**shell编程基础**

**一、、shell编程之变量**

**1、什么是变量？**

内存空间地址+数据，数据是放在内存中，即也叫有名称的内存地址

变量的值的类型，决定了存储格式、表示的数据范围、参与的运算

**2、变量赋值**

name=VALUE

**3、bash对变量的机制**

1. 所有都看作字符型
2. 不支持浮点数据，需借用外部机制
3. 变量无需事先申明，相当于，赋值和申明同时实现

**4、程序中变量引用的机制**

把变量名出现的位置，直接替换为其所指向的内存空间中的数据

**5、变量的命名规范**

1. 不能以数字开头，只能包含数字、字母、下划线
2. 变量名要做到见名知义
3. 不能使用程序语言的保留字（if else where等）

**6、变量的引用**

1. ${var\_name}
2. $var\_name

**7、变量的类型**

1. 本地变量:作用范围仅为当前shell;
2. 环境变量:作用范围当前进程和其子进程
3. 局部变量:作用范围为某片代码片；
4. 特殊变量:shell内置的，有特殊功用
   * 0 = 成功
   * 1-255 = 失败

**8、各变量的赋值、引用 、查看、删除**

* **本地变量**
  + 变量的定义

\* var\_name=VALUE

* + 变量的引用

\* ${var\_name} ,$name

* + 变量的查看

\* set

* + 变量的撤消

\* unset var\_name (此次不需要加$)

* **环境变量**
  + 变量的定义
  + \* export var\_name=value
  + \* var\_name=value --> export var\_name
  + \* declare -x var\_name=VALUE （-x 表示定义为一个环境变量）

\* declare -i var\_name=VALUE （直接将var\_name定义为整形）

* + 变量的引用
  + \* ${var\_name}

\* $var\_name

* + 变量的查看
  + \* export
  + \* declare -x
  + \* printenv

\* env

* + 变量的撤消

\* unset var\_name

* **只读变量的定义**
  + declare -r var\_name=VALUE (只读变量只有进程结果后才会消息，如果定义在环境配置文件中，需要手动清除)
  + readonly name
* **增强型变量赋值**
  + i=1
  + i=i+1]
  + let i+=1
  + let i+=2
* **变量做某种运算后存至变量中**
  + let i=$i+# 等同 let i+=#

 +=, -=, \*=, /=, %= 都需要使用`let`命令进行描述

* + 变量的自增

 var=$[$var+1]  == let var+=1 == let var++

* + 变量的自减

 var=$[$var-1] == let var-=1 == let var--

**二、bash之切片**

**1、基于字符串切片**

* Usage: ${var:offset: length}
* **使用示例：**
* [root@localhost ~]#mypath="/etc/sysconfig/network-scripts/"    #定义一个变量，等会切这个变量
* [root@localhost ~]# echo ${mypath:5}       #偏移5个字符显示
* sysconfig/network-scripts/
* [root@localhost ~]# echo ${mypath:10}     #偏移10个字符显示
* nfig/network-scripts/
* [root@localhost ~]# echo ${mypath:5:5}    #偏移5个字符，取5个字符
* sysco
* 取出字符串的最后几个字符：${var: -length}
* 注意：-length之前有空白字符；
* [root@localhost ~]# echo ${mypath: -10}

 k-scripts/

**2、基于模式取子串**

* **Usage:**
  1. ${var#\*word}:自左而右，查找var变量中存储的字符串中第一次出现的由word所指明的字符，删除此字符及其左侧的所有内容
  2. ${var##\*word}:自左而右，查找var变量中存储的字符串中最后一次出现的由word所指明的字符，删除此字符及其左侧的所有内容
  3. ${var%word\*}:自右而左，查找var变量中存储的字符串中第一次出现的由word所指明的字符，删除字符及其右侧的所有内容
  4. ${var%%word\*}:自右而左，查找var变量中存储的字符串中最后一次出现的由word所指明的字符，删除此字符及其右侧的所有内容
* **使用示例：**
* [root@localhost ~]#mypath="/etc/sysconfig/network-scripts"
* [root@localhost ~]# echo ${mypath#\*/}
* etc/sysconfig/network-scripts
* [root@localhost ~]#mypath="/etc/sysconfig/network-scripts"
* [root@localhost ~]# echo ${mypath##\*/}
* network-scripts
* [root@localhost ~]#mypath="/etc/sysconfig/network-scripts"
* [root@localhost ~]# echo ${mypath%c\*}
* /etc/sysconfig/network-s
* [root@localhost ~]# echo ${mypath%%c\*}

