目　录

致谢

当前文档 《Shell编程基础》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-03-01。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常生活、工作和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：http://www.bookstack.cn/books/ShellBasicProgramming

书栈官网：http://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

介绍

Shell-编程基础

该系列文集主要介绍的是，关于Linux下的Shell编程基础，包括以下内容：

来源：

> *https://github.com/solar555/ShellBasicProgramming*

01. Shell变量

1. Shell变量

变量

```
1. my_name="Asa"
```

命名规则：

- 首个字符必须为字母（a-z，A-Z）。
- 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）。

使用变量

```
1. my_name="Asa"
2. echo $my_name
3. echo ${my_name}
```

备注：后两句效果一样，均为输出变量的值。变量名外面的花括号是可选的，加花括号是为了帮助解释器识别变量的边界。

02. Linux下使用vi方向键乱码、删除键无效的解决方案

02. Linux下使用vi方向键乱码、删除键无效的解决方案

按如下步骤操作

1. 使用root权限打开文件vimrc.tiny，命令如下：

```
1. sudo vi /etc/vim/vimrc.tiny
```

2. 编辑

（1）解决方向键变“ABCD”问题

```
1. 将倒数第二句“set compatible”改为“set nocompatible”
```

（2）解决删除键无效的问题

```
1. 在“set nocompatible”后面增加一句“set backspace=2”
```

3. 保存并退出，问题解决

03. Shell 传递参数

3. Shell 传递参数

$* 与 $@ 区别

- 相同点：都是引用所有参数。
- 不同点：只有在双引号中体现出来。假设在脚本运行时写了三个参数 1、2、3，，则 “ * “ 等价于 “1 2 3”（传递了1个参数），而 “@” 等价于 “1” “2” “3”（传递了3个参数）。

```
# !/bin/bash
# author:Asa

echo "-- \$* 演示 ---"
for i in "$*"; do
    echo $i
done

echo "-- \$@ 演示 ---"
for i in "$@"; do
    echo $i
done
```

执行脚本，输出结果如下所示：

```
$ chmod +x test.sh
$ ./test.sh 1 2 3
-- $* 演示 ---
1 2 3
-- $@ 演示 ---
1
2
3
```

04. Shell 数组

4. Shell 数组

一、语法

```
array_name=(value1 value2 ... valuen)
```

实例

```
#!/bin/bash
# author:Asa

my_array=(A B "C" D)
```

二、获取元素

```
${array_name[index]}
```

实例一（获取单个元素）

```
#!/bin/bash
# author:Asa

my_array=(A B "C" D)

echo "第一个元素为: ${my_array[0]}"
echo "第二个元素为: ${my_array[1]}"
echo "第三个元素为: ${my_array[2]}"
echo "第四个元素为: ${my_array[3]}"
```

实例二（获取所有元素）

```
#!/bin/bash
# author:Asa

my_array[0]=A
my_array[1]=B
my_array[2]=C
my_array[3]=D

echo "所有元素为: ${my_array[*]}"
echo "所有元素为: ${my_array[@]}"
```

05. Shell 基本运算符

5. Shell 基本运算符

Shell支持的运算符:

- 算数运算符
- 关系运算符
- 布尔运算符
- 字符串运算符
- 文件测试运算符

一、算数运算符

原生bash不支持简单的数学运算，但是可以通过其他命令来实现，例如 awk 和 expr，expr 最常用。

expr 是一款表达式计算工具，使用它能完成表达式的求值操作。

例如，两个数相加(注意使用的是反引号 ` 而不是单引号 '):

```
#!/bin/bash

val=`expr 2 + 2`
echo "2+2=:$val"
```

注意:

1. 表达式和运算符之间要有空格，例如 2+2 是不对的，必须写成 2 + 2，这与我们熟悉的大多数编程语言不一样。
2. 完整的表达式要被“反单引号”(`)包含，注意这个字符不是常用的单引号，在 Esc 键下边。

二、关系运算符

```
a=10
b=20

if [ $a -eq $b ]
then
   echo "$a -eq $b : a 等于 b"
else
   echo "$a -eq $b : a 不等于 b"
fi
```

