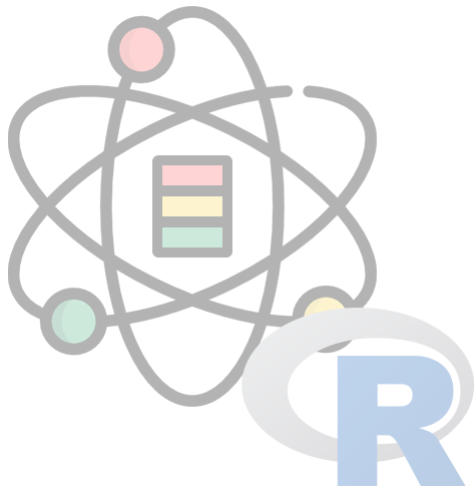
R 语言数据科学导论

Data Science Introduction with R

数据分析基础 (下)

Data Analytics Introduction - Part 2

范叶亮



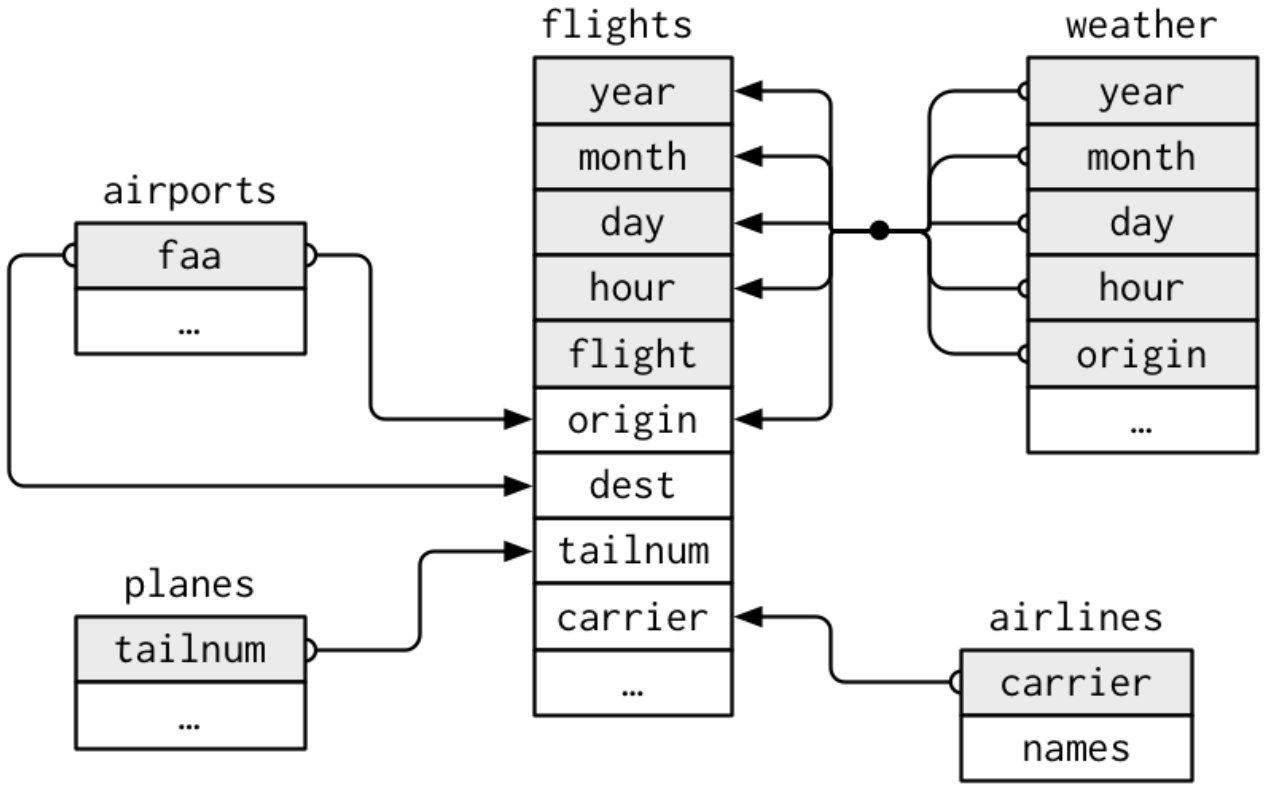
目录

- 关系数据处理
- 不同类型数据处理
- 函数式编程

关系数据处理

关系数据处理

下面将使用 nycflights13 包来介绍关系数据处理。nycflights13 中包含了与 `flights` 相关的 4 组数据，关系如下图所示：



关系数据处理

用于连接每对数据表的变量称为**键**。键是能唯一标识观测的变量（或变量集合）。简单情况下，单个变量就足以标识一个观测。例如，每架飞机都可以由 `tailnum` 唯一标识。其他情况可能需要多个变量。例如，要想标识 `weather` 中的观测，你需要 5 个变量：`year`、`month`、`day`、`hour` 和 `origin`。键的类型有两种：

- **主键**：唯一标识其所在数据表中的观测。例如，`planes$tailnum` 是一个主键，因为其可以唯一标识 `planes` 表中的每架飞机。
- **外键**：唯一标识另一个数据表中的观测。例如，`flights$tailnum` 是一个外键，因为其出现在 `flights` 表中，并可以将每次航班与唯一一架飞机匹配。

一个变量既可以是主键，也可以是外键。例如，`origin` 是 `weather` 表主键的一部分，同时也是 `airports` 表的外键。

主键与另一张表中与之对应的外键可以构成**关系**。关系通常是一对多的。例如，每个航班只有一架飞机，但每架飞机可以飞多个航班。在另一些数据中，你有时还会遇到一对一的关系。你可以将这种关系看作一对多关系的特殊情况。你可以使用多对一关系加上一对多关系来构造多对多关系。例如，在这份数据中，航空公司与机场之间存在着多对多关系：每个航空公司可以使用多个机场，每个机场可以服务多个航空公司。

合并连接

合并连接可以将两个表格中的变量组合起来，它先通过两个表格的键匹配观测，然后将一个表格中的变量复制到另一个表格中。和 mutate() 函数一样，连接函数也会将变量添加在表格的右侧，因此如果表格中已经有了很多变量，那么新变量就不会显示出来。

为了帮助掌握连接的工作原理，在此介绍用图形来表示连接的一种方法。

有颜色的列表示作为“键”的变量：它们用于在表间匹配行。灰色列表示“值”列，是与键对应的值。

```
x ← tribble(
  ~key, ~val_x,
     1, "x1",
     2, "x2",
     3, "x3"
)

y ← tribble(
  ~key, ~val_y,
     1, "y1",
     2, "y2",
     4, "y3"
)
```