 /et

**3、基于字串查找替换**

* **Usage:**
  1. ${var/pattern/replacement} :查找var变量存储的字符中第一次由pattern匹配到的内容，并替换为replacement
  2. ${var//pattern/replacement} :查找var变量存储的字符中所有能够由pattern匹配到的内容，并替换为replacement
  3. ${var/#pattern/replacement} :查找var变量存储的字符中最开始处能够由pattern匹配到的内容，并替换为replacement
  4. ${var/%pattern/replacement} : 查找var变量存储的字符中最后位置能够由pattern匹配到的内容，并替换为replacement

**示例：**

   [root@localhost ~]#url="http://www.baidu.com:80"

   [root@localhost ~]# echo ${url/www/WWW}

   http://WWW.baidu.com:80

   [root@localhost ~]# echo ${url/w/W}

   http://Www.baidu.com:80

   [root@localhost ~]# echo ${url//w/W}

   http://WWW.baidu.com:80

   [root@localhost ~]# userinfo="root:x:0:0:rootuser:/root:/bin/bash"

   [root@localhost ~]# echo ${userinfo/#root/ROOT}

   ROOT:x:0:0:root user:/root:/bin/bash

   [root@localhost ~]# userinfo="root:x:0:0:rootuser:/root:/bin/root"

   [root@localhost ~]# echo ${userinfo/%root/ROOT}

   root:x:0:0:root user:/root:/bin/ROOT

**4、基于字串查找删除**

* **Usage:**
  1. ${var/pattern}：查找var变量存储的字符中第一次由pattern匹配到的内容，并删除；
  2. ${var//pattern}：查找var变量存储的字符中所有能够由pattern匹配到的内容，并删除；
  3. ${var/#pattern}：查找var变量存储的字符中最开始处能够由pattern匹配到的内容，并删除；
  4. ${var/%pattern}：查找var变量存储的字符中最后位置能够由pattern匹配到的内容，并删除；
* **示例：**
* [root@localhost ~]# userinfo="root:x:0:0:rootuser:/root:/bin/root"
* [root@localhost ~]# echo ${userinfo/root}
* :x:0:0:root user:/root:/bin/root
* [root@localhost ~]# echo ${userinfo//root}
* :x:0:0: user:/:/bin/
* [root@localhost ~]# echo ${userinfo/#root}
* :x:0:0:root user:/root:/bin/root
* [root@localhost ~]# echo ${userinfo/%root}

 root:x:0:0:root user:/root:/bin/

**5、基于字符串大小写转换**

* **Usage:**
  1. ${var^^}：把var变量中的所有小写字母，统统替换为大写；
  2. ${var,,}：把var变量中的所有大写字母，统统替换为小写；
* **示例**
* [root@localhost ~]# echo $userinfo
* root:x:0:0:root user:/root:/bin/root
* [root@localhost ~]# myinfo=${userinfo^^}
* [root@localhost ~]# echo $myinfo
* ROOT:X:0:0:ROOT USER:/ROOT:/BIN/ROOT
* [root@localhost ~]# echo ${myinfo,,}

 root:x:0:0:root user:/root:/bin/root

**6、空变量判断赋值**

* **Usage:**
  1. ${var:-word}：如果变量var为空或未声明，则返回word所表示的字符串；否则，则返回var变量的值, 临时赋值
  2. [root@localhost ~]# echo $name
  3. #这行的值为空
  4. [root@localhost ~]# echo ${name:-tom}
  5. tom
  6. [root@localhost ~]# name=hello

