



Linux shell编程

——变量表达式及执行调试

信息工程学院



CentOS

shell 变量分为：环境变量、用户变量、预定义变量

用户变量：用户自己定义的变量，使用范围仅限于定义它的程序，执行完毕就不存在了。

环境变量分为：**系统自带环境变量**和**用户自定义环境变量**。

系统自带环境变量由系统定义对系统起作用，用户自定义环境变量自由定义对系统不起作用，但都是不会随shell执行的结束而消失。

预定义变量：BASH已经定义好的变量，其中包含位置参数变量。

shell 变量——用户变量

- ◆ 用户变量命名：
字母、数字和下划线组成，不能以数字开头。
注意：大小写的意义不同，建议小写。
- ◆ 用户变量设置： **变量名=值** （特别注意：赋值号两边不能有空格）
例如： `mydir=/home/student/wlw`
- ◆ 用户变量引用： **\$变量名**

```
$ echo $mydir  
/home/student/wlw  
$ echo mydir  
mydir  
$ echo $Mydir
```

- ◆ 给变量赋值时，=等号两边不能有空格。否则把变量当成命令，空格后的会当成命令参数处理。

```
[wlw@CM00 ~]$ a=1
[wlw@CM00 ~]$ echo "a=$a"
a=1
[wlw@CM00 ~]$ a = 1
bash: a: command not found
```

- ◆ 变量名= “字符串1 字符串2” ——此形式是变量值含有空格、制表符或换行符时。

说明：因为SHELL中对变量赋值时，等号=两边如果有空格，会把变量当成命令，空格后的字符当参数处理。

```
[wlw@CM00 ~]$ a=hello world
bash: world: command not found
[wlw@CM00 ~]$ a="hello world"
```

变量引用的注意点

- ◆ **`$ ( )`**——此形式是在把命令的结果作为变量值赋予变量时。

```
[wlw@CM00 ~]$ pwd
/home/wlw
[wlw@CM00 ~]$ p=$ (pwd)
[wlw@CM00 ~]$ echo $p
/home/wlw
```

- ◆ **`${ }`**——此形式是变量值须出现在长字符串的开头或中间，为了使变量名与其后字符区分开时。

```
[wlw@CM00 ~]$ a=hello
[wlw@CM00 ~]$ echo $aworld

[wlw@CM00 ~]$ echo ${a}world
helloworld
_
```

shell 变量

◆ 环境变量的命名

- 系统自带环境变量的命名由系统定义，不可更改。

常用系统自带环境变量：

HOME: 用户主目录的全路径名。如/home/may
PWD: 用户当前工作目录的路径。
PATH: shell 查找命令的目录列表，目录名用冒号隔开
USER: 保存当前的用户名
.....

- 用户自定义环境变量的命名规则与用户变量相同，但在命名时习惯上用大写字母以便与命令区分开。

```
[wlw@CM00 ~]$ echo curpath=$PWD
curpath=/home/wlw
[wlw@CM00 ~]$ echo curpath=$pwd
curpath=
[wlw@CM00 ~]$
```

◆ 环境变量设置

- 用户自定义环境变量的设置: **export 变量名=变量值**
- 用户自定义环境变量的引用: **\$变量名**
 - 对于用户定义的环境变量可以被当前Shell下启动的子Shell继承, 而用户变量不被继承;

```
[wlw@CM00 ~]$ a=1
[wlw@CM00 ~]$ echo $a
1
[wlw@CM00 ~]$ export B=2
[wlw@CM00 ~]$ echo $B
2
```

影响父进程变量值, 环境变量独立于子进程

```
[wlw@CM00 ~]$ bash
[wlw@CM00 ~]$ pstree | grep bash
    |-gnome-terminal-+-bash---bash-+-grep
[wlw@CM00 ~]$ echo $a

[wlw@CM00 ~]$ echo $B
2
```

```
[wlw@CM00 ~]$ pstree | grep bash
    |-gnome-terminal-+-bash-+-grep
```

```
[wlw@CM00 ~]$ echo $c
3
[wlw@CM00 ~]$ echo $B
2
[wlw@CM00 ~]$ █
```

```
[wlw@CM00 ~]$ c=3
[wlw@CM00 ~]$ echo $c
3
[wlw@CM00 ~]$ B=4
[wlw@CM00 ~]$ echo $B
4
[wlw@CM00 ~]$ exit
exit
```



◆ 环境变量设置

- 系统自带环境变量的设置：系统已定义（重点记住定义的含义）
- 系统自带环境变量的引用：\$变量名

例如：环境变量PATH的作用

```
[root@localhost ~]# ls  
abc  anaconda-ks.cfg  err  hello.sh  install.log  install.log.syslog  ok  
[root@localhost ~]#  
[root@localhost ~]#  
[root@localhost ~]# ./hello.sh  
111111111111111111111111!  
[root@localhost ~]#  
[root@localhost ~]#  
[root@localhost ~]# cp  hello.sh  /bin/  
[root@localhost ~]#  
[root@localhost ~]#  
[root@localhost ~]# hello.sh  
111111111111111111111111!
```

例如：想将一个路径加入到\$PATH中。

```
[root@localhost sh]# ls
hello.sh
[root@localhost sh]#
[root@localhost sh]#
[root@localhost sh]# hello.sh
-bash: /bin/hello.sh: 没有那个文件或目录
[root@localhost sh]#
[root@localhost sh]#
[root@localhost sh]# PATH="$PATH":/root/sh
[root@localhost sh]#
[root@localhost sh]#
[root@localhost sh]# echo $PATH
/usr/lib64/qt-3.3/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/root/bin:/root/sh
[root@localhost sh]#
[root@localhost sh]#
[root@localhost sh]# hello.sh
11111111111111111111111111111111!
[root@localhost sh]#
```

Linux下环境变量设置的三种方法

例如：想将一个路径加入到\$PATH中，可以像下面这样做：

- ◆ **export** 变量名=变量值，只对当前的shell起作用；

#export PATH="\$PATH:/NEW_PATH" (关闭shell会还原为原来的值)

- ◆ **修改 /etc/profile 文件**，该文件对所有用户有效，可能会给系统带来安全性问题。

vi /etc/profile

最后加入一行：**export PATH="\$PATH:/NEW_PATH"**

- ◆ **修改 ~/.bashrc 文件**，该文件在用户主目录下只对该用户有效，这种方法更为安全。

vi /home/may/.bashrc

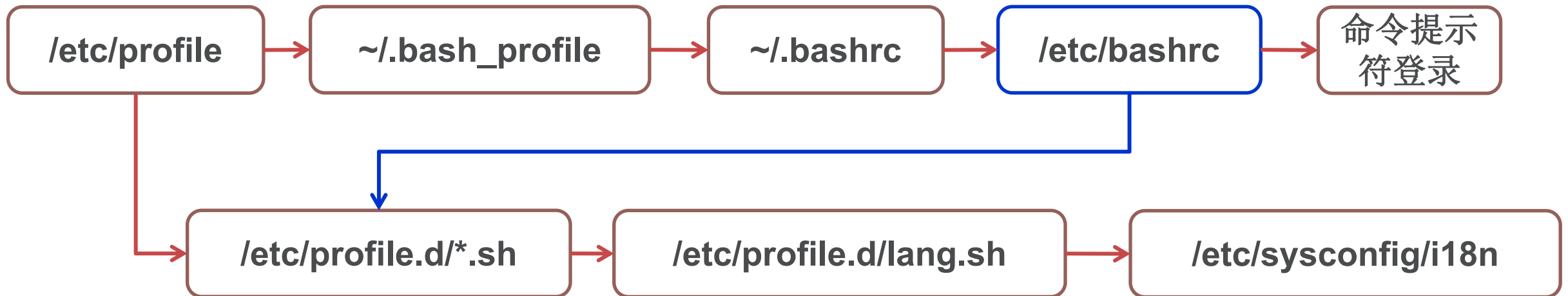
最后加入一行：**export PATH="\$PATH:/NEW_PATH"**

