Shell脚本编程学习笔记

——基于Linux嵌入式

目录

预备知识

1. shell 脚本语言：把经常使用的 Linux 命令存储在一个文件里面，shell 可以读取这个文件并***顺序执行***其中的命令，这样的文件被称为脚本文件。注意：shell 脚本按行解释。
2. shell 程序设计中，用 vi 或者 gedit 等编辑器编写的代码并不需要什么后缀名。如果硬要加上，则写上.sh 即可。有一点值得注意，当我们加上后缀之后，编辑器会使用高亮的语法显示，避免我们出错。
3. 编写的过程中，***echo 之后的引用为“`”,即 Esc 返回键的下面一个***，不要认为是单引号。准确来说，是反单引号。在 shell 命令中，一般用反单引号。
4. 编程结束之后，我们要先赋予文件的执行权利，可以用 sudo chmod 755 file，sudo chmod 777 file 或者 sudo chmod a+x file 来赋予程序的可执行权利。
5. 前面的 1，2，3 弄好了之后，就是运行了，直接在终端输入：./file 即可。
6. 不要为了美观，在命令中添加多余的空格，否则会出错。特别是在环境变量 PATH,export 等，等号两侧不要加多余的空格。
7. shell 中的全局变量有以下几个：
   $#: 执行 shell 脚本时的命令行参数，不包括脚本本身
   $?: 执行上一条 shell 命令的返回值
   $0: shell 脚本程序自身的名称
8. 用户在自定义变量时，要遵循以下命名规则：
   首个字符必须是字母（a~z, A~Z）。
   中间不能有空格，可以使用下划线（_）。
   不能使用标点符号。
   不能使用 shell 里的关键字（通过 help 命令查看）。
9. 在 Linux 中，自动（或者叫预定义）变量包括$@，$+，$^，$?，$<，$*等。其中（以下内容来源于 Makefile 的讲解），
   $@表示规则中的目标文件夹集；
   $+表示所有的依赖文件，以空格隔开，并以出现的先后为顺序，可能包含重复的依赖文件；
   $^表示所有的依赖文件，以空格隔开，不包含重复的依赖文件；
   $?表示所有比目标新的依赖目标集合；
   $<表示第一个依赖文件的名称；
   $*表示不包括扩展的目标文件名。

第一节　shell 运算符应用

表达式测试包括字符串测试、整数测试、文件测试及逻辑测试。
内置测试命令 test
通常用 test 命令来测试表达式的值，如下：

```
x=5; y=10
test $x -gt $y
echo $?
```

test 命令可以用方括号“[ ]”来代替：

```
x=5; y=10
```

```
[ $x -gt $y ]
echo $?
```

2.x 版本以上的 Bash 中可以用双方括号来测试表达式的值，此时可以使用通配符进行模式匹配，如：

```
name=Tom
[ $name = [Tt]?? ]
echo $?
```

或者：

```
[[ $name = [Tt]?? ]]
echo $?
```

检查空值：

```
[ "$name" = "" ]
[ ! "$name" ]
[ "X${name}" != "X" ]
```

1. 整数测试运算

test int1 -eq int2: 判断两个数是否相等
test int1 -ne int2: 判断两个数是否不相等
test int1 -gt int2: 判断整数 1 是否大于整数 2
test int1 -ge int2: 判断整数 1 是否大于等于整数 2
test int1 -lt int2: 判断整数 1 是否小于整数 2
test int1 -le int2: 判断整数 1 是否小于等于整数 2

整数测试也可以使用 let 命令或双圆括号
相关操作为：== 、!= 、> 、>= 、< 、<=
如：

```
x=1; [ $x -eq 1 ]; echo $?
x=1; let "$x == 1"; echo $?
x=1; (($x+1>= 2 )); echo $?
```

两种测试方法的区别：

使用的操作符不同
let 和 双圆括号中可以使用算术表达式，而中括号不能
let 和 双圆括号中，操作符两边可以不留空格

2. 字符串测试运算

test -z string: 判断字符串长度是否为 0，即判断字符串是否为空，为空返回真，非空返回假

test -n string: 判断字符串长度是否不为 0，即判断字符串是否非空，为空返回假，非空返回真

test str1 =str2:判断两个字符串是否相等
test str1!=str2:判断两个字符串是否不等

如：

```
name=Tom; [ -z $name ]; echo $?
name2=Andy; [ $name = $name2 ] ; echo $?
```

3. 文件测试运算

test -r filename:　判断用户对文件 filename 是否有读权限
test -w filename: 判断用户对文件 filename 是否有写权限
test -x filename:　判断用户对文件 filename 是否有可执行权限
test -f filename: 判断文件 filename 是否为普通文件
test -d filename: 判断文件 filename 是否为目录
test -c filename: 判断文件 filename 是否为字符设备
test -b filename: 判断文件 filename 是否为块设备
test -s filename: 判断文件 filename 是否大小不为 0
test -t fnum:　　判断与文件描述符 fnum（默认值为 1）相关的设备是否是一个终端设备

4. 逻辑运算

test 表达式 1 -a 表达式 2：与（and）逻辑判断。如果两个表达式同时为真则返回真，否者返回假。
test 表达式 1 -o 表达式 2：或（or） 逻辑判断。只要两个表达式有一个为真则返回真，否则返回假。

如：

```
x=1; name=Tom;
[ $x -eq 1 –a –n $name ]; echo $?
```

注：不能随便添加括号

第二节　在 shell 脚本中进行条件控制

1．Bash 中允许测试两种类型的条件：命令成功或失败，表达式为真或假
2．任何一种测试中，都要有退出状态（返回值），退出状态为 0
表示命令成功或表达式为真，非 0 则表示命令失败或表达式为假。
3．状态变量 $? 中保存命令退出状态的值

if 表达式有：

```
if [ 条件表达式 1 ]
then
    if [ 条件表达式 2 ]
    then
    ……
    ……
    else
    ……
    ……
    fi
      命令串;
else
     命令串;
fi
```

