Tmux教程

默认热键为`Ctrl + b`此教程是修改为`Ctrl + a`了

```
功能：
    (1) 分屏。
    (2) 允许断开Terminal连接后，继续运行进程。
结构：
    一个tmux可以包含多个session，一个session可以包含多个window，一个window可以包含多个pane。
    实例：
        tmux:
            session 0:
                window 0:
                    pane 0
                    pane 1
                    pane 2
                    ...
                window 1
                window 2
                ...
            session 1
            session 2
            ...
操作：
    (1) tmux：新建一个session，其中包含一个window，window中包含一个pane，pane里打开了一个shell对话框。
    (2) 按下Ctrl + a后手指松开，然后按%：将当前pane左右平分成两个pane。
    (3) 按下Ctrl + a后手指松开，然后按"（注意是双引号"）：将当前pane上下平分成两个pane。
    (4) Ctrl + d：关闭当前pane；如果当前window的所有pane均已关闭，则自动关闭window；如果当前session的所有window均已关闭，则自动关闭session。
    (5) 鼠标点击可以选pane。
    (6) 按下ctrl + a后手指松开，然后按方向键：选择相邻的pane。
    (7) 鼠标拖动pane之间的分割线，可以调整分割线的位置。
    (8) 按住ctrl + a的同时按方向键，可以调整pane之间分割线的位置。
    (9) 按下ctrl + a后手指松开，然后按z：将当前pane全屏/取消全屏。
    (10) 按下ctrl + a后手指松开，然后按d：挂起当前session。
    (11) tmux a：打开之前挂起的session。
    (12) 按下ctrl + a后手指松开，然后按s：选择其它session。
        方向键 —— 上：选择上一项 session/window/pane
        方向键 —— 下：选择下一项 session/window/pane
        方向键 —— 右：展开当前项 session/window
        方向键 —— 左：闭合当前项 session/window
    (13) 按下Ctrl + a后手指松开，然后按c：在当前session中创建一个新的window。
    (14) 按下Ctrl + a后手指松开，然后按w：选择其他window，操作方法与(12)完全相同。
    (15) 按下Ctrl + a后手指松开，然后按PageUp：翻阅当前pane内的内容。
    (16) 鼠标滚轮：翻阅当前pane内的内容。
    (17) 在tmux中选中文本时，需要按住shift键。（仅支持Windows和Linux，不支持Mac，不过该操作并不是必须的，因此影响不大）
    (18) tmux中复制/粘贴文本的通用方式：
        (1) 按下Ctrl + a后松开手指，然后按[
        (2) 用鼠标选中文本，被选中的文本会被自动复制到tmux的剪贴板
        (3) 按下Ctrl + a后松开手指，然后按]，会将剪贴板中的内容粘贴到光标处
```

