Shell

笔记

冰糖雪狸

2020

目录

[一. Shell概述 2](#_Toc47357590)

[二. Shell脚本的执行方式 3](#_Toc47357591)

[三. Shell脚本在不同环境编写的兼容性问题 4](#_Toc47357592)

[四. Bash的基本功能 5](#_Toc47357593)

[4.1 历史命令与命令补全 5](#_Toc47357594)

[4.2 命令别名与常用快捷键 6](#_Toc47357595)

[4.3 输入输出重定向 7](#_Toc47357596)

[4.4 多命令顺序执行与管道符 8](#_Toc47357597)

[4.5 通配符与其他特殊符号 9](#_Toc47357598)

[五. Bash 中的变量 10](#_Toc47357599)

[5.1 用户自定义变量 10](#_Toc47357600)

[5.2 环境变量 11](#_Toc47357601)

[5.3 位置参数变量 12](#_Toc47357602)

[5.4 预定义变量 13](#_Toc47357603)

[六. Bash 运算符 14](#_Toc47357604)

[6.1 数值运算与运算符 14](#_Toc47357605)

[6.2 变量的测试与内容替换 15](#_Toc47357606)

[七. 环境变量配置文件 16](#_Toc47357607)

[7.1 环境变量配置文件简介 16](#_Toc47357608)

[7.2 环境变量配置文件作用 17](#_Toc47357609)

[7.3 其他配置文件和登录信息 18](#_Toc47357610)

[八. 正则表达式 19](#_Toc47357611)

[九. 字符串截取命令 21](#_Toc47357612)

[9.1 cut字段提取命令 21](#_Toc47357613)

[9.2 printf命令 22](#_Toc47357614)

[9.3 awk命令 23](#_Toc47357615)

[9.4 sed 命令 25](#_Toc47357616)

[十. 字符处理命令 28](#_Toc47357617)

[十一. 条件判断 30](#_Toc47357618)

[十二. 流程控制 33](#_Toc47357619)

[12.1 if语句 33](#_Toc47357620)

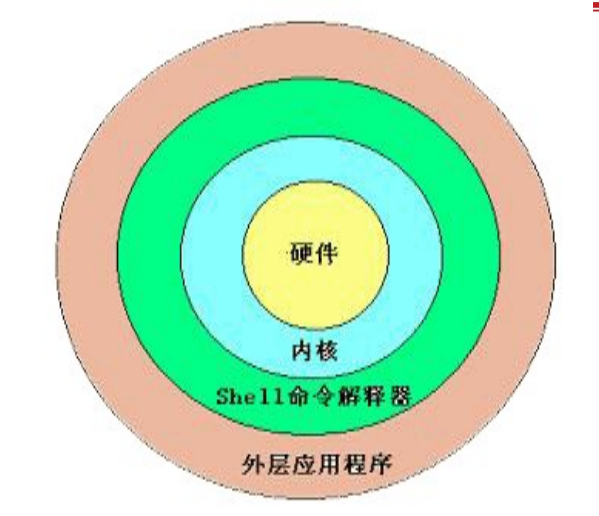
[12.2 case语句 37](#_Toc47357621)

[12.3 while循环与until循环 39](#_Toc47357622)

# 一. Shell概述

1. Shell是什么

* Shell是一个命令解释器，它为用户提供了一个向Linux内核发送请求以便运行程序是界面系统级程序，用户可以用Shell来启动、挂起、停止甚至是编写一些程序



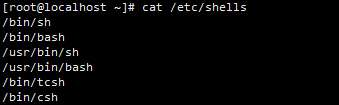
* Shell还是一个功能相当强大的编程语言，易编写，易调试，灵活性较强。Shell是解释执行的脚本语言，在Shell中可以直接调用Linux系统命令

2. Shell的分类

* **Bourne Shell：从1979起Unix就开始使用 Bourne Shell，Bourne Shell的主文件名为 sh**
* C Shell：C Shell主要在BSD版的Unix系统中使用，其语法和C语言相类似而得名
* Shell的两种主要语法类型有Bourne和C，**这两种语法彼此不兼容。**
  + Bourne家族主要包括sh、ksh、Bash、psh、zsh
  + C家族主 要包括：csh、tcsh
* **Bash： Bash与sh兼容，现在使用的Linux 就是使用Bash作为用户的基本Shell**

3. Linux支持的Shell

* 在目录/ect/shells

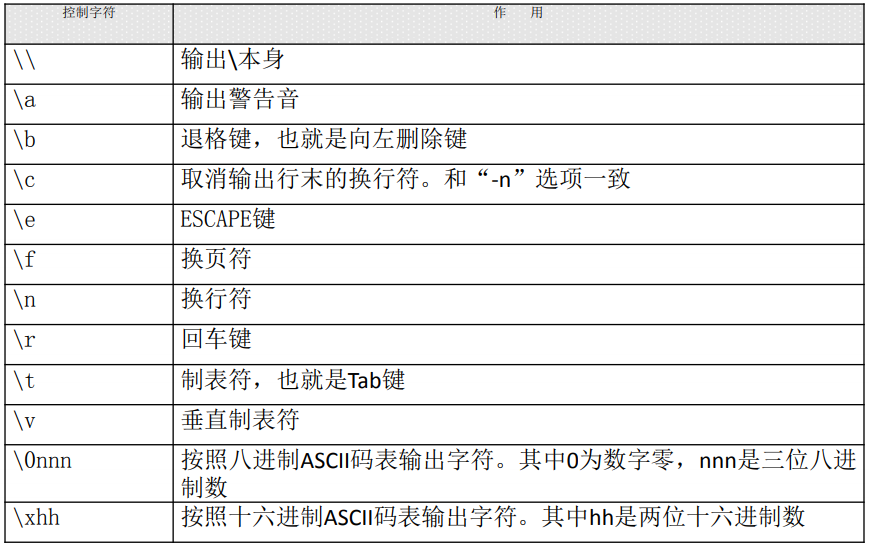


# 二. Shell脚本的执行方式

1. echo 输出命令

**echo [选型] [输出内容] 🡨 输出内容用单引号或双引号**

**-e 支持反斜杠控制的字符转换**



比如：

* 删除左侧字符： echo -e “ab\bc”
* 制表符和换行符： echo -e “a\tb\tc\nd\te\tf”
* 按照16进制ASCII码输出： echo -e "\x61\t\x62\t\x63\n\x64\t\x65\t\x66"
* 输出颜色： echo -e "\e[1;31m abcd \e[0m"
  + #30m=黑色，31m=红色，32m=绿色，33m=黄色
  + #34m=蓝色，35m=洋红，36m=青色，37m=白色

2. 第一个脚本

[root@localhost sh]# vi hello.sh

#!/bin/bash 🡨 第一行不是注释，而是一个shell 脚本不可缺的

# The first program

# 2020年7月31日12:38:36

echo -e “Hello world”

3. 脚本执行

* 赋予权限，直接运+行
  + chmod 755 hello.sh
  + ./hello.sh
* 通过Bash直接调用执行脚本
  + bash hello.sh 🡨 通过bash调用执行的可以不赋予执行权限，因为系统是通过bash调用的

# 三. Shell脚本在不同环境编写的兼容性问题

* 因为Windows中编写和Linux 编写的程序程序中存在些许不同
* **cat -A** 🡨 **查询所有的内容，包括隐藏字符**
* 在Linux中回车符识别为$符，而在Windows中不是^CTRL$
* 因为回车符符不同，所以Linux中执行Windows中编写的脚本程序，就会因为回车符的不同导致格式不匹配
* 把格式从Windows格式转换为Unix格式：
  + dos2unix 文件名
  + 如果没有安装，使用yum或rpm装上就好了

# 四. Bash的基本功能

## 4.1 历史命令与命令补全

1. 历史命令

**历史命令**

**history [选项] [历史命令保存文件]**

选项：

-c： 清空历史命令

-w： 把缓存中的历史命令写入历史命令保存文件 ~/.bash\_history

* 历史命令默认会保存1000条,可以在环境 变量配置文件/etc/profile中进行修改
* 历史命令的调用
  + 使用上、下箭头调用以前的历史命令
  + 使用 “**!n**” 执行第n条历史命令
  + 使用 “**!!**” 重复执行上一条命令
  + 使用 “**!字符串**” 重复执行最后一条以该**字符串开头**的命令

2. 命令与文件补全

* 在Bash中，命令与文件补全是非常方便与常用的功能，我们只要在输入命令或文件时，**按“Tab”键**就会自动进行补全

## 4.2 命令别名与常用快捷键

**1. 命令别名**

* + **设定命令别名 alias 别名=’原命令’**
  + **查询命令别名 alias**
  + **删除别名 unalias**

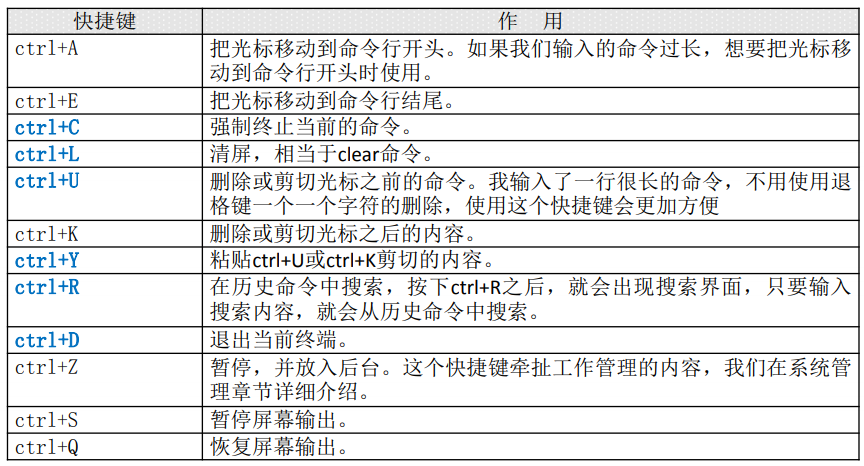
2. 命令执行时的顺序

1. 第一顺位执行 用绝对路径或相对路径执行的命令
2. 第二顺位执行 别名
3. 第三顺位执行 Bash的内部命令
4. 第四顺位执行 按照$PATH环境变量定义的目录查找到的第一个命令

