`Tidyverse` 优雅编程：从向量化、泛函式到数据思维

作者介绍

张敬信，博士，副教授。现任职哈尔滨商业大学（数学与应用数学系），博士毕业于哈尔滨工业大学基础数学，喜欢编程，尤其是`R`语言编程。

1 Tidyverse 简介

`Tidyverse` [1] 包是 Hadley Wickham 及团队的集大成之作，是专为数据科学而开发的一系列包的合集，基于整洁数据，提供了一致的底层设计哲学、一致的语法、一致的数据结构。

`Tidyverse` 用 “现代的”、“优雅的” 方式，以管道式、泛函式编程技术实现了数据科学的整个流程：数据导入、数据清洗、数据操作、数据可视化、数据建模、可重现与交互报告。

`Tidyverse` 操作数据的优雅，就体现在：

- 每一步要 “做什么”，就写 “做什么”，用管道依次做下去，得到最终结果
- 代码读起来，就像是在读文字叙述一样，顺畅自然，毫无滞涩

`ggplot2` 曾经是 `R` 语言的一张名片，受到广泛的赞誉；从与时俱进的角度来说，`tidyverse` 应该成为如今 `R` 语言的一张名片！

`Tidyverse` 的核心工作流如下图所示：

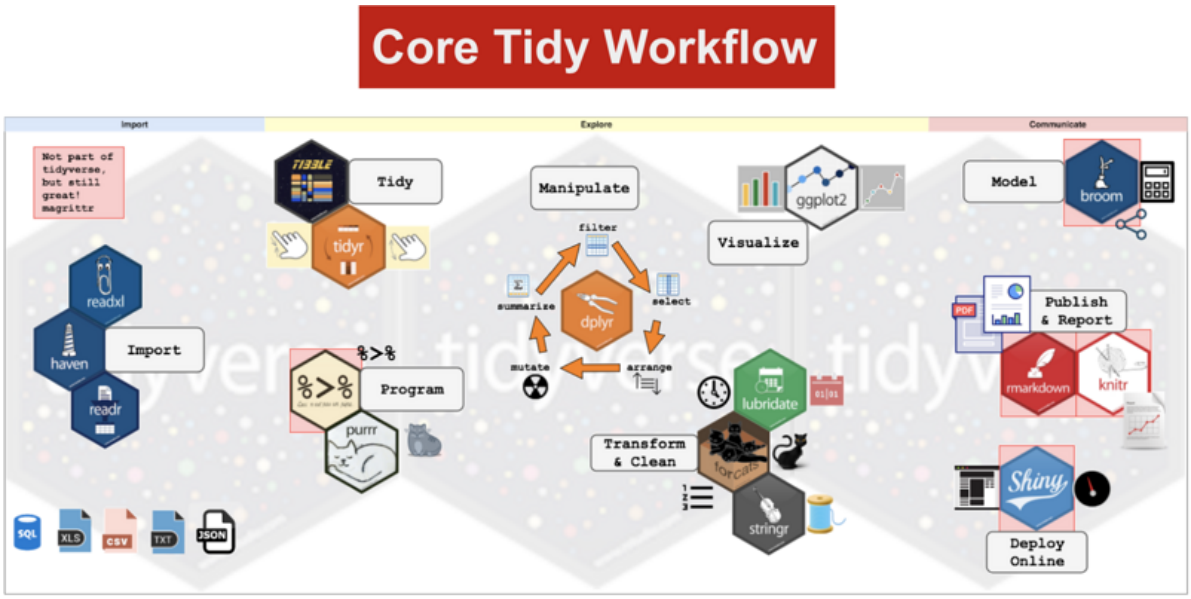

图1 Tidyverse核心工作流

从左到右分别包含了：

(1) 导入数据

主要有`readr`包（csv, txt等），`haven`包（SPSS, Stata, SAS 数据），`readxl`包（Excel 数据）。

(2) 探索数据

- 整洁模块（Tidy）：数据清洗；
- 编程技术模块（Program）：其中编程技术模块包括管道运算(`%>%`)和泛函式循环迭代(`purrr`包)；

- 数据操作模块 (Manipulate)：包括 5 大常用数据操作 `filter\select\arrange\mutate\summarise`；
- 变换清洗模块（Transform Clean）：专门用来处理特殊的数据，主要有 `lubridate` 包处理日期时间、`stringr` 包处理字符串（及正则表达式）、`forcats` 包处理因子；
- 可视化模块 (Visualize)：主要用 `ggplot2` 包来实现数据的可视化；

(3) 文档沟通

主要包括建模模块 (Model)：用 `broom` 包建模后输出整洁模型结果，之后通过 `markdown` 包、`knitr` 包还可以跟各种文档沟通，生成可重复性的研究报告，以及通过 `shiny` 包可以直接从 `R` 构建交互式 `Web` 应用程序。

所以说，这个 `tidyverse` 覆盖了整个数据科学的全部流程。我非常建议，作为一名R语言的新手，最好直接按照这个流程去学习这些新包就够了。完全可以把 `R base` 当成一个包，如果解决问题的时候，查到某个函数来自 `R base`，那就再借助帮助去用它就可以了。

