R 语言编程 — 基于 tidyverse

作者：张敬信
组织：哈尔滨商业大学
时间：2021-10-11
版本：*0.95*

用最 tidy 的方式学习 R 语言！

目录

插图目录

表格 目录

作者序

R 语言是专业的统计编程语言，具有顶尖水准的绘图功能，且开源免费有着丰富的扩展包和活跃的社区。R 语言这些优质的特性，使得它始终在数据统计分析领域的 SAS、Stata、SPSS、Python、Matlab 等同类软件中占据领先地位。

R 语言曾经最为人们津津乐道的是 Hadley 大神开发的 ggplot2 包，泛函式图层化语法赋予了绘图一种 ‘‘优雅” 美。近年来，R 语言在国外蓬勃发展，ggplot2 这个 ‘‘点’’ 在 2016 年以来，已被 Hadley 大神 ‘‘连成线、张成面、形成体（系）’’，这就是 tidyverse 包，集

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

整个数据科学流程于一身，而且是以 ‘‘现代的’’、‘‘优雅的” 方式，以管道式、泛函式编程技术实现。不夸张地说，tidyverse 操作数据比 pandas 更加好用、易用！再加上可视化本来就是 R 所擅长，可以说 R 在数据科学领域好于 Python。

这种整洁、优雅的 tidy-流，又带动了 R 语言在很多研究领域涌现出了一系列 tidy-风格的包：tidymodels（统计与机器学习）、mlr3verse（机器学习）、rstatix（应用统计）、tidybayes（贝叶斯模型）、tidyquant（金融）、fpp3（时间序列）、tidytext（文本挖掘）、tidygraph（网络图）、sf（空间数据分析）、tidybulk（生信）、sparklyr（大数据）等。

其中机器学习/数据挖掘领域，曾经的 R 靠单打独斗的包，如今也正在从整合技术上迎头赶上 Python，出现了 tidy-风格的 tidymodels 包，以及真正最新理念、最新技术、最新一代的机器学习 mlr3verse 包，它比 sklearn 还先进，基于 R6 类面向对象，data.table 神速数据底层，开创性的 Graph-流模式（图/网络流，区别于通常的线性流）。

写作本书的目的：

然而，我发现这些近几年出现的 R 语言新技术，在国内很少有人问津，绝大多数 R 语言的教师、教材、博客文章、R 学习者仍在沿用那些过时的、晦涩的 R 语法，对 R 语言的印象停留在 5 年前：语法晦涩难懂、速度慢，做统计分析和绘图还行，机器学习只有单独算法的包，做不了深度学习、大数据、工业部署.....

有感于此，我想写一本用最新 R 技术，方便新手真正快速入门 R 语言编程的书，来为 R 语言正名，以在国内推广已如此优秀好用的 R 语言。我是一名大学数学教师，热爱编程、热爱 R 语言，奉行终生学习理念，一直喜欢跟踪和学习新知识、新技能。我对编程和 R 语言有一些独到的理解体会，因为我觉得数学语言与编程语言是相通的，都是用语法元素来表达和解决问题，我想把这些理解体会用符合国人的语言习惯表达出来。

希望我这本书，如果有幸进入了您的法眼，能让您学到正确的编程思想，学到最新的 R 语言编程知识和编程思维，能真正让您完成 R 语言入门或 R 知识汰旧换新。

本书主要适合以下读者：

- 没有 R 语言基础，想要系统地学习 R 语言编程，特别是想要用最新 R 技术入门 R
- 具备一定的 R 语言基础，想升级 R 语言编程技术到最新
- 想要理解编程思想，锻炼向量化、函数式编程思维，以及真正的数据思维
- 想要以 R 作为工具，将来从事统计分析、数据挖掘、机器学习，特别是想使用最新机器学习包：`tidymodels`, `mlr3verse`
- 高校学习 R 语言及相关课程的学生、教师、科研人员，特别是将来想要在时间序列、金融、空间数据分析等领域，使用最新包 `fpp3`, `tidyquant`, `sf` 等

本书特色

1. 新

本书绝大部分内容都是参阅最新版本 R 包的相关文档，很少参阅书籍（而且尽量参阅最新的在线版本）。本书全面采用最新的 R 语言技术编写，特别是 `tidyverse`“整洁流、管道流、泛函流”数据科学。

2. 真正融入编程思维

很多国内 R 语言编程书只是罗列堆砌编程语法，国外有不少优秀的 R 语言编程书，但翻译版往往就只是“直译”，只把表面意思用生硬的汉语表达出来，很难让初学者学透它们。解决办法就是真正融入编程思维：编程思想引导，编程语法到底是怎么回事，应该用于何处、怎么使用。本书前言和第一章融入**向量化编程**与**函数式编程思维**；第二章主要融入**数据思维**。

我写东西的特点就是，每个知识点都搜集很多相关最新资料，自己先学得透彻明白，再把自己的理解用最通俗易懂语言表达出来。看过我知乎专栏文章的人，应当对此有所体会。

3. 精心准备实例

编程语法讲透彻还不够，必须配以合适的实例来演示，所以也请读者一定要将编程语法讲解与配套实例结合起来阅读，比起实例代码调试通过，更重要的是借助实例代码理解透彻该编程语法，所包含的编程思维。本书后半部分是 R 语言在应用统计、探索性数据分析、文档沟通方面的应用，所配案例力求能让读者上手使用。

4. 程序代码优雅、简洁、高效

本书程序代码都是基于最新的 `tidyverse`，自然就很优雅；简洁高效是能用向量化编程就不用逐元素，能用泛函式编程，就不用 `for` 循环。可以说，读者如果用我这本书入门 R 语言，或者更新您的 R 知识，就会自动跳过写低级啰嗦代码的阶段，直接进入写让别人羡慕的“高手级”代码的行列。

本书内容安排

本书的结构是围绕如何学习 R 语言编程来展开的，全书共分为 6 章。冯国双老师在《白话统计》序言中写道：

> 一本书如果没有作者自己的观点，而只是知识的堆叠，那么这类书是没有太大价值的。

尤其在当前网络发达的时代，几乎任何概念和知识点都可以从网络上查到。但有一点您很难查到，对于编程书来说，那就是编程思维。本书最大的特点之一就是无论是讲编程思想还是讲编程语法知识点，都把编程思维融入进去。

很多人学编程始终难以真正入门：自己写代码；学习编程语言在其编程思想的指导下才能事半功倍。本书的前言就先来谈编程思维，包括如何理解编程语言，用数学建模的思维引领读者跨越如何从实际问题到自己写代码解决问题，以及 R 语言的编程思想：面向函数、面向对象、面向向量。

第一章是讲述 R 语言编程的基本语法，同时渗透向量化编程、函数式编程思维。这些语法在其他编程语言中也是相通的，包括搭建 R 语言环境，常用数据结构（存放数据的容器）：向量、矩阵、数据框、因子、字符串（及正则表达式）、日期时间，分支结构，循环结构，自定义函数。这些基本语法是您写 R 代码的基本元素，学透它们非常重要，只有学透它们才能将其任意组合、恰当使用，以写出各种各样的解决具体问题的 R 代码。同样是讲 R 基本语法，本书不同之处在于，用 `tidyverse` 中更一致、更好用的相应包加以代替：用 `tibble` 代替 `data.frame`、用 `forcats` 包处理因子，用 `stringr` 讲字符串（及正则表达式）、用 `lubridate` 包讲日期时间、循环结构中用 `purrr` 包的 `map_*` 函数代替 `apply` 系列函数，其中特别讲到编程技术：泛函式编程。

第二章正式进入 `tidyverse` 核心部分：数据操作，侧重讲解数据思维。先简单介绍 `tidyverse` 包以及编程技术之管道操作，接着围绕各种常用数据操作展开，包括数据读写（各种常见数据文件的读写及批量读写、R 连接数据库、中文编码问题及解决办法），数据连接（数据按行/列拼接、SQL 数据库连接），数据重塑（‘‘脏’’ 数据变 ‘‘整洁’’ 数据，长宽表转换、拆分与合并列），数据操作（选择列、筛选行、对行排序、修改列、分组汇总）、其它数据操作（按行汇总、窗口函数、滑窗迭代、整洁计算），以及 `data.table` 基本使用（常用数据操作的 `dplyr` 语法与 `data.table` 语法对照）。`Tidyverse` 最大的优势就是以 ‘‘管道流’’、‘‘整洁语法’’ 操作数据，这些语法真正让数据操作从 `R base` 的晦涩难记难用，到 `tidyverse` 的 ‘‘一致’’‘‘整洁’’ 好记好用，比 `Python` 的 `pandas` 还好用！关键是用一次就能记住！为了最大限度地降低理解负担，本书特意选用中文的学生成绩数据作为演示数据，让读者只关心语法就好。另外，`tidyverse` 的这些数据操作，实际上已经在语法层面涵盖了日常 `Excel` 数据操作、SQL 数据库操作，活用 `tidyverse` 上述数据操作语法已经可以胜任这些工作。

第三章是可视化与建模技术。可视化只介绍最流行的可视化包 `ggplot2`，先从 `ggplot2` 的图层化绘图语法开始，依次介绍 `ggplot2` 的九大部件：数据、映射、几何对象、标度、统计变换、坐标系、分面、主题、输出；接着介绍图形从功能上的分类：类别比较图、数据关系图、数据分布图、时间序列图、局部整体图、地理空间图，对每一类图形分别选择其中代表性的用实例加以演示。建模技术包括三个内容：(1) 用 `broom` 包提取统计模型结果为整洁数据框，方便后续访问和使用；(2) `modelr` 包中一些有用的辅助建模函数；(3) 批量建模技术，比如要对全世界 170 多个国家的数据分别建立模型、提取模型结果，当然这可以用 `for` 循环实现，但这里采用更加优雅的 `map_*` 实现，以及 ‘‘行化迭代’’ 实现。

第四章，应用统计。R 语言是专业的统计分析软件，广泛应用于统计分析与计算。本章将从四个方面展开：(1) 描述性统计，介绍适合描述不同数据的统计量、统计图、列联表；(2) 参数估计，主要介绍点估计与区间估计，包括 `Bootstrap` 法估计置信区间，以及常用的参数估计方法：最小二乘估计、最大似然估计；(3) 假设检验，将介绍假设检验原理，基于理论的假设检验：以方差分析、卡方检验为例，并用整洁的 `rstatix` 包实现，以及基于重排的假设检验：以 t 检验为例，用最新的 `infer` 包实现；

(4) 回归分析，从线性回归原理、回归诊断，借助具体实例讲解多元线性回归的整个过程，并介绍广泛应用于机器学习的梯度下降法，以及广义线性模型原理。

第五章，探索性数据分析。主要讨论三方面内容：(1) 数据清洗，包括缺失值探索与处理、异常值识别与处理；(2) 特征工程，包括特征缩放（标准化/归一化/行规范化/数据平滑）、特征变换（非线性特征/正态性变换/连续变量离散化）、基于 PCA 的特征降维；(3) 探索变量间的关系，包括分类变量之间、分类变量与连续变量、连续变量之间的关系。

第六章，文档沟通，将讨论如何进行可重复研究，用 R markdown 家族生成各种文档，介绍 R markdown 的基本使用，R 与 Latex 交互编写期刊论文/幻灯片/书籍、R 与 Git/Github 交互进行版本控制、用 R Shiny 轻松制作交互网络应用程序（Web App）以及开发和发布 R 包的最新工作流程。

附录部分是正文内容的补充和扩展，将分别介绍 R6 类面向对象编程、实现 Excel 中的 VLOOKUP 与透视表、R 网络爬虫、R 高性能计算、R 最新机器学习框架：mlr3verse，tidymodels.

大家可以根据自己的需求选择阅读侧重点，不过我还是希望您能够按照顺序完整地阅读，这样才能让您彻底地更新一遍您的 R 知识，避免 R base 与 tidyverse 混着用，因为二者在写 R 代码上不是一个思维，强行搭在一起处处透着别扭，且事倍功半。

本书所用的软件

本书使用最新版本的 R 语言 4.1.1 和 RStudio 1.4，主要使用的 R 包是 tidyverse 1.3.1 系列。

本书的配套资源下载

本书的 R 程序均作为 Rmarkdown 中的代码调试通过，所有示例的数据、R 程序、教学幻灯片都可以在出版社官网、Github（https://github.com/zhjx19/introR）、码云（https://gitee.com/zhjx19/introR）下载。

致谢

感谢我的爱人及岳父岳母，在家庭生活方面给予我诸多照顾，让我能安心地创作；感谢我的兄弟及父母家人，特别感谢我远在河北老家的弟弟，在我无能为力的时候，义无反顾地照顾生病住院和在家养病的父亲，免去了我的后顾之忧。

感谢 Hadley 大神开发的 tidyverse 包让 R 语言用起来有如神助，谢益辉大神开发的 Rmarkdown/-bookdown 非常便于写书，黄湘云 & 叶飞整合的 ElegantBookdown 模板。

感谢知乎平台及知乎上的读者们，你们让本书有机会为广大的读者知晓。感谢“tidy-R”“Use R！”“‘数据之美”QQ 群的群主和群里的很多朋友，大家一起学习 R，一起解答问题，非常开心！也谢谢你们对我的支持以及对本书的期待，你们给了我写这本书的动力！谢谢群友们帮忙指出书中的错误，特别感谢好友楚新元、‘‘随鸟走天涯”“‘庄闪闪’’ 等，对本书部分章节中的内容给予很好的建议和很大的帮助。

感谢胡俊英编辑通过知乎平台找到我，并全力促成了本书的出版，期间对我的各种指导和督促，为本书的出版做了大量认真细致的工作。感谢在工作和生活中帮助过我的领导、同事、朋友们，感谢你们，正是因为有了你们，才有了本书的面世。

本书是在黑龙江省哲学社科项目青年项目：全面二孩政策对黑龙江省人口的影响及对策研究（项目号：17TJC134）资助下完成，一并表示感谢！

关于勘误

虽然花了很多时间和精力去核对书中的文字、代码和图片，但因为时间仓促和水平有限，书中仍难免会有一些错误和纰漏，如果大家发现有什么问题或疑问，恳请反馈给我，也非常欢迎大家与我探讨R语言编程相关的技术，相关信息可发到我的邮箱 zhjx_19@hrbcu.edu.cn，或者在本书的读者群 `tidy-R` QQ群（875664831）在线交流，或者在我的知乎专栏 https://www.zhihu.com/people/huc_zhangjingxin相关文章下面评论或私信，我肯定会努力回答疑问或者指出一个正确的方向。

前言

开篇先来谈一谈，我所理解的编程之道。具体来说，将讨论怎么学习编程语言、R 语言简介、R 语言编程思想，特别是向量化编程思维。

温馨提示

本章为了阐述需要，会涉及一些 R 语言代码，读者可以先忽略代码细节，只当它们是计算过程“翻译”而成，把关注点放在我想传达的编程思维上。

0.1 怎么学习编程语言?

编程语言是人与计算机沟通的一种形式语言，根据设计好的编程元素和语法规则，来严格规范地表达我们想要做的事情的每一步（程序代码），使得计算机能够明白并正确执行，得到期望的结果。

编程语言和数学语言很像，数学语言是最适合表达科学理论的形式语言，用数学符号、数学定义和逻辑推理，来规范严格地表达科学理论。

很多人说：“学数学，就是靠大量刷题”，“学编程，就是照着别人的代码敲代码”。

我不认可这种观点，这样的学习方法事倍功半，关键是这样做你学不会真正的数学，也学不会真正的编程！

那么应该怎么学习编程语言呢？

就好比要成为一个好的厨师，首先得熟悉各种常见食材的特点秉性，掌握各种基本的烹饪方法，然后就能根据客人需要随意组合食材和烹饪方法制作出各种可口的大餐。

数学的食材就是**定义**，烹饪方法就是**逻辑推理**，一旦你真正掌握了**定义 + 逻辑推理**，各种基本的数学题都不在话下，而且你还学会了数学知识。

同理，编程的食材和烹饪方法就是**编程元素**和**语法规则**，比如数据结构（容器）、分支/循环结构、自定义函数等，一旦你掌握了这些**编程元素**和**语法规则**，根据问题的需要，你就能信手拈来优化组合它们，从而自己写出代码解决问题。

所以，学习任何一门编程语言，根据我的经验，有这么几点建议（步骤）：

(1) 理解该编程语言的核心思想，比如 Python 是面向对象，R 语言是面向函数也面向对象，另外，高级编程语言还都倡导向量化编程。在核心思想的引领下去学它去思考去写代码。
(2) 学习该编程语言的基础知识，这些基础知识本质上是相通的同样的东西，只是在不同编程语言下披上了其特有的外衣（编程语法），基础知识包括：数据类型及数据结构（容器）、分支/循环结构、自定义函数、文件读写、可视化等。
(3) 前两步完成之后，就算基本入门[1]了，可以根据需要，根据遇到的问题，借助网络搜索、借助帮

[1]至少要经历过一种编程语言的入门，再学习其他编程语言就会很快。

助，遇到问题解决问题，逐步提升，用的越多会的越多，也越熟练。

以上是学习编程语言的正确、快速、有效的方法，切忌不学基础语法，用到哪就突击哪，找别人的代码一顿不知其所以然的瞎改，这样的结果是：**自以为节省时间，实际上是浪费了几十倍的时间**，关键是始终无法入门，更谈不上将来提高。

再来谈一个学编程过程中普遍存在的问题：如何跨越**“能看懂别人的代码”到“自己写代码”**的鸿沟。

绝大多数人在编程学习过程中，都要经历这样一个过程：

第 1 步：学习基本语法

第 2 步：能看懂和调试别人的代码

↓（编程之门）

第 3 步：自己写代码

前两步没有任何难度，谁都可以做到。从第 2 步到第 3 步是一个“坎”，很多人困惑于此而无法真正进入编程之门。网上也有很多讲到如何跨越这步的文章和说法，但基本都是脱离实际操作的空谈，比如多敲书上的代码之类，治标不治本（只能提升编程基本知识）。

我的理念说白了也很简单，无非就是：**分解问题 + 实例梳理 + 翻译及调试**，具体如下：

- 将难以入手大问题分解为可以逐步解决的小问题
- 用计算机的思维去思考解决每步小问题
- 借助类比的简单实例和代码片段，梳理出详细算法步骤
- 将详细算法步骤用逐片段地用编程语法翻译成代码并调试通过

关于调试补充一点，可以说高级编程语言的程序代码就是逐片段调试出来的，借助简单实例按照算法步骤，从上一步的结果调试得到下一步的结果，依次向前推进直到到达最终的结果。还有一点经验之谈：**写代码时，随时跟踪关注每一步执行，变量、数据的值是否到达你所期望的值，非常有必要！**

这是我用数学建模的思维得出的科学的操作步骤。为什么大家普遍自己写代码解决具体问题时感觉无从下手呢？

这是因为你总想一步就从**问题**到**代码**，没有中间的过程，即使编程高手也做不到。当然，编程高手可能能缩减这个过程，但不能省略这个过程。其实你平时看编程书是被欺骗了：编程书上只写问题（或者有简单分析）紧接着就是代码，给人的感觉就是应该从问题直接到代码，大家都是这么写出来的。其实不然。

所以，改变从**问题**直接到**代码**思维固式，按我上面说的步骤去操作，每一步是不是都不难解决。那么，自然就从无从下手，到能锻炼自己写代码了。

开始你可能只能自己写代码解决比较简单的问题，但是这种成就感再加上慢慢锻炼下来，你自己写代码的能力会越来越强，能解决问题也会越来越复杂，当然前提是，你已经真正掌握了基本编程语法，可以随意取用。当然二者也是相辅相成、共同促进和提高的关系。

好，说清了这个道理，下面拿一个具体的小案例来演示一下。

例 0.1 计算并绘制 ROC 曲线

ROC 曲线是二分类机器学习模型的性能评价指标，已知测试集或验证集每个样本真实类别及其模型预测概率值，就可以计算并绘制 ROC 曲线。

先来梳理一下问题，ROC 曲线是在不同分类阈值上对比真正率（TPR）与假正率（FPR）的曲线。

分类阈值就是根据预测概率判定预测类别的阈值，要让该阈值从 0 到 1 以足够小的步长变化，对于每个阈值 c，比如 0.85，则预测概率 ≥ 0.85 判定为 “Pos’’，< 0.85 判定为 “Neg.” 这样就得到了预测类别。

根据真实类别和预测类别，就能计算混淆矩阵：

		真实类别（y）	
		+	-
预测类别（$\hat{y}$）	+	TP（真正）	FP（假正）
	-	FN（假负）	TN（真负）

图 1: 混淆矩阵示意图

进一步就可以计算：

$$TPR = \frac{TP}{TP + FN}, \qquad FPR = \frac{FP}{FP + TN}$$

有一个阈值，就能计算一组 TPR 和 FPR，循环迭代都计算出来并保存。再以 FPR 为 x 轴，以 TPR 为 y 轴绘图，则得到 ROC 曲线。

于是，梳理一下经过分解后的问题：

- 让分类阈值以某步长在 $[1, 0]$ 上变化取值；
- 对某一个阈值，
  - 计算预测类别
  - 计算混淆矩阵
  - 计算 TPR 和 FPR
- 循环迭代，计算所有阈值的 TPR 和 FPR
- 根据 TPR 和 FPR 数据绘图

下面拿一个小数据算例，借助代码片段来推演上述过程。现在读者不用纠结于代码，更重要的是体会，自己写代码解决实际问题的这样一个思考过程。

```
library(tidyverse)
df = tibble(
  ID = 1:10,
  真实类别 = c("Pos","Pos","Pos","Neg","Pos","Neg","Neg","Neg","Pos","Neg"),
```

```
  预测概率 = c(0.95,0.86,0.69,0.65,0.59,0.52,0.39,0.28,0.15,0.06))
knitr::kable(df)
```

ID	真实类别	预测概率
1	Pos	0.95
2	Pos	0.86
3	Pos	0.69
4	Neg	0.65
5	Pos	0.59
6	Neg	0.52
7	Neg	0.39
8	Neg	0.28
9	Pos	0.15
10	Neg	0.06

先来解决对某一个阈值，计算 TPR 和 FPR。以 $c = 0.85$ 为例。

计算预测类别，实际上就是 `If-else` 语句根据条件赋值，当然是用整洁的 `tidyverse` 来做。顺便多做一件事情：把类别变量转化为因子型，以保证 “Pos” 和 “Neg” 的正确顺序，与混淆矩阵中一致。

```
c = 0.85
df1 = df %>%
  mutate(
    预测类别 = ifelse(预测概率 >= c, "Pos", "Neg"),
    预测类别 = factor(预测类别, levels = c("Pos", "Neg")),
    真实类别 = factor(真实类别, levels = c("Pos", "Neg")))
knitr::kable(df1)
```

ID	真实类别	预测概率	预测类别
1	Pos	0.95	Pos
2	Pos	0.86	Pos
3	Pos	0.69	Neg
4	Neg	0.65	Neg
5	Pos	0.59	Neg
6	Neg	0.52	Neg
7	Neg	0.39	Neg
8	Neg	0.28	Neg
9	Pos	0.15	Neg
10	Neg	0.06	Neg

计算混淆矩阵，实际上就是统计交叉频数，本来为 “Pos” 预测为 “Pos” 的有多少，等等。用 R 自带的 `table()` 函数就能实现：

```
cm = table(df1$预测类别, df1$真实类别)
cm
```

```
##
##       Pos Neg
##   Pos   2   0
##   Neg   3   5
```

计算 TPR 和 FPR。根据其的计算公式，从混淆矩阵中取数计算即可。这里咱们再高级一点，用向量化计算来实现，毕竟高级编程语言提倡向量化嘛。向量化编程，是对一列/矩阵的数同时做同样的操作，既提升程序效率又大大简化代码。

向量化编程，关键是要用整体考量的思维来思考、来表示运算。比如这里计算 TPR 和 FPR，通过观察可以发现：混淆矩阵的第 1 行各元素，都除以其所在列和，正好是 TPR 和 FPR。

```
cm["Pos",] / colSums(cm)
```

```
## Pos Neg
## 0.4 0.0
```

这就完成了本问题的核心部分。接下来，要循环迭代对每个阈值，都计算一遍 TPR 和 FPR。用 `for` 循环当然可以，但咱们仍然更高级一点：泛函式编程。

先把上述计算封装为一个 **自定义函数**，该函数只要接受一个前文原始的数据框 `df` 和一个阈值 `c`，就能返回来你想要的 TPR 和 FPR。然后，再把该函数 **应用**到数据框 `df` 和一系列的阈值上，循环迭代自然就完成了。这就是 **泛函式编程**。

```
cal_ROC = function(df, c) {
  df = df %>%
  mutate(
    预测类别 = ifelse(预测概率 >= c, "Pos", "Neg"),
    预测类别 = factor(预测类别, levels = c("Pos", "Neg")),
    真实类别 = factor(真实类别, levels = c("Pos", "Neg")))
  cm = table(df$预测类别, df$真实类别)
  t = cm["Pos",] / colSums(cm)
  list(TPR = t[[1]], FPR = t[[2]])
}
```

测试一下这个自定义函数，能不能算出来刚才的梳理时结果：

```
cal_ROC(df, 0.85)
```

```
## $TPR
```

```
## [1] 0.4
##
## $FPR
## [1] 0
```

没问题，下面将该函数应用到一系列的阈值上（循环迭代），并一步到位将每次计算的两个结果按行合并到一起，这就彻底完成数据计算：

```
c = seq(1, 0, -0.02)
rocs = map_dfr(c, cal_ROC, df = df)
head(rocs)      # 查看前 6 个结果
```

```
## # A tibble: 6 x 2
##     TPR   FPR
##   <dbl> <dbl>
## 1   0       0
## 2   0       0
## 3   0       0
## 4   0.2     0
## 5   0.2     0
## 6   0.2     0
```

最后，用著名的 ggplot2 包绘制 ROC 曲线图形：

```
rocs %>%
  ggplot(aes(FPR, TPR)) +
  geom_line(size = 2, color = "steelblue") +
  geom_point(shape = "diamond", size = 4, color = "red") +
  theme_bw()
```

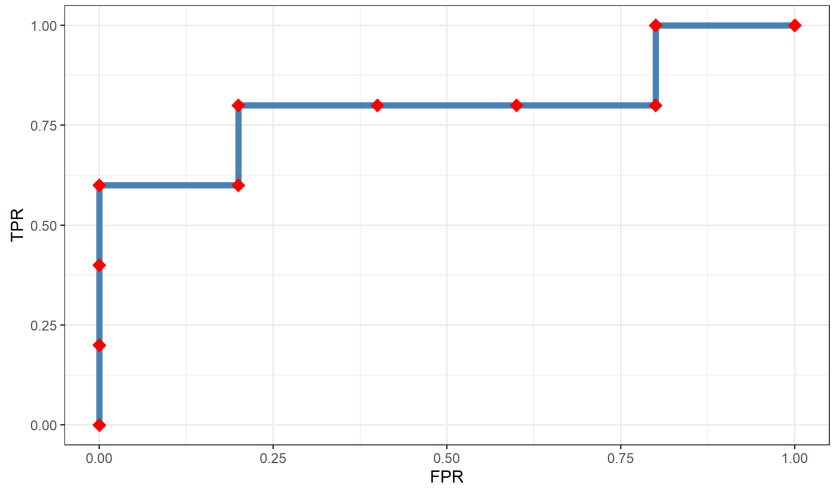

图 2: 绘制 ROC 曲线

以上就是我所主张的学习编程的正确方法，我不认为**照着别人的编程书敲代码**是学习编程的好方法。

0.2 R 语言简介

0.2.1 什么是数据科学?

数据科学是综合了统计学、计算机科学和领域知识的交叉学科，其基本内容就是用数据的方法研究科学，用科学的方法研究数据（鄂维南院士）。

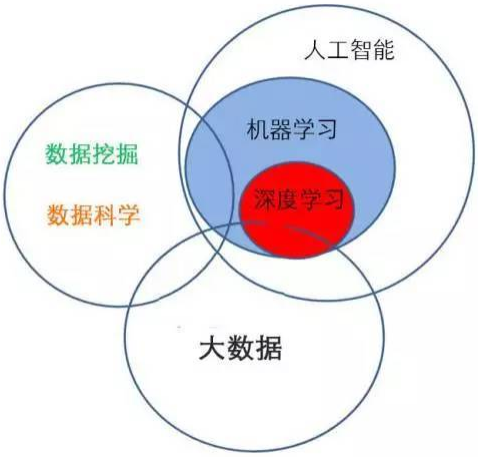

图 3: 数据科学的位置

Hadley Wickham 定义了数据科学的工作流程:

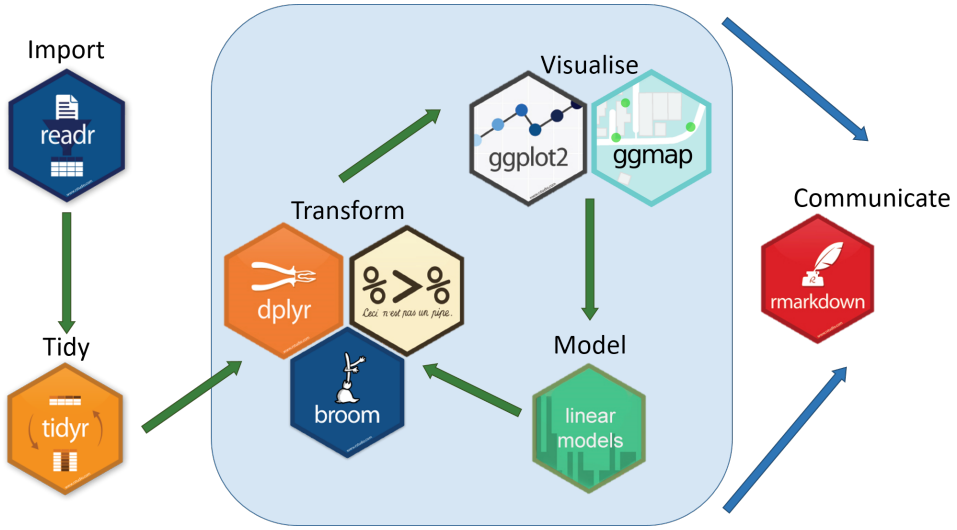

图 4: 数据科学的工作流程

即数据导入、数据清洗、数据变换、数据可视化、数据建模以及文档沟通，整个分析和探索过程，我们应当训练这样的数据思维。

0.2.2 什么是R语言?

1992 年，新西兰奥克兰大学统计学教授 Ross Ihaka 和 Robert Gentleman，为了方便地给学生教授统计学课程，他们设计开发了 R 语言（他们名字的首字母都是 R)。

- R 重要事件：
  - 2000 年，R 1.0.0 发布
  - 2005 年，ggplot2 包（2018.8 - 2019.8 下载量超过 1.3 亿次）
  - 2016 年，Rstudio 公司推出 `tidyverse` 包（数据科学当前最新 R 包）
  - 2021 年，R 4.1.0 发布，目前 CRAN 上的 R 包数量为 18073, 近两年增速明显加快

Aug 2021	Aug 2020	Change	Programming Language	Ratings	Change
1	1		C	12.57%	-4.41%
2	3	↑	Python	11.86%	+2.17%
3	2	↓	Java	10.43%	-4.00%
4	4		C++	7.36%	+0.52%
5	5		C#	5.14%	+0.46%
6	6		Visual Basic	4.67%	+0.01%
7	7		JavaScript	2.95%	+0.07%
8	9	↑	PHP	2.19%	-0.05%
9	14	⇈	Assembly language	2.03%	+0.99%
10	10		SQL	1.47%	+0.02%
11	18	⇈	Groovy	1.36%	+0.59%
12	17	⇈	Classic Visual Basic	1.23%	+0.41%
13	42	⇈	Fortran	1.14%	+0.83%
14	8	⇊	R	1.05%	-1.75%

图 5: TIOBE 最新编程语言排名

`IEEE Spectrum` 最新发布的 2021 年度编程语言排行榜，从涵盖社交网站、开源代码网站和求职网站的 8 个信息源：`CareerBuilder`、`GitHub`、`Google`、`Hacker News`、`IEEE`、`Reddit`、`Stack Overflow` 和 `Twitter`，按照 11 个指标收集数据，最终得到了数十种编程语言流行度的整体排名：

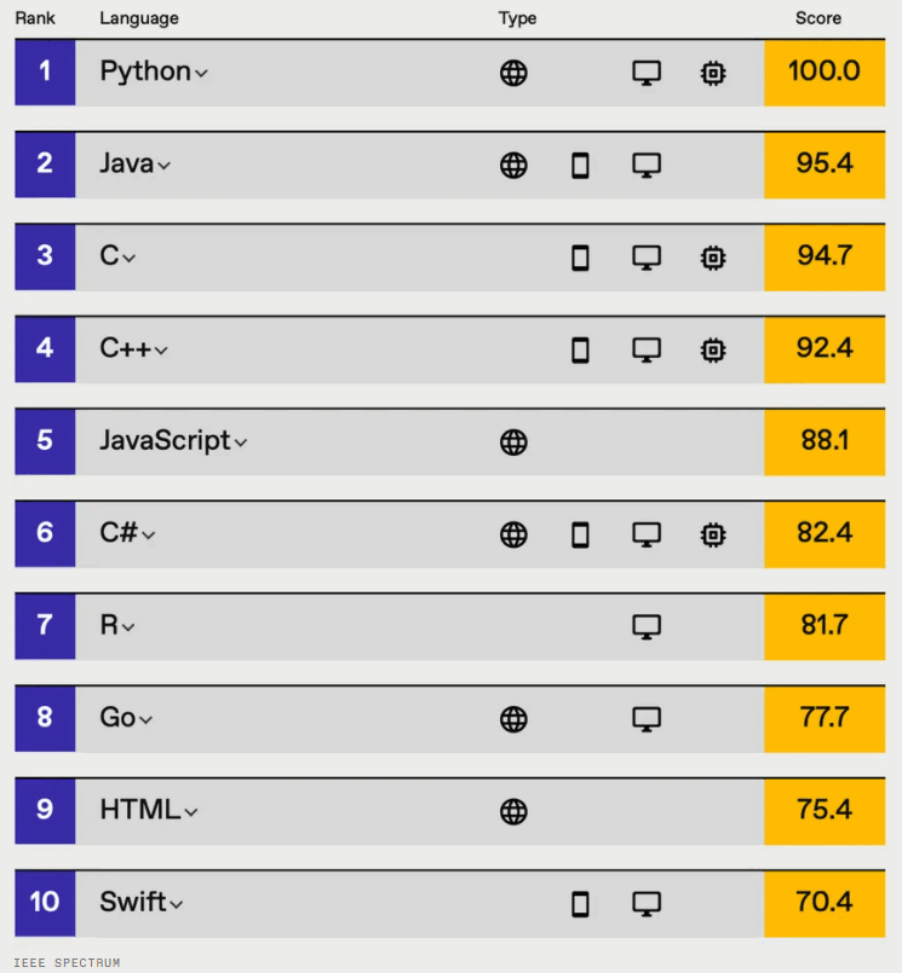

图 6: IEEE Spectrum 2021 年度编程语言排行榜

2019 年权威机构 KDnuggets 做过调研，显示数据科学领域最受欢迎的编程语言，包括 Python 和 R：

- Python 更全能，适合将来做程序员或在工业企业工作
- R 更侧重数据统计分析，适合将来做科研学术

R 语言是用于统计分析，图形表示和报告的编程语言：

- R 语言是统计学家开发，为统计计算、数据分析和可视化而设计
- R 语言适合做数据处理和数据建模（数据预处理、数据探索性分析、识别数据隐含的模式、数据可视化）。

R 语言的优势：

- 免费开源，软件体积小根据需要安装扩展包，兼容各种常见操作系统，有强大活跃的社区
- 专门为统计和数据分析开发的语言，有丰富的扩展包
- 拥有顶尖水准的制图功能
- 面向对象和函数，比 Python 简单易学

在热门的机器学习领域：

- 有足以媲美 Python 的 sklearn 机器学习库的 R 机器学习包：`mlr3verse` 或 `tidymodels` 见附录E.

本节部分内容参阅 (王敏杰 2020).

0.2.3 一个改变了R 的人

图 7: R 语言大神 — Hadley Wickham

- Hadley Wickham 博士，为统计应用领域做出了突出贡献的统计学家，被称为改变了 R 的人；
- 2019 年，被国际统计学年会上授予 COPSS 奖，该奖项是国际统计学领域的最高奖项，被誉为‘‘统计学的诺贝尔奖’’；
- Wickham 现为 Rstudio 首席科学家，同时为奥克兰大学、斯坦福大学和赖斯大学的统计系兼职教授；
- Wickham 为了使得数据科学更简洁、高效、有趣，开发/出版了大量知名 R 包及 R 相关书籍，主要包括：
- 数据科学：tidyverse
  - ggplot2, 数据可视化
  - dplyr, 数据操作
  - tidyr, 数据清洗
  - stringr, 处理字符串
  - lubridate, 处理日期时间
- 数据导入：
  - readr, 读入.csv/fwf 文件
  - readxl, 读入.xls/.xlsx 文件
  - haven, 读入 SAS/SPSS/Stata 文件.

    - httr, 与 web 交互的 APIs
    - rvest, 网页爬虫
    - xml2, 读入 XML 文件
- R 开发工具：
    - devtools, 开发 R 包
    - roxygen2, 生成内联（in-line）文档
    - testthat, 单元测试
    - pkgdown, 创建美观的包网页
- 出版书籍：
    - 《R for Data Science》, 介绍用 R 做数据科学的关键工具
    - 《ggplot2: elegant graphics for data analysis》, 展示如何使用 ggplot2 创建有助于理解数据的图形
    - 《Advanced R》, 帮助您掌握 R 编程语言，以及使用 R 的深层技巧
    - 《R packages》，讲授良好的 R 软件项目实践，科学地创建 R 包：打包文件、生成文档、测试代码

0.3 R 语言编程思想

0.3.1 面向对象

R 是一种基于对象的编程语言，即在定义类的基础上，创建与操作对象；数值向量、函数、图形等都是对象。Python 的**一切皆为对象**也适用于 R。

```
a = 1L
class(a)
```

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

```
b = 1:10
class(b)
```

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

```
f = function(x) x + 1
class(f)
```

```
## [1] "function"
```

早期和底层 R 语言中的面向对象编程是通过泛型函数来实现的，以 S3 类、S4 类为代表。新出现的 R6 类更适合用来实现通常所说的面向对象编程（OOP），包含类、属性、方法、继承、多态等概念。

面向对象的内容是 R 语言编程的高级内容，有上述基本了解即可，本书不做具体展开，只在附录放一个用 R6 类面向对象编程的简单示例，有兴趣的读者可参阅 (Wickham 2019a)。

0.3.2 面向函数

笼统来说，R 语言就两个事情：数据，对数据应用操作。这个操作就是函数，包括 R 自带的函数，各种扩展包里的函数，自定义的函数。

所以，使用 R 大部分时间都是在与函数打交道，学会了使用函数，R 语言也就学会了一半，**啊？R 这么简单？**确实差不多[2]，很多人说 R 简单易学，也是因为此。

编程中的函数，是用来实现某个功能。很多时候，我们使用 R 自带的或来自其它包中的现成函数就够了。

那么，如何找到并使用现成函数解决自己想要解决的问题？比如想做线性回归，通过 `Bing` 搜索知道是用自带 `lm()` 函数实现。那么先打开该函数的帮助：

```
?lm
```

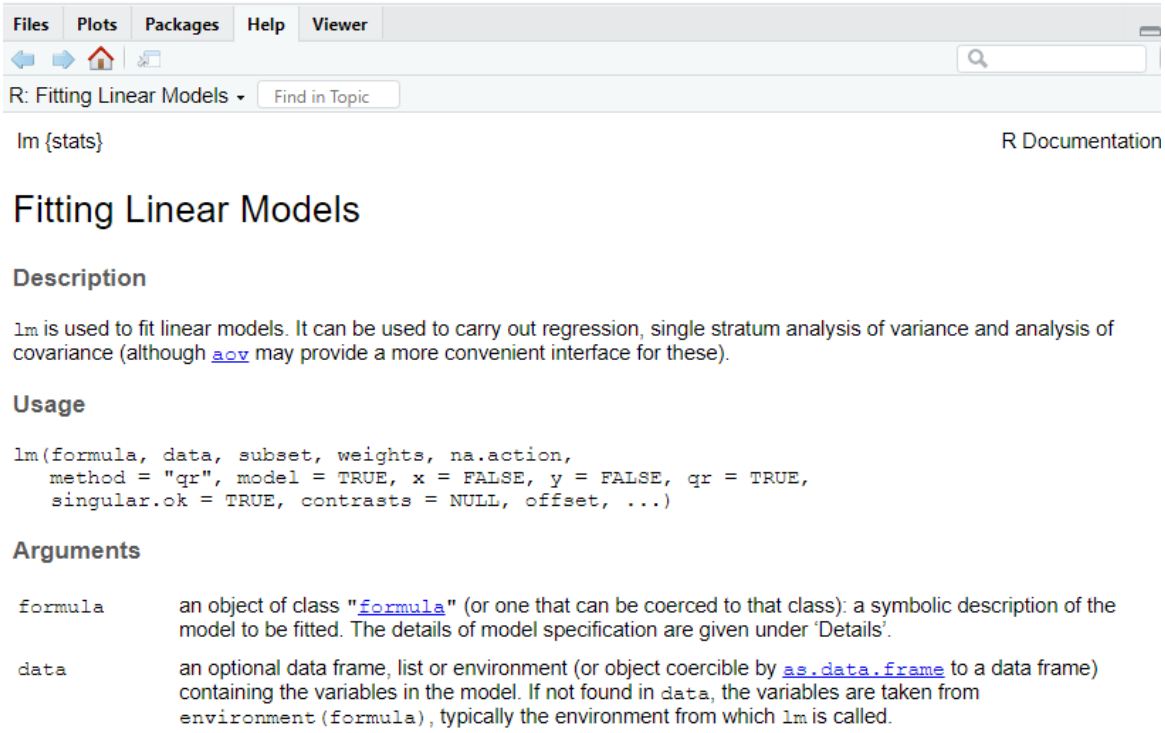

图 8: R 函数的帮助页面

执行 `?` 函数名，若函数来自扩展包需要事先加载包，则在 `Rstudio` 右下角窗口打开函数帮助界面，一般至少包括如下内容：

- 函数描述
- 函数语法格式
- 函数参数说明
- 函数返回值
- 函数示例

[2]前提是不把 R 当一门编程语言，只想简单套用现成的算法模型。

通过阅读函数描述、参数说明、返回值，再调试示例，就能快速掌握该函数的使用。

函数包含很多参数，常用参数往往只是前几个。比如 `lm()` 的常用参数是：

- `formula`: 设置线性回归公式形式：因变量 ~ 自变量 + 自变量
- `data`: 提供数据（框）

使用自带的 `mtcars` 数据集演示，按照函数参数要求的对象类型提供实参：

```
head(mtcars)
```

```
##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```
model = lm(mpg ~ disp, data = mtcars)
summary(model)      # 查看回归汇总结果
```

```
##
## Call:
## lm(formula = mpg ~ disp, data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.8922 -2.2022 -0.9631  1.6272  7.2305
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 29.599855   1.229720  24.070  < 2e-16 ***
## disp        -0.041215   0.004712  -8.747 9.38e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.251 on 30 degrees of freedom
## Multiple R-squared:  0.7183, Adjusted R-squared:  0.709
## F-statistic: 76.51 on 1 and 30 DF,  p-value: 9.38e-10
```

所有的 R 函数，即使是陌生的，都可以这样来使用。

编程中一种重要的思维就是**函数式思维**，包括自定义函数（把解决某问题的过程封装成函数）和

泛函式编程（把函数依次应用到一系列的对象上）。

如果找不到现成的函数解决自己的问题，那就需要自己自定义函数，R 自定义中函数的基本语法为：

```
函数名 = function(输入 1, ..., 输入 n) {
  ...
  return(输出)        # 若有多个输出，需要打包成一个 list
}
```

比如，想要计算很多圆的面积，就有必要把如何计算一个圆的面积定义成函数，需要输入半径，才能计算想要的面积：

```
AreaCircle = function(r) {
  S = pi * r * r
  return(S)
}
```

这样再计算圆的面积，你只需要把输入给它，它就能在内部进行相应处理，把你想要的输出结果返回给你。如果想批量计算圆的面积，按泛函式编程思维，只需要将该函数依次应用到一系列的半径上即可。

比如计算半径为 5 的圆的面积和批量计算半径为 2,4,7 的圆的面积：

```
AreaCircle(5)
```

```
## [1] 78.53982
```

```
rs = c(2,4,7)
map_dbl(rs, AreaCircle)      # purrr 包
```

```
## [1]  12.56637  50.26548 153.93804
```

所以，定义函数就好比创造一个模具，调用函数就好比用模具批量生成产品。使用函数最大的好处，就是将实现某个功能，封装成模具，从而可以反复使用。这就避免了写大量重复的代码，程序的可读性也大大加强。

0.3.3 向量化编程

高级编程语言都提倡 **向量化编程**[3]，说白了就是，对一列/矩阵/多维数组的数同时做同样的操作，既提升程序效率又大大简化代码。

向量化编程，关键是要用整体考量的思维来思考、来表示运算，这需要用到《线性代数》的知识，其实我觉得《线性代数》最有用的知识就是向量/矩阵化表示运算。

[3]向量化编程中的向量，泛指向量/矩阵/多维数组。

比如考虑 n 元一次线性方程组：

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

若从整体的角度来考量，引入矩阵和向量：

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

则前面的 n 元一次线性方程组，可以向量化表示为：

$$Ax = b$$

可见，向量化表示大大简化了表达式。这放在编程中，就相当于本来用两层 `for` 循环才能表示的代码，简化为了短短一行代码。

向量化编程其实并不难，关键是要转变思维惯式：很多人学完 C 语言后的后遗症，就是首先想到的总是逐元素的 `for` 循环。摆脱这种想法，调动头脑里的《线性代数》知识，尝试用向量/矩阵表示，长此以往，向量化编程思维就有了。

下面以计算决策树算法中的样本经验熵为例来演示向量化编程。

对于分类变量 D，$\frac{|D_k|}{|D|}$ 表示第 k 类所占的比例，则 D 的样本经验熵为

$$H(D) = -\sum_{k=1}^{K} \frac{|D_k|}{|D|} \ln \frac{|D_k|}{|D|}$$

其中，$|\cdot|$ 表示集合包含的元素个数。

实际中经常遇到要把数学式子变成代码，与前文自己写代码所谈到的一样，首先你要看懂式子，拿简单实例逐代码片段调试就能解决。

以著名的'' 西瓜书'' 中的西瓜分类数据中的因变量'' 好瓜'' 为例，表示是否为好瓜，取值为 ''是'' 和 ''否''：

```
y = c(rep(" 是", 8), rep(" 否", 9))
y
```

```
##  [1] "是" "是" "是" "是" "是" "是" "是" "是" "否" "否" "否" "否" "否"
## [14] "否" "否" "否" "否"
```

则 D 分为两类：D_1 为好瓜类，D_2 为坏瓜类。

从内到外先要计算 $|D_k|/|D|$, $k = 1, 2$, 用向量化的思维同时计算，就是统计各分类的样本数，再除以总样本数：

```
table(y)                    # 计算各分类的频数，得到向量
```

```
## y
## 否 是
##  9  8
```

```
p = table(y) / length(y)    # 向量除以标量
p
```

```
## y
##        否        是
## 0.5294118 0.4705882
```

继续代入公式计算，记住 R 自带函数天然就接受向量做输入参数：

```
log(p)                      # 向量取对数
```

```
## y
##         否         是
## -0.6359888 -0.7537718
```

```
p * log(p)                  # 向量乘以向量，对应元素做乘法
```

```
## y
##         否         是
## -0.3366999 -0.3547161
```

```
- sum(p * log(p))           # 向量求和
```

```
## [1] 0.6914161
```

看着挺复杂的公式用向量化编程，核心代码只有两行：计算 p 和最后一行。这个实例虽然简单，但基本涉及所有常用的向量化操作：

- 向量与标量做运算；
- 向量与向量做四则运算；
- 函数作用到向量。

第一章 基础语法

本章将学习 R 语言基本语法，也是与其他编程语言相通的部分，包括：搭建 R 环境、常用数据类型（数据结构）、控制结构（分支、循环）、自定义函数。

本章目的是让读者打好 R 语言基本语法的基础，以及训练**函数式编程思维**：自定义函数解决问题 + 泛函式循环迭代。

基本语法是编程的**编程元素**和**语法规则**，您编写的所有程序都是用它们组合出来的。**函数式编程**，是下一章训练**数据思维**的基础。**函数式编程和数据思维**，是 R 语言编程的最核心编程思维，这些也是学习本书和学习 R 语言的关键所在。

1.1 搭建 R 环境及常用操作

1.1.1 搭建 R 环境

R 语言原生官网速度慢，建议直接到 R 镜像站，目前国内有 9 个镜像站，我最近常用的是北京外国语大学的：

https://mirrors.bfsu.edu.cn/CRAN/

根据自己的操作系统，下载相应的最新版 R-4.1.1 安装即可，由于免费软件都是简单的下一步，不再赘述。Windows 系统安装时可根据系统只选择 32 位或 64 位版本。

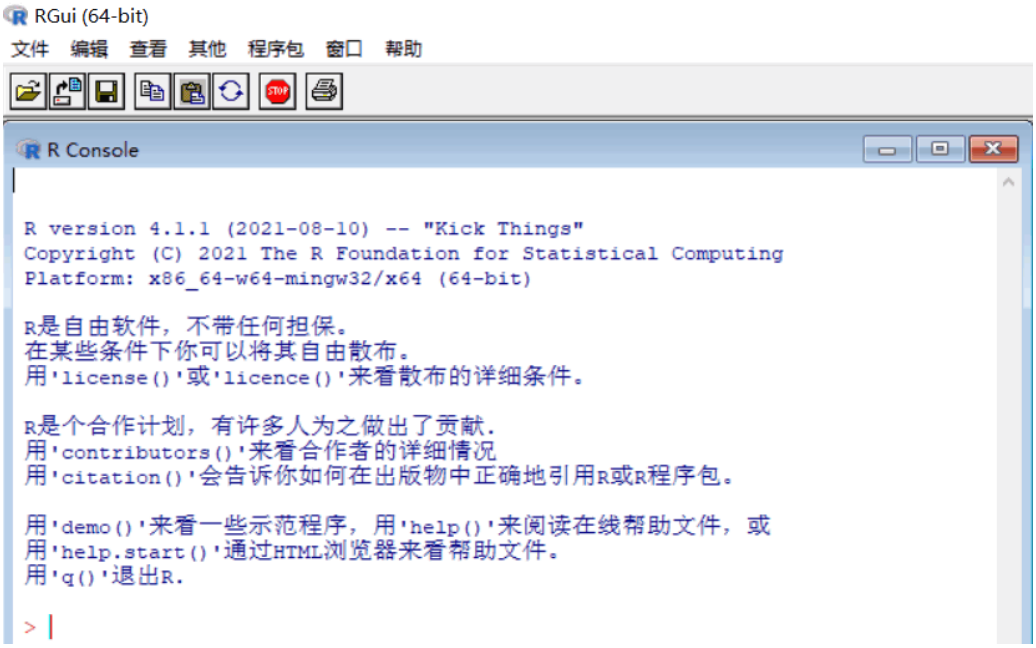

图 1.1: R 4.1.1 软件界面

建议安装在 D 盘，不要有中文路径，且路径不要有空格。
切记：若 Windows 系统用户名为中文，先改成英文！
注意，最好保证电脑里有且只有一个版本的 R，否则 RStudio 不会自动关联 R ，需要手动关联到其中一个 R.

安装 RStudio

不要直接使用 R, 而是使用更好用的 R 语言集成开发环境 Rstudio，官网下载地址：

https://www.rstudio.com/products/rstudio/download

下载安装（或直接下载 zip 版解压），将自动关联已安装的 R。

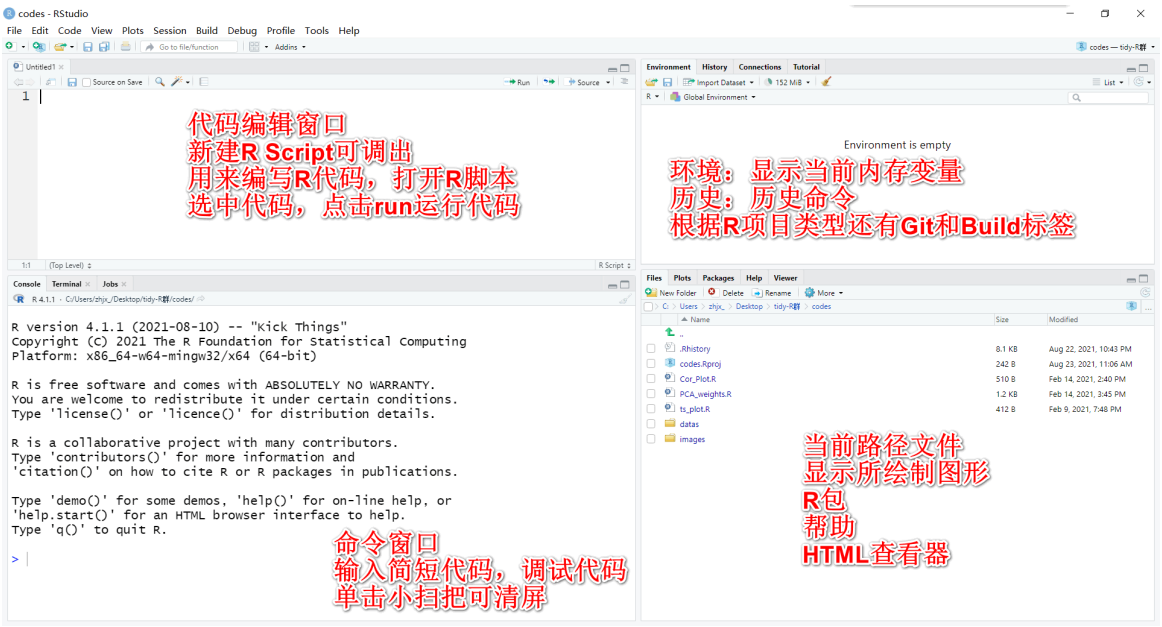

图 1.2: R Studio 操作界面

一些必要的设置

- 切换安装扩展包的国内镜像源（任选其一）

`Tools -> Global Options… -> Options -> Packages`，点击 `Change` 修改为，比如北京外国语大学镜像源：

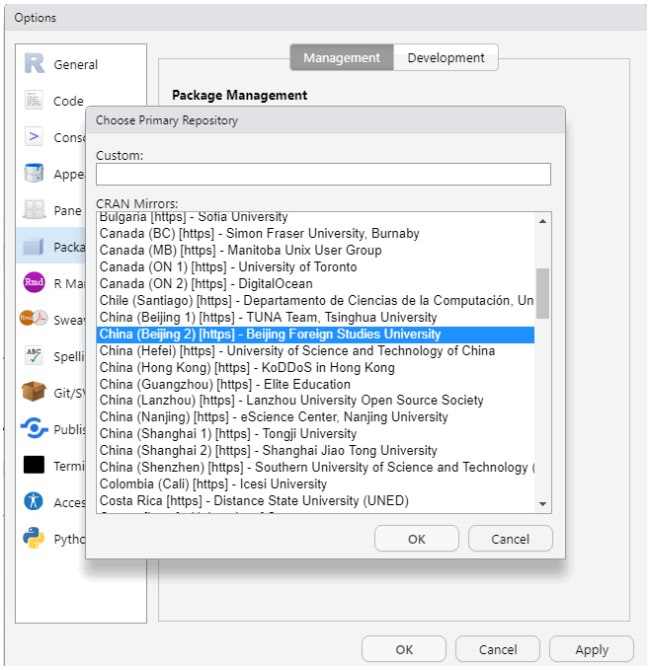

图 1.3: RStudio 设置国内镜像源

- 设置保存文件的默认编码方式为 UTF-8

Tools -> Global Options… -> code -> Saving，在 Default text encoding 框，点 change，修改为 UTF-8:

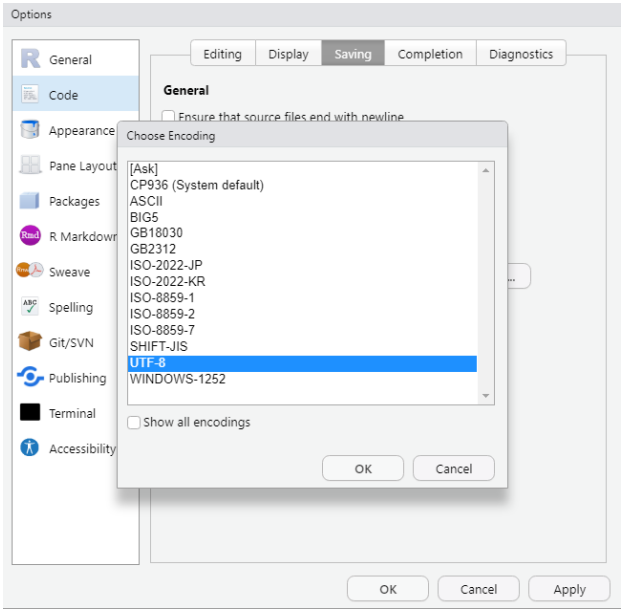

图 1.4: R Studio 设置 code 编码

这样保存出来的各种 R 文件，都将采用 UTF-8 编码，能够尽量避免含有中文的文件在其他电脑上打开乱码。

建议顺便再勾选 Code -> Display 下的 Rainbow parentheses 框，这样代码中的配对括号将用不同彩虹色匹配。

另外，还建议设置 General -> Workspace 取消勾选 Restore .RData into workspace at startup，并将其下的 save workspace to .RData on exit: 改为 Never. 避免每次打开 RStudio 都加载之前的当前内存数据。

1.1.2 常用操作

安装包

扩展包（package），简称包。通常 R 包都来自 CRAN，审核比较规范严格，包的质量相对更有保障。建议使用命令安装：

```
install.packages("openxlsx")
```

> openxlsx 为包名，必须要加引号 (R 中单双引号通用) 。

有些包不能自动安装，可以手动从网上搜索到下载.zip 或.tar.gz 文件到本地，再手动安装（不建议）：

Tools -> Install Packages，修改 Install from，然后浏览安装：

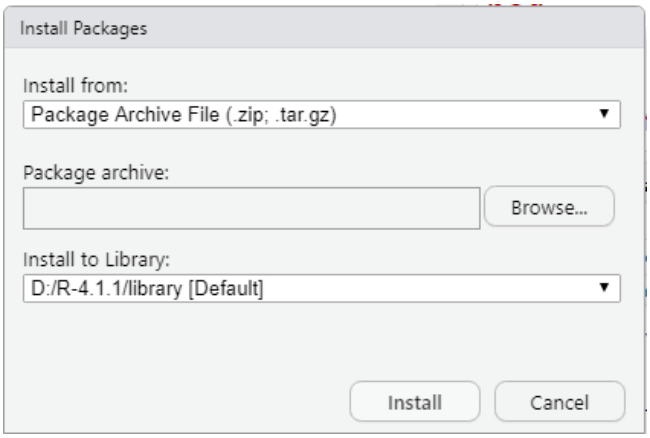

图 1.5: 手动安装 R 包

> 手动安装包，经常容易安装失败，通常是因为没有先安装该包的依赖包，故需要去包的网页查询其依赖包，确定若未安装，需要先安装它们。这往往又涉及依赖包的依赖包，所以最好不要手动安装包。另外，尽量用最新版本的 R 会减少很多安装包失败。

Github 也是 R 包的较大的来源，有些作者自己开发的 R 包只放在 Github，也有很多 CRAN R 包的最新开发版都位于 Github. 可以先安装 devtools 或 remotes 包，再用其 install_github() 安装 Github 来源的包：

```
devtools::install_github("tidyverse/dplyr")  # 或者
remotes::install_github("tidyverse/dplyr")
```

:: 前面是包名，这是不单独加载包，而使用包中函数的写法。
hadlley 为 Github 用户名，dplyr 为该用户的名为 dplyr 的 repository （仓库），也是包名。
注意，不是所有的仓库都是 R 包（含有 DESCRIPTION 文件是 R 包的标志）。

若网络等原因，导致直接从 Github 安装包失败，也可以将整个包文件夹从网页下载下来，解压缩到当前路径（或提供完整路径），再从本地安装它：

```
install.packages("dplyr-master", repos=NULL, type="source")
```

另外，生信领域在 R 中自成一派，有专门的包的大本营：

https://www.bioconductor.org

先安装 BiocManager 包，再用 install() 函数安装 bioconductor 来源的包：

```
BiocManager::install("openxlsx")
```

实用场景

R 包默认都安装在 .../R-4.x.x/library 路径下。
你在自己电脑上搭建好 R 语言环境，并安装好了很多常用包，然后你想到一台没有 R 环境、没有联网的电脑上复现你的代码。
方法非常简单：你只需要在那台电脑安装相同版本的 R 软件，安装到相同路径下，将新的 library 文件夹完全替换为你电脑里的 library 文件夹即可[a]，这样运行起 R 代码与在你电脑没有任何区别。

[a]可以用添加压缩包再解压的方式，速度能快一些。

加载包

```
library(openxlsx)
```

更新包

```
update.packages("openxlsx")
update.packages()                    # 更新所有包
```

删除包

```
remove.packages("openxlsx")
```

获取或设置当前路径

```
getwd()
setwd("D:/R-4.1.1/tests")
```

特别注意：路径中的 \ 必须用 / 或 \\ 代替。

注：关于更新 R 版本和更新包，笔者一般是紧跟最新 R 版本，顺便重新安装一遍所有包。为了省事，笔者是将所有自己常用包的 `install.package("...")` 都放在一个 R 脚本中，改为国内镜像，全部选中运行即可。

赋值

R 标准语法中赋值不是用 = ，而是 <- 或 ->

```
x <- 1:10
x + 2
```

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

R 也允许用 = 赋值，建议用更现代和简洁的 = 赋值。

基本运算

- 数学运算：
  - `+ - * /`、`^` (求幂)、`%%` (按模求余[1])、`%/%` (整除)
- 比较运算
  - `>`、`<`、`>=`、`<=`、`==`、`!=`
  - `identical(x,y)` —— 判断两个对象是否严格相等；
  - `all.equal(x,y)` 或 `dplyr::near(x,y)` —— 判断两个浮点数是否近似相等 (误差 `1.5e-8`)

```
0L == 0
```