Vim教程

功能：

(1) 命令行模式下的文本编辑器。

(2) 根据文件扩展名自动判别编程语言。支持代码缩进、代码高亮等功能。

(3) 使用方式：vim filename

如果已有该文件，则打开它。

如果没有该文件，则打开个一个新的文件，并命名为filename

模式：

(1) 一般命令模式

默认模式。命令输入方式：类似于打游戏放技能，按不同字符，即可进行不同操作。可以复制、粘贴、删除文本等。

(2) 编辑模式

在一般命令模式里按下i，会进入编辑模式。

按下ESC会退出编辑模式，返回到一般命令模式。

(3) 命令行模式

在一般命令模式里按下:/?三个字母中的任意一个，会进入命令行模式。命令行在最下面。

可以查找、替换、保存、退出、配置编辑器等。

操作：

(1) i：进入编辑模式

(2) ESC：进入一般命令模式

(3) h 或 左箭头键：光标向左移动一个字符

(4) j 或 向下箭头：光标向下移动一个字符

(5) k 或 向上箭头：光标向上移动一个字符

(6) l 或 向右箭头：光标向右移动一个字符

(7) n<Space>：n表示数字，按下数字后再按空格，光标会向右移动这一行的n个字符

(8) 0 或 功能键[Home]：光标移动到本行开头

(9) $ 或 功能键[End]：光标移动到本行末尾

(10) G：光标移动到最后一行

(11) :n 或 nG：n为数字，光标移动到第n行

(12) gg：光标移动到第一行，相当于1G

(13) n<Enter>：n为数字，光标向下移动n行

(14) /word：向光标之下寻找第一个值为word的字符串。

(15) ?word：向光标之上寻找第一个值为word的字符串。

(16) n：重复前一个查找操作

(17) N：反向重复前一个查找操作

(18) :n1,n2s/word1/word2/g：n1与n2为数字，在第n1行与n2行之间寻找word1这个字符串，并将该字符串替换为word2

(19) :1,$s/word1/word2/g：将全文的word1替换为word2

(20) :1,$s/word1/word2/gc：将全文的word1替换为word2，且在替换前要求用户确认。

(21) v：选中文本

(22) d：删除选中的文本

(23) dd: 删除当前行

(24) y：复制选中的文本

(25) yy: 复制当前行

(26) p: 将复制的数据在光标的下一行/下一个位置粘贴

(27) u：撤销

(28) Ctrl + r：取消撤销

(29) 大于号 >：将选中的文本整体向右缩进一次

(30) 小于号 <：将选中的文本整体向左缩进一次

(31) :w 保存

(32) :w! 强制保存

(33) :q 退出

(34) :q! 强制退出

(35) :wq 保存并退出

```
    (36) :set paste 设置成粘贴模式，取消代码自动缩进
    (37) :set nopaste 取消粘贴模式，开启代码自动缩进
    (38) :set nu 显示行号
    (39) :set nonu 隐藏行号
    (40) gg=G：将全文代码格式化
    (41) :noh 关闭查找关键词高亮
    (42) Ctrl + q：当vim卡死时，可以取消当前正在执行的命令
异常处理：
    每次用vim编辑文件时，会自动创建一个.filename.swp的临时文件。
    如果打开某个文件时，该文件的swp文件已存在，则会报错。此时解决办法有两种：
        (1) 找到正在打开该文件的程序，并退出
        (2) 直接删掉该swp文件即可
```

Shell语法

概论

shell是我们通过命令行与操作系统沟通的语言。

shell脚本可以直接在命令行中执行，也可以将一套逻辑组织成一个文件，方便复用。

Linux中常见的shell脚本有很多种，常见的有：

- Bourne Again Shell(`/bin/bash`)
- C Shell(`/usr/bin/csh`)
- K Shell(`/usr/bin/ksh`)
- ...

Linux系统中一般默认使用bash。
文件开头需要写`#! /bin/bash`，指明bash为脚本解释器。

注释

- 单行注释

```
# 这是一行注释

echo 'Hello World'  #  这也是注释
```

- 多行注释

```
:<<EOF
第一行注释
第二行注释
第三行注释
EOF

# 其中EOF可以换成其它任意字符串

:<<abc
第一行注释
第二行注释
第三行注释
```

```
abc

:<<!
第一行注释
第二行注释
第三行注释
!
```

变量

- **定义变量**

```
name1='yxc'  # 单引号定义字符串
name2="yxc"  # 双引号定义字符串
name3=yxc    # 也可以不加引号，同样表示字符串
```

- **使用变量**

使用变量，需要加上`$`符号，或者`${}`符号。花括号是可选的，主要为了帮助解释器识别变量边界

```
name=yxc
echo $name  # 输出yxc
echo ${name}  # 输出yxc
echo ${name}acwing  # 输出yxcacwing
```

- **只读变量**

使用`readonly`或者`declare`可以将变量变为只读

```
name=yxc
readonly name
declare -r name  # 两种写法均可

name=abc  # 会报错，因为此时name只读
```

