Linux Shell脚本

变量

声明变量

```
name="zhangsan"
```

注意：等号两侧不能有空格

变量命名规则：

- **只包含字母、数字和下划线**
- **不能以数字开头**
- **避免使用 Shell 关键字**
- **大写字母表示常量**

引用变量

变量名前加$符号为引用变量

```
echo $name
echo ${name}
```

只读变量

```
name="zhangsan"
readonly name
```

删除变量

```
name="zhangsan"
unset name
```

声明命令执行结果

```
# 使用`xxx`
PWD_PATH=`pwd`
# 使用$(xxx)
PWD_PATH=$(pwd)
```

变量类型

字符串类型

单引号字符串

```
name='zhangsan'

# 单引号内变量无效
hello_name_2='hello $name'

# 单引号拼接字符串
hello_name='hello'$name' ! '
```

单引号字符串的限制：

- 单引号里的**任何字符都会原样输出**，单引号字符串中的变量是无效的；
- 单引号字符串中不能出现单独一个的单引号（**对单引号使用转义符也不行**），但可成对出现，作为字符串拼接使用。

双引号字符串

```
name="zhangsan"

# 双引号内可以使用变量
hello_name="hello $name"

# 双引号转义
hello_name="hello  \"$name\" !"
```

双引号的优点：

- 双引号里可以有变量
- 双引号里可以出现转义字符

字符串拼接

```
name="zhangsan"

# 使用${}直接拼接字符串
echo ${name}123

# 双引号字符串拼接
echo "my name is $name, ok!"
echo "my name is "${name}", ok!"

# 单引号字符串拼接
echo 'my name is'$name',ok!'
```

字符串长度

```
name="zhangsan"
echo ${#name}
```

字符串截取

根据下标截取

格式：**${string: start :length}**

string：表示源字符串

start：截取字符串的起始位置，注意第一个字符下标为0

length：截取字符串的长度

```
str="my name is zhangsan"

# 第1个字符开始截取2个字符，输出"my"
substr=${str:0:2}
```

根据字符截取

- 使用 `#` 截取匹配字符后右侧的内容

  格式：**${string#*chars}** 从左往右匹配，命中第一个匹配字符，剩下右侧字符串

  格式：**${string#*chars}** 从左往右匹配， 命中最后一个匹配字符，剩下右侧字符串
- 使用 `%` 截取匹配字符后左侧的内容

  格式：**${string%chars*}** 从右往左匹配，命中第一个匹配字符，剩下左侧字符串

  格式：**${string%chars*}** 从右往左匹配，命中最后一个匹配字符，剩下左侧字符串

```
PWD_PATH="/var/run/gitlab/puma"
# 获取目录的文件名
echo ${PWD_PATH##*/}

# 获取父目录
echo ${PWD_PATH%/*}
```

字符串替换

格式：**${string/old/new}**

string：源字符串

old：源字符串中要被替换的内容

new：新字符串

```
file_name="package.tgz"
echo ${file_name/tgz/zip}
```

数字类型

声明

```
total=5
```

计算

shell

```
a=10
b=20

# 中括号内必须有空格，不能与表达式相连
c=$[ $a + $b ]

# 双括号允许在比较语句中使用高级数学表达式，也可以与美元符号搭配，用于整型数据计算
d=$(($a + $b + $c))
```

双括号支持的运算符：

运算符	含义
val++	后增
val--	后减
++val	先增
--val	先减
!	逻辑求反
~	按位求反
**	幂运算
<<	左移位
>>	右移位
&	布尔与
|	布尔或
&&	逻辑与
||	逻辑或

expr

```
a=10
b=20

val=`expr $a + $b`
echo "a + b : $val"

val=`expr $a - $b`
echo "a - b : $val"

val=`expr $a \* $b`
echo "a * b : $val"

val=`expr $b / $a`
echo "b / a : $val"

val=`expr $b % $a`
echo "b % a : $val"
```

let

```
a=10
b=20
let a=a+b
echo $a
let a++
echo $a
```

数组类型

初始化

```
arr=(1 'nice' '2days')

# 凡是以空格分割的数据即可作为数组
# 例如：ls 的输出结果即为数组
for name in `ls`;do
    echo $name
done
# 例如：将','替换为' '形成数组
for name in `echo "a,b,c" | sed '/,/ /g'`;do
    echo $name
done
```

输出

```
echo ${arr[*]}

# '*'可以使用'@'代替
echo ${arr[@]}
```

新增、修改元素

```
arr[0]='yum'
```

数组长度

```
echo ${#arr[*]}
```

索引元素

索引下标0位第一个元素

```
echo ${arr[0]}
```

遍历数组

```
for e in ${arr[*]};do
    echo $e
done
```

字典类型

声明变量

```
declare -A id_name_map
```

初始化

```
id_name_map=(["1001"]="tom" ["1002"]="jery")
```

输出

```
# 输出所有key
echo ${!id_name_map[*]}
# 输出所有value
echo ${id_name_map[*]}

# '*'可以使用'@'代替
echo ${!id_name_map[@]}
echo ${id_name_map[@]}
```

新增、修改

```
id_name_map["1003"]="sunny"
id_name_map["1001"]="zhang"
```

字典长度

```
echo ${#id_name_map[*]}
```

遍历字典

```
for key in ${!id_name_map[*]};do
    echo ${id_name_map[$key]}
done
```

运算符

运算符表达式

```
name="zhangsan"
age=10

# 字符串变量比较相同
[ "$name" = "zhangsan" ]

# 数字变量比较相同
[ $age -eq 5 ]

# 数字变量转换为字符串变量
[ "$age" = "10" ]
```

注意：

- **中括号内的表达式两侧必须有空格**
- **字符串比较时，变量最好使用双引号转义，部分Linux系统的shell会报错：过多的参数**

字符串运算符

运算符	说明	举例
=	检测两个字符串是否相等，相等返回 true。	[$a =$b] 返回 false。
!=	检测两个字符串是否不相等，不相等返回 true。	[$a! =$b] 返回 true。
-z	检测字符串长度是否为0，为0返回 true。	[-z $a] 返回 false。
-n	检测字符串长度是否不为 0，不为 0 返回 true。	[-n "$a"] 返回 true。
$	检测字符串是否不为空，不为空返回 true。	[$a] 返回 true。

示例：

```
a="abc"
b="efg"

if [ $a = $b ];then
   echo "$a = $b : a 等于 b"
else
   echo "$a = $b: a 不等于 b"
fi

if [ $a != $b ];then
```

```
   echo "$a != $b : a 不等于 b"
else
   echo "$a != $b: a 等于 b"
fi

if [ -z $a ];then
   echo "-z $a : 字符串长度为 0"
else
   echo "-z $a : 字符串长度不为 0"
fi

if [ -n "$a" ];then
   echo "-n $a : 字符串长度不为 0"
else
   echo "-n $a : 字符串长度为 0"
fi

if [ $a ];then
   echo "$a : 字符串不为空"
else
   echo "$a : 字符串为空"
fi
```

