# Shell语法

## shell语法

## 概论

#### 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。

AC Terminal中的命令行可以看成是一个“shell脚本在逐行执行”。

Linux中常见的shell脚本有很多种，常见的有：

Bourne Shell(/usr/bin/sh或/bin/sh)

Bourne Again Shell(/bin/bash)

C Shell(/usr/bin/csh)

K Shell(/usr/bin/ksh)

zsh

…

Linux系统中一般默认使用bash，所以接下来讲解bash中的语法。

文件开头需要写#! /bin/bash，指明bash为脚本解释器。

#### 学习技巧

不要死记硬背，遇到含糊不清的地方，可以在AC Terminal里实际运行一遍。

#### 脚本示例

新建一个test.sh文件，内容如下：

#! /bin/bash

echo "Hello World!"

#### 运行方式

作为可执行文件

acs@9e0ebfcd82d7:~$ chmod +x test.sh # 使脚本具有可执行权限

acs@9e0ebfcd82d7:~$ ./test.sh # 当前路径下执行

Hello World! # 脚本输出

acs@9e0ebfcd82d7:~$ /home/acs/test.sh # 绝对路径下执行

Hello World! # 脚本输出

acs@9e0ebfcd82d7:~$ ~/test.sh # 家目录路径下执行

Hello World! # 脚本输出

用解释器执行

acs@9e0ebfcd82d7:~$ bash test.sh

Hello World! # 脚本输出

## 注释

### 单行注释

每行中#之后的内容均是注释。

# 这是一行注释

echo 'Hello World' # 这也是注释

### 多行注释

格式：

:<<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。例如：

:<<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

定义变量，不需要加$符号，例如：

等号一定不要空格，新学者常犯错误

name1='yxc' # 单引号定义字符串 不解析变量，直接原样输出内容

name2="yxc" # 双引号定义字符串 若内容中有命令、变量、特殊转义，解析出结果后再输出

name3=yxc # 也可以不加引号，同样表示字符串，也是类似双引号的工作方式

### 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

自定义变量（局部变量）

子进程不能访问的变量

环境变量（全局变量）

子进程可以访问的变量

自定义变量改成环境变量：bash会在当前的bash程序下新开一个bash子进程，ctrl+D退出该子进程

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

环境变量改为自定义变量：

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）。例如：

创建文件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

然后执行该脚本：

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

### 其它参数相关变量



``反引号和$()用于产生标准输出；$?用于产生exit code

## 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

数组用小括号表示，元素之间用空格隔开。例如：

array=(1 abc "def" yxc)

也可以直接定义数组中某个元素的值：

array[0]=1

array[1]=abc

array[2]="def"

array[3]=yxc

### 读取数组中某个元素的值

格式：

${array[index]}

例如：

array=(1 abc "def" yxc)

echo ${array[0]}

echo ${array[1]}

echo ${array[2]}

echo ${array[3]}

### 读取整个数组

格式：

${array[@]} # 第一种写法

${array[\*]} # 第二种写法

例如：

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

例如：

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

表达式说明：

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。

示例：

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。其中乘法需要转义

() 可以改变优先级，但需要用反斜杠转义

示例：

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)

也可以将特殊字符用单引号引起来

### 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

示例：

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单引号引起来

echo `expr $a '>=' $b` # 输出0

echo `expr $a \<\= $b` # 输出1

c=0

d=5

总结输出先判断真假，再根据短路效应判断输出stdout的具体值

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（默认情况为键盘）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

实例：

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 开启换行转义

echo "acwing"

输出结果：

Hi

acwing

### 显示不换行

echo -e "Hi \c" # -e 开启转义 \c 不换行

echo "acwing"

输出结果：

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\"'

输出结果

$name\"

### 显示命令的执行结果

echo `date`

输出结果：

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。

命令格式：

printf format-string [arguments...]