x

1	x1
2	x2
3	x3

y

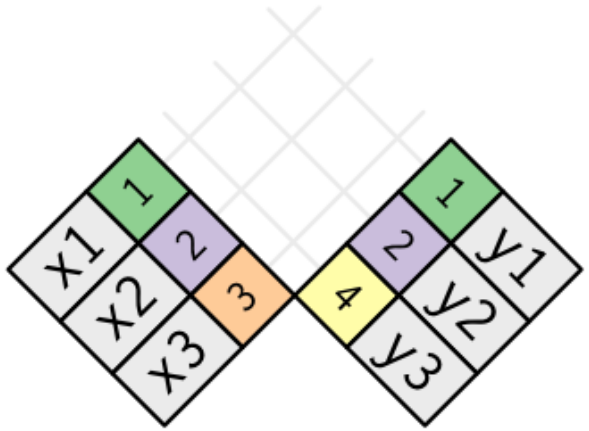
1	y1
2	y2
4	y3

合并连接

在以下的示例中，虽然键和值都是一个变量，但非常容易推广到多个键变量和多个值变量的情况。

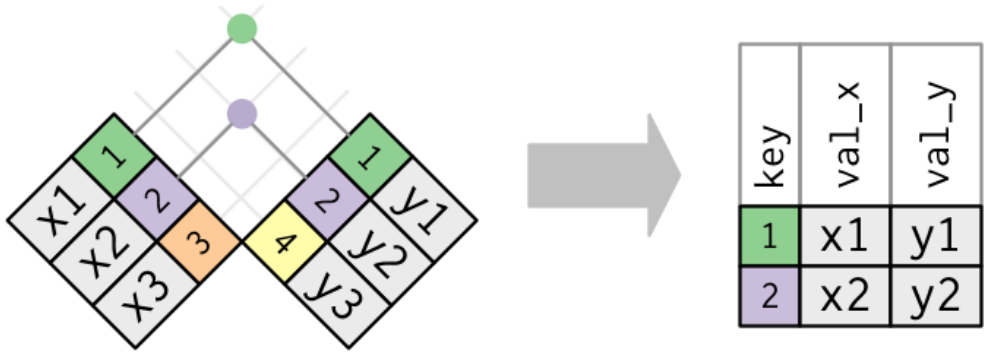
连接是将 x 中每行连接到 y 中 0 行、一行或多行的一种方法。

右图表示出了所有可能的匹配，匹配就是两行之间的交集。



内连接

内连接是最简单的一种连接。只要两个观测的键是相等的，内连接就可以匹配它们。



内连接最重要的性质是，没有匹配的行不会包含在结果中。这意味着内连接一般不适合在分析中使用，因为太容易丢失观测了。

内连接的结果是一个新数据框，其中包含键、x 值和 y 值。使用 by 参数告诉 dplyr 哪个变量是键：

```
x |>
  inner_join(y, by = "key")
```

```
## # A tibble: 2 × 3
##     key val_x val_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2
```

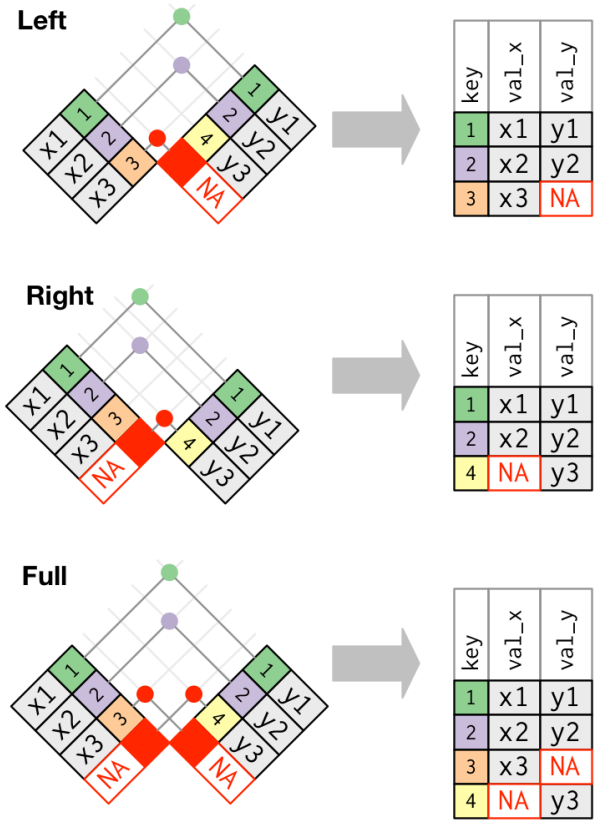
外连接

内连接保留同时存在于两个表中的观测，**外连接**则保留至少存在于一个表中的观测。外连接有 3 种类型。

- **左连接**：保留 x 中的所有观测
- **右连接**：保留 y 中的所有观测
- **全连接**：保留 x 和 y 中的所有观测

这些连接会向每个表中添加额外的“虚拟”观测，这个观测拥有总是匹配的键（如果没有其他键可匹配的话），其值则用 NA 来填充。

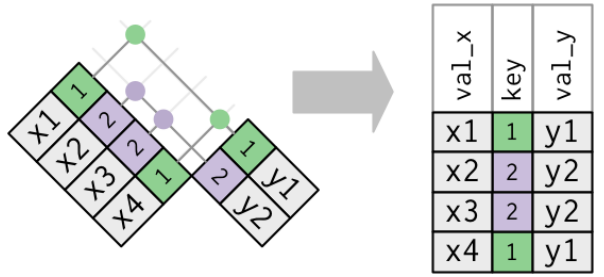
最常用的连接是左连接：只要想从另一张表中添加数据，就可以使用左连接，因为它会保留原表中的所有观测，即使它没有匹配。



重复键

至今为止，所有图都假设键具有唯一性，但情况并非总是如此。

一张表中具有重复键。通常来说，当存在一对多关系时，如果向表中添加额外信息，就会出现这种情况。




```
x <- tribble(
  ~key, ~val_x,
     1, "x1",
     2, "x2",
     2, "x3",
     1, "x4"
)
```

```
y <- tribble(
  ~key, ~val_y,
     1, "y1",
     2, "y2"
)
```

```
left_join(x, y, by = "key")
```

```
## # A tibble: 4 × 3
##     key val_x val_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2
## 3     2 x3    y2
## 4     1 x4    y1
```

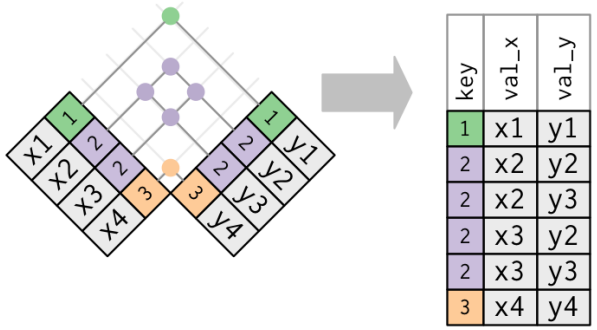
重复键

两张表中都有重复键。这通常意味着出现了错误，因为键在任意一张表中都不能唯一标识观测。

当连接这样的重复键时，你会得到所有可能的组合，即笛卡儿积：

```
x ← tribble(
  ~key, ~val_x,
     1, "x1",
     2, "x2",
     2, "x3",
     3, "x4"
)
```

```
y ← tribble(
  ~key, ~val_y,
     1, "y1",
     2, "y2",
     2, "y3",
     3, "y4"
)
```




```
left_join(x, y, by = "key")
```

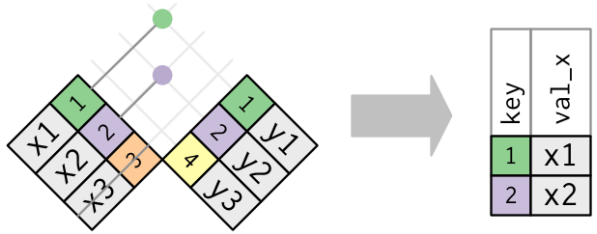
```
## # A tibble: 6 × 3
##     key val_x val_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2
## 3     2 x2    y3
## 4     2 x3    y2
## 5     2 x3    y3
## 6     3 x4    y4
```

筛选链接

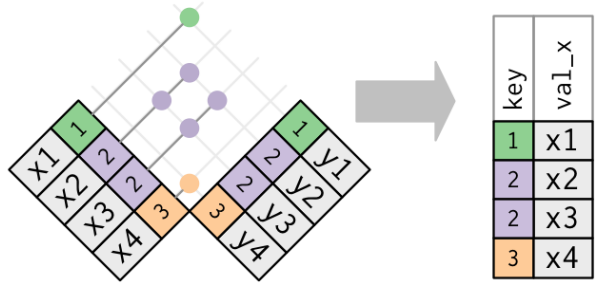
筛选连接匹配观测的方式与合并连接相同，但前者影响的是观测，而不是变量。筛选连接有两种类型。

- `semi_join(x, y)`：保留 x 表中与 y 表中的观测相匹配的所有观测。
- `anti_join(x, y)`：丢弃 x 表中与 y 表中的观测相匹配的所有观测。

