



第三章

Linux Shell编程入门

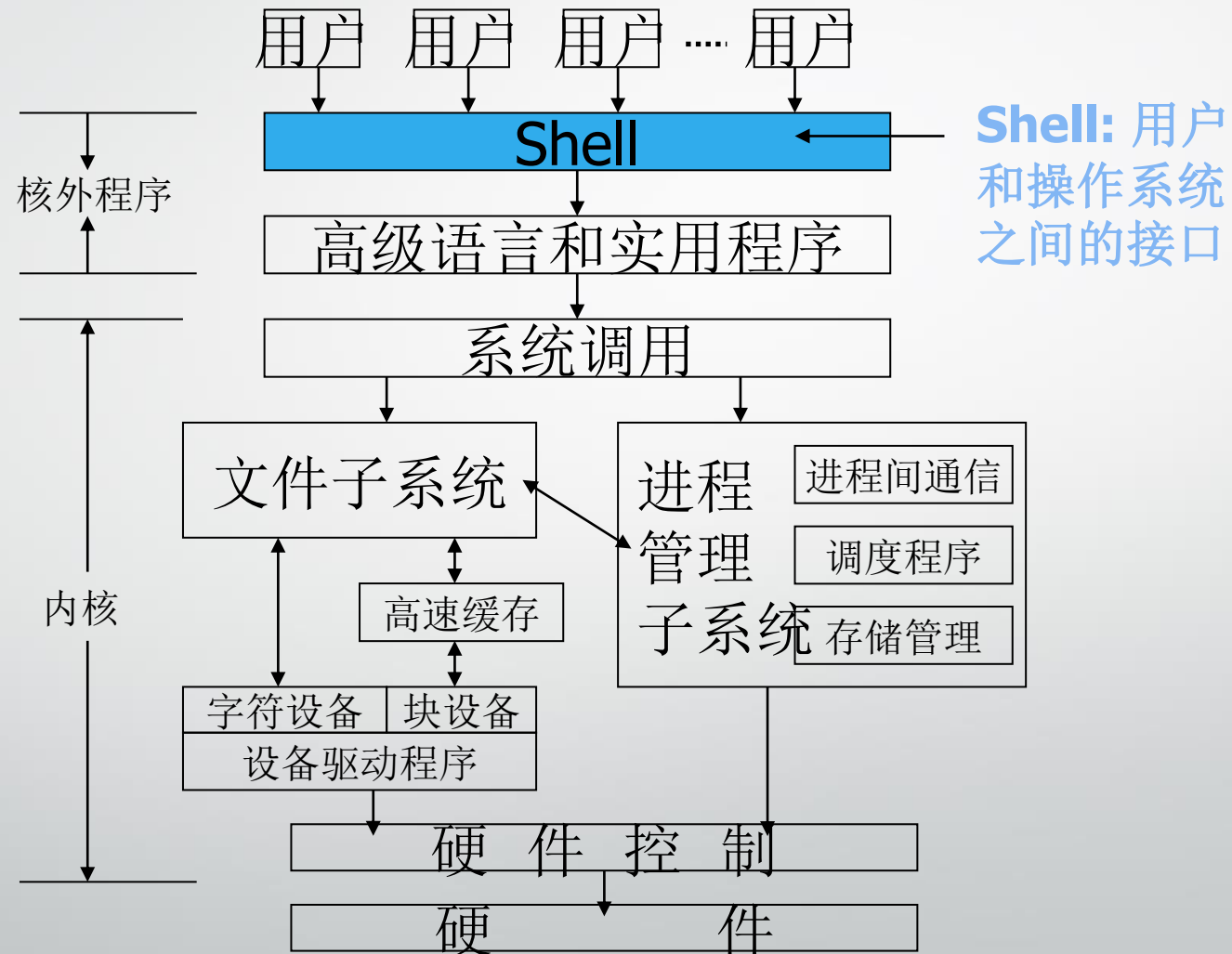
徐刚 工程师

华东师范大学软件工程学院

Outline

- Shell的工作原理
- Shell命令元字符
- Shell脚本
- Shell程序变量
- Shell程序语句
- Shell输入输出流

Shell



Shell的工作原理

- 当用户登录到系统时，有一个shell进程随之启动，并在用户注销时终止

```
draco@ubuntu:~$ ps
  PID TTY          TIME CMD
 37948 pts/20        00:00:00 bash
 38644 pts/20        00:00:00 ps
```