三、布尔运算符

```
a=10
b=20

if [ $a != $b ]
then
   echo "$a != $b : a 不等于 b"
else
   echo "$a != $b : a 等于 b"
fi
```

四、逻辑运算符

```
if [[ $a -lt 100 && $b -gt 100 ]]
then
   echo "返回 true"
else
   echo "返回 false"
fi
```

五、字符串运算符

```
a="abc"
b="efg"

if [ $a = $b ]
then
   echo "$a = $b : a 等于 b"
else
```

```
    echo "$a = $b : a 不等于 b"
fi
```

六、文件测试运算符

```
file="C:/helloworld.sh"
if [ -r $file ]
then
    echo "文件可读"
else
    echo "文件不可读"
fi
```

06. Shell echo命令

6. Shell echo命令

格式

```
1. echo string
```

1. 显示普通字符（双引号可以忽略）

```
1. echo "hello world"
2. echo hello world
```

2. 显示转移字符

```
1. echo "\"hello world\""
2. echo \"hello world\"
```

结果：

```
1. "hello world"
```

3. 显示变量
reed命令从标准输入中读取一行，并把输入行的每个字段的值指定给shell变量：

```
1. # !/bin/sh
2. read name
3. echo "$name is a test"
```

将以上代码保存为test.sh，name 接受标准输入的变量，运行结果：

```
1. asa@asa-virtual-machine:~$ sh hello.sh
2. Tom
3. Tom is my name
```

4. 显示换行

```
echo -e "OK! \n" # -e 开启转义
echo It is a test
```

输出结果：

```
OK!
It is a test
```

5. 显示不换行

```
# !/bin/sh
echo -e "OK! \c" # -e 开启转义 \c 不换行
echo It is a test
```

输出结果：

```
OK! It is a test
```

6. 显示结果定向至文件

```
echo It is a test > myfile
```

7. 原样输出字符串，不进行转义或取变量（用单引号）

```
echo '$name\"'
```

输出结果：

```
$name\"
```

8. 显示命令执行结果

```
echo `date`
```

注意：这里使用的是 反引号 `，而非单引号。
输出结果显示当前日期：

```
asa@asa-virtual-machine:~$ echo `date`
2016年 01月 22日 星期一 09:14:37 CST
```

07. Shell printf 命令

7. Shell printf 命令

语法

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

参数说明:

- format-string: 为格式控制字符串
- arguments: 为参数列表。

实例:

```
asa@asa-virtual-machine:~$ echo "Hello, Shell"
Hello, Shell
asa@asa-virtual-machine:~$ printf "Hello, Shell\n"
Hello, Shell
```

展现 printf 的一个强大功能:

```
# !/bin/bash
# author:Asa Guo

printf "%-10s %-8s %-4s\n" Name Sex Weight-kg
printf "%-10s %-8s %-4.2f\n" GuoJing male 66.1234
printf "%-10s %-8s %-4.2f\n" YangGuo male 48.6543
printf "%-10s %-8s %-4.2f\n" GuoFu female 47.9876
```

输出:

```
asa@asa-virtual-machine:~$ chmod +x ./printf.sh
asa@asa-virtual-machine:~$ ./printf.sh
Name        Sex      Weight-kg
GuoJing     male     66.12
YangGuo     male     48.65
GuoFu       female   47.99
```

> *%s %c %d %f*都是格式替代符
> *%-10s* 指一个宽度为*10*个字符（-表示左对齐，没有则表示右对齐），任何字符都会被显示在*10*个字符宽的字符内，如果不足则自动以空格填充，超过也会将内容全部显示出来。
> *%-4.2f* 指格式化为小数，其中.*2*指保留*2*位小数。

printf的转义序列

序列	说明
\a	警告字符，通常为ASCII的BEL字符
\b	后退
\c	抑制（不显示）输出结果中任何结尾的换行字符（只在%b格式指示符控制下的参数字符串中有效），而且，任何留在参数里的字符、任何接下来的参数以及任何留在格式字符串中的字符，都被忽略
\f	换页（formfeed）
\n	换行
\r	回车（Carriage return）
\t	水平制表符
\v	垂直制表符
\	一个字面上的反斜杠字符
\ddd	表示1到3位数八进制值的字符。仅在格式字符串中有效
\0ddd	表示1到3位的八进制值字符

