# Linux Shell简明教程（一）

**神奇的Linux**

Linux有多神奇，我就不说了。作为在一个IT界混的码农，或多或少的都会接触Linux，不管你是怎么接触Linux，对于其中的一些命令肯定也会或多或少的了解一些。Linux虽然不是你的必须装备，但是装备上了Linux，对于你的职业生涯肯定是有百利而无一害的。基于此，我接触了Linux，然后就学习了一下Shell脚本，以此文章，作为Shell脚本学习中的一个笔记，以备后期翻阅。

**Shell能干什么**

学习一项知识，凭的是兴趣，这是一个理由；另一个理由则是为了工作。好吧，我高尚一回，我觉对不是为了生计而去学习Shell脚本，纯粹是为了兴趣，弥补我大学时期没有学习Shell脚本而留下来的遗憾。而更多的人学习Shell并不是和我一样，更多的是为了工作，解决工作中的实际问题而决定要看看Shell脚本的，那么Shell脚本在工作中到底能干什么呢？先看看以下问题：

1. 你是否经常需要处理一些重复性的工作？
2. 你是否经常发现使用高级语言，如C++去处理一些事情发现大材小用？
3. 你工作中是否经常需要进行一些批量处理的任务？
4. ……

如果对于上面的问题，如果你回答“YES”。那你就非常有必要去学习一下Linux Shell脚本。好了，想必你已经决定了去看看Shell了。来吧。

**一段简单的Shell**

首先来一段简单的Shell脚本，整体上对Shell有一个简单的认识。

```
#!/bin/bash

cd ~ # 切换到用户的主目录，也就是/home/xxx
mkdir shell_test # 创建一个名为shell_test的文件夹

for ((i=0; i<10; i++)) # Shell的for循环
do
    touch test_$i.txt #创建空文件，文件名为test_0.txt, test_1.txt ...
done
```

上面是一段简单的Shell程序，实现在功能为：

1. 在用户主目录下创建一个shell_test文件夹；
2. 在用户主目录下创建10个规定命名格式的空文件。

由于是第一个Shell程序，所以有必要逐行解释一下，留下一个好的印象。

- 第一行：指定脚本解释器，这里是用/bin/bash做解释器；
- 第二行：切换到当前用户的home目录；
- 第三行：创建一个名为shell_test的空目录；
- 第四、五行：for循环，一共循环10次；
- 第六行：创建10个名称为test*0.txt, test*1.txt格式的空文件。

`cd`、`mkdir`和`touch`都是系统自带的程序，一般在`/bin`目录下；而`for`、`do`和`done`都是Shell的关键字。 Shell中使用`#`开头的行就是注释。
一个简单的程序之后，我们就开始进入Shell的语法学习吧。

**变量**

从变量开始学习这段旅程。

**基础知识**
1. 在Shell中，使用变量之前不需要事先声明，只是通过使用它们来创建它们；
2. 在默认情况下，所有变量都被看做是字符串，并以字符串来存储；
3. Shell变量是区分大小写的；
4. 在Shell中，通过在变量名前加一个`$`符号来访问它的内容，无论何时想要获取变量内容，都必须在它前面加一个`$`符号；

下面通过一段Shell脚本来详细的说明上面的内容：

```
#!/bin/bash

var1=10 #定义一个变量var1

#定义一个变量var2，对于字符串中有空格的情况，需要使用引号将字符串括起来，
#此外等号两边不能有空格
var2="Jelly Think"
```

```
echo $var1 #获取变量var1的值
echo $var2 #获取变量var2的值
```

**使用read**

在Shell中，我们可以使用`read`命令将用户的输入赋值给一个变量。这个命令需要一个参数，即准备读入用户输入数据的变量名，然后它会等待用户输入数据。通常情况下，在用户按下回车键时，read命令结束。例如以下代码：

```
#!/bin/bash

read var1 #读入用户输入的var1变量的值
echo $var1 #输出var1的值

exit 0
```

**引号的使用技巧**

在上面的代码中也说了，如果字符串中包含了空格，就需要使用引号将字符串括起来，而这只是引号的一个简单的使用。 像$var1这种的变量在引号中的行为取决于你所使用的引号类型。这句话有以下两层意思：

1. 如果把一个$变量表达式放在双引号中，程序执行到这一行时就会把变量替换为它的值；
2. 如果把一个$变量表达式放在单引号中，就不会发生替换现象；

但是，我们可以通过在$字符前面加上一个`\`字符取消它的特殊含义，也就是经常说的转义字符。下面通过一段简单的代码来理解上面内容的意思：

```
#!/bin/bash

var1="JellyThink"

echo "This is $var1" #输出:This is JellyThink
echo 'This is $var1' #输出:This is $var1
echo "This is \$var1" #输出:This is $var1

exit 0
```

**环境变量**

当一个Shell脚本程序开始执行时，一些变量根据环境设置中的值进行初始化，一般比较常用的有以下几个：

- $HOME:当前用户的主目录，例如：`/home/jelly`；
- $PATH:以冒号分隔的用来搜索命令的目录列表；
- $0:Shell脚本的名字；
- $#:传递给脚本的参数个数；
- $$:Shell脚本的进程号，脚本程序通常会用它来生成一个唯一的临时文件。

