1、基础入门

ping主机是否可用

```
ping -c1 www.baidu.com &>/dev/null && echo "wwww.baidu.com is ok!" || echo
"www.baidu.com is down!"
```

-c1 : 表示只ping一次

&> : 表示将ping的输出重定向到/dev/null， 也即是不输出

&& ： 表示前面的命令执行成功后执行

|| ： 表示前面的命令执行失败后执行

vim /opt/scripts/ping01.sh

```
#!/usr/bin/bash
ping -c1 www.baidu.com &>/dev/null && echo "wwww.baidu.com is ok!" || echo
"www.baidu.com is down!"

#在shell中调用python程序.使用<<-EOF python代码 EOF
/usr/bin/python <<-EOF
print("python")
EOF

# 又操作shell
echo "shell"
```

快捷键

CTRL + D : 退出

CTRL + R : 搜索历史命令

CTRL + A : 光标到行首

CTRL + E: 光标到行尾

CTRL + U : 删除光标前面的内容

CTRL + K : 删除光标后面的内容

tty: 查看终端

元字符

* ：表示任意字条

？ ： 表示任意一个字符

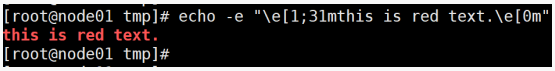
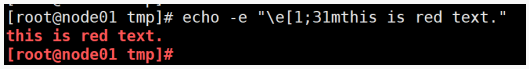
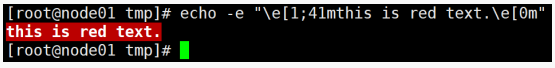
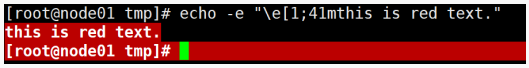
() ： 在子shell中执行 （cd /home; ls）

{} ：表示集合 。 cp id_dsa.pub id_dsa.pub.old 可以简写成：cp id_dsa.pub{,.old}

颜色输出

```
echo -e "\e[1;31m输出内容\e[0m"

文字的颜色是30~37
背景的颜色是40~47
```

<table>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>\e[0m:清除后面的颜色</td><td></td></tr>
<tr><td> 
</td><td> 
</td></tr>
</table>

位置变量

> $0 , $1, $2, $3

vim /opt/scripts/ping05.sh

```
#!/usr/bin/bash
ping -c1 $1 &>/dev/null

if [ $? -eq 0 ] ;then
        echo "$1 is ok."
else
        echo "$1 is down."
fi
```

执行： ./ping05.sh www.baidu.com ， $1就是www.baidu.com

自定义变量只在当前shell中生效；环境变量在当前shell及子shell中生效。

环境变量可以通过`env`命令来查看

变量替换

```
[root@node01 scripts]# t=`date +%F`
[root@node01 scripts]# echo $t
2022-03-05
```

" " : 强引用，会解析里面的变量

'' : 弱引用，不会解析里面的变量

`` : 反引号

```
# 2022-03-05_log.log
touch `date +%F`_log.log
```

变量的运算

1、 第一种方式：**expr**

```
[root@node01 scripts]# echo 1 + 3
1 + 3
[root@node01 scripts]# expr 1+3
1+3
[root@node01 scripts]# expr 1 + 3
4
```

2、第二种方式： **$(())**

```
[root@node01 scripts]# echo $((1 + 3))
4
[root@node01 scripts]# num1=10
[root@node01 scripts]# num2=20
[root@node01 scripts]# echo $((num1+num2))
30
```

3、第三种方式：**$[]**

```
[root@node01 scripts]# num1=10
[root@node01 scripts]# num2=20
[root@node01 scripts]# echo $[num1+num2]
30
```

4、第4种方式：**let**

```
[root@node01 scripts]# let num=num1+num2
[root@node01 scripts]# echo $num
30
```

小数的运算

```
echo "2*4"|bc
echo "2^4"|bc
echo "scale=2;6/4"|bc
awk 'BEGIN{print 1/2}'
echo "print (5.0/2)" |python
```

