ZshGuide

发布 *latest*

2020 年 08 月 01 日

Contents

CHAPTER 1

01 变量和语句

网上关于 zsh 的文章有很多，但其中超过 95% 的文章讲如何使用和配置，写如何用 zsh 编程的文章很少，能找到的多数也是只言片语，不成系统。国外有几本讲 zsh 的书，其中也有很多内容是配置、使用、编写补全脚本等等，对编程有用的篇幅占比并不多，而且比较零散不便于查询。至于官方文档？那是让即使有多年编程经验的开发者也会抓狂的神奇存在。可读性极差，而且基本没有例子，不熟悉文档结构和内容的话，很难找到自己想要的东西。但内容覆盖很全面，洋洋洒洒近 500 页，耐心去看总会找到的。还有一份官方“入门”文档，上次更新时间是 2002 年，也要 300 多页，至于可读性，比官网文档要稍微好一些吧，还是有一定的参考价值的。官网上还有一些链接，里边内容比较零散，也可以看看。

很多人在 zsh 中用 bash 语法写脚本，虽然也可以正常运行，但这样无法利用 zsh 的众多优秀特性，还是非常遗憾的。熟悉下 zsh 下独有的特性，对写脚本的帮助是很大的。

本系列文章无关 zsh 的安装、使用、配置（如果需要配置文件，可以参考我的.zshrc，里边有比较详细的注释），更无 oh-my-zsh 相关内容，安装 zsh 后无需配置即可开始学习编写脚本。读者不需要有 bash 的基础（最好了解一些），但需要接触过任何一门编程语言，对编程的一些基础概念要有了解。

1.1 为什么用 zsh 写脚本

很多人对 zsh 的了解停留在界面漂亮、主题多、插件多、补全强等等，而对 zsh 的语言特性了解并不多。因为 zsh 基本兼容 bash，不少人使用 bash 语法写 zsh 脚本，或者偶尔使用一些 zsh 特有的小技巧，很难体会出 zsh 作为一门编程语言的强大之处。

另外有些人认为 bash 几乎在所有类 Unix 系统都有默认安装，而 zsh 往往要自己安装，为了通用性而用 bash 写脚本比较好。这个说法也有一定的道理，但并不是对所有开发者来说都有影响。如果是开源软件的开发者，为了避免洁癖用户因为不想安装他用不到的 zsh 而不使用自己的软件，而避免使用 zsh，是有一定道理的（但

现在 zsh 的用户量也有一定的积累了）。除此之外，自己平时写脚本、公司内部使用等多数场景，都是不需要考虑这个因素的。

如果在公司使用，还涉及其他因素。

第一个是 zsh 的部署成本。但因为多数情况都需要部署其他软件，甚至自己的脚本可以和 zsh 打包部署（去掉用不到的文件后的 zsh 只有 1M 多），所以基本不成问题。而且如果使用系统默认的 bash 的话，还涉及版本不同导致的问题，比如不同系统的 bash 版本不一样，或者系统升级后，bash 的升级导致之前的脚本挂掉等等。所以即使使用 bash，最好也是统一部署或者自带一个特定的版本，而不是使用系统默认的，以减少不必要的麻烦。

第二个就是非常重要的学习成本。因为会写 bash 的人很多，但会写 zsh 的比较少，如果只有自己会写，那么和别人合作会出问题。但 zsh 的学习成本并没有那么大，尤其是对会 bash 开发者来说，要大致看懂 zsh 脚本基本只需要几十分钟的学习，而编写的话，循序渐进也是很自然的事情，而且想不起来的时候还可以用 bash 的语法写。所以学习成本没有那么可观。

第三个是使用 zsh 开发的好处。如果 zsh 和 bash 相比，没有明显的好处，为什么要学习和使用它呢？那么就要从 bash 痛点讲起了。我想经常写 bash 脚本的人，很少有人会举大拇指说 bash 真好用啊。相反，我曾经多次听某些开发者说我写过一个超过 2000（或者其他行数）行的 shell（bash）脚本。但几乎没有人会认为写一个超过 2000 行的 Python 脚本是一件多么特别的事情。蹩脚的语法（几乎所有从任何其他语言迁移过来的开发者，都要重新熟悉和习惯它的语法）、严重依赖外部命令（因为文件系统错误等问题，挂掉一个外部命令，脚本就休克了。命令版本不同会有用法上的微秒差别，调试测试困难。频繁起新进程性能低下）、功能孱弱蹩脚（很多需要频繁使用的功能不全面或者不好用，比如字符串处理和数组的用法）等等，让很多开发者非常头疼，其中有些人甚至主张禁止使用 shell 脚本，一律改用 Python 等等，但 Python 并非适用所有场景，而且也有另外的一些问题，这样做也是因噎废食。Zsh 并非将这些问题全部解决了，但和 bash 相比，有很大的改善。比如 zsh 支持多种风格的语法，开发者很容易找到亲切感；对外部命令的依赖比 bash 要轻很多，多数常用的功能不需要使用外部命令，性能更好，调试也更加方便；功能上和 bash 相比也有比较大的提升，处理不那么复杂的场景已经比较够用了。

有人可能会说，不如 “一步到位”，使用 Powershell。Powershell 的确比 Python 更适合作为一种 shell 脚本语言，但使用它的话会有其他问题。

首先 Powershell 的学习成本是绝对要比 zsh 高的，如果想省点事，这并不是好的选择。

其次 Linux 下的 Powershell 目前还是 beta 版，以后会不会有很多人用也很难说，如果很少有人用，那么生态环境就成问题。比如遇到问题后找不到解决办法，配套的软件和库不完善等等。

再次 Powershell 解释器的启动速度非常感人，在我的机器上，Windows 下的 Powershell 空脚本要执行将近 200 毫秒，Linux 下的要更长一些（我只在 WSL 里安装试用过，时间翻了几倍），而 zsh 的话，在 Linux 下不超过 5 毫秒，在 WSL 下也不超过 20 毫秒。如果写一个简单的脚本，运行时都要卡一下，是非常影响体验的。

最后如果平时就使用 Powershell 作为交互 shell，那么虽然脚本的启动时间问题有所缓解，但用户体验会差很多，而且以后也很难提升上来，很容易得不偿失。

1.2 Zsh 脚本样例

可以通过一个例子直观感受下用 zsh 写的脚本。这是一个删除当前目录以及所有子目录下重复文件的脚本，通过 md5 判断文件是否相同（不严谨）。熟悉 bash 的读者可以尝试用 bash 完成相同的功能，然后对比一下代码（我之前写过一个 bash 版本的，不贴上来了），就能比较直观地感受到 bash 和 zsh 的区别了。

```
#!/bin/zsh

local files=("${(f)$(md5sum **/*(.D))}")
local files_to_delete=()
local -A md5s

for i ($files) {
    local md5=$i[1,32]

    if (($+md5s[$md5])) {
        files_to_delete+=($i[35,-1])
    } else {
        md5s[$md5]=1
    }
}

(($#files_to_delete)) && rm -v $files_to_delete
```

1.3 为什么要使用 shell 脚本语言

对于没有接触过 shell 脚本的开发者或者用户来说，有一个更重要的问题，我为什么要学习和使用 shell 脚本呢?

那么要从 shell 脚本的使用场景说起。Shell 是一种和计算机系统交互的文本界面（CLI），简单说就是输入命令后返回结果（也有比较复杂的操作）。CLI 在某些场景要比图形界面（GUI）方便和高效很多，是不可取代的（即使有一天语音识别取代了文本输入，CLI 也会换汤不换药地继续存在）。那么使用 CLI 就必须约定好指令格式，而 shell 脚本就是一种用于 CLI 交互的指令格式。

因为这个比较特别的场景，shell 脚本有一些与其他编程语言不同的特点。一个很重要的特点，shell 脚本要比较简洁，容易输入。如果发送一条简单指令就要打几十个字符，那恐怕谁也无法接受。而为了达到可以接受的简洁程度，shell 脚本的语法，往往比其他编程语言的更加怪异。

有人可能会说，这搞混了两个事情。在 CLI 输入命令和写脚本文件然后执行命令是两回事，不需要使用同一种语言，而只是在 CLI 交互中，通常是没有必要写复杂逻辑的，也就是说 shell 脚本基本没有必要学习。

是两回事不假，但二者并不是不相关的。比如有人这么想后，决定在 shell 里只使用最简单的命令，不学习较为复杂的语法，如果需要写脚本，就用 Python 之类的语言写。那么有什么问题吗?

Python 是为通用的场景设计的，虽然也能处理 shell 脚本所做的事情，但往往要写出多几倍甚至几十倍（如果对 Python 也不甚了解的话）的代码出来。而很多时候，shell 脚本做的是一次性工作，运行完就直接删除，或者直接在一行敲完，回车即可，这样的场景用 Python 写成本要高出很多。而且并不是一个 Python 初学者就能用 Python 实现 shell 脚本的功能的，甚至熟练的 Python 开发者也很可能一时想不好怎么实现某个用 shell 脚本能很容易实现的功能。Shell 脚本的很多工作是和字符串和目录文件打交道，特点是要实现的功能复杂多样，没有固定模式，无论用什么语言写，都不容易。Python 自带的字符串和目录文件等类库功能非常基础，基本只能实现功能很单一的操作，稍微复杂点的功能都需要自己写。如果去找某些功能复杂的第三方库，那就会涉及一堆问题，比如同样有学习和部署成本，可能因为用户少所以有 bug 未被发现，可能已经没有人维护了，Python 的语法决定库怎么写都不能让语法太简洁等等。

而初步熟悉一门 shell 脚本只需要几十分钟，用多了自然就熟悉了，成本收益的权衡不言而喻。

1.4 格式约定

文中行首的 `%` 代表 zsh 的命令提示符（类似 bash 的 `$`，这个是可以自由定义的，具体是什么不重要），行首的 `>` 代表此行是换行后的输入内容，以 `#` 开头的为注释（非 root 用户的命令提示符，本系列文章不需要 root 用户），其余的是命令的输出内容。另外某些地方会贴成段的 zsh 代码，那样就省略开头的 `%`，比较容易分辨。

一个样例：

```
# 前两行是输入内容，第三行是输出内容
% echo "Hello \
> World"
Hello World
```

本系列文章使用的 zsh 版本是 5.4.1（写这篇文章时的最新版本），代码在老版本中可能运行不了或者结果有出入，尽量使用最新版本。

下面直接进入正题。

1.5 变量

接触一门新的编程语言，运行完 Hello World 后，首先要了解的基本就是如何定义和使用变量了。有了变量后可以比较变量内容，进而可以接触条件、循环、分支等语句，继而了解函数的用法，更高级的数据结构的使用，更多库函数，等等。这样就大概了解了一门面向过程的语言的基本用法，剩下的可以等到用的时候再查手册。

所以这一篇讲最基本的变量和语句。

zsh 有 5 种变量：整数、浮点数（bash 不支持）、字符串、数组、哈希表（或者叫关联数组或者字典，本系列文章统一使用“哈希表”这一名词），另外还有一些其他语言少有的东西，比如 alias（但主要是交互时使用，编程时基本用不到）。此篇只涉及整数、浮点数、字符串，并且不涉及数值计算和字符串处理等内容。

1.5.1 变量定义

Zsh 的变量多数情况不需要提前声明或者指定类型，可以直接赋值和使用（但哈希表是一个例外）。

```
# 等号两端不能有空格
% num1=123
% num2=123.456
% str1=abcde
# 如果字符串中包含空格等特殊字符，需要加引号
% str2='abc def'
# 也可以用双引号，但和单引号有区别，比如双引号里可以使用变量，而单引号不可以
% str3="abc def $num1"
# 在字符串中可以使用转义字符，单双引号均可
% str4="abc\tdef\ng"

# 输出变量，也可以使用 print
% echo $str1
abcde

# 简单的数值计算
% num3=$(($num1 + $num2))
# (( 中的变量名可以不用 $
% num3=$((num1 + num2))

# 简单的字符串操作
% str=abcdef
# 2 和 4 都是字符在数组的位置，从 1 开始数，逗号两边不能有空格
% echo $str[2,4]
bcd
# -1 是最后一个字符
% echo $str[4,-1]
def
```

1.5.2 变量比较

```
# 比较数值
% num=123
# (( )) 用于数值比较等操作，如果为真返回 0，否则返回 1
# && 后边的语句在前边的语句为真时才执行
# 注意这里只能使用双等号来比较
```

(下页继续)

(续上页)

```
% ((num == 123)) && echo good
good
# (( 里边可以使用与 (&&) 或 (||) 非 (!) 操作符，同 c 系列语言
% ((num == 1 || num == 2)) && echo good

# 比较字符串
% str=abc
# 比较字符串要用 [[，内侧要有空格，[[ 的具体用法之后会讲到
# 这里双等号可以替换成单等号，可以根据自己的习惯选用
# 本系列文章统一使用双等号，因为和 (( )) 一致，并且使用双等号的常用编程语言更多些
# $str 两侧不需要加双引号，即使 str 未定义或者 $str 中含空格和特殊符号
% [[ $str == abc ]] && echo good
good
# 可以和空字符串 "" 比较，未定义的字符串和空字符串比较结果为真
# [[ 里也可以用 && || !
% [[ $str == "" || $str == 123 ]] && echo good
```

1.6 语句

稍微了解下简单变量的使用后，快速进入语句部分。

zsh 支持多种风格的语法，包括经典的 posix shell （bash 的语法和它类似，但有一些扩展，可以归为一类）的，以及 csh 风格的等等。但 posix shell 的语法并不好用，我们没必要一定使用这个。我只选用一种我认为最方便简洁的语法，没有 `fi`、`then`、`do`、`done`、`esac`、`in` 等的关键字（虽然其中某些关键字其他编程语言也有，但基本用法都各异，而且容易混淆），也不需要多余的分号。如果不确定语法是否符合预期，可以定义一个函数然后使用 `which` 查看，内容会被转化成原始（posix shell 风格）的样子。熟悉 bash 并且喜欢使用 bash 语法的读者可以跳过这部分内容，语法的不同并不影响后续内容的阅读，继续使用 bash 风格语法写 zsh 也是没有问题的。

1.6.1 条件语句

```
# 格式
if [[ ]] {
} elif {
} else {
}
```

大括号也可以另起一行，本系列文章统一使用这种风格，缩进为 4 个空格。注意 `elif` 不可写作 `else if`。

`[[ ]]` 用于比较字符串、判断文件等，功能比较复杂多样，这里先使用最基础的用法。注意尽量不要用 `[[`

]] 比较数值，因为不留神的话，数值会被转化成字符串来比较，没有任何错误提示，但结果可能不符合预期，导致不必要的麻烦。

```
# 样例
if [[ "$str" == "name" || "$str" == "value" ]] {
    echo "$str"
}
```

(( )) 用于比较数值，里边可以调用各种数值相关的函数，格式类似 c 语言，变量前的 $ 可省略。

```
# 格式
if (( )) {
}
```

```
# 样例
if ((num > 3 && num + 3 < 10)) {
    echo $num
}
```

{ } 用于在当前 shell 运行命令并且判断运行结果。

```
# 格式
if { } {
}
```

```
# 样例
if {grep sd1 /etc/fstab} {
    echo good
}
```

( ) 用于在子 shell 运行命令并且判断运行结果，用法和 {} 类似，不再举例。

```
# 格式
if ( ) {
}
```

这几种括号可以一起使用，这样可以同时判断字符串、数值、文件、命令结果等等。最好不要混合使用 && ||，会导致可读性变差和容易出错。

```
# 格式
if [[ ]] && (( )) && { } {
}
```

1.6.2 循环语句

```
# 格式
while [[ ]] {
    break/continue
}
```

和 if 一样，这里的 [[ ]] 可以替换成其他几种括号，功能也是一样的，不再依次举例。break 用于结束循环，continue 用于直接进入下一次循环。所有的循环语句中都可以使用 break 和 continue，下边不再赘述。

```
# 样例 死循环
 while ((1)) {
    echo good
}
```

until 和 while 相反，不满足条件时运行，一旦满足则停止，其他的用法和 while 相同，不再举例。

```
# 格式
until [[ ]] {
}
```

for 循环主要用于枚举，这里的括号是 for 的特有用法，不是在子 shell 执行。括号内是字符串（可放多个，空格隔开）、数组（可放多个）或者哈希表（可放多个，哈希表是枚举值而不是键）。i 是用于枚举内容的变量名，变量名随意。

```
# 格式
for i ( ) {
}
```

```
# 样例
for i (aa bb cc) {
    echo $i
}

# 枚举当前目录的 txt 文件
for i (*.txt) {
    echo $i
}

# 枚举数组
array=(aa bb cc)
for i ($array) {
```

(下页继续)

(续上页)

```
    echo $i
}
```

经典的 c 风格 for 循环。

```
# 格式
for (( ; ; )) {
}
```

```
# 样例
for ((i=0; i < 10; i++)) {
    echo $i
}
```

这个样例只是举例，实际上多数情况不需要使用这种 for 循环，可以这样。

```
# 样例，{1..10} 可以生成一个 1 到 10 的数组
for i ({1..10}) {
    echo $i
}
```

repeat 语句用于循环固定次数，n 是一个整数或者内容为整数的变量。

```
# 格式
repeat n {
}
```

```
# 样例
repeat 5 {
    echo good
}
```

1.6.3 分支语句

分支逻辑用 if 也可以实现，但 case 更适合这种场景，并且功能更强大。

```
# 格式 + 样例
case $i {
    (a)
    echo 1
```

(下页继续)

(续上页)

```
    ;;

    (b)
    echo 2
    # 继续执行下一个
    ;&

    (c)
    echo 3
    # 继续向下匹配
    ;|

    (c)
    echo 33
    ;;

    (d)
    echo 4
    ;;

    (*)
    echo other
    ;;
}
```

;; 代表结束 case 语句，;& 代表继续执行紧接着的下一个匹配的语句（不再进行匹配），;| 代表继续往下匹配看是否有满足条件的分支。

1.6.4 用户输入选择语句

select 语句是用于根据用户的选择决定分支的语句，语法和 for 语句差不多，如果不 break，会循环让用户选择。

```
# 格式
select i ( ) {
}
```

```
# 样例
select i (aa bb cc) {
    echo $i
```

(下页继续)

(续上页)

```
}
```

输出是这样的。

```
1) aa  2) bb  3) cc
?#
```

按上边的数字加回车来选择。

1.6.5 异常处理语句

```
# 格式
{
    语句 1
} always {
    语句 2
}
```

无论语句 1 是否出错，都执行语句 2。

1.6.6 简化的条件语句

`if` 语句的简化版，在只有一个分支的情况下更简洁，功能和 `if` 语句类似，不赘述。

```
格式:
[[ ]] || {
}

[[ ]] && {
}
```

最好不要连续混合使用 `&& ||`，比如。

```
aa && bb || cc && dd
```

容易导致逻辑错误或者误解，可以用 `{ }` 把语句包含起来。

```
aa && { bb || { cc && dd } }
```

比较复杂的判断还是用 `if` 可读写更好，`&& ||` 通常只适用于简单的场景。

1.7 总结

本篇简单介绍了变量和语句的使用方法。变量部分只涉及了最基础常用的部分，后续文章会详细介绍。语句部分已经覆盖了所有需要使用的语句，实际上这些语句都不只有这一种语法，但本系列文章统一使用这个语法。但涉及到的几种括号的用法比较复杂，之后的文章也会详细介绍。

CHAPTER 2

02 字符串处理之常用操作

字符串处理是 shell 脚本的重点部分，因为 shell 脚本主要的工作是和文件或者其他程序打交道，数据格式通常是文本，而处理没有统一格式的文本文件出奇地复杂，shell 命令中也有很多都是处理文本的。用 bash 处理文本的话，因为自身的功能有限，经常需要调用像 `awk`、`sed`、`grep`、`cat`、`cut`、`comm`、`dirname`、`basename`、`expr`、`sort`、`uniq`、`head`、`tail`、`tac`、`tr`、`wc` 这样命令，不留神脚本就成了命令大聚会。命令用法各异，有的很简单（比如 `cut`、`tr`、`wc`），看一眼 man 就会用；有的很复杂（比如 `awk`、`sed`、`grep`），用了好多年基本也只会用很少一部分功能。互相配合也容易出现各种各样的问题（比如要命的空格和换行符问题），难以调试，调用命令的开销也很大。而用好了 zsh 的话，可以大幅减少这些命令的使用（并不能完全避免，因为某些场景确实比较适合用这样的命令处理，比如处理一个大文本文件），并且大幅提升脚本的性能（主要因为减少了进程启动的开销，比如一次简单的字符串替换，调用外部命令实现比内部实现的时间要多好几个数量级）。

但也因此 zsh 的字符串处理功能很复杂，可以说 zsh 的字符串处理功能，要比绝大多数编程语言自带的字符串函数库或者类库要强大（在不依赖外部命令的情况）。同时各种用法也比较怪异，很多时候简洁性和可读性是有矛盾的，很难兼顾。而 shell 的使用场景决定简洁性是不能被牺牲掉的，即使用 Python 这样比较简洁的语言来处理字符串，很多时候也只能写出冗长的代码，而 zsh 经常可以一行搞定（可能有人想到了 Perl，Perl 在处理文本方面确实有比较明显的优势，但使用 Perl 的话也要承担更多的成本），如果再加上适当地使用外部命令，基本可以应付大多数字符串处理场景。因为字符串处理的内容比较丰富，我会分多篇文章写。本篇只涉及最基础和常用的字符串操作，包括字符串的拼接、切片、截断、查找、遍历、替换、匹配、大小写转换、分隔等等。

字符串定义和简单比较，我已经在前一篇文章提过了，现在直接进入正题。

2.1 字符串长度

```
% str=abcde
% echo $#str
5

# 读取函数或者脚本的第一个参数的长度
% echo $#1
```

2.2 字符串拼接

```
% str1=abc
% str2=def

% str2+=$str1
% echo $str2
defabc

% str3=$str1$str2
abcdefabc
```

2.3 字符串切片

字符串切片之前也提过，这里简单复习一下。逗号前后不能有空格。字符位置是从 1 开始算起的。

```
% str=abcdef
% echo $str[2,4]
bcd
% echo $str[2,-1]
bcdef

# $1 是文件或者函数的第一个参数
echo ${1[2,4]}
```

字符串切片还有另一种风格的方法，即 bash 风格，功能大同小异。通常没有必要用这个，而且因为字符位置是从 0 开始算，容易混淆。

```
% str=abcdef
% echo ${str:1:3}
bcd
% echo ${str:1:-1}
bcde
```

2.4 字符串截断

```
% str=abcdeabcde

# 删除左端匹配到的内容，最小匹配
% echo ${str#*b}
cdeabcde

# 删除右端匹配到的内容，最小匹配
% echo ${str%d*}
abcdeabc

# 删除左端匹配到的内容，最大匹配
% echo ${str##*b}
cde

# 删除右端匹配到的内容
% echo ${str%%d*}
abc
```

2.5 字符串查找

子字符串定位。

```
% str=abcdef

# 这里用的是 i 的大写，不是 L 的小写
% echo $str[(I)cd]
3

# I 是从右往左找，如果找不到则为 0，方便用来判断
% (($str[(I)cd])) && echo good
```

