

R 语言教程

Dongfeng Li

2018-03-07

目录

前言	11
I 介绍与入门	13
1 R 语言介绍	15
1.1 R 的历史和特点	15
1.2 R 的下载与安装	17
1.3 练习	18
2 R 语言入门运行样例	19
2.1 命令行界面	19
2.2 四则运算	19
2.3 数学函数	20
2.4 输出	23
2.5 向量计算与变量赋值	24
2.6 工作空间介绍	25
2.7 绘图示例	26
2.8 汇总统计示例	29
2.9 运行源程序文件	32
2.10 帮助	33
2.11 附录：数据	33
II R 的数据类型与相应运算	35
3 常量与变量	37
3.1 常量	37
3.2 变量	37
3.3 R 数据类型	38
4 数值型向量及其运算	39

4.1	数值型向量	39
4.2	向量运算	40
4.3	向量函数	42
4.4	复数向量	44
4.5	练习	45
5	逻辑型向量及其运算	47
5.1	逻辑型向量与比较运算	47
5.2	逻辑运算	48
5.3	逻辑运算函数	49
6	字符型数据及其处理	51
6.1	字符型向量	51
6.2	<code>paste()</code> 函数	51
6.3	转换大小写	51
6.4	字符串长度	52
6.5	取子串	52
6.6	类型转换	52
6.7	字符串拆分	53
6.8	字符串替换功能	53
6.9	正则表达式	54
7	R 向量下标和子集	55
7.1	正整数下标	55
7.2	负整数下标	56
7.3	空下标与零下标	56
7.4	下标超界	56
7.5	逻辑下标	57
7.6	<code>which()</code> 、 <code>which.min()</code> 、 <code>which.max()</code> 函数	58
7.7	元素名	58
7.8	用 R 向量下标作映射	59
7.9	集合运算	60
7.10	练习	61
8	R 数据类型的性质	63
8.1	存储模式与基本类型	63
8.2	类属	65
8.3	类型转换	65
8.4	属性	66
8.5	<code>str()</code> 函数	67
8.6	关于赋值	68

9 R 日期时间	71
9.1 R 日期和日期时间类型	71
9.2 从字符串生成日期数据	71
9.3 日期显示格式	73
9.4 访问日期时间的组成值	74
9.5 日期舍入计算	75
9.6 日期计算	75
9.7 基本 R 软件的日期功能	79
9.8 练习	83
10 R 因子类型	85
10.1 因子	85
10.2 table() 函数	87
10.3 tapply() 函数	87
10.4 forcats 包的因子函数	87
10.5 练习	90
11 R 矩阵和数组	93
11.1 R 矩阵	93
11.2 矩阵子集	94
11.3 cbind() 和 rbind() 函数	97
11.4 矩阵运算	97
11.5 逆矩阵与线性方程组求解	100
11.6 apply() 函数	101
11.7 多维数组	102
12 数据框	105
12.1 数据框	105
12.2 数据框内容访问	106
12.3 数据框与矩阵的区别	109
12.4 gl() 函数	109
12.5 tibble 类型	110
12.6 练习	113
13 列表类型	115
13.1 列表	115
13.2 列表元素访问	116
13.3 列表类型转换	118
13.4 返回列表的函数示例-strsplit()	119
14 工作空间	121

III R 编程	123
15 R 输入输出	125
15.1 输入输出的简单方法	125
15.2 读取 CSV 文件	127
15.3 Excel 表访问	133
15.4 使用专用接口访问数据库	137
15.5 文件访问	139
15.6 目录和文件管理	142
16 程序控制结构	143
16.1 表达式	143
16.2 分支结构	143
16.3 循环结构	144
16.4 R 中判断条件	146
16.5 管道控制	147
17 函数	149
17.1 函数基础	149
17.2 变量作用域	155
17.3 函数进阶	159
17.4 程序调试	168
17.5 函数式编程介绍	171
18 R 程序效率	177
18.1 R 的运行效率	177
18.2 向量化编程	178
18.3 减少显式循环	182
18.4 R 的计算函数	188
18.5 并行计算	193
IV 用 R 制作研究报告和图书	201
19 用 R 制作研究报告	203
20 Markdown 格式	205
20.1 介绍	205
20.2 Markdown 格式文件的应用	205
20.3 pandoc 用法	206
20.4 markdown 格式说明	206
21 R Markdown 文件格式	219

21.1 R Markdown 文件	219
21.2 在 R Markdown 文件中插入 R 代码	220
21.3 输出表格	222
21.4 利用 R 程序插图	223
21.5 代码段选项	224
21.6 数学公式	231
21.7 属性设置	232
22 用 bookdown 制作图书	237
22.1 介绍	237
22.2 入门	237
22.3 一本书的设置	237
22.4 章节结构	240
22.5 书的编译	241
22.6 数学公式和公式编号	242
22.7 定理类编号	243
22.8 文献引用	243
22.9 插图	243
22.10 表格	244
22.11 数学公式的设置	245
22.12 使用经验	246
22.13 bookdown 的一些使用问题	247
23 用 R Markdown 制作简易网站	249
23.1 介绍	249
24 制作幻灯片	253
24.1 介绍	253
24.2 数学公式处理	254
24.3 slidy 幻灯片激光笔失效问题的修改	255
24.4 R Presentation 格式文件	256
V R 数据处理	257
25 数据读取技巧	259
25.1 日期数据	259
25.2 缺失值处理	262
25.3 练习	263
26 数据整理	265
26.1 tidyverse 系统	265

26.2 用 <code>filter()</code> 选择行子集	266
26.3 用 <code>select()</code> 选择列子集	267
26.4 用 <code>arrange()</code> 排序	270
26.5 用 <code>rename()</code> 修改变量名	271
26.6 用 <code>mutate()</code> 计算新变量	271
26.7 用管道连接多次操作	272
26.8 数据简单汇总	273
26.9 长宽表转换	277
26.10 拆分数据列	280
26.11 合并数据列	281
26.12 横向合并	281
26.13 利用第二个数据集筛选	284
26.14 数据集的集合操作	284
26.15 数据框纵向合并	284
26.16 标准化	286
26.17 用 <code>reshape</code> 包做长宽表转换	288
27 数据汇总	297
27.1 用 <code>summary()</code> 函数作简单概括	297
27.2 连续型变量概括函数	299
27.3 分类变量概括	300
27.4 数据框概括	305
27.5 分类概括	305
27.6 练习	312
VI 绘图	313
28 绘图	315
28.1 常用高级图形	315
28.2 低级图形函数	344
28.3 图形参数	354
28.4 图形输出	360
28.5 包含多种中文字体的图形	360
VII 统计分析	363
29 R 初等统计分析	365

VIII 用 Rcpp 连接 C++ 代码	367
30 Rcpp 介绍	369
30.1 Rcpp 的用途	369
30.2 Rcpp 入门样例	370
31 R 与 C++ 的类型转换	373
31.1 用 <code>wrap()</code> 把 C++ 变量返回到 R 中	373
31.2 用 <code>as()</code> 函数把 R 变量转换为 C++ 类型	374
31.3 <code>as()</code> 和 <code>wrap()</code> 的隐含调用	374
32 Rcpp 属性	375
32.1 Rcpp 属性介绍	375
32.2 在 C++ 源程序中指定要导出的 C++ 函数	376
32.3 在 R 中编译链接 C++ 代码	376
32.4 Rcpp 属性的其它功能	378
33 Rcpp 提供的 C++ 数据类型	381
33.1 <code>RObject</code> 类	381
33.2 <code>IntegerVector</code> 类	382
33.3 <code>NumericVector</code> 类	384
33.4 Rcpp 的其它向量类	387
33.5 Rcpp 提供的其它数据类型	389
34 Rcpp 糖	395
34.1 简单示例	395
34.2 向量化的运算符	396
34.3 用 Rcpp 访问数学函数	397
34.4 返回单一逻辑值的函数	399
34.5 返回糖表达式的函数	400
34.6 R 与 Rcpp 不同语法示例	402
35 用 Rcpp 帮助制作 R 扩展包	403
35.1 不用扩展包共享 C++ 代码的方法	403
35.2 生成扩展包	404
35.3 重新编译	406
35.4 建立 C++ 用的接口界面	406
IX 专题	407
36 R 语言的文本处理	409
36.1 简单的文本处理	409

36.2 文本文件读写	414
36.3 正则表达式	415
36.4 stringr 包	430
36.5 正则表达式应用例子	433
37 随机模拟	443
37.1 随机数	443
37.2 sample() 函数	444
37.3 随机模拟示例	444
X R 编程例子	451
38 R 编程例子	453
38.1 R 语言	453
38.2 概率	454
38.3 智者千虑必有一失	454
38.4 科学计算	454
38.5 统计计算	456
38.6 数据处理	460
38.7 文本处理	463

前言

李东风的《R 语言教程》的草稿。还在更新中。

书中的数学公式使用 MathJax 库显示，下面是数学公式测试。如果数学公式显示的中文不正常，在浏览器中用鼠标右键单击中文公式，在弹出的菜单中选择 **Math Settings--Math Renderer** 选 HTML-CSS 或 SVG 即可。

公式测试：

$$f(x) = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

中文公式测试：

$$\text{相对误差} = \frac{a - A}{A}$$

需要安装的软件包：tidyverse bookdown xtable microbenchmark reshape

本教程中用到的软件包：xtable microbenchmark bookdown tidyverse xml2 tibble tidyr stringr rstudioapi rvest rlang readxl reprex readr purr magrittr modelr lubridate httr jsonlite hms ggplot2 haven forcats dbplyr dplyr cli crayon broom pillar stringi selectr whisker rmarkdown knitr callr clipr cellranger openssl curl mime lazyeval gtable scales digest DBI tidymodels plogr BH Rcpp pkgconfig R6 glue assertthat bindrcpp psych reshape2 plyr rprojroot utf8 base64enc htmltools yaml markdown highr evaluate rematch labeling viridisLite munsell dichromat RColorBrewer bindr backports mnormt clrspace

Part I

介绍与入门

Chapter 1

R 语言介绍

1.1 R 的历史和特点

1.1.1 R 的历史

R 语言来自 S 语言，是 S 语言的一个变种。S 语言由 Rick Becker, John Chambers 等人在贝尔实验室开发，著名的 C 语言、Unix 系统也是贝尔实验室开发的。

S 语言第一个版本开发于 1976-1980，基于 Fortran；于 1980 年移植到 Unix，并对外发布源代码。1984 年出版的“棕皮书”(Becker and Chambers, 1984)总结了 1984 年为止的版本，并开始发布授权的源代码。这个版本叫做旧 S。与我们现在用的 S 语言有较大差别。

1989–1988 对 S 进行了较大更新，变成了我们现在使用的 S 语言，称为第二版。1988 年出版的“蓝皮书”(Becker et al., 1988)做了总结。

1992 年出版的“白皮书”(Chambers and Hastie, 1992)描述了在 S 语言中实现的统计建模功能，增强了面向对象的特性。软件称为第三版，这是我们现在用的多数版本。

1998 年出版的“绿皮书”(Chambers, 2008)描述了第四版 S 语言，主要是编程功能的深层次改进。现行的 S 系统并没有都采用第四版，S-PLUS 的第 5 版才采用了 S 语言第四版。

S 语言商业版本为 S-PLUS, 1988 年发布，现在为 Tibco Software 拥有。命运多舛，多次易主。

R 是一个自由软件，GPL 授权，最初由新西兰 Auckland 大学的 Ross Ihaka 和 Robert Gentleman 于 1997 年发布，R 实现了与 S 语言基本相同的功能和统计功能。现在由 R 核心团队开发，但全世界的用户都可以贡献软件包。R 的网站: <http://www.r-project.org/>

1.1.2 R 的特点

1.1.2.1 R 语言一般特点

- 自由软件，免费、开放源代码，支持各个主要计算机系统；
- 完整的程序设计语言，基于函数和对象，可以自定义函数，调入 C、C++、Fortran 编译的代码；
- 具有完善的数据类型，如向量、矩阵、因子、数据集、一般对象等，支持缺失值，代码像伪代码一样简洁、可读；
- 强调交互式数据分析，支持复杂算法描述，图形功能强；
- 实现了经典的、现代的统计方法，如参数和非参数假设检验、线性回归、广义线性回归、非线性回归、可加模型、树回归、混合模型、方差分析、判别、聚类、时间序列分析等。
- 统计科研工作者广泛使用 R 进行计算和发表算法。R 有上万软件包 (截止 2017 年 8 月有一万一千多个)。

1.1.2.2 R 语言和 R 软件的技术特点

- 函数编程 (functional programming)。R 语言虽然不是严格的 functional programming 语言，但可以遵照其原则编程，得到可验证的可靠程序。
- 支持对象类和类方法。基于对象的程序设计。
- 是动态类型语言，解释执行，运行速度较慢。
- 数据框是基本的观测数据类型，类似于数据库的表。
- 开源软件 (Open source software)。可深入探查，开发者和用户交互。
- 可以用作 C 和 C++、FORTRAN 语言编写的算法库的接口。
- 主要数值算法采用已广泛测试和采纳的算法实现，如排序、随机数生成、线性代数 (LAPACK 软件包)。

1.1.2.3 推荐参考书

- R.L. Kabacoff(2012) 《R 语言实战》，人民邮电出版社。
- Hadley Wickham and Garrett Grolemund(2017) “R for Data Science”，<http://r4ds.had.co.nz/>, O'Reilly
- Hadley Wickham(2014) “Advanced R”，<http://adv-r.had.co.nz/>, Chapman & Hall/CRC The R Series
- R 网站上的初学者手册 “An Introduction to R” 和其它技术手册。
- John M. Chambers(2008), “Software for Data Analysis-Programming with R”, Springer.
- Venables, W. N. & Ripley, B. D.(2002) “Modern Applied Statistics with S”, Springer
- 薛毅、陈立萍 (2007) 《统计建模与 R 软件》，清华大学出版社。
- 汤银才 (2008)，《R 语言与统计分析》，高等教育出版社。
- 李东风 (2006) 《统计软件教程》，人民邮电出版社。

1.2 R 的下载与安装

1.2.1 R 的下载

以 MS Windows 操作系统为例。R 的主网站在<https://www.r-project.org/>。从 CRAN 的镜像网站下载软件，其中一个镜像如<http://mirror.bjtu.edu.cn/cran/>。选 “Download R for Windows-base-Download R 3.4.1 for windows” (3.4.1 是版本号，应下载网站上给出的最新版本) 链接进行下载。在 “Download R for Windows” 链接的页面，除了 base 为 R 的安装程序，还有 contrib 为 R 附加的扩展软件包下载链接（一般不需要从这里下载），以及 Rtools 链接，是在 R 中调用 C、C++ 和 Fortran 程序代码时需要用的编译工具。

RStudio (<https://www.rstudio.com/>) 是功能更强的一个 R 图形界面，在安装好 R 的官方版本后安装 RStudio 可以更方便地使用 R。

1.2.2 R 软件安装

下载官方的 R 软件后按提示安装。安装后获得一个桌面快捷方式，如 “R i386 3.4.1”(这是 32 位版本)。如果是 64 位操作系统，可以同时安装 32 位版本和 64 位版本，对初学者这两种版本区别不大。

重要步骤：在 C 盘或 D 盘建立一个文件夹（也叫做子目录），如 `c:\work`。把 R 的快捷方式拷贝入此文件夹，在 Windows 资源管理器中，右键单击此快捷方式，在弹出菜单中选 “属性”，把 “快捷方式” 页面的 “起始位置” 的内容变成空白。启动在 work 文件夹中的 R 快捷方式，出现命令行界面。R 主要依靠命令行执行功能。

安装官方的 R 软件后，可以安装 RStudio。平时使用可以使用 RStudio，其界面更方便，对 R Markdown 格式 (.Rmd) 文件支持更好。

用 R 进行数据分析，不同的分析问题需要放在不同的文件夹中。如果仅使用官方的 R 软件，只要每个文件夹放置一个 “起始位置” 为空的 R 快捷方式图标，从对应的快捷图标启动。如果使用 RStudio，每个分析项目需要单独建立一个 “项目” (project)，每个项目也有一个工作文件夹。

1.2.3 辅助软件

R 可以把一段程序写在一个以 .r 或 .R 为扩展名的文本文件中，如 “date.r”，称为一个 __ 源程序 __ 文件，然后在 R 命令行用

```
source('date.r')
```

运行源程序。这样的文件可以用记事本生成和编辑。

在 MS Windows 操作系统中建议使用 notepad++ 软件，这是 MS Windows 下记事本程序的增强型软件。安装后，在 MS Windows 资源管理器中右键弹出菜单会有 “edit with notepadpp” 选项。notepad++ 可以方便地在不同的中文编码之间转换。

RStudio 则是一个集成环境，可以在 RStudio 内进行源程序文件编辑和运行。

1.2.4 R 扩展软件包的安装

R 扩展软件包提供了特殊功能。以安装 sos 包为例。sos 包用来搜索某些函数的帮助文档。在 R 图形界面选菜单“程序包-安装程序包”，在弹出的“CRAN mirror”选择窗口中选择一个中国的镜像如“China (Beijing 2)”，然后在弹出的“Packages”选择窗口中选择要安装的扩展软件包名称，即可完成下载和安装。

还可以用如下程序制定镜像网站 (例子中是位于清华大学的镜像网站) 并安装指定的扩展包：

```
options(repos=c(CRAN='http://mirror.tuna.tsinghua.edu.cn/CRAN/'))
install.packages('sos')
```

还可以选择扩展包的安装路径，如果权限允许，可以选择安装在 R 软件的主目录内或者用户自己的私有目录位置。由于用户的对子目录的读写权限问题，有时不允许一般用户安装扩展包到 R 的主目录中。用 `.libPaths()` 查看允许的扩展包安装位置，在 `install.packages()` 中用 `lib=` 指定安装位置：

```
print(.libPaths())
## [1] "D:/R/R-3.3.1/library"
install.packages('sos', lib=.libPaths()[1])
```

在 RStudio 中用“Tools”菜单的“Install Packages”安装软件包。

1.3 练习

1. 下载 R 安装程序，安装 R，建立 work 文件夹并在其中建立 R 的快捷方式。Windows 用户还需要下载 RTools 软件并安装。
2. 下载 RStudio 软件并安装。
3. 下载安装 notepad++ 软件。
4. 在 R 图形界面中下载安装 sos 扩展软件包。

Chapter 2

R 语言入门运行样例

2.1 命令行界面

启动 R 软件后进入命令行界面，每输入一行命令，就在后面显示计算结果。可以用向上和向下箭头访问历史命令；可以从已经运行过的命令中用鼠标拖选加亮后，用 `Ctrl+C` 复制后用 `Ctrl+V` 粘贴，或用 `Ctrl+X` 一步完成复制粘贴，粘贴的目标都是当前命令行。

如果使用 RStudio 软件，有一个“Console 窗格”相当于命令行界面。在 RStudio 中，可以用 New File–Script file 功能建立一个源程序文件（脚本文件），在脚本文件中写程序，然后用 Run 图标或者 `Ctrl+Enter` 键运行当前行或者选定的部分。

2.2 四则运算

四则运算如：

```
5 + (2.3 - 1.125)*3.2/1.1 + 1.23E3
```

```
## [1] 1238.418
```

结果为 1238.418，前面显示的结果在行首加了井号，这在 R 语言中表示注释。本教程的输出前面一般都加了井号以区分于程序语句。输出前面的方括号和序号 1 是在输出有多个值时提供的提示性序号，只有单个值时为了统一起见也显示出来了。这里 `1.23E3` 是科学记数法，表示 1.23×10^3 。用星号 `*` 表示乘法，用正斜杠/表示除法。

用 `^` 表示乘方运算，如

```
2^10
```

```
## [1] 1024
```

重要提示：关闭中文输入法，否则输入一些中文标点将导致程序错误。

2.2.1 计算例子

从 52 张扑克牌中任取 3 张，有多少种不同的组合可能？解答：有

$$C_{52}^3 = \frac{52!}{3!(52-3)!} = \frac{52 \times 51 \times 50}{3 \times 2 \times 1}$$

种，在 R 中计算如：

```
52*51*50/(3*2)
```

```
## [1] 22100
```

2.2.2 练习

1. 某人存入 10000 元 1 年期定期存款，年利率 3%，约定到期自动转存（包括利息）。问：

(1) 10 年后本息共多少元？

(2) 需要存多少年这 10000 元才能增值到 20000 元？

2. 成语说：“智者千虑，必有一失；愚者千虑，必有一得”。设智者作判断的准确率为 $p_1 = 0.99$ ，愚者作判断的准确率为 $p_2 = 0.01$ ，计算智者做 1000 次独立的判断至少犯一次错误的概率，与愚者做 1000 次独立判断至少对一次的概率。

2.3 数学函数

2.3.1 数学函数——平方根、指数、对数

例：

```
sqrt(6.25)
```

```
## [1] 2.5
```

```
exp(1)
```

```
## [1] 2.718282
```

```
log10(10000)
```

```
## [1] 4
```

`sqrt(6.25)` 表示 $\sqrt{6.25}$ ，结果为 2.5。`exp(1)` 表示 e^1 ，结果为 $e = 2.718282$ 。`log10(10000)` 表示 $\lg 10000$ ，结果为 4。`log` 为自然对数。

2.3.2 数学函数——取整

例:

```
round(1.1234, 2)
```

```
## [1] 1.12
```

```
round(-1.9876, 2)
```

```
## [1] -1.99
```

```
floor(1.1234)
```

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

```
floor(-1.1234)
```

```
## [1] -2
```

```
ceiling(1.1234)
```

```
## [1] 2
```

```
ceiling(-1.1234)
```

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

`round(1.1234, 2)` 表示把 1.1234 四舍五入到两位小数。`floor(1.1234)` 表示把 1.1234 向下取整, 结果为 1。`ceiling(1.1234)` 表示把 1.1234 向上取整, 结果为 2。

2.3.3 数学函数——三角函数

例:

```
pi
```

```
## [1] 3.141593
```

```
sin(pi/6)
```

```
## [1] 0.5
```

```
cos(pi/6)
```

```
## [1] 0.8660254
```

```
sqrt(3)/2
```

```
## [1] 0.8660254
```

```
tan(pi/6)
```

```
## [1] 0.5773503
```

pi 表示圆周率 π 。sin 正弦, cos 余弦, tan 正切, 自变量以弧度为单位。pi/6 是 30° 。

2.3.4 数学函数——反三角函数

例:

```
pi/6
```

```
## [1] 0.5235988
```

```
asin(0.5)
```

```
## [1] 0.5235988
```

```
acos(sqrt(3)/2)
```

```
## [1] 0.5235988
```

```
atan(sqrt(3)/3)
```

```
## [1] 0.5235988
```

asin 反正弦, acos 反余弦, atan 反正切, 结果以弧度为单位。

2.3.5 分布函数和分位数函数

例:

```
dnorm(1.98)
```

```
## [1] 0.05618314
```

```
pnorm(1.98)
```

```
## [1] 0.9761482
```

```
qnorm(0.975)
```

```
## [1] 1.959964
```

dnorm(x) 表示标准正态分布密度 $\phi(x) = \frac{1}{\sqrt{2\pi}}e^{-\frac{1}{2}x^2}$. pnorm(x) 表示标准正态分布函数 $\Phi(x) = \int_{-\infty}^x \phi(t) dt$. qnorm(y) 表示标准正态分布分位数函数 $\Phi^{-1}(x)$ 。还有其它许多分布的密度函数、分布函数和分位数函数。例如,

```
qt(1 - 0.05/2, 10)
```

```
## [1] 2.228139
```

求自由度为 10 的 t 检验的双侧临界值。其中 `qt(y,df)` 表示自由度为 `df` 的 t 分布的分位数函数。

2.4 输出

2.4.1 简单输出

命令行的计算结果直接显示在命令的后面。在用 `source()` 运行程序文件时，需要用 `print()` 函数显示一个表达式的结果，如：

```
print(sin(pi/2))
```

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

用 `cat()` 函数显示多项内容，包括数值和文本，文本包在两个单撇号或两个双撇号中，如：

```
cat('sin(pi/2)=', sin(pi/2), '\n')
```

```
## sin(pi/2)= 1
```

`cat()` 函数最后一项一般是 `'\n'`，表示换行。忽略此项将不换行。

再次提示：要避免打开中文输入法导致误使用中文标点。

2.4.2 用 `sink()` 函数作运行记录

R 使用经常是在命令行逐行输入命令（程序），结果紧接着显示在命令后面。如何保存这些命令和显示结果？在 R 命令行中运行过的命令会被保存在运行的工作文件夹中的一个名为 `.Rhistory` 的文件中。用 `sink()` 函数打开一个文本文件开始记录文本型输出结果。结束记录时用空的 `sink()` 即可关闭文件不再记录。如

```
sink('tmpres01.txt', split=TRUE)
print(sin(pi/6))
print(cos(pi/6))
cat('t(10)的双侧0.05分位数（临界值）=', qt(1 - 0.05/2, 10), '\n')
sink()
```

`sink()` 用作输出记录主要是在测试运行中使用，正常的输出应该使用 `cat()` 函数、`write.table()`、`write.csv()` 等函数。

2.4.3 练习

1. 用 `cat()` 函数显示

```
log10(2)=*** log10(5)=***
```

其中 *** 应该代以实际函数值。

2. 用 `sink()` 函数开始把运行过程记录到文件 “log001.txt” 中，在命令行试验几个命令，然后关闭运行记录，查看生成的 “log001.txt” 的内容。

2.5 向量计算与变量赋值

R 语言以向量为最小单位。用 `<-` 赋值。如

```
x1 <- 1:10
x1
```

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

一般的向量可以用 `c()` 生成，如

```
marks <- c(3, 5, 10, 5, 6)
```

在程序语言中，变量用来保存输入的值或计算的结果。变量可以存放各种不同类型的值，如单个数值、多个数值（称为向量）、单个字符串、多个字符串（称为字符型向量），等等。单个数值称为**标量**。

技术秘诀：用程序设计语言的术语描述，R 语言是动态类型的，其变量的类型不需要预先声明，运行过程中允许变量类型改变，实际上变量赋值是一种“绑定”（binding），将一个变量的名称（变量名）与实际的一个存储位置联系在一起。在命令行定义的变量称为**全局变量**。

用 `print()` 函数显示向量或在命令行中显示向量时，每行显示的行首会有方括号和数字序号，代表该行显示的第一个向量元素的下标。如

```
12345678901:12345678920
```

```
## [1] 12345678901 12345678902 12345678903 12345678904 12345678905
## [6] 12345678906 12345678907 12345678908 12345678909 12345678910
## [11] 12345678911 12345678912 12345678913 12345678914 12345678915
## [16] 12345678916 12345678917 12345678918 12345678919 12345678920
```

向量可以和一个标量作四则运算，结果是每个元素都和这个标量作四则运算，如：

```
x1 + 200
```

```
## [1] 201 202 203 204 205 206 207 208 209 210
```

```
2*x1
```

```
## [1] 2 4 6 8 10 12 14 16 18 20
```

```
2520/x1
```

```
## [1] 2520 1260 840 630 504 420 360 315 280 252
```


两个等长的向量可以进行四则运算，相当于对应元素进行四则运算，如

```
x2 <- x1 * 3
x2
```

```
## [1] 3 6 9 12 15 18 21 24 27 30
```

```
x2 - x1
```

```
## [1] 2 4 6 8 10 12 14 16 18 20
```

R 的许多函数都可以用向量作为自变量，结果是自变量的每个元素各自的函数值。如

```
sqrt(x1)
```

```
## [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
```

```
## [8] 2.828427 3.000000 3.162278
```

结果是 1 到 10 的整数各自的平方根。

2.6 工作空间介绍

在命令行中定义的变量，在退出 R 时，会提问是否保存工作空间，初学时可选择保存，真正用 R 进行数据分析时往往不保存工作空间。再次启动 R 后，能够看到以前定义的各个变量的值。

在使用 R 的官方版本时，如果在 Windows 中使用，一般把不同的数据分析项目放在不同的文件夹中。将 R 的程序快捷图标复制到每一个项目的文件夹中，并用右键菜单讲快捷图标的“属性”中“起始位置”改为空白。要分析哪一个项目的数据，就在那个项目文件夹中的 R 快捷图标启动，这样可以保证不同的项目有不同的工作空间。

如果使用 RStudio 软件，也需要把不同项目放在不同文件夹，并且每个项目在 RStudio 中单独建立一个“项目”（project）。要分析那个项目的数据，就打开那个项目。不同项目使用不同的工作空间。

2.6.1 练习

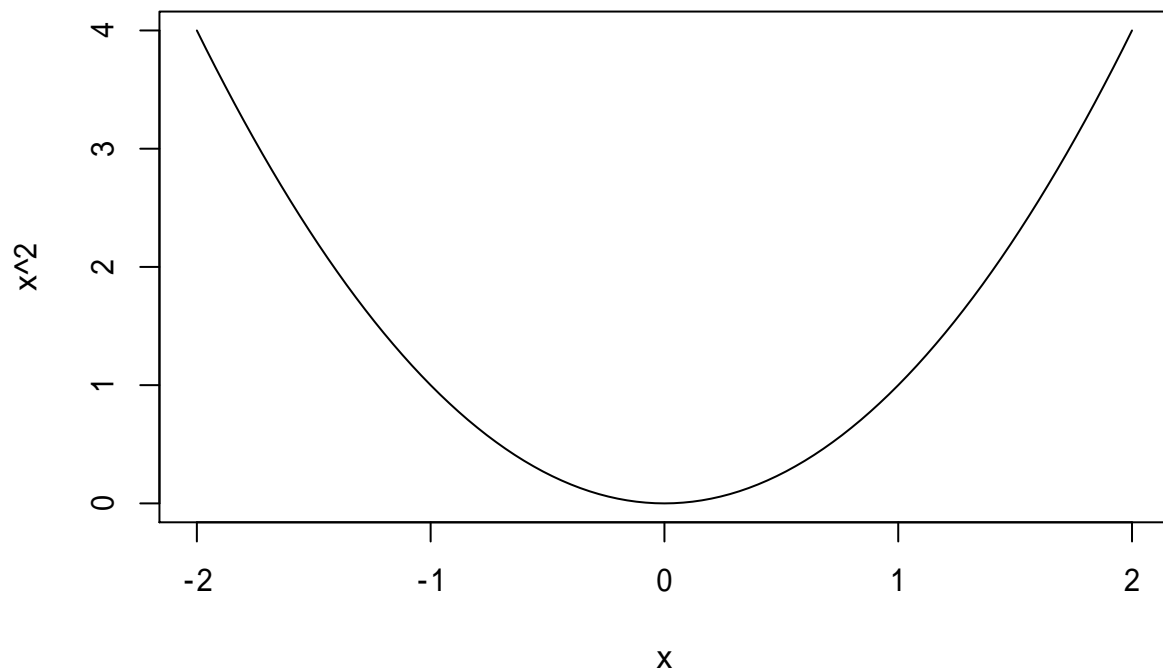
1. 某人存入 10000 元 1 年期定期存款，年利率 3%，约定到期自动转存（包括利息）。列出 1、2、……、10 年后的本息金额。
2. 显示 2 的 1,2,..., 20 次方。
3. 定义 x1 为 1 到 10 的向量，定义 x2 为 x1 的 3 倍，然后退出 R，再次启动 R，查看 x1 和 x2 的值。

2.7 绘图示例

2.7.1 函数曲线示例

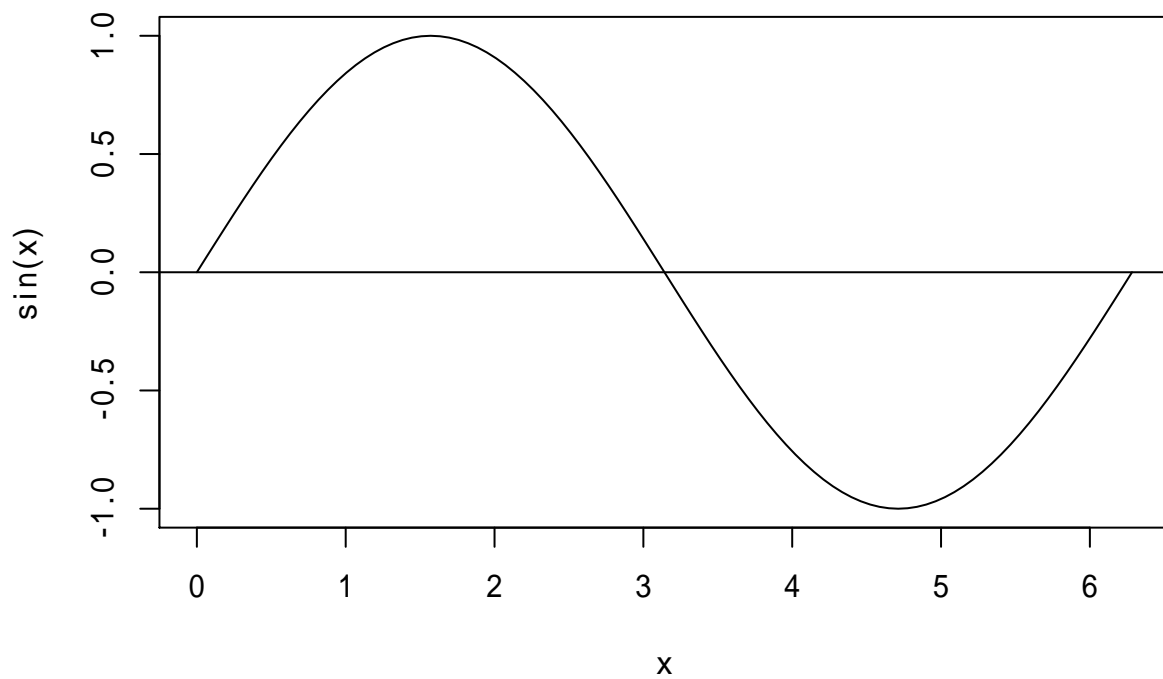
如下程序用 `curve()` 函数制作 $y = x^2$ 函数的曲线图, `curve()` 函数第二、第三自变量是绘图区间:

```
curve(x^2, -2, 2)
```



类似地, $\sin(x)$ 函数曲线图用如下程序可制作, 用 `abline()` 函数添加参考线:

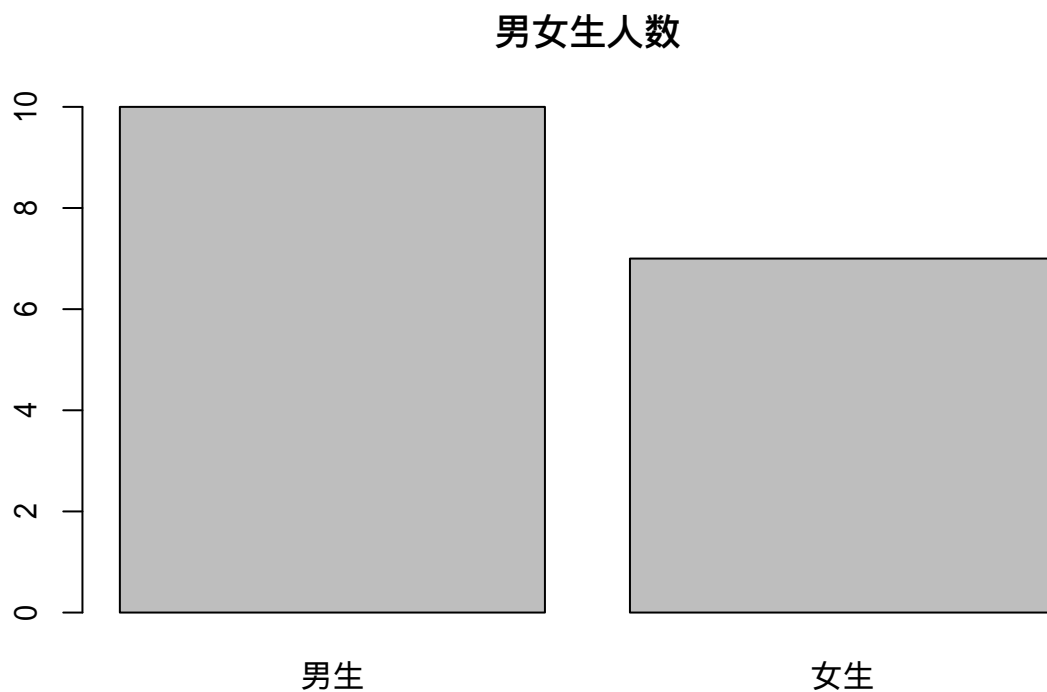
```
curve(sin(x), 0, 2*pi)
abline(h=0)
```



2.7.2 条形图示例

假设有 10 个男生，7 个女生，如下程序绘制男生、女生人数的条形图：

```
barplot(c(' 男生'=10, ' 女生'=7),  
        main=' 男女生人数')
```

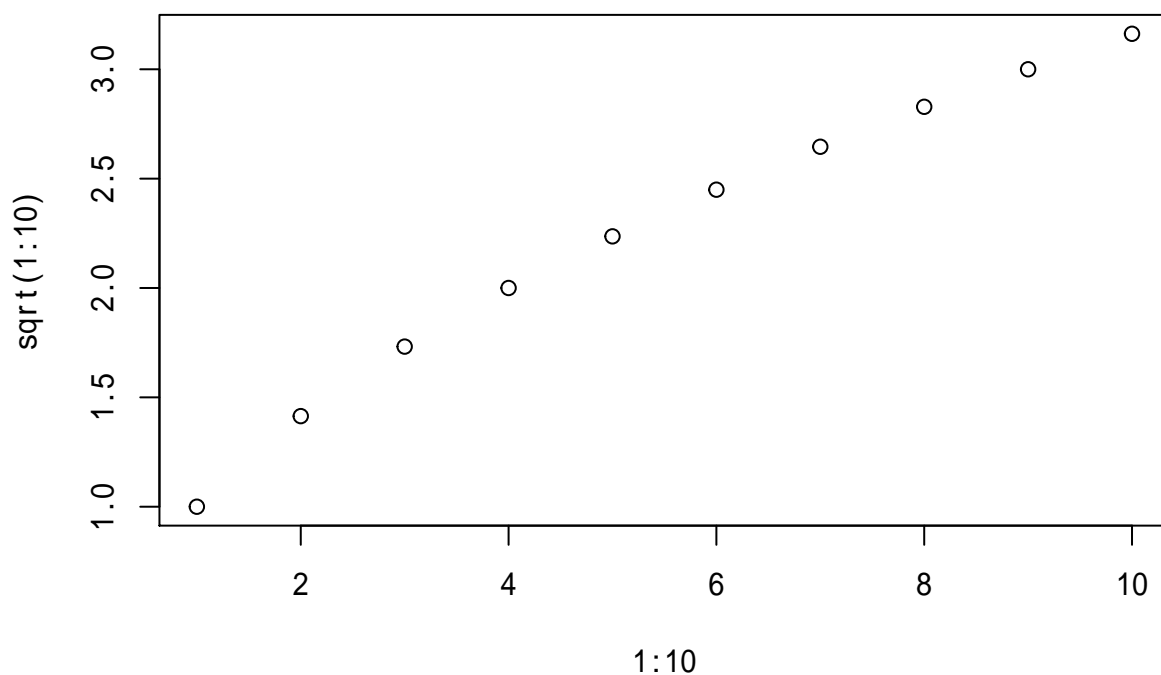


利用适当选项可以人为定制颜色、控制条形宽窄。实际问题中，个数（频数）一般是从数据中统计得到的。

2.7.3 散点图示例

下面的例子用 `plot()` 函数做了散点图, `plot()` 函数第一个自变量是各个点的横坐标值，第二个自变量是对应的纵坐标值。

```
plot(1:10, sqrt(1:10))
```



2.7.4 R 软件自带的图形示例

R 软件中自带了一些演示图形。通过如下程序可以调用：

```
demo("graphics")  
demo("image")
```

2.7.5 练习

1. 画 $\exp(x)$ 在 $(-2, 2)$ 区间的函数图形。
2. 画 $\ln(x)$ 在 $(0.01, 10)$ 区间的函数图形。

2.8 汇总统计示例

2.8.1 表格数据

统计用的输入数据典型样式是 Excel 表那样的表格数据。表格数据特点：每一列应该是相同的类型（或者都是数值，或者都是文字，或者都是日期），每一列应该有一个名字。

这样的表格数据，一般可以保存为.csv 格式：数据项之间用逗号分开，文件本身是文本型的，可以用普通记事本程序查看和编辑。Excel 表可以用“另存为”命令保存为.csv 格式。常用的数据库管理系统一般也可以把表保存为.csv 格式。

2.8.2 读入表格数据

例如，“taxsamp.csv”是这样一个 csv 格式表格数据文件，可以用 Excel 打开，也可以用记事本程序或 notepad++ 打开。内容和下载见本页面后面的附录。

用如下程序可以把.csv 文件读入到 R 中：

```
tax.tab <- read.csv("taxsamp.csv", header=TRUE, as.is=TRUE)
print(head(tax.tab))
```

程序中的选项 `header=TRUE` 指明第一行作为变量名行，选项 `as.is=TRUE` 说明字符型列要原样读入而不是转换为因子 (factor)。读入的变量 `tax.tab` 称为一个数据框 (data.frame)。`head()` 函数返回数据框或向量的前几项。比较大的表最好不要显示整个表，会使得前面的运行过程难以查看。

技术秘诀：`read.csv()` 的一个改进版本是 `readr` 扩展包的 `read_csv()` 函数，此函数读入较大表格速度要快得多，而且读入的转换设置更倾向于不做不必要的转换。

2.8.3 练习

1. 用 Excel 软件查看“taxsamp.csv”的内容 (双击即可)。
2. 用记事本程序或 notepad++ 软件查看“taxsamp.csv”的内容。
3. 读入“taxsamp.csv”到 R 数据框 `tax.tab` 中，查看 `tax.tab` 内容。

2.8.4 分类变量频数统计

在 `tax.tab` 中，“征收方式”是一个分类变量。用 `table()` 函数计算每个不同值的个数，称为频数 (frequency)：

```
table(tax.tab[, '征收方式'])
```

```
##
##      查帐征收  定期定额征收  定期定率征收
##           31             16             2
```

类似地可以统计“申报渠道”的取值频数：

```
table(tax.tab[, '申报渠道'])
```

```
##
##  大厅申报  网上申报
```

```
##          18          31
```

也可以用 `table()` 函数统计“征收方式”和“申报渠道”交叉分类频数，如：

```
table(tax.tab[, '征收方式'], tax.tab[, '申报渠道'])
```

```
##
##          大厅申报  网上申报
##  查帐征收          9        22
##  定期定额征收      9         7
##  定期定率征收      0         2
```

上述结果制表如下：

```
knitr::kable(table(tax.tab[, '征收方式'], tax.tab[, '申报渠道']))
```

	大厅申报	网上申报
查帐征收	9	22
定期定额征收	9	7
定期定率征收	0	2

2.8.5 数值型变量的统计

数值型变量可以计算各种不同的统计量，如平均值、标准差和各个分位数。`summary()` 可以给出最小值、最大值、中位数、四分之一分位数、四分之三分位数和平均值。如

```
summary(tax.tab[, '营业额'])
```

```
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##         0     650     2130  247327    9421 6048000
```

中位数是从小到大排序后排在中间的值。四分之一和四分之三分位数类似。

统计函数以一个数值型向量为自变量，包括 `sum`(求和)，`mean`(平均值)，`var`(样本方差)，`sd`(样本标准差)，`min`(最小值)，`max`(最大值)，`range`(最小值和最大值) 等。如

```
mean(tax.tab[, '营业额'])
```

```
## [1] 247327.4
```

```
sd(tax.tab[, '营业额'])
```

```
## [1] 1036453
```

如果数据中有缺失值，可以删去缺失值后计算统计量，这时在 `mean`、`sd` 等函数中加入 `na.rm=TRUE` 选项。

2.8.6 练习

用如下程序定义一个变量 `x`, 然后求 `x` 的平均值和最小值、最大值。

```
x <- c(3, 5, 10, 5, 6)
```

2.9 运行源程序文件

用 `source()` 函数可以运行保存在一个文本文件中的源程序。比如, 如下内容保存在文件 `ssq.r` 中:

```
sum.of.squares <- function(x){  
  sum(x^2)  
}
```

用如下 `source()` 命令运行:

```
source("ssq.r")
```

运行后就可以调用自定义函数 `sum.of.squares()` 了。

2.9.1 源文件编码

源程序文件存在编码问题。对于源程序编码与系统默认编码不同的情况, 在 `source()` 函数中可以添加 `encoding=` 选项。例如, 保存为 UTF-8 编码的源程序在简体中文 MS Windows 系统的 R 中运行, 可以在 `source()` 函数中可以添加 `encoding="UTF-8"` 选项。保存为 GBK 编码的源程序文件在 MAC 系统的 R 中运行, 可以在 `source()` 函数中可以添加 `encoding="GBK"` 选项。

在 RStudio 中, 可以打开一个源程序文件查看与编辑。用快捷键 “Ctrl+Enter” 或快捷图标 “Run” 可以运行当前行或者加亮选中行, 快捷图标 “Source” 可以运行整个文件。如果发现中文乱码, 可以用菜单 “Reopen with encoding” 选择合适的编码打开, 用菜单 “Save with encoding” 选择需要的编码保存。

2.9.2 当前工作目录

在用 `source()` 调用源程序文件或者用 `read.csv()` 读入数据文件时, 如果不写文件名的全路径, 就认为文件位置是在所谓 “当前工作目录”。用 `getwd()` 函数可以查询当前工作目录, 用 `setwd()` 函数可以设置当前工作目录。在 RStudio 中用菜单 “Session-Set working directory” 设置当前工作目录。

在 MS Windows 操作系统中使用 R 软件时, 一种好的做法是把某个研究项目所有数据和程序放在某个文件夹如 `c:\work` 中, 把 R 的程序快捷图标复制到该目录中, 在资源管理器中对该图标调用鼠标右键菜单 “属性”, 从弹出对话框中, 把 “起始位置” 一栏清除。这样, 每次从这个快捷图标启动 R, 就可以自动以所在子目录为当前工作目录, 工作空间和命令历史记录也默认存放在这里。

在 MS Windows 操作系统的 R 中使用文件路径时，要用正斜杠作为连接符，使用反斜杠则需要成对使用，如 `setwd('d:/work')` 或 `setwd('d:\\work')`。

如果使用 RStudio 软件，将某个研究项目所有数据和程序放在某个文件夹中，然后建立一个新项目 (project) 指向该文件夹。

2.9.3 练习

编辑生成 `ssq.r` 源程序文件并用 `source()` 函数运行，然后计算：

```
sum.of.squares(1:5)
```

2.10 帮助

用帮助菜单中“html 帮助”到互联网浏览器中查看帮助文档。“Search engine and keywords”项下面有分类的帮助。有软件包列表。

在命令行，用问号后面跟随函数名查询某函数的帮助。用 `example(函数名)` 的格式可以运行此函数的样例，如：

```
example(mean)
```

```
##
## mean> x <- c(0:10, 50)
##
## mean> xm <- mean(x)
##
## mean> c(xm, mean(x, trim = 0.10))
## [1] 8.75 5.50
```

安装 `sos` 附加包 (package)，用 `findfn(函数名)` 查询某个函数，结果显示在互联网浏览器软件中。

在 RStudio 中有一个单独的 Help 窗格，如果需要，可以用菜单“View-Panes-Zoom help”将其放大到占据整个窗口空间。

2.11 附录：数据

2.11.1 公司纳税数据样例

本数据是某地区 2013 年 12 月所属税款申报信息的一个子集，仅含 20 个公司的数据。

数据文件下载

数据文件显示

Part II

R 的数据类型与相应运算

Chapter 3

常量与变量

3.1 常量

R 语言基本的数据类型有数值型，逻辑型 (TRUE, FALSE)，文本 (字符串)。支持缺失值，有专门的复数类型。

常量是指直接写在程序中的值。

数值型常量包括整型、单精度、双精度等，一般不需要区分。写法如 123, 123.45, -123.45, -0.012, 1.23E2, -1.2E-2 等。为了表示 123 是整型，可以写成 123L。

字符型常量用两个双撇号或两个单撇号包围，如 "Li Ming" 或 'Li Ming'。字符型支持中文，如 "李明" 或 '李明'。国内的中文编码主要有 GBK 编码和 UTF-8 编码，有时会遇到编码错误造成乱码的问题，MS Windows 下 R 程序一般用 GBK 编码，但是 RStudio 软件采用 UTF-8 编码。在 R 软件内字符串一般用 UTF-8 编码保存。

逻辑型常量只有 TRUE 和 FALSE。

缺失值用 NA 表示。统计计算中经常会遇到缺失值，表示记录丢失、因为错误而不能用、节假日没有数据等。除了数值型，逻辑型和字符型也可以有缺失值，而且字符型的空白值不会自动辨识为缺失值，需要自己规定。R 支持特殊的 Inf 值，这是实数型值，表示正无穷大，不算缺失值。

复数常量写法如 2.2 + 3.5i, 1i 等。

3.2 变量

程序语言中的变量用来保存输入的值或者计算得到的值。在 R 中，变量可以保存所有的数据类型，比如标量、向量、矩阵、数据框、函数等。

变量都有变量名，R 变量名必须以字母、数字、下划线和句点组成，变量名的第一个字符不能取为数字。在

中文环境下，汉字也可以作为变量名的合法字符使用。变量名是区分大小写的，`y` 和 `Y` 是两个不同的变量名。

变量名举例: `x`, `x1`, `X`, `cancer.tab`, `diseaseData`。

用 `<-` 赋值的方法定义变量。`<-` 也可以写成 `=`，但是 `<-` 更直观。如

```
x5 <- 6.25
x6 = sqrt(x5)
```

R 的变量没有固定的类型，给已有变量赋值为新的类型，该变量就变成新的类型，但一般应避免这样的行为。R 是“动态类型”语言，赋值实际上是“绑定”（binding），即将一个变量名与一个存储地址联系在一起，同一个存储地址可以有多个变量名与其联系。

3.3 R 数据类型

R 语言基本的数据类型有数值，逻辑型（`TRUE`, `FALSE`），文本（字符串）。支持缺失值，有专门的复数类型。

R 语言数据结构包括向量，矩阵和数据框，多维数组，列表，对象等。数据中元素、行、列还可以用名字访问。最基本的是向量类型。向量类型数据的访问方式也是其他数据类型访问方式的基础。

Chapter 4

数值型向量及其运算

4.1 数值型向量

向量是将若干个基础类型相同的值存储在一起，各个元素可以按序号访问。如果将若干个数值存储在一起可以用序号访问，就叫做一个数值型向量。

用 `c()` 函数把多个元素或向量组合成一个向量。如

```
marks <- c(10, 6, 4, 7, 8)
x <- c(1:3, 10:13)
x1 <- c(1, 2)
x2 <- c(3, 4)
x <- c(x1, x2)
x
```

```
## [1] 1 2 3 4
```

10:13 这样的写法表示从 10 到 13 的整数组成的向量。

用 `print()` 函数显示向量或在命令行中显示向量时，每行显示的行首会有方括号和数字序号，代表该行显示的第一个向量元素的下标。如

```
12345678901:12345678920
```

```
## [1] 12345678901 12345678902 12345678903 12345678904 12345678905
## [6] 12345678906 12345678907 12345678908 12345678909 12345678910
## [11] 12345678911 12345678912 12345678913 12345678914 12345678915
## [16] 12345678916 12345678917 12345678918 12345678919 12345678920
```

`length(x)` 可以求 `x` 的长度。长度为零的向量表示为 `numeric(0)`。`numeric()` 函数可以用来初始化一个指定元素个数而元素都等于零的数值型向量，如 `numeric(10)` 会生成元素为 10 个零的向量。

4.2 向量运算

4.2.1 标量和标量运算

单个数值称为**标量**，R 没有单独的标量类型，标量实际是长度为 1 的向量。

R 中四则运算用 $+$ $-$ $*$ $/$ $^$ 表示 (加、减、乘、除、乘方)，如

```
1.5 + 2.3 - 0.6 + 2.1*1.2 - 1.5/0.5 + 2^3
```

```
## [1] 10.72
```

R 中四则运算仍遵从通常的优先级规则，可以用圆括号 $()$ 改变运算的先后次序。如

```
1.5 + 2.3 - (0.6 + 2.1)*1.2 - 1.5/0.5 + 2^3
```

```
## [1] 5.56
```

除了加、减、乘、除、乘方，R 还支持整除运算和求余运算。用 $\%/\%$ 表示整除，用 $\%\%$ 表示求余。如

```
5 %/% 3
```

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

```
5 %% 3
```

```
## [1] 2
```

```
5.1 %/% 2.5
```

```
## [1] 2
```

```
5.1 %% 2.5
```

```
## [1] 0.1
```

4.2.2 向量与标量运算

向量与标量的运算为每个元素与标量的运算，如

```
x <- c(1, 10)
```

```
x + 2
```

```
## [1] 3 12
```

```
x - 2
```

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

```
x * 2
```

```
## [1] 2 20
```



```
x / 2  
  
## [1] 0.5 5.0
```

```
x ^ 2  
  
## [1] 1 100
```

```
2 / x  
  
## [1] 2.0 0.2
```

```
2 ^ x  
  
## [1] 2 1024
```

一个向量乘以一个标量，就是线性代数中的数乘运算。

四则运算时如果有缺失值，缺失元素参加的运算相应结果元素仍缺失。如

```
c(1, NA, 3) + 10  
  
## [1] 11 NA 13
```

4.2.3 等长向量运算

等长向量的运算为对应元素两两运算。如

```
x1 <- c(1, 10)  
x2 <- c(4, 2)  
x1 + x2
```

```
## [1] 5 12
```

```
x1 - x2
```

```
## [1] -3 8
```

```
x1 * x2
```

```
## [1] 4 20
```

```
x1 / x2
```

```
## [1] 0.25 5.00
```

两个等长向量的加、减运算就是线性代数中两个向量的加、减运算。

4.2.4 不等长向量的运算

两个不等长向量的四则运算，如果其长度为倍数关系，规则是每次从头重复利用短的一个。如

```
x1 <- c(10, 20)
x2 <- c(1, 3, 5, 7)
x1 + x2
```

```
## [1] 11 23 15 27
```

```
x1 * x2
```

```
## [1] 10 60 50 140
```

不仅是四则运算, R 中有两个或多个向量按照元素一一对应参与某种运算或函数调用时, 如果向量长度不同, 一般都采用这样的规则。

如果两个向量的长度不是倍数关系, 会给出警告信息。如

```
c(1,2) + c(1,2,3)
```

```
## Warning in c(1, 2) + c(1, 2, 3): 长的对象长度不是短的对象长度的整倍数
```

```
## [1] 2 4 4
```

4.3 向量函数

4.3.1 向量化的函数

R 中的函数一般都是向量化的: 在 R 中, 如果普通的一元函数以向量为自变量, 一般会对每个元素计算。这样的函数包括 `sqrt`, `log10`, `log`, `exp`, `sin`, `cos`, `tan` 等许多。如

```
sqrt(c(1, 4, 6.25))
```

```
## [1] 1.0 2.0 2.5
```

为了查看这些基础的数学函数的列表, 运行命令 `help.start()`, 点击链接 “Search Engine and Keywords”, 找到 “Mathematics” 栏目, 浏览其中的 “arith” 和 “math” 链接中的说明。常用的数学函数有:

- 舍入: `ceiling`, `floor`, `round`, `signif`, `trunc`, `zapsmall`
- 符号函数 `sign`
- 绝对值 `abs`
- 平方根 `sqrt`
- 对数与指数函数 `log`, `exp`, `log10`, `log2`
- 三角函数 `sin`, `cos`, `tan`
- 反三角函数 `asin`, `acos`, `atan`, `atan2`
- 双曲函数 `sinh`, `cosh`, `tanh`
- 反双曲函数 `asinh`, `acosh`, `atanh`

有一些不太常用的数学函数:

- 贝塔函数 `beta`, `lbeta`
- 伽玛函数 `gamma`, `lgamma`, `digamma`, `trigamma`, `tetragamma`, `pentagamma`
- 组合数 `choose`, `lchoose`
- 富利叶变换和卷积 `fft`, `mvfft`, `convolve`
- 正交多项式 `poly`
- 求根 `polyroot`, `uniroot`
- 最优化 `optimize`, `optim`
- Bessel 函数 `besselI`, `besselK`, `besselJ`, `besselY`
- 样条插值 `spline`, `splinefun`
- 简单的微分 `deriv`

如果自己编写的函数没有考虑向量化问题，可以用 `Vectorize()` 函数将其转换成向量化版本。

4.3.2 排序函数

`sort(x)` 返回排序结果。`rev(x)` 返回把各元素排列次序反转后的结果。`order(x)` 返回排序用的下标。如

```
x <- c(33, 55, 11)
sort(x)
```

```
## [1] 11 33 55
```

```
rev(sort(x))
```

```
## [1] 55 33 11
```

```
order(x)
```

```
## [1] 3 1 2
```

```
x[order(x)]
```

```
## [1] 11 33 55
```

例子中，`order(x)` 结果中 3 是 `x` 的最小元素 11 所在的位置下标，1 是 `x` 的第二小元素 33 所在的位置下标，2 是 `x` 的最大元素 55 所在的位置下标。

4.3.3 统计函数

`sum`(求和)，`mean`(求平均值)，`var`(求样本方差)，`sd`(求样本标准差)，`min`(求最小值)，`max`(求最大值)，`range`(求最小值和最大值) 等函数称为统计函数，把输入向量看作样本，计算样本统计量。`prod` 求所有元素的乘积。

`cumsum` 和 `cumprod` 计算累加和累乘积。如

```
cumsum(1:5)
```

```
## [1] 1 3 6 10 15
```

```
cumprod(1:5)
```

```
## [1] 1 2 6 24 120
```

其它一些类似函数有 `pmax`, `pmin`, `cummax`, `cummin` 等。

4.3.4 生成规则序列的函数

`seq` 函数是冒号运算符的推广。比如, `seq(5)` 等同于 `1:5`。`seq(2,5)` 等同于 `2:5`。`seq(11, 15, by=2)` 产生 11,13,15。`seq(0, 2*pi, length.out=100)` 产生从 0 到 2π 的等间隔序列, 序列长度指定为 100。

从这些例子可以看出, S 函数可以带自变量名调用。每个函数的变量名和用法可以查询其帮助信息, 在命令行界面用 “? 函数名” 的方法查询。在使用变量名时次序可以颠倒, 比如 `seq(to=5, from=2)` 仍等同于 `2:5`。

`rep()` 函数用来产生重复数值。为了产生一个初值为零的长度为 `n` 的向量, 用 `x <- rep(0, n)`。`rep(c(1,3), 2)` 把第一个自变量重复两次, 结果相当于 `c(1,3,1,3)`。

`rep(c(1,3), c(2,4))` 则需要利用 R 的一般向量化规则, 把第一自变量的第一个元素 1 按照第二自变量中第一个元素 2 的次数重复, 把第一自变量中第二个元素 3 按照第二自变量中第二个元素 4 的次数重复, 结果相当于 `c(1,1,3,3,3,3)`。

如果希望重复完一个元素后再重复另一元素, 用 `each=` 选项, 比如 `rep(c(1,3), each=2)` 结果相当于 `c(1,1,3,3)`。

4.4 复数向量

复数常数表示如 `3.5+2.4i`, `1i`。用函数 `complex()` 生成复数向量, 指定实部和虚部。如 `complex(c(1,0,-1,0), c(0,1,0,-1))` 相当于 `c(1+0i, 1i, -1+0i, -1i)`。

在 `complex()` 中可以用 `mod` 和 `arg` 指定模和辐角, 如 `complex(mod=1, arg=(0:3)/2*pi)` 结果同上。

用 `Re(z)` 求 `z` 的实部, 用 `Im(z)` 求 `z` 的虚部, 用 `Mod(z)` 或 `abs(z)` 求 `z` 的模, 用 `Arg(z)` 求 `z` 的辐角, 用 `Conj(z)` 求 `z` 的共轭。

`sqrt`, `log`, `exp`, `sin` 等函数对复数也有定义, 但是函数定义域在自变量为实数时可能有限制而复数无限制, 这时需要区分自变量类型。如

```
sqrt(-1)
```

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

```
## Warning message:
```

```
## In sqrt(-1) : NaNs produced  
sqrt(-1 + 0i)  
## [1] 0+1i
```

4.5 练习

1. 显示 1 到 100 的整数的平方根和立方根（提示：立方根就是三分之一次方）。
2. 设有 10 个人的小测验成绩为:

77 60 91 73 85 82 35 100 66 75

- (1) 把这 10 个成绩存入变量 `x`;
 - (2) 从小到大排序;
 - (3) 计算 `order(x)`，解释 `order(x)` 结果中第 3 项代表的意义。
 - (4) 计算这些成绩的平均值、标准差、最小值、最大值、中位数。
3. 生成 $[0, 1]$ 区间上等间隔的 100 个格子点存入变量 `x` 中。

Chapter 5

逻辑型向量及其运算

5.1 逻辑型向量与比较运算

逻辑型是 R 的基本数据类型之一，只有两个值 TRUE 和 FALSE, 缺失时为 NA。逻辑值一般产生自比较，如

```
sele <- (log10(15) < 2); print(sele)
```

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

向量比较结果为逻辑型向量。如

```
c(1, 3, 5) > 2
```

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

```
(1:4) >= (4:1)
```

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

从例子可以看出，向量比较也遵从 R 的向量间运算的一般规则：向量与标量的运算是向量每个元素与标量都分别运算一次，等长向量的运算时对应元素的运算，不等长但长度为倍数关系的向量运算是把短的从头重复利用。

与 NA 比较产生 NA，如

```
c(1, NA, 3) > 2
```

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

```
NA == NA
```

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

为了判断向量每个元素是否 NA，用 `is.na()` 函数，如

```
is.na(c(1, NA, 3) > 2)
```

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

用 `is.finite()` 判断向量每个元素是否 `Inf` 值。

比较运算符包括

```
< <= > >= == != %in%
```

分别表示小于、小于等于、大于、大于等于、等于、不等于、属于。要注意等于比较用了两个等号。

`%in%` 是比较特殊的比较, `x %in% y` 的运算把向量 `y` 看成集合, 运算结果是一个逻辑型向量, 第 i 个元素的值为 `x` 的第 i 元素是否属于 `y` 的逻辑型值。如

```
c(1,3) %in% c(2,3,4)
```

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

```
c(NA,3) %in% c(2,3,4)
```

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

```
c(1,3) %in% c(NA, 3, 4)
```

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

```
c(NA,3) %in% c(NA, 3, 4)
```

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

函数 `match(x, y)` 起到和 `x %in% y` 运算类似的作用, 但是其返回结果不是找到与否, 而是对 `x` 的每个元素, 找到其在 `y` 中首次出现的下标, 找不到时取缺失值, 如

```
match(c(1, 3), c(2,3,4,3))
```

```
## [1] NA 2
```

5.2 逻辑运算

为了表达如“ $x > 0$ 而且 $x < 1$ ”, “ $x \leq 0$ 或者 $x \geq 1$ ”之类的复合比较, 需要使用逻辑运算把两个比较连接起来。逻辑运算符为 `&`, `|` 和 `!`, 分别表示“同时成立”、“两者至少其一成立”、“条件的反面”。比如, 设 `age<=3` 表示婴儿, `sex=='女'` 表示女性, 则 `age<=3 & sex=='女'` 表示女婴, `age<=3 | sex=='女'` 表示婴儿或妇女, `!(age<=3 | sex=='女')` 表示既非婴儿也非妇女。为了确定运算的先后次序可以用圆括号 `()` 指定。

用 `xor(x, y)` 表示 `x` 与 `y` 的异或运算, 即值不相等时为真值, 相等时为假值, 有缺失值参加运算时为缺失值。

逻辑向量与逻辑标量之间的逻辑运算，两个逻辑向量之间的逻辑运算规则遵从一般 R 向量间运算规则。

在右运算符是缺失值时，如果左运算符能够确定结果真假，可以得到非缺失的结果。例如，`TRUE | NA` 为 `TRUE`，`FALSE & NA` 为 `FALSE`。不能确定结果时返回 `NA`，比如，`TRUE & NA` 为 `NA`，`FALSE | NA` 为 `NA`。

`&&` 和 `||` 分别为短路的标量逻辑与和短路的标量逻辑或，仅对两个标量进行运算，如果有向量也仅使用第一个元素。一般用在 `if` 语句、`while` 语句中，且只要第一个比较已经决定最终结果就不计算第二个比较。例如

```
if(TRUE || sqrt(-1)>0) next
```

其中的 `sqrt(-1)` 部分不会执行。

这里结果为 `TRUE`，第二部分没有参加计算，否则第二部分的计算会发生函数自变量范围错误。

5.3 逻辑运算函数

因为 R 中比较与逻辑运算都支持向量之间、向量与标量之间的运算，所以在需要一个标量结果时要特别注意，后面讲到的 `if` 结构、`while` 结构都需要逻辑标量而且不能是缺失值。这时，应该对缺失值结果单独考虑。

若 `cond` 是逻辑向量，用 `all(cond)` 测试 `cond` 的所有元素为真；用 `any(cond)` 测试 `cond` 至少一个元素为真。`cond` 中允许有缺失值，结果可能为缺失值。如

```
c(1, NA, 3) > 2
```

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

```
all(c(1, NA, 3) > 2)
```

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

```
any(c(1, NA, 3) > 2)
```

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

```
all(NA)
```

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

```
any(NA)
```

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

函数 `which()` 返回真值对应的所有下标，如

```
which(c(FALSE, TRUE, TRUE, FALSE, NA))
```

```
## [1] 2 3
```

```
which((11:15) > 12)
```

```
## [1] 3 4 5
```

函数 `identical(x,y)` 比较两个 R 对象 `x` 与 `y` 的内容是否完全相同，结果只会取标量 `TRUE` 与 `FALSE` 两种。如

```
identical(c(1,2,3), c(1,2,NA))
```

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

```
identical(c(1L,2L,3L), c(1,2,3))
```

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

其中第二个结果假值是因为前一向量是整数型，后一向量是实数型。

函数 `all.equal()` 与 `identical()` 类似，但是在比较数值型时不区分整数型与实数型，而且相同时返回标量 `TRUE`，但是不同时会返回一个说明有何不同的字符串。如

```
all.equal(c(1,2,3), c(1,2,NA))
```

```
## [1] "'is.NA' value mismatch: 1 in current 0 in target"
```

```
all.equal(c(1L,2L,3L), c(1,2,3))
```

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

函数 `duplicated()` 返回每个元素是否为重复值的结果，如：

```
duplicated(c(1,2,1,3,NA,4,NA))
```

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

用函数 `unique()` 可以返回去掉重复值的结果。

Chapter 6

字符型数据及其处理

6.1 字符型向量

字符型向量是元素为字符串的向量。如

```
s1 <- c('abc', '', 'a cat', NA, ' 李明')
```

注意空字符串并不能自动认为是缺失值，字符型的缺失值仍用 `NA` 表示。

6.2 paste() 函数

针对字符型数据最常用的 R 函数是 `paste()` 函数。`paste()` 用来连接两个字符型向量，元素一一对应连接，默认用空格连接。如 `paste(c("ab", "cd"), c("ef", "gh"))` 结果相当于 `c("ab ef", "cd gh")`。

`paste()` 在连接两个字符型向量时采用 R 的一般向量间运算规则，而且可以自动把数值型向量转换为字符型向量。可以作一对多连接，如 `paste("x", 1:3)` 结果相当于 `c("x 1", "x 2", "x 3")`。

用 `sep=` 指定分隔符，如 `paste("x", 1:3, sep="")` 结果相当于 `c("x1", "x2", "x3")`。

使用 `collapse=` 参数可以把字符型向量的各个元素连接成一个单一的字符串，如 `paste(c("a", "b", "c"), collapse="")` 结果相当于 `"abc"`。

6.3 转换大小写

`toupper()` 函数把字符型向量内容转为大写，`tolower()` 函数转为小写。比如，`toupper('aB cd')` 结果为 `"AB CD"`，`tolower(c('aB', 'cd'))` 结果相当于 `c("ab" "cd")`。这两个函数可以用于不区分大小写的比较，比如，不论 `x` 的值是 `'JAN'`，`'Jan'` 还是 `'jan'`，`toupper(x)=='JAN'` 的结果都为 `TRUE`。

6.4 字符串长度

用 `nchar(x, type='bytes')` 计算字符型向量 `x` 中每个字符串的以字节为单位的长度, 这一点对中英文是有差别的, 中文通常一个汉字占两个字节, 英文字母、数字、标点占一个字节。用 `nchar(x, type='chars')` 计算字符型向量 `x` 中每个字符串的以字符个数为单位的长度, 这时一个汉字算一个单位。

在画图时可以用 `strwidth()` 函数计算某个字符串或表达式占用的空间大小。

6.5 取子串

`substr(x, start, stop)` 从字符串 `x` 中取出从第 `start` 个到第 `stop` 个的子串, 如

```
substr('JAN07', 1, 3)
```

```
## [1] "JAN"
```

如果 `x` 是一个字符型向量, `substr` 将对每个元素取子串。如

```
substr(c('JAN07', 'MAR66'), 1, 3)
```

```
## [1] "JAN" "MAR"
```

用 `substring(x, start)` 可以从字符串 `x` 中取出从第 `start` 个到末尾的子串。如

```
substring(c('JAN07', 'MAR66'), 4)
```

```
## [1] "07" "66"
```

6.6 类型转换

用 `as.numeric()` 把内容是数字的字符型值转换为数值, 如

```
substr('JAN07', 4, 5)
```

```
## [1] "07"
```

```
substr('JAN07', 4, 5) + 2000
```

```
## Error in substr("JAN07", 4, 5) + 2000 :
```

```
## non-numeric argument to binary operator
```

```
as.numeric(substr('JAN07', 4, 5)) + 2000
```

```
## [1] 2007
```

```
as.numeric(substr(c('JAN07', 'MAR66'), 4, 5))
```

```
## [1] 7 66
```

`as.numeric()` 是向量化的, 可以转换一个向量的每个元素为数值型。

用 `as.character()` 函数把数值型转换为字符型, 如

```
as.character((1:5)*5)
```

```
## [1] "5" "10" "15" "20" "25"
```

如果自变量本来已经是字符型则结果不变。

为了用指定的格式数值型转换成字符型，可以使用 `sprintf()` 函数，其用法与 C 语言的 `sprintf()` 函数相似，只不过是向量化的。例如

```
sprintf('file%03d.txt', c(1, 99, 100))
```

```
## [1] "file001.txt" "file099.txt" "file100.txt"
```

