1、shell中数组的定义及使用

```
1. 在shell中定义的数组只支持一维数组，不支持多维数组。

2. shell中定义数组并进行初始化的方式
    数组名=(初始值0 初始值1 初始值2 初始值3 .....)
    数组名=([0]=初始值0 [1]=初始值1 [2]=初始值2 [3]=初始值3 .....)

    定义数组时，可以指定只对部分成员初始化：
    数组名=([0]=初始值0 [3]=初始值3 [5]=初始值5 [8]=初始值8 .....)

    1> 数组中的每个成员依然属于字符串。
    2> 数组中每个成员的下标从0开始。

3> 修改数组中某个成员的值或者对数组的某个成员赋值
    数组名[下标]=值

4> 访问数组中的成员
    ${数组名[下标]}        --> 访问数组的某个成员
    ${数组名[*]}           --> 获取数组中的所有的成员
```

```
    ${数组名[@]}          --> 获取数组中的所有的成员

    ${#数组名[下标]}      --> 计算数组中成员的字符的个数
    ${#数组名[*]}         --> 统计数组中的所有的成员个数之和
    ${#数组名[@]}         --> 统计数组中的所有的成员个数之和

5> 数组的拼接
    数组名=(${数组名1[*]} ${数组名2[*]})
    数组名=(${数组名1[@]} ${数组名2[@]})
```

```bash
#!/bin/bash
# your code

echo "--- 1. 定义数组 ---"
array1=(aaa bbb ccc ddd eee)
echo "array1数组中的所有的成员 = ${array1[*]}"
echo "array1数组中的成员的个数 = ${#array1[*]}"

array2=([0]=111 [1]=222 [2]=333 [3]=444 [4]=555)
echo "array2数组中的所有的成员 = ${array1[@]}"
echo "array2数组中的成员的个数 = ${#array1[@]}"

# 定义数组时，对指定的成员初始化
array3=([1]=111 [2]=aaa [5]=555 [8]=eee [10]=fff)
echo "array3数组中的所有的成员 = ${array3[@]}"
# 数组中成员的个数是5个，跟下标的大小没有关系
echo "array3数组中的成员的个数 = ${#array3[@]}"

echo "--- 2. 可以追加数组的长度 ----"
array1[5]=fff
array1[6]=ggg
echo "array1数组中的所有的成员 = ${array1[*]}"
echo "array1数组中的成员的个数 = ${#array1[*]}"

array3[0]=000
array3[3]=333
array3[4]=bbb
echo "array3数组中的所有的成员 = ${array3[@]}"
echo "array3数组中的成员的个数 = ${#array3[@]}"

echo "---3. 数组的拼接 ---"
array=(${array1[*]} ${array2[*]})
echo "array数组中的所有的成员 = ${array[@]}"
echo "array数组中的成员的个数 = ${#array[@]}"

echo "--- 4. 计算数组中每个成员的字符串的长度 ---"
echo "array[3] 成员的字符串的长度 = ${#array[3]}"
```

