第 5 讲　Shell 脚本编程

王晓庆

wangxiaoqing@outlook.com

April 22, 2016

Outline

创建新命令

示例：统计当前有多少不同用户在线

```
who | cut -d" " -f1 | sort | uniq | wc -l
# 如需反复执行上述命令，则可以将其组织成一个新命令
echo 'who | cut -d" " -f1 | sort | uniq | wc -l' >nu
# 执行
1. sh <nu
2. sh nu
3. chmod +x nu
   ./nu
4. mkdir bin
   echo $PATH
   mv nu bin
   ls nu
   nu
```

命令参数

示例：创建名为 cx 的程序，为文件设置可执行权

```
cx nu     # 相当于 chmod +x nu 的缩写
# 问题：如何为 cx 传递文件名参数？
echo 'chmod +x $1' >cx
sh cx cx
mv cx bin
echo echo Hi, there >hello
cx hello
./hello
# 问题：如何处理多个参数？
chmod +x $1 $2 $3 $4 $5 $6 $7 $8 $9
chmod +x $*
echo 'wc -l $*' >lc
cx lc; mv lc bin
lc vim*
```

命令参数

- shell 脚本的参数不一定是文件名

示例：

假设文件/usr/share/phone-book保存了个人电话目录：

```
dial-a-joke 212-976-3838
dial-a-prayer 212-246-4200
dial santa 212-976-3636
dow jones report 212-976-4141
```

现在需要创建一个查号程序411：

```
echo 'grep $* /usr/share/phone-book' >411
cx 411; mv 411 bin
411 joke              #:-)
4111 'dow jones'      #:-(
```

I 节点

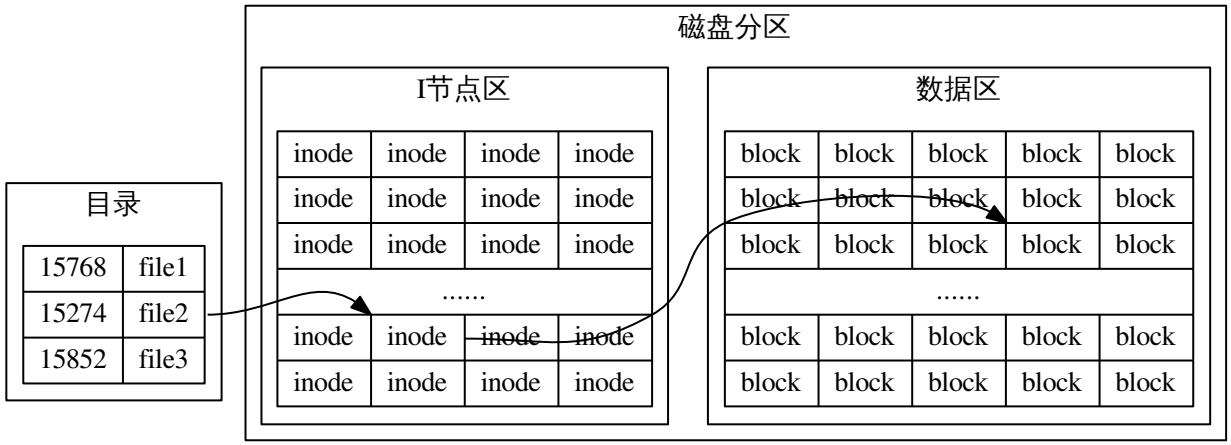
- 一个文件包含：文件名、内容及管理信息 (如权限、修改时间、文件长度、文件内容的存放位置等)
- 文件的管理信息存放在 I 节点中，可以认为 I 节点就是文件，文件的系统内部名称就是它的 I 节点号。

示例

```
date >junk
ls -i junk          # 查看文件的 I 节点号
ls -l junk          # 显示最近修改 (写) 时间
cat junk
ls -lu junk         # 显示最近访问 (读/执行) 时间
ls -lc junk         # 显示最近修改 I 节点 (如设置权限) 时间
chmod 444 junk
ls -lc junk
```

I 节点

- 目录、文件及 I 节点的关系如下图所示：

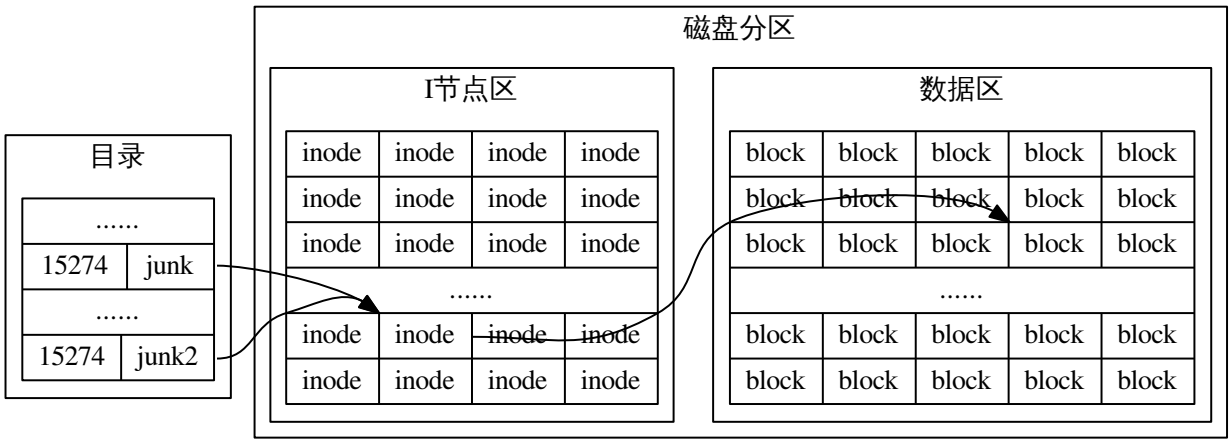


- 目录中的文件名被称作链，因为它把目录层次结构中的名称链接到它的 I 节点，因而也就链接到数据。
- 同一个 I 节点号可以出现在多个目录项中
- rm 命令并不真正删除 I 节点，它删除目录入口或链，只有当链接到文件的最后链消失后，系统才删除文件本身。