08. Shell test 命令

8. Shell test 命令

Shell中的 test 命令用于检查某个条件是否成立，它可以进行数值、字符和文件三个方面的测试。

数值测试

参数	说明
-eq	等于则为真
-ne	不等于则为真
-gt	大于则为真
-ge	大于等于则为真
-lt	小于则为真
-le	小于等于则为真

示例：

```
num1=100
num2=100
if test $[num1] -eq $[num2]
then
    echo '两个数相等！'
else
    echo '两个数不相等！'
fi
```

输出：

```
两个数相等！
```

代码中的 [] 执行基本的算数运算，如：

```
#!/bin/bash

a=5
b=6

result=$[a+b] # 注意等号两边不能有空格
echo "result 为： $result"
```

输出：

```
result 为： 11
```

字符串测试

参数	说明
=	等于则为真
!=	不相等则为真
-z 字符串	字符串的长度为零则为真
-n 字符串	字符串的长度不为零则为真

示例：

```
num1="ru1noob"
num2="runoob"
if test $num1 = $num2
then
    echo '两个字符串相等!'
else
    echo '两个字符串不相等!'
fi
```

输出：

```
两个字符串不相等!
```

文件测试

参数	说明

-e 文件名	如果文件存在则为真
-r 文件名	如果文件存在且可读则为真
-w 文件名	如果文件存在且可写则为真
-x 文件名	如果文件存在且可执行则为真
-s 文件名	如果文件存在且至少有一个字符则为真
-d 文件名	如果文件存在且为目录则为真
-f 文件名	如果文件存在且为普通文件则为真
-c 文件名	如果文件存在且为字符型特殊文件则为真
-b 文件名	如果文件存在且为块特殊文件则为真

示例：

```
cd /bin
if test -e ./bash
then
    echo '文件已存在!'
else
    echo '文件不存在!'
fi
```

输出：

```
文件已存在!
```

Shell还提供了与(-a)、或(-o)、非(!)三个逻辑操作符用于将测试条件连接起来，优先级从高到低依次为："!"，"-a"，"-o"。

示例：

```
cd /bin
if test -e ./notFile -o -e ./bash
then
    echo '有一个文件存在!'
else
    echo '两个文件都不存在'
fi
```

输出：

```
有一个文件存在!
```

09. Shell 流程控制

9. Shell 流程控制

和Java、PHP等语言不一样，sh的流程控制不可为空，如(以下为PHP流程控制写法)：

```
<?php
if (isset($_GET["q"])) {
    search(q);
}
else {
    // 不做任何事情
}
```

在sh/bash里可不能这么写，如果else分支没有语句执行，就不要写这个else。

---

if else

if

if 语句语法格式:

```
if condition
```

```
2. then
3.     command1
4.     command2
5.     ...
6.     commandN
7. fi
```

写成一行（适用于终端命令提示符）：

```
1. if [ $(ps -ef | grep -c "ssh") -gt 1 ]; then echo "true"; fi
```

末尾的fi就是if倒过来拼写，后面还会遇到类似的。

if else

if else 语法格式：

```
1. if condition
2. then
3.     command1
4.     command2
5.     ...
6.     commandN
7. else
8.     command
9. fi
```

if else-if else

if else-if else 语法格式：

```
1. if condition1
2. then
3.     command1
4. elif condition2
5. then
6.     command2
7. else
8.     commandN
9. fi
```

示例：

```
a=10
b=20
if [ $a == $b ]
then
   echo "a 等于 b"
elif [ $a -gt $b ]
then
   echo "a 大于 b"
elif [ $a -lt $b ]
then
   echo "a 小于 b"
else
   echo "没有符合的条件"
fi
```

