SHELL编程模块课程目标

- ① Shell的基本语法结构

  如：变量定义、条件判断、循环语句(for、until、while)、分支语句、函数和数组等；

  ② 基本正则表达式的运用;

  ③ 文件处理三剑客：grep、sed、awk工具的使用；

  ④ 使用shell脚本完成一些较复杂的任务，如：服务搭建、批量处理等。

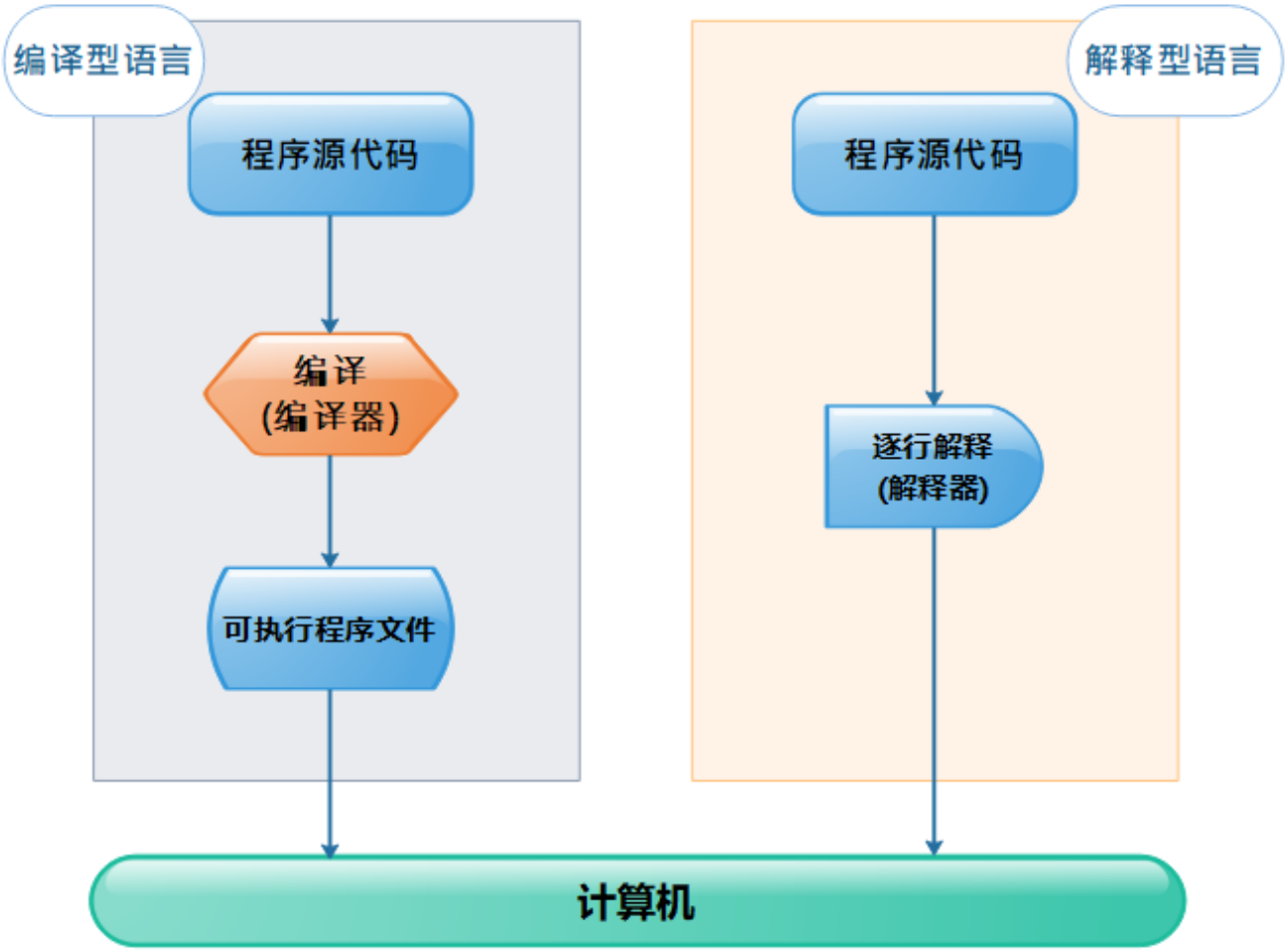
  说明：以上内容仅仅是基本要求，还有很多更深更难的语法需要扩充学习。

- **本节目标**
  - 熟练掌握shell变量的定义和获取（重点）
  - 能够进行shell简单的四则运算

一、SHELL介绍

前言：

