深究 R 中向量 vector

胡弘宇

10/27/2019

目录

导言

在 R 语言中，向量（vector）是数据存储的基本单位。

向量，即由同类型（mode）数据「数值、字符、逻辑、因子、日期」串接而成的有序元素集。向量的大小在创建时已经确定「内存空间」，因此如果想要添加或删除元素（若仅改变数据内容，可直接重新赋值），只能通过将目标向量重新赋值给原变量的方式（需要注意的是，潜在的内存分配操作将极大地限制 R 脚本执行的速度，尤其是在循环「loop」中。因此当我们自定义函数时，一个优化方向就是尽可能减少内存分配语句）。

与 C 语言类似，R 语言中向量是以单元（cell）的形式存储在某一固定内存空间（系统分配内存时决定）中的，对已生成的向量不能再插入或删除元素。而与 C 语言家族不同，R 语言中单个元素（scalar）没有单独的数据类型，它只不过是向量的一种特例。并且向量索引从 1 开始。[1]

创建向量

使用函数 vector() 或 c() 创建向量

我们希望生成一个长度（length）为 3 的向量，

```
a <- vector(length=3) # assign values to variable a
a # display the value of variable a
```

```
## [1] FALSE FALSE FALSE
```

```
a[1] <- 1; a
```

```
## [1] 1 0 0
```

利用连接函数 c() （concatenate）生成从 1 到 9 的整数（integer）向量，

```
b <- c(1:9); b
```

```
## [1] 1 2 3 4 5 6 7 8 9
```

[1]本文大部分内容来源于《The Art of R Programming》by Norman Matloff

```
mode(b) # which type of data in variable 'b'
```

```
## [1] "numeric"
```

```
class(b)
```

```
## [1] "integer"
```

利用 R 中默认变量 letters 快速获得按照字母排序的字符串,

```
letters # character
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

对向量中元素命名,

```
names(b) <- letters[1:9]; b
```

```
## a b c d e f g h i
## 1 2 3 4 5 6 7 8 9
```

当然，命名不影响所存储数据的类型，只是多了额外的元素名,

```
typeof(b)
```

```
## [1] "integer"
```

使用函数 sep() 及 rep() 快速创建蕴含数据规律的向量

seq() 生成等差数列

要生成内部数据有规律的向量,比「:」运算符与函数 c() 更为一般的函数是 sep()（sequence）,

```
seq(from=10, to=1, by = -2)
```

```
## [1] 10  8  6  4  2
```

```
seq(from=1, to=10, by = 2)
```

```
## [1] 1 3 5 7 9
```

```
seq(from=10, to=1, length=5)
```

```
## [1] 10.00  7.75  5.50  3.25  1.00
```

```
seq(from=1, to=10, length=5)
```

```
## [1]  1.00  3.25  5.50  7.75 10.00
```

注意，当函数 seq() 的实际参数为某一向量时，其将生成相等长度的从整数 1 开始的等差为 1 的整数向量，

```
seq(10:1)
```

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

```
seq(letters[1:10])
```

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

```
class(seq(1:10))
```

```
## [1] "integer"
```

这可用来解决 for 循环中「1:length(foo)」length(foo) 为零的 bug（在计算机术语中，foo 与 bar 只是用来代表名义变量，方便函数的阐述，并无实际意义）。即用 seq(foo) 代替「1:length(foo)」，使得当 foo 为 NULL 时，实际循环迭代次数为 0，而不是 1:length(foo) = (1, 0) 的两次。

```
x <- NULL
1:length(x)
```

```
## [1] 1 0
```

```
seq(x)
```

```
## integer(0)
```

使用函数 rep() 生成内容重复（repeat）的长向量

函数 rep() 拥有两个参数 times 和 each，

```
foo <- seq(1:3); foo
```

```
## [1] 1 2 3
```

```
rep(foo, times=3)
```

```
## [1] 1 2 3 1 2 3 1 2 3
```

```
foo <- letters[1:3]; foo
```

```
## [1] "a" "b" "c"
```

```
rep(foo, each=3)
```

```
## [1] "a" "a" "a" "b" "b" "b" "c" "c" "c"
```

循环补齐

在对两个向量使用运算符时，若两个向量长度不一致，R 将自动循环补齐，即重复较短的向量，直到它与另一个向量长度相匹配。这是 R 向量化运算的核心，必须牢记在心，不然容易出 Silly Bug。

