# R<给\*apply函数族中的FUN传递多个参数>

- 博客频道 - CSDN.NET

http://blog.csdn.net/gpwner/article/details/70207661

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

lapply(1:3,myfxn,var2=2,var3=100)

[[1]]

[1] 200

[[2]]

[1] 400

[[3]]

[1] 600

> sapply(1:3,myfxn,var2=2,var3=100)

[1] 200 400 600

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

版权声明：本文为博主原创文章，未经博主允许不得转载。

# R语言 apply函数家族详解

\_数据科学家快报\_新浪博客

http://blog.sina.com.cn/s/blog\_403aa80a010174dj.html

|  |  |  |
| --- | --- | --- |
| apply | Apply Functions Over Array Margins  对阵列行或者列使用函数 | apply(X, MARGIN, FUN, ...) |
| lapply | Apply a Function over a List or Vector  对列表或者向量使用函数 | lapply(X, FUN, ...) |
| sapply | Apply a Function over a List or Vector  对列表或者向量使用函数 | sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE) |
| vapply | Apply a Function over a List or Vector  对列表或者向量使用函数 | vapply(X, FUN, FUN.VALUE, ..., USE.NAMES = TRUE) |
| tapply | Apply a Function Over a Ragged Array  对不规则阵列使用函数 | tapply(X, INDEX, FUN = NULL, ..., simplify = TRUE) |
| eapply | Apply a Function Over Values in an Environment  对环境中的值使用函数 | eapply(env, FUN, ..., all.names = FALSE, USE.NAMES = TRUE) |
| mapply | Apply a Function to Multiple List or Vector Arguments  对多个列表或者向量参数使用函数 | mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE) |
| rapply | Recursively Apply a Function to a List  运用函数递归产生列表 | rapply(object, f, classes = "ANY", deflt = NULL,how = c("unlist", "replace", "list"), ...) |

apply {base}

通过对数组或者矩阵的一个维度使用函数生成值得列表或者数组、向量。

apply(X, MARGIN, FUN, ...)

X 阵列，包括矩阵

MARGIN  1表示矩阵行，2表示矩阵列，也可以是c(1,2)

例：

>xxx<-matrix(1:20,ncol=4)

>apply(xxx,1,mean)

[1]  8.5  9.5 10.5 11.5 12.5

>apply(xxx,2,mean)

[1]  3  8 13 18

>xxx

     [,1] [,2] [,3] [,4]

[1,]    1    6   11   16

[2,]    2    7   12   17

[3,]    3    8   13   18

[4,]    4    9   14   19

[5,]    5   10   15   20

lapply {base}

通过对x的每一个元素运用函数，生成一个与元素个数相同的值列表

lapply(X, FUN, ...)

X表示一个向量或者表达式对象，其余对象将被通过as.list强制转换为list

例：

> x <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE,FALSE,FALSE,TRUE))

> x

$a

 [1]  1  2  3  4  5  6  7  8  9 10

$beta

[1]  0.04978707  0.13533528  0.36787944  1.00000000  2.71828183  7.38905610

[7] 20.08553692

$logic

[1]  TRUE FALSE FALSE  TRUE

> lapply(x,mean)

$a

[1] 5.5

$beta

[1] 4.535125

$logic

[1] 0.5

sapply {base}

这是一个用户友好版本，是lapply函数的包装版。该函数返回值为向量、矩阵，如果simplify=”array”，且合适的情况下，将会通过simplify2array()函数转换为阵列。sapply(x, f, simplify=FALSE, USE.NAMES=FALSE)返回的值与lapply(x,f)是一致的。

sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)

X表示一个向量或者表达式对象，其余对象将被通过as.list强制转换为list

simplify 逻辑值或者字符串，如果可以，结果应该被简化为向量、矩阵或者高维数组。必须是命名的，不能是简写。默认值是TRUE，若合适将会返回一个向量或者矩阵。如果simplify=”array”，结果将返回一个阵列。

USE.NAMES  逻辑值，如果为TRUE，且x没有被命名，则对x进行命名。

例：

> sapply(k, paste,USE.NAMES=FALSE,1:5,sep="...")