后两种方法修改之后执行命令：**source 文件名**

//在当前shell环境下读取并执行文件，一般需要重新注销系统才能生效。

登录时环境变量-补充

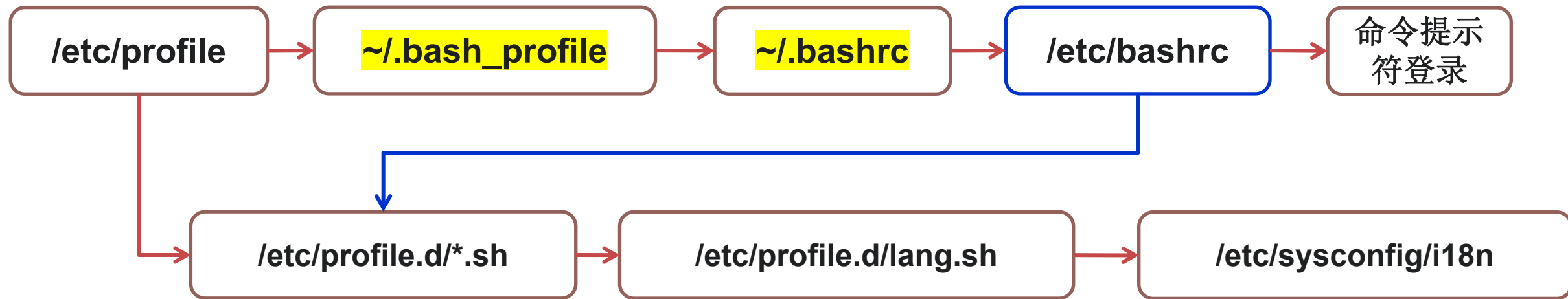
- ◆ Linux系统登录时主要生效的环境变量配置文件：
 - `/etc/profile`、`/etc/profile.d/*.sh`、`/etc/bashrc` 对所有用户生效
 - `~/.bash_profile`、`~/.bashrc` 对当前用户生效
- ◆ Linux系统环境变量配置文件的调用顺序
 - 红色线路是正常登录的环境变量调用顺序
 - 蓝色线路是非登录的环境变量调用顺序：
su切换到新用户、父bash→子bash



登录时环境变量-补充

◆ 正常登录的环境变量调用顺序

- 先由/etc/profile一直调用到/etc/sysconfig/i18n加载语言环境
- 再由/etc/profile调用~/.bash_profile加载PATH路径等，再调用~/.bashrc加载alias别名，再调用/etc/bashrc加载PS1确定提示符格式



当提示符出现 -bash-4.1#这种形式时，是什么原因？

- 可能误删了/etc/bashrc不能加载PS1，但通常不会删系统目录中的文件；
- 常见的原因是误删了主目录下的~/.bash_profile和~/.bashrc，因此也无法调用/etc/bashrc。

登录时环境变量-补充

- ◆ `/etc/issue` 保存登录时的欢迎信息，支持转义符
 - `/r` 显示内核版本 `/m` 显示硬件体系结构（默认使用）
 - `/d` 显示当前系统日期 `/s` 显示操作系统名称
 - `/n` 显示主机名 `/o` 显示域名
 - `/l` 显示登录终端号（常增加使用）

```
CentOS release 6.8 (Final)
Kernel \r on an \m
~
```

```
CentOS release 6.8 (Final)
Kernel 2.6.32-642.el6.x86_64 on an x86_64

localhost login:
```

```
CentOS release 6.8 (Final)
Kernel \r on an \m \l
~
```

```
CentOS release 6.8 (Final)
Kernel 2.6.32-642.el6.x86_64 on an x86_64 tty1

localhost login: _
```

shell变量

- ◆ 删除变量: `unset NAME`
 - 对于定义为`readonly`的变量, 不能`unset`。
 - 不要`unset`系统变量
- ◆ 查看环境变量和用户变量的方法:
 - `env`命令, 只显示环境变量。
 - `set`命令, 显示环境变量和用户变量。
 - **-u** 调用未声明变量时会报错 (默认无任何提示)

```
[wlw@CM00 ~]$ echo $file  
[wlw@CM00 ~]$ file=""  
[wlw@CM00 ~]$ echo $file  
[wlw@CM00 ~]$
```

为什么显示**name**
变量为报错?

```
[wlw@CM00 ~]$ set -u  
[wlw@CM00 ~]$ echo $name  
bash: name: unbound variable  
[wlw@CM00 ~]$ name=""  
[wlw@CM00 ~]$ echo $name  
[wlw@CM00 ~]$
```

用户变量与环境变量的区别

- ◆ 配置文件不同：
 - `/etc/bashrc` 配置用户变量
 - `/etc/profile` 配置环境变量
- ◆ 作用范围不同：
 - 用户变量是局部的，只在特定的 **shell** 中作用；
 - 环境变量是全局的，是在不同的 **shell** 中作用，设置好的环境变量可以被当前用户所运行的所有程序所使用。
- ◆ 设置方式不同
 - 环境变量用 **export** 设置，用户变量不用
- ◆ 查看命令不同
 - 环境变量用 **env**，用户变量用 **set**

Shell预定义变量

变量名	含义
\$0	脚本文件本身的名字
\$1, \$2...	第一、第二个命令行参数
“\$*”	将所有命令行参数作为一个整体单词看待
“\$@”	将命令行参数作为多个单词看待
\$#	除脚本名外，命令行上参数的个数
\$?	最后一次执行命令的返回码，0：正确执行，1：通用错误，126：命令或脚本没有执行权限，127：命令没找到
\$\$	当前进程的进程号PID
#!	后台运行的最后一个进程的进程号PID
\$_	SHELL启动时为正在执行的程序员的绝对路径，之后为上一条命令的最后一个参数。

```
[wlw@CM00 ~]$ pwd
/home/wlw
[wlw@CM00 ~]$ echo $?
0
[wlw@CM00 ~]$ PWD
bash: PWD: command not found
[wlw@CM00 ~]$ echo $?
127
[wlw@CM00 ~]$ █
```

```
#prevar.sh
#!/bin/bash
echo "当前的进程号=$$"
#后台方式运行myshell.sh
sh ./myshell.sh &
echo "最后的进程号=$_"
echo "执行的值=$_"
```

```
[wlw@CM00 ~]$ sh ./prevar.sh
当前的进程号=3947
最后的进程号=3948
执行的值=0
[wlw@CM00 ~]$ hello world!
^C
[wlw@CM00 ~]$ █
```

说明：后台执行的程序，
要CTRL+C结束

位置参数变量——属于预定义变量

- ◆ 位置参数是跟在脚本名后面，用空格隔开的每个字符串。
- ◆ 位置参数用于从命令行向Shell脚本传递参数；或调用 shell 函数时为其传递参数。
- ◆ 位置参数也称为命令行参数，与脚本中位置变量的对应关系：

```
[wlw@CM00 ~]$ ./add.sh 11 22
```

\$0	\$1	\$2	\$3	\$4	\$5	\$6	\$7	\$8	\$9	\${10}	\${11}

- \$0为脚本的名字。\$1表示第1个参数，\$2表示第二个参数，依次类推。从\${10}开始，参数号要用大括号括起来。
- 通过命令行上对应位置的实参传值。
- 位置变量不能通过一般赋值的方式直接赋值，要用set命令为位置参数赋值。



Shell预定义变量



变量名	含义
\$0	脚本文件本身的名字
\$1, \$2...	第一、第二个命令行参数，十以上参数用 { } 括起来
“\$*”	将所有命令行参数作为一个整体单词看待
“\$@”	将命令行参数作为多个单词看待
\$#	除脚本名外，命令行上参数的个数
\$?	最后一次执行命令的返回码，0：正确执行，1：通用错误，126：命令或脚本没有执行权限，127：命令没找到
\$\$	当前进程的进程号PID
\$_	后台运行的最后一个进程的进程号PID
\$_	SHELL启动时为正在执行的程序员的绝对路径，之后为上一条命令的最后一个参数。

```
#!/bin/bash
a=$1
b=$2
sum=$((a+b))
echo sum=$sum
```

分析：位置参数传参的特点？

不是给普通用户使用，一般用于程序员传参使用。

```
[w1w@CM00 ~]$ ./add.sh 11 22
```

```
sum=33
```

```
[w1w@CM00 ~]$
```

```
[w1w@CM00 ~]$ ./add.sh
```

```
./add.sh: line 4: +: syntax error: operand expected (error token is "+")
```

```
sum=
```

—

```
#!/bin/bash
#获取各个参数
echo "$0 $1 $2 $3"
echo "$*"
echo "$@"
echo "参数个数=$#"

```

```
[wlw@CM00 ~]$ sh ./posivar.sh
./posivar.sh
```

参数个数=0

```
[wlw@CM00 ~]$ sh ./posivar.sh A B C D E
./posivar.sh A B C
A B C D E
A B C D E
参数个数=5
```

“\$*” 将所有命令行参数作为一个整体单词看待:

“\$*” 等价于 "\$1 \$2 \$3..." 即 "A B C ..."

