Shell
解释脚本命令的程序有
如下几个：
1 #!/bin/sh
2 #!/bin/bash
3 #!/usr/bin/perl
4 #!/usr/bin/tcl
5 #!/bin/sed -f
6 #!/usr/awk -f
上边每一个脚本头的行都指定了一个不同的命令解释器,如果是/bin/sh,那么就是默认shell
(在 Linux 系统中默认是 Bash).

特殊字符

.
1）相当于 source 命令，执行文件
2）作为文件名的一部分，如果放在文件名开头表示隐藏文件。作为目录相对路径，一个点表示当前工作路径，两个点表示上一级路径
3）正则表达式中，用来匹配任何的单个字符命令等价于 source 命令(见 Example 11-20). 这是一个 bash 的内建命令.
‘(强引用) “(弱引用)
sunjohn@sunjohn-X550JD:~$ hell='a b cc   d'
sunjohn@sunjohn-X550JD:~$ echo $hell
a b cc d
sunjohn@sunjohn-X550JD:~$ echo "$hell"
a b cc   d
sunjohn@sunjohn-X550JD:~$ echo '$hell'
$hell
\ 转义字符.
! 取反操作符.
*
1）通配符
2）正则表达式中匹配任意个数的字符
3）在算数操作符上下文中，表示乘法运算
4）**表示幂运算
?
1）$?用来测试一个条件的结果
2）通配符，匹配单个字符的通配符，在正则表达式中表示一个字符
$
1）变量替换。
2）正则里的行结束符
${} 参数替换.
$*,$@ 位置参数.
$? 退出状态码变量.

$$ 进程 id.
{}
1）大括号扩展，如 cat {file1,file2,file3} > combined_file
2）创建一个匿名函数
[]
1）条件测试，相当于 test
2）数组元素
3）正则表达式中描述字符范围
[[]]
使用[[...]]条件判断结构，而不是[...]，能够防止脚本中的许多逻辑错误. 比如,&&, ||, <, 和> 操作符能够正常存在于[[ ]]条件判断结构中，但是如果出现在[ ]结构中的话，会报错.
> &> >& >> < <>
1）重定向
command &>filename 重定向 command 的 stdout 和 stderr 到 filename 中
command >&2 重定向 command 的 stdout 到 stderr 中
[i]<>filename 打开文件 filename 用来读写，并且分配文件描述符 i 给这个文件. 如果 filename 不存在，这个文件将会被创建
2）进程替换。(command)>　　<(command) #这部分不好理解，后续深入研究
3）字符串比较操作，整数比较操作（在双括号中使用）
<< 用在 here document 中的重定向.
| 管道. 分析前边命令的输出，并将输出作为后边命令的输入.
\<, \> 正则表达式中的单词边界.
>| 强制重定向.
|| 或.
& 与-逻辑操作.
&& 与-逻辑操作.
- 算术减号.
= 算术等号,有时也用来比较字符串.
+ 算术加号,也用在正则表达式中.
+ 选项,对于特定的命令来说使用"+"来打开特定的选项,用"-"来关闭特定的选项.
% 算术取模运算.也用在正则表达式中.
~ home 目录.相当于$HOME 变量.~bozo 是 bozo 的 home 目录,并且 ls ~bozo 将列出其中的
~+ 当前工作目录,相当于$PWD 变量.
~- 之前的工作目录,相当于$OLDPWD 内部变量.
=~ 用于正则表达式,这个操作将在正则表达式匹配部分讲解,只有 version3 才支持.
^ 行首,正则表达式中表示行首."^"定位到行首.
注意:命令是不能跟在同一行上注释的后边的,没有办法,在同一行上,注释的后边想要再使用命令,只能另起一行.
当然,在 echo 命令中被转义的#是不能作为注释的.
同样的,#也可以出现在特定的参数替换结构中或者是数字常量表达式中.