- **删除变量**

`unset`可以删除变量

```
name=yxc
unset name
echo $name  # 输出空行
```

- **变量类型**

1. 局部变量

   子进程不能访问

2. 环境变量（全局）

   子进程可以访问

局部变量改成环境变量

```
acs@9e0ebfcd82d7:~$ name=yxc  # 定义变量
acs@9e0ebfcd82d7:~$ export name  # 第一种方法
acs@9e0ebfcd82d7:~$ declare -x name  # 第二种方法
```

环境变量改成局部变量

```
acs@9e0ebfcd82d7:~$ export name=yxc  # 定义环境变量
acs@9e0ebfcd82d7:~$ declare +x name  # 改为自定义变量
```

- **字符串**

  字符串可以用单引号，也可以用双引号，也可以不用引号

  单引号与双引号的区别：

  - 单引号中的内容会原样输出，不会执行、不会取变量
  - 双引号中的内容可以执行、可以取变量

```
name=yxc  # 不用引号
echo 'hello, $name \"hh\"'  # 单引号字符串，输出 hello, $name \"hh\"
echo "hello, $name \"hh\""  # 双引号字符串，输出 hello, yxc "hh"
```

获取字符串长度

```
name="yxc"
echo ${#name}  # 输出3
```

提取子串

```
name="hello, yxc"
echo ${name:0:5}  # 提取从0开始的5个字符
```

默认变量

- **文件参数变量**

  在执行shell脚本时，可以向脚本传递参数。`$1`是第一个参数，`$2`是第二个参数，以此类推。

  特殊的，`$0`是文件名（包含路径）。

```
test.sh
#! /bin/bash

echo "文件名："$0
echo "第一个参数："$1
echo "第二个参数："$2
echo "第三个参数："$3
echo "第四个参数："$4
```

```
acs@9e0ebfcd82d7:~$ chmod +x test.sh
acs@9e0ebfcd82d7:~$ ./test.sh 1 2 3 4
文件名：./test.sh
第一个参数：1
第二个参数：2
第三个参数：3
第四个参数：4
```

如果参数过多可以使用`shift`移动变量

- **其他参数相关变量**

参数	说明
`$#`	代表文件传入的参数个数，如上例中值为4
`$*`	由所有参数构成的用空格隔开的字符串，如上例中值为`"$1 $2 $3 $4"`
`$@`	每个参数分别用双引号括起来的字符串，如上例中值为`"$1" "$2" "$3" "$4"`
`$$`	脚本当前运行的进程ID
`$?`	上一条命令的退出状态（注意不是`stdout`，而是`exit code`）0表示正常退出，其他值表示错误
`$(command)`	返回command这条命令的`stdout`（可嵌套）
`command`	返回command这条命令的`stdout`（不可嵌套）

数组

数组中可以存放多个不同类型的值，只支持一维数组，初始化时不需要指明数组大小。
数组下标从0开始。

- **定义**

```
array=(1 abc "def" yxc)

# or

array[0]=1
array[1]=abc
array[2]="def"
array[3]=yxc
```

- **读取数组中某个元素的值**

```
# ${array[index]}

array=(1 abc "def" yxc)
echo ${array[0]}
echo ${array[1]}
echo ${array[2]}
echo ${array[3]}
```

- **读取整个数组**

```
# ${array[@]}   第一种写法
# ${array[*]}   第二种写法

array=(1 abc "def" yxc)

echo ${array[@]}  # 第一种写法
echo ${array[*]}  # 第二种写法
```

- **数组长度**

  类似于字符串

```
# ${#array[@]}   第一种写法
# ${#array[*]}   第二种写法

array=(1 abc "def" yxc)

echo ${#array[@]}  # 第一种写法
echo ${#array[*]}  # 第二种写法
```

expr命令

`expr`命令用于求表达式的值，格式为：

```
expr 表达式
```

- 用空格隔开每一项
- 用反斜杠放在shell特定的字符前面（发现表达式运行错误时，可以试试转义）
- 对包含空格和其他特殊字符的字符串要用引号括起来
- `expr`会在`stdout`中输出结果。如果为逻辑关系表达式，则结果为真，`stdout`为`1`，否则为`0`。
- `expr`的`exit code`：如果为逻辑关系表达式，则结果为真，`exit code`为`0`，否则为`1`。