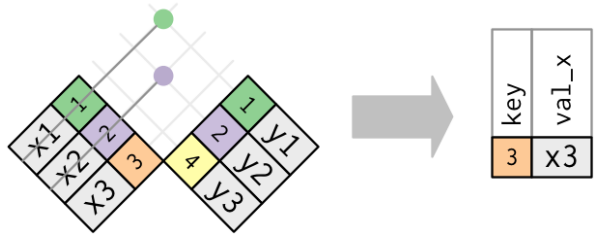
半连接的图形表示如下所示。



重要的是存在匹配，匹配了哪条观测则无关紧要。这说明筛选连接不会像合并连接那样造成重复的行。



半连接的逆操作是反连接。反连接保留 x 表中那些没有匹配 y 表的行。



不同类型数据处理

字符数据处理

处理字符串会使用到 stringr 包。

```
library(stringr)
```

str_length() 函数可以返回字符串中的字符数量：

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

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

要想组合两个或更多字符串，可以使用 str_c() 函数：

```
str_c("x", "y", "z")
```

```
## [1] "xyz"
```

可以使用 sep 参数来控制字符串间的分隔方式：

```
str_c("x", "y", sep = ", ")
```

```
## [1] "x, y"
```

str_c() 函数是向量化的，它可以自动循环短向量，使得其与最长的向量具有相同的长度：

```
str_c("p-", c("a", "b", "c"), "-s")
```

```
## [1] "p-a-s" "p-b-s" "p-c-s"
```

将字符向量合并为字符串，可以使用 collapse() 函数：

```
str_c(c("x", "y", "z"), collapse = ", ")
```

```
## [1] "x, y, z"
```

字符数据处理

可以使用 str_sub() 函数来提取字符串的一部分。

```
x ← c("Apple", "Banana", "Pear")
str_sub(x, 1, 3)
```

```
## [1] "App" "Ban" "Pea"
```

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

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

注意，即使字符串过短，str_sub() 函数也不会出错，它将返回尽可能多的字符：

```
str_sub("a", 1, 5)
```

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

还可以使用 str_sub() 函数的赋值形式来修改字符串：

```
str_sub(x, 1, 1) ← str_to_lower(str_sub(x, 1, 1))
x
```

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

字符数据处理

利用 str_to_lower() 函数可以将文本转换为小写，利用 str_to_upper() 函数可以将文本转换为大写，利用 str_to_title() 函数可以将文本转换为首字母大写。

大小写转换要比你想象的更复杂，因为不同的语言有不同的转换规则。你可以通过明确区域设置来选择使用哪种规则：

```
str_to_upper(c("i", "ı"))
```

```
## [1] "I" "I"
```

```
str_to_upper(c("i", "ı"), locale = "tr")
```

```
## [1] "İ" "I"
```

使用 str_sort() 和 str_order() 函数可以对字符串进行排序，它们可以使用 locale 参数来进行区域设置：

```
x ← c("apple", "eggplant", "banana")

str_sort(x, locale = "en")
```

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

```
str_sort(x, locale = "haw")
```

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

字符数据处理

要想确定一个字符向量能否匹配一种模式，可以使用 str_detect() 函数。它返回一个与输入向量具有同样长度的逻辑向量：

```
x ← c("apple", "banana", "pear")
str_detect(x, "e")
```

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

str_detect() 函数的一种常见用法是选取出匹配某种模式的元素。你可以通过逻辑取子集方式来完成这种操作，也可以使用便捷的 str_subset() 包装器函数：

```
words[str_detect(words, "x$")]
```

```
## [1] "box" "sex" "six" "tax"
```

```
str_subset(words, "x$")
```

```
## [1] "box" "sex" "six" "tax"
```

str_detect() 函数的一种变体是 str_count()，后者不是简单地返回是或否，而是返回字符串中匹配的数量：

```
x ← c("apple", "banana", "pear")
str_count(x, "a")
```

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

注意，匹配从来不会重叠。例如，在 abababa 中，模式 aba 会匹配 2 次，而不是 3 次。

字符数据处理

要想提取匹配的实际文本，我们可以使用 `str_extract()` 函数。我们将使用维基百科上的 Harvard sentences 数据集进行测试。

```
length(sentences)
```

```
## [1] 720
```

```
head(sentences)
```

```
## [1] "The birch canoe slid on the smooth planks."
## [2] "Glue the sheet to the dark blue background."
## [3] "It's easy to tell the depth of a well."
## [4] "These days a chicken leg is a rare dish."
## [5] "Rice is often served in round bowls."
## [6] "The juice of lemons makes fine punch."
```

假设我们想要找出包含一种颜色的所有句子。首先，我们需要创建一个颜色名称向量，然后将其转换成一个正则表达式：

```
colours <- c(
  "red", "orange", "yellow",
  "green", "blue", "purple")
colour_match <- str_c(colours, collapse = "|")
colour_match
```

```
## [1] "red|orange|yellow|green|blue|purple"
```

字符数据处理

现在我们可以选取出包含一种颜色的句子，再从中提取出颜色，就可以知道有哪些颜色了：

```
has_colour ← str_subset(sentences, colour_match)
matches ← str_extract(has_colour, colour_match)
head(matches)
```

```
## [1] "blue" "blue" "red"  "red"  "red"  "blue"
```

注意，`str_extract()` 只提取第一个匹配。这是 stringr 函数的一种通用模式，因为单个匹配可以使用更简单的数据结构。要想得到所有匹配，可以使用 `str_extract_all()` 函数，它会返回一个列表：

```
more ← sentences[
  str_count(sentences, colour_match) > 1]
str_extract_all(more, colour_match)
```

```
## [[1]]
## [1] "blue" "red"
## [[2]]
## [1] "green" "red"
## [[3]]
## [1] "orange" "red"
```

如果设置了 `simplify = TRUE`，那么 `str_extract_all()` 会返回一个矩阵，其中较短的匹配会扩展到与最长的匹配具有同样的长度。

字符数据处理

str_replace() 和 str_replace_all() 函数可以使用新字符串替换匹配内容。最简单的应用是使用固定字符串替换匹配内容：

```
x ← c("apple", "pear", "banana")
str_replace(x, "[aeiou]", "-")
```

```
## [1] "-pple"  "p-ar"   "b-nana"
```

```
str_replace_all(x, "[aeiou]", "-")
```

```
## [1] "-ppl-"  "p--r"   "b-n-n-"
```

通过提供一个命名向量，使用 str_replace_all() 函数可以同时执行多个替换：

```
x ← c("1 house", "2 cars", "3 people")

str_replace_all(
  x, c("1" = "one", "2" = "two", "3" = "three"))
```

```
## [1] "one house"    "two cars"     "three people"
```

字符数据处理

`str_split()` 函数可以将字符串拆分为多个片段。例如，我们可以将句子拆分成单词：

```
sentences |>
  head(3) |>
  str_split(" ")
```

```
## [[1]]
## [1] "The"     "birch"   "canoe"   "slid"    "on"
"the"     "smooth"
## [8] "planks."
## [[2]]
## [1] "Glue"        "the"         "sheet"       "to"
"the"
## [6] "dark"        "blue"        "background."
## [[3]]
## [1] "It's"  "easy"  "to"    "tell"  "the"   "depth"
"of"    "a"     "well."
```

返回列表的其他 stringr 函数一样，你可以通过设置 `simplify = TRUE` 返回一个矩阵：

```
sentences |>
  head(3) |>
  str_split(" ", simplify = TRUE)
```

```
##      [,1]   [,2]    [,3]    [,4]   [,5]  [,6]    [,
7]     [,8]          [,9]
## [1,] "The"  "birch" "canoe" "slid" "on"  "the"   "s
mooth" "planks."     ""
## [2,] "Glue" "the"   "sheet" "to"   "the" "dark"  "b
lue"   "background." ""
## [3,] "It's" "easy"  "to"    "tell" "the" "depth" "o
f"     "a"           "well."
```

因子数据处理

使用 forcats 包可以处理因子，这个包提供了能够处理分类变量的工具，还包括了处理因子的大量辅助函数。

```
library(forcats)
```

假设我们想要创建一个记录月份的变量：

```
x1 ← c("Dec", "Apr", "Jan", "Mar")
```

使用字符串来记录月份有两个问题。

- 月份只有 12 个取值，如果输入错误，那么代码不会有任何反应。
- 对月份的排序没有意义。

通过使用因子来解决以上两个问题。要想创建一个因子，必须先创建有效**水平**的一个列表：

```
month_levels ← c(
  "Jan", "Feb", "Mar", "Apr", "May", "Jun",
  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
)
```

