Bash shell 学习笔记

1. 引言

Shell 同时是一个命令解释器和一门编程语言。作为命令解释器（command interpreter），shell 为用户提供 UNIX/Linux 系统内核(kernel)的接口，使其能使用其多种工具集，而其编程语言功能使得这些工具能够连接起来。

UNIX/Linux shell 有很多种，比如 Bourne shell(又称 AT&T shell,标准的 UNIX shell, sh)、C shell(又称 Berkeley shell)、Korn shell(Bourne shell 的扩展集,ksh)、TC shell、Z shell、Bash shell 等。 这些 shell 的功能和使用都不尽相同。我们通常说 shell 编程，其实说精通所有 shell 编程是很难的，一般也没有必要。学习 shell 编程最好选定一种 shell 来学习。而如此众多的 shell 选那一种比较好呢？笔者的意见是选择 Bash，这不仅因为 Bash 几乎可以说是所有 shell 中功能最丰富的，也符合 IEEE 1003.2 POSIX/ISO 9945.2 Shell 和工具部分规范，同时 Bash 也是 Linux 上的默认 shell。

由于 shell 不仅仅是一门编程语言，学习 shell 的过程中除了要掌握其如何定义变量、使用条件、循环等功能外，还应特别注意掌握如何通过各种方式配置其环境。

本文是 Bash shell 的总结学习笔记，总结过程中尽量注意写明自己的心得、编程陷阱以及经过很长时间才发觉的好用的功能等。

2. Bash 简介

Bash 是“ Bourne-Again SHell”的缩写，可以看成是 Bourne shell 的升级版。它几乎完全向后与 sh 兼容，融进了 Korn shell 和 C shell 的有用功能，同时也有自己的功能特征。Bash 也是旨在符合 IEEE POSIX 规范(IEEE Standard 1003.1)中 IEEE POSIX Shell and Tools 部分的一种 shell，是 GNU 系统的标准 shell。Bash 诞生于 1988 年 1 月 10 星期天，最初由 Brian Fox 开发，目前由 Chet Ramey 维护。

Bash 相对 sh 的特征主要有：

- 目录处理，包含 pushd, popd 和 dirs 命令
- 作业控制，包括 fg 和 bg 命令，以及使用 CRTL-Z 停止作业
- 大括号扩展，可以产生任意的字符串
- ~扩展，指代用户根目录的缩写
- 别名，让你为命令或命令行定义缩写名称
- 命令历史，让你记得以前输入的指令

以上源于 C shell。

- 命令行编辑，使用 emacs 或 vi 风格
- 键盘绑定，让你设置自定义编辑的键序列
- 集成编程特性，包含几个 UNIX 外部命令的功能，包括 test、expr、getopt、echo 等，使得编程任务能更简洁有效地完成
- 控制结构，特别是 select 结构，能简单生成菜单
- 新的选项与变量，使得你有更多的途径来定制你的环境
- 一维数组，使得引用与操作数据列表更为简单
- 动态加载 built-in 命令，自定义命令并加载进 shell 的功能。

3. 启动 Bash

3.1Bash 启动选项

Bash 可以带以下选项启动：

Bash 选项	说 明
--login (-l)	使 shell 表现得像通过登录启动的
--init-file filename --rcfile filename	执行 filename 中（代替~/.bashrc）命令启动交互式 Bash
--noediting	不读入 GNU 命令行编辑库
--noprofile	Bash 作为登录 shell 启动时不执行系统配置文件/etc/profile 或者个人初始化文件~/.bash_profile,或~/.bash_login 或者~/.profile
--norc	在交互性 shell 中不读入~/.bashrc 文件。这一选项在 shell 是以 sh 启动时默认是开的
--restricted (-r)	以受限的 shell 启动

3.2 交互式启动

交互式包括两种方式：作为 login shell 或者 non-login shell。

交互式 login 启动时读取以下系统级别和用户级别的启动文件；交互式 non-login 启动时读取以下 Bash 级别启动文件。

因此，Bash 提供了三种级别的环境配置，分别与三种初始化文件对应：

（1） 系统级别，由/etc/profile 文件控制。这个文件在 Bash shell 启动时被执行。它可被系统所有 sh 和 ksh 用户使用。

（2） 用户级别，由~/.bash_profile（或者~/.bash_login，或者~/.profile）和~/.bash_logout 文件控制。这些文件控制登录用户的基本登录和退出环境。

（3） Bash 级别（子 shell 级别），由~/.bashrc 文件控制。每次一个新的 Bash shell 启动时将自动执行~/.bashrc 文件，用来配置只属于 Bash shell 的环境。~/.bashrc 一般会含有以下语句：

```
if [ -f /etc/bashrc ]; then
   . /etc/bashrc
 fi
```

/etc/bashrc 一般用来设置所有 Bash shell 公用的变量

说明：

- **系统配置文件/etc/profile 文件中一般有如下语句：**

```
for i in /etc/profile.d/*.sh ; do
        if [ -r "$i" ]; then
        . $i
        fi
 done
```

所以系统会在初始化时运行/etc/profile.d/目录下的所有可读.sh 脚本。

- **用户级别的三个登录配置文件(~/.bash_profile，~/.bash_login 和~/.profile)中，系统**

只会 source 一个(依次查找，一个不存在，查找下一个，找到就 source 它)。~/.profile 是为从 sh 和 ksh 的配置文件.profile 而来，~/.bash_login 是从 csh 的配置文件.login 而来。这种机制的一个应用是如果你经常使用 Bourne shell，你可以将其配置写在.profile 文件中，如果你又想增加 Bash 特有的环境控制命令，你可以将其在写入.bash_profile 文件中，然后在.bash_profile 中加入 source .profile 语句。

- 用户登录时默认不会 source ~/.bashrc 文件，如果要用需要自行在脚本中加入。这也是一般在.bash_profile 文件中有以下语句的原因：

```
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi
```

- .bash_profile 和/etc/profile 中定义的变量和 alias 定义默认不会继承到 bash 中。需要这两个文件定义的变量可以为子 shell 使用时可以使用 export 语句将其转化为环境变量。alias 定义尽量放在.bashrc 或/etc/bashrc 文件中。
- 所有配置文件的执行顺序如下

```
|-  /etc/profile → /etc/profile.d/*.sh
|-  ~/.bash_profile (or ~/.bash_login, or ~/.profile)
                |→~/.bashrc
                        |→/etc/bashrc
|- ~/.bash_logout (退出时)
```

但应注意，系统只自动读取/etc/profile, ~/.bash_profile 和~/.bash_logout，其余是自定义的。

3.3 非交互式启动

当 Bash 通过运行 shell 脚本的方式启动时就是非交互式的。非交互式启动时，它将查看环境变量 BASH_ENV，扩展其值并运行它，就像运行了以下命令：

```
if [ -n "$BASH_ENV" ] ; then    . "$BASH_ENV" ; fi
```