“\$@" 将命令行参数作为多个单词看待:

“\$@" 等价于 "\$1" "\$2" "\$3"..." 即 "A" "B" "C"..."

位置参数移动——shift命令

- ◆ 位置参数移动 格式: `shift [n]`
- 将位置参量列表依次左移n次, 缺省为左移一次
- 一旦位置参量列表被移动, 最左端的那个参数就会从列表中删除
- `shift`命令不能将\$0移走, 所以经`shift`右移位置参数后, \$0的值不会发生变化。
- 经常与循环结构语句一起使用, 以便遍历每一个位置参数

命令行:	<code>ex7</code>	<code>A</code>	<code>B</code>	<code>C</code>	<code>D</code>	<code>E</code>	<code>F</code>
原位置参数:	<code>\$0</code>	<code>\$1</code>	<code>\$2</code>	<code>\$3</code>	<code>\$4</code>	<code>\$5</code>	<code>\$6</code>
移位后位置参数:	<code>\$0</code>		<code>\$1</code>	<code>\$2</code>	<code>\$3</code>	<code>\$4</code>	<code>\$5</code>

```
#!/bin/bash
#获取各个参数
echo "$0 $1 $2 $3"
shift
echo "$0 $1 $2 $3"
echo "$*"
echo "$@"
echo "参数个数=$#"

```

```
[wlw@CM00 ~]$ sh ./posivar.sh
./posivar.sh
./posivar.sh
```

参数个数=0

```
[wlw@CM00 ~]$ sh ./posivar.sh 1 2 3 4 5 6
./posivar.sh 1 2 3
./posivar.sh 2 3 4
2 3 4 5 6
2 3 4 5 6
参数个数=5
```

变量小结

用户变量（小写）

- 名称：自定义 作用：自定义 内容：自定义

环境变量

用户自定义环境变量（大写）

- 名称：自定义 作用：自定义 内容：自定义

系统自带环境变量

- 名称：确定 作用：确定 内容：自定义

预定义变量（含位置参数变量）

- 名称：确定 作用：确定 内容：自定义

变量的定义不用\$，变量的引用要用\$

shell脚本中常用输入命令

◆ **read** [选项] [变量名] //读取用户输入的数据到指定变量中

选项:

- **-p “提示信息”** //在等read输入时输出提示信息
- **-t 秒数** //指定read命令等待输入的时间，否则一直等待
- **-s** //隐藏输入数据，适用于机密数据的输入
- **-n 字符数** //接受指定的字符数，如N/Y

变量名:

- **变量名可以自定义，如果未指定，则会保存入默认变量REPLY**
- **输入数据个数=变量个数，则从左至右对应赋值**
- **输入数据个数>变量个数，则从左至右对应赋值，但最后一个变量被赋予剩余的所有数据。**
- **输入数据个数<变量个数，则从左至右对应赋值，但没有数据与之对应的变量取空串**

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

```
a=$1
```

```
b=$2
```

```
sum=$((a+b))
```

```
echo sum=$sum
```

```
[wlw@CM00 ~]$ ./add.sh 11 22
```

```
sum=33
```

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

```
read -p "please input a num1:" num1
```

```
read -p "please input a num2:" num2
```

```
sum=$((num1+num2))
```

```
echo sum=$sum
```

```
[wlw@CM00 ~]$ sh add2.sh
```

```
please input a num1:11
```

```
please input a num2:22
```

```
sum=33
```

```
#readstr.sh
#!/bin/bash
echo "Please input 3 numbers:"
read x y z
echo "x=$x,y=$y,z=$z"
```

```
[wlw@CM00 ~]$ sh ./readstr.sh
Please input 3 numbers:
1 2 3
x=1,y=2,z=3
[wlw@CM00 ~]$ sh ./readstr.sh
Please input 3 numbers:
1 2 3 4 5
x=1,y=2,z=3 4 5
[wlw@CM00 ~]$ sh ./readstr.sh
Please input 3 numbers:
1 2
x=1,y=2,z=
[wlw@CM00 ~]$
```

```
#readstr.sh
#!/bin/bash
echo "Please input 3 numbers:"
read
echo "reply=$REPLY"
```

```
[wlw@CM00 ~]$ sh ./readstr.sh
Please input 3 numbers:
1 2 3
reply=1 2 3
[wlw@CM00 ~]$ sh ./readstr.sh
Please input 3 numbers:
1 2 3 4 5 6 7
reply=1 2 3 4 5 6 7
[wlw@CM00 ~]$ sh ./readstr.sh
Please input 3 numbers:
a b 1,2
reply=a b 1,2
[wlw@CM00 ~]$
```

数值运算方法——三种

◆ 用 declare 声明变量后运算

```
[wlw@CM00 ~]$ a=1  
[wlw@CM00 ~]$ b=2
```

```
[wlw@CM00 ~]$ declare -i c=$a+$b  
[wlw@CM00 ~]$ echo $c  
3
```

◆ 用 \$((运算式)) (最常用)

```
[wlw@CM00 ~]$ a=1  
[wlw@CM00 ~]$ b=2
```

```
[wlw@CM00 ~]$ echo $((a+b))  
3  
[wlw@CM00 ~]$ c=$((a+b))  
[wlw@CM00 ~]$ echo $c  
3
```

◆ 用 expr 或 let 数值运算工具

```
[wlw@CM00 ~]$ d=$((expr $a + $b))  
[wlw@CM00 ~]$ echo $d  
3  
[wlw@CM00 ~]$ d=$((expr $a+ $b))  
expr: 语法错误
```

```
[wlw@CM00 ~]$ e=$((a+b))  
[wlw@CM00 ~]$ echo $e  
1+2  
[wlw@CM00 ~]$ let e=$((a+b))  
[wlw@CM00 ~]$ echo $e  
3
```

declare 声明变量类型

◆ declare [+/-] [选项] 变量名

选项:

- - // 声明变量类型
- + // 取消变量类型声明
- -a // 声明为数组型
- -i // 声明为整型
- -x // 声明为环境变量，即export操作
- -f // 声明函数
- -p // 显示指定变量的被声明类型
- -r // 声明为只读变量，一旦声明则不能修改其值、
//不能删除该变量、不能取消只读属性，不建议用户使用。
//还好只读变量只能是命令行声明，只要重启就会消失，

◆ Bash 变量没有严格的类型定义

- 默认都是字符串型，如果要进行数值运算必须指定为数值型
- 若一个字面常量或变量的值是不包含字母或其他字符的**纯数字**的，Bash可以将其视为长整型值，并可做运算。

```
[wlw@CM00 ~]$ a=1
[wlw@CM00 ~]$ b=2
[wlw@CM00 ~]$ c=$a+$b
[wlw@CM00 ~]$ echo $c
1+2
```

—

```
[wlw@CM00 ~]$ a=1
[wlw@CM00 ~]$ b=2
[wlw@CM00 ~]$ c=$a+$b
[wlw@CM00 ~]$ echo $c
1+2
[wlw@CM00 ~]$ declare -i c=$a+$b
[wlw@CM00 ~]$ echo $c
3
```

◆ 用 declare 声明变量后运算

```
[w1w@CM00 ~]$ a=1  
[w1w@CM00 ~]$ b=2
```

```
[w1w@CM00 ~]$ declare -i c=$a+$b  
[w1w@CM00 ~]$ echo $c  
3
```

◆ 用 \$((运算式)) (最常用)

```
[w1w@CM00 ~]$ a=1  
[w1w@CM00 ~]$ b=2
```

```
[w1w@CM00 ~]$ echo $((a+b))  
3  
[w1w@CM00 ~]$ c=$((a+b))  
[w1w@CM00 ~]$ echo $c  
3
```

