多维数组

AWK 本身只支持多维数组，不过我们可以很容易地使用一维数组模拟实现多维数组。

如下示例为一个 3x3 的三维数组：

```
100 200 300
400 500 600
700 800 900
```

上面的示例中，array[0][0] 存储 100，array[0][1] 存储 200 ，依次类推。为了在 array[0][0] 处存储100, 我们可以使用如下语法：

```
array["0,0"] = 100
```

尽管在示例中，我们使用了 0,0 作为索引，但是这并不是两个索引值。事实上，它是一个字符串索引 0,0。

下面是模拟二维数组的例子：

```
[jerry]$ awk 'BEGIN {
array["0,0"] = 100;
array["0,1"] = 200;
array["0,2"] = 300;
array["1,0"] = 400;
array["1,1"] = 500;
array["1,2"] = 600;
 # print array elements
print "array[0,0] = " array["0,0"];
print "array[0,1] = " array["0,1"];
print "array[0,2] = " array["0,2"];
print "array[1,0] = " array["1,0"];
print "array[1,1] = " array["1,1"];
print "array[1,2] = " array["1,2"];
}'
```

执行上面的命令可以得到如下结果：

```
array[0,0] = 100
array[0,1] = 200
array[0,2] = 300
array[1,0] = 400
```

```
array[1,1] = 500
array[1,2] = 600
```

在数组上可以执行很多操作，比如，使用 asort 完成数组元素的排序，或者使用 asorti 实现数组索引的排序等等。我们会在后面的章节中介绍可以对数组进行操作的函数。

10

控制流

与其实的编程语言一样，AWK 同样提供了条件语句控制程序的执行流程。这一章中我们会介绍 AWK 中条件语句的使用方法。

IF 语句

条件语句测试条件然后根据条件选择执行相应的动作。下面是条件语句的语法：

```
if (condition)
   action
```

也可以使用花括号来执行一组操作：

```
if (condition)
{
   action-1
   action-1
   .
   .
   action-n
}
```

下面的例子判断数字是奇数还是偶数：

```
[jerry]$ awk 'BEGIN {num = 10; if (num % 2 == 0) printf "%d is even number.\n", num }'
```

执行上面的命令可以得到如下的结果：

```
10 is even number.
```

IF - ELSE 语句

if-else语句中允许在条件为假时执行另外一组的动作。下面为 if-else 的语法格式：

```
if (condition)
   action-1
else
   action-2
```

其中，条件为真时执行 action-1，条件为假时执行 action-2。下面是使用该语句判断数字是否为偶数的例子：

```
[jerry]$ awk 'BEGIN {num = 11;
  if (num % 2 == 0) printf "%d is even number.\n", num;
```

```
  else printf "%d is odd number.\n", num
                }'
```

执行上面的操作可以得到如下的结果：

```
11 is odd number.
```

if-else-if 梯

我们可以很轻松地使用多个 if-else 语句构造 if-else-if 梯从而实现多个条件的判断。示例如下：

```
[jerry]$ awk 'BEGIN {
a=30;
if (a==10)
  print "a = 10";
else if (a == 20)
  print "a = 20";
else if (a == 30)
  print "a = 30";
}'
```

执行上面的命令可以得到如下的结果：

```
a = 30
```

11

循环

除了前面介绍的条件语句，AWK 还提供了循环语句。该语句的作用就是当条件为真时重复执行一系列的命令。本章将讲解 AWK 中循环语句的使用方法。

For

For 循环的语法如下：

```
for (initialisation; condition; increment/decrement)
   action
```

for 语句首先执行初始化动作(initialisation)，然后再检查条件(condition)。如果条件为真，则执行动作(action)，然后执行递增(increment)或者递减(decrement)操作。只要条件为真循环就会一直执行。每次循环结束都会进条件检查，若条件为假则结束　循环。下面的例子使用 For 循环输出数字 1 至 5：

```
[jerry]$ awk 'BEGIN { for (i = 1; i <= 5; ++i) print i }'
```

执行上面的命令可以得到如下结果：

```
1
2
3
4
5
```

While

While 循环会一直执行动作直到逻辑条件为假为止。其使用方法如下：

```
while (condition)
   action
```

AWK 首先检查条件是否为真，若条件为真则执行动作。此过程一直重复直到条件为假时，则停止。下面是使用 While 循环输出数字 1 到 5 的例子：

```
[jerry]$ awk 'BEGIN {i = 1; while (i < 6) { print i; ++i } }'
```

执行上面的命令可以得到如下的结果：

```
1
2
3
```

```
4
5
```

Do-While

Do-While 循环与 While 循环相似，但是 Do-While 的条件测试放到了循环的尾部。下面是 do-while 的语法：

```
do
   action
while (condition)
```

在 do-while 循环中，无论条件是真是假，循环语句至少执行一次，执行后检查条件真假。下面是使用 do-While 循环输出数字 1 到 5 的例子：

```
[jerry]$ awk 'BEGIN {i = 1; do { print i; ++i } while (i < 6) }'
```

