《Bash教程》目录

基础篇

1. Bash简介

Bash是Unix系统和Linux系统的一种Shell（命令行环境），是目前绝大多数Linux发行版的默认Shell。

1.1 Shell的含义

学习Bash，首先需要理解Shell是什么。Shell这个单词的原意是“外壳”，跟kernel（内核）相对应，比喻内核外面的一层，即用户跟内核交互的对话界面。

具体来说，Shell这个词有多种含义。

首先，Shell是一个程序，提供一个与用户对话的环境。这个环境只有一个命令提示符，让用户从键盘输入命令，所以又称为命令行环境（commandline，简写为 CLI）。Shell接收到用户输入的命令，将命令送入操作系统执行，并将结果返回给用户。本书中，除非特别指明，Shell指的就是命令行环境。

其次，Shell是一个命令解释器，解释用户输入的命令。它支持变量、条件判断、循环操作等语法，所以用户可以用Shell命令写出各种小程序，又称为脚本（script）。这些脚本都通过Shell的解释执行，而不通过编译。

最后，Shell是一个工具箱，提供了各种小工具，供用户方便地使用操作系统的功能。

1.2 Shell的种类

Shell有很多种，只要能给用户提供命令行环境的程序，都可以看作是Shell。

历史上，主要的Shell有下面这些。

- BourneShell（sh）
- Bourne Again shell（bash）
- CShell（csh）
- TENEX CShell（tcsh）
- Korn shell（ksh）
- ZShell（zsh）
- Friendly InteractiveShell（fish）

Bash是目前最常用的Shell，除非特别指明，下文的Shell和Bash当作同义词使用，可以互换。

下面的命令可以查看当前运行的Shell。

```
$ echo $SHELL
/bin/bash
```

下面的命令可以查看当前的Linux系统安装的所有Shell。

```
$ cat /etc/shells
```

上面两个命令中，$是命令行环境的提示符，用户只需要输入提示符后面的内容。

Linux 允许每个用户使用不同的Shell，用户的默认Shell一般都是Bash，或者与Bash兼容。

1.3 命令行环境

终端模拟器

如果是不带有图形环境的Linux系统（比如专用于服务器的系统），启动后就直接是命令行环境。

不过，现在大部分的Linux发行版，尤其是针对普通用户的发行版，都是图形环境。用户登录系统后，自动进入图形环境，需要自己启动终端模拟器，才能进入命令行环境。

所谓“终端模拟器”（terminal emulator）就是一个模拟命令行窗口的程序，让用户在一个窗口中使用命令行环境，并且提供各种附加功能，比如调整颜色、字体大小、行距等等。

不同Linux发行版（准确地说是不同的桌面环境）带有的终端程序是不一样的，比如KDE桌面环境的终端程序是konsole，Gnome桌面环境的终端程序是gnome-terminal，用户也可以安装第三方的终端程序。

所有终端程序，尽管名字不同，基本功能都是一样的，就是让用户可以进入命令行环境，使用Shell。

命令行提示符

进入命令行环境以后，用户会看到Shell的提示符。提示符往往是一串前缀，最后以一个美元符号**$**结尾，用户可以在这个符号后面输入各种命令。

```
[user@hostname] $
```

上面例子中，完整的提示符是**[user@hostname] $**，其中前缀是用户名（**user**）加上**@**，再加主机名（**hostname**）。比如，用户名是**bill**，主机名是**home-machine**，前缀就是**bill@home-machine**。

注意，根用户（root）的提示符，不以美元符号（**$**）结尾，而以井号（**#**）结尾，用来提醒用户，现在具有根权限，可以执行各种操作，务必小心，不要出现误操作。这个符号是可以自己定义的，详见《命令提示符》一章。

为了简洁，后文的命令行提示符都只使用**$**表示。

进入和退出方法

进入命令行环境以后，一般就已经打开Bash了。如果你的Shell不是Bash，可以输入**bash**命令启动Bash。

```
$ bash
```

退出Bash环境，可以使用**exit**命令，也可以同时按下**Ctrl + d**。

```
$ exit
```

Bash的基本用法就是在命令行输入各种命令，非常直观。作为练习，可以试着输入**pwd**命令。按下回车键，就会显示当前所在的目录。

```
$ pwd
/home/me
```

如果不小心输入了**pwe**，会返回一个提示，表示输入出错，没有对应的可执行程序。

```
$ pwe
bash: pwe
```

1.4 Shell和Bash的历史

Shell伴随着Unix系统的诞生而诞生。

1969年，Ken Thompson 和 Dennis Ritchie 开发了第一版的Unix。

1971年，Ken Thompson 编写了最初的Shell，称为Thompson shell，程序名是**sh**，方便用户使用Unix。

1973年至1975年间，John R. Mashey 扩展了最初的Thompson shell，添加了编程功能，使得Shell成为一种编程语言。这个版本的Shell称为 Mashey shell。

1976年，Stephen Bourne结合Mashey shell的功能，重写一个新的Shell，称为Bourne shell。

1978年，加州大学伯克利分校的 Bill Joy 开发了 C shell，为Shell提供C语言的语法，程序名是**csh**。它是第一个真正替代**sh**的UNIX shell，被合并到Berkeley UNIX的2BSD版本中。

1979年，UNIX 第七版发布，内置了BourneShell，导致它成为Unix的默认Shell。注意，Thompson shell、 Mashey shell和Bourne shell都是贝尔实验室的产品，程序名都是**sh**。对于用户来说，它们是同一个东西，只是底层代码不同而已。

1983年，David Korn 开发了Korn shell，程序名是**ksh**。

1985年，Richard Stallman 成立了自由软件基金会（FSF），由于Shell的版权属于贝尔公司，所以他决定写一个自由版权的、使用GNU许可证的Shell程序，避免Unix的版权争议。

1988年，自由软件基金会的第一个付薪程序员 Brian Fox 写了一个Shell，功能基本上是Bourne shell的克隆， 叫做Bourne-Again SHell，简称Bash，程序名为**bash**，任何人都可以免费使用。后来，它逐渐成为Linux系统的标准Shell。

1989年，Bash发布1.0版。

1996年，Bash发布2.0版。

2004年，Bash发布3.0版。

2009年，Bash发布4.0版。

2019年，Bash发布5.0版。

用户可以通过环境变量**$BASH_VERSION**查看本机的Bash版本。

```
$ echo $BASH_VERSION
5.0.3(1)-release
```

1.5 更新Bash到最新版

首先查看bash版本。

```
$ echo $BASH_VERSION
4.4.20(1)-release
```

发现是老版本，可通过源码安装。

首先访问http://www.gnu.org/software/bash/bash.html，在DownloadingBash处下载。
一般选择国内镜像如　https://mirrors.ustc.edu.cn/gnu/bash/bash-5.0.tar.gz
。

解压文件并进入目录。

```
$ tar -zxvf bash-5.0.tar.gz
$ cd bash-5.0
```

配置编译选项，生成Makefile。此处–prefix是指定bash安装位置，可修改为你喜欢的任意路径。

```
$ ./configure --prefix=/bin/bash5.0
$ ll Makefile
-rw-rw-r-- 1 shieber shieber 81K 4   16 22:38 Makefile
```

正式编译，需要等待一段时间。

```
$ make
```

安装，此时会安装到–prefix指定的位置，也就是/bin/bash5.0/。

```
$ sudo make install
```

此时系统中有原来的/bin/bash，而新bash在/bin/bash5.0/bin/bash，此时可备份原bash并链接新bash。

```
$ sudo mv /bin/bash /bin/bash.back
$ sudo ln -s /bin/bash5.0/bin/bash /bin/bash
$ reboot #new bash
```

若不需要新bash，则可恢复到原bash。

```
$ sudo mv /bin/bash.back /bin/bash
$ sudo rm -rf /bin/bash5.0 #dangerous cmd, please check the dir.
$ reboot #old bash
```

2. Bash的基本语法

本章介绍Bash的最基本语法。

2.1 echo 命令

由于后面的例子会大量用到echo命令，这里先介绍这个命令。

echo命令的作用是在屏幕输出一行文本，可以将该命令的参数原样输出。

```
$ echo hello world
hello world
```

上面例子中，echo的参数是hello world，可以原样输出。

如果想要输出的是多行文本，即包括换行符。这时需要把多行文本放在引号里面。

```
$ echo "<HTML>
    <HEAD>
          <TITLE>Page Title</TITLE>
    </HEAD>
    <BODY>
          Page body.
    </BODY>
</HTML>"
```

上面例子中，echo可以原样输出多行文本。

-n参数

默认情况下，echo输出的文本末尾会有一个回车符。-n参数可以取消末尾的回车符，使得下一个提示符紧跟在输出内容的后面。

```
$ echo -n hello world
hello world$
```

上面例子中，world后面直接就是下一行的提示符$。

```
$ echo a;echo b
a
b

$ echo -n a;echo b
ab
```

上面例子中，-n参数可以让两个echo命令的输出连在一起，出现在同一行。

-e参数

-e参数会解释引号（双引号和单引号）里面的特殊字符（比如换行符\n）。如果不使用-e参数，即默认情况下，引号会让特殊字符变成普通字符，echo不解释它们，原样输出。

```
$ echo "Hello\nWorld"
Hello\nWorld

#
$ echo -e "Hello\nWorld"
Hello
World

#
$ echo -e 'Hello\nWorld'
Hello
World
```

上面代码中，-e参数使得\n解释为换行符，导致输出内容里面出现换行。

2.2 命令格式

命令行环境中，主要通过使用Shell命令，进行各种操作。Shell命令基本都是下面的格式。

```
$ command [ arg1 ... [ argN ]
```

上面代码中，command是具体的命令或者一个可执行文件，arg1 ... argN是传递给命令的参数，它们是可选的。

```
$ ls -l
```

上面这个命令中，ls是命令，-l是参数。

有些参数是命令的配置项，这些配置项一般都以一个连词线开头，比如上面的-l。同一个配置项往往有长和短两种形式，比如-l是短形式，--list是长形式，它们的作用完全相同。短形式便于手动输入，长形式一般用在脚本之中，可读性更好，利于解释自身的含义。

```
#
$ ls -r

#
$ ls --reverse
```

上面命令中，-r是短形式，--reverse是长形式，作用完全一样。前者便于输入，后者便于理解。

Bash单个命令一般都是一行，用户按下回车键，就开始执行。有些命令比较长，写成多行会有利于阅读和编辑，这时可以在每一行的结尾加上反斜杠，Bash就会将下一行跟当前行放在一起解释。

```
$ echo foo bar

#
$ echo foo \
bar
```

2.3 空格

Bash使用空格（或 Tab 键）区分不同的参数。

```
$ command foo bar
```

上面命令中，foo和bar之间有一个空格，所以Bash认为它们是两个参数。

如果参数之间有多个空格，Bash会自动忽略多余的空格。

```
$ echo this is a     test
this is a test
```

上面命令中，a和test之间有多个空格，Bash会忽略多余的空格。

2.4 分号

分号（;）是命令的结束符，使得一行可以放置多个命令，上一个命令执行结束后，再执行第二个命令。

```
$ clear; ls
```

上面例子中，Bash先执行clear命令，执行完成后，再执行ls命令。

注意，使用分号时，第二个命令总是接着第一个命令执行，不管第一个命令执行成功或失败。

2.5 命令的组合符&&和||

除了分号，Bash还提供两个命令组合符&&和||，允许更好地控制多个命令之间的继发关系。

```
Command1 && Command2
```

上面命令的意思是，如果Command1命令运行成功，则继续运行Command2命令。

```
Command1 || Command2
```

上面命令的意思是，如果Command1命令运行失败，则继续运行Command2命令。

下面是一些例子。

```
$ cat filelist.txt ; ls -l filelist.txt
```

上面例子中，只要cat命令执行结束，不管成功或失败，都会继续执行ls命令。

```
$ cat filelist.txt && ls -l filelist.txt
```

上面例子中，只有cat命令执行成功，才会继续执行ls命令。如果cat执行失败（比如不存在文件flielist.txt），那么ls命令就不会执行。

```
$ mkdir foo || mkdir bar
```

上面例子中，只有mkdir foo命令执行失败（比如foo目录已经存在），才会继续执行mkdir bar命令。如果mkdir foo命令执行成功，就不会创建bar目录了。

2.6 type 命令

Bash本身内置了很多命令，同时也可以执行外部程序。怎么知道一个命令是内置命令，还是外部程序呢？

type命令用来判断命令的来源。

```
$ type echo
echo is a shell builtin
$ type ls
ls is hashed (/bin/ls)
```

上面代码中，type命令告诉我们，echo是内部命令，ls是外部程序（/bin/ls）。

type命令本身也是内置命令。

```
$ type type
type is a shell builtin
```

如果要查看一个命令的所有定义，可以使用type命令的-a参数。

```
$ type -a echo
echo is shell builtin
```

```
echo is /usr/bin/echo
echo is /bin/echo
```

上面代码表示，echo命令即是内置命令，也有对应的外部程序。

type命令的-t参数，可以返回一个命令的类型：别名（alias），关键词（keyword），函数（function），内置命令（builtin）和文件（file）。

```
$ type -t bash
file
$ type -t if
keyword
```

上面例子中，bash是文件，if是关键词。

2.7 快捷键

Bash提供很多快捷键，可以大大方便操作。下面是一些最常用的快捷键，完整的介绍参见《行操作》一章。

- Ctrl + L：清除屏幕并将当前行移到页面顶部。
- Ctrl + C：中止当前正在执行的命令。
- Shift + PageUp：向上滚动。
- Shift + PageDown：向下滚动。
- Ctrl + U：从光标位置删除到行首。
- Ctrl + K：从光标位置删除到行尾。
- Ctrl + D：关闭Shell会话。
- Ctrl + Shift + V：将剪贴板内容粘贴到此处。
- ↑，↓：浏览已执行命令的历史记录。

除了上面的快捷键，Bash还具有自动补全功能。命令输入到一半的时候，可以按下Tab键，Bash会自动完成剩下部分比如，输入pw，然后按一下 Tab 键，Bash会自动补上d。

除了命令的自动补全，Bash还支持路径的自动补全。有时，需要输入很长的路径，这时只需要输入前面的部分，然后按下Tab键，就会自动补全后面部分。如有多个可能的选择，按两次Tab键，Bash会显示所有选项，让你选择。

3. Bash的模式扩展

3.1 简介

Shell接收到用户输入的命令以后，会根据空格将用户的输入，拆分成一个个词元（token）。
然后，Shell会扩展词元里面的特殊字符，扩展完成后才会调用相应的命令。

这种特殊字符的扩展，称为通配符扩展（wildcard expansion）或者模式扩展（globbing）。
Bash一共提供八种扩展，先进行扩展，然后再执行命令。

- 波浪线扩展
- ? 字符扩展
- * 字符扩展
- 方括号扩展
- 大括号扩展

- 变量扩展
- 子命令扩展
- 算术扩展

本章介绍这八种扩展。

Bash是先进行扩展，再执行命令。因此，扩展的结果是由Bash负责的，与所要执行的命令无关。命令本身并不存在参数扩展，收到什么参数就原样执行。这一点务必需要记住。

globbing这个词，来自于早期的Unix系统有一个/etc/glob文件，保存扩展的模板。后来Bash内置了这个功能，但是这个名字就保留了下来。

模式扩展与正则表达式的关系是，模式扩展早于正则表达式出现，可以看作是原始的正则表达式。它的功能没有正则那么强大灵活，但是优点是简单和方便。

Bash允许用户关闭通配符扩展。

```
$ set -o noglob
#
$ set -f
```

下面的命令可以重新打开通配符扩展。

```
$ set +o noglob
#
$ set +f
```

3.2 波浪线扩展

波浪线~会自动扩展成当前用户的主目录。

```
$ echo ~
/home/me
```

~/dir表示扩展成主目录的某个子目录，dir是主目录里面的一个子目录名。

```
#   /home/me/foo
$ cd ~/foo
```

~user表示扩展成用户user的主目录。

```
$ echo ~foo
/home/foo

$ echo ~root
/root
```

上面例子中，Bash会根据波浪号后面的用户名，返回该用户的主目录。

如果~user的user是不存在的用户名，则波浪号扩展不起作用。

```
$ echo ~nonExistedUser
~nonExistedUser
```

~+会扩展成当前所在的目录，等同于pwd命令。

```
$ cd ~/foo
$ echo ~+
/home/me/foo
```

3.3 ? 字符扩展

?字符代表文件路径里面的任意单字符，不含空字符。比如，Data???匹配所有Data后面跟着三个字符的文件名。

```
#    a.txt   b.txt
$ ls ?.txt
a.txt b.txt
```

上面命令中，?表示单个字符，所以会同时匹配a.txt和b.txt。

如果匹配多个字符，就需要多个?连用。

```
#    a.txt b.txt   ab.txt
$ ls ??.txt
ab.txt
```

上面命令中，??匹配了两个字符。

? 字符扩展属于文件名扩展，只有文件确实存在的前提下，才会发生扩展。如果文件不存在，扩展就不会发生。

```
#    a.txt
$ echo ?.txt
a.txt

#
$ echo ?.txt
?.txt
```

上面例子中，如果?.txt可以扩展成文件名，echo命令会输出扩展后的结果；如果不能扩展成文件名，echo就会原样输出?.txt。

3.4 * 字符扩展

*字符代表文件路径里面的任意数量的字符，包括零个字符。

```
#    a.txt b.txt   ab.txt
$ ls *.txt
a.txt b.txt ab.txt

#
$ ls *
```

下面是*匹配空字符的例子。

```
#    a.txt b.txt   ab.txt
$ ls a*.txt
```

```
a.txt ab.txt

$ ls *b*
b.txt ab.txt
```

注意，*不会匹配隐藏文件（以.开头的文件）。

```
#
$ echo .*

#
#          .   ..
$ echo .[!.]*
```

*字符扩展也属于文件名扩展，只有文件确实存在的前提下才会扩展。如果文件不存在，就会原样输出。

```
#      c
$ echo c*.txt
c*.txt
```

上面例子中，当前目录里面没有c开头的文件，导致c*.txt会原样输出。

3.5 方括号扩展

方括号扩展的形式是[...]，只有文件确实存在的前提下才会扩展。如果文件不存在，就会原样输出。
括号之中的任意一个字符。比如，[aeiou]可以匹配五个元音字母中的任意一个。

```
#    a.txt   b.txt
$ ls [ab].txt
a.txt b.txt

#     a.txt
$ ls [ab].txt
a.txt
```

上面例子中，[ab]可以匹配a或b，前提是确实存在相应的文件。

方括号扩展属于文件名匹配，即扩展后结果必须符合现有的文件路径。如果不存在匹配，就会保持原样，不扩展。

```
#     a.txt   b.txt
$ ls [ab].txt
ls:   '[ab].txt':
```

上面例子中，由于扩展后的文件不存在，[ab].txt就原样输出了，导致ls命名报错。

方括号扩展还有两种变体：[^...]和[!...]。它们表示匹配不在方括号里面的字符，这两种写法是等价的。
比如，[^abc]或[!abc]表示匹配除了a、b、c以外的字符。

```
#   aaa bbb aba
$ ls ?[!a]?
aba bbb
```

上面命令中，[!a]表示文件名第二个字符不是a的文件名，所以返回了aba和bbb两个文件。

注意，如果需要匹配[字符，可以放在方括号内，比如[[aeiou]。如果需要匹配连字号-，只能放在方括号内部的开头或结尾，比如[-aeiou]或[aeiou-]。

3.6 [start-end] 扩展

方括号扩展有一个简写形式[start-end]，表示匹配一个连续的范围。比如，[a-c]等同于[abc]，[0-9]匹配[0123456789]。

```
#    a.txt b.txt   c.txt
$ ls [a-c].txt
a.txt
b.txt
c.txt

#    report1.txt report2.txt   report3.txt
$ ls report[0-9].txt
report1.txt
report2.txt
report3.txt
...
```

下面是一些常用简写的例子。

- [a-z]：所有小写字母。
- [a-zA-Z]：所有小写字母与大写字母。
- [a-zA-Z0-9]：所有小写字母、大写字母与数字。
- [abc]*：所有以a、b、c字符之一开头的文件名。
- program.[co]：文件program.c与文件program.o。
- BACKUP.[0-9][0-9][0-9]：所有以BACKUP.开头，后面是三个数字的文件名。

这种简写形式有一个否定形式[!start-end]，表示匹配不属于这个范围的字符。比如，[!a-zA-Z]表示匹配非英文字母的字符。

```
$ echo report[!1-3].txt
report4.txt report5.txt
```