6.7 字符串拆分

用 `strsplit()` 函数可以把一个字符串按照某种分隔符拆分开，例如

```
x <- '10,8,7'
strsplit(x, ',', fixed=TRUE)[[1]]
```

```
## [1] "10" "8" "7"
```

```
sum(as.numeric(strsplit(x, ',', fixed=TRUE)[[1]]))
```

```
## [1] 25
```

因为 `strsplit()` 的结果是一个列表，这个函数延后再详细讲。

6.8 字符串替换功能

用 `gsub()` 可以替换字符串中的子串，这样的功能经常用在数据清理中。比如，把数据中的中文标点改为英文标点，去掉空格，等等。如

```
x <- '1, 3; 5'
gsub(';', ',', x, fixed=TRUE)
```

```
## [1] "1, 3, 5"
```

```
strsplit(gsub(';', ',', x, fixed=TRUE), ',')[[1]]
```

```
## [1] "1" "3" "5"
```

字符串 `x` 中分隔符既有逗号又有分号，上面的程序用 `gsub()` 把分号都换成逗号。

6.9 正则表达式

正则表达式 (regular expression) 是一种匹配某种字符串模式的方法。用这样的方法，可以从字符串中查找某种模式的出现位置，替换某种模式，等等。这样的技术可以用于文本数据的预处理，比如用网络爬虫下载的大量网页文本数据。R 中支持 perl 语言格式的正则表达式，`grep()` 和 `grepl()` 函数从字符串中查询某个模式，`sub()` 和 `gsub()` 替换某模式。比如，下面的程序把多于一个空格替换成一个空格

```
gsub('[:space:]+', ' ', 'a   cat   in a box', perl=TRUE)
```

```
## [1] "a cat in a box"
```

正则表达式功能强大但也不容易掌握。详细讲解延后处理。

Chapter 7

R 向量下标和子集

在 R 中下标与子集是极为强大的功能，需要一些练习才能熟练掌握，许多其它语言中需要多个语句才能完成的工作在 R 中都可以简单地通过下标和子集来完成。

7.1 正整数下标

对向量 `x`，在后面加方括号和下标可以访问向量的元素和子集。

设 `x <- c(1, 4, 6.25)`。`x[2]` 取出第二个元素；`x[2] <- 99` 修改第二个元素。`x[c(1,3)]` 取出第 1、3 号元素；`x[c(1,3)] <- c(11, 13)` 修改第 1、3 号元素。下标可重复。例如

```
x <- c(1, 4, 6.25)
x[2]
```

```
## [1] 4
```

```
x[2] <- 99; x
```

```
## [1] 1.00 99.00 6.25
```

```
x[c(1,3)]
```

```
## [1] 1.00 6.25
```

```
x[c(1,3)] <- c(11, 13); x
```

```
## [1] 11 99 13
```

```
x[c(1,3,1)]
```

```
## [1] 11 13 11
```

7.2 负整数下标

负下标表示扣除相应的元素后的子集，如

```
x <- c(1,4,6.25)
x[-2]
```

```
## [1] 1.00 6.25
```

```
x[-c(1,3)]
```

```
## [1] 4
```

负整数下标不能与正整数下标同时用来从某一向量中取子集，比如，`x[c(1,-2)]` 没有意义。

7.3 空下标与零下标

`x[]` 表示取 `x` 的全部元素作为子集。这与 `x` 本身不同，比如

```
x <- c(1,4,6.25)
x[] <- 999
x
```

```
## [1] 999 999 999
```

```
x <- c(1,4,6.25)
x <- 999
x
```

```
## [1] 999
```

`x[0]` 是一种少见的做法，结果返回类型相同、长度为零的向量，如 `numeric(0)`。相当于空集。

当 0 与正整数下标一起使用时会被忽略。当 0 与负整数下标一起使用时也会被忽略。

7.4 下标超界

设向量 `x` 长度为 n ，则使用正整数下标时下标应在 $\{1, 2, \dots, n\}$ 中取值。如果使用大于 n 的下标，读取时返回缺失值，并不出错。给超出 n 的下标元素赋值，则向量自动变长，中间没有赋值的元素为缺失值。例如

```
x <- c(1,4,6.25)
x[5]
```

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



```
x

## [1] 1.00 4.00 6.25

x[5] <- 9

x

## [1] 1.00 4.00 6.25 NA 9.00
```

虽然 R 的语法对下标超界不视作错误，但是这样的做法往往来自不良的程序思路，而且对程序效率有影响，所以实际编程中应避免下标超界。

7.5 逻辑下标

下标可以是与向量等长的逻辑表达式，一般是关于本向量或者与本向量等长的其它向量的比较结果，如

```
x <- c(1,4,6.25)
x[x > 3]
```

```
## [1] 4.00 6.25
```

取出 x 的大于 3 的元素组成的子集。

逻辑下标除了用来对向量取子集，还经常用来对数据框取子集，也用在向量化的运算中。例如，对如下示性函数

$$f(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

输入向量 x，结果 y 需要也是一个向量，程序可以写成

```
f <- function(x){
  y <- numeric(length(x))
  y[x >= 0] <- 1
  y[x < 0] <- 0 # 此语句多余
}
```

事实上，向量化的逻辑选择有一个 `ifelse()` 函数，比如，对上面的示性函数，如果 x 是一个向量，输出 y 向量可以写成 `y <- ifelse(x>=0, 1, 0)`。

要注意的是，如果逻辑下标中有缺失值，对应结果也是缺失值。所以，在用逻辑下标作子集选择时，一定要考虑到缺失值问题。正确的做法是加上 `!is.na` 前提，如

```
x <- c(1, 4, 6.25, NA)
x[x > 2]

## [1] 4.00 6.25 NA
```

```
x[!is.na(x) & x > 2]
```

```
## [1] 4.00 6.25
```

7.6 which()、which.min()、which.max() 函数

函数 `which()` 可以用来找到满足条件的下标，如

```
x <- c(3, 4, 3, 5, 7, 5, 9)
```

```
which(x > 5)
```

```
## [1] 5 7
```

```
seq(along=x)[x > 5]
```

```
## [1] 5 7
```

这里 `seq(along=x)` 会生成由 `x` 的下标组成的向量。用 `which.min()`、`which.max` 求最小值的下标和最大值的下标，不唯一时只取第一个。如

```
which.min(x)
```

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

```
which.max(x)
```

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

7.7 元素名

向量可以为每个元素命名。如

```
ages <- c("李明"=30, "张聪"=25, "刘颖"=28)
```

或

```
ages <- c(30, 25, 28)
```

```
names(ages) <- c("李明", "张聪", "刘颖")
```

或

```
ages <- setNames(c(30, 25, 28), c("李明", "张聪", "刘颖"))
```

这时可以用元素名或元素名向量作为向量的下标，如

```
ages["张聪"]
```

```
## 张聪
##    25

ages[c("李明", "刘颖")]
```

```
## 李明 刘颖
##    30  28
```

```
ages["张聪"] <- 26
```

这实际上建立了字符串到数值的映射表。

用字符串作为下标时，如果该字符串不在向量的元素名中，读取时返回缺失值结果，赋值时该向量会增加一个元素并以该字符串为元素名。

带有元素名的向量也可以是字符型或其它基本类型，如

```
sex <- c("李明"="男", "张聪"="男", "刘颖"="女")
```

除了给向量元素命名外，在矩阵和数据框中还可以给行、列命名，这会使得程序的扩展更为容易和安全。

R 允许仅给部分元素命名，这时其它元素名字为空字符串。不同元素的元素名一般应该是不同的，否则在使用元素作为下标时会发生误读，但是 R 语法允许存在重名。

用 `unname(x)` 返回去掉了元素名的 `x` 的副本，用 `names(x) <- NULL` 可以去掉 `x` 的元素名。

7.8 用 R 向量下标作映射

R 在使用整数作为向量下标时，允许使用重复下标，这样可以把数组 `x` 看成一个 $1:n$ 的整数到 `x[1]`, `x[2]`, ..., `x[n]` 的一个映射表，其中 n 是 `x` 的长度。比如，某商店有三种礼品，编号为 1,2,3，价格分别为 68, 88 和 168。令

```
price.map <- c(68, 88, 168)
```

设某个收银员在一天内分别售出礼品编号为 3,2,1,1,2,2,3，可以用如下的映射方式获得售出的这些礼品对应的价格：

```
items <- c(3,2,1,1,2,2,3)
y <- price.map[items]; print(y)
```

```
## [1] 168 88 68 68 88 88 168
```

R 向量可以用字符型向量作下标，字符型下标也允许重复，所以可以把带有元素名的 R 向量看成是元素名到元素值的映射表。比如，设 `sex` 为 10 个学生的性别（男、女）

```
sex <- c("男", "男", "女", "女", "男", "女", "女", "女", "女", "男")
```

希望把每个学生按照性别分别对应到蓝色和红色。首先建立一个 R 向量当作映射

```
sex.color <- c('男'='blue', '女'='red')
```

用 R 向量 `sex.color` 当作映射，可以获得每个学生对应的颜色

```
cols <- sex.color[sex]; print(cols)
```

```
##      男      男      女      女      男      女      女      女      女      男
## "blue" "blue" "red" "red" "blue" "red" "red" "red" "red" "blue"
```

这样的映射结果中带有不必要的元素名，用 `unnname()` 函数可以去掉元素名，如

```
unnname(cols)
```

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

7.9 集合运算

可以把向量 `x` 看成一个集合，但是其中的元素允许有重复。用 `unique(x)` 可以获得 `x` 的所有不同值。如

```
unique(c(1, 5, 2, 5))
```

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

用 `a %in% x` 判断 `a` 的每个元素是否属于向量 `x`，如

```
5 %in% c(1,5,2)
```

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

```
c(5,6) %in% c(1,5,2)
```

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

与 `%in%` 运算符类似，函数 `match(x, table)` 对向量 `x` 的每个元素，从向量 `table` 中查找其首次出现位置并返回这些位置。没有匹配到的元素位置返回 `NA_integer_` (整数型缺失值)。如

```
match(5, c(1,5,2))
```

```
## [1] 2
```

```
match(5, c(1,5,2,5))
```

```
## [1] 2
```

```
match(c(2,5), c(1,5,2,5))
```

```
## [1] 3 2
```

```
match(c(2,5,0), c(1,5,2,5))
```

```
## [1] 3 2 NA
```

用 `intersect(x,y)` 求交集，结果中不含重复元素，如

```
intersect(c(5, 7), c(1, 5, 2, 5))
```

```
## [1] 5
```

用 `union(x,y)` 求并集，结果中不含重复元素，如

```
union(c(5, 7), c(1, 5, 2, 5))
```

```
## [1] 5 7 1 2
```

用 `setdiff(x,y)` 求差集，即 `x` 的元素中不属于 `y` 的元素组成的集合，结果中不含重复元素，如

```
setdiff(c(5, 7), c(1, 5, 2, 5))
```

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

用 `setequal(x,y)` 判断两个集合是否相等，不受次序与重复元素的影响，如

```
setequal(c(1,5,2), c(2,5,1))
```

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

```
setequal(c(1,5,2), c(2,5,1,5))
```

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

7.10 练习

设文件 `class.csv` 内容如下：

```
name,sex,age,height,weight
Alice,F,13,56.5,84
Becka,F,13,65.3,98
Gail,F,14,64.3,90
Karen,F,12,56.3,77
Kathy,F,12,59.8,84.5
Mary,F,15,66.5,112
Sandy,F,11,51.3,50.5
Sharon,F,15,62.5,112.5
Tammy,F,14,62.8,102.5
Alfred,M,14,69,112.5
Duke,M,14,63.5,102.5
Guido,M,15,67,133
James,M,12,57.3,83
Jeffrey,M,13,62.5,84
```

```
John,M,12,59,99.5  
Philip,M,16,72,150  
Robert,M,12,64.8,128  
Thomas,M,11,57.5,85  
William,M,15,66.5,112
```

用如下程序可以把上述文件读入为 R 数据框 d.class, 并取出其中的 name 和 age 列到变量 name 和 age 中:

```
d.class <- read.csv('class.csv', header=TRUE, stringsAsFactors=FALSE)  
name <- d.class[, 'name']  
age <- d.class[, 'age']
```

- (1) 求出 age 中第 3, 5, 7 号的值;
- (2) 用变量 age, 求出达到 15 岁及以上的那些值;
- (3) 用变量 name 和 age, 求出 Mary 与 James 的年龄。
- (4) 求 age 中除 Mary 与 James 这两人之外的那些人的年龄值, 保存到变量 age1 中。
- (5) 假设向量 x 长度为 n , 其元素是 $\{1, 2, \dots, n\}$ 的一个重排。可以把 x 看成一个 i 到 $x[i]$ 的映射 (i 在 $\{1, 2, \dots, n\}$ 中取值)。求向量 y , 保存了上述映射的逆映射, 即: 如果 $x[i]=j$, 则 $y[j]=i$ 。

Chapter 8

R 数据类型的性质

8.1 存储模式与基本类型

R 的变量可以存储多种不同的数据类型，可以用 `typeof()` 函数来返回一个变量或表达式的类型。比如

```
typeof(1:3)
```

```
## [1] "integer"
```

```
typeof(c(1,2,3))
```

```
## [1] "double"
```

```
typeof(c(1, 2.1, 3))
```

```
## [1] "double"
```

```
typeof(c(TRUE, NA, FALSE))
```

```
## [1] "logical"
```

```
typeof('Abc')
```

```
## [1] "character"
```

```
typeof(factor(c('F', 'M', 'M', 'F')))
```

```
## [1] "integer"
```

注意因子的结果是 `integer` 而不是因子。函数 `mode()` 和 `storage.mode()` 以及 `typeof()` 类似，但返回结果有差别。这三个函数都是与存储类型有关，不依赖于变量和表达式的实际作用。

R 中数据的最基本的类型包括 `logical`, `integer`, `double`, `character`, `complex`, `raw`, 其它数据类型都是由基本类型组合或转变得到的。`character` 类型就是字符串类型，`raw` 类型是直接使用其二进制内

容的类型。为了判断某个向量 `x` 保存的基本类型，可以用 `is.xxx()` 类函数，如 `is.integer(x)`, `is.double(x)`, `is.numeric(x)`, `is.logical(x)`, `is.character(x)`, `is.complex(x)`, `is.raw(x)`。其中 `is.numeric(x)` 对 `integer` 和 `double` 内容都返回真值。

在 R 语言中数值一般看作 `double`，如果需要明确表明某些数值是整数，可以在数值后面附加字母 `L`，如

```
is.integer(c(1, -3))
```

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

```
is.integer(c(1L, -3L))
```

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

整数型的缺失值是 `NA`，而 `double` 型的特殊值除了 `NA` 外，还包括 `Inf`, `-Inf` 和 `NaN`，其中 `NaN` 也算是缺失值，`Inf` 和 `-Inf` 不算是缺失值。如：

```
c(-1, 0, 1)/0
```

```
## [1] -Inf NaN Inf
```

```
is.na(c(-1, 0, 1)/0)
```

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

对 `double` 类型，可以用 `is.finite()` 判断是否有限值，`NA`、`Inf`、`-Inf` 和 `NaN` 都不是有限值；用 `is.infinite()` 判断是否 `Inf` 或 `-Inf`；`is.na()` 判断是否 `NA` 或 `NaN`；`is.nan()` 判断是否 `NaN`。

严格说来，`NA` 表示逻辑型缺失值，但是当作其它类型缺失值时一般能自动识别。`NA_integer_` 是整数型缺失值，`NA_real_` 是 `double` 型缺失值，`NA_character_` 是字符型缺失值。

在 R 的向量类型中，`integer` 类型、`double` 类型、`logical` 类型、`character` 类型、还有 `complex` 类型和 `raw` 类型称为原子类型 (atomic types)，原子类型的向量中元素都是同一基本类型的。比如，`double` 型向量的元素都是 `double` 或者缺失值。除了原子类型的向量，在 R 语言的定义中，向量还包括后面要讲到的列表 (list)，列表的元素不需要属于相同的基本类型，而且列表的元素可以不是单一基本类型元素。用 `typeof()` 函数可以返回向量的类型，列表返回结果为 `"list"`：

```
typeof(list("a", 1L, 1.5))
```

```
## [1] "list"
```

原子类型的各个元素除了基本类型相同，还不包含任何嵌套结构，如：

```
c(1, c(2,3, c(4,5)))
```

```
## [1] 1 2 3 4 5
```


8.2 类属

R 具有一定的面向对象语言特征，其数据类型有一个 `class` 属性，函数 `class()` 可以返回变量类型的类属，比如

```
typeof(factor(c('F', 'M', 'M', 'F')))  
## [1] "integer"  
mode(factor(c('F', 'M', 'M', 'F')))  
## [1] "numeric"  
storage.mode(factor(c('F', 'M', 'M', 'F')))  
## [1] "integer"  
class(factor(c('F', 'M', 'M', 'F')))  
## [1] "factor"  
class(as.numeric(factor(c('F', 'M', 'M', 'F'))))  
## [1] "numeric"
```

R 有一个特殊的 `NULL` 类型，这个类型只有唯一的一个 `NULL` 值，表示不存在。要把 `NULL` 与 `NA` 区分开来，`NA` 是有类型的 (`integer`、`double`、`logical`、`character` 等)，`NA` 表示存在但是未知。用 `is.null()` 函数判断某个变量是否取 `NULL`。

8.3 类型转换

可以用 `as.xxx()` 类的函数在不同类型之间进行强制转换。如

```
as.numeric(c(FALSE, TRUE))  
## [1] 0 1  
as.character(sqrt(1:4))  
## [1] "1" "1.4142135623731" "1.73205080756888"  
## [4] "2"
```

类型转换也可能是隐含的，比如，四则运算中数值会被统一转换为 `double` 类型，逻辑运算中运算元素会被统一转换为 `logical` 类型。逻辑值转换成数值时，`TRUE` 转换成 1，`FALSE` 转换成 0。

在用 `c()` 函数合并若干元素时，如果元素基本类型不同，将统一转换成最复杂的一个，复杂程度从简单到复杂依次为：`logical<integer<double<character`。如

```
c(FALSE, 1L, 2.5, "3.6")
```

```
## [1] "FALSE" "1"      "2.5"    "3.6"
```

不同类型参与要求类型相同的运算时，也会统一转换为最复杂的类型，如：

```
TRUE + 10
```

```
## [1] 11
```

```
paste("abc", 1)
```

```
## [1] "abc 1"
```

8.4 属性

除了 NULL 以外，R 的变量都可以看成是对象，都可以有属性。在 R 语言中，属性是把变量看成对象后，除了其存储内容（如元素）之外的其它附加信息，如维数、类属等。对象 `x` 的所有属性可以用 `attributes()` 读取，如

```
x <- table(c(1,2,1,3,2,1)); print(x)
```

```
##
```

```
## 1 2 3
```

```
## 3 2 1
```

```
attributes(x)
```

```
## $dim
```

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

```
##
```

```
## $dimnames
```

```
## $dimnames[[1]]
```

```
## [1] "1" "2" "3"
```

```
##
```

```
##
```

```
## $class
```

```
## [1] "table"
```

`table()` 函数用了输出其自变量中每个不同值的出现次数，称为频数。从上例可以看出，`table()` 函数的结果有三个属性，前两个是 `dim` 和 `dimnames`，这是数组 (array) 具有的属性；另一个是 `class` 属性，值为 "table"。因为 `x` 是数组，可以访问如

```
x[1]
```

```
## 1
```

```
## 3
```

```
x["3"]
```

```
## 3
```

```
## 1
```

也可以用 `attributes()` 函数修改属性，如

```
attributes(x) <- NULL
```

```
x
```

```
## [1] 3 2 1
```

如上修改后 `x` 不再是数组，也不是 `table`。

`class` 属性是特殊的。如果一个对象具有 `class` 属性，某些所谓“通用函数 (generic functions)”会针对这样的对象进行专门的操作，比如，`print()` 函数在显示向量和回归结果时采用完全不同的格式。这在其它程序设计语言中称为“重载”(overloading)。

可以用 `attr(对象, " 属性名")` 读取和修改单个属性。向量的元素名是 `names` 属性，例如

```
ages <- c(" 李明"=30, " 张聪"=25, " 刘颖"=28)
```

```
names(ages)
```

```
## [1] "李明" "张聪" "刘颖"
```

```
attr(ages, "names")
```

```
## [1] "李明" "张聪" "刘颖"
```

```
attr(ages, "names") <- NULL
```

```
ages
```

```
## [1] 30 25 28
```

还可以用 `unname()` 函数返回一个去掉了 `names` 属性的副本。

8.5 str() 函数

用 `print()` 函数可以显示对象内容。如果内容很多，显示行数可能也很多。用 `str()` 函数可以显示对象的类型和主要结构及典型内容。例如

```
s <- 101:200
```

```
attr(s, 'author') <- ' 李小明'
```

```
attr(s, 'date') <- '2016-09-12'
```

```
str(s)
```

```
## atomic [1:100] 101 102 103 104 105 106 107 108 109 110 ...
## - attr(*, "author")= chr "李小明"
## - attr(*, "date")= chr "2016-09-12"
```

现在版本 (R 3.3.1) 的 `str()` 函数有不一致性问题。如

```
x <- c(2, 3)
attr(x, 'a') <- 1111
attr(x, 'b') <- 9999
str(x)
```

```
## atomic [1:2] 2 3
## - attr(*, "a")= num 1111
## - attr(*, "b")= num 9999
```

```
y <- c(5, 7)
names(y) <- c('u', 'v')
str(y)
```

```
## Named num [1:2] 5 7
## - attr(*, "names")= chr [1:2] "u" "v"
```

```
attr(y, 'f') <- 8888
str(y)
```

```
## atomic [1:2] 5 7
## - attr(*, "f")= num 8888
```

当有名向量 `y` 没有定义属性时, `str()` 正确将其识别为 `Named num`, 并显示其 `names` 属性; 但是, 当 `y` 增加了别的属性时, `str()` 就将其识别为 `atomic`, 不再显示其 `names` 属性。

8.6 关于赋值

要注意的是, 在 R 中赋值本质上是把一个存储的对象与一个变量名联系在一起 (binding), 多个变量名可以指向同一个对象。对于基本的数据类型如数值型向量, 两个指向相同对象的变量当一个变量被修改时自动制作副本, 如

```
x <- 1:5
y <- x
y[3] <- 0
x
```

```
## [1] 1 2 3 4 5
```

```
y
```

```
## [1] 1 2 0 4 5
```

这里如果 `y` 没有与其它变量指向同一对象，则修改时直接修改该对象而不制作副本。

但是对于有些比较复杂的类型，两个指向同一对象的变量是同步修改的。这样的类型的典型代表是闭包 (closure)，它带有一个环境，环境的内容是不自动制作副本的。

Chapter 9

R 日期时间

9.1 R 日期和日期时间类型

R 日期可以保存为 Date 类型，一般用整数保存，数值为从 1970-1-1 经过的天数。

R 中用一种叫做 POSIXct 和 POSIXlt 的特殊数据类型保存日期和时间，可以仅包含日期部分，也可以同时有日期和时间。技术上，POSIXct 把日期时间保存为从 1970 年 1 月 1 日零时到该日期时间的时间间隔秒数，所以数据框中需要保存日期时用 POSIXct 比较合适，需要显示时再转换成字符串形式；POSIXlt 把日期时间保存为一个包含年、月、日、星期、时、分、秒等成分列表，所以求这些成分可以从 POSIXlt 格式日期的列表变量中获得。日期时间会涉及到所在时区、夏时制等问题，比较复杂。

基础的 R 用 `as.Date()`、`as.POSIXct()` 等函数生成日期型和日期时间型，R 扩展包 `lubridate` 提供了多个方便函数，可以更容易地生成、转换、管理日期型和日期时间型数据。

```
library(lubridate)
```

```
## 载入需要的程辑包: methods
```

```
##
```

```
## 载入程辑包: 'lubridate'
```

```
## The following object is masked from 'package:base':
```

```
##
```

```
##      date
```

9.2 从字符串生成日期数据

函数 `lubridate::today()` 返回当前日期：

```
today()
```

```
## [1] "2018-03-07"
```

函数 `lubridate::now()` 返回当前日期时间:

```
now()
```

```
## [1] "2018-03-07 13:20:25 CST"
```

结果显示中出现的 `CST` 是时区, 这里使用了操作系统提供的当前时区。`CST` 不是一个含义清晰的时区, 在不同国家对应不同的时区, 在中国代表中国标准时间 (北京时间)。

用 `lubridate::ymd()`, `lubridate::mdy()`, `lubridate::dmy()` 将字符型向量转换为日期型向量, 如:

```
ymd(c("1998-3-10", "2018-01-17", "18-1-17"))
```

```
## [1] "1998-03-10" "2018-01-17" "2018-01-17"
```

```
mdy(c("3-10-1998", "01-17-2018"))
```

```
## [1] "1998-03-10" "2018-01-17"
```

```
dmy(c("10-3-1998", "17-01-2018"))
```

```
## [1] "1998-03-10" "2018-01-17"
```

在年号只有两位数字时, 默认对应到 1969-2068 范围。

`lubridate::make_date(year, month, day)` 可以从三个数值构成日期向量。如

```
make_date(1998, 3, 10)
```

```
## [1] "1998-03-10"
```

`lubridate` 包的 `ymd`、`mdy`、`dmy` 等函数添加 `hms`、`hm`、`h` 等后缀, 可以用于将字符串转换成日期时间。如

```
ymd_hms("1998-03-16 13:15:45")
```

```
## [1] "1998-03-16 13:15:45 UTC"
```

结果显示中 `UTC` 是时区, `UTC` 是协调世界时 (Universal Time Coordinated) 英文缩写, 是由国际无线电咨询委员会规定和推荐, 并由国际时间局 (BIH) 负责保持的以秒为基础的时间标度。`UTC` 相当于本初子午线 (即经度 0 度) 上的平均太阳时, 过去曾用格林威治平均时 (`GMT`) 来表示。北京时间比 `UTC` 时间早 8 小时, 以 1999 年 1 月 1 日 0000`UTC` 为例, `UTC` 时间是零点, 北京时间为 1999 年 1 月 1 日早上 8 点整。

在 `Date()`、`as.DateTime()`、`ymd()` 等函数中, 可以用 `tz=` 指定时区, 比如北京时间可指定为 `tz="Etc/GMT+8"` 或 `tz="Asia/Shanghai"`。

`lubridate::make_datetime(year, month, day, hour, min, sec)` 可以从最多六个数值组成日期时间, 其中时分秒缺省值都是 0。如


```
make_datetime(1998, 3, 16, 13, 15, 45.2)
```

```
## [1] "1998-03-16 13:15:45 UTC"
```

用 `lubridate::as_date()` 可以将日期时间型转换为日期型，如

```
as_date(as.POSIXct("1998-03-16 13:15:45"))
```

```
## [1] "1998-03-16"
```

用 `lubridate::as_datetime()` 可以将日期型数据转换为日期时间型，如

```
as_datetime(as.Date("1998-03-16"))
```

```
## [1] "1998-03-16 UTC"
```

9.3 日期显示格式

用 `as.character()` 函数把日期型数据转换为字符型，如

```
x <- as.POSIXct(c('1998-03-16', '2015-11-22'))
as.character(x)
```

```
## [1] "1998-03-16" "2015-11-22"
```

在 `as.character()` 中可以用 `format` 选项指定显示格式，如

```
as.character(x, format='%m/%d/%Y')
```

```
## [1] "03/16/1998" "11/22/2015"
```

格式中 “%Y” 代表四位的公元年号，“%m” 代表两位的月份数字，“%d” 代表两位的月内日期号。

"15Mar98" 这样的日期在英文环境中比较常见，但是在 R 中的处理比较复杂。在下面的例子中，R 日期被转换成了类似 "Mar98" 这样的格式，在 `format` 选项中用了 “%b” 代表三英文字母月份缩写，但是因为月份缩写依赖于操作系统默认语言环境，需要用 `Sys.setlocale()` 函数设置语言环境为 "C"。示例程序如下

```
x <- as.POSIXct(c('1998-03-16', '2015-11-22'))
old.lctime <- Sys.getlocale('LC_TIME')
Sys.setlocale('LC_TIME', 'C')
```

```
## [1] "C"
```

```
as.character(x, format='%b%y')
```

```
## [1] "Mar98" "Nov15"
```

```
Sys.setlocale('LC_TIME', old.lctime)
```

```
## [1] "Chinese (Simplified)_China.936"
```

`format` 选项中的 “%y” 表示两位数的年份，应尽量避免使用两位数年份以避免混淆。

包含时间的转换如

```
x <- as.POSIXct('1998-03-16 13:15:45')
as.character(x)
```

```
## [1] "1998-03-16 13:15:45"
```

```
as.character(x, format='%H:%M:%S')
```

```
## [1] "13:15:45"
```

这里 “%H” 代表小时（按 24 小时制），“%M” 代表两位的分钟数字，“%S” 代表两位的秒数。

9.4 访问日期时间的组成值

`lubridate` 包的如下函数可以取出日期型或日期时间型数据中的组成部分：

- `year()` 取出年
- `month()` 取出月份数值
- `mday()` 取出日数值
- `yday()` 取出日期在一年中的序号，元旦为 1
- `wday()` 取出日期在一个星期内的序号，但是一个星期从星期天开始，星期天为 1，星期一为 2，星期六为 7。
- `hour()` 取出小时
- `minute()` 取出分钟
- `second()` 取出秒

比如, 2018-1-17 是星期三，则

```
month(as.POSIXct("2018-1-17 13:15:40"))
```

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

```
mday(as.POSIXct("2018-1-17 13:15:40"))
```

```
## [1] 17
```

```
wday(as.POSIXct("2018-1-17 13:15:40"))
```

```
## [1] 4
```

`lubridate` 的这些成分函数还允许被赋值，结果就修改了相应元素的值，如

```
x <- as.POSIXct("2018-1-17 13:15:40")
year(x) <- 2000
month(x) <- 1
```

```
mday(x) <- 1
x
```

```
## [1] "2000-01-01 13:15:40 CST"
```

update() 可以对一个日期或一个日期型向量统一修改其组成部分的值，如

```
x <- as.POSIXct("2018-1-17 13:15:40")
y <- update(x, year=2000)
y
```

```
## [1] "2000-01-17 13:15:40 CST"
```

update() 函数中可以用 year, month, mday, hour, minute, second 等参数修改日期的组成部分。

9.5 日期舍入计算

lubridate 包提供了 floor_date(), round_date(), ceiling_date() 等函数，对日期可以用 unit= 指定一个时间单位进行舍入。时间单位为字符串，如 seconds, 5 seconds, minutes, 2 minutes, hours, days, weeks, months, years 等。

比如，以 10 minutes 为单位，floor_date() 将时间向前归一化到 10 分钟的整数倍，ceiling_date() 将时间向后归一化到 10 分钟的整数倍，round_date() 将时间归一化到最近的 10 分钟的整数倍，时间恰好是 5 分钟倍数时按照类似四舍五入的原则向上取整。例如

```
x <- ymd_hms("2018-01-11 08:32:44")
floor_date(x, unit="10 minutes")
```

```
## [1] "2018-01-11 08:30:00 UTC"
```

```
ceiling_date(x, unit="10 minutes")
```

```
## [1] "2018-01-11 08:40:00 UTC"
```

```
round_date(x, unit="10 minutes")
```

```
## [1] "2018-01-11 08:30:00 UTC"
```

如果单位是星期，会涉及到一个星期周期的开始是星期日还是星期一的问题。用参数 week_start=7 指定开始是星期日，week_start=1 指定开始是星期一。

9.6 日期计算

在 lubridate 的支持下日期可以相减，可以进行加法、除法。lubridate 包提供了如下的三种与时间长短有关的数据类型：

- 时间长度 (duration), 按整秒计算
- 时间周期 (period), 如日、周
- 时间区间 (interval), 包括一个开始时间和一个结束时间

9.6.1 时间长度

R 的 POSIXct 日期时间之间可以相减, 如

```
d1 <- ymd_hms("2000-01-01 0:0:0")
d2 <- ymd_hms("2000-01-02 12:0:5")
di <- d2 - d1; di
```

```
## Time difference of 1.500058 days
```

结果显示与日期之间差别大小有关系, 结果是类型是 difftime。

lubridate 包提供了 duration 类型, 处理更方便:

```
as.duration(di)
```

```
## [1] "129605s (~1.5 days)"
```

lubridate 的 dseconds(), dminutes(), dhours(), ddays(), dweeks(), dyears() 函数可以直接生成时间长度类型的数据, 如

```
dhours(1)
```

```
## [1] "3600s (~1 hours)"
```

lubridate 的时间长度类型总是以秒作为单位, 可以在时间长度之间相加, 也可以对时间长度乘以无量纲数, 如

```
dhours(1) + dseconds(5)
```

```
## [1] "3605s (~1 hours)"
```

```
dhours(1)*10
```

```
## [1] "36000s (~10 hours)"
```

可以给一个日期加或者减去一个时间长度, 结果严格按推移的秒数计算, 如

```
d2 <- ymd_hms("2000-01-02 12:0:5")
d2 - dhours(5)
```

```
## [1] "2000-01-02 07:00:05 UTC"
```

```
d2 + ddays(10)
```

```
## [1] "2000-01-12 12:00:05 UTC"
```

时间的前后推移在涉及到时区和夏时制时有可能出现未预料到的情况。

9.6.2 时间周期

时间长度的固定单位是秒，但是像月、年这样的单位，因为可能有不同的天数，所以日历中的时间单位往往没有固定的时长。

lubridate 包的 `seconds()`, `minutes()`, `hours()`, `days()`, `weeks()`, `years()` 函数可以生成以日历中正常的周期为单位的时间长度，不需要与秒数相联系，可以用于时间的前后推移。这些时间周期的结果可以相加、乘以无量纲整数：

```
years(2) + 10*days(1)
```

```
## [1] "2y 0m 10d 0H 0M 0S"
```

lubridate 的月度周期因为与已有函数名冲突，所以没有提供，需要使用 `lubridate::period(num, units="month")` 的格式，其中 `num` 是几个月的数值。

为了按照日历进行日期的前后平移，而不是按照秒数进行日期的前后平移，应该使用这些时间周期。例如，因为 2016 年是闰年，按秒数给 2016-01-01 加一年，得到的并不是 2017-01-01：

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

```
## [1] "2016-12-31"
```

使用时间周期函数则得到预期结果：

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

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

9.6.3 时间区间

lubridate 提供了 `%--%` 运算符构造一个时间期间。时间区间可以求交集、并集等。

构造如：

```
d1 <- ymd_hms("2000-01-01 0:0:0")
d2 <- ymd_hms("2000-01-02 12:0:5")
din <- (d1 %--% d2); din
```

```
## [1] 2000-01-01 UTC--2000-01-02 12:00:05 UTC
```

对一个时间区间可以用除法计算其时间长度，如

```
din / ddays(1)
```

```
## [1] 1.500058
```

```
din / dseconds(1)
```

```
## [1] 129605
```

生成时间区间，也可以用 `lubridate::interval(start, end)` 函数，如

```
interval(ymd_hms("2000-01-01 0:0:0"), ymd_hms("2000-01-02 12:0:5"))
```

```
## [1] 2000-01-01 UTC--2000-01-02 12:00:05 UTC
```

可以指定时间长度和开始日期生成时间区间，如

```
d1 <- ymd("2018-01-15")
din <- as.interval(dweeks(1), start=d1); din
```

```
## [1] 2018-01-15 UTC--2018-01-22 UTC
```

注意这个时间区间表面上涉及到 8 个日期，但是实际长度还是只有 7 天，因为每一天的具体时间都是按零时计算，所以区间末尾的那一天实际不含在内。

用 `lubridate::int_start()` 和 `lubridate::int_end()` 函数访问时间区间的端点，如：

```
int_start(din)
```

```
## [1] "2018-01-15 UTC"
```

```
int_end(din)
```

```
## [1] "2018-01-22 UTC"
```

可以用 `as.duration()` 将一个时间区间转换成时间长度，用 `as.period()` 将一个时间区间转换为可变时长的时间周期个数。

用 `lubridate::int_shift()` 平移一个时间区间，如

```
din2 <- int_shift(din, by=ddays(3)); din2
```

```
## [1] 2018-01-18 UTC--2018-01-25 UTC
```

用 `lubridate::int_overlaps()` 判断两个时间区间是否有共同部分，如

```
int_overlaps(din, din2)
```

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

时间区间允许开始时间比结束时间晚，用 `lubridate::int_standardize()` 可以将时间区间标准化成开始时间小于等于结束时间。`lubridate()` 现在没有提供求交集的功能，一个自定义求交集的函数如下：

```
int_intersect <- function(int1, int2){
  n <- length(int1)
  int1 <- lubridate::int_standardize(int1)
  int2 <- lubridate::int_standardize(int2)
```

```

sele <- lubridate::int_overlaps(int1, int2)
inter <- rep(lubridate::interval(NA, NA), n)
if(any(sele)){
  inter[sele] <-
    lubridate::interval(pmax(lubridate::int_start(int1[sele]),
                             lubridate::int_start(int2[sele])),
                        pmin(lubridate::int_end(int1[sele]),
                             lubridate::int_end(int2[sele])))
}
inter
}

```

测试如:

```

d1 <- ymd(c("2018-01-15", "2018-01-18", "2018-01-25"))
d2 <- ymd(c("2018-01-21", "2018-01-23", "2018-01-30"))
din <- interval(d1, d2); din

```

```

## [1] 2018-01-15 UTC--2018-01-21 UTC 2018-01-18 UTC--2018-01-23 UTC
## [3] 2018-01-25 UTC--2018-01-30 UTC

```

```
int_intersect(rep(din[1], 2), din[2:3])
```

```
## [1] 2018-01-18 UTC--2018-01-21 UTC NA--NA
```

此自定义函数还可以进一步改成允许两个自变量长度不等的情形。

9.7 基本 R 软件的日期功能

9.7.1 生成日期和日期时间型数据

对 yyyy-mm-dd 或 yyyy/mm/dd 格式的数据, 可以直接用 `as.Date()` 转换为 Date 类型, 如:

```
x <- as.Date("1970-1-5"); x
```

```
## [1] "1970-01-05"
```

```
as.numeric(x)
```

```
## [1] 4
```

`as.Date()` 可以将多个日期字符串转换成 Date 类型, 如

```
as.Date(c("1970-1-5", "2017-9-12"))
```

```
## [1] "1970-01-05" "2017-09-12"
```

对于非标准的格式，在 `as.Date()` 中可以增加一个 `format` 选项，其中用 `%Y` 表示四位数字的年，`%m` 表示月份数字，`%d` 表示日数字。如

```
as.Date("1/5/1970", format="%m/%d/%Y")
```

```
## [1] "1970-01-05"
```

用 `as.POSIXct()` 函数把年月日格式的日期转换为 R 的标准日期，没有时间部分就认为时间在午夜。如

```
as.POSIXct(c('1998-03-16'))
```

```
## [1] "1998-03-16 CST"
```

```
as.POSIXct(c('1998/03/16'))
```

```
## [1] "1998-03-16 CST"
```

年月日中间的分隔符可以用减号也可以用正斜杠，但不能同时有减号又有斜杠。

待转换的日期时间字符串，可以是年月日之后隔一个空格以“时: 分: 秒”格式带有时间。如

```
as.POSIXct('1998-03-16 13:15:45')
```

```
## [1] "1998-03-16 13:15:45 CST"
```

用 `as.POSIXct()` 可以同时转换多项日期时间，如

```
as.POSIXct(c('1998-03-16 13:15:45', '2015-11-22 9:45:3'))
```

```
## [1] "1998-03-16 13:15:45 CST" "2015-11-22 09:45:03 CST"
```

转换后的日期变量有 `class` 属性，取值为 `POSIXct` 与 `POSIXt`，并带有一个 `tzzone`（时区）属性。

```
x <- as.POSIXct(c('1998-03-16 13:15:45', '2015-11-22 9:45:3'))
attributes(x)
```

```
## $class
## [1] "POSIXct" "POSIXt"
##
## $tzzone
## [1] ""
```

在 `as.POSIXct()` 函数中用 `format` 参数指定一个日期格式。如

```
as.POSIXct('3/13/15', format='%m/%d/%y')
```

```
## [1] "2015-03-13 CST"
```

如果日期仅有年和月，必须添加日（添加 01 为日即可）才能读入。比如用‘1991-12’表示 1991 年 12 月，则如下程序将其读入为‘1991-12-01’:


```
as.POSIXct(paste('1991-12', '-01', sep=''), format='%Y-%m-%d')
```

```
## [1] "1991-12-01 CST"
```

又如

```
old.lctime <- Sys.getlocale('LC_TIME')
```

```
Sys.setlocale('LC_TIME', 'C')
```

```
## [1] "C"
```

```
as.POSIXct(paste('01', 'DEC91', sep=''), format='%d%b%y')
```

```
## [1] "1991-12-01 CST"
```

```
Sys.setlocale('LC_TIME', old.lctime)
```

```
## [1] "Chinese (Simplified)_China.936"
```

把 'DEC91' 转换成了 '1991-12-01'。

如果明确地知道时区，在 `as.POSIXct()` 和 `as.POSIXlt()` 中可以加选项 `tz=` 字符串。选项 `tz` 的缺省值为空字符串，这一般对应于当前操作系统的默认时区。但是，有些操作系统和 R 版本不能使用默认值，这时可以为 `tz` 指定时区，比如北京时间可指定为 `tz='Etc/GMT+8'`。如

```
as.POSIXct('1949-10-01', tz='Etc/GMT+8')
```

```
## [1] "1949-10-01 -08"
```

9.7.2 取出日期时间的组成值

把一个 R 日期时间值用 `as.POSIXlt()` 转换为 `POSIXlt` 类型，就可以用列表元素方法取出其组成的年、月、日、时、分、秒等数值。如

```
x <- as.POSIXct('1998-03-16 13:15:45')
```

```
y <- as.POSIXlt(x)
```

```
cat(1900+y$year, y$mon+1, y$mday, y$hour, y$min, y$sec, '\n')
```

```
## 1998 3 16 13 15 45
```

注意 `year` 要加 1900，`mon` 要加 1。另外，列表元素 `wday` 取值 1-6 时表示星期一到星期六，取值 0 时表示星期天。

对多个日期，`as.POSIXlt()` 会把它们转换成一个列表（列表类型稍后讲述），这时可以用列表元素 `year`，`mon`，`mday` 等取出日期成分。如

```
x <- as.POSIXct(c('1998-03-16', '2015-11-22'))
```

```
as.POSIXlt(x)$year + 1900
```

```
## [1] 1998 2015
```

9.7.3 日期计算

因为 Date 类型是用数值保存的，所以可以给日期加减一个整数，如：

```
x <- as.Date("1970-1-5")
x1 <- x + 10; x1
```

```
## [1] "1970-01-15"
```

```
x2 <- x - 5; x2
```

```
## [1] "1969-12-31"
```

所有的比较运算都适用于日期类型。

可以给一个日期加减一定的秒数，如

```
as.POSIXct(c('1998-03-16 13:15:45')) - 30
```

```
## [1] "1998-03-16 13:15:15 CST"
```

```
as.POSIXct(c('1998-03-16 13:15:45')) + 10
```

```
## [1] "1998-03-16 13:15:55 CST"
```

但是两个日期不能相加。

给一个日期加减一定天数，可以通过加减秒数实现，如

```
as.POSIXct(c('1998-03-16 13:15:45')) + 3600*24*2
```

```
## [1] "1998-03-18 13:15:45 CST"
```

这个例子把日期推后了两天。

用 `difftime(time1, time2, units='days')` 计算 time1 减去 time2 的天数，如

```
x <- as.POSIXct(c('1998-03-16', '2015-11-22'))
c(difftime(x[2], x[1], units='days'))
```

```
## Time difference of 6460 days
```

函数结果用 `c()` 包裹以转换为数值，否则会带有单位。

调用 `difftime()` 时如果前两个自变量中含有时间部分，则间隔天数也会带有小数部分。如

```
x <- as.POSIXct(c('1998-03-16 13:15:45', '2015-11-22 9:45:3'))
c(difftime(x[2], x[1], units='days'))
```

```
## Time difference of 6459.854 days
```

`difftime()` 中 `units` 选项还可以取为 `'secs'`, `'mins'`, `'hours'` 等。

9.8 练习

设文件 “dates.csv” 中包含如下内容：

```
"出生日期","发病日期"
"1941/3/8","2007/1/1"
"1972/1/24","2007/1/1"
"1932/6/1","2007/1/1"
"1947/5/17","2007/1/1"
"1943/3/10","2007/1/1"
"1940/1/8","2007/1/1"
"1947/8/5","2007/1/1"
"2005/4/14","2007/1/1"
"1961/6/23","2007/1/2"
"1949/1/10","2007/1/2"
```

把这个文件读入为 R 数据框 `dates.tab`，运行如下程序定义 `date1` 和 `date2` 变量：

```
date1 <- dates.tab[, '出生日期']
date2 <- dates.tab[, '发病日期']
```

- (1) 把 `date1`、`date2` 转换为 R 的 `POSIXct` 日期型。
- (2) 求 `date1` 中的各个出生年。
- (3) 计算发病时的年龄，以周岁论（过生日才算）。
- (4) 把 `date2` 中发病年月转换为 `'monyy'` 格式，这里 `mon` 是如 `FEB` 这样英文三字母缩写，`yy` 是两数字的年份。
- (5) 对诸如 `'FEB91'`，`'OCT15'` 这样的年月数据，假设 00—20 表示 21 世纪年份，21—99 表示 20 实际年份。编写 R 函数，输入这样的字符型向量，返回相应的 `POSIXct` 格式日期，具体日期都取为相应月份的 1 号。这个习题和后两个习题可以预习函数部分来做。
- (6) 对 R 的 `POSIXct` 日期，写函数转换成 `'FEB91'`，`'OCT15'` 这样的年月表示，假设 00—20 表示 21 世纪年份，21—99 表示 20 实际年份。
- (7) 给定两个 `POSIXct` 日期向量 `birth` 和 `work`，`birth` 为生日，`work` 是入职日期，编写 R 函数，返回相应的入职周岁整数值（不到生日时周岁值要减一）。

Chapter 10

R 因子类型

10.1 因子

R 中用因子代表数据中分类变量, 如性别、省份、职业。有序因子代表有序量度, 如打分结果, 疾病严重程度等。

用 `factor()` 函数把字符型向量转换成因子, 如

```
x <- c(" 男", " 女", " 男", " 男", " 女")
sex <- factor(x)
sex
```

```
## [1] 男 女 男 男 女
## Levels: 男 女
```

```
attributes(sex)
```

```
## $levels
## [1] "男" "女"
##
## $class
## [1] "factor"
```

因子有 `class` 属性, 取值为`"factor"`, 还有一个 `levels`(水平值) 属性, 此属性可以用 `levels()` 函数访问, 如

```
levels(sex)
```

```
## [1] "男" "女"
```

因子的 `levels` 属性可以看成是一个映射, 把整数 1,2,... 映射成这些水平值, 因子在保存时会保存成整数 1,2,... 等与水平值对应的编号。这样可以节省存储空间, 在建模计算的程序中也比较有利于进行数学

运算。

事实上, `read.csv()` 函数的默认操作会把输入文件的字符型列自动转换成因子, 这对于性别、职业、地名这样的列是合适的, 但是对于姓名、日期、详细地址这样的列则不合适。所以, 在 `read.csv()` 调用中经常加选项 `stringsAsFactors=FALSE` 选项禁止这样的自动转换, 还可以用 `colClasses` 选项逐个指定每列的类型。

用 `as.numeric()` 可以把因子转换为纯粹的整数值, 如

```
as.numeric(sex)
```

```
## [1] 1 2 1 1 2
```

因为因子实际保存为整数值, 所以对因子进行一些字符型操作可能导致错误。用 `as.character()` 可以把因子转换成原来的字符型, 如

```
as.character(sex)
```

```
## [1] "男" "女" "男" "男" "女"
```

为了对因子执行字符型操作 (如取子串), 保险的做法是先用 `as.character()` 函数强制转换为字符型。

`factor()` 函数的一般形式为

```
factor(x, levels = sort(unique(x), na.last = TRUE),
      labels, exclude = NA, ordered = FALSE)
```

可以用选项 `levels` 自行指定各水平值, 不指定时由 `x` 的不同值来求得。可以用选项 `labels` 指定各水平的标签, 不指定时用各水平值的对应字符串。可以用 `exclude` 选项指定要转换为缺失值 (NA) 的元素值集合。如果指定了 `levels`, 则当自变量 `x` 的某个元素等于第 j 个水平值时输出的因子对应元素值取整数 j , 如果该元素值没有出现在 `levels` 中则输出的因子对应元素值取 NA。`ordered` 取真值时表示因子水平是有次序的 (按编码次序)。

在使用 `factor()` 函数定义因子时, 如果知道自变量元素的所有可能取值, 应尽可能使用 `levels=` 参数指定这些不同可能取值, 这样, 即使某个取值没有出现, 此变量代表的含义和频数信息也是完整的。自己指定 `levels=` 的另一好处是可以按正确的次序显示因子的分类统计值。

因为一个因子的 `levels` 属性是该因子独有的, 所以合并两个因子有可能造成错误。如

```
li1 <- factor(c('男', '女'))
li2 <- factor(c('男', '男'))
c(li1, li2)
```

```
## [1] 1 2 1 1
```

结果不再是因子。正确的做法是

```
factor(c(as.character(li1), as.character(li2)))
```

```
## [1] 男 女 男 男
```

```
## Levels: 男 女
```

即恢复成字符型后合并，然后再转换为因子。在合并两个数据框时也存在这样的问题。当然，如果在定义 li1 和 li2 时都用了 levels=c('男', '女') 选项，c(li1, li2) 也能给出正确结果。

10.2 table() 函数

用 table() 函数统计因子各水平的出现次数（称为频数或频率）。也可以对一般的向量统计每个不同元素的出现次数。如

```
table(sex)
```

```
## sex  
## 男 女  
## 3 2
```

对一个变量用 table 函数计数的结果是一个特殊的有元素名的向量，元素名是自变量的不同取值，结果的元素值是对应的频数。单个因子或单个向量的频数结果可以用向量的下标访问方法取出单个频数或若干个频数的子集。

10.3 tapply() 函数

可以按照因子分组然后每组计算另一变量的概括统计。如

```
h <- c(165, 170, 168, 172, 159)  
tapply(h, sex, mean)
```

```
##      男      女  
## 168.3333 164.5000
```

这里第一自变量 h 与第二自变量 sex 是等长的，对应元素分别为同一人的身高和性别，tapply() 函数分男女两组计算了身高平均值。

10.4 forcats 包的因子函数

```
library(forcats)
```

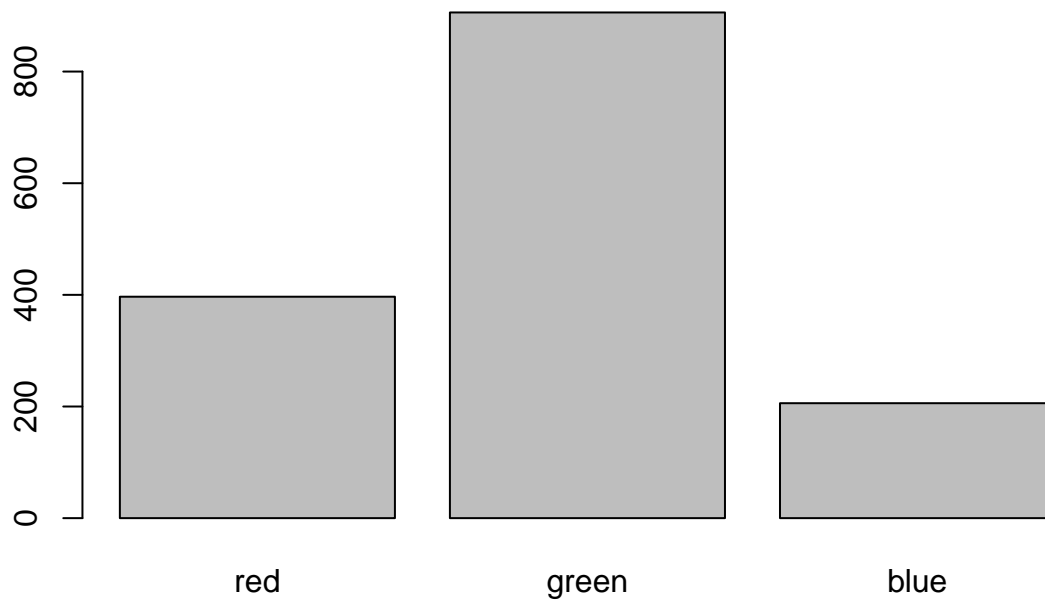
在分类变量类数较多时，往往需要对因子水平另外排序、合并等，forcats 包提供了一些针对因子的方便函数。

forcats::fac_reorder() 可以根据不同因子水平分成的组中另一数值型变量的统计量值排序。如：

```
set.seed(1)
fac <- sample(c("red", "green", "blue"), 30, replace=TRUE)
fac <- factor(fac, levels=c("red", "green", "blue"))
x <- round(100*(10+rt(30,2)))
res1 <- tapply(x, fac, sd); res1
```

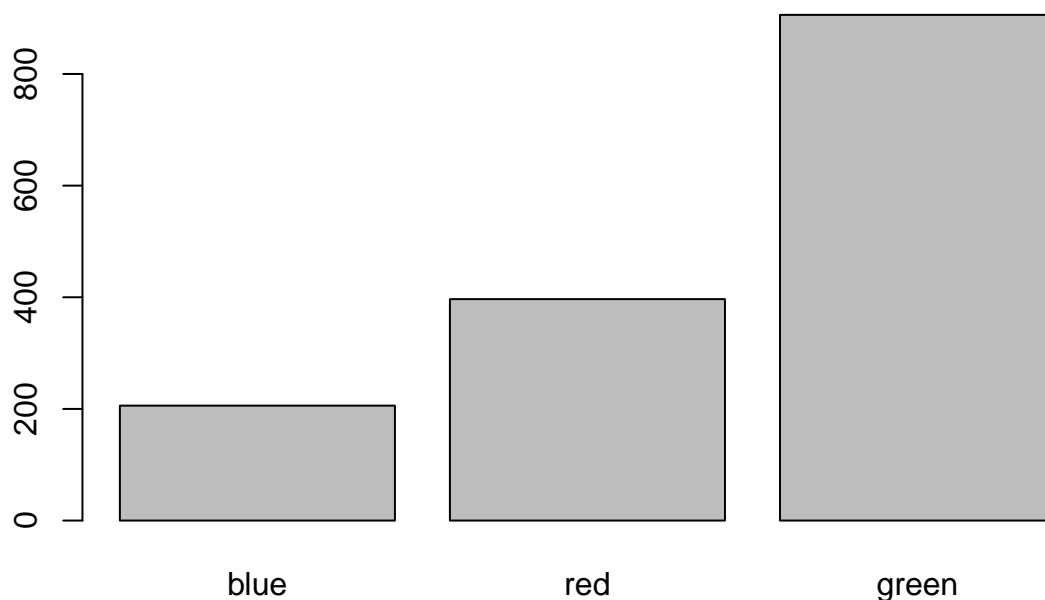
```
##      red    green    blue
## 396.6557 905.8818 205.9192
```

```
barplot(res1)
```



如果希望按照统计量次序对因子排序，可以用 `forcats::fct_reorder()` 函数，如

```
fac2 <- fct_reorder(fac, x, sd)
res2 <- tapply(x, fac2, sd)
barplot(res2)
```

新的因子 `fac2` 的因子水平次序已经按照变量 `x` 的标准差从小到大排列。

有时在因子水平数较多时仅想将特定的一个或几个水平次序放到因子水平最前面,可以用 `forcats::fct_relevel()` 函数, 如:

```
levels(fac)

## [1] "red" "green" "blue"

fac3 <- fct_relevel(fac, "blue"); levels(fac3)
```

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

`fct_relevel()` 第一个参数是要修改次序的因子, 后续可以有多个字符型参数表示要提前的水平。

`forcats::fct_reorder2(f, x, y)` 也调整因子 `f` 的水平的次序, 但是根据与每组中最大的 `x` 值相对应的 `y` 值大小调整次序, 这样在作多个因子水平对应的曲线图时可以比较容易地区分多条曲线。

`forcats::fct_recode()` 可以修改每个水平的名称, 如:

```
fac4 <- fct_recode(
  fac,
  "红"="red", "绿"="green", "蓝"="blue")
table(fac4)
```

```
## fac4
## 红 绿 蓝
## 9 11 10
```

`fct_recode()` 在修改水平名时允许多个旧水平对应到一个新水平，从而合并原来的水平。如果合并很多，可以用 `fct_collapse()` 函数，如

```
compf <- fct_collapse(
  comp,
  " 其它"=c("", " 无名", " 无应答"),
  " 联想"=c(" 联想", " 联想集团"),
  " 百度"=c(" 百度", " 百度集团")
)
```

如果某个因子频数少的水平很多,在统计时有过多水平不易展示主要的类别,可以用 `forcats::fct_lump(f)` 合并,缺省地从最少的类合并一直到“其它”类超过其它最小的类之前,可以用 `n=` 参数指定要保留多少个类。

10.5 练习

设文件 `class.csv` 中包含如下内容:

```
name,sex,age,height,weight
Alice,F,13,56.5,84
Becka,F,13,65.3,98
Gail,F,14,64.3,90
Karen,F,12,56.3,77
Kathy,F,12,59.8,84.5
Mary,F,15,66.5,112
Sandy,F,11,51.3,50.5
Sharon,F,15,62.5,112.5
Tammy,F,14,62.8,102.5
Alfred,M,14,69,112.5
Duke,M,14,63.5,102.5
Guido,M,15,67,133
James,M,12,57.3,83
Jeffrey,M,13,62.5,84
John,M,12,59,99.5
Philip,M,16,72,150
Robert,M,12,64.8,128
Thomas,M,11,57.5,85
```

William,M,15,66.5,112

用如下程序把该文件读入为 R 数据框 d.class, 其中的 sex 列已经自动转换为因子。取出其中的 sex 和 age 列到变量 sex 和 age 中

```
d.class <- read.csv('class.csv', header=TRUE)
sex <- d.class[, 'sex']
age <- d.class[, 'age']
```

- (1) 统计并显示列出 sex 的不同值频数;
- (2) 分男女两组分别求年龄最大值;
- (3) 把 sex 变量转换为一个新的因子, F 显示成 “Female”, M 显示成 “Male”。

Chapter 11

R 矩阵和数组

11.1 R 矩阵

矩阵用 `matrix` 函数定义，实际存储成一个向量，根据保存的行数和列数对应到矩阵的元素，存储次序为按列存储。定义如

```
A <- matrix(11:16, nrow=3, ncol=2); print(A)
```

```
##      [,1] [,2]
## [1,]   11   14
## [2,]   12   15
## [3,]   13   16
```

```
B <- matrix(c(1,-1, 1,1), nrow=2, ncol=2, byrow=TRUE); print(B)
```

```
##      [,1] [,2]
## [1,]    1  -1
## [2,]    1   1
```

`matrix()` 函数把矩阵元素以一个向量的形式输入，用 `nrow` 和 `ncol` 规定行数和列数，向量元素填入矩阵的缺省次序是按列填入，用 `byrow=TRUE` 选项可以转换成按行填入。

用 `nrow()` 和 `ncol()` 函数可以访问矩阵的行数和列数，如

```
nrow(A)
```

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

```
ncol(A)
```

```
## [1] 2
```

矩阵有一个 `dim` 属性，内容是两个元素的向量，两个元素分别为矩阵的行数和列数。`dim` 属性可以用 `dim()` 函数访问。如

```
attributes(A)
```

```
## $dim
## [1] 3 2
```

```
dim(A)
```

```
## [1] 3 2
```

函数 `t(A)` 返回 `A` 的转置。

11.2 矩阵子集

用 `A[1,]` 取出 `A` 的第一行，变成一个普通向量。用 `A[,1]` 取出 `A` 的第一列，变成一个普通向量。用 `A[c(1,3),1:2]` 取出指定行、列对应的子矩阵。如

```
A
```

```
##      [,1] [,2]
## [1,]   11   14
## [2,]   12   15
## [3,]   13   16
```

```
A[1,]
```

```
## [1] 11 14
```

```
A[,1]
```

```
## [1] 11 12 13
```

```
A[c(1,3), 1:2]
```

```
##      [,1] [,2]
## [1,]   11   14
## [2,]   13   16
```

用 `colnames()` 函数可以给矩阵每列命名，也可以访问矩阵列名，用 `rownames()` 函数可以给矩阵每行命名，也可以访问矩阵行名。如

```
colnames(A) <- c('X', 'Y')
```

```
rownames(A) <- c('a', 'b', 'c')
```

```
A
```

```
##      X Y
```

```
## a 11 14
## b 12 15
## c 13 16
```

矩阵可以有一个 `dimnames` 属性，此属性是两个元素的列表（列表见稍后部分的介绍），两个元素分别为矩阵的行名字符型向量与列名字符型向量。如果仅有其中之一，缺失的一个取为 `NULL`。

有了列名、行名后，矩阵下标可以用字符型向量，如

```
A[, 'Y']
```

```
## a b c
## 14 15 16
```

```
A['b',]
```

```
## X Y
## 12 15
```

```
A[c('a', 'c'), 'Y']
```

```
## a c
## 14 16
```

注意在对矩阵取子集时，如果取出的子集仅有一行或仅有一列，结果就不再是矩阵而是变成了 R 向量，R 向量既不是行向量也不是列向量。如果想避免这样的规则起作用，需要在方括号下标中加选项 `drop=FALSE`，如

```
A[, 1, drop=FALSE]
```

```
## X
## a 11
## b 12
## c 13
```

取出了 `A` 的第一列，作为列向量取出，所谓列向量实际是列数等于 1 的矩阵。如果用常量作为下标，其结果维数是确定的，不会出问题；如果用表达式作为下标，则表达式选出零个、一个、多个下标，结果维数会有不同，加 `drop=FALSE` 则是安全的做法。

矩阵也可以用逻辑下标取子集，比如

```
A
```

```
## X Y
## a 11 14
## b 12 15
## c 13 16
```

```
A[A[,1]>=2,'Y']
```

```
## a b c
## 14 15 16
```

矩阵本质上是一个向量添加了 `dim` 属性，实际保存还是保存成一个向量，其中元素的保存次序是按列填入，所以，也可以向对一个向量取子集那样，仅用一个正整数向量的矩阵取子集。如

```
A
```

```
##      X Y
## a 11 14
## b 12 15
## c 13 16
```

```
A[c(1,3,5)]
```

```
## [1] 11 13 15
```

为了挑选矩阵的任意元素组成的子集而不是子矩阵，可以用一个两列的矩阵作为下标，矩阵的每行的两个元素分别指定一个元素的行号和列号。如

```
ind <- matrix(c(1,1, 2,2, 3,2), ncol=2, byrow=TRUE)
```

```
A
```

```
##      X Y
## a 11 14
## b 12 15
## c 13 16
```

```
ind
```

```
##      [,1] [,2]
## [1,]    1    1
## [2,]    2    2
## [3,]    3    2
```

```
A[ind]
```

```
## [1] 11 15 16
```

用 `c(A)` 或 `A[]` 返回矩阵 `A` 的所有元素。如果要修改矩阵 `A` 的所有元素，可以对 `A[]` 赋值。

对矩阵 `A`，`diag(A)` 访问 `A` 的主对角线元素组成的向量。另外，若 `x` 为正整数值标量，`diag(x)` 返回 `x` 阶单位阵；若 `x` 为长度大于 1 的向量，`diag(x)` 返回以 `x` 的元素为主对角线元素的对角矩阵。

11.3 `cbind()` 和 `rbind()` 函数

若 `x` 是向量, `cbind(x)` 把 `x` 变成列向量, 即列数为 1 的矩阵, `rbind(x)` 把 `x` 变成行向量。

若 `x1, x2, x3` 是等长的向量, `cbind(x1, x2, x3)` 把它们看成列向量并在一起组成一个矩阵。`cbind()` 的自变量可以同时包含向量与矩阵, 向量的长度必须与矩阵行数相等。如

```
cbind(c(1,2), c(3,4), c(5,6))
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
cbind(A, c(1,-1,10))
```

```
##      X  Y
## a 11 14  1
## b 12 15 -1
## c 13 16 10
```

`cbind()` 的自变量中也允许有标量, 这时此标量被重复使用。如

```
cbind(1, c(1,-1,10))
```

```
##      [,1] [,2]
## [1,]    1    1
## [2,]    1   -1
## [3,]    1   10
```

`rbind()` 用法类似, 可以等长的向量看成行向量上下摞在一起, 可以是矩阵与长度等于矩阵列数的向量上下摞在一起, 向量长度为 1 也可以。

11.4 矩阵运算

11.4.1 四则运算

矩阵可以与标量作四则运算, 结果为每个元素进行相应运算, 如

```
A
```

```
##      X  Y
## a 11 14
## b 12 15
## c 13 16
```

```
C1 <- A + 2; C1
```

```
##      X  Y
## a 13 16
## b 14 17
## c 15 18
```

```
C2 <- A / 2; C2
```

```
##      X  Y
## a 5.5 7.0
## b 6.0 7.5
## c 6.5 8.0
```

当运算为矩阵乘以一个标量时，就是线性代数中的矩阵的数乘运算。

两个同形状的矩阵进行加、减运算，即对应元素相加、相减，用 $A + B$, $A - B$ 表示，如

```
C1 + C2
```

```
##      X  Y
## a 18.5 23.0
## b 20.0 24.5
## c 21.5 26.0
```

```
C1 - C2
```

```
##      X  Y
## a 7.5  9.0
## b 8.0  9.5
## c 8.5 10.0
```

这就是线性代数中矩阵的加、减运算。

对两个同形状的矩阵，用 $*$ 表示两个矩阵对应元素相乘 (注意这不是线性代数中的矩阵乘法)，用 $/$ 表示两个矩阵对应元素相除。如

```
C1 * C2
```

```
##      X  Y
## a 71.5 112.0
## b 84.0 127.5
## c 97.5 144.0
```

```
C1 / C2
```

```
##      X  Y
## a 2.363636 2.285714
```

```
## b 2.333333 2.266667
## c 2.307692 2.250000
```

11.4.2 矩阵乘法

用`%%`表示矩阵乘法而不是用`*`表示，注意矩阵乘法要求左边的矩阵的列数等于右边的矩阵的行数。如

```
A
```

```
##      X  Y
## a 11 14
## b 12 15
## c 13 16
```

```
B
```

```
##      [,1] [,2]
## [1,]    1  -1
## [2,]    1   1
```

```
C3 <- A %% B; C3
```

```
##      [,1] [,2]
## a    25   3
## b    27   3
## c    29   3
```

11.4.3 向量与矩阵相乘

矩阵与向量进行乘法运算时，向量按需要解释成列向量或行向量。当向量左乘矩阵时，看成行向量；当向量右乘矩阵时，看成立向量。如

```
B
```

```
##      [,1] [,2]
## [1,]    1  -1
## [2,]    1   1
```

```
c(1,1) %% B
```

```
##      [,1] [,2]
## [1,]    2   0
```

```
B %% c(1,1)
```

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

```
## [1,]    0
## [2,]    2

c(1,1) %*% B %*% c(1,1)
```

```
##      [,1]
## [1,]    2
```

注意矩阵乘法总是给出矩阵结果，即使此矩阵已经退化为行向量、列向量甚至于退化为标量也是一样。如果需要，可以用 `c()` 函数把一个矩阵转换成按列拉直的向量。

11.4.4 内积

设 x, y 是两个向量，计算向量内积，可以用 `sum(x*y)` 表示。

设 A, B 是两个矩阵， $A^T B$ 是广义的内积，也称为叉积 (crossprod)，结果是一个矩阵，元素为 A 的每列与 B 的每列计算内积的结果。 $A^T B$ 在 R 中可以表示为 `crossprod(A, B)`， $A^T A$ 可以表示为 `crossprod(A)`。要注意的是，`crossprod()` 的结果总是矩阵，所以计算两个向量的内积用 `sum(x,y)` 而不用 `crossprod(x,y)`。

11.4.5 外积

R 向量支持外积运算，记为 `%o%`，结果为矩阵。 $x \%o\% y$ 的第 i 行第 j 列元素等于 $x[i]$ 乘以 $y[j]$ 。如

```
c(1,2,3) %o% c(1, -1)
```

```
##      [,1] [,2]
## [1,]    1  -1
## [2,]    2  -2
## [3,]    3  -3
```

这种运算还可以推广到 x 的每一元素与 y 的每一元素进行其它的某种运算，而不限于乘积运算，可以用 `outer(x,y,f)` 完成，其中 f 是某种运算，或者接受两个自变量的函数。

11.5 逆矩阵与线性方程组求解

用 `solve(A)` 求 A 的逆矩阵，如

```
solve(B)
```

```
##      [,1] [,2]
## [1,]  0.5  0.5
## [2,] -0.5  0.5
```

用 `solve(A,b)` 求解线性方程组 $Ax = b$ 中的 x ，如

```
solve(B, c(1,2))
```

```
## [1] 1.5 0.5
```

求解了线性方程组

$$\begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix} x = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

11.6 apply() 函数

`apply(A, 2, FUN)` 把矩阵 `A` 的每一列分别输入到函数 `FUN` 中，得到对应于每一列的结果，如

```
D <- matrix(c(6,2,3,5,4,1), nrow=3, ncol=2); D
```

```
##      [,1] [,2]
## [1,]    6    5
## [2,]    2    4
## [3,]    3    1
```

```
apply(D, 2, sum)
```

```
## [1] 11 10
```

`apply(A, 1, FUN)` 把矩阵 `A` 的每一行分别输入到函数 `FUN` 中，得到与每一行对应的结果，如

```
apply(D, 1, mean)
```

```
## [1] 5.5 3.0 2.0
```

如果函数 `FUN` 返回多个结果，则 `apply(A, 2, FUN)` 结果为矩阵，矩阵的每一列是输入矩阵相应列输入到 `FUN` 的结果，结果列数等于 `A` 的列数。如

```
apply(D, 2, range)
```

```
##      [,1] [,2]
## [1,]    2    1
## [2,]    6    5
```

如果函数 `FUN` 返回多个结果，为了对每行计算 `FUN` 的结果，结果存入一个与输入的矩阵行数相同的矩阵，应该用 `t(apply(A, 1, FUN))` 的形式，如

```
t(apply(D, 1, range))
```

```
##      [,1] [,2]
## [1,]    5    6
## [2,]    2    4
## [3,]    1    3
```