(下页继续)

(续上页)

```
good

# 找不到则为 0
% echo $str[(I)cdd]
0

# 也可以使用小 i，小 i 是从左往右找，找不到则返回数组大小 + 1
% echo $str[(i)cd]
3

% echo $str[(i)cdd]
7
```

2.6 遍历字符

```
% str=abcd

% for i ({1..$#str}) {
>    echo $str[i]
>}
a
b
c
d
```

2.7 字符串替换

按内容替换和删除字符。

```
% str=abcabc

# 只替换找到的第一个
% echo ${str/bc/ef}
aefabc

# 删除匹配到的第一个
% echo ${str/bc}
```

(下页继续)

(续上页)

```
aabc

# 替换所有找到的
% echo ${str//bc/ef}
aefaef

# 删除匹配到的所有的
% echo ${str//bc}
aa



% str=abcABCabcABCabc

# /# 只从字符串开头开始匹配，${str/#abc} 也同理
% echo ${str/#abc/123}
123ABCabcABCabc

# /% 只从字符串结尾开始匹配，echo ${str/%abc} 也同理
% echo ${str/%abc/123}
abcABCabcABC123



% str=abc
# 如果匹配到了则输出空字符串
% echo ${str:#ab*}

# 如果匹配不到，则输出原字符串
% echo ${str:#ab}
abc

# 加 (M) 后效果反转
% echo ${(M)str:#ab}
```

按位置删除字符。

```
%str=abcdef

# 删除指定位置字符
% str[1]=
% echo $str
```

(下页继续)

(续上页)

```
bcdef

# 可以删除多个
% str[2,4]=
% echo $str
bf
```

按位置替换字符。

```
% str=abcdefg

# 一对一地替换
% str[2]=1
% echo $str
a1cdefg

# 可以多对多（也包括一对多和多对一）地替换字符，两边的字符数量不需要一致。
# 把第二、三个字符替换成 2345
% str[2,3]=2345
% echo $str
a2345defg
```

2.8 判断字符串变量是否存在

如果用 `[[ "$strxx" == "" ]]`，那无法区分变量是没有定义还是内容为空，在某些情况是需要区分二者的。

```
% (($+strxx)) && echo good

% strxx=""
% (($+strxx)) && echo good
good
```

`(($+var))` 的用法也可以用来判断其他类型的变量，如果变量存在则返回真（0），否则返回假（1）。

2.9 字符串匹配判断

判断是否包含字符串。

```
% str1=abcd
% str2=bc

% [[ $str1 == *$str2* ]] && echo good
good
```

正则表达式匹配。

```
% str=abc55def

# 少量字符串的话，尽量不要用 grep
# 本文不讲正则表达式格式相关内容
# 另外 zsh 有专门的正则表达式模块
% [[ $str =~ "c[0-9]{2}\de" ]] && echo a
a
```

2.10 大小写转换

```
% str="ABCDE abcde"

# 转成大写，(U) 和 :u 两种用法效果一样
% echo ${(U)str} --- ${str:u}
ABCDE ABCDE --- ABCDE ABCDE

# 转成小写，(L) 和 :l 两种用法效果一样
% echo ${(L)str} --- ${str:l}
abcde abcde --- abcde abcde

# 转成首字母大写
% echo ${(C)str}
Abcde Abcde
```

2.11 目录文件名截取

```
% filepath=/a/b/c.x

# :h 是取目录名，即最后一个 / 之前的部分，如果没有 / 则为 .
```

(下页继续)

(续上页)

```
% echo ${filepath:h}
/a/b

# :t 是取文件名，即最后一个 / 之后的部分，如果没有 / 则为字符串本身
% echo ${filepath:t}
c.x

# :e 是取文件扩展名，即文件名中最后一个点之后的部分，如果没有点则为空
% echo ${filepath:e}
x

# :r 是去掉末尾扩展名的路径
% echo ${filepath:r}
/a/b/c
```

2.12 相对路径转绝对路径

```
# ${filepath:A} 功能相当于 $(readlink -f $filepath)
% pwd
/tmp/test
% ls -lF
-rw-r--r-- 1 goreliu goreliu  0 Feb 15 13:14 a.txt
lrwxrwxrwx 1 goreliu goreliu 11 Feb 15 13:16 b -> /usr/bin/ls*
% filepath1=a.txt
% filepath2=b
% echo ${filepath1:A} ${filepath2:A}
/tmp/test/a.txt /usr/bin/ls
```

2.13 字符串分隔

```
# 使用空格作为分隔符，多个空格也只算一个分隔符
% str='aa bb cc dd'
% echo ${str[(w)2]}
bb
% echo ${str[(w)3]}
cc
```

(下页继续)

(续上页)

```
# 指定分隔符
% str='aa--bb--cc'
# 如果分隔符是 : 就用别的字符作为左右界，比如 ws.:.
% echo ${str[(ws:--:)3]}
cc
```

```
# 或者先转换成数组
% str="1:2::4"
% str_array=(${(s/:/)str})
% echo $str_array
1 2 4
% echo $str_array[2]
2
% echo $str_array[3]
4

# 保留其中的空字符串
% str_array=("${(@s/:/)str}")
% echo $str_array[3]

% echo $str_array[4]
4
```

2.14 多行字符串

字符串定义可以跨行。

```
% str="line1
> line2"
% echo $str
line1
line2
```

2.15 读取文件内容到字符串

```
# 比用 str=$(cat filename) 性能好很多
str=$(<filename)

# 比用 cat filename 性能好很多，引号不能省略
echo "$(<filename)"

# 遍历每行，引号不能省略
for i (${(f)"$(<filename)"}) {
    echo $i
}
```

读取文件指定行。

文件 test.txt 内容如下：

```
line 1. apple
line 2. orange
```

```
# 小文件或者需要频繁调用时，尽量不要用 sed
% echo ${"$(<test.txt)"[(f)2]}
line 2. orange

# 输出包含 “ang” 的第一行
% echo ${"$(<test.txt)"[(fr)*ang*]}
line 2. orange

# 输出包含 pp 的第一行，但从左截掉 “line” 4 个字符。
echo ${"$(<test.txt)"[(fr)*pp*]#line}
```

2.16 读取进程输出到字符串

读进程输出和读文件类似。

上边字符串相关的处理，直接把 $(<test.txt) 换成 $(命令) 即可。如果一定需要一个文件名，可以这样。

```
# 返回 fd 路径，优先使用，但某些场景会出错
% wc -l <(ps)
4 /proc/self/fd/11
```

(下页继续)

(续上页)

```
# 临时文件，会自动删除，适合上边用法出错的情况
% wc -l =(ps)
3 /tmp/zshMWDpqD
```

2.17 参考

http://tim.vanwerkhoven.org/post/2012/10/28/ZSH/Bash-string-manipulation

CHAPTER 3

03 字符串处理之转义字符和格式化输出

上一篇讲了 zsh 的常用字符串操作，这篇开始讲更为琐碎的转义字符和格式化输出相关内容。包括转义字符、引号、`print`、`printf` 的使用等等。其中很多内容没有必要记忆，作为手册参考即可。

3.1 转义字符

转义字符是很多编程语言中都有的概念，它主要解决某些字符因为没有对应键盘按键无法直接输出、字符本身有特殊含义（比如 `\`、`"`）或者显示不直观（比如难以区别多个空格和一个 tab）等问题。

最常用的转义字符是 `\n`（换行）、`\r`（回车）、`\t`（tab）。

直接用 `echo`、`print` 或者 `printf` 内置命令都可以正常输出转义字符，但包括转义字符的字符串需要用引号（单双引号都可以）扩起来。

```
% echo 'Hello\n\tWorld'
Hello
        World
```

常用转义字符对照表，不常用的可以去查 ASCII 码表，然后使用 `\xnn`（如 `\x14`）。

可以用 `hexdump` 命令查看字符的 ASCII 码值。

```
% echo ab= | hexdump -C
00000000  61 62 3d 0a                                       |ab=.|
00000004
```

还有一些字符是可选转义（通常有特殊含义的字符都是如此）的，比如空格、"、'、*、~、$、&、(、)、[、]、{、}、;、? 等等，即如果在引号里边则无需转义（即使转义也不出错，转义方法都说前边加一个 \），但如果在引号外边则需要转义。谨慎起见，包含半角符号的字符串全部用引号包含即可，可以避免不必要的麻烦。

可以这样检查一个字符在空格外是否需要转义，输出的字符中前边带 \ 的都是需要的。

```
% str='~!@#$%^&*()_+-={}|[]:;<>?,./"'
# -r 选项代表忽略字符串中的转义符合
# ${(q)str} 功能是为字符串中的特殊符号添加转义符号
% print -r ${(q)str}
\~\!@\#\$%\^\&\*\(\)_+-=\{\}\|\[\]:\;\<\>\?,./\"
```

3.2 单引号

单引号的左右主要是为了避免字符串里的特殊字符起作用。在单引号中，只有一个字符需要转义，转义符号 \ 。所以如果字符串里包含特殊符号时，最好使用单引号包含起来，避免不必要的麻烦。如果字符串需要包含单引号，可以使用这几种方法。

```
# 用双引号包含
% echo "a'b"
a'b

# 用转义符号
% echo a\'b
a'b

# 同时使用单引号和转义符号，用于包含单引号和其他特殊符号的场景
% echo 'a"\'\''b*?'
a"\'b*?
```

3.3 双引号

双引号的作用类似单引号，但没有单引号那么严格，有些特殊字符在双引号里可以继续起作用。

```
# 以使用变量
% str=abc
% echo "$str"
abc

# 可以使用 $( ) 运行命令
```

(下页继续)

(续上页)

```
% echo "$(ls)"
git
tmp

# 可以使用 ` ` 运行命令，不建议在脚本里使用 ` `
% echo "`date`"
Mon Aug 28 09:49:11 CST 2017

# 可以使用 $(( )) 计算数值
% echo "$((1 + 2))"
3

# 可以使用 $[ ] 计算数值
% echo "$[1 + 2]"
3
```

简单说，$ 加各种东西的用法在双引号里都是可以正常使用的，而其他特殊符号（比如 *、?、>）的功能通常不可用。

3.4 反引号

反引号是用来运行命令的，它会返回命令结果，以便保存到变量等等。

```
% str=`ls`
% echo $str
git
tmp

# 完全可以用 $( ) 取代
% str=$(ls)
% echo $str
git
tmp
```

反引号的功能和 $( ) 功能基本一样，但 $( ) 可以嵌套，而反引号不可以，而且反引号看起来更费事，某些字体中的反引号和单引号差别不大。所以在脚本里不建议使用反引号。

3.5 print 命令用法

print 是类似 echo 的内部命令（echo 命令很简单，不作介绍），但功能比 echo 强大很多。完全可以使用 print 代替 echo。

不加参数的 print 和 echo 的功能基本一样，但如果字符串里包含转义字符，某些情况可能不一致。如果需要输出转义字符，尽量统一使用 print，避免不一致导致的麻烦。

```
% print 'Line\tone\n\Line\ttwo'
Line    one
Line    two

# echo 的输出和 print 不一致
% echo 'Line\tone\n\Line\ttwo'
Line    one
\Line   two
```

print 有很多参数，在 zsh 里输入 print - 然后按 tab 即可查看选项帮助（如果没有效果，需要配置 ~/.zshrc 里的补全选项，网上有很多现成的配置）。

```
# - 后直接按 tab, C 是补全上去的
% print -C
 -- option --
-C  -- print arguments in specified number of columns
-D  -- substitute any arguments which are named directories using ~ notation
-N  -- print arguments separated and terminated by nulls
...
```

3.6 print 命令选项功能介绍

这里以常用程度的顺序依次介绍所有的选项，另外文末有“print 选项列表”方便查询。

-l 用于分行输出字符串：

```
# 每个字符串一行，字符串列表是用空格隔开的
% print -l aa bb
aa
bb

# 也可以接数组，数组相关的内容之后会讲到
# 命令后的多个字符串都可以用数组取代，效果是相同的
```

(下页继续)

(续上页)

```
% array=(aa bb)
% print -l $array
aa
bb
```

-n 用于不在输出内容的末尾自动添加换行符（echo 命令也有这个用法）：

```
% print abc
abc
# 下面输出 abc 后的 % 高亮显示，代表这一行末尾没有换行符
% print -n abc
abc%
```

-m 用于只输出匹配到的字符串：

```
% print -m "aa*" aabb abc aac
aabb aac
```

-o/-O/-i 用于对字符串排序：

```
# print -o 对字符串升序排列
% print -o a d c 1 b g 3 s
1 3 a b c d g s

# print -O 对字符串降序排列
% print -O a d c 1 b g 3 s
s g d c b a 3 1

# 加 -i 参数后，对大小写不敏感
% print -oi A B C a c A B C
A a A B B C c C

# 不加 -i 的话小写排在大写的前面
% print -o A B C a c A B C
a A A B B c C C
```

-r 用于不对字符串进行转义。print 默认是会对转义字符进行转义的，加 -r 后会原样输出：

```
% print -r '\n'
\n
```

-c 用于将字符串按列输出。如果对自动决定的列数不满意，可以用 -C 指定列数：

```
% print -c a bbbbb ccc ddddd ee ffffff gg hhhhhh ii jj kk
a       ccc     ee      gg      ii      kk
bbbbb   ddddd   ffffff  hhhhhh  jj
```

-C 用于按指定列数输出字符串：

```
# 从上到下
% print -C 3 a bb ccc dddd ee f
a     ccc   ee
bb    dddd  f

% print -C 3 a bb ccc dddd ee f g
a     dddd  g
bb    ee
ccc   f

# 加 -a 后，改成从左向右
% print -a -C 3 a bb ccc dddd ee f g
a     bb    ccc
dddd  ee    f
g
```

-D 用于将符合条件的路径名转化成带 ~ 的格式（~ 是家目录）：

```
% print -D /home/goreliu/git
~/git

# mine 是这样定义的 hash -d mine='/mnt/c/mine'
% print -D /mnt/c/mine
~mine
```

-N 用于将输出的字符串以 \x00（null）分隔，而不是空格。这样可能方便处理包含空格的字符串，xargs 等命令也可以接受以 \x00 分隔的字符串：

```
% print -N aa bb cc
aabbcc%

% print -N aa bb cc | hexdump -C
00000000  61 61 00 62 62 00 63 63  00                       |aa.bb.cc.|
00000009
```

-x 用于将行首的 tab 替换成空格。-x 是将行首的 tab 展开成空格，-x 后的参数是一个 tab 对应的空格数：

```
% print -x 2 '\t\tabc' | hexdump -C
00000000  20 20 20 20 61 62 63 0a                           |    abc.|
00000008

% print -x 4 '\t\tabc' | hexdump -C
00000000  20 20 20 20 20 20 20 20  61 62 63 0a              |        abc.|
0000000c
```

-X 用于将所有的 tab 补全成空格。注意不是简单地替换成空格。比如每行有一个 tab，-X 8，那么如果 tab 前（到行首或者上一个 tab）有 5 个字符，就补全 3 个空格，凑够 8，这么做是为了对齐每一列的。但如果前边有 8 个或者 8 个以上字符，那么依然是一个 tab 替换成 8 个字符，因为 tab 不能凭空消失，一定要转成至少一个空格才行。如果没理解就自己多试试找规律吧。

```
% print -X 2 'ab\t\tabc' | hexdump -C
00000000  61 62 20 20 20 20 61 62  63 0a                    |ab    abc.|
0000000a

% print -X 4 'ab\t\tabc' | hexdump -C
00000000  61 62 20 20 20 20 20 20  61 62 63 0a              |ab      abc.|
0000000c
```

-u 用于指定文件描述符（fd）输出。print 默认输出到 fd 1，即 stdout，可以指定成其他 fd（2 是 stderr，其他的可以运行 ls -l /proc/$$/fd 查看。

```
% print -u 2 good
good

# 和重定向输出效果一样
% print good >&2
```

-v 用于把输出内容保存到变量：

```
# 和 str="$(print aa bb cc)" 效果一样
% print -v str aa bb cc
% echo $str
aa bb cc
```

-s/-S 用于把字符串保存到历史记录：

```
% print -s ls -a
% history | tail -n 1
 2222  ls -a
```

(下页继续)

(续上页)

```
# -S 也类似，但需要用引号把命令引起来
% print -S "ls -a"
% history | tail -n 1
 2339  ls -a
```

-z 用于把字符串输出到命令行编辑区：

```
# _ 是光标位置
% print -z aa bb cc
% aa bb cc_
```

-f 用于按指定格式化字符串输出，同 printf，用法见“printf 命令用法”。

-P 用于输出带颜色和特殊样式的字符串，见“输出带颜色和特殊样式的字符串”。

-b 用于辨认出 bindkey 中的转义字符串，bindkey 是 Zle 的快捷键配置内容，写脚本用不到，不作介绍。

-R 用于模拟 echo 命令，只支持 -n 和 -e 选项，通常用不到。

3.7 printf 命令用法

printf 命令很像 c 语言的 printf 函数，用于输出格式化后的字符串：

```
# 末尾输出高亮的 % 代表该行末尾没有换行符
# printf 不会在输出末尾自动添加换行符
# 为了避免误解，之后的例子省略该 % 符号
% printf ":%d %f:" 12 34.56
:12 34.560000:%
```

printf 的第一个参数是格式化字符串，在 zsh 里输入 printf % 后按 tab，可以看到所有支持的用法。下面只举几个比较常用的例子：

```
# 整数 浮点数 字符串
% printf "%d %f %s" 12 12.34 abcd
12 12.340000 abcd%

# 取小数点后 1 位
% printf "%.1f" 12.34
12.3

# 科学计数法输出浮点数
```

(下页继续)

(续上页)

```
% printf "%e" 12.34
1.234000e+01

# 将十进制数字转成十六进制输出
% printf "%x" 12
c

# 补齐空格或者补齐 0
% printf "%5d\n%05d" 12 12
   12
00012
```

我把完整的格式贴在这里，方便搜索：

```
 -- print format specifier --
      -- leave one space in front of positive number from signed conversion
-     -- left adjust result
.     -- precision
'     -- thousand separators
*     -- field width in next argument
#     -- alternate form
%     -- a percent sign
+     -- always place sign before a number from signed conversion
0     -- zero pad to length
b     -- as %s but interpret escape sequences in argument
c     -- print the first character of the argument
E  e  -- double number in scientific notation
f     -- double number
G  g  -- double number as %f or %e depending on size
i  d  -- signed decimal number or with leading " numeric value of following character
n     -- store number of printed bytes in parameter specified by argument
o     -- unsigned octal number
q     -- as %s but shell quote result
s     -- print the argument as a string
u     -- unsigned decimal number
X  x  -- unsigned hexadecimal number, letters capitalized as x
```

3.8 输出带颜色和特殊样式的字符串

用 zsh 的 `print -P` 可以方便地输出带颜色和特殊样式的字符串，不用再和 `\033[41;36;1m` 之类莫名其妙的字符串打交道了。

```
# %B 加粗 %b 取消加粗
# %F{red} 前景色 %f 取消前景色
# %K{red} 背景色 %k 取消背景色
# %U 下滑线 %u 取消下滑线
# %S 反色 %s 取消反色
#
# black or 0  red     or 1
# green or 2  yellow  or 3
# blue  or 4  magenta or 5
# cyan  or 6  white   or 7

# 显示加粗的红色 abc
% print -P '%B%F{red}abc'
abc

# 没覆盖到的功能可以用原始的转义符号，可读性比较差
# 4[0-7] 背景色
# 3[0-7] 前景色
# 0m 正常 1m 加粗 2m 变灰 3m 斜体 4m 下滑线 5m 闪烁 6m 快速闪烁 7m 反色

# 显示闪烁的红底绿字 abc
% print "\033[41;32;5mabc\033[0m"
abc
```

3.9 print 选项列表

为了方便查询，我把 `print` 的选项列表放在这里：

3.10 参考

http://zsh.sourceforge.net/Guide/zshguide05.html

CHAPTER 4

04 字符串处理之通配符

这是字符串处理系列的第三篇文章。前两篇基本覆盖了字符串处理中的常用操作，但在字符串匹配方面，没有详细展开。

通配符（glob）是 shell 中的一个比较重要的概念，可以认为是正则表达式的简化版本。通配符在字符串匹配和文件名搜索等方面非常有用。本篇只讲它在字符串匹配上的用法。

4.1 通配符的基本用法

之前在讲字符串匹配判断时，通配符出现过，就是 `*$str*` 两边的星号。

```
% str1=abcd
% str2=bc

# 星号要在引号外边
% [[ $str1 == *$str2* ]] && echo good
good

# 注意带通配符的字符串必须放在右边
% [[ *$str2* == $str1 ]] && echo good
```

星号是最常用的通配符，用于匹配任意数量（包括 0 个）的任意字符。

```
# 问号用于匹配一个任意字符
% [[ abcd == ab?? ]] && echo good
good

# 中括号用于匹配出现在其中的单个字符
% [[ abcd == abc[bcd] ]] && echo good
good

# 如果中括号里第一个字符是 ^，则匹配除了除了中括号里的单个字符
% [[ abcd == abc[^de] ]] && echo good
% [[ abcd == abc[^ce] ]] && echo good
good

# 中括号里可以指定字符的范围
% [[ a4 == [a-b][2-5] ]] && echo good
good

# 可以指定多个字符范围，并且可以掺杂其他字符
% [[ B4 == [a-cdddA-B][2-5] ]] && echo good
good

# 尖括号用于匹配一定范围的单个整数
% [[ 123 == 12<3-4> ]] && echo good
good

# 可以匹配整个整数
% [[ 123 == <100-200> ]] && echo good
good

# 可以没有上下界，默认的下界是 0，上界是正无穷
% [[ 123 == <100-> && 123 == <-200> ]] && echo good
good

# 可以上下界都没有，那么会匹配任意正整数和 0
# 这个可以用来判断字符串是否构成整数
# [[ 123 == <-> ]] && echo good
good

# ( 1 | 2 | ... ) 用于同时判断多个条件，满足一个即可
%  [[ ab == (aa|ab) ]] && echo good
```

(下页继续)

(续上页)

```
good

# 如果中括号里要用 - 或者 ^，放在最后即可，不需要转义
% [[ -^3 == [a-c-][3^-][3^-] ]] && echo good
good
```

以上是通配符的基本用法，总结一下。

4.2 加强版通配符

Zsh 还支持加强版通配符，功能更多一些。如果使用加强版的通配符，需要先在代码里加上 `setopt EXTENDED_GLOB`。

此外还有一些更高级的用法，暂时先略过。

4.3 总结

字符串的内容先告一段落，但之后的文章依然会不断地涉及字符串，因为数组和哈希表里的内容通常是字符串，处理目录文件时也涉及大量的字符串操作等等，届时会有新的字符串处理方法。此外，如果我发现新的处理字符串的方法或者技巧，也会更新这几篇文章。

4.4 参考

http://www.bash2zsh.com/zsh_refcard/refcard.pdf

CHAPTER 5

05 数组

了解完结构比较简单的字符串后，我们来看更复杂一些的数组。其实字符串在 zsh 中也可以当字符数组操作，但很少有需要把字符串当数组来处理的场景。本篇中主要讲的是字符串数组，复杂度要比单个字符串高一些。