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

```
identical(0L, 0)
```

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

```
sqrt(2)^2 == 2
```

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

```
identical(sqrt(2)^2, 2)
```

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

```
all.equal(sqrt(2)^2, 2)
```

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

[1]可以关于小数按模求余，例如 `5.4 %% 2.3` 为 0.8

```
dplyr::near(sqrt(2)^2, 2)
```

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

- 逻辑运算：
  - | （或），& （与），! （非），xor() （异或）

> && 和 || 是短路运算，即遇到 TRUE （FALSE）则返回 TRUE （FALSE）而不继续往下计算；
> 而 & 和 | 是向量运算符，对向量中所有元素分别进行运算。

基本数据类型

- R 中的基本数据类型包括：
  - numeric —— 数值型，又分为 integer (整数型) 和 double (浮点型)
  - logical —— 逻辑型，只有 TRUE 和 FALSE，或 T 和 F
  - character —— 字符型，引号[2] 括起来的若干字符
- R 中用 NA 表示缺失值，NULL 表示空值，NaN 表示非数，Inf 表示无穷大
- 对于 R 中大多数函数，NA 具有传染性，即 NA 参与的运算，结果会变成 NA
- R 中注释一行代码用 #
- 可用函数 class(x) / typeof(x) / mode(x) 查看对象 x 的类型
  - 在展现数据的细节上，mode() < class() < typeof()
  - str(x) 显示对象 x 的结构

保存和载入数据

```
save(x, file = "datas/dat.Rda")
load("datas/dat.Rda")
```

关于相对路径与绝对路径

编程中的文件路径，可以用绝对路径也可以用相对路径。
绝对路径，是从盘符开始的完整路径：比如 E:/R 语言/datas/a123.csv。
相对路径，是指相对于当前路径的路径，因为通常操作的文件都是在当前路径，那么“从盘符到当前路径”这部分是大家所共有的，所以可以省略不写，只写从当前路径再往下的路径即可。比如，当前文件夹 E:/R 语言中有 datas 文件夹，里面有数据文件 a123.csv，要写能访问到它的路径，只需写 datas/a123.csv。

清屏和清除内存变量

Ctrl + L 或单击命令窗口右上角的**小刷子**可对命令窗口清屏。

若要清除当前变量，用：

[2] R 中单双引号通用。

```
rm(x)                          # 清除变量 x
rm(list = ls(all = TRUE))      # 清除所有当前变量
```

注：单击 Environment 窗口的**小刷子**也是清除所有当前变量。

获取帮助

编程语言最好的学习资料就是**帮助**。

- 函数帮助

命令窗口执行：

```
?plot
```

则在 help 窗口打开 `plot()` 函数的帮助: 包括函数来自哪个包、函数的描述、参数说明、更多解释、实例等。

- 在线帮助（需联网）

若想根据某算法的名字或关键词，搜索哪个包能实现该算法：

```
RSiteSearch("network")
```

注：很奇怪，现在只能查一个单词，在打开的网页，可以输入多个单词查询

- 其他主要网络资源
  R 官方镜像站（北外）

https://mirrors.bfsu.edu.cn/CRAN/

下的各种资源，建议自己去发掘。

比如，最常用的是包的帮助文档：在镜像站，点左侧的 Packagess，再点 sort by name，则出现所有可用的 CRAN 包列表。点击某个包名，则进入该包的介绍页：

图 1.6: tidyverse 包的官网介绍页

Reference manual 为参考手册，包含该包所有函数和自带数据集的说明，供查阅使用；Vignettes (若有)，是包的作者写的使用文档，它是该包的最佳学习资料。

在使用 R 语言过程中遇到各种问题，建议将报错信息设置为英文：`Sys.setenv(LANGUAGE = "en")`，建议用 `bing` 国际版搜索报错信息，更容易找到答案。另外，`Github` 是丰富的程序代码仓库，在 `bing` 搜索时，加上 `github` 关键词，可能有意想不到的收获。
其他开放的 R 社区：

- Stack overflow: https://stackoverflow.com/questions/tagged/r
- R-Bloggers: https://www.r-bloggers.com
- Tidyverse: https://www.tidyverse.org
- Rstudio: https://rstudio.com
- 统计之都：https://d.cosx.org

R Script 与 R Project

R 脚本是单个可执行的 R 代码文件，后缀名为.R，单击 `New File` 按钮，选择 R Script，或使用快捷键 `Ctrl + Shift + N`, 则新建 R 脚本。

R 脚本中都是可执行的 R 代码 + 注释，选中部分代码，点击 Run 运行选中的代码。

R 项目（Project）是完成某个项目或任务的一系列文件的合集 (文件夹)，包括数据文件、若干 R 脚本及其他附件，其中包含一个 `*.Rproj` 文件；

强烈建议使用 R 项目，它能方便系统地管理服务于共同目的一系列的文件，可以方便移动位置甚至是移到其他电脑，而不需要做任何路径设置就能成功运行。

创建 R 项目：单击 Create a Project 按钮

若在某个已存在的文件夹下创建项目，则选择 Existing Directory；若需要新建文件夹创建项目，则选择 New Directory。

创建完成后，在文件夹下出现一个 `*.Rproj` 文件，双击它（关联 RStudio 打开），则进入该 R 项目，做各种具体访问、编辑文件和运行脚本等操作。

Rmarkdown

后缀名为 `.Rmd` 的交互式文档，是 `markdown` 语法与 R 脚本的结合，可以将可执行 R 代码和不可执行的文字叙述，融为一个文件。

单击 New File 按钮，选择 R Markdown 创建 Rmarkdown，建议优先使用自带和来自网络的现成模板。

Rmarkdown 适合编写包含 R 语言代码的学习笔记、演示文档、论文、书籍等，可以生成 `docx`，`pptx`，`html`，`pdf` 等多种文档格式。更多 Rmarkdown 内容将在第五章展开讨论。

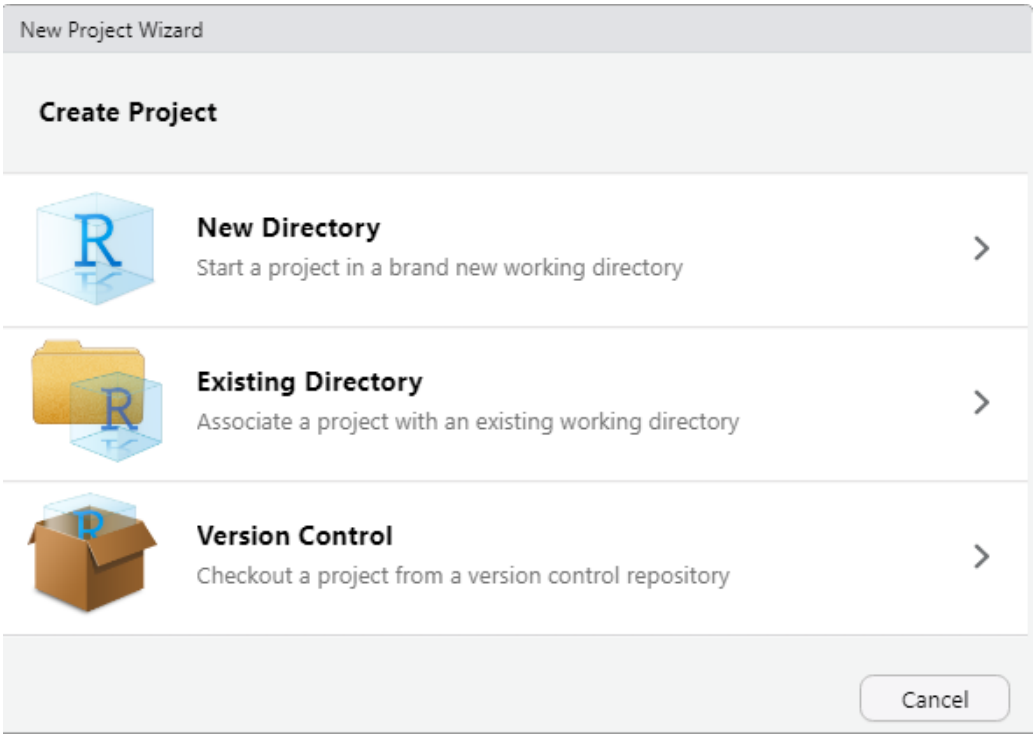

图 1.7: 创建 R Project

1.2 数据结构 imannumeral1}：向量、矩阵、多维数组

数据结构是为了便于存储不同类型的数据而设计的**数据容器**。学习数据结构，就是要把各个数据容器的特点、适合存取什么样的数据理解透彻，只有这样才能在实际中选择最佳的数据容器，数据容器选择的合适与否，直接关系到代码是否高效简洁，甚至能否解决问题。

R 中常用的数据结构可划分为：

- 同质数据类型 (homogeneous)，即所存储的一定是相同类型的元素，包括向量、矩阵、多维数组；
- 异质数据类型 (heterogeneous)，即可以存储不同类型的元素，这大大提高了存储的灵活性，但同时也降低了存储效率和运行效率，包括列表、数据框。

另外，还有字符串、日期时间数据、时间序列数据、空间地理数据等。

R 中的数据结构还有一种从**广义向量 (可称之为序列)**[3]的角度的划分：

- **原子向量**：各个值都是同类型的，包括 6 种类型：`logical`、`integer`、`double`、`character`、`complex`、`raw`，其中 `integer` 和 `double` 也统称为 `numeric`；
- **列表**：各个值可以是不同类型的，NULL 表示空向量 (长度为 0 的向量)

向量都有两个属性：`type` (类型)、`length` (长度)；还能以属性的方式向向量中任意添加额外的 `metadata` (元数据)，属性可用来创建扩展向量，以执行一些新的操作。常用的扩展向量有：

- 基于整数型向量构建的因子

[3]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。

- 基于数值型向量构建的日期和日期时间
- 基于数值型向量构建的时间序列
- 基于列表构建的数据框和 tibble

列表是序列，从这个角度有助于理解 `purrr::map_*()` 系列的泛函式编程。

1.2.1 向量 (一维数据)

向量是由一组相同类型的原始值构成的序列，可以是一组数值、一组逻辑值、一组字符串等。

常用的向量有：数值向量、逻辑向量、字符向量。

1. 数值向量

数值向量就是由数值组成的向量，单个数值是长度为 1 的数值向量

```
x = 1.5
x
```

```
## [1] 1.5
```

可以用 `numeric()` 来创建一个全为 0 的指定长度的数值向量：

```
numeric(10)
```

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

R 中经常用函数 `c()` 实现将多个对象合并到一起：

```
c(1, 2, 3, 4, 5)
```

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

```
c(1, 2, c(3, 4, 5))    # 将多个数值向量合并成一个数值向量
```

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

创建等差的数值向量，用 `:` 或者函数 `seq()`，基本格式为：

```
seq(from, to, by, length.out, along.with, ...)
```

`from`：设置首项 (默认为 1)；
`to`：设置尾项；
`by`：设置等差值 (默认为 1 或 -1)；
`length.out`：设置序列长度；
`along.with`：以该参数的长度作为序列长度。

```r
1:5                     # 同 seq(5) 或 seq(1,5)
```

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

```r
seq(1, 10, 2)           # 从 1 开始，到 10 结束，步长为 2
```

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

```r
seq(3, length.out=10)
```

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

创建重复的数值向量用函数 `rep()`，基本格式为：

```r
rep(x, times,length.out, each, ...)
```

`x`：为要重复的序列；
`times`：设置序列重复次数；
`length.out`：设置产生的序列的长度；
`each`：设置每个元素分别重复的次数 (默认为 1)。

```r
x = 1:3
rep(x, 2)
```

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

```r
rep(x, each = 2)
```

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

```r
rep(x, c(2, 1, 2))              # 按照规则重复序列 中的各元素
```

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

```r
rep(x, each = 2, length.out = 4)
```

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

```r
rep(x, each = 2, times = 3)
```

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

注意，R 中两个不同长度的向量做运算，短的会自动循环补齐以配合长的。

```r
2:3 + 1:5
```

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

2. 逻辑向量

逻辑向量，是一组逻辑值 (TRUE 或 FALSE, 或简写为 T 或 F) 的向量。

```
c(1, 2) > c(2, 1)          # 等价于 c(1 > 2, 2 > 1)
```

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

```
c(2, 3) > c(1, 2, -1, 3)  # 等价于 c(2 > 1, 3 > 2, 2 > -1, 3 > 3)
```

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

除了比较运算符外，还可以用`%in%`判断元素是否属于集合:

```
c(1, 4) %in% c(1, 2, 3)    # 左边向量每一个元素是否属于右边集合
```

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

3. 字符向量

字符 (串) 向量，是一组字符串组成的向量，R 中单引号和双引号都可以用来生成字符向量。

```
"hello, world!"
```

```
## [1] "hello, world!"
```

```
c("Hello", "World")
```

```
## [1] "Hello" "World"
```

```
c("Hello", "World") == "Hello, World"
```

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

要想字符串中出现单引号或双引号，可以单双引号错开，或者用转义符\来做转义，`writeLines()`函数输出纯字符串内容:

```
'Is "You" a Chinese name?'
```

```
# [1] "Is \"You\" a Chinese name?"
```

```
writeLines("Is \"You\" a Chinese name?")
```

```
# Is "You" a Chinese name?
```

> R 中还有不常用的复数向量、原 (raw) 向量。

4. 访问向量子集

即访问向量的一些特定元素或者某个子集。注意，R 中的索引是从 1 开始的。

使用元素的位置来访问：

```
v1 = c(1, 2, 3, 4)
v1[2]                # 第 2 个元素
v1[2:4]              # 第 2-4 个元素
v1[-3]               # 除了第 3 个之外的元素
```

也可以放任意位置的数值向量，但是注意不能既放正数又放负数：

```
v1[c(1,3)]
v1[c(1, 2, -3)]     # 报错
```

访问不存在的位置也是可以的，返回 NA：

```
v1[3:6]
```

使用逻辑向量来访问，输入与向量相同长度的逻辑向量，以此决定每一个元素是否要被获取：

```
v1[c(TRUE, FALSE, TRUE, FALSE)]
```

这可以引申为‘‘根据条件访问向量子集’’：

```
v1[v1 <= 2]        # 同 v1[which(v1 <= 2)] 或 subset(v1, v1<=2)
v1[v1 ^ 2 - v1 >= 2]
which.max(v1)      # 返回向量 v1 中最大值所在的位置
which.min(v1)      # 返回向量 v1 中最小值所在的位置
```

5. 对向量子集赋值，替换相应元素

对向量子集赋值，就是先访问到向量子集，再赋值。

```
v1[2] = 0
v1[2:4] = c(0, 1, 3)
v1[c(TRUE, FALSE, TRUE, FALSE)] = c(3, 2)
v1[v1 <= 2] = 0
```

注意，若对不存在的位置赋值，前面将用 NA 补齐：

```
v1[10] = 8
v1
```

6. 对向量元素命名

可以在创建向量的同时对其每个元素命名：

```
x = c(a = 1, b = 2, c = 3)
x
```

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

命名后，就可以通过名字来访问向量元素：

```
x[c("a", "c")]
x[c("a", "a", "c")]    # 重复访问也是可以的
x["d"]                 # 访问不存在的名字
```

获取向量元素的名字：

```
names(x)
```

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

更改向量元素的名字：

```
names(x) = c("x", "y", "z")
x["z"]
```

```
## z
## 3
```

移除向量元素的名字：

```
names(x) = NULL
x
```

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

[ ] 与 [[ ]] 的区别

[ ] 可以提取对象的子集，[[ ]] 可以提取对象中的元素。

二者的区别：以向量为例，可以将一个向量比作 10 盒糖果，你可以使用 [ ] 获取其中的 3 盒糖果，使用 [[ ]] 打开盒子并从中取出一颗糖果。

对于未对元素命名的向量，使用 [ ] 和 [[ ]] 取出一个元素会产生相同的结果。但对于已对元素命名的向量，二者会产生不同的结果：

```
x = c(a = 1, b = 2, c = 3)
x["a"]          # 取出标签为"a" 的糖果盒
```

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

```
x[["a"]]        # 取出标签为"a" 的糖果盒里的糖果
```

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

由于 [[ ]] 只能用于提取出一个元素，因此不适用于提取多个元素的情况，所以 [[ ]] 不能用于负整数，因为负整数意味着提取除特定位置之外的所有元素。

使用含有不存在的位置或名称来创建向量子集时将会产生缺失值。但当使用 [[ ]] 提取一个位置超出范围或者对应名称不存在的元素时，该命令将会无法运行并产生错误信息。

以下三个语句会报错：

```
x[[c(1, 2)]]
x[[-1]]
x[["d"]]
```

7. 对向量排序

向量排序函数 sort()，基本格式为：

```
sort(x, decreasing, na.last, ...)
```

- x：为排序对象 (数值型或字符型)；
- decreasing：默认为 FALSE 即升序，TRUE 为降序；
- na.last：默认为 FALSE，若为 TRUE，则将向量中的 NA 值放到序列末尾。

函数 order(), 返回元素排好序的索引，以其结果作为索引访问元素，正好是排好序的向量。

函数 rank()，返回值是该向量中对应元素的 “排名”。

```
x = c(1,5,8,2,9,7,4)
sort(x)
```

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

```
order(x)     # 默认升序，排名第 2 的元素在原向量的第 4 个位置
```

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

```
x[order(x)]  # 同 sort(x)
```

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

```
rank(x)      # 默认升序，第 2 个元素排名第 4 位
```

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

还有函数 rev(), 可将序列进行反转，即 1,2,3 变成 3,2,1。

1.2.2 矩阵 (二维数据)

矩阵是一个用两个维度表示和访问的向量。因此，适用于向量的性质和方法大多也适用于矩阵：矩阵也要求元素是同一类型，数值矩阵、逻辑矩阵等。

1. 创建矩阵

函数 matrix() 将一个向量创建为矩阵，其基本格式为：

```
matrix(x, nrow, ncol, byrow, dimnames, ...)
```

- x：为数据向量作为矩阵的元素；
- nrow：设定行数；
- ncol：设定列数；
- byrow：设置是否按行填充，默认为 FALSE (按列填充)；
- dimnames：用字符型向量表示矩阵的行名和列名。

```
matrix(c(1, 2, 3,
         4, 5, 6,
         7, 8, 9), nrow = 3, byrow = FALSE)
```

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

```
matrix(c(1, 2, 3,
         4, 5, 6,
         7, 8, 9), nrow = 3, byrow = TRUE)
```

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

对矩阵的行列命名：

```
matrix(1:9, nrow = 3, byrow = TRUE,
       dimnames = list(c("r1","r2","r3"), c("c1","c2","c3")))
```

```
##    c1 c2 c3
## r1  1  2  3
## r2  4  5  6
## r3  7  8  9
```

也可以创建后再命名：

```
m1 = matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), ncol = 3)
rownames(m1) = c("r1", "r2", "r3")
colnames(m1) = c("c1", "c2", "c3")
```

特殊矩阵：

```
diag(1:4, nrow = 4)      # 对角矩阵
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    2    0    0
## [3,]    0    0    3    0
## [4,]    0    0    0    4
```

函数 as.vector(), 可将矩阵转化为向量，元素按列读取。

2. 访问矩阵子集

矩阵是用两个维度表示和访问的向量，可以用一个二维存取器 [,] 来访问，这类似于构建向量子集时用的一维存取器 []。

可以为每个维度提供一个向量来确定一个矩阵的子集。方括号中的第 1 个参数是行选择器，第 2 个参数是列选择器。与构建向量子集一样，可以在两个维度中使用数值向量、逻辑向量和字符向量。

```
m1[1,2]              # 提取第 1 行，第 2 列的单个元素
m1[1:2, 2:4]         # 提取第 1 至 2 行，第 2 至 4 列的元素
m1[c("r1","r3"), c("c1","c3")]  # 提取行名为 r1 和 r3，列名为 c1 和 c3 的元素
```

若一个维度空缺，则选出该维度的所有元素：

```
m1[1,]              # 提取第 1 行，所有列元素
m1[,2:4]            # 提取所有行，第 2 至 4 列的元素
```

负数表示在构建矩阵子集时可排除该位置，这和向量中的用法一致：

```
m1[-1,]             # 提取除了第 1 行之外的所有元素
m1[,-c(2,4)]        # 提取除了第 2 和 4 列之外的所有元素
```

注意，矩阵是一个用两个维度表示和访问的向量，但它本质上仍然是一个向量。因此，向量的一维存取器也可以用来构建矩阵子集：

```
m1[3:7]
```

```
## [1] 3 4 5 6 7
```

由于向量只包含相同类型的元素，矩阵也是如此。所以它们的操作方式也相似。若输入一个不等式，则返回同样大小的逻辑矩阵：

```
m1 > 3
```

```
##       c1   c2   c3
## r1 FALSE TRUE TRUE
## r2 FALSE TRUE TRUE
## r3 FALSE TRUE TRUE
```

根据它就可以选择矩阵元素或赋值：

```
m1[m1 > 3]    # 注意选出来的结果是向量
```

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

矩阵运算

- `A+B, A-B, A*B, A/B`：矩阵四则运算，要求矩阵同型，类似 Matlab 中的点运算，分别将对应位置的元素做四则运行；
- `A %*% B`：矩阵乘法，要求 A 的列数 = B 的行数。

1.2.3 多维数组 (多维数据)

向量/矩阵向更高维度的自然推广。具体来说，多维数组就是一个维度更高 (通常大于 2) 、可访问的向量。数组也要求元素是同一类型。

1. 创建多维数组

函数 `array()` 将一个向量创建为多维数组，基本格式为：

```
array(x, dim, dimnames, ...)
```

- `x`：为数据向量作为多维数组的元素；
- `dim`：设置多维数组各维度的维数；
- `dimnames`：设置多维数组各维度的名称。

```
a1 = array(1:24, dim = c(3, 4, 2))
a1
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]   13   16   19   22
## [2,]   14   17   20   23
## [3,]   15   18   21   24
```

也可以在创建数组时对每个维度进行命名：

```
a1 = array(1:24, dim = c(3, 4, 2),
           dimnames=list(c("r1","r2","r3"),
                         c("c1","c2","c3","c4"), c("k1","k2")))
```

或者创建之后再命名[4]

```
a1 = array(1:24, dim = c(3, 4, 2))
dimnames(a1) = list(c("r1","r2","r3"),
                    c("c1","c2","c3","c4"), c("k1","k2"))
```

2. 访问多维数组子集

第 3 个维度姑且称为“页”

```
a1[2,4,2]           # 提取第 2 行，第 4 列，第 2 页的元素
a1["r2","c4","k2"]  # 提取第 r2 行，第 c4 列，第 k2 页的元素
a1[1,2:4,1:2]       # 提取第 1 行，第 2 至 4 列，第 1 至 2 页的元素
a1[,,2]             # 提取第 2 页的所有元素
dim(a1)             # 返回多维数组 a1 的各维度的维数
```

> 在想象多维数组时，为了便于形象地理解，可以将其维度依次想象为与“书”相关的概念：行、列、页、本、层、架、室.....

[4] `list` 是创建列表 (见下节) 。

本节部分内容参阅 (任坤 2017) 和 (G. G. Hadley Wickham 2017).

1.3 数据结构 iimannumeral2}：列表、数据框、因子

1.3.1 列表 (list)

列表，可以包含不同类型的对象，甚至可以包括其他列表。列表的灵活性使得它非常有用。

例如，用 R 拟合一个线性回归模型，其返回结果就是一个列表，其中包含了线性回归的详细结果，如线性回归系数 (数值向量) 、残差 (数值向量) 、QR 分解 (包含一个矩阵和其他对象的列表) 等。因为这些结果全都被打包到一个列表中，就可以很方便地提取所需信息，而无需每次调用不同的函数。

列表最大的好处就是，它能够将多个不同类型的对象打包到一起，使得可以根据位置和名字访问它们。

1. 创建列表

可以用函数 `list()` 创建列表。不同类型的对象可以放入同一个列表中。

例如，创建了一个列表，包含 3 个成分：一个单元素的数值向量、一个两元素的逻辑向量和一个长度为 3 的字符向量：

```
l0 = list(1, c(TRUE, FALSE), c("a", "b", "c"))
l0
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1]  TRUE FALSE
##
## [[3]]
## [1] "a" "b" "c"
```

可以在创建列表时，为列表的每个成分指定名字：

```
l1 = list(A = 1, B = c(TRUE, FALSE), C = c("a", "b", "c"))
l1
```

```
## $A
## [1] 1
##
## $B
## [1]  TRUE FALSE
##
```

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

也可以创建列表后再对列表成分命名或修改名字：

```
names(l1) = NULL       # 移除列表成分的名字
names(l1) = c("x","y","z")
```

2. 从列表中提取成分的内容

提取列表中成分下的内容，最常用的方法是用 $，通过成分名字来提取该成分下的内容：

```
l1$y
l1$m                   # 访问不存在的成分 m，将会返回 NULL
```

也可以用 [[n]] 来提取列表第 n 个成分的内容，n 也可以换成成分的名字：

```
l1[[2]]                # 同 l1[["y"]]
```

用 [[]] 提取列表中某个成分的内容更加灵活，可用在函数调用中，通过参数来传递成分名字：

```
p = "y"                # 想要提取其内容的成分名字
l1[[p]]
```

3. 提取列表子集

经常也需要从列表中提取多个成分及其内容，由这些成分组成的列表构成了原列表的一个子集。

就像提取向量和矩阵的子集一样，提取一个列表子集是用 []，可以取出列表中的一些成分，作为一个新的列表。

[] 中可以用字符向量表示成分名字，用数值向量表示成分位置，或用逻辑向量指定是否选择，来取出列表成分。

```
l1["x"]                    # 同 l1[1]
l1[c("x", "z")]            # 同 l1[c(1, 3)], l1[c(TRUE, FALSE, TRUE)]
```

> 用 [] 提取若干成分时，返回列表的子集，还是一个列表；用 [[ ]] 提取单个成分的元素，返回的是对应成分的元素。
> 总之，[] 提取对象的子集，类型仍是该对象；[[]] 提取对象的内容 (下一级元素) 。

4. 对列表的成分赋值

即先访问 (提取) 到列表的成分，再赋以相应的值。注意，若给一个不存在的成分赋值，列表会自动地在对应名称或位置下增加一个新成分。

```
l1$x = 0   # 将列表的成分 x 赋值为 0
```

也可以同时给多个列表成分赋值：

```
l1[c("x", "y")] = list(x = "new value for y", y = c(3, 1))
```

若要移除列表中的某些成分，只需赋值为 NULL：

```
l1[c("z", "m")] = NULL
```

5. 列表函数

用函数 as.list() 可将向量转换成列表：

```
l2 = as.list(c(a = 1, b = 2))
l2
```

```
## $a
## [1] 1
##
## $b
## [1] 2
```

用去列表化函数 unlist()，可将一个列表打破成分界线，强制转换成一个向量[5]：

```
unlist(l2)
```

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

tidyverse 系列中的 purrr 包为方便操作列表，提供了一系列列表相关的函数，建议读者查阅使用：

- pluck()：同 [[提取列表中的元素
- keep()：保留满足条件的元素
- discard()：删除满足条件的元素
- compact()：删除列表中的空元素
- append()：在列表末尾增加元素
- flatten()：摊平列表（只摊平一层）

1.3.2 数据框　数据表）

R 语言中做统计分析的样本数据，都是按数据框类型操作的。

数据框是指有若干行和列的数据集，它与矩阵类似，但并不要求所有列都是相同的类型；本质上

[5]若列表的成分具有不同类型，则自动向下兼容到统一类型。

讲，数据框就是一个列表，它的每个成分都是一个向量，并且长度相同，以表格的形式展现。总之，**数据框是由列向量组成、有着矩阵形式的列表**。

数据框与最常见的数据表是一致的：每一列代表一个变量属性，每一行代表一条样本数据：

```
knitr::kable(persons, caption = " 数据表示例", booktabs = TRUE)
```

表 1.1: 数据表示例

Name	Gender	Age	Major
Ken	Male	24	Finance
Ashley	Female	25	Statistics
Jennifer	Female	23	Computer Science

R 中自带的数据框是 `data.frame`，建议改用更现代的数据框：`tibble`[6]。

Hadley 在 `tibble` 包中引入一种 `tibble` 数据框，以代替 `data.frame`；而且 `tidyverse` 包都是基于 `tibble` 数据框。

`tibble` 对比 `data.frame` 的优势：

- `tibble()` 比 `data.frame()` 做的更少：不改变输入变量的类型 (**R 4.0.0 之前默认将字符串转化为因子！**)，不会改变变量名，不会创建行名；
- `tibble` 对象的列名可以是 R 中的 ‘‘非法名’’：非字母开头、包含空格，但定义和使用变量时都需要用倒引号 ‘ 括起来；
- `tibble` 在输出时不自动显示所有行，避免大数据框时显示很多内容；
- 用 `[]` 选取列子集时，即使只选取一列，返回结果仍是 `tibble`，而不自动简化为向量。

1. 创建数据框

用 `tibble()` 根据若干列向量创建 `tibble`：

```
library(tidyverse)        # 或 tibble
persons = tibble(
  Name = c("Ken", "Ashley", "Jennifer"),
  Gender = c("Male", "Female", "Female"),
  Age = c(24, 25, 23),
  Major = c("Finance", "Statistics", "Computer Science"))
persons
```

```
## # A tibble: 3 x 4
##   Name     Gender   Age Major
##   <chr>    <chr>  <dbl> <chr>
```

[6]读者若习惯用 R 自带的 `data.frame`, 只需要换个名字，将 `tibble` 改为 `data.frame` 即可。

```
## 1 Ken      Male      24 Finance
## 2 Ashley   Female    25 Statistics
## 3 Jennifer Female    23 Computer Science
```

用 `tribble()` 按行录入数据式创建 `tibble`：

```
tribble(
  ~Name, ~Gender, ~Age, ~Major,
  "Ken", "Male", 24, "Finance",
  "Ashley", "Female", 25, "Statistics",
  "Jennifer", "Female", 23, "Computer Science")
```

用 `as_tibble()` 将 `data.frame`, `matrix`, 各成分等长度的 `list`, 转换为 `tibble`。

对于不等长的列表转化为数据框：

```
a = list(A = c(1, 3, 4), B = letters[1:4])
a
```

```
## $A
## [1] 1 3 4
##
## $B
## [1] "a" "b" "c" "d"
```

```
# lengths() 获取 list 中每个元的长度
map_dfc(a, `length<-`, max(lengths(a)))   # map 循环参阅 1.6.2 节
```

```
## # A tibble: 4 x 2
##       A B
##   <dbl> <chr>
## 1     1 a
## 2     3 b
## 3     4 c
## 4    NA d
```

数据框既是列表的特例，也是矩阵的推广，因此访问这两类对象的方式都适用于数据框。例如与矩阵类似，对数据框的行列重新命名：

```
df = tibble(id = 1:4,
            level = c(0, 2, 1, -1),
            score = c(0.5, 0.2, 0.1, 0.5))
names(df) = c("id", "x", "y")
df
```

```
## # A tibble: 4 x 3
##      id     x     y
##   <int> <dbl> <dbl>
## 1     1     0   0.5
## 2     2     2   0.2
## 3     3     1   0.1
## 4     4    -1   0.5
```

2. 提取数据框的元素、子集

数据框是由列向量组成、有着矩阵形式的列表，所以可以用两种操作方式来访问数据框的元素和子集。

(1) 以列表方式提取数据框的元素、子集

若把数据框看作是由向量组成的列表，则可以沿用列表的操作方式来提取元素或构建子集。例如，可以用 `$` 按列名来提取某一列的值，或者用 `[[]]` 按照位置或列名提取。

例如，提取列名为 `x` 列的值，得到向量：

```
df$x                # 同 df[["x"]], df[[2]]
```

```
## [1]  0  2  1 -1
```

以列表形式构建子集完全适用于数据框，同时也会生成一个新的数据框。提取子集的操作符 `[]` 允许用数值向量表示列的位置，用字符向量表示列名，或用逻辑向量指定是否选择。

例如，提取数据框的一列或多列，得到子数据框：

```
df[1]               # 提取第 1 列，同 df["id"]
```

```
## # A tibble: 4 x 1
##      id
##   <int>
## 1     1
## 2     2
## 3     3
## 4     4
```

```
df[1:2]             # 同 df[c("id","x")], df[c(TRUE,TRUE,FALSE)]
```

```
## # A tibble: 4 x 2
##      id     x
##   <int> <dbl>
## 1     1     0
## 2     2     2
```

```
## 3     3     1
## 4     4    -1
```

(2) 以矩阵方式提取数据框的元素、子集

以列表形式操作并不支持行选择。以矩阵形式操作更加灵活，若将数据框看作矩阵，其二维形式的存取器可以很容易地获取一个子集的元素，同时支持列选择和行选择。

换句话说，可以使用 `[i, j]` 指定行或列来提取数据框子集，`[ , ]` 内可以是数值向量、字符向量或者逻辑向量。

若行选择器为空，则只选择列 (所有行)：

```
df[, "x"]
```

```
## # A tibble: 4 x 1
##       x
##   <dbl>
## 1     0
## 2     2
## 3     1
## 4    -1
```

```
df[, c("x","y")]   # 同 df[,2:3]
```

```
## # A tibble: 4 x 2
##       x     y
##   <dbl> <dbl>
## 1     0   0.5
## 2     2   0.2
## 3     1   0.1
## 4    -1   0.5
```

若列选择器为空，则只选择行 (所有列)：

```
df[c(1,3),]
```

```
## # A tibble: 2 x 3
##      id     x     y
##   <int> <dbl> <dbl>
## 1     1     0   0.5
## 2     3     1   0.1
```

同时选择行和列：

```
df[1:3, c("id","y")]
```

```
## # A tibble: 3 x 2
##      id     y
##   <int> <dbl>
## 1     1   0.5
## 2     2   0.2
## 3     3   0.1
```

根据条件筛选数据。例如用 `y >= 0.5` 筛选 `df` 的行，并选择 `id` 和 `y` 两列：

```
df[df$y >= 0.5, c("id","y")]
```

```
## # A tibble: 2 x 2
##      id     y
##   <int> <dbl>
## 1     1   0.5
## 2     4   0.5
```

按列名属于集合 `{x, y, w}` 来筛选 `df` 的列，并选择前两行：

```
ind = names(df) %in% c("x","y","w")
df[1:2, ind]
```

```
## # A tibble: 2 x 2
##       x     y
##   <dbl> <dbl>
## 1     0   0.5
## 2     2   0.2
```

3. 给数据框赋值

给数据框赋值，就是选择要赋值的位置，再准备好同样大小且格式匹配的数据，赋值给那些位置即可，所以同样有列表方式和矩阵方式。

(1) 以列表方式给数据框赋值

用 `$` 或 `[[ ]]` 对数据框的 1 列赋值

```
df$y = c(0.6,0.3,0.2,0.4)    # 同 df[["y"]] = c(0.6,0.3,0.2,0.4)
```

利用现有列，创建 (计算) 新列：

```
df$z = df$x + df$y
df
```

```
## # A tibble: 4 x 4
```

```
##      id     x     y     z
##   <int> <dbl> <dbl> <dbl>
## 1     1     0   0.5   0.5
## 2     2     2   0.2   2.2
## 3     3     1   0.1   1.1
## 4     4    -1   0.5  -0.5
```

```
df$z = as.character(df$z)   # 转换列的类型
df
```

```
## # A tibble: 4 x 4
##      id     x     y z
##   <int> <dbl> <dbl> <chr>
## 1     1     0   0.5 0.5
## 2     2     2   0.2 2.2
## 3     3     1   0.1 1.1
## 4     4    -1   0.5 -0.5
```

用 [] 可以对数据框的 1 列或多列进行赋值：

```
df["y"] = c(0.8,0.5,0.2,0.4)
df[c("x", "y")] = list(level = c(1,2,1,0),
                       score = c(0.1,0.2,0.3,0.4))
```

(2) 以矩阵方式给数据框赋值

以列表方式对数据框进行赋值时，也是只能访问列。若需要更加灵活地进行赋值操作，可以以矩阵方式进行。

```
df[1:3,"y"] = c(-1,0,1)
df[1:2,c("x","y")] = list(level = c(0,0),
                          score = c(0.9,1.0))
```

4. 一些有用函数

函数 str() 或 glimpse() 作用在 R 对象上，显示该对象的结构：

```
str(persons)
```

```
## tibble [3 x 4] (S3: tbl_df/tbl/data.frame)
##  $ Name  : chr [1:3] "Ken" "Ashley" "Jennifer"
##  $ Gender: chr [1:3] "Male" "Female" "Female"
##  $ Age   : num [1:3] 24 25 23
##  $ Major : chr [1:3] "Finance" "Statistics" "Computer Science"
```

summary() 作用在数据框/列表上，将生成各列/成分的汇总信息：

```
summary(persons)
```

```
##      Name               Gender               Age
##  Length:3           Length:3           Min.   :23.0
##  Class :character   Class :character   1st Qu.:23.5
##  Mode  :character   Mode  :character   Median :24.0
##                                        Mean   :24.0
##                                        3rd Qu.:24.5
##                                        Max.   :25.0
##     Major
##  Length:3
##  Class :character
##  Mode  :character
##
##
##
```

经常需要将多个数据框 (或矩阵) 按行或按列进行合并。用函数 rbind()，增加行 (样本数据)，要求宽度 (列数) 相同；用函数 cbind()，增加列 (属性变量)，要求高度 (行数) 相同。

例如，向数据框 persons 数据框中添加一个人的新记录：

```
rbind(persons,
      tibble(Name = "John", Gender = "Male",
             Age = 25, Major = "Statistics"))
```

```
## # A tibble: 4 x 4
##   Name     Gender   Age Major
##   <chr>    <chr>  <dbl> <chr>
## 1 Ken      Male      24 Finance
## 2 Ashley   Female    25 Statistics
## 3 Jennifer Female    23 Computer Science
## 4 John     Male      25 Statistics
```

向 persons 数据框中添加两个新列表示每个人是否已注册和其手头的项目数量：

```
cbind(persons, Registered = c(TRUE, TRUE, FALSE),
               Projects = c(3, 2, 3))
```

```
##       Name Gender Age            Major Registered Projects
## 1      Ken   Male  24          Finance       TRUE        3
## 2   Ashley Female  25       Statistics       TRUE        2
```

```
## 3 Jennifer Female  23 Computer Science      FALSE        3
```

> rbind() 和 cbind() 不会修改原始数据，而是生成一个添加了行或列的新数据框。

函数 expand.grid() 可生成多个属性水平值所有组合 (笛卡儿积) 的数据框：

```
expand.grid(type = c("A","B"), class = c("M","L","XL"))
```

```
##   type class
## 1    A     M
## 2    B     M
## 3    A     L
## 4    B     L
## 5    A    XL
## 6    B    XL
```

1.3.3 因子 (factor)

数据（变量）可划分为：定量数据（数值型）、定性数据（分类型），定性数据又分为名义型（无好坏顺序之分，如性别）、有序型（有好坏顺序之分，如疗效）。

R 提供了因子这一数据结构（容器），专门用来存放名义型和有序型的分类变量。因子本质上是一个带有水平 (level) 属性的整数向量，其中 ‘‘水平’’ 是指事前确定可能取值的有限集合。例如，性别有两个水平：男、女。

直接用字符向量也可以表示分类变量，但它只有字母顺序，不能规定想要的顺序，也不能表达有序分类变量。所以，有必要把字符型的分类变量转化为因子型，这更便于对其做后续描述汇总、可视化、建模等。

1. 创建与使用因子

函数 factor() 用来创建因子，基本格式为：

```
factor(x, levels, labels, ordered, ...)
```

- x：为创建因子的数据向量；
- levels：指定因子的各水平值，默认为 x 中不重复的所有值；
- labels：设置各水平名称 (前缀)，与水平一一对应；
- ordered：设置是否对因子水平排序，默认 FALSE 为无序因子, TRUE 为有序因子；

该函数还包含参数 exclude：指定有哪些水平是不需要的 (设为 NA)；nmax 设定水平数的上限。若不指定参数 levels，则因子水平默认按字母顺序。

比如，现有 6 个人的按等级的成绩数据，先以字符向量创建，并对其排序：

```
x = c(" 优", " 中", " 良", " 优", " 良", " 良")      # 字符向量
x
```

```
## [1] "优" "中" "良" "优" "良" "良"
```

```
sort(x)                                          # 排序是按字母顺序
```

```
## [1] "良" "良" "良" "优" "优" "中"
```

它的顺序只能是字母顺序，如果想规定顺序：中、良、优，正确的做法就是创建成因子，用 `levels` 指定想要的顺序：

```
x1 = factor(x, levels = c(" 中", " 良", " 优"))  # 转化因子型
x1
```

```
## [1] 优 中 良 优 良 良
## Levels: 中 良 优
```

```
as.numeric(x1)                                   # x 的存储形式：整数向量
```

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

> 注意：不能直接将因子数据当字符型操作，需要用 `as.character()` 转化。

转化为因子型后，数据向量显示出来（外在表现）与原来是一样的，但内在存储已经变了。因子型是以整数向量存储的，将各水平值按照规定的顺序分别对应到整数，将原向量的各值分别用相应的整数存储，输出和使用的时候再换回对应的水平值。整数是有顺序的，这样就相当于在不改变原数据的前提下规定了顺序，同时也节省了存储空间。

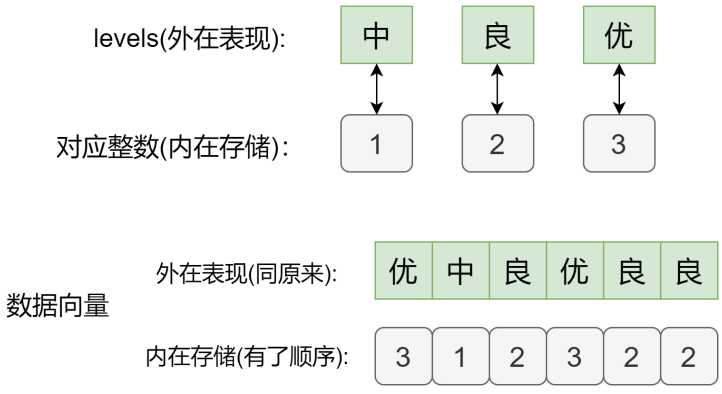

图 1.8: 因子的内在存储与外在表现

变成因子型后，无论是排序、统计频数、绘图等，都有了顺序：

```
sort(x1)
```

```
## [1] 中 良 良 良 优 优
```

```
## Levels: 中 良 优
```

```
table(x1)
```

```
## x1
## 中 良 优
##  1  3  2
```

```
ggplot(tibble(x1), aes(x1)) +
  geom_bar()
```

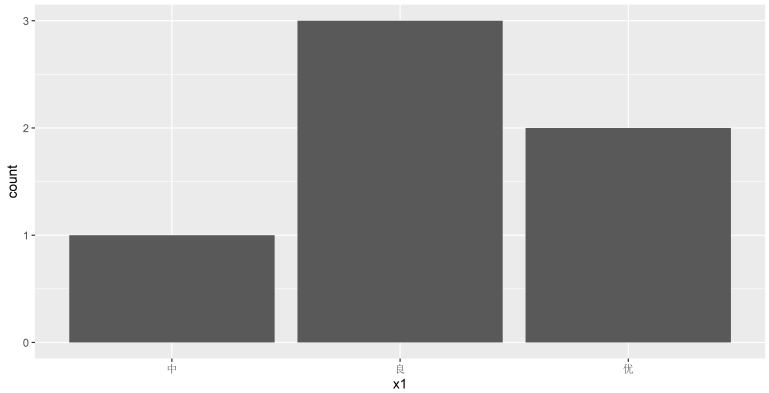

图 1.9: 用因子控制条形顺序

用 levels() 函数可以访问或修改因子的水平值（将改变数据的外在表现）：

```
levels(x1) = c("Excellent", "Good", "Fair")    # 修改因子水平
x1
```

```
## [1] Fair      Excellent Good      Fair      Good      Good
## Levels: Excellent Good Fair
```

有时候你可能更希望让水平的顺序与其在数据集中首次出现的次序相匹配，设置参数 levels = unique(x).

转化为因子型的另一个好处是，可以'' 识错''：因子数据只认识出现在水平值中的值，否则将识别为 NA.

很多人将因子固有的顺序与有序因子混淆，二者不是一回事：上述反复提到的顺序，可称为因子固有的顺序，正是有了它，才能方便地按想要的顺序排序、统计频数、绘图等；而无序因子与有序因子，是与变量本身的数据类型相对应的，名义型（无顺序好坏之分的分类变量）用无序因子存放，有序型（有顺序好坏之分的分类变量）用有序因子存放，该区分是用于不同类型的数据，建模时适用不同的模型。

示例的成绩数据是有好坏之分的，创建为有序因子：

```
x2 = factor(x, levels = c(" 中", " 良", " 优"), ordered = TRUE)
x2
```

```
## [1] 优 中 良 优 良 良
## Levels: 中 < 良 < 优
```

如果对 x2 做排序、统计频数、绘图，你会发现与无序因子时没有任何区别。它们的区别体现在对其建模时适用的模型不同。

2. 有用函数

函数 table()，可以统计因子各水平的出现次数 (频数)，也可以统计向量中每个不同元素的出现次数，返回结果为命名向量。

```
table(x)
```

```
## x
## 良 优 中
##  3  2  1
```

函数 cut()，用来做连续变量离散化：将数值向量切分为若干区间段，返回因子。基本格式为：

```
cut(x, breaks, labels, ...)
```

- x：为要切分的数值向量；
- breaks：切分的界限值构成的向量，或表示切分段数的整数。

该函数还包含参数 right 设置区间段是否左开右闭，include.lowest 设置是否包含下界，ordered_result 设置是否对结果因子排序。

```
Age = c(23,15,36,47,65,53)
cut(Age, breaks = c(0,18,45,100),
    labels = c("Young","Middle","Old"))
```

```
## [1] Middle Young  Middle Old    Old    Old
## Levels: Young Middle Old
```

函数 gl() 用来生成有规律的水平值组合因子。对于多因素试验设计，用该函数可以生成多个因素完全组合，基本格式为：

```
gl(n, k, length, labels, ordered, ...)
```

- n：为因子水平个数；
- k：为同一因子水平连续重复次数；
- length：为总的元素个数，默认为 n * k, 若不够则自动重复；
- labels：设置因子水平值；
- ordered：设置是否为有序，默认为 FALSE。

```
tibble(
  Sex = gl(2, 3, length = 12, labels = c(" 男"," 女")),
```

```
  Class = gl(3, 2, length = 12, labels = c(" 甲"," 乙"," 丙")),
  Score = gl(4, 3, length = 12, labels = c(" 优"," 良"," 中", " 及格")))
```

```
## # A tibble: 12 x 3
##   Sex   Class Score
##   <fct> <fct> <fct>
## 1 男    甲    优
## 2 男    甲    优
## 3 男    乙    优
## 4 女    乙    良
## 5 女    丙    良
## 6 女    丙    良
## # ... with 6 more rows
```

3. forcats 包

tidyverse 系列中的 forcats 包是专门为处理因子型数据而设计的，提供了一系列操作因子的方便函数：

- as_factor(): 转化为因子，默认按水平值的出现顺序
- fct_count(): 计算因子各水平频数、占比，可按频数排序
- fct_c(): 合并多个因子的水平
- 改变因子水平的顺序：
  - fct_relevel(): 手动对水平值重新排序
  - fct_infreq(): 按高频优先排序
  - fct_inorder(): 按水平值出现的顺序
  - fct_rev(): 将顺序反转
  - fct_reorder(): 根据其他变量或函数结果排序 (绘图时有用)

ggplot(data, aes(fct_reorder(x1,-y1), y1))

- 修改水平：
  - fct_recode(): 对水平值逐个重编码
  - fct_collapse(): 推倒手动合并部分水平
  - fct_lump_*(): 将多个频数小的水平合并为其他
  - fct_other(): 将保留之外或丢弃的水平合并为其他
- 增加或删除水平：
  - fct_drop(): 删除若干水平
  - fct_expand: 增加若干水平
  - fct_explicit_na(): 为 NA 设置水平

读者需要明白这样一个基本逻辑：操作因子是操作一个向量，该向量更多的时候是以数据框的一列的形式存在的。所以，来演示一下更常用的操作数据框中的因子列。涉及数据操作和绘图的语法，在第 2、3 章才会讲到。这里只需要知道大意和理解因子操作部分即可。

mpg 是汽车数据集，class 列是分类变量车型，先统计各种车型的频数，共有 7 类；对该列做因子合并，合并为 5 类 +Other 类再统计频数，频数少的类合并为 Other 类：

```
count(mpg, class)
```

```
## # A tibble: 7 x 2
##   class          n
##   <chr>      <int>
## 1 2seater        5
## 2 compact       47
## 3 midsize       41
## 4 minivan       11
## 5 pickup        33
## 6 subcompact    35
## # ... with 1 more row
```

```
mpg1 = mpg %>%
  mutate(class = fct_lump(class, n = 5))
count(mpg1, class)
```

```
## # A tibble: 6 x 2
##   class          n
##   <fct>      <int>
## 1 compact       47
## 2 midsize       41
## 3 pickup        33
## 4 subcompact    35
## 5 suv           62
## 6 Other         16
```

若直接对 class 各类绘制条形图，是按水平顺序，频数会参差不齐；改用根据频数多少排序，则条形图变的整齐易读：

```
p1 = ggplot(mpg, aes(class)) +
  geom_bar()
p2 = ggplot(mpg, aes(fct_infreq(class))) +
  geom_bar()
library(patchwork)
p1 | p2
```

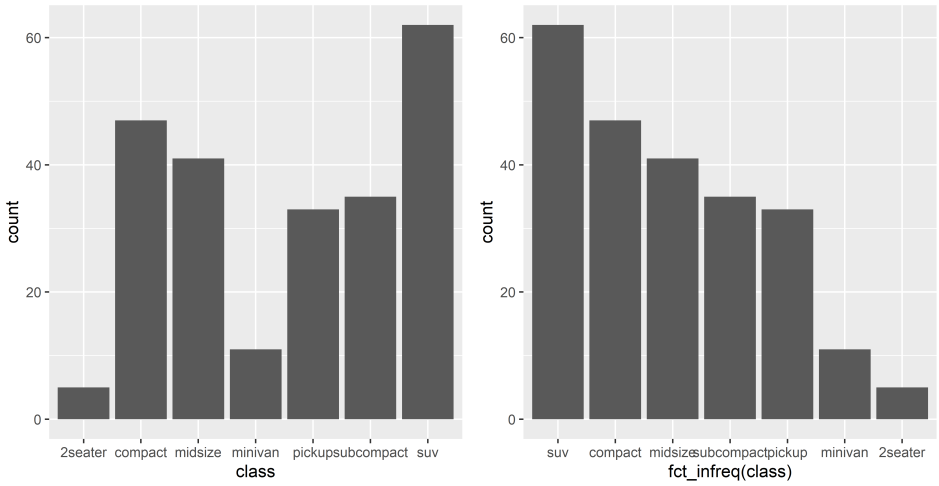

图 1.10: 按频数排序的条形图

本节部分内容参阅 (G. G. Hadley Wickham 2017), (任坤 2017), RStudio Cheatsheets: Factors with forcats, `forcats` 包文档。

1.4 数据结构 iiimannumeral3}：字符串、日期时间

1.4.1 字符串

字符串是用双引号或单引号括起来的若干字符，建议用双引号，除非字符串中包含双引号。字符串构成的向量，简称为字符向量。

字符串处理不是 R 语言中的主要功能，但也是必不可少的，数据清洗、可视化等操作都会用到。

`tidyverse` 系列中的 `stringr` 包提供了一系列接口一致的、简单易用的字符串操作函数，足以代替 R 自带字符串函数。这些函数都是向量化的，即作用在字符向量上，对字符向量中的每个字符串做某种操作。

```
library(stringr)
```

1. 字符串的长度 (包含字符个数)

```
str_length(c("a", "R for data science", NA))
```

```
## [1]  1 18 NA
```

```
str_pad(c("a", "ab", "abc"), 3)       # 填充到长度为 3
```

```
## [1] "  a" " ab" "abc"
```

```
str_trunc("R for data science", 10)   # 截断到长度为 10
```

```
## [1] "R for d..."
```

```
str_trim(c("a  ", "b  ", "a b"))      # 移除空格
```

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

后三个函数都包含参数 side=c("both", "left", "right") 设定操作的方向。

2. 字符串合并

```
str_c(..., sep = "", collapse = NULL)
```

- sep：设置间隔符，默认为空字符；-collapse：指定间隔符，将字符向量推倒合并为一个字符串。

```
str_c("x", 1:3, sep = "")  # 同 paste0("x", 1:3), paste("x", 1:3, sep="")
```

```
## [1] "x1" "x2" "x3"
```

```
str_c("x", 1:3, collapse = "_")
```

```
## [1] "x1_x2_x3"
```

注： 1:3 自动向下兼容以适应字符串运算，效果同 c("1","2","3")

将字符串重复 n 次，基本格式为：

```
str_dup(string, times)
```

- string：为要重复的字符向量；
- times：为重复的次数。

```
str_dup(c("A","B"), 3)
```

```
## [1] "AAA" "BBB"
```

```
str_dup(c("A","B"), c(3,2))
```

```
## [1] "AAA" "BB"
```

3. 字符串拆分

```
str_split(string, pattern)           # 返回列表
str_split_fixed(string, pattern, n) # 返回矩阵，n 控制返回的列数
```

- string：为要拆分的字符串；
- pattern：指定拆分的分隔符，可以是正则表达式。

```
x = "10,8,7"
str_split(x, ",")
```

```
## [[1]]
## [1] "10" "8"  "7"
```

```
str_split_fixed(x, ",", n = 2)
```

```
##      [,1] [,2]
## [1,] "10" "8,7"
```

4. 字符串格式化输出

只要在字符串内用 {变量名}，则函数 str_glue() 和 str_glue_data 就可以将字符串中的变量名替换成变量值，后者的参数 .x 支持引入数据框、列表等。

```
str_glue("Pi = {pi}")
```

```
## Pi = 3.14159265358979
```

```
name = " 李明"
tele = "13912345678"
str_glue(" 姓名：{name}", " 电话号码：{tele}", .sep="；")
```

```
## 姓名： 李明；电话号码：13912345678
```

5. 字符串排序

```
str_sort(x, decreasing, locale, ...)
str_order(x, decreasing, locale, ...)
```

默认 decreasing = FALSE 表示升序，前者返回排好序的元素，后者返回排好序的索引；参数 locale 可设定语言，默认为 "en" 英语。

```
x = c("banana", "apple", "pear")
str_sort(x)
```

```
## [1] "apple"  "banana" "pear"
```

```
str_order(x)
```

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

```
str_sort(c(" 香蕉", " 苹果", " 梨"), locale = "ch")
```

```
## [1] "梨"   "苹果" "香蕉"
```

6. 检测匹配

`str_detect(string, pattern, negate=FALSE)` —— 检测是否存在匹配
`str_which(string, pattern, negate=FALSE)` —— 查找匹配的索引
`str_count(string, pattern)` —— 计算匹配的次数
`str_locate(string, pattern)` —— 定位匹配的位置
`str_starts(string, pattern)` —— 检测是否以 pattern 开头
`str_ends(string, pattern)` —— 检测是否以 pattern 结尾

- `string`：为要检测的字符串；
- `pattern`：为匹配的模式，可以是正则表达式；
- `negate`：默认为 FALSE 表示正常匹配，若为 TRUE 则反匹配 (找不匹配)。

```
x
```

```
## [1] "banana" "apple"  "pear"
```

```
str_detect(x, "p")
```

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

```
str_which(x, "p")
```

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

```
str_count(x, "p")
```

```
## [1] 0 2 1
```

```
str_locate(x, "a.")   # 正则表达式，. 匹配任一字符
```

```
##      start end
## [1,]     2   3
## [2,]     1   2
## [3,]     3   4
```

7. 提取字符串子集

根据指定的起始和终止位置提取子字符串，基本格式为：

```
str_sub(string, start = 1, end = -1)
```

```
str_sub(x, 1, 3)
```

```
## [1] "ban" "app" "pea"
```

```
str_sub(x, 1, 5)     # 若长度不够，则尽可能多地提取
```

```
## [1] "banan" "apple" "pear"
```

```
str_sub(x, -3, -1)
```

```
## [1] "ana" "ple" "ear"
```

提取字符向量中匹配的字符串，基本格式为：

```
str_subset(string, pattern, negate=FALSE)
```

若 negate = TRUE，则返回不匹配的字符串。

```
str_subset(x, "p")
```

```
## [1] "apple" "pear"
```

8. 提取匹配的内容

```
str_extract(string, pattern)
str_match(string, pattern)
```

- str_extract() 只提取匹配的内容；
- str_match() 提取匹配的内容以及各个分组捕获，返回矩阵，每行对应于字符向量中的一个字符串，每行的第一个元素是匹配内容，其他元素是各个分组捕获，没有匹配则为 NA

```
x = c("1978-2000", "2011-2020-2099")
pat = "\\d{4}"          # 正则表达式，匹配 4 位数字
str_extract(x, pat)
```

```
## [1] "1978" "2011"
```

```
str_match(x, pat)
```

```
##      [,1]
## [1,] "1978"
## [2,] "2011"
```

9. 修改字符串

用新字符串赋值给 str_sub() 提取的子字符串；

做字符替换，基本格式为：

```
str_replace(string, pattern, replacement)
```

- pattern：要替换的子字符串或模式；
- replacement：要替换为的新字符串。

```
x
```

```
## [1] "1978-2000"      "2011-2020-2099"
```

```
str_replace(x, "-", "/")
```

```
## [1] "1978/2000"      "2011/2020-2099"
```

10. 其他函数

- 转化大小写
  - `str_to_upper()`: 转换为大写;
  - `str_to_lower()`: 转换为小写;
  - `str_to_title()`: 转换标题格式 (单词首字母大写)

```
str_to_lower("I love r language.")
```

```
## [1] "i love r language."
```

```
str_to_upper("I love r language.")
```

```
## [1] "I LOVE R LANGUAGE."
```

```
str_to_title("I love r language.")
```

```
## [1] "I Love R Language."
```

- `str_conv(string, encoding)`: 转化字符串的字符编码
- `str_view(string, pattern, match)`: 在 Viewer 窗口输出 (正则表达式) 模式匹配结果
- `word(string, start, end, sep = " ")`: 从英文句子中提取单词
- `str_wrap(string, width = 80, indent = 0, exdent = 0)`: 调整段落格式

关于 stringr 包的注

以上查找匹配的各个函数，只是查找第一个匹配，要想查找所有匹配，各个函数都有另一版本：加后缀 `_all`，例如 `str_extract_all()`
以上各个函数中的参数 `pattern` 都支持用 **正则表达式** (Regular Expression) 表示模式。关于正则表达式，请参阅下一节。

1.4.2 日期时间

日期时间值通常以字符串形式传入 R 中，然后转化为以数值形式存储的日期时间变量。

R 的内部日期是以 1970 年 1 月 1 日至今的天数来存储，内部时间则是以 1970 年 1 月 1 日至今的秒数来存储。

tidyverse 系列的 `lubridate` 包提供了更加方便的函数，生成、转换、管理日期时间数据，足以代替 R 自带的日期时间函数。

```
library(lubridate)
```

1. 识别 日期时间

```
today()
```

```
## [1] "2021-10-11"
```

```
now()
```

```
## [1] "2021-10-11 15:09:12 CST"
```

```
as_datetime(today())   # 日期型转 日期时间型
```

```
## [1] "2021-10-11 UTC"
```

```
as_date(now())         # 日期时间型转 日期型
```

```
## [1] "2021-10-11"
```

无论年月日/时分秒按什么顺序及以什么间隔符分隔，总能正确地识别成日期时间值：

```
ymd("2020/03~01")
```

```
## [1] "2020-03-01"
```

```
myd("03202001")
```

```
## [1] "2020-03-01"
```

```
dmy("03012020")
```

```
## [1] "2020-01-03"
```

```
ymd_hm("2020/03~011213")
```

```
## [1] "2020-03-01 12:13:00 UTC"
```

注：根据需要可以 `ymd_h/myd_hm/dmy_hms` 任意组合；可以用参数 `tz ="…"` 指定时区。

也可以用 `make_date()` 和 `make_datetime()` 从日期时间组件创建日期时间：

```
make_date(2020, 8, 27)
```

```
## [1] "2020-08-27"
```

```
make_datetime(2020, 8, 27, 21, 27, 15)
```

```
## [1] "2020-08-27 21:27:15 UTC"
```

2. 格式化输出日期时间

用 format() 函数

```
d = make_date(2020, 3, 5)
format(d, '%Y/%m/%d')
```

```
## [1] "2020/03/05"
```

用 stamp() 函数，按给定模板格式输出

```
t = make_datetime(2020, 3, 5, 21, 7, 15)
fmt = stamp("Created on Sunday, Jan 1, 1999 3:34 pm")
fmt(t)
```

```
## [1] "Created on Sunday, 03 05, 2020 21:07 下午"
```

3. 提取日期时间数据的组件

日期时间数据中的‘‘年、月、日、周、时、分、秒’’等，称为其组件。

表 1.2: 可用的日期时间组件

符号	描述	示例
%d	数字表示的日期	01~31
%a	缩写的星期名	Mon
%A	非缩写的星期名	Monday
%w	数字表示的星期几	0~6 (0为周日)
%m	数字表示的月份	00~12
%b	缩写月份	Jan
%B	非缩写月份	January
%y	二位数年份	21
%Y	四位数年份	2021
%H	24 小时制小时	00~23
%I	12 小时制小时	01~12
%p	AM/PM 指示	AM/PM
%M	十进制分钟	00~60
%S	十进制秒	00~60

```
t = ymd_hms("2020/08/27 21:30:27")
t
```

```
## [1] "2020-08-27 21:30:27 UTC"
```

```
year(t)
```

```
## [1] 2020
```

```
quarter(t)              # 第几季度
```

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

```
month(t)
```

```
## [1] 8
```

```
day(t)
```

```
## [1] 27
```

```
yday(t)                 # 当年的第几天
```

```
## [1] 240
```

```
hour(t)
```

```
## [1] 21
```

```
minute(t)
```

```
## [1] 30
```

```
second(t)
```

```
## [1] 27
```

```
weekdays(t)
```

```
## [1] "星期四"
```

```
wday(t)                 # 数值表示本周第几天，默认周日是第 1 天
```

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

```
wday(t,label = TRUE)    # 字符因子型表示本周第几天
```

```
## [1] 周四
## Levels: 周日 < 周一 < 周二 < 周三 < 周四 < 周五 < 周六
```

```
week(t)                 # 当年第几周
```

```
## [1] 35
```

```
tz(t)                   # 时区
```

```
## [1] "UTC"
```

用 with_tz() 将时间数据转换为另一个时区的同一时间; force_tz() 将时间数据的时区强制转换为另一个时区:

```
with_tz(t, tz = "America/New_York")
```

```
## [1] "2020-08-27 17:30:27 EDT"
```

```
force_tz(t, tz = "America/New_York")
```

```
## [1] "2020-08-27 21:30:27 EDT"
```

还可以模糊提取 (取整) 到不同时间单位:

```
round_date(t, unit="hour")      # 四舍五入取整到小时
```

```
## [1] "2020-08-27 22:00:00 UTC"
```

注: 类似地, 向下取整: floor_date(); 向上取整: ceiling_date()

rollback(dates, roll_to_first=FALSE, preserve_hms=TRUE): 回滚到上月最后一天或本月第一天

4. 时间段数据

- interval(): 计算两个时间点的时间间隔, 返回时间段数据

```
begin = ymd_hm("2019-08-10 14:00")
end = ymd_hm("2020-03-05 18:15")
gap = interval(begin, end)
gap
```

```
## [1] 2019-08-10 14:00:00 UTC--2020-03-05 18:15:00 UTC
```

```
time_length(gap, "day")     # 计算时间段的长度为多少天
```

```
## [1] 208.1771
```

```
time_length(gap, "minute")  # 计算时间段的长度为多少分钟
```

```
## [1] 299775
```

```
t %within% gap              # 判断 t 是否属于该时间段
```

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

- duration(): 以数值 + 时间单位存储时段的长度

```
duration(100, units = "day")
```

```
## [1] "8640000s (~14.29 weeks)"
```

```
int = as.duration(gap)
int
```

```
## [1] "17986500s (~29.74 weeks)"
```

- period(): 基本同 duration()

  二者 区别：duration 是基于数值线，不考虑闰年和闰秒；period 是基于时间线，考虑闰年和闰秒。

  比如，duration 中的 1 年总是 365.25 天，而 period 的平年 365 天闰年 366 天。

- 固定单位的时间段

  period 时间段：years(), months(), weeks(), days(), hours(), minutes(), seconds();

  duration 时间段：dyears(), dmonths(), dweeks(), ddays(), dhours(), dminutes(), dseconds().

```
dyears(1)
```

```
## [1] "31557600s (~1 years)"
```

```
years(1)
```

```
## [1] "1y 0m 0d 0H 0M 0S"
```

5. 日期的时间的计算

时间点 + 时间段生成一个新的时间点：

```
t + int
```

```
## [1] "2021-03-24 01:45:27 UTC"
```

```
leap_year(2020)              # 判断是否闰年
```

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

```
ymd(20190305) + years(1)     # 加 period 的一年
```

```
## [1] "2020-03-05"
```

```
ymd(20190305) + dyears(1)    # 加 duration 的一年, 365 天
```

```
## [1] "2020-03-04 06:00:00 UTC"
```

```
t + weeks(1:3)
```

```
## [1] "2020-09-03 21:30:27 UTC" "2020-09-10 21:30:27 UTC"
## [3] "2020-09-17 21:30:27 UTC"
```

除法运算：

```
gap / ddays(1)              # 除法运算，同 time_length(gap,'day')
```

```
## [1] 208.1771
```

```
gap %/% ddays(1)            # 整除
```

```
## [1] 208
```

```
gap %% ddays(1)             # 余数
```

```
## [1] 2020-03-05 14:00:00 UTC--2020-03-05 18:15:00 UTC
```

```
as.period(gap %% ddays(1))
```

```
## [1] "4H 15M 0S"
```

月份加运算：%m+%，表示日期按月数增加。例如，生成每月同一天的日期数据：

```
date = as_date("2019-01-01")
date %m+% months(0:11)
```

```
##  [1] "2019-01-01" "2019-02-01" "2019-03-01" "2019-04-01" "2019-05-01"
##  [6] "2019-06-01" "2019-07-01" "2019-08-01" "2019-09-01" "2019-10-01"
## [11] "2019-11-01" "2019-12-01"
```

prety_dates() 生成近似的时间刻度：

```
x = seq.Date(as_date("2019-08-02"), by = "year", length.out = 2)
pretty_dates(x, 12)
```

```
##  [1] "2019-08-01 UTC" "2019-09-01 UTC" "2019-10-01 UTC"
##  [4] "2019-11-01 UTC" "2019-12-01 UTC" "2020-01-01 UTC"
##  [7] "2020-02-01 UTC" "2020-03-01 UTC" "2020-04-01 UTC"
## [10] "2020-05-01 UTC" "2020-06-01 UTC" "2020-07-01 UTC"
## [13] "2020-08-01 UTC" "2020-09-01 UTC"
```

1.4.3 时间序列

为了研究某一事件的规律，依据时间发生的顺序将事件在多个时刻的数值记录下来，就构成了一个时间序列，用 $\{Y_t\}$ 表示。

例如，国家或地区的年度财政收入，股票市场的每日波动，气象变化，工厂按小时观测的产量等等。另外，随温度、高度等变化而变化的离散序列，也可以看作时间序列。

ts 对象

R base 提供的 ts 数据类型是专门为时间序列设计的，一个时间序列数据，其实就是一个数值型向量，且每个数都有一个时刻与之对应。

用 ts() 函数生成时间序列，基本格式为：

```
ts(data, start=1, end, frequency=1, ...)
```

- data：为数值向量或矩阵；
- start：设置起始时刻；
- end：设置结束时刻；
- frequency：设置时间频率，默认为 1，表示一年有 1 个数据。

```
ts(data = 1:10, start = 2010, end = 2019)     # 年度数据
```

```
## Time Series:
## Start = 2010
## End = 2019
## Frequency = 1
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
ts(data = 1:10, start = 2010, frequency = 4)  # 季度数据
```

```
##      Qtr1 Qtr2 Qtr3 Qtr4
## 2010    1    2    3    4
## 2011    5    6    7    8
## 2012    9   10
```

同理，月度数据则 frequency = 12，周度数则为 frequency = 52，日度数据则为 frequency = 365。

tsibble

fpp3 生态下的 tsibble 包提供了整洁的时间序列数据结构 tsibble.

