Bashの操作

——Bash确实需要系统得学习一下
基本参考的是 http://www.runoob.com/linux/linux-shell.html

Section1. Introduction

subsection1. Bourne Again Shell
subsection2. 解释器 #!/bin/bash

Shell 脚本

Shell 脚本（shell script），是一种为 shell 编写的脚本程序。

业界所说的 shell 通常都是指 shell 脚本，但读者朋友要知道，shell 和 shell script 是两个不同的概念。

由于习惯的原因，简洁起见，本文出现的 "shell编程" 都是指 shell 脚本编程，不是指开发 shell 自身。

Shell 环境

Shell 编程跟 java、php 编程一样，只要有一个能编写代码的文本编辑器和一个能解释执行的脚本解释器就可以了。

Linux 的 Shell 种类众多，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）
- Shell for Root（/sbin/sh）
- ……

本教程关注的是 Bash，也就是 Bourne Again Shell，由于易用和免费，Bash 在日常工作中被广泛使用。同时，Bash 也是大多数Linux 系统默认的 Shell。

在一般情况下，人们并不区分 Bourne Shell 和 Bourne Again Shell，所以，像 **#!/bin/sh**，它同样也可以改为 **#!/bin/bash**。

`#!` 告诉系统其后路径所指定的程序即是解释此脚本文件的 Shell 程序。

subsection3. 注释
3.1 单行注释
#
3.2 多行注释
:<<EOF
中间写需要注释的代码
亦可以使用其他字符替代EOF
EOF

Section2. 执行

码了HelloWorld
执行一下

```
1 #!/bin/bash
2
3 echo Hello World!
```

```
➜ BashTest git:(master) ✗ vim HelloWorld.sh
➜ BashTest git:(master) ✗ chmod u+x ./HelloWorld.sh
➜ BashTest git:(master) ✗ git add HelloWorld.sh
➜ BashTest git:(master) ✗ git commit -m 'Hello World'
[master (root-commit) f56bb58] Hello World
 1 file changed, 3 insertions(+)
 create mode 100755 HelloWorld.sh
➜ BashTest git:(master) git push -u origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 247 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/fz17/BashTest.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
➜ BashTest git:(master) ./HelloWorld.sh
Hello World!
```