在实际的脚本编写中，较少需要处理单个的字符串。往往需要处理从各个地方过来的大量文本，不可避免会用到数组。用好数组，会让文本处理工作事半功倍。

本篇只涉及数组的基础用法。

5.1 数组定义

数组可以直接赋值使用，不需要提前声明。等号和小括号之间不能有空格，小括号中的元素以空格隔开。

```
% array=(a bc ccc dddd)
# 用 $array 即可访问数组全部元素，输出时元素以空格分隔
% echo $array
a bc ccc dddd

# 使用 print -l 可以每行输出一个元素
% print -l $array
a
bc
ccc
dddd
```

(下页继续)

(续上页)

```
# 输出数组中的元素个数，用法和取字符串长度一样
% echo $#array
4

# 包含带空格的字符串
% array=(a "bc ccc" dddd)
% print -l $array
a
bc ccc
dddd

# 可以换行赋值，但如果行中间有空格，依然需要加引号
% array=(
> a
> bb
> "c c c"
> dddd
> )
```

5.2 元素读写

```
% array=(a bc ccc dddd)

# 用法和取字符串的第几个字符一样，从 1 开始算
% echo $array[3]
ccc
# -1 依然是最后一个元素，-2 是倒数第二个，以此类推
% echo $array[-1]
dddd

% array[3]=CCC

# 如果赋值的内容是一个空的小括号，则删除该元素
% array[2]=()

% print -l $array
a
```

(下页继续)

(续上页)

```
CCC
dddd

# 用 += 为数组添加一个新元素
% array+=eeeee
% print -l $array
a
CCC
dddd
eeeee

# 用 unset 可以删除整个数组
% unset array

# array 变量变成未定义状态
% echo $+array
0
```

5.3 数组拼接

```
% array1=(a b c d)
% array2=(1 2 3 4)

# 用 += 拼接数组
% array1+=(e f g)
% echo $array1
a b c d e f g

# 拼接另一个数组，小括号不可以省略，否则 array1 会被转成一个字符串
% array2+=($array1)
% echo $#array2
11

# 去掉小扩号后，array1 被转成了一个字符串
% array2+=$array1
% echo $#array2
12
% echo $array2[12]
```

(下页继续)

(续上页)

```
a b c d e f g


# 字符串可以直接拼接数组而转化成数组
% str=abcd
% str+=(1234)

% echo $#str
2
```

5.4 数组遍历

```
% array1=(a bb ccc dddd)
% array2=(1 2 3)

# 用 for 可以直接遍历数组，小括号不可省略
% for i ($array1) {
> echo $i
> }
a
bb
ccc
dddd

# 小括号里可以放多个数组，依次遍历
% for i ($array1 $array2) {
> echo $i
> }
a
bb
ccc
dddd
1
2
3
```

5.5 数组切片

数组切片和字符串切片操作方法完全相同。

```
% array=(a bb ccc dddd)

% echo $array[2,3]
bb ccc

# 依然可以多对多地替换元素
% array[3,-1]=(1 2 3 4)
% echo $array
a bb 1 2 3 4

# 也可以使用另一种语法，不建议使用
% echo ${array:0:3}
a bb 1
```

5.6 元素查找

数组的元素查找方法，和字符串的子字符串查找语法一样。

```
% array=(a bb ccc dddd ccc)

# 用小 i 输出从左到右第一次匹配到的元素位置
% echo $array[(i)ccc]
3

# 如果找不到，返回数组大小 + 1
% echo $array[(i)xxx]
6

# 用大 I 输出从右到左第一次匹配到的元素位置
% echo $array[(I)ccc]
5

# 如果找不到，返回 0
% echo $array[(I)xxx]
0
```

(下页继续)

(续上页)

```
# 可以用大 I 判断是否存在元素
% (($array[(I)dddd])) && echo good
good

% (($array[(I)xxx])) && echo good


% array=(aaa bbb aab bbc)
# n:2: 从指定的位置开始查找
% echo ${array[(in:2:)aa*]}
3
```

5.7 元素排序

```
% array=(aa CCC b DD e 000 AA 3 aa 22)

# 用小写字母 o 升序排列，从小到大
% echo ${(o)array}
000 22 3 aa aa AA b CCC DD e

# 用大写字母 O 降序排列，从大到小
% echo ${(O)array}
e DD CCC b AA aa aa 3 22 000

# 加 i 的话大小写不敏感
% echo ${(oi)array}
000 22 3 aa AA aa b CCC DD e


% array=(cc aaa b 12 115 90)
# 加 n 的话按数字大小顺序排
% echo ${(on)array}
12 90 115 aaa b cc

# Oa 用于反转数组元素的排列顺序
% echo ${(Oa)array}
90 115 12 b aaa cc
```

5.8 去除重复元素

```
% array=(ddd a bb a ccc bb ddd)

% echo ${(u)array}
ddd a bb ccc
```

5.9 使用连续字符或者数值构造数组

```
# 大括号中的逗号分隔的字符串会被展开
% array=(aa{bb,cc,11}) && echo $array
aabb aacc aa11

# .. 会将前后的数组连续展开
% array=(aa{1..3}) && echo $array
aa1 aa2 aa3

# 第二个 .. 后的数字是展开的间隔
% array=(aa{15..19..2}) && echo $array
aa15 aa17 aa19

# 也可以从大到小展开
% array=(aa{19..15..2}) && echo $array
aa19 aa17 aa15

# 可以添加一个或多个前导 0
% array=(aa{01..03}) && echo $array
aa01 aa02 aa03

# 单个字母也可以像数值那样展开，多个字母不行
% array=(aa{a..c}) && echo $array
aaa aab aac

# 字母是按 ASCII 码的顺序展开的
% array=(aa{Y..c}) && echo $array
aaY aaZ aa[ aa\ aa] aa^ aa_ aa` aaa aab aac
```

(下页继续)

(续上页)

```
# 这些用法都可以用在 for 循环里边
% for i (aa{a..c}) {
> echo $i
> }
aaa
aab
aac
```

5.10 从字符串构造数组

```
% str="a bb ccc dddd"

# ${=str} 可以将 str 内容按空格切分成数组
% array=(${=str})
% print -l $array[2,3]
bb
ccc


% str="a:bb:ccc:dddd"
# 如果是其他分隔符，可以设置 IFS 环境变量指定
% IFS=:
% array=(${=str})
% print -l $array[2,3]
bb
ccc


% str="a\nbb\nccc\ndddd"
# 如果是其他分隔符，也可以用 (s:x:) 指定
% array=(${(s:\n:)str})
% print -l $array[2,3]
bb
ccc


% str="a##bb##ccc##dddd"
# 分隔符可以是多个字符
```

(下页继续)

(续上页)

```
% array=(${(s:##:)str})
% print -l $array[2,3]
bb
ccc


% str="a:bb:ccc:dddd"
# 如果分隔符是 :, 可以 (s.:.)
% array=(${(s.:.)str})
% print -l $array[2,3]
bb
ccc
```

5.11 从文件构造数组

test.txt 内容。

```
a
bb
ccc
dddd
```

每行一个元素。

```
# f 的功能是将字符串以换行符分隔成数组
# 双引号不可省略, 不然会变成一个字符串, 引号也可以加在 ${ } 上
% array=(${(f)"$(<test.txt)"})
% print -l $array
a
bb
ccc
dddd

# 不加引号的效果
% array=(${(f)$(<test.txt)})
% print -l $array
a bb ccc dddd
```

(下页继续)

(续上页)

```
# 从文件构造数组，并将每行按分隔符 : 分隔后输出所有列
for i (${(f)"$(<test.txt)"}) {
    array=(${(s.:.)i})
    echo $array[1,-1]
}
```

5.12 从文件列表构造数组

```
# 这里的 * 即上一篇讲的通配符，所有的用法都可以在这里使用。
% array=(/usr/bin/vim*)
% print -l $array
/usr/bin/vim
/usr/bin/vimdiff
/usr/bin/vimtutor

# 要比 ls /usr/bin/[a-b]?? | wc -l 快很多
% array=(/usr/bin/[a-b]??) && print $#array
3
```

5.13 数组交集差集

```
% array1=(1 2 3)
% array2=(1 2 4)

# 两个数组的交集，只输出两个数组都有的元素
% echo ${array1:*array2}
1 2

# 两个数组的差集，只输出 array1 中有，而 array2 中没有的元素
% echo ${array1:|array2}
3

# 如果有重复元素，不会去重
% array1=(1 1 2 3 3)
% array2=(4 4 1 1 2 2)
% echo ${array1:*array2}
1 1 2
```

5.14 数组交叉合并

```
% array1=(a b c d)
% array2=(1 2 3)

# 从 array1 取一个，再从 array2 取一个，以此类推，一个数组取完了就结束
% echo ${array1:^array2}
a 1 b 2 c 3

# 如果用 :^^，只有一个数组取完了的话，继续从头取，直到第二个数组也取完了
% echo ${array1:^^array2}
a 1 b 2 c 3 d 1
```

5.15 对数组中的字符串进行统一的处理

一些处理字符串的方法（主要是各种形式的截取、替换、转换等等），也可以用在数组上，效果是对数组中所有元素统一处理。

```
% array=(/a/b.htm /a/c /a/b/c.txt)

# :t 是取字符串中的文件名，可以用在数组上，取所有元素的文件名
% print -l ${array:t}
b.htm
c
c.txt

# :e 是取扩展名，如果没有没有扩展名，结果数组中不会添加空字符串
% print -l ${array:e}
htm
txt

# 字符串替换等操作也可以对数组使用，替换所有字符串
% print -l ${array/a/j}
/j/b.txt
/j/c
/j/b/c.txt
```

:# 也可以在数组上用，但更实用一些。

```
% array=(aaa bbb ccc)

# :# 是排除匹配到的元素，类似 grep -v
% print ${array:#a*}
bbb ccc

# 前边加 (M)，是反转后边的效果，即只输出匹配到的元素，类似 grep
% print ${(M)array:#a*}
aaa

# 多个操作可以同时进行，(U) 是把字符串转成大写字母
% print ${(UM)array:#a*}
AAA
```

5.16 总结

本篇讲的是数组的基础用法，还有很多复杂的操作方法，以后会提到。

5.17 参考

http://zshwiki.org/home/scripting/array

http://www.bash2zsh.com/zsh_refcard/refcard.pdf

5.18 更新历史

20170830：增加“使用连续字符或者数值构造数组”。

20170909：修正“从字符串构造数组”中的错误。

20170910：增加“从字符串构造数组”中的部分内容。

CHAPTER 6

06 哈希表

哈希表是比数组更复杂的数据结构，在某些语言里被称作关联数组或者字典等等。简单说，哈希表用于存放指定键（key）对应的值（value），键和值的关系，就像字典中单词和释义的对应关系，通过单词可以快速找到释义，而不需要从头依次遍历匹配。准确地说，哈希表只是该功能的一种实现方式，也可以使用各种树或者其他数据结构来实现，不同的实现方式适合不同的场景，使用方法是一样的。但为了简化概念，统一使用哈希表这个名称。

6.1 哈希表定义

和其他变量类型不同，哈希表是需要提前声明的，因为哈希表的赋值语法和数组一样，如果不声明，是无法区分的。

```
% typeset -A table
# 或者用 local，二者功能是一样的
% local -A table

# 赋值的语法和数组一样，但顺序依次是键、值、键、值
% table=(k1 v1 k2 v2)

# 直接用 echo 只能输出值
% echo $table
v1 v2
```

(下页继续)

(续上页)

```
# 使用 (kv) 同时输出键和值，(kv) 会把键和值都放到同一个数组里
% echo ${(kv)table}
k1 v1 k2 v2

# 哈希表的大小是键值对的数量
% echo $#table
2
```

6.2 元素读写

读写哈希表的方法和数组类似，只是用于定位的数字变成了字符串。

```
# 可以声明和赋值写到一行
% local -A table=(k1 v1 k2 v2 k3 v3)
% echo $table[k2]
v2

% table[k2]="V2"

# 删除元素的方法和数组不同，引号不能省略
% unset "table[k1]"
% echo ${(kv)table}
k2 V2 k3 v3
```

6.3 哈希表拼接

```
# 追加元素的方法和数组一样
% table+=(k4 v4 k5 v5)
% echo $table
V2 v3 v4 v5



% local -A table1 table2
% table1=(k1 v1 k2 v2)
% table2=(k2 v222 k3 v3)

# 拼接哈希表，要展开成数组再追加
```

(下页继续)

(续上页)

```
% table1+=(${(kv)table2})
# 如果键重复，会直接替换值，哈希表的键是不重复的
% echo ${(kv)table1}
k1 v1 k2 v222 k3 v3
```

6.4 哈希表遍历

用 (kv) (k) 等先将哈希表转化成数组，然后再遍历。

```
% local -A table=(k1 v1 k2 v2 k3 v3)

# 只遍历值
% for i ($table) {
> echo $i
> }
v1
v2
v3

# 只遍历键
% for i (${(k)table}) {
> echo $i
> }
k1
k2
k3

# 同时遍历键和值
% for k v (${(kv)table}) {
> echo "$k -> $v"
> }
k1 -> v1
k2 -> v2
k3 -> v3
```

6.5 元素查找

判断键是否存在。

```
% local -A table=(k1 v1 k2 v2 k3 v3)
% (($+table[k1])) && echo good
good
% (($+table[k4])) && echo good
```

如果需要判断某个值是否存在，直接对值的数组判断即可。但这样做就体现不出哈希表的优势了。

```
% local -A table=(k1 v1 k2 v2 k3 v3)
# value 是值的数组，也可以用 local -a 强行声明为数组
% value=($table)

% (( $value[(I)v1] )) && echo good
good
% (( $value[(I)v4] )) && echo good
```

6.6 元素排序

对哈希表元素排序的方法，和数组类似，多了 k v 两个选项，其余的选项如 o（升序）、O（降序）、n（按数字大小）、i（忽略大小写）等通用，不再一一举例。

```
% local -A table=(aa 33 cc 11 bb 22)

# 只对值排序
% echo ${(o)table}
11 22 33

# 只对键排序
% echo ${(ok)table}
aa bb cc

# 键值放在一起排序
% echo ${(okv)table}
11 22 33 aa bb cc
```

6.7 从字符串、文件构造哈希表

因为哈希表可以从数组构造，所以从字符串、文件构造哈希表，和数组的操作是一样的，不再一一举例。

```
% str="k1 v1 k2 v2 k3 v3"
% local -A table=(${=str})
% echo $table
v1 v2 v3
```

6.8 对哈希表中的每个元素统一处理

对哈希表中的每个元素统一处理，和对数组的操作是类似的，多了 k v 两个选项用于指定是对键处理还是对值处理，可以一起处理。不再一一举例。

```
% local -A table=(k1 v1 k2 v2 k3 v3)
% print ${(U)table}
V1 V2 V3

% print ${(Uk)table}
K1 K2 K3

% print ${(Ukv)table}
K1 V1 K2 V2 K3 V3
```

:# 也可以在哈希表上用。

```
% local -A table=(k1 v1 k2 v2 k3 v3)

# 排除匹配到的值
% echo ${table:#v1}
v2 v3

# 只输出匹配到的键
% echo ${(Mk)table:#k[1-2]}
k1 k2
```

6.9 多维哈希表

Zsh 并不支持多维哈希表以及多维数组，但可以通过一些方法来模拟，以实现一部分功能。

6.9.1 用一维哈希表模拟多维哈希表

```
% local -A table
# 这里用 , 作为分隔符，也可以用其他符号。
% table[1,1]=a
% table[1,2]=b
% table[k,v]=c
% echo $table[1,1] $table[1,2] $table[k,v]
a b c
```

好处：使用方便，而且支持的维数不受限制。

坏处：功能太单一，比如不能对 table[1] 进行处理。

6.9.2 用字符串分割访问来模拟多维哈希表

```
% local -A table
# 分隔符为空格
% table[1]='a b'
% table[2]='c d'
% print -l $table[1] ${table[1][(w)2]} ${table[2][(w)1]}
a b
b
c

# 分隔符不是空格
% table[a]='aa,bb'
% table[b]='cc,dd'
% print -l $table[a] ${table[a][(ws:,:)2]} ${table[b][(ws:,:)1]}
aa,bb
bb
cc
```

好处：可以对 table[1] 进行处理。

坏处：不大方便，性能也不好。而且功能同样受限，比如第一维只能是数组，不能是哈希表。可以支持更多维，但需要再增加新的分隔符，使用起来更麻烦。

6.10 总结

本篇简单讲了哈希表的基本用法。篇幅不长，但因为哈希表的操作和数组类似，很多操作数组的方法都可以用作哈希表上，而且可以把键或者值单独作为数组处理，所以操作哈希表更为复杂一些。

另外还有一些更进阶的处理数组和哈希表方法，之后会讲到。

CHAPTER 7

07 数值计算

数值计算并非 zsh 的强项，但应付一些简单的场景还是没问题的。并且 zsh 提供一个数值计算库，里边有一些比较常用的数学函数。

7.1 整数和浮点数类型

Zsh 中通常不用指定变量类型，但也可以指定。对数值计算来说，区分整数和浮点数是很重要的，不指定变量类型会带来不方便。

```
# 整数
% integer i=123
# (t) 用于输出变量类型
% echo ${(t)i}
integer

# 浮点数
% float f=123.456
% echo ${(t)f}
float

# 注意一旦指定了变量类型，类型就不会变了，除非再重新指定其他类型，或者用 unset 删除掉
# 如果把浮点数赋值给整数变量，会取整
% i=12.34
```

(下页继续)

(续上页)

```
% echo $i
12
% a=-12.34
% echo $a
-12

# 整数是 64 位的带符号整数 (在 32 位系统下也是)
% echo $((-2 ** 63)) $((2 ** 63 - 1))
-9223372036854775808 9223372036854775807

# 浮点数是 64 位带符号浮点数 (在 32 位系统下也是)
% echo $((-1.79e-308)) $((1.79e308))
-1.79e-308 1.79e+308
```

7.2 运算符

数值计算主要是在 (( )) 或者 $(( )) 中进行的，在 $[ ] 或者 $var[ ] (可用于数组索引的计算) 中也能进行一部分，这里统一使用小括号。

```
% integer i=123
% float f=123.456

# $(( )) 会计算后返回数值
% echo $((i*f))
15185.088

# (( )) 用于判断数值比较的结果
% ((i < f && i + 1 > f)) && echo good

# 在 (( )) 中也可以给变量赋值
# (( )) 中的语法类似 c 语言，变量名前不需要 $，等号两边可以有空格
% float result
% ((result = i / f))
% echo $result
9.963063764e-01
```

运算符列表:

运算符的优先级和其他编程语言的差不多，不列出了，如果不确定可以加小括号。这部分内容和 c、java、javascript 等语言基本一致。

7.3 数学函数

Zsh 包含了一个数学模块，如果需要使用数学函数，需要先加载 zsh/mathfunc 模块。

```
% zmodload -i zsh/mathfunc

% echo $((sin(0) + ceil(14.4)))
15.0
```

函数列表:

更多函数:

acos、acosh、asin、asinh、atan、atanh、cos、cosh、erf、erfc、exp、expm1、fabs、gamma、j0、j1、lgamma、log1p、logb、sin、sinh、tan、tanh、y0、y1、ilogb、signgam、copysign、fmod、hypot、nextafter、jn、yn、ldexp、scalb

7.4 参考

http://www.bash2zsh.com/zsh_refcard/refcard.pdf

CHAPTER 8

08 变量修饰语

我们已经了解了字符串、数组、哈希表、整数、浮点数的基本用法，但应付某些复杂的场景依然力不从心。

变量修饰语是 zsh 中有一个很独特的概念，对变量进行操作，功能上和函数类似，但用起来更方便，在一行代码里实现复杂功能主要靠它了。而代价是可读性更差，怎么用就要自己权衡了。它也是 zsh 最有特色的部分之一。变量修饰语主要应用于数组和哈希表，但也有一小部分可以应用于字符串（整数和浮点数也会被当成字符串处理）。

8.1 变量修饰语的格式

其实前边的文章中，变量修饰语已经出现过，但当时没有详细说明。

比如在大小写转换的例子中。

```
% str="ABCDE abcde"

# 转成大写，(U) 和 :u 两种用法效果一样
% echo ${(U)str} --- ${str:u}
ABCDE ABCDE --- ABCDE ABCDE

# 转成小写，(L) 和 :l 两种用法效果一样
% echo ${(L)str} --- ${str:l}
abcde abcde --- abcde abcde
```

这里的 (U)、:l 等等都是变量修饰语。变量修饰语主要有两种格式。

```
${(x)var}
${var:x}
```

其中 var 是变量名，x 是一个或多个字母，不同字母的功能不同。第二行的冒号也可能是其他符号。${var} 和 $var 基本相同，大括号用于避免变量名中的字符和后边的字符粘连，通常情况是不需要加大括号的。但如果使用变量修饰语，大括号就必不可少（其实第二种格式中，大括号可以省略，但考虑可读性和错误提示等因素，还是加上比较好）。

变量修饰语可以嵌套使用。因为加了修饰语的变量依然是变量，可以和正常的变量一样处理。

```
% str=abc
% echo ${(U)str}
ABC
% echo ${(C)${(U)str}}
Abc

% echo ${${a:u}:l}
abc

# 可以简化成
% echo ${a:u:l}
abc

# 可以两种风格嵌套在一起
% echo ${(C)${a:u}}
Abc
```

这里要注意 $ 之后全程不能有空格，否则会有语法错误。也就是说不能通过加空格来避免因为字符挤在一起造成的可读性变差。但熟悉了格式后，就可以比较容易识别出代码的功能。比较复杂的逻辑可以换行继续写，而没必要一定嵌套使用。

知道了变量修饰语的用法后，重要的就是都有哪些可以使用的变量修饰语了。

8.2 变量默认值

和变量默认值（读取变量时如果变量为空或者不存在，使用的默认值）相关的操作，变量可以是任何类型的。

```
% var=123

# 如果变量有值，就输出变量值
% echo ${var:-abc}
```

（下页继续）

(续上页)

```
123

# 如果变量没有值 (变量不存在, 为空字符串、空数组、空哈希表等), 输出 abc
% echo ${varr:-abc}
abc



% var=""
# 和 :- 类似, 但只有变量不存在时才替换成默认值
% echo ${var-abc}
% echo ${varr-abc}
abc



% var=""
# 和 :- 类似, 但如果变量没有值, 则赋值为 abc
% echo ${var:=abc}
abc
% echo $var
abc



% var=abc
# 不管 var 有没有值, 都赋值为 123
% echo ${var::=123}
123
% echo $var
123



% var=""
# 如果 var 没有值, 直接报错
% echo ${var:?error}
zsh: var: error



% var=abc
# 如果 var 有值, 输出 123
% echo ${var:+123}
% echo ${varr:+123}
```

8.3 数组拼接成字符串

```
% array=(aa bb cc dd)

# 用换行符拼接
% echo ${(F)array}
aa
bb
cc
dd

# 用空格拼接
% str=$array
% echo $str
aa bb cc dd

# 使用其他字符或字符串拼接
% echo ${(j:-=:)array}
aa-=bb-=cc-=dd
```