现在你可以创建因子了：

```
y1 ← factor(x1, levels = month_levels)
y1
```

```
## [1] Dec Apr Jan Mar
## Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov
Dec
```

因子数据处理

```
sort(y1)
```

```
## [1] Jan Mar Apr Dec
## Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov
Dec
```

不在有效水平集合内的所有值都会自动转换为 NA：

```
x2 ← c("Dec", "Apr", "Jam", "Mar")
y2 ← factor(x2, levels = month_levels)
y2
```

```
## [1] Dec  Apr  <NA> Mar
## Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov
Dec
```

如果想要显示错误信息，那么你可以使用 `readr::parse_factor()` 函数：

```
y2 ← parse_factor(x2, levels = month_levels)
```

如果省略了定义水平的这个步骤，那么会将按字母顺序排序的数据作为水平：

```
factor(x1)
```

```
## [1] Dec Apr Jan Mar
## Levels: Apr Dec Jan Mar
```

因子数据处理

有时你会想让因子的顺序与初始数据的顺序保持一致。在创建因子时，将水平设置为 `unique(x)`，或者在创建因子后再对其使用 `fct_inorder()` 函数，就可以达到这个目的：

```
f1 ← factor(x1, levels = unique(x1))
f1
```

```
## [1] Dec Apr Jan Mar
## Levels: Dec Apr Jan Mar
```

```
f2 ← x1 |> factor() |> fct_inorder()
f2
```

```
## [1] Dec Apr Jan Mar
## Levels: Dec Apr Jan Mar
```

如果想要直接访问因子的有效水平集合，那么可以使用 `levels()` 函数：

```
levels(f2)
```

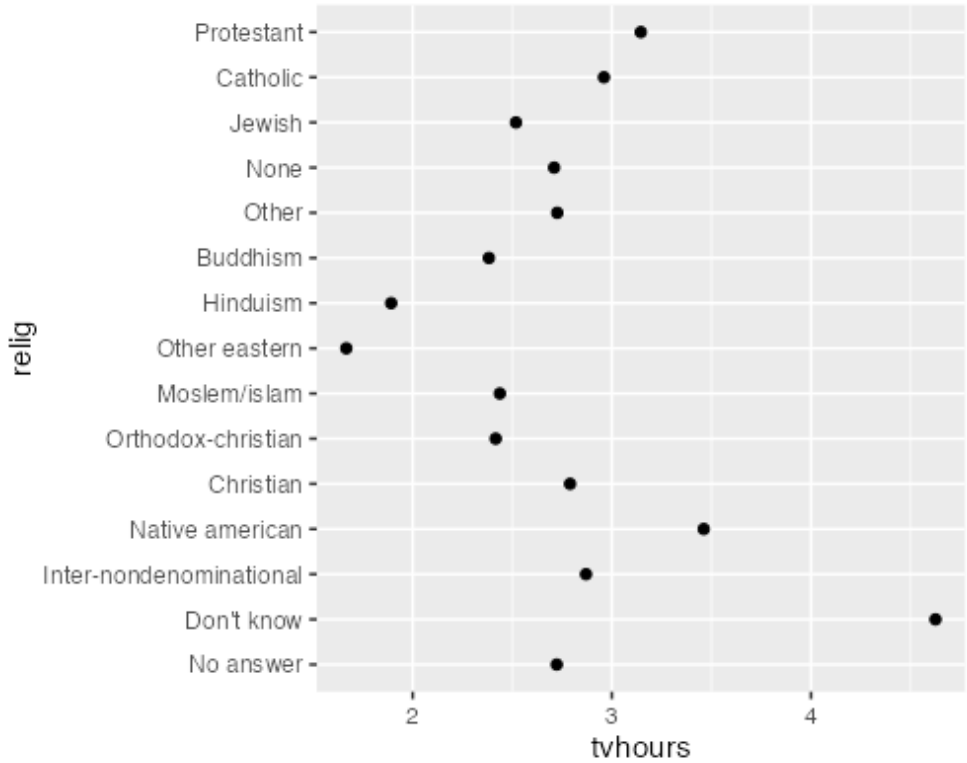
```
## [1] "Dec" "Apr" "Jan" "Mar"
```

因子数据处理

在可视化中修改因子水平的顺序很有用。例如：假设需要探索不同民族的人每天平均看电视的时间：

```
relig_summary <- gss_cat |>
  group_by(relig) |>
  summarise(
    age = mean(age, na.rm = TRUE),
    tvhours = mean(tvhours, na.rm = TRUE),
    n = n()
  )
```

```
ggplot(relig_summary, aes(tvhours, relig)) +
  geom_point()
```

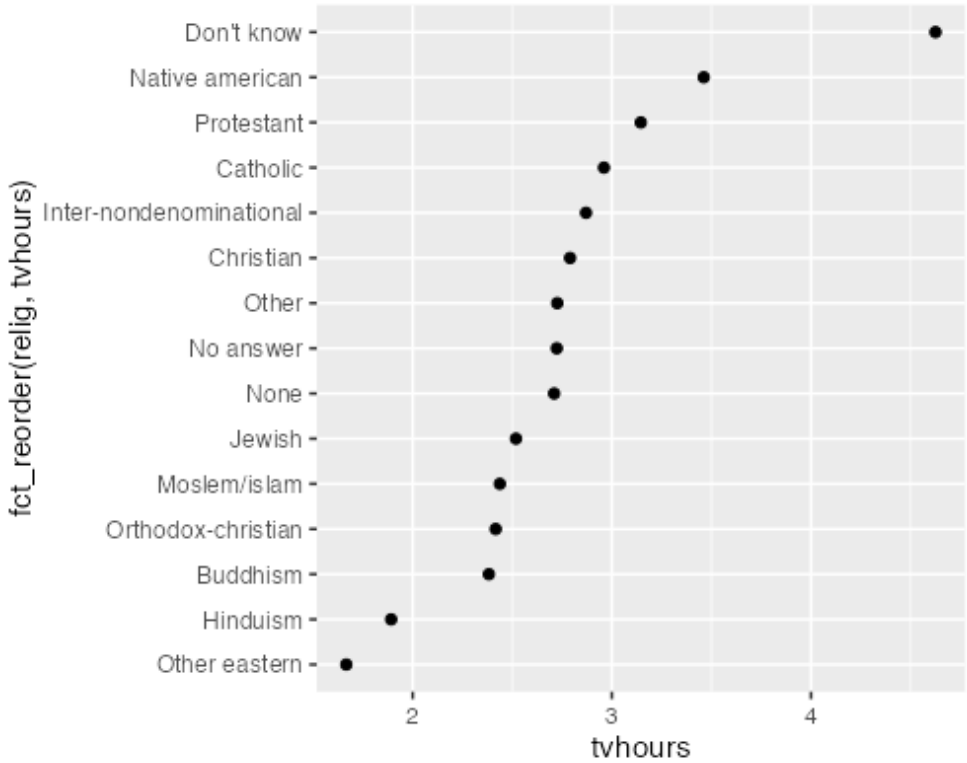


因子数据处理

由于没有整体的模式，因此很难解释上图。通过 `fct_reorder()` 可以重新对民族的因子水平进行重排。`fct_reorder()` 接受 3 个参数：

1. `f`：需要重排的因子
2. `x`：用于重排的数值向量
3. `fun`：可选参数，当对于 `f` 有多个 `x` 值时，所采用的计算方式，默认为 `median`

```
ggplot(relig_summary,
  aes(tvhours, fct_reorder(relig, tvhours))) +
  geom_point()
```



因子数据处理

比修改因子水平顺序更强大的操作是修改水平的值。修改水平最常用、最强大的工具是 fct_recode() 函数，它可以对每个水平进行修改或重新编码。

```
gss_cat |> count(partyid) |> tail(3)
```

```
## # A tibble: 3 × 2
##   partyid              n
##   <fct>            <int>
## 1 Ind,near dem      2499
## 2 Not str democrat  3690
## 3 Strong democrat   3490
```