但是 PATH 变量不用来查找文件名。

说明：

- 判断 shell 是否是交互式启动可通过以下方式：

(1) 检查'-'特殊参数:

```
case "$-" in
    *i*) echo This shell is interactive ;;
     *) echo This shell is not interactive    ;;
esac
```

(2) 检查 PS1，非交互式 shell 它不会被设置：

```
if [ -z "$PS1" ]; then
        echo This shell is not interactive
else
        echo This shell is interactive
fi
```

- 当 shell 交互式启动时，有以下不同于非交互式启动的行为：

(1) 读入初始化文件；

(2) 作业控制默认启动；

（3） 显示 PS1,PS2；
（4） 命令行编辑功能；
（5） 命令历史功能
（6） 别名扩展
（7） 定期检查邮件，根据 MAIL, MAILPATH, MAILCHECK 配置
（8） 扩展错误、重定向错误、exec 错误、解析命令错误等不会引起 Bash 退出
（9） 检查 TMOUT 的值，如果固定秒在打印$PS1 后没有命令读入退出

4. 注释与帮助

Shell 中的注释是跟在#号后面的行,注释可以单独成为一行，也可以跟在命令后面，与命令共处一行；脚本的第一行的#号是个例外（如#!/bin/bash），#！称为幻数，内核根据它确定哪个程序来解释脚本的行。

如果是针对 Bash 编程，一定要在首行加上#!/bin/bash，以防止脚本被错误的 shell 执行。

Shell 常见的帮助命令是 man 和 info。man 工具可以显示系统手册页中的内容，这些内容大多数都是对命令的解释信息。通过查看系统文档中的 man 页可以得到程序的更多相关主题信息和 Linux 的更多特性。与 man 相比，info 工具可显示更完整的最新的 GNU 工具信息。应明白最新的帮助在 info 而不是 man，因为很多 man 已经停止维护，而 info 则维护良好。另外，可以使用 command –h (--help)命令对 command 的使用方式进行查询。**Bash 独特之处是它还提供了一个 help 命令，其提供的帮助信息简单明了。**

5. 变量与数组

5.1 变量分类

Bash 的变量按作用域可分为两类：局部变量和环境变量（全局变量）。局部变量只在创建它们的 shell 中可用。而环境变量在创建他们的 shell 及其派生子进程中使用。

Bash 的变量又可以按照是否是用户创建的分为：用户定义变量和专有 shell 变量。

说明：局部变量可以通过简单赋值或一个变量名来设置，或者用 declare 内置函数来设置(不含-x 选项)。内置函数 local 可以用来创建局部变量，但仅限于在函数内使用。函数不使用 local 的变量在整个 shell 中是可见的。环境变量可以用 declare –x 命令或者 export varname=value 命令来设置。

5.2 变量的命名与赋值

变量的命名规则：变量名必须以字母或下划线字符开头，其余的字符可以是字母、数字或下划线字符。任何其他字符标志着变量名的终止。变量名大小敏感。

变量赋值：给变量赋值时，等号周围不能有任何空白符。为了给变量赋空值，可以在等号后跟一个换行符。格式: name=[value]

注意：

- **初学者很容易犯在等号左右加空格的错误，必须注意！这是 shell 极特别的地方，**

因为几乎所有的编程语言都允许甚至建议=左右加空格。

- **变量定义时前面不加$符号，而在引用时需要加。Bash 的这一特点与 AWK 和 Perl 都有差别：AWK 无论定义和引用都不加$符号，而 Perl 的标量无论定义和引用都需要加$符。**
- **$name 是更一般形式的${name}的简单形式。两者在大多数时候是可以互换的，但是在可能引起歧义时需要加{}表示对某一变量的引用。如果变量名跟随一个不是字母、数字或下划线的字符，则可以省略括号。**
- **给变量赋值的另一方式是变量替换。Bash 的标准方式是$(UNIX command)，同时 bash 也兼容 Bourne shell 的`Unix command`方式。但是建议使用 Bash 的标准方式，因为这种方式更灵活，而且不容易出错（``方式很容易与单引号混淆）。**

5.3 declare 选项

Bash 提供两个内置命令创建变量：declare 和 typeset。typeset 是为了与 Korn shell 兼容，建议使用 declare。declare 选项如下：

选项	含 义
-a	将变量当作一个数组。即分配元素
-f	列出函数的名称和定义
-F	只列出函数名
-i	将变量设为整型
-r	将变量设为只读
-x	将变量输出到子 shell 中，相当于定义后用 export 将其变为环境变量

5.4 位置和特殊参量

首先说说参量与变量的区别：参量(parameter)是储存值的实体，它可以是一个名字、一个数或者特殊的字符；变量(variable)是以名字命名的参量。变量有一个值和 0 或多个属性，属性通过以上 declare 选项来设置。

位置和特殊参量	指 代 对 象
$0	脚本名
$#	位置参数个数
$*	所有的位置参数
$@	未加引号时与$*的含义相同
$1…${N}	单独的位置变量
$$	当前 shell 的 PID
$-	当前的 sh 选项设置
$?	已执行的上一条命令的退出值
$!	最后一个进入后台的作业 PID
$_	上一条命令的最后一个参数

说明：以上参量除了$*和$@的区别外都容易理解。以下重点说明这两者的区别：

- **$*是一个由所有位置参量组成的单一的字符串，由环境变量 IFS 第一个字符分隔；相当于"$1c$2c…", c 是 IFS 的第一个字符；$@是 N 个独立的双引字符串，由空格**

分隔，即相当于”$1” “$2” … “$N”。

- **举例说明：**

```
set a b c d
echo “$*”          # a b c d
echo “$@”          #a b c d
IFS=’,’
echo “$*”          # a,b,c,d
echo “$@”          #a b c d

function countargs
{
      echo “$# args”
}
countargs “$*”     #1 args
countargs “$@”     #3 args
```

5.5 环境变量

环境变量是已经用 export 内置命令导出的变量，可在创建它们的 shell 及其子 shell 中使用。环境变量可能在/bin/login 程序，.bash_profile 文件中定义。Bash 内置的环境变量如下：

变量名	含　义	说 明
CDPATH	cd 命令的搜索路径，以冒号分隔的目录列表	sh 变量
HOME	主目录。未指定目录时，cd 命令指向该目录	sh 变量
IFS	字段分隔符，一般是空格符、制表符和换行符，用于由命令替换，循环结构中的表和读取的输入产生的词的字段划分	sh 变量
MAIL	如果该参数被设置为某邮件文件的名称，而 MAILPATH 未被设置，当邮件到达 MAIL 指定的文件时，shell 会通知用户	sh 变量
MAILPATH	由冒号分隔的文件名列表。如果设置了这个参数，只要有邮件到达任何一个由它指定的文件，shell 都会通知用户。可以用?分隔指定邮件到来时显示的消息（默认为 You have mail）	sh 变量
OPTARG	上一个有 getopts 内置命令处理的选项参数的值	sh 变量
OPTIND	下一个有 getopts 内置命令处理的参数的序号	sh 变量
PATH	命令搜索路径。一个由冒号分隔的目录列表，shell 用它来搜索命令。**空目录名指代当前目录。**	sh 变量
PS1	主提示字符串。默认值是’\s-\v\$’	sh 变量
PS2	次提示符，默认值是’>’	sh 变量
BASH	调用 bash 实例是使用的全路径名	
BASH_ARGC	当前 bash 执行 call 栈中每一帧的参数个数数组	
BASH_ARGV	当前执行 call 栈中所有参数的数组	
BASH_COMMAN	当前或即将执行的 bash 命令	