注意：上述 if 语法中，中括号“[]”中的内容用于进行条件测试。使用”[]“进行条件测试时，要注意空格的使用。在 if 与”[“间要有空格，在”[“与后面的条件表达式之间要有空格，在”]“与前面的条件表达式之间也要有空格。

还有另外一种：

```
if expr1                # 如果 expr1 为真(返回值为 0)
then                    # 那么
    commands1           # 执行语句块 commands1
elif expr2              # 若 expr1 不真，而 expr2 为真
then                    # 那么
    commands2           # 执行语句块 commands2
  ... ...               # 可以有多个 elif 语句
else                    #  else 最多只能有一个
    commands4           # 执行语句块 commands4
fi                      # if 语句必须以单词 fi 终止
```

说明：

1.elif 可以有任意多个（0 个或多个）

2.else 最多只能有一个（0 个或 1 个）

3.if 语句必须以 fi 表示结束

4.expr 通常为条件测试表达式；也可以是多个命令，以最后一个命令的退出状态为条件值。

5.commands 为可执行语句块，如果为空，需使用 shell 提供的空命令 “ : ”，即冒号。该命令不做任何事情，只返回一个退出状态 0

6. if 语句可以嵌套使用,如：

ex4if.sh，chkperm.sh，chkperm2.sh，

name_grep，tellme，tellme2，idcheck.sh

使用 case 语句有：

```
case string in
    str1)
    命令串 1;;  # 执行语句块命令，注意后面为“;;”双分号
    str2)
    命令串 2;;
    *)
    默认处理命令串;
esac            # esac 实际上就是 case 反过来写
```

case 详解：

```
case expr in        # expr 为表达式，关键词 in 不要忘！
pattern1)           # 若 expr 与 pattern1 匹配，注意括号
commands1           # 执行语句块 commands1
;;                  # 跳出 case 结构
pattern2)           # 若 expr 与 pattern2 匹配
commands2           # 执行语句块 commands2
;;                  # 跳出 case 结构
  ... ...           # 可以有任意多个模式匹配
```

```
*)              # 若 expr 与上面的模式都不匹配
commands        # 执行语句块 commands
;;              # 跳出 case 结构，为两个双分号
esac            # case 语句必须以 esac 终止
```

几点说明：

1. 表达式 expr 按顺序匹配每个模式，一旦有一个模式匹配成功，则执行该模式后面的所有命令，然后退出 case。

2. 如果 expr 没有找到匹配的模式，则执行缺省值“*)”后面的命令块（类似于 if 中的 else）；“*)”可以不出现。

3. 所给的匹配模式 pattern 中可以含有通配符和“|”。

4. 每个命令块的最后必须有一个双分号，可以独占一行，或放在最后一个命令的后面。

5. 一般来说，在条件判断中，if then……[else]……if,(else 可以不要)，是成对出现的，也是遵循 C 语言中 if 的就近原则，即 else、fi 都是与其最近相邻的 if 配对的。对于 case,也是类似 C 语言中的 switch 语句，参照 switch 理解，每一个 case 一定有一个 esac 与之配对。

第三节　在 shell 脚本中进行 for 循环

for 的第一种形式：

```
for var in list
do
命令串
done
```

说明：

list: 列表，可以是由空格分隔的变量（$a $b）或者是值（1 2 3 等）。

对于 list 中的每一项，都将循环一次

var : 每次循环的值。对于 list 中的每一项都要进行一次循环，而每次循环时，就取出 list 中的第几项放在 var 中，可以在命令串中通过$var 的方式进行引用

循环执行过程：

执行第一轮循环时，将 list 中的第一个词赋给循环变量，并把该词从 list 中删除，然后进入循环体，执行 do 和 done 之间的命令。下一次进入循环体时，则将第二个词赋给循环变量，并把该词从 list 中删除，再往后的循环也以此类推。当 list 中的词全部被移走后，循环就结束了。

for 的第二种形式：

```
for var
do
命令串
done
```

说明：

与第一种方式相比，少了 in list 项。此时，for 循环省略了 list,list 值由当前脚本程序的命令行参数代替。也就是说，在这种情况下，没循环一次，var 中存储的就是一个命令行参数。

对于 for 语句,我的理解是，如果没有 in list，则循环参数则由命令行带参使用，即以命令行参数为 for 循环值列表。

第四节　在 shell 脚本中进行 while 循环

在某些情况下，需要依据某个条件进行判断，如果条件为真则继续循环，否则结束循环，这种情况下，就需要使用 while 循环。while 循环是当某个判定条件的值为假时退出循环。

语法如下：

```
while 条件表达式
do
命令串;
done
```

执行过程：

先执行条件表达式，如果其退出状态为 0，就执行循环体。执行到关键字 done 后，回到循环的顶部，while 命令再次检查条件表达式的退出状态。以此类推，循环将一直继续下去，直到条件表达式的退出状态非 0 为止。

第五节　在 shell 脚本中使用 util 循环

util 循环和 while 循环实现的功能基本相同，不过 util 是判定条件为假时才继续循环。

语法如下：

```
util 条件表达式
do
命令行;
done
```

第六节　在 shell 脚本中使用函数

对于在脚本中经常重复使用的功能模块，可以将其封装成函数。在 shell 脚本中使用函数，可以使程序代码更加简洁，程序的可读性更好。shell 脚本中函数的定义有如下两个：

```
函数名()
{
    ……
}
```

或者：

```
function 函数名()
{
    ……
}
```

在调用函数时，可以向函数传递任意个数的参数。在函数中可以通过位置参数 $1,$2 的方式进行引用。函数也可以有返回值，可以通过查询全局变量$?的值获取函数的返回值。

第七节 shell 脚本之 echo 和 expr 讲解

echo 命令

功能说明：显示文字。
语　法：echo [-ne][字符串] 或 echo [--help][--version]
补充说明：echo 会将输入的字符串送往标准输出。输出的字符串间以空白字符隔开, 并在最后加上换行号。

-n 不进行换行

-e 若字符串中出现以下字符，则特别加以处理，而不会将它当成一般文字输出 \n 换行 \b 空格...

