

## 第三节 shell 语法

### 1. 概论

shell 是我们通过命令行与操作系统沟通的语言。

shell 脚本可以直接在命令行中执行，也可以将一套逻辑组织成一个文件，方便复用。

AC Terminal 中的命令行可以看成是一个 shell 脚本在逐行执行。

Linux 中常见的 shell 脚本有很多种，常见的有：

- Bourne shell(/usr/bin/sh 或 /bin/sh)
- Bourne Again Shell(bin/bash)
- C Shell(/usr/bin/csh)
- K Shell( /usr/bin/ksh)
- zsh
- ...

Linux 系统中一般默认使用 bash，所以接下来讲解 bash 中的语法

文件开头需要写 `#!/bin/bash`，指明 bash 为脚本解释器

### 脚本示例

新建一个 `test.sh` 文件，内容如下：

```
1  #! /bin/bash
2  echo "Hello world!"
```

### 运行方式

作为可执行文件

```
1  chmod +x test.sh #使脚本具有可执行权限
2  ./test.sh #当前路径下执行
3
4  /home/acs/test.sh # 绝对路径下执行
5
6  ~/test.sh #家目录路径下执行
7
8  #用解释器执行
9  bash test.sh
10
11 #1. 为test文件加执行权限，
12 acs@740d1bdabef2:~$ chmod +x test.sh      acs@740d1bdabef2:~$ ls
      homework  main.cpp  test.sh    #文件列表，此时test.sh 会改变颜色
13 #2. 直接执行当前目录的test.sh
14 acs@740d1bdabef2:~$ ./test.sh
15 Hello world! #脚本输出
16
17 #3. 绝对路径下执行文件
18 acs@740d1bdabef2:~$ /home/acs/test.sh
```

```
19 Hello world!      #脚本输出
20
21 #4.家目录下执行
22 acs@740d1bdabef2:~$ ~/test.sh
23 Hello world!      #脚本输出
24
```

```
Hello World!
acs@740d1bdabef2:~$ chmod +x test.sh
acs@740d1bdabef2:~$ ls
homework  main.cpp  test.sh
acs@740d1bdabef2:~$ ./test.sh
Hello World!
acs@740d1bdabef2:~$ /home/acs/test.sh
Hello World!
acs@740d1bdabef2:~$ ~/test.sh
Hello World!
acs@740d1bdabef2:~$ |
```

## 2.注释

### 单行注释

每行中 # 之后的内容为注释内容

```
1 # 这是一行注释
2
3 echo "Hello world" # 这也是注释
4
```

### 多行注释

格式:

```
1 :<<EOF
2 第一行注释
3 第二行注释
4 第三行注释
5 EOF
6
```

其中, EOF 可以换成其它任意字符。例如:

```
1 :<<ZS
2 注释改为ZS
3 第二行注释
4 第三行注释
5 ZS
6
```

### 3.变量

#### 定义变量

定义变量，不需要加 `$` 符号，例如：

```
1 name1 = 'jtx' #单引号定义字符串
2 name2 = "jtx" #双引号定义字符串
3 name3 = jtx #也可以不加引号，同样表示字符串
```

#### 使用变量

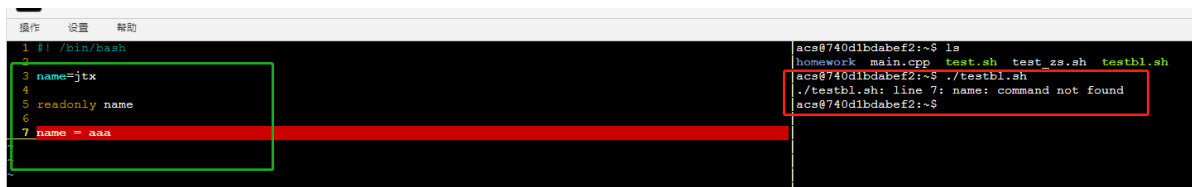
使用变量，需要加上 `$` 符号，或者 `${}` 符号。花括号是可选的，主要为了帮助解释器识别变量边界。

```
1 name=jtx
2 echo $name #输出jtx
3 echo ${name} #输出jtx
4 echo ${name}string #输出jtxstring
5
```

#### 只读变量

使用 `readonly` 或者 `declare` 可以将变量变为只读。

```
1 name=jtx
2 readonly name
3 declare -r name #两种写法都可以
4
5 name=aaa # 会报错，因为此时name只读
```



A terminal window showing a script execution. The script sets `name=jtx`, marks it as `readonly`, and then attempts to assign `name=aaa`. This last line is highlighted in red, and the terminal output shows an error: `./testbl.sh: line 7: name: command not found`.



