R 语言编程：基于 tidyverse

第 06 讲　控制结构：分支, 循环

张敬信

2022 年 2 月 10 日

哈尔滨商业大学

一. 分支结构

- 编程中的控制结构，是指分支结构和循环结构。
- 正常程序结构与一步一步解决问题是一致的，即顺序结构，过程中可能需要对不同情形选择走不同的支路，即分支结构[1]，是用**条件语句**做判断以实现分支：

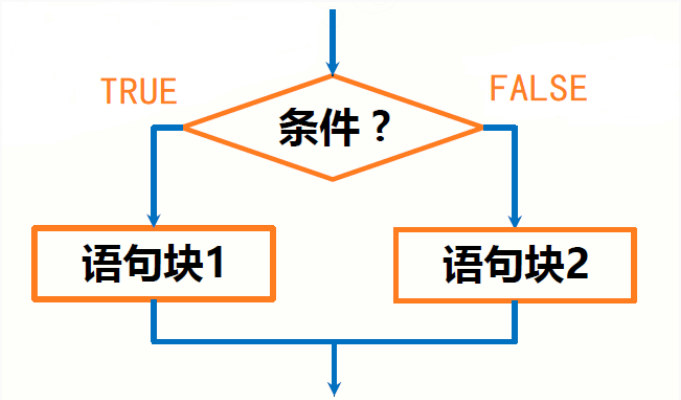


图 1: 分支结构示意图

[1]分支的本意就是，不同分支之间不存在交叉（重叠）.

条件语句

1. **一个分支**

```
if(条件) {
   执行体
}
```

2. **两个分支**

```
if(条件) {
  执行体 1
} else {
  执行体 2
}
```

- 例如，实现计算 $|x|$:

```
if(x < 0) {
  y = -x
} else {
  y = x
}
```

3. 多个分支

```
if(条件 1) {
  执行体 1
} else if(条件 2) {
  执行体 2
} else {
  执行体 n
}
```

- 多个分支的意思是，若满足 “条件 1”，则执行 “执行体 1”；“其他的若” 满足 “条件 2”，则执行 “执行体 2”；“其他的”，执行 “执行体 n”。若需要，中间可以有任意多个 `else if` 块。

switch() 分支

```r
x = "b"
v = switch(x, "a"="apple", "b"="banana", "c"="cherry")
v
#> [1] "banana"
```

- 应用场景：自定义函数时，若需要根据参数不同指示值执行不同代码块。

例 1.5 实现将百分制分数转化为五级制分数

```
if(score >= 90) {                    # 注意不能先写 >=60
  res = " 优"
} else if(score >= 80) {
  res = " 良"
} else if(score >= 70) {
  res = " 中"
} else if(score >= 60) {
  res = " 及格"
} else {
  res = " 不及格"
}
```

关于“条件”

- “条件”是用逻辑表达式表示，必须是返回一个逻辑值 TRUE 或 FALSE；
- 多个逻辑表达式，可以通过逻辑运算符组合以表示复杂条件；
- 多个逻辑值的逻辑向量，可以借助函数 any() 和 all() 得到一个逻辑值；
- 函数 ifelse() 可简化代码，仍以计算 $|x|$ 为例：

```
ifelse(x < 0, -x, x)
```

二. 循环结构

- 编程中减少代码重复的两个工具，一是循环，一是函数。
- **循环**，用来处理对多个同类输入做相同事情（即迭代），如对向量的每个元素做相同操作，对数据框不同列做相同操作、对不同数据集做相同操作等。

