[Edit](http://maxiang.info/#/?provider=evernote&guid=87a63dc2-8888-4983-862d-1b4e8780f8aa&notebook=linux)

# Linux the three musketeers

linux grep sed awk

linux中三剑客分别是grep, sed, awk. 他们都支持正则表达式，在文本处理上非常高效。现在就此三个命令做简单介绍

* [Linux the three musketeers](https://app.yinxiang.com/note/87a63dc2-8888-4983-862d-1b4e8780f8aa?usn=4980#linux-the-three-musketeers)
  + [grep, egrep, fgrep](https://app.yinxiang.com/note/87a63dc2-8888-4983-862d-1b4e8780f8aa?usn=4980#grep-egrep-fgrep)
  + [sed stream editor for filtering and transforming text](https://app.yinxiang.com/note/87a63dc2-8888-4983-862d-1b4e8780f8aa?usn=4980#sed-stream-editor-for-filtering-and-transforming-text)
    - [sed 参数](https://app.yinxiang.com/note/87a63dc2-8888-4983-862d-1b4e8780f8aa?usn=4980#sed-参数)
    - [位址](https://app.yinxiang.com/note/87a63dc2-8888-4983-862d-1b4e8780f8aa?usn=4980#位址)
    - [指令](https://app.yinxiang.com/note/87a63dc2-8888-4983-862d-1b4e8780f8aa?usn=4980#指令)
    - [sed 的小技巧](https://app.yinxiang.com/note/87a63dc2-8888-4983-862d-1b4e8780f8aa?usn=4980#sed-的小技巧)
  + [gawk - pattern scanning and processing language](https://app.yinxiang.com/note/87a63dc2-8888-4983-862d-1b4e8780f8aa?usn=4980#gawk-pattern-scanning-and-processing-language)
    - [Pattern](https://app.yinxiang.com/note/87a63dc2-8888-4983-862d-1b4e8780f8aa?usn=4980#pattern)
    - [Actions](https://app.yinxiang.com/note/87a63dc2-8888-4983-862d-1b4e8780f8aa?usn=4980#actions)
    - [format 格式控制符](https://app.yinxiang.com/note/87a63dc2-8888-4983-862d-1b4e8780f8aa?usn=4980#format-格式控制符)
    - [内建函数](https://app.yinxiang.com/note/87a63dc2-8888-4983-862d-1b4e8780f8aa?usn=4980#内建函数)
    - [awk 字段](https://app.yinxiang.com/note/87a63dc2-8888-4983-862d-1b4e8780f8aa?usn=4980#awk-字段)
    - [awk 数组](https://app.yinxiang.com/note/87a63dc2-8888-4983-862d-1b4e8780f8aa?usn=4980#awk-数组)
    - [awk程序使用shell命令](https://app.yinxiang.com/note/87a63dc2-8888-4983-862d-1b4e8780f8aa?usn=4980#awk程序使用shell命令)
    - [案例](https://app.yinxiang.com/note/87a63dc2-8888-4983-862d-1b4e8780f8aa?usn=4980#案例)

## grep, egrep, fgrep

egrep, fgrep 是grep的扩展，其主要功能便是打印符合样式的行

其控制打印的行数的参数有

* -A NUM, –after-context=NUM Print NUM lines of trailing context after matching lines.
* -B NUM, –before-context=NUM Print NUM lines of leading context before matching lines.
* -C NUM, –context=NUM Print NUM lines of output context.

也就是说A 打印匹配到样式及其后几行，B 打印匹配到样式及其前几行，C 便是匹配到样式的上下文下面仅以A举例，BC，同理

$ grep -A2 "sandow" /etc/passwd

sandow:x:500:500::/home/sandow:/bin/bash

oldboy:x:501:501::/home/oldboy:/bin/bash

oldgirl:x:502:502::/home/oldgirl:/bin/bash

控制匹配结果的参数有：

* -i ,–ignore-case. Ignore case distinctions 顾名思义，便是忽略大小写
* -v, –inver-match. Invert the sense of matching, to select non-matching lines 反向匹配
* -E, –extended-regexp. 扩展正则表达式，相当于egrep
* -F, –fixed-strings, –fixed-regexp, 把范本样式视为固定字符串的列表，相当于fgrep
* -o, –only-matching 只显示匹配到的那部分