### 用法示例

脚本内容：

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

输出结果：

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。

例如：

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" 用短路原则实现？：三元运算符

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

命令格式：

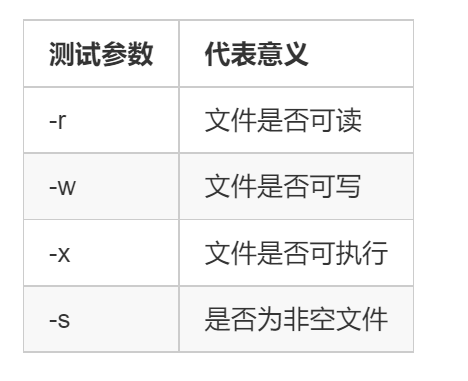
test -e filename # 判断文件是否存在



### 文件权限判断

命令格式：

test -r filename # 判断文件是否可读



### 整数间的比较

命令格式：

test $a -eq $b # a是否等于b

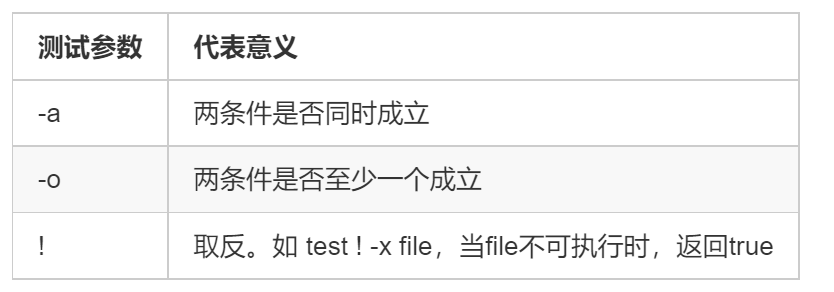


### 字符串比较

### 多重条件判定

命令格式：

test -r filename -a -x filename



### 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。

例如：

[ 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

注意：

[]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

例如：

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

类似于C/C++中的if-else语句。

#### 单层if

命令格式：

if condition

then

语句1

语句2

...

fi

示例：

a=3

b=4

if [ "$a" -lt "$b" ] && [ "$a" -gt 2 ]

then

echo ${a}在范围内

fi

输出结果：

3在范围内

#### 单层if-else

命令格式

if condition

then

语句1

语句2

...

else

语句1

语句2

...

fi

示例：

a=3

b=4

if ! [ "$a" -lt "$b" ]

then

echo ${a}不小于${b}

else

echo ${a}小于${b}

fi

输出结果：

3小于4

#### 多层if-elif-elif-else

命令格式

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

示例：

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

输出结果：

其他

#### case…esac形式

类似于C/C++中的switch语句。

命令格式

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

示例：

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

## 循环语句

### for…in…do…done

命令格式：

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…done

命令格式：

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

该示例每读入非EOF的字符串，会输出一遍1-7。

该程序可以输入Ctrl+d文件结束符来结束，也可以直接用Ctrl+c杀掉该进程。

### 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。

### 死循环的处理方式

如果AC Terminal可以打开该程序，则输入Ctrl+c即可。

否则可以直接关闭进程：

1.使用top命令找到进程的PID

2.输入kill -9 PID即可关掉此进程

## 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"

}

func

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。

示例：

创建脚本test.sh，内容如下：

#! /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

执行该脚本：

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



### 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

## 2. 创建作业 & 测试作业的正确性

homework 3 create 可以重新创建所有lesson\_3的作业

homework 3 create id 可以单独创建lesson\_3的第id个作业. e.g.

