Tidyverse 代码风格指南

发布 *0.0.1*

2020 年 08 月 30 日

Contents

本项目是 *The tidyverse style guide* 中文翻译。

在翻译 *R Packages 2nd editon* 时，第 6 章 *R Code* 中，6.2、6.3 节的内容被删去，放到了此处

CHAPTER 1

相关链接

The tidyverse style guide:

- **原书**: The tidyverse style guide
- **原书网页文档版**: The tidyverse style guide
- **个人译本**: The tidyverse style guide zh-CN
- GitHub Repo: https://github.com/YuanchenZhu2020/the_tidyverse_style_guide_zh_CN

R Packages:

- **原书**: R Packages
- **原书网页文档版**: R Packages
- **个人译本**: R Packages zh-CN
- GitHub Repo: https://github.com/YuanchenZhu2020/R_Packages_zh_CN

CHAPTER 2

翻译进度

：翻译完成

：还未翻译

- Welcome
- Files
- Syntax
- Fuctions
- Pipes
- ggplot2
- Files
- Documentation
- Tests
- Error messages
- News
- Git/GitHub

CHAPTER 3

全书目录

3.1 The tidyverse style guide

Hadley Wickham

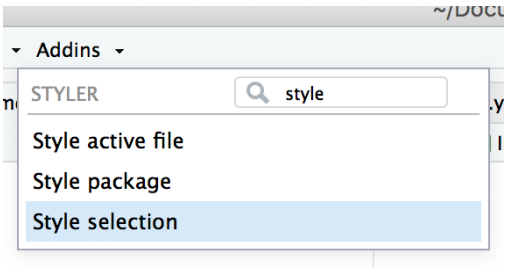
3.2 前言（Welcome）

好的编码风格就像正确的标点符号：没有标点符号你也能应付，但它能让我们更加容易阅读（butitsuremakesthingseasiertoread）。这个网站描述了整个 tidyverse 的代码风格。它源于 Google 最初的 R 代码风格指南（R Style Guide），但是 Google 当前的指南 是从 tidyverse 代码风格指南（tidyverse style guide）中派生出来的。

所有的代码风格指南基本上都是固执己见的。有些决策确实使代码更易于使用（特别是将缩进与编程结构相匹配），但许多决策都是武断的。代码风格指南最重要的是提供了一致性，使代码更易于编写，因为您需要做的决策更少。

两个 R 程序包支持此代码风格指南：

- styler 允许您交互式地重新设置所选文本、文件或整个项目的样式。它包含一个 RStudio 插件（Add-in），这是重新设计现有代码样式的最简单方法。



- lintr 执行自动检查以确认您的代码符合样式指南。

3.3 1 文件

3.3.1 1.1 文件名

文件名应该是有意义的并且以 .R 结尾。避免在文件名中使用特殊字符——请使用数字、字母、- 和 -。

```
# Good
fit_models.R
utility_functions.R

# Bad
fit models.R
foo.r
stuff.r
```

如果文件应该以特定的顺序运行，请在它们前面加上数字。如果有 10 个以上的文件，请在数字左侧添加 0 补足位数：

```
00_download.R
01_explore.R
...
09_model.R
10_visualize.R
```

如果你后来意识到自己错过了一些步骤，那么使用 02a、02b 等名字是很有诱惑力的。但是，我认为通常还是咬紧牙关，重命名所有文件。

请注意大小写，因为您或您的一些合作者使用的操作系统可能具有大小写不敏感的文件系统（例如 Microsoft Windows 或 OS X），这可能会导致修订控制系统（区分大小写）出现问题。请更倾向于使用所有字母小写的文件名，永远不要使用只在大小写上有区别的名称。

3.3.2 1.2 代码组织方式

很难准确地描述如何组织多个文件中的代码。我认为最好的经验法则是：如果能给一个文件起一个简洁的名字，同时又能让人想起里面的内容，那么你已经找到了一个好的代码组织方式。但要做到这一点很难。

3.3.3 1.3 内部结构

使用 - 和 = 的注释行将文件拆分为易于阅读的代码块。

```
# Load data ---------------------------

# Plot data ---------------------------
```