执行上面的命令可以得到如下的结果：

```
1
2
3
4
5
```

Break

顾名思义，break 用以结束循环过程。在下面的示例子中，当计算的和大于 50 的时候使用 break 结束循环过程：

```
[jerry]$ awk 'BEGIN {
   sum = 0; for (i = 0; i < 20; ++i) {
      sum += i; if (sum > 50) break; else print "Sum =", sum
                              }
            }'
```

执行上面的命令可以得到如下的结果：

```
Sum = 0
Sum = 1
Sum = 3
Sum = 6
Sum = 10
Sum = 15
```

```
Sum = 21
Sum = 28
Sum = 36
Sum = 45
```

Continue

Continue 语句用于在循环体内部结束本次循环，从而直接进入下一次循环迭代。当我们希望跳过循环中某处数据处理时就会用到 Continue。下面的例子输出 1 到 20 之间的偶数：

```
[jerry]$ awk 'BEGIN {for (i = 1; i <= 20; ++i) {if (i % 2 == 0) print i ; else continue} }'
```

执行上面的命令可以得到如下的结果：

```
2
4
6
8
10
12
14
16
18
20
```

Exit

Exit 用于结束脚本程序的执行。该函数接受一个整数作为参数表示 AWK 进程结束状态。 如果没有提供该参数，其默认状态为 0 。下面例子中当和大于 50 时结束 AWK 程序。

```
[jerry]$ awk 'BEGIN {
   sum = 0; for (i = 0; i < 20; ++i) {
      sum += i; if (sum > 50) exit(10); else print "Sum =", sum
                                }
             }'
```

执行上面的命令可以得到如下的结果：

```
Sum = 0
Sum = 1
Sum = 3
Sum = 6
Sum = 10
```

```
Sum = 15
Sum = 21
Sum = 28
Sum = 36
Sum = 45
```

让我们检查一下脚本执行后的返回状态：

```
[jerry]$ echo $?
```

执行上面的命令可以得到如下的结果：

```
19
```

12

内置函数

AWK 为程序开发者提供了丰富的内置函数。这一章节会讲解 AWK 提供的算术函数、字符串操作函数、时间操作相关的函数、位操作函数以及其它各种各样的函数。

算术函数

AWK 提供了如下的内置算术运算函数：

atan2(y,x)

该函数返回正切值 y/x 的角度值，角度以弧度为单位。示例如下：

```
[jerry]$ awk 'BEGIN {
  PI = 3.14159265
  x = -10
  y = 10
  result = atan2 (y,x) * 180 / PI;

  printf "The arc tangent for (x=%f, y=%f) is %f degrees\n", x, y, result
}'
```

执行上面的命令得到如下结果：

```
The arc tangent for (x=-10.000000, y=10.000000) is 135.000000 degrees
```

cos(expr)

该函数返回 expr 的余弦值， 输入参数以弧度为单位。示例如下：

```
[jerry]$ awk 'BEGIN {
  PI = 3.14159265
  param = 60
  result = cos(param * PI / 180.0);

  printf "The cosine of %f degrees is %f.\n", param, result
}'
```

执行上面的命令得到如下的结果：

```
The cosine of 60.000000 degrees is 0.500000.
```

exp(expr)

此函数返回自然数 e 的 expr 次幂。

```
[jerry]$ awk 'BEGIN {
  param = 5
  result = exp(param);

  printf "The exponential value of %f is %f.\n", param, result
}'
```

执行上面的命令可以得到如下的结果：

```
The exponential value of 5.000000 is 148.413159.
```

int(expr)

此函数返回数值 expr 的整数部分。示例如下：

```
[jerry]$ awk 'BEGIN {
  param = 5.12345
  result = int(param)

  print "Truncated value =", result
}'
```

执行上面的命令可以得到如下的结果：

```
Truncated value = 5
```

log(expr)

此函数计算 expr 自然对数。

```
[jerry]$ awk 'BEGIN {
  param = 5.5
  result = log (param)

  printf "log(%f) = %f\n", param, result
}'
```

执行上面的命令可以得到如下的结果：

```
log(5.500000) = 1.704748
```

rand

rand 函数返回一个大于等于 0 小于 1 的随机数 N（0<= N < 1）。示例如下：

```
[jerry]$ awk 'BEGIN {
  print "Random num1 =" , rand()
  print "Random num2 =" , rand()
  print "Random num3 =" , rand()
}'
```

执行上面的命令可以得到如下的结果：

```
Random num1 = 0.237788
Random num2 = 0.291066
Random num3 = 0.845814
```

sin(expr)

正弦函数返回角度 expr 的正弦值，角度以弧度为单位。示例如下：

```
[jerry]$ awk 'BEGIN {
  PI = 3.14159265
  param = 30.0
  result = sin(param * PI /180)

  printf "The sine of %f degrees is %f.\n", param, result
}'
```

执行上面的命令可以得到如下的结果：

```
The sine of 30.000000 degrees is 0.500000.
```

sqrt(expr)

此函数计算 expr 的平方根。

```
[jerry]$ awk 'BEGIN {
  param = 1024.0
  result = sqrt(param)
```

```
  printf "sqrt(%f) = %f\n", param, result
}'
```

