L5: R 流程控制

Nan He, School of Basic Medical Sciences, Southern Medical University

2026-01-23

目录

R 流程控制

1. 引言

在前几节中，我们学习了 R 的数据结构和基于 `tidyverse` 的数据处理范式。但在实际编程中，我们还需要控制代码的执行流程：根据条件执行不同操作、重复执行某些代码块、在满足条件时跳出循环等。

本节目标：掌握 R 语言中的条件语句、循环语句，以及相关的辅助函数，为编写复杂的分析脚本和自定义函数打下基础。

2. 条件语句

条件语句用于根据条件的真假来决定执行哪段代码。

2.1 if-else 语句

最基本的条件语句结构：

```
if (条件) {
  # 条件为 TRUE 时执行
```

```
} else {
  # 条件为 FALSE 时执行
}
```

示例：判断基因表达是否上调

```
log2FC <- 1.5
threshold <- 1

if (log2FC > threshold) {
  result <- " 上调基因"
} else {
  result <- " 非上调基因"
}

print(result)
```

```
## [1] "上调基因"
```

这里需要注意两点，第一不要忘记 `if` 后面的括号，第二是花括号一定要对齐。因为很多时候 R 和 python 混用，而 python 中是通过缩进来控制模块运行的，而不是花括号。

2.2 if-else if-else 多条件判断

当需要判断多个条件时，可以使用 `else if` 来控制其他条件：

```
log2FC <- -2.3

if (log2FC > 1) {
  status <- " 显著上调"
} else if (log2FC < -1) {
  status <- " 显著下调"
} else {
  status <- " 无显著变化"
}

print(status)
```

```
## [1] "显著下调"
```

注意事项：- 条件必须返回单个 `TRUE` 或 `FALSE`，不能是向量 - 条件判断按顺序执行，一旦匹配成功就不再检查后续条件

2.3 ifelse() 向量化条件

if-else 只能处理单个值，而 ifelse() 可以对向量进行条件判断：

```
# 语法：ifelse(条件向量, TRUE 时的值, FALSE 时的值)

scores <- c(85, 62, 90, 45, 78)
grades <- ifelse(scores >= 60, " 及格", " 不及格")

print(grades)
```

```
## [1] "及格"   "及格"   "及格"   "不及格" "及格"
```

嵌套使用 ifelse() 实现多条件：

```
log2FC_vec <- c(2.5, 0.3, -1.8, -0.5, 1.2)

status_vec <- ifelse(log2FC_vec > 1, " 上调",
                ifelse(log2FC_vec < -1, " 下调", " 无变化"))

data.frame(log2FC = log2FC_vec, status = status_vec)
```

```
##   log2FC status
## 1    2.5   上调
## 2    0.3 无变化
## 3   -1.8   下调
## 4   -0.5 无变化
## 5    1.2   上调
```

2.4 dplyr::case_when() 多条件向量化

嵌套 ifelse() 可读性差，而 case_when() 提供了更优雅的解决方案：

```
library(dplyr)

log2FC_vec <- c(2.5, 0.3, -1.8, -0.5, 1.2, NA)
pvalue_vec <- c(0.001, 0.2, 0.01, 0.5, 0.03, 0.001)

result <- case_when(
  is.na(log2FC_vec) ~ " 缺失",
  log2FC_vec > 1 & pvalue_vec < 0.05 ~ " 显著上调",
  log2FC_vec < -1 & pvalue_vec < 0.05 ~ " 显著下调",
  TRUE ~ " 无显著变化"  # 默认情况（类似 else）
)
```

```r
data.frame(log2FC = log2FC_vec, pvalue = pvalue_vec, result = result)
```

```
##   log2FC pvalue     result
## 1    2.5  0.001   显著上调
## 2    0.3  0.200 无显著变化
## 3   -1.8  0.010   显著下调
## 4   -0.5  0.500 无显著变化
## 5    1.2  0.030   显著上调
## 6     NA  0.001       缺失
```