数字运算符

运算符	说明	举例
-eq	检测两个数是否相等，相等返回 true。	[$a -eq b] 返回 false。
-ne	检测两个数是否不相等，不相等返回 true。	[$a -ne b] 返回 true。
-gt	检测左边的数是否大于右边的，如果是，则返回 true。	[$a -gt b] 返回 false。
-lt	检测左边的数是否小于右边的，如果是，则返回 true。	[$a -lt b] 返回 true。
-ge	检测左边的数是否大于等于右边的，如果是，则返回 true。	[$a -ge b] 返回 false。
-le	检测左边的数是否小于等于右边的，如果是，则返回 true。	[$a -le b] 返回 true。

示例：

```
a=10
b=20

if [ $a -eq $b ];then
   echo "$a -eq $b : a 等于 b"
else
   echo "$a -eq $b: a 不等于 b"
```

```
fi

if [ $a -ne $b ];then
   echo "$a -ne $b: a 不等于 b"
else
   echo "$a -ne $b : a 等于 b"
fi

if [ $a -gt $b ];then
   echo "$a -gt $b: a 大于 b"
else
   echo "$a -gt $b: a 不大于 b"
fi

if [ $a -lt $b ];then
   echo "$a -lt $b: a 小于 b"
else
   echo "$a -lt $b: a 不小于 b"
fi

if [ $a -ge $b ];then
   echo "$a -ge $b: a 大于或等于 b"
else
   echo "$a -ge $b: a 小于 b"
fi

if [ $a -le $b ];then
   echo "$a -le $b: a 小于或等于 b"
else
   echo "$a -le $b: a 大于 b"
fi
```

文件与目录运算符

用于判断文件或目录是否存在、是否具有相关权限等

操作符	说明
-s file	检测文件是否为空（文件大小是否大于0），不为空返回 true。
-f file	检测文件是否是普通文件（既不是目录，也不是设备文件），如果是，则返回 true。
-d file	检测文件是否是目录，如果是，则返回 true。
-e file	检测文件（包括目录）是否存在，如果是，则返回 true。
-c file	检测文件是否是字符设备文件，如果是，则返回 true。
-b file	检测文件是否是块设备文件，如果是，则返回 true。
-S file	检测文件是否为socket类型
-L file	检测文件是否为符号链接
-p file	检测文件是否是有名管道，如果是，则返回 true。
-g file	检测文件是否设置了 SGID 位，如果是，则返回 true。
-u file	检测文件是否设置了 SUID 位，如果是，则返回 true。
-r file	检测文件是否可读，如果是，则返回 true。
-w file	检测文件是否可写，如果是，则返回 true。
-x file	检测文件是否可执行，如果是，则返回 true。

示例：

```
data_path=/data
properties_file=$data_path/cupid/application.properties

if [ -d $data_path ];then
    if [ -f $properties_file ];then
        cat $properties_file
    else
        echo "$properties_file not exit !"
    fi
else
    echo "$data_path not exit !"
fi
```

逻辑运算符

[ ] 逻辑运算符

运算符	说明	举例
!	非运算，表达式为 true 则返回 false，否则返回 true。	[! false] 返回 true。
-o	或运算，有一个表达式为 true 则返回 true。	[$a -lt 20 -o$b -gt 100] 返回 true。
-a	与运算，两个表达式都为 true 才返回 true。	[$a -lt 20 -a$b -gt 100] 返回 false。

[[ ]] 逻辑运算符

运算符	说明	举例
&&	逻辑的 AND	[[$a -lt 100 && $b -gt 100]] 返回 false
||	逻辑的 OR	[[$a -lt 100 || $b -gt 100]] 返回 true

参数传递

shell脚本执行时，可携带参数传入脚本中

```
#!/bin/bash

# shell文件路径
echo $0

# 第一个参数
echo $1

# 第二个参数
echo $2
```

参数说明：

参数	说明
$#	传递到脚本的参数个数
$*	以一个单字符串显示所有向脚本传递的参数。 如"*”用「”」括起来的情况、以”1 2 ... n"的形式输出所有参数。
$@	与*相同，但是使用时加引号，并在引号中返回每个参数。如”@"用「"」括起来的情况、以" 1 ””2" ... "$n" 的形式输出所有参数。
$$	脚本运行的当前进程ID号
$!	后台运行的最后一个进程的ID号

参数	说明
$-	显示Shell使用的当前选项，与set命令功能相同。
$?	显示最后命令的退出状态。0表示没有错误，其他任何值表明有错误。shell脚本中exit语句的返回值

流程控制

if

示例：

```
name="zhang"

if [ "$name" = "zhang" ];then
    echo "1"
elif [ "$name" = "wang" ];then
    echo "2"
else
    echo "none"
fi
```

if的判断语句有多种：

[表达式]

单中括号中，表达式两侧应有空格，且表达式逻辑运算符使用"-a"、"-o"、"!"

[[表达式]]

双中括号中，表达式两侧应有空格，且表达式逻辑运算符使用"&&"，"||"

((...))

双括号中，表达式可以使用">"、"<"、"=="等运算符

for

```
# 循环目录下的文件
for file in `ls /data`;do
    echo "file name is $file"
done

# 循环1至10
for i in `seq 1 10`;do
    echo $i
done
```

while

```
count=0
while true;do
    if [ $count -eq 5 ];then
        continue
    fi
    if [ $count -ge 10 ];then
        break
    fi
    echo $count
done
```

函数

```
# 定义方法，返回值必须是整数，取值范围(0-255)
function add() {
    a=$1
    b=$2
    return $(($a + $b))
}

# 执行方法，参数传递
add 1 2

# 获取返回值
echo $?
```

文件包含

shell可以将其他脚本声明的变量等导入当前脚本

```
. filename   # 注意点号(.)和文件名中间有一空格

或

source filename
```

Linux 命令

seq

Linux seq 命令用以指定增量从首数开始打印数字到尾数。

```
seq [选项]... 尾数
seq [选项]... 首数 尾数
seq [选项]... 首数 增量 尾数
```

参数说明：

- -f, --format=格式 使用printf 样式的浮点格式
- -s, --separator=字符串使用指定字符串分隔数字(默认使用：\n)
- -w, --equal-width 在列前添加0 使得宽度相同

tr

语法说明

Linux tr 命令用于转换或删除文件中的字符。

tr 指令从标准输入设备读取数据，经过字符串转译后，将结果输出到标准输出设备。

```
tr [OPTION]… SET1 [SET2]
```

参数说明：

- -c, --complement：反选设定字符。也就是符合 SET1 的部份不做处理，不符合的剩余部份才进行转换
- -d, --delete：删除指令字符
- -s, --squeeze-repeats：缩减连续重复的字符成指定的单个字符
- -t, --truncate-set1：削减 SET1 指定范围，使之与 SET2 设定长度相等