对水平的描述太过简单，而且不一致。我们将其修改为较为详细的排比结构：

```
gss_cat |>
  mutate(partyid = fct_recode(partyid,
    "Republican, strong"    = "Strong republican",
    "Republican, weak"      = "Not str republican",
    "Independent, near rep" = "Ind,near rep",
    "Independent, near dem" = "Ind,near dem",
    "Democrat, weak"        = "Not str democrat",
    "Democrat, strong"      = "Strong democrat"
  )) |> count(partyid) |> tail(3)
```

```
## # A tibble: 3 × 2
##   partyid                    n
##   <fct>                  <int>
## 1 Independent, near dem   2499
## 2 Democrat, weak          3690
## 3 Democrat, strong        3490
```

因子数据处理

`fct_recode()` 会让没有明确提及的水平保持原样，如果不小心修改了一个不存在的水平，那么它也会给出警告。

你可以将多个原水平赋给同一个新水平，这样就可以合并原来的分类。使用这种操作时一定要小心：如果合并了原本不同的分类，那么就会产生误导性的结果。

```
gss_cat |>
  mutate(partyid = fct_recode(partyid,
    "Republican, strong"    = "Strong republican",
    "Republican, weak"      = "Not str republican",
    "Independent, near rep" = "Ind,near rep",
    "Independent, near dem" = "Ind,near dem",
    "Democrat, weak"        = "Not str democrat",
    "Democrat, strong"      = "Strong democrat",
    "Other"                 = "No answer",
    "Other"                 = "Don't know",
    "Other"                 = "Other party"
  )) |> count(partyid)
```

```
## # A tibble: 8 × 2
##   partyid                    n
##   <fct>                  <int>
## 1 Other                    548
## 2 Republican, strong      2314
## 3 Republican, weak        3032
## 4 Independent, near rep   1791
## 5 Independent             4119
## 6 Independent, near dem   2499
## # i 2 more rows
```

因子数据处理

如果想要合并多个水平，那么可以使用 fct_recode() 函数的变体 fct_collapse() 函数。对于每个新水平，你都可以提供一个包含原水平的向量：

```
gss_cat |>
  mutate(partyid = fct_collapse(partyid,
    other = c("No answer", "Don't know",
              "Other party"),
    rep = c("Strong republican",
            "Not str republican"),
    ind = c("Ind,near rep", "Independent",
            "Ind,near dem"),
    dem = c("Not str democrat",
            "Strong democrat")
  )) |> count(partyid)
```

```
## # A tibble: 4 × 2
##   partyid     n
##   <fct>   <int>
## 1 other     548
## 2 rep      5346
## 3 ind      8409
## 4 dem      7180
```

日期时间数据处理

使用 lubridate 包可以处理日期和时间，它可以使得 R 对日期和时间的处理更加容易。

```
library(lubridate)
```

通过确定年、月和日在日期数据中的顺序，然后按照同样的顺序排列 y、m 和 d，这样就可以组成能够解析日期的 lubridate 函数名称。

```
ymd("2017-01-31")
```

```
## [1] "2017-01-31"
```

```
mdy("January 31st, 2017")
```

```
## [1] "2017-01-31"
```

```
dmy("31-Jan-2017")
```

```
## [1] "2017-01-31"
```

这些函数也可以接受不带引号的数值。

```
ymd(20170131)
```

```
## [1] "2017-01-31"
```

日期时间数据处理

ymd() 和类似的其他函数可以创建日期。要想创建日期时间型数据，可以在后面加一个下划线，以及 h、m 和 s 之中的一个或多个字母，这样就可以得到解析日期时间的函数了：

```
ymd_hms("2017-01-31 20:11:59")
```

```
## [1] "2017-01-31 20:11:59 UTC"
```

```
mdy_hm("01/31/2017 08:01")
```

```
## [1] "2017-01-31 08:01:00 UTC"
```

通过添加一个时区参数，你可以将一个日期强制转换为日期时间：

```
ymd(20170131, tz = "UTC")
```

```
## [1] "2017-01-31 UTC"
```

```
ymd_hms("2017-01-31 20:11:59", tz = "Asia/Shanghai")
```

```
## [1] "2017-01-31 20:11:59 CST"
```

日期时间数据处理

除了单个字符串，日期时间数据的各个成分还经常分布在表格的多个列中。如果想要按照这种表示方法来创建日期或时间，可以使用 make_date() 函数创建日期，使用 make_datetime() 函数创建日期时间：

```
library(nycflights13))

flights |>
  select(year, month, day, hour, minute) |>
  mutate(departure = make_datetime(
    year, month, day, hour, minute))
```

```
## # A tibble: 336,776 × 6
##    year month   day  hour minute departure
##   <int> <int> <int> <dbl>  <dbl> <dttm>
## 1  2013     1     1     5     15 2013-01-01 05:15:0
0
## 2  2013     1     1     5     29 2013-01-01 05:29:0
0
## 3  2013     1     1     5     40 2013-01-01 05:40:0
0
## 4  2013     1     1     5     45 2013-01-01 05:45:0
0
## 5  2013     1     1     6      0 2013-01-01 06:00:0
0
## 6  2013     1     1     5     58 2013-01-01 05:58:0
0
## # i 336,770 more rows
```

日期时间数据处理

有时你需要在日期时间型数据和日期型数据之间进行转换，这正是 as_datetime() 和 as_date() 函数的功能：

```
as_datetime(today())
```

```
## [1] "2024-02-11 UTC"
```

```
as_date(now())
```

```
## [1] "2024-02-11"
```

如果想要提取出日期中的独立成分，可以使用以下访问器函数：year()、month()、mday()（一个月中的第几天）、yday()（一年中的第几天）、wday()（一周中的第几天）、hour()、minute() 和 second()：

```
datetime ← ymd_hms("2016-07-08 12:34:56")

year(datetime)
```

```
## [1] 2016
```

```
month(datetime)
```

```
## [1] 7
```

```
mday(datetime)
```

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

```
yday(datetime)
```

```
## [1] 190
```

日期时间数据处理

对于 month() 和 wday() 函数，你可以设置 label = TRUE 来返回月份名称和星期数的缩写，还可以设置 abbr = FALSE 来返回全名：

```
month(datetime, label = TRUE)
```

```
## [1] Jul
## 12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul
< Aug < Sep < ... < Dec
```

```
wday(datetime, label = TRUE, abbr = FALSE)
```

```
## [1] Friday
## 7 Levels: Sunday < Monday < Tuesday < Wednesday < T
hursday < ... < Saturday
```

通过 floor_date()、round_date() 和 ceiling_date() 函数将日期舍入到临近的一个时间单位。函数的参数都包括一个待调整的时间向量，以及时间单位名称，函数会将这个向量舍下、入上或四舍五入到这个时间单位。

```
floor_date(datetime, "week")
```

```
## [1] "2016-07-03 UTC"
```

通过 update() 函数创建一个新日期时间。这样也可以同时设置多个成分：

```
update(datetime, year = 2020, month = 2, mday = 2, h
our = 2)
```

```
## [1] "2020-02-02 02:34:56 UTC"
```

日期时间数据处理

在 R 中，如果将两个日期相减，那么你会得到不同的对象：

```
l_age <- today() - ymd(19910109)
l_age
```

```
## Time difference of 12086 days
```

表示时间差别的对象记录时间间隔的单位可以是秒、分钟、小时、日或周。因为这种模棱两可的对象处理起来非常困难，所以 lubridate 提供了总是使用秒为单位的另一种计时对象 - **时期**：

```
as.duration(l_age)
```

```
## [1] "1044230400s (~33.09 years)"
```

可以使用很多方便的构造函数来创建时期：

```
dseconds(15)
```

```
## [1] "15s"
```

```
dminutes(10)
```

```
## [1] "600s (~10 minutes)"
```

```
dhours(c(12, 24))
```

```
## [1] "43200s (~12 hours)" "86400s (~1 days)"
```

```
dweeks(3)
```

```
## [1] "1814400s (~3 weeks)"
```

