第 3 章 数据储存结构及数据处理

R 语言中常用的数据存储结构类型有数据框(Data. Frame)、向量(Vector)、列表(List)、矩阵(Matrix)等,使用这些数据类型的方法不太相同,需要读者逐个学习。

这些数据结构存储在 R 环境中,可以通过 ls()或者在 RStudio 右侧的 Environment 中查看名称及概要信息。在代码脚本窗口输入对象名称并选中,运行后可以显示其内容。大部分数据结构可以直接或使用 head()函数后通过 view()函数预览,显示结果更加直观。

对于数据稍大的计算过程,通过 rm(对象名)可以从 R 环境中删除该对象,并使用 gc()函数回收内存,这是一个提高计算效率的小技巧。当然,对于大数据有更多的处理方法,如使用 data. table 包、通过 sparklyr 或 sparkR 调用 spark 计算框架、在 Rcpp 中调用 C 代码等。

上述数据类型指的是存储数据类型,还有一种概念是变量数据类型,基本的变量类型可分为双整型(double)、整型(integer)、字符型(character)、逻辑型(logical)、复数类型(complex)等,可参考第 1 章中的相关介绍。

3.1 数据框

Data. Frame 和 Python 中的 Data. Frame 基本类似。熟悉 Excel 的读者在刚接触 Data. Frame 时也可以将其理解为一个没有格式的表格,并且不能有合并单元格、不规则表头名称等内容。数据框列表头即变量名称是必需的,行名称可以有。熟悉关系型数据库的读者可以把 Dataframe 类比数据库中的表。Data. Frame 可以实现数据关联操作(left_join、right_join 等)、筛选(filter)、分组计算等。tidyverse 包中的数据类型 tibble 也可以视为 Data. Frame 的优化版。

有几个实用函数可以帮助读者快速了解数据框:对于 Data. Frame 可以使用 head()函数显示数据的前几行及每个变量的数据类型,方便查看并理解原始数据;对应的 tail()函数可以显示数据的最后几行。str()函数更详细地列出了数据结构,包含变量名称、变量的数据类型等。summary()函数用于快速生成数据摘要统计信息。class()函数可以识别是否是数据框等,通常返回值为 data. frame,以及更多的内容,这就表明对象可能具有复合属性。

通过 readr 及其他包导入的数据大多存储为数据框，as. data. frame()可以将对象转换为数据框，使用 data. frame()函数可以通过输入新建数据框，使用'a <— data. frame()'将输入的数据框存储到特定对象 a 中。前面学习到的将绘图中的外部数据导入 R 中就是采用 Data. Frame 方式进行存储的。Data. Frame 是 ggplot2 可视化高频使用的数据结构，读者务必要掌握。下面的例子描述了如何使用 data. frame()函数新建数据框，新建数据框的代码如下：

```
#代码 3-1 新建数据框
test_df <- data.frame(country = c('a','b'),gdp = c(786,329))
test_df
##country gdp
##1       a 786
##2       b 329
```

3.2 向量

生成向量可以使用 c()函数或者直接输入，下面是一些示例，代码如下：

```
#代码 3-2 新建向量
#使用 c()函数生成一个包含 1～10 的向量
c(1:10)
##[1]  1  2  3  4  5  6  7  8  9  10
#当然也可以直接使用 1:10 实现，和用 c()函数的效果一致
1:10
##[1]  1  2  3  4  5  6  7  8  9  10
#生成一个存储文本的向量
c('a','b','c')
##[1] "a" "b" "c"
```

向量大多适用于手工输入，或者从现有的数据集中提取。大多作为数据筛选匹配条件及生成特定因子等内容使用。

3.3 列表

列表是 R 环境中非常强大的一种复杂数据结构，可以用来保存不同类型的数据，如数字、字符串、向量、子列表等，还可以包含矩阵、函数、绘图对象。用文字不容易说明，初学者可以将其看作一列火车，每节车厢可以承载不同的货物、人、物品等。也可以将其视同为 Excel 表格，单元格中不仅可以是文字、数字，还可以是明细表格、绘图对象等。

列表有多种创建方法：可以使用 list()函数创建，可以通过 split()拆分现有数据框创建，也可以通过导入外部数据创建等，代码如下：

```
#代码 3-3 新建列表
#通过 list()函数生成一个列表
```

```
#建议读者使用 view 函数查看 test_list,更能直观理解上面所讲的立体结构
test_list <- list(a = c('a','b','c'),b = data.frame(category = c('TV','MOBILE')))
```

列表相对抽象,刚接触的读者若现在不理解也不要紧,其主要用于存储复杂结构的数据。另外将数据切割为列表,对于子元素通过并行计算手段实现大数据高速处理。Purr 包中 map 族函数和列表结合使用,也是经典用法,虽然比较抽象,但是值得深入研究学习。

3.4　矩阵

用 matrix()函数可以生成矩阵,实际存储成一个向量,通过行数和列数对应矩阵的元素,存储次序默认为按列存储,代码如下:

```
#代码 3-4 新建矩阵
#用 1～50 生成一个 5 行 8 列的矩阵(按照默认列存储)
matrix(1:50,nrow = 5,ncol = 10)
##[,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##[1,]    1    6   11   16   21   26   31   36   41    46
##[2,]    2    7   12   17   22   27   32   37   42    47
##[3,]    3    8   13   18   23   28   33   38   43    48
##[4,]    4    9   14   19   24   29   34   39   44    49
##[5,]    5   10   15   20   25   30   35   40   45    50
#用 1～50 生成一个 5 行 8 列的矩阵(按照行存储)
matrix(1:50,nrow = 5,ncol = 10,byrow = TRUE)
##[,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##[1,]    1    2    3    4    5    6    7    8    9    10
##[2,]   11   12   13   14   15   16   17   18   19    20
##[3,]   21   22   23   24   25   26   27   28   29    30
##[4,]   31   32   33   34   35   36   37   38   39    40
##[5,]   41   42   43   44   45   46   47   48   49    50
```

下面学习数据处理相关内容。本节主要学习 tidyverse 包,该包其实是一系列包的集合,包含下面 8 个包:

(1) ggplot2 包是现在非常流行的可视化包,也是本书介绍的主要绘图包。

(2) tibble 包中新增加了 tibble 数据结构,可以视作 Data.Frame 的升级版本。

(3) tidyr 包主要用于数据塑形,经典函数包含 gather()函数,用于将宽数据转换为窄数据,spread()函数用于将窄数据转换为宽数据。

(4) readr 包用于将输入导入 R 环境,如前面用到的 read_csv()。

(5) purrr 包用于精简或替代循环,主要是 map 簇函数。

(6) dplyr 包是最为强大的数据处理包之一,包含数据列选取、行筛选、汇总、拆分、塑形等功能,并能作为前台结合 sparklyr 操作 Spark 集群。

(7) stringr 包用于处理文本或字符串,函数以 str 开头,是 stringi 包的常用功能优化版。

(8) forcats 包用于处理分类变量,将其转换为因子,并可以修改因子的顺序、标签等内容。

3.5 readr 包介绍

readr 包主要用于实现 R 与外部数据进行交互传递,实现将 R 环境外的数据写入 R 环境,以及在 R 环境中将数据导出到本地计算机。

3.5.1 read_csv()函数

CSV 文件比较友好,因此读写 CSV 文件是读者必须掌握的技能。本书尽可能地以它为数据存储格式,方便读者学习。通常若无特殊需求,则可在函数中直接写入文件地址,运行后便可导入文件。注意地址不同于 Windows 中的地址,需要使用双斜杠而不是单个斜杠。一般情况下,如果导入失败,则大概率是字符集的问题,将文件字符集修改为 UTF-8 后问题一般会得到解决,代码如下:

```
#代码 3-5 read_csv()函数
library(readr)
data1 <- read_csv('D://Per//MB//bookfile//Mbook//data//salesdata.csv')
head(data1)
###A tibble: 6 × 5
##date       category region quantity  sales
##<chr>   <chr>  <chr>   <dbl>  <dbl>
##1 2021-1-1 A        US          8   877.
##2 2021-1-1 B        US         79 86089.
##3 2021-1-1 C        US          2  1775.
##4 2021-1-1 B        US         21 21050.
##5 2021-1-1 B        US          3  2066.
##6 2021-1-1 C        JP          1   120.
```

read_csv()函数中常用的参数如下:

(1) file 代表要导入文件,可以是完整的路径,也可以是 R 工作路径,通过 setwd()设置。

(2) col_names 可以选择参数 TRUE,代表以第 1 行为列名称。

(3) skip 表示从第几行之后开始导入(有的数据源开头可能有空行)。

(4) n_max 代表导入的最大行数。

同样地,生成的数据结果经常需要从 R 环境导出,这时可以使用 write_csv()函数,格式为 write_csv(R 中的数据集,'导出文件名称.csv'),第 2 个参数可以是完整的文件路径加文件名称,如果仅写文件名称及扩展名,则生成的文件将会存储在当前 R 程序的工作路径中。

当前的工作路径可以使用 getwd()函数获取,并可使用 setwd()函数设定或改变工作路

径。设定路径后，导入/导出文件参数可以只使用包含扩展名的文件名称。

对于导出数据，有时可能存在中文乱码，可以从基础包中使用 read.csv 函数替代，相对来讲处理中文较友好些(只是会增加一列序号，即索引列)。另外，使用下面的方法大概率也可以处理中文乱码问题：使用 WPS 表格打开文件之后另存或者使用 Notepad＋＋打开后将字符集更改为 UTF-8。

read_csv()函数对某些字符集可能在导入时会解析失败，从而导致出错，这时可以使用 guess_econding()函数识别已经导入的数据集字符集，之后在 read_csv()函数中设置 locale 参数中的字符集，重新导入即可。另外，使用 txt 方式打开文件，之后在保存选项中使用字符集 UTF-8，保存时覆盖原始文件即可。一般使用上面的方法可以解决导入出错问题，详细内容可以参看帮助文件。

3.5.2　其他主要函数

read_tsv()用于读取以制表符 Tab 为分隔符号的 CSV 文件。read_delim()用于读取以指定符号为分隔符的文件，可以理解为 read_csv、read_tsv 的通用版本。read_fwf()用于读取固定宽度的文件，可以理解为被读取文件中各列的宽度是固定的。read_table()为常用函数，可以读取 txt 文件。

对应的 R 环境数据导出功能以 write_开头，可以在帮助文件中查看各自的使用方法。

3.6　tidyr 包

tidyr 包主要用于数据处理，也就是数据塑形及长短数据转换等功能。

3.6.1　expand_grid()函数

expand_grid()函数用于把多个已知向量中的元素生成所有组合，例如从 1、2 与 a、b 这两组向量中分别抽取一个元素，生成组合 1-a、1-b、2-a、2-b，代码如下：

```
#代码 3-6 expand_grid()函数
library(tidyr)
expand_grid(c(1,2),c('a','b'))
###A tibble: 4 × 2
##`c(1, 2)` `c("a", "b")`
##<dbl><chr>
##1          1 a
##2          1 b
##3          2 a
##4          2 b
```

expand_grid()函数主要用于在数据拼接过程中生成关键字段的最大集，如在 left_join 中生成左表关键字段，可以类比理解为数据库中的多表在拼接时需先生成一个最全变量作

为左表主键。

类似地，使用 crossing ()函数或基础包中的 merge()函数都可以实现上述操作，具体操作比较简单，代码如下：

```
#代码 3-7 crossing()及 merge()函数
library(tidyr)
crossing(c(1,2),c('a','b'))
##A tibble: 4 × 2
##`c(1, 2)` `c("a", "b")`
##<dbl><chr>
##1         1 a
##2         1 b
##3         2 a
##4         2 b
merge(c(1,2),c('a','b'))
##x y
##1 1 a
##2 2 a
##3 1 b
##4 2 b
```

merge()函数相比较前两个函数的结果，对于不同的结果变量名称，merge()函数会使用 x 和 y 表示，而前两个函数使用的是原始向量作为变量名称。另外，merge()函数不支持两个以上向量生成新组合，expand_grid()及 crossing()函数支持两个以上向量作为输入参数，生成新的组合，但是 crossing()及 expand_grid()的限制条件是输入的多个向量不能有重复的向量，即输入类似 c('a', 'b')和 c('a', 'b')，则代码会出错。有这类特殊需求的读者可以使用循环来处理。

3.6.2 drop_na()函数

drop_na()函数用来删除存在 NA 值的行。drop_na()如果不指定 NA 所在的变量，则会去除所有包含 NA 的行。第 2 个参数若指定了 NA 所在列，则只去除该列中包含 NA 值的行，代码如下：

```
#代码 3-8 drop_na()函数
library(dplyr)
#生成一个数据框 df
df <- data.frame(x = c(1, 2, NA), y = c("a", NA, "b"))
#删除数据框中任意列有 NA 值的行
drop_na(df)
##x y
##1 1 a
#删除数据框中 x 列有 NA 值的行
drop_na(df,x)
##x    y
```

```
##1 1    a
##2 2 <NA>
#删除数据框中 y 列有 NA 值的行
drop_na(df,y)
## x y
##1  1 a
##2 NA b
```

基础包中的 na.omit()函数也可以用来去除错误值，但是只能实现上例中的第 1 种用法，相对不够灵活，代码如下：

```
#代码 3-9 na.omit()函数
#生成一个数据框 df
df <- data.frame(x = c(1, 2, NA), y = c("a", NA, "b"))
#删除数据框中任意列有 NA 值的行
na.omit(df)
## x y
##1 1 a
```

3.6.3　replace_na()函数

replace_na()函数用于将 NA 值替换为指定的值。注意其中参数需要以 list 形式输入，代码如下：

```
#代码 3-10 replace_na()函数 1
#生成 df 数据框
df <- data.frame(x = c(1, 2, NA))
#打印 df 数据框
print(df)
## x
##1  1
##2  2
##3 NA
#将 df 数据框中 x 中的 NA 替换为 0
df %>% replace_na(list(x = 0))
## x
##1 1
##2 2
##3 0
```

其中第 2 个参数需要以 list 形式输入，表面增加了代码输入，略显烦琐，但是当需要对多变量中的 NA 值进行替换时，则表现出极大的灵活性，如在下面的数据框中，分别用 0、1 替换变量 x、y 中的 NA 值，代码如下：

```
#代码 3-11 replace_na()函数 2
#生成 df 数据框
```

```
df <- data.frame(x = c(1, 2, NA),y = c(2,NA,3))
#打印 df 数据框
print(df)
##x  y
##1  1  2
##2  2 NA
##3 NA  3
#将 df 数据框中 x 中的 NA 替换为 0,将 y 中的 NA 替换为 1
df %>% replace_na(list(x = 0,y = 1))
##x y
##1 1 2
##2 2 1
##3 0 3
```

3.6.4 extract()函数

extract()函数主要用于对数据框中的某列按照特定字符拆分,类似于 Excel 中的分列操作。该函数的难点在于参数 regex,其使用的是正则表达式的匹配模式。如下列将现有 x 列按照其中的"-"符号拆分为"第 1 列""第 2 列",代码如下:

```
#代码 3-12 extract()函数
library(dplyr)
#生成数据框 df
df <- data.frame(x = c(NA, "a-b", "a-d", "b-c", "d-e"))
#将数据框 df 中的 x 列按照"-"拆分为 2 列,即第 1 列、第 2 列
extract(df,col = x, into = c("第 1 列", "第 2 列"), regex = "([a-z]+)-([a-z]+)")
##第 1 列 第 2 列
##1   <NA>   <NA>
##2      a      b
##3      a      d
##4      b      c
##5      d      e
```

regex='([a-z]+)-([a-z]+)'表示要拆分的内容分为 3 部分,第 1 部分是 a-z 中的任意值,接下来是-号,最后是 a-z 中的任意值,()在正则表达式中表示一个组合。正则表达式匹配模式是非常复杂的内容,有兴趣的读者可以参考其他专业书籍。和 Excel 函数中的分列比较,当分隔符出现多次时,Excel 会按照出现分隔符的次数进行分列。extract()函数遇到这类情况就比较难处理了。