- 用户输入的命令是shell的输入
- shell处理命令的步骤：
 - 查找命令中的元字符
 - 把这些元字符替换成对应的实际操作参数
 - 将重新生成的指令传给内核执行
 - 等待命令完成，提示符重新出现，等待下一条命令

Shell种类

- Bourne系列
 - Bourne Shell (/bin/sh)
 - Korn Shell (/bin/ksh)
 - Bash (/bin/bash)
- C Shell系列
 - C Shell (/bin/csh)
 - Tcsh (/bin/tcsh)
- 可以用命令 `echo $SHELL` 查看当前使用的shell种类

元字符

- 通配符
- 转义符
- 命令替换
- 重定向

通配符

- 用来表示文件名的某种模式
- 在解释时被替换成其他字符

通配符	匹配内容
*	任意数量的任意字符
?	单个任意字符
[abc]	a, b, c中的任一个字符
[a-z]	ASCII码值在a与z中间的任一个字符
[!a-z]	不在a-z范围内的任一个字符
!(fname)	除fname之外的所有文件名
!(f1 f2)	除f1和f2之外的所有文件名

- 示例：

课堂问答

- 请解释下列命令的功能
 - `ls .??*`
 - `rm *.c`
 - `rm * .c`
 - `ls c*[!0-9]`
 - `cp file f[1-3]`
 - `cp f[1-3] file`

转义符

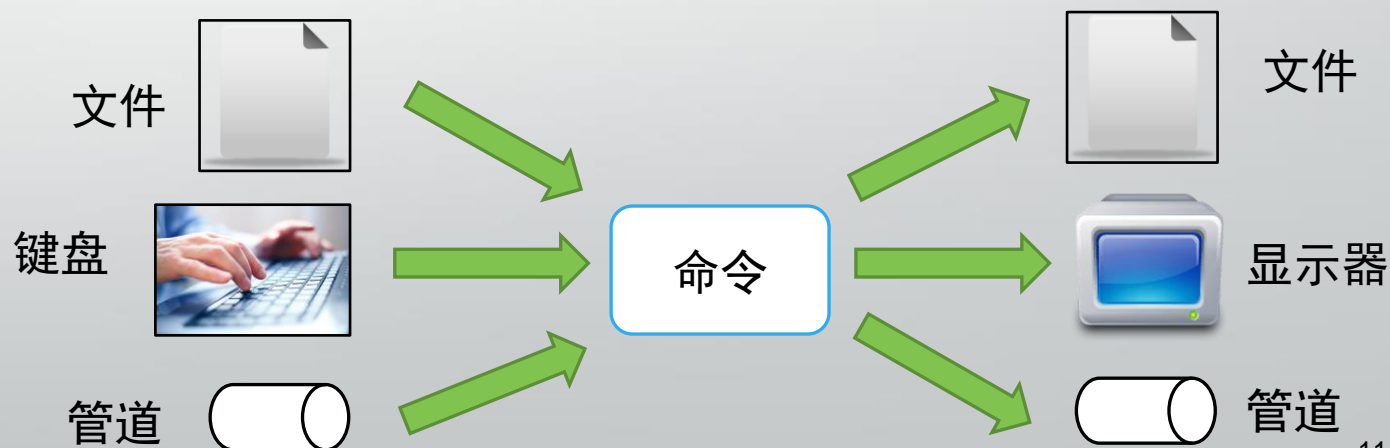
- 在处理文件名中出现通配符的文件时，常常需要用转义符来防止通配符起作用
- 在通配符前使用一个\
 - *
 - \?
 - \[\]
 - \[Enter] 换行继续输入
- 使用引号
 - 双引号：禁止通配符替换
 - 单引号：禁止通配符、变量名和命令替换

命令替换

- 反引号 `命令`：使用命令的结果来替换命令
 - `echo -e "The user list is \n `who`"`
 - `echo The current time is `date``
- `$(命令)`：整个替换成括号中命令的结果
 - `echo The current time is $(date)`

命令的输入输出流

- 输入和输出终端
 - 键盘、显示器
- 命令的输入和输出字符流
 - 标准输入流：连接到键盘的输入
 - 标准输出流：连接到显示器的输出
 - 标准错误流：连接到显示器的错误消息输出
- 三种标准输入输出源
 - 终端
 - 文件
 - 管道