字符集合的范围：

- \\ 反斜杠
- \b Ctrl-H 退格符
- \f Ctrl-L 走行换页
- \n Ctrl-J 新行
- \r Ctrl-M 回车
- \t Ctrl-I tab键
- \v Ctrl-X 水平制表符
- CHAR1-CHAR2： 字符范围从 CHAR1 到 CHAR2 的指定，范围的指定以 ASCII 码的次序为基础，只能由小到大，不能由大到小，例如：a-z、A-Z。
- [CHAR*]： 这是 SET2 专用的设定，功能是重复指定的字符到与 SET1 相同长度为止
- [CHAR*REPEAT]： 这也是 SET2 专用的设定，功能是重复指定的字符到设定的 REPEAT 次数为止 (REPEAT 的数字采 8 进位制计算，以 0 为开始)
- [:digit:]： 所有数字
- [:alpha:]： 所有字母字符
- [:alnum:]： 所有字母字符与数字
- [:lower:]： 所有小写字母
- [:upper:]： 所有大写字母
- [:punct:]： 所有标点字符
- [:blank:]： 所有水平空格
- [:cntrl:]： 所有控制字符
- [:graph:]： 所有可打印的字符(不包含空格符)
- [:print:]： 所有可打印的字符(包含空格符)

- [:space:]： 所有水平与垂直空格符
- [:xdigit:]： 所有16 进位制的数字
- [=CHAR=]： 所有符合指定的字符(等号里的 CHAR，代表你可自订的字符)

替换

小写转换为大写

```
echo "abcDEFG" | tr a-z A-Z
echo "abcDEFG" | tr [:lower:] [:upper:]
```

tab替换为四个空格

```
echo "  123" | tr  \t  '    '
```

删除

删除小写字符

```
echo "abcDEFG" | tr -d a-z
```

grep

语法说明

grep命令主要用于文本过滤，查询出需要的行

```
grep [OPTION]... PATTERN [FILE]...

模式选择:
  -E, --extended-regexp     扩展正则表达式匹配
  -F, --fixed-strings       固定字符串匹配
  -G, --basic-regexp        基础正则表达式(默认)
  -P, --perl-regexp         Perl语言正则表达式匹配
  -e, --regexp=PATTERN      使用指定的PATTERN进行匹配（用于多个匹配条件）
  -f, --file=FILE           从文件中读取匹配内容
  -i, --ignore-case         忽略大小写
  -w, --word-regexp         强制匹配整个单词
  -x, --line-regexp         强制匹配整行
  -z, --null-data           以0字节作为换行判断依据

杂项:
  -s, --no-messages         阻止异常输出
  -v, --invert-match        反选

输出控制:
  -m, --max-count=NUM       匹配NUM行后停止
  -b, --byte-offset         打印命中字符的在当前行的字节偏移量
```

```
  -n, --line-number          打印命中行的行号
      --line-buffered        每行刷新输出
  -H, --with-filename        打印文件名
  -h, --no-filename          阻止打印文件名
      --label=LABEL          使用LABLE作为文件名
  -o, --only-matching        只显示匹配的内容
  -q, --quiet, --silent      阻止所有输出
      --binary-files=TYPE    假定文件类型为binary;
                             TYPE包括：'binary', 'text','without-match'
  -a, --text                 假定文件类型为text
  -I                         equivalent to --binary-files=without-match
  -d, --directories=ACTION   如何处理目录;
                             ACTION包括：'read', 'recurse', or 'skip'
  -D, --devices=ACTION       如何处理devices, FIFOs and sockets;
                             ACTION包括：'read' or 'skip'
  -r, --recursive            like --directories=recurse（递归）
  -R, --dereference-recursive  likewise, but follow all symlinks
      --include=FILE_PATTERN  search only files that match FILE_PATTERN
      --exclude=FILE_PATTERN  skip files and directories matching FILE_PATTERN
      --exclude-from=FILE   skip files matching any file pattern from FILE
      --exclude-dir=PATTERN  directories that match PATTERN will be skipped.
  -L, --files-without-match  没有选中的行只打印文件名
  -l, --files-with-matches  选中的行只打印文件名
  -c, --count                仅打印匹配的行数
  -T, --initial-tab          打印列对齐
  -Z, --null                 打印文件名之后追加0字节

上下文控制：
  -B, --before-context=NUM  打印匹配行之前的NUM行
  -A, --after-context=NUM   打印匹配行之后的NUM行
  -C, --context=NUM         打印匹配行之前和之后的NUM行
  -NUM                      类似 --context=NUM
      --color[=WHEN],
      --colour[=WHEN]       匹配的内容进行标记并高亮显示;
                            WHEN包括：'always', 'never', or 'auto'
  -U, --binary              EOL不去掉CR字符 (MSDOS/Windows)
```

正则表达式

符号	含义
c	匹配字符
\c	匹配转义后的字符c
.	匹配一个非换行符的字符
^	锚定行的开始
$	锚定行的结束
[abc...]	匹配一个指定范围内的字符，如[Gg]rep匹配Grep和grep
[^abc...]	匹配一个不在指定范围内的字符，如：[^A-FH-Z]rep，不以A-F和H-Z开头且紧跟rep
r1|r2	匹配条件 r1 或 r2
r1r2	匹配条件 r1 和 r2
r?	匹配0次或一次
r+	匹配一次以上
r*	匹配任意次
(r)	匹配组
r{n}	匹配n次
r{n,}	匹配n次以上
r{n,m}	匹配n至m次
\<	匹配单词左边界
\>	匹配单词右边界
\s	匹配任意空白字符
\S	匹配任意非空白字符
\w	匹配单词组成字符(大小写字母、数字、下划线)
\W	匹配非单词组成字符
\y	匹配单词左右边界部分的空字符位置 "hello world"
\B	和\y相反，匹配单词内部的空字符位置，例如"crate" ~ `/c\Brat\Be/` 成功
\`	匹配字符串的绝对行首 "abc\ndef"
\'	匹配字符串的绝对行尾

示例：

```
# 过滤出包含Excepiton的行，并打印其之前的5行和之后的20行
grep -B 5 -A 20 'Exception'  /data/cupid/webserver/logs/sajjm-error.log

# 统计日志中Exception出现的次数
grep -c Exception /data/cupid/webserver/logs/sajjm-error.log

# 过滤出包含kafka的进程，且过滤掉执行grep命令的进程
ps -ef |grep kafka | grep -v grep

# 多个匹配条件，过滤出包含‘Exception’或'com.venus'的行
grep -e 'Exception' -e 'com.venus' /data/cupid/webserver/logs/sajjm-error.log
```

sed

语法说明

sed命令主要用于文本的替换和修改

语法：

- sed [选项] 'sed编辑命令' 输入文件
- shell 命令 | sed [选项] 'sed编辑命令'
- sed [选项] -f sed脚本文件 输入文件