控制输出结果的参数：

* -c, –count， 仅统计匹配到的结果，匹配到两个就会返回2
* -n, –line-number , 显示匹配到行的行数
* –color=auto 把匹配到的显示高亮显示

$ ifconfig eth0

eth0 Link encap:Ethernet HWaddr 00:0C:29:1E:00:6F

inet addr:10.0.0.17 Bcast:10.0.0.255 Mask:255.255.255.0

inet6 addr: fe80::20c:29ff:fe1e:6f/64 Scope:Link

UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1

RX packets:2410 errors:0 dropped:0 overruns:0 frame:0

TX packets:649 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:1000

RX bytes:229147 (223.7 KiB) TX bytes:135613 (132.4 KiB)

$ ifconfig eth0|grep Bcast|egrep -o "[0-9.]\*"

10.0.0.17

10.0.0.255

255.255.255.0

grep 也可以指定目录，它会搜索目录下所有文件里的内容并输出匹配符合样式的行

## sed stream editor for filtering and transforming text

sed 批量的编辑文本，sed 擅长处理行。sed 可以删除(delete), 改变（change）, 添加(insert), 合并，交换文件中的数据行，或读入其它文件的数据到文件中，也可以替换(substuite) 它他其中的字符串。是一个非交互式的效率非常高的命令

### sed 参数

* -n, –quiet, –silent 只打印匹配到的行
* -e script, 执行后面的命令
* -f script-file, 执行后面文件中的命令

语法sed -e 'command1' -e 'command2' filename 注意这个单引号并不是一成不变的，当在shell里需要引用变量时便可以用双引号

例如，删除sed.txt文件中第1到10行并且把magdre改为sandow 便可以这样

$ sed -e '1,10d' -e 's/magdre/sandow/g' sed.txt

### 位址

sed 的编辑指令分为位址(address)和指令 。 位址负责定位需要编辑的内容，指令负责编辑。其语法格式为 [address1 [,address2]][function[argument]]

当没有位址时便会对所有文本处理，一个位址时位会处理该位置定位到的行，两个位址时便会处理从位址1到位址2的行数。当然如果使用正则表达式便是匹配字符的行咯。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **位址中的指令** | **描述** |  | **位址中的指令** | **描述** |
| $ | 最后一行 |  | first~step | 从first开始步长为setp的行 |
| /regexp/ | 用正则来匹配行 |  | 0,addr2 | 匹配到前addr2行 |
| addr1,+n | 从addr1开始后n行 |  | addr1,~n | 这个不重要 |

看起来很复杂吧，先来几个例子

$ echo -e "hello\nmy name is sandow\nmy bolg is magdre.github.io" >test.txt

$ sed -n '$p' test.txt

my bolg is magdre.github.io

$ sed -n '/sandow/p' test.txt

my name is sandow

### 指令

上面看到很多d , p, g, s 等这些都是什么意思呢，他们都需要函数，那么接下来就来说说他们到底代表的什么意思

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **指令** | **描述** |  | **指令** | **描述** |
| = | 打印行数 |  | r | 从文件中记取数据 |
| a\ | 在当前行下追加数据 |  | i\ | 在当前行上追加数据 |
| c\ | 把选定行改为新文本 |  | d | 删除所选择的行 |
| s | 替换指定字符 |  | q | 退出sed编辑 |
| n | 读取下一个输入行，用下一个命令处理新的行 |  | N | 这个百度去吧 |
| h | 复制pattern space到内存缓存区 |  | H | 追加到缓存区 |
| g | 获取缓存区内容替换pattern space |  | G | 追加到 pattern space后 |
| p | 打印当前pattern space |  | P | 打印模版块的第一行 |
| : label | 建立参照位址 |  | b label | 将指令跳到参照位址 |
| w | 写入文件 |  |  |  |