上面代码中，[!1-3]表示排除1、2和3。

3.7 大括号扩展

大括号扩展{...}表示分别扩展成大括号里面的所有值，各个值之间使用逗号分隔。
比如，{1,2,3}扩展成1 2 3。

```
$ echo {1,2,3}
1 2 3

$ echo d{a,e,i,u,o}g
dag deg dig dug dog
```

```
$ echo Front-{A,B,C}-Back
Front-A-Back Front-B-Back Front-C-Back
```

注意，大括号扩展不是文件名扩展。它会扩展成所有给定的值，而不管是否有对应的文件存在。

```
$ ls {a,b,c}.txt
ls:   'a.txt':
ls:   'b.txt':
ls:   'c.txt':
```

上面例子中，即使不存在对应的文件，{a,b,c}依然扩展成三个文件名，导致ls命令报了三个错误。

另一个需要注意的地方是，大括号内部的逗号前后不能有空格。否则，大括号扩展会失效。

```
$ echo {1 , 2}
{1 , 2}
```

上面例子中，逗号前后有空格，Bash就会认为这不是大括号扩展，而是三个独立的参数。

逗号前面可以没有值，表示扩展的第一项为空。

```
$ cp a.log{,.bak}

#
# cp a.log a.log.bak
```

大括号可以嵌套。

```
$ echo {j{p,pe}g,png}
jpg jpeg png

$ echo a{A{1,2},B{3,4}}b
aA1b aA2b aB3b aB4b
```

大括号也可以与其他模式联用，并且总是先于其他模式进行扩展。

```
$ echo {cat,d*}
cat dawg dg dig dog doug dug
```

上面例子中，会先进行大括号扩展，然后进行*扩展。

大括号可以用于多字符的模式，方括号不行（只能匹配单字符）。

```
$ echo {cat,dog}
cat dog
```

由于大括号扩展{...}不是文件名扩展，所以它总是会扩展的。这与方括号扩展[...]完全不同，如果匹配的文件不存在，方括号就不会扩展。这一点要注意区分。

```
#    a.txt   b.txt
$ echo [ab].txt
[ab].txt
```

```
$ echo {a,b}.txt
a.txt b.txt
```

上面例子中，如果不存在a.txt和b.txt，那么[ab].txt就会变成一个普通的文件名，而{a,b}.txt可以照样扩展。

3.8 {start..end} 扩展

大括号扩展可简写为{start..end}，表示扩展成一个连续序列。比如，{a..z}可以扩展成26个小写英文字母。

```
$ echo {a..c}
a b c

$ echo d{a..d}g
dag dbg dcg ddg

$ echo {1..4}
1 2 3 4

$ echo Number_{1..5}
Number_1 Number_2 Number_3 Number_4 Number_5
```

这种简写形式支持逆序。

```
$ echo {c..a}
c b a

$ echo {5..1}
5 4 3 2 1
```

注意，如果遇到无法理解的简写，大括号模式就会原样输出，不会扩展。

```
$ echo {a1..3c}
{a1..3c}
```

这种简写形式可以嵌套使用，形成复杂的扩展。

```
$ echo .{mp{3..4},m4{a,b,p,v}}
.mp3 .mp4 .m4a .m4b .m4p .m4v
```

大括号扩展的常见用途为新建一系列目录。

```
$ mkdir {2007..2009}-{01..12}
```

上面命令会新建36个子目录，每个子目录的名字都是”年份-月份“。

这个写法的另一个常见用途，是直接用于for循环。

```
for i in {1..4}
do
  echo $i
done
```

上面例子会循环4次。

如果整数前面有前导0，扩展输出的每一项都有前导0。

```
$ echo {01..5}
01 02 03 04 05

$ echo {001..5}
001 002 003 004 005
```

这种简写形式还可以使用第二个双点号（start..end..step），用来指定扩展的步长。

```
$ echo {0..8..2}
0 2 4 6 8
```

上面代码将0扩展到8，每次递增的长度为2，所以一共输出5个数字。

多个简写形式连用，会有循环处理的效果。

```
$ echo {a..c}{1..3}
a1 a2 a3 b1 b2 b3 c1 c2 c3
```

3.9 变量扩展

Bash将美元符号$开头的词元视为变量，将其扩展成变量值，详见《Bash变量》一章。

```
$ echo $SHELL
/bin/bash
```

变量名除了放在美元符号后面，也可以放在${}里面。

```
$ echo ${SHELL}
/bin/bash
```

${!string*}或${!string@}返回所有匹配给定字符串string的变量名。

```
$ echo ${!S*}
SECONDS SHELL SHELLOPTS SHLVL SSH_AGENT_PID SSH_AUTH_SOCK
```

上面例子中，${!S*}扩展成所有以S开头的变量名。

3.10 子命令扩展

$(...)可以扩展成另一个命令的运行结果，该命令的所有输出都会作为返回值。

```
$ echo $(date)
Tue Jan 28 00:01:13 CST 2020
```

上面例子中，$(date)返回date命令的运行结果。

还有另一种较老的语法，子命令放在反引号之中，也可以扩展成命令的运行结果。

```
$ echo `date`
Tue Jan 28 00:01:13 CST 2020
```

$(...)可以嵌套，比如$(ls $(pwd))。

3.11 算术扩展

$((...))可以扩展成整数运算的结果，详见《Bash的算术运算》一章。

```
$ echo $((2 + 2))
4
```

3.12 字符类

[[:class:]]表示一个字符类，扩展成某一类特定字符之中的一个。常用的字符类如下。

- [[:alnum:]]：匹配任意英文字母与数字
- [[:alpha:]]：匹配任意英文字母
- [[:blank:]]：空格和 Tab 键。
- [[:cntrl:]]：ASCII 码 0-31 的不可打印字符。
- [[:digit:]]：匹配任意数字 0-9。
- [[:graph:]]：A-Z、a-z、0-9 和标点符号。
- [[:lower:]]：匹配任意小写字母 a-z。
- [[:print:]]：ASCII 码 32-127 的可打印字符。
- [[:punct:]]：标点符号（除了 A-Z、a-z、0-9 的可打印字符）。
- [[:space:]]：空格、Tab、LF（10）、VT（11）、FF（12）、CR（13）。
- [[:upper:]]：匹配任意大写字母 A-Z。
- [[:xdigit:]]：16进制字符（A-F、a-f、0-9）。

请看下面的例子。

```
$ echo [[:upper:]]*
```

上面命令输出所有大写字母开头的文件名。

字符类的第一个方括号后面，可以加上感叹号!，表示否定。比如，[![:digit:]]匹配所有非数字。

```
$ echo [![:digit:]]*
```

上面命令输出所有不以数字开头的文件名。

字符类也属于文件名扩展，如果没有匹配的文件名，字符类就会原样输出。

```
#
$ echo [[:upper:]]*
[[:upper:]]*
```

上面例子中，由于没有可匹配的文件，字符类就原样输出了。

3.13 使用注意点

通配符有一些使用注意点，不可不知。

（1）通配符是先解释，再执行。

Bash接收到命令以后，发现里面有通配符，会进行通配符扩展，然后再执行命令。

```
$ ls a*.txt
ab.txt
```

上面命令的执行过程是，Bash先将a*.txt扩展成ab.txt，然后再执行ls ab.txt。

（2）文件名扩展在不匹配时，会原样输出。

文件名扩展在没有可匹配的文件时，会原样输出。

```
#    r
$ echo r*
r*
```

上面代码中，由于不存在r开头的文件名，r*会原样输出。

下面是另一个例子。

```
$ ls *.csv
ls: *.csv: No such file or directory
```

另外，前面已经说过，大括号扩展{...}不是文件名扩展。

（3）只适用于单层路径。

所有文件名扩展只匹配单层路径，不能跨目录匹配，即无法匹配子目录里面的文件。或者说，?或*这样的通配符，不能匹配路径分隔符（/）。

如果要匹配子目录里面的文件，可以写成下面这样。

```
$ ls */*.txt
```

（4）文件名可以使用通配符。

Bash允许文件名使用通配符，即文件名包括特殊字符。这时引用文件名，需要把文件名放在单引号里面。

```
$ touch 'fo*'
$ ls
fo*
```

上面代码创建了一个fo*文件，这时*就是文件名的一部分。

3.14 量词语法

量词语法用来控制模式匹配的次数。它只有在Bash的extglob参数打开的情况下才能使用，不过一般默认打开。下面的命令可以查询。

```
$ shopt extglob
extglob         on
```

量词语法有下面几个。

- ?(pattern-list)：匹配零个或一个模式。
- *(pattern-list)：匹配零个或多个模式。
- +(pattern-list)：匹配一个或多个模式。
- @(pattern-list)：只匹配一个模式。

- !(pattern-list)：匹配零个或一个以上的模式，但不匹配单独一个的模式。

```
$ ls abc?(.)txt
abctxt abc.txt
```

上面例子中，?(.)匹配零个或一个点。

```
$ ls abc?(def)
abc abcdef
```

上面例子中，?(def)匹配零个或一个def。

```
$ ls abc+(.txt|.php)
abc.php abc.txt
```

上面例子中，+(.txt|.php)匹配文件有一个.txt或.php后缀名。

```
$ ls abc+(.txt)
abc.txt abc.txt.txt
```

上面例子中，+(.txt)匹配文件有一个或多个.txt后缀名。

量词语法也属于文件名扩展，如果不存在可匹配的文件，就会原样输出。

```
#   abc
$ ls abc?(def)
ls:   'abc?(def)':
```

上面例子中，由于没有可匹配的文件，abc?(def)就原样输出，导致ls命令报错。

3.15 shopt 命令

shopt命令可以调整Bash的行为。它有好几个参数跟通配符扩展有关。

shopt命令的使用方法如下。

```
#
$ shopt -s [optionname]

#
$ shopt -u [optionname]

#
$ shopt [optionname]
```

（1）dotglob 参数

dotglob参数可以让扩展结果包括隐藏文件（即点开头的文件）。

正常情况下，扩展结果不包括隐藏文件。

```
$ ls *
abc.txt
```

打开dotglob，就会包括隐藏文件。

```
$ shopt -s dotglob
$ ls *
abc.txt .config
```

（2）nullglob 参数

nullglob参数可以让通配符不匹配任何文件名时，返回空字符。

默认情况下，通配符不匹配任何文件名时，会保持不变。

```
$ rm b*
rm:   'b*':
```

上面例子中，由于当前目录不包括b开头的文件名，导致b*不会发生文件名扩展，保持原样不变，所以rm命令报错没有b*这个文件。

打开nullglob参数，就可以让不匹配的通配符返回空字符串。

```
$ shopt -s nullglob
$ rm b*
rm:
```

上面例子中，由于没有b*匹配的文件名，所以rm b*扩展成了rm，导致报错变成了”缺少操作数“。

（3）failglob 参数

failglob参数使得通配符不匹配任何文件名时，Bash会直接报错，而不是让各个命令去处理。

```
$ shopt -s failglob
$ rm b*
bash:   : b*
```

上面例子中，打开failglob以后，由于b*不匹配任何文件名，Bash直接报错了，不再让rm命令去处理。

（4）extglob 参数

extglob参数使得Bash支持 ksh 的一些扩展语法。它默认应该是打开的。

```
$ shopt extglob
extglob         on
```

它的主要应用是支持量词语法。如果不希望支持量词语法，可以用下面的命令关闭。

```
$ shopt -u extglob
```

（5）nocaseglob 参数

nocaseglob参数可以让通配符扩展不区分大小写。

```
$ shopt -s nocaseglob
$ ls /windows/program*
/windows/ProgramData
/windows/Program Files
/windows/Program Files (x86)
```

上面例子中，打开nocaseglob以后，program*就不区分大小写了，可以匹配ProgramData等。

3.16 参考链接

- Think You Understand Wildcards? Think Again
- Advanced Wildcard Patterns Most People Don’ t Know

4. 引号和转义

Bash只有一种数据类型，就是字符串。不管用户输入什么数据，Bash都视为字符串。
因此，字符串相关的引号和转义，对Bash来说就非常重要。

4.1 转义

某些符号在Bash里面有特殊含义（比如$、&、&）。

```
$ echo $date

$
```

上面例子中，输出$date不会有任何结果，因为$是一个特殊字符。

如果想原样输出这些特殊字符，就必须在前面加上反斜杠，使其变成普通字符。这就叫做“转义”（escape）。

```
$ echo \$date
$date
```

上面命令中，只有在特殊字符$前面加反斜杠，才能原样输出。

反斜杠本身也是特殊字符，如果想要原样输出反斜杠，就需要对它自身转义，连续使用两个反斜线（\\）。

```
$ echo \\
\
```

上面例子输出了反斜杠本身。

反斜杠除了用于转义，还可以表示一些不可打印的字符。

- \a：响铃
- \b：退格
- \n：换行
- \r：回车
- \t：制表符

如果想要在命令行使用这些不可打印的字符，可以把它们放在引号里面，然后使用echo命令的-e参数。

```
$ echo a\tb
atb

$ echo -e "a\tb"
a       b
```

上面例子中，命令行直接输出不可打印字符，Bash不能正确解释。必须把它们放在引号之中，
然后使用echo命令的-e参数。

由于反斜杠可以对换行符转义，使得Bash认为换行符是一个普通字符，从而可以将一行命令写成多行。

```
$ mv \
/path/to/foo \
/path/to/bar

#
$ mv /path/to/foo /path/to/bar
```

上面例子中，如果一条命令过长，就可以在行尾使用反斜杠，将其改写成多行。这是常见的多行命令的写法。

4.2 单引号

Bash允许字符串放在单引号或双引号之中，加以引用。

单引号用于保留字符的字面含义，各种特殊字符在单引号里面，都会变为普通字符，比如星号（*）、美元符号（$）、反斜杠（\）等。

```
$ echo '*'
*

$ echo '$USER'
$USER

$ echo '$((2+2))'
$((2+2))

$ echo '$(echo foo)'
$(echo foo)
```

上面命令中，单引号使得Bash扩展、变量引用、算术运算和子命令，都失效了。如果不使用单引号，它们都会被Bash自动扩展。

由于反斜杠在单引号里面变成了普通字符，所以如果单引号之中，还要使用单引号，不能使用转义，需要在外层的单引号前面加上一个美元符号（$），然后再对里层的单引号转义。

```
#
$ echo it's

#
$ echo 'it\'s'

#
$ echo $'it\'s'
```

不过，更合理的方法是改在双引号之中使用单引号。

```
$ echo "it's"
it's
```

4.3 双引号

双引号比单引号宽松，可以保留大部分特殊字符的本来含义，但是三个字符除外：美元符号$、反引号和反斜杠（\）。也就是说，这三个字符在双引号之中，会被Bash自动扩展。

```
$ echo "*"
*
```

上面例子中，通配符*放在双引号之中，就变成了普通字符，会原样输出。这一点需要特别留意，双引号里面不会进行文件名扩展。

```
$ echo "$SHELL"
/bin/bash

$ echo "`date`"
Mon Jan 27 13:33:18 CST 2020
```

上面例子中，美元符号和反引号在双引号中，都保持特殊含义。美元符号用来引用变量，反引号则是执行子命令。

```
$ echo "I'd say: \"hello!\""
I'd say: "hello!"

$ echo "\\"
\
```

上面例子中，反斜杠在双引号之中保持特殊含义，用来转义。所以，可以使用反斜杠，在双引号之中插入双引号，或者插入反斜杠本身。

由于双引号将换行符解释为普通字符，所以可以利用双引号，在命令行输入多行文本。

```
$ echo "hello
world"
hello
world
```

上面命令中，Bash正常情况下会将换行符解释为命令结束，但是换行符在双引号之中就是普通字符，所以可以输入多行。echo命令会将换行符原样输出，显示的时候正常解释为换行。

双引号的另一个常见的使用场合是，文件名包含空格。这时就必须使用双引号，将文件名放在里面。

```
$ ls "two words.txt"
```

上面命令中，two words.txt是一个包含空格的文件名，否则就会被Bash当作两个文件。

双引号会原样保存多余的空格。

```
$ echo "this is a     test"
this is a     test
```

双引号还有一个作用，就是保存原始命令的输出格式。

```
#
$ echo $(cal)
  2020           1 2 3 ... 31
```

```
#
$ echo "$(cal)"
        2020

          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

上面例子中，如果`$(cal)`不放在双引号之中，`echo`就会将所有结果以单行输出，丢弃了所有原始的格式。

4.4 Here 文档

Here 文档（here document）是一种输入多行字符串的方法，格式如下。

```
<< token
text
token
```

它的格式分成开始标记（`<< token`）和结束标记（`token`）。开始标记是两个小于号
+ Here 文档的名称，名称可以随意取；结束标记是单独一行的 Here 文档名称。两者之间就是多行字符串的内容。

下面是一个通过 Here 文档输出 HTML 代码的例子。

```
$ cat << _EOF_
<html>
<head>
    <title>
    The title of your page
    </title>
</head>

<body>
    Your page content goes here.
</body>
</html>
_EOF_
```

Here 文档内部会发生变量替换和通配符扩展，但是双引号和单引号都失去语法作用，变成了普通字符。

```
$ foo='hello world'
$ cat << _example_
$foo
"$foo"
'$foo'
_example_
```

```
hello world
"hello world"
'hello world'
```

上面例子中，变量`$foo`发生了替换，但是双引号和单引号都原样输出了，表明它们已经失去了引用的功能。

如果不希望发生变量替换和通配符扩展，可以把 Here 文档的开始标记放在单引号之中。

```
$ foo='hello world'
$ cat << '_example_'
$foo
"$foo"
'$foo'
_example_

$foo
"$foo"
'$foo'
```

上面例子中，Here 文档的开始标记（`_example_`）放在单引号之中，导致变量替换失效了。

Here 文档的本质是重定向，它将字符串重定向输出给某个命令，相当于包含了`echo`命令。

```
$ command << token
  string
token

#

$ echo string | command
```

上面代码中，Here 文档相当于`echo`命令的重定向。

所以，Here 字符串只适合那些可以接受标准输入作为参数的命令，对于其他命令无效，比如`echo`命令就不能用 Here 文档作为参数。

```
$ echo << _example_
hello
_example_
```

上面例子不会有任何输出，因为 Here 文档对于`echo`命令无效。

此外，Here 文档也不能作为变量的值，只能用于命令的参数。

Here 字符串

Here 文档还有一个变体，叫做 Here 字符串（Here string），使用三个小于号（`<<<`）表示。

```
<<< string
```

它的作用是将字符串通过标准输入，传递给命令。

有些命令直接接受给定的参数，与通过标准输入接受参数，结果是不一样的。所以才有了这个语法，使得将字符串通过标准输入传递给命令更方便，比如cat命令只接受标准输入传入的字符串。

```
$ cat <<< 'hi there'
#
$ echo 'hi there' | cat
```

上面的第一种语法使用了 Here 字符串，要比第二种语法看上去语义更好，也更简洁。

```
$ md5sum <<< 'ddd'
#
$ echo 'ddd' | md5sum
```

上面例子中，md5sum命令只能接受标准输入作为参数，不能直接将字符串放在命令后面，会被当作文件名，即md5sum ddd里面的ddd会被解释成文件名。这时就可以用 Here 字符串，将字符串传给md5sum命令。

5. Bash变量

Bash变量分成环境变量和自定义变量两类。

5.1 简介

环境变量是Bash环境自带的变量，进入Shell时已经定义好了，可以直接使用。它们通常是系统定义好的，也可以由用户从父Shell传入子Shell。

env命令或printenv命令，可以显示所有环境变量。

```
$ env
#
$ printenv
```

下面是一些常见的环境变量。

- BASHPID：Bash进程的进程 ID。
- BASHOPTS：当前Shell的参数，可以用shopt命令修改。
- DISPLAY：图形环境的显示器名字，通常是:0，表示 X Server 的第一个显示器。
- EDITOR：默认的文本编辑器。
- HOME：用户的主目录。
- HOST：当前主机的名称。
- IFS：词与词之间的分隔符，默认为空格。
- LANG：字符集以及语言编码，比如zh_CN.UTF-8。
- PATH：由冒号分开的目录列表，当输入可执行程序名后，会搜索这个目录列表。
- PS1：Shell提示符。
- PS2： 输入多行命令时，次要的Shell提示符。
- PWD：当前工作目录。
- RANDOM：返回一个0到32767之间的随机数。
- SHELL：Shell的名字。
- SHELLOPTS：启动当前Shell的set命令的参数，参见《set 命令》一章。
- TERM：终端类型名，即终端仿真器所用的协议。

- UID：当前用户的 ID 编号。
- USER：当前用户的用户名。

很多环境变量很少发生变化，而且是只读的，可以视为常量。由于它们的变量名全部都是大写，所以传统上，如果用户要自己定义一个常量，也会使用全部大写的变量名。

注意，Bash变量名区分大小写，HOME和home是两个不同的变量。

查看单个环境变量的值，可以使用printenv命令或echo命令。

