Bash Shell 编程

Bash Shell 脚本的第一行用来说明程序执行脚本中的哪一行，称做 shbang 行；#！被称做魔术数字（magic number），用于供内核确认哪个程序将翻译并执行这个脚本，该行必须是第一行。

一、读取输入

1、变量

（1）变量类型

（a）局部变量：

其作用范围仅局限于创建变量的 shell。其变量名必须以字母或字母开始，其余部分则可以是字符、数字或下划线。

变量的设置方法一般有直接赋值、内建命令 declare 或 typeset 创建变量两种：

格式：

variable=value

declare variable=value

注：如果用 declare 来创建只读变量，变量不能被撤消，但可以被重新赋值。

declare 选项

选项	含义
-f	列出函数名及其定义
-r	将变量设置为只读
-x	将变量名导出到子 shell
-i	将变量设为整型
-a	将变量当作数组赋值

（b）环境变量：

即全局变量，其变量名一般约定为大写字母。

环境变量可以由 export 或 declare -x 导出：

格式：

export variable=value

variable=value；export variable

declare -x variable=value

注：通常，环境变量在用户主目录下的 .bash_profile 文件中定义。

export 选项

选项	含义
-f	name-value 形式被视为函数而不是变量
-n	将已导出的全局变量转换成局部变量
-p	显示所有全局变量

（2）变量替换：

当遇到要给变量值添加一个后缀的情形时，需要用花括号将变量名括起来。

花括号表示变量替换的形式

表示形式	说明
${variable}	基本变量替换。花括号限定变量名的开始和结束
${variable:-DEFAULT }	如果 variable 没有值，则返回 DEFAULT 的值
${variable:=DEFAULT }	如果 variable 没有值，则返回 DEFAULT 的值，另外将其值赋给 variable
${variable:+VALUE}	如果 variable 被设置，则返回 VALUE 的值，否则返回一个空串
${ #variable}	返回 variable 值的长度；若 variable 是*或@，则返回$*或$@所表示的元素的个数
${variable:? MESSAGE}	如果 variable 没有值，则返回 MESSAGE 的值，同时 shell 显示出 variable 的名字

Eg：

```
#!/bin/sh
# scripname: varsub
name="chauney_hang"
echo "Show \${name}"
echo -e "the name is ${name}.\n" #替换格式中花括号与变量间不需要留空格
unset name

echo "Show \${name:- ^_^}"
echo the name is ${name:- ^_^}. #替换格式中 name 与冒号":"和减号"-"间不需要空格，但后面的输出值可以留空格，下同。
echo -e "the name is $name .\n"
unset name

echo "Show \${name:= ^_^}"
echo "the name is ${name:= ^_^}."
echo -e "the name is $name .\n"
unset name

name="chauney_hang"
```

```
echo "Show \${name:+ The name is set.}"
echo ${name:+ The name is set.}
unset name
echo the next line is blank because name has been unset.
echo ${name:+ The name is set.}
echo -e "\33[33mTo emphasis the preview blank line.\33[0m\n"

name="chauney_hang"
echo "Show \${#name}"
echo -e "There are ${#name} characters in $name.\n"
unset name

echo "Show \${ name:? Please specify a name!}"
echo the name is ${name:? Please specify a name!}
[root@localhost root]# ./varsub
Show ${name}
the name is chauney_hang.
Show ${name:- ^_^}
the name is ^_^.
the name is .
Show ${name:= ^_^}
the name is ^_^.
the name is ^_^ .
Show ${name:+ The name is set.}
The name is set.
the next line is blank because name has been unset.
To emphasis the preview blank line.
Show ${#name}
There are 12 characters in chauney_hang.
Show ${ name:? Please specify a name!}
./varsub: line 31: name: Please specify a name!
```

2、read 命令

read 用于从终端或者文件中读取输入，它读取整行输入，而末尾的换行符被翻译成 null（空字符串）。如果没有指定名称，读取的行就被赋值到特定的变量 REPLY 中。同时，read 命令还可以用来使程序暂时停下来等待用户输入回车。

read 命令

格式	含义
read first last	读取输入到第一个空格或回车，将输入的第一个单词放入 first 中，而其他的则放在 last 中
read -a array	将单词清单放入 array 数组中

read -p prompt	打印提示，等待输入，并将输入存入 REPLY 中
read -r line	允许输入中包含反斜杠 "\"