如果你的脚本使用了附加程序包，请在文件的最开始一次性加载它们。这比在代码中散布 `library()` 调用或在启动文件（如 `.Rprofile`）中加载的隐藏依赖项更加透明。

3.4 2 语法

3.4.1 2.1 对象名称

> "There are only two hard things in Computer Science: cache invalidation and naming things."
>
> —Phil Karlton

变量名和函数名只能使用小写字母、数字和 _。在一个名字里用下划线（_）来分隔单词（即所谓的蛇形命名法）。

```
# Good
day_one
day_1

# Bad
DayOne
dayone
```

Base R 在函数名（`contrib.url()`）和类名（`data.frame`）中使用英文句点 .，但最好只为 S3 对象系统保留点。在 S3 中，方法被命名为 `function.class`；如果你也在函数名和类名中使用 .，您最终会得到一些令人困惑的方法，比如 `as.data.frame.data.frame()`。

如果您发现自己试图将数据填充到变量名中（例如 `model_2018`、`model_2019`、`model_2020`），请考虑使用列表或数据框。

一般来说，变量名应该是名词，函数名应该是动词。力求名字简洁而有意义（这并不容易！）。

```
# Good
day_one

# Bad
first_day_of_the_month
djm1
```

尽可能避免重复使用常用函数和变量的名称。这将给代码的读者带来混乱。

```
# Bad
T <- FALSE
c <- 10
mean <- function(x) sum(x)
```

3.4.2 2.2 使用空格

2.2.1 逗号

就像普通英语一样，在逗号后面总是放一个空格，而不是在它之前。

```
# Good
x[, 1]

# Bad
x[,1]
x[ ,1]
x[ , 1]
```

2.2.2 圆括号

对于常规的函数调用，不要在圆括号之内或之外添加空格。

```
# Good
mean(x, na.rm = TRUE)

# Bad
mean (x, na.rm = TRUE)
mean( x, na.rm = TRUE )
```

与 `if`、`for` 或 `while` 一起使用时，请在 `()` 前后放置空格。

```
# Good
if (debug) {
    show(x)
}

# Bad
if(debug){
    show(x)
}
```

对于函数参数，在 () 后面放置一个空格：

```
# Good
function(x) {}

# Bad
function (x) {}
function(x){}
```

2.2.3 Embracing

拥抱运算符（embracing operator）{{ }} 应该始终有内部空格来帮助强调其特殊行为：

```
# Good
max_by <- function(data, var, by) {
data %>%
    group_by({{ by }}) %>%
    summarise(maximum = max({{ var }}, na.rm = TRUE))
}

# Bad
max_by <- function(data, var, by) {
data %>%
    group_by({{by}}) %>%
    summarise(maximum = max({{var}}, na.rm = TRUE))
}
```

2.2.4 中缀运算符

大多数中缀运算符（=，+，-，<- 等）应始终由空格包围：

```
# Good
height <- (feet * 12) + inches
mean(x, na.rm = TRUE)

# Bad
height<-feet*12+inches
mean(x, na.rm=TRUE)
```

但是有一些例外情况，它们不应被空格包围：

- 具有高优先级的运算符：`::`，`:::`，`$`，`@`，`[`，`[[`，`^`，一元 `-`，一元 `+`，和 `:`。

  ```
  # Good
  sqrt(x^2 + y^2)
  df$z
  x <- 1:10

  # Bad
  sqrt(x ^ 2 + y ^ 2)
  df $ z
  x <- 1 : 10
  ```

- 右侧是单个标识符（single identifier）的单边公式（single-sided formulas）：

  ```
  # Good
  ~foo
  tribble(
  ~col1, ~col2,
  "a",   "b"
  )

  # Bad
  ~ foo
  tribble(
  ~ col1, ~ col2,
  "a", "b"
  )
  ```

  请注意，右侧复杂的单边公式确实需要一个空格：

  ```
  # Good
  ~ .x + .y
  ```

（下页继续）

(续上页)

```
# Bad
~.x + .y
```

- 用于整洁评估（tidy evalutaion）时 `!!``(bang-bang)` 还有 `!!!``(bang-bang-bang)`（因为优先级相当于一元 `-/+`）

```
# Good
call(!!xyz)

# Bad
call(!! xyz)
call( !! xyz)
call(! !xyz)
```

