Shell Programming

理解 shell

GUI 和 命令行的目的都是控制电脑，但是真正能控制电脑的只有操作系统内核（kernel）。

为了保护内核，一般程序不能直接接触 OS 内核，需要一个“代理”，接收用户的命令，做简单处理传递给内核，在 Linux，这个代理就是 shell。

Shell 处理可以解释用户的输入，将它传递给内核，还能：

1. 调用其他程序
2. 在多个程序间传递数据
3. Shell 本身也可以被其他程序调用

Shell 将 OS 内核、程序、用户连接起来，shell 也算一种编程语言，平时所说的 shell 一般指连接用户和 kernel 的程序

Shell 是一种脚本语言

两类语言：

- C/C++、Pascal、Go、汇编等，必须在程序运行之前将代码翻译成二进制——编译，他们是编译型语言。
  编译型语言执行速度快、对硬件要求低、保密性好，适合开发操作系统、大型应用程序、数据库等。
- Shell、JS、Python、PHP 等，需要一边执行一边翻译，不会生成可执行文件，用户需要拿到源码才能运行程序，程序运行时即是翻译——解释，它们是解释型语言，或脚本语言 script。
  脚本语言使用灵活、部署容易、跨平台性，适合 web 开发及小型工具制作。

Linux 运维工程师：OPS

知乎、豆瓣、YouTube、Instagram 都是用 Python 开发

常见的 shell

- Sh:Bourn shell,A&AT 开发
- Csh
- Tcsh
- Ash
- Bash

Shell 是一个程序，一般放在/bin 或者/user/bin 目录下，当前 Linux 系统可用的 Shell 都记录在/etc/shells 文件中

使用 shell

- 进入控制台

现在的 Linux 系统会创建几个虚拟控制台，一个供图形界面使用，其他保留原格式。

按下 Ctrl + Alt + Fn(n=1,2,3,4,5...)

CentOS 在启动时会创建 6 个虚拟控制台，按下快捷键 Ctrl + Alt + Fn(n=2,3,4,5,6)可以从图形界面模式切换到控制台模式，按下 Ctrl + Alt + F1 可以从控制台模式再切换回图形界面模式（1 号是 GUI）

- 使用 terminal 终端

Shell 提示符

Echo：输出命令

```
Echo something
```

Shell 通过 PS1 和 PS2 两个环境变量来控制提示符格式：

- PS1 控制最外层命令行的提示符格式。
- PS2 控制第二层命令行的提示符格式。

```
[edwinxu@localhost ~]$ echo $PS1
[\u@\h \W]\$
[edwinxu@localhost ~]$ echo $PS2
>
[edwinxu@localhost ~]$
```

可以修改：

```
[edwinxu@localhost ~]$ PS1="[\t][\u]\$ "
[09:28:36][edwinxu]$
```

以\为前导的特殊字符来表示命令提示符中包含的要素：

\a	铃声字符
\d	格式为“日 月 年”的日期
\e	ASCII转义字符
\h	本地主机名
\H	完全合格的限定域主机名
\j	shell当前管理的作业数
\1	shell终端设备名的基本名称
\n	ASCII换行字符
\r	ASCII回车
\s	shell的名称
\t	格式为“小时:分钟:秒”的24小时制的当前时间
\T	格式为“小时:分钟:秒”的12小时制的当前时间
\@	格式为am/pm的12小时制的当前时间
\u	当前用户的用户名

\v	bash shell的版本
\V	bash shell的发布级别
\w	当前工作目录
\W	当前工作目录的基本名称
\!	该命令的bash shell历史数
\#	该命令的命令数量
\$	如果是普通用户，则为美元符号 `$` ；如果超级用户（root 用户），则为井号 `#` 。
\nnn	对应于八进制值 nnn 的字符
\\	斜杠
\[	控制码序列的开头
\]	控制码序列的结尾

第一个 shell 脚本

新建文件 mysh.sh，扩展名只是让你明白它是 shell 程序，其他没什么意义。

输入：

```
#!/bin/bash
echo "Hello World"
```

#!: 告诉系统这个脚本需要什么解释器来执行。(现在一般使用 bash 解释器)

运行方式一:

命令行输入:

```
chmod +x ./mysh.sh  #使脚本具有执行权限
./mysh.sh  #执行脚本
```

./mysh.sh 不能直接写成 mysh.sh

运行方式二:

直接运行解释器, 参数就是脚本:

```
/bin/sh test.sh
/bin/php test.php
```

注释: #

Read 命令

使用 read 命令可以从是指定中获取输入并赋值

Read value (将输入赋值给 value)

```
echo "What is your name?"
read name  #读取
echo "Hello, $name" #使用$value即可转化为字符串
```

变量

Bash shell 中, **每一个变量的值都是字符串**, 不需要指明类型

三种定义方式:

```
a = str #str不包含空格, 直接赋值
b = ' s t r' #含空格需要引号
c = " s t r" #单双引号有小区别
```

1. **单引号: 不解析任何变量、命令, 原生输出**
2. **双引号: 先解析里面的变量和命令**

使用变量

```
author="严长生"
echo $author
echo ${author}
```

$value 即使用

{}是可选的，用于识别边界

建议都加{}，避免错误

将命令的执行结果赋值给变量：

```
variable=`command`
variable=$(command)
```

推荐后者

=前后不能由空格？？？

```
#!/bin/bash
myls = $(cat helloworld.sh)
echo $myls
```

空格，执行失败！

一般先定义，后赋值，不能一起

只读变量

readonly 命令可以将变量定义为只读变量

```
a="str" #先定义
readonly a #修饰为只读
echo a

#readonly只是将一个已经存在的变量改为只读，它不能定义，不像关键字

#所以这是错误的
#readonly b
#b = "!21"
#echo b
```

删除变量——unset

Unset value

变量删除后不能在使用，不能删除只读变量

变量类型

1. 局部变量：仅在 shell 实例中有效，其他程序不能访问
2. 环境变量：所有程序都不能访问环境变量，必要时可以定义环境变量。

Edwin Xu

3. Shell 变量：由 shell 程序设置的特殊变量

特殊变量

- **$:表示 shell 进程的 id——pid**

```
$echo $$
```

变量	含义
$0	当前脚本的文件名
$n	传递给脚本或函数的参数。n 是一个数字，表示第几个参数。例如，第一个参数是$1，第二个参数是$2。
$#	传递给脚本或函数的参数个数。
$*	传递给脚本或函数的所有参数。
$@	传递给脚本或函数的所有参数。被双引号(" ")包含时，与 $* 稍有不同，下面将会讲到。
$?	上个命令的退出状态，或函数的返回值。
$$	当前Shell进程ID。对于 Shell 脚本，就是这些脚本所在的进程ID。

$* 和 $@ 的区别

$* 和 $@ 都表示传递给函数或脚本的所有参数，不被双引号(" ")包含时，都以"$1" "$2" … "$n" 的形式输出所有参数。

但是当它们被双引号(" ")包含时，"$*" 会将所有的参数作为一个整体，以"$1 $2 … $n"的形式输出所有参数；"$@" 会将各个参数分开，以"$1" "$2" … "$n" 的形式输出所有参数。

退出状态

$? 可以获取上一个命令的退出状态。**所谓退出状态，就是上一个命令执行后的返回结果。**

退出状态是一个数字，一般情况下，大部分命令执行成功会返回 0，失败返回 1。不过，也有一些命令返回其他值，表示不同类型的错误。

Shell 替换：Shell 变量替换，命令替换，转义字符

如果表达式中包含特殊字符，Shell 将会进行替换。例如，在双引号中使用变量就是一种替换，转义字符也是一种替换。

```
a=10
echo -e "Value of a is $a \n"
```

这里 -e 表示对转义字符进行替换。如果不使用 -e 选项，将会原样输出：

```
Value of a is 10\n
```

转义字符	含义
\\	反斜杠
\a	警报，响铃
\b	退格（删除键）
\f	换页(FF)，将当前位置移到下页开头
\n	换行
\r	回车
\t	水平制表符（tab键）
\v	垂直制表符

