1. shell脚本

1.1 shell概念

shell是一种脚本语言

脚本：本质来讲就是一个文件，文件里面存放的时特定格式的指令，系统可以使用脚本解析器 翻译或者解析指令从而执行，shell即使应用程序又是一种脚本语言（应用程序解析脚本语言。）

注意：本质上讲就是将之前学过的csa初级里面的命令写成文件的形式，所有，如果在命令行中都无法实现需求的话，那么shell就很难写好。

1.2 shell命令解析器

系统提供的shell命令解释器：sh ash bash

目前使用最多shell命令解释器是bash

```
# 查看系统默认的解释器类型
[root@localhost ~]# echo $SHELL
/bin/bash
```

1.3 shell的基本元素

使用shell最主要的原因就是为了减少重复手动编写命令的麻烦，所以shell将一系列Linux命令放入一个文件中。

shell脚本面向的用户是会使用shell和linux的运维工作者。所以说，shell最基本的要求是按照你的手册能够正常执行！

1.3.1 shell脚本格式

```
[root@localhost git-test]# touch showDate.sh
[root@localhost git-test]# vim showDate.sh
[root@localhost git-test]# cat showDate.sh
#!/bin/bash
# date: 2022-07-30
# author: xxr
# show date
date

[root@localhost git-test]# chmod u+x ./showDate.sh
[root@localhost git-test]# ls
hello.sh  LICENSE  README.en.md  README.md  showDate.sh
[root@localhost git-test]# ./showDate.sh
Sat Jul 30 10:02:57 CST 2022
```

- shell脚本的后缀一般是.sh
- shell中脚本中第一行#!/bin/bash，其中#！为shell脚本的起始符，作用就是为了告诉Linux系统这个文件想要执行需要使用的解释器的路径，其中/bin/bash就是bash解释器的路径。
- # date: 2022-07-30,# author: xxr ， #show date都是注释内容， （#可以是整行，也可以放在某一行的后面）

```
[root@localhost git-test]# cat showDate.sh
#!/bin/bash
# date: 2022-07-30
# author: xxr
date   # show date
```

- date: 就是具体执行的命令，每一行编写一条命令。

编写一个shell脚本，创建用户student04，并且设置默认密码为1，并且完成后打印用户的信息（id）

```
[root@localhost shell_0730]# vim addStudent04.sh
[root@localhost shell_0730]# chmod u+x addStudent04.sh
[root@localhost shell_0730]# ./addStudent04.sh
uid=1005(student04) gid=1006(student04) groups=1006(student04)
[root@localhost shell_0730]# cat addStudent04.sh
#!/bin/bash
# date: 2022-07-30
# auther: xxr

useradd student04    &> /dev/null
echo "1" | passwd --stdin student04  &> /dev/null
id student04
```

补充：

- 以非交互的形式来设置密码
    - echo "密码" | passwd --stdin 用户名
- 如果不想让用户看到输出的信息，可以使用>输出重定向到其他地方，例如">/dev/null"

1.3.2 执行shell脚本

- 给脚本赋予执行权限：chmod u+x 脚本路径
- ./脚本路径
- 脚本的绝对路径

目的就是通过写明脚本路径的方式，告诉系统去哪执行脚本。

```
[root@localhost shell_0730]# /root/shell/git-
test/shellStudy/shell_0730/showDate.sh
Sat Jul 30 10:48:51 CST 2022
[root@localhost shell_0730]# ./showDate.sh
Sat Jul 30 10:48:57 CST 2022
```

2. 变量

变量主要可以分为三种：

- 本地变量: 随着shell的执行而产生，也随之shell脚本的结束而消亡。
- 环境变量：适用于所有由登录进程所产生的子进程（用户登录成功后，就可以适用环境变量）
- 位置变量: 用于向shell脚本传递参数，是只读。

2.1 变量的赋值和取值

- 赋值：
  - 变量名=值
    - 等号两边不可以有空格
    - 如果值里面有空格，必须用双引号括起来
    - 变量名只能是大小写字母，数字和下划线等符号，且不能以数字开头
- 清除变量值
  - unset 变量名
- 取值
  - $变量名
  - ${变量名}: 如果变量名和其他字符紧挨着，可以用花括号表名变量部分。

