

1、shell 简介

- 1、shell 本意是壳，可以理解为人机交互的一层
- 2、shell 是一类语言，就是脚本语言，而不是一种语言
- 3、shell 语言有：sh、bash、csh、ksh、perl、python 等
- 4、shell 在嵌入式中一般用来做配置
- 5、linux 中最常用的 shell 是 bash
- 6、shell 脚本的运行机制：解释运行

- 1、C/C++ 是编写程序后需要先编译再运行。脚本语言不需要编译，可以直接运行。
- 2、所谓解释运行，执行程序同时。解释器也在逐行解释 shell 代码。一行一行执行。
- 3、解释运行：运行到这行时才开始解释运行。C 语言是提前编译链接好。
- 7、shell 脚本也可以理解为把命令行中敲的东西写到一起。

比如在屏幕打印 hello world:

- 1、在命令行中输入：echo "helloworld"
- 2、编写一个脚本程序，在脚本程序中添加代码：echo "helloworld"，然后执行脚本两者的效果是一样的。

2、shell 的编写和运行

shell 的编写和运行:

- 1、在 windows 下的编辑器换行符是“\r\n”，而 linux 下的换行符是“\n”会导致该脚本无法运行，所以需要使用 linux 下的文办编辑器来编写 shell

- 2、shell 的三种运行方式:

- 1、当做可执行程序使用:

需要先修改权限为可执行

再./xx.sh

eg:

chmod a+x a.sh

./a.sh

- 2、使用 source 命令:

直接 source a.sh 就可以，不需要修改可执行权限

eg:

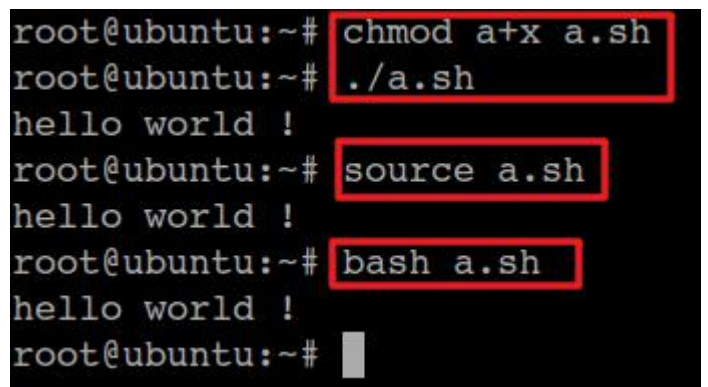
source a.sh

- 3、使用 bash 脚本解释器来实行

直接 bash a.sh 即可不需要修改可执行权限

eg:

bash a.sh



```
root@ubuntu:~# chmod a+x a.sh
root@ubuntu:~# ./a.sh
hello world !
root@ubuntu:~# source a.sh
hello world !
root@ubuntu:~# bash a.sh
hello world !
root@ubuntu:~#
```

3、第一行

```
1 #!/bin/sh
2
3 echo "hsaddas"
```

#!/bin/sh

shell 第一行的意思是解释器的目录是/bin/sh

```
lrwxrwxrwx 1 root root 4 3月 25 2014 /bin/sh -> dash
```

也可以使用

#!/bin/bash

```
-rwxr-xr-x 1 root root 920788 4月 3 2012 /bin/bash
```

(1)shell 程序的第一行一般都是： `#!/bin/sh` 这句话以`#!`开始，后面加上一个 `pathname`，这句话的意思就是指定 `shell` 程序执行时被哪个解释器解释执行。所以我们这里写上`/bin/sh` 意思就是这个 `shell` 将来被当前机器中`/bin` 目录下的 `sh` 可执行程序执行。

可以将第一行写为：`#!/bin/bash` 来指定使用 `bash` 执行该脚本。

注意：在 `ubuntu` 上面默认使用的解释器 `sh` 其实不是 `bash`，而是 `dash`。`dash` 是 `ubuntu` 中默认使用的脚本解释器。

(2)脚本中的注释使用`#`，`#`开头的行是注释行。如果有多行需要注释，每行前面都要加`#`。（`#`就相当于 C 语言中的`//`）

(3)shell 程序的正文，由很多行 `shell` 语句构成。

4、变量的定义、引号

变量的定义、赋值、引用

```
string="hello world"
```

```
#!/bin/sh
```

```
string="hello world" #初始化
```

```
string="new hello" #赋值
```

```
echo $string #引用，使用$符号引用定义好的变量
```

1、shell 是弱类型语言。对变量类型没有特别明显的区分

2、赋值时=两边不能有空格

3、变量引用使用`$`符号，`$`后面的变量如果没有被定义，执行时并不会被报错。而是将其内容解析为空。

4、引用的方式有两种

```
echo $string
```

```
echo ${string}
```

推荐下面的这种，比如

如果不加花括号，后面在引用时无法识别出 `str`，而是识别出 `strxxxxxxxxxx`

```

1 #!/bin/sh
2
3 str="ttttttttttttt"
4 string3="aaaaaaaa$strxxxxxxxxxxxxxx"
5 echo ${string3}