时间序列数据，无非就是指标数据 + 时间索引（或者再 + 分组索引）

注：多元时间序列，就是包含多个指标列。

对于分组时间序列数据，首先是一个数据框，若有分组变量需采用'' 长格式'' 作为一列（长宽格式及转化见 2.4 节），只需要指定时间索引、分组索引，就能变成时间序列数据结构。

例如，现有 tibble 格式的 3 个公司 2017 年的日度股票数据，其中存放 3 只股票的 Stock 列为分组索引：

```
load("datas/stocks.rda")
df
```

```
## # 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.
## 6 2017-01-04 Apple   116.
## # ... with 747 more rows
```

用 `as_tsibble()` 将数据框转化为时间序列对象 `tsibble`, 只需要指定时间索引（index)、分组索引（key)：

```
library(fpp3)
stocks = as_tsibble(df, key = Stock, index = Date)
stocks
```

```
## # A tsibble: 753 x 3 [1D]
## # Key:       Stock [3]
##   Date       Stock  Close
##   <date>     <chr>  <dbl>
## 1 2017-01-03 Amazon  754.
## 2 2017-01-04 Amazon  757.
## 3 2017-01-05 Amazon  780.
## 4 2017-01-06 Amazon  796.
## 5 2017-01-09 Amazon  797.
## 6 2017-01-10 Amazon  796.
## # ... with 747 more rows
```

`tsibble` 对象非常便于后续处理和探索：

```
stocks %>%
  group_by_key() %>%
  index_by(weeks = ~ yearweek(.)) %>%    # 周度汇总
  summarise(max_week = mean(Close))
autoplot(stocks)                         # 可视化
```

更多时间序列领域的相关包和资料，可参阅 (Hyndman and Athanasopoulos 2021) 和 Github 社区：business-science.

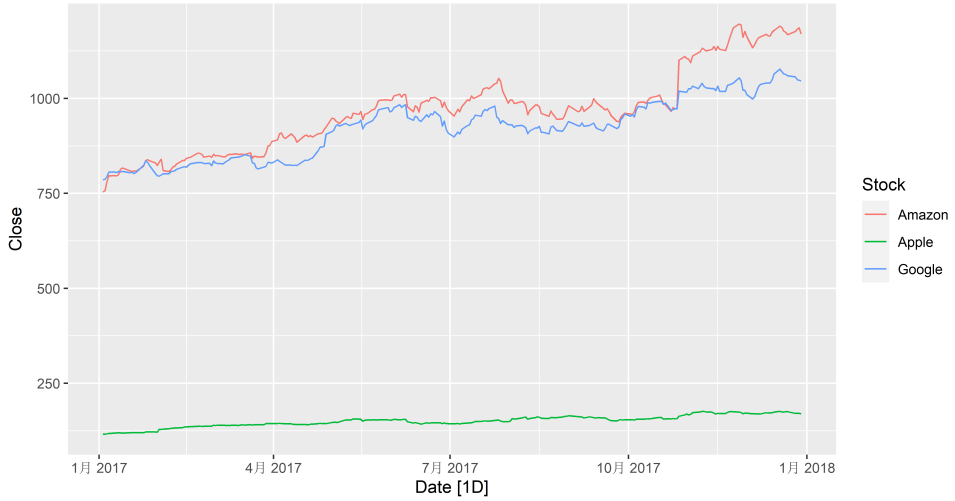

图 1.11: 可视化股票数据

本节部分内容参阅 (G. G. Hadley Wickham 2017), RStudio Cheatsheets: stringr, lubridate.

1.5 正则表达式

正则表达式，是根据字符串规律按一定法则，简洁表达一组字符串的表达式。正则表达式通常就是从貌似无规律的字符串中发现规律性，进而概括性地表达它们所共有的规律或模式，以方便地操作处理它们，这是真正的**化繁为简，以简驭繁**的典范。

几乎所有的高级编程语言都支持正则表达式，正则表达式广泛应用于文本挖掘、数据预处理，例如：

- 检查文本中是否含有指定的特征词
- 找出文本中匹配特征词的位置
- 从文本中提取信息
- 修改文本

正则表达式包括：只能匹配自身的普通字符 (如英文字母、数字、标点等) 和被转义了的特殊字符 (称为 ‘‘元字符”) 。

1.5.1 基本语法

1. 常用的元字符

表 1.3: 常用的元字符

符号	描述
.	匹配除换行符 “/n’’ 以外的任意字符
\\	转义字符，匹配元字符时，使用 “\\元字符”
|	表示或者，即 | 前后的表达式任选一个
^	匹配字符串的开始

符号	描述
( )	提取匹配的字符串，即括号内的看成一个整体，即指定子表达式
[ ]	可匹配方括号内任意一个字符
{ }	前面的字符或表达式的重复次数：{n} 表示重复 n 次；{n,} 重复 n 次到更多次；{n, m} 表示重复 n 次到 m 次
*	前面的字符或表达式重复 0 次或更多次
+	前面的字符或表达式重复 1 次或更多次
?	前面的字符或表达式重复 0 次或 1 次

其他语言中的转义字符一般是 \；在多行模式下，^ 和 $ 就表示行的开始和结束。

```
# 创建多行模式的正则表达式
pat = regex("^\\(.+?\\)$", multiline = TRUE)
```

2. 特殊字符类与反义

表 1.4: 特殊字符类与反义

符号	描述
\\d 与\\D	匹配数字，匹配非数字
\\s 与\\S	匹配空白符，匹配非空白符
\\w 与\\W	匹配字母或数字或下划线或汉字，匹配非\w 字符
\\b 与\\B	匹配单词的开始或结束的位置，匹配非\b 的位置
\\h 与\\H	匹配水平间隔，匹配非水平间隔
\\v 与\\V	匹配垂直间隔，匹配非垂直间隔
[^...]	匹配除了...以外的任意字符

- \\S+: 匹配不包含空白符的字符串
- \\d: 匹配数字，同 [0-9]
- [a-zA-Z0-9]: 匹配字母和数字
- [\u4e00-\u9fa5] 匹配汉字
- [^aeiou]: 匹配除 aeiou 之外的任意字符，即匹配辅音字母

3. POSIX 字符类

表 1.5: POSIX 字符类

符号	描述
[[:lower:]]	小写字母

符号	描述
[[:upper:]]	大写字母
[[:alpha:]]	大小写字母
[[:digit:]]	数字 0~9
[[:alnum:]]	字母和数字
[[:blank:]]	空白符包括空格、制表符、换行符、中文全角空格等
[[:cntrl:]]	控制字符
[[:punct:]]	标点符号包括！” # % & ’ () * + - . / : ；等
[[:space:]]	空格字符：空格, 制表符, 垂直制表符, 回车, 换行符, 换页符
[[:xdigit:]]	十六进制数字：0-9 A-F a-f
[[:print:]]	打印字符：[[:alpha:]], [[:punct:]], [[:space:]]
[[:graph:]]	图形化字符：[[:alpha:]], [[:punct:]]

4. 运算优先级

圆括号括起来的表达式最优先，其次是表示重复次数的操作 (即 * + { })；再次是连接运算 (即几个字符放在一起，如 abc)；最后是或者运算 (|)。

另外，正则表达式还有若干高级用法，常用的有**零宽断言**和**分组捕获**，将在下面实例中进行演示。

1.5.2 若干实例

以上正则表达式语法组合起来使用，就能产生非常强大的匹配效果，对于匹配到的内容，根据需要可以提取它们，可以替换它们。

正则表达式与 stringr 包连用

若只是调试和查看正则表达式的匹配效果，可用 str_view() 及其 _all 后缀版本，将在 RStudio 的 Viewer 窗口显示匹配结果，在原字符向量中高亮显示匹配内容，非常直观。
若要提取正则表达式匹配到的内容，则用 str_extract() 及其 _all 后缀版本。
若要替换正则表达式匹配到的内容，则用 str_replace() 及其 _all 后缀版本。

使用正则表达式关键是，能够从貌似没有规律的字符串中发现规律性，再将规律性用正则表达式语法表示出来。下面看几个正则表达式比较实用的实例。

例 1.1 直接匹配

适合想要匹配的内容具有一定规律性，该规律性可用正则表达式表示出来。比如，数据中包含字母、符号、数值，我们想提取其中的数值，按正则表达式语法规则直接把要提取的部分表示出来：

```
x = c("CDK 弱 (+)10%+", "CDK(+)30%-", "CDK(-)0+", "CDK(++)60%*")
str_view(x, "\\d+%")
str_view(x, "\\d+%?")
```

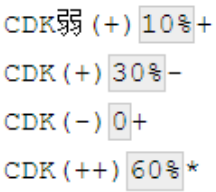

图 1.12: Viewer 窗口显示匹配效果

\\d 表示匹配一位数字，+ 表示前面数字重复 1 次或多次，接着 % 原样匹配 %.
若后面不加 ? 则必须匹配到 % 才会成功，故第 3 个字符串就不能成功匹配；若后面加上 ? 则表示匹配前面的 % 0 次或 1 次，从而能成功匹配第 3 个字符串。

例 1.2 (零宽断言) 匹配两个标志之间的内容

适合想要匹配的内容没有规律性，但该内容位于两个有规律性的标志之间，标志也可以是开始和结束。

通常想要匹配的内容不包含两边的 ‘‘标志’’，这就需要用**零宽断言**。简单来说，就是一种引导语法告诉既要匹配到 ‘‘标志’’，但又不包含 ‘‘标志’’。左边标志的引导语法是 (?<= 标志)，右边标志的引导语法是 (?= 标志)，而真正要匹配的内容放在它们中间。

比如，来自问卷星 ‘‘来自 IP” 数据，想要提取 IP、省份。

```
x = c("175.10.237.40(湖南-长沙)", "114.243.12.168(北京-北京)",
      "125.211.78.251(黑龙江-哈尔滨)")
# 提取省份
str_extract(x, "\\(.*-")            # 对比，不用零宽断言
```

```
## [1] "(湖南-"   "(北京-"   "(黑龙江-"
```

```
str_extract(x, "(?<=\\().*(?=-)")  # 用零宽断言
```

```
## [1] "湖南"   "北京"   "黑龙江"
```

```
# 提取 IP
# str_extract(x, "\\d.*\\d")        # 直接匹配
str_extract(x, "^.*(?=\\()")        # 用零宽断言
```

```
## [1] "175.10.237.40"  "114.243.12.168" "125.211.78.251"
```

省份位于两个标志 “(” 和 “-” 之间，但又不包含该标志，这就需要用到零宽断言。
IP 位于两个标志 ‘‘开始” 和 “(” 之间，左边用开始符号 ^，右边用零宽断言。

再比如，用零宽断言提取专业 (位于 ‘‘级’’ 和数字之间) :

```
x = c("18 级能源动力工程 2 班", "19 级统计学 1 班")
str_extract(x, "(?<= 级).*?(?=[0-9])")
```

```
## [1] "能源动力工程" "统计学"
```

关于懒惰匹配

正则表达式正常都是贪婪匹配，即重复直到文本中能匹配的最长范围，例如匹配小括号：

```
str_extract("(1st) other (2nd)", "\\(.+\\)")
```

```
## [1] "(1st) other (2nd)"
```

若想只匹配到第 1 个右小括号，则需要懒惰匹配，在重复匹配后面加上 ? 即可：

```
str_extract("(1st) other (2nd)", "\\(.+?\\)")
```

```
## [1] "(1st)"
```

例 1.3 分组捕获

正则表达式中可以用圆括号来分组，作用是

- 确定优先规则
- 组成一个整体
- 拆分出整个匹配中的部分内容 (称为捕获)
- 捕获内容供后续引用或者替换。

比如，来自瓜子二手车的数据：若型号是中文，则品牌与型号中间有空格；若型号为英文或数字，则品牌与型号中间没有空格。

若用正则表达式匹配 ‘‘字母或数字” 并分组，然后捕获该分组并用添加空格替换：

```
x = c(" 宝马 X3 2016 款", " 大众 速腾 2017 款", " 宝马 3 系 2012 款")
str_replace(x, "([a-zA-Z0-9])", " \\1")
```

```
## [1] "宝马 X3 2016款"    "大众 速腾  2017款" "宝马 3系 2012款"
```

后续再用空格分割列即可。更多分组的引用还有 `\\2`, `\\3`, . . .

:::: {.rmdnote data-latex=“{}”} 最后，再推荐一个来自 Github 可以推断正则表达式的包 inferregex，用函数 `infer_regex()` 可根据字符串推断正则表达式。

本节部分内容参阅 (G. G. Hadley Wickham 2017), (李东风 2020), 中国大学生慕课网, 嵩天：Python 网络爬虫与信息提取，deerchao 博客园, 正则表达式 30 分钟入门教程。

1.6 控制结构

编程中的控制结构，是指分支结构和循环结构。

1.6.1 分支结构

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，是用条件语句做判断以实现分支：

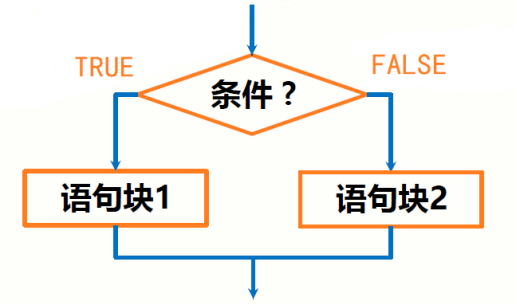

图 1.13: 分支结构示意图

R 语言中的条件语句的一般格式为：

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。

特别注意：分支的本意就是，不同分支之间不存在交叉 (重叠) 。

另一种多分支的写法是用 `switch()`:

```
x = "b"
v = switch(x, "a"="apple", "b"="banana", "c"="cherry")
v
```

```
## [1] "banana"
```

它的一个应用场景是：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，见 1.7.1 节。

例 1.5 实现将百分制分数转化为五级制分数

```
if(score >= 90) {
  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"
}
```

注意，若先写 `>=60`，结果就不对了。

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)
```

1.6.2 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。

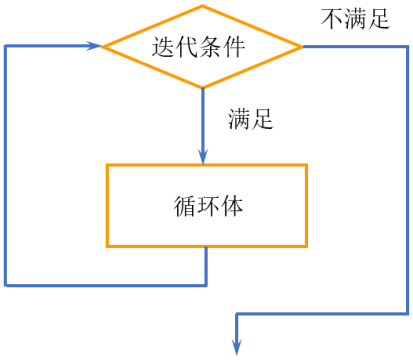

图 1.14: 循环结构示意图

R 语言有三种方式实现循环:

- for 循环、while 循环、repeat 循环
- apply 函数族
- 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

两点说明

关于 “for 循环运行速度慢” 的说法，实际上已经过时了，现在的 R、Matlab 等软件经过多年的内部优化已经不慢了，之所以表现出来慢，是因为你没有注意两个关键点:

- 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。

apply 函数族和 purrr 泛函式编程能够更加高效简洁地实现一般的 for 循环、while 循环，但这不代表 for 循环、while 循环就没用了，它们可以在更高的层次使用 (相对于在逐元素级别使用)。

1. for 循环

(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] 5.843333
```

```
mean(df[[2]])
```

```
## [1] 3.057333
```

```
mean(df[[3]])
```

```
## [1] 3.758
```

```
mean(df[[4]])
```

```
## [1] 1.199333
```

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] 5.843333 3.057333 3.758000 1.199333
```

for 循环有三个组件：

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

在循环开始之前，最好为输出分配足够的存储空间，这样效率更高：若每循环一次，就用 `c()` 合并一次，效率会很低下。

通常是用 `vector()` 函数创建一个给定长度的空向量，它有两个参数：向量类型 (logical, integer, double, character 等)、向量长度。

(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`.

有时候会用 `1:length(df)`, 但更安全的做法是用 `seq_along(df)` ，它能保证即使不小心遇到长度为 0 的向量时，仍能正确工作。

(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]])`
- ……

(2) for 循环的几种常用操作

(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]`.

若要创建命名的输出，可按如下方式命名结果向量：

```
results = vector("list", length(x))
names(results) = names(x)
```

用数值索引迭代是最常用的形式，因为名字和元素都可以根据索引提取：

```
for (i in seq_along(x)) {
    name = names(x)[i]
    value = x[i]
}
```

(ii) 将每次循环得到的结果合并为一个整体对象

这种情形，在 `for` 循环中经常遇到。此时要尽量避免“每循环一次，就做一次拼接”，这样效率很低。更好的做法是：先将结果保存为列表，等循环结束再将列表 `unlist()` 或 `purrr::flatten_dbl()` 成一个向量。

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)
}
```

另外两种类似的情形是：

- 生成一个长字符串。不是用 `str_c()` 函数将上一次的迭代结果拼接到一起，而是将结果保存为字符向量，再用函数 `str_c(output, collapse= " ")` 合并为一个单独的字符串；
- 生成一个大的数据框。不是依次用 `rbind()` 函数合并每次迭代的结果，而是将结果保存为列表，再用 `dplyr::bind_rows(output)` 函数合并成一个单独的数据框，或者直接一步到位用 `purrr::map_dfr()`。

所以，遇到上述模式时，要先转化为更复杂的结果对象，最后再做一步合并。

2. while 循环

适用于迭代次数未知。

while 循环更简单，因为它只包含两个组件：条件、循环体：

```
while (condition) {
  # 循环体
}
```

While 循环是比 for 循环更一般的循环，因为 for 循环总可以改写为 while 循环，但 while 循环不一定能改写为 for 循环：

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

下面用 while 循环实现：反复随机生成标准正态分布随机数 (见 1.7.2 节)，若值大于 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] -0.5604756
## [1] -0.2301775
## [1] 1.558708
```

while 循环并不常用，但在模拟时常用，特别是预先不知道迭代次数的情形。

3. repeat 循环

重复执行循环体，直到满足退出条件：

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

注意，repeat 循环至少会执行一次。

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
```

4. apply 函数族

更建议弃用 apply 函数族，直接用 purrr::map 系列。

(1) apply()

apply() 函数是最常用的代替 for 循环的函数。可以对矩阵、数据框、多维数组，按行或列进行循环计算，即将行或列的元素逐个传递给函数 FUN 进行计算。其基本格式为：

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

- x：为数据对象 (矩阵、多维数组、数据框)；
- MARGIN：1 表示按行，2 表示按列；
- FUN：表示要作用的函数。

```
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
##     5.843333     3.057333     3.758000     1.199333
```

(2) tapply()

该函数可以按照因子分组计算分组统计：

```
height = c(165, 170, 168, 172, 159)
sex = factor(c(" 男", " 女", " 男", " 男", " 女"))
tapply(height, sex, mean)
```

```
##       男       女
## 168.3333 164.5000
```

注意, height 与 sex 是等长的，对应元素分别为同一人的身高和性别，tapply() 函数分男女两组计算了身高平均值。

(3) lapply()

lapply() 函数是一个最基础循环操作函数，用来对 vector、list、data.frame 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。其基本格式为：

```
lapply(x, FUN, ...)
```

- x：为数据对象 (列表、数据框、向量) ；
- FUN：表示要作用的函数。

```
lapply(df, mean)      # 对前文 df 计算各列的均值
```

```
# $Sepal.Length
# [1] 5.843333
#
# $Sepal.Width
# [1] 3.057333
#
# $Petal.Length
# [1] 3.758
#
# $Petal.Width
# [1] 1.199333
```

(4) sapply()

sapply() 函数是 lapply() 的简化版本，多了一个参数 simplify，若 simplify=FALSE，则同 lapply()，若为 TRUE，则将输出的 list 简化为向量或矩阵。其基本格式为：

```
sapply(x, FUN, simplify = TRUE, ...)
```

```
sapply(df, mean)      # 对前文 df 计算各列的均值
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.843333     3.057333     3.758000     1.199333
```

5. purrr 泛函式编程

相对于 apply 族，purrr 泛函式编程提供了更多的一致性、规范性和便利性，更容易记住和使用。

(1) 先来理解几个概念

循环迭代

就是将函数依次应用（映射）到序列的每一个元素上，做相同的操作

序列，是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。原子向量和列表都是序列。

泛函式编程

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。表示出来不就是泛函式[7]：`map(x, f)`。

purrr 泛函式编程解决循环迭代问题的逻辑是：

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（每个元素作用后返回一个结果）打包到一起返回，并且可以根据想要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 系列函数都有后缀形式，以决定循环迭代之后返回的数据类型，这是 purrr 比 apply 函数族更先进和便利的一大优势。常用后缀如下：

- `map_chr(.x, .f)`: 返回字符型向量
- `map_lgl(.x, .f)`: 返回逻辑型向量
- `map_dbl(.x, .f)`: 返回实数型向量
- `map_int(.x, .f)`: 返回整数型向量
- `map_dfr(.x, .f)`: 返回数据框列表，再 `bind_rows` 按行合并为一个数据框
- `map_dfc(.x, .f)`: 返回数据框列表，再 `bind_cols` 按列合并为一个数据框

purrr 风格公式

在序列上做循环迭代（应用函数），经常需要自定义函数，但有些简单的函数也用 `function` 定义一番，毕竟是麻烦和啰嗦。所以，purrr 包提供了对 purrr 风格公式（匿名函数）的支持。

熟悉其他语言的匿名函数的话，很自然地就能习惯。

前面说了，purrr 包实现迭代循环是用 `map(x, f)`，`f` 是要应用的函数，想用匿名函数来写它，它要应用在序列 `x` 上，就是要和序列 `x` 相关联，那么就限定用序列参数名关联好了，即将该序列参数名作为匿名函数的参数使用：

- 一元函数：序列参数是 `.x` 比如，$f(x) = x^2 + 1$, 其 purrr 风格公式就写为：`~ .x ^ 2 + 1`
- 二元函数：序列参数是 `.x`, `.y` 比如，$f(x, y) = x^2 - 3y$, 其 purrr 风格公式就写为：`~ .x ^ 2 - 3 * .y`
- 多元函数：序列参数是 `..1`, `..2`, `..3` 等比如，$f(x, y, z) = ln(x + y + z)$, 其 purrr 风格公式就写为：`~ log(..1 + ..2 + ..3)`

> 所有序列参数，可以用 `...` 代替，比如，`sum(...)` 同 `sum(..1, ..2, ..3)`

(2) `map()`: 依次应用一元函数到一个序列的每个元素

[7] 将序列（要操作的数据）作为第一个参数，是为了便于使用管道。

```
map(.x, .f, ...)
map_*(.x, .f, ...)
```

- `.x` 为序列
- `.f` 为要应用的一元函数，或 `purrr` 风格公式（匿名函数）
- `...` 可设置函数`.f`的其他参数

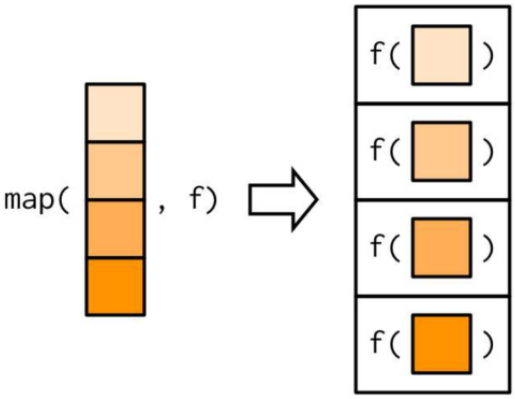

图 1.15: map 函数作用机制示意图

`map()`返回结果列表，基本同 `lapply()`。例如，计算前文 `df`，每列的均值，即依次将 `mean()` 函数，应用到第 1 列，第 2 列，...；并控制返回结果为 `double` 向量：

```
map(df, mean)
```

```
## $Sepal.Length
## [1] 5.843333
##
## $Sepal.Width
## [1] 3.057333
##
## $Petal.Length
## [1] 3.758
##
## $Petal.Width
## [1] 1.199333
```

说明：`df` 是数据框 (特殊的列表)，作为序列其元素依次是：`df[[1]]`, `df[[2]]`, `.....` 所以，`map(df, mean)` 相当于依次计算：`mean(df[[1]])`, `mean(df[[2]])`, `.....`

返回结果是 `double` 型数值，所以更好的做法是，控制返回类型为数值向量，只需：

```
map_dbl(df, mean)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.843333     3.057333     3.758000     1.199333
```

另外，mean() 函数还有其他参数，如 na.rm，若上述计算过程需要设置忽略缺失值，只需：

```
map_dbl(df, mean, na.rm = TRUE)        # 数据不含 NA，故结果同上（略）
map_dbl(df, ~mean(.x, na.rm = TRUE))   # purrr 风格公式写法
```

有了 map() 函数，对于自定义只接受标量的一元函数，比如 f(x)，想要让它支持接受向量作为输入，根本不需要改造原函数，只需：

```
map_*(xs, f)                           # xs 表示若干个 x 构成的序列
```

(3) map2()：依次应用二元函数到两个序列的每对元素

```
map2(.x, .y .f, ...)
map2_*(.x, .y, .f, ...)
```

.x 为序列 1
.y 为序列 2
.f 为要应用的二元函数，或 purrr 风格公式 (匿名函数)
... 可设置函数 .f 的其他参数

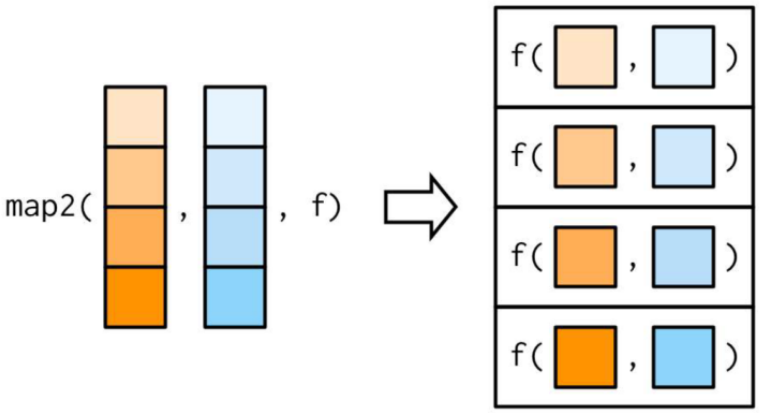

图 1.16: map2 函数作用机制示意图

例如，根据身高、体重数据计算 BMI 指数

```
height = c(1.58, 1.76, 1.64)
weight = c(52, 73, 68)

cal_BMI = function(h, w) w / h ^ 2      # 定义计算 BMI 的函数
map2_dbl(height, weight, cal_BMI)
```

```
## [1] 20.83000 23.56663 25.28257
```

说明：序列 1 其元素为：height[[1]], height[[2]], ...

序列 2 其元素为：weight[[1]], weight[[2]], ...

所以，`map2_dbl(height, weight, cal_BMI)` 相当于依次计算：

`cal_BMI(height[[1]], weight[[1]])`, `cal_BMI(height[[2]], weight[[2]])`, ……

更简洁的 `purrr` 风格公式写法 (省了自定义函数)：

```
map2_dbl(height, weight, ~ .y / .x^2)
```

同样，有了 `map2()` 函数，对于自定义只接受标量的二元函数，比如 `f(x, y)`，想要让它支持接受向量作为输入，根本不需要改造原函数，只需：

```
map2_*(xs, ys, f)              # xs, ys 分别表示若干个 x, y 构成的序列
```

(4) `pmap()`：应用多元函数到多个序列的每组元素，可以实现对数据框逐行迭代

多个序列的长度相同，长度相同的列表，不就是数据框吗。所以，`pmap()` 的多元迭代就是依次在数据框的每一行上进行迭代！

```
pmap(.l, .f, ...)
pmap_*(.l, .f, ...)
```

- `.l` 为数据框，
- `.f` 为要应用的多元函数
- `...` 可设置函数 `.f` 的其他参数

注： `.f` 是几元函数，对应数据框 `.l` 有几列，`.f` 将依次在数据框 `.l` 的每一行上进行迭代。

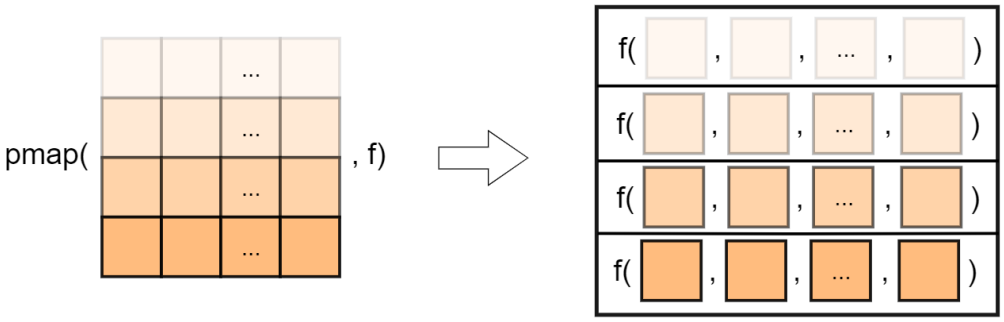

图 1.17: pmap 函数作用机制示意图

例如，分别生成不同数量不同均值、标准差的正态分布随机数。

```
df = tibble(
  n = c(1, 3, 5),
  mean = c(5, 10, -3),
  sd = c(1, 5, 10))
df
```

```
## # A tibble: 3 x 3
```

```
##       n  mean    sd
##   <dbl> <dbl> <dbl>
## 1     1     5     1
## 2     3    10     5
## 3     5    -3    10
```

```
set.seed(123)
pmap(df, rnorm)
```

```
## [[1]]
## [1] 4.439524
##
## [[2]]
## [1]  8.849113 17.793542 10.352542
##
## [[3]]
## [1]  -1.707123  14.150650   1.609162 -15.650612  -9.868529
```

说明：这里的 `rnorm(n, mean, sd)` 是三元函数，`pmap(df, rnorm)` 相当于将三元函数 `rnorm()` 依次应用到数据框 `df` 的每一行上，即依次执行：

```
rnorm(1, 5, 1), rnorm(3, 10, 5), rnorm(5, -3, 10)
```

注意，这里 `df` 中的列名，必须与 `rnorm()` 函数的参数名相同（列序随便）。若要避免这种局限，可以使用 purrr 风格公式写法：

```
pmap(df, ~ rnorm(..1, ..2, ..3))    # 或者简写为
pmap(df, ~ rnorm(...))
```

`pmap_*()` 提供了一种行化操作数据框的办法。

```
pmap_dbl(df, ~ mean(c(...)))        # 按行求均值
```

```
## [1] 2.333333 6.000000 4.000000
```

```
pmap_chr(df, str_c, sep = "-")      # 将各行拼接在一起
```

```
## [1] "1-5-1"   "3-10-5"  "5--3-10"
```

其它 purrr 函数

- `imap_*(.x, .f)`: 带索引的 `map_*()` 系列，迭代的时候既迭代元素，又迭代元素的索引（位置或名字），purrr 风格公式中用.y 表示索引；
- `invoke_map_*(.f, .x, ...)`: 将多个函数依次应用到序列，相当于依次执行：`.f[[1]](.x, ...)`, `.f[[2]](.x, ...)`, ……
- `walk` 系列：`walk(.l, .f, ...)`, `walk2(.l, .f, ...)`, `pwalk(.l, .f, ...)`
  - 将函数依次作用到序列上，不返回结果。有些批量操作是没有或不关心返回结果的，例如批量保存到文件比如批量保存到文件等。
- `modify` 系列：`modify(.x, .f, ...)`, `modify2(.x, .y, .f, ...)`, `modify_depth(.x, .depth, .f, ...)`
  - 将函数 `.f` 依次作用到序列 `.x`，并返回修改后的序列 `.x`
- `reduce()`: 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，……直到所有的元都被 “reduced”
  - `reduce(1:100, sum)` 是对 1:100 求累加和；
  - `reduce()` 可用于批量数据连接
- `accumulate()`: 与 `reduce()` 作用方式相同，不同之处是：`reduce()` 只返回最终的结果，而 `accumulate()` 会返回所有中间结果。

本节部分内容参阅 (G. G. Hadley Wickham 2017), Hadley Wickham: The joy of functional programming, Charlotte Wickham. Solving Iteration Problems With Purrr, Hendrik van Broekhuizen: purrr beyond map, functional programming in R, RStudio Cheatsheet: purrr.

1.7 自定义函数

编程中的函数，是用来实现某个功能，其一般形式为：

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

你只要把输入给它，它就能在内部进行相应处理，把你想要的返回值给你。

这些输入和返回值，在函数定义时，都要有固定的类型（模具）限制，叫作形参（形式上的参数）；在函数调用时，必须给它对应类型的具体数值，才能真正的去做处理，这叫作实参（实际的参数）。

所以，定义函数就好比创造一个模具，调用函数就好比用模具批量生成产品。

使用函数最大的好处，就是将实现某个功能，封装成模具，从而可以反复使用。这就避免了写大量重复的代码，程序的可读性也大大加强。

以前文的**百分制分数转化为五级制分数**为例，如果来一个百分制分数，就这样转化一次，10 个学生分数，就得写 100 多行代码。所以有必要封装成一个函数。

1.7.1 自定义函数

1. 自定义函数的一般语法

R 中，自定义函数的一般格式为：

```
函数名 = function(输入 1, ..., 输入 n) {
  函数体
  return(返回值)
}
```

> 注意，return 并不是必需的，默认函数体最后一行的值作为返回值，也就是说”return(返回值)‘‘完全可以换成” 返回值 ‘‘。

2. 怎么自定义一个函数

我们想要自定义一个函数，能够实现**把百分制分数转化为五级制分数**的功能。

基于前面函数的理解，

第一步，分析输入和输出，设计函数外形

- 输入有几个，分别是什么，适合用什么数据类型存放；
- 输出有几个，分别是什么，适合用什么数据类型存放。

本问题，输入有 1 个：百分制分数，数值型；输出有 1 个：五级制分数，字符串

- 然后就可以设计自定义函数的外形：

```
Score_Conv = function(score) {
# 实现将一个百分制分数转化为五级分数
# 输入参数：score 为数值型，百分制分数
# 返回值：res 为字符串型，五级分数
...
}
```

> 函数名和变量可以随便起名，但是建议用有含义的单词。另外，为函数增加注释是一个好习惯。这些都是为了代码的可读性。

第二步，梳理功能的实现过程

前言中在谈如何自己写代码时讲到：‘‘分解问题 + 实例梳理 + 翻译及调试’’，完全适用于这里，不再赘述。

拿一组本例（只有一个）具体的形参的值作为输入，比如 76 分，分析怎么到达对应的五级分数‘‘良’’。这依赖于对五级分数界限的选取，选定之后做分支判断即可实现，即前文的条件语句中的示例那样。

复杂的功能，就需要更耐心的梳理和思考甚至借助一些算法，当然也离不开逐代码片段的调试。

```
score = 76
if(score >= 90) {
  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"
}
res
```

```
## [1] "中"
```

拿一组具体的形参值作为输入，通过逐步调试，得到正确的返回值结果，这一步骤非常关键和有必要。

第三步，将第二步的代码封装到函数体

```
Score_Conv = function(score) {
  if(score >= 90) {
    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"
  }
  res
}
```

基本就是原样作为函数体放入函数，原来的变量赋值语句不需要了，只需要形参。

3. 调用函数

要调用自定义函数，必须要先加载到当前变量窗口（内存），有两种方法：

- 需要选中并执行函数代码，或者

- 将函数保存为同名的 `Score_Conv.R` 文件，注意勾选”Source on save” 再保存，然后执行 `source("Score_Conv.R", encoding="UTF-8")`

然后就可以调用函数了，给它一个实参 76，输出结果为’’ 中’’：

```
Score_Conv(76)
```

```
## [1] "中"
```

关于函数传递参数

要调用一个函数，比如 `f(x, y)`, 首先要清楚其形参 `x, y` 所要求的类型，假设 `x` 要求是数值向量, `y` 要求是单个逻辑值。

那么，要调用该函数，首先需要准备与形参类型相符的实参（同名异名均可），比如

```
a = c(3.56, 2.1)
b = FALSE
```

再调用函数：

```
f(a, b)      # 同直接给值: f(c(3.56,2.1), FALSE)
```

调用函数时若不指定参数名，则默认是根据位置关联形参，即以 `x = a, y = b` 的方式进入函数体。

调用函数时若指定参数名，则根据参数名关联形参，位置不再重要，比如

```
f(y = b, x = a)      # 效果同上
```

4. 向量化改进

我们希望自定义函数，也能处理向量输入，即输入多个百分制分数，能一下都转化为五级分数。这也是所谓的’’ 向量化编程’’ 思维，就是要习惯用向量（矩阵）去思考、去表达。

方法一：修改 自定义 函数

将输入参数设计为数值向量，函数体也要相应的修改，借助循环依次处理向量中的每个元素，就相当于再套一层 `for` 循环。

```
Score_Conv2 = function(score) {
  n = length(score)
  res = vector("character", n)
  for(i in 1:n) {
    if(score[i] >= 90) {
      res[i] = " 优"
    } else if(score[i] >= 80) {
      res[i] = " 良"
    } else if(score[i] >= 70) {
```

```
      res[i] = " 中"
    } else if(score[i] >= 60) {
      res[i] = " 及格"
    } else {
      res[i] = " 不及格"
    }
  }
  res
}

# 测试函数
scores = c(35, 67, 100)
Score_Conv2(scores)
```

```
## [1] "不及格" "及格"   "优"
```

方法二：借助 apply 族或 map 系列函数

简单的循环语句，基本都可以改用 apply 族或 map 系列函数实现，其作用相当于是依次'' 应用'' 某函数，到序列的每个元素上。

也就是说，不需要修改原函数，直接就能实现向量化操作：

```
scores = c(35, 67, 100)
map_chr(scores, Score_Conv)
```

```
## [1] "不及格" "及格"   "优"
```

5. 处理多个返回值

若自定义函数需要有多个返回值，R 的处理方法是，将多个返回值放入一个列表（或数据框），再返回一个列表。

例如，自定义函数，实现计算一个数值向量的均值和标准差：

```
MeanStd = function(x) {
  mu = mean(x)
  std  = sqrt(sum((x-mu)^2) / (length(x)-1))
  list(mu=mu, std=std)
}
# 测试函数
x = c(2, 6, 4, 9, 12)
MeanStd(x)
```

```
## $mu
## [1] 6.6
##
## $std
## [1] 3.974921
```

6. 默认参数值

有时候需要为输入参数设置默认值。

以前面的计算数值向量的均值和标准差的函数为例。我们知道，标准差的计算公式有两种形式，一种是总体标准差除以 n，一种是样本标准差除以 $n-1$.

此时，没有必要写两个版本的函数，只需要再增加一个指示参数，将用的多的版本设为默认即可。

```
MeanStd2 = function(x, type = 1) {
  mu = mean(x)
  n = length(x)
  if(type == 1) {
    std  = sqrt(sum((x - mu) ^ 2) / (n - 1))
  } else {
    std  = sqrt(sum((x - mu) ^ 2) / n)
  }
  list(mu = mu, std = std)
}
# 测试函数
x = c(2, 6, 4, 9, 12)
# MeanStd2(x)                  # 同 MeanStd(x)
MeanStd2(x, 2)
```

```
## $mu
## [1] 6.6
##
## $std
## [1] 3.555278
```

用 `type = 1` 来指示表意并不明确，可以用表意更明确的字符串来指示，这就需要用到 `switch()`，让不同的指示值 `=` 相应的代码块，因为代码块往往是多行，需要用大括号括起来，注意分支与分支之间的逗号不能少。

```
MeanStd3 = function(x, type = "sample") {
  mu = mean(x)
  n = length(x)
```

```
  switch(type,
         "sample" = {
           std = sqrt(sum((x - mu) ^ 2) / (n - 1))
           },
          "population" = {
            std = sqrt(sum((x - mu) ^ 2) / n)
          })
  list(mu = mu, std = std)
}
MeanStd3(x)
```

```
## $mu
## [1] 6.6
##
## $std
## [1] 3.974921
```

```
MeanStd3(x, "population")
```

```
## $mu
## [1] 6.6
##
## $std
## [1] 3.555278
```

7. “...” 参数

一般函数参数只接受一个对象，即使不指定参数名，也会按位置对应参数。例如

```
my_sum = function(x, y) {
    sum(x, y)
}
my_sum(1, 2)
```

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

但是，如果想对 3 个数加和，怎么办？直接 `my_sum(1, 2, 3)` 会报错。

`...` 是一个特殊参数，可以接受任意多个对象，并作为一个列表传递它们：

```
dots_sum = function(...) {
    sum(...)
}
dots_sum(1)
```

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

```
dots_sum(1, 2, 3, 4, 5)
```

```
## [1] 15
```

几乎所有 R 自带函数都在用 ... 这样传递参数。若参数 ... 后面还有其他参数，为了避免歧义调用函数时需要对其随后参数命名。

1.7.2 R 自带函数

除了自定义函数，还可以使用现成的函数：

- 来自 R base：可直接使用
- 来自各种扩展包：需载入包，或加上包名前缀：包名::函数名()

这些函数的使用，可以通过 ?函数名查阅其帮助，以及查阅包页面的 Reference manual 和 Vignettes (若有)。

下面对常用的 R 自带函数做分类总结。

1. 基本数学函数

```
round(x, digits)     # IEEE 754 标准的四舍五入，保留 n 位小数
signif(x, digits)    # 四舍五入，保留 n 位有效数字
ceiling(x)           # 向上取整，例如 ceiling(pi) 为 4
floor(x)             # 向下取整，例如 floor(pi) 为 3
sign(x)              # 符号函数
abs(x)               # 取绝对值
sqrt(x)              # 求平方根
exp(x)               # e 的 x 次幂
log(x, base)         # 对 x 取以... 为底的对数，默认以 e 为底
log2(x)              # 对 x 取以 2 为底的对数
log10(x)             # 对 x 取以 10 为底的对数
Re(z)                # 返回复数 z 的实部
Im(z)                # 返回复数 z 的虚部
Mod(z)               # 求复数 z 的模
Arg(z)               # 求复数 z 的辐角
Conj(z)              # 求复数 z 的共轭复数
```

2. 三角函数与双曲函数

```
sin(x)               # 正弦函数
cos(x)               # 余弦函数
```

```
tan(x)              # 正切函数
asin(x)             # 反正弦函数
acos(x)             # 反余弦函数
atan(x)             # 反正切函数
sinh(x)             # 双曲正弦函数
cosh(x)             # 双曲余弦函数
tanh(x)             # 双曲正切函数
asinh(x)            # 反双曲正弦函数
acosh(x)            # 反双曲余弦函数
atanh(x)            # 反双曲正切函数
```

3. 矩阵函数

```
nrow(A)             # 返回矩阵 A 的行数
ncol(A)             # 返回矩阵 A 的列数
dim(A)              # 返回矩阵 x 的维数（几行 × 几列）
colSums(A)          # 对矩阵 A 的各列求和
rowSums(A)          # 对矩阵 A 的各行求和
colMeans(A)         # 对矩阵 A 的各列求均值
rowMeans(A)         # 对矩阵 A 的各行求均值
t(A)                # 对矩阵 A 转置
det(A)              # 计算方阵 A 的行列式
crossprod(A, B)     # 计算矩阵 A 与 B 的内积，t(A) %*% B
outer(A, B)         # 计算矩阵的外积（叉积），A %o% B
diag(x)             # 取矩阵对角线元素，或根据向量生成对角矩阵
diag(n)             # 生成 n 阶单位矩阵
solve(A)            # 求逆矩阵（要求矩阵可逆）
solve(A, B)         # 解线性方程组 AX=B
ginv(A)             # 求矩阵 A 的广义逆（Moore-Penrose 逆）
eigen()             # 返回矩阵的特征值与特征向量（列）
kronecker(A, B)     # 计算矩阵 A 与 B 的 Kronecker 积
svd(A)              # 对矩阵 A 做奇异值分解，A=UDV'
qr(A)               # 对矩阵 A 做 QR 分解：A=QR，Q 为酉矩阵，R 为阶梯形矩阵
chol(A)             # 对正定矩阵 A 做 Choleski 分解，A=P'P，P 为上三角矩阵
A[upper.tri(A)]     # 提取矩阵 A 的上三角矩阵
A[lower.tri(A)]     # 提取矩阵 A 的下三角矩阵
```

4. 概率函数

```
factorial(n)          # 计算 n 的阶乘
choose(n, k)          # 计算组合数
gamma(x)              # Gamma 函数
beta(a, b)            # beta 函数
combn(x, m)           # 生成 x 中任取 m 个元的所有组合，x 为向量或整数 n
```

```
combn(4, 2)
```

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

```
combn(c(" 甲"," 乙"," 丙"," 丁"), 2)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,] "甲" "甲" "甲" "乙" "乙" "丙"
## [2,] "乙" "丙" "丁" "丙" "丁" "丁"
```

R 中，常用的概率函数有密度函数、分布函数、分位数函数、生成随机数函数，其写法为：

- d = 密度函数 (density)
- p = 分布函数 (distribution)
- q = 分位数函数 (quantile)
- r = 生成随机数 (random)

上述 4 个字母 + 分布缩写，就构成通常的概率函数，例如：

```
dnorm(3, 0, 2)                  # 正态分布 N(0, 4) 在 3 处的密度值
```

```
## [1] 0.0647588
```

```
pnorm(1:3, 1, 2)                # N(1,4) 分布在 1,2,3 处的分布函数值
```

```
## [1] 0.5000000 0.6914625 0.8413447
```

```
# 命中率为 0.02，独立射击 400 次，至少击中两次的概率
1 - sum(dbinom(0:1, 400, 0.02))
```

```
## [1] 0.9971655
```

```
pnorm(2, 1, 2) - pnorm(0, 1, 2)  # X~N(1, 4)，求 P{0<X<=2}
```

```
## [1] 0.3829249
```

```
qnorm(1-0.025,0,1)                # N(0,1) 的上 0.025 分位数
```

```
## [1] 1.959964
```

生成随机数[8]:

```
set.seed(123)          # 设置随机种子，以重现随机结果
rnorm(5, 0, 1)         # 生成 5 个服从 N(0,1) 分布的随机数
```

```
## [1] -0.56047565 -0.23017749  1.55870831  0.07050839  0.12928774
```

表 1.6: 常用概率分布及缩写

分布名称	缩写	参数及默认值
二项分布	binom	size, prob
多项分布	multinom	size, prob
负二项分布	nbinom	size, prob
几何分布	geom	prob
超几何分布	hyper	m, n, k
泊松分布	pois	lambda
均匀分布	unif	min=0, max=1
指数分布	exp	rate=1
正态分布	norm	prob
对数正态分布	lnorm	meanlog=0, stdlog=1
t 分布	t	df
卡方分布	chisq	df
F 分布	f	df1, df2
Wilcoxon 符号秩分布	signrank	n
Wilcoxon 秩和分布	wilcox	m, n
柯西分布	cauchy	location=0, scale=1
Logistic 分布	logis	location=0, scale=1
Weibull 分布	weibull	shape, scale=1
Gamma 分布	gamma	shape, scale=1
Beta 分布	beta	shape1, shape2

随机抽样:

[8]自然界中的随机现象是真正随机发生不可重现的，计算机中模拟随机现象，包括生成随机数、随机抽样并不是真正的随机，而是可以重现的。通过设置为相同的起始种子值就可以重现，故称为“伪随机”。

sample() 函数，用来从向量中重复或非重复地随机抽样，基本格式为：

```
sample(x, size, replace = FALSE, prob)
```

- x：向量或整数；
- size：设置抽样次数；
- replace：设置是否重复抽样；
- prob：设定抽样权重。

```
set.seed(2020)
sample(c(" 正"," 反"), 10, replace=TRUE)  # 模拟抛 10 次硬币
```

```
##  [1] "反" "反" "正" "反" "反" "正" "正" "反" "反" "反"
```

```
sample(1:10, 10, replace=FALSE)          # 随机生成 1~10 的某排列
```

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

5. 统计函数

```
min(x)              # 求最小值
cummin(x)           # 求累计最小值
max(x)              # 求最大值
cummax(x)           # 求累计最大值
range(x)            # 求 x 的范围：[最小值，最大值]（向量）
sum(x)              # 求和
cumsum(x)           # 求累计和
prod(x)             # 求积
cumprod(x)          # 求累计积
mean(x)             # 求平均值
median(x)           # 求中位数
quantile(x, pr)     # 求分位数，x 为数值向量，pr 为概率值
sd(x)               # 求标准差
var(x)              # 求方差
cov(x)              # 求协方差
cor(x)              # 求相关系数
scale(x, center=TRUE, scale=FALSE)  # 对数据做中心化：减去均值
scale(x, center=TRUE, scale=TRUE)   # 对数据做标准化
```

自定义极差标准化函数：

```
rescale = function(x, type=1) {
  # type=1 正向指标，type=2 负向指标
  rng = range(x, na.rm = TRUE)
```

```
  if (type == 1) {
    (x - rng[1]) / (rng[2] - rng[1])
  } else {
    (rng[2] - x) / (rng[2] - rng[1])
  }
}

x = c(1, 2, 3, NA, 5)
rescale(x)
```

```
## [1] 0.00 0.25 0.50   NA 1.00
```

```
rescale(x, 2)
```

```
## [1] 1.00 0.75 0.50   NA 0.00
```

6. 时间序列函数

lag() 函数，用来计算时间序列的滞后，基本格式为：

```
lag(x, k, ...)
```

- x：为数值向量/矩阵或一元/多元时间序列；
- k：为滞后阶数，默认为 1.

diff() 函数，用来计算时间序列的差分，基本格式为：

```
diff(x, lag = 1, difference = 1, ...)
```

- x：为数值向量/矩阵；
- lag：为滞后阶数，默认为 1；
- difference：为差分阶数，默认为 1.

Y_t 的 j 阶滞后为 Y_{t-j}：

```
x = ts(1:8, frequency = 4, start = 2015)
x
```

```
##      Qtr1 Qtr2 Qtr3 Qtr4
## 2015    1    2    3    4
## 2016    5    6    7    8
```

```
stats::lag(x, 4)       # 避免被 dplyr::lag() 覆盖
```

```
##      Qtr1 Qtr2 Qtr3 Qtr4
## 2014    1    2    3    4
```

```
## 2015    5    6    7    8
```

Y_t 的一阶差分为 $\Delta Y_t = Y_t - Y_{t-1}$, 二阶差分为 $\Delta^2 Y_t = \Delta Y_t - \Delta Y_{t-1}, \ldots$

```
x = c(1, 3, 6, 8, 10)
x
```

```
## [1]  1  3  6  8 10
```

```
diff(x, differences = 1)
```

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

```
diff(x, differences = 2)
```

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

```
diff(x, lag = 2, differences = 1)
```

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

7. 其他函数

```
unique(x, ...)           # 返回唯一值，即去掉重复元素或观测
duplicated(x, ...)       # 判断元素或观测是否重复（多余），返回逻辑值向量
anyDuplicated(x, ...)    # 返回重复元素或观测的索引
rle(x)                   # 统计向量中连续相同值的长度
inverse.rle(x)           # rle() 的反向版本，x 为 list(lengths, values)
```

本节部分内容参阅 (G. G. Hadley Wickham 2017).

第二章 数据操作

前面章节已涵盖了 R 语言基本语法，特别是让读者训练了**向量化编程思维**（同时操作一堆数据）、**函数式编程思维**（自定义函数解决问题 + 泛函式循环迭代）。

R 语言更多的是与数据打交道，本章正式进入 tidyverse 系列，将全面讲解“管道流、整洁流”操作数据的基本语法，包括：数据读写、数据连接、数据重塑，以及各种数据操作。

本章最核心的目的是训练读者的数据思维，那么什么是数据思维？我的理解最关键的两点是：

(1) 更进一步，将向量化编程思维和函数式编程思维，纳入到数据框或更高级的数据结构中来

比如，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；函数式编程变成为想做的操作自定义函数（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。

(2) 将复杂数据操作分解为若干基本数据操作的能力

复杂数据操作都可以分解为若干简单的基本数据操作：数据重塑、筛选行、选择列、修改列、分组汇总等。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。

很多从 C 语言等过来的编程新手，有着根深蒂固地逐个元素 for 循环操作、每个计算都得“眼见为实”的习惯，这都是训练数据思维的大忌，是最应该首先摒弃的恶习。

2.1 tidyverse 简介与管道

2.1.1 tidyverse 包简介

tidyverse 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、语法、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

在 tidyverse 包的引领下，近年来涌现出一系列具体研究领域的 tidy* 版本的包：tidymodels（统计与机器学习）、mlr3verse（机器学习）、rstatix（应用统计）、tidybayes（贝叶斯模型）、tidyquant（金融）、fpp3（时间序列）、tidytext（文本挖掘）、tidygraph（网络图）、sf（空间数据分析）、tidybulk（生信）、sparklyr（大数据）等。

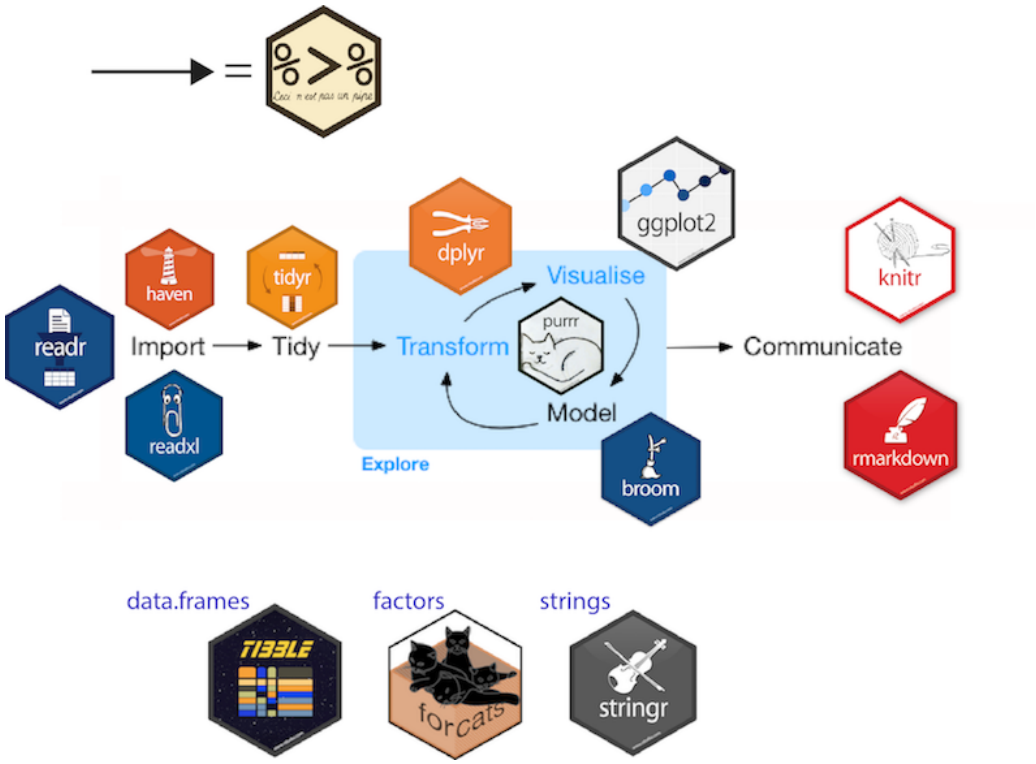

图 2.1: tidyverse 核心包

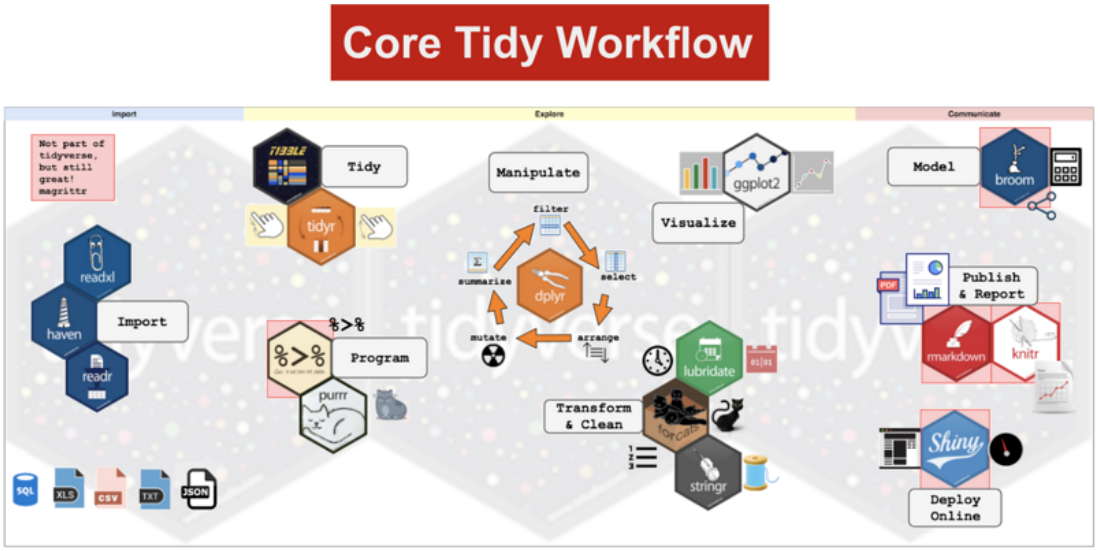

图 2.2: tidyverse 整洁工作流

tidyverse 与 data.table

tidyverse 操作数据语法优雅、容易上手，但效率与主打高效的 data.table 包不可同日而语，处理几 G 甚至十几 G 的数据，需要用 data.table.