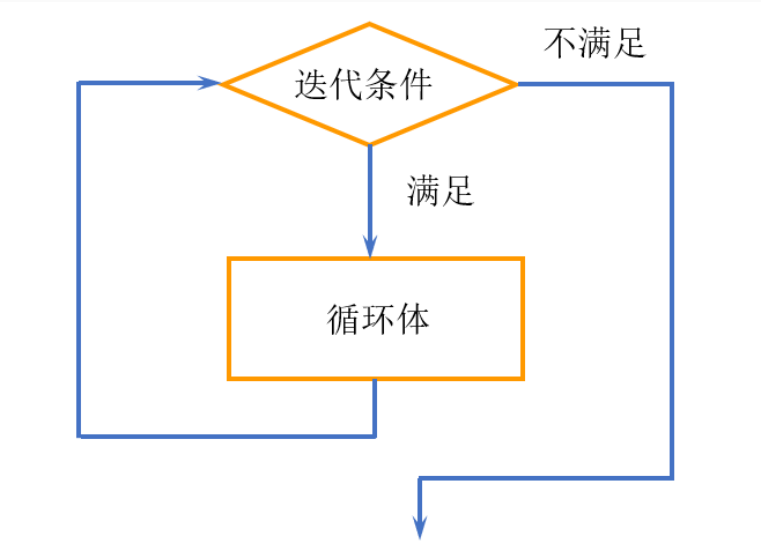


图 2: 循环结构示意图

R 循环的三层境界：

- 第一层：for 循环、while 循环、repeat 循环
- 第二层：apply 函数族
- 第三层：purrr 泛函式编程

关于跳出循环：

- 用关键字 next 跳出本次循环，进入下次循环
- 用关键词 break 跳出循环
- 如今 for 并不慢，只需要注意：
    - 提前为保存循环结果分配存储空间
    - 为循环体中涉及的数据选择合适的数据结构

三. 自带循环迭代

1. for 循环（最常用）

(1) 基本 for 循环

```
library(tidyverse)
df = as_tibble(iris[,1:4])
```

- 用“复制-粘贴”法，计算前 4 列的均值：

```
mean(df[[1]])
#> [1] 5.84
mean(df[[2]])
#> [1] 3.06
mean(df[[3]])
#> [1] 3.76
mean(df[[4]])
#> [1] 1.2
```

- 为了避免 “粘贴-复制多于两次”，改用 `for` 循环实现：

```
output = vector("double", 4)         # 1. 输出
for (i in 1:4) {                     # 2. 迭代器
  output[i] = mean(df[[i]])          # 3. 循环体
}
output
#> [1] 5.84 3.06 3.76 1.20
```

for 循环的三个组件

(1) **输出**：`output = vector("double", 4)`

- 在循环开始之前先设计好输出效率更高，避免每循环一次合并一次结果。

(ii) **迭代器**：`i in 1:4`

- 确定怎么循环：每次 `for` 循环将对 `i` 赋一个 `1:4` 中的值，可将 `i` 理解为代词 `it`.

(iii) **循环体**：`output[i] = mean(df[[i]])`

- 即执行具体操作的代码，它将重复执行，每次对不同的 `i` 值：
  - 第 1 次迭代将执行：`output[1] = mean(df[[1]])`
  - 第 2 次迭代将执行：`output[2] = mean(df[[2]])`
  - ……

for 循环常用操作

(i) 循环模式

- 根据数值索引: `for(i in seq_along(xs))`, 迭代中使用 `x[i]`.
- 根据元素值: `for(x in xs)`, 迭代中使用 `x`.
- 根据名字: `for(nm in names(xs))`, 迭代中使用 `x[nm]`.

(ii) 将每次循环得到的结果合并为一个整体对象

- 避免 “每循环一次，就做一次拼接”。先将结果保存为列表，等循环结束再将列表展开或合并。

- 比如，先创建空列表，再将每次循环的结果依次存入列表：

```
output = list()          # output = NULL 也行
# output = vector("list", 3)
for(i in 1:3) {
  output[[i]] = c(i, i^2)
}
```

2. **while 循环**
   - 适用于迭代次数未知，常用于模拟，while 循环只包含两个组件：条件、循环体

```
while (condition) {
  # 循环体
}
```

- for 循环总可以改写为 while 循环，反之不一定：

```
for (i in seq_along(x)) {
  # 循环体
}
# 等价于
i = 1
while (i <= length(x)) {
  # 循环体
  i = i + 1
}
```

- 用 `while` 循环实现：反复随机生成标准正态分布随机数，若出现值大于 1 则停止

```
set.seed(123)    # 设置随机种子，让结果可重现
while(TRUE) {
  x = rnorm(1)
  print(x)
  if(x > 1) break
}
#> [1] -0.56
#> [1] -0.23
#> [1] 1.56
```

3. repeat 循环

- 重复执行循环体，直到满足退出条件；注意，repeat 循环至少会执行一次循环体

```
repeat{
  # 循环体
  if(退出条件) break
}
```