二、数学计算

1、整数

declare -i 命令将变量声明为整数，该整数变量可以是不同进制的数（范围从 2 进制~36 进制）。

其声明格式：

variable=base#number-in-that-base

eg：

```
[root@localhost root]# declare -i x=012
[root@localhost root]# x=2#101
[root@localhost root]# echo $x
5
```

let 命令：用来做数学计算和数字表示法检测。

eg：

```
[root@localhost ScriptFiles]# let x+=1
[root@localhost ScriptFiles]# echo $x
6
[root@localhost ScriptFiles]# ((x+=1))
[root@localhost ScriptFiles]# echo $x
7
[root@localhost ScriptFiles]# x+=1
bash: x+=1: command not found
[root@localhost ScriptFiles]# let "x = x + 1"
[root@localhost ScriptFiles]# echo $x
8
[root@localhost ScriptFiles]# declare -i y=6
[root@localhost ScriptFiles]# let n="x>y?x:y"
[root@localhost ScriptFiles]# echo $n
8
```

注：双括号用来代替 let，如果参数中包含空格则需要引用。

2、浮点数

bash 支持整数的数学计算，如果要进行更加复杂的计算，可用 bc 或 awk 来解决。

eg：

```
[root@localhost root]# n=`echo "scale=3; 13/3" | bc`
[root@localhost root]# echo $n
4.333
```

[root@localhost root]# n=`awk -v x=2.33 -v y=4.56 'BEGIN{printf "%.2f", x*y}'`
[root@localhost root]# echo $n
10.62
注：赋值给 n 时是用反单引号“ ` ”来得到输出结果的。

三、位置参量与命令行参数

1、位置参量

命令行中，脚本名称后面以空格分隔的单词称为参数。命令行参数在脚本中可以按照其位置提供参考作用。

位置参量

位置参量	含义
$0	引用脚本名称
$#	获取位置参量的个数
$* $@	列出位置参量的清单
“$*”	使所有位置参量变成同一个字符串
“$@”	每一个参量都被看做分开的字符串
$1……${10}	引用单个位置参数

Set 命令：set 可以用来重置位置参量，一旦重置，原来的位置参量将被清空，但 $0 永远是脚本名而不会被改变。Set—用来恢复位置参量。

Eg：（in the scriptfile）

```
#!/bin/bash
# Scriptname: example
# Mainly to show the every positioen vector's meaning and the arguments!
echo The name of the script is $0.
${1:?"Please input the arguments first!"}
echo The number of the arguments is $#.
echo The arguments are $*.
echo The first argument is $1.
echo The second argument is $2.
oldargu=$*
set 'Jackey chen' Lily Kell
echo The new group is $*.
for name in $@
do
echo Hello $name
done
for name in "$*"
do
```

```
echo They are $name
done
for name in "$@"
do
echo See you $name
done
set $(date)
echo The date is $2 $3,$6
echo The old arguments is $oldargu.
set $oldargu
echo Bye to $*.
[root@localhost root]# sh argument Ann Shery Mill Can
The name of the script is argument.
The number of the arguments is 4.
The arguments are Ann Shery Mill Can.
The first argument is Ann.
The second argument is Shery.
The new group is Jackey chen Lily Kell.
Hello Jackey
Hello chen
Hello Lily
Hello Kell
They are Jackey chen Lily Kell
See you Jackey chen
See you Lily
See you Kell
The date is 12 月  6,2004
The old arguments is Ann Shery Mill Can.
Bye to Ann Shery Mill Can.
[root@localhost root]# sh argument
The name of the script is argument.
argument: line 5: 1: Please input the arguments first!
```

注：特殊变量编辑符？判断$1 是否有值，如果没有就退出脚本，打印信息。

四、条件结构与流控制

1、内建 test 命令

test 可以用一系列的括号来代替，当只有 test 命令或使用方括号时，表达式不能赋值。

test 命令操作符

判断操作符		判断是否为真
字符串判断	[sring1 =string2]	String1 等于 string2
	[sring1 ==string2]	String1 等于 string2
	[sring1 !=string2]	String1 不等于 string2
	[string]	string 不空
	[-z string]	string 长度为 0
	[-n string]	string 长度不为 0
	[-l string]	string 的长度
逻辑判断	[string1 -a string2]	string1 和 string2 都为真
	[string1 -a string2]	string1 或 string2 为真
	[!string1]	string1 不匹配
逻辑复合判断	[[pattern1 && pattern2]]	Pattern1 和 pattern2 都为真
	[[pattern1 || pattern2]]	Pattern1 或 pattern2 为真
	[[!pattern]]	pattern 不匹配
整数判断	[int1 -eq int2]	int1 等于 int2
	[int1 -ne int2]	int1 不等于 int2
	[int1 -gt int2]	int1 大于 int2
	[int1 -ge int2]	int1 大于或者等于 int2
	[int1 -lt int2]	int1 小于 int2
	[int1 -le int2]	Int1 小于或者等于 int2
文 件判断	[file1 -nt file2]	file1 比 file2 新
	[file1 -ot file2]	file1 比 file2 旧
	[file1 -ef file2]	file1 与 file2 有相同的设备或 I 节点数

let 命令操作符

操作符	含义
! ~	逻辑否
<< >>	位逻辑左、右移位
&	位逻辑与
^	位逻辑非
|	位逻辑或
&& ||	逻辑和、或
*= /= %= += -= <<= >>= &= ^= |=	快捷方式

