Shell中的色彩处理

echo -n

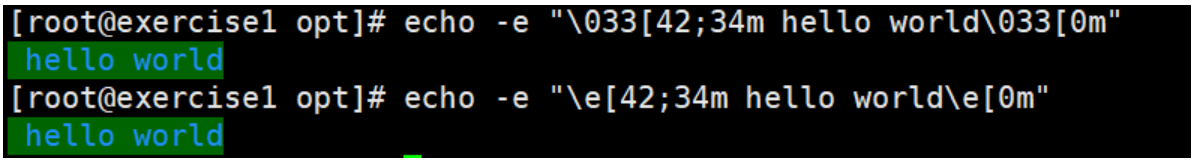
shell脚本中echo显示内容带颜色显示

echo显示带颜色，需要使用参数-e

格式 1： echo -e "\033[背景颜色;文字颜色m 要输出的字符 \033[0m"

格式 2：echo -e "\e[背景颜色;文字颜色m 要输出的字符\e[0m"

```
例子：绿底蓝字
[root@exercise1 opt]# echo -e "\033[42;34m hello world\033[0m"
[root@exercise1 opt]# echo -e "\e[42;34m hello world\e[0m"
```



注:其中42的位置代表底色，34的位置代表的是字的颜色，0m是清除所有格式

1、字背景颜色和文字颜色之间是英文的分号";"

2、文字颜色后面有个m

3、字符串前后可以没有空格，如果有的话，输出也是同样有空格

4、echo显示带颜色，需要使用参数-e, -e允许对下面列出的加反斜线转义的字符进行解释.

控制选项:

\033[0m 关闭所有属性

\033[1m 设置高亮度，加粗

\033[5m 闪烁

```
[root@exercise1 opt]# echo -e "\e[42;34m hello world\e[5m" #执行后，发现后期所有输出都闪烁状态，如何关闭？

[root@exercise1 opt]# echo -e "\e[42;34m hello world\e[0m" #可以使用\033[0m关闭所有属性
```

常见shell输入带颜色文字

3x 代表字的颜色，4x 代表背景色

```
echo -e "\033[30m 黑色字 \033[0m"
echo -e "\033[31m 红色字 \033[0m"
echo -e "\033[32m 绿色字 \033[0m"
echo -e "\033[33m 黄色字 \033[0m"
echo -e "\033[34m 蓝色字 \033[0m"
echo -e "\033[35m 紫色字 \033[0m"
echo -e "\033[36m 天蓝字 \033[0m"
echo -e "\033[37m 白色字 \033[0m"
echo -e "\033[40;37m 黑底白字 \033[0m"
echo -e "\033[41;37m 红底白字 \033[0m"
echo -e "\033[42;37m 绿底白字 \033[0m"
echo -e "\033[43;37m 黄底白字 \033[0m"
echo -e "\033[44;37m 蓝底白字 \033[0m"
echo -e "\033[45;37m 紫底白字 \033[0m"
echo -e "\033[46;37m 天蓝底白字 \033[0m"
echo -e "\033[47;30m 白底黑字 \033[0m"
```