很明显，这里的 case 就是实例类别，when 就是条件，具体就是条件 ~ 类别的写法。

case_when() 优势：- 可读性强，逻辑清晰 - 支持多条件组合 - 自动处理向量化操作 - 可以优雅处理 NA 值

2.5 switch() 多分支选择

当需要根据一个变量的不同取值执行不同操作时，switch() 比多个 if-else 更简洁：

```r
# 根据分析类型选择统计方法
analysis_type <- "ttest"

method <- switch(analysis_type,
  "ttest" = "Student's t-test",
  "wilcox" = "Wilcoxon rank-sum test",
  "anova" = "One-way ANOVA",
  " 未知方法"  # 默认值
)

print(method)
```

```
## [1] "Student's t-test"
```

用于函数选择：

```r
calculate_stat <- function(x, type = "mean") {
  result <- switch(type,
    "mean" = mean(x, na.rm = TRUE),
    "median" = median(x, na.rm = TRUE),
    "sd" = sd(x, na.rm = TRUE),
    "var" = var(x, na.rm = TRUE),
    stop(" 未知的统计类型: ", type)  # 报错处理
  )
```

```r
  return(result)
}

data <- c(1, 2, 3, 4, 5, NA)
calculate_stat(data, "mean")
```

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

```r
calculate_stat(data, "median")
```

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

3. 循环语句

循环用于重复执行代码块。R 支持三种循环：for、while 和 repeat。

3.1 for 循环

for 循环遍历一个向量或列表的每个元素：

```r
# 基本语法
for (变量 in 序列) {
  # 循环体
}
```

示例 1：遍历向量

```r
genes <- c("TP53", "BRCA1", "EGFR", "KRAS")

for (gene in genes) {
  print(paste(" 正在分析基因:", gene))
}
```

```
## [1] "正在分析基因: TP53"
## [1] "正在分析基因: BRCA1"
## [1] "正在分析基因: EGFR"
## [1] "正在分析基因: KRAS"
```

示例 2：使用索引遍历

```r
values <- c(10, 20, 30, 40, 50)
results <- numeric(length(values))  # 预分配结果向量

for (i in seq_along(values)) {
  results[i] <- values[i] * 2 + 1
```

```
}

print(results)
```

```
## [1]  21  41  61  81 101
```

示例 3：遍历数据框的行

```
df <- data.frame(
  sample = c("S1", "S2", "S3"),
  value = c(100, 150, 80)
)

for (i in 1:nrow(df)) {
  cat(sprintf(" 样本 %s 的值为 %d\n", df$sample[i], df$value[i]))
}
```

```
## 样本 S1 的值为 100
## 样本 S2 的值为 150
## 样本 S3 的值为 80
```

示例 4：嵌套循环

```
# 生成矩阵
mat <- matrix(0, nrow = 3, ncol = 4)

for (i in 1:nrow(mat)) {
  for (j in 1:ncol(mat)) {
    mat[i, j] <- i * j
  }
}

print(mat)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    2    4    6    8
## [3,]    3    6    9   12
```

3.2 while 循环

`while` 循环在条件为 TRUE 时持续执行：

```
# 基本语法
while (条件) {
  # 循环体
  # 必须在循环内修改条件，否则会无限循环
}
```

示例：迭代计算直到收敛

```
# 牛顿法求平方根
x <- 100
guess <- x / 2
tolerance <- 1e-6

while (abs(guess^2 - x) > tolerance) {
  guess <- (guess + x / guess) / 2
  cat(" 当前估计值:", guess, "\n")
}
```

```
## 当前估计值: 26
## 当前估计值: 14.92308
## 当前估计值: 10.81205
## 当前估计值: 10.0305
## 当前估计值: 10.00005
## 当前估计值: 10
```

```
cat("sqrt(100) ", guess, "\n")
```

```
## sqrt(100)  10
```

示例：模拟随机游走

```
set.seed(2026)
position <- 0
steps <- 0
max_steps <- 100

while (abs(position) < 10 && steps < max_steps) {
  position <- position + sample(c(-1, 1), 1)
  steps <- steps + 1
}

cat(" 经过", steps, " 步后，位置为", position, "\n")
```

```
## 经过 84 步后，位置为 -10
```