但 data.table 的语法高度抽象、不容易上手。本书不对 data.table 做过多展开，只讲一下基本使用。另一种不错的方案是使用专门的转化包：有不少包尝试底层用 data.table，上层用 tidyverse 语法包装（转化），如 dtplyr，tidyfst 等。

2.1.2 管道操作

1. 什么是管道操作?

magrittr 包引入了管道操作，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。

例如，对数据集 mtcars，先按分类变量 cyl 分组，再对连续变量 mpg 做分组汇总计算均值：

```
library(tidyverse)
mtcars %>%
  group_by(cyl) %>%
  summarise(mpg_avg = mean(mpg))
```

```
## # A tibble: 3 x 2
##     cyl mpg_avg
##   <dbl>   <dbl>
## 1     4    26.7
## 2     6    19.7
## 3     8    15.1
```

管道运算符 %>% (Windows 快捷键：Shift+Ctrl+M) 的意思是：将左边的运算结果，以输入的方式传给右边函数。若干个函数通过管道链接起来，叫作**管道 (pipeline)**。

```
x %>% f() %>% g()                    # 等同于 g(f(x))
```

对该管道示例应该这样理解：**依次对数据进行若干操作：先对 x 进行 f 操作, 接着对结果进行 g 操作**。

管道，也支持 R base 函数：

```
month.abb %>%                        # 内置月份名缩写字符向量
  sample(6) %>%
  tolower() %>%
  str_c(collapse = "|")
```

```
## [1] "dec|jun|apr|jan|may|sep"
```

注： R 4.1 增加了同样功能的管道运算符：|>.

使用管道的好处是：

- 避免使用过多的中间变量；
- 程序可读性大大增强：

管道操作的过程，读起来就是对原数据集依次进行一系列操作的过程。而非管道操作，读起来与操作的过程是相反的，比如同样实现上例：

```
str_c(tolower(sample(month.abb, 6)), collapse="|")
```

2. 常用管道操作

管道默认将数据传给下一个函数的第 1 个参数，且它可以省略

```
c(1, 3, 4, 5, NA) %>%
  mean(., na.rm = TRUE)      # "." 可以省略
c(1, 3, 4, 5, NA) %>%
  mean(na.rm = TRUE)         # 建议写法
```

这种机制使得管道代码看起来就是：从数据开始，依次用函数对数据施加一系列的操作（变换数据），各个函数都直接从非数据参数开始写即可，而不用再额外操心数据的事情，数据会自己沿管道向前“流动”。

正是这种管道操作，使得 tidyverse 能够**优雅**地操作数据。

所以，tidyverse 中的函数都设计为数据作为第 1 个参数，自定义的函数也建议这样做。

数据可以在下一个函数中使用多次

数据经过管道默认传递给函数的第 1 个参数（通常直接省略）；若在非第 1 个参数处使用该数据，必须用“.”代替（绝对不能省略），这使得管道作用更加强大和灵活。下面看一些具体实例：

```
# 数据传递给 plot 第一个参数作为绘图数据 (. 省略),
# 同时用于拼接成字符串给 main 参数用于图形标题
c(1, 3, 4, 5) %>%
  plot(main = str_c(., collapse=","))
# 数据传递给第二个参数 data
mtcars %>% plot(mpg ~ disp, data = .)
# 选择列
iris %>% .$Species                    # 选择 Species 列内容
iris %>% .[1:3]                       # 选择 1-3 列子集
```

再来看一个更复杂的例子：分组批量建模

```
mtcars %>%
  group_split(cyl) %>%                    # . 相当于 mtcars
  map(~ lm(mpg ~ wt, data = .x))
```

split() 是将数据框 mtcars 根据其 cyl 列（包含 3 个水平的分类变量）分组，得到包含 3 个成分的列表；列表接着传递给 map(.x, .f) 的第一个参数（直接省略），~ lm(mpg ~ wt, data = .x) 是第二参数 .f，为 purrr 风格公式写法。

整体来看，实现的是分组建模：将数据框根据分类变量分组，再用 map 循环机制依次对每组数据建立线性回归模型。

> 建议进行区分：. 用于管道操作中代替数据；.x 用于 purrr 风格公式（匿名函数）。

本节部分内容参阅 (G. G. Hadley Wickham 2017), (Desi Quintans 2019).

2.2 数据读写

2.2.1 数据读写的包与函数

先来罗列一下读写常见数据文件的包和函数，具体使用可查阅其帮助。

1. readr 包

读写带分隔符的文本文件，如 csv 和 tsv; 也能读写序列化的 R 对象 rds，若想保存数据集后续再加载回来，rds 将保存元数据和该对象的状态，如分组和数据类型。

readr 2.0 版本发布，read_csv() 采用 vroom 引擎读取性能大大提升，同时支持批量读取文件。

- 读入数据到数据框：read_csv() 和 read_tsv()
- 读入欧式格式数据[1]：read_csv2() 和 read_tsv2()
- 读写 rds 数据：read_rds() 和 write_rds()
- 写出数据到文件：write_csv(), write_tsv(), write_csv2(), write_tsv2()
- 转化数据类型：parse_number(), parse_logical(), parse_factor() 等

2. readxl 包

专门读取 Excel 文件，包括同一个工作簿中的不同工作表：

- read_excel(): 自动检测 xls 或 xlsx 文件
- read_xls(): 读取 xls 文件
- read_xlsx(): 读取 xlsx 文件

读写 Excel 文件好用的包，还有 openxlsx.

[1]欧式格式数据以 ";" 为分隔符，"," 为小数位.

3. haven 包

读写 SPSS, Stata, SAS 数据:

- 读: read_spss(), read_dta(), read_sas()
- 写: write_spss(), write_stata(), write_sas()

4. jsonlite 包

读写 JSON 数据，与 R 数据结构相互转换:

- 读: read_json(), fromJSON()
- 写: write_json(), toJSON()

5. readtext 包

读取全部文本文件的内容到数据框，每个文件变成一行，常用于文本挖掘[2]或数据收集；readtext 包还支持读取 csv, tab, json, xml, html, pdf, doc, docx, rtf, xls, xlsx 等。

- readtext(): 返回数据框，doc_id 列为文档标识，text 列为读取的全部文本内容（1 个字符串）。

```
library(readtext)
document = readtext("datas/十年一觉.txt")
document
```

```
## readtext object consisting of 1 document and 0 docvars.
## # Description: df [1 x 2]
##   doc_id       text
##   <chr>        <chr>
## 1 十年一觉.txt "\"     “这位公子爷\"..."
```

2.2.2 数据读写实例

以读取 csv 和 Excel 文件为例演示，读取其他类型的数据文件，换成其他读取函数即可。

```
read_csv(file, col_names, col_types, locale, skip, na, n_max, ...)
```

- file: 数据文件所在相对或绝对路径、网址、压缩包、批量文件路径等
- col_names: 第一行是否作为列名
- skip: 开头跳过的行数
- na: 设置什么值解读为缺失值
- n_max: 读取的最大行数
- col_select: 支持 dplyr 选择列语法选择要读取的列

[2]做文本挖掘 R 包有 tidytext，中文文本挖掘相比英文多了 jiebaR 包分词的前期步骤.

- col_types: 设置列类型[3]，默认 NULL（全部猜测），可为每列单独设置，例如设置 3 列的列类型（缩写）：coltypes="cnd"
- locale: 设置区域语言环境（时区, 编码方式, 小数标记、日期格式），主要是用来设置所读取数据文件的编码方式，如从默认"UTF-8" 编码改为"GBK" 编码：locale = locale(encoding = "GBK")

还有参数 comment（忽略的注释标记），skip_empty_rows 等。

```
read_xlsx(path, sheet, range, col_names, col_types, skip, na, n_max, ...)
```

- path: 数据文件所在相对或绝对路径
- sheet: 要读取的工作表
- range: 要读取的单元格范围
- col_names: 第一行是否作为列名
- col_types: 设置列类型[4]，可总体设置一种类型（循环使用）或为每列单独设置，默认 NULL（全部猜测）

也有参数：skip, na, n_max.

> readr 包读取数据的函数，默认会保守猜测各列的列类型。若在读取数据时部分列有丢失信息，则建议先将数据以文本（字符）型读取进来，再用 dplyr 修改列类型。

1. 读取 csv 文件

```
df = read_csv("datas/六 1 班学生成绩.csv")
df
```

```
## # A tibble: 4 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
```

2. 批量读取 Excel 文件

批量读取的数据文件往往具有相同的列结构（列名、列类型），读入后紧接着需要按行合并为一个数据框。批量读取并合并，道理很简单，总共分三步：

- 获取批量数据文件的路径
- 循环机制批量读取

[3] read_csv() 可选列类型："c"（字符型），"i"（整数型），"n"（数值型），"d"（浮点型），"l"（逻辑型），"f"（因子型），"D"（日期型），"T"（日期时间型），"t"（时间型），"?"（猜测该列类型），"_ 或-"（跳过该列）.

[4] read_xlsx() 可选列类型："skip"（跳过该列），"guess"（猜测该列），"logical," "numeric," "date," "text," "list."

- 合并成一个数据文件

强大的 purrr 包，使得后两步可以同时做，即借助

```
map_dfr(.x, .f, .id)
```

将函数 .f 依次应用到序列 .x 的每个元素返回数据框，再 bind_rows 按行合并为一个数据框，.id 可用来增加新列描述来源。

比如，在 read_datas 文件夹下有 5 个 xlsx 文件，每个文件的列名都是相同的：

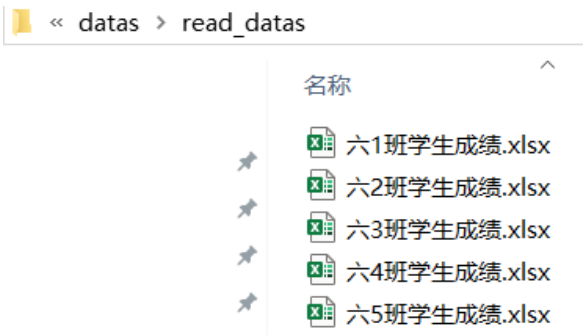

图 2.3: 批量读取的数据文件

首先要得到要导入的全部 Excel 文件的完整路径，可以任意嵌套，只需将参数 recurse 设为 TRUE:

```
files = fs::dir_ls("datas/read_datas", recurse = TRUE, glob = "*.xlsx")
files
```

```
## datas/read_datas/六1班学生成绩.xlsx
## datas/read_datas/六3班学生成绩.xlsx
## datas/read_datas/六4班学生成绩.xlsx
## datas/read_datas/六5班学生成绩.xlsx
## datas/read_datas/新建文件夹/六2班学生成绩.xlsx
```

接着，用 map_dfr() 在该路径向量上做迭代，应用 read_xlsx() 到每个文件路径，再按行合并。另外，再多做一步：用 set_names() 将文件路径字符向量创建为命名向量，再结合参数 .id 将路径值作为数据来源列。

```
library(readxl)
df = map_dfr(files, read_xlsx)
head(df)
```

```
## # A tibble: 6 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
## 6 六3班 何诗婷 女       76    53    72
```

```
# 若想增加一列表明数据来 自哪个文件
df = map_dfr(set_names(files), read_xlsx, .id = " 来源")
head(df)
```

```
## # A tibble: 6 x 7
##   来源                              班级  姓名  性别   语文  数学  英语
##   <chr>                             <chr> <chr> <chr> <dbl> <dbl> <dbl>
## 1 datas/read_datas/六1班学生成绩~ 六1班 何娜  女       87    92    79
## 2 datas/read_datas/六1班学生成绩~ 六1班 黄才~ 女       95    77    75
## 3 datas/read_datas/六1班学生成绩~ 六1班 陈芳~ 女       79    87    66
## 4 datas/read_datas/六1班学生成绩~ 六1班 陈学~ 男       82    79    66
## 5 datas/read_datas/六3班学生成绩~ 六3班 江佳~ 女       80    69    75
## 6 datas/read_datas/六3班学生成绩~ 六3班 何诗~ 女       76    53    72
```

`files` 是文件路径构成的字符向量（未命名，只有索引访问），`set_names(files)` 是将该字符向量，变成命名字符向量，名字就用元素值；参数 `.id` 定义的新列'' 来源''，将使用这些名字。

函数 `read_xlsx()` 的其他控制读取的参数，可直接‘‘作为”`map_dfr` 参数在后面添加，或改用 `purrr` 风格公式形式：

```
map_dfr(set_names(files), read_xlsx, sheet = 1, .id = " 来源")   # 或者
map_dfr(set_names(files), ~ read_xlsx(.x, sheet = 1), .id = " 来源")
```

若批量 Excel 数据是来自同一 xlsx 的多个 sheet，比如还是上述数据，只是在 ‘‘学生成绩.xlsx” 的 5 个 `sheet` 中：

	A	B	C	D	E	F
1	班级	姓名	性别	语文	数学	英语
2	六1班	何娜	女	87	92	79
3	六1班	黄才菊	女	95	77	75
4	六1班	陈芳妹	女	79	87	66
5	六1班	陈学勤	男	82	79	66
6						

六1班 | 六2班 | 六3班 | 六4班 | 六5班

图 2.4: 包含多个 sheet 的 Excel 工作簿

```
path = "datas/学生成绩.xlsx"                # Excel 文件路径
df = map_dfr(set_names(excel_sheets(path)),
             ~ read_xlsx(path, sheet = .x), .id = "sheet")
head(df)
```

```
## # A tibble: 6 x 7
##   sheet 班级  姓名   性别   语文  数学  英语
##   <chr> <chr> <chr>  <chr> <dbl> <dbl> <dbl>
## 1 六1班 六1班 何娜   女       87    92    79
## 2 六1班 六1班 黄才菊 女       95    77    75
## 3 六1班 六1班 陈芳妹 女       79    87    66
## 4 六1班 六1班 陈学勤 男       82    79    66
## 5 六2班 六2班 黄祖娜 女       94    88    75
## 6 六2班 六2班 徐雅琦 女       92    86    72
```

> excel_sheets() 函数作用在该 Excel 文件上，提取各个 sheet 名字，得到字符向量；然后同样是实现批量读取，只是这次是在 sheet 名字的字符向量上循环而已。

另外，readr 2.0 提供了非常简单方法实现批量读取 + 合并 csv 文件（列名/列类型相同）：

```
files = fs::dir_ls("datas/read_datas", recurse = TRUE, glob = "*.csv")
df = read_csv(files)
```

3. 写出到一个 Excel 文件

用 readr 包中的 write_csv() 和 write_rds()，或 writexl 包中的 write_xlsx() 可以保存数据到文件。

以写出到 Excel 文件为例：

```
library(writexl)
write_xlsx(df, "datas/output_file.xlsx")
```

4. 批量写出到 Excel 文件

比如有多个数据框，存在一个列表中，依次将它们写入文件，需要准备好文件名；在该数据框列表和文件名上，依次应用写出函数 write_xlsx()，又不需要返回值，故适合用 purrr 包中的 walk2() 函数：

```
df = iris %>%
  group_split(Species)          # 鸢尾花按组分割，得到数据框列表
files = str_c("datas/", levels(iris$Species), ".xlsx")  # 准备文件名
walk2(df, files, write_xlsx)
```

若要多个数据框分别写入一个 Excel 文件的多个 sheet，先将多个数据框创建为命名列表（名字将作为 sheet 名），再用 write_xlsx() 写出即可：

```
df = df %>%
  set_names(levels(iris$Species))
write_xlsx(df, "datas/iris.xlsx")
```

5. 保存与载入 rds 数据

除了 save() 和 load() 函数外，下面以导出数据到 .rds 文件为例，因为它能保存数据框及其元数据，如数据类型和分组等。

```
write_rds(iris, "my_iris.rds")
dat = read_rds("my_iris.rds")          # 导入.rds 数据
```

2.2.3 连接数据库

R 操作数据是先将数据载入内存，当数据超过内存限制时，可能会让您束手无策。一种解决办法是，将大数据存放在远程数据库（远程服务器或本地硬盘），然后建立与 R 的连接，再从 R 中执行查询、探索、建模等。

注意，内存可以应付的数据集，是没有必要这样操作的。

dplyr 是 tidyverse 操作数据的最核心包，而 dbplyr 包是用于数据库的 dplyr 后端，让您能够操作远程数据库中的数据表，就像它们是内存中的数据框一样。安装 dbplyr 包时，还会自动安装 DBI 包，它提供了通用的接口，使得能够使用相同的代码与许多不同的数据库连用。

常见的主流数据库软件：SQL Server，MySQL，Oracle 等都能支持，但还需要为其安装特定的驱动，比如

- RMariaDB 包：连接到 MySQL 和 MariaDB
- RPostgres 包：连接到 Postgres 和 Redshift
- RSQLite 包：嵌入 SQLite 数据库[5]
- odbc 包：通过开放数据库连接协议连接到许多商业数据库
- bigrquery 包：连接到谷歌的 BigQuery

下面以 R 连接 MySQL 数据库为例，用小数据集演示基本操作。连接其他数据库也是类似的。

(1) 配置 MySQL 开发环境

我这里是用的 MySQL zip 版 + Navicat（数据库管理工具），具体操作可参阅（知乎）八咫镜：mysql 安装及配置。

(2) 新建 MySQL 连接和数据库

在 Navicat 新建 MySQL 连接，输入连接名（随便起名）和配置 MySQL 时设好的用户名及相应密码：

打开该连接，右键新建数据库，数据库名为 mydb（随便起名），选择字符编码和排序规则：

[5] SQLlite 已经嵌入到 R 包中，是不需要额外安装数据库软件就能直接用的轻量级数据库.

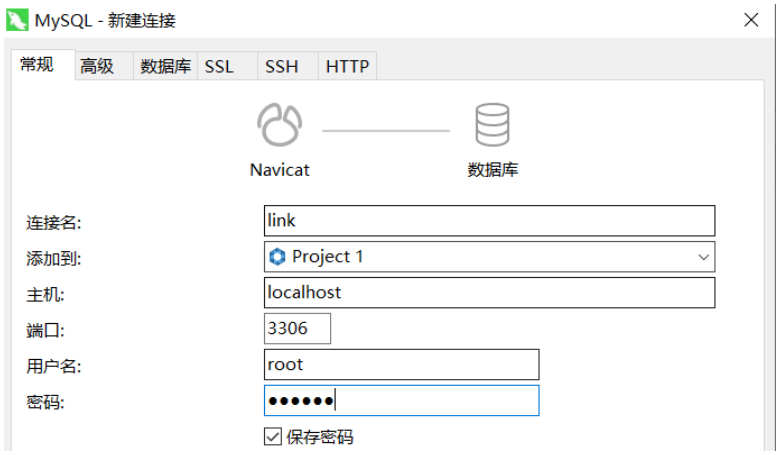

图 2.5: 在 Navicat 中新建 MySQL 连接

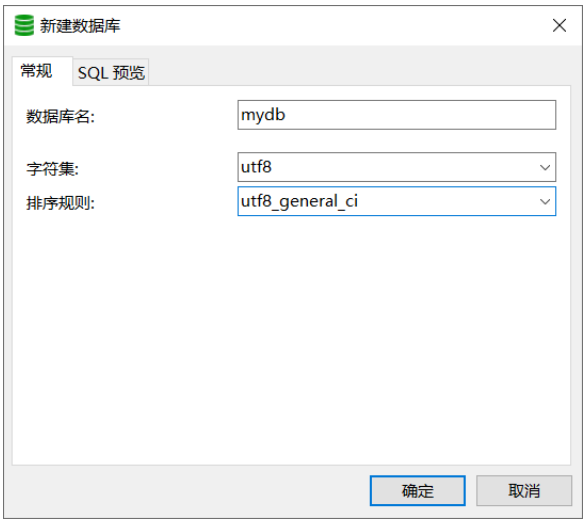

图 2.6: 新建数据库

(3) 建立 R 与 MySQL 的连接

先加载 RMariaDB 包，再用 dbConnect() 函数来建立连接，需要提供数据库后端、用户名、密码、数据库名、主机：

```
library(RMariaDB)
con = dbConnect(MariaDB(), user = "root", password = "123456",
                dbname = "mydb", host = "localhost")
dbListTables(con)     # 查看 con 连接下的数据表
```

```
## [1] "exam"
```

这表明该连接下还没有数据表。

(4) 创建数据表

在该连接下，若已有 MySQL 数据表则直接进入下一步，否则有两种方法创建数据表：

- 在 MySQL 端，从 Navicat 创建表，可从外部数据文件导入到数据表
- 在 R 端，读取数据，再通过函数 dbWriteTable() 写入到数据表；若是大数据，可以借助循环逐块地读取和追加写入。

```
datas = read_xlsx("datas/ExamDatas.xlsx")
dbWriteTable(con, name = "exam", value = datas, overwrite = TRUE)
dbListTables(con)
```

```
## [1] "exam"
```

(5) 数据表引用

用函数 tbl() 获取数据表的引用，引用是一种浅拷贝机制，能够不做物理拷贝而使用数据，一般处理大数据都采用该策略。

```
df = tbl(con, "exam")
df
```

```
## # Source:   table<exam> [?? x 8]
## # Database: mysql [root@localhost:NA/mydb]
##   class name   sex   chinese  math english moral science
##   <chr> <chr>  <chr>   <dbl> <dbl>   <dbl> <dbl>   <dbl>
## 1 六1班 何娜   女         87    92      79     9      10
## 2 六1班 黄才菊 女         95    77      75     8       9
## 3 六1班 陈芳妹 女         79    87      66     9      10
## 4 六1班 陈学勤 男         82    79      66     9      10
## 5 六1班 陈祝贞 女         76    79      67     8      10
## 6 六1班 何小薇 女         83    73      65     8       9
```

```
## # ... with more rows
```

输出数据表引用，看起来和 tibble 几乎一样，主要区别就是它是来自远程 MySQL 数据库。

(6) 数据表查询

与数据库交互，通常是用 SQL（结构化查询语言），几乎所有的数据库都在使用 SQL.

dbplyr 包让 R 用户用 dplyr 语法就能执行 SQL 查询，就像用在 R 中操作数据框一样：

```
df %>%
  group_by(sex) %>%
  summarise(avg = mean(math, na.rm = TRUE))
```

```
## # Source:   lazy query [?? x 2]
## # Database: mysql [root@localhost:NA/mydb]
##   sex     avg
##   <chr> <dbl>
## 1 女     69.1
## 2 男     65.2
```

普通数据框与远程数据库查询之间最重要的区别是，您的 R 代码被翻译成 SQL 并在远程服务器上的数据库中执行，而不是在本地机器上的 R 中执行。当与数据库一起工作时，dplyr 试图尽可能地懒惰：

- 除非明确要求（接 collect()），否则它不会把数据拉到 R 中；
- 它把任何工作都尽可能地推迟到最后一刻：把您想做的所有事情合在一起，然后一步送到数据库中。

dbplyr 包还提供了将 dplyr 代码翻译成 SQL 查询代码的函数 show_query(). 可以进一步用于 MySQL, 或 dbSendQuery(), dbGetQuery():

```
df %>%
  group_by(sex) %>%
  summarise(avg = mean(math, na.rm = TRUE)) %>%
  show_query()
```

```
## <SQL>
## SELECT `sex`, AVG(`math`) AS `avg`
## FROM `exam`
## GROUP BY `sex`
```

```
dbGetQuery(con, "SELECT `sex`, AVG(`math`) AS `avg`
                 FROM `exam`
                 GROUP BY `sex`")
```

```
##   sex      avg
## 1  女 69.11538
## 2  男 65.20833
```

最后，关闭 R 与 MySQL 的连接：

```
dbDisconnect(con)
```

2.2.4 关于中文编码

中文乱码是让很多编程者头痛的问题。

1. 什么是编码?

文字符号在计算机中是用 0 和 1 的字节序列表示的，编码就是将字节序列与所要表示的文字符号建立起映射。

要把各个国家不同的所有文字符号（字符集）正常显示和使用，需要做两件事情：

- 各个国家不同的所有文字符号一一对应地建立数字编码
- 数字编码按一定编码规则用 0-1 表示出来

第一件事情已有一种 Unicode 编码（万国码）来解决：它给全世界所有语言的所有文字符号规定了独一无二的数字编码，字符间分隔的方式是用固定长度字节数。

这样各个国家只需要做第二件事情：为自己国家的所有文字符号设计一种编码规则来表示对应的 Unicode 编码[6]。

> 从 Unicode 到各国具体编码，称为**编码过程**；从各国具体编码到 Unicode，称为**解码过程**。

再来说中国的第二件事情：汉字符号（中文）编码。历史原因产生了多种中文编码，从图来看更直观：

所谓兼容性，可以理解为子集，同时存在也不冲突。由图 2.7 可见，ASCII（128 个字母和符号，英文够用）被所有编码兼容，而最常见的 UTF-8 与 GBK 之间除了 ASCII 部分之外没有交集。

文件采用什么编码方式，就用什么编码方式打开。只要是用不兼容的编码方式打开文件，就会出现乱码，日常最容易导致乱码场景就是：

> 用 UTF-8（GBK）编码方式去读取 GBK（UTF-8）编码的文字，就会出现各种乱码

GBK（国标扩展）系列，根据包含汉字符号从少到多，依次是

- GB2312: 只包含 6763 个汉字
- GBK: 包含 20902 个汉字，基本足够用

[6] Unicode 为了表示'' 万国'' 语言，额外增大了存储开销，这第二件事也顺便节省存储开销。

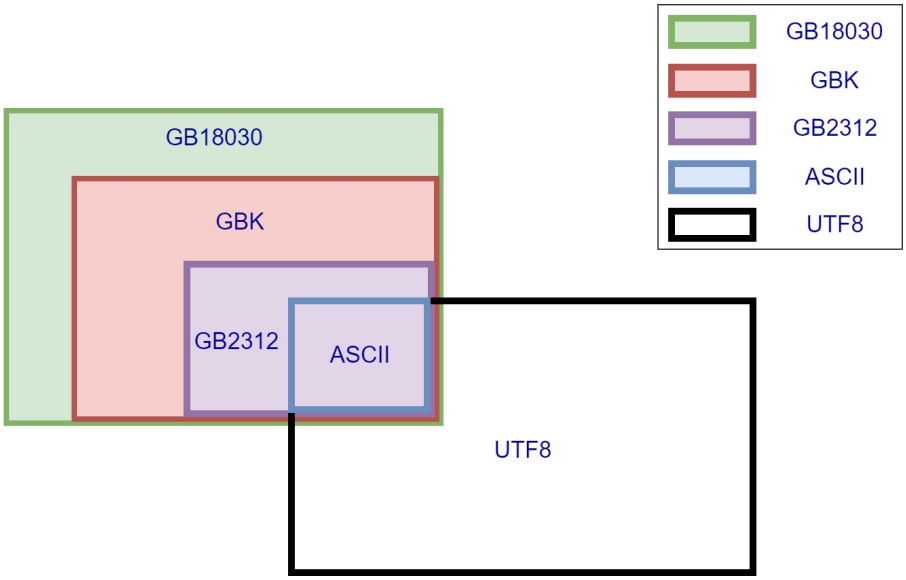

图 2.7: 几种中文编码及兼容性

- GB18030: 又分 GB1830-2000 和 GB1830-2005，包含七万多个汉字

GBK 编码的汉字基本是 2 字节，节省空间，但只适合国内中文环境。

UTF-8 编码（Unicode 转换格式），是 Unicode 的再表示，支持各个国家的文字符号，兼容性非常好。所以，目前 UTF-8 有**一统天下**的趋势。

UTF-8 是一种变长编码，解决字符间分隔的方式是通过对二进制中最高位连续 1 的个数来决定该字是几字节编码。所有常用汉字的 Unicode 值均可用 3 字节的 UTF-8 表示出来。

UTF-8 通常不带 BOM（字节序标记 EF BB BF，位于文件的前 3 个字节）也不需要带 BOM，但 Windows 历史遗留问题又会经常遇到有 BOM UTF-8 的数据文件。

其他常见的编码:

- ANSI: 不是真正的编码，而是 Windows 系统的默认编码的统称，对于简体中文系统就是 GB2312；对于繁体中文系统就是 Big5 等
- Latin1: 又称 ISO-8859-1，欧洲人发明的编码，也是 MySQL 的默认编码
- Unicode big endian: 用 UCS-2 格式存储 Unicode 时，根据两个字节谁在前谁在后，分为 Little endian（小端）和 Big endian（大端）
- UTF-16, UTF-32: Unicode 的另两种再表示，分别用 2 字节和 4 字节。

2. 中文乱码的解决办法

首先，查看并确认你的 windows 系统的默认编码方式:

```
Sys.getlocale("LC_CTYPE")       # 查看系统默认字符集类型
```

```
## [1] "Chinese (Simplified)_China.936"
```

代码 936 就表明是 ‘‘中国 - 简体中文（GB2312）’’。

注意：不建议修改系统的默认编码方式，因为可能会导致一些软件、文件乱码。

大多数中文乱码都是 GBK 与 UTF-8 不兼容导致的，常见的有两种情形。

R 文件中的中文乱码

在你的电脑不中文乱码的 R 脚本、Rmarkdown 等，拷贝到另一台电脑上时出现中文乱码。

解决办法：前文在配置 Rstudio 时已讲到，设置 code – saving 的 Default text encoding 为兼容性更好的 UTF-8.

读写数据文件中文乱码

数据文件采用什么编码方式，就用什么编码方式打开或读取。采用了不兼容的另一种编码打开或读取，肯定出现中文乱码。

下面以最常见的中文编码 GBK、UTF-8、BOM UTF-8 来讲解。

R 自带函数读取 GBK 或 UTF-8

- 与所用操作系统默认编码相同的数据文件，即 GBK，R 自带的函数 read.csv()、read.table()、readLines() 都可以正常读取但不能直接读取 UTF-8
- 但在 read.csv() 和 read.table() 中设置参数 fileEncoding = "UTF-8"，可以读取 UTF-8，但无论如何不能读取 BOM UTF-8
- 在 readLines() 中设置参数 encoding = "UTF-8"，可以读取 UTF-8 和 BOM UTF-8

```
read.csv("datas/bp-gbk.csv")          # GBK, 直接读取
read.csv("datas/bp-utf8nobom.csv",    # UTF-8, 设置参数读取
         fileEncoding = "UTF-8")

readLines("datas/bp-gbk.csv")         # GBK, 直接读取
# UTF-8 和 BOM UTF-8, 设置参数读取
readLines("datas/bp-utf8nobom.csv", encoding = "UTF-8")
readLines("datas/bp-utf8bom.csv", encoding = "UTF-8")
```

readr 包读取 GBK 或 UTF-8

- readr 包中的 read_csv()、read_table2()、read_lines() 默认读取 UTF-8 和 BOM UTF-8;
- 但不能直接读取 GBK, 需要设置参数 locale = locale(encoding="GBK")

```
read_csv("datas/bp-utf8nobom.csv")          # UTF-8, 直接读取
read_csv("datas/bp-utf8bom.csv")            # BOM UTF-8, 直接读取
read_csv("datas/bp-gbk.csv",
         locale = locale(encoding="GBK"))   # GBK, 设置参数读取
```

写入 GBK 或 UTF-8 文件

- R 自带的 write.csv(), writeLines() 仍是跟随操作系统默认编码，即默认写出为 GBK 文件；设置参数 fileEncoding = "UTF-8" 可写为 UTF-8
- readr 包中的 write_csv(), write_lines() 默认写为 UTF-8, 但不能被 Excel 软件正确打开
- readr::write_excel_csv() 可以写为 BOM UTF-8, Excel 软件能正确打开

```
write.csv(df, "file-GBK.csv")              # 写出为 GBK 文件
write.csv(df, "file-UTF8.csv",
          fileEncoding = "UTF-8")          # 写出为 UTF-8 文件

write_csv(df, "file-UTF8.csv")             # 写出为 UTF-8 文件
write_excel_csv(df, "file-BOM-UTF8.csv")   # 写出为 BOM UTF-8 文件
```

不局限于上述编码，一个数据文件只要知道了其编码方式，就可以通过在读写时指定该编码而避免乱码。那么关键的问题就是：**怎么确定一个数据文件的编码？**

AkelPad 是一款优秀开源小巧的文本编辑器，用它打开数据文件，自动在窗口下方显示文件的编码：

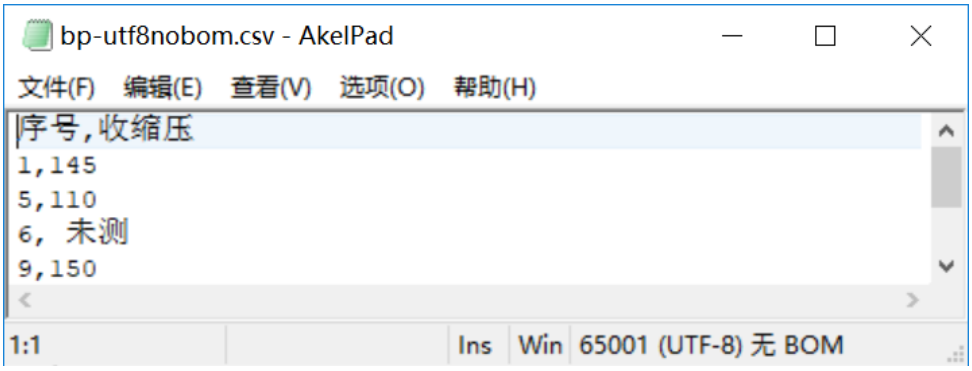

图 2.8: 用 AkelPad 检测文件编码

若要转换编码，只需要点文件另存为，在代码页下拉框选择想要的编码方式，保存即可。

另外，readr 包和 rvest 包（爬虫）都提供了函数 guess_encoding()，可检测文本和网页的编码方式；python 有一个 chardet 库在检测文件编码方面更强大。

本节部分内容参阅 (李东风 2020), (Desi Quintans 2019), RStudio: Databases using R, jagg19: Using MySQL with R, （知乎）程序员必备：彻底弄懂常见的 7 种中文字符编码。

2.3 数据连接

数据分析经常会涉及相互关联的多个数据表，称为关系数据库。关系数据库通用语言是 SQL（结构化查询语言），dplyr 包提供了一系列类似 SQL 语法的函数，可以很方便地操作关系数据库。

关系是指两个数据表之间的关系，更多数据表之间的关系总可以表示为两两之间的关系。

一个项目的数据，通常都是用若干数据表分别存放，它们之间通过“键”连接在一起，根据数据分析的需要，通过键匹配进行数据连接。

例如，纽约机场航班数据的关系结构：

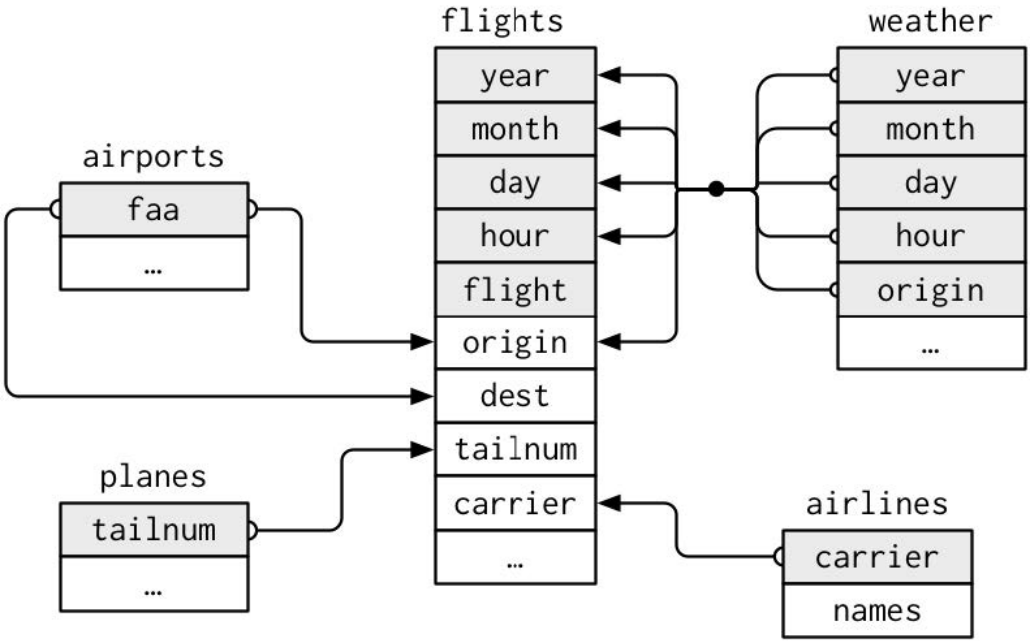

图 2.9: 数据库中数据表的关系结构示意图

比如，想要考察天气状况对航班的影响，就需要先将数据表 `flights` 和 `wheather` 根据其键值匹配连接为一个新数据表。

键列（可以不止 1 列），能够唯一识别自己或别人数据表的每一个观测。要判断某（些）列是否是键列，可以先用 `count()` 计数，再看是否没有 `n > 1` 出现：

```
load("datas/planes.rda")
planes %>%
  count(tailnum) %>%
  filter(n > 1)
```

```
## # A tibble: 0 x 2
## # ... with 2 variables: tailnum <chr>, n <int>
```

```
load("datas/weather.rda")
weather %>%
  count(year, month, day, hour, origin) %>%
  filter(n > 1)
```

```
## # A tibble: 0 x 6
## # ... with 6 variables: year <int>, month <int>, day <int>,
## #   hour <int>, origin <chr>, n <int>
```

注：不唯一匹配的列，也可以作为键列进行数据连接，只是当有“一对多”关系时，会按“多”重复生成观测，有时候这恰好是需要的。

2.3.1 合并行与合并列

合并数据框最基本的方法是：

- 合并行：下方堆叠新行，根据列名匹配列，注意列名相同，否则作为新列（NA 填充）；
- 合并列：右侧拼接新列，根据位置匹配行，行数必须相同。

分别用 dplyr 包中的 bind_rows() 和 bind_cols() 实现。

```
bind_rows(
  sample_n(iris, 2),
  sample_n(iris, 2),
  sample_n(iris, 2)
)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1          4.9         2.4          3.3         1.0 versicolor
## 2          6.1         2.9          4.7         1.4 versicolor
## 3          6.1         3.0          4.9         1.8  virginica
## 4          5.8         2.7          5.1         1.9  virginica
## 5          6.3         2.8          5.1         1.5  virginica
## 6          6.8         3.0          5.5         2.1  virginica
```

```
one = mtcars[1:4, 1:3]
two = mtcars[1:4, 4:5]
bind_cols(one, two)
```

```
##                  mpg cyl disp  hp drat
## Mazda RX4       21.0   6  160 110 3.90
## Mazda RX4 Wag   21.0   6  160 110 3.90
## Datsun 710      22.8   4  108  93 3.85
## Hornet 4 Drive 21.4   6  258 110 3.08
```

> 利用 purrr 包中 map_dfr() 和 map_dfc() 函数可以在批量读入数据的同时做合并行/合并列。

2.3.2 根据值匹配合并数据框

只介绍最常用的六种合并：**左连接、右连接、全连接、内连接、半连接、反连接**，前四种连接又称为**修改连接**，后两种连接又称为**过滤连接**。

这六种连接对应的六个接口一致的函数，其基本格式为：

```
left_join(x, y, by)
right_join(x, y, by)
full_join(x, y, by)
inner_join(x, y, by)
semi_join(x, y, by)
anti_join(x, y, by)
```

下面以 dplyr 包自带的两个小数据集进行演示：

```
band = band_members
band
```

```
## # A tibble: 3 x 2
##   name  band
##   <chr> <chr>
## 1 Mick  Stones
## 2 John  Beatles
## 3 Paul  Beatles
```

```
instrument = band_instruments
instrument
```

```
## # A tibble: 3 x 2
##   name  plays
##   <chr> <chr>
## 1 John  guitar
## 2 Paul  bass
## 3 Keith guitar
```

1. 左连接：left_join()

外连接至少保留一个数据表中的所有观测，分为左连接、右连接、全连接，其中最常用的是左连接：保留 x 所有行，合并匹配的 y 中的列。

```
band %>%
  left_join(instrument, by = "name")
```

```
## # A tibble: 3 x 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 Mick  Stones  <NA>
## 2 John  Beatles guitar
## 3 Paul  Beatles bass
```

band

name	band
Mick	Stones
John	Beatles
Paul	Beatles

+

instrument

name	plays
John	guitar
Paul	bass
Keith	guitar

=

name	band	plays
Mick	Stones	<NA>
John	Beatles	guitar
Paul	Beatles	bass

图 2.10: 左连接示意图

若两个表中的键列列名不同，用 `by = c("name1" = "name2")`；若根据多个键列匹配，用 `by = c("name1", "name2")`.

2. 右连接：`right_join()`

保留 y 所有行，合并匹配的 x 中的列。

```
band %>%
  right_join(instrument, by = "name")
```

```
## # A tibble: 3 x 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 John  Beatles guitar
## 2 Paul  Beatles bass
## 3 Keith <NA>    guitar
```

band

name	band
Mick	Stones
John	Beatles
Paul	Beatles

+

instrument

name	plays
John	guitar
Paul	bass
Keith	guitar

=

name	band	plays
John	Beatles	guitar
Paul	Beatles	bass
Keith	<NA>	guitar

图 2.11: 右连接示意图

3. 全连接：`full_join()`

保留 x 和 y 中的所有行，合并匹配的列。

```
band %>%
  full_join(instrument, by = "name")
```

```
## # A tibble: 4 x 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 Mick  Stones  <NA>
```

```
## 2 John  Beatles guitar
## 3 Paul  Beatles bass
## 4 Keith <NA>    guitar
```

band

name	band
Mick	Stones
John	Beatles
Paul	Beatles

+

instrument

name	plays
John	guitar
Paul	bass
Keith	guitar

=

name	band	plays
Mick	Stones	<NA>
John	Beatles	guitar
Paul	Beatles	bass
Keith	<NA>	guitar

图 2.12: 全连接示意图

4. 内连接：inner_join()

内连接是保留两个数据表中所共有的观测：只保留 x 中与 y 匹配的行，合并匹配的 y 中的列。

```
band %>%
  inner_join(instrument, by = "name")
```

```
## # A tibble: 2 x 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 John  Beatles guitar
## 2 Paul  Beatles bass
```

band

name	band
Mick	Stones
John	Beatles
Paul	Beatles

+

instrument

name	plays
John	guitar
Paul	bass
Keith	guitar

=

name	band	plays
John	Beatles	guitar
Paul	Beatles	bass

图 2.13: 内连接示意图

5. 半连接：semi_join()

根据在 y 中，来筛选 x 中的行。

```
band %>%
  semi_join(instrument, by = "name")
```

```
## # A tibble: 2 x 2
##   name  band
##   <chr> <chr>
## 1 John  Beatles
```

```
## 2 Paul  Beatles
```

band

name	band
Mick	Stones
John	Beatles
Paul	Beatles

\+

instrument

name	plays
John	guitar
Paul	bass
Keith	guitar

=

name	band
John	Beatles
Paul	Beatles

图 2.14: 半连接示意图

6. 反连接：anti_join()

根据不在 y 中，来筛选 x 中的行。

```
band %>%
  anti_join(instrument, by = "name")
```

```
## # A tibble: 1 x 2
##   name  band
##   <chr> <chr>
## 1 Mick  Stones
```

band

name	band
Mick	Stones
John	Beatles
Paul	Beatles

\+

instrument

name	plays
John	guitar
Paul	bass
Keith	guitar

=

name	band
Mick	Stones

图 2.15: 反连接示意图

前面讨论的都是连接两个数据表，若要连接多个数据表，将连接两个数据表的函数结合 purrr 包中的 reduce() 使用即可。

比如 achieves 文件夹有 3 个 Excel 文件：

图 2.16: 批量读取并做全连接（单独存放）

需要批量读取它们，再依次做全连接（做其他连接也是类似的）。reduce() 可以实现先将前两个表做全连接，再将结果表与第三个表做全连接（更多表就依次这样做下去）：

```
files = list.files("datas/achieves/", pattern = "xlsx", full.names = TRUE)

map(files, read_xlsx) %>%
  reduce(full_join, by = " 人名")                    # 读入并依次做全连接
```

```
## # A tibble: 7 x 4
##   人名  `3月业绩` `4月业绩` `5月业绩`
##   <chr>     <dbl>     <dbl>     <dbl>
## 1 小明         80        NA        NA
## 2 小李         85        NA        80
## 3 小张         90        50        NA
## 4 小红         NA        70        90
## 5 小白         NA        60        NA
## 6 小王         NA        40        NA
## # ... with 1 more row
```

若还是上述数据，但是在一个工作簿的多个工作表中，批量读取并依次做全连接：

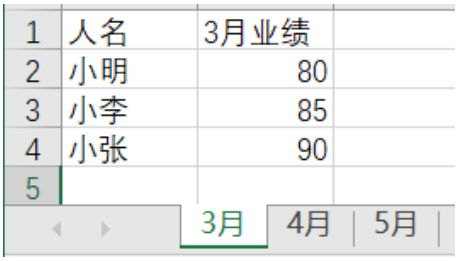

图 2.17: 批量读取并做全连接（工作簿的多个 sheet）

```
path = "datas/3-5 月业绩.xlsx"
map(excel_sheets(path),
```

```
    ~ read_xlsx(path, sheet = .x)) %>%
  reduce(full_join, by = "人名")              # 读入并依次做全连接
```

2.3.3 集合运算

集合运算有时候很有用，都是针对所有行，通过比较变量的值来实现。这就需要数据表 x 和 y 具有相同的变量，并将观测看成是集合中的元素：

```
intersect(x, y)      # 返回 x 和 y 共同包含的观测；
union(x, y)          # 返回 x 和 y 中所有的（唯一）观测；
setdiff(x, y)        # 返回在 x 中但不在 y 中的观测。
```

本节部分内容参阅 (G. G. Hadley Wickham 2017), (Amelia McNamara 2020), (Desi Quintans 2019).

2.4 数据重塑

2.4.1 什么是整洁数据?

采用 Hadley 的表述，脏的/不整洁的数据往往具有如下特点：

- 首行（列名）是值，不是变量名
- 多个变量放在一列
- 变量既放在行也放在列
- 多种类型的观测单元在同一个单元格
- 一个观测单元放在多个表

而整洁数据具有如下特点：

- 每个**变量**构成一列
- 每个**观测**构成一行
- 每个观测的每个变量**值**构成一个单元格

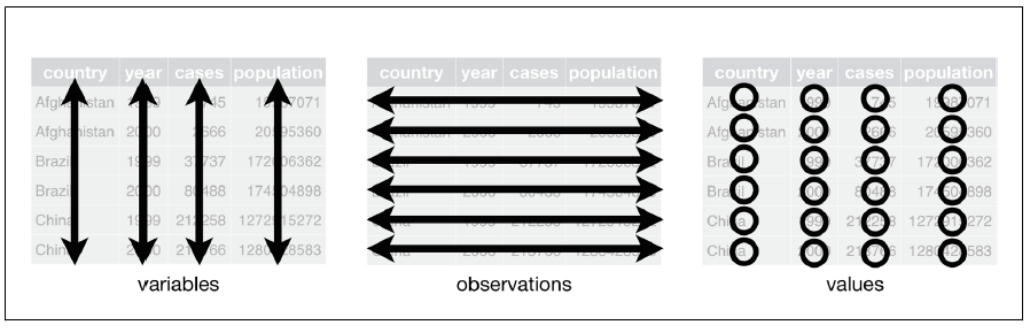

图 2.18: 整洁数据的 3 个特点

`tidyverse` 系列包中的函数操作的都是这种整洁数据框，而不整洁数据，首先需要变成整洁数据，这就是**数据重塑**。

数据重塑主要包括长宽表转化、拆分/合并列。长宽表转化最初是 `reshape2` 包的 `melt()` 和 `cast()`; 后来又发展到早期 `tidyr` 包的 `gather()` 和 `spread()`, 现在 `tidyr 1.0` 之后提供了更加易用的 `pivot_longer()` 和 `pivot_wider()`.

先看一个不整洁数据与整洁数据对比的例子:

```
dt = tribble(
  ~observation, ~A_count, ~B_count, ~A_dbh, ~B_dbh,
  "Richmond(Sam)",   7,       2,      100,    110,
  "Windsor(Ash)",   10,       5,      80,      87,
  "Bilpin(Jules)",   5,       8,      95,      90)
knitr::kable(dt, align="c")
```

observation	A_count	B_count	A_dbh	B_dbh
Richmond(Sam)	7	2	100	110
Windsor(Ash)	10	5	80	87
Bilpin(Jules)	5	8	95	90

该数据框不整洁，表现在:

- `observation` 列有两个变量数据
- 列名中的 `A/B` 应是分类变量 `species` 的两个水平值
- 测量值列 `count` 和 `dbh` 应各占 1 列，而不是 2 列

下面借助 `tidyr` 包中数据重塑函数，将其变成整洁数据，读者可以学完本节内容再回头看这段代码:

```
tidy_dt = dt %>%
  pivot_longer(-observation,
               names_to = c("speices", ".value"),
               names_sep = "_") %>%
  separate(observation, into = c("site", "surveyor"))
knitr::kable(tidy_dt, align = "c")
```

site	surveyor	speices	count	dbh
Richmond	Sam	A	7	100
Richmond	Sam	B	2	110
Windsor	Ash	A	10	80
Windsor	Ash	B	5	87
Bilpin	Jules	A	5	95
Bilpin	Jules	B	8	90

这里的关键是，要学会区分哪些是**变量、观测、值**。

2.4.2 宽表变长表

宽表的特点是：表比较宽，本来该是“值”的，却出现在“变量（名）”中。

这就需要给它变到“值”中，新起个列名存为一列，这就是所谓的宽表变长表。

用 tidyr 包中的 pivot_longer() 函数来实现宽表变长表，其基本格式为：

```
pivot_longer(data, cols, names_to, values_to, values_drop_na, ...)
```

- data: 要重塑的数据框
- cols: 用选择列语法选择要变形的列
- names_to: 为存放变形列的列名中的’’ 值’’，指定新列名
- values_to: 为存放变形列中的’’ 值’’，指定新列名
- values_drop_na: 是否忽略变形列中的 NA

若变形列的列名除了’’ 值’’ 外，还包含前缀、变量名 + 分隔符、正则表达式分组捕获模式，则可以借助参数 names_prefix，names_sep，names_pattern 来提取出“值”。

1. 每一行只有 1 个观测的情形

也是最简单的情形，以分省年度 GDP 数据为例，每一行只有一个观测，关于一个省份的信息。

```
df = read_csv("datas/分省年度 GDP.csv")
df
```

```
## # A tibble: 4 x 4
##   地区     `2019年` `2018年` `2017年`
##   <chr>       <dbl>    <dbl>    <dbl>
## 1 北京市      35371.   33106.   28015.
## 2 天津市      14104.   13363.   18549.
## 3 河北省      35105.   32495.   34016.
## 4 黑龙江省    13613.   12846.   15903.
```

- 要变形的列是除了 地区列之外的列
- 变量（名）中的 2019 年、2018 年等是年份的值，需要作为 1 列“值”来存放，新起一个列名 年份
- 2019 年、2018 年等列中的值，属于同一个变量 GDP，新起一个列名 GDP 来存放：

```
df %>%
  pivot_longer(-地区, names_to = " 年份", values_to = "GDP")
```

```
## # A tibble: 12 x 3
##   地区    年份       GDP
```

```
##   <chr>  <chr>   <dbl>
## 1 北京市 2019年 35371.
## 2 北京市 2018年 33106.
## 3 北京市 2017年 28015.
## 4 天津市 2019年 14104.
## 5 天津市 2018年 13363.
## 6 天津市 2017年 18549.
## # ... with 6 more rows
```

2. 每一行有多个观测的情形

以如下的 family 数据集为例，每一行有两个观测，关于 child1 和 child2 的信息。

```
load("datas/family.rda")
knitr::kable(family, align = "c")
```

family	dob_child1	dob_child2	gender_child1	gender_child2
1	1998-11-26	2000-01-29	1	2
2	1996-06-22	NA	2	NA
3	2002-07-11	2004-04-05	2	2
4	2004-10-10	2009-08-27	1	1
5	2000-12-05	2005-02-28	2	1

- 要变形的列是除了 family 列之外的列；
- 变形列的列名以 “_” 分割为两部分，用 names_to 指定这两部分的用途：“.value” 指定第一部分不用管将继续留作列名，而第二部分，即包含 “child1’’、“child2’’，作为新变量 child 的 ‘‘值”
- 忽略变形列中的缺失值

```
family %>%
  pivot_longer(-family,
               names_to = c(".value", "child"),
               names_sep = "_",
               values_drop_na = TRUE)
```

```
## # A tibble: 9 x 4
##   family child  dob        gender
##    <int> <chr>  <date>      <int>
## 1      1 child1 1998-11-26      1
## 2      1 child2 2000-01-29      2
## 3      2 child1 1996-06-22      2
## 4      3 child1 2002-07-11      2
## 5      3 child2 2004-04-05      2
```

```
## 6      4 child1 2004-10-10      1
## # ... with 3 more rows
```

再来看一个数学建模报名信息整理的实例：每一行有 3 个观测，关于 3 名队员的信息，变成每一行只有 1 名队员的信息。用到 names_pattern 参数和正则表达式分组捕获。

```
df = read_csv("datas/参赛队信息.csv")
df
```

```
## # A tibble: 2 x 6
##   队员1姓名 队员1专业 队员2姓名 队员2专业 队员3姓名 队员3专业
##   <chr>     <chr>     <chr>     <chr>     <chr>     <chr>
## 1 张三      数学      李四      英语      王五      统计学
## 2 赵六      经济学    钱七      数学      孙八      计算机
```

```
df %>%
  pivot_longer(everything(),
    names_to = c(" 队员", ".value"),
    names_pattern = "(.*\\d)(.*)")
```

```
## # A tibble: 6 x 3
##   队员  姓名  专业
##   <chr> <chr> <chr>
## 1 队员1 张三  数学
## 2 队员2 李四  英语
## 3 队员3 王五  统计学
## 4 队员1 赵六  经济学
## 5 队员2 钱七  数学
## 6 队员3 孙八  计算机
```

2.4.3 长表变宽表

长表的特点是：表比较长。

有时候需要将分类变量的若干水平值，变成变量（列名）。这就是长表变宽表，它与宽表变长表正好相反（二者互逆）。

用 tidyr 包中的 pivot_wider() 函数来实现长表变宽表，其基本格式为：

```
pivot_wider(data, id_cols, names_from, values_from, values_fill, ...)
```

- data: 要重塑的数据框
- id_cols: 唯一识别观测的列，默认是除了 names_from 和 values_from 指定列之外的列
- names_from: 指定列名来自哪个变量列
- values_from: 指定列'' 值'' 来自哪个变量列

- values_fill: 若变宽后单元格值缺失，设置用何值填充

另外还有若干帮助修复列名的参数：names_prefix，names_sep，names_glue.

