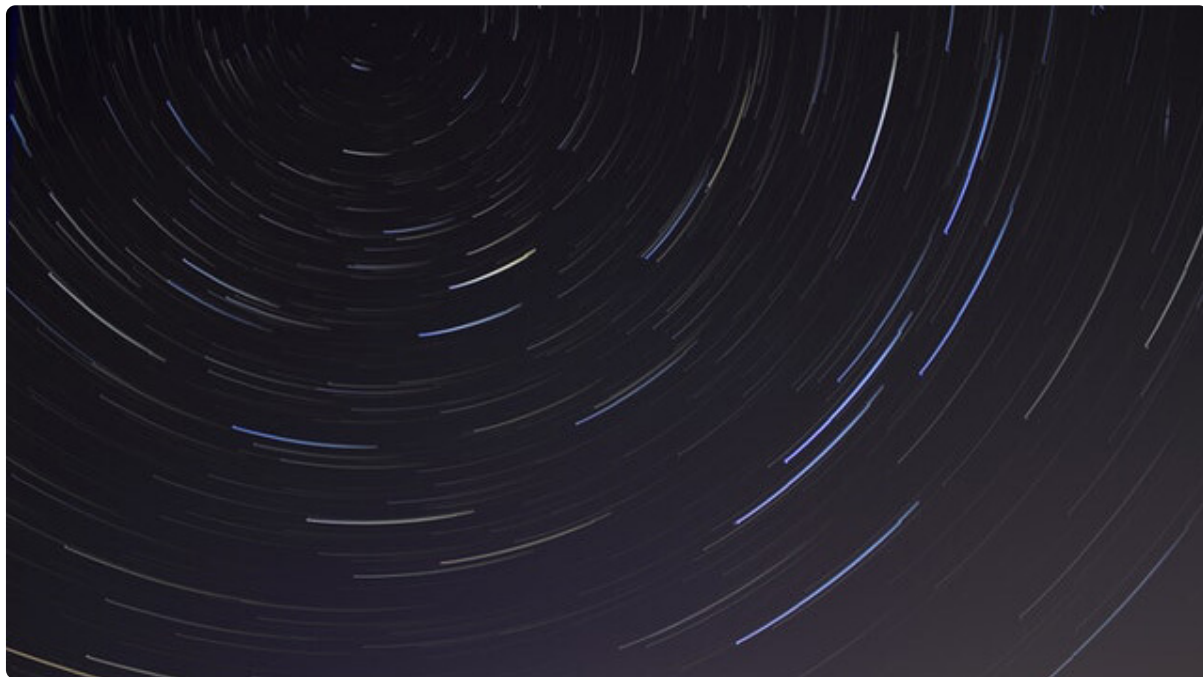


## 45 一朝Shell函数倾，斗转星移任我行

更新时间：2019-08-14 10:06:27



“

书是人类进步的阶梯。

——高尔基

”

