# R 语言函数式编程阅读笔记

Cheng Jun

https://github.com/chengjun90

2018 年 5 月 30 日

## 目录

# 1 准备工作

## 1.1 说明

Mailund（2017）的 Functional Programming in R: Advanced Statistical Programming for Data Science,Analysis and Finance 是一本关于函数式编程的书籍。不过这个副标题有点长，不仅关乎数据科学，关于数据分析还关乎金融分析。其实这本书主要在讲 R 语言中的函数式编程的应用。

S 语言的设计者、R 语言核心团队成员 Chambers（2016）在 Extending R 一书中提到 R 语言中的三大理念:

> Everything that exists in R is an object. OBJECT
> Everything that happens in R is a function call.FUNCTION
> Interfaces to other software are part of R. INTERFACE

所以学一点函数式编程对于提高 R 语言应用能力是很有益处。

## 1.2 写作环境

版本信息:

```
devtools::session_info()
```

```
## Session info -------------------------------------------------------------

##  setting  value
##  version  R version 3.5.0 (2018-04-23)
##  system   x86_64, mingw32
##  ui       RTerm
##  language (EN)
##  collate  Chinese (Simplified)_China.936
##  tz       Asia/Taipei
##  date     2018-05-30

## Packages -----------------------------------------------------------------

##  package    * version date       source
##  backports    1.1.2   2017-12-13 CRAN (R 3.5.0)
##  base       * 3.5.0   2018-04-23 local
##  compiler     3.5.0   2018-04-23 local
##  datasets   * 3.5.0   2018-04-23 local
```

```
##  devtools    1.13.5  2018-02-18 CRAN (R 3.5.0)
##  digest      0.6.15  2018-01-28 CRAN (R 3.5.0)
##  evaluate    0.10.1  2017-06-24 CRAN (R 3.5.0)
##  graphics  * 3.5.0   2018-04-23 local
##  grDevices * 3.5.0   2018-04-23 local
##  htmltools   0.3.6   2017-04-28 CRAN (R 3.5.0)
##  knitr       1.20    2018-02-20 CRAN (R 3.5.0)
##  magrittr    1.5     2014-11-22 CRAN (R 3.5.0)
##  memoise     1.1.0   2017-04-21 CRAN (R 3.5.0)
##  methods   * 3.5.0   2018-04-23 local
##  Rcpp        0.12.17 2018-05-18 CRAN (R 3.5.0)
##  rmarkdown   1.9     2018-03-01 CRAN (R 3.5.0)
##  rprojroot   1.3-2   2018-01-03 CRAN (R 3.5.0)
##  rticles     0.4.1   2017-05-16 CRAN (R 3.5.0)
##  stats     * 3.5.0   2018-04-23 local
##  stringi     1.2.2   2018-05-02 CRAN (R 3.5.0)
##  stringr     1.3.1   2018-05-10 CRAN (R 3.5.0)
##  tools       3.5.0   2018-04-23 local
##  utils     * 3.5.0   2018-04-23 local
##  withr       2.1.2   2018-03-15 CRAN (R 3.5.0)
##  yaml        2.1.19  2018-05-01 CRAN (R 3.5.0)
```

### 1.3 许可协议说明



# 2 Functions in R

这章节主要的内容是如何写一个函数。

### 2.1 手写一个函数

来，写一个自定义描述性统计 su 函数。

```
su <- function(var) {
    obs = length(var)
```

```
    mean = mean(var)
    median = median(var)
    sd = sd(var)
    return(list(
    "obs" = obs,
    "mean" = mean,
    "median" = median,
    "sd" = sd
    ))
}
```

function(var){} 就是一个函数（可以看成一个匿名函数），su 是函数名字，var 是函数参数。调用一下 su() 函数。

```
su(mtcars$mpg)
```

```
## $obs
## [1] 32
##
## $mean
## [1] 20.09062
##
## $median
## [1] 19.2
##
## $sd
## [1] 6.026948
```

或者换一个输出方式。

```
unlist(su(mtcars$mpg))
```

```
##       obs      mean    median        sd
## 32.000000 20.090625 19.200000  6.026948
```

## 2.2 默认参数

```
myfun <- function(x = 1, y = 2) {
    z = x + y
    return(z)
}
```

如果不指定 x，y 参数值。myfun() 直接应用默认的 1+2。

```
myfun()
```

```
## [1] 3
```

```
myfun(10,20)
```

```
## [1] 30
```

## 2.3 意犹未尽的...

来看一下 su 函数，如果 var 含有缺失值呢，su 函数就不好处理了。

```
su(c(mtcars$mpg,NA))
```

```
## $obs
## [1] 33
##
## $mean
## [1] NA
##
## $median
## [1] NA
##
## $sd
## [1] NA
```

此时，可以用...来重新写 su 函数。更多可以参考 $help("...")$。

```r
su_dot <- function(var,...) {
    obs = length(var)
    mean = mean(var,...)
    median = median(var,...)
    sd = sd(var,...)
    return(list(
        "obs" = obs,
        "mean" = mean,
        "median" = median,
        "sd" = sd
    ))
}
```

```r
su_dot(c(mtcars$mpg,NA), na.rm = TRUE)
```

```
## $obs
## [1] 33
##
## $mean
## [1] 20.09062
##
## $median
## [1] 19.2
##
## $sd
## [1] 6.026948
```