-n 不要在最后自动换行

-e 若字符串中出现以下字符，则特别加以处理，而不会将它当成一般文字输出：

\a 发出警告声；
\b 删除前一个字符；
\c 最后不加上换行符号；
\f 换行但光标仍旧停留在原来的位置；
\n 换行且光标移至行首；
\r 光标移至行首，但不换行；
\t 插入 tab；
\v 与\f 相同；
\\ 插入\字符；
\nnn 插入 nnn（八进制）所代表的 ASCII 字符；
--help 显示帮助
--version 显示版本信息

expr 命令

生成随机数的特殊变量：

echo $RANDOM　　范围是: [0, 32767]

expr：通用的表达式计算命令
注意：表达式中参数与操作符必须以空格分开，表达式中的运算可以是算术运算，比较运算，字符串运算和逻辑运算

1．expr 命令一般用于整数值，但也可用于字符串。
如：

```
expr 5 % 3
expr 5 \* 3   # 乘法符号必须被转义
```

2．expr 也是一个手工命令行计数器。如：

$expr 10 + 10　　打印：20
$expr 30 / 3　　打印：10
$expr 30 / 3 / 2　　打印： 5

注意：使用乘号时，必须用反斜线屏蔽其特定含义。因为 shell 可能会误解显示星号的意义（shell 中的反斜杠“\”，有大部分是这种转意的意思）。如：
expr 30 * 3　　打印：90

3．数值测试。可以用 expr 测试一个数。如果试图计算非整数，将返回错误。如：

```
$rr=1.1
$expr $rr + 1          以上两句将打印：expr: non-numeric argument
$rr=2
$expr $rr + 1          以上两句将打印：3
```

这里需要将一个值赋予变量（不管其内容如何），进行数值运算，并将输出导入 dev/null，然后测试最后命令状态，如果为 0，证明这是一个数，其他则表明为非数值。

```
$value=12
$expr $value + 10 > /dev/null 2>&1
$echo $?                    以上三句将打印：0　这是一个数。
$value=hello
$expr $value + 10 > /dev/null 2>&1
$echo $?                    以上三句将打印：2　这是一个非数值字符。
```

expr 也可以返回其本身的退出状态，不幸的是返回值与系统最后退出命令刚好相反，成功返回 1，任何其他值为无效或错误。下面的例子测试两个字符串是否相等，这里字符串为“hello”和“hello”。

```
$value=hello
$expr $value = "hello"          此时打印：1
$echo $?                        此时打印：0
```

expr 返回 1。不要混淆了，这表明成功。现在检验其最后退出状态，返回 0 表示测试成功，“hello”确实等于“hello”。

4．模式匹配。expr 也有模式匹配功能。可以使用 expr 通过指定冒号选项计算字符串中字符数。.*意即任何字符重复 0 次或多次。 如：

```
$value=accounts.doc
$expr $value : '.*'
```

在 expr 中可以使用字符串匹配操作，这里使用模式. d o c 抽取文件附属名。

```
$expr $value : '(.*).doc'
```

则会打印：accounts

对于 expr 的解释，估计有人会说，写得很罗嗦。我想说的是，expr 的确很重要！在后面的脚本学习中，对于大型程序，很多地方用到了 expr。

第八节 shell 脚本循环终止之 break 和 continue

break [n]

1.用于强行退出当前循环。

2.如果是嵌套循环，则 break 命令后面可以跟一数字 n，表示退出第 n 重循环（最里面的为第一重循环）。

continue [n]

1.用于忽略本次循环的剩余部分，回到循环的顶部，继续下一次循环。

2.如果是嵌套循环，continue 命令后面也可跟一数字 n，表示回到第 n 重循环的顶部。

第九节 shell 脚本之 exit 和 sleep

exit 命令： exit n

exit 命令用于退出脚本或当前进程。n 是一个从 0 到 255 的整数，

0 表示成功退出，非零表示遇到某种失败而非正常退出。该整数被保存在状态变量 $? 中。

sleep 命令：sleep n

暂停 n 秒钟

第十节 shell 脚本之 select 循环与菜单

语法结构：

```
select variable in list
do                  # 循环开始的标志
   commands         # 循环变量每取一次值，循环体就执行一遍
done                # 循环结束的标志
```

说明：

1.select 循环主要用于创建菜单，按数字顺序排列的菜单项将显示在标准错误上，并显示 PS3 提示符，等待用户输入

2.用户输入菜单列表中的某个数字，执行相应的命令

3.用户输入被保存在内置变量 REPLY 中。

select 与 case：

1.select 是个无限循环，因此要记住用 break 命令退出循环，
或用 exit 命令终止脚本。也可以按 ctrl+c 退出循环。

2.select 经常和 case 联合使用

3.与 for 循环类似，可以省略 in list ，此时使用位置参量

第十一节 shell 脚本之循环控制 shift 命令

shift [n]

1.用于将参量列表 list 左移指定次数，缺省为左移一次。

2.参量列表 list 一旦被移动，最左端的那个参数就从列表中删除。

while 循环遍历位置参量列表时，常用到 shift。

例如：

```
./doit.sh a b c d e f g h
./shft.sh a b c d e f g h
```

第十二节 shell 脚本之字符串操作

字符串操作：

${#var}	返回字符串变量 var 的长度
${var:m}	返回${var}中从第 m 个字符到最后的部分，其中，m 的取值从 0 到${#var}-1，下同
${var:m:len}	返回${var}中从第 m 个字符开始，长度为 len 的部分