控制字符

修改终端或文本显示的行为.控制字符以 CONTROL + key 组合.
控制字符在脚本中不能正常使用.
Ctl-B 光标后退,这应该依赖于 bash 输入的风格,默认是 emacs 风格的.
Ctl-C Break,终止前台工作.
Ctl-D 从当前 shell 登出(和 exit 很像)
"EOF"(文件结束符).这也能从 stdin 中终止输入.
在 console 或者在 xterm window 中输入的时候,Ctl-D 将删除光标下字符.
当没有字符时,Ctrl-D 将退出当前会话.在 xterm window 也有关闭窗口
的效果.
Ctl-G beep.在一些老的终端,将响铃.
Ctl-H backspace,删除光标前边的字符
Ctl-I 就是 tab 键.
Ctl-J 新行.
Ctl-K 垂直 tab.(垂直 tab?新颖,没听过)
作用就是删除光标到行尾的字符.
Ctl-L clear,清屏.
Ctl-M 回车
Ctl-Q 继续(等价于 XON 字符),这个继续的标准输入在一个终端里
Ctl-S 挂起(等价于 XOFF 字符),这个被挂起的 stdin 在一个终端里,用 Ctl-Q 恢复
Ctl-U 删除光标到行首的所有字符,在某些设置下,删除全行.
Ctl-V 当输入字符时,Ctl-V 允许插入控制字符.比如,下边 2 个例子是等价的
echo -e '\x0a'
echo <Ctl-V><Ctl-J>
Ctl-V 在文本编辑器中十分有用,在 vim 中一样.
Ctl-W 删除当前光标到前边的最近一个空格之间的字符.
在某些设置下,删除到第一个非字母或数字的字符.
Ctl-Z 终止前台工作
Ctl-B 光标后退,这应该依赖于 bash 输入的风格,默认是 emacs 风格的.
Ctl-C Break,终止前台工作.
Ctl-D 从当前 shell 登出(和 exit 很像)
Ctl-G beep.在一些老的终端,将响铃.
Ctl-H backspace,删除光标前边的字符
Ctl-I 就是 tab 键.
Ctl-J 新行.
Ctl-K 垂直 tab.(垂直 tab?新颖,没听过)
作用就是删除光标到行尾的字符.
Ctl-L clear,清屏.
Ctl-M 回车
Ctl-Q 继续(等价于 XON 字符),这个继续的标准输入在一个终端里

变量赋值 =

1）赋值操作，等号前后不能有空白
2）如果对变量不使用引号引用，则 echo 时会去掉多余的 tab 和换行符，例如:
a=`ls -l` -----这里是反引号

```
# 把'ls -l'的结果赋值给'a'
 echo $a
# 然而, 如果没有引号的话将会删除 ls 结果中多余的 tab 和换行符.
 echo "$a"
 # 如果加上引号的话, 那么就会保留 ls 结果中的空白符.
```

Bash 变量是不区分类型的

bash 变量不区分类型，本质都是字符串。是否允许整数操作和比较操作，取决于变量中是否只有数字.

特殊的变量类型

1）局部变量。只有在代码块或者函数中才可见的的局部变量 local

2）环境变量。

脚本设置环境变量，需要执行 export 才生效。

3）位置参数。从命令行传递到脚本的参数。

$0 就是脚本文件自身的名字，$1 是第一个参数，$9 之后的位置参数就必须用大括号括起来${10}。$*和$@ 表示所有的位置参数。

4）shift

是把所有的位置参数都向左移动一个位置，$1 <--- $2, $2 <--- $3, $3 <--- $4。原来的$1 就跳出了, 但是$0 (脚本名)是不会改变的。

如果传递了大量的位置参数到脚本中，shift 命令允许你访问的位置参数的数量超过 10 个,当然{}标记法也提供了这样的功能。

转义

转义是一种引用单个字符的方法.一个具有特殊含义的字符前边放上一个转义符(\)就告诉 shell 这个字符失去了特殊的含义.