当需要对一个位址执行多个指令时需要用{}来括起来

addr{

commend1

commend2

}

来来来，例子给你形象说明

$ cat test.txt

hello

my name is sandow

my bolg is magdre.github.io

$ sed '2a I am a gentleman' test.txt

hello

my name is sandow

I am a gentleman

my bolg is magdre.github.io

$ sed -e '2c\I am a gentleman' -e '1i\stay hungry stay foolish' test.txt

stay hungry stay foolish

hello

I am a gentleman

my bolg is magdre.github.io

这里有一个”!” 其功能很特别需要在这里说下，如下例

[root@centOs ~]# sed '/sandow/!d' test.txt

**my** name **is** sandow

这时会删除除匹配到的那行外的所有行

**替换字符举例**

s/regexp/replacement/[flag] 用正则表达式来匹配，匹配到后用replacement来做替换 在replacement里可以用\1到\9来输出前面regexp里的子表达式匹配到的内容。 另在replacement 里可以用 ‘&’ 来代表前面所匹配到的内容。我们用flag来控制如何替换

* g 代表替换所有匹配到的内容
* num 当为数字为是从第几个匹配到的内容开始替换
* p 标准输出
* w filename 替换后输入名为filename的文件中
* 空 仅替换第一个匹配到的内容

$ ifconfig eth0 | sed -nr '2s/.\*:(.\*) .\*:(.\*) .\*:(.\*)/\1 \2 \3/gp'

10.0.0.17 10.0.0.255 255.255.255.0

$ ifconfig eth0 | sed -nr '/Bcast/s/.\*:(.\*) .\*:(.\*) .\*:(.\*)/\1 \2 \3/gp'

10.0.0.17 10.0.0.255 255.255.255.0

**变形**

说实话这个不怎么用，所以只附上其英文解释

y/source/dest/ Transliterate the characters in the pattern space which appear in source to the corresponding character in dest.

### sed 的小技巧

sed -n '$=' test.txt 计算行数

ifconfig eth0 |sed 2q 显示前两行内容

sed '$!d' 或 ` sed -n ‘$p’ 显示最后一行内容

## gawk - pattern scanning and processing language

使用AWK写程序比起使用其它语言便利节省时间，AWK具有一些内建功能，使得AWK擅于处理具有数据行（record)，字段(field)型态的资料，而且AWK内建有pipe的功能，可将处理中的数据传送给外部shell命令加以处理，再将shell命令处理后的数据传回awk程序。

shell与awk数据交互两种方式

1. awk output指令 | “shell 接受的命令”  
   $ awk ‘{print $1, $2}’ arr.dat|”sort -k 1”
2. “shell 接受的命令”|awk input 指令  
   “ls” |getline

为了方便测试，将使用特下面数据集

员工ID 姓名 薪资率 实际工时

A125 Jenny 100 210

A341 Dan 110 215

P158 Max 130 209

P148 John 125 220

A123 Linda 95 210

员工ID中第一个为部门识别码A,P分别表示“组装”及“包装”部门

数据行：AWK从数据文件上读取数据的基本单位，表示第一行

字段： 在数据和f上被分隔符（默认为空格）分隔开的字符串

awk语法

awk 'awk程序' 数据文件名

$ awk -f awk 程序文件名 数据文件文件名

awk程序中主要的语法 **Pattern { Actions}** 所以AWK程序型态为

Pattern1 { Actions1 }

Pattern2 { Actions2 }

...

awk 会先判断(Evaluate) 该 Pattern 的值, 若 Pattern 判断后的值为 true (或不为 0 的数字,或不是空的字符串), 则 awk 将执行该Pattern 所对应的 Actions.反之, 若 Pattern 之值不为 true, 则 awk将不执行该 Pattern 所对应的 Actions.若Pattern部分被省略，只剩下{Actions}那么这种情况下无条件执行这个Actions

### Pattern

awk 可接受许多不同形态的Pattern，一般常使用“关系表达式(Relational expression)” 来当成Paatern, 例如 x > 34 或者 x == y 便是一个Pattern

关系运算符(Relational Operators) 有 <, >, <=, >=, != 。 以及 ~(match), !~(not match)