3. 让别名永久生效

* 修改vi /root/.bashrc
* 用**alias 别名=’原命令’** 设置的别名在关机重启后就会失效

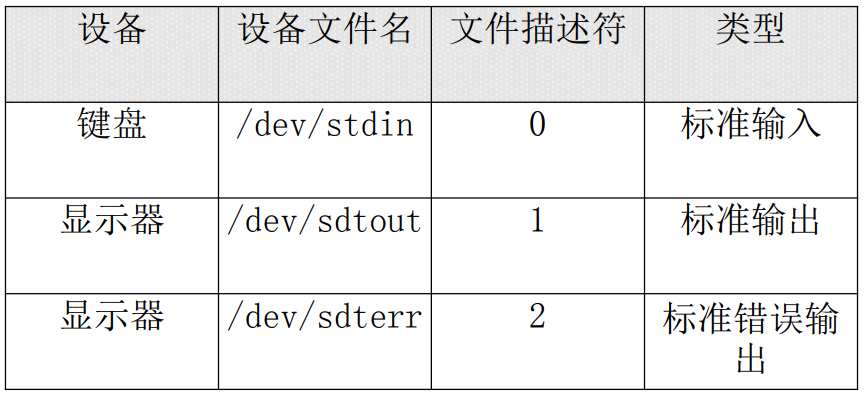
4. Bash常用快捷键



图中应该为小写，大写只是为了便于查看

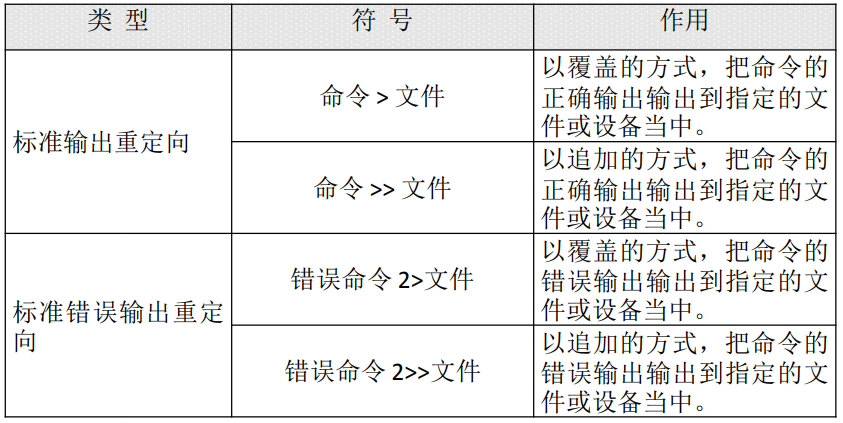
## 4.3 输入输出重定向

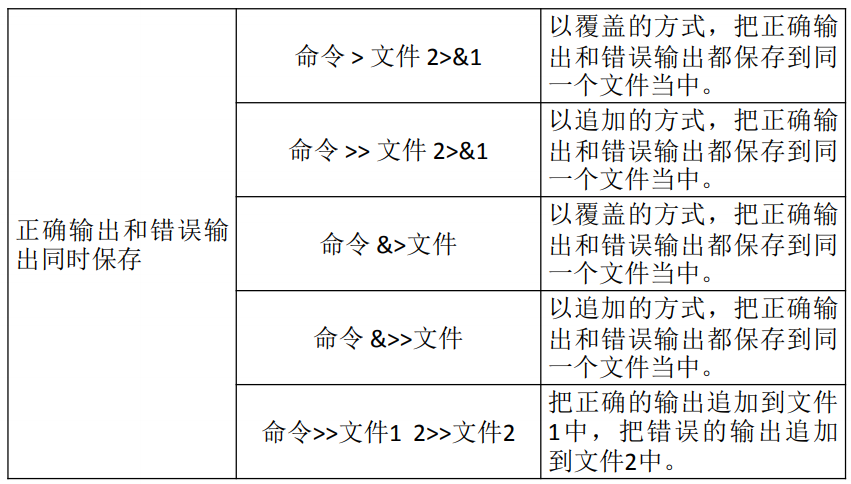
1. 标准输入输出



**2. 输出重定向**

* 输出重定向可以看作是本应当输出到屏幕上，但是输出到了文件中（重定向到文件中）





**3. 输入重定向**

* 命令<文件 把文件作为命令的输入

**输入重定向**

**wc [选项] [文件名]**

选项：

-c 统计字节数

-w 统计单词数

-l 统计行数

* 命令<<标识符

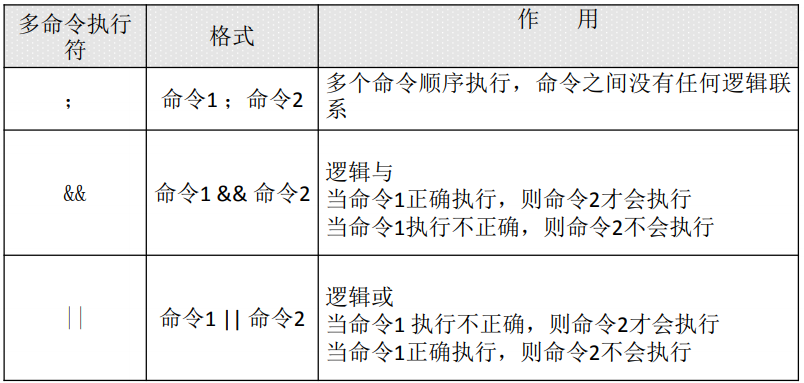
……

标识符 🡨 遇到相同的标识符停止输入

把标识符之间的内容作为命令的输入

## 4.4 多命令顺序执行与管道符

1. 多命令顺序执行



例子：

* [root@localhost ~]# **ls ; date ; cd /user ; pwd**
* [root@localhost ~]# **dd if=输入文件 of=输出文件 bs=字节数 count=个数**

选项：

if=输入文件 指定源文件或源设备

of=输出文件 指定目标文件或目标设备

bs=字节数 指定一次输入/输出多少字节，即把这些字节看做一个数据块

count=个数 指定输入/输出多少个数据块

例子： [root@localhost ~]# date ; dd if=/dev/zero of=/root/testfile bs=1k count=100000 ; date

* [root@localhost ~]# ls anaconda-ks.cfg && echo yes
* [root@localhost ~]# ls /root/test || echo no
* [root@localhost ~]# **命令 && echo yes || echo no**  🡸 此命令用于判断命令是否正确执行

2. 管道符

**管道符**

**命令一 | 命令二**

解释：用命令一的正确输出作为命令二的操作对象

例子：

[root@localhost ~]# ll -a /etc/ | more

[root@localhost ~]# netstat -an | grep "ESTABLISHED"