[root@localhost ~]# echo ${name:-tom}            hello

* 1. ${var:=word}：如果变量var为空或未声明，则返回word所表示的字符串，并且把word赋值为var变量；否则，则返回var变量的值，直接等值
  2. [root@localhost ~]# echo $name
  3. #这行的值为空
  4. [root@localhost ~]# name=${name:-tom}
  5. [root@localhost ~]# echo $name
  6. tom
  7. [root@localhost ~]# name=${name:-jerry}
  8. [root@localhost ~]# echo $name

tom

* 1. ${var:?error}：如果变量var为空或未声明，则返回error为错误信息；否则，则返回var变量的值；
  2. [root@localhost ~]# echo "User's name is${name:?wrong}"
  3. -bash: name: wrong
  4. [root@localhost ~]# name=tom
  5. [root@localhost ~]# echo "User's name is${name:?wrong}"

User's name is tom

* 1. ${var:+word}：如果变量var为空或未声明，忽略；否则，则返回word；
  2. [root@localhost ~]# unset name
  3. [root@localhost ~]# echo "User's name is${name:+wrong}"
  4. User's name is
  5. [root@localhost ~]# name=tom
  6. [root@localhost ~]# echo "User's name is${name:+wrong}"

User's name is wrong

===============

**三、bash多命令执行**

**1、无逻辑关系的**

* COMMAND1;COMMAND2;COMMAND3...

**2、有逻辑关系的**

* 逻辑的运算结果状态：
  + ! 1 = 0
  + ! 0 = 1
  + 1 || 0 = 1
  + 1 || 1 = 1
  + 0 || 1 = 1
  + 0 || 0 = 0

 只要一个为真，即为真，（类似电路并链）

* + 1&&1=1
  + 1&&0=0
  + 0&&1=0
  + 0&&0=0

只有两个都为真，结果才为真，否则为假（遵守短路法则）

* + 与运算（乘法）
  + 或运算 ：加法
  + 非：取反
  + 真（True ，1 ）
  + 假 (False ,0 ）
* 异或：判断是否不同
  + 不同者为真，相同者为零假

**3、短路法则：**

* COMMAND1 && COMMAND2
* COMMAND1为『假』，COMMAND2不会再执行，

 COMMAND1为『真』，COMMAND2必须执行

* COMMAND1 || COMMAND2
* COMMAND1为『真』；COMMAND2不会执行

 COMMAND1为『假』；COMMAND2必须执行

**四、bash的配置文件**

**1、bash配置文件的分类**

* profile类：为交互式shell提供配置文件
  + ~/.bash\_profile
  + /etc/profile
  + /etc/profile.d/\*.sh
  + 全局配置
  + 用户个人
* bashrc类: 为非交互式用户提供配置文件
  + ~/.bashrc
  + /etc/bashrc
  + 全局配置
  + 用户个人

**2、profile、bashrc类的功用**

* profile类
  + 定义用户的全局变量
  + 运行命令和脚本
* bashrc类
  + 定义本地变量
  + 定义别名

**3、登录式shell、非登录shell读取配置文件的顺序**

1. 登录式shell读取顺序
   * /etc/profile --> /etc/profile.d/\*.sh --> ~/.bash\_profile --> ~/.bashrc --> /etc/bashrc
2. 非登录式shell读取顺序
   * ~/.bashrc --> /etc/bashrc --> /etc/profile.d/\*.sh

**4、让定义的配置文件立即生效**

* source /PATH/TO/PROFILE
* . /PATH/TO/PROFILE

**注意： 需要定义全局的别名，需修改/etc/bashrc文件**

**五、bash之算术运算**

1. let命令
   * let SUM=$num1+$num2
   * echo $SUM
2. 算术运算表达式一：
   * echo $[ $num1 + $num2 ]
3. 算术运算表达式二：
   * echo $(($num1 + $num2))
4. 算术运算表达式三：
   * expr $num1 + $num2
5. bash的运算符
   * + - \* / \*\* %

**注意：乘法符号在有些场景中使用需要加转义符**

**六、Bash之条件测试：**

判断某需求是否满足，需要由测试机制来实现

**1. 使用执行命令，并利用命令状态返回值来判断**

* 0：成功，
* 1-255：失败

**2. 测试表达式格式**

* test expression
* [ expression ] :括号前后两端必须有空格，否则为语法错误
* [[  expression ]] :括号前后两端必须有空格，否则为语法错误, 多为测试字符串