对于特定的转义符的特殊的含义

在 echo 和 sed 中所使用的

\n 意味着新的一行

\r 回车

\t tab 键

\v vertical tab(垂直 tab),查前边的 Ctl-K

\b backspace,查前边的 Ctl-H

\a "alert"(如 beep 或 flash)

\0xx 转换成 8 进制 ASCII 解码,等价于 oxx

文件测试操作

-e 文件存在

-a 文件存在这个选项的效果与-e 相同.但是它已经被弃用了,并且不鼓励使用

-f file 是一个 regular 文件(不是目录或者设备文件)

-s 文件长度不为 0

-d 文件是个目录

-b 文件是个块设备(软盘,cdrom 等等)

-c 文件是个字符设备(键盘,modem,声卡等等)

-p 文件是个管道
-h 文件是个符号链接
-L 文件是个符号链接
-S 文件是个 socket
-t 关联到一个终端设备的文件描述符
这个选项一般都用来检测是否在一个给定脚本中的 stdin[-t0]或[-t1]是一个终端
-r 文件具有读权限(对于用户运行这个 test)
-w 文件具有写权限(对于用户运行这个 test)
-x 文件具有执行权限(对于用户运行这个 test)
-g set-group-id(sgid)标志到文件或目录上
如果一个目录具有 sgid 标志,那么一个被创建在这个目录里的文件,这个目录属于创建这个目录的用户组,并不一定与创建这个文件的用户的组相同.对于 workgroup 的目录共享来说,这非常有用.见<<UNIX 环境高级编程中文版>>第 58 页.
-u set-user-id(suid)标志到文件上
如果运行一个具有 root 权限的文件,那么运行进程将取得 root 权限,即使你是一个普通用户.[1]这对于需要存取系统硬件的执行操作(比如 pppd 和 cdrecord)非常有用.如果没有 suid 标志的话,那么普通用户(没有 root 权限)将无法运行这种程序.
-k 设置粘贴位,
对于"sticky bit",save-text-mode 标志是一个文件权限的特殊类型.如果设置了这个标志,那
么这个文件将被保存在交换区,为了达到快速存取的目的.如果设置在目录中,它将限制写权限.对于设置了 sticky bit 位的文件或目录,权限标志中有"t".
-O 你是文件的所有者.
-G 文件的 group-id 和你的相同.
-N 从文件最后被阅读到现在,是否被修改.
f1 -nt f2 文件 f1 比 f2 新
f1 -ot f2 f1 比 f2 老
f1 -ef f2 f1 和 f2 都硬连接到同一个文件.
! 非--反转上边测试的结果(如果条件缺席,将返回 true)

其他比较操作
二元比较操作符,比较变量或者比较数字.注意数字与字符串的区别.
整数比较
-eq 等于,如:if ["$a" -eq "$b"]
-ne 不等于,如:if ["$a" -ne "$b"]
-gt 大于,如:if ["$a" -gt "$b"]
-ge 大于等于,如:if ["$a" -ge "$b"]
-lt 小于,如:if ["$a" -lt "$b"]
-le 小于等于,如:if ["$a" -le "$b"]
< 小于(需要双括号),如:(("$a" < "$b"))
<= 小于等于(需要双括号),如:(("$a" <= "$b"))
> 大于(需要双括号),如:(("$a" > "$b"))
>= 大于等于(需要双括号),如:(("$a" >= "$b"))
字符串比较

= 等于,如:if ["$a" = "$b"]
== 等于,如:if ["$a" == "$b"],与=等价
注意:==的功能在[[]]和[]中的行为是不同的,如下:
1) [[$a == z*]] # 如果$a 以"z"开头(模式匹配)那么将为 true
2) [[$a == "z*"]] # 如果$a 等于 z*(字符匹配),那么结果为 true
3) [$a == z*] # File globbing 和 word splitting 将会发生
4) ["$a" == "z*"] # 如果$a 等于 z*(字符匹配),那么结果为 true
!= 不等于,如:if ["$a" != "$b"]
这个操作符将在[[]]结构中使用模式匹配.
< 小于,在 ASCII 字母顺序下.如:
if [["$a" < "$b"]]
if ["$a" \< "$b"]注意:在[]结构中"<"需要被转义.
> 大于,在 ASCII 字母顺序下.如:
if [["$a" > "$b"]]
if ["$a" \> "$b"]
注意:在[]结构中">"需要被转义.
-z 字符串为"null".就是长度为 0.
-n 字符串不为"null"
注意:使用-n 在[]结构中测试必须要用""把变量引起来.使用一个未被""的字符串来使用!
-z
或者就是未用""引用的字符串本身,放到[]结构中

