



Linux shell编程

——shell基本功能

信息工程学院

◆ 为什么学SHELL?

- **Linux**运维工程师在进行服务器集群管理时，需要编写**Shell**程序来进行服务器管理，日常运维工作常常是重复的，让机器搞定一切；
- 对于**JavaEE**和**Python**程序员来说，工作的需要常常需要编写一些**Shell**脚本，比如编写一个定时备份数据库的脚本；
- 对于大数据程序员来说，需要编写**Shell**程序来管理集群。

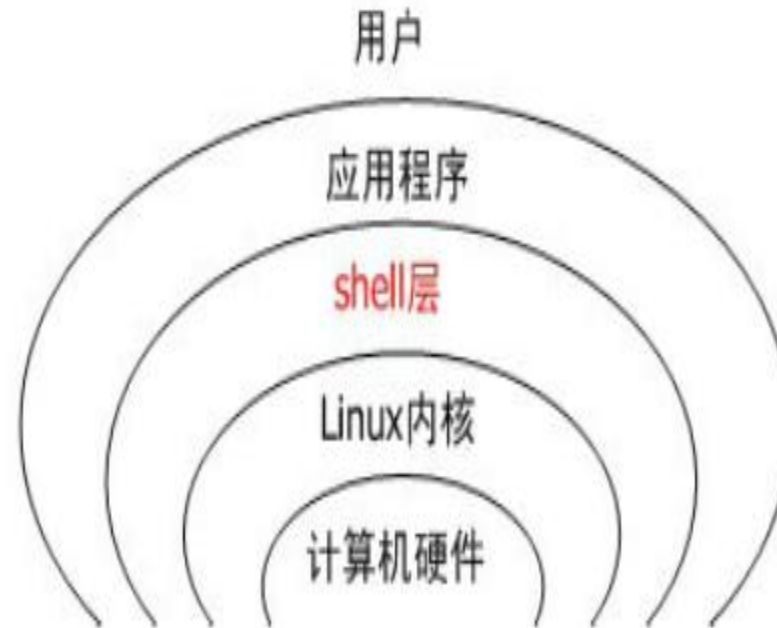
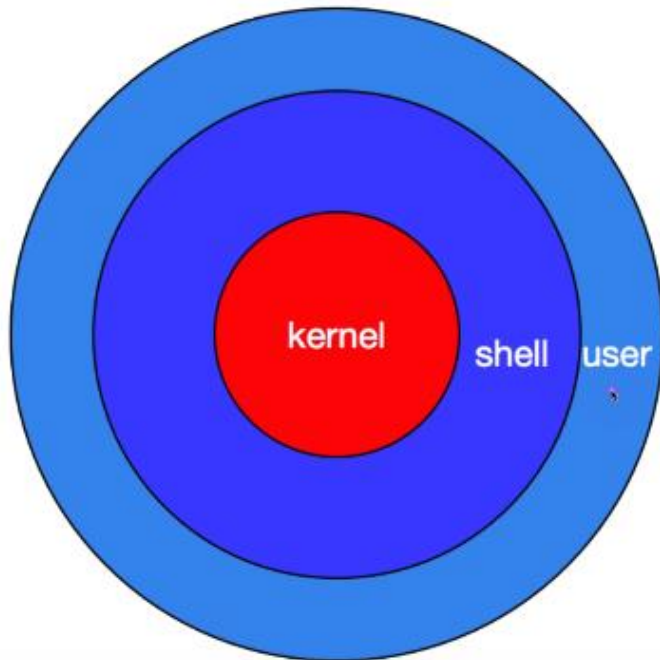
◆ SHELL脚本学习进阶

- 能看**SHELL**脚本、能改**SHELL**脚本、
- 能自己写**SHELL**脚本、能优化**SHELL**脚本

什么是shell

★ **shell**就是围绕在**Kernel**的外层，与**user**通信的。

★ **Kernel**是管理硬件的，只能识别二进制语言；而用户是发出命令使用硬件的，不认识二进制，使用的是高级语言；语言不同，故要通过**shell**向使**kernel**与**user**通信。