计算机只能认识（识别）机器语言(0和1)，如（11000000 这种）。但是，我们的程序猿们不能直接去写01这样的代码，所以，要想将程序猿所开发的代码在计算机上运行，就必须找"人"（工具）来翻译成机器语言，这个"人"(工具)就是我们常常所说的**编译器**或者**解释器**。



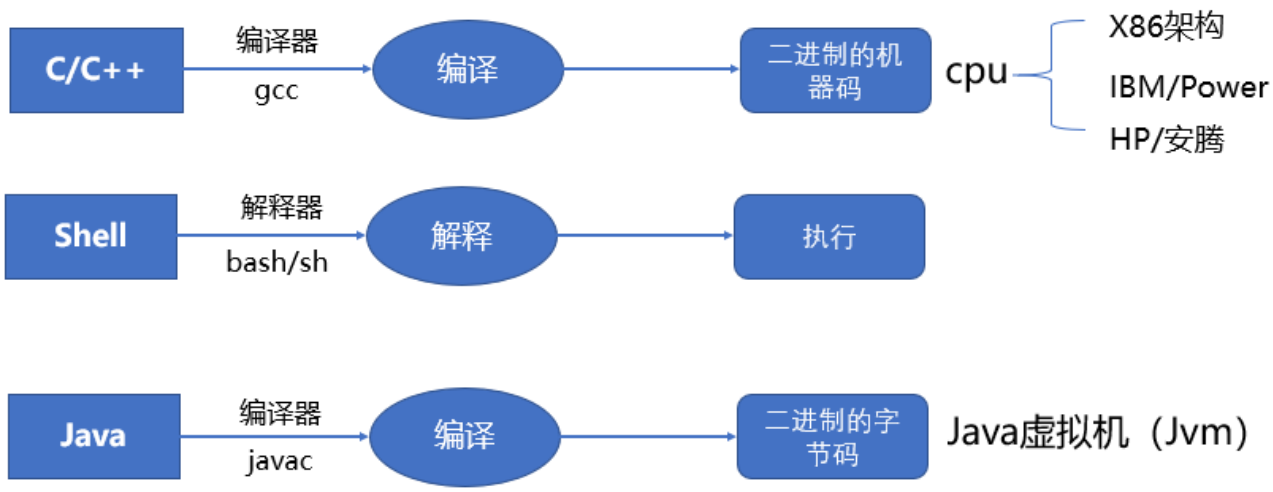
1. 编程语言分类

- **编译型语言：**

程序在执行之前需要一个专门的编译过程，把程序编译成为机器语言文件，运行时不需要重新翻译，直接使用编译的结果就行了。程序执行效率高，依赖编译器，跨平台性差些。如C、C++

- **解释型语言：**

程序不需要编译，程序在运行时由**解释器**翻译成机器语言，每执行一次都要翻译一次。因此效率比较低。比如Python/JavaScript/ Perl /ruby/Shell等都是解释型语言。