A terminal window showing a script execution. The script sets `name=jtx`, uses `declare -r name` to mark it as read-only, and then attempts to assign `name=aaa`. This last line is highlighted in red, and the terminal output shows an error: `./testbl.sh: line 7: name: command not found`.

#### 删除变量

`unset` 可以删除变量。

```
1 name=jtx
2 unset name
3 echo $name #输出空行，因为name变量已经被删除
```



A terminal window showing a script execution. The script sets `name=jtx`, uses `unset name` to delete the variable, and then echoes `$name`. The output is an empty line, and the `unset name` line is highlighted in red.

## 变量类型

- 自定义变量(局部变量)
  - 子进程不能访问的变量
- 环境变量(全局变量)
  - 子进程可以访问的变量

### 1. 自定义变量改成环境变量:

```
1  #注意: 一个bash相当于一个子进程
2
3  acs@740d1bdabef2:~$ name=jtx  #定义name变量
4  acs@740d1bdabef2:~$ echo $name  #在当前bash下输出name的值
5  jtx                             #输出结果
6  acs@740d1bdabef2:~$ bash      #新建一个bash
7  acs@740d1bdabef2:~$ echo $name  #在新建的bash下输出name的值
8                                   #由于此时变量为局部变量, 故此时输出为空
9  acs@740d1bdabef2:~$ exit      #退出新建的bash
10 exit
11 acs@740d1bdabef2:~$ export name  #将name修改为环境变量
12 acs@740d1bdabef2:~$ bash        #新建一个bash
13 acs@740d1bdabef2:~$ echo $name  #在新建的bash中输出name的值
14 jtx                             #此时 name为环境变量, 故新的bash可以访问到
15 acs@740d1bdabef2:~$ exit
16 exit
```

### 2. 环境变量改为自定义变量:

```
1  #通过此例, 可以看到 declare 的用法
2  acs@740d1bdabef2:~$ export name=jtx  #定义全局变量
3  acs@740d1bdabef2:~$ bash            #新建一个子进程
4  acs@740d1bdabef2:~$ echo $name      #子进程可以访问到name
5  jtx
6  acs@740d1bdabef2:~$ exit            #退出当前子进程
7  exit
8  acs@740d1bdabef2:~$ declare +x name  #通过declare将name变为局部变量
9
9  acs@740d1bdabef2:~$ bash
10 acs@740d1bdabef2:~$ echo $name      #子进程无法访问到name
11
12 acs@740d1bdabef2:~$ exit
13 exit
14 acs@740d1bdabef2:~$ declare -x name  #通过declare 将name变为全局变量
15 acs@740d1bdabef2:~$ bash
16 acs@740d1bdabef2:~$ echo $name      #子进程又可以访问到name
17
17 jtx
18 acs@740d1bdabef2:~$ exit
19 exit
```

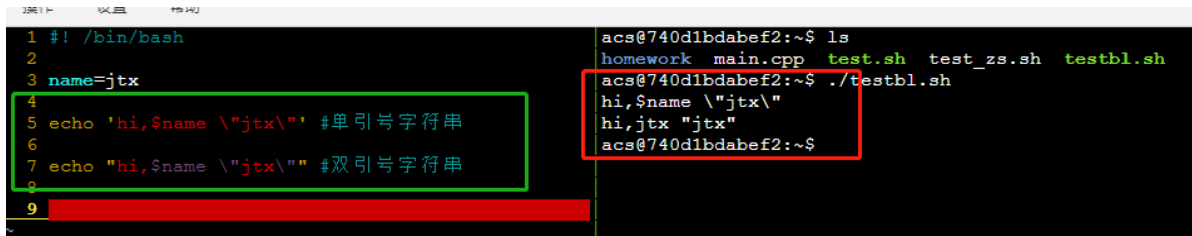
## 字符串

字符串可以使用单引号，也可以使用双引号，也可以不用引号。

单引号与双引号的区别：

- 单引号中的内容会原样输出，不会执行，不会获取变量的值
- 双引号中的内容可以执行、可以获取变量的值

```
1 name=jtx
2 echo 'hi, $name \'jtx\'' #单引号字符串，输出 hi,$name \'jtx\'
3 echo "hi, $name \'jtx\'" #双引号字符串，输出 hi,jtx "jtx"
```



```
1 #!/bin/bash
2
3 name=jtx
4
5 echo 'hi,$name \'jtx\'' #单引号字符串
6
7 echo "hi,$name \'jtx\'" #双引号字符串
8
9
```