     [,1]    [,2]    [,3]

[1,] "a...1" "b...1" "c...1"

[2,] "a...2" "b...2" "c...2"

[3,] "a...3" "b...3" "c...3"

[4,] "a...4" "b...4" "c...4"

[5,] "a...5" "b...5" "c...5"

> sapply(k, paste,USE.NAMES=TRUE,1:5,sep="...")

     a       b       c

[1,] "a...1" "b...1" "c...1"

[2,] "a...2" "b...2" "c...2"

[3,] "a...3" "b...3" "c...3"

[4,] "a...4" "b...4" "c...4"

[5,] "a...5" "b...5" "c...5"

> sapply(k, paste,USE.NAMES=TRUE,1:5,sep="...",simplyfy=TRUE)

     a              b              c

[1,] "a...1...TRUE" "b...1...TRUE" "c...1...TRUE"

[2,] "a...2...TRUE" "b...2...TRUE" "c...2...TRUE"

[3,] "a...3...TRUE" "b...3...TRUE" "c...3...TRUE"

[4,] "a...4...TRUE" "b...4...TRUE" "c...4...TRUE"

[5,] "a...5...TRUE" "b...5...TRUE" "c...5...TRUE"

> sapply(k, paste,simplify=TRUE,USE.NAMES=TRUE,1:5,sep="...")

     a       b       c

[1,] "a...1" "b...1" "c...1"

[2,] "a...2" "b...2" "c...2"

[3,] "a...3" "b...3" "c...3"

[4,] "a...4" "b...4" "c...4"

[5,] "a...5" "b...5" "c...5"

> sapply(k, paste,simplify=FALSE,USE.NAMES=TRUE,1:5,sep="...")

$a

[1] "a...1" "a...2" "a...3" "a...4" "a...5"

$b

[1] "b...1" "b...2" "b...3" "b...4" "b...5"

$c

[1] "c...1" "c...2" "c...3" "c...4" "c...5"

vapply {base}

vapply类似于sapply函数，但是它的返回值有预定义类型，所以它使用起来会更加安全，有的时候会更快

在vapply函数中总是会进行简化，vapply会检测FUN的所有值是否与FUN.VALUE兼容，以使他们具有相同的长度和类型。类型顺序：逻辑<</span>整型<</span>实数<</span>复数

vapply(X, FUN, FUN.VALUE, ..., USE.NAMES = TRUE)

X表示一个向量或者表达式对象，其余对象将被通过as.list强制转换为list

simplify 逻辑值或者字符串，如果可以，结果应该被简化为向量、矩阵或者高维数组。必须是命名的，不能是简写。默认值是TRUE，若合适将会返回一个向量或者矩阵。如果simplify=”array”，结果将返回一个阵列。

USE.NAMES  逻辑值，如果为TRUE，且x没有被命名，则对x进行命名。

FUN.VALUE   一个通用型向量，FUN函数返回值得模板

例：

> x<-data.frame(a=rnorm(4,4,4),b=rnorm(4,5,3),c=rnorm(4,5,3))

> vapply(x,mean,c(c=0))

         a          b          c

 1.8329043  6.0442858 -0.1437202

> k<-function(x)

+ {

+ list(mean(x),sd(x))

+ }

> vapply(x,k,c(c=0))

错误于vapply(x, k, c(c = 0)) : 值的长度必需为1，

 但FUN(X[[1]])结果的长度却是2

> vapply(x,k,c(c=0,b=0))

错误于vapply(x, k, c(c = 0, b = 0)) : 值的种类必需是'double'，

 但FUN(X[[1]])结果的种类却是'list'

> vapply(x,k,c(list(c=0,b=0)))

  a        b        c

c 1.832904 6.044286 -0.1437202

b 1.257834 1.940433 3.649194

tapply {base}

对不规则阵列使用向量，即对一组非空值按照一组确定因子进行相应计算

tapply(X, INDEX, FUN, ..., simplify = TRUE)

x  一个原子向量，典型的是一个向量

INDEX  因子列表，和x长度一样，元素将被通过as.factor强制转换为因子

simplify  若为FALSE，tapply将以列表形式返回阵列。若为TRUE，FUN则直接返回数值

例：

> height <- c(174, 165, 180, 171, 160)

> sex<-c("F","F","M","F","M")

> tapply(height, sex, mean)

  F     M

170   170

eapply {base}

eapply函数通过对environment中命名值进行FUN计算后返回一个列表值，用户可以请求所有使用过的命名对象。

eapply(env, FUN, ..., all.names = FALSE, USE.NAMES = TRUE)

env  将被使用的环境

all.names  逻辑值，指示是否对所有值使用该函数

USE.NAMES  逻辑值，指示返回的列表结果是否包含命名

例：

> require(stats)

>

> env <- new.env(hash = FALSE) # so the order is fixed

> env$a <- 1:10

> env$beta <- exp(-3:3)

> env$logic <- c(TRUE, FALSE, FALSE, TRUE)

> # what have we there?

