Linux高级编程（二）

张勇涛

1

SHELL程序设计

2

Shell的历史

- sh（Bourne Shell）：由Steve Bourne开发，各种UNIX系统都配有sh。
- csh（C Shell）：由Bill Joy开发，随BSD UNIX发布，它的流程控制语句很像C语言，支持很多Bourne Shell所不支持的功能：作业控制，命令历史，命令行编辑。
- ksh（Korn Shell）：由David Korn开发，向后兼容sh的功能，并且添加了csh引入的新功能，是目前很多UNIX系统标准配置的Shell，在这些系统上/bin/sh往往是指向/bin/ksh的符号链接。
- tcsh（TENEX C Shell）：是csh的增强版本，引入了命令补全等功能，在FreeBSD、Mac OS X等系统上替代了csh。
- bash（Bourne Again Shell）：由GNU开发的Shell，主要目标是与POSIX标准保持一致，同时兼顾对sh的兼容，bash从csh和ksh借鉴了很多功能，是各种Linux发行版标准配置的Shell，在Linux系统上/bin/sh往往是指向/bin/bash的符号链接。虽然如此，bash和sh还是有很多不同的，一方面，bash扩展了一些命令和参数，另一方面，bash并不完全和sh兼容，有些行为并不一致，所以bash需要模拟sh的行为：当我们通过sh这个程序名启动bash时，bash可以假装自己是sh，不认扩展的命令，并且行为与sh保持一致。

Shell的作用

- 自动化管理的重要依据
- 追踪与管理系统的重要工作
- 简单的入侵检测功能
- 连续命令单一化
- 简单的数据处理
- 跨平台与缩短学习历程

shell的执行方式

- Shell的作用是解释执行用户的命令，用户输入一条命令，Shell就解释执行一条，这种方式称为交互式（Interactive）
- Shell的另一种执行命令的方式称为批处理（Batch），用户事先写一个Shell脚本（Script），其中有很多条命令，让Shell一次把这些命令执行完，而不必一条一条地敲命令。

Shell如何执行命令

- **执行交互式命令**

  用户在命令行输入命令后，一般情况下Shell会fork并exec该命令，但是Shell的内建命令例外，执行内建命令相当于调用Shell进程中的一个函数，并不创建新的进程。以前学过的cd、alias、exit等命令即是内建命令，凡是用which命令查不到程序文件所在位置的命令都是内建命令

- **执行脚本**

执行脚本

1. 首先编写一个简单的脚本，保存为script.sh

```
#! /bin/sh
cd ..
ls
```

2. $ chmod +x script.sh

3. $ ./script.sh
或$bash script.sh

- Shell脚本中用#表示注释，相当于C语言的//注释。但如果#位于第一行开头，并且是#!（称为Shebang）则例外，它表示该脚本使用后面指定的解释器/bin/sh解释执行。

分析执行过程

1.交互Shell（bash）fork/exec一个子Shell（sh）用于执行脚本，父进程bash等待子进程sh终止。

2.sh读取脚本中的cd ..命令，调用相应的函数执行内建命令，改变当前工作目录为上一级目录。

3.sh读取脚本中的ls命令，fork/exec这个程序，列出当前工作目录下的文件，sh等待ls终止。

4.ls终止后，sh继续执行，读到脚本文件末尾，sh终止。

5.sh终止后，bash继续执行，打印提示符等待用户输入。

SHELL的基本语法

变量

按照惯例，Shell变量由全大写字母加下划线组成

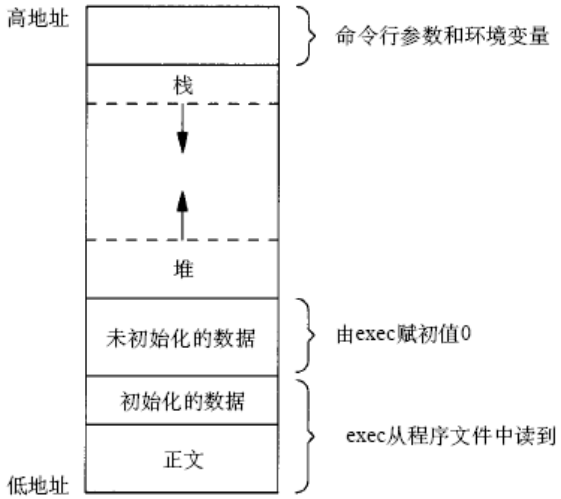
有两种类型的Shell变量：

环境变量: 环境变量可以从父进程传给子进程，因此Shell进程的环境变量可以从当前Shell进程传给fork出来的子进程。用printenv命令可以显示当前Shell进程的环境变量。