**grep文件内字符串搜索**

**grep [选项] "搜索内容" 文件名**

选项：

-i： 忽略大小写

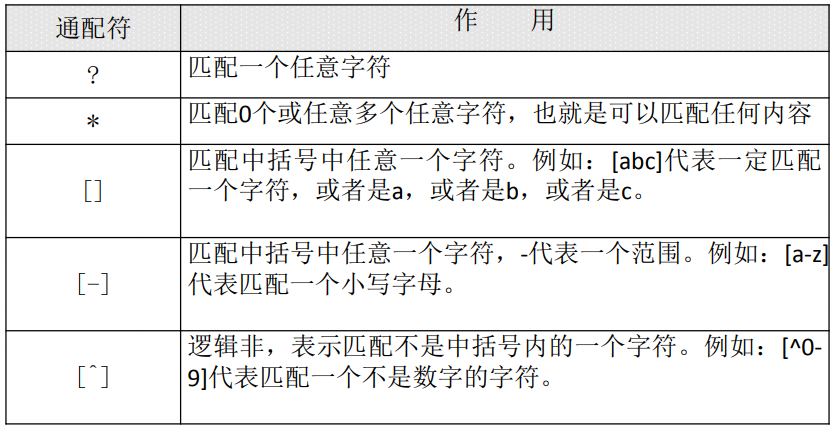
-n： 输出行号

-v： 反向查找

--color=auto 搜索出的关键字用颜色显示

## 4.5 通配符与其他特殊符号

1. 通配符



比如：

[root@localhost ~]# cd /tmp/

[root@localhost tmp]# rm -rf \*

[root@localhost tmp]# touch abc

[root@localhost tmp]# touch abcd

[root@localhost tmp]# touch 012

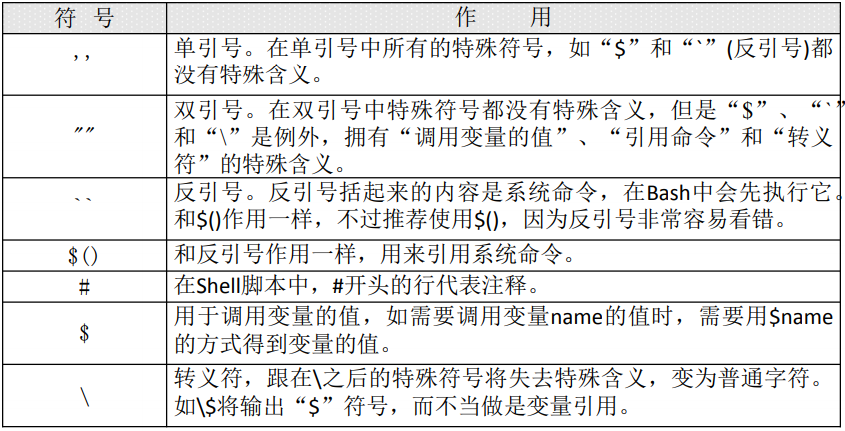
[root@localhost tmp]# touch 0abc

[root@localhost tmp]# ls ?abc

[root@localhost tmp]# ls [0-9]\*

[root@localhost tmp]# ls [^0-9]\*

2. Bash中其他特殊符号



* 反引号与$()
  + [root@localhost ~]# echo `ls`
  + [root@localhost ~]# echo $(date) 🡨 两者的用法是相同的
* 单引号和双引号
  + [root@localhost ~]# name=sc
  + [root@localhost ~]# echo '$name' 🡨 单引号中的所有内容视为字符串
  + [root@localhost ~]# echo "$name" 🡨
  + [root@localhost ~]# echo ‘$(date)'
  + [root@localhost ~]# echo “$(date)" 🡨 双引号中的 【$ ` \】具有特殊含义

# 五. Bash 中的变量

## 5.1 用户自定义变量

1. 什么是变量

* 变量是计算机内存的单元，其中存放的值可以改变。当Shell脚本需要保存一些信息时，如一个文件名或是一个数字，就把它存放在一个变量中。每个变量有一个名字，所以很容易引用它。使用变量可以保存有用信息，使系统获知用户相关设置，变量也可以用于保存暂时信息

2. 变量的设置规则

* 变量名称可以由字母、数字和下划线组成，但是**不能以数字开头**。如果变量名是 “2name” 则是错误的
* 在Bash中，变量的**默认类型都是字符串型**，**如果要进行数值运算，则必修指定变量类型为数值型**
* 变量用等号连接值，**等号左右两侧不能有空格**
* 变量的值如果有空格，需要使用单引号或双引号包括
* 在变量的值中，可以使用“\”转义符
* 如果需要增加变量的值，那么可以进行变量值的叠加。不过**变量需要用双引号包含 “$变量名”或用${变量名}包含**
  + **tmp=”&tmp”456**
  + **tmp=${tmp}789**
* 如果是把**命令的结果**作为**变量值**赋予变量，则需要使用 `反引号` 或 $() 包含命令
* **环境变量名**建议大写，便于区分

**3. 变量分类**

* 用户自定义变量
* 环境变量：这种变量中主要保存的是和系统操作环境相关的数据
* 位置参数变量：这种变量主要是用来向脚本当中传递参数或数据的，变量名不能自定义，变量作用是固定的（位置参数变量也是预定义变量的一种，因为数量过多所以单独讲解）
* 预定义变量：是Bash中已经定义好的变量，变 量名不能自定义，变量作用也是固定的

4. 本地变量

* **变量定义**
  + [root@localhost ~]# name="shen chao"
* **变量叠加**
  + [root@localhost ~]# aa=123
  + [root@localhost ~]# aa="$aa"456 🡸 两种的作用是相同的
  + [root@localhost ~]# aa=${aa}789
* **变量的调用**
  + [root@localhost ~]# echo $name
* **变量查看**
  + [root@localhost ~]# **set**
* **变量删除**
  + [root@localhost ~]# **unset name**

## 5.2 环境变量

1. 环境变量是什么

* 用户自定义变量只在**当前**的Shell中生效，而环境变量会在**当前Shell和这个Shell的所有子Shell当中**生效。如果把环境变量写入相应的配置文件，那么这个环境变量就会在所有的Shell中生效

**2. 设置环境变量**

* **声明变量 export 变量名=变量值 export-输出，出口**
* **查询变量 env**
* **删除变量 unset 变量名**

3. 常见的系统环境变量

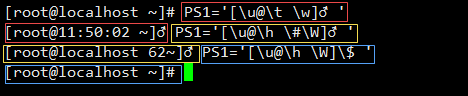
* PATH：系统查找命令的路径
  + [root@localhost ~]# echo $PATH



* + 解释：系统在执行一个命令时会在以上目录中查找，比如说ls。🡺 这就是为什么，系统命令不需要路径也可以执行的原因
* PATH变量叠加 🡪 将自定义脚本添加到、PATH路径中
  + PATH="$PATH":/root/sh
* PS1：定义系统提示符的变量
  + 查看当前的PS1 – **echo “$PS1”**

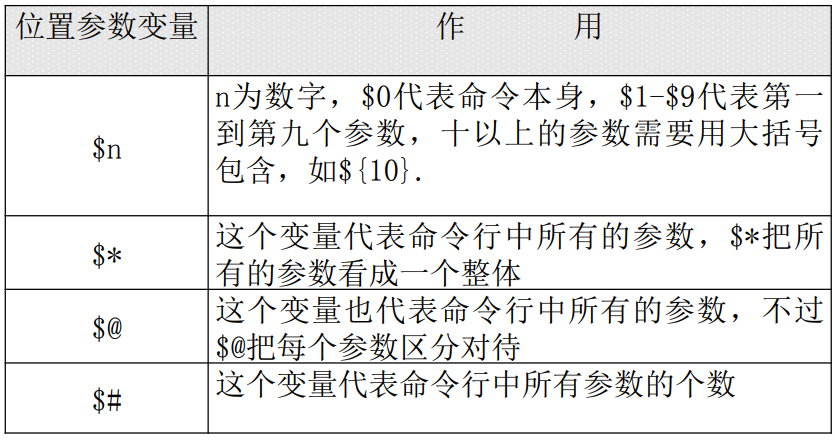


* + 举例：



## 5.3 位置参数变量

1. 位置参数变量



例一：

1. #!/bin/bash
2. num1=$1  🡨 用num1代表第一个参数
3. num2=$2  🡨 用num2代表第二个参数
4. sum=$(( $num1 + $num2))  #变量sum的和是num1加num2
5. echo $sum #打印变量sum的值

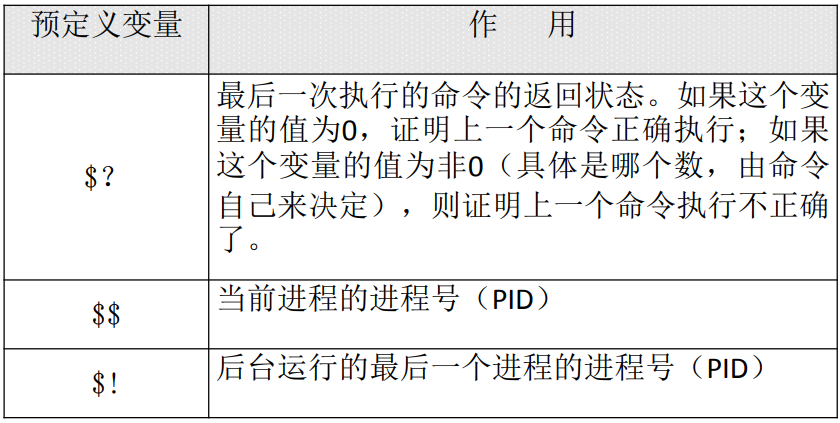
例二：

1. #!/bin/bash
2. echo "A total of $# parameters" #使用$#代表所有参数的个数
3. echo "The parameters is: $\*" #使用$\*代表所有的参数  但是看作为一个整体
4. echo "The parameters is: $@" #使用$@也代表所有参数  将参数区分对待

例三：$\*与$@的区别

1. #!/bin/bash
2. **for** i **in** "$\*"  # $\*中的所有参数看成是一个整体，所以这个**for**循环只会循环一次
3. **do**
4. echo "The parameters **is**: $i"
5. done
7. x=1
9. **for** y **in** "$@"  # $@中的每个参数都看成是独立的，所以“$@”中有几个参数，就会循环几次
10. **do**
11. echo "The parameter$x **is**: $y"
12. x=$(( $x +1 ))
13. done

## 5.4 预定义变量



1. #!/bin/bash
3. echo "The current process **is** $$"
4. #输出当前进程的PID。
5. #这个PID就是variable.sh这个脚本执行时，生成的进程的PID
7. find /root -name hello.sh &  🡸 一个$代表将命令放入后台执行
8. #使用find命令在root目录下查找hello.sh文件
9. #符号&的意思是把命令放入后台执行，工作管理我们在系统管理章节会详细介绍
11. echo "The last one Daemon process **is** $!"