算术操作符
+ 加法
- 减法
* 乘法
/ 除法
** 幂运算
% 取模
+= 加等于(通过常量增加变量)
let "var += 5" #var 将在本身值的基础上增加 5
-= 减等于
*= 乘等于
let "var *= 4"
/= 除等于
%= 取模赋值,算术操作经常使用 expr 或者 let 表达式.

位操作符
<< 左移 1 位(每次左移都将乘 2)
<<= 左移几位,=号后边将给出左移几位
let "var <<= 2"就是左移 2 位(就是乘 4)
>> 右移 1 位(每次右移都将除 2)
>>= 右移几位
& 按位与

&= 按位与赋值
| 按位或
|= 按位或赋值
~ 按位非
! 按位否
^ 按位异或 XOR
^= 异或赋值

特殊参数

$-
传递给脚本的 falg(使用 set 命令).
$!
在后台运行的最后的工作的 PID(进程 ID).
$?
命令,函数或者脚本本身的退出状态(见 Example 23-7)
$$
脚本自身的进程 ID.

测试和循环

if then elif or else：
if 语句测试条件，测试条件返回真（0）或假（1）后，可相应执行一系列语句。 if 语句结构对错误检查非常有用。其格式为：

```
if 条件 1
then 命令 1
elif 条件 2
then 命令 2
else 命令 3
fi
```

==========

If 条件 1 如果条件 1 为真
Then 那么
命令 1 执行命令 1
elif 条件 2 如果条件 1 不成立
then 那么
命令 2 执行命令 2
else 如果条件 1，2 均不

for 循环一般格式为：

```
for 变量名 in 列表
do
命令 1
命令 2
done
```

```
for loop in 1 2 3 4 5
do
  echo $loop
done

for loop in `seq 1 100`
do
  echo $loop
done

for loop in `ls /tmp`
do
   echo $loop
done 成立
命令 3 那么执行命令 3
fi 完成
```

while:
while 循环用于不断执行一系列命令，也用于从输入文件中读取数据，其格式为：

```
while 命令
do
命令 1
命令 2
. . .
done
```

虽然通常只使用一个命令，但在 while 和 do 之间可以放几个命令。命令通常用作测试条件。只有当命令的退出状态为 0 时（若有 exit，则不是 true），do 和 done 之间命令才被执行，如果退出状态不是 0，则循
环终止。命令执行完毕，控制返回循环顶部，从头开始直至测试条件为假。
break 跳出循环。循环终止
continue 循环继续，继续跳到条件判断。重新开始判断。直到这个条件为真。

until：
until 循环执行一系列命令直至条件为真时停止。until 循环与 while 循环在处理方式上刚好
相反。一般 while 循环优于 until 循环，但在某些时候—也只是极少数情况下， until 循环更加
有用。
until 循环格式为：

```
until 条件
命令 1
. . .
done
```

条件可为任意测试条件，测试发生在循环末尾，因此循环至少执行一次—请注意这一

点。

case 语句：
case 语句为多选择语句。可以用 case 语句匹配一个值与一个模式，如果匹配成功，执行相匹配的命令。case 语句格式如下：
case 值 in
模式 1 }
命令 1
. . .
; ;
模式 2)
命令 2
. . .
; ;
esac
case 工作方式如上所示。取值后面必须为单词 in，每一模式必须以右括号结束。取值可以为变量或常数。匹配发现取值符合某一模式后，其间所有命令开始执行直至；；。
取值将检测匹配的每一个模式。一旦模式匹配，则执行完匹配模式相应命令后不再继续其他模式。如果无一匹配模式，使用星号 *捕获该值，再接受其他输入