D		
BASH_ENV	每一个新的 bash shell(包括脚本)启动时执行的环境文件。	
BASH_EXECUTION_STRING	bash –c 选项启动的命令参数	
BASH_LINENO	与 FUNCNAME 的每一成员对应的源文件的行数变量数组，${BASH_LINENO[$i]}是调用${FUNCNAME[$i]}时对应的行数，对应文件名${BASH_SOURCE[$i]}，使用 LINENO 得到当前行数	
BASH_REMATCH	[[条件命令通过=~赋值的变量数组	
BASH_SOURCE	FUNCNAME 数组变量对应源文件名数组变量	
BASH_SUBSHELL	每次一个子 shell 或子 shell 环境产生时加 1。初始值为 0	
BASH_VERSINFO	当前 Bash 的版本信息数组 BASH_VERSINFO[0] 主版本号 BASH_VERSINFO[1] 小版本号 BASH_VERSINFO[2] 补丁级别 BASH_VERSINFO[3] build 版本 BASH_VERSINFO[4] release 状态 BASH_VERSINFO[5] MACHTYPE 值	
BASH_VERSION	**当前 Bash 的版本信息**	
COLUMNS	设置该变量就给 shell 编辑模式和选择的命令定义了编辑窗口的宽度	
COMP_CWORD COMP_LINE COMP_POINT COMP_WORDBREAKS COMP_WORDS COMPREPLY	可编程完成相关变量	
EMACS	如果 Bash 在环境中发现这一变量当 shell 以值 ’t’ 启动时，它假定 shell 是运行在 emacs shell 缓冲中，禁用行编辑	
EUID	只读当前用户的 ID	
FCEDIT	fc 命令-e 选项使用的默认编辑器	
FIGNORE	冒号(:)分隔的进行文件名补全时忽略的后缀列表,比如 ’.o:~’	
FUNCNAME	在当前执行调用栈中所有的 shell 函数名数组	
GLOBIGNORE	冒号分隔变量,定义在文件名扩展(globbing)时被忽略的文件列表	
GROUPS	当前用户所属的组	
histchars	最多三个字符用以控制命令历史扩展、快速替换和 tokenization。第一个字符（通常为!）是历史扩展符；第二个字符（通常^）为快速替换符；可选的第三个字符（通	

	常为#）表示当一行第一个字符是这个字符时为注释	
HISTCMD	当前命令的命令历史号	
HISTCONTROL	冒号分隔的值，控制命令怎样保存在命令历史列表中。如果设置 ignorespace 值,以一个空格符开头的行将不会进入历史清单 ;如果设置了 ignoredups,与前一个历史行匹配的行不会写入。ignoreboth 是前两者的缩写	
HISTFILE	指定保存命令行历史的文件名。默认值为~/.bash_history。如果 unset，交互式 shell 退出时将不保存命令行历史	
HISTFILESIZE	历史文件能包含的最大行数。默认值 500	
HISTIGNORE	冒号分隔的模式，决定怎样的命令保存在命令历史中	
HISTSIZE	记录在命令行历史中的命令数。默认为 500 **注意 HISTSIZE 与 HISTFILESIZE 的不同：HISTSIZE 定义一个 session 中保存的命令数，HISTFILESIZE 定义 HISTFILE 文件保存的行数，是跨 session 的**	
HISTTIMEFORMAT	如果此变量被设置并且为非空值，它将用来格式化 strftime 的值，在 history 输出命令时附加时间戳。此变量有时比较有用	
HOSTFILE	包含一个格式和/etc/hosts 一样的文件的名称，当 shell 需要补全一个主机名时将读取该文件。文件可以交互式更改。下一次试图补全主机名时，bash 将新文件的内容添加到已经存在的数据库中	
HOSTNAME	当前的主机名	
HOSTTYPE	自动设置正在运行的机器类型。默认值由系统决定	
IGNOREEOF	控制 shell 接收到单独一个 EOF 字符作为输入的行为	
INPUTRC	Readline 启动文件的文件名，默认为~/.inputrc	
LANG	用来为没有以 LC_开头的变量明确选取的种类确定 locale 类	
LC_ALL	忽略 LANG 和任何其他 LC_变量的值	
LC_COLLATE	确定对路径名扩展的结果进行排序是的整理顺序,以及匹配文件名与模式时的范围表达式，等价类和整理序列的行为	
LC_CTYPE	确定字符解释和文件名扩展和模式匹配时字符类的行为	
LC_MESSAGES	确定用于转换前面有一个$的双引号串的 locale	
LC_NUMERIC	确定进行数值格式化的 locale	
LINENO	每次 shell 在一个脚本或函数中替换代表当前连续行号	
LINES	内置命令 select 使用来去定所选列表的列长度。在接收 SIGWINCH 时自动设置	
MACHTYPE	包含一个描述正在运行 bash 的系统的串	
MAILCHECK	定义 shell 将隔多长时间（以 s 为单位）检查一次由参数 MAILPATH 或 MAILFILE 指定的文件，看看是否有邮件到达。默认值 600。如果它设为 0,shell 每次输出主提示符之前都会去检查邮件	
OLDPWD	前一个工作目录	
OPTERR	如果设置为 1，显示来自 getopts 内置命令的错误信息	

OSTYPE	描述正在运行 bash 的操作系统 （比如 linux gnu）	
PIPESTATUS	一个数组,包含一列最近在管道执行的前台作业的进程退出状态值	
POSIXLY_CORRECT	如果这一变量在 bash 启动时设置，shell 进入 POSIX 模式	
PPID	父进程的进程 ID	
PROMPT_COMMAND	赋给这个变量的命令将在主提示符显示前执行	
PS3	与 select 命令一起使用的选择提示符串，默认值为#?	
PS4	当开启追踪时使用的调试字符串，默认值为+。追踪可以用 set －x 开启	
PWD	当前工作目录，为 cd 设置	
RANDOM	每次引用该变量，就产生一个随即整数（0~32767）。随机数序列 seeding 可以通过给 RANDOM 赋值来初始化	
REPLY	给 read 内置命令提供的默认参数	
SECONDS	调用 shell 以来的秒数	
SHELL	保存 shell 的全路径名，如果未设置，赋值当前用户登录 shell 的全路径名	
SHELLOPTS	冒号分隔的 shell 选项，比如：braceexpand:emacs:hashall:histexpand:history:interactive-comments:monitor	
SHLVL	每次一个新 Bash Instance 启动时加 1，决定 Bash 嵌套的深度	
TIMEFORMAT	时间格式设定参数	
TMOUT	设置退出前等待输入的秒数	
TMPDIR	如果设置，Bash 用来作为保存临时文件的目录	
UID	当前用户的用户 ID,在 shell 启动时初始化	

