第三讲：Linux 环境和 Shell 编程

申屠慧
能源与环境系统工程（智慧能源班）3210103417

1 重定向、管道

Shell 中一切都是文件，包括标准输入输出。比如 `ls -l` 的输出结果默认是输出到 `stdout`，也就是标准输出设备，但你可以用重定向操作符将其重定向为一个文件，比如：

```
ls -l > output.txt
```

这个就是把 `ls -l` 的输出结果保存在 `output.txt` 中。

操作符 `>` 是重写一个新文件，而 `>>` 是追加在原文件末尾。尝试一下。

我们知道除了标准输出文件 `stdout`，还有一个标准错误文件 `stderr`，这个往往和 `stdout` 指向同一个输出设备。但可以通过重定向分离。一般默认的输出是标准输出文件，而 2 则值标准错误文件。比如：

```
kill -HUP 1234
```

删除进程 1234, 显示

```
bash: kill: (1234) - No such process
```

因为一般不会正好有这个进程号。这里全部都是标准错误文件输出的，所以如果运行

```
kill -HUP 1234 > output.txt
```

则得到一个空文本文件。要

```
kill -HUP 1234 2> output.txt
```

才能截获错误信息到文本文件。如果我们想同时截获标准输出和错误输出并放在一个文件里，可以

```
kill -HUP 1234 >killout.txt 2>&1
```

注意，`2>&1` 之间不能有空格。如果输出中有很多垃圾信息你不想看，可以将其重定向到一个垃圾桶：

```
kill -l 1234 2>/dev/null
```

既然标准输出可以重定向，标准输入自然也可以

```
ls ~ -lR > output.txt
```

先搞一个大一点的输出，把它丢给 `more` 做输入，功能是把输入一屏幕一屏幕输出。

```
more < output.txt
```

注意这里操作符反向了。

然后这里有一件很土的事情，就是这个 `output.txt` 就是一个中转站，何必一定要存下来呢？就不能直接转移过去？也就是把一个命令的输出当作另一个命令的输入。这个当然可以，就叫管道，操作符是 |。

```
ls ~ -lR | more
```

大家可能注意到我曾经在输出中只显示有关键字的部分，用的也是管道：

```
ls ~ -lR | grep usr
```

管道和重定向可以根据需要多重连接。

2 循环过程

Shell 提供了表示循环和分支判定的结构，如：

```
for file in *
do
if grep -l figure $file
then
ls $file -l
fi
done
```

这里

```
for file in *
do
...something...
done
```

是一个循环结构，`file` 是一个用户定义的变量，`*` 表示当前目录下所有文件，`in` 表示遍历。所以这段命令的意思就是遍历当前目录下所有文件，对每个文件重复做 `...something...` 这件事。比如：

```
for fAd in *
do
echo $fAd
done
```

就是遍历当前目录，将全部文件名都打印一遍。命令 `echo` 是打印输出的意思。

3 分支过程

然后看内层是一个分支结构：

```
if [ ...condition... ]
then
...do something...
fi
```

以上是一个标准的 `if ... fi` 结构。但我们用的时候没有 `...condition...`，类似 C 语言，只要 `...condition...` 部分不为空，我们就认为条件是真。所以这里的意思是

```
if grep -l figure $file
then
ls $file -l
fi
```

只要

```
grep -l figure $file
```

过滤的结果不为空，即之前遍历到的 `$file` 这些文件里存在包含 `figure` 单词的，就把它的详细信息列出来。你自己可以根据需要做自由的调整，比如：

```
for file in *
do
if grep -l figure $file
then
cp $file ~/Documents
fi
done
```