8.4 字符串切分成数组

```
% str=a##b##c##d

% array=(${(s:##:)str})
% print -l $array
a
b
c
d
```

8.5 输出变量类型

```
# 注意如果不加 integer 或者 float，都为字符串，但计算时会自动转换类型
% integer i=1
% float f=1.2
% str=abc
```

(下页继续)

(续上页)

```
% array=(a b c)
% local -A table=(k1 v1 k2 v2)

% echo ${(t)i} ${(t)f} ${(t)str} ${(t)array} ${(t)table}
integer float scalar array association
```

8.6 字符串、数组或哈希表嵌套取值

可以嵌套多层。

```
% str=abcde
% echo ${${str[3,5]}[3]}
e

% array=(aa bb cc dd)
% echo ${${array[2,3]}[2]}
cc
# 如果只剩一个元素了，就取字符串的字符
% echo ${${array[2]}[2]}
b

% local -A table=(k1 v1 k2 v2 k3 v3)
% echo ${${table[k1]}[2]}
1
```

8.7 字符串内容作为变量名再取值

不需要再通过繁琐的 eval 来做这个。

```
% var=abc
% abc=123

% echo ${(P)var}
123
```

8.8 对齐或截断数组中的字符串

```
% array=(abc bcde cdefg defghi)

# 只取每个字符串的最后两个字符
% echo ${(l:2:)array}
bc de fg hi

# 用空格补全字符串并且右对齐
% print -l ${(l:7:)array}
    abc
   bcde
  cdefg
 defghi

# 用指定字符补全
% print -l ${(l:7::0:)array}
0000abc
000bcde
00cdefg
0defghi

# 用指定字符补全，第二个字符只用一次
% print -l ${(l:7::0::1:)array}
0001abc
001bcde
01cdefg
1defghi

# 左对齐
% print -l ${(r:7::0::1:)array}
abc1000
bcde100
cdefg10
defghi1
```

8.9 总结

文中只介绍了几个比较常用的变量修饰语，还有一些没有提及，可能后续再补充。

8.10 参考

http://www.bash2zsh.com/zsh_refcard/refcard.pdf

CHAPTER 9

09 函数和脚本

很多时候，我们写的代码并不是只运行一次就不再用了，那就需要保存到文件里。我们通常称包含解释性编程语言代码的可执行文件为脚本文件，简称脚本。而在脚本内部，也会有一些可以复用的代码，我们可以把这样的代码写成函数，供其他部分调用。Zsh 中函数和脚本基本上一样的，可以认为脚本就是以文件名为函数名的函数。脚本和函数的编写方法基本相同，所以在一起讲。

先从函数开始，因为涉及更少的细节。

9.1 函数定义

```
# 一个很简单的函数
fun() {
    echo good
}

# 也可以在前边加一个 function 关键字
function fun() {
    echo good
}
```

这样就可以定义一个函数了。小括号一定是空的，即使函数有参数，也无需在里边写参数列表。

直接输入函数名即可调用函数。

```
fun() {
    echo good
}

% fun
good
```

用 unfunction 可以删除函数。

```
fun() {
    echo good
}

% unfunction fun
% fun
zsh: command not found: fun
```

9.2 参数处理

函数可以有参数，但 zsh 中无需显式注明有几个参数，直接读取即可。

```
fun() {
    echo $1 $2 $3
    echo $#
}

% fun aa
aa
1
% fun aa bb cc
aa bb cc
3
% fun aa bb cc dd
aa bb cc
4
```

$n 是第 n 个参数，$# 是参数个数。如果读取的时候没有对应参数传进来，那和读取一个未定义的变量效果是一样的。函数的参数只能是字符串类型，如果把整数、浮点数传进函数里，也会被转成字符串。可以把数组传给函数，然后数组中的元素会依次成为各个参数。

```
fun() {
    echo $1 $2 $3
    echo $#
}

% array=(11 22 33)
% fun $array
11 22 33
3
```

这样用的好处是可以更方便地处理带空格的参数。

```
# 遍历所有参数，$* 是包含所有参数的数组
fun() {
    for i ($*) {
        echo $i
    }
}

% fun a b c
a
b
c
```

可以用 $+n 快速判断第 n 个参数是否存在。

```
fun() {
    (($+1)) && {
        echo $1
    }
}
```

关于 `$*` 和 `$@`。在 bash 中，`$*` 和 `$@` 的区别是一个比较麻烦的事情，但在 zsh 中，通常没有必要使用 `$@`，所以不用踩这个坑。Bash 中需要使用 `$@` 的原因是如果使用 `$*` 并且参数中有空格的话，就分不清哪些空格是参数里的，哪些空格是参数之间的间隔符（bash 里的 `$*` 是一个字符串）。而如果使用 `"$*"` 的话，所有的参数都合并成一个字符串了。而 `"$@"` 可以保留参数中的空格，所以通常使用 `"$@"`。但是有些时候需要把所有参数拼接成一个字符串，那么又要使用 `"$*"`，所以很混乱。

而 zsh 中的 `$*` 会包括参数中的空格（zsh 里的 `$*` 是一个数组），所以效果和 bash 的 `"$@"` 是差不多的。另外在 zsh 中用 `"$*"` 和在 bash 中的 `"$*"` 效果一样，所以只用 `$*` 和 `"$*"` 就足够了。

9.3 函数嵌套

函数可以嵌套定义。

```
fun() {
    fun2() {
        echo $2
    }

    fun2 $1 $2
}

% fun aa bb
bb
```

fun2 函数是在 fun 执行过才会被定义的，但最外边也能直接访问 fun2 函数。如果想要最外边访问不了，可以在 fun 结束前调用 unfunction fun2 删除 fun2 函数。

9.4 返回值

函数需要返回一个代表函数是否正确执行的返回值，如果是 0，代表正确执行，如果不是 0，代表有错误。

```
#!/bin/zsh

fun() {
    (($+1)) && {
        return
    }

    return 1
}

% fun 111 && echo good
good
% fun || echo bad
bad

% fun
# 也可以用 $? 获取函数返回值
% echo $?
```

遇到 return 后，函数立即结束。return 即 return 0。

注意返回值不是用来返回数据的，如果函数需要将字符串、整数、浮点数等返回给调用者，直接用 echo 或者 print 等命令输出即可，然后调用者用 $(fun) 获取。如果需要返回数组或者哈希表，只能通过变量（全局变量或者函数所在层次的局部变量）传递。

```
fun() {
    echo 123.456
}

% echo $(($(fun) *2))
246.91200000000001
```

通过全局变量返回。

```
array=()
fun() {
    array=(aa bb)
}

% fun
% echo $array
aa bb
```

9.5 局部变量

在函数中可以直接读写函数外边的变量，并且在函数中定义的新变量在函数退出后依然存在。

```
str1=abcd

fun() {
    echo $str1
    str2=1234
}

% fun
abcd
% echo $str2
1234
```

这通常是不符合预期的。为了避免函数内的变量“渗透”到函数外，可以使用局部变量，使用 local 定义变量。

```
str1=abcd

fun() {
    echo $str1
    local str2=1234
}

% fun
abcd
% echo $str2
```

函数中的变量，除非确实需要留给外部使用，不然最好全部使用局部变量，避免引发 bug。

9.6 脚本

可以认为脚本也是一个函数，但它是单独写到一个文件里的。

test.zsh 内容。

```
#!/bin/zsh

echo good
```

这是一个非常简单的脚本文件。第一行是固定的，供系统找到 zsh 解释器，#! 后加 zsh 的绝对路径即可。如果需要使用环境变量访问，可以用 #!/bin/env zsh （或者!/usr/bin/env zsh，如果 env 在 /usr/bin/ 里边）。

从第二行开始，就和函数中的内容一样了。上边函数体里的内容（去掉首尾行的 fun() { 和 }，都可以写在这里边。

执行的话，在 test.zsh 所在目录，运行 zsh test.zsh 加参数即可（就像调用了一个名为 zsh test.zsh 的函数。也可以 chmod u+x test.zsh 给它添加可执行权限后，直接运行./test.zsh 加参数。

脚本的参数和返回值的处理方法，和函数的完全一样，这里就不举例了。

但函数和脚本中执行的时候是有区别的，函数是在当前的 zsh 进程里执行（也可以调用的时候加小括号在子进程执行），而脚本是在新的子进程里执行，执行完子进程即退出了，所以脚本中的变量值外界是访问不到的，无需使用 local 定义（使用也没问题）。

9.7 exit 命令

脚本可以使用 return 返回，也可以使用 exit 命令。exit 命令用法和 return 差不多，如果不加参数则返回 0。但在代码的任何地方，调用 exit 命令即退出脚本，即使是在一个嵌套很深的函数里边理调用的。

9.8 用 getopts 命令处理命令行选项

有时我们写的脚本需要支持比较复杂的命令行选项，比如 demo -i aa -t bb -cx ccc ddd，这样的话，手动处理就会很麻烦。可以使用内置的 getopts 命令。

```
#!/bin/zsh

# i: 代表可以接受一个带参数的 -i 选项
# c 代表可以接受一个不带参数的 -c 选项
while {getopts i:t:cv arg} {
    case $arg {
        (i)
        # $OPTARG 存放选项对应的参数
        echo $arg option with arg: $OPTARG
        ;;

        (t)
        echo $arg option with arg: $OPTARG
        ;;

        (c)
        echo $arg option
        ;;

        (v)
        echo version: 0.1
        ;;

        (?)
        echo error
        return 1
        ;;
    }
}

# $OPTIND 指向剩下的第一个未处理的参数
echo $*[$OPTIND,-1]

# 或者用 shift 把之前用过的参数移走
# shift $((OPTIND - 1))
# echo $*
```

运行结果：

```
% ./demo -i aaa -t bbb -cv ccc ddd
i option with arg: aaa
t option with arg: bbb
c option
version: 0.1
ccc ddd

# 可以只加部分选项
% ./demo -i aaa -v bbb ccc
i option with arg: aaa
version: 0.1
bbb ccc

# 可以一个选项也不加
% ./demo aaa bbb
aaa bbb

# 如果选项不带参数，多个选项可以合并到一个 - 后
% ./demo -i aaa -cv bbb ccc
i option with arg: aaa
c option
version: 0.1
bbb ccc

# 如果该带参数的选项不带参数，会报错
% ./demo -i aaa -t
i option with arg: aaa
./demo:3: argument expected after -t option
error

# 加了不支持的选项也会报错
% ./demo -i aaa -a bbb ccc
i option with arg: aaa
./demo:3: bad option: -a
error

# 如果该带参数的选项不带参数，然后后边紧接着另一个选项，那么选项会被当作参数
% ./demo -i -c aaa bbb
i option with arg: -c
```

(下页继续)

(续上页)

```
aaa bbb
```

getopts 的使用还是很方便的，但它不支持长选项（如 –log aaa）。如果需要使用长选项，可以用 getopt 命令，它是一个外部命令，可以 man getopt 查看用法。

9.9 总结

本文简单介绍了函数和脚本的写法，重点是参数处理和返回值等等，还有很多没覆盖的地方，以后可能继续补充。

9.10 参考

https://my.oschina.net/lenglingx/blog/410565

9.11 更新历史

20170901：增加用 $? 获取函数返回值的内容。

20170902：增加“用 getopts 命令处理命令行选项”。

CHAPTER 10

10 文件查找和批量处理

寻找满足特定条件的文件路径，简称文件查找，是 shell 脚本的常见任务，因为条件复杂多样，这样的任务并不轻松。很多人使用 find 命令来做，但 find 只能覆盖一部分功能，其他的要自己进一步处理，而且 find 并不好用，和脚本其他部分配合也比较麻烦，容易出错。用 zsh 的话，基本不需要 find 命令，借助 zsh 自身的功能便足以应付多数场景，而且语法更优雅简洁不易出错。

10.1 简单例子

列出 /usr/bin 目录下以 zsh 开头的文件。

```
# 加 -l 为了换行显示更易读，如果需要操作这些文件，将 print -l 换成其他命令即可
% print -l /usr/bin/zsh*
/usr/bin/zsh
/usr/bin/zsh-5.4.1
/usr/bin/zshdb
```

有人可能会说用 ls /usr/bin/zsh* 就行。如果用 ls 的话，就平添了不少额外工作，因为 zsh* 已经匹配一次文件路径，结果出来了，传给 ls 后，ls 又去 stat 了一下那些文件，而这完全是多余工作，如果文件列表长的话，要多消耗不少时间。很多看起来理所当然的 shell 用法都存在类似这样的问题。所以打命令或者写脚本时，不能看结果正确就可以了，要知其所以然。如果嫌 print -l 太长，alias 个 pl 就可以了，print -l 非常常用。

删除 /tmp 下所有的形如 abc1234.tmp（前边字母后边数字，个数不限，但至少有一个字母和一个数字）的文件，包括子目录里的。

```
% setopt EXTENDED_GLOB
# /**/ 是递归搜索文件
# ## 是前边的内容至少重复一次，<-> 是任何正整数或 0
% rm -v /tmp/**/[a-zA-Z]##<->.tmp
removed '/tmp/yaourt-tmp-goreliu/abc123.tmp'
```

setopt EXTENDED_GLOB 是启用扩展的通配符支持，本文后续的内容默认该选项已开启，不然通配符功能太弱，建议写到.zshrc 里边。通配符的内容之前已经讲过，可以当手册参考。

两个小例子热完身后开始进入正题。

10.2 按文件属性查找

除了匹配文件路径外，很多时候我们还需要按文件属性查找，比如根据文件类型、权限、大小、修改时间等等。这里需要使用一个新东西，通配符修饰语。

先举个例子看看它的样子。列出当前目录及子目录中的所有普通文件（即 ls -l 结果中第一位是 - 的文件，非目录、符号链接、设备文件、socket、FIFO 等等）。

```
% print -l **/*(.)
a.txt
b/htm
```

这里比之前的例子多了个末尾的小括号，里边有一个点。这个小括号及里边的内容便是通配符修饰语，专门用于按文件属性来匹配文件。点（.）代表普通文件。

更多例子：

```
# 列出当前目录下的非空目录，F 是 FULL，满的意思
% print -l *(/F)

# 列出当前目录下的空目录，^ 是取反
% print -l *(/^F)

# 列出当前目录下的符号链接文件和可执行的普通文件，多种文件类型用逗号隔开
% print -l *(@,.x)

# 列出符合 0644 权限的普通文件
% print -l *(.f0644)
```

那么我们来看下都有哪些可用的通配符修饰语，然后再举更复杂的例子。

10.3 通配符修饰语列表

10.4 更复杂的用法

10.4.1 按文件时间查找文件

```
# 列出最近一天修改过内容的文件
% print -l *(.m-1)

# 列出最近一个月没有读取过的文件
% print -l *(.aM+1)
```

m 后边可加单位，如果没有单位，默认是天。其他单位：M（月）、w（周）、h（小时）、m（分钟）、s（秒）。+ 是指定时间之前，- 是指定时间之内。

a 是最后访问时间（atime），但注意如果分区挂载时指定了 noatime 或者 realtime（可以查看 /proc/mount 确认），那么 atime 并不是真正的最后访问时间。m 是最后修改时间（mtime），这里指内容修改，而不包括文件属性（如权限）的修改。c 是最后状态修改时间（ctime），如果文件内容没有修改，而文件属性发生变化，这个时间会更新。如果不能理解请在网上搜索相关文章。

10.4.2 按文件大小查找文件

```
# 列出当前目录下所有空文件
% print -l *(.L0)

# 列出当前目录下小于 2k 的文件
% print -l *(.Lk-2)

# 列出当前目录下大于 1m 的文件
% print -l *(.Lm+1)

# 注意这样只能找到空文件，因为以 m 为单位的话，文件只能是 0 m 或者 1 m，不能 0.5 m
# 所以比 1 小就是 0 m，是空文件
% print -l *(.Lm-1)
```

默认的单位是字节，还可以使用 k、m 和 p（512 字节的块），也可以使用大写的 K、M、P，含义一样。

10.4.3 文件排序

```
# 按文件名排序，同一目录下的文件和目录名会一起排，而不是先排目录再排文件
% print -l **/*(.on)
bb.txt
cc/aa.txt
cc/dd.txt
zz.txt

# 按文件的目录深度逆序排，d 是从深往浅排，O 是逆序
% print -l **/*(.Od)
zz.txt
bb.txt
cc/dd.txt
cc/aa.txt

# 先按文件名排序，然后再按大小排序，这样大小相同的文件依然是按文件名排的
% print -l **/*(.onoL)
bb.txt
cc/aa.txt
cc/dd.txt
cc.txt
```

像第三个例子那样，可以排多次。

可供排序的因素：n（文件名，如果不指定排序选项，默认按文件名排，即 on）、L（大小）、l（硬连接数）、a（atime）、m（mtime）、c（ctime）、d（所在目录深度，从深到浅排）。

10.4.4 组合使用

现在我们大概了解了都有哪些可供使用的通配符修饰语，单个使用已经没有什么问题了。但如果同时使用多个，就涉及到怎么组合在一起的问题。

类型和类型之间要用逗号隔开，如果不指定类型，代表所有类型都可以，逗号前后的内容互不干扰（取反 ^ 操作只影响到逗号之前内容）。同一个类型可以同时加多个选项，依次添加即可。

```
# 当前目录下的两天内修改过的目录
# 加上小于 3 m 的普通文件从小到大排
# 再加上所有的符号链接文件（包括隐藏文件）
% print -l *(/m-2,.Lm-3oL,@D)
```

10.5 文件批量重命名

对文件进行批量重命名，是一个比较常见的场景。Zsh 中有一个非常方便的命令 zmv，它可以让批量重命名变得很简单。

```
# 使用前需要先加载进来
% autoload -U zmv

# 将所有 txt 文件扩展名改成 conf
# 参数要用单引号扩起来，$1 代表第一个参数中括号中的内容
%  zmv '(*).txt' '$1.conf'

# 如果加了 -W 参数，zmv 会自动识别文件名中需要保留的部分
%  zmv -W '*.txt' '*.conf'

# 调整文件名各部分的前后顺序
% zmv '(*).(*).txt' '$2.$1.txt'
# 加 -n 预览而不实际运行
% zmv -n '(*).(*).txt' '$2.$1.txt'
mv -- a.b.txt b.a.txt

# 0 1 2 ... 前添加 0，以便和 10 11 12 ... 宽度一致
% zmv '([0-9]).(*)' '0$1.$2'
# 去掉开头的一个 0
% zmv '(0)(*)' '$2'

# 文件整理到目录
% zmv '(*) - (*) - (*).txt' '$1/$2 - $3.txt'

# 转换大小写
% zmv '(*).txt' '${(U)1}.txt'
% zmv '(*).txt' '${(L)1}.txt'
```

10.6 不展开通配符

有时我们不想展开通配符，比如我写了一个计算的函数叫做 calc：

```
calc() {
    zmodload zsh/mathfunc
    echo $(($*))
```

(下页继续)

(续上页)

```
}

% calc 12+12
24
```

但如果我想计算 12 * 12：

```
% calc 12*12
zsh: no matches found: 12*12
```

如果不加引号的话，星号会被作为通配符使用，然后去找符合 12*12 的文件名，没找到所有报错了。但我并不想找文件。

noglob 命令可以禁止展开后边内容的通配符，这样就不需要加引号了。

```
% noglob calc 12*12
144
```

然后可以写个 alias：

```
% alias js="noglob calc"
% js 12*12
144
```

这样就可以更方便地使用计算器了。

10.7 总结

本文介绍了文件查找中的通配符修饰语的用法，并且列出来大多数常用的通配符修饰语，还有一小部分更复杂或者更少用的暂时空缺，以后可能会补上。这些通配符修饰语没有必要全部记下来，熟悉常用的，其余的等用的时候再查询即可。

10.8 参考

http://www.bash2zsh.com/zsh_refcard/refcard.pdf

http://blog.sina.com.cn/s/blog_687bd5d50101epna.html

10.9 更新历史

2017.08.31：增加“不展开通配符”和“文件批量重命名”。

CHAPTER 11

11 变量的进阶内容

之前我们已经依次讲过 zsh 下的五种变量（字符串、数组、哈希表、整数、浮点数）的基本用法。但变量的使用方面，还有一些比较进阶的内容，这对一些比较特别的场景很有帮助。

11.1 typeset 命令

typeset 命令用于对变量进行详细的设置。我们之前在哈希表那篇见过它。typeset -A 可以用来定义哈希表。

```
% typeset -A table=(aa bb cc dd)
```

但我们后续都使用 local，因为 local 的功能和 typeset 是一样的（除了不能用 -f 和 -g，这两个选项不常用），并且更短更容易输入。这里提到 typeset 命令，因为这个名称很好地反映了它的功能。但知道了这个后，我们可以继续使用 local 命令，毕竟它们是一样的。

typeset 命令有很多选项，可以作用在变量上，起到各种各样的效果。

11.2 强制字符串内容为小写或者大写

```
# 强制字符串内容为小写
% local -l str=abcABC && echo $str
abcabc
```

(下页继续)

(续上页)

```
# 强制字符串内容为大写
% local -u str=abcABC && echo $str
ABCABC
```

11.3 设置变量为环境变量

```
% local -x str=abc
# 通常使用 export, 功能一样
% export str=abc
```

环境变量可以被子进程读取。

11.4 设置变量为只读变量

```
% local -r str1=abc
# 通常使用 readonly, 功能一样
% readonly str2=abc

% str1=bcd
zsh: read-only variable: str1
% str2=bcd
zsh: read-only variable: str2
```

11.5 设置数组不包含重复元素

```
% local -U array=(aa bb aa cc) && echo $array
aa bb cc
```

11.6 设置整数的位数

```
# 如果位数不够, 输出内容会用 0 补全
% local -Z 3 i=5 && echo $i
005
```

(下页继续)

(续上页)

```
# 如果超出范围会被截断
% local -Z 3 i=1234 && echo $i
234
```

11.7 进制转换

设置整数为其他进制显示：

```
% local -i 16 i=255
% echo $i
16#FF
```

可以设置 2 到 36 之间任意进制。设置几进制显示，并不影响计算，只是显示格式不同。

用 [#n] num 也可以显示十进制数为 n 进制：

```
% echo $(([#16] 255))
16#FF
```

可以用 n#num 来显示 n 进制整数为十进制：

```
% echo $((16#ff))
255
```

我们可以定义一系列函数来快捷地转换进制，不需要使用 bc 等外部命令：

```
0x() {
    echo $((16#$1))
}

0o() {
    echo $((8#$1))
}

0b() {
    echo $((2#$1))
}

p16() {
    echo $(([#16] $1))
```

(下页继续)

(续上页)

```
}

p8() {
    echo $(([#8] $1))
}

p2() {
    echo $(([#2] $1))
}


# 其他进制转十进制
% 0x ff
255
% 0b 1101
13

# 十进制转其他进制
% p16 1234
16#4D2
```

