SHELL 脚本编程

1 开场白

如果把 shell 命令比作盖房子的砖瓦，那 shell 脚本就是用一块块砖瓦建起来的房子了。我们可以通过一些约定的格式来将那些小巧的命令组合起来，实现更加自动化更加智能的所谓 shell 脚本。

所谓约定的格式，其实就是 shell 脚本的语法规则，就像 C 语言一样，将很多语句按照一定的规则组合起来形成一个程序。但是这里要强调的是，C 语言编写出来的程序是需要经过编译器编译，生成另一个称为 ELF 格式的文件之后才能执行的，但是 shell 脚本是不需要编译而可以直接执行的，这种脚本语言称为解释型语言。

废话少说，下面来各个击破。

2 脚本格式

要把 shell 命令放到一个“脚本”当中，有一个要求：脚本的第一行必须写成类似这样的格式：

```
#!/bin/bash
```

聪明如你一定立即明白，这是给系统指定一款 shell 解释器，来解释下面所出现的命令的。比如，你的第一个最简单的脚本 first_script.sh，也许是这样的：

```
vincent@ubuntu:~/ch01/1.3$ cat first_script.sh -n
     1  #!/bin/bash
     2
     3  echo "hello!"
```

这个脚本指定一款在/bin/下名字叫 bash 的 shell 解释器，来解释接下来的任何命令。如果你的系统用的是其他的解释器，就要将/bin/bash 改成相应的名字。

注意，脚本文件缺省是没有执行权限的，要使得脚本可以执行必须给他添加权限：

```
vincent@ubuntu:~/ch01/1.3$ chmod +x first_script.sh
vincent@ubuntu:~/ch01/1.3$ ./first_script.sh
echo “hello!”
hello!
vincent@ubuntu:~$
```

3 变量

shell 脚本是一种弱类型语言，在脚本当中使用变量不需要也无法指定变量的“类型”。缺省状态下，shell 脚本的变量都是字符串，即一连串的单词列表。下面将 shell 中关于变量的技术点各个击破：

1，变量的定义和赋值

```
myname=”Michael Jackson”
```

请严重注意：赋值号的两边没有空格！在 SHELL 脚本中，任何时候要给变量赋值，赋值号两边一定不能有空格。
另外，变量名也有类似于 C 语言那样的规定：只能包含英文字母和数字，且不能以数字开头。

2，变量的引用
使用变量时，需要在变量的前面加一个美元符号：$myname 这表示对变量的引用，比如：

```
vincent@ubuntu:~$ echo $myname
```

这样就把 myname 的值打印出来了。

3，变量的种类
SHELL 脚本中有这么几种变量：

A 普通的用户自定义变量，比如上面的 myname。
B 系统预定义好的环境变量，比如 PATH。
C 命令行变量，比如$#、$*等。

系统的环境变量可以通过如下命令来查看：

```
vincent@ubuntu:~$ env
......
PATH=/usr/lib/lightdm/lightdm:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
DESKTOP_SESSION=gnome-classic
PWD=/mnt/hgfs/codes/shell/struct
GNOME_KEYRING_PID=8070
LANG=en_US.UTF-8
MANDATORY_PATH=/usr/share/gconf/gnome-classic.mandatory.path
UBUNTU_MENUPROXY=libappmenu.so
GDMSESSION=gnome-classic
SHLVL=1
HOME=/home/vincent
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
LOGNAME=vincent
......
```

可以看到，系统中的环境变量有很多，每个环境变量都用大写字母表示，比如 PATH。每个环境变量都有一个值，就是等号右边的字符串。根据环境变量的不同，它们各自的含义不同。

设置环境变量：
以 PATH 环境变量为例子，如果想要将至修改为 dir/，只需执行以下命令：

```
vincent@ubuntu:~$ export PATH=dir/
```

当然，PATH 环境变量的作用是保存系统中可执行程序或脚本的所在路径，因此它的值都是一些以分号隔开的目录，我们经常的使用办法是：不改变其原有的值，而给它再增加一个我们自己需要设置的目录 dir/，因此更有用的命令可能是类似于以下这样的：

```
vincent@ubuntu:~$ export PATH=$PATH:dir/
```