◆ 用 expr 或 let 数值运算工具

```
[w1w@CM00 ~]$ d=$((expr $a + $b))  
[w1w@CM00 ~]$ echo $d  
3  
[w1w@CM00 ~]$ d=$((expr $a+ $b))  
expr: 语法错误
```

```
[w1w@CM00 ~]$ e=$((a+b))  
[w1w@CM00 ~]$ echo $e  
1+2  
[w1w@CM00 ~]$ let e=$((a+b))  
[w1w@CM00 ~]$ echo $e  
3
```

\$((算术表达式))

- ◆ 只有使用 **\$((算术表达式))** 形式才能返回表达式的值

```
[wlw@CM00 ~]$ a=1
[wlw@CM00 ~]$ b=2
[wlw@CM00 ~]$ echo $((a+b))
3
[wlw@CM00 ~]$ echo $(($a+$b))
3
[wlw@CM00 ~]$ echo $((a + b))
3
```

```
[wlw@CM00 ~]$ echo ((a+b))
bash: syntax error near unexpected token `('
[wlw@CM00 ~]$ echo (($a+$b))
bash: syntax error near unexpected token `('
```

数值运算方法——三种

◆ 用 declare 声明变量后运算

```
[wlw@CM00 ~]$ a=1  
[wlw@CM00 ~]$ b=2
```

```
[wlw@CM00 ~]$ declare -i c=$a+$b  
[wlw@CM00 ~]$ echo $c  
3
```

◆ 用 \$((运算式)) (最常用)

```
[wlw@CM00 ~]$ a=1  
[wlw@CM00 ~]$ b=2
```

```
[wlw@CM00 ~]$ echo $((a+b))  
3  
[wlw@CM00 ~]$ c=$((a+b))  
[wlw@CM00 ~]$ echo $c  
3
```

◆ 用 expr 或 let 数值运算工具

```
[wlw@CM00 ~]$ d=$((expr $a + $b))  
[wlw@CM00 ~]$ echo $d  
3  
[wlw@CM00 ~]$ d=$((expr $a+ $b))  
expr: 语法错误
```

```
[wlw@CM00 ~]$ e=$((a+b))  
[wlw@CM00 ~]$ echo $e  
1+2  
[wlw@CM00 ~]$ let e=$((a+b))  
[wlw@CM00 ~]$ echo $e  
3
```

◆ expr 命令将运算表达式当作参数求值

- 运算可以是算术运算，比较运算，字符串运算和逻辑运算。
- 参数与操作符必须以空格分开。
- 表达式中+、-、%不需要转义，*、（、）需要转义

```
expr 5 % 3
```

```
expr 5 \* 3          # 乘法符号必须被转义
```

```
expr 2 + 5 \* 2 - 3 % 2
```

```
expr \( 2 + 5 \) \* 2 - 3  # 括号必须被转义
```

◆ let 命令用于算术运算: `let arg ...`

其中, `arg`是使用C语言语法的算术表达式。

如: `let "j=i*6+2"`

➤ 替代格式为: `((算术表达式))`

如: `((j=i*6+2))`

用 `let` 命令进行算术运算时, 最好加双引号。

```
let num2=4 + 1
```

```
let "num2=4 + 1" # 用引号忽略空格的特殊含义
```

当表达式中有`shell`的特殊字符时, 必须用双引号括起来。

let命令的使用说明：

- 在没有“号”的表达式中，赋值符号和运算符两边不能留空格！
- 如果将字符串赋值给一个整型变量时，则变量的值为 0
- 如果变量的值是字符串，则进行算术运算时设为 0

```
[root@localhost test]# let n=4+3;echo $n
```

```
7
```

```
[root@localhost test]# let n=4 + 3;echo $n
```

```
-bash: let: +: syntax error: operand expected (error token is "+")
```

```
4
```

```
[root@localhost test]# let "n=yes";echo $n
```

```
0
```

```
[root@localhost test]# var=yes
```

```
[root@localhost test]# let "n=$var+3";echo $n
```

```
3
```

◆ $x=\$(\dots)$ 与 $x=\$(\dots)$ 用法的区别:

- 两对圆括号用于算术替换
- 一对圆括号用于命令的执行和获取输出

例: $x=\$(x + 1)$ 与 $x=\$(\text{expr } \$x + 1)$ 运行结果相同

```
[wlw@CM00 ~]$ x=1
[wlw@CM00 ~]$ x=$((x+1))
[wlw@CM00 ~]$ echo $x
2
[wlw@CM00 ~]$ x=$(expr x + 1)
expr: 参数数目错误
[wlw@CM00 ~]$ x=$(expr $x + 1)
[wlw@CM00 ~]$ echo $x
1
[wlw@CM00 ~]$ x=$(expr $x + 1)
[wlw@CM00 ~]$ echo $x
2
_
```

运算符

+ 、 -	(单目运算)
! 、 ~	(逻辑非、按位取反或补)
**	(方幂运算符)
* 、 / 、 %	(幂运算 和 模运算，乘、除)
+ 、 -	(加、减)
<< 、 >>	(按位左移 和 按位右移)
< 、 > 、 <= 、 >=	(关系运算)
== 、 !=	(关系运算)
& 、 ^ 、 	(按位与、按位异或 和 按位 或)
&& 、 	(逻辑与 和 逻辑 或)
= 、 += 、 -= 、 *= 、 /= 、 %= 、 <<= 、 >>= 、 &= 、 ^= 、 =	(赋值运算)

其运算符及其优先级和结合性基本上与C语言的相同。
从上至下优先级越来越低。



【实验题】判断给定的某一年是否是闰年。如果某年号能被4 整除而不能被100整除、或者能被400整除，那么它就是闰年；否则是平年。



CentOS

```
$ cat leapyear
#!/bin/bash
#determing if a year is a leap year
echo "Input a year number"
read year
let "leap=year%4==0&&year%100!=0||year%400==0"
if [ $leap -eq 0 ]
then echo "$year is not a leap year. "
else echo " $year is a leap year. "
fi
$ leapyear
Input a year number
2008
2008 is a leap year.
$ leapyear
Input a year number
2010
2010 is not a leap year.
```

#提示输入一个年号
#读取输入的年号
#判断年号
#若leap等于0，则不是闰年
#输出不是闰年信息
#否则，输出闰年信息

(用户输入一个年份)
(显示运行结果)

(用户再输入一个年份)
(显示运行结果)

参数置换变量（选讲）

参数置换变量是另一种为变量赋值的方式，是根据不同条件给变量赋予不同的值。

变量置换方式	参数为空/未设置	参数非空
变量=\${参数:-字符串}	变量=字符串 参数不变	变量=参数 参数不变
	若参数为空或未设置，返回字符串，参数不变。	
变量=\${参数:=字符串}	变量=字符串 参数=字符串	变量=参数 参数不变
	若参数为空或未设置，返回字符串，同时参数设为字符串	
变量=\${参数:+字符串}	变量=空 参数不变	变量=字符串 参数不变
	若参数为空，返回字符串，参数不变。	
变量=\${参数:?字符串} 该参数可以是位置参数 (最常用)	输出”脚本名:参数:字符串”，并退出shell， 变量不变	变量=参数 参数不变
	若参数为空或未设置，则输出字符串给出的错误信息，若包含空白符，则需要用引号。	

参数置换变量（选讲）

变量=**`${参数:-word}`**和变量=**`${参数:=word}`**的区别是：

- 当参数未设置或为空时，第一种参数的值仍为空，第二种参数的值被赋了**word**的值。
- 如果参数已被设置过，则执行两种的结果一样。

```
[root@localhost ~]# name=${use:-"who"}
```

```
[root@localhost ~]# echo $name
```

```
who
```

```
[root@localhost ~]# echo $use
```

(无显示)

```
[root@localhost ~]# name=${use:="who"}
```

```
[root@localhost ~]# echo $name
```

```
who
```

```
[root@localhost ~]# echo $use
```

```
who
```

