shell扩展

主要内容

- 查看命令的类型和位置： type/which/whereis/help
- bash环境文件
- 花括号扩展
- 命令替换
- 单词分割
- shell命令结构

内置命令和外部命令

- shell内置命令: 在当前Shell环境下执行，影响当前shell环境，比如cd/type/exit等
- 外部命令: 可以独立运行的可执行程序，启动子进程执行，继承父进程（这里为Shell）的环境。子进程执行过程中环境的改变与父进程无关
- 为了避免多次搜索PATH，以前使用的外部命令会保存在一个hash表中，可通过内置命令hash查看
- **内置命令type**可以查看命令的类型以及（别名情况下）真正的命令、（外部命令时）所在的位置、（函数时）包含的shell代码
  - 缺省为显示第一个匹配的命令
  - 选项-a 可列出所有可能匹配的命令,包括别名、函数、内置命令以及外部命令
  - 选项-P 只查找外部命令，等价于which
- **外部命令which**可以显示一个外部命令的位置，不支持内部命令
- **外部命令whereis命令**与which类似，还会查找与命令相关的源代码和文档所在的位置

windows系统下有一个类似的where命令，而且可以根据模式搜索

```
demo@mars:~$ type id who ls sudo su visudo
adduser which type cd man dequote
id is /usr/bin/id
who is /usr/bin/who
ls is aliased to `ls --color=auto'
sudo is /usr/bin/sudo
su is /bin/su
visudo is /usr/sbin/visudo
adduser is /usr/sbin/adduser
which is /usr/bin/which
type is a shell builtin
cd is a shell builtin
man is hashed (/usr/bin/man)
dequote is a function
dequote ()
{
    eval printf %s "$1" 2> /dev/null
}
demo@mars:~$ type -a ls
ls is aliased to `ls --color=auto'
ls is /bin/ls
dlmao@mars:/etc$ type -a pwd
pwd is a shell builtin
pwd is /bin/pwd
```

```
demo@mars:~$ which id who ls sudo su visudo
adduser which type cd man dequote
/usr/bin/id
/usr/bin/who
/bin/ls
/usr/bin/sudo
/bin/su
/usr/sbin/visudo
/usr/sbin/adduser
/usr/bin/which
/usr/bin/man
demo@mars:~$ which type  # 命令执行失败
demo@mars:~$ echo $?
1
demo@mars:~$ whereis id
id: /usr/bin/id /usr/share/man/man1/id.1.gz
```

查看内置命令帮助:help

- **help内置命令**可查看shell内置命令(包括在shell编程时使用的关键字for、if、while等)相应的帮助信息。 -s 选项仅查看语法

```
demo@mars:~$ help
GNU bash, version 4.3.42(1)-release (x86_64-pc-linux-gnu)
These shell commands are defined internally.  Type `help' to see this list.
Type `help name' to find out more about the function `name'.
Use `info bash' to find out more about the shell in general.
Use `man -k' or `info' to find out more about commands not in this list.
...
dlmao@mars:~$ help -s cd pwd
cd: cd [-L|[-P [-e]] [-@]] [dir]
pwd: pwd [-LP]
dlmao@mars:~$ help for
for: for NAME [in WORDS ... ] ; do COMMANDS; done
    Execute commands for each member in a list.

    The `for' loop executes a sequence of commands for each member in a
    list of items.  If `in WORDS ...;' is not present, then `in "$@"' is
    assumed.  For each element in WORDS, NAME is set to that element, and
    the COMMANDS are executed.

    Exit Status:
    Returns the status of the last command executed.
```

主要内容

- 查看命令的类型和位置： type/which/whereis/help
- bash环境文件
- 花括号扩展
- 命令替换
- 单词分割
- shell命令结构

Shell环境

- 登录(login)Shell：通过控制台或ssh远程登录进入，需要验证用户身份
- 非登录Shell: GUI开启新的终端或执行bash命令进入，无需验证用户身份
- shell环境的配置文件：
  - 可以分为系统配置文件以及用户配置文件
  - 登录shell的配置文件包括用于登录和注销的文件以及环境文件
  - 非登录shell的配置文件为环境文件