关于论证 `tidyverse` 好用的这个说法，在这里我引用 `Matt Dancho` 的观点。`Matt Dancho` 是Business Science 公司创始人，数据科学专家，也是金融时间序列领域 `tidyquant, timetk, modeltime` 等包的作者，他在今年发表博文：`R` 用于研究，`Python` 用于生产（译）[2]，其中这样总结到：

> 对于做研究来说，R （tidyverse）是非凡的：做可视化，数据洞见，生成报告以及用 shiny 制作 MVP 级应用。从概念（想法）到执行（代 码），R 用户完成这些任务往往能够比 Python（Pandas）用户更快 3 到 5 倍，从而使研究工作的效率很高。”

所以，在这里我个人也是非常主张推广和使用 `tidyverse`. 可以说，我每天打开 R Studio 第一件事就是先加载 `tidyverse` 包。

2 Tidyverse 优雅编程思维

我理解的 `tidyverse` 优雅编程思维：

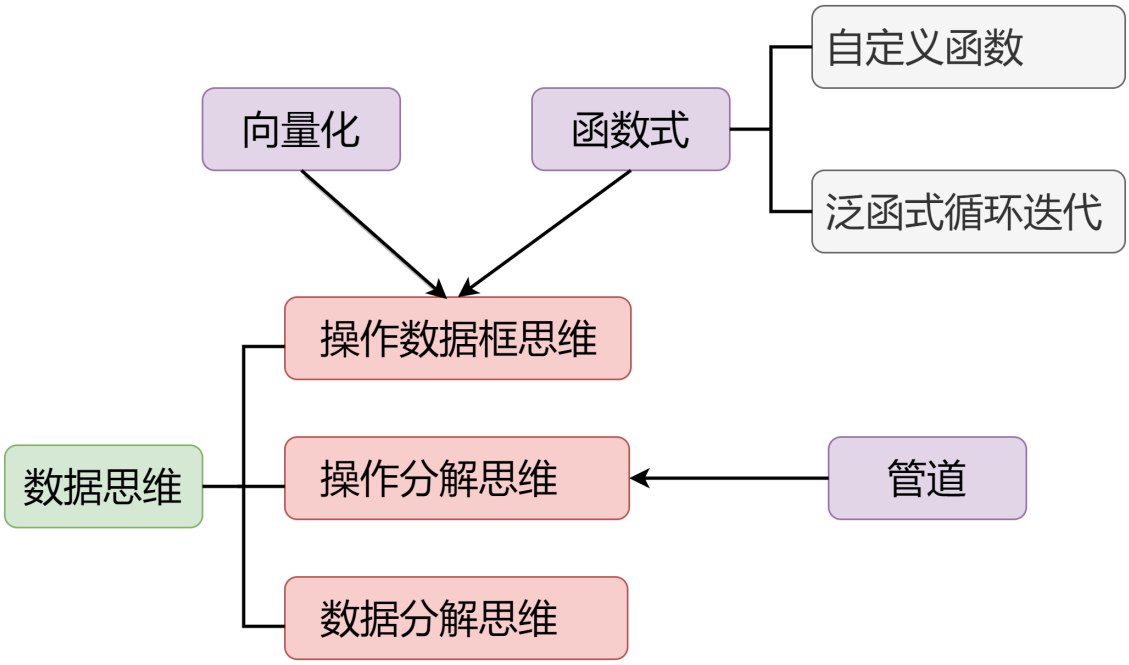

图2 Tidyverse 优雅编程思维

这个图，是围绕数据思维来展开的，这里的数据思维不是那种宏观上的空泛的数据导向的思维，而是具体的操作数据的思维：包括操作数据框思维、操作分解思维以及数据分解思维。其中一些基本的编程思维，比如向量化，还有函数式编程，它们其实根据需要是可以嵌入到操作数据框的这个思维里面去的。这里的函数式编程：又分为自定义函数，解决一个具体问题，调试通过后通常把它写成一个函数，然后

泛函式循环迭代，相当于是把函数批量地应用到一系列的元素上去。

下面，我就围绕这个图所包含的每一个部分来展开，同时会配一些例子来帮助大家理解。

2.1 数据结构

首先补充一点，关于数据结构。**数据结构，是编程的基础，是为了存储不同类型数据而设计的数据容器。**它非常重要，学习编程语言首先需要把数据结构掌握，掌握了数据结构，再去写代码，才能选择最合适的容器。选择容器合适与否？是直接能关系到你的代码是否简洁高效，甚至能否解决问题。这点可能很多人会忽略，但是数据结构是非常重要的，我在这里专门强调一下。

`R` 里面最重要的数据结构，就是数据框，它是有矩阵形式的列表：每一列是一个变量，其实就是一个向量，每一行是一个样本。作为`R`用户，需要发挥数据框操作数据的优势。避免`for`循环逐元素操作，再拼接为数据框。