这是什么意思？已经具有收集功能了。

4 通配符（wildcard expansion）

注意到这里 * 是一个典型的通配符，意思是全部文件；类似的通配符还有：

- ?，任一单个字符，比如：lec0?，那么 lec01，lec02 和 lec0k 都匹配；[]，括号中任一字母。比如：lec0[abc]，那么 lec0a，lec0b 和 lec0c 都匹配；
- [^] 或 [!], 不含括号内的任一字符，比如：lec0[!12]，那么 lec01 和 lec02 都不匹配；
- {},包含大括号内任一字串,比如:lec03{.log,.pdf} 则匹配 lec03.pdf 或 lec03.log。

以上通配符可自由组合，比如：*0?.{t??,a*}。

然后 $ 操作符主要有两个意思，要么是表示将一个变量替换成变量值，如之前的 $file；也可以表示对命令的替换。比如：

```
grep -l crazyfish *
```

表示列出目录下所有含有 crazyfish 字符串的文件名，那么

```
ls -l $(grep -l crazyfish *)
```

就表示进一步列出这些文件的详细信息。

5 脚本代码 (script)

现在我们可以用连续的命令加上流程控制加上通配符实现比较复杂的功能。这些命令可以形成一个脚本，并用 bash 去运行。具体做法是依次写下命令，然后在第一行写上 #!/bin/bash 表示这个脚本用哪个 shell 执行。下面是一个完整的脚本的例子：

```
#!/bin/sh
# first
# This file looks through all the files in the current
# directory for the string POSIX, and then prints the names of
# those files to the standard output.
for file in *
do
if grep -q crazyfish $file
```

```
then
echo $file
fi
done
exit 0
```

将它存成一个文本文件，比如叫 `test.sh`，然后将其设置成可执行，然后就可以像运行一个命令一样运行它。实际上，它构成一个组合命令，或称批处理。这里 `grep -q` 表示安静地运行，或者说，不要实际输出，因为我们这里只要求 `grep` 返回是否找到这样一个状态就行。

6 变量

既然可以写脚本了，变量自然很重要。这里变量直接使用就可以：

```
salutation=Hello
echo $salutation
```

一赋值一输出。默认就是字符串，引不引号无所谓：

```
salutation="Yes Dear"
echo $salutation
```

就算是给数字，还是字符串：

```
salutation=7+5
echo $salutation
```

命令 `read` 可以由用户给变量赋值：

```
read salutation
echo $salutation
```

从上面的过程我们可以看到，在 shell 中，双引号和不加引号都是字符串，而单引号和 \ 才是真正的引号：

```
#!/bin/sh
myvar="Hi there"
echo $myvar
echo "$myvar"
echo '$myvar'
echo \$myvar
echo Enter some text
```

```
read myvar
echo '$myvar' now equals $myvar
exit 0
```

(参见：example/scripts/quoting.sh)

7 环境变量

除了我们自定义的变量，shell 还内置了很多环境变量，供我们和系统交互。

- `$HOME`，用户目录；
- `$PATH`，执行程序的路径，放在这些路径下的可执行文件能够直接被运行；
- `$PS1`，命令行提示符，普通用户是 `$`，超级用户是 `#`；
- `$PS2`，第二提示符，比如等待输入状态，通常是 `>`；
- `$IFS`，输入参数的间隔符，一般是一个空格，也可以是 TAB 或换行；

以上都可以用 `echo` 输出查看。但是

```
echo $PS1
```

会得到

```
${debian_chroot:+($debian_chroot)}\u@\h:\w\$
```

这里转义字符 `\u` 表示用户名，`\h` 表示主机名，`\w` 表示当前路径。以下环境变量专门给 scripts 使用，

- `$0`，脚本名；
- `$#`，传递给脚本的参数个数；
- `$$`，脚本的进程编号；
- `$1`，`$2`，...，传递给脚本的第 1 个，第 2 个参数，...；
- `$*`，传递给脚本的全部参数，作为一个字符串；
- `$@`，传递给脚本的全部参数，有几个参数就是几个字符串；

测试脚本 (try_var.sh)

```
#!/bin/bash

echo "The program $0 is now running"
echo "The first parameter was $1"
echo "The second parameter was $2"
echo "The third parameter was $3"
echo "The all parameters were $* "

echo "The parameter list was: "
for para in $@
do
echo $para
done

exit 0
```