3.3 repeat 循环

repeat 创建无限循环，必须用 break 跳出：

```
# 基本语法
repeat {
  # 循环体
  if (条件) break
}
```

示例：

```
count <- 0

repeat {
  count <- count + 1
  cat(" 第", count, " 次迭代\n")

  if (count >= 5) {
    cat(" 达到最大迭代次数，退出循环\n")
    break
  }
}
```

```
## 第 1 次迭代
## 第 2 次迭代
## 第 3 次迭代
## 第 4 次迭代
## 第 5 次迭代
## 达到最大迭代次数，退出循环
```

3.4 break 和 next

- break：立即跳出当前循环
- next：跳过本次迭代，进入下一次

```
# break 示例：找到第一个大于 50 的值
values <- c(10, 30, 55, 20, 80)

for (v in values) {
  if (v > 50) {
    cat(" 找到第一个大于 50 的值:", v, "\n")
```

```
    break
  }
  cat(" 检查值:", v, "\n")
}
```

```
## 检查值: 10
## 检查值: 30
## 找到第一个大于50的值: 55
```

```
# next 示例: 跳过 NA 值
data <- c(1, NA, 3, NA, 5)
total <- 0

for (x in data) {
  if (is.na(x)) {
    next  # 跳过 NA
  }
  total <- total + x
}

cat(" 总和（跳过 NA）:", total, "\n")
```

```
## 总和（跳过NA）: 9
```

3.5 循环的性能优化

原则 1：预分配结果向量

```
n <- 10000

# 错误方式: 动态增长向量（慢）
system.time({
  bad_result <- c()
  for (i in 1:n) {
    bad_result <- c(bad_result, i^2)
  }
})
```

```
##    user  system elapsed
##   0.062   0.027   0.089
```

```
# 正确方式: 预分配（快）
system.time({
```

```
  good_result <- numeric(n)
  for (i in 1:n) {
    good_result[i] <- i^2
  }
})
```

```
##    user  system elapsed
##   0.000   0.000   0.001
```

原则 **2**：尽量使用向量化操作

```
# 循环方式
system.time({
  result1 <- numeric(n)
  for (i in 1:n) {
    result1[i] <- sqrt(i)
  }
})
```

```
##    user  system elapsed
##   0.001   0.000   0.001
```

```
# 向量化方式（更快更简洁）
system.time({
  result2 <- sqrt(1:n)
})
```

```
##    user  system elapsed
##       0       0       0
```

4. apply 家族函数

在 L3 中我们简单介绍过 apply 家族，这里更深入地讲解它们如何替代循环。

4.1 apply() - 矩阵/数组操作

```
## 首先创建一个矩阵
mat <- matrix(1:12, nrow = 3, ncol = 4)
rownames(mat) <- paste0("Gene", 1:3)
colnames(mat) <- paste0("Sample", 1:4)
mat
```

```
##       Sample1 Sample2 Sample3 Sample4
## Gene1       1       4       7      10
```

```
## Gene2       2       5       8      11
## Gene3       3       6       9      12
```

```
# 按行计算（MARGIN = 1）
apply(mat, 1, mean)   # 每行均值
```

```
## Gene1 Gene2 Gene3
##   5.5   6.5   7.5
```

```
apply(mat, 1, max)    # 每行最大值
```

```
## Gene1 Gene2 Gene3
##    10    11    12
```

```
# 按列计算（MARGIN = 2）
apply(mat, 2, sd)     # 每列标准差
```

```
## Sample1 Sample2 Sample3 Sample4
##       1       1       1       1
```

```
apply(mat, 2, range)  # 每列范围（返回矩阵）
```

```
##      Sample1 Sample2 Sample3 Sample4
## [1,]       1       4       7      10
## [2,]       3       6       9      12
```

自定义函数：

```
# 计算每行的变异系数（CV = SD/Mean）
apply(mat, 1, function(x) {
  sd(x) / mean(x) * 100
})
```

```
##    Gene1    Gene2    Gene3
## 70.41788 59.58436 51.63978
```

4.2 lapply() 和 sapply() - 列表操作

```
# 创建一个通路基因列表
gene_lists <- list(
  pathway1 = c("TP53", "MDM2", "CDKN1A"),
  pathway2 = c("BRCA1", "BRCA2"),
  pathway3 = c("EGFR", "ERBB2", "ERBB3", "ERBB4")
)
```

```r
# lapply: 返回列表
lapply(gene_lists, length)
```

```
## $pathway1
## [1] 3
##
## $pathway2
## [1] 2
##
## $pathway3
## [1] 4
```

```r
# sapply: 简化为向量
sapply(gene_lists, length)
```

```
## pathway1 pathway2 pathway3
##        3        2        4
```