./ filename 执行

亦可以使用下面这句命令执行（若如此做，就不需要#!/bin/bash）
/bin/bash filename 执行

Section3. 变量常规操作

subsection1. 定义变量

定义变量时，不用$，等号两侧无空格

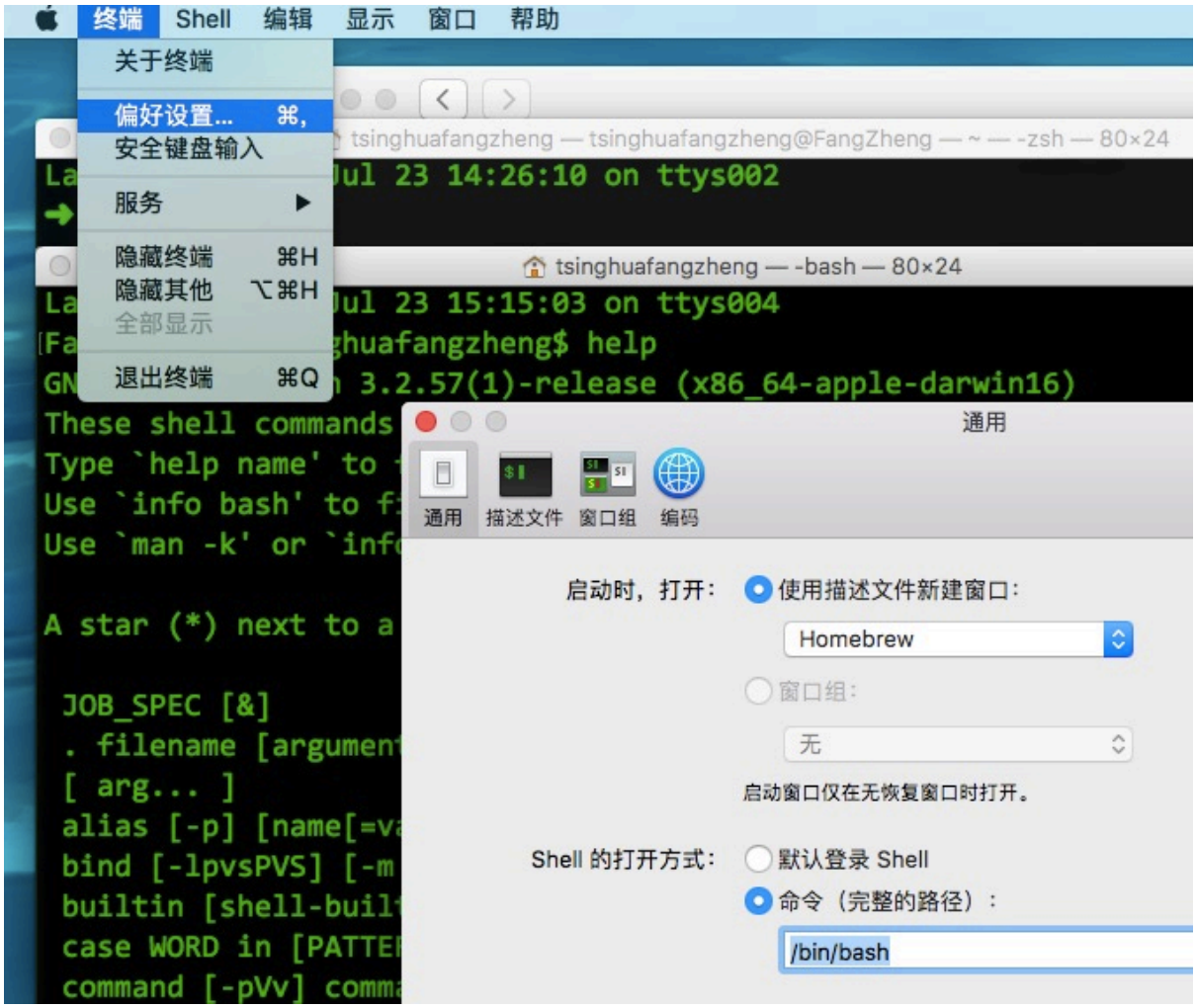
定义变量时，变量名不加美元符号（$，PHP语言中变量需要），如：

```
your_name="runoob.com"
```

注意，变量名和等号之间不能有空格，这可能和你熟悉的所有编程语言都不一样。同时，变量名的命名须遵循如下规则：

- 命名只能使用英文字母，数字和下划线，首个字符不能以数字开头。
- 中间不能有空格，可以使用下划线（_）。
- 不能使用标点符号。
- 不能使用bash里的关键字（可用help命令查看保留关键字）。

题外话，由于我使用的是zsh，不支持bash下的 help 命令，
所以需要切换回 bash，再 help，具体操作如下：



subsection2. 使用变量

使用，前置$符号
推荐使用大括号（$默认管到空格符前）

```
1 #!/bin/bash
2
3 my_name='Zixiangninglu'
4
5 echo my name is ${my_name}
6
```

```
➜ BashTest git:(master) ✗ vim Variable.sh
➜ BashTest git:(master) ✗ ./Variable.sh
my name is Zixiangninglu
```

subsection3. 重新赋值

重新赋值时，不能用$

```
1 #!/bin/bash
2
3 my_name='Zixiangninglu'
4
5 my_name='LittleFangZheng'
6
7 echo my name is ${my_name}
8
```

```
➜ BashTest git:(master) ✗ vim Variable.sh
➜ BashTest git:(master) ✗ ./Variable.sh
my name is Zixiangninglu
➜ BashTest git:(master) ✗ vim Variable.sh
➜ BashTest git:(master) ✗ ./Variable.sh
my name is LittleFangZheng
```

subsection4. 指定只读变量
readonly my_name
用于赋完值后，后续便不可更改

subsection5. 删除变量
unset my_name

Section4. 字符串变量

subsection1. 单双引号
单引号原封不动输出，不支持内置变量，不支持内置转义字符；
双引号支持内置变量，支持内置转义字符。

subsection2. 字符拼接
支持字符串直接无空格拼接

```
#!/bin/bash

str1="LittleFangZheng"

str2="Hello ${str1}!"

str3="Hello \"${str1}\"!"

str4="Hello "${str1}"!"


echo ${str2}
echo ${str3}
echo ${str4}
```

```
➜ BashTest git:(master) ✗ vim String.sh
➜ BashTest git:(master) ✗ ./String.sh
Hello LittleFangZheng!
Hello "LittleFangZheng"!
Hello LittleFangZheng!
```