输出结果：

```
a 小于 b
```

---

if else语句经常与test命令结合使用，如下所示：

```
num1=$[2*3]
num2=$[1+5]
if test $[num1] -eq $[num2]
then
    echo '两个数字相等!'
else
    echo '两个数字不相等!'
fi
```

输出：

```
两个数字相等!
```

---

for 循环

一般格式：

```
for var in item1 item2 ... itemN
do
    command1
```

```
    command2
    ...
    commandN
done
```

写成一行：

```
for var in item1 item2 ... itemN; do command1; command2… done;
```

当变量值在列表里，for循环即执行一次所有命令，使用变量名获取列表中的当前取值。命令可为任何有效的shell命令和语句。in列表可以包含替换、字符串和文件名。

in列表是可选的，如果不用它，for循环使用命令行的位置参数。

例如，顺序输出当前列表中的数字：

```
for loop in 1 2 3 4 5
do
    echo "The value is: $loop"
done
```

输出：

```
The value is: 1
The value is: 2
The value is: 3
The value is: 4
The value is: 5
```

顺序输出字符串中的字符：

```
for str in 'This is a string'
do
    echo $str
done
```

输出结果：

```
This is a string
```

while语句

while循环用于不断执行一系列命令，也用于从输入文件中读取数据；命令通常为测试条件。其格式为：

```
while condition
do
    command
done
```

以下是一个基本的while循环，测试条件是：如果int小于等于5，那么条件返回真。int从0开始，每次循环处理时，int加1。运行上述脚本，返回数字1到5，然后终止。

```
#!/bin/sh
int=1
while(( $int<=5 ))
do
        echo $int
        let "int++"
done
```

输出：

```
1
2
3
4
5
```

使用中使用了 Bash let 命令，它用于执行一个或多个表达式，变量计算中不需要加上 $ 来表示变量，具体可查阅：Bash let 命令
。while循环可用于读取键盘信息。下面的例子中，输入信息被设置为变量FILM，按结束循环。

```
echo '按下 <CTRL-D> 退出'
echo -n '输入你最喜欢的电影名: '
while read FILM
do
    echo "是的！$FILM 是一部好电影"
done
```

输出：

```
按下 <CTRL-D> 退出
输入你最喜欢的电影名: w3cschool菜鸟教程
是的！w3cschool菜鸟教程 是一部好电影
```

无限循环

无限循环语法格式：

```
while :
do
    command
done
```

或者

```
while true
do
    command
done
```

或者

```
for (( ; ; ))
```

---

until 循环

until循环执行一系列命令直至条件为真时停止。
until循环与while循环在处理方式上刚好相反。
一般while循环优于until循环，但在某些时候—也只是极少数情况下，until循环更加有用。
until 语法格式：

```
until condition
do
    command
done
```

条件可为任意测试条件，测试发生在循环末尾，因此循环至少执行一次—请注意这一点。

---

case

Shell case语句为多选择语句。可以用case语句匹配一个值与一个模式，如果匹配成功，执行相匹配的命令。case语句格式如下：

```
case 值 in
模式1)
    command1
    command2
    ...
    commandN
    ;;
模式2)
    command1
    command2
    ...
    commandN
    ;;
esac
```

case工作方式如上所示。取值后面必须为单词in，每一模式必须以右括号结束。取值可以为变量或常数。匹配发现取值符合某一模式后，其间所有命令开始执行直至 ;;。
取值将检测匹配的每一个模式。一旦模式匹配，则执行完匹配模式相应命令后不再继续其他模式。如果无一匹配模式，使用星号 * 捕获该值，再执行后面的命令。
下面的脚本提示输入1到4，与每一种模式进行匹配：

```
echo '输入 1 到 4 之间的数字:'
echo '你输入的数字为:'
read aNum
case $aNum in
    1)  echo '你选择了 1'
    ;;
    2)  echo '你选择了 2'
    ;;
    3)  echo '你选择了 3'
    ;;
    4)  echo '你选择了 4'
    ;;
    *)  echo '你没有输入 1 到 4 之间的数字'
    ;;
esac
```