```
$ printenv PATH
#
$ echo $PATH
```

注意，printenv命令后面的变量名，不用加前缀$。

自定义变量是用户在当前Shell里面自己定义的变量，必须先定义后使用，而且仅在当前Shell可用。一旦退出当前Shell，该变量就不存在了。

set命令可以显示所有变量（包括环境变量和自定义变量），以及所有的Bash函数。

```
$ set
```

5.2 创建变量

用户创建变量的时候，变量名必须遵守下面的规则。

- 字母、数字和下划线字符组成。
- 第一个字符必须是一个字母或一个下划线，不能是数字。
- 不允许出现空格和标点符号。

变量声明的语法如下。

```
variable=value
```

上面命令中，等号左边是变量名，右边是变量。注意，等号两边不能有空格。

如果变量的值包含空格，则必须将值放在引号中。

```
myvar="hello world"
```

Bash没有数据类型的概念，所有的变量值都是字符串。

下面是一些自定义变量的例子。

```
a=z                     #   a     z
b="a string"            #
c="a string and $b"     #
d="\t\ta string\n"      #
e=$(ls -l foo.txt)      #
f=$((5 * 7))            #
```

变量可以重复赋值，后面的赋值会覆盖前面的赋值。

```
$ foo=1
$ foo=2
$ echo $foo
2
```

上面例子中，变量foo的第二次赋值会覆盖第一次赋值。

5.3 读取变量

读取变量的时候，直接在变量名前加上$就可以了。

```
$ foo=bar
$ echo $foo
bar
```

每当Shell看到以$开头的单词时，就会尝试读取这个变量名对应的值。

如果变量不存在，Bash不会报错，而会输出空字符。

由于$在Bash中有特殊含义，把它当作美元符号使用时，一定要非常小心，

```
$ echo The total is $100.00
The total is 00.00
```

上面命令的原意是输入$100，但是Bash将$1解释成了变量，该变量为空，因此输入就变成了00.00。
所以，如果要使用$的原义，需要在$前面放上反斜杠，进行转义。

```
$ echo The total is \$100.00
The total is $100.00
```

读取变量的时候，变量名也可以使用花括号{}包围，比如$a也可以写成${a}。
这种写法可以用于变量名与其他字符连用的情况。

```
$ a=foo
$ echo $a_file

$ echo ${a}_file
foo_file
```

上面代码中，变量名a_file不会有任何输出，因为Bash将其整个解释为变量，而这个变量是不存在的。
只有用花括号区分$a，Bash才能正确解读。

事实上，读取变量的语法$foo，可以看作是${foo}的简写形式。

如果变量的值本身也是变量，可以使用${!varname}的语法，读取最终的值。

```
$ myvar=USER
$ echo ${!myvar}
ruanyf
```

上面的例子中，变量myvar的值是USER，${!myvar}的写法将其展开成最终的值。

5.4 删除变量

unset命令用来删除一个变量。

```
unset NAME
```

这个命令不是很有用。因为不存在的Bash变量一律等于空字符串，所以即使unset命令删除了变量，还是可以读取这个变量，值为空字符串。

所以，删除一个变量，也可以将这个变量设成空字符串。

```
$ foo=''
$ foo=
```

上面两种写法，都是删除变量foo。由于不存在的值默认为空字符串，所以后一种写法可以在等号右边不写任何值。

5.5 输出变量，export 命令

用户创建的变量仅可用于当前Shell，子Shell默认读取不到父Shell定义的变量。为了把变量传递给子Shell，需要使用export命令。这样输出的变量，对于子Shell来说就是环境变量。

export命令用来向子Shell输出变量。

```
$ NAME=foo
$ export NAME
```

上面命令输出了变量NAME。变量的赋值和输出也可以在一个步骤中完成。

```
$ export NAME=value
```

上面命令执行后，当前Shell及随后新建的子Shell，都可以读取变量$NAME。

子Shell如果修改继承的变量，不会影响父Shell。

```
#    $foo
$ export foo=bar

#   Shell
$ bash

#   $foo
$ echo $foo
bar

#
$ foo=baz

#   Shell
$ exit

#   $foo
```

```
$ echo $foo
bar
```

上面例子中，子Shell修改了继承的变量$foo，对父Shell没有影响。

5.6 特殊变量

Bash提供一些特殊变量。这些变量的值由Shell提供，用户不能进行赋值。

（1）$?

$?为上一个命令的退出码，用来判断上一个命令是否执行成功。返回值是0，表示上一个命令执行成功；如果是非零，上一个命令执行失败。

```
$ ls doesnotexist
ls: doesnotexist: No such file or directory

$ echo $?
1
```

上面例子中，ls命令查看一个不存在的文件，导致报错。$1为1，表示上一个命令执行失败。

（2）$$

$$为当前Shell的进程 ID。

```
$ echo $$
10662
```

这个特殊变量可以用来命名临时文件。

```
LOGFILE=/tmp/output_log.$$
```

（3）$_

$_为上一个命令的最后一个参数。

```
$ grep dictionary /usr/share/dict/words
dictionary

$ echo $_
/usr/share/dict/words
```

（4）$!

$!为最近一个后台执行的异步命令的进程 ID。

```
$ firefox &
[1] 11064

$ echo $!
11064
```

上面例子中，firefox是后台运行的命令，$!返回该命令的进程 ID。

（5）$0

$0为当前Shell的名称（在命令行直接执行是）或者脚本名（在脚本中执行时）。

```
$ echo $0
bash
```

上面例子中，$0返回当前运行的是Bash。

（6）$-

$-为当前Shell的启动参数。

```
$ echo $-
himBHs
```

（7）$@和$#

$@和$#表示脚本的参数数量，参见脚本一章。

5.7 变量的默认值

Bash提供四个特殊语法，跟变量的默认值有关，目的是保证变量不为空。

```
${varname:-word}
```

上面语法的含义是，如果变量varname存在且不为空，则返回其值，否则返回word。它的目的是返回一个默认值，比如${count:-0}表示变量count不存在时返回0。

```
${varname:=word}
```

上面语法的含义是，如果变量varname存在且不为空，则返回它的值，否则将它设为word，并且返回word。它的目的是设置变量的默认值，比如${count:=0}表示变量count不存在时返回0，且将count设为0。

```
${varname:+word}
```

上面语法的含义是，如果变量名存在且不为空，则返回word，否则返回空值。它的目的是测试变量是否存在，比如${count:+1}表示变量count存在时返回1（表示true），否则返回空值。

```
${varname:?message}
```

上面语法的含义是，如果变量varname存在且不为空，则返回它的值，否则打印出varname: message，并中断脚本的执行。如果省略了message，则输出默认的信息“parameter null or not set.”。它的目的是防止变量未定义，比如${count:?"undefined!"}表示变量count未定义时就中断执行，抛出错误，返回给定的报错信息undefined!。

上面四种语法如果用在脚本中，变量名的部分可以用到数字1到9，表示脚本的参数。

```
filename=${1:?"filename missing."}
```

上面代码出现在脚本中，1表示脚本的第一个参数。如果该参数不存在，就退出脚本并报错。

5.8 declare 命令

declare命令可以声明一些特殊类型的变量，为变量设置一些限制，比如声明只读类型的变量和整数类型的变量。

它的语法形式如下。

```
declare OPTION VARIABLE=value
```

declare命令的主要参数（OPTION）如下。

- -a：声明数组变量。
- -f：输出所有函数定义。
- -F：输出所有函数名。
- -i：声明整数变量。
- -l：声明变量为小写字母。
- -p：查看变量信息。
- -r：声明只读变量。
- -u：声明变量为大写字母。
- -x：该变量输出为环境变量。

declare命令如果用在函数中，声明的变量只在函数内部有效，等同于local命令。

不带任何参数时，declare命令输出当前环境的所有变量，包括函数在内，等同于不带有任何参数的set命令。

```
$ declare
```

（1）-i参数

-i参数声明整数变量以后，可以直接进行数学运算。

```
$ declare -i val1=12 val2=5
$ declare -i result
$ result=val1*val2
$ echo $result
60
```

上面例子中，如果变量result不声明为整数，val1*val2会被当作字面量，不会进行整数运算。
另外，val1和val2其实不需要声明为整数，因为只要result声明为整数，它的赋值就会自动解释为整数运算。

注意，一个变量声明为整数以后，依然可以被改写为字符串。

```
$ declare -i val=12
$ var=foo
$ echo $var
foo
```

上面例子中，变量foo声明为整数，但是覆盖以后，就变成了字符串，Bash不会报错。

（2）-x参数

-x参数等同于export命令，可以输出一个变量为子Shell的环境变量。

```
$ declare -x foo
#
$ export foo
```

（3）-r参数

-r参数可以声明只读变量，无法改变变量值，也不能unset变量。

```
$ declare -r bar=1

$ bar=2
bash: bar
$ echo $?
1

$ unset bar
bash: bar
$ echo $?
1
```

上面例子中，后两个赋值语句都会报错，命令执行失败。

（4）-u参数

-u参数声明变量为大写字母，可以自动把变量值转成大写字母。

```
$ declare -u foo
$ foo=upper
$ echo $foo
UPPER
```

（5）-l参数

-l参数声明变量为小写字母，可以自动把变量值转成小写字母。

```
$ declare -l bar
$ bar=LOWER
$ echo $bar
lower
```

（6）-p参数

-p参数输出变量信息。

```
$ foo=hello
$ declare -p foo
declare -- foo="hello"
$ declare -p bar
bar
```

上面例子中，declare -p可以输出已定义变量的值，对于未定义的变量，会提示找不到。

如果不提供变量名，declear -p输出所有变量的信息。

```
$ declare -p
```

（7）-f参数

-f参数输出当前环境的所有函数，包括它的定义。

```
$ declare -f
```

（8）-F参数

-F参数输出当前环境的所有函数名，不包含函数定义。

```
$ declare -F
```

5.9 readonly 命令

readonly命令等同于declare -r，用来声明只读变量，不能改变变量值，也不能unset变量。

```
$ readonly foo=1
$ foo=2
bash: foo
$ echo $?
1
```

上面例子中，更改只读变量foo会报错，命令执行失败。

readonly命令有三个参数。

- -f：声明的变量为函数名。
- -p：打印出所有的只读变量。
- -a：声明的变量为数组。

5.10 let 命令

let命令声明变量时，可以直接执行算术表达式。

```
$ let foo=1+2
$ echo $foo
3
```

上面例子中，let命令可以直接计算1 + 2。

let命令的参数表达式如果包含空格，就需要使用引号。

```
$ let "foo = 1 + 2"
```

let可以同时对多个变量赋值，赋值表达式之间使用空格分隔。

```
$ let "v1 = 1" "v2 = v1++"
$ echo $v1,$v2
2,1
```

上面例子中，let声明了两个变量v1和v2，其中v2等于v1++，表示先返回v1的值，然后v1自增。

这种语法支持的运算符，参考《Bash的算术运算》一章。

6. 字符串操作

本章介绍Bash字符串操作的语法。

6.1 字符串的长度

获取字符串长度的语法如下。

```
${#varname}
```

下面是一个例子。

```
$ myPath=/home/cam/book/long.file.name
$ echo ${#myPath}
29
```

大括号{}是必需的，否则Bash会将$#理解成脚本的参数个数，将变量名理解成文本。

```
$ echo $#myvar
0myvar
```

上面例子中，Bash将$#和myvar分开解释了。

6.2 子字符串

字符串提取子串的语法如下。

```
${varname:offset:length}
```

上面语法的含义是返回变量$varname的子字符串，从位置offset开始（从0开始计算），长度为length。

```
$ count=frogfootman
$ echo ${count:4:4}
foot
```

上面例子返回字符串frogfootman从4号位置开始的长度为4的子字符串foot。

这种语法不能直接操作字符串，只能通过变量来读取字符串，且不会改变原始字符串。变量前面的美元符号可省略。

```
# arise error
$ echo ${"hello":2:3}
```

上面例子中，"hello"不是变量名，导致Bash报错。

如果省略length，则从位置offset开始，一直返回到字符串的结尾。

```
$ count=frogfootman
$ echo ${count:4}
footman
```

上面例子是返回变量count从4号位置一直到结尾的子字符串。

如果offset为负值，表示从字符串的末尾开始算起。注意，负数前面必须有一个空格，
以防止与${variable:-word}的变量的设置默认值语法混淆。这时，如果还指定length，则length不能小于零。

```
$ foo="This string is long."
$ echo ${foo: -5}
long.
$ echo ${foo: -5:2}
lo
```

上面例子中，offset为-5，表示从倒数第5个字符开始截取，所以返回long.。如果指定长度为2，则返回lo。

6.3 搜索和替换

Bash提供字符串搜索和替换的多种方法。

（1）字符串头部的模式匹配。

以下两种语法可以检查字符串开头，是否匹配给定的模式。如果匹配成功，就删除匹配的部分，返回剩下的部分。原始变量不会发生变化。

```
#   pattern    variable
#
${variable#pattern}

#   pattern    variable
#
${variable##pattern}
```

上面两种语法会删除变量字符串开头的匹配部分（将其替换为空），返回剩下的部分。区别是一个是最短匹配（又称非贪婪匹配），另一个是最长匹配（又称贪婪匹配）。

匹配模式pattern可以使用*、?、[]等通配符。

```
$ myPath=/home/cam/book/long.file.name

$ echo ${myPath#/*/}
cam/book/long.file.name

$ echo ${myPath##/*/}
long.file.name
```

上面例子中，匹配的模式是/*/，其中*可以匹配任意数量的字符，所以最短匹配是/home/，最长匹配是/home/cam/book/。

下面写法可以删除文件路径的目录部分，只留下文件名。

```
$ path=/home/cam/book/long.file.name

$ echo ${path##*/}
long.file.name
```

上面例子中，模式*/匹配目录部分，所以只返回文件名。

下面再看一个例子。

```
$ phone="555-456-1414"
$ echo ${phone#*-}
456-1414
$ echo ${phone##*-}
1414
```

如果匹配不成功，则返回原始字符串。

```
$ phone="555-456-1414"
$ echo ${phone#444}
555-456-1414
```

上面例子中，原始字符串里面无法匹配模式444，所以原样返回。

如果要将头部匹配的部分，替换成其他内容，采用下面的写法。

```
#
${variable/#pattern/string}

#
$ foo=JPG.JPG
$ echo ${foo/#JPG/jpg}
jpg.JPG
```

上面例子中，被替换的JPG必须出现在字符串头部，所以返回jpg.JPG。

（2）字符串尾部的模式匹配。

以下两种语法可以检查字符串结尾，是否匹配给定的模式。如果匹配成功，就删除匹配的部分，返回剩下的部分。原始变量不会发生变化。

```
#   pattern    variable
#
${variable%pattern}

#   pattern    variable
#
${variable%%pattern}
```

上面两种语法会删除变量字符串结尾的匹配部分（将其替换为空），返回剩下的部分。区别是一个是最短匹配（又称非贪婪匹配），另一个是最长匹配（又称贪婪匹配）。

```
$ path=/home/cam/book/long.file.name

$ echo ${path%.*}
/home/cam/book/long.file

$ echo ${path%%.*}
/home/cam/book/long
```

上面例子中，匹配模式是.*，其中*可以匹配任意数量的字符，所以最短匹配是.name，最长匹配是.file.name。

下面写法可以删除路径的文件名部分，只留下目录部分。

```
$ path=/home/cam/book/long.file.name

$ echo ${path%/*}
/home/cam/book
```

上面例子中，模式/*匹配文件名部分，所以只返回目录部分。

下面的写法可以替换文件的后缀名。

```
$ file=foo.png
$ echo ${file%.png}.jpg
foo.jpg
```

上面的例子将文件的后缀名，从.png改成了.jpg。

下面再看一个例子。

```
$ phone="555-456-1414"
$ echo ${phone%-*}
555-456
$ echo ${phone%%-*}
555
```

如果匹配不成功，则返回原始字符串。

如果要将尾部匹配的部分，替换成其他内容，采用下面的写法。

```
#
${variable/%pattern/string}

#
$ foo=JPG.JPG
$ echo ${foo/%JPG/jpg}
JPG.jpg
```

上面例子中，被替换的JPG必须出现在字符串尾部，所以返回JPG.jpg。

（3）任意位置的模式匹配。

以下两种语法可以检查字符串内部，是否匹配给定的模式。如果匹配成功，就删除匹配的部分，换成其他的字符串返回。原始变量不会发生变化。

```
#   pattern    variable
#          string
${variable/pattern/string}

#   pattern    variable
#          string
${variable//pattern/string}
```

上面两种语法都是最长匹配（贪婪匹配）下的替换，前一个语法仅替换第一个匹配，后一个语法替换所有匹配。

```
$ path=/home/cam/foo/foo.name

$ echo ${path/foo/bar}
/home/cam/bar/foo.name

$ echo ${path//foo/bar}
/home/cam/bar/bar.name
```

上面例子中，前一个命令只替换了第一个foo，后一个命令将两个foo都替换了。

下面的例子将分隔符从:换成换行符。

```
$ echo -e ${PATH//:/'\n'}
/usr/local/bin
/usr/bin
/bin
...
```

上面例子中，echo命令的-e参数，表示将替换后的字符串的\n字符，解释为换行符。

模式部分可以使用通配符。

```
$ phone="555-456-1414"
$ echo ${phone/5?4/-}
55-56-1414
```

上面的例子将5-4替换成-。

如果省略了string部分，那么就相当于匹配的部分替换成空字符串，即删除匹配的部分。

```
$ path=/home/cam/foo/foo.name

$ echo ${path/.*/}
/home/cam/foo/foo
```

上面例子中，第二个斜杠后面的string部分省略了，所以模式.*匹配的部分.name被删除后返回。

前面提到过，这个语法还有两种扩展形式。

```
#
${variable/#pattern/string}

#
${variable/%pattern/string}
```

6.4改变大小写

下面的语法可以改变变量的大小写。

```
#
${varname^^}
```

```
#
${varname,,}
```

下面是一个例子。

```
$ foo=heLLo
$ echo ${foo^^}
HELLO
$ echo ${foo,,}
hello
```

7. Bash的算术运算

7.1 算术表达式

((...))语法可以进行整数的算术运算。

```
$ ((foo = 5 + 5))
$ echo $foo
10
```

((...))会自动忽略内部的空格，所以下面的写法都正确，得到同样的结果。

```
$ ((2+2))
$ (( 2+2 ))
$ (( 2 + 2 ))
```

这个语法不返回值，命令执行的结果根据算术运算的结果而定。只要算术结果不是0，命令就算执行成功。

```
$ (( 3 + 2 ))
$ echo $?
0
```

上面例子中，3 + 2的结果是5，命令就算执行成功，环境变量$?为0。

如果算术结果为0，命令就算执行失败。

```
$ (( 3 - 3 ))
$ echo $?
1
```

上面例子中，3 - 3的结果是0，环境变量$?为1，表示命令执行失败。

如果要读取算术运算结果，需要在((...))前加上美元符号$((...))，使其变成算术表达式，返回算术运算值。

```
$ echo $((2 + 2))
4
```

((...))语法支持的算术运算符如下。

- +：加法
- -：减法
- *：乘法

- /：除法（整除）
- %：余数
- **：指数
- ++：自增运算（前缀或后缀）
- --：自减运算（前缀或后缀）

注意，除法运算符的返回结果总是整数，比如5除以2，得到的结果是2，而不是2.5。

```
$ echo $((5 / 2))
2
```

++和--这两个运算符有前缀和后缀的区别。作为前缀是先运算后返回值，作为后缀是先返回值后运算。

```
$ i=0
$ echo $i
0
$ echo $((i++))
0
$ echo $i
1
$ echo $((++i))
2
$ echo $i
2
```

上面例子中，++作为后缀是先返回值，执行echo命令，再进行自增运算；作为前缀则是先进行自增运算，再返回值执行echo命令。

$((...))内部可以用圆括号改变运算顺序。

```
$ echo $(( (2 + 3) * 4 ))
20
```

上面例子中，内部的圆括号让加法先于乘法执行。

$((...))结构可以嵌套。

```
$ echo $(((5**2) * 3))
75
#
$ echo $(($((5**2)) * 3))
75
```

这个语法只能计算整数，否则会报错。

```
#
$ echo $((1.5 + 1))
bash:
```

$((...))的圆括号之中，不需要在变量名之前加上$，不过加上也不报错。

```
$ number=2
$ echo $(($number + 1))
3
```

上面例子中，变量number前面有没有美元符号，结果都是一样的。

如果在$((...))里面使用字符串，Bash会认为那是一个变量名。如果不存在同名变量，Bash就会将其作为空值，因此不会报错。