选项：

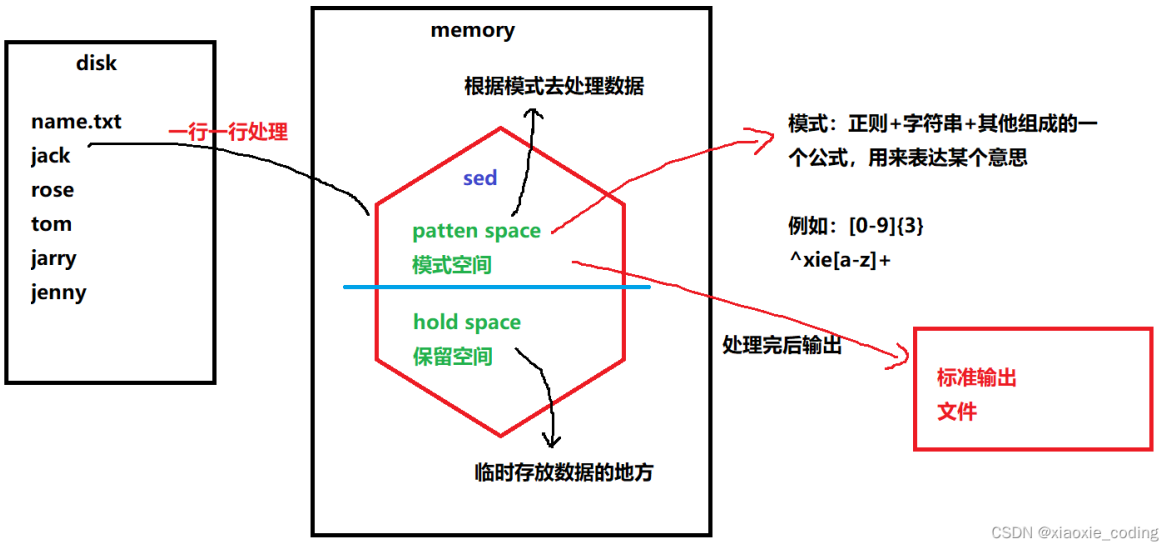
- -n：只显示匹配处理的行（否则会输出所有）
- -e：执行多个编辑命令时（一般用 ; 代替）
- -i：直接在文件中进行修改，而不是输出到屏幕
- -r：支持扩展正则表达式
- -f：从脚本文件中读取内容并执行（文件中的编辑命令每行一个,不用；隔开）

编辑命令：

- p：打印匹配行 print
- a：在匹配行后面追加 append
- i：在匹配行前面插入 insert
- c：整行替换
- s：字符串替换（匹配正则表达式）substitution
- d：删除指定行 delete
- r：将文件的内容读入 read
- w：将文本写入文件 write
- =：输出行号

检索匹配方式：

- 行号匹配
- 模式匹配(正则表达式=字符+特殊符号)



行号匹配

匹配格式：**sed -n '行号1，行号2命令' 输入文件**

```
# 输出第一行
sed -n '1p' /etc/passwd
# 输出最后一行
sed -n '$p' /etc/passwd
# 输出一到五行
sed -n '1,5p' /etc/passwd
# '+'设置行数，输出第四行及其后面五行
sed -n '4,+5p' /etc/passwd
# '~'设置步长值，输出单数行，步长为2
cat -n /etc/passwd | sed -n '1~2p'
# '!'匹配取反，4到最后一行不显示
sed -n '4,$!p' /etc/passwd
# 多个命令用';'分割，输出1行，3至5行，7行及之后的2行
sed -n '1p;3,5p;7,+2p' /etc/passwd
```

模式匹配

匹配格式：**sed -n '/模式/命令' 输入文件**

```
# 输出有root的行
sed -n '/root/p' /etc/passwd
# 以#或者$开头的行不显示
cat /etc/ssh/sshd_config |sed -rn '/^#|^$/!p'
# 显示以/结尾的行，需要转义
df -Th| sed -n '/\/$/p'
```

p:打印命令

```
# 输出第一行和包含data字符串的行
sed -n '1p;/data/p' rc.local
```

a:追加命令

格式：sed -i **'匹配模式a 追加内容'** 输入文件

```
# 第一行后增加一行，内容为：abc
sed -i '1a abc' rc.local
# 所有包含'/data'的行后追加一行，内容为：abc
sed -i '/\/data/a abc' rc.local
```

i: 插入命令

```
# 第一行前增加一行，内容为：abc
sed -i '1i abc' rc.local
# 所有包含'/data'的行前追加一行，内容为：abc
sed -i '/\/data/i abc' rc.local
```

d:删除命令

```
# 以正则表达式匹配删除，内容为abc的行被删除
sed -ri '/^abc$/d' rc.local
# 通配符匹配删除
sed -i '/xda-web.stub/d' /etc/rc.local
# '/'需要转义
sed -i '/(\/data\/cupid\/xda-web.stub &)/d' /etc/rc.local
```

c:整行替换命令

```
# 将第一行替换为#/bin/bash
sed -i '1c #/bin/bash' /etc/rc.ocal
# 将包含data字符串的行替换为123
sed -i '/data/c 123' /etc/rc.local
```

s:字符串替换命令

sed -i **'[行号或模式]s/查找内容/替换内容/[替换标记]'** 输入文件

替换标记：

- 数字：替换每行的第几个
- g：全局替换，否则只替换第一个字符串。例如ng从第n个开始替换
- p：显示被执行替换操作的行，和-n合用
- w：将执行替换操作的行输出到指定文件

```
# 第11行的10替换为20
sed -i '11 s/10/20/' rc.local

# 将','替换为空格，从而转换成数组
echo "apple,pen,orange" | sed  's/,/ /g'

# 将内容包含'/data'的行的'cupid'全部替换为'xda-web'
sed -i '/\/data/ s/cupid/xda-web/g'  /etc/rc.local

# 支持\r、\n、\t的替换，将所有\t转换为四个空格
sed -i 's/\t/    /g' /etc/rc.local
```

s命令可以使用任意分隔符作为定界符(即转义字符)

```
# 当替换路径时'/'作为定界符需要频繁转义，可以使用其他定界符，s后的第一个字符为定界符，此处示例使用#为定界符
# 将'/data/cupid'替换为'/data/xda-web/'
sed -i 's#/data/cupid#/data/xda-web/g' /etc/rc.local
```

多条s命令的执行

- 使用 -e 选项
- 使用 ; 分割

```
# -e 选择
sed -e 's/feng/fdy/' -e '/lxf/ s/500/200/' ip.txt

# ;分割命令
sed 's/feng/fdy/; /lxf/ s/500/200/' ip.txt

# {;}分割命令
sed '{s/feng/fdy/; /lxf/ s/500/200/}' ip.txt
```

&：模糊匹配内容修改

```
# 将fat和cat增加双引号

# 错误命令
echo "fat and cat" | sed 's/.at/".at"/g'

# 正确命令
echo "fat and cat" | sed 's/.at/"&"/g'
```

标签：正则表达式元组匹配修改

- 标签：sed使用圆括号定义替换模式的部分字符
- 标签可以方便在后面引用，每行指令最多使用9个标签
- \1 代表第一个圆括号定义的内容，\2 代表第二个，以此类推

```
# 把后面 .* 部分删除
sed -r 's/(^[0-z]+)(.*)/\1/' /etc/passwd

# 把前面 ^[0-z]+ 部分删除
sed -r 's/(^[0-z]+)(.*)/\2/' /etc/passwd

# 倒序输出
echo aaa bbb ccc | sed -r 's/([a-z]+) ([a-z]+) ([a-z]+)/\3 \2 \1/'
ccc bbb aaa

# 其实有些操作awk更方便
echo aaa bbb ccc| awk '{print $3,$2,$1}'
ccc bbb aaa
```