echo ${#str_name} 输出字符串长度

Section5. 数组

subsection1. 定义数组

subsection2. 取数
数组下标从0开始

subsection3. 操作
bash好像天生就不是为科学计算设计的
在bash中，运算（后文 section7 详述）需要
3.1 在`expr ` 中运行
3.2 在 $(( )) 中运行

```
#!/bin/bash

array=(1 0 2 3 5 8 0)

array[1]=1

array[6]=`expr ${array[4]} + ${array[5]}`
array[6]=$((${array[4]} + ${array[5]}))

echo ${array[6]}
echo ${#array[*]}
~
```

```
➜ BashTest git:(master) ✗ vim Array.sh
➜ BashTest git:(master) ✗ ./Array.sh
13
7
```

Section6. 传递参数

subsection1. Regular

$后加数字，

可指代传入的第几个参数

（下面的图片，附带处理了echo的一个小问题）

```
#!/bin/bash

echo 'The file is $0'
echo 'The first input is $1'
echo 'The second input is $2'
~
```

这样并不能达到我们的意图，可这样：

```
#!/bin/bash

echo The file is $0
echo The first input is $1
echo "The second input is $2"

echo Total variable $#
```

运行之后是这样的：

```
➜  BashTest git:(master) ✗ vim Transfer.sh
➜  BashTest git:(master) ✗ ./Transfer.sh 05 22
The file is $0
The first input is $1
The second input is $2
➜  BashTest git:(master) ✗ vim Transfer.sh
➜  BashTest git:(master) ✗ ./Transfer.sh 05 22
The file is ./Transfer.sh
The first input is 05
The second input is 22
```

subsection2. Others
这里引一张表，详述了$的各种：

$0	程序的名字
$1~$9	命令行参数1~9(常用户脚本传递参数)
$*	所有命令行参数值
$@	所有命令行参数值
$#	命令行参数的总个数
$$	当前进程的ID
$?	最近一次命令的退出状态
$!	最近一次后台进程的ID号

```
1 #!/bin/bash
2
3 echo The file is $0
4 echo The first input is $1
5 echo The second input is $2
6
7 echo Total variable $#
8
9 echo All $*
```

顺道提一下，$* 与 $@ 基本没区别，
仅仅是在外套了双引号之后， “$*" 与 “$@" 略有区别：
"$*" 相当于python中的 list('$1 $2')
“$@" 相当于python中的 list((‘$1’, ‘$2'))

```
➜ BashTest git:(master) ✗ vim Transfer.sh
➜ BashTest git:(master) ✗ ./Transfer.sh 05 22
The file is ./Transfer.sh
The first input is 05
The second input is 22
Total variable 2
All 05 22
```

Section7. 基本运算

这章很重要，内容也比较多

subsection1. 算术运算符

算术运算符

下表列出了常用的算术运算符，假定变量 a 为 10，变量 b 为 20：

运算符	说明	举例
+	加法	`expr $a + $b` 结果为 30。
-	减法	`expr $a - $b` 结果为 -10。
*	乘法	`expr $a \* $b` 结果为 200。
/	除法	`expr $b / $a` 结果为 2。
%	取余	`expr $b % $a` 结果为 0。
=	赋值	a=$b 将把变量 b 的值赋给 a。
==	相等。用于比较两个数字，相同则返回 true。	[$a == $b] 返回 false。
!=	不相等。用于比较两个数字，不相同则返回 true。	[$a != $b] 返回 true。