变量类
let
let 命令将执行变量的算术操作.在许多情况下,它被看作是复杂的 expr 版本的一个简化版.
set
set 命令用来修改内部脚本变量的值.一个作用就是触发选项标志位来帮助决定脚本的行为.另一个应用就是以一个命令的结果(set `command`)来重新设置脚本的位置参数.脚本将会从命令的输出中重新分析出位置参数.
unset
unset 命令用来删除一个 shell 变量,效果就是把这个变量设为 null.注意:这个命令对位置参数无效.
export
export 命令将会使得被 export 的变量在运行的脚本(或 shell)的所有的子进程中都可用.不幸的是,没有办法将变量 export 到父进程(就是调用这个脚本或 shell 的进程)中.
关于 export 命令的一个重要的使用就是用在启动文件中,启动文件是用来初始化并且设置环境变量,让用户进程可以存取环境变量

命令类
Ture
一个返回成功(就是返回 0)退出码的命令,但是除此之外什么事也不做.
Flase
一个返回失败(非 0)退出码的命令,但是除此之外什么事也不做.
type[cmd]
与 which 扩展命令很相像.

bind
bind 内建命令用来显示或修改 readline[5]的键绑定.
help
获得 shell 内建命令的一个小的使用总结.这与 whatis 命令比较象,但是 help 是内建命令.

作业控制命令
jobs
在后台列出所有正在运行的作业,给出作业号.
fg,bg
fg 命令可以把一个在后台运行的作业放到前台来运行.而 bg 命令将会重新启动一个挂起的作业,并且在后台运行它.如果使用 fg 或者 bg 命令的时候没指定作业号,那么默认将对当前正在运行的作业做操作.

作业标识符
%N | 作业号[N]
%S | 以字符串 S 开头的被(命令行)调用的作业
%?S | 包含字符串 S 的被(命令行)调用的作业
%% | 当前作业(前台最后结束的作业,或后台最后启动的作业)
%+ | 当前作业(前台最后结束的作业,或后台最后启动的作业)
%- | 最后的作业
$! | 最后的后台进程

find
-name 按照文件名查找
-perm 按照文件权限来查找
-prune 可以使用 find 命令排除当前文件夹，不查找
-user 可以按照文件属主来查找
-group 可以按照文件数组来查找
-mtime -n +n 按照文件的更改时间来查找
-n 表示文件更改距离在 n 天以内 +n 表示文件更改时间
距离现在 n 天以前.
-nogroup 查找无效所属组的文件
-nouser 查找无效所属主的文件
-type 查找某一类型的文件
b 代表设备块
d 目录
c 字符设备文件
l 符号链接文件
f 普通文件
-size 查找文件长度或者大小
-depth 查找文件时，首先查找当前文件，当目录中的文件，然后再在其子目录当中查找
-mount 在查找文件系统时不跨越 mount 的文件系统
-follow 如果 find 命令遇到符号链接文件，就跟踪链接文件指向的源文件

正则表达式

通配符和正则除了？和*号的用法不同，其他好像都一样：

下面几个符号通配符和正则同样适用：

[a-zA-Z] [a-z0-9A-Z]

ls [^0-9] 匹配非数字 [^]----------非该指定区域

[:space:] 空白字符

[:punct:] 标点符号

[:lower:] 小写字母

[:upper:] 大写字母

[:alpha:] 大小写字母

[:digit:] 数字

[:alnum:] 数字和大小写字母

bash 通配符情况下元字符

*代表任意 0 个或多个

？代表一个

[]括号中的任意一个

[^]括号以外的任意一个

{}，通配符中的花括号是花括号里的都要匹配

%代表匹配最短的 word 字符 %%表示匹配最长的，比如那么我们的$1 就是 /lib/librt.so.1，如果用${1%/*}就会只截取到目录/lib，$1 代表/lib/librt.so.1

