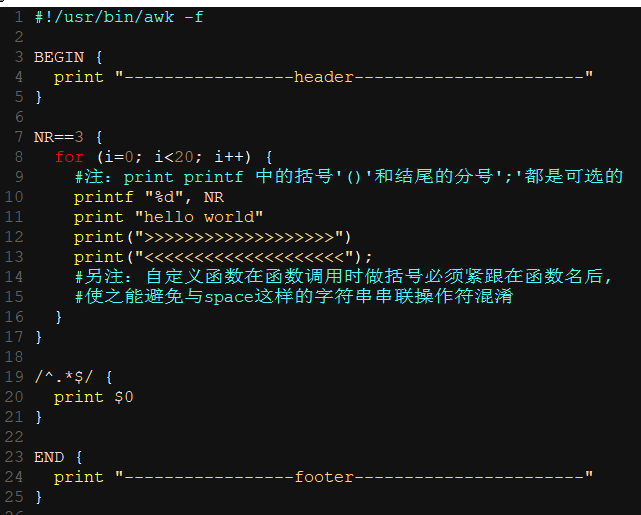
awk --help

man awk

info awk #其中有更多的example

注：awk中用的正则表达式同egrep是一样的

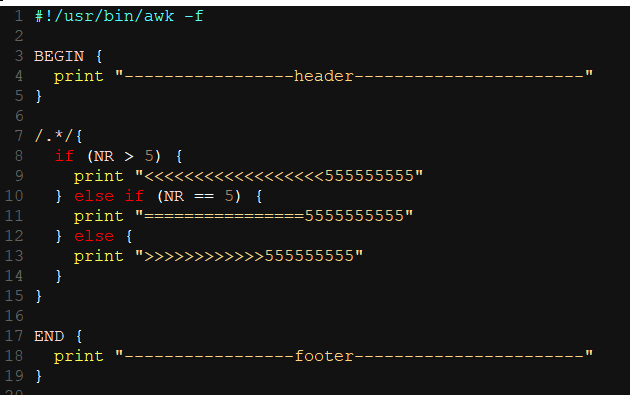




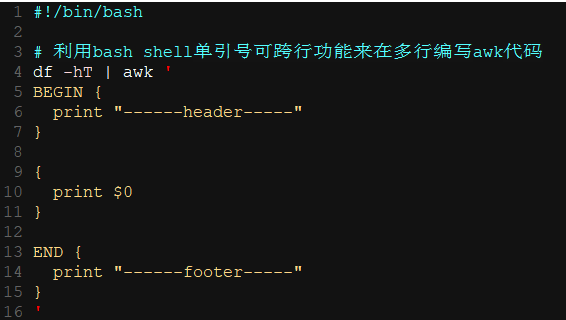




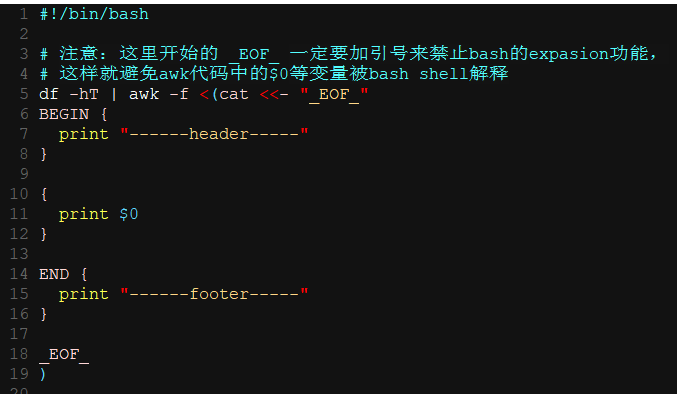
一个if...else的例子



一个利用bash shell单引号可跨行功能来在多行编写awk代码的例子



一个将awk与here document结合的例子(同时利用了进程替换特性)



# 注：$0 代表整行

[root@bogon dir]# awk -F: '{print $0;}' /etc/passwd

[root@bogon dir]# awk -F: '{print $1;}' /etc/passwd

[root@bogon dir]# awk -F: '{print $1, $2, $3}' /etc/passwd

定义变量

[root@bogon dir]# echo '' | awk -v name="martin" '{print "name is: ", name}'