字符串表达式

- `length STRING`

  返回`STRING`的长度

- `index STRING CHARSET`

  `CHARSET`中任意单个字符在`STRING`中最前面的字符位置，下标从`1`开始。如果在`STRING`中完全不存在`CHARSET`中的字符，则返回`0`。

- `substr STRING POSITION LENGTH`
  返回`STRING`字符串中从`POSITION`开始，长度最大为`LENGTH`的子串。如果`POSITION`或`LENGTH`为负数，`0`或非数值，则返回空字符串。

```
str="Hello World!"

echo `expr length "$str"`  # ``不是单引号，表示执行该命令，输出12
echo `expr index "$str" awd`  # 输出7，下标从1开始
echo `expr substr "$str" 2 3`  # 输出 ell
```

整数表达式

`expr`支持普通的算术操作，算术表达式优先级低于字符串表达式，高于逻辑关系表达式。

- `+ -`
  加减运算。两端参数会转换为整数，如果转换失败则报错
- `* / %`
  乘，除，取模运算。两端参数会转换为整数，如果转换失败则报错
- `()`
  可以该表优先级，但需要用反斜杠转义

```
a=3
b=4

echo `expr $a + $b`  # 输出7
echo `expr $a - $b`  # 输出-1
echo `expr $a \* $b`  # 输出12，*需要转义
echo `expr $a / $b`  # 输出0，整除
echo `expr $a % $b` # 输出3
echo `expr \( $a + 1 \) \* \( $b + 1 \)`  # 输出20，值为(a + 1) * (b + 1)
```

逻辑关系表达式

- `|`
  如果第一个参数非空且非0，则返回第一个参数的值，否则返回第二个参数的值，但要求第二个参数的值也是非空或非0，否则返回0。如果第一个参数是非空或非0时，不会计算第二个参数。
- `&`
  如果两个参数都非空且非0，则返回第一个参数，否则返回0。如果第一个参为0或为空，则不会计算第二个参数。
- `< <= = == != >= >`
  比较两端的参数，如果为true，则返回1，否则返回0。”==”是”=”的同义词。”expr”首先尝试将两端参数转换为整数，并做算术比较，如果转换失败，则按字符集排序规则做字符比较。
- `()` 可以该表优先级，但需要用反斜杠转义

```
a=3
b=4

echo `expr $a \> $b`  # 输出0，>需要转义
echo `expr $a '<' $b`  # 输出1，也可以将特殊字符用引号引起来
```

```
echo `expr $a '>=' $b`  # 输出0
echo `expr $a \<\= $b`  # 输出1

c=0
d=5

echo `expr $c \& $d`  # 输出0
echo `expr $a \& $b`  # 输出3
echo `expr $c \| $d`  # 输出5
echo `expr $a \| $b`  # 输出3
```

read命令

`read`命令用于从标准输入中读取单行数据。当读到文件结束符时，`exit code`为1，否则为0。

参数说明

- `-p`: 后面可以接提示信息
- `-t`: 后面跟秒数，定义输入字符的等待时间，超过等待时间后会自动忽略此命令

```
acs@9e0ebfcd82d7:~$ read name  # 读入name的值
acwing yxc  # 标准输入
acs@9e0ebfcd82d7:~$ echo $name  # 输出name的值
acwing yxc  #标准输出
acs@9e0ebfcd82d7:~$ read -p "Please input your name: " -t 30 name  # 读入name的值，等待时间30秒
Please input your name: acwing yxc  # 标准输入
acs@9e0ebfcd82d7:~$ echo $name  # 输出name的值
acwing yxc  # 标准输出
```

echo命令

`echo`用于输出字符串。命令格式:

```
echo STRING
```

显示普通字符串

```
echo "Hello AC Terminal"
echo Hello AC Terminal  # 引号可以省略
```

显示转义字符

```
echo "\"Hello AC Terminal\""  # 注意只能使用双引号，如果使用单引号，则不转义
echo \"Hello AC Terminal\"  # 也可以省略双引号
```

