L3: R basic structure

Nan He

2026-01-18

目录

R basic structures introduction

1. 引言

在生信分析中，最常见的三类基础数据结构为矩阵、列表以及数据框。矩阵可以是基因表达矩阵，也可以是距离矩阵；数据框则更为常见，所有的表格数据都可以用它来存储；列表通常用于存储复杂的、异质的数据对象。

本节目标：理解三者结构差异与索引方式，并掌握 base R 方法对三者结构的清洗与整理范式。

后续我们会在 L4 介绍 `dplyr` 和 `tidyr` 主导的 `tidyverse` 生态，即更高级更简洁的数据分析流程。

2. 列表

列表是 R 中最” 灵活” 的容器：每个元素可以是不同类型（向量/矩阵/数据框/甚至另一个列表）。很多函数的返回结果都是一个列表（如 `strsplit()`、`lm()`、`t.test()` 等）。

2.1 创建列表

R 中构造一个列表很简单，可以直接用 `list()` 函数然后赋值给一个对象：

```
l1 <- list(
  nums = 1:5,
  chars = c("a", "b"),
```

```
  mat = matrix(1:6, nrow = 2),
  df = data.frame(id = 1:3, group = c("A","B","A")),
  nested = list(x = 10, y = 20)
)

l1
```

```
## $nums
## [1] 1 2 3 4 5
##
## $chars
## [1] "a" "b"
##
## $mat
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## $df
##   id group
## 1  1     A
## 2  2     B
## 3  3     A
##
## $nested
## $nested$x
## [1] 10
##
## $nested$y
## [1] 20
```

这里创建了一个由 5 个不同对象组成的列表，并且 **nested** 也是一个列表，所以列表是可以嵌套的。

2.2 查看与结构

返回对应的长度、列表的名称以及结构：

```
length(l1)
```

```
## [1] 5
```

```r
names(l1)
```

```
## [1] "nums"   "chars"  "mat"    "df"     "nested"
```

```r
str(l1)
```

```
## List of 5
##  $ nums  : int [1:5] 1 2 3 4 5
##  $ chars : chr [1:2] "a" "b"
##  $ mat   : int [1:2, 1:3] 1 2 3 4 5 6
##  $ df    :'data.frame':  3 obs. of  2 variables:
##   ..$ id   : int [1:3] 1 2 3
##   ..$ group: chr [1:3] "A" "B" "A"
##  $ nested:List of 2
##   ..$ x: num 10
##   ..$ y: num 20
```

2.3 列表索引

- [] 返回子列表
- [[]] 返回元素本体
- $xx 按名称取元素

```r
l1[1]        # 返回一个只包含第一个元素的列表
```

```
## $nums
## [1] 1 2 3 4 5
```

```r
l1[[1]]      # 返回第一个元素本身（数值向量）
```

```
## [1] 1 2 3 4 5
```

```r
l1$nums      # 通过名称访问元素
```

```
## [1] 1 2 3 4 5
```

2.4 修改与追加

列表支持原位修改并能够添加新的元素：

```r
l1$new_item <- rnorm(3)
l1$nums <- l1$nums * 10


l1
```

```
## $nums
```

```
## [1] 10 20 30 40 50
##
## $chars
## [1] "a" "b"
##
## $mat
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## $df
##   id group
## 1  1     A
## 2  2     B
## 3  3     A
##
## $nested
## $nested$x
## [1] 10
##
## $nested$y
## [1] 20
##
##
## $new_item
## [1] -0.7804762  0.2327688  0.1577329
```

这里追加了一个新的元素 `new_item`，并且修改了原来的 `nums` 元素，将这些数值都变成了原来的 10 倍。

2.5 列表的循环操作

后面我们会详细介绍 `apply()` 家族函数，这里先简单了解 `lapply()` 和 `sapply()`：

```r
set.seed(202501)
res <- list(
  ctrl = rnorm(5),
  treat = rnorm(5, mean = 1)
)

# 对每组做同一种统计
lapply(res, mean)   # 返回列表
```

```
## $ctrl
## [1] -0.6378527
##
## $treat
## [1] 0.6747636
```

```
sapply(res, sd)     # 返回简化的向量
```

```
##      ctrl     treat
## 0.9544665 0.2998659
```