### 内容简介

1. 前言
2. 函数的作用
3. 函数的定义
4. 传递参数
5. 返回值
6. 变量作用范围
7. 重载命令
8. 函数的设计
9. 总结

### 1. 前言

上一课 [带你玩转Linux和Shell编程 | 第五部分第六课：循环往复，Shell开路](#) 中，我们学习了 Shell 的循环语句。

这一课我们来学习对于大多数编程语言来说很重要的内容：函数。

### 2. 函数的作用

函数到底是什么呢？

以前在学校里学习数学的时候，也学过函数。例如：

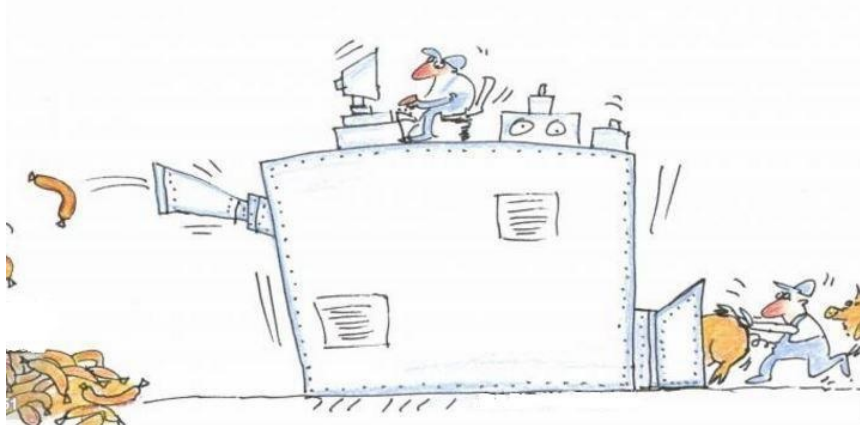
$$y = 2x + 1$$

这个公式也被称为一个函数，对于每一个给定的  $x$  值，都会有一个  $y$  值被计算出来，就是 2 倍的  $x$ ，加上 1。

编程的基础是数学，因此程序中的函数也有点数学中函数的味道：给它一定输入，它会为你产出一定的东西。

函数在英语中是 **function**，表示“功能，作用”的意思。因此，你可以这么理解：“函数是实现一定功能的代码块”。

我们可以把函数比作一个香肠制造机，在输入那一头你把猪装进去，输出那一头就出来香肠了。



函数还是重用代码（**reuse code**）的很好方式。

为什么这么说呢？

因为到目前为止，我们写的所有 **Shell** 文件，都是定义一些变量，然后执行一些命令，这些命令还都是逐行依次执行。

目前我们写的例子程序都比较短小，因此看不出有什么问题。假如这个文件的内容一多，达到好几万行甚至更多，而且有不少重复的内容，那么要维护这样一个庞杂的文件就极为困难。

这时候，函数就可以出马帮我们解围。因为函数可以把一块代码包裹起来，使之成为一个整体，完成某些任务。而且，我们在程序的其它地方，还可以多次使用这块代码。

这样，我们的程序就会变得有条理，也没有那么多重复的相似代码。你可以把函数想象成 **Shell** 脚本里的小脚本。

### 3. 函数的定义

说了这么多，也许还是不太好理解，我们不如实际来使用一下函数。

如果你学过其它编程语言，比如 **C** 语言，**Java**，**C++**等等，那么其实 **Shell** 中的函数机理是与之类似的。但是，**Shell** 比较“任性”，它的函数形式和主流编程语言有不少区别。

定义（或创建）**Shell** 函数是很容易的。有两种方式：

```
函数名(){  
    函数体  
}
```

或：

```
function 函数名 {  
    函数体  
}
```

这两种方式都是可行的。看你个人喜好用哪一种方式。我个人喜欢第一种方式，因为 `()` 比 `function` 少好多个字符。要知道，程序员是要懂得偷懒的，少输入一点就节省时间。

函数名后面跟着的圆括号里不加任何参数：这一点与主流编程语言很不相同。`C` 语言，`Java`，`C++` 等语言中，函数的圆括号中是可以放置参数的（也就是函数的一部分输入），但是 **Shell** 中的函数的圆括号里不能放置参数。

函数的完整定义必须置于函数的调用之前。

我们通过一个小例子来加深理解：

首先，用 `Vim` 或其他文本编辑器来创建一个文件，叫作 `function.sh`：

```
vim function.sh
```

然后在里面输入以下内容：

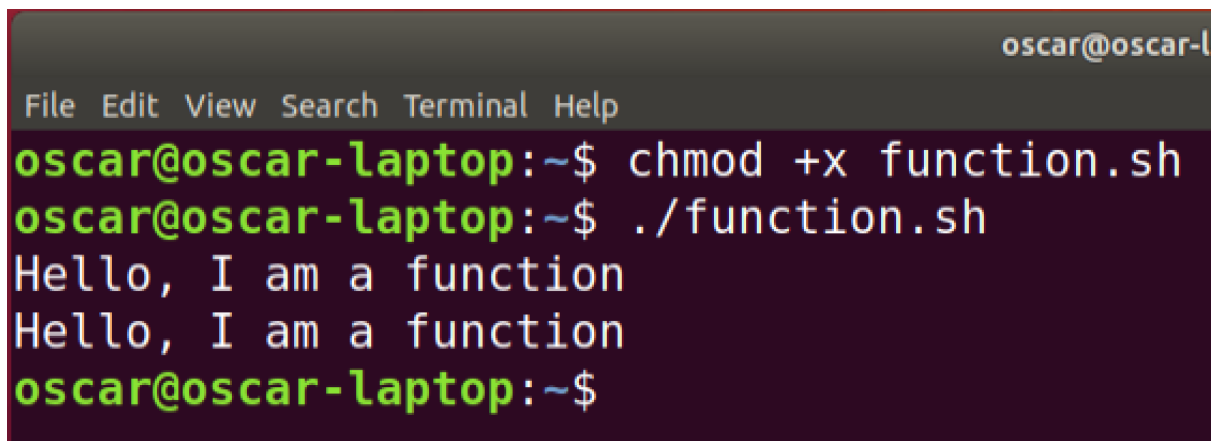
```
#!/bin/bash

print_something () {
    echo "Hello, I am a function"
}

print_something
print_something
```

