Shell 编程基础

目录

1、Linux shell 简介

1.1、Shell 概述

Shell 本身是一个用 C 语言编写的程序，它是用户使用 Unix/Linux 的桥梁，用户的大部分工作都是通过 Shell 完成的。Shell 既是一种命令语言，又是一种程序设计语言：
作为命令语言，它交互式地解释和执行用户输入的命令；
作为程序设计语言，它定义了各种变量和参数，并提供了许多在高级语言中才具有的控制结构，包括循环和分支。

Shell 它虽然不是 Unix/Linux 系统内核的一部分，但它调用了系统核心的大部分功能来执行程序、建立文件并以并行的方式协调各个程序的运行。Shell 是用户与内核进行交互操作的一种接口，目前最流行的 Shell 称为 bash Shell（Bourne Again Shell）

Shell 是一门编程语言(解释型的编程语言)，即 shell 脚本(就是在用 linux 的 shell 命令编程)，Shell 脚本程序从脚本中一行一行读取并执行这些命令，相当于一个用户把脚本中的命令一行一行敲到 Shell 提示符下执行

Shell 是一种脚本语言，那么，就必须有解释器来执行这些脚本
Unix/Linux 上常见的 Shell 脚本解释器有 bash、sh、csh、ksh 等，习惯上把它们称作一种 Shell。我们常说有多少种 Shell，其实说的是 Shell 脚本解释器，可以通过 **cat /etc/shells** 命令查看系统中安装的 shell，不同的 shell 可能支持的命令语法是不相同的

sh 是 Unix 标准默认的 shell，由 Steve Bourne 开发，是 Bourne Shell 的缩写。
bash 是 Linux 标准默认的 shell，本教程也基于 bash 讲解。bash 由 Brian Fox 和 Chet Ramey 共同完成，是 Bourne Again Shell 的缩写。

Shell 本身支持的命令并不多，内部命令一共有 40 个，但是它可以调用其他的程序，每个程序就是一个命令，这使得 Shell 命令的数量可以无限扩展，其结果就是 Shell 的功能非常强大，完全能够胜任 Linux 的日常管理工作，如文本或字符串检索、文件的查找或创建、大规模软件的自动部署、更改系统设置、监控服务器性能、发送报警邮件、抓取网页内容、压缩文件等。

1.2、Shell 基本格式

代码写在普通文本文件中，通常以.sh 结尾，虽然不是强制要求，但希望大家最好这么做

例子：
[root@hadoop02 bin]# **vi helloworld.sh**

#!/bin/bash ## 表示用哪一种 shell 解析器来解析执行我们的这个脚本程序，这句话只对自执行有效，对于使用 sh helloworld.sh 无效

```
echo "hello world"        ## 注释也可以写在这里
```

保存退出即可

在这里，我们就写好了一个 shell 脚本，第一行是固定需要的，表明用哪一种 shell 解析器来执行我们的这个脚本程序。本质上，shell 脚本里面的代码都是就是一些流程控制语句加一些特殊语法再加 shell 命令组成。其中，我们可以当做每一个命令就是 shell 编程当中的关键字。

1.3、Shell 执行方式

1、sh 方式或者 bash 方式

sh helloworld.sh

bash helloworld.sh　　## 直接指定用系统默认的 bash shell 解释执行

2、source 方式或者.方式

source 命令也称为“点命令”，也就是一个点符号（.）,是 bash 的内部命令。

功能：使 Shell 读入指定的 Shell 程序文件并依次执行文件中的所有语句

source 命令通常用于重新执行刚修改的初始化文件，使之立即生效，而不必注销并重新登录。

用法：

source filename 或 . filename

```
[root@hadoop02 bin]# . helloworld.sh
hello world
[root@hadoop02 bin]# source helloworld.sh
hello world
```

注意：.和脚本名称之间是有空格的

3、直接执行该脚本文件

可以有两种方式，不过这两种方式的执行，都需要该文件有执行权限

所以在执行之前，我们要更改他的执行权限

这句话说明"./xxx.sh"或者以绝对路径方式执行，这两种执行方式需要该文件有执行权限,而前面所讲的所有执行方式是不需要该文件有执行权限就可以执行的

1、 切换到该文件所在的路径然后执行命令：

[root@hadoop02 bin]# **./helloworld.sh**

```
[linux@linux ~]$ ll
total 248
drwxr-xr-x. 2 linux linux   4096 Apr 18 12:02 Desktop
drwxr-xr-x. 3 linux linux   4096 Apr 17 22:40 Documents
-rwxr-xr-x. 1 linux linux     32 Apr 21 03:42 hellworld.sh
-rw-rw-r--. 1 linux linux 235373 Apr 18 00:10 hw.txt
-rw-rw-r--. 1 linux linux    214 Apr 18 10:08 test.txt
[linux@linux ~]$ ./hellworld.sh
helloworld
[linux@linux ~]$
```

2、 直接以绝对路径方式执行

[root@hadoop02 bin]# /home/linux/hellworld.sh

```
[linux@linux ~]$ ll
total 248
drwxr-xr-x. 2 linux linux   4096 Apr 18 12:02 Desktop
drwxr-xr-x. 3 linux linux   4096 Apr 17 22:40 Documents
-rwxr-xr-x. 1 linux linux     32 Apr 21 03:42 hellworld.sh
-rw-rw-r--. 1 linux linux 235373 Apr 18 00:10 hw.txt
-rw-rw-r--. 1 linux linux    214 Apr 18 10:08 test.txt
[linux@linux ~]$ ./hellworld.sh
helloworld
[linux@linux ~]$ /home/linux/hellworld.sh
helloworld
[linux@linux ~]$
```

1.4、Shell 注释

单行注释：Shell 脚本中以#开头的代码就是注释

多行注释：Shell 脚本中也可以使用多行注释：

:<<!

echo "dd"　　## 这句话是注释，也就是说在 :<<!　注释语句　!　中间的都是注释

!

