Bash脚本15分钟进阶教程

04/23. 2014

这里的技术技巧最初是来自谷歌的“Testing on the Toilet” (TOTT)。这里是一个修订和扩增版本。

脚本安全

我的所有bash脚本都以下面几句为开场白：

```
1  #!/bin/bash
2     set -o nounset
3     set -o errexit
```

这样做会避免两种常见的问题：

1. 引用未定义的变量(缺省值为“”)
2. 执行失败的命令被忽略

需要注意的是，有些Linux命令的某些参数可以强制忽略发生的错误，例如“mkdir -p” 和 “rm -f”。

还要注意的是，在“errexit”模式下，虽然能有效的捕捉错误，但并不能捕捉全部失败的命令，在某些情况下，一些失败的命令是无法检测到的。(更多细节请参考这个帖子。)

脚本函数

在bash里你可以定义函数，它们就跟其它命令一样，可以随意的使用；它们能让你的脚本更具可读性：

```
1     ExtractBashComments() {
2         egrep "^#"
3     }
4
5     cat myscript.sh | ExtractBashComments | wc
6
7     comments=$(ExtractBashComments &lt; myscript.sh)
```

还有一些例子：

```
1      SumLines() {  # iterating over stdin - similar to awk
2          local sum=0
3          local line=””
4          while read line ; do
5              sum=$((${sum} + ${line}))
6          done
7          echo ${sum}
8      }
```

```
 9
10      SumLines &lt; data_one_number_per_line.txt
11
12      log() {  # classic logger
13         local prefix="[$(date +%Y/%m/%d\ %H:%M:%S)]: "
14         echo "${prefix} $@" &gt;&amp;2
15      }
16
17      log "INFO" "a message"
```

尽可能的把你的bash代码移入到函数里，仅把全局变量、常量和对“main”调用的语句放在最外层。

变量注解

Bash里可以对变量进行有限的注解。最重要的两个注解是：

1. `local`(函数内部变量)
2. `readonly`(只读变量)

```
1      # a useful idiom: DEFAULT_VAL can be overwritten
2      #       with an environment variable of the same name
3      readonly DEFAULT_VAL=${DEFAULT_VAL:-7}
4
5      myfunc() {
6         # initialize a local variable with the global default
7         local some_var=${DEFAULT_VAL}
8         ...
9      }
```

这样，你可以将一个以前不是只读变量的变量声明成只读变量：

```
1  x=5
2  x=6
3  readonly x
4  x=7   # failure
```

尽量对你bash脚本里的所有变量使用`local`或`readonly`进行注解。

用`$()`代替反单引号(`)

反单引号很难看，在有些字体里跟正单引号很相似。`$()`能够内嵌使用，而且避免了转义符的麻烦。

```
1  # both commands below print out: A-B-C-D
2  echo "A-`echo B-`echo C-\\`echo D\\````"
3  echo "A-$(echo B-$(echo C-$(echo D)))"
```

用`[[]]`(双层中括号)替代`[]`

使用`[[]]`能避免像异常的文件扩展名之类的问题，而且能带来很多语法上的改进，而且还增加了很多新功能：

操作符	功能说明
||	逻辑or(仅双中括号里使用)
&&	逻辑and(仅双中括号里使用)
<	字符串比较(双中括号里不需要转移)
-lt	数字比较
=	字符串相等
==	以Globbing方式进行字符串比较(仅双中括号里使用，参考下文)
=~	用正则表达式进行字符串比较(仅双中括号里使用，参考下文)
-n	非空字符串
-z	空字符串
-eq	数字相等
-ne	数字不等

单中括号：

```
1 [ "${name}" \&gt; "a" -o ${name} \&lt; "m" ]
```

双中括号

```
1 [[ "${name}" &gt; "a" &amp;&amp; "${name}" &lt; "m"  ]]
```

正则表达式/Globbing

使用双中括号带来的好处用下面几个例子最能表现：

```
1 t="abc123"
2 [[ "$t" == abc* ]]         # true (globbing比较)
```

```
[[ "$t" == "abc*" ]]        # false (字面比较)
[[ "$t" =~ [abc]+[123]+ ]] # true (正则表达式比较)
[[ "$t" =~ "abc*" ]]        # false (字面比较)
```