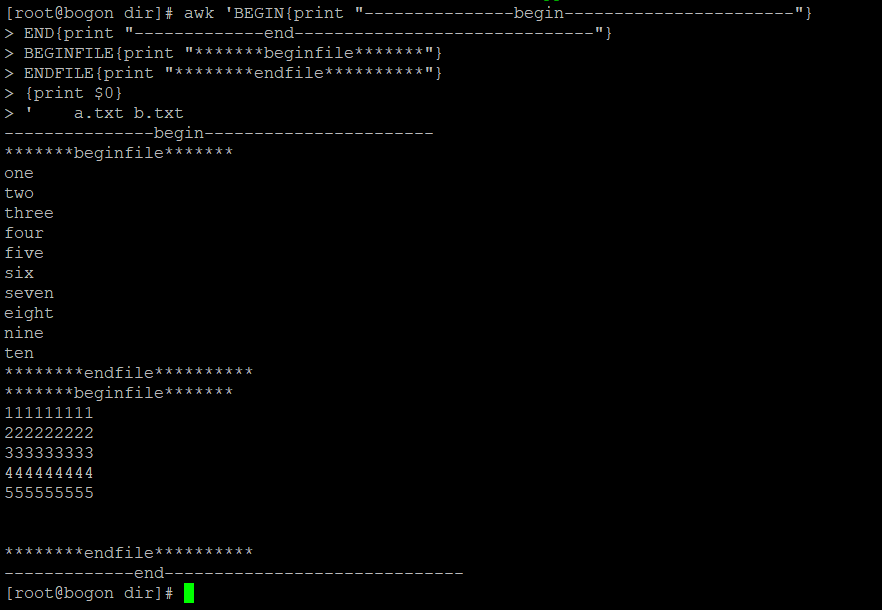
[root@bogon ~]# echo '' | awk 'BEGIN{name="mike"}{print "name is: ", name}'

可通过多次指定-v选项定义多个变量

[root@bogon dir]# awk -v name="root" -v password="1234" 'BEGIN{printf "%s : %s\n", name, password}'#数字不用先定义,可直接使用

[root@bogon ~]# awk 'BEGIN{print number++}'

AWK PROGRAM EXECUTION



VARIABLES, RECORDS AND FIELDS

awk变量特点：

动态，第一次使用时被创建

类型（可分自定义和预定义变量）：注：awk中变量或字段具体解释为那种类型由使用上下文决定

浮点型

字符串

一维数组（可以模拟多维数组）

To force a variable to be treated as a number, add 0 to it; to force it to be treated as a string, concatenate it with the null string.

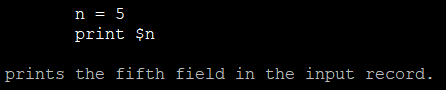
Records

变量RS： record separator, 默认为’\n’

Fields

变量FS：field separator， （a space by default）

Each field in the input record may be referenced by its position, $1, $2, and so on. $0 is the whole record. Fields need not be referenced by constants:



变量NF: number of fields, 表示输入记录的字段的总个数

|  |
| --- |
| 变量的一些说明：  References to non-existent fields (i.e. fields after $NF) produce the null-string. However, assigning to a non-existent field (e.g., $(NF+2) = 5) increases the value of NF,  creates any intervening fields with the null string as their value, and causes the value of $0 to be recomputed, with the fields being separated by the value of OFS. References  to negative numbered fields cause a fatal error. Decrementing NF causes the values of fields past the new value to be lost, and the value of $0 to be recomputed, with the  fields being separated by the value of OFS.  Assigning a value to an existing field causes the whole record to be rebuilt when $0 is referenced. Similarly, assigning a value to $0 causes the record to be resplit, creating  new values for the fields. |