链接

- 创建硬链接

```
ln junk junk2
```

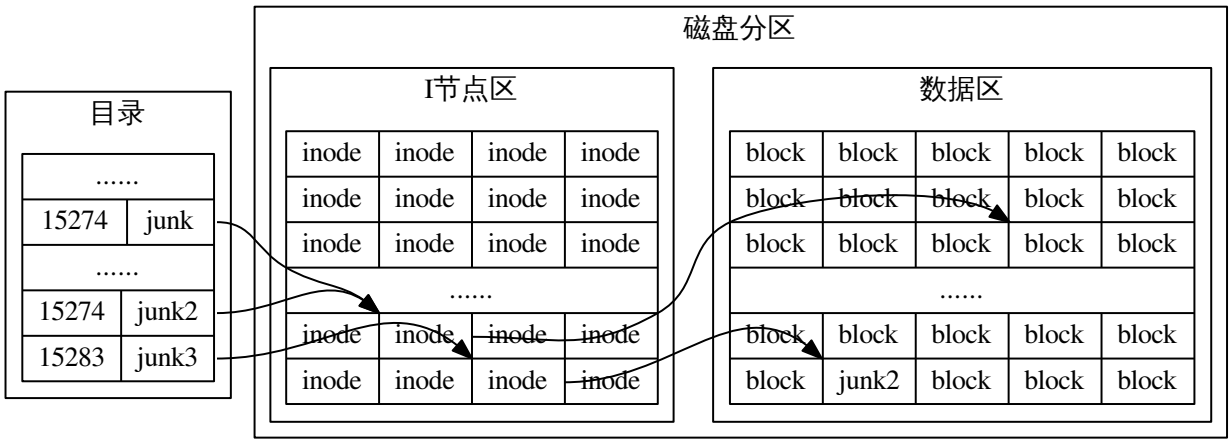


- 不能跨文件系统创建硬链接
- 不能为目录创建硬链接

链接

- 创建符号 (软) 链接

```
ln -s junk2 junk3
```



- 符号链接保存的是路径 (相对路径/绝对路径)
- 符号链接依赖于其目标文件

链接

试一试

假设 mike 的主目录下有一个文件 scratch，能否在 mary 的主目录下为该文件建立一个硬链接，使得 mike 和 mary 能够共享该文件？

命令参数

- 参数 $0 是指正在执行的程序的名字

示例：将输入进行多列打印

```
ls | pr -t -5 # 将输入按 5 列输出
echo 'pr -t -5' >5
cx 5; mv 5 bin
ls | 5
若要按 2 列、3 列、4 列、6 列输出呢？
echo 'pr -t -$0' >5
ln 5 2;ln 5 3;ln 5 4;ln 5 6
ls | 5          #:-(
```

程序输出作为参数

前面的程序有什么问题呢？

```
vim 5
echo pr -t -$0 # 调试技巧：将命令语句打印出来
ls | 5
vim 5
pr -t -`basename $0`   # 反引号：命令替换
ls | 5
```

命令替换

```
# 打印命令 cat 所在的目录
echo `dirname \`which cat\``
echo $(dirname $(which cat))
```

shell 变量

- set 命令可以查看所有变量的值
- 一个变量的值与创建它的 shell 有关，其值并不会自动传递给子 shell

示例

```
x=Hello     # 变量无需声明，注意：= 两边不能有空格！
sh          # 进入子 shell
echo $x
Ctrl-d      # 离开子 shell
echo $x
# 因为 shell 脚本是由子 shell 运行的，所以不能修改变量的值
echo -e 'x="Bye"\necho $x' >setx
sh setx
echo $x
```

行内赋值

- 行内赋值可用于临时改变变量的值并传给脚本

示例

```
echo 'echo $x' >echox
cx echox;mv echox bin
x=300
x=500 echox
echo $x
```

在当前 shell 中执行 shell 脚本

- 能否想办法用 shell 脚本来改变 shell 变量的值？

示例

```
echo $PATH
echo 'PATH=$PATH:/sbin' >>.bash_profile
sh .bash_profile
echo $PATH
. .bash_profile
echo $PATH
```

说明

source命令与. 命令的意义相同：

```
source .bash_profile
```

here 文档 (1)

- here 文档：把命令的标准输入和命令放在一起

示例 1

```
cat 411
grep "$*" <<End
dial-a-joke 212-976-3838
dial-a-prayer 212-246-4200
dial santa 212-976-3636
dow jones report 212-976-4141
End
#End(可自行选取其他单词) 用于开始和终止输入 (here 文档)
#           <<End: here 文档内的 $、``和\会被替换
#<<\End 和<<'End': here 文档内的 $、``和\不被替换
```

here 文档 (2)

示例 2

```
cat avi
#!/bin/bash
if [ "$#" -ne 1 ]; then
  echo "Usage: avi file" 1>&2; exit 1
fi
vi $1 &>/dev/null <<EOF
iTo be or not to be,
It is a problem.^[
ZZ
EOF
# 注意：^[代表 ESC 键，需先按 C-v，再按 ESC 进行输入
avi problem  # 执行 avi
```

循环

基本的 for 循环

```
# 多行
for i in a "b c" d
do
  echo $i
done
# 单行
for i in a 'b c' d; do echo $i; done
# 通过命令生成列表
for i in `seq 10 20`; do
  echo $i
done
```

循环

基本的 for 循环

```
# 循环与管道
for i in *; do
  ls -l $i
done | grep '\.doc$' | wc -l
# 循环与参数
for i in $*; do echo $i; done
for i in "$*"; do echo $i; done
for i in $@; do echo $i; done
for i in "$@"; do echo $i; done
for i; do echo $i; done
```

问题

- mike 要将自己 bin 目录内的多个脚本文件通过邮件发送给 mary，为了方便，他想把所有脚本打包成一个文件发送，而且希望 mary 能够通过用 shell 执行该文件自动还原出所有脚本。

bundle

bundle 程序

```
cat bundle
echo "# To unbundle, sh this file."
for i; do
  echo "echo $i 1>&2"
  echo "cat >$i <<'End of $i'"
  cat $i
  echo "End of $i"
done
```

测试

```
bundle nu cx >junk      # 打包
mkdir test; cd test
sh ../junk              # 解包
```

#! 行和注释

- #! 行：shell 执行脚本时启动该行指定的程序对脚本进行解释执行
- # 单行注释
- shell 并不直接支持多行注释，但可以用以下方法实现多行注释

```
:<<COMMENT
...
...
COMMENT
```

shell 变量 (1)

- shell 是一种动态类型语言和弱类型语言
  - 动态型：变量的数据类型无需显式地声明
  - 弱类型：变量的数据类型会根据不同的操作有所变化
- 准确地说，shell 变量并不分数据类型，统一按字符串存储。

shell 变量 (2)