变量内容的替换和删除

#： 从前往后，最短匹配

##: 从前往后，贪婪匹配

```
[root@node01 scripts]# url=www.sina.com.cn
# 1个#表示从前往后删除，直到第一个.为止
[root@node01 scripts]# echo ${url#*.}
sina.com.cn
# 2个##也表示从前往后删，直到最后一个.为止
[root@node01 scripts]# echo ${url##*.}
cn
# 删除www
[root@node01 scripts]# echo ${url#www.}
sina.com.cn
# 删除sina.前面所有的内容
[root@node01 scripts]# echo ${url#*sina.}
com.cn
```

%： 从后往前数，最短匹配

%% ： 从后往前数，贪婪匹配

```
[root@node01 scripts]# echo ${url}
www.sina.com.cn
[root@node01 scripts]# echo ${url%%.*}
www
[root@node01 scripts]# echo ${url%.*}
www.sina.com
```

索引的切片

```
[root@node01 scripts]# echo ${url}
www.sina.com.cn
[root@node01 scripts]# echo ${url:2:6}  # 从索引2开始，往后切6个
w.sina
[root@node01 scripts]# echo ${url:2}  # 从索引2往后切完
w.sina.com.cn
```

替换

${变量/旧的内容/替换内容}

```
[root@node01 scripts]# echo ${url/sina/baidu}
www.baidu.com.cn
```

${变量名-新的变量值}

变量没有被赋值：会使用"新的变量值"替代

变量有被赋值(包括空值)： 不会被替代

```
# 取消var变量的赋值
[root@node01 scripts]# unset var
[root@node01 scripts]# echo ${var-aaaa}
aaaa
[root@node01 scripts]# var2=222
[root@node01 scripts]# echo ${var2-bbbb}
222
# var定义了一个空值
[root@node01 scripts]# var3=
[root@node01 scripts]# echo ${var3-cccc}

[root@node01 scripts]#
```

${变量名:-新的变量值}

变量没有被赋值(包括空值)：会使用"新的变量值"替代

变量有被赋值： 不会被替代

() 子shell中运行

(()) 数字比较，运算

$() 命令替换， 相当于``

$(()) 整数运算

{} : 集合

${} : 变量替换

[] 条件测试

[[]] 条件测试，支持正则， 它比[] 强， =~ ： 表示正则匹配

$[] 整数运算

执行脚本

./01.sh 需要执行权限， 在子shell中执行

bash 01.sh 不需要执行权限， 在子shell中执行

. 01.sh　不需要执行权限，在当前shell中执行

source 01.sh　不需要执行权限，在当前shell中执行

调试脚本

sh -n 01.sh　仅调试语言错误，syntax error

sh -vx 01.sh　以调试的方式执行，查询整个执行过程

2、条件测试

> 条件测试就是看条件是不是成立

格式1：test 条件表达式

格式2：[条件表达式]

格式3：[[条件表达式]]

帮助信息使用`man test`查看

```
# test -d 判断是不是目录
if test -d /home ; then
    echo "this is a dir"
else
    echo "this is not a dir"
fi
```

 # vim ping101.sh

```
#!/usr/bin/bash
# 清空ip.txt这个文件中的内容
>ip.txt

# 同步执行
for i in 192.168.79.{3..254}
do
        # {}& 会开启一个后台子shell去执行{}中的内容
        {
        ping -c1 -W1 $i &>/dev/null
        if [ $? -ne 0 ];then
                # 创建一个文件ip.txt并将管道前面的内容apppend进去
                echo "$i" | tee -a ip.txt
        fi
        }&

done

# 等待前面所有的后台shell进程结束
wait
```

```
echo "ping finished"
```

替换文件内容

sed -ri '/原数据/c替换数据' 文件

3、while循环

 # vim /opt/scripts/day3/while_create_user.sh

```
#!/usr/bin/bash
# 使用while创建用户

# 读数据，user接收
# while读取文件时以回车分隔
while read user
do
        echo $user
#将user2.txt文件中的内容重定向给while循环
done < user2.txt
```