> utils::ls.str(env)

a :  int [1:10] 1 2 3 4 5 6 7 8 9 10

beta :  num [1:7] 0.0498 0.1353 0.3679 1 2.7183 ...

logic :  logi [1:4] TRUE FALSE FALSE TRUE

>

> # compute the mean for each list element

>        eapply(env, mean)

$logic

[1] 0.5

$beta

[1] 4.535125

$a

[1] 5.5

> unlist(eapply(env, mean, USE.NAMES = FALSE))

[1] 0.500000 4.535125 5.500000

>

> # median and quartiles for each element (making use of "..." passing):

> eapply(env, quantile, probs = 1:3/4)

$logic

25% 50% 75%

0.0 0.5 1.0

$beta

      25%       50%       75%

0.2516074 1.0000000 5.0536690

$a

 25%  50%  75%

3.25 5.50 7.75

> eapply(env, quantile)

$logic

  0%  25%  50%  75% 100%

 0.0  0.0  0.5  1.0  1.0

$beta

         0%         25%         50%         75%        100%

 0.04978707  0.25160736  1.00000000  5.05366896 20.08553692

$a

   0%   25%   50%   75%  100%

 1.00  3.25  5.50  7.75 10.00

mapply {base}

mapply是sapply的多变量版本。将对...中的每个参数运行FUN函数，如有必要，参数将被循环。

mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)

MoreArgs   FUN函数的其他参数列表

SIMPLIFY   逻辑或者字符串，可以减少结果成为一个向量、矩阵或者更高维阵列，详见sapply的simplify参数

USE.NAMES   逻辑值，如果第一个参数...已被命名，将使用这个字符向量作为名字

例：

> mapply(rep, 1:4, 4:1)

[[1]]

[1] 1 1 1 1

[[2]]

[1] 2 2 2

[[3]]

[1] 3 3

[[4]]

[1] 4

rapply {base}

rapply是lapply的递归版本

rapply(X, FUN, classes = "ANY", deflt = NULL, how = c("unlist", "replace", "list"), ...)

X  一个列表

classes  关于类名的字符向量，或者为any时则匹配任何类

deflt  默认结果，如果使用了how=”replace”，则不能使用

how  字符串匹配三种可能结果

# 2@掌握R语言中的apply函数族 | 粉丝日志

http://blog.fens.me/r-apply/

## 掌握R语言中的apply函数族

[R的极客理想系列文章](http://blog.fens.me/series-r/)，涵盖了R的思想，使用，工具，创新等的一系列要点，以我个人的学习和体验去诠释R的强大。

R语言作为统计学一门语言，一直在小众领域闪耀着光芒。直到大数据的爆发，R语言变成了一门炙手可热的数据分析的利器。随着越来越多的工程背景的人的加入，R语言的社区在迅速扩大成长。现在已不仅仅是统计领域，教育，银行，电商，互联网….都在使用R语言。

要成为有理想的极客，我们不能停留在语法上，要掌握牢固的数学，概率，统计知识，同时还要有创新精神，把R语言发挥到各个领域。让我们一起动起来吧，开始R的极客理想。

**关于作者：**

* 张丹(Conan), 程序员Java,R,PHP,Javascript
* weibo：@Conan\_Z
* blog: [http://blog.fens.me](http://blog.fens.me/)
* email: bsspirit@gmail.com

**转载请注明出处：**  
<http://blog.fens.me/r-apply/>

[](http://blog.fens.me/wp-content/uploads/2016/04/apply-title.png)

**前言**

刚开始接触R语言时，会听到各种的R语言使用技巧，其中最重要的一条就是不要用循环，效率特别低，要用向量计算代替循环计算。

那么，这是为什么呢？原因在于R的循环操作for和while，都是基于R语言本身来实现的，而向量操作是基于底层的C语言函数实现的，从性能上来看，就会有比较明显的差距了。那么如何使用C的函数来实现向量计算呢，就是要用到apply的家族函数，包括apply, sapply, tapply, mapply, lapply, rapply, vapply, eapply等。