- 变量的定义
  - shell 变量无需先定义，第一次为某个变量名赋值时，实际上就同时定义了这个变量。在变量作用域内都可以使用该变量。

示例

```
cat var.sh
echo $a
a=300; b="hello"
echo $a $b
unset a          # 删除变量
echo $a $b
```

shell 变量 (3)

- 变量的定义
  - 为了更好地控制变量相关属性，bash 提供了 declare 命令来声明变量

示例

```
x=6/3; echo $x     #x 的值为 6/3
declare -i x       # 声明 x 为整数
echo $x            #x 的值仍为 6/3
x=6/3; echo $x     # 重新赋值后，x 的值为 2
x=3.14; echo $x    # 不支持浮点数，值变为 0
declare -r x       # 声明 x 为只读变量
x=100
declare -p x       # 显示变量 x 的声明
declare -p         # 显示所有变量
# -a(声明数组) -f(声明函数) -x(声明环境变量)
```

shell 变量 (4)

- 特殊变量与 shift 命令

特殊变量

```
$1~$9     # 第 1~9 个位置参数
${10}     # 第 10 个位置参数
$*, $@    # 所有位置参数
$#        # 位置参数个数
$0        # 当前脚本路径名
$ $       # 当前脚本进程号 (注：两个 $ 应该靠在一起)
$?        # 上一条命令的返回值
```

shift [n]

所有位置参数左移 n 个位置 (默认左移 1 个位置)，最左边的 n 个参数被移除

shell 变量 (5)

示例

```
cat shift.sh
#!/bin/bash
echo "pid: $$"
echo "arg counts: $#"
echo "args: $@    first arg: $1"
shift; echo "after shift"
echo "arg counts: $#"
echo "args: $@    first arg: $1"
shift 3; echo "after shift 3"
echo "arg counts: $#"
echo "args: $@    first arg: $1"

sh shift.sh 1 2 3 4 5 6 7 8 9 &
echo $?
```

退出

- exit [n] (n=0~255)
  - 返回 0 表示成功，否则返回非 0
  - 省略 n，则返回 exit 命令前一条命令的返回值
- $?
  - 上一条命令的返回值

示例

```
who; echo $?
woh; echo $?
true; echo $?
false; echo $?
:; echo $?
```

变量的作用域

- 普通变量
  - 普通变量在被定义后，可在该 shell 中被访问，直至退出该 shell 或被删除
- 环境变量
  - 环境变量不仅可在定义的 shell 及其所有子 shell 中被访问

示例

```
a=300; echo $a
sh; echo $a
exit
export a; export b="hello"
sh; echo $a $b
a=500; echo $a $b
exit
echo $a $b
```

变量替换 (1)

示例

```
$ a=teach
$ echo "he is a $aer"
$ echo "he is a ${a}er"
$ echo 'he is a ${a}er'
$ date=’ 07/23/2010’
$ echo ${date}
$ echo $(date)
```

变量替换 (2)

- 条件变量替换

${var:-string}

若 var 存在且非空, 则返回 var 的值, 否则返回 string

${var:=string}

若 var 存在且非空, 则返回 var 的值, 否则把 string 赋给 var, 并返回 string

变量替换 (3)

- 条件变量替换 (2)

${var:?message}

若 var 存在且非空, 则返回 var 的值, 否则显示字符串“var:”并在其后显示“message”

${var:+message}

若 var 存在且非空, 则返回“message”, 否则返回 null

变量替换 (4)

- 条件变量替换 (3)

示例

```
name=Tom
echo $name
echo $place
echo ${name:-John} ${place:-Beijing}
echo ${place:?"var place not defined."}
echo ${name:+"var name has been defined"}
echo ${place:="Nanchang"}
echo ${name:-John} ${place:-Beijing}
```

变量替换 (4)

- 截取变量替换

${var%pattern}

从 var 右边去掉模式 pattern 的最短匹配内容

${var%%pattern}

从 var 右边去掉模式 pattern 的最长匹配内容只有在 pattern 中用了 * 时, 二者效果才不同

变量替换 (5)

- 截取变量替换 (2)

${var#pattern}

从 var 左边去掉模式 pattern 的最短匹配内容

${var##pattern}

从 var 左边去掉模式 pattern 的最长匹配内容

- 注意: 在上述替换中并不会修改变量的值

变量替换 (6)

示例

```
var=testcase
echo ${var%s*e}    # 从右边删除最短匹配
echo ${var%%s*e}   # 从右边删除最长匹配
echo ${var}        # 查看变量是否已被改变
echo ${var#t*s}    # 从左边删除最短匹配
echo ${var##t*s}   # 从左边删除最长匹配
fname="game.tar.gz"
echo ${fname%%.*}
echo ${fname#*.}
cat mybasename
echo ${1##*/}
./mybasename `pwd`
```

变量替换 (7)

- 取变量长度和子串

取变量长度

```
var=123456
echo ${#var}
```

取变量子串

```
str="GNU's Not Unix"
echo ${str:0:3}
echo ${str::3}
echo ${str:6:3}
echo ${str:6}
```

变量替换 (8)

${var/pattern/string} 查找替换

```
var=banana
echo ${var/na/la}       # 替换一次 pattern
echo ${var//na/la}      # 全部替换 (pattern 以/开头)
echo ${var/#ba/la}      # 仅替换开头 (pattern 以 # 开头)
echo ${var/%na/ma}      # 仅替换结尾 (pattern 以% 开头)
echo ${var/a/}          # 删除第一个 a(string 为空)
echo ${var//a/}         # 删除所有 a
```

算术运算 (1)

引例

```
x=8+10
echo $x
```

- shell 中的变量默认没有数据类型，都以字符串形式对待
- shell 中要进行算术运算，有多种方法
  1. declare 命令 (内部命令)
  2. expr 命令 (外部命令)
  3. let 命令 (内部命令)
  4. 算术扩展 $(())(bash 特性)
  5. $[](bash 特性)
  6. 调用 bc(外部命令)

算术运算 (2)

- declare 命令

示例

```
declare -i x
x=8+10; echo $x    # 在赋值操作符、算术运算符两边不能有空格
x=x-9; echo $x
x=x*4; echo $x
x=x/12; echo $x
x=x**3; echo $x
x=x%10; echo $x
```

算术运算 (3)