扩展

```
echo -e “\033[31m 红色字 \033[0m”
echo -e “\033[34m 黄色字 \033[0m”
echo -e “\033[41;33m 红底黄字 \033[0m”
echo -e “\033[41;37m 红底白字 \033[0m”

echo -e "\033[30m 黑色字 \033[0m"
```

```
echo -e "\033[31m 红色字 \033[0m"
echo -e "\033[32m 绿色字 \033[0m"
echo -e "\033[33m 黄色字 \033[0m"
echo -e "\033[34m 蓝色字 \033[0m"
echo -e "\033[35m 紫色字 \033[0m"
echo -e "\033[36m 天蓝字 \033[0m"
echo -e "\033[37m 白色字 \033[0m"

echo -e "\033[40;37m 黑底白字 \033[0m"
echo -e "\033[41;37m 红底白字 \033[0m"
echo -e "\033[42;37m 绿底白字 \033[0m"
echo -e "\033[43;37m 黄底白字 \033[0m"
echo -e "\033[44;37m 蓝底白字 \033[0m"
echo -e "\033[45;37m 紫底白字 \033[0m"
echo -e "\033[46;37m 天蓝底白字 \033[0m"
echo -e "\033[47;30m 白底黑字 \033[0m"

#控制选项
\33[0m 关闭所有属性
\33[1m 设置高亮度
\33[4m 下划线
\33[5m 闪烁
\33[7m 反显
\33[8m 消隐
\33[30m – \33[37m 设置前景色
\33[40m – \33[47m 设置背景色
\33[nA 光标上移n行
\33[nB 光标下移n行
\33[nC 光标右移n行
\33[nD 光标左移n行
\33[y;xH设置光标位置
\33[2J 清屏
\33[K 清除从光标到行尾的内容
\33[s 保存光标位置
\33[u 恢复光标位置
\33[?25l 隐藏光标
\33[?25h 显示光标
```

```
shell色彩应用实战：

[root@exercise1 opt]# echo -e "\033[31m 部署 vsftp 环境 \033[0m" && yum -y install vsftpd &> /dev/null && systemctl start vsftpd && echo -e "\033[42m vsftp 启动成功 \033[0m" || echo -e "\033[41m 启动失败！ \033[0m"

#安装服务，部署开始时输出红色字体。部署成功并启动服务后输出绿底start字样，失败则输出红底error字样。
```

```
[root@exercise1 opt]# echo -e "\033[42;34m hello world\033[0m"
 hello world
[root@exercise1 opt]# echo -e "\e[42;34m hello world\e[0m"
 hello world
```

跳出循环

在我们使用循环语句进行循环的过程中,有时候需要在未达到循环结束条件时强制跳出循环，

那么Shell给我们提供了两个命令来实现该功能：break和continue

---

break和continue

Break：跳出整个循环

Continue：跳过本次循环，进行下次循环

break 概述：跳出当前整个循环或结束当前循环，在for、while等循环语句中，用于跳出当前所在的循环体，执行循环体之后的语句，后面如果什么也不加，表示跳出当前循环等价于break 1，也可以在后面加数字,假设break 3表示跳出第三层循环

continue概述：忽略本次循环剩余的代码，直接进行下一次循环；在for、while等循环语句中，用于跳出当前所在的循环体，执行循环体之后的语句，如果后面加的数字是1，表示忽略本次条件循环，如果是2的话，忽略下来2次条件的循环

```
例子1：例1：写一个shell菜单，当按数字键4时退出，否则一直循环显示
```

```
[root@exercise1 opt]# vim /opt/break-continue.sh
#! /bin/sh
while true
do
   echo "**********"
   echo "Please select your operation:"
   echo " 1 Copy"
   echo " 2 Delete"
   echo " 3 Backup"
   echo " 4 Quit"
   echo "**********"
   read -p "请输入你的选择:" op
case $op in
  1)
continue #这里加了 continue 后，后面的 echo 命令就不执行了
echo "your selection is Copy"
   ;;
2)
    echo "your selection is Delete"
   ;;
3)
   echo "your selection is Backup"
   ;;
4)
   echo "Exit ..."
   break #跳出循环体
   ;;
   *)
   echo "invalide selection,please try again"
  esac
done
```

例子2：使用交互式方法批量添加用户

```
[root@exercise1 opt]# vim /opt/useradd.sh
#!/bin/bash
echo "**********"
```

```
read -p "请输入要创建的用户名：" name
read -p "请输入要创建的用户数：" num
read -p "请输入要创建用户密码：" pas
echo "**********"
for ((i=1;i<=$num;i=i+1))
do
    useradd $name$i &> /dev/null
    echo "$pas" | passwd --stdin $name$i &> /dev/null
done
echo "创建用户完成，结果是..."
tail -$num /etc/passwd
[root@exercise1 opt]# sh !$
```

```
[root@exercise1 opt]# vim /opt/useradd.sh
[root@exercise1 opt]# sh !$
sh /opt/useradd.sh
**********
请输入要创建的用户名：test
请输入要创建的用户数：5
请输入要创建用户密码：123456
**********
创建用户完成，结果是...
test1:x:1005:1005::/home/test1:/bin/bash
test2:x:1006:1006::/home/test2:/bin/bash
test3:x:1007:1007::/home/test3:/bin/bash
test4:x:1008:1008::/home/test4:/bin/bash
test5:x:1009:1009::/home/test5:/bin/bash
```

