Bash

xbZhong

2025-12-19

本页PDF

Shell

`Shell` 是什么，可以从以下几个方面来理解

- Shell是用户跟内核交互的对话界面
- 是一个程序，提供与用户对话的环境，即命令行环境（command line interface，简写为 `CLI` ）
- 同时也是命令解释器和工具箱

而 `Bash` 就是 `Shell` 里面的一类，是目前**最常用的** `Shell`

Bash

在 `Linux` 系统中，用户的默认 `Shell` 都是 `Bash` ，但是用户可以使用不同的 `Shell`

通过 `chsh` **命令改变系统的默认** `Shell`

- `chsh` 是 `change shell` 的缩写
- `-s` 表示 `shell` ，后面紧跟想切换的 `Shell` 路径
- `/usr/bin/fish` 表示 `fish shell` 程序在文件系统中的绝对路径

```
$ chsh -s /usr/bin/fish
```

可以通过输入 `bash` 命令启动bash

```
$ bash
```

退出 `bash` 环境，可以使用 `exit` 命令

```
$ exit
```

查看 `bash` 的版本

```
$ bash --version
# 或者
$ echo $BASH_VERSION
```

在上述命令中，`$`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**

echo命令

主要作用：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
# 输出
hello world
```

若想要输出**多行文本**，且包括换行符，则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>"
```

`-n`参数

默认情况下，`echo`输出的文本末尾会有一个换行符，`-n`参数可以取消末尾的回车符

```
$ echo -n hello world
# 输出
hello world$
```

`-e`参数

`-e`参数会解释引号里的特殊字符，如换行符`\n`，若不使用`-e`参数，会原样输出

```
$ echo "hello\nworld"
# 输出
hello\nworld

$ echo -e "hello\nworld"
# 输出
hello
world
```

命令格式

`bash` 单个命令一般是一行，如果需要把**一行的命令写成多行**便于理解，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
# 等同于
$ echo foo \
bar
```

分号 `;` 是命令的结束符，使得一行可以放多个命令，上一个命令结束后再执行第二个命令

```
$ clear; ls
```

命令组合符 `&&` 和 `||`

- 如果 `command1` 运行成功，则继续运行 `command2` 命令

```
command1 && command2
```

- 如果 `command1` 命令运行失败，则继续运行 `command2` 命令

```
command1 || command2
```

模式扩展

工作流程： `Shell` 接受到用户输入的命令后，会将用户的输入拆分为一个个词元，然后 `Shell` 会**扩展词元里面的特殊字符**

模式扩展也被称为 `globbing` ，其中有些用到了**通配符**，又称为**通配符扩展**， `Bash` 一共提供八种扩展

- 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

`Bash` 允许用户开启扩展

```
$ set -o noglob
# 或者
$ set -f
```

`Bash` 也允许用户关闭扩展

```
$ set +o noglob
# 或者
$ set +f
```

波浪线扩展

波浪线 `~` 会自动扩展成当前用户的主目录

- `~/dir` 表示扩展成主目录的某个子目录， `~` 是主目录里面的一个子目录名
- `~user` 表示扩展成用户 `user` 的主目录
- `~+` 会扩展成当前所在的目录，等同于 `pwd` 命令

```
$ echo ~
# 输出
/home/me
```

字符串扩展

`?` 字符代表文件路径里面的任意单个字符，不包括空字符

- 如果需要匹配多个字符，就需要使用多个 `?`

`*` 字符扩展

`*` 字符代表文件路径里面的任意数量的任意字符，包括**空字符**

- `*` 只匹配当前目录，不会匹配子目录

```
# 存在文件 a.txt、b.txt 和 ab.txt
$ ls *.txt
# 输出
a.txt b.txt ab.txt
```

但是 `*` 不会匹配隐藏文件，因为隐藏文件以 `.` 开头，如果要匹配隐藏文件，需要写成 `.*`

```
# 显示所有隐藏文件
$ echo .*
```

方括号扩展

形式是 `[...]` ，只有文件**确实存在的前提**下才会进行扩展

- 比如， `[aeiou]` 可以匹配五个元音字母中的任意一个
- 如果需要匹配连字号 `-` ，只能放在方括号内部的开头或结尾

```
# 存在文件 a.txt 和 b.txt
$ ls [ab].txt
# 输出
a.txt b.txt
```

两个变体

- `[^...]`：表示匹配不在方括号里面的字符
- `[!...]`：表示匹配不在方括号里面的字符

[start-end]扩展

表示匹配一个**连续的范围**，比如，`[a-c]`等同于`[abc]`，`[0-9]`匹配`[0123456789]`

下面是一些常用简写例子：

- `[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

简写的否定形式`[!start-end]`，表示**匹配不属于这个范围的字符**