3.6.5 fill()函数

对于数据框中的 NA 值,运用 fill()函数可以使用相邻非 NA 值替换,其中,通过参数 direction 设置填充值的来源:up 来自 NA 值下方,down 来自 NA 值上方,downup 首先用上方值填充,当上方值是 NA 时,使用下方值填充。updown 和 downup 的填充顺序相反,代码如下:

```
#代码 3-13 fill()函数
df <- data.frame(x = c( "a/b",NA, "a/d", "b/c", "d/e"))
df %>% fill(x,.direction = 'up')
##x
##1 a/b
##2 a/d
##3 a/d
##4 b/c
##5 d/e
df %>% fill(x,.direction = 'down')
##x
##1 a/b
##2 a/b
##3 a/d
##4 b/c
##5 d/e
```

当把数据中存在合并单元格的 Excel 表格转换为 CSV 文件或者直接导入 R 环境都会造成原来合并单元格区域只有第 1 个单元格值可以正确显示，其余的为 NA。使用 fill()函数向下填充，能快速解决此问题。Excel 中类似的情况也比较容易处理，一般使用查找功能找到所有 NA 所在单元格，之后输入公式，最后按下 Ctrl＋Enter 快捷键即可。当然使用 VBA 也可以实现上述功能。

3.6.6　gather()函数

gather()函数是数据塑形中重要的函数，将宽表格整理为长表格，类似于 Excel PowerQuery 中的逆透视列。在一些相对陈旧的 R 代码中会遇到 Reshape2 包中的 melt()函数，其功能类似。

该函数主要的结构如下：

gather(数据，key＝行数据折叠为一列，新列名字， value＝在折叠列下方的数值，会被折叠为一列，这一列的新名称，-c(指定不参与折叠的列))

其中，-c(指定不参与折叠的列)也可以换为 c(指定参与折叠的列)。

在下面的例子中，原始数据是按照月份排列的销售数据，用 gather()函数将其折叠，最终月份、销售分别显示在两列，变为一个长表，代码如下：

```
#代码 3-14 gather()函数
library(tidyr)
#新建一个 data.frame: 各个品类在 1 月和 2 月的销售数据
df_wide <- data.frame(category = c('a','b','c','d'),Jan = c(1,2,4,6),Feb = c(1,1,1,1))

#打印显示 df_wide 数据框
print(df_wide)
##category Jan Feb
##1        a   1   1
```

```
## 2        b   2   1
## 3        c   4   1
## 4        d   6   1
#将 df_wide 中的月份'Jan/Feb'折叠到列'month',将销售数据折叠到'Sales'列
gather(df_wide,key = 'month',value = 'Sales', - c(category))
## category month Sales
## 1        a   Jan     1
## 2        b   Jan     2
## 3        c   Jan     4
## 4        d   Jan     6
## 5        a   Feb     1
## 6        b   Feb     1
## 7        c   Feb     1
## 8        d   Feb     1
```

长表是绘图常用的格式,特别适合使用 ggplot2 绘图。另外,长表也适合使用 group_by()函数及 summarise()函数,实现分组汇总、计算等更多内容。当然,分组后可以一次性对每组数据使用相同的操作,不一定汇总,如给每组加上行序号等。

3.6.7 pivot_longer()函数

pivot_longer()函数可以认为是 gather()函数的升级版本,也用于将数据由宽表转换为长表。pivot_longer 中的第 1 个参数是待处理的数据框,被折叠的变量名称会存储到一列,第 2 个参数 names_to 是新列的名称。原来的数据区域会被折叠到一列中,values_to 参数决定了这一列的名称。最后一个参数-c()表示不被折叠的列是哪几列,同理如果使用参数 c(),则指定需要被折叠的列范围,代码如下:

```
#代码 3 - 15 pivot_longer()函数
library(tidyr)
#新建一个 data.frame: 各个品类在 1 月和 2 月的销售数据
df_wide < - data.frame(category = c('a','b','c','d'),Jan = c(1,2,4,6),Feb = c(1,1,1,1))

#打印显示 df_wide 数据框
print(df_wide)
## category Jan Feb
## 1        a   1   1
## 2        b   2   1
## 3        c   4   1
## 4        d   6   1
#将 df_wide 中的月份'Jan/Feb'折叠到列'month',将销售数据折叠到'Sales'列
pivot_longer(df_wide,names_to = 'month',values_to = 'Sales', - c(category))
### A tibble: 8 × 3
## category month Sales
## < chr >    < chr > < dbl >
## 1 a         Jan       1
## 2 a         Feb       1
```

```
##3 b        Jan       2
##4 b        Feb       1
##5 c        Jan       4
##6 c        Feb       1
##7 d        Jan       6
##8 d        Feb       1
```

3.6.8　spread()函数

spread()函数与 gather()函数实现相反的功能，即将长表转换为宽表。spread()函数的第 1 个参数表示待处理的数据框，第 2 个参数 key 表示将哪一列变为行标题，最后一个参数 value 表示用哪列填充数据区域，代码如下：

```
#代码 3-16 spread()函数
#生成 3 个向量，用于生成长表
category <- c("a","a","b","b","c","c","d","d")
month <- c("Jan","Feb","Jan","Feb","Jan","Feb","Jan","Feb")
Sales <- c(1,1,2,1,4,1,6,1)
#将上述 3 个向量生成一个长表 df_longer(等同于上列的计算结果)
df_longer <- data.frame(category,month,Sales)
print(df_longer)
## category month Sales
##1        a   Jan     1
##2        a   Feb     1
##3        b   Jan     2
##4        b   Feb     1
##5        c   Jan     4
##6        c   Feb     1
##7        d   Jan     6
##8        d   Feb     1
#将月份由 month 映射到列，生成宽表
spread(df_longer,key = 'month',value = 'Sales')
## category Feb Jan
##1        a   1   1
##2        b   1   2
##3        c   1   4
##4        d   1   6
```

宽表常用来计算增长率等内容。如上例将 Jan 和 Feb 销售额并列，就可以方便地计算每个品类 2 月间的销售增长率。当然结合 lag()或 lead()函数也可以在长表中实现上述操作。

3.6.9　pivot_wider()函数

pivot_wider()函数是 spread()函数的升级版本，其功能也是将长表转换为宽表。pivot_wider()中的第 1 个参数表示待处理的数据框，names_from 参数表示将哪一个变量映

射为多列、values_from 区域表示将哪一列数值映射到变形后的区域，代码如下：

```
#代码 3-17 pivot_wider()函数
#生成 3 个向量,用于生成长表
category <- c("a","a","b","b","c","c","d","d")
month <- c("Jan","Feb","Jan","Feb","Jan","Feb","Jan","Feb")
Sales <- c(1,1,2,1,4,1,6,1)
#将上述 3 个向量生成一个长表 df_longer(等同于上列的计算结果)
df_longer <- data.frame(category,month,Sales)
print(df_longer)
##category month Sales
##1        a   Jan     1
##2        a   Feb     1
##3        b   Jan     2
##4        b   Feb     1
##5        c   Jan     4
##6        c   Feb     1
##7        d   Jan     6
##8        d   Feb     1
#将月份由 month 映射到列,生成宽表
pivot_wider(df_longer,names_from = 'month',values_from = 'Sales')
### A tibble: 4 × 3
##category   Jan   Feb
##<chr>    <dbl><dbl>
##1 a            1     1
##2 b            2     1
##3 c            4     1
##4 d            6     1
```