**3、bash的测试类型：**

**数据测试**

* **数值比较**
  + 注意：可以使用[],[[ ]],建议使用[]
  + -eq : 是否等；[ $nu1 -eq $num2 ]
  + -ne : 是否不等于；
  + -gt : 是否大于
  + -ge : 大于等于
  + -lt : 是否小于
  + -le : 是否小于等于
* **字符串测试  : 1、字符串要加引号引用 ；2、做比较时使用双中括号[[ ]]**
  + == ：是否等于
  + > : 是否大于，此处也要使用双中括号
  + < : 是否小于，此处也要使用双中括号
  + != :是否不等于
  + =~ ：左侧的字符串是否能被右侧的pattern所匹配，匹配为一部分，不是精确匹配
  + -z “STRING” : 判断指定的字串是否为空，空则为真，不空则假
  + -n “string” :判断指定的字串是否不空，空则为假，不空则真

**文件测试**

* **存在性测试：**
  + -a FILE
  + -e FILE

 文件的存在性测试，存在则为真，否则则为假

* **存在性及文件类型**
  + -b FILE : 是否存在，并且为块设备文件
  + -c FILE : 是否存在，并且为字符设备文件
  + -d FILE : 是否存在，并且为目录文件
  + -f FILE ： 是否存在， 并且为普通文件
  + -h FILE 或者 -L FILE : 是否存在并且为符号连接文件
  + -p FILE : 是否存在， 且为命名管道文件
  + -S FILE ：是否存在，且为套接字文件
* **文件权限测试**
  + -r FILE : 是否存在，并且可读，对当前用户可读
  + -w FILE : 是否存在， 并且可写
  + -x FILE : 是否存在， 并且可执行
* **特殊权限测试**
  + -g FILE : 是否存在， 并且拥有SGID权限
  + -u FILE : 是否存在，并且拥有SUID权限
  + -k FILE : 是否存在， 且拥有sticky权限
* **文件是否有内容**
  + -s FILE :是否存在，并且是否有内容
* **时间戳测试**
  + -N FILE : 文件自从上一次读取操作后，是否被修改过
* **从属关系测试：**
  + -O  FILE : 当前用户是否为文件的属主
  + -G  FILE ： 当前用户是否属于文件的属组
* **双目测试（文件新旧对比）**
  + FILE1 -ef FILE2 : FILE1 与FILE2是否为指向同一个文件系统上的相同的inode的硬连接
  + FILE1 -nt FILE2 : FILE1是否新于FILE2
  + FILE1 -ot FILE2 : FILE1是否旧于FILE2
* **组合测试条件**
  + **第一种：**
  + **第二种：**
  + 例：[ -O FILE ] && [ -r FILE] :是否可读并且为属主
  + COMMAND1 && COMMAND2
  + COMMAND1 || COMMAND2
  + !COMMAND
  + 例： [ -O FILE -a -x FILE ] : 是否是属主并且有执行权限
  + [  expression1 -a expression2  ]
  + [  expression1 -o expression2  ]
  + ![  expression1 ]
  + **逻辑运算：**

**注意：-a -o 只能在[]中使用，如果需要在双括号中使用逻辑换算，需要使用 && || 来代替**

**七、Bash脚本之状态返回值：**

* 默认是脚本中执行的最后一条命令的状态返回值。
* 自定义状态退出状态码：
  + exit [n] :n为自己指定的状态码；0表示成功，非零表示失败

注意：shell进程遇到exit时，即会中止，因此整个脚本执行即为结束

**八、bash编程之向脚本传递参数**

1. **位置参数变量**
   * 例：myscript.sh argu1 argu2
2. **脚本中引用方式:**
   * 2…….{11},….
3. **位置参数变量轮替**
   * shift   : 把引用过的参数T掉，在脚本中永远只使用$1引用下一个

 例： shift 2 : 踢两个，这样就可以在脚本中永远以$1和$2来引用后面的参数

**九、bash编程之特殊变量**

* **$0 :脚本文件路径本身； 取文件名：basename $0**
* **$# : 脚本参数个数**
* **$\* : 所有参数，每个参数当作为一个字串，一般用于保存用户的参数**
* **$@: 所有参数，所有参数当作为一个字串，一般用于保存用户的参数**