```
a <- c(1,2,3)
a + 2
```

```
## [1] 3 4 5
```

故上例中，R 实质运算过程为先执行循环补齐操作，对加项 2 拓展为长度为 3（因为此处较长向量 a 的长度为 3）的向量 c(2,2,2)，然后执行向量化二元计算 c(1,2,3) + c(2,2,2)，得到向量 c(3,4,5)。后文再结合具体实例展开详述。

索引向量

常规索引：整数序、元素名称、等长度逻辑值

我们希望获得某一向量中的部分元素，在 R 中通常使用函数 '[' 来索引，其格式为向量 1[向量 2]。向量 1 是我们的原始向量，向量 2 为索引角标（index）或条件（condition）。向量 2 可有三种形式：整数序、名称、等长度逻辑值「允许重复提取某元素，但当索引值为逻辑值（logical value）时，由于是向量式条件索引，故无法实现在单步索引中重复筛选」。

```
names(a) <- c(letters[1:3])
a[c(1,1)]
```

```
## a a
## 1 1
```

```
a[c('a', 'a')]
```

```
## a a
## 1 1
```

```
a[c(T,T,F)]
```

```
## a b
## 1 2
```

对向量中元素进行全部索引的两种方案，

```
a[1:length(a)] # 利用 length 函数
```

```
## a b c
## 1 2 3
```

```
a[T] # 利用向量的循环补齐
```

```
## a b c
## 1 2 3
```

利用 which() 函数获得索引值

筛选是从向量中提取满足某一条件的元素。而某些情况下我们希望获得向量中满足条件元素所在的位置，此时可以用 which() 函数，

```
a <- c(5:1)
which(a > 3)
```

```
## [1] 1 2
```

其实现原理是，首先比较变量 a 中元素与循环补齐后的 c(3,3,3,3,3) 值的相对大小，得到逻辑向量 c(TRUE TRUE FALSE FALSE FALSE)，然后 which() 函数将报告出该逻辑向量中哪些位置元素值为 TRUE。

向量化运算

R 语言向量化编程的魅力在于，当一个函数使用了向量化的运算符时，那么该函数也被向量化，即向量输入，向量输出。省掉格式化的循环，使得代码变得简洁又易读。例如，

```
a <- c(1:3)
add2 <- function(x) return(x+2)
add2(a)
```

```
## [1] 3 4 5
```

回忆上文提到的循环补齐策略，R 实质做的运算为 c(1,2,3) + c(2,2,2) = c(3, 4, 5)。此外，牢记 R 是一种函数式语言，它的每一个运算符实际上都是函数，包括「+、-、*、/」。我们采取调用函数的方式进行加减运算，

```
'+'(1,2)
```

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

注意「*」代表向量中元素与元素相乘，而「%*%」才是向量相乘。而%/% 表示对相除结果取整数，%% 则对被除数取余数，

```
a * 2
```

```
## [1] 2 4 6
```

```
a %*% a
```

```
##      [,1]
## [1,]   14
```

```
5 %/% 2
```

```
## [1] 2
```

```
5 %% 2
```

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

使用函数 all() 和 any() 生成单一逻辑值

有点特殊的是函数 all() 和 any()，它们分别报告其参数是否至少有一个或全部为真「TRUE」的逻辑值，即函数返回结果为「TRUE」or 「FALSE」。

```
any(c(1:5) >= 3)
```

```
## [1] TRUE
```

```
all(c(1:5) >= 3)
```

```
## [1] FALSE
```

其计算逻辑为第一步计算「c(1:5) > 3」，得到结果向量 FALSE FALSE TRUE TRUE TRUE。然后 any() 函数将判断这些值中是否至少有一个为 TRUE，all() 函数将判断是否全部为 TRUE，然后函数返回一个真或假的逻辑值。

测试向量是否相等

在解决实际问题中，我们常要测试两个向量是否相等。我们可以使用 all() 函数实现这一目标，

```
a <- c(1:5)
b <- a
all(a == b)
```

```
## [1] TRUE
```

同样，我们可以使用 identical() 函数来判断两个对象是否完全一样，

```
identical(a, b)
```

```
## [1] TRUE
```

因此，通过不同方式实现同一目标，在计算机语言中是非常普遍的，没有标准答案。更多时候是根据我们的需求，对代码做简洁与可阅读性的取舍。在这里还有必要强调数值的类型，请看下例，