日期时间数据处理

时期总是以秒为单位来记录时间间隔。使用标准比率（1 分钟为 60 秒、1 小时为 60 分钟、1 天为 24 小时、1 周为 7 天、1 年为 365 天）将分钟、小时、日、周和年转换为秒，从而建立具有更大值的对象。

可以对时期进行加法和乘法操作：

```
2 * dyears(1)
```

```
## [1] "63115200s (~2 years)"
```

```
dyears(1) + dweeks(12) + dhours(15)
```

```
## [1] "38869200s (~1.23 years)"
```

然而，因为时期表示的是以秒为单位的一段精确时间，所以有时你会得到意想不到的结果：

```
one_pm ← ymd_hms(
  "2016-03-12 13:00:00",
  tz = "America/New_York")

one_pm
```

```
## [1] "2016-03-12 13:00:00 EST"
```

```
one_pm + ddays(1)
```

```
## [1] "2016-03-13 14:00:00 EDT"
```

因为夏令时，3 月 12 日只有 23 个小时，因此如果我们加上一整天的秒数，那么就会得到一个不正确的时间。

日期时间数据处理

阶段也是一种时间间隔，但它不以秒为单位；相反，它使用“人工”时间，比如日和月。这使得它们使用起来更加直观：

```
one_pm
```

```
## [1] "2016-03-12 13:00:00 EST"
```

```
one_pm + days(1)
```

```
## [1] "2016-03-13 13:00:00 EDT"
```

可以对阶段进行加法和乘法操作：

```
10 * (months(6) + days(1))
```

```
## [1] "60m 10d 0H 0M 0S"
```

当然，阶段可以和日期相加。与时期相比，阶段更容易符合我们的预期：

```
ymd("2016-01-01") + dyears(1)
```

```
## [1] "2016-12-31 06:00:00 UTC"
```

```
ymd("2016-01-01") + years(1)
```

```
## [1] "2017-01-01"
```

日期时间数据处理

显然，dyears(1) / ddays(365) 应该返回 1，因为时期总是以秒来表示的，表示 1 年的时期就定义为相当于 365 天的秒数。

那么 years(1) / days(1) 应该返回什么呢？如果年份是 2015 年，那么结果就是 365，但如果年份是 2016 年，那么结果就是 366！没有足够的信息让 lubridate 返回一个明确的结果。lubridate 的做法是给出一个估计值，同时给出一条警告：

```
years(1) / days(1)
```

```
## [1] 365.25
```

如果需要更精确的测量方式，那么你就必须使用**区间**。区间是带有起点的时期，这使得其非常精确，你可以确切地知道它的长度：

```
next_year ← today() + years(1)
(today() %--% next_year) / ddays(1)
```

```
## [1] 366
```

要想知道一个区间内有多少个阶段，可以使用整除：

```
(today() %--% next_year) %/% days(1)
```

```
## [1] 366
```

函数式编程

函数式编程

将函数作为参数传入另一个函数的这种做法是一种非常强大的功能，它是促使 R 成为函数式编程语言的因素之一。使用 purrr 函数代替 for 循环的目的是将常见的列表处理问题分解为独立的几个部分。

- 对于列表中的单个元素，你能找到解决问题的方法吗？如果找到了解决方法，那么你就可以使用 purrr 将这种方法扩展到列表中的所有元素。
- 如果你面临的是一个非常复杂的问题，那么如何将其分解为几个可行的子问题，然后循序渐进地解决，直至完成最终的解决方案？使用 purrr，你可以解决很多子问题，然后再通过管道操作将这些问题的结果组合起来。

先对向量进行循环，然后对其每个元素进行一番处理，最后保存结果。这种模式太普遍了，因此 purrr 包提供了一个函数族来替你完成这种操作。每种类型的输出都有一个相应的函数：

- `map()`：用于输出列表
- `map_lgl()`：用于输出逻辑型向量
- `map_int()`：用于输出整型向量
- `map_dbl()`：用于输出双精度型向量
- `map_chr()`：用于输出字符型向量

每个函数都使用一个向量作为输入，并对向量的每个元素应用一个函数，然后返回和输入向量同样长度（同样名称）的一个新向量。向量的类型由映射函数的后缀决定。

映射函数

可能有些人会告诉你不要使用 for 循环，因为它们很慢。这些人完全错了！至少他们已经赶不上时代了，因为 for 循环已经有很多年都不慢了。使用 `map()` 函数的主要优势不是速度，而是简洁：它们可以让你的代码更易编写，也更易读。

```
df ← tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
```

映射函数的重点在于需要执行的操作，而不是在所有元素中循环所需的跟踪记录以及保存结果。

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

```
##           a           b           c           d
## -0.44035038  0.20582271 -0.12759328 -0.09144579
```

```
map_dbl(df, median)
```

```
##           a           b           c           d
## -0.86197579  0.49090621  0.06775493 -0.47913311
```

```
map_dbl(df, sd)
```

```
##         a         b         c         d
## 1.0945595 0.9677670 0.8015442 1.0429700
```

映射函数

所有 purrr 函数都是用 C 实现的。这使得它们的速度非常快，但牺牲了一些可读性。第二个参数（即 `.f`，要应用的函数）可以是一个公式、一个字符向量或一个整型向量。

对于参数 `.f`，你可以使用几种快捷方式来减少输入量。假设你想对某个数据集中的每个分组都拟合一个线性模型。以下这个示例将 `mtcars` 数据集拆分成 3 个部分（按照气缸的值分类），对每个部分拟合一个线性模型：

```
models ← mtcars |>
  split(mtcars$cyl) |>
  map(function(df) lm(mpg ~ wt, data = df))
```

因为 R 中创建匿名函数的语法比较繁琐，所以 purrr 提供了一种更方便的快捷方式 - **单侧公式**：

```
models ← mtcars |>
  split(mtcars$cyl) |>
  map(~lm(mpg ~ wt, data = .))
```

我们在以上示例中使用了 `.` 作为一个代词：它表示当前列表元素，与 `for` 循环中用 `i` 表示当前索引是一样的。

映射函数

当检查多个模型时，有时你会需要提取出像 R^2 这样的摘要统计量。要想完成这个任务，需要先运行 `summary()` 函数，然后提取出结果中的 `r.squared`。我们可以使用匿名函数的快捷方式来完成这个操作：

```
models |>
  map(summary) |>
  map_dbl(~.$r.squared)
```

```
##         4         6         8
## 0.5086326 0.4645102 0.4229655
```

因为提取命名成分的这种操作非常普遍，所以 purrr 提供了一种更为简洁的快捷方式：使用字符串。

```
models |>
  map(summary) |>
  map_dbl("r.squared")
```

```
##         4         6         8
## 0.5086326 0.4645102 0.4229655
```

你还可以使用整数按照位置来选取元素：

```
x <- list(
  list(1, 2, 3), list(4, 5, 6), list(7, 8, 9))
x |> map_dbl(2)
```

```
## [1] 2 5 8
```

对操作失败的处理

当使用映射函数重复多种操作时，某次操作失败的概率会大大增加。当这种情况发生时，你不仅会收到一条错误消息，而且不会得到任何结果。safely() 是一个修饰函数（副词），它接受一个函数（动词），对其进行修改并返回修改后的函数。这样一来，修改后的函数就不会抛出错误。相反，它总是会返回由以下两个元素组成的一个列表。

- result：原始结果。如果出现错误则为 NULL。
- error：错误对象。如果操作成功则为 NULL。

我们使用一个简单的 log 函数来进行说明：

```
safe_log ← safely(log)
```

```
str(safe_log(10))
```

```
## List of 2
##  $ result: num 2.3
##  $ error : NULL
```

```
str(safe_log("a"))
```

```
## List of 2
##  $ result: NULL
##  $ error :List of 2
##   ..$ message: chr "non-numeric argument to mathema
tical function"
##   ..$ call   : language .Primitive("log")(x, base)
##   ..- attr(*, "class")= chr [1:3] "simpleError" "er
ror" "condition"
```