输入输出重定向到文件

- - 标准输入
 - `cat - foo`
- > 输出重定向
 - `echo content > file`
- >> 追加输出重定向
 - `echo content >> file`
- < 输入重定向
 - `wc < file`
- 2> 标准错误输出重定向
 - `cat foo 2> errorfile`
- 相互重定向
 - `1>&2`
 - `2>&1`

文件描述符	含义
0	标准输入
1	标准输出
2	标准错误
3...	其他文件

过滤器

- 同时用到输入和输出流的命令
- 对于过滤器来说，输入输出流的顺序是不重要的
 - `wc < infile > outfile`
 - `wc > outfile < infile`
 - `>outfile < infile wc`
- 示例：
 - `cat - f2 < f1 > f3 2> f4`

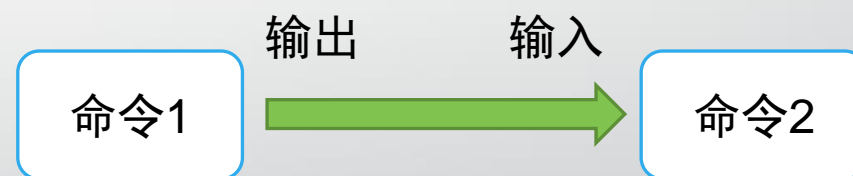


命令组合

- 使用圆括号()和花括号{ }将多个命令组合起来
- 组合命令由一个子shell执行，共享输入和输出源
- 组合命令能够共享同一个重定向符号
- 示例：
 - (ls -l *.txt; wc *.txt) > all

管道

- 将一条命令的输出作为另一条命令的输入
- 格式：命令1 | 命令2
- | 左边的命令使用标准输出，| 右边的命令使用标准输入
- 示例：
 - `who -a | cat > userlist`
 - `(cat f1; echo var2) | wc -l`
 - `ls | cat - file | wc`
- 优点：
 - 处理速度快
 - 不产生中间文件结果



- ; 依次执行命令
- && 前一个命令成功，则继续后面的命令
- || 前一个命令失败，则继续后面的命令

Shell 变量

- 环境变量：shell已经定义的变量，如HOME，SHELL，PATH
- 用户自定义变量：
 - 变量名：必须以字母开头，其他可以是数字和_，区分大小写
 - 变量类型：shell变量都是字符串类型
 - 变量不需要提前声明
 - 变量赋值：variable=value
 - 变量引用：\$var_name （可使用转义符\和单引号来避免替换）
- 示例：
 - ```
dracoking@ubuntu:~/Documents$ route=~/Documents
dracoking@ubuntu:~/Documents$ ls -l $route
```

# 环境变量

- 设置环境变量: `export 变量名=变量值`
- 引用环境变量: `$变量名`
- 查看环境变量: `env`命令
- 常用环境变量:
  - HOME 用户目录
  - PATH 命令执行路径, 用冒号分隔, Shell会按PATH中给出的顺序搜索目录
  - PWD 当前工作目录的绝对路径名
  - UID 当前用户的识别号
  - RANDOM 一个随机数

# 设置和修改环境变量

- 在shell中直接修改，在shell中生效
- 在启动配置文件中修改，在每次登录后生效
  - 系统配置文件：/etc/profile
  - 用户配置文件： ~/.profile, ~/.bashrc
- 设置路径变量PATH
  - export PATH="**\$PATH**:newpath"

# 用户自定义变量的引用和赋值

- | 变量引用                  | 变量替换  |
|-----------------------|-------|
| <code>\$变量名</code>    | 变量值   |
| <code>\${变量名}</code>  | 变量值   |
| <code>\${#变量名}</code> | 变量值长度 |

- | 变量赋值                       | 赋值语义                     |
|----------------------------|--------------------------|
| <code>y=\${x-value}</code> | 若x有值，则y=x，否则y=value      |
| <code>y=\${x=value}</code> | 若x没有值，则y=x=value，否则y=x   |
| <code>y=\${x+value}</code> | 若x有值，则y=value，否则不操作      |
| <code>y=\${x?value}</code> | 若x有值，则y=x，否则在标准错误输出value |