```

认为 strxxxxxxxxxxxxxx 没有定义，输出为空

```

aaaaaaaa

```

加了花括号就可以识别出

```

1 #!/bin/sh
2
3 str="asdaad"
4 string3="aaaaaaaa${str}xxxxxxxxxxxxxx"
5 echo ${string3}

```

输出

```

aaaaaaaaatttttttttttttxxxxxxxxxxxxxx

```

无引号、单引号、双引号

三者主要区别在于当字符串中有转义字符时的处理：

\$ ' " \

无引号：

遇到转义字符输出不可控（会按照该字符的脚本含义解析，不会当做普通字符）

单引号：

单引号不认识什么单引号之外的转义字符，是个瞎子，

双引号：

不能直接输出转义字符，会按照该字符的脚本含义解析，不会当做普通字符。需要在转义字符前加反斜杠 \

总结：

- 1、单引号内的没有转义字符（除了单引号。一切皆普通字符），输入什么，输出就是什么，其他两种情况中遇到转义字符会直接执行字符的脚本含义。但是可以通过在转义字符前面加反斜杠\来取消脚本含义，当做普通字符处理。
- 2、单引号中不能把转义字符‘单引号’输出，因为会与前面的单引号先结合。

规范使用：

变量引用加花括号

字符变量赋值加双引号，转义字符用反斜杠

```
str="ttttttttttttt"  
string3="aaaaaaaa${str}xxxxxxxxxxxxx"  
echo ${string3}
```

5、shell 中调用 linux 命令 典型的两种情况

1、直接调用命令

```
#!/bin/sh  
mkdir dir  
cd dir  
touch 1.c  
cd ..
```

2、使用命令的返回值：使用反引号括起来``

或者\$()

```
#!/bin/sh  
  
echo "MYPATH=`pwd` "  
echo "MYPATH=$(pwd)"
```

执行结果

```
root@ubuntu:/mnt/hgfs/share/uboot_study/2、shell# source a.sh  
MYPATH=/mnt/hgfs/share/uboot_study/2、shell
```

6、shell 中的分支结构 1：if

if 语句

```
if [条件]  
then  
    xxx  
else  
    yyy
```

```
fi
```

判断条件及举例

1、判断文件是否存在 -f

```
#!/bin/sh
```

```
if [ -f 1.c ]
then
    echo "yes"
else
    echo "no"
    touch 1.c
fi
```

2、判断目录是否存在 -d

```
#!/bin/sh
```

```
aa=dir
if [ -d $aa ]
then
    echo "yes"
else
    echo "no"
    mkdir $aa
    touch $aa/2.c
fi
```

3、判断字符串是否相等

```
#!/bin/sh
```

```
str1="abc"
str2="abc"
if [ ${str1} = ${str2} ]
then
    echo "yes"
else
    echo "no"
fi
```

注意是一个等号

4、判断字符串是否为空 -z

```
#!/bin/sh
```

```
str1="abc"
```

```

if [ -z ${str1} ]
then
    echo "yes"
else
    echo "no"
fi

```

注意：当被判断变量是未定义的时，shell 不认为他是空的

```
#!/bin/sh
```

```

str1="abc"
if [ -z ${str2} ]
then
    echo "yes"
else
    echo "no"
fi

```

结果会输出 no

5、数字之间大小判断

-eq equal 相等

-gt gteater than 大于

-lt less than 小于

-ge greater equal 大于等于

-le less qeual 小于等于

```
#!/bin/sh
```

```

if [ 20 -eq 21 ]
then
    echo "equal"
else
    echo "not equal"
fi

```

6、用 -o 来连接判断条件中的或运算(-o 相当于 C 语言中的 |)

```
#!/bin/sh
```

```

str1="abc"
str2="abc"
a=1
b=2
if [ $a -eq $b -o $str1 = $tr2 ]
then
    echo "yes"
else

```

```
    echo "no"
fi
```

相当于

```
if( (16==16) | ("abc"=="abc") )
```

7、shell 中 || 和 && 的巧妙使用，if 的简写

格式：

[A] || B

[A] && B

A 是判断条件，要符合 if 语句中[]内的书写格式

B 是执行的语句

```
#!/bin/sh
```

```
str1=""
```

```
[ -z $str1 ] || echo "not kong"
```

```
[ -z $str1 ] && echo "kong"
```

1、||

当||前面不成立，就会执行后面

2、&&

当&&前面成立，就会执行后面

7、shell 中的分支结构 2：循环结构

for 循环举例

```
#!/bin/sh
```

```
#打印 1 到 5
```

```
for i in 1 2 3 4 5
```

```
do
```

```
    echo $i
```

```
done
```

```
#打印当前目录下的所有文件名
```

```
for i in `ls`
```

```
do
```

```
    echo $i
```

```
done
```

```
#打印 1 到 100
```

```
for i in {1..100}      #1 和 100 之间是两个点
```

```
do
```

```
    echo $i
```

done

while 循环

```
#!/bin/sh

#打印 1 到 100
i=1
j=100
while [ $i -le $j ]
do
    echo $i
    i=$((i+1)) #i++
done

#打印 1 到 100 所有奇数
i=1
j=100
while [ $i -le $j ]
do
    echo $i
    i=$((i+2)) #i+=2
done
```