2、if 命令

格式：

（1） 一般

```
if command
then
```

command（s）
fi
（2）test 模式
if test expression
then
command（s）
fi
或
if [string/numeric expression]
then
command（s）
fi
或
if [[string exprssion]]
then
command（s）
fi
（3）let 模式
if ((numeric expression))
then
command（s）
fi

嵌套 if 命令：

格式：

if command
then
command（s）
else
command（s）
fi

if/elif/else 命令：

格式：

if command
then
command(s)
elif command
then
command(s)
……
else
command(s)
fi

3、文件检验

文件检验操作符

检验操作符	检验是否为真
-d filename	目录存在
-e filename	文件存在
-f filename	非目录普通文件存在
-G filename	文件存在并属于一个有效的 GID
-L filename	文件是一个符号链接
-O filename	文件存在并属于一个有效的 UID
-r filename	文件可读
-s filename	文件尺寸非 0
-w filename	文件可写
-x filename	文件可执行

4、null 命令

null 命令用冒号：表示，是个内建的什么也不做 的命令，返回状态值为 0.如果 if 命令后面没有内容，同时又要避免产生错误信息，就需要在 then 后面加 null 语句。

5、case 命令

case 值中允许出现 shell 通配符和竖线（|）作为 OR 操作符。*) 的作用相当于 else 语句。

格式：

```
case variable in
value1)
command(s)
;;
value2)
command(s)
;;
……
*)
command(s)
;;
esac
```

用 here 文档和 case 命令建立菜单：

here 文档用来建立在屏幕上显示出来的菜单选项；case 则执行选择。

Eg:

```
#!/bin/sh
# scriptname: eth0
name=`cat /etc/modprobe.conf | awk '/eth0/{print $3}'` # 获取网卡名并存入 name 变量
echo -e "Your network card\047s name is: \33[35m$name\33[0m"
/sbin/ifconfig | awk '/eth0/{print "Device:",$1}' | grep eth0 > /dev/null #检测网卡是否被激活。
if [ $? = '0' ]
then #如果网卡被激活，则显示其 ip
    ip=`ifconfig eth0 | awk '/addr:/{print $2}' | cut -d: -f2`
    echo The network is already bring up! And the IP address is: $ip
else   #若网卡没有被激活，则作如下操作
    echo -e "\33[31mThe device doesn\047t work!\nplease try to ifconfig it up or modprobe it into the module!\33[0m"
    echo Next you should select a number to bring up your network [ 1 or 2 or 3 ]:
#下面是 here 文档和 case 相结合建立菜单
    cat << END
        [1] Open the network setup x-windows!
        [2] Bring up the network!
        [3] Quit!
END
    read number
    case $number in
    1)  setup;;
    2)  ifconfig eth0 up
        echo -e "The network maybe work abnormally, so you should restart the computer!\nInput Y or y, then the computer will reboot! Other keys will abort!"
        read respond
        if [[ "$respond" = [Yy] ]] # test 命令检测是否匹配
        then reboot
        else exit 0
        fi ;;
    3)  exit 0 ;;
    *)  echo "Error! Please select 1 or 2 or 3 !";;
    esac
    fi
fi
```

五、循环命令

1、for 循环

格式：

```
for variable in word_list
do
command(s)
done
```

2、while 循环

格式：

```
while command
do
command(s)
done
```

3、until 命令

until 的用法跟 while 的类似，只是在 until 后面的语句为假的时候执行循环体。

格式：

```
until command
do
command（s）
done
```

4、select 命令和菜单

select 的主要作用就是建立菜单，PS3 用来提示用户输入，默认的 PS3 是#？。在 PS3 提示显示以后，shell 等待输入。LINES 和 COLUMNS 变量用来决定在终端显示的菜单的层。Select 命令是循环命令，需要用 break 退出循环或 exit 退出脚本。

格式：

```
select variable in word_list
do
command(s)
done
```

注：bash 有四种提示符：（1）PS1：[\u@\h \W]\$ （2）PS2：> （3）PS3： （4）PS4：+

5、looping 命令

（1）shift 命令：

用来把参量列表位移指定次数，没有参数的 shift 把参量列表向左移动一位。一旦位移发生，被位移出列表的参数就被永远删除了。通常在 while 循环中，shift 用来读取列表中的参量。

格式：

```
shift [n]
```

（2）break 命令：

用来从循环中强行退出，但不退出程序。Break 后加一个数字可用来指定 break 强行退出的循环的层数。其中，最外面的循环数是 1,依次往里是 2、3……

格式：

break [n]