A ~ B A为字符串，B为正则表达式，判断A中是否包含能匹配B表达式的子字符串。

例如 "banana" ~ /an/

这里匹配可以这样玩$0 ~ /[0-9]+\.c/ {print "this is a c program"}因为$0可以缺省所以可以下拉写成这样/[0-9]+\.c/ {print "this is a c program"}

多个pattern一起使用中间可以用逻辑关系式(&&, ||, !)来结合

BEGIN，当awk程序一开始执行，尚未读取任何数据之前

**BEGIN** {

**FS** = "[ \t:]" # 于程序一开始时, 改变 awk 切割字段的方式

RS = "" # 于程序一开始时, 改变 awk 分隔数据行的方式

count = 100 # 设定变量 count 的起始值

print " This is a title line " # 印出一行 title

}

end 在awk处理完所有数据，既奖离开程序时执行

整个是一个 Pattern. 因为banana中含有或以匹配的/an/的子字符串，故此关系式成立

### Actions

Actions 是由许多 awk 指令构成. 而 awk 的指令与 C 语言中的指令十分类似.

例如 :

awk 的 I/O 指令 : print, printf( ), getline…

awk 的 流程控制指令 : if(…){..} else{..}, while(…){…}…

awk 指令有

* 表达式 ( function calls, assignments..)
* print 表达式列表
* printf( 格式化字符串, 表达式列表)
* if( 表达式 ) 语句 [else 语句]   
  if( $1 > 25 )   
  print “The 1st field is larger than 25”   
  else print “The 1st field is not larger than 25”
* while( 表达式 ) 语句   
  while( match(buffer,/[0-9]+.c/ ) ){   
  print “Find :” substr( buffer,RSTART, RLENGTH)   
  buff = substr( buffer, RSTART + RLENGTH)   
  }
* do 语句 while( 表达式)   
  do{   
  print “Enter y or n ! ”   
  getline data   
  } while( data !~ /^[YyNn]$/)
* for( 表达式; 表达式; 表达式) 语句   
  awk ’   
  BEGIN{   
  X[1]= 50; X[2]= 60; X[“last”]= 70   
  for( any in X )   
  printf(“X[%s] = %d\n”, any, X[any] )   
  }’
* for( variable in array) 语句   
  for(ｉ=1; i< =10; i++) sum = sum + i
* delete
* break
* continue   
  for( index in X\_array)   
  {   
  if( index !~ /[0-9]+/ ) continue   
  print “There is a digital index”, index   
  }
* next
* exit [表达式]
* 语句

### format 格式控制符

printf(“ID# : %s Ave Score : %d\n”, id, ave)

format 部分是由 一般的字串(String Constant) 及格式控制字符(Formatcontrol letter, 其前会加上一个%字符)所构成. 以上式为例”ID# : ” 及 ” Ave Score : ” 为一般字串. %s 及 %d 为格式控制字符.

数值运算 +-*/%^ +=,-=,*=,/=,%=,^=, ++, –,

条件判断

？ value1 : value2

若判断条件成立则返回value1否则返回value2

逻辑判断 &&(and), ||(or), !(not)

关系运算 <, >, <=, >=, =, ==, ~=, ~, !~

优先级为 指数(^) << +, -, ! 正负非 << \*,/ , % << +, - <<关系运算 << 匹配 ~,!~ << 逻辑判断

### 内建函数

* index(string,match), 在string中找match，找到返回第一次出现的位置，未到到返回0
* match(string, pattern),使用正则表达式pattern来匹配string并返回相应位置，其中有两个内置函数 RSTART(找到相应字符的位置),RLENGTH（找到字符串的长度)，   
  awk ’ BEGIN {   
  match( “banana”, /(an)+/ )   
  print RSTART, RLENGTH   
  } ’   
  返回 2 4
* length(string) ：返回字符串的长度
* split(string, 数组名称，分隔字符), 把string按分隔字符分隔成一个个field并且使用数组名称来记录这些field   
  ArgLst = “5P12p89”   
  split( ArgLst, Arr, /[Pp]/)   
  执行后 : Arr[1]=5, Arr[2]=12, Arr[3]=89
* sprintf(string,format1, format2,…), 与prinf相同，不过sprintf()会把要求打印出来结果当成一个字串返回，   
  x = 28   
  x = sprintf(“%.2f”,x   
  执行后 x = “28.00”
* sub(patern, newstr, string) 将原字串第一个合乎指定的正则表达式的子字符串改以新字串取代   
  A = “a6b12anan212.45an6a”   
  sub( /(an)+[0-9]\*/, “[&]”, A)   
  print A   
  结果输出