2、read和echo命令的使用

2.1 read命令

```
作用 ：  从终端接收字符串赋值给变量

read  var1      ----> 接收字符串赋值给变量var1
read  var1  var2   ---> 从终端接收字符串分别赋值给var1和var2
                        输入的字符串以空格或者tab键分隔，不可以使用回车分隔

read -p "提示字符串" var1 ---> 先在终端输出提示字符串之后，
                    然后在提示字符串的后边输入字符串，并接收对应的字符串，
                    赋值给变量var1
read -t 秒数  var1  ----> “秒数”之后，停止接收并退出
read -n 个数  var1  ----> 接收"个数"字符之后，退出停止接收
read -s var1        ----> 接收但是不回显，类似密码
```

```
对于单个命令的测试在终端执行对应的命令即可：
linux@ubuntu:day03$ read name
zhangsan
linux@ubuntu:day03$ echo $name
zhangsan
linux@ubuntu:day03$ read name  sex
zhangsan man        ---> 空格分隔-->ok
linux@ubuntu:day03$ echo $name $sex
zhangsan man
linux@ubuntu:day03$ read name  sex
lisi    woman       ---> tab键分隔，  ok
linux@ubuntu:day03$ echo $name $sex
lisi woman
linux@ubuntu:day03$ read name  sex
wanger              ---> 回车分隔，提前结束
linux@ubuntu:day03$ echo $name $sex
wanger

linux@ubuntu:day03$ read -p "please input your name & sex > " name sex
                    ---> 先输出提示符，在输入对应的内容
please input your name & sex > zhangsan man
linux@ubuntu:day03$ echo $name $sex
zhangsan man
linux@ubuntu:day03$ read -n 5 number        ---> 输入5个字符之后退出
12345linux@ubuntu:day03$ echo $number
12345
linux@ubuntu:day03$ read -t 10 number       ---> 10s之后退出
linux@ubuntu:day03$ read -s passwd      ---> 取消输入的回显
linux@ubuntu:day03$ echo $passwd

```

2.2 echo 命令

```
作用：输出字符串或变量的值到终端
echo  "hello world"
echo hello world
echo 'hello world'          # OK

echo ${var1}        #ok
echo ${var1}  ${var2}
echo -n "hello world"       --> 取消换行符的输出
echo '' | echo | echo ""    ---> 输出一个换行符
```

3、ubuntu系统环境变量的配置

```
1. 自己编译的程序和shell命令对比
    自己编译的程序的默认名字为a.out, a.out是一个elf格式的文件，
要想执行自己编译的应用程序需要指定可执行程序路径。
可执行程序的路径/a.out。比如./a.out

    shell命令本质也是一个elf格式的可执行文件，但是我们执行shell命令时，
可以直接在终端输入命令的名字就可以执行了，不需要指定shell命令的路径。
shell命令不需要指定对应路径的原因是系统环境变量PATH起作用。
当输入对应的shell命令之后，就会从PATH环境变量对应的路径下查找对应的
shell命令，如果找到对应的命令则执行，如果没有找到对应的命令，则会报
命令没有发现的错误。

思考：查看grep命令的文件的信息？
    1> 查看grep命令对应路径
    linux@ubuntu:~$ sudo find /usr/bin -name grep
    /usr/bin/grep


    2> file命令查看gerp可执行程序的文件的信息
    linux@ubuntu:~$ file /usr/bin/grep
    /usr/bin/grep: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=f4564437ed282494b69a191fcb07a56235cce8ff, for GNU/Linux
3.2.0, stripped

2. 打印系统提供的环境变量
    env  ---> 可以获取系统所有的环境变量的值
    echo  ${环境变量名}

    SHELL=/bin/bash
    PWD=/home/linux
    HOME=/home/linux
    USERNAME=linux
    USER=linux

PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/gam
es:/usr/local/games:/snap/bin

3. 分析PATH环境变量的作用
    PATH是系统默认的shell命令对应的环境变量，
    当在终端输入某个shell命令时，就会从PATH环境变量对应的路径
    下查找对应的命令，如果找到命令则执行，如果没有找到命令，
```

```
    则报命令没有发现的错误。

4. PATH环境变量的格式
    PATH=路径1:路径2:路径3:路径4:....
```

3.1 配置PATH环境变量，将自己的可执行程序配置到PATH环境变量中

3.1.1 方式1：将自己的可执行程序拷贝到PATH对应的路径下

```
    sudo cp hello  /usr/bin
    此种用法一般不使用，如果你进行以上操作之后，测试完成之后要删除。
    sudo rm /usr/bin/hello
```