所以，处理文件时，使用while更合适；

4、for循环

 # vim /opt/scripts/day4/for_test.sh

```
#!/usr/bin/bash
# for i in $@ 或者是 for i in #*, 下面的for更简略
for i
do
        let sum+=$i
done
echo $sum
```

调用脚本及输出

[root@node01 day4]# ./for_test.sh 1 2 3 4
10

总结： for i 默认就会接收脚本所有的位置参数；它就相当于 `for i in $@` 或者是 `for i in $*`

5、并发控制

管道

匿名管道

[root@node01 day3]# ls | grep sh

命名管道

创建 一个命名管道

[root@node01 day3]# mkfifo /tmp/fifi01

查看类型

[root@node01 day3]# file /tmp/fifi01
/tmp/fifi01: fifo (named pipe)

往管道中写数据

[root@node01 ~]# ls /dev/ > /tmp/fifi01

查看管道中的数据

[root@node01 day3]# cat /tmp/fifi01

 # vim /opt/scripts/day3/ping_multi_thread.sh

```
#!/usr/bin/bash

for ip in 192.168.79.{1..254}
do
        {
        ping -c1 -W1 $ip &>/dev/null
        if [ $? -eq 0 ] ;then
                echo "$ip is up."
        else
                echo "$ip is down."
        fi
        }&
done
wait
echo "all finish...."
```

这个示例中，每次do...done循环中就会创建一个{}&后台进程， 所以此处的并发是254

 # vim /opt/scripts/day3/ping_multi_thread2.sh

```
#!/usr/bin/bash
# 定义并发数量
thread=5
# 定义一个管道
tmp_fifo=/tmp/$$.fifo
# 创建管道
mkfifo $tmp_fifo
# 打开管道,且将其fb定义为8，以后操作8就相当于操作管道tmp_fifo
exec 8<> $tmp_fifo
# 删除管道，但是因为exec打开了管道，所以管道的fd还存在
rm $tmp_fifo
```

```
for i in `seq $thread`
do
        # 将回车符写到fd8中，即是写到管道中。
        # echo 本身就是个回车符
        echo >&8
done


for ip in 192.168.79.{1..254}
do
        # 从管道中读取数据，读到数据就执行后面的逻辑，如果没有从管道中读到数据，就会阻塞
        # read -u 后面跟一个fd ,所以这里不需要使用&8
        read -u 8
        {
        ping -c1 -W1 $ip &>/dev/null
        if [ $? -eq 0 ] ;then
                echo "$ip is up."
        else
                echo "$ip is down."
        fi
        echo >&8
        }&
done
wait
# 释放文件描述符
exec 8>&-
echo "all finish...."
```

此示例，并发控制数为5。

6、数组

普通数组

books=(linux shell java python)，取值： ${books[0]}

关联数组(相当于python中的字典,所以它没有下标),shell默认不支持关联数组，所以在使用前需要先申明这个关联数组

declare -A array1

array1=([name]=admin [sex]=male [age]=34)

取值 echo ${array1[name]}

```
[root@node01 day3]# declare -A info
[root@node01 day3]# info=([name]=admin [age]=36)
[root@node01 day3]# echo ${info[age]}
36
```

可以通过`declare -a`查看系统中的普通数组

可以通过`declare -A`查看系统中的关联数组

echo ${array[0]} 访问数组中的第一个元素

echo ${array[@]} 访问数组中的所有元素，相当于echo ${array[*]}

echo ${array[#]} 查看数组中的元素个数

echo ${!array[@]} 获取数组中的索引

echo ${array[@]:1} 从数组下标1开始遍历

echo ${array[@]:1:2} 从数组下标1开始，访问两个元素

示例

性别统计,男女各有多少人

原数据:

vim sex.txt

```
qeiquwe f
hfdkh m
dkhfdkhf f
dfkhdk f
dkfhdkh f
iwirw m
```

一条命令搞定

```
[root@node01 day3]# awk '{print $2}' sex.txt  |sort|uniq -c
      4 f
      2 m
```