homework 3 create 0 可以只重新创建lesson\_3的第0个作业

homework 3 test 可以评测lesson\_3的所有作业

## 3. 作业

创建好作业后，先进入文件夹/home/acs/homework/lesson\_3/，然后：

(0) 进入homework\_0文件夹，编写自动完成lesson\_1作业的脚本helper.sh。要求：

[1] 当前目录下仅包含helper.sh

[2] helper.sh具有可执行权限

[3] 在任意路径依次执行下列命令后，lesson\_1的作业可以得到满分：

1) homework 1 create

2) /home/acs/homework/lesson\_3/homework\_0/helper.sh

(1) 进入homework\_1文件夹，编写脚本check\_file.sh。要求：

[1] 当前目录下仅包含check\_file.sh。

[2] check\_file.sh具有可执行权限。

[3] check\_file.sh接收一个传入参数。格式为 ./check\_file.sh file

[4] 判断传递参数，分别在标准输出中输出如下内容（不包括双引号）：

1) 如果传入参数个数不是1，则输出一行："arguments not valid"，然后退出，退出状态等于1。

2) 如果file文件不存在，则输出一行："not exist"，然后退出，退出状态等于2。

3) 如果file文件存在，则输出分别进行如下5个判断，然后退出，退出状态等于0。

1] 如果file为普通文件，则输出一行："regular file"

2] 如果file为目录（文件夹），则输出一行："directory"

3] 如果file具有可读权限，则输出一行："readable"

4] 如果file具有可写权限，则输出一行："writable"

5] 如果file具有可执行权限，则输出一行："executable"

(2) 进入homework\_2文件夹，编写脚本main.sh。要求：

[1] 当前目录下仅包含main.sh

[2] main.sh具有可执行权限

[3] 该文件从stdin(标准输入)中读取一个整数n

[4] 在stdout(标准输出)输出斐波那契数列的第n项。即：a[0] = 1, a[1] = 1, a[i] = a[i - 1] + a[i - 2], 求a[n]。

[5] 数据保证 0 <= n <= 20，脚本不需要判断n的合法性。

(3) 进入homework\_3文件夹，编写脚本main.sh。要求：

[1] 当前目录下仅包含main.sh

[2] main.sh具有可执行权限

[3] 该文件从stdin(标准输入)中读取两行整数n和m

[4] 在stdout(标准输出)中输出1~n的按字典序从小到大的顺序数第m个全排列，输出一行，用空格隔开所有数，行末可以有多余空格。

[5] 数据保证 1 <= n <= 10, 1 <= m <= min(100, n!)，脚本不需要判断数据的合法性。

(4) 进入homework\_4文件夹，编写脚本main.sh。要求：

[1] 当前目录下仅包含main.sh

[2] main.sh具有可执行权限

[3] main.sh接收两个传入参数。格式为 ./main.sh input\_file output\_file

[4] 从input\_file中读取一个正整数n，然后将前n个正整数的平方和写入output\_file中

[5] 数据保证 1 <= n <= 100，脚本不需要判断所有数据的合法性。

作业代码：

#! /bin/bash

#--------------homework\_0-----------------

homework 1 create 0

dir\_0="/home/acs/homework/lesson\_1/homework\_0/"

for i in dir\_a dir\_b dir\_c

do

mkdir $dir\_0$i

done

#--------------homework\_1-----------------

homework 1 create 1

dir\_1="/home/acs/homework/lesson\_1/homework\_1/"

for i in a.txt b.txt c.txt

do

cp "${dir\_1}${i}" "${dir\_1}${i}.bak"

done

#--------------homework\_2-----------------

homework 1 create 2

dir\_2="/home/acs/homework/lesson\_1/homework\_2/"

for i in a b c

do

mv "${dir\_2}${i}.txt" "${dir\_2}${i}\_new.txt"

done

#--------------homework\_3-----------------

homework 1 create 3

dir\_3="/home/acs/homework/lesson\_1/homework\_3/"

for i in a.txt b.txt c.txt

do

mv "${dir\_3}dir\_a/${i}" "${dir\_3}dir\_b"

done

#--------------homework\_4-----------------

homework 1 create 4

dir\_4="/home/acs/homework/lesson\_1/homework\_4/"

rm ${dir\_4}\*.txt

#--------------homework\_5-----------------

homework 1 create 5

dir\_5="/home/acs/homework/lesson\_1/homework\_5/"

#echo ${dir\_5}

rm ${dir\_5}\* -r

#--------------homework\_6-----------------

homework 1 create 6

dir\_6="/home/acs/homework/lesson\_1/homework\_6/"

mv "${dir\_6}task.txt" "${dir\_6}/done.txt"

mkdir ${dir\_6}dir\_a

mv "${dir\_6}done.txt" "${dir\_6}/dir\_a"

