第一次专题讨论资料（apply 和 dplyr）

邓光宏

目录

1 高效数据处理工具（一）：*apply() & plyr

1.1 引言

在 R 语言中你可能最常听到的告诫就是“避免写 for 语句”。的确，当你写 Matlab 之类的程序时，你可能很习惯于下面这样的计算方式：

- 建立一个向量
- 用 for 语句索引向量的每一个元素
- 依据索引分步计算

但是在 R 语言中我们几乎不会这样去做。一个原因是 R 语言对于循环语句（loop）的处理效率很低。许多刚接触 R 的用户经常抱怨，为什么我的程序运行如此之慢，但实际上许多时候并不是 R 语言本身效率的问题，而是代码处理方式的问题。另一个原因是，过多的循环语句将使得我们很难追踪计算过程。尤其在多个 for 语句嵌套的时候，我们很难直观地了解每一层循环的目的，最终使得程序的修改和维护变得非常困难。

那么 R 语言中我们应该如何处理相关的问题？这里我们简要介绍相关的 *apply() 函数以及 plyr 这个包提供给我们的工具。

1.2 apply()

几个例子

假如我们要做下面这个计算：生成 1000(0,1) 均匀分布随机数，计算每个随机数的平方根，并且把这 1000 个平方根存入到 res 向量中。

如果写 for 语句，我们可能会这样处理：

```
xs <- runif(1e3)
res <- c()
for (x in xs) {
  res <- c(res, sqrt(x))
}
```

```
xs <- runif(1e3)
res <- numeric(length(xs))
for (i in seq_along(xs)) {
  res[i] <- sqrt(xs[i])
}
```

（这个例子来自Advanced R, Functionals, My first functional: lapply()。

然而这两种方式对 R 语言来说都很糟糕，尤其是第一段程序。实际上我们要做的事情很清晰明了，但 for 语句不仅冗长，而且模糊了我们这段程序的目的。如果你之前习惯于 Matlab 的话，你也很可能会按照第一段程序的方式去进行运算。这种计算我们一般称为 Growing Vector，通过不断的向向量尾增加元素的方式来生成一个有完整结果的向量。但在 R 语言里，每次增长一个元素，计算机都会完全复制一次已存在的元素，再进行元素添加，这使得计算非常缺乏效率。

lapply()

实际上在 R 语言中我们对此的处理方式非常简明。R 语言中有一组和 apply 相关的函数，帮助我们处理这类计算。最基础的函数是 `lapply()`。它有三个参数 `lapply(X, FUN, ...)`。第一个参数 `X` 是要进行循环的变量，`FUN` 是每次循环要作用的函数，`...` 是 `FUN` 函数中可以添加的其他参数。它返回的结果是一个由各次运算结果拼接成的 `list`。

比如同样是前述的计算，我们可以用下面的方法来做：

```
xs <- runif(1e3)
res <- unlist(lapply(xs, sqrt))
```

我们再举一个 X 为 list 的例子来理解 lapply() 的机制：

```
l <- replicate(20, runif(sample(1:10, 1)), simplify = FALSE)

# With a for loop
out <- vector("list", length(l))
for (i in seq_along(l)) {
  out[[i]] <- length(l[[i]])
}
unlist(out)

# With lapply
unlist(lapply(l, length))

# another task: ( decreasingly) sort each vector of l
lapply(l, sort, decreasing = TRUE)
```

sapply()

有时我们每次作用 FUN 函数时得到的结果是等长的一些向量，我们其实更希望它能自动整理成一个恰当的表格。sapply() 可以直接帮助我们的完成这个简化的过程。

```
xs <- runif(1e3)
res <- sapply(xs, sqrt, simplify = TRUE)
```

mapply()

"mapply is a multivariate version of sapply. mapply applies FUN to the first elements of each (???) argument, the second elements, the third elements, and so on.

多参数版本的 sapply()。第一次计算传入各组向量的第一个元素到 FUN，进行结算得到结果；第二次传入各组向量的第二个元素，得到结果；第三次传入各组向量的第三个元素…以此类推。

```
l1 <- list(a = c(1:10), b = c(11:20))
l2 <- list(c = c(21:30), d = c(31:40))
mapply(sum, l1$a, l1$b, l2$c, l2$d)
```

```
##  [1]  64  68  72  76  80  84  88  92  96 100
```

```
# equal to
res <- c()
for(i in 1:10) {
  res[i] <- sum(l1$a[i], l1$b[i], l2$c[i], l2$d[i])
}
```

apply()

适用于多维的 array，它可以先对指定的维度做切片，再应用函数做运算。

```
# create a matrix of 10 rows x 2 columns
m <- matrix(1:20, nrow = 4, ncol = 5)
# mean of the rows
apply(m, 1, mean)
```

```
## [1]  9 10 11 12
```

```
# mean of the columns
apply(m, 2, mean)
```

```
## [1]  2.5  6.5 10.5 14.5 18.5
```

其中 MARGIN 指定要进行切片的维度。

tapply()