使用数组来实现

```
#!/usr/bin/bash

# 定义一个关联数组sex
declare -A sex

# 使用while循环读sex.txt文件中的数据
while read line
do
    # 注意这里，每次都搞忘
    type=`echo $line | awk '{print $2}'`
    let sex[$type]++
done < sex.txt

# 关联数组的遍历，关联数组的索引就是字典的key值！~
for i in ${!sex[@]}
do
    echo "$i:${sex[$i]}"
done
```

`watch -n1 ./xxx.sh` **可以每隔1秒监控这个脚本的运行**

`unset sex` 清空数组

7、函数

传参： $1, $2

变量： local

返回值： return $?

1、定义函数

函数名（）{

}

function 函数名{

}

2、调用函数

函数名

函数名 参数1 参数2

示例

 # vim /opt/scripts/day4/factorial.sh

```
#!/usr/bin/bash
# 求阶乘

factorial(){
        ret=1
        # 这个$1是函数的位置参数
        for i in `seq $1`
        do
                #ret=$[ret * $i]
                let ret*=$i
        done
        echo "$1的阶乘是:$ret"
}

# $1, $2, $3是脚本的位置参数
factorial $1
factorial $2
factorial $3
```

示例： 函数返回值

```
#!/usr/bin/bash

func(){
    read -p "Number: "  num
    #通过echo来输出的函数的返回值
    #return 返回值的范围0~255,超过255只能使用echo来输出
    echo $[2*$num]
}
# 接收函数的返回值
result=`func`
echo "结果:$result"
```

```
#!/usr/bin/bash
#如果不是3个参数，就退出程序
if [ $# -ne 3 ] ;then
        echo "usage: `basenane $0` par1 par2 par3 "
        exit
fi

func(){
        echo $[$1 * $2 * $3]
}
# =前后不能有空格
result=`func $1 $2 $3`

echo "result is : $result"
```

```
#!/usr/bin/bash
num=(1 2 3 3 4 5)

func(){
        # 加了local,变量只在函数内部生效
        local factorial=1
        # for i in $*这样也行，但是for i in "$*"不行
        for i in $@
        do
                let factorial*=$i
        done
        echo $factorial
}

result=`func ${num[@]}`
echo $result
```

```
#!/usr/bin/bash
arr=(1 2 3)
func(){
        # 通过($@)来接收函数传过来的数据，并将它重新赋给新的变量newarr,此处使用`$@`不行，只能使用小括号
        local newarr=($@)
        local i
        # $# 表示函数接收到参数个数
```

```
        for((i=0;i<$#;i++))
        do
                newarr[$i]=$[${newarr[$i]} * 5]
        done
        echo "newarr is : ${newarr[@]}"

}
# 函数调用，传一个数组进去
func ${arr[@]}
```

```
#!/usr/bin/bash
arr=(1 2 3)

func(){
        local newarr=()
        local j
        for i in $*
        do
                newarr[j++]=$[$i * 5]
        done
        # 这里只能这样操作，使用$newarr不行，就像入参不能使用${arr}一样的道理
        echo ${newarr[@]}

}
result=`func ${arr[@]}`
echo "result is $result"
```

shift的使用

vim shift.sh

```
#!/usr/bin/bash
while [ $# -ne 0 ]
do
        # $1就是第一个位置的变量
        let sum+=$1
        # shift 参数位置往左移， 1 可以省略
        shift 1
done
echo $sum
```

使用： ./shift.sh 1 2 3 9

第一次while循环时，$#==4 ,$1==1 , 进入循环体后，shift一次后，结束本次循环；第二次进入while循环，此时$# ==3 ,$1==2,再次进入循环体... ,直到shift4次之后，$#==0 ,跳出循环，结束。

8、正则表达式

9、grep家族

grep [选项] 正则 filename1 filename2...