注意，从bash 3.2版开始，正则表达式和globbing表达式都不能用引号包裹。如果你的表达式里有空格，你可以把它存储到一个变量里：

```
r="a b+"
[[ "a bbb" =~ $r ]]         # true
```

按Globbing方式的字符串比较也可以用到case语句中：

```
case $t in
abc*)  &lt;action&gt; ;;
esac
```

字符串操作

Bash里有各种各样操作字符串的方式，很多都是不可取的。

基本用户

```
    f="path1/path2/file.ext"

    len="${#f}" # = 20 (字符串长度)

    # 切片操作: ${&lt;var&gt;:&lt;start&gt;} or ${&lt;var&gt;:&lt;start&gt;:&lt;length&gt
    slice1="${f:6}" # = "path2/file.ext"
    slice2="${f:6:5}" # = "path2"
    slice3="${f: -8}" # = "file.ext"(注意："-"前有空格)
    pos=6
    len=5
    slice4="${f:${pos}:${len}}" # = "path2"
```

替换操作**(**使用**globbing)**

```
    f="path1/path2/file.ext"

    single_subst="${f/path?/x}"    # = "x/path2/file.ext"
    global_subst="${f//path?/x}"   # = "x/x/file.ext"

    # 字符串拆分
    readonly DIR_SEP="/"
    array=(${f//${DIR_SEP}/ })
    second_dir="${arrray[1]}"      # = path2
```

删除头部或尾部**(**使用**globbing)**

```
f="path1/path2/file.ext"

# 删除字符串头部
extension="${f#*.}"  # = "ext"

# 以贪婪匹配方式删除字符串头部
filename="${f##*/}"  # = "file.ext"

# 删除字符串尾部
dirname="${f%/*}"    # = "path1/path2"

# 以贪婪匹配方式删除字符串尾部
root="${f%%/*}"      # = "path1"
```

避免使用临时文件

有些命令需要以文件名为参数，这样一来就不能使用管道。这个时候 <() 就显出用处了，它可以接受一个命令，并把它转换成可以当成文件名之类的什么东西：

```
# 下载并比较两个网页
diff &lt;(wget -O - url1) &lt;(wget -O - url2)
```

还有一个非常有用处的是"here documents"，它能让你在标准输入上输入多行字符串。下面的'MARKER'可以替换成任何字词。

```
# 任何字词都可以当作分界符
command  &lt;&lt; MARKER
...
${var}
$(cmd)
...
MARKER
```

如果文本里没有内嵌变量替换操作，你可以把第一个MARKER用单引号包起来：

```
command &lt;&lt; 'MARKER'
...
no substitution is happening here.
$ (dollar sign) is passed through verbatim.
...
MARKER
```

内置变量

变量	说明
$0	脚本名称

$n	传给脚本/函数的第n个参数
$$	脚本的PID
$!	上一个被执行的命令的PID(后台运行的进程)
$?	上一个命令的退出状态(管道命令使用${PIPESTATUS})
$#	传递给脚本/函数的参数个数
$@	传递给脚本/函数的所有参数(识别每个参数)
$*	传递给脚本/函数的所有参数(把所有参数当成一个字符串)

提示
使用$*很少是正确的选择。
$@能够处理空格参数，而且参数间的空格也能正确的处理。
使用$@时应该用双引号括起来，像”$@”这样。

调试

对脚本进行语法检查：

```
1 bash -n myscript.sh
```

跟踪脚本里每个命令的执行：

```
1 bash -v myscripts.sh
```

跟踪脚本里每个命令的执行并附加扩充信息：

```
1 bash -x myscript.sh
```

你可以在脚本头部使用`set -o verbose`和`set -o xtrace`来永久指定`-v`和`-o`。当在远程机器上执行脚本时，这样做非常有用，

用它来输出远程信息。

什么时候不应该使用**bash**脚本

- 你的脚本太长，多达几百行
- 你需要比数组更复杂的数据结构
- 出现了复杂的转义问题
- 有太多的字符串操作

不太需要调用其它程序和跟其它程序管道交互

担心性能

这个时候，你应该考虑一种脚本语言，比如Python或Ruby。