11.7 多维数组

矩阵是多维数组 (array) 的特例。矩阵是 $x_{ij}, i = 1, 2, \dots, n, j = 1, 2, \dots, m$ 这样的两下标数据的存储格式，三维数组是 $x_{ijk}, i = 1, 2, \dots, n, j = 1, 2, \dots, m, k = 1, 2, \dots, p$ 这样的三下标数据的存储格式， s 维数组则是有 s 个下标的数据的存储格式。实际上，给一个向量添加一个 `dim` 属性就可以把它变成多维数组。

多维数组的一般定义语法为

```
数组名 <- array(数组元素,
  dim=c(第一下标个数, 第二下标个数, ..., 第s下标个数))
```

其中数组元素的填入次序是第一下标变化最快，第二下标次之，最后一个下标是变化最慢的。这种次序称为 FORTRAN 次序。

下面是一个三维数组定义例子。

```
ara <- array(1:24, dim=c(2,3,4)); ara
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]   13   15   17
## [2,]   14   16   18
##
## , , 4
##
##      [,1] [,2] [,3]
## [1,]   19   21   23
## [2,]   20   22   24
```

这样的数组保存了 $x_{ijk}, i = 1, 2, j = 1, 2, 3, k = 1, 2, 3, 4$ 。三维数组 `ara` 可以看成是 4 个 2×3 矩阵。取出其中一个如 `ara[, , 2]` (取出第二个矩阵)

```
ara[:,2]
```

```
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
```

多维数组可以利用下标进行一般的子集操作，比如 `ara[2, 2:3]` 是 $x_{ijk}, i = 1, 2, j = 2, k = 2, 3$ 的值，结果是一个 2×2 矩阵：

```
ara[2,2:3]
```

```
##      [,1] [,2]
## [1,]    9   15
## [2,]   10   16
```

多维数组在取子集时如果某一维下标是标量，则结果维数会减少，可以在方括号内用 `drop=FALSE` 选项避免这样的规则发生作用。

类似于矩阵，多维数组可以用一个矩阵作为下标，如果是三维数组，矩阵就需要有 3 列，四维数组需要用 4 列矩阵。下标矩阵的每行对应于一个数组元素。

Chapter 12

数据框

12.1 数据框

统计分析中最常见的原始数据形式是类似于数据库表或 Excel 数据表的形式。这样形式的数据在 R 中叫做数据框 (data.frame)。数据框类似于一个矩阵，有 n 行、 p 列，但各列允许有不同类型：数值型向量、因子、字符型向量、日期时间向量。同一列的数据类型相同。在 R 中数据框是一个特殊的列表，其每个列表元素都是一个长度相同的向量。事实上，数据框还允许一个元素是一个矩阵，但这样会使得某些读入数据框的函数发生错误。

函数 `data.frame()` 可以生成数据框，如

```
d <- data.frame(
  name=c(" 李明", " 张聪", " 王建"),
  age=c(30, 35, 28),
  height=c(180, 162, 175),
  stringsAsFactors=FALSE)
print(d)
```

```
##   name age height
## 1 李明  30   180
## 2 张聪  35   162
## 3 王建  28   175
```

`data.frame()` 函数会将字符型列转换成因子，加选项 `stringsAsFactors=FALSE` 可以避免这样的转换。

数据框每列叫做一个变量，每列都有名字，称为列名或变量名，可以用 `names()` 函数和 `colnames()` 函数访问。如

```
names(d)

## [1] "name"  "age"   "height"
```

```
colnames(d)
```

```
## [1] "name" "age" "height"
```

给 `names(d)` 或 `colnames(d)` 赋值可以修改列名。

用 `as.data.frame(x)` 可以把 `x` 转换成数据框。如果 `x` 是一个向量，转换结果是以 `x` 为唯一一列的数据框。如果 `x` 是一个列表并且列表元素都是长度相同的向量，转换结果中每个列表变成数据框的一列。如果 `x` 是一个矩阵，转换结果把矩阵的每列变成数据框的一列。

数据框是一个随着 R 语言前身 S 语言继承下来的概念，现在已经有一些不足之处，`tibble` 包提供了 `tibble` 类，这是数据框的一个改进版本。

12.2 数据框内容访问

数据框可以用矩阵格式访问，如

```
d[2,3]
```

```
## [1] 162
```

访问单个元素；

```
d[[2]]
```

```
## [1] 30 35 28
```

访问第二列，结果为向量；

```
d[,2]
```

```
## [1] 30 35 28
```

也访问第二列，但是这种作法与 `tibble` 不兼容，所以应避免使用。

按列名访问列可用如

```
d[["age"]]
```

```
## [1] 30 35 28
```

```
d[, "age"]
```

```
## [1] 30 35 28
```

```
d$age
```

```
## [1] 30 35 28
```

其中第二种做法与 `tibble` 不兼容，应避免使用。

因为数据框的一行不一定是相同数据类型，所以数据框的一行作为子集，结果还是数据框，而不是向量。如

```
x <- d[2,]; x
```

```
##   name age height
## 2 张聪  35   162
```

```
is.data.frame(x)
```

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

可以同时取行子集和列子集，如

```
d[1:2, 'age']
```

```
## [1] 30 35
```

```
d[1:2, c('age', 'height')]
```

```
##   age height
## 1  30    180
## 2  35    162
```

```
d[d[, 'age'] >= 30,]
```

```
##   name age height
## 1 李明  30    180
## 2 张聪  35    162
```

与矩阵类似地是，用如 `d[, 'age']`, `d[, 2]` 这样的方法取出的数据框的单个列是向量而不再是数据框。但是，如果取出两列或者两列以上，结果则是数据框。如果取列子集时不能预先知道取出的列个数，则子集结果有可能是向量也有可能是数据框，容易造成后续程序错误。对一般的数据框，可以在取子集的方括号内加上 `drop=FALSE` 选项，确保取列子集的结果总是数据框。数据框的改进类型 `tibble` 在取出列子集时保持为 `tibble` 格式。

对数据框变量名按照字符串与集合进行操作可以实现复杂的列子集筛选，参见小标题分汇总的例子。

数据框每一行可以有行名，这在原始的 S 语言和传统的 R 语言中是重要的技术，但是在改进类型 `tibble` 中则取消了行名，需要用列名实现功能一般改用 `left_join()` 函数实现。

比如，每一行定义行名为身份证号，则可以唯一识别各行。下面的例子以姓名作为行名：

```
rownames(d) <- d$name
d$name <- NULL
d
```

```
##   age height
## 李明  30    180
## 张聪  35    162
```

```
## 王建 28 175
```

用数据框的行名可以建立一个值到多个值的对应表。比如，有如下的数据框：

```
dm <- data.frame(
  ' 年级' = 1:6,
  ' 出游' = c(0, 2, 2, 2, 2, 1),
  ' 疫苗' = c(T, F, F, F, T, F)
)
```

其中“出游”是每个年级安排的出游次数，“疫苗”是该年级有全体无计划免疫注射。把年级变成行名，可以建立年级到出游次数与疫苗注射的对应表：

```
rownames(dm) <- dm[[' 年级']]
dm[[' 年级']] <- NULL
```

这样，假设某个社区的小学中抽取的 4 个班的年级为 `c(2,1,1,3)`，其对应的出游和疫苗注射信息可查询如下：

```
x <- c(2,1,1,3)
dm[as.character(x),]
```

```
##      出游  疫苗
## 2      2 FALSE
## 1      0  TRUE
## 1.1    0  TRUE
## 3      2 FALSE
```

结果中包含了不必要也不太合适的行名，可以去掉，以上程序改成：

```
x <- c(2,1,1,3)
xx <- dm[as.character(x),]
rownames(xx) <- NULL
xx
```

```
##      出游  疫苗
## 1      2 FALSE
## 2      0  TRUE
## 3      0  TRUE
## 4      2 FALSE
```

如果要从多个值建立映射，比如，从省名与县名映射到经度、纬度，可以预先用 `paste()` 函数把省名与县名合并在一起，中间以适当字符（如“-”）分隔，以这样的合并字符串为行名。

对于代替数据框的 `tibble` 类型，如果要想实现行名的功能，可以将行名作为单独的一列，然后用 `dplyr` 包的 `inner_join()`、`left_join()`、`full_join()` 等函数横向合并数据集。

12.3 数据框与矩阵的区别

数据框不能作为矩阵参加矩阵运算。需要时，可以用 `as.matrix()` 函数转换数据框或数据框的子集为矩阵。
如

```
d2 <- as.matrix(d[,c("age", "height")])
d3 <- crossprod(d2); d3
```

```
##           age height
## age      2909  15970
## height  15970  89269
```

这里 `crossprod(A)` 表示 $A^T A$ 。

12.4 gl() 函数

可以用数据框保存试验结果，对有多个因素的试验，往往需要生成多个因素完全搭配并重复的表格。函数 `gl()` 可以生成这样的重复模式。比如，下面的例子：

```
d <- data.frame(
  group=gl(3, 10, length=30),
  subgroup=gl(5,2,length=30),
  obs=gl(2,1,length=30))
print(d)
```

```
##    group subgroup obs
## 1      1         1   1
## 2      1         1   2
## 3      1         2   1
## 4      1         2   2
## 5      1         3   1
## 6      1         3   2
## 7      1         4   1
## 8      1         4   2
## 9      1         5   1
## 10     1         5   2
## 11     2         1   1
## 12     2         1   2
## 13     2         2   1
## 14     2         2   2
## 15     2         3   1
```

```
## 16      2      3      2
## 17      2      4      1
## 18      2      4      2
## 19      2      5      1
## 20      2      5      2
## 21      3      1      1
## 22      3      1      2
## 23      3      2      1
## 24      3      2      2
## 25      3      3      1
## 26      3      3      2
## 27      3      4      1
## 28      3      4      2
## 29      3      5      1
## 30      3      5      2
```

结果的数据框 `d` 有三个变量: `group` 是大组, 共分 3 个大组, 每组 10 个观测; `subgroup` 是子组, 在每个大组内分为 5 个子组, 每个子组 2 个观测。共有 $3 \times 5 \times 2 = 30$ 个观测 (行)。

`gl()` 第一个参数是因子水平个数, 第二个参数是同一因子水平连续重复次数, 第三个参数是总共需要的元素个数, 所有水平都出现后则重复整个模式直到长度满足要求。

12.5 tibble 类型

tibble 类型是一种改进的数据框。readr 包的 `read_csv()` 函数是 `read.csv()` 函数的一个改进版本, 它将 CSV 文件读入为 tibble 类型, 如如:

```
library(tibble)
library(readr)
t.class <- read_csv("class.csv")
```

```
## Parsed with column specification:
```

```
## cols(
##   name = col_character(),
##   sex = col_character(),
##   age = col_integer(),
##   height = col_double(),
##   weight = col_double()
## )
```

```
t.class
```

```
## # A tibble: 19 x 5
##   name    sex    age height weight
##   <chr>  <chr> <int>  <dbl>  <dbl>
## 1 Alice  F      13    56.5   84.0
## 2 Becka  F      13    65.3   98.0
## 3 Gail   F      14    64.3   90.0
## 4 Karen  F      12    56.3   77.0
## 5 Kathy  F      12    59.8   84.5
## 6 Mary   F      15    66.5  112
## 7 Sandy  F      11    51.3   50.5
## 8 Sharon F      15    62.5  112
## 9 Tammy  F      14    62.8  102
## 10 Alfred M      14    69.0  112
## 11 Duke   M      14    63.5  102
## 12 Guido  M      15    67.0  133
## 13 James  M      12    57.3   83.0
## 14 Jeffrey M      13    62.5   84.0
## 15 John   M      12    59.0   99.5
## 16 Philip M      16    72.0  150
## 17 Robert M      12    64.8  128
## 18 Thomas M      11    57.5   85.0
## 19 William M      15    66.5  112
```

tibble 类型的类属依次为 tbl_df, tbl, data.frame:

```
class(t.class)
```

```
## [1] "tbl_df"      "tbl"        "data.frame"
```

用 `as_tibble()` 可以将一个数据框转换为 tibble, dplyr 包提供了 `filter()`、`select()`、`arrange()`、`mutate()` 等函数用来对 tibble 选取行子集、列子集, 排序、修改或定义新变量, 等等。

可以用 `tibble()` 函数生成小的 tibble, 如

```
t.bp <- tibble(
  `序号` = c(1, 5, 6, 9, 10, 15),
  `收缩压` = c(145, 110, "未测", 150, "拒绝", 115))
t.bp

## # A tibble: 6 x 2
##   序号 收缩压
##   <dbl> <chr>
```

```
## 1  1.00 145
## 2  5.00 110
## 3  6.00 未测
## 4  9.00 150
## 5 10.0  拒绝
## 6 15.0  115
```

用 `tribble` 可以按类似于 CSV 格式输入一个 tibble, 如

```
t.bp <- tribble(
  ~`序号`, ~`收缩压`,
  1, 145,
  5, 110,
  6, " 未测",
  9, 150,
  10, " 拒绝",
  15, 115
)
t.bp
```

```
## # A tibble: 6 x 2
##   序号 收缩压
##   <dbl> <chr>
## 1  1.00 145
## 2  5.00 110
## 3  6.00 未测
## 4  9.00 150
## 5 10.0  拒绝
## 6 15.0  115
```

注意 `tribble()` 中数据每行末尾也需要有逗号, 最后一行末尾没有逗号。这比较适用于在程序中输入小的数据集。

tibble 与数据框的一大区别是在显示时不自动显示所有内容, 这样可以避免显示很大的数据框将命令行的所有显示都充满。可以在 `print()` 用 `n=` 和 `width=` 选项指定要显示的行数和列数。

另外, 用单重的方括号取列子集时, 即使仅取一列, 从 tibble 取出的一列结果仍是 tibble 而不是向量, 这时应使用双方括号格式或 `$` 格式。因为这个原因有些原来的程序输入 tibble 会出错, 这时可以用 `as.data.frame()` 转换成数据框。如:

```
t.bp[, " 收缩压"]
```

```
## # A tibble: 6 x 1
##   收缩压
```



```
##    <chr>
## 1 145
## 2 110
## 3 未测
## 4 150
## 5 拒绝
## 6 115
```

```
t.bp[["收缩压"]]
```

```
## [1] "145" "110" "未测" "150" "拒绝" "115"
```

tibble 在定义时不需要列名为合法变量名，但是作为变量名使用时需要用反单撇号包裹。tibble 不使用行名，需要行名时，将其保存为 tibble 的一列。原来用行名完成的功能，可以改用 dplyr 包的 `left_join()` 等函数，这些函数进行数据框的横向连接。

12.6 练习

假设 “class.csv” 已经读入为 R 数据框 d.class, 其中的 sex 列已经自动转换为因子。

- (1) 显示 d.class 中年龄至少为 15 的行子集；
- (2) 显示女生且年龄至少为 15 的学生姓名和年龄；
- (3) 取出数据框中的 age 变量赋给变量 x。

Chapter 13

列表类型

13.1 列表

R 中列表 (list) 类型来保存不同类型的数据。一个主要目的是提供 R 分析结果输出包装：输出一个变量，这个变量包括回归系数、预测值、残差、检验结果等等一系列不能放到规则形状数据结构中的内容。实际上，数据框也是列表的一种，但是数据框要求各列等长，而列表不要求。

列表可以有多个元素，但是与向量不同的是，列表的不同元素的类型可以不同，比如，一个元素是数值型向量，一个元素是字符串，一个元素是标量，一个元素是另一个列表。

定义列表用函数 `list()`，如

```
rec <- list(name=" 李明", age=30,  
            scores=c(85, 76, 90))  
rec
```

```
## $name  
## [1] "李明"  
##  
## $age  
## [1] 30  
##  
## $scores  
## [1] 85 76 90
```

用 `typeof()` 函数判断一个列表，返回结果为 `list`。可以用 `is.list()` 函数判断某个对象是否是列表类型。

13.2 列表元素访问

列表的一个元素也可以称为列表的一个“变量”，单个列表元素必须用两重方括号格式访问，如

```
rec[[3]]

## [1] 85 76 90

rec[[3]][2]

## [1] 76

rec[["age"]]
```

```
## [1] 30
```

如果使用单重方括号对列表取子集，结果还是列表而不是列表元素，如

```
rec[3]

## $scores
## [1] 85 76 90
```

列表的单个元素也可以用 `$` 格式访问，如

```
rec$age

## [1] 30
```

列表一般都应该有元素名，元素名可以看成是变量名，列表中的每个元素看成一个变量。用 `names()` 函数查看和修改元素名。如

```
names(rec)

## [1] "name" "age" "scores"

names(rec)[names(rec)=='scores'] <- '三科分数'
names(rec)

## [1] "name" "age" "三科分数"

rec[["三科分数"]]
```

```
## [1] 85 76 90
```

可以修改列表元素内容。如

```
rec[["三科分数"]][2] <- 0
print(rec)

## $name
## [1] "李明"
```

```
##
## $age
## [1] 30
##
## $三科分数
## [1] 85 0 90
```

直接给列表不存在的元素名定义元素值就添加了新元素，而且不同于使用向量，对于列表而言这是很正常的做法，比如

```
rec[['身高']] <- 178
print(rec)
```

```
## $name
## [1] "李明"
##
## $age
## [1] 30
##
## $三科分数
## [1] 85 0 90
##
## $身高
## [1] 178
```

把某个列表元素赋值为 `NULL` 就删掉这个元素。如

```
rec[['age']] <- NULL
print(rec)
```

```
## $name
## [1] "李明"
##
## $三科分数
## [1] 85 0 90
##
## $身高
## [1] 178
```

在 `list()` 函数中允许定义元素为 `NULL`，这样的元素是存在的，如：

```
li <- list(a=120, b='F', c=NULL); li
```

```
## $a
```

```
## [1] 120
##
## $b
## [1] "F"
##
## $c
## NULL
```

但是，要把已经存在的元素修改为 NULL 值而不是删除此元素，或者给列表增加一个取值为 NULL 的元素，这时需要用单重的方括号取子集，这样的子集会保持其列表类型，给这样的子列表赋值为 `list(NULL)`，如：

```
li['b'] <- list(NULL)
li['d'] <- list(NULL)
li
```

```
## $a
## [1] 120
##
## $b
## NULL
##
## $c
## NULL
##
## $d
## NULL
```

13.3 列表类型转换

用 `as.list()` 把一个其它类型的对象转换成列表；用 `unlist()` 函数把列表转换成基本向量。如

```
li1 <- as.list(1:3)
li1
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
```

```
li2 <- list(x=1, y=c(2,3))
unlist(li2)
```

```
## x y1 y2
## 1 2 3
```

13.4 返回列表的函数示例—strsplit()

`strsplit()` 输入一个字符型向量并指定一个分隔符，返回一个项数与字符型向量元素个数相同的列表，列表每项对应于字符型向量中一个元素的拆分结果。如

```
x <- c('10, 8, 7', '5, 2, 2', '3, 7, 8', '8, 8, 9')
res <- strsplit(x, ','); res
```

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

为了把拆分结果进一步转换成一个数值型矩阵，可以使用 `sapply()` 函数如下：

```
t(sapply(res, as.numeric))
```

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

`sapply()` 函数是 `apply` 类函数之一，稍后再详细进行讲解。

Chapter 14

工作空间

R 把在命令行定义的变量都保存到工作空间中，在退出 R 时可以选择是否保存工作空间。这也是 R 与其他如 C、Java 这样的语言的区别之一。

用 `ls()` 命令可以查看工作空间中的内容。

随着多次在命令行使用 R，工作空间的变量越来越多，使得重名的可能性越来越大，而且工作空间中变量太多也让我们不容易查看其内容。在命令行定义的变量称为“全局变量”，在编程实际中，全局变量是需要慎用的。

可以用 `rm()` 函数删除工作空间中的变量，格式如

```
rm(d, h, name, rec, sex, x)
```

要避免工作空间杂乱，最好的办法还是所有的运算都写到自定义函数中。自定义函数中定义的变量都是临时的，不会保存到工作空间中。这样，仅需要时才把变量值在命令行定义，这样的变量一般是读入的数据或自定义的函数（自定义函数也保存在工作空间中）。

可以定义如下的 `sandbox()` 函数：

```
sandbox <- function(){  
  cat(' 沙盘：接连的空行回车可以退出。\\n')  
  browser()  
}
```

运行 `sandbox()` 函数，将出现如下的 `browser` 命令行：

沙盘：接连的空行回车可以退出。

Called from: sandbox()

Browser[1]>

提示符变成了“Browser[n]”，其中 `n` 代表层序号。在这样的 `browser` 命令行中随意定义变量，定义的变量不会保存到工作空间中。用“Q”命令可以退出这个沙盘环境，接连回车也可以退出。

Part III

R 编程

Chapter 15

R 输入输出

15.1 输入输出的简单方法

15.1.1 简单的输出

用 `print()` 函数显示某个变量或表达式的值，如

```
x <- 1.234
print(x)
```

```
## [1] 1.234
```

```
y <- c(1,3,5)
print(y[2:3])
```

```
## [1] 3 5
```

在命令行使用 R 时，直接以变量名或表达式作为命令可以起到用 `print()` 函数显示的相同效果。

用 `cat()` 函数把字符串、变量、表达式连接起来显示，其中变量和表达式的类型一般是标量或向量，不能是矩阵、列表等复杂数据。如

```
cat("x =", x, "\n")
```

```
## x = 1.234
```

```
cat("y =", y, "\n")
```

```
## y = 1 3 5
```

注意 `cat()` 显示中需要换行需要在自变量中包含字符串 `"\n"`，即换行符。

`cat()` 默认显示在命令行窗口，为了写入指定文件中，在 `cat()` 调用中用 `file=` 选项，这时如果已有文件会把原有内容覆盖，为了在已有文件时不覆盖原有内容而是在末尾添加，在 `cat()` 中使用 `append=TRUE`

选项。如:

```
cat("=== 结果文件 ===\n", file="res.txt")
cat("x =", x, "\n", file="res.txt", append=TRUE)
```

函数 `sink()` 可以用来把命令行窗口显示的运行结果转向保存到指定的文本文件中, 如果希望保存到文件的同时也在命令行窗口显示, 使用 `split=TRUE` 选项。如

```
sink("allres.txt", split=TRUE)
```

为了取消这样的输出文件记录, 使用不带自变量的 `sink()` 调用, 如

```
sink()
```

在 R 命令行环境中定义的变量、函数会保存在工作空间中, 并在退出 R 会话时可以保存到硬盘文件中。用 `save()` 命令要求把指定的若干个变量 (直接用名字, 不需要表示成字符串) 保存到用 `file=` 指定的文件中, 随后可以用 `load()` 命令恢复到工作空间中。这样保存的 R 特殊格式的文件是通用的, 不依赖于硬件和操作系统, 但字符串内容可能有编码问题。如

```
save(x, y, file="x-y.RData")
load("x-y.RData")
```

对于一个数据框, 可以用 `write.csv()` 或 `readr::write_csv()` 将其保存为逗号分隔的文本文件, 这样的文件可以很容易地被其它软件识别访问, 如 Microsoft Excel 软件可以很容易地把这样的文件读成电子表格。用如

```
da <- tibble('name'=c(' 李明', ' 刘颖', ' 张浩'),
             'age'=c(15, 17, 16))
write_csv(da, path="mydata.csv")
```

结果生成的 `mydata.csv` 文件内容如下:

```
name,age
李明,15
刘颖,17
张浩,16
```

15.1.2 简单的输入

用 `scan()` 函数可以输入文本文件中的数值向量, 文件名用 `file=` 选项给出。文件中数值之间以空格分开。如

```
cat(1:12, "\n", file="d:/work/x.txt")
x <- scan("d:/work/x.txt")
```

程序中用全路径给出了输入文件位置, 注意路径中用了正斜杠/作为分隔符, 如果在 MS Windows 环境下

使用\作为分隔符，在 R 的字符串常量中\必须写成\\。

如果 `scan()` 中忽略输入文件参数，此函数将从命令行读入数据。可以在一行用空格分开多个数值，可以用多行输入直到空行结束输入。

这样的方法也可以用来读入矩阵。设文件 `mat.txt` 包含如下矩阵内容：

```
3  4  2
5 12 10
7  8  6
1  9 11
```

可以先把文件内容读入到一个 R 向量中，再利用 `matrix()` 函数转换成矩阵，注意要使用 `byrow=TRUE` 选项，而且只要指定 `nrow` 选项，可以忽略 `ncol` 选项。如

```
M <- matrix(scan('mat.txt', quiet=TRUE), nrow=4, byrow=TRUE)
M
```

`scan()` 中的 `quite=TRUE` 选项使得读入时不自动显示读入的数值项数。

上面读入数值矩阵的方法在数据量较大的情形也可以使用，与之不同的是，`read.table()` 或 `readr::read_table()` 函数也可以读入这样的数据，但是会保存成数据框而不是矩阵，而且 `read.table()` 函数在读入大规模的矩阵时效率很低。

15.2 读取 CSV 文件

对于保存在文本文件中的电子表格数据，R 可以用 `read.csv()`, `read.table()`, `read.delim()`, `read.fwf()` 等函数读入，但是建议在 `readr` 包的支持下用 `read_csv()`, `read_table2()`, `read_delim()`, `read_fwf()` 等函数读入，这些将读入的数据框保存为 `tibble` 类型，`tibble` 是数据框的一个变种，改善了数据框的一些不适当的设计。`readr` 的读入速度比基本 R 软件的 `read.csv()` 等函数的速度快得多，速度可以相差 10 倍，也不自动将字符型列转换成因子，不自动修改变量名为合法变量名，不设置行名。

对于中小规模的数据，CSV 格式作为文件交换格式比较合适，兼容性强，各种数据管理软件与统计软件都可以很容易地读入和生成这样格式的文件，但是特别大型的数据读入效率很低。

CSV 格式的文件用逗号分隔开同一行的数据项，一般第一行是各列的列名（变量名）。对于数值型数据，只要表示成数值常量形式即可。对于字符型数据，可以用双撇号包围起来，也可以不用撇号包围。但是，如果数据项本身包含逗号，就需要用双撇号包围。例如，下面是一个名为 `testcsv.csv` 的文件内容，其中演示了内容中有逗号、有双撇号的情况。

```
id,words
1,"PhD"
2,Master's degree
3,"Bond,James"
4,"A ""special"" gift"
```

为读入上面的内容，只要用如下程序:

```
d <- read_csv("testcsv.csv")
```

读入的数据框显示如下:

```
# A tibble: 4 × 2
      id      words
  <int>    <chr>
1     1      PhD
2     2 Master's degree
3     3  Bond,James
4     4 A "special" gift
```

read_csv() 还可以从字符串读入一个数据框，如

```
d.small <- read_csv("name,x,y
John, 33, 95
Kim, 21, 64
Sandy, 49, 100
")
d.small
```

```
## # A tibble: 3 × 3
##   name      x      y
##   <chr> <int> <int>
## 1 John     33     95
## 2 Kim      21     64
## 3 Sandy    49    100
```

read_csv() 的 skip= 选项跳过开头的若干行。当数据不包含列名时，只要指定 col_names=FALSE，变量将自动命名为 X1, X2, ..., 也可以用 col_names= 指定各列的名字，如

```
d.small <- read_csv("John, 33, 95
Kim, 21, 64
Sandy, 49, 100
", col_names=c("name", "x", "y") )
d.small
```

```
## # A tibble: 3 × 3
##   name      x      y
##   <chr> <int> <int>
## 1 John     33     95
## 2 Kim      21     64
## 3 Sandy    49    100
```


`read_csv()` 将空缺的值读入为缺失值，将“NA”也读入为缺失值。可以用 `na=` 选项改变这样的设置。也可以将带有缺失值的列先按字符型原样读入，然后再进行转换。

CSV 文件是文本文件，是有编码问题的，尤其是中文内容的文件。`readr` 包的默认编码是 UTF-8 编码。例如，文件 `bp.csv` 以 GBK 编码（有时称为 GB18030 编码，这是中文 Windows 所用的中文编码）保存了如下内容：

序号,收缩压

1,145

5,110

6, 未测

9,150

10, 拒绝

15,115

如果直接用 `read_csv()`：

```
d <- read_csv("bp.csv")
```

可能在读入时出错，或者访问时出错。为了读入用 GBK 编码的中文 CSV 文件，需要利用 `locale` 参数和 `locale()` 函数：

```
d <- read_csv("bp.csv", locale=locale(encoding="GBK"))
```

```
## Parsed with column specification:
```

```
## cols(
```

```
##   序号 = col_integer(),
```

```
##   收缩压 = col_character()
```

```
## )
```

```
d
```

```
## # A tibble: 6 x 2
```

```
##   序号 收缩压
```

```
##   <int> <chr>
```

```
## 1     1 145
```

```
## 2     5 110
```

```
## 3     6 未测
```

```
## 4     9 150
```

```
## 5    10 拒绝
```

```
## 6    15 115
```

对每列的类型，`readr` 用前 1000 行猜测合理的类型，并在读取后显示猜测的每列类型。

但是有可能类型改变发生在 1000 行之后。`col_types` 选项可以指定每一列的类型，如"`col_double()`", "`col_integer()`", "`col_character()`", "`col_factor()`", "`col_date()`", "`col_datetime()`" 等。

`cols()` 函数可以用来规定各列类型，并且有一个 `.default` 参数指定缺省类型。对因子，需要在 `col_factor()` 中用 `levels=` 指定因子水平。

可以复制 `readr` 猜测的类型作为 `col_types` 的输入，这样当数据变化时不会因为偶尔猜测错误而使得程序出错。如

```
d <- read_csv("bp.csv", locale=locale(encoding="GBK"),
              col_types=cols(
                `序号` = col_integer(),
                `收缩压` = col_character()
              ))
d
```

```
## # A tibble: 6 x 2
##   序号 收缩压
##   <int> <chr>
## 1     1  145
## 2     5  110
## 3     6 未测
## 4     9  150
## 5    10 拒绝
## 6    15  115
```

当猜测的文件类型有问题的时候，可以先将所有列都读成字符型，然后用 `type_convert()` 函数转换，如：

```
d <- read_csv("filename.csv",
              col_types=cols(.default = col_character()))
d <- type_convert(d)
```

读入有错时，对特大文件可以先少读入一些行，用 `nmax=` 可以指定最多读入多少行。调试成功后再读入整个文件。

设文件 `class.csv` 内容如下：

```
name,sex,age,height,weight
Alice,F,13,56.5,84
Becka,F,13,65.3,98
Gail,F,14,64.3,90
Karen,F,12,56.3,77
Kathy,F,12,59.8,84.5
Mary,F,15,66.5,112
Sandy,F,11,51.3,50.5
Sharon,F,15,62.5,112.5
Tammy,F,14,62.8,102.5
```

```
Alfred,M,14,69,112.5
Duke,M,14,63.5,102.5
Guido,M,15,67,133
James,M,12,57.3,83
Jeffrey,M,13,62.5,84
John,M,12,59,99.5
Philip,M,16,72,150
Robert,M,12,64.8,128
Thomas,M,11,57.5,85
William,M,15,66.5,112
```

最简单用 `read_csv()` 读入上述 CSV 文件，程序如：

```
d.class <- read_csv('class.csv')
```

```
## Parsed with column specification:
## cols(
##   name = col_character(),
##   sex = col_character(),
##   age = col_integer(),
##   height = col_double(),
##   weight = col_double()
## )
```

```
knitr::kable(d.class)
```

name	sex	age	height	weight
Alice	F	13	56.5	84.0
Becka	F	13	65.3	98.0
Gail	F	14	64.3	90.0
Karen	F	12	56.3	77.0
Kathy	F	12	59.8	84.5
Mary	F	15	66.5	112.0
Sandy	F	11	51.3	50.5
Sharon	F	15	62.5	112.5
Tammy	F	14	62.8	102.5
Alfred	M	14	69.0	112.5
Duke	M	14	63.5	102.5
Guido	M	15	67.0	133.0
James	M	12	57.3	83.0
Jeffrey	M	13	62.5	84.0
John	M	12	59.0	99.5
Philip	M	16	72.0	150.0
Robert	M	12	64.8	128.0
Thomas	M	11	57.5	85.0
William	M	15	66.5	112.0

从结果看出，读入后显示了每列的类型。对性别变量，没有自动转换成因子，而是保存为字符型。为了按自己的要求转换各列类型，用了 `read_csv()` 的 `coltypes=` 选项和 `cols()` 函数如下：

```
ct <- cols(
  .default = col_double(),
  name=col_character(),
  sex=col_factor(levels=c("M", "F"))
)
d.class <- read_csv('class.csv', col_types=ct)
str(d.class)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    19 obs. of  5 variables:
## $ name   : chr  "Alice" "Becka" "Gail" "Karen" ...
## $ sex    : Factor w/ 2 levels "M","F": 2 2 2 2 2 2 2 2 2 1 ...
## $ age    : num   13 13 14 12 12 15 11 15 14 14 ...
## $ height: num   56.5 65.3 64.3 56.3 59.8 66.5 51.3 62.5 62.8 69 ...
## $ weight: num    84 98 90 77 84.5 ...
## - attr(*, "spec")=List of 2
## ..$ cols   :List of 5
## .. ..$ name : list()
```

```
## .. ..- attr(*, "class")= chr "collector_character" "collector"
## .. ..$ sex :List of 3
## .. ..$ levels : chr "M" "F"
## .. ..$ ordered : logi FALSE
## .. ..$ include_na: logi FALSE
## .. ..- attr(*, "class")= chr "collector_factor" "collector"
## .. ..$ age : list()
## .. ..- attr(*, "class")= chr "collector_double" "collector"
## .. ..$ height: list()
## .. ..- attr(*, "class")= chr "collector_double" "collector"
## .. ..$ weight: list()
## .. ..- attr(*, "class")= chr "collector_double" "collector"
## ..$ default: list()
## ..- attr(*, "class")= chr "collector_double" "collector"
## ..- attr(*, "class")= chr "col_spec"
```

其中 `str()` 函数可以显示数据框的行数 (obs.) 和变量数 (variables), 以及每个变量 (列) 的类属等信息。

除了 `read_csv()` 函数以外, R 扩展包 `readr` 还提供了其它的从文本数据读入数据框的函数, 如 `read_table2()`, `read_tsv()`, `read_fwf()` 等。这些函数读入的结果保存为 `tibble`。`read_table2()` 读入用空格作为间隔的文本文件, 同一行的两个数据项之间可以用一个或多个空格分隔, 不需要空格个数相同, 也不需要上下对齐。`read_tsv()` 读入用制表符分隔的文件。`read_fwf()` 读入上下对齐的文本文件。

另外, `read_lines()` 函数将文本文件各行读入为一个字符型向量。`read_file()` 将文件内容读入成一整个字符串, `read_file_raw()` 可以不管文件编码将文件读入为一个二进制字符串。

对特别大的文本格式数据, `data.table` 扩展包的 `fread()` 读入速度更快。

`readr` 包的 `write_csv()` 函数将 `tibble` 保存为 `csv` 文件, 总是使用 UTF-8 编码。

文本格式的文件都不适用于大型数据的读取与保存。大型数据可以通过数据库接口访问, 可以用 R 的 `save()` 和 `load()` 函数按照 R 的格式访问, 还有一些特殊的针对大数据集的 R 扩展包。

15.3 Excel 表访问

15.3.1 借助于文本格式

为了把 Microsoft Excel 格式的数据读入到 R 中, 最容易的办法是在 Excel 软件中把数据表转存为 CSV 格式, 然后用 `read.csv()` 读取。

为了把 R 的数据框保存为 Excel 格式, 只要用 `write.csv()` 把数据框保存成 CSV 格式, 然后在 Excel 中打开即可。例如, 下面的程序演示了 `write_csv()` 的使用:

```
d1 <- tibble("学号"=c("101", "103", "104"),
             "数学"=c(85, 60, 73),
             "语文"=c(90, 78, 80))
write_csv(d1, path="tmp1.csv")
```

保存在文件中的结果显示如下：

```
学号,数学,语文
101,85,90
103,60,78
104,73,80
```

15.3.2 使用剪贴板

为了把 Excel 软件中数据表的选中区域读入到 R 中，可以借助于剪贴板。在 Excel 中复制选中的区域，然后在 R 中用如

```
myDF <- read.delim("clipboard")
```

就可以把选中部分转换成一个 R 的数据框。如果复制的区域不含列名，应加上 `header=FALSE` 选项。

这种方法也可以从 R 中复制数据到在 Excel 中打开的电子表格中，例如

```
write.table(iris, file="clipboard", sep = "\t", col.names = NA)
```

首先把指定的数据框（这里是 `iris`）写入到了剪贴板，然后在用 Excel 软件打开的工作簿中只要粘贴就可以。上述程序中 `write.table()` 函数把指定的数据框写入到指定的文件中，其中的 `col.names=NA` 选项是一个特殊的约定，这时保存的文件中第一行是列名，如果有行名的话，行名所在的列对应的列名是空白的（但是存在此项）。

如果从 R 中复制数据框到打开的 Excel 文件中时不带行名，但是带有列名，可以写这样一个通用函数

```
write.clipboard <- function(df){
  write.table(df, file="clipboard", sep='\t',
              row.names=FALSE)
}
```

15.3.3 利用 readxl 扩展包

`readxl` 扩展包的 `readxl()` 函数利用独立的 C 和 C++ 库函数读入 .xls 和 .xlsx 格式的 Excel 文件。一般格式为

```
read_excel(path, sheet = 1, col_names = TRUE,
            col_types = NULL, na = "", skip = 0)
```

结果返回读入的表格为一个数据框。各个自变量为：

- **path**: 要读入的 Excel 文件名，可以是全路径，路径格式要符合所用操作系统要求。
- **sheet**: 要读入哪个工作簿 (sheet)，可以是整数序号，也可以是工作簿名称的字符串。
- **col_names**: 是否用第一行内容作为列名，缺省为是。
- **col_types**: 可以在读入时人为指定各列的数据类型，缺省时从各列内容自动判断，有可能会不够准确。人为指定时，指定一个对应于各列的字符型向量，元素可取值为：
 - **blank**: 自动判断该列；
 - **numeric**: 数值型；
 - **date**: 日期；
 - **text**: 字符型。

15.3.4 利用 RODBC 访问 Excel 文件

还可以用 RODBC 扩展包访问 Excel 文件。这样的方法不需要借助于 CSV 文件这个中间格式。RODBC 是一个通过 ODBC 协议访问数据文件与数据库的 R 扩展包。

先给出把 R 数据框保存为 Excel 文件的例子。如下的程序定义了两个数据框：

```
d1 <- data.frame("学号"=c("101", "103", "104"),
                 "数学"=c(85, 60, 73),
                 "语文"=c(90, 78, 80))
d2 <- data.frame("学号"=c("101", "103", "104"),
                 "性别"=c("女", "男", "男"))
```

在写入到 Excel 文件时，如果文件已经存在，会导致写入失败。比如，要写入到 `testwrite.xls` 中，可以用如下程序在文件已存在时先删除文件：

```
fname <- "testwrite.xls"
if(file.exists(fname)) file.remove(fname)
```

其中 `file.exists()` 检查文件是否已存在，`file.remove()` 删除指定文件。

使用 RODBC 比较麻烦，需要先用 `odbcConnectExcel()` 函数打开目的文件，然后可以用 `sqlSave()` 函数把数据框保存到目的文件中，保存完毕后需要用 `close()` 函数关闭打开的目的文件。目前 RODBC 的 `odbcConnectExcel()` 只能在 32 位版本的 R 软件中使用，而且操作系统中必须安装有 32 位的 ODBC 驱动程序。示例如下（需要使用 32 位 R 软件且需要操作系统中有 32 位版本的 ODBC 驱动程序）：

```
library(RODBC)
con <- odbcConnectExcel(fname, readOnly=FALSE)
res <- sqlSave(con, d1, tablename="成绩",
              rownames=F, colnames=F, safer=T)
res <- sqlSave(con, d2, tablename="性别",
```

```
rownames=F, colnames=F, safer=T)
close(con)
```

用 `odbcConnectExcel2007()` 可以访问或生成 Excel 2007/2010 版本的.xlsx 文件, 此函数可以用在 64 位的 R 软件中, 但是这时需要操作系统中安装有 64 位的 ODBC 驱动程序, 而不能有 32 位的 ODBC 驱动程序。如果安装了 Office 软件, Office 软件是 32 位的, 相应的 ODBC 驱动程序必须也是 32 位的; Office 软件是 64 位的, 相应的 ODBC 驱动程序必须也是 64 位的。

RODBC 对 Excel 文件的支持还有一些其它的缺点, 比如表名不规范, 数据类型自动转换不一定合理等。在 Excel 中读入或者保存 CSV 格式会使得问题变得简单。大量数据或大量文件的问题就不应该使用 Excel 来管理了, 一般会使用关系数据库系统, 如 Oracle, MySQL 等。

为了读入 Excel 文件内容, 先用 `odbcConnectExcel()` 函数打开文件, 用 `sqlFetch()` 函数读入一个数据表为 R 数据框, 读取完毕后用 `close()` 关闭打开的文件。如

```
require(RODBC)
con <- odbcConnectExcel('testwrite.xls')
rd1 <- sqlFetch(con, sqtable='成绩')
close(con)
```

读入的表显示如下:

	学号	数学	语文
1	101	85	90
2	103	60	78
3	104	73	80

15.3.5 用 RODBC 访问 Access 数据库

RODBC 还可以访问其他微机数据库软件的数据库。假设有 Access 数据库在文件 `c:/Friends/birthdays.mdb` 中, 内有两个表 Men 和 Women, 每个表包含域 Year, Month, Day, First Name, Last Name, Death。域名应尽量避免用空格。

下面的程序把女性记录的表读入为 R 数据框:

```
require(RODBC)
con <- odbcConnectAccess("c:/Friends/birthdays.mdb")
women <- sqlFetch(con, sqtable='Women')
close(con)
```

RODBC 还有许多与数据库访问有关的函数, 比如, `sqlQuery()` 函数可以向打开的数据库提交任意符合标准的 SQL 查询。

15.4 使用专用接口访问数据库

15.4.1 访问 Oracle 数据库

Oracle 是最著名的数据库服务器软件。要访问的数据库，可以是安装在本机上的，也可以是安装在网络上某个服务器中的。如果是远程访问，需要在本机安装 Oracle 的客户端软件。

假设已经在本机安装了 Oracle 服务器软件，并设置 orcl 为本机安装的 Oracle 数据库软件或客户端软件定义的本地或远程 Oracle 数据库的标识，test 和 oracle 是此数据库的用户名和密码，testtab 是此数据库中的一个表。

为了在 R 中访问 Oracle 数据库服务器中的数据库，在 R 中需要安装 ROracle 包。这是一个源代码扩展包，需要用户自己编译安装。在 MS Windows 环境下，需要安装 R 软件和 RTools 软件包（在 CRAN 网站的 Windows 版本软件下载栏目中）。在 MS Windows 命令行窗口，用如下命令编译 R 的 ROracle 扩展包：

```
set OCI_LIB32=D:\oracle\product\10.2.0\db_1\bin
set OCI_INC=D:\oracle\product\10.2.0\db_1\oci\include
set PATH=D:\oracle\product\10.2.0\db_1\bin;C:\Rtools\bin;C:\Rtools\gcc-4.6.3\bin;"%PATH%"
C:\R\R-3.2.0\bin\i386\rcmd INSTALL ROracle_1.2-1.tar.gz
```

其中的前三个 set 命令设置了 Oracle 数据库程序或客户端程序链接库、头文件和可执行程序的位置，第三个 set 命令还设置了 RTools 编译器的路径。这些路径需要根据实际情况修改。这里的设置是在本机运行的 Oracle 10g 服务器软件的情况。最后一个命令编译 ROracle 扩展包，相应的 rcmd 程序路径需要改成自己的安装路径。

如果服务器在远程服务器上，设远程服务器的数据库标识名为 ORCL，本机需要安装客户端 Oracle instant client 软件，此客户端软件需要与服务器同版本号，如 instantclient-basic-win32-10.2.0.5.zip，这个软件不需要安装，只需要解压到一个目录如 C:\instantclient_10_2 中。在本机（以 MS Windows 操作系统为例）中，双击系统，选择高级-环境变量，增加如下三个环境变量：

```
NLS_LANG = SIMPLIFIED CHINESE_CHINA.ZHS16GBK
ORACLE_HOME = C:\instantclient_10_2
TNS_ADMIN = C:\instantclient_10_2
```

并在环境变量 PATH 的值的末尾增加 Oracle 客户端软件所在的目录 verb|C:\instantclient_10_2, 并与前面内容用分号分开。

然后，在 client 所在的目录 C:\instantclient_10_2 中增加如下内容的 tnsnames.ora 文件

```
orcl =
(DESCRIPTION =
  (ADDRESS = (PROTOCOL = TCP)(HOST = 192.168.1.102 )
    (PORT = 1521))
(CONNECT_DATA =
  (SERVER = DEDICATED)
```

```

        (SERVICE_NAME = orcl)
    )
)

```

其中 HOST 的值是安装 Oracle 服务器的服务器的 IP 地址, orcl 是一个服务器实例名, 能够在服务器端的 tnsnames.ora 文件中查到, 等号前面的 orcl 是对数据库给出的客户端别名, 这里就干脆用了和服务器的数据库标识名相同的名字 orcl。

不论是在本机的数据框服务器还是在本机安装设置好客户端后, 在 R 中用如下的程序可以读入数据库中的表:

```

require(ROracle)
drv <- dbDriver("Oracle")

conn <- dbConnect(drv, username="test",
                  password="oracle", dbname="orcl")

rs <- dbSendQuery(conn, "select * from testtab")
d <- fetch(rs)

```

可以用 dbGetTable() 取出一个表并存入 R 数据框中。用 dbSendQuery() 发出一个 SQL 命令, 用 fetch() 可以一次性取回或者分批取回, 在表行数很多时这种方法更适用。

15.4.2 MySQL 数据库访问

MySQL 是高效、免费的数据库服务器软件, 在很多行业尤其是互联网行业占有很大的市场。为了在 R 中访问 MySQL 数据库, 只要安装 RMySQL 扩展包 (有二进制版本)。假设服务器地址在 192.168.1.111, 可访问的数据库名为 world, 用户为 test, 密码为 mysql。设 world 库中有表 country。

在 R 中要访问 MySQL 数据框, 首先要建立与数据库服务器的连接:

```

con <- dbConnect(RMySQL::MySQL(),
                 dbname='world',
                 username='test', password='mysql',
                 host='192.168.1.111')

```

下列代码列出 world 库中的所有表, 然后列出其中的 country 表的所有变量:

```

dbListTables(con)
dbListFields(con, 'country')

```

下列代码取出 country 表并存入 R 数据框 d.country 中:

```

d.country <- dbReadTable(con, 'country')

```

下列代码把 R 中的示例数据框 USArrests 写入 MySQL 库 world 的表 arrests 中:

```
data(USArrests)
dbWriteTable(con, 'arrests', USArrests,
             overwrite=TRUE)
```

当然，这需要用户对该库有写权限。

可以用 `dbGetQuery()` 执行一个 SQL 查询并返回结果，如

```
dbGetQuery(con, 'select count(*) from arrests')
```

当表很大时，可以用 `dbSendQuery()` 发送一个 SQL 命令，返回一个查询结果指针对象，用 `dbFetch()` 从指针对象位置读取指定行数，用 `dbHasCompleted()` 判断是否已读取结束。如

```
res <- dbSendQuery(con, "SELECT * FROM country")
while(!dbHasCompleted(res)){
  chunk <- dbFetch(res, n = 5)
  print(chunk[,1:2])
}
dbClearResult(res)
```

数据库使用完毕时，需要关闭用 `dbConnect()` 打开的连接：

```
dbDisconnect(con)
```

15.5 文件访问

15.5.1 连接

输入输出可以针对命令行，针对文件，R 支持扩展的文件类型，称为“连接 (connection)”。

函数 `file()` 生成到一个普通文件的连接，函数 `url()` 生成一个到指定的 URL 的连接，函数 `gzfile`, `bzfile`, `xzfile`, `unz` 支持对压缩过的文件的访问（不是压缩包，只对一个文件压缩）。这些函数大概的用法如下：

```
file("path", open="", blocking=T,
     encoding = getOption("encoding"),
     raw = FALSE)

url(description, open = "", blocking = TRUE,
     encoding = getOption("encoding"))

textConnection(description, open="r",
               local = FALSE,
```

```
encoding = c("", "bytes", "UTF-8"))

gzfile(description, open = "",
        encoding = getOption("encoding"),
        compression = 6)

bzfile(description, open = "",
        encoding = getOption("encoding"),
        compression = 9)

xzfile(description, open = "",
        encoding = getOption("encoding"),
        compression = 6)

unz(description, filename, open = "",
     encoding = getOption("encoding"))
```

生成连接的函数不自动打开连接。给定一个未打开的连接，读取函数从中读取时会自动打开连接，函数结束时自动关闭连接。用 `open()` 函数打开连接，返回一个句柄；生成连接时可以用 `open` 参数要求打开连接。要多次从一个连接读取时就应该先打开连接，读取完毕用 `close` 函数关闭。

函数 `textConnection()` 打开一个字符串用于读写。

在生成连接与打开连接的函数中用 `open` 参数指定打开方式，取值为：

- `r`—文本型只读；
- `w`—文本型只写；
- `a`—文本型末尾添加；
- `rb`—二进制只读；
- `wb`—二进制只写；
- `ab`—二进制末尾添加；
- `r+` 或 `r+b`—允许读和写；
- `w+` 或 `w+b`—允许读和写，但刚打开时清空文件；
- `a+` 或 `a+b`—末尾添加并允许读。

15.5.2 文本文件访问

函数 `readLines()`, `scan()` 可以从一个文本型连接读取。

给定一个打开的连接 `con`，用 `readLines` 函数可以把文件各行读入为字符型向量的各个元素。可以指定要读的行数。如

```
ll <- readLines(file('class.csv'))
print(ll)
```

用 `writeLines` 函数可以把一个字符型向量各元素作为不同行写入一个文本型连接。如

```
vnames <- strsplit(ll, ',')[[1]]
writeLines(vnames, con='class-names.txt')
```

其中的 `con` 参数应该是一个打开的文本型写入连接，但是可以直接给出一个要写入的文件名。

15.5.3 二进制文件访问

函数 `save` 用来保存 R 变量到文件，函数 `load` 用来从文件中读取保存的 R 变量。

函数 `readBin` 和 `writeBin` 对 R 变量进行二进制文件存取。

如果要访问其它软件系统的二进制文件，请参考 R 手册中的“R Data Import/Export Manual”。

15.5.4 字符型连接

函数 `textConnection` 打开一个字符串用于读取或写入，是很好用的一个 R 功能。可以把一个小文件存放在一个长字符串中，然后用 `textConnection` 读取，如

```
fstr <-
"name,score
王芳,78
孙莉,85
张聪,80
"
d <- read.csv(textConnection(fstr), header=T)
print(d)
```

读取用的 `textConnection` 的参数是一个字符型变量。

在整理输出结果时，经常可以向一个字符型变量连接写入，最后再输出整个字符串值。例如：

```
tc <- textConnection("sres", open="w")
cat('Trial of text connection.\n', file=tc)
cat(1:10, '\n', file=tc, append=T)
close(tc)
print(sres)
```

注意写入用的 `textConnection` 的第一个参数是保存了将要写入的字符型变量名的字符串，而不是变量名本身，第二个参数表明是写入操作，使用完毕需要用 `close` 关闭。

15.6 目录和文件管理

目录和文件管理函数:

- `getwd()`—返回当前工作目录。
- `setwd(path)`—设置当前工作目录。
- `list.files()` 或 `dir()`—查看目录中内容。`list.files(pattern='.*[.]r$')` 可以列出所有以“r”结尾的文件。
- `file.path()`—把目录和文件名组合得到文件路径。
- `file.info(filename)`—显示文件的详细信息。
- `file.exists()`—查看文件是否存在。
- `file.access()`—考察文件的访问权限。
- `create.dir()`—新建目录。
- `file.create()`—生成文件。
- `file.remove()` 或 `unlink()`—删除文件。`unlink()` 可以删除目录。
- `file.rename()`—为文件改名。
- `file.append()`—把两个文件相连。
- `file.copy()`—复制文件。
- `basename()` 和 `dirname()`— 从一个全路径文件名获取文件名和目录。

Chapter 16

程序控制结构

16.1 表达式

R 是一个表达式语言, 其任何一个语句都可以看成是一个表达式。表达式之间以分号分隔或用换行分隔。表达式可以续行, 只要前一行不是完整表达式 (比如末尾是加减乘除等运算符, 或有未配对的括号) 则下一行为上一行的继续。若干个表达式可以放在一起组成一个复合表达式, 作为一个表达式使用, 复合表达式的值为最后一个表达式的值, 组合用大括号表示, 如:

```
{  
  x <- 15  
  x  
}
```

16.2 分支结构

分支结构包括 if 结构:

if (条件) 表达式 1

或

if (条件) 表达式 1 else 表达式 2

其中的“条件”为一个标量的真或假值, 表达式可以用大括号包围的复合表达式。如

```
if(is.na(lambda)) lambda <- 0.5
```

又如

```
if(x>1) {  
  y <- 2.5  
} else {  
  y <- -y  
}
```

16.2.1 用逻辑下标代替分支结构

R 是向量化语言，尽可能少用标量运算。比如， x 为一个向量，要定义 y 与 x 等长，且 y 的每一个元素当且仅当 x 的对应元素为正数时等于 1，否则等于零。

这样是错误的：

```
if(x>0) y <- 1 else y <- 0
```

正解为：

```
y <- numeric(length(x))  
y[x>0] <- 1  
y
```

函数 `ifelse()` 可以根据一个逻辑向量中的多个条件，分别选择不同结果。如

```
x <- c(-2, 0, 1)  
y <- ifelse(x >= 0, 1, 0); print(y)
```

```
## [1] 0 1 1
```

函数 `switch()` 可以建立多分枝结构。

16.3 循环结构

16.3.1 计数循环

为了对向量每个元素、矩阵每行、矩阵每列循环处理，语法为

`for(循环变量 in 序列) 语句`

其中的语句一般是复合语句。如：

```
x <- rnorm(5)  
y <- numeric(length(x))  
for(i in 1:5){  
  if(x[i]>=0) y[i] <- 1 else y[i] <- 0
```



```
}
print(y)
```

```
## [1] 0 1 1 1 0
```

其中 `rnorm(5)` 会生成 5 个标准正态分布随机数。`numeric(n)` 生成有 n 个 0 的数值型向量（基础类型为 `double`）。

如果需要对某个向量 x 按照下标循环，获得所有下标序列的标准写法是 `seq(along=x)`，而不用 `1:n` 的写法，因为在特殊情况下 n 可能等于零，这会导致错误下标，而 `seq(along=x)` 在 x 长度为零时返回零长度的下标。

例如，设序列 x_n 满足 $x_0 = 0$, $x_n = 2x_{n-1} + 1$, 求 $S_n = \sum_{i=1}^n x_n$:

```
x <- 0.0
s <- x
n <- 5
for(i in 2:n){
  x <- 2*x + 1
  s <- s + x
}
print(s)
```

```
## [1] 26
```

在 R 中应尽量避免 `for` 循环：其速度比向量化版本慢一个数量级以上，而且写出的程序不够典雅。比如，前面那个随机数例子实际上可以简单地写成

```
x <- rnorm(5)
y <- ifelse(x >= 0, 1, 0)
print(y)
```

```
## [1] 1 1 1 0 0
```

16.3.2 while 循环和 repeat 循环

用

`while(循环继续条件)` 语句

进行当型循环。其中的语句一般是复合语句。仅当条件成立时才继续循环，而且如果第一次条件就已经不成立就一次也不执行循环内的语句。

用

`repeat` 语句

进行无条件循环（一般在循环体内用 `if` 与 `break` 退出）。其中的语句一般是复合语句。如下的写法可以制作一个直到型循环：

```
repeat{
  ...
  if(循环退出条件) break
}
```

直到型循环至少执行一次，每次先执行... 代表的循环体语句，然后判断是否满足循环退出条件，满足条件就退出循环。

用 `break` 语句退出所在的循环。用 `next` 语句进入所在循环的下一轮。

例如，常量 e 的值可以用泰勒展开式表示为

$$e = 1 + \sum_{k=1}^{\infty} \frac{1}{k!}$$

R 函数 `exp(1)` 可以计算 e 的。为了计算 e 的值，下面用泰勒展开逼近计算 e 的值：

```
e0 <- exp(1.0)
s <- 1.0
x <- 1
k <- 0
repeat{
  k <- k+1
  x <- x/k
  s <- s + x

  if(x < .Machine$double.eps) break
}
err <- s - e0
cat("k=", k, " s=", s, " e=", e0, " 误差 =", err, "\n")
```

```
## k= 18  s= 2.718282  e= 2.718282  误差= 4.440892e-16
```

其中 `.Machine$double.eps` 称为机器 ε ，是最小的加 1 之后可以使得结果大于 1 的正双精度数，小于此数的正双精度数加 1 结果还等于 1。用泰勒展开公式计算的结果与 `exp(1)` 得到的结果误差在 10^{-16} 左右。

16.4 R 中判断条件

`if` 语句和 `while` 语句中用到条件。条件必须是标量值，而且必须为 `TRUE` 或 `FALSE`，不能为 `NA` 或零长度。这是 R 编程时比较容易出错的地方。

16.5 管道控制

数据处理中经常会对同一个变量（特别是数据框）进行多个步骤的操作，比如，先筛选部分有用的变量，再定义若干新变量，再排序。R 的 `magrittr` 包提供了一个 `%>%` 运算符实现这样的操作流程。比如，变量 `x` 先用函数 `f(x)` 进行变换，再用函数 `g(x)` 进行变换，一般应该写成 `g(f(x))`，用 `%>%` 运算符，可以表示成 `x %>% f() %>% g()`。更多的处理，如 `h(g(f(x)))` 可以写成 `x %>% f() %>% g() %>% h()`。这样的表达更符合处理发生的次序，而且插入一个处理步骤也很容易。

处理用的函数也可以带有其它自变量，在管道控制中不要写第一个自变量。某个处理函数仅有一个自变量时，可以省略空的括号。

`tibble` 类型的数据框尤其适用于如此的管道操作。

将管道控制开始变量设置为 `.`，可以定义一个函数。

`magrittr` 包定义了 `%T%` 运算符，`x %T% f()` 返回 `x` 本身而不是用 `f()` 修改后的返回值 `f(x)`，这在中间步骤需要显示或者绘图但是需要进一步对输入数据进行处理时有用。

`magrittr` 包定义了 `%%$%` 运算符，此运算符的作用是将左运算元的各个变量（这时左运算元是数据框或列表）暴露出来，可以直接在右边调用其中的变量，类似于 `with()` 函数的作用。

`magrittr` 包定义了 `%<>%` 运算符，用在管道链的第一个连接，可以将处理结果存入最开始的变量中，类似于 C 语言的 `+=` 运算符。

如果一个操作是给变量加 `b`，可以写成 `add(b)`，给变量乘 `b`，可以写成 `multiply_by(b)`。

Chapter 17

函数

17.1 函数基础

17.1.1 介绍

在现代的编程语言中使用自定义函数，优点是代码复用、模块化设计。

在编程时，把编程任务分解成小的模块，每个模块用一个函数实现，可以降低复杂性，防止变量混杂。

函数的自变量是只读的，函数中定义的局部变量只在函数运行时起作用，不会与外部或其它函数中同名变量混杂。

函数返回一个对象作为输出，如果需要返回多个变量，可以用列表进行包装。

17.1.2 函数定义

函数定义使用 `function` 关键字，一般格式为

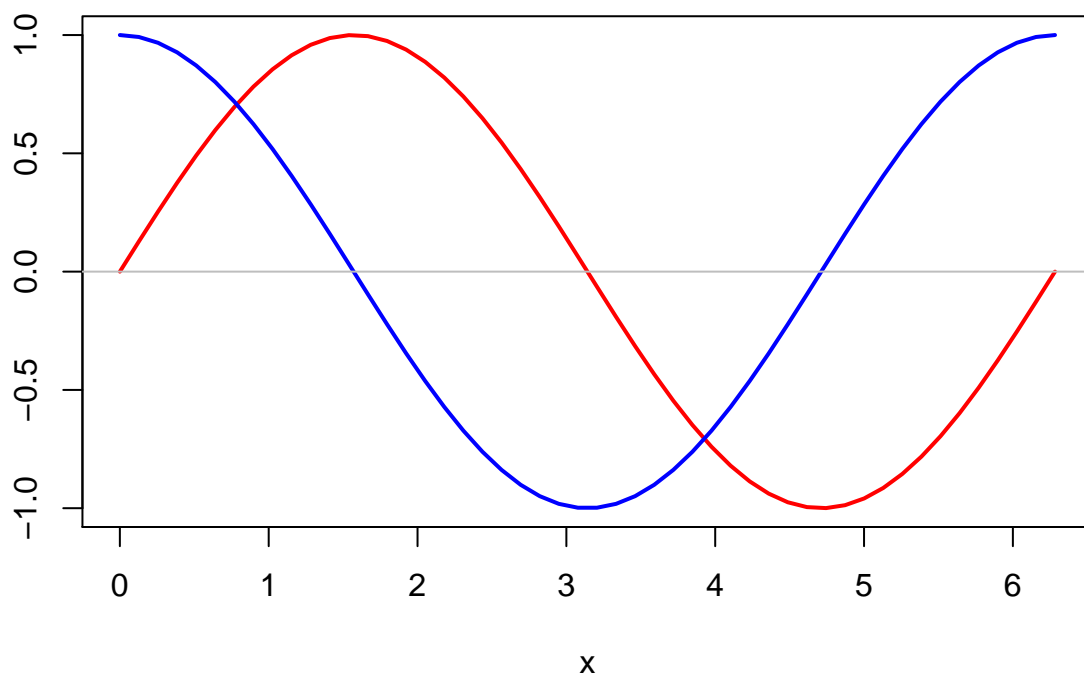
函数名 <- function(形式参数表) 函数体

函数体是一个表达式或复合表达式（复合语句），以复合表达式中最后一个表达式为返回值，也可以用 `return(x)` 返回 `x` 的值。如果函数需要返回多个结果，可以打包在一个列表（list）中返回。形式参数表相当于函数自变量，可以是空的，形式参数可以有缺省值，R 的函数在调用时都可以用“形式参数名 = 实际参数”的格式输入自变量值。

下面的例子没有参数，仅画一个示例图：

```
f <- function() {  
  x <- seq(0, 2*pi, length=50)  
  y1 <- sin(x)  
  y2 <- cos(x)
```

```
plot(x, y1, type='l', lwd=2, col='red',
     xlab='x', ylab='')
lines(x, y2, lwd=2, col='blue')
abline(h=0, col='gray')
}
f()
```



注意此自定义函数虽然没有参数，但是在定义与调用时都不能省略圆括号。

自定义函数也可以是简单的一元函数，与数学中一元函数基本相同，例如

```
f <- function(x) 1/sqrt(1 + x^2)
```

基本与数学函数 $f(x) = 1/\sqrt{1+x^2}$ 相对应。定义中的自变量 x 叫做形式参数或形参 (formal arguments)。函数调用时，形式参数得到实际值，叫做实参 (actual arguments)。R 函数有一个向量化的好处，在上述函数调用时，如果形式参数 x 的实参是一个向量，则结果也是向量，结果元素为实参向量中对应元素的变换值。如

```
f(0)
```

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

```
f(c(-1, 0, 1, 2))
```

```
## [1] 0.7071068 1.0000000 0.7071068 0.4472136
```

第一次调用时，形式参数 x 得到实参 0，第二次调用时，形式参数 x 得到向量实参 $c(-1, 0, 1, 2)$ 。

函数实参是向量时，函数体中也可以计算对向量元素进行汇总统计的结果。例如，设 x_1, x_2, \dots, x_n 是一个总体的简单随机样本，其样本偏度统计量定义如下：

$$\hat{w} = \frac{n}{(n-1)(n-2)} \sum_{i=1}^n \left(\frac{x_i - \bar{x}}{S} \right)^3$$

其中 \bar{x} 与 S 分别是样本均值与样本标准差。如下的 R 函数可以把观测样本的值保存在一个向量中输入，计算并输出其样本偏度统计量值：

```
f <- function(x) {
  n <- length(x)
  xbar <- mean(x)
  S <- sd(x)
  n/(n-1)/(n-2)*sum( (x - xbar)^3 ) / S^3
}
```

函数体的最后一个表达式是函数返回值。

在函数体最后一个表达式中巧妙地利用了 R 的向量化运算 $((x - \text{xbar})^3)$ 与内建函数 (sum) 。这比用 `for` 循环计算效率高得多，计算速度相差几十倍。

请比较如下两个表达式：

```
n/(n-1)/(n-2)*sum( (x - xbar)^3 ) / S^3
n/(n-1)/(n-2)*sum( ((x - xbar)/S)^3 )
```

这两个表达式的值相同。表面上看，第二个表达式更贴近原始数学公式，但是在编程时，需要考虑计算效率问题，第一个表达式关于 S 只需要除一次，而第二个表达式关于 S 除了 n 次，所以第一个表达式效率更高。

函数定义中的形式参数可以有多个，还可以指定缺省值。例如

```
fsub <- function(x, y=0){
  cat("x=", x, " y=", y, "\n")
  x - y
}
```

这里 x, y 是形式参数，其中 y 指定了缺省值为 0，有缺省值的形式参数在调用时可以省略对应的实参，省略时取缺省值。

实际上，“`function(参数表)` 函数体”这样的结构本身也是一个表达式，其结果是一个函数对象。在通常的函数定义中，函数名只不过是赋值给某个函数对象，或者说是“绑定”(bind) 到某个函数对象上面。R 允许使用没有函数名的函数对象。

因为函数也是 R 对象，也可以拥有属性。所谓对象，就是 R 的变量所指向的各种不同类型的统称。

一个自定义 R 函数由三个部分组成：函数体 `body()`，即要函数定义内部要执行的代码；`formals()`，即函数的形式参数表以及可能存在的缺省值；`environment()`，是函数定义时所处的环境，这会影响到参数表中缺省值与函数体中非局部变量的的查找。注意，函数名并不是函数对象的必要组成部分。如

```
body(fsub)
```

```
## {  
##   cat("x=", x, " y=", y, "\n")  
##   x - y  
## }
```

```
formals(fsub)
```

```
## $x  
##  
##  
## $y  
## [1] 0
```

```
environment(fsub)
```

```
## <environment: R_GlobalEnv>
```

“环境”是 R 语言比较复杂的概念，后面再详细解释。

17.1.3 函数调用

函数调用时最基本的调用方式是把实参与形式参数按位置对准，这与我们在数学中使用多元函数的习惯类似。例如

```
fsub(3, 1)
```

```
## x= 3  y= 1  
## [1] 2
```

相当于以 `x=3, y=1` 调用。

调用时可选参数可以省略实参，如

```
fsub(3)
```

```
## x= 3  y= 0  
## [1] 3
```

相当于以 `x=3, y=0` 调用。

R 函数调用时全部或部分形参对应的实参可以用“形式参数名 = 实参”的格式给出，这样格式给出的实参不用考虑次序，不带形式参数名的则按先后位置对准。如

```
fsub(x=3, y=1)
## x= 3  y= 1
## [1] 2
fsub(y=1, x=3)
## x= 3  y= 1
## [1] 2
fsub(x=3)
## x= 3  y= 0
## [1] 3
fsub(3, y=1)
## x= 3  y= 1
## [1] 2
fsub(1, x=3)
## x= 3  y= 1
## [1] 2
fsub(x=3, 1)
## x= 3  y= 1
## [1] 2
```

注意作为好的程序习惯应该避免 `fsub(x=3, 1)` 这样的做法。虽然 R 的语法没有强行要求，调用 R 函数时，如果既有按位置对应的参数又有带名参数，按位置对应的参数都写在前面，带名参数写在后面，不遵守这样的约定容易使得程序被误读。

R 的形参、实参对应关系可以写成一个列表，如 `fsub(3, y=1)` 中的对应关系可以写成列表 `list(3, y=1)`，如果调用函数的形参、实参对应关系保存在列表中，可以用函数 `do.call()` 来表示函数调用，如

```
do.call(fsub, list(3, y=1))
```

与

```
fsub(3, y=1)
```

效果相同。

在自定义 R 函数的形参中，还允许有一个特殊的... 形参（三个小数点）。在函数调用时，所有没有形参与之匹配的实参，不论是带有名字还是不带有名字的，都自动归入这个参数，这个参数的类型是一个列表。虽然很奇怪，这个语法在 R 里面是常用的，通常用来把函数内调用的其它函数的实参传递进来。

例如，`sapply(X, FUN, ...)` 中的形式参数 `FUN` 需要函数实参，此函数有可能需要更多的参数。例如，为了把 `1:5` 的每个元素都减去 2，可以写成

```
sapply(1:5, fsub, y=2)
```

```
## x= 1  y= 2
## x= 2  y= 2
## x= 3  y= 2
## x= 4  y= 2
## x= 5  y= 2

## [1] -1  0  1  2  3
```

或

```
sapply(1:5, fsub, 2)
```

```
## x= 1  y= 2
## x= 2  y= 2
## x= 3  y= 2
## x= 4  y= 2
## x= 5  y= 2

## [1] -1  0  1  2  3
```

实际上, R 语法中的大多数运算符如 `+`, `-`, `*`, `/`, `[`, `[[`, `(`, `{`等都是函数。这些特殊名字的函数要作为函数使用, 需要使用反向单撇号 ‘包围, 比如

```
1 + 2
```

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

```
`+`(1, 2)
```

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

效果相同。

这样, 为了给 `1:5` 每个元素减去 2, 还可以写成

```
sapply(1:5, `-`, 2)
```

```
## [1] -1  0  1  2  3
```

或

```
sapply(1:5, "-", 2)
```

```
## [1] -1  0  1  2  3
```

在后一写法中 `sapply` 的第二参数用了函数名字符串作为实参。

17.2 变量作用域

17.2.1 全局变量和工作空间

在所有函数外面（如 R 命令行）定义的变量是全局变量。在命令行定义的所有变量都保存在工作空间（workspace）中。用 `ls()` 查看工作空间内容。`ls()` 中加上 `pattern` 选项可以指定只显示符合一定命名模式的变量，如

```
ls(pattern='^tmp[.]')
```

显示所有以 `tmp.` 开头的变量。用 `object.size()` 函数查看变量占用存储大小。

因为 R 的函数调用时可以读取工作空间中的全局变量值，工作空间中过多的变量会引起莫名其妙的程序错误。用 `rm()` 函数删除指定的变量。`rm()` 中还可以用 `list` 参数指定一个要删除的变量名表。如

```
rm(list=ls(pattern='^tmp[.]'))
```

