Shell

文本处理工具

grep

> *grep是行过滤工具，用于根据 关键字进行过滤*

```
# grep [option] 'keyworkd' filename
```

- 常见选项
  - i：忽略大小写
  - v：查找不包含指定内容的行 ，反向选择
  - w：按单词搜索
  - o：打印 匹配关键字
  - c：统计匹配到的次数
  - n：显示行号
  - r：逐层遍历目录查找
  - A：显示匹配行及后面多少行
  - B：显示匹配行及前面多少行
  - C：显示匹配行及前后多少行
- 关键字
  - ^key：以该关键字开头的行
  - key$：以该关键字结尾的行

> ***一般情况下，将grep自动添加上--color=auto显示关键字高亮会使得效率更高，更不容易出错***

```
# vim /etc/bashrc(或者vim .bashrc)
# 进入文件后，直接在文件最后一行插入
# alias grep = 'grep --color=auto'
# :wq
```

cut

> *cut是列截取工具，用于列的截取*

```
# cut [option] filename
```

- 常见 选项

  - c：以字符为单位进行分割，截取
  - d：自定义分隔符，默认为制表符\t
  - f：与-d选项一起使用，指定截取哪一块区域

- 案例：

```
# cut -d: -f1,6,7 test.txt
以冒号为分割符，截取第1,6,7列的内容
```

sort

> *sort用于排序，它将文件的每一行作为一个单位，从首字母向后，依次按照ASCII码进行比较，最后将他们按升序输出*

```
# sort [option] filename
```

- 常见选项
  - u：去除重复行
  - r：降序排列，默认是升序
  - o：将排序的结果输出到文件中，类似输出重定向
  - n：以数字排序，默认是按字符排列
  - t：分隔符
  - k：第N行
  - b：忽略前导空格
  - R：随即排序

uniq

> *uniq用于去除连续的重复行*

```
# uniq [option] filename
```

- 常用选项
  - i：忽略大小写
  - c：统计重复行次数

- d：只显示重复行

tee

> *tee从标准输入读取并且写入到标准输出和文件，即：双向覆盖重定向(屏幕输出|文本输入)*

```
# echo hello | tee filename
```

- 常用选项
  - a：双向追加重定向

diff

> *diff工具用于逐行比较文件的不同*
> *注意：**diff描述两个文件不同的方式是告诉我们怎样改变第一个文件之后与第二个文件匹配***

```
# diff [option] filename1 filename2
```

- 常用选项
  - b：不检查空行
  - B：不检查空白行
  - i：不检查大小写
  - w：忽略所有空格
  - c：上下文格式显示
  - u：合并格式显示
  - --normal：正常格式显示(默认)

paste

> *paste用于合并文件行*

```
# paste [option] filename
```

- 常用选项
  - d：自定义间隔符，默认是tab
  - s：串行处理，非并行

tr

> *tr用于字符 转换，替换和删除。主要用于 删除文件中控制字符或进行字符转换*

- 常用选项

> - d：删除string1中的所有输入字符
> - s：删除所有重复出现字符序列，只保留第一个。即将重复出现字符串压缩为一个字符串

- 案例：

```
用法1：命令的执行结果交给tr处理，其中string1用于查询，string2用于转换处理
# comands  | tr 'string1' 'string2'

用法2：tr处理的内容来自文件，记住要使用"<"标准输入
# tr 'string1' 'string2' < filename

用法3：匹配string1进行相应操作，如删除操作
# tr option  'string1' < filename
```

- 常匹配字符串

字符串	含义	备注
a-z或[:lower:]	匹配所有小写字母	
A-Z或[:upper:]	匹配所有大写字母	
0-9或[:digit:]	匹配所有数字	
[:alnum:]	匹配所有字母和数字	
[:alpha:]	匹配所有字母	
[:blank:]	匹配所有水平空白	
[:punct:]	匹配所有标点符号	
[:space:]	匹配所有水平或垂直的空白	
[:cntrl:]	匹配所有控制字符	\f、\n等

bash操作

常见快捷键

```
Ctrl + c #终止前台运行的程序
Ctrl + d #退出，等价exit
Ctrl + l #清屏
Ctrl + a | home #将光标移动到命令行的最前端
```

```
Ctrl + e | end #将光标移动到命令行的最后端
Ctrl + u #删除光标前所有字符
Ctrl + k #删除光标后所有字符
Ctrl + r #搜索历史命令
```