在命令行中编写代码：声明变量studentName值为student04，然后使用变量来删除该用户，主要使用-r选项，完成后截图

```
[root@localhost shell_0730]# studentName=student04
[root@localhost shell_0730]# echo $studentName
student04
[root@localhost shell_0730]# userdel -r $studentName
[root@localhost shell_0730]# id student04
id: ‘student04’: no such user
```

2.2 环境变量

环境变量不仅用在shell编程方面，而且在Linux系统管理方面也起着特别重要的作用

2.2.1 定义环境变量

```
# 定义环境变量的基本格式
# 注意： 环境变量名一般大写
环境变量名=值

# 声明环境变量
export  环境变量名
```

使用env可以查看所有的环境变量

2.2.2 常见的环境变量

- PWD： 记录当前的目录路径
- OLDPWD: 记录上一次的工作目录
- PATH: 该环境变量极为重要，用于帮助shell找到用户所输入的命令，用户在终端中输入的命令，实际上都是保存在系统中各种目录的下的可执行文件，PATH里面就是记录了这些可执行文件所在的路径，多个路径之间用:隔开，如果突然发现，命令失效，就去查看PATH环境变量有无问题
- HOME: 记录当前用过户的家目录
- SHELL：系统默认shell
- PS1和PS2： 命令提示符，其中PS1为一级shell命令提示符，PS2为二级命令提示符

定义PS1的命令提示符，加入时间 \t，完成后截图

```
# ps1常用的值
\d：代表日期，格式为weekday month date，例如：“Mon Aug 1”
\H ：完整的主机名称
\h ：仅取主机的第一个名字
\t ：显示时间为24小时格式，如：HH：MM：SS
\T ：显示时间为12小时格式
\A ：显示时间为24小时格式：HH：MM
\u ：当前用户的账号名称
\v ：BASH的版本信息
\w ：完整的工作目录名称
\W ：利用basename取得工作目录名称，所以只会列出最后一个目录
\# ：下达的第几个命令
\\$ ：提示字符，如果是root时，提示符为：# ，普通用户则为：$
\n：表示换行
```

如何修改终端提示符的颜色：https://blog.csdn.net/ikkyphoenix/article/details/119282386

如果想要环境变量永久生效的话，需要将它写入环境变量的配置文件（分为2种）

- 全局生效：/etc/profile和/etc/bashrc
- 用户生效：.bash_profile 和.bashrc以及.bash_logout（用户登出的时候加载）

1. 当用户登录时，shell会自动执行/etc/profile和/etc/bashrc（包括当前用户下的环境配置文件）
2. .bash_logout在用户注销时执行加载

注意: 执行加载配置文件时，要使用source，而不是使用bash来执行脚本

区别如下:

- source命令执行脚本代表在当前环境下执行命令，并使变量生效，直接可以在当前终端使用，如果想在shell脚本种加载外部文件，也可以使用该方法
- 而使用bash或者./shell脚本方法，是在当前终端中启动一个子shell来执行命令，当前环境中没有生效。

2.3 位置参数

位置参数是一种特殊的shell变量，用于从命令行向shell脚本传递参数。

- $0为脚本名
- $1为第一个参数
- $2为第二个参数

  ……
- 从第${10}开始。参数数字需要用花括号括起来
- $*和$@表示所有输入的参数
- $# ： 传入脚本的参数个数
- $? : 获取命令退出的状态，0代表命令运行没有错误，非0代表有错误

```
[root@localhost shell_0730]# ./test_.sh  hello world tom
hello
world
-------
hello world tom
3
filename: ./test_.sh
[root@localhost shell_0730]# cat test_.sh
#!/bin/bash
# date: 2022-07-30
# author: xxr

echo "$1"
echo "$2"
echo "-------"
echo "$*"
echo "$#"
echo "filename: $0"
```

利用位置参数，创建用户student04，并从外部传入密码为1,完成后截图

```
[root@localhost shell_0730]# cat ./createStudent.sh
#!/bin/bash
# date: 2022-07-30
# author: xxr

useradd $1
echo "$2" | passwd --stdin $1
id $1
[root@localhost shell_0730]# ./createStudent.sh student04 1
Changing password for user student04.
passwd: all authentication tokens updated successfully.
uid=1005(student04) gid=1006(student04) groups=1006(student04)
```