（3）continue 命令：

如果某些条件为真，continue 就控制跳转到循环的顶部，所以 continue 后面的语句将被忽略。Continue 后加一个数字可控制跳转到任何层的循环顶部重新开始执行。其中，最内的循环号是 1,往外依次是 2、3……

格式：

continue [n]

6、I/O 重定向

每个程序都有一个对应的标准输入（STDIN）、标准输出（STDOUT）和标准错误（SIDERR），它们指向发生输入和输出的地方，通常 STDIN 指向键盘，而 STDOUT 和 STDERR 则指向监视器。在所包含的范围之下，它们是文件句柄，并分别与文件描述字 0、1 和 2 想对应。

（1）、控制输入

（a）从文件取得输入：

格式：

command < input_file

（b）在线输入（Here 文档）：

其格式如下：

```
command << END
line one
line two
……
END
```

（2）、控制输出

如果输出到文件时要防止原目标文件被盖写，那么就需要使用内置 set 命令打开 noclobber 选项（set -o noclobber 命令可阻止覆盖现有文件）。

Eg：

```
[root@localhost root]# set -o noclobber
[root@localhost root]# echo hello > find.log
bash: find.log: cannot overwrite existing file
```

多数程序都将其错误信息发送给 STDERR，而不是给 STDOUT，这样就可以很方便地分离正常输出和错误输出。上面的错误信息可通过 STDERR 重定向来阻止其显示。

Eg：

```
[root@localhost root]# echo hello 2> error.dat > find.log
[root@localhost root]# echo $?
1
```

注：" 2> "这种表示形式表明将 STDERR 重定向。

如果使用" 2>> "形式也可以避免错误信息的显示，但这只是追加到文本后面而不是

覆盖它。

当 noclobber 被打开时，" >| "形式将不考虑 noclobber 选项而直接覆盖文件。

Eg：

```
[root@localhost root]# echo hello 2> error.dat > find.log
bash: error.dat: cannot overwrite existing file #这里出错的原因是 error.dat 已经存在，不能被覆盖。
[root@localhost root]# echo hello 2>| error.dat > find.log
[root@localhost root]# cat error.dat
bash: find.log: cannot overwrite existing file
```

注：如果想简单的忽略程序的输出，可以把该输出重定向到/dev/null，当程序结束后，/dev/null 就关闭，并被自动删除。

（3）、管道

文件中的输入可以通过管道重定向给一个循环，输出也可以通过管道从定向给一个文件,即 bash 循环输出可以通过管道给另一个 linux 命令或重定向到一个文件。

（4）、高级重定向

使用 m >& n 的形式可以合并任意两个输出流，送往 m 文件描述字的输出与送往 n 文件描述字的输出将合并在一起。

Eg：

```
[root@localhost root]# (date '+%B %d';ls -l *.foo) >| ls.log 2>&1
[root@localhost root]# cat ls.log
十二月 15
ls: *.foo: 没有那个文件或目录
[root@localhost root]# status=`(date '+%B%d' >&5 | echo hello 2>| error.dat > find.log) 5>&1`
[root@localhost root]# echo $status
十二月 15
```

注：status 的简化形式是：status=`（……） 5&>1`，可看出它表示将通往文字描述字 5 的输出流被重定向到文字描述字 1.

7、IFS 和循环

shell 内部域分隔符 IFS 默认是空格、制表符和换行符，如果在列表中要使用不同的分隔符，可自定义 IFS，在修改 IFS 前把原始值保存到另外一个变量中可以在需要时还原成默认值。

Examples：

Eg1：

```
#!/bin/sh
# scriptname: for
oldifs="$IFS" #保存原来的 IFS 分隔符
IFS=":" #设置 IFS 分隔符为冒号“：”
set Jim:Helly:Tommy
for person in $*
```

```
do
        echo Hello $person !
done
IFS="$oldifs" #还原 IFS 默认值
for name in 32 33 34 35 36 41 42 43 44 45 46 47 50 51 52 53 54 55 56 57
do
        echo -ne "$name \\$name\t"
        if [[ "$name" = 45 || "$name" = 57 ]];then printf "\n"
fi
done
[root@localhost root]# ./for
Hello Jim !
Hello Helly !
Hello Tommy !
32 # 33 # 34 # 35 # 36 # 41 ! 42 " 43 # 44 $ 45 %
46 & 47 ' 50 ( 51 ) 52 * 53 + 54 , 55 - 56 . 57 /
```