对操作失败的处理

当函数成功运行时，result 元素中包含原始结果，error 元素的值是 NULL；当函数运行失败时，result 元素的值是 NULL，error 元素中包含错误对象。

safely() 也可以与 map() 函数共同使用：

```
x ← list(1, 10, "a")
y ← x |> map(safely(log))
```

```
str(y)
```

```
## List of 3
##  $ :List of 2
##   ..$ result: num 0
##   ..$ error : NULL
##  $ :List of 2
##   ..$ result: num 2.3
##   ..$ error : NULL
##  $ :List of 2
##   ..$ result: NULL
##   ..$ error :List of 2
##   .. ..$ message: chr "non-numeric argument to math
ematical function"
##   .. ..$ call   : language .Primitive("log")(x, bas
e)
##   .. ..- attr(*, "class")= chr [1:3] "simpleError"
"error" "condition"
```

对操作失败的处理

如果将以上结果转换为两个列表，一个列表包含所有错误对象，另一个列表包含所有原始结果，那么处理起来就会更加容易。可以使用 purrr::transpose() 函数轻松完成这个任务：

```
y <- y |> transpose()
```

```
str(y)
```

```
## List of 2
##  $ result:List of 3
##   ..$ : num 0
##   ..$ : num 2.3
##   ..$ : NULL
##  $ error :List of 3
##   ..$ : NULL
##   ..$ : NULL
##   ..$ :List of 2
##   .. ..$ message: chr "non-numeric argument to math
ematical function"
##   .. ..$ call   : language .Primitive("log")(x, bas
e)
##   .. ..- attr(*, "class")= chr [1:3] "simpleError"
"error" "condition"
```

对操作失败的处理

purrr 还提供了另外两个有用的修饰函数。

- 与 safely() 类似，possibly() 函数也总是会成功返回。它比 safely() 还要简单一些，因为可以设定出现错误时返回一个默认值：

```
x ← list(1, 10, "a")
x |> map_dbl(possibly(log, NA_real_))
```

```
## [1] 0.000000 2.302585       NA
```

- quietly() 函数与 safely() 的作用基本相同，但前者的结果中不包含错误对象，而是包含输出、消息和警告：

```
x ← list(1, -1)
x |> map(quietly(log)) |> str()
```

```
## List of 2
##  $ :List of 4
##   ..$ result  : num 0
##   ..$ output  : chr ""
##   ..$ warnings: chr(0)
##   ..$ messages: chr(0)
##  $ :List of 4
##   ..$ result  : num NaN
##   ..$ output  : chr ""
##   ..$ warnings: chr "NaNs produced"
##   ..$ messages: chr(0)
```

多参数映射

迄今为止，我们的映射函数都是对单个输入进行映射。但我们经常会有多个相关的输入需要同步迭代，这就是 map2() 和 pmap() 函数的用武之地。例如，假设你想模拟几个均值不同的随机正态分布，我们已经知道了如何使用 map() 函数来完成这个任务：

```
mu ← list(5, 10, -3)
mu |>
  map(rnorm, n = 5) |>
  str()
```

```
## List of 3
##  $ : num [1:5] 3.7 7.17 5.91 5.39 4.48
##  $ : num [1:5] 11.51 11.12 6.95 10.56 9.49
##  $ : num [1:5] -1.66 -2.41 -2.14 -3.2 -2.3
```

如果还想让标准差也不同，那么该怎么办呢？其中一种方法是使用均值向量和标准差向量的索引进行迭代：

```
sigma ← list(1, 5, 10)
seq_along(mu) |>
  map(~rnorm(5, mu[[.]], sigma[[.]])) |>
  str()
```

```
## List of 3
##  $ : num [1:5] 5.32 3.51 5.23 6.36 5.17
##  $ : num [1:5] 8.54 6.81 13.59 8.69 2.27
##  $ : num [1:5] -4.59 -5.64 10.31 -24.05 -3.68
```

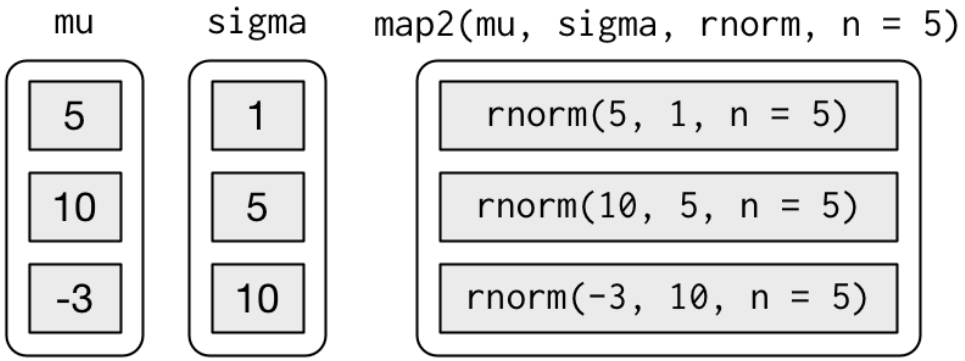
多参数映射

但是这种方法很难让人理解代码的本意。相反，我们应该使用 map2() 函数，它可以对两个向量进行同步迭代：

```
map2(mu, sigma, rnorm, n = 5) |> str()
```

```
## List of 3
##  $ : num [1:5] 4.06 6.33 4.51 4.24 5.93
##  $ : num [1:5] 14.77 8.89 8.52 10.98 10.64
##  $ : num [1:5] 3.47 -8.04 7.28 -9.94 -20.25
```

map2() 函数可以生成以下一系列函数调用：



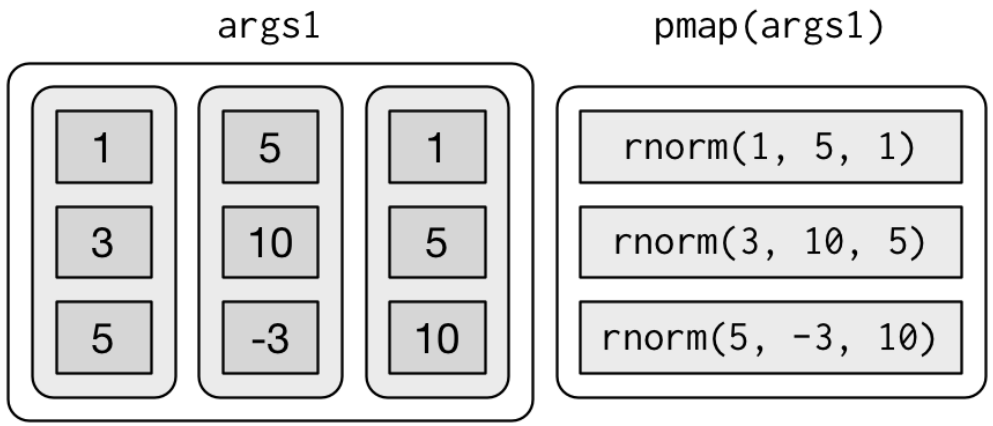
注意，每次调用时值发生变化的参数（这里是 mu 和 sigma）要放在映射函数（这里是 rnorm）的前面，值保持不变的参数（这里是 n ）要放在映射函数的后面。

多参数映射

purrr 提供了 pmap() 函数，它可以将一个列表作为参数。如果你想生成均值、标准差和样本数量都不相同的正态分布，那么就可以使用这个函数：

```
n ← list(1, 3, 5)
args1 ← list(n, mu, sigma)
args1 |>
  pmap(rnorm) |>
  str()
```

```
## List of 3
##  $ : num 4.62
##  $ : num [1:3] 8.33 9.96 2.77
##  $ : num [1:5] -2.5 -5.63 -12.1 6.09 3.45
```