许多`R`新手，就是遇到问题，他首先想到的是`for`循环逐元素操作，然后再怎么给他合并成一个数据框。我觉得这样的话，数据框这个数据结构的优势就完全没有发挥出来。为什么引入数据框？就是为了方便操作数据表。然后你不去使用它的优势，反而写那种很啰嗦、效率很低的代码。所以，我希望引导大家如何用简洁高效的方式去写代码。

2.2 向量化

向量化，就是同时操作一列，或者是一个矩阵数据，就是对每个元素同时做相同的操作。

这里面的关键，就是要**用整体的思维来思考、来表示运算。**我觉得这也是《线性代数》这门课程里边最最有用的一个思想：就是用向量、矩阵来表达一些运算。

我举的例子就是《线性代数》中的 n 元一次方程组：

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \quad\cdots\cdots \qquad\quad \cdots\cdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \end{cases}$$

假如说用逐元素法去写的话，就是这样一个很大的数学式子。如果我们引入这向量和矩阵：

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

A 是系数矩阵, x 是自变量，b 是常数项。该线性方程组，就可以简单地写成

$$Ax = b$$

这就相当于说，假如按照向量化编程去写，通过 4 个符号就可以把它表达出来。但是如果要用逐元素去写的话，就需要一个两层的`for`循环。

所以我建议大家，在思考问题的时候，优先使用向量化的思维去思考。因为现在各个编程语言基本上都支持向量化运算。所以要发挥向量运算的优势，一个是形式上非常简洁，再一个是运行效率上比写`for`循环要更高。

例1 计算样本经验熵

通过本例来认识一下常用的向量化操作。该例来自著名的西瓜书中决策树算法算例。

```
# 西瓜数据
y = c(rep("好瓜", 8), rep("坏瓜", 9))
table(y)                          # 计算各类频数，得到向量
#> y
#> 好瓜坏瓜
#> 8 9
p = table(y) / length(y)          # 向量除以标量
p
#> y
#> 好瓜坏瓜
#> 0.471 0.529
```

这个是西瓜数据，先生成向量`y`是因变量，包含 8 个好瓜， 9 个坏瓜。然后用`table()`函数计算“好瓜”和“坏瓜”的频数。结果显示有两类：“好瓜”“坏瓜”，对应频数是 8 和 9 。该结果是一个长度为 2 的一个向量。

接下来，就用这个向量继续往下计算，长度是 2 的向量`table(y)`除以标量`length(y)`，这就是向量与标量做除法，相当于是向量的每一个元素同时除以标量。算出来是各个类别的占比，也叫频率。

```
log2(p)                       # 函数作用向量
#> y
#> 好瓜坏瓜
#> -1.087 -0.918
p * log2(p)                   # 向量乘以向量，对应元素做乘法
#> y
#> 好瓜坏瓜
#> -0.512 -0.486

sum(p * log2(p))              # 向量求和
#> [1] 0.998
```

继续对该频率向量计算以 2 为底的对数，即函数直接作用在向量上。`R`里边几乎所有函数都支持向量化操作，所以你直接把一个向量传给函数就可以了，不需要去写一个`for`循环逐元素计算它的对数。

接着，`p*log2(p)`演示的是向量和向量做乘法。这也是最常用的向量化运算，指的是对应元素分别做乘法。

最后再求和，还是函数直接作用向量。

总之，向量化操作最常用的就这 3 种：向量与标量、向量与向量（对应元素分别做运算）、函数直接作用在向量上。

2.3 自定义函数

自定义函数，是在解决一个具体问题的时候。我一般建议，首先拿一个简单的实例。如果没有的话，就自己去设计一个简单的实例；然后去把它逐步地调试通过；再改写（封装）成一个函数。

自定义函数，在编程里面实际上就是做一件事情。你要把它写成函数的话，有非常大的好处，就是可以一步到位，而且可以重复的用，可以批量的用，还可以给别人用。所以把要解决的问题，写成一个函数，这个可以说是编程中非常普遍的一个操作。

那么编程中函数的一般形式，这里不是只指`R`语言，所有编程语言通用的一个写法：

```
(返回值1,..., 返回值m) = 函数名(输入1, ..., 输入n)
```

输入是根据你问题的需要决定的，如果没有这个“输入”，函数体就没有办法继续往前计算。输出返回值是你自己设计的，你想要返回什么就设计为返回值。中间的处理过程就把它封装在函数体里面。所以自定义函数，就相当于创造了一个模具，调用函数就好比是用模具批量的生产产品。

例2 向量归一化

- 设计简单实例 $x = [15, 8, 26, \mathrm{NA}, 12]$:

```
x = c(15, 8, 26, NA, 12)
type = "pos"                              # 标记正向指标
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])
}
#> [1] 0.389 0.000 1.000 NA 0.222
```

做向量归一化的时候，需要考虑到指标的方向，可能是正向或负向，需要区分。这里设计了一个简单的实例，比如`x`等于，随便找了几个数，这里边还设计了一个缺失值`NA`，这样是为了适用范围更广一些。