**目录**

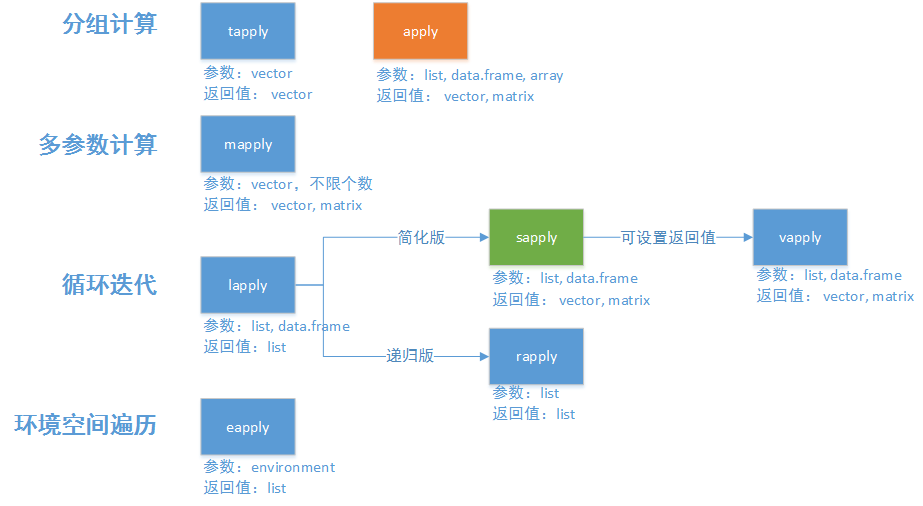
1. apply的家族函数
2. apply函数
3. lapply函数
4. sapply函数
5. vapply函数
6. mapply函数
7. tapply函数
8. rapply函数
9. eapply函数

## 1. apply的家族函数

apply函数族是R语言中数据处理的一组核心函数，通过使用apply函数，我们可以实现对数据的循环、分组、过滤、类型控制等操作。但是，由于在R语言中apply函数与其他语言循环体的处理思路是完全不一样的，所以apply函数族一直是使用者玩不转一类核心函数。

很多R语言新手，写了很多的for循环代码，也不愿意多花点时间把apply函数的使用方法了解清楚，最后把R代码写的跟C似得，我严重鄙视只会写for的R程序员。

apply函数本身就是解决数据循环处理的问题，为了面向不同的数据类型，不同的返回值，apply函数组成了一个函数族，包括了8个功能类似的函数。这其中有些函数很相似，有些也不是太一样的。

[](http://blog.fens.me/wp-content/uploads/2016/04/apply.png)

我一般最常用的函数为apply和sapply，下面将分别介绍这8个函数的定义和使用方法。

## 2. apply函数

apply函数是最常用的代替for循环的函数。apply函数可以对矩阵、数据框、数组(二维、多维)，按行或列进行循环计算，对子元素进行迭代，并把子元素以参数传递的形式给自定义的FUN函数中，并以返回计算结果。

函数定义：

apply(X, MARGIN, FUN, ...)

参数列表：

* X:数组、矩阵、数据框
* MARGIN: 按行计算或按按列计算，1表示按行，2表示按列
* FUN: 自定义的调用函数
* …: 更多参数，可选

比如，对一个矩阵的每一行求和，下面就要用到apply做循环了。

> x<-matrix(1:12,ncol=3)

> apply(x,1,sum)

[1] 15 18 21 24

下面计算一个稍微复杂点的例子，按行循环，让数据框的x1列加1，并计算出x1,x2列的均值。

# 生成data.frame

> x <- cbind(x1 = 3, x2 = c(4:1, 2:5)); x

x1 x2

[1,] 3 4

[2,] 3 3

[3,] 3 2

[4,] 3 1

[5,] 3 2

[6,] 3 3

[7,] 3 4

[8,] 3 5

# 自定义函数myFUN，第一个参数x为数据

# 第二、三个参数为自定义参数，可以通过apply的'...'进行传入。

> myFUN<- function(x, c1, c2) {

+ c(sum(x[c1],1), mean(x[c2]))

+ }

# 把数据框按行做循环，每行分别传递给myFUN函数，设置c1,c2对应myFUN的第二、三个参数

> apply(x,1,myFUN,c1='x1',c2=c('x1','x2'))

[,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]

[1,] 4.0 4 4.0 4 4.0 4 4.0 4

[2,] 3.5 3 2.5 2 2.5 3 3.5 4

通过这个上面的自定义函数myFUN就实现了，一个常用的循环计算。

如果直接用for循环来实现，那么代码如下：

# 定义一个结果的数据框

> df<-data.frame()