还有，在命令行敲下如上的命令只会在当前的 shell 中临时有效，如果要永久有效，就必须将命令 export PATH=$PATH:dir/ 写入~/.bashrc 中。然后执行

vincent@ubuntu:~$ **source ~/.bashrc**

使之生效。

而 SHELL 脚本中的所谓命令行变量，指的是在脚本内部使用用户从命令行中传递进来的参数，例如：

vincent@ubuntu:~$ **./example.sh abcd 1234**

这里的脚本名叫 example.sh，咱们在执行他的时候顺便给了他两个参数，分别是 abcd 和 1234，要访问这两个参数以及相关的其他值，就必须使用命令行变量，以下是具体情况：（以命令./example.sh abcd 1234 为例子）

1，$# ：代表命令行参数个数，即 2
2，$* ：代表所有的参数，即 abcd 1234
3，$@ ：同上
4，$n ：第 n 个参数，比如$1 即 abcd，而$2 就是 1234

SHELL 脚本中还有几个跟命令行变量形式很类似的特殊变量，他们是：

1，$? ：代表最后一个命令执行之后的返回值
2，$$ ：代表当前 shell 的进程号 PID

（事实上以上变量前面的$是变量的引用符，他们的真正的名字是紧跟$后面的那个字符）

4 特殊符号们

SHELL 脚本有好几种特殊的符号，各自有各自的神通，他们分别是：引号、竖杠（管道）、和大于小于号（重定向），以下分别进行各个击破。

第一，引号。

引号有三种，他们是：双引号 “ ”、 单引号 ‘ ’、 反引号（抑音符） ` `

1，双引号的作用是将一些“单词”括起来形成单个的“值”。比如：

myname=“Michael Jackson”

在此变量的定义中如果没有双引号将会报错，因为这个字符串有两个单词，第二个单词会被认为是一个命令，但显然不对，因为 Jackson 不是命令而只是 myname 的一部分。

双引号所包含的内容还可以包括对变量的引用，比如：

fruit=apple
mytree=“$fruit tree”

由于对别的变量进行了引用，因此 mytree 的最终值是 apple tree

双引号所包含的内容还可以是一个命令，比如：

today=“today is `date`”

请注意：date 是一个 shell 命令，用来获得系统的当前时间，因为此命令出现在双引

号内部，默认情况下脚本会把它当做一个普通的单词而不是命令，要让脚本识别出该命令必须用一对反引号（` `）包含他。

2，单引号

以上对双引号的分析间接也澄清了单引号的作用了：如果一个字符串被单引号所包含，那么其内部的任何成分都将被视为普通的字符，而不是变量的引用或者命令，比如：

var='$myname, today: `date`'

这个变量 var 的值不会引用 myname 也不会执行 date。

3，反引号的作用就是在双引号中标识出命令。

编写以下脚本，可以立即理解这三个引号的区别：

```
#!/bin/bash
var=calender
echo "var: date"          # 直接打印出 var 和 date
echo "$var: `date`"       # 打印出变量 var 的值，以及命令 date 的执行结果
echo '$var: `date`'       # 打印出$var: `date`
```

第二：竖杠|（管道）。

SHELL 命令的一大优点是秉承了 UNIX/LINUX 的哲学：小而美。一个个小巧而精致的命令，各自完成各自的功能，不罗嗦，不繁杂。但有时候，我们需要他们相互协作，共同完成任务，就像采购负责买菜，回来交给洗涮工加工，再交给厨师烹饪，再交给服务员端给客人，我们常常需要将一个命令所达成的结果，给到另一个命令进行再加工，这时候就需要用到管道。例如：

vincent@ubuntu:~$ **ls -l | wc**

管道就像水管一样，将前面的命令的执行结果输送给后面的命令。ls -l 负责收集当前目录下的文件的信息，然后将这些文件名作为结果输送到管道，wc 这个命令接着从管道中把他们读取出来，并计算出行数、单词个数和总字符数。