就先拿这样一个具体的小实例`x`，然后通过一个`type`变量来标记该向量是正向还是负向。之后就是做归一化的计算，如果是正向指标，就是用前面公式，如果是负向指标，就用后面的公式。用到了`if-else`分支结构，之后就返回了向量归一化后的结果。其中，缺失值通过`na.rm=TRUE`做了一个处理，这样在计算的时候会忽略缺失值。

这个自定义函数的过程就相当于你在解决一个具体问题的过程，自己先设计一个小实例，然后通过代码调试出结果，调试出结果后再把它改写（封装）成函数。

```
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])
  }
}

Rescale(x)
#> [1] 0.389 0.000 1.000 NA 0.222
Rescale(x, "neg")
#> [1] 0.611 1.000 0.000 NA 0.778
```

改造成函数其实非常简单，函数体它只需要形参，不需要实参。`R`里边默认最后一条语句的返回值就是输出。这样，只需要开始的赋值变成函数的输入，其余部分原样作为函数体，函数就定义好了。

接着进行一个简单的测试，比如还用刚才的`x`，第2个参数缺少，就使用默认参数，即`type = "pos"`，表示是正向指标，然后计算出它的归一化；如果是负向指标，就让`type`取`“neg”`，这样就当负项指标去做归一化。

2.4 泛函式循环迭代

泛函式是我的一种叫法，因为我是数学老师，也教《泛函分析》，那么在数学上，函数的函数称为泛函，**在编程中表示函数作用在函数上，或者说函数包含其它函数作为参数，这就是泛函式。**

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。

表示出来就是泛函式：`map_*(x, f)`

`purrr` 包提供的 `map` 系列函数就专门做循环迭代的，就是把函数 `f` 依次应用到 `x` 的每个元素上，所以一般情况不需要写`for`循环，你只要用这个思路做一步就可以了。另外这里的序列，也是我的一个叫法，它是由一系列可以根据位置索引的元素构成。

这里的元素可以很复杂和不同类型；向量、列表、数据框都是序列；而将 `x` 作为第一个参数，是为了便于使用管道。

`purrr` 泛函式循环迭代解决问题的通用流程：

- 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[1]打包到一起返回；
- 可以根据想让结果返回什么类型选用 `map` 后缀：
    - `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

在这里有个技巧，就是 `purrr` 风格公式（匿名函数），这里的参数 `.f` 提供了一种简写，就是你只需要写清楚它是如何操作序列 `.x` 就可以。比如，

- 一元函数 $f(x)=x^2+1$，就专用 `.x` 代表这个一元函数的自变量，对应地写为 `.f = ~.x ^ 2 + 1`
- 二元函数 $f(x,y)=x^2-3y$，就专用 `.x, .y` 分别作为二元函数的自变量，对应地写为 `.f = .x ^ 2 - 3 * .y`
- 还有三元函数自变量 `..1, ..2, ..3`，所有自变量 `...`

注意，`.x` 是序列中的一个（代表）元素。这里面包含了分解的思维，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。