执行上面的命令可以得到如下的结果：

```
sqrt(1024.000000) = 32.000000
```

srand([expr])

此函数使用种子值生成随机数，数值　expr 作为随机数生成器的种子值。如果没有指定 expr 的值则函数默认使用当前系统时间作为种子值。

```
[jerry]$ awk 'BEGIN {
  param = 10

  printf "srand() = %d\n", srand()
  printf "srand(%d) = %d\n", param, srand(param)
}'
```

执行上面的命令得到如下的结果：

```
srand() = 1
srand(10) = 1417959587
```

字符串函数

AWK 提供了下面所示的字符串操作函数：

asort(arr,[, d [,how]])

asort 函数使用 GAWK 值比较的一般规则排序 arr 中的内容，然后用以 1 开始的有序整数替换排序内容的索引。

```
[jerry]$ awk 'BEGIN {
    arr[0] = "Three"
    arr[1] = "One"
    arr[2] = "Two"

    print "Array elements before sorting:"
    for (i in arr) {
        print arr[i]
    }
```

```
    asort(arr)

    print "Array elements after sorting:"
    for (i in arr) {
        print arr[i]
    }
}'
```

执行上面的命令可以得到如下的结果：

```
Array elements before sorting:
Three
One
Two
Array elements after sorting:
One
Three
Two
```

asorti(arr,[, d [,how]])

asorti 函数的行为与 asort 函数的行为很相似，二者的差别在于 aosrt 对数组的值排序，而 asorti 对数组的索引排序。

```
[jerry]$ awk 'BEGIN {
    arr["Two"] = 1
    arr["One"] = 2
    arr["Three"] = 3

    asorti(arr)

    print "Array indices after sorting:"
    for (i in arr) {
        print arr[i]
    }
}'
```

执行上面的命令可以得到如下的结果：

```
Array indices after sorting:
One
Three
Two
```

gsub(regx,sub, string)

gsub 是全局替换(global substitution)的缩写。它将出现的子串（ sub ）替换为 regx。第三个参数 string 是可选的，默认值为 $0，表示在整个输入记录中搜索子串。

```
[jerry]$ awk 'BEGIN {
    str = "Hello, World"

    print "String before replacement = " str

    gsub("World", "Jerry", str)

    print "String after replacement = " str
}'
```

执行上面的命令可以得到如下的结果：

```
String before replacement = Hello, World
String after replacement = Hello, Jerry
```

index(str,sub)

index 函数用于检测字符串 sub 是否是 str 的子串。如果 sub 是 str 的子串，则返回子串 sub 在字符串 str 的开始位置；若不是其子串，则返回 0。str 的字符位置索引从 1 开始计数。

```
[jerry]$ awk 'BEGIN {
    str = "One Two Three"
    subs = "Two"

    ret = index(str, subs)

    printf "Substring \"%s\" found at %d location.\n", subs, ret
}'
```

执行上面的命令可以得到如下的结果：

```
Substring "Two" found at 5 location.
```

length(str)

length 函数返回字符串的长度。

```
[jerry]$ awk 'BEGIN {
   str = "Hello, World !!!"

   print "Length = ", length(str)
}'
```

执行上面的命令可以得到如下的结果：

```
Length = 16
```

match(str, regex)

match 返回正则表达式在字符串 str 中第一个最长匹配的位置。如果匹配失败则返回0。

```
[jerry]$ awk 'BEGIN {
   str = "One Two Three"
   subs = "Two"

   ret = match(str, subs)

   printf "Substring \"%s\" found at %d location.\n", subs, ret
}'
```

执行上面的命令可以得到如下的结果：

```
Substring "Two" found at 5 location.
```

split(str, arr,regex)

split 函数使用正则表达式 regex 分割字符串 str。分割后的所有结果存储在数组 arr 中。如果没有指定 regex 则使用 FS 切分。

```
[jerry]$ awk 'BEGIN {
   str = "One,Two,Three,Four"

   split(str, arr, ",")

   print "Array contains following values"

   for (i in arr) {
      print arr[i]
   }
}'
```

执行上面的命令可以得到如下的结果：

```
Array contains following values
One
Two
Three
Four
```

sprintf(format,expr-list)

sprintf 函数按指定的格式（ format ）将参数列表 expr-list 构造成字符串然后返回。

```
[jerry]$ awk 'BEGIN {
   str = sprintf("%s", "Hello, World !!!")

   print str
}'
```

执行上面的命令可以得到如下的结果：

```
Hello, World !!!
```

strtonum(str)

strtonum 将字符串 str 转换为数值。 如果字符串以 0 开始，则将其当作十进制数；如果字符串以 0x 或 0X 开始，则将其当作十六进制数；否则，将其当作浮点数。

```
[jerry]$ awk 'BEGIN {
   print "Decimal num = " strtonum("123")
   print "Octal num = " strtonum("0123")
   print "Hexadecimal num = " strtonum("0x123")
}'
```