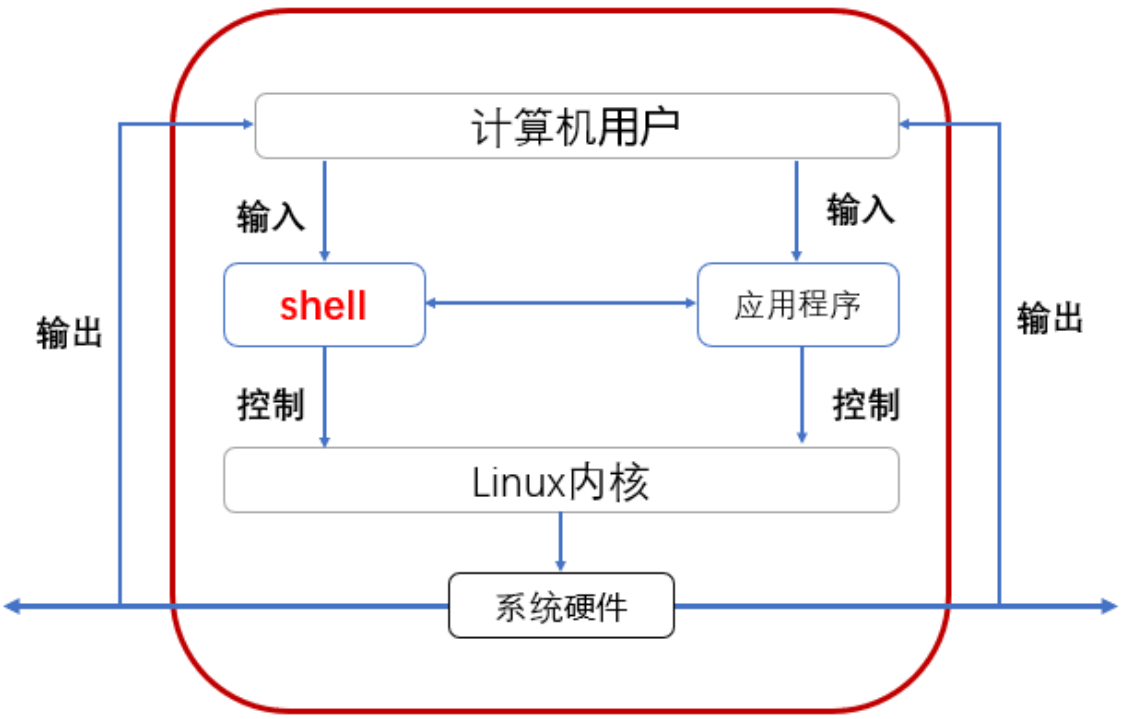


- **总结**

编译型语言比解释型语言速度较快，但是不如解释型语言跨平台性好。如果做底层开发或者大型应用程序或者操作系开发一般都用编译型语言；如果是一些服务器脚本及一些辅助的接口，对速度要求不高、对各个平台的兼容性有要求的话则一般都用解释型语言。

2. shell简介

shell介于内核与用户之间，负责命令的解释



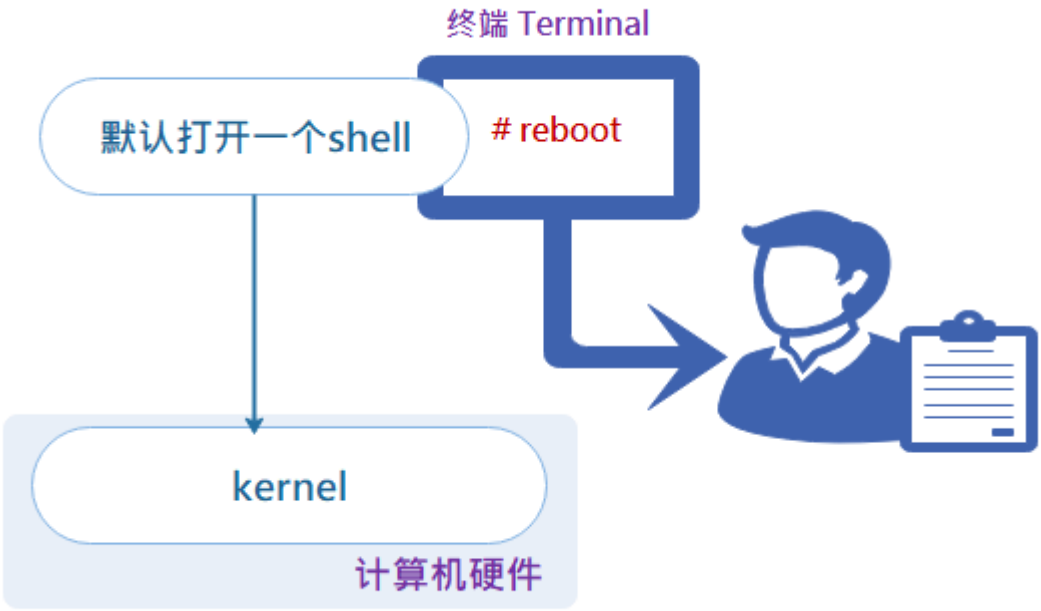
总结:

- shell就是人机交互的一个桥梁
- shell的种类

```
[root@MissHou ~]# cat /etc/shells
/bin/sh             #是bash的一个快捷方式
/bin/bash           #bash是大多数Linux默认的shell，包含的功能几乎可以涵盖shell所有的功能
/sbin/nologin       #表示非交互，不能登录操作系统
/bin/dash           #小巧，高效，功能相比少一些

/bin/csh            #具有C语言风格的一种shell，具有许多特性，但也有一些缺陷
/bin/tcsh           #是csh的增强版，完全兼容csh
```

思考：终端和shell有什么关系?



3. shell脚本

㈠ 什么是shell脚本?

- 一句话概括

简单来说就是将需要执行的命令保存到文本中，按照顺序执行。它是解释型的，意味着不需要编译。

- 准确叙述

若干命令 + 脚本的基本格式 + 脚本特定语法 + 思想= shell脚本

㈡ 什么时候用到脚本?

重复化、复杂化的工作，通过把工作的命令写成脚本，以后仅仅需要执行脚本就能完成这些工作。

㈢ shell脚本能干啥?

①自动化软件部署 LAMP/LNMP/Tomcat...

②自动化管理 系统初始化脚本、批量更改主机密码、推送公钥...

③自动化分析处理 统计网站访问量

④自动化备份 数据库备份、日志转储...

⑤自动化监控脚本

㈣ 如何学习shell脚本?

1. 尽可能记忆更多的命令(记忆命令使用功能和场景)
2. 掌握脚本的标准的格式（指定魔法字节、使用标准的执行方式运行脚本）
3. 必须**熟悉掌握**脚本的基本语法（重点）

㈤ 学习shell脚本的秘诀

多看（看懂）——>模仿（多练）——>多思考（多写）

㈥ shell脚本的基本写法

1）**脚本第一行**，魔法字符**#!**指定解释器【必写】

`#!/bin/bash` 表示以下内容使用bash解释器解析

注意： 如果直接将解释器路径写死在脚本里，可能在某些系统就会存在找不到解释器的兼容性问题，所以可以使用:`#!/bin/env 解释器`

2）**脚本第二部分**，注释(#号)说明，对脚本的基本信息进行描述【可选】

```
#!/bin/env bash

# 以下内容是对脚本的基本信息的描述
# Name: 名字
# Desc:描述describe
# Path:存放路径
# Usage:用法
# Update:更新时间

#下面就是脚本的具体内容
commands
...
```

3）**脚本第三部分**，脚本要实现的具体代码内容

㈦ shell脚本的执行方法

- 标准脚本执行方法（建议）

```
1）编写人生第一个shell脚本
[root@MissHou shell01]# cat first_shell.sh
#!/bin/env bash

# 以下内容是对脚本的基本信息的描述
# Name: first_shell.sh
# Desc: num1
```

```
# Path: /shell01/first_shell.sh
# Usage:/shell01/first_shell.sh
# Update:2019-05-05

echo "hello world"
echo "hello world"
echo "hello world"

2）脚本增加可执行权限
[root@MissHou shell01]# chmod +x first_shell.sh

3）标准方式执行脚本
[root@MissHou shell01]# pwd
/shell01
[root@MissHou shell01]# /shell01/first_shell.sh
或者
[root@MissHou shell01]# ./first_shell.sh

注意：标准执行方式脚本必须要有可执行权限。
```

- 非标准的执行方法（不建议）

1. 直接在命令行指定解释器执行

```
[root@MissHou shell01]# bash first_shell.sh
[root@MissHou shell01]# sh first_shell.sh
[root@MissHou shell01]# bash -x first_shell.sh
+ echo 'hello world'
hello world
+ echo 'hello world'
hello world
+ echo 'hello world'
hello world

-x:一般用于排错，查看脚本的执行过程
-n:用来查看脚本的语法是否有问题
```

2. 使用 `source` 命令读取脚本文件,执行文件里的代码

```
[root@MissHou shell01]# source first_shell.sh
hello world
hello world
hello world
```

小试牛刀：写一个木有灵魂的脚本，要求如下：

1. 删除/tmp/目录下的所有文件
2. 然后在/tmp目录里创建3个目录，分别是dir1~dir3
3. 拷贝/etc/hosts文件到刚创建的dir1目录里
4. 最后打印"报告首长，任务已于2019-05-05 10:10:10时间完成"内容

```
echo "报告首长，任务已于$(date +'%F %T')"
```

二、变量的定义

1. 变量是什么?