```
$ ls report[!1–3].txt
# 输出
report4.txt report5.txt
```

上面代码中，`[!1-3]`表示排除1、2和3

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**

- 大括号内部的逗号前后**不能有空格**，否则会失效

```
$ echo {1,2,3}
# 输出
1 2 3

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

大括号可以**嵌套**

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

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 /bin/{cat,b*}
# 输出
/bin/cat /bin/b2sum /bin/base32 /bin/base64 ... ...
```

且可以用于**多字符模式**

```
$ echo {cat,dog}
# 输出
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，**即如果匹配的文件不存在，它也会扩展**

```
# 不存在 a.txt 和 b.txt
$ echo [ab].txt
# 输出
[ab].txt
$ echo {a.b}.txt
# 输出
a.txt b.txt
```

{start..end}扩展

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}
# 输出
a b c

$ echo d{a..d}g
# 输出
dag dbg dcg ddg

$ echo {1..4}
# 输出
1 2 3 4
```

支持逆序

```
$ echo {c..a}
# 输出
c b a

$ echo {5..1}
# 输出
5 4 3 2 1
```

若遇到无法理解的简写，会按照原样输出

```
$ echo {a2..3c}
# 输出
{a2..3c}
```

这种简写方法可以嵌套使用，形成复杂扩展

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

大括号扩展的常见用途为新建一系列目录

- 先扩展前面再扩展后面

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

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

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

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

```
$ echo $SHELL
# 输出
/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
```

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

```
$ ehco $(date)
# 输出
Tue Jan 28 00:01:13 CST 2020
```

引号和转义

`Bash` 只有字符串一种数据类型，因此字符串相关的引号和转义，对 `Bash` 来说是非常重要的

转义

某些字符，如 `$` 、 `&` 、 `*` ，在 `Bash` 里面有特殊含义

```
$ echo $date
# 不会输出结果
```

需要加上 `\` 进行转义

```
$ echo \$date
# 输出
$date
```

如果要输出反斜杠 `\` 本身，需要对它自身进行转义

```
$ echo \\
# 输出
\
```

单引号

`Bash` 允许字符串放在单引号或者双引号中间，各种特殊字符在单引号里面都会被视为普通字符

```
$ echo '*'
# 输出
*

$ echo '$USER'
# 输出
$USER
```

双引号

双引号比单引号宽松，大部分特殊字符在双引号里面会失去特殊含义，但 `$` 、 `` ` `` 、 `\` ，在双引号之中仍会被进行扩展

```
$ echo "$SHELL"
# 输出
/bin/bash

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

双引号还可以保留原始命令的输出格式

```
# 单行输出
$ echo $(cal)
一月 2020 日 一 二 三 四 五 六 1 2 3 ... 31

# 原始格式输出
$ echo "$(cal)"
      一月 2020
日 一 二 三 四 五 六
          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
```

Here文档

是一种输入多行字符串的方法，格式如下

```
<< token
text
token
```

分为开始标记 `<< token` 和结束标记 `token`

- 开始标记是 `<<` 加上Here文档的名称，可任取，后面必须是换行符
- 结束标记是单独一行的Here文档名称，若不顶格，则不起作用
- 在文档内部，**双引号和单引号会失去作用，不支持通配符控制，支持反斜杠转义**

变量

可分为**自定义变量和环境变量**两类

环境变量

是 `Bash` 环境自带的变量，可以直接使用

- 大小写敏感

可以使用 `env` 或者 `printenv` 显示所有环境变量

```
$ env
# 或者
$ printenv
```

查看单个环境变量的值，使用 `printenv` 或者 `echo` 命令

- `printenv` 后面不加 `$`

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

常见的环境变量

- `HOME`：用户的主目录
- `PWD`：当前工作目录
- `USER`：当前用户的用户名
- `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

自定义变量

是用户在当前 `Shell` 里面自己定义的变量，一旦退出当前 `Shell`，变量就不存在了

`set` 命令可以显示所有变量

```
$ set
```

创建变量

有下述要求：

- 字母、数字和下划线组成
- 第一个字符不能是数字
- 不能出现空格和标点符号

语法如下：

```
variable=value
```

读取变量

直接在变量名前面加上 `$` 即可

```
foo=bar
$ echo $foo
# 输出
bar
```

读取变量的时候，变量名也可以使用花括号 `{}` 包围

```
$ a=foo
$ echo $a_file
# 无输出

$ echo ${a}_file
# 输出
foo_file
```

