R 语言编程：基于 tidyverse

第 05 讲　数据结构 III: 因子, 字符串, 日期时间

张敬信

2022 年 12 月 1 日

哈尔滨商业大学

六. 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

- factor(x, levels, labels, ordered, ...): 将向量 x 创建为因子，levels 与数据中的值是一致的，若不指定则因子水平默认按字母顺序；labels 是你输出因子水平时，想要显示的标签值。

```
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"
```

- 若想规定顺序：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
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。

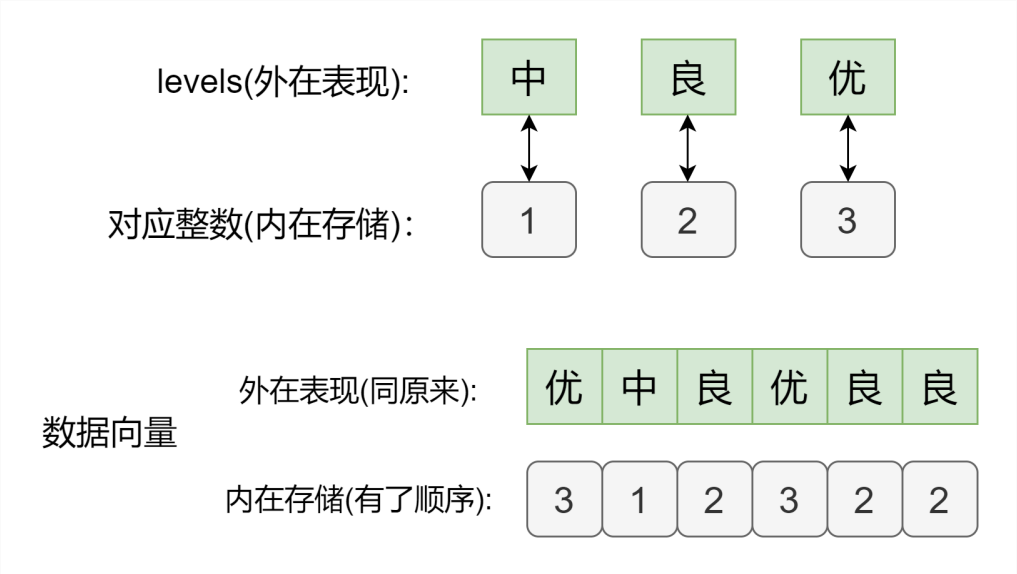


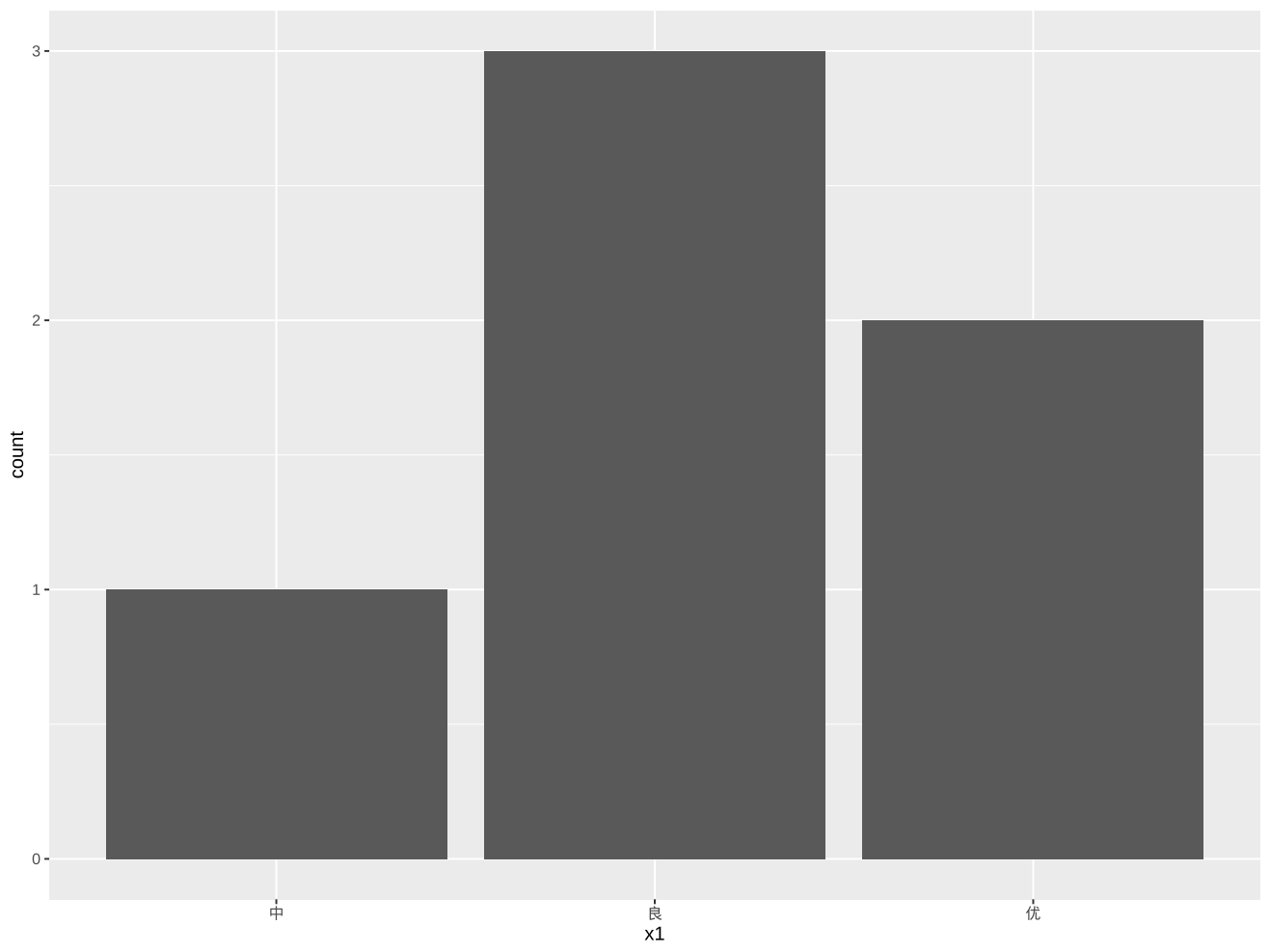
图 1: 因子的内在存储与外在表现

注：标签（labels）是因子水平（levels）的别名。

- 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()
```



- 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("Fair", "Good", "Excellent")  # 修改因子水平
x1
#> [1] Excellent Fair      Good      Excellent Good      G
#> Levels: Fair Good Excellent
```

- 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.

区分：因子固有的顺序与有序因子

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

- table(): 统计因子各水平（或向量各元素）的出现次数（频数）

```
table(x)
#> x
#> 良 优 中
#>  3  2  1
```

- cut(x, breaks, labels, ...): 将连续（数值）变量离散化，即切分为若干区间段，返回因子:

```
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
```