Eg2:

```
#!/bin/bash
# scriptname: numberic
# The script is use to put line numbers on all lines of chose file.
echo please input the filename you want to put line numbers in:
read name
while [ -z $name ] #判断 name 是否为空
do
        echo Error! Please input the filename first!
        read name
done
if [[ -f $name && -w $name ]] #判断文件是否存在且为非目录可写文件
then    count=1
         cat $name | while read line #cat  输出通过管道传到 while 循环并保存到 line 变量
         do
               let $((count == 1)) && echo Processing file $name > /dev/tty #let 判断当 count=1
        时执行 echo 并将输出传到屏幕上
              echo -e "$count\t$line"
              let count+=1
        done > tmp$$ #将输出重定向到 tmp$$，$$表示当前 shell 的 PID，通过在文件名末尾
追加 PID，目的在于保持文件名的唯一性
        mv -f tmp$$ $name
        rm -f tmp$$
else echo -e "\33[31mThe filename you input maybe is a directory or unwritable or
inexistent!\33[0m"
exit 1
fi
```

Eg3:

```
#!/bin/bash
# scriptname: select
COLUMNS=60 #COLUMNS 表示 select 建立的菜单的列宽度，默认值是 80.
LINES=1 #LINES 控制 select 菜单的垂直显示，默认是 24 行。
PS3="Please input a number of chioce:" #更改 PS3 提示符
select chioce in "Check the time of now!" "Look where you are!" "To see the modules that already running!"
do
    case $REPLY in
        1)  date | awk '{ print $4 }';break;;
        2)  pwd; break;;
        3)  lsmod; break;;
        *)  echo -e "$REPLY is not a valid chioce. Try again!\n"
            REPLY= ;;    #REPLY 为空时再次显示菜单
    esac
done
```

六、函数

函数本身就是一个命令或一组命令的名字。函数可以使程序模块化，提高效率，可以就在当前的 shell 环境中执行，即在执行像 ls 等可执行程序时不产生自进程。

使用函数的规则：

1、shell 总是先执行别名，然后是函数、内建命令，最后才执行可执行程序。

2、函数使用前必须先定义。

3、函数在当前环境下执行，它和调用它的脚本分享变量，并通过位置参量传递参数。通过 local 函数可以在函数内部建立本地变量。

4、如果在函数中使用 exit，则可以退出整个脚本，而退出函数则只是返回到调用函数的地方。

5、return 命令返回函数中最后一个命令的退出状态值或者是给定的参数值。

6、使用 export -f 可以将函数输出到子 shell。

7、使用 declare -f 可显示定义的函数清单，而 declare -F 则只显示函数的名字。

8、函数内部的陷阱是全局的，它们可以被脚本和脚本激活的函数共享。如果一个陷阱被定义为函数，它就可以被脚本共享，但可能产生意想不到的效果。

9、如果函数保存在其他文件中，必须通过 source 或者 dot 命令把它们装入当前脚本。

10、函数可以递归，且其递归次数没有限制。

1、函数定义

（1）构建函数

格式：

```
function function_name { comand(s); command(s); }
```

（2）复位函数：使用 unset 命令将函数从内存中删除。

格式：

unset -f function_name

（3）输出函数：输出到给子 shell。

格式：

export -f function_name

2、函数参数和返回值

（1）内建 local 函数：

local 创建的变量只在函数内部使用，退出函数变量即实效。

（2）参数：

通过位置参量可以向函数传递参数，该参数不会影响函数外使用的任何位置参量。

（3）内建 return 函数：

return 用来退出函数并返回到调用函数的地方。如果没有给 return 指定参数，返回的函数值就是最后一行的退出状态值。Return 返回的值只能是 0~256 之间的整数，且该值保存在“？”中，同时可以使用命令替换来捕捉函数的输出，即把整个函数放在括号内，前面加$（即：$(function_name)），或者通过引号把输出赋值给一个变量。

Eg:

```
[root@localhost root]# cat square
#!/bin/sh
function square {
    local sq
    let "sq=$1*$1"
    echo -e "the $1\047s square is $sq."
}
echo -n "Please input a number: "
read number
while [[ "$number" = "" ]]
do
    echo error!Please input a number!
    read number
done
square $number
echo REPLY=$?
re=$( square $number )
echo $re
[root@localhost root]# ./square
Please input a number: 78
the 78's square is 6084.
REPLY=0
```

the 78's square is 6084.