“Hello, I am a function” 是英语“你好，我是一个函数”的意思。

运行：

A terminal window with a dark background and light green text. The prompt is 'oscar@oscar-laptop:~\$'. The user enters 'chmod +x function.sh' and then './function.sh'. The output shows 'Hello, I am a function' printed twice. The prompt returns to 'oscar@oscar-laptop:~\$'.

```
oscar@oscar-laptop:~$ chmod +x function.sh
oscar@oscar-laptop:~$ ./function.sh
Hello, I am a function
Hello, I am a function
oscar@oscar-laptop:~$
```

可以看到程序打印了两次 “Hello, I am a function”。

我们逐行来解释这个程序：

- 第 3 行：我们开始了一个函数的定义，给它起了一个名字，叫做 `print_something`（`print` 是英语“打印”的意思，`something` 是英语“某些东西”的意思）。
- 第 4 行：在大括号 `{}` 中，我们可以写入多个命令。
- 第 7 和 8 行：在函数定义之后，我们就可以调用它任意多次。这里连续调用了两次。

#### 4. 传递参数

我们上面的函数并没有处理参数，有时候我们希望函数能为我们处理一些参数，然后输出一些结果。

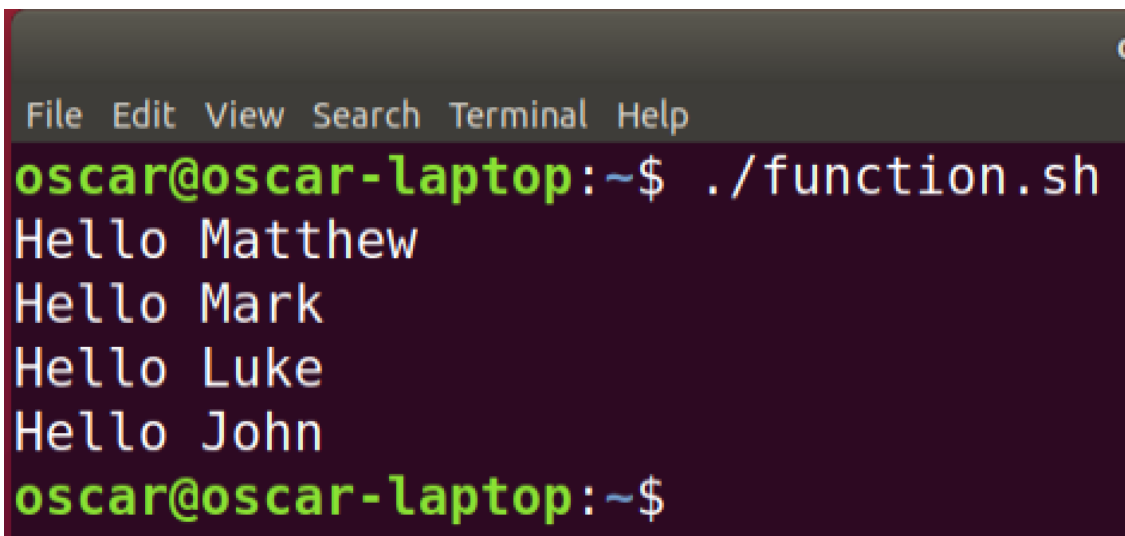
在 **Shell** 函数中，我们给它传递参数的方式其实很像给 **Shell** 脚本传递命令行参数。我们把参数直接置于函数名字后面，然后就像我们之前 **Shell** 脚本的参数那样：**\$1**，**\$2**，**\$3** 等等。

我们来看一个例子：

```
#!/bin/bash

print_something () {
    echo Hello $1
}

print_something Matthew
print_something Mark
print_something Luke
print_something John
```

A terminal window with a dark background and light green text. The menu bar at the top shows 'File Edit View Search Terminal Help'. The prompt is 'oscar@oscar-laptop:~\$'. The user has entered './function.sh'. The output shows four lines: 'Hello Matthew', 'Hello Mark', 'Hello Luke', and 'Hello John'. The prompt returns to 'oscar@oscar-laptop:~\$'.