- repeat 循环等价于：

```
while (TRUE) {
  # 循环体
  if(退出条件) break
}
```

- 例如，用如下泰勒公式近似计算 e:

$$e = 1 + \sum_{k=1}^{\infty} \frac{1}{k!}$$

```
s = 1.0; x = 1; k = 0
repeat{
  k = k + 1
  x = x / k
  s = s + x
  if(x < 1e-10) break
}
stringr::str_glue(" 迭代 {k} 次, 得到 e = {s}")
#> 迭代 14 次, 得到 e = 2.71828182845823
```

四. apply 函数族

- 更建议弃用 apply 函数族，直接用 purrr::map 系列。

⑴ apply(x, MARGIN, FUN, ...)

- 对矩阵、数据框、多维数组，按行或列或页进行循环迭代，即将逐行或逐列或逐页的元素分别传递给函数 FUN 进行迭代计算。
- MARGIN：1 表示按行，2 表示按列，3 表示按页；

```r
x = matrix(1:6, ncol = 3)
x
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
apply(x, 1, mean)           # 按行求均值
#> [1] 3 4
apply(x, 2, mean)           # 按列求均值
#> [1] 1.5 3.5 5.5
apply(df, 2, mean)          # 对前文 df 计算各列的均值
#> Sepal.Length  Sepal.Width Petal.Length  Petal.Width
#>         5.84         3.06         3.76         1.20
```

(2) `tapply(x, INDEX, FUN, ...)`

- 按照因子分组，实现逐分组迭代

```
height = c(165, 170, 168, 172, 159)
sex = factor(c(" 男", " 女", " 男", " 男", " 女"))
tapply(height, sex, mean)   # 计算男女平均身高
#>  男  女
#> 168 164
```

(3) lapply(x, FUN, ...)

- 对向量、列表、数据框逐元、逐成分、逐列分别应用函数 FUN，并返回和 x 长度相同的 list 对象

```
lapply(df, mean)       # 对前文 df 计算各列的均值
#> $Sepal.Length
#> [1] 5.84
#>
#> $Sepal.Width
#> [1] 3.06
#>
#> $Petal.Length
#> [1] 3.76
#>
#> $Petal.Width
#> [1] 1.2
```

(4) `sapply(X, FUN, ..., simplify = TRUE)`

- 是 `lapply()` 的简化版本，多了参数 `simplify`，默认自动简化结果

```
sapply(df, mean)       # 对前文 df 计算各列的均值
#> Sepal.Length  Sepal.Width Petal.Length  Petal.Width
#>         5.84         3.06         3.76         1.20
```

五. purrr 泛函式循环迭代

- 数学上，函数的函数称为泛函；编程中，表示函数作用在函数上，或者说函数包含其他函数作为参数
- 循环迭代，本质上就是将一个函数依次应用（映射）到序列的每一个元素上，表示出来即 `purrr::map_*(x, f)`
- 两点说明：
    - 序列：由一系列可以根据位置索引的元素构成，元素可以很复杂和不同类型；向量、列表、数据框都是序列
    - 将 `x` 作为第一个参数，是便于使用管道

- purrr 泛函式编程解决循环迭代问题的逻辑：
  - 针对序列每个单独的元素，怎么处理它得到正确的结果，将之定义为函数，再 map 到序列中的每一个元素，将得到的多个结果[2]打包到一起返回，并且可以根据想让结果返回什么类型选用 map 后缀。

- 循环迭代返回类型的控制：
  - map_chr, map_lgl, map_dbl, map_int: 返回相应类型向量
  - map_dfr, map_dfc: 返回数据框列表，再按行、按列合并为一个数据框

[2]每个元素作用后返回一个结果.

- purrr 风格公式（匿名函数）：函数参数.f 的一种简写；只需要写清楚它是如何操作序列参数.x 的
  - 一元函数序列参数为.x, 例如 $f(x) = x^2 + 1$ 表示为`.f = ~ .x ^ 2 + 1`
  - 二元函数序列参数为.x, .y, 例如 $f(x, y) = x^2 - 3y$ 表示为`.f = ~ .x ^ 2 - 3 * .y`
  - 还有三元函数序列参数: ..1, ..2, ..3, 所有序列参数...