11.8 同时对多个变量赋相同的值

```
% local {i,j,k}=123
% echo $i $j $k
123 123 123
```

11.9 绑定字符串和数组

```
% local -T DIR dir
% dir=(/a /b/c /b/d /e/f)
% echo $DIR
/a:/b/c:/b/d:/e/f

# 删除 dir 后, DIR 也会被删除 (反之亦然)
% unset dir
```

(下页继续)

(续上页)

```
% echo $+DIR
0
```

Linux 下经常需要处理带分隔符冒号的字符串（比如 $PATH）。如果只修改其中某一个字段，比较麻烦。local -T 可以把字符串绑定到数组上，这样直接修改数组，字符串内容也会同步变化（反之亦然）。其实在 zsh 中，$PATH 字符串就是和 $path 数组绑定的，可以直接通过修改 $path 来达到修改 $PATH 的目的，这在某些场景会方便很多。

11.10 显示变量的定义方式

```
% array=(aa bb cc)
% local -p array
typeset -a array=(aa bb cc)

% array+=(dd)
% local -p array
typeset -a array=(aa bb cc dd)
```

11.11 什么地方该加双引号

用过 bash 的读者大概会对里边的双引号印象比较深刻，很多地方不加双引号都会出错，为了避免出错，很多人每个变量左右都加上双引号，麻烦不说，代码看起来也比较乱。

其实 zsh 中已经没有那些问题了，变量两边无需加双引号，不会出现莫名其妙的错误。但有些地方还是需要加双引号的。

需要加双引号的场景：

1. 像这样的包含字符或者特殊符号的字符串 `"aa bb \t \n *"` 出现在代码中时，两边要加双引号，这个基本不需要说明。
2. 在用 `$()` 调用命令时，如果希望结果按一个字符串处理，需要加上双引号，`"$()"`，不然的话，如果命令结果中有空格，`$()` 会被展开成多个字符串。
3. 如果想将数组当单个字符串处理，需要加双引号，`array=(a b); print -l "$array"`。
4. 其他的原本不是单个字符串的东西，需要转成单个字符串的场景，要加双引号。

其余情况通常都不需要加双引号，典型的情况：

1. 任何情况下，字符串变量的两边都不需要加双引号，无论里边的内容多么特殊，或者变量存不存在，都没有关系，如 `$str`。
2. 如果不转换类型（比如数组转成字符串），任何变量的两边都不需要加双引号。

3. `$1 $2 $*` 这些参数（其实它们也都是单个字符串），都不需要加双引号，无论内容是什么，或者参数是否存在。

以上的 7 种情况几乎覆盖了所有场景，如果有没覆盖到的，试一下即可（让里边的内容包含空格、换行和其他特殊字符等等，看看结果是否符合预期）。

11.12 总结

本文简单介绍了一些比较使用的 typeset（或者 local）命令的用法，typeset 命令还有很多其他参数，但一般很少用，以后我也会继续更新。

11.13 参考

http://www.bash2zsh.com/zsh_refcard/refcard.pdf

http://www.linux-mag.com/id/1079/

11.14 更新历史

20170831：新增“什么地方该加双引号”

CHAPTER 12

12 [[ ]] 的用法

[[ ]] 是我们比较熟悉的符号了，从第一篇开始我们就一直在用，但我一直没有详细介绍它的用法，只用到了它的一小部分功能。本文详细介绍 [[ ]] 的用法。

12.1 比较字符串

[[ ]] 最常用的功能之一是比较字符串，这也是我们一直在用的功能。

```
# 匹配
% [[ abc == abc ]] && echo good
good

# = 和 == 是一样的，最好统一使用一种
% [[ abc = abc ]] && echo good
good

# 不匹配
% [[ abc != abd ]] && echo good
good

# 正则表达式匹配
% [[ abc =~ a.c ]] && echo good
good
```

(下页继续)

(续上页)

```
# 前者字符序比后者小
% [[ abc < bcd ]] && echo good
good

# 前者字符序比后者大
% [[ cde > bcd ]] && echo good
good

# 没有 >= 和 <=
% [[ cde >= bcd ]] && echo good
zsh: parse error near `bcd'
```

除了在里边用等号、不等号之类比较外，还可以判断字符串是否为空：

```
% str=abc
# 判断字符串内容长度是否大于 0，等同于 (($#str))
% [[ -n $str ]] && echo good
good

% str=""
# 判断字符串是否为空，等同于 ((! $#str))
% [[ -z $str ]] && echo good
good
```

但这两种用法，我们都有更方便的其他实现方法，没有必要用它们。

12.2 判断文件

[[ ]] 另一类很重要的功能是判断文件，比如判断某一个文件是否存在、是否是目录、是否可读等等。

判断 /bin/zsh 文件是否存在：

```
% [[ -e /bin/zsh ]] && echo good
good
% [[ -e /bin/zshh ]] && echo good
```

-e 可以替换成如下的选项，用法是一致的：

还有一个比较特殊的 -t 选项：

```
# $$ 是当前的进程 id
% ls /proc/$$/fd
0  1  10  11  2
% [[ -t 10 ]] && echo good
good
% [[ -t 3 ]] && echo good
```

-t 后要接数字（如果不是，相当于 0），判断当前进程是否打开了对应的 fd（进程默认会打开 0、1、2 这三个 fd，分别对应标准输入、标准输出和错误输出，此外每打开一个文件、管道或者网络连接，都会对应一个 fd，关掉后对应 fd 会消失）。

12.3 比较文件

除了判断单个文件是否符合条件外，[[ ]] 还可以用来比较两个文件。

```
# file1 比 file2 新
% [[ file1 -nt file2 ]]

# file1 比 file2 旧
% [[ file1 -ot file2 ]]

# file1 和 file2 是否对应同一个文件（路径相同或者互为硬连接）
% [[ file1 -ef file2 ]]
```

12.4 比较数值

[[ ]] 也可以用来比较数值，注意不是用等号、大于号、小于号等等比较，有一系列专门的符号。通常我们没必要用 [[ ]] 来比较数值，用 (( )) 更方便一些。

```
# -eq 是判断两个数值是否相等
% [[ 12 -eq 12 ]] && echo good
good
```

-eq 可以替换成下列符号，用法一样：

12.5 组合使用

```
# && 是逻辑与
% [[ a == a && b == b ]] && echo good
good

# || 是逻辑或
%  [[ a == a || a == b ]] && echo good
good

# ! 是逻辑非
% [[ ! a == b ]] && echo good
good

# 可以一起用，! 优先级最高，其次 &&，再次 ||
% [[ ! a == b && b == a || b == b ]] && echo good
good

# 如果不确定优先级，可以加小括号
% [[ ((! a == b) && b == a) || b == b ]] && echo good
good
```

需要注意一下空格，[[ ]] 内侧和内容之间需要空格隔开，== 两边也需要空格。如果是在 zsh 中直接敲入，！后边也要加一个空格，不然会被解析成历史命令。

12.6 [ ] 符号

除了 [[ ]] 符号，[ ] 符号（它是古老的 test 命令化身）也可以用来判断字符串、文件、数值等等，但功能没有 [[ ]] 全，只支持上边列的一部分功能（不支持 ==、=~、>、<、(、)，并且逻辑与或的语法不一样，不能调整优先级，用起来很不方便），通常没有必要使用 [ ]（如需使用，可以 man test 查看用法）。

12.7 总结

本文详细介绍了 [[ ]] 的用法，基本覆盖全面了。

12.8 参考

http://www.bash2zsh.com/zsh_refcard/refcard.pdf

CHAPTER 13

13 管道和重定向

到目前为止，我们已经大致了解了 zsh 的语法特性，可以写一些功能不复杂的脚本了。但 shell 脚本主要的应用场景并不是闭门造车写独立的程序，而是和外部环境交互。所以要写出实用的脚本，要了解 zsh 如何和外部环境交互。这里的外部环境包括其他进程、文件系统、网络等等。本篇主要讲管道和重定向，这是和其他进程、文件系统等交互的基础。

本文中的命令主要是为了演示管道的用法，在实际脚本中通常不需要使用这些命令，因为可以用 zsh 代码直接实现。另外本系列文章不详细讲任何外部命令的用法，因为相关文档或者书籍特别多。如果看不懂本文的某些内容，可以暂时跳过，基本不影响其余部分的理解。

13.1 管道

管道是类 Unix 系统中的一个比较基础也特别重要的概念，它用于将一个程序的输出作为另一个程序的输入，进而两个程序的数据可以互通。如果只是使用管道，还是非常简单易懂的，并不需要了解管道的实现细节。

管道的基本用法：

```
% ls
git  tmp
# wc -l 功能是计算输入内容的行数
% ls | wc -l
2
```

| 即管道，在键盘上是主键盘区右侧 \ 对应的上档键字符。如果只输入 wc -l，wc 会等待用户输入，这时可以输入字符串，然后回车继续输入，直到按 ctrl + d 结束输入。然后 wc 会统计用户一共输入了多少行，然

后输出行数。

```
# 敲 wc -l 回车后，依次按 a 回车 b 回车 ctrl + d
% wc -l
a
b
2
```

但如果前边有个管道符号，ls | wc -l，那么 wc 就不等待用户输入了，而是直接将 ls 的结果作为输入读取过来，然后统计行数，输出结果。

13.2 关于管道的更多细节

我们再运行一个简单的例子：

```
% cat | wc -l

# 查看 cat 进程打开的 fd
% ls -l /proc/$(pidof cat)/fd
total 0
lrwx------ 1 goreliu goreliu 0 2017-08-30 21:15 0 -> /dev/pts/1
l-wx------ 1 goreliu goreliu 0 2017-08-30 21:15 1 -> pipe:[2803]
lrwx------ 1 goreliu goreliu 0 2017-08-30 21:15 2 -> /dev/pts/1

# 查看 wc 进程打开的 fd
% ls -l /proc/$(pidof wc)/fd
total 0
lr-x------ 1 goreliu goreliu 0 2017-08-30 21:16 0 -> pipe:[2803]
lrwx------ 1 goreliu goreliu 0 2017-08-30 21:16 1 -> /dev/pts/1
lrwx------ 1 goreliu goreliu 0 2017-08-30 21:16 2 -> /dev/pts/1
```

cat 命令的效果是等待用户输入，等用户输入一行，它就把这行再输出来，直到用户按 ctrl - d。所以 cat | wc -l 也会等待用户输入。

我们看下 fd 的指向，/dev/ps1/1 是指向伪终端设备文件的，进程就是通过这个来读取用户的输入和输出自己的内容。0 是标准输入（即用户输入端），1 是标准输出（即正常情况的输出端），2 是错误输出（即异常情况的输出端）。但是 cat 的输出端指向了一个管道，并且 wc 的输入端指向了一个相同的管道，这代表两个进程的输入输出端是通过管道连接的。这种管道是匿名管道，即只在内核中存在，是没有对应的文件路径的。

13.3 重定向

重定向，指的便是 fd 的重定向，管道也是重定向的一种方法。但用得更多的是将进程的 fd 重定向到文件。

一个最简单的例子是输出内容到文件。

```
% echo abce > test.txt
% cat test.txt
abce
```

因为这个用法太常见了，大家可能习以为常了。我们依然来看下更多的细节。

```
% cat > test.txt

# 在另一个 zsh 中运行
% ls -l /proc/$(pidof cat)/fd
total 0
lrwx------ 1 goreliu goreliu 0 Aug 30 21:43 0 -> /dev/pts/1
l-wx------ 1 goreliu goreliu 0 Aug 30 21:43 1 -> /tmp/test.txt
lrwx------ 1 goreliu goreliu 0 Aug 30 21:43 2 -> /dev/pts/1
```

可以看到标准输出已经指向 test.txt 文件了。

除了标准输出可以重定向，标准输入（fd 0），错误输出（fd 2）也都可以。

```
% touch 0.txt 1.txt 2.txt
% sleep 1000 <0.txt >1.txt 2>2.txt

# 在另一个 zsh 中运行
% ls -l /proc/$(pidof sleep)/fd
total 0
lr-x------ 1 goreliu goreliu 0 Aug 30 21:46 0 -> /tmp/0.txt
l-wx------ 1 goreliu goreliu 0 Aug 30 21:46 1 -> /tmp/1.txt
l-wx------ 1 goreliu goreliu 0 Aug 30 21:46 2 -> /tmp/2.txt
```

<0.txt 是重定向标准输入，2>2.txt 是重定向错误输出，>1.txt（即 1>1.txt）是重定向到标准输出。然后我们看到 3 个文件已经各就各位，全部被重定向了。但因为 sleep 并不去读写任何东西，重定向它的输入输出没有什么意义。

13.4 更多重定向的用法

一个 fd 只能重定向到一个文件，一一对应。但在 zsh 中，我们可以把一个 fd 对应到多个文件。

```
% cat >0.txt >1.txt >2.txt
```

输入完成后，3 个文件的内容都更新了，这是怎么回事呢?

其实是 zsh 进程做了中介。

```
% pstree -p | grep cat
        `-tmux: server(1172)-+-zsh(1173)---cat(1307)---zsh(1308)

% ls -l /proc/1307/fd
total 0
lrwx------ 1 goreliu goreliu 0 Aug 30 21:57 0 -> /dev/pts/1
l-wx------ 1 goreliu goreliu 0 Aug 30 21:57 1 -> pipe:[2975]
lrwx------ 1 goreliu goreliu 0 Aug 30 21:57 2 -> /dev/pts/1

% ls -l /proc/1308/fd
total 0
l-wx------ 1 goreliu goreliu 0 Aug 30 21:58 12 -> /tmp/0.txt
l-wx------ 1 goreliu goreliu 0 Aug 30 21:58 13 -> /tmp/1.txt
lr-x------ 1 goreliu goreliu 0 Aug 30 21:58 14 -> pipe:[2975]
l-wx------ 1 goreliu goreliu 0 Aug 30 21:58 15 -> /tmp/2.txt
```

可以看到 cat 的标准输出是重定向到管道了，管道对面是 zsh 进程，然后 zsh 打开了那三个文件。实际将内容写入文件的是 zsh，而不是 cat。但不管是谁写入的，这个用法很方便。

标准输入、错误输出也可以重定向多个文件。

```
% echo good >0.txt >1.txt >2.txt

% cat <0.txt <1.txt <2.txt
good
good
good
```

给 cat 的标准输出重定向 3 个文件，它将 3 个文件的内容全部读取了出来。

除了能同时重定向 fd 到多个文件外，还可以同时重定向到管道和文件。

```
# 敲完 a b c 后 ctrl -d 退出
% cat >0.txt >1.txt | wc -l
a
b
c
```

(下页继续)

(续上页)

```
3

% cat 0.txt 1.txt
a
b
c
a
b
c
```

可以看到输入的内容写入了文件，并且通过管道传给了 wc -l，不用说，这又是 zsh 在做背后工作，将数据分发给了文件和管道。所以在 zsh 中是不需要使用 tee 命令的。

13.5 命名管道

除了匿名管道，我们还可以使用命名管道，这样更容易控制。命名管道所使用的文件即 FIFO（First Input First Output，先入先出）文件。

```
# mkfifo 用来创建 FIFO 文件
% mkfifo fifo
% ls -l
prw-r--r-- 1 goreliu goreliu 0 2017-08-30 21:29 fifo|

# cat 写入 fifo
% cat > fifo

# 打开另一个 zsh，运行 wc -l 读取 fifo
% wc -l < fifo
```

然后在 cat 那边输入一些内容，按 ctrl - d 退出，wc 这边就会统计输入的行数。

在输入完成之前，我们也可以看一下 cat 和 wc 两个进程的 fd 指向哪里：

```
% ls -l /proc/$(pidof cat)/fd
total 0
lrwx------ 1 goreliu goreliu 0 Aug 30 21:35 0 -> /dev/pts/2
l-wx------ 1 goreliu goreliu 0 Aug 30 21:35 1 -> /tmp/fifo
lrwx------ 1 goreliu goreliu 0 Aug 30 21:35 2 -> /dev/pts/2

% ls -l /proc/$(pidof wc)/fd
```

(下页继续)

(续上页)

```
total 0
lr-x------ 1 goreliu goreliu 0 Aug 30 21:34 0 -> /tmp/fifo
lrwx------ 1 goreliu goreliu 0 Aug 30 21:34 1 -> /dev/pts/1
lrwx------ 1 goreliu goreliu 0 Aug 30 21:34 2 -> /dev/pts/1
```

可以看到之前的匿名管道已经变成了我们刚刚创建的 fifo 文件，其他的并无不同。

13.6 exec 命令的用法

说起重定向，就不得不提 exec 命令。exec 命令主要用于启动新进程替换当前进程以及对 fd 做一些操作。

用 exec 启动新进程：

```
% exec cat
```

看上去效果和直接运行 cat 差不多。但如果运行 ctrl + d 退出 cat，终端模拟器就关闭了，因为在运行 exec cat 的时候，zsh 进程将已经被 cat 取代了，回不去了。

但在脚本中很少直接这样使用 exec，更多情况是用它来操作 fd：

```
# 将当前 zsh 的错误输出重定向到 test.txt
% exec 2>test.txt
# 随意敲入一个不存在的命令，错误提示不出现了
% fdsafds
# 错误提示被重定向到 test.txt 里
% cat test.txt
zsh: command not found: fdsafds
```

更多用法：

更多例子：

```
# 把错误输出关闭，这样错误内容就不再显示
% exec 2>&-
% fsdafdsa

% exec 3>test.txt
% echo good >&3
% exec 3>&-
# 关闭后无法再输出
% echo good >&3
zsh: 3: bad file descriptor
```

(下页继续)

(续上页)

```
% exec 3>test.txt
# 将 fd 4 的输出重定向到 fd 3
% exec 4>&3
% echo abcd >&4
# 输出内容到 fd 4，test.txt 内容更新了
% cat test.txt
abcd
```

通常情况我们用 exec 主要为了重定向输出和关闭输出，比较少操作输入。

13.7 总结

本文讲了管道和重定向的基本概念和各种用法。Zsh 中的重定向还是非常灵活好用的，之后的文章会详细讲在实际场景中怎样使用。

13.8 参考

http://adelphos.blog.51cto.com/2363901/1601563

13.9 更新历史

20170901：增加“exec 命令的用法”。

CHAPTER 14

14 文件读写

之前我们也偶尔接触过读写文件的方法，本篇会系统讲读写文件的各种方法。

14.1 写文件

写文件要比读文件简单一些，最常用的用法是使用 > 直接将命令的输出重定向到文件。如果文件存在，内容会被覆盖；如果文件不存在，会被创建。

```
% echo abc > test.txt
```

如果不想覆盖之前的文件内容，可以追加写入：

```
% echo abc >> test.txt
```

这样如果文件存在，内容会被追加写入进去；如果文件不存在，也会被创建。

14.1.1 创建文件

有时我们只想先创建个文件，等以后需要的时候再写入。

touch 命令用于创建文件（普通文件）：

```
% touch test1.txt test2.txt
```

(下页继续)

(续上页)

```
# 或者用 echo 输出重定向，效果和 touch 一样
# 加 -n 是因为不加的话 echo 会输出一个换行符
% echo -n >>test1.txt >>test2.txt

# 或者使用输入重定向
% >>test1.txt >>test2.txt </dev/null

# mkdir 用来创建目录，如果需要在新目录创建文件
% mkdir dir1 dir2
```

如果文件已经存在，touch 命令会更新它的时间（mtime、ctime、atime 一起更新，其余两种方法不会）到当前时间。另外下边的清空文件方法，也都可以用来创建文件。touch 命令的使用比较方便，但如果想尽量少依赖外部命令，可以使用后两种方法。

因为文件创建过程通常不存在性能瓶颈，不用过多考虑性能因素。如果需要创建大量文件，可以在自己的环境分别用这几种方法试验几次，看需要多少时间。

我在树莓派 3B 简单测试一下：

```
# 三个脚本，分别创建 1000 个文件
% cat test1 test2 test3
#!/bin/zsh

touch test1{1..1000}.txt
#!/bin/zsh

echo -n >>test2{1..1000}.txt
#!/bin/zsh

>>test3{1..1000}.txt </dev/null
```

```
# 运行了几次，结果差不多
% time ./test1; time ./test2; time ./test3
./test1  0.02s user 0.03s system 86% cpu 0.058 total
./test2  0.02s user 0.02s system 70% cpu 0.056 total
./test3  0.03s user 0.01s system 72% cpu 0.055 total
```

另外如果文件数量太多的话，方法二、三要按批次创建，因为一个进程能打开的 fd 总数是有上限的。

14.1.2 清空文件

有时我们需要清空一个现有的文件：

```
# 使用 echo 输出重定向
% echo -n >test.txt

# 使用输入重定向
% >test.txt </dev/null

# 也可以使用 truncate 命令清空文件
% truncate -s 0 test.txt
```

通常使用第一种方法即可，比较简单易懂。非特殊场景尽量不要用像 truncate 这样不常见的命令。

14.1.3 删除文件

删除文件的方法比较单一，用 rm 命令即可。

```
% rm test1.txt test2.txt

# -f 参数代表即使文件不存在也不报错
% rm -f test1.txt test2.txt

# -r 参数可以递归删除目录和文件
% rm -r dir1 dir2 test*.txt

# -v 参数代表 rm 会输出删除文件的过程
% rm -v test*.txt
removed 'test1.txt'
removed 'test2.txt'
```

删除文件必须借助 rm 命令。如果一定要不依赖外部命令的话，zsh/files 模块里也有一个 rm 命令，可以用 zmodload zsh/files 加载，然后 rm 就变成了内部命令，用法基本相同。

```
% zmodload zsh/files
% which -a rm
rm: shell built-in command
/usr/bin/rm
```

此外 zsh/files 中还有内置的 chgrp、chown、ln、mkdir、mv、rmdir、sync 命令。如果不想依赖外部命令，或者系统环境出问题了用不了外部命令，可以使用这些。这可以作为命令不存在或者因为命令本身问题执行异常的一个 fallback 方案，来提高脚本的健壮性。

14.1.4 多行文本写入

通常我们写文件时不会每一行都单独写入，这样效率太低。

可以先把字符串拼接起来，然后一次性写入，这样比多次写入效率更高：

```
% str=ab
% str+="\ncd"
% str +="\n$str"

echo $str > test.txt
```

可以直接把数组写入到文件，每行一个元素：

```
% array=(aa bb cc)

% print -l $array > test.txt
```

如果是将一段内容比较固定的字符串写入到文件，可以这样：

```
# 在脚本中也是如此，第二行以后的行首 > 代表换行，非输入内容
# <<EOF 代表遇到 EOF 时会终止输入内容
# 里边也可以使用变量
% > test.txt <<EOF
> aa
> bb
> cc dd
> ee
> EOF

% cat test.txt
aa
bb
cc dd
ee
```

14.1.5 用 mapfile 读写文件

如果不喜欢使用重定向符号，还可以用哈希表来操作文件。Zsh 有一个 zsh/mapfile 模块，用起来很方便：

```
% zmodload zsh/mapfile

```

(下页继续)

(续上页)

```
# 这样就可以创建文件并写入内容，如果文件存在则会被覆盖
% mapfile[test.txt]="ab cd"
% cat test.txt
ab cd

# 判断文件是否存在
% (($+mapfile[test.txt])) && echo good
good

# 读取文件
% echo $mapfile[test.txt]
ab cd

# 删除文件
% unset "mapfile[test.txt]"

# 遍历文件
% for i (${(k)mapfile}) {
> echo $i
> }
test1.txt
test2.txt
```