找到： grep的退出状态是0

找不到： grep 的退出状态是1

找不到指定文件： grep的退出状态是2

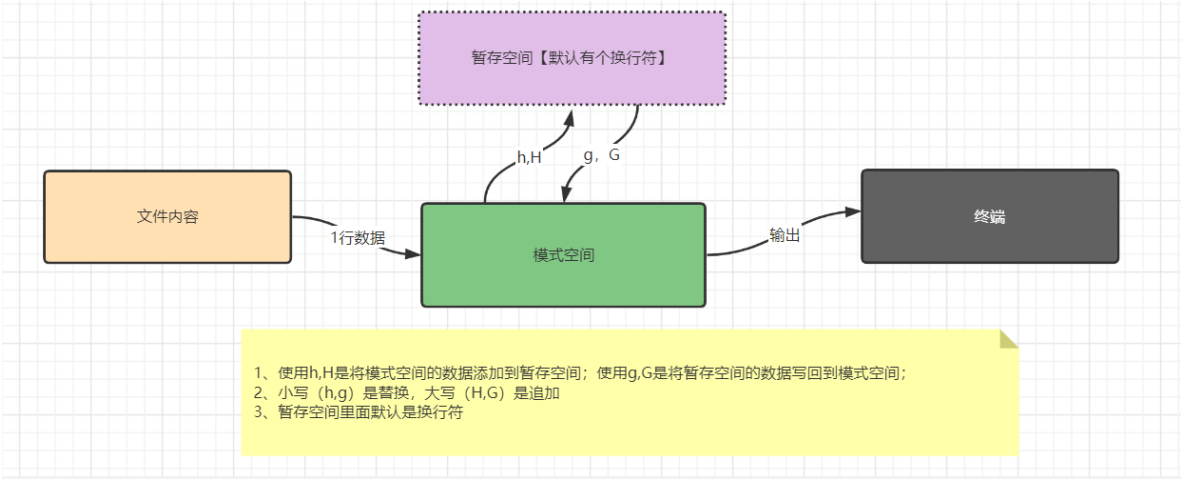
grep -q "root" /etc/passwd ; echo $?

-q : 不输出找到的内容

grep '^#' file ： 找到 以#开头的行， -v 取反

10、sed

sed 有两个空间需要特别注意： 模式空间、暂存空间



模式空间、暂存空间示例

1、`[root@node01 day4]# sed -r '1h;$G' passwd`

将第1行的数据由模式空间添加到暂存空间，然后再将暂存空间的数据添加到模式空间最后一行的后面

2、`[root@node01 day4]# sed -r 'g' passwd`

模式空间的每一行数据都由暂存空间的数据来进行替换，因为暂存空间默认是换行符，所以替换之后在终端输出的都是换行符

3、`[root@node01 day4]# sed -r 'G' passwd`

模式空间每一行数据的后面追加暂存空间的数据(换行符)，最后在终端进行输出

4、`sed -r '1{h;d};$G' passwd`

将模式空间的第一行数据添加到暂存空间去，然后删除掉第1行数据，最后再将暂存空间中的数据添加到模式空间最后一条数据的后面， 所以，最终结果是将第一行数据移动到了最后一行进行输出

5、`sed -r '1h;2,$g' passwd`

将第一行放到暂存空间，然后将第2行到最后一行的数据用暂存空间的数据进行覆盖，即是都变成了第1行

6、`sed -r '1h;2,3H;$G' passwd`

将模式空间的第1行覆盖暂存空间，然后再将模式空间2，3行的数据追加到暂存空间去，最后将暂存空间的数据追加到模式空间最后一行后面，即是将1，2，3的数据添加到原数据的后面。

注意：暂存空间原本有个换行符，所以不能：`sed -r '1,3H;$G' passwd`

暂存空间和模式空间互换命令：x

7、`sed -r '4h;5x;6G' passwd`

模式空间第4行覆盖暂存空间的数据，此时暂存空间是第4行的数据；5x: 将模式空间和暂存空间的数据进行互换，所以此时终端会再次打印第4行的数据，但是此时暂存空间是第5行的数据；6G：模式空间再读到第6行数据时，会把暂存空间的数据（第5行）追加到后面，最后再终端进行输出。