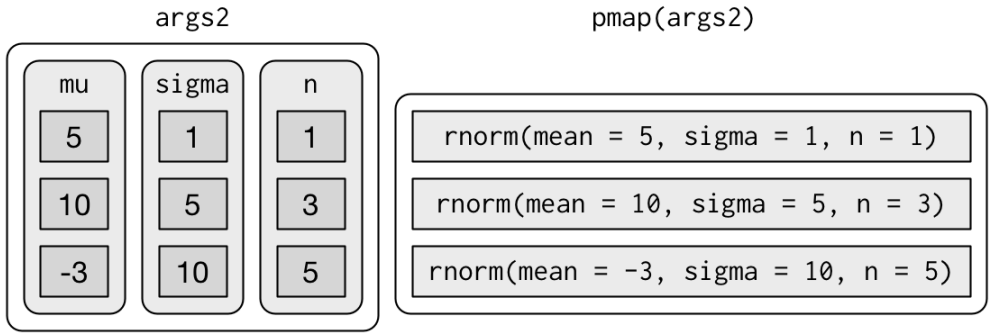
多参数映射

如果没有为列表的元素命名，那么 pmap() 在调用函数时就会按照位置匹配。这样做比较容 易出错，而且会让代码的可读性变差，因此最好使用命名参数：

```
args2 ← list(mean = mu, sd = sigma, n = n)
args2 |>
  pmap(rnorm) |>
  str()
```

```
## List of 3
##  $ : num 6.23
##  $ : num [1:3] 14.5 24.7 13
##  $ : num [1:5] 1.683 8.822 -10.755 -0.663 5.689
```

这样生成的函数调用更长一些，但更安全：



多参数映射

因为长度都是相同的，所以可以将各个参数保存在一个数据框中：

```
params ← tribble(
  ~mean, ~sd, ~n,
    5,     1,  1,
   10,     5,  3,
   -3,    10,  5
)
```

当代码变得比较复杂时，我们认为使用数据框是一种非常好的方法，因为这样可以确保每列都具有名称，而且与其他列具有相同的长度。

```
params |>
  pmap(rnorm)
```

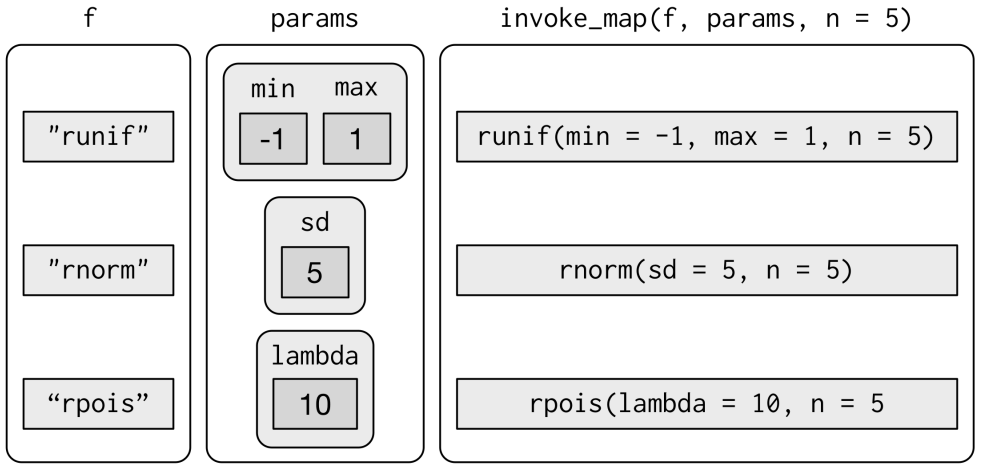
```
## [[1]]
## [1] 4.777998
## [[2]]
## [1] 13.36652 18.03639 12.13867
## [[3]]
## [1] -4.0268679 -6.2087283  0.3550966 -9.2375610 -6.
7430222
```

多参数映射

还有一种更复杂的情况：不但传给函数的参数不同，甚至函数本身也是不同的。为了处理这种情况，你可以使用 invoke_map() 函数：

```
f ← c("runif", "rnorm", "rpois")
param ← list(
  list(min = -1, max = 1),
  list(sd = 5),
  list(lambda = 10)
)
invoke_map(f, param, n = 5) |> str()
```

```
## List of 3
##  $ : num [1:5] 0.264 0.302 -0.297 -0.867 -0.223
##  $ : num [1:5] -0.5155 9.298 -0.0469 -0.69 0.5341
##  $ : int [1:5] 10 13 12 9 9
```



第一个参数是一个函数列表或包含函数名称的字符向量。第二个参数是列表的一个列表，其中给出了要传给各个函数的不同参数。随后的参数要传给每个函数。

游走函数

如果调用函数的目的是利用其副作用，而不是返回值时，那么就应该使用游走函数，而不是映射函数。通常来说，使用这个函数的目的是在屏幕上提供输出或者将文件保存到磁盘 - 重要的是操作过程，而不是返回值。以下是一个非常简单的示例：

```
x ← list(1, "a", 3)

x |>
  walk(print)
```

```
## [1] 1
## [1] "a"
## [1] 3
```

一般来说，walk() 函数不如 walk2() 和 pwalk() 实用。例如，如果有一个图形列表和一个文件名向量，那么你就可以使用 pwalk() 将每个文件保存到相应的磁盘位置：

```
library(ggplot2)
plots ← mtcars |>
  split(.$cyl) |>
  map(~ggplot(., aes(mpg, wt)) + geom_point())
paths ← stringr::str_c(names(plots), ".pdf")

pwalk(list(paths, plots), ggsave, path = tempdir())
```

walk()、 walk2() 和 pwalk() 都会隐式地返回 .x，即第一个参数。这使得它们非常适用于管道操作。

for 循环的其他模式

keep() 和 discard() 函数可以分别保留输入中预测值为 TRUE 和 FALSE 的元素：

```
iris |> keep(is.factor) |> str()
```

```
## 'data.frame':    150 obs. of  1 variable:
##  $ Species: Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```
iris |> discard(is.factor) |> str()
```

```
## 'data.frame':    150 obs. of  4 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
```

for 循环的其他模式

some() 和 every() 函数分别用来确定预测值是否对某个元素为真以及是否对所有元素为真：

```
x ← list(1:5, letters, list(10))
x |> some(is_character)
```

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

```
x |> every(is_vector)
```

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

detect() 函数可以找出预测值为真的第一个元素，detect_index() 函数则可以返回这个元 素的位置：

```
x ← sample(10)
x
```

```
##  [1]  9  1  7  8  4 10  3  6  5  2
```

```
x |> detect(~ . > 5)
```

```
## [1] 9
```

```
x |> detect_index(~ . > 5)
```

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

for 循环的其他模式

head_while() 和 tail_while() 分别从向量的开头和结尾找出预测值为真的元素：

```
x |> head_while(~ . > 5)
```

```
## [1] 9
```

```
x |> tail_while(~ . > 5)
```

```
## integer(0)
```

对于一个复杂的列表，有时你想将其归约为一个简单列表，方式是使用一个函数不断将两个元素合成一个。如果想要将两表间的一个 dplyr 操作应用于多张表，那么这种方法是非常适合的。例如，如果你有一个数据框列表，并想要通过不断将两个数据框连接成一个的方式来最终生成一个数据框：

```
dfs <- list(
  age = tibble(name = "John", age = 30),
  sex = tibble(name = c("John", "Mary"),
               sex = c("M", "F")),
  trt = tibble(name = "Mary", treatment = "A")
)
```

for 循环的其他模式

```
dfs |> reduce(full_join)
```

```
## Joining with `by = join_by(name)`
## Joining with `by = join_by(name)`
```

```
## # A tibble: 2 × 4
##   name    age sex   treatment
##   <chr> <dbl> <chr> <chr>
## 1 John     30 M     <NA>
## 2 Mary     NA F     A
```

或者你想要找出一张向量列表中的向量间的交集：

```
vs <- list(
  c(1, 3, 5, 6, 10),
  c(1, 2, 3, 7, 8, 10),
  c(1, 2, 3, 4, 8, 9, 10)
)
```

```
vs |> reduce(intersect)
```

```
## [1]  1  3 10
```

reduce() 函数使用一个“二元”函数（即具有两个基本输入的函数），将其不断应用于一 个列表，直到最后只剩下一个元素为止。累计函数与归约函数很相似，但前者会保留所有中间结果。你可以使用它来实现累计求和：

```
x <- sample(10)
x
```

```
##  [1]  9 10  5  3  6  4  7  1  8  2
```

```
x |> accumulate(`+`)
```

```
##  [1]  9 19 24 27 33 37 44 45 53 55
```

感谢倾听