注意：条件表达式要放在方括号之间，并且要有空格，例如: **[$a==$b]** 是错误的，必须写成 **[$a == $b]**。

还有，值得注意的是，`expr`中，乘号*前需加上反斜杠组成 * 才能表达乘法

subsection2. 关系运算符

关系运算符

关系运算符只支持数字，不支持字符串，除非字符串的值是数字。

下表列出了常用的关系运算符，假定变量 a 为 10，变量 b 为 20：

运算符	说明		举例
-eq	检测两个数是否相等，相等返回 true。	equal	[$a -eq $b] 返回 false。
-ne	检测两个数是否不相等，不相等返回 true。	not equal	[$a -ne $b] 返回 true。
-gt	检测左边的数是否大于右边的，如果是，则返回 true。	great than	[$a -gt $b] 返回 false。
-lt	检测左边的数是否小于右边的，如果是，则返回 true。	less than	[$a -lt $b] 返回 true。
-ge	检测左边的数是否大于等于右边的，如果是，则返回 true。	great equal	[$a -ge $b] 返回 false。
-le	检测左边的数是否小于等于右边的，如果是，则返回 true。	less than	[$a -le $b] 返回 true。

subsection3. 布尔运算符

布尔运算符

下表列出了常用的布尔运算符，假定变量 a 为 10，变量 b 为 20：

运算符	说明	举例
-o	或运算，有一个表达式为 true 则返回 true。	[$a -lt 20 -o $b -gt 100] 返回 true。
-a	与运算，两个表达式都为 true 才返回 true。	[$a -lt 20 -a $b -gt 100] 返回 false。

subsection4. 逻辑运算符

逻辑运算符

以下介绍 Shell 的逻辑运算符，假定变量 a 为 10，变量 b 为 20:

运算符	说明	举例
&&	逻辑的 AND	[[$a -lt 100 && $b -gt 100]] 返回 false
||	逻辑的 OR	[[$a -lt 100 || $b -gt 100]] 返回 true

subsection5. 字符串运算符

字符串运算符

下表列出了常用的字符串运算符，假定变量 a 为 "abc"，变量 b 为 "efg"：

运算符	说明	举例
=	检测两个字符串是否相等，相等返回 true。	[$a = $b] 返回 false。
!=	检测两个字符串是否相等，不相等返回 true。	[$a != $b] 返回 true。
str	检测字符串是否为空，不为空返回 true。	[$a] 返回 true。

subsection6. 文件测试运算符

文件测试运算符

文件测试运算符用于检测 Unix 文件的各种属性。

属性检测描述如下：

操作符	说明	举例
-b file	检测文件是否是块设备文件，如果是，则返回 true。	[-b $file] 返回 false。
-c file	检测文件是否是字符设备文件，如果是，则返回 true。	[-c $file] 返回 false。
-d file	检测文件是否是目录，如果是，则返回 true。	[-d $file] 返回 false。
-f file	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。	[-f $file] 返回 true。
-g file	检测文件是否设置了 SGID 位，如果是，则返回 true。	[-g $file] 返回 false。
-k file	检测文件是否设置了粘着位(Sticky Bit)，如果是，则返回 true。	[-k $file] 返回 false。
-p file	检测文件是否是有名管道，如果是，则返回 true。	[-p $file] 返回 false。
-u file	检测文件是否设置了 SUID 位，如果是，则返回 true。	[-u $file] 返回 false。
-r file	检测文件是否可读，如果是，则返回 true。	[-r $file] 返回 true。
-w file	检测文件是否可写，如果是，则返回 true。	[-w $file] 返回 true。
-x file	检测文件是否可执行，如果是，则返回 true。	[-x $file] 返回 true。
-s file	检测文件是否为空（文件大小是否大于0），不为空返回 true。	[-s $file] 返回 true。
-e file	检测文件（包括目录）是否存在，如果是，则返回 true。	[-e $file] 返回 true。

觉得后五个有些用处

```
 1 #!/bin/bash
 2
 3 file="./Array.sh"
 4
 5 if [ -e $file ]
 6 then
 7     echo 'Exist'
 8 else
 9     echo 'Do not exist'
10 fi
11
12 if [ -r $file ]
13 then
14     echo 'Can be read'
15 else
16     echo 'Can not be read'
17 fi
18
19
20 if [ -w $file ]
21 then
22     echo 'Can be written'
23 else
24     echo 'Can not be written'
25 fi
26
27
28 if [ -x $file ]
29 then
30     echo 'Can be execute'
31 else
32     echo 'Can not be execute'
33 fi
~
```

```
➜ BashTest git:(master) ✗ vim Operator.sh
➜ BashTest git:(master) ✗ ./Operator.sh
Exist
Can be read
Can be written
Can be execute
```