2. 接收键盘输入

**接收键盘输入**

**read [选项] [变量名]**

选项：

-p **“**提示信息**”** 在等待read输入时，输出提示信息

-t 秒数 read命令会一直等待用户输入，使用此选项可以指定等待时间

-n 字符数 read命令只接受指定的字符数，就会执行

-s 隐藏输入的数据，适用于机密信息的输入

1. #!/bin/bash
3. **read -t 30 -p "Please input your name: " name** #提示“请输入姓名”并等待30秒，把用户的输入保存入变量name中
4. echo "Name **is** $name "
5. echo -e "\n"
7. **read -s -t 30 -p "Please enter your age: " age** #年龄是隐私，所以我们用“-s”选项隐藏输入
8. echo -e "Age **is** $age \n" 🡨 顺带学习下两种换行打印方式
10. **read -n 1 -t 30 -p "Please select your gender[M/F]: " gender** #使用“-n 1”选项只接收一个输入字符就会执行（都不用输入回车）
11. echo "Sex **is** $gender"

# 六. Bash 运算符

## 6.1 数值运算与运算符

1. declare声明变量类型

**declare声明变量类型**

**declare [+/-][选项] 变量名**

选项：

**-： 给变量设定类型属性**

**+： 取消变量的类型属性**

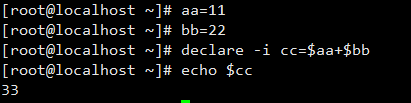
**-i： 将变量声明为整数型（integer）**

**-x： 将变量声明为环境变量 之前说过的export本质上就是添加了x选项**

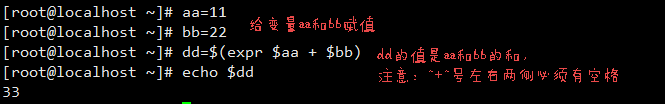
**-p： 显示指定变量的被声明的类型**

2. 数值运算

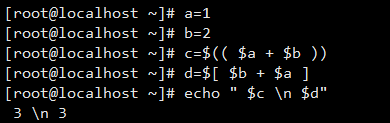
* 方法一



* 方法二 – expr 或 let 数值运算工具 【不推荐】



* **方法三 - $((运算式)) 或 $[] 【推荐】**

 🡨 有没有空格都可以

3. 运算符

**aa=$(( (11+3)\*3/2 ))**

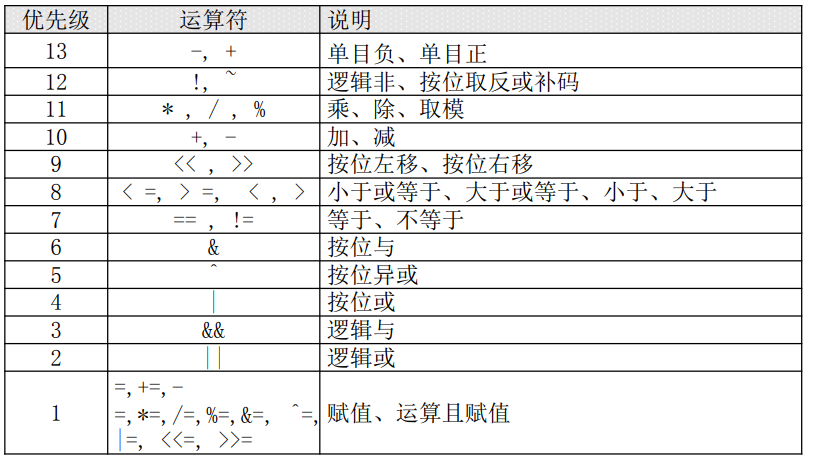
#虽然乘和除的优先级高于加，但是通过小括号可以调整运算优先级

**bb=$(( 14%3 ))**

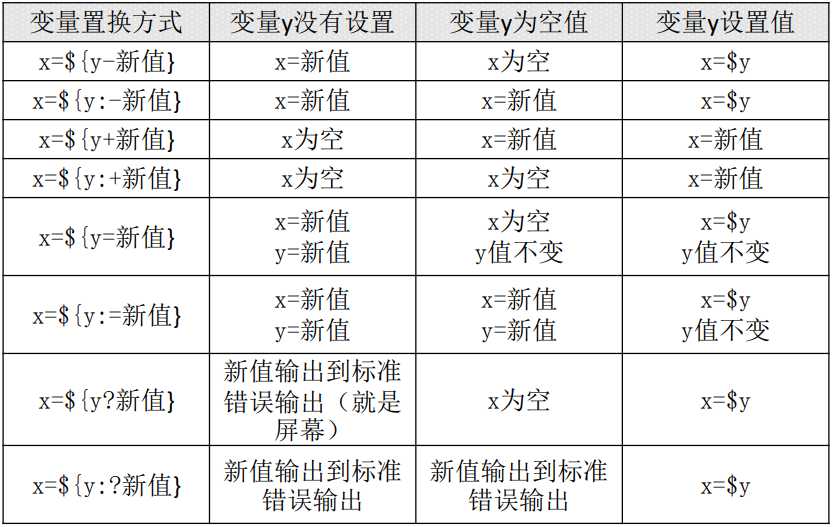
#14不能被3整除，余数是2

**cc=$(( 1 && 0 ))**

#逻辑与运算只有想与的两边都为真（不为0），与的结果才是1，否则与的结果是0



## 6.2 变量的测试与内容替换



* 测试一
  + [root@localhost ~]# unset y  #删除变量y
  + [root@localhost ~]# x=${y-new}  #进行测试
  + [root@localhost ~]# echo $x  #因为变量y不存在，所以x=new
  + new
* 测试二
  + [root@localhost ~]# y=""  #给变量y赋值为空
  + [root@localhost ~]# x=${y-new}  #进行测试
  + [root@localhost ~]# echo $x  #因为y为空，所以x也为空
* 测试三
  + [root@localhost ~]# y=old  #给变量y赋值
  + [root@localhost ~]# x=${y-new}  #进行测试
  + [root@localhost ~]# echo $x   # 因为y中有值所以x等于y中的值
  + old

# 七. 环境变量配置文件

## 7.1 环境变量配置文件简介

1. source命令

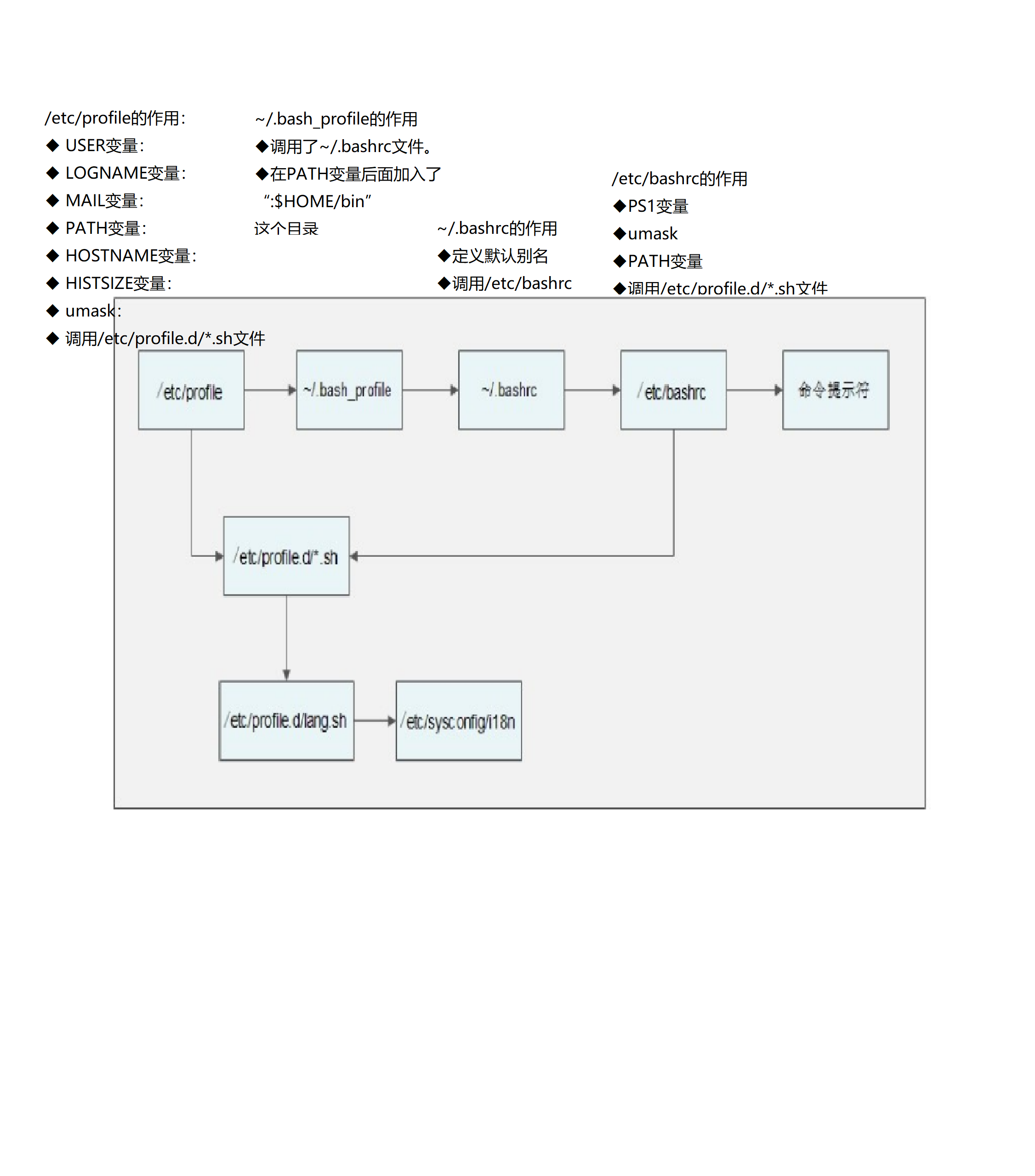
* [root@localhost ~]# source 配置文件
* [root@localhost ~]# **.** 配置文件 🡨 让环境配置文件立刻生效，两个的作用是一样的