交互式	登录文件	环境文件	注销文件
系统文件	/etc/profile	/etc/bash.bashrc	
用户文件	.bash_profile .bash_login .profile	.bashrc	.bash_logout
登录shell	√ 先执行/etc/profile，然后寻找HOME目录中的3个用户文件，寻找第一个可读的文件执行	√ (在登录文件中调用)	√
非登录shell		√	

环境配置文件bashrc

- 系统bash环境文件/etc/bash.bashrc
- 用户bash环境文件~/.bashrc
- 环境文件也是一个shell脚本，其中给出了顺序执行的一系列命令，也支持条件、循环结构等
- .bashrc一般完成如下工作:
  - 设置命令历史相关SHELL变量，开启相关shell选项
  - 设置shell提示符PS1
  - 设置环境变量（如PATH/CDPATH等）
  - 设置命令别名
  - 显示相关信息等

bash命令： bash [options] [command_string | file]

- -l, --login 以login shell方式执行
- -i，强制以交互式shell方式执行
- -c 以非交互式方式执行某个命令，该命令来自于后面的第一个参数
- -x 开启xtrace选项，执行具体命令前输出要执行的命令和参数

如果后面的参数没有，表示启动一个新的**交互式**的shell

如果包含选项-c，则参数部分为以**非交互式方式**执行的命令，比如 bash -c date

否则后面的参数为以**非交互式方式**执行的bash脚本文件，比如 bash script

- 以非交互式方式执行时，不会执行bash的环境文件
- 特殊参数$-保存了执行bash所使用的选项，如果其中包含了i，则为交互式方式

✓ 通配符扩展在当前shell下已经展开了，而不是在新的环境中再展开

```
dlmao@mars:~$ echo $-
himBHs
dlmao@mars:~$ ls -dF [a-e]*
bin/  Desktop/  dlmao/  Documents/  Downloads/  examples.desktop
dlmao@mars:~$ LC_COLLATE=C ls -dF [a-e]*
Desktop/  Documents/  Downloads/  bin/  dlmao/  examples.desktop
dlmao@mars:~$ LC_COLLATE=C bash -c 'ls -dF [a-e]*'
bin/  dlmao/  examples.desktop
```

bash脚本:Shebang行

- bash [options] script 通过bash解释执行文件script里面的命令
- 如果某个文件script有**可执行权限，且为文本文件**，可与其他二进制外部命令一样执行
  - 允许script的拥有者执行：chmod u+x script
  - 缺省采用用户当前的shell(echo $SHELL)执行该脚本文件
  - 如果脚本的第一行为shebang(hashbang)行，使用指定的解释器执行该脚本

    #! interpreter arg
    - 前两个字符为#!，表示shebang行
    - interpreter给出了解释器所在的位置，而后面的arg可选，给出了执行时使用的参数
    - script 1 2，等价于调用 interpreter arg script 1 2

#!/bin/bash 解释器为/bin目录下的bash程序

#!/usr/bin/env bash 解释器为PATH路径中找到的名为bash的程序

env [NAME=VALUE]... [COMMAND [ARGS]...]

设置相应的变量，用于执行COMMAND [ARGS]

bash脚本

#! /bin/bash

或者

#! /usr/bin/env bash

.py文件包含如下行：

#! /usr/bin/env python3

bash脚本:执行

- bash [options] script
- 改变权限(chmod u+x script)允许执行后 通过./script或script（要求当前目录在PATH中）运行该脚本
- 上述两种方式都会**启动一个新的子进程**，然后执行相应的脚本，子进程会继承父shell的环境变量，但是脚本里面对于环境变量的改变不会影响父shell的环境变量
- 在当前shell环境下执行脚本script： **source script**或**. script**
  - 如果在script里面给变量赋值，或调用内置命令cd改变了工作目录等，则在执行完该script之后，这些改变仍然有效
  - 环境脚本(.bashrc)等应该通过这种方式执行
  - 如果参数没有目录部分，搜索PATH中的目录列表找到相应的文件script，如果没有找到，则在当前目录中查找文件script
- 用户登录时， shell环境配置文件（包括.bashrc)都是使用source命令执行的

主要内容

- 查看命令的类型和位置： type/which/whereis/help
- bash环境文件
- 花括号扩展
- 命令替换
- 单词分割
- shell命令结构

bash：解析命令

普通命令：[NAME=VALUE]... [COMMAND [ARGS]...]