执行上面的命令可以得到如下的结果：

```
Decimal num = 123
Octal num = 83
Hexadecimal num = 291
```

sub(regex,sub,string)

sub 函数执行一次子串替换。它将第一次出现的子串用 regex 替换。第三个参数是可选的，默认为 $0。

```
[jerry]$ awk 'BEGIN {
   str = "Hello, World"

   print "String before replacement = " str

   sub("World", "Jerry", str)

   print "String after replacement = " str
}'
```

执行上面的命令可以得到如下的结果：

```
String before replacement = Hello, World
String after replacement = Hello, Jerry
```

substr(str, start, l)

substr 函数返回 str 字符串中从第 start 个字符开始长度为 l 的子串。如果没有指定 l 的值，返回 str 从第 start 个字符开始的后缀子串。

```
[jerry]$ awk 'BEGIN {
   str = "Hello, World !!!"
   subs = substr(str, 1, 5)

   print "Substring = " subs
}'
```

执行上面的命令可以得到如下的结果：

```
Substring = Hello
```

tolower(str)

此函数将字符串 str 中所有大写字母转换为小写字母然后返回。注意，字符串 str 本身并不被改变。

```
[jerry]$ awk 'BEGIN {
   str = "HELLO, WORLD !!!"

   print "Lowercase string = " tolower(str)
}'
```

执行上面的命令可以得到如下的结果：

```
Lowercase string = hello, world !!!
```

toupper(str)

此函数将字符串 str 中所有小写字母转换为大写字母然后返回。注意，字符串 str 本身不被改变。

```
[jerry]$ awk 'BEGIN {
   str = "hello, world !!!"

   print "Uppercase string = " toupper(str)
}'
```

执行上面命令可以得到如下的结果：

```
Uppercase string = HELLO, WORLD !!!
```

时间函数

AWK 提供了如下的内置时间函数：

systime

此函数返回从 Epoch 以来到当前时间的秒数（在 POSIX 系统上，Epoch 为1970-01-01 00:00:00 UTC）。

```
[jerry]$ awk 'BEGIN {
   print "Number of seconds since the Epoch = " systime()
}'
```

执行上面的命令可以得到如下的结果：

```
Number of seconds since the Epoch = 1418574432
```

mktime(dataspec)

此函数将字符串 dataspec 转换为与 systime 返回值相似的时间戳。 dataspec 字符串的格式为 YYYY MM D D HH MM SS。

```
[jerry]$ awk 'BEGIN {
   print "Number of seconds since the Epoch = " mktime("2014 12 14 30 20 10")
}'
```

执行上面的命令可以得到如下的结果：

```
Number of seconds since the Epoch = 1418604610
```

strftime([format [, timestamp[, utc-flag]]])

此函数根据 format 指定的格式将时间戳 timestamp 格式化。

```
[jerry]$ awk 'BEGIN {
   print strftime("Time = %m/%d/%Y %H:%M:%S", systime())
}'
```

执行上面的的命令可以得到如下的结果：

```
Time = 12/14/2014 22:08:42
```

下面是 AWK 支持的不同的日期格式说明：

SN	描述
%a	星期缩写(Mon-Sun)。
%A	星期全称（Monday-Sunday）。
%b	月份缩写（Jan）。
%B	月份全称（January）。
%c	本地日期与时间。
%C	年份中的世纪部分，其值为年份整除100。
%d	十进制日期(01-31)
%D	等价于 %m/%d/%y.
%e	日期，如果只有一位数字则用空格补齐
%F	等价于 %Y-%m-%d，这也是 ISO 8601 标准日期格式。
%g	ISO8610 标准周所在的年份模除 100（00-99)。比如，1993 年 1 月 1 日属于 1992 年的第 53 周。所以，虽然它是 1993 年第 1 天，但是其 ISO8601 标准周所在年份却是 1992。同样，尽管 1973 年 12 月 31 日属于 1973 年但是它却属于 1994 年的第一周。所以 1973 年 12 月 31 日的 ISO8610 标准周所在的年是 1974 而不是 1973。
%G	ISO 标准周所在年份的全称。
%h	等价于 %b.
%H	用十进制表示的 24 小时格式的小时(00-23)
%I	用十进制表示的 12 小时格式的小时（00-12）
%j	一年中的第几天（001-366）
%m	月份（01-12）
%M	分钟数（00-59)
%n	换行符 (ASCII LF)
%p	十二进制表示法（AM/PM）
%r	十二进制表示法的时间（等价于 %I:%M:%S %p）。

SN	描述
%R	等价于 %H:%M。
%S	时间的秒数值（00-60）
%t	制表符 (tab)
%T	等价于 %H:%M:%S。
%u	以数字表示的星期(1-7),1 表示星期一。
%U	一年中的第几个星期（第一个星期天作为第一周的开始），00-53
%V	一年中的第几个星期（第一个星期一作为第一周的开始），01-53。
%w	以数字表示的星期（0-6），0表示星期日 。
%W	十进制表示的一年中的第几个星期（第一个星期一作为第一周的开始），00-53。
%x	本地日期表示
%X	本地时间表示
%y	年份模除 100。
%Y	十进制表示的完整年份。
%z	时区，表示格式为+HHMM（例如，格式要求生成的 RFC 822或者 RFC 1036 时间头）
%Z	时区名称或缩写，如果时区待定则无输出。