2. 环境变量配置文件简介

* 环境变量配置文件中主要是定义对系统的操作环境生效的系统默认环境变量，比如 PATH、HISTSIZE、PS1、HOSTNAME等 默认环境变量
* /etc/profile
* /etc/profile.d/\*.sh
* ~/.bash\_profile
* ~/.bashrc
* /etc/bashrc
* 以上文件中，/etc/…… 下的对所有用户生效，而~/ 下的仅对某个用户生效，因为此文件存放于此用户的家目录

## 7.2 环境变量配置文件作用

* /etc/profile
* /etc/profile.d/\*.sh
* ~/.bash\_profile
* ~/.bashrc
* /etc/bashrc
* 以上文件中，/etc/…… 下的对所有用户生效，而~/ 下的仅对某个用户生效，因为此文件存放于此用户的家目录



## 7.3 其他配置文件和登录信息

1. 注销时生效的环境变量配置文件

* ~/.bash\_logout

2. 其他配置文件

* ~/bash\_history 🡪 比如在用户退出时将history写入相应文件

3. Shell登录信息

* 本地终端欢迎信息：/etc/issue



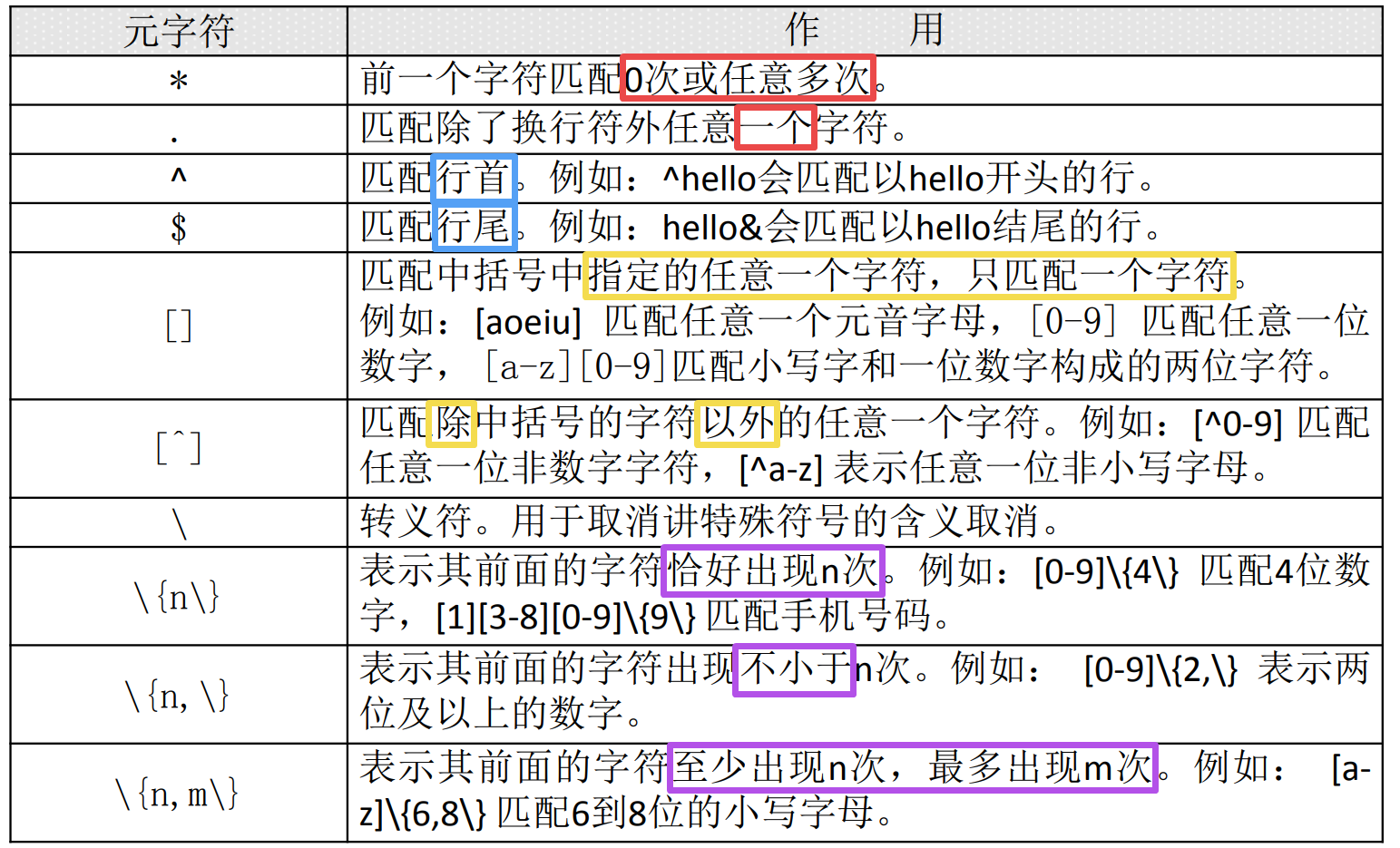
* 远程终端欢迎信息： /etc/issue.net
  + 转义符在/etc/issue.net文件中不能使用
  + 是否显示此欢迎信息，由ssh的配置文件 /etc/ssh/sshd\_config决定，加入“Banner /etc/issue.net”行才能显示（记得重启SSH服务）

# 八. 正则表达式

1. 正则表达式与通配符

* 正则表达式用来在文件中匹配符合条件的字符串，**正则是包含匹配**。grep、awk、 sed等命令可以支持正则表达式
* 通配符用来匹配符合条件的文件名，**通配符是完全匹配**。ls、find、cp这些命令不 支持正则表达式，所以只能使用shell自己 的通配符来进行匹配了

2. 正则表达式基础



* **“\*”前一个字符匹配0次，或任意多次**
  + grep "a\*" test\_rule.txt #匹配所有内容，包括空白行
  + grep "aa\*" test\_rule.txt #匹配至少包含有**一个a**的行
  + grep "aaa\*" test\_rule.txt #匹配最少包含**两个连续a**的字符串
  + grep "aaaaa\*" test\_rule.txt #则会匹配最少包含四个个连续a的字符串
* **“.” 匹配除了换行符外任意一个字符**
  + grep "s..d" test\_rule.txt #“s..d”会匹配在s和d这两个字母之间一定有两个字符的单词
  + grep "s.\*d" test\_rule.txt #匹配在s和d字母之间有任意字符
  + grep ".\*" test\_rule.txt #匹配所有内容
* **“^”匹配行首，“$”匹配行尾**
  + grep "^M" test\_rule.txt #匹配以大写“M”开头的行
  + grep "n$" test\_rule.txt #匹配以小写“n”结尾的行
  + grep -n "^$" test\_rule.txt #会匹配空白行
* **“[]” 匹配中括号中指定的任意一个字符，只匹配一个字符**
  + grep "s[ao]id" test\_rule.txt #匹配s和i字母中，要不是a、要不是o
  + grep "[0-9]" test\_rule.txt #匹配任意一个数字
  + grep "^[a-z]" test\_rule.txt #匹配用小写字母开头的行
* **“[^]” 匹配除中括号的字符以外的任意一个字符**
  + grep "^[^a-z]" test\_rule.txt #匹配不用小写字母开头的行
  + grep "^[^a-zA-Z]" test\_rule.txt #匹配不用字母开头的行
* **“\” 转义符**
  + grep "\.$" test\_rule.txt #匹配使用“.”结尾的行
* **“\{n\}”表示其前面的字符恰好出现n次和\{n,n\}是一样的**
  + grep "a\{3\}" test\_rule.txt #匹配a字母连续出现三次的字符串
  + grep "[0-9]\{3\}" test\_rule.txt #匹配包含连续的三个数字的字符串
* **“\{n,\}”表示其前面的字符出现不小于n次**
  + grep "^[0-9]\{3,\}[a-z]" test\_rule.txt #匹配最少用连续三个数字开头的行
* **“\{n,m\}”匹配其前面的字符至少出现n次，最多出现m次**
  + grep "sa\{1,3\}i" test\_rule.txt #匹配在字母s和字母i之间有最少一个a，最多三个a

# 九. 字符串截取命令

## 9.1 cut字段提取命令

**cut截取命令**

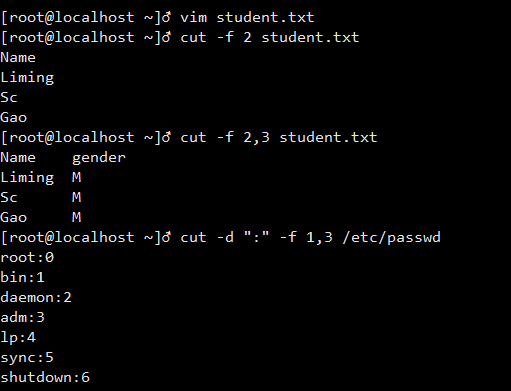
**cut [选项] 文件名**

选项：

-f 列号： 提取第几列

-d 分隔符： 按照指定分隔符分割列

**cut 是截取列**



举例：

[root@localhost ~]# vi student.txt

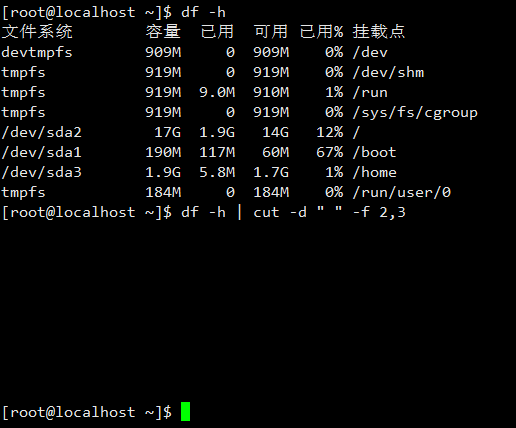
ID Name gender Mark

1 Liming M 86

2 Sc M 90

3 Gao M 83

**cut命令的局限性**

* 如果遇到以空格为分隔的文件或者命令输出，则不能正确的进行分隔
* 比如df命令的输出就是用空格分隔的
* 

## 9.2 printf命令

* print和print**f**的区别
  + 在awk命令的输出中支持print和printf命令
  + **print**：print会在每个输出之后自动加入一个换行符（Linux默认没有print命令）
  + **printf**：printf是标准格式输出命令，并不会自动加入换行符，如果需要换行，需要手工加入换行符
  + 总结：print会在输出完自动输出回车，而printf不会