输入不同的内容，会有不同的结果，例如：

```
输入 1 到 4 之间的数字:
你输入的数字为:
3
你选择了 3
```

跳出循环

在循环过程中，有时候需要在未达到循环结束条件时强制跳出循环，Shell使用两个命令来实现该功能：break和continue。

break 命令

break命令允许跳出所有循环（终止执行后面的所有循环）。
下面的例子中，脚本进入死循环直至用户输入数字大于5。要跳出这个循环，返回到shell提示符下，需要使用break命令。

```
#!/bin/bash
while :
do
    echo -n "输入 1 到 5 之间的数字:"
    read aNum
    case $aNum in
        1|2|3|4|5) echo "你输入的数字为 $aNum!"
        ;;
        *) echo "你输入的数字不是 1 到 5 之间的! 游戏结束"
            break
        ;;
    esac
done
```

执行以上代码，输出结果为：

```
输入 1 到 5 之间的数字:3
你输入的数字为 3!
输入 1 到 5 之间的数字:7
你输入的数字不是 1 到 5 之间的! 游戏结束
```

continue

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。
对上面的例子进行修改：

```
#!/bin/bash
while :
do
    echo -n "输入 1 到 5 之间的数字: "
    read aNum
    case $aNum in
```

```
        1|2|3|4|5) echo "你输入的数字为 $aNum!"
        ;;
        *) echo "你输入的数字不是 1 到 5 之间的!"
            continue
            echo "游戏结束"
        ;;
    esac
done
```

运行代码发现，当输入大于5的数字时，该例中的循环不会结束，语句 echo “Game is over!” 永远不会被执行。

---

esac

case的语法和C family语言差别很大，它需要一个esac（就是case反过来）作为结束标记，每个case分支用右圆括号，用两个分号表示break。

10. Shell 函数

10. Shell 函数

linux shell 可以用户定义函数，然后在shell脚本中可以随便调用。

格式:

```
[ function ] funname [()]
{
    action;
    [return int;]
}
```

说明:

- 可以带function fun() 定义，也可以直接fun() 定义,不带任何参数。
- 参数返回，可以显示加 return 返回，若不加，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)

实例1（无return）:

```
#!/bin/bash
# author:Asa Guo

myFun(){
    echo "This is my first shell function"
```

```
}
echo "-----function start-----"
myFun
echo "-----function start-----"
```

输出：

```
asa@asa-virtual-machine:~$ ./printf.sh
-----func start-----
This is my first shell func
-----func start-----
```

实例2（有return）：

```
#!/bin/bash
# author:Asa Guo

funWithReturn(){
    echo "Two nums add each other..."
    echo "Input the first num1:"
    read num1
    echo "Input the second num2:"
    read num2
    echo "Two nums are $num1 and $num2"
    return $(($num1+$num2))
}

funWithReturn
echo "num1 + num2 = $?"
```

输出：

```
asa@asa-virtual-machine:~$ ./printf.sh
Two nums add each other...
Input the first num1:
1
Input the second num2:
5
Two nums are 1 and 5 !
num1 + num2 = 6
```

函数返回值在调用该函数后通过 $? 来获得。

注意：所有函数在使用前必须定义。这意味着必须将函数放在脚本开始部分，直至shell解释器首次发现它时，才可以使用。调用函数仅使用其函数名即可。

函数参数

在Shell中，调用函数时可以向其传递参数。在函数体内部，通过 $n 的形式来获取参数的值，例如，$1表示第一个参数，$2表示第二个参数…

实例：

```
#!/bin/bash
# author:Asa Guo

funWithParam(){
    echo "The first param is $1"
    echo "The second param is $2"
    echo "The tenth param is $10"
    echo "The tenth param is ${10}"
    echo "The eleventh param is ${11}"
    echo "Total count is $# "
    echo "A string of all params is $*"
}
funWithParam 1 2 3 4 5 6 7 8 9 34 73
```

输出：

```
asa@asa-virtual-machine:~$ ./printf.sh
first param is 1
second param is 2
tenth param is 10
tenth param is 34
eleventh param is 73
Total count is 11
A string of all params is 1 2 3 4 5 6 7 8 9 34 73
```