- 首先基于分割单词的元字符 | < > & ; ( ) space tab newline将命令分解成单词<word>
- 如果为**交互式shell**，首先进行
  - 历史事件扩展
  - 别名替换
- 接下来按照如下顺序进行shell扩展：
  - 花括号扩展{1,2,3}: 按照预定的模式生成一组字符串
  - 波浪号扩展~:展开用户主目录
  - 参数和变量扩展$：获得参数或变量的值
  - 算术扩展$(( ))：获得表达式运算的值
  - **命令替换``or $( )：嵌套另一条命令**
  - 进程替换：进程的输入或输出替换为一个文件名
  - 单词分隔(word splitting)：前面**参数扩展、算术扩展、命令替换执行后的结果**再根据IFS(空格、Tab、换行等字符)分隔单词
  - (文件名)通配符扩展：展开成匹配的一个或多个文件名
  - 注：上述某个扩展展开后不再适用**当前以及之前**的扩展。花括号和命令替换允许嵌套
- 扩展完后决定命令部分和参数部分，进行环境变量设置和重定向，然后执行相应的命令

✓ 最初用于分割单词的元字符如下：
| < > & ; ( ) space tab newline
✓ 参数/算术/命令替换扩展的结果再基于IFS(缺省**space tab newline**)进行单词分割

花括号扩展（brace expansion）

- (文件)通配符扩展匹配的是与模式匹配**且存在**的文件名
- 如果单词出现花括号，进行花括号扩展。有一个例外，${...}不会进行花括号扩展。按照模式产生多个字符串，这些字符串之间以空格隔开。这些字符串与文件名无关，**不一定要存在实际的文件**
- 每个单词**独立进行花括号扩展**
- 花括号扩展模式前面有一个可选的前导(preamble)，后面有一个可选的附言（postscript）
  - 前导和附言部分存在时要求**原封不动地**出现在最后扩展的每个字符串中
  - 中间的扩展模式有两种格式，比如**{1,2,3,4}**或**{1..4}**
- 第一种扩展模式为通过逗号(,)分隔的多个字符串，即{str1,str2,⋯,strN}
  - 扩展为多个字符串，分别为str1,str2直到strN
  - 支持空字符串，即花括号开始和结束位置可为逗号，也允许出现连续的逗号，表示相应的位置为空字符串，如{**,**str2,str3}、{str1,str2**,**}、{str1**,,**str3}

echo file{1,2,3,4}.txt	file1.txt file2.txt file3.txt file4.txt
echo a2.c{,.bak}	a2.c a2.c.bak

花括号扩展（brace expansion）

- 第二种扩展模式是顺序表达式，其格式为**{x..y[..step]}**，x和y可以是整数或单个字符，**step**可选，为一个整数
  - 扩展为多个字符串，分别为x和y间（包括x和y在内）的各种取值
  - **step**为采用的步长。如果没有给出**step**，根据x和y的大小关系，缺省为1（x<y时）或-1（x>y时）
  - x和y为整数时，可在高位填充0表示产生的整数填充到对应的位数，{01..03} →01 02 03
  - x和y为单个字符时，表示采用按照**区域C**（即ASCII码顺序，0..9 A..Z a...z）排序确定的x 和y之间的字符。虽然也可 {D..d}，但是较少使用
  - 注意：x和y不能一个是整数(包括单个数字)，一个是单个字符，比如{0..A}

file{1..4}.txt	file1.txt file2.txt file3.txt file4.txt
file{6..4}.txt	file6.txt file5.txt file4.txt
file{1..6..2}.txt	file1.txt file3.txt file5.txt
t{9..011}.txt	t009.txt t010.txt t011.txt
sd{a..d}	sda sdb sdc sdd
f{d..a}.txt	fd.txt fc.txt fb.txt fa.txt