- 帮助运算符

```
# Good
package?stats
?mean

# Bad
package ? stats
? mean
```

2.2.5 额外的空格

添加额外的空格可以改善 = 或 <- 的对齐方式。

```
# Good
list(
    total = a + b + c,
    mean  = (a + b + c) / n
)

# Also fine
list(
    total = a + b + c,
    mean = (a + b + c) / n
)
```

不要在通常不允许使用空格的地方增加额外的空格。

3.4.3 2.3 函数调用

2.3.1 命名参数

函数的参数通常分为两大类：一类提供要计算的**数据**；另一类控制计算的**细节**。调用函数时，通常会忽略数据参数的名称，因为它们的用法非常普遍。如果重写参数的默认值，请使用全名：

```
# Good
mean(1:10, na.rm = TRUE)

# Bad
mean(x = 1:10, , FALSE)
mean(, TRUE, x = c(1:10, NA))
```

避免参数的部分匹配。

2.3.2 赋值（Assignment）

避免在函数调用中进行赋值：

```
# Good
x <- complicated_function()
if (nzchar(x) < 1) {
    # do something
}

# Bad
if (nzchar(x <- complicated_function()) < 1) {
    # do something
}
```

唯一的例外是使用捕获副作用的函数：

```
output <- capture.output(x <- f())
```

3.4.4 2.4 控制流

2.4.1 代码块

大括号 {} 定义了 R 代码最重要的层次结构。要使此层次结构易于查看，请执行以下操作：

- { 应该是行内的最后一个字符。相关代码（如 if 子句、函数声明、尾部逗号……）必须与左大括号位于同一行。
- 内容应该缩进两个空格。
- } 应该是行内的第一个字符。

```
# Good
if (y < 0 && debug) {
    message("y is negative")
}

if (y == 0) {
    if (x > 0) {
        log(x)
    } else {
        message("x is negative or zero")
    }
} else {
    y^x
}

test_that("call1 returns an ordered factor", {
    expect_s3_class(call1(x, y), c("factor", "ordered"))
})

tryCatch(
    {
        x <- scan()
        cat("Total: ", sum(x), "\n", sep = "")
    },
    interrupt = function(e) {
        message("Aborted by user")
    }
)

# Bad
if (y < 0 && debug) {
message("Y is negative")
}

if (y == 0)
```

(下页继续)

(续上页)

```
{
        if (x > 0) {
            log(x)
        } else {
    message("x is negative or zero")
        }
} else { y ^ x }
```

2.4.2 行内语句

只要不产生副作用，就可以不使用大括号来处理只适合一行的简单语句。

```
# Good
y <- 10
x <- if (y < 20) "Too low" else "Too high"
```

影响控制流的函数调用（如 `return()`、`stop()` 或 `continue`）应始终位于它们自己的 `{}` 代码块中：

```
# Good
if (y < 0) {
    stop("Y is negative")
}

find_abs <- function(x) {
    if (x > 0) {
        return(x)
    }
    x * -1
}

# Bad
if (y < 0) stop("Y is negative")

if (y < 0)
    stop("Y is negative")

find_abs <- function(x) {
    if (x > 0) return(x)
    x * -1
}
```

2.4.3 隐式类型强制转换（Implicit type coercion）

避免在 `if` 语句中使用隐式类型强制转换（例如从数值类型强制转换为逻辑类型）：

```
# Good
if (length(x) > 0) {
    # do something
}

# Bad
if (length(x)) {
    # do something
}
```

2.4.4 Switch 语句

- 避免使用基于位置的 `switch()` 语句（即首选名称）。
- 每个元素都应该在自己的行上。
- 采用后面元素的值的元素在 `=` 后应具有空格。
- 除非您以前验证过输入，否则请提供一个失败抛出错误。

```
# Good
switch(x
    a = ,
    b = 1,
    c = 2,
    stop("Unknown `x`", call. = FALSE)
)

# Bad
switch(x, a = , b = 1, c = 2)
switch(x, a =, b = 1, c = 2)
switch(y, 1, 2, 3)
```