下面看几个常用的 `map` 函数作用机制示意图。

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

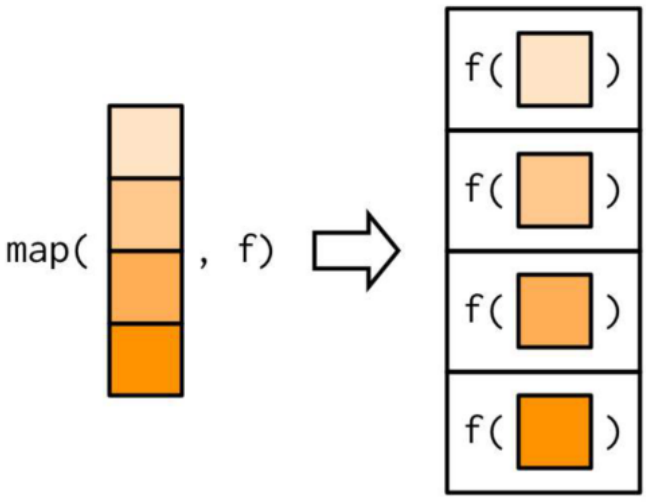

图3 map函数作用机制示意图

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

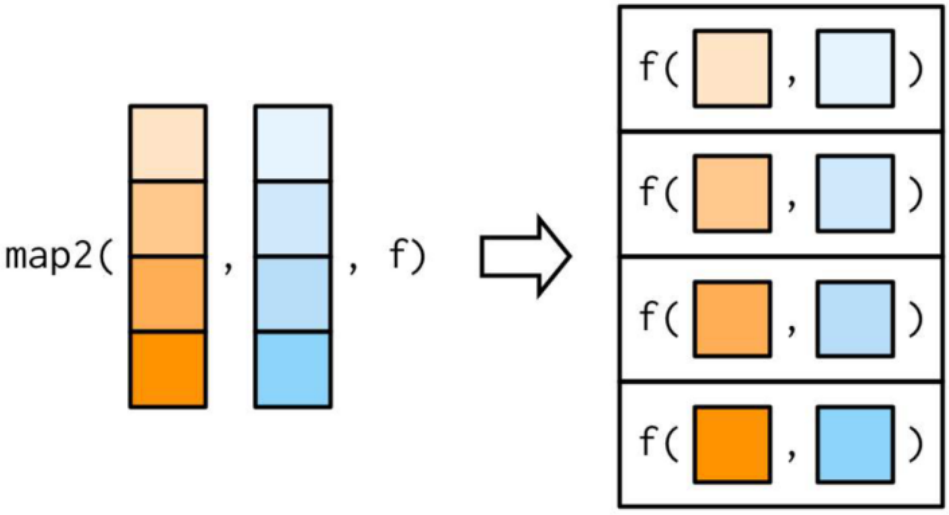

图4 map2函数作用机制示意图

`pmap_*(.l, .f, ...)`，表示依次应用多元函数 `.f` 到多个序列 `.l` 的每层元素，可实现对数据框逐行迭代，因为这多个序列的长度一样，所以实际上就相当于说是依次应用到数据框的每一行上，这样就更好理解，`...` 可设置 `.f` 的其它参数：

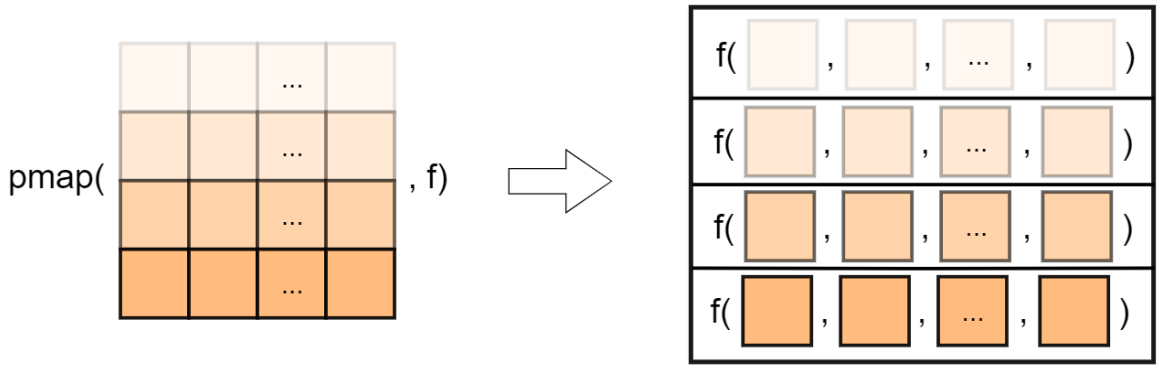

图5 pmap函数作用机制示意图

另外，还有

- `walk_*` 系列，只循环迭代做事情但不返回结果，比如批量保存数据/图形到文件；
- 元素与索引一起迭代有 `imap_*()` 系列。具体用法可查看帮助文档。

例3 purrr 循环迭代

- 对数据框逐列迭代

数据框是序列（列表），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"
```

所以，对一个数据框做上述 `map` 映射，就相当于把函数 `mean` 依次应用到它的每一列上。这里数据框 `df` 是鸢尾花数据的前 4 列数值列，因为求均值返回值是数值，所以 `map` 后缀用 `dbl`。如果你想返回值是字符向量，修改 `map` 后缀为 `chr 即可。

- 对数据框各列做归一化, 若均为正向指标：

```
map_dfc(df, Rescale)
# 同 map_dfc(df, Rescale, type = "pos")
# 同 map_dfc(df, ~ Rescale(.x, "pos"))
#> # 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
```

将前文定义过的归一化函数，依次应用到数据框 `df` 的每一列就可以了。

`Rescale()` 函数默认是正向指标，如果想写其它参数，有两种写法：(1) 作为 `map_*()` 的 `...` 参数（传递 `.f` 的其它参数）； （2） `purrr` 公式写法，将 `.f` 写完整。

如果想对各列按不同正负方向，比如分别为正向，负向，负向，正向，做归一化。这相当于是同时在两个序列上迭代，第 1 个序列数据框（ 1 至 4 列），第 2 个序列是指定它们正向、负向的字符向量，这就适合用 `map2` 来写：

```
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
```

- `pmap` 对数据框逐行迭代

比如，计算数据框每一行的均值，为了节省页面选的是前 10 行

```
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
```

再比如，分别按多组参数批量生成正态随机数

```
tribble(~n, ~mean, ~ sd,      # 按行创建数据框
  2, 5, 1,                    # 生成2个N(5,1)随机数
  3, 10, 2) %>%               # 生成3个N(10,2)随机数
pmap(~ rnorm(...))
#> [[1]]
#> [1] 6.45 4.37
#>
#> [[2]]
#> [1] 6.36 8.77 6.65
```

更多其它案例，可以参阅：批量读写数据文件 [3], 批量绘图并保存图片。

2.5 管道

`magrittr` 包引入了管道操作 `%>%` （现在 `R 4.1` 也开始支持管道 `|>`），能够通过管道将数据从一个函数传给另一个函数，从而用若干函数构成的管道依次变换你的数据：