反向选择：！

8、`sed -r '3d' passwd`

删除第3行

9、`sed -r '3!d' passwd`

只有第3行不删除

多重编译模式：e

10、`sed -r -e '1,3d' -e 's/mail/e-mail/g' passwd`

执行了两条命令： 第一条命令是将1~3行进行删除；第2条命令是全局将mail 修改成了 e-mail

上面这条命令等价于：`sed -r '1,3d;s/mail/e-mail/g' passwd`

11、`sed -r '1s/root/ROOT/g; 1s/bash/BASH/g' passwd`

将第1行的root换成ROOT,将第1行的bash换成BASH;

上面的命令等价于：`sed -r '1{s/root/ROOT/g;s/bash/BASH/g}' passwd`

12、`sed -r '$a\123' passed`

$: 表示最后一行

a: 表示追加

最终就是在passwd的最后一行后面追加123

13、`sed -r '/^bin/cAAAAA' passwd`

c: 整行替换

上面命令的含义就是将以bin开头的替换成AAAAA。注意c与s的含义不一样，命令：`sed -r 's/^bin/AAAAA/' passwd` 只找到以bin开头的行，并将bin替换成AAAAA

14、给文件行添加注释

`sed -r 's/^/#/' passwd` ：将行的开头替换成#

`sed -r 's/(.*)/#\1/' passwd`：(.*)表示整个行；#\1 表示将前面匹配的数据前面加个#

`sed -r '2,5s/(.*)/#\1/' passwd` ：将2~5行的前面加个#

`sed -r '2,5s/.*/#&/' passwd`：跟上面的命令是一行的效果；&：表示前面（.*）正则匹配的数据内容

`sed -r 's/^#*/#/' passwd` ：将以零个#或多个#开头的行替换成1个# 。 ^#* ： 以0个#开头或多个#开头的

sed引用外部变量

15、`sed -r '$a'"$var" passwd`

给最后一行后面追加var变量的值， '$a'： 表示最后一行，本来应该是'$a$var', 但单引号中$var不会解析数据，所以只能使用双引号，但是双引号去解析$var, 但是sed命令又不能识别双引号， 所以只能将sed命令中的`$a` 用单引号分隔出来；或者是将最后一行的$转义一下，命令：`sed -r "\$a$var" passwd`

`sed -r "2a$var" passwd` ：在第2行后面追加var变量的值

额外知识：

`cat passwd` 正序查看一个文件内容； `tac passwd` 倒序查看一个文件的内容； `rev passed` 反向打印每一行的内容；

11、awk

awk [options] 'commands' filename

awk 命令格式

awk 'pattern' filename : `awk '/^root/' passwd`

awk '{action}' filename : `awk '{print $1}' passwd`

awk 'pattern {action}' filename ： `awk '/^root/ {print $1, $2}' passwd`

command | awk 'pattern {action}' : `df -P | grep '^/' | awk '$4>858196{print $1}'`

awk 内部变量

NR： 行数记录

FNR：按文件区分行数记录

FS：字段分隔符

NF：字段总列数记录

RS: 记录分隔符，默认是换行符。可以在BEGIN{RS=“ ”} 进行修改，这样就可以将一行数据分隔多行

ORS: 输出分隔符，默认也是换行符。 可以在BEGIN{ORS=“ ”} 进行修改，这样就可以将多行数据合并成一行

awk 打印输出

1、可以使用print进行打印输出；

2、可以使用printf进行格式打印输出;

`awk -F: '{printf "%-15s %-10s %-10s\n", $1,$2,$3}' passwd` : %-15 : 表示占15个长度；%-10: 表示占10个长度；s:字符串；\n:表示换行，printf默认不换行；

```
root            x          0
sync            x          5
shutdown        x          6
halt            x          7
mail            x          8
operator        x          11
games           x          12
ftp             x          14
nobody          x          99
```