常见通配符

```
* : 匹配0或多个任意字符
? : 匹配任意单个字符
[list] : 匹配[list]中的任意单个字符
[!list] : 匹配除[list]中的任意单个字符
{string1, string2, ...} : 匹配string1,string2或多个字符串
```

bash中的引号

- 双引号“”：会把引号中的内容当成整体来看待，允许通过$的符号引用 其他变量值
- 单引号‘’：会把引号中的内容当成整体来看待，禁止引用其他变量值，shell中特殊符号都被视为普通字符
- 反撇号``：反撇号和$()一样，引号或括号里的命令会优先执行，如果存在嵌套，反撇号不能用

Shell编程

> *shell介于内核与用户之间，负责命令的解释*

shell的种类

- /bin/sh
  - 是bash的一种快捷方式
- /bin/bash
  - bash是 大多数linux默认的shell,包含的功能几乎可以涵盖shell所有功能
- /sbin/nologin
  - 表示非交互，不能登陆操作系统
- /bin/dash
  - 小巧，高效，功能相对少一点
- /bin/csh
  - 具有C风格的一种shell,具有许多特性，但也有一些缺陷
- /bin/tcsh
  - 是csh的增强版，完全兼容csh

什么是shell脚本

> *简单来说就是将**需要执行的命令**保存到文本中，按照**顺序执行**。*
> *若干命令 + 脚本的基本格式 + 脚本的特定语法 + 思想 = shell脚本*

shell脚本的基本语法

- 脚本的第一行，**必须指定出哪一个解释器**

```
#!/bin/bash
```

- 注意：如果直接将解释器路径写在脚本里，可能在某些系统就会存在找不到解释器的**兼容性问题**， 所以可以使用：

```
#!/bin/env 解释器
```

- shell脚本的注释使用#作为标识符

变量

```
A=hello   定义变量A

echo $A   调用变量A
echo ${A}   调用变量A

unset A   取消变量
```

- 注意：
  - **调用变量必须使用$符号**
  - 取消变量使用**unset**关键字
- 注意：
  - **Shell中变量严格区分大小写**
  - **禁止使用特殊符号作为变量名**
  - 对有空格符的，使用""号包括起来
  - **变量名不能以数字开头**
  - **赋值号的两边不能空格**

切片操作

```
A=123456
echo ${A:2:3}
```

- 注意
  - var:start:len
  - Shell中是从**0开始计算的**

交互式定义变量

> *让用户自己给变量赋值，比较灵活*

```
read [option] varname
```

- 常用选项
  - p：定义提示用户的信息
  - n：定义字符数(限制变量值的长度)
  - s：不显示
  - t：定义超时时间，默认单位为秒
- 案例：

```
用法1：用户自己定义变量值
read name -p "请输入用户名"

用法2：变量值来自于文件
read -p "Input your IP and Netmask: " ip mask < filename
```

- 注意：
  - p选项后面**必须**跟着提示信息
  - 文件默认以**空格**分割

定义有类型的变量

> *给变量做一些限制，固定变量的类型，比如：整形，只读*

```
declare [option] var=value
```

- 常用选项
  - i：将变量看成整数

- r：定义只读变量
- a：定义普通数组，查看普通数组
- A：定义关联数组，查看关联数组
- x：将变量通过环境导出

变量的分类

本地变量

> *当前用户自定义的变量。当前进程中有效，其他进程及当前进程的子进程无效*

环境变量

> *当前进程有效，并且能够被子进程调用*

- env：查看当前用户的环境变量
- set：查询当前用户的所有变量(临时变量与环境变量)
- export var=value 或者 var=value; export var
- 环境变量全局配置文件 **/etc/profile**

全局变量

> *全局所有的用户和程序都能调用，且继承、新建的用户也默认能调用*

文件名	说明	备注
$HOME/.bashcr	当前用户的bash信息，用户登陆时读取	定义别名，umask、函数等
$HOME/.bash_profile	当前用户的环境变量，用户登陆时读取	
$HOME/.bash_logout	当前用户退出当前shell时最后读取	定义用户退出时执行的程序等
/etc/bashrc	全局的bash信息，所有用户都生效	
/etc/profile	全局环境变量信息	

- 注意：
  - 以上文件修改后，都需要**source对应文件**让其生效或重新退出登陆
  - 用户登陆系统**读取相关文件顺序**
    1. /etc/profile
    2. $HOME/.bash_profile
    3. $HOME/.bashrc

> 4. /etc/bashrc
> 5. $HOME/.bash_logout

系统变量

- 系统变量(内置bash变量)：shell本身已经固定好了名字和作用

内置变量	含义
$?	上一条命令执行后返回的状态，为0表示正常，非0为异常
$0	当前执行的程序或脚本名
$#	脚本后的参数个数
$*	脚本后面所有参数，参数当成一个整体输出，每一个参数之间以空格隔开
$@	脚本后面所有参数，参数是独立的，也是全部输出
$1-$9	脚本后面的位置参数，$1表示第一个位置参数，依次类推
${10} - ${n}	扩展位置参数，两位数以上必须用{}括起来
$$	当前所在进程的进程号，比如echo $$
$!	后台运行的最后一个进程号(当前终端)
!$	调用最后一条命令历史中的参数