更复杂的操作:

```r
# 对每个通路的基因进行处理
lapply(gene_lists, function(genes) {
  data.frame(
    gene = genes,
    length = nchar(genes)
  )
})
```

```
## $pathway1
##     gene length
## 1   TP53      4
## 2   MDM2      4
## 3 CDKN1A      6
##
## $pathway2
##    gene length
## 1 BRCA1      5
## 2 BRCA2      5
##
## $pathway3
##    gene length
## 1  EGFR      4
```

```
## 2 ERBB2      5
## 3 ERBB3      5
## 4 ERBB4      5
```

4.3 mapply() - 多参数并行迭代

```r
# 语法: mapply(FUN, ..., MoreArgs)

# 示例: 计算多组数据的加权均值
values <- list(c(1, 2, 3), c(4, 5, 6), c(7, 8, 9))
weights <- list(c(1, 1, 1), c(1, 2, 1), c(1, 1, 2))

mapply(weighted.mean, values, weights)
```

```
## [1] 2.00 5.00 8.25
```

```r
# 创建多个序列
mapply(seq, from = 1:3, to = 4:6)
```

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

4.4 tapply() - 分组操作

```r
# 使用 iris 数据集
data(iris)

# 按物种计算花瓣长度均值
tapply(iris$Petal.Length, iris$Species, mean)
```

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

```r
# 按物种计算多个统计量
tapply(iris$Sepal.Length, iris$Species, function(x) {
  c(mean = mean(x), sd = sd(x), n = length(x))
})
```

```
## $setosa
##       mean         sd          n
```

```
##  5.0060000  0.3524897 50.0000000
##
## $versicolor
##       mean         sd          n
##  5.9360000  0.5161711 50.0000000
##
## $virginica
##       mean         sd          n
##  6.5880000  0.6358796 50.0000000
```

4.5 apply 家族 vs purrr

purrr 包（tidyverse 的一部分）提供了更现代的函数式编程工具：

```
library(purrr)

# map() 类似 lapply()
map(gene_lists, length)
```

```
## $pathway1
## [1] 3
##
## $pathway2
## [1] 2
##
## $pathway3
## [1] 4
```

```
# map_dbl() 返回 double 向量
map_dbl(gene_lists, length)
```

```
## pathway1 pathway2 pathway3
##        3        2        4
```

```
# map_chr() 返回字符向量
map_chr(gene_lists, ~ paste(.x, collapse = ", "))
```

```
##                    pathway1                    pathway2
##        "TP53, MDM2, CDKN1A"             "BRCA1, BRCA2"
##                    pathway3
## "EGFR, ERBB2, ERBB3, ERBB4"
```

```r
# map2() 双参数迭代
map2_dbl(values, weights, weighted.mean)
```

```
## [1] 2.00 5.00 8.25
```

purrr 的优势：- 返回类型可预测（map_dbl、map_chr 等）- 支持简洁的公式语法（~ .x）- 更好的错误处理（safely()、possibly()）

5. 异常处理

在编写健壮的代码时，异常处理至关重要。

5.1 stop() - 抛出错误

使用 stop，可以用来检测输入的合理性，如果不合理，自动报错，防止后面污染：

```r
validate_input <- function(x) {
  if (!is.numeric(x)) {
    stop(" 输入必须是数值型！ ")
  }
  if (any(x < 0)) {
    stop(" 输入不能包含负数！ ")
  }
  return(sqrt(x))
}

validate_input(c(1, 4, 9))    # 正常
```

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

```r
validate_input(c(1, -4, 9))   # 报错
```

```
## Error in validate_input(c(1, -4, 9)): 输入不能包含负数！
```

5.2 warning() - 发出警告

相比 error，warning 一般是被默许存在的，对代码的整体运行没有致命的影响：

```r
safe_divide <- function(x, y) {
  if (any(y == 0)) {
    warning(" 除数包含 0，结果中将出现 Inf")
  }
  return(x / y)
}
```

```r
safe_divide(10, c(2, 0, 5))
```

```
## [1]   5 Inf   2
```