2.4 外部传参

read命令

- read命令被用来从标准输入中读取单行数据，该命令可以用来读取键盘输入的内容，当使用重定向符时，可以读取文件中的一行数据，搭配循环用，可以遍历文件中的所有内容。

常用选项

- -p 后面加入提示信息，即在输入前提示的内容
- -s 安静模式，输入的字符不会再屏幕上显示，特别再输入密码时可以使用。
- -t 后面加秒数，定义输入字符的等待时间。
- -a 选项后面需要加上一个变量，然后改变量就会被认为是一个数组，然后给其赋值，默认分隔符是空格。

-a 选项的使用

```
[root@localhost shell_0730]# cat testRead.sh
#!/bin/bash
# date: 2022-07-30
# author: xxr

read -p "please input hello "  -a  world
echo "${world[0]}"
echo "${world[*]}"
[root@localhost shell_0730]# ./testRead.sh
please input hello h e l l o
h
h e l l o
```

```
[root@localhost shell_0730]# cat ./testRead.sh
#!/bin/bash
# date: 2022-07-30
# author: xxr

read -sp "please input hello "  world
echo "$world"
```

题目: 修改主机名脚本，从终端让用户输入想要修改的主机名，修改完成后，提示"successful"

"请输入修改后主机名："

```
[root@xxr shell_0730]# ./changeHost.sh
请输入修改后主机名：localhost
successful
现在主机名为:localhost.localdomain
[root@localhost shell_0730]# cat changeHost.sh
#!/bin/bash
# date: 2022-07-30
# author: xxr
read -p "请输入修改后主机名：" hostname
hostnamectl set-hostname  $hostname
echo "successful"
echo "现在主机名为:$(hostname)"
bash
```

- 读取文件

每次调用read命令都会读取文件中的一行文本，当文件没有可读内容时，read命令将会以非0状态退出

```
[root@localhost shell_0730]# read line < /etc/passwd
[root@localhost shell_0730]# echo $line
root:x:0:0:root:/root:/bin/bash
```

利用while循环和输入重定向读取文件中的每一行

```
[root@localhost shell_0730]# cat  testReadFile.sh
#!/bin/bash
# date: 2022-07-30
# author: xxr
while read line
do
    echo $line
done < /etc/passwd
```

在当前脚本路径下，创建文件username.txt，其中里面内容为

user01

user02

user03

然后利用while循环和read命令，读取username.txt中内容，批量创建用户，密码默认和用户名同名,创建完成后并输出用户信息。 完成后截图

```
[root@localhost shell_0730]# cat createUsers.sh
#!/bin/bash
# date: 2022-7-30
# author: xxr
while read a
do
    useradd $a
    echo $a | passwd --stdin $a
    id $a

done  < ./username.txt
```

2.5 数组

数组可以存放多个值。bash shell只支持一维数组（不支持多维数组），初始化时不要定义数据的大小和类型。

与大部分编程语言类型，数组元素的下标时从0开始。

- shell数组用括号来表示，元素用空格分开，语法如下：

```
array_name=(value1 value2 ... valuen)
```

- 使用数字下标来定义数组

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

- 读取数组的方法

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

- 使用@或*可以获取数组中的所有元素

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

补充: 将脚本所在的路径加入到环境变量PATH中的目录,那么在任意地方都可以执行该脚本

题目1：

编写一个命令vish编写文件时，会自动在文件内部生成如下内容

#!/bin/bash

Sat Jul 30 15:38:20 CST 2022

xxr

```
[root@xxr ~]# cat /usr/bin/vish
#!/bin/bash
# date: 2022-07-30
# auther: xxr
if [ -f $1 ];then
    vim ./$1
else
touch $1
echo "#!/bin/bash" >> ./$1
echo "# date: $(date)" >> ./$1
echo "# author: xxr" >> ./$1
vim ./$1
fi
```

题目2：

创建数组delusers，将user01,user02,user03设置为数组的值，然后利用数组中的值，删除用户，注意加上-r

3. 引用

常见引用符号和作用

符号	作用
""	引用除了美元符，反引号和反斜线之外的所有字符
''	引用所有字符
`` ` | 反引号，shell会将反引号内的内容解释为系统命令，\ `linux命令`=$(linux命令)	
\	(将一个字符从一种意义转为另一种意义)将有特殊含义的字符，转义成原本含义。然后在echo,sed，awk等命令中，转义符加上某些字母也可以表示特殊含义，\n换行符，\t制表符等。