- expr 命令

示例

```
expr 3 + 2 #3、+、2 都看作 expr 的参数，因此要用空格分隔
x=`expr 4 - 7`; echo $x
x=`expr 3 \* 5`; echo $x # 要对 * 进行转义
x=`expr $x / 4`; echo $x # 要对 x 进行取值
expr 4 ** 2 # 错误，expr 没有乘幂运算符
expr 22 % 5
x=`expr 1 \< 2`; echo $x
x=`expr 2 = 2`; echo $x
x=`expr 3 \>= 2`; echo $x
x=100; r=`expr $x \| 1`; echo $r # 返回 100
x=0; r=`expr $x \| 1`; echo $r   # 返回 1
```

算术运算 (4)

- let 命令

示例

```
let i=8+16; echo $i
let x=(i-4)/5*9; echo $x
let i++; echo $i
let x/=6; echo $x
let y="x>10?1:0"; echo $y
```

算术运算 (5)

- $((express)) 算术扩展

示例

```
x=3
echo $((x+8))
echo $((x*9-10/2))
echo $((x++)); echo $x
echo $((--x)); echo $x
echo $((x**3%5)) $((x**(3%5)))
echo $((8<<2)) $((-8>>2))          # 左移和右移
echo $((x&6)) $((x|6)) $((x^6)) # 按位与、或、异或
((x++)); echo $x
((x*=6)); echo $x
((y=x<20?0:1)); echo $y
```

算术运算 (6)

- $[express] 算术扩展：与 $((express)) 用法类似
- 调用 bc：对于非整数运算可以通过 bc 进行计算

示例

```
echo "scale=10; 37/7" | bc
bc <<<"scale=10; 37/7"     #here 字符串
x=`echo "scale=10; 37/7" | bc`; echo $x
cat bc.task
12*34
34/12
scale=3; 34/12
a=1;b=2; a+b
cat bc.task | bc
bc <bc.task
```

数值

- Shell 脚本按十进制解释字符串中的数值, 除非有特殊前缀:
  - 前缀为 0: 八进制
  - 前缀为 0x(0X): 十六进制
  - 前缀为 n#: n 进制

示例

```
let x=32; echo $x
let x=032; echo $x
let x=0x32; echo $x
let x=2#111010100101; echo $x
```

测试 (1)

test 命令

```
test condition  #condition返回0表示true, 返回1表示false
[ condition ]   #同上
```

各种比较

1. 字符串比较
2. 整数比较
3. 文件状态/属性
4. 条件组合

测试 (2)

字符串比较

```
str="foo"
test "$str" = "for"; echo $?   # 相等
test "$str" != "for"; echo $?  # 不相等
test "$str"; echo $?           # 非空
test -n "$str"; echo $?        # 长度>0
test -z "$str"; echo $?        # 长度 =0
test "str" \< "for"; echo $?   # 小于
test "str" \> "for"; echo $?   # 大于

test $a = "bar"                #:-(
test "$a" = "bar"              #:-)
```

测试 (3)

整数比较

```
x=123
[ $x -eq 100 ]      # 相等
[ $a -ne 100 ]      # 不相等
[ $a -gt 100 ]      # 大于
[ $a -ge 100 ]      # 大于等于
[ $a -lt 100 ]      # 小于
[ $a -le 100 ]      # 小于等于
```

测试 (4)

- 文件测试

```
[ -r file ]    # 可读
[ -w file ]    # 可写
[ -x file ]    # 可执行
[ -f file ]    # 普通文件
[ -c file ]    # 字符设备
[ -b file ]    # 块设备
[ -d file ]    # 目录
[ -e file ]    # 文件存在
[ -h file ]    # 符号链接
[ -L file ]    # 同上
```

```
[ -u file ]    #suid 权限
[ -g file ]    #sgid 权限
[ -k file ]    #skicky 权限
[ -p file ]    # 命名管道
[ -s file ]    # 长度>0
[ -M file ]    # 共享内存
[ -H file ]    # 信号量
[ f1 -ef f2 ] # 硬链接
[ f1 -nt f2 ] #f1 比 f2 新
[ f1 -ot f2 ] #f1 比 f2 旧
```

测试 (5)

组合条件

```
genda=male; age=21
[ "$genda" = "male" -a $age -eq 21 ]; echo $?
[ "$genda" = "female" -o $age -gt 20 ]; echo $?
[ ! "$age" -lt 20 ]; echo $?
[ "$genda" = "male" ] && [ $age -eq 21 ]; echo $?
[[ "$genda" = "female" || $age -gt 20 ]]; echo $?
```

选择 (1)

- 问题：编写脚本根据当前时间问候早上 (0~11 时)/下午 (12 ~ 18)/晚上 (19~23) 好

版本 1: greeting1

```
#!/bin/bash
hour=$(date +%H)
if [ $hour -ge 0 ] && [ $hour -le 11 ]; then
  echo 'Good morning!'
else
  if [ $hour -ge 12 ] && [ $hour -le 18 ]; then
    echo 'Good afternoon!'
  else
    echo 'Good evening!'
  fi
fi
```

选择 (2)

版本 2: greeting2

```
#!/bin/bash
hour=$(date +%H)
if [ $hour -ge 0 ] && [ $hour -le 11 ]; then
  echo 'Good morning!'
elif [ $hour -ge 12 ] && [ $hour -le 18 ]; then
  echo 'Good afternoon!'
else
  echo 'Good evening!'
fi
```

选择 (3)

版本 3: greeting3

```
#!/bin/bash
hour=`date+%H`
case $hour in
0?|1[01]) echo 'Good morning!';;
1[2-8])   echo 'Good afternoon!';;
*)        echo 'Good evening';;
esac
```

循环 (1)

- while 循环

计算 n 的阶乘 (版本 1)

```
cat fac1
#!/bin/bash
if [ "$#" -ne 1 ]; then
  echo "usage: fac1 n" 1>&2; exit 1
fi
fac=1; i=2
while [ $i -le $1 ]; do
  fac=`expr $fac \* $i`
  i=`expr $i + 1`
done
echo "fac($1)=$fac"
```

循环 (2)

- until 循环

计算 n 的阶乘 (版本 2)

```
cat fac2
#!/bin/bash
case "$#" in
  0) echo "usage: fac2 n" 1>&2; exit 1;;
esac
fac=1; i=2
until [ $i -gt $1 ]; do
  ((fac*=i))
  ((i++))
done
echo "fac($1)=$fac"
```

循环 (3)

- for 循环 (1)