```
a <- 1:5
b <- c(1,2,3,4,5)
identical(a, b)
```

```
## [1] FALSE
```

```
mode(a)
```

```
## [1] "numeric"
```

```
mode(b)
```

```
## [1] "numeric"
```

```
typeof(a)
```

```
## [1] "integer"
```

```
typeof(b)
```

```
## [1] "double"
```

这里变量 a 不等于 b 的原因在于计算机会根据不同精度存储数值，符号: 生成的是整数，而 c() 生成的是浮点数。就像酱油和醋，都是调味品（数值），外表也分辨不出来（都是黑色），但酸甜（化学性质）是不同的。虽然说在数学上二者等同，但在计算机内存中，由于存储类型（bites）的不同二者是不相等的。当我们编写自己的程序时，必须注意到这一点。

向量化的 ifelse() 函数

R 的官方文档对 ifelse() 函数介绍如下：ifelse returns a value with the same shape as test which is filled with elements selected from either yes or no depending on whether the element of test is TRUE or FALSE. Missing values in test give missing values in the result. 其调用形式如下，

```
ifelse(test, yes, no)
```

其中 test 即为 for 循环中的条件语句，根据返回的逻辑值相应返回 yes 或 no。若 test 为真，则 ifelse 函数的返回 yes 中相应位数的值，若 test 为假，则返回 no 中相应位数的值。注意其与传统 if-else-then 结构的不同。传统 if-else-then 函数只针对单元数据调用函数，而 ifelse() 由于是向量化函数，故自带 for 循环，例如，

```
ifelse(c(1:10) %% 2 == 1, rep(101:105, each=2), rep(201:205, each=2) )
```

```
##  [1] 101 201 102 202 103 203 104 204 105 205
```

除 0 以外的整数，非奇则为偶，故我们只需看最后两位数，即可知道 1:10 中有五位奇数，五位偶数（该自定义函数不 general，此处只为强调 ifelse 函数的向量化运算）。当我们填入 ifelse() 函数 yes 和 no 的实际参数为一元向量时，

```
ifelse(c(1:10) %% 2 == 1, c(100), c(200) )
```

```
##  [1] 100 200 100 200 100 200 100 200 100 200
```

必须理解透此处的彻循环补齐。我们希望有一个函数帮我们判断奇偶数，当输入值为奇数时返回值 100，偶数时返回值 200。当检验对象为 1:10 时，对应到 test 的实际参数就是 TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE，对应到 yes 的实际参数 100，通过循环补齐成为 rep(100, 10)，no 同样循环补齐为 rep(200, 10)。即虽然我们输入的是标量，但 R 实际执行中将返回循环补齐后的向量中对应元素的值。

NaN 与 NULL 值

在统计数据集中，我们经常遇到缺失值（存在但未知的值），R 中表示为 NaN。而 NULL 表示不存在的值。当数据集中包含 NaN 时，R 中大多数函数将返回 NaN，运算失败。这时我们可以通过将默认参数 na.rm （移除 NaN）设为 TRUE 而获得非空值的统计值。如，

```
a <- c(1:5, NaN, 6:10)
mean(a)
```

```
## [1] NaN
```

```
mean(a, na.rm=TRUE)
```

```
## [1] 5.5
```

相比之下，R 内置函数会自动跳过空值 NULL，

```
a <- c(1:5, NULL, 6:10)
mean(a)
```

```
## [1] 5.5
```

故 NULL 的一个常用用法是在循环中创建向量，其中每次迭代都在原向量基础之上新增一个元素（迭代结果）。在下面这个简单的例子中我们建立了偶数向量，

```
Z <- NULL
for (i in 1:10) if (i %% 2 == 0) Z <- c(Z, i)
Z
```

```
## [1]  2  4  6  8 10
```

此例中，我们首先为变量 Z 分配内存空间（如此方能在后面的语句 Z <- c(Z, i) 中完成规律化的循环过程），没开始循环之前，Z 为空，当第一次循环结束，由于 1 为奇数，故选择不执行 if 条件语句后的函数体，第二次循环开始，2 为偶数，因此对 Z 重新赋值，