ab12[anan212].45an6a

### awk 字段

$0 字符串，内容为目前AWK读入的数据行

$1 $0上第一个字段的数据

$2 $0上第二个字段的数据

NF 一个整数，值为$0 上所存在的字段数目

NR 整数，值为AWK已读入的数据行数目

FILENAME 正在处理的数据文件名。

执行 awk 时, 它会反复进行下列四步骤.

1. 自动从指定的数据文件中读取一个数据行.
2. 自动更新(Update)相关的内建变量之值. 如 : NF, NR, $0…
3. 依次执行程序中所有 的 Pattern { Actions } 指令.
4. 当执行完程序中所有 Pattern { Actions } 时, 若数据文件中   
   还有未读取的数据, 则反复执行步骤 1 到步骤 4.

$ awk '{ print $2, $3 \* $4 }' emp.dat

执行结果如下 :

屏幕出现 :

Jenny 21000

Dan 23650

Max 27170

John 27500

Linda 19950

$ cat pay2.awk

{ **printf**("%6s Work hours: %3d Pay: %5d\n", $2,$3, $3\* $4) }

$ awk -f pay2.awk emp.dat

Jenny Work hours: 100 Pay: 21000

Dan Work hours: 110 Pay: 23650

Max Work hours: 130 Pay: 27170

John Work hours: 125 Pay: 27500

Linda Work hours: 95 Pay: 19950

%6s %为格式化输出后面的参数，6s为长度为6的字符串

%3d 为长充为3的整数

$ cat adjust1.awk

$1 ~ /^A.\*/ { $3 \*= 1.05 }

$3<100 { $3 = 100 }

{ **printf**("%s %8s %d\n", $1, $2, $3)}

$ awk -f adjust1.awk emp.dat

A125 Jenny 105

A341 Dan 115

P158 Max 130

P148 John 125

A123 Linda 100

# 也可以写成一行

$ awk '$1 ~ /^A.\*/ {$3 \*= 1.05} $3<100 {$3 = 100} {printf("%s %8s %d\n",$1,$2, $3)}' emp.dat

A125 Jenny 105

A341 Dan 115

P158 Max 130

P148 John 125

A123 Linda 100

### awk 数组

awk中每一个数组都有下标(index)，其下标可以是数列，也可以是特定字符。使用数组前不须宣告数组名及其大小

使用的数据文件 reg.dat,

$ cat reg.dat

Mary O.S. Arch. Discrete

Steve D.S. Algorithm Arch.

Wang Discrete Graphics O.S.

Lisa Graphics A.I.

Lily Discrete Algorithm

#统计报的课程名称

$ cat course.awk

{ **for**( i=2; i <= NF; i++) Number[$i]++ }

END{**for**(course **in** Number) **printf**("%10s %d\n", course,

Number[course] )}

$ awk -f course.awk reg.dat

Discrete 3

O.S. 2

A.I. 1

D.S. 1

Graphics 2

Algorithm 2

Arch. 2

END 表示当AWK处理完所有数据，即将离开程序时才会为TURE，同理

BEGIN 和 END 同为 awk 中的一种 Pattern. 以 BEGIN 为 Pattern 的

Actions ,只有在 awk 开始执行程序,尚未开启任何输入文件前, 被执行

一次.(注意: 只被执行一次)

### awk程序使用shell命令

awk允许使用shell指令，并且提供解决AWK与系统间数据传递的问题。

统计线上人数

$ cat count.awk

BEGIN {

**while** ("who"|getline) n++

**print** n

}

$ awk -f count.awk

2

#一条命令

$ awk 'BEGIN {while ("who"|getline) n++; print n}'

2

getline为awk提供的输入指令

语法为 getline var < file (读入数据)