2、Shell 基本语法

2.1、变量

2.1.1、系统变量

Linux Shell 中的变量分为“系统变量”和“用户自定义变量”

系统变量可以通过 set 命令查看，用户环境变量可以通过 env 查看：

```
[root@hadoop02 bin]# set
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:expand_aliases:extquote:force_fignore:hostcomplete:interactive
p:promptvars:sourcepath
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
BASH_CMDS=()
BASH_LINENO=()
BASH_SOURCE=()
BASH_VERSINFO=([0]="4" [1]="1" [2]="2" [3]="1" [4]="release" [5]="x86_64-redhat-linux-gnu")
BASH_VERSION='4.1.2(1)-release'
COLORS=/etc/DIR_COLORS
COLUMNS=121
CVS_RSH=ssh
DIRSTACK=()
```

常用系统变量：$PWD　　$SHELL　　$USER　　$HOME

```
[root@hadoop02 bin]# echo $PWD
/root/bin
[root@hadoop02 bin]# echo $SHELL
/bin/bash
[root@hadoop02 bin]# echo $USER
root
[root@hadoop02 bin]# echo $HOME
/root
[root@hadoop02 bin]#
```

那自定义变量呢。？

2.1.2、自定义变量

1、 语法

变量＝值 （例如 STR=abc）

等号两侧不能有空格

变量名称一般习惯为大写

使用变量： $STR

```
[linux@linux bin]$ STR='huangbo'
[linux@linux bin]$ echo $STR
huangbo
[linux@linux bin]$ echo '$STR'
$STR
[linux@linux bin]$ echo "$STR"
huangbo
```

2、 示例

```
[root@hadoop02 bin]# ABC=huang bo
-bash: bo: command not found
[root@hadoop02 bin]# ABC='huang bo'
[root@hadoop02 bin]# echo $ABC
huang bo
[root@hadoop02 bin]# CD='xu zheng'
[root@hadoop02 bin]# echo $CD
xu zheng
[root@hadoop02 bin]# echo 'xu zheng $ABC'
xu zheng $ABC
[root@hadoop02 bin]# echo "xu zheng $ABC"
xu zheng huang bo
[root@hadoop02 bin]#
```

解释：

命令：ABC=huang bo，定义变量时中间带有空格，那么一定要带引号，不然不能定义

命令：ABC='huang bo'，带了单引号则原样输出。表示引号中间的值是整体字符串

命令：ABC="huang bo"，带了双引号，表是字符串中运行出现引用变量和转移字符等

在引号当中要引用变量的时候，单引号和双引号就有区别啦：

命令：echo 'xu zheng $ABC' 和 echo "xu zheng $ABC"

请看区别：

如果是单引号，则引号当中的任何东西都当做字符串，即特殊字符会被脱意

如果是双引号，那么$ABC 能打印出变量的值

单引号和双引号总结：

单引号： 1、单引号里的任何字符都会原样输出，单引号字符串中的变量是无效的 2、单引号字串中不能出现单引号（对单引号使用转义符后也不行）
双引号： 1、双引号里可以有变量 2、双引号里可以出现转义字符

那假如命令是这样的：echo "xu zheng $ABCabc"，请问还能不能打印出变量 ABC 的值呢？请看结果：

```
[root@hadoop02 bin]# echo "xu zheng$ABCabc"
xu zheng
[root@hadoop02 bin]# echo "xu zheng${ABC}abc"
xu zhenghuang boabc
[root@hadoop02 bin]#
```

解决方法是：把变量名用大括号包起来

3、 变量高级用法

撤销变量：unset ABC

声明静态变量：readonly ABC= 'abc'　　特点是这种变量是只读的，不能 unset

请先看一个例子，我现在写两个脚本，在 a.sh 中调用 b.sh 执行，那我们想知道 a 脚本能不能获取到 b 脚本的变量，b 脚本能不能获取到 a 脚本的变量？

```
[root@hadoop02 bin]# vi a.sh
#!/bin/bash
A="A in a.sh"
echo $A
echo $B
/root/bin/b.sh
```

```
[root@hadoop02 bin]# vi b.sh
#!/bin/bash
B="B in a.sh"
echo $A
echo $B
```

执行 a.shz 之后的结果：

```
[root@hadoop02 bin]# ./a.sh
A in a.sh

B in a.sh
[root@hadoop02 bin]#
```

在 a.sh 中只打印出了变量 A 的值，这个好理解，因为 shell 是顺序解析执行的，在打印变量 B 的时候，就算它能获取到 b.sh 当中的变量，它也还没执行，所以，肯定获取不到