5.3 message() - 输出信息

message 一般是用来在函数中或者自建流程中打印一些必要的步骤提醒：

```r
process_data <- function(data) {
  message(" 开始处理数据...")
  result <- data * 2
  message(" 处理完成！ ")
  return(result)
}

process_data(1:5)
```

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

5.4 tryCatch() - 捕获异常

tryCatch() 允许你捕获并处理错误，避免程序崩溃：

```r
safe_log <- function(x) {
  tryCatch(
    {
      # 尝试执行的代码
      result <- log(x)
      return(result)
    },
    error = function(e) {
      # 发生错误时执行
      message(" 发生错误: ", e$message)
      return(NA)
    },
    warning = function(w) {
      # 发生警告时执行
      message(" 发生警告: ", w$message)
      return(log(abs(x)))
    }
  )
}
```

```r
safe_log(10)       # 正常
```

```
## [1] 2.302585
```

```r
safe_log(-5)       # 警告，返回 log(5)
```

```
## [1] 1.609438
```

```r
safe_log("abc")    # 错误，返回 NA
```

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

批量处理时的应用：

```r
# 批量读取文件，即使某些文件不存在也不中断
files <- c("file1.csv", "file2.csv", "nonexistent.csv")

results <- lapply(files, function(f) {
  tryCatch(
    {
      # 这里用 readr::read_csv(f) 读取真实文件
      message(" 成功读取: ", f)
      return(data.frame(file = f, status = "success"))
    },
    error = function(e) {
      message(" 读取失败: ", f)
      return(data.frame(file = f, status = "failed"))
    }
  )
})

do.call(rbind, results)
```

```
##              file  status
## 1       file1.csv success
## 2       file2.csv success
## 3 nonexistent.csv success
```

5.5 purrr::safely() 和 possibly()

purrr 提供了更优雅的异常处理方式：

```r
library(purrr)
```

```r
# safely() 包装函数，返回 result 和 error 两个组件
safe_sqrt <- safely(sqrt)

safe_sqrt(4)    # 成功
```

```
## $result
## [1] 2
##
## $error
## NULL
```

```r
safe_sqrt("a")  # 失败但不中断
```

```
## $result
## NULL
##
## $error
## <simpleError in .Primitive("sqrt")(x): non-numeric argument to mathematical function>
```

```r
# possibly() 提供默认值
possible_sqrt <- possibly(sqrt, otherwise = NA)

map_dbl(list(4, "a", 9, NULL), possible_sqrt)
```

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

6. 实战案例

6.1 批量处理多个数据文件

```r
# 假设有多个基因表达文件需要处理
file_list <- list.files(path = "data/", pattern = "*.csv", full.names = TRUE)

# 使用 for 循环
all_results <- list()

for (i in seq_along(file_list)) {
  file <- file_list[i]

  tryCatch({
    # 读取数据
    data <- read.csv(file)
```

```
    # 处理数据
    processed <- data |>
      filter(pvalue < 0.05) |>
      mutate(significant = abs(log2FC) > 1)

    # 存储结果
    all_results[[basename(file)]] <- processed
    message(" 成功处理: ", file)

  }, error = function(e) {
    warning(" 处理失败: ", file, " - ", e$message)
  })
}

# 合并所有结果
final_result <- bind_rows(all_results, .id = "source_file")
```

6.2 参数扫描

```
# 测试不同阈值对差异基因数量的影响
set.seed(42)
mock_data <- data.frame(
  gene = paste0("Gene", 1:1000),
  log2FC = rnorm(1000, 0, 1.5),
  pvalue = runif(1000, 0, 1)
)

# 定义阈值范围
fc_thresholds <- c(0.5, 1, 1.5, 2)
p_thresholds <- c(0.01, 0.05, 0.1)

# 参数扫描
results <- data.frame()

for (fc in fc_thresholds) {
  for (p in p_thresholds) {
    n_sig <- sum(abs(mock_data$log2FC) > fc & mock_data$pvalue < p)

    results <- rbind(results, data.frame(
```

```
      fc_threshold = fc,
      p_threshold = p,
      n_significant = n_sig
    ))
  }
}

print(results)
```

```
##    fc_threshold p_threshold n_significant
## 1           0.5        0.01             8
## 2           0.5        0.05            35
## 3           0.5        0.10            70
## 4           1.0        0.01             5
## 5           1.0        0.05            24
## 6           1.0        0.10            53
## 7           1.5        0.01             4
## 8           1.5        0.05            15
## 9           1.5        0.10            31
## 10          2.0        0.01             1
## 11          2.0        0.05             8
## 12          2.0        0.10            17
```