Built-in Variables

其他一些内置变量：

|  |  |
| --- | --- |
| **FILENAME** | The name of the current input file. If no files are specified on the command line, the value of FILENAME is “-”. However, FILENAME is undefined inside the BEGIN block (unless set by getline). |
| **FNR** | The input record number in the current input file. |
| **NF** | The number of fields in the current input record. |
| **NR** | The total number of input records seen so far. |
| **OFS** | The output field separator, a space by default. |
| **ORS** | The output record separator, by default a newline. |
| **OFMT** | The output format for numbers, "%.6g", by default. |
| **PROCINFO** | The elements of this array provide access to information about the running AWK program.(详细见man awk) |
| **RS** | The input record separator, by default a newline. |
|  |  |
| **SUBSEP** | The character used to separate multiple subscripts in array elements, by default "\034". |
|  |  |

Arrays

[root@bogon dir]# awk 'BEGIN{

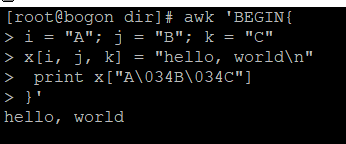
> i = "A"; j = "B"; k = "C"

> x[i, j, k] = "hello, world\n"

> print x["A\034B\034C"]

> }'

hello, world



[root@bogon dir]# awk 'BEGIN{

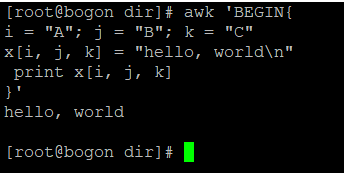
i = "A"; j = "B"; k = "C"

x[i, j, k] = "hello, world\n"

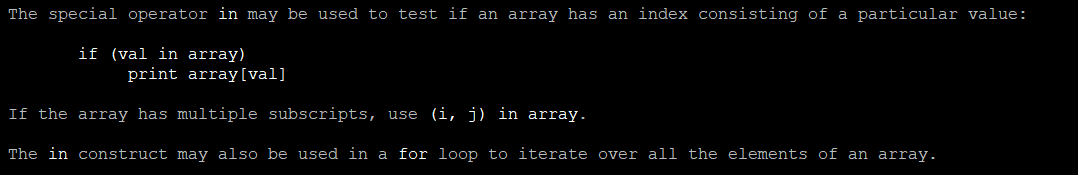
print x[i, j, k]

}'

hello, world



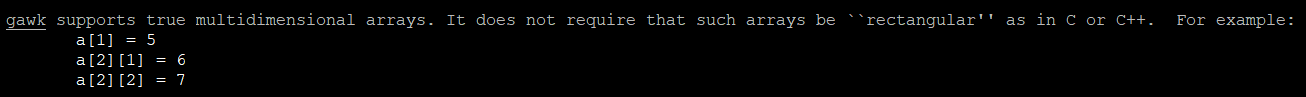
操作符**in可用于**判断数组是否包含相应的key或index(这种用法类似于javascript中的in操作符)，同时in还可用于遍历数组



操作符delete可用于删除数组元素（类似javascript中的delete操作符），同时delete还可用于删除整个数组本身，只要在删除数组时不带下标即可，这样就表示删除数组变量本身。

|  |
| --- |
| An element may be deleted from an array using the delete statement. The delete statement may also be used to delete the entire contents of an array, just by specifying the array name without a subscript. |