这两个函数都对列表的每个元素应用指定的函数，区别在于返回结果的形式。

3. 矩阵

矩阵本质上是一个二维的原子向量。矩阵的所有元素必须是同一类型，不同类型会被强制转换。矩阵有两个维度：行和列。

3.1 矩阵的构建

矩阵是二维的，有行和列的区别，所以构建时需要指定其结构特征：

```
# 创建一个 3 行 4 列的矩阵，按行填充数据
m1 <- matrix(1:12, nrow = 3, byrow = TRUE)
m1
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
```

```
# 给行列名
rownames(m1) <- paste0("Gene", 1:3)
colnames(m1) <- paste0("Sample", 1:4)
m1
```

```
##       Sample1 Sample2 Sample3 Sample4
## Gene1       1       2       3       4
## Gene2       5       6       7       8
## Gene3       9      10      11      12
```

这里的 `nrow` 设置矩阵的行数，列数则是 `ncol`。`byrow=TRUE` 指按行填充矩阵（默认是按列填充）。

`rownames()` 和 `colnames()` 可以用来访问和修改矩阵的行列名。

3.2 查看矩阵信息

可以用以下函数来查看矩阵的基础信息，方便我们快速了解矩阵的结构：

```
# 查看维度
dim(m1)
```

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

```
# 查看行数列数
nrow(m1); ncol(m1)
```

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

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

```
# 查看矩阵结构
typeof(m1)
```

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

```
str(m1)
```

```
##  int [1:3, 1:4] 1 5 9 2 6 10 3 7 11 4 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:3] "Gene1" "Gene2" "Gene3"
##   ..$ : chr [1:4] "Sample1" "Sample2" "Sample3" "Sample4"
```

3.3 矩阵索引

矩阵的索引规则如下：

```
m1[1, 2]        # 第 1 行第 2 列的元素
```

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

```
m1[1, ]         # 第 1 行的所有元素
```

```
## Sample1 Sample2 Sample3 Sample4
##       1       2       3       4
```

```
m1[, 3]         # 第 3 列的所有元素
```

```
## Gene1 Gene2 Gene3
##     3     7    11
```

```
m1[1:2, 2:4]    # 子矩阵
```

```
##       Sample2 Sample3 Sample4
## Gene1       2       3       4
```

```
## Gene2       6       7       8
```

```
# 按行名列名索引
m1["Gene2", "Sample3"]
```

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

```
m1[c("Gene1","Gene3"), c("Sample1","Sample4")]
```

```
##       Sample1 Sample4
## Gene1       1       4
## Gene3       9      12
```

3.4 常见矩阵运算

我们通常需要对矩阵按行或列进行统计，R 中提供了如下函数来实现：

```
rowSums(m1)  # 求每行元素总和
```

```
## Gene1 Gene2 Gene3
##    10    26    42
```

```
colMeans(m1) # 求每列元素均值
```

```
## Sample1 Sample2 Sample3 Sample4
##       5       6       7       8
```

```
apply(m1, 1, sd)  # 求每行元素标准差
```

```
##    Gene1    Gene2    Gene3
## 1.290994 1.290994 1.290994
```

```
apply(m1, 2, max) # 求每列元素的最大值
```

```
## Sample1 Sample2 Sample3 Sample4
##       9      10      11      12
```