那再看 b.sh 打印出来的结果：可以发现虽然 a.sh 当中的语句执行完了再调用 b.sh 来执

行，但是 b.sh 脚本依然也没法获取到 a.sh 的变量

那怎么解决？
使用 export 关键字
export A="A in a.sh"
意味着把变量提升为当前 shell 进程中的全局环境变量，可供其他子 shell 程序使用，A 变量就成了 a.sh 脚本所在 bash 进程的全局变量，该进程的所有子进程都能访问到变量 A

另外一种使用方式：
如果在 a.sh 脚本中用如下方式调用 b.sh：

```
. /root/bin/b.sh      ## 注意：重点关注最前面那个“.”号
```

或者

```
source /root/bin/b.sh
```

用上述两种方式意味着：
b.sh 就在 a.sh 所在的 bash 进程空间中运行

总结：
1、a.sh 中直接调用 b.sh，会让 b.sh 在 A 所在的 bash 进程的“子进程”空间中执行
2、而子进程空间只能访问父进程中用 export 定义的变量
3、一个 shell 进程无法将自己定义的变量提升到父进程空间中去
4、source 或者“.”号执行脚本时，会让脚本在调用者所在的 shell 进程空间中执行

4、 反引号赋值
a=`ls -l /root/bin` ##反引号，运行里面的命令，并把结果返回给变量 a
另外一种写法：
a=$(ls -l /root/bin)

```
[root@hadoop02 bin]# a=`ls -l /root/bin`
[root@hadoop02 bin]# echo $a
total 16 -rwxr-xr-x. 1 root root 64 Dec 30 02:54 a.sh -rwxr-xr-x. 1 root root 42
t 47 Dec 30 03:01 c.sh -rwxr-xr-x. 1 root root 31 Dec 30 01:52 helloworld.sh
[root@hadoop02 bin]# echo -e "$a\n"
total 16
-rwxr-xr-x. 1 root root 64 Dec 30 02:54 a.sh
-rwxr-xr-x. 1 root root 42 Dec 30 02:45 b.sh
-rwxr-xr-x. 1 root root 47 Dec 30 03:01 c.sh
-rwxr-xr-x. 1 root root 31 Dec 30 01:52 helloworld.sh

[root@hadoop02 bin]# echo -e "$a\n" | grep -v '^$'
total 16
-rwxr-xr-x. 1 root root 64 Dec 30 02:54 a.sh
-rwxr-xr-x. 1 root root 42 Dec 30 02:45 b.sh
-rwxr-xr-x. 1 root root 47 Dec 30 03:01 c.sh
-rwxr-xr-x. 1 root root 31 Dec 30 01:52 helloworld.sh
[root@hadoop02 bin]#
```

5、 变量有用技巧

形式	说明
${var}	变量本来的值
${var:-word}	如果变量 var 为空或已被删除(unset)，那么返回 word，但不改变 var 的值
${var:+word}	如果变量 var 被定义，那么返回 word，但不改变 var 的值

${var:=word}	如果变量 var 为空或已被删除(unset)，那么返回 word，并将 var 的值设置为 word
${var:?message}	如果变量 var 为空或已被删除(unset)，那么将消息 message 送到标准错误输出，可以用来检测变量 var 是否可以被正常赋值。 若此替换出现在 Shell 脚本中，那么脚本将停止运行

2.1.3、特殊变量

先来看看各个常用的特殊变量的概念：

$? 表示上一个命令退出的状态码
$$ 表示当前进程编号
$0 表示当前脚本名称
$n 表示 n 位置的输入参数（n 代表数字，n>=1）
$# 表示参数的个数，常用于循环
$*和$@ 都表示参数列表

示例：

```
[root@hadoop02 bin]# vi d.sh
#!/bin/bash
echo "test var"
echo $$
echo $0
echo $1
echo $2
echo $#
echo $*
echo $@
echo $?
```

```
[root@hadoop02 bin]# chmod 755 d.sh
[root@hadoop02 bin]# d.sh a b c
test var
5495
/root/bin/d.sh
a
b
3
a b c
a b c
0
[root@hadoop02 bin]#
```

注：$*与$@区别

$* 和 $@ 都表示传递给函数或脚本的所有参数

不被双引号" "包含时 ==== $* 和 $@ 都以$1 $2 … $n 的形式组成参数列表

当它们被双引号" "包含时 ====
"$*" 会将所有的参数作为一个整体，以"$1 $2 … $n"的形式组成一个整串；
"$@" 会将各个参数分开，以"$1" "$2" … "$n" 的形式组成一个参数列表

区别示例：

```
[root@hadoop02 bin]# vi e.sh
#!/bin/bash
echo $*
echo $@
echo "$*"
echo "$@"
echo --------------------
for a in $*
do echo $a
done
echo --------------------
for b in $@
do echo $b
done
echo --------------------
for a in "$*"
do echo $a
done
echo --------------------
for b in "$@"
do echo $b
done
```

```
[root@hadoop02 bin]# e.sh 1 2 3 a
1 2 3 a
1 2 3 a
1 2 3 a
1 2 3 a
--------------------
1
2
3
a
--------------------
1
2
3
a
--------------------
1 2 3 a
--------------------
1
2
3
a
[root@hadoop02 bin]#
```