显示变量

```
name=yxc
echo "My name is $name"  # 输出 My name is yxc
```

显示换行

```
echo -e "Hi\n"  # -e 开启转义
echo "acwing"

# Hi
#
# acwing
```

显示不换行

```
echo -e "Hi \c" # -e 开启转义 \c 不换行
echo "acwing"

# Hi acwing
```

显示结果定向至文件

```
echo "Hello World" > output.txt  # 将内容以覆盖的方式输出到output.txt中
```

原样输出字符串，不进行转义或取变量(用单引号)

```
name=acwing
echo '$name\"'

# $name\"
```

显示命令的执行结果

```
echo `date`

# Wed Sep 1 11:45:33 CST 2021
```

printf命令

`printf`命令用于格式化输出，类似于`C/C++`中的`printf`函数

默认不会在字符串末尾添加换行符

`printf format-string [arguments...]`

```
printf "%10d.\n" 123  # 占10位，右对齐
printf "%-10.2f.\n" 123.123321  # 占10位，保留2位小数，左对齐
printf "My name is %s\n" "yxc"  # 格式化输出字符串
printf "%d * %d = %d\n"  2 3 `expr 2 \* 3` # 表达式的值作为参数
```

```
       123.
123.12    .
My name is yxc
2 * 3 = 6
```

test命令与判断符号[]

逻辑运算符`&&`和`||`

- `&&` 表示与，`||` 表示或
- 二者具有短路原则：
  `expr1 && expr2`：当expr1为假时，直接忽略expr2
  `expr1 || expr2`：当expr1为真时，直接忽略expr2
- 表达式的`exit code`为0，表示真；为非零，表示假。（与C/C++中的定义相反）

test命令

在命令行中输入`man test`，可以查看`test`命令的用法。

`test`命令用于判断文件类型，以及对变量做比较。

`test`命令用`exit code`返回结果，而不是使用`stdout`。0表示真，非0表示假

```
test 2 -lt 3  # 为真，返回值为0
echo $?  # 输出上个命令的返回值，输出0
```

```
acs@9e0ebfcd82d7:~$ ls  # 列出当前目录下的所有文件
homework  output.txt  test.sh  tmp
acs@9e0ebfcd82d7:~$ test -e test.sh && echo "exist" || echo "Not exist"
exist  # test.sh 文件存在
acs@9e0ebfcd82d7:~$ test -e test2.sh && echo "exist" || echo "Not exist"
Not exist  # testh2.sh 文件不存在
```

文件类型判断

```
test -e filename  # 判断文件是否存在
```

测试参数	代表意义
-e	文件是否存在
-f	是否为文件
-d	是否为目录

文件权限判断

```
test -r filename  # 判断文件是否可读
```

测试参数	代表意义
-r	文件是否可读
-w	文件是否可写
-x	文件是否可执行
-s	是否为非空文件

整数间的比较

```
test $a -eq $b  # a是否等于b
```

测试参数	代表意义
-eq	a是否等于b
-ne	a是否不等于b
-gt	a是否大于b
-lt	a是否小于b
-ge	a是否大于等于b
-le	a是否小于等于b

字符串比较

测试参数	代表意义
test -z STRING	判断STRING是否为空，如果为空，则返回true
test -n STRING	判断STRING是否非空，如果非空，则返回true（-n可以省略）
test str1 == str2	判断str1是否等于str2
test str1 != str2	判断str1是否不等于str2

多重条件判定

测试参数	代表意义
-a	两条件是否同时成立
-o	两条件是否至少一个成立
!	取反。如 test ! -x file，当file不可执行时，返回true

判断符号[]

`[]`与`test`用法几乎一模一样，更常用于`if`语句中。另外`[[]]`是`[]`的加强版，支持的特性更多

```
[ 2 -lt 3 ]  # 为真，返回值为0
echo $?  # 输出上个命令的返回值，输出0
```

```
acs@9e0ebfcd82d7:~$ ls  # 列出当前目录下的所有文件
homework  output.txt  test.sh  tmp
acs@9e0ebfcd82d7:~$ [ -e test.sh ] && echo "exist" || echo "Not exist"
exist  # test.sh 文件存在
acs@9e0ebfcd82d7:~$ [ -e test2.sh ] && echo "exist" || echo "Not exist"
Not exist  # testh2.sh 文件不存在
```

notice:

- `[]`内的每一项都要用空格隔开
- 中括号内的变量，最好用双引号括起来
- 中括号内的常数，最好用单或双引号括起来

```
name="acwing yxc"
[ $name == "acwing yxc" ]  # 错误，等价于 [ acwing yxc == "acwing yxc" ]，参数太多
[ "$name" == "acwing yxc" ]  # 正确
```

判断语句

if...then形式

```
if condition
then
    语句1
    语句2
    ...
fi
```

```
a=3
b=4

if [ "$a" -lt "$b" ] && [ "$a" -gt 2 ]
then
    echo ${a}在范围内
fi

# 3在范围内
```

单层if-else

```
if condition
then
    语句1
    语句2
    ...
else
    语句1
    语句2
    ...
fi
```

```
a=3
b=4

if ! [ "$a" -lt "$b" ]
then
    echo ${a}不小于${b}
else
    echo ${a}小于${b}
fi

# 3小于4
```

多层if-elif-elif-else

```
if condition
then
    语句1
    语句2
    ...
elif condition
then
    语句1
    语句2
    ...
elif condition
then
    语句1
    语句2
else
    语句1
    语句2
    ...
fi
```

```
a=4

if [ $a -eq 1 ]
then
    echo ${a}等于1
elif [ $a -eq 2 ]
then
    echo ${a}等于2
elif [ $a -eq 3 ]
then
```

```
    echo ${a}等于3
else
    echo 其他
fi

# 其他
```

case...esac形式

```
case $变量名称 in
    值1)
        语句1
        语句2
        ...
        ;;  # 类似于C/C++中的break
    值2)
        语句1
        语句2
        ...
        ;;
    *)  # 类似于C/C++中的default
        语句1
        语句2
        ...
        ;;
esac
```

```
a=4

case $a in
    1)
        echo ${a}等于1
        ;;
    2)
        echo ${a}等于2
        ;;
    3)
        echo ${a}等于3
        ;;
    *)
        echo 其他
        ;;
esac

# 其他
```

循环语句

for...in...do...done

```
for var in val1 val2 val3
do
    语句1
    语句2
    ...
done
```

示例1，输出a 2 cc，每个元素一行

```
for i in a 2 cc
do
    echo $i
done
```

示例2，输出当前路径下的所有文件名，每个文件名一行

```
for file in `ls`
do
    echo $file
done
```

示例3，输出1-10

```
for i in $(seq 1 10)
do
    echo $i
done
```

示例4，使用`{1..10}`或者`{a..z}`

```
for i in {a..z}
do
    echo $i
done
```

for ((...;...;...)) do...don

```
for ((expression; condition; expression))
do
    语句1
    语句2
done
```

示例，输出1-10，每个数占一行

```
for ((i=1; i<=10; i++))
do
    echo $i
done
```

while...do...done循环

命令格式

```
while condition
do
    语句1
    语句2
    ...
done
```

示例，文件结束符为Ctrl+d，输入文件结束符后read指令返回false。

```
while read name
do
    echo $name
done
```

until...do...done循环
当条件为真时结束。

命令格式：

```
until condition
do
    语句1
    语句2
    ...
done
```

示例，当用户输入yes或者YES时结束，否则一直等待读入。

```
until [ "${word}" == "yes" ] || [ "${word}" == "YES" ]
do
    read -p "Please input yes/YES to stop this program: " word
done
```

break命令
跳出当前一层循环，注意与C/C++不同的是：break不能跳出case语句。

示例

```
while read name
do
    for ((i=1;i<=10;i++))
    do
        case $i in
            8)
                break
                ;;
            *)
                echo $i
```

```
                ;;
        esac
    done
done
```

continue命令
跳出当前循环。

示例：

```
for ((i=1;i<=10;i++))
do
    if [ `expr $i % 2` -eq 0 ]
    then
        continue
    fi
    echo $i
done
```