```
$ echo $(( "hello" + 2))
2
$ echo $(( "hello" * 2))
0
```

上面例子中，"hello"会被当作变量名，返回空值，而$((...))会将空值当作0，所以乘法的运算结果就是0。同理，如果$((...))里面使用不存在的变量，也会当作0处理。

如果一个变量的值为字符串，跟上面的处理逻辑是一样的。即该字符串如果不对应已存在的变量，在$((...))里面会被当作空值。

```
$ foo=hello
$ echo $(( foo + 2))
2
```

上面例子中，变量foo的值是hello，而hello也会被看作变量名。这使得有可能写出动态替换的代码。

```
$ foo=hello
$ hello=3
$ echo $(( foo + 2 ))
5
```

上面代码中，foo + 2取决于变量hello的值。

最后，$[...]是以前的语法，也可以做整数运算，不建议使用。

```
$ echo $[2+2]
4
```

7.2 数值的进制

Bash的数值默认都是十进制，但是在算术表达式中，也可以使用其他进制。

- number：没有任何特殊表示法的数字是十进制数（以10为底）。
- 0number：八进制数。
- 0xnumber：十六进制数。
- base#number：base进制的数。

下面是一些例子。

```
$ echo $((0xff))
255
$ echo $((2#11111111))
255
```

上面例子中，0xff是十六进制数，2#11111111是二进制数。

7.3 位运算

$((...))支持以下的二进制位运算符。

- <<：位左移运算，把一个数字的所有位向左移动指定的位。
- >>：位右移运算，把一个数字的所有位向右移动指定的位。
- & ：位的“与”运算，对两个数字的所有位执行一个AND操作。
- |：位的“或”运算，对两个数字的所有位执行一个OR操作。
- ~：位的“否”运算，对一个数字的所有位取反。
- !：逻辑“否”运算
- ^：位的异或运算（exclusive or），对两个数字的所有位执行一个异或操作。

下面是右移运算符>>的例子。

```
$ echo $((16>>2))
4
```

下面是左移运算符<<的例子。

```
$ echo $((16<<2))
64
```

下面是17（二进制1001）和3（二进制11）的各种二进制运算的结果。

```
$ echo $((17&3))
1
$ echo $((17|3))
19
$ echo $((17^3))
18
```

7.4 逻辑运算

$((...))支持以下的逻辑运算符。

- <：小于
- >：大于
- <=：小于或相等
- >=：大于或相等
- ==：相等
- !=：不相等
- &&：逻辑与
- ||：逻辑或
- expr1?expr2:expr3：三元条件运算符。若表达式expr1的计算结果为非零值（算术真），则执行表达式expr2，否则执行表达式expr3。

如果逻辑表达式为真，返回1，否则返回0。

```
$ echo $((3 > 2))
1
$ echo $(( (3 > 2) || (4 <= 1) ))
1
```

三元运算符执行一个单独的逻辑测试。它用起来类似于`if/then/else`语句。

```
$ a=0
$ echo $((a<1 ? 1 : 0))
1
$ echo $((a>1 ? 1 : 0))
0
```

上面例子中，第一个表达式为真时，返回第二个表达式的值，否则返回第三个表达式的值。

7.5 赋值运算

算术表达式`$((...))`可以执行赋值运算。

```
$ echo $((a=1))
1
$ echo $a
1
```

上面例子中，`a=1`对变量`a`进行赋值。这个式子本身也是一个表达式，返回值就是等号右边的值。

`$((...))`支持的赋值运算符，有以下这些。

- `parameter = value`：简单赋值。
- `parameter += value`：等价于`parameter = parameter + value`。
- `parameter -= value`：等价于`parameter = parameter - value`。
- `parameter *= value`：等价于`parameter = parameter * value`。
- `parameter /= value`：等价于`parameter = parameter / value`。
- `parameter %= value`：等价于`parameter = parameter % value`。
- `parameter <<= value`：等价于`parameter = parameter << value`。
- `parameter >>= value`：等价于`parameter = parameter >> value`。
- `parameter &= value`：等价于`parameter = parameter & value`。
- `parameter |= value`：等价于`parameter = parameter | value`。
- `parameter ^= value`：等价于`parameter = parameter ^ value`。

下面是一个例子。

```
$ foo=5
$ echo $((foo*=2))
10
```

如果在表达式内部赋值，可以放在圆括号中，否则会报错。

```
$ echo $(( a<1 ? (a+=1) : (a-=1) ))
```

7.6 求值运算

逗号,在$((...))内部是求值运算符，执行前后两个表达式，并返回后一个表达式的值。

```
$ echo $((foo = 1 + 2, 3 * 4))
12
$ echo $foo
3
```

上面例子中，逗号前后两个表达式都会执行，然后返回后一个表达式的值12。

7.7 expr 命令

expr命令支持算术运算，可以不使用((...))语法。

```
$ expr 3 + 2
5
```

expr命令支持变量替换。

```
$ foo=3
$ expr $foo + 2
5
```

expr命令也不支持非整数参数。

```
$ expr 3.5 + 2
expr:
```

上面例子中，如果有非整数的运算，expr命令就报错了。

8. Bash行操作

8.1 简介

Bash内置了 Readline 库，具有这个库提供的很多“行操作”功能，比如命令的自动补全，可以大大加快操作速度。

这个库默认采用 Emacs 快捷键，也可以改成 Vi 快捷键。

```
$ set -o vi
```

下面的命令可以改回 Emacs 快捷键。

```
$ set -o emacs
```

如果想永久性更改编辑模式（Emacs / Vi），可以将命令写在~/.inputrc文件，这个文件是 Readline 的配置文件。

```
set editing-mode vi
```

本章介绍的快捷键都属于 Emacs 模式。Vi 模式的快捷键，读者可以参考 Vi 编辑器的教程。

Bash默认开启这个库，但是允许关闭。

```
$ bash --noediting
```

上面命令中，`--noediting`参数关闭了 Readline 库，启动的Bash就不带有行操作功能。

8.2 光标移动

Readline 提供快速移动光标的快捷键。

- `Ctrl + a`：移到行首。
- `Ctrl + b`：向行首移动一个字符，与左箭头作用相同。
- `Ctrl + e`：移到行尾。
- `Ctrl + f`：向行尾移动一个字符，与右箭头作用相同。
- `Alt + f`：移动到当前单词的词尾。
- `Alt + b`：移动到当前单词的词首。

上面快捷键的 Alt 键，也可以用 ESC 键代替。

8.3 清除屏幕

`Ctrl + l`快捷键可以清除屏幕，即将当前行移到屏幕的第一行，与`clear`命令作用相同。

8.4 编辑操作

下面的快捷键可以编辑命令行内容。

- `Ctrl + d`：删除光标位置的字符（delete）。
- `Ctrl + w`：删除光标前面的单词。
- `Ctrl + t`：光标位置的字符与它前面一位的字符交换位置（transpose）。
- `Alt + t`：光标位置的词与它前面一位的词交换位置（transpose）。
- `Alt + l`：将光标位置至词尾转为小写（lowercase）。
- `Alt + u`：将光标位置至词尾转为大写（uppercase）。

使用`Ctrl + d`的时候，如果当前行没有任何字符，会导致退出当前Shell，所以要小心。

剪切和粘贴快捷键如下。

- `Ctrl + k`：剪切光标位置到行尾的文本。
- `Ctrl + u`：剪切光标位置到行首的文本。
- `Alt + d`：剪切光标位置到词尾的文本。
- `Alt + Backspace`：剪切光标位置到词首的文本。
- `Ctrl + y`：在光标位置粘贴文本。

同样地，Alt 键可以用 Esc 键代替。

8.5 自动补全

命令输入到一半的时候，可以按一下 Tab 键，Readline 会自动补全命令或路径。比如，输入`cle`，再按下 Tab 键，Bash会自动将这个命令补全为`clear`。

如果符合条件的命令或路径有多个，就需要连续按两次 Tab 键，Bash会提示所有符合条件的命令或路径。

除了命令或路径，Tab还可以补全其他值。如果一个值以$开头，则按下Tab键会补全变量；如果以~开头，则补全用户名；如果以@开头，则补全主机名（hostname），主机名以列在/etc/hosts文件里面的主机为准。

自动补全相关的快捷键如下。

- Tab：完成自动补全。
- Alt + ?：列出可能的补全，与连按两次 Tab 键作用相同。
- Alt + /：尝试文件路径补全。
- Ctrl + x /：先按Ctrl + x，再按/，等同于Alt + ?，列出可能的文件路径补全。
- Alt + !：命令补全。
- Ctrl + x !：先按Ctrl + x，再按!，等同于Alt + !，命令补全。
- Alt + ~：用户名补全。
- Ctrl + x ~：先按Ctrl + x，再按~，等同于Alt + ~，用户名补全。
- Alt + $：变量名补全。
- Ctrl + x $：先按Ctrl + x，再按$，等同于Alt + $，变量名补全。
- Alt + @：主机名补全。
- Ctrl + x @：先按Ctrl + x，再按@，等同于Alt + @，主机名补全。
- Alt + *：在命令行一次性插入所有可能的补全。
- Alt + Tab：尝试用.bash_history里面以前执行命令，进行补全。

上面的Alt键也可以用 ESC 键代替。

8.6 操作历史

基本用法

Bash会保留用户的操作历史，即用户输入的每一条命令都会记录。退出当前Shell的时候，Bash会将用户在当前Shell的操作历史写入~/.bash_history文件，该文件默认储存500个操作。

环境变量HISTFILE总是指向这个文件。

```
$ echo $HISTFILE
/home/me/.bash_history
```

有了操作历史以后，就可以使用方向键的↑和↓，快速浏览上一条和下一条命令。

下面的方法可以快速执行以前执行过的命令。

```
$ echo Hello World
Hello World

$ echo Goodbye
Goodbye

$ !e
echo Goodbye
Goodbye
```

上面例子中，!e表示找出操作历史之中，最近的那一条以e开头的命令并执行。Bash会先输出那一条命令echo Goodbye，然后直接执行。

同理，!echo也会执行最近一条以echo开头的命令。

```
$ !echo
echo Goodbye
Goodbye

$ !echo H
echo Goodbye H
Goodbye H

$ !echo H G
echo Goodbye H G
Goodbye H G
```

注意，!string语法只会匹配命令，不会匹配参数。所以!echo H不会执行echo Hello World，而是会执行echo Goobye，并把参数H附加在这条命令之后。同理，!echo H G也是等同于echo Goodbye命令之后附件H G。

最后，按下Ctrl + r会显示操作历史，可以用方向键上下移动，选择其中要执行的命令。也可以键入命令的首字母，Shell就会自动在历史文件中，查询并显示匹配的结果。

history 命令

history命令能显示操作历史，即.bash_history文件的内容。

```
$ history
...
498 echo Goodbye
499 ls ~
500 cd
```

使用该命令，而不是直接读取.bash_history文件的好处是，它会在所有的操作前加上行号，最近的操作在最后面，行号最大。

通过定制环境变量HISTTIMEFORMAT，可以显示每个操作的时间。

```
$ export HISTTIMEFORMAT='%F %T  '
$ history
1  2013-06-09 10:40:12   cat /etc/issue
2  2013-06-09 10:40:12   clear
```

上面代码中，%F相当于%Y - %m - %d，%T相当于%H : %M : %S。

只要设置HISTTIMEFORMAT这个环境变量，就会在.bash_history文件保存命令的执行时间戳。如果不设置，就不会保存时间戳。

如果不希望保存本次操作的历史，可以设置环境变量HISTSIZE等于0。

```
export HISTSIZE=0
```

如果HISTSIZE=0写入用户主目录的~/.bashrc文件，那么就不会保留该用户的操作历史。如果写入/etc/profile，整个系统都不会保留操作历史。

如果想搜索某个以前执行的命令，可以配合grep命令搜索操作历史。

```
$ history | grep /usr/bin
```

上面命令返回.bash_history文件里面，那些包含/usr/bin的命令。

操作历史的每一条记录都有编号。知道了命令的编号以后，可以用 ! + 执行该命令。
如果想要执行.bash_history里面的第8条命令，可以像下面这样操作。

```
$ !8
```

history命令的-c参数可以清除操作历史。

```
$ history -c
```

相关快捷键

下面是一些与操作历史相关的快捷键。

- Ctrl + p：显示上一个命令，与向上箭头效果相同（previous）。
- Ctrl + n：显示下一个命令，与向下箭头效果相同（next）。
- Alt + <：显示第一个命令。
- Alt + >：显示最后一个命令，即当前的命令。
- Ctrl + o：执行历史文件里面的当前条目，并自动显示下一条命令。这对重复执行某个序列的命令很有帮助。

感叹号!的快捷键如下。

- !!：执行上一个命令。
- !n：执行历史文件里面行号为n的命令。
- !-n：执行当前命令之前n条的命令。
- !string：执行最近一个以指定字符串string开头的命令。
- !?string：执行最近一条包含字符串string的命令。
- ^string1^string2：执行最近一条包含string1的命令，将其替换成string2。

8.7 其他快捷键

- Ctrl + j：等同于回车键（LINEFEED）。
- Ctrl + m：等同于回车键（CARRIAGE RETURN）。
- Ctrl + o：等同于回车键，并展示操作历史的下一个命令。
- Ctrl + v：将下一个输入的特殊字符变成字面量，比如回车变成^M。
- Ctrl + [：等同于 ESC。
- Alt + .：插入上一个命令的最后一个词。
- Alt + _：等同于Alt + .。

上面的Alt + .快捷键，对于很长的文件路径，有时会非常方便。因为Unix命令的最后一个参数通常是文件路径。

```
$ mkdir foo_bar
$ cd #  Alt + .
```

上面例子中，在cd命令后按下Alt + .，就会自动插入foo_bar。

9. 目录堆栈

为了方便用户在不同目录之间切换，Bash提供了目录堆栈功能。

9.1 cd -

Bash可以记忆用户进入过的目录。默认情况下，只记忆前一次所在的目录，cd -命令可以返回前一次的目录。

```
#     /path/to/foo
$ cd bar

#    /path/to/foo
$ cd -
```

上面例子中，用户原来所在的目录是/path/to/foo，进入子目录bar以后，使用cd -可以回到原来的目录。

9.2 pushd，popd

如果希望记忆多重目录，可以使用pushd命令和popd命令。它们用来操作目录堆栈。

pushd命令的用法类似cd命令，可以进入指定的目录。

```
$ pushd dirname
```

上面命令会进入目录dirname，并将该目录放入堆栈。

第一次使用pushd命令时，会将当前目录先放入堆栈，然后将所要进入的目录也放入堆栈，位于前一个记录上方。以后每次使用pushd命令，都会将所要进入的目录，放在堆栈的顶部。

popd命令不带有参数时，会移除堆栈的顶部记录，并进入新的堆栈顶部目录（即原来的第二条目录）。

下面是一个例子。

```
#
$ pwd
/home/me

#   /home/me/foo
#     /home/me/foo /home/me
$ pushd ~/foo

#   /etc
#     /etc /home/me/foo /home/me
$ pushd /etc

#   /home/me/foo
#     /home/me/foo /home/me
$ popd
```

```
#   /home/me
#    /home/me
$ popd

#
$ popd
```

这两个命令的参数如下。

（1）-n 参数

-n的参数表示仅操作堆栈，不改变目录。

```
$ popd -n
```

上面的命令仅删除堆栈顶部的记录，不改变目录，执行完成后还停留在当前目录。

（2）整数参数

这两个命令还可以接受一个整数作为参数，该整数表示堆栈中指定位置的记录（从0开始），作为操作对象。这时不会切换目录。

```
#    3   0
$ pushd +3

#    3   0
$ pushd -3

#     3   0
$ popd +3

#     3   0
$ popd -3
```

上面例子的整数编号都是从0开始计算，popd +0是删除第一个目录，popd +1是删除第二个，popd -0是删除最后一个目录，，popd -1是删除倒数第二个。

（3）目录参数

pushd可以接受一个目录作为参数，表示将该目录放到堆栈顶部，并进入该目录。

```
$ pushd dir
```

popd没有这个参数。

9.3 dirs 命令

dirs命令可以显示目录堆栈的内容，一般用来查看pushd和popd操作后的结果。

```
$ dirs
```

它有以下参数。

- -c：清空目录栈。
- -l：用户主目录不显示波浪号前缀，而打印完整的目录。
- -p：每行一个条目打印目录栈，默认是打印在一行。
- -v：每行一个条目，每个条目之前显示位置编号（从0开始）。
- +N：N为整数，表示显示堆顶算起的第 N 个目录，从零开始。
- -N：N为整数，表示显示堆底算起的第 N 个目录，从零开始。

10. Bash脚本入门

脚本（script）就是包含一系列命令的一个文本文件。Shell读取这个文件，依次执行里面的所有命令，就好像这些命令直接输入到命令行一样。所有能够在命令行完成的任务，都能够用脚本完成。

脚本的好处是可以重复使用，也可以指定在特定场合自动调用，比如系统启动或关闭时自动执行脚本。

10.1 Shebang 行

脚本的第一行通常是指定解释器，即这个脚本必须通过什么解释器执行。这一行以**#!**字符开头，这个字符称为Shebang，所以这一行就叫做 Shebang 行。

#!后面就是脚本解释器的位置，Bash脚本的解释器一般是**/bin/sh**或**/bin/bash**。

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

#!与脚本解释器之间有没有空格，都是可以的。

如果Bash解释器不放在目录**/bin**，脚本就无法执行了。为了保险，可以写成下面这样。

```
#!/usr/bin/env bash
```

上面命令使用**env**命令（这个命令总是在**/usr/bin**目录），返回Bash可执行文件的位置。
env命令的详细介绍，请看后文。

Shebang 行不是必需的，但是建议加上这行。如果缺少该行，就需要手动将脚本传给解释器。
举例来说，脚本是**script.sh**，有 Shebang 行的时候，可以直接调用执行。

```
$ ./script.sh
```

上面例子中，**script.sh**是脚本文件名。脚本通常使用**.sh**后缀名，不过这不是必需的。

如果没有 Shebang 行，就只能手动将脚本传给解释器来执行。

```
$ /bin/sh ./script.sh
#
$ bash ./script.sh
```

10.2 执行权限和路径

前面说过，只要指定了 Shebang 行的脚本，可以直接执行。这有一个前提条件，就是脚本需要有执行权限。
可以使用下面的命令，赋予脚本执行权限。

```
#
$ chmod +x script.sh

#
$ chmod +rx script.sh
#
$ chmod 755 script.sh

#
$ chmod u+rx script.sh
```

脚本的权限通常设为755（拥有者有所有权限，其他人有读和执行权限）或者700（只有拥有者可以执行）。

除了执行权限，脚本调用时，一般需要指定脚本的路径（比如path/script.sh）。如果将脚本放在环境变量$PATH指定的目录中，就不需要指定路径了。因为Bash会自动到这些目录中，寻找是否存在同名的可执行文件。

建议在主目录新建一个~/bin子目录，专门存放可执行脚本，然后把~/bin加入$PATH。

```
export PATH=$PATH:~/bin
```

上面命令改变环境变量$PATH，将~/bin添加到$PATH的末尾。可以将这一行加到~/.bashrc文件里面，然后重新加载一次.bashrc，这个配置就可以生效了。

```
$ source ~/.bashrc
```

以后不管在什么目录，直接输入脚本文件名，脚本就会执行。

```
$ script.sh
```

上面命令没有指定脚本路径，因为script.sh在$PATH指定的目录中。

10.3 env 命令

env命令总是指向/usr/bin/env文件，或者说，这个二进制文件总是在目录/usr/bin。

#!/usr/bin/env NAME这个语法的意思是，让Shell查找$PATH环境变量里面第一个匹配的NAME。如果你不知道某个命令的具体路径，或者希望兼容其他用户的机器，这样的写法就很有用。

/usr/bin/env bash的意思就是，返回bash可执行文件的位置，前提是bash的路径是在$PATH里面。其他脚本文件也可以使用这个命令。比如 Node.js 脚本的 Shebang 行，可以写成下面这样。

```
#!/usr/bin/env node
```

env命令的参数如下。

- -i，--ignore-environment：不带环境变量启动。
- -u，--unset=NAME：从环境变量中删除一个变量。
- --help：显示帮助。
- --version：输出版本信息。

下面是一个例子，新建一个不带任何环境变量的Shell。

```
$ env -i /bin/sh
```

10.4 注释

Bash脚本中，#表示注释，可以放在行首，也可以放在行尾。

```
#
echo 'Hello World!'

echo 'Hello World!' #
```

建议在脚本开头，使用注释说明当前脚本的作用，这样有利于日后的维护。

10.5 脚本参数

调用脚本的时候，脚本文件名后面可以带有参数。