参数置换变量（选讲）

变量=\${参数:?字符串} 若参数为空或未设置，则按“脚本名:参数:字符串”格式的信息输出提示，如果省略了字符串，则显示标准信息，并从shell中退出；如果参数非空，则变量=参数。
该种方式常用于出错指示。

```
#!/bin/bash
default_num=$1
s=0
for((i=0; i<=$default_num; i++))
do
    s=$((s+i))
done
echo sum=$s
```

```
[wlw@CM00 ~]$ sh varpara.sh
```

```
varpara.sh: line 4: ((: 1<=: syntax error: operand expected (error token is "<="
)
```

```
sum=0
```

```
[wlw@CM00 ~]$ sh varpara.sh 10
```

```
sum=55
```

```
#!/bin/bash
default_num=${1:? "missing one parameter! "}
s=0
for((i=0; i<=$default_num; i++))
do
    s=$((s+i))
done
echo sum=$s
```

```
[wlw@CM00 ~]$ sh varpara.sh
```

```
varpara.sh: line 2: 1: missing one parameter!
```

```
[wlw@CM00 ~]$ sh varpara.sh 10
```

```
sum=55
```

字符串操作——补充：正则表达式

◆ 正则表达式和通配符的区别

- 正则表达式是用来匹配符合条件的字符串，如grep
- 通配符用来匹配符合条件的文件名，如ls、find等

◆ 正则表达式的常用作用

- 模糊匹配
- 过滤文本

字符串操作——补充：正则表达式

例：过滤邮箱，如wlw@163.com, 123@QQ.COM

`[0-9a-zA-Z_]+@[0-9a-zA-Z_]+(\.[0-9a-zA-Z_]+){1,3}`

- `[ ]`是匹配中括号内指定的任意一个字符，`+`是表示之前的子表达式匹配1次或多次。

`[0-9a-zA-Z_]+`即表示匹配大小写字符、数字、下划线任意多次。

- `\.`是表示匹配. 字符，`\`表示取消后继字符的特殊含义，`( )`表示括号内作为整体字符，`{n, m}`表示其前续字符匹配最少n次，最多m次。
`(\.[0-9a-zA-Z_]+){1,3}`表示匹配. 开始的字母或数字串1~3次。

字符串操作——补充：正则表达式

基本正则 使用grep “基本正则表达式” 文件名

元字符	作 用
*	前一个字符匹配 0 次或任意多次。
.	匹配除了换行符外任意一个字符。
^	匹配行首。例如：^hello 会匹配以 hello 开头的行。
\$	匹配行尾。例如：hello&会匹配以 hello 结尾的行。
[]	匹配中括号中指定的任意一个字符，只匹配一个字符。 例如：[aoeiu] 匹配任意一个元音字母，[0-9] 匹配任意一位数字， [a-z][0-9]匹配小写字和一位数字构成的两位字符。
[^]	匹配除中括号的字符以外的任意一个字符。例如：[^0-9] 匹配任意一位非数字字符，[^a-z] 表示任意一位非小写字母。
\	转义符。用于取消讲特殊符号的含义取消。
\{n\}	表示其前面的字符恰好出现 n 次。例如：[0-9]\{4\} 匹配 4 位数字， [1][3-8][0-9]\{9\} 匹配手机号码。
\{n, \}	表示其前面的字符出现不小于 n 次。例如： [0-9]\{2, \} 表示两位及以上的数字。
\{n, m\}	表示其前面的字符至少出现 n 次，最多出现 m 次。例如： [a-z]\{6, 8\} 匹配 6 到 8 位的小写字母。

{ } 号前的\，在正则表达式中不需要

元字符*——匹配前一字符0次或多次

- ◆ `grep "o*" asic.txt` 是将含有一个或多个o或不含o的都列出，相当于把所有内容都列出
- ◆ `grep "oo*" asic.txt` 是把o之后含有一个或多个o或不含o的都列出，即至少含有一个o列出。

```
[root@CM00 ~]# grep "o*" asic.txt
helloo
ok123

dog123 dog
Dooog nice

123good body
google

[root@CM00 ~]#
```

```
[root@CM00 ~]# grep "oo*" asic.txt
helloo
ok123
dog123 dog
Dooog nice
123good body
google
[root@CM00 ~]#
```

元字符*——举例说明使用特点

- ◆ 无限位符的情况时，如grep “ooo*” asic.txt与grep “oo” asic.txt结果相同，含oo及多个o的都列出，因此，可以不用*，grep “o” asic.txt即可。
- ◆ 有限位符的情况时，如grep “goo*g” asic.txt与grep “goog” asic.txt结果不同，前者含有goo及g多个o的都列出，后者只列出含有goo的。

```
[root@CM00 ~]# grep "oo" asic.txt
Doo nice
123good body
google
gooogle
gooooooogle
[root@CM00 ~]# grep "ooo*" asic.txt
Doo nice
123good body
google
gooogle
gooooooogle
[root@CM00 ~]#
```

```
[root@CM00 ~]# grep "goo*g" asic.txt
google
gooogle
gooooooogle
[root@CM00 ~]# grep "goog" asic.txt
google
```

元字符. ——匹配任意一个字符

- ◆ 正则字符中 “.” 表示任意字符，通配符中 “*” 表示任意字符

```
[ root@CM00 ~]# grep "s..d" asic.txt  
She said this article.  
He said these words.
```

```
[ root@CM00 ~]# grep "s.*d" asic.txt  
She said this article.  
He said these words.
```

元字符^ 和 \$——匹配行首 和 行尾

◆ 匹配数字开头的—— `^[0-9]`

```
[root@CM00 ~]# grep "^[0-9]" asic.txt  
123good body  
[root@CM00 ~]# █
```

◆ 匹配.结尾的—— `\.$`

```
[root@CM00 ~]# grep "\.$" asic.txt  
She said this article.  
He said these words.  
[root@CM00 ~]# █
```

◆ 匹配空行—— `^$`

```
[root@CM00 ~]# grep "^$" asic.txt
```

```
[root@CM00 ~]# █
```

如要找非空白行，用 **grep -v** “`^$`” asic.txt

元字符[]——匹配括号中的任意一个字符

- ◆ 匹配大写字母——[A-Z]
- ◆ 匹配小写字母——[a-z]
- ◆ 匹配数字——[0-9]

特别注意^在[]之内和之外的区别：

- [^]用于匹配除中括号字符以外的任意一个字符
- ^[]用于匹配行首
- ◆ 匹配非字母开头的行
 `^[^a-zA-Z]`

扩展正则 使用grep -E “扩展正则表达式” 文件名

扩展元字符	作 用
+	前一个字符匹配 1 次或任意多次。 如 “go+gle” 会匹配 “gogle”、“google” 或 “gooogle”，当然如果 “o” 有更多个，也能匹配。
?	前一个字符匹配 0 次或 1 次。 如 “colou?r” 可以匹配 “colour” 或 “color”。
	匹配两个或多个分支选择。 如 “was his” 会匹配既包含 “was” 的行，也匹配包含 “his” 的行。
()	匹配其整体为一个字符，即模式单元。可以理解为由多个单个字符组成的大字符。 如 “(dog)+” 会匹配 “dog”、“dogdog”、“dogdogdog” 等，因为被 () 包含的字符会当成一个整体。但 “hello (world earth)” 会匹配 “hello world” 及 “hello earth”。

正则表达式中的符号

- ◆ ****
将下一个字符标记为一个特殊字符、或一个原义字符、或一个 后向引用、或一个八进制转义符。
例如, 'n' 匹配字符 "n"。'\n' 匹配一个换行符。序列 '\\ 匹配 "\" 而 \"(\" 则匹配 "("。
- ◆ **^**
匹配输入字符串的开始位置。
- ◆ **\$**
匹配输入字符串的结束位置。
- ◆ *****
匹配前面的子表达式零次或多次。例如, **zo*** 能匹配 "z" 以及 "zoo"。 * 等价于 {0,}。 最少可以0个
- ◆ **+**
匹配前面的子表达式一次或多次。例如, '**zo+**' 能匹配 "zo" 以及 "zoo", 但不能匹配 "z"。 + 等价于 {1,}。 最少只有1个
- ◆ **?**
匹配前面的子表达式零次或一次。例如, "**do(es)?**" 可以匹配 "do" 或 "does" 中的"do" 。 ? 等价于 {0,1}。 最少0个, 最多一个

正则表达式中的符号

◆ {n}

n 是一个非负整数。匹配确定的 n 次。例如, 'o{2}' 不能匹配 "Bob" 中的 'o', 但是能匹配 "food" 中的两个 o。

◆ {n,}

n 是一个非负整数。至少匹配n 次。例如, 'o{2,}' 不能匹配 "Bob" 中的 'o', 但能匹配 "foooooo" 中的所有 o。'o{1,}' 等价于 'o+'。'o{0,}' 则等价于 'o*'。

◆ {n,m}

m 和 n 均为非负整数, 其中 $n \leq m$ 。最少匹配 n 次且最多匹配 m 次。 "o{1,3}" 将匹配 "foooooo" 中的前三个 o。'o{0,1}' 等价于 'o?'。请注意在逗号和两个数之间不能有空格。

◆ ?