使用 expand.grid() + purrr 的更优雅写法:

```
library(purrr)

params <- expand.grid(
  fc_threshold = fc_thresholds,
  p_threshold = p_thresholds
)

results_purrr <- params |>
  mutate(
    n_significant = map2_dbl(fc_threshold, p_threshold, function(fc, p) {
      sum(abs(mock_data$log2FC) > fc & mock_data$pvalue < p)
    })
  )

print(results_purrr)
```

```
##    fc_threshold p_threshold n_significant
## 1           0.5        0.01             8
## 2           1.0        0.01             5
## 3           1.5        0.01             4
## 4           2.0        0.01             1
## 5           0.5        0.05            35
## 6           1.0        0.05            24
## 7           1.5        0.05            15
## 8           2.0        0.05             8
## 9           0.5        0.10            70
## 10          1.0        0.10            53
## 11          1.5        0.10            31
## 12          2.0        0.10            17
```

6.3 条件判断在数据分析中的应用

```r
# 基因分类函数
classify_genes <- function(data, fc_cutoff = 1, p_cutoff = 0.05) {
  data |>
    mutate(
      regulation = case_when(
        is.na(log2FC) | is.na(pvalue) ~ "Unknown",
        pvalue >= p_cutoff ~ "Not Significant",
        log2FC > fc_cutoff ~ "Up-regulated",
        log2FC < -fc_cutoff ~ "Down-regulated",
        TRUE ~ "Stable"
      )
    )
}

# 应用分类
classified <- classify_genes(mock_data)
table(classified$regulation)
```

```
##
##  Down-regulated Not Significant          Stable    Up-regulated
##              14             947              29              10
```

7. 最佳实践总结

7.1 何时使用循环 vs 向量化

场景	推荐方法
简单数学运算	向量化操作
对向量/列表每个元素应用函数	`apply` 家族或 `purrr`
需要访问前一次迭代的结果	`for` 或 `while` 循环
条件复杂、难以向量化	`for` 循环 + 预分配
需要中途退出	`for` + `break`

7.2 代码可读性建议

1. **优先使用向量化**：R 的核心优势
2. **善用管道符**：`|>` 使流程清晰
3. **使用 `case_when()`**：替代嵌套 `ifelse()`
4. **预分配结果**：避免循环中动态增长向量
5. **添加异常处理**：使用 `tryCatch()` 或 `safely()`
6. **适当添加注释**：说明循环目的和退出条件

7.3 调试技巧

```r
# 1. 使用 print/cat 输出中间结果
for (i in 1:10) {
  result <- some_function(i)
  cat("i =", i, ", result =", result, "\n")
}

# 2. 使用 browser() 进入交互式调试
for (i in 1:10) {
  if (i == 5) browser()  # 在第 5 次迭代时暂停
  result <- some_function(i)
}

# 3. 使用 debug() 跟踪函数执行
debug(my_function)
my_function(x)
undebug(my_function)
```

总结

本节介绍了 R 语言中的流程控制结构：

1. **条件语句**
    - `if-else`：单值条件判断
    - `ifelse()`：向量化条件判断
    - `case_when()`：多条件向量化（推荐）
    - `switch()`：多分支选择
2. **循环语句**
    - `for`：遍历序列
    - `while`：条件循环
    - `repeat`：无限循环 + `break`
    - `break` 和 `next`：控制循环流程
3. **apply 家族**
    - `apply()`：矩阵/数组
    - `lapply()`/`sapply()`：列表/向量
    - `mapply()`：多参数并行
    - `tapply()`：分组操作
4. **异常处理**
    - `stop()`/`warning()`/`message()`
    - `tryCatch()`：捕获并处理异常
    - `purrr::safely()`/`possibly()`

再下一节中，我们会介绍如何在 R 中编写函数。