5.6 数组

Bash 支持一维数组，指标默认从 0 开始。数组赋值的方式有：

```
names=([2]=alice [0]=hatter [1]=duchess)
names=(hatter duchess alice)
names=(hatter [5]=duchess alice)    #names[0]=hatter names[5]=duchess names[6]=alice
```

数组元素的引用方式: ${array[i]}

所有元素使用(@和*)：

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

使用 ${!array[@]} 来获取有值的数组指标。使用 ${#array[i]}来获取某一数组元素的长度，${#array[@]}或${#array[*]}是数组长度（非 null 元素个数）

如果对数组重新赋值，原数组元素将会消失。可以使用 unset 命令删除整个数组或数组的某一个元素： unset array 或者 unset array[i]

说明：Bash 的数组与 C 的区别：C 中如果定义了 array[1000],则 array[0]~array[999]都一定是存在的，而 Bash 则不然。

5.7 算术运算

Bash 的算术运算可以通过多种方式实现：

（1） 直接赋值：使用 declare –i 命令后可以直接对变量赋值或计算。但是这种方式很容易出错，应注意空格问题，引号问题。如

declare –i num

num=hello #error!

num=5 + 5 #error！不能有空格

num=”4 * 6” #带引号，可以有空格

num=6.789 #num=6 去尾

（2） 使用 let 内置命令 i=5

let i=i+1

let “i = i +2” #带空格需要加引号

let “i+=1”

let 支持的操作符如下：

操作符	含 义
-	负号
！	逻辑非
~	按位取反
*	乘法
/	除法
%	求模
+	加法
-	减法
<<	左移位
>>	右移位
<= >= < > == !=	关系运算符
&	按位与
^|	异或
&&	逻辑与
||	逻辑或
！	逻辑非
=	赋值
*= /= %= += -= <<= >>= &= ^= |=	赋值简写符
++ --	**自增 自减**
**	**乘方**
expr? expr: expr	条件表达式
expr1, expr2	逗号表达式

（3） ((expression)) 方式。等价于 let “expression”，但是更推荐使用，如果想返回表达式的值，可使用$((expression)) 方式

（4） $[expression]方式

（5） expr expression 方式，以上 expr 是内置命令，但应注意乘法的使用方式：expr 3 * 2,

说明：

- **以上（3）（4）两种方式中括号前后可以有或没有空格；里面的变量不需要带$符。**
- **可以使用[base#]n 来使用 2~64 进制数。**
- **Bash 只支持整数运算，要进行更复杂运算需要借用 bc、awk 等工具。比如：**

```
n=$(echo “scale=3; 13 / 2” | bc)
echo $n
```

5.8 字符串操作符

${}形式的变量引用允许字符串操作符，进行确保变量存在、为变量设置默认值、捕获未定义变量的错误以及删除匹配某一模式的变量值部分等。

操作符	含 义
${variable:-word}	如果变量 variable 已被设置且非空，则代入它的值，否则代入 word
${variable:=word}	设置一个变量的默认值如果其未定义
${variable:+word}	如果变量已被设置且值非空，代入 word,否则返回 null(临时替换变量值)
${variable:?message}	如果变量已被设置且非空，返回其值；否则打印变量名，输出 message,并从 shell 中退出
${variable:offset:length}	获取变量值中从 offset 开始的长度为 length 的子串（第一个字符为 0,如果未设定 length,则获取 offset 之后的所有字符串）
${variable#pattern}	将变量值的头部与模式进行最小匹配，并将匹配到的部分删除
${variable##pattern}	将变量值的头部与模式进行最大匹配，并将匹配到的部分删除
${variable%pattern}	将变量值的尾部与模式进行最小匹配，并将匹配到的部分删除
${variable%%pattern}	将变量值的尾部与模式进行最大匹配，并将匹配到的部分删除
${#variable}	替换为变量的字符个数。如果是*或@，长度则是位置参量的个数
$**{*variable/pattern/string*} **${*variable//pattern/string*}	变量中最长匹配到 pattern 被替换为 string,第一种形式只替换第一个匹配；第二种形式替换所有匹配。如果 pattern 以#开头，必须匹配变量开头；如果以%开头，必须匹配变量结尾。
${!prefix*} ${!prefix@}	扩展到以 prefix 开始的变量名，以 IFS 头一个字符分隔
${!name[@]} ${!name[*]}	如果 name 是数组变量，扩展至数组 key
*（patternlist）	0 或多个指定模式匹配
+（patternlist）	1 或多个指定模式匹配
？（patternlist）	0 或 1 个指定模式匹配
@（patternlist）	刚好 1 个指定模式匹配
!（patternlist）	非指定模式匹配

（举例）从完整文件名获取文件目录与文件名的方法：

```
filepath=”/home/test/a.txt”
dirname=${filepath%/*}          # 得到目录名/home/test
filename=${filepath##*/}        # 得到文件名 a.txt
```

6. 条件结构

6.1 if 语句

Bash if 语句的格式如下：

```
if test-commands ; then
    consequence-command;
[elif more-test-commands; then
    more-consequents; ]
[else alternate-consequents; ]
fi
```

test-commands 执行后，如果其返回值是 0，执行 consequence-command 语句。

说明：

- **注意在 Bash 中 0 相当于 C 中的 true(1)，而非 0 为 false(0)。这一点与 C 语言相反，值得特别注意。Bash 如此定义 True/False 的原因是可以用多种方式定义错误原因，而成功只有一种（0）。**
- 语句 if 中 test-commands 可以有多种形式，主要有：

（1） 命令或命令的并与或

```
if command ;then
if command1 && command2; then
if command1 || command2 ; then
```

（2） test 表达式 （等价于 [ ] **由于[是一个 builtin，所以与后面的 expr 应有空格**）

以下是 test 表达式的使用说明（参考 info test 命令）

操作符	测试内容
字符串测试	
STRING1 = STRING2	True 如果两字符串相等
STRING1 == STRING2	与上同，但不合 POSIX 标准
STRING1 != STRING2	True 如果两字符串不相等
STRING1 < STRING2	STRING1 小于 STRING2
STRING1 > STRING2	STRING1 大于 STRING2
[-n STRING [STRING	True 如果字符串长度不为 0
[-z STRING	True 如果字符串长度为 0
连接测试(逻辑测试)	
[! EXPR	True 如果 EXPR 是 false
[EXPR1 –a EXPR2	如果 EXPR1 和 EXPR2 都为 true 是 True
[EXPR1 –o EXPR2	如果 EXPR1 或 EXPR2 都为 true 是 True
数值测试	
ARG1 –eq ARG2	相等测试
ARG1 –ne ARG2	不等
ARG1 –gt ARG2	大于

