1.4 数据结构 iii：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)
```

其中，

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)
```

其中，
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
```

其中，
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
其中，
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 = TURE, 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，其它元素是各个分组捕获，没有匹配则为 NA

```
x <- c("1978-2000", "2011-2020-2099")
pat <- "\\b(19|20)([0-9]{2})\\b"      # 正则表达式
str_extract(x, pat)
```

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

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

```
##      [,1]   [,2] [,3]
## [1,] "1978" "19" "78"
## [2,] "2011" "20" "11"
```

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)
```

其中，

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. 其它函数

- 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。
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。正则表达式是表示一组字符串的特征（或模式），正则表达式是处理文本数据非常强大的工具，将在下一节专门来讲解。

1.4.2 日期时间

日期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] "2020-08-28"
```

```
now()
```

```
## [1] "2020-08-28 23:07:11 CST"
```

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

```
## [1] "2020-08-28 UTC"
```

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

```
## [1] "2020-08-28"
```

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")
```

```
## Multiple formats matched: "Created on Sunday, %Om %d, %Y %H:%M %Op"(1), "Created on Sunday, %
```

```
## Using: "Created on Sunday, %Om %d, %Y %H:%M %Op"
```

```
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 日期时间组件

日期时间组件	含义	示例
%d	数字表示的日期（00~31）	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）	00~12
%b	缩写的月份	Jan
%B	非缩写的月份	January
%y	二位数的年份	16
%Y	四位数的年份	2016
%H	24 小时制小时（00~23）	00~23
%I	12 小时制小时（01~12）	01~12
%p	AM/PM 指示	AM/PM
%M	十进制分钟（00~60）	00~60
%S	十进制秒（00~60）	00~60

图 1.8: R 中的日期时间组件

```
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 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 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, ...)
```

其中，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。

时间序列领域的最新包，可参阅：

- tidyverts: https://github.com/tidyverts
- business-science: https://github.com/business-science