3、source（或者 dot）命令

函数通常被定义到.profile 中，当需要使用函数时，可以使用 source 或者 dot 加文件名来激活这些在文件中定义的函数。

Eg:

```
[root@localhost root]# cat greet
function greeting { echo "Hi $1! Wlecome to chauney's home!"; }
[root@localhost root]# source greet
[root@localhost root]# greeting Jam
Hi Jam! Wlecome to chauney's home!
```

七、陷阱信号

当程序运行时，按下 Control-C 或者 Control-/后程序就立刻终止，但很多时候当不希望信号到达时程序就立刻停止运行，而是希望忽略这个信号继续运行下去或者在程序退出前做些清除操作。这些可以通过 trap 命令来控制程序在收到信号以后的行为。

信号是由一个进程发送给另外一个进程的，或者在特定的键按下以后由操作系统发送给进程的，又或者在异常情况下发生时，由数字组成的非同步的消息。Trap 命令告诉 shell 根据收到的信号而以不同的方式终止当前的进程。

如果 trap 后面跟着一个用引号引用的命令，则在接收到指定的信号数字时就执行这个命令。Shell 共读取两次命令字符串，一次是在设置 trap 时，一次是在信号到达时。如果命令字符串被双引号引用，在第一次 trap 设置时就执行变量和命令替换。如果是用单引号引用，那么等到信号到达 trap 开始执行时，才运行变量和命令替换。

格式：

```
trap 'command;command' signal_number
trap 'command;command' signal_name
```

eg:

```
trap 'rm tmp*;exit 1' 0 1 2 15
trap 'rm tmp*;exit 1' EXIT HUP INT TERM
```

注：当 1（挂起）、2（中断）或者 15（软件终止）任何一个信息到达时就执行 rm 并 exit。

信号数字与信号（k:u-l）

号码	英文代号	类型	说明
0	EXIT	终止	当脚本退出时收到这个信号
1	SIGHUP	终止	报告用户终端被取消连接
2	SIGINT	终止	当用户键入 INTR 或中断字符（Contorl-C）时收到该信号
3	SIGQUIT	终止	当用户键入 QUIT 字符（Contorl-\）时收到该信号，并生成内存信息转储（core dump）

7	SIGBUS	程序错误	表明访问内存的地址不合法
8	SIGFPE	程序错误	表明出现严重的算术错误
9	SIGKILL	终止	这个信号永远是致命的，它不能被操纵或者忽略
11	SIGSEGV	程序错误	表明对合法的内存地址进行非法存取
13	SIGPIPE	程序错误	表明问题出自管道断开
14	SIGALRM	报警	表明计时器到时了
17	SIGCHLD	作业控制	表明子进程终止了
18	SIGCONT	作业控制	该信号不能被操纵或忽略，它直接启动一个停止的进程
19	SIGSTOP	作业控制	该信号不能被操纵或者忽略，它直接停止一个进程
20	SIGTSTP	作业控制	请求停止程序或脚本，当进程停止之前，文件和内存需要进入一个特定状态时，就要用它
21	SIGTTIN	作业控制	表明后台进程试图读取终端输入
22	SIGTTOU	作业控制	表明后台进程试图把输出写到终端上

Bash 允许在信号上使用象征性名称，例如没有前缀（SIG）或者用数字作为信号的名称。一个叫做 EXIT 的或者数字 0 的伪信号，将在 shell 退出时，导致一个陷阱的执行。

（1）信号复位：

trap 后面加一个信号或者数字，可把信号复位为默认动作。一旦调用了函数，函数设置的陷阱可以被调用这个函数的 shell 识别。同时，在函数外设置的陷阱也可被函数识别。

Eg： trap 2 或者 trap INT，表示恢复当 Control-C 按下就杀死当前进程的默认动作。

trap ‘trap 2’ 2 表示用户必须按两次 Control-C 才能终止程序，其中第一个陷阱捕捉信号，第二个复位信号默认动作为杀死进程。

（2）忽略信号：

如果 trap 后面跟一对空括号，列表中的信号将被进程忽略。

Eg： trap " " 1 2 or trap " " HUP INT

注：信号 1、2 将被 shell 进程忽略。

（3）陷阱列表：

通过 trap 命令可显示陷阱列表和分配给陷阱的命令清单。

Eg：

```
[root@localhost root]# trap 'echo hehe!;exit 1' 2
[root@localhost root]# trap
trap -- 'echo hehe!;exit 1' SIGINT
```

（4）函数中的陷阱：

如果使用陷阱处理函数中的信号，一旦函数被激活，它将影响整个脚本，即陷阱对于脚本来说是全局的。

Eg:

```
[root@localhost root]# more trapper
```

```
function trapper() {
        echo In trapper!
        trap 'echo Caught you ^_^' 2 #当 control-C 按下时就执行 echo 并继续执行脚本。
}
[root@localhost root]# cat main
#!/bin/bash
source trapper #调用 trapper 函数
while : #这是个无限循环
do
        echo In the main Script!
        trapper
        echo Still in main!
        sleep 5
done
[root@localhost root]# ./main
In the main Script!
In trapper!
Still in main!
^CCaught you ^_^
In the main Script!
In trapper!
Still in main!
```