本地变量: 只存在于当前Shell进程，用set命令可以显示当前Shell进程中定义的所有变量（包括本地变量和环境变量）和函数。

环境变量是任何进程都有的概念，而本地变量是Shell特有的概念。

环境变量



环境变量

命令行参数argv类似，环境变量表也是一组字符串

```
#include <stdio.h>
extern char **environ;

int main(void)
{
        int i;
        for(i=0; environ[i]!=NULL; i++)
          printf("%s\n", environ[i]);

        return 0;
}
```

历史上UNIX支持main函数带有三个参数:

int main(int argc, char *argv[], char *envp[])

定义环境变量

在Shell中，**环境变量**和**本地变量**的定义和用法相似。

在Shell中定义或赋值一个变量：

$ VARNAME=value

注意等号两边都不能有空格，否则会被Shell解释成命令和命令行参数。

一个变量定义后仅存在于当前Shell进程，它是本地变量

export命令可以把本地变量导出为环境变量

定义和导出环境变量通常可以一步完成：

$ export VARNAME=value

取消环境变量或本地变量

- 用unset命令可以删除已定义的环境变量或本地变量。
- $ unset VARNAME

变量的使用

- 如果一个变量叫做**VARNAME**，用**${VARNAME}**可以表示它的值，在不引起歧义的情况下也可以用**$VARNAME**表示它的值。

- **注意**: 在定义变量时不用$，取变量值时要用$。

  和C语言不同的是，Shell变量不需要明确定义类型，事实上Shell变量的值都是字符串，比如我们定义VAR=45，其实VAR的值是字符串45而非整数。

- Shell变量不需要先定义后使用，如果对一个没有定义的变量取值，则值为空字符串。

文件名代换（Globbing）：*？[]

用于匹配的字符称为通配符（Wildcard），具体如下

*	匹配0个或多个任意字符
?	匹配一个任意字符
[若干字符]	匹配方括号中任意一个字符的一次出现

```
$ ls /dev/ttyS*
$ ls ch0?.doc
$ ls ch0[0-2].doc
$ ls ch[012][0-9].doc
```

注意： Globbing所匹配的文件名是由Shell展开的，也就是说在参数还没传给程序之前已经展开了，比如上述ls ch0[012].doc命令，如果当前目录下有ch00.doc和ch02.doc，则传给ls命令的参数实际上是这两个文件名，而不是一个匹配字符串。

命令代换：`或$()

- 由**反引号**括起来的也是一条命令，Shell先执行该命令，然后将输出结果立刻代换到当前命令行中。

- 例如定义一个变量存放date命令的输出：

```
$ DATE=`date`

$ echo $DATE命令代换也可以用$()表示：
$ DATE=$(date)
```


算术代换：$(())

- 用于算术计算，$(())中的Shell变量取值将转换成整数

例如：

```
$ VAR=45
$ echo $(($VAR+3))
```

$(())中只能用+-*/和()运算符，并且只能做整数运算。

转义字符\

- 和C语言类似，\在Shell中被用作转义字符，用于去除紧跟其后的单个字符的特殊意义（回车除外），换句话说，紧跟其后的字符取字面值。

例如：

```
$ echo $SHELL
/bin/bash
$ echo \$SHELL
$SHELL
$ echo \\
```

\比如创建一个文件名为“$ $”的文件可以这样：

```
$ touch \$\ \$
```


用\来续行

- \还有一种用法，在\后敲回车表示续行，Shell并不会立刻执行命令，而是把光标移到下一行，给出一个续行提示符>，等待用户继续输入，最后把所有的续行接到一起当作一个命令执行。