删除变量

`unset` 命令用于删除变量，但仅仅是把变量设定成了空字符串

```
unset NAME
```

输出变量

主要作用是**设置或显示环境变量**，让之后运行的所有子 `Shell` 都可以读取到这个变量

```
NAME=foo
export NAME
# 等同于
export NAME=foo
```

子 `Shell` 如果修改继承的变量，不会影响父 `Shell`

字符串操作

获取字符串长度

语法如下，大括号不可省略

```
${#varname}
```

获取子字符串

语法如下

- 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`
- 只能通过变量读取字符串

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

改变大小写

转为大写

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

转为小写

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

算术运算

`((...))` 语法可以进行整数的算术运算

- 你能想到的位运算，逻辑运算，赋值运算都支持，这里不赘述

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

行操作

一些快捷键

- `Ctrl + a`：移到行首
- `Ctrl + e`：移到行尾
- `Ctrl + l`：清除屏幕

目录堆栈

cd -

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

pushd，popd

可以记忆多重目录，`pushd` 和 `popd` 用来操作目录堆栈

`pushd`：用于进入指定的目录，同时把进入的目录放入堆栈

- 目录参数：`pushd` 可以接受一个目录作为参数，把该目录放到堆栈顶部，并进入该目录

```
$ pushd dirname
```

`popd`：不带参数时为移除堆顶的顶部记录，并进入新的栈顶目录

- `-n`：仅操作堆栈，不改变目录

```
$ popd
```

dirs

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

- `-c`：清空目录栈

```
$ dirs
# 输出
~/foo/bar ~/foo ~
```

操作历史

`Bash` 会保留用户的操作历史，默认保存**最近的500条命令**

- 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
# 输出
/home/me/.bash_history
```

`history` 命令会输出 `.bash_history` 文件的全部内容，即输出操作历史

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

- 返回包含 `/usr/bin` 的命令

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

使用 `history` 命令的 `-c` 参数清除操作历史，即清空 `.bash_history` 文件

```
$ history -c
```

脚本入门

执行脚本时**会创建一个子** `Shell` **执行脚本**

Shebang行

脚本的第一行通常是指定解释器，即这个脚本通过什么解释器执行

`#!` 后面就是脚本解释器的位置

```
#!/bin/sh
# 或者
#!/bin/bash
```

执行权限

使用 `chmod` 命令修改权限，全称为 `change mode`

权限表

	读(r)	写(w)	执行(x)
用户(u)			

	读(r)	写(w)	执行(x)
用户组(g)			
其他人(o)			

正常写法

语法：用户+权限，如 `u+r` ， `o-w` ，后面跟文件名

```
$ chmod u+x test.sh
```

权限的数字写法

对于不同用户的权限设置可以抽象为三个数字 `a b c`

- `a` ：用户
- `b` ：组
- `c` ：其他人

而不同的权限可以抽象成三个数字，**在给不同用户进行权限设置的时候在这三个数字选几个加起来**

- `r` （读）：4
- `w` （写）：2
- `x` （执行）：1

举个例子

- `755` ：拥有者拥有 `rwx` 的权限，用户组和其他人拥有 `r-x` 的权限
- `644` ：拥有者拥有 `rw-` 的权限，用户组和其他人拥有 `r--` 的权限

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

```
# 给所有用户执行权限
$ chmod +x script.sh

# 给所有用户读权限和执行权限
$ chmod +rx script.sh
# 或者
$ chmod 755 script.sh

# 只给脚本拥有者读权限和执行权限
$ chmod u+rx script.sh
```

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

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

可以把上面一行写入 `~/.bashrc` ，因为每次开一个新的终端，系统都会自动加载 `~/.bashrc` 里面的配置，也就是每次都会更新上面这个环境变量，方便我们快速执行脚本

立即重新加载配置

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

脚本参数

调用脚本的时候，脚本文件吗后面可以带有参数

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

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

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

同时它也可以接收一个整数作为参数，指定所要移除的参数个数，默认为 `1`

```
$ shift 3
```

exit命令

用于终止当前脚本执行

- 退出值为0，成功

```
$ exit 0
```

- 退出值为1，失败

```
$ exit 1
```

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**，也可以简化为一个 `.`

```
$ source ~/.bashrc
# 或者
$ . ~/.bashrc
```

alias命令

`alias` 用来为一个命令指定别名

- `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**