通过分组进行切片，再作用函数。通过 tapply() 我们可以方便地完成一些分组统计工作。

```
df <- data.frame(Rand = rnorm(1e3), Group = sample(LETTERS[1:6], replace = TRUE, siz
tapply(df$Rand, INDEX = df$Group, FUN = sd)
```

```
##         A         B         C         D         E         F
## 0.9697036 1.0861310 0.9481479 1.0601347 1.0583761 0.9549656
```

匿名函数（Anonymous Functions）

上面我们的函数都比较简单，但是如果这些函数不能实现我们的复杂计算该怎么办呢？实际上我们可以自写函数来完成运算，比如下面的例子：

```
l <- replicate(20, runif(sample(1:10, 1)), simplify = FALSE)

sapply(l, function(vec) c(Mean = mean(vec, trim = 0.05), Sd = sd(vec)))

# equal to
summ_fun <- function(vec) {
  return(c(Mean = mean(vec, trim = 0.05), Sd = sd(vec)))
}

sapply(l, summ_fun)
```

1.3 plyr

Split-Apply-Combine Strategy

Hadley Wickham 2011 年的文章 The Split-Apply-Combine Strategy for Data Analysis很精练地提出了数据分析中非常常用的一个策略（非常建议读一下这篇文章）。Split-apply-combine 的模式几乎存在于数据分析的每一个角落，它主要描述了数据处理的三个过程：

- Split：对数据切片。
- Apply：对每份切片的数据作用函数，并得到每个切片的结果。
- Combine：将每个切片的结果汇总整理成适当的形式。

plyr

实际上 `*apply()` 这些函数都是 split-apply-combine strategy 的一些应用：`lapply()` 提供了一个最 general 的切片的范式，但是为了保持结果的一般性，它并没有做 combine，而 `sapply()` 实现了 combine 的过程。Hadley Wickham 写了 plyr 这个包，提供了一组规范的数据结构转换形式。

Input/Output	list	data frame	array
list	llply()	ldply()	laply()
data frame	dlply()	ddply()	daply()
array	alply()	adply()	aaply()

一个简单的例子

iris 是内置在 R 语言中的一份数据（这个数据集是很早用来表示 Fisher 聚类的一组数据，你可以在 Wiki 上找到它。

我们先做一个简单的分析：每种花的 Sepal 和 Petal 的 length 和 width 的均值分别是多少？

```
## Split
iris.set <- iris[iris$Species == "setosa", -5]
iris.versi <- iris[iris$Species == "versicolor", -5]
iris.virg <- iris[iris$Species == "virginica", -5]

## Apply
mean.set <- colMeans(iris.set)
mean.versi <- colMeans(iris.versi)
mean.virg <- colMeans(iris.virg)

## Combine
mean.iris <- rbind(mean.set, mean.versi, mean.virg)
rownames(mean.iris) <- c("setosa", "versicolor", "virginica")

mean.iris
```

上面提供了一个 `Split-Apply-Combine Strategy` 的详细过程，但显然，它很不精练，而在 plyr 中实现这类分析非常简单：

```
library(plyr)
ddply(iris, .variables = "Species", .fun = function(df_sub){
  colMeans(df_sub[, -5])
})
```

我们也可以很方便地做一些更复杂的事情，比如我们想做一组 Petal.Length 对 Petal.Width 的回归，并把每组回归的结果都整理到同一张表格中。

```
ddply(iris, .variables = "Species", .fun = function(df_sub) {
  model <- lm(Petal.Width ~ Petal.Length, data = df_sub)
  return(c(model$coefficients, R2 = summary(model)$r.squared))
})
```

我们也可以保留每组回归的全部信息，并把各组回归结果存放在一个 list 里：

```
group_lm <- dlply(iris, .variables = "Species", .fun = function(df_sub) {
  lm(Petal.Width ~ Petal.Length, data = df_sub)
})

str(group_lm)
```

plyr 使得 R 语言中数据结构的转换模式非常灵活，我们在此也不可能穷尽所有的情形，但是希望本文对 *apply() 和 plyr 包的简单引入和例子能够对你更高效地完成数据分析和整理提供帮助。

不适用的情况

但是并非所有情形都能够依赖 *apply() 和 Split-Apply-Combine Strategy 完成，有一些情形必须使用循环。比如下面这个生成 Fibonacci 数列的例子：

```
n <- 50
fib <- c(1, 1)
```

```
for( i in 3:50 ){
  fib[i] <- fib[i-1] + fib[i-2]
}
```

*apply() 和 Split-Apply-Combine Strategy 假设每步计算是独立的，也就是这次计算的参数输入不依赖于其他计算的结果。当不能满足这个条件时，我们只能用循环迭代的方式完成计算。一些复杂的算法往往需要重复迭代，这种情形我们可能更推荐用 Rcpp 等工具来完成。

1.3.1 Further Reading

- 关于 apply() 函数族的一个简明介绍可以参考这篇文章：A brief introduction to "apply??? in R。本文的部分例子来源于此。
- Advanced R, For loop functionals: friends of lapply()中清晰地描述了这些函数的一般机制。本文的部分例子来源于此。
- 去年 WISE RClub 的材料中也介绍了相关内容，并提供了简单的例子：R-Training 2014, Chapter 4, apply() family
- 关于 plyr 包可以详细参考 Handley Wickham 在 Journal of Statistical Software 上的文章The Split-Apply-Combine Strategy for Data Analysis。