12 函数基础

函数基础

与大多数编程语言一样，Shell同样支持函数功能。

函数的优点：封装、复用、可读性

一、函数的基本语法

函数是Shell脚本中自定义的**命令序列**，一般来说函数应该设置有**返回值**（正确返回0，错误返回非0。对于错误返回，可以定义返回其他非0正值来细化错误）。

1.1 函数的定义及调用

定义Shell函数的语法格式如下。

```shell
函数名(){
    函数体
}
```

或

```shell
function 函数名 {
    函数体
}
```

关键字 `function` 表示定义一个函数，可以省略，其后是函数名，两个大括号之间是函数体。

创建的函数可以在别的脚本中调用。**调用函数时只要调用函数名即可**。

案例1：定义函数

```shell
[root@Shell ~]# vi function_define.sh
#! /bin/bash
#定义函数
function sayHello(){
    echo Hello
}
#调用函数
sayHello
[root@Shell ~]# . function_define.sh
Hello
```

案例2：统计文件行数

```shell
[root@Shell ~]# vi function_define2.sh
#! /bin/bash
FILE=/etc/passwd

function countLine(){
    i=0
    while read line
    do
        let ++i
    done < $FILE
    echo "$FILE have $i lines"
}
countLine
[root@Shell ~]# . function_define2.sh
/etc/passwd have 21 lines
```

1.2 函数的返回值

Shell中的函数的**返回值**又叫**函数的退出状态码**，实际上是一种通信方式。

函数可以使用“返回值”的方式来给调用者反馈信息（使用 `return` 关键字）。

如果函数体中**没有 `return` 语句**，那么使用**默认的退出状态码**，也就是**最后一条命令的退出状态码**，即 `return $?` 。

Shell 函数的返回值只能是**一个介于 0~255 之间的整数**，其中只有 `0` 表示成功，其它值都表示失败。

特殊变量 `$?` 也是获取函数返回值的主要方式。

案例3：获取返回值

Shell 复制代码

```
[root@Shell ~]# vi function_return.sh
#! /bin/bash
#定义函数
function sayHello(){
    echo Hello
    return 999
}
#调用函数
sayHello
#获取返回值
echo $?
[root@Shell ~]# . function_return.sh
Hello
231
#设置返回值为999，得到的也不是999
```

案例4：检测文件是否存在

Shell 复制代码

```
[root@Shell ~]# vi function_return2.sh
#! /bin/bash
FILE=/etc/passwd1

function checkFile(){
    if [ -f $FILE ]; then
        retrun 0
    else
        return 1
    fi
}
checkFile
if [ $? -q 0 ]; then
    echo "$FILE已存在"
else
    echo "$FILE不存在"
fi
[root@Shell ~]# . function_return2.sh
-bash: [: -q: binary operator expected
/etc/passwd1不存在
```

⚠️ 函数返回值的使用方式：

执行函数后，通过 `$?` 获取返回值，然后再跟进返回值执行相关操作。

💬 *有人可能会疑惑，既然 `return` 表示退出状态，那么该如何得到函数的处理结果呢?*

这个问题有两种解决方案：

- *一种是借助全局变量，将得到的结果赋值给全局变量。*
- *一种是在函数内部使用 `echo` 、 `printf` 命令将结果输出，在函数外部使用命令替换捕获结果。*

案例5：获取返回值

Shell | 复制代码

```
[root@Shell ~]# vi function_return3.sh
#!/bin/bash
fun(){
read -p "enter num: " num
echo $[2*$num]
}
fun
[root@Shell ~]# . function_return3.sh
enter num: 3
6
[root@Shell ~]# vi function_return4.sh
#!/bin/bash
fun(){
read -p "enter num: " num
echo $[2*$num]
}
result=`fun` #把函数的执行结果赋给变量
echo "fun return value: $result "
[root@Shell ~]# . function_return4.sh
enter num: 3
fun return value: 6
```

1.3 函数的参数

前面的案例使用了固定的 `FILE` 变量，这会造成想要判断不同的文件是否存在时，需要修改脚本中的 `FILE` 变量—也就是要对代码本身的内容进行修改，这也是典型的代码和数据没有分开而导致的问题。

事实上，可以通过定义带参数的函数解决这个问题。

在Shell中，**向函数传递参数**也是使用**位置参数**来实现的。

案例6：函数的参数

```shell
[root@Shell ~]# vi function_param1.sh
#!/bin/bash
funParams(){
    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 $* !"
}
funParams 1 2 3 4 5 6 7 8 9 34 73
[root@Shell ~]# . function_param1.sh
第一个参数为 1 !
第二个参数为 2 !
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 !
```

注意，当 `n>=10` 时，需要使用 `${n}` 来获取参数。

复习下常用的位置参数。

参数处理	说明
`$#`	传递到脚本或函数的参数个数
`$*`	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。