位操作函数

AWK 提供了如下的内置的位操作函数：

and

执行位与操作。

```
[jerry]$ awk 'BEGIN {
   num1 = 10
   num2 = 6

   printf "(%d AND %d) = %d\n", num1, num2, and(num1, num2)
}'
```

执行上面的命令可以得到如下的结果：

```
(10 AND 6) = 2
```

compl

按位求补。

```
[jerry]$ awk 'BEGIN {
    num1 = 10

    printf "compl(%d) = %d\n", num1, compl(num1)
}'
```

执行上面的命令可以得到如下的结果：

```
compl(10) = 9007199254740981
```

lshift

左移位操作。

```
[jerry]$ awk 'BEGIN {
    num1 = 10

    printf "lshift(%d) by 1 = %d\n", num1, lshift(num1, 1)
}'
```

执行上面的命令可以得到如下的结果：

```
lshift(10) by 1 = 20
```

rshift

向右移位操作。

```
[jerry]$ awk 'BEGIN {
    num1 = 10

    printf "rshift(%d) by 1 = %d\n", num1, rshift(num1, 1)
}'
```

执行上面的命令可以得到如下的结果：

```
rshift(10) by 1 = 5
```

or

按位或操作。

```
[jerry]$ awk 'BEGIN {
    num1 = 10
    num2 = 6

    printf "(%d OR %d) = %d\n", num1, num2, or(num1, num2)
}'
```

执行上面的命令可以得到如下的结果：

```
(10 OR 6) = 14
```

xor

按位异或操作。

```
[jerry]$ awk 'BEGIN {
    num1 = 10
    num2 = 6

    printf "(%d XOR %d) = %d\n", num1, num2, xor(num1, num2)
}'
```

执行上面的命令可以得到如下的结果：

```
(10 bitwise xor 6) = 12
```

其它函数

其它函数中主要包括:

close(expr)

关闭管道的文件。

```
[jerry]$ awk 'BEGIN {
    cmd = "tr [a-z] [A-Z]"
    print "hello, world !!!" |& cmd
    close(cmd, "to")
    cmd |& getline out
    print out;
    close(cmd);
}'
```

执行上面的命令可以得到如下的结果：

```
HELLO, WORLD !!!
```

脚本的内容看上去很神秘吗？让我们来揭开它神秘的面纱。

- 第一条语句 cmd = "tr [a-z] [A-Z]" 在　AWK 中建立了一个双向的通信通道。
- 第二条语句 print 为 tr 命令提供输入。&| 表示双向通信。
- 第三条语句 close(cmd, "to") 完成执行后关闭 to 进程。
- 第四条语句 cmd |& getline out 使用 getline 函数将输出存储到 out 变量中。
- 接下来的输出语句打印输出的内容，最后 close 函数关闭 cmd。

delete

delete 被用于从数组中删除元素。下面的例子演示了如何使用 delete：

```
[jerry]$ awk 'BEGIN {
   arr[0] = "One"
   arr[1] = "Two"
   arr[2] = "Three"
   arr[3] = "Four"

   print "Array elements before delete operation:"
   for (i in arr) {
      print arr[i]
   }

   delete arr[0]
   delete arr[1]

   print "Array elements after delete operation:"
   for (i in arr) {
      print arr[i]
   }
}'
```

执行上面的命令可以得到如下的结果：

```
Array elements before delete operation:
One
Two
Three
```

```
Four

Array elements after delete operation:
Three
Four
```

exit

该函数终止脚本执行。它可以接受可选的参数 expr 传递 AWK 返回状态。示例如下：

```
[jerry]$ awk 'BEGIN {
   print "Hello, World !!!"

   exit 10

   print "AWK never executes this statement."
}'
```

执行上面的命令可以得到如下的结果：

```
Hello, World !!!
```

flush

flush 函数用于刷新打开文件或管道的缓冲区。 使用方法如下：

```
fflush([output-expr])
```

如果没有提供 output-expr，fflush 将刷新标准输出。若 output-epxr 是空字符串 (""), fflush 将刷新所有打开的文件和管道。

getline

getline 函数读入下一行。示例中使用 getline 从文件 marks.txt 中读入一行并输出：

```
[jerry]$ awk '{getline; print $0}' marks.txt
```

执行上面的命令可以得到如下的结果：

```
2)  Rahul   Maths   90
4)  Kedar   English   85
5)  Hari    History   89
```

脚本看似工作正常，但是第一行去哪儿了呢？让我们理一下整个过程。刚启动时，AWK 从文件 marks.txt 中读入一行存储到变量 $0 中。在下一条语句中，我们使用 getline 读入下一行。 因此 AWK 读入第二行并存储到 $0 中。最后，AWK 使用 print 输出第二行的内容。这个过程一直到文件结束。

next

next 停止处理当前记录，并且进入到下一条记录的处理过程。下面的例子中，当模式串匹配成功后程序并不执行任何操作：

```
[jerry]$ awk '{if ($0 ~/Shyam/) next; print $0}' marks.txt
```