#--------------homework\_7-----------------

homework 1 create 7

dir\_7="/home/acs/homework/lesson\_1/homework\_7/"

for ((i=0;i<3;i++))

do

mkdir "${dir\_7}dir\_${i}"

for j in a b c

do

cp "${dir\_7}${j}.txt" "${dir\_7}dir\_${i}/${j}${i}.txt"

done

done

#--------------homework\_8-----------------

homework 1 create 8

dir\_8="/home/acs/homework/lesson\_1/homework\_8/"

rm ${dir\_8}dir\_a/a.txt

mv ${dir\_8}dir\_b/b.txt ${dir\_8}dir\_b/b\_new.txt

cp ${dir\_8}dir\_c/c.txt ${dir\_8}dir\_c/c.txt.bak

#--------------homework\_9-----------------

homework 1 create 9

dir\_9="/home/acs/homework/lesson\_1/homework\_9/"

rm ${dir\_9}\*.txt

cd homeowrk\_0 # 进入作业目录

vim helper.sh # 创建作业脚本

chmod +x helper.sh # 添加可执行权限

小技巧：如何将服务器中的文件整体复制出来？

退出tmux

cat filename：展示filename的文件内容

鼠标选中文本开头的若干字符

用滚轮滑到文件结尾

按住Shift，同时鼠标点击文件结尾，此时会选中文件所有内容

Windows/Linux下，按Ctrl + insert可以复制全文；Mac下，按Command + c可以复制全文。

helpers.sh的内容：

#! /bin/bash

# \*\*\*\*\*\*\*\*\* homework\_0 \*\*\*\*\*\*\*\*\*\*

dir0=/home/acs/homework/lesson\_1/homework\_0

for i in dir\_a dir\_b dir\_c

do

mkdir ${dir0}/$i

done

# \*\*\*\*\*\*\*\*\* homwork\_1 \*\*\*\*\*\*\*\*\*\*

dir1=/home/acs/homework/lesson\_1/homework\_1

for i in a.txt b.txt c.txt

do

cp ${dir1}/${i} ${dir1}/${i}.bak

done

# \*\*\*\*\*\*\*\*\* homwork\_2 \*\*\*\*\*\*\*\*\*\*

dir2=/home/acs/homework/lesson\_1/homework\_2

for i in a b c

do

mv ${dir2}/${i}.txt ${dir2}/${i}\_new.txt

done

# \*\*\*\*\*\*\*\*\* homwork\_3 \*\*\*\*\*\*\*\*\*\*

dir3=/home/acs/homework/lesson\_1/homework\_3

for i in a.txt b.txt c.txt

do

mv ${dir3}/dir\_a/$i ${dir3}/dir\_b/

done

# \*\*\*\*\*\*\*\*\* homwork\_4 \*\*\*\*\*\*\*\*\*\*

dir4=/home/acs/homework/lesson\_1/homework\_4

rm ${dir4}/\*

# \*\*\*\*\*\*\*\*\* homwork\_5 \*\*\*\*\*\*\*\*\*\*

dir5=/home/acs/homework/lesson\_1/homework\_5

rm ${dir5}/\* -r

# \*\*\*\*\*\*\*\*\* homwork\_6 \*\*\*\*\*\*\*\*\*\*

dir6=/home/acs/homework/lesson\_1/homework\_6

mv ${dir6}/task.txt "${dir6}/done.txt"

mkdir ${dir6}/dir\_a

mv "${dir6}/done.txt" ${dir6}/dir\_a

# \*\*\*\*\*\*\*\*\* homwork\_7 \*\*\*\*\*\*\*\*\*\*

dir7=/home/acs/homework/lesson\_1/homework\_7

for ((i=0;i<3;i++))