**参数变量**

如果脚本程序在调用时带有参数，一些额外的变量就会被创建。即使没有传递任何参数，环境变量$#也依然存在，只不过它的值是0罢了。例如以下实力代码：

```
#!/bin/bash

#参数的命名是$1、$2.....
#$0表示脚本的名字

#输出参数个数
echo "参数个数为：$#"
echo "脚本名字：$0"
echo "第一个参数：$1"
echo "第二个参数：$2"

exit 0
```

输出如下：

```
参数个数为：2
脚本名字：./argsDemo.sh
第一个参数：Jelly
第二个参数：Think
```

**条件判断**

我们知道，所有程序设计语言的基础是对条件进行测试判断，并根据测试结果采取不同行动的能力。在总结之前，先看看Shell脚本程序里可以使用的条件结构，然后再来看看使用这些条件的控制结构。

一个Shell脚本能够对任何可以从命令行上调用的命令的退出码进行判断，这其中也包括我们写的Shell脚本程序。这就是为什么要在所有自己编写的脚本程序的结尾包括一条返回值的`exit`命令的重要原因。

**test或[命令**

在实际工作中，大多数脚本程序都会广泛使用Shell的布尔判断命令`[`或`test`。在一些系统上，这两个命令其实是一样的，只是为了增强可读性，当使用`[`命令时，我们还使用符号`]`来结尾。写了这么多程序了，还是头一次见，将一个`[`作为一个命令。算我孤陋寡闻了。下面就通过一个简单的例子来看看`test`和`[`是如何使用的。

```
#!/bin/bash

if test -f testDemo.sh
then
    echo "testDemo.sh exists."
fi

if [ -f test.c ]
then
    echo "test.c exists."
else
    echo "I cannot find exists."
fi

exit 0
```

我们通过使用`-f`来判断指定文件是否存在。`test`命令可以使用的条件类型可以归为3类：

- 字符串比较；
- 算术比较；
- 文件相关的条件测试。

下面就分别看看。

| 字符串比较 | 结果 |
| --- | --- |
| string1 = string2 | 如果两个字符串相同，结果就为真 |
| string1 != string2 | 如果两个字符串不同，结果就为真 |
| -n string | 如果字符串不为空，则结果为真 |
| -z string | 如果字符串为一个空串（null），则结果为真 |

| 算术比较 | 结果 |
| --- | --- |
| expression1 -eq expression2 | 如果两个表达式相等，则结果为真 |
| expression1 -ne expression2 | 如果两个表达式不等，则结果为真 |
| expression1 -gt expression2 | 如果expression1大于expression2，则为真 |
| expression1 -ge expression2 | 如果expression1大于等于expression2,则为真 |
| expression1 -lt expression2 | 如果expression1小于expression2，则为真 |
| expression1 -le expression2 | 如果expression1小于等于expression2，则为真 |
| !expression | 表达式为假，则结果就为真；反之亦然 |

| 文件条件测试 | 结果 |
| --- | --- |
| -d file | 如果文件是一个目录，则为真 |
| -f file | 如果文件是一个普通文件，则为真；也可以用来测试文件是否存在 |
| -r file | 如果文件可读，则结果为真 |
| -s file | 如果文件大小不为0，则结果为真 |
| -w file | 如果文件可写，则结果为真 |
| -x file | 如果文件可执行，则结果为真 |

**控制结构**

Shell也有一组控制结构，它们与其它程序语言中的控制接口很相似，下面就分开来总结它们。

**if语句**

结构如下：

```
if condition
then
    statements
else
    statements
fi
```

代码示例：

```
#!/bin/bash

var1=Jelly #这个是赋值，不能有空格分隔
var2=Think

if [ $var1 = $var2 ] #‘=’号两边都需要空格分隔
then
    echo "$var1 = $var2"
else
    echo "$var1 != $var2"
fi

exit 0
```

**elif语句**

结构如下

```
if condition
then
    statements
elif condition
then
    statements
else
    statements
fi
```

一个很容易出错的问题。例如以下代码：

```
#!/bin/bash

echo "Is it morning? please answer yes or no."
read timeofday

if [ $timeofday = "yes" ]
then
    echo "Good morning."
elif [ $timeofday = "no" ]
then
    echo "Good afternoon."
else
    echo "Sorry, $timeofday not recognized. Enter yes or no."
    exit 1
fi

exit 0
```

上面这段代码很简单，运行以下，如果你不输入`yes`或`no`，就会运行出错，得到以下提示信息：

```
./elifDemo1.sh: line 6: [: =: unary operator expected
./elifDemo1.sh: line 9: [: =: unary operator expected
```

这是为何？代码中有`if [ $timeofday = "yes" ]`，当我不输入任何内容时，这个`if`语句就会变成这样`if [ = "yes" ]`，很明显，这不是一个合法的条件。为了避免出现这种情况，我们必须给变量加上引号，改成这样`if [ "$timeofday" = "yes" ]`。这样就没有问题了。