14.1.6 从文件中间位置写入

有时我们需要从一个文件的中间位置（比如从第 100 的字符或者第三行开始）继续写入，覆盖之后的内容。Zsh 并不直接提供这样的方法，但我们可以迂回实现，先用 truncate 命令把文件截断，然后追加写。如果文件后边的内容还需要保留，可以在截断之前先读取进来（见下文读文件部分的例子），最后再写回去。

```
% echo 1234567890 > test.txt
# 只保留前 5 个字符
% truncate -s 5 test.txt
% cat test.txt
12345
% echo abcde >> test.txt
% cat test.txt
12345abcde
```

14.2 读文件

14.2.1 读取整个文件

读取整个文件比较容易：

```
% str=$(<test.txt)
% echo $str
aa
bb
cc dd
ee
```

14.2.2 按行遍历文件

如果文件比较大，那读取整个文件会消耗很多资源，可以按行遍历文件内容：

```
% while {read i} {
> echo $i
> } <test.txt
aa
bb
cc dd
ee
```

read 命令是从标准输入读取一行内容，把标准输入重定向后，就变成了从文件读取。

14.2.3 读取指定行

如果只需要读取指定的某行或者某些行，不需要用上边的方法加自己计数。

```
# (f)2 是读取第二行
% echo ${"$(<test.txt)"[(f)2]}
bb
```

14.2.4 读取文件到数组

读取文件内容到数组中，每行是数组的一个元素：

```
% array=(${(f)"$(<test.txt)"})
```

14.2.5 读取指定数量的字符

有时我们需要按字节数来读取文件内容，而不是按行读取。

```
% cat test.txt
1234567890
# -k5 是只最多读取 5 个字节，-u 0 是从 fd 0 读取，不然会卡住
% read -k 5 -u 0 str <test.txt
% echo $str
12345
```

14.2.6 向文件中间插入内容

有时我们会遇到比较麻烦的场景，在某个文件中间插入一些内容，而前后的内容保持不变。

Zsh 并没有直接提供这样的功能，但我们可以迂回实现。

```
% echo -n 1234567890 > test.txt
# 先全部读进来
% str=$(<test.txt)
# 截断文件
% truncate -s 5 test.txt
# 插入内容
% echo -n abcde >> test.txt
# 将后半部分文件追加回去
% echo -n $str[6,-1] >> test.txt
% cat test.txt
12345abcde67890
```

但如果比较比较大的话，就不能将整个文件全部读进来，可以先在循环里用 read -k num 一次读固定数量的字符，然后写入一个中间文件，然后再 truncate 原文件，插入内容。最后再 cat 中间文件 >> 原文件追加原来的后半部分内容即可。

另外这种从文件中间写入或者读取内容的场景，都可以使用 dd 命令实现，可以自行搜索 dd 命令的用法。

14.3 总结

本文比较详细地介绍了各种读写文件的方法，基本可以覆盖常用的场景。

CHAPTER 15

15 进程与作业控制

通常情况 zsh 脚本是在一个进程中（并且单线程）执行的，但有时我们需要并行执行一些代码，因为现在的 CPU 基本都是多核的，这样可以加快运行速度。这就涉及到进程与作业控制。这里不讲进程的概念。

15.1 在子进程中执行代码

之前我们提到过，小括号中的代码是在子进程中执行的：

```
% (sleep 1000 && echo good)

# 然后再另一个 zsh 里查看进程
% pstree | grep sleep
     `-tmux: server-+-zsh---zsh---sleep
```

里边有两个 zsh 进程。如果不加小括号的话：

```
% sleep 1000 && echo good

# 然后再另一个 zsh 里查看进程
% pstree | grep sleep
     `-tmux: server-+-zsh---sleep
```

就只有一个 zsh 进程。这说明使用小括号时，里边的代码是在子进程（一个新的 zsh 进程）执行的。但需要注意的时，如果括号里只有一个命令（比如 sleep 1000），那么并不会再开一个子进程来执行了。

那么在子进程里执行代码有什么意义呢？如果像上边那样放着前台运行，是没有什么意义。但我们可以把它放后台运行。

15.2 在后台运行进程

首先我们先看下怎么把单个程序放后台运行。

```
% sleep 1000 &
[1] 850
```

在 sleep 1000 后边加一个 &，就会把它放后台运行。然后会输出一行内容，[1] 是进程的作业（job）号，850 是进程号（PID）。我们可以继续运行别的命令，不需要等待 sleep 结束了。

jobs 命令可以查看当前在后台运行的所有作业：

```
% jobs
[1]  + running    sleep 1000

# -l 会输出进程号
% jobs -l
[1]  + 850 running    sleep 1000
```

fg 命令可以把后台的作业切换回前台：

```
# 然后会继续等待 sleep 运行
% fg
[1]  + running    sleep 1000
```

如果进程已经运行起来了，我们想再把它放到后台，可以这样：

```
# 回车后按 ctrl + z
% sleep 1000
^Z
zsh: suspended  sleep 1000
# 这时可以运行 jobs 看一下，sleep 是处于挂起状态的
% jobs
[1]  + suspended  sleep 1000
# 可以用 bg 让 sleep 恢复运行
% bg
[1]  + continued  sleep 1000
# 这样 sleep 就运行在后台了
```

（下页继续）

(续上页)

```
% jobs
[1]  + running    sleep 1000
```

其实 jobs、fg、bg 这些命令并不常用，大概了解下用法即可。比如现在在用 vim 编辑文件，文件还没有保存，但我想退到终端运行个命令，然后再回到 vim。可以按 ctrl + z 让 vim 挂起，然后运行命令，最后再运行 fg 让 vim 恢复。但通常我们可以启动多个终端模拟器，或者开一个新终端模拟器标签，或者用 tmux，没必要在一个 shell 里这么折腾。

15.3 在脚本中使用后台进程执行代码

那么回答之前的场景，要在后台进程里执行 sleep 1000 && echo good：

```
% {sleep 1000 && echo aa} &
```

这样大括号里的代码都会在后台进程里执行，脚本里可以继续写别的。如果做完了后需要再等大括号里边的代码运行。

```
#!/bin/zsh

{sleep 5 && echo p1} &
# $! 是上一个运行的后台进程的进程号
pid=$!
{sleep 10 && echo p2} &
echo aaa
# 要做的其他事情先做完
sleep 2
echo bbb
# wait 加进程号用来等待进程结束，类似 fg，但脚本中不能用 fg
wait $pid
echo ccc
```

结果：

```
% ./test.zsh
aaa
bbb
p1
ccc
# p2 是脚本运行完过几秒才输出的
% p2
```

这样我们就可以同时操作多个进程来为自己服务了。而进程之间的通信，可以用命名管道或者普通文件来做，也可以使用 socket 文件（Zsh 中有 zsh/net/socket 模块，使用它可以通过 socket 文件来通信。管道是单向的，而 socket 双向的，更灵活一些，后续我们会了解它的用法），或者使用网络通信（如果脚本分布在不同的机器，zsh 中有 zsh/net/tcp 模块，这样无需外部命令就可进行 tcp 通信，后续也会讲到它）。

15.4 信号

运行中的进程可以接受信号然后对信号做出响应。kill 命令用来给进程发送信号。

15（SIGTERM）是最常用的信号，也是 kill 不加参数的默认信号，用于终止一个进程。kill num 即可终止进程号是 num 的进程。但 15 信号可以被进程捕获，然后并不退出。如果要强行杀掉一个进程，可以用 9 信号（SIGKILL），它是进程无法捕获的，但这样的话进程正在做的事情会突然中断，可能会有严重的影响，所以通常情况不要使用 9 信号杀进程。

在脚本中捕获信号：

```
#!/bin/zsh

# SIGINT 是 2 信号，ctrl + c 会触发
TRAPINT() {
    # 处理一些退出前的善后工作
    sleep 333
}

sleep 1000
```

然后运行这个脚本，然后 ctrl + c，脚本没有退出，因为在执行 sleep 333，要再按一次才会退出。

在脚本中使用信号，通常是给其他进程发（主要是 15），而不是给自己发。在脚本中也很少需要捕获信号处理。信号相关的更多内容，以后可能会补充。

15.5 总结

本文大概讲了进程与作业控制相关内容，主要用于在脚本里使用多进程执行代码，而不是在终端里进行作业控制（因为很少需要这样做）。关于脚本中的多个进程如何配合的内容还需要继续完善。

CHAPTER 16

16 alias 和 eval 的用法

alias（别名）在 shell 中是非常常用的，它主要用于给命令起别名，简化输入。但主要用于交互场景，在脚本中基本用不到。eval 是一个非常强大的命令，它的功能是将字符串解析成代码再执行，但也会额外增加很多复杂性，非必要场景尽量少用。alias 和 eval 看起来好像没什么关系，但功能上有相似之处，所以放在一起讲。

16.1 alias

最典型的例子是将 ls -l 简化成 ll:

```
% alias ll='ls -l'
% ll
total 0
drwx------ 0 goreliu goreliu 512 Aug 31 13:55 tmux-1000
drwxr-xr-x 0 goreliu goreliu 512 Aug 31 13:37 yaourt-tmp-goreliu
```

alias 的效果相当于直接将字符串替换过来，比较好理解。

```
# 直接运行 alias，会列出所有的 alias
% alias
ll='ls -l'
lla='ls -F --color --time-style=long-iso -lA'
...
```

这样的 alias 只有在行首出现时，才会被解析。但 zsh 中还有一种功能更强大的全局 alias，不在行首也能被解析：

```
% alias -g G='| grep'

% ls G tmux
tmux-1000
```

但这样需要格外注意可能导致的副作用，比如我想创建一个名为 G 的文件：

```
% touch G
touch: missing file operand
Try 'touch --help' for more information.
Usage: grep [OPTION]... PATTERN [FILE]...
Try 'grep --help' for more information.
```

结果 G 被替换了，只能在 G 两边加引号。

如果全局 alias 没用好，可能导致灾难性的后果，比如误删重要文件（像把某个全局 alias 传给 rm 后，恰好删除了 alias 字符串中的某些文件），所以需要执行权衡后再使用，并且用的时候要多加注意。

16.2 eval

eval 的功能是将字符串作为代码来执行。看上去好像很简单，但实际涉及很复杂的内容，主要是符号转义导致的语义问题。

在 bash 中，eval 的一个重要的使用场景是将变量的值当变量名，然后取它的变量值，类似于 c 语言中指向变量的指针：

```
% str1=str2
% str2=abc
% eval echo \$$str1
abc
```

注意这里有一个 \ 和两个 $，原因是第二个 $ 是和平时一样，正常取 str1 的值的，而第一个 $ 需要转义，因为它要在 eval 执行的过程中取 str2 的值，不能现在就展开。

这个用法很容易出问题，而且可读性很差。幸好 zsh 中无需这么用，有更好的办法：

```
% str1=str2
% str2=abc
% echo ${(P)str1}
abc
```

(P) 专门用于这种场景，不需要再去转义 $。

此外 eval 有时也用来动态执行代码，比如一个脚本接受用户的输入，而这输入也是一段脚本代码，就可以用 eval 来运行它。但这种用法是极其危险的，因为脚本中可能有各种危险操作，而且 shell 的语法很灵活，很难通过静态扫描的方法判断是否有危险操作。不可靠的代码根本不应该去运行。即使一定要运行，也可以先写到文件里再运行，避免传过来的代码影响到自身的逻辑。

但也不是说 zsh 中就完全没有必要用 eval 了，在某些特别的场景（比如用于改造语法加语法糖）还是有用的。但如果要使用，就一定要注意它可能导致的副作用，利弊只能自己权衡了。eval 的具体用法，和 bash 中的基本没有区别，可以去网上搜索 bash eval 用法来了解，这里就不介绍了。

16.3 总结

本文简单介绍了 alias 的用法和 eval 的场景使用场景。alias 很简单，主要在.zshrc 里使用。eval 很复杂，非必要场景尽量避免使用。

CHAPTER 17

17 使用 socket 文件和 TCP 实现进程间通信

就像我之前提到的，zsh 脚本是可以直接使用 socket 文件（UNIX domain socket 所使用）或者 TCP 和其他进程通信的。如果进程都在本地，用 socket 文件效率更高些，并且不要占用端口，权限也更好控制。如果是在不同机器，可以使用 TCP。

17.1 Socket 文件

UNIX domain socket 是比管道更先进的进程通信方法，是全双工的方式，并且稳定性更好。但性能比管道差一些，不过一般性能瓶颈都不会出现在这里，不用考虑性能问题。而且在一个 socket 文件上可以建立多个连接，更容易管理。另外如果通信方式从 socket 文件改成 TCP，只需要修改很少的代码（建立和关闭连接的代码稍微改一下），而从管道改成 TCP 则要麻烦很多。

所以建议用 zsh 写进程交互脚本的话，直接使用 socket 文件，而不是命名管道（匿名管道就能满足需求的简单场景忽略不计）。

Socket 文件的用法：

```
# 监听连接端
# 首先要加载 socket 模块
% zmodload zsh/net/socket

% zsocket -l test.sock
% listenfd=$REPLY
# 此处阻塞等待连接
```

(下页继续)

(续上页)

```
% zsocket -a $listenfd
# 连接建立完成
% fd=$REPLY
% echo $fd
5

# 然后 $fd 就可读可写
% cat <&$fd
good
```

```
# 发起连接端
# 首先要加载 socket 模块
% zmodload zsh/net/socket

% zsocket test.sock
# 连接建立完成
% fd=$REPLY
% echo $fd
4

# 然后 $fd 就可读可写
% echo good >&$fd
```

连接建立后，怎么用就随意了。实际使用时，要判断 fd 看连接是否正常建立了。通常使用 socket 文件要比在网络环境使用 TCP 稳定性高很多，一般不会连接中断或者出其他异常。另外可以在 zsocket 后加 -v 参数，查看详细的信息（比如使用的 fd 号）。

关闭连接：

```
# 发起连接端
# fd 是之前存放 fd 号的变量，不需要加 $
% exec {fd}>&-

# 监听连接端
% exec {listenfd}>&-
% exec {fd}>&-
# 删除 socket 文件即可，如果下次再使用会重新创建，该文件不能重复使用
% rm test.sock
```

17.2 TCP

使用 TCP 连接的方式和使用 socket 文件基本一样。

```
# 监听连接端
# 首先要加载 tcp 模块
% zmodload zsh/net/tcp

% ztcp -l 1234
% listenfd=$REPLY
# 此处阻塞等待连接
% ztcp -a $listenfd
# 连接建立完成
% fd=$REPLY
% echo $fd
3

# 然后 $fd 就可读可写
% cat <&$fd
good
```

```
# 发起连接端
# 首先要加载 tcp 模块
% zmodload zsh/net/tcp

% ztcp 127.0.0.1 1234
# 连接建立完成
% fd=$REPLY
% echo $fd
3

# 然后 $fd 就可读可写
% echo good >&$fd
```

关闭连接:

```
# 发起连接端
# fd 是之前存放 fd 号的变量
% ztcp -c $fd

# 监听连接端
```

(下页继续)

(续上页)

```
% ztcp -c $listenfd
% ztcp -c $fd
```

17.3 程序样例

recv_tcp，监听指定端口，并输出发送过来的消息。使用方法：recv_tcp 端口

```
#!/bin/zsh

zmodload zsh/net/tcp

(($+1)) || {
    echo "Usage: ${0:t} port"
    exit 1
}

ztcp -l $1
listenfd=$REPLY

[[ $listenfd == <-> ]] || exit 1

while ((1)) {
    ztcp -a $listenfd
    fd=$REPLY
    [[ $fd == <-> ]] || continue

    cat <&$fd
    ztcp -c $fd
}
```

send_tcp，用来向指定机器的指定端口发一条消息。使用方法：send_tcp 机器名端口消息（机器名可选，如果没有则发到本机，消息可以包含空格）

```
#!/bin/zsh

zmodload zsh/net/tcp

(($# >= 2)) || {
    echo "Usage: ${0:t} [hostname] port message"
```

(下页继续)

(续上页)

```
    exit 1
}

if [[ $1 == <0-65535> ]] {
    ztcp 127.0.0.1 $1
} else {
    ztcp $1 $2
    shift
}

fd=$REPLY
[[ "$fd" == <-> ]] || exit 1

echo ${*[2,-1]} >&$fd
ztcp -c $fd
```

17.4 总结

本文介绍了使用 socket 文件或者 TCP 来实现两个脚本之间通信的方法。

CHAPTER 18

18 更多内置模块的用法

除了 zsh/mathfunc、zsh/net/socket、zsh/net/tcp，zsh 还内置了一些其他的内置模块。本文简单讲几个比较常用的模块。

18.1 模块的使用方法

```
# 使用 zmodload 加模块名来加载模块
% zmodload zsh/mathfunc

# 如果不加参数，可以查看现在已经加载了的模块
% zmodload
zsh/complete
zsh/complist
zsh/computil
zsh/main
zsh/mathfunc
zsh/parameter
zsh/stat
zsh/zle
zsh/zutil

# 加 -u 参数可以卸载模块
```

(下页继续)

(续上页)

```
% zmodload -u zsh/mathfunc

# 还有其他参数，可以补全查看帮助，不详细介绍了
% zmodload -<tab>
 -- option --
-A  -- create module aliases
-F  -- handle features
-I  -- define infix condition names
-L  -- output in the form of calls to zmodload
-P  -- array param for features
-R  -- remove module aliases
-a  -- autoload module
-b  -- autoload module for builtins
-c  -- autoload module for condition codes
-d  -- list or specify module dependencies
-e  -- test if modules are loaded
-f  -- autoload module for math functions
-i  -- suppress error if command would do nothing
-l  -- list features
-m  -- treat feature arguments as patterns
-p  -- autoload module for parameters
-u  -- unload module
```

18.2 日期时间相关模块

我们知道使用 date 命令可以查看当前时间，也可以用来做日期时间的格式转换。但如果脚本里需要频繁地读取或者处理时间（比如打日志的时候，每一行加一个时间戳），那么调用 date 命令的资源消耗就太大了。Zsh 的 zsh/datetime 模块提供和 date 命令类似的功能。

```
% zmodload zsh/datetime

# 输出当前时间戳（从 1970 年年初到现在的秒数），和 date +%s 一样
% echo $EPOCHSECONDS
1504231297

# 输出高精度的当前时间戳，浮点数
% echo $EPOCHREALTIME
1504231373.9913284779
```

(下页继续)

(续上页)

```
# 输出当前时间戳的秒和纳秒部分，是一个数组
# 可以用 epochtime[1] 和 epochtime[2] 分别读取
% echo $epochtime
1504231468 503125900

# 安装指定格式输出当前时间，和 date +%... 效果一样
# 格式字符串可以 man date 或者 man strftime 查看
% strftime "%Y-%m-%d %H:%M:%S (%u)" $EPOCHSECONDS
2017-09-01 10:06:47 (5)

# 如果加了 -s str 参数，将指定格式的时间存入 str 变量而不输出
% strftime -s str "%Y-%m-%d %H:%M:%S (%u)" $EPOCHSECONDS
% echo $str
2017-09-01 10:10:58 (5)

# 如果加了 -r 参数，从指定的时间字符串反解出时间戳，之前操作的逆操作
# 也可以同时加 -s 参数来讲结果存入变量
% strftime -r "%Y-%m-%d %H:%M:%S (%u)" "2017-09-01 10:10:58 (5)"
1504231858
```

这基本覆盖了 date 的常用功能，而运行速度比 date 命令快很多。

18.3 读写 gdbm 数据库

有时我们的脚本需要将某些数据持久化到本地文件，但像哈希表之类的数据，如果存放到普通文件里，载入和保存的资源消耗都比较大，而且如果脚本突然异常退出，数据会丢失。而且某些时候，我们可能需要操作一个巨大的哈希表，并不能全部将它载入到内存中。那么我们可以使用 gdbm 数据库文件。

Gdbm 是一个很轻量的 Key-Value 数据库，可以认为它就像一个保存在文件里的哈希表。Zsh 的 zsh/db/gdbm 模块可以很方便地读写 gdbm 数据库文件。

```
% zmodload zsh/db/gdbm

# 声明数据库文件对应的哈希表
% local -A sampledb
# 创建数据库文件，文件名是 sample.gdbm，对应 sampledb 哈希表
# 如果该文件已经存在，则会继续使用该文件
% ztie -d db/gdbm -f sample.gdbm sampledb

# 然后正常使用 sampledb 哈希表即可，数据会同步写入到数据库文件中
```

(下页继续)

(续上页)

```
% sampledb[k1]=v1
% sampledb+=(k2 v2 k3 v3)
% echo ${(kv)sampledb}
k1 v1 k2 v2 k3 v3

# 获取数据库文件路径
% zgdbmpath sampledb
% echo $REPLY
/home/goreliu/sample.gdbm

# 释放数据库文件
% zuntie -u sampledb



# 也可以用只读的方式加载数据库文件
% ztie -r -d db/gdbm -f sample.gdbm sampledb
# 但这样的话，需要用 zuntie -u 释放数据库文件
% zuntie -u sampledb
```

如果数据量比较大，或者有比较特别的需求，要先了解下 gdbm 是否符合自己的场景再使用。

18.4 调度命令

有时我们需要在未来的某个时刻运行某一个命令。虽然也可以 sleep 然后运行，但这样要多占两个进程，而且不好控制（比如要取消运行其中的某一个）。Zsh 的 zsh/sched 模块用于调度命令的运行。

```
% zmodload zsh/sched

# 5 秒后运行 ls 命令
% sched +5 ls
# 可以随便做些别的
% date
Fri Sep  1 10:36:16 DST 2017
# 五秒后，ls 命令被运行
git  sample.gdbm  tmp

# 不加参数可以查看已有的待运行命令
% sched
  1 Fri Sep  1 21:16:05 date
```

(下页继续)

(续上页)

```
  2 Fri Sep  1 21:16:30 date
  3 Fri Sep  1 21:17:12 date

# -n 可以去除第 n 个待运行命令
% sched -2
% sched
  1 Fri Sep  1 21:16:05 date
  2 Fri Sep  1 21:17:12 date
```

18.5 底层的文件读写命令

有时我们可能需要更精细地操作文件，zsh 提供了一个 zsh/system 模块，里边包含一些底层的文件读写命令（对应 open、read、write 等系统调用）。使用这些函数，可以更精细地控制文件的读写，比如控制每次读写的数据量、从中间位置读写、上文件锁等等。这些命令的用法比较复杂，参数也比较多，这里就不列出了。如果需要使用，可以 man zshmodules 然后搜索 zsh/system 查看文档。