${var#pattern}	删除${var}中开头部分与 pattern 匹配的最小部分
${var##pattern}	删除${var}中开头部分与 pattern 匹配的最大部分
${var%pattern}	删除${var}中结尾部分与 pattern 匹配的最小部分
${var%%pattern}	删除${var}中结尾部分与 pattern 匹配的最大部分
${var/old/new}	用 new 替换${var}中第一次出现的 old
${var//old/new}	用 new 替换${var}中所有的 old(全局替换)

注：pattern，old 中可以使用通配符，

第十三节 shell 脚本之数组实现

Shell 中的数组用下面的方式来定义：
$varname[0]=value1
$varname[1]=value2
....
用$echo ${varname[0]}方式来引用

第十四节 shell 脚本之脚本调试

1. sh –x 脚本名
说明：该选项可以使用户跟踪脚本的执行，此时 shell 对脚本中每条命令的处理过程为：先执行替换，然后显示，再执行它。shell 显示脚本中的行时，会在行首添加一个加号“+”。
2. sh –v 脚本名
说明：在执行脚本之前，按输入的原样打印脚本中的各行，打印一行执行一行。
3. sh –n 脚本名
说明：对脚本进行语法检查，但不执行脚本。如果存在语法错误，shell 会报错，如果没有错误，则不显示任何内容。

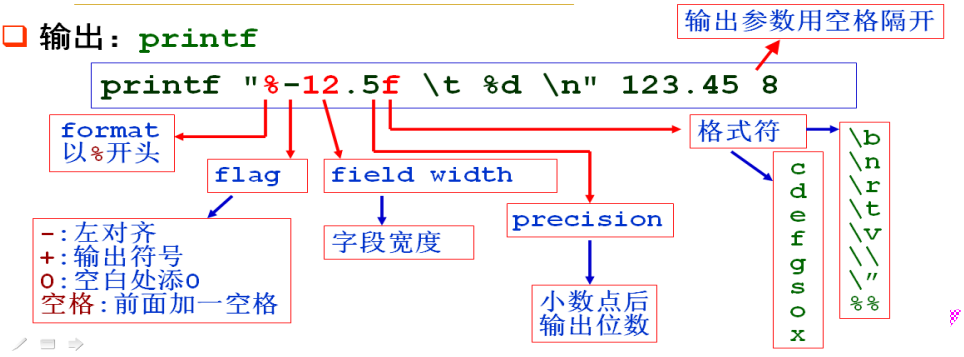
第十五节 shell 脚本之编程小结

变量：
1. 局部变量、环境变量（export、declare -x）
2. 只读变量、整型变量
例：declare -i x; x="hello"; echo $x　# 打印 0
3. 位置参量（$0,$1,...,$*,$@,$#,$$,$?）
4. 变量的间接引用（eval, ${!str}）
例：name="hello"; x="name"; echo ${!x} # 打印 hello
5. 命令替换（`cmd`、$(cmd)）
6. 整数运算
declare 定义的整型变量可以直接进行运算，否则需用 let 命令或 $[...]、$((...)) 进行整数运算。

输入输出：
1. 输入：read
read var1 var2 ...

read □-> REPLY
read –p "提示" -> REPLY

2. 输出：printf



程序例程

1. autoftp

```
#!/bin/bash
ftp -n 192.168.137.165<<!   # 使用非交互模式的 ftp，用！作分隔符
user abc password           # 用户名和密码
get   $1                    # 以命令行参数 1 作为要获取的文件名
bye                         # 退出 ftp
!                           # ！分隔符，命令结束
```

2. computesum

```
# 从命令行输入多个以空格分隔的数字，输出全部数字的和
#!/bin/bash
sum=0
for ccc
do
    if [ `expr $ccc \>= 0` = 1 ]
    then
        echo -e "$ccc+\c"              # \c 表示输出加数之后不按<Enter>键
        sum=`expr $sum + $ccc`
    fi
done
    echo -e "\b=$sum"                 # \b 表示退格
```

3. diff

```
# shell 编写的 diff 脚本，比较从命令行输入的两个整数的值是否相等
    #!/bin/bash
    if test $1 = $2
    then
        echo $1=$2
    else
        echo $1!=$2
    fi
# 在命令的逻辑判断语句中，用 fi 表示结束 if 语句
```

4. echodate

```
    #!/bin/bash
    echo `date +%y%m%d`
```

5. fact.sh

```
# 从命令行输入数字，计算该数字的阶乘并输出
# 函数中使用递归，函数的两个参数$1=本次阶乘因子，
# $2=上次阶乘结果
#!/bin/bash
function fact()
{
                    # 如果阶乘因子等于 0，表明阶乘计算完成，输出结果返回
    if [ $1 -eq 0 ]
        then
            echo $2
            return
        fi
# 将当前的阶乘因子与上一次递归计算所得的阶乘结果相乘，
# 并将结果存放于变量 t 中
        t=`expr $2 \* $1`              # 调用 expr 计算两个数的积
#递归调用
        fact `expr $1 - 1` $t              # 递归调用当前函数
}

# 调用函数 fact 计算阶乘，第一个参数为从命令行输入的要求结成的变量的值
# 第二个参数是初始的递归结果，设为 1
fact $1 1                    # 调用函数 fact 计算命令行参数 1 的阶乘
```

6. sum

```
#!/bin/bash
sum=`expr $1 + $2`     # 在使用 expr 进行计算时，运算符两边应该加空格
echo $1+$2=$sum
```

7. testcmdline

```
# 从命令行输入多个以空格分隔的参数，使用 while 循环枚举全部参数
    #!/bin/bash
    n=1          # 初始化变量 n
                 # while 循环，测试$*是否为空
                 # $*包含了全部命令行参数，在循环体中调用 shift 向参
                 # 数向左移除，导致$*最终变为空而循环结束
    while [ -n "$*" ]
    do
        echo  第$n 个参数=$1

        n=`expr $n + 1`
```

```
        shift
    done

#注意：在循环体中使用 shift 将命令行参数左移，这样，没循环一次，命令行参
#       数将从左侧被移走一个，是原来处于第二位命令行参数变成第一位。最
#       终$*所#代表的全部命令行参数将为空，循环结束。
```