很好记忆：`row` 对应行，`col` 对应列，与具体功能组合就成为一个函数。

3.5 矩阵的拼接

在处理数据时，我们经常需要将两个矩阵或数据框”粘”在一起。R 语言提供了两个非常直观的函数：`rbind()` 和 `cbind()`。

- **`rbind()`** (Row bind)：按行拼接，即把一个矩阵”堆”在另一个上面。要求列数 **(ncol)** 必须相同。
- **`cbind()`** (Column bind)：按列拼接，即把一个矩阵”贴”在另一个右边。要求行数 **(nrow)** 必须相同。

```
# 1. 创建两个小的模拟矩阵
# m_a: 2 行 3 列
```

```r
m_a <- matrix(1:6, nrow = 2, ncol = 3)
colnames(m_a) <- c("Gene1", "Gene2", "Gene3")

# m_b: 2 行 3 列
m_b <- matrix(11:16, nrow = 2, ncol = 3)
colnames(m_b) <- c("Gene1", "Gene2", "Gene3")

print(" 矩阵 m_a:")
```

```
## [1] "矩阵 m_a:"
```

```r
print(m_a)
```

```
##      Gene1 Gene2 Gene3
## [1,]     1     3     5
## [2,]     2     4     6
```

```r
print(" 矩阵 m_b:")
```

```
## [1] "矩阵 m_b:"
```

```r
print(m_b)
```

```
##      Gene1 Gene2 Gene3
## [1,]    11    13    15
## [2,]    12    14    16
```

```r
# 场景 1: 你又测了两个新样本 (m_b), 想加入到旧数据 (m_a) 中
m_total_row <- rbind(m_a, m_b)
print("rbind 结果 (行拼接):")
```

```
## [1] "rbind 结果 (行拼接):"
```

```r
print(m_total_row)
```

```
##      Gene1 Gene2 Gene3
## [1,]     1     3     5
## [2,]     2     4     6
## [3,]    11    13    15
## [4,]    12    14    16
```

```r
# 场景 2: 你给原来的样本测了新的基因
# 创建新的基因数据
m_new_genes <- matrix(21:24, nrow = 2, ncol = 2)
colnames(m_new_genes) <- c("Gene4", "Gene5")
```

```r
m_total_col <- cbind(m_a, m_new_genes)
print("cbind 结果 (列拼接):")
```

```
## [1] "cbind 结果 (列拼接):"
```

```r
print(m_total_col)
```

```
##      Gene1 Gene2 Gene3 Gene4 Gene5
## [1,]     1     3     5    21    23
## [2,]     2     4     6    22    24
```

3.6 apply 函数家族的用法

3.6.1 为什么要用 apply 在 C 语言或 Python 中，我们习惯用 `for` 循环来处理重复任务。但在 R 语言中，for 循环运行效率相对较低（尽管近年来已有优化）。

R 是一种向量化语言，`apply` 家族函数是 R 的核心特性之一。它们的作用是：将一个函数" 应用" 到数据集合（矩阵、列表、向量）的每一个元素上。代码简洁，速度快，因为底层是 C 优化过的。

3.6.2 apply() - 对矩阵/数组应用函数 语法：

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

- `X`: 你的数据对象，通常是矩阵或数组
- `MARGIN`: 操作的维度，1 代表行，2 代表列
- `FUN`: 要执行的函数，如 `sum`、`mean`，或自定义函数
- `...`: 传递给 `FUN` 的额外参数

示例：

```r
# 对 m1 每一行求标准差
apply(m1, 1, sd)
```

```
##    Gene1    Gene2    Gene3
## 1.290994 1.290994 1.290994
```

```r
# 自定义函数：返回每行大于 10 的元素个数
res <- apply(m1, 1, function(x) sum(x > 10))
res
```

```
## Gene1 Gene2 Gene3
##     0     0     2
```

3.6.3 lapply() - 对列表/向量应用函数，返回列表 `lapply` 的输入是列表或向量，输出永远是列表：

```r
# 创建一个列表
my_list <- list(
  A = c(1, 2, 3, 4, 5),
  B = c(10, 20, 30))

# 计算列表每个元素的平均值
res_list <- lapply(my_list, mean)

class(res_list)
```

```
## [1] "list"
```

```r
print(res_list)
```

```
## $A
## [1] 3
##
## $B
## [1] 20
```

3.6.4 sapply() - 简化版的 lapply sapply 是 lapply 的”聪明版”。如果结果可以简化（比如每个元素的计算结果都只有一个数字），它会自动把结果简化为向量或矩阵：

```r
res_vec <- sapply(my_list, mean)

class(res_vec) # 变成了数值向量
```