引用shell变量

```
# 双引号内引用shell变量
sed -i "/my is $name/d" /etc/rc.local

# 单引号外拼接引用shell变量
sed -i 'my is '$name'/d' /etc/rc.local
```

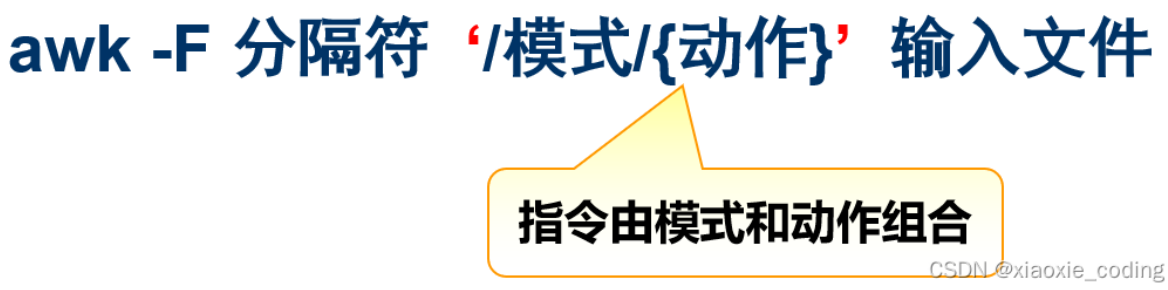
awk

语法说明

awk是一门**模式扫描和处理的编程语言**，主要用于**文本截取和分析**，支持流程控制和正则表达式，详细可参见

```
awk [ -- ] program-text file ...
awk -f program-file [ -- ] file ...
awk -e program-text [ -- ] file ...
```

常用格式




```
awk -F '分隔符' '/模式/{动作}'  输入文件
```

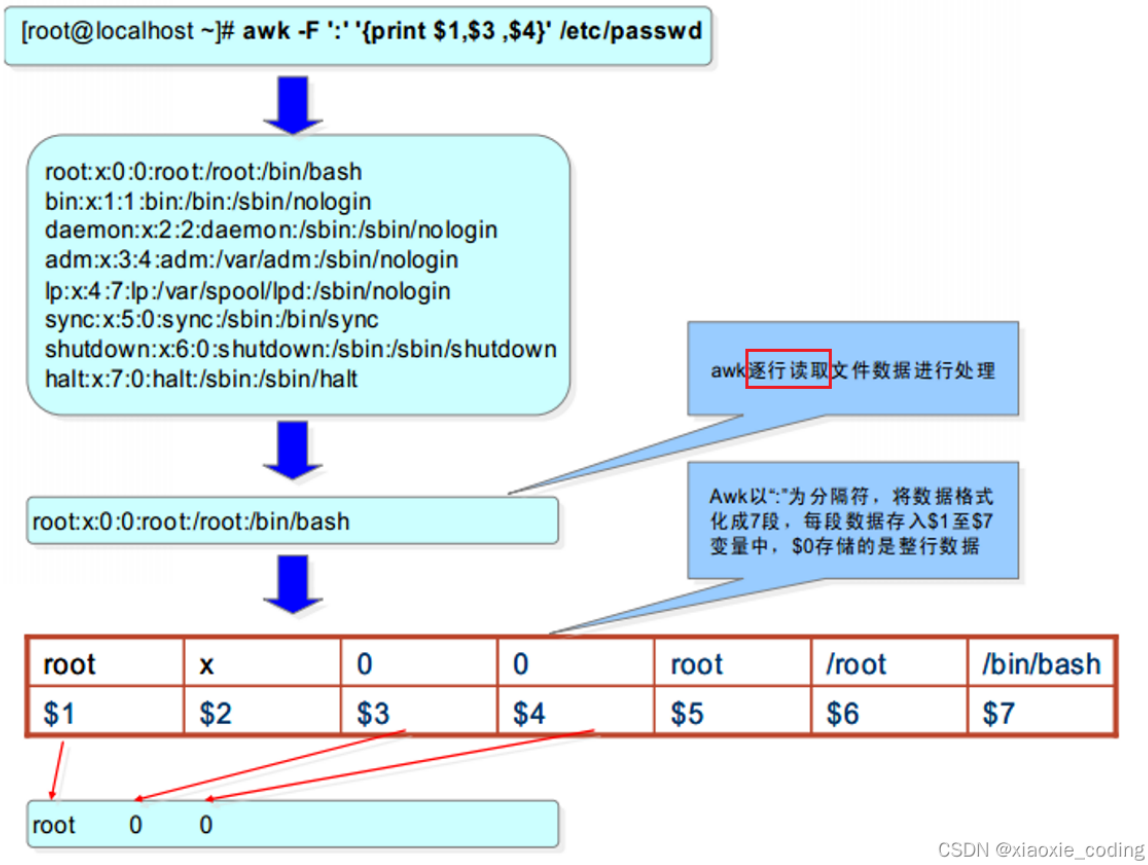
- 分隔符用于切分行字符串，可以使用正则表达式

- 指令包括模式和动作，**使用单引号引起，单引号内$不会被shell解析为变量**
- 模式可以使用正则表达式、条件表达式或两种组合
  - 正则表达式要用定界符 / 包裹
  - 条件表达式在下面单独说明
- 动作必须使用大括号引起，多个动作之间用 ; 分开
- 动作引用自定义变量，不需要接$符号

选项

- -F fs 指定输入字段分隔符(FS预定义变量也可设置)
- -v var=value 自定义变量，常用于引入shell中的变量
- -n 识别文件输入中的8进制数(0开头)和16进制数(0x开头)，例如：`echo '030' | awk -n '{print $1+0}'`
- -f program-file 指定读取程序的文件名
- -e program-text 指定awk程序表达式，可结合-f选项同时使用，在使用了-f选项后，如果不使用-e，awk program是不会执行的，它会被当作ARGV的一个参数
- -o [filename] 格式化awk代码。不指定filename时，则默认保存到awkprof.out，指定为`-`时，表示输出到标准输出

简要执行流程



分隔符

分隔符用于切分输入和拼接输出

- 输入分隔符
  - 行分隔符RS
  - 列分隔符FS、FIELDWIDTHS、FPAT

- 输出分隔符
  - 行分隔符RT
  - 列分隔符OFS

RS

RS用于指定输入记录分隔符，RS通常设置在BEGIN代码块中，因为要先于读取文件就确定好RS分隔符，RS支持使用正则表达式分割记录。

- RS="\n+" ： 默认匹配方式，按行读取，包含空行
- RS="^$"： 一次性读取所有数据
- RS="" ： 按段落读取

FS

FS以分隔符切分行，获取未匹配分隔符的内容构成字段，行字段以指定分隔符拼接时适合此分隔符。

分隔符支持正则表达式，分隔符包括输入分隔符FS、输出分隔符OFS，-F用于设置输入分隔符FS

- 默认分隔符为空格或TAB
- 默认设置的分隔符为输入分隔符
- 输出分隔符需要显性设置

```
# 以':'拆分输入行，输出结果以空格拼接
awk -F ':' '{OFS=" "} /gitlab/{print $1,$2,$3}' /etc/passwd
```