3.4.5 2.5 长的行

尽量将代码限制在每行 80 个字符。这适合一个大小合理的字体打印页面。如果您发现自己的空间不足，这是一个很好的指示，您应该将一些工作封装在一个单独的函数中。

如果函数调用太长，无法放在一行中，请为函数名、每个参数和结束符) 分别使用一行。这使得代码更易于以后的阅读和更改。

```
# Good
do_something_very_complicated(
    something = "that",
    requires = many,
    arguments = "some of which may be long"
)

# Bad
do_something_very_complicated("that", requires, many, arguments,
                              "some of which may be long"
                              )
```

如 [参数名称] 中所述，您可以省略非常常见的参数（即几乎在每次函数调用中都会使用的参数）的名称。短的未命名参数也可以与函数名称位于同一行，即使整个函数调用跨越了多行。

```
map(x, f,
    extra_argument_a = 10,
    extra_argument_b = c(1, 43, 390, 210209)
)
```

如果参数彼此密切相关，也可以将多个参数放在同一行上，例如，调用 `paste()` 或 `stop()` 时的字符串。在构建字符串时，尽可能将一行代码与一行输出相匹配。

```
# Good
paste0(
    "Requirement: ", requires, "\n",
    "Result: ", result, "\n"
)

# Bad
paste0(
    "Requirement: ", requires,
    "\n", "Result: ",
    result, "\n")
```

3.4.6 2.6 分号

不要把 ; 放在一行的末尾，并且不要使用 ; 把多个命令放在同一行中。

3.4.7 2.7 赋值（Assignment）

使用 <- 而不是 = 来进行赋值。

```
# Good
x <- 5

# Bad
x = 5
```

3.4.8 2.8 数据

2.8.1 字符向量

引用文本时使用 " 而不是 '。唯一的例外是文本已经包含双引号，并且不包含单引号。

```
# Good
"Text"
'Text with "quotes"'
'<a href="http://style.tidyverse.org">A link</a>'

# Bad
'Text'
'Text with "double" and \'single\' quotes'
```

2.8.2 逻辑向量

相比于 T 和 F，应该更倾向于使用 TRUE 和 FALSE。

3.4.9 2.9 注释

注释的每一行都应以注释符号和一个空格开头：#

在数据分析的代码中，使用注释来记录重要的发现和分析决策。如果您需要注释来解释您的代码在做什么，请考虑重写代码以使其更加清楚。如果你有更多的注释，可以考虑使用 R Markdown。

3.5 3 函数

3.5.1 3.1 命名

除了遵循 对象名称 的一般建议外，尽量使用动词作为函数名：

```
# Good
add_row()
permute()

# Bad
row_adder()
permutation()
```

3.5.2 3.2 长的行

如果函数定义占用了多行，则将第二行缩进到定义开始的位置。

```
# Good
long_function_name <- function(a = "a long argument",
                               b = "another argument",
                               c = "another long argument") {
    # As usual code is indented by two spaces.
}

# Bad
long_function_name <- function(a = "a long argument",
    b = "another argument",
    c = "another long argument") {
    # Here it's hard to spot where the definition ends and the
    # code begins
}
```

3.5.3 3.3 return()

只对前部的返回值使用 `return()`。否则，依赖 R 返回最后计算的表达式的结果。

```
# Good
find_abs <- function(x) {
    if (x > 0) {
        return(x)
    }
    x * -1
}
add_two <- function(x, y) {
```

(下页继续)

(续上页)

```
    x + y
}

# Bad
add_two <- function(x, y) {
    return(x + y)
}
```

返回语句应该总是在自己的行上，因为它们对控制流有重要影响。另请参见 inline statements。

```
# Good
find_abs <- function(x) {
    if (x > 0) {
        return(x)
    }
    x * -1
}

# Bad
find_abs <- function(x) {
    if (x > 0) return(x)
    x * -1
}
```

如果函数的调用主要是为了它的副作用（如打印文字、打印图片或保存到磁盘），那么它应该以不可见的方式返回第一个参数。这使得可以将函数用作管道（pipe）的一部分。`print` 方法通常应该这样做，例如 httr 的示例：

```
print.url <- function(x, ...) {
    cat("Url: ", build_url(x), "\n", sep = "")
    invisible(x)
}
```

