子程序结构

内容提要

1. 理解在何种情况下使用函数
2. 掌握函数的定义和调用方法
3. 掌握结构化编程方法
4. 明确 return 和 exit 的区别
5. 重点体会位置参数和 shift 语句的使用
6. 回顾字符串截取操作和变量替换扩展的使用

函数简介

bash 提供了子程序调用的功能，即函数。通常在如下情况使用函数：

- 简化程序代码，实现脚本代码重用。一次定义多次调用。
- 实现结构化编程，将一个大型脚本的动作划分为多个组，每个组定义为一个函数，从而增强脚本的可读性。
- 为了加快任务的运行，管理员可以将常用的功能定义为多个函数并将其保存在一个文件中（类似其他语言的"模块"），然后在 ~/bashrc 或在命令行使用 source（.）命令调用这个文件，这样所定义的函数就被调入内存，从而加快运行速度。

函数的定义和调用

下面给出函数的定义方法：

传统风格的定义方法

```
function name {
       statement1
       statement2
       ..........
}
```

C语言风格的定义方法

```
name()
{
       statement1
       statement2
       ..........
}
```

'name' 就是函数的名称。函数要在调用之前定义。要调用这个函数可以直接呼叫 'name'。任何传递至函数的参数可以加在 'name' 的后面，在函数中可以使用位置参数（$1、$2 ...）来使用这些参数。为了提高程序通用性，通常使用 C 风格的函数定义方法，以便在 csh/tcsh 中也能使用。

函数使用举例

下面举几个使用函数的例子：

例1～例5 是对一个求和函数的功能逐步丰富的例子。

例1：

```
#!/bin/sh
# filename: sum_function.sh
# 定义函数
function sum {
  if [ $# -ne 1 ]; then
    echo "Usage: $0 <num>"
    return 1
  fi
  i=0; s=0
  while [ $i -lt $1 ]; do
   let "i=$i+1"; let "s=$s+$i"
  done
  echo sum\(1..$1\)=$s
}

# 调用函数
sum 10
sum            # 返回使用信息
sum 10 20      # 返回使用信息
```

return 和 **exit** 的区别

本例使用了 return 1 而非 exit 1，两者的区别在于：

1. return 退出函数的执行，返回到主程序调用函数之后继续执行
2. exit 退出当前脚本的执行

若将本例中的 return 1 改为 exit 1，则只能看到一行使用信息，因为在执行 sum 调用时就退出了脚本，根本没有执行 sum 10 20 调用。

例2：

```
#!/bin/bash
# filename: sum_function2.sh
# 定义函数
sum()
{
  if (($#>1)); then
    echo "Usage: $0 [<num>]"
    return 1
  fi
  # 将 $1 的值赋予变量 to
  # 当 $1 为空（即主程序没有使用位置参数调用本函数），将 100 赋予变量 to
  to=${1:-100}
  # 上面的语句与如下两条 if 语句均等价
  # if [ $# -eq 0 ]; then to=100; else to=$1; fi
  # if [ -z $1 ]; then to=100; else to=$1; fi
  ((i=0,s=0))
  while ((i<$to)); do ((i++,s+=i)); done
  echo sum\(1..$to\)=$s
}

# 调用函数
sum
sum 10
sum 10 20      # 返回使用信息
```

例3：

```
#!/bin/bash
# filename: sum_function3.sh
# 定义函数
sum()
{

  if (($# == 0)) ; then
     from=1; to=100
  elif (($# == 1)) ; then
     from=1; to=$1
  elif (($# == 2)) ; then
     from=$1; to=$2
  else
     echo "Usage: $0 [<from>] [<to>]"
     return 1
  fi

  # 注意： for 循环体内没有任何要执行的语句，要使用 : （: 表示空语句）
  for (( i=1,s=$from; i<$to; i++,s+=i )) do : ; done
  echo sum\($from..$to\)=$s
}

# 调用函数
sum
sum 10
sum 3 201
sum 10 20 30     # 返回使用信息
```