3.1.2 方式2：将自己的可执行程序的路径追加到PATH环境变量中

```
在终端中执行以下命令：
    export  PATH=$PATH:自己可执行程序的路径，使用绝对路径
        |    |     |  |        |---> 添加自己的可执行程序的路径，使用绝对路径
        |    |     |  |----> 路径的分隔符
        |    |     |----> 将PATH变量之前的值展开(路径的拼接)
        |    |---> 给PATH变量赋值
        |----> 导出环境变量
特点：
    只在当前终端有效，其他终端无效，终端重启也会失效。
```

3.1.3 方式3：将自己的可执行程序的路径追加到PATH环境变量中

```
修改/etc/bash.bashrc配置文件
1. 打开 sudo vi /etc/bash.bashrc文件

2. 在此文件的最后一行添加以下内容：
    export  PATH=$PATH:自己可执行程序的路径，使用绝对路径

3. 使配置立即生效
    source /etc/bash.bashrc

4. 特点：
    对所有的用户都有效。
```

3.1.4 方式4：将自己的可执行程序的路径追加到PATH环境变量中

```
修改/etc/profile配置文件
1. 打开 sudo vi /etc/profile文件

2. 在此文件的最后一行添加以下内容：
    export  PATH=$PATH:自己可执行程序的路径，使用绝对路径

3. 使配置立即生效
    source /etc/profile

4. 特点：
    对所有的用户都有效。
```

3.1.5 方式5：将自己的可执行程序的路径追加到PATH环境变量中

```
修改/etc/environment配置文件
1. 打开 sudo vi /etc/environment文件

2. 对此文件中的内容进行修改

PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:添加自己的路径，使用绝对路径"

3. 使配置立即生效
    source /etc/environment

4. 特点：
    对所有的用户都有效。
```

3.1.6 方式6：将自己的可执行程序的路径追加到PATH环境变量中

```
修改~/.bashrc配置文件
1. 打开 vi ~/.bashrc文件

2. 在此文件的最后一行添加以下内容：
    export  PATH=$PATH:自己可执行程序的路径，使用绝对路径

3. 使配置立即生效
    source ~/.bashrc

4. 特点：
    只对当前的用户有效。
```

4、shell脚本文件中的算数运算

```
    shell脚本文件本身不擅长进行算数运算，shell脚本文件更加擅长的是命令的执行，
以及文件的判断和处理。
    由于shell脚本文件中定义的变量的值，本身都是字符串，
要想使用shell完成算数运算，需要借助一定的算数运算符。
    常用的算数运算：
    ((表达式))
    $[表达式]
```

4.1 (())完成算数运算

```
1. 语法格式：
    ((算数表达式1, 算数表达式2, 算数表达式3,...))
    ((ret1=算数表达式1, ret2=算数表达式2, ret3=算数表达式3,...))

    (())之间可以书写多个不同的算数表达式，

2. 获取运算的结果
    ret=$((算数表达式1, 算数表达式2, 算数表达式3,...))
            ----> 最后一个算数表达式的结果被返回
    ((ret1=算数表达式1, ret2=算数表达式2, ret3=算数表达式3,...))
            ----> 可以获得每一个表达式的结果

3. 特点：
    1> (())中可以包含多个不同的算数表达式，最后一个表达式的结果被返回
    2> 算数表达式中如果使用变量时，变量名前可以加$,也可以不加$.
```

```
#!/bin/bash
# your code

echo "--- 1. (()) 可以有多个不同的表达式，最后一个表达式的结果被返回---"
ret=$((100+200, 100 * 200, 100 / 20))
echo "ret = $ret"

echo "--- 2. (())中的所有的表达式都会被运算，可以在内部获取每个表达式的值---"
ret=$((ret1=100*(10 + 20),ret2=105%10, ret3=100 && 200))

echo "ret = $ret"
echo "ret1 = $ret1"
echo "ret2 = $ret2"
echo "ret3 = $ret3"

echo "--- 3. (())中使用变量时，变量名前可以加$,也可以不加$"
read -p "请输入两个整数，进行加法运算 > " a b
sum=$((a + b))
echo "$a + $b = $sum"

read -p "请输入两个整数，进行减法运算 > " a b
sub=$(($a - $b))
echo "$a - $b = $sub"
```