## 5. 返回值

大多数主流编程语言都有函数返回值的概念，可以让函数回传一些数据。

**Shell** 的函数却没办法做到。但是 **Shell** 的函数可以返回一个状态，有点类似一个程序或命令退出时会有一个退出状态，表明是否成功。

**Shell** 函数要返回状态，也用 **return** 这个关键字（**return** 是英语“返回”的意思）。

来看一个例子：

```
#!/bin/bash

print_something () {
    echo Hello $1
    return 1
}

print_something Luke
print_something John
echo Return value of previous function is $?
```

- 第 5 行：返回的状态不一定要是被硬编码的（比如上例中的 **1**），也可以是一个变量。
- 第 10 行：变量 **\$?** 包含前一次被运行的命令或函数的返回状态。

运行：

```
oscar@
File Edit View Search Terminal Help
oscar@oscar-laptop:~$ ./function.sh
Hello Luke
Hello John
Return value of previous function is 1
oscar@oscar-laptop:~$
```

“Return value of previous function is” 表示“上一个函数的返回值是”。

一般来说，返回状态 0 表示一切顺利；一个非零值表示有错误。

如果你实在想要函数返回一个数值（比如说一个计算的值），那么你也可以用命令的执行结果。

如下：

```
#!/bin/bash

lines_in_file () {
    cat $1 | wc -l
}

line_num=$(lines_in_file $1)

echo The file $1 has $line_num lines
```

运行：

```
oscar@oscar-laptop: ~
File Edit View Search Terminal Help
oscar@oscar-laptop:~$ ./function.sh myFile
The file myFile has 13 lines
oscar@oscar-laptop:~$ cat myFile
Hello Linuxers !

I just write some lines in Vim for filling the screen.
I just write some lines in Vim for filling the screen.
I just write some lines in Vim for filling the screen.
I just write some lines in Vim for filling the screen.
I just write some lines in Vim for filling the screen.
I just write some lines in Vim for filling the screen.
I just write some lines in Vim for filling the screen.
I just write some lines in Vim for filling the screen.
You see, once we switch to Insert Mode, writing things is not complicated.
:P
Really ?
oscar@oscar-laptop:~$
```

上面的脚本就是输出参数 1 所代表的文件的行数。因为 myFile 这个文件有 13 行。所以输出：“The file myFile has 13 lines”，表示“myFile 这个文件有 13 行”。

## 6. 变量作用范围

变量的作用范围意味着一个 **Shell** 脚本的哪些部分可以访问到这个变量。

默认来说，一个变量是“全局的”（**global**），意味着在脚本的任何地方都可以访问它。

我们也可以创建局部（**local**）变量。当我们在函数中创建局部变量时，这个变量就只能在这个函数中被访问。

要定义一个局部变量，我们只要第一次给这个变量赋值时在变量名前加上关键字 **local** 即可（**local** 是英语“本地的”的意思）。

定义局部变量有一个好处，就是可以防止被脚本的其它地方的代码意外改变数值。

来看一个例子：

```
#!/bin/bash

local_global () {
    local var1='local 1'
    echo Inside function: var1 is $var1 : var2 is $var2
    var1='changed again' # 这里的 var1 是函数中定义的局部变量
    var2='2 changed again' # 这里的 var2 是函数外定义的全局变量
}

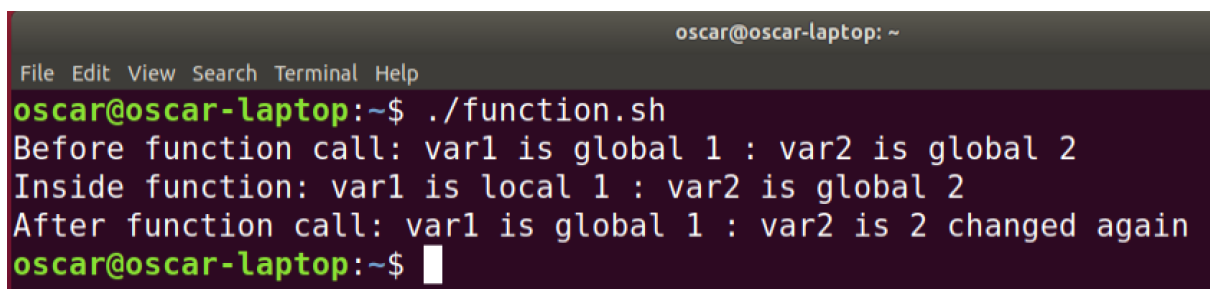
var1='global 1'
var2='global 2'

echo Before function call: var1 is $var1 : var2 is $var2

local_global

echo After function call: var1 is $var1 : var2 is $var2
```