最简单的情形是，只有一个列名列和一个值列，比如 animals 数据集：

```
load("datas/animals.rda")
animals
```

```
## # A tibble: 228 x 3
##   Type    Year  Heads
##   <chr>  <dbl>  <dbl>
## 1 Sheep   2015 24943.
## 2 Cattle  1972  2189.
## 3 Camel   1985   559
## 4 Camel   1995   368.
## 5 Camel   1997   355.
## 6 Goat    1977  4411.
## # ... with 222 more rows
```

用 names_form 指定列名来自哪个变量；values_from 指定 ‘‘值” 来自哪个变量：

```
animals %>%
  pivot_wider(names_from = Type, values_from = Heads, values_fill = 0)
```

```
## # A tibble: 48 x 6
##    Year  Sheep Cattle Camel   Goat Horse
##   <dbl>  <dbl>  <dbl> <dbl>  <dbl> <dbl>
## 1  2015 24943.  3780.  368. 23593. 3295.
## 2  1972 13716.  2189.  625.  4338. 2239.
## 3  1985 13249.  2408.  559   4299. 1971
## 4  1995     0   3317.  368.  8521. 2684.
## 5  1997 14166.  3613.  355. 10265. 2893.
## 6  1977 13430.  2388.  609   4411. 2104.
## # ... with 42 more rows
```

还可以有多个列名列或多个值列，比如 us_rent_income 数据集有两个值列：

```
us_rent_income
```

```
## # A tibble: 104 x 5
##   GEOID NAME    variable estimate   moe
##   <chr> <chr>   <chr>       <dbl> <dbl>
## 1 01    Alabama income      24476   136
## 2 01    Alabama rent          747     3
```

```
## 3 02    Alaska  income      32940   508
## 4 02    Alaska  rent         1200    13
## 5 04    Arizona income      27517   148
## 6 04    Arizona rent          972     4
## # ... with 98 more rows
```

```
us_rent_income %>%
  pivot_wider(names_from = variable, values_from = c(estimate, moe))
```

```
## # A tibble: 52 x 6
##   GEOID NAME       estimate_income estimate_rent moe_income moe_rent
##   <chr> <chr>                <dbl>         <dbl>      <dbl>    <dbl>
## 1 01    Alabama              24476           747        136        3
## 2 02    Alaska               32940          1200        508       13
## 3 04    Arizona              27517           972        148        4
## 4 05    Arkansas             23789           709        165        5
## 5 06    California           29454          1358        109        3
## 6 08    Colorado             32401          1125        109        5
## # ... with 46 more rows
```

长变宽时，经常会遇到两个问题：

- 长变宽正常会压缩行，为什么行数没变呢？
- 值不能被唯一识别，输出将包含列表列

比如，现有这样的数据：

```
df = tibble(
  x = 1:6,
  y = c("A","A","B","B","C","C"),
  z = c(2.13,3.65,1.88,2.30,6.55,4.21))
df
```

```
## # A tibble: 6 x 3
##       x y         z
##   <int> <chr> <dbl>
## 1     1 A      2.13
## 2     2 A      3.65
## 3     3 B      1.88
## 4     4 B      2.3
## 5     5 C      6.55
## 6     6 C      4.21
```

想让 y 列提供变量名，z 列提供值，做长变宽，但是

```
df %>%
  pivot_wider(names_from = y, values_from = z)
```

```
## # A tibble: 6 x 4
##       x     A     B     C
##   <int> <dbl> <dbl> <dbl>
## 1     1  2.13 NA    NA
## 2     2  3.65 NA    NA
## 3     3 NA     1.88 NA
## 4     4 NA     2.3  NA
## 5     5 NA    NA     6.55
## 6     6 NA    NA     4.21
```

这就是前面说到的第一个问题，本来该压缩成 2 行，但是由于 x 列的存在，无法压缩，只能填充 NA，这不是想要的效果。所以，在长变宽时要注意，是不能带着类似 x 列这种唯一识别各行的 ID 列的。

那去掉 x 列，重新做长变宽，但是又遇到了前面说的第二个问题：

```
df = df[-1]
df %>%
  pivot_wider(names_from = y, values_from = z)
```

```
## # A tibble: 1 x 3
##   A         B         C
##   <list>    <list>    <list>
## 1 <dbl [2]> <dbl [2]> <dbl [2]>
```

值不能唯一识别，结果变成了列表列，同样不是想要的结果。

这里的值唯一识别，指的是各分组（A 组 B 组 C 组）组内元素必须要能唯一识别。咱们来增加一个各组的唯一识别列：

```
df = df %>%
  group_by(y) %>%
  mutate(n = row_number())
df
```

```
## # A tibble: 6 x 3
## # Groups:   y [3]
##   y         z     n
##   <chr> <dbl> <int>
## 1 A      2.13     1
## 2 A      3.65     2
## 3 B      1.88     1
```

```
## 4 B      2.3      2
## 5 C      6.55     1
## 6 C      4.21     2
```

这才是能够长变宽的标准数据，再来做长变宽：

```
df %>%
  pivot_wider(names_from = y, values_from = z)
```

```
## # A tibble: 2 x 4
##       n     A     B     C
##   <int> <dbl> <dbl> <dbl>
## 1     1  2.13  1.88  6.55
## 2     2  3.65  2.3   4.21
```

这回是想要的结果，新增加的列 n 若不想要，删除列即可。

回头再看一下，所谓的各组内值唯一识别，比如 A 组有两个数 2.13 和 3.65, 给了它们唯一识别: n = 1 和 n = 2, 当然 1 和 2 换成其他的两个不同值也是一样的，这样就知道谁作为第一个样本（行），谁作为第二个样本（行）。否则两个数无法区分，只能放到一个列表里了，就是前面的错误结果 + 警告。

最后再看一个特殊的实例：不规则通讯录整理。

```
contacts = tribble(  ~field, ~value,
                     " 姓名", " 张三",
                     " 公司", " 百度",
                     " 姓名", " 李四",
                     " 公司", " 腾讯",
                     "Email", "Lisi@163.com",
                     " 姓名", " 王五")
contacts = contacts %>%
  mutate(ID = cumsum(field == " 姓名"))
contacts
```

```
## # A tibble: 6 x 3
##   field value           ID
##   <chr> <chr>        <int>
## 1 姓名  张三             1
## 2 公司  百度             1
## 3 姓名  李四             2
## 4 公司  腾讯             2
## 5 Email Lisi@163.com     2
## 6 姓名  王五             3
```

```
contacts %>%
  pivot_wider(names_from = field, values_from = value)
```

```
## # A tibble: 3 x 4
##      ID 姓名  公司  Email
##   <int> <chr> <chr> <chr>
## 1     1 张三  百度  <NA>
## 2     2 李四  腾讯  Lisi@163.com
## 3     3 王五  <NA>  <NA>
```

2.4.4 拆分列与合并列

拆分列与合并列也是正好相反（二者互逆）。

用 separate() 函数来拆分列，其基本语法为：

```
separate(data, col, into, sep, ...)
```

- col: 要拆分的列
- into: 拆开的新列，
- sep: 指定根据什么分隔符拆分

```
table3
```

```
## # A tibble: 6 x 3
##   country      year rate
## * <chr>       <int> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## 4 Brazil       2000 80488/174504898
## 5 China        1999 212258/1272915272
## 6 China        2000 213766/1280428583
```

```
table3 %>%
  separate(rate, into = c("cases", "population"), sep = "/",
           convert = TRUE)        # 同时转化为数值型
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
```

```
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

还有 `separate_rows()` 函数，可对不定长的列进行分列，并按行堆叠放置：

```
df = tibble(Class = c("1 班", "2 班"),
            Name = c(" 张三, 李四, 王五", " 赵六, 钱七"))
df
```

```
## # A tibble: 2 x 2
##   Class Name
##   <chr> <chr>
## 1 1班   张三, 李四, 王五
## 2 2班   赵六, 钱七
```

```
df1 = df %>%
  separate_rows(Name, sep = ", ")
df1
```

```
## # A tibble: 5 x 2
##   Class Name
##   <chr> <chr>
## 1 1班   张三
## 2 1班   李四
## 3 1班   王五
## 4 2班   赵六
## 5 2班   钱七
```

若要逆操作还原回去：

```
df1 %>%
  group_by(Class) %>%
  summarise(Name = str_c(Name, collapse = ", "))
```

用 `unite()` 函数来合并列，其基本语法为：

```
unite(data, col, sep, ...)
```

- `col`: 要合并的列
- `sep`: 指定合并各列添加的分隔符

```
table5
```

```
## # A tibble: 6 x 4
##   country     century year  rate
```

```
## * <chr>       <chr>   <chr> <chr>
## 1 Afghanistan 19      99    745/19987071
## 2 Afghanistan 20      00    2666/20595360
## 3 Brazil      19      99    37737/172006362
## 4 Brazil      20      00    80488/174504898
## 5 China       19      99    212258/1272915272
## 6 China       20      00    213766/1280428583
```

```
table5 %>%
  unite(new, century, year, sep = "")
```

```
## # A tibble: 6 x 3
##   country     new   rate
##   <chr>       <chr> <chr>
## 1 Afghanistan 1999  745/19987071
## 2 Afghanistan 2000  2666/20595360
## 3 Brazil      1999  37737/172006362
## 4 Brazil      2000  80488/174504898
## 5 China       1999  212258/1272915272
## 6 China       2000  213766/1280428583
```

最后看一个综合示例：重塑世界银行人口数据。

```
world_bank_pop
```

```
## # A tibble: 1,056 x 20
##   country indicator       `2000` `2001` `2002` `2003`  `2004`  `2005`
##   <chr>   <chr>            <dbl>  <dbl>  <dbl>  <dbl>   <dbl>   <dbl>
## 1 ABW     SP.URB.TOTL   42444    4.30e4 4.37e4 4.42e4 4.47e+4 4.49e+4
## 2 ABW     SP.URB.GROW       1.18 1.41e0 1.43e0 1.31e0 9.51e-1 4.91e-1
## 3 ABW     SP.POP.TOTL   90853    9.29e4 9.50e4 9.70e4 9.87e+4 1.00e+5
## 4 ABW     SP.POP.GROW       2.06 2.23e0 2.23e0 2.11e0 1.76e+0 1.30e+0
## 5 AFG     SP.URB.TOTL 4436299    4.65e6 4.89e6 5.16e6 5.43e+6 5.69e+6
## 6 AFG     SP.URB.GROW       3.91 4.66e0 5.13e0 5.23e0 5.12e+0 4.77e+0
## # ... with 1,050 more rows, and 12 more variables: 2006 <dbl>,
## #   2007 <dbl>, 2008 <dbl>, 2009 <dbl>, 2010 <dbl>, 2011 <dbl>,
## #   2012 <dbl>, 2013 <dbl>, 2014 <dbl>, 2015 <dbl>, 2016 <dbl>,
## #   2017 <dbl>
```

先从最显然的入手：年份跨过了多个列，应该宽表变长表：

```
pop2 = world_bank_pop %>%
  pivot_longer(`2000`:`2017`, names_to = "year", values_to = "value")
```

```
pop2
```

```
## # A tibble: 19,008 x 4
##   country indicator   year  value
##   <chr>   <chr>       <chr> <dbl>
## 1 ABW     SP.URB.TOTL 2000  42444
## 2 ABW     SP.URB.TOTL 2001  43048
## 3 ABW     SP.URB.TOTL 2002  43670
## 4 ABW     SP.URB.TOTL 2003  44246
## 5 ABW     SP.URB.TOTL 2004  44669
## 6 ABW     SP.URB.TOTL 2005  44889
## # ... with 19,002 more rows
```

再来考虑 `indicator` 变量：

```
pop2 %>%
  count(indicator)
```

```
## # A tibble: 4 x 2
##   indicator       n
##   <chr>       <int>
## 1 SP.POP.GROW  4752
## 2 SP.POP.TOTL  4752
## 3 SP.URB.GROW  4752
## 4 SP.URB.TOTL  4752
```

这里，`SP.POP.GROW` 为人口增长率，`SP.POP.TOTAL` 为总人口，`SP.URB.*` 也类似，只是城市的。将该列值拆分为两个变量：`area`（`URB`, `POP`）和 `variable`（`GROW`, `TOTL`）：

```
pop3 = pop2 %>%
  separate(indicator, c(NA, "area", "variable"))
pop3
```

```
## # A tibble: 19,008 x 5
##   country area  variable year  value
##   <chr>   <chr> <chr>    <chr> <dbl>
## 1 ABW     URB   TOTL     2000  42444
## 2 ABW     URB   TOTL     2001  43048
## 3 ABW     URB   TOTL     2002  43670
## 4 ABW     URB   TOTL     2003  44246
## 5 ABW     URB   TOTL     2004  44669
## 6 ABW     URB   TOTL     2005  44889
## # ... with 19,002 more rows
```

最后，再将分类变量 variable 的水平值变为列名（长表变宽表），就完成重塑：

```
pop3 %>%
  pivot_wider(names_from = variable, values_from = value)
```

```
## # A tibble: 9,504 x 5
##   country area  year   TOTL  GROW
##   <chr>   <chr> <chr> <dbl> <dbl>
## 1 ABW     URB   2000  42444 1.18
## 2 ABW     URB   2001  43048 1.41
## 3 ABW     URB   2002  43670 1.43
## 4 ABW     URB   2003  44246 1.31
## 5 ABW     URB   2004  44669 0.951
## 6 ABW     URB   2005  44889 0.491
## # ... with 9,498 more rows
```

2.4.5 方形化

方形化（Rectangling）是将一个深度嵌套的列表（通常来自 JSON 或 XML）驯服成一个整齐的行和列的数据集。主要通过组合使用以下函数实现：

- unnest_longer()：提取列表列的每个元，再按行存放（横向展开）
- unnest_wider()：提取列表列的每个元，再按列存放（纵向展开）
- unnest_auto()：提取列表列的每个元，猜测按行或按列存放
- hoist()：类似 unnest_wider()，但只取出选择的组件，且可以深入多个层

以权力游戏角色数据集 got_chars 为例，它是个长度为 30 的列表，里面又嵌套很多列表。一种技巧是，先把它创建成 tibble 方便后续操作：

```
library(repurrrsive)    # 使用 got_chars 数据集
chars = tibble(char = got_chars)
chars
```

```
## # A tibble: 30 x 1
##   char
##   <list>
## 1 <named list [18]>
## 2 <named list [18]>
## 3 <named list [18]>
## 4 <named list [18]>
## 5 <named list [18]>
## 6 <named list [18]>
## # ... with 24 more rows
```

char 是嵌套列表列，每个元素又是长度为 18 的列表，先横向展开它们：

```
chars1 = chars %>%
  unnest_wider(char)
chars1
```

```
## # A tibble: 30 x 18
##   url       id name   gender culture born  died  alive titles aliases
##   <chr>  <int> <chr>  <chr>  <chr>   <chr> <chr> <lgl> <list> <list>
## 1 https~  1022 Theon~ Male   "Ironb~ "In ~ ""    TRUE  <chr ~ <chr [~
## 2 https~  1052 Tyrio~ Male   ""      "In ~ ""    TRUE  <chr ~ <chr [~
## 3 https~  1074 Victa~ Male   "Ironb~ "In ~ ""    TRUE  <chr ~ <chr [~
## 4 https~  1109 Will   Male   ""      ""    "In ~ FALSE <chr ~ <chr [~
## 5 https~  1166 Areo ~ Male   "Norvo~ "In ~ ""    TRUE  <chr ~ <chr [~
## 6 https~  1267 Chett  Male   ""      "At ~ "In ~ FALSE <chr ~ <chr [~
## # ... with 24 more rows, and 8 more variables: father <chr>,
## #   mother <chr>, spouse <chr>, allegiances <list>, books <list>,
## #   povBooks <list>, tvSeries <list>, playedBy <list>
```

生成一个表，以匹配人物角色和他们的外号，name 直接选择列，外号来自列表列 titles，纵向展开它：

```
chars1 %>%
  select(name, title = titles) %>%
  unnest_longer(title)
```

```
## # A tibble: 60 x 2
##   name              title
##   <chr>             <chr>
## 1 Theon Greyjoy     Prince of Winterfell
## 2 Theon Greyjoy     Captain of Sea Bitch
## 3 Theon Greyjoy     Lord of the Iron Islands (by law of the green la~
## 4 Tyrion Lannister  Acting Hand of the King (former)
## 5 Tyrion Lannister  Master of Coin (former)
## 6 Victarion Greyjoy Lord Captain of the Iron Fleet
## # ... with 54 more rows
```

或者改用 hoist() 直接从内层提取想要的列，再对列表列 title 做纵向展开：

```
chars %>%
  hoist(char, name = "name", title = "titles") %>%
  unnest_longer(title)
```

```
## # A tibble: 60 x 3
```

```
##   name              title                                char
##   <chr>             <chr>                                <list>
## 1 Theon Greyjoy     Prince of Winterfell                 <named list~
## 2 Theon Greyjoy     Captain of Sea Bitch                 <named list~
## 3 Theon Greyjoy     Lord of the Iron Islands (by law of~ <named list~
## 4 Tyrion Lannister  Acting Hand of the King (former)     <named list~
## 5 Tyrion Lannister  Master of Coin (former)              <named list~
## 6 Victarion Greyjoy Lord Captain of the Iron Fleet       <named list~
## # ... with 54 more rows
```

另外，还有 `tibblify` 包专门做嵌套列表转化为 `tibble` 数据框。

本节部分内容参阅 (G. G. Hadley Wickham 2017), (Desi Quintans 2019), `Vignettes of tidyr`.

2.5 数据操作

用 `dplyr` 包实现各种数据操作，通常的数据操作无论多么复杂，往往都可以分解为若干基本数据操作步骤的组合。

共有 5 种基本数据操作：

- `select()` —— 选择列
- `filter()/slice()` —— 筛选行
- `arrange()` —— 对行排序
- `mutate()` —— 修改列/创建新列
- `summarize()` —— 汇总

这些函数都可以与

- `group_by()` —— 分组

连用，以改变数据操作的作用域：作用在整个数据框，或数据框的每个分组。

这些函数组合使用就足以完成各种数据操作，它们的相同之处是：

- 第 1 个参数是数据框，方便管道操作
- 根据列名访问数据框的列，且列名不用加引号
- 返回结果是一个新数据框，不改变原数据框

从而，可以方便地实现：“将多个简单操作，依次用管道连接，实现复杂的数据操作”。

另外，若要同时对所选择的多列应用函数，还有强大的 `across()` 函数，它支持各种**选择列语法**，搭配 `mutate()` 和 `summarise()` 使用，产生非常强大同时修改/汇总多列的效果；类似地，`dplyr` 包又提供了 `if_any()`, `if_all()` 函数，搭配 `filter()` 使用，产生强大的根据多列的值筛选行的效果。

2.5.1 选择列

选择列，包括对数据框做选择列、调整列序、重命名列。

下面以虚拟的学生成绩数据来演示，包含随机生成的 20 个 NA：

```
df = read_xlsx("datas/ExamDatas_NAs.xlsx")
df
```

```
## # A tibble: 50 x 8
##    class name   sex   chinese  math english moral science
##    <chr> <chr>  <chr>   <dbl> <dbl>   <dbl> <dbl>   <dbl>
## 1 六1班 何娜   女         87    92      79     9      10
## 2 六1班 黄才菊 女         95    77      75    NA       9
## 3 六1班 陈芳妹 女         79    87      66     9      10
## 4 六1班 陈学勤 男         NA    79      66     9      10
## 5 六1班 陈祝贞 女         76    79      67     8      10
## 6 六1班 何小薇 女         83    73      65     8       9
## # ... with 44 more rows
```

1. 选择列语法

(1) 用列名或索引选择列

```
df %>%
  select(name, sex, math)   # 或者 select(2, 3, 5)
```

```
## # A tibble: 50 x 3
##    name   sex    math
##    <chr>  <chr> <dbl>
## 1 何娜   女       92
## 2 黄才菊 女       77
## 3 陈芳妹 女       87
## 4 陈学勤 男       79
## 5 陈祝贞 女       79
## 6 何小薇 女       73
## # ... with 44 more rows
```

(2) 借助运算符选择列

- 用: 选择连续的若干列
- 用! 选择变量集合的余集（反选）
- & 和 | 选择变量集合的交或并
- c() 合并多个选择

(3) 借助选择助手函数

- **选择指定列**:
  - `everything()`: 选择所有列
  - `last_col()`: 选择最后一列，可以带参数，如 `last_col(5)` 选择倒数第 6 列
- **选择列名匹配的列**:
  - `starts_with()`: 以某前缀开头的列名
  - `ends_with()`: 以某后缀结尾的列名
  - `contains()`: 包含某字符串的列名
  - `matches()`: 匹配正则表达式的列名
  - `num_range()`: 匹配数值范围的列名，如 `num_range("x", 1:3)` 匹配 `x1, x2, x3`
- **结合函数选择列**:
  - `where()`: 应用一个函数到所有列，选择返回结果为 `TRUE` 的列，比如与 `is.numeric` 等函数连用

2. 一些选择列的示例

```
df %>%
  select(starts_with("m"))
```

```
## # A tibble: 50 x 2
##    math moral
##   <dbl> <dbl>
## 1    92     9
## 2    77    NA
## 3    87     9
## 4    79     9
## 5    79     8
## 6    73     8
## # ... with 44 more rows
```

```
df %>%
  select(ends_with("e"))
```

```
## # A tibble: 50 x 3
##   name   chinese science
##   <chr>    <dbl>   <dbl>
## 1 何娜        87      10
## 2 黄才菊      95       9
## 3 陈芳妹      79      10
## 4 陈学勤      NA      10
## 5 陈祝贞      76      10
```

```
## 6 何小薇      83       9
## # ... with 44 more rows
```

```
df %>%
  select(contains("a"))
```

```
## # A tibble: 50 x 4
##   class name    math moral
##   <chr> <chr>  <dbl> <dbl>
## 1 六1班 何娜      92     9
## 2 六1班 黄才菊    77    NA
## 3 六1班 陈芳妹    87     9
## 4 六1班 陈学勤    79     9
## 5 六1班 陈祝贞    79     8
## 6 六1班 何小薇    73     8
## # ... with 44 more rows
```

- 根据正则表达式匹配选择列：

```
df %>%
  select(matches("m.*a"))
```

```
## # A tibble: 50 x 2
##    math moral
##   <dbl> <dbl>
## 1    92     9
## 2    77    NA
## 3    87     9
## 4    79     9
## 5    79     8
## 6    73     8
## # ... with 44 more rows
```

- 根据条件（逻辑判断）选择列，例如选择所有数值型的列：

```
df %>%
  select(where(is.numeric))
```

```
## # A tibble: 50 x 5
##   chinese  math english moral science
##     <dbl> <dbl>   <dbl> <dbl>   <dbl>
## 1      87    92      79     9      10
## 2      95    77      75    NA       9
## 3      79    87      66     9      10
```

```
## 4      NA    79      66     9      10
## 5      76    79      67     8      10
## 6      83    73      65     8       9
## # ... with 44 more rows
```

- 也可以自定义返回 TRUE 或 FALSE 的判断函数，支持 purrr 风格公式写法。例如，选择 列和 > 3000 的列：

```
df[, 4:8] %>%
  select(where(~ sum(.x, na.rm = TRUE) > 3000))
```

```
## # A tibble: 50 x 2
##   chinese  math
##     <dbl> <dbl>
## 1      87    92
## 2      95    77
## 3      79    87
## 4      NA    79
## 5      76    79
## 6      83    73
## # ... with 44 more rows
```

再比如，结合 n_distinct() 选择唯一值数目 < 10 的列：

```
df %>%
  select(where(~ n_distinct(.x) < 10))
```

```
## # A tibble: 50 x 4
##   class sex   moral science
##   <chr> <chr> <dbl>   <dbl>
## 1 六1班 女        9      10
## 2 六1班 女       NA       9
## 3 六1班 女        9      10
## 4 六1班 男        9      10
## 5 六1班 女        8      10
## 6 六1班 女        8       9
## # ... with 44 more rows
```

3. 用 “-” 删除列

```
df %>%
  select(-c(name, chinese, science))  # 或者 select(-ends_with("e"))
```

```
## # A tibble: 50 x 5
##    class sex    math english moral
##    <chr> <chr> <dbl>   <dbl> <dbl>
## 1 六1班 女       92      79     9
## 2 六1班 女       77      75    NA
## 3 六1班 女       87      66     9
## 4 六1班 男       79      66     9
## 5 六1班 女       79      67     8
## 6 六1班 女       73      65     8
## # ... with 44 more rows
```

```
df %>%
  select(math, everything(), -ends_with("e"))
```

```
## # A tibble: 50 x 5
##    math class sex   english moral
##   <dbl> <chr> <chr>   <dbl> <dbl>
## 1    92 六1班 女         79     9
## 2    77 六1班 女         75    NA
## 3    87 六1班 女         66     9
## 4    79 六1班 男         66     9
## 5    79 六1班 女         67     8
## 6    73 六1班 女         65     8
## # ... with 44 more rows
```

注意： `-ends_with()` 要放在 `everything()` 后面，否则删除的列就全回来了。

4. 调整列的顺序

列是根据被选择的顺序排列：

```
df %>%
  select(ends_with("e"), math, name, class, sex)
```

```
## # A tibble: 50 x 6
##    name   chinese science  math class sex
##    <chr>    <dbl>   <dbl> <dbl> <chr> <chr>
## 1 何娜        87      10    92 六1班 女
## 2 黄才菊      95       9    77 六1班 女
## 3 陈芳妹      79      10    87 六1班 女
## 4 陈学勤      NA      10    79 六1班 男
## 5 陈祝贞      76      10    79 六1班 女
## 6 何小薇      83       9    73 六1班 女
```

```
## # ... with 44 more rows
```

everything() 返回未被选择的所有列，将某一列移到第一列时很方便：

```
df %>%
  select(math, everything())
```

```
## # A tibble: 50 x 8
##    math class name   sex   chinese english moral science
##   <dbl> <chr> <chr>  <chr>   <dbl>   <dbl> <dbl>   <dbl>
## 1    92 六1班 何娜   女         87      79     9      10
## 2    77 六1班 黄才菊 女         95      75    NA       9
## 3    87 六1班 陈芳妹 女         79      66     9      10
## 4    79 六1班 陈学勤 男         NA      66     9      10
## 5    79 六1班 陈祝贞 女         76      67     8      10
## 6    73 六1班 何小薇 女         83      65     8       9
## # ... with 44 more rows
```

用 relocate() 函数，将选择的列移到某列之前或之后，基本语法为：

```
relocate(.data, ..., .before, .after)
```

例如，将数值列移到 name 列的后面：

```
df %>%
  relocate(where(is.numeric), .after = name)
```

```
## # A tibble: 50 x 8
##   class name   chinese  math english moral science sex
##   <chr> <chr>    <dbl> <dbl>   <dbl> <dbl>   <dbl> <chr>
## 1 六1班 何娜        87    92      79     9      10 女
## 2 六1班 黄才菊      95    77      75    NA       9 女
## 3 六1班 陈芳妹      79    87      66     9      10 女
## 4 六1班 陈学勤      NA    79      66     9      10 男
## 5 六1班 陈祝贞      76    79      67     8      10 女
## 6 六1班 何小薇      83    73      65     8       9 女
## # ... with 44 more rows
```

5. 重命名列

set_names() 为所有列设置新列名：

```
df %>%
  set_names(" 班级", " 姓名", " 性别", " 语文",
            " 数学", " 英语", " 品德", " 科学")
```

```
## # A tibble: 50 x 8
##   班级  姓名   性别   语文  数学  英语  品德  科学
##   <chr> <chr>  <chr> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 六1班 何娜   女       87    92    79     9    10
## 2 六1班 黄才菊 女       95    77    75    NA     9
## 3 六1班 陈芳妹 女       79    87    66     9    10
## 4 六1班 陈学勤 男       NA    79    66     9    10
## 5 六1班 陈祝贞 女       76    79    67     8    10
## 6 六1班 何小薇 女       83    73    65     8     9
## # ... with 44 more rows
```

rename() 只修改部分列名，格式为：新名 = 旧名

```
df %>%
  rename(数学 = math, 科学 = science)
```

```
## # A tibble: 50 x 8
##   class name   sex   chinese  数学 english moral  科学
##   <chr> <chr>  <chr>   <dbl> <dbl>   <dbl> <dbl> <dbl>
## 1 六1班 何娜   女         87    92      79     9    10
## 2 六1班 黄才菊 女         95    77      75    NA     9
## 3 六1班 陈芳妹 女         79    87      66     9    10
## 4 六1班 陈学勤 男         NA    79      66     9    10
## 5 六1班 陈祝贞 女         76    79      67     8    10
## 6 六1班 何小薇 女         83    73      65     8     9
## # ... with 44 more rows
```

还有更强大的 rename_with(.data, .fn, .cols)，参数.col 支持用选择列语法选择要重命名的列，.fn 是对所选列重命名的函数，将原列名的字符向量变成新列名的字符向量。比如，将包含”m’’ 的列名，都拼接上前缀”new_“：

```
df %>%
  rename_with(~ paste0("new_", .x), matches("m"))
```

```
## # A tibble: 50 x 8
##   class new_name sex   chinese new_math english new_moral science
##   <chr> <chr>    <chr>   <dbl>    <dbl>   <dbl>     <dbl>   <dbl>
## 1 六1班 何娜     女         87       92      79         9      10
## 2 六1班 黄才菊   女         95       77      75        NA       9
## 3 六1班 陈芳妹   女         79       87      66         9      10
## 4 六1班 陈学勤   男         NA       79      66         9      10
## 5 六1班 陈祝贞   女         76       79      67         8      10
## 6 六1班 何小薇   女         83       73      65         8       9
```

```
## # ... with 44 more rows
```

6. 强大的 across() 函数

函数 across() 人如其名，让零个/一个/多个函数**穿过**所选择的列，即**同时对所选择的多列应用若干函数**，基本格式为：

```
across(.cols = everything(), .fns = NULL, ..., .names)
```

- .cols 为根据**选择列语法**选定的列范围；
- .fns 为应用到选定列上的函数[7]，它可以是：
  - NULL：不对列作变换；
  - 一个函数，如 mean；
  - 一个 purrr 风格的匿名函数，如 ~ .X * 10
  - 多个函数或匿名函数构成的列表
- .names 用来设置输出列的列名样式，默认为 {col}_{fn}

across() 支持各种选择列语法，与 mutate() 和 summarise() 连用，产生非常强大的同时修改/（多种）汇总多列效果；

across() 也能与 group_by()，count() 和 distinct() 连用，此时 .fns 为 NULL，只起选择列的作用。

across() 函数的引入，使得可以弃用那些限定列范围的后缀：_all，_if，_at：

- across(everything(), .fns)：在所有列范围内，代替后缀 _all
- across(where(), .fns)：在满足条件的列范围内，代替后缀 _if
- across(.cols, .fns)：在给定的列范围内，代替后缀 _at

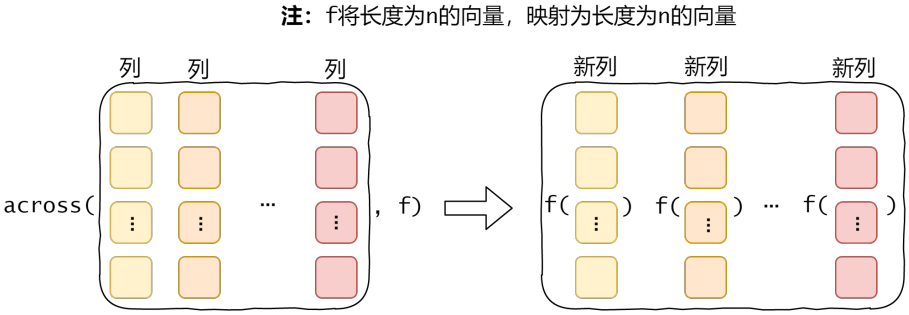

图 2.19: across 函数示意图

2.5.2 修改列

修改列，即修改数据框的列，计算新列。

[7]这些函数内部可以使用 cur_column() 和 cur_group() 以访问当前列和分组键值.

1. 创建新列

用 dplyr 包中的 mutate() 创建或修改列，返回原数据框并增加新列；若改用 transmute() 则只返回增加的新列，新列默认加在最后一列，参数.before，.after 可以设置新列的位置。

若只给新列 1 个值，则循环使用得到值相同的一列:

```
df %>%
  mutate(new_col = 5)
```

```
## # A tibble: 50 x 9
##    class name   sex   chinese  math english moral science new_col
##    <chr> <chr>  <chr>   <dbl> <dbl>   <dbl> <dbl>   <dbl>   <dbl>
## 1 六1班 何娜   女         87    92      79     9      10       5
## 2 六1班 黄才菊 女         95    77      75    NA       9       5
## 3 六1班 陈芳妹 女         79    87      66     9      10       5
## 4 六1班 陈学勤 男         NA    79      66     9      10       5
## 5 六1班 陈祝贞 女         76    79      67     8      10       5
## 6 六1班 何小薇 女         83    73      65     8       9       5
## # ... with 44 more rows
```

正常是以长度等于行数的向量赋值:

```
df %>%
  mutate(new_col = 1:n())
```

```
## # A tibble: 50 x 9
##    class name   sex   chinese  math english moral science new_col
##    <chr> <chr>  <chr>   <dbl> <dbl>   <dbl> <dbl>   <dbl>   <int>
## 1 六1班 何娜   女         87    92      79     9      10       1
## 2 六1班 黄才菊 女         95    77      75    NA       9       2
## 3 六1班 陈芳妹 女         79    87      66     9      10       3
## 4 六1班 陈学勤 男         NA    79      66     9      10       4
## 5 六1班 陈祝贞 女         76    79      67     8      10       5
## 6 六1班 何小薇 女         83    73      65     8       9       6
## # ... with 44 more rows
```

注： n() 返回当前分组的样本数, 未分组则为总行数。

2. 计算新列

用数据框的列计算新列，若修改当前列，只需要赋值给原列名。

```
df %>%
  mutate(total = chinese + math + english + moral + science)
```

```
## # A tibble: 50 x 9
##   class name   sex   chinese  math english moral science total
##   <chr> <chr>  <chr>   <dbl> <dbl>   <dbl> <dbl>   <dbl> <dbl>
## 1 六1班 何娜   女         87    92      79     9      10   277
## 2 六1班 黄才菊 女         95    77      75    NA       9    NA
## 3 六1班 陈芳妹 女         79    87      66     9      10   251
## 4 六1班 陈学勤 男         NA    79      66     9      10    NA
## 5 六1班 陈祝贞 女         76    79      67     8      10   240
## 6 六1班 何小薇 女         83    73      65     8       9   238
## # ... with 44 more rows
```

注意：不能用 sum(), 它会将整个列的内容都加起来，类似的还有 mean().

在同一个 mutate() 中可以同时创建或计算多个列，它们是从前往后依次计算，所以可以使用前面新创建的列，例如

- 计算 df 中 math 列的中位数
- 创建标记 math 是否大于中位数的逻辑值列
- 用 as.numeric() 将 TRUE/FALSE 转化为 1/0

```
df %>%
  mutate(med = median(math, na.rm = TRUE),
         label = math > med,
         label = as.numeric(label))
```

```
## # A tibble: 50 x 10
##   class name   sex   chinese  math english moral science   med label
##   <chr> <chr>  <chr>   <dbl> <dbl>   <dbl> <dbl>   <dbl> <dbl> <dbl>
## 1 六1班 何娜   女         87    92      79     9      10    73     1
## 2 六1班 黄才菊 女         95    77      75    NA       9    73     1
## 3 六1班 陈芳妹 女         79    87      66     9      10    73     1
## 4 六1班 陈学勤 男         NA    79      66     9      10    73     1
## 5 六1班 陈祝贞 女         76    79      67     8      10    73     1
## 6 六1班 何小薇 女         83    73      65     8       9    73     0
## # ... with 44 more rows
```

3. 修改多列

结合 across() 和**选择列语法**可以应用函数到多列，从而实现同时修改多列。

(1) 应用函数到所有列

将所有列转化为字符型：

```
df %>%
  mutate(across(everything(), as.character))
```

```
## # A tibble: 50 x 8
##   class name   sex   chinese math  english moral science
##   <chr> <chr>  <chr> <chr>   <chr> <chr>   <chr> <chr>
## 1 六1班 何娜   女    87      92    79      9     10
## 2 六1班 黄才菊 女    95      77    75      <NA>  9
## 3 六1班 陈芳妹 女    79      87    66      9     10
## 4 六1班 陈学勤 男    <NA>    79    66      9     10
## 5 六1班 陈祝贞 女    76      79    67      8     10
## 6 六1班 何小薇 女    83      73    65      8     9
## # ... with 44 more rows
```

(2) 应用函数到满足条件的列

对所有数值列做归一化：

```
rescale = function(x) {
  rng = range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
df %>%
  mutate(across(where(is.numeric), rescale))
```

```
## # A tibble: 50 x 8
##   class name   sex   chinese  math english  moral science
##   <chr> <chr>  <chr>   <dbl> <dbl>   <dbl>  <dbl>   <dbl>
## 1 六1班 何娜   女      0.843 0.974   1      0.875   1
## 2 六1班 黄才菊 女      1     0.776   0.926 NA       0.833
## 3 六1班 陈芳妹 女      0.686 0.908   0.759  0.875   1
## 4 六1班 陈学勤 男     NA     0.803   0.759  0.875   1
## 5 六1班 陈祝贞 女      0.627 0.803   0.778  0.75    1
## 6 六1班 何小薇 女      0.765 0.724   0.741  0.75    0.833
## # ... with 44 more rows
```

(3) 应用函数到指定的列

将 `iris` 中的 `length` 和 `width` 测量单位从厘米变成毫米：

```
as_tibble(iris) %>%
  mutate(across(contains("Length") | contains("Width"), ~ .x * 10))
```

```
## # A tibble: 150 x 5
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width 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
## 6           54          39           17           4 setosa
## # ... with 144 more rows
```

4. 替换 NA

(1) replace_na()

实现用某个值替换一列中的所有 NA 值，该函数接受一个命名列表，其成分为 列名 = 替换值：

```
starwars %>%
  replace_na(list(hair_color = "UNKNOWN",
                  height = mean(.$height, na.rm = TRUE)))
```

```
## # A tibble: 87 x 14
##   name  height  mass hair_color skin_color eye_color birth_year sex
##   <chr>  <dbl> <dbl> <chr>      <chr>      <chr>          <dbl> <chr>
## 1 Luke~    172    77 blond      fair       blue            19   male
## 2 C-3PO    167    75 UNKNOWN    gold       yellow         112   none
## 3 R2-D2     96    32 UNKNOWN    white, bl~ red             33   none
## 4 Dart~    202   136 none       white      yellow          41.9 male
## 5 Leia~    150    49 brown      light      brown           19   fema~
## 6 Owen~    178   120 brown, gr~ light      blue            52   male
## # ... with 81 more rows, and 6 more variables: gender <chr>,
## #   homeworld <chr>, species <chr>, films <list>, vehicles <list>,
## #   starships <list>
```

(2) fill()

用前一个（或后一个）非缺失值填充 NA。有些表在记录时，会省略与上一条记录相同的内容，如下表：

```
load("datas/gap_data.rda")
knitr::kable(gap_data, align="c")
```

site	species	sample_num	bees_present
Bilpin	A. longiforlia	1	TRUE

site	species	sample_num	bees_present
NA	NA	2	TRUE
NA	NA	3	TRUE
NA	A. elongata	1	TRUE
NA	NA	2	FALSE
NA	NA	3	TRUE
Grose Vale	A. terminalis	1	FALSE
NA	NA	2	FALSE
NA	NA	2	TRUE

`tidyr` 包中的 `fill()` 适合处理这种结构的缺失值, 默认是向下填充，即用上一个非缺失值填充:

```
gap_data %>%
  fill(site, species)
```

```
## # A tibble: 9 x 4
##   site   species        sample_num bees_present
##   <chr>  <chr>               <dbl> <lgl>
## 1 Bilpin A. longiforlia          1 TRUE
## 2 Bilpin A. longiforlia          2 TRUE
## 3 Bilpin A. longiforlia          3 TRUE
## 4 Bilpin A. elongata             1 TRUE
## 5 Bilpin A. elongata             2 FALSE
## 6 Bilpin A. elongata             3 TRUE
## # ... with 3 more rows
```

5. 重新编码

实际中，经常需要对列中的值进行重新编码。

(1) 两类别情形：`if_else()`

用 `if_else()` 作 是/否决策以确定用哪个值做重新编码:

```
df %>%
  mutate(sex = if_else(sex == " 男", "M", "F"))
```

```
## # A tibble: 50 x 8
##   class name   sex   chinese  math english moral science
##   <chr> <chr>  <chr>   <dbl> <dbl>   <dbl> <dbl>   <dbl>
## 1 六1班 何娜   F          87    92      79     9      10
## 2 六1班 黄才菊 F          95    77      75    NA       9
```

```
## 3 六1班 陈芳妹 F          79    87      66     9      10
## 4 六1班 陈学勤 M          NA    79      66     9      10
## 5 六1班 陈祝贞 F          76    79      67     8      10
## 6 六1班 何小薇 F          83    73      65     8       9
## # ... with 44 more rows
```

(2) 多类别情形：case_when()

用 case_when() 做更多条件下的重新编码，避免使用很多 if_else() 嵌套：

```
df %>%
  mutate(math = case_when(math >= 75 ~ "High",
                          math >= 60 ~ "Middle",
                          TRUE       ~ "Low"))
```

```
## # A tibble: 50 x 8
##   class name   sex   chinese math   english moral science
##   <chr> <chr>  <chr>   <dbl> <chr>    <dbl> <dbl>   <dbl>
## 1 六1班 何娜   女         87 High        79     9      10
## 2 六1班 黄才菊 女         95 High        75    NA       9
## 3 六1班 陈芳妹 女         79 High        66     9      10
## 4 六1班 陈学勤 男         NA High        66     9      10
## 5 六1班 陈祝贞 女         76 High        67     8      10
## 6 六1班 何小薇 女         83 Middle      65     8       9
## # ... with 44 more rows
```

case_when() 中用的是公式形式，

- 左边是返回 TRUE 或 FALSE 的表达式或函数
- 右边是若左边表达式为 TRUE，则重新编码的值，也可以是表达式或函数
- 每个分支条件将从上到下的计算，并接受第一个 TRUE 条件
- 最后一个分支直接用 TRUE 表示若其他条件都不为 TRUE 时怎么做

(3) 更强大的重新编码函数

基于 tidyverse 设计哲学，sjmisc 包实现了对变量做数据变换，如重新编码、二分或分组变量、设置与替换缺失值等；sjmisc 包也支持标签化数据，这对操作 SPSS 或 Stata 数据集特别有用。

重新编码函数 rec(), 可以将变量的旧值重新编码为新值，基本格式为：

```
rec(x, rec, append, ...)
```

- x：为数据框（或向量）；
- append：默认为 TRUE, 则返回包含重编码新列的数据框，FALSE 则只返回重编码的新列；
- rec：设置重编码模式，即哪些旧值被哪些新值取代，具体如下：

- 重编码对：每个重编码对用“;”隔开，例如 `rec="1=1; 2=4; 3=2; 4=3"`
- 多值：多个旧值（逗号分隔）重编码为一个新值，例如 `rec="1,2=1; 3,4=2"`
- 值范围：用冒号表示值范围，例如 `rec="1:4=1; 5:8=2"`
- 数值型值范围：带小数部分的数值向量，值范围内的所有值将被重新编码，例如 `rec="1:2.5=1; 2.6:3=2"`[8]
- “min”和“max”：最小值和最大值分别用 `min` 和 `max` 表示，例如 `rec = "min:4=1; 5:max=2"`（`min` 和 `max` 也可以作为新值，如 `5:7=max`, 表示将 5~7 编码为 `max(x)`）
- “else”：所有未设定的其他值，用 `else` 表示，例如 `rec="3=1; 1=2; else=3"`
- “copy”：`else` 可以结合 `copy` 一起使用，表示所有未设定的其他值保持原样（从原数值 copy），例如 `rec="3=1; 1=2; else=copy"`
- `NAs`：`NA` 既可以作为旧值，也可以作为新值，例如 `rec="NA=1; 3:5=NA"`
- “rev”：设置反转值顺序
- 非捕获值：不匹配的值将设置为 `NA`, 除非使用 `else` 和 `copy`.

```
library(sjmisc)
df %>%
  rec(math, rec = "min:59= 不及格; 60:74= 中; 75:85= 良; 85:max= 优",
      append = FALSE) %>%
  frq()                          # 频率表
```

```
##
## math_r <character>
## # total N=50  valid N=50  mean=3.28  sd=1.26
##
## Value  |  N | Raw % | Valid % | Cum. %
## ------------------------------------
## -Inf   |  3 |  6.00 |    6.00 |      6
## 不及格 | 14 | 28.00 |   28.00 |     34
## 良     | 10 | 20.00 |   20.00 |     54
## 优     | 12 | 24.00 |   24.00 |     78
## 中     | 11 | 22.00 |   22.00 |    100
## <NA>   |  0 |  0.00 |    <NA> |   <NA>
```

注：新值的值标签可以在重新编码时一起设置，只需要在每个重编码对后接上中括号标签。

2.5.3 筛选行

筛选行，即按行选择数据子集，包括过滤行、对行切片、删除行。

先创建一个包含重复行的数据框：

[8]注意 `2.55` 因未包含在值范围将不被重新编码.

```
set.seed(123)
df_dup = df %>%
  slice_sample(n = 60, replace = TRUE)
```

1. 用 filter() 根据条件筛选行

提供筛选条件给 filter() 则返回满足该条件的行。筛选条件可以是长度同行数的逻辑向量，更一般的是基于能返回这样逻辑向量的列表达式。

```
df_dup %>%
  filter(sex == " 男", math > 80)
```

```
## # A tibble: 8 x 8
##   class name   sex   chinese  math english moral science
##   <chr> <chr>  <chr>   <dbl> <dbl>   <dbl> <dbl>   <dbl>
## 1 六2班 陈华健 男          92    84      70     9      10
## 2 六2班 陈华健 男          92    84      70     9      10
## 3 六4班 <NA>   男          84    85      52     9       8
## 4 六2班 陈华健 男          92    84      70     9      10
## 5 六4班 李小龄 男          90    87      69    10      10
## 6 六4班 李小龄 男          90    87      69    10      10
## # ... with 2 more rows
```

注：多个条件之间用“,”隔开，相当于 and.

```
df_dup %>%
  filter(sex == " 女", (is.na(english) | math > 80))
```

```
## # A tibble: 11 x 8
##   class name   sex   chinese  math english moral science
##   <chr> <chr>  <chr>   <dbl> <dbl>   <dbl> <dbl>   <dbl>
## 1 六4班 周婵   女          92    94      77    10       9
## 2 六1班 陈芳妹 女          79    87      66     9      10
## 3 六5班 陆曼   女          88    84      69     8      10
## 4 六5班 陆曼   女          88    84      69     8      10
## 5 六2班 徐雅琦 女          92    86      72    NA       9
## 6 六5班 陆曼   女          88    84      69     8      10
## # ... with 5 more rows
```

```
df_dup %>%
  filter(between(math, 70, 80))      # 闭区间
```

```
## # A tibble: 15 x 8
```

```
##   class name   sex   chinese  math english moral science
##   <chr> <chr>  <chr>   <dbl> <dbl>   <dbl> <dbl>   <dbl>
## 1 六2班 杨远芸 女         93    80      68     9      10
## 2 六5班 容唐   女         83    71      56     9       7
## 3 六4班 关小孟 男         84    78      49     8       5
## 4 六1班 陈祝贞 女         76    79      67     8      10
## 5 六1班 陈欣越 男         57    80      60     9       9
## 6 六1班 雷旺   男         NA    80      68     8       9
## # ... with 9 more rows
```

2. 在限定列范围内根据条件筛选行

dplyr 1.0.4 最新提供了函数 if_any() 和 if_all()，基本格式为

- if_any(.cols, .fns, ...)
- if_all(.cols, .fns, ...)

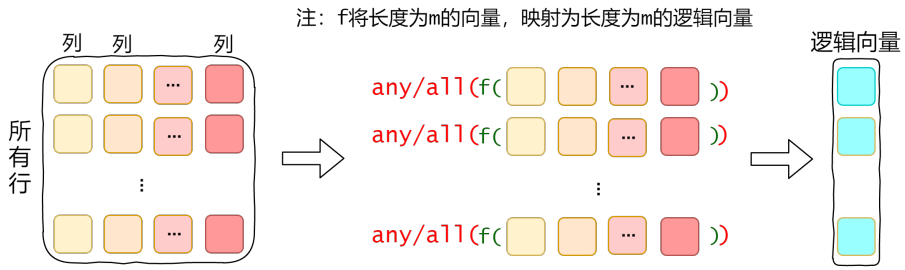

图 2.20: ifany,ifall 函数筛选行示意图

其操作逻辑类似 across(), 只是返回的是关于行的逻辑向量（长度同行数），用于根据多列的值筛选行：在 .cols 所选择的列范围内，分别对每一行做逻辑判断，前者返回各行是否存在某列满足函数 .fns 所表示的判断条件；后者返回各行是否所有列都满足函数 .fns 所表示的判断条件。

(1) 限定列范围内，筛选" 所有值都满足某条件的行"

借助 if_all() 函数可以轻松实现，其操作逻辑是，在所选列范围内，根据条件做判断，得到多列逻辑值，再借助 all （所有）语义，合成一个逻辑值向量，用于 filter() 筛选行。

选出第 4-6 列范围内，所有值都 > 75 的行：

```
df %>%
  filter(if_all(4:6, ~ .x > 75))
```

```
## # A tibble: 3 x 8
##   class name   sex   chinese  math english moral science
##   <chr> <chr>  <chr>   <dbl> <dbl>   <dbl> <dbl>   <dbl>
## 1 六1班 何娜   女         87    92      79     9      10
## 2 六4班 周婵   女         92    94      77    10       9
```

```
## 3 六5班 符苡榕 女          85    89      76     9      NA
```

选出所有列范围内，所有值都不是 NA 的行

```
df_dup %>%
  filter(if_all(everything(), ~ !is.na(.x)))
```

```
## # A tibble: 38 x 8
##   class name   sex   chinese  math english moral science
##   <chr> <chr>  <chr>   <dbl> <dbl>   <dbl> <dbl>   <dbl>
## 1 六4班 周婵   女         92    94      77    10       9
## 2 六2班 杨远芸 女         93    80      68     9      10
## 3 六2班 陈华健 男         92    84      70     9      10
## 4 六1班 陈芳妹 女         79    87      66     9      10
## 5 六5班 陆曼   女         88    84      69     8      10
## 6 六5班 胡玉洁 女         74    61      52     9       6
## # ... with 32 more rows
```

(2) 限定列范围内，筛选” 存在值满足某条件的行”

借助 if_any() 函数可以轻松实现，其操作逻辑是，在所选列范围内，根据条件做判断，得到多列逻辑值，再借助 any （存在）语义，合成一个逻辑值向量，用于 filter() 筛选行。

选出所有列范围内，存在值包含 “bl” 的行

```
starwars %>%
  filter(if_any(everything(), ~ str_detect(.x, "bl")))
```

```
## # A tibble: 47 x 14
##   name  height  mass hair_color skin_color eye_color birth_year sex
##   <chr>  <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr>
## 1 Luke~    172    77 blond      fair       blue              19 male
## 2 R2-D2     96    32 <NA>       white, bl~ red               33 none
## 3 Owen~    178   120 brown, gr~ light      blue              52 male
## 4 Beru~    165    75 brown      light      blue              47 fema~
## 5 Bigg~    183    84 black      light      brown             24 male
## 6 Obi-~    182    77 auburn, w~ fair       blue-gray         57 male
## # ... with 41 more rows, and 6 more variables: gender <chr>,
## #   homeworld <chr>, species <chr>, films <list>, vehicles <list>,
## #   starships <list>
```

选出数值列范围内，存在值 > 90 的行

```
df %>%
  filter(if_any(where(is.numeric), ~ .x > 90))
```

```
## # A tibble: 8 x 8
##   class name   sex   chinese  math english moral science
##   <chr> <chr>  <chr>   <dbl> <dbl>   <dbl> <dbl>   <dbl>
## 1 六1班 何娜   女         87    92      79     9      10
## 2 六1班 黄才菊 女         95    77      75    NA       9
## 3 六2班 黄祖娜 女         94    88      75    10      10
## 4 六2班 徐雅琦 女         92    86      72    NA       9
## 5 六2班 陈华健 男         92    84      70     9      10
## 6 六2班 杨远芸 女         93    80      68     9      10
## # ... with 2 more rows
```

从字符列范围内，选择包含（存在）NA 的行：

```
df_dup %>%
  filter(if_any(where(is.character), is.na))
```

```
## # A tibble: 3 x 8
##   class name   sex   chinese  math english moral science
##   <chr> <chr>  <chr>   <dbl> <dbl>   <dbl> <dbl>   <dbl>
## 1 六4班 <NA>   男         84    85      52     9       8
## 2 <NA>  徐达政 男         90    86      72     9      10
## 3 六5班 符芳盈 <NA>       58    85      48     9      10
```

3. 对行切片：slice_*()

slice 就是对行切片的意思，该系列函数的共同参数：

- n: 用来指定要选择的行数
- prop: 用来指定选择的行比例

```
slice(df, 3:7)                      # 选择 3-7 行
slice_head(df, n, prop)             # 从前面开始选择若干行
slice_tail(df, n, prop)             # 从后面开始选择若干行
slice_min(df, order_by, n, prop)    # 根据 order_by 选择最小的若干行
slice_max(df, order_by, n, prop)    # 根据 order_by 选择最大的若干行
slice_sample(df, n, prop)           # 随机选择若干行
```

选择 math 列值中前 5 大的行：

```
df %>%
  slice_max(math, n = 5)
```

```
## # A tibble: 5 x 8
##   class name   sex   chinese  math english moral science
```

```
##   <chr> <chr>  <chr>   <dbl> <dbl>   <dbl> <dbl>   <dbl>
## 1 六4班 周婵   女         92    94      77    10       9
## 2 六4班 陈丽丽 女         87    93      NA     8       6
## 3 六1班 何娜   女         87    92      79     9      10
## 4 六5班 符苡榕 女         85    89      76     9      NA
## 5 六2班 黄祖娜 女         94    88      75    10      10
```

4. 删除行

(1) 删除重复行

用 dplyr 包中的 distinct() 删除重复行（只保留第 1 个，删除其余）。

```
df_dup %>%
  distinct()
```

```
## # A tibble: 35 x 8
##   class name   sex   chinese  math english moral science
##   <chr> <chr>  <chr>   <dbl> <dbl>   <dbl> <dbl>   <dbl>
## 1 六4班 周婵   女         92    94      77    10       9
## 2 六2班 杨远芸 女         93    80      68     9      10
## 3 六2班 陈华健 男         92    84      70     9      10
## 4 六1班 陈芳妹 女         79    87      66     9      10
## 5 六5班 陆曼   女         88    84      69     8      10
## 6 六5班 胡玉洁 女         74    61      52     9       6
## # ... with 29 more rows
```

也可以只根据某些列判定重复：

```
df_dup %>%
  distinct(sex, math, .keep_all = TRUE)  # 只根据 sex 和 math 判定重复
```

```
## # A tibble: 32 x 8
##   class name   sex   chinese  math english moral science
##   <chr> <chr>  <chr>   <dbl> <dbl>   <dbl> <dbl>   <dbl>
## 1 六4班 周婵   女         92    94      77    10       9
## 2 六2班 杨远芸 女         93    80      68     9      10
## 3 六2班 陈华健 男         92    84      70     9      10
## 4 六1班 陈芳妹 女         79    87      66     9      10
## 5 六5班 陆曼   女         88    84      69     8      10
## 6 六5班 胡玉洁 女         74    61      52     9       6
## # ... with 26 more rows
```

注：默认只返回选择的列，要返回所有列，需要设置参数 .keep_all = TRUE。

(2) 删除包含 NA 的行

用 tidyr 包中的 drop_na() 删除所有包含 NA 的行：

```
df_dup %>%
  drop_na()
```

删除NA

```
## # A tibble: 38 x 8
##   class name   sex   chinese  math english moral science
##   <chr> <chr>  <chr>   <dbl> <dbl>   <dbl> <dbl>   <dbl>
## 1 六4班 周婵   女         92    94      77    10       9
## 2 六2班 杨远芸 女         93    80      68     9      10
## 3 六2班 陈华健 男         92    84      70     9      10
## 4 六1班 陈芳妹 女         79    87      66     9      10
## 5 六5班 陆曼   女         88    84      69     8      10
## 6 六5班 胡玉洁 女         74    61      52     9       6
## # ... with 32 more rows
```

也可以只删除某些列包含 NA 的行：

```
df_dup %>%
  drop_na(sex:math)
```

```
## # A tibble: 50 x 8
##   class name   sex   chinese  math english moral science
##   <chr> <chr>  <chr>   <dbl> <dbl>   <dbl> <dbl>   <dbl>
## 1 六4班 周婵   女         92    94      77    10       9
## 2 六2班 杨远芸 女         93    80      68     9      10
## 3 六2班 陈华健 男         92    84      70     9      10
## 4 六1班 陈芳妹 女         79    87      66     9      10
## 5 六5班 陆曼   女         88    84      69     8      10
## 6 六5班 胡玉洁 女         74    61      52     9       6
## # ... with 44 more rows
```

若要删除某些列都是 NA 的行，借助 if_all() 也很容易实现：

```
df_dup %>%
  filter(!if_all(where(is.numeric), is.na))
```

2.5.4 对行排序

用 dplyr 包中的 arrange() 对行排序，默认是递增。

```
df_dup %>%
  arrange(math, sex)
```

```
## # A tibble: 60 x 8
##   class name   sex   chinese  math english moral science
##   <chr> <chr>  <chr>   <dbl> <dbl>   <dbl> <dbl>   <dbl>
## 1 六3班 邹嘉伟 男          67    18      62     8      NA
## 2 六3班 刘虹均 男          72    23      74     3       6
## 3 六3班 刘虹均 男          72    23      74     3       6
## 4 六3班 黄凯丽 女          70    23      61     4       4
## 5 六3班 黄凯丽 女          70    23      61     4       4
## 6 六3班 黄凯丽 女          70    23      61     4       4
## # ... with 54 more rows
```

若要递减排序，套一个 desc() 或变量名前加-：

```
df_dup %>%
  arrange(-math)              # 同 desc(math)，递减排序
```

```
## # A tibble: 60 x 8
##   class name   sex   chinese  math english moral science
##   <chr> <chr>  <chr>   <dbl> <dbl>   <dbl> <dbl>   <dbl>
## 1 六4班 周婵   女          92    94      77    10       9
## 2 六4班 陈丽丽 女          87    93      NA     8       6
## 3 六5班 符苡榕 女          85    89      76     9      NA
## 4 六5班 符苡榕 女          85    89      76     9      NA
## 5 六1班 陈芳妹 女          79    87      66     9      10
## 6 六4班 李小龄 男          90    87      69    10      10
## # ... with 54 more rows
```

2.5.5 分组汇总

分组汇总，相当于 Excel 的透视表功能。

对未分组的数据框，一些操作如 mutate() 是在所有行上执行——或者说，整个数据框是一个分组，所有行都属于它。

若数据框被分组，则这些操作是分别在每个分组上独立执行。可以认为是，将数据框拆分为更小的多个数据框。在每个更小的数据框上执行操作，最后再将结果合并回来。

1. 创建分组

用 group_by() 创建分组，只是对数据框增加了分组信息（用 group_keys() 查看），并不是真的将数据分割为多个数据框。

```
df_grp = df %>%
  group_by(sex)
```

```
group_keys(df_grp)          # 分组键值（唯一识别分组）
group_indices(df_grp)       # 查看每一行属于哪一分组
group_rows(df_grp)          # 查看每一组包含哪些行
ungroup(df_grp)             # 解除分组
```

其他分组函数

- 真正将数据框分割为多个分组：group_split()，返回列表，其每个成分是一个分组数据框
- 将数据框分组（group_by），再做嵌套（nest），生成嵌套数据框：group_nest()

```
iris %>%
  group_nest(Species)
```

```
## # A tibble: 3 x 2
##   Species                  data
##   <fct>      <list<tibble[,4]>>
## 1 setosa               [50 x 4]
## 2 versicolor           [50 x 4]
## 3 virginica            [50 x 4]
```

- purrr 风格的分组迭代：将函数 .f 依次应用到分组数据框 .data 的每个分组上
  - group_map(.data, .f, ...)：返回列表
  - group_walk(.data, .f, ...)：不返回，只关心副作用
  - group_modify(.data, .f, ...)：返回修改后的分组数据框

```
iris %>%
  group_by(Species) %>%
  group_map(~ head(.x, 2))      # 提取每组的前两个观测
```

```
## [[1]]
## # A tibble: 2 x 4
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
##          <dbl>       <dbl>        <dbl>       <dbl>
## 1          5.1         3.5          1.4         0.2
## 2          4.9         3            1.4         0.2
##
## [[2]]
## # A tibble: 2 x 4
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
##          <dbl>       <dbl>        <dbl>       <dbl>
## 1          7           3.2          4.7         1.4
## 2          6.4         3.2          4.5         1.5
```

```
##
## [[3]]
## # A tibble: 2 x 4
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
##          <dbl>       <dbl>        <dbl>       <dbl>
## 1          6.3         3.3          6           2.5
## 2          5.8         2.7          5.1         1.9
```

> **分组**是一种强大的数据思维，当您想分组并分别操作（包括汇总）每组数据时，应该优先采用 `group_by()` + 操作，而不是分割数据 + 循环迭代。

2. 分组汇总

对数据框做分组最主要的目的就是做分组汇总，汇总就是以某种方式组合行，用 `dplyr` 包中的 `summarise()` 函数实现，结果只保留分组列唯一值和新创建的汇总列。

(1) summarise()

可以与很多自带或自定义的汇总函数连用，常用的汇总函数有：

- `n()`: 观测数
- `n_distinct(var)`: 变量 var 的唯一值数目
- `sum(var), max(var), min(var), ...`
- `mean(var), median(var), sd(var), IQR(var), ...`

```
df %>%
  group_by(sex) %>%
  summarise(n = n(),
            math_avg = mean(math, na.rm = TRUE),
            math_med = median(math))
```

```
## # A tibble: 3 x 4
##   sex       n math_avg math_med
##   <chr> <int>    <dbl>    <dbl>
## 1 男       24     64.6       NA
## 2 女       25     70.8       NA
## 3 <NA>      1     85         85
```

函数 `summarise()` ，配合 `across()` 可以对所选择的列做汇总。好处是可以借助辅助选择器或判断条件选择多列，还能在这些列上执行多个函数，只需要将它们放入一个列表。

(2) 对某些列做汇总

```
df %>%
  group_by(class, sex) %>%
  summarise(across(contains("h"), mean, na.rm = TRUE))
```

```
## # A tibble: 12 x 5
## # Groups:   class [6]
##   class sex   chinese  math english
##   <chr> <chr>   <dbl> <dbl>   <dbl>
## 1 六1班 男       57    79.7    64.7
## 2 六1班 女       80.7  77.2    67.4
## 3 六2班 男       75.4  68.8    42.6
## 4 六2班 女       92.2  73.8    63.8
## 5 六3班 男       66    30.4    67.6
## 6 六3班 女       68.4  49.2    67.8
## # ... with 6 more rows
```

(3) 对所有列做汇总

```
df %>%
  select(-name) %>%
  group_by(class, sex) %>%
  summarise(across(everything(), mean, na.rm = TRUE))
```

```
## # A tibble: 12 x 7
## # Groups:   class [6]
##   class sex   chinese  math english moral science
##   <chr> <chr>   <dbl> <dbl>   <dbl> <dbl>   <dbl>
## 1 六1班 男       57    79.7    64.7  8.67    9.33
## 2 六1班 女       80.7  77.2    67.4  8.33    9.57
## 3 六2班 男       75.4  68.8    42.6  8.8     9.25
## 4 六2班 女       92.2  73.8    63.8  8.33    9
## 5 六3班 男       66    30.4    67.6  4.6     4.75
## 6 六3班 女       68.4  49.2    67.8  6.25    7.2
## # ... with 6 more rows
```

(4) 对满足条件的列做多种汇总

```
df_grp = df %>%
  group_by(class) %>%
  summarise(across(where(is.numeric),
                   list(sum=sum, mean=mean, min=min), na.rm = TRUE))
df_grp
```

```
## # A tibble: 6 x 16
##   class chinese_sum chinese_mean chinese_min math_sum math_mean
##   <chr>       <dbl>        <dbl>       <dbl>    <dbl>     <dbl>
## 1 六1班         622         77.8          57      702      78
## 2 六2班         746         82.9          66      570      71.2
## 3 六3班         606         67.3          44      349      38.8
## 4 六4班         850         85            72      771      77.1
## 5 六5班         726         72.6          58      720      72
## 6 <NA>           90         90            90       86      86
## # ... with 10 more variables: math_min <dbl>, english_sum <dbl>,
## #   english_mean <dbl>, english_min <dbl>, moral_sum <dbl>,
## #   moral_mean <dbl>, moral_min <dbl>, science_sum <dbl>,
## #   science_mean <dbl>, science_min <dbl>
```

可读性不好，再来个宽变长：

```
df_grp %>%
  pivot_longer(-class, names_to = c("Vars", ".value"), names_sep = "_")
```

```
## # A tibble: 30 x 5
##   class Vars      sum  mean   min
##   <chr> <chr>   <dbl> <dbl> <dbl>
## 1 六1班 chinese   622 77.8     57
## 2 六1班 math      702 78       55
## 3 六1班 english   666 66.6     54
## 4 六1班 moral      76  8.44     8
## 5 六1班 science    95  9.5      9
## 6 六2班 chinese   746 82.9     66
## # ... with 24 more rows
```

(5) 支持多返回值的汇总函数

summarise() 以前只支持一个返回值的汇总函数，如 sum，mean 等。现在也支持多返回值（返回向量值、甚至是数据框）的汇总函数，如 range()，quantile() 等。

```
qs = c(0.25, 0.5, 0.75)

df_q = df %>%
  group_by(sex) %>%
  summarise(math_qs = quantile(math, qs, na.rm = TRUE), q = qs)
df_q
```

```
## # A tibble: 9 x 3
```

```
## # Groups:   sex [3]
##   sex   math_qs     q
##   <chr>   <dbl> <dbl>
## 1 男       57.5  0.25
## 2 男       69    0.5
## 3 男       80    0.75
## 4 女       55    0.25
## 5 女       73    0.5
## 6 女       86.5  0.75
## # ... with 3 more rows
```

可读性不好，再来个长变宽：

```
df_q %>%
  pivot_wider(names_from = q, values_from = math_qs, names_prefix = "q_")
```

```
## # A tibble: 3 x 4
## # Groups:   sex [3]
##   sex   q_0.25 q_0.5 q_0.75
##   <chr>  <dbl> <dbl>  <dbl>
## 1 男      57.5    69   80
## 2 女      55      73   86.5
## 3 <NA>    85      85   85
```

3. 分组计数

用 `count()` 按分类变量 `class` 和 `sex` 分组，并按分组大小排序：

```
df %>%
  count(class, sex, sort = TRUE)
```

```
## # A tibble: 12 x 3
##   class sex       n
##   <chr> <chr> <int>
## 1 六1班 女        7
## 2 六4班 男        6
## 3 六2班 男        5
## 4 六3班 男        5
## 5 六3班 女        5
## 6 六5班 女        5
## # ... with 6 more rows
```

对已分组的数据框，用 `tally()` 计数：

```
df %>%
  group_by(math_level = cut(math, breaks = c(0, 60, 75, 80, 100), right = FALSE)) %>%
  tally()
```

```
## # A tibble: 5 x 2
##   math_level     n
##   <fct>      <int>
## 1 [0,60)        14
## 2 [60,75)       11
## 3 [75,80)        5
## 4 [80,100)      17
## 5 <NA>           3
```

注： `count()` 和 `tally()` 都有参数 `wt` 设置加权计数。

用 `add_count()` 和 `add_tally()` 可为数据集增加一列按分组变量分组的计数：

```
df %>%
  add_count(class, sex)
```

```
## # A tibble: 50 x 9
##   class name   sex   chinese  math english moral science     n
##   <chr> <chr>  <chr>   <dbl> <dbl>   <dbl> <dbl>   <dbl> <int>
## 1 六1班 何娜   女         87    92      79     9      10     7
## 2 六1班 黄才菊 女         95    77      75    NA       9     7
## 3 六1班 陈芳妹 女         79    87      66     9      10     7
## 4 六1班 陈学勤 男         NA    79      66     9      10     3
## 5 六1班 陈祝贞 女         76    79      67     8      10     7
## 6 六1班 何小薇 女         83    73      65     8       9     7
## # ... with 44 more rows
```

本节部分内容参阅 (G. G. Hadley Wickham 2017), (Desi Quintans 2019), `Vignettes of dplyr`.

2.6 其他数据操作

2.6.1 按行汇总

通常的数据操作逻辑都是**按列方式 (colwise)**，这使得按行汇总很困难。

`dplyr` 包提供了 `rowwise()` 函数为数据框创建**按行方式 (rowwise)**，使用 `rowwise()` 后并不是真的改变数据框，只是创建了按行元信息，改变了数据框的操作逻辑：

```
rf = df %>%
  rowwise()
```

```
rf %>%
  mutate(total = sum(c(chinese, math, english)))
```

```
## # A tibble: 50 x 9
## # Rowwise:
##   class name   sex   chinese  math english moral science total
##   <chr> <chr>  <chr>   <dbl> <dbl>   <dbl> <dbl>   <dbl> <dbl>
## 1 六1班 何娜   女         87    92      79     9      10   258
## 2 六1班 黄才菊 女         95    77      75    NA       9   247
## 3 六1班 陈芳妹 女         79    87      66     9      10   232
## 4 六1班 陈学勤 男         NA    79      66     9      10    NA
## 5 六1班 陈祝贞 女         76    79      67     8      10   222
## 6 六1班 何小薇 女         83    73      65     8       9   221
## # ... with 44 more rows
```

函数 `c_across()` 是为**按行方式 (rowwise)** 在选定的列范围汇总数据而设计的，它没有提供 `.fns` 参数，只能选择列。

```
rf %>%
  mutate(total = sum(c_across(where(is.numeric))))
```

```
## # A tibble: 50 x 9
## # Rowwise:
##   class name   sex   chinese  math english moral science total
##   <chr> <chr>  <chr>   <dbl> <dbl>   <dbl> <dbl>   <dbl> <dbl>
## 1 六1班 何娜   女         87    92      79     9      10   277
## 2 六1班 黄才菊 女         95    77      75    NA       9    NA
## 3 六1班 陈芳妹 女         79    87      66     9      10   251
## 4 六1班 陈学勤 男         NA    79      66     9      10    NA
## 5 六1班 陈祝贞 女         76    79      67     8      10   240
## 6 六1班 何小薇 女         83    73      65     8       9   238
## # ... with 44 more rows
```

若只是做按行求和或均值，直接用 `rowSums() / rowMeans()` 速度更快（不需要分割-汇总-合并），这里的 `rowwise` 行化后提供可以做更多的按行汇总的可能。

```
df %>%
  mutate(total = rowSums(across(where(is.numeric))))
```

```
## # A tibble: 50 x 9
##   class name   sex   chinese  math english moral science total
##   <chr> <chr>  <chr>   <dbl> <dbl>   <dbl> <dbl>   <dbl> <dbl>
## 1 六1班 何娜   女         87    92      79     9      10   277
```

```
## 2 六1班 黄才菊 女          95    77      75    NA       9    NA
## 3 六1班 陈芳妹 女          79    87      66     9      10   251
## 4 六1班 陈学勤 男          NA    79      66     9      10    NA
## 5 六1班 陈祝贞 女          76    79      67     8      10   240
## 6 六1班 何小薇 女          83    73      65     8       9   238
## # ... with 44 more rows
```

按行方式 (rowwise) 可以理解为一种特殊的分组：每一行作为一组。为 `rowwise()` 提供行 ID，用 `summarise()` 做汇总更能体会这一点。要解除行化模式，用 `ungroup()`.