- ◆ **SHELL**是一个用**C**语言编写的程序，也是一种命令语言，又是一种解释性编程语言，命令行解释器。
- ◆
 - 运行**LINUX**命令时，进入的**SHELL**界面，是**SHELL**作为命令解释程序出现，是**SHELL**最常见的使用方式。
 - 用户可以用**Shell**来启动、挂起、停止甚至是编写一些程序。
 - **SHELL**还是一种高级程序设计语言，它有变量，关键字，有各种控制语句，如**if**、**case**、**while**、**for**等语句，支持函数模块，有自己的语法结构。

Shell脚本和Shell编程

- ◆ 当命令不在命令行中执行，而是从一个文件中执行时，该文件就称为 **Shell 脚本**。
 - **Shell 脚本**是纯文本文件。
 - **Shell 脚本**通常以 **.sh** 作为后缀名，但不是必须。
 - **Shell 脚本**是以行为单位的，在执行脚本的时候会分解成一行一行依次执行。
- ◆ **Shell脚本的例子**

```
#myshell.sh  
#!/bin/bash  
echo "hello world!"
```

shell脚本的建立和执行

◆ shell脚本的建立

\$ vi myshell.sh

指明该脚本执行需要的命令解释器
特定情况可省略，不要省略



```
#myshell.sh
#!/bin/bash
echo "hello world!"
```

说明:

- ◆ **#!**后面给出的路径必须正确,否则报错 “**Command not found**” ;
- ◆ **#!**也可忽略，但这样脚本无法使用 **shell** 内建的指令;
- ◆ 行注释是每行的开始是**#**;
- ◆ 段注释的开始是“:**<<!**”，最后是“**!**”结束;

◆ shell脚本的执行方式一

(1) 先将**shell**脚本的权限设置为可执行，因为**shell**脚本文件是纯文本文件，是没有执行权限的。

```
[root@CM00 ~]# ll myshell.sh
-rw-r--r--. 1 root root 44 3月 28 17:31 myshell.sh
[root@CM00 ~]# chmod 744 myshell.sh
[root@CM00 ~]# ll myshell.sh
-rwxr--r--. 1 root root 44 3月 28 17:31 myshell.sh
```

(2) 然后可以用相对路径和绝对路径的方式运行。

```
[root@CM00 ~]# ./myshell.sh
hello world!
[root@CM00 ~]# /root/myshell.sh
hello world!
```

shell脚本的建立和执行

◆ shell脚本的执行方式二

以脚本名作为参数运行，不用赋予执行权限。

```
[root@CM00 ~]# sh myshell.sh  
hello world!
```

```
[root@CM00 ~]# sh ./myshell.sh  
hello world!
```

```
[root@CM00 shell]# sh ./myshell.sh  
sh: ./myshell.sh: 没有那个文件或目录
```

为什么？

调试脚本时的sh命令

sh -x 脚本名 //以调试模式执行脚本

该选项可以使用户跟踪脚本的执行，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 “+”。

```
#myshell.sh
#!/bin/bash
echo "hello world!"
a=1
echo "a=$a"
unset a
echo "a=$a"
```

```
[wlw@CM00 ~]$ sh ./myshell.sh
hello world!
a=1
a=
```

```
[wlw@CM00 ~]$ sh -x ./myshell.sh
+ echo 'hello world!'
hello world!
+ a=1
+ echo a=1
a=1
+ unset a
+ echo a=
a=
[wlw@CM00 ~]$
```

shell脚本中常用输出命令

➤ **echo** [选项] [string] //按指定要求显示字符串或变量值

- **-n** 输出后不换行
- **-e** 解释参数中的转义符，即如果**echo**命令带有选项“**-e**”，那么在其后的参数中可以有以下转义字符：**\a**响铃，**\n**换行，**\t**跳格，**\b**，**\c**，**\e**，**\f**，**\r**，**\v**，**\l**等
- **-E** 不解释参数中的转义符

```
#echostr.sh
#!/bin/bash
echo "Input nums1:\n"
echo "Input nums2:"
echo -n "Input nums3:"
echo "Input nums4:"
echo -e "\nInput nums5:"
echo
```

```
[wlw@CM00 ~]$ sh ./echostr.sh
Input nums1:\n
Input nums2:
Input nums3:Input nums4:

Input nums5:

[wlw@CM00 ~]$
```