```
x %>% f() %>% g()
```

表示，先对 `x` 做函数 `f` 的操作，对其结果，再进行函数 `g` 的操作。

管道就相当于说，数据自动从一个函数传给另一个函数，你只需要写你每一步想对数据施加的操作就行。

管道的好处，是避免使用多余的中间变量（节省内存），再一个让程序可读性大大加强。因为它这个顺序就跟我们操作数据的逻辑是一致的，先做一个操作，再做一个操作，再做下一个操作。

注意，数据默认传递给下一个函数的第 1 个参数，否则需用 `.` 代替数据。

2.6 数据思维 I：操作数据框的思维

前面都是基本编程中的技术，接下来要把它们应用到数据思维当中。

操作数据框的思维，实际上就是将向量化和函数式（自定义函数+ 泛函式循环迭代）编程思维，纳入到数据框中来：

- 原来向量化编程同时操作一个向量的数据，变成在数据框中操作一列的数据，或者同时操作数据框的多列，甚至分别操作数据框每个分组的多列；
- 原来函数式编程变成将想做的操作自定义成函数（或现成函数），再依次应用到数据框的多个列上，以修改列或做汇总。

这里的数据框是指 `tibble`，支持列表列（嵌套数据框）；记住：**每次至少操作一列数据！**

例4 操作数据框

- 操作一列（计算一个新列）

```
df = as_tibble(iris) %>%                    # 准备数据
  set_names(str_c("x",1:4), "Species")      # 缩短列名

df %>%
  mutate(x1 = x1 * 10)
#> # A tibble: 150 x 5
#> x1 x2 x3 x4 Species
#> <dbl> <dbl> <dbl> <dbl> <fct>
#> 1 51 3.5 1.4 0.2 setosa
#> 2 49 3 1.4 0.2 setosa
#> 3 47 3.2 1.3 0.2 setosa
```

```
#> 4 46 3.1 1.5 0.2 setosa
#> 5 50 3.6 1.4 0.2 setosa
#> # ... with 145 more rows
```

`x1` 是数据框的 1 列，1 个向量，向量化计算乘以 10，赋给同列名，效果是替换旧列。

```
df %>%
  mutate(avg = pmap_dbl(.[1:4], ~ mean(c(...))))
#> # A tibble: 150 x 6
#> x1 x2 x3 x4 Species avg
#> <dbl> <dbl> <dbl> <dbl> <fct> <dbl>
#> 1 5.1 3.5 1.4 0.2 setosa 2.55
#> 2 4.9 3 1.4 0.2 setosa 2.38
#> 3 4.7 3.2 1.3 0.2 setosa 2.35
#> 4 4.6 3.1 1.5 0.2 setosa 2.35
#> 5 5 3.6 1.4 0.2 setosa 2.55
#> # ... with 145 more rows
```

前文用 `pmap` 计算过数据框的行平均，得到向量，现在赋值给一个新列 `avg`。

这就是向量化思维，用到了数据框里的思维。

- 同时操作多列

```
df %>%
  mutate(across(1:4, ~ .x * 10))
#> # A tibble: 150 x 5
#> x1 x2 x3 x4 Species
#> <dbl> <dbl> <dbl> <dbl> <fct>
#> 1 51 35 14 2 setosa
#> 2 49 30 14 2 setosa
#> 3 47 32 13 2 setosa
#> 4 46 31 15 2 setosa
#> 5 50 36 14 2 setosa
#> # ... with 145 more rows
```

这里就需要借助这个 `across` 函数（后面单独再讲）。`across` 函数先选择你要操作的列，之后再写你要对每一列应用的函数就可以了，结果是将数据框 `df` 的 1 至 4 列，同时乘以 10 .

再比如，将前文定义的归一化函数，应用到数据框的 1 至 4 列：

```
df %>%
  mutate(across(1:4, Rescale))
#> # A tibble: 150 x 5
#> x1 x2 x3 x4 Species
#> <dbl> <dbl> <dbl> <dbl> <fct>
#> 1 0.222 0.625 0.0678 0.0417 setosa
#> 2 0.167 0.417 0.0678 0.0417 setosa
#> 3 0.111 0.5 0.0508 0.0417 setosa
#> 4 0.0833 0.458 0.0847 0.0417 setosa
#> 5 0.194 0.667 0.0678 0.0417 setosa
#> # ... with 145 more rows
```

这就是，将函数式思维，用到了数据框里的思维。

2.7 数据思维II：操作分解的思维

将复杂操作分解为若干简单操作。复杂数据操作，都可以分解为若干简单的基本数据操作：

- 数据连接
- 数据重塑（长宽转换、拆分/合并列）
- 筛选行
- 排序行
- 选择列
- 修改列
- 分组汇总

一旦完成问题的梳理和分解，又熟悉每个基本数据操作，用“管道” 流依次对数据做操作即可。

例5 管道分解操作