用 `save()` 函数保存工作空间中选择的某些变量；用 `load()` 函数载入保存在文件中的变量。如

```
save(my.large.data,  
     file='my-large-data.RData')  
load('my-large-data.RData')
```

实际上，R 的工作空间是 R 的变量搜索路径中的一层，大体相当于全局变量空间。R 的已启用的软件包中的变量以及用 `attach()` 命令引入的变量也在这个搜索路径中。

17.2.2 局部变量

在计算机语言中，“变量”实际是计算机内存中的一段存储空间。函数的参数（自变量）在定义时并没有对应的存储空间，所以也称函数定义中的参数为“形式参数”。

函数的形式参数在调用时被赋值为实参值（这是一般情形），形参变量和函数体内被赋值的变量都是局部的。这一点符合函数式编程 (functional programming) 的要求。所谓局部变量，就是仅在函数运行时才存在，一旦退出函数就不存在的变量。

17.2.2.1 自变量的局部性

在函数被调用时，形式参数（自变量）被赋值为实际的值（称为实参），如果实参是变量，形式参数实际变成了实参的一个副本，在函数内部对形式参数作任何修改在函数运行完成后都不影响原来的实参变量，而且函数运行完毕后形式参数对应的变量不再存在。

在下例中，在命令行定义了全局变量 `xv`, `x1`，然后作为函数 `f()` 的自变量值（实参）输入到函数中，函数中对两个形式参数作了修改，函数结束后实参变量 `xv`, `x1` 并未被修改，形参变量也消失了。例子程序如下：

```
xv <- c(1,2,3)
xl <- list(a=11:15, b='James')
if(exists("x")) rm(x)
f <- function(x, y){
  cat(' 输入的 x=', x, '\n')
  x[2] <- -1
  cat(' 函数中修改后的 x=', x, '\n')
  cat(' 输入的 y 为:\n'); print(y)
  y[[2]] <- 'Mary'
  cat(' 函数中修改过的 y 为:\n'); print(y)
}
f(xv, xl)
## 输入的 x= 1 2 3
## 函数中修改后的 x= 1 -1 3
## 输入的 y 为:
## $a
## [1] 11 12 13 14 15
##
## $b
## [1] "James"
##
## 函数中修改过的 y 为:
## $a
## [1] 11 12 13 14 15
##
## $b
## [1] "Mary"
##
cat(' 函数运行完毕后原来变量 xv 不变: ', xv, '\n')
## 函数运行完毕后原来变量 xv 不变: 1 2 3
cat(' 函数运行完毕后原来变量 xl 不变: :\n'); print(xl)
## 函数运行完毕后原来变量 xl 不变: :
## $a
## [1] 11 12 13 14 15
##
## $b
## [1] "James"
##
cat(' 函数运行完毕后形式参数 x 不存在: :\n'); print(x)
```

```
## 函数运行完毕后形式参数 x 不存在：
## Error in print(x) : object 'x' not found
```

R 语言的这种特点对于传递超大的数据是不利的，所以 R 中会容纳超大数据的类型往往涉及成修改副本时不占用不必要的额外存储空间，比如，tibble 类型就有这样的特点。

17.2.2.2 修改自变量

为了修改某个自变量，在函数内修改其值并将其作为函数返回值，赋值给原变量。

比如定义了如下函数：

```
f <- function(x, inc=1){
  x <- x + inc
  x
}
```

调用如

```
x <- 100
cat(' 原始 x=', x, '\n')
```

```
## 原始 x= 100
```

```
x <- f(x)
cat(' 修改后 x=', x, '\n')
```

```
## 修改后 x= 101
```

17.2.2.3 函数内的局部变量

在函数内部用赋值定义的变量都是局部变量，即使在工作空间中有同名的变量，此变量在函数内部被赋值时就变成了局部变量，原来的全局变量不能被修改。局部变量在函数运行结束后就会消失。如

```
if('x' %in% ls()) rm(x)
f <- function(){
  x <- 123
  cat(' 函数内: x = ', x, '\n')
}
f()
cat(' 函数运行完毕后: x=', x, '\n')
## 函数内: x = 123
> cat(' 函数运行完毕后: x=', x, '\n')
## Error in cat(" 函数运行完毕后: x=", x, "\n") : object 'x' not found
```

再比如，下面的函数试图知道自己被调用了多少次，但是因为每次函数调用完毕局部变量就消失，这样的程序不能达到目的：

```
f <- function(){
  if(!exists("runTimes")){
    runTimes <- 1
  } else {
    runTimes <- runTimes + 1
  }
  print(runTimes)
}
f()
```

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

```
f()
```

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

这个问题可以用 R 的 closure 来解决。

17.2.3 在函数内访问全局变量

函数内部可以读取全局变量的值，但一般不能修改全局变量的值。在现代编程指导思想中，全局变量容易造成不易察觉的错误，应谨慎使用，当然，也不是禁止使用，有些应用中不使用全局变量会使得程序更复杂且低效。

在下面的例子中，在命令行定义了全局变量 `x.g`，在函数 `f()` 读取了全局变量的值，但是在函数内给这样的变量赋值，结果得到的变量就变成了局部变量，全局变量本身不被修改：

```
x.g <- 9999
f <- function(x){
  cat(' 函数内读取：全局变量 x.g = ', x.g, '\n')
  x.g <- -1
  cat(' 函数内对与全局变量同名的变量赋值： x.g = ', x.g, '\n')
}
f()
```

```
## 函数内读取：全局变量 x.g = 9999
```

```
## 函数内对与全局变量同名的变量赋值： x.g = -1
```

```
cat(' 退出函数后原来的全局变量不变： x.g = ', x.g, '\n')
```

```
## 退出函数后原来的全局变量不变： x.g = 9999
```

在函数内部如果要修改全局变量的值，用 `<<-` 代替 `<-` 进行赋值。如

```
x.g <- 9999
f <- function(x){
  cat(' 函数内读取：全局变量 x.g = ', x.g, '\n')
  x.g <<- -1
  cat(' 函数内用"<<-" 对全局变量变量赋值： x.g = ', x.g, '\n')
}
f()
```

```
## 函数内读取：全局变量 x.g = 9999
```

```
## 函数内用"<<-"对全局变量变量赋值： x.g = -1
```

```
cat(' 退出函数后原来的全局变量被修改了： x.g = ', x.g, '\n')
```

```
## 退出函数后原来的全局变量被修改了： x.g = -1
```

后面将进一步解释函数在嵌套定义时 <<-的不同含义。

17.3 函数进阶

17.3.1 嵌套定义与句法作用域 (lexical scoping)

R 语言允许在函数体内定义函数。比如，

```
x <- -1
f0 <- function(x){
  f1 <- function(){
    x + 100
  }
  f1()
}
```

其中内嵌的函数 f1() 称为一个 closure(闭包)。

内嵌的函数体内在读取某个变量值时，如果此变量在函数体内还没有被赋值，它就不是局部的，会向定义的外面一层查找，外层一层找不到，就继续向外查找。上面例子 f1() 定义中的变量 x 不是局部变量，就向外一层查找，找到的会是 f0 的自变量 x，而不是全局空间中 x。如

```
f0(1)
```

```
## [1] 101
```

最后 x+100 中 x 取的是 f0 的实参值 x=1，而不是全局变量 x=-1。

这样的变量查找规则叫做句法作用域 (lexical scoping)，即函数运行时查找变量时，从其定义时的环境向外层逐层查找，而不是在运行时的环境中查找。句法作用域指的是可能有多个同名变量时查找变量按照定义时

的环境查找，不是指查找变量值的规则。

例如，

```
f0 <- function(){  
  f1 <- function(){  
    x <- -1  
    f2 <- function(){  
      x + 100  
    }  
    f2()  
  }  
  x <- 1000  
  f1()  
}  
f0()
```

```
## [1] 99
```

其中 `f2()` 运行时，用到的 `x` 是 `f1()` 函数体内的局部变量 `x=-1`，而不是被调用时 `f0()` 函数体内的局部变量 `x=1000`，所以结果是 $-1 + 100 = 99$ 。

“句法作用域”指的是函数调用时查找变量是查找其定义时的变量对应的存储空间，而不是定义时变量所取的历史值。函数运行时在找到某个变量对应的存储空间后，会使用该变量的当前值，而不是函数定义的时候该变量的历史值。例如

```
f0 <- function(){  
  x <- -1  
  f1 <- function(){  
    x + 100  
  }  
  x <- 1000  
  f1()  
}  
f0()
```

```
## [1] 1100
```

结果为什么不是 $-1 + 100 = 99$ 而是 $1000 + 100 = 1100$ ？这是因为，`f1()` 在调用时，使用的 `x` 是 `f0` 函数体内局部变量 `x` 的值，但是要注意的是程序运行时会访问该变量的当前值，即 1000，而不是函数定义的时候 `x` 的历史值-1。这个规则叫做“动态查找”(dynamic lookup)，句法作用域与动态查找一个说的是如何查找某个变量对应的存储空间，一个说的是使用该存储空间何时的存储值，程序运行时两个规则需要联合使用。

句法作用域不仅适用于查找变量，也适用于函数体内调用别的函数时查找函数。查找函数的规则与查找变量

规则相同。

17.3.1.1 辅助嵌套函数

有时内嵌函数仅仅是函数内用来实现模块化的一种工具，和正常的函数作用相同，没有任何特殊作用。例如，如下的程序在自变量 x 中输入一元二次方程 $ax^2 + bx + c = 0$ 的三个系数，输出解：

```
solve.sqe <- function(x){
  fd <- function(a, b, c) b^2 - 4*a*c
  d <- fd(x[1], x[2], x[3])
  if(d >= 0){
    return( (-x[2] + c(1,-1)*sqrt(d))/(2*x[1]) )
  } else {
    return( complex(real=-x[2], imag=c(1,-1)*sqrt(-d))/(2*x[1]) )
  }
}
```

在这个函数中内嵌的函数 `fd` 仅起到一个计算二次判别式公式的作用，没有用到任何的闭包特性，其中的形参变量 `a`, `b`, `c` 都是局部变量。运行如

```
solve.sqe(c(1, -2, 1))
```

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

```
solve.sqe(c(1, -2, 0))
```

```
## [1] 2 0
```

```
solve.sqe(c(1, -2, 2))
```

```
## [1] 1+1i 1-1i
```

17.3.1.2 泛函

许多函数需要用函数作为参数，称这样的函数为泛函。比如，`apply` 类函数。这样的函数具有很好的通用性，因为需要进行的操作可以输入一个函数来规定，输入的函数规定什么样的操作，

用户可以自定义这样的函数。比如，希望对一个数据框中所有的数值型变量计算某种统计量，用来计算统计量的函数作为参数输入：

```
summary.df.numeric <- function(df, FUN, ...){
  vn <- names(df)
  vn <- vn[vapply(df, is.numeric, TRUE)]
  if(length(vn) > 0){
    sapply(df[,vn, drop=FALSE], FUN, ...)
```

```

} else {
  numeric(0)
}
}

```

这里参数 FUN 是用来计算统计量的函数。例如对 d.class 中每个数值型变量计算最小值：

```

d.class <- readr::read_csv("class.csv")

## Parsed with column specification:
## cols(
##   name = col_character(),
##   sex = col_character(),
##   age = col_integer(),
##   height = col_double(),
##   weight = col_double()
## )

summary.df.numeric(d.class, min, na.rm=TRUE)

##      age height weight
##  11.0   51.3   50.5

```

17.3.1.3 函数工厂

利用嵌套定义在函数内的函数，可以解决上面的记录函数已运行次数的问题。如

```

f.gen <- function(){
  runTimes <- 0

  function(){
    runTimes <- runTimes + 1
    print(runTimes)
  }
}

f <- f.gen()
f()

## [1] 1

f()

## [1] 2

```

在此例中, `f.gen` 中有局部变量 `runTimes`, `f.gen()` 的输出是一个函数, 输出结果保存到变量名 `f` 中, 所以 `f` 是一个函数, 调用 `f` 时, 查找变量 `runTimes` 时, 如果 `f` 的局部变量中没有 `runTimes`, 就从其定义的环境中逐层向外查找, 在 `f` 定义中用了 `<-` 赋值, 这样赋值的含义是逐层向外查找变量是否存在, 在哪里找到变量就给那里的该变量赋值。`f` 调用时向外查找到的变量在 `f.gen` 的局部空间中, 这是 `f` 函数的定义环境, 函数的定义环境是随函数本身一同保存的, 所以起到了把变量值 `runTimes` 与函数共同使用的效果。定义在函数内的函数称为一个 closure(闭包)。closure 最重要的作用就是定义能够保存历史运行状态的函数。

上面的 `f.gen` 这样的函数称为一个函数工厂, 因为它的结果是一个函数, 而且是一个闭包。闭包在 R 中的主要作用是带有历史状态的函数。

下面的例子也用了 closure, 可以显示从上次调用到下次调用之间经过的时间:

```
make_stop_watch <- function(){
  saved.time <- proc.time()[3]

  function(){
    t1 <- proc.time()[3]
    td <- t1 - saved.time
    saved.time <- t1
    cat(" 流逝时间 (秒): ", td, "\n")
    invisible(td)
  }
}

ticker <- make_stop_watch()
ticker()
## 流逝时间 (秒):  0
for(i in 1:1000) sort(runif(10000))
ticker()
## 流逝时间 (秒):  1.53
```

其中 `proc.time()` 返回当前的 R 会话已运行的时间, 结果在 MS Windows 系统中有三个值, 分别是用户时间、系统时间、流逝时间, 其中流逝时间比较客观。

17.3.2 懒惰求值

R 函数在调用执行时, 除非用到某个形式变量的值才求出其对应实参的值。这一点在实参是常数时无所谓, 但是如果实参是表达式就不一样了。形参缺省值也是只有在函数运行时用到该形参的值时才求值。

例如,

```
f <- function(x, y=ifelse(x>0, TRUE, FALSE)){
  x <- -111
```

```

    if(y) x*2 else x*10
  }
f(5)

```

```
## [1] -1110
```

可以看出，虽然形参 x 输入的实参值为 5，但是这时形参 y 并没按 $x=5$ 被赋值为 `TRUE`，而是到函数体中第二个语句才被求值，这时 x 的值已经变成了 -111，故 y 的值是 `FALSE`。

17.3.3 递归调用

在函数内调用自己叫做递归调用。递归调用可以使得许多程序变得简单，但是往往导致程序效率很低，需谨慎使用。

R 中在递归调用时，最好用 `Recall` 代表调用自身，这样保证函数即使被改名（在 R 中函数是一个对象，改名后仍然有效）递归调用仍指向原来定义。

斐波那契数列是如下递推定义的数列：

$$x_0 = 0, \quad x_1 = 1$$

$$x_n = x_{n-2} + x_{n-1}$$

这个数列可以用如下递归程序自然地实现：

```

fib1 <- function(n){
  if(n == 0) return(0)
  else if(n == 1) return(1)
  else if(n >= 2) {
    Recall(n-1) + Recall(n-2)
  }
}
for(i in 0:10) cat('i =', i, ' x[i] =', fib1(i), '\n')

```

```

## i = 0  x[i] = 0
## i = 1  x[i] = 1
## i = 2  x[i] = 1
## i = 3  x[i] = 2
## i = 4  x[i] = 3
## i = 5  x[i] = 5
## i = 6  x[i] = 8
## i = 7  x[i] = 13
## i = 8  x[i] = 21
## i = 9  x[i] = 34
## i = 10 x[i] = 55

```

17.3.4 向量化

自定义的函数，如果其中的计算都是向量化的，那么函数自动地可以接受向量作为输入，结果输出向量。比如，将每个元素都变成原来的平方的函数：

```
f <- function(x){
  x^2
}
```

如果输入一个向量，结果也是向量，输出的每个元素是输入的对应该元素的相应的平方值。

但是，如下的分段函数：

$$g(x) = \begin{cases} x^2, & |x| \leq 1, \\ 1, & |x| > 1 \end{cases}$$

其一元函数版本可以写成

```
g <- function(x){
  if(abs(x) <= 1) {
    y <- x^2
  } else {
    y <- 1
  }

  y
}
```

但是这个函数不能处理向量输入，因为 `if` 语句的条件必须是标量条件。一个容易想到的修改是

```
gv <- function(x){
  y <- numeric(length(x))
  sele <- abs(x) <= 1
  y[sele] <- x[sele]^2
  y[!sele] <- 1.0

  y
}
```

或者

```
gv <- function(x){
  ifelse(abs(x) <= 1, x^2, 1)
}
```

对于没有这样简单做法的问题，可以将原来的逻辑包在循环中，如

```
gv <- function(x){
  y <- numeric(length(x))
  for(i in seq(along=x)){
    if(abs(x[i]) <= 1) {
      y[i] <- x[i]^2
    } else {
      y[i] <- 1
    }
  }
  y
}
```

函数 `Vectorize` 可以将这样的操作自动化。如

```
g <- function(x){
  if(abs(x) <= 1) {
    y <- x^2
  } else {
    y <- 1
  }
  y
}
gv <- Vectorize(g)
gv(c(-2, -0.5, 0, 0.5, 1, 1.5))
```

```
## [1] 1.00 0.25 0.00 0.25 1.00 1.00
```

17.3.5 纯函数与副作用

理想的自定义函数最好是像一般的数学函数那样，只要输入相同，输出也不变，而且除了利用输出值之外不能对程序环境做其它改变。这样的函数称为“纯函数”。R 的函数不能修改实参的值，这有助于实现纯函数的要求。

如果函数对相同的输入可以有不同的输出当然不是纯函数，例如 R 中的随机数函数 (`sample()`, `runif()`, `rnorm` 等)。

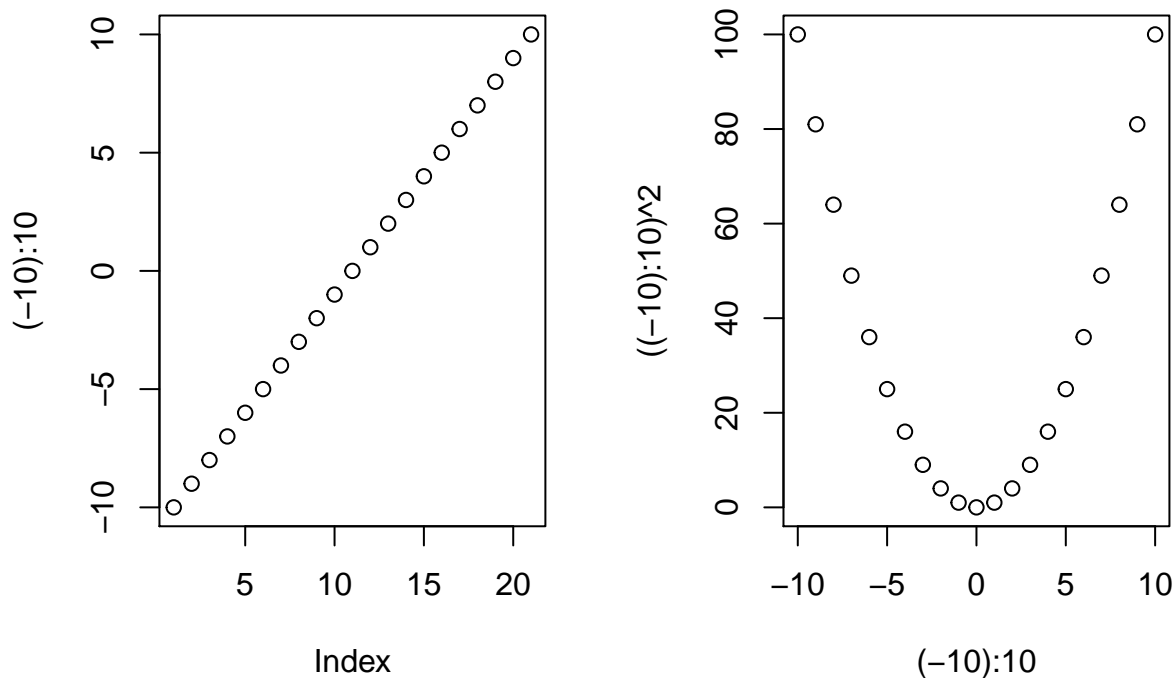
如果函数除了输出之外还在其它方面影响了运行环境，这样的函数就不是纯函数。所有画图函数 (`plot` 等)、输出函数 (`cat`, `print`, `save` 等) 都是这样的函数。这些对运行环境的改变叫做“副作用” (side effects)。又比如，`library()` 函数会引入新的函数和变量，`setwd()`, `Sys.setenv()`, `Sys.setlocale()` 会改变 R 运行环境，`options()`, `par()` 会改变 R 全局设置。自定义 R 函数中如果调用了非纯函数也就变成了非纯函

数。编程中要尽量控制副作用而且意识到副作用的影响，尤其是全局设置与全局变量的影响。

有些函数不可避免地要修改运行环境，如果可能的话，在函数结束运行前，应该恢复对运行环境的修改。为此，可以在函数体的前面部分调用 `on.exit()` 函数，此函数的参数是在函数退出前要执行的表达式或复合表达式。

例如，绘图的函数中经常需要用 `par()` 修改绘图参数，这会使得后续程序出错。为此，可以在函数开头保存原始的绘图参数，函数结束时恢复到原始的绘图参数。如

```
f <- function(){  
  opar <- par(mfrow=c(1,2))  
  on.exit(par(opar))  
  plot((-10):10)  
  plot((-10):10, ((-10):10)^2)  
}  
f()
```



如果函数中需要多次调用 `on.exit()` 指定多个恢复动作，除第一个调用的 `on.exit()` 以外都应该加上 `add=TRUE` 选项。

17.4 程序调试

17.4.1 跟踪调试

函数定义一般都包含多行，所以一般不在命令行定义函数，而是把函数定义写在源程序文件中，用 `source` 命令调入。用 `source` 命令调入运行的程序与在命令行运行的效果基本相同，这样定义的变量也是全局变量。

考虑如下函数定义：

```
f <- function(x){  
  for(i in 1:n){  
    s <- s + x[i]  
  }  
}
```

运行发现有错误：

```
f(1:5)  
## Error in f(1:5) : object 'n' not found
```

简单的函数可以直接仔细检查发现错误，用 `cat`，`print` 等输出中间结果查找错误。R 提供了一个 `browser()` 函数，在程序中插入对 `browser()` 函数的调用，可以进入跟踪调试状态，可以实时地查看甚至修改运行时变量的值。

程序运行遇到 `browser()` 函数时程序进入 Browser 的调试命令行。在调试命令行，用 `n` 命令逐句运行，用 `s` 命令跟踪进调用的函数内部逐句运行，用 `c` 命令恢复正常运行，用 `q` 命令强制终止程序运行。可以如同在 R 命令行一样查看变量的值或修改变量的值。在 RStudio 中进入跟踪状态后有相应的运行控制图标，可以用鼠标点击某行程序的行号设置断点，重新 `source()` 之后就可以在断点处进入跟踪状态。

为调试如上函数 `f` 的程序，在定义中插入对 `browser()` 的调用如：

```
f <- function(x){  
  browser()  
  for(i in 1:n){  
    s <- s + x[i]  
  }  
}
```

调试运行过程如下：

```
f(1:5)  
## Called from: f(1:5)  
## Browse[1]> n  
## debug at #3: for (i in 1:n) {
```



```
##      s <- s + x[i]
## }
## Browse[2]> n
## Error in f(1:5) : object 'n' not found
```

发现是在 `for(i in 1:n)` 行遇到未定义的变量 `n`。

在源文件中把出错行改为 `for(i in 1:length(x))`，再次运行，发现在运行 `s <- s + x[i]` 行时，遇到未定义的变量 `s`。这是忘记初始化引起的。在 `for` 语句前添加 `s <- 0` 语句，函数定义变成：

```
f <- function(x){
  browser()
  s <- 0
  for(i in 1:length(x)){
    s <- s + x[i]
  }
}
```

再次运行，在 `Browse[1]>` 提示下命令 `c` 表示恢复正常运行，程序不显示错误但是也没有显示求和结果。检查可以看出错误是忘记把函数返回值写在函数定义最后。

在函数定义最后添加 `s` 一行，再次运行，程序结果与手工验算结果一致。函数变成

```
f <- function(x){
  browser()
  n <- length(x)
  s <- 0
  for(i in 1:n){
    s <- s + x[i]
  }
  s
}
```

自定义函数应该用各种不同输入测试其正确性和稳定性。比如，上面的函数当自变量 `x` 为零长度向量时应该返回 `0` 才合适，但是上面的写法会返回一个 `numeric(0)` 结果，这个结果表示长度为零的向量：

```
f(numeric(0))
## Called from: f(numeric(0))
## Browse[1]> c
## numeric(0)
```

程序输入了零长度自变量，我们期望其输出为零而不是 `numeric(0)`。在自变量 `x` 为零长度时，函数中 `for(i in 1:length(x))` 应该一次都不进入循环，跟踪运行可以发现实际对 `i=1` 和 `i=0` 共运行了两轮循环。把这里的 `1:length(x)` 改成 `seq(along=x)` 解决了问题，`seq(along=x)` 生成 `x` 的下标序列，如果 `x` 是零长

度的则下标序列为零长度向量。

函数不需要修改后，可以把对 `browser()` 的调用删除或注释掉。函数最终修改为：

```
f <- function(x){
  s <- 0
  for(i in seq(along=x)){
    s <- s + x[i]
  }
  s
}
```

这里只是用这个简单函数演示如何调试程序，求向量和本身是不需要我们去定义新函数的，`sum` 函数本来就是完成这样的功能。实际上，许多我们认为需要自己编写程序作的事情，在 R 网站都能找到别人已经完成的程序。

17.4.2 出错调试选项

比较长的程序在调试时如果从开头就跟踪，比较耗时。可以设置成出错后自动进入跟踪模式，检查出错时的变量值。只要进行如下设置：

```
options(error=recover)
```

则在出错后可以选择进入出错的某一层函数内部，在 `browser` 环境中跟踪运行。

例如，如上设置后，前面那个求向量元素和的例子程序按最初的定义，运行时出现如下的选择：

```
## Error in f(1:5) : object 'n' not found
##
## Enter a frame number, or 0 to exit
##
## 1: f(1:5)
##
## Selection: f(1:5)
##
## Selection: 1
## Called from: top level
## Browse[1]>
```

在 `Selection` 后面输入了 1，就进入了函数内部跟踪。用 `Q` 终止运行并退出整个 `browser` 跟踪。当函数调用函数时可以选择进入哪一个函数进行跟踪。

17.4.3 警告处理

有些警告信息实际是错误，用 `options()` 的 `warn` 参数可以设置警告级别，如设置 `warn=2` 则所有警告当作错误处理。设置如

```
options(warn=2)
```

17.4.4 `stop()`、`warning()`、`message()`

编写程序时应尽可能提前发现不合法的输入和错误的状态。发现错误时，可以用 `stop(s)` 使程序运行出错停止，其中 `s` 是一个字符型对象，用来作为显示的出错信息。

发现某些问题后如果不严重，可以不停止程序运行，但用 `warning(s)` 提交一个警告信息，其中 `s` 是字符型的警告信息。警告信息的显示可能比实际运行要滞后一些。

函数 `message()` 与 `stop()`、`warning()` 类似，不算是错误或者警告，但仍算是某种非正常的信息输出。

17.4.5 预防性设计

在编写自定义函数时，可以检查自变量输入以确保输入符合要求。函数 `stopifnot` 可以指定自变量的若干个条件，当自变量不符合条件时自动出错停止。

例如，函数 `f()` 需要输入两个数值型向量 `x`, `y`, 需要长度相等，可以用如下的程序

```
f <- function(x, y){  
  stopifnot(is.numeric(x),  
            is.numeric(y),  
            length(x)==length(y))  
  ## 函数体程序语句...  
}
```

17.5 函数式编程介绍

R 可以算是一个函数式语言 (functional language):

1. R 语言的设计主要用函数求值来进行运算;
2. R 的用户主要使用函数调用来访问 R 的功能。

按照函数式编程的要求，每个 R 函数必须功能清晰、定义确切。比较容易控制的函数是纯函数，纯函数必须像数学中单值函数那样给定自变量输入有唯一确定的输出。比如，多个函数用全局变量传递信息，就不能算是纯函数。

R 支持类 (class) 和方法 (method)，实际提供了适用于多种自变量的通用函数 (generic function)，不同自变量类型调用该类特有的方法，但函数名可以保持不变。

函数式编程语言提供了定义纯函数的功能。这样的函数不能有副作用 (side effects)：函数返回值包含了函数执行的所有效果。函数定义仅由对所有可能的自变量值确定返回值来确定，不依赖于任何外部信息（也就不能依赖于全局变量与系统设置值）。函数定义返回值的方式是隐含地遍历所有可能的参数值给出返回值，而不是用过程式的计算来修改对象的值。

函数式编程的目的是提供可理解、可证明正确的软件。R 虽然带有函数式编程语言特点，但并不强求使用函数式编程规范。典型的函数式编程语言如 Haskell, Lisp 的运行与 R 的显式的、顺序的执行方式相差很大。

17.5.1 函数式编程的要求

- 没有副作用。调用一个函数对后续运算没有影响，不管是再次调用此函数还是调用其它函数。这样，用全局变量在函数之间传递信息就是不允许的。其它副作用包括写文件、打印、绘图等，这样的副作用对函数式要求破坏不大。
- 不受外部影响。函数返回值只依赖于其自变量及函数的定义。
- 不受赋值影响。函数定义不需要反复对内部对象（所谓“状态变量”）赋值或修改。

R 只能部分满足这些要求。一个 R 函数是否满足这些要求不仅要看函数本身，还要看函数内部调用的其它函数。

像 `options()` 函数这样修改全局运行环境的功能会破坏函数式要求。尽可能让自己的函数不依赖于 `options()` 中的参数。

与具体硬件、软件环境有关的一些因素也破坏纯函数要求，如不同的硬件常数、精度等。调用操作系统的功能对函数式要求破坏较大。减少赋值主要需要减少循环，可以用 R 的向量化方法解决。

17.5.2 Map、Reduce、Filter

R 提供了 `Map`, `Reduce`, `Filter`, `Find`, `Negate`, `Position` 等支持函数式编程的工具函数。这些函数包含对列表每一项进行变换，列表数据汇总，列表元素筛选等功能。

17.5.2.1 Map 函数

`Map()` 以一个函数作为参数，可以对其它参数的每一对应元素进行变换，结果为列表。

例如，对数据框 `d`，如下的程序可以计算每列的平方和：

```
d <- data.frame(  
  x = c(1, 7, 2),  
  y = c(3, 5, 9))  
Map(function(x) sum(x^2), d)
```

```
## $x
## [1] 54
##
## $y
## [1] 115
```

实际上，这个例子也可以用 `lapply()` 改写成

```
lapply(d, function(x) sum(x^2))
```

```
## $x
## [1] 54
##
## $y
## [1] 115
```

`Map()` 比 `lapply()` 增强的地方在于它允许对多个列表的对应元素逐一处理。例如，为了求出 `d` 中每一行的最大值，可以用

```
Map(max, d$x, d$y)
```

```
## [[1]]
## [1] 3
##
## [[2]]
## [1] 7
##
## [[3]]
## [1] 9
```

可以用 `unlist()` 函数将列表结果转换为向量，如

```
unlist(Map(max, d$x, d$y))
```

```
## [1] 3 7 9
```

17.5.2.2 Reduce 函数

`Reduce` 函数把输入列表（或向量）的元素逐次地用给定的函数进行合并计算。例如，

```
Reduce(sum, 1:4)
```

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

实际执行的是 $((1 + 2) + 3) + 4$ 。当然，求 `1:4` 的和只需要 `sum(1:4)`，但是 `Reduce` 可以对元素为复杂类型的列表进行逐项合并计算。

例如，`intersect` 函数可以计算两个集合的交集；对多个集合，如何计算交集？下面的例子产生了 4 个集合，然后反复调用 `intersect()` 求出了交集：

```
set.seed(2)
x <- replicate(4, sample(1:5, 5, replace=TRUE), simplify=FALSE); x
```

```
## [[1]]
## [1] 1 4 3 1 5
##
## [[2]]
## [1] 5 1 5 3 3
##
## [[3]]
## [1] 3 2 4 1 3
##
## [[4]]
## [1] 5 5 2 3 1
```

```
intersect(intersect(intersect(x[[1]], x[[2]]), x[[3]]), x[[4]])
```

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

也可以用 `magrittr` 包的 `%>%` 符号写成：

```
library(magrittr)
x[[1]] %>% intersect(x[[2]]) %>% intersect(x[[3]]) %>% intersect(x[[4]])
```

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

还可以写成循环：

```
y <- x[[1]]
for(i in 2:4) y <- intersect(y, x[[i]])
y
```

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

都比较繁琐。

利用 `Reduce` 函数，只要写成

```
Reduce(intersect, x)
```

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

`Reduce` 函数还可以用 `right` 参数选择是否从右向左合并，用参数 `init` 给出合并初值，用参数 `accumulate` 要求保留每一步合并的结果（累计）。这个函数可以把很多仅适用于两个运算元的运算推广到多个参数的情形。

17.5.2.3 Filter、Find、Position 函数

`Filter(f, x)` 用一个判断真假的一元函数 `f` 作为筛选规则，从列表或向量 `x` 中筛选出用 `f` 作用后为真值的元素子集。`f` 必须返回标量的 `TRUE` 或者 `FALSE`，这样的函数称为示性函数 (predicate functions)。例如

```
f <- function(x) x > 0 & x < 1
Filter(f, c(-0.5, 0.5, 0.8, 1))
```

```
## [1] 0.5 0.8
```

当然，这样的简单例子完全可以改写成：

```
f <- function(x) x > 0 & x < 1
x <- c(-0.5, 0.5, 0.8, 1)
x[x>0 & x < 1]
```

```
## [1] 0.5 0.8
```

但是，对于比较复杂的判断，特别是需要用许多个语句计算后进行的判断，就需要把判断写成一个函数，然后可以用 `Filter` 比较简单地表达按照判断规则取子集的操作。

`Find()` 作用与 `Filter()` 类似，但是仅返回满足条件的第一个，也可以用参数 `right=TRUE` 要求返回满足条件的最后一个。

`Position()` 作用与 `Find()` 类似，但不是返回满足条件的元素而是返回第一个满足条件的元素所在的下标位置。

Chapter 18

R 程序效率

18.1 R 的运行效率

R 是解释型语言，在执行单个运算时，效率与编译代码相近；在执行迭代循环时，效率较低，与编译代码的速度可能相差几十倍。R 以向量、矩阵为基础运算单元，在进行向量、矩阵运算时效率很高，应尽量采用向量化编程。

另外，R 语言的设计为了方便进行数据分析和统计建模，有意地使语言特别灵活，比如，变量为动态类型而且内容可修改，变量查找在当前作用域查找不到可以向上层以及扩展包中查找，函数调用时自变量仅在使用其值时才求值（懒惰求值），这样的设计都为运行带来了额外的负担，使得运行变慢。

在计算总和、元素乘积或者每个向量元素的函数变换时，应使用相应的函数，如 `sum`, `prod`, `sqrt`, `log` 等。

对于从其它编程语言转移到 R 语言的学生，如果不细究 R 特有的编程模式，编制的程序可能效率比正常 R 程序慢上几十倍，而且繁琐冗长。

为了提高 R 程序的运行效率，需要尽可能利用 R 的向量化特点，尽可能使用已有的高效函数，还可以把运行速度瓶颈部分改用 C++、FORTRAN 等编译语言实现，可以用 R 的 `profiler` 工具查找运行瓶颈。对于大量数据的长时间计算，可以借助于现代的并行计算工具。

对已有的程序，仅在运行速度不满意时才需要进行改进，否则没必要花费宝贵的时间用来节省几秒钟的计算机运行时间。要改善运行速度，首先要找到运行的瓶颈，这可以用专门的性能分析（`profiling`）功能实现。R 软件中的 `Rprof()` 函数可以执行性能分析的数据收集工作，收集到的性能数据用 `summaryRprof()` 函数可以显示运行最慢的函数。如果使用 RStudio 软件，可以用 **Profile** 菜单执行性能数据收集与分析，可以在图形界面中显示程序中哪些部分运行花费时间最多。

在改进已有程序的效率时，第一要注意的就是不要把原来的正确算法改成一个速度更快但是结果错误的算法。这个问题可以通过建立试验套装，用原算法与新算法同时试验看结果是否一致来避免。多种解决方案的正确性都可以这样保证，也可以比较多种解决方案的效率。

本章后面部分描述常用的改善性能的方法。对于涉及到大量迭代的算法，如果用 R 实现性能太差不能满足

要求，可以改成 C++ 编码，用 Rcpp 扩展包连接到 R 中。Rcpp 扩展包的使用将单独讲授。

R 的运行效率也受到内存的影响，占用内存过多的算法有可能受到物理内存大小限制无法运行，过多复制也会影响效率。

如果要想实现一个比较单纯的不需要利用 R 已有功能的算法，发现用 R 计算速度很慢的时候，也可以考虑先用 Julia 语言实现。Julia 语言设计比 R 更先进，运算速度快得多，运算速度经常能与编译代码相比，缺点是刚刚诞生几年的时间，可用的软件包还比较少。

18.2 向量化编程

18.2.1 示例 1

假设要计算如下的统计量：

$$w = \frac{1}{n} \sum_{i=1}^n |x_i - \hat{m}|,$$

其中 x_1, x_2, \dots, x_n 是某总体的样本， \hat{m} 是样本中位数。用传统的编程风格，把这个统计量的计算变成一个 R 函数，可能会写成：

```
f1 <- function(x){  
  n <- length(x)  
  mhat <- median(x)  
  s <- 0.0  
  for(i in 1:n){  
    s <- s + abs(x[i] - mhat)  
  }  
  s <- s/n  
  return(s)  
}
```

用 R 的向量化编程，函数体只需要一个表达式：

```
f2 <- function(x) mean( abs(x - median(x)) )
```

其中 `x - median(x)` 利用了向量与标量运算结果是向量每个元素与标量运算的规则，`abs(x - median(x))` 利用了 `abs()` 这样的一元函数如果以向量为输入就输出每个元素的函数值组成的向量的规则，`mean(...)` 避免了求和再除以 `n` 的循环也不需要定义多余的变量 `n`。

显然，第二种做法的程序比第一种做法简洁的多，如果多次重复调用，第二种做法的计算速度比第一种要快几十倍甚至上百倍。在 R 中，用 `system.time()` 函数可以求某个表达式的计算时间，返回结果的第 3 项是流逝时间。下面对 `x` 采用 10000 个随机数，并重复计算 1000 次，比较两个程序的效率：

```
nrep <- 1000
x <- runif(10000)
y1 <- numeric(nrep); y2 <- y1
system.time(for(i in 1:nrep) y1[i] <- f1(x) )[3]
## elapsed
##      10.08
system.time(for(i in 1:nrep) y1[i] <- f2(x) )[3]
## elapsed
##      0.48
```

速度相差二十倍以上。

有一个 R 扩展包 `microbenchmark` 可以用来测量比较两个表达式的运行时间。如:

```
x <- runif(10000)
microbenchmark::microbenchmark(
  f1(x),
  f2(x)
)

## Unit: microseconds
##      expr      min       lq      mean     median        uq      max neval
##  f1(x) 1778.635 1796.627 2174.6538 1821.5590 2061.8800 6811.246   100
##  f2(x)  351.615  360.354  460.1988  366.2655  471.3895 2068.562   100
```

就平均运行时间（单位：毫秒）来看，`f2()` 比 `f1()` 快大约 30 倍。

18.2.2 示例 2

假设要编写函数计算

$$f(x) = \begin{cases} 1 & x \geq 0, \\ 0 & \text{其它} \end{cases}$$

利用传统思维，程序写成

```
f1 <- function(x){
  n <- length(x)
  y <- numeric(n)

  for(i in seq(along=x)){
    if(x[i] >= 0) y[i] <- 1
    else y[i] <- 0
  }
}
```

```

    y
}

```

实际上, `y <- numeric(n)` 使得 `y` 的每个元素都初始化为 0, 所以程序中 `else y[i] <- 0` 可以去掉。

利用向量化与逻辑下标, 程序可以写成:

```

f2 <- function(x){
  n <- length(x)
  y <- numeric(n)
  y[x >= 0] <- 1

  y
}

```

但是, 利用 R 中内建函数 `ifelse()`, 可以把函数体压缩到仅用一个语句:

```

f2 <- function(x) ifelse(x >= 0, 1, 0)

```

18.2.3 示例 3

考虑一个班的学生存在生日相同的概率。假设一共有 365 个生日 (只考虑月、日)。设一个班有 n 个人, 当 n 大于 365 时 {至少两个人有生日相同} 是必然事件 (概率等于 1)。

当 n 小于等于 365 时:

$$\begin{aligned}
 &P(\text{至少有两人同生日}) \\
 &= 1 - P(n \text{ 个人生日彼此不同}) \\
 &= 1 - \frac{365 \times 364 \times \cdots \times (365 - (n - 1))}{365^n} \\
 &= 1 - \frac{365 - 0}{365} \cdot \frac{365 - 1}{365} \cdots \frac{365 - (n - 1)}{365}
 \end{aligned}$$

对 $n = 1, 2, \dots, 365$ 来计算对应的概率。完全用循环 (两重循环), 程序写成:

```

f1 <- function(){
  ny <- 365
  x <- numeric(ny)
  for(n in 1:ny){
    s <- 1
    for(j in 0:(n-1)){
      s <- s * (365-j)/365
    }
    x[n] <- 1 - s
  }
}

```

```
x
}
```

注意，不能先计算 $365 \times 364 \times \cdots \times (365 - (n - 1))$ 和 365^n 再相除，这会造成数值溢出。

用 `prod()` 函数可以向量化内层循环：

```
f2 <- function(){
  ny <- 365
  x <- numeric(ny)
  for(n in 1:ny){
    x[n] <- 1 - prod((365:(365-n+1))/365)
  }
  x
}
```

程序利用了向量与标量的除法，以及内建函数 `prod()`。

把程序用 `cumprod()` 函数改写，可以完全避免循环：

```
f3 <- function(){
  ny <- 365
  x <- 1 - cumprod((ny:1)/ny)
  x
}
```

用 `microbenchmark` 比较：

```
microbenchmark::microbenchmark(
  f1(),
  f2(),
  f3()
)
```

```
## Unit: microseconds
##  expr      min       lq      mean  median      uq      max neval
##  f1() 4423.969 4579.9850 5001.87630 4605.431 4674.3140 11856.196   100
##  f2()  473.960  505.0605  772.89922  521.767  707.0845  6982.428   100
##  f3()   1.542    2.8275   31.96412    4.113    5.9115  2702.395   100
```

`f2()` 比 `f1()` 快大约 35 倍，`f3()` 比 `f2()` 又快了大约 150 倍，`f3()` 比 `f1()` 快了五千倍以上！

18.3 减少显式循环

显式循环是 R 运行速度较慢的部分，有循环的程序也比较冗长，与 R 的向量化简洁风格不太匹配。另外，在循环内修改数据子集，例如数据框子集，可能会先制作副本再修改，这当然会损失很多效率。R 3.1.0 版本以后列表元素在修改时不制作副本。

前面已经指出，利用 R 的向量化运算可以减少很多循环程序。

R 中的有些运算可以用内建函数完成，如 `sum`, `prod`, `cumsum`, `cumprod`, `mean`, `var`, `sd` 等。这些函数以编译程序的速度运行，不存在效率损失。

R 的 `sin`, `sqrt`, `log` 等函数都是向量化的，可以直接对输入向量的每个元素进行变换。

对矩阵，用 `apply` 函数汇总矩阵每行或每列。`colMeans`, `rowMeans` 可以计算矩阵列平均和行平均，`colSums`, `rowSums` 可以计算矩阵列和与行和。

`apply` 类函数有多个，包括 `apply`, `sapply`, `lapply`, `tapply`, `vapply`, `replicate` 等。这些函数不一定能提高程序运行速度，但是使用这些函数更符合 R 的程序设计风格，使程序变得简洁，当然，程序更简洁并不等同于程序更容易理解，要理解这样的程序，需要更多学习与实践。

18.3.1 `lapply()`、`sapply()` 和 `vapply()` 函数

对列表，`lapply` 函数操作列表每个元素，格式为

```
lapply(X, FUN)
```

其中 `X` 是一个列表或向量，`FUN` 是一个函数（可以是有名或无名函数），结果也总是一个列表，结果列表的第 i 个元素是将 `X` 的第 i 个元素输入到 `FUN` 中的返回结果。如果输入不是列表，就转换为列表再对每一元素做变换。

`sapply` 与 `lapply` 函数类似，但是 `sapply` 试图简化输出结果为向量或矩阵，在不可行时才和 `lapply` 返回列表结果。如果 `X` 长度为零，结果是长度为零的列表；如果 `FUN(X[i])` 都是长度为 1 的结果，`sapply()` 结果是一个向量；如果 `FUN(X[i])` 都是长度相同且长度大于 1 的向量，`sapply()` 结果是一个矩阵，矩阵的第 i 列保存 `FUN(X[i])` 的结果。因为 `sapply()` 的结果类型的不确定性，在自定义函数中应慎用。

`vapply()` 函数与 `sapply()` 函数类似，但是它需要第三个参数即函数返回值类型的例子，格式为

```
vapply(X, FUN, FUN.VALUE)
```

其中 `FUN.VALUE` 是每个 `FUN(X[i])` 的返回值的例子，要求所有 `FUN(X[i])` 结果类型和长度相同。

18.3.1.1 示例 1：数据框变量类型

`typeof()` 函数求变量的存储类型，如

```
d.class <- read.csv('class.csv', header=TRUE)
typeof(d.class[, 'age'])
```

```
## [1] "integer"
```

这里 `d.class` 是一个数据框，数据框也是一个列表，每个列表元素是数据框的一列。如下程序使用 `sapply()` 求每一列的存储类型：

```
sapply(d.class, typeof)
```

```
##      name      sex      age  height  weight
## "integer" "integer" "integer" "double" "double"
```

注意因为 CSV 文件读入为数据框时把姓名、性别都转换成了因子，所以这两个变量的存储类型也是整数。为了避免这样的转换，在 `read.csv()` 中使用选项 `stringsAsFactors=FALSE` 选项。

关于一个数据框的结构，用 `str()` 函数可以得到更为详细的信息：

```
str(d.class)
```

```
## 'data.frame':   19 obs. of  5 variables:
## $ name  : Factor w/ 19 levels "Alfred","Alice",...: 2 3 5 10 11 12 15 16 17 1 ...
## $ sex   : Factor w/ 2 levels "F","M": 1 1 1 1 1 1 1 1 1 2 ...
## $ age   : int   13 13 14 12 12 15 11 15 14 14 ...
## $ height: num   56.5 65.3 64.3 56.3 59.8 66.5 51.3 62.5 62.8 69 ...
## $ weight: num   84 98 90 77 84.5 ...
```

18.3.1.2 示例 2: `strsplit()` 函数结果处理

假设有 4 个学生的 3 次小测验成绩，每个学生的成绩记录到了一个以逗号分隔的字符串中，如：

```
s <- c('10, 8, 7',
      '5, 2, 2',
      '3, 7, 8',
      '8, 8, 9')
```

对单个学生，可以用 `strsplit()` 函数把三个成绩拆分，如：

```
strsplit(s[1], ',', fixed=TRUE)[[1]]
```

```
## [1] "10" " 8" " 7"
```

注意这里 `strsplit()` 的结果是仅有一个元素的列表，用了 “[[...]]” 格式取出列表元素。拆分的结果可以用 `as.numeric()` 转换为有三个元素的数值型向量：

```
as.numeric(strsplit(s[1], ',', fixed=TRUE)[[1]])
```

```
## [1] 10 8 7
```

还可以求三次小测验的总分：

```
sum(as.numeric(strsplit(s[1], ',', fixed=TRUE)[[1]]))
```

```
## [1] 25
```

用 `strsplit()` 处理有 4 个字符串的字符型向量 `s`, 结果是长度为 4 的列表：

```
tmp.res <- strsplit(s, ',', fixed=TRUE); tmp.res
```

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

用 `sapply()` 和 `as.numeric()` 可以把列表中所有字符型转为数值型，并以矩阵格式输出：

```
sapply(tmp.res, as.numeric)
```

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

但是，在通用程序中使用 `sapply()` 有可能会发生结果类型可变的情况。为此，上面可以用 `vapply()` 改写成

```
vapply(tmp.res, as.numeric, numeric(3))
```

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

其中第三个参数 `numeric(3)` 给出了对 `tmp.res` 中的任意一项应用 `as.numeric` 函数的结果的例子。

如果要求每个学生的小测验总分，只要对结果矩阵每列求和：


```
colSums(vapply(tmp.res, as.numeric, numeric(3)))
```

```
## [1] 25  9 18 25
```

这样的程序写法简洁，但是对于初学者需要比较长的时间来读懂，需要更长的时间用到自己的程序中。

18.3.1.3 示例 3: 不等长结果

调用 `sapply(列表, 函数)` 时, 如果“函数”结果长度有变化, 结果只能以列表输出。这时, `sapply` 与 `lapply` 返回相同的结果。一般地, `sapply` 试图把结果简化为向量、矩阵、多维数组, 在无法简化时就返回列表; `lapply` 总是返回列表。

设数据框 `d` 中有两列数, 希望将每列变成没有重复值的。数据例子如下:

```
d1 <- data.frame(x1=c(1,3,3,2), x2=c(3,5,5,3)); d1
```

```
##   x1 x2
## 1  1  3
## 2  3  5
## 3  3  5
## 4  2  3
```

因为 `x1` 和 `x2` 两列的无重复值个数不同, 结果只能是列表:

```
sapply(d1, unique)
```

```
## $x1
## [1] 1 3 2
##
## $x2
## [1] 3 5
```

与 `lapply(d, unique)` 效果相同。

如果使用了 `vapply`, 在遇到结果长度变化时会明确报错, 如:

```
vapply(d1, unique, numeric(3))
## Error in vapply(d1, unique, numeric(3)) : values must be length 3,
## but FUN(X[[2]]) result is length 2
```

在以上的例子中, 输入的 `d1` 数据框如果无重复个数相同, `sapply` 结果就是矩阵, 而 `lapply` 结果仍然是列表:

```
d2 <- data.frame(x1=c(1,3,3,2), x2=c(3,5,5,7)); d2
```

```
##   x1 x2
## 1  1  3
```

```
## 2 3 5
## 3 3 5
## 4 2 7
```

```
sapply(d2, unique)
```

```
##      x1 x2
## [1,]  1  3
## [2,]  3  5
## [3,]  2  7
```

```
lapply(d2, unique)
```

```
## $x1
## [1] 1 3 2
##
## $x2
## [1] 3 5 7
```

`sapply()` 的这种特点会造成编程判断困难，所以在不能确定函数结果长度是否保持不变时，应该用 `lapply()` 代替 `sapply()`，`lapply()` 总是返回列表。如果能够确定函数结果长度保持不变，在通用程序中应该用 `vapply()` 取代 `sapply()`，使得程序结果总是一致的。

18.3.1.4 示例 4：无名函数

`sapply`、`vapply` 和 `lapply` 中要做的函数变换可以当场定义，不需要函数名。

仍使用示例 2 的数据，任务是从输入的逗号分隔成绩中求每个学生的三科总分。

用 `strsplit()` 拆分可得列表，每个列表是由三个成绩字符串的字符型向量。如下代码可以求得总分：

```
s <- c('10, 8, 7',
      '5, 2, 2',
      '3, 7, 8',
      '8, 8, 9')
sapply( strsplit(s, ',', fixed=TRUE),
        function(ss) sum(as.numeric(ss)) )
```

```
## [1] 25  9 18 25
```

这里没有预先定义处理函数，也没有函数名，而是直接对 `sapply` 的第二自变量使用了一个无名函数。实际上，R 的函数定义也是函数名被赋值为一个函数对象。

18.3.2 replicate() 函数

`replicate()` 函数用来执行某段程序若干次，类似于 `for()` 循环但是没有计数变量。常用于随机模拟。`replicate()` 的缺省设置会把重复结果尽可能整齐排列成一个多维数组输出。

语法为

```
replicate(重复次数, 要重复的表达式)
```

其中的表达式可以是复合语句，也可以是执行一次模拟的函数。

下面举一个简单模拟例子。设总体 X 为 $N(0, 1)$ ，取样本量 $n = 5$ ，重复地生成模拟样本共 $B = 6$ 组，输出每组样本的样本均值和样本标准差。模拟可以用如下的 `replicate()` 实现：

```
set.seed(1)
replicate(6, {
  x <- rnorm(5, 0, 1);
  c(mean(x), sd(x)) })

##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
## [1,] 0.1292699 0.1351357 0.03812297 0.4595670 0.08123054 -0.3485770
## [2,] 0.9610394 0.6688342 1.49887443 0.4648177 1.20109623  0.7046822
```

结果是一个矩阵，矩阵行数与每次模拟的结果（均值、标准差）个数相同，这里第一行都是均值，第二行都是标准差；矩阵每一列对应于一次模拟。此结果转置可能更合适。

18.3.3 Map() 和 mapply()

`lapply()`、`sapply()`、`vapply()` 只能针对单个列表 X 的每个元素重复处理。如果有两个列表 X 和 Y 要进行对应元素的处理，用这三个函数不容易做到，这时可以用 `Map()` 或 `mapply()`。`Map()` 的格式为

```
Map(f, ...)
```

其中 f 是一个函数，`Map` 的其它参数都是每次取出对应的元素作为 $f()$ 的输入。`Map()` 的结果总是列表。

例如，下面有两个向量：

```
x <- c(1, 7, 2)
y <- c(3, 5, 9)
```

为了求得 x 和 y 的每个对应元素的最大值，可以用

```
Map(max, x, y)
```

```
## [[1]]
## [1] 3
##
## [[2]]
```

```
## [1] 7
##
## [[3]]
## [1] 9
```

结果是一个列表。为了把列表转换为普通向量，可以用 `unlist()` 函数，如

```
unlist(Map(max, x, y))
```

```
## [1] 3 7 9
```

这个例子演示了 `Map` 的用法。实际上，为了求多个向量对应元素的最大值，可以用 `pmax` 函数，如

```
pmax(x, y)
```

```
## [1] 3 7 9
```

`mapply()` 函数与 `Map()` 类似，但是可以自动简化结果类型，可以看成是 `sapply()` 推广到了可以对多个输入的对应元素逐项处理。`mapply()` 可以用参数 `MoreArgs` 指定逐项处理时一些共同的参数。简单的调用格式为

```
mapply(FUN, ...)
```

如

```
mapply(max, x, y)
```

```
## [1] 3 7 9
```

18.4 R 的计算函数

R 中提供了大量的数学函数、统计函数和特殊函数，可以打开 R 的 HTML 帮助页面，进入“Search Enging & Keywords”链接，查看其中与算数、数学、优化、线性代数等有关的专题。

这里简单列出一部分常用函数，对函数 `filter`, `fft`, `convolve` 进行说明。

18.4.1 数学函数

常用数学函数包括 `abs`, `sign`, `log`, `log10`, `sqrt`, `exp`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`, `sinh`, `cosh`, `tanh`。还有 `gamma`, `lgamma`(伽玛函数的自然对数)。

用于取整的函数有 `ceiling`, `floor`, `round`, `trunc`, `signif`, `as.integer` 等。这些函数是向量化的一元函数。

`choose(n,k)` 返回从 n 中取 k 的组合数。`factorial(x)` 返回 $x!$ 结果。`combn(x,m)` 返回从集合 x 中每次取出 m 个的所有不同取法，结果为一个矩阵，矩阵每列为一种取法的 m 个元素值。

18.4.2 概括函数

`sum` 对向量求和, `prod` 求乘积。

`cumsum` 和 `cumprod` 计算累计, 得到和输入等长的向量结果。

`diff` 计算前后两项的差分 (后一项减去前一项)。

`mean` 计算均值, `var` 计算样本方差或协方差矩阵, `sd` 计算样本标准差, `median` 计算中位数, `quantile` 计算样本分位数。 `cor` 计算相关系数。

`colSums`, `colMeans`, `rowSums`, `rowMeans` 对矩阵的每列或每行计算总和或者平均值, 效率比用 `apply` 函数要高。

`rle` 和 `inverse.rle` 用来计算数列中“连”长度及其逆向恢复, “连”经常用在统计学的随机性检验中。

18.4.3 最值

`max` 和 `min` 求最大和最小, `cummax` 和 `cummin` 累进计算。

`range` 返回最小值和最大值两个元素。

对于 `max`, `min`, `range`, 如果有多个自变量可以把这些自变量连接起来后计算。

`pmax(x1,x2,...)` 对若干个等长向量计算对应元素的最大值, 不等长时短的被重复使用。 `pmin` 类似。比如, `pmax(0, pmin(1,x))` 把 `x` 限制到 `[0,1]` 内。

18.4.4 排序

`sort` 返回排序结果。可以用 `decreasing=TRUE` 选项进行降序排序。 `sort` 可以有一个 `partial=` 选项, 这样只保证结果中 `partial=` 指定的下标位置是正确的。比如:

```
sort(c(3,1,4,2,5), partial=3)
```

```
## [1] 2 1 3 4 5
```

只保证结果的第三个元素正确。可以用来计算样本分位数估计。

在 `sort()` 中用选项 `na.last` 指定缺失值的处理, 取 `NA` 则删去缺失值, 取 `TRUE` 则把缺失值排在最后面, 取 `FALSE` 则把缺失值排在最前面。

`order` 返回排序用的下标序列, 它可以有多个自变量, 按这些自变量的字典序排序。可以用 `decreasing=TRUE` 选项进行降序排序。如果只有一个自变量, 可以使用 `sort.list` 函数。

`rank` 计算秩统计量, 可以用 `ties.method` 指定同名次处理方法, 如 `ties.method=min` 取最小秩。

`order`, `sort.list`, `rank` 也可以有 `na.last` 选项, 只能为 `TRUE` 或 `FALSE`。

`unique()` 返回去掉重复元素的结果, `duplicated()` 对每个元素用一个逻辑值表示是否与前面某个元素重复。如

```
unique(c(1,2,2,3,1))
```

```
## [1] 1 2 3
```

```
duplicated(c(1,2,2,3,1))
```

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

`rev` 反转序列。

18.4.5 一元定积分 `integrate`

`integrate(f, lower, upper)` 对一元函数 `f` 计算从 `lower` 到 `upper` 的定积分。使用自适应算法保证精度。如:

```
integrate(sin, 0, pi)
```

```
## 2 with absolute error < 2.2e-14
```

函数的返回值不仅仅包含定积分数值, 还包含精度等信息。

18.4.6 一元函数求根 `uniroot`

`uniroot(f, interval)` 对函数 `f` 在给定区间内求一个根, `interval` 为区间的两个端点。要求 `f` 在两个区间端点的值异号。即使有多个根也只能给出一个。如

```
uniroot(function(x) x*(x-1)*(x+1), c(-2, 2))
```

```
## $root
```

```
## [1] 0
```

```
##
```

```
## $f.root
```

```
## [1] 0
```

```
##
```

```
## $iter
```

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

```
##
```

```
## $init.it
```

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

```
##
```

```
## $estim.prec
```

```
## [1] 2
```

对于多项式，可以用 `polyroot` 函数求出所有的复根。

18.4.7 离散傅立叶变换 `fft`

R 中 `fft` 函数使用快速傅立叶变换算法计算离散傅立叶变换。设 x 为长度 n 的向量， $y=\text{fft}(x)$ ，则

```
y[k] = sum(x * complex(
  argument = -2*pi * (0:(n-1)) * (k-1)/n))
```

即

$$y_{k+1} = \sum_{j=0}^{n-1} x_{j+1} \exp\left(-i2\pi \frac{kj}{n}\right), \quad k = 0, 1, \dots, n-1.$$

注意没有除以 n ，结果是复数向量。

另外，若 $y=\text{fft}(x)$ ， $z=\text{fft}(y, \text{inverse}=T)$ ，则 $x == z/\text{length}(x)$ 。

快速傅立叶变换是数值计算中十分常用的工具，R 软件包 `fftw` 可以进行优化的快速傅立叶变换。

18.4.8 用 `filter` 函数作迭代

R 在遇到向量自身迭代时很难用向量化编程解决，`filter` 函数可以解决其中部分问题。`filter` 函数可以进行卷积型或自回归型的迭代。语法为

```
filter(x, filter,
  method = c("convolution", "recursive"),
  sides=2, circular =FALSE, init)
```

下面用例子演示此函数的用途。

18.4.8.1 示例 1：双侧滤波

对输入序列 $x_t, t = 1, 2, \dots, n$ ，希望进行如下滤波计算：

$$y_t = \sum_{j=-k}^k a_j x_{t-j}, \quad k+1 \leq t \leq n-k-1,$$

其中 $(a_{-k}, \dots, a_0, \dots, a_k)$ 是长度为 $2k+1$ 的向量。注意公式中 a_j 与 x_{t-j} 对应。

假设 x 保存在向量 x 中， $(a_{-k}, \dots, a_0, \dots, a_k)$ 保存在向量 f 中， y_{k+1}, \dots, y_{n-k} 保存在向量 y 中，无定义部分取 NA，程序可以写成

```
y <- filter(x, f, method="convolution", sides=2)
```

比如，设 $x = (1, 3, 7, 12, 17, 23)$ ， $(a_{-1}, a_0, a_1) = (0.1, 0.5, 0.4)$ ，则

$$y_t = 0.1 \times x_{t+1} + 0.5 \times x_t + 0.4 \times x_{t-1}, \quad t = 2, 3, \dots, 5$$

用 `filter()` 函数计算, 程序为:

```
y <- filter(c(1,3,7,12,17,23), c(0.1, 0.5, 0.4),
            method="convolution", sides=2)

y
## Time Series:
## Start = 1
## End = 6
## Frequency = 1
## [1] NA 2.6 5.9 10.5 15.6 NA
```

18.4.8.2 示例 2: 单侧滤波

对输入序列 $x_t, t = 1, 2, \dots, n$, 希望进行如下滤波计算:

$$y_t = \sum_{j=0}^k a_j x_{t-j}, \quad k+1 \leq t \leq n,$$

其中 (a_0, \dots, a_k) 是长度为 $k+1$ 的向量。注意公式中 a_j 与 x_{t-j} 对应。

假设 x 保存在向量 `x` 中, (a_0, \dots, a_k) 保存在向量 `f` 中, y_{k+1}, \dots, y_n 保存在向量 `y` 中, 无定义部分取 NA, 程序可以写成

```
y <- filter(x, f, method="convolution", sides=1)
```

比如, 设 $x = (1, 3, 7, 12, 17, 23)$, $(a_0, a_1, a_2) = (0.1, 0.5, 0.4)$, 则

$$y_t = 0.1 \times x_t + 0.5 \times x_{t-1} + 0.4 \times x_{t-2}, \quad t = 3, 4, \dots, 6$$

程序为

```
y <- filter(c(1,3,7,12,17,23), c(0.1, 0.5, 0.4),
            method="convolution", sides=1)

y
## Time Series:
## Start = 1
## End = 6
## Frequency = 1
## [1] NA NA 2.6 5.9 10.5 15.6
```

18.4.8.3 示例 3: 自回归迭代

设输入 $e_t, t = 1, 2, \dots, n$, 要计算

$$y_t = \sum_{j=1}^k a_j y_{t-j} + e_t, \quad t = 1, 2, \dots, n,$$

其中 (a_1, \dots, a_k) 是 k 个实数, (y_{-k+1}, \dots, y_0) 已知。

设 x 保存在向量 \mathbf{x} 中, (a_1, \dots, a_k) 保存在向量 \mathbf{a} 中, (y_1, \dots, y_n) 保存在向量 \mathbf{y} 中。

如果 (y_{-k+1}, \dots, y_0) 都等于零, 可以用如下程序计算 y_1, y_2, \dots, y_n :

```
filter(x, a, method="recursive")
```

如果 (y_0, \dots, y_{-k+1}) 保存在向量 \mathbf{b} 中 (注意与时间顺序相反), 可以用如下程序计算 y_1, y_2, \dots, y_n :

```
filter(x, a, method="recursive", init=b)
```

比如, 设 $e = (0.1, -0.2, -0.1, 0.2, 0.3, -0.2)$, $(a_1, a_2) = (0.9, 0.1)$, $y_{-1} = y_0 = 0$, 则

$$y_t = 0.9 \times y_{t-1} + 0.1 \times y_{t-2} + e_t, \quad t = 1, 2, \dots, 6$$

迭代程序和结果为

```
y <- filter(c(0.1, -0.2, -0.1, 0.2, 0.3, -0.2),
            c(0.9, 0.1), method="recursive")
print(y, digits=3)
## Time Series:
## Start = 1
## End = 6
## Frequency = 1
## [1] 0.1000 -0.1100 -0.1890 0.0189 0.2981 0.0702
```

这个例子中, 如果已知 $y_0 = 200, y_{-1} = 100$, 迭代程序和结果为:

```
y <- filter(c(0.1, -0.2, -0.1, 0.2, 0.3, -0.2),
            c(0.9, 0.1), init=c(200, 100),
            method="recursive")
print(y, digits=6)
## Time Series:
## Start = 1
## End = 6
## Frequency = 1
## [1] 190.100 190.890 190.711 190.929 191.207 190.979
```

18.5 并行计算

现代桌面电脑和笔记本电脑的 CPU 通常有多个核心或虚拟核心 (线程), 如 2 核心或 4 虚拟核心。通常 R 运行并不能利用全部的 CPU 能力, 仅能利用其中的一个虚拟核心。使用特制的 BLAS 库 (非 R 原有) 可以并发置信多个线程, 一些 R 扩展包也可以利用多个线程。利用多台计算机、多个 CPU、CPU 中的多核心和多线程同时完成一个计算任务称为并行计算。

想要充分利用多个电脑、多个 CPU 和 CPU 内的虚拟核心，技术上比较复杂，涉及到计算机之间与进程之间的通讯问题，在要交流的数据量比较大时会造成并行计算的瓶颈。

实际上，有些问题可以很容易地进行简单地并行计算。比如，在一个统计研究中，需要对 100 组参数组合进行模拟，评估不同参数组合下模型的性能。假设研究人员有两台安装了 R 软件的计算机，就可以在两台计算机上进行各自 50 组参数组合的模拟，最后汇总在一起就可以了。

R 的 `parallel` 包提供了一种比较简单的利用 CPU 多核心的功能，思路与上面的例子类似，如果有多个任务互相之间没有互相依赖，就可以分解到多个计算机、多个 CPU、多个虚拟核心中并行计算。最简单的情形是一台具有单个 CPU、多个虚拟核心的台式电脑或者笔记本电脑。但是，统计计算中最常见耗时计算任务是随机模拟，随机模拟要设法避免不同进程的随机数序列的重复可能，以及同一进程中不同线程的随机数序列的重复可能。

`parallel` 包提供了 `parLapply()`、`parSapply()`、`parApply()` 函数，作为 `lapply()`、`sapply()`、`apply()` 函数的并行版本，与非并行版本相比，需要用一个临时集群对象作为第一自变量。

18.5.1 例 1：完全不互相依赖的并行运算

考虑如下计算问题：

$$S_{k,n} = \sum_{i=1}^n \frac{1}{i^k}$$

下面的程序取 n 为一百万， k 为 2 到 21，循环地用单线程计算。

```
f10 <- function(k=2, n=1000){
  s <- 0.0
  for(i in seq(n)) s <- s + 1/i^k
  s
}
f11 <- function(n=1000000){
  nk <- 20
  v <- sapply(2:(nk+1), function(k) f10(k, n))
  v
}
system.time(f11())[3]
## elapsed
##      2.87
```

因为对不同的 k ， $f0(k)$ 计算互相不依赖，也不涉及到随机数序列，所以可以简单地并行计算而没有任何风险。先查看本计算机的虚拟核心（线程）数：

```
library(parallel)
detectCores()
## [1] 8
```

用 `makeCluster()` 建立临时的有 8 个节点的单机集群:

```
cpuc1 <- makeCluster(8)
```

用 `parSapply()` 或者 `parLapply()` 关于 k 并行地循环:

```
f12 <- function(n=1000000){
  f10 <- function(k=2, n=1000){
    s <- 0.0
    for(i in seq(n)) s <- s + 1/i^k
    s
  }

  nk <- 20
  v <- parSapply(cpuc1, 2:(nk+1), function(k) f10(k, n))
  v
}
system.time(f12())[3]
## elapsed
##      1.19
```

并行版本速度提高了 140% 左右。

并行执行结束后, 需要解散临时的集群, 否则可能会有内存泄漏:

```
stopCluster(cpuc1)
```

注意并行版本的程序还需要一些在每个计算节点上的初始化, 比如调入扩展包, 定义函数, 初始化不同的随机数序列等。parallel 包的并行执行用的是不同的进程, 所以传送给每个节点的计算函数要包括所有的依赖内容。比如, `f2()` 中内嵌了 `f0()` 的定义, 如果不将 `f0()` 定义在 `f2()` 内部, 就需要预先将 `f0()` 的定义传递给每个节点。

parallel 包的 `clusterExport()` 函数可以用来把计算所依赖的对象预先传送到每个节点。比如, 上面的 `f2()` 可以不包含 `f0()` 的定义, 而是用 `clusterExport()` 预先传递:

```
cpuc1 <- makeCluster(8)
clusterExport(cpuc1, c("f10"))
f13 <- function(n=1000000){
  nk <- 20
  v <- parSapply(cpuc1, 2:(nk+1), function(k) f10(k, n))
  v
}
system.time(f13())[3]
## elapsed
##      1.08
```

```
stopCluster(cpuc1)
```

如果需要在每个节点预先执行一些语句，可以用 `clusterEvalQ()` 函数执行，如

```
clusterEvalQ(cpuc1, library(dplyr))
```

18.5.2 例 2：使用相同随机数序列的并行计算

为了估计总体中某个比例 p 的置信区间，调查了一组样本，在 n 个受访者中选“是”的比例为 \hat{p} 。令 λ 为标准正态分布的双侧 α 分位数，参数 p 的近似 $1 - \alpha$ 置信区间为

$$\frac{\hat{p} + \frac{\lambda^2}{2n}}{1 + \frac{\lambda^2}{n}} \pm \frac{\lambda}{\sqrt{n}} \frac{\sqrt{\hat{p}(1 - \hat{p}) + \frac{\lambda^2}{4n}}}{1 + \frac{\lambda^2}{n}}$$