当该字符紧跟在任何一个其他限制符 (*, +, ?, {n}, {n,}, {n,m}) 后面时, 匹配模式是非贪婪的。非贪婪模式尽可能少的匹配所搜索的字符串, 而默认的贪婪模式则尽可能多的匹配所搜索的字符串。例如, 对于字符串 "oooo", 'o+?' 将匹配单个 "o", 而 'o+' 将匹配所有 'o'。

◆ .

匹配除 "\n" 之外的任何单个字符。要匹配包括 '\n' 在内的任何字符, 请使用象 '[.\n]' 的模式。

补充: **cut** 提取列 (grep 提取行)

◆ **cut** [选项] 文件名

- **-f** 列号1, 列号n //提取第几列
- **-d** 分隔符 //按指定分隔符分隔列（默认为TAB键，不识别空格）
- **-c** 字符范围 //按字符范围来进行字段提取。“n-”从第n个字符到行尾；“-m”从第1个字符到第m个字符；“n-m”从第n个字符到第m个字符，行首

```
[root@localhost sh]# cat student.txt
```

ID	Name	gender	Mark
1	Liming	M	86
2	Sc	M	90
3	Tg	M	83

```
[root@localhost sh]# cut -f 2 student.txt
```

```
Name  
Liming  
Sc  
Tg
```

说明：文本文件中的字段分隔要用**TAB**键，**cut**不识别空格

补充: cut 提取列——举例说明

```
[root@localhost sh]# cat student.txt
```

ID	Name	gender	Mark
1	Liming	M	86
2	Sc	M	90
3	Tg	M	83

```
[root@localhost sh]# grep -v "Name" student.txt
```

1	Liming	M	86
2	Sc	M	90
3	Tg	M	83

```
[root@localhost sh]# grep -v "Name" student.txt | cut -f 2
```

Liming
Sc
Tg

```
[root@localhost sh]# grep -v "Name" student.txt | cut -f 2,4
```

Liming	86
Sc	90
Tg	83

补充: cut 提取列——举例说明

```
[root@localhost sh]# cut -d ":" -f 1,3 /etc/passwd
root:0
bin:1
daemon:2!
adm:3
lp:4
sync:5
shutdown:6
halt:7
mail:8
```

这提取了第一列，但没法判断是否是普通用户。

```
[root@localhost sh]# grep "/bin/bash" /etc/passwd | grep -v "root"
user1:x:500:500:~/home/user1:/bin/bash
user2:x:501:501:~/home/user2:/bin/bash
[root@localhost sh]#
[root@localhost sh]# grep "/bin/bash" /etc/passwd | grep -v "root" | cut -d ":" -f 1
user1
user2
```

这先过滤掉系统用户，再过滤`root`用户的信息，即留下普通用户，再提取第一列即用户名。



补充：awk编程——比cut强大，能识别空格分隔符

```
[root@localhost ~]# 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。

动作 (Action)：格式化输出

流程控制语句

格式化输出：一般用printf和print。

因为print在linux中不能直接运行，一般常用printf。

printf在管道后无法识别，因此，它一般都用于awk中。

```
[root@localhost ~]# printf '输出类型输出格式' 输出内容
```

输出类型：

%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

补充：awk编程——printf格式化输出

例：无条件输出student.txt文件中的第二和第六列。

```
[root@localhost sh]# awk '{printf $2 "\t" $6 "\n"}' student.txt
```

```
Name      Average
Lm         87.66
Sc         85.66
Tg         91.66
```

I

```
[root@localhost sh]# cat student.txt
ID      Name      PHP      Linux      MySQL      Average
1       Liming    82       95        86        87.66
2       Sc       74       96        87        85.66
3       Tg       99       83        93        91.66
```

- 说明：1、在awk的条件动作作用单引号 ‘ ’ 括起来，因此，printf或print中的输出格式要用双引号 “ ” 括起来。
- 2、print不用加\n会自动换行，printf则需要加\n才行。

```
[root@localhost sh]# awk '{print $2 "\t" $6 }' student.txt
```

```
Name      Average
Lm         87.66
Sc         85.66
Tg         91.66
```

```
[root@localhost sh]# awk '{printf $2 "\t" $6}' student.txt
```

```
Name      AverageLm         87.66Sc 85.66Tg 91.66 [root@localhost sh]#
```

补充实例：提取对某系统硬盘占用率超过80%的数据。

```
[root@localhost sh]# df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda3        19G   2.1G   16G   12% /
tmpfs            495M    0   495M    0% /dev/shm
/dev/sda1        190M   33M   147M   19% /boot
```

直接提取的Use列，没办法区分是哪个系统的。

```
[root@localhost sh]# df -h | awk '{print $5}'
Use%
12%
0%
19%
```

先过滤出系统的行，再提取相应列，得到一个带%的数，判断时只要数字，再%作分隔符提取前面的数字，则可以将运行结果送到变量去判断运行。

```
[root@localhost sh]# df -h | grep "dev/sda3"
/dev/sda3        19G   2.1G   16G   12% /
[root@localhost sh]#
[root@localhost sh]#
[root@localhost sh]# df -h | grep "dev/sda3" | awk '{print $5}'
12%
[root@localhost sh]# df -h | grep "dev/sda3" | awk '{print $5}' | cut -d "%" -f 1
12
```

补充：awk编程

```
[root@localhost ~]# 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。

动作 (Action)：格式化输出

流程控制语句

条件的类型	条 件	说 明
awk 保留字	BEGIN	在 awk 程序一开始时，尚未读取任何数据之前执行。BEGIN 后的动作只在程序开始时执行一次
	END	在 awk 程序处理完所有数据，即将结束时执行。END 后的动作只在程序结束时执行一次
关系运算符	>	大于
	<	小于
	>=	大于等于
	<=	小于等于
	==	等于。用于判断两个值是否相等，如果是给变量赋值，请使用“=”号
	!=	不等于
	A~B	判断字符串 A 中是否包含能匹配 B 表达式的子字符串
正则表达式	A!~B	判断字符串 A 中是否不包含能匹配 B 表达式的子字符串
	/正则/	如果在“//”中可以写入字符，也可以支持正则表达式

补充实例：查看平均成绩大于86的学员。

```
[root@localhost sh]# cat student.txt
```

ID	Name	PHP	Linux	MySQL	Average
1	Lm	82	95	86	87.66
2	Sc	74	96	87	85.66
3	Tg	99	83	93	91.66

```
[root@localhost sh]# grep -v "Name" student.txt
```

1	Lm	82	95	86	87.66
2	Sc	74	96	87	85.66
3	Tg	99	83	93	91.66

```
[root@localhost sh]# grep -v "Name" student.txt | awk ' $6 >=86 {print $2}'
```

Lm

Tg

加入了条件之后，只有条件成立动作才会执行，如果条件不满足，则动作则不运行。

虽然 awk 是列提取命令，但是也要按行来读入的。这个命令的执行过程是这样的：

1) 如果有 BEGIN 条件，则先执行 BEGIN 定义的动作

2) 如果没有 BEGIN 条件，则读入第一行，把第一行的数据依次赋予\$0、\$1、\$2 等变量。其中\$0 代表此行的整体数据，\$1 代表第一字段，\$2 代表第二字段。

2) 依据条件类型判断动作是否执行。如果条件符合，则执行动作，否则读入下一行数据。如果没有条件，则每行都执行动作。

3) 读入下一行数据，重复执行以上步骤。

补充: awk编程

条件的类型	条 件	说 明
awk 保留字	BEGIN	在 awk 程序一开始时, 尚未读取任何数据之前执行。BEGIN 后的动作只在程序开始时执行一次
	END	在 awk 程序处理完所有数据, 即将结束时执行。END 后的动作只在程序结束时执行一次

```
[root@localhost sh]# awk 'BEGIN{print 1111111111111111} {print $2 "\t" $6}' student.txt
1111111111111111
Name      Average
Lm        87.66
Sc        85.66
Tg        91.66
I

[root@localhost sh]#
[root@localhost sh]# awk 'END{print 1111111111111111} {print $2 "\t" $6}' student.txt
Name      Average
Lm        87.66
Sc        85.66
Tg        91.66
1111111111111111
_
```