例子3：批量删除刚刚用于测试创建的用户

```
[root@exercise1 opt]# vim /opt/userdel.sh
#!/bin/bash

echo "**********"
read -p "请输入要删除的用户名：" name
read -p "请输入要删除的用户数：" num
echo "**********"
for ((i=1;i<=$num;i=i+1))
do
    userdel -r $name$i &> /dev/null
done
echo "批量删用户完成，结果是..."
```

```
tail -$num /etc/passwd

[root@exercise1 opt]# sh !$
```

```
[root@exercise1 opt]# vim /opt/userdel.sh
[root@exercise1 opt]# sh !$
sh /opt/userdel.sh
**********
请输入要删除的用户名：test
请输入要删除的用户数：5
**********
批量删用户完成，结果是...
ab:x:1000:1000::/home/ab:/bin/bash
cd:x:1001:1001::/home/cd:/bin/bash
ef:x:1002:1002::/home/ef:/bin/bash
gh:x:1003:1003::/home/gh:/bin/bash
ij:x:1004:1004::/home/ij:/bin/bash
```

```
例子3：批量删除普通用户(默认创建不改变家目录和shell类型的情况)
[root@exercise1 opt]# vim userdel2.sh
#!/bin/bash
echo "**********"
echo "开始准备批量删除普通用户……"
echo "**********"

for i in $(cat /etc/passwd|grep "/bin/bash"|grep -v "root"|cut -d: -f1)   #过滤shell类型是bash的用户同时
#取反去掉root用户，然后进行切割以冒号为分隔符读取第一段即用户名
do
        userdel -r $i
done
echo "批量删用户完成，结果是..."
tail -10 /etc/passwd
[root@exercise1 opt]# sh userdel2.sh
```

```
[root@exercise1 opt]# sh userdel2.sh
**********
开始准备批量删除普通用户……
**********
批量删用户完成，结果是...
operator:x:11:0:operator:/root:/sbin/nologin
games:x:12:100:games:/usr/games:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
nobody:x:99:99:Nobody:/:/sbin/nologin
systemd-network:x:192:192:systemd Network Management:/:/sbin/nologin
dbus:x:81:81:System message bus:/:/sbin/nologin
polkitd:x:999:997:User for polkitd:/:/sbin/nologin
postfix:x:89:89::/var/spool/postfix:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
chrony:x:998:996::/var/lib/chrony:/sbin/nologin
```

```
例子4：就是可能不是默认创建用户的情况下，就根据UID来删除
[root@exercise1 opt]# vim userdel3.sh
#!/bin/bash
for i in $(cat /etc/passwd|grep -v "root"|cut -d: -f3)
  #取反去掉root用户，然后进行切割以冒号为分隔符读取第三段即UID
do
        if [[ $i -ge 1000 ]];
        then
                echo "删除`id $i|tr -s "=" " "| cut -d " " -f2|cut -d "(" -f2|cut -d ")" -f1`成功"
                #id命令接UID查找出信息，以空格为分隔符读取第二段即UID(username),然后
                #按照括号切割取得括号里的username
                userdel -r $(id $i|tr -s "=" " "| cut -d " " -f2|cut -d "(" -f2|cut -d ")" -f1)
        fi
done
echo "************************************"
tail -10 /etc/passwd
[root@exercise1 opt]# sh userdel3.sh
```

```
[root@exercise1 opt]# sh userdel3.sh
删除test1成功
删除test2成功
删除test3成功
删除test4成功
删除test5成功
operator:x:11:0:operator:/root:/sbin/nologin
games:x:12:100:games:/usr/games:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
nobody:x:99:99:Nobody:/:/sbin/nologin
systemd-network:x:192:192:systemd Network Management:/:/sbin/nologin
dbus:x:81:81:System message bus:/:/sbin/nologin
polkitd:x:999:997:User for polkitd:/:/sbin/nologin
postfix:x:89:89::/var/spool/postfix:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
chrony:x:998:996::/var/lib/chrony:/sbin/nologin
```

Shift参数左移指令

hift命令用于对参数的移动(左移)，通常用于在不知道传入参数个数的情况下依次遍历每个参数然后进行相应处理（常见于Linux中各种程序的启动脚本)