# 定义for循环

> for(i in 1:nrow(x)){

+ row<-x[i,] # 每行的值

+ df<-rbind(df,rbind(c(sum(row[1],1), mean(row)))) # 计算，并赋值到结果数据框

+ }

# 打印结果数据框

> df

V1 V2

1 4 3.5

2 4 3.0

3 4 2.5

4 4 2.0

5 4 2.5

6 4 3.0

7 4 3.5

8 4 4.0

通过for循环的方式，也可以很容易的实现上面计算过程，但是这里还有一些额外的操作需要自己处理，比如构建循环体、定义结果数据集、并合每次循环的结果到结果数据集。

对于上面的需求，还有第三种实现方法，那就是完成利用了R的特性，通过向量化计算来完成的。

> data.frame(x1=x[,1]+1,x2=rowMeans(x))

x1 x2

1 4 3.5

2 4 3.0

3 4 2.5

4 4 2.0

5 4 2.5

6 4 3.0

7 4 3.5

8 4 4.0

那么，一行就可以完成整个计算过程了。

接下来，我们需要再比较一下3种操作上面性能上的消耗。

# 清空环境变量

> rm(list=ls())

# 封装fun1

> fun1<-function(x){

+ myFUN<- function(x, c1, c2) {

+ c(sum(x[c1],1), mean(x[c2]))

+ }

+ apply(x,1,myFUN,c1='x1',c2=c('x1','x2'))

+ }

# 封装fun2

> fun2<-function(x){

+ df<-data.frame()

+ for(i in 1:nrow(x)){

+ row<-x[i,]

+ df<-rbind(df,rbind(c(sum(row[1],1), mean(row))))

+ }

+ }

# 封装fun3

> fun3<-function(x){

+ data.frame(x1=x[,1]+1,x2=rowMeans(x))

+ }

# 生成数据集

> x <- cbind(x1=3, x2 = c(400:1, 2:500))

# 分别统计3种方法的CPU耗时。

> system.time(fun1(x))

用户 系统 流逝

0.01 0.00 0.02

> system.time(fun2(x))

用户 系统 流逝

0.19 0.00 0.18

> system.time(fun3(x))

用户 系统 流逝

0 0 0

从CPU的耗时来看，用for循环实现的计算是耗时最长的，apply实现的循环耗时很短，而直接使用R语言内置的向量计算的操作几乎不耗时。通过上面的测试，对同一个计算来说，优先考虑R语言内置的向量计算，必须要用到循环时则使用apply函数，应该尽量避免显示的使用for,while等操作方法。

## 3. lapply函数

lapply函数是一个最基础循环操作函数之一，用来对list、data.frame数据集进行循环，并返回和X长度同样的list结构作为结果集，通过lapply的开头的第一个字母’l’就可以判断返回结果集的类型。

函数定义：

lapply(X, FUN, ...)

参数列表：

* X:list、data.frame数据
* FUN: 自定义的调用函数
* …: 更多参数，可选

比如，计算list中的每个KEY对应该的数据的分位数。

# 构建一个list数据集x，分别包括a,b,c 三个KEY值。

> x <- list(a = 1:10, b = rnorm(6,10,5), c = c(TRUE,FALSE,FALSE,TRUE));x

$a

[1] 1 2 3 4 5 6 7 8 9 10

$b

[1] 0.7585424 14.3662366 13.3772979 11.6658990 9.7011387 21.5321427

$c

[1] TRUE FALSE FALSE TRUE

# 分别计算每个KEY对应该的数据的分位数。

> lapply(x,fivenum)

$a

[1] 1.0 3.0 5.5 8.0 10.0

$b

[1] 0.7585424 9.7011387 12.5215985 14.3662366 21.5321427

$c

[1] 0.0 0.0 0.5 1.0 1.0

lapply就可以很方便地把list数据集进行循环操作了，还可以用data.frame数据集按列进行循环，但如果传入的数据集是一个向量或矩阵对象，那么直接使用lapply就不能达到想要的效果了。

比如，对矩阵的列求和。

# 生成一个矩阵

> x <- cbind(x1=3, x2=c(2:1,4:5))

> x; class(x)

x1 x2

[1,] 3 2

[2,] 3 1

[3,] 3 4

[4,] 3 5

[1] "matrix"