2.2、运算符

2.2.1、算数运算符

1、用 expr

格式 expr m + n 注意 **expr 运算符间要有空格**

例如计算（2＋3）×4 的值

1、分步计算

```
S=`expr 2 + 3`
expr $S \* 4          ##    *号需要转义
```

2、一步完成计算

```
expr `expr 2 + 3 ` \* 4
echo `expr \`expr 2 + 3\` \* 4`
```

```
[root@hadoop02 bin]# vi f.sh
#!/bin/bash
S=`expr 2 + 3`
expr $S \* 4
```

结果：

```
[root@hadoop02 bin]# expr `expr 2 + 3` \* 4
20
[root@hadoop02 bin]# echo `expr \`expr 2 + 3\` \* 4`
20
[root@hadoop02 bin]# expr 2 + 3
5
[root@hadoop02 bin]# expr 2+3
2+3
[root@hadoop02 bin]# echo $((2+3))
5
[root@hadoop02 bin]# echo $((2 + 3))
5
[root@hadoop02 bin]#
```

取余：expr 4 % 3

```
[hadoop@hadoop04 data]$ expr 4 % 3
1
```

用 expr 还可以计算字符串的长度，子字符串出现的位置，截取字符串等等

```
[hadoop@hadoop04 data]$ NAME='huangbohuanglei'   ## 定义
[hadoop@hadoop04 data]$ expr length $NAME        ## 求长度
[hadoop@hadoop04 data]$ expr index $NAME huang   ## 求子字符串首次出现的位置
[hadoop@hadoop04 data]$ expr substr $NAME 6 2    ## 从 6 开始，截取 2 两个字符
```

```
[hadoop@hadoop04 data]$ NAME='huangbohuanglei'
[hadoop@hadoop04 data]$ expr length $NAME
15
[hadoop@hadoop04 data]$ expr index $NAME huang
1
[hadoop@hadoop04 data]$ expr substr $NAME 6 2
bo
[hadoop@hadoop04 data]$
```

详情请翻阅：expr --help

2、用(())

```
((1+2))
(((2+3)*4))
count=1
((count++))
((++count))
echo $count

但是要想取到运算结果，需要用$引用
a=$((1+2))
```

3、用$[]

```
SS=$[2+3]
echo $SS
SS=$[2*3]
echo $SS
echo $[(2 + 3)*3]
```

```
[linux@linux bin]$ SS=$[2+3]
[linux@linux bin]$ echo $SS
5
[linux@linux bin]$ SS=$[2*3]
[linux@linux bin]$ echo $SS
6
[linux@linux bin]$ echo $[(2 + 3)*3]
15
[linux@linux bin]$
```

4、 用 let

first=1

second=2

let third=first+second

echo ${third}

```
[hadoop@hadoop02 ~]$ first=1
[hadoop@hadoop02 ~]$ second=2
[hadoop@hadoop02 ~]$ let result=first+second
[hadoop@hadoop02 ~]$ echo $result
3
[hadoop@hadoop02 ~]$ let first++
[hadoop@hadoop02 ~]$ echo $first
2
[hadoop@hadoop02 ~]$ let first--
[hadoop@hadoop02 ~]$ echo $first
1
[hadoop@hadoop02 ~]$ let first+=10
[hadoop@hadoop02 ~]$ echo $first
11
```

5、 注意：以上命令都只对整形数值有效，不适用于浮点数

```
[hadoop@hadoop02 ~]$ first=1
[hadoop@hadoop02 ~]$ second=2.3
[hadoop@hadoop02 ~]$ result=$[first+second]
-bash: 2.3: syntax error: invalid arithmetic operator (error token is ".3")
[hadoop@hadoop02 ~]$ result=$((first+second))
-bash: 2.3: syntax error: invalid arithmetic operator (error token is ".3")
[hadoop@hadoop02 ~]$ echo `expr $first + $second`
expr: non-numeric argument

[hadoop@hadoop02 ~]$ let result=$first+$second
-bash: let: result=1+2.3: syntax error: invalid arithmetic operator (error token is ".3")
[hadoop@hadoop02 ~]$
```

如果有浮点数参与运算，可以将 echo 与 bc 命令结合起来使用，代码如下：

```
[hadoop@hadoop02 ~]$ echo "$first+$second"
1+2.3
[hadoop@hadoop02 ~]$ echo "$first+$second" | bc
3.3
[hadoop@hadoop02 ~]$ echo "$first*$second" | bc
2.3
[hadoop@hadoop02 ~]$
```

且看 bc 的一些强大作用：

```
echo "1.212*3" | bc              ##  简单浮点运算
echo "scale=2;3/8" | bc          ##将输出结果设置为 2 位
echo "obase=2;127" | bc          ##输出运算结果的二进制
echo "obase=10;ibase=2;101111111" | bc       ##将二进制转换成十进制
echo "10^10" | bc                ##求幂指数
echo "sqrt(100)" | bc            ##开平方
```

除了用 bc 做尽职转换以外，还可以这样做：

echo $((base#number))　表示把任意 base 进制的数 number 转换成十进制

例子：

echo $((8#377))　　返回 255

echo $((025))　　返回 21 ，　八进制

echo $((0xA4))　　返回 164 ，　十六进制

使用 bc 还可以用来比较浮点数的大小：

```
[root@hadoop02 bin]# echo "1.2 < 2" |bc
1
[root@hadoop02 bin]# echo "1.2 > 2" |bc
0
[root@hadoop02 bin]# echo "1.2 == 2.2" |bc
0
[root@hadoop02 bin]# echo "1.2 != 2.2" |bc
1
```