注： .x 是序列中的一个（代表）元素。这也是**分解**的思维，循环迭代要依次对序列中每个元素做某操作，只需要把对一个元素做的操作写清楚（即.f），剩下的交给 map_*() 就行了。

- `map_*(.x, .f, ...)`: 依次应用一元函数`.f`到一个序列`.x`的每个元素，`...`可设置`.f`的其它参数

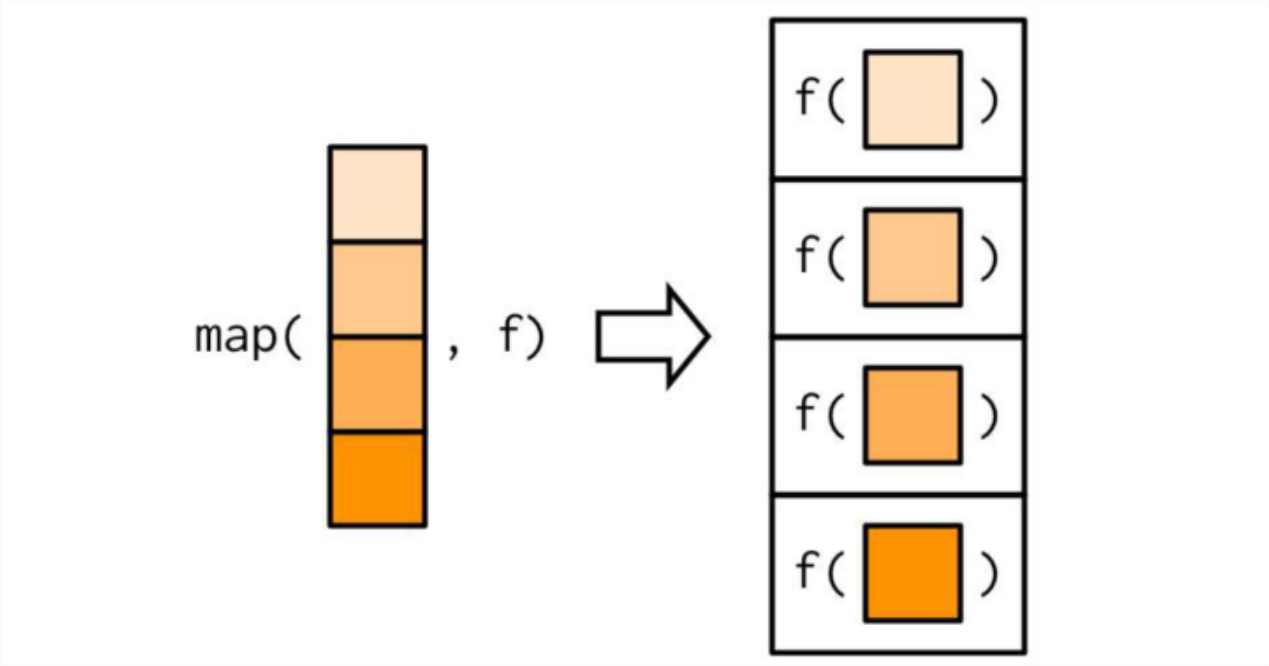


图 3: map 函数作用机制示意图

- `map2_*(.x, .y, .f, ...)`: 依次应用二元函数`.f`到两个序列`.x`, `.y`的每对元素，`...`可设置`.f`的其它参数

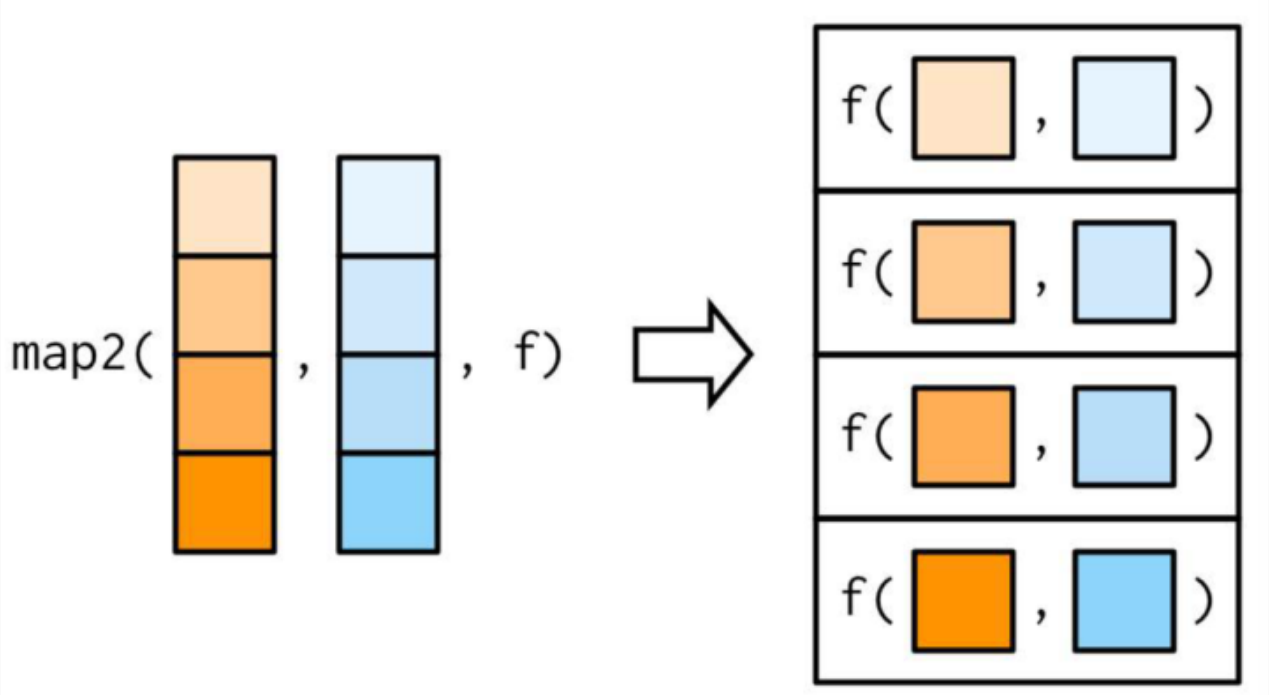


图 4: map2 函数作用机制示意图