**for语句**

结构如下：

```
for variable in values
do
    statements
done
```

代码示例：

```
#!/bin/bash

for foo in Jelly Think Website
do
    echo $foo
done

exit 0
```

在文章的开头，那段简单的代码中，也包含了`for`的简单使用。

**while语句**

结构如下：

```
while condition
do
    statements
done
```

代码示例：

```
#!/bin/bash

echo "Enter password: "
read pwd

while [ "$pwd" != "root" ]
do
    echo "Sorry, try again."
    read pwd
done

exit 0
```

**until语句**

结构如下：

```
until condition
do
    statements
done
```

它与`while`循环很相似，只是把条件测试反过来了；也就是说，循环将反复执行直到条件为真。

**case语句**

结构如下：

```
case variable in
    pattern [ | pattern] ...) statements;;
    pattern [ | pattern] ...) statements;;
    ...
esac
```

`case`的代码结构相对来说是比较复杂的。`case`结构具备匹配多个模式，然后执行多条相关语句的能力，这使得它非常适合于处理用户的输入。下面通过一个实例来

看看case的具体使用。

```
#!/bin/bash

echo "Is it morning? Please answer yes or no."
read input

case "$input" in
    yes ) echo "Good Morning.";;
    no ) echo "Good Afternoon.";;
    y ) echo "Good Morning.";;
    n ) echo "Good Afternoon.";;
    * ) echo "Sorry, answer not recognized";;
esac

exit 0
```

当case语句被执行时，它会把变量input的内容与各字符串依次进行比较。一旦某个字符串与输入匹配成功，case命令就会执行紧随右括号)后面的代码，然后就结束。 在代码中，最后面的*表示匹配任何字符串，我们在写代码时，总是在其它匹配之后再添加一个*以确保如果没有字符串得到匹配，case语句也会执行某个默认动作。 由于case比较复杂，所以不得不再来一段代码，究其用法，如下：

```
#!/bin/bash

echo "Is it morning? Please answer yes or no."
read input

case "$input" in
    yes | y | Yes | YES ) echo "Good Morning.";;
    n* | N* ) echo "Good Afternoon.";;
    * ) echo "Sorry, answer not recognized.";;
esac

exit 0
```

上面这段代码，使用了|符号，也就是说，也就是合并了多个匹配模式；同时还使用了*通配符；没有问题，上面的代码运行的很好。

**&&和||操作符**

Shell中也支持&&和||符号，和C++语言中的是一样；比如：

```
statement1 && statement2 && statement3 && ...
```

从左开始顺序执行每条命令，如果一条命令返回的是true，它右边的下一条命令才能执行。如果此持续直到有一条命令返回false，或者列表中的所有命令都执行完毕；遵循“短路”规则。

```
statement1 || statement2 || statement3 || ...
```

从左开始顺序执行每条命令，如果一条命令返回的是false，它右边的下一条命令才能够被执行。如此持续直到有一条命令返回true，或者列表中的所有命令都执行完毕。

**语句块**

在Shell中也有语句块，使用{}来定义一个语句块。我们可以把多条语句放到一个语句块中执行。

**函数**

函数，这么NB的东西，Shell怎么能少呢。定义函数的结构如下：

```
function_name()
{
    statements
}
```

代码示例：

```
#!/bin/bash

foo #运行到这里，还没有定义foo函数，会报错

foo() {
```

```
    echo "Function foo is executing."
}

echo "script starting"
foo #输出"Function foo is executing."
echo "script ended"

exit 0
```

脚本程序从自己的顶部开始执行，当它遇到了`foo() {`结构时，它知道脚本正在定义一个名为`foo`的函数。当执行到单独的`foo`时，Shell就知道应该去执行刚才定义的函数了。函数执行完毕以后，脚本接着`foo`后的代码继续执行。 当一个函数被调用时，脚本程序的位置参数（`$*`、`$@`、`$#`、`$1`、`$2`等）会被替换为函数的参数，这也是我们读取传递给函数的参数的方法。当函数执行完毕后，这些参数会恢复为它们先前的值。 我们可以使用`local`关键字在Shell函数中声明局部变量，局部变量将仅在函数的作用范围内有效。此外，函数可以访问全局作用范围内的其它Shell变量。如果一个局部变量和一个全局变量的名字相同，前者就会覆盖后者，但仅限于函数的作用范围之内，例如以下代码：

```
#!/bin/bash

sample_text="global variable"

foo()
{
    local sample_text="local variable"
    echo "Function foo is executing..."
    echo $sample_text
}

echo "script starting..."
echo $sample_text

foo

echo "script ended..."
echo $sample_text

exit 0
```

输出如下：

```
script starting...
global variable
Function foo is executing...
local variable
script ended...
global variable
```

**总结**

终于总结完第一部分了，还有第二部分，在下一篇博文中将主要总结Shell中经常使用的一些命令。这篇对Shell中的一些基础进行总结的博文就到此了，下一篇再见。

2014年10月28日 于深圳。

===修改日志===

2015年7月15日 修改if then…elif then的错误。