R 函数编程概念

周世祥

2021/5/27

R 函数式编程

R 语言的核心式一门函数式编程语言，为我们提供大量的创建和操作函数的工具。

R 有所谓的一级函数，适用于向量的所有操作也都适用于函数，可以将函数赋值给变量，将函数存储在列表中，将函数作为参数传递给其他函数，在函数内再创建一个函数，甚至可以把函数作为一个函数的结果返回。

例如：

```
set.seed(1014)
df <- data.frame(replicate(6,sample(c(1:10,-99),6,rep=TRUE)))

names(df)<-letters[1:6]

df
```

```
##   a   b   c   d  e f
## 1 7   5 -99   2  5 2
## 2 5   5   5   3  6 1
## 3 6   8   5   9  9 4
## 4 4   2   2   6  6 8
## 5 6   7   6 -99 10 6
## 6 9 -99   4   7  5 1
```

```
df$a[df$a == -99]<- NA
df$b[df$b == -99]<- NA
df$c[df$c == -98]<- NA
df$d[df$d == -99]<- NA
df$e[df$e == -99]<- NA
df$f[df$g == -99]<- NA
```

```
# 使用复制黏贴很容易出错，若缺失值从-99 修改为 9999，就必须在多个地方修改。
df
```

```
##   a  b   c  d  e f
## 1 7  5 -99  2  5 2
## 2 5  5   5  3  6 1
## 3 6  8   5  9  9 4
## 4 4  2   2  6  6 8
## 5 6  7   6 NA 10 6
## 6 9 NA   4  7  5 1
```

为避免这类错误，并使代码更灵活，采用 “不要自我重复”(do not repeat yourself)，即 DRY 原则，《Pragmatic Programmers》Dave Thomas 作。

```
fix_missing <- function(x){
  x[x == -99] <- NA
  x
}

df$a <- fix_missing(df$a)
df$b <- fix_missing(df$b)
df$c <- fix_missing(df$c)
df$d <- fix_missing(df$d)
df$e <- fix_missing(df$e)
df$f <- fix_missing(df$f)
df
```

```
##   a  b  c  d  e f
## 1 7  5 NA  2  5 2
## 2 5  5  5  3  6 1
## 3 6  8  5  9  9 4
## 4 4  2  2  6  6 8
## 5 6  7  6 NA 10 6
## 6 9 NA  4  7  5 1
```

上述方法减少了出错的范围，但不能完全消除，仍有可能将变量名弄错，下一步就是将两个函数结合到一起来避免这种错误的发生。

lapply 函数可以将这种操作应用到数据框的每一列。lapply 函数接收三个输入：x 一个列表，f 一个函数，… 传给 f() 的其他参数。

lapply 是一个泛函，泛函是函数式编程中一个非常重要的部分。

```
rm(list=ls())
set.seed(1014)
df <- data.frame(replicate(6,sample(c(1:10,-99),6,rep=TRUE)))

names(df)<-letters[1:6]
fix_missing <- function(x){
  x[x==-99] <- NA
  x
}

df[1:5] <- lapply(df[1:5],fix_missing)
df
```

```
##   a  b  c  d  e f
## 1 7  5 NA  2  5 2
## 2 5  5  5  3  6 1
## 3 6  8  5  9  9 4
## 4 4  2  2  6  6 8
## 5 6  7  6 NA 10 6
## 6 9 NA  4  7  5 1
```

```
#df[1:5] <- lapply(df[1:5],fix_missing)
```

```
df[]<- lapply(df,fix_missing)
df
```

```
##   a  b  c  d  e f
## 1 7  5 NA  2  5 2
## 2 5  5  5  3  6 1
## 3 6  8  5  9  9 4
## 4 4  2  2  6  6 8
## 5 6  7  6 NA 10 6
## 6 9 NA  4  7  5 1
```

无论有多少列都可以使用它，不会丢掉任何一列，所有列操作都是相同的，关键思想是函数组合，将两个简单函数组合起来，一个函数修复缺失值一个对每一列做同样的操作。

如果每一列使用不同的值来替换缺失值又该怎么办？

```
fix_missing_99 <- function(x){
  x[x==-99]<-NA
  x
}
```

```
fix_missing_999 <- function(x){
  x[x==-999]<-NA
  x
}
```

如前面一样，它容易出错，可以使用闭包，它是创建并返回函数的函数。闭包允许我们基于模板来创建函数。

```
missing_fixer <- function(na_value){
  function(x){
    x[x == na_value] <- NA
    x
  }
}
fix_missing_99 <- missing_fixer(-99)
fix_missing_999 <- missing_fixer(-999)
#fix_missing_99 <- missing_fixer(c(-99,-999))
```

泛函

泛函在数学中非常常见，极限，最大值，求根，以及定积分都是泛函：给定一个函数，它们返回一个向量，实现它们的算法中都包含迭代。

R 中内置的数学泛函：

```
integrate(sin,0,pi) # 计算曲线面积
```

```
## 2 with absolute error < 2.2e-14
```

```
# uniroot() # 求根

uniroot(sin,pi*c(1/2,3/2))
```

```
## $root
## [1] 3.141593
##
## $f.root
## [1] 1.224606e-16
##
## $iter
## [1] 2
```

```
##
## $init.it
## [1] NA
##
## $estim.prec
## [1] 6.103516e-05
```

```
optimise(sin,c(0,2*pi))# 最小值
```

```
## $minimum
## [1] 4.712391
##
## $objective
## [1] -1
```

高阶函数就是以函数作为输入，并以函数作为输出的函数，前面学习了一种高阶函数：闭包，由另一个函数返回的函数。

```
# 泛函例子
randomise <- function(f){
  f(runif(1e3))
}

randomise(mean)
```

```
## [1] 0.5088265
```

```
randomise(mean)
```

```
## [1] 0.4954888
```

```
randomise(sum)
```

```
## [1] 492.6025
```

三个常用的泛函为：lapply(),apply(),tapply() 都可以接收一个函数作为输入，并返回一个向量作为输出。

泛函的一个常用功能就是替代循环，循环的最大缺点就是表达不够清晰，for 是对某事进行迭代，但不能清晰地表达更高层次的目的。

lapply() 接收一个函数，并将这个函数应用到列表中的每一个元素，最后再将结果以列表的形式返回。

lapply 是一个常见的 for 循环模式的包装器，为输出创建一个容器，将 f() 应用到列表中的每一个元素，将结果填充到容器中，其他 for 循环泛函都是这一思路的变形。

R 语言的设计限制了它的最大理论性能，慢的方面，不是因为它们的定义，而是因为它的实现。

R 核心有 80 万行代码，大约 45% 是 c 语言代码，19% 为 R 代码，17% 为 fortran 代码，只有 R 的核心成员，20 个人，活跃在前台的大约有 6 位，没有一位是全职工作在 R 上，大多数人是统计学家，所以在接受新代码上倾向于比较保守。

R 版的函数是完全向量化的，所以它已经很快了，对一个包含 100 万个元素的向量 y 进行计算，需要 8 毫秒，c++ 函数比它快两倍，4 毫秒，但假设编写一个 c++ 函数花费 10 分钟，这样是看得不偿失的。

参考文献

1 、《高级 R 语言编程指南》，机械工业出版社，哈德利，维克汉姆

2、https://bookdown.org/yihui/bookdown/

3、https://github.com/shixiangbupt/adv-r