体会一下 `@*` 和 `$@` 的区别。

8 判断条件（condition）

实际上，正经的 `if` 后面应该跟一个条件判断，条件的写法是：

```
if ...condition...
```

这里 `...condition...` 的有两种表达，`test` 或者 `[ ]`，后者方括号两端必须各有一个空格。

下面是一个如何测试/bin/bash 文件状态的示例：

```
#!/bin/sh
if [ -f /bin/bash ]
then
echo "file␣␣/bin/bash␣␣exists"
fi
if [ -d /bin/bash ]
then
echo "/bin/bash␣␣is␣␣a␣␣directory"
else
echo "/bin/bash␣␣is␣␣NOT␣␣a␣␣directory"
fi
```

表 1: 字符串逻辑运算关系表

字符串比较	结果
string1 = string2	True 如果字符串相等
string1 != string2	True 如果字符串不想等
-n string	True 如果字符串非空
-z string	True 如果字符串空

表 2: 算术逻辑运算关系表

算术表达式	结果
expression1 -eq expression2	True 如果表达式值相等
expression1 -ne expression2	True 如果表达式值不想等
expression1 -gt expression2	True 如果 expression1 大于 expression2
expression1 -ge expression2	True 如果 expression1 大于等于 expression2
expression1 -lt expression2	True 如果 expression1 小于 expression2
expression1 -le expression2	True 如果 expression1 小于等于 expression2
! expression	True 如果 expression false, 或反之

表 3: 文件逻辑运算关系表

文件判断	结果
-d file	True 如果 file 是目录
-e file	True 如果 file 存在
-f file	True 如果 file 是正经的文件
-g file	True 如果 file 设置了 group id
-r file	True 如果 file 可读
-s file	True 如果 file 大小不是零
-u file	True 如果 file 设置了 user id
-w file	True 如果 file 可写
-x file	True 如果 file 可执行

在测试为 true 之前，所有的文件条件测试都要求该文件也存在。这个列表只包含 test 命令更常用的选项，因此要获得完整的列表，请参考手动条目。如果你使用的是内置了 test 的 bash，可以使用 help test 命令来获取更多详细信息。

9 控制结构

在下面几节中，语句是在条件满足时、同时或直到满足为止要执行的一系列命令。

9.1 if

9.1.1 if

if 语句非常简单: 它测试一个命令的结果，然后有条件地执行一组语句.

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

常见用法:

```
#!/bin/sh
echo "Is it morning? Please answer yes or no"
read timeofday
if [ $timeofday = "yes" ]; then
echo "Good morning"
else
echo "Good afternoon"
fi
exit 0
```

输出为:

```
Is it morning? Please answer yes or no
yes
Good morning
$
```

在 if 语句内部使用了额外的空格来缩进语句。这只是为了方便人类读者; shell 会忽略额外的空白。

9.1.2 elif

你可以通过使用 elif 结构来防止出现其他情况，它允许你添加第二个条件，当执行 if 的 else 部分时进行检查。

```
#!/bin/sh
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
```

注意：变量的问题

按回车键 (或某些键盘上的回车键)，而不是回答问题。你会得到这样的错误信息:

[: =: 期望的一元运算符

出了什么问题? 问题出在第一个 if 子句中。在测试变量 timeofday 时，它由一个空字符串组成。因此，if 子句看起来像

```
If [= " yes "]
```

这不是一个有效的条件。为了避免这种情况，你必须在变量两边使用引号:

```
If [" $timeofday " = " yes "]
```

然后一个空变量给出了有效的测试:

```
If [" " = " yes "]
```

新脚本如下:

```
#!/bin/sh
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
```

9.2 for

使用 for 结构来遍历一个范围的值，这个范围可以是任何一组字符串。它们可以简单地在程序中列出，或者，更常见的是，文件名的 shell 扩展的结果。

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

9.2.1 固定字符串的 for 循环

```
#!/bin/sh
for foo in bar fud 43
do
echo $foo
done
exit 0
```