13点45继续，完成后截图

```
[root@localhost ~]# a=`hostname  | awk -F . '{print $1}'`
[root@localhost ~]# echo $a
localhost
```

题目： 获取nvme0n1磁盘的大小，并将值赋给变量n1size，5分钟，13点52继续

```
[root@localhost ~]# n1size=`lsblk | grep "^nvme0n1" | awk '{print($4)}' | awk -F "G" '{print($1)}'`
[root@localhost ~]# echo $n1size
20
```

3.1 echo命令详解

echo命令用于将输入的字符串输出到终端屏幕上，默认输出的字符一空白字符隔开，并且最后会加上换行符。

选项	描述
-n	输出的结构不进行换行
-e	允许对转义符后面的字符进行解释。

1. 特殊字符的使用

特殊字符	描述
\a	发出警告声
\e	用于颜色输出上面，效果相当于\033
\n	换行符
\t	制表符

3.1.1 自定义颜色

echo使用-e选项搭配特殊字符\e或者\033可以使输出的内容加入颜色

```
echo -e  "\033[背景色数值;文字颜色数值m 展示的文字 \033[0m"
或者
echo -e  "\e[背景色数值;文字颜色数值m 展示的文字 \e[0m"
```

注意：背景数值是从40开始到47结束，文件颜色数值从30开始到37结束。

- 背景色和文字颜色分别代表：
  - （40，30）黑色，
  - （41，31）红色，
  - （42，32）绿色，
  - （43，33）黄色，
  - （44，34）蓝色，
  - （45，35）紫色，
  - （46，36）天蓝色，
  - （47，37）白色

下课休息一下，然后自己输出一下echo的内容，测试一下不同的颜色。完成后截图

3.2 算术运算

原生bash不支持简单数学运算，但是可以利用其他命令来完成，例如awk和expr,expr最常用。

expr用于对表达式的求值操作。

运算符	说明
+	加法
-	减法
/	除法，默认只能取整
%	取余
*	乘法

注意: 条件表达式的两边要加上空格

```
[root@localhost ~]# expr  3/2
3/2
[root@localhost ~]# expr  3 / 2
1

# 补充：如果想要进行浮点数运算，可以使用awk命令
[root@localhost ~]# awk "BEGIN{print(3/2)}"
1.5
```

补充： 如果想要完成变量的自增或自减，可以使用let命令，参考代码如下

```
[root@localhost shell_0801]# cat testUntil.sh
#!/bin/bash
# date: Mon Aug  1 14:08:42 CST 2022
# author: xxr
i=0
until [ $i -eq "5" ]
do
    # 写成 let i++ 也是可以的
    let i=i+1
    echo "$i"
done
```

4. 命令退出状态

在linux中，命令执行完毕后，系统会返回一个退出状态，而退出状态用整数值来表示

状态值	含义
0	运行成功
1~125	运行失败
126	找到运行的命令，但无法执行
127	未找到命令
> 128	命令被系统强制终止

在判断和循环语句中，可以使用exit命令来控制表达式的流程。

5. 判断

1. 单支语句

```
if 条件测试;then
命令1
命令2
fi
```

请输入数字1

```
[root@localhost shell_0731]# cat testif.sh
#!/bin/bash
# date: Sun Jul 31 15:02:58 CST 2022
# author: xxr
read -p "请输入数字1 " num
if [ $num -eq 1 ];then
echo -e "\e[32msuccessfully!\e[0m"
fi
```

5.1 条件测试

[测试内容]

注意： "["是启动测试的命令，要求与"]" 配对，因为它本身也是命令，命令和命令之间需要用空格隔开，所以空格是必不可少的。

当前条件测试内容为真时，就会执行then后面的语句

5.1.1 算术比较

```
[root@localhost shell_0731]# [ 1 -eq 0 ]
[root@localhost shell_0731]# echo $?
1
[root@localhost shell_0731]# [ 0 -eq 0 ]
[root@localhost shell_0731]# echo $?
0
```