**printf**

**printf ’输出类型输出格式’ 输出内容**

选项：

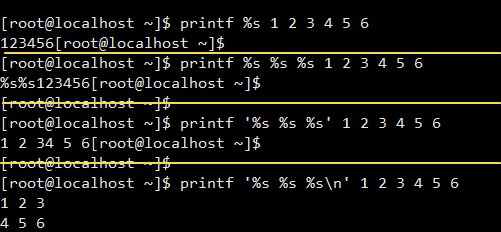
**%ns**  输出**字符串**。n是数字指代输出几个字符

**%ni**  输出**整数**。n是数字指代输出几个数字

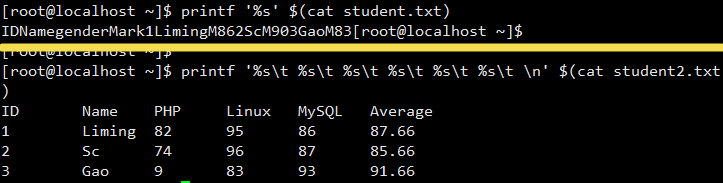
**%m.nf** 输出**浮点数**。m和n是数字，指代输出的整数位数和小数位数。

如%8.2f代表共输出8位数，其中2位是小数，6位是整数

* 输出格式
  + \a 输出警告声音
  + \b 输出退格键，也就是Backspace键
  + \f 清除屏幕
  + \n 换行
  + \r 回车，也就是Enter键
  + \t 水平输出退格键，也就是Tab键
  + \v 垂直输出退格键，也就是Tab键
* 实验一



* 实验二



## 9.3 awk命令

**awk**

**awk ‘条件1{动作1} 条件2{动作2}…’ 文件名**

条件（Pattern）：

**一般使用关系表达式作为条件**

x > 10 判断变量 x是否大于10

x>=10 大于等于

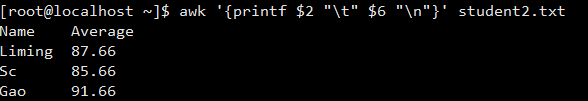
x<=10 小于等于

动作（Action）：

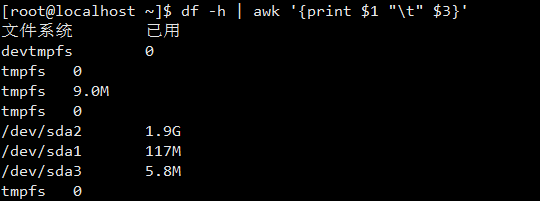
格式化输出

流程控制语句

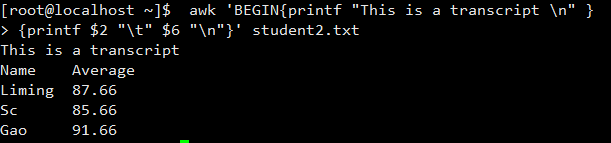
* 可以正确输出以制表符分隔的内容



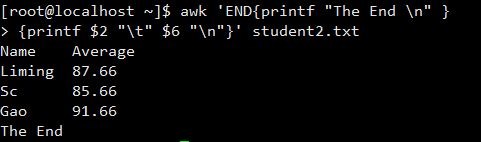
* 可以正确输出以空格作为分隔的内容



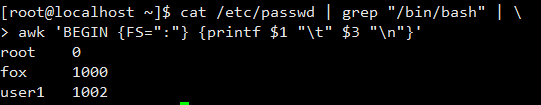
* awk默认第一行是不会按照格式输出的可以添加**BEGIN**进行强行识别
  + BEGIN表示内容的第一行



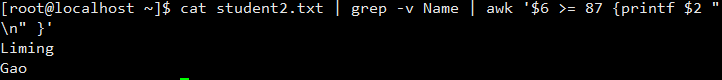
* + END表示内容尾



* **FS内置变量 🡪 指定分隔符**



* **关系运算符**



**注意：awk是一个非常强大的指令，这里只是简单介绍**

## 9.4 sed 命令

* sed 是一种几乎包括在所有 UNIX 平台（ 包括 Linux）的**轻量级流编辑器**。sed主要是用来将数据进行选取、替换、删除、新增的命令
* **sed可以在不改变文件内容的情况下进行按照条件输出**

**sed**

**sed [选项] ‘[动作]’ 文件名**

**选项：**

-n **一般sed命令会把所有数据都输出到屏幕**，如果加入此选择，则**只会把经过sed命令处理的行输出到屏幕**

-e 允许对输入数据应用多条sed命令编辑，命令之间使用“ **‘**XXXX**；**XXXX**’** ”分隔

-I **用sed的修改结果直接修改读取数据的文件，而不是由屏幕输出**

**动作：**

a \ 追加，在当前行后添加一行或多行。添加多行时，**除最后一行外**，每行末尾需要用“\”代表数据未完结

c \ 行替换，用**c后面的字符串替换原数据行**，替换多行时，**除最后一行外**，每行末尾需用“\”代表数据未完结

i \ 插入，在当期行前插入一行或多行。插入多行时，**除最后一行外**，每行末尾需要用“\”代表数据未完结

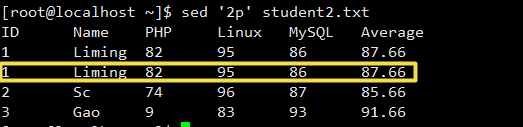
d 删除，删除指定的行

p 打印，输出指定的行

s 字串替换，用一个字符串替换另外一个字符串。

格式为“行范 围s/旧字串/新字串/g”（和vim中的替换格式类似）

* 行数据操作
  + 查看文件的第二行

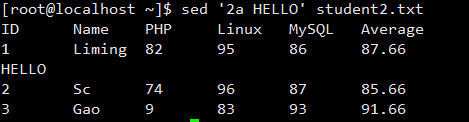




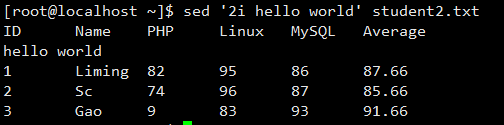
* + 删除第二行到第四行的数据，但不修改文件本身



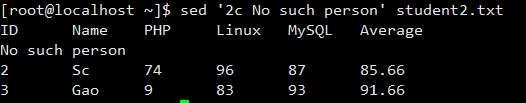
* + 在第二行后追加hello



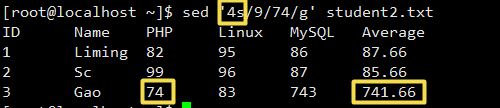
* + 在第二行前插入两行数据



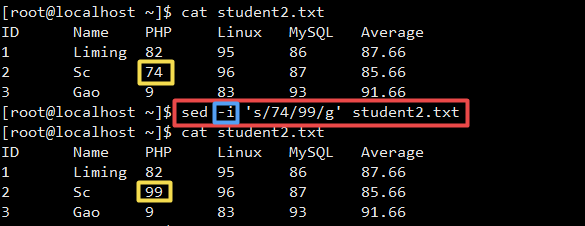
* + 数据替换



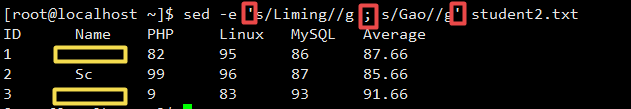
* 字符串替换：**sed ‘s/旧字串/新字串/g’ 文件名**
  + 在**第四行**中，把9换成74