8. weekday

```
# 输入数字，输出该数字对应的是星期几
      #!/bin/bash
      if [ $# -lt 1 ]
      then
        echo Usage:$0  数字
        exit
      fi

        case $1 in
            1)
            echo monday;;
            2)
            echo tuesday;;
            3)
            echo wednesday;;
            4)
            echo thursday;;
            5)
            echo friday;;
            6)
            echo saturday;;
            7)
            echo sunday;;
            *)
            echo invalidate;
        esac
```

9. greetings

```
#!/bin/bash
# This is the first Bash shell program
# ScriptName: greetings
    echo
    echo –e "Hello $LOGNAME, \c"
```

```
    echo      "it's nice talking to you."
    echo      "Your present working directory is:"pwd
# Show the name of present directory
    echo
    echo –e "The time is `date +%T`!. \nBye"
    echo
```

10. ex4read

```
# !/bin/bash
# This script is to test the usage of read
# Scriptname: ex4read
    echo "=== examples for testing read ==="
    echo -e "What is your name? \c"
    read name
    echo "Hello $name"
    echo
    echo -n "Where do you work? "
    read
    echo "I guess $REPLY keeps you busy!"
    echo
    read -p "Enter your job title: "     #自动读给 REPLY
    echo "I thought you might be an $REPLY."
    echo
    echo "=== End of the script ==="
# read variable：  读取变量给 variable
# read x y：  可同时读取多个变量
# read：  自动读给 REPLY
# read –p “Please input: ”自动读给 REPLY
```

11. ex4if

```
#!/bin/bash
# scriptname: ex4if
    echo -n "Please input x,y: "
    read x y
    echo "x=$x, y=$y"
    if (( x > y )); then
        echo "x is larger than y"
    elif (( x == y)); then
        echo "x is equal to y"
    else
        echo "x is less than y"
    fi
```

12. chkperm

```
#!/bin/bash
# Using the old style test command: [ ]
# filename: chkperm
    file=testing
    if [ -d $file ]
    then
        echo "$file is a directory"
    elif [ -f $file ]
    then
        if [ -r $file -a -w $file -a -x $file ]
        then        # nested if command
            echo "You have read,write,and execute permission on $file."
        fi
    else
        echo "$file is neither a file nor a directory. "
    fi
```

13. chkperm2

```
#!/bin/bash
# Using the new style test command: [[ ]]
# filename: chkperm2
    file=./testing
    if [[ -d $file ]]
    then
        echo "$file is a directory"
    elif [[ -f $file ]]
    then
        if [[ -r $file && -w $file && -x $file ]]
        then        # nested if command
            echo "You have read,write,and execute permission on $file."
        fi
    else
        echo "$file is neither a file nor a directory. "
    fi
```

14. name_grep

```
#!/bin/bash
# filename: name_grep
    name=Tom
    if grep "$name" /etc/passwd >& /dev/null
```

```
then
    :
else
    echo "$name not found in /etc/passwd"
    exit 2
fi
```

15. tellme

```
#!/bin/bash
    echo -n "How old are you? "
    read age
    if [ $age -lt 0 -o $age -gt 120 ]
    then
        echo    "Welcome to our planet! "
        exit 1
    fi
    if    [ $age -ge 0 -a $age -le 12 ]
    then
        echo "Children is the flowers of the country"
    elif [ $age -gt 12 -a $age -le 19 ]
    then
        echo "Rebel without a cause"
    elif [ $age -gt 19 -a    $age -le 29 ]
    then
        echo "You got the world by the tail!!"
    elif [ $age -ge    30 -a    $age -le 39 ]
    then
        echo "Thirty something..."
    else
        echo "Sorry I asked"
    fi
```

16. tellme2

```
#!/bin/bash
    echo -n "How old are you? "
    read age
    if (( age < 0 || age > 120 ))
    then
        echo "Welcome to our planet! "
        exit 1
    fi
    if ((age >= 0 && age <= 12))
    then
```

```
        echo "Children is the flowers of the country"
    elif ((age >= 13 && age <= 19 ))
    then
        echo "Rebel without a cause"
    elif (( age >= 19 &&    age <=    29 ))
    then
        echo "You got the world by the tail!!"
    elif    (( age >=    30 &&    age <= 39 ))
    then
        echo "Thirty something..."
    else
        echo "Sorry I asked"
    fi
```

17. idcheck.sh

```
#!/bin/bash
# Scriptname: idcheck.sh
# purpose: check user id to see if user is root.
# Only root has a uid of 0.
# Format for id output: uid=501(tt) gid=501(tt) groups=501(tt)
# root's uid=0 ：    uid=0(root) gid=0(root) groups=0(root)…
    id=`id | awk -F'[=(]'    '{print $2}'`        # get user id
    echo "your user id is: $id"
    if    (( id == 0 ))        # [ $id -eq 0 ]
    then
        echo "you are superuser."
    else
        echo "you are not superuser."
    fi
```

18. yes_no.sh

```
#!/bin/bash
# test case
# scriptname: yes_no.sh
    echo -n "Do you wish to proceed [y/n]: "
    read ans
    case $ans in
        y|Y|yes|Yes)
            echo "yes is selected"
            ;;
        n|N|no|No)
            echo "no is selected"
```

```
        ;;
    *)
        echo "`basename $0`: Unknown response"
        exit 1
        ;;
esac
```

19. forloop.sh

```
#!/bin/bash
# Scriptname: forloop.sh
    for name in Tom Dick Harry Joe
    do
        echo "Hi $name"
    done
    echo "out of loop"
```