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

```r
print(res_vec)
```

```
##  A  B
##  3 20
```

3.6.5 tapply() - 分组应用函数 tapply 根据一个因子对数据进行分组，然后分别计算。这相当于 dplyr 包里的 group_by + summarise：

```r
# 使用 iris 数据集
# 任务：计算不同物种 (Species) 的花瓣长度 (Petal.Length) 均值
data(iris)

# tapply(数据向量, 分组因子, 函数)
tapply(iris$Petal.Length, iris$Species, mean)
```

```
##     setosa versicolor  virginica
##      1.462      4.260      5.552
```

3.6.6 apply 家族总结

函数	输入	输出	主要用途
apply()	矩阵/数组	向量/矩阵/列表	按行/列计算
lapply()	列表/向量	列表	对每个元素应用函数
sapply()	列表/向量	向量/矩阵	lapply 的简化版
tapply()	向量 + 因子	数组	分组统计

3.7 缺失值与安全计算

在数据处理中，比如我们需要对有年龄数据的患者计算平均值，此时不能包含年龄缺失的患者。在 R 中，如果任何数值与 NA 进行计算，结果都是 NA，比如：

```
print(1 + NA)
```

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

这时可以设置 na.rm = TRUE 来忽略 NA 值：

```
m2 <- m1 # 先复制一个矩阵，避免原位修改
m2[1, 2] <- NA
m2
```

```
##       Sample1 Sample2 Sample3 Sample4
## Gene1       1      NA       3       4
## Gene2       5       6       7       8
## Gene3       9      10      11      12
```

```
rowMeans(m2)                # 含有 NA
```

```
## Gene1 Gene2 Gene3
##    NA   6.5  10.5
```

```
rowMeans(m2, na.rm = TRUE)   # 忽略 NA
```

```
##     Gene1     Gene2     Gene3
##  2.666667  6.500000 10.500000
```

3.8 稀疏矩阵

在生物信息分析（特别是单细胞测序）中，我们的数据往往非常庞大，但其中绝大多数数值都是 0。普通矩阵会在内存里存储每一个 0，而稀疏矩阵只存储非零值及其坐标，极大节省了内存。

可以用 Matrix::sparseMatrix() 来创建稀疏矩阵：

```
library(Matrix)

# 假设我们知道只有 3 个位置有值：
# (1, 1) 的值是 10
# (2, 3) 的值是 5
# (5, 4) 的值是 9

i <- c(1, 2, 5)   # 行索引
j <- c(1, 3, 4)   # 列索引
x <- c(10, 5, 9)  # 具体数值

# dims 指定矩阵的总大小 (5 行 5 列)
m_sparse <- sparseMatrix(i = i, j = j, x = x, dims = c(5, 5))

print(m_sparse)
```

```
## 5 x 5 sparse Matrix of class "dgCMatrix"
##
## [1,] 10 . . . .
## [2,]  . . 5 . .
## [3,]  . . . . .
## [4,]  . . . . .
## [5,]  . . . 9 .
```

对比普通矩阵和稀疏矩阵存储的大小差别：

```
# 1. 创建一个 1000 x 1000 的矩阵，初始全为 0
big_dense <- matrix(0, nrow = 1000, ncol = 1000)

# 2. 随机填入 1000 个非零值 (仅占 0.1%)
set.seed(123)
idx <- sample(1:1000000, 1000)
big_dense[idx] <- runif(1000)

# 3. 转换为稀疏矩阵
big_sparse <- Matrix(big_dense, sparse = TRUE)

# 4. 对比内存大小
size_dense <- object.size(big_dense)
size_sparse <- object.size(big_sparse)
```

```r
print(paste(" 普通矩阵占用:", format(size_dense, units = "Kb")))
```

```
## [1] "普通矩阵占用: 7812.7 Kb"
```

```r
print(paste(" 稀疏矩阵占用:", format(size_sparse, units = "Kb")))
```

```
## [1] "稀疏矩阵占用: 17.1 Kb"
```

```r
# 计算节省了多少倍
ratio <- as.numeric(size_dense) / as.numeric(size_sparse)
print(paste(" 稀疏矩阵节省了约", round(ratio, 1), " 倍的内存! "))
```

```
## [1] "稀疏矩阵节省了约 457.1 倍的内存! "
```

结果显示稀疏矩阵可以节省数百倍的内存空间!