ARG1 –ge ARG2	大于等于
ARG1 –lt ARG2	小于
ARG1 –le ARG2	小于等于
文件特性测试	
-e FILE	True 如果文件存在
-s FILE	如果文件存在且大小大于 0 True
FILE1 –nt FILE2	如果文件 FILE1 比 FILE2 新(修改时间)，或者 FILE1 存在 FILE2 不存在，返回 True
FILE1 –ot FILE2	如果文件 FILE1 比 FILE2 旧，或者 FILE2 存在 FILE1 不存在，返回 True
FILE1 –ef FILE2	如果文件 FILE1 和 FILE2 有相同的设备数和 inode 则返回 True 比如他们是 hard links to each other
文件类型测试	
-b FILE	块专用文件
-c FILE	字符专用文件
-d FILE	FILE 存在并且是一个目录
-f FILE	FILE 存在并且是一个普通文件
-h FILE -L FILE	FILE 存在并且是一个符号链接
-p FILE	FILE 存在并且是一个命名管道
-S FILE	FILE 存在并且是一个 socket
-t FD	如果 FS 是一个文件描述符并且被一个终端打开
文件访问测试	
-g FILE	FILE 存在并且 Set – group- ID 被设置
-k FILE	FILE Sticky 位被设置
-r FILE	FILE 存在并且可读
-u FILE	FILE 存在并且 Set-user-ID 位被设置
-w FILE	FILE 存在并且可写
-x FILE	FILE 存在并且可执行
-O FILE	FILE 存在且属于有效用户 ID
-G FILE	FILE 存在且属于有效组 ID

（3） 复合 test 测试新格式[[ ]]

复合命令操作符[[ ]]与普通 test 命令的区别有：

- 中间操作符与变量以及[[之间不需要有空格
- 允许在字符串表达式中进行 shell 元字符扩展
- 变量不需要像老 test 命令中那样用引号括起来，即使它包括多个词的时候也是如此
- 双等号可用来代替单个等号
- ！ &&、||使用

（4） 使用 let 命令测试数字(( )) (新格式)

比如 x =2 ; y=3;

```
if  ((  x>2 ))
if  ((  x < 2  ))
if  (( x == 2 && y ==3 ))
```

if ((x > 2 || y < 3))

注意以上变量前都不需要加$号！

6.2 case 语句

case 命令是一个多路分支命令，可用来代替 if/elif 命令。

case 命令的格式如下：

```
case variable in
  value1)
        commands    ;;
  value2)
        commands    ;;
  *)
        Commands ;;
 esac
```

比如：

```
echo –n “Enter the name of an animal : “
read ANIMAL
echo –n “The $ANIMAL has “
case $ANIMAL in
    horse | dog |cat) echo –n “four” ;;
    man | kangaroo ) echo –n “two” ;;
    *) echo –n “an unknown number of “;
esac
echo “ legs.”
```

注意事项：

- **case 分支是以双分号（;;）结束，这一点很特别**
- ***）表示默认执行值**

6.2 select 语句

select 语句主要用于创建菜单。按数字顺序排列的菜单项将列表显示在标准输出上，并显示 PS3 提示符请求用户输入（PS3 默认值为’#?’）。显示 PS3 提示符后，shell 等待用户输入，输入应当是菜单列表中的一个数字。输入值保存在 shell 特殊变量 REPLY 中。

select 命令的格式是：

```
select name [ in words … ]; do commands ; done
```

比如：

```
select fname in * ;
do
echo you picked $fname \($REPLY\)
break;
done
```

说明：

- 如果 in words 省略，相当于 in “$@”
- select 与 case 经常可以配合使用
- select 是一个循环命令，如果需要退出，使用 break 命令或者使用 exit 退出脚本

6. 循环结构

Bash 提供了以下三种类型的循环命令：

7.1 for 命令

格式为：

for name [in words …]; do commands; done

如果 in words 不存在，相当于 in $@。

另一格式的循环语句格式为：

for ((expr1; expr2; expr3)) ; do commands; done

这种形式与 C 语言中使用的方式已经很相似了。

举例 1：

```
IFS=:

for dir in $PATH
do
    echo $dir
done
```

举例 2：

```
for (( i=0; i<10; i++))
do
  echo $i
done
```

7.2 while 命令

格式为：

while test-commands; do consequent-commands; done

举例 1：

```
num=0
while (( $num < 10 ))
do
        echo –n “$num”
        let num+=1
done
```

举例 2：

```
path=$PATH:
while [ $path ]; do
    ls -ld ${path%%:*}
    path=${path#*:}
done
```

7.3 until 命令

格式：

until test-commands; do consequent-commands; done

举例【3】：

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

说明：

- **由于 until test-commands; do consequent-commands; done 完全等价于**
  while ! test-commands; do consequent-commands; done
  我们也可以不使用 until 语句
- **Bash 的 until 语句与 C 语言中的 do until 循环是不同的：C 语言的 do until 总是执行 do 之后的语句直到 until 之后的条件满足；而 Bash 中 until 语句只有 until 之后的条件不满足时才执行。**

7.4 循环控制命令

Bash 提供以下循环控制命令：

shift [n]: 将参数列表左移指定次数

break [n]: 强行退出循环，但不退出程序；break 命令用数字作为参数，指定跳出哪层循环

continue [n]: 跳回循环的顶部

7. 函数

Bash 函数定义的格式如下：

[function] name () compound-command [redirections]

关于 Bash 函数的说明如下：

- **保留字 function 是可选的，可以使用 name() { }; 或者 function name { }**
- **由于历史原因，在函数定义中{}前后必须有空格，以为{和}在 Bash 中是作为保留字看待的，这一点与 C 语言有着很大区别，值得注意。初学者很容易在这一点上犯错误。即以 function f{command;}的方式定义函数会报错。**
- **Bash shell 按以下顺序来判断究竟是别名、函数、内置命令还是磁盘上的可执行程序：先在别名中找，然后是函数、内置命令，最后才是可执行程序**

- **函数必须先定义，后使用（也就是函数必须定义在脚本开始位置或者用 source 引入后使用）**
- **函数在当前环境中运行。函数共享调用它的脚本中的变量，可以使用 local 命令创建局部变量**
- **函数定义可以用 unset –f 命令删除，或 export –f 命令导出到子脚本中**
- **函数中 return 语句，返回函数执行的最后一条命令的退出状态，或者返回指定的参数值**
- **如果在函数中使用 exit 命令，将会退出整个脚本**
- **函数允许递归，且递归的次数没有限制**
- **函数参数。可以使用位置参数向函数传递参数。位置参数是函数私有的，也就是说，函数对参数的操作不会影响在函数外使用的任何位置参量。**
- **函数可以与内置命令同名，要在函数中使用内置命令，可以使用 builtin command 方式命令 command 会取消别名和函数查找。**