gawk支持真正非规则的多维数组(即无需是一个矩阵)：



Variable Typing And Conversion

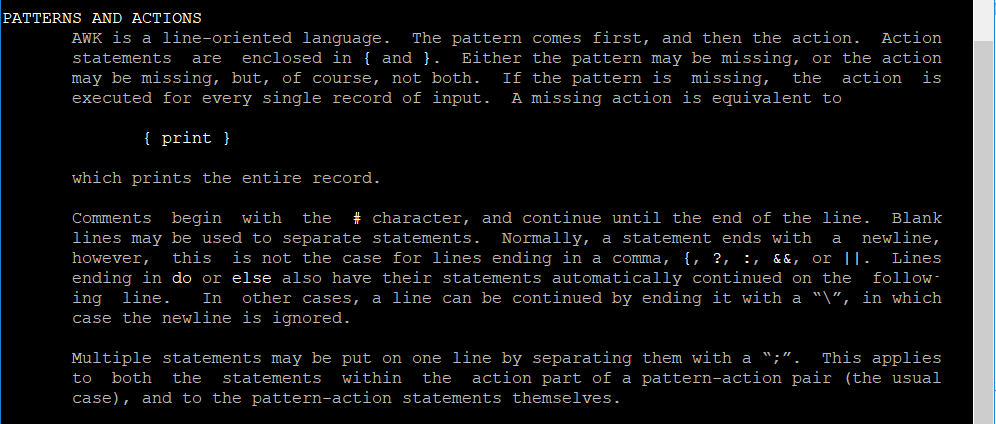
类型: （具体解释为那种类型由使用上下文决定）

浮点数

字符串

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| To force a variable to be treated as a number, add 0 to it; to force it to be treated as a string, concatenate it with the null string.  When a string must be converted to a number, the conversion is accomplished using strtod(3). A number is converted to a string by using the value of CONVFMT as a format string for sprintf(3), with the numeric value of the variable as the argument. However, even though all numbers in AWK are floating-point, integral values are always converted as  integers. Thus, given  CONVFMT = "%2.2f"  a = 12  b = a ""  the variable b has a string value of "12" and not "12.00".  注：Gawk中，两个变量比较，如果一个包含number值，另一个包含 “numeric string”值，则比较方式按照numerical比较（切记这里一定要满足包含“numeric string”的变量，因为类似"57"这样的常量字符串不是numeric strings）。  Note that string constants, such as "57", are not numeric strings, they are string constants. The idea of “numeric string” only applies to fields, getline input, FILENAME, ARGV elements, ENVIRON elements and the elements of an array created by split() or patsplit() that are numeric strings. The basic idea is that user input, and only user input, that looks numeric, should be treated that way.  未初始化的变量具有默认的number初始值0或空字符串值"":  Uninitialized variables have the numeric value 0 and the string value "" (the null, or empty, string).  Octal and Hexadecimal Constants  遵循c语言风格：   |  |  | | --- | --- | | 十进制 | 八进制 | | 9 | 011 |  |  |  | | --- | --- | | 十进制 | 十六进制 | | 17 | 0x11 |   String Constants  字符串用双引号引用(类似C语言)  The escape sequences may also be used inside constant regular expressions (e.g., /[ \t\f\n\r\v]/ matches whitespace characters).  In compatibility mode, the characters represented by octal and hexadecimal escape sequences are treated literally when used in regular expression constants. Thus, /a\52b/ is equivalent to /a\\*b/. |

PATTERNS AND ACTIONS



**Patterns**

AWK patterns may be one of the following:

**BEGIN**

**END**

**BEGINFILE**

**ENDFILE**

**/regular expression/**

**relational expression**

**pattern && pattern**

**pattern || pattern**

**pattern ? pattern : pattern**

**(pattern)**

**! pattern**

**pattern1, pattern2**

|  |
| --- |
| BEGINFILE and ENDFILE are additional special patterns whose bodies are executed before reading the first record of each command line input file and after reading the last record  of each file. Inside the BEGINFILE rule, the value of **ERRNO** will be the empty string if the file could be opened successfully. Otherwise, there is some problem with the file  and the code should use **nextfile** to skip it. If that is not done, gawk produces its usual fatal error for files that cannot be opened.  **pattern1, pattern2**  The pattern1, pattern2 form of an expression is called a range pattern. It matches all input records starting with a record that matches pattern1, and continuing until a record  that matches pattern2, inclusive. It does not combine with any other sort of pattern expression. |

Regular Expressions

Regular expressions are the extended kind found in egrep.

Actions

Action statements are enclosed in braces, { and }. Action statements consist of the usual assignment, conditional, and looping statements found in most languages. The operators, control statements, and input/output statements available are patterned after those in C.