```
load("datas/SQL50datas.rda")
head(score, 2)
#> # A tibble: 2 x 3
#> 学号课程编号成绩
#> <chr> <chr> <dbl>
#> 1 01 01 80
#> 2 01 02 90
head(student, 2)
#> # A tibble: 2 x 4
#> 学号姓名生日性别
#> <chr> <chr> <chr> <chr>
#> 1 01 赵雷1990-01-01 男
#> 2 02 钱电1990-12-21 男
```

问题：查询平均成绩 `≥60` 分的学生学号、姓名和平均成绩。
分解问题：

- 先按学号分组汇总计算平均成绩
- 然后根据条件筛选行
- 再根据学号连接学生信息
- 最后选择想要的列

```
score %>%
  group_by(学号) %>%
  summarise(平均成绩= mean(成绩)) %>%
  filter(平均成绩>= 60) %>%
  left_join(student, by = "学号") %>%
  select(学号, 姓名, 平均成绩)
#> # A tibble: 5 x 3
#> 学号姓名平均成绩
#> <chr> <chr> <dbl>
#> 1 01 赵雷89.7
#> 2 02 钱电70
#> 3 03 孙风80
#> 4 05 周梅81.5
#> 5 07 郑竹93.5
```

2.8 数据思维III: 数据分解的思维

分组修改：想对数据框进行分组，分别对每组数据做操作，整体来想这是不容易想透的复杂事情，实际上只需做 `group_by()` 分组，然后把你要对一组数据做的操作实现

- `group_by + summarise`：分组汇总，结果是"有几个分组就有几个样本"
- `group_by + mutate`：分组修改，结果是"原来几个样本还是几个样本"

同时操作多列：`across()` 同时操作多列，实际上你只需把对一列要做的操作实现

这些就是数据分解的操作思维，这些函数会帮你**分解+ 分别操作+ 合并结果**，你只需要关心**分别操作**的部分，它就是一件简单的事情。

例6 分组修改数据

```
load("datas/stocks.rda")
stocks
#> # A tibble: 753 x 3
#> Date Stock Close
#> <date> <chr> <dbl>
#> 1 2017-01-03 Google 786.
#> 2 2017-01-03 Amazon 754.
#> 3 2017-01-03 Apple 116.
#> 4 2017-01-04 Google 787.
#> 5 2017-01-04 Amazon 757.
#> # ... with 748 more rows
```

问题：分别计算每支股票的收盘价与前一天的差价。

该股票的数据，相当于是面板数据，包含三支股票，在这里想算每支股票的收盘价与前一天的差价。大家普遍能想到的方法是，根据股票把数据框分割成 3 个子数据框，然后循环迭代分别计算，再把结果合并回来。

但是用数据分解的思维，你只需要对 `Stock` 分组，之后再把对一组（一支股票的）数据，怎么计算收盘价与前一天的差价的代码，写出来即可：

```
stocks %>%
  group_by(Stock) %>%
  mutate(delta = Close - lag(Close))
#> # A tibble: 753 x 4
#> # Groups: Stock [3]
#> Date Stock Close delta
#> <date> <chr> <dbl> <dbl>
#> 1 2017-01-03 Google 786. NA
#> 2 2017-01-03 Amazon 754. NA
#> 3 2017-01-03 Apple 116. NA
#> 4 2017-01-04 Google 787. 0.760
#> 5 2017-01-04 Amazon 757. 3.51
#> # ... with 748 more rows
```

- `across()`函数

注：f将长度为n的向量，映射为长度为n的向量

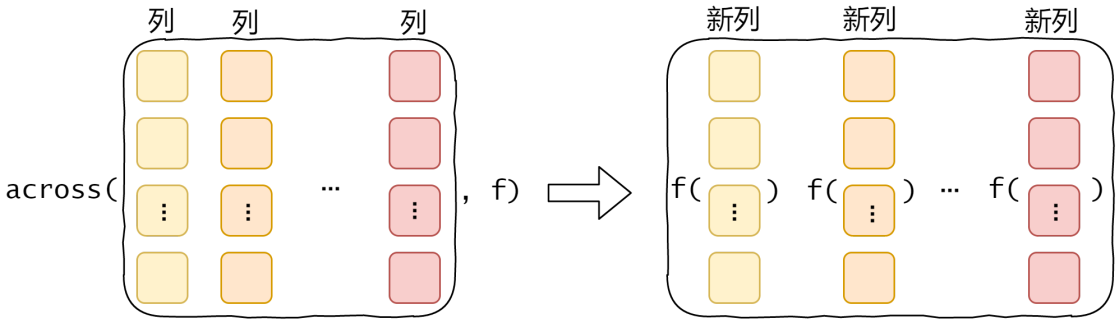

图6 across 函数作用机制示意图

`across` 函数专门用来同时操作多列，其操作逻辑就是：先把要同时操作的多列选出来，支持各种 `dplyr` 选择列语法，之后是要对每一列应用的函数 `f`，你只需要把对一列做的操作写清楚就可以了，剩下的就交给 `across` 函数，它会帮你一次把 `f` 应用到选择的各列上，之后再替换[2]为新列（你选择了几列，那就会对应生成几个新列）。所以，这个 `f` 的输入参数是一个向量，它的返回值也是一个长度一样的向量。

再回头看一下，刚才的例子：