# 求和

> lapply(x, sum)

[[1]]

[1] 3

[[2]]

[1] 3

[[3]]

[1] 3

[[4]]

[1] 3

[[5]]

[1] 2

[[6]]

[1] 1

[[7]]

[1] 4

[[8]]

[1] 5

lapply会分别循环矩阵中的每个值，而不是按行或按列进行分组计算。

如果对数据框的列求和。

> lapply(data.frame(x), sum)

$x1

[1] 12

$x2

[1] 12

lapply会自动把数据框按列进行分组，再进行计算。

## 4. sapply函数

sapply函数是一个简化版的lapply，sapply增加了2个参数simplify和USE.NAMES，主要就是让输出看起来更友好，返回值为向量，而不是list对象。

函数定义：

sapply(X, FUN, ..., simplify=TRUE, USE.NAMES = TRUE)

参数列表：

* X:数组、矩阵、数据框
* FUN: 自定义的调用函数
* …: 更多参数，可选
* simplify: 是否数组化，当值array时，输出结果按数组进行分组
* USE.NAMES: 如果X为字符串，TRUE设置字符串为数据名，FALSE不设置

我们还用上面lapply的计算需求进行说明。

> x <- cbind(x1=3, x2=c(2:1,4:5))

# 对矩阵计算，计算过程同lapply函数

> sapply(x, sum)

[1] 3 3 3 3 2 1 4 5

# 对数据框计算

> sapply(data.frame(x), sum)

x1 x2

12 12

# 检查结果类型，sapply返回类型为向量，而lapply的返回类型为list

> class(lapply(x, sum))

[1] "list"

> class(sapply(x, sum))

[1] "numeric"

如果simplify=FALSE和USE.NAMES=FALSE，那么完全sapply函数就等于lapply函数了。

> lapply(data.frame(x), sum)

$x1

[1] 12

$x2

[1] 12

> sapply(data.frame(x), sum, simplify=FALSE, USE.NAMES=FALSE)

$x1

[1] 12

$x2

[1] 12

对于simplify为array时，我们可以参考下面的例子，构建一个三维数组，其中二个维度为方阵。

> a<-1:2

# 按数组分组

> sapply(a,function(x) matrix(x,2,2), simplify='array')

, , 1

[,1] [,2]

[1,] 1 1

[2,] 1 1

, , 2

[,1] [,2]

[1,] 2 2

[2,] 2 2

# 默认情况，则自动合并分组

> sapply(a,function(x) matrix(x,2,2))

[,1] [,2]

[1,] 1 2

[2,] 1 2

[3,] 1 2

[4,] 1 2

对于字符串的向量，还可以自动生成数据名。

> val<-head(letters)

# 默认设置数据名

> sapply(val,paste,USE.NAMES=TRUE)

a b c d e f

"a" "b" "c" "d" "e" "f"

# USE.NAMES=FALSE，则不设置数据名

> sapply(val,paste,USE.NAMES=FALSE)

[1] "a" "b" "c" "d" "e" "f"

## 5. vapply函数

vapply类似于sapply，提供了FUN.VALUE参数，用来控制返回值的行名，这样可以让程序更健壮。

函数定义：

vapply(X, FUN, FUN.VALUE, ..., USE.NAMES = TRUE)

参数列表：

* X:数组、矩阵、数据框
* FUN: 自定义的调用函数
* FUN.VALUE: 定义返回值的行名row.names
* …: 更多参数，可选
* USE.NAMES: 如果X为字符串，TRUE设置字符串为数据名，FALSE不设置

比如，对数据框的数据进行累计求和，并对每一行设置行名row.names

# 生成数据集

> x <- data.frame(cbind(x1=3, x2=c(2:1,4:5)))

# 设置行名，4行分别为a,b,c,d

> vapply(x,cumsum,FUN.VALUE=c('a'=0,'b'=0,'c'=0,'d'=0))

x1 x2

a 3 2

b 6 3

c 9 7

d 12 12

# 当不设置时，为默认的索引值

> a<-sapply(x,cumsum);a

x1 x2

[1,] 3 2

[2,] 6 3

[3,] 9 7

[4,] 12 12

# 手动的方式设置行名

> row.names(a)<-c('a','b','c','d')

> a

x1 x2

a 3 2

b 6 3

c 9 7

d 12 12

通过使用vapply可以直接设置返回值的行名，这样子做其实可以节省一行的代码，让代码看起来更顺畅，当然如果不愿意多记一个函数，那么也可以直接忽略它，只用sapply就够了。