该程序输出1-10中的所有奇数

函数

bash中的函数类似于C/C++中的函数，但return的返回值与C/C++不同，返回的是exit code，取值为0-255，0表示正常结束。

如果想获取函数的输出结果，可以通过`echo`输出到`stdout`中，然后通过`$(function_name)`来获取`stdout`中的结果。

函数的return值可以通过$?来获取。

命令格式：

```
[function] func_name() {  # function关键字可以省略
    语句1
    语句2
    ...
}
```

不获取`return`值和`stdout`值
示例

```
func() {
    name=yxc
    echo "Hello $name"
}
输出结果：
```

```
Hello yxc
```

获取`return`值和`stdout`值
不写`return`时，默认`return 0`。

示例

```
func() {
    name=yxc
    echo "Hello $name"

    return 123
}

output=$(func)
ret=$?

echo "output = $output"
echo "return = $ret"
```

输出结果：

```
output = Hello yxc
return = 123
```

函数的输入参数
在函数内，`$1`表示第一个输入参数，`$2`表示第二个输入参数，依此类推。

注意：函数内的`$0`仍然是文件名，而不是函数名。

示例：

```
func() {  # 递归计算 $1 + ($1 - 1) + ($1 - 2) + ... + 0
    word=""
    while [ "${word}" != 'y' ] && [ "${word}" != 'n' ]
    do
        read -p "要进入func($1)函数吗？请输入y/n： " word
    done

    if [ "$word" == 'n' ]
    then
        echo 0
        return 0
    fi

    if [ $1 -le 0 ]
    then
        echo 0
        return 0
    fi

    sum=$(func $(expr $1 - 1))
    echo $(expr $sum + $1)
}

echo $(func 10)
```

输出结果：

```
55
```

函数内的局部变量
可以在函数内定义局部变量，作用范围仅在当前函数内。

可以在递归函数中定义局部变量。

命令格式：

```
local 变量名=变量值
```

例如：

```
#! /bin/bash

func() {
    local name=yxc
    echo $name
}
func

echo $name
```

输出结果：

```
yxc
```

第一行为函数内的name变量，第二行为函数外调用name变量，会发现此时该变量不存在。

exit命令

`exit`命令用来退出当前shell进程，并返回一个退出状态；使用`$?`可以接收这个退出状态。

`exit`命令可以接受一个整数值作为参数，代表退出状态。如果不指定，默认状态值是 0。

`exit`退出状态只能是一个介于 0~255 之间的整数，其中只有 0 表示成功，其它值都表示失败。

```
#! /bin/bash

if [ $# -ne 1 ]  # 如果传入参数个数等于1，则正常退出；否则非正常退出。
then
    echo "arguments not valid"
    exit 1
else
    echo "arguments valid"
    exit 0
fi
```

```
acs@9e0ebfcd82d7:~$ chmod +x test.sh
acs@9e0ebfcd82d7:~$ ./test.sh acwing
arguments valid
acs@9e0ebfcd82d7:~$ echo $?  # 传入一个参数，则正常退出，exit code为0
0
acs@9e0ebfcd82d7:~$ ./test.sh
arguments not valid
acs@9e0ebfcd82d7:~$ echo $?  # 传入参数个数不是1，则非正常退出，exit code为1
1
```

文件重定向

每个进程默认打开3个文件描述符：

- `stdin`标准输入，从命令行读取数据，文件描述符为0
- `stdout`标准输出，向命令行输出数据，文件描述符为1
- `stderr`标准错误输出，向命令行输出数据，文件描述符为2
  可以用文件重定向将这三个文件重定向到其他文件中。

重定向命令列表

命令	说明
command > file	将`stdout`重定向到file中
command < file	将`stdin`重定向到file中
command >> file	将`stdout`以追加方式重定向到file中
command n> file	将文件描述符n重定向到file中
command n>> file	将文件描述符n以追加方式重定向到file中