3.5.4 3.4 注释

在代码中，使用注释来解释“为什么”，而不是“什么”或“如何”。注释的每一行都应该以注释符号和一个空格开始：`#`。

```
# Good
```

(下页继续)

(续上页)

```
# Objects like data frames are treated as leaves
x <- map_if(x, is_bare_list, recurse)



# Bad

# Recurse only with bare lists
x <- map_if(x, is_bare_list, recurse)
```

注释应该以句子的形式组织，并且只有在至少包含两个句子时才以句号结尾：

```
# Good

# Objects like data frames are treated as leaves
x <- map_if(x, is_bare_list, recurse)

# Do not use `is.list()`. Objects like data frames must be treated
# as leaves.
x <- map_if(x, is_bare_list, recurse)



# Bad

# objects like data frames are treated as leaves
x <- map_if(x, is_bare_list, recurse)

# Objects like data frames are treated as leaves.
x <- map_if(x, is_bare_list, recurse)
```

3.6 4 管道

3.6.1 4.1 介绍

使用 %>% 来强调一系列操作，而不是要对其执行操作的对象。

避免在以下情况下使用管道：

- 一次需要操纵多个对象。为应用于一个主要对象的一系列步骤保留管道符。
- 有一些有意义的中间对象可以被赋予信息性的名称。

3.6.2 4.2 空格

%>% 前面应该总是有一个空格，后面通常应该有一个新行。在第一步之后，每行应该缩进两个空格。这种结构使添加新步骤（或重新排列现有步骤）更加容易，并且更难忽略步骤。

```
# Good
iris %>%
    group_by(Species) %>%
    summarize_if(is.numeric, mean) %>%
    ungroup() %>%
    gather(measure, value, -Species) %>%
    arrange(value)

# Bad
iris %>% group_by(Species) %>% summarize_all(mean) %>%
ungroup %>% gather(measure, value, -Species) %>%
arrange(value)
```

3.6.3 4.3 长的行

如果函数的参数不能全部放在一行，请将每个参数放在自己的行上并缩进：

```
iris %>%
    group_by(Species) %>%
    summarise(
        Sepal.Length = mean(Sepal.Length),
        Sepal.Width = mean(Sepal.Width),
        Species = n_distinct(Species)
    )
```

3.6.4 4.4 短管道

单步管道可以保持在一行上，但是除非您计划以后扩展它，否则应该考虑将其重写为常规函数调用。

```
# Good
iris %>% arrange(Species)

iris %>%
    arrange(Species)

arrange(iris, Species)
```

有时在较长的管道中包含一个短管道作为函数的参数是很有用的。仔细考虑代码可读性在使用短的内联管道（不需要在其他地方查找）时是否更高，还是将代码移到管道之外并为其指定一个准确的名称会更好。

```
# Good
x %>%
    select(a, b, w) %>%
    left_join(y %>% select(a, b, v), by = c("a", "b"))

# Better
x_join <- x %>% select(a, b, w)
y_join <- y %>% select(a, b, v)
left_join(x_join, y_join, by = c("a", "b"))
```

3.6.5 4.5 没有参数

magritr 允许您在没有参数的函数上省略 ()。但是建议避免使用此功能。

```
# Good
x %>%
    unique() %>%
    sort()

# Bad
x %>%
    unique %>%
    sort
```

3.6.6 4.6 赋值

有三种可接受的赋值形式：

- 变量名和赋值操作在分开的行上：

  ```
  iris_long <-
      iris %>%
      gather(measure, value, -Species) %>%
      arrange(-value)
  ```

- 变量名和赋值操作在同一行上：

```
iris_long <- iris %>%
    gather(measure, value, -Species) %>%
    arrange(-value)
```

- 使用 -> 在管道末端赋值：

```
iris %>%
    gather(measure, value, -Species) %>%
    arrange(-value) ->
    iris_long
```

我认为这是最自然的书写方式，但会让阅读变得更困难：当变量名排在第一位时，它可以作为一个标题来提醒你管道的用途。

magritr 程序包提供 %<>% 运算符作为就地修改对象的快捷方式。建议避开这个操作。

```
# Good
x <- x %>%
    abs() %>%
    sort()

# Bad
x %<>%
    abs() %>%
    sort()
```