- -eq 等于
- -ne 不等于
- -gt 大于
- -ge 大于等于
- -le 小于等于
- -lt 小于

5.1.2 文件相关的测试

- [-f 文件路径]: 如果给定的时正常且存在的普通文件则返回真
- [-d 文件路径] 如果给定的文件时一个目录，则返回真
- [-e 文件路径] 如果文件存在则返回真
- [-c 文件路径] 如果给定的文件是一个字符设备文件，则返回真
- [-b 文件路径] 如果给定的文件是一个块设备文件，则返回真
- [-L 文件路径] 如果给定的文件是一个链接文件，则返回真
- [-x 文件路径]: 如果给定的文件有执行权限，则返回真
- [-w 文件路径]: 如果给定的文件有写权限，则返回真
- [-r 文件路径]: 如果给定的文件有读权限，则返回真

5.1.3 字符串的比较

进行字符串比较时，最好使用双括号号，因为有时候单个中括号会产生错误。注意：双中括号只能bash解释器使用，是属于bash扩展功能。

1. 测试2个字符串是否相同

```
[root@localhost shell_0731]# [[ "a" == "B" ]]
[root@localhost shell_0731]# echo $?
1
[root@localhost shell_0731]# [[ "a" == "a" ]]
[ro
```

2. 测试两个字符串是否不同

```
[root@localhost shell_0731]# [[ "a" != "a" ]]
[root@localhost shell_0731]# echo $?
1
[root@localhost shell_0731]# [[ "a" != "B" ]]
[root@localhost shell_0731]# echo $?
0
```

3. 字符比较大小

字符串是根据ASCII的值进行比较，

```
[root@localhost shell_0731]# [[ "a" > "b" ]]
[root@localhost shell_0731]# echo $?
1
[root@localhost shell_0731]# [[ "a" < "b" ]]
[root@localhost shell_0731]# echo $?
0
```

4. [[-z $变量名]]： 如果变量值为空，而返回真
5. [[-n $变量名]]： 如果变量值不为空，而返回真

补充说明： 使用 &&（与） 和 ||（或） 能够将多个条件组合起来

- 如何根据命令的退出值来进行判断，命令是否运行成功。

```
# 方法一，利用$?来获取上一个命令的退出值，然后判断命令是否执行成功
dfsfdls &> /dev/null
if [ $? -eq 0 ];then
        echo -e "\e[32m命令执行成功\e[0m"

else
        echo -e "\e[31m命令执行失败\e[0m"

fi


# 方法二，直接根据命令运行的结果进行判断，成功的话则为真
#!/bin/bash
# date: Sun Jul 31 15:54:16 CST 2022
# author: xxr
if lsdsfsdf &> /dev/null;then
        echo -e "\e[32m命令执行成功\e[0m"

else
        echo -e "\e[31m命令执行失败\e[0m"

fi
```

题目1： 从终端输入一个文件，然后脚本会判断，当前路径下该文件是否存在，如果存在并且有内容的话，那么不做任何操作，**提示文件存在且不为空**，如果不存在则创建该文件，**提示文件已创建**，如果存在且没有内容，**提示该文件为空**

```
#!/bin/bash
# date: Mon Aug  1 09:35:54 CST 2022
# author: xxr
read -p "输入一个文件: " filename
# 首先判断文件是否存在，不存在则创建文件
# 在测试条件前加！代表结果取反
if [[ ! -e $filename ]];then
    touch $filename
```

```
    echo -e "\e[32m文件已创建\e[0m"
else
# 如果文件存在，那么判断内容是否为空
    contxt=`cat $filename`
    if [[ -n $contxt ]];then
        echo -e "\e[32m文件存在且不为空\e[0m"
    else
        echo -e "\e[32m文件存在且为空\e[0m"
    fi

fi
```