3.7 dplyr 包

在 R 语言的当前数据处理领域，dplyr 包占据了核心位置。dplyr 的功能包含数据列选取、行筛选、汇总、排序、表间拼接等功能，其内容非常丰富。由于篇幅限制，本书仅介绍较为常用的几个函数。笔者强烈建议，读者如果对数据处理技能有更进一步的要求，则可以对整个包中的函数进行学习，一定会有不少收获。

3.7.1 select()函数

select()函数主要用于选取数据框中的列，配合其中的参数 starts_with 等，可以实现灵活地按列筛选的效果。以经典的 datasets 包自带的鸢尾花卉数据集 iris 为例子：该数据集包含的列有 Sepal. Length 花萼长度、Sepal. Width 花萼宽度、Petal. Length 花瓣长度、Petal. Width 花瓣宽度、Species 种类。Species 字段中包含 Iris setosa 山鸢尾、Iris versicolour 杂色鸢尾、Iris virginica 弗吉尼亚鸢尾。

在下面的代码中会使用 magrittr 包中的%>%管道符号，该符号将其左侧计算结果自

动传递给右侧函数，十分方便，代码如下：

```
#代码 3-18 select()函数 1
#在下面的代码中为节省显示空间，使用 head(5)显示上述条件下的前 5 行
library(dplyr)
library(magrittr)
##
## Attaching package: 'magrittr'
## The following object is masked from 'package:purrr':
##
## set_names
## The following object is masked from 'package:tidyr':
##
## extract
#选取单一列 Sepal.Length 花萼长度，并选取前 5 行
select(iris,'Sepal.Length') %>% head(5)
## Sepal.Length
## 1          5.1
## 2          4.9
## 3          4.7
## 4          4.6
## 5          5.0
#同时选取多列，并选取前 5 行
select(iris,c('Sepal.Length','Sepal.Width')) %>% head(5)
## Sepal.Length Sepal.Width
## 1          5.1         3.5
## 2          4.9         3.0
## 3          4.7         3.2
## 4          4.6         3.1
## 5          5.0         3.6
#通过 starts_with 参数智能地选取以 S 开头的列，并选取前 5 行
select(iris,starts_with('S')) %>% head(5)
## Sepal.Length Sepal.Width Species
## 1          5.1         3.5  setosa
## 2          4.9         3.0  setosa
## 3          4.7         3.2  setosa
## 4          4.6         3.1  setosa
## 5          5.0         3.6  setosa
```

select(iris, 'Sepal. Length')表示从数据集 iris 中选取'Sepal. Length'这 1 列(或者描述为选择'Sepal. Length'变量)。select(iris,c('Sepal. Length', 'Sepal. Width'))表示从数据集 iris 中选取'Sepal. Length'和'Sepal. Width'这 2 列，需要选择的多列以向量方式输入。在 select()函数中和 starts_with 类似的参数还有 ends_with()、contains()、matches()、num_range()。除了 num_range()外，其他几个函数都比较好理解。如果一个数据框中包含了类似变量'A1'、'A2'、'A3'等有规律的变量，则使用 num_range()可以非常有效率地选择它们，代码如下：

```
#代码 3-19 select()函数 2
library(dplyr)
library(magrittr)
#新建一个数据框
df <- data.frame(A1 = c(1:5),A2 = c(2:6),A3 = c(3:7),A4 = c(9,8,3,4,7))
#选择变量 A1、A2、A3
df %>% select(num_range("A",1:3))
##A1 A2 A3
##1  1  2  3
##2  2  3  4
##3  3  4  5
##4  4  5  6
##5  5  6  7
```

除可以使用 num_range()外,也可以将筛选条件构建为向量,之后将该条件代入 select()函数中,下面使用 paste0("A",1:3)生成向量 A1、A2、A3 作为筛选条件,同样可以实现代码 3-19 中的内容,代码如下:

```
#代码 3-20 select()函数 3
library(dplyr)
library(magrittr)
#新建一个数据框
df <- data.frame(A1 = c(1:5),A2 = c(2:6),A3 = c(3:7),A4 = c(9,8,3,4,7))
#选择变量 A1、A2、A3
df %>% select(paste0("A",1:3))
##A1 A2 A3
##1  1  2  3
##2  2  3  4
##3  3  4  5
##4  4  5  6
##5  5  6  7
```

代码 3-18 中管道符号只是用在最后的 head()函数前, select()前也可以使用管道符号,充分利用管道操作的便利性,最终结果和上面代码的结果是一致的,代码如下:

```
#代码 3-21 select()函数 4
#在下面的代码中为节省显示空间,使用 head(5)显示上述条件下的前 5 行
library(dplyr)
library(magrittr)
#选取单一列 Sepal.Length 花萼长度,并选取前 5 行
iris %>% select('Sepal.Length') %>% head(5)
##Sepal.Length
##1          5.1
##2          4.9
##3          4.7
##4          4.6
##5          5.0
```

```
#同时选取多列,并选取前5行
iris %>% select(c('Sepal.Length','Sepal.Width')) %>% head(5)
##Sepal.Length Sepal.Width
##1          5.1         3.5
##2          4.9         3.0
##3          4.7         3.2
##4          4.6         3.1
##5          5.0         3.6
#通过starts_with参数智能地选取以S开头的列,并选取前5行
iris %>% select(starts_with('S')) %>% head(5)
##Sepal.Length Sepal.Width Species
##1          5.1         3.5  setosa
##2          4.9         3.0  setosa
##3          4.7         3.2  setosa
##4          4.6         3.1  setosa
##5          5.0         3.6  setosa
```

除了可以通过模糊匹配、精确输入等方式选择列,也可以选择连续相邻的几列。下面选择数据框中的A1、A2、A3列,由于它们在原始数据框中是邻近的,所以使用A1:A3输入select()中即可,代码如下:

```
#代码3-22 select()函数5
library(dplyr)
library(magrittr)
#新建一个数据框
df <- data.frame(A1 = c(1:5),A2 = c(2:6),A3 = c(3:7),A4 = c(9,8,3,4,7))
#选择变量A1、A2、A3
df %>% select(A1:A3)
##A1 A2 A3
##1  1  2  3
##2  2  3  4
##3  3  4  5
##4  4  5  6
##5  5  6  7
```

select()函数结合everything()参数也非常有用,向select()中单独输入everything()参数表示选择所有列,代码如下:

```
#代码3-23 select()函数与everything()参数结合使用1
library(dplyr)
library(magrittr)
#新建一个数据框
df <- data.frame(A1 = c(1:5),A2 = c(2:6),A3 = c(3:7),A4 = c(9,8,3,4,7))
#选择变量A1、A2、A3
df %>% select(everything())
##A1 A2 A3 A4
##1  1  2  3  9
```

```
##2  2  3  4  8
##3  3  4  5  3
##4  4  5  6  4
##5  5  6  7  7
```

向 select()中输入具体列名称之后输入 everything()参数,表示在选择时先选择具体列,接下来选择其他所有列。这种方法常用来把某些列排序到数据框的最左侧,代码如下:

```
#代码 3-24 select()函数与 everything()参数结合使用 2
library(dplyr)
library(magrittr)
#新建一个数据框
df <- data.frame(A1 = c(1:5),A2 = c(2:6),A3 = c(3:7),A4 = c(9,8,3,4,7))
#选择变量 A1、A2、A3
df %>% select(A4,everything())
##A4 A1 A2 A3
##1  9  1  2  3
##2  8  2  3  4
##3  3  3  4  5
##4  4  4  5  6
##5  7  5  6  7
```