例如：

```
$ ls \
> -l  (ls -l命令的输出)
```

单引号

- 和C语言不一样，Shell脚本中的单引号和双引号一样都是字符串的界定符（双引号下一节介绍），而不是字符的界定符。
- 单引号用于保持引号内所有字符的字面值，即使引号内的\和回车也不例外，但是字符串中不能出现单引号。如果引号没有配对就输入回车，Shell会给出续行提示符，要求用户把引号配上对。

例如：

```
$ echo '$SHELL'
$SHELL
$ echo 'ABC\（回车）
> DE'（再按一次回车结束命令）
ABC\
DE
```


双引号

- 双引号用于保持引号内所有字符的字面值（回车也不例外）

但以下情况除外：

$加变量名可以取变量的值
反引号仍表示命令替换
\$表示$的字面值
\`表示`的字面值
\”表示”的字面值
\\表示\的字面值
除以上情况之外，在其它字符前面的\无特殊含义，只表示字面值

例子

```
$ echo "$SHELL"
/bin/bash

$ echo "`date`"
Sun Apr 20 11:22:06 CEST 2003

$ echo "I'd say: \"Go for it\""
I'd say: "Go for it "

$ echo "\"（回车）
>"（再按一次回车结束命令）
"

$ echo "\\"
\
```


SHELL脚本语法

条件测试：test [

- 命令test或[可以测试一个条件是否成立，如果测试结果为真，则该命令的Exit Status为0，如果测试结果为假，则命令的Exit Status为1（注意与C语言的逻辑表示正好相反）。

- 例如测试两个数的大小关系：

```
$ VAR=2
$ test $VAR  –gt  1
$ echo $?
0

$ test $VAR  -gt  3
$ echo $?
1

$ [  $VAR  -gt  3  ]
$ echo $?
1
```


测试命令

[-d DIR]	如果DIR存在并且是一个目录则为真
[-f FILE]	如果FILE存在且是一个普通文件则为真
[-z STRING]	如果STRING的长度为零则为真
[-n STRING]	如果STRING的长度非零则为真
[STRING1 = STRING2]	如果两个字符串相同则为真
[STRING1 != STRING2]	如果字符串不相同则为真
[ARG1 OP ARG2]	ARG1和ARG2应该是整数或者取值为整数的变量，OP是-eq（等于）-ne（不等于）-lt（小于）-le（小于等于）-gt（大于）-ge（大于等于）之中的一个

带与、或、非的测试命令

[! EXPR]	EXPR可以是上表中的任意一种测试条件，!表示逻辑反
[EXPR1 -a EXPR2]	EXPR1和EXPR2可以是上表中的任意一种测试条件，-a表示逻辑与
[EXPR1 -o EXPR2]	EXPR1和EXPR2可以是上表中的任意一种测试条件，-o表示逻辑或

```
$ VAR=abc
$ [ -d Desktop -a $VAR = 'abc‘ ]
$ echo $?
0
```


warning:

- 注意，如果上例中的$VAR变量事先没有定义，则被Shell展开为空字符串，会造成测试条件的语法错误（展开为[-d Desktop -a = 'abc']），作为一种好的Shell编程习惯，应该总是把变量取值放在双引号之中（展开为[-d Desktop -a "" = 'abc']）：

```
$ unset VAR
$ [ -d Desktop –a $ VAR =  'abc' ]
bash: [: too many arguments
$ [ -d Desktop –a  "$VAR" = 'abc' ]
$ echo $?
1
```

if/then/elif/else/fi

- 和C语言类似，在Shell中用if、then、elif、else、fi这几条命令实现分支控制。

```
if [ -f ~/.bashrc ]; then
   . ~/.bashrc
fi
```