看出规律了嘛？运算如果为真返回 1，否则返回 0，写一个例子：

```
[root@hadoop02 bin]# [ $(echo "2.2 > 2" |bc) -eq 1 ] && echo yes || echo no
yes
[root@hadoop02 bin]# [ $(echo "2.2 < 2" |bc) -eq 1 ] && echo yes || echo no
no
```

2.2.2、关系运算符

下面给出一张关系运算符的列表：

运算符	等同运算符	说明
-eq	=	检测两个数是否相等，相等返回 true
-ne	!=	检测两个数是否相等，不相等返回 true
-ge	>=	检测左边的数是否大等于右边的，如果是，则返回 true
-gt	>	检测左边的数是否大于右边的，如果是，则返回 true
-le	<=	检测左边的数是否小于等于右边的，如果是，则返回 true
-lt	<	检测左边的数是否小于右边的，如果是，则返回 true

2.2.3、布尔运算符

运算符	等同运算符	说明
!	!	非运算，表达式为 true 则返回 false，否则返回 true
-a	&&	与运算，两个表达式都为 true 才返回 true
-o	||	或运算，有一个表达式为 true 则返回 true

2.2.4、字符串运算符

运算符	说明
=	检测两个字符串是否相等，相等返回 true
!=	检测两个字符串是否相等，不相等返回 true
-z	检测字符串长度是否为 0，为 0 返回 true
-n	检测字符串长度是否为 0，不为 0 返回 true
str	检测字符串是否为空，不为空返回 true

2.2.5、文件运算符

运算符	说明
-d	检测文件是否是目录，如果是，则返回 true
-f	检测文件是否是普通文件（既不是目录，也不是设备文件），如果是，则返回 true
-e	检测文件（包括目录）是否存在，如果是，则返回 true
-s	检测文件是否为空（文件大小是否大于 0），不为空返回 true
-r	检测文件是否可读，如果是，则返回 true
-w	检测文件是否可写，如果是，则返回 true
-x	检测文件是否可执行，如果是，则返回 true
-b	检测文件是否是块设备文件，如果是，则返回 true
-c	检测文件是否是字符设备文件，如果是，则返回 true

2.3、流程控制

2.3.1、if

语法格式：

```
if condition
        then
                statements
[elif condition
                then statements. ..]
[else
                statements ]
fi
```

示例程序：

```
[root@hadoop02 bin]# vi g.sh
#!/bin/bash
## read a value for NAME from stdin
read -p "please input your name:" NAME
## printf '%s\n' $NAME
if [ $NAME = root ]
        then
                echo "hello ${NAME},    welcome !"
elif [ $NAME = hadoop ]
        then
                echo "hello ${NAME},    welcome !"
else
```

```
[root@hadoop02 bin]# chmod 755 g.sh
[root@hadoop02 bin]# g.sh
please input your name:root
hello root,  welcome !
[root@hadoop02 bin]# g.sh
please input your name:hadoop
hello hadoop,  welcome !
[root@hadoop02 bin]# g.sh
please input your name:spark
not any one, get out here !
[root@hadoop02 bin]#
```

echo "I don't know you !" **fi**	

规则解释：

[condition] (注：condition 前后要有空格)

#非空返回 true，可使用$?验证（0 为 true，>1 为 false）

[hadoop]

#空返回 false

[ ]

```
[root@hadoop02 bin]# a=2
[root@hadoop02 bin]# b=2
[root@hadoop02 bin]# [ a = b ]
[root@hadoop02 bin]# echo $?
1
[root@hadoop02 bin]# [ $a = $b ]
[root@hadoop02 bin]# echo $?
0
[root@hadoop02 bin]# [ ]
[root@hadoop02 bin]# echo $?
1
[root@hadoop02 bin]# [ $a ]
[root@hadoop02 bin]# echo $?
0
[root@hadoop02 bin]# if [ a = b ];then echo ok;else echo notok;fi
notok
[root@hadoop02 bin]# if [ $a = $b ];then echo ok;else echo notok;fi
ok
[root@hadoop02 bin]# if [ a=b ];then echo ok;else echo notok;fi
ok
[root@hadoop02 bin]# if [a=b];then echo ok;else echo notok;fi
-bash: [a=b]: command not found
notok
[root@hadoop02 bin]#
```

注意[ ]内部的=周边的空格

短路运算符（理解为三元运算符）

[condition] && echo OK || echo notok

条件满足，执行&&后面的语句；条件不满足，执行||后面的语句

```
[root@hadoop02 bin]# [ a = b ] && echo "OK" || echo "not OK"
not OK
[root@hadoop02 bin]# [ $a = $b ] && echo "OK" || echo "not OK"
OK
[root@hadoop02 bin]#
```

条件判断组合

条件判断组合有两种使用方式：

[] 和 [[]]

注意它们的区别：

[[ ]] 中逻辑组合可以使用 && || 符号

[] 里面逻辑组合可以用 -a -o

```
[root@hadoop02 bin]# if [ a = b -a b = b ]; then echo ok;else echo notok;fi
notok
[root@hadoop02 bin]# if [ a = b -o b = b ]; then echo ok;else echo notok;fi
ok
[root@hadoop02 bin]#
[root@hadoop02 bin]# if [[ a = b && b = b ]]; then echo ok;else echo notok;fi
notok
[root@hadoop02 bin]# if [[ a = b || b = b ]]; then echo ok;else echo notok;fi
ok
[root@hadoop02 bin]#
```