输出为:

```
bar
fud
43
```

9.2.2 带有通配符扩展的 for 循环

```
#!/bin/sh
for file in $(ls f*.sh); do
lpr $file
done
exit 0
```

shell 展开 f*.sh，给出所有匹配此模式的文件名称。

注意，shell 脚本中所有变量的展开都是在脚本执行时完成的，永远不会在脚本编写时完成，因此变量声明中的语法错误只会在执行时发现，如前面我们引用空变量时所示。

9.3 while

当你需要重复一个命令序列，但事先不知道它们应该执行多少次时，通常会使用 while 循环，它的语法如下:

```
while condition do
  statements
done
```

示例:

```
#!/bin/sh
echo "Enter␣password"
read trythis
while [ "$trythis" != "secret" ␣];␣␣do
echo␣␣"Sorry, try again"
read␣␣trythis
done
exit␣␣0
```

输出:

```
Enter password
password
Sorry, try again
secret
$
```

9.4 until

```
until condition
do
  statements
done
```

这与 while 循环非常相似，但条件测试相反。即循环会继续，直到条件为真，而不是在条件为真时继续。一般来说，如果循环应该至少执行一次，就使用 while 循环; 如果它可能根本不需要执行，则使用 until 循环。

```
#!/bin/bash
until who | grep "$1" >/dev/null
do
sleep 60
done
# now ring the bell and announce the expected user .
echo -e '\a'
echo "****␣$1␣has␣just␣logged␣in␣****"
exit 0
```

9.5 case

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

case 结构使你能够以相当复杂的方式将变量的内容与模式进行匹配，然后根据匹配的模式，允许执行不同的语句。/par 每个模式行都以双分号 (;;) 结尾。你可以在每个模式和下一个模式之间放多个语句，所以需要用双分号来标记一个语句的结束和下一个模式的开始。

示例一:

```
#!/bin/sh
echo "Is␣it␣morning?␣Please␣answer␣yes␣or␣no"
read timeofday
case "$timeofday" 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 语句执行时，它获取 timeofday 的内容并依次与每个字符串进行比较。一旦字符串与输入匹配，case 命令就会执行紧跟在) 之后的代码并完成。

示例二：

```
#!/bin/sh
echo "Is it morning? Please answer yes or no"
read timeofday

case "$timeofday" in
yes | y | Yes | YES ) echo "Good Morning";;
n* | N* ) echo "Good Afternoon";;
*  ) echo "Good Afternoon";;
esac

exit 0
```

这个脚本在 case 的每个条目中使用多个字符串，以便 case 为每个可能的语句测试多个不同的字符串。这段代码还展示了如何使用 * 通配符，尽管这可能会匹配意想不到的模式。例如，如果用户输入 never，那么它将被 n* 匹配，并将显示 Good Afternoon，这不是预期的行为。还要注意，* 通配符表达式不能在引号中使用。

示例三：

```
#!/bin/sh
echo "Is it morning? Please answer yes or no"
read timeofday

case "$timeofday" in
  yes | y | Yes | YES )
    echo "Good Morning"
    echo "Up bright and early this morning"
```

```
    ;;
  [nN]*)
    echo "Good Afternoon"
    ;;
  *)
    echo "Sorry, answer not recognized"
    echo "Please answer yes or no"
    exit 1
    ;;
esac

exit 0
```

这段代码改变了 no 分支的匹配方式。您还可以看到如何为 case 语句中的每个模式执行多个语句。你必须小心地将最显式的匹配放在第一位，最一般的匹配放在最后。这一点很重要，因为 case 会执行它找到的第一个匹配项，而不是最佳匹配项。如果你把 *) 放在第一位，无论输入什么，它总是会被匹配。

为了使分支匹配更强大，你可以使用这样的东西:

```
  [yY] | [yY] [Ee][Ss])
```

这在允许多种答案的同时限制了允许的字母，比 * 通配符提供了更多的控制。

10 列表

有时你想把命令串联起来，比如

```
if [ -f this_file ]; then
  if [ -f that_file ]; then
    if [ -f the_other_file ]; then
      echo "All files present, and correct"
    fi
  fi
fi
```

或者你可能希望一系列条件中至少有一个为真，

```
if [ -f this_file ]; then
```

```
  foo=" True"
elif [ -f that_file ]; then
  foo=" True"
elif [ -f the_other_file ]; then
  foo="True"
else
  foo="False"
fi
if [ "$foo" = "True" ]; then
  echo "One of the files exists"
fi
```

虽然这些可以用多个 if 语句实现，但你会发现结果很尴尬。shell 有一对特殊的结构来处理命令列表:AND 列表和 OR 列表。

10.1 AND 列表

AND list 结构使您能够执行一系列命令，只有在前面的所有命令都成功时才执行下一个命令。语法为:

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

从左边开始，每条语句都被执行; 如果返回 true，则执行右边的下一个语句。这个过程会一直持续下去，直到有一条语句返回 false，在这之后，列表中的所有语句都不会再被执行。每条语句都是独立执行的。如果所有命令都成功执行，AND 列表作为一个整体就成功了，否则就失败了。

示例:

```
#!/bin/sh

touch file_one
rm -f file_two

if [ -f file_one ] && echo "hello" && [ -f file_two ] && echo "␣there"
then
  echo "in␣if"
else
  echo "in␣else"
fi
exit 0
```

输出:

```
hello
in else
```

10.2 OR 列表

OR 列表结构使我们能够执行一系列命令，直到其中一条成功，然后就不再执行了

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

从左边开始，每条语句都被执行。如果返回 false，则执行右边的下一个语句。这个过程会一直持续下去，直到有一条语句返回 true，此时不再执行其他语句.

将之前的示例复制修改一下:

```
#!/bin/sh

touch file_one
rm -f file_two

if [ -f file_one ] || echo "hello" || [ -f file_two ] || echo "␣there"
then
  echo "in␣if"
else
  echo "in␣else"
fi
exit 0
```

输出:

```
hello
in if
```

如果将两种逻辑结合起来，例如

```
[ -f file_one ] && command for true || command for false
```

如果测试成功，则执行第一个命令，否则执行第二个命令。通常你应该使用大括号来强制求值的顺序。

11 语句块

如果你想在一个只允许使用一个语句的地方使用多个语句，例如在 AND 或 or 列表中，你可以将它们放在大括号中，形成一个语句块。例如:

```
get_confirm && {
  grep -v "$cdcatnum" $tracks_file > $temp_file
  cat $temp_file > $tracks_file
  echo
  add_record_tracks
}
```

12 函数

要定义一个 shell 函数，只需编写它的名称，后跟空括号，并将语句括在大括号中:

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

示例:

```
#!/bin/sh

foo() {
  echo "Function foo is executing"
}

echo "script starting"
foo
echo "script ended"

exit 0
```

输出:

```
script starting
Function foo is executing
script ending
```

你必须始终在调用之前定义一个函数。

你可以使用 return 命令让函数返回数值。让函数返回字符串的通常方法是，函数将字符串存储在一个变量中，然后可以在函数结束后使用该变量。或者，你可以 echo 一个字符串并捕获结果，如下所示:

```
foo () {echo JAY;}
...
result="$(foo)"
```

你可以使用 local 关键字在 shell 函数中声明局部变量。这样变量就只能在函数的作用域中了。否则，函数可以访问其他在作用域中本质上是全局的 shell 变量。如果局部变量与全局变量同名，它会覆盖该变量，但仅在函数内部。例如，你可以对前面的脚本进行以下更改，以查看实际效果:

```
#!/bin/sh
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
```

下面是一个有返回值的示例:

```
#!/bin/sh

yes_or_no() {
  echo "Is your name $*?"
  while true
  do
    echo -n "Enter yes or no: "
    read x
    case "$x" in
      y | yes ) return 0;;
```

```
        n | no ) return 1;;
        * ) echo "Answer yes or no"
      esac
    done
}
#定义函数 yes_or_no

echo "Original parameters are $*"
if yes_or_no "$1"
then
  echo "Hi $1, nice name"
else
  echo "Never mind"
fi
exit 0
#程序的主要部分
```

典型输出:

```
$ ./my_name Rick Neil
Original parameters are Rick Neil
Is your name Rick ?
Enter yes or no: yes
Hi Rick, nice name
$
```

13 命令

13.1 break

在满足控制条件之前，使用 break 对包围的 for、while 或 until 循环进行转义。你可以给 break 一个额外的数值参数，也就是要跳出循环的次数，但这会使脚本非常难以阅读。默认情况下，break 会转义一个关卡。

```
#!/bin/sh

rm -rf fred*
echo > fred1
```

```
echo > fred2
mkdir fred3
echo > fred4

for file in fred*
do
  if [ -d "$file" ]; then
    break;
    fi
done

echo first directory starting fred was $file

rm -rf fred*
exit 0
```

13.2 : 命令

冒号命令为 null 命令。它偶尔可以简化条件的逻辑，作为 true 的别名。由于它是内置的，: 运行速度比 true 快，尽管它的输出可读性也差得多。你可能会看到它被用作 while 循环的条件;`While:` 实现无限循环，代替了更常见的 While true。结构在变量的条件设置中也很有用。

```
: ${var:=value}
```

示例:

```
#!/bin/sh
rm -f fred
if [ -f fred ]; then
  :
else
  echo file fred did not exist
fi
exit 0
```

13.3 continue

这个命令使封闭的 for、while 或 until 循环在下一次迭代时继续，循环变量取列表中的下一个值:

```
#!/bin/sh

rm -rf fred*
echo > fred1
echo > fred2
mkdir fred3
echo > fred4

for file in fred*
do
if [ -d "$file" ]; then
  echo "skipping directory $file"
  continue
  fi
  echo file is $file
done

rm -rf fred*
exit 0
```

Continue 可以将要恢复的封闭循环编号作为可选参数，以便您可以部分跳出嵌套循环。这个参数很少使用，因为它经常使脚本更加难以理解。例如,

```
  for x in 1 2 3
  do
    echo before $x
    continue 1
    echo after $x
  done
```

结果为:

```
  before 1
  before 2
  before 3
```

13.4 . 命令

在当前 shell 中执行命令。

```
. ./shell_script
```

默认情况下，在执行 shell 脚本时，会创建一个新的环境，因此脚本对环境变量所做的任何更改都会丢失。而 dot 命令则允许被执行的脚本改变当前环境。

示例一:

```
#!/bin/sh
version=classic
PATH=/usr/local/old_bin:/usr/bin:/bin:.
PS1="classic> "
```

示例二:

```
#!/bin/sh
version=latest
PATH=/usr/local/new_bin:/usr/bin:/bin:.
PS1="latest version> "
```

示例三: (脚本与 dot 命令结合设置环境)

```
$ . ./classic_set
classic> echo $version
classic
classic> . /latest_set
latest version> echo $version
latest
latest version>
```

13.5 echo

使用 echo 命令输出一个后跟换行符的字符串。常见的问题是如何抑制换行符。Linux 中常用的方法是使用

```
echo -n "string to output"
```

13.6 eval

eval 命令使你能够评估参数。它内置在 shell 中，通常不作为一个单独的命令存在。也许最好用一个借用 X/Open 规范本身的简短例子来说明:

```
foo=10
x=foo
y='$'$x
echo $y
```

这将输出 $foo

```
foo=10
x=foo
eval y='$'$x
echo $y
```

将输出 10

13.7 exec

exec 命令有两种不同的用途。它的典型用途是用一个不同的程序替换当前的 shell。exec 的第二个用途是修改当前的文件描述符。

13.8 exit n

exit 命令会让脚本退出，退出代码为 n。如果你允许你的脚本退出而没有指定退出状态，那么脚本中执行的最后一个命令的状态将被用作返回值。

示例:

```
#!/bin/sh
if [ -f .profile ]; then
  exit 0
fi
exit 1
```

13.9 export

export 命令使作为其参数命名的变量在子 shell 中可用。

示例:

脚本 2:

```
#!/bin/sh
echo "$foo"
echo "$bar"
```

脚本 1:

```
#!/bin/sh
foo="The first meta-syntactic variable"
export bar="The second meta-syntactic variable"
export2
```

输出

```
$ ./export1
The second meta-syntactic variable
$
```

13.10 expr

expr 命令将其参数作为一个表达式进行计算.

```
x=`expr $x + 1`
```

13.11 printf

转换说明符由% 构成。后面跟着转换字符。

13.12 return

return 命令会导致函数返回，正如我们之前在学习函数时提到的那样。

13.13 set

```
#!/bin/sh
echo the date is $(date)
set $(date)
echo The month is $2
exit 0
```

表 4: 退出码表

退出代码	描述
0	正常退出
1-125	脚本可用错误码
126	文件无法执行
127	没有找到命令
128 及以上	出现信号

表 5: 表达式求值表

表达式求值	描述
expr1 | expr2	如果 expr1 非 0, 则为 expr1, 否则为 expr2
expr1 & expr2	如果其中一个表达式为为 0 则为 0
expr1 = expr2	相等则为 1
expr1 > expr2	大于
expr1 >= expr2	大于等于
...	...

表 6: 转义序列

转义序列	描述
\”	”
\\	\
\a	响铃
\b	退格
\c	抑制进一步输出
\n	换行
\r	回车
...	...

表 7: 转义序列

转义序列	描述
C	输出字符
D	输出十进制数
S	字符串
%	%

13.14 shift

shift 命令将所有参数变量向下移动 1，这样 $2 变成 $1，$3 变成 $2，以此类推。之前 $1 的值被丢弃，而 $0 保持不变。

```
#!/bin/sh
while [ "$1" != "" ]; do
echo "$1"
shift
done
exit 0
```

13.15 trap

trap 命令被传递要采取的动作，后面跟着要启动 trap 的信号名称。

示例：

```
#!/bin/sh
trap 'rm -f /tmp/my_tmp_file_$$' INT
echo creating file /tmp/my_tmp_file_$$
date > /tmp/my_tmp_file_$$
echo "press interrupt (CTRL-C) to interrupt ...."
while [ -f /tmp/my_tmp_file_$$ ]; do
echo File exists
sleep 1
done
echo The file no longer exists
trap INT
echo creating file /tmp/my_tmp_file_$$
date > /tmp/my_tmp_file_$$
echo "press interrupt (control-C) to interrupt ...."
while [ -f /tmp/my_tmp_file_$$ ]; do
echo File exists
sleep 1
done
echo we never get here
exit 0
```

如果你运行这个脚本，按住 Ctrl，然后在每个循环中按 C(或任何你的中断

键组合)，你会得到以下输出:

```
creating file /tmp/my_tmp_file_141
press interrupt (CTRL-C) to interrupt ....
File exists
File exists
File exists
File exists
The file no longer exists
creating file /tmp/my_tmp_file_141
press interrupt (CTRL-C) to interrupt ....
File exists
File exists
File exists
File exists
```

13.16 unset

```
# !/bin/sh
foo= "␣Hello␣World␣" echo $foo
Unset foo echo $foo
```

14 两个更有用的命令和正则表达式

14.1 find

```
# find / -mount -name test -print
/usr/bin/test
#

find [path] [options] [tests] [actions]
```

14.2 grep

```
grep [options] PATTERN [FILES]
```