```
$ script.sh word1 word2 word3
```

上面例子中，script.sh是一个脚本文件，word1、word2和word3是三个参数。

脚本文件内部，可以使用特殊变量，引用这些参数。

- $0：脚本文件名，即script.sh。
- $1~$9：对应脚本的第一个参数到第九个参数。
- $#：参数的总数。
- $@：全部的参数，参数之间使用空格分隔。
- $*：全部的参数，参数之间使用变量$IFS值的第一个字符分隔，默认为空格，但是可以自定义。

如果脚本的参数多于9个，那么第10个参数可以用${10}的形式引用，以此类推。

注意，如果命令是command -o foo bar，那么-o是$1，foo是$2，bar是$3。

下面是一个脚本内部读取命令行参数的例子。

```
#!/bin/bash
# script.sh

echo "   " $@
echo "     " $#
echo '$0 = ' $0
echo '$1 = ' $1
echo '$2 = ' $2
echo '$3 = ' $3
```

执行结果如下。

```
$ ./script.sh a b c
   a b c
     3
$0 =  script.sh
$1 =  a
$2 =  b
$3 =  c
```

用户可以输入任意数量的参数，利用for循环，可以读取每一个参数。

```
#!/bin/bash

for i in "$@"; do
  echo $i
done
```

上面例子中，$@返回一个全部参数的列表，然后使用for循环遍历。

如果多个参数放在双引号里面，视为一个参数。

```
$ ./script.sh "a b"
```

上面例子中，Bash会认为"a b"是一个参数，$1会返回a b。注意，返回时不包括双引号。

10.6 shift 命令

shift命令可以改变脚本参数，每次执行都会移除脚本当前的第一个参数（$1），使得后面的参数向前一位，即$2变成$1、$3变成$2、$4变成$3，以此类推。

while循环结合shift命令，也可以读取每一个参数。

```
#!/bin/bash

echo "    $#   "

while [ "$1" != "" ]; do
  echo "  $#   "
  echo "  $1"
  shift
done
```

上面例子中，shift命令每次移除当前第一个参数，从而通过while循环遍历所有参数。

shift命令可以接受一个整数作为参数，指定所要移除的参数个数，默认为1。

```
shift 3
```

上面的命令移除前三个参数，原来的$4变成$1。

10.7 getopts 命令

getopts命令用在脚本内部，可以解析复杂的脚本命令行参数，通常与while循环一起使用，取出脚本所有的带有前置连词线（-）的参数。

```
getopts optstring name
```

它带有两个参数。第一个参数optstring是字符串，给出脚本所有的连词线参数。比如，某个脚本可以有三个配置项参数-l、-h、-a，其中只有-a可以带有参数值，而-l和-h是开关参数，那么getopts的第一个参数写成lha:，顺序不重要。注意，a后面有一个冒号，表示该参数带有参数值，getopts规定带有参数值的配置项

参数，后面必须带有一个冒号（:）。getopts的第二个参数name是一个变量名，用来保存当前取到的配置项参数，即l、h或a。

下面是一个例子。

```
while getopts 'lha:' OPTION; do
  case "$OPTION" in
    l)
      echo "linuxconfig"
      ;;

    h)
      echo "h stands for h"
      ;;

    a)
      avalue="$OPTARG"
      echo "The value provided is $OPTARG"
      ;;
    ?)
      echo "script usage: $(basename $0) [-l] [-h] [-a somevalue]" >&2
      exit 1
      ;;
  esac
done
shift "$(($OPTIND - 1))"
```

上面例子中，while循环不断执行getopts 'lha:' OPTION命令，每次执行就会读取一个连词线参数（以及对应的参数值），然后进入循环体。变量OPTION保存的是，当前处理的那一个连词线参数（即l、h或a）。如果用户输入了未指定参数（比如-x），那么OPTION等于?。循环体内使用case判断，处理这四种不同的情况。

如果某个连词线参数带有参数值，比如-a foo，那么处理a参数的时候，环境变量$OPTARG保存的就是参数值。

注意，只要遇到不带连词线的参数，getopts就会执行失败，从而退出while循环。比如，getopts可以解析command -l foo，但不可以解析command foo -l。另外，多个连词线参数写在一起的形式，比如command -lh， getopts也可以正确处理。

变量$OPTIND在getopts开始执行前是1，然后每次执行就会加1。等到退出while循环，就意味着连词线参数全部处理完毕。这时，$OPTIND - 1就是已经处理的连词线参数个数，使用shift命令将这些参数移除，保证后面的代码可以用$1、$2等处理命令的主参数。

10.8 配置项参数终止符 --

变量当作命令的参数时，有时希望指定变量只能作为实体参数，不能当作配置项参数，这时可以使用配置项参数终止符--。

```
$ myPath="~/docs"
$ ls -- $myPath
```

上面例子中，--强制变量$myPath只能当作实体参数（即路径名）解释。

如果变量不是路径名，就会报错。

```
$ myPath="-l"
$ ls -- $myPath
ls:   '-l':
```

上面例子中，变量myPath的值为-l，不是路径。但是，--强制$myPath只能作为路径解释，导致报错“不存在该路径”。

10.9 别名，alias 命令

alias命令用来为一个命令指定别名，这样更便于记忆。下面是alias的格式。

```
alias NAME=DEFINITION
```

上面命令中，NAME是别名的名称，DEFINITION是别名对应的原始命令。注意，等号两侧不能有空格，否则会报错。

一个常见的例子是为grep命令起一个search的别名。

```
alias search=grep
```

alias也可以用来为长命令指定一个更短的别名。下面是通过别名定义一个today的命令。

```
$ alias today='date +"%A, %B %-d, %Y"'
$ today
  ,   6, 2020
```

有时为了防止误删除文件，可以指定rm命令的别名。

```
$ alias rm='rm -i'
```

上面命令指定rm命令是rm -i，每次删除文件之前，都会让用户确认。

alias定义的别名也可以接受参数，参数会直接传入原始命令。

```
$ alias echo='echo It says: '
$ echo hello world
It says: hello world
```

上面例子中，别名定义了echo命令的前两个参数，等同于修改了echo命令的默认行为。

指定别名以后，就可以像使用其他命令一样使用别名。一般来说，都会把常用的别名写在~/.bashrc的末尾。另外，只能为命令定义别名，为其他部分（比如很长的路径）定义别名是无效的。

直接调用alias命令，可以显示所有别名。

```
$ alias
```

unalias命令可以解除别名。

```
$ unalias lt
```

10.10 exit 命令

exit命令用于终止当前脚本的执行，并向Shell返回一个退出值。

```
$ exit
```

上面命令中止当前脚本，将最后一条命令的退出状态，作为整个脚本的退出状态。

exit命令后面可以跟参数，该参数就是退出状态。

```
#   0
$ exit 0

#   1
$ exit 1
```

退出时，脚本会返回一个退出值。脚本的退出值，0表示正常，1表示发生错误，2表示用法不对，126表示不是可执行脚本，127表示命令没有发现。如果脚本被信号N终止，则退出值为128 + N。 简单来说，只要退出值非0，就认为执行出错。

下面是一个例子。

```
if [ $(id -u) != "0" ]; then
  echo "      "
  exit 1
fi
```

上面的例子中，id -u命令返回用户的 ID，一旦用户的 ID 不等于0（根用户的ID），脚本就会退出，并且退出码为1，表示运行失败。

exit与return命令的差别是，return命令是函数的退出，并返回一个值给调用者，脚本依然执行。exit是整个脚本的退出，如果在函数之中调用exit，则退出函数，并终止脚本执行。

10.11 命令执行结果

命令执行结束后，会有一个返回值。0表示执行成功，非0（通常是1）表示执行失败。环境变量$?可以读取前一个命令的返回值。

利用这一点，可以在脚本中对命令执行结果进行判断。

```
cd $some_directory
if [ "$?" = "0" ]; then
  rm *
else
  echo "    " 1>&2
  exit 1
fi
```

上面例子中，cd $some_directory这个命令如果执行成功（返回值等于0），就删除该目录里面的文件，否则退出脚本，整个脚本的返回值变为1，表示执行失败。

由于if可以直接判断命令的执行结果，执行相应的操作，上面的脚本可以改写成下面的样子。

```
if cd $some_directory; then
  rm *
else
  echo "Could not change directory! Aborting." 1>&2
  exit 1
fi
```

更简洁的写法是利用两个逻辑运算符&&（且）和||（或）。

```
#
cd $some_directory && rm *

#
cd $some_directory || exit 1
```

10.12 参考链接

- How to use getopts to parse a script options, Egidio Docile

11. read命令

11.1 用法

有时，脚本需要在执行过程中，由用户提供一部分数据，这时可以使用read命令。它将用户的输入存入一个变量，方便后面的代码使用。用户按下回车键，就表示输入结束。

read命令的格式如下。

```
read [-options] [variable...]
```

上面语法中，options是参数选项，variable是用来保存输入数值的一个或多个变量名。如果没有提供变量名，环境变量REPLY会包含用户输入的一整行数据。

下面是一个例子demo.sh。

```
#!/bin/bash

echo -n "    > "
read text
echo "   $text"
```

上面例子中，先显示一行提示文本，然后会等待用户输入文本。用户输入的文本，存入变量text，并在下一行显示。

```
$ bash demo.sh
    >
```

read可以接受用户输入的多个值。

```
#!/bin/bash
echo Please, enter your firstname and lastname
```

```
read FN LN
echo "Hi! $LN, $FN !"
```

上面例子中，read根据用户的输入，同时为两个变量赋值。

如果用户的输入项少于read命令给出的变量数目，那么额外的变量值为空。如果用户的输入项多于定义的变量，那么多余的输入项会包含到最后一个变量中。

如果read命令之后没有定义变量名，那么环境变量REPLY会包含所有的输入。

```
#!/bin/bash
# read-single: read multiple values into default variable
echo -n "Enter one or more values > "
read
echo "REPLY = '$REPLY'"
```

上面脚本的运行结果如下。

```
$ read-single
Enter one or more values > a b c d
REPLY = 'a b c d'
```

read命令除了读取键盘输入，可以用来读取文件。

```
while read myline
do
  echo "$myline"
done < $filename
```

上面的例子通过read命令，读取一个文件的内容。done命令后面的定向符<，将文件导向read命令，每次读取一行，存入变量myline，直到文件读取完毕。

11.2 参数

read命令的参数如下。

（1）-t 参数

read命令的-t参数，设置了超时的秒数。如果超过了指定时间，用户仍然没有输入，脚本将放弃等待，继续向下执行。

```
#!/bin/bash

echo -n "    > "
if read -t 3 response; then
  echo "    "
else
  echo "    "
fi
```

上面例子中，输入命令会等待3秒，如果用户超过这个时间没有输入，这个命令就会执行失败。if根据命令的返回值，转入else代码块，继续往下执行。

环境变量TMOUT也可以起到同样作用，指定read命令等待用户输入的时间（单位为秒）。

```
$ TMOUT=3
$ read response
```

上面例子也是等待3秒，如果用户还没有输入，就会超时。

（2）-p 参数

-p参数指定用户输入的提示信息。

```
read -p "Enter one or more values > "
echo "REPLY = '$REPLY'"
```

上面例子中，先显示Enter one or more values >，再接受用户的输入。

（3）-a 参数

-a参数把用户的输入赋值给一个数组，从零号位置开始。

```
$ read -a people
alice duchess dodo
$ echo ${people[2]}
dodo
```

上面例子中，用户输入被赋值给一个数组people，这个数组的2号成员就是dodo。

（4）-n 参数

-n参数指定只读取若干个字符作为变量值，而不是整行读取。

```
$ read -n 3 letter
abcdefghij
$ echo $letter
abc
```

上面例子中，变量letter只包含3个字母。

（5）其他参数

- -d delimiter：定义字符串delimiter的第一个字符作为用户输入的结束，而不是一个换行符。
- -r：raw 模式，表示不把用户输入的反斜杠字符解释为转义字符。
- -s：使得用户的输入不显示在屏幕上，这常常用于输入密码或保密信息。
- -u fd：使用文件描述符fd作为输入。

11.3 IFS 变量

read命令读取的值，默认是以空格分隔。可以通过自定义环境变量IFS（内部字段分隔符，Internal Field Separator 的缩写），修改分隔标志。

IFS的默认值是空格、Tab 符号、换行符号，通常取第一个（即空格）。

如果把IFS定义成冒号（:）或分号（;），就可以分隔以这两个符号分隔的值，这对读取文件很有用。

```
#!/bin/bash
# read-ifs: read fields from a file

FILE=/etc/passwd

read -p "Enter a username > " user_name
file_info="$(grep "^$user_name:" $FILE)"

if [ -n "$file_info" ]; then
  IFS=":" read user pw uid gid name home shell <<< "$file_info"
  echo "User = '$user'"
  echo "UID = '$uid'"
  echo "GID = '$gid'"
  echo "Full Name = '$name'"
  echo "Home Dir. = '$home'"
  echo "Shell= '$shell'"
else
  echo "No such user '$user_name'" >&2
  exit 1
fi
```

上面例子中，IFS设为冒号，然后用来分解/etc/passwd文件的一行。IFS的赋值命令和read命令写在一行，这样的话，IFS的改变仅对后面的命令生效，该命令执行后IFS会自动恢复原来的值。
如果不写在一行，就要采用下面的写法。

```
OLD_IFS="$IFS"
IFS=":"
read user pw uid gid name home shell <<< "$file_info"
IFS="$OLD_IFS"
```

另外，上面例子中，<<<是 Here 字符串，用于将变量值转为标准输入，因为read命令只能解析标准输入。

如果IFS设为空字符串，就等同于将整行读入一个变量。

```
#!/bin/bash
input="/path/to/txt/file"
while IFS= read -r line
do
  echo "$line"
done < "$input"
```

上面的命令可以逐行读取文件，每一行存入变量line，打印出来以后再读取下一行。

12. 条件判断

本章介绍Bash脚本的条件判断语法。

12.1 if 结构

if是最常用的条件判断结构，只有符合给定条件时，才会执行指定的命令。它的语法如下。

```
if commands; then
  commands
[elif commands; then
  commands...]
[else
  commands]
fi
```

这个命令分成三个部分：if、elif和else。其中，后两个部分是可选的。

if关键字后面是主要的判断条件，elif用来添加在主条件不成立时的其他判断条件，else则是所有条件都不成立时要执行的部分。

```
if test $USER = "foo"; then
  echo "Hello foo."
else
  echo "You are not foo."
fi
```

上面的例子中，判断条件是环境变量$USER是否等于foo，如果等于就输出Hello foo.，否则输出其他内容。

if和then写在同一行时，需要分号分隔。分号是Bash的命令分隔符。它们也可以写成两行，这时不需要分号。

```
if true
then
  echo 'hello world'
fi

if false
then
  echo 'it is false' #
fi
```

上面的例子中，true和false是两个特殊命令，前者代表操作成功，后者代表操作失败。if true 意味着命令部分总是会执行，if false意味着命令部分永远不会执行。

除了多行的写法，if结构也可以写成单行。

```
$ if true; then echo 'hello world'; fi
hello world

$ if false; then echo "It's true."; fi
```

注意，if关键字后面也可以是一条命令，该条命令执行成功（返回值0），就意味着判断条件成立。

```
$ if echo 'hi'; then echo 'hello world'; fi
hi
```

```
hello world
```

上面命令中，if后面是一条命令echo 'hi'。该命令会执行，如果返回值是0，则执行then的部分。

if后面可以跟任意数量的命令。这时，所有命令都会执行，但是判断真伪只看最后一个命令，即使前面所有命令都失败，只要最后一个命令返回0，就会执行then的部分。

```
$ if false; true; then echo 'hello world'; fi
hello world
```

上面例子中，if后面有两条命令（false;true;），第二条命令（true）决定了then的部分是否会执行。

elif部分可以有多个。

```
#!/bin/bash

echo -n "  1 3      > "
read character
if [ "$character" = "1" ]; then
    echo 1
elif [ "$character" = "2" ]; then
    echo 2
elif [ "$character" = "3" ]; then
    echo 3
else
    echo
fi
```

上面例子中，如果用户输入3，就会连续判断3次。

12.2 test 命令

if结构的判断条件，一般使用test命令，有三种形式。

```
#
test expression

#
[ expression ]

#
[[ expression ]]
```

上面三种形式是等价的，但是第三种形式还支持正则判断，前两种不支持。

上面的expression是一个表达式。这个表达式为真，test命令执行成功（返回值为0）；表达式为伪，test命令执行失败（返回值为1）。注意，第二种和第三种写法，[和]与内部的表达式之间必须有空格。

```
$ test -f /etc/hosts
$ echo $?
0
```

```
$ [ -f /etc/hosts ]
$  echo $?
0
```

上面的例子中，test命令采用两种写法，判断/etc/hosts文件是否存在，这两种写法是等价的。命令执行后，返回值为0，表示该文件确实存在。

下面把test命令的三种形式，用在if结构中，判断一个文件是否存在。

```
#
if test -e /tmp/foo.txt ; then
  echo "Found foo.txt"
fi

#
if [ -e /tmp/foo.txt ] ; then
  echo "Found foo.txt"
fi

#
if [[ -e /tmp/foo.txt ]] ; then
  echo "Found foo.txt"
fi
```

12.3 判断表达式

if关键字后面，跟的是一个命令。这个命令可以是test命令，也可以是其他命令。命令的返回值为0表示判断成立，否则表示不成立。因为这些命令主要是为了得到返回值，所以可以视为表达式。

常用的判断表达式有下面这些。

文件判断

以下表达式用来判断文件状态。

- [-a file]：如果 file 存在，则为true。
- [-b file]：如果 file 存在并且是一个块（设备）文件，则为true。
- [-c file]：如果 file 存在并且是一个字符（设备）文件，则为true。
- [-d file]：如果 file 存在并且是一个目录，则为true。
- [-e file]：如果 file 存在，则为true。
- [-f file]：如果 file 存在并且是一个普通文件，则为true。
- [-g file]：如果 file 存在并且设置了组 ID，则为true。
- [-G file]：如果 file 存在并且属于有效的组 ID，则为true。
- [-h file]：如果 file 存在并且是符号链接，则为true。
- [-k file]：如果 file 存在并且设置了它的“sticky bit”，则为true。
- [-L file]：如果 file 存在并且是一个符号链接，则为true。
- [-N file]：如果 file 存在并且自上次读取后已被修改，则为true。
- [-O file]：如果 file 存在并且属于有效的用户 ID，则为true。

- `[ -p file ]`：如果 file 存在并且是一个命名管道，则为`true`。
- `[ -r file ]`：如果 file 存在并且可读（当前用户有可读权限），则为`true`。
- `[ -s file ]`：如果 file 存在且其长度大于零，则为`true`。
- `[ -S file ]`：如果 file 存在且是一个网络 socket，则为`true`。
- `[ -t fd ]`：如果 fd 是一个文件描述符，并且重定向到终端，则为`true`。这可以用来判断是否重定向了标准输入 / 输出错误。
- `[ -u file ]`：如果 file 存在并且设置了 setuid 位，则为`true`。
- `[ -w file ]`：如果 file 存在并且可写（当前用户拥有可写权限），则为`true`。
- `[ -x file ]`：如果 file 存在并且可执行（有效用户有执行 / 搜索权限），则为`true`。
- `[ file1 -nt file2 ]`：如果 FILE1 比 FILE2 的更新时间最近，或者 FILE1 存在而 FILE2 不存在，则为`true`。
- `[ file1 -ot file2 ]`：如果 FILE1 比 FILE2 的更新时间更旧，或者 FILE2 存在而 FILE1 不存在，则为`true`。
- `[ FILE1 -ef FILE2 ]`：如果 FILE1 和 FILE2 引用相同的设备和 inode 编号，则为`true`。

下面是一个示例。

```
#!/bin/bash

FILE=~/.bashrc

if [ -e "$FILE" ]; then
  if [ -f "$FILE" ]; then
    echo "$FILE is a regular file."
  fi
  if [ -d "$FILE" ]; then
    echo "$FILE is a directory."
  fi
  if [ -r "$FILE" ]; then
    echo "$FILE is readable."
  fi
  if [ -w "$FILE" ]; then
    echo "$FILE is writable."
  fi
  if [ -x "$FILE" ]; then
    echo "$FILE is executable/searchable."
  fi
else
  echo "$FILE does not exist"
  exit 1
fi
```

上面代码中，`$FILE`要放在双引号之中。这样可以防止`$FILE`为空，因为这时`[ -e ]`会判断为真。

而放在双引号之中，返回的就总是一个空字符串，`[ -e "" ]`会判断为伪。

字符串判断

以下表达式用来判断字符串。

- `[ string ]`：如果string不为空（长度大于0），则判断为真。
- `[ -n string ]`：如果字符串string的长度大于零，则判断为真。
- `[ -z string ]`：如果字符串string的长度为零，则判断为真。
- `[ string1 = string2 ]`：如果string1和string2相同，则判断为真。
- `[ string1 == string2 ]` 等同于`[ string1 = string2 ]`。
- `[ string1 != string2 ]`：如果string1和string2不相同，则判断为真。
- `[ string1 '>' string2 ]`：如果按照字典顺序string1排列在string2之后，则判断为真。
- `[ string1 '<' string2 ]`：如果按照字典顺序string1排列在string2之前，则判断为真。

注意，test命令内部的>和<，必须用引号引起来（或者是用反斜杠转义）。
否则，它们会被shell解释为重定向操作符。

下面是一个示例。