4. 数据框

数据框可能是我们平常最常见的一类数据结构了。简单来说就是一张 Excel 表，每一列都是一个向量，允许不同列是不同类型。就好比一张存储班级学生信息的表，可以有数值型的年龄，也可以有字符型的姓名。

4.1 创建数据框

使用 `data.frame()` 来创建:

```r
df <- data.frame(
  sample_id = paste0("S", 1:6),
  group = c("Ctrl","Ctrl","Treat","Treat","Treat","Ctrl"),
  age = c(45, 52, 60, 58, NA, 49),
  stringsAsFactors = FALSE)

df
```

```
##   sample_id group age
## 1        S1  Ctrl  45
## 2        S2  Ctrl  52
## 3        S3 Treat  60
## 4        S4 Treat  58
## 5        S5 Treat  NA
## 6        S6  Ctrl  49
```

这里显式指定 *列名 = 数据*。向量长度必须一致，因为这是一个表格结构。

参数 `stringsAsFactors = FALSE` 表示不自动将字符串转换为因子类型。在 R 4.0 之后，该参数默认为 `FALSE`。

4.2 查看与结构

```
str(df)
```

```
## 'data.frame':    6 obs. of  3 variables:
##  $ sample_id: chr  "S1" "S2" "S3" "S4" ...
##  $ group    : chr  "Ctrl" "Ctrl" "Treat" "Treat" ...
##  $ age      : num  45 52 60 58 NA 49
```

```
dim(df)
```

```
## [1] 6 3
```

```
head(df, 3)
```

```
##   sample_id group age
## 1        S1  Ctrl  45
## 2        S2  Ctrl  52
## 3        S3 Treat  60
```

```
summary(df)
```

```
##   sample_id            group                age
##  Length:6           Length:6           Min.   :45.0
##  Class :character   Class :character   1st Qu.:49.0
##  Mode  :character   Mode  :character   Median :52.0
##                                        Mean   :52.8
##                                        3rd Qu.:58.0
##                                        Max.   :60.0
##                                        NA's   :1
```

4.3 数据框索引

遵循 `[row, col]` 的 subset 规则。用美元符 `$` 来取某一列：

```
df[1, ]                          # 第 1 行
```

```
##   sample_id group age
## 1        S1  Ctrl  45
```

```
df[, "group"]                    # group 列
```

```
## [1] "Ctrl"  "Ctrl"  "Treat" "Treat" "Treat" "Ctrl"
```

```
df[1:3, c("sample_id", "age")]   # 前三行的 sample_id 和 age 列
```

```
##   sample_id age
```

```
## 1        S1  45
## 2        S2  52
## 3        S3  60
```

```
df$age                                  # 取出 age 列，返回向量
```

```
## [1] 45 52 60 58 NA 49
```

4.4 合并与拼接

在数据分析中，我们经常需要合并两个表。比如有人口学数据和结局数据两张表，需要根据共同的患者 ID 合并到一起。

合并需要两张表具有相同的”关键列”（key），根据这一列进行 `merge()`：

```
# 创建一个 metadata 数据框
meta <- data.frame(
  sample_id = paste0("S", c(1, 2, 3, 4, 5, 6)),
  batch = c("B1", "B1", "B2", "B2", "B1", "B2"),
  stringsAsFactors = FALSE
)

# 按照 sample_id 进行合并
df_merge <- merge(df, meta, by = "sample_id")

df_merge
```

```
##   sample_id group age batch
## 1        S1  Ctrl  45    B1
## 2        S2  Ctrl  52    B1
## 3        S3 Treat  60    B2
## 4        S4 Treat  58    B2
## 5        S5 Treat  NA    B1
## 6        S6  Ctrl  49    B2
```