注意：

$10 不能获取第十个参数，获取第十个参数需要${10}。当n>=10时，需要使用${n}来获取参数。

几个处理参数的特殊字符：

参数处理	说明
$#	传递到脚本的参数个数
$*	以一个单字符串显示所有向脚本传递的参数
$$	脚本运行的当前进程ID号
$!	后台运行的最后一个进程的ID号
$@	与$*相同，但是使用时加引号，并在引号中返回每个参数。
$-	显示Shell使用的当前选项，与set命令功能相同。
$?	显示最后命令的退出状态。0表示没有错误，其他任何值表明有错误。

11. Shell 输入、输出重定向.md

11. Shell 输入/输出重定向

大多数 UNIX 系统命令从你的终端接受输入并将所产生的输出发送回到您的终端。一个命令通常从一个叫标准输入的地方读取输入，默认情况下，这恰好是你的终端。同样，一个命令通常将其输出写入到标准输出，默认情况下，这也是你的终端。

命令	说明
command > file	将输出重定向到 file。
command < file	将输入重定向到 file。
command >> file	将输出以追加的方式重定向到 file。
n > file	将文件描述符为 n 的文件重定向到 file。
n >> file	将文件描述符为 n 的文件以追加的方式重定向到 file。
n >& m	将输出文件 m 和 n 合并。
n <& m	将输入文件 m 和 n 合并。
<< tag	将开始标记 tag 和结束标记 tag 之间的内容作为输入。

> 需要注意的是文件描述符 *0* 通常是标准输入（*STDIN*），*1* 是标准输出（*STDOUT*），*2* 是标准错误输出（*STDERR*）。

输出重定向

重定向一般通过在命令间插入特定的符号来实现。特别的，这些符号的语法如下所示：

```
1. command1 > file1
```

上面这个命令执行command1然后将输出的内容存入file1。

注意任何file1内的已经存在的内容将被新内容替代。如果要将新内容添加在文件末尾，请使用>>操作符。

实例

执行下面的 who 命令，它将命令的完整的输出重定向在用户文件中(users):

```
$ who > users
```

执行后，并没有在终端输出信息，这是因为输出已被从默认的标准输出设备（终端）重定向到指定的文件。
你可以使用 cat 命令查看文件内容:

```
$ cat users
_mbsetupuser console  Oct 31 17:35
tianqixin    console  Oct 31 17:35
tianqixin    ttys000  Dec  1 11:33
```

输出重定向会覆盖文件内容，请看下面的例子:

```
$ echo "Hello world" > users
$ cat users
Hello world
```

如果不希望文件内容被覆盖，可以使用 >> 追加到文件末尾，例如:

```
$ echo "Hello world" >> users
$ cat users
Hello world
Hello world
```

输入重定向

和输出重定向一样，Unix 命令也可以从文件获取输入，语法为:

```
command1 < file1
```

这样，本来需要从键盘获取输入的命令会转移到文件读取内容。
注意:输出重定向是大于号(>)，输入重定向是小于号(<)。

实例

接着以上实例，我们需要统计 users 文件的行数,执行以下命令：

```
$ wc -l users
       2 users
```

也可以将输入重定向到 users 文件：

```
$  wc -l < users
       2
```

注意：上面两个例子的结果不同：第一个例子，会输出文件名；第二个不会，因为它仅仅知道从标准输入读取内容。

```
command1 < infile > outfile
```

同时替换输入和输出，执行command1，从文件infile读取内容，然后将输出写入到outfile中。

重定向深入讲解

一般情况下，每个 Unix/Linux 命令运行时都会打开三个文件：

- 标准输入文件(stdin)：stdin的文件描述符为0，Unix程序默认从stdin读取数据。
- 标准输出文件(stdout)：stdout 的文件描述符为1，Unix程序默认向stdout输出数据。
- 标准错误文件(stderr)：stderr的文件描述符为2，Unix程序会向stderr流中写入错误信息。
  默认情况下，command > file 将 stdout 重定向到 file，command < file 将stdin 重定向到 file。