Operators

The operators in AWK, in order of decreasing precedence, are

|  |  |
| --- | --- |
| **(...)** | Grouping |
| **$** | Field reference. |
| **++ --** | Increment and decrement, both prefix and postfix. |
| **^** | Exponentiation (\*\* may also be used, and \*\*= for the assignment operator). |
| **+ - !** | Unary plus, unary minus, and logical negation. |
| **\* / %** | Multiplication, division, and modulus. |
| **+ -** | Addition and subtraction. |
| **space** | String concatenation. |
| **| |&** | Piped I/O for getline, print, and printf. |
| **< > <= >= != ==** | The regular relational operators. |
| **~ !~** | Regular expression match, negated match. NOTE: Do not use a constant regular expression (/foo/) on the left-hand side of a ~ or !~. Only use one on the right-hand side. The expression /foo/ ~ exp has the same meaning as (($0 ~ /foo/) ~ exp). This is usually not what was intended.  切记：不要在运算符~和!~的左侧使用正则表达式，而应该只在其右侧使用 |
| **in** | Array membership. |
| **&&** | Logical AND. |
| **||** | Logical OR. |
| **?:** | The C conditional expression. This has the form expr1 ? expr2 : expr3. If expr1 is true, the value of the expression is expr2, otherwise it is expr3. Only one of expr2 and expr3 is evaluated. |
| **= += -= \*= /= %= ^=** | Assignment. Both absolute assignment (var = value) and operator-assignment (the other forms) are supported. |

**Control Statements**

The control statements are as follows:

**if (condition) statement [ else statement ]**

**while (condition) statement**

**do statement while (condition)**

**for (expr1; expr2; expr3) statement**

**for (var in array) statement**

**break**

**continue**

**delete array[index]**

**delete array**

**exit [ expression ]**

**{ statements }**

**switch (expression) {**

**case value|regex : statement**

**...**

**[ default: statement ]**

**}**

**I/O Statements**

**The printf Statement**

**Special File Names**

Numeric Functions

String Functions

Time Functions

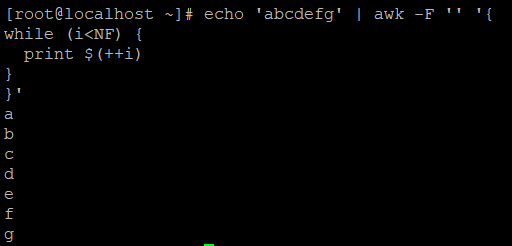
Bit Manipulations Functions

Type Function

Internationalization Functions

USER-DEFINED FUNCTIONS

EXAMPLES

If FS is the null string, then each individual character becomes a separate field.

当FS是一个字符串而非单个char是，则awk是FS为一个正则表达式



Special File Names

/dev/stdin The standard input.

/dev/stdout The standard output.

/dev/stderr The standard error output.

/dev/fd/n The file associated with the open file descriptor n.

These are particularly useful for error messages. For example:

print "You blew it!" > "/dev/stderr"

whereas you would otherwise have to use

print "You blew it!" | "cat 1>&2"

Numeric Functions

|  |  |
| --- | --- |
| **arithmetic functions** | |
| atan2(y, x) | Return the arctangent of y/x in radians. |
| cos(expr) | Return the cosine of expr, which is in radians. |
| exp(expr) | The exponential function. |
| **int(expr)** | Truncate to integer. |
| log(expr) | The natural logarithm function. |
| rand() | Return a random number N, between 0 and 1, such that 0 ≤ N < 1. |
| sin(expr) | Return the sine of expr, which is in radians. |
| sqrt(expr) | The square root function. |
| srand([expr]) | Use expr as the new seed for the random number generator. If no expr is provided, use the time of day. The return value is the previous seed for the random number generator. |
|  |  |