可以使用 echo 命令的 -E 选项禁止转义，默认也是不转义的；使用 -n 选项可以禁止插入换行符。

命令替换

命令替换是指 Shell 可以先执行命令，将输出结果暂时保存，在适当的地方输出。

```
`command`
注意是反引号，不是单引号
```

```
DATE=`date`
echo "Date is $DATE"
USERS=`who | wc -l`
echo "Logged in user are $USERS"
UP=`date ; uptime`
echo "Uptime is $UP"
```

变量替换

变量替换可以根据变量的状态（是否为空、是否定义等）来改变它的值

形式	说明
${var}	变量本来的值
${var:-word}	如果变量 var 为空或已被删除(unset)，那么返回 word，但不改变 var 的值。
${var:=word}	如果变量 var 为空或已被删除(unset)，那么返回 word，并将 var 的值设置为 word。
${var:?message}	如果变量 var 为空或已被删除(unset)，那么将消息 message 送到标准错误输出，可以用来检测变量 var 是否可以被正常赋值。 若此替换出现在Shell脚本中，那么脚本将停止运行。
${var:+word}	如果变量 var 被定义，那么返回 word，但不改变 var 的值。

```
echo ${var:-"Variable is not set"}
echo "1 - Value of var is ${var}"

echo ${var:="Variable is not set"}
echo "2 - Value of var is ${var}"

unset var
echo ${var:+"This is default value"}
echo "3 - Value of var is $var"

var="Prefix"
echo ${var:+"This is default value"}
echo "4 - Value of var is $var"

echo ${var:?"Print this message"}
echo "5 - Value of var is ${var}"
```

```
Variable is not set
1 - Value of var is
Variable is not set
2 - Value of var is Variable is not set

3 - Value of var is
This is default value
4 - Value of var is Prefix
Prefix
5 - Value of var is Prefix
```

Shell 运算符

Bash 支持很多运算符，包括算数运算符、关系运算符、布尔运算符、字符串运算符和文件测试运算符。

原生 bash 不支持简单的数学运算，但是可以通过其他命令来实现，例如 awk 和 expr，expr 最常用。

expr 是一款表达式计算工具，使用它能完成表达式的求值操作。

```
val=`expr 2 + 2`
echo "Total value : $val"
```

```
a=10
```

```
b=20
val=`expr $a \* $b`
echo "a * b : $val"
```

乘号(*)前边必须加反斜杠(\)才能实现乘法运算;

运算符	说明	举例
+	加法	`expr $a + $b` 结果为 30。
-	减法	`expr $a - $b` 结果为 10。
*	乘法	`expr $a \* $b` 结果为 200。
/	除法	`expr $b / $a` 结果为 2。
%	取余	`expr $b % $a` 结果为 0。
=	赋值	a=$b 将把变量 b 的值赋给 a。
==	相等。用于比较两个数字，相同则返回 true。	[$a == $b] 返回 false。
!=	不相等。用于比较两个数字，不相同则返回 true。	[$a != $b] 返回 true。

关系运算符

关系运算符只支持数字，不支持字符串，除非字符串的值是数字。

运算符	说明	举例
-eq	检测两个数是否相等，相等返回 true。	[$a -eq $b] 返回 true。
-ne	检测两个数是否相等，不相等返回 true。	[$a -ne $b] 返回 true。
-gt	检测左边的数是否大于右边的，如果是，则返回 true。	[$a -gt $b] 返回 false。
-lt	检测左边的数是否小于右边的，如果是，则返回 true。	[$a -lt $b] 返回 true。
-ge	检测左边的数是否大等于右边的，如果是，则返回 true。	[$a -ge $b] 返回 false。
-le	检测左边的数是否小于等于右边的，如果是，则返回 true。	[$a -le $b] 返回 true。

```
a=10
b=20
if [ $a -eq $b ]
then
   echo "$a -eq $b : a is equal to b"
else
   echo "$a -eq $b: a is not equal to b"
fi
```

```
if [exp]
then
  op
else
  op
fi
```

布尔运算符

运算符	说明	举例
!	非运算，表达式为 true 则返回 false，否则返回 true。	[! false] 返回 true。
-o	或运算，有一个表达式为 true 则返回 true。	[$a -lt 20 -o $b -gt 100] 返回 true。
-a	与运算，两个表达式都为 true 才返回 true。	[$a -lt 20 -a $b -gt 100] 返回 false。