* + sed操作的数据直接写入文件



* + 同时把“Liming”和“Gao”替换为空



# 十. 字符处理命令

**排序命令sort**

**sort [选项] 文件名**

选项：

-f 忽略大小写

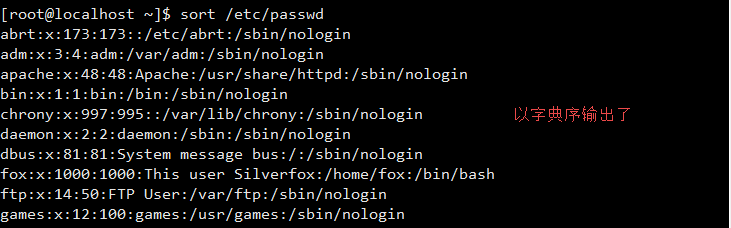
**-n 以数值型进行排序，默认使用字符串型排序**

-r 反向排序

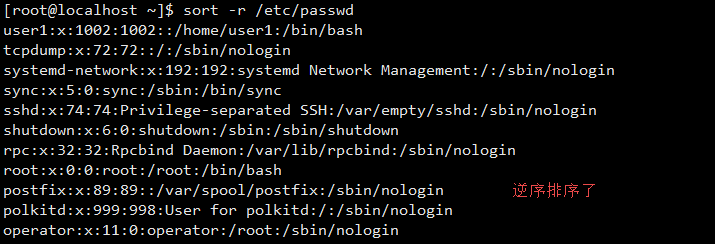
-t 指定分隔符，默认是分隔符是制表符

-k n[,m] 按照指定的字段范围排序。从第n字段开始，m字段结束（默认到行尾）

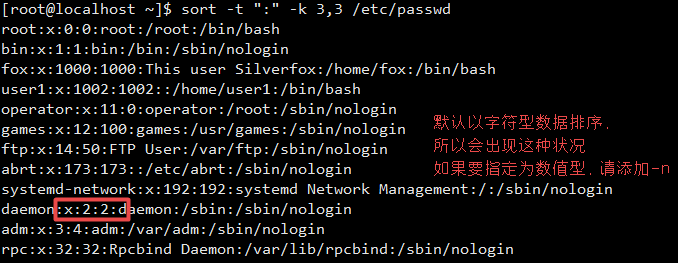
* 排序用户信息文件

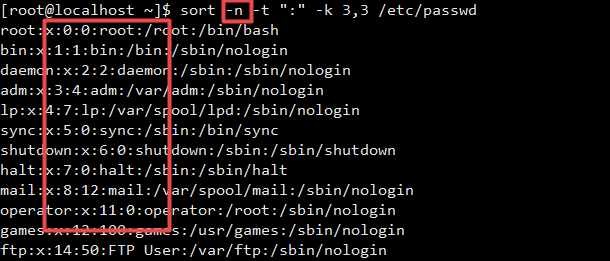


* 反向排序



* 指定分隔符是“**：**”，用第三字段开头，第三字段结尾排序，就是只用第三字段排序





**2. 统计命令wc**

**统计命令**

**wc [选项] 文件名**

选项：

-l： 只统计行数

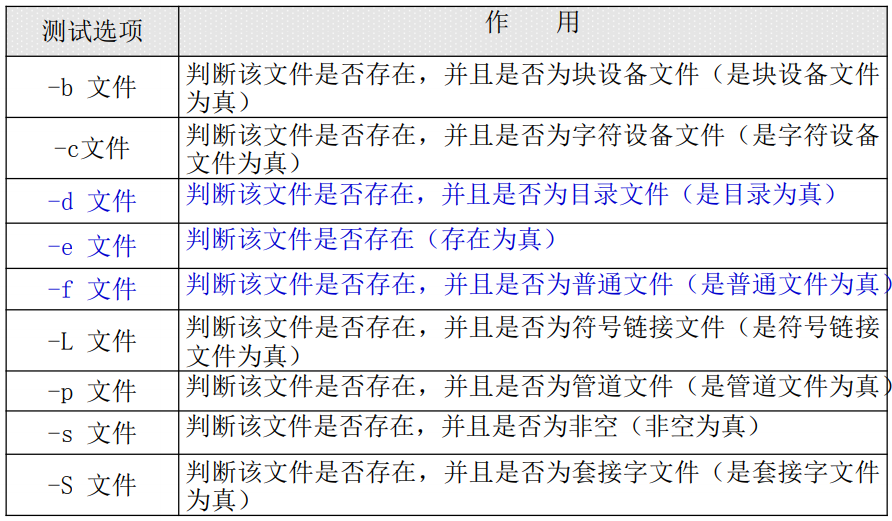
-w： 只统计单词数

-m： 只统计字符数

# 十一. 条件判断

**[ -e /root/install.log ] 🡸 条件判断建议用中括号！！并且前后要加空格！！**

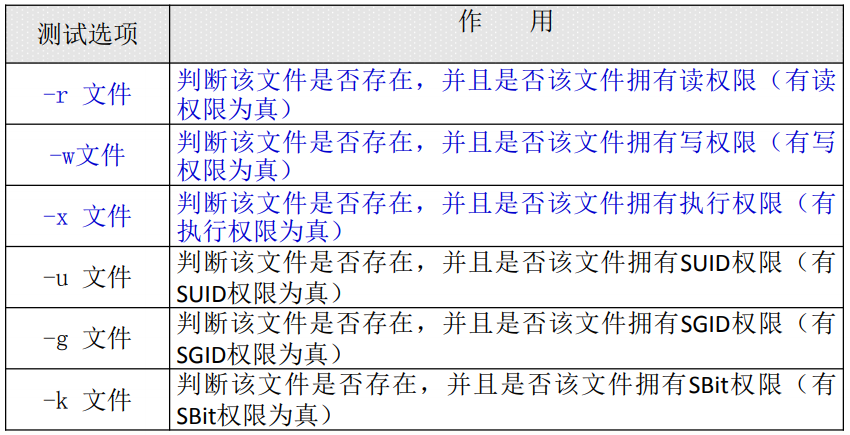
1. 按照文件类型进行判断



两种判断格式：

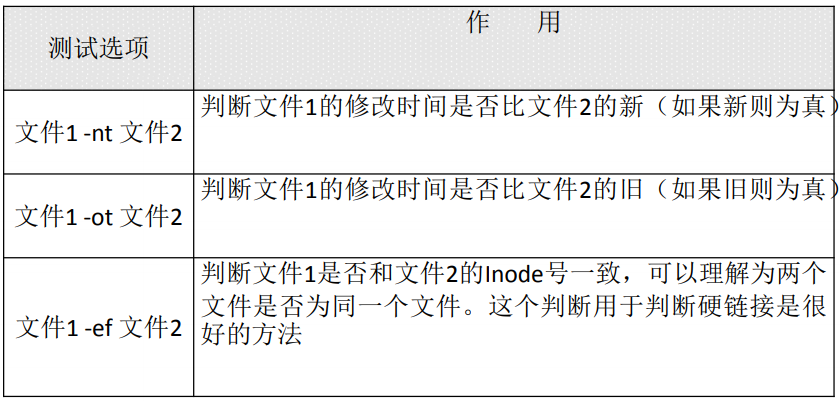
* **test -e /root/install.log**
* **[\_-e /root/install.log\_] 🡸 注意两侧的空格，如果没有，系统会无法识别命令！！！**
* [ -d /root ] && echo "yes" || echo "no"
  + #第一个判断命令如果正确执行，则打印“yes”，否则打 印“no ”

2. 按照文件权限进行判断



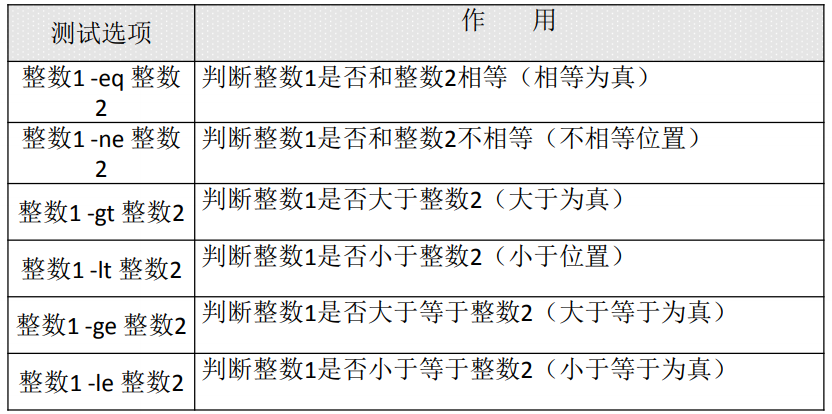
* **注意：这样判断的文件是否有某权限，只要有一个用户有即为真！**
  + 也就是说 -rw-r—r--. 为真，但是普通用户并没有写权限！

3. 两个文件之间进行比较



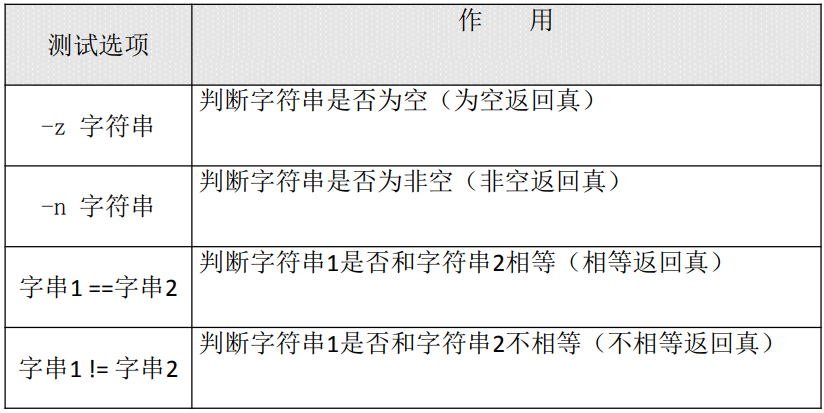
* ln /root/student.txt /tmp/stu.txt #创建个硬链接吧
* [ /root/student.txt -ef /tmp/stu.txt ] && echo "yes" || echo "no" yes #用test测试下，果然很有用

4. 两个整数之间进行比较



* [ 23 -ge 22 ] && echo "yes" || echo "no"
  + yes #判断23是否大于等于22，当然是了
* [ 23 -le 22 ] && echo "yes" || echo "no"
  + no #判断23是否小于等于22，当然不是了