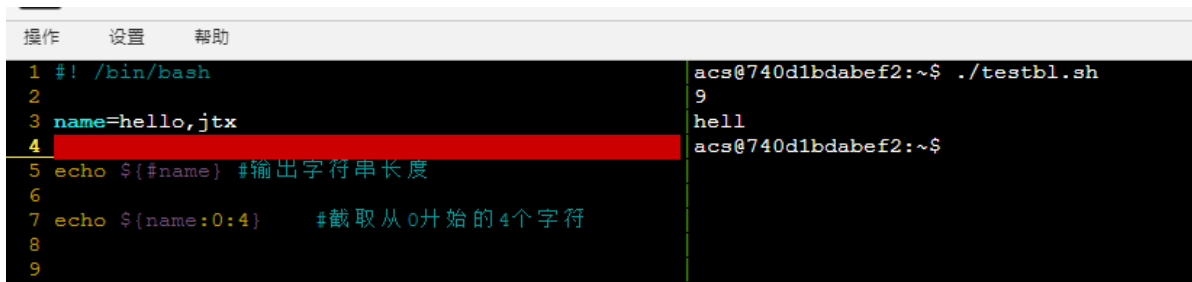
```
acs@740d1bdabef2:~$ ls
homework main.cpp test.sh test_zs.sh testbl.sh
acs@740d1bdabef2:~$ ./testbl.sh
hi,$name \'jtx\'
hi,jtx "jtx"
acs@740d1bdabef2:~$
```

获取字符串长度

```
1 name=jtx
2 echo ${#name} #输出3
```

截取子串

```
1 name="hello,jtx"
2 echo ${name:0:4} #截取从0开始的4个字符
```



```
1 #!/bin/bash
2
3 name=hello,jtx
4
5 echo ${#name} #输出字符串长度
6
7 echo ${name:0:4} #截取从0开始的4个字符
8
9
```

```
acs@740d1bdabef2:~$ ./testbl.sh
9
hell
acs@740d1bdabef2:~$
```

## 4.默认变量

### 文件参数变量

在执行 shell 脚本时，可以向脚本传递参数。

\$0 是文件名(包含路径)。例如：

```
1 | #!/bin/bash
2 |
3 | echo "文件名:"$0
4 | echo "第一个参数: "$1
5 | echo "第二个参数: "$2
6 | echo "第三个参数: "$3
7 | echo "第四个参数: "$4
8 |
```

然后执行该脚本：

```
1 | acs@740d1bdabef2:~$ ./test4.sh          #没有参数的情况下， $1,$2,$3,$4均为空
2 |
3 | ./test4.sh
4 |
5 |
6 |
7 | acs@740d1bdabef2:~$ ./test4.sh 1 2 3 4
8 | ./test4.sh
9 | 第二个参数: 1
10 | 第一个参数: 2
11 | 第一个参数: 3
12 | 第一个参数: 4
```

其它参数相关变量

参数	说明
<code>\$#</code>	代表w文件传入的参数个数，如上例中值为4
<code>\$*</code>	由所有参数构成的用空格隔开的字符串，如上例中值 <code>"\$1" "\$2" "\$3" "\$4"</code>
<code>@</code>	每个参数分别用双引号括起来的字符串，如上例中值为 <code>"\$1" "\$2" "\$3" "\$4"</code>
<code>\$\$</code>	脚本当前运行的进程ID
<code>\$?</code>	上一条命令的退出状态(注意不是 <code>stdout</code> ，而是 <code>exit code</code> )。0表示正常退出，其他值表示错误
<code>\$(command)</code>	返回 <code>command</code> 这条命令的 <code>stdout</code> （可嵌套）
<code>command</code>	返回 <code>command</code> 这条命令的 <code>stdout</code> (不可嵌套)

5.数组

数组中可以存放多个不同类型的值，只支持一维数组，初始化不需要指明数组大小。

数组下标从0开始。

定义

数组用小括号表示，元素之间用空格隔开。例如：

```
1 | array=(1 abc "def" jtx)
```

也可以直接定义数组中某个元素的值：

```
1 array[0]=1
2 array[1]=abc
3 array[2]="def"
4 array[3]=jtx
```