计算 n 的阶乘 (版本 3)

```
cat fac3
#!/bin/bash
n=${1:-1}; fac=1
for i in `seq 2 $n`; do
  let fac*=i
  let i++
done
echo "fac($n)=$fac"
```

循环 (4)

- for 循环 (2)

计算 n 的阶乘 (版本 4)

```
cat fac4
#!/bin/bash
if [ "$#" -ne 1 ]; then
  echo "usage: fac4 [n=1]" 1>&2
fi
n=${1:-1}; fac=1
for ((i=2;i<=n;i++)); do
  ((fac*=i))
done
echo "fac($n)=$fac"
```

break 与 continue

break [n=1]

停止并跳出 n 层 (1 为本层循环，2 为本层循环和上一层循环，...) 循环

continue [n=1]

停止当前循环，跳至第 n 层 (1 为本层循环，2 为上一层循环，...) 循环的下一次循环

空语句

- : 空语句，仅返回 0 (外部命令 true 与内部命令: 类似)

示例

```
while :
do
  sleep 1
  echo $((++i))
done

if [ -f file ]; then
  :
else
  touch file
fi
```

&& 和 ||

- 与结构: cmd1 && cmd2
  如果 cmd1 返回 0(true)，则执行 cmd2，否则不执行 cmd2
- 或结构: cmd1 || cmd2
  如果 cmd1 返回 1(false)，则执行 cmd2，否则不执行 cmd2

示例

```
cat ison
#!/bin/bash
if [ "$#" -eq 0 ]; then
  echo "usage: ison username"; exit 1
fi
who | grep "^$1" &>/dev/null &&\
 echo "$1 is loggoed on" ||\
 echo "$1 is not logged on"
```

与用户交互 (1)

read 命令

```
cat welcome
#!/bin/bash
echo -e "login: \c"    #\c 表示取消换行
read user
read -p "password: " -s pass  #-p 提示, -s 关闭回显
echo
if [ "$user" = "tom" ] && [ "$pass" = "123" ]; then
  echo "Welcome, $user"!; exit 0
else
  echo "login failed."; exit 1
fi
```

与用户交互 (2)

read 命令

```
cat whichkey
#!/bin/bash
until [ "$key" = "q" ]; do
  read -n 1 -s -p "please press a key" key #-n 读指定字符数
  echo -e "\n\tyou have pressed the key $key"
done
```

select 语句

示例

```
cat whichcolor
#!/bin/bash
PS3="Please choose your color: "
colors="red green blue white black quit"
select c in $colors; do
  if [ "$c" == "quit" ]; then
    exit;
  else
    echo "You have choose [$REPLY]: $c"
  fi
done
```

示例：猜数游戏

产生 100 以内随机数给用户猜直至猜中为止

```
cat guess
#!/bin/bash
n=$(($RANDOM%100))
until [ $g -eq $n ]; do
  echo -e "Please input your guess: \c"
  read g
  if [ $g -lt $n ]; then
    echo "too small, try again."
  else
    echo "too big, try again."
  fi
done
echo 'Wow! you are a genius!'
```

set 命令 (1)

- 位置参数的值不能直接修改，但 set 命令可重置位置参数

示例: 统计文件单词数 (版本 1)

```
cat cwords1
#!/bin/bash
if [ "$#" -ne 1 ]; then
  echo "usage: cwords file" 1>&2; exit 1
fi
fname="$1"
cat $fname | while read line; do
  set $line; ((n+=$#))
done
echo "$n $fname"

cwords1 emp.data    #:-(
```

set 命令 (2)

- 管道中的循环在子 shell 中运行，导致 n 值无法传出

示例：统计文件单词数 (版本 2)

```
cat cwords2
#!/bin/bash
if [ "$#" -ne 1 ]; then
  echo "usage: cwords file" 1>&2; exit 1
fi
fname="$1"
while read line; do
  set $line; let n+=$#
done <$fname  # 改用输入重定向
echo "$fname: $n words"

cwords2 emp.data    #:-)
cwords2 vim-creep   #:-(
```

set 命令 (3)

- set 的第一个参数若以-开头，会被 set 误以为是选项！

示例

```
set 3 + 4 = 7        #:-)
set -3 + 7 = 4       #:-(
set -- -3 + 7 = 4    #:-)
```

说明

–选项告诉 set 选项到此为止，后面都是参数，也可以防止 set 在没有参数时显示处所有的变量。

set 命令 (4)

示例：统计文件单词数 (版本 3)

```
cat cwords3
#!/bin/bash
if [ "$#" -ne 1 ]; then
  echo "usage: cwords file" 1>&2; exit 1
fi
fname="$1"
while read line; do
  set -- $line; let n+=$#
done <$fname  # 改用输入重定向
echo "$fname: $n words"

cwords2 emp.data    #:-)
cwords2 vim-creep   #:-)
```

set 命令 (5)

- 可以使用 set 或 shopt 命令修改 shell 的默认处理行为，定制自己的运行环境。

示例

```
set -o                  # 查看所有 set 选项
set -o noclobber        # 启用 noclobber 特性
set +o noclobber        # 关闭 noclobber 特性

shopt                   # 查看所有 shopt 选项
shopt -s cmdhist        # 启用 cmdhist 特性
shopt -u cmdhist        # 关闭 cmdhist 特性
```

命名管道

- 命名管道是 Unix/Linux 中最古老的进程间通信方式

示例

```
mkfifo fifo                    # 创建命名管道
ls -l fifo
cat emp.data >fifo &           # 向命名管道写入（注意要放在后台）
wc -l <filo                    # 从命名管道读取

tar -cf fifo dir &                  # 向命名管道写
bzip2 -c <fifo >dir.tar.bz2         # 从命名管道读
rm fifo                             # 删除命名管道
tar -tf dir.tar.bz2
```

进程替换

- 进程替换可让我们把标准输出，一次倒给多个进程作为输入，或者将多个进程的输出倒给一个进程去处理。

示例

```
#comm 命令要求被比较的两个文件事先排好序
comm <(sort file1) <(sort file2)
cmd1 <(cmd2) #cmd1 通过设备文件/dev/fd/n 读取 cmd2 的输出
cmd1 >(cmd2) #cmd2 通过设备文件/dev/fd/n 读取 cmd1 的输出
echo <(true); echo >(true)
# 下面的命令等价于 tar -czf dir.tar.gz dir
gzip -c <(tar -c dir) >dir.tar.gz
# 下面的命令等价于 tar -cjf dir.tar.bz2 dir
tar cf >(bzip2 -c >dir.tar.bz2) dir
```