管道不仅可以连接两个命令，也可以连接多个命令，类似于这样：

vincent@ubuntu:~$ **ccat /etc/passwd | awk -F=/ '{print $1}' | wc**

这就将三个命令连接起来，每个命令的输出都作为下一个命令的输入，连接起来就能完成强大的功能。

第三：大于号> 和小于号<（重定向）。

每一个进程在刚开始运行的时候，系统都会为他们默认地打开了三个文件，他们分别是标准输入、标准输出、标准出错，其文件描述符和对应设备关系如下图所示：

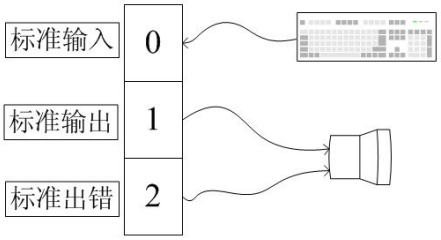

图1-36 缺省打开的三个设备文件

这三个标准文件对应两个硬件设备：标准输入是键盘，标准输出和标准出错是显示器（是的，显示器设备被打开了两次，第一次打开为行缓冲类型的标准输出，第二次打开为不缓冲类型的标准出错）。绝大多数的 SHELL 命令，默认的输入输出都是这三个文件。

比如，当我们执行命令 ls 的时候，他会默认地将结果打印到显示器上，就是因为 ls 本来就被设计为将结果往 1 号描述符（即标准输出，当执行成功的时候）或者 2 号描述符（即标准出错，当执行失败的时候）。而当我们打开普通文件的时候，系统也会帮我们产生一系列后续的数字（文件描述符）来表示这些文件，比如我们紧跟着打开了文件 a.txt 和 b.doc，

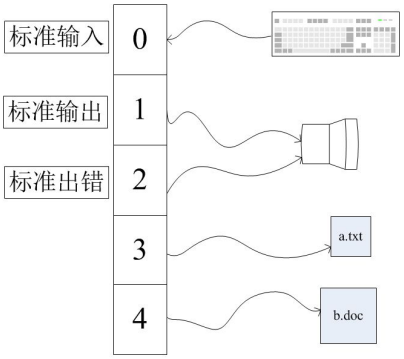

图 1-37 再打开两个普通文件之后

此时进程的的描述符情况如下：

第一：假如需要将 ls 命令的成功的输出结果（本来会被默认地输送到 1 号文件描述符的信息）重定向到 a.txt 文件中去，方法如下：

vincent@ubuntu:~$ **ls 1> a.txt**

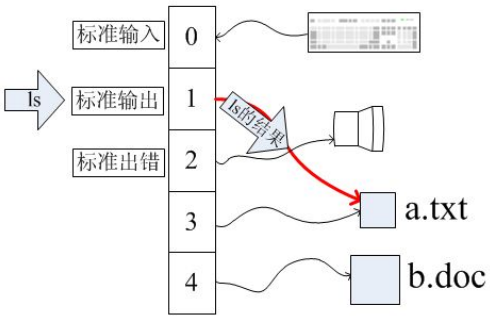

图 1-38 重定向 1 号文件描述符

第二：假如需要将 ls 命令的失败的输出结果（本来会被默认地输送到 2 号文件描述符的信

息）重定向到 a.txt 文件中去，方法如下：
vincent@ubuntu:~$ **ls notexist 2> a.txt** （notexist 是一个不存在的文件，所以 ls 命令执行会失败）

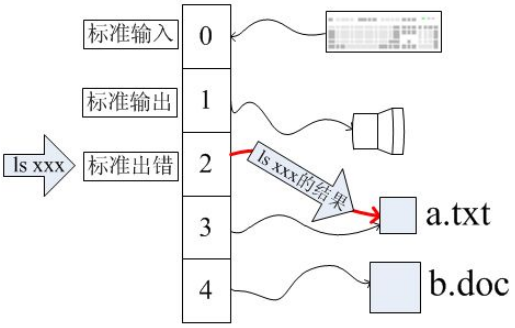

图1-39 重定向2号文件描述符

第三：重定向标准输入也是类似的，比如直接执行 echo 命令，他将会默认地从标准输入（即键盘）读取信息，然后打印出来。但是我们可以将标准输入重定向为 b.doc 文件：
vincent@ubuntu:~$ **echo 0< b.doc**