在扫描处理脚本程序的参数时，经常要用到的shift命令，如果你的脚本需要10个或10个以上的参数，你就需要用shift命令来访问第10个及其后面的参数

作用：每执行一次，参数序列顺次左移一个位置，$#的值减1，用于分别处理每个参数，移出去的参数，不再可用


```
例子：加法计算器
[root@exercise1 opt]# vim shift.sh

if [ $# -le 0 ];then
echo "没有足够的参数"
exit
fi
sum=0
while [ $# -gt 0 ] ; do
#sum=$(expr $sum + $1)
sum=$[$sum+$1]
shift
```

```
# shift 2 一次移动 2 个参数
done
echo result is $sum
[root@exercise1 opt]# sh shift.sh 11 2 3 4
result is 20
[root@exercise1 opt]#
```

函数的使用

函数是一个脚本代码块，你可以对它进行自定义命名，并且可以在脚本中任意位置使用这个函数，要使用这个函数，只要使用这个函数名称就可以了。

使用函数的好处：模块化，代码可读性强。

创建语法:

```
方法1：
function    name {
        commands
}
name
注意：name是函数唯一的名称

方法2：name后面的括号表示你正在定义一个函数
name(){
    commands
}
name
```

调用函数语法：
函数名 参数1 参数2...
调用函数时，可以传递参数。在函数中用1、2...来引用传递的参数

函数的使用

```
例子：
```

```
[root@exercise1 opt]# vim fun-1.sh
#!/bin/bash
function  fun_1 {    #定义函数
    echo  "this   is  function"
}
fun_1   #调用函数

[root@exercise1 opt]# sh fun-1.sh
this is function
[root@exercise1 opt]#

相当于fun_1=`this   is  function`

注意：函数名的使用，如果在一个脚本中定义了重复的函数名，那么以最后一个为准

[root@exercise1 opt]# vim fun-2.sh
#!/bin/bash
function fun_1 {
    echo "this is function"
}
function fun_1 {
    echo "this is 2222222"
}
fun_1
[root@exercise1 opt]# sh fun-2.sh
this is 2222222
[root@exercise1 opt]#
```

返回值

使用return命令来退出函数并返回特定的退出码

```
[root@exercise1 opt]# vim fun-3.sh
#!/bin/bash
function fun_1 {
echo "this is function"
ls /etc/passwd
return 3
}
```

```
fun_1

[root@exercise1 opt]# sh fun-3.sh   #查看结果
this is function
/etc/passwd
[root@exercise1 opt]# echo $?
3
[root@exercise1 opt]#
```

注：状态码的确定必需要在函数一结束就运行return返回值；状态码的取值范围（0~255）

互动：exit数字和return数字的区别？

exit整个脚本就直接退出，返回数字；return只能让函数后面的命令不执行，然后返回数字，无法强制退出整个脚本的。

把函数值赋给变量使用

例子：函数名就相当于一个命令

```
[root@exercise1 opt]# vim fun-4.sh
#!/bin/bash
fun1(){
read -p "Input a value: " va
echo $[$va*5]
}
num=$(fun1)
echo current num is $num

[root@exercise1 opt]# sh fun-4.sh
Input a value:22
current num is 110
[root@exercise1 opt]#
```

函数的参数传递

第一种：通过脚本传递参数给函数中的位置参数$1

```
[root@exercise1 opt]# vim fun-5.sh
#!/bin/bash
fun1(){
    rm -rf $1   #第二步
}
fun1 $1  #第一步

[root@exercise1 opt]# sh fun-5.sh a.txt   #a.txt就是位置参数
[root@exercise1 opt]#
```

第二种：调用函数时直接传递参数

```
[root@exercise1 opt]# touch /opt/a.txt   #创建测试文件
[root@exercise1 opt]# vim fun-6.sh
#!/bin/bash
fun1(){
    rm -rf $1
}
fun1 /opt/a.txt
[root@exercise1 opt]# sh fun-6.sh
[root@exercise1 opt]# ls /opt/a.txt   #检验
ls: 无法访问/opt/a.txt: 没有那个文件或目录
[root@exercise1 opt]#
```

第三种：函数中多参数传递和使用方法

```
[root@exercise1 opt]# vim fun-7.sh
#!/bin/bash
fun1(){
    echo $[$1*5]
    echo $[$2*2]
}
fun1 5 2 #直接传两个参数
[root@exercise1 opt]# sh fun-7.sh
25
4
[root@exercise1 opt]#
```