4.2 $[]进行算数运算

```

```

```
1. 语法格式：
    ret=$[算数表达式1, 算数表达式2, 算数表达式3,...]
    ret=$[ret1=算数表达式1, ret2=算数表达式2, ret3=算数表达式3,...]

    $[]之间可以书写多个不同的算数表达式，

2. 获取运算的结果
    ret=$[算数表达式1, 算数表达式2, 算数表达式3,...]
            ----> 最后一个算数表达式的结果被返回
    ret=$[算数表达式1, ret2=算数表达式2, ret3=算数表达式3,...]
            ----> 可以获得每一个表达式的结果

3. 特点：
    1> $[]中可以包含多个不同的算数表达式，最后一个表达式的结果被返回
    2> 算数表达式中如果使用变量时，变量名前可以加$,也可以不加$.
    3> $[]进行算数运算时，必须使用变量接收$[表达式]的返回值
```

```bash
#!/bin/bash
# your code

echo "--- 1. $[] 可以有多个不同的表达式，最后一个表达式的结果被返回---"
ret=$[100+200, 100 * 200, 100 / 20]
echo "ret = $ret"

echo "--- 2. $[]中的所有的表达式都会被运算，可以在内部获取每个表达式的值---"
ret=$[ret1=100*(10 + 20),ret2=105%10, ret3=100 && 200]

echo "ret = $ret"
echo "ret1 = $ret1"
echo "ret2 = $ret2"
echo "ret3 = $ret3"

echo "--- 3. $[]中使用变量时，变量名前可以加$,也可以不加$"
read -p "请输入两个整数，进行加法运算 > " a b
sum=$[a + b]
echo "$a + $b = $sum"

read -p "请输入两个整数，进行减法运算 > " a b
sub=$[$a - $b]
echo "$a - $b = $sub"

echo "--- 4. $[]进行算数运算时，必须使用变量接收返回值---"
# $[100 * 200]         # 执行报错，报命令没有发现的错误
ret=$[100 * 200]         # ok,将结果赋值给变量
```

5、if...else分支语句

5.1 if...else分支语句的语法格式

```
1. 格式1
    if ((表达式))           ---> if ((表达式)) ; then
    then
        shell语句
```

```
    fi

2. 格式2
    if ((表达式))           ---> if ((表达式)) ; then
    then
        shell语句
    else
        shell语句
    fi

3. 格式3:
    if ((表达式1))          ---> if ((表达式1)) ; then
    then
        shell语句
    elif ((表达式2))        ---> elif ((表达式2)) ; then
    then
        shell语句
    elif ((表达式3))        ---> elif ((表达式3)) ; then
    then
        shell语句
    ## 省略很多elif语句
    else
        shell语句
    fi
```

5.2 分支语句的练习题

```
通过终端输入三只小猪的体重，判断最终的小猪的体重。
#!/bin/bash
# your code

read -p "please input three pig weigth > " pig1 pig2 pig3
if ((pig1 <= 5 || pig2 <= 5 || pig3 <= 5))
then
    echo "输入的小猪的体重不合理，请重新执行!"
    exit
fi

if (( pig1 > pig2)) ; then
    if ((pig1 > pig3)) ; then
        echo "pig1 体重最重"
    elif ((pig1 < pig3)) ; then
        echo "pig3 体重最终"
    else
        echo "pig1 和 pig3 体重相等"
    fi
elif (( pig1 < pig2)) ; then
    if (( pig2 > pig3)) ; then
        echo "pig2 体重最重"
    elif ((pig2 < pig3)) ; then
        echo "pig3 体重最终"
    else
        echo "pig2 和 pig3体重相等"
    fi
```

```
else    # pig1==pig2
    echo "pig1 和 pig2 体重相等"
fi
```

```
练习题：
    从终端输入成绩，对成绩分类?
#!/bin/bash
# your code
read -p "请输入成绩 > " score
if ((score > 100 || score < 0)) ; then
    echo "输入成绩不合理，请重新执行"
    exit
fi

if ((score >= 90)) ; then
    echo "A"
elif ((score >= 80)) ; then
    echo "B"
elif ((score >= 70)) ; then
    echo "C"
elif ((score >= 60)) ; then
    echo "D"
else
    echo "E"
fi

练习题：
    从终端输入年份，判断闰年平年？

#!/bin/bash
# your code

read -p "请输入年份 > " year
if ((year < 0)); then
    echo "输入年份错误，请重新执行输入"
    exit
fi

if ((year % 4 == 0 && year % 100 != 0 || year % 400 == 0))
then
    echo "$year年是闰年"
else
    echo "$year年是平年"
fi
```