匹配 1-255 整数,记住要加括号,因为不加括号,就会只去匹配词首部是[1-9]词尾是 25[0-5], 中间的就会被忽略

egrep '\<([1-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9])|25[0-5])\>'

匹配 ip 地址

ifconfig|egrep

'(\<([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\>\.){3}\<([0-9]|[1-9][1-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\>'

加了括号的就是分组，这里该分组重复三次

只匹配 ABC 类地址

ifconfig|egrep

'\<([0-9]|[1-9][0-9]|1[0-9][0-9]|2[01][0-9]|22[0-3])\>(\.\<([0-9]|[1-9][1-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\>\.){2}\<([0-9]|[1-9][1-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\>'

函数

```
function_name()
{
command ...
}
```

只需要简单的调用函数名，函数就会被调用或触发 .
接收位置参数的函数

```
TWOSUM() {
echo $[$1+$2]
}

SUM=`TWOSUM 5 6`
echo $SUM
~
~
~
```

```
TWOSUM() {
echo $[$1+$2]
}

for I in {1..10}; do
  let  J=$[$I+1]
  TWOSUM $I $J
done
~
~
```

数组

- 声明数组
  - 格式：数组名=（参数1 参数2 ....）
- 多个ip地址存入数组中

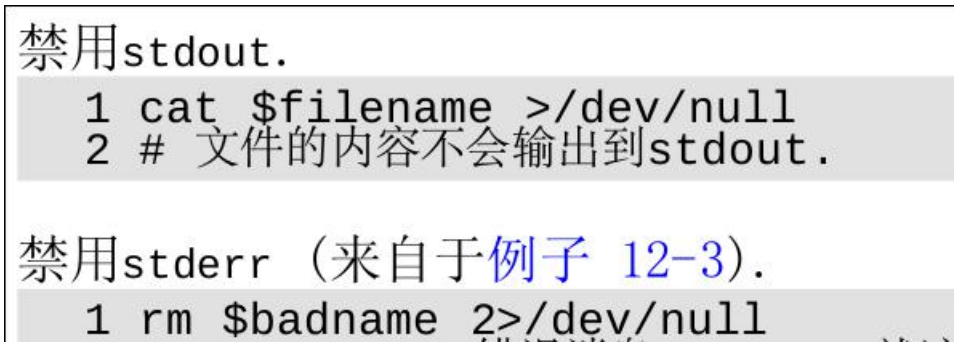
```
[root@localhost ~]# ip=( 192.168.0.1 192.168.0.2
   192.168.0.3 )
[root@localhost ~]# echo ${ip[0]}
192.168.0.1
[root@localhost ~]# echo ${ip[1]}
192.168.0.2
[root@localhost ~]# echo ${ip[2]}
192.168.0.3
[root@localhost ~]# ip[3]=192.168.0.4
[root@localhost ~]# echo ${ip[3]}
192.168.0.4
```

❖ 查看数组里面的元素个数

/dev/zero 与 /dev/null

使用 /dev/null

可以把 /dev/null 想象为一个"黑洞". 它非常接近于一个只写文件. 所有写入它的内容都会永远丢失. 而如果想从它那读取内容, 则什么也读不到. 但是, 对于命令行和脚本来说, /dev/null 却非常的有用



使用 /dev/zero

类似于 /dev/null , /dev/zero 也是一个伪文件, 但事实上它会产生一个 null 流(二进制的 0 流, 而不是 ASCII 类型). 如果你想把其他命令的输出写入它的话, 那么写入的内容会消失, 而且如果你想从 /dev/zero 中读取一连串 null 的话, 也非常的困难, 虽然可以使用 od 或者一个 16 进制编辑器来达到这个目的. /dev/zero 的主要用途就是用来创建一个指定长度, 并且初始化为空的文件, 这种文件一般都用作临时交换文件.