⚠️ 注意！不要把函数参数和脚本的参数混淆了，虽然它们的语法相同。

案例7：函数参数与脚本参数混用

Shell | 复制代码

```
[root@Shell ~]# vi function_param2.sh
#!/bin/bash
funParams(){
    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 $* !"
}
funParams 1 2 3 $1 5 6 7 8 9 34 73
[root@Shell ~]# . function_param2.sh 10
第一个参数为 1 !
第二个参数为 2 !
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 10 5 6 7 8 9 34 73 !
```

二、综合案例：检测系统服务状态

使用函数检查服务是否启动的案例脚本。

要点：函数体中的 `$@` 读取函数的所有参数，函数调用中的 `$@` 读取脚本的所有参数。

```shell
[root@Shell ~]# vi check_service.sh
#!/bin/bash
date_time=$(date +'%Y-%m-%dT%H:%M:%S%z')

function check_services() {
    for i in "$@"
    do
        if systemctl --quiet is-active ${i}.service; then
            echo "[$date_time]: service $i is active"
        else
            echo "[$date_time]: service $i is not active"
        fi
    done
}

check_services $@
[root@Shell ~]# . check_service.sh sshd
[2022-02-06T22:24:06+0800]: service sshd is active
[root@Shell ~]# . check_service.sh sshd vsftpd httpd
[2022-02-06T22:24:15+0800]: service sshd is active
[2022-02-06T22:24:15+0800]: service vsftpd is not active
[2022-02-06T22:24:15+0800]: service httpd is not active
```

三、系统函数库

在前面的案例中，定义的函数和函数的调用都在同一个文件中，这样使用起来很不方便。**分离函数定义和执行函数语句的脚本文件**更方便。对某些很常用的功能，必须考虑将其独立出来，集中存放在一些**独立的文件**中，这些文件就称为“函数库”。这么做的好处是在后期开发的过程中可以**直接利用这些库函数**写出高质量的代码。

很多Linux发行版中都有 `/etc/init.d` 目录，这是系统中放置所有**开机启动脚本的目录**，这些开机脚本在脚本开始运行时都会加载 **`/etc/init.d/functions`** 或 **`/etc/rc.d/init.d/functions`** **系统函数库**（实际上这两个函数库的内容是完全一样的）。

系统函数库中重要的函数如下。

函数	功能

`checkpid`	检查是否已存在pid，如果有一个存在，返回0（通过查看 `/proc` 目录）
`daemon`	启动某个服务。 `/etc/init.d` 目录部分脚本的start使用到这个
`killproc`	杀死某个进程。 `/etc/init.d` 目录部分脚本的stop使用到这个
`pidfileofproc`	寻找某个进程的pid
`pidofproc`	类似上面的，只是还查找了 `pidof` 命令
`status`	返回一个服务的状态
`action`	打印某个信息并执行给定的命令，它会根据命令执行的结果来调用success,failure方法
`strstr`	判断 `$1` 是否含有 `$2`
`confirm`	显示 " `Start service $1 (Y)es/(N)o/(C)ontinue? [Y]` "的提示信息，并返回选择结果

Shell | 复制代码

```
[root@Shell ~]# cat /etc/init.d/functions|grep status
# Returns LSB exit code for the 'status' action.
status() {
                echo $"Usage: status [-p pidfile] {program}"
                systemctl status ${0##*/}.service
                        echo $"${base} status unknown due to insufficient p
rivileges."
```

案例8：使用系统函数库

要点：先运行系统函数库文件，再调用函数。

```shell
# 直接调用系统函数中的函数会提示错误
[root@Shell ~]# status
-bash: status: command not found
# 运行系统函数库文件
[root@Shell ~]# . /etc/init.d/functions
# 运行系统函数库函数
[root@Shell ~]# status
Usage: status [-p pidfile] {program}
[root@Shell ~]# status sshd
sshd (pid  1429) is running...
# 编写脚本调用系统函数库
[root@Shell ~]# vi function_system.sh
#!/bin/bash
. /etc/init.d/functions
status sshd
[root@Shell ~]# . function_system.sh
sshd (pid  1429) is running...
```

由于Shell是一门面向过程的脚本型语言，而且用户主要是Linux系统管理人员，所以并没有非常活跃的社区，这也造成了Shell缺乏第三方函数库，所以在很多时候需要系统管理人员根据实际工作的需要**自行开发函数库**。

自定义函数库非常简单，将**自己所需函数**放在**自定义函数库脚本**中，**使用函数前先运行自定义函数库脚本，再调用函数**。当然还有一种更方便的方法——将函数代码追加到系统函数库文件中，即 `/etc/init.d/functions` 。

小结

- 函数的基本语法：返回值
- 带参数的函数：注意函数参数与脚本参数的区别
- 系统函数库：先运行函数库，再调用函数

课程目标

- 知识目标：熟练掌握函数的基本语法。
- 技能目标：能够根据实际需求编写函数。

课外拓展

- 进一步了解函数的应用场景。

参考资料

- `man bash` `/functions`
- 《Linux Shell核心编程指南》，丁明一，电子工业出版社