3.7 5 ggplot2

3.7.1 5.1 介绍

用于分隔 ggplot2 图层的 + 的代码风格建议与管道中 %>% 的代码风格建议非常相似。

3.7.2 5.2 空格

+ 前面应该总是有一个空格，后面应该有一个新行。即使你的图像只有两个图层也是如此。在第一步之后，每行应该缩进两个空格。

如果您正在创建 dplyr 管道的 ggplot，那么应该只有一个缩进级别。

```
# Good
iris %>%
```

(下页继续)

(续上页)

```
    filter(Species == "setosa") %>%
    ggplot(aes(x = Sepal.Width, y = Sepal.Length)) +
    geom_point()

# Bad
iris %>%
    filter(Species == "setosa") %>%
    ggplot(aes(x = Sepal.Width, y = Sepal.Length)) +
        geom_point()

# Bad
iris %>%
    filter(Species == "setosa") %>%
    ggplot(aes(x = Sepal.Width, y = Sepal.Length)) + geom_point()
```

3.7.3 5.3 长的行

如果 ggplot2 图层的参数不能全部放在一行，请将每个参数放在自己的行上并缩进：

```
# Good
ggplot(aes(x = Sepal.Width, y = Sepal.Length, color = Species)) +
    geom_point() +
    labs(
        x = "Sepal width, in cm",
        y = "Sepal length, in cm",
        title = "Sepal length vs. width of irises"
    )

# Bad
ggplot(aes(x = Sepal.Width, y = Sepal.Length, color = Species)) +
    geom_point() +
    labs(x = "Sepal width, in cm", y = "Sepal length, in cm", title = "Sepal length vs.␣
↪width of irises")
```

ggplot2 允许您在 data 参数内进行数据操作，例如过滤或切片。请不要使用这种方式，应该在开始绘制之前在管道中执行数据操作。

```
# Good
iris %>%
    filter(Species == "setosa") %>%
```

(下页继续)

(续上页)

```
    ggplot(aes(x = Sepal.Width, y = Sepal.Length)) +
    geom_point()

# Bad
ggplot(filter(iris, Species == "setosa"), aes(x = Sepal.Width, y = Sepal.Length)) +
    geom_point()
```

3.8 6 程序包中的文件

第一章中的大部分建议也适用于程序包中的文件。重要区别如下所述。

3.8.1 6.1 名字

- 如果一个文件包含一个函数，请给该文件指定与该函数相同的名称。
- 如果一个文件包含多个相关函数，请给它一个简洁但又能引起联想的名称。
- 弃用的函数应该位于前缀为 `deprec-` 的文件中。
- 兼容性函数应该位于前缀为 `compat-` 的文件中。

3.8.2 6.2 代码组织方式

在包含多个函数的文件中，公共函数及其文档应该首先出现，而私有函数则出现在所有文档化函数之后。如果多个公共函数共享同一个文档，则它们都应该一个接一个地放在文档块之后。

有关在程序包中撰写函数文档的更详细的指导，请参见 7。

```
# Bad
help_compute <- function() {
    # ... Lots of code ...
}

#' My public function
#'
#' This is where the documentation of my function begins.
#' ...
#' @export
do_something_cool <- function() {
    # ... even more code ...
```

(下页继续)

(续上页)

```
    help_compute()
}
```

```
# Good
#' Lots of functions for doing something cool
#'
#' ... Complete documentation ...
#' @name something-cool
NULL

#' @describeIn something-cool Get the mean
#' @export
get_cool_mean <- function(x) {
    # ...
}

#' @describeIn something-cool Get the sum
#' @export
get_cool_sum <- function(x) {
    # ...
}
```

3.9 8 测试

3.9.1 8.1 文件组织

测试文件的组织应该与 `R/` 文件的组织相匹配：如果一个函数位于 `R/foofy.R` 中，那么它的测试应该位于 `tests/testthat/test-foofy.R` 中。

使用 `usethis::use_test()` 自动创建具有正确名称的文件。

`context()` 不是很重要；testthat 的未来版本将在输出中显示文件名而不是上下文。

- genindex
- modindex
- search