8、echo 重定向

向文件中添加内容（更改文件内容） >

echo "#include <stdio.h>">new.c

将 new.c 中所有内容删除。修改为#include <stdio.h>

注意：

是全部替换，危险性较高

```
root@ubuntu:/mnt/hgfs/share/uboot_study/2、shell# ls -l mkv210
-rwxrwxrwx 1 root root 2627 8月 2 15:54 mkv210_image.c
```

```
root@ubuntu:/mnt/hgfs/share/uboot_study/2、shell# echo "aa">mkv210_image.c
```

大小变小

```
root@ubuntu:/mnt/hgfs/share/uboot_study/2、shell# ls -l mkv210
-rwxrwxrwx 1 root root 3 8月 2 15:55 mkv210_image.c
```


向文件中追加内容 >>

echo "int main(void)">>new.c

在 new.c 中的最后追加内容 int main(void)

```
root@ubuntu:/mnt/hgfs/share/uboot_study/2、 shell# echo "aa">>
root@ubuntu:/mnt/hgfs/share/uboot_study/2、 shell#
root@ubuntu:/mnt/hgfs/share/uboot_study/2、 shell#
root@ubuntu:/mnt/hgfs/share/uboot_study/2、 shell# ls -l mkv21
-rwxrwxrwx 1 root root 6 8月 2 15:57 mkv210_image.c
```

9、case 语句

举例：

```
#!/bin/sh

val=5
case $val in
1) echo "1";;
2) echo "2";;
esac
```

注意：

- 1、shell 中的 case 语句没有 break，默认带 break。条件匹配会自动跳出
- 2、case 中的 break 是用来跳出外层的循环的。
- 3、两个分号的含义，第一个人好用来表示 echo "1"写完。第二个分号用来表示 case 匹配成功后的全部操作完成，当 case 匹配成功有多个操作时：见下面代码

```
#!/bin/sh

val=5
case $val in
1) echo "1";touch 1.c;echo "aa";;
2) echo "2";;
esac
```

10、shell 的调用传参

- 1、\$#表示参数的个数（从脚本名字的下一个开始数，脚本名字不算）
- 2、\$0、\$1、\$2...表示第 0 个、第 1 个、第 2 个参数（从脚本名字开始数，\$0 是脚本名字）

argv.sh:

```
#!/bin/sh
```

```
echo $#  
echo $0  
echo $1  
echo $2  
echo $3
```

在中断中执行 source argv.h aa bb cc

```
root@ubuntu:/mnt/hgfs/share/uboot_study/2、shell# source argv  
3  
bash  
aa  
bb  
cc
```

注意：

- 1、参数的个数是从脚本名字后面开始数的，脚本名字是不算的，区别于 C 语言的调用传参
- 2、\$0 表示的是脚本解释器的名字，从\$1 开始才是真正的参数
- 3、与 C 语言对比

./a.out aa bb cc argc=4, argv[0]=a.out, argv[1]=aa, argv[2]=bb, argv[3]=cc

./argv.sh aa bb cc \$#=3, \$0=bash, \$1=aa, \$2=bb, \$3=cc

shift 的使用

shift 使用一次，表示将参数左移一次（第一个参数消失。第二个参数变为第一个，第三个变为第二个）

例子：

argv.sh

```
#!/bin/sh
```

```
echo $#  
echo $0  
echo $1  
echo $2  
echo $3  
shift  
echo $#  
echo $0  
echo $1  
echo $2  
echo $3
```

在中断中执行 source argv.h aa bb cc

```

root@ubuntu:/mnt/hgfs/share/uboot_study/2、shell# source argv
3
bash
aa
bb
cc
2
bash
bb
cc

```

可以看到 `shift` 一次后，参数个数-1，参数全部（除了\$0 脚本文件名）左移一个位置

代码解读（视频中看到的 `while` 和 `case` 嵌套使用）

```

.....
while [ $# -gt 0 ]
do
    case "$1" in
        --) shift; break;;
        -a) shift; APPEND=yes;;
        -n) shift; BORAD_NAME="${1%_config}"; shift;;
        *) break;;
    esac
done
.....

```

解释：

- 1、`break` 是用来跳出外面的 `while` 循环的。`case` 不需要 `break`，自带
- 2、看起来像个死循环，其实几乎每个 `case` 匹配后都有一个 `shift` 左移一次，每遇到一次 `shift`，`$#` 的值就会减少 1。
- 3、每 `shift` 一次，下一次的 `$1` 的值就会改变一次。`$#` 的值就会减少 1。这个 `while` 循环的功能相当于一道自动前进的流水线、不断把下一个产品送上来检验。

11、退出 exit

`exit 0`

表示正常运行结束并退出程序

`exit 1`

表示非正常运行导致退出程序