一句话概括：变量是用来临时保存数据的，该数据是可以变化的数据。

2. 什么时候需要定义变量?

- 如果某个内容需要多次使用，并且在代码中**重复出现**，那么可以用变量代表该内容。这样在修改内容的时候，仅仅需要修改变量的值。
- 在代码运作的过程中，可能会把某些命令的执行结果保存起来，后续代码需要使用这些结果，就可以直接使用这个变量。

3.变量如何定义?

变量名=变量值

变量名：用来临时保存数据的

变量值：就是临时的可变化的数据

```
[root@MissHou ~]# A=hello          定义变量A
[root@MissHou ~]# echo $A          调用变量A，要给钱的，不是人民币是美元"$"
hello
[root@MissHou ~]# echo ${A}        还可以这样调用，不管你的姿势多优雅，总之要给钱
hello
[root@MissHou ~]# A=world          因为是变量所以可以变，移情别恋是常事
[root@MissHou ~]# echo $A          不管你是谁，只要调用就要给钱
world
[root@MissHou ~]# unset A          不跟你玩了，取消变量
[root@MissHou ~]# echo $A          从此，我单身了，你可以给我介绍任何人
```

4. 变量的定义规则

虽然可以给变量（变量名）赋予任何值；但是，对于变量名也是要求的！😒

(一) 变量名区分大小写

```
[root@MissHou ~]# A=hello
[root@MissHou ~]# a=world
[root@MissHou ~]# echo $A
hello
[root@MissHou ~]# echo $a
world
```

㈡ 变量名不能有特殊符号

```
[root@MissHou ~]# *A=hello
-bash: *A=hello: command not found
[root@MissHou ~]# ?A=hello
-bash: ?A=hello: command not found
[root@MissHou ~]# @A=hello
-bash: @A=hello: command not found

特别说明：对于有空格的字符串给变量赋值时，要用引号引起来
[root@MissHou ~]# A=hello world
-bash: world: command not found
[root@MissHou ~]# A="hello world"
[root@MissHou ~]# A='hello world'
```

㈢ 变量名不能以数字开头

```
[root@MissHou ~]# 1A=hello
-bash: 1A=hello: command not found
[root@MissHou ~]# A1=hello
注意：不能以数字开头并不代表变量名中不能包含数字呦。
```

㈣ 等号两边不能有任何空格

```
[root@MissHou ~]# A =123
-bash: A: command not found
[root@MissHou ~]# A= 123
-bash: 123: command not found
[root@MissHou ~]# A = 123
-bash: A: command not found
[root@MissHou ~]# A=123
[root@MissHou ~]# echo $A
123
```

㈤ 变量名尽量做到见名知意

```
NTP_IP=10.1.1.1
DIR=/u01/app1
TMP_FILE=/var/log/1.log
...

说明：一般变量名使用大写（小写也可以），不要同一个脚本中变量全是a,b,c等不容易阅读
```

5. 变量的定义方式有哪些?

㈠ 基本方式

> 直接赋值给一个变量

```
[root@MissHou ~]# A=1234567
[root@MissHou ~]# echo $A
1234567
[root@MissHou ~]# echo ${A:2:4}      表示从A变量中第3个字符开始截取，截取4个字符
3456

说明:
$变量名 和 ${变量名}的异同
相同点：都可以调用变量
不同点：${变量名}可以只截取变量的一部分，而$变量名不可以
```

㈡ 命令执行结果赋值给变量

```
[root@MissHou ~]# B=`date +%F`
[root@MissHou ~]# echo $B
2019-04-16
[root@MissHou ~]# C=$(uname -r)
[root@MissHou ~]# echo $C
2.6.32-696.el6.x86_64
```

㈢ 交互式定义变量(read)