# 3 Pure Functional Programming

纯函数可以简单理解为一个函数是相同的输入，相同的输出。

在小型和中型数据分析项目，主要是以交互式分析为主，很快就能判断代码的对错。此时，可以不用太在意纯函数的问题。

# 4 Scopes and Environments

```
gx <- 1:100
f <- function(px) sqrt(sum(px))
f(gx**2)
```

```
## [1] 581.6786
```

# 5 Scope and Closures

这里介绍到闭包。

闭包可以简单理解为一个函数嵌套了一个函数，且内部函数用到外部函数的变量。

count_value 是来计算某个变量大于特定值的个数。function(var) 就用到 function(value) 中的 value。value 在 count_value 函数用来指定门槛值。

```
count_value <- function(value) {
    function(var) {
        x = sum(var > value)
        return(x)
    }
}
```

来看一下 count_3

```
count_3 <- count_value(3)
count_3(mtcars$drat)
```

```
## [1] 28
```

```
count_3(mtcars$wt)
```

```
## [1] 20
```

来看一下 count_3

```
count_4 <- count_value(4)
count_4(mtcars$drat)
```

```
## [1] 7
```

```
count_4(mtcars$wt)
```

```
## [1] 4
```

# 6 Higher-Order Functions

Higher-order function 翻译成中文就是高阶函数。

高阶函数就是把一个函数当做另外一个函数的参数。

例如 lapply 系列就是高阶函数。$lapply(mtcars, mean)$ 计算 mtcars 各个列的均值，$mean()$ 函数成了 $lapply()$ 的参数。

```
lapply(mtcars,mean)
```

```
## $mpg
## [1] 20.09062
##
## $cyl
## [1] 6.1875
##
## $disp
## [1] 230.7219
##
## $hp
## [1] 146.6875
##
## $drat
## [1] 3.596563
##
## $wt
## [1] 3.21725
##
```

```
## $qsec
## [1] 17.84875
##
## $vs
## [1] 0.4375
##
## $am
## [1] 0.40625
##
## $gear
## [1] 3.6875
##
## $carb
## [1] 2.8125
```

下来，再写一个简单的高阶函数。

```
myfun <- function(x, y, z) {
    tmp <- z(x,y)
    return(tmp)
}

a <- mtcars$mpg
b <- mtcars$disp

myfun(a,b,cov)
```

```
## [1] -633.0972
```

```
myfun(a,b,cor)
```

```
## [1] -0.8475514
```

在 $function(x, y, z)$ 中 $z$ 用来传递函数。

# 7 Filter, Map, and Reduce

Filter, Map, and Reduce 是函数式编程常见的三个函数。

下面的例子中是用 Filter 筛选大于 20 的样本观测值:

```
Filter(function(x) x > 20, mtcars$mpg)
```

```
##  [1] 21.0 21.0 22.8 21.4 24.4 22.8 32.4 30.4 33.9 21.5 27.3 26.0 30.4 21.4
```

下面是对 mtcars 各个列求均值。

```
Map(mean,mtcars)
```

```
## $mpg
## [1] 20.09062
##
## $cyl
## [1] 6.1875
##
## $disp
## [1] 230.7219
##
## $hp
## [1] 146.6875
##
## $drat
## [1] 3.596563
##
## $wt
## [1] 3.21725
##
## $qsec
## [1] 17.84875
##
## $vs
## [1] 0.4375
##
## $am
## [1] 0.40625
##
```

```
## $gear
## [1] 3.6875
##
## $carb
## [1] 2.8125
```

```
Reduce(`+`, c(1,2,3,4))
```

```
## [1] 10
```

```
Reduce(`*`, c(1,2,3,4))
```

```
## [1] 24
```

上面这三个函数用到的少，基本了解即可。更多的可以参看第三方包purrr 。purrr提供了类似函数。

# 8 Point-Free Programming

Point-Free Programming 可以看出无形参函数。书的例子理解起来还比较麻烦。

```
compose <- function(g, f) function(...) g(f(...))
```

用 compose 来实现函数的组合。下面我自己写两个小函数，然后组合起来看看效果。

```
fun1 <- function(x) {
    return(x * 10)
}
fun2 <- function(x) {
    return(x + 0.001)
}

new_fun <- compose(fun1, fun2)
new_fun(1)
```

```
## [1] 10.01
```

$new_fun(1)$ 计算结果为 10.01。其实就是 $(1 + 0.001) * 10$。书中的 compose 函数简洁，但是不直观。下面改写为 compose_good。

```
compose_good <- function(g, f){
    function(...){
        g(f(...))
    }
}

new_fun_good <- compose_good(fun1,fun2)
new_fun_good(1)
```

```
## [1] 10.01
```

计算结果一致。

Point-Free Programming 在 R 语言的应用之一就是管道操作，数据从一个函数流淌至下一个，行云流水。

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
mtcars %>% select(vs, mpg, disp) %>%
    group_by(vs) %>%
    summarise(mean_mpg = mean(mpg),
              mean_disp = mean(disp))
```

```
## # A tibble: 2 x 3
##      vs mean_mpg mean_disp
##   <dbl>    <dbl>     <dbl>
## 1     0     16.6      307.
## 2     1     24.6      132.
```

# 9 结语

在 R 语言自带的函数之外，还有第三方包提供了函数式编程的支持。purrr提高了很多函数式编程的小函数。早在 2017 年的时候就在国内重点介绍过purrr的应用。

- R 语言函数式编程 purrr 的应用
- 模拟、嵌套与回归：函数式编程 purrr

值得进一步学习和掌握purrr。