### 读取数组中某个元素的值

格式：

```
1 ${array[index]}
```

例如：

```
1 array=(1 abc "def" jtx)
2 echo ${array[0]}
3 echo ${array[1]}
4 echo ${array[2]}
5 echo ${array[3]}
```

### 读取整个数组

格式：

```
1 ${array[@]} #第一种写法
2 ${array[*]} #第二种写法
```

例如：

```
1 array=(1 abc "def" jtx)
2
3 echo ${array[@]} #第一种读取整个数组的写法
4 echo ${array[*]} #第二种读取整个数组的写法
```

### 数组长度

类似于字符串

```
1 ${#array[@]} #第一种写法
2 ${#array[*]} #第二种写法
```

例如：

```
1 array=(1 abc "def" jtx)
2
3 echo ${#array[@]} #第一种写法
4 echo ${#array[*]} #第二种写法
```

```
操作 设置 帮助
acs@740d1bdabef2:~$ ls
EOF      test.sh  test6.sh
homework test4.sh test_zs.sh
main.cpp test5.sh testbl.sh
acs@740d1bdabef2:~$ ./test5.sh
读取数组中的每个元素：
1
abc
def
jtx
读取整个数组：
第一种方式:1 abc def jtx
第二种方式:1 abc def jtx
数组长度：
第一种方式，数组长度：4
第二种方式，数组长度：4
acs@740d1bdabef2:~$

1 #!/bin/bash
2
3 array=(1 abc "def" jtx)
4
5 echo "读取数组中的每个元素："
6 echo ${array[0]}
7 echo ${array[1]}
8 echo ${array[2]}
9 echo ${array[3]}
10
11
12 echo "读取整个数组："
13 echo "第一种方式:${array[*]}"
14 echo "第二种方式:${array[@]}"
15
16 echo "数组长度："
17 echo "第一种方式，数组长度:${#array[@]}"
18 echo "第二种方式，数组长度:${#array[*]}"
19
20
```

## 6. expr 命令

expr 命令用于求表达式的值，格式为：

```
1 | expr 表达式
```

表达式说明：

- 用空格隔开每一项
- 用反斜杠放在 shell 特定的字符前面(发现表达式运行错误时，可以试试转义)
- 对包含空格和其它特殊字符的字符串要用括号括起来
- expr 会在 stdout 中输出结果。如果为逻辑关系表达式，则结果为真，stdout 为1，否则为0。
- expr 的 exit code：如果为逻辑关系表达式，则结果为真，exit code 为0，否则为1。

### 字符串表达式

- length string

返回 string 的长度

- index string charset

charset 中任意单个字符在 string 中最前面的字符位置，下标从1开始。如果在 string 中完全不存在 charset 的字符，则返回0。

- substr string position length

返回 string 字符串从 position 开始，长度最大为 length 的子串。如果 position 或 length 为负数，0或非数值，则返回空字符串。

例如：

```
1 | str="Hello world!"
2
3 | echo `expr length "$str"` # 表达式要放在 `` 里面，表示执行改命令，输出12
4 | echo `expr index "$str" awd` #输出7，下标从1开始
5 | echo `expr substr "$str" 3 3` #输出llo
```



## 整数表达式

`expr` 支持普通的算术操作，算术表达式优先级低于字符串表达式，高于逻辑关系表达式。

- `+ -`

加减运算。两端参数会转换为整数，如果转换失败则报错。

- `* / %`

乘，除，取模。两端参数会转换为整数，如果转换失败则报错。

- `()` 括号里面的内容优先运算，但需要用反斜杠转义。

例如：

```
1 a=3
2 b=4
3
4 echo `expr $a + $b` #输出7
5 echo `expr $a - $b` #输出-1
6 echo `expr $a \* $b` #输出12， *需要转义
7 echo `expr $a / $b` #输出0，整除
8 echo `expr $a % $b` #输出3
9 echo `expr \($a + 1\) \* \($b + 1\)` #输出20， 值为(a+1) * (b+1)
```