注意事项： - `merge()` 默认进行内连接（inner join），只保留两表都有的记录 - 可以用 `all.x = TRUE` 进行左连接，`all.y = TRUE` 进行右连接 - `all = TRUE` 进行全连接

4.5 行名

在 Excel 中，每一行只有行号（1, 2, 3…）。但在 R 的数据框中，每一行可以拥有自己的名字（Row names）。在生信分析（如 DESeq2, limma）中，我们通常把基因名作为行名，每一列作为样本。

行名必须是唯一的。

```r
# 1. 查看当前行名（默认是 "1", "2", "3"...）
rownames(df)
```

```
## [1] "1" "2" "3" "4" "5" "6"
```

```r
# 2. 修改行名：把 sample_id 这一列变成行名
rownames(df) <- df$sample_id

df
```

```
##    sample_id group age
## S1        S1  Ctrl  45
## S2        S2  Ctrl  52
## S3        S3 Treat  60
## S4        S4 Treat  58
## S5        S5 Treat  NA
## S6        S6  Ctrl  49
```

```r
# 3. 这时候取数据就可以用行名了
df["S1", "age"]
```

```
## [1] 45
```

注意：行名不允许重复！如果有两个样本叫 *"S1"*，赋值行名时会报错。

4.6 增加与修改列

在分析过程中经常需要计算新变量（例如根据身高体重算 BMI）。最常用的方法是 $ 符号：

```r
# 1. 增加一列新数据 (BMI)
df$bmi <- c(22.5, 24.0, 26.5, 23.0, 21.0, 24.5)

# 2. 修改现有列
# 比如发现 S2 的年龄录错了，应该是 53
df$age[2] <- 53

df
```

```
##    sample_id group age  bmi
## S1        S1  Ctrl  45 22.5
## S2        S2  Ctrl  53 24.0
## S3        S3 Treat  60 26.5
## S4        S4 Treat  58 23.0
## S5        S5 Treat  NA 21.0
```

```
## S6        S6  Ctrl  49 24.5
```

4.7 缺失值处理

在 df 中，S5 的 age 是缺失的，如果不进行处理，计算时会出现问题。

```
# 直接计算均值会得到 NA
mean(df$age)
```

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

```
# 忽略 NA 计算 (na.rm = TRUE)
mean(df$age, na.rm = TRUE)
```

```
## [1] 53
```

```
# 筛选出没有缺失值的行 (na.omit)
# 这在做回归分析前非常常用，剔除所有含有 NA 的样本
df_clean <- na.omit(df)
df_clean
```

```
##    sample_id group age  bmi
## S1        S1  Ctrl  45 22.5
## S2        S2  Ctrl  53 24.0
## S3        S3 Treat  60 26.5
## S4        S4 Treat  58 23.0
## S6        S6  Ctrl  49 24.5
```

4.8 drop=FALSE 的陷阱

这是 Base R 中一个非常隐蔽的坑。当你从数据框中只取某一列时，R 会默认把它降维成一个向量。但有时候我们希望它依然保持为数据框（一列的数据框），以免后续代码报错。

```
# 默认情况: 返回向量
class(df[, "age"])
```

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

```
# 加上 drop = FALSE: 返回依然是数据框
single_col_df <- df[, "age", drop = FALSE]

class(single_col_df)
```

```
## [1] "data.frame"
```

```
single_col_df
```

```
##    age
## S1  45
## S2  53
## S3  60
## S4  58
## S5  NA
## S6  49
```

最佳实践：在编写可重用代码时，建议总是使用 `drop = FALSE`，或者直接使用 `df["age"]` 这种方式（它默认不会 drop）。

总结

本节介绍了 R 语言中三种最重要的数据结构：

1. **列表（List）**：最灵活，可以存储异质数据，是函数返回复杂结果的常用形式
2. **矩阵（Matrix）**：二维同质数据，适合数值计算，在生信中常用于基因表达矩阵
3. **数据框（Data Frame）**：最常用的表格结构，允许不同列有不同类型

掌握这三种数据结构及其操作方法，是进行数据分析的基础。在下一节课中，我们将学习 `tidyverse` 生态系统，它提供了更优雅的数据处理方式。