例:

```
#!/bin/bash
function count {            #函数名为count
sum=0
while [ $# -gt 0 ] ; do
sum=$[$sum+$1]
done
echo result is $sum
}

count 99            #函数传参 $1=99

echo $1 $2 $3 $4 $5


[root@home opt]# sh m.sh 1 2 3 4 5 6
result is 99            #输出的结果是99，函数的$1,而不是m.sh的$1
1 2 3 4 5

```

函数中变量的处理

函数使用的变量类型有两种:

- 局部变量
- 全局变量

全局变量，默认情况下，你在脚本中定义的变量都是全局变量(针对于脚本是全局变量，但对于系统来说是局部变量)，
你在函数外面定义的变量在函数内也可以使用

```
[root@exercise1 opt]# vim fun-8.sh
#!/bin/bash
function fun1 {
        num1=$[var1*2]
}
read -p "input a num:" var1
fun1
echo the new value is : $num1

[root@exercise1 opt]# sh fun-8.sh
input a num:2
the new value is : 4
[root@exercise1 opt]#
```

随机数的生成

1.生成随机字符

```
[root@exercise1 opt]# cat /dev/urandom |tr -dc [:alnum:] |head -c 8
nRSCSps2
```

2.生成随机数

注：RANDOM 为系统自带的系统变量，值为 0-32767 的随机数

```
[root@exercise1 opt]# echo $[RANDOM%7]    #确定范围 echo $[RANDOM%7] 随机1个数，范围在（0-6）
1

[root@exercise1 opt]# echo $[$[RANDOM%7]+31]    #随机1个数，范围在（31-37）
32
```

实战-12个脚本实战

实战1.脚本生成一个 100 以内的随机数，提示用户猜数字，根据用户的输入，提示用户猜对了， 猜小了或猜大了，直至用户猜对脚本结束。

实战2.自动创建10个网卡配置文件在/opt目录下(for语句)，自动修改静态为自定义IP（同一网段），网卡命名ens+ip的最后一位，
规定IP范围不超过254（if，）：提示不能超过254的范围

实战3.编写脚本，实现人机<石头，剪刀，布>游戏
提示1：#通过随机数获取计算机的出拳
num=$[RANDOM%3]

实战4.检查特定的软件包是否已经安装

实战5.编写符合需求的功能性脚本

1.需要有菜单功能让用户选择，如下：

2.可以让用户自行选择添加用户，要判断用户是否存在，不存在则创建，并输出到密码文件中

3.可以让用户自行选择删除用户，要判断用户是否曾经创建过，如创建过，需清空家目录

4.可以让用户自行选择修改密码，并更新到密码文件中

5.可以让用户查看当前内存使用率

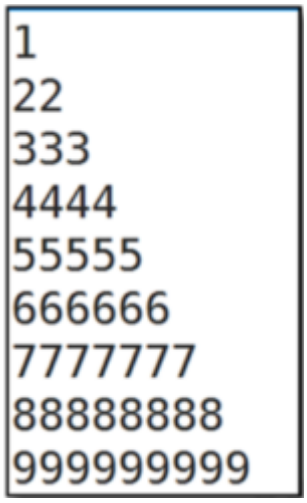
6.可以让用户查看当前磁盘使用率

7.可以让用户查看当前cpu使用率

8.有俄罗斯方块游戏玩

9.直接退出

实战6.打印以下图形



实战7.打印以下图形

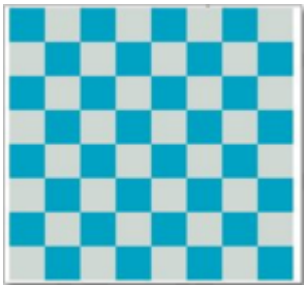
实战8.打印以下图形

实战9.输入不同数字会有不同的命令，

需求1：需要检测输入的内容不能为空，不能非数字

需求2：1 为ls命令；2为id命令；3为who命令；4为ll命令

实战10：打印国标象棋



实战11：输入行数，就可以打印几行的彩色等腰三角形

```
[root@home opt]# sh k.sh
Please input a num: 5
     *
    ***
   *****
  *******
*********
```