字符串运算符

运算符	说明	举例
=	检测两个字符串是否相等，相等返回 true。	[$a = $b] 返回 false。
!=	检测两个字符串是否相等，不相等返回 true。	[$a != $b] 返回 true。
-z	检测字符串长度是否为0，为0返回 true。	[-z $a] 返回 false。
-n	检测字符串长度是否为0，不为0返回 true。	[-z $a] 返回 true。
str	检测字符串是否为空，不为空返回 true。	[$a] 返回 true。

文件测试运算符

文件测试运算符用于检测 Unix 文件的各种属性。

文件测试运算符列表

操作符	说明	举例
-b file	检测文件是否是块设备文件，如果是，则返回 true。	[-b $file] 返回 false。
-c file	检测文件是否是字符设备文件，如果是，则返回 true。	[-b $file] 返回 false。
-d file	检测文件是否是目录，如果是，则返回 true。	[-d $file] 返回 false。
-f file	检测文件是否是普通文件（既不是目录，也不是设备文件），如果是，则返回 true。	[-f $file] 返回 true。
-g file	检测文件是否设置了 SGID 位，如果是，则返回 true。	[-g $file] 返回 false。
-k file	检测文件是否设置了粘着位(Sticky Bit)，如果是，则返回 true。	[-k $file] 返回 false。
-p file	检测文件是否是具名管道，如果是，则返回 true。	[-p $file] 返回 false。
-u file	检测文件是否设置了 SUID 位，如果是，则返回 true。	[-u $file] 返回 false。
-r file	检测文件是否可读，如果是，则返回 true。	[-r $file] 返回 true。
-w file	检测文件是否可写，如果是，则返回 true。	[-w $file] 返回 true。
-x file	检测文件是否可执行，如果是，则返回 true。	[-x $file] 返回 true。
-s file	检测文件是否为空（文件大小是否大于0），不为空返回 true。	[-s $file] 返回 true。
-e file	检测文件（包括目录）是否存在，如果是，则返回 true。	[-e $file] 返回 true。

Shell 注释

以“#”开头的行就是注释，会被解释器忽略。

sh 里没有多行注释，只能每一行加一个#号

Shell 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

单引号字符串的限制：

1. 单引号里的任何字符都会**原样输出**，单引号字符串中的变量是无效的；
2. 单引号字串中不能出现单引号（对单引号使用转义符后也不行）。

双引号的优点：

1. 双引号里可以有变量
2. 双引号里可以出现转义字符

拼接字符串

```
your_name="qinjx"
greeting="hello, "$your_name" !"
greeting_1="hello, ${your_name} !"
echo $greeting $greeting_1
```

获取字符串长度

```
string="abcd"
echo ${#string} #输出 4
```

提取子字符串

```
string="alibaba is a great company"
echo ${string:1:4} #输出 liba
```

查找子字符串

```
string="alibaba is a great company"
echo `expr index "$string" is`
```

Shell 数组

Shell 在编程方面比 Windows 批处理强大很多，无论是在循环、运算。

bash **支持一维数组（不支持多维数组）**，并且没有限定数组的大小。

定义数组

在 Shell 中，用括号来表示数组，数组元素用“空格”符号分割开。定义数组的一般形式为：

```
array_name=(value1 ... valuen)
```

注意：不是逗号分隔

```
array_name=(value0 value1 value2 value3)

或者:
array_name=(
value0
value1
value2
value3
)
```

还可以单独定义数组的各个分量：

```
array_name[0]=value0
array_name[1]=value1
array_name[2]=value2
```

读取数组

读取数组元素值的一般格式是：

```
${array_name[index]}
```

```
arr=(1 2 3 4 5)
echo ${arr[1]}
```