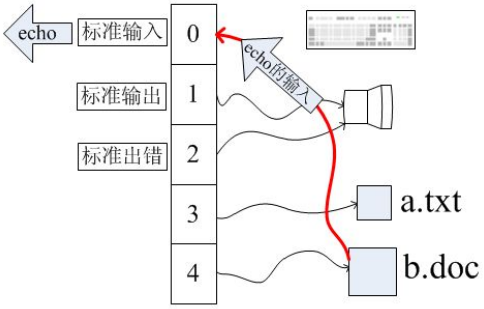

图1-40 重定向0号文件描述符

这样， echo 就将会从 b.doc 读取数据，而不是从键盘读取了。此处额外提一点，在 SHELL 脚本中，在重定向符的右边，标准输入输出设备文件描述符要写成&0、&1 和&2。比如要将一句话输出到标准出错设备中去，语句是：

echo “hello world” 1>&2

5 字符串处理

SHELL 中对字符串的处理，除了使用 1.2.3 节介绍的神器 sed 和 awk，在某些比较简单的场合，其实有更简便的办法。

1，计算一个字符串的字符个数：

vincent@ubuntu:~$ **var="apple tree"**
vincent@ubuntu:~$ **echo "${#var}"**
10

2，删除一个字符串左边部分字符：

vincent@ubuntu:~$ **path="/etc/rc0.d/K20openbsd-inetd"**

```
vincent@ubuntu:~$ level=${path#/etc/rc[0-9].d/[SK]}
vincent@ubuntu:~$ echo $level
20openbsd-inetd
```

3，删除一个字符串右边部分字符：

```
vincent@ubuntu:~$ path="/etc/rc0.d/K20openbsd-inetd"
vincent@ubuntu:~$ level=${path#/etc/rc[0-9].d/[SK]}
vincent@ubuntu:~$ level=${level%%[a-zA-Z]*}
vincent@ubuntu:~$ echo $level
20
```

注意：两个%%表示贪婪匹配，具体含义是：使用通配符[a-zA-Z]*从右向左“尽可能多地”匹配字符（贪婪原则）。如果只写一个%，则无贪婪原则，那么[a-zA-Z]*将按照最少原则匹配，即匹配 0 个字符（因为方括号星号*的含义是 0 个或多个字符）。这个道理对于删除左边字符的井号#也是适用的：双井号##代表从左到右的贪婪匹配。

6 测试语句

有一个叫 test 的命令，专门用来实现所谓的测试语句，测试语句可以测试很多不同的情形，比如：

```
vincent@ubuntu:~$ test -e file
```

以上语句用以判断文件 file 是否存在，如果存在返回 0，否则返回 1。那么除了可以判断一个文件存在与否，还有没有别的功能呢？晒出以下列表给各位看官鉴赏：

语句	含义	说明
test -e file	判断文件 file 是否存在	存在返回 0，否则返回 1
test -r file	判断文件 file 是否可读	可读返回 0，否则返回 1
test -w file	判断文件 file 是否可写	可写返回 0，否则返回 1
test -x file	判断文件 file 是否可执行	可执行返回 0，否则返回 1
test -d file	判断文件 file 是否是目录	是目录返回 0，否则返回 1
test -f file	判断文件 file 是否是普通文件	是普通文件返回 0，否则返回 1
test -s file	判断文件 file 是否非空	非空返回 0，否则返回 1
test s1 = s2	判断字符串 s1 和 s2 是否相同	相同返回 0，否则返回 1
test s1 != s2	判断字符串 s1 和 s2 是否不同	不同返回 0，否则返回 1
test s1 < s2	判断字符串 s1 是否小于 s2	s1 小于 s2 返回 0，否则返回 1
test s1 > s2	判断字符串 s1 是否大于 s2	s1 大于 s2 返回 0，否则返回 1
test -n s	判断字符串 s 长度是否为非 0	s 长度为非 0 返回 0，否则返回 1
test -z s	判断字符串 s 长度是否为 0	s 长度为 0 返回 0，否则返回 1
test n1 -eq n2	判断数值 n1 是否等于 n2	n1 等于 n2 返回 0，否则返回 1
test n1 -ne n2	判断数值 n1 是否不等于 n2	n1 不等于 n2 返回 0，否则返回 1
test n1 -gt n2	判断数值 n1 是否大于 n2	n1 大于 n2 返回 0，否则返回 1
test n1 -ge n2	判断数值 n1 是否大于等于 n2	n1 大于等于 n2 返回 0，否则返回 1
test n1 -lt n2	判断数值 n1 是否小于 n2	n1 小于 n2 返回 0，否则返回 1