**十、bash编程之语句类型**

**1、顺序执行：逐条运行**

**2、选择执行：**

* 两个或者以上的分支：满足条件时只会执行其中一个满足条件的分支

**3、循环执行：**

* 某代码片断（循环体）要执行0、1或多个来回

**十一、脚本之语法错误检查**

* bash -n script.sh
* bash -x script.sh

**十二、脚本之用户交互的实现**

1. read [option]...[name...]
   * -p PROMPT: 直接提示输入信息，保存至变量
   * -t TIMEOUT : 超时设置，默认秒为单位

**十三、选择执行：**

**1、单分支的if语句**

if 测试条件；then

   代码分支

fi

**2、双分支的if语句**

if 测试条件;then

   条件为真时执行的分支

else

   条件为假时执行的分支

fi

**3、多分支的if语句**

if 测试条件；then

   条件为真时执行

elif

   条件为真时执行

else

   条件为假是执行

fi

**十四、循环执行**

**1、for循环语句**

for VAR in LIST;do

   循环体

done

* **进入条件：只要列表有可用元素，即可进入循环**
* **退出条件：列表中的元素遍历完成后，即退出循环**
* **列表的生成方式：**
  + a:{start..end}
  + b: $(seq 起始数值 [步长] 结束数值)
  + 直接给出
  + 整数列表

**2、while循环语句**

   while CONDITION1;do

       循环体

       循环控制变量修正表达式

   done

* **条件进入：CONDITION测试为真**
* **条件退出：CONDITION测试为假**

**3、until循环语句**

   until CONDITION;do

       循环体

       循环控制变量修正表达式

   done

* **条件进入：CONDITION测试为假**
* **条件退出：CONDITION测试为真**

**4、case选择执行语法**

case $variable in

   pattern1)

       分支1

       ；；

   pattern2)

       分支2

       ；；

   pattern3)

       分支3

       ；；

   \*）

       分支N

esac

* **case支持glob网络的通配符：**
* \*
* ？
* []

 a|b :a或者b

**十五、在循环中控制循环的方法**

**1、continue**

表示提前结束本轮循环，而直接进入下一轮循环的条件测试

   while CONDITION1;do

       COMMNAD1

       if CONDITION2;then

           continue

       fi

           COMMNAD N

   done

* **示例 ： 求100以内的所有偶数之和**
* #!/bin/bash
* declare  -i evensum=0
* declare -i i=0
* while [ $i -le 100 ];do
* let i++
* if [ $[$i%2] -eq 1];then
* continue
* fi
* let evensum+=$i

         done

**2、break**

表示提前跳出循环

* **示例：100以内所有奇数之和**
* #!/bin/bash
* declare  -i evensum=0
* declare -i i=1
* while true;do
* let $oddsum+=$i
* let i+=2
* if [ $i -gt 100 ];then
* break
* fi

 done

**3、while的特殊用法之遍历文件的行**

while read VARIABLE;do

   循环休

done < /path/form/somefile

**4、for循环的特殊用法**

for ((控制变量初始化；条件判断表达式；控制变量的修正语句))；do

   循环体

done

**控制变量初始化：仅在循环代码开始运行时，执行一次**

**控制变量的修正语句：每轮循环结束会先进行控制变量修正运算，而后再做判断**

**十六、shell函数**

把一段独立功能的代码当作一个整体，并定义一个函数名字，命名的代码段，就称之为一个函数。定义的函数代码段不会自动执行，需在调用的时候执行。调用函数就是批是代码中给出指定的函数名即可。函数可出现在任何位置，在代码执行时，都会被自动替换为函数代码。其函数命名不应该为命令名。其可在过程式编程中实现代码重用。以此实现模块化编程和结构化编程.

**1、函数定义方式**

语法一

   function f\_name {

       ...函数体...

   }

语法二

   f\_name () {

       ...函数体...