```
df %>%
  rowwise(name) %>%
  summarise(total = sum(c_across(where(is.numeric))))
```

```
## # A tibble: 50 x 2
## # Groups:   name [50]
##   name   total
##   <chr>  <dbl>
## 1 何娜     277
## 2 黄才菊    NA
## 3 陈芳妹   251
## 4 陈学勤    NA
## 5 陈祝贞   240
## 6 何小薇   238
## # ... with 44 more rows
```

`rowwise` 行化操作的缺点是速度相对更慢，更建议用 1.6.2 节讲到的 `pmap()` 逐行迭代。

`rowwise` 行化更让人惊喜的是：它的逐行处理的逻辑 + 嵌套数据框可以更好地实现批量建模，在 `rowwise` 行化模式下，批量建模就像计算新列一样自然。批量建模，见 3.3.3 节, 可以用 “嵌套数据框 + `purrr::map_*()`” 实现，但这种 `rowwise` 技术，具有异曲同工之妙。

总结**逐行迭代**，除了 `for` 循环通常有四种做法：

```
iris[1:4] %>%                          # apply
  mutate(avg = apply(., 1, mean))
iris[1:4] %>%                          # rowwise (慢)
  rowwise() %>%
  mutate(avg = mean(c_across()))
iris[1:4] %>%                          # pmap
  mutate(avg = pmap_dbl(., ~ mean(c(...))))
iris[1:4] %>%                          # asplit(逐行分割) + map
  mutate(avg = map_dbl(asplit(., 1), mean))
```

2.6.2 窗口函数

汇总函数如 sum() 和 mean() 接受 n 个输入，返回 1 个值。而窗口函数是汇总函数的变体：接受 n 个输入，返回 n 个值。

例如，cumsum()、cummean()、rank()、lead()、lag() 等。

1. 排名和排序函数

共有 6 个排名函数，只介绍最常用的 min_rank()：从小到大排名（ties.method="min"），若要从大到小排名需要套一个 desc()

```
df %>%
  mutate(ranks = min_rank(desc(math))) %>%
  arrange(ranks)
```

```
## # A tibble: 50 x 9
##   class name   sex   chinese  math english moral science ranks
##   <chr> <chr>  <chr>   <dbl> <dbl>   <dbl> <dbl>   <dbl> <int>
## 1 六4班 周婵   女         92    94      77    10       9     1
## 2 六4班 陈丽丽 女         87    93      NA     8       6     2
## 3 六1班 何娜   女         87    92      79     9      10     3
## 4 六5班 符苡榕 女         85    89      76     9      NA     4
## 5 六2班 黄祖娜 女         94    88      75    10      10     5
## 6 六1班 陈芳妹 女         79    87      66     9      10     6
## # ... with 44 more rows
```

2. 移位函数

- lag(): 取前一个值，数据整体右移一位，相当于将时间轴滞后一个单位
- lead(): 取后一个值，数据整体左移一位，相当于将时间轴超前一个单位

```
library(lubridate)
dt = tibble(
  day = as_date("2019-08-30") + c(0,4:6),
  wday = weekdays(day),
  sales = c(2,6,2,3),
  balance = c(30, 25, -40, 30)
)
dt %>%
  mutate(sales_lag = lag(sales), sales_delta = sales - lag(sales))
```

```
## # A tibble: 4 x 6
##   day        wday   sales balance sales_lag sales_delta
```

```
##   <date>     <chr>  <dbl>   <dbl>     <dbl>       <dbl>
## 1 2019-08-30 星期五     2      30        NA          NA
## 2 2019-09-03 星期二     6      25         2           4
## 3 2019-09-04 星期三     2     -40         6          -4
## 4 2019-09-05 星期四     3      30         2           1
```

注：默认是根据行序移位，可用参数 `order_by` 设置根据某变量值大小顺序做移位。

3. 累计汇总

R base 已经提供了 `cumsum()`、`cummin()`、`cummax()`、`cumprod()`

dplyr 包又提供了 `cummean()`、`cumany()`、`cumall()`，后两者可与 `filter()` 连用选择行：

- `cumany(x)`: 用来选择遇到第一个满足条件之后的所有行
- `cumany(!x)`: 用来选择遇到第一个不满足条件之后的所有行
- `cumall(x)`: 用来选择所有行直到遇到第一个不满足条件的行
- `cumall(!x)`: 用来选择所有行直到遇到第一个满足条件的行

```
dt %>%
  filter(cumany(balance < 0))      # 选择第一次透支之后的所有行
```

```
## # A tibble: 2 x 4
##   day        wday   sales balance
##   <date>     <chr>  <dbl>   <dbl>
## 1 2019-09-04 星期三     2     -40
## 2 2019-09-05 星期四     3      30
```

```
dt %>%
  filter(cumall(!(balance < 0)))   # 选择所有行直到第一次透支
```

```
## # A tibble: 2 x 4
##   day        wday   sales balance
##   <date>     <chr>  <dbl>   <dbl>
## 1 2019-08-30 星期五     2      30
## 2 2019-09-03 星期二     6      25
```

2.6.3 滑窗迭代

“窗口函数”术语来自 SQL，意味着逐窗口浏览数据，将某函数重复应用于数据的每个“窗口”。窗口函数的典型应用包括滑动平均、累计和以及更复杂如滑动回归。

slider 包提供了 `slide_*()` 系列函数实现滑窗迭代，其基本格式为：

```
slide_*(.x, .f, ..., .before, .after, .step, .complete)
```

- .x: 为窗口所要滑过的向量
- .f: 要应用于每个窗口的函数，支持 purrr 风格公式写法
- ...: 用来传递 .f 的其他参数
- .before, .after: 设置窗口范围当前元往前、往后几个元，可以取 Inf（往前、往后所有元）
- .step: 每次函数调用，窗口往前移动的步长
- .complete: 设置两端处是否保留不完整窗口，默认为 FALSE

slider::slide_*() 系列函数与 purrr::map_*() 是类似的，只是将 ‘‘逐元素迭代’’ 换成了 ‘‘逐窗口迭代’’。

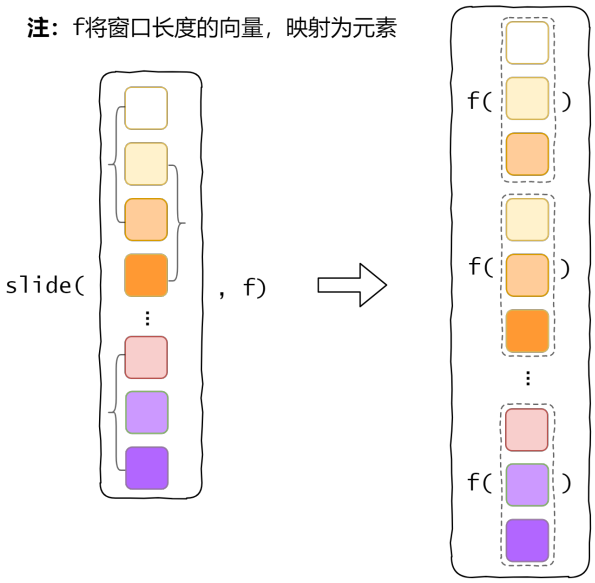

图 2.21: slide 滑窗迭代示意图

金融时间序列数据经常需要计算滑动平均，比如计算 sales 的 3 日滑动平均：

```
library(slider)
dt %>%
  mutate(avg_3 = slide_dbl(sales, mean, .before = 1, .after = 1))
```

```
## # A tibble: 4 x 5
##   day         wday   sales balance avg_3
##   <date>      <chr>  <dbl>   <dbl> <dbl>
## 1 2019-08-30 星期五      2      30  4
## 2 2019-09-03 星期二      6      25  3.33
## 3 2019-09-04 星期三      2     -40  3.67
## 4 2019-09-05 星期四      3      30  2.5
```

输出每个滑动窗口更便于理解该 3 日滑动平均是如何计算的：

```
slide(dt$sales, ~ .x, .before = 1, .after = 1)
```

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

细心的读者可能发现了：上面计算的并不是真正的 3 日滑动平均，而是连续 3 个值的滑动平均。这是因为 `slide()` 函数默认是以行索引来滑动，如果日期也是连续日期这是没有问题的。但是若日期有跳跃，则结果可能不是你想要的。

那么，怎么计算真正的 3 日滑动平均呢？需要改用 `slide_index()` 函数，并提供日期索引，其基本格式为：

```
slide_index(.x, .i, .f, ...)
```

其中参数 `.i` 用来传递索引向量，实现根据 “.i 的当前元 + 其前/后若干元” 创建相应的 `.x` 的滑动窗口。

来看一下的连续 3 日滑动窗口与连续 3 值滑动窗口的区别：

```
slide(dt$day, ~ .x, .before = 1, .after = 1)
```

连续三值，滑动

```
## [[1]]
## [1] "2019-08-30" "2019-09-03"
##
## [[2]]
## [1] "2019-08-30" "2019-09-03" "2019-09-04"
##
## [[3]]
## [1] "2019-09-03" "2019-09-04" "2019-09-05"
##
## [[4]]
## [1] "2019-09-04" "2019-09-05"
```

```
slide_index(dt$day, dt$day, ~ .x, .before = 1, .after = 1)
```

连续3日，滑动

.x .i

```
## [[1]]
## [1] "2019-08-30"
##
## [[2]]
## [1] "2019-09-03" "2019-09-04"
##
## [[3]]
## [1] "2019-09-03" "2019-09-04" "2019-09-05"
##
## [[4]]
## [1] "2019-09-04" "2019-09-05"
```

最后，计算 sales 真正的 3 日滑动平均：

```
dt %>%
  mutate(avg_3 = slide_index_dbl(sales, day, mean, .before = 1, .after = 1))
```

```
## # A tibble: 4 x 5
##   day        wday   sales balance avg_3
##   <date>     <chr>  <dbl>   <dbl> <dbl>
## 1 2019-08-30 星期五     2      30  2
## 2 2019-09-03 星期二     6      25  4
## 3 2019-09-04 星期三     2     -40  3.67
## 4 2019-09-05 星期四     3      30  2.5
```

另外，slide_*() 系列作用在数据框上，默认（窗口长度为 1）是逐行迭代的逻辑，所以，也可以用来做按行汇总。例如计算每个学生三科的总分：

```
df %>%
  mutate(total = slide_dbl(.[4:6], sum))
```

2.6.4 整洁计算

tidyverse 代码之所以这么“整洁、优雅”，访问列只需要提供列名，不需要加引号，不需要加数据框环境 df$, 这是因为它内部采用了一套**整洁计算**（tidy evaluation）框架。

如果我们也想自定义这样的“整洁、优雅”函数，即在自定义函数中页这样“整洁、优雅”地传递参数，就需要掌握一点**整洁计算**的技术。

1. 数据屏蔽与整洁选择

整洁计算的两种基本形式是：

- 数据屏蔽：使得可以不用带数据框（环境变量）名字，就能使用数据框内的变量（数据变量），便于在数据集内计算值

- 整洁选择：即各种选择列语法，便于使用数据集中的列

数据屏蔽为直接使用带来了代码简洁，但作为函数参数时的间接使用，正常是环境变量，要想作为数据变量使用，则需要用两个大括号括起来{{var}}：

```
var_summary = function(data, var) {
  data %>%
    summarise(n = n(), mean = mean({{var}}))
}
mtcars %>%
  group_by(cyl) %>%
  var_summary(mpg)
```

```
## # A tibble: 3 x 3
##     cyl     n  mean
##   <dbl> <int> <dbl>
## 1     4    11  26.7
## 2     6     7  19.7
## 3     8    14  15.1
```

若是字符向量形式，想作为数据变量，则需要在函数体中使用 .data[[var]]，这里 .data 是代替数据集的代词：

```
var_summary = function(data, var) {
  data %>%
    summarise(n = n(), mean = mean(.data[[var]]))
}
mtcars %>%
  group_by(cyl) %>%
  var_summary("mpg")
```

```
## # A tibble: 3 x 3
##     cyl     n  mean
##   <dbl> <int> <dbl>
## 1     4    11  26.7
## 2     6     7  19.7
## 3     8    14  15.1
```

还可用于对列名向量的循环机制，比如对因子型各列计算各水平值频数：

```
mtcars[,9:10] %>%
  names() %>%
  map(~ count(mtcars, .data[[.x]]))
```

```
## [[1]]
##   am  n
## 1  0 19
## 2  1 13
##
## [[2]]
##   gear  n
## 1    3 15
## 2    4 12
## 3    5  5
```

同样地，整洁选择作为函数参数时的间接使用，也需要用两个大括号括起来 {{vars}}：

```
summarise_mean = function(data, vars) {
  data %>%
    summarise(n = n(), across({{vars}}, mean))
}
mtcars %>%
  group_by(cyl) %>%
  summarise_mean(where(is.numeric))
```

```
## # A tibble: 3 x 12
##     cyl     n   mpg  disp    hp  drat    wt  qsec    vs    am  gear
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1     4    11  26.7  105.  82.6  4.07  2.29  19.1 0.909 0.727  4.09
## 2     6     7  19.7  183. 122.   3.59  3.12  18.0 0.571 0.429  3.86
## 3     8    14  15.1  353. 209.   3.23  4.00  16.8 0     0.143  3.29
## # ... with 1 more variable: carb <dbl>
```

若是字符向量形式，则需要借助函数 all_of() 或 any_of()，取决于你的选择：

```
vars = c("mpg", "vs")
mtcars %>% select(all_of(vars))
mtcars %>% select(!all_of(vars))
```

最后，再来看使用数据屏蔽或整洁选择同时修改列名的用法：

```
my_summarise = function(data, mean_var, sd_var) {
  data %>%
    summarise("mean_{{mean_var}}" := mean({{mean_var}}),
              "sd_{{sd_var}}" := mean({{sd_var}}))
}
mtcars %>%
```

```
  group_by(cyl) %>%
  my_summarise(mpg, disp)
```

```
## # A tibble: 3 x 3
##     cyl mean_mpg sd_disp
##   <dbl>    <dbl>   <dbl>
## 1     4     26.7    105.
## 2     6     19.7    183.
## 3     8     15.1    353.
```

```
my_summarise = function(data, group_var, summarise_var) {
  data %>%
    group_by(across({{group_var}})) %>%
    summarise(across({{summarise_var}}, mean, .names = "mean_{.col}"))
}
mtcars %>%
  my_summarise(c(am, cyl), where(is.numeric))
```

```
## # A tibble: 6 x 11
## # Groups:   am [2]
##      am   cyl mean_mpg mean_disp mean_hp mean_drat mean_wt mean_qsec
##   <dbl> <dbl>    <dbl>     <dbl>   <dbl>     <dbl>   <dbl>     <dbl>
## 1     0     4     22.9     136.     84.7      3.77    2.94      21.0
## 2     0     6     19.1     205.    115.       3.42    3.39      19.2
## 3     0     8     15.0     358.    194.       3.12    4.10      17.1
## 4     1     4     28.1      93.6    81.9      4.18    2.04      18.4
## 5     1     6     20.6     155     132.       3.81    2.76      16.3
## 6     1     8     15.4     326     300.       3.88    3.37      14.6
## # ... with 3 more variables: mean_vs <dbl>, mean_gear <dbl>,
## #   mean_carb <dbl>
```

对于字符串列名，同时修改列名的方法：

```
var_summary = function(data, var) {
  data %>%
    summarise(n = n(),
              !!enquo(var) := mean(.data[[var]]))
}
mtcars %>%
 group_by(cyl) %>%
 var_summary("mpg")
```

```
## # A tibble: 3 x 3
##     cyl     n   mpg
##   <dbl> <int> <dbl>
## 1     4    11  26.7
## 2     6     7  19.7
## 3     8    14  15.1
```

```
var_summary = function(data, var) {
  data %>%
    summarise(n = n(),
              !!str_c("mean_", var) := mean(.data[[var]]))
}
mtcars %>%
 group_by(cyl) %>%
 var_summary("mpg")
```

```
## # A tibble: 3 x 3
##     cyl     n mean_mpg
##   <dbl> <int>    <dbl>
## 1     4    11     26.7
## 2     6     7     19.7
## 3     8    14     15.1
```

2. 引用与反引用

创建 tidyverse 风格的整洁函数，另一种做法是使用引用与反引用机制。这需要额外的两个步骤：

- 用 enquo() 让函数自动引用其参数
- 用 !! 反引用该参数

以自定义计算分组均值函数为例：

```
grouped_mean = function(data, summary_var, group_var) {
  summary_var = enquo(summary_var)
  group_var = enquo(group_var)
  data %>%
    group_by(!!group_var) %>%
    summarise(mean = mean(!!summary_var))
}
grouped_mean(mtcars, mpg, cyl)
```

```
## # A tibble: 3 x 2
##     cyl  mean
```

```
##   <dbl> <dbl>
## 1     4  26.7
## 2     6  19.7
## 3     8  15.1
```

要想修改结果列名，可借助 `as_label()` 函数从引用中提取名字：

```
grouped_mean = function(data, summary_var, group_var) {
  summary_var = enquo(summary_var)
  group_var = enquo(group_var)

  summary_nm = str_c("mean_", as_label(summary_var))
  group_nm = str_c("group_", as_label(group_var))

  data %>%
    group_by(!!group_nm := !!group_var) %>%
    summarise(!!summary_nm := mean(!!summary_var))
}
grouped_mean(mtcars, mpg, cyl)
```

```
## # A tibble: 3 x 2
##   group_cyl mean_mpg
##       <dbl>    <dbl>
## 1         4     26.7
## 2         6     19.7
## 3         8     15.1
```

要传递多个参数可以用特殊参数 `...`。比如，我们还想让用于计算分组均值的 `group_var` 可以是任意多个，这就需要改用 `...` 参数，为了更好地应付这种传递特意将该参数放在最后一个位置。另外，将其他函数参数都增加 `.` 前缀是一个好的做法，因为可以降低与 `...` 参数的冲突风险。

```
grouped_mean = function(.data, .summary_var, ...) {
  summary_var = enquo(.summary_var)
  .data %>%
    group_by(...) %>%
    summarise(mean = mean(!!summary_var))
}
grouped_mean(mtcars, disp, cyl, am)
```

```
## # A tibble: 6 x 3
## # Groups:   cyl [3]
##     cyl    am  mean
##   <dbl> <dbl> <dbl>
```

```
## 1     4     0 136.
## 2     4     1  93.6
## 3     6     0 205.
## 4     6     1 155
## 5     8     0 358.
## 6     8     1 326
```

... 参数不需要做引用和反引用就能正确工作，但若要修改结果列名就不行了，仍需要借助引用和反引用，但是要改用 enques() 和 !!!

```
grouped_mean = function(.data, .summary_var, ...) {
  summary_var = enquo(.summary_var)
  group_vars = enquos(..., .named = TRUE)
  summary_nm = str_c("avg_", as_label(summary_var))
  names(group_vars) = str_c("groups_", names(group_vars))
  .data %>%
    group_by(!!!group_vars) %>%
    summarise(!!summary_nm := mean(!!summary_var))
}
grouped_mean(mtcars, disp, cyl, am)
```

```
## # A tibble: 6 x 3
## # Groups:   groups_cyl [3]
##   groups_cyl groups_am avg_disp
##        <dbl>     <dbl>    <dbl>
## 1          4         0    136.
## 2          4         1     93.6
## 3          6         0    205.
## 4          6         1    155
## 5          8         0    358.
## 6          8         1    326
```

另外，参数 ... 也可以传递表达式：

```
filter_fun = function(df, ...) {
  filter(df, ...)
}
mtcars %>%
  filter_fun(mpg > 25 & disp > 90)
```

```
##                mpg cyl  disp  hp drat    wt qsec vs am gear carb
## Porsche 914-2 26.0   4 120.3  91 4.43 2.140 16.7  0  1    5    2
## Lotus Europa  30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
```

最后，再来看一个自定义绘制散点图的模板函数：

```
scatter_plot = function(df, x_var,y_var) {
  x_var = enquo(x_var)
  y_var = enquo(y_var)
  ggplot(data = df, aes(x = !!x_var, y = !!y_var)) +
    geom_point() +
    theme_bw() +
    theme(plot.title = element_text(lineheight = 1, face = "bold", hjust = 0.5)) +
    geom_smooth() +
    ggtitle(str_c(as_label(y_var), " vs. ", as_label(x_var)))
}
scatter_plot(mtcars, disp, hp)
```

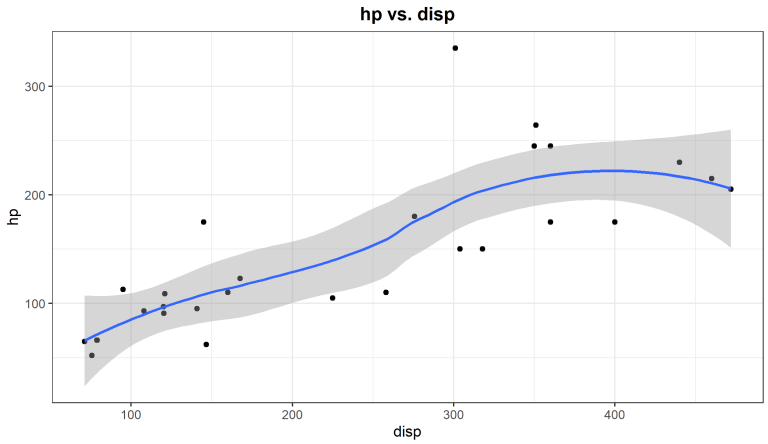

图 2.22: 整洁绘制散点图

本节部分内容参阅 `Vignettes of dplyr`, `Vignettes of slider`, (Lionel Henry 2020b), R-Bloggers, Jesse Cambon: Practical Tidy Evaluation, (Lionel Henry 2020b).

2.7 数据处理神器：data.table 包

`data.table` 包是 `data.frame` 的高性能版本，不依赖其他包就能胜任各种数据操作，速度超快，让个人电脑都能轻松处理几 G 甚至十几 G 的数据。`data.table` 的高性能来源于内存管理（引用语法）、并行化和大量精细优化。

但是，与 `tidyverse` **一次用一个函数做一件事，通过管道依次连接整洁地完成复杂事情**的理念截然不同，`data.table` 语法高度抽象、简洁、一致：

```
dt[i, j, by]
```

图 2.23: data.table 包最简语法

一句话概括：**用 i 选择行，用 j 操作列，根据 by 分组**。

其中，j 表达式非常强大和灵活，可以**选择、修改、汇总、计算新列**，甚至可以接受任意表达式。需要记住的最关键一点是：**只要返回等长元素或长度为 1 元素的 list，每个 list 元素将转化为结果 data.table 的一列**。

data.table 高度抽象的语法无疑增加了学习成本，但它的高效性能和处理大数据能力，使得非常有必要学习它。当然，读者如果既想要 data.table 的高性能，又想要 tidyverse 的整洁语法，也可以借助一些衔接二者的中间包，如 dtplyr，tidyfst 等。

为了节省篇幅，本节将只展示代码演示 data.table 语法，尽量忽略输出结果；有些复杂操作采用同前文一样的数据，得到同样的结果（故略过）。

2.7.1 通用语法

创建 data.table

```
library(data.table)
dt = data.table(
  x = 1:2,
  y = c("A", "B"))
dt
```

```
##    x y
## 1: 1 A
## 2: 2 B
```

用 as.data.table() 可将数据框、列表、矩阵等转化为 data.table；若只想按引用转化，用 setDT().

引用语法

高效计算的编程都支持引用语法，也叫浅拷贝。

浅拷贝[9]，只是拷贝列指针向量（对应数据框的列），而实际数据在内存中不做物理拷贝；而相对的概念：深拷贝，则拷贝整个数据到内存中的另一位置，深拷贝这种冗余的拷贝极大地影响性能，特别是大数据的情形。

data.table 使用 := 算符，做整列或部分列替换时都不做任何拷贝，因为 := 算符是通过引用就地（in-place）更新 data.table 的列。

若想要复制不想按引用（修改数据本身），使用 DT2 = copy(DT1).

键与索引

data.table 支持设置**键**和**索引**，使得选择行、做数据连接更加方便快速（快 170 倍）。

[9]引用语法，相当于是对象只有一个在内存放着，不做多余复制，用两个指针都指向该同一对象，操作哪个指针，都是在修改该同一对象.

- 键：一级有序索引
- 索引：自动二级索引

```
setkey(dt, v1, v3)          # 设置键
setindex(dt, v1, v3)        # 设置索引
```

二者的主要不同：

- 使用键时，数据在内存中做物理重排序；而使用索引时，顺序只是保存为属性；
- 键是显式定义的；索引可以手动创建，也可以运行时创建（比如用 `==` 或 `%in%` 时）；
- 索引与参数 `on` 连用；键的使用是可选的，但为了可读性建议使用键。

特殊符号

`data.table` 提供了一些辅助操作的特殊符号：

- `.()`: 代替 `list()`
- `:=`: 按引用方式增加、修改列
- `.N`: 行数
- `.SD`: 每个分组的数据子集，除了 `by` 或 `keyby` 的列
- `.SDcols`: 与 `.SD` 连用，用来选择包含在`.SD` 中的列
- `.BY`: 包含所有 `by` 分组变量的 `list`
- `.I`: 整数向量 `seq_len(nrow(x))`，例如 `DT[, .I[which.max(somecol)], by=grp]`
- `.GRP`: 分组索引，1 代表第 1 分组，2 代表第 2 分组，...
- `.NGRP`: 分组数
- `.EACHI`: 用于 `by/keyby = .EACHI` 表示根据 `i` 表达式的每一行分组

链式操作

`data.table` 也有自己专用的管道操作，称为链式操作：

```
DT[…][…][…]                 # 或者写开为
DT[
   …
  ][
    …
    ][
      …
      ]
```

2.7.2 数据读写

函数 `fread()` 和 `fwrite()` 是 `data.table` 最强大的函数之二。它们最大的优势，仍是读取大数据时速度超快（100 倍），且非常稳健，分隔符、列类型、行数都是自动检测；它们非常通用，可以处理

不同的文件格式（但不能直接读取 Excel 文件），还可以接受 URLs 甚至是操作系统指令。

读入数据

```
fread("DT.csv")
fread("DT.txt", sep = "\t")
# 选择部分行列读取
fread("DT.csv", select = c("V1", "V4"))
fread("DT.csv", drop = "V4", nrows = 100)
# 读取压缩文件
fread(cmd = "unzip -cq myfile.zip")
fread("myfile.gz")
# 批量读取
c("DT.csv", "DT.csv") %>%
  lapply(fread) %>%
  rbindlist()                # 多个数据框/列表按行合并
```

写出数据

```
fwrite(DT, "DT.csv")
fwrite(DT, "DT.csv", append = TRUE)            # 追加内容
fwrite(DT, "DT.txt", sep = "\t")
fwrite(setDT(list(0, list(1:5))), "DT2.csv")   # 支持写出列表列
fwrite(DT, "myfile.csv.gz", compress = "gzip") # 写出到压缩文件
```

> vroom 包提供了速度更快的文件读写函数：`vroom()`, `vroom_write()`.

2.7.3 数据连接

`data.table` 提供了简单的按行合并函数：

- `rbind(DT1, DT2, ...)`: 按行堆叠多个 `data.table`
- `rbindlist(DT_list, idcol)`: 堆叠多个 `data.table` 构成的 `list`

最常用的六种数据连接：左连接、右连接、内连接、全连接、半连接、反连接。

左连接

外连接至少保留一个数据表中的所有观测，分为左连接、右连接、全连接，其中最常用的是左连接：保留 x 所有行，合并匹配的 y 中的列。

```
y[x, on = "v1"]                          # 注意是以 x 为左表
y[x]                                     # 若 v1 是键
merge(x, y, all.x = TRUE, by = "v1")
```

> 若表 x 与 y 中匹配列的列名不同，可以用 by.x = "c1", by.y = "c2", 多个匹配列，套 c() 即可。

上面代码提供了左连接的三种不同实现，为了易记性和可读性, 更建议用第三种 merge() 函数。

> 注意，只要加载了 data.table 包，将自动用更快速的 data.table::merge(), 而不是 R base 中的 merge(), 二者语法相同。

右连接

保留 y 所有行，合并匹配的 x 中的列：

```
merge(x, y, all.y = TRUE, by = "v1")
```

内连接

内连接是保留两个数据表中所共有的观测：只保留 x 中与 y 匹配的行，合并匹配的 y 中的列：

```
merge(x, y, by = "v1")
```

全连接

保留 x 和 y 中的所有行，合并匹配的列：

```
merge(x, y, all = TRUE, by = "v1")
```

半连接

根据在 y 中，来筛选 x 中的行：

```
x[y$v1, on = "v1", nomatch = 0]
```

反连接

根据不在 y 中，来筛选 x 中的行：

```
x[!y, on = "v1"]
```

集合运算

```
fintersect(x, y)
fsetdiff(x, y)
```

```
funion(x, y)
fsetequal(x, y)
```

非等连接

通常的数据连接是等值连接，即匹配列的值相等，才认为匹配成功，再将匹配成功的其他列连接进来。

很多时候需要非等连接，相当于是按条件连接，即匹配列的值不要求必须相等，只要满足一定条件就认为是匹配成功，再将匹配成功的其他列连接进来。

非等连接的实用场景有很多，下面以房屋租赁数据集为例演示。

```
Houses                  # 房源信息
```

```
##    id district          address bedrooms rent
## 1:  1    South   Rose Street, 5        4 3000
## 2:  2    North  Main Street, 12        3 2250
## 3:  3    South   Rose Street, 5        4 3000
## 4:  4     West   Nice Street, 3        2 1750
## 5:  5     West  Park Avenue, 10        4 3500
## 6:  6    South Little Street, 7        4 3000
## 7:  7    North   Main Street, 8        3 2100
```

```
Renters                 # 租客信息
```

```
##    id          name preferred min_bedrooms min_rent max_rent
## 1:  1    Helen Boss     South            3     2500     3200
## 2:  2  Michael Lane      West            2     1500     2500
## 3:  3 Susan Sanders      West            4     2500     4000
## 4:  4     Tom White     North            3     2200     2500
## 5:  5   Sofia Brown     North            3     1800     2300
```

```
Deals                   # 已租赁信息
```

```
##    id      date renter_id house_id agent_fee
## 1:  1 2020/1/30         1        1       600
## 2:  2  2020/2/3         2        4       350
## 3:  3 2020/3/12         3        5       700
## 4:  4 2020/4/10         4        2       450
```

- 找出可以合租的人，即租客具有相同的首选区域

这是 Renters 表的自连接，根据首选区域等值匹配，非等匹配 `id < id` 避免两两租客的重复出现：

```
Renters[Renters, on = .(preferred, id < id)][
  , .(name, preferred, i.name)
] %>%
  na.omit()
```

```
##            name preferred        i.name
## 1: Michael Lane      West Susan Sanders
## 2:    Tom White     North   Sofia Brown
```

同样的操作，还可以用来识别重复，比如找出重复登记房屋，即地址相同, 房屋 id 不同。

- 找出满足租客要求的可用房源，需要满足
  - 是租客的首选区域
  - 在租客接受的价格范围内
  - 卧室数满足租客所需
  - 房屋还没有被租赁（即不在 Deals 表中)

以 Renters 为左表，连接表是经未被租赁筛选后的 Houses 表，租客要求的条件体现在非等匹配中：房屋地区等值匹配租客首选地区，房屋租金介于租客能接受的最低租金和最高租金之间，房屋卧室数大于租客最小需要。

```
Houses[! id %in% Deals$house_id][
  Renters, on = .(district == preferred, rent >= min_rent,
                  rent <= max_rent, bedrooms >= min_bedrooms)
  ][, -(5:6)] %>%
  na.omit()
```

```
##    id district          address bedrooms i.id        name
## 1:  3    South   Rose Street, 5        3    1  Helen Boss
## 2:  6    South Little Street, 7        3    1  Helen Boss
## 3:  7    North   Main Street, 8        3    5 Sofia Brown
```

滚动连接

滚动连接也是很有用的一种数据连接，往往涉及日期时间，特别是处理在时间上有先后关联的两个事件。基本语法为：

```
x[y, on = .(id = id, date = date), roll = TRUE]   # 同 roll = inf
```

根据 `id` 等值匹配行，`date` 是滚动匹配，匹配与左表 `y` 日期最接近的前一个日期，匹配成功的列合并进来；`roll = -inf` 则匹配最接近的后一个日期。

下面以网页会话与支付这一对关联事件为例来演示。

```
website                          # 网页会话数据
```

```
##        name        session_time session_id
##  1:  Isabel 2016-01-01 03:01:00          1
##  2:  Isabel 2016-01-02 00:59:00          2
##  3:  Isabel 2016-01-05 10:18:00          3
##  4:  Isabel 2016-01-07 11:03:00          4
##  5:  Isabel 2016-01-08 11:01:00          5
##  6:   Sally 2016-01-03 02:00:00          6
##  7: Francis 2016-01-02 05:09:00          7
##  8: Francis 2016-01-03 11:22:00          8
##  9: Francis 2016-01-08 00:44:00          9
## 10: Francis 2016-01-08 12:22:00         10
## 11: Francis 2016-01-10 09:36:00         11
## 12: Francis 2016-01-15 08:56:00         12
## 13:   Erica 2016-01-04 11:12:00         13
## 14:   Erica 2016-01-04 13:05:00         14
## 15:  Vivian 2016-01-01 01:10:00         15
## 16:  Vivian 2016-01-08 18:15:00         16
```

```
paypal                           # 支付数据
```

```
##       name       purchase_time payment_id
## 1:  Isabel 2016-01-08 11:10:00          1
## 2:   Sally 2016-01-03 02:06:00          2
## 3:   Sally 2016-01-03 02:15:00          3
## 4: Francis 2016-01-03 11:28:00          4
## 5: Francis 2016-01-08 12:33:00          5
## 6: Francis 2016-01-10 09:46:00          6
## 7:   Erica 2016-01-03 00:02:00          7
## 8:     Mom 2015-12-02 09:58:00          8
```

```
# 创建用于连接的单独时间列
website[, join_time:=session_time]
paypal[, join_time:=purchase_time]
# 设置 key
setkey(website, name, join_time)
setkey(paypal, name, join_time)
```

- **前滚连接**：查看哪些客户在每次支付前进行了网页会话？

由于已经设置了键，故可以省略 on 语句。以 paypal 为左表，先等值匹配 id，若同一 id 的支付时

间之前有网页会话时间，则匹配成功，合并右表的新列进来：

```
website[paypal, roll = TRUE] %>%
  na.omit()
```

```
##       name        session_time session_id           join_time
## 1: Francis 2016-01-03 11:22:00          8 2016-01-03 11:28:00
## 2: Francis 2016-01-08 12:22:00         10 2016-01-08 12:33:00
## 3: Francis 2016-01-10 09:36:00         11 2016-01-10 09:46:00
## 4:  Isabel 2016-01-08 11:01:00          5 2016-01-08 11:10:00
## 5:   Sally 2016-01-03 02:00:00          6 2016-01-03 02:06:00
## 6:   Sally 2016-01-03 02:00:00          6 2016-01-03 02:15:00
##          purchase_time payment_id
## 1: 2016-01-03 11:28:00          4
## 2: 2016-01-08 12:33:00          5
## 3: 2016-01-10 09:46:00          6
## 4: 2016-01-08 11:10:00          1
## 5: 2016-01-03 02:06:00          2
## 6: 2016-01-03 02:15:00          3
```

- **后滚连接**：哪些网页会话之后产生了支付？

以 website 为左表，先等值匹配 id，若同一 id 的网页会话时间之后有支付时间，则匹配成功，合并右表的新列进来：

```
paypal[website, roll = -Inf] %>%
  na.omit()
```

```
##        name       purchase_time payment_id           join_time
##  1: Francis 2016-01-03 11:28:00          4 2016-01-02 05:09:00
##  2: Francis 2016-01-03 11:28:00          4 2016-01-03 11:22:00
##  3: Francis 2016-01-08 12:33:00          5 2016-01-08 00:44:00
##  4: Francis 2016-01-08 12:33:00          5 2016-01-08 12:22:00
##  5: Francis 2016-01-10 09:46:00          6 2016-01-10 09:36:00
##  6:  Isabel 2016-01-08 11:10:00          1 2016-01-01 03:01:00
##  7:  Isabel 2016-01-08 11:10:00          1 2016-01-02 00:59:00
##  8:  Isabel 2016-01-08 11:10:00          1 2016-01-05 10:18:00
##  9:  Isabel 2016-01-08 11:10:00          1 2016-01-07 11:03:00
## 10:  Isabel 2016-01-08 11:10:00          1 2016-01-08 11:01:00
## 11:   Sally 2016-01-03 02:06:00          2 2016-01-03 02:00:00
##            session_time session_id
##  1: 2016-01-02 05:09:00          7
##  2: 2016-01-03 11:22:00          8
```

```
##  3: 2016-01-08 00:44:00          9
##  4: 2016-01-08 12:22:00         10
##  5: 2016-01-10 09:36:00         11
##  6: 2016-01-01 03:01:00          1
##  7: 2016-01-02 00:59:00          2
##  8: 2016-01-05 10:18:00          3
##  9: 2016-01-07 11:03:00          4
## 10: 2016-01-08 11:01:00          5
## 11: 2016-01-03 02:00:00          6
```

- **滚动窗口连接**：哪些支付发生在网页会话之后的 12 小时之内？

同前面一样，只是多了对时间之后的时间窗口有限制，时间窗口之外的不再成功匹配：

```
twelve_hours = 60*60*20    # 转化为秒
paypal[website, roll = -twelve_hours] %>%
  na.omit()
```

```
##       name       purchase_time payment_id           join_time
## 1: Francis 2016-01-03 11:28:00          4 2016-01-03 11:22:00
## 2: Francis 2016-01-08 12:33:00          5 2016-01-08 00:44:00
## 3: Francis 2016-01-08 12:33:00          5 2016-01-08 12:22:00
## 4: Francis 2016-01-10 09:46:00          6 2016-01-10 09:36:00
## 5:  Isabel 2016-01-08 11:10:00          1 2016-01-08 11:01:00
## 6:   Sally 2016-01-03 02:06:00          2 2016-01-03 02:00:00
##           session_time session_id
## 1: 2016-01-03 11:22:00          8
## 2: 2016-01-08 00:44:00          9
## 3: 2016-01-08 12:22:00         10
## 4: 2016-01-10 09:36:00         11
## 5: 2016-01-08 11:01:00          5
## 6: 2016-01-03 02:00:00          6
```

另外，还有参数 `rollends` 为二元逻辑向量，可设置是否对每个分组第 1 个值之前、最后 1 个值之后的值滚动。

2.7.4 数据重塑

宽变长

宽表的特点是：表比较宽，本来该是“值”的，却出现在“变量（名）”中。这就需要给它变到“值”中，新起个列名存为一列，这就是所谓的宽表变长表。

- 每一行只有 1 个观测的情形

```
DT = fread("datas/分省年度 GDP.csv", encoding = "UTF-8")
DT %>%
  melt(measure = 2:4, variable = " 年份", value = "GDP")
```

参数 measure 是用整数向量指定要变形的列，也可以使用正则表达式 patterns(" 年 $"), 也可以改用参数 id 指定不变形的列；若需要忽略缺失值，设置参数 na.rm = TRUE.

对比 tidyr::pivot_longer() 实现：

```
DT %>%
  pivot_longer(-地区, names_to = " 年份", values_to = "GDP")
```

两种语法基本相同，都是指定要变形的列、为存放变形列的列名中的'' 值'' 指定新列名、为存放变形列中的'' 值'' 指定新列名。

- 每一行有多个观测的情形

```
load("datas/family.rda")
DT = as.data.table(family)          # family 数据
DT %>%
  melt(measure = patterns("^dob", "^gender"),
       value = c("dob", "gender"), na.rm = TRUE)
```

长变宽

长表的特点是：表比较长。

有时候需要将分类变量的若干水平值，变成变量（列名）。这就是长表变宽表，它与宽表变长表正好相反（二者互逆）。

- 只有 1 个列名列和 1 个值列的情形

```
load("datas/animals.rda")
DT = as.data.table(animals)           # 农场动物数据
DT %>%
  dcast(Year ~ Type, value = "Heads", fill = 0)
```

对比 tidyr::pivot_wider() 实现：

```
DT %>%
  pivot_wider(names_from = Type, values_from = Heads, values_fill = 0)
```

dcast() 函数第 1 个参数是公式形式，~ 左边是不变的列，右边是'' 变量名'' 来源列，参数 value 指定'' 值'' 的来源列。

- 有多个列名列和多个值列的情形

```
us_rent_income %>%
  as.data.table() %>%
  dcast(GEOID + NAME ~ variable, value = c("estimate", "moe"))
```

数据分割与合并

函数 `split(DT, by)` 可将 data.table 分割为 list, 然后就可以接 `map_*()` 函数逐分组迭代。

- 拆分列

```
DT = as.data.table(table3)
# 将 case 列拆分为两列，并删除原列
DT[, c("cases", "population") := tstrsplit(DT$rate, split = "/")][,
                                                    rate := NULL]
```

- 合并列

```
DT = as.data.table(table5)
# 将 century 和 year 列合并为新列 new，并删除原列
DT[, new := paste0(century, year)][, c("century", "year") := NULL]
```

2.7.5 数据操作

选择行

用 `i` 表达式，选择行。

- 根据索引

```
dt[3:4,]                    # 或 dt[3:4]
dt[!3:7,]                   # 反选，或 dt[-(3:7)]
```

- 根据逻辑表达式

```
dt[v2 > 5]
dt[v4 %chin% c("A","C")]    # 比 %in% 更快
dt[v1==1 & v4=="A"]
```

- 删除重复行

```
unique(dt)
unique(dt, by = c("v1","v4"))              # 返回所有列
```

- 删除包含 NA 的行

```
na.omit(dt, cols = 1:4)
```

- 行切片

```
dt[sample(.N, 3)]                                # 随机抽取 3 行
dt[sample(.N, .N * 0.5)]                         # 随机抽取 50% 的行
dt[frankv(-v1, ties.method = "dense") < 2]       # v1 值最大的行
```

- 其他

```
dt[v4 %like% "^B"]                               # v4 值以 B 开头
dt[v2 %between% c(3,5)]                          # 闭区间
dt[between(v2, 3, 5, incbounds = FALSE)]         # 开区间
dt[v2 %inrange% list(-1:1, 1:3)]                 # v2 值属于多个区间的某个
dt[inrange(v2, -1:1, 1:3, incbounds = TRUE)] # 同上
```

排序行

```
dt[order(v1)]                                    # 默认按 v1 从小到大
dt[order(-v1)]                                   # 按 v1 从大到小
dt[order(v1, -v2)]                               # 按 v1 从小到大, v2 从大到小
```

若按引用对行重排序：

```
setorder(DT, V1, -V2)
```

> data.table 包还提供了函数 fsort() 和 frank(), R base 中 sort() 和 rank() 的快速版本。

操作列

用 j 表达式操作列。

- 选择一列或多列

```
# 根据索引
dt[[3]]                                  # 或 dt[["v3"]], dt$v3, 返回向量
dt[, 3]                                  # 或 dt[, "v3"], 返回 data.table
# 根据列名
dt[, .(v3)]                              # 或 dt[, list(v3)]
dt[, .(v2,v3,v4)]
dt[, v2:v4]
dt[, !c("v2","v3")]                      # 反选列
```

- 反引用列名

tidyverse 提供了丰富的选择列的辅助函数，而 data.table 需要字符串函数、正则表达式构造出列名向量，再通过反引用选择相应的列。

```
cols = c("v2", "v3")
dt[, ..cols]
dt[, !..cols]

cols = paste0("v", 1:3)              # v1, v2, ...
cols = union("v4", names(dt))        # v4 列提到第 1 列
cols = grep("v", names(dt))          # 列名中包含"v"
cols = grep("^(a)", names(dt))       # 列名以"a" 开头
cols = grep("b$", names(dt))         # 列名以"b" 结尾
cols = grep(".2", names(dt))         # 正则匹配".2" 的列
cols = grep("v1|X", names(dt))       # v1 或 x
dt[, ..cols]
```

- 调整列序

```
cols = rev(names(DT))               # 或其他列序
setcolorder(DT, cols)
```

- 修改列名

```
setnames(DT, old, new)
```

- 修改因子水平

```
DT[, setattr(sex, "levels", c("M", "F"))]
```

tidyverse 是用 mutate() 修改列，不修改原数据框，必须赋值结果；data.table 修改列，是用列赋值符号 := （不执行复制），直接对原数据框修改。

- 修改或增加一列

```
dt[, v1 := v1 ^ 2][]                    # 修改列，加 [] 输出结果
dt[, v2 := log(v1)]                     # 增加新列
dt[, .(v2 = log(v1), v3 = v2 + 1)]      # 只保留新列
```

注意，代码 v3 = v2 + 1 中的 v2 是原始的 v2 列，而不是前面新计算的 v2 列；若想使用新计算的列，可以用

```
dt[, c("v2", "v3") := .(temp <- log(v1), v3 = temp + 1)]
```

- 增加多列

```
dt[, c("v6","v7") := .(sqrt(v1), "x")]  # 或者
dt[, ':='(v6 = sqrt(v1),
          v7 = "x")]                    # v7 列的值全为 x
```

- 同时修改多列

tidyverse 是借助 across() 或 _all, _if, _at 后缀选择并同时操作多列；而 data.table 选择并操作多列是借助 lapply() 以及特殊符号：

- .SD: 每个分组的数据子集，除了 by 或 keyby 的列
- .SDcols: 与 .SD 连用，用来选择包含在 .SD 中的列，支持索引、列名、连选、反选、正则表达式、条件判断函数

```
# 使用不带 NA 的考试成绩数据
DT = readxl::read_xlsx("datas/ExamDatas.xlsx") %>%
  as.data.table()
```

```
# 应用函数到所有列
DT[, lapply(.SD, as.character)]
# 应用函数到满足条件的列
DT[, lapply(.SD, rescale),      # rescale() 为自定义的归一化函数
   .SDcols = is.numeric]
# 应用函数到指定列
DT =  as.data.table(iris)
DT[, .SD * 10, .SDcols = patterns("(Length)|(Width)")]
```

> 注意，上述同时修改多列的代码，都是只保留新列，若要保留所有列，需要准备新列名 cols, 再在 j 表达式中使用 (cols) := ...

- 删除列

```
dt[, v1 := NULL]
dt[, c("v2","v3") := NULL]
cols = c("v2","v3")
dt[, (cols) := NULL]    # 注意，不是 dt[, cols := NULL]
```

- 重新编码

```
# 一分支
dt[v1 < 4, v1 := 0]
# 二分支
dt[, v1 := fifelse(v1 < 0, -v1, v1)]
# 多分支
dt[, v2 := fcase(v2 < 4, "low",
                 v2 < 7, "middle",
                 default = "high")]
```

- 前移/后移运算

```
shift(x, n = 1, fill = NA, type = "lag")      # 1,2,3 -> NA,1,2
shift(x, n = 1, fill = NA, type = "lead")     # 1,2,3 -> 2,3,NA
```

2.7.6 分组汇总

用 by 表达式指定分组。

data.table 是根据 by 或 keyby 分组，区别是，keyby 会排序结果并创建键，使得更快地访问子集。

未分组数据框相当于整个数据框作为 1 组，数据操作是在整个数据框上进行，若汇总则得到的是 1 个结果。

分组数据框，相当于整个数据框分成了 m 个数据框，数据操作是分别在每个数据框上进行，若汇总则得到的是 m 个结果。

```
# 使用带 NA 值的考试成绩数据
DT = readxl::read_xlsx("datas/ExamDatas_NAs.xlsx") %>%
  as.data.table()
```

- 未分组汇总

```
DT[, .(math_avg = mean(math, na.rm = TRUE))]
```

```
##    math_avg
## 1: 68.04255
```

- 简单的分组汇总

```
DT[, .(n = .N,
       math_avg = mean(math,  na.rm = TRUE),
       math_med = median(math)),
   by = sex]
```

```
##     sex  n math_avg math_med
## 1:   女 25 70.78261       NA
## 2:   男 24 64.56522       NA
## 3: <NA>  1 85.00000       85
```

可以直接在 by 中使用判断条件或表达式，特别是根据整合单位的日期时间汇总：

```
date = as.IDate("2021-01-01") + 1:50
DT = data.table(date, a = 1:50)
DT[, mean(a), by = list(mon = month(date))]   # 按月平均
```

data.table 提供快速处理日期时间的 IDateTime 类，更多信息可查阅帮助。

- 对某些列做汇总

```
DT[, lapply(.SD, mean), .SDcols = patterns("h"),
   by = .(class, sex)]              # 或用 by = c("class", "sex")
```

- 对所有列做汇总

```
DT[, name := NULL][, lapply(.SD, mean, na.rm = TRUE),
                    by = .(class, sex)]
```

- 对满足条件的列做汇总

```
DT[, lapply(.SD, mean, na.rm = TRUE), by = class,
   .SDcols = is.numeric]
```

- 分组计数

```
DT = na.omit(DT)
DT[, .N, by = .(class, cut(math, c(0, 60, 100)))] %>%
  print(topn = 2)
```

```
##     class      cut N
##  1: 六1班 (60,100] 5
##  2: 六1班   (0,60] 1
## ---
##  8: 六5班 (60,100] 5
##  9: 六5班   (0,60] 1
```

上述分组计数会忽略频数为 0 的分组，若要显示出来可以用：

```
DT[, Bin := cut(math, c(0, 60, 100))]
DT[CJ(class = class, Bin = Bin, unique = TRUE),
   on = c("class","Bin"), .N, by = .EACHI]
```

其中，函数 `CJ()` 相当于 `expand_grid()`, 生成所有两两组合（笛卡儿积）。

- 分组选择行

`data.table` 也提供了辅助函数：`first()`, `last()`, `uniqueN()`. 比如提取每组的 `first/nth` 观测：

```
DT[, first(.SD), by = class]
DT[, .SD[3], by = class]              # 每组第 3 个观测
DT[, tail(.SD, 2), by = class]        # 每组后 2 个观测
# 选择每个班男生数学最高分的观测
DT[sex == " 男", .SD[math == max(math)], by = class]
```

最后注

本节是分别按 `i`，`j`，`by` 讲解语法，真正使用的时候，是三者组合起来使用，即同时**对 `i` 选择的行，根据 `by` 分组，做 `j` 操作**;
`data.table` 中习惯用 `lapply()`, 换成 `map()` 也是一样的，好处是支持函数的 `purrr`-风格公式写法。

本节部分内容参阅 `Vignettes of data.table`, Atrebas: A data.table and dplyr tour, Learning data.table eBook, Robert Norberg: Understanding data.table Rolling Joins, Kateryna Koidan: Practical Examples of When to Use Non-Equi JOINs in SQL.

第三章 可视化与建模技术

可视化历来是 R 语言的强项，本章将介绍最经典的 ggplot2 绘图。统计建模技术，将围绕整洁模型结果、建模辅助函数、批量建模展开。

3.1 ggplot2 基础语法

3.1.1 ggplot2 概述

ggplot2 是最流行的 R 可视化包，最初是 Hadley Wickham 读博期间的作品；ggplot2 基于图层化语法：图形是一层一层的图层叠加而成，先进的绘图理念、优雅的语法代码、美观大方的生成图形，让 ggplot2 迅速走红。

ggplot2 几乎是 R 语言的代名词，提起 R 语言，人们首先想到的是 R 强大的可视化：**ggplot2**；这曾经无比正确，现在我希望提起 R 语言，人们首先想到的是 tidyverse (ggplot2 扩展到整个数据科学流程)。

ggplot2 绘图语法

ggplot2 的绘图语法是从数据产生图形的一系列语法：

> 选取整洁**数据**将其**映射**为**几何对象** (如点、线等)，几何对象具有**美学特征** (如坐标轴、颜色等)，若需要则对数据做**统计变换**，**调整标度**，将结果投影到**坐标系**，再根据喜好选择**主题**。

先来看一下，ggplot2 基本的绘图流程示意图，来自 rstudio-conf-2020, Kieran Healy: dataviz.

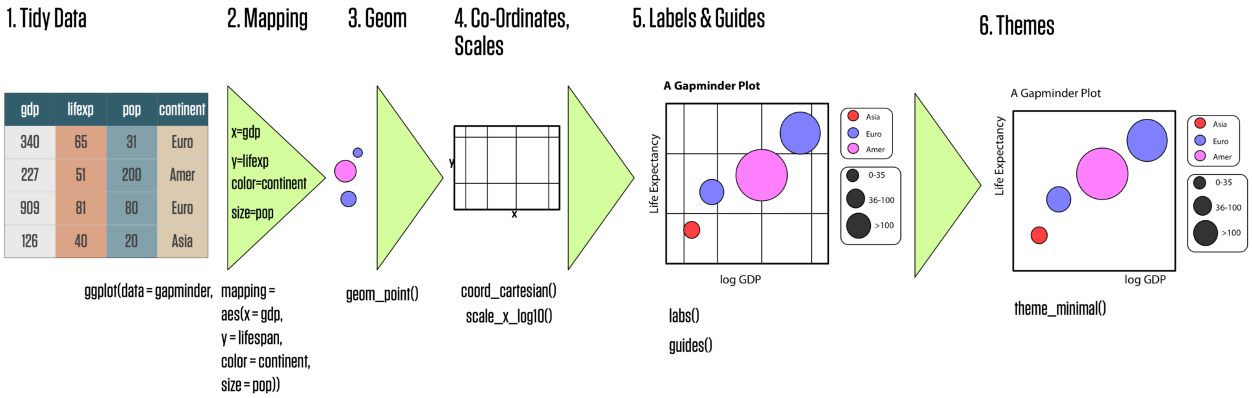

图 3.1: ggplot2 绘图流程

ggplot 的语法包括 10 个部件：

- **数据 (data)**
- **映射 (mapping)**
- **几何对象 (geom)**

- 标度 (scale)
- 统计变换 (stats)
- 坐标系 (coord)
- 位置调整 (Position adjustments)
- 分面 (facet)
- 主题 (theme)
- 输出 (output)

10 个部件中，前 3 个是必须的，其他部件 ggplot2 会自动帮你做好它认为'' 最优'' 的配置，当然也都可以手动定制。

ggplot2 基本绘图模板：

```
ggplot(data = <DATA>,
       mapping = aes(<MAPPINGS>)) +
       <GEOM_FUNCTION>(
                      mapping = aes(<MAPPINGS>),
                      stat = <STAT>,
                      position = <POSITION>) +
       <SCALE_FUNCTION> +
       <COORDINATE_FUNCTION> +
       <FACET_FUNCTION> +
       <THEME_FUNCTION>
```

注意：添加图层的加号只能放在行尾，而不能放在下一行行头。

3.1.2 数据、映射、几何对象

数据 (data)

数据：用于绘图的数据，需要是整洁的数据框。本节用 ggplot2 自带数据集演示。

```
library(tidyverse)
head(mpg, 4)
```

```
## # A tibble: 4 x 11
##   manufacturer model displ  year   cyl trans  drv     cty   hwy fl
##   <chr>        <chr> <dbl> <int> <int> <chr>  <chr> <int> <int> <chr>
## 1 audi         a4      1.8  1999     4 auto(~ f        18    29 p
## 2 audi         a4      1.8  1999     4 manua~ f        21    29 p
## 3 audi         a4      2    2008     4 manua~ f        20    31 p
## 4 audi         a4      2    2008     4 auto(~ f        21    30 p
## # ... with 1 more variable: class <chr>
```

用 ggplot() 创建一个坐标系统，先只提供数据，此时只是创建了一个空的图形：

```
ggplot(data = mpg)
```

映射 (mapping)

函数 aes() 是 ggplot2 中的映射函数, 所谓映射就是将数据集中的变量数据映射 (关联) 到相应的图形属性，也称为 ‘‘美学映射’’ 或 ‘‘美学’’。

映射：指明了变量与图形所见元素之间的联系，告诉 ggplot 图形元素想要关联哪个变量数据；

最常用的映射 (美学) 有：

- x：x 轴
- y：y 轴
- color：颜色
- size：大小
- shape：形状
- fill：填充
- alpha：透明度

最需要的美学是 x 和 y, 分别映射到变量 displ 和 hwy, 再将美学 color 映射到 drv，此时图形就有了坐标轴和网格线，color 美学在绘制几何对象前还体现不出来：

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy, color = drv))
```

注意：映射不是直接为出现在图形中的颜色、外形、线型等设定特定值，而是建立数据中的变量与可见的图形元素之间的联系，经常将图形的美学 color，size 等映射到数据集的分类变量，以实现不同分组用不同的美学来区分。所以，若要为美学指定特定值，比如 color = "red", 是不能放在映射 aes() 中的。

几何对象 (Geometric)

每个图形都是采用不同的视觉对象来表达数据，称为是 ‘‘几何对象’’。

通常用不同类型的 ‘‘几何对象’’ 从不同角度来表达数据，如散点图、平滑曲线、线形图、条形图、箱线图等。

ggplot2 提供了 50 余种 ‘‘几何对象’’，均以 geom_xxxx() 的方式命名，常用的有：

- geom_point(): 散点图
- geom_line(): 折线图
- geom_smooth(): 光滑 (拟合) 曲线
- geom_bar()/geom_col(): 条形图
- geom_histogram(): 直方图
- geom_density(): 概率密度图

- `geom_boxplot()`: 箱线图
- `geom_abline()`: 参考直线

要绘制几何对象，就是添加图层即可。先来绘制散点图，为了简洁省略前文已知的函数参数名：

```
ggplot(mpg, aes(displ, hwy, color = drv)) +
  geom_point()
```

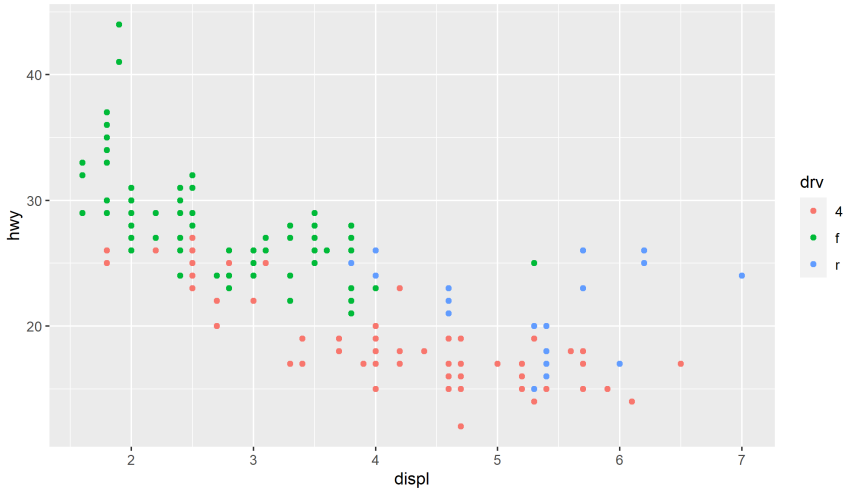

图 3.2: 简单的分组散点图

不同的几何对象支持的美学会有些不同，美学映射也可以放在几何对象中，上面代码可改写为：

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = drv))
```

前面提到，为图形美学设置特定值也是可以的，但不能放在映射 `aes()` 中：

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(color = "blue")
```

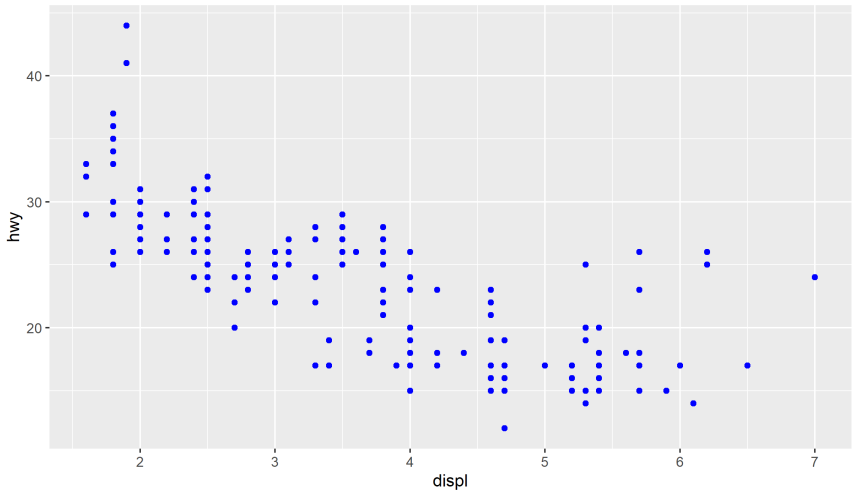

图 3.3: 手动设置颜色的散点图

图层是可以依次叠加的，再添加一个几何对象：光滑曲线，然后来区分一下如下两个图形：

```
ggplot(mpg, aes(displ, hwy, color = drv)) +
  geom_point() +
  geom_smooth()
```

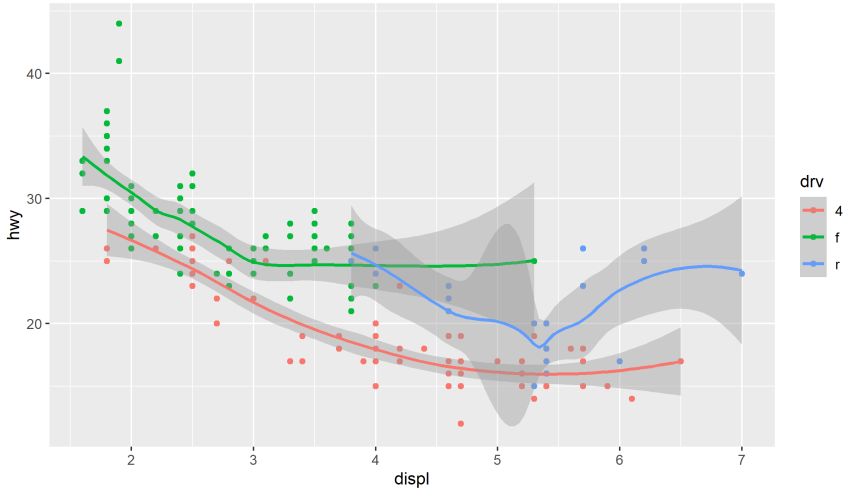

图 3.4: 带分组光滑曲线的散点图

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = drv)) +
  geom_smooth()
```

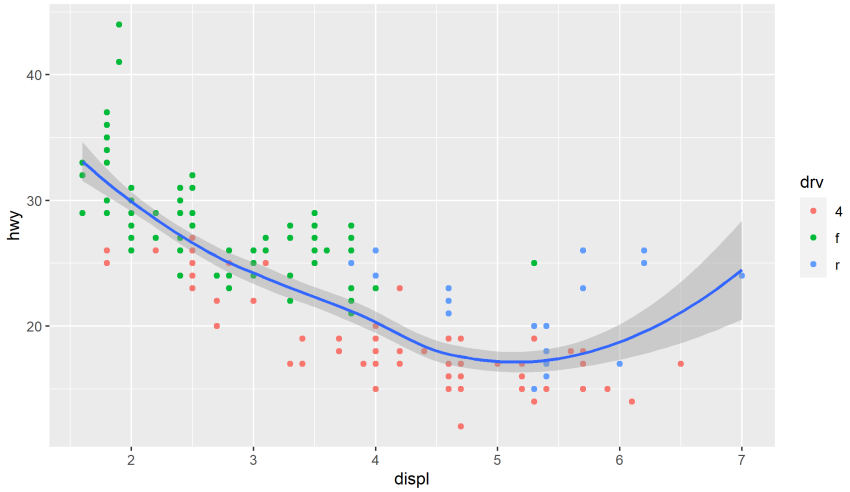

图 3.5: 带全局光滑曲线的散点图

为什么会出现这种不同呢？这就涉及 ggplot2 ‘‘全局’’ 与 ‘‘局部’’ 的约定：

- ggplot() 中的数据和映射，是全局的，可供所有几何对象共用；
- 而位于’’ 几何对象’’ 中的数据和映射，是局部的，只供该几何对象使用；
- ‘‘几何对象’’ 优先使用局部的，局部没有则用全局的。

关于分组美学 (group)

在前例中，用 aes(color = drv) 将颜色映射到分类变量 drv, 实际上就是实现了一种分组，对不

同 drv 值的数据，按不同颜色分别绘图。

但是下面这种情况，对世界各国预期寿命数据集 gapminder, 绘制预期寿命与年份之间的折线图，如果不考虑区分国家，每个年份都对应 142 个（国家的）预期寿命值：

```
load("datas/gapminder.rda")
gapminder
```

```
## # A tibble: 1,704 x 6
##    country     continent  year lifeExp      pop gdpPercap
##    <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
## 1 Afghanistan Asia       1952    28.8  8425333      779.
## 2 Afghanistan Asia       1957    30.3  9240934      821.
## 3 Afghanistan Asia       1962    32.0 10267083      853.
## 4 Afghanistan Asia       1967    34.0 11537966      836.
## 5 Afghanistan Asia       1972    36.1 13079460      740.
## 6 Afghanistan Asia       1977    38.4 14880372      786.
## # ... with 1,698 more rows
```

```
ggplot(gapminder, aes(year, lifeExp)) +
  geom_line()
```

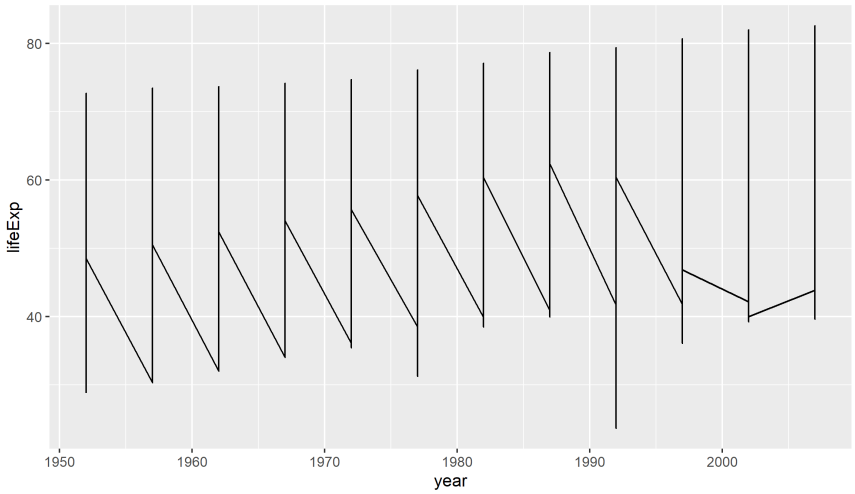

图 3.6: 多组数据的不分组折线图

这个图形显然不是我们想要的，应该区分不同国家，这就需要显式地映射分组美学：aes(group = country).

