目 录

1 基本脚本函数

1.1 函数格式

函数的格式如下所示：

```
# name是函数名
# commands是组成函数的命令
function name {
    commands
}
```

函数的另一种格式如下：

```
name(){
    commands
}
```

1.2 使用函数

函数可以定义在除 shell 脚本第一行以外的任何位置，但是必须在函数定义处的后面使用函数。例子如下：

```
#!/bin/bash
count=1
echo "This line comes before the function definition"

function func1 {
    echo "This is an example of a function"
}

while [ $count -le 5 ]
do
    func1
    count=$[ $count + 1 ]
done

echo "This is the end of the loop"
# 在函数定义处前调用函数，将会报错
func2

function func2 {
    echo "This is an example of a function"
}
```

需要注意的是，如果重复定义相同函数名的函数，将使用最新的那个函数定义，例子如下所示：

```
#!/bin/bash
```

```
2
3     function func1 {
4         echo "This is the first definition of the function name"
5     }
6
7     func1
8
9     function func1 {
10        echo "This is a repeat of the same function name"
11    }
12
13    func1
```

2 函数返回值

2.1 return 命令

bash shell 将函数看作小型脚本，在默认情况下，函数的返回值是函数最后一条命令返回的退出状态。但是一般情况下，我们倾向于使用 return 命令以特定退出状态退出函数。例子如下所示：

```
1     #!/bin/bash
2     function db1 {
3         read -p "Enter a value: " value
4         echo "doubling the value"
5         return $[ $value * 2 ]
6     }
7
8     db1
9     # $?存放着函数的返回值
10    echo "The new value is $?"
```

需要注意的是，$? 的取值范围是 0 ～ 255，而且需要在函数完成后尽快提取返回值，否则在任何一条新的命令执行后，$? 将发生变化。

2.2 使用函数输出

类似于用反引号获得命令的输出，我们也可以使用反引号获得函数的输出，这种方法可以从函数获取任意类型的输出并给变量赋值。例子如下：

```
1     #!/bin/bash
2
3     function db1 {
4         read -p "Enter a value: " value
5         echo $[ $value * 2 ]
6     }
7
```

```
8     result=`db1`
9     echo "The new value is $result"
```

这个方法还可以返回浮点数和字符串值，能够比较灵活地从函数返回数据。

3 在函数中使用变量

3.1 向函数传递参数

bash shell 将函数看作小型脚本，所以可以用命令行参数的方法向函数传递参数。例子如下：

```
1   #!/bin/bash
2
3   function addem {
4       if [ $# -eq 0 ] || [ $# -gt 2 ]
5       then
6           echo "-1"
7       elif [ $# -eq 1 ]
8       then
9           echo $[ $1 + $1 ]
10      else
11          echo $[ $1 + $2 ]
12      fi
13  }
14
15  echo -n "Adding 10 and 15: "
16  value=`addem 10 15`
17  echo $value
18
19  echo -n "Let's try adding just one number: "
20  value=`addem 10`
21  echo $value
22
23  echo -n "Now trying adding no numbers: "
24  value=`addem`
25  echo $value
26
27  echo -n "Finally, try adding three numbers: "
28  value=`addem 10 15 20`
29  echo $value
```

需要注意的是，shell 脚本中函数的命令行参数和脚本的命令行参数是相互独立的，函数无法使用脚本的命令行参数。

3.2 在函数中处理变量

函数使用两种类型的变量：全局变量和局部变量。

全局变量是在 shell 脚本中处处有效的变量，默认情况下在脚本中定义的变量就是全局变量，不管是在主代码中定义还是函数中定义。例子如下：

```
#!/bin/bash

function db1 {
    value=$[ $value * 2 ]
}

read -p "Enter a value: " value
db1
echo "The new value is: $value"
```

函数内部使用的变量是局部变量，需要在变量声明前面冠以 local 关键字，如下所示：

```
local variable
```

局部变量的使用如下例所示：

```
#!/bin/bash

function func1 {
    local temp=$[ $value + 5 ]
    result=$[ $temp * 2 ]
}

temp=4
value=6

func1
echo "The result is $result"
if [ $temp -gt $value ]
then
    echo "temp is larger"
else
    echo "temp is smaller"
fi
```