- `pmap_*(.l, .f, ...)`: 依次应用多元函数`.f`到多个序列`.l`的每层元素，可实现对数据框逐行迭代，`...`可设置`.f`的其它参数

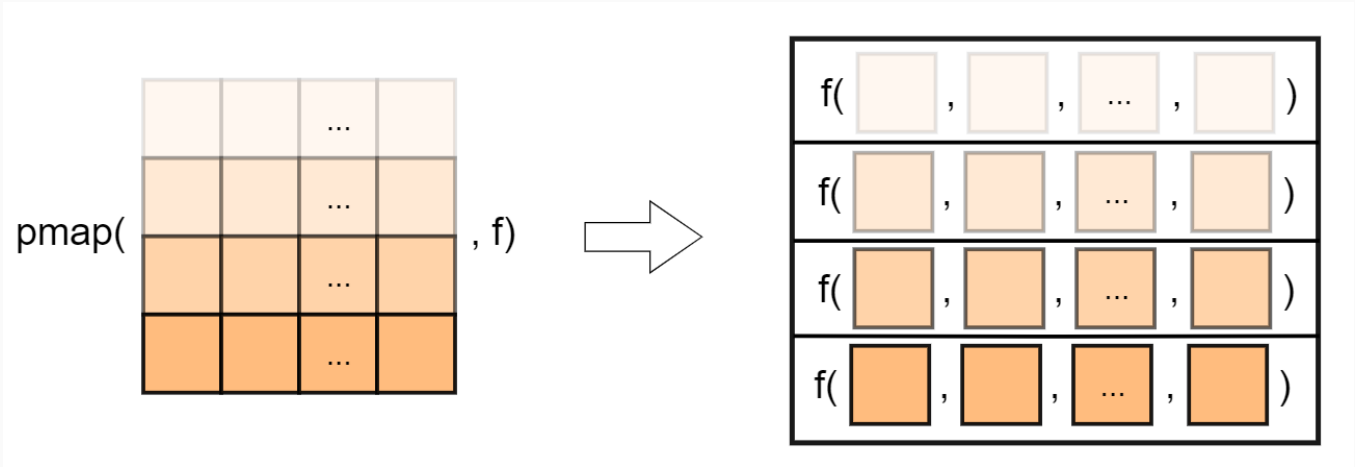


图 5: pmap 函数作用机制示意图

- 其它 purrr 函数
  - walk_*() 系列：只循环迭代做事不返回结果，比如批量保存数据/图形到文件；
  - imap_*() 系列：元素与索引一起迭代；
  - modify_*() 系列：原地依次修改序列对象；
  - reduce()/accumulate(): 可先对序列前两个元素应用函数，再对结果与第 3 个元素应用函数，再对结果与第 4 个元素应用函数，……前者只返回最终结果，后者会返回所有中间结果。