FIELDWIDTHS

FIELDWIDTHS以字符宽度读取行构成字段，行内容以固定宽度打印时适合此分隔符

```
# a.txt
ID  name    gender  age  email          phone

1   Bob     male    28   abc@qq.com     18023394012

2   Alice   female  24   def@gmail.com  18084925203

3   Tony    male    21   aaa@163.com    17048792503

4   Kevin   male    21   bbb@189.com    17023929033

5   Alex    male    18   ccc@xyz.com    18185904230

6   Andy    female  22   ddd@139.com    18923902352

7   Jerry   female  25   exdsa@189.com  18785234906

8   Peter   male    20   bax@qq.com     17729348758

9   Steven  female  23   bc@sohu.com    15947893212

10  Bruce   female  27   bcbd@139.com   13942943905

# 以字符长度读取,第一个字段去读4个字符，第二个字段读取8个字符，以此类推
```

```
awk 'BEGIN{FIELDWIDTHS="4 8 8 5 15 11"} length($0) > 1 && strtonum($1) > 0
{printf "行字符长度:%d ID:%d 姓名:%s  性别:%s  年龄:%s  邮箱:%s  电话:%s\r\n",
length($0),$1, $2, $3, $4, $5, $6}' a.txt
```

FPAT

FPAT是取得匹配的字符部分作为字段，当使用分隔符无法正确切分时，使用FPAT更合适

```
echo 'Robbins,Arnold,"1234 A Pretty Street, NE",MyTown,MyState,12345-6789,USA'
|\
awk '
    BEGIN{FPAT="[^,]*|(\"[^\"]*\")"}
    {
        for (i=1;i<NF;i++){
            print "<"$i">"
        }
    }
'
```

指令

执行过程：

- 执行begin的commands
- 每行匹配pattern，命中的执行后面的commands，partten和command组成的部分可存在多个
- 最后一下end的commands

模式（pattern）

模式用于过滤数据，模式匹配如下所示：

- BEGIN
- END
- BEGINFILE
- ENDFILE
- /regular expression/
- relational expression
- pattern && pattern
- pattern || pattern
- pattern ? pattern : pattern
- (pattern)
- ! pattern
- pattern1, pattern2

示例：

```
#开头和结尾增加打印输出
awk -F : 'BEGIN{print "-start-"} /bash/{print $1,$3} END{print "-end-"}'
/etc/passwd

# 统计并打印行数
awk 'BEGIN{i=0}{i++}END{print i}' /etc/passwd
```

正则表达式

正则匹配模式与grep类似，用于过滤行

```
# 以root开头的行
awk -F : '/^root/{print $0}' /etc/passwd
```

条件表达式

- 精确条件匹配模式： 使用比较运算符（==、>=等）进行过滤

```
# 过滤出切分后第一列为值为root的行
awk -F : '$1 == "root" {print NR,$0}' /etc/passwd
```

- 模糊条件匹配模式：使用'~'运算符
- \< 单词界定符，定义开头
- \> 单词界定符，定义结尾
- . 定义任意字符

```
# /etc/passwd用户名包含gitlab的行
awk -F : '$1 ~ /gitlab/ {print NR, $0}'  /etc/passwd

# /etc/passwd用户名以gitlab开头且以www结尾的行
awk -F ":" '$1 ~ /\<gitlab.*www\>/ {print $0}' /etc/passwd

# 所有TCP监听端口号
 netstat -antp | awk -F " +|:|::1|:::|/" '($1 == "tcp" || $1=="tcp6") && $8 ==
"LISTEN" {print $5}'
```

动作（action）

动作用于执行命令，动作内容使用大括号引起｛command｝，多个命令使用';'分割

```
# 打印/etc/passwd中用户名包含a的账号，且统计总数
awk -F: 'BEGIN{i=0} {if($1 ~/a/) {print $1;i++}} END{print "total="i}'
/etc/passwd
```

编程语言

变量

变量声明

awk在command中声明变量，无需定义变量类型，引用变量是也无需使用$符号，与javascript语法类似。

内置变量

内置变量名	含义
ARGC	arguments count ： 命令参数的总数
ARGIND	index in ARGV of the current file：当前参数的序号
ARGV	Array of command line arguments：包含命令参数的数组
BINMODE	Binary mode for all file I/O：二进制模式读取文件
CONVFMT	Conversion format for numbers：数字类型转换为字符串类型时的格式化字符串，默认为`%.6g`
ENVIRON	包含当前环境变量的数组，示例：ENVIRON["HOME"]
ERRNO	命令执行异常错误信息
FIELDWIDTHS	A whitespace separated list of field widths：使用字符宽度进行分割字段，
FILENAME	The name of the current input file：当前文件的文件名
FNR	record number in the current input file：每个文件的行号计数器
FPAT	A regular expression describing the contents of the fields in a record：
FS	The input field separator：输入字段分隔符， 默认为空白字符，即-F参数所设
IGNORECASE	Controls the case-sensitivity of all regular expression and string operations：控制所有正则模式匹配的大小写，`IGNORECASE=0`忽略大小写
NF	Number of Field in the current input record：当前行的列字段数计数器
NR	Number of Record：行数计数器，当多个文件时，为所有文件的行号计数器
OFMT	Output format for numbers：输出数字格式化，默认为`%.6g`
OFS	Output field separator：输出字段分隔符， 默认为一个空格
ORS	Output record separator：输出行分隔符，默认为`\n`
PROCINFO	
RS	Input Record Separator：输入行分隔符。`RS=""`以段落分行；`RS="^$"`读取所有数据为一行；`RS="\n+"`按行读取，忽略空行；
RT	Record Terminator：输出行分隔符，例如：`awk 'BEGIN{RS=":"}{print $0,RT}' /etc/passwd`
RSTART	内置函数match(s,r)，匹配命中字符串在整个字符串中的起始位置
RLENGTH	内置函数match(s,r)，匹配命中字符串的长度，例如：`awk 'BEGIN{ match("abcde", "bcd"); print RSTART,RLENGTH}'`
SUBSEP	复杂索引数组的分隔符，默认分隔符为八进制\034，例如：`awk 'BEGIN{ SUBSEP="#"; arr[1,2]=10; print arr["1#2"] }'`