do

mkdir ${dir7}/dir\_$i

for j in a b c

do

cp ${dir7}/${j}.txt ${dir7}/dir\_${i}/${j}${i}.txt

done

done

# \*\*\*\*\*\*\*\*\* homwork\_8 \*\*\*\*\*\*\*\*\*\*

dir8=/home/acs/homework/lesson\_1/homework\_8

rm ${dir8}/dir\_a/a.txt

mv ${dir8}/dir\_b/b.txt ${dir8}/dir\_b/b\_new.txt

cp ${dir8}/dir\_c/c.txt ${dir8}/dir\_c/c.txt.bak

# \*\*\*\*\*\*\*\*\* homwork\_9 \*\*\*\*\*\*\*\*\*\*

dir9=/home/acs/homework/lesson\_1/homework\_9

rm ${dir9}/\*.txt

#! /bin/bash

if [ $# -ne 1 ]

then

echo "arguments not valid"

exit 1

fi

if ! [ -e "$1" ]

then

echo "not exist"

exit 2

fi

if [ -f "$1" ]

then

echo "regular file"

fi

if [ -d "$1" ]

then

echo "directory"

fi

if [ -r "$1" ]

then

echo "readable"

fi

if [ -w "$1" ]

then

echo "writable"

fi

if [ -x "$1" ]

then

echo "executable"

fi

#! /bin/bash

if [ $# -ne 1 ]

then

echo arguments not valid

exit 1

fi

if [ ! -e "$1" ]

then

echo not exist

exit 2

fi

if [ -f "$1" ]

then

echo regular file

fi

if [ -d "$1" ]

then

echo directory

fi

if [ -r "$1" ]

then

echo readable

fi

if [ -w "$1" ]

then

echo writable

fi

if [ -x "$1" ]

then

echo executable

fi

#! /bin/bash

read n

a[0]=1

a[1]=1

for((i=2;i<=n;i++))

do

x=$(expr ${a[i-1]})

y=$(expr ${a[i-2]})

a[${i}]=$(expr $x + $y)

done

echo ${a[n]}

#! /bin/bash

read n

a[0]=1

a[1]=1

for ((i = 2; i <= n; i ++ ))

do

x=$(expr $i - 1)

y=$(expr $i - 2)

a[$i]=$(expr ${a[$x]} + ${a[$y]})

done

echo ${a[$n]}

#! /bin/bash

read n

read m

for((i=1;i<=n;i++))

do

st[i]=0

done

dfs(){

#echo $1

#read tmp

if [ $1 -eq $n ]

then

m=`expr ${m} - 1`

if [ ${m} -eq 0 ]

then

for((k=0;k<n;k++))

do

echo -e "${path[${k}]} \c"

done

return 0

fi

return 1

fi

local j;

for((j=1;j<=n;j++))

do

if [ ${st[j]} -eq 0 ]

then

path[$1]=$j

st[$j]=1

if dfs `expr ${1} + 1`

then

return 0

fi

st[$j]=0

path[$1]=0

fi

done

return 1

}

dfs 0

#! /bin/bash

read n

read m

for ((i = 1; i <= n; i ++ ))

do

st[$i]=0

done

dfs() {

if [ $1 -eq $n ]

then

m=`expr $m - 1`

if [ $m -eq 0 ]

then

echo ${path[@]}

return 0

fi

return 1

fi

local j=0

for ((j = 1; j <= n; j ++ ))

do

if [ ${st[$j]} -eq 0 ]

then

path[$1]=$j

st[$j]=1

if dfs `expr $1 + 1`

then

return 0

fi

st[$j]=0

fi

done

return 1

}

dfs 0

#! /bin/bash

input\_file=$1

output\_file=$2

read n < ${input\_file}

sum=0

for ((i=1;i<=n;i++))

do

sqr=`expr ${i} \\* ${i}`

sum=`expr $sum + $sqr`

done

echo $sum > $output\_file

#! /bin/bash

input\_file=$1

output\_file=$2

read n < $input\_file

sum=0

for ((i = 1; i <= n; i ++ ))

do

sqr=`expr $i \\* $i`

sum=`expr $sum + $sqr`

done

echo $sum > $output\_file