```
#!/bin/bash

ANSWER=maybe

if [ -z "$ANSWER" ]; then
  echo "There is no answer." >&2
  exit 1
fi
if [ "$ANSWER" = "yes" ]; then
  echo "The answer is YES."
elif [ "$ANSWER" = "no" ]; then
  echo "The answer is NO."
elif [ "$ANSWER" = "maybe" ]; then
  echo "The answer is MAYBE."
else
  echo "The answer is UNKNOWN."
fi
```

上面代码中，首先确定$ANSWER字符串是否为空。如果为空，就终止脚本，并把退出状态设为1。
注意，这里的echo命令把错误信息There is no answer.重定向到标准错误，这是处理错误信息的常用方法。
如果$ANSWER字符串不为空，就判断它的值是否等于yes、no或者maybe。

注意，字符串判断时，变量要放在双引号之中，比如`[ -n "$COUNT" ]`，否则变量替换成字符串以后，
test命令可能会报错，提示参数过多。另外，如果不放在双引号之中，变量为空时，命令会变成
`[ -n ]`，这时会判断为真。如果放在双引号之中，`[ -n "" ]`就判断为伪。

整数判断

下面的表达式用于判断整数。

- `[ integer1 -eq integer2 ]`：如果integer1等于integer2，则为true。
- `[ integer1 -ne integer2 ]`：如果integer1不等于integer2，则为true。

- [integer1 -le integer2]：如果integer1小于或等于integer2，则为true。
- [integer1 -lt integer2]：如果integer1小于integer2，则为true。
- [integer1 -ge integer2]：如果integer1大于或等于integer2，则为true。
- [integer1 -gt integer2]：如果integer1大于integer2，则为true。

下面是一个用法的例子。

```
#!/bin/bash

INT=-5

if [ -z "$INT" ]; then
  echo "INT is empty." >&2
  exit 1
fi
if [ $INT -eq 0 ]; then
  echo "INT is zero."
else
  if [ $INT -lt 0 ]; then
    echo "INT is negative."
  else
    echo "INT is positive."
  fi
  if [ $((INT % 2)) -eq 0 ]; then
    echo "INT is even."
  else
    echo "INT is odd."
  fi
fi
```

上面例子中，先判断变量$INT是否为空，然后判断是否为0，接着判断正负，最后通过求余数判断奇偶。

正则判断

[[expression]]这种判断形式，支持正则表达式。

```
[[ string1 =~ regex ]]
```

上面的语法中，regex是一个正则表示式，=~是正则比较运算符。

下面是一个例子。

```
#!/bin/bash

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
  echo "INT is an integer."
  exit 0
```

```
else
  echo "INT is not an integer." >&2
  exit 1
fi
```

上面代码中，先判断变量INT的字符串形式，是否满足^-?[0-9]+$的正则模式，如果满足就表明它是一个整数。

test 判断的逻辑运算

通过逻辑运算，可以把多个test判断表达式结合起来，创造更复杂的判断。三种逻辑运算AND，OR，和NOT，都有自己的专用符号。

- AND运算：符号&&，也可使用参数-a。
- OR运算：符号||，也可使用参数-o。
- NOT运算：符号!。

下面是一个AND的例子，判断整数是否在某个范围之内。

```
#!/bin/bash

MIN_VAL=1
MAX_VAL=100

INT=50

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
  if [[ $INT -ge $MIN_VAL && $INT -le $MAX_VAL ]]; then
    echo "$INT is within $MIN_VAL to $MAX_VAL."
  else
    echo "$INT is out of range."
  fi
else
  echo "INT is not an integer." >&2
  exit 1
fi
```

上面例子中，&&用来连接两个判断条件：大于等于$MIN_VAL，并且小于等于$MAX_VAL。

使用否定操作符!时，最好用圆括号确定转义的范围。

```
if [ ! \( $INT -ge $MIN_VAL -a $INT -le $MAX_VAL \) ]; then
    echo "$INT is outside $MIN_VAL to $MAX_VAL."
else
    echo "$INT is in range."
fi
```

上面例子中，test命令内部使用的圆括号，必须使用引号或者转义，否则会被Bash解释。

算术判断

Bash还提供了((...))作为算术条件，进行算术运算的判断。

```
if ((3 > 2)); then
  echo "true"
fi
```

上面代码执行后，会打印出true。

注意，算术判断不需要使用test命令，而是直接使用((...))结构。这个结构的返回值，决定了判断的真伪。

如果算术计算的结果是非零值，则表示判断成立。这一点跟命令的返回值正好相反，需要小心。

```
$ if ((1)); then echo "It is true."; fi
It is true.
$ if ((0)); then echo "It is true."; else echo "it is false."; fi
It is false.
```

上面例子中，((1))表示判断成立，((0))表示判断不成立。

算术条件((...))也可以用于变量赋值。

```
$ if (( foo = 5 ));then echo "foo is $foo"; fi
foo is 5
```

上面例子中，((foo = 5))完成了两件事情。首先把5赋值给变量foo，然后根据返回值5，判断条件为真。

注意，赋值语句返回等号右边的值，如果返回的是0，则判断为假。

```
$ if (( foo = 0 ));then echo "It is true.";else echo "It is false."; fi
It is false.
```

下面是用算术条件改写的数值判断脚本。

```
#!/bin/bash

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
  if ((INT == 0)); then
    echo "INT is zero."
  else
    if ((INT < 0)); then
      echo "INT is negative."
    else
      echo "INT is positive."
    fi
    if (( ((INT % 2)) == 0)); then
      echo "INT is even."
    else
      echo "INT is odd."
```

```
    fi
  fi
else
  echo "INT is not an integer." >&2
  exit 1
fi
```

只要是算术表达式，都能用于((...))语法，详见《Bash的算术运算》一章。

普通命令的逻辑运算

如果if结构使用的不是test命令，而是普通命令，比如上一节的((...))算术运算，
或者test命令与普通命令混用，那么可以使用Bash的命令控制操作符&&（AND）和||（OR），
进行多个命令的逻辑运算。

```
$ command1 && command2
$ command1 || command2
```

对于&&操作符，先执行command1，只有command1执行成功后， 才会执行command2。对于||操作符，
先执行command1，只有command1执行失败后， 才会执行command2。

```
$ mkdir temp && cd temp
```

上面的命令会创建一个名为temp的目录，执行成功后，才会执行第二个命令，进入这个目录。

```
$ [ -d temp ] || mkdir temp
```

上面的命令会测试目录temp是否存在，如果不存在，就会执行第二个命令，创建这个目录。
这种写法非常有助于在脚本中处理错误。

```
[ ! -d temp ] && exit 1
```

上面的命令中，如果temp子目录不存在，脚本会终止，并且返回值为1。

下面就是if与&&结合使用的写法。

```
if [ condition ] && [ condition ]; then
  command
fi
```

下面是一个示例。

```
#! /bin/bash

filename=$1
word1=$2
word2=$3

if grep $word1 $filename && grep $word2 $filename
then
  echo "$word1 and $word2 are both in $filename."
fi
```

上面的例子只有在指定文件里面，同时存在搜索词word1和word2，就会执行if的命令部分。

下面的示例演示如何将一个&&判断表达式，改写成对应的if结构。

```
[[ -d "$dir_name" ]] && cd "$dir_name" && rm *

#

if [[ ! -d "$dir_name" ]]; then
  echo "No such directory: '$dir_name'" >&2
  exit 1
fi
if ! cd "$dir_name"; then
  echo "Cannot cd to '$dir_name'" >&2
  exit 1
fi
if ! rm *; then
  echo "File deletion failed. Check results" >&2
  exit 1
fi
```

12.4 case 结构

case结构用于多值判断，可以为每个值指定对应的命令，跟包含多个elif的if结构等价，但是语义更好。它的语法如下。

```
case expression in
  pattern )
    commands ;;
  pattern )
    commands ;;
  ...
esac
```

上面代码中，expression是一个表达式，pattern是表达式的值或者一个模式，可以有多条，用来匹配多个值，每条以两个分号（;）结尾。

```
#!/bin/bash

echo -n "  1 3      > "
read character
case $character in
  1 ) echo 1
    ;;
  2 ) echo 2
    ;;
  3 ) echo 3
    ;;
```

```
  * ) echo
esac
```

上面例子中，最后一条匹配语句的模式是*，这个通配符可以匹配其他字符和没有输入字符的情况，类似if的else部分。

下面是另一个例子。

```
#!/bin/bash

OS=$(uname -s)

case "$OS" in
  FreeBSD) echo "This is FreeBSD" ;;
  Darwin) echo "This is Mac OSX" ;;
  AIX) echo "This is AIX" ;;
  Minix) echo "This is Minix" ;;
  Linux) echo "This is Linux" ;;
  *) echo "Failed to identify this OS" ;;
esac
```

上面的例子判断当前是什么操作系统。

case的匹配模式可以使用各种通配符，下面是一些例子。

- a)：匹配a。
- a|b)：匹配a或b。
- [[:alpha:]])：匹配单个字母。
- ???)：匹配3个字符的单词。
- *.txt)：匹配.txt结尾。
- *)：匹配任意输入，通过作为case结构的最后一个模式。

```
#!/bin/bash

echo -n "      > "
read character
case $character in
  [[:lower:]] | [[:upper:]] ) echo "    $character"
                              ;;
  [0-9] )                     echo "    $character"
                              ;;
  * )                         echo "    "
esac
```

上面例子中，使用通配符[[:lower:]] | [[:upper:]]匹配字母，[0-9]匹配数字。

Bash4.0之前，case结构只能匹配一个条件，然后就会退出case结构。Bash4.0之后，允许匹配多个条件，这时可以用;;&终止每个条件块。

```
#!/bin/bash
# test.sh
```

```
read -n 1 -p "Type a character > "
echo
case $REPLY in
  [[:upper:]])    echo "'$REPLY' is upper case." ;;&
  [[:lower:]])    echo "'$REPLY' is lower case." ;;&
  [[:alpha:]])    echo "'$REPLY' is alphabetic." ;;&
  [[:digit:]])    echo "'$REPLY' is a digit." ;;&
  [[:graph:]])    echo "'$REPLY' is a visible character." ;;&
  [[:punct:]])    echo "'$REPLY' is a punctuation symbol." ;;&
  [[:space:]])    echo "'$REPLY' is a whitespace character." ;;&
  [[:xdigit:]])   echo "'$REPLY' is a hexadecimal digit." ;;&
esac
```

执行上面的脚本，会得到下面的结果。

```
$ test.sh
Type a character > a
'a' is lower case.
'a' is alphabetic.
'a' is a visible character.
'a' is a hexadecimal digit.
```

可以看到条件语句结尾添加了;;&以后，在匹配一个条件之后，并没有退出case结构，而是继续判断下一个条件。

12.5 参考链接

- TheLinuxCommand Line, William Shotts

13. 循环

Bash提供三种循环语法for、while和until。

13.1 while 循环

while循环有一个判断条件，只要符合条件，就不断循环执行指定的语句。

```
while condition; do
  commands
done
```

上面代码中，只要满足条件condition，就会执行命令commands。然后，再次判断是否满足条件condition，只要满足，就会一直执行下去。只有不满足条件，才会退出循环。

循环条件condition可以使用test命令，跟if结构的判断条件写法一致。

```
#!/bin/bash

number=0
```

```
while [ "$number" -lt 10 ]; do
  echo "Number = $number"
  number=$((number + 1))
done
```

上面例子中，只要变量$number小于10，就会不断加1，直到$number等于10，然后退出循环。

关键字do可以跟while不在同一行，这时两者之间不需要使用分号分隔。

```
while true
do
  echo 'Hi, while looping ...';
done
```

上面的例子会无限循环，可以按下 Ctrl + c 停止。

while循环写成一行，也是可以的。

```
$ while true; do echo 'Hi, while looping ...'; done
```

while的条件部分也可以是执行一个命令。

```
$ while echo 'ECHO'; do echo 'Hi, while looping ...'; done
```

上面例子中，判断条件是echo 'ECHO'。由于这个命令总是执行成功，所以上面命令会产生无限循环。

while的条件部分可以执行任意数量的命令，但是执行结果的真伪只看最后一个命令的执行结果。

```
$ while true; false; do echo 'Hi, looping ...'; done
```

上面代码运行后，不会有任何输出，因为while的最后一个命令是false。

13.2 until 循环

until循环与while循环恰好相反，只要不符合判断条件（判断条件失败），就不断循环执行指定的语句。一旦符合判断条件，就退出循环。

```
until condition; do
  commands
done
```

关键字do可以与until不写在同一行，这时两者之间不需要分号分隔。

```
until condition
do
  commands
done
```

下面是一个例子。

```
$ until false; do echo 'Hi, until looping ...'; done
Hi, until looping ...
Hi, until looping ...
```

```
Hi, until looping ...
^C
```

上面代码中，until的部分一直为false，导致命令无限运行，必须按下 Ctrl + c 终止。

```
#!/bin/bash

number=0
until [ "$number" -ge 10 ]; do
  echo "Number = $number"
  number=$((number + 1))
done
```

上面例子中，只要变量number小于10，就会不断加1，直到number大于等于10，就退出循环。

until的条件部分也可以是一个命令，表示在这个命令执行成功之前，不断重复尝试。

```
until cp $1 $2; do
  echo 'Attempt to copy failed. waiting...'
  sleep 5
done
```

上面例子表示，只要cp $1 $2这个命令执行不成功，就5分钟后再尝试一次，直到成功为止。

until循环都可以转为while循环，只要把条件设为否定即可。上面这个例子可以改写如下。

```
while ! cp $1 $2; do
  echo 'Attempt to copy failed. waiting...'
  sleep 5
done
```

一般来说，until用得比较少，完全可以统一都使用while。

13.3 for…in 循环

for...in循环用于遍历列表的每一项。

```
for variable in list
do
  commands
done
```

上面语法中，for循环会依次从list列表中取出一项，作为变量variable，然后在循环体中进行处理。

关键词do可以跟for写在同一行，两者使用分号分隔。

```
for variable in list; do
  commands
done
```

下面是一个例子。

```
#!/bin/bash

for i in word1 word2 word3; do
  echo $i
done
```

上面例子中，word1 word2 word3是一个包含三个单词的列表，变量i依次等于word1、word2、word3，命令echo $i则会相应地执行三次。

列表可以由通配符产生。

```
for i in *.png; do
  ls -l $i
done
```

上面例子中，*.png会替换成当前目录中所有 PNG 图片文件，变量i会依次等于每一个文件。

列表也可以通过子命令产生。

```
#!/bin/bash

count=0
for i in $(cat ~/.bash_profile); do
  count=$((count + 1))
  echo "Word $count ($i) contains $(echo -n $i | wc -c) characters"
done
```

上面例子中，cat ~/.bash_profile命令会输出~/.bash_profile文件的内容，然后通过遍历每一个词，计算该文件一共包含多少个词，以及每个词有多少个字符。

in list的部分可以省略，这时list默认等于脚本的所有参数$@。但是，为了可读性，最好还是不要省略，参考下面的例子。

```
for filename; do
  echo "$filename"
done

#

for filename in "$@" ; do
  echo "$filename"
done
```

在函数体中也是一样的，for...in循环省略in list的部分，则list默认等于函数的所有参数。

13.4 for 循环

for循环还支持 C 语言的循环语法。

```
for (( expression1; expression2; expression3 )); do
  commands
```

```
done
```

上面代码中，expression1用来初始化循环条件，expression2用来决定循环结束的条件，expression3在每次循环迭代的末尾执行，用于更新值。

注意，循环条件放在双重圆括号之中。另外，圆括号之中使用变量，不必加上美元符号$。

它等同于下面的while循环。

```
(( expression1 ))
while (( expression2 )); do
  commands
  (( expression3 ))
done
```

下面是一个例子。

```
for (( i=0; i<5; i=i+1 )); do
  echo $i
done
```

上面代码中，初始化变量i的值为0，循环执行的条件是i小于5。每次循环迭代结束时，i的值加1。

for条件部分的三个语句，都可以省略。

```
for ((;;))
do
  read var
  if [ "$var" = "." ]; then
    break
  fi
done
```

上面脚本会反复读取命令行输入，直到用户输入了一个点（.）位为止，才会跳出循环。

13.5 break，continue

Bash提供了两个内部命令break和continue，用来在循环内部跳出循环。

break命令立即终止循环，程序继续执行循环块之后的语句，即不再执行剩下的循环。

```
#!/bin/bash

for number in 1 2 3 4 5 6
do
  echo "number is $number"
  if [ "$number" = "3" ]; then
    break
  fi
done
```

上面例子只会打印3行结果。一旦变量$number等于3，就会跳出循环，不再继续执行。

continue命令立即终止本轮循环，开始执行下一轮循环。

```
#!/bin/bash

while read -p "What file do you want to test?" filename
do
  if [ ! -e "$filename" ]; then
    echo "The file does not exist."
    continue
  fi

  echo "You entered a valid file.."
done
```

上面例子中，只要用户输入的文件不存在，continue命令就会生效，直接进入下一轮循环（让用户重新输入文件名），不再执行后面的打印语句。

13.6 select 结构

select结构主要用来生成简单的菜单。它的语法与for...in循环基本一致。

```
select name
[in list]
do
  commands
done
```

Bash会对select依次进行下面的处理。

1. select生成一个菜单，内容是列表list的每一项，并且每一项前面还有一个数字编号。
   1.Bash提示用户选择一项，输入它的编号。
2. 用户输入以后，Bash会将该项的内容存在变量name，该项的编号存入环境变量REPLY。如果用户没有输入，就按回车键，Bash会重新输出菜单，让用户选择。
3. 执行命令体commands。
4. 执行结束后，回到第一步，重复这个过程。

下面是一个例子。

```
#!/bin/bash
# select.sh

select brand in Samsung Sony iphone symphony Walton
do
  echo "You have chosen $brand"
done
```

执行上面的脚本，Bash会输出一个品牌的列表，让用户选择。

```
$ ./select.sh
1) Samsung
```

```
2) Sony
3) iphone
4) symphony
5) Walton
#?
```

如果用户没有输入编号，直接按回车键。Bash就会重新输出一遍这个菜单，直到用户按下`Ctrl + c`，退出执行。

`select`可以与`case`结合，针对不同项，执行不同的命令。

```
#!/bin/bash

echo "Which Operating System do you like?"

select os in Ubuntu LinuxMint Windows8 Windows7 WindowsXP
do
  case $os in
    "Ubuntu"|"LinuxMint")
      echo "I also use $os."
    ;;
    "Windows8" | "Windows10" | "WindowsXP")
      echo "Why don't you try Linux?"
    ;;
    *)
      echo "Invalid entry."
      break
    ;;
  esac
done
```

上面例子中，`case`针对用户选择的不同项，执行不同的命令。

13.7 参考链接

- BashSelect Command, Fahmida Yesmin

14. Bash函数

本章介绍Bash函数的用法。

14.1 简介

函数（function）是可以重复使用的代码片段，有利于代码的复用。它与别名（alias）的区别是，别名只适合封装简单的单个命令，函数则可以封装复杂的多行命令。

函数总是在当前Shell执行，这是跟脚本的一个重大区别，Bash会新建一个子Shell执行脚本。如果函数与脚本同名，函数会优先执行。但是，函数的优先级不如别名，即如果函数与别名同名，那么别名优先执行。

Bash函数定义的语法有两种。

```
#
fn() {
  # codes
}

#
function fn() {
  # codes
}
```

上面代码中，fn是自定义的函数名，函数代码就写在大括号之中。这两种写法是等价的。

下面是一个简单函数的例子。

```
hello() {
  echo "Hello $1"
}
```

上面代码中，函数体里面的$1表示函数调用时的第一个参数。

调用时，就直接写函数名，参数跟在函数名后面。

```
$ hello world
hello world
```

下面是一个多行函数的例子，显示当前日期时间。

```
today() {
  echo -n "Today's date is: "
  date +"%A, %B %-d, %Y"
}
```

删除一个函数，可以使用unset命令。

```
unset -f functionName
```

查看当前Shell已经定义的所有函数，可以使用declare命令。

```
$ declare -f
```

上面的declare命令不仅会输出函数名，还会输出所有定义。输出顺序是按照函数名的字母表顺序。由于会输出很多内容，最好通过管道命令配合more或less使用。

declare命令还支持查看单个函数的定义。