## 2@6. mapply函数

mapply也是sapply的变形函数，类似多变量的sapply，但是参数定义有些变化。第一参数为自定义的FUN函数，第二个参数’…’可以接收多个数据，作为FUN函数的参数调用。

函数定义：

mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE,USE.NAMES = TRUE)

参数列表：

* FUN: 自定义的调用函数
* …: 接收多个数据
* MoreArgs: 参数列表
* SIMPLIFY: 是否数组化，当值array时，输出结果按数组进行分组
* USE.NAMES: 如果X为字符串，TRUE设置字符串为数据名，FALSE不设置

比如，比较3个向量大小，按索引顺序取较大的值。

> set.seed(1)

# 定义3个向量

> x<-1:10

> y<-5:-4

> z<-round(runif(10,-5,5))

# 按索引顺序取较大的值。

> mapply(max,x,y,z)

[1] 5 4 3 4 5 6 7 8 9 10

再看一个例子，生成4个符合正态分布的数据集，分别对应的均值和方差为c(1,10,100,1000)。

> set.seed(1)

# 长度为4

> n<-rep(4,4)

# m为均值，v为方差

> m<-v<-c(1,10,100,1000)

# 生成4组数据，按列分组

> mapply(rnorm,n,m,v)

[,1] [,2] [,3] [,4]

[1,] 0.3735462 13.295078 157.57814 378.7594

[2,] 1.1836433 1.795316 69.46116 -1214.6999

[3,] 0.1643714 14.874291 251.17812 2124.9309

[4,] 2.5952808 17.383247 138.98432 955.0664

由于mapply是可以接收多个参数的，所以我们在做数据操作的时候，就不需要把数据先合并为data.frame了，直接一次操作就能计算出结果了。

## 7. tapply函数

tapply用于分组的循环计算，通过INDEX参数可以把数据集X进行分组，相当于group by的操作。

函数定义：

tapply(X, INDEX, FUN = NULL, ..., simplify = TRUE)

参数列表：

* X: 向量
* INDEX: 用于分组的索引
* FUN: 自定义的调用函数
* …: 接收多个数据
* simplify : 是否数组化，当值array时，输出结果按数组进行分组

比如，计算不同品种的鸢尾花的花瓣(iris)长度的均值。

# 通过iris$Species品种进行分组

> tapply(iris$Petal.Length,iris$Species,mean)

setosa versicolor virginica

1.462 4.260 5.552

对向量x和y进行计算，并以向量t为索引进行分组，求和。

> set.seed(1)

# 定义x,y向量

> x<-y<-1:10;x;y

[1] 1 2 3 4 5 6 7 8 9 10

[1] 1 2 3 4 5 6 7 8 9 10

# 设置分组索引t

> t<-round(runif(10,1,100)%%2);t

[1] 1 2 2 1 1 2 1 0 1 1

# 对x进行分组求和

> tapply(x,t,sum)

0 1 2

8 36 11

由于tapply只接收一个向量参考，通过’…’可以把再传给你FUN其他的参数，那么我们想去y向量也进行求和，把y作为tapply的第4个参数进行计算。

> tapply(x,t,sum,y)

0 1 2

63 91 66

得到的结果并不符合我们的预期，结果不是把x和y对应的t分组后求和，而是得到了其他的结果。第4个参数y传入sum时，并不是按照循环一个一个传进去的，而是每次传了完整的向量数据，那么再执行sum时sum(y)=55，所以对于t=0时，x=8 再加上y=55，最后计算结果为63。那么，我们在使用’…’去传入其他的参数的时候，一定要看清楚传递过程的描述，才不会出现的算法上的错误。

## 8. rapply函数

rapply是一个递归版本的lapply，它只处理list类型数据，对list的每个元素进行递归遍历，如果list包括子元素则继续遍历。

函数定义：

rapply(object, f, classes = "ANY", deflt = NULL, how = c("unlist", "replace", "list"), ...)