## 逻辑关系表达式

- `|`

如果第一个参数非空且非0，则返回第一个参数的值。否则，当第二个参数非空且非零时，返回第二个参数；否则返回0。如果第一个参数非空或非零，不会计算第二个参数

- `&`

如果两个参数都非空且非零，则返回第一个参数，否则返回0。如果第一个参数为空或为零时，不会计算第二个参数。

- `< <= = == != >= >`

- 比较两端的参数，如果为 `true`，则返回1，否则返回0。`==` 是 `=` 的同义词。`expr` 首先尝试将两端转为整数，并做算术比较，如果转换失败，则按字符集排序规则做字符比较。

- `()`

括号里面的内容优先运算，但需要用反斜杠转义。

例如：

```
1 a=3
2 b=4
3
4 echo `expr $a \> $b` #输出0，需要转义
5 echo `expr $a '<' $b` #输出1，用''也可以
6 echo `expr $a '>=' $b` #输出0
7 echo `expr $a \<\ $b` #输出1
8
9 c=0
10 d=5
11
12 echo `expr $c \& $d` #输出0
13 echo `expr $a \& $b` #输出3
14 echo `expr $c \|| $d` #输出5
```

```
15 | echo `expr $a \| $b` #输出3
```

```
acs@740d1bdabef2:~$ ls
EOF      main.cpp  test4.sh  test_zs.sh
homework test.sh  test5.sh  testbl.sh
acs@740d1bdabef2:~$ ./test5.sh
test5.sh
7
-1
12
0
3
20
0
3
5
3
acs@740d1bdabef2:~$

1 #!/bin/bash
2
3 echo "test5.sh"
4
5 a=3
6 b=4
7
8 echo `expr $a + $b` #输出7
9 echo `expr $a - $b` #输出-1
10 echo `expr $a \* $b` #输出12, *要转义
11 echo `expr $a / $b` #输出0, 整除
12 echo `expr $a % $b` #输出3, 取模
13
14 echo `expr \( $a + 1 \| \) \* \( $b + 1 \| \)` #输出20, (a+1)*(b+1)
15
16 c=0
17 d=5
18
19 echo `expr $c \% $d` #输出0
20 echo `expr $a \% $b` #输出3
21 echo `expr $c \| $d` #输出5
22 echo `expr $a \| $b` #输出3
23
24
```

## 7. read 命令

read 命令用于从标准输入中读取单行数据。当读到文件结束符时，exit code 为1，否则为0。

### 参数说明

- -p: 后面可以接提示信息
- -t: 后面跟秒数，定义输入字符的等待时间，超过等待时间后会自动忽略此命令，但 -t 后面的命令继续执行

例如：

```
1 | acs@740d1bdabef2:~$ read name #读入name的值
2 | hello jtx #输入name的值
3 | acs@740d1bdabef2:~$ echo $name
4 | hello jtx #输出name的值
5 | acs@740d1bdabef2:~$ read -p "what's your name?" -t 10 name #读入name的值，等待时
   | 间为10s, -p提示语
6 | what's your name?my name is jtx #输入name的值
7 | acs@740d1bdabef2:~$ echo $name #输出name的值
8 | my name is jtx #标准输出
9 | acs@740d1bdabef2:~$
```

```
acs@740d1bdabef2:~$ read name
hello jtx
acs@740d1bdabef2:~$ echo $name
hello jtx
acs@740d1bdabef2:~$ read -p "what's your name?" -t 10 name
what's your name?my name is jtx
acs@740d1bdabef2:~$ echo $name
my name is jtx
acs@740d1bdabef2:~$ |
```

## 8. echo 命令

echo 用于输出字符串。命令格式：

```
1 | echo string
```

## 显示普通字符串

```
1 echo "Hello jtx"
2 echo Hello jtx #可以直接输出想表达的字符串
```

## 显示转义字符

```
1 echo "\"Hello jtx\"" # 转义双引号
2 echo \"Hello jtx\" #省略外层双引号
3
4 #示例
5 acs@740d1bdabef2:~$ echo "\"Hello jtx\""
6 "Hello jtx"
7 acs@740d1bdabef2:~$ echo \"Hello jtx\"
8 "Hello jtx"
9
```

## 显示变量

```
1 name=jtx
2 echo "wo shi $name" #输出 wo shi jtx
3
4 #示例
5 acs@740d1bdabef2:~$ name=jtx
6 acs@740d1bdabef2:~$ echo "wo shi $name"
7 wo shi jtx #输出结果
```