例 3 对数据框逐列迭代

- 数据框是序列，第 1 个元素是第 1 列 `df[[1]]`, 第 2 个元素是第 2 列 `df[[2]]`, ……

```
df = iris[,1:4]
map_dbl(df, mean)        # 求各列均值
#> Sepal.Length  Sepal.Width Petal.Length  Petal.Width
#>         5.84         3.06         3.76         1.20
map_chr(df, mean)
#> Sepal.Length  Sepal.Width Petal.Length  Petal.Width
#>   "5.843333"   "3.057333"   "3.758000"   "1.199333"
```

- 自定义归一化函数

```r
Rescale = function(x, type = "pos") {
  rng = range(x, na.rm = TRUE)   # 计算最小值最大值
  if(type == "pos") {
    (x - rng[1]) / (rng[2] - rng[1])
  } else {
    (rng[2] - x) / (rng[2] - rng[1])
  }
}
```

- 对各列做归一化, 若均为正向指标:

```
map_dfc(df, Rescale)
#> # A tibble: 150 x 4
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width
#>          <dbl>       <dbl>        <dbl>       <dbl>
#> 1        0.222       0.625       0.0678      0.0417
#> 2        0.167       0.417       0.0678      0.0417
#> 3        0.111       0.5         0.0508      0.0417
#> 4        0.0833      0.458       0.0847      0.0417
#> 5        0.194       0.667       0.0678      0.0417
#> # ... with 145 more rows
# 同 map_dfc(df, Rescale, type = "pos")
# 同 map_dfc(df, ~ Rescale(.x, "pos"))
```

- 对各列做归一化，若各列分别为正向, 负向, 负向, 正向

```
type = c("pos", "neg", "neg", "pos")
map2_dfc(df, type, Rescale)
#> # A tibble: 150 x 4
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width
#>          <dbl>       <dbl>        <dbl>       <dbl>
#> 1        0.222       0.375        0.932      0.0417
#> 2        0.167       0.583        0.932      0.0417
#> 3        0.111       0.5          0.949      0.0417
#> 4        0.0833      0.542        0.915      0.0417
#> 5        0.194       0.333        0.932      0.0417
#> # ... with 145 more rows
```

例 4 对数据框逐行迭代

```
pmap_dbl(df[1:10,], ~ mean(c(...)))     # 逐行平均
#>  [1] 2.55 2.38 2.35 2.35 2.55 2.85 2.42 2.52 2.23 2.40
map_dbl(asplit(df[1:10,], 1), mean)
#>    1    2    3    4    5    6    7    8    9   10
#> 2.55 2.38 2.35 2.35 2.55 2.85 2.42 2.52 2.23 2.40
pmap_dbl(df[1:10,], max)                # 逐行最大
#>  [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9
apply(df[1:10,], 1, max)
#>   1   2   3   4   5   6   7   8   9  10
#> 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9
```