使用@ 或 * 可以获取数组中的所有元素，例如：

```
${array_name[*]}
${array_name[@]}
```

获取数组的长度

获取数组长度的方法与获取字符串长度的方法相同

```
# 取得数组元素的个数
length=${#array_name[@]}
# 或者
length=${#array_name[*]}
# 取得数组单个元素的长度
lengthn=${#array_name[n]}
```

printf 命令：格式化输出语句

printf 命令用于格式化输出， 是echo 命令的增强版。它是C 语言 printf()库函数的一个有限的变形，并且在语法上有些不同。

注意：printf 由 POSIX 标准所定义，移植性要比 echo 好。

```
printf "Hello, Shell\n"
```

printf 不像 echo 那样会自动换行，必须显式添加换行符(\n)。

printf 命令的语法：

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

与C 语言 printf()函数的不同：

1. printf 命令不用加括号
2. format-string 可以没有引号，但最好加上，单引号双引号均可。
3. 参数多于格式控制符(%)时，format-string 可以重用，可以将所有参数都转换。
4. arguments 使用空格分隔，不用逗号。

```
# format-string为双引号
$ printf "%d %s\n" 1 "abc"
1 abc
# 单引号与双引号效果一样
$ printf '%d %s\n' 1 "abc"
1 abc
# 没有引号也可以输出
$ printf %s abcdef
abcdef
# 格式只指定了一个参数，但多出的参数仍然会按照该格式输出，format-string 被重用
$ printf %s abc def
abcdef
$ printf "%s\n" abc def
abc
def
$ printf "%s %s %s\n" a b c d e f g h i j
a b c
d e f
g h i
j
# 如果没有 arguments，那么 %s 用NULL代替，%d 用 0 代替
$ printf "%s and %d \n"
```

```
and 0
# 如果以 %d 的格式来显示字符串，那么会有警告，提示无效的数字，此时默认置为 0
$ printf "The first program always prints'%s,%d\n'" Hello Shell
-bash: printf: Shell: invalid number
The first program always prints 'Hello,0'
$
```

if else 语句

Shell 有三种 if ... else 语句：

1. if ... fi 语句；

2. if ... else ... fi 语句；

3. if ... elif ... else ... fi 语句。

if ... else 语句

if ... else 语句的语法：

```
if [ expression ]
then
   Statement(s) to be executed if expression is true
fi
```

最后**必须以 fi 来结尾闭合 if，fi 就是 if 倒过来拼写**

注意：expression 和方括号([ ])之间必须有空格，否则会有语法错误。

if ... else ... fi 语句

if ... elif ... fi 语句

test 命令

test 命令用于检查某个条件是否成立，它可以进行数值、字符和文件三个方面的测试。

数值测试：

-eq 等于则为真

-ne 不等于则为真

-gt 大于则为真

-ge 大于等于则为真

-lt 小于则为真

-le 小于等于则为真

```
num1=100
num2=100
if test $[num1] -eq $[num2]
then
    echo 'The two numbers are equal!'
else
    echo 'The two numbers are not equal!'
fi
```

字符串测试:

= 等于则为真

!= 不相等则为真

-z 字符串 字符串长度伪则为真

-n 字符串 字符串长度不伪则为真

```
num1=100
num2=100
if test num1=num2
then
    echo 'The two strings are equal!'
else
    echo 'The two strings are not equal!'
fi
```

文件测试:

-e 文件名 如果文件存在则为真

-r 文件名 如果文件存在且可读则为真

-w 文件名 如果文件存在且可写则为真

-x 文件名 如果文件存在且可执行则为真

-s 文件名 如果文件存在且至少有一个字符则为真

-d 文件名 如果文件存在且为目录则为真

-f 文件名 如果文件存在且为普通文件则为真

-c 文件名 如果文件存在且为字符型特殊文件则为真

-b 文件名 如果文件存在且为块特殊文件则为真

```
if test -e ./bash
```

Shell 还提供了与(!)、或(-o)、非(-a)三个逻辑操作符用于将测试条件连接起来，其优先级为:“!”最高,“-a”次之,“-o”最低

case esac 语句

case ... esac 与其他语言中的 switch ... case 语句类似，是一种多分枝选择结构。

case 语句匹配一个值或一个模式，如果匹配成功，执行相匹配的命令。case 语句格式如下：

```
case 值 in
模式 1)
    command1
    command2
    command3
    ;;
模式 2）
    command1
    command2
    command3
    ;;
*)
    command1
    command2
    command3
    ;;
esac
```