输入和输出重定向

```
echo -e "Hello \c" > output.txt  # 将stdout重定向到output.txt中
echo "World" >> output.txt  # 将字符串追加到output.txt中

read str < output.txt  # 从output.txt中读取字符串

echo $str  # 输出结果：Hello World
```

同时重定向stdin和stdout
创建bash脚本：

```
#! /bin/bash

read a
read b

echo $(expr "$a" + "$b")
```

创建input.txt，里面的内容为：

```
3
4
```

执行命令：

```
acs@9e0ebfcd82d7:~$ chmod +x test.sh  # 添加可执行权限
acs@9e0ebfcd82d7:~$ ./test.sh < input.txt > output.txt  # 从input.txt中读取内容，将输出写入output.txt中
acs@9e0ebfcd82d7:~$ cat output.txt  # 查看output.txt中的内容
7
```

引入外部脚本

类似于C/C++中的include操作，bash也可以引入其他文件中的代码。

语法格式：

```
. filename  # 注意点和文件名之间有一个空格
```

或

```
source filename
```

示例
创建test1.sh，内容为：

```
#! /bin/bash

name=yxc  # 定义变量name
```

然后创建test2.sh，内容为：

```
#! /bin/bash

source test1.sh # 或 . test1.sh

echo My name is: $name  # 可以使用test1.sh中的变量
```

执行命令：

```
acs@9e0ebfcd82d7:~$ chmod +x test2.sh
acs@9e0ebfcd82d7:~$ ./test2.sh
My name is: yxc
```

Docker

镜像

1. `docker pull ubuntu:20.04`：拉取一个镜像
2. `docker images`：列出本地所有镜像
3. `docker image rm ubuntu:20.04 或 docker rmi ubuntu:20.04`：删除镜像`ubuntu:20.04`
4. `docker [container] commit CONTAINER IMAGE_NAME:TAG`：创建某个`container`的镜像
5. `docker save -o ubuntu_20_04.tar ubuntu:20.04`：将镜像`ubuntu:20.04`导出到本地文件`ubuntu_20_04.tar`中
6. `docker load -i ubuntu_20_04.tar`：将镜像`ubuntu:20.04`从本地文件`ubuntu_20_04.tar`中加载出来

容器

1. `docker [container] create -it ubuntu:20.04`：利用镜像`ubuntu:20.04`创建一个容器。
2. `docker ps -a`：查看本地的所有容器
3. `docker [container] start CONTAINER`：启动容器
4. `docker [container] stop CONTAINER`：停止容器
5. `docker [container] restart CONTAINER`：重启容器
6. `docker [contaienr] run -itd ubuntu:20.04`：创建并启动一个容器
7. `docker [container] attach CONTAINER`：进入容器
   先按`Ctrl-p`，再按`Ctrl-q`可以挂起容器
8. `docker [container] exec CONTAINER COMMAND`：在容器中执行命令
9. `docker [container] rm CONTAINER`：删除容器
10. `docker container prune`：删除所有已停止的容器
11. `docker export -o xxx.tar CONTAINER`：将容器`CONTAINER`导出到本地文件`xxx.tar`中
12. `docker import xxx.tar image_name:tag`：将本地文件`xxx.tar`导入成镜像，并将镜像命名为`image_name:tag`
13. `docker export/import`与`docker save/load`的区别：
14. `export/import`会丢弃历史记录和元数据信息，仅保存容器当时的快照状态
15. `save/load`会保存完整记录，体积更大
16. `docker top CONTAINER`：查看某个容器内的所有进程
17. `docker stats`：查看所有容器的统计信息，包括`CPU、内存、存储、网络等`信息
18. `docker cp xxx CONTAINER:xxx 或 docker cp CONTAINER:xxx xxx`：在本地和容器间复制文件
19. `docker rename CONTAINER1 CONTAINER2`：重命名容器
20. `docker update CONTAINER --memory 500MB`：修改容器限制