函数 (1)

- 函数要先定义后调用，shell 函数不能与 shell 变量同名！

函数定义

```
[function] func_name(){ # 关键字 function 可省略
  cmd1                  # 若省略 function，则 () 不可省略
  cmd2                  # 若不省略 function，则 () 可省略
  ...
  cmdn
}
func_name(){ cmd1; cmd2; ...; cmdn; } # 注意空格与分号
```

函数调用

```
func_name par1 par2 ... parn
```

函数 (2)

- 函数的返回值

return [n]

1. exit 会退出整个脚本，而 return 仅从函数返回。
2. 如果省略 n，则返回值为 return 前一条命令的返回值。

函数简单示例

```
nu(){ who | wc -l; }   # 注意空格和分号
nu
type nu
declare -f nu
unset -f nu; nu
```

函数 (3)

函数参数处理示例

```
cat funcarg
#!/bin/bash
echoargs(){
  echo "in function:"
  echo -e "\targs counts: $#"
  echo -e "\targs: $@"
}
echo "out function:"
echo -e "\targs counts: $#"
echo -e "\targs: $@"
echoargs $2 $4   # 调用函数并传入参数

funcarg 1 2 3 4 5
```

函数 (4)

函数的局部变量 (作用域从定义处开始至函数结束处为止)

```
cat localvar
#!/bin/bash
func1(){
  echo "global var x is $x"
  local x=hello  # 定义局部变量
  echo "local var x is $x"
}
x=100
func1
echo "global var x is $x"
```

函数 (5)

- 函数库的使用
  开发较大 shell 程序时，可把一些公共函数放在单个脚本中形成函数库

函数库使用示例

```
cat lib.sh          # 函数库文件
#!/bin/bash
error(){ echo "ERROR: " $@ 1>&2; }
warn(){ echo "WARNING: " $@ 1>&2; }

cat main.sh         # 主脚本
#!/bin/sh
. lib.sh            # 导入函数库 (要用. 命令执行！)
msg="file not found"
error $msg          # 调用函数库函数
```

数组 (1)

数组的定义

```
declare -a season
season[0]="spring"
season[1]="summer"
season[2]="autumn"
season[3]="winter"

weekday=("Mon" "Tues" "Wed" "Thur" "Fri" "Sat" "Sun")

users=("alice" [4]="bob" "mary" [8]="susie")
suids=(`find /usr/bin -perm -4000 | sed 's#.*/##'`)
declare -a
declare -a suids
```

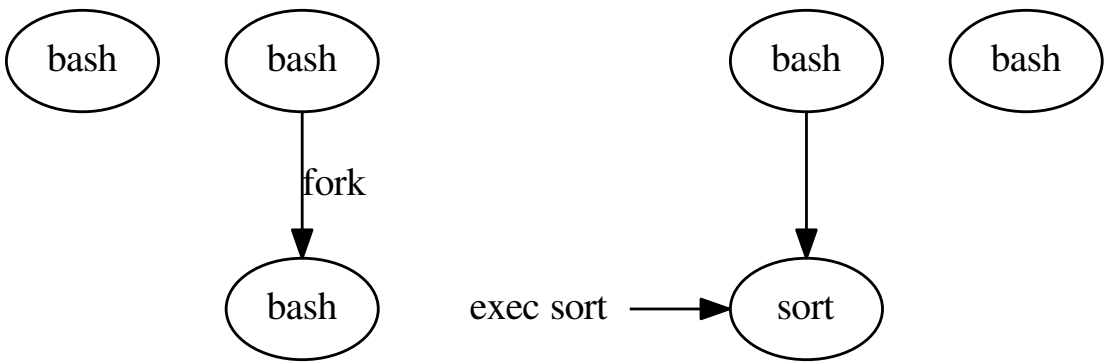
数组 (2)

数组操作

```
echo $weekday
echo ${weekday[0]}
echo ${weekday[*]}
echo ${suids[@]}
echo ${#weekday}
echo ${#weekday[0]}
echo ${#weekday[*]}
echo ${#users[*]}
users[4]="jack"; users[6]="mike"
unset users[4]          # 删除 users 数组的 4 号元素
unset users             # 删除 users 数组
a=100; echo ${a[0]}     # 变量其实是仅包含一个元素的数组
```

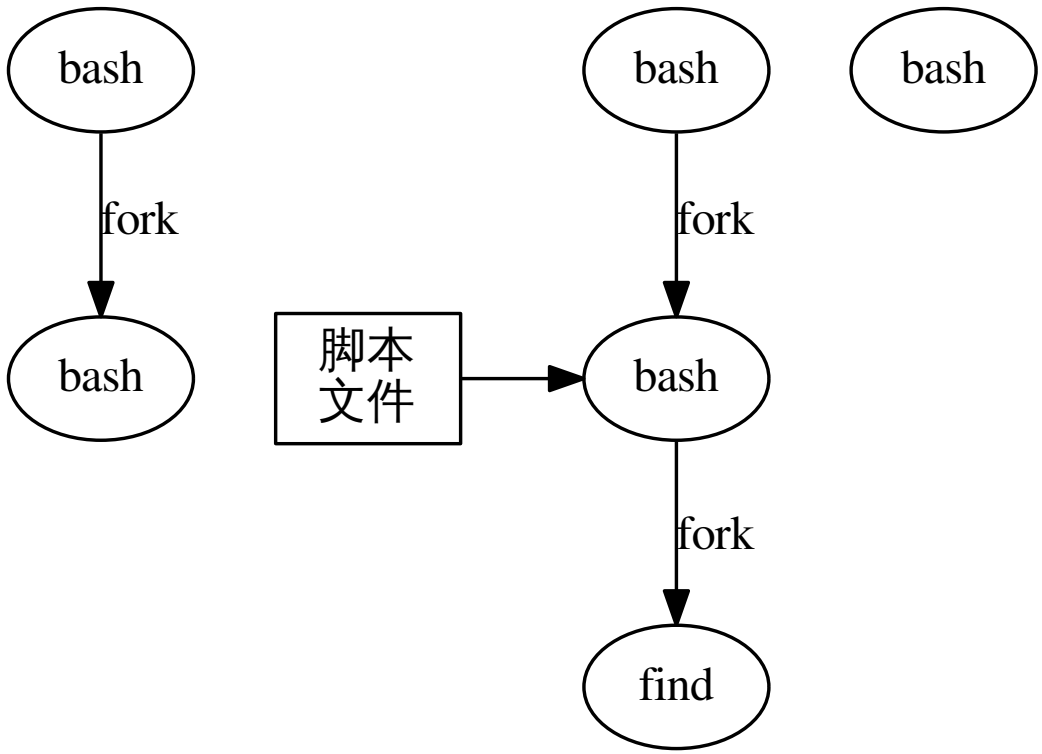
shell 命令的执行过程

- 内部命令本身就是 shell 进程的一部分，所以执行内部命令无需启动新进程。
- shell 需要创建一个进程来执行外部命令，并等待其结束。



shell 脚本的执行过程

- 当前 shell 创建一个子 shell 并让子 shell 依次执行 shell 脚本中的命令，子 shell 执行完脚本中所有命令后结束，父 shell 结束等待状态，开始重新执行。



exec 命令 (1)