正则表达式	/正则/	如果在“//”中可以写入字符, 也可以支持正则表达式
-------	------	----------------------------

```
[root@localhost ~]# df -h | awk '/sda[0-9]/ {printf $1 "\t" $5 "\n"} '
#查询包含有 sda 数字的行, 并打印第一字段和第五字段
I
```

补充: awk编程

awk 内置变量	作 用
\$0	代表目前 awk 所读入的整行数据。我们已知 awk 是一行一行读入数据的, \$0 就代表当前读入行的整行数据。
\$n	代表目前读入行的第 n 个字段。
NF	当前行拥有的字段(列)总数。
NR	当前 awk 所处理的行, 是总数据的第几行。
FS	用户定义分隔符。awk 的默认分隔符是任何空格, 如果想要使用其他分隔符(如“:”), 就需要 FS 变量定义。
ARGC	命令行参数个数。
ARGV	命令行参数数组。
FNR	当前文件中的当前记录数(对输入文件起始为 1)。
OFMT	数值的输出格式(默认为%.6g)。
OFS	输出字段的分隔符(默认为空格)。
ORS	输出记录分隔符(默认为换行符)。
RS	输入记录分隔符(默认为换行符)。



CentOS

补充：awk编程——最常用的FS-分隔符的使用

awk默认分隔符为TAB或空格，其他分隔符如何使用FS处理。

例：列出可登录用户的用户名、列出ID号为500的用户名。

```
[root@localhost sh]# cat /etc/passwd | grep "/bin/bash"
root:x:0:0:root:/root:/bin/bash
user1:x:500:500::/home/user1:/bin/bash
user2:x:501:501::/home/user2:/bin/bash
```

```
[root@localhost sh]# cat /etc/passwd | grep "/bin/bash" | awk ' {FS=":"} {print $1}'
root:x:0:0:root:/root:/bin/bash
user1
user2
```

上例中为什么第一行没有处理？

上例中awk是先将第一行给\$0，再识别FS分隔符为:号。
将FS赋新值要在awk执行之前，否则第一行执行不了。

```
[root@localhost sh]# cat /etc/passwd | grep "/bin/bash" | awk 'BEGIN{FS=":"} {print $1}'
root
user1
user2
```

```
[root@localhost sh]# cat /etc/passwd | grep "/bin/bash" | awk 'BEGIN{FS=":"} $3=="500" {print $1}'
user1
```



补充：awk编程——NF-当前数据总列数、NR-当前行号的使用



CentOS

例：列出可登录用户的用户名和序号以及每个用户信息的列数？

```
[root@localhost sh]# cat /etc/passwd | grep "/bin/bash"
root:x:0:0:root:/root:/bin/bash
user1:x:500:500::/home/user1:/bin/bash
user2:x:501:501::/home/user2:/bin/bash
```

```
[root@localhost sh]# cat /etc/passwd | grep "/bin/bash" | awk 'BEGIN{FS=":"} {print $1}'
root
user1
user2
```

过滤出可登录用户，再将每个用户当前序号列出，以及每个用户在/etc/passwd文档的总列数给出。

```
[root@localhost sh]# cat /etc/passwd | grep "/bin/bash" | awk 'BEGIN{FS=":"} {print $1"\t" NR "\t" NF }'
root      1         7
user1     2         7
user2     3         7
```

补充: sed命令-在程序中修改数据

sed 主要是用来将数据进行选取、替换、删除、新增的命令,

```
[root@localhost ~]# 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。
- f 脚本文件名: 从 sed 脚本中读入 sed 操作。和 awk 命令的-f 非常类似。
- r: 在 sed 中支持扩展正则表达式。
- 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: 字符串替换, 用一个字符串替换另外一个字符串。格式为 “行范围 s/旧字符串/新字符串/g” (和 vim 中的替换格式类似)。

对 sed 命令大家要注意, sed 所做的修改并不会直接改变文件的内容 (如果是用管道符接收的命令的输出, 这种情况连文件都没有), 而是把修改结果只显示到屏幕上, 除非使用 “-i” 选项才会直接修改文件。

匹配URL地址，判断URL地址的有效性

- ◆ `$ url="ftp://anonymous:ftp@mirror.lzu.edu.cn/software/scim-1.4.7.tar.gz"`
- ◆ `// 截取服务器类型`
- ◆ `$ echo ${url%%:*}`
- ◆ `ftp`
- ◆ `// 截取域名`
- ◆ `$ tmp=${url##*@} ; echo ${tmp%%/*}`
- ◆ `mirror.lzu.edu.cn`
- ◆ `// 截取路径`
- ◆ `$ tmp=${url##*@} ; echo ${tmp%%/*}`
- ◆ `mirror.lzu.edu.cn/software`
- ◆ `// 截取文件名`
- ◆ `$ echo ${url##*/}`
- ◆ `scim-1.4.7.tar.gz`

Shell 基本语法—字符串操作（选讲）

- ◆ 提取子串（按正则表达式要求从字符串开始的位置提取）
- `expr match "$string" '\($substring\)'`
- `expr "$string" : '\($substring\)'`

从 `$string` 的开始位置提取 `$substring`.

说明：“:”号前后要有空格，`$substring` 是一个正则表达式

```
str=abcABC123ABCabc
```

```
echo `expr "$str" : '\(abc[A-Z]*.2\) '` #结果为abcABC12
```

```
echo `expr "$str" : '\(.[b-c]*[A-Z]..[0-9]\)` #结果为abcABC1
```

```
echo `expr "$str" : '\(.....\) '` # 结果为abcABC1
```

Shell 基本语法—字符串操作（选讲）

- ◆ 从字符串开始的位置匹配子串的长度
 - ✓ **expr match "\$string" '\$substring'**
 - ✓ **expr "\$string" : '\$substring'**

说明：“:”号前后要有空格，\$substring 是一个正则表达式

```
str=abcABC123ABCabc
```

```
echo `expr "$str" : '\(abc[A-Z]*.2\) '` #结果为abcABC12
```

```
echo `expr "$str" : 'abc[A-Z]*.2'` #结果为8
```

```
echo `expr "$str" : '\([b-c]*[A-Z]..[0-9]\) '` #结果为abcABC1
```

```
echo `expr "$str" : '[b-c]*[A-Z]..[0-9]'` #结果为7
```

```
echo `expr "$str" : '\(.....\) '` # 结果为abcABC1
```

```
echo `expr "$str" : '.....'` # 结果为7
```

Shell 基本语法—字符串操作（选讲）

◆ 提取子串（从指定位置开始提取指定长度的子串）

➤ **`${string:position}`**

在 **string** 中从位置 **\$position** 开始提取子串. 如果 **\$string** 为 “*” 或 “@”, 那么将提取从位置 **\$position** 开始的位置参数。

➤ **`${string:position:length}`** 与

`expr substr $string $position $length`

在 **string** 中从位置 **\$position** 开始提取 **\$length** 长度的子串。

```
str=abcABC123ABCabC
```

```
echo ${str:0}      # abcABC123ABCabC
```

```
echo ${str:1}      # bcABC123ABCabC
```

```
echo ${str:7}      # 23ABCabC
```

```
echo ${str:7:3}     # 23A
```

有没有可能从字符结尾开始, 反向提取子串?

使用圆括号或者添加一个空格来转义这个位置参数.