5. 字符串的判断



* name=sc #给name变量赋值

[ -z "$name" ] && echo "yes" || echo "no"

no #判断name变量是否为空，因为不为空，所以返回no

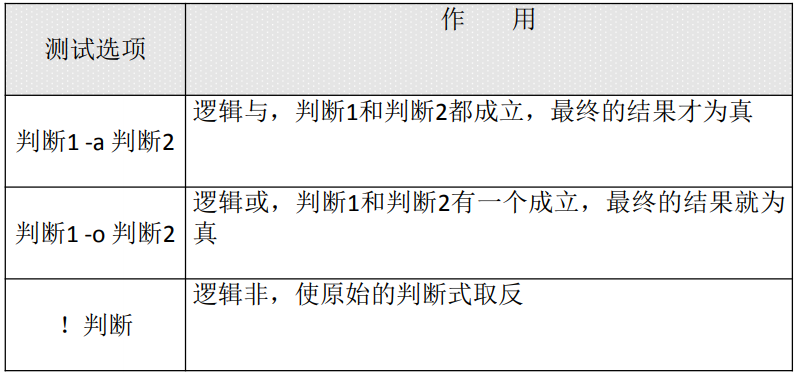
* aa=11

bb=22 #给变量aa和变量bb赋值

[ "$aa" == "bb" ] && echo "yes" || echo "no"

no #判断两个变量的值是否相等，明显不相等，所以返回no

6. 多重条件判断



* aa=11

[ -n "$aa" -a "$aa" -gt 23 ] && echo "yes" || echo "no"

no

#判断变量aa是否有值，同时判断变量aa的是否大于23

#因为变量aa的值不大于23，所以即使第一个判断值为真，返回的结果也是假

* aa=24

[ -n "$aa" -a "$aa" -gt 23 ] && echo "yes" || echo "no"

yes

# 十二. 流程控制

## 12.1 if语句

**1. 单分支if条件语句**

1. **if** [ 条件判断式 ];**then**   🡸 [] 两侧要有空格
2. 程序
3. **fi**

或者

1. **if** [ 条件判断式 ]   🡸 [] 两侧要有空格
2. **then**
3. 程序
4. **fi**

单分支条件语句需要注意的几个点

* **if**语句使用**fi**结尾，和一般语言使用大括号结尾不同
* **[ 条件判断式 ]就是使用test命令判断，所以中括号和条件判断式之间必须有空格**
* **then后面跟符合条件之后执行的程序，可以放在[]之后，用“；”分割。也可以换行写入，就不需要“；”了**

例子：

1. #!/bin/bash
2. # 统计根分区使用率
3. # 2020年8月3日10:58:44

6. # 把根分区使用率赋给rate     ↓截取第五列  ↓以百分号为分割，截取第一列
7. rate=$(df -h | grep "/dev/sda2" | awk '{print $5}' | cut -d "%" -f 1)
9. **if** [ $rate -ge 10 ]
10. **then**
11. echo "---------------------------------------"
12. echo "Warning! The rate **of** /dev/sda2 **is** $rate"
13. echo "$(date)"
14. echo "---------------------------------------"
15. fi

**2. 双分支if条件语句**

1. **if** [ 条件判断式 ]
2. **then**
3. 条件成立时，执行的程序
4. **else**
5. 条件不成立时，执行的另一个程序
6. **fi**

例子一：

备份MySQL数据库

1. #!/bin/bash
2. # 备份MySQL数据库
3. # 2020年8月3日11:29:26
5. # 同步系统时间
6. ntpdate asia.pool.ntp.org $> /dev/null # 类似于回收站
8. # 将当前系统时间赋给date
9. date=$(date +%y%m%d)
11. # 统计MySQL数据库大小，并将结果赋给size
12. size=$(du -sh /**var**/lib/mysql)
14. **if** [ -d /tmp/dbbak ]
15. **then**
16. echo "Data: $date" > /tmp/dbbak/dbinfo.txt
17. echo "Size: $size" >> /tmp/dbbak/dbinfo.txt
18. cd /tmp/dbbak
19. tar -zcf mysql-lib-$date.tar.gz /**var**/lib/mysql dbinfo.txt $> /dev/null
20. rm -rf /tmp/dbbak/dbinfo.txt
22. **else**
23. mkdir /tmp/dbbak
24. echo "Data: $date" > /tmp/dbbak/dbinfo.txt
25. echo "Size: $size" >> /tmp/dbbak/dbinfo.txt
26. cd /tmp/dbbak
27. tar -zcf mysql-lib-$date.tar.gz /**var**/lib/mysql dbinfo.txt $> /dev/null
28. rm -rf /tmp/dbbak/dbinfo.txt
29. **fi**

**例二：**

**判断Apache是否启动**

1. #!/bin/bash
2. # 判断Apache是否启动
3. # 2020年8月3日11:45:14
5. # 使用nmap命令扫描服务器，并截取Apache服务状态，赋予变量port
6. # 如果无法找到命令，请安装nmap命令
7. port=$(nmap -sT 192.168.0.6 | grep tcp | grep htpp | awk '{print $2}')
9. **if** [ "$port" == "open" ]
10. **then**
11. echo "$(date) Apache-httpd **is** running" >> /tmp/autostart-acc.log
13. **else**
14. # 适用于rpm包安装的阿帕奇
15. #/etc/rc.d/init.d/httpd start $> /dev/null
17. # 适用于源码包安装的阿帕奇
18. /usr/local/apache/bin/apachectl start &> /dev/null
20. echo "$(date) restart Apache-httpd" >> /tmp/autostart-err.log
21. **fi**

**3. 多分支if条件语句**

1. **if** [ 条件判断式1 ]
2. **then**
3. 当条件判断式1成立时，执行程序1
4. **elif**[ 条件判断式2 ]
5. **then**
6. 当条件判断式2成立时，执行程序2
7. …省略更多条件…
9. **else**
10. 当所有条件都不成立时，最后执行此程序
11. **fi**

**判断用户输入的是什么文件**

1. #!/bin/bash
2. # 判断用户输入的是什么文件
4. # 键盘接受输入，赋予变量file\_name
5. read -t 30 -p "Please input the **file** name: " file\_name
7. # 判断file\_name 是否为空
8. **if** [ -z "$file\_name"  ]
9. **then**
10. echo "Error! The **file** name can **not** be empty"
12. # 判断**file**的值是否存在
13. elif [ ! -e "$file\_name" ]
14. **then**
15. echo "Not a **file**"
16. exit 2
18. # 判断是否为普通文件
19. elif [ -f "$file\_name" ]
20. **then**
21. echo "$file\_name **is** a regulare **file**"
22. exit 3
24. elif [ -d "$file\_name" ]
25. **then**
26. echo "$file\_name **is** a directory"
27. exit 4
29. **else**
30. echo "$file\_name **is** other **file**"
32. fi

## 12.2 case语句

1. case语句和if…elif…else语句一样都是多分支条件语句，不过和if多分支条件语句不同的是，case语句只能判断一种条件关系，而if语句可以判断多种条件关系
2. **case** $变量名 **in**
3. **"值1"）**
4. 如果变量的值等于值1，则执行程序1
5. **;;**
6. **"值2"）**
7. 如果变量的值等于值2，则执行程序2
8. **;;**
9. …省略其他分支…
10. \*）
11. 如果变量的值都不是以上的值，则执行此程序
12. **;;**
13. **esac**

判断用户输入

1. #!/bin/bash
2. # 判断用户输入的是什么文件
3. # 2020年8月3日13:36:09
5. # 键盘接受输入，赋予变量file\_name
6. read -t 30 -p "Please input the **file** name: " file\_name
8. # 判断file\_name 是否为空
9. **if** [ -z "$file\_name"  ]
10. **then**
11. echo "Error! The **file** name can **not** be empty"
13. # 判断**file**的值是否存在
14. elif [ ! -e "$file\_name" ]
15. **then**
16. echo "Not a **file**"
17. exit 2
19. # 判断是否为普通文件
20. elif [ -f "$file\_name" ]
21. **then**
22. echo "$file\_name **is** a regulare **file**"
23. exit 3
25. elif [ -d "$file\_name" ]
26. **then**
27. echo "$file\_name **is** a directory"
28. exit 4
30. **else**
31. echo "$file\_name **is** other **file**"
33. fi

## 12.3 while循环与until循环

1. while循环是不定循环，也称作条件循环。只要条件判断式成立，循环就会一直继续，直到条件判断式不成立，循环才会停止。这就和for的固定循环不太一样了
2. **while** [ 条件判断式 ]
3. **do**
4. 程序
5. **done**

小程序：

1. #!/bin/bash
2. #从1加到100
4. i=1
5. s=0
6. **while** [ $i -le 100 ]
7. #如果变量i的值小于等于100，则执行循环
8. **do**
9. s=$(( $s+$i ))
10. i=$(( $i+1 ))
11. done
12. echo "The sum **is**: $s"

* until循环，和while循环相反，until循环时只要条件判断式不成立则进行循环，并执行循环程序。一旦循环条件成立，则终止循环

1. **until** [ 条件判断式 ]
2. **do**
3. 程序
4. **done**

**小程序：**

1. #!/bin/bash
2. #从1加到100
3. i=1
4. s=0
5. **until** [ $i -gt 100 ]
6. #循环直到变量i的值大于100，就停止循环
7. **do**
8. s=$(( $s+$i ))
9. i=$(( $i+1 ))
10. done
11. echo "The sum **is**: $s"