```
df %>%
  mutate(across(1:4, Rescale))
#> # A tibble: 150 x 5
#> x1 x2 x3 x4 Species
#> <dbl> <dbl> <dbl> <dbl> <fct>
#> 1 0.222 0.625 0.0678 0.0417 setosa
#> 2 0.167 0.417 0.0678 0.0417 setosa
#> 3 0.111 0.5 0.0508 0.0417 setosa
#> 4 0.0833 0.458 0.0847 0.0417 setosa
#> 5 0.194 0.667 0.0678 0.0417 setosa
#> # ... with 145 more rows
```

我想对数据框的 1 至 4 列做归一化，需要先自定义归一化函数 `Rescale`，它能对一个向量做归一化得到新的向量，然后借助 `across` 选择想要操作的列，并把 `Rescale` 应用上去就行了。

我把这些叫做是数据分解的思维，本来是复杂的问题：想对一系列对象做一件事情（循环迭代）、想分别对每组数据做操作、想同时对多列做归一化，有了`map_*`，`group_by` 和 `across`，真正需要你写代码解决的问题变成：对一个对象怎么做这件事、对一组数据怎么做操作、对一列数据怎么做归一化。

通过 `tidyverse` 里面的这些方便的函数，能把复杂的问题给你分解成简单的问题，让你更容易解决。

另外，还有与 `across` 函数同样或类似的逻辑函数：

- `if_any()`, `if_all()`: 是配合 `filter()` 根据多列的值构造条件，筛选行
- `slider::slide_*()`: 滑窗迭代，构造想要的滑动窗口，先保证滑动窗口正确，然后你只需要思考对每个窗口数据做什么操作，写成函数

最后，总结一下贯穿始终的分解思维：

- 解决无从下手的复杂问题：分解为若干可上手的简单问题
- 循环迭代：分解为把解决一个元素的过程写成函数，再 map 到一系列的元素
- 复杂的数据操作：分解为若干简单数据操作，再用管道连接
- 操作多组数据：分解为 `group_by` 分组+ 操作明白一组数据
- 修改多列：分解为 `across` 选择列+ 操作明白一列数据

以上内容，是我的 R 语言新书 [3] 第1、2 章所涉及编程思维的脉络梳理，感谢黄湘云 [4] 在 Github 提供的R markdown [5] 模板。

参考文献

[1] Hadley Wickham, R. (2021). tidyverse: R packages for data science

[2] Matt Dancho, J. C.. R is for Research, Python is for Production.
[3] 张敬信. R 语言编程：基于 tidyverse. 人民邮电出版社, 2022.
[4] 黄湘云. Github: R-Markdown-Template. 2021.
[5] 谢益辉. rmarkdown: Dynamic Documents for R. 2021.

答疑环节Q_A:

Q: 如何在分组之后生成一个排序变量?

A: 这就是分组修改，先用 `group_by` 分组，然后再用 `mutate` 计算新列：`mutate(r = 1:n())`，其中，`n()` 返回当前数据框样本数，这样相当于分别对每个分组，构建新列，其内容是向量 `1:行数`。

Q: `tibble` 和 `data frame` 的区别是什么?

A: `tibble` 可以说是更现代的数据框，它与 `data frame` 基本上是一致的，可以用 `tibble` 代替原生的 `data.frame`，它比 `data.frame` 更多一些小的规范性，比如说它的输出方式，`tibble` 会自适应窗口，只显示 10 行和部分列，其他显示不下的列概述在后面，每列均带有其类型缩写的标签；再一个就是 `tibble`不会做强制转换，你对它选择一列，它还是一个数据框，它默认不会做化简。但 `data.frame`，你如果选择它的一列，它默认会简化成一个向量；还有`tibble`支持列表列，可以用嵌套数据框；`tibble` 取消了支持行名。`data.frame` 和 `tibble` 就是在一些细节上稍微有点差别，但是整体上它们是一样的。在处理数据的逻辑上是一样的。补充一点，如果是与 `tidyverse` 配合使用，强烈建议用 `tibble`.

最后，顺便推广一下我的 R 语言新书：

> 掌握 tidyverse 优雅编程，用最 tidy 的方式学习R 语言!
> 《R 语言编程：基于tidyverse》，张敬信，人民邮电出版社
> 预计2022 年夏上市，敬请期待!
> 我的知乎专栏:
> https://www.zhihu.com/people/huc_zhangjingxin/columns
> Email: zhjx_19@hrbcu.edu.cn

更多关于本书的学习讨论，可以关注我的知乎专栏：

https://www.zhihu.com/people/huc_zhangjingxin/columns

欢迎加入QQ 读者群: 875664831 (群名称：tidy-R语言)

群名称:tidy-R语言
群　号:875664831　读者群

---

1. 每一个元素作用后返回一个结果. ↩
2. 设置列名参数，若要保留旧列，可设置 `.names` 参数. ↩