4 数组变量与函数

4.1 向函数传递数组

如果试图将数组变量作为单个参数传递，是无法正常工作的，函数只会提取数组变量的第一个取值，如下例所示：

```
#!/bin/bash

function testit {
    echo "The parameters are: $@"
```

```
5       thisarray=$1
6       echo "The received array is ${thisarray[*]}"
7   }
8
9   myarray={1 2 3 4 5}
10  echo "The original array is: ${myarray[*]}"
11  testit $myarray
```

为了向函数传递数组，需要将数组变量拆分为单个元素，然后使用这些元素的值作为函数参数，函数内部再将这些参数重组委新数组变量，如下例所示：

```
1   #!/bin/bash
2
3   function testit {
4       local newarray
5       newarray={`echo "$@"`}
6       echo "The new array value is: ${newarray[*]}"
7   }
8
9   myarray={1 2 3 4 5}
10  echo "The original array is ${myarray[*]}"
11  # 向函数传递数组的正确方法
12  testit ${myarray[*]}
```

4.2 从函数返回数组

函数可以使用 echo 语句以恰当顺序输出数组各元素的值，然后脚本必须将这些数据重组为新数组变量，如下例所示：

```
1   #!/bin/bash
2
3   function arraydblr {
4       local origarray
5       local newarray
6       local elements
7       local i
8       origarray={`echo "$@"`}
9       newarray={`echo "$@"`}
10      elements=$[] $# - 1 ]
11      for (( i = 0; i <= $elements; i++ ))
12      do
13          newarray[$i]=$[ ${origarray[$i]} * 2 ]
14      done
15      echo ${newarray[*]}
16  }
17
18  myarray={1 2 3 4 5}
19  echo "The original array is: ${myarray[*]}"
20  arg1=`echo ${myarray[*]}`
21  result={`arraydblr $arg1`}
22  echo "The new array is: ${result[*]}"
```

5 函数递归

递归函数的一个经典示例如下所示：

```
#!/bin/bash

function factorial {
    if [ $1 -eq 1]
    then
        echo 1
    else
        local temp=$[ $1 - 1 ]
        local result=`factorial $temp`
        echo $[ $result * $temp]
    fi
}

read -p "Enter value: " value
result=`factorial $value`
echo "The factorial of $value is: $result"
```

6 创建库

bash shell 可以创建函数的库文件，然后在不同脚本中引用该库文件。

首先创建库文件，如下例所示：

```
# 库文件名为myfuncs
function addem {
    echo $[ $1 + $2 ]
}

function multem {
    echo $[ $1 * $2 ]
}

function divem {
    if [ $2 -ne 0 ]
    then
        echo $[ $1 / $2 ]
    else
        echo "-1"
    fi
}
```

下一步是将库文件 myfuncs 包含进需要调用库函数的脚本文件。可以使用 source 命令在 shell 脚本内部运行库文件脚本，这样脚本就可以使用这些函数。source 有一个短小的别名，称为点操作符。如果想在 shell 脚本中调用 myfuncs 库文件，只需要添加如下命令行：

```
. ./myfuncs
```

通过上述命令就可以使用库文件，如下例所示：

```
#!/bin/bash
. ./myfuncs

value1=10
value2=5
result1=`addem $value1 $value2`
result2=`multem $value1 $value2`
result3=`divem $value1 $value2`
echo "The result of adding them is: $result1"
echo "The result of multiplying them is: $result2"
echo "The result of dividing them is: $result3"
```

7 在.bashrc 文件中定义函数

可以在.bashrc 文件中定义函数，这样就可以在命令行中直接使用函数。可以直接在.bashrc 文件的末尾添加自定义函数，如下所示：

```
# .bashrc
# ...
function addem {
    echo ${ $1 + $2 }
}
```

该函数在下一次启动新 bash shell 时生效。这样，该函数就可以在系统任意位置使用了。

也可以将库文件的函数包含进.bashrc 脚本，如下所示：

```
# .bashrc
# ...
# 将/home/psd/libraries/myfuncs库文件包含进.bashrc文件中
. /home/psd/libraries/myfuncs
```

当在.bashrc 文件中定义函数以后，shell 脚本也可以使用.bashrc 文件中的函数。