称为 Wilson 置信区间。

假设要估计不同 $1 - \alpha, n, p$ 情况下，置信区间的覆盖率（即置信区间包含真实参数 p 的概率）。可以将这些参数组合定义成一个列表，列表中每一项是一种参数组合，对每一组合分别进行随机模拟，估计覆盖率。因为不同参数组合之间没有互相依赖的关系，随机数序列完全可以使用同一个序列。

不并行计算的程序示例：

```
wilson <- function(n, x, conf){
  hatp <- x/n
  lam <- qnorm((conf+1)/2)
  lam2 <- lam^2 / n
  p1 <- (hatp + lam2/2)/(1 + lam2)
  delta <- lam / sqrt(n) * sqrt(hatp*(1-hatp) + lam2/4) / (1 + lam2)
  c(p1-delta, p1+delta)
}

f20 <- function(cpar){
  set.seed(101)
  conf <- cpar[1]
  n <- cpar[2]
  p0 <- cpar[3]
  nsim <- 100000
  cover <- 0
  for(i in seq(nsim)){
    x <- rbinom(1, n, p0)
    cf <- wilson(n, x, conf)
    if(p0 >= cf[1] && p0 <= cf[2]) cover <- cover+1
  }
  cover/nsim
}
```

```

}
f21 <- function(){
  dp <- rbind(rep(c(0.8, 0.9), each=4),
              rep(rep(c(30, 100), each=2), 2),
              rep(c(0.5, 0.1), 4))
  lp <- as.list(as.data.frame(dp))
  res <- sapply(lp, f20)
  res
}
system.time(f21())[3]

```

约运行 4.1 秒。

改为并行版本：

```

cpuc1 <- makeCluster(4)
clusterExport(cpuc1, c("f20", "wilson"))
f22 <- function(){
  dp <- rbind(rep(c(0.8, 0.9), each=4),
              rep(rep(c(30, 100), each=2), 2),
              rep(c(0.5, 0.1), 4))
  lp <- as.list(as.data.frame(dp))
  res <- parSapply(cpuc1, lp, f20)
  res
}
system.time(f22())[3]
stopCluster(cpuc1)

```

运行约 1.8 秒，速度提高 130% 左右。这里模拟了 8 种参数组合，每种参数组合模拟了十万次，每种参数组合模拟所用的随机数序列是相同的。

18.5.3 例 3：使用独立随机数序列的并行计算

大量的耗时的统计计算是随机模拟，有时需要并行计算的部分必须使用独立的随机数序列。比如，需要进行一千次重复模拟，每次使用不同的随机数序列，可以将其分解为 10 组模拟，每组模拟一百万次，这就要求这 10 组模拟使用的随机数序列不重复。

R 中实现了 L'Ecuyer 的多步迭代复合随机数发生器，此随机数发生器周期很长，而且很容易将发生器的状态前进指定的步数。parallel 包的 `nextRNGStream()` 函数可以将该发生器前进到下一段的开始，每一段都足够长，可以用于一个节点。

以 Wilson 置信区间的模拟为例。设 $n = 30$, $p = 0.01$, $1 - \alpha = 0.95$ ，取重复模拟次数为 1 千万次，估计

Wilson 置信区间的覆盖率。单线程版本为:

```
f31 <- function(){
  set.seed(101)
  n <- 30; p0 <- 0.01; conf <- 0.95
  nsim <- 1E7
  cover <- 0
  for(i in seq(nsim)){
    x <- rbinom(1, n, p0)
    cf <- wilson(n, x, conf)
    if(p0 >= cf[1] && p0 <= cf[2]) cover <- cover+1
  }
  cover/nsim
}
system.time(cvg1 <- f31())[3]
## elapsed
## 50.17
```

单线程版本运行了大约 50 秒。

改成并行版本。比例 2 多出的部分是为每个节点分别计算一个随机数种子将不同的种子传给不同节点。parallel 包的 `clusterApply()` 函数为临时集群的每个节点分别执行同一函数，但对每个节点分别使用列表的不同元素作为函数的自变量。

```
cpuc1 <- makeCluster(8)
each.seed <- function(s){
  assign(".Random.seed", s, envir = .GlobalEnv)
}
RNGkind("L'Ecuyer-CMRG")
set.seed(101)
seed0 <- .Random.seed
seeds <- as.list(1:4)
for(i in 1:4){ # 给每个节点制作不同的种子
  seed0 <- nextRNGStream(seed0)
  seeds[[i]] <- seed0
}
## 给每个节点传送不同种子:
junk <- clusterApply(cpuc1, seeds, each.seed)

f32 <- function(isim){
  n <- 30; p0 <- 0.01; conf <- 0.95
  nsim <- 1E6
```

```

cover <- 0
for(i in seq(nsim)){
  x <- rbinom(1, n, p0)
  cf <- wilson(n, x, conf)
  if(p0 >= cf[1] && p0 <= cf[2]) cover <- cover+1
}
cover
}

clusterExport(cpucl, c("f32", "wilson"))

f33 <- function(){
  nbatch <- 10; nsim <- 1E6
  cvs <- parSapply(cpucl, 1:nbatch, f32)
  print(cvs)
  sum(cvs)/(nsim*nbatch)
}

system.time(cvg2 <- f33())[3]
## [1] 963759 963660 963885 963739 963714 964171 963615 963822 963720 963939
## elapsed
## 18.76
stopCluster(cpucl)

```

并行版本运行了大约 20 秒，速度提高 160%。从两个版本各自一千万次重复模拟结果来看，用随机模拟方法得到的覆盖率估计的精度大约为 3 位有效数字。

更大规模的随机模拟问题，可以考虑使用多 CPU 的计算工作站或者服务器，或用多台计算机通过高速局域网组成并行计算集群。

还有一种选择是租用云计算服务。

Part IV

用 R 制作研究报告和图书

Chapter 19

用 R 制作研究报告

一个统计或数据分析的科研项目，最后都有一个研究报告。因为使用统计与数据分析不可避免地有很多计算涉及在内，这里假设使用 R 软件做了计算。科研是一个不断改进的过程，所以每一次重新做了计算，研究报告中的汇总表格、图形都要更新。这样的任务比较繁琐，也容易出错。

“文学式编程”(literate programming) 是这样一种思想，把撰写报告与计算程序有机地结合在一起，用一个源文件文件既包含报告内容，又包含计算程序。每次产生研究报告时，先运行源文件中的计算程序得到计算结果，这些结果包括文字性内容与图形，然后利用适当软件自动地把这些原始文字、计算结果组合成最终的报告。利用这样的思想，可以自动生成重复的例行报告，还可以作为“可重复科学研究”的载体。

上述的源文件一般是文本文件，格式可以是 Markdown 格式，也可以是 LaTeX 格式等许多格式。R 的 knitr 软件包就是用来支持这样的编程思想的一个扩展包。

Markdown 是一种很简单的文本文件格式，通常保存为.md 扩展名。Markdown 文件里面有一些简单的格式标注方法，比如两个星号之间的文字会转化为斜体，缩进四个空格或一个制表符的内容会看成代码。Markdown 仅适用于比较简单的文章、源程序说明等，不太适用于复杂的含有大量数学公式、图标文章，从 markdown 格式比较适合转换为 html(网页) 格式，也可以转换为 MS Word 的 docx 格式，通过 docx 格式可以转存为 PDF 格式。

LaTeX 是一个文档排版系统，功能强大，结果美观，设计合理。缺点是需要学习类似于 HTML 的另一种语言。LaTeX 源文件主要是编译为 PDF 和 PostScript，也可以编译为 HTML。

R 扩展包 knitr 包支持在 Markdown 格式、LaTeX 格式等类型的文件中插入 R 代码，经过转换 R 代码可以变成代码的结果文字、表格、图形与原有报告文字有机地结合在一起。

插入了 R 代码的文件一般以 .Rmd 为扩展名，R 的 rmarkdown 扩展包和 knitr 包一起为 Rmd 格式提供了支持。

R 的 bookdown 包进一步增强了 R Markdown 格式的功能，支持生成 PDF、多文件互相链接的 HTML、Word 等输出，其中的表格、图形可以变成浮动表，公式、定理可以自动编号并支持公式、定理、图表、章节的引用和链接，所以比较适用于编写一本书或论文、研究报告。

knitr 包也支持在 LaTeX 源文件中插入 R 代码，结果使得 R 代码运行结果、图形自动插入生成的研究报告中。原来有一个 Sweave 扩展包是支持在 LaTeX 中插入 R 代码的，现在 knitr 的功能已经涵盖并增强了此包功能。

下面几章分别介绍 markdown 格式、R markdown 格式、bookdown 扩展包、简易网站制作和幻灯片制作。

Chapter 20

Markdown 格式

20.1 介绍

Markdown 是一种很简单的文本文件格式，通常保存为.md 扩展名。Markdown 中文内容应该使用 UTF-8 编码。Markdown 文件里面有一些简单的格式标注方法，比如两个星号之间的文字会转化为斜体，缩进四个空格或一个制表符的内容会看成代码。

Markdown 适用于比较简单的文章、源程序说明等，不太适用于复杂的含有大量数学公式、图表的文章，从 markdown 格式比较适合转换为 html(网页) 格式，也可以转换为 MS Word 的 docx 格式，通过 docx 格式可以转存为 PDF 格式。R 扩展包 knitr、rmarkdown 和 bookdown 与 pandoc 软件一起大大扩展了 markdown 格式的适用范围。

如果需要作为一本书发布在网站上或者出版，可考虑使用 R 的 bookdown 扩展包。如果需要对出版格式进行精确控制，可考虑用 LaTeX 格式，LaTeX 格式很复杂，学习比较困难，但是表达能力强。

20.2 Markdown 格式文件的应用

Markdown 格式用在一些微博、论坛中作为缺省格式，用户在网络浏览器软件的输入框中按照 markdown 格式输入，网站自动将其转换为 html 富文本内容显示出来。

因为 markdown 格式就是纯文本，而且其格式十分简单，所以可以仅仅用普通文本编辑器编写 markdown 格式的文件，不需要转换为其它格式。

有一些独立运行的编辑软件，在其中输入了 markdown 格式文件后，可以并列地显示转化为 html 富文本后的结果，很多还支持自动备份到云服务中。但是，这些独立运行的编辑器大都有商业因素。

RStudio 软件不是专用的 Markdown 编辑器，它是一个 R 程序的集成编辑、运行环境，对个人用户免费，在 R Studio 中可以编辑 Markdown 文件和含有 R 代码的 Markdown 文件，可以一键将其转化成 HTML、MS Word docx 文件，在单独安装的 LaTeX 编译软件支持下还可以直接编译成 PDF。RStudio 支持下的增

强的 Markdown 格式, 称为 RMarkdown 格式, 以 .Rmd 为扩展名, 支持大多数的 LaTeX 公式, 在 bookdown 扩展包支持下还支持公式、定理、图表等自动编号和引用、链接。

20.3 pandoc 用法

pandoc 是一个开源软件, 在命令行运行, 可以用来在多种不同的文件格式之间进行转换, 输入格式一般是文本格式, 比如 markdown、LaTeX、MediaWiki、reStructured Text、HTML、DocBook, 但是也可以输入 MS Word docx、Open Office ODT、EPUB 格式。输出可以是这些文本格式, 也可以是 MS Word docx、Open Office ODT、EPUB、LaTeX 等格式, 在安装了 LaTeX 编译系统如 MikTeX 时可以输出为 PDF。RStudio 软件中包括了一份 pandoc 软件程序。

可以用普通文本编辑器编写 markdown 格式的文件, 用 pandoc 转换为 HTML 或 MS Word docx 等格式。因为 pandoc 需要 UTF8 编码的输入文件, 所以应该把 markdown 文件保存为 UTF 格式, MS Windows 下的 Notepad++ 软件可以很容易地编辑文本文件并在各种编码之间转换。实际上, MS Windows 下有一个 markdown 编辑程序 Smarks 就是基于 pandoc 软件的。

首先, 从 pandoc 网站下载并安装 pandoc。安装程序很奇怪地安装到了 C:\Users\登录用户名\AppData\Local\Pandoc 中, 请将此子目录复制到一个合适的位置, 比如 C:\Pandoc。如果把此路径加入到 Windows 系统的可执行文件搜索路径中 (在“控制面板-系统和安全-系统-高级系统设置-环境变量”中, 为系统变量的 Path 添加一个分号分开的路径 C:\Pandoc 即可以不用全路径访问 pandoc.exe 可执行文件。

为了用 pandoc 转化某个 markdown 文件, 首先在该文件所在子目录打开一个 Windows 命令行窗口, 在 MS Win10 系统中只要在文件管理器的“文件”菜单选择“打开命令提示符”即可。比如, 文件名是 test.md, 用如下 pandoc 命令可以转换为.docx 文件 (MS Word 文件的新版本):

```
C:\Pandoc\pandoc -o test.docx test.md
```

用如下 pandoc 命令可以转换为.html 文件:

```
C:\Pandoc\pandoc -s -o --mathjax test.html test.md
```

如果把 pandoc.exe 加入了 Windows 操作系统的 Path 环境变量中, 上面的 C:\Pandoc\pandoc 可以简写为 pandoc。

pandoc 在其它操作系统中也有相应的版本。软件下载与文档见 Pandoc 的网站<http://pandoc.org>。

如果使用 RStudio 编辑 markdown 文件或者 R Markdown 文件, 它会自动调用内置的 pandoc 程序。

20.4 markdown 格式说明

这部分内容参考了 markdown 手册和 Rstudio 的文档。

20.4.1 概述

Markdown 格式是 John Gruber 于 2004 年创造的，Markdown 的目标是实现“易读易写”。Markdown 定义了一种简单好用的文本文件格式，作为单独的文本文件，此格式没有什么多余的标签，又可以转化为很多其它的格式。

Markdown 的语法全由一些符号所组成，这些符号经过精挑细选，其作用一目了然。比如：在文字两旁加上星号，看起来就像强调。Markdown 的列表看起来就像我们平常在邮件中写一个列表的方法。Markdown 的区块引用看起来就真的像是引用一段文字，就像你曾在电子邮件中见过的那样。

需要时，可以直接在 markdown 中写 HTML 标记内容。markdown 能实现的功能是 HTML 的一部分，但是比 HTML 内容更干净，没有掺杂过多的与要表达的意思无关的标签。Markdown 的理念是，能让文档更容易读、写和随意改。

20.4.2 段落

一个段落由一行或连续的多行组成。段落之间以空行分隔。同一段落内的不同行在转换成 HTML 或 docx 等格式后会重新排列，原来的段内换行被当成了空格，这样的规定与 LaTeX 类似。普通段落不该用空格或制表符来缩进。

为了在段内换行并且转化后仍保持段内换行，输入时在前面行的末尾输入两个或两个以上空格。例如：

白日依山尽，黄河入海流。
欲穷千里目，更上一层楼。

显示结果为：

白 日 依 山 尽， 黄 河 入 海 流。
欲穷千里目，更上一层楼。

这样做的缺点是末尾的空格时不可见的。可以使用 HTML 的 `
` 标签在段内换行，如：

白日依山尽，黄河入海流。
欲穷千里目，更上一层楼。

结果为

白日依山尽，黄河入海流。欲穷千里目，更上一层楼。

为了得到多行的人为控制换行的结果，可以使用 Markdown 引用或者列表。

20.4.3 段内文字格式

在一段内，用星号或下划线包围的内容是强调格式。用双星号或双下划线包围的内容是加重格式。星号、下划线与要强调或加重的内容之间不要空开，否则会当作普通星号或下划线解释，在行首还会当作列表。为了插入普通的星号或下划线，可以使用反斜杠保护，或者写成段内代码格式。

在普通段落内一部分内容希望显示成代码，对其中的特殊字符不进行解释，只要包在两个反向单撇号内。如 `'if(_x_ > 0) y=1;'`会变成 `if(_x_ > 0) y=1;`。

在 Markdown 文件中，为了使得某些有特殊意义的字符不作特殊解释，可以在该字符前面加上反斜杠\，取消其特殊含义。

20.4.4 标题和分隔线

以一个井号 # 开始的行是一级标题，以两个井号 # 开始的行是二级标题，.....，以六个井号 # 开始的行是六级标题。标题行前面应该空一行，否则可能把某些偶然出现在行首的 # 号误认为标题行的标志。

对一级标题，也可以用标题内容下面输入一行等于号 = 表示上一行内容是一级标题。对二级标题，可以用标题内容下面输入一行减号-表示上一行内容是二级标题。等于号和减号的个数不限。

用三个或三个以上连续的星号组成的行，可以转换成分隔线。下面是一个分隔线：

20.4.5 引用段落

可以用类似 Email 的回复包含原始邮件内容的办法输入引用段落，即，在段落的每行前面加一个大于号。比如下面的诗：

> 白日依山尽，黄河入海流。
> 欲穷千里目，更上一层楼。

转换成

白日依山尽，黄河入海流。欲穷千里目，更上一层楼。

注意引用也是段落模式，内容中的换行不起作用，空行导致分段。

引用段落也可以仅在段落第一行写大于号，其它行顶格写，例如下面的两段引用：

> 远上寒山石径斜，
白云生处有人家。
>
> 停车坐爱枫林晚，
霜叶红于二月花。

转换成

远上寒山石径斜，白云生处有人家。

停车坐爱枫林晚，霜叶红于二月花。

引用也可以嵌套，如：


```
> 张三说：李四这样说过
>
>> 不想当将军的木匠不是好厨子。
>
```

转换成

```
    张三说：李四这样说过
    不想当将军的木匠不是好厨子。
```

注意嵌套内容前后都有空的引用行，否则不能实现嵌套引用。

引用内也可以嵌套其它的 Markdown 格式如标题、列表等。引用前后应该有空行把引用内容与其他内容分隔开。

20.4.6 列表

列表分为不编号的列表和编号的列表。不编号的列表转化后通常显示圆点开头的列表项。

在 Markdown 中，用星号表示一个不编号的列表项。星号也可以替换成加号或减号，后面必须有一个或多个空格。每个列表项可以输入多行，各行的内容最好对齐，这样仅使用文本格式时较易阅读，但不是必须的。两个列表项之间不要空行。例如：

```
* 白日依山尽，
    黄河入海流。
* 欲穷千里目，
    更上一层楼。
```

转换为

- 白日依山尽，黄河入海流。
- 欲穷千里目，更上一层楼。

段落顶头的数字加句点和空格表示编号列表，两个列表项之间尽量不要空行。例如：

```
1. 第一种解决方法，
    收买敌人的高官。
2. 第二种解决方法，
    尽可能拖延。
```

转换为

1. 第一种解决方法，收买敌人的高官。
2. 第二种解决方法，尽可能拖延。

标准的 markdown 编号列表不能自己定义数字的显示格式，不允许开始值不等于 1。pandoc 支持更自由的列表，允许输入时有括号或右括号，允许使用字母和罗马数字，但是括号会被去掉。如

- (1) 顶层一；
 - a) 内层二；
 - b) 内层三；
- (2) 顶层二。

转换为

- (1) 顶层一；
 - a) 内层二；
 - b) 内层三；
- (2) 顶层二。

为了避免错误地产生非本意的编号列表，在行首写数字加句点和空格时，可以在句点前加反斜杠，或者在句点前加空格。例如，下面是一个年号：

2016\.

如上输入将不会错误地解释为有序列表。

如果列表项目中有多个段落，这时两个列表项之间应该以空行分隔，每个项目除了第一行外，输入的每行内容都应该缩进 4 个空格或者一个制表符。例如：

- * R语言第一个版本开发于1976-1980，基于Fortran；
 于1980年移植到Unix，并对外发布源代码。
 1984年出版的“棕皮书”
 总结了1984年为止的版本，并开始发布授权的源代码。
 这个版本叫做旧S。与我们现在用的S语言有较大差别。

1989--1988对S进行了较大更新，
 变成了我们现在使用的S语言，称为第二版。
 1988年出版的“蓝皮书”做了总结。

- * 1992年出版的“白皮书”描述了在S语言中实现的统计建模功能，
 增强了面向对象的特性。软件称为第三版，这是我们现在用的多数版本。

1998年出版的“绿皮书”描述了第四版S语言，主要是编程功能的深层次改进。
 现行的S系统并没有都采用第四版，S-PLUS的第5版才采用了S语言第四版。

转换为

- R语言第一个版本开发于 1976-1980，基于 Fortran；于 1980 年移植到 Unix, 并对外发布源代码。
 1984 年出版的“棕皮书”总结了 1984 年为止的版本，并开始发布授权的源代码。这个版本叫做旧 S。
 与我们现在用的 S 语言有较大差别。
- 1989-1988 对 S 进行了较大更新，变成了我们现在使用的 S 语言，称为第二版。1988 年出版的“蓝皮书”做了总结。

- 1992 年出版的“白皮书”描述了在 S 语言中实现的统计建模功能，增强了面向对象的特性。软件称为第三版，这是我们现在用的多数版本。

1998 年出版的“绿皮书”描述了第四版 S 语言，主要是编程功能的深层次改进。现行的 S 系统并没有都采用第四版，S-PLUS 的第 5 版才采用了 S 语言第四版。

列表项目内如果有引用段落，需要都缩进 4 个空格。如果有程序代码，需要缩进 4 个空格后用三个反单撇号表示开始与结束。

列表可以嵌套，嵌套的列表需要缩进 4 个空格，中间不需要空行。例如：

1. 第一类工作包括：
 - + 技术服务；
 - + 咨询服务。
2. 其它工作略。

转换为

1. 第一类工作包括：
 - 技术服务；
 - 咨询服务。
2. 其它工作略。

如果需要把每个列表项当作段落排版，可以在每个列表项后空行。

20.4.7 源程序

为了让源程序能够自动显示成源程序的样式，而不至于自动分行、特殊字符解释，用空行把源程序与其它内容隔开，并把源程序行都缩进 4 个空格（或以上）或者一个制表符。源程序格式持续到不缩进 4 个空格的地方为止。

例如，下面的输入：

```
f <- function(x){
  n <- length(x)
  y <- numeric(n)
  y[x >= 0] <- 1
  ##y[x < 0] <- 0

  y
}
```

转换为

```
f <- function(x){
  n <- length(x)
```

```

    y <- numeric(n)
    y[x >= 0] <- 1
    ##y[x < 0] <- 0

    y
}

```

更适当的做法是用三个连续的反向单撇号表示代码开头与代码结束，中间就会当作源程序代码处理。例如下面的输入

```

...
> x <- rnorm(100)
> hist(x)
...

```

转换为

```

> x <- rnorm(100)
> hist(x)

```

R 的 knitr 包在 Markdown 格式的文件中插入 R 可执行代码时，就用了这样的方法。而且，R Markdown 格式的代码块不需要用空行与前后分隔开。

在 pandoc 程序的支持下，代码段还可以采用栅栏式代码段，在代码段开头前面一行加上至少三个连续 ~ 符号，在结尾后面一行加同样数目的 ~ 符号。这样的代码段前后也必须空行以与其它内容分开。另外，如果代码内本身含有 ~ 行，只要使得开头与结尾标志中的 ~ 个数更多就可以了。例如下面的输入

```

~~~
#include <math.h>
double sqr(double x){
    return(x*x);
}
~~~

```

转换为

```

#include <math.h>
double sqr(double x){
    return(x*x);
}

```

使用栅栏式代码段时可以在开始行尾写大括号，在大括号内写选项。其中一种选项是要求安装某种编程语言对结果进行彩色语法显示，如 .cpp 表示 C++，.c 表示 C，.r 表示 R，.python 表示 python 等。选项 .numberLines 要求该代码行编号，选项 startFrom= 指定开始行号。如如：

```

~~~{.cpp .numberLines startFrom=101}

```

```
#include <math.h>
double sqr(double x){
    return(x*x);
}
~~~
```

转换为

```
101 #include <math.h>
102 double sqr(double x){
103     return(x*x);
104 }
```

20.4.8 链接

最简单的链接是原样显示的可点击的链接，只要把链接地址用小于号和大于号包在中间，两边用空格和其它内容隔开。如果是网页，需要加 `http://`，如果是邮箱，需要加 `mailto:`。例如，如下代码：

北京大学的网页地址是： `<http://www.pku.edu.cn/>` 。

显示为

北京大学的网页地址是： `http://www.pku.edu.cn/` 。

除此之外，Markdown 还支持两种形式的链接语法：行内式和引用式两种形式。不管是哪一种，链接的显示文字都是用方括号 [...] 来标记。

要建立一个行内式的链接，只要在方括号内写链接的显示文字，右方括号后面紧接着圆括号，并在圆括号中间插入网址链接即可。方括号部分与圆括号部分之间不能断开。例如：

请参考：[李东风的教学主页](`http://www.math.pku.edu.cn/teachers/lidf/course/index.htm`)

变成了

请参考：李东风的教学主页

在圆括号中，链接后面还可以包含用双撇号包围的标题文字，与链接之间用空格分开。

引用式的链接，需要在某处（比如文章结尾）定义一些链接的标识符，然后用方括号包围链接的显示文字，后面紧接着方括号包围着链接的标识符。

为了定义链接标识符，用方括号包围链接标识符，前面可以有至多 3 个空格缩进，右方括号后面紧接着冒号和一个空格，空格后写链接地址，然后空格，在两个双撇号中间写一个链接地址的标题。例如

```
[baidu]: http://baidu.com/ "百度"
[pku]: http://www.pku.edu.cn/ "北大"
```

这时，在文章中就可以用标识符调用链接，如 [北京大学主页][pku] 将变成链接北京大学主页。



图 20.1:

使用引用式的链接，有些像论文中把所有参考文献排列在文章末尾，文中用到某一篇文章只要提及其序号。

还有一种链接是内部链接，仅在生成 HTML 结果时使用。在各级标题行的末尾，可以添加 `{# 自定义标签}` 这样的内容，其中“自定义标签”是自己写的一个标识符，标识符仅使用英文字母、数字、下划线、减号，用来区分不同的位置。比如，本文第一节“介绍”添加了 `markdown-intro` 为标签，就可以用“[回到介绍](#markdown-intro)”产生链接回到介绍。

20.4.9 插入图形

图形只能用链接形式，不可能保存到一个纯文本文件内。也是使用行内式和参考式两种形式。转化成 HTML、PDF、Word 格式后可以把图形内嵌在输出文件内部。

行内式的图片链接，是普通行内链接格式前面添加了一个叹号，惊叹后面紧接着方括号，方括号内写图片的标题，标题可以空缺，在右方括号后面紧接着圆括号，圆括号内写图片的链接。

例如，下面的代码可以插入百度的一个 logo，使用的是网上的资源：

```

```

结果为

为了插入保存在本地的与 Markdown 源文件在同一子目录或下级子目录的图形，只要在圆括号中写图片文件名（如果与 Markdown 源文件在同一子目录）或相对路径。例如，在 `D:\work\figs` 下的图形文件 `baidu_logo.gif` 在本 Markdown 源文件所在目录的 `figs` 子目录中，可以用代码

```

```

结果为

这样含有网上图片和本地图片的 Markdown 源文件转化为 HTML 和 docx 格式，都可以正常显示插入的图片。

与链接类似，也可以在文章某处（比如末尾）定义图片的标识符，然后把行内图片引用中图片地址替换成图片标识符即可。

20.4.10 表格

Markdown 文本格式的表格就像是用减号、等号、竖线画的文本格式表格一样，转化为 HTML、docx 等格式后就变成了富文本的表格。有如下几种表格：

- 管道表
- 简单表



图 20.2:

- 换行表
- 有格表

20.4.10.1 管道表

管道表在两列之间用竖线分开，在列标题下面用减号画横线，用如下方法指定各对齐方式：

- 在列标题下的横线开始加冒号，表示左对齐；
- 在列标题下的横线末尾加冒号，表示右对齐；
- 在列标题下的横线两端加冒号，表示居中对齐；
- 列标题下面仅有横线没有冒号，表示缺省对齐方式，一般是左对齐。

这种方法不需要输入内容上下对齐，适用于中文内容。后面所讲的简单表、换行表、有格表需要能够输入内容对齐，对于中英文混合内容很难做到对齐，所以仅管道表比较适合中文内容。

例如：

姓 名	收 入	职 业	颜 色 偏 好
赵四海	123456	业务经理	红
刘 英	50	无	蓝
钱德里	3200	保洁	灰

Table：管道表示例

结果为

姓名	收入	职业	颜色偏好
赵四海	123456	业务经理	红
刘英	50	无	蓝
钱德里	3200	保洁	灰

管道表不允许输入单元格换行，单元格内容太宽时转换结果可能自动换行，自动换行时列宽度与输入的列标题下横线宽度成比例。

20.4.10.2 简单表

简单表的格式是，第一行是各列标题，第二行是各标题下面用减号组成的表格线，同一行的不同列要用空格分开，从第三行开始是内容。

在表格前或表格后用空行隔开的以 **Table:** 开头的行是表格说明或标题。

为了确定表格每列单元格内容如何对齐，用列标题下的表格线给出提示：

- 表格线与列标题右对齐，表示该列右对齐；
- 表格线与列标题左对齐，表示该列左对齐；
- 列标题在表格线中间，表示该列居中对齐；
- 列标题左右都与表格线对齐，表示该列为缺省对齐方式，一般是左对齐。

一定要使用一个等宽字体来编辑这样的表格，否则对齐与否无法准确分辨。单元格内容不能超出表格线左端。经过试验发现，中文内容很难按这种方法对齐。

例如：

Name	Income	Job	Color
-----	-----	-----	-----
Jane	123456	Research Assistant	red
John	50	N/A	blue
William	3200	Cleaner	blue

Table: 一个简单表的例子

结果为：

表 20.2: 一个简单表的例子

Name	Income	Job	Color
Jane	123456	Research Assistant	red
John	50	N/A	blue
William	3200	Cleaner	blue

20.4.10.3 换行表

换行表在输入列标题和单元格内容时，允许输入内容拆分行，但是转化后并不拆分行。这样的表以一行减号开始，以一行减号结束，中间的表格用空行分开实际的不同行。例如：

```

-----
Name
of
Subject      Income      Job      color
-----
Jane      123456      Research Assistant      red
Ayer

John      50      N/A      blue
Tukey

William      3200      Cleaner      blue
Tale
-----

```

Table: 一个换行表的例子

结果为：

表 20.3: 一个换行表的例子

Name of Subject	Income	Job	color
Jane Ayer	123456	Research Assistant	red
John Tukey	50	N/A	blue
William Tale	3200	Cleaner	blue

换行表输入时各列的输入宽度是有作用的，输入较宽的列结果也较宽。

20.4.10.4 有格表

完全用减号、竖线、等于号、加号画出表格线。这样的表在文本格式下呈现出很好的表格形状。转化后不能指定对齐方式。

例如：

Table: 有格表示例

```
+-----+-----+-----+
| Fruit      | Price      | Advantages      |
+=====+=====+=====+
| Bananas    | $1.34      | - built-in wrapper |
|            |            | - bright color    |
+-----+-----+-----+
| Oranges    | $2.10      | - cures scurvy    |
|            |            | - tasty           |
+-----+-----+-----+
```

结果为

表 20.4: 有格表示例

Fruit	Price	Advantages
Bananas	\$1.34	<ul style="list-style-type: none">• built-in wrapper• bright color
Oranges	\$2.10	<ul style="list-style-type: none">• cures scurvy• tasty

Chapter 21

R Markdown 文件格式

21.1 R Markdown 文件

借助于 R 的 knitr 和 rmarkdown 扩展包的帮助，可以把 Markdown 格式的源文件中插入 R 代码，使得 R 代码的结果能够自动插入到最后生成的研究报告中。这种格式称为 R Markdown 格式，相应的源文件扩展名为.Rmd。输出格式可以是 HTML、docx、pdf、beamer 等。

knitr 的详细文档参见网站knitr 文档。

RStudio 是一个集成的 R 软件环境，可以用来编辑和执行 R 程序，这个软件也可以用来编辑和编译 R Markdown 格式的文件，使得 R Markdown 格式的文件变得容易使用。在 RStudio 中可以直接用一个快捷图标一次性地把 R 代码结果插入内容中并编译为 HTML 或 MS Word docx 格式，还支持 Markdown 中 LaTeX 格式的数学公式。建议使用 RStudio 软件作为 R Markdown 文件的编辑器。

在 RStudio 软件中，用菜单“File-New File-R Markdown”新建一个 R Markdown 文件，扩展名为.Rmd。用快捷图标可以将文件转换成 HTML 格式、PDF 格式（需要安装 LaTeX 编译软件）、MS Word 格式。

从 HTML 格式可以转换成 PDF 格式。为此，安装 Google 的 Chrome 浏览器，在 Chrome 中打开 HTML 文件后，从 Chrome 浏览器的菜单中找到打印菜单，从中选择打印机为“保存到 PDF”选项，就可以将 HTML 网页转换成 PDF，其中的数学公式、表格、图形都可以比较好地转换。

从 Word 文件也可以转换成 PDF 格式，用 MS Word 软件的“文件-导出”或者“文件-另存为”功能即可。

如果不借助于 RStudio 软件，可以用 R 软件、knitr 包、rmarkdown 包、pandoc 软件来完成 R Markdown 源文件的编译。比如，假设 test.Rmd 是一个这样的 R Markdown 格式的文件，注意一定要使用 UTF-8 编码保存，在不使用 RStudio 软件时，可以在 R 中运行如下命令以生成含有运行结果的 html 文件：

```
rmarkdown::render("myfile.Rmd", output_format = "html_document", encoding="UTF-8")
```

其中 myfile.Rmd 是源文件，产生的 HTML 文件带有图形、支持数学公式。

在 R 中可以用如下命令把.Rmd 文件转化为 MS Word docx 格式：

```
rmarkdown::render("myfile.Rmd", output_format = "word_document", encoding="UTF-8")
```

使用 RStudio 软件使得这些任务可以一键完成，而且有很好的数学公式支持，所以建议编辑 R Markdown 文件还是使用 RStudio 软件。

21.2 在 R Markdown 文件中插入 R 代码

插入的 R 代码分为行内代码与代码块。

行内代码的结果插入到一个段落中间，代码以 ``r` 开头，以 ``` 结尾，如 ``sin(pi/2)`` 在结果中会显示为 1。

代码块则把结果当作单独的段落，按照 Markdown 格式的规定，代码块的前后需要有空行，但是 R Markdown 实际上放松了这个要求，允许前后不空行。R 代码段以单独的一行开头，此行以三个反单撇号开始，然后是 `{r}`。代码段以占据单独一行的三个反单撇号结尾。如

```
` `{r}
1:5
sum(1:5)
` `
```

结果将变成

```
1:5
```

```
## [1] 1 2 3 4 5
```

```
sum(1:5)
```

```
## [1] 15
```

可以看出，代码段程序会被插入到最终结果中，代码段的文本型输出会插入到程序的后面。

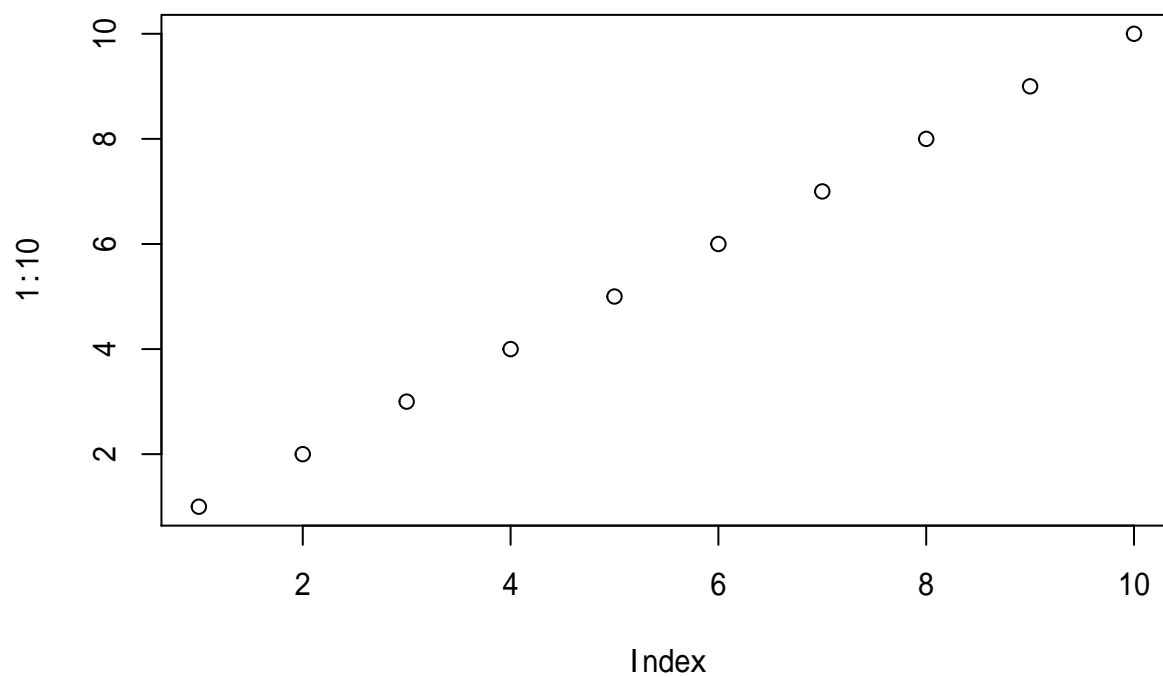
代码块也可以嵌入到引用、列表等环境中。

代码块中作的图将自动插入到当前位置。下面的程序：

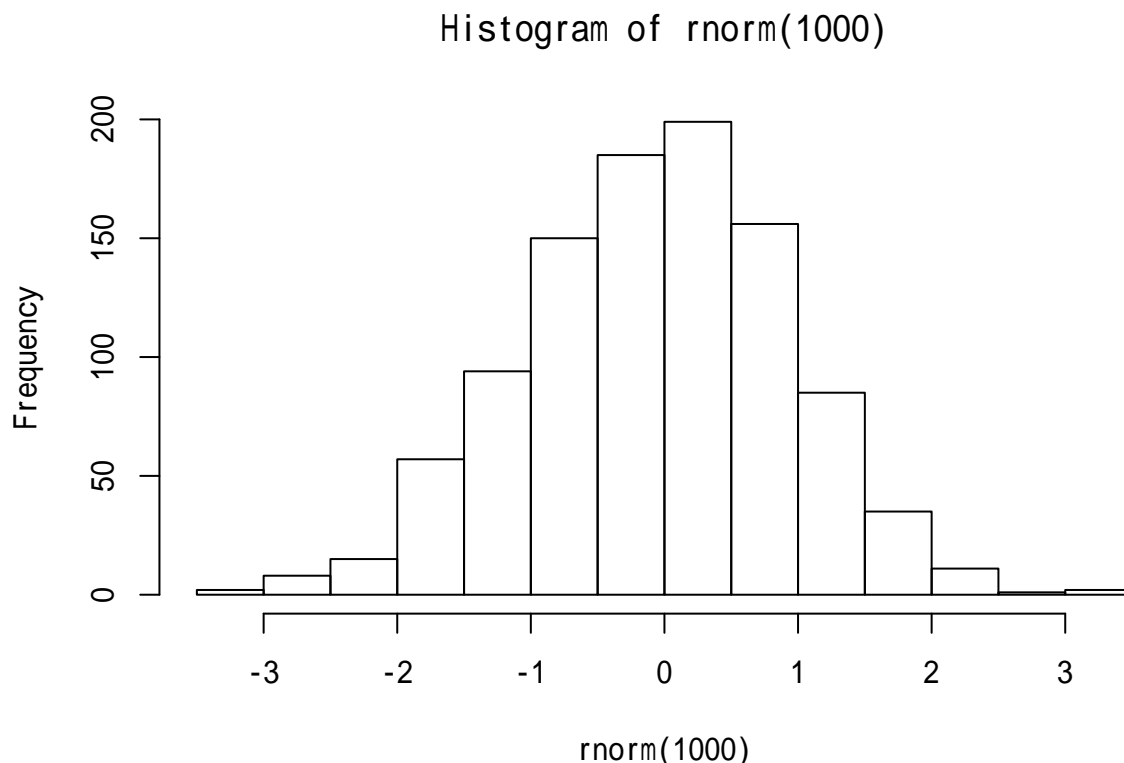
```
` `{r}
plot(1:10)
hist(rnorm(1000))
` `
```

结果将显示为:

```
plot(1:10)
```



```
hist(rnorm(1000))
```



为了将这样的 R 代码段包括首尾标志原样显示，需要将代码段整体地缩进 4 个空格，并在开始的三个反单撇号前面加上 ‘r’，即生成一个空字符串的行内 R 代码，原来的三个反单撇号的标志变成了 ‘r’‘’‘’。

在 RStudio 中，可以用 Insert 快捷图标插入代码段，还可以用 Ctrl+Alt+I 快捷键插入代码段。

21.3 输出表格

knitr 包提供了一个 `kable()` 函数可以用来把数据框或矩阵转化成有格式的表格，支持 HTML、docx、LaTeX 等格式。

例如，计算线性回归后，`summary()` 函数的输出中有 `coefficients` 一项，是一个矩阵，如果直接文本显示比较难看：

```
x <- 1:10; y <- x^2; lmr <- lm(y ~ x)
co <- summary(lmr)$coefficients
print(co)
```

```
##           Estimate Std. Error  t value    Pr(>|t|)
## (Intercept)      -22  5.5497748 -3.964125 4.152962e-03
## x                11  0.8944272 12.298374 1.777539e-06
```

可以用 knitr 包的 kable 函数来显示:

```
knitr::kable(co)
```

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	-22	5.5497748	-3.964125	0.0041530
x	11	0.8944272	12.298374	0.0000018

R 扩展包 xtable 提供了一个 xtable() 函数, 也可以用来生成 HTML 格式和 LaTeX 格式的表格, 但是需要指定要输出的格式。xtable 对比较多的 R 数据类型和输出类型提供了表格式显示功能, 包括矩阵、数据框、回归分析结果、方差分析结果、主成分分析结果、若干分析结果的 summary 结果等。例如, 上面的回归结果用 xtable() 函数显示如:

```
print(xtable::xtable(lmr), type='html')
```

Estimate

Std. Error

t value

Pr(>|t|)

(Intercept)

-22.0000

5.5498

-3.96

0.0042

x

11.0000

0.8944

12.30

0.0000

这个代码段用了选项 results='asis', 因为 xtable 生成的是直接用来插入到结果中的 html 代码。注意这里指定了输出为 HTML 类型。如果将本文件转化为 docx, xtable 的结果不可用。

21.4 利用 R 程序插图

Rmd 文件的插图有两种, 一种是已经保存为图形文件的, 主要是 png 和 pdf 图片; 另一种是文中的 R 代码生成的图形。

已经有图形文件的，可以用 markdown 格式原来的插图方法，见 markdown 格式介绍。但是，这样做不能给图形自动编号，另外因为制作图书是有网页和 PDF 书两种主要输出格式的，原有的插图方式在这两种输出格式上有细微的不一致。所以，最好是统一使用 Rmd 的插图方法。

Rmd 的插图方法就是写一段 R 代码段来插图，如果是用程序作图，则代码中写作图的代码；如果是已有的图形文件，可以在一个单独的 R 代码段中用类似下面的命令插图：

```
knitr::include_graphics("figs/myfig01.png")
```

其中 `figs` 是存放图形文件的子目录名，`myfig01.png` 是要插入的图形文件名。这样，如果同时还有 `myfig01.pdf` 的话，则 HTML 输出使用 png 图片而 PDF 输出自动选用 pdf 文件。另外，插图的选项在代码段的选项中规定：用代码段的 `fig.with` 和 `fig.height` 选项指定作图的宽和高（英寸），用 `out.width` 和 `out.height` 选项指定在输出中实际显示的宽和高，实际显示的宽和高如果使用如 "90%" 这样的百分数单位则可以自动适应输出的大小。

由于 PDF 中中文不能自动识别，所以在每个源文件的开头应该加上如下的设置，使得生成 PDF 图时中文能够正确显示：

```
```{r setup-pdf, include=FALSE}
pdf.options(family="GB1")
```
```

其中 `include=FALSE` 表示要不显示代码段的代码，有运行结果也不插入到输出结果中，是否允许视缺省的 `eval=` 的值而定。

21.5 代码段选项

独立代码段以三个反向单撇号和 `{r}` 开头，在大括号内还可以写一些选项，选项之间以及与开始的 `r` 之间用逗号分隔，所有选项写在同一行内不要换行。选项都使用“选项名 = 选项值”的格式，选项值除了使用常量外也可以使用全局变量名或表达式。在大括号内除了开头的 `r` 写一个不是选项名的名字，并与开头的 `r` 用空格分隔，作为代码段的标签。如

```
```{r firstCode}
cat('This is 第一段，有标签.\n')
```
```

21.5.1 代码和文本输出结果格式

R 代码块和 R 代码块的运行结果通常是代码块原样输出，运行结果用井号保护起来，这样有利于从文章中复制粘贴代码。如：

```
```{r}
s <- 0
```



```
for(x in 1:5) s <- s + x^x
s
...
```

结果为:

```
s <- 0
for(x in 1:5) s <- s + x^x
s
```

```
[1] 3413
```

### 21.5.1.1 highlight 选项

转化后的 R 代码块缺省显示为彩色加亮形式。用选项 `highlight=FALSE` 关闭彩色加亮功能。

### 21.5.1.2 prompt 和 comment 选项

如果希望代码用 R 的大于号提示符开始，用选项 `prompt=TRUE`。如果希望结果不用井号保护，使用选项 `comment=''`。例如：

```
```{r prompt=TRUE, comment='' }
sum(1:5)
...`
```

结果为:

```
> sum(1:5)
```

```
[1] 15
```

21.5.1.3 echo 选项

如果希望不显示代码，加选项 `echo=FALSE`。如

```
```{r echo=FALSE}
print(1:5)
...`
```

结果为:

```
[1] 1 2 3 4 5
```

### 21.5.1.4 tidy 选项

加选项 `tidy=TRUE` 可以自动重新排列代码段，使得代码段格式更符合规范。例如：

```
```${r tidy=TRUE}
s <- 0
for(x in 1:5) {s <- s + x^x; print(s)}
```
```

结果为:

```
s <- 0
for (x in 1:5) {
 s <- s + x^x
 print(s)
}
```

```
[1] 1
[1] 5
[1] 32
[1] 288
[1] 3413
```

#### 21.5.1.5 eval 选项和 include 选项

加选项 `eval=FALSE`, 可以使得代码仅显示而不实际运行。这样的代码段如果有标签, 可以在后续代码段中被引用。

加选项 `include=FALSE`, 则本代码段仅运行, 但是代码和结果都不写入到生成的文档中。

#### 21.5.1.6 child 选项

加选项 `child=' 文件名.Rmd'` 可以调入另一个.Rmd 文件的内容。如果有多个.Rmd 文件依赖于相同的代码, 可以用这样的方法。

#### 21.5.1.7 collapse 选项

一个代码块的代码、输出通常被分解为多个原样文本块中, 如果一个代码块希望所有的代码、输出都写到同一个原样文本块中, 加选项 `collapse=TRUE`。例如, 没有这个选项时:

```
```${r}
sin(pi/2)
cos(pi/2)
```
```

结果为:

```
sin(pi/2)
```

```
[1] 1
```

```
cos(pi/2)
```

```
[1] 6.123032e-17
```

代码和结果被分成了 4 个原样文本块。加上 `collapse=TRUE` 后，结果为：

```
sin(pi/2)
```

```
[1] 1
```

```
cos(pi/2)
```

```
[1] 6.123032e-17
```

代码和结果都在一个原样文本块中。

### 21.5.1.8 results 选项

用选项 `results=` 选择文本型结果的类型。取值有：

- `markup`, 这是缺省选项，会把文本型结果变成 HTML 的原样文本格式。
- `hide`, 运行了代码后不显示运行结果。
- `hold`, 一个代码块所有的代码都显示完，才显示所有的结果。
- `asis`, 文本型输出直接进入到了 HTML 文件中，这需要 R 代码直接生成 HTML 标签，knitr 包的 `kable()` 函数可以把数据框转换为 HTML 代码的表格。

例如：`results='hold'` 的示例：

```
```${r collapse=TRUE, results='hold'}``  
sin(pi/2)  
cos(pi/2)  
```
```

结果为：

```
sin(pi/2)
```

```
cos(pi/2)
```

```
[1] 1
```

```
[1] 6.123032e-17
```

## 21.5.2 图形选项

### 21.5.2.1 图形大小

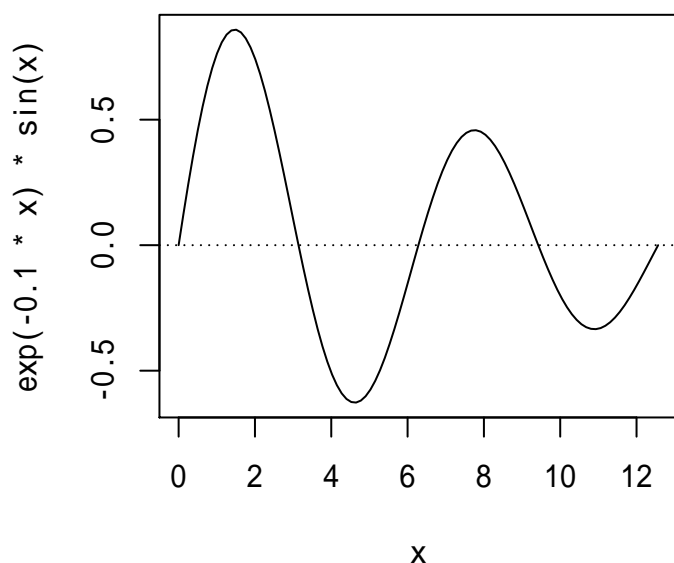
用 `fig.width=` 指定生成的图形的宽度，用 `fig.height=` 指定生成的图形的高度，单位是英寸（1 英寸等于 2.54 厘米）。

下面给出一个长宽都是 10 厘米的图例。

```
```{r fig.width=10/2.54, fig.height=10/2.54}
curve(exp(-0.1*x)*sin(x), 0, 4*pi)
abline(h=0, lty=3)
```
```

结果为:

```
curve(exp(-0.1*x)*sin(x), 0, 4*pi)
abline(h=0, lty=3)
```



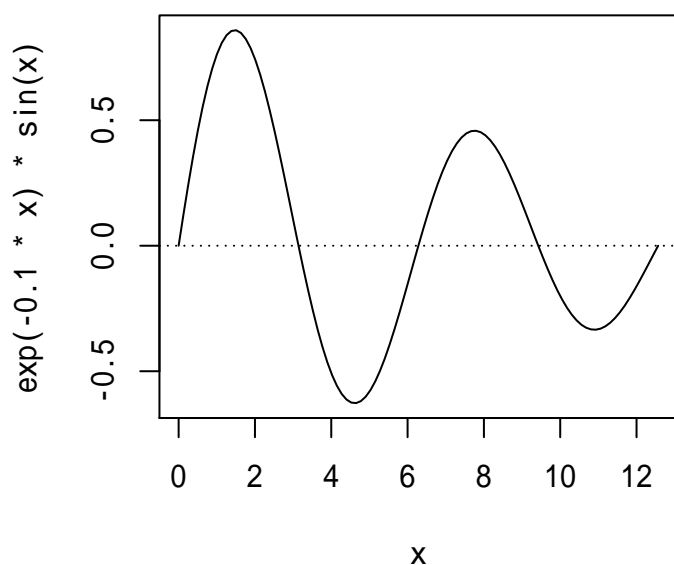
`fig.width=` 和 `fig.height=` 规定的是生成的图形大小，实际生成图形的显示大小会受到 `dpi`（分辨率）影响，默认 `dpi` 是 72（每英寸 72 个点），也可以用 `dpi=` 选择分辨率。转化后的 HTML 文件显示时不一定按原始大小显示。用 `out.width=` 和 `out.height=` 可以指定显示大小，但是可以是最终文件格式承认的单位，比如 HTML 的图形大小单位是点（平常说屏幕分辨率的单位）。也可以写成如 90% 这样的百分比格式。例如在上面的例子中加上输出长宽都是 600 点的选项：

```
```{r fig.width=10/2.54, fig.height=10/2.54, out.width=600, out.height=600}
```

```
curve(exp(-0.1*x)*sin(x), 0, 4*pi)
abline(h=0, lty=3)
...
```

注意所有选项都要写在一行中，不能换行。结果为：

```
curve(exp(-0.1*x)*sin(x), 0, 4*pi)
abline(h=0, lty=3)
```

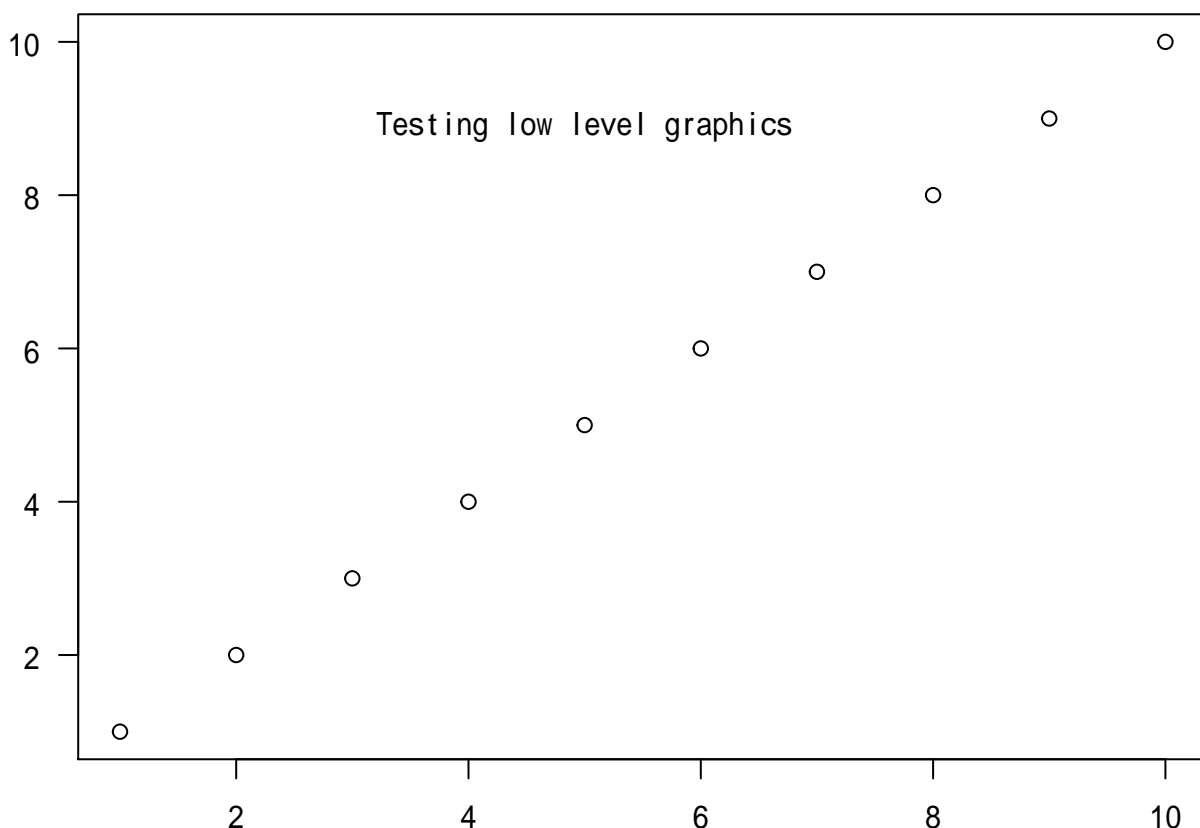


在 knitr 的 LaTeX 版本（扩展名为.Rnw）中，连续两个图形如果指定的宽度加起来比行宽窄会自动并排显示，R 的 bookdown 扩展包提供了 R Markdown 格式的扩展，其中的图形可以变成浮动图，而且很容易并列放置。bookdown 插图的 `out.width` 和 `out.height` 选项可以使用百分数，如 `out.width="90%"`，bookdown 包会自动将百分数按照不同输出格式的要求进行转换。

21.5.2.2 图形结果选择

用 `fig.keep=` 选项可以选择保留哪些 R 代码生成的图。缺省是 `fig.keep='high'`，即保留每个高级图形函数的结果图形，低级图形函数对高级图形函数的更改不单独保存而汇总到高级图形函数结果中。如

```
par(mar = c(3, 3, 0.1, 0.1))
plot(1:10, ann = FALSE, las = 1)
text(5, 9, "Testing low level graphics")
```



其中 `text()` 函数的结果与高级图形函数 `plot()` 的结果一起显示。

`fig.keep` 还可以取: `all`, 会把低级图形函数修改后的结果单独保存; `last`, 仅保留最后一个图形; `first`, 仅保留第一个图; `none`, 所有图都不显示出来。

21.5.3 缓存 (cache) 选项

当 R Markdown 文章比较长, 包含的 R 代码比较多, 或者代码段运行需要比较长时间时, 反复编译整篇文章会造成不必要的计算, 因为有些代码段并没有修改, 依赖的数据也没有改变。knitr 提供了缓存功能, 代码段选项 `cache=TRUE` 对代码段打开缓存, 允许暂存上次运行的结果 (包括文本结果和图形) 而不需要重复运行代码段。当代码段被修改时, 缓存被放弃, 编译时重新运行代码段。

缓存这种功能需要慎重使用, 免得错误地使用了旧的结果。当后面的代码段需要使用前面代码段结果时, 如果前面结果改了, 后面的代码段就不能使用缓存的结果而必须重新计算。为此, 在后面的代码段中应该加上 `dependson=` 选项, 比如 `dependson=c('codeA', 'codeB')`, 其中 `codeA` 和 `codeB` 是前面的缓存了的代码段的标签, 其结果会用在本地代码段中。也可以使用代码段选项 `autodep=TRUE`, knitr 试图自动确定前后代码段之间的依赖关系, 每当前面的代码段改变时, 后面的用到其结果代码段也自动重新计算而不使用缓存的旧结果。

建议仅对计算一次需要较长时间的代码段使用缓存功能, 后面依赖于其结果的代码一定要加上 `dependson=` 选项。建议代码段尽可能有标签, 这样在编译失败时能马上看出失败的地点。

21.6 数学公式

21.6.1 在 Markdown 中输入数学公式

原始的 Markdown 格式并不支持数学公式。Pandoc 扩展的 markdown 格式提供了对数学公式的支持，可以在 Markdown 文件中插入 LaTeX 格式的数学公式。虽然不能提供所有的 LaTeX 公式能力，但是常用的数学公式还是能做得很好，转换到 HTML、docx 都可以得到正常显示的公式。

用 RStudio 软件编译 Markdown 文件，可以在其中插入 LaTeX 格式的数学公式，编译成 HTML 或者 docx 格式后都可以正常显示数学公式。使用 bookdown 包可以在另外安装的 LaTeX 编译器的支持下将.Rmd 格式编译为 PDF 格式。

如果不使用 RStudio, R 扩展包 rmarkdown 的 `markdownToHTML()` 函数可以把含有数学公式的内容转换成可显示公式的 HTML 文件，R 扩展包 knitr 的 `knit2html()` 也可以实现此功能。用 rmarkdown 扩展包的 `markdownToHTML()` 函数生成的含有数学公式的内容。

单独使用 pandoc 软件时，也可以用普通文本编辑器在 Markdown 文件中输入 LaTeX 格式的数学公式，然后用 pandoc 软件转化为带有数学公式的 docx 文件，不需要额外选项。但是，为了能够在转换的 HTML 文件中正常显示数学公式，需要在运行 Pandoc 时增加运行选项 `--mathjax`，如

```
pandoc --mathjax -s -o test.html test.md
```

经试验，这样转化的 HTML 文件可以在 Firefox 以及 Microsoft 的 Edge 浏览器和 Internet Explorer 浏览器中正常显示数学公式。如果公式中的中文显示不正常，对显示的公式右键单击弹出选项菜单，选择“Math Settings-Math Renderer-SVG”或者“HTML-CSS”。

21.6.2 数学公式类别

数学公式分为行内公式和独立公式。行内公式和段落的文字混排，写在两个美元符号 `$` 中间，或者 `\(` 和 `\)` 之间。例如 `$f(x)=\frac{1}{2} \int_0^1 \sin^2(t x) dt$` 变成 $f(x) = \frac{1}{2} \int_0^1 \sin^2(tx)dt$ 。开头的 `$` 后面不能紧跟着空格，结尾的 `$` 不能紧跟在空格后面。

独立公式写在成对的美元符号中间，或者 `\[` 和 `\]` 之间。例如：

```
$$
f(x) = \frac{1}{2} \sum_{j=1}^{\infty} \int_0^1 \sin^2(j t x) dt .
$$
```

显示为

$$f(x) = \frac{1}{2} \sum_{j=1}^{\infty} \int_0^1 \sin^2(jtx)dt.$$

在数学公式中，用下划线表示下标，比如 `x_1` 结果为 x_1 。用 `^` 表示上标，如 `x^2` 结果为 x^2 。上下标都有，如 `x_1^2` 结果为 x_1^2 。

公式中用大括号表示一个整体, 比如 $x^{(1)}$ 不能得到 $x^{(1)}$, 需要用 $x^{\{1\}}$ 。

公式中用反斜杠开始一个命令, 命令仅包含字母而不能包含数字, 数字只能作为参数。如 $\frac{1}{2}$ 和 $\frac{1}{2}$ 都可以生成 $\frac{1}{2}$ 。类似的命令如 $\sqrt{2}$ 为 $\sqrt{2}$ 。

求和如 $\sum_{i=1}^n x_i$ 变成 $\sum_{i=1}^n x_i$ 。乘积如 $\prod_{i=1}^n x_i$ 变成 $\prod_{i=1}^n x_i$ 。积分如 $\int_0^1 f(x) dx$ 变成 $\int_0^1 f(x) dx$ 。

为了产生对齐的公式, 在独立公式中使用 aligned 环境。公式中的环境以 `\begin{环境名}` 开始, 以 `\end{环境名}` 结束, 用 `\\` 表示换行, 用 `&` 表示一个上下对齐位置。如

```
$$
\begin{aligned}
f(x) &= \sum_{k=0}^{\infty} \frac{1}{k!} x^k \\
&= e^x
\end{aligned}
$$
```

转化成

$$f(x) = \sum_{k=0}^{\infty} \frac{1}{k!} x^k = e^x$$

关于数学公式的软件设置参加下面设置部分的内容。

21.7 属性设置

21.7.1 YAML 元数据

可以在 RStudio 软件中编辑 markdown 文件与 Rmarkdown 文件, RStudio 的 markdown 支持来自 knitr、rmarkdown、bookdown 扩展包和 pandoc 软件, .Rmd 文件支持一些特殊的文件设置, 这些设置写在 markdown 文件和 .Rmd 文件的开头位置, 用三个减号组成的行作为开始标记与结束标记, 称为 YAML 元数据块 (YAML meta data block), 块后面必须用空行分隔。元数据块中可以用 “设置属性名: 设置属性值” 的办法设置属性, 用缩进表示属性内容, 用上下对齐的行表示多项列表。常用属性有 title(标题)、author(作者)、date(日期)、output_format 等, 如

```
---
title: "临时测试"
author: "李东风"
date: "2016年7月6日"
---
```

这三个设置会出现在转换后结果的标题部分。RStudio 的中文支持还是有时出现问题, 如果出现涉及到 YAML 的错误, 先将中文内容替换为英文试一试。

因为冒号：在属性设置中有特殊意义，属性设置值如果含有冒号，需要把整个属性值两边用单撇号界定。

属性值可以是列表或多项，例如：

```
---
title: 'This is the title: it contains a colon'
author:
  - name: Author One
    affiliation: University of Somewhere
  - name: Author Two
    affiliation: University of Nowhere
tags: [nothing, nothingness]
abstract: |
  This is the abstract.

  It consists of two paragraphs.
---
```

此例中有 title、author、tags、abstract 四个属性。author 属性包含两个作者，每个作者又有 name 和 affiliation 两个属性。tags 属性是一个两项的列表。abstract 属性用管道符号 | 表示开头，不需要结束标志，内容缩进。

21.7.2 输出格式设置

.Rmd 文件开头的 YAML 元数据中，属性 output 选择输出的格式，如 html_document 是 HTML 输出，pdf_document 是 PDF 输出，word_document 是 docx 格式的 Word 文件输出，等等。在每种输出格式后面还可以继续添加该输出特有属性。如：

```
output:
  html_document:
    toc: yes
  word_document:
    toc: yes
```

上例中，output 属性的值是两项，两项用缩进表示；html_document 又有自己的属性 toc，属性值 yes 表示要自动生成可点击的目录。设置某种输出格式的属性 number_sections 为 yes 可以自动对章节编号。

如果输出格式没有自己的属性，可以写成 default，如：

```
output:
  html_document: default
  word_document: default
```

21.7.3 章节目录链接问题

对于英文文件, R Markdown 基于的 pandoc 软件可以自动从标题生成适当的链接标签, 对于中文文件的支持差一些, 所以中文文件要自动生成可点击的目录, 需要在每个标题行的末尾, 空格后添加 `{#label}`, 其中 `label` 是自己指定的标签内容, 但是建议仅使用英文字母、数字、减号。如

```
### 第三章 第一节标题 {#c3-s1-int}
```

21.7.4 关于数学公式支持的设置

Rmd 格式支持数学公式, 在生成的 HTML、Word、PDF 中都可以正常显示数学公式。以 HTML 为输出时, 使用 MathJax 库显示数学公式。这是一个用于在浏览器中显示数学公式的 Javascript 程序库, 显示效果很好, 还支持多种显示实现方式, 支持包括 LaTeX 在内的多种数学公式输入方法。这个库很大, 所以一般是从其网站按需远程调用的, 但是远程调用 MathJax 库在网络不畅通时会使得公式显示极为缓慢甚至无法显示, 所以 Rmd 允许将 MathJax 本地化。在.Rmd 文件头部 YAML 元数据的某种 HTML 输出格式的属性中, 设置属性 `mathjax: local` 和就可以使得 MathJax 库被放在生成的 HTML 的一个下层目录中。设置如

```
output:
  html_document:
    toc: yes
    self_contained: yes
    mathjax: local
```

21.7.5 输出设置文件

如果一个项目包含多个.Rmd 文件, 每个文件单独用 YAML 元数据进行设置比较繁琐, 修改时也很麻烦。可以在项目目录中增加一个 `_output.yml` 文件, 其中包含所有.Rmd 文件共用的 `output` 属性的设置, 各个.Rmd 文件还可以有自己单独的 `output` 属性, .Rmd 文件 YAML 元数据中的属性优先级高于共同设置的属性。

例如, `_output.yml` 文件内容如下:

```
html_document:
  toc: yes
  mathjax: "../MathJax/MathJax.js"
  self_contained: yes
  includes:
    in_header: "_header.html"
```

上面设置了使用本地硬盘的 MathJax 库, 位于项目目录的上层目录中的 MathJax 子目录中。这样可以使得多个项目共享一个 MathJax 库。将生成的结果当作网站发布时, 只要将 MathJax 库也安装到项目在网

站的目录的上层的 MathJax 子目录中就可以。其中 `_header.html` 内容如下，是关于 MathJax 具体的一些设置：

```
<script type="text/x-mathjax-config">
MathJax.Hub.Config({
  jax: ["input/TeX","output/SVG"],
  extensions: ["tex2jax.js","MathMenu.js","MathZoom.js"],
  TeX: {
    extensions: ["AMSmath.js","AMSsymbols.js","noErrors.js","noUndefined.js"]
  }
});
</script>
```


Chapter 22

用 bookdown 制作图书

22.1 介绍

R 的 bookdown 扩展包 (<https://github.com/rstudio/bookdown>) 是继 knitr 和 rmarkdown 扩展包之后, 另一个增强 markdown 格式的扩展, 使得 Rmd 格式可以支持公式、定理、图表自动编号和引用、链接, 文献引用和链接等适用于编写书籍的功能。在 bookdown 的管理下一本书的内容可以分解成多个 Rmd 文件, 其中可以有可执行的 R 代码, R 代码生成的文字结果、表格、图形可以自动插入到生成的内容中, 表格和图形可以是浮动排版的。输出格式主要支持 gitbook 格式的网页图书, 这种图书在左侧显示目录, 右侧显示内容, 并可以自动链接到上一章和下一章; 通过单独安装的 LaTeX 编译器支持将书籍转换为一个 PDF 文件, 支持中文; 可以生成 ePub 等格式的电子书。

主要用于编写有多个章节的书籍, 也可以用来生成单一文件的研究报告。

22.2 入门

建议使用 RStudio 集成环境, 其中内建了一键编译整本书的功能。需要安装 bookdown 扩展包的最新版本。bookdown 扩展包现在还比较新, 还有一些 BUG, 所以尽可能使用最新版的 bookdown 扩展包并且及时更新 RStudio 软件。查看编译的网站建议使用 Google Chrome 浏览器, 此浏览器对 gitbook 的支持较好。

为了新写一本书或者从已有的书转换, 最简单的做法是从 bookdown 的网站下载 bookdown 配套的例书的 zip 文件 (见<https://github.com/rstudio/bookdown-demo>), 将其解压到本地硬盘某个子目录, 然后修改其中的内容适应自己的书的需要。

22.3 一本书的设置

一本用 bookdown 管理的书, 一般放置在某个子目录下, 并作为一个 RStudio 项目 (project) 用 RStudio 管理。

也可以自己新建一个目录，然后编辑生成必要的文件。注意，所有的文本文件都要使用 UTF-8 编码。一本 bookdown 书，一般都需要有一个 `index.Rmd` 文件，这是最后生成的网站的主页的原始文件，可以在这个文件中写一些书的说明，并在开头的 YAML 元数据部分进行有关设置，如标题、作者、日期等。`index.Rmd` 的一个例子如下：

```
---
title: "统计计算"
author: "李东风"
date: "2018-03-07"
site: bookdown::bookdown_site
output: bookdown::gitbook
documentclass: book
bibliography: [myrefs.bib]
biblio-style: apalike
link-citations: yes
description: "本科生《统计计算》教材。采用R的bookdown制作，输出格式为bookdown::gitbook."
---

# 前言 {-}
```

统计计算研究如何将统计学的问题用计算机正确、高效地实现。

其中在三个减号组成的两行之间的内容叫做 YAML 元数据，是一本书的设置，上例中有书的标题、作者名、日期（用 R 程序自动生成）、描述。其中的 `site` 选项很重要，一定要有这个选项，`site: bookdown::bookdown_site` 使得 RStudio 软件能辨认这是一个 bookdown 图书项目，从而为其提供一键编译快捷方式。元数据中 `output` 项指定默认的输出格式。`documentclass` 项为借助 LaTeX 编译 PDF 格式指定 LaTeX 的模板，现在还不能支持 `ctexbook` 模板所以使用了 `book` 模板。`bibliography` 项指定一个或者几个 .bib 格式的文献数据库。