常用判断运算符：

1、字符串比较

= 判断相等

!= 判断不相等

-z 字符串长度是为 0 返回 true

-n 字符串长度是不为 0 返回 true

```
[root@hadoop02 bin]# if [ 'aa' = 'bb' ]; then echo ok; else echo notok;fi
notok
[root@hadoop02 bin]# if [ 'aa' != 'bb' ]; then echo ok; else echo notok;fi
ok
[root@hadoop02 bin]# if [ -n "aa" ]; then echo ok; else echo notok;fi
ok
[root@hadoop02 bin]# if [ -z "aa" ]; then echo ok; else echo notok;fi
notok
[root@hadoop02 bin]# if [ -z "" ]; then echo ok; else echo notok;fi
ok
[root@hadoop02 bin]# if [ -n "" ]; then echo ok; else echo notok;fi
notok
[root@hadoop02 bin]# if [ -n " " ]; then echo ok; else echo notok;fi
ok
[root@hadoop02 bin]# if [ -z " " ]; then echo ok; else echo notok;fi
notok
[root@hadoop02 bin]#
```

2、整数比较

-lt 小于 less than

-le 小于等于

-eq 等于

-gt 大于 great than

-ge 大于等于

-ne 不等于

```
[root@hadoop02 bin]# if [ 2 -lt 3 ]; then echo ok; else echo notok;fi
ok
```

3、文件判断

-d 是否为目录

if [-d /bin]; then echo ok; else echo notok;fi

-f 是否为文件

if [-f /bin/ls]; then echo ok; else echo notok;fi

-e 是否存在

if [-e /bin/ls]; then echo ok; else echo notok;fi

2.3.2、while

语法格式：

```
while expression
do
    command
    ……
done
```

```
i=1
while ((i<=3))
do
   echo $i
   let i++
done
```

```
#!/bin/bash
i=1
while [ $i -le 3 ]
do
   echo $i
   let i++
done
```

命令执行完毕，控制返回循环顶部，从头开始直至测试条件为假
换种方式：循环体会一直执行，直到条件表达式 expression 为 false
注意：上述 let i++ 可以写成 i=$(($i+1))或者 i=$((i+1))

2.3.3、case

Case 语法（通过下面这个例子展示）：

```
case $1 in
start)
      echo "starting"
      ;;
stop)
      echo "stoping"
      ;;
*)
      echo "Usage: {start|stop}"
esac
```

2.3.4、for

语法格式：

```
for 变量 in 列表
do
      command
      ……
done
```

列表是一组值（数字、字符串等）组成的序列，每个值通过空格分隔。每循环一次，就将列表中的下一个值赋给变量

三种方式
方式一：

```
for N in 1 2 3; do echo $N; done
```

```
[root@hadoop02 bin]# for N in 1 2 3; do echo $N; done
1
2
3
[root@hadoop02 bin]#
```

方式二：

for N in {1..3}; do echo $N; done

```
[root@hadoop02 bin]# for N in {1..3}; do echo $N; done
1
2
3
[root@hadoop02 bin]#
```

方式三：

for ((i=0; i<=2; i++)); do echo "welcome $i times"; done

```
[root@hadoop02 bin]# for ((i = 0; i <= 2; i++)); do echo "welcome $i times"; done
welcome 0 times
welcome 1 times
welcome 2 times
[root@hadoop02 bin]# for ((i=0; i<=2; i++)); do echo "welcome $i times"; done
welcome 0 times
welcome 1 times
welcome 2 times
[root@hadoop02 bin]#
```

2.3.5、util

语法结构：

```
until expression
do
    command
    ……
done
```

expression 一般为条件表达式，如果返回值为 false，则继续执行循环体内的语句，否则跳出循环。

换种方式说：循环体会一直执行，直到条件表达式 expression 为 true

示例：

```
#!/bin/bash
## vi util.sh
a=0
until [ ! $a -lt 3 ]
do
    echo $a
    a=`expr $a + 1`
done
```

```
[hadoop@hadoop04 myshell]$ sh util.sh
0
1
2
```

2.3.6、跳出循环

两个命令：
break 命令允许跳出所有循环（终止执行后面的所有循环）
continue 命令与 break 命令类似，只有一点差别，它不会跳出所有循环，仅仅跳出当前循环

2.4、数组

在 Shell 中，用括号来表示数组，数组元素用“空格”符号分割开。定义数组的一般形式为：
array_name=(value1 ... valuen)
例子：**mingxing=(huangbo xuzheng wangbaoqiang)**
也可以单独定义：**mingxing[3]=liujialing**
读取数组元素的格式是：**${array_name[index]}**

```
[linux@linux ~]$ mingxing=(huangbo xuzheng wangbaoqiang)
[linux@linux ~]$ echo $mingxing
huangbo
[linux@linux ~]$ echo $mingxing[2]
huangbo[2]
[linux@linux ~]$ echo ${mingxing[2]}
wangbaoqiang
[linux@linux ~]$ mingxing[3]=liujialing
[linux@linux ~]$ echo ${mingxing[3]}
liujialing
[linux@linux ~]$
```