目的：让用户自己给变量赋值，比较灵活。

语法：`read [选项] 变量名`

常见选项：

选项	释义
-p	定义提示用户的信息
-n	定义字符数（限制变量值的长度）
-s	不显示（不显示用户输入的内容）
-t	定义超时时间，默认单位为秒（限制用户输入变量值的超时时间）

举例说明：

```
用法1：用户自己定义变量值
[root@MissHou ~]# read name
harry
[root@MissHou ~]# echo $name
harry
[root@MissHou ~]# read -p "Input your name:" name
Input your name:tom
[root@MissHou ~]# echo $name
tom

用法2：变量值来自文件
[root@MissHou ~]# cat 1.txt
```

```
10.1.1.1 255.255.255.0

[root@MissHou ~]# read ip mask < 1.txt
[root@MissHou ~]# echo $ip
10.1.1.1
[root@MissHou ~]# echo $mask
255.255.255.0
```

㈣ 定义有类型的变量(declare)

目的：给变量做一些限制，固定变量的类型，比如：整型、只读

用法：`declare 选项 变量名=变量值`

常用选项：

选项	释义	举例
-i	将变量看成整数	declare -i A=123
-r	定义只读变量	declare -r B=hello
-a	定义普通数组；查看普通数组	
-A	定义关联数组；查看关联数组	
-x	将变量通过环境导出	declare -x AAA=123456 等于 export AAA=123456

举例说明：

```
[root@MissHou ~]# declare -i A=123
[root@MissHou ~]# echo $A
123
[root@MissHou ~]# A=hello
[root@MissHou ~]# echo $A
0

[root@MissHou ~]# declare -r B=hello
[root@MissHou ~]# echo $B
hello
[root@MissHou ~]# B=world
-bash: B: readonly variable
[root@MissHou ~]# unset B
-bash: unset: B: cannot unset: readonly variable
```

6. 变量的分类

㈠ 本地变量

- **本地变量**：当前用户自定义的变量。当前进程中有效，其他进程及当前进程的子进程无效。

㈡ 环境变量

- **环境变量**：当前进程有效，并且能够被**子进程**调用。
  - `env` 查看当前用户的环境变量
  - `set` 查询当前用户的所有变量(临时变量与环境变量)
  - `export 变量名=变量值` 或者 `变量名=变量值; export 变量名`

```
[root@MissHou ~]# export A=hello     临时将一个本地变量（临时变量）变成环境变量
[root@MissHou ~]# env|grep ^A
A=hello

永久生效:
vim /etc/profile 或者 ~/.bashrc
export A=hello
或者
A=hello
export A

说明：系统中有一个变量PATH，环境变量
export PATH=/usr/local/mysql/bin:$PATH
```

㈢ 全局变量

- **全局变量**：全局所有的用户和程序都能调用，且继承，新建的用户也默认能调用.
- **解读相关配置文件**

文件名	说明	备注
$HOME/.bashrc	当前用户的bash信息,用户登录时读取	定义别名、umask、函数等
$HOME/.bash_profile	当前用户的环境变量，用户登录时读取	
$HOME/.bash_logout	当前用户退出当前shell时最后读取	定义用户退出时执行的程序等
/etc/bashrc	全局的bash信息，所有用户都生效	
/etc/profile	全局环境变量信息	系统和所有用户都生效
$HOME/.bash_history	用户的历史命令	history -w 保存历史记录 history -c 清空历史记录

说明：以上文件修改后，都需要重新source让其生效或者退出重新登录。

- **用户登录**系统**读取**相关文件的顺序
  1. `/etc/profile`
  2. `$HOME/.bash_profile`
  3. `$HOME/.bashrc`
  4. `/etc/bashrc`

5. `$HOME/.bash_logout`

㈣ 系统变量

- **系统变量(内置bash中变量)**： shell本身已经固定好了它的名字和作用.