- 功能 1：执行新进程并用新进程取代当前进程

示例

```
cat excmd
uname -a
exec date
echo "This line is never displayed"

exec_excmd # 由于 exec 不返回到调用位置，因此其后的命令将无法
```

exec 命令 (2)

功能 2：打开和关闭文件描述符

```
#bash 最多允许同时使用 10 个文件描述符 (n=0~9)
exec [n=0]<file   # 为标准输入重定向打开 file
exec [n=1]>file   # 为标准输出重定向 (覆盖) 打开 file
exec [n=1]>>file  # 为标准输出重定向 (追加) 打开 file
exec n<>file      # 为标准输入输出重定向打开 file
cmd <&n           # 输入重定向到文件描述符 n
cmd >&n           # 输出重定向 (覆盖) 到文件描述符 n
cmd >>&n          # 输出重定向 (追加) 到文件表述符 n
exec n>&m         # 把 m 复制到 n, 将输出同时重定向到 m 和 n
exec <&-          # 关闭标准输入
exec >&-          # 关闭标准输出
exec n<&-         # 关闭重定向为标准输入的文件描述符 n
exec n>&-         # 关闭重定向为标准输出的文件描述符 n
```

exec 命令 (3)

diff2：比较两个文本文件是否内容相同

```
#!/bin/bash
if [ "$#" -ne 2 ]; then
  echo "usage: `basename $0` file1 file2" 1>&2; exit 1
elif [ ! -f "$1" ]; then
  echo "$1 is not a regular file" 1>&2; exit 2
elif [ ! -f "$2" ]; then
  echo "$2 is not a regular file" 1>&2; exit 3
fi
```

exec 命令 (4)

diff2：(续 1)

```
file1="$1"; file2="$2"; i=1
exec 3<"$file1"; exec 4<"$file2"
while read line1 0<&3; do
  if read line2 0<&4; then
    if [ "$line1" != "$line2" ]; then
      echo "different at line $i" 1>&2; exit 1
    fi
  else
    echo "$file1 is longer than $file2" 1>&2; exit 2
  fi
  ((i++))
done
```

exec 命令 (5)

diff2：(续 2)

```
if read line2 0<&4; then
  echo "$file1 is shorter than $file2" 1>&2; exit 3
else
  echo "$file1 and $file2 are the same"; exit 0
fi
exec 3<&-; exec 4<&-

diff2 file1 file2
```

trap 命令 (1)

- trap 用于捕捉信号并指定收到信号时所执行的操作。

示例

```
trap -l # 查看信号列表, 同 kill -l
trap 'echo "Ctrl-C disabled."' 2 # 捕获信号 2 并执行指定命令
trap -p # 查看当前信号捕获情况
trap    # 同上
trap '' TERM INT # 捕获并忽略信号 2 和 15
trap - 1 2 15    # 恢复信号 1, 2, 15 的系统默认处理
trap 1 2 15      # 同上
```

注意

1. 比较重要的、能够捕捉的常用信号：0, 1, 2, 3, 15, 20
2. 不能捕捉2个信号：9, 19
3. 不应捕捉4个信号：10, 12, 17, 30

trap 命令 (2)

leave：短暂离开时锁住终端

```
#!/bin/bash
# 锁屏
trap '' 1 2 3 15 20   # 忽略信号
clear
stty -echo            # 关闭回显
echo -e "Enter your password: \c"
read pass1
echo -e "\nEnter your password again: \c"
read pass2
if [ "$pass1" != "$pass2" ]; then
  echo "Passwords do not match." 1>&2
  stty echo           # 恢复回显
  exit 1
fi
```

trap 命令 (3)

leave：(续)

```
# 解锁
until [ "$code" = "$pass1" ]; do
  clear
  echo -n "Enter the password: "
  read code
done
clear
echo 'Welcome back!'
stty echo                # 恢复回显
exit 0
```

eval 命令 (1)

- eval args...

  eval 将所有参数合成一个字符串，并将得到的字符串作为命令执行。

示例 1

```
x=100; y=x
echo $y
echo '$'$y
eval echo '$'$y # 等价于 echo $x, 同 echo ${!y}
eval $y=200     # 等价于 x=200
echo $x
```

lastarg：打印传递给脚本的最后一个参数

```
#!/bin/bash
eval echo '$'$#
```

echo 高级输出 (1)

输出彩色字符和彩色背景

```
echo -e "\033[31;46mhello world\033[0m"
#\033 代表 ESC，而“ESC[参数 m”可用来设置显示属性
#31-前景色为红色，46-背景色为青色
#0-恢复默认设置

#\033 也可写成\e
echo -e "\e[33;44mhello world\e[0m"
#33-前景色为棕色，44-背景色为蓝色
```

echo 高级输出 (2)

常用显示属性值

0 恢复默认值
1 加粗
4 下划线
7 反白显示

30 黑色(前景)	40 黑色(背景)
31 红色(前景)	41 红色(背景)
32 绿色(前景)	42 绿色(背景)
33 棕色(前景)	43 棕色(背景)
34 蓝色(前景)	44 蓝色(背景)
35 品红(前景)	45 品红(背景)
36 青色(前景)	46 青色(背景)
37 白色(前景)	47 白色(背景)

```
man console_codes   # 查看更多属性
```

tput 命令 (1)

tput 命令可以通过 terminfo 数据库调用终端功能

```
bel       # 响铃
cols      # 打印屏幕列数
clear     # 清屏
smso      # 开始突出显示模式
rmso      # 结束突出显示模式
smul      # 开始下划线模式
rmul      # 结束下划线模式
rev       # 反白显示
ed        # 从光标位置到屏幕底部清屏
el        # 从光标位置到行尾清除字符
sgr0      # 关闭所有属性
bold      # 粗体显示
cup r c   # 把光标移到 r 行 c 列
```

tput 命令 (2)

