Bash基础介绍

What is shell

Linux可以分为三层，Shell、API和Kernel

- shell是一个命令解释器，即处理您在终端仿真器（交互模式）中输入的命令或处理shell脚本（包含命令的文本文件）（批处理模式）的程序
- API是通用术语定义开发人员在使用库和编程语言编写代码时必须使用的界面
- 内核是与硬件（CPU，RAM，磁盘，网络，...）交互的低级程序，其中应用程序正在运行

可以说，Shell是类Unix系统中一种与系统交互的独特方式。

Shell支持的功能

Unix shell既是命令解释器又是编程语言

- 作为命令解释器，shell为丰富的GNU实用程序提供了用户界面
- 作为编程语言，shell支持创建包含命令的文件，并成为新的command，并允许自动执行
- Shell可以分为同步执行和异步执行
- Shell支持从键盘和文件读取输入
- Shell支持输出重定向

什么是Bash

Bash全称Borne Again Shell. Bash 融合了KShell和CShell的有用功能，提供对交互式和编程使用的功能改进。

Linux系统中默认的Shell就是Bash

如何理解Shell和Bash

Shell是所有unix系统中命令解释器的统称，Bash则是一种更加完善的、支持unixShell功能的编程语言

如何查看系统的默认shell

```
ls -l $(which sh)
```

Bash名词解释

> `POSIX`：基于Unix的开放系统标准系列

> `builtin`: 系统自建命令,如cd、ls、pwd等
>
> `control operator`: 控制符，如||、&&、&、; 等
>
> `exit status`: 脚本的退出返回值, [0,255]
>
> `job`: 一组进程的集合，包括pipeline和从当前进程中fork出来的其他进程
>
> `job control`: 任务控制， 包括停止(suspend)和恢复(resume)进程
>
> `return status`: 函数的返回值，[0,255]
>
> `signal`: 信号
>
> `交互式Shell`: 输入和输出都连接到终端的命令

Shell Synatax

转义符

Bash 转义符是 `\`

- 当 `\` 用于单个字符时，将下一个字符转为原始值
- 当 `\` 用于换行符时，回车将不会执行command，而是继续接收下一行输入

单引号

- 被 `'` 包含的字符串，将不会做任何替换

举个例子:

`echo 'test for ${SingleQuotes}'` 将输出 `test for ${SingleQuotes}`

双引号

- 被 `"` 包含的字符串，会对特殊字符进行替换，例如 `$` 、 `!` 等

`echo "test for ${SingleQuotes}"` 将输出 `test for`

ANSI-C字符

> `\a` alert
>
> `\b` backspace
>
> `\e` `\E` an escape character

> `\n` newline
>
> `\r` carriage return
>
> `\t` horizontal tab
>
> `\v` vertical tab
>
> `\\` backslash
>
> `\nnn` 八进制
>
> `\xHH` 十六进制
>
> `\uHHHH` Unicode
>
> `\uHHHHHHHH` Unicode

区域转换

> `$"String"`: 以 `$` 开头，被 `"` 包围的字符串将根据地区进行转换

注释

> `#` 用于注释一行代码

Notice: 交互式Shell不允许注释

Shell Command

Pipeline

Pipeline是使用 `|` 或者 `&` 进行分隔的一系列有序命令, 其中上一个命令的输出是下一个命令的数据

Format for a pipeline:

```
command1 [; or && or || ] command2 [&] []
```

> `|`: command1的标准输出到command2的标准输入
>
> `|&`: command1的标准输出和标准错误输出到command2的标准输入
>
> `2>&1`: 同 `|&`

命令列表

> `&`: 异步执行一条指令, 也成为后台执行

> `;`: 顺序执行指令, 不关心上一条指令的执行是否成功
>
> `&&`: 顺序执行指令，并且在上一条指令出错时退出
>
> `||`: 在上一条指令执行失败时，才执行下一条指令

举个例子, 测试字符串是否为空:

`test -z "NotEmpty" && echo "true" || echo "false"`

流程控制

循环

> 执行consequent-commands直到test-commands返回非0

```
until test-commands
do
    consequent-commands
done
```

> 当test-commands返回非0, 执行consequent-commands

```
while test-commands
do
    consequent-commands
done
```

> for循环分为forEach和普通for循环

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

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

> 例子: `for file in $(ls); do echo ${file}; done`

条件语句

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

```
case word in [ [(] pattern [| pattern]…) command-list ;;]… esac
```

> `;;` 类似于break，执行完之后将直接跳出case语句
>
> `;&` 执行完之后将执行下一个分支

组合指令

`(command1 && command2)` 将产生一个新的subshell执行环境，command1和command2将在新的subshell执行

`{command1 && command2}` command1和command2将在当前当前shell环境中执行

Shell Function

function定义了一系列的操作集合，举个例子:

```
function echoParams {
    local param1=${1}
    local param2=${2}
    return 0
}
```