# 变量操作命令\${var}

- 给定变量名x，\${x}返回变量的值（与\$x相同）
- 字符串截断和提取

| 操作          | 返回值               | 示例: var=~ /my/file.txt |
|-------------|-------------------|------------------------|
| \${var#s}   | 去掉从左边开始第一个匹配s的子串  | \${var#*/}=my/file.txt |
| \${var##s}  | 去掉从左边开始最后一个匹配s的子串 | \${var##*/}=file.txt   |
| \${var%s}   | 去掉从右边开始第一个匹配s的子串  | \${var%/*}=~/my        |
| \${var%%s}  | 去掉从右边开始最后一个匹配s的子串 | \${var%%/*}=~          |
| \${var:m:n} | 提取从m位置开始往后的n个连续字符 | \${var:2:5}=my/fi      |

- 字符串替换

| 操作           | 返回值            | 示例: var=~ /my/file.txt     |
|--------------|----------------|----------------------------|
| \${var/s/t}  | 把第一个匹配s的串替换成t  | \${var/t/e}=~/my/file.ext  |
| \${var//s/t} | 把所有的匹配s的串都替换成t | \${var//t/e}=~/my/file.exe |

# 运算符

- 数值运算

- let 命令 (shell内部命令)
- `$(( 表达式 ))` (shell扩展命令)
- expr 命令
- bc 浮点数计算器

- 示例:

- ```
dracoking@ubuntu:~/Documents$ let x=1; echo $x
1
dracoking@ubuntu:~/Documents$ let x=$x+1; echo $x
2
```

- ```
dracoking@ubuntu:~/Documents$ x=1
dracoking@ubuntu:~/Documents$ echo $x+1
1+1
dracoking@ubuntu:~/Documents$ echo $((x+1))
2
```

# bc 命令

- 一个支持浮点数运算的计算器
- 通常采用管道的方式接受输入并返回计算结果
- 常用参数
  - `scale=n` 保留小数点后n位 (整型数运算中无效)
  - `ibase=k` 指定输入为k进制
  - `obase=k` 指定输出为k进制
- 常用运算符: `+`, `-`, `*`, `/`, `^`, `%`
- 示例:
  - ```
dracoking@ubuntu:~/Documents$ echo "scale=2;3+3*8" | bc
27
```
 - ```
dracoking@ubuntu:~/Documents$ echo "scale=2;obase=2;4^2" | bc
10000
```

# 数组

- 数组的声明
  - declare -a 数组变量名
- 数组赋值
  - array[0]=value0
  - array=(0 1 2 3)
  - array=(0 [3]=1 2)
  - 使用循环语句赋值
- 数组引用
  - \${array[i]} 访问数组array的第i个元素
  - \${array[\*]}或\${array[@]} 访问数组array的所有元素
  - \${#array[\*]}或\${#array[@]} 数组array的元素个数
- 数组销毁
  - unset array 销毁数组array
  - unset array[i] 销毁数组array中的第i个元素



# shell 脚本

- 将一组命令保存在文件中，然后逐条执行，类似于Windows的批处理文件
- 可以使用各种文件编辑软件来编写shell脚本

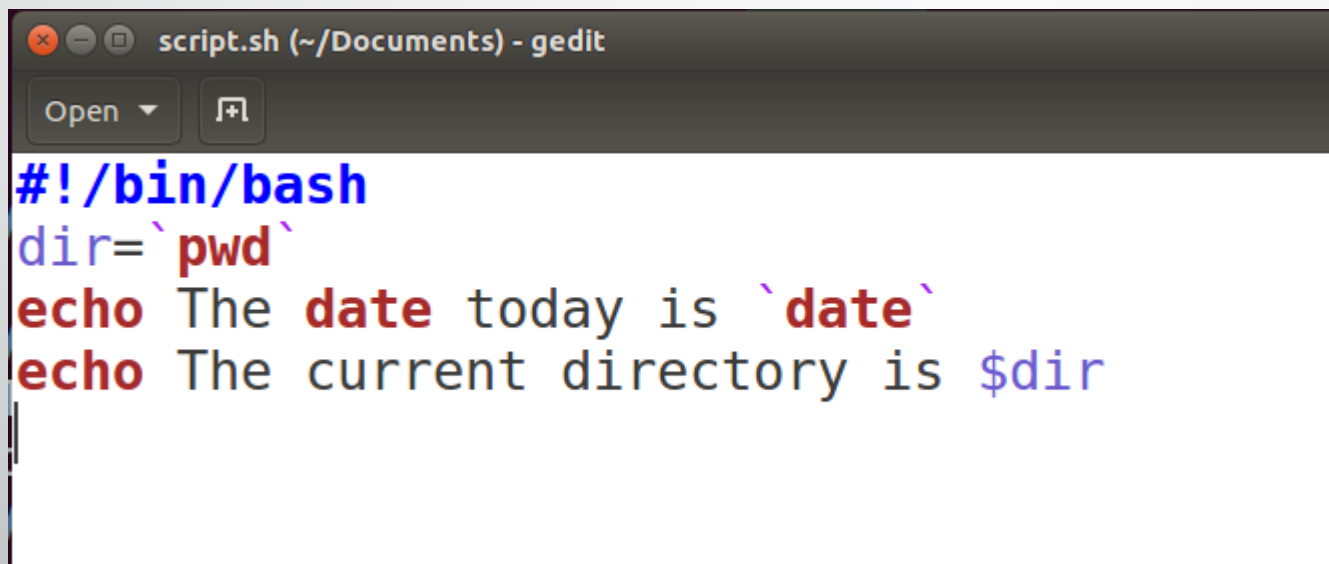
- ```
dracoking@ubuntu:~/Documents$ cat > script.sh
dir=`pwd`
echo The date today is `date`
echo The current directory is $dir
```