## 显示换行

```
1 echo -e "Hi\n" # -e 开启转义
2 echo "jwjtx"
3
4 #示例
5 acs@740d1bdabef2:~$ echo -e "Hi \n jwjtx"
6 Hi #输出，换行
7
8 jwjtx
9
10 acs@740d1bdabef2:~$ echo "Hi \n jwjtx"
11 Hi \n jwjtx #输出，不换行
```

## 显示不换行

```
1 echo -e "Hi \c" # -e 开启转义 \c 不换行
2 echo "jwjtx"
3
4 #示例
5 acs@740d1bdabef2:~$ echo -e "Hi \c"
6 Hi acs@740d1bdabef2:~$ #输出 Hi 后，没有换行
```

## 显示结果定向至文件

```
1 | echo "Hello world" > out.txt    #将内容输出到out.txt中
```

## 原样输出字符串，不进行转义或取变量(用单引号)

```
1 | name=jtx
2 | echo '$name\''
3 |
4 | #输出结果
5 | acs@740d1bdabef2:~$ name=jtx
6 | acs@740d1bdabef2:~$ echo '$name\''
7 | $name\''          #输出结果
```

## 显示命令的执行结果

```
1 | echo `date`
2 |
3 | #输出结果
4 | acs@740d1bdabef2:~$ echo `date`
5 | Tue Dec 21 16:02:24 CST 2021
6 |
```

## 9.printf命令

printf 命令用于格式化输出，类似于 C++ 的 printf 函数。

默认不会在字符串末尾添加换行符。

命令格式：

```
1 | printf format-string [arguments...]
```

### 用法示例

脚本内容：

```
1 | printf "%10d.\n" 123    # 占10位，右对齐
2 | printf "%-10.2f.\n" 566.545532 #占10位，保留2位小数，左对齐
3 | printf "my name is %s\n" "jtx" #格式化输出字符串
4 | printf "%d * %d = %d\n" 2 3 `expr 2 \* 3` #表达式的值作为参数
5 |
6 | #输出样例
7 |      123.
8 | 566.55 .
9 | my name is jtx
10 | 2 \* 3 = 6
11 |
```

```
AC terminal
操作 设置 帮助

1 #!/bin/bash
2
3 echo "test8.sh"
4
5 printf "%10d.\n" 123      #占10位，右对齐
6 printf "%-10.2f, \n" 566.54532
7 printf "my name is %s \n" jtx      #格式化输出字符串
8 printf "%d \* %d = %d\n" 2 3 `expr 2 \* 3` #表达式的值作为参数
9
10

acs@740d1bdabef2:~$ ls
EOF homework main.cpp tes
acs@740d1bdabef2:~$ ./test8.sh
test8.sh
      123.
566.55 .
my name is jtx
2 \* 3 = 6
acs@740d1bdabef2:~$ |
```

## 10. test 命令与判断符号 []

### 逻辑运算符 && 和 ||

- && 表示与，|| 表示或
- 二者具有短路原则：
  - expr1 && expr2: 当 expr1 为假时，直接忽略 expr2
  - expr1 || expr2: 当 expr1 为真时，直接忽略 expr2
- 表达式的 exit code 为0，表示真；为非零，表示假。(与我们平常所见相反)

### test 命令

在命令行中输入 `man test`，可以查看 `test` 命令的用法。

`test` 命令用于判断文件类型，以及对变量做比较。

`test` 命令用 `exit code` 返回结果，而不是使用 `stdout`。0表示真，非0表示假。

例如：

```
1 test 2 -lt 3      # 2<3, 为真, 返回值为0
2 echo $?          #输出上个命令的返回值, 输出0
3
4 acs@740d1bdabef2:~$ test 2 -lt 3
5 acs@740d1bdabef2:~$ echo $?
6 0
```

```
1 acs@740d1bdabef2:~$ ls
2 EOF homework main.cpp test.sh test4.sh test5.sh test6.sh test8.sh
  test_zs.sh testbl.sh
3 acs@740d1bdabef2:~$ test -e test.sh && echo "exist" || echo "not exist" #判断
  test.sh是否存在, 存在输出"exist", 不存在输出"not exist"
4 exist
5 acs@740d1bdabef2:~$ test -e test1.sh && echo "exist" || echo "not exist" #同理
  判断test1.sh是否存在
6 not exist
```