2分钟，9点55继续

题目2： 从终端输入用户名（请使用read），判断该用户是否存在，如果不存在则创建该用户，密码默认和用户名一致，如果存在输出提示"某某用户已存在

```
#!/bin/bash
# date: Mon Aug  1 09:57:20 CST 2022
# author: xxr
read -p "输入用户名: " username

# 方法一：根据/etc/passwd中能否获取到用户名来判断。
contxt=`grep "^$username" /etc/passwd`
if [[ -n $contxt ]];then
    echo -e "\e[32m${username}用户已存在\e[0m"
else
    useradd $username
    echo "$username" | passwd --stdin $username &> /dev/null
    echo -e "\e[32m${username}用户已创建，且默认密码为用户名\e[0m"
fi


# 方法二 利用操作的退出状态值进行判断
#!/bin/bash
# date: Mon Aug  1 09:57:20 CST 2022
# author: xxr
read -p "输入用户名: " username

if id $username &> /dev/null;then
    echo -e "\e[32m${username}用户已存在\e[0m"
else
    useradd $username
    echo "$username" | passwd --stdin $username &> /dev/null
    echo -e "\e[32m${username}用户已创建，且默认密码为用户名\e[0m"
fi

#方法三，就是方法二的另一种写法
#!/bin/bash
# date: Mon Aug  1 09:57:20 CST 2022
# author: xxr
read -p "输入用户名: " username

id $username &> /dev/null
```

```
if [ $? -eq 0 ];then
        echo -e "\e[32m${username}用户已存在\e[0m"
else
        useradd $username
        echo "$username" | passwd --stdin $username &> /dev/null
        echo -e "\e[32m${username}用户已创建，且默认密码为用户名\e[0m"
fi
```

5.2 exit命令

在判断和循环中可以使用exit来设定退出状态的值，如果不设置的话，默认以最后一个命令的退出状态值为准。注意：不要随便乱用退出值，否则会产生误解。

```
[root@localhost ~]# (if [ -e /1.txt ];then echo  -e "\e[32mfile /1.txt\e[0m";
else echo -e "\e[31mno file\e[0m";exit 1;fi)
no file
[root@localhost ~]# echo $?
1
```

补充说明： 在当前终端中如果将命令用（）括起来，代表的是在当前终端的子shell中运行，不会在当前终端运行。

5.3 if/else结构

```
# 格式1:
if [ 条件1 ];then
命令1
else
命令2
fi

# 格式2:
if [ 条件1 ];then
    命令1
elif[ 条件2 ];then
    命令2
else
    命令3
fi
```

题目：让用户输入数字

如果输入的是1，那么输出hello

如果输入的是2，那么输出world

其他内容，一律显示hello world!

```
#!/bin/bash
# date: Mon Aug  1 10:57:28 CST 2022
# author: xxr
read -p "输入数字 " num
if [ $num  -eq 1 ];then
    echo -e "\e[32mhello\e[0m"
elif [ $num -eq 2 ];then
       echo -e "\e[32mworld\e[0m"
else
    echo -e "\e[32mhello world\e[0m"
fi
```

优化上方代码，判断用户输入的是否为数字，如果不是数字，则提示用户重新输入数字，grep正则之类的[0-9]

```
#!/bin/bash
# date: Mon Aug  1 10:57:28 CST 2022
# author: xxr
read -p "输入数字 " num
# 将用户输入的字符串提取非数字部分
newNum=`echo $num | grep -o "[^0-9]"`
# 只要提取的内容不为空，即含有非数字的部分，那么肯定输入的不是纯数字
if [[ -n $newNum ]];then
        echo "重新输入数字 "
else
        if [ $num  -eq 1 ];then
                echo -e "\e[32mhello\e[0m"
        elif [ $num -eq 2 ];then
                echo -e "\e[32mworld\e[0m"
        else
                echo -e "\e[32mhello world\e[0m"
        fi
fi
```

6. case结构

如果判断条件的是一个精确的值，那么推荐使用case结构，如果判断的条件是一个范围，推荐使用if。

```
case 变量 in
条件1 )
    执行命令1
    ;;
条件2 )
   执行命令2
   ;;
......
 * )
   无匹配后执执行的命令
esac
```

```
# 示例
[root@localhost shell_0801]# cat testcase.sh
#!/bin/bash
# date: Mon Aug  1 13:34:54 CST 2022
# author: xxr
read -p "input a number: " num
case $num in
    1)
        echo "hello"
        ;;
    2)
        echo "world"
        ;;
    *)
        echo "hello world"
esac
```

题目：编写代码，可以实现对防火墙的控制，输入数字1，开启防护墙，输入数字2，关闭防护墙，如果输入其他，则提示“ 请按要求输入数字”