```
$ alias NAME=DEFINITION
```

为 `grep` 命令起一个 `search` 的别名

```
$ alias search=grep
```

直接调用 `alias` 命令可以显示所有别名

```
$ alias
```

`unalias` 可以解除别名

```
$ unalias lt
```

条件判断

使用 `if` 、 `elif` 、 `else` ，结构如下

- `commands` 为判断条件，可以用 `test` 命令
- `if` 和 `then` 写在同一行时，需要分号分隔
- `if` 可以接任意数量的命令，但是判断真伪只看最后一个命令

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

test命令

写法有三种形式，**第三种支持正则判断**，其它两种不支持

- `expression` 是一个表达式，表达式成功返回0，否则返回1
- `[` 和 `]` 与内部的表达式之间必须有空格

```
# 写法一
test expression

# 写法二
[ expression ]

# 写法三
[[ expression ]]
```

逻辑运算

- `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 `!`

常见判断

判断文件状态

- `[ -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`

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]` ：按照字典序比较 `string1` 和 `string2`
  - 命令内部的 `>` 和 `<` 必须**要用引号引起来**

整数判断，不能简单用 `>` 和 `<` （会被误认为字符串比较）

- `-eq` ：判断是否相等
- `-ne` ：判断是否不相等
- `le` ：判断是否小于等于
- `lt` ：判断是否小于
- `ge` ：判断是否大于等于
- `gt` ：判读是否大于

```
if [integer1 -eq integer2]; then
    echo "相等"
    exit 1
fi
```

循环

都支持 `break` 和 `continue`

while循环

结构如下

- `condition` 也可以用 `test` 命令
- `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 condition; do
  commands
done
```

until循环

结构如下

- 只要不符合判断条件就不断循环执行指定语句

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

for...in 循环

用于遍历**列表**的每一项

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

for循环

结构如下

- `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

- 循环条件放在**双重圆括号**之中，圆括号之中使用变量不用加上美元符号 `$`

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

函数

优先级：别名 > 函数 > 脚本 > 外部命令

定义：两种定义方法

```
# 第一种
fn() {
  # codes
}

# 第二种
function fn() {
  # codes
}
```

调用时直接写函数名，**参数跟在函数后面**

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

$ hello world
Hello world
```

删除函数

```
$ unset -f functionName
```

查看当前 `Shell` 已定义的所有函数

```
$ declare -f
```

查看单个函数的定义

```
$ declare -f functionName
```

查看所有已经定义的函数名，不包括函数体

```
$ declare -F
```

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
- `$@`：函数的全部参数，参数之间使用空格分隔
- 如果函数的参数多于9个，那么第10个参数可以用 `${10}` 的形式引用

变量作用域

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

如果需要声明局部变量，需要使用 `local` 命令

```
fn () {
  local foo
  foo=1
  echo "fn: foo = $foo"
}
```

数组

创建数组

采用**逐个赋值**的方法创建

```
ARRAY[INDEX]=value

# 例子

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

采用**一次性赋值**的方式创建

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

# 等同于

ARRAY=(
  value1
  value2
  value3
)
```

定义数组的时候可以使用通配符

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

可以使用 `declare` 命令先声明数组

```
$ declare -a ARRAYNAME
```

读取数组

读取数组指定位置的成员

```
$ echo ${array[i]} # i是索引
```

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

- 使用这俩个特殊索引的时候**最好放在双引号里**

```
$ echo ${foo[@]}

# 放在双引号里
$ hobbies=( "${activities[@]}" )

# 还可以添加成员
$ hobbies=( "${activities[@]}" diving )
```

数组长度

使用下面两种语法

```
${#array[*]}
${#array[@]}
```

提取数组序号

使用下面两种语法，可以返回数组的成员序号，即哪些位置是有值的

```
${!array[@]}
${!array[*]}
```

提取数组成员

`${array[@]:position:length}` 的语法可以提取数组成员

```
${array[@]:position:length}
```

追加数组成员

可以使用 `+=` 运算符

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

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

删除数组

使用 `unset` 命令删除一个元素

```
$ 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
```

使用 `unset` 命令删除整个数组

```
unset ArrayName
```

set命令

脚本会运行在子 `Shell` 中，而 `set` 命令用于改变子 `Shell` 环境的运行参数，保证脚本的安全性

set -u

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

```
set -u

echo $a
echo bar
```

set -x

用来在运行结果之前，先输出执行的那一行命令，若要关闭命令输出可以使用 `set +x`

```
set -x
echo bar

# 输出

+ echo bar
bar
```

set -e

只要脚本发生错误就终止执行，用 `set +e` 表示关闭这个功能

```
set -e

foo
echo bar

# 会终止执行
```

set -n

不运行命令，只检查语法是否正确

```
set -n

foo
echo bar
```