8. 输入输出

9.1 标准 I/O

Shell I/O 基本概念：

- I/O 重定向通常与 FD(File Descriptor)有关，shell 的 FD 通常有 10 个，即 0~9
- 常见的 FD 有 3 个，为 0（stdin）、1（stdout）、2(stdout)，默认与键盘、显示器、显示器相连。
- [n]<word 用来重定向输入。word 代表的文件将打开代替标准输入，如果 n 未指定，默认为 0
- [n]>[|]word 用来重定向输出。word 代表的文件将打开写 FD n。如果 n 未指定，默认为 1。如果文件不存在将被创建，如果文件已经存在将被清除为 0 长度。word 前面加|用来忽略 noclobber 设置（如果不加，当 noclobber 为 on 时若文件存在将提示 cannot overwrite existing file。
- [n]>>word 用来重定向和追加输出
- [n]>&-用来关闭 FD n 的输出，n 不指定默认为 1
- [n]<&-用来关闭 FD n 的输入，n 不指定默认为 0
- 管道“|”(pipe line):上一个命令的 stdout 接到下一个命令的 stdin
- tee 命令是在不影响原本 I/O 的情况下，将 stdout 复制一份到文件
- （）将 command group 置于 sub-shell 去执行，也称 nested sub-shell，它有一点非常重要的特性是：继承父 shell 的标准输入输出和错误和所有打开的 FD。
- exec 命令：常用来替代当前 shell 并重新启动一个 shell，换句话说，并没有启动子 shell。使用这一命令时任何现有环境都将会被清除。exec 在对文件描述符进行操作的时候，也只有在这时，exec 不会覆盖你当前的 shell 环境。
- 常见的 FD 有：

  /dev/fd/n
  /dev/stdin (0)
  /dev/stdout (1)
  /dev/stderr (2)

```
/dev/tcp/host/port  (TCP socket)
/dev/udp/host/port  (UDP socket)
```

9.2 重定向

重定向操作符	功　能
&> filename	重定向标准输出和标准错误输出
>& filename	重定向标准输出和标准错误输出(首选方式) 等价于 >word 2>&1
2>&1	将标准错误输出重定向到输出的去处
1>&2	将输出重定向到错误输出的去处
[n]<>filename	如果是一个设备文件，使用文件作为标准输入和标准输出

9.3 Here 文档

Here 文档是一种特殊方式的引用。Here 文档为需要输入数据的程序接收内置文本，直到收到用户自定义的终止符。格式为

```
<<[-] word
   Here-document
delimiter
```

若加-,则 delimiter 前面的 TAB 将被忽略。

9.4 字符串 I/O(echo、printf 和 read 命令)

9.4.1 echo 命令

命令 echo 将它的参数显示在标准输出上，echo 的选项和转义序列如下：

选　项	含　义
-e	允许解释转义序列
-E	禁止解释转义字符
-n	删除输出结果中行尾的换行符
转义序列	
\a	Bell
\b	退格
\c	不带换行符打印一行
\f	换页
\n	换行
\r	回车
\t	制表符
\v	纵向制表符
\\	反斜杠
\nnn	ASCII 码是 nnn（八进制）的字符

9.4.2 printf 命令

命令 printf 是 GNU 版本可以用来编排打印输出的格式。它以和 C printf 函数相同的方式打印格式串。但是与 C 语言的使用方式不同：采用 printf “fmt” arg1 … 而不是 printf(“fmt”, arg1,arg2)。

9.4.3 read 命令

内置命令 read 用于从终端或文件读取输入，如果未指定变量，读入的行将赋给变量 REPLY。read 命令的使用格式如下：

格 式	含 义
read answer	从标准输入读取一行赋值给变量 answer
read first last	从标准输入读取一行赋值给变量 first last
read	从标准输入读取一行赋值给变量 REPLY
read –a arrayname	从标准输入读取一行赋值给数组 arrayname
read -e	在交互式 shell 中启动编辑器
read –p prompt	打印提示符，等待输入，并将输入赋值给 REPLY 变量
read –r line	允许输入包含反斜杠

10 作业控制

格 式	含 义
%n	号为 n 的作业
%% %+ %	当前作业
%-	前一作业
%string	以 string 开始的作业
%?string	含 string 的作业
bg [jobspec]	后台恢复作业
fg [jobspec]	恢复作业到前台，使之称为当前作业
jobs [-lnprs] [jobspec]	显示作业状态 -r 列出所有运行的作业 –s 列出所有挂起的作业
kill [-s sigspec] [-n signum] [-sigspec] jobspec or pid	Kill 作业
wait [jobspec or pid]	等待知道进程退出，返回退出状态
disown [-ar] [-h] [jobspec]	将作业从活动作业列表中移除

11 命令历史

Bash 提供两个内置命令来操作历史列表和历史文件：fc 和 history
fc 用来进行命令编辑:
fc [-e ename] [-nlr] first last
fc –s [pat=rep] command
history 命令的使用如下：

格 式	含 义
history [n]	显示（最后 n 条作业）
history -c	清除命令历史
history -d offset	清除位置在 offset 处的作业
history [-anrw] filename	附加、读、写历史文件
history –ps arg	进行历史替换
事件指示器	
!	说明开始历史替换
!!	重新执行上一条命令
!N	重新执行历史清单中的第 N 条命令
!-N	重新执行从当前命令往回数的第 N 条命令
!string	重新执行最后一条
!?string?%	重新执行历史清单中最近一条包含串 string 的命令行参数
!$	用上一条命令的最后一个参数作为当前命令行
!!string	将 string 添加到上一条命令的最后并执行
!N string	将 string 添加到历史清单中的第 N 条命令的最后并执行
!N:s/old/new/	将前面的第 N 条命令中，将第一次出现的 old 串替换成 new 串
!N:gs/old/new/	将前面的第 N 条命令中，将所有出现的 old 串替换成 new 串
^old^new^	在上一条命令中，用 new 串替换 old 串
Command !N:wn	在当前命令后添加一个来自前第 N 条命令的参数(wn)并执行它。wn 是一个从 0,1,..开始的数

12 杂项

12.1 set 命令和选项

命令 set 用来打开或关闭 shell 选项，打开选项在前加一个减号(-)，关闭一个选项，在选项前加加号(+)。**用 set –o 命令来显**示所有 set 选项的状态。set 选项如下：

快捷开关	选项名	含 义
-a	allexport	从这个选项被设置开始就自动表明要输出的新变量或修改过的变量，直到选项关闭
-b	notify	通知用户什么时候后台作业完成
-e	errexit	当一个命令返回一个非零退出状态时，退出。读取