- 创建文件后使用chmod命令使其具有执行权限
- 执行该文件：

```
dracoking@ubuntu:~/Documents$ ./script.sh
The date today is Tue Sep 27 20:51:05 CST 2016
The current directory is /home/dracoking/Documents
```

解释器行

- `#!/bin/bash`
- 在shell脚本的第一行
- 用来指定运行该脚本的shell种类

A screenshot of a gedit text editor window. The title bar shows 'script.sh (~/.Documents) - gedit'. Below the title bar is a toolbar with an 'Open' button and a file icon. The main text area contains a shell script with syntax highlighting: the first line is '#!/bin/bash' in blue, followed by 'dir=`pwd`' in blue, 'echo The date today is `date`' in red, and 'echo The current directory is \$dir' in blue. The cursor is at the end of the last line.

```
#!/bin/bash
dir=`pwd`
echo The date today is `date`
echo The current directory is $dir
```

命令行位置参数

- 运行脚本时给定的用户输入参数
- 可以通过特定的形式在脚本中引用这些参数

位置参数	说明
\$0	所执行命令的名字
\$1--\$9	命令行参数1到9的名字
\$*	所有命令行参数组合成的字符串
\$#	命令行参数的个数
\$\$	当前shell的进程ID号
\$?	最近一次命令的退出状态（正常0）
\$_	最近一次后台进程的ID号

- 超过9个参数时，可使用shift命令将参数移位

读取用户输入

- read 变量表
- 从标准输入读取用户的输入，存入变量表中的变量
- -p 提示字符串
- -n 指定数量读取
- -t 指定时间读取
- -s 隐藏显示输入字符
- 示例：
 - `read -n 2 -t 10 -p "Input your name and age: " name age`

命令退出状态

- 所有命令和程序都会返回一个退出状态值
- 可以通过exit命令显示指定脚本的退出状态值
 - 正常退出 exit 0
 - 出错退出 exit 1
- 使用exit命令是一种好的编程习惯
- 可以使用参数\$?查看上一条命令的退出状态
- 示例：

- ```
draco@ubuntu:~/workspace/shell$ ls -l;echo $?
total 4
-rwxrw-r-- 1 draco draco 369 Oct 5 14:04 demo_shift
0
```

- ```
draco@ubuntu:~/workspace/shell$ ls demo;echo $?  
ls: cannot access 'demo': No such file or directory  
2
```

Shell控制结构语句

- 顺序执行：在同一行内使用;分隔命令
- 条件执行：使用逻辑运算符&&和||分隔命令
 - `cmd1 && cmd2` 若cmd1成功，则执行cmd2
 - `cmd1 || cmd2` 若cmd1失败，则执行cmd2
- 命令的成功与否取决于命令的退出状态值
- 示例：

- ```
draco@ubuntu:~/workspace/shell$ ls -l && echo "Command Successful"
total 4
-rwxrw-r-- 1 draco draco 369 Oct 5 14:04 demo_shift
Command Successful
```

- ```
draco@ubuntu:~/workspace/shell$ ls demo || echo "Command Fail"
ls: cannot access 'demo': No such file or directory
Command Fail
```

条件选择语句 if

- 格式1：

```
if 条件测试命令
    then
        条件为真时的命令串
    else
        条件为假时的命令串
fi
```
- 格式2：

```
if 条件测试命令
    then
        条件为真时的命令串
    elif 条件测试命令
        then 条件为真时的命令串
    else 条件为假时的命令串
fi
```