简单四则运算

> *默认情况下，shell就只能支持简单的整数运算*

四则运算符号

表达式	举例
$(())	echo $((1 + 1))
$[]	echo $[10 - 5]
expr	expr 10 / 5
let	n=1; let n+= 1 等价于 let n=n+1

条件判断语句

语法格式

- 格式一

> - test条件表达式

- 格式二
  - [条件表达式]
- 格式三
  - [[条件表达式]] 支持正则
- **注意**:
  - **[]的两边必须留有空格**

条件判断的相关参数

判断文件类型

判断参数	含义
e	判断文件是否存在(任意类型文件)
f	判断文件是否存在且是一个普通文件
d	判断文件是否存在且是一个目录
L	判断文件是否存在且是一个软连接文件
b	判断文件是否存在且是一个块设备文件
S	判断文件是否存在且是一个套接字文件
c	判断文件是否存在且是一个字符设备文件
p	判断文件是否存在且是一个命名管道文件
s	判断文件是否存在且是一个非空文件

- 案例:

```
用法1: 判断filename文件是否存在
test -e filename

用法2: 判断filename是否为目录
[ -d ./filename ]
```

判断文件权限

判断参数	含义
r	当前用户对其是否可读

判断参数	含义
w	当前用户对其是否可写
x	当前用户对其是否可执行
u	是否有suid,高级权限冒险位
g	是否有sgid，高级权限强制位
k	是否有t位，高级权限粘滞位

判断文件新旧

判断参数	含义
file1 -nt file2	比较file1是否比file2新
file1 -ot file2	比较file1是否比file2旧
file1 -ef file2	比较是否为同一个文件，或者用于判断软硬链接，是否指向同一个inode

判断整数

判断参数	含义
eq	相等
ne	不等
gt	大于
lt	小于
ge	大于等于
le	小于等于

判断字符串

判断符号	含义	举例
a 和 &&	逻辑与	[1 -eq 1 -a 1 -ne 0] [1 -eq 1] && [1 -ne 0]
-o 和 ||	逻辑或	[1 -eq 1 -o 1 -ne 1] [1 -eq 1] || [1 -ne 1]

- 注意：
  - && 前面的表达式为真，才会执行后面的代码
  - || 前面的表达式为假，才会执行后面的代码
  - ; 只用于分割命令或表达式

类C风格判断数值

```
(( 1 == 1 )); echo $?
```

[ ] 和 [[ ]]的区别

流程控制语句

基本语法结构

if结构

- F：表示flase,为假
- T：表示true,为真

```
if [ condition ]; then
    comand
    command
fi

if test condition; then
    command
fi

if [[ condition ]]; then
    command
fi

[ condition ] && command
```

- 注意：
  - 在if判断之后，**需要加分号，然后then**
  - if的结束是**通过fi结束的**

if...else结构

```
if [ condition ]; then
    comand1
    else
        command2
fi

[ condition ] && command1 || command2
```

- 小练习：

> *让用户自己输入字符串，如果用户输入的是hello,请打印world,否则打印“请输入hello”*

if...elif...else结构

```
if [ condition ]; then
    command1
    elif [ codition ]; then
        command2
        else
            command3
fi
```

pgrep命令

> *pgrep命令以名称为依据从运行队列中查找进程，并显示查到的进程ID*

```
# pgrep proname
```

- 常见选项
  - o：仅显示找到的最小(起始)进程号

循环控制语句

FOR语法结构

列表循环

> *列表for循环：用于将一组命令执行**已知的次数***

```
for variable in {list}
    do
        command
        ...
    done
```

- 注意：
  - **列表的格式**：{1..2..1}
  - {start..end..step}
- tip：
  - 可以使用seq来构成循环

```
for i in $(seq 10)
    do
        command
    done
```

不带列表循环

> *不带列表的for循环执行时由用户指定参数和参数的个数*

```
for variable
    do
        command
    done
```

类C风格的for

```
for (( expr1; expr2; expr3 ))
    do
        command
    done

for (( i=1; i<=5; i++ ))
    do
        echo $i
    done
```

WHILE循环结构

```
while condition
    do
        command
    done

while [ 1 -eq 1 ] 或者 (( 1 > 2 ))
    do
        command
    done
```