test n1 -le n2	判断数值 n1 是否小于等于 n2	n1 小于等于 n2 返回 0，否则返回 1

图 1-41 test 语句

举个例子，假如要判断一个文件 file 是否存在，而且可读，如果都满足的话就将其显示在屏幕上，脚本可以写成这样：

```
#!/bin/bash
if test -e file && test -r file
then
    cat file
fi
```

以上代码中，我们依靠 test 语句来决定是否要执行 cat 命令，这是脚本语言中最简单的条件判断语句，根 C 语言的 if-else 结构很类似，只不过 C 语言的 if 语句跟着的是一对儿括号，看起来更加直观，其实，脚本也可以使用括号来代替 test 语句，看起来更顺眼：

```
#!/bin/bash
if [ -e file ] && [ -r file ]
then
    cat file
fi
```

语法上讲这两种写法完全等价，因此方括号[ ]其实就是 test 语句，但是从可读性来看，显然方括号的写法更容易理解，这里一定要特别注意的是：方括号的左右两边都必须有空格！

7 脚本语法单元

跟 C 语言很类似，SHELL 脚本也需要一套基本单元来控制整个逻辑的执行，包括所谓的控制流（就是常见的分支控制和循环控制）、函数、数值处理等。废话少说，以下各个击破。

分支控制

事实上前一小节的范例已经为我们展示了脚本中的分支控制语句，现将之摘抄下来：

```
1    if [ -e file ] && [ -r file ]
2    then
3        cat file
4    fi
```

代码中的分支语句的语法要点有这么几处：

1，每一个 if 语句都有一个 fi（即倒过来写的 if）作为结束标记。

2，分支结构中使用 then 作为起始语句。

3，当 if 语句后面的语句执行结果为真（即为 0）时，then 以下的语句才会被执行。

当然，if 语句还可以跟 else 配对使用，跟 C 语言类似，比如：

```
1    if [ -e file ] && [ -r file ]
2    then
3        cat file       # 如果文件存在且可读，则显示该文件内容
4    elif [ -e file ]
5    then
```

```
6       chmod u+r file
7       cat file        # 如果文件存在但不可读，则加了读权限之后再显示其内容
8   else
9       touch file      # 如果文件不存在，则创建该空文件
10  fi
```

注意：elif 而不是 else if，其后也要跟 if 一样紧随 then 语句。

如果是多路分支，可以使用 case 语句，这个类似于 C 语言中的 switch 语句。比如实现这么一个功能：要求用户输入一个数字，判断如果输入的是 1，则输出 one，如果输入的是 2，则输出 two，输入其他数字则输出 unkown，代码即注解如下：

```
1   read VAR        # 从键盘接收一个用户输入
2   case $VAR in    # 判断用户输入的值$VAR
3       1) echo "one" # 如果$VAR 的值为 1，则显示 one
4          ;;       # 每个分支都必须以双分号作为结束（最后一个分支除外）
5       2) echo "two"
6          ;;
7       *) echo "unknown"    # 星号*是 shell 中的通配符，代表任意字符。
8   esac
```

注意：变量 VAR 的值实际上是字符串，因此上述代码中的 1) 也可写成 "1")
整个 case 结构必须 esac 作为结束。

循环控制

几乎与 C 语言一样，SHELL 脚本中有 3 种可用的循环结构，他们分别是 while 循环、until 循环和 for 循环，其中 for 循环用法比较特殊。下面一一讲解。