例 5 批量读取数据并按行合并

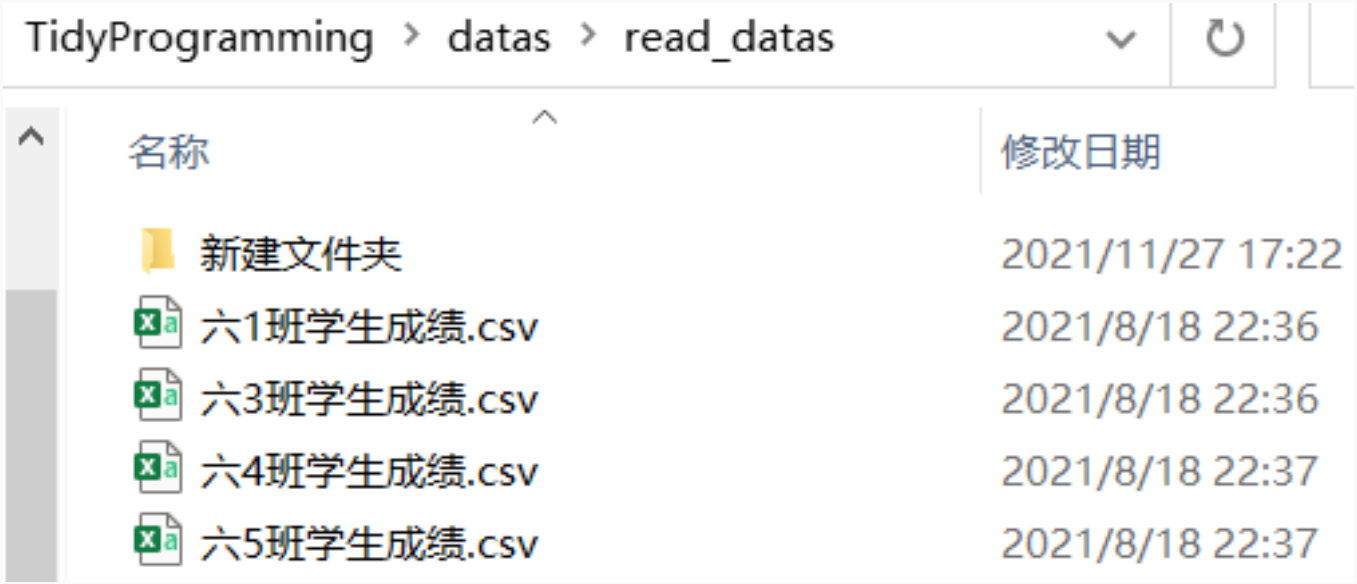


图 6: pmap 函数作用机制示意图

- 获取文件路径
- 循环迭代读取，同时合并结果

```
files = list.files("datas", pattern = "csv",
                   full.names = TRUE, recursive = TRUE)
files
#> [1] "datas/六 1 班学生成绩.csv"                "datas/六 3 班学
#> [3] "datas/六 4 班学生成绩.csv"                "datas/六 5 班学
#> [5] "datas/新建文件夹/六 2 班学生成绩.csv"
```