如果希望 stderr 重定向到 file，可以这样写：

```
$ command 2 > file
```

如果希望 stderr 追加到 file 文件末尾，可以这样写：

```
$ command 2 >> file
```

2 表示标准错误文件(stderr)。
如果希望将 stdout 和 stderr 合并后重定向到 file，可以这样写：

```
$ command > file 2>&1
```

或者

```
$ command >> file 2>&1
```

如果希望对 stdin 和 stdout 都重定向，可以这样写：

```
$ command < file1 >file2
```

command 命令将 stdin 重定向到 file1，将 stdout 重定向到 file2。

Here Document

Here Document 是 Shell 中的一种特殊的重定向方式，用来将输入重定向到一个交互式 Shell 脚本或程序。
它的基本的形式如下：

```
command << delimiter
    document
delimiter
```

它的作用是将两个 delimiter 之间的内容(document) 作为输入传递给 command。

> 注意：
>
> - 结尾的*delimiter* 一定要顶格写，前面不能有任何字符，后面也不能有任何字符，包括空格和 *tab* 缩进。
> - 开始的*delimiter*前后的空格会被忽略掉。

实例

在命令行中通过 wc -l 命令计算 Here Document 的行数：

```
$ wc -l << EOF
    Hello
    World
    Tom
EOF
3          # 输出结果为 3 行
```

我们也可以将 Here Document 用在脚本中，例如：

```
#!/bin/bash
# author:Asa

cat << EOF
    Hello
    World
    Tom
EOF
```

执行以上脚本，输出结果：

```
    Hello
    World
    Tom
```

/dev/null 文件

如果希望执行某个命令，但又不希望在屏幕上显示输出结果，那么可以将输出重定向到 /dev/null：

```
$ command > /dev/null
```

/dev/null 是一个特殊的文件，写入到它的内容都会被丢弃；如果尝试从该文件读取内容，那么什么也读不到。但是 /dev/null 文件非常有用，将命令的输出重定向到它，会起到"禁止输出"的效果。
如果希望屏蔽 stdout 和 stderr，可以这样写：

```
$ command > /dev/null 2>&1
```

> 注意：*0* 是标准输入（*STDIN*），*1* 是标准输出（*STDOUT*），*2* 是标准错误输出（*STDERR*）。

12. Shell 文件包含

12. Shell 文件包含

和其他语言一样，Shell 也可以包含外部脚本。这样可以很方便的封装一些公用的代码作为一个独立的文件。
Shell 文件包含的语法格式如下：

```
. filename   # 注意点号(.)和文件名中间有一空格
或
source filename
```

再次声明:

点号(.)和文件名中间有一个空格

实例

创建两个 shell 脚本文件。
test1.sh:

```
#!/bin/bash
# author: Asa Guo

url="http://www.baidu.com"
```

test2.sh:

```
#!/bin/bash
# author: Asa Guo

#方法1：使用点号(.)来引用test1.sh
. ./test1.sh

#方法2：使用source命令
# source ./test1.sh

```

```
10. echo "百度：$url"
```

运行test2.sh:

```
1. $ chmod +x test2.sh
2. $ ./test2.sh
3. 百度：http://www.baidu.com
```

> 注：被包含的文件 *test1.sh* 不需要可执行权限。

13. Bash let 命令

14. Bash let 命令

let 命令是 BASH 中用于计算的工具，用于执行一个或多个表达式，变量计算中不需要加上 $ 来表示变量。如果表达式中包含了空格或其他特殊字符，则必须引起来。

语法格式

```
let arg [arg ...]
```

参数说明:

```
arg：要执行的表达式
```

实例:

```
#!/bin/bash
# author:Asa Guo

let no++        # 自增
let no--        # 自减

let no+=10    # 与下式相同
let no=no+10

let no-=20    # 与下式相同
let no=no-20
```

加减运算:

```
#!/bin/bash

let a=5+4
```

```
let b=9-3
echo $a $b
```

输出：

```
9 6
```

14. That’s all

14. That’s all

That’s all.