Section8. echo

subsection1. 转义
每句echo自动换行，
echo 默认不转义，
开启转义需使用 -e 选项
\n 换行
\c 不换行

subsection2. 与read交互
read命令读取一行，并赋给read之后的变量
若有输入（以空格分隔）比需求多，多的部分全赋给最后一变量

```
 1 #!/bin/bash
 2
 3 echo "A"
 4 echo "B\n"
 5 echo -e "C"
 6 echo -e "D\n"
 7 echo -e "E\c"
 8 echo "F"
 9
10 read str1
11 echo "Input is ${str1}"
12
13 read str2 str3
14 echo "Left input is ${str2}; Right input is ${str3}"
```

```
➜ BashTest git:(master) ✗ vim Echo.sh
➜ BashTest git:(master) ✗ chmod u+x ./Echo.sh
➜ BashTest git:(master) ✗ ./Echo.sh
A
B\n
C
D

EF
G
Input is G
H I J K
Left input is H; Right input is I J K
```

Section9. printf

与我学的C 大部分一致
但，这里的 printf 不需 ( )
亦不用 ， 而用空格分隔

```
1 #!/bin/bash
2
3 printf "%d is %s\n" 1 "A"
4
5 printf "%-10s wights %4.2fkg\n" "LittleFangZheng" 58.5
```

-是左对齐，不加为默认右对齐

```
➜ BashTest git:(master) ✗ ./Printf.sh
1 is A
LittleFangZheng wights 58.50kg
```

Section10. 流程控制

subsection1. 判断
1.1 if then fi
1.2 if then else fi
1.3 if then elif then elif then … else fi

```
 1 #!/bin/bash
 2
 3 read score
 4
 5 if [ ${score} -lt 60 ]
 6 then
 7     echo N
 8 elif [ ${score} -lt 80 ]
 9 then
10     echo B
11 else
12     echo A
13 fi
~
```

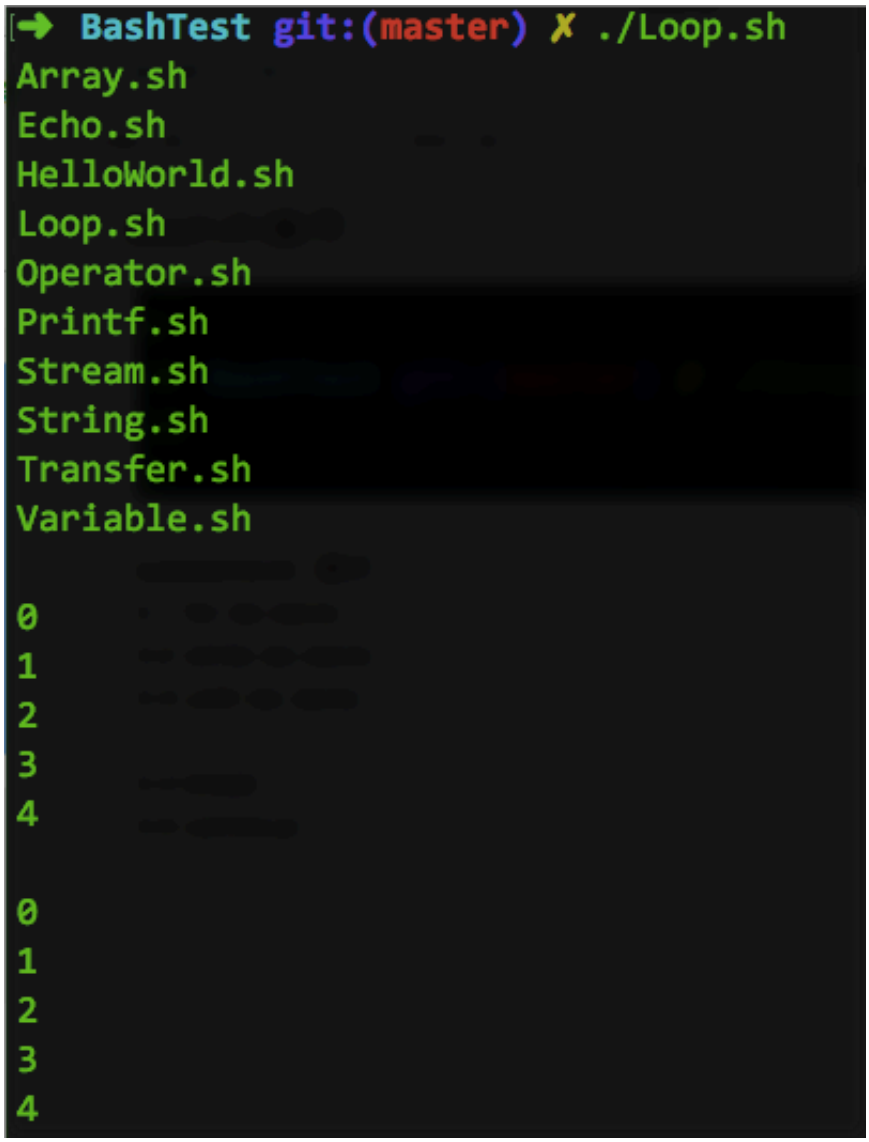
提请注意，[ ]中括号内侧均有空格

```
➜ BashTest git:(master) ✗ ./Stream.sh
75
B
➜ BashTest git:(master) ✗ ./Stream.sh
95
A
➜ BashTest git:(master) ✗ ./Stream.sh
42
N
➜ BashTest git:(master) ✗ ./Stream.sh
60
B
```