花括号扩展（brace expansion）

- 不同单词中的花括号扩展是独立的，每个单词的花括号扩展展开为空格隔开的多个字符串
- 如果一个单词中包括2个或多个**平行的花括号模式**，则产生的字符串为这些模式展开后的**笛卡尔积**
- 花括号扩展还允许嵌套，即一个逗号隔开的花括号扩展模式内部进一步包含花括号扩展
  - 内部的花括号扩展首先展开，其**展开的各个字符串以逗号**隔开，展开后替换原有的部分
  - 比如{…,{…}, …}，假设内部的花括号扩展为s1/s2/s3，替换为{…,s1,s2,s3, …}，进行进一步的扩展

```
$ echo {a..b} {1..2}
a b 1 2
$ echo {a..b}{1..2}
a1 a2 b1 b2
$ echo {a..b}{x,y}{1..2}
ax1 ax2 ay1 ay2 bx1 bx2 by1 by2
$ echo {a{1..3},b1,b2}.py      #1 {a1,a2,a3,b1,b2}.py
a1.py a2.py a3.py b1.py b2.py
$ echo src/{a{1..2},b{3,4}}.py    #1 {a1,a2,b3,b4}.py
src/a1.py src/a2.py src/b3.py src/b4.py
$ echo src/{a{1..2},b{3,4}}.{py,c}  #1 {a1,a2,b3,b4}.{py,c}
src/a1.py src/a1.c src/a2.py src/a2.c src/b3.py src/b3.c src/b4.py src/b4.c
```

花括号扩展（brace expansion）

- 每个单词独立进行花括号扩展
- 花括号扩展无法识别时并不会报错，而是原样输出
- 花括号扩展是最早展开的，展开后的内容可再进行其他扩展(包括参数扩展、文件通配符扩展等)

echo {1, 2}.txt	{1, 2}.txt 因为逗号后面有空格，被识别为两个单词， 即{1, 和2}.txt
echo {1..a}.txt	{1..a}.txt 花括号扩展无法识别时原样输出
echo {/home/*,/root}/.*profile	花括号扩展展开为/home/*/.*profile /root/.*profile 然后通配符扩展起作用，匹配/home目录下的子目录中所有以点开始最后为profile的文件以及/root目录下以点开始最后为profile的文件

```
dlmao@mars:~$ mkdir {2015..2016}-{01..12}
dlmao@mars:~$ ls -d 20*
2015-01  2015-04  2015-07  2015-10  2016-01  2016-04  2016-07  2016-10
2015-02  2015-05  2015-08  2015-11  2016-02  2016-05  2016-08  2016-11
2015-03  2015-06  2015-09  2015-12  2016-03  2016-06  2016-09  2016-12
dlmao@mars:~$ rmdir {2015..2016}-{01..12}
```

主要内容

- 查看命令的类型和位置： type/which/whereis/help
- bash环境文件
- 花括号扩展
- 命令替换
- 单词分割
- shell命令结构

命令替换（command substitute）

- 命令替换扩展用来嵌入另一条命令
  - 该部分的内容被所嵌入的命令执行后写到**标准输出中的内容**所替代
  - 替代时，如果内嵌命令的输出最后为一个或多个换行符时，那些换行符会被移走。但前面的换行符不会被移走
- 命令替换后的内容不会再进行花括号扩展、波浪号扩展、**参数扩展**和算术扩展等，但还可能会进行单词分割和文件通配符扩展
  - 内嵌命令输出中的换行符、空格、制表符在单词分割阶段作为单词间的分隔符
  - 双引号引用将包括的内容作为一个整体，同时也允许命令替换等扩展，因此可使用双引号避免单词分割和文件通配符扩展
- 可采用反引号(backquote)，即 `command`
- 也可采用$(command)格式，建议使用这种格式
- 赋值语句**不支持花括号/单词分割/通配符，但支持波浪号、参数、命令替换，扩展后结果不会进行单词分割**

```
$ ls -l $(which cp) #或 ls -l `which cp`
-rwxr-xr-x 1 root root 151024 Feb 18  2016 /bin/cp
$ NOW=$(date)
```

```
$ echo $(echo -e 'line1\nline2\n\n\n')
++ echo -e 'line1\nline2\n\n\n'
+ echo line1 line2
line1 line2
$ echo "$(echo -e 'line1\nline2\n\n\n')"
++ echo -e 'line1\nline2\n\n\n'
+ echo 'line1
line2'
line1
line2
```