   }

**2、函数的生命周期**

* 每次被调用时创建，返回时终止

**3、函数的返回值**

* 函数返回值：
  + (1) 默认取决于函数体中执行的最后一条命令的退出状态码；
  + (2) 自定义：return
  + (1) 使用echo或printf命令进行输出；
  + (2) 函数体中调用的命令的执行结果；
  + 函数的执行结果返回值：
  + 函数的退出状态码：
* **示例：给定一个用户名，取得用户的id号和默认shell**
* #!/bin/bash
* #
* userinfo() {
* if id "$username" &> /dev/null; then
* grep "^$username\>" /etc/passwd | cut -d: -f3,7
* else
* echo "No such user."
* fi
* }
* username=$1
* userinfo
* username=$2
* userinfo

**4、传递参数给函数**

* 在函数体中当中，可以使用2, …引用传递给函数的参数；还可以函数中使用@引用所有参数，$#引用传递的参数的个数；
* 在调用函数时，在函数名后面以空白符分隔给定参数列表即可，例如，testfunc  arg1 arg2 arg3 ..
* **示例：添加10个用户，其添加用户的功能使用函数实现，用户名做为参数传递给函数**
* #!/bin/bash
* #
* # 5: user exists
* addusers() {
* if id $1 &> /dev/null; then
* return 5
* else
* useradd $1
* retval=$?
* return $retval
* fi
* }
* for i in {1..10}; do
* addusers ${1}${i}
* retval=$?
* if [ $retval -eq 0 ]; then
* echo "Add user ${1}${i} finished."
* elif [ $retval -eq 5 ]; then
* echo "user ${1}${i} exists."
* else
* echo "Unkown Error."
* fi

             done

**5、函数变量的作用域**

* 局部变量
  + lacal VARIABLE=VALUE
  + 作用域是函数的生命周期，在函数结束时被自动销毁，定义的方法如下：
* 本地变量
  + 作用域是运行脚本的shell进程的生命周期，作用范围为当前shell脚本程序文件
* **示例**
* #!/bin/bash
* #
* name=tom
* f\_name() {
* local name=jerry
* }
* f\_name
* echo $name

 注意：以上正常情况下会显示name=tom,如果不使用local定义为本地函数变量， 那么应该name=jerry.

**注意：在函数中定义变量尽量使用local来定义变量**

**十七、bash之数组**

**1、什么是数组？**

存储多个元素的连续的内存空间，其数组只有一个名字，其数组的索引号从0开始。

**2、定义一个数组**

* declare -a NAME : 声明一个索引数组
* declare -A NAME : 声明一个关联数组
  + 索引数组为索引号（下标）从0开始，关联数组的索引号可以自定义，其bash4及以后版本支持关联数组

**3、数组中元素的赋值方式**

* 1、一次只赋值一个元素
  + NAME[0]=pig
  + NAME[1]=dog
* 2、一次赋值全部元素
  + NAME=(“VAL1”,”VAL2”,”VAL3”,…)
* 3、只赋值特定元素
  + NAME=（[0]=”val1” [3]=”val3”,….）: 稀疏格式的数组定义
* 4、read -a array\_name
  + val1 val2 val3…..(直接在提示符中写，写后回车即可)
* 5、declare -A world
  + world[us]=”america”
  + world[uk]=”United kingdom”  :这种方式即关联数组，直接给定下标名称

**4、数组中的元素引用**

* echo ${name[0]}
  + 引用时只给数组名，表示引用为下标为0的元素

**5、数组的长度引用 （即数组中元素的个数 ）**

* ${#ARRAY\_NAME[\*]}
* ${#ARRAY\_NAME[@]}

 注意：${#ARRAY\_NAME}，表示引用第一个元素的字符串长度

**5、引用数组中的所有元素**

* ${array\_name[\*]}
* ${array\_name[@]}

**6、数组元素切片**

* ${array\_name[@]:offset:number}
  + offset :要跳过元素的个数
  + number :要取出元素的个数，省略number时，表示取偏移量之后的所有元素

**7、向非稀疏数组中追加元素**

* array\_name[${array\_name[\*]}]=

**8、删除数组中的某元素**

* unset array[index]

**9、关联数组的定义**

* declare -A array\_name
* array\_name=([index\_name1]=”val1” [index\_name2]=”val2”)

**十八、bash之信号捕捉**

**1、trap 命令**

* -l : 列出所有信号
* man 7 signal : 查看所有信号的功用
* kill -l : 查看所有信号
* **向脚本传递信号示例**
* #!/bin/bash
* #
* trap 'echo "dou ni wan er."' INT
* trap 'echo "quit";exit1' INT

 注意：可以把捕捉信号定义在一个函数里，在其它位置调用，脚本参考<脚本合集>