特别说明：在命令行状态直接用**echo**命令显示的字符串时，
如果字符串最后有“！”，要在！后加空格，否则出错。

```
[wlw@CM00 ~]$ echo "11111"
11111
[wlw@CM00 ~]$ echo "11111!"
bash: !": event not found
[wlw@CM00 ~]$ echo "11111! "
11111!
```

查看历史命令 **history**

使用命令历史机制，可以方便地调用或修改以前的命令，或者把全部或部分先前的命令作为新命令予以快捷执行。

- 历史命令文件是`~/.bash_history`，由带序号的表格构成。
- 默认保存1000条，也可以在`/etc/profile`文件中更改**HISTSIZE=1000**值。

◆ 显示历史命令：**history [option] [arg...]**

- 不带任何参数，则**history**命令会显示历史命令的清单
- 给出一正整数，如**5**就只显示历史表中的最后**5**行命令

常用的选项有：

- **-w** 把当前缓存中的历史命令写到历史文件中，覆盖原有内容。
- **-c** 删除历史清单中所有的项。

执行历史命令！

◆ 执行历史命令：以字符“！”开头

格 式	意 义
上下箭头	调出历史命令
!!	重复上一条命令，也就是“!-1”
!n	重新执行第n条历史命令
!-n	重新执行倒数第n条历史命令。!-1就等于!!
!string	重新执行以字符串string开头的最近的历史命令行。
!?string?	重新执行最近的、包含字符串string的那条历史命令
!\$	重复上一条命令的最后一个参数

说明：“!”之后不能有空格类字符(空格、制表、换行、等号或开括号);
“!”之前不能有反斜线“\”。

- ◆ **# history 10**
(注: **history**和**10**中间有空格)
- ◆ **# !99**
(注: **!**和**99**中间不能有空格)
- ◆ **# !!**
- ◆ **# !ls**
- ◆ **# history | more**
- ◆ **# history -c**
- ◆ **# history -w**
- ◆ **# history 10** //列出最近**10**条记录
- ◆ **# !99** //执行历史清单的第**99**条命令
- ◆ **# !!**重复执行上一个命令.
- ◆ **# !ls**执行最后一次以**ls**开头的命令
(Shell从最后一条历史命令向前搜索,最先匹配的一条命令将会被执行).
- ◆ **# history | more**逐屏列出所有的历史记录.
- ◆ **# history -c**立即清空当前所有历史命令的记录.
- ◆ **# history -w** 将缓存中的历史命令写入历史命令清单文件中

TAB键补全功能

命令及文件名补全功能

- ◆ 可以输入目录名或文件名的开头部分，然后按**Tab**键，**Linux**根据输入的字母查找以这些字母开头的目录或文件，并自动补全剩余的部分。
 - 如果系统可以唯一确定哪个目录或文件，则自动补全相应名称。
 - 如果找到不止一个文件，如果系统不能唯一确定，它会把文件名补全到相同部分的最后一个字符，然后响铃提示，要求用户进一步输入名字中后面的字符。
 - 如果输入过程中，不知道后面的字符，也，可以连续按两次**Tab**键代替，列出当前目录下所有可以匹配已输入字符的文件名或命令名。

alias别名命令

alias 别名=“原命令” //用新字符串替换原命令名，原命令保持不变。

unalias 别名 //取消命令别名

- ◆ 将别名写入环境变量配置文件~/.bashrc中可永久生效，用**source**命令使其立即生效。

```
[wlw@CM00 myshfile]$ grep root /etc/passwd  
root:x:0:0:root:/root:/bin/bash  
operator:x:11:0:operator:/root:/sbin/nologin
```

```
[wlw@CM00 myshfile]$ alias grep="grep --color=auto"
```

```
[wlw@CM00 myshfile]$ grep root /etc/passwd  
root:x:0:0:root:/root:/bin/bash  
operator:x:11:0:operator:/root:/sbin/nologin
```