		初始化文件时不设置
-f	noglob	禁止文件名扩展
-h	hashall	当查找执行时定位和存储(hash)命令。默认开
-k	keyword	所有的 keyword 参数,而不只是那些命令名前的 keyword 参数,被放在环境命令中
-m	monitor	允许作业控制
-n	noexec	读取命令但是不执行，这个用来检查脚本的语法错误
-o [option-name]		不加 option-name 用来显示所有 set 参数设置，加用来设置某一项
-p	privileged	打开特权模式(在此模式,$ENV 文件被处理,不能从环境中继承 shell 函数.如果是有效用户 ID 而不是实用户组则自动启动.关闭此选项将使得有效用户和组 ID 设置实用户和组 ID)
-t	onecmd	在读取和执行一条命令后退出
-u	nounset	如果一个变量没有被设置就显示一个错误
-v	verbose	当读取时打印 shell 输入行
-x	xtrace	打印命令的跟踪（调试）
-B	braceexpand	打开花括号扩展，默认设置
-C	noclobber	防止文件在重定向时被重写
-E	errtrace	设置后，所有 ERR trap 将被子 shell 中的函数、命令替换和命令继承
-H	history	打开命令行历史，默认开
-P	physical	如果设置,当执行命令 cd 时,不使用符号链接,而是使用实际物理目录
-T	functrace	如果设置,所有 DEBUG 和 RETURN 的 trap 在子 shell 环境中将被函数、命令替换和命令继承
	vi	使用 vi 内置编辑器进行命令编辑
	emacs	使用 emacs 编辑器进行命令编辑，此项设置,则 vi 模式自动关闭
--		如果此后不跟选项，位置参量被 unset,否则，位置参量被设为后面的值
-		标志选项的结束，所有剩下的参数都被赋值给位置参量

12.2 shopt 命令和选项

Bash 2.x 以上的 shopt 命令也可以用来打开或关闭 shell 选项。
shopt –s option-name 用来设置选项
shopt –u option-name 用来关闭选项
shopt [-p] 显示所有选项
shopt [-q] Suppress normal output (quiet mode)
shopt –o 限制选项名为 set 选项
shopt 命令选项如下：

选 项	含 义
cdable_vars	如果给 cd 内置命令的参数不是一个目录，就假定它是一个变量名，变量的值是将要转到的目录
cdspell	纠正 cd 命令中目录名的较小拼写错误
checkhash	在试图执行一个命令前，先在 hash 中查找，以确定命令是否存在
checkwinsize	在每个命令后检查窗口大小，必要时更新 LINES 和 COLUMNS 值
cmdhist	设置后试图将一个多行命令的所有行保存在同一个历史项中。这使得多行命令的重新编辑很方便
dotglob	在文件名扩展中包括以点(.)开头的文件名
execfail	如果一个非交互式 shell 不能执行指定给 exec 内置命令作为参数的文件，它不会退出。如果 exec 失败，一个交互式 shell 不会退出
expand_aliases	别名被扩展，默认开
extdebug	如果设置，调试功能打开
extglob	打开扩展的模式匹配特性
extquote	如果设置，$’string’和$”string”引用在${parameter}扩展中使用。默认开
failglob	如果设置，文件名匹配错误将报错
force_fignore	如果 set, FIGNORE 变量定义的后缀在词补全时被忽略，默认开
gnu_errfmt	如果 set, 以 GNU 错误信息格式写错误信息
histappend	Shell 退出时历史清单将添加到以 HISTFILE 变量的值命名的文件中，而不是覆盖文件
histeedit	如果 readline 正被使用，用户有机会重新编辑一个失败的历史替换
histverify	如果设置,且 readline 正被使用,历史替换的结果不会立即传递给 shell 解释器。而是将结果行装入 readline 编辑缓冲区，允许进一步修改
hostcomplete	如果设置，且 readline 正被使用,当正在完成一个包含@的词时 Bash 将试图执行主机名补全
huponexit	如果设置，当一个交互式登录 shell 退出时，bash 将发送一个 SIGHUP 给所有的作业
interactive_comments	在一个交互式 shell 中，允许以#开头的词以及同一行中其他的字符被忽略，默认打开
lithist	如果打开，且 cmdlist 选项也打开，多行命令将用嵌入的换行符保存到历史中，而无需在可能的地方用分号分隔
mailwarn	如果设置，且 bash 用来检查邮件的文件自从上次检查后已经被访问，

	将显示信息”The mail in mailfile has been read”
nocaseglob	执行文件名扩展时，Bash 不区分大小写的方式匹配文件名
nullglob	允许没有匹配任何文件的文件名模式扩展成一个空串，而不是它本身
promptvars	提示串在扩展后再经历变量和参量扩展，默认开
restricted_shell	受限模式启动 shell
shift_verbose	如果设置，移动计数超过位置参量个数时，shift 内置命令将打印一个错误信息
sourcepath	如果设置，source 内置命令使用 PATH 的值来寻找包含作为参数提供的文件的目录，默认开
xpg_echo	如果设置，echo 命令扩展转义序列(\)

12.3 exec 命令

命令 exec 能够在不启动新进程的前提下，将当前正在运行的程序替换为一个新的程序。使用 exec 命令，不需创建子 shell 就能改变标准输入和输出。

12.4 trap 命令

命令 trap 的使用方式如下：

```
trap cmd sig1 sig2 ...
```

当其捕获到信号 sig1,sig2…后执行命令 cmd。

所有信号及其编号如下（用 kill -l 或 trap -l 命令显示）：

```
 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     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
```