3.7.2 filter()函数

filter()函数主要用于对数据集列进行筛选,其中条件筛选中:“等于”使用双等号“==”“或”使用“|”“且”使用“&”。如果需要类似 SQL 中的 like 语句功能,则可以结合基础包中的 grepl()函数或者 stringr 包中的 str_detect()函数实现,代码如下:

```
#代码 3-25 filter()函数
#在下面的代码中为节省显示空间,使用 head(5)显示上述条件下的前 5 行
library(dplyr)
library(magrittr)
#筛选品类为'setosa'的记录,显示符合品类等于 Iris setosa 的所有列

iris %>% filter(Species == 'setosa') %>% head(5)
##Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##1          5.1         3.5          1.4         0.2  setosa
##2          4.9         3.0          1.4         0.2  setosa
##3          4.7         3.2          1.3         0.2  setosa
##4          4.6         3.1          1.5         0.2  setosa
##5          5.0         3.6          1.4         0.2  setosa
#筛选品类为'setosa'或为'versicolor'的记录
iris %>% filter(Species == 'setosa'|Species == 'versicolor') %>% head(5)
##Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##1          5.1         3.5          1.4         0.2  setosa
##2          4.9         3.0          1.4         0.2  setosa
```

```
##3          4.7         3.2          1.3         0.2  setosa
##4          4.6         3.1          1.5         0.2  setosa
##5          5.0         3.6          1.4         0.2  setosa
#筛选品类为'setosa'且 Sepal.Length>6.5 的记录
iris %>% filter(Species == 'setosa',Sepal.Length>6.5) %>% head(5)
##[1] Sepal.Length Sepal.Width  Petal.Length Petal.Width  Species
##<0 rows> (or 0-length row.names)
#筛选品类名称中包含'ir'的所有行,类似于 SQL 语句中的%like%
iris %>% filter(grepl('ir',Species)) %>% head(5)
##Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
##1          6.3         3.3          6.0         2.5 virginica
##2          5.8         2.7          5.1         1.9 virginica
##3          7.1         3.0          5.9         2.1 virginica
##4          6.3         2.9          5.6         1.8 virginica
##5          6.5         3.0          5.8         2.2 virginica
```

iris %>% filter(Species==='setosa')表示筛选 iris 中 Species 等于 setosa 的所有记录。iris %>% filter(Species==='setosa'|Species==='versicolor')表示选择 iris 中 Species 等于 setosa 或 versicolor 的所有记录,使用%in%也可以实现该功能。filter(grepl('ir',Species))表示筛选 Species 包含'ir'这两个字母的所有行,使用 stringr 包中的 str_detect()函数也可以实现上述功能。上面提到的%in%、str_detect()函数的用法可参考下面的例子,代码如下:

```
#代码 3-26 filter()函数模糊选取
library(dplyr)
library(magrittr)
#筛选品类为'setosa'或为'versicolor'的记录
iris %>% filter(Species %in% c('setosa','versicolor')) %>% head(5)
##Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##1          5.1         3.5          1.4         0.2  setosa
##2          4.9         3.0          1.4         0.2  setosa
##3          4.7         3.2          1.3         0.2  setosa
##4          4.6         3.1          1.5         0.2  setosa
##5          5.0         3.6          1.4         0.2  setosa
#筛选品类名称中包含'ir'的所有行,类似于 SQL 语句中的%like%
iris %>% mutate(Species=as.character(Species)) %>%
filter(stringr::str_detect('to',Species)) %>% head(5)
##[1] Sepal.Length Sepal.Width  Petal.Length Petal.Width  Species
##<0 rows> (or 0-length row.names)
```

3.7.3　mutate()函数

mutate()函数主要对数据框中的现有列进行修改,此外还可以新增列,当原数据中该列已经存在时,mutate()函数代码执行的是更新列动作;当列不存在时,执行的是新增列动作,代码如下:

```
#代码 3-27 mutate()函数新建或更改变量
#在下面的代码中为节省显示空间,使用 head(5)显示前 5 行
library(dplyr)
library(magrittr)

#在原来数据集中增加一列 sequence(序号列)
iris %>% mutate(sequence = c(1:NROW(.))) %>% head(5)
##Sepal.Length Sepal.Width Petal.Length Petal.Width Species sequence
##1          5.1         3.5          1.4         0.2  setosa        1
##2          4.9         3.0          1.4         0.2  setosa        2
##3          4.7         3.2          1.3         0.2  setosa        3
##4          4.6         3.1          1.5         0.2  setosa        4
##5          5.0         3.6          1.4         0.2  setosa        5
#在原来数据集中增加一列:将 Sepal.Width 与 Petal.Width 中的数值相加得到 Width_Total 列
iris %>% mutate(Width_Total = Sepal.Width + Petal.Width) %>% head(5)
##Sepal.Length Sepal.Width Petal.Length Petal.Width Species Width_Total
##1          5.1         3.5          1.4         0.2  setosa         3.7
##2          4.9         3.0          1.4         0.2  setosa         3.2
##3          4.7         3.2          1.3         0.2  setosa         3.4
##4          4.6         3.1          1.5         0.2  setosa         3.3
##5          5.0         3.6          1.4         0.2  setosa         3.8
```

1:NROW(.)与 1:NROW(iris)表示的内容一致,即都生成一个从 1 到最大行数的序列。NROW(iris)可以获取数据框 iris 的行数,其中简写为标点符号“.”指代管道操作符号前的数据框 iris 数据集。当使用 mutate()函数新增加列时,如果只希望保留新增加的列,则可以在 mutate()后面使用 select()函数选择该列,也可以使用 transmute()函数实现只保留生成列的效果,代码如下:

```
#代码 3-28 transmute()函数
#在下面的代码中为节省显示空间,使用 head(5)显示前 5 行
library(dplyr)
library(magrittr)

#在原来数据集中增加一列:将 Sepal.Width 与 Petal.Width 中的数值相加得到 Width_Total 列
#并且只保留该新增加的列
iris %>% transmute(Width_Total = Sepal.Width + Petal.Width) %>% head(5)
##Width_Total
##1         3.7
##2         3.2
##3         3.4
##4         3.3
##5         3.8
```

transmute()实现了新增加 Width_Total 列,并删除了其他列。上面介绍了 mutate()新增或者更新现有列的常见方法,还有几个函数可以增强这些功能,这些函数包含 lag()、lead()、cumsum()、cummin()、cummax()等。下面先介绍偏移函数 lag()和 lead(),详细用法可参

考下面的例子，代码如下：

```
#代码 3-29 lag()及 lead()函数
library(dplyr)
library(magrittr)
df <- data.frame(Year = c('2020','2021','2022'),
                 Sales = c(125,368,478))
df %>% mutate(lag_sales = lag(Sales),
              lead_sales = lead(Sales))
##Year Sales lag_sales lead_sales
##1 2020   125        NA        368
##2 2021   368       125        478
##3 2022   478       368         NA
```

lag(Sales)表示将 Sales 这一列整体往下移动一行，最终在原始数据框中 2020 年的 Sales 数值 125 将显示在 2021 年，2021 年的 Sales 数值 368 将显示在 2022 年。lead(Sales)将数据提前一行，其他逻辑同 lag()函数。由于 lag(Sales)和 lead(Sales)将数据移动错位了，所以会出现 NA 值，可以设置 default 参数将 NA 替换为指定的值，一般设置为 0，代码如下：

```
#代码 3-30 在 lag()及 lead()函数中使用 default 参数
library(dplyr)
library(magrittr)
df <- data.frame(Year = c('2020','2021','2022'),
                 Sales = c(125,368,478))
df %>% mutate(lag_sales = lag(Sales,default = 0),
              lead_sales = lead(Sales,default = 0))
##Year Sales lag_sales lead_sales
##1 2020   125         0        368
##2 2021   368       125        478
##3 2022   478       368          0
```