Function定义需要注意:

- `function funcName` or `funcName()` 无需显式定义形参
- local关键字用于在方法内部定义内部变量
- 方法的参数使用 `blank` 和 `tab` 进行分个，调用时推荐使用 `funcName "var1" "var2"` 的格式保证参数个数正确

Shell Parameters

普通变量

> `name=[value]` 赋值一个变量给name。注意 `=` 中间不能有空格
>
> `declare -n` 定义一个引用
>
> `declare -A` 定义个数组
>
> `local` 定义一个局部变量. Bash默认所有变量都是全局变量
>
> `unset` 删除一个变量
>
> #### 位置变量

> Bash使用位置标示命令行参数和方法参数的变量.

例如 `${1}` 表示第一个参数 `${2}` 表示第二个参数

对于脚本文件来说， `${0}` 代表运行脚本的命令

特殊变量

> `${*}` 获取所有使用 `"` 括起来的位置参数，使用 `blank` 进行分隔

```
for i in "${@}"; do echo "@ '$i'"; done
./testvar foo bar baz 'long arg'
输出:
@ 'foo'
@ 'bar'
@ 'baz'
@ 'long arg'
```

> `${@}` 将所有位置参数拼成一个字符串输出

```
for i in "${*}"; do echo "* '$i'"; done
./testvar foo bar baz 'long arg'
输出:
* 'foo bar baz long arg'
```

> `${#}` 获取位置参数的参数个数。 不包含 `${0}`

```
echo "args count: ${#}"
./testvar foo bar baz 'long arg'
输出:
args count: 4
```

> `${?}` 获取方法、文件的返回值

```
function func {
    return 1
}
echo "return: ${?}"
输出:
return: 1
```

> `${$}` 获取当前shell进程的进程ID。如果被 `()` 包含，那么返回调用进程的进程ID，而不是subshell的进程ID. `echo ${$}; (echo ${$})` 输出结果一致
>
> `${!}` 返回最近一次执行的后台进程的进程ID

```
ls &
echo ${!}
```

> `${0}` 执行的Shell名称或者Shell脚本的名字

```
echo ${0}
输出:
-bash
```

Shell Expansions

大括号展开

> `{a,b,c}` 展开为单个字符

```
echo a{d,c,b}e
输出:
ade ace abe
```

> `{1..10}` 将拓展为1~10的数组

```
echo {0..10}
输出:
0 1 2 3 4 5 6 7 8 9 10
```

波浪符展开

> `~` The value of $HOME。 `cd ~` 切换到$HOME目录
>
> `~/foo` $HOME/foo `cd ~/foo` 将切换到$HOME/foo目录
>
> `~fred/foo` 用户fred的foo目录
>
> `~+/foo` $PWD/foo
>
> `~-/foo` ${OLDPWD-'~-'}/foo

参数拓展

> `${parameter:-word}` 当 `parameter` 是unset或者null时，使用*word*内容来替换 `parameter`

```
echo ${a:-test}
输出:
test
```

> `${parameter:=word}` 当 `parameter` 是unset或者null时，使用*word*内容来替换 `parameter` .
> 但是对于 `${0}` 这样的位置参数无效

```
echo ${a:+test}
输出:
test
```

> `${parameter:?word}` 当 `parameter` 是unset或者null，并且处于前台运行时, 打印标准错误

```
echo ${a:?test}
输出:
-bash: a: test
```

> `${parameter:+word}` 当 `parameter` 不是null时，才替换为*word*

```
a=10 && echo ${a:+test}
输出:
test
```

> `${parameter:offset[:length]}` 截取 `parameter` 从*offset*开始，*length*个长度的字符串

```
string=01234567890abcdefgh
echo ${string:7}
输出:
7890abcdefgh

echo ${string:7:2}
输出:
78

echo ${string:7:-2}
输出:
7890abcdef

string=01234567890abcdefgh
echo ${string:-7:0}
输出:
7890abcdef

echo ${string: -7:-2}
bcdef
```

```
array=$(echo {1..10})
```

> `${!array[@]}` 获取数组 `array` 所有的 `key`

```
array=(1 2 3 4 5 6)
echo ${!array[*]}
输出：
0 1 2 3 4 5
```

> `${#parameter}` 获取 `parameter` 的长度

```
string=12345
echo ${#string}
输出：
5
```

命令展开

> `$(command)` 用于将子shell的标准输出连接到变量中

```
files=$(ls -la) && echo "${files}"

输出：
drwxr-xr-x    3 mingweiliu  staff   102  8 22 12:18 .
drwx------+ 252 mingweiliu  staff  8568  8 22 12:18 ..
-rw-r--r--    1 mingweiliu  staff     0  8 22 12:18 hellworld.sh
```