```
c(NULL, 2)
```

```
## [1] 2
```

```
length(c(NULL, 2))
```

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

注意，此时 c() 函数将选择性移除空值 NULL。然后依次向 Z 中添加 2、4 等偶数。若我们在前例中选择使用 NaN，则会得到多余的 NaN 值，

```
Z <- NaN
for (i in 1:10) if (i %% 2 == 0) Z <- c(Z, i)
Z
```

```
## [1] NaN   2   4   6   8  10
```

使用 subset() 函数做筛选

对向量使用 subset() 函数是，它与普通的筛选方法的区别在于处理 NaN 值，

```
a <- c(1:5, NaN, 6:10)
subset(a, a> 5)
```

```
## [1]  6  7  8  9 10
```

使用之前提到得普通方法筛选向量元素时，对于 NaN 值 R 会认为是未知的，因此其逻辑值是否大于 5 仍是未知的。而当使用 subset() 函数是，subset() 将默认从结果中剔除 NaN 值。

附录

关于 R 中数据存储格式与形式的说明

在 R 中，向量（vector）是数据存储的基本单位。向量，即由同类型（mode）数据「数值、字符、逻辑、因子、日期」串接而成的有序元素集（注意，R 中并不存在标量，单个 cell 实际上是一元向量）。注意，R 向量的元素的索引（index, subscrip）从 1 开始。

向量和列表（list）的元素可由函数 '[' 及相应数值角标（index）或逻辑值（logical value）、名称（names）来访问。列表是加强版向量，可同时存储不同类型数据，并且每个 cell 可以递归存储任意形式数据（向量、列表、矩阵/数据框），是多类别数据存储的有力工具。

矩阵的定义：矩形（各行、列长度相等）的数值数组。在 R 中，矩阵就是多列（columns）排布的一个数值向量，只不过比向量多了两个附加的属性：行数「nrow」和列数「ncol」。因此矩阵使用双下标作为索引。此外，由于矩阵实际上是个长向量，因此对于矩阵中某单元格数据也可使用单个数值来索引，数值大小为从左到右，以列为序。

一个典型的数据集包括多种不同类型数据，因此有数据框的存在。R 语言中的数据框其实是列表，只不过该列表中每个 cell 由 "矩阵" 数据的一列向量构成。故数据框是各元素都为向量的列表，list 转 data.frame 可直接下指令,

```
bar <- data.frame(list(foo))
```

矩阵是向量的特例，数据框是列表的特例，列表、矩阵、数组实际上都是向量，只不过多了额外的类属性。查看数据属性函数：

1. mode() ； typeof()

查看数据类型，「数值『integer/numeric/double』、字符、逻辑、因子『levels』、日期『origin』」

2. attribute(foo, “which”) <- value

查看类属性，「class, comment, dim, dimnames, names, row.names and tsp」，大多数 R 对象仅仅是「列表 + 类」的组合。

3. structure() 「str」

查看数据结构，将综合显示数据的 mode、class、attribute。

漫谈向量化运算

从内存角度来看，R 采用的是内存计算模式（In-Memory），被处理的数据需要预取到主存（RAM）中。其优点是计算效率高、速度快，但缺点是这样一来能处理的问题规模就非常有限（小于 RAM 的大小）。另一方面，R 的核心是一个单线程的程序，当计算任务多而繁重时，我们就有必要考虑并行化计算以最大化利用计算机的性能，减少等待时间。由于我目前并没有跑过并行计算的任务，因而此节内容只涉及从 C 语言的单步循环与 R 中的向量化运算。

向量化计算是并行计算的天然先驱。向量化于 wikipedia 中的定义是：Vectorization is the more limited process of converting a computer program from a scalar implementation, which processes a single pair of operands at a time, to a vector implementation which processes one operation on multiple pairs of operands at once「向量化计算是一种特殊的并行计算的方式，相比于一般程序在同一时间只执行一个操作的方式，它可以在同一时间执行多次操作，通常是对不同的数据执行同样的一个或一批指令，或者说把指令应用于一个数组/向量」。

例如，在下列自定义函数 f 中当形式参数 x 和 n 都为向量 c(1,2,3) 时，R 采取的运算顺序为先将 x 和 n 的实际参数相加，得到向量 c(2,4,6)，再对该向量运用指数运算符「^」，分别得到 2、4、6 的平方 4、16、36，然后一次性返回结果向量。想想在 for 循环中我们需要做的操作吧，首先要定义一个维度确定的存储计算结果的

变量，以避免多次分配计算机内存而造成时间上的等待，然后构思 for 循环中的访问索引，计算 + 赋值 + 返回结果。而当条件 if 一多我们无法确定最终结果的维度时，还只能对每次循环结果采取拼接的方案，这就是说每次循环结束都伴随一次内存分配的操作……