- ◆ 别名的优先级比命令高。命令执行时的优先顺序是：
 - 1、执行用绝对路径或相对路径执行的命令
 - 2、执行别名
 - 3、执行**bash**内部命令。
 - 4、执行按照**\$PATH**定义的目录查找顺序找到的第一个命令。

shell的特殊字符——常用通配符

通配符	作用
?	匹配任意一个字符
*	匹配0个或任意多个字符
[字符组]	1) 匹配[]中任意一个字符。如f[abc]代表一定匹配fa或fb或fc。 2) 字符组可由-字符组成表示范围。如f[a-c]与f[abc]作用相同。
[!字符组] [^字符组]	逻辑非，表示匹配不是[]内的一个字符。如f[!1-9]表示以f开头其后一个字符不是数字1 ~ 9的字符串，即匹配fa,fb等。

说明：

- 1、*代表任何字符串，但要注意，文件名前的圆点（.）和路径名中的斜线（/）必须显示匹配。例如：*不能匹配.file，而.*才可以匹配.file
- 2、-在[]内是通配符表示范围，若在[]外面就成为普通字符了；
*和?在[]外面是通配符，若在[]之内，则成为普通的字符。

例如“-a[*?]abc”只有一对方括号是通配符，-、*和?均为普通字符，因此，它匹配的字符串只能是-a*abc和-a?abc。

shell的特殊字符

```
# cp -v /etc/i* /tmp
```

```
# cp -v /etc/*.conf /tmp
```

```
# cp -v /etc/i*.conf /tmp
```

```
# cp -v /etc/i?.conf /tmp
```

```
# cp -v /etc/i[abcd]*.conf /tmp
```

```
# cp -v /etc/[^0-9][0-9] /tmp
```

```
# cp -v /etc/[^0-9]*[0-9] /tmp/b
```

```
# cp -v /etc/[^a-zA-Z][a-zA-Z]* /tmp/c
```

- ◆复制 **/etc** 下所有以 **i** 开头的文件
- ◆复制 **/etc** 下所有以 **.conf** 结尾的文件
- ◆复制 **/etc** 下以 **i** 开头,以 **.conf** 结尾的文件
- ◆复制 **/etc** 下以 **i** 开头,以 **.conf** 结尾,中间包含一个任意字符的文件
- ◆复制 **/etc** 下以 **i** 开头,以 **.conf** 结尾,中间包含以 **a、b、c、d** 中任意一个字母开头的文件
- ◆复制 **/ect** 下以一个非数字字符和一个数字组合命名的文件
- ◆复制 **/etc** 下以任意一个非数字开头,以数字结尾的文件至 **/tmp/b** 中
- ◆复制 **/etc** 下以非字母开头,后面跟了一个字母,及任意长度的字符文件至 **/tmp/c** 中

shell的特殊字符

◆ 转义符“\”和 注释符“#”

“\x”表示使用x符号的字面意义而不是特殊意义。

```
[ wlw@CM00 ~]$ echo aa
aa
[ wlw@CM00 ~]$ echo #aa
[ wlw@CM00 ~]$ echo \#aa
#aa
```

—

shell的特殊字符——引号

1. 单引号

- 由单引号括起来的字符都作为普通字符出现。即使用转义字符也失去作用，单引号不能嵌套。

2. 双引号

- 由双引号括起来的字符除\$、反引号（`）和反斜线（\）外均作为普通字符对待。故需使用转义符使用这几个特殊字符的本身含义，如“\\$”、“\\”、“\`”。

```
[wlw@CM00 ~]$ echo aa
aa
[wlw@CM00 ~]$ echo '\aa'
\aa
[wlw@CM00 ~]$ echo "\aa"
\aa
```