备注: source FileName
作用:在当前bash环境下读取并执行FileName中的命令。
注：该命令通常用命令“.”来替代。

- 其实是三条命令，if [-f ~/.bashrc]是第一条，then . ~/.bashrc是第二条，fi是第三条。如果两条命令写在同一行则需要用;号隔开，一行只写一条命令就不需要写;号了，另外，then后面有换行，但这条命令没写完，Shell会自动续行，把下一行接在then后面当作一条命令处理。和[命令一样，要注意命令和各参数之间必须用空格隔开。

if/then/elif/else/fi

if命令的参数组成一条子命令，如果该子命令的Exit Status为0（表示真），则执行then后面的子命令，如果Exit Status非0（表示假），则执行elif、else或者fi后面的子命令。
if后面的子命令通常是测试命令，但也可以是其它命令。
Shell脚本没有{}括号，所以用fi表示if语句块的结束。

```
#! /bin/sh

if [ -f /bin/bash ]
then echo "/bin/bash is a file"
else  echo "/bin/bash is NOT a file"
fi

if :; then echo "always true"; fi
```

:是一个特殊的命令，称为空命令，该命令不做任何事，但Exit Status总是真。
此外，也可以执行/bin/true或/bin/false得到真或假的Exit Status。

例子:if.sh

```
#! /bin/sh

echo "Is it morning? Please answer yes or no."
read  YES_OR_NO
if [ "$YES_OR_NO" = "yes" ]; then
  echo "Good morning!"
elif [ "$YES_OR_NO" = "no" ]; then
  echo "Good afternoon!"
else
  echo "Sorry, $YES_OR_NO not recognized. Enter yes or no."
  exit 1
fi
exit 0
```


case/esac

- case命令可类比C语言的switch/case语句，esac表示case语句块的结束。
- C语言的case只能匹配整型或字符型常量表达式，而Shell脚本的case可以匹配字符串和Wildcard，每个匹配分支可以有若干条命令，末尾必须以;;结束，执行时找到第一个匹配的分支并执行相应的命令，然后直接跳到esac之后，不需要像C语言一样用break跳出。

case.sh

```
#! /bin/sh

echo "Is it morning? Please answer yes or no."
read YES_OR_NO
case "$YES_OR_NO" in
yes|y|Yes|YES)
  echo "Good Morning!";;
[nN]*)
  echo "Good Afternoon!";;
*)
  echo "Sorry, $YES_OR_NO not recognized. Enter yes or no."
  exit 1;;
esac
exit 0
```


case的实用案例

- 使用case语句的例子可以在系统服务的脚本目录/etc/init.d中找到
- 例如:/etc/init.d/nfs-kernel-server
- case $1 in
- $1是一个特殊变量，在执行脚本时自动取值为第一个命令行参数，也就是start，所以进入start)分支执行相关的命令。同理，命令行参数指定为stop、reload或restart可以进入其它分支执行停止服务、重新加载配置文件或重新启动服务的相关命令。

for/do/done

- Shell脚本的for循环结构和C语言很不一样，它类似于某些编程语言的foreach循环

```
#! /bin/sh

for FRUIT in apple banana pear; do
  echo "I like $FRUIT"
done
```

- FRUIT是一个循环变量，第一次循环$FRUIT的取值是apple，第二次取值是banana，第三次取值是pear。

应 用

- 要将当前目录下的chap0、chap1、chap2等文件名改为chap0~、chap1~、chap2~等（按惯例，末尾有~字符的文件名表示临时文件）
- 这个命令可以这样写：
- $ for FILENAME in chap?; do mv $FILENAME $FILENAME~; done

- 也可以这样写：
- $ for FILENAME in `ls chap?`; do mv $FILENAME $FILENAME~; done

while/do/done

while的用法和C语言类似
例子: while.sh

```
#! /bin/sh

echo "Enter password:"
read TRY
while [ "$TRY" != "secret" ]; do
  echo "Sorry, try again"
  read TRY
done
```


控制循环次数

- 下面的例子通过算术运算控制循环的次数:
- whilecount.sh

```
#! /bin/sh

COUNTER=1
while [ "$COUNTER" -lt 10 ]; do
  echo "Here we go again"
  COUNTER=$(($COUNTER+1))
done
```