在上面的 lag()和 lead()函数中可以设置 n 参数值来确定移动的行数，在默认情况下 n=1。偏移函数在实际运用中用来计算变动额、变动率非常方便。下面计算 2020、2021、2022 这 3 年间 Sales 的增长额、增长率，代码如下：

```
#代码 3-31 lag()及 lead()函数的实际运用
library(dplyr)
library(magrittr)
df <- data.frame(Year = c('2020','2021','2022'),
                 Sales = c(125,368,478))
df %>% mutate(change = Sales - lag(Sales),
              change_percent = Sales/lag(Sales) - 1)
##Year Sales change change_percent
##1 2020   125     NA             NA
##2 2021   368    243       1.944000
##3 2022   478    110       0.298913
```

如果不使用偏移函数，则整个计算过程会稍显烦琐，下面使用循环来处理，代码如下：

```
#代码 3-32 模拟 lag()及 lead()函数
library(dplyr)
library(magrittr)
df <- data.frame(Year = c('2020','2021','2022'),
                 Sales = c(125,368,478))
df $ change <- 0
df $ change_percent <- 0
for (i in 2:NROW(df) ){
  df $ change[i] <- df $ Sales[i] - df $ Sales[i-1]
  df $ change_percent[i] <- df $ Sales[i]/df $ Sales[i-1] - 1
}

df
##Year Sales change change_percent
##1 2020   125      0       0.000000
##2 2021   368    243       1.944000
##3 2022   478    110       0.298913
```

代码 3-32 首先新增加列 change、change_percent 并赋值为 0，之后从第 2 行开始循环计算，每行 change 值等于当前行 Sales 值减去上一行 Sales 值。当前行 Sales 值用 dfSales[i]表示，上一行 Sales 值用 dfSales[i-1]表示，其中 i 的值为 2 和 3，3 通过 NROW(df)获得。

cumsum()函数用于计算滚动累加，用于在数据框中对变量和向量进行累加，代码如下：

```
#代码 3-33 cumsum()函数实现累加
library(dplyr)
library(magrittr)
df <- c(1,2,3,1,6,7,1,2,9)
cumsum(df)
##[1]  1  3  6  7 13 20 21 23 32
```

另外，实际运用较频繁的是 cumsum()，结合后面的 group_by()可以实现组内的上述功能。cummax()、cummax()的用法和 cumsum()的用法类似，可以求累计最大值、累计最小值。cumprod()则可以实现累乘的效果，如将数据 1～10 滚动累乘，代码如下：

```
#代码 3-34 cumprod()函数实现累乘
library(dplyr)
library(magrittr)
df <- c(1:10)
cumprod(df)
##[1]       1       2       6      24     120     720    5040   40320  362880
##[10] 3628800
```

3.7.4　group_by()与 summarise()函数

group_by()函数与 SQL 中的 group by 语法类似，在括号内输入一个或多个分类变量作为分组依据。group_by()函数与 summarise()函数结合使用，可实现分组求汇总值、最大值等功能，代码如下：

```
#代码 3-35 分组汇总
library(dplyr)
library(magrittr)

#在 iris 数据集中计算每个品类有多少行(记录数)
iris %>% group_by(Species) %>% summarise(rows_count = n())
###A tibble: 3 × 2
##Species    rows_count
##<fct>           <int>
##1 setosa           50
##2 versicolor       50
##3 virginica        50
#在 iris 数据集中计算每个品类 Sepal.Length 的最大值
iris %>% group_by(Species) %>%
summarise(Sepal.Length_max = max(Sepal.Length))
###A tibble: 3 × 2
##Species    Sepal.Length_max
##<fct>                 <dbl>
##1 setosa                5.8
##2 versicolor            7
##3 virginica             7.9
#在 iris 数据集中计算每个品类 Sepal.Length 的最大值及最小值
iris %>% group_by(Species) %>%
summarise(Sepal.Length_max = max(Sepal.Length),
          Sepal.Length_min = min(Sepal.Length))
###A tibble: 3 × 3
##Species    Sepal.Length_max Sepal.Length_min
##<fct>                 <dbl>            <dbl>
##1 setosa                5.8              4.3
##2 versicolor            7                4.9
##3 virginica             7.9              4.9
```

在上面的代码中 group_by(Species)对数据按照 Species 建立分组属性，之后通过 summarise()函数结合 n()、max()、min()函数获取指定组内的行数、指定变量 Sepal.Length 该组内的最大值及最小值。使用 group_by()将数据汇总生成新的数据集，新数据集仍旧包含了分组属性，有时在此基础上操作会受到分组属性干扰，可以使用 ungroup()把分组属性删除后再进行新操作。

在 summarise()中使用函数聚合等时，如果变量中存在 NA 值，则可能导致最终的值显示 NA。对于这种情况可以使用前面介绍的函数 drop_na()和 replace_na()等进行处理，也可以在聚合函数中设置参数 na.rm＝TRUE，如使用 sum(待汇总变量，na.rm＝TRUE)。

group_by()函数还有许多变体，如 group_split()、group_walk()等可实现不同的功能。

group_by()建立分组属性后，不仅可以使用上面的 summarise()函数汇总数据，还可以有其他灵活的运用，下面提取每组第 1 行和最后一行记录，代码如下：

```
#代码 3-36 分组抽取特殊行
library(dplyr)
library(magrittr)

#在 iris 数据集中提取每个品类的第 1 行和最后一行记录
iris %>% group_by(Species) %>% mutate(row_number = row_number()) %>%
  filter(row_number %in% c(1,n()))
###A tibble: 6 × 6
###Groups:   Species [3]
##Sepal.Length Sepal.Width Petal.Length Petal.Width Species    row_number
##<dbl>        <dbl>         <dbl>       <dbl> <fct>           <int>
#1          5.1         3.5          1.4         0.2 setosa              1
#2          5           3.3          1.4         0.2 setosa             50
#3          7           3.2          4.7         1.4 versicolor          1
#4          5.7         2.8          4.1         1.3 versicolor         50
#5          6.3         3.3          6           2.5 virginica           1
#6          5.9         3            5.1         1.8 virginica          50
```

代码 3-36 首先创建每组的行序号，之后使用 filter()结合%in%筛选第 1 行及最后一行。下面使用 slice()函数，可以不用先计算每组的行序号，操作更加简洁，代码如下：

```
#代码 3-37 使用 slice()函数分组抽取特殊行
library(dplyr)
library(magrittr)

#在 iris 数据集中提取每个品类的第 1 行和最后一行记录
iris %>% group_by(Species) %>%
  slice(c(1,n()))
###A tibble: 6 × 5
###Groups:   Species [3]
##Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##<dbl>        <dbl>         <dbl>       <dbl> <fct>
#1          5.1         3.5          1.4         0.2 setosa
#2          5           3.3          1.4         0.2 setosa
#3          7           3.2          4.7         1.4 versicolor
#4          5.7         2.8          4.1         1.3 versicolor
#5          6.3         3.3          6           2.5 virginica
#6          5.9         3            5.1         1.8 virginica
```

与 group_by()相关的几个函数还有 first()、last()、nth()。分组后如果使用 first()函数，则会选取每组指定变量的第 1 个值。last()函数用于选取指定变量的最后 1 个值。在 nth()函数中通过输入参数 n 控制选取每组中指定变量第几行中的值，代码如下：

```
#代码 3-38 first()等分组抽取特殊行
library(dplyr)
```

```
library(magrittr)

#在 iris 数据集中提取每个品类的选择变量的第 1 个值和最后一个值
iris %>% group_by(Species) %>%
  summarise(Sepal.Length_first = first(Sepal.Length),
            Sepal.Length_last = last(Sepal.Length))
##A tibble: 3 × 3
##Species    Sepal.Length_first Sepal.Length_last
##<fct>                   <dbl>             <dbl>
##1 setosa                  5.1               5
##2 versicolor              7                 5.7
##3 virginica               6.3               5.9
library(dplyr)
library(magrittr)

#在 iris 数据集中提取每个品类的选择变量的第 2 个值
iris %>% group_by(Species) %>%
  summarise(Sepal.Length_second = nth(Sepal.Length,2)
            )
##A tibble: 3 × 2
##Species    Sepal.Length_second
##<fct>                    <dbl>
##1 setosa                   4.9
##2 versicolor               6.4
##3 virginica                5.8
```