取值后面必须为关键字 in，每一模式必须以右括号结束。取值可以为变量或常数。匹配发现取值符合某一模式后，其间所有命令开始执行直至 ;;。;; 与其他语言中的 break 类似，意思是跳到整个 case 语句的最后。

```
echo 'Input a number between 1 to 4'
echo 'Your number is:\c'
read aNum
case $aNum in
    1)  echo 'You select 1'
    ;;
    2)  echo 'You select 2'
    ;;
    3)  echo 'You select 3'
    ;;
    4)  echo 'You select 4'
```

```
    ;;
    *)  echo 'You do not select a number between 1 to 4'
    ;;
esac
```

for 循环

for 循环一般格式为：

```
for 变量 in 列表
do
    command1
    command2
    ...
    commandN
done
```

```
for loop in 1 2 3 4 5
do
    echo "The value is: $loop"
done
```

```
for str in 'This is a string'
do
    echo $str
done
```

```
for FILE in $HOME/.bash*
do
   echo $FILE
done
```

while 循环

```
while command
do
   Statement(s) to be executed if command is true
done
```

```
COUNTER=0
```

```
while [ $COUNTER -lt 5 ]
do
    COUNTER='expr $COUNTER+1'
    echo $COUNTER
done
```

until 循环

until 循环执行一系列命令直至条件为 true 时停止。until 循环与 while 循环在处理方式上刚好相反。一般 while 循环优于 until 循环，但在某些时候，也只是极少数情况下，until 循环更加有用。

```
until command
do
   Statement(s) to be executed until command is true
done
```

break 和 continue 命令

break 命令允许跳出所有循环

在嵌套循环中，**break 命令后面还可以跟一个整数，表示跳出第几层循环**。例如：

```
break n
```

continue 命令与 break 命令类似，只有一点差别，**它不会跳出所有循环，仅仅跳出当前循环**。

同样，continue 后面也可以跟一个数字，表示跳出第几层循环。

Shell 函数

Shell 函数的定义格式如下：

```
function_name () {
    list of commands
    [ return value ]
}
```

如果你愿意，也可以在函数名前加上关键字 function：

```
function function_name () {
    list of commands
    [ return value ]
}
```

函数返回值，可以显式增加 return 语句；**如果不加，会将最后一条命令运行结果作为返回值。**

Shell 函数返回值只能是整数，一般用来表示函数执行成功与否，0 表示成功，其他值表示失败。如果 return 其他数据，比如一个字符串，往往会得到错误提示：“numeric argument required”。

```
#!/bin/bash
# Define your function here
Hello () {
   echo "Url is http://see.xidian.edu.cn/cpp/shell/"
}
# Invoke your function
Hello
```

在 Shell 中，调用函数时可以向其传递参数。在函数体内部，通过 $n 的形式来获取参数的值，例如，$1 表示第一个参数，$2 表示第二个参数...
(不需要声明形参)

```
funWithParam(){
    echo "The value of the first parameter is $1 !"
    echo "The value of the second parameter is $2 !"
    echo "The value of the tenth parameter is $10 !"
    echo "The value of the tenth parameter is ${10} !"
    echo "The value of the eleventh parameter is ${11} !"
    echo "The amount of the parameters is $# !"  # 参数个数
    echo "The string of the parameters is $* !"  # 传递给函数的所有参数
}
funWithParam 1 2 3 4 5 6 7 8 9 34 73
```

另外，还有几个特殊变量用来处理参数：

特殊变量	说明
$#	传递给函数的参数个数。
$*	显示所有传递给函数的参数。
$@	与$*相同，但是略有区别，请查看Shell特殊变量。
$?	函数的返回值。

输入输出重定向

命令	说明
command > file	将输出重定向到 file。
command < file	将输入重定向到 file。
command >> file	将输出以追加的方式重定向到 file。
n > file	将文件描述符为 n 的文件重定向到 file。
n >> file	将文件描述符为 n 的文件以追加的方式重定向到 file。
n >& m	将输出文件 m 和 n 合并。
n <& m	将输入文件 m 和 n 合并。
<< tag	将开始标记 tag 和结束标记 tag 之间的内容作为输入。

Shell 文件 import

像其他语言一样，Shell 也可以包含外部脚本，将外部脚本的内容合并到当前脚本。

Shell 中包含脚本可以使用：

```
. filename
或者
source filename
```

```
#!/bin/bash
. ./subscript.sh
echo $url
```

Edwin Xu