13 命令行处理过程

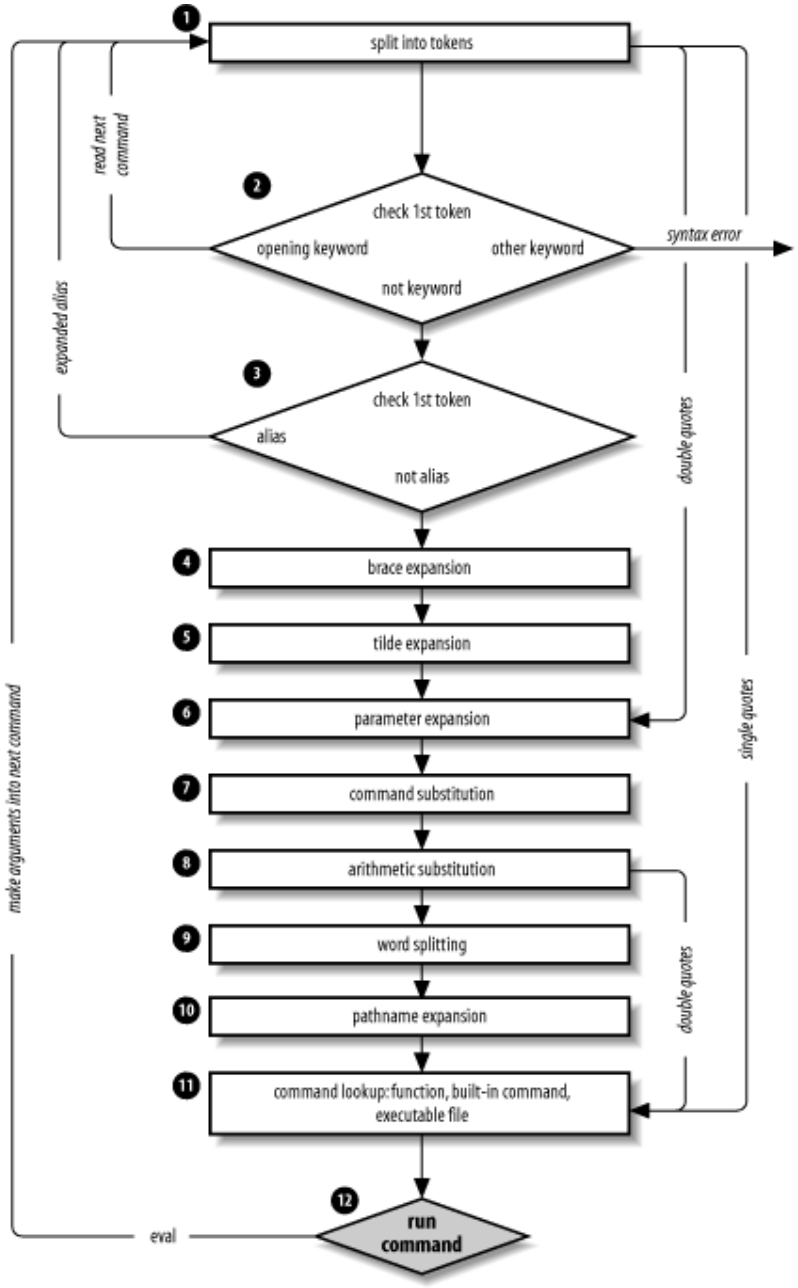
Bash shell 的命令行处理过程如下：

- 1.以 SPACE、TAB、NEWLINE、;、 (、)、 <、 >,、|、和 &等固定的元字符分割命令成符号，符号的类型包括词、关键字、I/O 重定向符和分号。
- 2.检查第一个符号，看它是否一个没有引号或反斜线的关键字。如果其是一个开启关键字，比如 if 或其它控制结构开启关键字，function、{或(，则命令是一个复合命令。Shell 为复合命令进行内部设置后，读入下一个命令，重新开始这一过程。如果关键字不是复合命令开启命令（比如控制结构中间符号像 then、else 或 do，结束符号像 fi 或 done 或逻辑操作符）,shell 将报错。
- 3.别名替换。检查第一个字是否命令别名。如果是，以别名定义替换并返回到第 1 步；否则到第 4 步。
- 4. 进行花括号扩展。比如 a{b,c}变成 ab ac
- 5. 波形符(~)扩展。以$HOME 代替~，以用户的主目录代替~user
- 6. 参数和变量扩展。替换任何以$号开始的表达式
- 7. 命令替换。对任何$(command)或`command`的格式进行命令替换
- 8. 算术扩展。 对 $((expression))中的算术表达式进行计算
- 9. 字切割(Word Splitting)。Shell 将$IFS 参量的每一个字符作为一个分隔符，对其它扩展的结果进行字切割。如果 IFS 未设置，默认为<space><tab><newline>。
- 10. 文件名扩展。对*?和[]对进行文件名扩展（将其视为模式）
- 10.5. **去除引号。在以上扩展之后，所有未用引号引起的字符\，’,和”号将被去除。（这一点需注意，因为它决定对命令执行时的整个语句的理解）**
- 11. 以第一个字按照函数、内置命令或$PATH 中的一个文件的顺序作为命令
- 12. 在设置好 I/O 重定向或者另外事情后运行命令。

另外，说明引号使用与 eval 命令：

- 单引号(‘)绕过步骤 1-10。所有在单引号中的字符都原封不动，包括别名；也不能在单引号之间加单引号，即使加\（’abc’\”def’可起到类似作用)。
- 双引号绕过步骤 1-5，以及 9、10，也就是说，它只进行参量替换、命令替换和算术表达式替换。在双引号之间的单引号会保持不变。
- 命令 eval 用来对后面的参数进行命令行解析并执行，不跳过任何过程。比如$listpage=”ls | more”，直接使用$listpage 会报错，而使用 eval $listpage 则 OK。

命令行处理过程图示如下：



. (命令行处理过程，引自文献【3】)

13 Shell 脚本调试

13.1 调试选项

应知道以下常见调试选项：

Set 选项	命令行选项	作用
noexec	-n	不运行命令，只检查语法错误
verbose	-v	运行命令前显示它
xtrace	-x	在命令行处理后显示命令

13.2 编程风格

以下编程风格可以帮助减少程序错误：

- 注释标明程序用途，在注释上偷工减料会为今后调试埋下隐患
- 定义有意义的变量名，可把它们集中放在程序头部，以利于发现拼写错误和空值
- 条件和循环命令中使用缩进
- 使用 echo 命令或 verbose 开关跟踪程序执行
- 熟悉运算符的使用，它们因 shell 而异，而且经常出错误
- 确保程序的健壮性
- 使用简短的测试内容

13.3 常见错误类型

- 未定义的变量和误写变量
- 变量前后不能有任何空格
- 未完成的编程语句
- 运算符错误
- 操作符错误
- 引用错误（注意单引、双引和反引用的使用）
- 在[或[[之后要有一个空格
- 在 fun()的定义中花括号前后必须有空格

14 对参考文献的评论

文献【1】是目前 Bash 主要维护者写的参考手册，内容全面而权威，但是却不适合作为学习用书，有基础后全面看一遍会很有收获；文献【2】是 Bash 的很好的简介，其中有很好的 bash、ksh 与 sh 的比较材料；推荐使用文献【3】作为学习教材，因为其循序渐进而且通俗易懂；文献【4】是个大部头书，对目前主流的 shell 都有介绍，对于普通读者，会觉得其重复的内容较多，另外，这本书中对 grep、sed 和 awk 也有较详细的介绍，可以作为很好的中文参考书。

15 参考文献

【1】 Chet Ramey etc. Bash Reference Manual.
【2】 Chet Ramey. Bash- The GNU shell
【3】 Cameron Newham. Learning the bash shell, 3rd Edition. O'Reilly. March 2005.
【4】 Ellie Quigley 著.李化等译. UNIX Shells 范例精解(第 4 版). 清华大学出版社. 2007.5
【5】 Linux shell I/O 重定向详解.
http://www.diybl.com/course/3_program/shell/shelljs/200862/119822.html

廖海仁(liaohairen@gmail.com)
2009．4 初稿，2010.12 Revised