```
ggplot(gapminder, aes(year, lifeExp)) +
  geom_line(aes(group = country), alpha = 0.2) +
  geom_smooth(se = FALSE, size = 1.2)
```

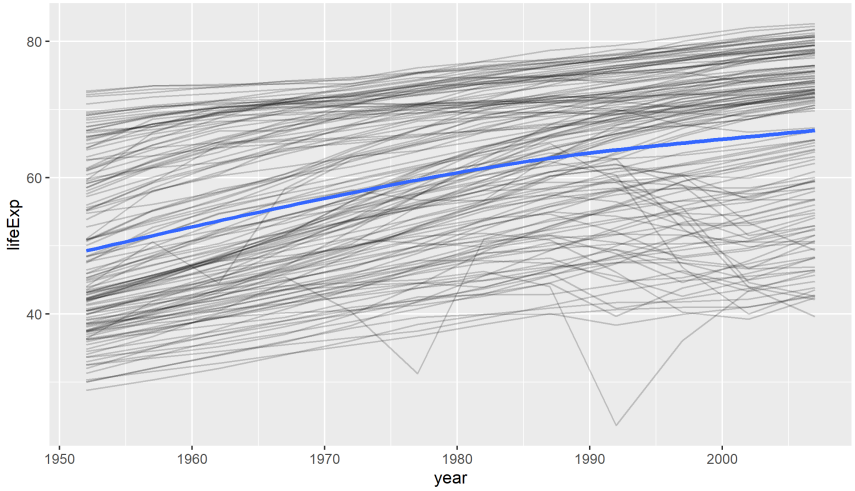

图 3.7: 多组数据的分组折线图

3.1.3 标度

通常 ggplot2 会自动根据输入变量选择最优的坐标刻度方案，若要手动设置或调整，就需要用到标度函数：`scale_<MAPPING>_<KIND>()`.

标度函数控制几何对象中的标度映射：不只是 `x`，`y` 轴，还有 `color`，`fill`，`shape`，`size` 产生的图例。它们是数据中的连续或分类变量的可视化表示，这需要关联到标度，所以要用到映射。

常用的标度函数有：

- `scale_*_continuous()`：`*` 为 `x` 或 `y`
- `scale_*_discrete()`：`*` 为 `x` 或 `y`
- `scale_x_date()`
- `scale_x_datetime()`
- `scale_*_log10()`，`scale_*_sqrt()`，`scale_*_reverse()`：`*` 为 `x` 或 `y`
- `scale_*_gradient()`，`scale_*_gradient2()`：`*` 为 `color`, `fill` 等

scales 包提供了很多现成的设置刻度标签风格的函数。

先通过来自 (Wickham 2020a) 的图3.8了解一下图例与坐标轴组件：

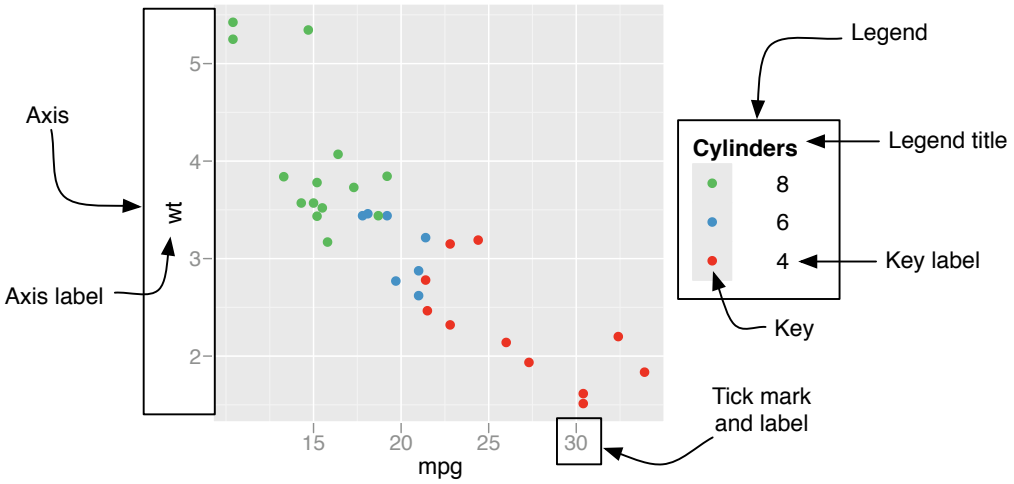

图 3.8: 图例与坐标轴的组件

1. 修改坐标轴刻度及标签

用 `scale_*_continuous()` 修改连续变量坐标轴的刻度和标签：

- 参数 `breaks` 设置各个刻度的位置
- 参数 `labels` 设置各个刻度对应的标签

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  scale_y_continuous(breaks = seq(15, 40, by = 10),
                     labels = c(" 一五"," 二五"," 三五"))
```

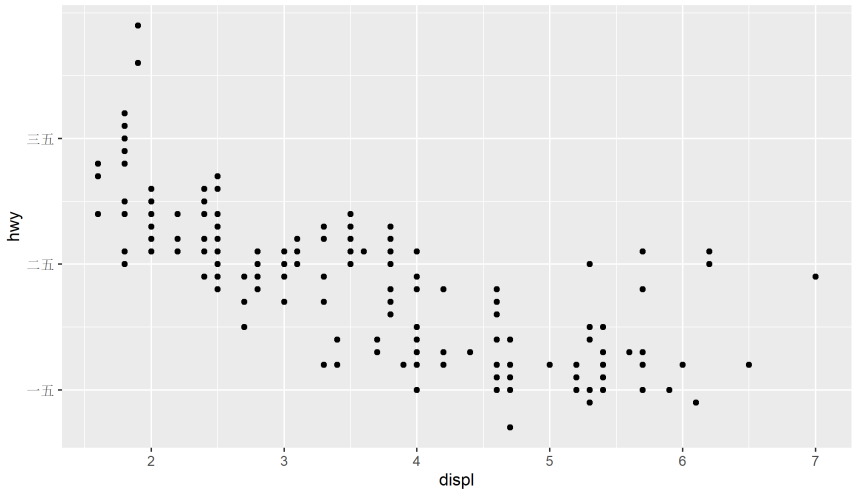

图 3.9: 手动设置坐标刻度和标签

用 `scale_*_discrete()` 修改离散变量坐标轴的标签：

```
ggplot(mpg, aes(x = drv)) +
  geom_bar() +      # 条形图
  scale_x_discrete(labels = c("4" = " 四驱", "f" = " 前驱", "r" = " 后驱"))
```

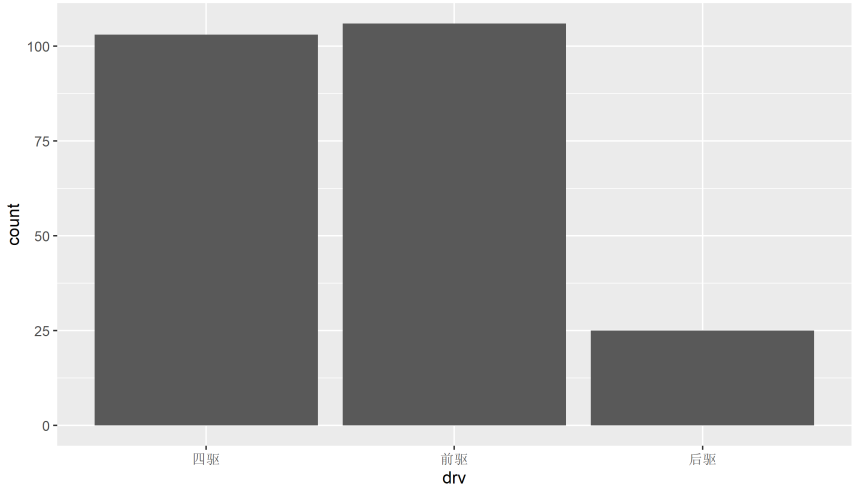

图 3.10: 手动设置离散变量坐标轴标签

用 scale_x_date() 设置日期刻度，参数 date_breaks 设置刻度间隔，date_labels 设置标签的日期格式；借助 scales 包中的函数设置特殊格式，比如百分数 (percent)、科学计数法 (scientific)、美元格式 (dollar) 等。

```
economics
```

```
## # A tibble: 574 x 6
##   date         pce    pop psavert uempmed unemploy
##   <date>     <dbl>  <dbl>   <dbl>   <dbl>    <dbl>
## 1 1967-07-01  507. 198712    12.6     4.5     2944
## 2 1967-08-01  510. 198911    12.6     4.7     2945
## 3 1967-09-01  516. 199113    11.9     4.6     2958
## 4 1967-10-01  512. 199311    12.9     4.9     3143
## 5 1967-11-01  517. 199498    12.8     4.7     3066
## 6 1967-12-01  525. 199657    11.8     4.8     3018
## # ... with 568 more rows
```

```
ggplot(tail(economics, 45), aes(date, uempmed / 100)) +
  geom_line() +
  scale_x_date(date_breaks = "6 months", date_labels = "%b%Y") +
  scale_y_continuous(labels = scales::percent)
```

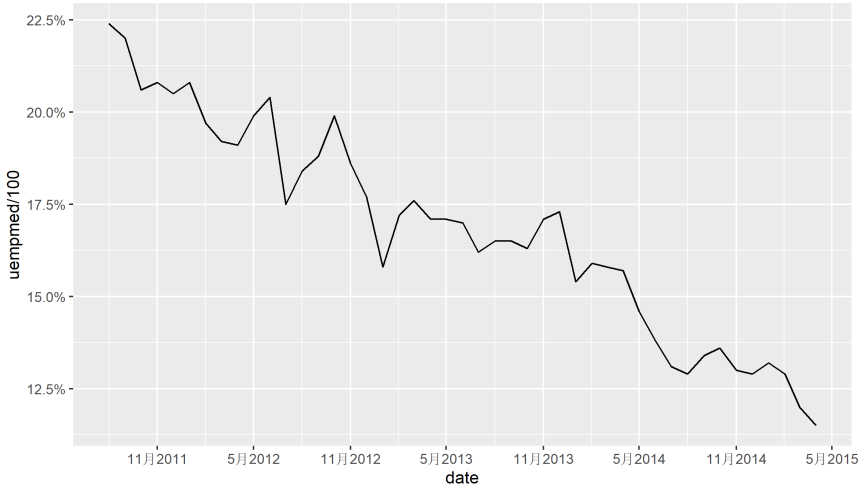

图 3.11: 用 scales 包设置坐标轴特殊格式标签

2. 修改坐标轴标签、图例名及图例位置

用 `labs()` 函数的参数 `x`，`y`，或者函数 `xlab()`，`ylab()`，设置 x 轴、y 轴标签，前面已使用 `color` 美学，则可以在 `labs()` 函数中使用参数 `color` 修改颜色的图例名。

图例位置是在 `theme` 图层通过参数 `legend.position` 设置，可选取值有 “none,” “left,” “right,” “bottom,” “top.”

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = drv)) +
  labs(x = " 引擎大小 (L)", y = " 高速燃油率 (mpg)", color = " 驱动类型") + # 或者
  # xlab(" 引擎大小 (L)") + ylab(" 高速燃油率 (mpg)")
  theme(legend.position = "top")
```

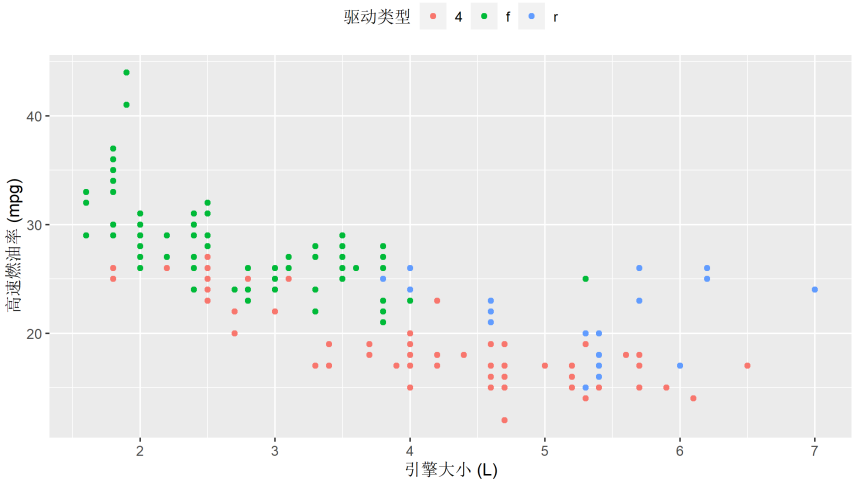

图 3.12: 设置坐标轴标签、图例位置及标签

3. 设置坐标轴范围

用 `coord_cartesian()` 函数的参数 `xlim` 和 `ylim`, 或者用 `xlim()`, `ylim()` 函数，设置 x 轴和 y 轴的范围：

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = drv)) +
  coord_cartesian(xlim = c(5, 7), ylim = c(10, 30))    # 或者
  # xlim(5, 7) + ylim(10, 30)
```

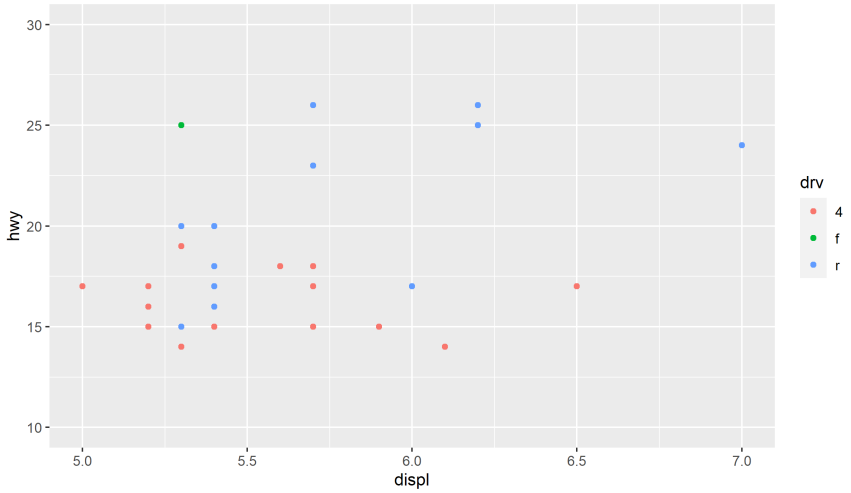

图 3.13: 设置坐标轴范围

4. 变换坐标轴

变换数据再绘图，比如对数变换，坐标刻度也会变成变换之后的，这使得图形不好理解。

ggplot2 提供的坐标变换函数 `scale_x_log10()` 等是变换坐标系，能够在视觉效果相同的情况下，使用原始数据的坐标刻度：

```
p = ggplot(gapminder, aes(gdpPercap, lifeExp)) +
  geom_point() +
  geom_smooth()
p + scale_x_continuous(labels = scales::dollar)
```

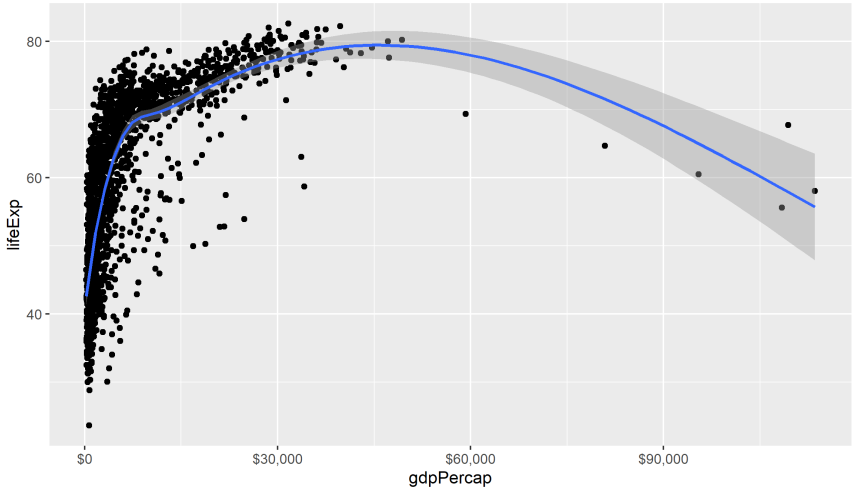

图 3.14: 原始坐标下的图形

```
p + scale_x_log10(labels = scales::dollar)
```

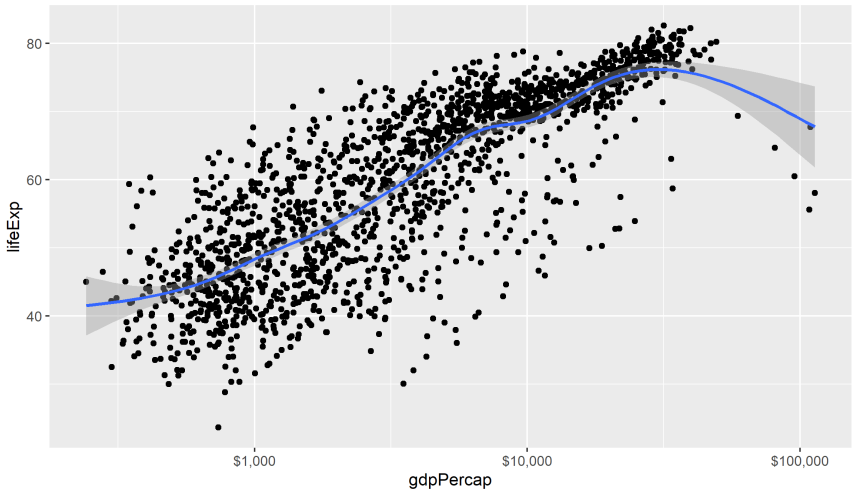

图 3.15: 坐标变换后的图形

5. 设置图形标题

用 `labs()` 函数的参数 `title`, `subtitle`, `caption` 设置标题、副标题、脚注标题（默认右下角）：

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = drv)) +
  geom_smooth(se = FALSE) +
  labs(title = " 燃油效率随引擎大小的变化图",
       subtitle = " 两座车 (跑车) 因重量小而符合预期",
       caption = " 数据来自 fueleconomy.gov")
```

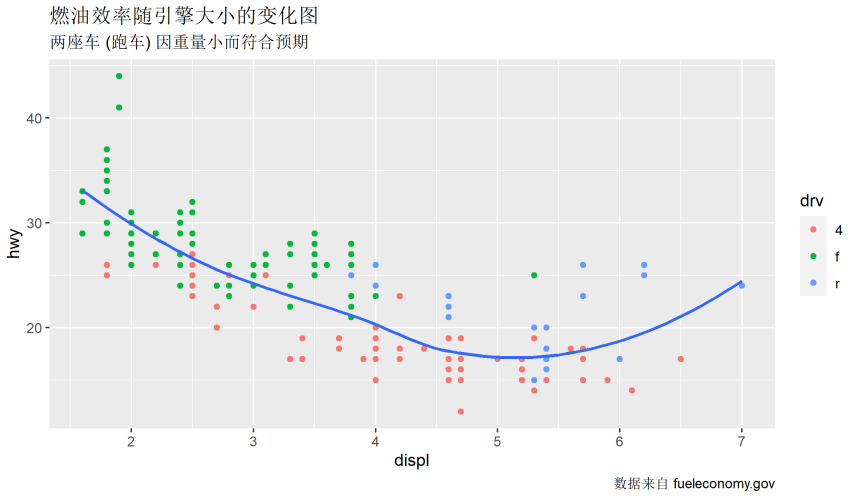

图 3.16: 设置图形标题

国外习惯图形标题位于顶部左端，如果想改成顶部居中，需要加 theme 图层专门设置：

```
+ theme(plot.title = element_text(hjust = 0.5),
        plot.subtitle = element_text(hjust = 0.5))    # 标题居中
```

6. 设置 fill，color 颜色

数据的某个维度信息可以通过颜色来展示，颜色直接影响图形的美感。可以直接使用颜色值，但是更建议使用 RColorBrewer（调色板）或 colorspace 包。

(1) 离散变量

- manual: 直接指定分组使用的颜色
- hue: 通过改变色相 (hue) 饱和度 (chroma) 亮度 (luminosity) 来调整颜色
- brewer: 使用 ColorBrewer 的颜色
- grey: 使用不同程度的灰色

用 scale_*_manual() 手动设置颜色，并修改图例及其标签：

```
ggplot(mpg, aes(displ, hwy, color = drv)) +
    geom_point() +
    scale_color_manual(" 驱动方式",          # 修改图例名
                       values = c("red", "blue", "green"),
                       # breaks = c("4", "f", "r"),
                       labels = c(" 四驱", " 前驱", " 后驱"))
```

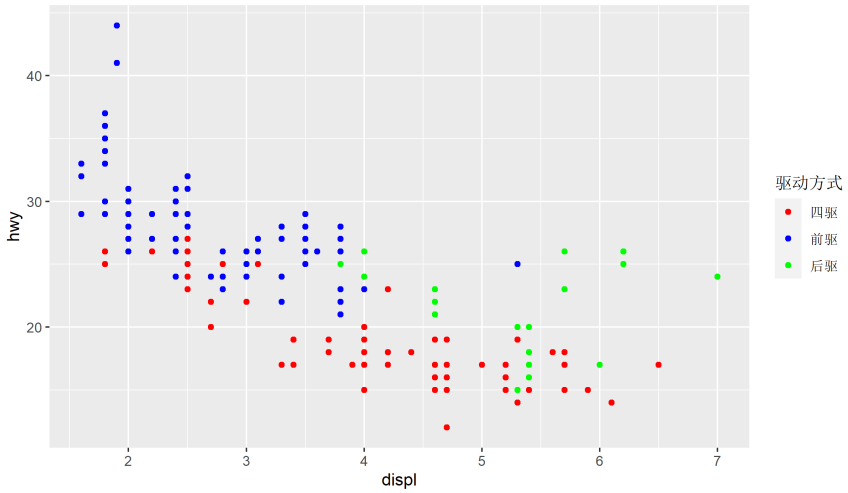

图 3.17: 手动设置离散变量颜色并修改对应图例

用 `scale_*_brewer()` 调用调色版中的颜色：

```
ggplot(mpg, aes(x = class, fill = class)) +
  geom_bar() +
  scale_fill_brewer(palette = "Dark2")    # 使用 Dark2 调色版
```

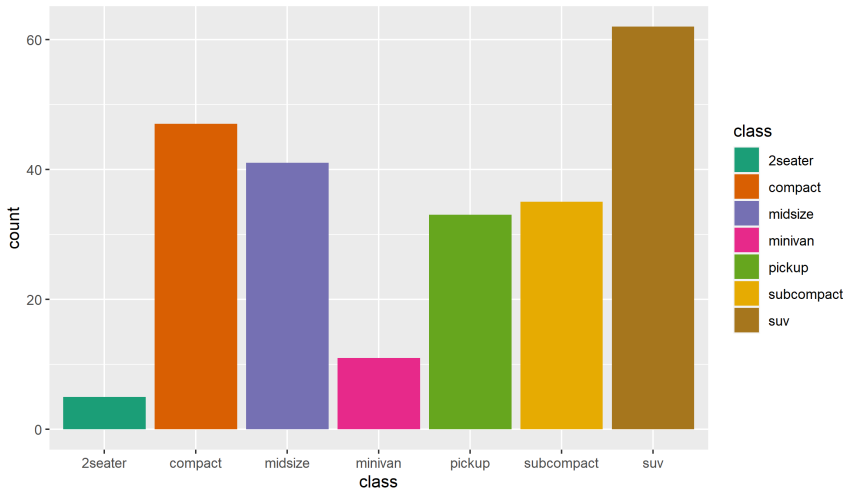

图 3.18: 使用调色板颜色设置离散变量颜色

查看所有可用的调色版：`RColorBrewer::display.brewer.all()`；查看所有可用的颜色空间：`hcl_palettes::hcl_palettes(plot = TRUE)`.

(2) 连续变量

- `gradient`: 设置二色渐变色
- `gradient2`: 设置三色渐变色
- `distiller`: 使用 ColorBrewer 的颜色
- `identity` 使用 color 变量对应的颜色，对离散型和连续型都有效

用 `scale_color_gradient()` 设置二色渐变色：

```
ggplot(mpg, aes(displ, hwy, color = hwy)) +
  geom_point() +
  scale_color_gradient(low = "green", high = "red")
```

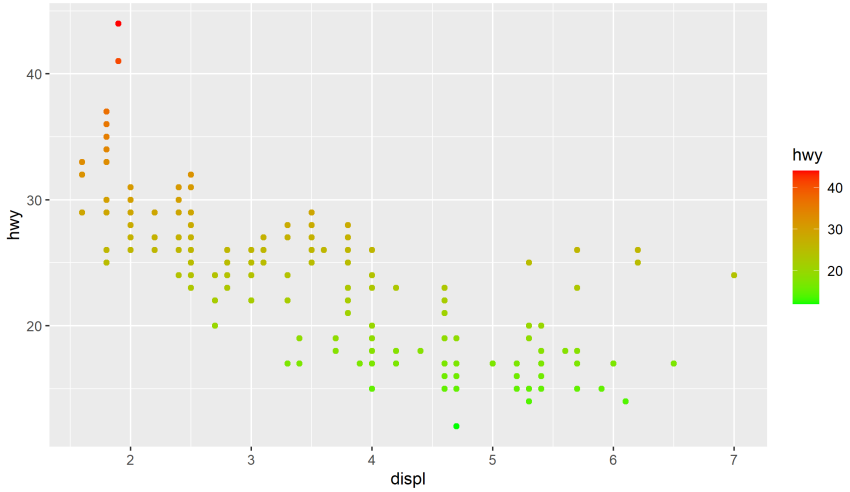

图 3.19: 使用渐变色设置连续变量颜色

用 `scale_*_distiller()` 调用调色版中的颜色：

```
ggplot(mpg, aes(displ, hwy, color = hwy)) +
  geom_point()  +
  scale_color_distiller(palette = "Set1")
```

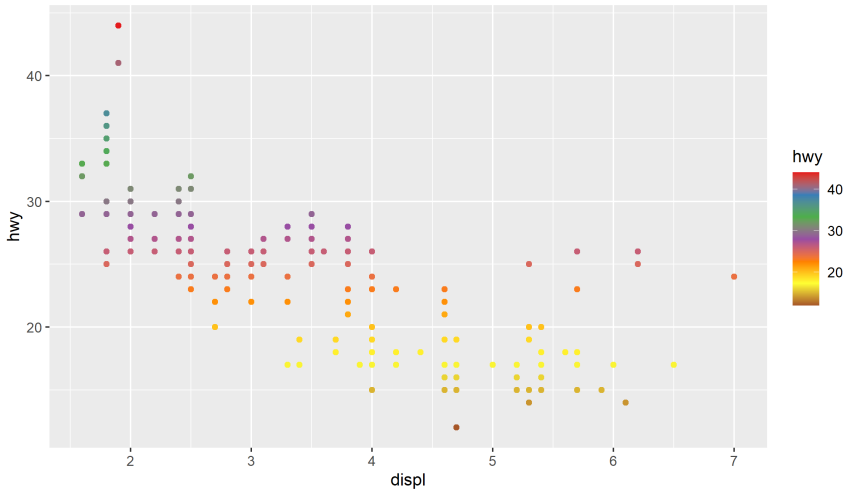

图 3.20: 使用调色板设置连续变量颜色

7. 添加文字标注

`ggrepel` 包提供了 `geom_label_repel()` 和 `geom_text_repel()` 函数，为图形添加文字标注。

首先要准备好标记点的数据，然后增加文字标注的图层，需要提供标记点数据，以及要标注的文字给 `label` 美学，若来自数据变量，则需要用映射。

```
library(ggrepel)
best_in_class = mpg %>%        # 选取每种车型 hwy 值最大的样本
  group_by(class) %>%
  slice_max(hwy, n = 1)
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_label_repel(data = best_in_class, aes(label = model))
```

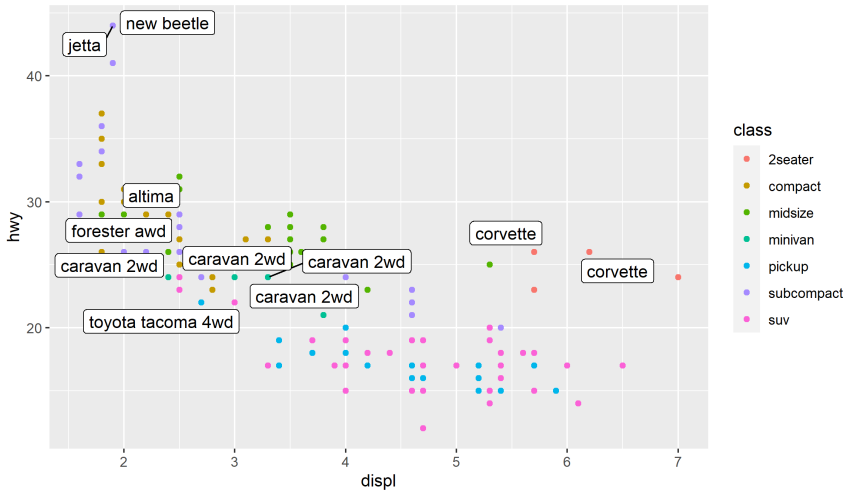

图 3.21: 为图形添加文字标注

若要在图形某坐标位置添加文本注释，则用 `annotate()` 函数，需要提供添加文本的中心坐标位置，和要添加的文字内容：

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  annotate(geom = "text", x = 6, y = 40,
           label = " 引擎越大\n燃油效率越高!", size = 4, color = "red")
```

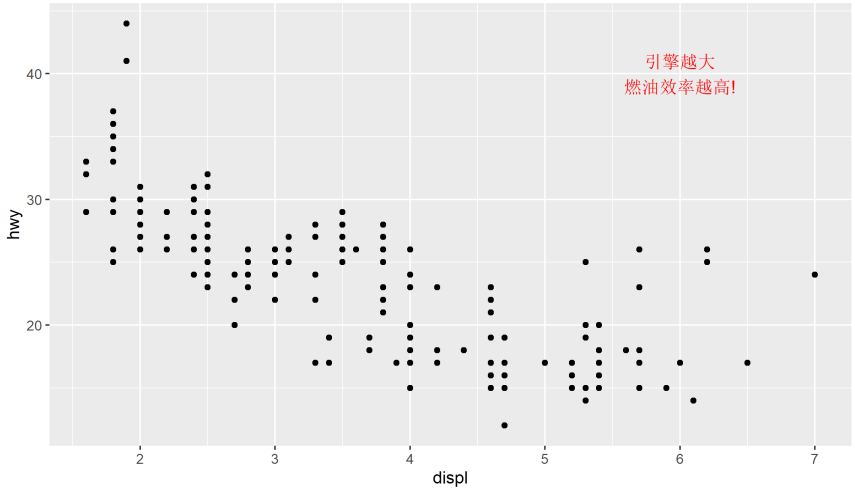

图 3.22: 为图形添加文字注释

3.1.4 统计变换、坐标系、位置调整

统计变换 (Statistics)

构建新的统计量进而绘图，称为“统计变换”，简称“统计”。比如，条形图、直方图都是先对数据分组，再计算分组频数（落在每组的样本点数）绘图；箱线图计算稳健的分布汇总，并用特殊盒子展示出来；平滑曲线用来根据数据拟合模型，进而绘制模型预测值……

ggplot2 强大的一点就是，把统计变换直接融入绘图语法中，而不必先在外面对数据做统计变换，再回来绘图。

ggplot2 中的提供了 30 多种“统计”，均以 stat_xxxx() 的方式命名。可以分为两类：

- 可以在几何对象函数 geom_*() 中创建，通常直接使用后者即可：
  - stat_bin(): geom_bar(), geom_freqpoly(), geom_histogram()
  - stat_bindot(): geom_dotplot()
  - stat_boxplot(): geom_boxplot()
  - stat_contour(): geom_contour()
  - stat_quantile(): geom_quantile()
  - stat_smooth(): geom_smooth()
  - stat_sum(): geom_count()
- 不能在几何对象函数 geom_*() 中创建：
  - stat_ecdf(): 计算经验累积分布图
  - stat_function(): 根据 x 值的函数计算 y 值
  - stat_summary(): 在 x 唯一值处汇总 y 值
  - stat_qq(): 执行 Q-Q 图计算
  - stat_spoke(): 转换极坐标的角度和半径为直角坐标位置
  - stat_unique(): 剔除重复行

用 stat_summary() 做统计汇总并绘图。通过传递函数做统计计算，首先注意 x 和 y 美学映射到

calss 和 hwy; fun = mean 是根据 x 计算 y，故对每个车型计算一个平均的 hwy；fun.max，fun.min 同样根据 x 分别计算 y 的均值加减标准差；统计计算的结果将传递给几何对象参数 geom 用于绘图：

```
ggplot(mpg, aes(x = class, y = hwy)) +
  geom_violin(trim = FALSE, alpha = 0.5, color = "green") +  # 小提琴图
  stat_summary(fun = mean,
               fun.min = function(x) {mean(x) - sd(x)},
               fun.max = function(x) {mean(x) + sd(x)},
               geom = "pointrange", color = "red")
```

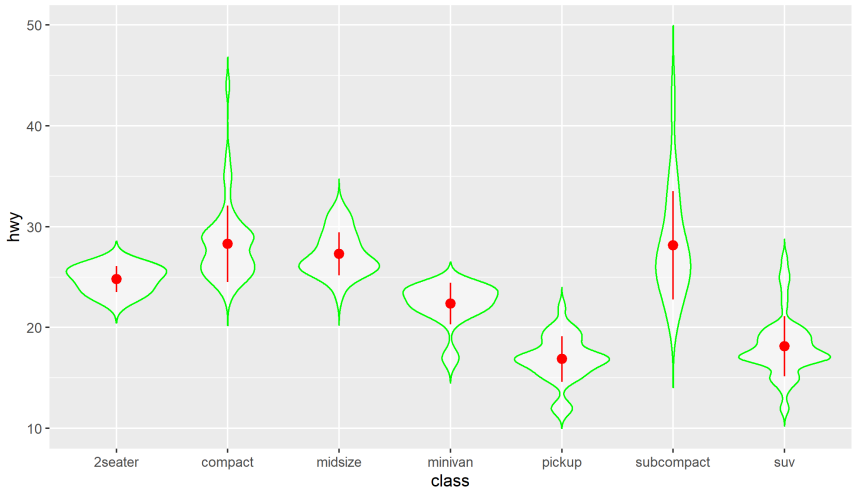

图 3.23: 标注均值和标准差的小提琴图

用 stat_smooth(), 与 geom_smooth() 相同, 添加光滑曲线：

- method: 指定平滑曲线的统计函数，如 lm 线性回归, glm 广义线性回归, loess 多项式回归, gam 广义加法模型 (mgcv 包) , rlm 稳健回归 (MASS 包) 等
- formula: 指定平滑曲线的方程，如 y ~ x, y ~ poly(x, 2), y ~ log(x) ，需要与 method 参数搭配使用
- se: 设置是否绘制置信区间

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  stat_smooth(method = "lm",
              formula = y ~ splines::bs(x, 3),
              se = FALSE)                  # 不绘制置信区间
```

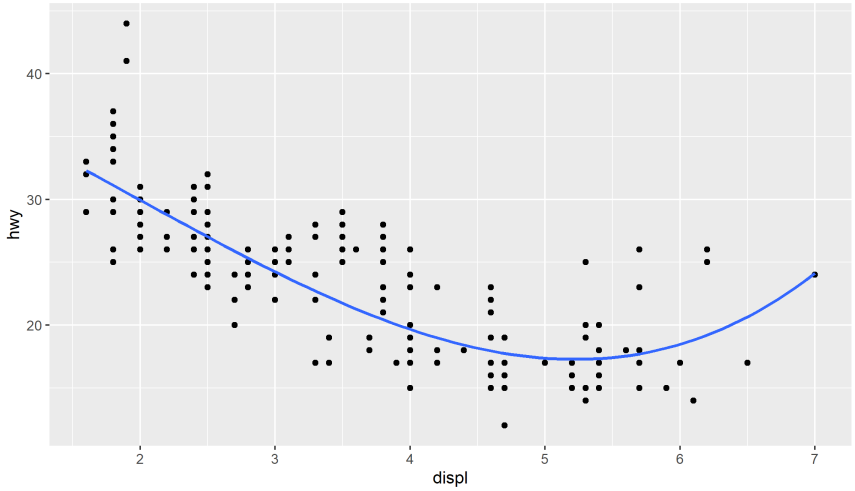

图 3.24: 三次样条光滑曲线

坐标系 (Coordinante)

ggplot2 默认坐标系是直角坐标系 coord_cartesian()，常用的坐标系操作还有：

- coord_flip()：坐标轴翻转，即 x 轴与 y 轴互换，比如绘制水平条形图
- coord_fixed()：固定 ratio = y / x 的比例
- coord_polar()：转化为极坐标系，比如条形图转为极坐标系即为饼图
- coord_trans()：彻底的坐标变换，不同于 scale_x_log10() 等
- coord_map(), coord_quickmap()：与 geom_polygon() 连用，控制地图的坐标投影
- coord_sf()：与 geom_sf() 连用，控制地图的坐标投影

坐标轴翻转，从水平图到竖直图：

```
ggplot(mpg, aes(class, hwy)) +
  geom_boxplot() +                  # 箱线图
  coord_flip()                      # 从竖直变成水平
```

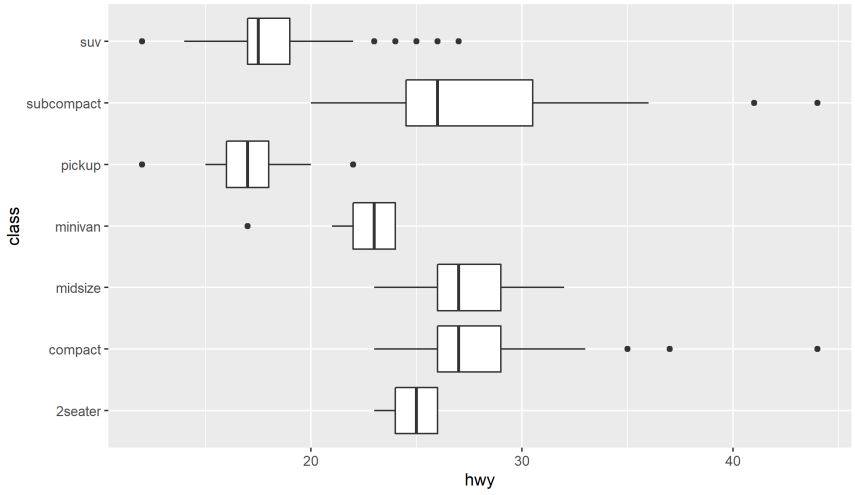

图 3.25: 水平箱线图

直角坐标下的条形图，转化为极坐标下的风玫瑰图：

```
ggplot(mpg, aes(class, fill = drv)) +
  geom_bar() +
  coord_polar()
```

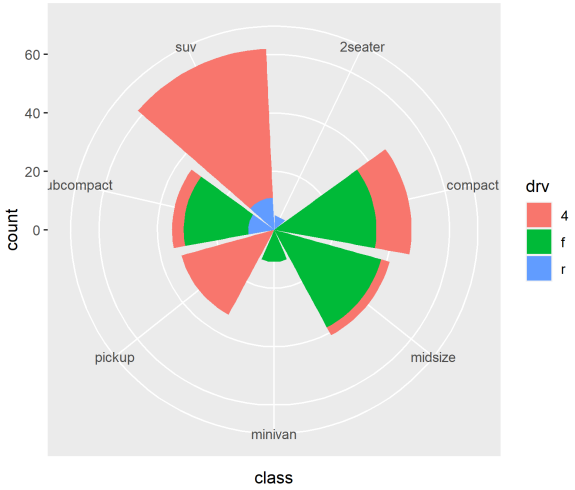

图 3.26: 风玫瑰图

位置调整 (Position adjustments)

条形图中的条形位置调整：

- `position_stack()`: 竖直堆叠
- `position_fill()`: 竖直 (百分比) 堆叠，按比例放缩保证总高度为 1
- `position_dodge()`, `position_dodge2()`: 水平堆叠

```
ggplot(mpg, aes(class, fill = drv)) +
  geom_bar(position = position_dodge(preserve = "single"))
  # geom_bar(position = "dodge")
```

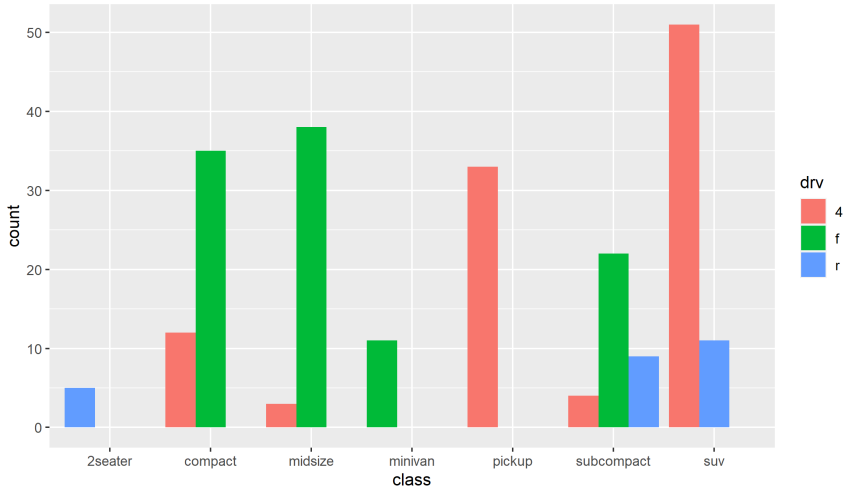

图 3.27: 堆叠条形图

散点图中的散点位置调整：

- `position_nudge()`: 将散点移动固定的偏移量
- `position_jitter()`: 给每个散点增加一点随机噪声 (抖散图)
- `position_jitterdodge()`: 增加一点随机噪声并躲避组内的点，特别用于箱线图 + 散点图

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(position = "jitter")    # 避免有散点重叠
```

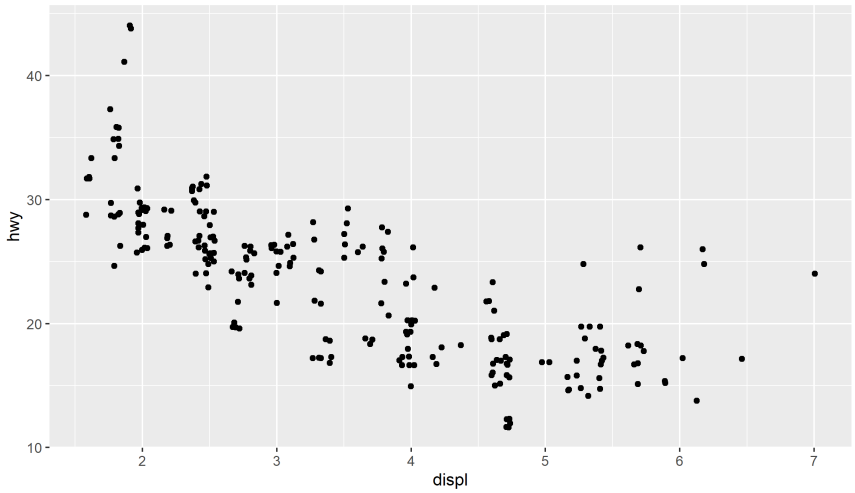

图 3.28: 抖散图

有时候需要将多个图形排布在画板中，借助 `patchwork` 包更方便。

```
library(patchwork)
p1 = ggplot(mpg, aes(displ, hwy)) +
  geom_point()
p2 = ggplot(mpg, aes(drv, displ)) +
  geom_boxplot()
p3 = ggplot(mpg, aes(drv)) +
  geom_bar()
p1 | (p2 / p3)
```

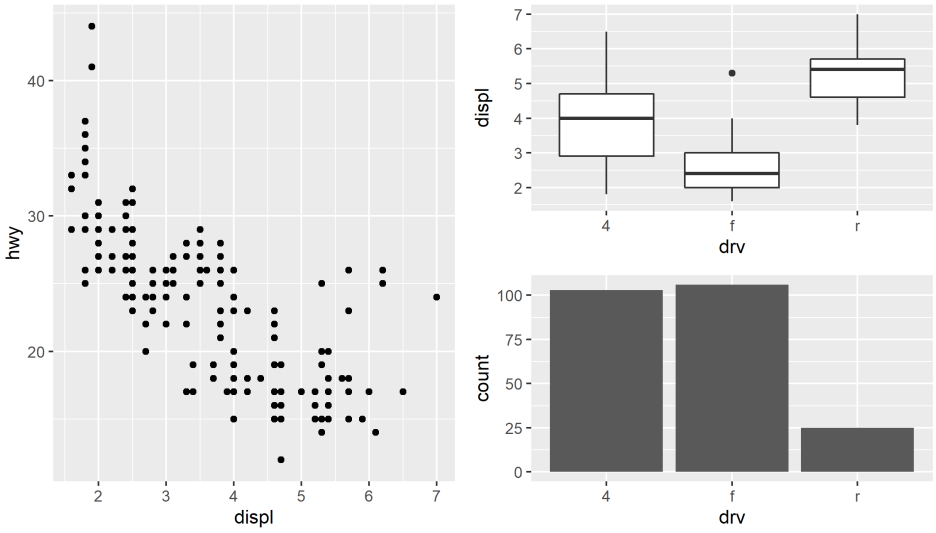

图 3.29: 图形排布

3.1.5 分面、主题、输出

分面 (Facet)

利用分类变量将图形分为若干个“面”（子图），即对数据分组再分别绘图，称为“分面”。

(1) `facet_wrap()`

封装分面，先生成一维的面板系列，再封装到二维中。

- 分面形式：`~ 分类变量`，`~ 分类变量 1 + 分类变量 2`
- `scales` 参数设置是否共用坐标刻度，`"fixed"`（默认，共用），`"free"`（不共用），也可以用 `free_x`，`free_y` 单独设置
- 参数 `nrow` 和 `ncol` 可设置子图的放置方式

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  facet_wrap(~ drv, scales = "free")
```

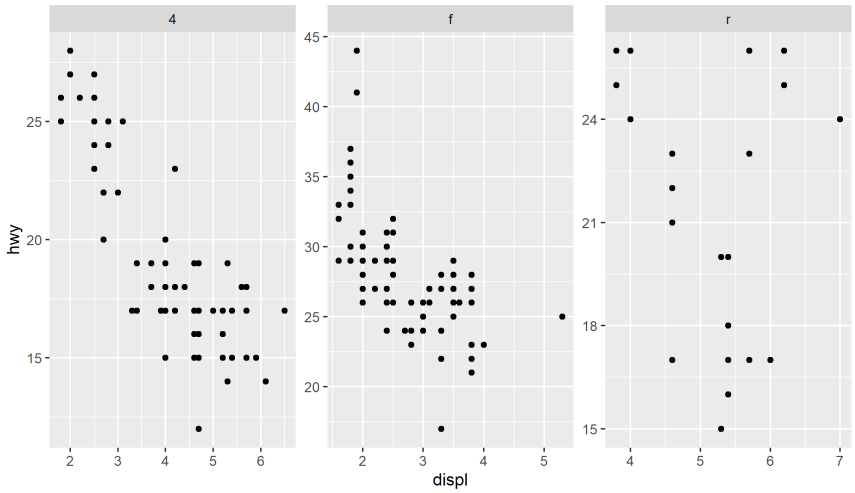

图 3.30: 一个分类变量分面

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  facet_wrap(~ drv + cyl)
```

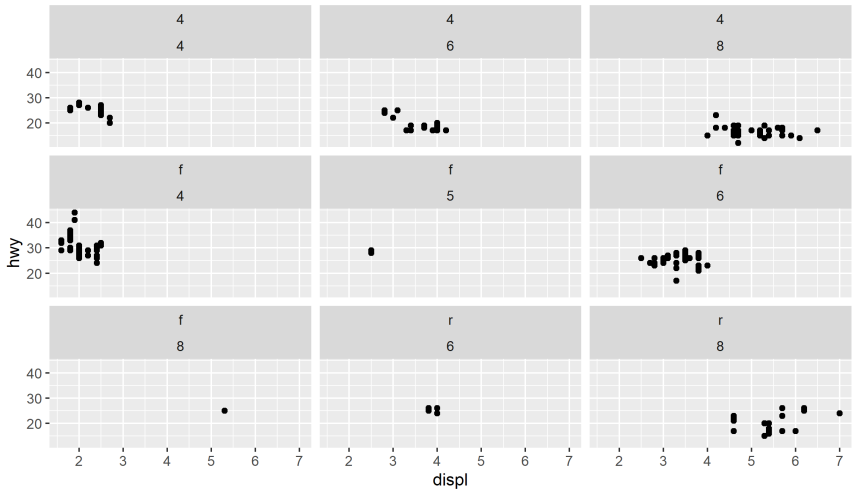

图 3.31: 两个分类变量分面

(2) facet_grid()

网格分面，生成二维的面板网格，面板的行与列通过分面变量定义。

- 分面形式：行分类变量 ~ 列分类变量

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  facet_grid(drv ~ cyl)
```

主题 (theme)

你可以为图形选择不同风格的主题（外观），ggplot2 提供了 8 套可选主题：

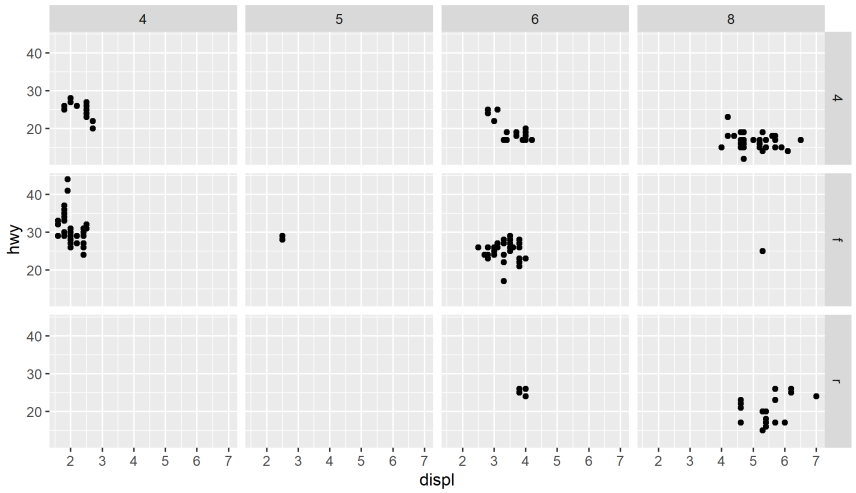

图 3.32: 网格分面

- theme_bw()
- theme_light()
- theme_classic()
- theme_gray(): 默认
- theme_linedraw()
- theme_dark()
- theme_minimal()
- theme_void()

使用或修改主题，只需要添加主题图层：

```
ggplot(mpg, aes(displ, hwy, color = drv)) +
  geom_point() +
  theme_bw()
```

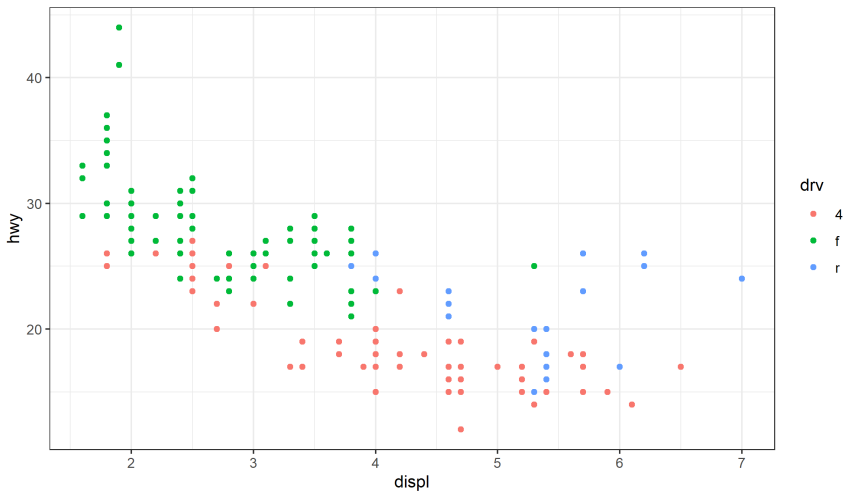

图 3.33: 选择主题

更多的主题，还可以用 ggthemes 包，其中包含一些顶级期刊专用绘图主题；当然也可以用 theme()

函数定制自己的主题（略）。

输出 (output)

用 ggsave() 函数，将当前图形保存为想要格式的图形文件，如 png，pdf 等：

```
ggsave("my_plot.pdf", width = 8, height = 6, dpi = 300)
```

参数 width 和 height 通常只设置其中一个，另一个自动，以保持原图形宽高比。

最后，再补充一点关于图形中使用中文字体导出到 pdf 等图形文件出现乱码问题的解决办法。

出现中文乱码是因为 R 环境只载入了 “sans (Arial),” “serif (Times New Roman),” “mono (Courier New)” 三种英文字体，没有中文字体可用。

解决办法就是从系统字体中载入中文字体，用 showtext 包 (依赖 sysfonts 包) 更简单一些。

- font_paths(): 查看系统字体路径，windows 默认是 C:\Windows\Fonts
- font_files(): 查看系统自带的所有字体文件
- font_add(): 从系统字体中载入字体，需提供 family 名字，字体路径

载入字体后，再执行一下 showtext_auto() (启用/关闭功能), 就可以使用该字体了。

ggpplot2 中各种设置主题、文本相关的函数 *_text()，annotate() 等，都提供了 family 参数，设定为 font_add() 中一致的 family 名字即可。