注：该脚本是死循环，且无法用 Control-C 中断，而只能用 kill 命令来杀死进程。

八、调试

脚本的调试可用 bash 或 set 命令来实现。Bash 或 set 跟踪调试时，执行脚本中的每行都会在前面加一个（+）号。

除错选项

命令	选项	含义
bash -x scriptname	显示选项	命令替换后，执行以前显示脚本的每一行
bash -v scriptname	长选项	按照输入时候的样子在执行以前显示脚本的每一行
bash -n scriptname	不执行选项	解释但不执行命令
set -x	打开显示	跟踪脚本执行
set +x	关闭显示	关闭跟踪

九、其他

1、用 getopts 处理命令行选项

编写脚本时，也许还需要在其中包括一些选项，脚本根据选项的设置执行不同的动作。但选项必须是经过有效验证的，一种有效性验证方法需要脚本分析选项、验证选项，然后基于选项调整其执行方式，此方法应该由所有参数出发，并分别处理它们。

一种更好的方法是使用 getopts 命令，getopts 能读取命令行，如果有选项，便逐个取选项，并验证，然后再处理它。

（1）、无值选项

最简单的情况是只允许选项，而没有任何值。

无值选项由一个减号（getopts 不能处理加号选项）和一个字母定义，此选项可以彼此分开或结合。

格式：

getopts xyz variable #xyz 是允许的选项

getopts 从命令行逐个读取选项，并将选项（不包括减号）保存到其变量中。如果想获取所有选项，必须使用循环。当选项读取完毕后，getopts 退出状态值为 false。如果 getopts 找到一个不属于参数列表的选项，它显示选项并继续读取选项，直到最后选项为止。

Eg：

```
#!/bin/bash
# scriptname: Nopts
# To show options with no values.
while getopts xy choice 2> /dev/null
do
case $choice in
x) echo your option is X ! ;;
y) echo your option is Y ! ;;
*) echo Invalid option: Quiting!; exit 1 ;;
esac
done
[root@localhost root]# ./nopts -x
your option is X !
[root@localhost root]# ./nopts -xy
your option is X !
your option is Y !
[root@localhost root]# ./nopts -xz
your option is X !
Invalid option: Quiting!
```

（2）、有值选项

脚本也许需要选项有值（选项参数），例如，如果想编写一个脚本，接受一个区别

于默认的分隔符，这就需要把分隔符传递给脚本。Getopts 提供一个 OPTARG 的预定义变量，用来保存选项的值。

格式：

```
getopts x:y: variable
```

eg:

```
#!/bin/bash
#scriptname: Withopts
#To show option with values!
while getopts x:y: variable 2>/dev/null
do
    case $variable in
        x)  echo -e "your option is X !\nAnd the Value of X is $OPTARG!" ;;
        y)  echo -e "your option is Y !\nAnd the Value of Y is $OPTARG!" ;;
        *)  echo Invalid option: Quiting!; break ;;
    esac
done
echo The all Parameters contain $1 $2 $3 $4
[root@localhost root]# ./withopts -x me -y you -z he
your option is X !
And the Value of X is me!
your option is Y !
And the Value of Y is you!
Invalid option: Quiting!
The all Parameters contain -x me -y you
```

另一个内置变量 OPTIND，在命令选项后提供第一个参数的编号，它可以在位置参量内移动选项，使第一个参数（$1）拥有第一个实际的参数。

Eg：

```
#!/bin/bash
#scriptname: Withopts1
while getopts xy: variable 2>/dev/null
do
    case $variable in
        x)  echo your option is X ! ;;
        y)  echo -e "your option is Y !\nAnd the Value of Y is $OPTARG!" ;;
        *)  echo Invalid option: Quiting!; break ;;
    esac
done
echo Processing in body:
echo OPTIND contain $OPTIND
let n=$OPTIND-1
shift $n
echo \$1 contains $1
echo \$2 contains $2
[root@localhost root]# ./withopts1 -xy Me You He
```

```
your option is X !
your option is Y !
And the Value of Y is Me!
Processing in body:
OPTIND contain 3
$1 contains You
$2 contains He
```

2、使用 exec

exec 常用于改变当前 shell。如：exec /bin/csh 将导致启动 CShell 而不是 Bash。

exec 也可用于半永久的将输出重定向到一个文件描述字。同时，exec 还可重定向 STDOUT，然后将 STDOUT 恢复正常。

Eg：

```
[root@localhost root]# cat logfile
#!/bin/bash
#scriptname: logfile
exec 3> /tmp/log$$
for file in $0
do
echo "`date` - PID:$$ - Processing $file" >&3
exec > /dev/tty
echo "`date` - PID:$$ - Processing $file"
done
[root@localhost root]# ./logfile
三 12 月 15 09:42:05 CST 2004 - PID:3244 - Processing ./logfile
[root@localhost root]# cat /tmp/log3244
三 12 月 15 09:42:05 CST 2004 - PID:3244 - Processing ./logfile
```