### 文件类型判断

命令格式：

```
1 test -e filename      #判断文件是否存在
```

相关参数：

- `-e`：代表文件是否存在
- `-f`：判断是否为文件
- `-d`：判断是否为目录

## 文件权限判断

命令格式：

```
1 | test -r filename    #判断文件是否可读
```

相关参数：

- `-r`：文件是否可读
- `-w`：文件是否可写
- `-x`：文件是否可执行
- `-s`：是否为非空文件

## 整数间的比较

命令格式：

```
1 | test $a -eq $b    # a 是否等于 b
```

相关参数：

- `-eq`：两端是否相等
- `-ne`：两端是否不相等
- `-gt`：左边是否大于右边
- `-lt`：左边是否小于右边
- `-ge`：左边是否大于等于右边
- `-le`：左边是否小于等于右边

## 字符串比较

测试参数	代表意义
<code>test -z string</code>	判断string是否为空，如果为空，则返回 <code>true</code>
<code>test -n string</code>	判断string是否非空，如果非空，则返回 <code>true</code> (-n 可以省略)
<code>test str1 == str2</code>	判断 str1 是否等于 str2
<code>test str1 != str2</code>	判断 str1 是否不等于 str2

## 多重条件判定

命令格式：

```
1 | test -r filename -a -x filename
```

相关参数：

- `-a`：两条件是否同时成立
- `-o`：两条件是否至少成立一个

- `!`: 取反。

## 判断符号 `[]`

`[]` 和 `test` 用法几乎一模一样，更常用于 `if` 语句中。另外 `[]` 是 `[]` 的加强版，支持的特性更多。

例如：

```
1 [ 2 -lt 3 ] #2<3,为真, 返回值为0
2 echo $? #输出上个命令的返回值, 0
3
4 #样例
5 acs@740d1bdabef2:~$ [ 2 -lt 3 ]
6 acs@740d1bdabef2:~$ echo $?
7 0
```

```
1 acs@740d1bdabef2:~$ ls
2 EOF homework main.cpp test.sh test4.sh test5.sh test6.sh test8.sh
   test_zs.sh testbl.sh
3 acs@740d1bdabef2:~$ [ -e test.sh ] && echo "exist" || echo "not exist"
4 exist
5 acs@740d1bdabef2:~$ [ -e test1.sh ] && echo "exist" || echo "not exist"
6 not exist
```

注意：

- `[]` 内的每一项都要用空格隔开
- 中括号内的变量，最好用双引号括起来
- 中括号内的常数，最好用单或双引号括起来

例如：

```
1 name="hi jtx"
2 [ $name == "hi jtx" ] #错误, 等价于[ hi jtx == "hi jtx" ]
3 [ "$name" == "hi jtx" ] #正确
4
5 #示例
6 acs@740d1bdabef2:~$ name="hi jtx"
7 acs@740d1bdabef2:~$ [ $name == "hi jtx" ] #错误, 参数太多
8 -bash: [: too many arguments
9 acs@740d1bdabef2:~$ [ "$name" == "hi jtx" ] #正确, 0为真
10 acs@740d1bdabef2:~$ echo $?
11 0
```

```
acs@740d1bdabef2:~$ ls
EOF homework main.cpp test.sh test4.sh test5.sh test6.sh test8.sh test_zs.sh
acs@740d1bdabef2:~$ test -e test.sh && echo "exist" || echo "not exist"
exist
acs@740d1bdabef2:~$ test -e test1.sh && echo "exist" || echo "not exist"
not exist
acs@740d1bdabef2:~$ [ 2 -lt 3]
-bash: [: missing `]'
acs@740d1bdabef2:~$ [ 2 -lt 3 ]
acs@740d1bdabef2:~$ echo $?
0
acs@740d1bdabef2:~$ [ -e test.sh ] && echo "exist" || echo "not exist"
exist
acs@740d1bdabef2:~$ [ -e test1.sh ] && echo "exist" || echo "not exist"
not exist
acs@740d1bdabef2:~$
acs@740d1bdabef2:~$
acs@740d1bdabef2:~$ name="jtx"
acs@740d1bdabef2:~$ name="hi jtx"
acs@740d1bdabef2:~$ [ $name == "hi jtx" ]
-bash: [: too many arguments
acs@740d1bdabef2:~$ [ "$name" == "hi jtx" ]
acs@740d1bdabef2:~$ echo $?
0
acs@740d1bdabef2:~$ |
```

## 11.判断语句

待更新....

## 12.循环语句

待更新....

## 13.函数

待更新...

## 14.exit 命令

待更新....

## 15.文件重定向

待更新...

## 16.引入外部脚本

待更新...