20. forloop2.sh

```
#!/bin/bash
# Scriptname: forloop2.sh
    for name in `cat namelist`
    do
        echo "Hi $name"
    done
    echo "out of loop"
```

21. mybackup.sh

```
#!/bin/bash
# Scriptname: mybackup.sh
# Purpose: Create backup files and store
# them in a backup directory.
#
    backup_dir=backup
    mkdir $backup_dir
    for file in *.sh
    do
        if [ -f $file ]
        then
            cp $file $backup_dir/${file}.bak
            echo "$file is backed up in $backup_dir"
        fi
    done
```

22. greet.sh

```
#!/bin/bash
# Scriptname: greet.sh
# usage: greet.sh Tom John Anndy

    echo "== using \$* =="
    for name in $*    # same as for name in $@
    do
        echo Hi $name
    done
    echo "== using \$@ =="
    for name in $@    # same as for name in $*
    do
        echo Hi $name
    done
    echo '== using "$*" =='
    for name in "$*"
    do
        echo Hi $name
    done
    echo '== using "$@" =='
    for name in "$@"
    do
        echo Hi $name
    done
```

23. permx.sh

```
#!/bin/bash
# Scriptname: permx.sh
#
    for file    # Empty wordlist
    do
      if [[ -f $file && ! -x $file ]]
      then
        chmod +x $file
        echo " == $file now has execute permission"
      fi
    done
```

24. months.sh

```
#!/bin/bash
# Scriptname: months.sh
    for month in Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
    do
      for week in    1 2 3 4
      do
        echo -n "Processing the month of $month. OK? "
        read ans
        if [ "$ans" = n    -o -z "$ans" ]
        then
          continue 2
        else
         echo -n "Process week $week of $month? "
                read ans
          if [ "$ans" = n -o -z "$ans"    ]
          then
            continue
          else
            echo "Now processing week $week of $month."
            sleep 1    # Commands go here
          echo    "Done processing..."
          fi
        fi
      done
    done
```

25. runit.sh

```
#!/bin/bash
# Scriptname: runit.sh

    PS3="Select a program to execute: "
    select program    in 'ls -F' pwd date
    do
        $program
    done
```

26. goodboy.sh

```
#!/bin/bash
# Scriptname: goodboys.sh
    PS3="Please choose one of the three boys : "
```

```
    select choice in tom dan guy
#select choice
    do
      case $choice in
        tom)
            echo Tom is a cool dude!
            break;;    # break out of the select loop
        dan | guy )
            echo Dan and Guy are both wonderful.
            break;;
        *)
            echo "$REPLY is not one of your choices"
            echo "Try again."
            ;;
       esac
    done
```

27. doit.sh

```
#!/bin/bash
# Name: doit.sh
# Purpose: shift through command line arguments
# Usage: doit.sh [args]
    while (( $# > 0 ))    # or [ $# -gt 0 ]
    do
      echo   $*
      shift
    done
```

28. shft.sh

```
#!/bin/bash
# Using 'shift' to step through all the positional parameters.

    until [ -z "$1" ]    # Until all parameters used up...
    do
      echo "$1"
      shift
    done
    echo                        # Extra line feed.
    exit 0
```

29. ex4str

```
#!/bin/bash
    dirname="/usr/bin/local/bin";
    echo "dirname=$dirname"
    echo -n '${#dirname}='; sleep 4;echo "${#dirname}"
    echo
    echo -n '${dirname:4}=';    sleep 4;echo "${dirname:4}"
    echo
    echo -n '${dirname:8:6}='; sleep 4; echo ${dirname:8:6}
    echo
    echo -n '${dirname#*bin}='; sleep 4; echo ${dirname#*bin}
    echo
    echo -n '${dirname##*bin}='; sleep 4;echo ${dirname##*bin}
    echo
    echo -n '${dirname%bin}='; sleep 4;echo ${dirname%bin}
    echo
    echo -n '${dirname%%bin}='; sleep 4;echo ${dirname%%bin}
    echo
    echo -n '${dirname%bin*}='; sleep 4;echo ${dirname%bin*}
    echo
    echo -n '${dirname%%bin*}='; echo ${dirname%%bin*}
    echo
    echo -n '${dirname/bin/sbin}='; echo ${dirname/bin/sbin}
    echo
    echo -n '${dirname//bin/lib}='; echo ${dirname//bin/lib}
    echo
    echo -n '${dirname/bin*/lib}='; echo ${dirname/bin*/lib}
```

30. ex4fun2.sh

```
#!/bin/bash
    JUST_A_SECOND=1
    fun ()
    {                   # A somewhat more complex function
      i=0
      REPEATS=5
      echo
      echo "And now the fun really begins."
      echo
      sleep $JUST_A_SECOND              # Hey, wait a second!
      while [ $i -lt $REPEATS ]
      do
        echo "----------FUNCTIONS---------->"
```

```
        echo "<------------ARE------------"
        echo "<------------FUN------------>"
      echo
      let "i+=1"
      done
}
                    # Now, call the functions.
fun
exit 0
```

31. ex4fun3.sh

```
# f1
# Will give an error message, since function "f1" not yet defined.
# declare -f f1
# This doesn't help either.
# f1
# Still an error message.
# However...
    f1 () {
    echo "Calling function \"f2\" from within function \"f1\"."
    f2
    }
    f2 () {
    echo "Function \"f2\"."
    }
# f1
# Function "f2" is not actually called until this point
# although it is referenced before its definition.
# This is permissible.
```

32. ex4fun4.sh

```
#!/bin/bash
# Functions and parameters

    DEFAULT=default # Default param value.
    func2 () {
      if [ -z "$1" ]          # Is parameter #1 zero length?
      then
        echo "-Parameter           #1 is zero length -"
      else
        echo "-Param          #1 is \"$1\" -"
```

```
fi
variable=${1:-$DEFAULT}
echo "variable = $variable"
if [ -n "$2" ]
then
   echo "- Parameter        #2 is \"$2\" -"
fi
return 0
}
echo
echo "Nothing passed"
func2                  # Called with no params
echo
echo "One parameter passed."
func2 first            # Called with one param

echo
echo "Two parameters passed."
func2 first second     # Called with two params

echo
echo "\"\" \"second\" passed."
func2 "" second
                  # The first parameter is of zero?length

echo
exit 0
                  # End of script
```