参数列表：

* object:list数据
* f: 自定义的调用函数
* classes : 匹配类型, ANY为所有类型
* deflt: 非匹配类型的默认值
* how: 3种操作方式，当为replace时，则用调用f后的结果替换原list中原来的元素；当为list时，新建一个list，类型匹配调用f函数，不匹配赋值为deflt；当为unlist时，会执行一次unlist(recursive = TRUE)的操作
* …: 更多参数，可选

比如，对一个list的数据进行过滤，把所有数字型numeric的数据进行从小到大的排序。

> x=list(a=12,b=1:4,c=c('b','a'))

> y=pi

> z=data.frame(a=rnorm(10),b=1:10)

> a <- list(x=x,y=y,z=z)

# 进行排序，并替换原list的值

> rapply(a,sort, classes='numeric',how='replace')

$x

$x$a

[1] 12

$x$b

[1] 4 3 2 1

$x$c

[1] "b" "a"

$y

[1] 3.141593

$z

$z$a

[1] -0.8356286 -0.8204684 -0.6264538 -0.3053884 0.1836433 0.3295078

[7] 0.4874291 0.5757814 0.7383247 1.5952808

$z$b

[1] 10 9 8 7 6 5 4 3 2 1

> class(a$z$b)

[1] "integer"

从结果发现，只有$z$a的数据进行了排序，检查$z$b的类型，发现是integer，是不等于numeric的，所以没有进行排序。

接下来，对字符串类型的数据进行操作，把所有的字符串型加一个字符串’++++’，非字符串类型数据设置为NA。

> rapply(a,function(x) paste(x,'++++'),classes="character",deflt=NA, how = "list")

$x

$x$a

[1] NA

$x$b

[1] NA

$x$c

[1] "b ++++" "a ++++"

$y

[1] NA

$z

$z$a

[1] NA

$z$b

[1] NA

只有$x$c为字符串向量，都合并了一个新字符串。那么，有了rapply就可以对list类型的数据进行方便的数据过滤了。

## 9. eapply函数

对一个环境空间中的所有变量进行遍历。如果我们有好的习惯，把自定义的变量都按一定的规则存储到自定义的环境空间中，那么这个函数将会让你的操作变得非常方便。当然，可能很多人都不熟悉空间的操作，那么请参考文章 [揭开R语言中环境空间的神秘面纱](http://blog.fens.me/r-environments/)，[解密R语言函数的环境空间](http://blog.fens.me/r-environments-function/)。

函数定义：

eapply(env, FUN, ..., all.names = FALSE, USE.NAMES = TRUE)

参数列表：

* env: 环境空间
* FUN: 自定义的调用函数
* …: 更多参数，可选
* all.names: 匹配类型, ANY为所有类型
* USE.NAMES: 如果X为字符串，TRUE设置字符串为数据名，FALSE不设置

下面我们定义一个环境空间，然后对环境空间的变量进行循环处理。

# 定义一个环境空间

> env

# 向这个环境空间中存入3个变量

> env$a <- 1:10

> env$beta <- exp(-3:3)

> env$logic <- c(TRUE, FALSE, FALSE, TRUE)

> env

# 查看env空间中的变量

> ls(env)

[1] "a" "beta" "logic"

# 查看env空间中的变量字符串结构

> ls.str(env)

a : int [1:10] 1 2 3 4 5 6 7 8 9 10

beta : num [1:7] 0.0498 0.1353 0.3679 1 2.7183 ...

logic : logi [1:4] TRUE FALSE FALSE TRUE

计算env环境空间中所有变量的均值。

> eapply(env, mean)

$logic

[1] 0.5

$beta

[1] 4.535125

$a

[1] 5.5

再计算中当前环境空间中的所有变量的占用内存大小。

# 查看当前环境空间中的变量

> ls()

[1] "a" "df" "env" "x" "y" "z" "X"

# 查看所有变量的占用内存大小

> eapply(environment(), object.size)

$a

2056 bytes

$df

1576 bytes

$x

656 bytes

$y

48 bytes

$z

952 bytes

$X

1088 bytes

$env

56 bytes

eapply函数平时很难被用到，但对于R包开发来说，环境空间的使用是必须要掌握的。特别是当R要做为工业化的工具时，对变量的精确控制和管理是非常必要的。

本文全面地介绍了，R语言中的数据循环处理的apply函数族，基本已经可以应对所有的循环处理的情况了。同时，在apply一节中也比较了，3种数据处理方面的性能，R的内置向量计算，要优于apply循环，大幅优于for循环。那么我们在以后的R的开发和使用过程中，应该更多地把apply函数使用好。

忘掉程序员的思维，换成数据的思维，也许你就一下子开朗了。

**转载请注明出处：**  
<http://blog.fens.me/r-apply/>