内置变量	含义
$?	上一条命令执行后返回的状态；状态值为0表示执行正常，非0表示执行异常或错误
$0	当前执行的程序或脚本名
$#	脚本后面接的参数的个数
$*	脚本后面所有参数，参数当成一个整体输出，每一个变量参数之间以空格隔开
$@	脚本后面所有参数，参数是独立的，也是全部输出
$1~$9	脚本后面的位置参数，$1表示第1个位置参数，依次类推
${10}~${n}	扩展位置参数,第10个位置变量必须用{}大括号括起来(2位数字以上扩起来)
$$	当前所在进程的进程号，如`echo $$`
$!	后台运行的最后一个进程号 (当前终端)
!$	调用最后一条命令历史中的参数

- 进一步了解位置参数`$1~${n}`

```
#!/bin/bash
#了解shell内置变量中的位置参数含义
echo "\$0 = $0"
echo "\$# = $#"
echo "\$* = $*"
echo "\$@ = $@"
echo "\$1 = $1"
echo "\$2 = $2"
echo "\$3 = $3"
echo "\$11 = ${11}"
echo "\$12 = ${12}"
```

- 进一步了解$*和$@的区别

`$*`：表示将变量看成一个整体 `$@`：表示变量是独立的

```
#!/bin/bash
for i in "$@"
do
echo $i
done

echo "======我是分割线======="
```

```
for i in "$*"
do
echo $i
done

[root@MissHou ~]# bash 3.sh a b c
a
b
c
======我是分割线=======
a b c
```

三、简单四则运算

算术运算：默认情况下，shell就只能支持简单的整数运算

运算内容：加(+)、减(-)、乘(*)、除(/)、求余数（%）

1. 四则运算符号

表达式	举例
$(( ))	echo $((1+1))
$[ ]	echo $[10-5]
expr	expr 10 / 5
let	n=1;let n+=1 等价于 let n=n+1

2.了解i++和++i

- 对变量的值的影响

```
[root@MissHou ~]# i=1
[root@MissHou ~]# let i++
[root@MissHou ~]# echo $i
2
[root@MissHou ~]# j=1
[root@MissHou ~]# let ++j
[root@MissHou ~]# echo $j
2
```

- 对表达式的值的影响

```
[root@MissHou ~]# unset i j
[root@MissHou ~]# i=1;j=1
[root@MissHou ~]# let x=i++          先赋值，再运算
[root@MissHou ~]# let y=++j          先运算，再赋值
[root@MissHou ~]# echo $i
2
[root@MissHou ~]# echo $j
2
[root@MissHou ~]# echo $x
1
[root@MissHou ~]# echo $y
2
```

四、扩展补充

1. 数组定义

㈠ 数组分类

- 普通数组：只能使用整数作为数组索引(元素的下标)
- 关联数组：可以使用字符串作为数组索引(元素的下标)

㈡ 普通数组定义

- 一次赋予一个值

```
数组名[索引下标]=值
array[0]=v1
array[1]=v2
array[3]=v3
```

- 一次赋予多个值

```
数组名=(值1 值2 值3 ...)
array=(var1 var2 var3 var4)

array1=(`cat /etc/passwd`)        将文件中每一行赋值给array1数组
array2=(`ls /root`)
array3=(harry amy jack "Miss Hou")
array4=(1 2 3 4 "hello world" [10]=linux)
```

㈢ 数组的读取

```
${数组名[元素下标]}

echo ${array[0]}          获取数组里第一个元素
echo ${array[*]}          获取数组里的所有元素
echo ${#array[*]}         获取数组里所有元素个数
echo ${!array[@]}         获取数组元素的索引下标
echo ${array[@]:1:2}      访问指定的元素；1代表从下标为1的元素开始获取；2代表获取后面几个元素

查看普通数组信息:
[root@MissHou ~]# declare -a
```

㈣ 关联数组定义

①首先声明关联数组

```
declare -A asso_array1
declare -A asso_array2
declare -A asso_array3
```

② 数组赋值

- 一次赋一个值

```
数组名[索引or下标]=变量值
# asso_array1[linux]=one
# asso_array1[java]=two
# asso_array1[php]=three
```

- 一次赋多个值

```
# asso_array2=([name1]=harry [name2]=jack [name3]=amy [name4]="Miss Hou")
```

- 查看关联数组

```
# declare -A
declare -A asso_array1='([php]="three" [java]="two" [linux]="one" )'
declare -A asso_array2='([name3]="amy" [name2]="jack" [name1]="harry" [name4]="Miss Hou" )'
```