subsection2. 循环
2.1 for do done
2.2 while do done
2.3 until do done

```bash
#!/bin/bash

for filename in `ls ./`
do
    echo ${filename}
done
echo


i=0
while [ ${i} -lt 5 ]
do
    echo ${i}
    i=`expr ${i} + 1`
done
echo


i=0
until [ ! ${i} -lt 5 ]
do
    echo ${i}
    i=`expr ${i} + 1`
done

```

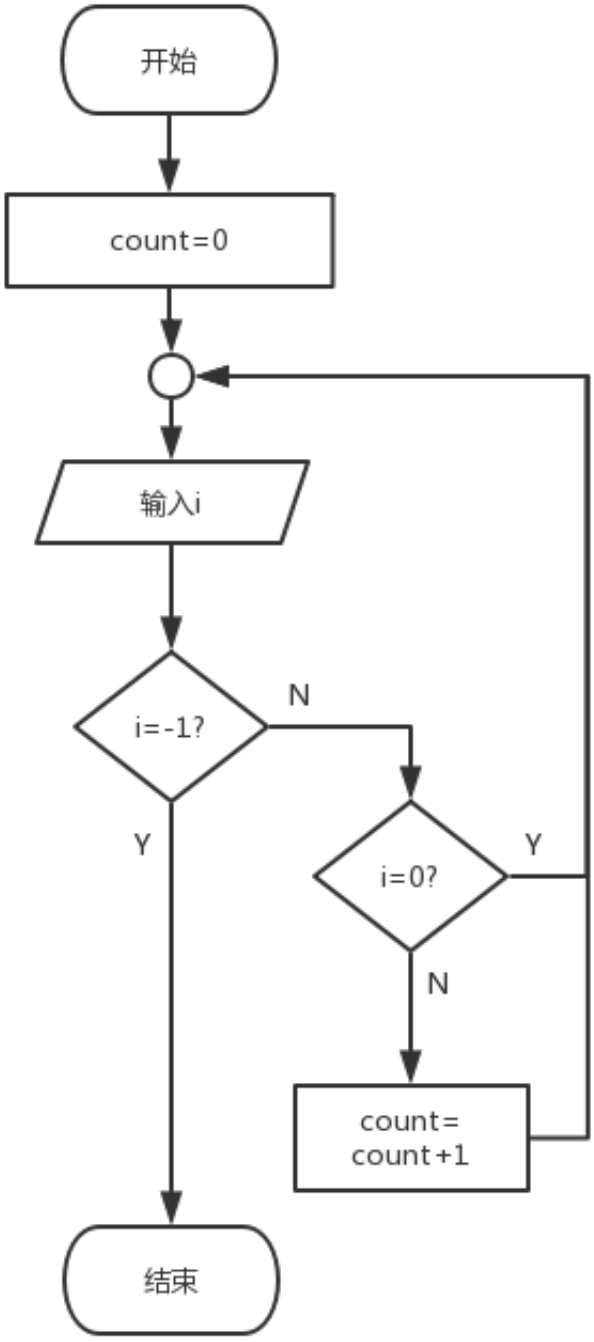
提请注意，赋值=两侧，无空格
另外，bash还支持case，
但个人感觉case较鸡肋，不介绍