这是Bash最常用的展开，用于获取一个方法或者一个指令的执行输出

计算表达式展开

> `$(( expression ))` 用于展开一个算数表达式, 并输出计算结果

```
echo $(( 10 + 21 ))
输出：
31
```

重定向

重定向是一个复杂的工作，可以分成三种:

- 重定向标准输出
- 重定向标准输入
- 重定向标出出错

描述符

> `/dev/stdin` 标准输入, fd=0

> `/dev/stdout` 标准输出, fd=1
>
> `/dev/stderr` 标准错误, fd=2
>
> `/dev/tcp/host/port` 连接到相应的tcp服务
>
> `/dev/udp/host/port` 连接到相应的udp队服

一般来说，标准输入、标准输出和标准错误使用的比较多。

重定向操作符

> `[fd=0]<word` 从*word*文件中读取数据
>
> `[fd=1]>word` 覆盖*word*标标识的文件，*fd*用于标示描述符
>
> `[fd=1]>>word` 追加*word*标标识的文件，*fd*用于标示描述符
>
> `&>word` `>&word` `>word 2>&1` 标准错误和标准输出都覆盖*word*文件
>
> `&>>word` `>>word 2>&1` 标准错误和标准输出追加到*word*文件

重定向的优先级

考虑两种写法: `ls > dirlist 2>&1` 和 `ls 2>&1 > dirlist`

这两种写法将导致完全不同的结果

- 第一种将标准输出和标准出错全部重定向到dirlist中
- 第二种将标准输出重定向到dirlist，标准错误重定向到标准输出

出现不同结果的原因是，第二种写法当标准错误重定向到标准输入时，标准输入还是输出到控制台的，所以当后面标准输出重定向到*dirlist*时，标准错误依然是输出到控制台的

高级特性

Shell的高级特性，例如多进程、任务控制、信号控制等功能可以参考manpage，不在本文档的覆盖范围

常用命令介绍

日常工作中，一些常用的工具和命令在这里介绍下

gnu命令核心包

OSX下的Bash命令和gnu下的还是有一些区别的，如果想在OSX下体验，可以安装gnu核心包. 安装命令: `brew install coreutils`

grep

grep家族的命令用于从一行数据中匹配某些字符或者字符串

> `grep -[A|B|C] [n] word` 获取*word*关键字所在行的[After|Before|After and Before] *n*行数据
>
> `grep -c word` 获取包含*word*关键字的总行数
>
> `grep -E [pattern]` 使用正则表达式进行匹配

sed

sed是一个文件行流式编辑器，执行问本行的替换、删除、截取等操作。 基本语法是:
`sed [-Ealn] [-e command] [-f command_file] [-i extension] [file ...]`

> `sed 's/str/replace/g` 将*str*全局替换成*replace*
>
> `sed -n '1,10p'` 打印1~10行
>
> `sed '1,10d'`删除1~10行

awk

awk是一门新的编程语言，主要用户问本行切分、计算等操作。 这里只做一些最简单的介绍，完整的用法可以参考《Effective AWK Programming》

> `awk -F ',' '{print $0}'` 根据`,`分隔，打印每一行
>
> `awk -F ',' '{for(i=1;i<=NF;++i) printf("%s\n", $i);}'` 根据`,`分隔，将每一列转为一行
>
> `awk -F ',' '{sum+=$3;count+=1;}'END'{print sum, sum/count;}'` 根据`,`分隔，计算第三列的总和平均值

sort和uniq

sort用于将行文本排序.

uniq是一个过滤重复数据的命令.

通常来说sort和uniq是一堆好基友

> `sort -n -r` 按照数字类型倒序排列
>
> `sort -k1` 按照第一列进行排序
>
> `sort -k1 | uniq -c |sort -k1 -r -n|head -5` 四条指令用于从一个文件中根据获取第一

> 列关键字出现次数前5的行

read

从控制台或者文件中读取数据。read在进行分隔时，使用IFS变量进行分隔

> `read a b c` 从控制台读取*value1 value2 value3* 格式的值到a b c中
>
> `read -t 3 -n 10 value` 从控制台读取字符串到value. 在等待3秒或者输入10个字符后退出
>
> `read -p "$"` 增加提示符 `$`

echo

打印变量或者日志到控制台

> `echo -n "test"` 不输出换行符
>
> `echo -e "\033[31m red output \033[0m"` 输出红色*red output*

xargs

xargs用于从标准输入读取数据，并且执行其他命令。

考虑一种情况，需要从文件中逐条获取关键字，并且计算该关键字在其他文件中出现的次数， 那么仅仅使用管道就不够了，此时需要用到xargs来运行grep命令

> `cat index.txt |xargs -I {} -t grep {} test.txt -c` 从test.txt中获取所有存在于index.txt中的关键词出现的次数

从上述例子可以看到，xargs可以帮住我们重新整理命令在管道中的执行顺序，达到自由组合的功能

其他参数和功能可以参见ManPage