练习

- 用while循环求1～100的和

常用的位置参数和特殊变量

$0	相当于C语言main函数的argv[0]
$1、$2...	这些称为位置参数（Positional Parameter），相当于C语言main函数的argv[1]、argv[2]...
$#	相当于C语言main函数的argc - 1，注意这里的#后面不表示注释
$@	表示参数列表"$1" "$2" ...，例如可以用在for循环中的in后面。
$?	上一条命令的Exit Status
$$	当前Shell的进程号

参数

- 位置参数可以用shift命令左移。比如shift 3表示原来的$4现在变成$1，原来的$5现在变成$2等等，原来的$1、$2、$3丢弃，$0不移动。不带参数的shift命令相当于shift 1。例如 shift.sh：

```
#! /bin/sh

echo "The program $0 is now running"
echo "The first parameter is $1"
echo "The second parameter is $2"
echo "The parameter list is $@"
shift
echo "The first parameter is $1"
echo "The second parameter is $2"
echo "The parameter list is $@"
```


函数

- 和C语言类似，Shell中也有函数的概念，但是函数定义中没有返回值也没有参数列表。
- 例如 fun.sh：

```
#! /bin/sh

foo(){ echo "Function foo is called";}
echo "-=start=-"
foo
echo "-=end=- “
```

注意函数体的左花括号{和后面的命令之间必须有空格或换行，如果将最后一条命令和右花括号}写在同一行，命令末尾必须有;号。

函数的特点

- 在定义foo()函数时并不执行函数体中的命令，就像定义变量一样，只是给foo这个名字一个定义，到后面调用foo函数的时候（注意Shell中的函数调用不写括号）才执行函数体中的命令。Shell脚本中的函数必须先定义后调用，一般把函数定义都写在脚本的前面，把函数调用和其它命令写在脚本的最后（类似C语言中的main函数，这才是整个脚本实际开始执行命令的地方）。

- Shell函数没有参数列表并不表示不能传参数，事实上，函数就像是迷你脚本，调用函数时可以传任意个参数，在函数内同样是用$0、$1、$2等变量来提取参数，函数中的位置参数相当于函数的局部变量，改变这些变量并不会影响函数外面的$0、$1、$2等变量。函数中可以用return命令返回，如果return后面跟一个数字则表示函数的Exit Status。

函数例子

- 下面这个脚本可以一次创建多个目录，各目录名通过命令行参数传入，脚本逐个测试各目录是否存在，如果目录不存在，首先打印信息然后试着创建该目录。
- funtest.sh
- 注意is_directory()返回0表示真返回1表示假。

Shell脚本的调试方法

Shell提供了一些用于调试脚本的选项，如下所示：

- -n
  读一遍脚本中的命令但不执行，用于检查脚本中的语法错误
- -v
  一边执行脚本，一边将执行过的脚本命令打印到标准错误输出
- -x
  提供跟踪执行信息，将执行的每一条命令和结果依次打印出来

如何使用这些方法

- 一是在命令行提供参数
  $ sh -x ./script.sh
- 二是在脚本开头提供参数
  #! /bin/sh –x
- 第三种方法是在脚本中用set命令启用或禁用参数

```
#! /bin/sh
if [ -z "$1" ]; then
  set -x
  echo "ERROR: Insufficient Args."
  exit 1
  set +x
Fi
```

set -x和set +x分别表示启用和禁用-x参数，这样可以只对脚本中的某一段进行跟踪调试。

作业

- 需要完成一个程序，用户输入百分制的分数，之后返回“A”“B”“C”“D”“E”的等级。其中，A等级为90至100分，B等级为80至89分，C等级为70至79分，D等级为60至69分，E等级为0至59分。
- 输出用户通过命令行参数形式提供的多个整数的平均数。具体调用格式如下：
  - ✓$ ./avg.sh 12 34 56 78
  - ✓Average : 45
- 编写一个Shell程序test.sh，输入一个字符串，如果是目录，则显示目录下的信息，如为文件显示文件的内容。

联航精英训练营
UNIGRESS ELITE TRAINING CAMP

专业铸就品质 梦想成就未来

The Specialty Casts Quality,the Dream Gains Future.