6、 test命令

6.1 test命令的介绍

```
1. test是一个shell命令，可以使用man test查看命令的帮助手册

2. test命令的功能：检查文件类型和比较值
    可以用于逻辑判断，字符串判断，字符串比较，整数判断
```

```
    文件的比较，文件类型判断，  文件的权限的判断

3. test命令经常和if分支语句配合使用，完成文件类型和数值的比较

4. test的格式
    test  表达式
    [ 表达式 ]     ---> 最终调用的就是test命令

5. test命令和if...else配合使用的语法格式
    if [ 表达式 ]          ----> if test 表达式
    then
        shell语句
    fi

    if [ 表达式 ]          ----> if test 表达式
    then
        shell语句
    else
        shell语句
    fi

    if [ 表达式 ]          ----> if test 表达式
    then
        shell语句
    elif [ 表达式 ]        ----> elif test 表达式
    then
        shell语句

    .........           ............

    else
        shell语句
    fi

6. test中的表达式的格式要求
    1> 如果在表达式中使用变量时，比如在变量名前加$
        推荐使用变量时，最好使用""将变量名括起来。

    2> 表达式的参数，运算符，中括号，前后必须有空格
        test "hello" < "world"      --> OK
        [ "hello" < "world" ]       --> OK

        test "hello"<"world"        --> error
        [ "hello"<"world" ]         --> error
        ["hello" < "world"]         --> error
        test命令本身是一个elf的可执行程序，
        当执行test命令时，后边的数据作为参数传递给test程序中的
        main函数，main函数中有两个参数argc和argv,通过这两个参数
        可以获得执行test程序时传递的参数，因此在表达式中，出现参数，
        运算符，中括号前后必须添加空格。

    3> 在test命令中部分运算符需要进行转义
        \> \<  \>=  \<=   \( \)  ....
```

6.2 test命令对字符串的判断

```
字符串对象（一定要注意在进行字符串的判断的时候都需要加上""，"$a" "hello"）
    -z 判断字符串是否为空(零) ----> 为空返回真，非空返回假
    -n 判断字符串是否为非空(零)  ---> 非空为真，空为假
    =或== 都是用来判读字符串是否相等    ---> 成立返回真，不成立返回假
    != 不等于    ---> 成立返回真，不成立返回假
    \> 大于 （防止误认为重定向）    ---> 成立返回真，不成立返回假
    \< 小于      ---> 成立返回真，不成立返回假
    \>= 大于等于
    \<= 小于等于

表达式的格式：
[ -z STRING ]   <====>  test -z STRING

[ STRING1 = STRING2]   <====> test STRING1 = STRING2
```