获取数组下标：
[linux@linux ~]$ **echo ${!mingxing[@]}**

```
[linux@linux ~]$ echo ${!mingxing[@]}
0 1 2 3
[linux@linux ~]$ echo ${!mingxing[*]}
0 1 2 3
[linux@linux ~]$
```

输出数组的所有元素：
[linux@linux ~]$ **echo ${mingxing[*]}**
[linux@linux ~]$ **echo ${mingxing[@]}**

```
[linux@linux ~]$ echo ${mingxing[*]}
huangbo xuzheng wangbaoqiang liujialing
[linux@linux ~]$ echo ${mingxing[@]}
huangbo xuzheng wangbaoqiang liujialing
[linux@linux ~]$
```

获取数组的长度：

[linux@linux ~]$ **echo ${#mingxing[*]}**

[linux@linux ~]$ **echo ${#mingxing[@]}**

```
[linux@linux ~]$ echo ${#mingxing[*]}
4
[linux@linux ~]$ echo ${#mingxing[@]}
4
[linux@linux ~]$
```

数组对接：

[linux@linux ~]$ **mingxing+=(liuyifei liuyufeng)**

```
[linux@linux ~]$ mingxing+=(liuyifei liuyufeng)
[linux@linux ~]$ echo ${mingxing[*]}
huangbo xuzheng wangbaoqiang liujialing liuyifei liuyufeng
[linux@linux ~]$
```

删除数组元素，但是会保留数组对应位置，就是该值的下标依然会保留，会空着，之后，还可以填充其他的值进来。

删除第一个元素：[linux@linux ~]$ **unset mingxing[0]**

```
[linux@linux ~]$ unset mingxing[0]
[linux@linux ~]$ echo ${mingxing[*]}
xuzheng wangbaoqiang liujialing liuyifei liuyufeng
[linux@linux ~]$ echo ${!mingxing[*]}
1 2 3 4 5
[linux@linux ~]$ echo ${mingxing[0]}

[linux@linux ~]$ echo ${mingxing[1]}
xuzheng
[linux@linux ~]$
```

遍历数组：

```
#!/bin/bash
IP=(192.168.1.1 192.168.1.2 192.168.1.3)
# 第一种方式
for ((i=0;i<${#IP[*]};i++))
do
    echo ${IP[$i]}
done
#第二种方式
for ip in ${IP[*]}
do
    echo $ip
done
```

效果：

```
[linux@linux bin]$ chmod 755 array.sh
[linux@linux bin]$ array.sh
192.168.1.1
192.168.1.2
192.168.1.3
```

数组的分片：

${arr[@]:number1:number2}

这里 number1 从下标 number1 开始取值，number2 往后取几个元素，即取到的新的数组的长度

```
[hadoop@hadoop08 ~]$ arr=(1 2 3 4 5 6 7 8 9)
[hadoop@hadoop08 ~]$ echo "{arr[@]:0:3} ---  ${arr[@]:0:3}"
{arr[@]:0:3} ---  1 2 3
[hadoop@hadoop08 ~]$ echo "{arr[@]:3:3} ---  ${arr[@]:3:3}"
{arr[@]:3:3} ---  4 5 6
[hadoop@hadoop08 ~]$ echo "{arr[@]:4:4} ---  ${arr[@]:4:4}"
{arr[@]:4:4} ---  5 6 7 8
```

2.5、函数使用

函数的语法使用示例：

```
[root@hadoop02 bin]# vi i.sh
#!/bin/sh
hello(){
        echo "`date +%Y-%m-%d`"
        # return 2
}
hello
echo “huangbo”
# echo $?
A="mazhonghua"
echo $A
```

```
[root@hadoop02 bin]# i.sh
2016-12-08
mazhonghua
[root@hadoop02 bin]#
```

函数的调用方式就是直接写函数名就 OK 了

注意：

1、必须在调用函数地方之前，先声明函数，shell 脚本是逐行运行。不会像其它语言一样先预编译

2、函数返回值，只能通过$? 系统变量获得，可以显示加：return 返回，如果不加，将以最后一条命令运行结果，作为返回值。 return 后跟数值 n(0-255)

脚本调试：

使用-x 选项跟踪脚本调试 shell 脚本，能打印出所执行的每一行命令以及当前状态

sh -x i.sh

或者在代码中加入：set -x

```
[root@hadoop02 bin]# sh -x i.sh
+ hello
++ date +%Y-%m-%d
+ echo 2016-12-08
2016-12-08
+ A=mazhonghua
+ echo mazhonghua
mazhonghua
```

2.6、函数参数

```
[root@hadoop02 bin]# vi funcWithParam.sh
#!/bin/bash
# filename=funcWithParam
funcWithParam(){
        echo "第一个参数为 $1 !"
        echo "第二个参数为 $2 !"
        echo "第十个参数为 $10 !"
        echo "第十个参数为 ${10} !"
        echo "第十一个参数为 ${11} !"
        echo "参数总数有 $# 个!"
        echo "作为一个字符串输出所有参数 $* !"
}
funcWithParam 1 2 3 4 5 6 7 8 9 34 73
```

```
[root@hadoop02 bin]# funcWithParam.sh
第一个参数为 1 !
第二个参数为 2 !
第十个参数为 10 !
第十个参数为 34 !
第十一个参数为 73 !
参数总数有 11 个!
作为一个字符串输出所有参数 1 2 3 4 5 6 7 8 9 34 73 !
[root@hadoop02 bin]#
```