示例：

```
# 显示行号、列数、第一列、最后一列
awk -F : '{print NR, NF, $1,$(NF-1)}' /etc/passwd

# 显示3-5行
awk -F : 'NR >=3 && NR <=5 {print NR,$0}' /etc/passwd

# 读取多个文件
awk -F ":" '{print "file_name="FILENAME, "NR="NR, "FNR="FNR,$0}' /etc/passwd
/etc/group

# ifconfig结果使用RS以段落分行，获取网卡名称
ifconfig  | awk -F " +|:" 'BEGIN{RS=""} {print NR, $1}'

# 循环打印每一个切分后的field
cat a.txt | awk -F " " '{ for(i=1;i<=NF;i++) { printf "第%d行 第%d列 值:%s\r\n",
NR, i, $i; } }'
```

变量类型

变量类型主要包括**数值**和**字符串**

类型转换

- 隐式转换
  - 算术运算符计算后转换为数值类型，**无效字符串将转换成0**
  - 数值连接空字符串转换为字符串类型

```
# 字符串转换为数字类型
awk 'BEGIN{ n = "123" + 1; print n }'
awk 'BEGIN{ print "abc" + 1 }'
awk 'BEGIN{ print "123abc" + 1 }'

# 数字转换为字符串类型
awk 'BEGIN{ print 123"abc" }'
```

- 显式转换
  - 数值 转 字符串： 使用CONVFMT或sprintf()函数，CONVFMT默认格式化时值保留6位小数

```
# 使用CONVFMT
awk 'BEGIN{a=123.4567;CONVFMT="%.2f";print a""}' #123.46

# sprintf()函数
awk 'BEGIN{a=123.4567;print sprintf("%.2f", a)}' #123.46

# printf()函数
awk 'BEGIN{a=123.4567;printf("%.2f",a)}'
```

  - 字符串 转 数值：strtonum()函数

```
gawk 'BEGIN{a="123.4567";print strtonum(a)}'
```

变量打印

```
属性方法
    OFS          输出格式的列分隔符，缺省是空格
    ORS          输出记录分隔符,输出时用指定符号代替换行符

print 函数
    print [item1,item2,...]

printf 函数
    printf [-v var] format [item1,item2,...]
    注意：
        printf输出需要指定换行符号，format的格式必须与后面item对应
        常见格式：
            %c       显示字符的ASCII码
            %d|i     显示十进制整数
            %e|E     显示科学计数法数值
            %u       显示无符号整数
            %f       显示浮点数
            %s       显示字符串
            %%       显示%本身
        修饰符：
            %#[.#]   第一个#控制显示宽度，第二个#表示小数点后的精度，例如%3.1f
            %-       左对齐，%-15s
            %+       显示数值的正负符号，%+d
```

示例：

```
awk -F ':' -v OFS='#' '{print NR,NF,$1}' /etc/passwd
```

数组

awk的数组是关联数组(即key/value方式的hash数据结构)，索引下标可为数值(甚至是负数、小数等)，也可为字符串，**其实是map**

- awk数组的索引实际都是字符串，即使是数值索引在使用时内部也会转换成字符串
- awk的数组元素的遍历顺序和插入顺序很可能是不同的

定义

```
awk 'BEGIN{
    arr[1]= 11
    arr[-1]  = -11 ;
    arr[4.3] = 4.33 ;
    arr["1"] = 111  ;
    arr["a"] = "aa"  ;
    arr["x","y"] = 123 ;
    print arr[1];
}'
```

长度

`length()`函数获取数组长度

```
awk 'BEGIN{
    arr[1]= 11;
    arr[2]  = -11 ;
    arr[3] = 4.33 ;
    arr[4] = 111  ;
    arr[5] = "aa"  ;
    arr[6] = 123 ;
    print length(arr);
}'
```

删除

- 删除元素： `delete arr[idx]` ： 删除数组`arr[idx]`元素，允许删除不存在的元素
- 删除数组：`delete arr`

判断类型

`isarray(arr)`可用于检测arr是否是数组，如果是数组则返回1，否则返回0

判断元素是否存在

```
# 错误示例
if(arr["x"] != ""){...}
```

- 如果不存在arr[“x”]，则会立即创建该元素，并将其值设置为空字符串
- 有些元素的值本身就是空字符串

```
# 正确示例：使用'in',如果存在则返回1，不存在则返回0
if (i in arr){...}
```

遍历数组

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

awk 'BEGIN{
    arr[1]= 11
    arr[2]  = -11 ;
    arr[3] = 4.33 ;
    arr[4] = 111  ;
    arr[5] = "aa"  ;
    arr[6] = 123 ;
    for (i in arr) {print i};
}'
```

运算符

运算符列表如下，运算符优先级参考C语言。

运算符名称	含义
(...)	运算组
$	取值
in	数组成员
space	字符串拼接，例如：`12 " " 23`
?:	三目运算符
+ - * / %	算术运算符：加、减、乘、除、求余
++ --	算术运算符：自加、自减，支持前置、后置
^	算术运算符：幂
&& || !	逻辑运算符：与、或、非
< > <= >= != ==	比较运算符：小于、大于、小于等于、大于等于、不等于、等于
~ !~	正则匹配运算符：正则匹配、正则匹配取反，返回值：1为匹配成功，0为匹配失败
= += -= *= /= %= ^=	赋值运算符

示例：

```
awk 'BEGIN{
    a=(10+2-3)*4; print a;
    a=a/5; print a;
    a=a%5; print a;
    a=int(a%5); print a;
    a=2^10; print a;
    a=log(100); print a;
    print (a > 0 ? "true" : "false");
    b="a1b2c3";print b ~ "b2";
}'
```

流程控制

- if (condition) statement [else if (condition) statement] [else statement]
- while (condition) statement
- do statement while (condition)
- for (expr1; expr2; expr3) statement
- for (var in array) statement
- break
- continue
- **next ：读取下一行并附带continue动作**
- **nextfile ：读取下一个文件**
- exit [expression] ： 退出awk程序
- { statements }
- switch (expression) {

  case value|regex : statement

  ...

  [default: statement]

  }

```
# if : 单行输出1，双行输出2
awk -F :  '{if(NR % 2 == 0) print NR,2; else print NR,1}' /etc/passwd