示例

```
tput clear; tput cup 10 20; echo "hello"
bell=`tput bel`; echo $bell
line=`tput smul`; offline=`tput rmul`
tput clear
tput cup 10 20; echo "${line}hello${offline}"

ls
tput cup 5 0; tput ed

tput -S <<!
clear
cup 10 10
!
```

选项与参数处理 (1)

- getopts 命令：解析命令行选项，检查选项合法性。

示例：handleopts

```
cat handleopts
#!/bin/bash
while getopts o:r:nt opt; do
  case "$opt" in
    o) output_file="$OPTARG";;
    r) report_file="$OPTARG";;
    n) number_option="yes";;
    t) title="no";;
    *) exit 1;
  esac
  echo "option: $opt next: $OPTIND"
done
```

选项与参数处理 (2)

示例：handleopts(续)

```
echo "Output_file = $output_file"
echo "Report_file = $report_file"
echo "Number_option = ${number_option:-no}"
echo "Title = ${title:-yes}"
echo "Arguments before shift: $*"
shift `expr $OPTIND - 1`
echo "Arguments after shift: $*"
exit 0
```

选项与参数选项 (3)

说明

1. while 循环通过调用 getopts 每次获取一个选项并存入字符串变量 opt 中，直至读取不到新的选项或读取到–选项。
2. o:r:nt 表示接受 4 个选项，其中 o 和 r 带选项参数
3. $OPTARG 表示当前选项的参数
4. $OPTIND 初值为 1，每处理完一个选项后，指向下一个要处理的选项/参数位置。处理完所有选项后，将指向第一个非选项参数的位置参数，即第一个命令参数。如果位置参数的值为–，则忽略之。

选项与参数处理 (4)

测试：handleopts

```
handleopts -tn -o file.out -r file.rep unit.txt
handleopts -t -n -o file.out -r file.rep unit.txt
handleopts unit.txt
handleopts -tn -o file.out -r     #:-(
```

给 awk 传递参数 (1)

field n：打印输入的第 n 个字段

```
cat field1
#!/bin/bash
awk "{ print \$ $1 }"   # 必须用双引号 (两个 $ 本应紧挨着)

cat field2
#!/bin/bash
awk '{ print $'$1' }'

who | field1 1
who | field2 2
```

给 awk 传递参数 (2)

addup n：将输入的第 n 列求和

```
cat addup n
#!/bin/bash
awk '{ s+=$'$1' }
  END{ print s }'

cat score.list | addup 2
```

给 awk 传递参数 (3)

sumup m n：将输入的第 m 至 n 列分别求和，并累加这些和

```
cat sumup
#!/bin/bash
awk '
BEGIN { m='$1'; n='$2' }
      { for (i=m; i<=n; i++) sum[i]+=$i }
  END { for (i=m; i<=n; i++) {
          printf("sum[%d] = %d\n", i, sum[i])
          total += sum[i]
          }
        printf("total = %d\n", total)
      }'

sumup 2 5 <score.list
```

命令组

(cmd1;cmd2;...) 在子 shell 中执行

```
x=3
(y=5; let x+=y; echo $x)
echo $x
```

{ cmd1;cmd2;...; } 在当前 shell 中执行

```
x=3
{ y=5; let x+=y; echo $x; } # 注意空格与分号
echo $x
```

脚本调试 (1)

shell 的-v 和-x 选项

-v选项：shell将显示每条原始命令及其执行结果。
-x选项：shell将显示每条扩展后的命令行及其执行结果。

示例

```
cat hour
#!/bin/bash
h=`date +%H`
echo $h

sh -v hour
sh -x hour
```

脚本调试 (2)

在脚本内部开启和关闭调试

set -v/x: 开启调试
set +v/x: 关闭调试

示例

```
cat hour2
#!/bin/bash
set -v    # 开启调试
h=`date +%H`
set +v    # 关闭调试
echo $h
```

脚本调试 (3)

- 利用 trap 命令进行调试 (1)
  - 伪信号：由 shell 产生的信号，信号则是由操作系统产生的
- 伪信号列表

伪信号	产生时机
EXIT(0)	从函数或脚本中退出
RETURN	调用函数后或用. 命令执行其他脚本之后
ERR	某条命令返回非 0 状态 (不成功) 时
DEBUG	脚本中每条命令执行之前

脚本调试 (4)

- 利用 trap 命令进行调试 (2)

示例 1：trap-debug1

```
#!/bin/bash
trap 'echo "EXIT: min=$min max=$max"' EXIT
min=$1
max=$2
if [ $min -gt $max ]; then
  exit 1
fi

trap-debug1 6 8
```

脚本调试 (5)

- 利用 trap 命令进行调试 (3)

示例 2：trap-debug2

```
#!/bin/bash
trap 'echo "ERR($LINENO): name=$name"' ERR
name=`grep william /etc/passwd`
exit 0

trap-debug2
```

脚本调试 (6)

- 利用 trap 命令进行调试 (4)

示例 3：trap-debug3

```
#!/bin/bash
trap 'echo "DEBUG($LINENO): var=$var"' DEBUG
var=29
let var=var*2
exit 0

trap-debug3
```

脚本调试 (7)

- 可以结合调试变量和 echo 语句进行调试

示例

```
cat sum2n
#!/bin/bash
n=${1:-100}
for i in $(seq $n); do
  let sum+=i
  [ "$debug" = "on" ] && echo "DEBUG: i=$i sum=$sum"
done
echo "1+...+$n=$sum"

sum2n 10                # 正常运行
debug=on sum2n 10       # 调试运行
```

综合实例

- 个人书籍管理系统

书籍信息字段

```
ID          #编号 (5位数字,如00023)
Title       #书名
Author      #作者
Class       #类别 (os-operating system
                   se-software engineering
                   pl-programming language
                   cn-computer networks
                   db-database
                   ob-other books)
state       #状态 (in-未借出,out-已借出)
bname       #借阅人
btime       #借出时间
```

shell 编程实践

试一试

1. 请试一试完善和改进现有个人书籍管理系统
2. 请试一试用 shell 开发一个其他信息管理系统