下方可以通过输入数字来实现对防火墙的控制

1----开启防火墙

2----关闭防火墙

请输入数字 1

完成后截图

```
[root@localhost shell_0801]# cat ./casefirewalld.sh
#!/bin/bash
# date: Mon Aug  1 13:49:27 CST 2022
# author: xxr
echo "下方可以通过输入数字来实现对防火墙的控制"
echo "1----开启防火墙"
echo "2----关闭防火墙"
read -p "请输入数字 " num
case $num in
    1 )
        systemctl start firewalld
        echo -e  "\e[32mstart!\e[0m"
        ;;
    2)
        systemctl stop firewalld
        echo -e  "\e[32mstop!\e[0m"
        ;;
    *)
        echo -e "\e[31m请按要求输入数字\e[0m"
esac
```

7. for

```
# for循环语句
for 变量名 in [ 取值列表 ]
do
    循环体
done


# 简单的计数循环
for  变量名  in {1..5}
do
    循环体
done
```

示例：

```
[root@localhost shell_0801]# cat testfor.sh
#!/bin/bash
# date: Mon Aug  1 13:58:11 CST 2022
# author: xxr

#  for循环语句
for i in 1 2 3 4 5
do
    echo $i
done

echo "=============="

arrays=(a b c d)
for i in ${arrays[*]}
do
    echo "$i"
done

echo "============="
for i in {1..5}
do
    echo "$i"
done
```

8. while

```
# 当条件测试为真，执行循环
while  条件测试
do
    循环体
done
```

补充：条件测试内容，参考上方if后面的条件测试内容

9. until

```
# 符合条件测试时，退出循环。
until  条件测试
do
  循环体
done
```

```
[root@localhost shell_0801]# cat testUntil.sh
#!/bin/bash
# date: Mon Aug  1 14:08:42 CST 2022
# author: xxr
i=0
until [ $i -eq "5" ]
do
    # 写成 let i++ 也是可以的
    let i=i+1
    echo "$i"
done
```

补充：使用let命令完成变量的自增

10. break,continue

- break命令允许跳出所有的循环（终止执行后面的所有循环）
- continue命令和break命令类似，只有一点区别，它不会跳出所有循环，仅仅跳出当前循环。
- 使用exit status值也可以退出循环体

```
[root@localhost shell_0801]# cat ./testcase.sh
#!/bin/bash
# date: Mon Aug  1 13:34:54 CST 2022
# author: xxr

#while read -p "input a number: " num
while true
do
read -p "input a number: " num
case $num in
    0)
        echo "Bye"
        break
        ;;
    1)
        echo "hello"
        ;;
    2)
        echo "world"
        ;;
    *)
        echo "hello world"
esac
done
```

优化一下防火墙的脚本，利用循环可以让用户一直停留在该选择页面，且提供一个退出选项，完成后截图

```
[root@localhost shell_0801]# cat casefirewalld.sh
#!/bin/bash
# date: Mon Aug  1 13:49:27 CST 2022
# author: xxr
while true
do
echo -e "下方可以通过输入数字来实现对防火墙的控制"
echo "1----开启防火墙"
echo "2----关闭防火墙"
read -p "请输入数字 " num
case $num in
    0 )
        echo "Bye"
        break
        ;;
    1 )
        systemctl start firewalld
        echo -e  "\e[32mstart!\e[0m"
        ;;
    2)
        systemctl stop firewalld
        echo -e  "\e[32mstop!\e[0m"
        ;;
    *)
        echo -e "\e[31m请按要求输入数字\e[0m"
esac
done
```

11. 函数

```
[ function ] funname ()
{
  命令1
  命令2
  [return 返回状态值]
}
```

- 可以使用function 函数名（），也可以直接函数名（）声明函数
- 函数的退出返回值可以使用return来设定，效果等同exit 退出状态值，如果不写，默认以运行的结果作为返回值。

示例：

```
#!/bin/bash
# date: Mon Aug  1 14:55:11 CST 2022
# author: xxr
function echoHello(){
  echo "hello"
}

echoHello
```

11.1 函数参数

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代表第一个值，$2 代表第二个参数值。

```
[root@localhost shell_0801]# cat  testfun.sh
#!/bin/bash
# date: Mon Aug  1 14:55:11 CST 2022
# author: xxr
function echoHello(){
  echo "hello"
  echo  "$1"
  echo  "$2"
  echo  "传入参数的个数: $#"
  echo "传入的所有参数: $*"
}

echoHello a b c d
```