命令替换（command substitute）

命令替换还可以嵌套

- 采用$(…)格式更加方便
- 采用反引号的格式时
  - 反引号表示命令替换的开始和结束，内部嵌套的命令替换中的shell元字符(包括反引号和反斜线)要通过反斜线转义
  - 每嵌套一层，都要对这些shell元字符进行一次转义，非常繁琐

```
echo $(which touch)
# 改变目录到touch命令所在的目录
cd $(dirname $(which touch))
cd `dirname \`which touch\``
# 如果要改变目录到touch命令所在目录的父目录
cd $(dirname $(dirname $(which touch)))
cd `dirname \`dirname \\\`which touch\\\`\``
```

单词分割(word splitting)

- 单词分割：之前的参数扩展、算术扩展及命令替换扩展后的内容，read命令读取的输入
- **双引号引用**将包括的内容作为一个单词，且支持参数扩展、算术扩展及命令替换，因此可使用双引号来**避免**单词分割和文件通配符扩展
- 根据IFS(**Internal Field Separator)** 分割成一个一个的单词
  - IFS缺省为空格、制表、换行符。如果IFS未设置，使用缺省值
  - **IFS为空时表示不进行单词分割**
- 对于赋值语句，各种扩展完成后的结果看成一个单词（等价于加上了双引号），不会再对结果进行单词分割

```
$ sentence="Push that word      away from me."
$ echo $sentence    # 单词分割起作用
Push that word away from me.
$ echo "$sentence"    # 不进行单词分割
Push that word      away from me.
$ foo=$sentence   #  可不加上引号，赋值语句不支持花括号、单词分割扩展、通配符扩展，但支持波浪号、参数、命令替换
$ PATH=$PATH:~/bin
$ cat patterns
??*
.?*
$ ls -d  $(cat patterns)   #等价于 ls -d ??* .?*
```

- echo:将命令行的内容放到标准输出
- 命令替换：标准输出的内容放到命令行
- cat:将文件(包括标准输入)中的内容放到标准输出

read命令

read [-s] [-n nchars] [-t timeout] [-p prompt] [var1 var2 …varN]

从标准输入读取一行字符，然后根据IFS分割成单词后逐个赋值给指定的变量，如果变量个数不够时，剩下的那些单词都赋值给最后一个变量varN。如果不提供变量，缺省为REPLY

- -p prompt首先输出提示符
- -n nchars等待用户输入nchars个字符或已输入换行符
- -t timeout等待用户输入的时间
- -s 不回显，用于输入密码等隐私信息
- -u fd 从文件描述字fd读，缺省从标准输入(fd 0)读

```
#!/usr/bin/env bash
read -p 'Please input your name...' name
echo "Nice to meet $name"
read -n 1 -p 'Press any key to quit....' key
echo "you press $key"
read -t 1 -p 'wait a second'; echo
```

```
$ echo 'idle,100,45' | (IFS=',';read name scores; echo "name=$name scores=$scores")
name=idle scores=100,45
```

- 可通过重定向自动执行交互式脚本，如echo -e 'mars\ny' | read-demo

主要内容

- 查看命令的类型和位置： type/which/whereis/help
- bash环境文件
- 花括号扩展
- 命令替换
- 单词分割
- shell命令结构

返回码(Exit code)

- 每个命令执行完成后会返回一个返回码(状态码)表示执行情况
- 状态码为0-255的整数，0表示执行成功，其他表示失败
- 建议1表示通用错误，127表示命令没有找到，126表示无法执行，128+n表示收到信号n而退出
- 特殊的参数$? 纪录了前一个命令执行返回的状态码
- 内置命令exit [n] 退出shell，可用于脚本中，结束脚本的执行，参数n为返回的状态码，如果没有参数时，返回最后一个命令执行的状态码
- 内置命令true(还有空命令:)和false仅返回状态码，分别为0和1
- 非运算符！command
  - 注意!之后需要有空格。否则表示历史事件扩展
  - 执行后面的命令。如果命令执行成功(状态码为0)，则!改变其状态码为失败(1)，如果命令执行失败，则状态码为成功(0)

```
$ :
$ echo $?
0
$ false
$ echo $?
1
$ true
$ echo $?
0
$ ! true
$ echo $?
1
```

```
# check if already connected to Internet
if  ! ping -c 1 -w 3 www.baidu.com &> /dev/null; then
    ~/bin/fd-net-auth.sh
fi
```

~/.profile

shell命令结构

- 简单命令：设置相应的NAME变量，执行COMMAND [ARGS]

  [NAME=VALUE]... [COMMAND [ARGS]...]