```
library(showtext)
font_add("heiti", "simhei.ttf")
font_add("kaiti", "simkai.ttf")
showtext_auto()
ggplot(mpg, aes(displ, hwy, color = drv)) +
  geom_point() +
  theme(axis.title = element_text(family = "heiti"),
        plot.title = element_text(family = "kaiti")) +
  xlab(" 发动机排量 (L)") +
  ylab(" 高速里程数 (mpg)") +
  ggtitle(" 汽车发动机排量与高速里程数") +
  annotate("text", 5, 35, family = "kaiti", size = 8,
           label = " 设置中文字体", color = "red")
ggsave("images/font_example.pdf", width = 7, height = 4)
```

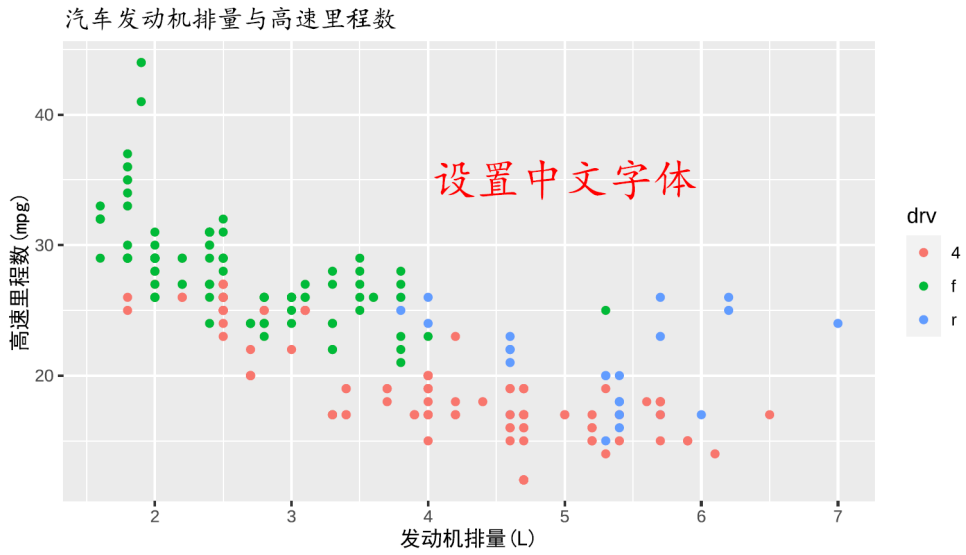

图 3.34: 解决中文字体导出到 pdf 乱码

本节部分内容参阅 (Wickham 2020a), (G. G. Hadley Wickham 2017), (王敏杰 2020), Kieran Healy: A Practical Introduction to Data Visualization with ggplot2, 西游东行: ggplot2| 详解八大基本绘图要素, The R Graph Gallery.

3.2 ggplot2 图形示例

俗话说，‘‘一图胜千言,” 数据可视化能够:

- 真实、准确、全面地展示数据信息
- 发现数据中隐含的关系和模式

Nathan Yau 将数据可视化的过程总结为如下的 4 个思索:

- 你拥有什么样的数据?
- 你想要表达什么样的数据信息?
- 你会什么样的数据可视化方法?
- 你从图表中能获得什么样的数据信息?

上述思索需要您对数据可视化的图形种类有所了解，本节借用 (张杰 2019) 的类别划分：类别比较图、数据关系图、数据分布图、时间序列图、局部整体图、地理空间图。下面将对各类图形进行概述，选择其中常用的、代表性图形，进行实例展示，还有一些常用统计图、探索变量间关系的图，将在后续第 4 章和第 5 章中展示。

另外，`ggpubr` 包提供了很多函数，轻松绘制适合用于期刊论文发表的图形。更多图形绘制参阅 (张杰 2019), (赵鹏, 谢益辉, and 黄湘云 2021), (Chang 2018), R Graph Gallery, from Data to Viz 等。

读者针对自己的数据绘图时，建议首先根据展示目的在本节的类别划分图中，选择想要绘制的图形，再查阅相关资料，完成相应图形的绘制。

3.2.1 类别比较图

类别比较图，通常是展示和比较分类变量或分类变量组合的频数。采用表示位置和长度的视觉元素的不同，产生了多种类别比较图：

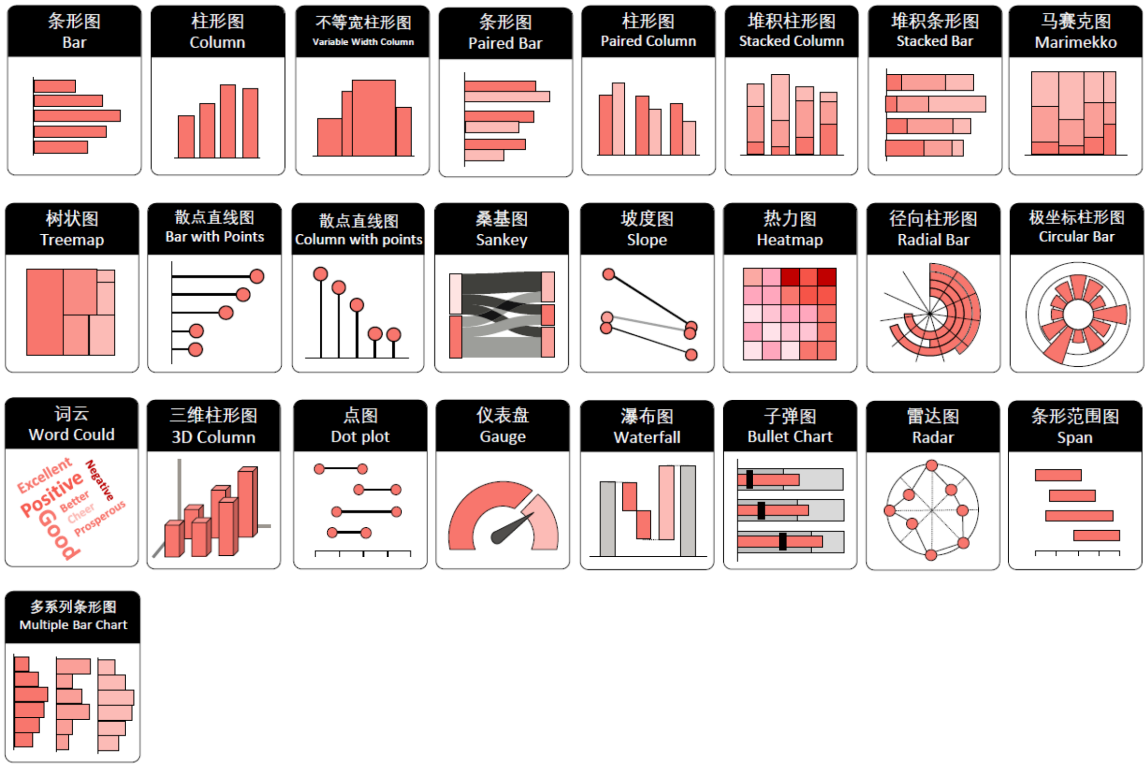

图 3.35: 类别比较图

热图

两个分类变量的交叉频数，可以用热图展示，两分类各水平确定交叉网格，其上的频数 (或其他数值) 对应到颜色深度。邻接矩阵、混淆矩阵、相关系数矩阵也可以用热图来可视化展示。

对 mpg 按车型和驱动方式统计频数，并绘制热图，需要注意别漏下 0 频数：

```
df = mpg %>%
  mutate(across(c(class, drv), as.factor)) %>%
  count(class, drv, .drop = FALSE)
df
df %>%
  ggplot(aes(class, drv)) +
  geom_tile(aes(fill = n)) +
  geom_text(aes(label = n)) +
  scale_fill_gradient(low = "white", high = "darkred")
```

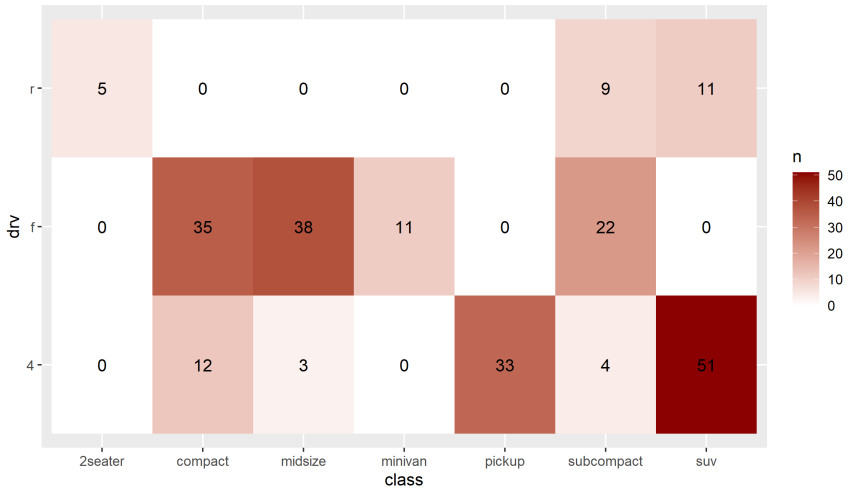

图 3.36: 热图

ComplexHeatmap 包可绘制更复杂的热图：带层次聚类的热图。

3.2.2 数据关系图

数据关系图，主要包括

- 数据相关性图：展示两个或多个变量之间的关系，比如散点图、气泡图、曲面图等
- 数据流向图：展示两个或多个状态或情形之间的流动量或关系强度，比如网络图等

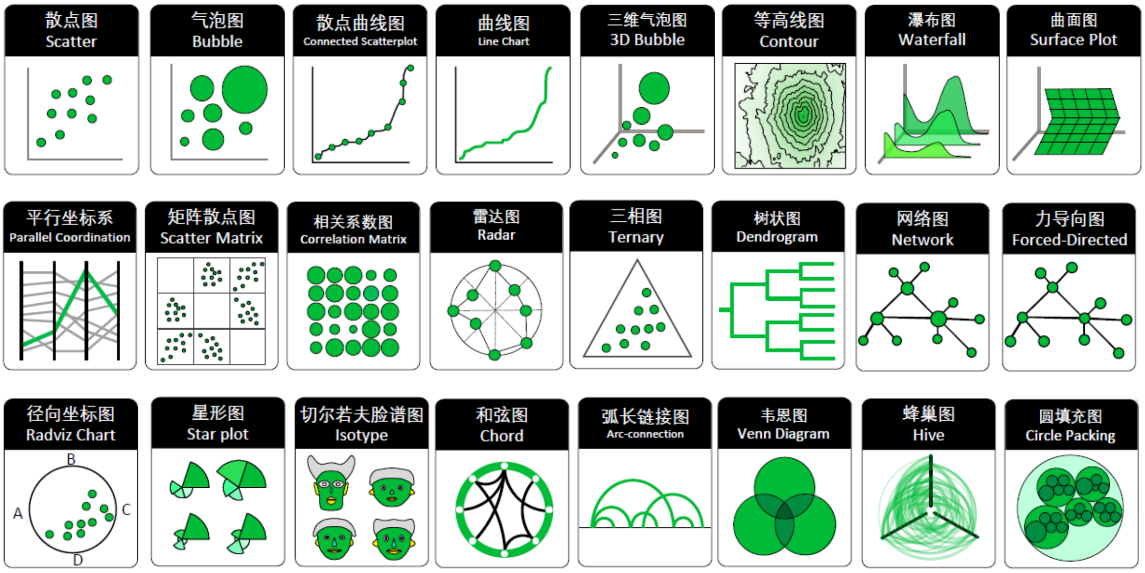

图 3.37: 数据关系图

网络图

网络图可以可视化实体 (个体/事物) 间的内部关系，比如社会媒体网络、朋友网络、合作网络、疾病传播网络等。

可视化网络图的包有：`igraph`, `tidygraph + ggraph`. 还有更加强大的 `visNetwork` 包，这里只

给一个简单示例：国家之间电话数据。用 `visNetwork()` 函数实现，需要准备好结点和边的数据。

- 结点数据，包括 `id`（用于边数据），`label`（用于图显示），`group`（设置分组颜色），`value`（权重，关联结点的大小）等
- 边数据，包括 `from`（起点），`to`（终点），`label`（用于图显示），`value`（权重，关联边的粗细）等

```
load("datas/phone_call.rda")
nodes
```

```
## # A tibble: 16 x 3
##   id       label    value
##   <chr>    <chr>    <dbl>
## 1 France   France      25
## 2 Belgium  Belgium      7
## 3 Germany  Germany     28
## 4 Danemark Danemark     2
## 5 Croatia  Croatia      6
## 6 Slovenia Slovenia     2
## # ... with 10 more rows
```

```
edges
```

```
## # A tibble: 18 x 3
##   from    to          value
##   <chr>   <chr>       <dbl>
## 1 France  Germany         9
## 2 Belgium France          4
## 3 France  Spain           3
## 4 France  Italy           4
## 5 France  Netherlands     2
## 6 France  UK              3
## # ... with 12 more rows
```

```
library(visNetwork)
visNetwork(nodes, edges)
```

3.2.3 数据分布图

数据分布图主要展示数据中数值出现的频率或分布规律，比如直方图、概率密度图、箱线图等。

人口金字塔图

人口金字塔图是展示人口数量或百分比 (x 轴) 随年龄段 (y 轴) 和性别（左右两侧）的分布的图形，可以方便地了解人口的构成以及当前人口增长的趋势：

图 3.38: 网络关系图

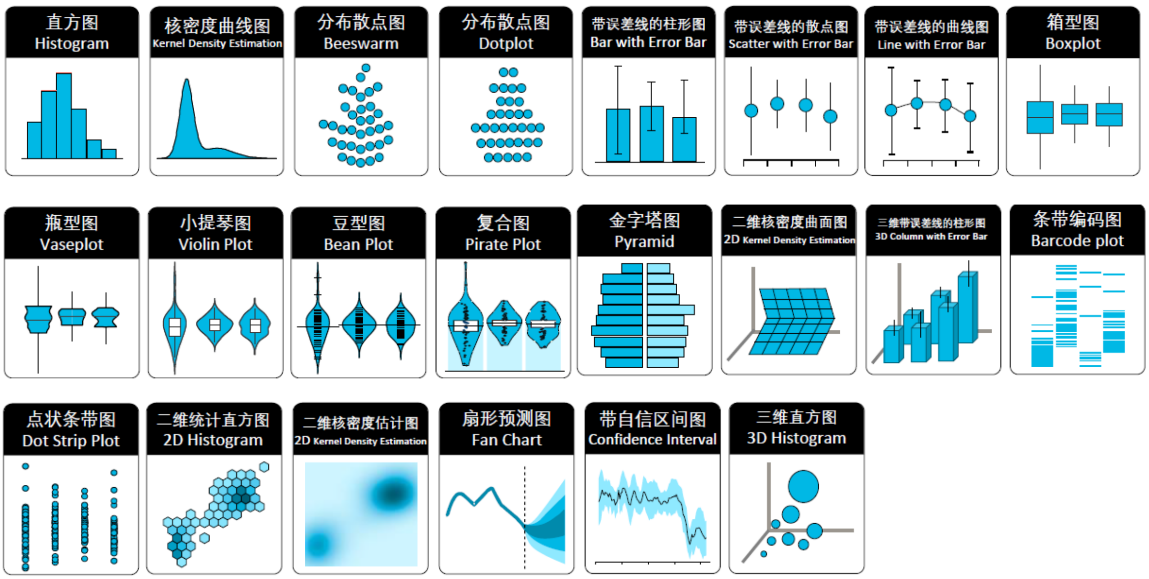

图 3.39: 数据分布图

- 若呈长方形，表明人口增长速度较慢；老一代人正被规模大致相同的新一代人所取代
- 若呈金字塔形，则表明人口以较快的速度增长；老一代人正在产生更大的新一代人

以黑龙江省 2019 年人口数据为例：

```
pops = read_csv("datas/hljPops.csv") %>%
  mutate(Age = as_factor(Age)) %>%
  pivot_longer(-Age, names_to = " 性别", values_to = "Pops")   # 宽变长
pops
ggplot(pops, aes(x = Age, fill = 性别,
                 y = ifelse(性别 == " 男", -Pops, Pops))) +
  geom_bar(stat = "identity") +
  scale_y_continuous(labels = abs, limits = c(-200,200)) +
  xlab(" 年龄段") + ylab(" 人口数 (万)") +
  coord_flip()
```

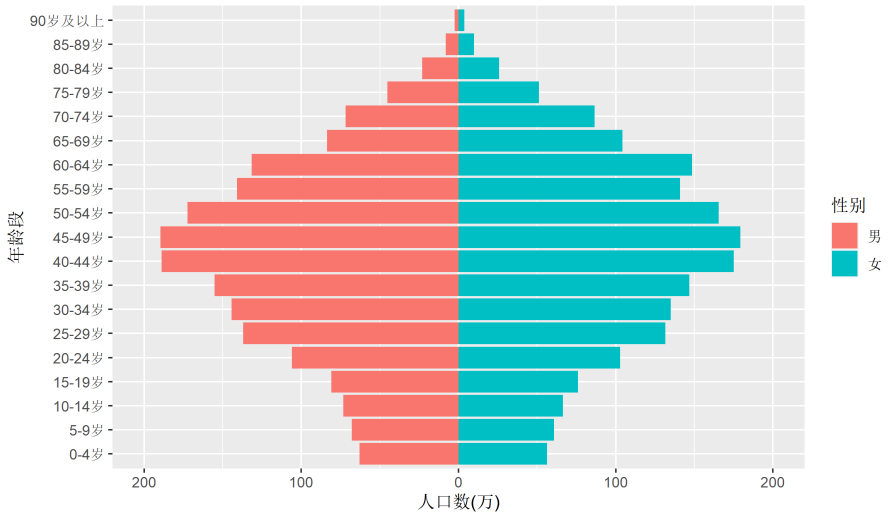

图 3.40: 人口金字塔图

3.2.4 时间序列图

时间序列图，展示数据随时间的变化规律或者趋势。比如，括折线图、面积图等。

折线图与面积图

折线图是按 x 从小到大对数据排序，再用直线依次连接各个散点，用 `geom_line()` 绘制。类似的 `geom_path()` 绘制路径图，不按 x 排序，按数据原始顺序用直线依次连接各个散点。

面积图是折线图下方再做填充，用 `geom_area()` 绘制。

```
p1 = ggplot(economics, aes(date, uempmed)) +
  geom_line(color = "red")
p2 = ggplot(economics, aes(date, uempmed)) +
```

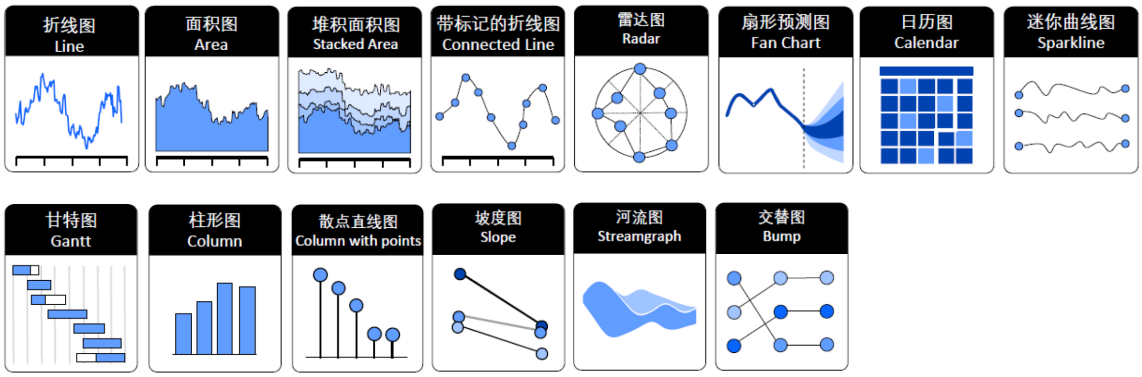

图 3.41: 时间序列图

```
  geom_area(color = "red", fill = "steelblue")
p1 | p2
```

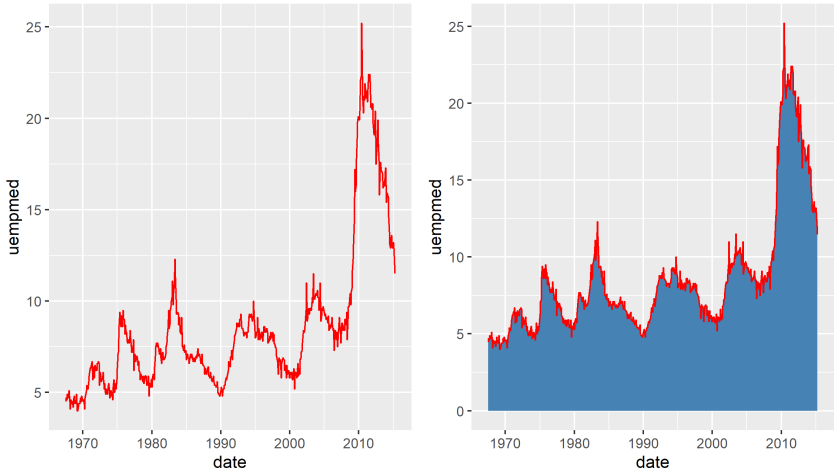

图 3.42: 折线图与面积图

3.2.5 局部整体图

局部整体图，展示部分与整体的关系。比如，饼图、树状图等。

这里提供一个绘制饼图的模板：

```
piedat = mpg %>%                # 先准备绘制饼图的数据
  group_by(class) %>%
  summarize(n = n(), labels = str_c(round(100 * n / nrow(.), 2), "%"))
piedat
ggplot(piedat, aes(x = "", y = n, fill = class)) +
  geom_bar(width = 1, stat = "identity") +
  coord_polar("y", start = 0) +
  geom_text(aes(label = labels),
            position = position_stack(vjust = 0.5)) +
```

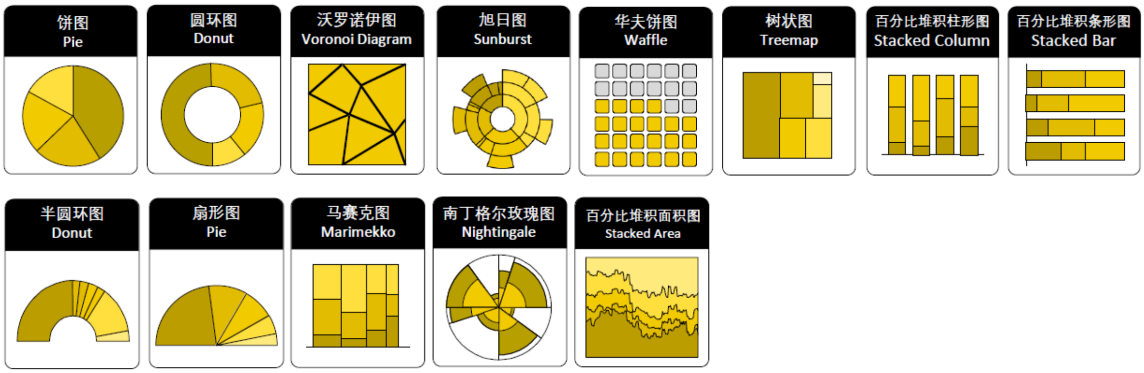

图 3.43: 局部整体图

```
theme_void()
```

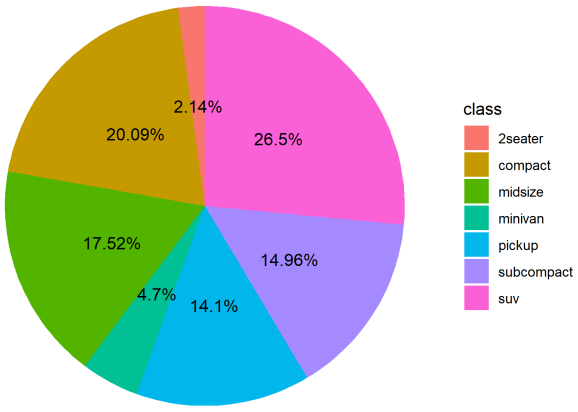

图 3.44: 饼图

3.2.6 地理空间图

地理空间图，是在地图上展示数据关系，即与地理位置信息联系起来绘图。地理位置通常是用经度、纬度表示。

现在的 GIS 类软件广泛使用简单要素 (Simple Features) 标准，主要是用二维几何图形对象（点、线、多边形、点族、线族、多边形族等）表示地理矢量数据，还可以包含坐标参照系统和用于描述对象的属性（如名称、值、颜色等）。

`sf` 包实现了将简单要素表示为 R 中的 `data.frame`, 以及一系列处理此类数据的工具。`sf` 格式数据框中，属性要素是正常的列，几何要素 (geometry) 存放为列表列。

geometry 列是最重要的列，它指定了每个地理区域的空间几何，每个元素都是一个多边形族，即包含一个或多个多边形的顶点的数据，它们能确定多边形区域的边界。

有了 `sf` 格式的数据，就可以用 `geom_sf()` 和 `coord_sf()` 绘制地图，甚至无需设定任何参数和美学映射。

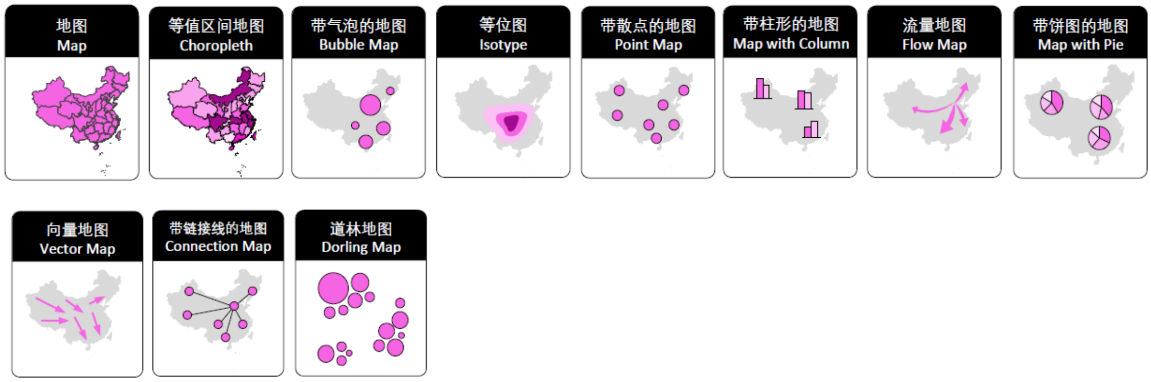

图 3.45: 地理空间图

以绘制中国地图为例，sf 格式的中国地图数据来自 mapchina 包中的 china 数据集。该数据为县级地图数据 (若想使用市级、省级数据，只需要按 Name_Perfecture / Name_Province 向上合并 geometry 列即可)，但缺少南海诸岛、南海九段线。为此，我从民政部网站下载并提取这部分地图数据，合并进 china 数据集，保存为 China_map_all.rda，只要载入就可以使用。

下面绘制 2019 年各省 GDP 地图，首先要准备各省 GDP 数据：

```
gdp = read.csv("datas/2019 分省 GDP.csv") %>%
  mutate(GDP = round(GDP / 10000, 2),
         GDPrank = cut(GDP, c(0,3,6,12), c(" 低"," 中"," 高"))) %>%
  as_tibble()
gdp
```

加载 sf 格式的中国地图数据，因为是县级地图数据，要绘制省级地图，需要用函数 sf::st_union() 按省份名 Name_Province 向上合并 geometry，得到省级地图数据；然后将 GDP 数据合并进来 (左连接)：

```
library(sf)
load("datas/China_map_all.rda")
sf_use_s2(FALSE)              # 临时解决：最新版本 sf，有个 Geometry 数据无效
sheng = chinamap %>%
  group_by(Name_Province) %>%
  summarise(geometry = sf::st_union(geometry))
sheng = left_join(sheng, gdp, by = c("Name_Province" = " 地区"))
sheng
```

先来针对连续变量 GDP 绘制地图，只需要将填充美学 fill 映射到连续变量 GDP，并按连续变量颜色设置语法设置颜色；用函数 geom_sf_label() 可以在各个省中心位置添加文字标注。

```
ggplot(sheng) +
  geom_sf(aes(fill = GDP)) +
  coord_sf() +
```

```
  scale_fill_gradient(low = "green", high = "red") +
  geom_sf_label(aes(label = GDP), size = 2.5) +
  labs(x = NULL, y = NULL)
```

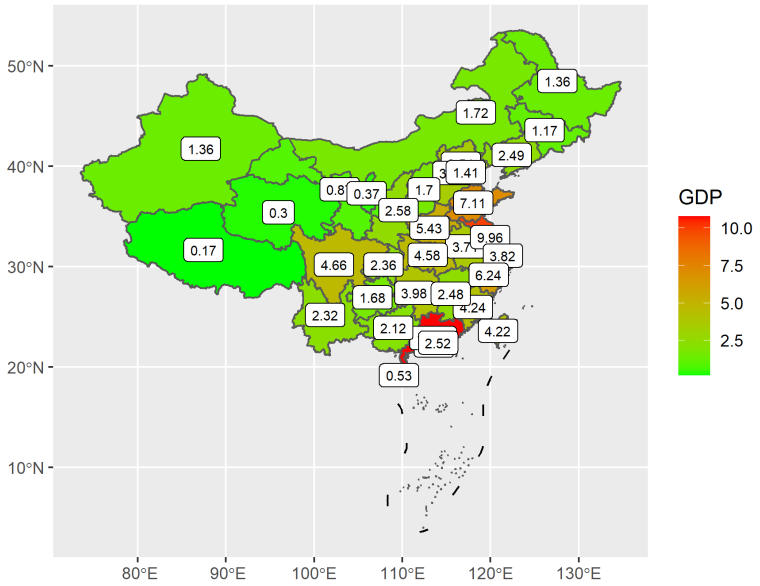

图 3.46: 中国地图可视化连续变量数据

再来针对离散变量 GDPrank 绘制地图，只需要将填充美学 fill 映射到离散变量 GDPrank, 并按离散变量颜色设置语法设置颜色。

这里多处理一个问题：南海诸岛、南海九段线的 GDP 和 GDPrank 均为 NA, 若不加处理直接绘图，图例中会多一类 NA. 解决办法是分别绘图，先对非缺失数据绘图，再单独绘制南海诸岛、南海九段线。

```
ggplot(na.omit(sheng)) +
  geom_sf(aes(fill = GDPrank)) +
  geom_sf(data = filter(sheng, str_detect(Name_Province, " 南海"))) +
  coord_sf() +
  scale_fill_brewer(palette = "Dark2") +
  labs(x = NULL, y = NULL, fill = "GDP 等级")
```

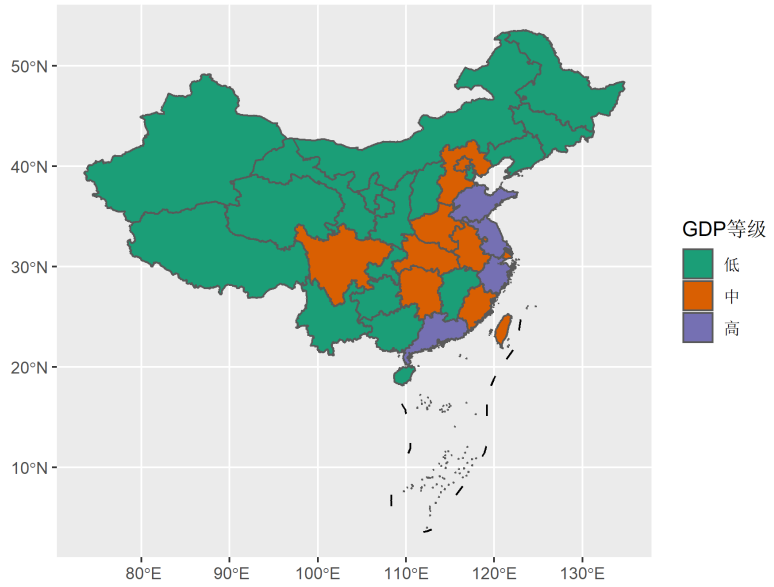

图 3.47: 中国地图可视化离散变量数据

如果想要将南海部分作为小窗绘制在右下角，可以先截取该部分数据单独绘图，并绘制矩形边框线，再对主体部分（不包含南海诸岛、南海九段线）绘图，最后借助 cowplot 包将两个图叠加放置即可（略）。

3.2.7 动态交互图

ggplotly 包能够在 ggplot2 的基础上生成动态可交互图形。只要对 ggplot2 绘制的图形对象套一个 ggplotly() 函数，则图形变成可交互状态：当鼠标移动到图形元素上时，将自动显示对应的数值。

```
library(plotly)
load("datas/gapminder.rda")
p = gapminder %>%
  filter(year == 2007) %>%
  ggplot(aes(gdpPercap, lifeExp, size = pop, color = continent)) +
  geom_point() +
  theme_bw()
ggplotly(p)
```

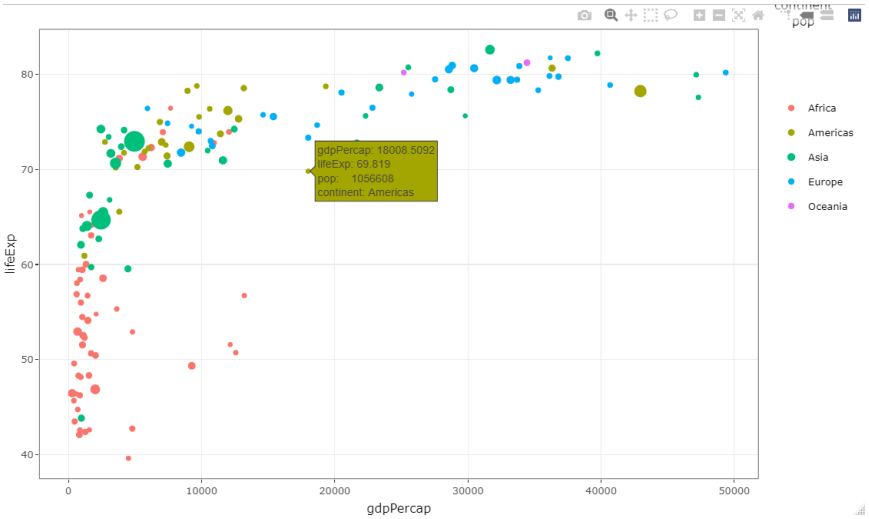

图 3.48: plotly 绘制动态可交互图形

gganimate 包是基于 ggplot2 的动态可视化拓展包，让图形元素随时间等逐帧变化起来，可导出为.gif 动图。

下面绘制一个动态散点图，反映不同国家在 1952-2007 年间，预期寿命与人均 GDP 的变化：

```
library(gganimate)
ggplot(gapminder, aes(gdpPercap, lifeExp, size = pop)) +
  geom_point() +
  geom_point(aes(color = continent)) +
  scale_x_log10() +
  labs(title = " 年份: {frame_time}", x = " 人均 GDP", y = " 预期寿命") +
  transition_time(year)
anim_save("output/gapminder.gif")      # 保存为 gif 文件
```

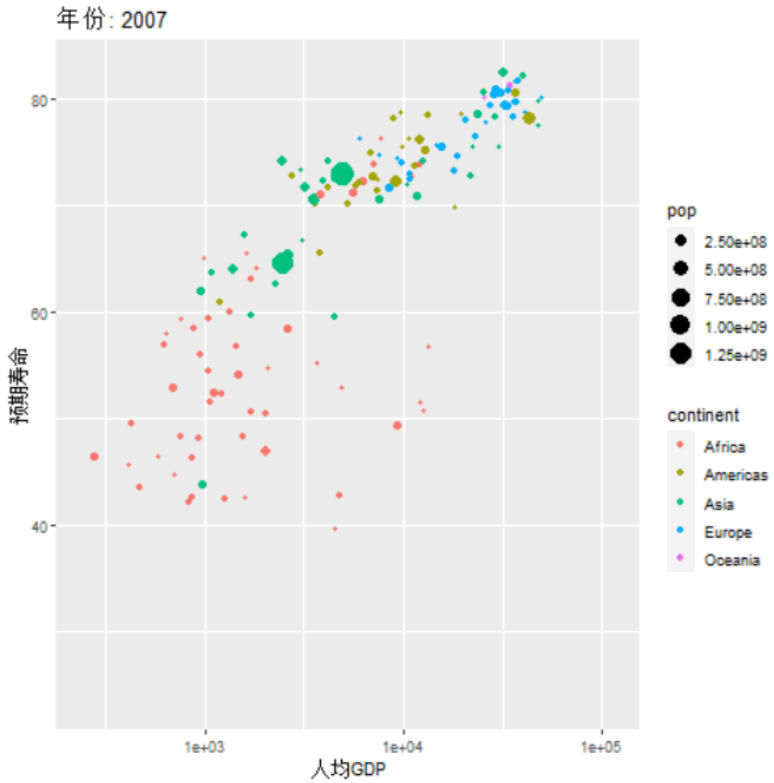

图 3.49: gganimate 绘制动态可视化图形

本节部分内容参阅 (张杰 2019), (Chang 2018), mapchina 包文档.

3.3 统计建模技术

3.3.1 整洁模型结果

`tidyverse` 主张以 ‘‘整洁的’’ 数据框作为输入，但是 `lm`, `nls`, `t.test`, `kmeans` 等模型的输出结果，却是 ‘‘不整洁的’’ 列表。

`broom` 包实现将模型输出结果转化为整洁的 `tibble`，且列名规范一致，方便后续取用；另外，与 `tidyr` 包中的 `nest()/unnest()` 函数以及 `purrr` 包中的 `map_*()` 系列函数连用，非常便于批量建模和批量整合模型结果。

`broom` 包主要提供了如下 3 个函数：

`tidy()`: 模型系数估计及其统计量

返回结果 `tibble` 的每一行通常表达的都是具有明确含义的概念，如回归模型的一项，一个统计检验，一个聚类或类；各列包括：

- `term`: 回归或模型中要估计的项
- `estimate`: 参数估计值
- `statistic`: 检验统计量
- `p.value`: 检验统计量的 p 值

- conf.low, conf.high: estimate 的置信区间界
- df: 自由度

glance(): 模型诊断信息

返回一行的 tibble, 各列是模型诊断信息:

- r.squared: R^2
- adj.r.suquared: 根据自由度修正的 R^2
- sigma: 残差标准差估计值
- AIC, BIC: 信息准则

augment(): 增加预测值列、残差列等

augment(model, data) 为原始数据增加新列，若 data 参数缺失，则只包含模型涉及的列。

返回结果 tibble 的每一行都对应原始数据的一行；新增加的列包括:

- .fitted: 预测值，与原始数据同量纲
- .resid: 残差，真实值减去预测值
- .cluster: 聚类结果

以线性回归模型整洁化结果为例演示，其他统计模型、假设检验、K 均值聚类等都是类似的。

```
library(broom)
model = lm(mpg ~ wt, data = mtcars)
model %>% tidy()
```

```
## # A tibble: 2 x 5
##   term        estimate std.error statistic  p.value
##   <chr>          <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)    37.3      1.88      19.9  8.24e-19
## 2 wt             -5.34     0.559     -9.56 1.29e-10
```

```
model %>% glance()
```

```
## # A tibble: 1 x 12
##   r.squared adj.r.squared sigma statistic  p.value    df logLik   AIC
##       <dbl>         <dbl> <dbl>     <dbl>    <dbl> <dbl>  <dbl> <dbl>
## 1     0.753         0.745  3.05      91.4 1.29e-10     1  -80.0  166.
## # ... with 4 more variables: BIC <dbl>, deviance <dbl>,
## #   df.residual <int>, nobs <int>
```

```
model %>% augment()
```

```
## # A tibble: 32 x 9
```

```
##   .rownames           mpg    wt .fitted .resid   .hat .sigma  .cooksd
##   <chr>             <dbl> <dbl>   <dbl>  <dbl>  <dbl>  <dbl>    <dbl>
## 1 Mazda RX4          21    2.62    23.3 -2.28  0.0433   3.07  1.33e-2
## 2 Mazda RX4 Wag      21    2.88    21.9 -0.920 0.0352   3.09  1.72e-3
## 3 Datsun 710         22.8  2.32    24.9 -2.09  0.0584   3.07  1.54e-2
## 4 Hornet 4 Drive     21.4  3.22    20.1  1.30  0.0313   3.09  3.02e-3
## 5 Hornet Sportabout  18.7  3.44    18.9 -0.200 0.0329   3.10  7.60e-5
## 6 Valiant            18.1  3.46    18.8 -0.693 0.0332   3.10  9.21e-4
## # ... with 26 more rows, and 1 more variable: .std.resid <dbl>
```

有了这些模型信息，就可以方便地筛选数据或绘图：

```
model %>% augment() %>%
  ggplot(aes(x = wt, y = mpg)) +
  geom_point() +
  geom_line(aes(y = .fitted), color = "blue") +
  geom_segment(aes(xend = wt, yend = .fitted), color = "red")
```

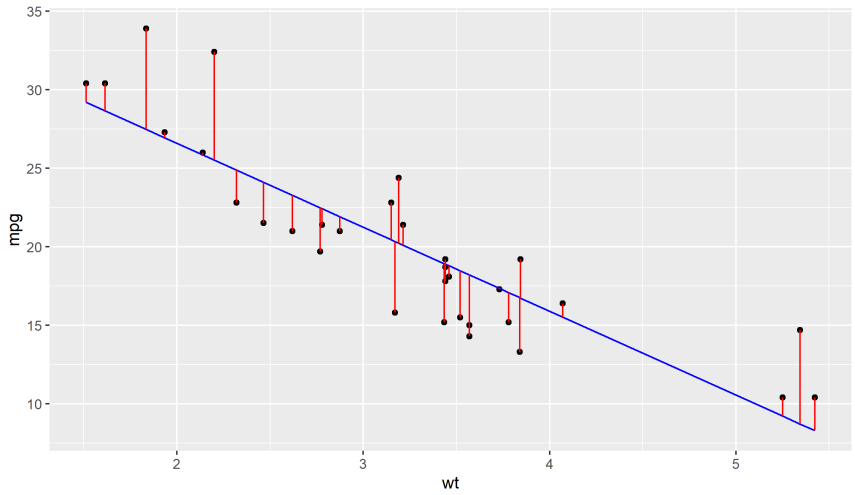

图 3.50: 线性回归偏差图

```
model %>% augment() %>%
  ggplot(aes(x = wt, y = .resid)) +
  geom_point() +
  geom_hline(yintercept = 0, color = "blue")
```

3.3.2 辅助建模

modelr 包提供了一系列辅助建模的函数，便于在 tidyverse 框架下讲授建模。

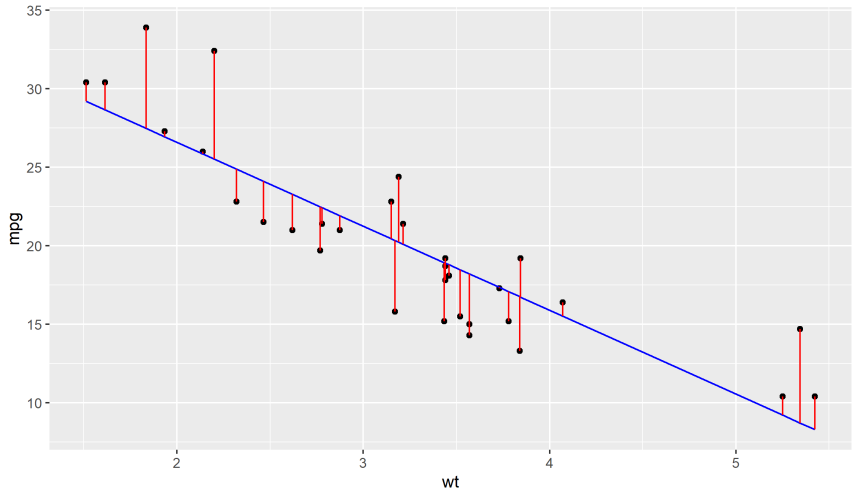

图 3.51: 线性回归残差图

`resample_*()`: 重抽样

重抽样，就是反复从数据集中抽取样本形成若干个 ‘‘替代” 数据集，用于统计推断或模型性能评估。常用的重抽样方法有，简单重抽样 (留出, Holdout)，自助重抽样 (Bootstrap)、交叉验证重抽样 (Cross Validation)、置换重抽样 (Permutation).

- `rsample(data, idx)`: 根据整数向量 `idx` 从数据集 `data` 中重抽样
- `resample_partition(data, p)`: 生成 1 个简单重抽样，即按概率 `p` 对数据集进行划分，比如划分训练集和测试集
- `resample_bootstrap(data)`: 生成 1 个 bootstrap 重抽样
- `bootstrap(data, n)`: 生成 n 个 bootstrap 重抽样
- `crossv_kfold(data, k)`: 生成 k 折交叉验证重抽样
- `crossv_loo(data)`: 生成留一交叉验证重抽样
- `crossv_mc(data, n, test)`: 按测试集占比 `test`, 生成 n 对蒙特卡罗交叉验证
- `resample_permutation(data, columns)`: 按列 `columns` 生成 1 个置换重抽样
- `permute(data, n, columns)`: 按列 `columns` 生成 n 个置换重抽样

这些重抽样结果:

- 为了避免低效操作数据，都是保存原数据的指针;
- 重抽样数据集都存放在返回结果的列表列，借助 `purrr::map` 函数便于批量建模
- 对每个重抽样数据集，应用 `as.data.frame()/as_tibble()` 可转化为数据框，可不用转化直接应用于模型函数

另外，`rsample` 包提供了基本工具创建和分析数据集不同类型的重抽样，更适合与机器学习包 `tidymodels` 连用。

模型性能度量函数

- `rmse(model, data)`: 均方根误差

- mae(model, data): 平均绝对误差
- qae(model, data, probs): 分位数绝对误差
- mape(model, data): 平均绝对百分比误差
- rsae(model, data): 绝对误差相对和
- mse(model, data): 均方误差
- rsquare(model, data): R^2

生成模型数据

- seq_range(x, n): 根据向量 x 值范围生成等间隔序列
- data_grid(data, f1, f2): 生成唯一值的所有组合
- model_matrix(): model.matrix() 的包装，生成模型（设计）矩阵，特别是用于虚拟变量处理

增加预测值列、残差列

- add_predictions()
- add_residuals()

```
library(modelr)
ex = resample_partition(mtcars, c(test = 0.3, train = 0.7))
mod = lm(mpg ~ wt, data = ex$train)
rmse(mod, ex$test)
```

```
## [1] 3.165931
```

```
mod = lm(mpg ~ wt + cyl + vs, data = mtcars)
data_grid(mtcars, wt = seq_range(wt, 10), cyl, vs) %>%
  add_predictions(mod)
```

```
## # A tibble: 60 x 4
##      wt   cyl    vs  pred
##   <dbl> <dbl> <dbl> <dbl>
## 1  1.51     4     0  28.4
## 2  1.51     4     1  28.9
## 3  1.51     6     0  25.6
## 4  1.51     6     1  26.2
## 5  1.51     8     0  22.9
## 6  1.51     8     1  23.4
## # ... with 54 more rows
```

最后，看一个 10 折交叉验证建模的例子。实际上这属于机器学习，经常有人在统计建模时也这么做。

通常将数据集划分为训练集（90%）和测试集（10%），在训练集上训练一个模型，在测试集上评

估模型效果。

只这样做一轮的话，模型效果可能具有偶然性，再一个对数据集利用的也不够充分。k 折交叉验证是克服该缺陷的更好做法，以 10 折交叉验证为例：

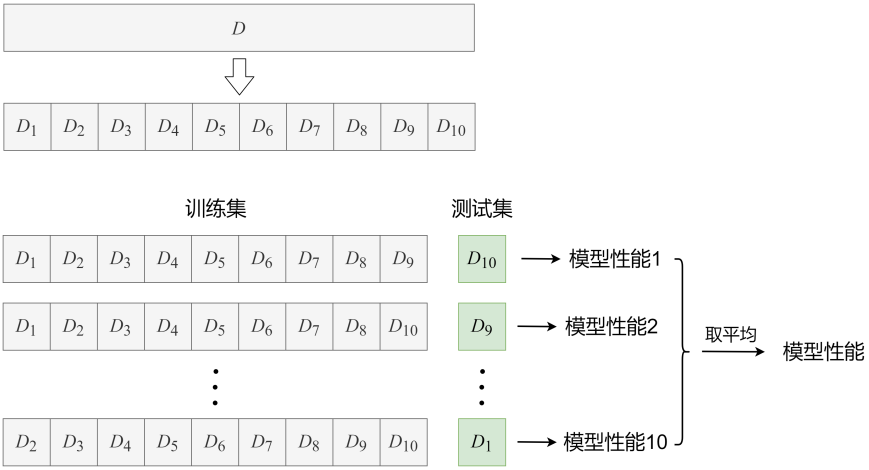

图 3.52: 10 折交叉验证示意图

是将数据集随机分成 10 份，分别以其中 1 份为测试集，其余 9 份为训练集，组成 10 组数据，训练 10 个模型，评估 10 次模型效果，取平均作为最终模型效果。

下面对 mtcars 数据集，采用 10 折交叉验证法构建关于 mpg ~ wt 的线性回归模型，根据 rmse 评估每个模型效果。

先用 crossv_kfold() 生成 10 折交叉验证的数据：

```
cv10 = crossv_kfold(mtcars, 10)
cv10
```

```
## # A tibble: 10 x 3
##   train               test                .id
##   <named list>        <named list>        <chr>
## 1 <resample [28 x 11]> <resample [4 x 11]> 01
## 2 <resample [28 x 11]> <resample [4 x 11]> 02
## 3 <resample [29 x 11]> <resample [3 x 11]> 03
## 4 <resample [29 x 11]> <resample [3 x 11]> 04
## 5 <resample [29 x 11]> <resample [3 x 11]> 05
## 6 <resample [29 x 11]> <resample [3 x 11]> 06
## # ... with 4 more rows
```

结果为 10 行的嵌套数据框，分别对应交叉组成的 10 组训练集 (train)、测试集 (test) 数据。接着是批量建模（具体见下节），与普通的修改列是一样的：（用 map）计算新列 + 赋值。

```
cv10 %>%
  mutate(models = map(train, ~ lm(mpg ~ wt, data = .)),
         rmse = map2_dbl(models, test, rmse))
```

```
## # A tibble: 10 x 5
##    train                test                .id   models        rmse
##    <named list>         <named list>        <chr> <named list> <dbl>
## 1 <resample [28 x 11]> <resample [4 x 11]> 01    <lm>          3.43
## 2 <resample [28 x 11]> <resample [4 x 11]> 02    <lm>          2.68
## 3 <resample [29 x 11]> <resample [3 x 11]> 03    <lm>          2.29
## 4 <resample [29 x 11]> <resample [3 x 11]> 04    <lm>          2.35
## 5 <resample [29 x 11]> <resample [3 x 11]> 05    <lm>          2.15
## 6 <resample [29 x 11]> <resample [3 x 11]> 06    <lm>          2.43
## # ... with 4 more rows
```

要计算最终的平均模型效果，对 `rmse` 列做汇总均值即可（略）。

3.3.3 批量建模

有时候需要对数据做分组，批量地对每个分组建立同样模型，并提取和使用批量的模型结果，这就是批量建模。

批量建模通常是作为探索性数据分析的一种手段，批量建立简单模型以理解复杂的数据集。

批量建模"笨方法"是手动写 `for` 循环实现，再手动提取、合并模型结果。本节要介绍的是 `tidyverse` 中的两种优雅、简洁的做法：

- 用嵌套数据框 + `purrr::map` 实现
- 用 `dplyr` 包的 `rowwise` 技术，具有异曲同工之妙

下面用 `gapminder` 数据集演示，包含 142 个国家的人口、预期寿命、人均 GDP 等。

```
load("datas/gapminder.rda")
gapminder
```

```
## # A tibble: 1,704 x 6
##    country     continent  year lifeExp      pop gdpPercap
##    <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
## 1 Afghanistan Asia       1952    28.8  8425333      779.
## 2 Afghanistan Asia       1957    30.3  9240934      821.
## 3 Afghanistan Asia       1962    32.0 10267083      853.
## 4 Afghanistan Asia       1967    34.0 11537966      836.
## 5 Afghanistan Asia       1972    36.1 13079460      740.
## 6 Afghanistan Asia       1977    38.4 14880372      786.
```

```
## # ... with 1,698 more rows
```

利用嵌套数据框 + purrr::map

先来介绍一个概念：嵌套数据框 (列表列)。我们想要对每个国家 (数据子集) 做重复操作：

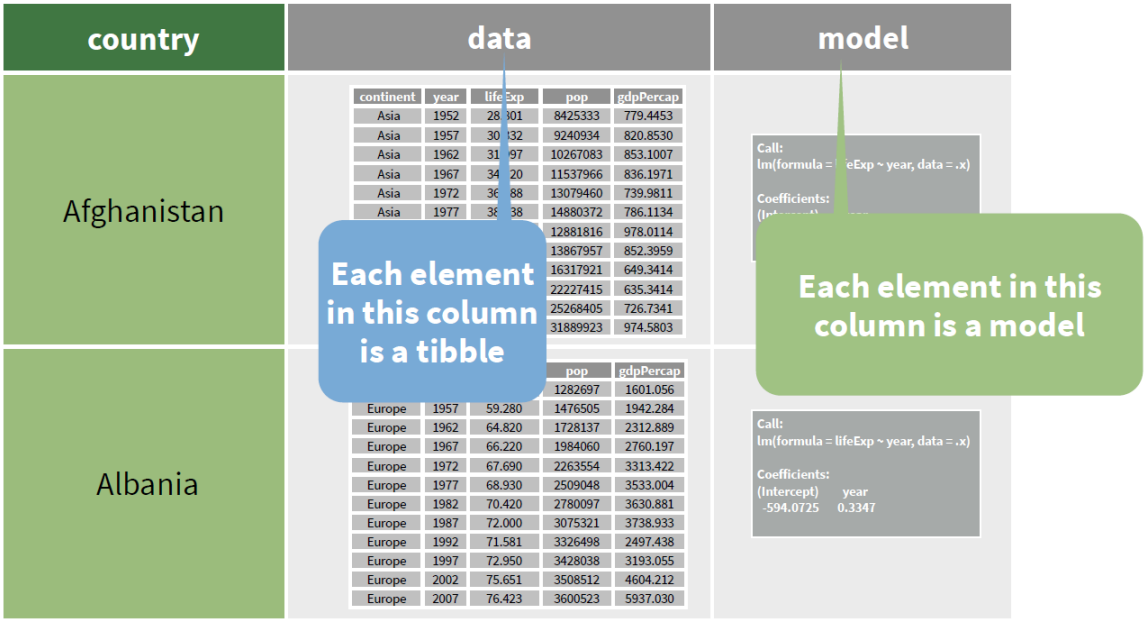

图 3.53: 嵌套数据框示例

先对数据框用 `group_nest()` 关于分组变量 `country` 做分组嵌套，就得到嵌套数据框，每组数据作为数据框嵌套到列表列 `data`；嵌套数据框的每一行是一个分组，表示一个国家的整个时间跨度内的所有观测，而不是某个单独时间点的观测。

```
by_country = gapminder %>%
  group_nest(country)
by_country
```

```
## # A tibble: 142 x 2
##   country                    data
##   <fct>       <list<tibble[,5]>>
## 1 Afghanistan           [12 x 5]
## 2 Albania               [12 x 5]
## 3 Algeria               [12 x 5]
## 4 Angola                [12 x 5]
## 5 Argentina             [12 x 5]
## 6 Australia             [12 x 5]
## # ... with 136 more rows
```

```
by_country$data[[1]]          # 查看列表列的第 1 个元素的内容
unnest(by_country, data)      # 解除嵌套，还原到原数据
```

嵌套数据框与普通数据框一样操作，比如 `filter()` 筛选行、`mutate()` 修改列。这里对嵌套的 `data` 列，用 `mutate()` 修改列，增加一列模型列 `model`，保存为每个国家对应的 `data` 拟合线性回归模型，这就实现了批量建模：

```
by_country = by_country %>%
  mutate(model = map(data, ~ lm(lifeExp ~ year, data = .x)))
by_country
```

```
## # A tibble: 142 x 3
##   country                  data model
##   <fct>       <list<tibble[,5]>> <list>
## 1 Afghanistan           [12 x 5] <lm>
## 2 Albania               [12 x 5] <lm>
## 3 Algeria               [12 x 5] <lm>
## 4 Angola                [12 x 5] <lm>
## 5 Argentina             [12 x 5] <lm>
## 6 Australia             [12 x 5] <lm>
## # ... with 136 more rows
```

继续用 `mutate()` 修改列，借助 `map_*` 函数从模型列、数据列计算均方根误差、R^2、斜率、p 值：

```
by_country %>%
  mutate(rmse = map2_dbl(model, data, rmse),
         rsq = map2_dbl(model, data, rsquare),
         slope = map_dbl(model, ~ coef(.x)[[2]]),
         pval = map_dbl(model, ~ glance(.x)$p.value))
```

```
## # A tibble: 142 x 7
##   country                  data model   rmse   rsq slope     pval
##   <fct>       <list<tibble[,5]>> <list> <dbl> <dbl> <dbl>    <dbl>
## 1 Afghanistan           [12 x 5] <lm>   1.12  0.948 0.275 9.84e- 8
## 2 Albania               [12 x 5] <lm>   1.81  0.911 0.335 1.46e- 6
## 3 Algeria               [12 x 5] <lm>   1.21  0.985 0.569 1.81e-10
## 4 Angola                [12 x 5] <lm>   1.28  0.888 0.209 4.59e- 6
## 5 Argentina             [12 x 5] <lm>   0.267 0.996 0.232 4.22e-13
## 6 Australia             [12 x 5] <lm>   0.567 0.980 0.228 8.67e-10
## # ... with 136 more rows
```

也可以配合 broom 包的函数 `tidy()`，`glance()`，`augment()` 批量、整洁地提取模型结果，这些结果仍是嵌套的列表列，若要完整地显示出来，需要借助 `unnest()` 函数解除嵌套。

- 批量提取模型系数估计及其统计量：

```
by_country %>%
  mutate(result = map(model, tidy)) %>%
  select(country, result) %>%
  unnest(result)
```

```
## # A tibble: 284 x 6
##   country     term         estimate std.error statistic  p.value
##   <fct>       <chr>           <dbl>     <dbl>     <dbl>    <dbl>
## 1 Afghanistan (Intercept)  -508.      40.5       -12.5  1.93e- 7
## 2 Afghanistan year            0.275    0.0205     13.5  9.84e- 8
## 3 Albania     (Intercept)  -594.      65.7        -9.05 3.94e- 6
## 4 Albania     year            0.335    0.0332     10.1  1.46e- 6
## 5 Algeria     (Intercept) -1068.      43.8       -24.4  3.07e-10
## 6 Algeria     year            0.569    0.0221     25.7  1.81e-10
## # ... with 278 more rows
```

- 批量提取模型诊断信息：

```
by_country %>%
  mutate(result = map(model, glance)) %>%
  select(country, result) %>%
  unnest(result)
```

```
## # A tibble: 142 x 13
##   country     r.squared adj.r.squared sigma statistic  p.value    df
##   <fct>           <dbl>         <dbl> <dbl>     <dbl>    <dbl> <dbl>
## 1 Afghanistan     0.948         0.942 1.22      181.  9.84e- 8     1
## 2 Albania         0.911         0.902 1.98      102.  1.46e- 6     1
## 3 Algeria         0.985         0.984 1.32      662.  1.81e-10     1
## 4 Angola          0.888         0.877 1.41       79.1 4.59e- 6     1
## 5 Argentina       0.996         0.995 0.292    2246.  4.22e-13     1
## 6 Australia       0.980         0.978 0.621     481.  8.67e-10     1
## # ... with 136 more rows, and 6 more variables: logLik <dbl>,
## #   AIC <dbl>, BIC <dbl>, deviance <dbl>, df.residual <int>,
## #   nobs <int>
```

- 批量增加预测值列、残差列等：

```
by_country %>%
  mutate(result = map(model, augment)) %>%
  select(country, result) %>%
  unnest(result)
```

```
## # A tibble: 1,704 x 9
##   country     lifeExp  year .fitted  .resid   .hat .sigma   .cooksd
##   <fct>         <dbl> <int>   <dbl>   <dbl>  <dbl>  <dbl>     <dbl>
## 1 Afghanistan    28.8  1952    29.9 -1.11   0.295    1.21 0.243
## 2 Afghanistan    30.3  1957    31.3 -0.952  0.225    1.24 0.113
## 3 Afghanistan    32.0  1962    32.7 -0.664  0.169    1.27 0.0360
## 4 Afghanistan    34.0  1967    34.0 -0.0172 0.127    1.29 0.0000165
## 5 Afghanistan    36.1  1972    35.4  0.674  0.0991   1.27 0.0185
## 6 Afghanistan    38.4  1977    36.8  1.65   0.0851   1.15 0.0923
## # ... with 1,698 more rows, and 1 more variable: .std.resid <dbl>
```

利用 dplyr 包的 rowwise 技术

dplyr 包的 rowwise 按行方式，可以理解为一种特殊的分组：每一行作为一组。

若对 gapminder 数据框用 nest_by() 做嵌套就得到这样 rowwise 化的嵌套数据框：

```
by_country = gapminder %>%
  nest_by(country)
by_country
```

```
## # A tibble: 142 x 2
## # Rowwise:  country
##   country                  data
##   <fct>       <list<tibble[,5]>>
## 1 Afghanistan           [12 x 5]
## 2 Albania               [12 x 5]
## 3 Algeria               [12 x 5]
## 4 Angola                [12 x 5]
## 5 Argentina             [12 x 5]
## 6 Australia             [12 x 5]
## # ... with 136 more rows
```

注意，这回多了 Rowwise: country 信息。

一个国家的数据占一行，rowwise 化的逻辑，就是按行操作数据，正好适合逐行地对每个嵌套的数据框建模和提取模型信息。

这些操作是与 mutate() 和 summarise() 连用来实现，前者会保持 rowwise 模式，但需要计算结果的行数保持不变；后者相当于对每行结果做汇总，结果行数可变 (变多)，不再具有 rowwise 模式。

```
by_country = by_country %>%
  mutate(model = list(lm(lifeExp ~ year, data = data)))
by_country
```

```
## # A tibble: 142 x 3
## # Rowwise:  country
##    country                    data model
##    <fct>       <list<tibble[,5]>> <list>
## 1 Afghanistan           [12 x 5] <lm>
## 2 Albania               [12 x 5] <lm>
## 3 Algeria               [12 x 5] <lm>
## 4 Angola                [12 x 5] <lm>
## 5 Argentina             [12 x 5] <lm>
## 6 Australia             [12 x 5] <lm>
## # ... with 136 more rows
```

下面结果与前文相同，故略过。

直接用 `mutate()` 修改列，从模型列、数据列计算均方根误差、R^2、斜率、p 值：

```
by_country %>%
  mutate(rmse = rmse(model, data),
         rsq = rsquare(model, data),
         slope = coef(model)[[2]],
         pval = glance(model)$p.value)
```

也可以配合 `broom` 包的函数 `tidy()`, `glance()`, `augment()` 批量、整洁地提取模型结果。

- 批量提取模型系数估计及其统计量：

```
by_country %>%
  summarise(tidy(model))
```

- 批量提取模型诊断信息：

```
by_country %>%
  summarise(glance(model))
```

- 批量增加预测值列、残差列等：

```
by_country %>%
  summarise(augment(model))
```

> `rowwise` 化方法的代码更简洁，但速度不如嵌套数据框 + `purrr::map` 快。

本节部分内容参阅 (G. G. Hadley Wickham 2017), (Desi Quintans 2019), Vignettes of `broom`, `modelr`, `dplyr`.