- 获取关联数组值

```
# echo ${asso_array1[linux]}
one
# echo ${asso_array1[php]}
three
# echo ${asso_array1[*]}
three two one
# echo ${!asso_array1[*]}
php java linux
# echo ${#asso_array1[*]}
```

```
3
# echo ${#asso_array2[*]}
4
# echo ${!asso_array2[*]}
name3 name2 name1 name4
```

2. 其他变量定义

- 取出一个目录下的目录和文件：`dirname`和`basename`

```
# A=/root/Desktop/shell/mem.txt
# echo $A
/root/Desktop/shell/mem.txt
# dirname $A   取出目录
/root/Desktop/shell
# basename $A  取出文件
mem.txt
```

- 变量"内容"的删除和替换

```
一个“%”代表从右往左去掉一个/key/
两个“%%”代表从右往左最大去掉/key/
一个“#”代表从左往右去掉一个/key/
两个“##”代表从左往右最大去掉/key/

举例说明:
# url=www.taobao.com
# echo ${#url}          获取变量的长度
# echo ${url#*.}
# echo ${url##*.}
# echo ${url%.*}
# echo ${url%%.*}
```

- 以下了解，自己完成

```
替换: / 和 //
 1015  echo ${url/ao/AO}
 1017  echo ${url//ao/AO}    贪婪替换

替代:  - 和 :-  +和:+
 1019  echo ${abc-123}
 1020  abc=hello
 1021  echo ${abc-444}
 1022  echo $abc
 1024  abc=
 1025  echo ${abc-222}

${变量名-新的变量值} 或者 ${变量名=新的变量值}
变量没有被赋值：会使用“新的变量值“ 替代
变量有被赋值（包括空值）：  不会被替代
```

```
 1062  echo ${ABC:-123}
 1063  ABC=HELLO
 1064  echo ${ABC:-123}
 1065  ABC=
 1066  echo ${ABC:-123}
```

${变量名:-新的变量值} 或者 ${变量名:=新的变量值}
变量没有被赋值或者赋空值：会使用“新的变量值“ 替代
变量有被赋值： 不会被替代

```
 1116  echo ${abc=123}
 1118  echo ${abc:=123}
```

```
[root@MissHou ~]# unset abc
[root@MissHou ~]# echo ${abc:+123}

[root@MissHou ~]# abc=hello
[root@MissHou ~]# echo ${abc:+123}
123
[root@MissHou ~]# abc=
[root@MissHou ~]# echo ${abc:+123}
```

${变量名+新的变量值}
变量没有被赋值或者赋空值：不会使用“新的变量值“ 替代
变量有被赋值： 会被替代

```
[root@MissHou ~]# unset abc
[root@MissHou ~]# echo ${abc+123}

[root@MissHou ~]# abc=hello
[root@MissHou ~]# echo ${abc+123}
123
[root@MissHou ~]# abc=
[root@MissHou ~]# echo ${abc+123}
123
```

${变量名:+新的变量值}
变量没有被赋值：不会使用“新的变量值“ 替代
变量有被赋值（包括空值）： 会被替代

```
[root@MissHou ~]# unset abc
[root@MissHou ~]# echo ${abc?123}
-bash: abc: 123

[root@MissHou ~]# abc=hello
[root@MissHou ~]# echo ${abc?123}
hello
[root@MissHou ~]# abc=
[root@MissHou ~]# echo ${abc?123}
```

${变量名?新的变量值}
变量没有被赋值:提示错误信息
变量被赋值（包括空值）：不会使用“新的变量值“ 替代

```
[root@MissHou ~]# unset abc
```

```
[root@MissHou ~]# echo ${abc:?123}
-bash: abc: 123
[root@MissHou ~]# abc=hello
[root@MissHou ~]# echo ${abc:?123}
hello
[root@MissHou ~]# abc=
[root@MissHou ~]# echo ${abc:?123}
-bash: abc: 123
```

${变量名:?新的变量值}
变量没有被赋值或者赋空值时:提示错误信息
变量被赋值：不会使用“新的变量值“ 替代

说明：?主要是当变量没有赋值提示错误信息的，没有赋值功能