- 管道(pipeline)：通过管道元字符(|或|&)分割的多个命令，简单命令是一个特殊管道命令

  [!] command1 | command2… |& command…

  - | 表示上一个命令的标准输出作为下一个命令的标准输入
  - |& 表示上一命令的标准和错误输出为下一命令的标准输入，即 2>&1 |
  - 管道中的每个命令都是启动一个新的进程来执行的
  - 管道的返回状态码缺省为最后一个命令执行后的返回状态码
  - 如果pipefail选项开启(set –o pipefail)，组成管道的命令中最后一个执行
  - 不成功的命令的状态码作为管道的状态码，如果所有命令都成功时为0

- 最前面可选的!，就是前面介绍的非运算符，表示改变管道命令最后返回的状态码

```
$ ls |grep _not_found_ | wc -l
0
$ echo $?
0
$ set -o pipefail
$ ls |grep _not_found_ | wc -l
0
$ echo $?
1
```

命令列表(list)

- 命令列表指的是通过；& &&或者 ||分割的一个或多个管道命令(包括简单的命令)，最后可能以；&或换行符结束
- ; 分号分割的命令表示顺序执行，如 cmd1; cmd2，这些顺序执行的命令列表的状态码为最后一个命令执行的状态码
- & 表示启动一个后台进程来执行命令，并不需要等待后台进程完成，返回状态码为0，如 cmd&
- &&和||为条件执行运算符，采用短路逻辑运算。&&优先级更高
  - cmd1 && cmd2 与(and)运算符，前一个命令执行成功(echo $?返回0) 时执行下一条命令
  - cmd1 || cmd2 或(or)运算符，前一个命令执行不成功时执行另一条命令
  - 返回的状态码为最后执行的那条命令返回的状态码
  - 注意shell中的逻辑运算不是对于命令的输出结果进行运算，而是对命令执行的状态码进行运算

```
test -r ~/.bash_aliases && cat ~/.bash_aliases
#只有在创建目录成功时才切换到该目录
mkdir d && cd d
test -r ~/.bash_aliases || echo ".bash_aliases doesn't exist"
rm file || { echo 'Could not delete file!' >&2; exit 1; }
```

组命令和子shell

bash用于分割单词的元字符如下：
< > | & ; ( ) space tab newline

- 组命令 { list; }
  - 在**当前shell环境**下执行命令列表中的多条命令。相比子shell速度更快
  - 花括号{后必须有一个空格，list必须以分号或换行结束
  - 组命令将多条命令当成一个整体，一般用于描述shell函数的函数体，或者进行逻辑非和逻辑与运算，或者用在需要将list中的命令的输出都进行重定向的场景
- 子shell （list)
  - 启动一个新的子shell，在新shell环境中执行list中的命令，对于环境的修改不会影响父shell
  - 圆括号与list之间不需空格，list的最后一条命令也无需以分号或换行结束

```
{ cd /usr/bin;ls;} >/tmp/bin_list.txt
(cd /usr/bin; ls)  >/tmp/bin_list2.txt
{cd /usr/bin;ls;}  >/tmp/bin_list.txt
{ cd /usr/bin;ls} >/tmp/bin_list.txt
{ cd /usr/bin;ls } >/tmp/bin_list.txt
```

- 采用组命令，工作目录也改变为/usr/bin
- 采用子shell，工作目录不变
- 组命令中，如果{后面没有空格，解析为{cd
- 组命令中，如果}前面没有;，解析为ls}
- 组命令中，如果}前面没有;，而是空格，解析为ls }

```
rm file || { echo 'Could not delete file!' >&2; exit 1; }
echo 'idle,100,45' | (IFS=',';read name scores; echo "name=$name scores=$scores")
```