6.3 test命令进行逻辑运算

```
-a  ---> 逻辑与运算(&&)
-o  ---> 逻辑或运算(||)
!   ---> 逻辑非运算(!)
```

```
练习题：从终端输入两个字符串，判断两个字符串的大小
#!/bin/bash
# your code
read -p "请输入两个字符串 > " s1 s2
# 如果在[]中两个表达式进行逻辑运算使用-a或者-o
if [ -n "$s1" -a -n "$s2" ] ; then
    echo "两个字符串都是非空"
else
    echo "两个字符串至少有一个为空"
    exit
fi
# 两个[]进行逻辑判断，使用&&和||
if [ -z "$s1" ] || [ -z "$s2" ] ; then
    echo "两个字符串至少有一个为空"
    exit
else
    echo "两个字符串都是非空"
fi

if [ "$s1" \> "$s2" ] ; then
    echo "$s1 > $s2"
elif [ "$s1" \< "$s2" ] ; then
    echo "$s1 < $s2"
else
    echo "$s1 == $s2"
fi
```

6.4 整数的比较

```
    -eq : 相等
    -ne : 不相等
    -gt : 大于
    -ge : 大于等于
    -lt : 小于
    -le : 小于等于

表达式的格式：
    [ INTEGER1 -eq INTEGER2 ]  <=等价于=> test INTEGER1 -eq INTEGER2
```

```
练习题：
    成绩的分类，
    根据薪资选择不同的交通工具
    小猪体重
```

6.5 文件的比较

```
filename1 -nt filename2 : 判断filename1文件比filename2文件的时间新
filename1 -ot filename2 : 判断filename1文件比filename2文件的时间旧
filename1 -ef filename2 : 判断filename1文件比filename2文件的inode是否一致

表达式的格式：
    [ filename1 -nt filename2 ] <====> test filename1 -nt filename2
```

```
案例：
    执行脚本文件时，传递两个普通文件的名字，判断文件的时间戳。
    #!/bin/bash
# your code

if [ $# -ne 2 ] ; then
    echo "执行脚本文件时传递的参数不合理，请重新执行"
    echo "usage : ./$0 fileName1 fileName2"
    exit
fi

# 判断文件是否存在，并且是否为普通文件
if [ -f "$1" -a -f "$2" ] ; then
    if [ "$1" -nt "$2" ] ; then
        echo "$1文件的时间戳比$2新"
    else
        echo "$1文件的时间戳比$2旧"
    fi
else
    echo "文件不存在，或者不是普通文件"
fi
```

6.6 文件类型的判断

```
-b filename 判断文件是否存在，是否是块设备
-c filename 判断文件是否存在，是否是字符设备
-d filename 判断文件是否存在，是否是目录
-f filename 判断文件是否存在，是否是普通文件
-p filename 判断文件是否存在，是否是管道文件
-L filename 判断文件是否存在，是否是链接文件 （经测试链接文件也是普通文件）
-S filename 判断文件是否存在，是否是套接字文件
-e filename 判断文件是否存在
-s filename 判断文件是否存在，判断文件是否为空

表达式的格式：
    [ -b  FILENAME ]  <===> test -b FILENAME
```

```
练习题：
    执行脚本文件名时，传递一个文件名，判断文件是目录还是普通的文件。
```

6.7 文件的权限的判断

```
-r filename 判断文件是否存在，是否有可读权限
-w filename 判断文件是否存在，是否有可写权限
-x filename 判断文件是否存在，是否有可执行权限

表达式的格式：
    [ -r  FILENAME ] <====> test -r FILENAME
```

```
练习题：
    执行脚本文件时，传递一个脚本文件的名字，判断此文件是否为脚本文件，
    如果为脚本文件判断此脚本文件是否具有可执行的权限，如果没有可执行的权限
    给脚本文件添加对应的可执行权限。
    ./***.sh  01fisrst.sh
    prfi=`echo $1 | cut -d '.' -f 2 `

    name=`file 01array.sh | cut -d " " -f 3`

```

7、作业

```
1. C试卷-》第6套,前20题
2. shell编程中的if分支语句，和test的结合
```

8、明天授课内容

```
1. case...in
2. select...in
3. for循环
4. while循环
5. break/continue
6. 函数

1. 宏定义的总结：带返回值的宏函数
2. goto,动态内存分配(malloc/free)
```