运行：



```
oscar@oscar-laptop: ~
File Edit View Search Terminal Help
oscar@oscar-laptop:~$ ./function.sh
Before function call: var1 is global 1 : var2 is global 2
Inside function: var1 is local 1 : var2 is global 2
After function call: var1 is global 1 : var2 is 2 changed again
oscar@oscar-laptop:~$
```

- “Before function call” 表示“在函数调用前”。
- “Inside function” 表示“在函数中”。
- “After function call” 表示“在函数调用后”。

在函数中，尽量用局部变量。只有实在不行才用全局变量，毕竟全局变量不太安全。

## 7. 重载命令

我们可以用函数来实现命令的重载，也就是说把函数的名字取成与我们通常在命令行用的命令相同的名字。

例如，也许我们每次在脚本中调用 **ls** 命令时，其实是想要实现 **ls -lh** 的效果。那么我们可以这么做：

```
#!/bin/bash

ls(){
    command ls -lh
}

ls
```

第 4 行如果没有 `command` 这个关键字（`command` 是英语“命令”的意思），那么程序会陷入无限循环。  
如果你不小心忘了 `command` 关键字而陷入无限循环，可以用 `Ctrl + c` 的组合快捷键来停止程序。

运行：

```
oscar@oscar-laptop: ~
File Edit View Search Terminal Help
oscar@oscar-laptop:~$ ./function.sh
total 502M
-rw-r--r--  1 oscar oscar   20 May 10 07:15 chinese.txt
drwxr-xr-x  3 oscar oscar  4.0K May 23 08:01 compression
-rwxr-xr-x  1 oscar oscar  220 Jun  1 16:56 condition.sh
-rwxr-xr-x  1 oscar oscar  220 Jun  2 09:27 condition.sh-copy
-rw-r--r--  1 oscar oscar    0 May 19 13:02 date
-rw-r--r--  1 oscar oscar 292M Apr 27 13:19 debian-9.9.0-amd64-netinst.iso
drwxr-xr-x  2 oscar oscar  4.0K May  1 10:48 Desktop
drwxr-xr-x  2 oscar oscar  4.0K May  1 10:48 Documents
drwxr-xr-x  2 oscar oscar  4.0K May  1 10:48 Downloads
-rwxr-xr-x  1 oscar oscar  63M May 14 21:29 emacs-26.2-copy.tar.gz
-rwxrwxrwx  1 oscar oscar  63M May 14 21:26 emacs-26.2.tar.gz
-rw-r--r--  1 oscar oscar   51 May 11 17:12 errors.log
-rw-r--r--  1 oscar oscar  31K May 16 19:17 find_log
-rwxr-xr-x  1 oscar oscar   46 Jun  2 10:15 function.sh
-rw-r--r--  1 oscar oscar  15M May 16 22:36 grep_log
-rwxr-xr-x  1 oscar oscar  8.8K Jun  1 16:26 helloworld
-rw-r--r--  1 oscar oscar  204 Jun  1 16:25 helloworld.cpp
drwxr-xr-x 11 oscar oscar  4.0K May 29 13:57 htop-2.2.0
-rwxrwxrwx  1 oscar oscar 301K May 29 13:47 htop-2.2.0.tar.gz
-rwxr-xr-x  1 oscar oscar   55 Jun  2 09:35 loop.sh
-rwxr-xr-x  1 oscar oscar   67 Jun  2 09:27 loop.sh-copy
drwxr-xr-x  2 oscar oscar  4.0K May  1 10:48 Music
```

## 8. 函数的设计

我们已经看到：在 **Shell** 中定义函数是很简单的。但是要定义容易维护和易于扩展的函数却需要经验和时间。

有时候，好的设计意味着更少的代码；有时候意味着需求变更时最少的改动；有时候意味着不容易引起错误。

如果一个任务需要被执行多次，那么把它放到一个函数里是不错的选择。

设计模式（**Design Pattern**）中有一个原则是“单一职责原则”（**Single Responsibility Principle**）。这种原则也适用于函数的设计，尽量不要让一个函数执行很多任务。

## 9. 总结

函数使我们可以轻松地重用代码，使程序更有条理，更易理解和扩展；

我们有两种定义函数的方式：`function 函数名` 或 `函数名()`；

关键字 **return** 可以用于返回状态值；

关键字 **local** 可以用于定义局部变量；

关键字 **command** 使得函数可以重载命令。

今天的课就到这里，一起加油吧！

}