33. ex4fun5.sh

```
#!/bin/bash
# function and command line arguments
# Call this script with a command line argument,
# something like $0 arg1.
    func ()
    {
    echo "$1"
    }

    echo "First call to function: no arg passed."
    echo "See if command-line arg is seen."
    Func               # No! Command-line arg not seen.
    echo   "================================="
    echo
```

```
echo "Second call to function: command-line arg passed explicitly."
func $1
                    # Now it's seen!
exit 0
```

34. ex4fun6.sh

```
#!/bin/bash
# purpose: Maximum of two integers.
    max2 ()             # Returns larger of two numbers.
    {if [ -z $2 ]
      then
        echo "Need to pass two parameters to the function."
        exit 1
      fi
      if [[ $1 == $2 ]]     # [ $1 -eq $2 ]
      then
        echo "The two numbers are equal."
        exit 0
      else
        if [ $1 -gt $2 ]
        then
          return $1
        else
        return $2
        fi
      fi
    }
    read num1 num2
    echo "num1=$num1, num2=$num2"

    max2 $num1 $num2

    return_val=$?

    echo "The larger of the two numbers is：  $return_val."

    exit 0
```

习题实训

1、 编写一个 shell 脚本，判断用户输入的字母，如 A~D。
2、 编写一个 shell 脚本，在 while 循环中判断用户输入的数字，当数字大雨 5 时，跳出循环。
3、 编写一个 shell 脚本，输出 1~10 中的所有奇数，并计算它们的和。
4、 编写一个 shell 脚本，从键盘输入两个数，使用函数计算并输出它们的和与差。

【实训 1】编写一个 shell 脚本，在屏幕上输出操作系统的系统信息，包括计算机名，Linux 发布版本，Linux 内核版本和当前 IP 地址。

关键代码：

```
#!/bin/sh
echo Computer Name:
hostname
echo Kernel Version:
uname –a
echo IP Address:
ifconfig
```

【实训 2】假设当前计算机用户根目录下存在目录 Documents，要求用户编写一个 bash 脚本对该目录进行备份。压缩为 Linux 系统中常见的 tar.gz 格式。

关键代码：

```
#!/bin/sh
if [ -d ~/Documents ]
then
    bakdate=$(date +%Y%d%H%M)
    tar –csvPf Documents_${bakdate}.tar.gz    Documents/
else
    echo “Directory not exit!”
fi
```

【实训 3】现有目录 A 和目录 B，目录中不包含子目录，要求用户编写一个 bash 脚本，比较两个目录内文件的差异。

关键代码：

```
#!/bin/sh
if [ $# != 2 ]
then
    echo “Parameters invalid!”
exit 0
fi

if [ -d $1 && -d $2 ]
then
```

```
        diff –c –a $1 $2
    else
        echo “Directory not exit!”
    fi
```

说明：在学完上述的知识点之后，对于上面的题目大家应该有能力自主完成了，后面的三个实训，我给出了具体的代码，希望大家在此基础上，加以拓展，让你的 shell 脚本完成更大的功能。下面，我给出一个具有一定综合性的例子供大家参考。