2.7、跨脚本调用函数

编写一个 base.sh 脚本，里面放放一个 test 函数

```
#!/bin/bash
test(){
        echo "hello"
}
```

再编写一个 other.sh 脚本，里面引入 base.sh 脚本，并且调用 test 函数：

```
#!/bin/bash
. /home/linux/bin/base.sh   ## 引入脚本
test      ##调用引入脚本当中的 test 函数
```

效果：

```
[linux@linux bin]$ other.sh
hello
[linux@linux bin]$
```

3、Shell 综合案例

3.1、打印 9*9 乘法表

示例代码：

```
#!/bin/bash

for((i=1;i<=9;++i))
do
	for((j=1;j<=i;j++))
	do
		echo -ne "$i*$j=$((i*j))\t"
	done
	echo
done
```

解释

-n 不加换行符

-e 解释转义符

echo 换行

效果图

```
[root@hadoop02 bin]# 99.sh
1*1=1
2*1=2   2*2=4
3*1=3   3*2=6   3*3=9
4*1=4   4*2=8   4*3=12  4*4=16
5*1=5   5*2=10  5*3=15  5*4=20  5*5=25
6*1=6   6*2=12  6*3=18  6*4=24  6*5=30  6*6=36
7*1=7   7*2=14  7*3=21  7*4=28  7*5=35  7*6=42  7*7=49
8*1=8   8*2=16  8*3=24  8*4=32  8*5=40  8*6=48  8*7=56  8*8=64
9*1=9   9*2=18  9*3=27  9*4=36  9*5=45  9*6=54  9*7=63  9*8=72  9*9=81
[root@hadoop02 bin]#
```

3.2、自动部署集群的 JDK

1、 需求描述

公司内有一个 N 个节点的集群，需要统一安装一些软件（jdk）

需要开发一个脚本，实现对集群中的 N 台节点批量自动下载、安装 jdk

2、 思路

思考一下：我们现在有一个 JDK 安装包在一台服务器上。那我们要实现这个目标：

1、 把包传到每台服务器，或者通过本地 yum 源的方式去服务器取
2、 给每台一台机器发送一个安装脚本，并且让脚本自己执行
3、 要写一个启动脚本，用来执行以上两部操作

3、 Expect 的使用

蛋疼点：假如在没有配置 SSH 免密登录的前提下，我们要要是 scp 命令从一台机器拷贝文件夹到另外的机器，会有人机交互过程，那我们怎么让机器自己实现人机交互？

灵丹妙药：expect

命令	描述
set	可以设置超时，也可以设置变量
timeout	超时等待时间，默认 10s
spawn	执行一个命令
expect ""	匹配输出的内容
exp_continue	继续执行下面匹配

思路：模拟该人机交互过程，在需要交互的情况下，通过我们的检测给输入提前准备好的值即可

示例：观看配置 SSH 免密登录的过程

实现脚本：

[root@hadoop02 bin]# vi testExpect.sh

```
#!/bin/bash

## 定义一个函数
sshcopyid(){
        expect -c "
                spawn ssh-copy-id $1
                expect {
                        \"(yes/no)?\" {send \"yes\r\";exp_continue}
                        \"password:\" {send \"$2\r\";exp_continue}
                }
        "
}

## 调用函数执行
sshcopyid $1 $2
```

注意：如果机器没有 expect，则请先安装 expect

yum install -y expect

4、 脚本实现

见代码

1、 启动脚本 initInstallJDK.sh

```
#!/bin/bash

SERVERS="192.168.123.201"
PASSWORD=hadoop
BASE_SERVER=192.168.123.202

auto_ssh_copy_id() {
	expect -c "set timeout -1;
		spawn ssh-copy-id $1;
		expect {
			*(yes/no)* {send -- yes\r;exp_continue;}
			*password:* {send -- $2\r;exp_continue;}
			eof {exit 0;}
		}";
}

ssh_copy_id_to_all() {
	for SERVER in $SERVERS
	do
		auto_ssh_copy_id $SERVER $PASSWORD
	done
}

ssh_copy_id_to_all

for SERVER in $SERVERS
do
	scp installJDK.sh root@$SERVER:/root
	ssh root@$SERVER chmod 755 installJDK.sh
	ssh root@$SERVER /root/installJDK.sh
done
```

2、 安装脚本 installJDK.sh

```
#!/bin/bash

BASE_SERVER=192.168.123.202
yum install -y wget
wget $BASE_SERVER/soft/jdk-8u73-linux-x64.tar.gz
tar -zxvf jdk-8u73-linux-x64.tar.gz -C /usr/local

cat >> /etc/profile << EOF
export JAVA_HOME=/usr/local/jdk1.8.0_73
export PATH=\$PATH:\$JAVA_HOME/bin
EOF
```

4、总结

写脚本注意事项：

1、开头加解释器： #!/bin/bash，和注释说明。

2、命名建议规则：变量名大写、局部变量小写，函数名小写，名字体现出实际作用。

3、默认变量是全局的，在函数中变量 local 指定为局部变量，避免污染其他作用域。

4、set -e 遇到执行非 0 时退出脚本， set -x 打印执行过程。

5、写脚本一定先测试再到生产上。