- `gl(n, k, length, labels, ordered, ...)`: 生成有规律的水平值组合因子，可用于多因素试验设计

```
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", "
#> # 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
#> # ... with 7 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)

- 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

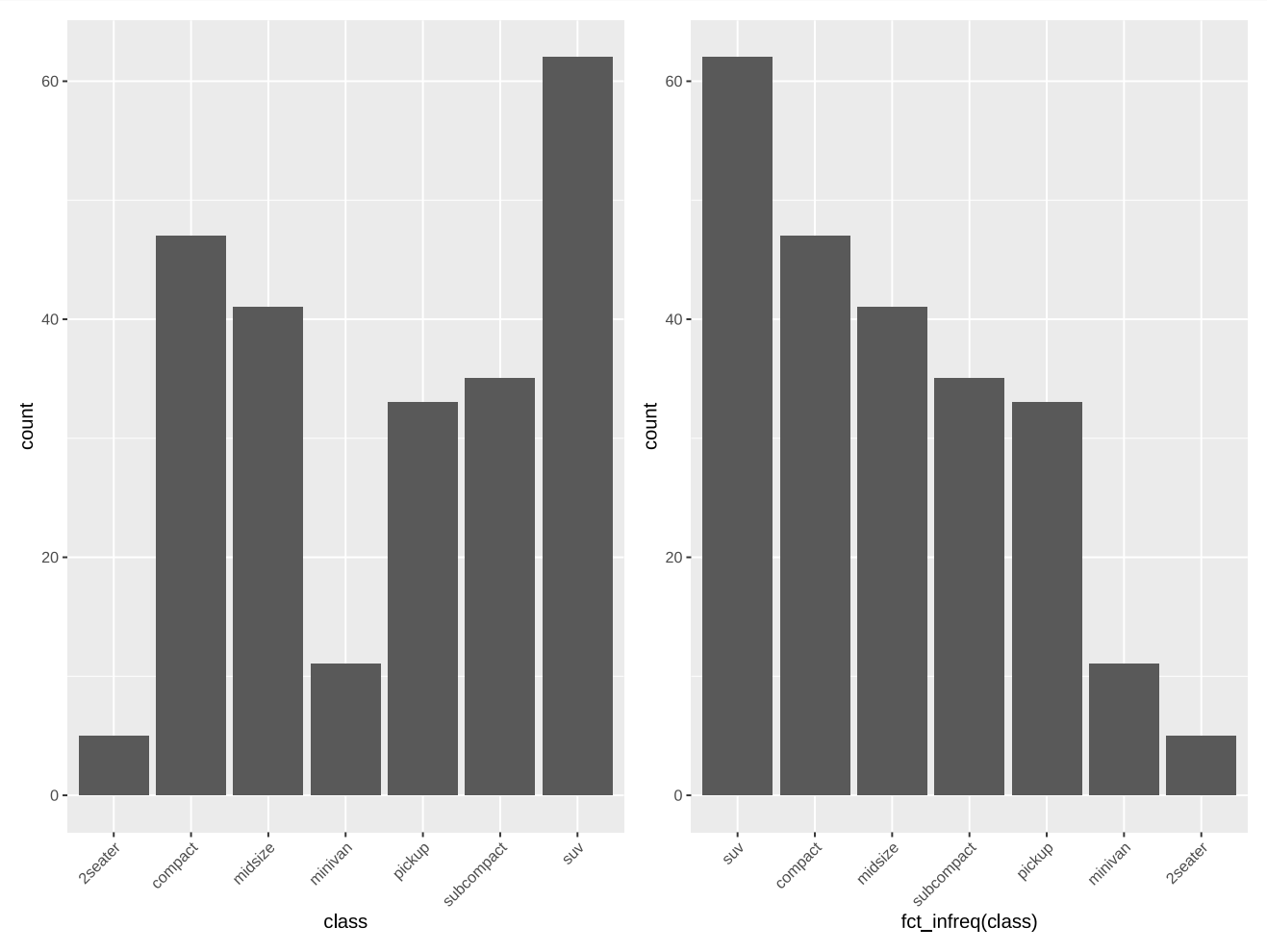
- **基本逻辑**：操作因子是操作一个向量，该向量更多的时候是以数据框的一列的形式存在的。

```
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
#> # ... with 2 more rows
```

```r
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
#> # ... with 1 more row
```

- 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() +
  theme(axis.text.x = element_text(angle = 45,
                                   vjust = 1, hjust = 1))
p2 = ggplot(mpg, aes(fct_infreq(class))) +
  geom_bar() +
  theme(axis.text.x = element_text(angle = 45,
                                   vjust = 1, hjust = 1))
library(patchwork)
p1 | p2
```