|  |
| --- |
| 其他参考：  <http://www.grymoire.com/Unix/Awk.html>  <https://likegeeks.com/awk-command/> 30 Examples For Awk Command In Text Processing **print和printf后的括号’()’是可选的，除非一些特殊情况必须使用（如在想使用关系运算符大于号“>”时，为了避免与重定向符号混淆，此时必须要加括号来辅助区分），另注：分号;也是可选的**  <http://www.grymoire.com/Unix/Awk.html> [PRINTF - formatting output](http://www.grymoire.com/Unix/Awk.html" \l "toc-uh-27) The *printf* is very similar to the C function with the same name. C programmers should have no problem using *printf* function.  *Printf* has one of these syntactical forms:  printf ( format); printf ( format, arguments...); printf ( format) >expression; printf ( format, arguments...) > expression;  The parenthesis and semicolon are optional. I only use the first format to be consistent with other nearby *printf* statements. A *print* statement would do the same thing. *Printf* reveals it's real power when formatting commands are used.  The first argument to the *printf* function is the format. This is a string, or variable whose value is a string. This string, like all strings, can contain special escape sequences to print control characters.  <https://www.oreilly.com/library/view/effective-awk-programming/0596000707/ch04.html> The print Statement The print statement is used to produce output with simple, standardized formatting. Specify only the strings or numbers to print, in a list separated by commas. They are output, separated by single spaces, followed by a newline. The statement looks like this:  print *item1*, *item2*, ...  The entire list of items may be optionally enclosed in parentheses. The parentheses are necessary if any of the item expressions uses the > relational operator; otherwise, it could be confused with a redirection (see [Section 4.6](https://www.oreilly.com/library/view/effective-awk-programming/0596000707/ch04.html#awkprog3-CHP-4-SECT-6)later in this chapter).  The items to print can be constant strings or numbers, fields of the current record (such as $1), variables, or any awk expression. Numeric values are converted to strings and then printed.  The simple statement print with no items is equivalent to print $0: it prints the entire current record. To print a blank line, use print "", where "" is the empty string. To print a fixed piece of text, use a string constant, such as "Don't Panic", as one item. If you forget to use the double-quote characters, your text is taken as an awk expression, and you will probably get an error. Keep in mind that a space is printed between any two items.  <https://superuser.com/questions/365452/how-to-write-awk-here-document> [how to write `awk here document`](https://superuser.com/questions/365452/how-to-write-awk-here-document)   <http://tldp.org/LDP/abs/html/process-sub.html>  <https://www.linuxquestions.org/questions/red-hat-31/fast-line-count-for-large-files-583991/>  awk 'END {print NR}' file  <https://stackoverflow.com/questions/12716570/count-lines-in-large-files>  Your limiting speed factor is the I/O speed of your storage device, so changing between simple newlines/pattern counting programs won't help, because the execution speed difference between those programs are likely to be suppressed by the way slower disk/storage/whatever you have.  But if you have the same file copied across disks/devices, or the file is distributed among those disks, you can certainly perform the operation in parallel. I don't know specifically about this Hadoop, but assuming you can read a 10gb the file from 4 different locations, you can run 4 different line counting processes, each one in one part of the file, and sum their results up:  $ dd bs=4k count=655360 if=/path/to/copy/on/disk/1/file | wc -l &  $ dd bs=4k skip=655360 count=655360 if=/path/to/copy/on/disk/2/file | wc -l &  $ dd bs=4k skip=1310720 count=655360 if=/path/to/copy/on/disk/3/file | wc -l &  $ dd bs=4k skip=1966080 if=/path/to/copy/on/disk/4/file | wc -l &  Notice the & at each command line, so all will run in parallel; dd works like cat here, but allow us to specify how many bytes to read (count \* bs bytes) and how many to skip at the beginning of the input (skip \* bs bytes). It works in blocks, hence, the need to specify bs as the block size. In this example, I've partitioned the 10Gb file in 4 equal chunks of 4Kb \* 655360 = 2684354560 bytes = 2.5GB, one given to each job, you may want to setup a script to do it for you based on the size of the file and the number of parallel jobs you will run. You need also to sum the result of the executions, what I haven't done for my lack of shell script ability.  If your filesystem is smart enough to split big file among many devices, like a RAID or a distributed filesystem or something, and automatically parallelize I/O requests that can be paralellized, you can do such a split, running many parallel jobs, but using the same file path, and you still may have some speed gain.  EDIT: Another idea that occurred to me is, if the lines inside the file have the same size, you can get the exact number of lines by dividing the size of the file by the size of the line, both in bytes. You can do it almost instantaneously in a single job. If you have the mean size and don't care exactly for the the line count, but want an estimation, you can do this same operation and get a satisfactory result much faster than the exact operation. |