3、使用 eval

eval 把它的参数连在一起，然后执行新创建的命令，有效的引起第二轮变量替换。

Eg：

```
[root@localhost root]# new="ls -l *.sxw"
[root@localhost root]# eval $new
-rw-r--r-- 1 root root 13315 11 月 30 07:40 program.sxw
-rw-r--r-- 1 root root 11174 12 月 13 14:26 sed.sxw
```

注：利用这个特性可间接运行命令

bash 对数组没有提供任何支持，但可用 eval 来创建伪数组。

Eg:

```
[root@localhost ScriptFiles]# more eval.dat
mm=12 dd=12 yy=2004 msg=' Appt with Kell '
mm=03 dd=09 yy=2004 msg=' Attend a meeting'
mm=04 dd=11 yy=2004 msg=' An nice day'
```

```
[root@localhost ScriptFiles]# cat eval.sh
#!/bin/bash
# scriptname: eval.sh
if [ $# -ne 3 -a $# -ne 0 ]
then echo Usage: eval.sh {mm} {dd} {yyy};exit 0
fi
while read DATA
do
eval $DATA
eval "DATEBOOK_${mm}_${dd}_$yy='$msg'"
done < eval.dat
if [ $# -eq 3 ]
then curdate="$1_$2_$3"
else curdate="`date '+%m_%d_%Y'`"
fi
echo Datebook Entry for $curdate:
eval echo "$"DATEBOOK_$curdate
[root@localhost ScriptFiles]# ./eval.sh 03 09 2004
Datebook Entry for 03_09_2004:
Attend a meeting
```

注：eval 用法三例：

例一:

```
#寻找合符条件的变量名,然后将该变量的值赋予另一变量
v1=aaa ; v2=bbb ; c=1
vname=v$c #找到符合条件的变量名为 v1
eval vvv="$"$vname ; echo vvv: $vvv #将变量 v1 的值赋予 vvv,即,使 vvv=aaa
eval vvv='$'$vname ; echo vvv: $vvv #将变量 v1 的值赋予 vvv,即,使 vvv=aaa
#eval vvv=$$vname ; echo vvv: $vvv #错误用法
```

例二:

```
以变量 v1 的值 aaa 作为变量名,将变量 vaaa 的值赋予这一新定义的变量 aaa
v1=aaa ; vaaa="This is aaa"
#eval $v1=$vaaa ; echo aaa: $aaa #错误用法
#eval $v1="$vaaa" ; echo aaa: $aaa #错误用法
eval $v1='$vaaa' ; echo aaa: $aaa
```

例三:

```
以变量 v1 的值 aaa 作为变量名,并将变量名字串作为值赋予自身
v1=aaa ; vaaa="This is aaa"
eval $v1=$v1 ; echo aaa: $aaa #与例二的错误用法不同,这一用法是正确的
eval $v1="$v1" ; echo aaa: $aaa #与例二的错误用法不同,这一用法是正确的
eval $v1='$v1' ; echo aaa: $aaa
```

4、使用后台进程

当一个程序需要花费很长时间才能完成，并且不需要提供输入时，可在命令行后面加入&字符，将其放入后台执行，当后台进程完成时，shell 就显示一个信息，给出该进程的退出码。

Eg：

```
[root@localhost root]# find / -name "nvidia" -print > find.log &
[1] 2275
[root@localhost root]# ps -e | grep 2275
2275 pts/0 00:00:02 find
[1]+ Exit 1 find / -name "nvidia" -print >find.log
[root@localhost root]# cat find.log
/root/nvidia
/mnt/software/Program Files/RivaTuner/Databases/nvidia
/mnt/software/Program Files/RivaTuner/PatchScripts/nvidia
/mnt/software/Program Files/RivaTuner/Presets/nvidia
/var/lib/nvidia
/proc/driver/nvidia
/sys/class/nvidia
/sys/bus/pci/drivers/nvidia
```

注：每次只能把一个程序放入后台。

shell 中有特殊含义的字符

字符	说明
"	双引号用来是 shell 无法认出空格、制表符和其他大多数特殊字符
'	单引号使 shell 无法认出其中所有的特殊字符，除了结尾的单引号以外
、	倒引号用于命令替换，在倒引号中间的命令产生的输出将代替原命令中的倒引号字符串
\	反斜线使 shell 无法认出其随后的字符
;	分号允许一行中放多个命令
&	当一个命令的末尾放上一个&，就把这个命令放在后台运行
( )	圆括号用来创建组命令，其命令表在同一 shell 内部执行，命令完成后不留下影响
{ }	花括号用来创建 shell 过程的命令块
|	管道用来把一个命令的输出连接到另一个命令的输入
< > &	用来表示需要进行重定向
* ? [ ] ！	用于模式匹配
$	$字符表示一个变量名的开头
#	#字符用于脚本作注释开始