执行上面的命令可以得到如下的结果：

```
1)  Amit   Physics   80
2)  Rahul   Maths   90
4)  Kedar   English   85
5)  Hari   History   89
```

nextfile

nextfile 停止处理当前文件，从下一个文件第一个记录开始处理。下面的的例子中，匹配成功时停止处理第一个文件转而处理第二个文件：
首先创建两个文件。 file1.txt 内容如下:

```
file1:str1
file1:str2
file1:str3
file1:str4
```

文件 file2.txt 内容如下：

```
file2:str1
file2:str2
file2:str3
file2:str4
```

现在我们来测试 nextfile 函数。

```
[jerry]$ awk '{ if ($0 ~ /file1:str2/) nextfile; print $0 }' file1.txt file2.txt
```

执行上面的命令可以得到如下的结果：

```
file1:str1
file2:str1
file2:str2
file2:str3
file2:str4
```

return

return 用于从用户自定义的函数中返回值。请注意，如果没有指定返回值，那么的返回值是未定义的。下面的例子演示了 return 的使用方法：
首先，创建文件 functions.awk，内容如下：

```
function addition(num1, num2)
{
   result = num1 + num2

   return result
}

BEGIN {
   res = addition(10, 20)
   print "10 + 20 = " res
}
```

执行上面的命令可以得到如下的结果：

```
10 + 20 = 30
```

system

system 函数可以执行特定的命令然后返回其退出状态。返回值为 0 表示命令执行成功；非 0 表示命令执行失败。下面的示例中执行 Date 显示当前的系统时间，然后输出命令的返回状态：

```
[jerry]$ awk 'BEGIN { ret = system("date"); print "Return value = " ret }'
```

执行上面的命令可以得到如下的结果：

```
Sun Dec 21 23:16:07 IST 2014
Return value = 0
```

13

用户自定义函数

函数是程序的基本构造部分。AWK 允许我们自定义函数。事实上，大部分的程序功能都可以被切分成多个函数，这样每个函数可以独立的编写与测试。函数不仅提高了代码的复用度也提高代码的鲁棒性。

下面是用户自定义函数的一般形式：

```
function function_name(argument1, argument2, ...)
{
   function body
}
```

上述定义函数的语法中：

function_name 是用户自定义函数的名称。函数名称应该以字母开头，其后可以是数字、字母或下划线的自由组合。AWK 保留的关键字不能作为用户自定义函数的名称。

自定义函数可以接受多个输入参数，这些参数之间通过逗号分隔。参数并不是必须的。我们也可以定义没有任何输入参数的函数。

function body 是函数体部分，它包含 AWK 程序代码。

下面我们实现了两个简单函数，它们分别返回两个数值中的最小值和最大值。我们在主函数 main 中调用了这两个函数。 文件 functions.awk 内容如下：

```

\# Returns minimum number
function find_min(num1, num2)
{
  if (num1 < num2)
    return num1
  return num2
}

\# Returns maximum number
function find_max(num1, num2)
{
  if (num1 > num2)
    return num1
  return num2
}

\# Main function
function main(num1, num2)
{
  # Find minimum number
  result = find_min(10, 20)
```

```
  print "Minimum =", result

  # Find maximum number
  result = find_max(10, 20)
  print "Maximum =", result
}

\# Script execution starts here
BEGIN {
  main(10, 20)
}
```

执行上面的命令可以得到如下的结果：

```
Minimum = 10
Maximum = 20
```

14

输出重定向

到目前为止我们输出的数据都是输出到标准输出流中。不过我们也可以将数据输出重定向到文件中。重定向操作往往出现在 print 或者 printf 语句中。 AWK 中的重定向方法与 shell 重定向十分相似，除了 AWK 重定向只用于 AWK 程序中外。本章节将讲述重定向的使用方法：

重定向操作符

重定向操作符的使用方法如下：

```
print DATA > output-file
```

上面重定向操作将输出数据重定向到 output-file 中。如果 output-file 文件不存在，则先创建该文件。使用这种重定向方式时，数据输出前会将 output-file 文件中原有的数据删除。下面的示例将 Hello,World!!! 消息重定向输出到文件中。

先创建文件并在文件中输入一些数据。

```
[jerry]$ echo "Old data" > /tmp/message.txt
[jerry]$ cat /tmp/message.txt
```

执行上面的命令可以得到如下的结果：

```
Old data
```

再用 AWK 重定向操作符重定向数据到文件 message.txt 中。

```
[jerry]$ awk 'BEGIN { print "Hello, World !!!" > "/tmp/message.txt" }'
[jerry]$ cat /tmp/message.txt
```

执行上面的命令可以得到如下的结果：

```
Hello, World !!!
```

追加重定向

追加重定向操作符的语法如下：

```
print DATA >> output-file
```

用这种重定向方式将数据追加到 output-file 文件的末尾。如果文件不存在则先创建该文件。示例如下：

创建文件并输入一些数据：

```
[jerry]$ echo "Old data" > /tmp/message.txt
[jerry]$ cat /tmp/message.txt
```