```
$ declare -f functionName
```

declare -F可以输出所有已经定义的函数名，不含函数体。

```
$ declare -F
```

14.2 参数变量

函数体内可以使用参数变量，获取函数参数。函数的参数变量，与脚本参数变量是一致的。

- $1~$9：函数的第一个到第9个的参数。
- $0：函数所在的脚本名。
- $#：函数的参数总数。
- $@：函数的全部参数，参数之间使用空格分隔。
- $*：函数的全部参数，参数之间使用变量$IFS值的第一个字符分隔，默认为空格，但是可以自定义。

如果函数的参数多于9个，那么第10个参数可以用${10}的形式引用，以此类推。

下面是一个示例脚本test.sh。

```
#!/bin/bash
# test.sh

function alice {
  echo "alice: $@"
  echo "$0: $1 $2 $3 $4"
  echo "$# arguments"

}

alice in wonderland
```

运行该脚本，结果如下。

```
$ bash test.sh
alice: in wonderland
test.sh: in wonderland
2 arguments
```

上面例子中，由于函数alice只有第一个和第二个参数，所以第三个和第四个参数为空。

下面是一个日志函数的例子。

```
function log_msg {
  echo "[`date '+ %F %T'` ]: $@"
}
```

使用方法如下。

```
$ log_msg "This is sample log message"
[ 2018-08-16 19:56:34 ]: This is sample log message
```

14.3 return 命令

return命令用于从函数返回一个值。函数执行到这条命令，就不再往下执行了，直接返回了。

```
function func_return_value {
  return 10
```

```
}
```

函数将返回值返回给调用者。如果命令行直接执行函数，下一个命令可以用$?拿到返回值。

```
$ func_return_value
$ echo "Value returned by function is: $?"
Value returned by function is: 10
```

return后面不跟参数，只用于返回也是可以的。

```
function name {
  commands
  return
}
```

14.4 全局变量和局部变量，local 命令

Bash函数体内直接声明的变量，属于全局变量，整个脚本都可以读取。这一点需要特别小心。

```
#   test.sh
fn () {
  foo=1
  echo "fn: foo = $foo"
}

fn
echo "global: foo = $foo"
```

上面脚本的运行结果如下。

```
$ bash test.sh
fn: foo = 1
global: foo = 1
```

上面例子中，变量$foo是在函数fn内部声明的，函数体外也可以读取。

函数体内不仅可以声明全局变量，还可以修改全局变量。

```
foo=1

fn () {
  foo=2
}

echo $foo
```

上面代码执行后，输出的变量$foo值为2。

函数里面可以用local命令声明局部变量。

```
#   test.sh
fn () {
```

```
  local foo
  foo=1
  echo "fn: foo = $foo"
}

fn
echo "global: foo = $foo"
```

上面脚本的运行结果如下。

```
$ bash test.sh
fn: foo = 1
global: foo =
```

上面例子中，local命令声明的$foo变量，只在函数体内有效，函数体外没有定义。

14.5 参考链接

- How to define and use functions in LinuxShellScript, by Pradeep Kumar

15. 数组

数组（array）是一个包含多个值的变量。成员的编号从0开始，数量没有上限，也没有要求成员被连续索引。

15.1 创建数组

数组可以采用逐个赋值的方法创建。

```
ARRAY[INDEX]=value
```

上面语法中，ARRAY是数组的名字，可以是任意合法的变量名。INDEX是一个大于或等于零的整数，也可以是算术表达式。注意数组第一个元素的下标是0， 而不是1。

下面创建一个三个成员的数组。

```
$ array[0]=val
$ array[1]=val
$ array[2]=val
```

数组也可以采用一次性赋值的方式创建。

```
ARRAY=(value1 value2 ... valueN)

#

ARRAY=(
  value1
  value2
  value3
)
```

采用上面方式创建数组时，可以按照默认顺序赋值，也可以在每个值前面指定位置。

```
$ array=(a b c)
$ array=([2]=c [0]=a [1]=b)

$ days=(Sun Mon Tue Wed Thu Fri Sat)
$ days=([0]=Sun [1]=Mon [2]=Tue [3]=Wed [4]=Thu [5]=Fri [6]=Sat)
```

只为某些值指定位置，也是可以的。

```
names=(hatter [5]=duchess alice)
```

上面例子中，hatter是数组的0号位置，duchess是5号位置，alice是6号位置。

没有赋值的数组元素的默认值是空字符串。

定义数组的时候，可以使用通配符。

```
$ mp3s=( *.mp3 )
```

上面例子中，将当前目录的所有 MP3 文件，放进一个数组。

先用declare -a命令声明一个数组，也是可以的。

```
$ declare -a ARRAYNAME
```

read -a命令则是将用户的命令行输入，读入一个数组。

```
$ read -a dice
```

上面命令将用户的命令行输入，读入数组dice。

15.2 读取数组

读取单个元素

读取数组指定位置的成员，要使用下面的语法。

```
$ echo ${array[i]}     # i
```

上面语法里面的大括号是必不可少的，否则Bash会把索引部分[i]按照原样输出。

```
$ array[0]=a

$ echo ${array[0]}
a

$ echo $array[0]
a[0]
```

上面例子中，数组的第一个元素是a。如果不加大括号，Bash会直接读取$array首成员的值，然后将[0]按照原样输出。

读取所有成员

@和*是数组的特殊索引，表示返回数组的所有成员。

```
$ foo=(a b c d e f)
$ echo ${foo[@]}
a b c d e f
```

这两个特殊索引配合for循环，就可以用来遍历数组。

```
for i in "${names[@]}"; do
  echo $i
done
```

@和*放不放在双引号之中，是有差别的。

```
$ activities=( swimming "water skiing" canoeing "white-water rafting" surfing )
$ for act in ${activities[@]}; \
do \
echo "Activity: $act"; \
done

Activity: swimming
Activity: water
Activity: skiing
Activity: canoeing
Activity: white-water
Activity: rafting
Activity: surfing
```

上面的例子中，数组activities实际包含5个元素，但是for...in循环直接遍历${activities[@]}，会导致返回7个结果。为了避免这种情况，一般把${activities[@]}放在双引号之中。

```
$ for act in "${activities[@]}"; \
do \
echo "Activity: $act"; \
done

Activity: swimming
Activity: water skiing
Activity: canoeing
Activity: white-water rafting
Activity: surfing
```

上面例子中，${activities[@]}放在双引号之中，遍历就会返回正确的结果。

${activities[*]}不放在双引号之中，跟${activities[@]}不放在双引号之中是一样的。

```
$ for act in ${activities[*]}; \
do \
echo "Activity: $act"; \
```

```
done

Activity: swimming
Activity: water
Activity: skiing
Activity: canoeing
Activity: white-water
Activity: rafting
Activity: surfing
```

${activities[*]}放在双引号之中，所有元素就会变成单个字符串返回。

```
$ for act in "${activities[*]}"; \
do \
echo "Activity: $act"; \
done

Activity: swimming water skiing canoeing white-water rafting surfing
```

所以，拷贝一个数组的最方便方法，就是写成下面这样。

```
$ hobbies=( "${activities[@]}" )
```

上面例子中，数组activities被拷贝给了另一个数组hobbies。

这种写法也可以用来为新数组添加成员。

```
$ hobbies=( "${activities[@]" diving )
```

上面例子中，新数组hobbies在数组activities的所有成员之后，又添加了一个成员。

默认位置

如果读取数组成员时，没有读取指定哪一个位置的成员，默认使用0号位置。

```
$ declare -a foo
$ foo=A
$ echo ${foo[0]}
A
```

上面例子中，foo是一个数组，赋值的时候不指定位置，实际上是给foo[0]赋值。

引用一个不带下标的数组变量，则引用的是0号位置的数组元素。

```
$ foo=(a b c d e f)
$ echo ${foo}
a
$ echo $foo
a
```

上面例子中，引用数组元素的时候，没有指定位置，结果返回的是0号位置。

15.3 数组的长度

要想知道数组的长度（即一共包含多少成员），可以使用下面两种语法。

```
${#array[*]}
${#array[@]}
```

下面是一个例子。

```
$ a[100]=foo

$ echo ${#a[*]}
1

$ echo ${#a[@]}
1
```

上面例子中，把字符串赋值给100位置的数组元素，这时的数组只有一个元素。

注意，如果用这种语法去读取具体的数组成员，就会返回该成员的字符串长度。这一点必须小心。

```
$ a[100]=foo
$ echo ${#a[100]}
3
```

上面例子中，`${#a[100]}`实际上是返回数组第100号成员`a[100]`的值（`foo`）的字符串长度。

15.4 提取数组序号

`${!array[@]}`或`${!array[*]}`，可以返回数组的成员序号，即哪些位置是有值的。

```
$ arr=([5]=a [9]=b [23]=c)
$ echo ${!arr[@]}
5 9 23
$ echo ${!arr[*]}
5 9 23
```

上面例子中，数组的5、9、23号位置有值。

利用这个语法，也可以通过`for`循环遍历数组。

```
arr=(a b c d)

for i in ${!arr[@]};do
  echo ${arr[i]}
done
```

15.5 提取数组成员

`${array[@]:position:length}`的语法可以提取数组成员。

```
$ food=( apples bananas cucumbers dates eggs fajitas grapes )
$ echo ${food[@]:1:1}
bananas
$ echo ${food[@]:1:3}
bananas cucumbers dates
```

上面例子中，`${food[@]:1:1}`返回从数组1号位置开始的1个成员，`${food[@]:1:3}`返回1号位置开始的3个成员。

如果省略长度参数`length`，则返回从指定位置开始的所有成员。

```
$ echo ${food[@]:4}
eggs fajitas grapes
```

上面例子返回从4号位置开始到结束的所有成员。

15.6 追加数组成员

数组末尾追加成员，可以使用`+=`赋值运算符。它能够自动地把值追加到数组末尾。
否则，就需要知道数组的最大序号，比较麻烦。

```
$ foo=(a b c)
$ echo ${foo[@]}
a b c

$ foo+=(d e f)
$ echo ${foo[@]}
a b c d e f
```

15.7 删除数组

删除一个数组成员，使用`unset`命令。

```
$ foo=(a b c d e f)
$ echo ${foo[@]}
a b c d e f

$ unset foo[2]
$ echo ${foo[@]}
a b d e f
```

上面例子中，删除了数组中的第三个元素，下标为2。

删除成员也可以将这个成员设为空值。

```
$ foo=(a b c d e f)
$ foo[1]=''
$ echo ${foo[@]}
a c d e f
```

上面例子中，将数组的第二个成员设为空字符串，就删除了这个成员。

由于空值就是空字符串，所以下面这样写也可以，但是不建议这种写法。

```
$ foo[1]=
```

上面的写法也相当于删除了数组的第二个成员。

直接将数组变量赋值为空字符串，相当于删除数组的第一个成员。

```
$ foo=(a b c d e f)
$ foo=''
$ echo ${foo[@]}
b c d e f
```

上面的写法相当于删除了数组的第一个成员。

`unset ArrayName`可以清空整个数组。

```
$ unset ARRAY

$ echo ${ARRAY[*]}
<--no output-->
```

15.8 关联数组

Bash的新版本支持关联数组。关联数组使用字符串而不是整数作为数组索引。

`declare -A`可以声明关联数组。

```
declare -A colors
colors["red"]="#ff0000"
colors["green"]="#00ff00"
colors["blue"]="#0000ff"
```

整数索引的数组，可以直接使用变量名创建数组，关联数组则必须用带有`-A`选项的`declare`命令声明创建。

访问关联数组成员的方式，几乎与整数索引数组相同。

```
echo ${colors["blue"]}
```

16. set命令

`set`命令是Bash脚本的重要环节，却常常被忽视，导致脚本的安全性和可维护性出问题。
本章介绍`set`的基本用法，帮助你写出更安全的Bash脚本。

16.1 简介

我们知道，Bash执行脚本时，会创建一个子Shell。

```
$ bash script.sh
```

上面代码中，`script.sh`是在一个子Shell里面执行。这个子Shell就是脚本的执行环境，
Bash默认给定了这个环境的各种参数。

set命令用来修改子Shell环境的运行参数，即定制环境。一共有十几个参数可以定制，官方手册有完整清单，本章介绍其中最常用的几个。

顺便提一下，如果命令行下不带任何参数，直接运行set，会显示所有的环境变量和Shell函数。

```
$ set
```

16.2 set -u

执行脚本时，如果遇到不存在的变量，Bash默认忽略它。

```
#!/usr/bin/env bash

echo $a
echo bar
```

上面代码中，$a是一个不存在的变量。执行结果如下。

```
$ bash script.sh

bar
```

可以看到，echo $a输出了一个空行，Bash忽略了不存在的$a，然后继续执行echo bar。大多数情况下，这不是开发者想要的行为，遇到变量不存在，脚本应该报错，而不是一声不响地往下执行。

set -u就用来改变这种行为。脚本在头部加上它，遇到不存在的变量就会报错，并停止执行。

```
#!/usr/bin/env bash
set -u

echo $a
echo bar
```

运行结果如下。

```
$ bash script.sh
bash: script.sh: 4: a:
```

可以看到，脚本报错了，并且不再执行后面的语句。

-u还有另一种写法-o nounset，两者是等价的。

```
set -o nounset
```

16.3 set -x

默认情况下，脚本执行后，只输出运行结果，没有其他内容。如果多个命令连续执行，它们的运行结果就会连续输出。有时会分不清，某一段内容是什么命令产生的。

set -x用来在运行结果之前，先输出执行的那一行命令。

```
#!/usr/bin/env bash
set -x
```

```
echo bar
```

执行上面的脚本，结果如下。

```
$ bash script.sh
+ echo bar
bar
```

可以看到，执行echo bar之前，该命令会先打印出来，行首以+表示。这对于调试复杂的脚本是很有用的。

-x还有另一种写法-o xtrace。

```
set -o xtrace
```

脚本当中如果要关闭命令输出，可以使用set +x。

```
#!/bin/bash

number=1

set -x
if [ $number = "1" ]; then
  echo "Number equals 1"
else
  echo "Number does not equal 1"
fi
set +x
```

上面的例子中，只对特定的代码段打开命令输出。

16.4 Bash的错误处理

如果脚本里面有运行失败的命令（返回值非0），Bash默认会继续执行后面的命令。

```
#!/usr/bin/env bash

foo
echo bar
```

上面脚本中，foo是一个不存在的命令，执行时会报错。但是，Bash会忽略这个错误，继续往下执行。

```
$ bash script.sh
script.sh: 3: foo:
bar
```

可以看到，Bash只是显示有错误，并没有终止执行。

这种行为很不利于脚本安全和除错。实际开发中，如果某个命令失败，往往需要脚本停止执行，防止错误累积。这时，一般采用下面的写法。

```
command || exit 1
```

上面的写法表示只要command有非零返回值，脚本就会停止执行。

如果停止执行之前需要完成多个操作，就要采用下面三种写法。

```
#
command || { echo "command failed"; exit 1; }

#
if ! command; then echo "command failed"; exit 1; fi

#
command
if [ "$?" -ne 0 ]; then echo "command failed"; exit 1; fi
```

另外，除了停止执行，还有一种情况。如果两个命令有继承关系，只有第一个命令成功了，才能继续执行第二个命令，那么就要采用下面的写法。

```
command1 && command2
```

16.5 set -e

上面这些写法多少有些麻烦，容易疏忽。set -e从根本上解决了这个问题，它使得脚本只要发生错误，就终止执行。

```
#!/usr/bin/env bash
set -e

foo
echo bar
```

执行结果如下。

```
$ bash script.sh
script.sh: 4: foo:
```

可以看到，第4行执行失败以后，脚本就终止执行了。

set -e根据返回值来判断，一个命令是否运行失败。但是，某些命令的非零返回值可能不表示失败，或者开发者希望在命令失败的情况下，脚本继续执行下去。这时可以暂时关闭set -e，该命令执行结束后， 再重新打开set -e。

```
set +e
command1
command2
set -e
```

上面代码中，set +e表示关闭-e选项，set -e表示重新打开-e选项。

还有一种方法是使用command || true，使得该命令即使执行失败，脚本也不会终止执行。

```
#!/bin/bash
set -e
```

```
foo || true
echo bar
```

上面代码中，true使得这一行语句总是会执行成功，后面的echo bar会执行。

-e还有另一种写法-o errexit。

```
set -o errexit
```

16.6 set -o pipefail

set -e有一个例外情况，就是不适用于管道命令。

所谓管道命令，就是多个子命令通过管道运算符（|）组合成为一个大的命令。Bash会把最后一个子命令的返回值，作为整个命令的返回值。也就是说，只要最后一个子命令不失败，管道命令总是会执行成功，
因此它后面命令依然会执行，set -e就失效了。

请看下面这个例子。

```
#!/usr/bin/env bash
set -e

foo | echo a
echo bar
```

执行结果如下。

```
$ bash script.sh
a
script.sh: 4: foo:
bar
```

上面代码中，foo是一个不存在的命令，但是foo | echo a这个管道命令会执行成功，导致后面的
echo bar会继续执行。

set -o pipefail用来解决这种情况，只要一个子命令失败，整个管道命令就失败，脚本就会终止执行。

```
#!/usr/bin/env bash
set -eo pipefail

foo | echo a
echo bar
```

运行后，结果如下。

```
$ bash script.sh
a
script.sh: 4: foo:
```

可以看到，echo bar没有执行。

16.7 其他参数

set命令还有一些其他参数。

- set -n：等同于set -o noexec，不运行命令，只检查语法是否正确。
- set -f：等同于set -o noglob，表示不对通配符进行文件名扩展。
- set -v：等同于set -o verbose，表示打印Shell接收到的每一行输入。

上面的-f和-v参数，可以分别使用set +f、set +v关闭。

16.8 set 命令总结

上面重点介绍的set命令的四个参数，一般都放在一起使用。

```
#
set -euxo pipefail

#
set -eux
set -o pipefail
```

这两种写法建议放在所有Bash脚本的头部。

另一种办法是在执行Bash脚本的时候，从命令行传入这些参数。

```
$ bash -euxo pipefail script.sh
```

16.9 shopt 命令

shopt命令用来调整Shell的参数，跟set命令的作用很类似。之所以会有这两个类似命令的主要原因是，set是从 Ksh 继承的，属于 POSIX 规范的一部分，而shopt是Bash特有的。

直接输入shopt可以查看所有参数，以及它们各自打开和关闭的状态。

```
$ shopt
```

-s用来打开某个参数。

```
$ shopt -s optionNameHere
```

-u用来关闭某个参数。

```
$ shopt -u optionNameHere
```

举例来说，histappend这个参数表示退出当前Shell时，将操作历史追加到历史文件中。这个参数默认是打开的，如果使用下面的命令将其关闭，那么当前Shell的操作历史将替换掉整个历史文件。

```
$ shopt -u histappend
```

16.10 参考链接

- The Set Builtin
- Safer bash scripts with ‘set -euxo pipefail’

- Writing RobustBashShellScripts

17. 脚本除错

本章介绍如何对Shell脚本除错。

17.1 常见错误

编写Shell脚本的时候，一定要考虑到命令失败的情况，否则很容易出错。

```
#! /bin/bash

dir_name=/path/not/exist

cd $dir_name
rm *
```

上面脚本中，如果目录$dir_name不存在，cd $dir_name命令就会执行失败。这时，就不会改变当前目录，脚本会继续执行下去，导致rm *命令删光当前目录的文件。

如果改成下面的样子，也会有问题。

```
cd $dir_name && rm *
```

上面脚本中，只有cd $dir_name执行成功，才会执行rm *。但是，如果变量$dir_name为空，cd就会进入用户主目录，从而删光用户主目录的文件。

下面的写法才是正确的。

```
[[ -d $dir_name ]] && cd $dir_name && rm *
```

上面代码中，先判断目录$dir_name是否存在，然后才执行其他操作。

如果不放心删除什么文件，可以先打印出来看一下。

```
[[ -d $dir_name ]] && cd $dir_name && echo rm *
```

上面命令中，echo rm *不会删除文件，只会打印出来要删除的文件。

17.2 bash的-x参数

bash的-x参数可以在执行每一行命令之前，打印该命令。这样就不用自己输出执行的命令，一旦出错，比较容易追查。

下面是一个脚本script.sh。

```
# script.sh
echo hello world
```

加上-x参数，执行每条命令之前，都会显示该命令。

```
$ bash -x script.sh
+ echo hello world
hello world
```

上面例子中，行首为+的行，显示该行是所要执行的命令，下一行才是该命令的执行结果。

下面再看一个-x写在脚本内部的例子。

```
#! /bin/bash -x
# trouble: script to demonstrate common errors

number=1
if [ $number = 1 ]; then
  echo "Number is equal to 1."
else
  echo "Number is not equal to 1."
fi
```

上面的脚本执行之后，会输出每一行命令。