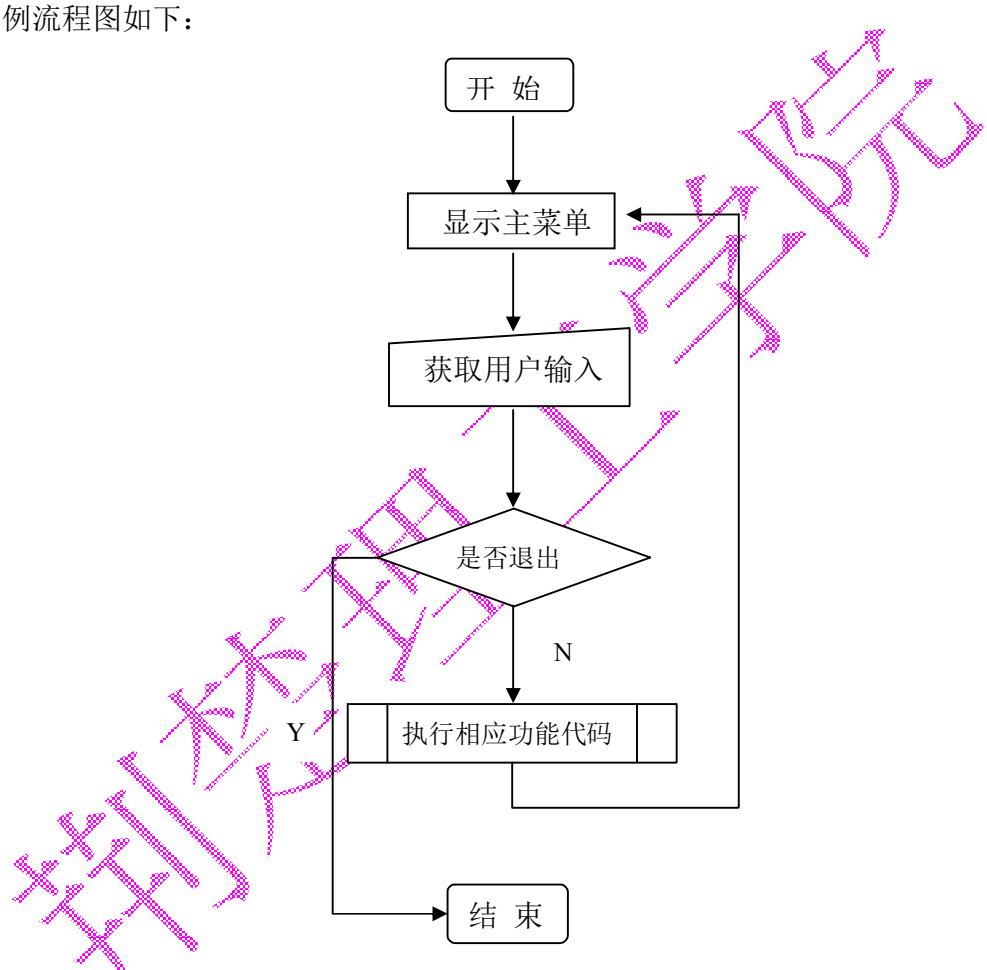
综合实例

1. 需求分析

用户在进行 Linux 系统管理的过程中，经常需要用到查看进程的信息、用户的信息等常用的功能。本例针对这一需要，使用 shell 编程实现了基本的系统管理功能。通过本程序，可以按照要求实现查看进程信息、查看当前登录系统的用户等功能。程序主界面的颜色可以修改，也可以显示帮助信息。

2. 系统设计

实例流程图如下：



3. 程序代码

根据上述流程图，程序可以分为几个模块：界面修改模块，主界面模块，查看进程信息模块，查看用户信息模块和帮助模块，各模块封装成函数予以实现。

```
#  界面修改模块
color()                              #定义函数 color，根据输入颜色信息修改界面颜色
{
    case $1 in                       #调用 case 判断函数参数 1 的值
        black_green)                 #黑底绿字
        echo -e "\033[40;32m"
```

```
                        #调用 echo,通过向显示器输出控制字符达到特殊效果
        ;;              #case 局部结束符
         black_yellow)  #黑底黄字
         echo -e "\033[40;33m"
         ;;
        white_blcak)            #白底黑字
         echo -e "\033[40;37m"
         ;;
        black_white)            #黑底白字
         echo -e "\033[40;36m"
         ;;
        black_blue)             #黑底蓝字
         echo -e "\033[40;34m"
          ;;
    esac                        #case 结束
}

#  主界面模块
echo -e "\033[2J"              #清屏
trap "" 1 2 3                  #设置信号处理
mday=`date +%d/%m/%y`   #日期信息
mhost=`hostname`               #机器名信息
mwho=`whoami`                  #当前用户信心
while :                        #循环显示主菜单
do
cat <<mmenu
    ------------------------------------------------------
        $mwho              $mhost              $mday
    ------------------------------------------------------
                1: 改变字体颜色
                2: 查看进程信息
                3: 查看用户信息
                h: 帮助
                q: 退出
    ------------------------------------------------------
mmenu
echo -e -n "\t  输入您的选择[1,2,3,h,q]: "
read Cho
case $Cho in
    1)
    while :
    do
    cat <<kcol
    ------------------------------------------------------
```

```
        $mwho  $mhost  $mday
    ------------------------------------------------------------
        1:黑绿   2:黑黄   3:白黑   4:黑白   5:黑蓝   0:返回
    ------------------------------------------------------------
kcol
    echo -e -n "\t 请输入选择的颜色[1,2,3,4,5]: "
    read choice
        if [ "$choice" = "1" ]
        then
            color black_green
        elif [ "$choice" = "2" ]
        then
            color black_yellow
        elif [ "$choice" = “3” ]
        then
            color white_black
        elif [ "$choice" = "4" ]
        then
            color black_white
        elif [ "$choice" = "5" ]
        then
            color blace_blue
        elif [ "$choice" = "0" ]
        then
            break
        else
            echo -e "\033[2J"
            continue
        fi
    clear
done
;;

# 查看进程信息模块
2)
ps aux|sort -rn|head -10
;;

# 查看用户信息模块
3)
who
;;
```

```
# 帮助模块
H|h)
cat <<mmenu
      ------------------------------------------------------------
            选择改变颜色可以修改当前界面的字体颜色
            选择查看进程信息可以获取当前占用资源最高的前 10 个进程
            选择查看用户可以获取当前登录系统的用户信息
      ------------------------------------------------------------
mmenu
      ;;
Q|q)
exit 0
;;
*)
echo -e "\033[2J";
continue;
;;
esac
echo -e -n "\t  按任意键继续"
read J
clear
done
```

说明：

1. 脚本程序已经测试，完全通过，只是里面的颜色部分有点小问题，大家可以查一查，在 shell 中颜色的数值码是多少，由于五一回家电脑没有网，所以希望大家完善这个代码。
2. 在编写程序时，不要在里面刻意的加一些空格或者缩进，否则会报错。由于 shell 不比 C 语言那么自由，里面的空格，缩进都有讲究，希望大家认真仔细。
3. 代码来源于资料，仅供参考。

声明

本资料来源于网上及自己总结，仅限于交流学习使用，如有疑问，请联系作者 QQ:1028150787。由于学习 shell 的时间不长，对于 shell 没有加深的掌握，只是学了这些表面的知识。但是，对于我们这些学 linux 嵌入式的同学来说，掌握这些知识，足够了。大家可以参照着这看 Linux 内核或者 U-BOOT 里面的 Makefile，会发现，很多东西都很简单，只是大家的知识面太窄了！这些资料对 shell 脚本编程进行了一个比较全面的介绍，全篇分为 15 个小节，涵盖了 shell 脚本编程几乎所有的知识。附有接近 40 个 shell 脚本源代码，是前面知识点的举例，而且没有给出运行结果，意在让大家可以亲自实践一下，写出自己的 shell 脚本。还是那句老话，想要掌握 shell 编程，还得多学多练。希望大家都能掌握 shell 编程。希望大家共同进步，学好嵌入式，找个好工作！由于水平有限，文中难免有遗漏和不足，恳请大家提出宝贵意见，相互交流，共同学习！