60
40
20
0
count
2seater
compact
midsize
minivan
pickup
subcompact
suv
class
60
40
20
0
count
suv
compact
midsize
subcompact
pickup
minivan
2seater
fct_infreq(class)

七. 字符串

- 字符串是用双引号或单引号括起来的若干字符[1]，字符串构成的向量，称为字符向量。数据清洗、可视化等操作都会用到字符串处理。
- `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[2]。
- 两点说明：
  - 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，各个函数都有后缀 `_all` 版本；
  - 各个函数中的参数 `pattern` 都支持用**正则表达式**。

[1]建议用双引号，除非字符串中包含双引号.

[2]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.

1. 字符串的长度（包含字符个数）

```
library(stringr)
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"
```

注：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)`：参数 `sep` 设置间隔符，默认为空字符；参数 `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 = "")
#> [1] "x1" "x2" "x3"
# 同 paste0("x", 1:3), paste("x", 1:3, sep="")
str_c("x", 1:3, collapse = "_")
#> [1] "x1_x2_x3"
str_c("x", str_c(sprintf("%03d", 1:3)))
#> [1] "x001" "x002" "x003"
```

- `str_dup(string, times)`: 将字符串重复 `n` 次

```
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

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

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
```

```r
df = mtcars[1:3,]
str_glue_data(df, "{rownames(df)} 总功率为 {hp} kW.")
#> Mazda RX4 总功率为 110 kW.
#> Mazda RX4 Wag 总功率为 110 kW.
#> Datsun 710 总功率为 93 kW.
```

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

注: 参数 `negate` 若为 `TRUE` 则反匹配。

```r
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

- `str_sub(string, start = 1, end = -1)`: 根据指定的起始和终止位置提取子字符串:

```
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"
```

- `str_subset(string, pattern, negate=FALSE)`: 提取字符向量中匹配的字符串

```
str_subset(x, "p")
#> [1] "apple" "pear"
```

8. 提取匹配的内容

- `str_extract(string, pattern)`：只提取匹配的内容；
- `str_match(string, pattern)`：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

- `str_replace(string, pattern, replacement)`: 用新字符串替换查找到的匹配字符串;

```
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

八. 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 包提供了一系列更方便的函数，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。
- **重要**：不要直接按字符串处理日期时间，先转化为日期时间对象（合适的数据结构）。

1. 识别日期时间

```
library(lubridate)
today()
#> [1] "2022-12-01"
now()
#> [1] "2022-12-01 20:08:56 CST"
as_datetime(today())    # 日期型转日期时间型
#> [1] "2022-12-01 UTC"
as_date(now())          # 日期时间型转日期型
#> [1] "2022-12-01"
```

- 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。

- 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 pm"
```

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。

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

```r
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
```

```r
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
```

- 修改时区
  - `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：

```
tz(t)                              # 时区
#> [1] "UTC"
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)  # 同 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
time_length(gap, "minute")   # 计算时间段的长度为多少分钟
#> [1] 3e+05
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
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

```
date = as_date("2019-01-01")
date %m+% months(0:11)
#>  [1] "2019-01-01" "2019-02-01" "2019-03-01" "2019-04-01
#>  [6] "2019-06-01" "2019-07-01" "2019-08-01" "2019-09-01
#> [11] "2019-11-01" "2019-12-01"
```

本篇主要参阅 (张敬信, 2022), (Hadley Wickham, 2017) 以及包文档，
模板感谢 (黄湘云, 2021), (谢益辉, 2021).

参考文献

Hadley Wickham, G. G. (2017). *R for Data Science*. O′ Reilly, 1 edition. ISBN 978-1491910399.

张敬信 (2022). *R* 语言编程：基于 *tidyverse*. 人民邮电出版社, 北京.

谢益辉 (2021). *rmarkdown: Dynamic Documents for R*.

黄湘云 (2021). *Github: R-Markdown-Template*.