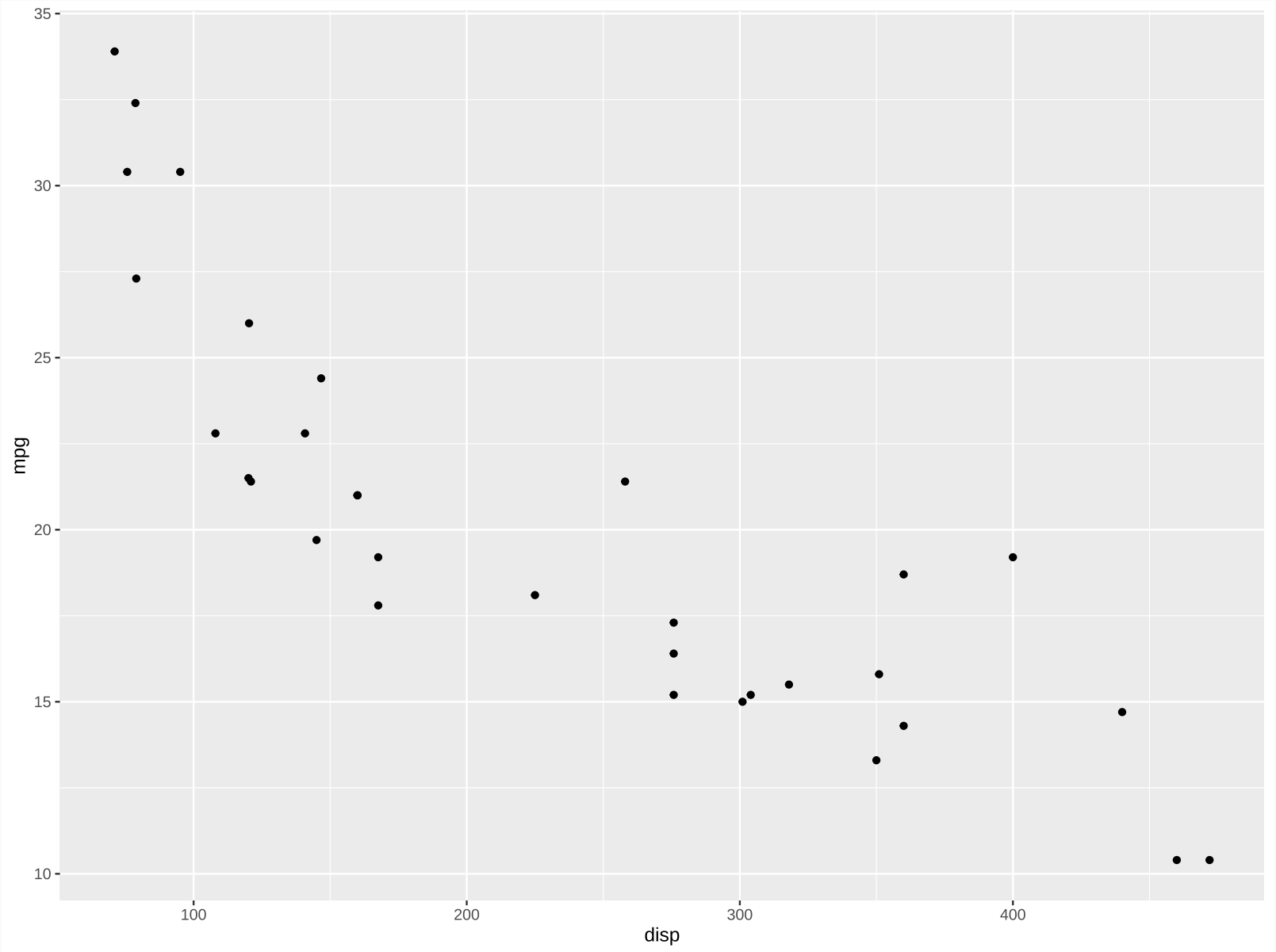
```
map_dfr(files, read_csv)
#> # A tibble: 20 x 6
#>   班级  姓名   性别   语文  数学  英语
#>   <chr> <chr>  <chr> <dbl> <dbl> <dbl>
#> 1 六 1 班 何娜   女       87    92    79
#> 2 六 1 班 黄才菊 女       95    77    75
#> 3 六 1 班 陈芳妹 女       79    87    66
#> 4 六 1 班 陈学勤 男       82    79    66
#> 5 六 3 班 江佳欣 女       80    69    75
#> # ... with 15 more rows
```

例 6 批量绘图并保存图片

以 mtcars 为例，用不同的数值列作为 x 轴，以 mpg 列作为 y 轴，批量绘制散点图，并保存为以列名命名的 png 文件。

- 需要在多个列迭代，将列名作为传递参数是个好主意，还可以用来命名 png 文件。
- 先对一个列名完成绘制散点图

```
x = "disp"
mtcars %>%
  ggplot(aes(.data[[x]], mpg)) +    # 管道中列名传参方式
  geom_point()
```


35
30
25
20
15
10
mpg
100
200
300
400
disp

- 改写为函数

```
plot_scatter = function(x) {
  mtcars %>%
    ggplot(aes(.data[[x]], mpg)) +
    geom_point()
}
```

- 批量绘图

```
cols = names(mtcars)[2:7]          # 要绘制的多个列名
ps = map(cols, plot_scatter)       # 批量绘图
```

- 批量导出到 png 文件

```
files = str_c("images/", cols, ".png")   # 准备多个文件路径
walk2(files, ps, ggsave)
```

本篇主要参阅 (张敬信, 2022), (Hadley Wickham, 2017) 以及 `purrr` 讲座、包文档，模板感谢 (黄湘云, 2021), (谢益辉, 2021).

参考文献

Hadley Wickham, G. G. (2017). *R for Data Science*. O' Reilly, 1 edition. ISBN 978-1491910399.

张敬信 (2022). *R* 语言编程：基于 *tidyverse*. 人民邮电出版社, 北京.

谢益辉 (2021). *rmarkdown: Dynamic Documents for R*.

黄湘云 (2021). *Github: R-Markdown-Template*.