一个 bookdown 图书项目除了 `index.Rmd` 文件之外，一般还应该有一个 `_bookdown.yml` 文件存放与整本书有关的 YAML 元数据。例如

```
new_session: true
book_filename: 'statcompc'
language:
  label:
    thm: '定理'
    def: '定义'
    exm: '例'
    proof: '证明: '
    solution: '解: '
    fig: '图'
```

```

    tab: '表'
  ui:
    chapter_name: ''
delete_merged_file: true

```

其中 `new_session: true` 设置很重要, 这使得每一个 Rmd 文件中的 R 程序都在一个单独的 R 会话中独立地运行, 避免了不同 Rmd 文件之间同名变量和同名标签的互相干扰。`book_filename` 是最终生成的 LaTeX PDF 图书或者 ePub 电子书的主文件名。`language` 下可以定制一些与章节名、定理名等有关的名称。

另外一个需要的设置文件是 `_output.yml` 文件, 用于输出格式的设置。这部分内容也可以包含在 `index.Rmd` 的元数据部分。内容如

```

bookdown::gitbook:
  includes:
    in_header: mathjax-local.html
  config:
    toc:
      before: |
        <li><a href="http://www.math.pku.edu.cn/teachers/lidf/course/statcomp/_book/index.html">统
      after: |
        <li><a href="http://www.math.pku.edu.cn/teachers/lidf/" target="blank">编著: 李东风</a></li>
    download: ["pdf"]
bookdown::pdf_book:
  includes:
    in_header: preamble.tex
  latex_engine: xelatex
  citation_package: natbib
  keep_tex: yes
bookdown::epub_book: default

```

其中的 `style.css` 是自定义的 CSS 显示格式, 可以去掉这一行, 使用默认的 CSS 格式。这个例子文件分为三部分, `gitbook`、`pdf_book` 和 `epub_book` 三种输出格式分别设置了一些输出选项。在 `gitbook` 部分, 设置了目录上方显示的书的主页的链接 (`before` 项) 和目录下方显示的作者信息。在 `in_header` 部分插入了一部分个性化的 HTML 代码, 这部分代码是使用本地的数学公式显示支持以免外网不同时数学公式不能显示, 插入的内容将出现在每个生成的 HTML 文件的 `head` 部分。

在 `pdf_book` 部分, 设置了通过 LaTeX 编译整本书为 PDF 的一些选项。指定了 `latex_engine` 为 `xelatex`, 这对中文支持很重要。`in_header` 选项要求在 LaTeX 文件导言部分插入一个 `preamble.tex` 文件, 内容如:

```
\usepackage{ctex}
```

```
%\usepackage{xltextra} % XeLaTeX的一些额外符号
% 设置中文字体
\setCJKmainfont[BoldFont={黑体},ItalicFont={楷体}]{新宋体}

\usepackage{amsthm,mathrsfs}
\usepackage{booktabs}
\usepackage{longtable}
\makeatletter
\def\thm@space@setup{%
  \thm@preskip=8pt plus 2pt minus 4pt
  \thm@postskip=\thm@preskip
}
\makeatother
```

其中很重要的是使用 `ctex` 包来支持中文。

在 `bookdown` 项目中与 `index.Rmd` 同级的所有 `.Rmd` 文件都自动作为书的一章，除非文件名以下划线开头。这样做的好处是作者可以任意地增删章节，编译整本书时章节编号会自动调整。但是，章节的顺序将按照文件名的字典序排列，所以，所有的包含一章内容的 `.Rmd` 文件，最好命名为类似 `0201-rng.Rmd` 这样的名字，文件名前面人为地加上排序用的序号，使得章节按照自己的次序排列。实际上，也可以设置不自动将每个 `.Rmd` 文件都作为一章，而是在 `_output.yml` 中设置一项 `rmf_files`，列出所有需要作为一章的文件，并以列出次序编译，如

```
rmf_files: ["index.Rmd", "rng.Rmd", "simulation.Rmd", "refs.Rmd"]
```

这时，应该添加一个 `_site.yml` 文件，内容如：

```
site: "bookdown::bookdown_site"
output: bookdown::gitbook
```

22.4 章节结构

除了 `index.Rmd` 文件，项目中每个 `.Rmd` 文件都作为一章。每个 `.Rmd` 文件第一行，应该是以一个井号和空格开头的一级标题，后面再加空格然后有大括号内以井号开头的章标签，如

```
# 随机数 {#rng}
```

这些章标签去掉井号后会作为生成的 HTML 文件的名字，所以一定要有章标签，而且章节标签在全书中都不要重复以免冲突。文件内可以用两个井号和一个空格开始的行表示节标题，最后也应该有大括号内以井号开头的节标签，如

```
# 随机数 {#rng}
```

```
## 均匀随机数发生器 {#rng-unif}
```


使用 bookdown 写书，一般每章不要太长，否则编译预览很慢，读者浏览网页格式也慢。

内容相近的章节可以作为一个“部分”。为此，在一个部分的第一个章节文件的章标题前面增加一行，以 # (PART) 开头，以 {-} 结尾，中间是部分的名称，如

```
# (PART) 随机数和随机模拟 {-}
```

```
# 随机数 {#rng}
```

书的最后可以有附录，附录的章节将显示为 A.1, B.1 这样的格式。为此，在附录章节的第一个文件开头加如下前两行的标题行：

```
# (APPENDIX) 附录 {-}
```

```
# (PART) 附录 {-}
```

```
# 一些定理的证明 {#formula}
```

22.5 书的编译

建议使用 RStudio 软件编辑内容，管理和编译整本书。

在 index.Rmd 或者 _bookdown.yml 中设置 site: bookdown::bookdown_site 后，RStudio 就能识别这个项目是一个 bookdown 项目，这时 RStudio 会有一个 Build 窗格，其中有“Build book”快捷图标，从下拉菜单中选择一个输出格式（包括 gitbook、pdf_book、epub_book），就可以编译整本书。对 gitbook 格式，即 HTML 网页格式，编译完成后会弹出一个预览窗口，其中的“Open in Browser”按钮可以将内容在操作系统默认的网络浏览器中打开。也可以在命令窗口用如下命令编译（以输出 gitbook 为例）：

```
bookdown::render_book("index.Rmd",  
  output_format="bookdown::gitbook", encoding="UTF-8")
```

编译结果默认保存在 _book 子目录中，可以在 _bookdown.yml 中设置 output_dir 项改为其它子目录。编译整本书为 pdf_book 格式时，如果成功编译，也会弹出一个 PDF 预览窗口。可以在 _book 子目录中找到这个 PDF 文件。

将书编译为 PDF 需要利用 LaTeX 编译器，这需要单独安装 LaTeX 编译软件，如 Windows 下的 CTEX 套装软件。LaTeX 编译器对输入要求十分严格，一丁点儿错误都会造成整本书的编译失败，所以对于不熟悉 LaTeX 的用户，不建议使用 bookdown 的 pdf_book 输出格式。

对于较短的书，做了一定修改后都可以重新编译 gitbook 结果和 pdf_book 结果。在书比较长了以后，每次编译都花费很长时间，所以可以仅编译 gitbook 格式的一章，修改满意后再编译整本书。仅编译一章也需要所有的.Rmd 文件都是已经编译过一遍的，新增的 Rmd 文件和图形文件会使得编译单章出错，每次新增了 Rmd 文件和图形文件都应该重新编译整本书，但是内容修改后不必要重新编译整本书，可以仅编译单章。

编译单章现在没有快捷图标，只能在 RStudio 控制台（命令行）运行如下命令：

```
bookdown::preview_chapter("chap-name.Rmd",
  output_format="bookdown::gitbook", encoding="UTF-8")
```

其中 `chap-name.Rmd` 是要编译的单章的文件名。编译完成后在结果目录（默认是 `_book`）中找到相应的 HTML 文件打开查看，再次编译后仅需在浏览器中重新载入文件。建议使用 Google chrome 浏览器，用 MS IE 或者 Edge 浏览器对 gitbook 的 Javascript 支持不够好，使得目录的层级管理、自动滚动、单章编译后的目录更新不正常，而 chrome 则没有问题。

编译单章也不能解决所有的问题，有些问题还是需要编译整本书，而章节很多时整本书编译又太慢。为此，可以在项目中增加一个临时的部分内容子目录，如 `testing` 子目录，在子目录中存放相同的设置文件 `index.Rmd`、`_bookdown.yml`、`_output.yml`，以及图形文件、文献数据库文件，并将要检查的若干章节复制到 `testing` 子目录中，在 `testing` 中新建一个 bookdown 项目，然后编译其中的整本书。这在调试部分章节的 HTML 和 PDF 输出时很有效。解决问题后主要将修改过的章节复制回原始的书的目录中。

有时仅仅想验证某个长数学公式或者表格，用上述的编译单章或者单独一个小规模测试项目的办法也不经济。这时，单独开一个备用的普通 RStudio 项目，不能是 bookdown 项目，在其中的 Rmd 文件中验证数学公式和表格的编排，这样效率最快了。

22.6 数学公式和公式编号

通过 R 的 knitr 和 rmarkdown 扩展包，.Rmd 格式文件已经支持数学公式，见 markdown 说明。

在用 `$$` 符号在两端界定的公式后面，可以用 `\tag{标号}` 命令增加人为的公式编号，如

```
$$
y = f(x)
\tag{*}
$$
```

结果显示为

$$y = f(x) \tag{*}$$

要注意的是，在 `$$` 界定的数学公式内用了 `aligned` 环境后，仅能在 `\end{aligned}` 之后加 `\tag{标号}` 命令，而不能写在 `aligned` 环境内。这样，多行的公式将不能为每行编号。

用 `\tag` 命令人为编号比较简单易用，但是在有大量公式需要编号时就很不方便，只要增加了一个公式就需要人为地重新编号并修改相应的引用。bookdown 包支持对公式自动编号，并可以按公式标签引用公式，引用带有超链接。

bookdown 的自动编号对 LaTeX 的 `equation` 环境、`align` 环境都可以使用，而且不需要在两端用 `$$` 界定。在公式的末尾或者一行公式的 `\\` 换行符之前，写 `(\#eq:mylabel)`，其中 `mylabel` 是自己给公式的文字标签，文字标签可以使用英文字母、数字、减号、下划线。如

```
\begin{align}
f(x) &= \sum_{k=0}^{\infty} \frac{1}{k!} x^k \label{eq:efunc-sum} \\
&= e^x \label{eq:efunc-ex}
\end{align}
```

将会对两行公式自动编号。引用公式时，用如`\@ref{eq:mylabel}`，其中 `mylabel` 是公式的自定义标签，编译后这样的引用会变成带有链接的圆括号内的编号。

公式编号在全书中都不要有冲突（不同的公式定义了相同的编号）。一种办法是，自定义的公式标签的开头以章节文件名开头。

22.7 定理类编号

定理、引理、命题、例题等，使用特殊的 markdown 代码格式，以三个反单撇号开头，以三个反单撇号结尾，在开头的三个反单撇号后面写 `{theorem}` 表示定理。在 `theorem` 后面，用逗号分隔后写一个定理的自定义标签，因为现在 bookdown 的功能还不完善，所以所有的定理类都应该自定义一个标签。可以用 `name="定理名称"` 指定一个显示的定理名。标签也可以写成 `label="mythlabel"`，其中 `mythlabel` 是自定义的标签。

设某个定理的自定义标签是 `mythlabel`，则可以用如`\@ref{thm:mythlabel}` 引用此定理的编号，编号有自动生成的链接。

bookdown 提供了证明环境，但是不太实用。

对例题，将 `theorem` 替换成 `example`，在引用时将 `thm` 替换成 `exm`。

22.8 文献引用

bookdown 使用 .bib 格式的文献数据库，关于 .bib 格式的文献数据库请参考 LaTeX 的有关说明。在 `index.Rmd` 的 YAML 元数据部分或者 `_bookdown.yml` 中用 `bibliography` 可以设置使用的一个或者多个 .bib 格式的文献数据库文件。设某篇文章的 .bib 索引键是 `Qin2007:comp`，用 `@Qin2007:comp` 可以引用此文献，用 `[@Qin2007:comp]` 可以生成带有括号的引用，引用都有超链接。

22.9 插图

bookdown 图书的插图有两种，一种是已经保存为图形文件的，主要是 png 和 pdf 图片；另一种是文中的 R 代码生成的图形。

已经有图形文件的，可以用 markdown 格式原来的插图方法，见 markdown 格式介绍。但是，这样做不能给图形自动编号，另外因为制作图书是有网页和 PDF 书两种主要输出格式的，原有的插图方式在这两种输出格式上有细微的不一致。所以，最好是统一使用 Rmd 的插图方法。

Rmd 的插图方法就是写一段 R 代码段来插图，如果是用程序作图，则代码中写作图的代码；如果是已有的图形文件，可以在一个单独的 R 代码段中用类似下面的命令插图：

```
knitr::include_graphics("figs/myfig01.png")
```

其中 `figs` 是存放图形文件的子目录名，`myfig01.png` 是要插入的图形文件名。这样，如果同时还有 `myfig01.pdf` 的话，则 HTML 输出使用 png 图片而 PDF 输出自动选用 pdf 文件。另外，插图的选项在代码段的选项中规定：用代码段的 `fig.with` 和 `fig.height` 选项指定作图的宽和高（英寸），用 `out.width` 和 `out.height` 选项指定在输出中实际显示的宽和高，实际显示的宽和高如果使用如 "90%" 这样的百分数单位则可以自动适应输出的大小。

为了使得插图可以自动编号并可以被引用，为代码段指定标签并增加一个 `fig.cap="..."` 选项指定图形标题。代码段的标签变成浮动图形的标签，如 `myfiglabel`，则为了引用这个图只要用 `\@ref(fig:myfiglabel)`。注意，在整本书中这些标签都不能重复，否则编译 LaTeX 支持的 PDF 输出会失败。

由于 PDF 中中文不能自动识别，所以在每个源文件的开头应该加上如下的设置，使得生成 PDF 图时中文能够正确显示：

```
```{r setup-pdf, include=FALSE}
pdf.options(family="GB1")
```
```

其中 `include=FALSE` 表示要不显示代码段的代码，有运行结果也不插入到输出结果中，是否允许视缺省的 `eval=` 的值而定。

22.10 表格

bookdown 书的表格也有两种，一种是原来 markdown 格式的表格，最好仅使用管道表，管道表对中文内容支持最好。为了对这样的表格自动编号，需要在表格的前面或者后面空开一行的位置，写

Table: `\label{tab:mylabel}` 表的说明

其中 `mylabel` 是自定义的表格标签。在引用这个表时用如 `\@ref(tab:mylabel)`。

另一种表格是 R 代码生成的表格，主要使用 `knitr::kable()` 函数。在 `knitr::kable()` 函数中用选项 `caption=` 指定表格的说明文字（标题），这时生成表格的 R 代码段的标签，如 `myfiglab`，就自动构成了表格的引用标签主干，实际引用如 `\@ref(tab:myfiglab)`。

为了适应较长的表，可以在 LaTeX 的 `preamble.tex` 中引入 `longtable` 包，并在 `knitr::kable()` 中加选项 `longtable=TRUE`。

22.11 数学公式的设置

bookdown 在生成 PDF 时使用 LaTeX 软件，所以 PDF 输出的数学公式的支持很好，但是 LaTeX 编译器也很挑剔，稍微一点错误也造成编译失败。比如，在行内公式内部如果紧邻 $\$$ 符号有多余的空格，如 $y \$$ ，编译 PDF 时会出错。

bookdown 0.6 版生成的 gitbook 格式的网页书籍，在有数学公式时，使用 MathJax 库在浏览器中显示数学公式。MathJax 是用于网络浏览器中显示数学公式的优秀的 Javascript 程序库，可免费使用。但是，当数学公式中含有中文（用 `\text{}` 或 `\mbox{}` 命令）时，数学公式可能会显示不正常。另外，数学公式默认使用远程服务器上的 MathJax 程序库处理，在网络不通畅时显示很慢或者无法显示。

MathJax 有多种渲染输出选择，gitbook 的模板中已经固化了一种 CommonHTML，这种输出与 gitbook 的 `style.css` 配合，当中文公式在 `section` 标记内时，使得中文显示不正常。如果不修改 gitbook 的模板，通过在 `_output.yml` 设置文件中插入设置命令的办法不奏效，因为 gitbook 的模板是动态调入 MathJax 库的，不能在 HTML 文件头或尾插入静态设置命令修改输出格式。

为此，直接修改 bookdown 包中的 gitbook.html 模板，删去文件末尾动态调入 MathJax 的部分，改为在 `_output.yml` 中要求调入一个设置文件。为了避免远程调用 MathJax 程序库的麻烦，改为本地使用。将 MathJax 安装在了书生成的网站主目录的上三层，用 `../../../MathJax/mathjax.js` 路径访问。假设下载整个 MathJax 库后解压放在了书的网页文件所在子目录的上三层的位置。

增加设置文件 `mathjax-local.html`:

```
<script type="text/x-mathjax-config">
MathJax.Hub.Config({
  jax: ["input/TeX","output/HTML-CSS"],
  extensions: ["tex2jax.js","MathMenu.js","MathZoom.js"],
  TeX: {
    extensions: ["AMSmath.js","AMSsymbols.js","noErrors.js","noUndefined.js"]
  }
});
</script>
<script type="text/javascript"
  src="../../../MathJax/MathJax.js">
</script>
```

如果希望用远程服务器上的 MathJax，可以使用如下的 `mathjax-cdnjs.html` 设置文件:

```
<script type="text/x-mathjax-config">
MathJax.Hub.Config({
  jax: ["input/TeX","output/HTML-CSS"],
  extensions: ["tex2jax.js","MathMenu.js","MathZoom.js"],
  TeX: {
    extensions: ["AMSmath.js","AMSsymbols.js","noErrors.js","noUndefined.js"]
  }
});
```

```

    }
  });
</script>
<script type="text/javascript"
  src="https://cdnjs.cloudflare.com/ajax/libs/mathjax/2.7.2/MathJax.js">
</script>

```

为了调用这样的设置文件，在 `_output.yml` 设置文件中如下设置：

```

bookdown::gitbook:
  css: style.css
  includes:
    in_header: mathjax-local.html

```

注意，MathJax 要求先设置再调入。由于浏览器对 MathJax 输出方式有记忆，所以如果不能正常显示中文公式，需要右键单击公式，选择“Math Settings—Math Renderer”中的“SVG”或者“HTML-CSS”。

22.12 使用经验

如果已经有用 LaTeX 写的书，要转换为 bookdown 的 Rmd 格式，可以用 RStudio 的支持 RegEx 的替换模式，如

- `\\keyword{([~]+?)}` 替换成 `**\1**`。
- `\\textbf{([~]+?)}` 替换成 `**\1**`。
- `\\texttt{([~]+?)}` 替换成 `\1`。

可以写一个函数对 LaTeX 文件进行转换生成 Rmd 文件，再手工修改。函数如

```

latex2rmd <- function(fname="tmpfiles/tomd.tex"){
  lines <- readLines(fname, encoding="UTF-8")
  print(head(lines, 10))
  lines <- gsub("\\\\textbf{([~]+?)}", "**\\1**", lines, perl=TRUE)
  lines <- gsub("\\\\texttt{([~]+?)}", "\\1", lines, perl=TRUE)
  lines <- gsub("\\\\label{([~]+?)}", "(\\1)", lines, perl=TRUE)
  lines <- gsub("\\\\eqref{([~]+?)}", "@ref(\\1)", lines, perl=TRUE)
  lines <- gsub("\\\\(S)(\\d)", "$\\2", lines, perl=TRUE)
  lines <- gsub("\\\\(S) ", "$ ", lines, perl=TRUE)
  lines <- gsub("\\\\argmin_", "\\mathop{\\text{argmin}}_", lines, perl=TRUE)
  print(head(lines, 20))
  writeLines(lines, "tmpfiles/fromtex-clean.Rmd", useBytes=TRUE)
}
latex2rmd()

```

Rmd 格式对算法编排的支持不够好。编排算法可以用表格来换行，仍用`\qqquad`和`\qquad`缩进，不要用空格缩进，因为在 LaTeX 转 PDF 时空格会损失。但是算法内容中有公式含有竖线时还是无法与表格线区分开来。

编排算法也可以用数学公式，写在 `$$` 界定范围内，用 `aligned` 环境分行，缩进使用`\quad`和`\qquad`。只是无法对 `if` 这样的关键字用重体排印。

为了能够生成中文的 PDF，不要指定 `doclass` 为 `ctexbook`，而是指定为 `book`，然后在 `preamble.tex` 中引入 `ctex` 包。

为了 PDF 输出，不要引入太多的数学包，因为从 markdown 到 HTML 不支持复杂的数学。

用 R 作图时如果图形中有汉字，在代码块选项中加上 `dev="png"`，`dpi=300`。否则生成 PDF 时会有中文编码问题。另一办法是在每个 Rmd 文件开头的 `setup` 源代码段插入

```
pdf.options(family="GB1")
```

这样可以生成支持中文字的 PDF 图形。

在编译出错时，会在主目录留下编译的 `tex` 源文件记相关文件。但是，此 `tex` 源文件中使用的 R 生成的图片路径不对，需要将 `tex` 源文件复制到 `bookdown_files` 目录，将直接插入的图片也复制到这个目录，然后编译 `tex` 文件发现问题，逐个修复。建议建立小的测试项目专门调试有问题的文件。

连分数的加号是`\genfrac{}{}{0pt}{}{}{+}`。

22.13 bookdown 的一些使用问题

在数学公式中用`\text{\@ref{eq:label}}`引用公式，HTML 成功，LaTeX 版本会有 BUG，重复了`\`。如果使用文内的链接，则 LaTeX 成功，HTML 不成功。

YAML 部分有的中文文字会出错，代以 ASCII 则不出错。

example 的自动编号有问题，不加 `label` 的 example，其 HTML 结果在不同章之间会编号混淆，避免问题的临时办法是 example 都加上 `label`。

Chapter 23

用 R Markdown 制作简易网站

23.1 介绍

为了从多个.Rmd 源文件制作网站，可以使用 blogdown 扩展包或者 bookdown 扩展包。bookdown 扩展包可以生成 gitbook 格式的网站，带有左侧的章节目录和前后页面的导航链接，很适用于一本书的网站。但是，bookdown 使用时需要重新编译整个网站才能产生正确的链接。

一种简易的制作网站的办法是仅仅利用 rmarkdown 包的 `render.site()` 功能。为了使得 rmarkdown 和 RStudio 将一个子目录（项目）看成一个网站项目，只要此项目中含有 `index.Rmd` 和 `_site.yml` 文件。

`index.Rmd` 是主页内容，可以在此处人工加入其它页面链接（参见 markdown 格式说明中链接写法），也可以制作单独的目录页面。内容如

```
---
title: "概率论"
---

* [概率分布](dist.html)
* [期望](expect.html)
* [极限定理](limits.html)
```

`_site.yml` 是一个 YAML 文件，其中包含站点的设定和输出设定。内容如

```
name: "概率论"
output_dir: "_probbook"
output:
  html_document:
    toc: true
    mathjax: "../..../MathJax/MathJax.js"
    self_contained: false
```

```

includes:
  in_header: "_header.html"
navbar:
  title: "R语言教程（草稿）"
  left:
    - text: "Home"
      href: index.html
    - text: "Contents"
      href: contents.html
    - text: "About"
      href: about.html

```

这里 `name` 设置站点名称, `output_dir` 给出生成的 html 及其他辅助文件的存放目录, 缺省时用 `_site` 子目录。navbar 域设置了网站的菜单, 这里包括 Home(主页), Contents(目录), About(关于) 三个页面的导航菜单。output 域设定输出的选项, 其中 `html_document` 就是用 `rmarkdown::render_site()` 生成的网站的输出文件格式, 其中 `toc: true` 说明每个页面都有自身内容的目录, 关于 `mathjax`, `self_contained`, `includes` 都与数学公式设置有关, 这里设置数学公式使用与本网站在同一目录系统或同一网站地址存放的 MathJax 数学公式显示库, 该库放在生成的 HTML 文件所在目录向上三层的 MathJax 子目录中。`in_header` 指定一个要插入到生成的每个 HTML 的 head 部分的内容文件, 这里内容文件 `_header.html` 的内容是

```

<script type="text/x-mathjax-config">
MathJax.Hub.Config({
  jax: ["input/TeX","output/SVG"],
  extensions: ["tex2jax.js","MathMenu.js","MathZoom.js"],
  TeX: {
    extensions: ["AMSmath.js","AMSsymbols.js","noErrors.js","noUndefined.js"]
  }
});
</script>

```

是对 MathJax 的一些设置, 主要使用 LaTeX 作为输入格式。

具有 `index.Rmd` 和 `_site.yml` 文件的项目, 在 RStudio 软件中会显示一个 Build 窗格, 点击其中的 Build Website 可以自动调用 `rmarkdown::render_site()` 生成整个站点, 存放在 `output_dir` 指定的输出目录中。为了将生成的站点发布到网站, 只要将输出目录的所有内容全部复制到服务器的某个子目录中就可以了。

`rmarkdown::render_site()` 生成整个站点时会自动逐个编译.Rmd 文件, 并将项目目录中其它的文件和子目录复制到输出目录中, 以句点或者下划线开头的文件和子目录不复制, .R, .r, .Rmd 文件不复制。

因为是自动依次编译所有的.Rmd 文件, 所以中间某个源文件编译错误就会使得整个编译失败, 修正错误后还是需要再次编译所有文件, 这一点不太方便。但是, `rmarkdown::render_site()` 提供了这样一种操作模

式：将有错的文件先移动到其它目录中，编译整个站点，这时生成的站点仅包含确认无误的各个页面；然后，再将可能有错的文件移动回到项目目录，逐个地打开每个未成功编译的文件，用 `Knit` 按钮单独编译，编译成功后结果文件也自动进入输出目录。这是用 `rmarkdown::render_site()` 制作网站比用 `bookdown` 扩展包制作网站的一个优点，`rmarkdown::render_site()` 可以人工控制一个一个地编译 `Rmd` 文件，编译成功一个就存放到输出目录中一个；因为网站的目录是自己编写的链接，所以这样逐个编译不会破坏网站的链接。编译单个文件的命令如

```
rmarkdown::render("sec1.Rmd", output_format = "html_document", encoding="UTF-8")
```

如果用 `bookdown` 包，只有编译所有文件才能生成正确的网站目录，`bookdown` 也能编译单个文件，但是仅能作为调试，调试成功后还是需要编译所有文件才能使得网站目录正确。

`bookdown` 站点的优势在于自动生成网站目录而不需要人工管理目录，而且 `bookdown` 支持图书的公式、定理、图标自动编号和链接，文献列表和文献引用，所以 `bookdown` 是比 `rmarkdown::render_site()` 编译调试速度比较慢但是功能更强大的一种制作网站的工具，而且还可以生成 PDF 版图书。

所以，一本书如果不需要很多的公式自动编号、图表编号、文献，在编写阶段可以使用 `rmarkdown::render_site()` 制作，定稿后再改成 `bookdown` 图书格式，实际上每个源文件内容部分不需要改变，只需要修改一下头部就可以了。站点的设置需要将 `_site.yml` 中的 `site` 域的值改为 `bookdown::bookdown_site`，并增加 `_bookdown.yml` 和 `_output.yml` 两个设置文件。

Chapter 24

制作幻灯片

24.1 介绍

R Markdown 文件 (.Rmd) 文件支持多种输出, 如网页 (html_document)、MS Word(word_document)、PDF(pdf_document, 需要 LaTeX 编译器支持) 等, 还支持生成网页格式的幻灯片 (ioslides_presentation, slidy_presentation), 以及 LaTeX beamer 格式的 PDF 幻灯片 (beamer_presentation)。

其中 slidy_presentation 生成网页格式的幻灯片, 并具有缩放字体大小、显示幻灯片目录等功能。只要在.Rmd 文件开头的 YAML 元数据部分指定 `output: slidy_presentation`。

幻灯片分为多个页面 (屏), 每页用二级标题作为标志并以其为标题。二级标题就是行首以两个井号和空格开始的行。用一级标题作为单独的分节页面, 将单独显示在一个页面中。

页面也可以没有标题, 比如仅有照片的页面, 这时, 用三个或三个以上的减号连在一起标志新页面的开始。

幻灯片用 RStudio 的 knit 按钮编译, 选择输出格式为 `slidy_presentation`, 结果在浏览器中播放, 最好使用外部浏览器而不使用 RStudio 自带的浏览器。除了使用 knit 按钮, 还可以用类似如下命令:

```
rmarkdown::render("mydemo.Rmd", output_format = "slidy_presentation", encoding="UTF-8")
```

其中 `mydemo.Rmd` 是源文件。

播放时, 用如下方式控制:

- 鼠标左键单击、光标右移键、翻页键向下键空格键都可以翻到下一页;
- 光标左移键、翻页键向上键回退一页;
- 单击下方的 Contents 或单击 C 键显示幻灯片目录列表, 可单击转移到任意页面;
- Home 键回到幻灯片开头;
- 用 A 键切换是否将所有页面合并成一个长的网页, 这样便于打印或者转存为 PDF, 但是打印生成的 PDF 仍是每页仅有原来的一个显示页面;
- 用 S 键缩写字体, 用 B 键放大字体。

为了制作幻灯片，最好单独设置一个 RStudio 项目，并且此项目仅生成幻灯片，而不生成普通网页、Word、PDF 等输出，否则可能造成结果混乱。希望 rmarkdown 包的后续版本能取消这个限制。

一个简单的 `slidy_presentation` 幻灯片源文件 `example-slidy.Rmd`，内容如：

```
---
title: "R幻灯片演示样例"
author: "李东风"
date: "2017-11-16"
output: slidy_presentation
---

# 用R Markdown的slidy输出作幻灯片

## 幻灯片结构

- 用二级标题标志一个页面开始
- 用一级标题制作单独的分节页面
- 用三个或三个以上减号标志没有标题的页面开始
- 每个页面一般用markdown列表显示若干个项目

## 幻灯片编译

- 用RStudio编辑
- 用RStudio的Knit按钮，选`slidy_presentation`作为输出格式

-----

[一个演示画面](figs/demoscreen.png)
```

24.2 数学公式处理

讲课用的幻灯片经常会有数学公式，比较关键的问题是数学公式如何处理。网页中的数学公式一般使用一个公开的自由 Javascript 库 MathJax 显示，但是这个库很大，如果使用远程的库，在网络不畅通时显示公式就不正常。更好的办法是使用局部的 MathJax 库或将 MathJax 库安装在临近的网站服务器上。

为了使用局部的 MathJax 库，简单的办法是在 YAML 的 `ioslides_presentation` 项目下面指定 `mathjax: local`，如：

```
title: "R幻灯片演示样例"
output:
```

```
slidy_presentation:
  mathjax: local
```

上述办法容易使用，缺点是多个不同的演示项目无法共用一个局部的 MathJax 库，生成的结果包含了许多小的支持文件。

为此，可以将 MathJax 库装在演示项目所在目录的上层（比如上三层的 MathJax 目录内），将设置 MathJax 的代码放在 `_header.html` 文件中，`_header.html` 中的内容如：

```
<script type="text/x-mathjax-config">
MathJax.Hub.Config({
  jax: ["input/TeX","output/SVG"],
  extensions: ["tex2jax.js","MathMenu.js","MathZoom.js"],
  TeX: {
    extensions: ["AMSmath.js","AMSsymbols.js","noErrors.js","noUndefined.js"]
  }
});
</script>
```

在演示项目所在目录中增加一个 `_output.yml` 文件，这是该项目所有输出的共用的输出设置，内容如：

```
slidy_presentation:
  mathjax: "../../../MathJax/MathJax.js"
  self_contained: false
  includes:
    in_header: "_header.html"
```

这里关闭了 `self_contained` 选项，设置了 MathJax 库在本地目录，具体是演示项目所在目录上面三层的 MathJax 目录中。

公式中如果有中文，开始时可能显示不正常，这时右键点击公式弹出菜单选择“Math Settings–Math Renderer”，取为“HTML-CSS”或“SVG”应可解决问题。

24.3 slidy 幻灯片激光笔失效问题的修改

slidy 幻灯片翻页是用空格、左右光标、上下翻页键，而一般激光笔翻页是模拟上下光标键。为此，在安装的 R 软件目录的 `library/rmarkdown/rmd/slidy/Slidy2/scripts` 子目录中，找到 `slidy.js` 文件，用编辑器打开，用编辑器的搜索功能搜索 `key == 37`，将其替换成 `key == 37 || key == 38`，这里 37 是向左光标的编码，替换后就是向左或者向上光标。用编辑器的搜索功能搜索 `key == 39`，将其替换成 `key == 39 || key == 40`，这里 39 是向右光标的编码，替换后就是向右或者向上光标。修改完毕后保存，然后将 `slidy.js` 用文件压缩程序（如 7zip）压缩为 `slidy.js.gz`。这样就可以在用 rmarkdown 制作的 `slidy_presentation` 结果中支持激光笔翻页了。

24.4 R Presentation 格式文件

R Studio 软件单独提供了对一种 R Presentation 格式的源文件的支持，以.Rpres 扩展名结尾，是一种特殊的 R Markdown 文件，与 slidy 的源文件也类似。

Rpres 文件编译为 HTML 格式的幻灯片，使用 reveal.js 控制显示。reveal.js 中也有对激光笔支持不好的问题，这是因为 reveal.js 中用向右光标键翻页，对向下光标另有定义，激光笔一般是模拟向右和向左光标键来翻页的。为了支持激光笔，找到 RStudio 的安装目录，在 `resources/presentation/revealjs/js` 中找到 `reveal.js` 文件，在文件编辑器中打开，通过搜索找到 `case 38:`，将其剪切到 `case 33:` 后面，变成 `case 33: case 38:`。找到 `case 40:`，将其剪切到 `case 34:` 后面，变成 `case 34: case 40:`。同一目录还有一个 `reveal.min.js` 文件，也进行上述修改。

Part V

R 数据处理

Chapter 25

数据读取技巧

25.1 日期数据

设文件 “dates.csv” 中包含如下内容，并设其文件编码为 GBK:

序号,出生日期,发病日期

```
1,1941/3/8,2007/1/1
2,1972/1/24,2007/1/1
3,1932/6/1,2007/1/1
4,1947/5/17,2007/1/1
5,1943/3/10,2007/1/1
6,1940/1/8,2007/1/1
7,1947/8/5,2007/1/1
8,2005/4/14,2007/1/1
9,1961/6/23,2007/1/2
10,1949/1/10,2007/1/2
```

先把日期当作字符串读入:

```
d.dates <- read_csv('dates.csv', locale=locale(encoding="GBK"))
```

```
## Parsed with column specification:
## cols(
##   序号 = col_integer(),
##   出生日期 = col_character(),
##   发病日期 = col_character()
## )
```

然后用 `as.POSIXct` 函数转换为 R 日期类型:

```
d.dates[[" 出生日期 ct"]] <- as.POSIXct(
  d.dates[[" 出生日期"]], format='%Y/%m/%d', tz='Etc/GMT+8')
d.dates[[" 发病日期 ct"]] <- as.POSIXct(
  d.dates[[" 发病日期"]], format='%Y/%m/%d', tz='Etc/GMT+8')
```

经过转换后的数据为:

```
knitr::kable(d.dates)
```

| 序号 | 出生日期 | 发病日期 | 出生日期 ct | 发病日期 ct |
|----|-----------|----------|------------|------------|
| 1 | 1941/3/8 | 2007/1/1 | 1941-03-08 | 2007-01-01 |
| 2 | 1972/1/24 | 2007/1/1 | 1972-01-24 | 2007-01-01 |
| 3 | 1932/6/1 | 2007/1/1 | 1932-06-01 | 2007-01-01 |
| 4 | 1947/5/17 | 2007/1/1 | 1947-05-17 | 2007-01-01 |
| 5 | 1943/3/10 | 2007/1/1 | 1943-03-10 | 2007-01-01 |
| 6 | 1940/1/8 | 2007/1/1 | 1940-01-08 | 2007-01-01 |
| 7 | 1947/8/5 | 2007/1/1 | 1947-08-05 | 2007-01-01 |
| 8 | 2005/4/14 | 2007/1/1 | 2005-04-14 | 2007-01-01 |
| 9 | 1961/6/23 | 2007/1/2 | 1961-06-23 | 2007-01-02 |
| 10 | 1949/1/10 | 2007/1/2 | 1949-01-10 | 2007-01-02 |

以上读入日期是比较保险的做法。还可以直接在 `read_csv()` 函数中指定某列为 `col_date()`:

```
d.dates <- read_csv(
  'dates.csv', locale=locale(encoding="GBK"),
  col_types=cols(
    `序号`=col_integer(),
    `出生日期`=col_date(format="%Y/%m/%d"),
    `发病日期`=col_date(format="%Y/%m/%d")
  ))
print(d.dates)
```

```
## # A tibble: 10 x 3
##   序号 出生日期 发病日期
##   <int> <date>   <date>
## 1     1 1941-03-08 2007-01-01
## 2     2 1972-01-24 2007-01-01
## 3     3 1932-06-01 2007-01-01
## 4     4 1947-05-17 2007-01-01
## 5     5 1943-03-10 2007-01-01
## 6     6 1940-01-08 2007-01-01
## 7     7 1947-08-05 2007-01-01
```

```
## 8      8 2005-04-14 2007-01-01
## 9      9 1961-06-23 2007-01-02
## 10     10 1949-01-10 2007-01-02
```

25.1.1 日期差计算

R 的日期可以用 `difftime` 计算差值。为了计算发病时的年龄，包括小数部分，可以这样计算：

```
d.dates[, '发病年龄（带小数年）'] <- as.numeric(
  difftime(d.dates[["发病日期"]], d.dates[["出生日期"]], units='days')/365.25)
knitr::kable(d.dates, digits=2)
```

| 序号 | 出生日期 | 发病日期 | 发病年龄（带小数年） |
|----|------------|------------|------------|
| 1 | 1941-03-08 | 2007-01-01 | 65.82 |
| 2 | 1972-01-24 | 2007-01-01 | 34.94 |
| 3 | 1932-06-01 | 2007-01-01 | 74.58 |
| 4 | 1947-05-17 | 2007-01-01 | 59.63 |
| 5 | 1943-03-10 | 2007-01-01 | 63.81 |
| 6 | 1940-01-08 | 2007-01-01 | 66.98 |
| 7 | 1947-08-05 | 2007-01-01 | 59.41 |
| 8 | 2005-04-14 | 2007-01-01 | 1.72 |
| 9 | 1961-06-23 | 2007-01-02 | 45.53 |
| 10 | 1949-01-10 | 2007-01-02 | 57.98 |

25.1.2 计算周岁

如果按照我们通常计算周岁的方法计算年龄，算法就不仅包括年的差，还要判断是否到了本年的生日。用函数实现周岁计算如下：

```
age.int <- function(birth, now){
  date1 <- as.POSIXlt(birth)
  date2 <- as.POSIXlt(now)
  age <- date2$year - date1$year
  sele <- (date2$mon * 100 + date2$mday
           < date1$mon * 100 + date1$mday)
  ## sele 是那些没有到生日的人
  age[sele] <- age[sele] - 1

  age
}
```

用 `d.dates()` 计算发病时周岁年龄:

```
d.dates[[" 发病年龄"]] <- age.int(d.dates[[" 出生日期"]], d.dates[[" 发病日期"]])
knitr::kable(d.dates, digits=2)
```

| 序号 | 出生日期 | 发病日期 | 发病年龄（带小数年） | 发病年龄 |
|----|------------|------------|------------|------|
| 1 | 1941-03-08 | 2007-01-01 | 65.82 | 65 |
| 2 | 1972-01-24 | 2007-01-01 | 34.94 | 34 |
| 3 | 1932-06-01 | 2007-01-01 | 74.58 | 74 |
| 4 | 1947-05-17 | 2007-01-01 | 59.63 | 59 |
| 5 | 1943-03-10 | 2007-01-01 | 63.81 | 63 |
| 6 | 1940-01-08 | 2007-01-01 | 66.98 | 66 |
| 7 | 1947-08-05 | 2007-01-01 | 59.41 | 59 |
| 8 | 2005-04-14 | 2007-01-01 | 1.72 | 1 |
| 9 | 1961-06-23 | 2007-01-02 | 45.53 | 45 |
| 10 | 1949-01-10 | 2007-01-02 | 57.98 | 57 |

25.2 缺失值处理

设有如下的“bp.csv”数据，以 GBK 编码保存:

序号,收缩压

1,145

5,110

6,未测

9,150

10,拒绝

15,115

其中的血压有非数值内容。直接用 `read.csv` 读入:

```
d.bp <- read_csv('bp.csv', locale=locale(encoding="GBK"))
```

```
## Parsed with column specification:
```

```
## cols(
```

```
##   序号 = col_integer(),
```

```
##   收缩压 = col_character()
```

```
## )
```

```
print(d.bp)
```

```
## # A tibble: 6 x 2
```

```
##   序号 收缩压
```

```
##      <int> <chr>
## 1      1 145
## 2      5 110
## 3      6 未测
## 4      9 150
## 5     10 拒绝
## 6     15 115
```

读入的收缩压被当成了字符型列，无法进行计算。

把字符型的收缩压转换为数值型：

```
d.bp[[" 收缩压数值"]] <- as.numeric(d.bp[[" 收缩压"]])
```

```
## Warning: 强制改变过程中产生了NA
```

```
knitr::kable(d.bp)
```

| 序号 | 收缩压 | 收缩压数值 |
|----|-----|-------|
| 1 | 145 | 145 |
| 5 | 110 | 110 |
| 6 | 未测 | NA |
| 9 | 150 | 150 |
| 10 | 拒绝 | NA |
| 15 | 115 | 115 |

收缩压中非数值的项被转换为数值型缺失值，并在转换时有警告信息。注意这里同时保存了原始输入的收缩压和转换为数值的收缩压，这样便于数据核对。

25.3 练习

- 读入 patients.csv，把其中的日期转换为 R 日期类型，计算发病年龄。保存为 tibble 数据框 d.patients。
- 读入 cancer.csv，保存为 d.cancer。v0 是放疗前肿瘤体积，v1 是放疗后肿瘤体积。计算放疗后肿瘤缩减率。

Chapter 26

数据整理

26.1 tidyverse 系统

假设数据以 tibble 格式保存（tibble 是数据框类型的改进，readr 包的 `read_csv()` 会生成此类）。数据集经常需要选行子集、选列子集、排序、定义新变量、横向合并等操作，而且经常会用若干个连续的操作分步处理，magrittr 包的管道运算符 `%>%` 特别适用于这种分步处理。dplyr 包和 tidyr 包定义了一系列“动词”，可以用比较自然的方式进行数据整理。

为了使用这些功能，可以载入 tidyverse 包，则 magrittr 包，readr 包，dplyr 包和 tidyr 包都会被自动载入：

```
library(tidyverse)
```

下面的例子中用如下的一个班的学生数据作为例子，保存在如下 class.csv 文件中：

```
name,sex,age,height,weight
Alice,F,13,56.5,84
Becka,F,13,65.3,98
Gail,F,14,64.3,90
Karen,F,12,56.3,77
Kathy,F,12,59.8,84.5
Mary,F,15,66.5,112
Sandy,F,11,51.3,50.5
Sharon,F,15,62.5,112.5
Tammy,F,14,62.8,102.5
Alfred,M,14,69,112.5
Duke,M,14,63.5,102.5
Guido,M,15,67,133
James,M,12,57.3,83
```

```

Jeffrey,M,13,62.5,84
John,M,12,59,99.5
Philip,M,16,72,150
Robert,M,12,64.8,128
Thomas,M,11,57.5,85
William,M,15,66.5,112

```

读入为 tibble:

```

d.class <- read_csv(
  "class.csv",
  col_types=cols(
    .default = col_double(),
    name=col_character(),
    sex=col_factor(levels=c("M", "F"))
  )
)

```

26.2 用 filter() 选择行子集

数据框的任何行子集仍为数据框，即使只有一行而且都是数值也是如此。行子集可以用行下标选取，如 `d.class[8:12,]`。函数 `head()` 取出数据框的前面若干行，`tail()` 取出数据框的最后若干行。

`dplyr` 包的 `filter()` 函数可以按条件选出符合条件的行组成的子集。下例从 `d.class` 中选出年龄在 13 岁和 13 岁以下的女生：

```

d.class %>%
  filter(sex=="F", age<=13)

```

```

## # A tibble: 5 x 5
##   name  sex    age height weight
##   <chr> <fct> <dbl>   <dbl>   <dbl>
## 1 Alice F      13.0    56.5    84.0
## 2 Becka F      13.0    65.3    98.0
## 3 Karen F      12.0    56.3    77.0
## 4 Kathy F      12.0    59.8    84.5
## 5 Sandy F      11.0    51.3    50.5

```

`filter()` 函数第一个参数是要选择的数据框，后续的参数是条件，这些条件是需要同时满足的，另外，条件中取缺失值的观测自动放弃。`filter()` 会自动舍弃行名，如果需要行名只能将其转换成数据框的一列。`filter()` 的结果为行子集数据框。用在管道操作当中的时候第一自变量省略（是管道传递下来的）。

26.3 用 `select()` 选择列子集

`dplyr` 包的 `select()` 选择列子集，并返回列子集结果。

可以指定变量名（不用写成字符串形式），如

```
d.class %>%  
  select(name, age)
```

```
## # A tibble: 19 x 2  
##   name      age  
##   <chr>   <dbl>  
## 1 Alice    13.0  
## 2 Becka    13.0  
## 3 Gail     14.0  
## 4 Karen    12.0  
## 5 Kathy    12.0  
## 6 Mary     15.0  
## 7 Sandy    11.0  
## 8 Sharon   15.0  
## 9 Tammy    14.0  
## 10 Alfred  14.0  
## 11 Duke     14.0  
## 12 Guido   15.0  
## 13 James   12.0  
## 14 Jeffrey 13.0  
## 15 John     12.0  
## 16 Philip  16.0  
## 17 Robert  12.0  
## 18 Thomas  11.0  
## 19 William 15.0
```

可以用冒号表示列范围，如

```
d.class %>%  
  select(age:weight)
```

```
## # A tibble: 19 x 3  
##   age height weight  
##   <dbl> <dbl> <dbl>  
## 1  13.0   56.5   84.0  
## 2  13.0   65.3   98.0  
## 3  14.0   64.3   90.0
```

```
## 4 12.0 56.3 77.0
## 5 12.0 59.8 84.5
## 6 15.0 66.5 112
## 7 11.0 51.3 50.5
## 8 15.0 62.5 112
## 9 14.0 62.8 102
## 10 14.0 69.0 112
## 11 14.0 63.5 102
## 12 15.0 67.0 133
## 13 12.0 57.3 83.0
## 14 13.0 62.5 84.0
## 15 12.0 59.0 99.5
## 16 16.0 72.0 150
## 17 12.0 64.8 128
## 18 11.0 57.5 85.0
## 19 15.0 66.5 112
```

参数中前面写负号表示扣除，如

```
d.class %>%
  select(-name, -age)
```

```
## # A tibble: 19 x 3
##   sex    height weight
##   <fct>   <dbl>   <dbl>
## 1 F      56.5    84.0
## 2 F      65.3    98.0
## 3 F      64.3    90.0
## 4 F      56.3    77.0
## 5 F      59.8    84.5
## 6 F      66.5   112
## 7 F      51.3    50.5
## 8 F      62.5   112
## 9 F      62.8   102
## 10 M      69.0   112
## 11 M      63.5   102
## 12 M      67.0   133
## 13 M      57.3    83.0
## 14 M      62.5    84.0
## 15 M      59.0    99.5
## 16 M      72.0   150
```

```
## 17 M      64.8  128
## 18 M      57.5   85.0
## 19 M      66.5  112
```

如果要选择的变量名已经保存为一个字符型向量，可以用 `one_of()` 函数引入，如

```
vars <- c("name", "sex")
d.class %>%
  select(one_of(vars))
```

```
## # A tibble: 19 x 2
##   name    sex
##   <chr>  <fct>
## 1 Alice  F
## 2 Becka  F
## 3 Gail   F
## 4 Karen  F
## 5 Kathy  F
## 6 Mary   F
## 7 Sandy  F
## 8 Sharon F
## 9 Tammy  F
## 10 Alfred M
## 11 Duke   M
## 12 Guido  M
## 13 James  M
## 14 Jeffrey M
## 15 John   M
## 16 Philip M
## 17 Robert M
## 18 Thomas M
## 19 William M
```

`select()` 有若干个配套函数可以按名字的模式选择变量列，如

- `starts_with("se")`: 选择名字以 “se” 开头的变量列;
- `ends_with("ght")`: 选择名字以 “ght” 结尾的变量列;
- `contains("no")`: 选择名字中含有子串 “no” 的变量列;
- `matches("^[[:alpha:]]+[[:digit:]]+$")`, 选择列名匹配某个正则表达式模式的变量列，这里匹配前一部分是字母，后一部分是数字的变量名。
- `num_range("x", 1:3)`, 选择 `x1`, `x2`, `x3`。
- `everything()`: 代指所有选中的变量，这可以用来将指定的变量次序提前，其它变量排在后面。

R 的字符串函数（如 `paste()`）和正则表达式函数可以用来生成变量名子集。

R 函数 `subset` 也能对数据框选取列子集和行子集。

26.4 用 `arrange()` 排序

`dplyr` 包的 `arrange()` 按照数据框的某一列或某几列排序，返回排序后的结果，如

```
d.class %>%  
  arrange(sex, age)
```

```
## # A tibble: 19 x 5  
##   name    sex    age height weight  
##   <chr>  <fct> <dbl> <dbl> <dbl>  
## 1 Thomas M      11.0  57.5  85.0  
## 2 James  M      12.0  57.3  83.0  
## 3 John   M      12.0  59.0  99.5  
## 4 Robert M      12.0  64.8  128  
## 5 Jeffrey M      13.0  62.5  84.0  
## 6 Alfred M      14.0  69.0  112  
## 7 Duke   M      14.0  63.5  102  
## 8 Guido  M      15.0  67.0  133  
## 9 William M      15.0  66.5  112  
## 10 Philip M      16.0  72.0  150  
## 11 Sandy  F      11.0  51.3  50.5  
## 12 Karen  F      12.0  56.3  77.0  
## 13 Kathy  F      12.0  59.8  84.5  
## 14 Alice  F      13.0  56.5  84.0  
## 15 Becka  F      13.0  65.3  98.0  
## 16 Gail   F      14.0  64.3  90.0  
## 17 Tammy  F      14.0  62.8  102  
## 18 Mary   F      15.0  66.5  112  
## 19 Sharon F      15.0  62.5  112
```

用 `desc()` 包裹想要降序排列的变量，如

```
d.class %>%  
  arrange(sex, desc(age))
```

```
## # A tibble: 19 x 5  
##   name    sex    age height weight  
##   <chr>  <fct> <dbl> <dbl> <dbl>
```

```
## 1 Philip M      16.0  72.0  150
## 2 Guido  M      15.0  67.0  133
## 3 William M     15.0  66.5  112
## 4 Alfred M     14.0  69.0  112
## 5 Duke    M     14.0  63.5  102
## 6 Jeffrey M    13.0  62.5   84.0
## 7 James   M    12.0  57.3   83.0
## 8 John    M    12.0  59.0   99.5
## 9 Robert  M    12.0  64.8  128
## 10 Thomas M    11.0  57.5   85.0
## 11 Mary   F     15.0  66.5  112
## 12 Sharon F     15.0  62.5  112
## 13 Gail   F     14.0  64.3   90.0
## 14 Tammy  F     14.0  62.8  102
## 15 Alice  F     13.0  56.5   84.0
## 16 Becka  F     13.0  65.3   98.0
## 17 Karen  F     12.0  56.3   77.0
## 18 Kathy  F     12.0  59.8   84.5
## 19 Sandy  F     11.0  51.3   50.5
```

排序时不论升序还是降序，所有的缺失值都自动排到末尾。

R 函数 `order()` 可以用来给出数据框的排序次序从而将数据框排序。

26.5 用 `rename()` 修改变量名

在 `dplyr` 包的 `rename()` 中用“新名字 = 旧名字”格式修改变量名，如

```
d2.class <- d.class %>%
  rename(h=height, w=weight)
```

注意这样改名字不是对原始数据框修改而是返回改了名字后的新数据框。

26.6 用 `mutate()` 计算新变量

`dplyr` 包的 `mutate()` 可以为数据框计算新变量，返回含有新变量以及原变量的数据框。如

```
d.class %>%
  mutate(
    rwh=weight/height,
    sexc=ifelse(sex=="F", " 女", " 男"))
```

```
## # A tibble: 19 x 7
##   name    sex    age height weight   rwh sexc
##   <chr>  <fct> <dbl>  <dbl>  <dbl> <dbl> <chr>
## 1 Alice  F      13.0   56.5   84.0  1.49  女
## 2 Becka  F      13.0   65.3   98.0  1.50  女
## 3 Gail   F      14.0   64.3   90.0  1.40  女
## 4 Karen  F      12.0   56.3   77.0  1.37  女
## 5 Kathy  F      12.0   59.8   84.5  1.41  女
## 6 Mary   F      15.0   66.5  112    1.68  女
## 7 Sandy  F      11.0   51.3   50.5  0.984 女
## 8 Sharon F      15.0   62.5  112    1.80  女
## 9 Tammy  F      14.0   62.8  102    1.63  女
## 10 Alfred M      14.0   69.0  112    1.63  男
## 11 Duke   M      14.0   63.5  102    1.61  男
## 12 Guido  M      15.0   67.0  133    1.99  男
## 13 James  M      12.0   57.3   83.0  1.45  男
## 14 Jeffrey M      13.0   62.5   84.0  1.34  男
## 15 John   M      12.0   59.0   99.5  1.69  男
## 16 Philip M      16.0   72.0  150    2.08  男
## 17 Robert M      12.0   64.8  128    1.98  男
## 18 Thomas M      11.0   57.5   85.0  1.48  男
## 19 William M      15.0   66.5  112    1.68  男
```

注意这样生成新变量不是在原来的数据框中添加，原来的数据框没有被修改，而是返回添加了新变量的新数据框。R 软件的巧妙设计保证了这样虽然是生成了新数据框，但是与原来数据框重复的列并不会重复保存。

新变量可以与老变量名相同，这样就在输出中修改了老变量。

函数 `transmute()` 用法与 `mutate()` 类似，但是不保留原始变量。

定义新变量也可以直接为数据框的新变量赋值：

```
d.class[["rwh"]] <- d.class[["weight"]] / d.class[["height"]]
```

这样的做法与 `mutate()` 的区别是这样不会生成新数据框，新变量是在原数据框中增加的。

给数据框中某个变量赋值为 `NULL` 可以修改数据框，从数据框中删去该变量。

26.7 用管道连接多次操作

管道运算符特别适用于对同一数据集进行多次操作。例如，对 `t.class` 数据，先选出所有女生，再去掉性别和 `age` 变量：


```
d.class %>%
  filter(sex=="F") %>%
  select(-sex, -age)

## # A tibble: 9 x 4
##   name    height weight   rwh
##   <chr>   <dbl>   <dbl> <dbl>
## 1 Alice    56.5    84.0  1.49
## 2 Becka    65.3    98.0  1.50
## 3 Gail     64.3    90.0  1.40
## 4 Karen    56.3    77.0  1.37
## 5 Kathy    59.8    84.5  1.41
## 6 Mary     66.5   112    1.68
## 7 Sandy    51.3    50.5  0.984
## 8 Sharon   62.5   112    1.80
## 9 Tammy    62.8   102    1.63
```

管道操作的结果可以保存为新的 tibble，如：

```
class_F <- d.class %>%
  filter(sex=="F") %>%
  select(-sex, -age)
```

26.8 数据简单汇总

dplyr 包的 `summarise()` 函数可以对数据框计算统计量。这个函数针对少量变量时很方便，有大量变量需
要对变量统一处理时不太方便。

以肺癌病人化疗数据为例，有 34 个肺癌病人的数据：

```
d.cancer <- read_csv(
  "cancer.csv", locale=locale(encoding="GBK"))

## Parsed with column specification:
## cols(
##   id = col_integer(),
##   age = col_integer(),
##   sex = col_character(),
##   type = col_character(),
##   v0 = col_double(),
##   v1 = col_double()
## )
```

```
d.cancer
```

```
## # A tibble: 34 x 6
##       id   age sex   type      v0      v1
##   <int> <int> <chr> <chr> <dbl> <dbl>
## 1     1     70 F     腺癌    26.5    2.91
## 2     2     70 F     腺癌   135    35.1
## 3     3     69 F     腺癌   210    74.4
## 4     4     68 M     腺癌    61.0   35.0
## 5     5     67 M     鳞癌   238    128
## 6     6     75 F     腺癌   330    112
## 7     7     52 M     鳞癌   105    32.1
## 8     8     71 M     鳞癌    85.2   29.2
## 9     9     68 M     鳞癌   102    22.2
## 10    10     79 M     鳞癌    65.5   21.9
## # ... with 24 more rows
```

求年龄 (age) 的平均值、标准差:

```
d.cancer %>%
  summarise(mean.age=mean(age, na.rm=TRUE),
            sd.age=sd(age, na.rm=TRUE))
```

```
## # A tibble: 1 x 2
##   mean.age sd.age
##   <dbl> <dbl>
## 1   64.1   9.16
```

更重要的是分组汇总。dplyr 包的 `group_by()` 函数对数据框 (或 tibble) 分组, 随后的 `summarise()` 将按照分组汇总。比如, 按不同性别计算人数与年龄平均值:

```
d.cancer %>%
  group_by(sex) %>%
  summarise(count=n(), mean.age=mean(age, na.rm=TRUE))
```

```
## # A tibble: 2 x 3
##   sex   count mean.age
##   <chr> <int>   <dbl>
## 1 F         13    66.1
## 2 M         21    63.2
```

其中 `n()` 计算某类的观测数 (行数)。为了计算某列的非缺失值个数, 用 `sum(!is.na(x))`。

常用的汇总函数有:

- 位置度量: `mean()`, `median()`。
- 分散程度 (变异性) 度量: `sd()`, `IQR()`, `mad()`。
- 分位数: `min()`, `max()`, `quantile()`。
- 按下标查询, 如 `first(x)` 取出 `x[1]`, `last(x)` 取出 `x` 的最后一个元素, `nth(x,2)` 取出 `x[2]`。可以提供一个缺省值以防某个下标位置不存在。
- 计数: `n()` 给出某个组的观测数, `sum(!is.na(x))` 统计 `x` 的非缺失值个数, `n_distinct(x)` 统计 `x` 的不同值个数 (缺失值也算一个值)。 `count(x)` 给出 `x` 的每个不同值的个数 (类似于 `table()` 函数)。

这里有些函数是 `dplyr` 包提供的, 仅适用于 `tibble` 类型。

下面的程序对 `d.cancer` 数据框分性别与病理类型分别统计人数:

```
d.cancer %>%
  group_by(sex, type) %>%
  summarise(freq=n())
```

```
## # A tibble: 4 x 3
## # Groups:   sex [?]
##   sex   type   freq
##   <chr> <chr> <int>
## 1 F     鳞癌     4
## 2 F     腺癌     9
## 3 M     鳞癌    18
## 4 M     腺癌     3
```

事实上, 不需要用 `group_by()`, 交叉分类计算频数可以用 `dplyr` 的 `count()` 函数, 如:

```
d.cancer %>%
  count(sex, type)
```

```
## # A tibble: 4 x 3
##   sex   type     n
##   <chr> <chr> <int>
## 1 F     鳞癌     4
## 2 F     腺癌     9
## 3 M     鳞癌    18
## 4 M     腺癌     3
```

用 `group_by()` 分组后除了可以分组汇总, 还可以分组筛选:

```
d.cancer %>%
  group_by(sex) %>%
  filter(rank(desc(v0)) <= 2) %>%
  arrange(sex, desc(v0))
```

```
## # A tibble: 4 x 6
## # Groups:   sex [2]
##      id   age sex   type      v0    v1
##   <int> <int> <chr> <chr> <dbl> <dbl>
## 1     6    75 F     腺癌    330  112
## 2    25    NA F     鳞癌    223  25.6
## 3     5    67 M     鳞癌    238  128
## 4    16    76 M     鳞癌    231  113
```

以上程序按性目标分组后，在每组中找出术前体积排名在前两名的。

在分组后也可以根据每组的统计量用 `mutate()` 定义新变量。

用 `group_by()` 分组汇总后的结果不是普通的 tibble，总是带有分组信息。这在后续的使用中可能会产生问题，为此，可以用 `ungroup()` 函数取消分组。例如

```
d.cancer %>%
  group_by(sex, type) %>%
  summarise(freq=n()) %>%
  summarise(ntotal=sum(freq))
```

```
## # A tibble: 2 x 2
##   sex   ntotal
##   <chr>   <int>
## 1 F         13
## 2 M         21
```

可以看出并没有能够通过男、女分别的人数计算总人数。加入 `ungroup()`：

```
d.cancer %>%
  group_by(sex, type) %>%
  summarise(freq=n()) %>%
  ungroup() %>%
  summarise(ntotal=sum(freq))
```

```
## # A tibble: 1 x 1
##   ntotal
##   <int>
## 1     34
```

得到了需要的结果。

26.9 长宽表转换

考虑如下的宽表，保存在 CSV 文件 “widetab.csv” 中：

```
"subject","x_1","x_2","x_3","y_1","y_2","y_3"
1,5,7,8,9,7,6
2,8,2,10,1,1,9
3,7,2,5,10,8,3
4,1,5,6,10,1,1
5,9,7,10,8,8,10
```

这个数据是 5 名病人 3 次检查的记录，每次检查有 x 和 y 两个测量项目。每个病人的所有各次检查以及所有测量项目都在同一行，称这样的表为宽表。读入为

```
d.wide <- read_csv("widetab.csv")

## Parsed with column specification:
## cols(
##   subject = col_integer(),
##   x_1 = col_integer(),
##   x_2 = col_integer(),
##   x_3 = col_integer(),
##   y_1 = col_integer(),
##   y_2 = col_integer(),
##   y_3 = col_integer()
## )
```

现在希望将其中的时间分离出来，不同时间变成不同的观测，每个病人的三次检查转换成三个观测。tidyr 包的 `gather()` 函数将宽表变成长表，但是将不同测量项目也转换到了不同的行：

```
d.wide %>%
  gather(x_1, x_2, x_3, y_1, y_2, y_3, key="variable", value="value")

## # A tibble: 30 x 3
##   subject variable value
##   <int> <chr>    <int>
## 1      1 x_1      5
## 2      2 x_1      8
## 3      3 x_1      7
## 4      4 x_1      1
## 5      5 x_1      9
## 6      1 x_2      7
## 7      2 x_2      2
```

```
## 8      3 x_2      2
## 9      4 x_2      5
## 10     5 x_2      7
## # ... with 20 more rows
```

tidyr 包的 `separate()` 可以用来帮助拆分 `x_1`, `y_1` 这样的名字, 这样可以将测量项目名称与时间分离开来, 如:

```
d.wide %>%
  gather(x_1, x_2, x_3, y_1, y_2, y_3, key="variable", value="value") %>%
  separate(variable, into=c("variable", "time"), sep="_")
```

```
## # A tibble: 30 x 4
##   subject variable time  value
## *   <int> <chr>    <chr> <int>
## 1     1 x      1      5
## 2     2 x      1      8
## 3     3 x      1      7
## 4     4 x      1      1
## 5     5 x      1      9
## 6     1 x      2      7
## 7     2 x      2      2
## 8     3 x      2      2
## 9     4 x      2      5
## 10    5 x      2      7
## # ... with 20 more rows
```

tidyr 包的函数 `spread()` 可以将用变量名和变量值分别存储的变量, 恢复为每个变量一列的形式, 如:

```
d.wide %>%
  gather(x_1, x_2, x_3, y_1, y_2, y_3, key="variable", value="value") %>%
  separate(variable, into=c("variable", "time"), sep="_") %>%
  spread(key=variable, value=value)
```

```
## # A tibble: 15 x 4
##   subject time      x      y
## *   <int> <chr> <int> <int>
## 1     1 1      5      9
## 2     1 2      7      7
## 3     1 3      8      6
## 4     2 1      8      1
## 5     2 2      2      1
## 6     2 3     10      9
```

```
## 7      3 1      7 10
## 8      3 2      2 8
## 9      3 3      5 3
## 10     4 1      1 10
## 11     4 2      5 1
## 12     4 3      6 1
## 13     5 1      9 8
## 14     5 2      7 8
## 15     5 3     10 10
```

这个结果已经变成每个病人每次检查为一个观测（记录）的形式。

如果变量名不像 `x_1`, `y_3` 这样整齐，为了拆分变量名与时间，也可以用字符串函数。比如，我们将 `d.wide` 中变量 `x_1` 改名为 `x1`, `x_3` 改名为 `x3`, `y_1` 改名为 `abc1`，删去其它变量：

```
d.wide %>%
  select(subject, x_1, x_3, y_1) %>%
  rename(x1=x_1, x3=x_3, abc1=y_1)
```

```
## # A tibble: 5 x 4
##   subject    x1    x3 abc1
##   <int> <int> <int> <int>
## 1     1     5     8     9
## 2     2     8    10     1
## 3     3     7     5    10
## 4     4     1     6    10
## 5     5     9    10     8
```

要将上述数据的变量与时间信息分离，因为没有变量名与时间之间没有明确的分隔符也没有固定宽度，`separate()` 函数难以分离这样的变量名与时间，可以利用正则表达式：

```
d.wide %>%
  select(subject, x_1, x_3, y_1) %>%
  rename(x1=x_1, x3=x_3, abc1=y_1) %>%
  gather(x1, x3, abc1, key="variable", value="value") %>%
  mutate(time=as.numeric(gsub("^[[:alpha:]]+([[:digit:]]+)$", "\\2", variable)),
         variable=gsub("^[[:alpha:]]+([[:digit:]]+)$", "\\1", variable)) %>%
  spread(key=variable, value=value)
```

```
## # A tibble: 10 x 4
##   subject time  abc    x
## *   <int> <dbl> <int> <int>
## 1     1  1.00     9     5
```

```
## 2      1 3.00    NA     8
## 3      2 1.00     1     8
## 4      2 3.00    NA    10
## 5      3 1.00    10     7
## 6      3 3.00    NA     5
## 7      4 1.00    10     1
## 8      4 3.00    NA     6
## 9      5 1.00     8     9
## 10     5 3.00    NA    10
```

tidyr 包还有一些方便函数。长宽表转换问题中的某些数据是缺失的，这些缺失值在某些形式中可见，如上表中变量 abc 在 time=3 时的值，在某些形式中根本没有表现出来。在用 `gather()` 将宽表变长表时，可以加 `na.rm=TRUE` 选项将不必要的缺失值观测删去。函数 `complete()` 可以指定若干列，使得这些列的所有不同组合均出现。有时某个单元格缺失是表示该值等于其上一行的值，这时可以用函数 `fill()` 指定该列按照这样的规则填充值。

26.10 拆分数数据列

有时应该放在不同列的数据用分隔符分隔后放在同一列中了。比如，下面数据集中 “succ/total” 列存放了用 “/” 分隔开的成功数与试验数：

```
d.sep <- read_csv(
  "testdata, succ/total
1, 1/10
2, 3/5
3, 2/8
")
d.sep
```

```
## # A tibble: 3 x 2
##   testid `succ/total`
##   <int> <chr>
## 1     1 1 1/10
## 2     2 2 3/5
## 3     3 3 2/8
```

用 `tidyr::separate()` 可以将这样的列拆分为各自的变量列，如

```
d.sep %>%
  separate(`succ/total`, into=c("succ", "total"),
    sep="/", convert=TRUE)
```



```
## # A tibble: 3 x 3
##   testid succ total
## *   <int> <int> <int>
## 1     1     1    10
## 2     2     3     5
## 3     3     2     8
```

其中 `into` 指定拆分后新变量名, `sep` 指定分隔符, `convert=TRUE` 要求自动将分割后的值转换为适当的类型。`sep` 还可以指定取子串的字符位置, 按位置拆分各个子串。

选项 `extra` 指出拆分时有多余内容的处理方法, 选项 `fill` 指出有不足内容的处理方法。

函数 `extract()` 可以按照某种正则表达式表示的模式从指定列拆分出对应于正则表达式中捕获组的一列或多列内容。

26.11 合并数据列

`tidyr::unite()` 函数可以将同一行的两列或多列的内容合并成一列。这是 `separate()` 的反向操作, 如:

```
d.sep %>%
  separate(`succ/total`, into=c("succ", "total"),
           sep="/", convert=TRUE) %>%
  unite(ratio, succ, total, sep=":")
```

```
## # A tibble: 3 x 2
##   testid ratio
## *   <int> <chr>
## 1     1 1 1:10
## 2     2 2 3:5
## 3     3 3 2:8
```

`unite()` 的第一个参数是要修改的数据框, 这里用管道 `%>%` 传递进来, 第二个参数是合并后的变量名 (`ratio` 变量), 其它参数是要合并的变量名, `sep` 指定分隔符。实际上用 `mutate()`、`paste()` 或者 `sprintf()` 也能完成合并。

26.12 横向合并

实际数据往往没有存放在单一的表中, 需要从多个表查找数据。多个表之间的连接, 一般靠关键列 (`key`) 对准来连接。连接可以是一对一的, 一对多的。多对多连接应用较少, 因为多对多连接是所有两两组合。

在规范的数据库中, 每个表都应该有主键, 这可以是一列, 也可以是多列的组合。为了确定某列是主键, 可以用 `count()` 和 `filter()`, 如

```
d.class %>%
  count(name) %>%
  filter(n>1)
```

```
## # A tibble: 0 x 2
## # ... with 2 variables: name <chr>, n <int>
```

没有发现重复出现的 `name`, 说明 `d.class` 中 `name` 可以作为主键。

为了演示一对一的横向连接, 我们将 `d.class` 拆分为两个数据集 `d1.class` 和 `d2.class`, 两个数据集都有主键 `name`, `d1.class` 包含变量 `name`, `sex`, `d2.class` 包含变量 `name`, `age`, `height`, `weight`, 并删去某些观测:

```
d1.class <- d.class %>%
  select(name, sex) %>%
  filter(!(name %in% "Becka"))
d2.class <- d.class %>%
  select(name, age, height, weight)
```