条件测试命令

- test命令（缩写[]）
 - 比较两个数值
 - 比较两个字符串，或与null进行比较
 - 查看文件特性
- test没有任何输入，只记录退出状态值\$？
- 示例：

```
draco@ubuntu:~/workspace/shell$ x=1;y=2;z=3
draco@ubuntu:~/workspace/shell$ test $x -eq $y;echo $?
1
draco@ubuntu:~/workspace/shell$ [ $x -lt $z ];echo $?
0
```

```
draco@ubuntu:~/workspace/shell$ x="str 1";y="str 2"
draco@ubuntu:~/workspace/shell$ [ -n "$x" ];echo $?
0
draco@ubuntu:~/workspace/shell$ [ "$x" = "$y" ];echo $?
1
```

数值比较运算符	意义
-eq	等于
-ne	不等于
-gt	大于
-ge	大于等于
-lt	小于
-le	小于等于

字符串比较运算符	意义
=	等于
!=	不等于
-n str	str不为空
-z str	str为空
str	str被赋值且不为空

文件特性检查

- test命令检查文件特性

命令	意义
[-e file]	file存在
[-f file]	file存在并是一个常规文件
[-d file]	file存在并是一个目录
[-L file]	file存在并是一个符号链接
[-rwx file]	file存在且可读可写可执行
[-s file]	file存在且大小大于0
[f1 -nt f2]	f1比f2更新
[f1 -ot f2]	f1比f2更旧
[f1 -ef f2]	f1被链接到f2

复合条件

- 可以在if中使用复合条件运算符
 - && 和 ||
 - `if ["$0"="cmd"] || ["$0"="./cmd"]`
 - -a 和 -o
 - `if ["$0"="cmd" -o "$0"="./cmd"]`

case语句

- 格式：`case 表达式 in`
 模式1) 命令串1 ;;
 模式2) 命令串2 ;;
 ...
 模式 *) 命令串n
 `esac`
- 按照从上到下的顺序匹配表达式的值，如匹配成功则执行相应的命令串
- 最后一个模式匹配任何未与之前匹配的表达式
- case语句也支持通配符
- 示例：srt_case

循环语句 for

- 格式：

```
for 变量 in 列表
do
    命令串
done
```
- 每次迭代中列表中的值被依次赋给变量，并执行循环体
- 当列表结束时，循环结束
- 示例：srt_for

循环语句while

- 格式：while 条件测试命令
do
 命令串
done
- 当条件测试命令返回真时，执行循环体
- 当条件测试命令返回假时，循环结束
- 示例：srt_while

循环语句until

- 格式：
until 条件测试命令
do
 命令串
 if xxx continue;
 other commands
done
- 当条件测试命令返回假时，执行循环体
- 当条件测试命令返回真时，循环结束
- 示例：srt_until

break和continue

- break语句：跳出循环体，执行done之后的语句
- ★ • continue语句：跳到done的位置，重新执行循环

shell函数

- 函数声明：

```
function() {  
    statements  
    return value  
}
```
- 函数调用：使用函数名进行调用，并可在名称后面给出命令行参数
- 函数支持对命令行位置参数的引用
- 函数的退出状态由return返回
- 函数可通过标准输出向调用者返回输出字符串
- 示例：srt_function

练习

- 编写脚本exe1，该脚本接收一个命令行参数，并根据其类型做以下操作：
 - 若参数为普通文件，则显示其内容
 - 若参数为压缩文件，则解压缩（如同目录下有同名文件则放弃）
 - 若参数为目录，则将其归档并压缩（如已有同名压缩文件则放弃）
 - 若参数不存在，给出错误提示并退出
- 编写脚本exe2，由用户输入一组数（以end表示输入结束），输出这些数的和，结果保留2位小数。要求使用函数做输入类型检查，并给出错误提示信息。
- 编写脚本exe3，该脚本对比两个目录dir1和dir2（通过参数给出），将dir2中符合下列条件的文件复制到dir1，并将每一条复制记录存储到文件record中：
 - 该文件不在dir1中
 - 该文件比dir1中的同名文件更新

作业

- 完成上述练习中的三个脚本，并提交到课程repo的CH03分支
- 命名格式：学号_姓名_ch3.tar.gz
- 以下情况不给分：
 - 迟交
 - 命名格式错误
 - 无法正常解压
 - 代码抄袭