例4：

```
#!/bin/bash
# filename: sum_function4.sh
# 定义函数
sum()
{
  # 若位置参数大于1个，就对所有的位置参数求和
  if (($#>=2)) ; then
    s=0; sumstr=""            # 赋初值
    # echo  "$@" ,  $i  ,  $s  ,  $sumstr
    for i ; do
      ((s=s+i))               # 做整数累加
      sumstr=$sumstr+$i       # 做字符串连接
      shift
      # echo  "$@" ,  $i  ,  $s  ,  $sumstr
    done
    echo ${sumstr:1}=$s
  else
    echo "Usage: $0 <num1> <num2> ..."
    return 1
  fi
}

# 调用函数
sum 3 201
sum 10 20 30
sum 24 35 16 7 21
sum          # 输出使用信息
sum 10       # 输出使用信息
```

说明：

1. 在本例的 for 语句中省略了 in wordList 部分，相当于 in "$@"
2. 在本例中使用了 shift 语句，他的功能是将 $2赋给$1，$3赋给$2，$4赋给$3 ……
3. 下面列表给出执行 sum 24 35 16 7 21 函数调用的 for 循环每次执行时各个变量的值

	"$@"	**$i**	**$s**	**$sumstr**	说明
循环之前的值	24 35 16 7 21		0		

第1次循环后	35 16 7 21	24	24	+24	
第2次循环后	16 7 21	35	59	+24+35	
第3次循环后	7 21	16	75	+24+35+16	
第4次循环后	21	7	82	+24+35+16+7	
第5次循环后		21	103	+24+35+16+7+21	当前“$@”的值为空，所以不再进行下一次循环

因为退出循环后 $sumstr 的值为 +24+35+16+7+21，所以用 ${sumstr:1} 输出删除了最前面“+”的字符串。有关字符串操作的详细信息请参见 深入 Shell 变量操作 一节。

例5：

```
#!/bin/bash
# filename: sum_function5.sh
# 定义函数
sum()
{
  # 若位置参数大于两个，就对所有的位置参数求和
  if (($#>2)) ; then
    # echo  "$@" ,  $i  ,  $s  ,  $sumstr
    s=$1 ; sumstr=$1 ; shift
    # echo  "$@" ,  $i  ,  $s  ,  $sumstr
    for i ; do
      ((s=s+i))               # 做整数累加
      sumstr=$sumstr+$i       # 做字符串连接
      shift
      # echo  "$@" ,  $i  ,  $s  ,  $sumstr
    done
    echo $sumstr=$s
  else
    # 若位置参数在两个之内，做累加运算
    if (($# == 0)) ; then
       from=1; to=100
    elif (($# == 1)) ; then
       from=1; to=$1
    elif (($# == 2)) ; then
       from=$1; to=$2
    fi
    ((i=1,s=$from))
    while ((i<$to)); do ((i++,s+=i)); done
    echo sum\($from..$to\)=$s
  fi
}

# 调用函数
sum
sum 10
sum 3 201
sum 10 20 30
sum 24 35 16 7 21
sum 0 22 33                   # 计算两个整数的和
```

下面列表给出执行 sum 24 35 16 7 21 函数调用的 for 循环每次执行时各个变量的值

	“$@”	$i	$s	$sumstr	说明
刚进入函数时	24 35 16 7 21				
循环之前的值	35 16 7 21		24	24	因为循环之前执行了一次 Shift
第1次循环后	16 7 21	35	59	24+35	
第2次循环后	7 21	16	75	24+35+16	
第3次循环后	21	7	82	24+35+16+7	
第4次循环后		21	103	24+35+16+7+21	当前“$@”的值为空，所以不再进行下一次循环

例6：

```
#!/bin/bash
# filename: switch-lang3.sh
# 定义两个函数
prompt()
{
  cat <<EOF
Please choice a locale:
    1) -- en_US.utf8
    2) -- en_US.iso88591
    3) -- zh_CN.gb18030
    4) -- zh_CN.utf8
EOF
}

uasge()
{
  echo "Usage : . $0 <1|2|3|4>"
}

# 主程序
prompt
read choice
case $choice in
1) export LANG="en_US.utf8" ;;
2) export LANG="en_US.iso88591" ;;
3) export LANG="zh_CN.gb18030" ;;
4) export LANG="zh_CN.utf8" ;;
*) uasge ;;
esac
```

- 显示源文件
- 登录