```
f <- function(x, n) return((x+n)^2)
a <- c(1,2,3)
f(a,a)
```

```
## [1]  4 16 36
```

```
f(a,1)
```

```
## [1]  4  9 16
```

另一方面，我们知道 R 没有标量，当形式参数 n 的取值为单元素向量时，实际计算中 R 将采取循环补齐的策略。如上例中，当我们将形式参数 n 的值设为 1 时，调用函数 f 我们获得结果 c(4,9,16)。这怎么计算出来的呢？c(1,2,3) + c(1,1,1) = c(2,3,4)，然后再进行向量化指数运算，一次性返回 2、3、4 平方的向量。因此，当数据比较复杂而对 R 中的向量化运算符的概念理解不深时，我们就很可能忽略循环补齐步骤而写出含有非常难以察觉的 Error。向量化运算 + 循环补齐 -> Silly Bug

上面介绍的是针对向量的二元运算，那当循环运算的是个性化的函数呢？运算对象不再是一维向量而是二维矩阵呢？那就用 apply 族函数！R 的官方文档对 apply 函数的说明：Returns a vector or array or list of values obtained by applying a function to margins of an array or matrix. 其形式参数有三，X 为我们的数据集；MARGIN 为循环方向（1 代表逐行运用用户自定义函数 FUN，2 代表逐列），当自定义函数参数有多项时，直接以逗号分隔的形式作为 apply 函数的参数即可。由于本文主要在阐述 R 中向量部分内容，而 apply 族函数主要属于矩阵或数组部分的内容，故这里只介绍了 apply 族函数的概念及目标，具体实现方法请大家自行 Google。牢记，R 是函数式编程向量化内核的一门语言，看起来为二维的矩阵亦是以列（columns）为存储单元的列表（list）。因此用向量化运算替代 for 循环是我们的不二选择。

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

使用 apply 族函数还有一个好处，就是强迫我们写短而 general 的自定义函数。若该函数重复操作度较高，我们则将之写入我们本地的 R 用户文件.Rprofile。具体

实现方法是将下面语句添加到本地.Rprofile 文本文件中，即启动 R 终端时作为用户自定义设置自动完成加载，foo.R 中存储的就是我们写好的希望可直接调用得函数命令。如此，兼得代码的简洁与易读性。

```
source("foo.R", keep.source = FALSE)
```

如果问题很难向量化怎么办？当然，R 中不是不能使用 for 循环，而是当使用 for 循环时由于 R 的内存计算模式，会给我们带来不必要的等待时间。当只有一层 for 循环，并且数据集不大时，愿意 for 就 for ！如果是两层 for 循环（下角标量为 3），可以考虑用 apply 去掉一层，如果有三层（4 项下角标得数据比较少见），可以考虑重定义数据存储形式或者优化算法，如考虑运用 list 存储数据。

初入门 R 时，我们可以将 R 的向量化运算用 C 语言的思维将其简单理解为对标量运算的简单循环，然后返回一个包含所有运算结果的向量（这也是在 R 中对向量和列表（list）运用 apply 族函数，相较传统 for 与 while 循环省时省力的优势）。但请注意，二者本质是不同的，for 循环采取的是单步串行运算，而 R 中的向量化运算是对内存中变量的各元素同时做运算（还不是多核心的并行计算）。

R 中常用二元运算符

operators	evaluate
:: :::	access variables in a namespace
$ @	component / slot extraction
[[[	indexing
^	exponentiation (right to left)
- +	unary minus and plus
:	sequence operator
%any%	special operators (including %% an
/	multiply, divide
+ -	(binary) add, subtract
< > <= >= == !=	ordering and comparison
!	negation
& &&	and
| ||	or
~	as in formulate
-> ->»	rightwards assignmet

operators	evaluate
<- «-	assignment (right to left
=	assignment (right to left
?	help (unary and binary)