先来看 while 循环和 until 循环，假设现在要实现打印 1 到 100 的功能，分别用这两种语句实现，代码和注解如下：

第一，while 循环语句：

```
1   declare -i n=0      # 在定义变量 n 前面加上 declare -i 表示该变量为数值
2   while [ $n -le 100 ] # 如果 n 的值小于等于 100，则循环
3   do                  # 循环体用 do 和 done 包含起来
4       echo "$n"
5       n=$n+1          # 使 n 的值加 1
6   done
```

第二，until 循环语句：

```
1   declare -i n=0
2   until [ $n -gt 100 ] # 如果 n 的值大于 100，则退出循环
3   do
4       echo "$n"
5       n=$n+1
6   done
```

其次，再来看 for 循环，假设现在要实现：列出当前目录下每个普通文件所包含的行数，

代码及注解如下：

第三，for 循环语句：

```
1  files=`ls`        # 在当前目录下执行 ls，将所有的文件名保存在变量 files 中
2  for a in $files   # 循环地将 files 里面的每个单词赋给 a，赋完则退出循环
3  do
4      if [ -f $a ]  # 如果文件$a 是一个普通文件，那么就计算他的行数
5      then
6          wc -l $a
7      fi
8  done
```

注意：for 循环中，in 后面接的是一个字符串，字符串里面包含几个单词循环体就执行几遍，每执行一遍 a 的值都轮换地等于字符串里边的各个单词。

函数

SHELL 脚本在有些时候也可以编写模块化代码，将具有某一特定功能的代码封装起来，以供别处调用。比如：编写一个可以检测某用户是否在线的函数如下：

```
 1  check_user( )  # 定义一个函数 check_user( )，注意括号里面没有空格
 2  {
 3      if [ $1 = "quit" ]    # 若函数的第一个参数$1 为"quit"，则立即结束脚本
 4      then
 5          exit
 6      fi
 7
 8      USER=`who | grep $1 | wc -l`
 9      if [ $USER -eq 0 ]
10      then
11          return 0    # 判断用户$1 是否在线，是则返回 1，否则返回 0
12      else
13          return 1
14      fi
15  }
16
17  while true
18  do
19      echo -n "input a user name:"
20      read USER
21
22      check_user $USER  # 调用 check_user，并传递参数$USER
23
24      if [ $? -eq 1 ]   # 判断 check_user 的返回值$?是否为 1
25      then
26          echo "[$USERNAME] online."
27      else
```

```
28              echo "[$USERNAME] offline."
29      fi
30  done
```

注意几点：函数的定义中，括号里面不能写任何东西（第一行）；函数必须定义在调用之前；给函数传参的时候，传递的参数在函数的定义里用$n 来表示第 n 个参数。$?代表函数调用的返回值。

trap

脚本中经常有信号处理的语句，最常见的情况是：当脚本收到某个信号的时候，需要处理一些清理工作，然后再退出，类似于 POSIX 编程中的信号处理。脚本中使用 trap 来达到这个目的。例如：

```
trap “” INT
```

上面语句的含义是：当脚本收到信号 SIGINT 时，忽略该信号。在 LINUX 中所支持的信号可以使用命令 trap -l 来查看。信号名称的前缀要省略。trap 除了可以“忽略”信号，也可以“捕获”信号，比如：

```
trap do_something INT QUIT HUP
```

上面语句的含义是：当脚本收到 INT、QUIT 或者 HUP 信号时执行函数 do_something。此外还可以指定脚本正常退出时的默认动作，比如：

```
trap on_exit EXIT
```

以上语句的含义是：当脚本正常退出时，执行函数 on_exit。有时候，当一 SHELL 脚本收到某一个信号的时候，我们需要该脚本立即终止，且要执行正常退出时的清理函数，可以这样写：

```
trap on_exit EXIT
trap “:” INT HUP
```

以上两句语句的含义是：当脚本正常退出时执行函数 on_exit，当脚本收到信号 INT 或者 HUP 时执行空指令（此处冒号代表一个空指令，如果没有冒号，脚本将完全忽略该信号，不做相应，不能立即退出），完了之后正常退出，此时触发 EXIT 从而执行函数 on_exit。