用 `dplyr` 包的 `inner_join()` 函数将两个数据框按键值横向合并, 仅保留能匹配的观测。因为 `d1.class` 中丢失了 `Becka` 的观测, 所以合并后的数据框中也没有 `Becka` 的观测:

```
d1.class %>%
  inner_join(d2.class)
```

```
## Joining, by = "name"
```

```
## # A tibble: 18 x 5
##   name      sex      age height weight
##   <chr>    <fct> <dbl>  <dbl>  <dbl>
## 1 Alice    F      13.0   56.5   84.0
## 2 Gail     F      14.0   64.3   90.0
## 3 Karen    F      12.0   56.3   77.0
## 4 Kathy    F      12.0   59.8   84.5
## 5 Mary     F      15.0   66.5  112
## 6 Sandy    F      11.0   51.3   50.5
## 7 Sharon   F      15.0   62.5  112
## 8 Tammy    F      14.0   62.8  102
## 9 Alfred   M      14.0   69.0  112
## 10 Duke     M      14.0   63.5  102
## 11 Guido    M      15.0   67.0  133
## 12 James    M      12.0   57.3   83.0
## 13 Jeffrey  M      13.0   62.5   84.0
## 14 John     M      12.0   59.0   99.5
## 15 Philip   M      16.0   72.0  150
```

```
## 16 Robert M      12.0  64.8  128
## 17 Thomas M      11.0  57.5  85.0
## 18 William M     15.0  66.5  112
```

横向连接自动找到了共同的变量 `name` 作为连接的键值，可以在 `inner_join()` 中用 `by=` 指定键值变量名，如果有不同的变量名，可以用 `by = c("a"="b")` 的格式指定左数据框的键值 `a` 与右数据框的键值 `b` 匹配进行连接。

两个表的横向连接，经常是多对一连接。例如，`d.stu` 中有学生学号、班级号、姓名、性别，`d.cl` 中有班级号、班主任名、年级，可以通过班级号将两个表连接起来：

```
d.stu <- tibble(
  sid=c(1,2,3,4,5,6),
  cid=c(1,2,1,2,1,2),
  sname=c("John", "Mary", "James", "Kitty", "Jasmine", "Kim"),
  sex=c("M", "F", "M", "F", "F", "M"))
d.stu
```

```
## # A tibble: 6 x 4
##   sid  cid sname  sex
##   <dbl> <dbl> <chr>  <chr>
## 1  1.00  1.00 John    M
## 2  2.00  2.00 Mary    F
## 3  3.00  1.00 James   M
## 4  4.00  2.00 Kitty   F
## 5  5.00  1.00 Jasmine F
## 6  6.00  2.00 Kim     M
```

```
d.cl <- tibble(
  cid=c(1,2),
  tname=c("Philip", "Joane"),
  grade=c("2017", "2016")
)
d.cl
```

```
## # A tibble: 2 x 3
##   cid tname  grade
##   <dbl> <chr>  <chr>
## 1  1.00 Philip 2017
## 2  2.00 Joane 2016
```

```
d.stu %>%
  left_join(d.cl, by="cid")
```

```
## # A tibble: 6 x 6
##   sid  cid sname  sex  tname  grade
##   <dbl> <dbl> <chr>  <chr> <chr>  <chr>
## 1  1.00  1.00 John    M    Philip 2017
## 2  2.00  2.00 Mary    F    Joane   2016
## 3  3.00  1.00 James   M    Philip 2017
## 4  4.00  2.00 Kitty   F    Joane   2016
## 5  5.00  1.00 Jasmine F    Philip 2017
## 6  6.00  2.00 Kim     M    Joane   2016
```

`left_join()` 按照 `by` 变量指定的关键列匹配观测，左数据集所有观测不论匹配与否全部保留，右数据集仅使用与左数据集能匹配的观测。不指定 `by` 变量时，使用左、右数据集的共同列作为关键列。如果左右数据集关键列变量名不同，可以用 `by=c("左名"="右名")` 的格式。

类似地，`right_join()` 保留右数据集的所有观测，而仅保留左数据集中能匹配的观测。`innre_join()` 仅保留能匹配的观测。

26.13 利用第二个数据集筛选

`left_join()` 将右表中与左表匹配的观测的额外的列添加到左表中。如果希望按照右表筛选左表的观测，可以用 `semi_join()`，函数 `anti_join()` 则是要求保留与右表不匹配的观测。

26.14 数据集的集合操作

R 的 `intersect()`, `union()`, `setdiff()` 本来是以向量作为集合进行集合操作。`dplyr` 包也提供了这些函数，但是将两个 `tibble` 的各行作为元素进行集合操作。

26.15 数据框纵向合并

矩阵或数据框要纵向合并，使用 `rbind` 函数即可。要求变量集合是相同的，变量次序可以不同。

比如，有如下两个分开男生、女生的数据框：

```
d3.class <- d.class %>%
  select(name, sex, age) %>%
  filter(sex=="M")
d4.class <- d.class %>%
  select(name, sex, age) %>%
  filter(sex=="F")
```

合并行如下:

```
rbind(d3.class, d4.class)
```

```
## # A tibble: 19 x 3
##   name    sex    age
##   <chr>  <fct> <dbl>
## 1 Alfred M      14.0
## 2 Duke   M      14.0
## 3 Guido  M      15.0
## 4 James  M      12.0
## 5 Jeffrey M      13.0
## 6 John   M      12.0
## 7 Philip M      16.0
## 8 Robert M      12.0
## 9 Thomas M      11.0
## 10 William M     15.0
## 11 Alice  F      13.0
## 12 Becka  F      13.0
## 13 Gail   F      14.0
## 14 Karen  F      12.0
## 15 Kathy  F      12.0
## 16 Mary   F      15.0
## 17 Sandy  F      11.0
## 18 Sharon F      15.0
## 19 Tammy  F      14.0
```

将下面的数据框的变量列次序打乱，合并不受影响:

```
rbind(d3.class, d4.class[, c("age", "name", "sex")])
```

```
## # A tibble: 19 x 3
##   name    sex    age
##   <chr>  <fct> <dbl>
## 1 Alfred M      14.0
## 2 Duke   M      14.0
## 3 Guido  M      15.0
## 4 James  M      12.0
## 5 Jeffrey M      13.0
## 6 John   M      12.0
## 7 Philip M      16.0
## 8 Robert M      12.0
```

```
## 9 Thomas M      11.0
## 10 William M     15.0
## 11 Alice F       13.0
## 12 Becka F       13.0
## 13 Gail F        14.0
## 14 Karen F       12.0
## 15 Kathy F       12.0
## 16 Mary F        15.0
## 17 Sandy F       11.0
## 18 Sharon F      15.0
## 19 Tammy F       14.0
```

26.16 标准化

设 x 是各列都为数值的列表 (包括数据框和 tibble) 或数值型矩阵, 用 `scale(x)` 可以把每一列都标准化, 即每一列都减去该列的平均值, 然后除以该列的样本标准差。用 `scale(x, center=TRUE, scale=FALSE)` 仅中心化而不标准化。如

```
d.class %>%
  select(height, weight) %>%
  scale()
```

```
##           height      weight
## [1,] -1.13843504 -0.70371312
## [2,]  0.57794313 -0.08897522
## [3,]  0.38290015 -0.44025402
## [4,] -1.17744363 -1.01108207
## [5,] -0.49479323 -0.68175819
## [6,]  0.81199469  0.52576268
## [7,] -2.15265850 -2.17469309
## [8,]  0.03182280  0.54771760
## [9,]  0.09033569  0.10861910
## [10,] 1.29960213  0.54771760
## [11,] 0.22686577  0.10861910
## [12,] 0.90951618  1.44786952
## [13,] -0.98240066 -0.74762297
## [14,]  0.03182280 -0.70371312
## [15,] -0.65082761 -0.02311045
## [16,]  1.88473105  2.19433697
## [17,]  0.48042164  1.22832027
```

```
## [18,] -0.94339207 -0.65980327
## [19,]  0.81199469  0.52576268
## attr("scaled:center")
##      height      weight
## 62.33684 100.02632
## attr("scaled:scale")
##      height      weight
##  5.127075 22.773933
```

为了把 `x` 的每列变到 `[0,1]` 内，可以用如下的方法：

```
d.class %>%
  select(height, weight) %>%
  scale(center=apply(., 2, min),
        scale=apply(., 2, max) - apply(., 2, min))
```

其中的 `.` 在管道操作中表示被传递处理的变量（一般是数据框）。也可以写一个自定义的进行零一标准化的函数：

```
scale01 <- function(x){
  mind <- apply(x, 2, min)
  maxd <- apply(x, 2, max)
  scale(x, center=mind, scale=maxd-mind)
}
d.class %>%
  select(height, weight) %>%
  scale01
```

```
##           height      weight
## [1,] 0.2512077 0.3366834
## [2,] 0.6763285 0.4773869
## [3,] 0.6280193 0.3969849
## [4,] 0.2415459 0.2663317
## [5,] 0.4106280 0.3417085
## [6,] 0.7342995 0.6180905
## [7,] 0.0000000 0.0000000
## [8,] 0.5410628 0.6231156
## [9,] 0.5555556 0.5226131
## [10,] 0.8550725 0.6231156
## [11,] 0.5893720 0.5226131
## [12,] 0.7584541 0.8291457
## [13,] 0.2898551 0.3266332
```

```
## [14,] 0.5410628 0.3366834
## [15,] 0.3719807 0.4924623
## [16,] 1.0000000 1.0000000
## [17,] 0.6521739 0.7788945
## [18,] 0.2995169 0.3467337
## [19,] 0.7342995 0.6180905
## attr("scaled:center")
## height weight
##    51.3    50.5
## attr("scaled:scale")
## height weight
##    20.7    99.5
```

注意在管道操作中某个操作除了被传递的第一自变量外没有其它自变量时，可以不写函数调用的空括号（）。

函数 `sweep()` 可以执行对每列更一般的变换。

26.17 用 `reshape` 包做长宽表转换

前面已经讲到，用 `tidyr` 的 `gather()`、`separate()`、`spread()` 等函数可以进行长宽表的转换。`reshape` 包是另一个可以进行长宽表转换的扩展包，这里列出其使用方法作为参考，建议主要使用 `tidyr` 的方法。

设数据框 `d.long` 变量为 `subject`(病人号，有 5 个不同值)，`time`(随访序号，取 1,2,3)，变量 `x`, `y`(每个病人每次随访的两个测量指标值)。数据框有 $5 \times 3 = 15$ 个观测（行）。数据在如下的 `longtab.csv` 中：

```
subject,time,x,y
1,1,5,9
1,2,7,7
1,3,8,6
2,1,8,1
2,2,2,1
2,3,10,9
3,1,7,10
3,2,2,8
3,3,5,3
4,1,1,10
4,2,5,1
4,3,6,1
5,1,9,8
5,2,7,8
5,3,10,10
```


读入：

```
d.long <- as.data.frame(read_csv('longtab.csv'))
```

```
## Parsed with column specification:
## cols(
##   subject = col_integer(),
##   time = col_integer(),
##   x = col_integer(),
##   y = col_integer()
## )
```

(reshape 包对 tibble 类型支持不好，所以转换成普通的数据框)

实际中，常常需要把每个病人的 3 次随访的 6 个测量指标合并到一行当中，这称为长表变宽表问题；反之则称为宽表变长表问题。

原始数据列表如下：

```
knitr::kable(d.long)
```

| subject | time | x | y |
|---------|------|----|----|
| 1 | 1 | 5 | 9 |
| 1 | 2 | 7 | 7 |
| 1 | 3 | 8 | 6 |
| 2 | 1 | 8 | 1 |
| 2 | 2 | 2 | 1 |
| 2 | 3 | 10 | 9 |
| 3 | 1 | 7 | 10 |
| 3 | 2 | 2 | 8 |
| 3 | 3 | 5 | 3 |
| 4 | 1 | 1 | 10 |
| 4 | 2 | 5 | 1 |
| 4 | 3 | 6 | 1 |
| 5 | 1 | 9 | 8 |
| 5 | 2 | 7 | 8 |
| 5 | 3 | 10 | 10 |

26.17.1 用 melt() 融化

reshape 包用 `melt()` 函数把数据框转换为一个容易变形的格式，称为“融化”。`melt()` 函数把变量分为两种：分组用 (`id.var`)，测量用 (`measure.var`)。融化保持分组不变，但是将所有的测量变量合并到一列中，列名为 `value`；相应的变量名保存到一列中，列名为 `variable`。

如下程序把 d.long 转化为“融化”格式：

```
library(reshape)
```

```
##
```

```
## 载入程辑包: 'reshape'
```

```
## The following object is masked from 'package:dplyr':
```

```
##
```

```
##      rename
```

```
## The following objects are masked from 'package:tidyr':
```

```
##
```

```
##      expand, smiths
```

```
melt.long <- melt(as.data.frame(d.long),  
  id.vars=c('subject', 'time'),  
  measure.vars=c('x', 'y'))
```

融化后的数据框为：

```
knitr::kable(melt.long)
```

| subject | time | variable | value |
|---------|------|----------|-------|
| 1 | 1 | x | 5 |
| 1 | 2 | x | 7 |
| 1 | 3 | x | 8 |
| 2 | 1 | x | 8 |
| 2 | 2 | x | 2 |
| 2 | 3 | x | 10 |
| 3 | 1 | x | 7 |
| 3 | 2 | x | 2 |
| 3 | 3 | x | 5 |
| 4 | 1 | x | 1 |
| 4 | 2 | x | 5 |
| 4 | 3 | x | 6 |
| 5 | 1 | x | 9 |
| 5 | 2 | x | 7 |
| 5 | 3 | x | 10 |
| 1 | 1 | y | 9 |
| 1 | 2 | y | 7 |
| 1 | 3 | y | 6 |
| 2 | 1 | y | 1 |
| 2 | 2 | y | 1 |
| 2 | 3 | y | 9 |
| 3 | 1 | y | 10 |
| 3 | 2 | y | 8 |
| 3 | 3 | y | 3 |
| 4 | 1 | y | 10 |
| 4 | 2 | y | 1 |
| 4 | 3 | y | 1 |
| 5 | 1 | y | 8 |
| 5 | 2 | y | 8 |
| 5 | 3 | y | 10 |

26.17.2 用 `cast()` 函数变形

用 `cast()` 函数把融化的数据框转换为要求的格式。例如,下面的程序把 `melt.long` 重新转化成了原来 `d.long` 的格式:

```
d1 <- cast(melt.long, subject + time ~ variable)
```

`cast()` 的第二自变量是公式, 波折号左边是分组变量, 右边是要从纵向转为横向的变量, 这里把 `variable`

中的 `x` 和 `y` 转为横向，相应的变量值从 `value` 中取出。结果：

```
knitr::kable(d1)
```

| subject | time | x | y |
|---------|------|----|----|
| 1 | 1 | 5 | 9 |
| 1 | 2 | 7 | 7 |
| 1 | 3 | 8 | 6 |
| 2 | 1 | 8 | 1 |
| 2 | 2 | 2 | 1 |
| 2 | 3 | 10 | 9 |
| 3 | 1 | 7 | 10 |
| 3 | 2 | 2 | 8 |
| 3 | 3 | 5 | 3 |
| 4 | 1 | 1 | 10 |
| 4 | 2 | 5 | 1 |
| 4 | 3 | 6 | 1 |
| 5 | 1 | 9 | 8 |
| 5 | 2 | 7 | 8 |
| 5 | 3 | 10 | 10 |

如下的程序把融化后的数据框转换为宽表，5 位病人每人的 3 次随访的 6 个测量值都合并到同一行中：

```
d2 <- cast(melt.long, subject ~ variable + time)
```

这里公式以病人作为仅有的分类，而变量 `x`, `y` 和 3 个 `time`(时间) 都变成了横向：

```
knitr::kable(d2)
```

| subject | x_1 | x_2 | x_3 | y_1 | y_2 | y_3 |
|---------|-----|-----|-----|-----|-----|-----|
| 1 | 5 | 7 | 8 | 9 | 7 | 6 |
| 2 | 8 | 2 | 10 | 1 | 1 | 9 |
| 3 | 7 | 2 | 5 | 10 | 8 | 3 |
| 4 | 1 | 5 | 6 | 10 | 1 | 1 |
| 5 | 9 | 7 | 10 | 8 | 8 | 10 |

变量名与时间之间以下划线连接。

26.17.3 宽表变成长表

当数据框是宽表，变量名中用序号表示时间时，需要先把变量名与时间分离出来。比如，`d.wide` 数据框是如下的一个子集：

```
d.wide <- data.frame(
  subject=1:5,
  x1=c(5,8,7,1,9),
  x2=c(7,2,2,5,7),
  y1=c(9,1,10,10,8)
)
d.wide
```

```
##   subject x1 x2 y1
## 1      1  5  7  9
## 2      2  8  2  1
## 3      3  7  2 10
## 4      4  1  5 10
## 5      5  9  7  8
```

还是先融化为长表:

```
melt.wide <- melt(
  d.wide, id.vars=c("subject"), measure.vars=c("x1", "x2", "y1"))
```

宽表融化后结果:

```
knitr::kable(melt.wide)
```

| subject | variable | value |
|---------|----------|-------|
| 1 | x1 | 5 |
| 2 | x1 | 8 |
| 3 | x1 | 7 |
| 4 | x1 | 1 |
| 5 | x1 | 9 |
| 1 | x2 | 7 |
| 2 | x2 | 2 |
| 3 | x2 | 2 |
| 4 | x2 | 5 |
| 5 | x2 | 7 |
| 1 | y1 | 9 |
| 2 | y1 | 1 |
| 3 | y1 | 10 |
| 4 | y1 | 10 |
| 5 | y1 | 8 |

这个结果没有将时间分离出来, 为此使用字符串处理:

```
melt.wide[, "time"] <- as.numeric(substring(melt.wide[, "variable"], 2))
melt.wide[, "variable"] <- substring(melt.wide[, "variable"], 1, 1)
```

```
knitr::kable(melt.wide)
```

| subject | variable | value | time |
|---------|----------|-------|------|
| 1 | x | 5 | 1 |
| 2 | x | 8 | 1 |
| 3 | x | 7 | 1 |
| 4 | x | 1 | 1 |
| 5 | x | 9 | 1 |
| 1 | x | 7 | 2 |
| 2 | x | 2 | 2 |
| 3 | x | 2 | 2 |
| 4 | x | 5 | 2 |
| 5 | x | 7 | 2 |
| 1 | y | 9 | 1 |
| 2 | y | 1 | 1 |
| 3 | y | 10 | 1 |
| 4 | y | 10 | 1 |
| 5 | y | 8 | 1 |

现在的 `melt.wide` 已经有分类变量 `subject`，时间变量 `time`，变量名列 `variable` 和变量值列 `value`，可以用 `cast()` 转换了。例如，转换成每个病人两次随访的格式，这时因为 `y` 没有第二次随访值，会等于缺失值：

```
cast.wide <- cast(melt.wide, subject + time ~ variable)
knitr::kable(cast.wide)
```

| subject | time | x | y |
|---------|------|---|----|
| 1 | 1 | 5 | 9 |
| 1 | 2 | 7 | NA |
| 2 | 1 | 8 | 1 |
| 2 | 2 | 2 | NA |
| 3 | 1 | 7 | 10 |
| 3 | 2 | 2 | NA |
| 4 | 1 | 1 | 10 |
| 4 | 2 | 5 | NA |
| 5 | 1 | 9 | 8 |
| 5 | 2 | 7 | NA |

变量名与时间之间更复杂的连接关系，如 `abc1`, `abc2`, `x1`, `y1`，不能简单用取子串完成，可以借助正则表达

式拆开。

26.17.4 变形同时进行概括

如果 `cast()` 后每组有不止一个值，可以指定一个统计函数进行汇总。下面的程序计算每个病人的所有 3 次随访的 `x` 平均值和 `y` 平均值：

```
cast(melt.long, subject ~ variable, mean)
```

```
##   subject      x      y
## 1      1 6.666667 7.333333
## 2      2 6.666667 3.666667
## 3      3 4.666667 7.000000
## 4      4 4.000000 4.000000
## 5      5 8.666667 8.666667
```

下面的程序对每个病人的每个随访时间，计算 `x` 和 `y` 的最大值：

```
cast(melt.long, subject + time ~ ., max)
```

```
##   subject time (all)
## 1      1     1     9
## 2      1     2     7
## 3      1     3     8
## 4      2     1     8
## 5      2     2     2
## 6      2     3    10
## 7      3     1    10
## 8      3     2     8
## 9      3     3     5
## 10     4     1    10
## 11     4     2     5
## 12     4     3     6
## 13     5     1     9
## 14     5     2     8
## 15     5     3    10
```

注意公式一端没有变量指定时用句点。

`reshape` 包的某些函数与 `tidyverse` 系统中函数冲突，所以使用完 `reshape` 后应卸载：

```
detach("package:reshape")
```


Chapter 27

数据汇总

前面讲了用 dplyr 包的 `summarise()` 函数进行简单概括的方法。下面描述其它一些数据概括方法。

27.1 用 `summary()` 函数作简单概括

在 `d.cancer` 中保存了 34 个病人的代码 (`id`)、年龄 (`age`)、性别 (`sex`)、病理类型 (`type`)、放疗前肿瘤体积 (`v0`)、放疗后肿瘤体积 (`v1`)。其中, `sex`、`type` 用来作为分类, 年龄可以作为连续型取值变量, 也可以分组后作为分类。体积是连续型取值变量。

对数值型向量 `x`, 用 `summary(x)` 可以获得变量的平均值、中位数、最小值、最大值、四分之一和四分之三分位数。如

```
summary(d.cancer[["v0"]])
```

```
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  12.58   43.77   93.40  110.08  157.18  330.24
```

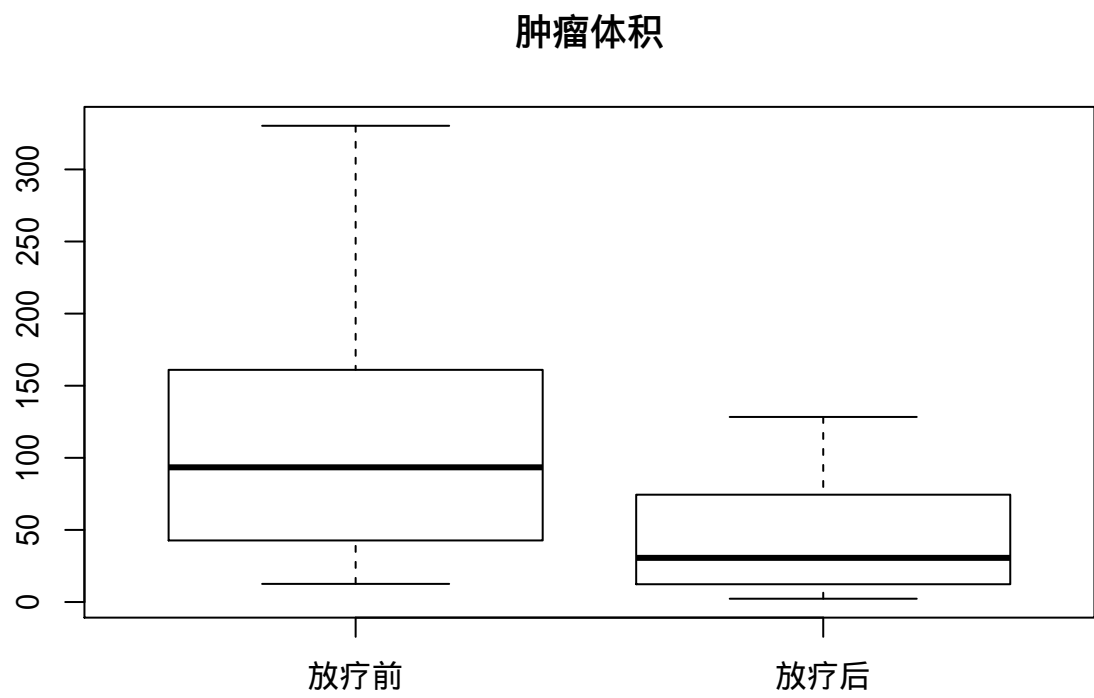
```
summary(d.cancer[["v1"]])
```

```
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   2.30   12.73   30.62   44.69   72.94  128.34
```

可以看出放疗后体积减小了很多。

可以用盒形图表现类似的信息, 如

```
boxplot(list(' 放疗前'=d.cancer[["v0"]],
            ' 放疗后'=d.cancer[["v1"]]), main=' 肿瘤体积')
```



对一个数据框 `d`, 用 `summary(d)` 可以获得每个连续型变量的基本统计量, 和每个离散取值变量的频率。如

```
summary(d.cancer)
```

```
##      id      age      sex      type
## Min.   : 1.00  Min.   :49.00 Length:34 Length:34
## 1st Qu.: 9.25  1st Qu.:55.00 Class :character Class :character
## Median :17.50  Median :67.00 Mode  :character Mode  :character
## Mean   :17.50  Mean   :64.13
## 3rd Qu.:25.75  3rd Qu.:70.00
## Max.   :34.00  Max.   :79.00
##
##      NA's :11
##      v0      v1
## Min.   : 12.58  Min.   : 2.30
## 1st Qu.: 43.77  1st Qu.: 12.73
## Median : 93.40  Median : 30.62
## Mean   :110.08  Mean   : 44.69
## 3rd Qu.:157.18  3rd Qu.: 72.94
## Max.   :330.24  Max.   :128.34
##
```

对数据框 `d`, 用 `str(d)` 可以获得各个变量的类型和取值样例。如

```
str(d.cancer)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':   34 obs. of  6 variables:
## $ id   : int   1 2 3 4 5 6 7 8 9 10 ...
## $ age  : int   70 70 69 68 67 75 52 71 68 79 ...
## $ sex  : chr   "F" "F" "F" "M" ...
## $ type: chr   "腺癌" "腺癌" "腺癌" "腺癌" ...
## $ v0   : num   26.5 135.5 209.7 61 237.8 ...
## $ v1   : num    2.91 35.08 74.44 34.97 128.34 ...
## - attr(*, "spec")=List of 2
## ..$ cols   :List of 6
## .. ..$ id   : list()
## .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
## .. ..$ age  : list()
## .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
## .. ..$ sex  : list()
## .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
## .. ..$ type: list()
## .. .. ..- attr(*, "class")= chr  "collector_character" "collector"
## .. ..$ v0   : list()
## .. .. ..- attr(*, "class")= chr  "collector_double" "collector"
## .. ..$ v1   : list()
## .. .. ..- attr(*, "class")= chr  "collector_double" "collector"
## ..$ default: list()
## .. ..- attr(*, "class")= chr  "collector_guess" "collector"
## ..- attr(*, "class")= chr "col_spec"
```

用 `head(d)` 可以返回数据框（或向量、矩阵）的前几行，用 `tail(d)` 可以返回数据框的后几行。

27.2 连续型变量概括函数

对连续取值的变量 `x`, 可以用 `mean`, `std`, `var`, `sum`, `prod`, `min`, `max` 等函数获取基本统计量。加 `na.rm=TRUE` 选项可以仅对非缺失值计算。

`sort(x)` 返回排序后的结果。`rev(x)` 把 `x` 所有元素次序颠倒后返回。`quantile(x, c(0.05, 0.95))` 可以求 `x` 的样本分位数。`rank(x)` 对 `x` 求秩得分（即名次，但从最小到最大排列）。

27.3 分类变量概括

分类变量一般输入为因子。对因子 `x`，`table(x)` 返回 `x` 的每个不同值的频率（出现次数），结果为一个类（class）为 `table` 的一维数组。每个元素有对应的元素名，为 `x` 的各水平值。如

```
res <- table(d.cancer[["sex"]]); res
```

```
##
##  F  M
## 13 21
```

```
res['F']
```

```
##  F
## 13
```

对单个分类变量，`table` 结果是一个有元素名的向量。用 `as.data.frame()` 函数把 `table` 的结果转为数据框：

```
as.data.frame(res)
```

```
##   Var1 Freq
## 1    F   13
## 2    M   21
```

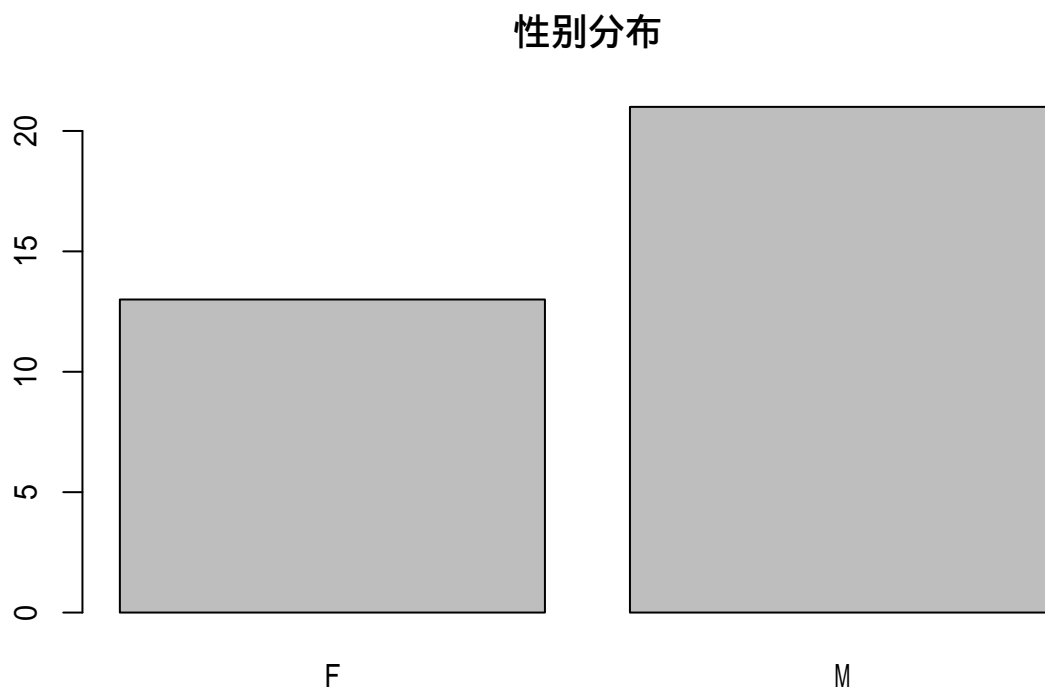
用 `prop.table()` 将频数转换成百分比：

```
prop.table(res)
```

```
##
##           F           M
## 0.3823529 0.6176471
```

`table` 作的单变量频数表可以用 `barplot` 表现为图形，如：

```
barplot(res, main=' 性别分布')
```



对两个分类变量 `x1` 和 `x2`，其每个组合的出现次数可以用 `table(x1,x2)` 函数统计，结果叫做列联表。如

```
res2 <- with(d.cancer, table(sex, type)); res2
```

```
##      type
## sex 鳞癌 腺癌
##  F    4    9
##  M   18    3
```

结果是一个类为 `table` 的二维数组（矩阵），每行以第一个变量 `x1` 的各水平值为行名，每列以第二个变量 `x2` 的各水平值为列名。这里用了 `with()` 函数引入一个数据框，后续的参数中的表达式可以直接使用数据框的变量。

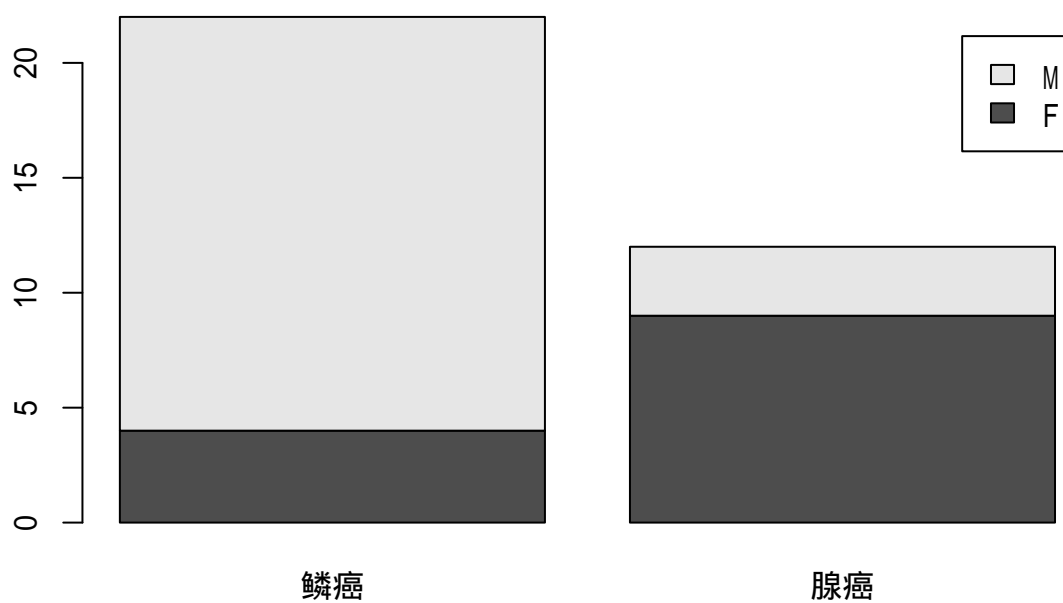
对两个分类变量, `table` 结果是一个矩阵。用 `as.data.frame` 函数把 `table` 的结果转为数据框:

```
as.data.frame(res2)
```

```
##    sex type Freq
## 1  F 鳞癌    4
## 2  M 鳞癌   18
## 3  F 腺癌    9
## 4  M 腺癌    3
```

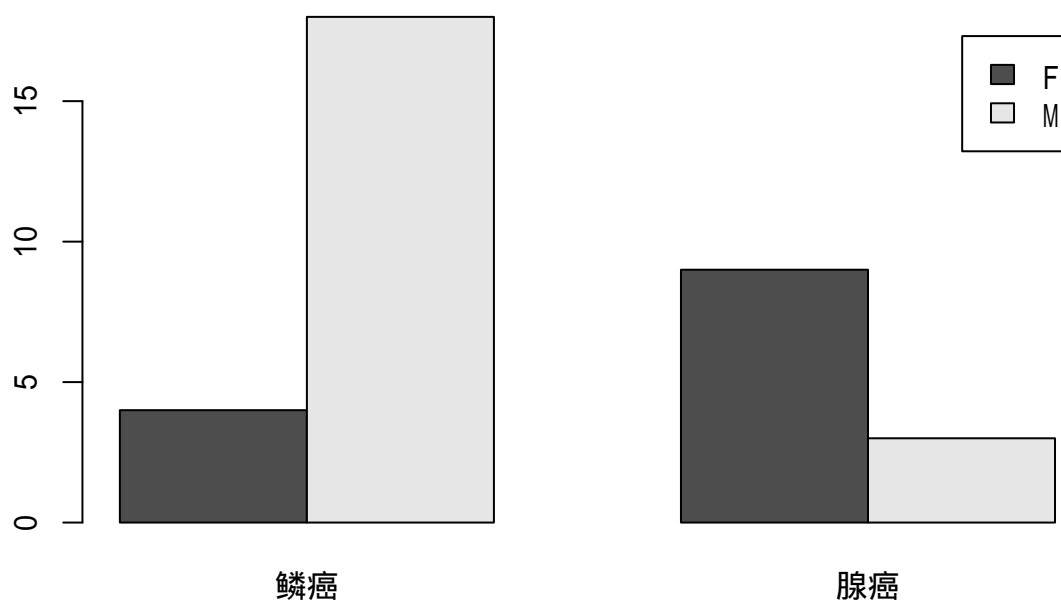
列联表的结果可以用条形图表示。如

```
barplot(res2, legend=TRUE)
```



或

```
barplot(res2, legend=TRUE, beside=TRUE)
```



对于 `table()` 的结果列联表，可以用 `addmargins()` 函数增加行和与列和。如

```
addmargins(res2)
```

```
##      type
## sex   鳞癌  腺癌 Sum
##  F      4    9  13
##  M     18    3  21
##  Sum   22   12  34
```

用 `margin.table()` 可以计算列联表行或列的和并返回，如

```
margin.table(res2, 1)
```

```
## sex
##  F  M
## 13 21
```

```
margin.table(res2, 2)
```

```
## type
## 鳞癌 腺癌
##   22  12
```

用 `prop.table(r)` 把一个列联表 `r` 转换成百分比表。如

```
prop.table(res2)
```

```
##      type
## sex      鳞癌      腺癌
##   F 0.11764706 0.26470588
##   M 0.52941176 0.08823529
```

用 `prop.table(res,1)` 把列联表 `res` 转换成行百分比表。用 `prop.table(res,2)` 把列联表 `res` 转换成列百分比表。如

```
prop.table(res2, 1)
```

```
##      type
## sex      鳞癌      腺癌
##   F 0.3076923 0.6923077
##   M 0.8571429 0.1428571
```

```
prop.table(res2, 2)
```

```
##      type
## sex      鳞癌      腺癌
##   F 0.1818182 0.7500000
##   M 0.8181818 0.2500000
```

在有多个分类变量时，用 `as.data.frame(table(x1, x2, x3))` 形成多个分类变量交叉分类的频数统计数据框。

`dplyr` 包的 `count()` 功能与 `table()` 类似。如

```
d.cancer %>%
  count(sex)
```

```
## # A tibble: 2 x 2
##   sex      n
##   <chr> <int>
## 1 F      13
## 2 M      21
```

又如

```
d.cancer %>%
  count(sex, type)
```

```
## # A tibble: 4 x 3
##   sex  type      n
```



```
##    <chr> <chr> <int>
## 1 F      鳞癌      4
## 2 F      腺癌      9
## 3 M      鳞癌     18
## 4 M      腺癌      3
```

27.4 数据框概括

用 `colMeans()` 对数据框或矩阵的每列计算均值，用 `colSums()` 对数据框或矩阵的每列计算总和。用 `rowMeans()` 和 `rowSums()` 对矩阵的每行计算均值或总和。

数据框与矩阵有区别，某些适用于矩阵的计算对数据框不适用，例如矩阵乘法。用 `as.matrix()` 把数据框的数值子集转换成矩阵。

对矩阵，用 `apply(x, 1, FUN)` 对矩阵 `x` 的每一行使用函数 `FUN` 计算结果，用 `apply(x, 2, FUN)` 对矩阵 `x` 的每一列使用函数 `FUN` 计算结果。

如果 `apply(x,1,FUN)` 中的 `FUN` 对每个行变量得到多个 m 结果，结果将是一个矩阵，行数为 m ，列数等于 `nrow(x)`。如果 `apply(x,2,FUN)` 中的 `FUN` 对每个列变量得到多个 m 结果，结果将是一个矩阵，行数为 m ，列数等于 `ncol(x)`。例如：

```
apply(as.matrix(iris[,1:4]), 2,
      function(x)
        c(n=sum(!is.na(x)),
          mean=mean(x, na.rm=TRUE),
          sd=sd(x, na.rm=TRUE)))
```

```
##      Sepal.Length Sepal.Width Petal.Length Petal.Width
## n      150.0000000 150.0000000   150.000000 150.0000000
## mean     5.8433333   3.0573333    3.758000   1.1993333
## sd       0.8280661   0.4358663    1.765298   0.7622377
```

上面的例子如果改用 `dplyr` 包的 `summarise` 函数，需要对每个列分别写出汇总结果，会比较罗嗦。

27.5 分类概括

27.5.1 用 `dplyr` 包分类概括

用 `dplyr` 包的 `group_by()` 与 `summarise()` 配合可以比较简单地进行分类概括。适用于要概括的变量个数比较少的情形。

例如，按性别分组，计算 `v0` 的平均值：

```
d.cancer %>%
  group_by(sex) %>%
  summarise(mean.v0=mean(v0, na.rm=TRUE))
```

```
## # A tibble: 2 x 2
##   sex   mean.v0
##   <chr>   <dbl>
## 1 F         113
## 2 M         108
```

下面的程序按性别分组，分别计算 v0 与 v1 的平均值：

```
d.cancer %>%
  group_by(sex) %>%
  summarise(mean.v0=mean(v0, na.rm=TRUE),
            mean.v1=mean(v1, na.rm=TRUE))
```

```
## # A tibble: 2 x 3
##   sex   mean.v0 mean.v1
##   <chr>   <dbl>   <dbl>
## 1 F         113     42.7
## 2 M         108     46.0
```

下面的程序按性别分组，计算 v0 和 v1 的平均值、标准差：

```
d.cancer %>%
  group_by(sex) %>%
  summarise(mean.v0=mean(v0, na.rm=TRUE),
            mean.v1=mean(v1, na.rm=TRUE),
            sd.v0=sd(v0, na.rm=TRUE),
            sd.v1=sd(v1, na.rm=TRUE))
```

```
## # A tibble: 2 x 5
##   sex   mean.v0 mean.v1 sd.v0 sd.v1
##   <chr>   <dbl>   <dbl> <dbl> <dbl>
## 1 F         113     42.7  100    41.7
## 2 M         108     46.0  66.5   37.3
```

以上结果如果每个变量的统计量分别占一行就好了，否则当需要分析的变量个数和统计量个数较多时结果表格可能过宽。从上面的例子还可以看出，当变量比较多、统计量比较多时，用 `summarise()` 写出的程序比较冗长。plyr 包功能更强，变量个数多、统计量多的时候能够统一处理，也能按用户需求排列结果，但是使用比 dplyr 包复杂一些。

27.5.2 用 `tapply()` 分组概括向量

用 `tapply()` 函数进行分组概括, 格式为:

```
tapply(X, INDEX, FUN)
```

其中 `X` 是一个向量, `INDEX` 是一个分类变量, `FUN` 是概括统计函数。

比如, 下面的程序分性别组计算疗前体积的均值:

```
with(d.cancer, tapply(v0, sex, mean))
```

```
##           F           M
## 113.2354 108.1214
```

27.5.3 用 `aggregate()` 分组概括数据框

`aggregate` 函数对输入的数据框用指定的分组变量(或交叉分组)分组进行概括统计。例如, 下面的程序按性别分组计算年龄、疗前体积、疗后体积的平均值:

```
aggregate(d.cancer[,c("age", "v0", "v1")],
  list(sex=d.cancer[["sex"]]), mean, na.rm=TRUE)
```

```
##   sex      age      v0      v1
## 1  F 66.14286 113.2354 42.65538
## 2  M 63.25000 108.1214 45.95524
```

`aggregate()` 第一个参数是数据框, 第二个参数是列表, 列表元素是用来分组或交叉分组的变量, 第三个参数是概括用的函数, 概括用的函数的选项可以在后面给出。

可以同时计算多个概括统计量, 如:

```
aggregate(d.cancer[,c('age', 'v0', 'v1')],
  list(sex=d.cancer[["sex"]]), summary)
```

上面的结果是两个观测、19 个变量的数据框, 作为表格太宽了。后面讲的 `plyr` 包可以做出更合理的表格。

可以交叉分组后概括, 如

```
with(d.cancer,
  aggregate(cbind(v0, v1), list(sex=sex, type=type), mean))
```

```
##   sex type      v0      v1
## 1  F 鳞癌 126.99250 45.54750
## 2  M 鳞癌 113.55722 49.65556
## 3  F 腺癌 107.12111 41.37000
## 4  M 腺癌  75.50667 23.75333
```

27.5.4 用 `split()` 函数分组后概括

`split` 函数可以把数据框的各行按照一个或几个分组变量分为子集的列表，然后可以用 `sapply()` 或 `vapply()` 对每组进行概括。如

```
sp <- split(d.cancer[,c('v0','v1')], d.cancer[, 'sex'])
sapply(sp, colMeans)
```

```
##           F           M
## v0 113.23538 108.12143
## v1  42.65538  45.95524
```

返回矩阵，行为变量 `v0`, `v1`，列为不同性别，值为相应的变量在各性别组的平均值。当 `sapply()` 对列表每项的操作返回一个向量时，总是列表每项的输出保存为结果的一列。`colMeans` 函数计算分组后数据框子集每列的平均值。

27.5.5 用 `plyr` 包进行分类概括

`plyr` 则是一个专注于分组后分别分析然后将分析结果尽可能合理地合并的扩展包，功能强大，`dplyr` 包仅针对数据框，使用更方便，但是对于复杂情况功能不如 `plyr` 包强。

`plyr` 的输入支持数组、数据框、列表，输出支持数组、数据框、列表或无输出。分组分析的函数输出格式需要与指定的输出格式一致。

这里主要介绍从数据框分组概括并将结果保存为数据框的方法，使用 `plyr` 包的 `ddply()` 函数。实际上，`dplyr` 包的这种功能更方便。`plyr` 包的优点是可以自定义概括函数，使得结果表格符合用户的预期，处理多个变量时程序更简洁。

`plyr` 包与 `dplyr` 包的函数名冲突比较大，所以需要先卸载 `dplyr` 包再调用 `plyr` 包：

```
if("dplyr" %in% .packages()) detach("package:dplyr")
library(plyr)
```

```
## -----
## You have loaded plyr after dplyr - this is likely to cause problems.
## If you need functions from both plyr and dplyr, please load plyr first, then dplyr:
## library(plyr); library(dplyr)
## -----
##
## 载入程辑包: 'plyr'
##
## The following object is masked from 'package:purrr':
##
## compact
```

`ddply()` 函数第一自变量是要分组的数据框，第二自变量是分组用的变量名，第三自变量是一个概括函数，此概括函数以一个数据框子集（数据类型是数据框）为输入，输出是一个数值、一个数值型向量或者一个数据框，但最好是数据框。例如，按性别分组，计算 `v0` 的平均值：

```
ddply(d.cancer, 'sex',
      function(d) c(mean.v0 = mean(d[["v0"]], na.rm=TRUE)))
```

```
##   sex  mean.v0
## 1   F 113.2354
## 2   M 108.1214
```

下面的程序按性别分组，分别计算 `v0` 与 `v1` 的平均值：

```
ddply(d.cancer, 'sex',
      function(d) colMeans(d[,c('v0', 'v1')]))
```

```
##   sex      v0      v1
## 1   F 113.2354 42.65538
## 2   M 108.1214 45.95524
```

下面的程序按性别分组，计算 `v0` 和 `v1` 的平均值、标准差：

```
f1 <- function(dsub){
  tab <- tibble(
    '变量'=c('v0', 'v1'),
    '均值'=c(mean(dsub[, 'v0'], na.rm=TRUE),
              mean(dsub[, 'v1'], na.rm=TRUE)),
    '标准差'=c(sd(dsub[, 'v0'], na.rm=TRUE),
                sd(dsub[, 'v1'], na.rm=TRUE)))
  tab
}
ddply(d.cancer, 'sex', f1)
```

```
##   sex 变量      均值      标准差
## 1   F  v0 113.23538 100.06621
## 2   F  v1  42.65538  41.72226
## 3   M  v0 108.12143  66.45374
## 4   M  v1  45.95524  37.27592
```

注意 `f1()` 结果是一个数据框。程序有些重复内容，对每个变量和每个统计量都需要分别写出，如果这样用 `plyr` 包就不如直接用 `dplyr::summarise()` 了。下面用 `vapply()` 简化程序。

按性别分组，然后 `v0`、`v1` 各自一行结果，计算非缺失值个数、均值、标准差、中位数：

```
f2 <- function(d){
  variables <- c('v0', 'v1')
```

```

d1 <- d[,variables,drop=FALSE]
nnotmiss <- vapply(d1, function(x) sum(!is.na(x)), 1L)
xm <- vapply(d1, mean, 0.0, na.rm=TRUE)
xsd <- vapply(d1, sd, 1.0, na.rm=TRUE)
xmed <- vapply(d1, median, 0.0, na.rm=TRUE)
data.frame(variable=variables,
            n=nnotmiss,
            mean=xm,
            sd=xsd,
            median=xmed)
}
ddply(d.cancer, 'sex', f2)

```

```

##   sex variable  n      mean      sd median
## 1   F      v0 13 113.23538 100.06621  67.37
## 2   F      v1 13  42.65538  41.72226  27.32
## 3   M      v0 21 108.12143  66.45374 101.65
## 4   M      v1 21  45.95524  37.27592  32.10

```

f2() 函数针对分组后的数据框子集，这样的函数可以先在一个子集上试验。在 f2() 函数中，设输入的数据子集为 d，要分析的变量组成的数据框为 d1，用 vapply() 函数对 d1 的每一列计算一种统计量，然后将每种统计量作为结果数据框的一列。vapply() 函数类似于 lapply() 和 sapply()，但是用第三个自变量表示要应用的变换函数的返回值类型和个数，用举例的方法给出。

ddply() 也可以对交叉分类后每个类分别汇总，例如按照性别与病理类型交叉分组后汇总 v0、v1：

```
ddply(d.cancer, c('sex', 'type'), f2)
```

```

##   sex type variable  n      mean      sd median
## 1   F 鳞癌      v0  4 126.99250  83.82544 119.000
## 2   F 鳞癌      v1  4  45.54750  23.55433  40.920
## 3   F 腺癌      v0  9 107.12111 110.67144  42.700
## 4   F 腺癌      v1  9  41.37000  48.95945   9.450
## 5   M 鳞癌     v0 18 113.55722  68.88281 103.275
## 6   M 鳞癌     v1 18  49.65556  38.96325  33.730
## 7   M 腺癌     v0  3  75.50667  44.36592  61.000
## 8   M 腺癌     v1  3  23.75333  11.32141  23.960

```

上面的程序写法适用于已知要分析的变量名的情况。如果想对每个数值型变量都分析，而且想把要计算的统计量用统一的格式调用，可以写成：

```

f3 <- function(d){
  ff <- function(x){

```

```

      c(n=sum(!is.na(x)),
        each(mean, sd, median)(x, na.rm=TRUE))
    }
    ldply(Filter(is.numeric, d), ff)
  }
ddply(d.cancer, 'sex', f3)

```

```

##   sex .id  n      mean      sd median
## 1  F  id 13  17.92308  12.325188  19.00
## 2  F age  7  66.14286   6.792853  69.00
## 3  F v0 13 113.23538 100.066207  67.37
## 4  F v1 13  42.65538  41.722263  27.32
## 5  M  id 21  17.23810   8.502381  17.00
## 6  M age 16  63.25000  10.096204  66.50
## 7  M v0 21 108.12143  66.453742 101.65
## 8  M v1 21  45.95524  37.275917  32.10

```

`ff()` 函数对输入的一个数值型向量计算 4 种统计量，返回一个长度为 4 的数值型向量，用来对分组后的数据子集中的一列计算 4 种统计量。`plyr` 包的 `each()` 函数接受多个函数，返回一个函数可以同时得到这几个函数的结果，结果中各元素用输入的函数名命名。`f3()` 函数中的 `Filter` 函数用于从列表或数据框中取出满足条件的项，这里取出输入的数据子集 `d` 中所有的数值型列。`f3()` 函数中的 `ldply()` 函数接受一个列表或看成列表的一个数据框，对数据框的每列应用 `ff()` 函数计算 4 种统计量，然后合并所有各列的统计量为一个数据框，结果数据框的每行对应于 `d` 中的一列。程序中的 `ddply()` 函数接受一个数据框，第二自变量指定用来将数据框分组的变量，第三自变量 `f3()` 是对分组后的数据框子集进行分析的函数，此函数接受一个数据框，输出一个数据框。

上面的程序也可以利用无名函数写成：

```

f4 <- function(x){
  c(n=sum(!is.na(x)),
    each(mean, sd, median)(x, na.rm=TRUE))
}
ddply(d.cancer, 'sex',
      function(d)
        ldply(Filter(is.numeric, d), f4))

```

```

##   sex .id  n      mean      sd median
## 1  F  id 13  17.92308  12.325188  19.00
## 2  F age  7  66.14286   6.792853  69.00
## 3  F v0 13 113.23538 100.066207  67.37
## 4  F v1 13  42.65538  41.722263  27.32
## 5  M  id 21  17.23810   8.502381  17.00

```

```
## 6   M age 16  63.25000  10.096204  66.50
## 7   M v0 21 108.12143  66.453742 101.65
## 8   M v1 21  45.95524  37.275917  32.10
```

27.6 练习

- 把 “patients.csv” 读入 “d.patients” 中，并计算发病年龄、发病年、发病月、发病年月（格式如 “200702” 表示 2007 年 2 月份）。
- 把 “现住地址国标” 作为字符型，去掉最后两位，仅保留前 6 位数字，保存到变量 “地址编码” 中。
- 按照地址编码和发病年月交叉分类汇总发病人数，保存到数据框 d.pas1 中，然后保存为 CSV 文件 “分区分年月统计.csv” 中。要求结果有三列：“地址编码”、“发病年月”、“发病人数”。
- 按照地址编码和发病月分类汇总发病人数，保存到数据框 d.pas2 中，然后保存为 CSV 文件 “分区分月统计.csv” 中。要求每个地址编码占一行，各列为地址编码以及 1、2、.....、12 各月份，每行为同一地址编码各月份的发病数。
- 按发病年月和性别汇总发病人数，并计算同年月不分性别的发病总人数。结果保存到数据框 d.pas3 中，然后保存到 CSV 文件 “分年月分性别统计.csv” 中。要求每个不同年月占一行，变量包括年月、男性发病数、女性发病数、总计。
- 分析病人的职业分布，保存到数据框 d.pas4 中，然后保存到 CSV 文件 “职业构成.csv” 中。要求各列为职业、发病人数、百分比（结果乘以 100 并保留一位小数）。
- 把年龄分成 0—9, 11—19,, 70 以上各段，保存为 “年龄段” 变量。用年龄段和性别交叉汇总发病人数和百分比（结果乘以 100 并保留一位小数），保存到 “年龄性别分布.csv” 中。要求将每个年龄段的男性发病人数、发病率、女性发病人数、发病率存为一行。

Part VI

绘图

Chapter 28

绘图

R 语言的前身是 S 语言，S 语言的设计目的就是交互式数据分析、绘图。所以绘图是 R 的重要功能。

R 有最初的基本绘图，这是从 S 语言继承过来的，还有一些功能更易用、更强大的绘图系统，如 `lattice`、`ggplot2`。基本绘图使用简单，灵活性强，但是为了做出满意的图形需要比较多的调整。这里先讲解 R 语言的基本绘图功能。

R 的基本绘图功能有两类图形函数：高级图形函数，直接针对某一绘图任务作出完整图形；低级图形函数，在已有图形上添加内容。具备有限的与图形交互的能力（函数 `locator` 和 `identify`）。

28.1 常用高级图形

28.1.1 条形图

`d.cancer` 数据框包含了肺癌病人放疗的一些数据。

```
d.cancer <- readr::read_csv("cancer.csv", locale=locale(encoding="GBK"))
```

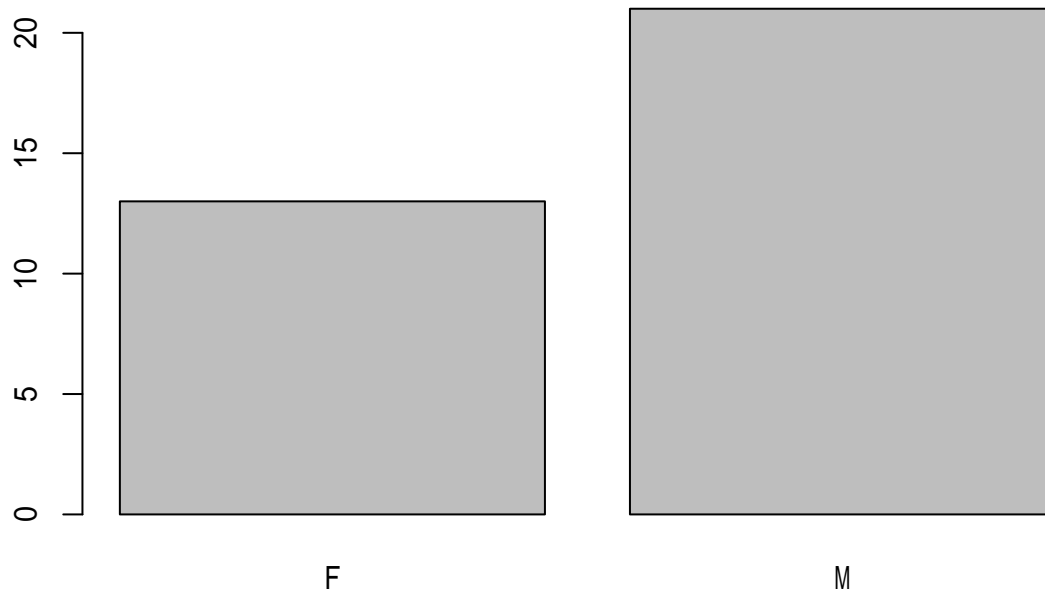
```
## Parsed with column specification:
## cols(
##   id = col_integer(),
##   age = col_integer(),
##   sex = col_character(),
##   type = col_character(),
##   v0 = col_double(),
##   v1 = col_double()
## )
```

统计男女个数并用条形图表示：

```
res1 <- table(d.cancer[, 'sex']); print(res1)
```

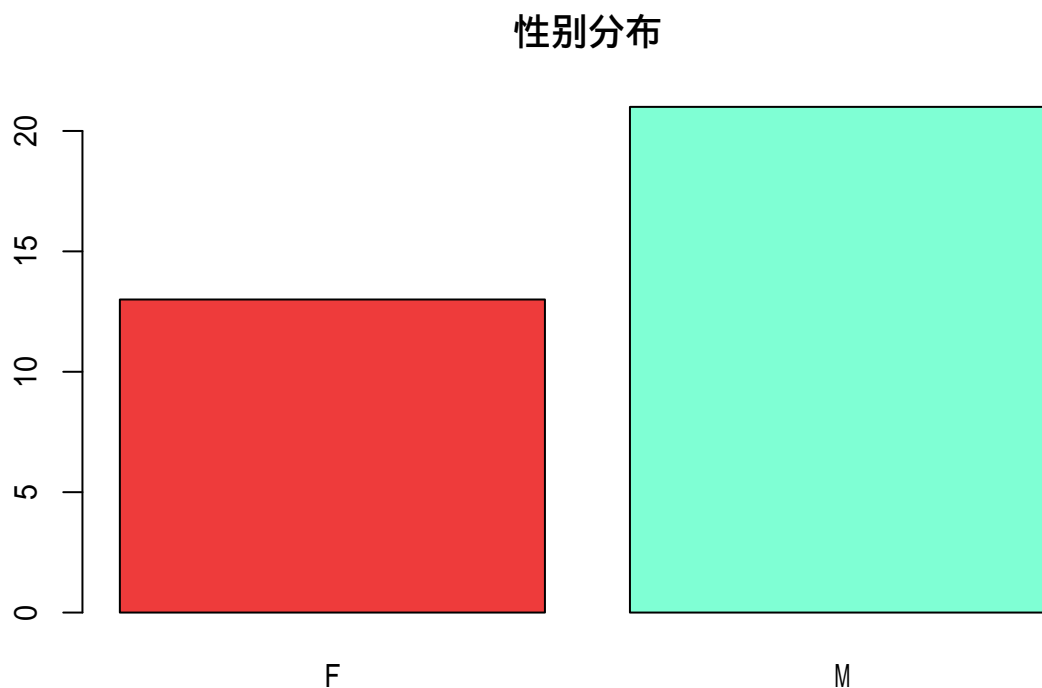
```
##  
##  F  M  
## 13 21
```

```
barplot(res1)
```



可以增加标题，采用不同的颜色：

```
barplot(res1, main=" 性别分布",  
        col=c("brown2", "aquamarine1"))
```



R 函数 `colors()` 可以返回 R 中定义的用字符串表示的六百多种颜色名字。如

```
head(colors(), 6)
```

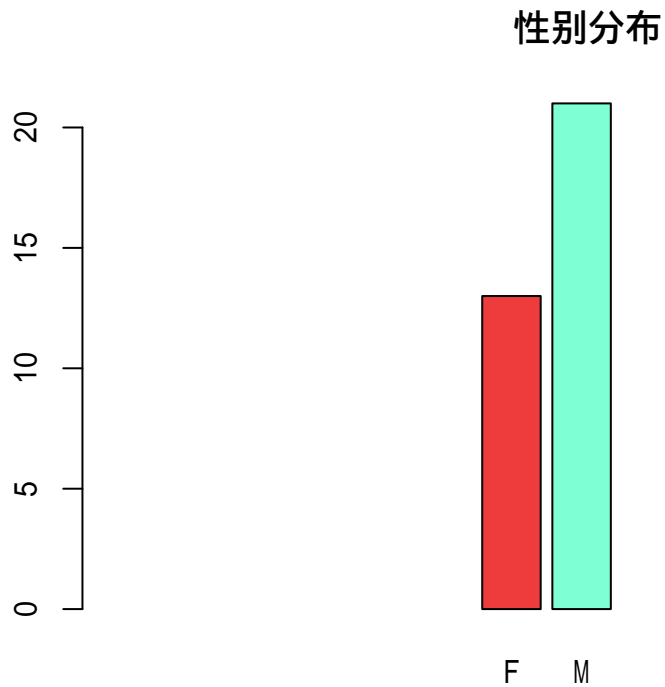
```
## [1] "white"          "aliceblue"      "antiquewhite"  "antiquewhite1"
## [5] "antiquewhite2" "antiquewhite3"
```

下面的函数可以用来挑选颜色，鼠标点击画出的颜色就可以挑选，结果返回挑选出的颜色名：

```
select.colors <- function(){
  nc <- length(colors())
  x <- rep(seq(26), 26)[1:nc]
  y <- rep(seq(26), each=26)[1:nc]
  cols <- colors()
  plot(x, y, type="p", pch=16, cex=2,
       col=cols)
  res <- cols[identify(x,y, labels=cols)]
  res
}
```

用 `width` 选项与 `xlim` 选项配合可以调整条形宽度，如

```
barplot(res1, width=0.5, xlim=c(-3, 5),  
        main=" 性别分布",  
        col=c("brown2", "aquamarine1"))
```



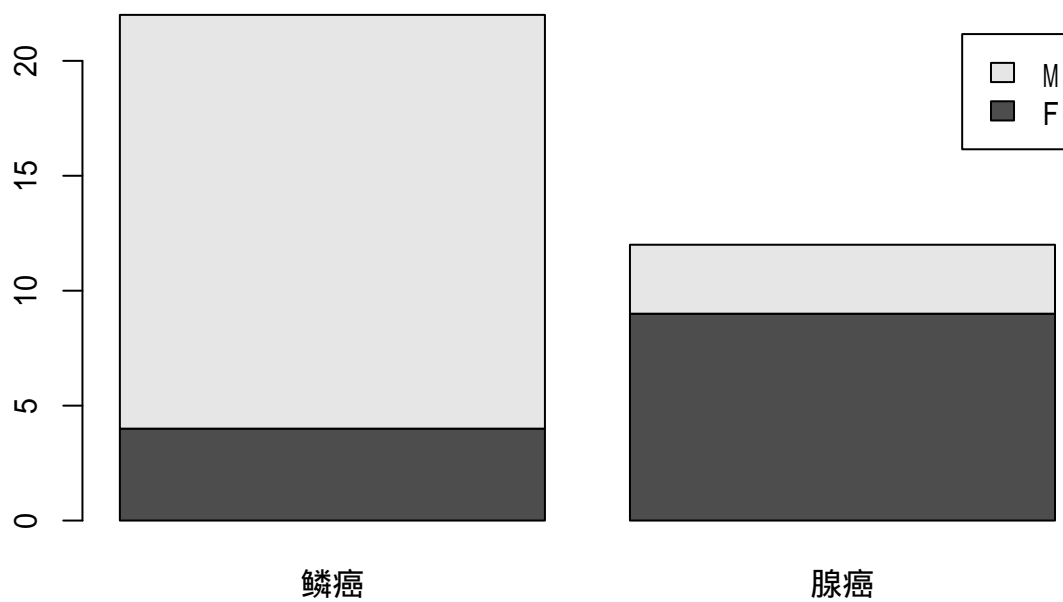
按性别与病理类型交叉分组后统计频数，结果称为列联表：

```
res2 <- with(d.cancer, table(sex, type)); res2
```

```
##      type  
## sex 鳞癌 腺癌  
##  F    4    9  
##  M   18    3
```

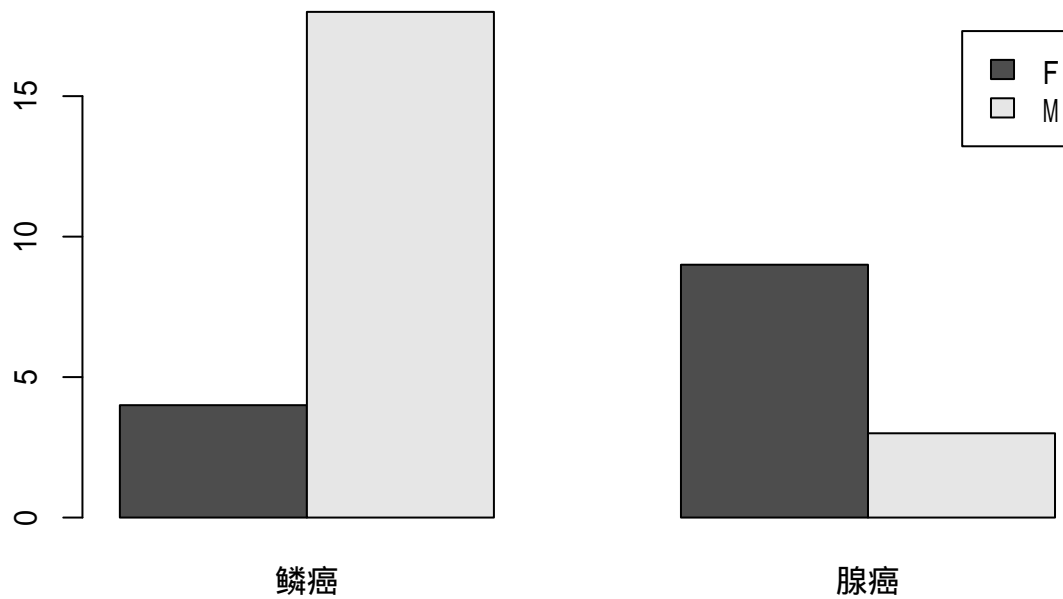
用分段条形图表现交叉分组频数，交叉频数表每列为一条：

```
barplot(res2, legend=TRUE)
```



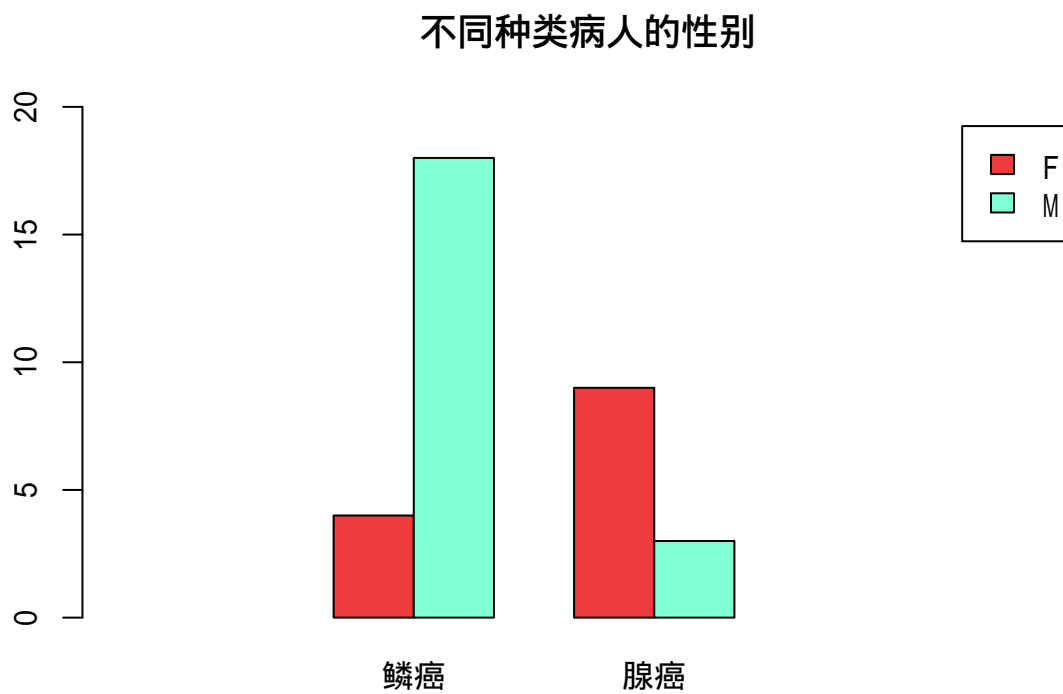
用并排条形图表现交叉分组频数，交叉频数表每列为一组：

```
barplot(res2, beside=TRUE, legend=TRUE)
```



增加标题，指定颜色，调整图例位置，调整条形宽度：

```
barplot(res2, beside=TRUE, legend=TRUE,  
        main=' 不同种类病人的性别',  
        ylim=c(0, 20),  
        xlim=c(-1, 6), width=0.6,  
        col=c("brown2", "aquamarine1"))
```

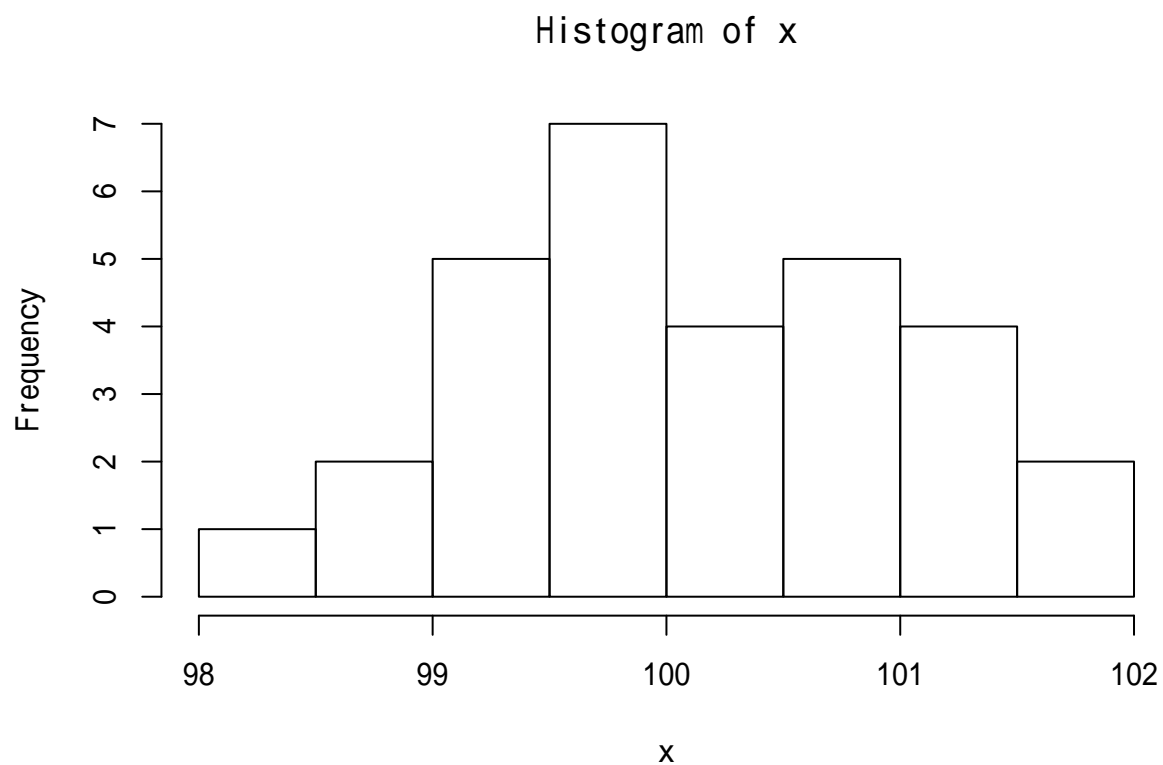
28.1.2 直方图和密度估计图

用 `hist` 作直方图以了解连续取值变量分布情况。例如，下面的程序模拟正态分布数据并做直方图：

```
x <- rnorm(30, mean=100, sd=1)
print(round(x,2))
```