3.7.5　arrange()函数

arrange()函数可以实现对数据框进行排序，默认为以升序方式排列，结合 desc()函数可以实现降序排列，结合 match()函数可以按照已知序列的顺序进行排序，代码如下：

```
#代码 3-39 数据框排序
library(dplyr)
library(magrittr)
#为了优化页面显示内容，结果使用 head(5)展示前 5 行

#按照变量 Sepal.Length 升序排序
iris %>% arrange(Sepal.Length) %>% head(5)
##Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##1          4.3         3.0          1.1         0.1  setosa
##2          4.4         2.9          1.4         0.2  setosa
##3          4.4         3.0          1.3         0.2  setosa
##4          4.4         3.2          1.3         0.2  setosa
##5          4.5         2.3          1.3         0.3  setosa
#结合 desc 函数降序排序，按照变量 Sepal.Length 降序排序
iris %>% arrange(desc(Sepal.Length)) %>% head(5)
##Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
##1          7.9         3.8          6.4         2.0 virginica
```

```
##2          7.7         3.8          6.7         2.2 virginica
##3          7.7         2.6          6.9         2.3 virginica
##4          7.7         2.8          6.7         2.0 virginica
##5          7.7         3.0          6.1         2.3 virginica
#结合 match()函数自定义排序:按照'virginica'、'setosa'、'versicolor'的先后顺序进行排序
iris %>% arrange(match(Species,c('virginica','setosa','versicolor'))) %>% head(5)
## Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
##1          6.3         3.3          6.0         2.5 virginica
##2          5.8         2.7          5.1         1.9 virginica
##3          7.1         3.0          5.9         2.1 virginica
##4          6.3         2.9          5.6         1.8 virginica
##5          6.5         3.0          5.8         2.2 virginica
```

iris %>% arrange(Sepal.Length)表示对整个数据框按照 Sepal.Length 进行升序排列。在 arrange()函数中可以输入多个参数,表示按照参数的输入顺序对整个数据框进行排序。

3.7.6 join()函数集合

join()函数集合中包含的常用函数有 left_join()、right_join()、inner_join()。用法及原理和 SQL 中的 left join、right join、inner join 是一致的,只是表达方式不同,如将表 dt_1 和表 dt_2 拼接在一起的 SQL 语句为"select * from dt_1 left join dt_2 on dt_1.category= dt_2.category"。在 R 语言中使用 dplyr 包中的 left_join 函数的代码为"dt_1 %>% left_join(dt_2,by='category') ",代码更加简洁。

要深刻理解上述函数,需要有一定的关系型数据库基础:主键、外键、第一范式、第二范式等,没有基础的读者可以先记住规则,即使用 left_join()时右侧表中连接字段内容需唯一,使用 right_join()时左侧表中连接字段内容需要唯一,否则主表数据会出现重复数据,也就是通常所讲的笛卡儿积。在实际运用中这样的笛卡儿积大多数情况不是读者需要的,所以需要遵守上面的规则,有意为之则除外。详见下面的例子,代码如下:

```
#代码 3-40 join()函数实现数据拼接
library(dplyr)
library(magrittr)
dt_1 <- data.frame(category = c('A','B','C'),sales = c(10,30,20))
dt_2 <- data.frame(category = c('A','B'),profit = c(5,6))

#dt_1 左连接 dt_2,by = 'category'可以省略
dt_1 %>% left_join(dt_2,by = 'category')
## category sales profit
##1        A    10      5
##2        B    30      6
##3        C    20     NA
#dt_2 右连接 dt_1,by = 'category'可以省略
dt_2 %>% right_join(dt_1,by = 'category')
## category profit sales
```

```
##1         A       5     10
##2         B       6     30
##3         C      NA     20
#dt_2 内连接 dt_1,by = 'category'可以省略,只保留相匹配的行
dt_2 %>% inner_join(dt_1,by = 'category')
##category profit sales
##1         A       5     10
##2         B       6     30
merge(dt_1,dt_2,by = 'category',how = 'left')
##category sales profit
##1         A    10       5
##2         B    30       6
```

基础包中的 merge()函数也可以实现上述功能,代码如下:

```
#代码 3-41 merge()函数实现数据拼接
library(dplyr)
library(magrittr)
dt_1 <- data.frame(category = c('A','B','C'),sales = c(10,30,20))
dt_2 <- data.frame(category = c('A','B'),profit = c(5,6))

#dt_1 左连接 dt_2,by = 'category'可以省略
dt_1 %>% merge(dt_2,by = 'category',how = 'left')
##category sales profit
##1         A    10       5
##2         B    30       6
#dt_2 右连接 dt_1,by = 'category'可以省略
dt_2 %>% merge(dt_1,by = 'category',how = 'right')
##category profit sales
##1         A       5     10
##2         B       6     30
#dt_2 内连接 dt_1,by = 'category'可以省略,只保留相匹配的行
dt_2 %>% merge(dt_1,by = 'category',how = 'inner')
##category profit sales
##1         A       5     10
##2         B       6     30
```

另外,如果对数据框进行追加或对多个数据框进行简单拼接,则可以使用基础包中的函数 rbind()、cbind()函数。当使用 rbind()时需要待拼接的表列名相同,当使用 cbind()时需要待拼接的表行数相同,但是可以没有主键,只是把数据简单地横向合并到一起。

rbind()需要待拼接的两个数据框的列名相同,对于列名不完全相同的两个数据框合并生成最大集可以使用 bind_rows()。最终结果是求同存异,即多个数据框的相同列会被追加在同列,不同的列均在新的合并结果中保留,缺失的数据以 NA 填充。

同理 bind_cols()函数与 cbind()函数对应,二者的关系可类比上面的 rbind()函数与 bind_rows()函数的关系。

3.7.7 R 语言循环及判断

在任何语言的编程过程中循环和判断运用的场景都非常广泛，在 R 语言中也不例外。对于循环语句，虽然 R 语言强调向量化及函数式编程，但是在特殊场景下更加灵活、更能满足实际需要，另外，如果有其他语言的编程背景，则 R 语言中的循环语句更容易为使用者掌握。如果需要将一个文件夹下的 CSV 文件汇总，并且这些 CSV 会被频繁地更新，则由于来源繁杂导致文件字符集、格式等存在不同的可能性，使用循环编程可以快速定位到报错或者预期外的某个点，便于进一步处理。在这类情况下，比直接使用 map()或 apply()函数更容易优化调试代码。

由于关于 R 编程中循环及条件判断使用的学习资料非常丰富，本节仅介绍 for 循环语句及 if 判断语句。for 循环的基本语法为 for(变量，变量循环范围){具体代码}，if 语句的基本语法为 if(判断真或否){如果判断为真，则执行代码}。

在下面的例子里，使用循环及判断将数据框中的空缺值 NA 替换为 0，这里没有使用现成的已经封装好的函数，代码相对烦琐，主要是为读者学习循环及判断有个总体概念。首先生成一个数据框 raw_data，每个变量中均有 NA 值。使用 for 循环按照行列循环，逐个判断是否是 NA，之后将是 NA 值的部分赋值为 0。代码中 ncol()函数用于获得数据框的列数 3，1：ncol(raw_data))生成 1～3 的连续向量，等同于 c(1,2,3)。代码中的 nrow()函数用于返回数据框的行数 6，1：nrow(raw_data)用于生成 1～6 的连续向量，等同于 c(1,2,3,4,5,6)。代码运行完毕后查看 raw_data，可以发现所有 NA 被替换为 0，具体参见下面的例子，代码如下：

```
#代码 3-42 R语言循环使用1
raw_data <- data.frame(category = c('a','b','d',NA,'E',NA),
                       sub_category = c('A','C',NA,NA,NA,'D'),
                       amount = c(1,2,NA,5,NA,8))

for (i in 1:ncol(raw_data)){
  for (j in 1:nrow(raw_data)){
   if (is.na(raw_data[j,i])){
       raw_data[j,i] <- 0
     }

   }
}
raw_data
##category sub_category amount
##1        a            A      1
##2        b            C      2
##3        d            0      0
##4        0            0      5
##5        E            0      0
##6        0            D      8
```