# while循环

# for循环
awk -F: '{for(i=10;i>0;i--){print $0}}' /etc/passwd
```

内置函数

数字函数

- atan2(y, x) 求y/x的反正切值
- cos(expr) 求余弦
- exp(expr) 求指数

- int(expr)　数字截断转换为整数
- log(expr)　求对数.
- rand()　生成≤0且< 1的随机数
- sin(expr)　求正弦值
- sqrt(expr)　请平方根
- srand([expr]) 获取随机数，以expr作为随机数生成的种子，如果未设置以当前时间作为种子

字符串函数

- **asort**(s [, d [, how]])
  返回源数组 s 中的元素数量。使用 gawk 比较值的正常规则对 s 的内容进行排序，并将排序值 s 的索引替换为以下开头的连续整数1. 如果指定了可选目标数组d，则首先将s复制到d中，然后对d进行排序，保持源数组 s 的索引不变。可选字符串how控制方向和比较模式。
- **asorti**(s [, d [, how]])
  返回源数组 s 中的元素数量。其行为与 asort() 相同，只是使用数组索引进行排序，而不是数组值。完成后，数组将以数字方式索引，并且值是原始索引的值。原始值丢失；因此，如果您希望保留原始数组，请提供第二个数组。可选字符串how的用途与asort()相同
- **gensub(**r**,** *s***,** *h* **[,** *t***])**

  在目标字符串 *t* 中搜索正则表达式 *r* 的匹配项。如果 *h* 是以 **g** 或 **G** 开头的字符串，则将 *r* 的所有匹配项替换为 *s* 。否则，*h* 是一个数字，指示要替换 *r* 的哪个匹配项。如果未提供 *t*，请使用 **$0** 代替。在替换文本 *s* 中，序列 **n**,，其中 *n* 是从 1 到 9 的数字，可以用于指示仅与第 *n* 个括号子表达式匹配的文本。序列 **\0** 代表整个匹配文本，字符 **&** 也是如此。与**sub()**和**gsub()**不同，修改后的字符串作为函数的结果返回，而原始目标字符串*没有*改变。
- **gsub(**r**,** *s* **[,** *t***])**

  对于字符串*t*中与正则表达式*r*匹配的每个子字符串，替换字符串*s*，并返回替换次数。如果未提供 *t*，请使用**$0**。替换文本中的 **&** 将替换为实际匹配的文本。使用 **&** 获取文字 **&**。
- **index(**s**,** *t***)**

  返回字符串 *s* 中字符串 *t* 的索引，如果 *t* 不存在则返回0。 （这意味着字符索引从 1 开始。）
- **length(**[*s*]**)**

  返回字符串 *s* 的长度，如果未提供 *s*，则返回 **$0** 的长度。对于数组参数，**length()** 返回数组中的元素数量
- **match(**s**,** *r* **[**, *a***]**)

  s是代表字符串，r代表正则表达式，match的作用是返回匹配r的子串在s中的首个位置
- **patsplit(**s**,** *a* [**,** *r* [**,** *seps*]]**)**

  根据正则表达式*r*将字符串*s*拆分为数组*a*和分隔符数组*seps*，并返回字段数量。元素值是 *s* 中与 *r* 匹配的部分。 **seps[*i*] 的值是 a[*i* 之后出现的可能为空的分隔符]. seps[0] 的值是可能为空的前导分隔符。如果省略*r*，则使用**FPAT**。首先清除数组*a*和*seps*。拆分的行为与使用 **FPAT** 进行字段拆分的行为相同
- **split(**s**,** *a* [**,** *r* [**,** *seps*]]**)**

  根据正则表达式*r*将字符串*s*拆分为数组*a*和分隔符数组*seps*，并返回字段数量。如果省略*r*，则使用**FS**。首先清除数组*a*和*seps*。 **seps[*i*] 是 a[< 之间与 *r* 匹配的字段分隔符em>i] 和 a[*i*+1]。** 拆分的行为与字段拆分相同
- **sprintf(**fmt**,** *expr-list***)**

  根据*fmt*打印*expr-list*，并返回结果字符串
- **strtonum(**str**)**

检查*str*，并返回其数值。如果 *str* 以前导 **0** 开头，则将其视为八进制数。如果*str*以前导**0x**或**0X**开头，则将其视为十六进制数。否则，假设它是十进制数

- **sub(**r**,** *s* [**,** *t*]**)**

  就像**gsub()**一样，但只替换第一个匹配的子字符串。返回零或一。

- **substr(**s**,** *i* [**,** *n*]**)**

  返回从 *i* 开始的 *s* 中最多 *n* 个字符的子字符串。如果省略*n*，则使用其余的*s*。

- **tolower(**str**)**

  返回字符串 *str* 的副本，其中 *str* 中的所有大写字符均转换为相应的小写字符。非字母字符保持不变。

- **toupper(**str**)**

  返回字符串 *str* 的副本，其中 *str* 中的所有小写字符均转换为相应的大写字符。非字母字符保持不变。

时间函数

- **mktime(**datespec [**,** *utc-flag*]**)**

  将*datespec*转换为与**systime()**返回的形式相同的时间戳，并返回结果。如果 utc-flag 存在且非零或非空，则假定时间位于 UTC 时区；否则，时间被假定为本地时区。如果*datespec*不包含足够的元素或者结果时间超出范围，**mktime()**返回-1

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

  根据*格式*中的规范设置*时间戳*的格式。如果 utc-flag 存在且非零或非空，则结果采用 UTC 格式，否则结果采用当地时间。 *时间戳*应该与**systime()**返回的形式相同。如果*时间戳*丢失，则使用当前时间。如果缺少 *format*，则使用与 *date*(1) 的输出等效的默认格式。默认格式可在 **PROCINFO["strftime"]** 中找到。请参阅 ISO C 中的 **strftime()** 函数规范，了解保证可用的格式转换

- **systime()**

  返回当前时间作为自纪元（POSIX 系统上的 1970-01-01 00:00:00 UTC）以来的秒数

位操作函数

- **and(*v1*,** *v2* [, ...]**)**

  返回参数列表中提供的值的按位与。必须至少有两个。

- **compl(*val*)**

  返回 *val* 的按位补码。

- **lshift(*val*,** *count***)**

  返回 *val* 的值，左移 *count* 位。

- **or(*v1*,** *v2* [, ...]**)**

  返回参数列表中提供的值的按位或。必须至少有两个。

- **rshift(*val*,** *count***)**

  返回 *val* 的值，右移 *count* 位。

- **xor(*v1*,** *v2* [, ...]**)**

  返回参数列表中提供的值的按位异或。必须至少有两个

功能函数

- **isarray(**x**)**

  如果 *x* 是数组，则返回 true，否则返回 false

- **typeof**(x)　**（注：4.2及以上版本持此函数）**

  返回一个字符串，指示 *x* 的类型。该字符串将为 "array"、"number"、"regexp"、"string" 之一，“strnum”、“未分配”或“未定义”

自定义函数

函数定义

```
# 声明函数
function name(parameter list) { statements }

# 引用函数
@include "filename" pattern   { action statements }
```

函数参数

- 函数参数声明类型与传递的值类型不一致将报错
- 数组参数是引用传递，其他为值传递

函数返回值

函数返回值使用return语句

函数变量作用域

与shell脚本类似，函数内变量的作用域为全局作用域

linux分区修复

查看分区信息

```
# 查看分区列表
cat /etc/fstab

# 查看挂载情况
df -h

# 查看硬盘分区的详细信息
fdisk -l

# 查看硬盘分区的概要信息
lsblk
```

```
#查看文件系统类型信息（包含UUID）
blkid
```

修复分区

```
# 卸载分区
umount 分区名

# 如果非逻辑分区使用fsck
fsck -y 分区名

# 如果使用的XFS文件系统
xfs_check 分区名
xfs_repair 分区名
```

Linux信息获取

```
# 系统启动时间
uptime -s

# 获取硬盘序列号（虚拟机无法获取）
lsblk --nodeps -no serial /dev/sda

# 查看设备序列号
dmidecode -t 1
```