函数列表：sysopen、sysread、sysseek、syswrite、zsystem flock、systell、syserror

18.6 其他模块

其余的在脚本编写方面可能用的上的模块还有：

zsh/pcre（使用 pcre 正则表达式库，默认使用的是 POSIX regex 库）

zsh/stat（内部的 stat 命令，可用于取代 stat 命令）

zsh/zftp（内置的 ftp 客户端）

zsh/zprof（Zsh 脚本的性能追踪工具）

zsh/zpty（操作 pty 的命令）

zsh/zselect（select 系统调用的封装）

可以用 man zshmodules 查看。

18.7 自己编写模块

如果因为性能等因素，要自己写 zsh 模块来调用，也是比较方便的。Zsh 的源码中 Src/Modules 是模块目录，里边有一个实例模块 example（example.c 和 example.mdd 文件）。可以参考代码编写自己的模块，难度并不是很大。

18.8 总结

本文介绍了几个比较常用的 zsh 内置模块，以后可能继续补充更多模块的用法。

CHAPTER 19

19 脚本实例讲解

本文将讲解一些比较简单的 zsh 脚本实例。

19.1 实例一：复制一个目录的目录结构

功能：

将一个目录及它下边的所有目录复制到另一个目录中（即创建同名目录），但不复制目录下的其他类型文件。

例子：

```
src 的目录结构：

src
   a
   b
      1.txt
      2
          3.txt
   c.txt
   d
   e f
      g
          4.txt
```

(下页继续)

(续上页)

```
  g h -> e f

要构造一个 dst 目录，只包含 src 下的目录，内容如下：

dst
  src
      a
      b
         2
      d
      e f
          g
```

思路：

1. 首先需要先将 src 目录下的目录名筛选出来，可以用 **/*(/) 匹配。
2. 然后用 mkdir -p 在 dst 目录中创建对应的目录。

```
# 参数 1：src 目录
# 参数 2：待创建的 dst 目录

#!/bin/zsh

for i ($1/**/*(/)) {
    # -p 参数是递归创建目录，这样不用考虑目录的创建顺序
    mkdir -p $2/$i
}
```

19.2 实例二：寻找不配对的文件

功能：

需要当前目录下有一些.txt 和.txt.md5sum 的文件，需要寻找出没有对应的.md5sum 文件的.txt 文件。（实际的场景是寻找已经下载完成的文件，未下载完的文件都对应某个带后缀的文件。）

例子：

```
当前目录的所有文件：

aa.txt
bb.txt
```

(下页继续)

(续上页)

```
bb.txt.md5sum
cc dd.txt
cc dd.txt.md5sum
ee ff.txt.md5sum
gg.txt
hh ii.txt

需要找出没有对应 .md5sum 的 .txt 文件:
aa.txt
gg.txt
hh ii.txt
```

思路:

1. 找到所有.md5sum 文件，然后把文件名中的.md5sum 去掉，即为那些需要排除的.txt 文件（a）。
2. 所有的文件，排除掉.m5sum 文件，再排除掉 a，即结果。

实现:

```
#!/bin/zsh

all_files=(*)
bad_files=(*.md5sum)
bad_files+=(${bad_files/.md5sum})

# 数组差集操作
echo ${all_files:|bad_files}
```

19.3 实例三：用 sed 批量重命名文件

功能:

用形如 sed 命令的用法批量重命名文件。

例子:

```
# 实现 renamex 命令，接受的第一个参数为 sed 的主体参数，其余参数是文件列表
# 效果是根据 sed 对文件名的修改重命名这些文件

% tree
.
```

(下页继续)

(续上页)

```
   aaa_aaa.txt
   aaa.txt
   ccc.txt
   xxx
       aaa bbb.txt
       bbb ccc.txt

% renamex s/aaa/bbb/g **/*
'aaa_aaa.txt' -> 'bbb_bbb.txt'
'aaa.txt' -> 'bbb.txt'
'xxx/aaa bbb.txt' -> 'xxx/bbb bbb.txt'

% tree
.
   bbb_bbb.txt
   bbb.txt
   ccc.txt
   xxx
       bbb bbb.txt
       bbb ccc.txt
```

思路:

1. 要找出所有的文件名，然后用 sed 替换成新文件名。
2. 如果文件名有变化，用 mv 命令移动

实现:

```
#!/bin/zsh

(($+2)) || {
    echo 'Usage: renamex s/aaa/bbb/g *.txt'
    return
}

for name ($*[2,-1]) {
    local new_name="$(echo $name | sed $1)"
    [[ $name == $new_name ]] && continue
    mv -v $name $new_name
}
```

19.4 实例四：根据文件的 md5 删除重复文件

功能：

删除当前目录以及子目录下所有的重复文件（根据 md5 判断，不是很严谨）。

思路：

1. 用 md5sum 命令计算所有文件的 md5。
2. 使用哈希表判断 md5 是否重复，删除哈希表里已经有 md5 的后续文件。

实现：

```
#!/bin/zsh

# D 是包含以 . 开头的隐藏文件
local files=("${(f)$(md5sum **/*(.D))}")
local files_to_delete=()
local -A md5s

for i ($files) {
    # 取前 32 位，即 md5 的长度
    local md5=$i[1,32]

    if (($+md5s[$md5])) {
        # 取 35 位之后的内容，即文件路径，md5 后边有两个空格
        files_to_delete+=($i[35,-1])
    } else {
        md5s[$md5]=1
    }
}

(($#files_to_delete)) && rm -v $files_to_delete
```

19.5 实例五：转换 100 以内的汉字数字为阿拉伯数字

功能：

转换 100 以内的汉字数字为阿拉伯数字，如六十八转换成 68。

思路：

1. 建一个哈希表存放汉字与数字的对应关系。
2. 比较麻烦的是“十”，在不同的位置，转换成的数字不同，需要分别处理。

实现:

```
#!/bin/zsh

local -A table=(
零 0
一 1
二 2
三 3
四 4
五 5
六 6
七 7
八 8
九 9
)

local result

if [[ $1 == 十 ]] {
    result= 一零
} elif [[ $1 == 十 * ]] {
    result=${1/十/一}
} elif [[ $1 == * 十 ]] {
    result=${1/十/零}
} elif [[ $1 == * 十 * ]] {
    result=${1/十}
} else {
    result=$1
}

for i ({1..$#result}) {
    result[i]=$table[$result[i]]

    if [[ -z $result[i] ]] {
        echo error
        return 1
    }
}

echo $result
```

(下页继续)

(续上页)

运行结果:

```
% ./convert 一
1
% ./convert 十
10
% ./convert 十五
15
% ./convert 二十
20
% ./convert 五十六
56
% ./convert 一百
error
```

19.6 实例六：为带中文汉字数字的文件名重命名成以对应数字开头

功能:

见下边例子。

例子:

当前目录有如下文件:

```
Zsh-开发指南（第一篇-变量和语句）.md
Zsh-开发指南（第七篇-数值计算）.md
Zsh-开发指南（第三篇-字符串处理之转义字符和格式化输出）.md
Zsh-开发指南（第九篇-函数和脚本）.md
Zsh-开发指南（第二篇-字符串处理之常用操作）.md
Zsh-开发指南（第五篇-数组）.md
Zsh-开发指南（第八篇-变量修饰语）.md
Zsh-开发指南（第六篇-哈希表）.md
Zsh-开发指南（第十一篇-变量的进阶内容）.md
Zsh-开发指南（第十七篇-使用-socket-文件和-TCP-实现进程间通信）.md
Zsh-开发指南（第十三篇-管道和重定向）.md
Zsh-开发指南（第十九篇-脚本实例讲解）.md
Zsh-开发指南（第十二篇-[[-]]-的用法）.md
Zsh-开发指南（第十五篇-进程与作业控制）.md
```

(下页继续)

(续上页)

```
Zsh-开发指南（第十八篇-更多内置模块的用法）.md
Zsh-开发指南（第十六篇-alias-和-eval-的用法）.md
Zsh-开发指南（第十四篇-文件读写）.md
Zsh-开发指南（第十篇-文件查找和批量处理）.md
Zsh-开发指南（第四篇-字符串处理之通配符）.md

需要重命名成这样:

01_Zsh-开发指南（第一篇-变量和语句）.md
02_Zsh-开发指南（第二篇-字符串处理之常用操作）.md
03_Zsh-开发指南（第三篇-字符串处理之转义字符和格式化输出）.md
04_Zsh-开发指南（第四篇-字符串处理之通配符）.md
05_Zsh-开发指南（第五篇-数组）.md
06_Zsh-开发指南（第六篇-哈希表）.md
07_Zsh-开发指南（第七篇-数值计算）.md
08_Zsh-开发指南（第八篇-变量修饰语）.md
09_Zsh-开发指南（第九篇-函数和脚本）.md
10_Zsh-开发指南（第十篇-文件查找和批量处理）.md
11_Zsh-开发指南（第十一篇-变量的进阶内容）.md
12_Zsh-开发指南（第十二篇-[[-]]-的用法）.md
13_Zsh-开发指南（第十三篇-管道和重定向）.md
14_Zsh-开发指南（第十四篇-文件读写）.md
15_Zsh-开发指南（第十五篇-进程与作业控制）.md
16_Zsh-开发指南（第十六篇-alias-和-eval-的用法）.md
17_Zsh-开发指南（第十七篇-使用-socket-文件和-TCP-实现进程间通信）.md
18_Zsh-开发指南（第十八篇-更多内置模块的用法）.md
19_Zsh-开发指南（第十九篇-脚本实例讲解）.md
```

思路:

1. 首先需要写将汉字数字转成阿拉伯数字的函数。
2. 然后需要从文件名中截取汉字数字，然后转成阿拉伯数字。
3. 拼接文件名，然后移动文件。

实现:

```
#!/bin/zsh

# 转换数字的逻辑和上一个实例一样

local -A table=(
```

(下页继续)

(续上页)

```
零 0
一 1
二 2
三 3
四 4
五 5
六 6
七 7
八 8
九 9
)

convert() {
    local result

    if [[ $1 == 十 ]] {
        result= 一零
    } elif [[ $1 == 十 * ]] {
        result=${1/十/一}
    } elif [[ $1 == * 十 ]] {
        result=${1/十/零}
    } elif [[ $1 == * 十 * ]] {
        result=${1/十}
    } else {
        result=$1
    }

    for i ({1..$#result}) {
        result[i]=$table[$result[i]]

        if [[ -z $result[i] ]] {
            echo error
            return 1
        }
    }

    echo $result
}

for i (Zsh*.md) {
```

(下页继续)

(续上页)

```
    # -Z 2 是为了在前边补全一个 0
    # 把文件名“第”之前和“篇”之后的全部去除
    local -Z 2 num=$(convert ${${i#* 第}% 篇 *})
    mv -v $i ${num}_$i
}
```

19.7 实例七：统一压缩解压工具

功能：

Linux 下常用的压缩、归档格式众多，参数各异，写一个用法统一的压缩解压工具，用于创建、解压 .zip .7z .tar .tgz .tbz2 .txz .tar.gz .tar.bz2 .tar.xz .cpio .ar .gz .bz2 .xz 等文件。(类似 atool，但 atool 很久没更新了，一些新的格式不支持，没法定制。而且是用 perl 写的，很难看懂。所以还是决定自己写一个，只覆盖 atool 的一部分常用功能。)

例子：

```
# a 用于创建压缩文件
% a a.tgz dir1 file1 file2
dir1/
file1
file2

# al 用于列出压缩文件中的文件列表
% al a.tgz
drwxr-xr-x goreliu/goreliu   0 2017-09-13 11:23 dir1/
-rw-r--r-- goreliu/goreliu   3 2017-09-13 11:23 file1
-rw-r--r-- goreliu/goreliu   3 2017-09-13 11:23 file2

# x 用于解压文件
% x a.tgz
dir1/
file1
file2
a.tgz  ->  a

# 如果解压后的文件名或目录名中当前目录下已经存在，则解压到随机目录
% x a.tgz
dir1/
file1
```

(下页继续)

(续上页)

```
file2
a.tgz  ->  /tmp/test/x-c4I
```

思路：

1. 压缩文件时，根据传入的文件名判断压缩文件的格式。
2. 解压和查看压缩文件内容时，根据传入的文件名和 `file` 命令结果判断压缩文件的格式。
3. 为了复用代码，多个命令整合到一个文件，然后 `ln -s` 成多个命令。

实现：

```
#!/bin/zsh

get_type_by_name() {
    case $1 {
        (*.zip|*.7z|*.jar)
        echo 7z
        ;;

        (*.rar|*.iso)
        echo 7z_r
        ;;

        (*.tar|*.tgz|*.txz|*.tbz2|*.tar.*)
        echo tar
        ;;

        (*.cpio)
        echo cpio
        ;;

        (*.cpio.*)
        echo cpio_r
        ;;

        (*.gz)
        echo gz
        ;;

        (*.xz)
        echo xz
```

(下页继续)

(续上页)

```
        ;;

        (*.bz2)
        echo bz2
        ;;

        (*.lzma)
        echo lzma
        ;;

        (*.lz4)
        echo lz4
        ;;

        (*.ar)
        echo ar
        ;;

        (*)
        return 1
        ;;
    }
}

get_type_by_file() {
    case $(file -bz $1) {
        (Zip *|7-zip *)
        echo 7z
        ;;

        (RAR *)
        echo 7z_r
        ;;

        (POSIX tar *|tar archive)
        echo tar
        ;;

        (*cpio archive*)
        echo cpio
```

(下页继续)

(续上页)

```
        ;;

        (*gzip *)
        echo gz
        ;;

        (*XZ *)
        echo xz
        ;;

        (*bzip2 *)
        echo bz2
        ;;

        (*LZMA *)
        echo lzma
        ;;

        (*LZ4 *)
        echo lz4
        ;;

        (current ar archive)
        echo ar
        ;;

        (*)
        return 1
        ;;
    }
}


(($+commands[tar])) || alias tar=bsdtar
(($+commands[cpio])) || alias cpio=bsdcpio

case ${0:t} {
    (a)

    (($#* >= 2)) || {
```

(下页继续)

(续上页)

```
    echo Usage: $0 target files/dirs
    return 1
}

case $(get_type_by_name $1) {
    (7z)
    7z a $1 $*[2,-1]
    ;;

    (tar)
    tar -cavf $1 $*[2,-1]
    ;;

    (cpio)
    find $*[2,-1] -print0 | cpio -H newc -0ov > $1
    ;;

    (gz)
    gzip -cv $*[2,-1] > $1
    ;;

    (xz)
    xz -cv $*[2,-1] > $1
    ;;

    (bz2)
    bzip2 -cv $*[2,-1] > $1
    ;;

    (lzma)
    lzma -cv $*[2,-1] > $1
    ;;

    (lz4)
    lz4 -cv $2 > $1
    ;;

    (ar)
    ar rv $1 $*[2,-1]
    ;;
```

(下页继续)

(续上页)

```
    (*)
    echo $1: error
    return 1
    ;;
}
;;

(al)

(($#* >= 1)) || {
    echo Usage: $0 files
    return 1
}

for i ($*) {
    case $(get_type_by_name $i || get_type_by_file $i) {
        (7z|7z_r)
        7z l $i
        ;;

        (tar)
        tar -tavf $i
        ;;

        (cpio|cpio_r)
        cpio -itv < $i
        ;;

        (gz)
        zcat $i
        ;;

        (xz)
        xzcat $i
        ;;

        (bz2)
        bzcat $i
        ;;
```

(下页继续)

(续上页)

```
            (lzma)
            lzcat $i
            ;;

            (lz4)
            lz4cat $i
            ;;

            (ar)
            ar tv $i
            ;;

            (*)
            echo $i: error
            ;;
        }
    }
    ;;

    (x)

    (($#* >= 1)) || {
        echo Usage: $0 files
        return 1
    }

    for i ($*) {
        local outdir=${i%.*}

        [[ $outdir == *.tar ]] && {
            outdir=$outdir[1, -5]
        }

        if [[ -e $outdir ]] {
            outdir="$(mktemp -d -p $PWD x-XXX)"
        } else {
            mkdir $outdir
        }
```

(下页继续)

(续上页)

```
        case $(get_type_by_name $i || get_type_by_file $i) {
            (7z|7z_r)
            7z x $i -o$outdir
            ;;

            (tar)
            tar -xavf $i -C $outdir
            ;;

            (cpio|cpio_r)
            local file_path=$i
            [[ $i != /* ]] && file_path=$PWD/$i
            cd $outdir && cpio -iv < $file_path && cd ..
            ;;

            (gz)
            zcat $i > $outdir/$i[1,-4]
            ;;

            (xz)
            xzcat $i > $outdir/$i[1,-4]
            ;;

            (bz2)
            bzcat $i > $outdir/$i[1,-5]
            ;;

            (lzma)
            lzcat $i > $outdir/$i[1,-6]
            ;;

            (lz4)
            lz4cat $i > $outdir/$i[1,-5]
            ;;

            (ar)
            local file_path=$i
            [[ $i != /* ]] && file_path=$PWD/$i
            cd $outdir && ar x $file_path && cd ..
            ;;
```

(下页继续)

(续上页)

```
            (*)
            echo $i: error
            ;;
        }

        local files=$(ls -A $outdir)

        if [[ -z $files ]] {
            rmdir $outdir
        } elif [[ -e $outdir/$files && ! -e $files ]] {
            mv -v $outdir/$files . && rmdir $outdir
            echo $i " -> " $files
        } else {
            echo $i " -> " $outdir
        }
    }
    ;;

    (*)
    echo error
    return 1
    ;;
}
```

19.8 实例八：方便并发运行命令的工具

功能：

我们经常会遇到在循环里批量处理文件的场景（比如将所有 jpg 图片转换成 png 图片），那么就会遇到一个麻烦：如果在前台处理文件，那同一时间只能处理一个，效率太低；如果在后台处理文件，那么瞬间就会启动很多个进程，占用大量资源，系统难以承受。我们希望的是在同一时间最多同时处理固定数量（比如 10 个）的文件，如果已经达到了这个数量，那么就先等一会，直到有退出的进程后再继续。`parallel` 命令中在一定程度上能满足这个需求，但用起来太麻烦。

例子：

```
# rr 是一个函数（可放在 .zshrc 中），直接 rr 加命令即可使用
# 命令中支持变量、重定向等等，格式上和直接输入命令没有区别（不支持 alias）
% rr sleep 5
```

(下页继续)

(续上页)

```
[4] 5031
% rr sleep 5
[5] 5032

# 如果不加参数，则显示当前运行的进程数、最大进程并发数和运行中进程的进程号
# 默认最大进程并发数是 10
% rr
running/max: 2/10
pid: 5031 5032
# 5 秒之后，运行结束
% rr
running/max: 0/10



# 用 -j 来指定最大进程并发数，指定一次即可，如需修改可再次指定
# 可以只调整最大进程并发数而不运行命令
% rr -j2 sleep 10
[4] 5035
% rr sleep 10
[5] 5036

# 超过了最大进程并发数，等待，并且每一秒检查一次是否有进程退出
# 如果有进程退出，则继续在后台运行当前命令
% rr sleep 10
running/max: 2/2, wait 1s ...
pid: 5035 5036
running/max: 2/2, wait 1s ...
pid: 5035 5036
[4]  - done       $*
[4] 5039



# 实际使用场景，批量将 jpg 图片转换成 png 图片，gm 是 graphicsmagick 中的命令
# 转换图片格式比较耗时，顺序执行的话需要很久
% for i (*.jpg) { rr gm convert $i ${i/jpg/png} }
[4] 5055
[5] 5056
[6] 5057
[7] 5058
[8] 5059
```

(下页继续)

(续上页)

```
[9] 5060
[10] 5061
[11] 5062
[12] 5063
[13] 5064
running/max: 10/10, wait 1s ...
pid: 5060 5061 5062 5063 5064 5055 5056 5057 5058 5059
running/max: 10/10, wait 1s ...
pid: 5060 5061 5062 5063 5064 5055 5056 5057 5058 5059
[11]    done       $*
[5]    done       $*
[5] 5067
[12]    done       $*
[11] 5068
[6]    done       $*
[6] 5069
[12] 5070
running/max: 10/10, wait 1s ...
pid: 5070 5060 5061 5064 5055 5067 5068 5069 5058 5059
[13]  - done       $*
[4]    done       $*
[4] 5072
[13] 5073
running/max: 10/10, wait 1s ...
pid: 5070 5060 5072 5061 5073 5067 5068 5069 5058 5059
[5]    done       $*
[6]    done       $*
[5] 5075
[6] 5076
running/max: 10/10, wait 1s ...
pid: 5070 5060 5072 5061 5073 5075 5076 5068 5058 5059
...
```

思路：

1. 需要在全局变量里记录最大进程并发数和当前运行的进程（哈希表）。
2. 每运行一个进程，将对应的进程号放入哈希表中。
3. 如果当前运行进程数达到最大进程并发数，则循环检查哈希表里的进程是否退出。

实现：

```
rr() {
    (($+max_process)) || typeset -g max_process=10
    (($+running_process)) || typeset -gA running_process=()

    [[ $1 == -j<1-> ]] && {
        max_process=${1[3,-1]}
        shift
    }

    (($# == 0)) && {
        for i (${(k)running_process}) {
            [[ -e /proc/$i ]] || unset "running_process[$i]"
        }

        echo "running/max: $#running_process/$max_process"
        (($#running_process > 0)) && echo "pid: ${(k)running_process}"
        return
    }

    while ((1)) {
        local running_process_num=$#running_process

        if (($running_process_num < max_process)) {
            $* &
            running_process[$!]=1
            return
        }

        for i (${(k)running_process}) {
            [[ -e /proc/$i ]] || unset "running_process[$i]"
        }

        (($#running_process == $running_process_num)) && {
            echo "running/max: $running_process_num/$max_process, wait 1s ..."
            echo "pid: ${(k)running_process}"
            sleep 1
        }
    }
}
```

使用 inotifywait 的版本（无需循环 sleep 等待）：

```
rr() {
    (($+max_process)) || typeset -gi max_process=10
    (($+running_process)) || typeset -gA running_process=()

    while {getopts j:h arg} {
        case $arg {
            (j)
            ((OPTARG > 0)) && max_process=$OPTARG
            ;;

            (h)
            echo "Usage: $0 [-j max_process] [cmd] [args]"
            return
            ;;
        }
    }

    shift $((OPTIND - 1))

    (($# == 0)) && {
        for i (${(k)running_process}) {
            [[ -e $i ]] || unset "running_process[$i]"
        }

        echo "running/max: $#running_process/$max_process"
        (($#running_process > 0)) && echo "pids:" ${${(k)running_process/\/proc\/}/\/exe}
        return 0
    }

    while ((1)) {
        local running_process_num=$#running_process

        if (($running_process_num < max_process)) {
            $* &
            running_process[/proc/$!/exe]=1
            return
        }

        for i (${(k)running_process}) {
            [[ -e $i ]] || unset "running_process[$i]"
```

(下页继续)

(续上页)

```
        }

        (($#running_process == $running_process_num)) && {
            echo "wait $running_process_num pids:" ${${(k)running_process/\/proc\/}/\/
↪exe}
            inotifywait -q ${(k)running_process}
        }
    }
}
```

19.9 实例九：批量转换图片格式

功能：

将当前目录及子目录的所有常见图片格式转换成 jpg 格式（jpg 格式也要转换一遍，可以减少文件体积），然后删除原图片。需要用 5 个并发进程来处理。注意避免仅扩展名不同的文件互相覆盖的情况。