代码 3-42 是按照传统基础方法编写的代码，下面对代码进行简化，将每个变量(每列)

当作一个向量，统一从这个向量选出是 NA 的部分并赋值为 0，代码如下：

```
#代码 3-43 R语言循环使用2
raw_data <- data.frame(category = c('a','b','d',NA,'E',NA),
                       sub_category = c('A','C',NA,NA,NA,'D'),
                       amount = c(1,2,NA,5,NA,8))

for (i in 1:ncol(raw_data)){
 raw_data[,i][is.na(raw_data[,i])] <- 0
}
raw_data
##category sub_category amount
##1        a            A      1
##2        b            C      2
##3        d            0      0
##4        0            0      5
##5        E            0      0
##6        0            D      8
```

raw_data[，i][is.na(raw_data[，i])]的逻辑可以理解为数据框[，第 i 列][判断条件]，如 i 为 1 本例中返回 category 中的错误值，即 2 个 NA(实际还包含 NA 所在的位置等信息)，读者可以单独运行该部分代码。

R 语言中强调向量化编程、函数式编程。下面使用 mutate()函数结合 across()函数等去除数据框中的 NA 值，across()中的第 1 个函数 everything()表示选取数据框中的所有列，第 2 个参数表示对选取的列执行 str_replace_na()函数。整个过程比循环更加简洁、高效，代码如下：

```
#代码 3-44 R语言向量化编程
raw_data <- data.frame(category = c('a','b','d',NA,'E',NA),
                       sub_category = c('A','C',NA,NA,NA,'D'),
                       amount = c(1,2,NA,5,NA,8))


raw_data %>% mutate(across(everything(),~ stringr::str_replace_na(.,0)))
##category sub_category amount
##1        a            A      1
##2        b            C      2
##3        d            0      0
##4        0            0      5
##5        E            0      0
##6        0            D      8
```

3.8　map()函数群

purrr 包中的 map_*()函数群可以替代循环，实现将函数或功能运用到原始数据的每个组中。map()函数输入的参数必须是列表 list 或者向量 vector。下面的例子用于生成 5 组

随机正态数，每组均含 20 个值，每组均值分别为 1,2,3,4,5。若用循环调用 rnorm(20,每组均值)5 次即可得到结果。下面使用 map()函数来完成，map()函数将 function(x) rnorm(20,x)视作一个函数，分别运用到 c(1,2,3,4,5)中的每个值，并将每个值视作 rnorm(20,x)中的均值 x，代码如下：

```
#代码 3-45 map()函数
library(magrittr)
library(purrr)
1:5 %>% map(function(x) rnorm(20,x))
##[[1]]
##[1]  0.2572601   0.1852584   0.9165663  2.0523200   0.1045350   2.7092042
##[7]  0.5332354   1.0469657   0.7356070  1.7118330   0.7720200   1.2340902
##[13] 0.5276446   1.2614835 -0.7773102  0.3052802   0.8429647 -1.5871845
##[19] 1.7093577 -0.9988739
##
##[[2]]
##[1]  1.7169736   4.1085748   1.8441503  2.8377471 -0.8253852   2.9829919
##[7]  1.5629016   1.7508358   2.5541466  2.9860964   2.1310005   2.0358309
##[13] 2.7629535   1.6225803   1.8191693  1.3105945   2.5088678   2.0430187
##[19] 0.2738276   1.7170434
##
##[[3]]
##[1]  3.796902 3.607153 2.872400 1.982365 4.056843 4.253985 3.290552 4.775438
##[9]  1.963672 2.505948 1.485144 2.297968 2.345866 2.664615 2.918104 1.409204
##[17] 2.740322 2.848264 2.034280 1.399345
##
##[[4]]
##[1]  3.609766 3.941724 5.204866 4.556049 1.911595 3.187335 4.576478 4.894244
##[9]  3.350930 5.343355 2.823849 4.112550 3.115842 3.213386 3.088823 3.455909
##[17] 5.042646 3.650304 4.833275 3.791341
##
##[[5]]
##[1]  4.429534 5.692255 4.224121 5.324190 5.906389 3.232494 7.270667 6.350881
##[9]  5.571604 3.788809 5.808625 6.998314 5.603432 5.888048 4.880133 4.799995
##[17] 4.731004 6.242625 5.547776 3.558754
```

上面代码返回的结果是 list 结构，如果希望以数据框的形式存储结果，则可使用 map_df()函数。由于数据框需要变量名称，因此需要原来的向量有名称，下面的代码先生成向量 my_seq，然后通过 names()函数给每个值赋一个名称，代码如下：

```
#代码 3-46 map_df()函数 1
library(magrittr)
library(purrr)
my_seq <- 1:5
names(my_seq) <- paste0("median_",1:5)
my_seq %>% map_df(function(x) rnorm(20,x))
###A tibble: 20 × 5
```

```
##median_1 median_2 median_3 median_4 median_5
##<dbl>    <dbl>    <dbl>    <dbl>    <dbl>
##1   -0.243     1.34     3.09     4.57     4.63
##2    2.79      3.18     2.33     2.96     5.46
##3    1.90      3.35     2.54     4.55     5.93
##4    2.52      0.348    4.32     3.53     3.96
##5    1.37      3.78     2.78     3.64     4.17
##6    0.0531    4.08     2.53     3.85     6.53
##7   -0.463     2.25     2.61     4.65     6.84
##8    0.809     1.99     3.94     4.11     5.33
##9   -0.0589    2.64     2.07     3.46     5.65
##10   2.14      1.79     4.78     3.79     5.63
##11   1.72      2.48     3.27     4.96     4.01
##12   0.538     1.67     4.44     3.59     4.51
##13   1.31      1.93     3.49     5.31     4.82
##14   2.19      0.789    0.334    3.55     5.30
##15  -0.0915   -0.0491   3.28     4.29     4.66
##16   1.63      0.789    2.78     2.57     5.49
##17   1.98      1.24     3.23     3.85     5.52
##18   2.77      1.88     3.05     3.76     4.18
##19   0.871     1.98     3.20     3.26     7.16
##20   0.720     1.46     2.79     3.55     4.80
```

在3.7.7节，通过循环判断替换了其中的NA值，下面通过map_df()函数来完成。原始数据是数据框，通过as.list()转变为list，之后传递给map_df()函数。map_df()中使用了stringr包中的str_replace_na()函数，str_replace_na()函数会先判断第1个参数中是否是NA值，如果是NA值，则替换为0值，代码如下：

```
#代码 3-47 map_df()函数 2
library(magrittr)
library(purrr)
raw_data <- data.frame(category = c('a','b','d',NA,'E',NA),
                       sub_category = c('A','C',NA,NA,NA,'D'),
                       amount = c(1,2,NA,5,NA,8))

raw_data %>% as.list() %>% map_df(function(x) stringr::str_replace_na(x,0))
###A tibble: 6 × 3
##category sub_category amount
##<chr>    <chr>        <chr>
##1 a        A            1
##2 b        C            2
##3 d        0            0
##4 0        0            5
##5 E        0            0
##6 0        D            8
```

基础包中的apply()函数群的功能与此类似。map_*()函数群还包含其他丰富的函数，即只有星号部分不同，但笔者认为上述两个函数最为常用。