```
echo ${str: (-4)}   # CabC
```

Shell 基本语法—字符串操作（选讲）

◆ 字符串长度

✓ **`${#string}`**

✓ **`expr length $string`**

```
str=1234567890
```

```
echo ${#str}           # 输出结果10
```

```
echo `expr length $str` # 输出结果10
```

```
str=abcABC123ABCabc
```

```
str1=`expr "$str" : '\(abc[A-Z]*.2\) '` #结果为abcABC12
```

```
echo ${#str1}           #结果为8
```

```
str2=`expr "$str" : '\(.[b-c]*[A-Z]..[0-9]\) ` #结果为abcABC1
```

```
echo ${#str2}           #结果为7
```

```
str3=`expr "$str" : '\(.....\) ` # 结果为abcABC1
```

```
echo ${#str3}           #结果为7
```

Shell 基本语法—字符串操作（选讲）

◆ 字符串索引

➤ `expr index $string $substring`

匹配到子串的第一个字符出现的位置 .

例：

```
str=abcABC123ABCabc
```

```
echo `expr index "$str" C12` #C出现最早的位置在第6
```

```
echo `expr index "$str" 1c` #结果为3
```

```
echo `expr index "$str" 2c` #结果为3
```

```
echo `expr index "$str" b2c` #结果为2
```

```
echo `expr index "$str" a45c` #结果为1
```

Shell 基本语法—字符串操作（选讲）

◆ 字符串削除

➤ **`${string#substring}`**

从`$string` 的左边截掉第一个匹配的 `$substring`

➤ **`${string##substring}`**

从`$string` 的左边截掉最后一个匹配的`$substring`

```
str=abcABC123ABCAbc
```

```
#      |----|
```

```
#      |-----|
```

```
echo ${str#a*C}      # 123ABCAbc
```

```
echo ${str##a*C}     # abc
```

➤ **`${string%substring}`**

从`$string` 的右边截掉第一个匹配的 `$substring`

➤ **`${string%%substring}`**

从`$string` 的右边截掉最后一个匹配的`$substring`

```
str=abcABC123ABCabc
```

```
echo ${str%b*c}    # abcABC123ABCa
```

```
# 从$str 的后边开始截掉'b'和'c'之间的最近的匹配
```

```
echo ${str%%b*c}   # a
```

```
# 从$str 的后边开始截掉'b'和'c'之间的最远的匹配
```

Shell 基本语法—字符串操作（选讲）

◆ 子串替换

➤ **`${string/substring/replacement}`**

使用**`$replacement`** 来替换第一个匹配的**`$substring`**.

➤ **`${string//substring/replacement}`**

使用**`$replacement`** 来替换所有匹配的**`$substring`**

```
str=abcABC123ABCAbc
```

```
echo ${str/abc/xyz}  
# xyzABC123ABCAbc
```

```
echo ${str//abc/xyz}  
# xyzABC123ABCxyz
```

Shell 基本语法—字符串操作（选讲）

➤ `${string/#substring/replacement}`

如果`$substring`匹配`$string` 的开头部分,那么就用`$replacement` 来替换`$substring`.

➤ `${string/%substring/replacement}`

如果`$substring`匹配`$string` 的结尾部分,那么就用`$replacement` 来替换`$substring`.

```
str=abcABC123ABCabc
echo ${str/#abc/XYZ}
# XYZABC123ABCabc
echo ${str/%abc/XYZ}
# abcABC123ABCXYZ
```

Shell 基本语法—关于字符串操作小结（选讲）

- ◆ 1, 取得字符串长度
`${#string}`
- ◆ 2, 字符串所在位置
`expr index $string $substring`
- ◆ 3, 从字符串开头到子串的最大长度
`expr "$string" : '$substring'`
- ◆ 4, 字符串截取
见下面详解
- ◆ 5, 字符串匹配并替换
见下面详解

Shell 基本语法—关于字符串操作小结（选讲）

- ◆ 1. # 号截取，删除第一个左边字符，保留右边字符
- ◆ 2. ## 号截取，删除最后一个左边字符，保留右边字符

`${string#substring}` `${string##substring}`

- ◆ 3. %号截取，删除第一个右边字符，保留左边字符
- ◆ 4. %% 号截取，删除最后一个右边字符，保留左边字符

`${string%substring}` `${string%%substring}`

- ◆ 5. 从第几个字符开始截取几个字符
- ◆ 6. 从第几个字符开始截取直到结束

`${string:position}` `${string:position:length}`

Shell 基本语法—关于字符串操作小结（选讲）

◆ 7， 替换第一个匹配字符串

◆ 8， 替换所有匹配的字符串

`${string/substring/replacement}`

`${string//substring/replacement}`

◆ 9， 替换开头部分匹配的字符串

◆ 10， 替换结尾部分匹配的字符串

`${string/#substring/replacement}`

`${string/%substring/replacement}`



Shell 基本语法—关于字符串操作小结（选讲）



- ◆ `$ url="ftp://anonymous:ftp@mirror.lzu.edu.cn/software/scim-1.4.7.tar.gz"`
- ◆ `//` 匹配URL地址，判断URL地址的有效性
- ◆ `//` 截取服务器类型
- ◆ `$ echo ${url%%:*}`
- ◆ `ftp`
- ◆ `//` 截取域名
- ◆ `$ tmp=${url##*@} ; echo ${tmp%/*}`
- ◆ `mirror.lzu.edu.cn`
- ◆ `//` 截取路径
- ◆ `$ tmp=${url##*@} ; echo ${tmp%/*}`
- ◆ `mirror.lzu.edu.cn/software`
- ◆ `//` 截取文件名
- ◆ `$ echo ${url##*/}`
- ◆ `scim-1.4.7.tar.gz`

补充：关于运行进程与子进程的问题

```
[may@CentOS32 ~]$ cat fileps.sh
#!/bin/bash
echo
echo "Current ps number: $$"
echo "Current file name: $0"
echo

a=1
echo "a=$a in file"
```

```
[may@CentOS32 ~]$ bash fileps.sh

Current ps number: 3379
Current file name: fileps.sh

a=1 in file
[may@CentOS32 ~]$ ps
```

PID	TTY	TIME	CMD
3286	pts/0	00:00:00	bash
3384	pts/0	00:00:00	ps

用**bash,sh**运行都
是在新建的子
SHELL进程中运行。

```
[may@CentOS32 ~]$ sh fileps.sh

Current ps number: 3399
Current file name: fileps.sh

a=1 in file
[may@CentOS32 ~]$ ps
```

PID	TTY	TIME	CMD
3286	pts/0	00:00:00	bash
3401	pts/0	00:00:00	ps

补充：关于运行进程与子进程的问题

```
[may@CentOS32 ~]$ cat fileps.sh
#!/bin/bash
echo
echo "Current ps number: $$"
echo "Current file name: $0"
echo

a=1
echo "a=$a in file"
```

```
[may@CentOS32 ~]$ chmod +x fileps.sh
[may@CentOS32 ~]$ PATH=$PATH:.
```

用文件名，绝对路径，相对路径运行都是在新建的子SHELL进程中运行。

```
[may@CentOS32 ~]$ fileps.sh

Current ps number: 3411
Current file name: ./fileps.sh

a=1 in file
[may@CentOS32 ~]$ ps
```

PID	TTY	TIME	CMD
3286	pts/0	00:00:00	bash
3417	pts/0	00:00:00	ps

```
[may@CentOS32 ~]$ ./fileps.sh

Current ps number: 3437
Current file name: ./fileps.sh

a=1 in file
[may@CentOS32 ~]$ ps
```

PID	TTY	TIME	CMD
3286	pts/0	00:00:00	bash
3438	pts/0	00:00:00	ps

```
[may@CentOS32 ~]$ /home/may/fileps.sh

Current ps number: 3469
Current file name: /home/may/fileps.sh

a=1 in file
[may@CentOS32 ~]$ ps
```

PID	TTY	TIME	CMD
3286	pts/0	00:00:00	bash
3470	pts/0	00:00:00	ps

补充：关于运行进程与子进程的问题

```
[may@CentOS32 ~]$ cat fileps.sh
#!/bin/bash
echo
echo "Current ps number: $$"
echo "Current file name: $0"
echo

a=1
echo "a=$a in file"
```

用. 和source运行
都是在当前**SHELL**
进程中运行。

```
[may@CentOS32 ~]$ . fileps.sh

Current ps number: 3286
Current file name: bash

a=1 in file
[may@CentOS32 ~]$ ps
  PID TTY          TIME CMD
 3286 pts/0        00:00:00 bash
 3445 pts/0        00:00:00 ps
```

```
[may@CentOS32 ~]$ source fileps.sh

Current ps number: 3286
Current file name: bash

a=1 in file
[may@CentOS32 ~]$ ps
  PID TTY          TIME CMD
 3286 pts/0        00:00:00 bash
 3451 pts/0        00:00:00 ps
```