执行上面的命令可以得到如下的结果：

```
Old data
```

再使用 AWK 追加操作符追加内容到文件中：

```
[jerry]$ awk 'BEGIN { print "Hello, World !!!" > "/tmp/message.txt" }'
[jerry]$ cat /tmp/message.txt
```

执行上面的命令可以得到如下的结果：

```
Old data
Hello, World !!!
```

管道

除了使用文件在程序之间传递数据之外，AWK 还提供使用管道将一个程序的输出传递给另一个程序。这种重定向方式会打开一个管道，将对象的值通过管道传递给管道另一端的进程，然后管道另一端的进程执行命令。下面是管道的使用方法：

```
print items | command
```

下面的例子中我们使用 tr 命令将小写字母转换成大写。

```
[jerry]$ awk 'BEGIN { print "hello, world !!!" | "tr [a-z] [A-Z]" }'
```

执行上面的命令可以得到如下的结果：

```
HELLO, WORLD !!!
```

双向通信通道

AWK 允许使用 |& 与一个外部进程通信，并且可以双向通信。下面的例子中，我们仍然使用 tr 命令将字母转换为大写字母。 command.awk 文件内容如下：

```
BEGIN {
   cmd = "tr [a-z] [A-Z]"
   print "hello, world !!!" |& cmd
   close(cmd, "to")
   cmd |& getline out
```

```
    print out;
    close(cmd);
}
```

执行上面的命令可以得到如下的结果：

```
HELLO, WORLD !!!
```

脚本的内容看上去很神秘吗？让我们一步一步揭开它神秘的面纱。

- 第一条语句 cmd = "tr [a-z] [A-Z]" 在AWK 中建立了一个双向的通信通道。
- 第二条语句 print 为 tr 命令提供输入。&| 表示双向通信。
- 第三条语句 close(cmd, "to") 执行后关闭 to 进程。
- 第四条语句 cmd |& getline out 使用 getline 函数将输出存储到 out 变量中。
- 接下来的输出语句打印输出的内容，最后 close 函数关闭 cmd。

15

优雅地输出

前面我们已经用过了 AWK 中的 print 函数与 printf 函数，它们将数据输出到标准输出流中。其实 printf 函数的功能远比我们前面演示的强大。这个函数是从 C 语言中借鉴来而的，主要用于生成格式化的输出。下面是 printf 的使用方法：

```
printf fmt, expr-list
```

其中，fmt 是字符串常量或者格式规格说明字符串，expr-list 是与格式说明相对应的参数列表。

转义序列

与一般字符串一样，格式化字符串也能内嵌转义序列。 AWK 支持的转义序列如下：

换行符

下面的例子中使用换行符将 Hello 与 World 分开输出到独立两行：

```
[jerry]$ awk 'BEGIN { printf "Hello\nWorld\n" }'
```

执行上面的命令可以得到如下的结果：

```
Hello
World
```

水平制表符

如下示例，使用制表符显示不同的域：

```
[jerry]$ awk 'BEGIN { printf "Sr No\tName\tSub\tMarks\n" }'
```

执行上面的命令可以得到如下的结果：

```
Sr No    Name    Sub Marks
```

垂直制表符

如下示例，使用垂直制表符输出不同域：

```
[jerry]$ awk 'BEGIN { printf "Sr No\vName\vSub\vMarks\n" }'
```

执行上面的命令可以得到如下的结果：

```
Sr No
   Name
      Sub
         Marks
```

退格符

下面的例子中，我们在每个域输出后都再输出退格符（最后一个域除外）。这样前三个域的每一域的最后一个字符都会被删除。比如说，Field 1 输出为 Field。因为最后一个字符被退格符删除。不过Field 4可以正常显示，因为在Field 4输出后没有输出退格符。

```
[jerry]$ awk 'BEGIN { printf "Field 1\bField 2\bField 3\bField 4\n" }'
```

执行上面的命令可以得到如下的结果：

```
Field Field Field Field 4
```

回车

下面的例子中，我们在每个域输出后输出一个回车符，随后输出的域会覆盖之前输出的内容。也就是说，我们只能看到最后输出的 Field 4。

```
[jerry]$ awk 'BEGIN { printf "Field 1\rField 2\rField 3\rField 4\n" }'
```

执行上面的命令可以得到如下的结果：

```
Field 4
```

换页符

下面的例子中每个域后输出后输出一个换页符：

```
[jerry]$ awk 'BEGIN { printf "Sr No\fName\fSub\fMarks\n" }'
```

执行上面的命令可以得到如下的结果：

```
Sr No
   Name
      Sub
         Marks
```

格式说明符

与 C 语言一样，AWK 也定义了格式说明符。 AWK 的 printf 允许如下的格式的转换：

%c

输出单个字符。如果参数是个数值，那么数值也会被当作字符然后输出。如果参数是字符串，那么只会输出字符串的第一个字符。

```
[jerry]$ awk 'BEGIN { printf "ASCII value 65 = character %c\n", 65 }'
```