shell的特殊字符——引号

3. 反引号（倒引号）

- 反引号括起来的字符串表示命令替换。**shell**会先执行该命令，并以它的标准输出结果取代整个反引号部分。
- 反引号还可以嵌套使用。但应注意，嵌套使用时内层的反引号必须用反斜线（\）将其转义。
- 反引号`command`与\$(command)作用相同，为避免混肴，建议使用后者。

```
[wlw@CM00 ~]$ echo pwd
pwd
[wlw@CM00 ~]$ echo `pwd`
/home/wlw
```

```
[wlw@CM00 ~]$ echo `pwd`
/home/wlw
[wlw@CM00 ~]$ echo $(pwd)
/home/wlw
```

shell的特殊字符

◆ 引号举例

例: `$ cat exam9`

<code>echo "current directory is `pwd`"</code>	#倒引号表示命令替换
<code>echo "home directory is \$HOME"</code>	#以HOME的值代替\$HOME
<code>echo "file*.*"</code>	#其中字符均为普通字符
<code>echo "directory '\$HOME'"</code>	#单引号仍为普通字符
<code>echo "Filename is No\\$*"</code>	#其中有转义字符

`$ . exam9`

`current directory is /home/mengqc/dir1`

`home directory is /home/mengqc`

`file*.*`

`directory '/home/mengqc'`

`Filename is No$\*`

引号的区别举例

例：若 `[root@localhost ~]$ string=hello`，以下命令运行结果如何？

- `[root@localhost ~]$ echo $string`
- `[root@localhost ~]$ echo '$string'`
- `[root@localhost ~]$ echo "$string"`
- `[root@localhost ~]$ echo \"string\"`
- `[root@localhost ~]$ echo \"$string\"`
- `[root@localhost ~]$ echo "\\string\"`

总结：对于变量的引用要用双引号

运行结果：

- `hello`
- `$string`
- `hello`
- `\string`
- `$string`
- `\hello`

引号的区别举例

例：以下命令运行结果如何？

- `[root@localhost ~]$ echo pwd`
- `[root@localhost ~]$ echo 'pwd'`
- `[root@localhost ~]$ echo `pwd``
- `[root@localhost ~]$ echo $pwd`
- `[root@localhost ~]$ echo $(pwd)`

总结：当要将执行命令的结果作为变量的值或参数时才需要用反引号

运行结果：

- `pwd`
- `pwd`
- `/root`
- (无显示)
- `/root`

()和{ }——使用较少

对一串的命令执行可以用()或{ }

- ◆{ }形式 如: `$ { echo "hello"; cat /etc/passwd ; } | more`
- ◆()形式 如: `$ (echo "hello";cat /etc/passwd) | more`

区别:

- 都是把一串的命令放在括号内, 且命令之间用;号隔开。
- 括号内某个命令的重定向只影响该命令, 但括号外的重定向则影响到括号里的所有命令。
- ()最后一个命令可以不用分号, { }最后一个命令要用分号。
- ()内各命令不必和括号有空格, { }的第一个命令和左括号之间必须要有一个空格。
- 二者执行过程相同, 但存在的重要区别:
 - { }成组命令只在本shell内执行命令表, 不产生新的进程;
 - ()成组命令在新的子shell内执行, 要建立新的子进程。

()和{ }的区别举例

◆ \$ var=test

\$ (var=notest; echo \$var) #变量var的值在新进程子shell中有效

notest

\$ echo \$var #父shell不受子shell影响，值为test

test

◆ \$ var=test

\$ { var=notest; echo \$var;} #变量var的值在当前shell中，未建新进程

notest

\$ echo \$var #父shell受子shell的影响，值变为notest

notest

()和{ }的区别举例

```
$ var=test
```

◆ \$ { var1=test1;var2=test2;echo \$var1>a;echo \$var2;}

#输出test1被重定向到文件a中，

```
test2
```

#而test2输出则仍输出到标准输出中。

```
$ cat a
```

```
test1
```

◆ \$ { var1=test1;var2=test2;echo \$var1;echo \$var2;}>a

#括号内命令的标准输出全部被重定向到文件a中

```
$ cat a
```

```
test1
```

```
test2
```