### 案例

数据

$ cat arr.dat

1034 7:26

1025 7:27

1101 7:32

1006 7:45

1012 7:46

1028 7:49

1051 7:51

1029 7:57

1042 7:59

1008 8:01

1052 8:05

1005 8:12

一、现在使用awk把他重新定向到 today\_rpt1中并加上表头

在到班资料文件 arr.dat 之前增加一行抬头”ID Number Arrvial Time”, 并产生报表输出到文件 today\_rpt1 中

$ cat reformat1.awk

BEGIN { **print** " ID Number Arrival Time" > "today\_rpt1"

**print** "===========================" >> "today\_rpt1"

} {

**printf**(" %s %s\n", $1,$2 ) >> "today\_rpt1" }

$ awk -f reformat1.awk arr.dat

$ cat today\_rpt1

ID Number Arrival Time

===========================

1034 7:26

1025 7:27

1101 7:32

1006 7:45

1012 7:46

1028 7:49

1051 7:51

1029 7:57

1042 7:59

1008 8:01

1052 8:05

1005 8:12

上面文件名称必须使用双引号括住，表示today\_rpt1为一字符串常量，不括会被awk当做一个变量，awk中变量不需要声明，转为初始值为**“”**者**0.**

二、 将 today\_rpt1 上的数据按员工代号排序, 并加注执行当日日期;

产生文件 today\_rpt2

$ cat reformat2.awk

BEGIN {

"date" | getline

**print** " Today is " , $2, $3 >"today\_rpt2"

**print** "=========================" > "today\_rpt2"

**print** " ID Number Arrival Time" >"today\_rpt2"

close( "today\_rpt2" )

}

{**printf**( "%s %s\n", $1 ,$2 ) | "sort -k 1 >>today\_rpt2"}

$ awk -f reformat2.awk arr.dat

三、将 awk 程序包含在一个 shell script 文件中

awk提供另一个调用shell命令的方法，即使用awk函数system(“shell command”)

举例：

$ awk '

BEGIN{

system("date > date.dat")

getline < "date.dat"

print "tody is ", \$2, \$3

}

'

四、 于 today\_rpt2 每日报表上, 迟到者之前加上”\*”, 并加注当日平均到班时间;产生文件 today\_rpt3

这里需要分别取出分钟数与小时数，下面介绍两种方法

length(string) 计算字符串的长度

substr(string,start,length) 返回从起始位置起指定长度的子字符串，不指定长度则返回到字符串末尾的子字符串

1：使用**substr**

小时数 = substr($2, 1, length($2) - 3)

分钟数 = substr($2 length($2) - 2)

2： 指定分隔符，awk默认分隔符为空格

FS = “[ \t:]+”

现在开始做第4题

$ cat reformat3

#!/bin/sh

awk '

BEGIN {

FS = "[ \t:]+"

"date"|getline

print "Today is ", $2, $3 > "today\_rpt3"

print "=========================">"today\_rpt3"

print " ID Number Arrival Time" > "today\_rpt3"

close( "today\_rpt3" )

}

{ #已更改字段切割方式, $2 表到达小时数, $3 表分钟数

arrival = HM\_to\_M($2, $3)

printf(" %s %s:%s %s\n", $1, $2, $3, arrival > 480 ? "\*": "" ) | "sort -k 1 >> today\_rpt3"

total += arrival

}

END {

close("today\_rpt3")

close("sort -k 1 >> today\_rpt3")

printf(" Average arrival time : %d:%d\n",total/NR/60,(total/NR)%60 ) >> "today\_rpt3"

}

function HM\_to\_M(hour, min) {

return hour\*60 +min

}

' $\*

上面的arrival > 480 ? "\*": ""意思是当arrival 大于480时就返回 \* 否则就返回”“

指令 close( “sort -k 1 >> today\_rpt3” ), 其意思为 close 程序中置于 “sort -k 1 >> today\_rpt3 ” 之前的 Pipe , 并立刻调用 Shell来执行”sort -k 1 >> today\_rpt3”. (若未执行这指令, awk 必须于结束该程序时才会进行上述动作;则这 12 笔 sort 后的数据将被 append 到文件 today\_rpt3 中

从文件中读取当月迟到次数, 并根据当日出勤状况更新迟到累计数.