例子：

```
% tree
.
  mine
      信.txt
      第一封信.jpg
      第二封信.JPG
  搞笑
      卖萌.GIF
      猫吃鱼.gif
      猫抢东西吃.gif
  素材
      104 按键模板.jpg
      104 按键模板.psd
      ahk
          ahk_bg.jpg
          ahk_home_logo.jpg
          ahk_home_logo.txt
          ahk_home_qr.jpg
          ahk_home_qr_small.jpg
          ahk_logo.png
      stp_fc_cw_png_pk
```

(下页继续)

(续上页)

```
          HD.PNG
          newimage.png
          nshd.PNG
          std.png
      地球.jpg
      星系.JPEG
      木纹 背景.GIF
      木纹 背景.jpeg
      木纹 背景.jpg

5 directories, 23 files

% alltojpg
running/max: 0/5
running: 5, wait 1.0000000000s ...
pid: 5953 5954 5955 5956 5957
running: 5, wait 1.0000000000s ...
pid: 5965 5966 5967 5968 5959

% tree
.
  mine
      信.txt
      第一封信.jpg
      第二封信.jpg
  搞笑
      卖萌_g.jpg
      猫吃鱼_g.jpg
      猫抢东西吃_g.jpg
  素材
      104 按键模板.jpg
      104 按键模板.psd
      ahk
          ahk_bg.jpg
          ahk_home_logo.jpg
          ahk_home_logo.txt
          ahk_home_qr.jpg
          ahk_home_qr_small.jpg
          ahk_logo_p.jpg
      stp_fc_cw_png_pk
```

(下页继续)

(续上页)

```
          HD_p.jpg
          newimage_p.jpg
          nshd_p.jpg
          std_p.jpg
      地球.jpg
      星系_e.jpg
      木纹 背景_e.jpg
      木纹 背景_g.jpg
      木纹 背景.jpg

5 directories, 23 files
```

思路：

1. 并发运行命令的方法见上一个实例。
2. 转换图片格式用 `gm convert` 命令（graphicsmagick 中）或者 `convert` 命令（imagemagick 中）。
3. 常见的图片文件扩展名有 `jpg jpeg png gif`，另外可能是大写的扩展名。
4. 为了避免类似 `a.gif` 覆盖 `a.jpg` 的情况，为不同的文件格式添加不同后缀，这样可以无需检查是否有同名文件，加快速度。

实现：

```
#!/bin/zsh

# rr 是上一个实例中的代码
rr() {
    (($+max_process)) || typeset -gi max_process=10
    (($+running_process)) || typeset -gA running_process=()

    while {getopts j:h arg} {
        case $arg {
            (j)
            ((OPTARG > 0)) && max_process=$OPTARG
            ;;

            (h)
            echo "Usage: $0 [-j max_process] [cmd] [args]"
            return
            ;;
        }
```

(下页继续)

(续上页)

```
    }

    shift $((OPTIND - 1))

    (($# == 0)) && {
        for i (${(k)running_process}) {
            [[ -e $i ]] || unset "running_process[$i]"
        }

        echo "running/max: $#running_process/$max_process"
        (($#running_process > 0)) && echo "pids:" ${${(k)running_process/\/proc\/}/\/exe}
        return 0
    }

    while ((1)) {
        local running_process_num=$#running_process

        if (($running_process_num < max_process)) {
            $* &
            running_process[/proc/$!/exe]=1
            return
        }

        for i (${(k)running_process}) {
            [[ -e $i ]] || unset "running_process[$i]"
        }

        (($#running_process == $running_process_num)) && {
            echo "wait $running_process_num pids:" ${${(k)running_process/\/proc\/}/\/
↪exe}
            inotifywait -q ${(k)running_process}
        }
    }
}


# JPG 作为中间扩展名
rename .JPG .jpg **/*.JPG

# 设置进程并发数为 5
```

(下页继续)

(续上页)

```
rr -j5

for i (**/*.(jpg|png|PNG|jpeg|JPEG|gif|GIF)) {
    rr gm convert $i $i.JPG
}

# 等所有操作结束
wait

# 删除原文件
rm **/*.(jpg|png|PNG|jpeg|JPEG|gif|GIF)

# 避免覆盖同名文件
rename .jpg.JPG .jpg **/*.JPG
rename .png.JPG _p.jpg **/*.JPG
rename .PNG.JPG _p.jpg **/*.JPG
rename .jpeg.JPG _e.jpg **/*.JPG
rename .JPEG.JPG _e.jpg **/*.JPG
rename .gif.JPG _g.jpg **/*.JPG
rename .GIF.JPG _g.jpg **/*.JPG
```

19.10 总结

本文讲解了几个比较实用的 zsh 脚本，后续可能会补充更多个。

19.11 更新历史

2017.09.13：新增“实例七”、“实例八”和“实例九”。

2017.10.09：“示例八”和“示例九”中，新增使用 inotifywait 的 rr 函数。

CHAPTER 20

20 代码风格

因为 shell 脚本语法比较灵活，写 shell 脚本的开发者熟悉的编程语言也有较大差异，大家很容易写出风格迥异的代码出来。如果只有自己一个人用还好，如果是大家合作开发同一个项目，代码风格不同就会造成不小的麻烦。所以约定一个代码风格是很有必要的。

本文中的代码风格约定只是我的个人建议，可以根据自己的需求或者喜好来调整。本文的代码风格约定，在一定程度上也适用于 bash。

注意需要有丰富 shell 编程经验的人制定和维护代码风格约定，不然很容易无法执行或者流于形式而解决不了实际问题。代码风格约定不只需要约定代码怎么写，而且要说明为什么要这么写，不然容易因为难以服众而无法推广。

20.1 缩进

- 统一使用 4 个空格来缩进。

原因：

1. 要用空格而不是 tab。因为在终端上 `cat` `less` `diff` 等命令都将 tab 显示成 8 个空格的宽度，有些命令是不可配置的（即使可配置，要让所有机器配置同步也是件麻烦的事情）。如果自己在编辑器上配置 tab 为 4 个或者 2 个空格，那么就会和 `cat` `less` 等命令的显示方法不一致，会导致很多麻烦。
2. 8 个空格太长，缩进几次就会导致行太长，而 shell 脚本每行不宜过长。
3. 2 个空格的话，如果缩进比较频繁，看起来比较费劲。另外如果写代码时不小心多了或者少了一个空格，在某些场景，不看逻辑的话，就无法确定是多个一个还是少了一个，更容易导致他人错误的修改，或者代码越改越乱。

4. 对于 4 个空格也可能导致缩进层数多时行太长的问题，通过修改逻辑减少缩进层数或者折行的方法，而不是减少缩进的空格数量来解决。

20.2 每行代码最多字符数

- 非特殊场景，每行代码不超过 100 个字符。

原因：

1. 代码过长，阅读起来不方便，用 `diff` 之类工具对代码进行分析处理也不方便，所以需要约定最长字符数。
2. 经典的 80 个字符的约定，是受当时的输出设备限制而产生的标准，而现在的屏幕基本都是宽屏的，终端模拟器也都是可调大小的（而不是固定的 80x24）没必要削足适履迎合陈旧的标准，浪费屏幕空间。而且如果使用 80 个字符的约定，很容易遇到需要折行的情况，反而会导致可读性下降。
3. 如果一行超过了 100 个字符，通常说明逻辑太多，需要分行或者折行。
4. 某些特殊场景，比如显示一个 ASCII 字符组成的图片，会有一行超过 100 个字符的需求，所有不能严格执行每行必须不超过 100 个字符的约定。如果分行或者折行会不可避免地导致代码可读性下降，那么优先考虑可读性。

20.3 折行

- 在前一行尾部加一个空格和 \ 折行，折行后缩进一层（4 个空格）。
- 如果缩进的是一个文本块，可以使用对齐缩进，也可以使用 4 个空格的固定缩进。
- 如果是在 `aa && bb || cc`、`[[ ]]` 或者 `(( ))` 中折行，`&& ||` 放在下一行的行首。

原因：

1. 折行的缩进和普通的缩进都是为了体现代码的递进关系，没必要区分对待（比如折行缩进两层）。
2. 如果为了看起来美观，使用对齐缩进而不是固定缩进。那么因为每个人的审美不同，很容易产生不同的缩进方法，从而产生不必要的麻烦。但对文本块来说比较特殊，因为通常对齐缩进不会产生争议。
3. `&&` 和 `||` 在逻辑上属于后半个语句，在自然语言中也是这样，比如 *明天我去公园或者去逛街*，如果需要拆成两个子句，那么会是 *明天我去公园，或者去逛街*，而不是 *明天我去公园或者，去逛街*。对代码来说也是一样。而且把 `&&` 或 `||` 放在行首更容易对齐，看起来更舒服。

20.4 空格

- 在缩进和对齐之外的场景，不允许出现逻辑上不必要的连续多个空格。
- `+ && |` 等双元运算符左右要加一个空格。

- `!` `~` 等一元运算符和作用对象之间不加空格。
- `(` `)` 和 `((` `))` `{` `}` 内侧不加空格，`[[` `]]` 因为语法需要，内侧加一个空格。
- `;` 之前不加空格，之后加一个空格。
- 定义函数时（以及在 `((` `))` 中调用函数时），函数名和 `(` 之间不加空格。
- `if` `while` 等关键字和后边的内容之间加一个空格。
- `if` `[[` `]]` `{` 等场景中，`{` 和前边的内容之间加一个空格。
- 变量和 `[` `]` 之间不加空格，用 `[` `]` 取数组或者哈希表值时，`[` `]` 内侧不加空格。
- `>` `<` 等重定向符号和文件或者文件描述符之间不加空格。

原因：

1. 适量地添加空格可以让代码更清晰易读。
2. 这些约定基本属于很多编程语言代码风格中约定成俗的习惯，符合多数人的审美。

20.5 空行

- 非特殊场景，不允许出现超过两个连续空行。
- `#!/bin/zsh` 后加一个空行。
- `if` `while` 等语句块之后加一个空行。
- 定义函数后加一个空行。
- 逻辑关系不强的两行（或者两块）代码之间，根据逻辑关系强弱（自行判断），加一个或两个空行。

原因：

1. 适量添加空格，可以让代码逻辑按照空行分隔，提高可读性。
2. 因为添加空行的方法涉及诸多因素，很难详细约定，主要靠开发者自行判断。

20.6 括号

- 在判断条件的场景，不使用 `[` `]`，用 `[[` `]]` 代替。
- 在数值计算的场景，使用 `$((` `))` 而不是 `$[` `]`。

原因：

1. 在判断条件的场景，`[` `]` 的功能没有 `[[` `]]` 丰富，而且二者的用法存在差异，混合使用容易出问题。
2. 在数值比较或者计算的场景，`$[` `]` 的功能没有 `$((` `))` 丰富，混合使用容易出问题。
3. `[` `]` 在各种地方功能不一致，非必要场景尽量避免使用。

20.7 常量

- 字符串常量中如果没有特殊符号，两端可以不加引号，也可以加引号。
- 使用数值时，两端不加引号。

原因：

1. 如果任何字符串常量两端都加引号，容易让代码中充斥着引号，影响可读性。并且如果不小心误删引号，容易导致难以定位错误。
2. shell 脚本和很多其他编程语言不同，处理字符串的逻辑占很大部分，每个字符串常量两边都加引号的话，会增加很多额外工作量。

20.8 变量

- 用 `$var` 取变量值时，两边不加双引号，除非需要将非字符串变量转换成字符串。
- 在非必须场景，不需要加 `${var}` 中的大括号。
- 变量使用前要明确指明是局部变量（用 `local` 定义）还是全局变量（用 `typeset -g` 定义）。
- 能用局部变量的地方全部使用局部变量（用 `local` 定义）。
- 变量名中的单词可以使用下划线分隔或者驼峰风格，在不影响可读性的情况也可以使用全小写字母，但在同一个文件中要一致。

原因：

1. 和 bash 不同，zsh 在使用 `$var` 读取变量内容时，不用因为变量不存在、值为空、包含特殊符号而产生各种逻辑错误，所以无需在两端加双引号。
2. 用 `$var` 读变量是很多编程语言都有的用法，而 `${var}` 几乎是 shell 中特有的用法，并且输入更麻烦，没必要推广这种用法。而且因为不加大括号导致变量名粘连而出错的情况，编写代码时即可识别出来，和外部输入无关，不需要为了避免不存在的问题而输入很多额外的大括号。
3. 如果不指明变量是全局变量还是局部变量，默认是全局变量，有时候很难简单地判断一个变量是作为全局变量还是局部变量使用的，这样会给脚本的维护者带来很多麻烦。
4. 如果能使用局部变量的地方使用全局变量，更容易出现全局变量重名而互相影响导致错误的情况。这种错误是很难排查的（因为不会产生语法错误，容易让人怀疑是代码逻辑的问题，而不去检查是否有全局变量重名的情况），往往会浪费开发或者测试人员大量的时间。
5. 不同编程语言的开发者对变量名的风格偏好不同，不宜规定统一风格。

20.9 引号

- 字符串常量两端可以添加双引号或者单引号，但同一个文件中风格要一致。

原因：

1. 双引号和单引号的功能不同，混合使用是不可避免的。
2. 在双引号和单引号都适用的场景，统一使用一种引号，可以让代码更整洁易读。
3. 编程语言背景不同的开发者，对单双引号的偏好不同，不宜强行规定默认使用的引号。

20.10 函数

- 可以使用 `name()` 或者 `function name()` 定义函数，但同一个文件中风格要一致。

原因：

1. 如果约定统一使用 `name()` 定义函数，那么没有照顾 JavaScript 等编程语言开发者的习惯，而且 `function` 关键字有助于代码的搜索。
2. 如果约定统一使用 `function name()` 定义函数，需要额外输入 9 个字符，而意义有限，投入比产出要大。

20.11 脚本行数

- 非特殊场景，单个脚本文件不超过 1000 行。

原因：

1. 因为 shell 脚本的特性，单个脚本文件过长容易导致各种问题（比如全局变量互相影响）。1000 行代码对于多数场景都够用了。
2. 如果写的是安装脚本之类需要分发的脚本，那么分发单个文件要比分发多个文件（需要打包解包等额外工作）容易很多，这种场景可能需要写长脚本。所以不宜强行规定单个脚本文件最大行数。

20.12 语句风格

- 条件、循环、选择等语句，可以使用本系列教程中的风格，也可以使用 POSIX shell 风格，但同一个文件的风格要一致。

原因：

1. 本系列教程的中语句风格简洁易懂，并且和 c、Java、JavaScript 等语言的语句风格相近。
2. 从 bash 迁移过来的开发者习惯使用 POSIX shell 风格语句，需要兼顾。

本系列教程语句风格实例：

```
if [[ ... ]] {
} elif ((...)) {
} else {
}

case $i {
    (a)
    ...
    ;;

    (*)
    ...
    ;;
}
```

POSIX shell 语句风格实例：

```
if [[ ... ]]; then
elif ((...)); then
else
fi

case $i in
    (a)
    ...
    ;;

    (*)
    ...
    ;;
esac
```

20.13 总结

本文介绍了我建议的 zsh 代码风格，可以适当参考。

CHAPTER 21

21 测试方法以及编写可测试代码的方法

在正式的场景，代码写完后都是需要测试的，shell 脚本也不例外。但 shell 脚本的特性导致测试方法和其他语言有所不同。

21.1 单元测试

作为一种重要的测试方法，单元测试在很多种编程语言程序测试中起到举重轻重的作用。但不幸的是，单元测试基本不适用于 shell 脚本。并不是说 shell 脚本不能被单元测试，而是说单元测试能测试出来的问题很少，投入却很大。为了让 shell 脚本能被单元测试，50 行的代码很可能要改写成 100 多行甚至更多行。更重要的是 shell 脚本严重依赖外部环境，多数问题需要对脚本整体进行功能测试才能发现，而不是对单个函数进行单元测试。对单元测试的精力投入很可能会减少在功能测试的精力投入。

所以不建议推行 shell 脚本的单元测试，这不仅会让开发者很痛苦，也很难减少问题的出现几率，甚至有可能适得其反。

21.2 单个脚本的功能测试

Shell 脚本的最小测试粒度是单个脚本。必须保证单个脚本是容易测试的，不能多个脚本耦合太紧密而难以对其中某一个进行单独测试。

有主体逻辑的脚本依赖的外部环境必须是容易模拟的。比如需要从数据库中读取数据，对数据进行处理，然后写入到文件中，这些功能不能在同一个脚本中完成。因为数据库这个外部环境不容易模拟，会导致测试困难。需要把读写数据库的功能独立成单独的脚本，功能尽量简单，测试该脚本时只需要关心数据是否正常读取了出来，格式是否被正确转换等等，而不需要关心处理数据的具体逻辑。处理数据的主体逻辑代码要独立

成一个（或者多个）脚本，测试该脚本时，无需准备数据库环境，直接用另一个脚本或者数据文件取代读取数据库的脚本，提供测试数据。如果文件写入的环境复杂（比如文件或者目录结构复杂，或者要写入到分布式文件系统等等），也需要将文件写入的脚本独立出来以便更易于测试。

对有主体逻辑的脚本进行功能测试，不能手动进行，必须写测试脚本，可以自动运行。每次脚本改动后进行回归测试。项目稳定后，可以在每次提交代码后自动运行测试脚本。测试脚本必须覆盖正常和异常情况，不能只覆盖正常情况。异常情况的多少，要根据脚本的复杂度而定。

有复杂外部依赖的脚本，功能必须单一，逻辑尽量简单，代码尽量稳定，不经常改动。比如读写数据库、启停进程、复杂的目录文件操作等有复杂外部依赖的脚本，功能必须单一，只与一个特定的外部依赖交互，提供尽量和外部依赖无关的中间数据，尽量不包含和外部环境无关的逻辑。该类脚本要容易模拟，以便在测试其他部分时不再需要依赖外部环境。

对于有复杂外部依赖的脚本，可以写脚本自动测试，也可以手动测试，测试时需要包含正常和异常的情况，不能只测试正常情况。

21.3 功能测试示例

需要写脚本完成如下功能：

如果 process1 和 process2 两个进程都存在，以 process2 进程 cwd 目录中的 `data/output.txt` 为输入，做一些比较复杂的处理，然后输出到 process1 进程 cwd 目录中的 `data/input.txt` 文件（如果该文件已存在，则不处理），处理完后，删除之前的 `data/output.txt`。

分析：

process1 和 process2 两个进程都是复杂的外部依赖，不能在主体逻辑脚本里直接依赖它们，所以要把检查进程是否存在的逻辑独立成单独的脚本。输入和输出文件的路径依赖进程路径，为了测试方便，也要把获取文件路径的逻辑独立成单独的脚本。

脚本功能实现：

检查进程是否存在和获取进程 cwd 目录的 util.zsh 脚本：

```
#!/bin/zsh

check_process() {
    pidof $1
}

get_process_cwd() {
    readlink /proc/$1/cwd
}
```

主体逻辑脚本 main.zsh：

```
#!/bin/zsh

# 有错误即退出，可以省掉很多错误处理的代码
set -e

# 切换到脚本当前目录
cd ${0:h}

# 加载依赖的脚本
source ./util.zsh

# 检查进程是否存在
local process1_pid=$(check_process process1)
local process2_pid=$(check_process process2)

# 这里的 input 和 output 是相对脚本来说的
local input_file=$(get_process_cwd $process2_pid)/data/output.txt
local output_file=$(get_process_cwd $process1_pid)/data/input.txt

# 如果输入文件不存在，直接退出
[[ -e $input_file ]] || {
    echo $input_file not found.
    exit 1
}

# 如果输出文件已存在，也直接退出
[[ -e $output_file ]] && {
    echo $output_file already exists.
    exit 0
}

# 处理 $input_file 内容
# 省略

# 将结果输出到 $output_file
# 省略
```

功能测试方法：

util.zsh 里的两个函数功能过于简单，无需测试。

测试 main.zsh 时，需要构造一系列测试用的 util.zsh，用于模拟各种情况：

```
# 进程存在的情况
check_process() {
    echo $$
}

# 进程不存在的情况
check_process() {
    return 1
}

# 进程 process1 存在而 process2 不存在的情况
check_process() {
    [[ $1 == process1 ]] && echo 1234 && return
    [[ $1 == process2 ]] && return 1
}

# 输出了进程号，但实际进程不存在的情况
check_process() {
    echo 0
}

# 其他情况
# 省略


# 路径存在的情况
get_process_cwd() {
    [[ $1 == process1 ]] && echo /path/to/cwd1 && return
    [[ $1 == process2 ]] && echo /path/to/cwd2 && return
}

# 路径不存在的情况
get_process_cwd() {
    return 1
}

# 输出了路径，但路径实际不存在的情况
get_process_cwd() {
    echo /wrong/path
}
```

(下页继续)

(续上页)

```
# 其他情况
# 省略
```

然后组合这些情况，写测试脚本判断 main.zsh 的处理是否符合预期。

其中一个测试脚本样例：

util_test1.zsh 内容：

```
#!/bin/zsh

# 进程存在
check_process() {
    echo $$
}

# 直接返回正确的路径
get_process_cwd() {
    [[ $1 == process1 ]] && echo /path/to/cwd1 && return
    [[ $1 == process2 ]] && echo /path/to/cwd2 && return
}
```

test.zsh 内容：

```
#!/bin/zsh

# 用于测试的函数，可以独立成单独脚本以便复用
assert_ok() {
    (($1 == 0)) || {
        echo Error, retcode: $1
        exit 1
    }
}

check_output_file() {
    # 检查输出文件是否符合预期
    # 省略
}

# 应用 util_test1.zsh
ln -sf util_test1.zsh util.zsh
```

(下页继续)

(续上页)

```
# 运行脚本
./main.zsh

# 检查返回值是否正常
assert_ok $?

# 检查输出文件是否符合预期
check_output_file /path/to/output/file

# 其他检查
# 省略

# 应用 util_test2.zsh
ln -sf util_test2.zsh util.zsh

# 省略
```

21.4 集成测试

测试完每个脚本的功能后，需要将各个脚本以及其他程序整合起来测试互相调用过程是否正常。如果功能比较复杂，需要分批整合，测试各个逻辑单元是否能正常工作。在这部分测试中，和外部环境交互的脚本如果逻辑较为简单，可以不参与，用模拟脚本替代。可以手动测试或自动测试。同样不能只测试正常情况。

21.5 系统测试

将所有相关组件整合起来，测试整个系统或者子系统的功能。模拟脚本不能参与系统测试，必须使用真实的外部环境。系统测试通常需要手动进行，可以用自动化测试系统来辅助。需要覆盖尽可能多的情况，不能只测试系统的正常功能。

21.6 总结

本文简单介绍了 shell 脚本的测试方法，以及编写可测试代码的方法。

CHAPTER 22

22 bash 和 zsh 用法简明对照表

习惯写 bash 的开发者容易将 bash 下的用法用在 zsh 上，虽然多数情况并不会产生错误，但往往会多做很多不必要的工作，让脚本显得更臃肿或难以理解。

22.1 Bash 和 zsh 用法简明对照表

22.2 总结

本文简单列出了一些 zsh 中已经不再需要的 bash 用法，以及 zsh 和 bash 行为不一致的用法。待补充。