执行上面的命令可以得到如下的结果：

```
ASCII value 65 = character A
```

%d 与 %i

输出十进制数的整数部分。

```
[jerry]$ awk 'BEGIN { printf "Percentags = %d\n", 80.66 }'
```

执行上面的命令可以得到如下的结果：

```
Percentags = 80
```

%e 与 %E

以 [−]d.dddddde[+−]dd 的格式输出浮点数。

```
[jerry]$ awk 'BEGIN { printf "Percentags = %E\n", 80.66 }'
```

执行上面的命令可以得到如下的结果：

```
Percentags = 8.066000e+01
```

%E 格式使用 E 而不是 e。

```
[jerry]$ awk 'BEGIN { printf "Percentags = %e\n", 80.66 }'
```

执行上面的命令可以得到如下的结果：

```
Percentags = 8.066000E+01
```

%f

以 [−]ddd.dddddd 的格式输出浮点数。

```
[jerry]$ awk 'BEGIN { printf "Percentags = %f\n", 80.66 }'
```

执行上面的命令可以得到如下的结果：

```
Percentags = 80.660000
```

%g 与 %G

输出浮点数，使用 %e 或 %E 转换。但它们会删除那些对数值无影响的 0。

```
[jerry]$ awk 'BEGIN { printf "Percentags = %g\n", 80.66 }'
```

执行上面的命令可以得到如下的结果：

```
Percentags = 80.66
```

%G 使用 %E 格式化，而不是 %e。

```
[jerry]$ awk 'BEGIN { printf "Percentags = %e\n", 80.66 }'
```

执行上面的命令可以得到如下的结果：

```
Percentags = 80.66
```

%o

无符号八进制输出。

```
[jerry]$ awk 'BEGIN { printf "Octal representation of decimal number 10 = %o\n", 10}'
```

执行上面的命令可以得到如下的结果：

```
Octal representation of decimal number 10 = 12
```

%u

无符号十进制数输出。

```
[jerry]$ awk 'BEGIN { printf "Unsigned 10 = %u\n", 10 }'
```

执行上面的命令可以得到如下的结果：

```
Unsigned 10 = 10
```

%x 与 %X

输出十六进制无符号数。%X 中使用大写字母，%x 使用小写字母。

```
[jerry]$ awk 'BEGIN { printf "Hexadecimal representation of decimal number 15 = %x\n", 15}'
```

执行上面的命令可以得到如下的结果：

```
Hexadecimal representation of decimal number 15 = f
```

使用 %X 的输出结果如下：

```
[jerry]$ awk 'BEGIN { printf "Hexadecimal representation of decimal number 15 = %X\n", 15}'
```

执行上面的命令可以得到如下的结果：

```
Hexadecimal representation of decimal number 15 = F
```

%%

输出百分号（%），不需要输入参数。

```
[jerry]$ awk 'BEGIN { printf "Percentags = %d%%\n", 80.66 }'
```

执行上面的命令可以得到如下的结果：

```
Percentags = 80%
```

% 的可选参数

% 可以使用如下可选参数：

宽度

输出域会被填充满足宽度要求。默认情况下使用空格字符填充。但是，当标志 0 被设置后会使用 0 填充。

```
[jerry]$ awk 'BEGIN { num1 = 10; num2 = 20; printf "Num1 = %10d\nNum2 = %10d\n", num1, num2 }'
```

执行上面的命令可以得到如下的结果：

```
Num1 =         10
Num2 =         20
```

前导零

紧接在 % 后的零被当作标示，表示输出应该使用零填充而不是空格字符。请注意，只有当域的宽度比要求宽度小时该标示才会有效。示例如下：

```
[jerry]$ awk 'BEGIN { num1 = -10; num2 = 20; printf "Num1 = %05d\nNum2 = %05d\n", num1, num2 }'
```

执行上面的命令可以得到如下的结果：

```
Num1 = -0010
Num2 = 00020
```

左对齐

输出域被设置为左对齐。当输出字符串字符数比指定宽度少时，你可能希望在输出它时能左对齐。比如，在右边添加空格符。在 % 之后数字之前使用减号（ - ）即可指定输出左对齐。下面的例子中，AWK 的输出做为 cat 的输入，在 cat 中输出行结束符号（ $ ）。

```
[jerry]$ awk 'BEGIN { num = 10; printf "Num = %-5d\n", num }' | cat -vte
```

执行上面的命令可以得到如下的结果：

```
Num1 = -0010
Num2 = 00020
```

符号前缀

输出数值的符号，正号也输出。

```
[jerry]$ awk 'BEGIN { num1 = -10; num2 = 20; printf "Num1 = %+d\nNum2 = %+d\n", num1, num2 }'
```

执行上面的命令可以得到如下的结果：

```
Num1 = -10
Num2 = +20
```

哈希（Hash）

使用 Hash 可以为 %o 的结果前添加0，为 %x 或 %X 输出的结果前添加 0x 或 0X（结果不为零时），为 %e，%E，%f，%F添加小数点；对于 %g 或 %G，使用哈希可以保留尾部的零。使用示例如下：

```
[jerry]$ awk 'BEGIN { printf "Octal representation = %#o\nHexadecimal representaion = %#X\n", 10, 10}'
```

执行上面的命令可以得到如下的结果：

```
Octal representation = 012
Hexadecimal representation = 0XA
```