```
$ trouble
+ number=1
+ '[' 1 = 1 ']'
+ echo 'Number is equal to 1.'
Number is equal to 1.
```

输出的命令之前的+号，是由系统变量PS4决定，可以修改这个变量。

```
$ export PS4='$LINENO + '
$ trouble
5 + number=1
7 + '[' 1 = 1 ']'
8 + echo 'Number is equal to 1.'
Number is equal to 1.
```

另外，set命令也可以设置Shell的行为参数，有利于脚本除错，详见《set命令》一章。

17.3 环境变量

有一些环境变量常用于除错。

LINENO

变量LINENO返回它在脚本里面的行号。

```
#!/bin/bash

echo "This is line $LINENO"
```

执行上面的脚本test.sh，$LINENO会返回3。

```
$ ./test.sh
This is line 3
```

FUNCNAME

变量FUNCNAME返回一个数组，内容是当前的函数调用堆栈。该数组的0号成员是当前调用的函数，1号成员是调用当前函数的函数，以此类推。

```
#!/bin/bash

function func1()
{
  echo "func1: FUNCNAME0 is ${FUNCNAME[0]}"
  echo "func1: FUNCNAME1 is ${FUNCNAME[1]}"
  echo "func1: FUNCNAME2 is ${FUNCNAME[2]}"
  func2
}

function func2()
{
  echo "func2: FUNCNAME0 is ${FUNCNAME[0]}"
  echo "func2: FUNCNAME1 is ${FUNCNAME[1]}"
  echo "func2: FUNCNAME2 is ${FUNCNAME[2]}"
}

func1
```

执行上面的脚本test.sh，结果如下。

```
$ ./test.sh
func1: FUNCNAME0 is func1
func1: FUNCNAME1 is main
func1: FUNCNAME2 is
func2: FUNCNAME0 is func2
func2: FUNCNAME1 is func1
func2: FUNCNAME2 is main
```

上面例子中，执行func1时，变量FUNCNAME的0号成员是func1，1号成员是调用func1的主脚本main。执行func2时，变量FUNCNAME的0号成员是func2，1号成员是调用func2的func1。

BASH_SOURCE

变量BASH_SOURCE返回一个数组，内容是当前的脚本调用堆栈。该数组的0号成员是当前执行的脚本，1号成员是调用当前脚本的脚本，以此类推，跟变量FUNCNAME是一一对应关系。

下面有两个子脚本lib1.sh和lib2.sh。

```
# lib1.sh
function func1()
```

```
{
  echo "func1: BASH_SOURCE0 is ${BASH_SOURCE[0]}"
  echo "func1: BASH_SOURCE1 is ${BASH_SOURCE[1]}"
  echo "func1: BASH_SOURCE2 is ${BASH_SOURCE[2]}"
  func2
}

# lib2.sh
function func2()
{
  echo "func2: BASH_SOURCE0 is ${BASH_SOURCE[0]}"
  echo "func2: BASH_SOURCE1 is ${BASH_SOURCE[1]}"
  echo "func2: BASH_SOURCE2 is ${BASH_SOURCE[2]}"
}
```

然后，主脚本main.sh调用上面两个子脚本。

```
#!/bin/bash
# main.sh

source lib1.sh
source lib2.sh

func1
```

执行主脚本main.sh，会得到下面的结果。

```
$ ./main.sh
func1: BASH_SOURCE0 is lib1.sh
func1: BASH_SOURCE1 is ./main.sh
func1: BASH_SOURCE2 is
func2: BASH_SOURCE0 is lib2.sh
func2: BASH_SOURCE1 is lib1.sh
func2: BASH_SOURCE2 is ./main.sh
```

上面例子中，执行函数func1时，变量BASH_SOURCE的0号成员是func1所在的脚本lib1.sh，1号成员是主脚本main.sh；执行函数func2时，变量BASH_SOURCE的0号成员是func2所在的脚本lib2.sh，1号成员是调用func2的脚本lib1.sh。

BASH_LINENO

变量BASH_SOURCE返回一个数组，内容是每一轮调用对应的行号。${BASH_LINENO[$i]}跟${FUNCNAME[$i]}是一一对应关系，表示${FUNCNAME[$i]}在调用它的脚本文件${BASH_SOURCE[$i+1]}里面的行号。

下面有两个子脚本lib1.sh和lib2.sh。

```
# lib1.sh
function func1()
{
  echo "func1: BASH_LINENO is ${BASH_LINENO[0]}"
```

```
  echo "func1: FUNCNAME is ${FUNCNAME[0]}"
  echo "func1: BASH_SOURCE is ${BASH_SOURCE[1]}"

  func2
}

# lib2.sh
function func2()
{
  echo "func2: BASH_LINENO is ${BASH_LINENO[0]}"
  echo "func2: FUNCNAME is ${FUNCNAME[0]}"
  echo "func2: BASH_SOURCE is ${BASH_SOURCE[1]}"
}
```

然后，主脚本main.sh调用上面两个子脚本。

```
#!/bin/bash
# main.sh

source lib1.sh
source lib2.sh

func1
```

执行主脚本main.sh，会得到下面的结果。

```
$ ./main.sh
func1: BASH_LINENO is 7
func1: FUNCNAME is func1
func1: BASH_SOURCE is main.sh
func2: BASH_LINENO is 8
func2: FUNCNAME is func2
func2: BASH_SOURCE is lib1.sh
```

上面例子中，函数func1是在main.sh的第7行调用，函数func2是在lib1.sh的第8行调用的。

18. mktemp命令，trap命令

Bash脚本有时需要创建临时文件或临时目录。常见的做法是，在/tmp目录里面创建文件或目录，这样做有很多弊端，使用mktemp命令是最安全的做法。

18.1 临时文件的安全问题

直接创建临时文件，尤其在/tmp目录里面，往往会导致安全问题。

首先，/tmp目录是所有人可读写的，任何用户都可以往该目录里面写文件。创建的临时文件也是所有人可读的。

```
$ touch /tmp/info.txt
$ ls -l /tmp/info.txt
```

```
-rw-r--r-- 1 ruanyf ruanyf 0 12  28 17:12 /tmp/info.txt
```

上面命令在/tmp目录直接创建文件，该文件默认是所有人可读的。

其次，如果攻击者知道临时文件的文件名，他可以创建符号链接，链接到临时文件，可能导致系统运行异常。攻击者也可能向脚本提供一些恶意数据。因此，临时文件最好使用不可预测、每次都不一样的文件名，防止被利用。

最后，临时文件使用完毕，应该删除。但是，脚本意外退出时，往往会忽略清理临时文件。

生成临时文件应该遵循下面的规则。

- 创建前检查文件是否已经存在。
- 确保临时文件已成功创建。
- 临时文件必须有权限的限制。
- 临时文件要使用不可预测的文件名。
- 脚本退出时，要删除临时文件（使用trap命令）。

18.2 mktemp 命令的用法

mktemp命令就是为安全创建临时文件而设计的。虽然在创建临时文件之前，它不会检查临时文件是否存在，但是它支持唯一文件名和清除机制，因此可以减轻安全攻击的风险。

直接运行mktemp命令，就能生成一个临时文件。

```
$ mktemp
/tmp/tmp.4GcsWSG4vj

$ ls -l /tmp/tmp.4GcsWSG4vj
-rw------- 1 ruanyf ruanyf 0 12  28 12:49 /tmp/tmp.4GcsWSG4vj
```

上面命令中，mktemp命令生成的临时文件名是随机的，而且权限是只有用户本人可读写。

Bash脚本使用mktemp命令的用法如下。

```
#!/bin/bash

TMPFILE=$(mktemp)
echo "Our temp file is $TMPFILE"
```

为了确保临时文件创建成功，mktemp命令后面最好使用 OR 运算符（||），保证创建失败时退出脚本。

```
#!/bin/bash

TMPFILE=$(mktemp) || exit 1
echo "Our temp file is $TMPFILE"
```

为了保证脚本退出时临时文件被删除，可以使用trap命令指定退出时的清除操作。

```
#!/bin/bash

trap 'rm -f "$TMPFILE"' EXIT
```

```
TMPFILE=$(mktemp) || exit 1
echo "Our temp file is $TMPFILE"
```

18.3 mktemp 命令的参数

-d参数可以创建一个临时目录。

```
$ mktemp -d
/tmp/tmp.Wcau5UjmN6
```

-p参数可以指定临时文件所在的目录。默认是使用$TMPDIR环境变量指定的目录，如果这个变量没设置，那么使用/tmp目录。

```
$ mktemp -p /home/ruanyf/
/home/ruanyf/tmp.FOKEtvs2H3
```

-t参数可以指定临时文件的文件名模板，模板的末尾必须至少包含三个连续的X字符，表示随机字符，建议至少使用六个X。默认的文件名模板是tmp.后接十个随机字符。

```
$ mktemp -t mytemp.XXXXXXX
/tmp/mytemp.yZ1HgZV
```

18.4 trap 命令

trap命令用来在Bash脚本中响应系统信号。

最常见的系统信号就是 SIGINT（中断），即按 Ctrl + C 所产生的信号。trap命令的-l参数，可以列出所有的系统信号。

```
$ trap -l
 1) SIGHUP   2) SIGINT   3) SIGQUIT  4) SIGILL   5) SIGTRAP
 6) SIGABRT  7) SIGBUS   8) SIGFPE   9) SIGKILL 10) SIGUSR1
11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP
21) SIGTTIN 22) SIGTTOU 23) SIGURG  24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF 28) SIGWINCH    29) SIGIO   30) SIGPWR
31) SIGSYS  34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

trap的命令格式如下。

```
$ trap [ ] [ 1] [ 2] ...
```

上面代码中，“动作”是一个Bash命令，“信号”常用的有以下几个。

- HUP：编号1，脚本与所在的终端脱离联系。

- INT：编号2，用户按下 Ctrl + C，意图让脚本中止运行。
- QUIT：编号3，用户按下 Ctrl + 斜杠，意图退出脚本。
- KILL：编号9，该信号用于杀死进程。
- TERM：编号15，这是kill命令发出的默认信号。
- EXIT：编号0，这不是系统信号，而是Bash脚本特有的信号，不管什么情况，只要退出脚本就会产生。

trap命令响应EXIT信号的写法如下。

```
$ trap 'rm -f "$TMPFILE"' EXIT
```

上面命令中，脚本遇到EXIT信号时，就会执行rm -f "$TMPFILE"。

trap 命令的常见使用场景，就是在Bash脚本中指定退出时执行的清理命令。

```
#!/bin/bash

trap 'rm -f "$TMPFILE"' EXIT

TMPFILE=$(mktemp) || exit 1
ls /etc > $TMPFILE
if grep -qi "kernel" $TMPFILE; then
  echo 'find'
fi
```

上面代码中，不管是脚本正常执行结束，还是用户按 Ctrl + C 终止，都会产生EXIT信号，从而触发删除临时文件。

注意，trap命令必须放在脚本的开头。否则，它上方的任何命令导致脚本退出，都不会被它捕获。

如果trap需要触发多条命令，可以封装一个Bash函数。

```
function egress {
  command1
  command2
  command3
}

trap egress EXIT
```

18.5 参考链接

- Working with Temporary Files and Directories inShellScripts, Steven Vona
- Using Trap to ExitBashScripts Cleanly
- Sending and Trapping Signals

19. Bash启动环境

19.1 Session

用户每次使用Shell，都会开启一个与Shell的 Session（对话）。

Session 有两种类型：登录 Session 和非登录 Session，也可以叫做 login shell 和 non-login shell。

登录 Session

登录 Session 是用户登录系统以后，系统为用户开启的原始 Session，通常需要用户输入用户名和密码进行登录。

登录 Session 一般进行整个系统环境的初始化，启动的初始化脚本依次如下。

- /etc/profile：所有用户的全局配置脚本。
- /etc/profile.d目录里面所有.sh文件
- ~/.bash_profile：用户的个人配置脚本。如果该脚本存在，则执行完就不再往下执行。
- ~/.bash_login：如果~/.bash_profile没找到，则尝试执行这个脚本（C shell的初始化脚本）。如果该脚本存在，则执行完就不再往下执行。
- ~/.profile：如果~/.bash_profile和~/.bash_login都没找到，则尝试读取这个脚本（Bourne shell和Korn shell 的初始化脚本）。

Linux 发行版更新的时候，会更新/etc里面的文件，比如/etc/profile，因此不要直接修改这个文件。如果想修改所有用户的登陆环境，就在/etc/profile.d目录里面新建.sh脚本。

如果想修改你个人的登录环境，一般是写在~/.bash_profile里面。下面是一个典型的.bash_profile文件。

```
# .bash_profile
PATH=/sbin:/usr/sbin:/bin:/usr/bin:/usr/local/bin
PATH=$PATH:$HOME/bin

SHELL=/bin/bash
MANPATH=/usr/man:/usr/X11/man
EDITOR=/usr/bin/vi
PS1='\h:\w\$ '
PS2='> '

if [ -f ~/.bashrc ]; then
. ~/.bashrc
fi

export PATH
export EDITOR
```

可以看到，这个脚本定义了一些最基本的环境变量，然后执行了~/.bashrc。

bash命令的--login参数，会强制执行登录 Session 会执行的脚本。

```
$ bash --login
```

bash命令的--noprofile参数，会跳过上面这些 Profile 脚本。

```
$ bash --noprofile
```

非登录 Session

非登录 Session 是用户进入系统以后，手动新建的 Session，这时不会进行环境初始化。

比如，在命令行执行bash命令，就会新建一个非登录 Session。

非登录 Session 的初始化脚本依次如下。

- /etc/bash.bashrc：对全体用户有效。
- ~/.bashrc：仅对当前用户有效。

对用户来说，~/.bashrc通常是最重要的脚本。非登录 Session 默认会执行它，而登陆 Session 一般也会通过调用执行它。由于每次执行Bash脚本，都会新建一个非登录 Session，所以~/.bashrc也是每次执行脚本都会执行的。

bash命令的--norc参数，可以禁止在非登录 Session 执行~/.bashrc脚本。

```
$ bash --norc
```

bash命令的--rcfile参数，指定另一个脚本代替.bashrc。

```
$ bash --rcfile testrc
```

.bash_logout

~/.bash_logout脚本在每次退出 Session 时执行，通常用来做一些清理工作和记录工作，比如删除临时文件，记录用户在本次 Session 花费的时间。

如果没有退出时要执行的命令，这个文件也可以不存在。

19.2 启动选项

为了方便 Debug，有时在启动Bash的时候，可以加上启动参数。

- -n：不运行脚本，只检查是否有语法错误。
- -v：输出每一行语句运行结果前，会先输出该行语句。
- -x：每一个命令处理完以后，先输出该命令，再进行下一个命令的处理。

```
$ bash -n scriptname
$ bash -v scriptname
$ bash -x scriptname
```

19.3 键盘绑定

Bash允许用户定义自己的快捷键。全局的键盘绑定文件默认为/etc/inputrc，你可以在主目录创建自己的键盘绑定文件.inputrc文件。如果定义了这个文件，需要在其中加入下面这行，保证全局绑定不会被遗漏。

```
$include /etc/inputrc
```

.inputrc文件里面的快捷键，可以像这样定义，"\C-t":"pwd\n"表示将Ctrl + t绑定为运行pwd命令。

19.4 source 命令

source命令用于执行一个脚本，通常用于重新加载一个配置文件。

```
$ source .bashrc
```

source命令最大的特点是在当前Shell执行脚本，不像直接执行脚本时，会新建一个子Shell。
所以，source命令执行脚本时，不需要export变量。

```
#!/bin/bash
# test.sh
echo $foo
```

上面脚本输出$foo变量的值。

```
#  Shell    foo
$ foo=1

#    1
$ source test.sh
1

#
$ bash test.sh
```

上面例子中，当前Shell的变量foo并没有export，所以直接执行无法读取，但是source执行可以读取。

source命令的另一个用途，是在脚本内部加载外部库。

```
#!/bin/bash

source ./lib.sh

function_from_lib
```

上面脚本在内部使用source命令加载了一个外部库，然后就可以在脚本里面，使用这个外部库定义的函数。

source有一个简写形式，可以使用一个点（.）来表示。

```
$ . .bashrc
```

20. 命令提示符

用户进入Bash以后，Bash会显示一个命令提示符，用来提示用户在该位置后面输入命令。

20.1 环境变量 PS1

命令提示符通常是美元符号$，对于根用户则是井号#。这个符号是环境变量PS1决定的，执行下面的命令，可以看到当前命令提示符的定义。

```
$ echo $PS1
```

Bash允许用户自定义命令提示符，只要改写这个变量即可。改写后的PS1，可以放在用户的Bash配置文件.bashrc里面，以后新建Bash对话时，新的提示符就会生效。要在当前窗口看到修改后的提示符，可以执行下面的命令。

```
$ source ~/.bashrc
```

命令提示符的定义，可以包含特殊的转义字符，表示特定内容。

- \a：响铃，计算机发出一记声音。
- \d：以星期、月、日格式表示当前日期，例如“Mon May 26”。
- \h：本机的主机名。
- \H：完整的主机名。
- \j：运行在当前Shell会话的工作数。
- \l：当前终端设备名。
- \n：一个换行符。
- \r：一个回车符。
- \s：Shell的名称。
- \t：24小时制的hours:minutes:seconds格式表示当前时间。
- \T：12小时制的当前时间。
- \@：12小时制的AM/PM格式表示当前时间。
- \A：24小时制的hours:minutes表示当前时间。
- \u：当前用户名。
- \v：Shell的版本号。
- \V：Shell的版本号和发布号。
- \w：当前的工作路径。
- \W：当前目录名。
- \!：当前命令在命令历史中的编号。
- \#：当前 shell 会话中的命令数。
- \$：普通用户显示为$字符，根用户显示为#字符。
- \[：非打印字符序列的开始标志。
- \]：非打印字符序列的结束标志。

举例来说，[\u@\h \W]\$这个提示符定义，显示出来就是[user@host ~]$（具体的显示内容取决于你的系统）。

```
[user@host ~]$ echo $PS1
[\u@\h \W]\$
```

改写PS1变量，就可以改变这个命令提示符。

```
$ PS1="\A \h \$ "
17:33 host $
```

注意，$后面最好跟一个空格，这样的话，用户的输入与提示符就不会连在一起。

20.2 颜色

默认情况下，命令提示符是显示终端预定义的颜色。Bash允许自定义提示符颜色。

使用下面的代码，可以设定其后文本的颜色。

- \033[0;30m：黑色
- \033[1;30m：深灰色

- \033[0;31m：红色
- \033[1;31m：浅红色
- \033[0;32m：绿色
- \033[1;32m：浅绿色
- \033[0;33m：棕色
- \033[1;33m：黄色
- \033[0;34m：蓝色
- \033[1;34m：浅蓝色
- \033[0;35m：粉红
- \033[1;35m：浅粉色
- \033[0;36m：青色
- \033[1;36m：浅青色
- \033[0;37m：浅灰色
- \033[1;37m：白色

举例来说，如果要将提示符设为红色，可以将PS1设成下面的代码。

```
PS1='\[\033[0;31m\]<\u@\h \W>\$'
```

但是，上面这样设置以后，用户在提示符后面输入的文本也是红色的。为了解决这个问题，可以在结尾添加另一个特殊代码\[\033[00m\]，表示将其后的文本恢复到默认颜色。

```
PS1='\[\033[0;31m\]<\u@\h \W>\$\[\033[00m\]'
```

除了设置前景颜色，Bash还允许设置背景颜色。

- \033[0;40m：蓝色
- \033[1;44m：黑色
- \033[0;41m：红色
- \033[1;45m：粉红
- \033[0;42m：绿色
- \033[1;46m：青色
- \033[0;43m：棕色
- \033[1;47m：浅灰色

下面是一个带有红色背景的提示符。

```
PS1='\[\033[0;41m\]<\u@\h \W>\$\[\033[0m\] '
```

20.3 环境变量PS2，PS3，PS4

除了PS1，Bash还提供了提示符相关的另外三个环境变量。

环境变量PS2是命令行折行输入时系统的提示符，默认为>。

```
$ echo "hello
> world"
```

上面命令中，输入hello以后按下回车键，系统会提示继续输入。这时，第二行显示的提示符就是PS2定义的>。

环境变量PS3是使用select命令时，系统输入菜单的提示符。

环境变量PS4默认为+。它是使用Bash的-x参数执行脚本时，每一行命令在执行前都会先打印出来，并且在行首出现的那个提示符。

比如下面是脚本test.sh。

```
#!/bin/bash

echo "hello world"
```

使用-x参数执行这个脚本。

```
$ bash -x test.sh
+ echo 'hello world'
hello world
```

上面例子中，输出的第一行前面有一个+，这就是变量PS4定义的。