2.4 break
2.5 continue

```
 1 #!/bin/bash
 2
 3 count=0
 4 while true
 5 do
 6     read i
 7     if [ ${i} -eq -1 ]
 8     then
 9         break
10     elif [ ${i} -eq 0 ]
11     then
12         continue
13     else
14         count=`expr ${count} + 1`
15     fi
16 done
17
18 echo "The counter is ${count}"
~
```

流程图是这样的：


开始
count=0
输入i
i=-1?
N
Y
i=0?
Y
N
count=
count+1
结束

```
➜ BashTest git:(master) ✗ ./BandC.sh
0
5
0
22
522
0
-1
The counter is 3
```

Section11. 函数

通过以下形式定义函数

```
function func_name ()
{
   sentence1
   sectence2
   …
   return int
}
```

```
 1 #!/bin/bash
 2
 3 function my_add_one(){
 4     read ele1 ele2
 5     return `expr ${ele1} + ${ele2}`
 6 }
 7
 8 my_add_one
 9 echo "First sum is $?"
10
11 function my_add_two(){
12     echo "Second sum is `expr $1 + $2`"
13 }
14 my_add_two 5 22
~
```

使用了两种方法，其中第一种中的

$? 调用最近一次命令的退出状态

我觉得第二种
优秀很多

```
➜ BashTest git:(master) ✗ ./Function.sh
5 22
First sum is 27
Second sum is 27
➜ BashTest git:(master) ✗ vim Function.sh
```

section12. 重定向

subsection1. 输出重定向

>

subsection2. 输入重定向

<

```
➜ BashTest git:(master) ✗ who
tsinghuafangzheng console  Jul 24 14:27
tsinghuafangzheng ttys000  Jul 24 14:29
➜ BashTest git:(master) ✗ who > ./tempout
➜ BashTest git:(master) ✗ cat ./tempout
tsinghuafangzheng console  Jul 24 14:27
tsinghuafangzheng ttys000  Jul 24 14:29
➜ BashTest git:(master) ✗ wc -l ./tempout
       2 ./tempout
➜ BashTest git:(master) ✗ wc -l < ./tempout
       2
```

subsection3. 结合

从in_file获取输入，将输出写入out_file

command < in_file > out_file

subsection4. stdin stdout stdeer 的重定向

4.1 将stderr重定向到file

command 2 > file

4.2 将stderr和stdout合并后，重定向到file

command > file 2>&1

subsection5. Here Document

```
command << delimiter
   sentence1
   sentence2
   ...
delimiter
```

将sentences 重定向输入到 command

Section13. 管道

管道（pipe）左侧命令的输出作为右侧命令的输入

```
➜ BashTest git:(master) ✗ cat Echo.sh
#!/bin/bash

echo "A"
echo "B\n"
echo -e "C"
echo -e "D\n"
echo -e "E\c"
echo "F"

read str1
echo "Input is ${str1}"

read str2 str3
echo "Left input is ${str2}; Right input is ${str3}"

➜ BashTest git:(master) ✗ cat Echo.sh | grep -n 'echo'
3:echo "A"
4:echo "B\n"
5:echo -e "C"
6:echo -e "D\n"
7:echo -e "E\c"
8:echo "F"
11:echo "Input is ${str1}"
14:echo "Left input is ${str2}; Right input is ${str3}"
```

Section14. 包含

Bash也有类似于python中import的功能

这是SourceOne.sh

```
  1 #!/bin/bash
  2
  3 my_name="LittleFangZheng"
  4
~
```

这是SourceTwo.sh

```
  1 #!/bin/bash
  2
  3 source ./SourceOne.sh
  4
  5 echo "My name is ${my_name}"
~
```

运行

```
➜  BashTest git:(master) ✗ vim SourceOne.sh
➜  BashTest git:(master) ✗ vim SourceTwo.sh
➜  BashTest git:(master) ✗ chmod u+x ./Source*
➜  BashTest git:(master) ✗ ./SourceTwo.sh
My name is LittleFangZheng
```