%s： 字符类型

%d: 数字类型

%f： 浮点类型

-: 表示左对齐，默认是右对齐

awk 模式和动作

模式 : 可以是条件语句或复合语句 ， 或者正则表达式

一、正则表达式

匹配整行操作

1、`awk '/^alice/' passwd`

2、 `awk '$0~/^alice/' passwd` ： $0匹配以alice开头的记录，它等价于`awk '/^alice/' passwd`

3、 `awk '!/^alice/' passwd` : 不是以alice开头的记录

4、 `awk '$0!~/^alice/' passwd` ： $0不匹配以alice开头的记录，它等价于`awk '!/^alice/' passwd`

匹配某列操作： 匹配操作符（~， !~）

1、`awk -F: '$1 ~ /^alice/' passwd` :以分号进行切分之后，匹配第1列是以alice开头的记录

2、`awk -F: '$NF !~ /bash$/' passwd` :以分号进行切分之后，匹配最后一列不是以bash进行结尾的记录

二、关系运算符

< 、<= 、==、!=、>=、>

== , != 可以比较字符串和数字

1、`awk -F: '$3 == 0' passwd` : 匹配第3列等于的0的记录

2、`awk -F: '$1 == "alice"' passwd` : 匹配第1列等于的alice的记录

三、条件表达式

1、`awk -F: '$3>2{print $1,$3}' passwd` : 第3列大于2

2、awk -F: '{if($3>2) print $1,$3}' passwd ：第3列大于2

3、`awk -F: '{if($3>2) { print $1,$3}}' passwd` : 第3列大于2

4、awk -F: '{if($3>2){print $1,$3} else {print $1,$2}}' passwd : 如果第3列的值大于2，打印1，3列，否则打印1，2两列

四、算术运算

+ - * / %(模) ^(幂)

1、`awk -F: '$3 * 10 > 500 {print $1,$3}' passwd` : 第3列大于2

五、逻辑操作符和复合模式

&&

||

!

1、`awk '$1~/alice/ && $3 > 10' passwd`

六、范围模式

`awk '/Tom/,/jet/' passwd`

示例

1、[root@node01 day4]# `awk 'BEGIN{FS=":"}{print $1}END{}' passwd`
root
#bin
#daemon
#adm
#lp
sync
shutdown
halt
mail
operator
games
ftp
nobody
123

FS：修改字段切分符，相当于`-F`

2、[root@node01 day4]# `awk 'BEGIN{FS=":"; OFS="-----"}{print $1,$2}END{}' passwd`
root-----x
#bin-----x
#daemon-----x

#adm-----x
#lp-----x
sync-----x
shutdown-----x
halt-----x
OFS: 字段连接规则

3、[root@node01 day4]# `awk '/^root/' passwd`
rootx:0:0:root:/root:/bin/bash

匹配以root开头的行

4、`awk -F: '/^root/{print $1}' passwd`

打印以root开头的行的第1个字段

5、[root@node01 day4]# `df -P | grep '^/' | awk '$4>858196{print $1}'`
/dev/mapper/centos-root

过滤出第4列大于858196的行， 然后打印第1列

6、`awk 'print($7 > 4 ? "hight: " $7 : "lows: " $6)' xxx.txt`

三目运算， 如果$7大于4， 打印hight:$7 ,否则打印lows:$6

7、 `awk '$3=="alice" {$3="jet"; print $0}' xxx.txt`

赋值操作

8、`awk '/alice/{$8+=12; print $8}' xxx.txt`

+=操作符

9、`awk '{$7%=3;print $7}' xxx.txt`

模等于3，然后打印$7

注意： 上面的这些赋值操作，模式运算，三目操作都是放在{action}中操作的。

数组操作

1、`awk -F: '{username[x++]=$1}END{for (i in username) print i, username[i]}'passws`
: 按索引遍历数组，i就是索引

4 mail
5 operator
6 games
7 ftp
8 nobody
0 root
1 sync
2 shutdown
3 halt

2、`awk -F: '{count[$NF]++} END{for(i in count){print i, count[i]}}' passwd`

统计passwd 文件中最后一列数量