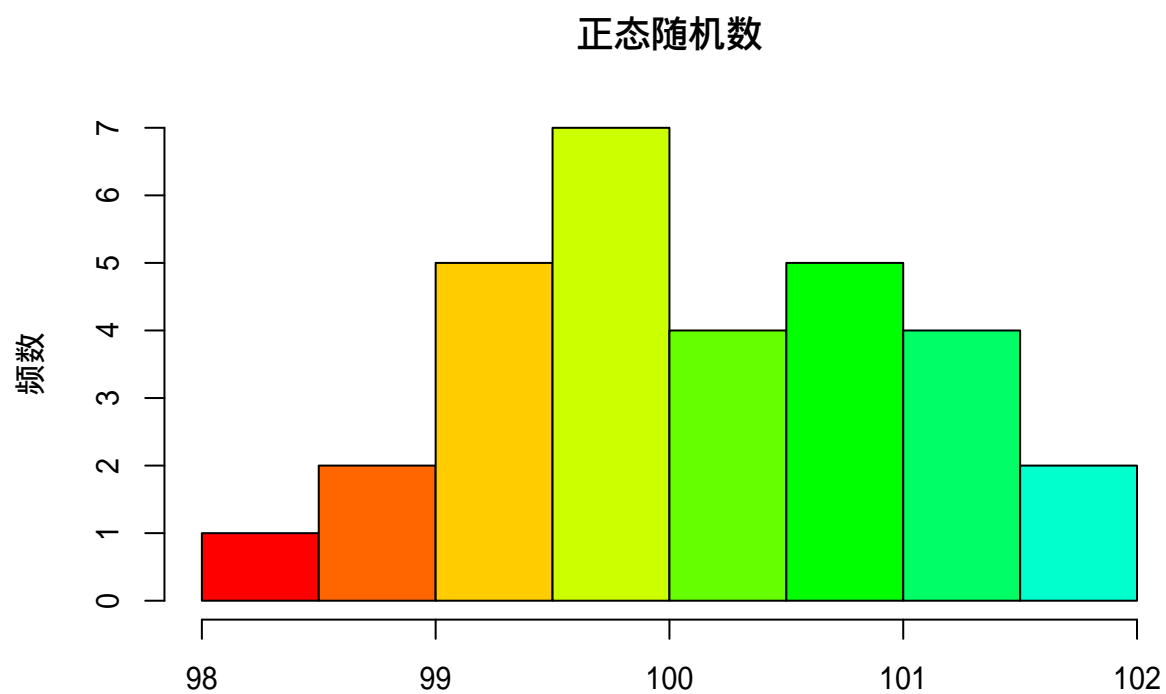
```
## [1] 99.39 101.76 100.34 100.94 99.48 100.24 101.00 99.71 101.00 99.21
## [11] 100.56 100.34 98.88 99.95 100.58 99.64 101.71 98.84 100.15 100.68
## [21] 100.85 101.13 99.53 101.03 99.38 99.90 99.31 99.70 98.22 99.83
```

```
hist(x)
```



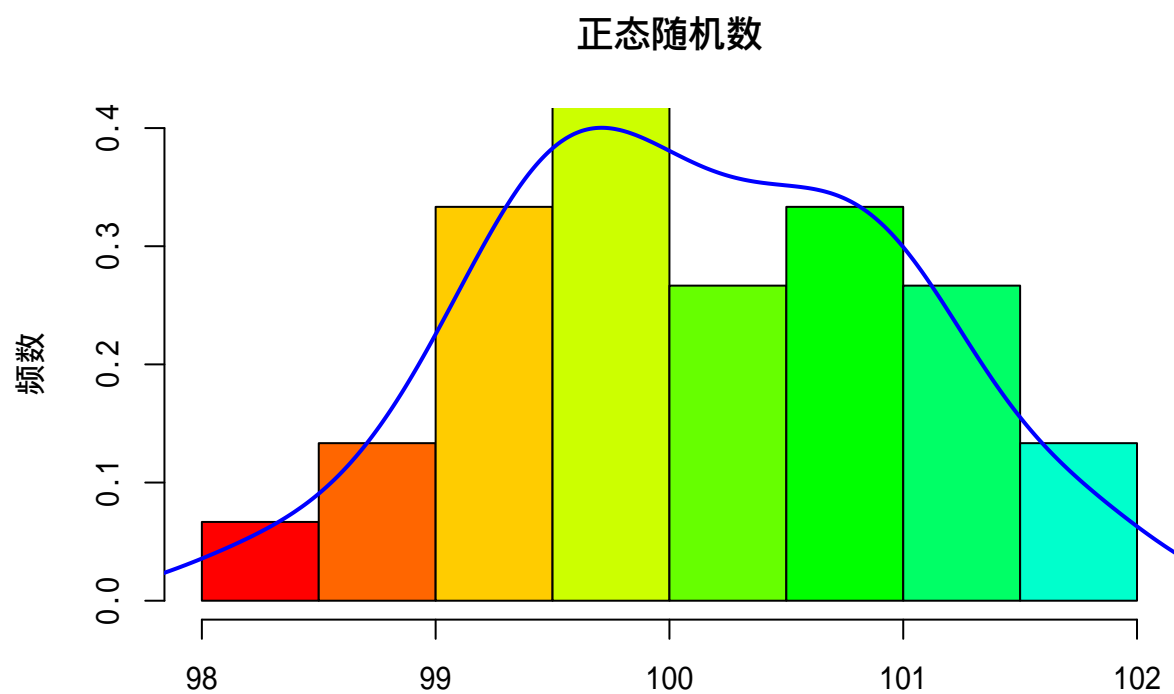
可以用 `main=`、`xlab=`、`ylab=` 等选项，可以用 `col=` 指定各个条形的颜色，如：

```
hist(x, col=rainbow(15),  
     main=' 正态随机数', xlab='', ylab=' 频数')
```



函数 `density()` 估计核密度。下面的程序作直方图，并添加核密度曲线：

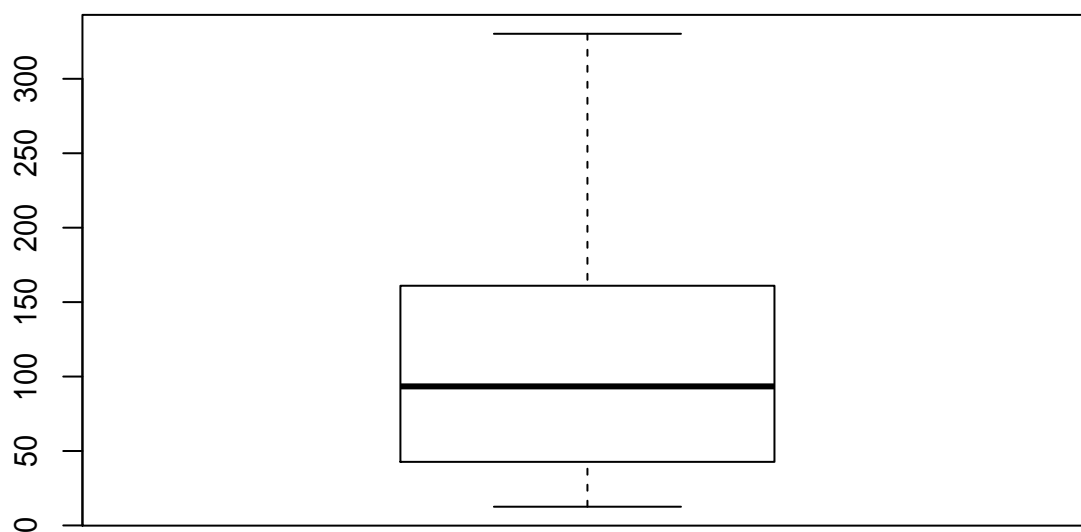
```
tmp.dens <- density(x)
hist(x, freq=FALSE,
     ylim=c(0,max(tmp.dens$y)),
     col=rainbow(15),
     main=' 正态随机数 ',
     xlab='', ylab=' 频数 ')
lines(tmp.dens, lwd=2, col='blue')
```



28.1.3 盒形图

盒形图可以简洁地表现变量分布，如

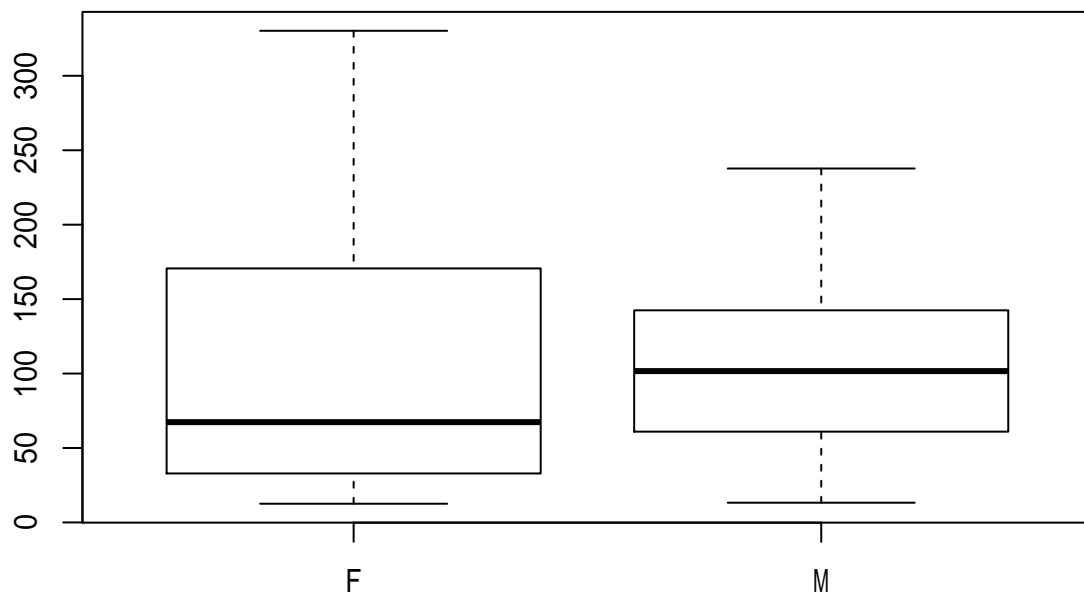
```
with(d.cancer, boxplot(v0))
```



其中中间粗线是中位数，盒子上下边缘是 $\frac{3}{4}$ 和 $\frac{1}{4}$ 分位数，两条触须线延伸到取值区域的边缘。

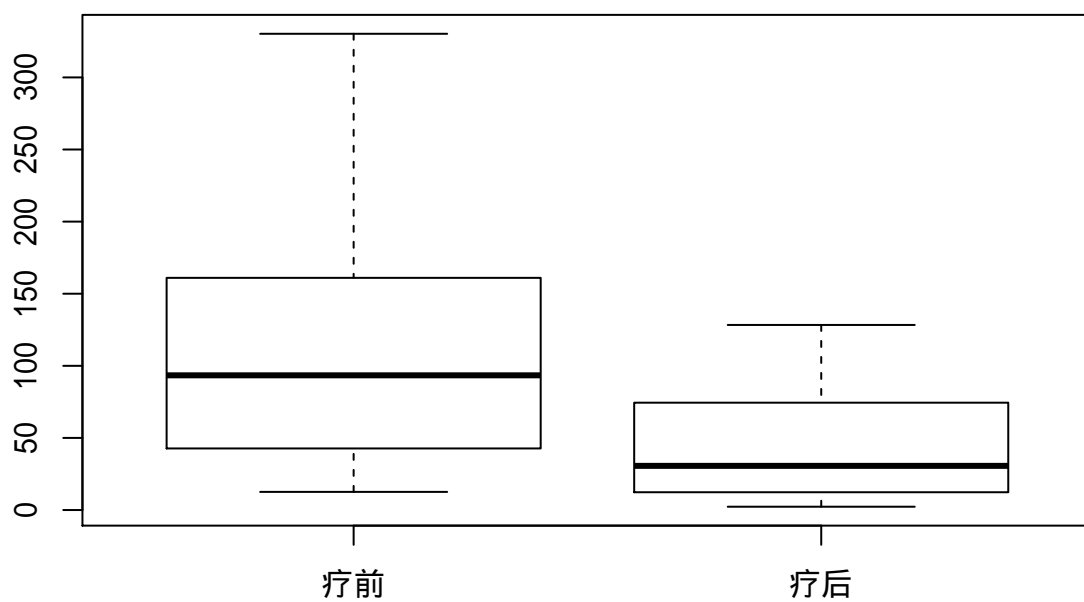
盒形图可以很容易地比较两组或多组，如

```
with(d.cancer, boxplot(v0 ~ sex))
```



也可以画若干个变量的并排盒形图，如

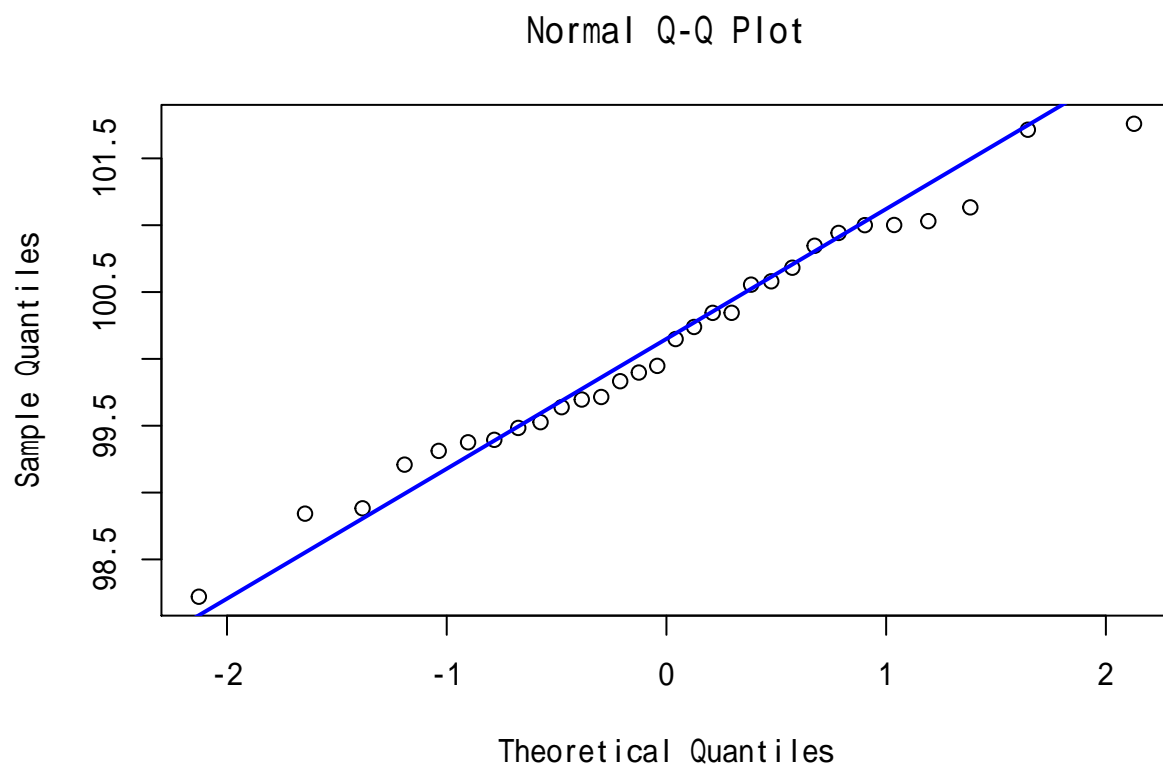
```
with(d.cancer,  
      boxplot(list(' 疗前'=v0, ' 疗后'=v1)))
```



28.1.4 正态 QQ 图

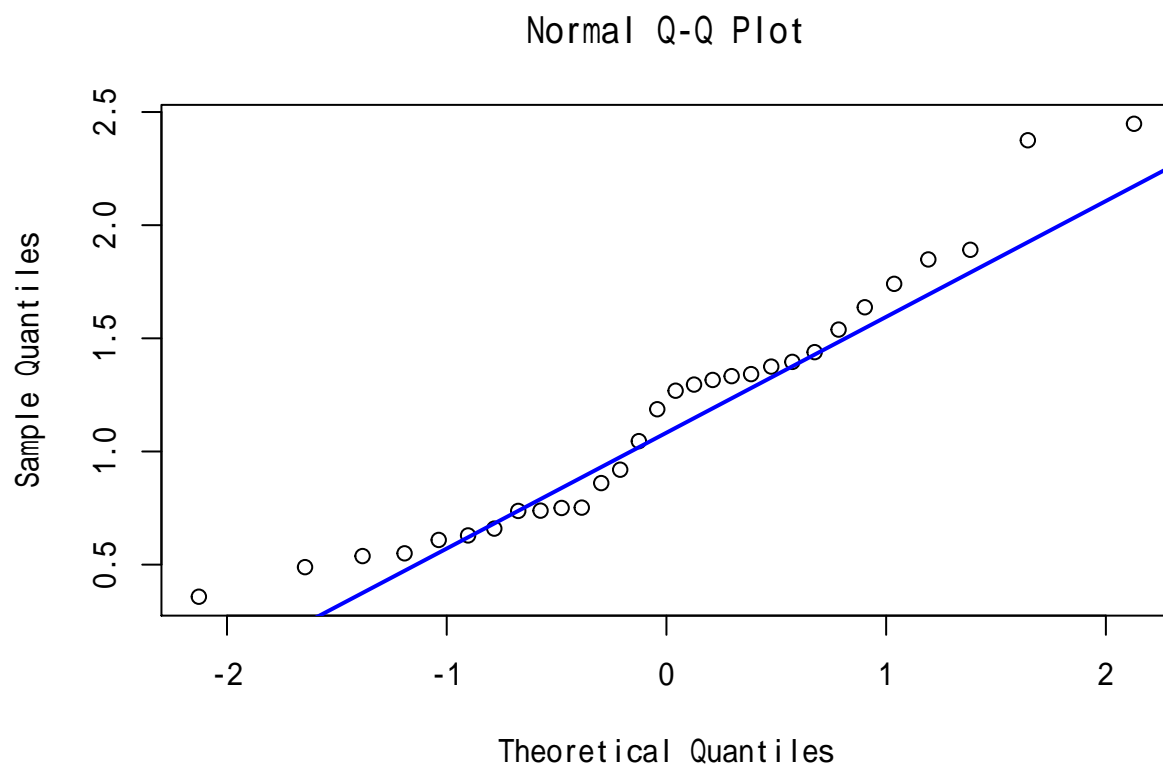
用 `qqnorm` 和 `qqline` 作正态 QQ 图。当变量样本来自正态分布总体时，正态 QQ 图的散点近似在一条直线周围。下面的程序模拟正态分布随机数，并作正态 QQ 图：

```
qqnorm(x)
qqline(x, lwd=2, col='blue')
```



下面的程序模拟对数正态数据，并作正态 QQ 图:

```
z <- 10^rnorm(30, mean=0, sd=0.2)
qqnorm(z)
qqline(z, lwd=2, col='blue')
```

28.1.5 散点图

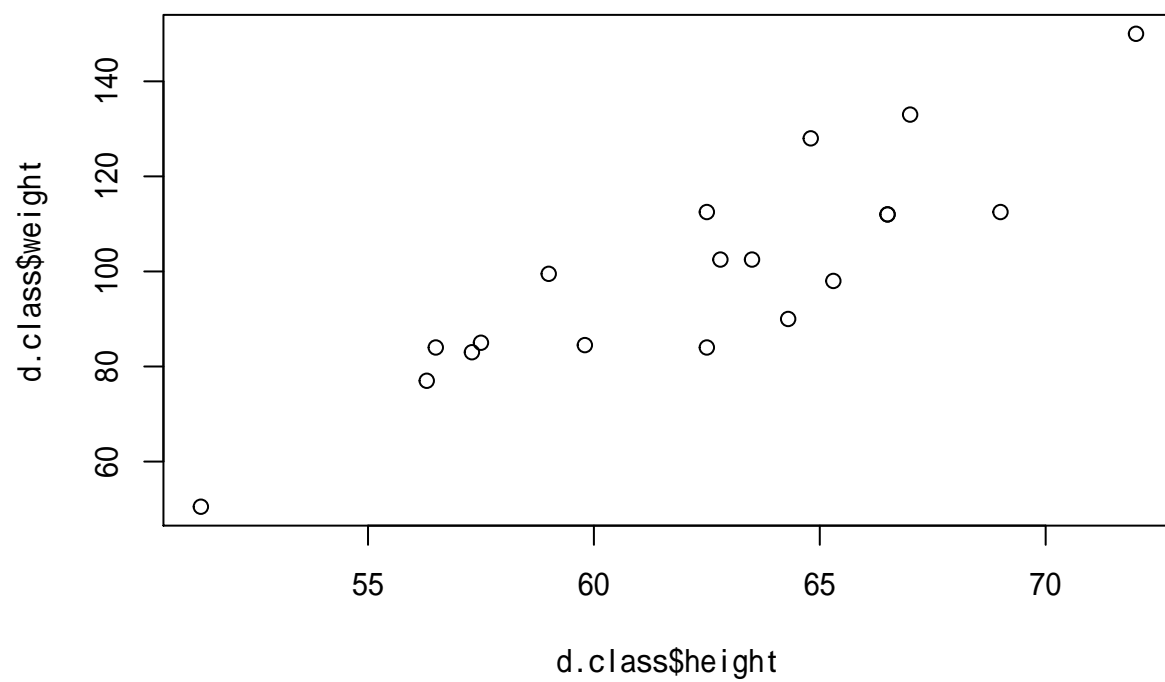
以 d.class 数据为例，有 name, sex, age, height, weight 等变量。

```
d.class <- read_csv("class.csv")
```

```
## Parsed with column specification:  
## cols(  
##   name = col_character(),  
##   sex = col_character(),  
##   age = col_integer(),  
##   height = col_double(),  
##   weight = col_double()  
## )
```

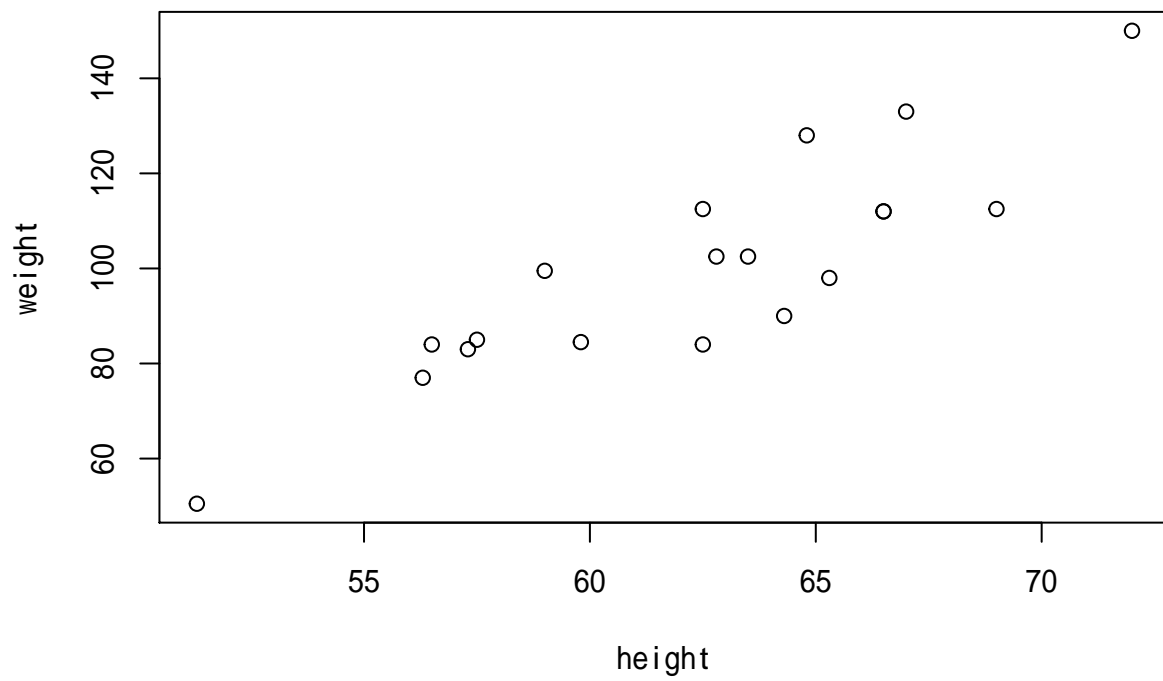
体重对身高的散点图：

```
plot(d.class$height, d.class$weight)
```



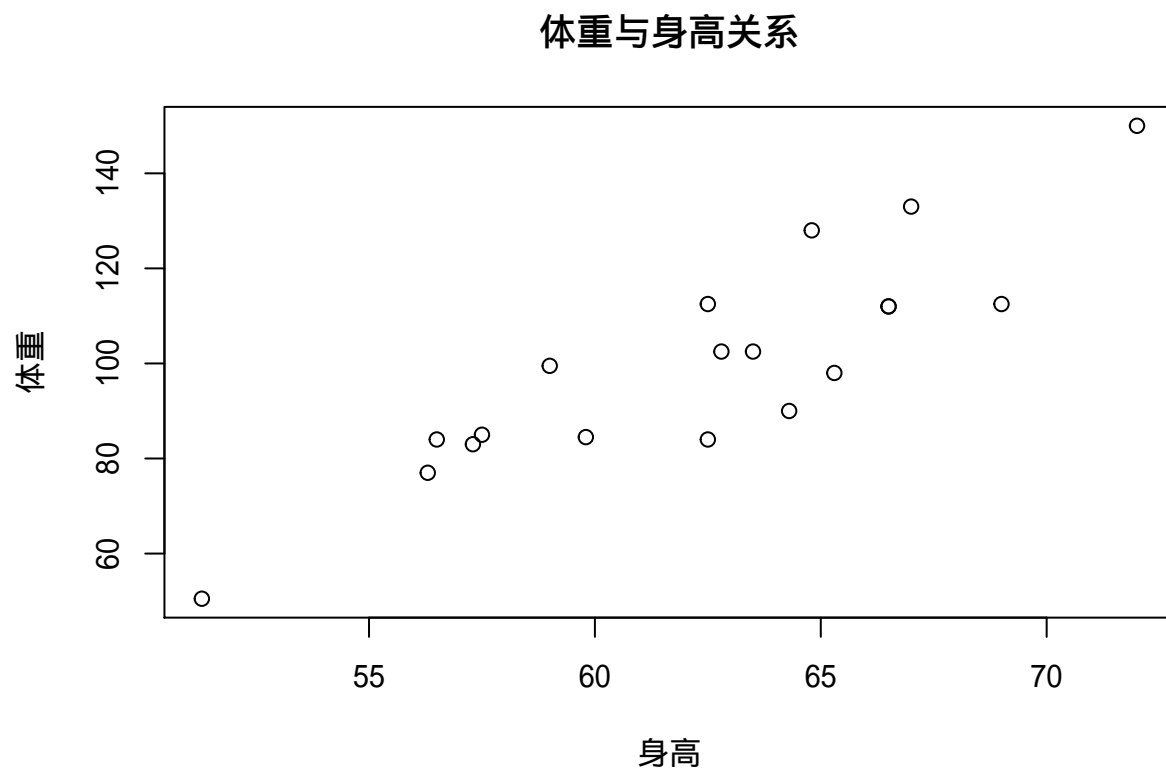
用 `with()` 函数简化数据框变量访问格式:

```
with(d.class,  
      plot(height, weight))
```



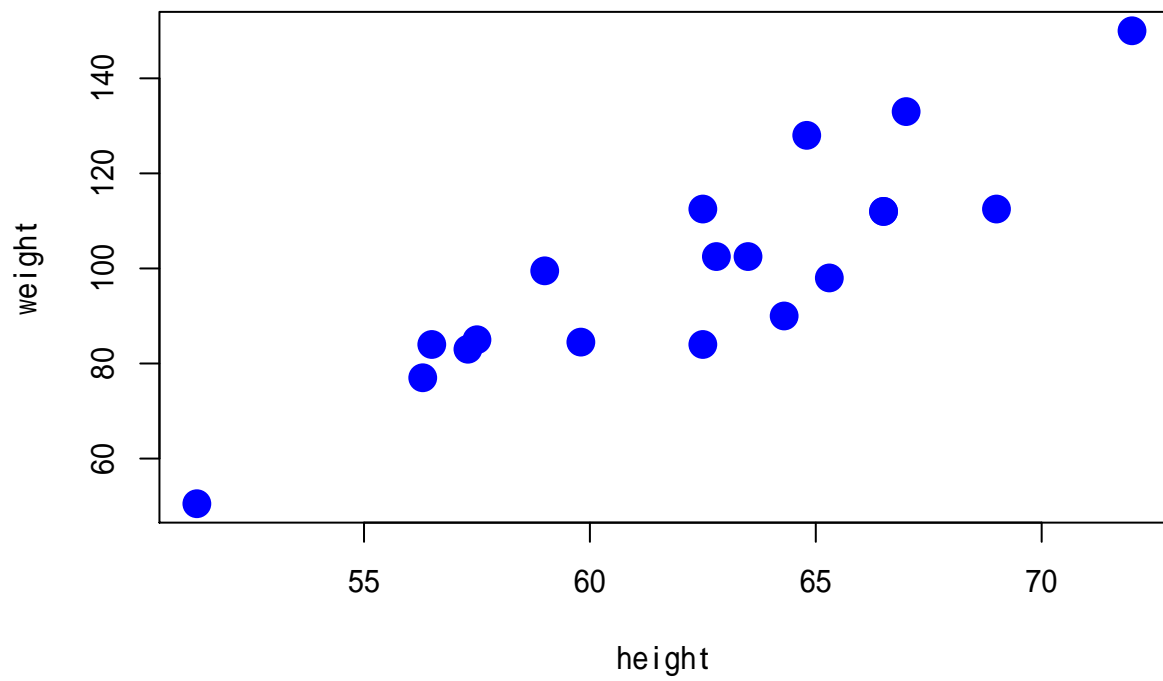
在 `plot()` 函数内用 `main` 参数增加标题，用 `xlab` 参数指定横轴标注，用 `ylab` 参数指定纵轴标注，如

```
with(d.class,  
  plot(height, weight,  
        main=' 体重与身高关系',  
        xlab=' 身高', ylab=' 体重'))
```



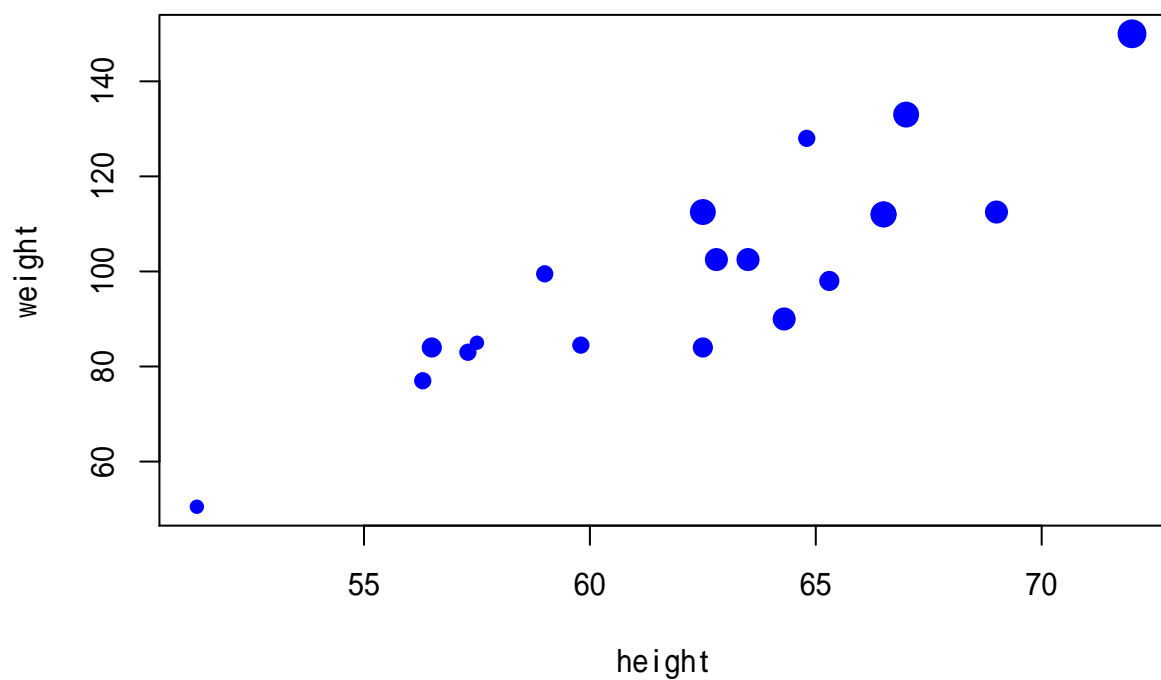
用 `pch` 参数指定不同散点形状，用 `col` 参数指定颜色，用 `cex` 参数指定大小，如：

```
with(d.class,  
  plot(height, weight,  
        pch=16, col='blue',  
        cex=2))
```



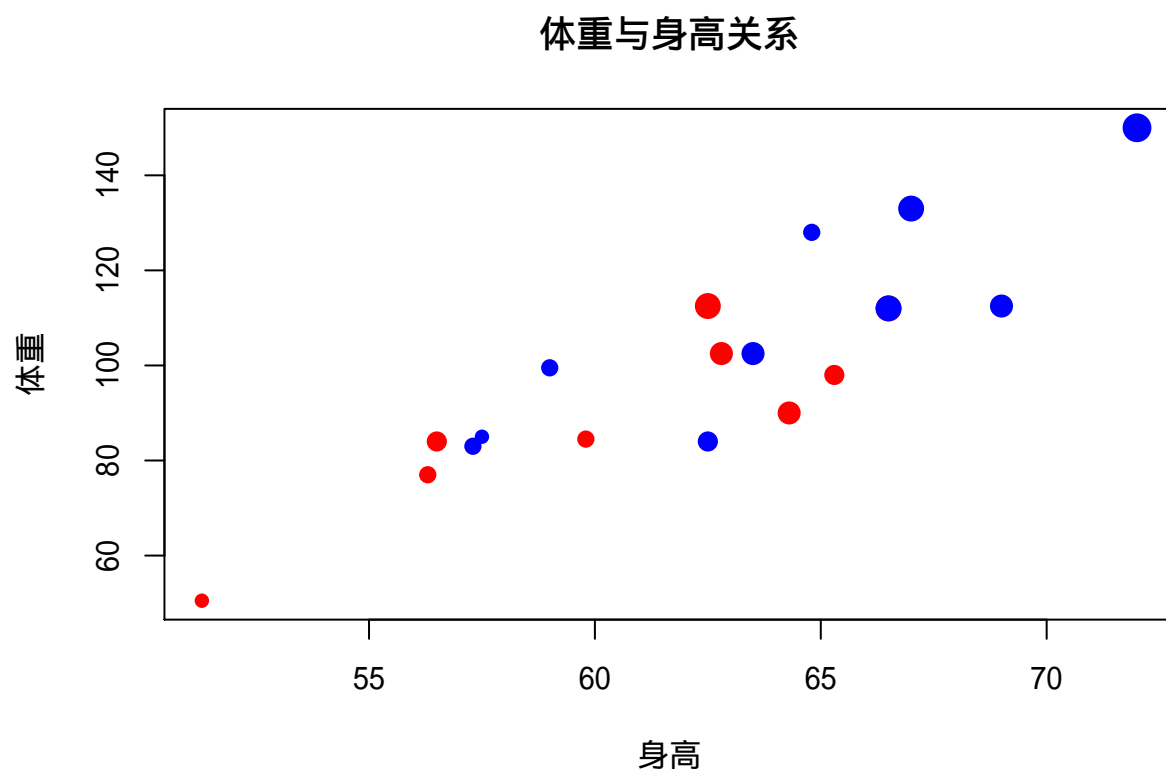
用气泡大小表现第三维（年龄）：

```
with(d.class,
  plot(height, weight,
    pch=16, col='blue',
    cex=1 + (age - min(age))/(max(age)-min(age)))
```



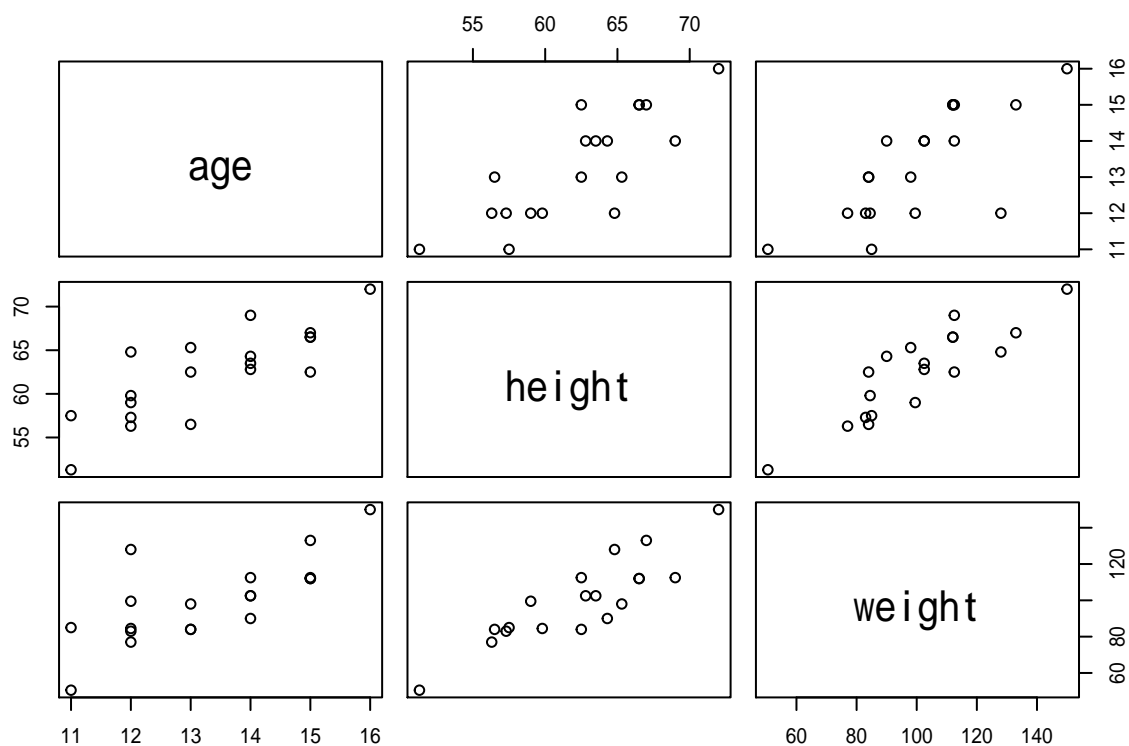
用气泡大小表现年龄，用颜色区分性别：

```
with(d.class,
      plot(height, weight,
            main=' 体重与身高关系',
            xlab=' 身高', ylab=' 体重',
            pch=16,
            col=ifelse(sex=='M', 'blue', 'red'),
            cex=1 + (age - min(age))
                     / (max(age) - min(age)))
```



用 `pairs()` 函数可以做散点图矩阵:

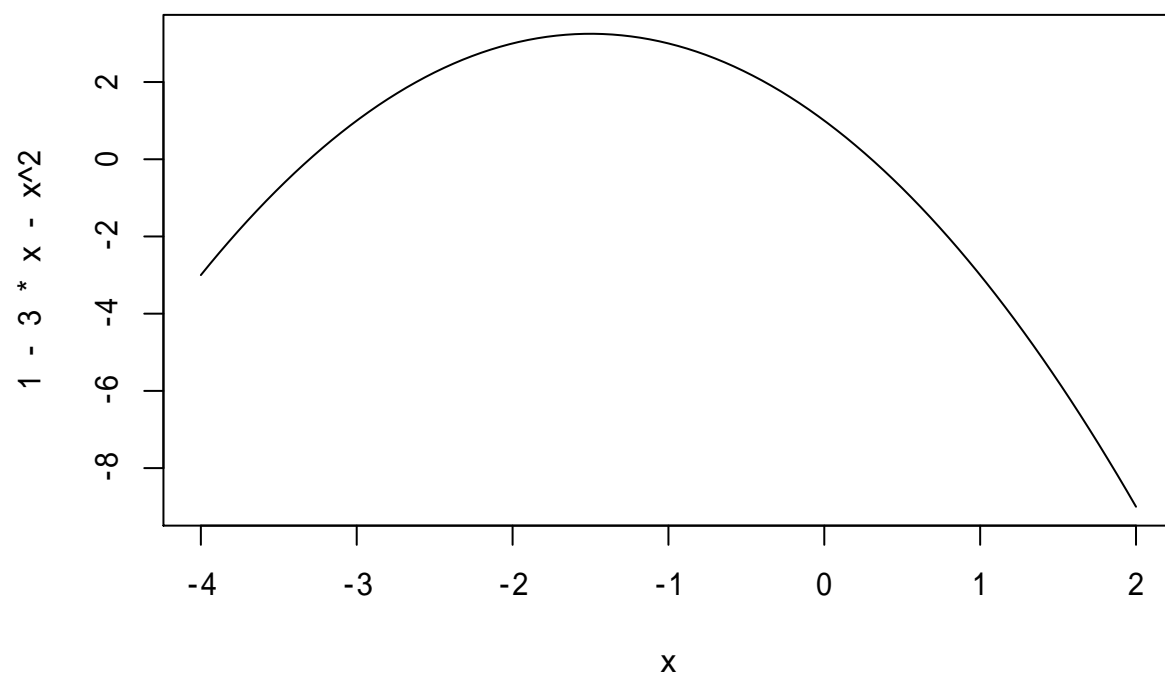
```
pairs(d.class[, c('age', 'height', 'weight')])
```



28.1.6 曲线图

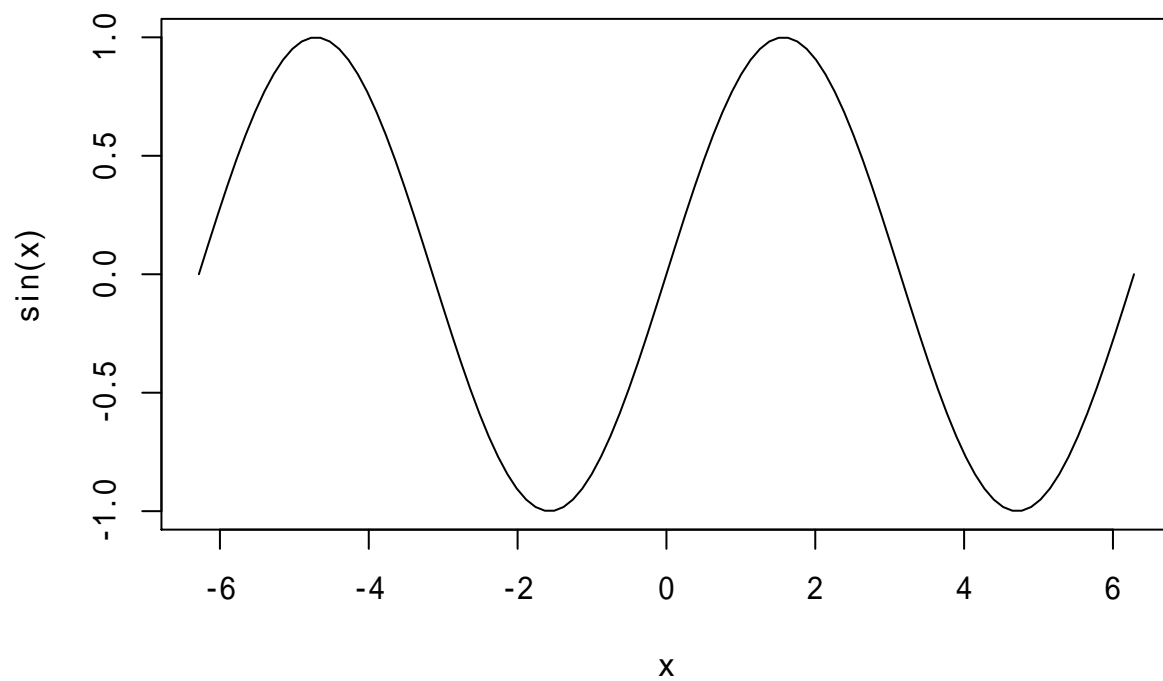
`curve()` 函数接受一个函数，或者一个以 `x` 为变量的表达式，以及曲线的自变量的左、右端点，绘制函数或者表达式的曲线图，如：

```
curve(1 - 3*x - x^2, -4, 2)
```

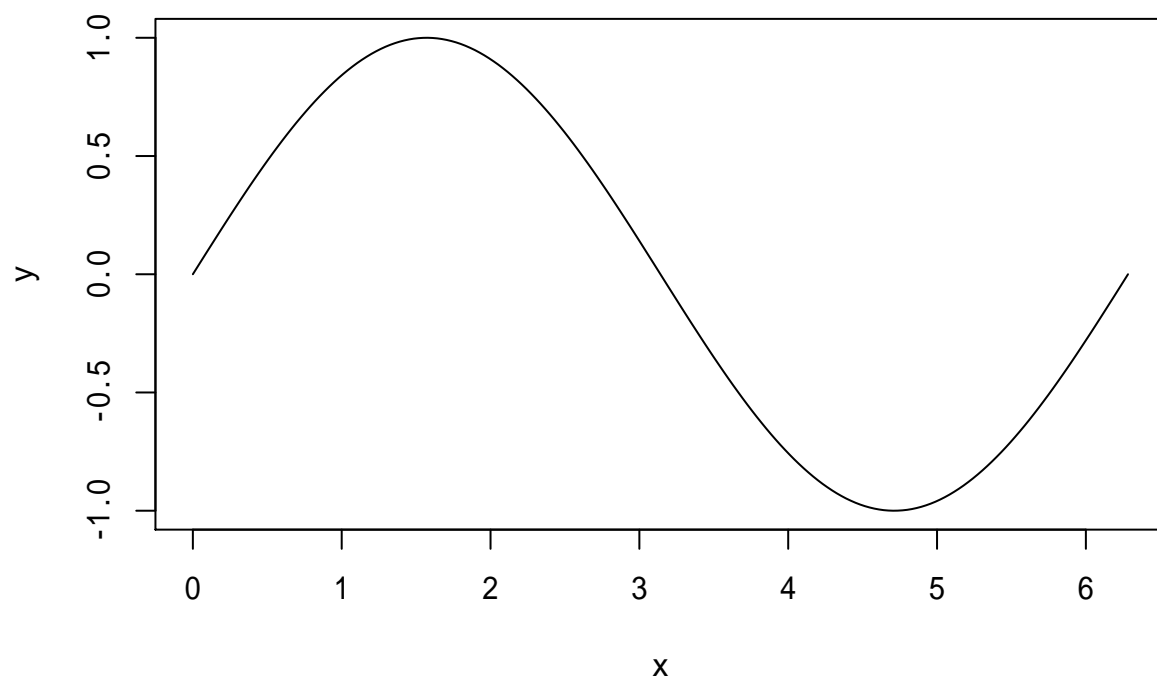
又如:

```
curve(sin, -2*pi, 2*pi)
```



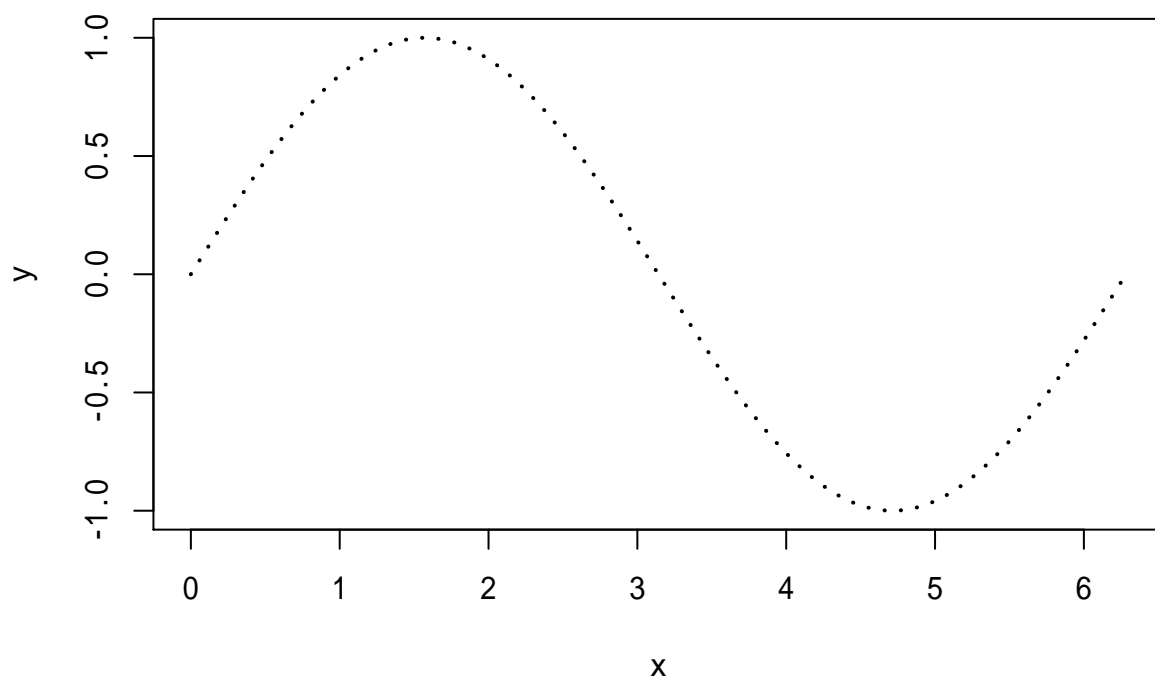
在 `plot` 函数中使用 `type='l'` 参数可以作曲线图，如

```
x <- seq(0, 2*pi, length=200)
y <- sin(x)
plot(x,y, type='l')
```



除了仍可以用 `main`, `xlab`, `ylab`, `col` 等参数外, 还可以用 `lwd` 指定线宽度, `lty` 指定虚线, 如

```
plot(x,y, type='l', lwd=2, lty=3)
```



28.1.7 三维图

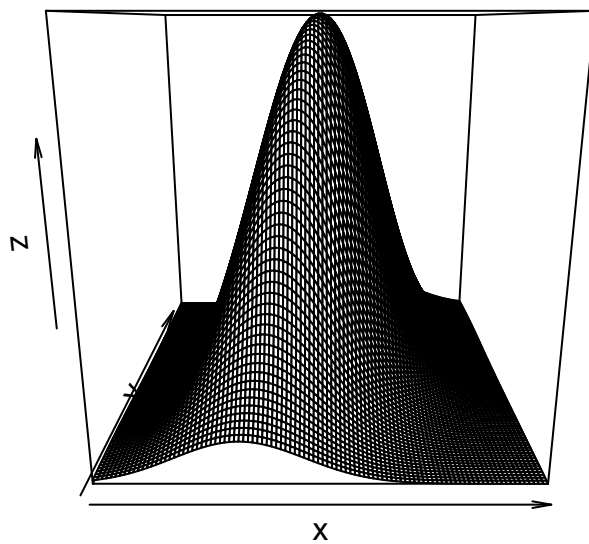
用 `persp` 函数作三维曲面图, `contour` 作等值线图, `image` 作色块图。坐标 `x` 和 `y` 构成一张平面网格, 数据 `z` 是包含 `z` 坐标的矩阵, 每行对应一个横坐标, 每列对应一个纵坐标。

下面的程序生成二元正态分布密度曲面数据:

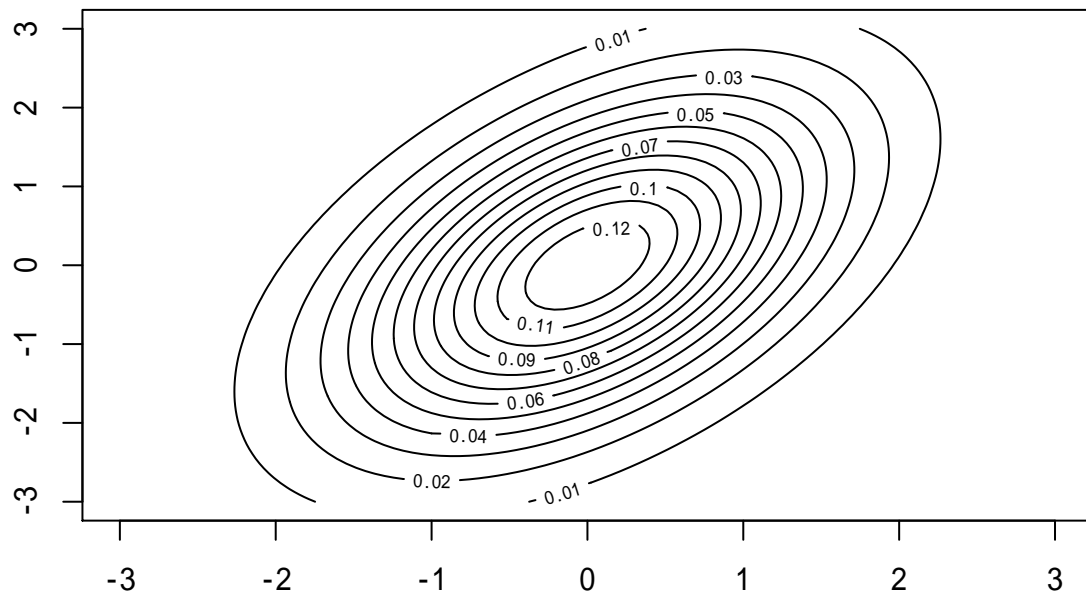
```
x <- seq(-3,3, length=100)
y <- x
f <- function(x,y,ssq1=1, ssq2=2, rho=0.5){
  det1 <- ssq1*ssq2*(1 - rho^2)
  s1 <- sqrt(ssq1)
  s2 <- sqrt(ssq2)
  1/(2*pi*sqrt(det1)) * exp(-0.5 / det1 * (
    ssq2*x^2 + ssq1*y^2 - 2*rho*s1*s2*x*y))
}
z <- outer(x, y, f)
```

作二元正态密度函数的三维曲面图、等高线图、色块图:

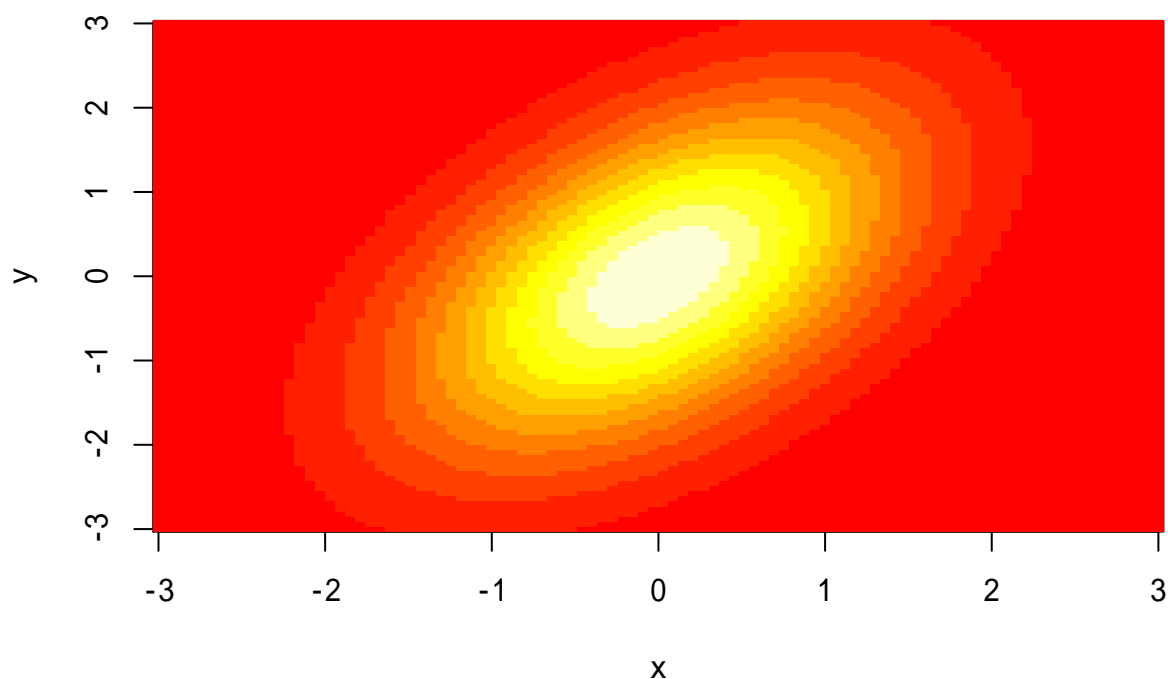
```
persp(x, y, z)
```



```
contour(x, y, z)
```



```
image(x, y, z)
```



28.1.8 动态三维图

rgl 包能制作动态的三维散点图与曲面图。

```
library(rgl)
```

iris 数据框包含了 3 种鸢尾花的各 50 个样品的测量值，测量值包括花萼长、宽，花瓣长、宽。用 rgl 的 plot3d() 作动态三维散点图如下：

```
with(iris, plot3d(
  Sepal.Length, Sepal.Width, Petal.Length,
  type="s", col=as.numeric(Species)))
```

这个图可以用鼠标拖动旋转。其中 type="s" 表示绘点符号是球体形状。还可选 "p"(点)、"l"(连线)、"h"(向 z=0 连线)。可以用 size= 指定大小倍数（缺省值为 3）。

用 rgl 的 persp3d() 函数作曲面图。如二元正态分布密度曲面：

```
x <- seq(-3,3, length=100)
y <- x
f <- function(x,y,ssq1=1, ssq2=2, rho=0.5){
```

```

det1 <- ssq1*ssq2*(1 - rho^2)
s1 <- sqrt(ssq1)
s2 <- sqrt(ssq2)
1/(2*pi*sqrt(det1)) * exp(-0.5 / det1 * (
  ssq2*x^2 + ssq1*y^2 - 2*rho*s1*s2*x*y))
}
z <- outer(x, y, f)
persp3d(x=x, y=y, z=z, col='red')

```

rgl 也有低级图形函数支持向已有图形添加物体、文字等，也支持并列多图。适当设置可以在 R Markdown 生成的 HTML 结果中动态显示三维图。

28.2 低级图形函数

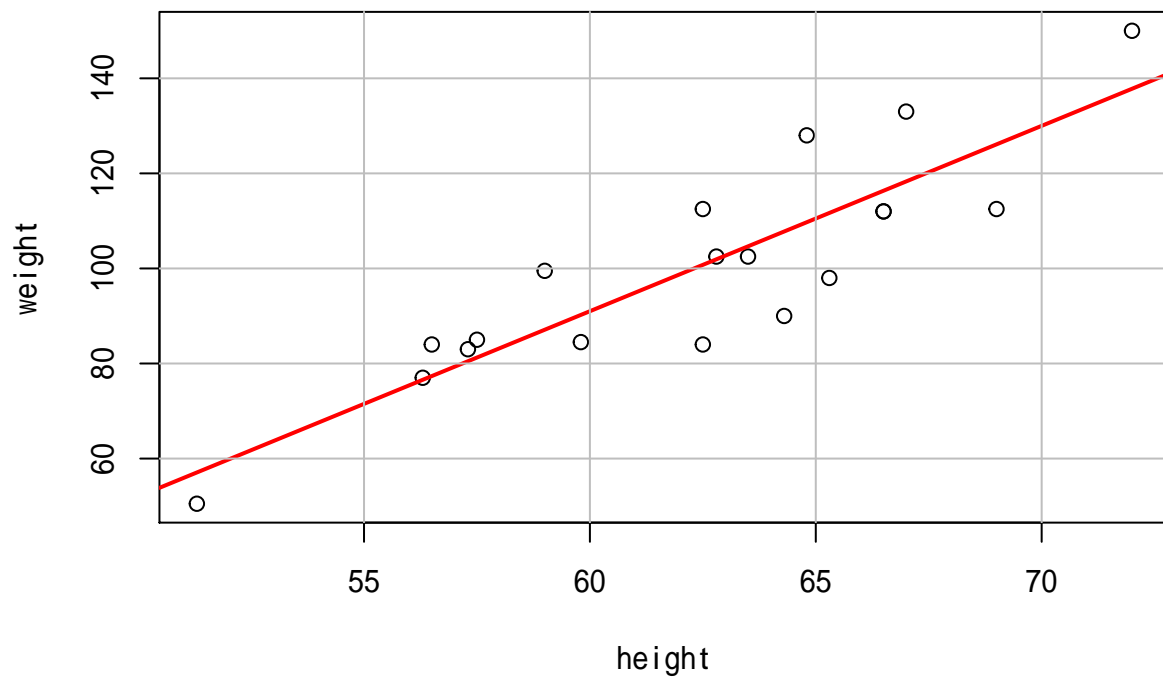
28.2.1 abline()

用 `abline` 函数在图中增加直线。可以指定截距和斜率，或为竖线指定横坐标 (用参数 `v`)，为水平线指定纵坐标 (用参数 `h`)。如

```

with(d.class, plot(height, weight))
abline(-143, 3.9, col="red", lwd=2)
abline(v=c(55,60,65,70), col="gray")
abline(h=c(60,80,100,120,140), col="gray")

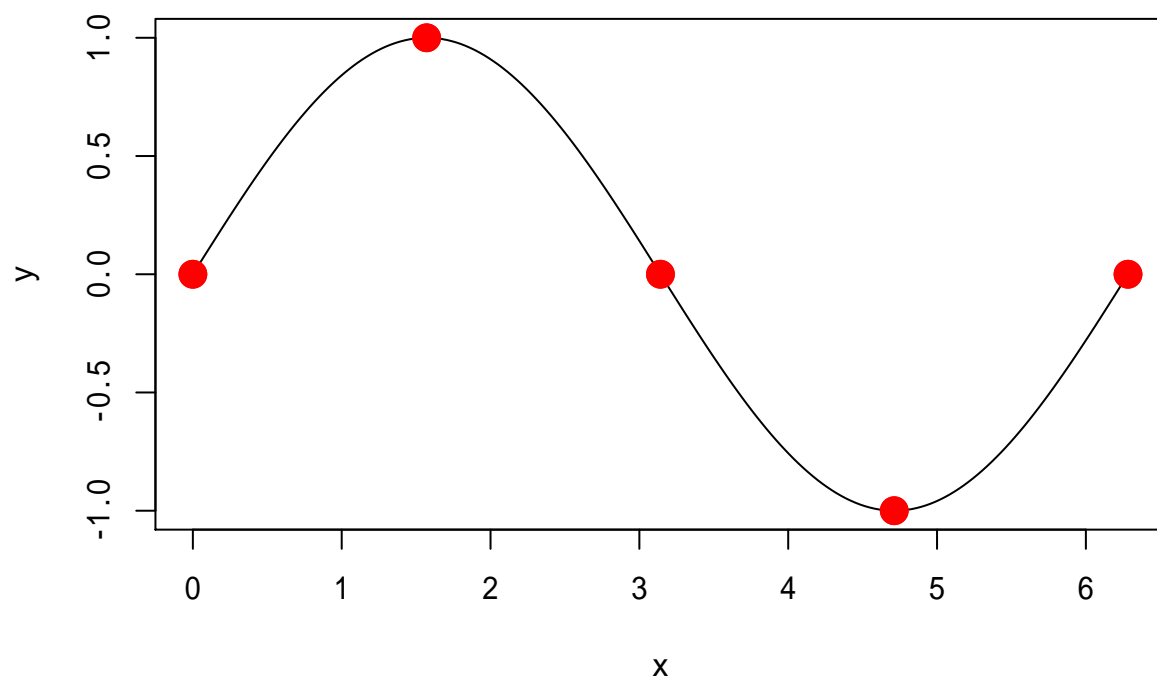
```

28.2.2 points()

用 `points` 函数增加散点，如：

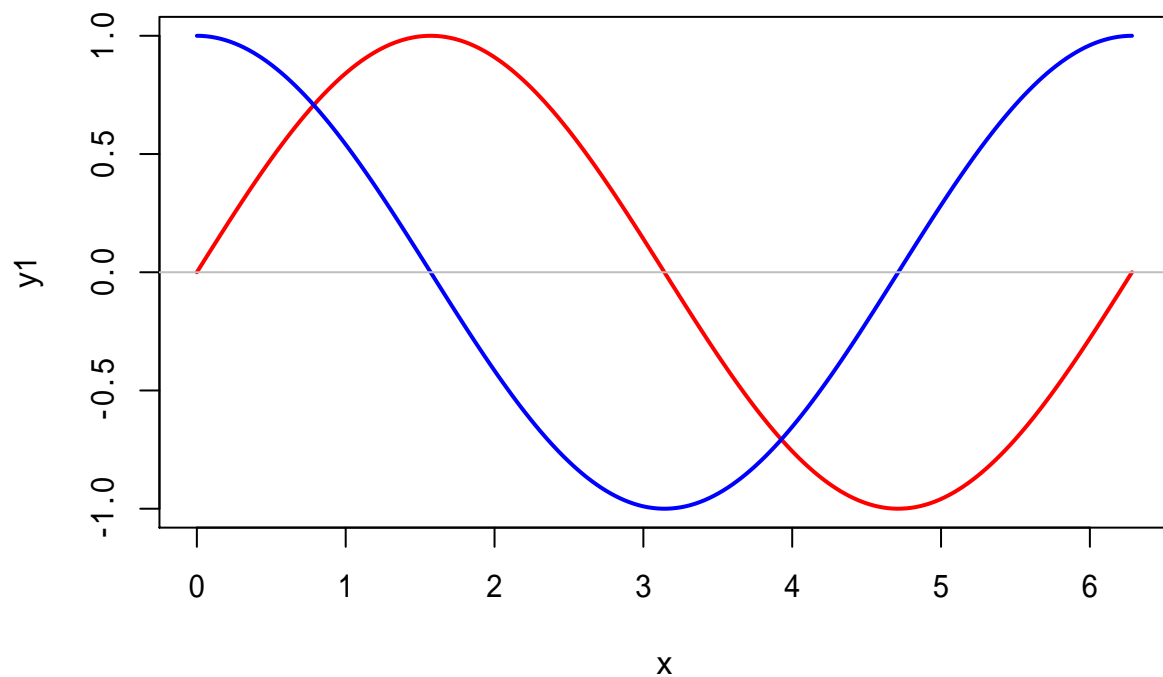
```
x <- seq(0, 2*pi, length=200)
y <- sin(x)
special <- list(x=(0:4)*pi/2, y=sin((0:4)*pi/2))
plot(x, y, type='l')
points(special$x, special$y,
       col="red", pch=16, cex=2)
points(special, col="red", pch=16, cex=2)
```



28.2.3 lines()

用 `lines` 函数增加曲线，如：

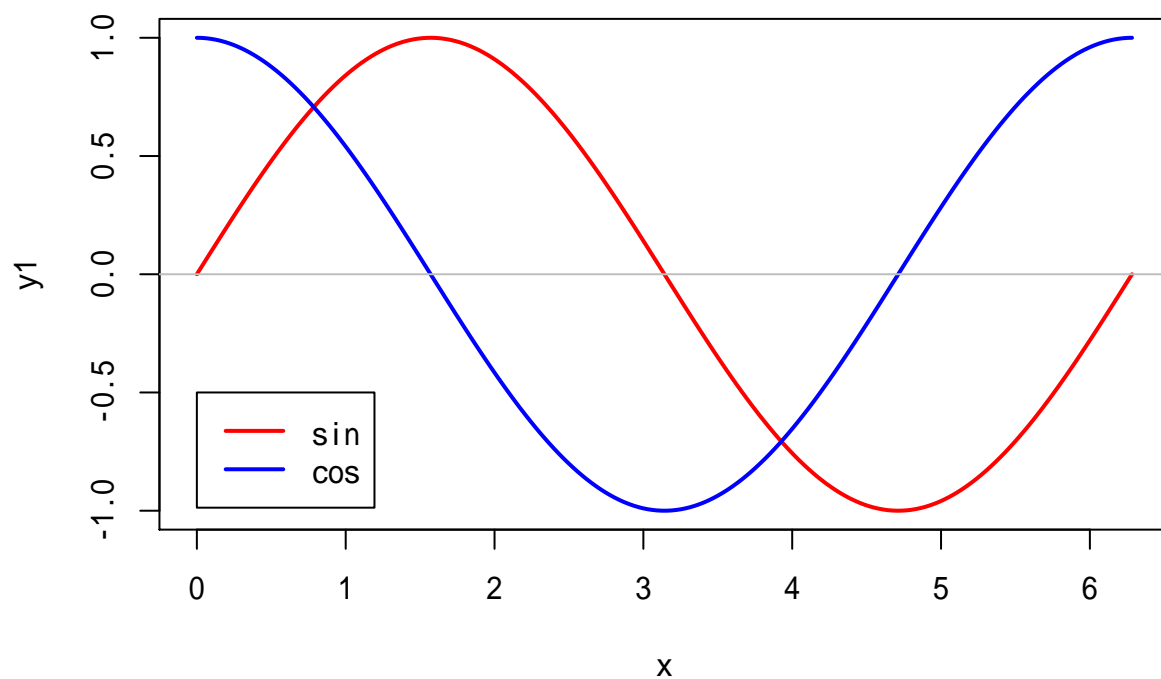
```
x <- seq(0, 2*pi, length=200)
y1 <- sin(x)
y2 <- cos(x)
plot(x, y1, type='l', lwd=2, col="red")
lines(x, y2, lwd=2, col="blue")
abline(h=0, col='gray')
```



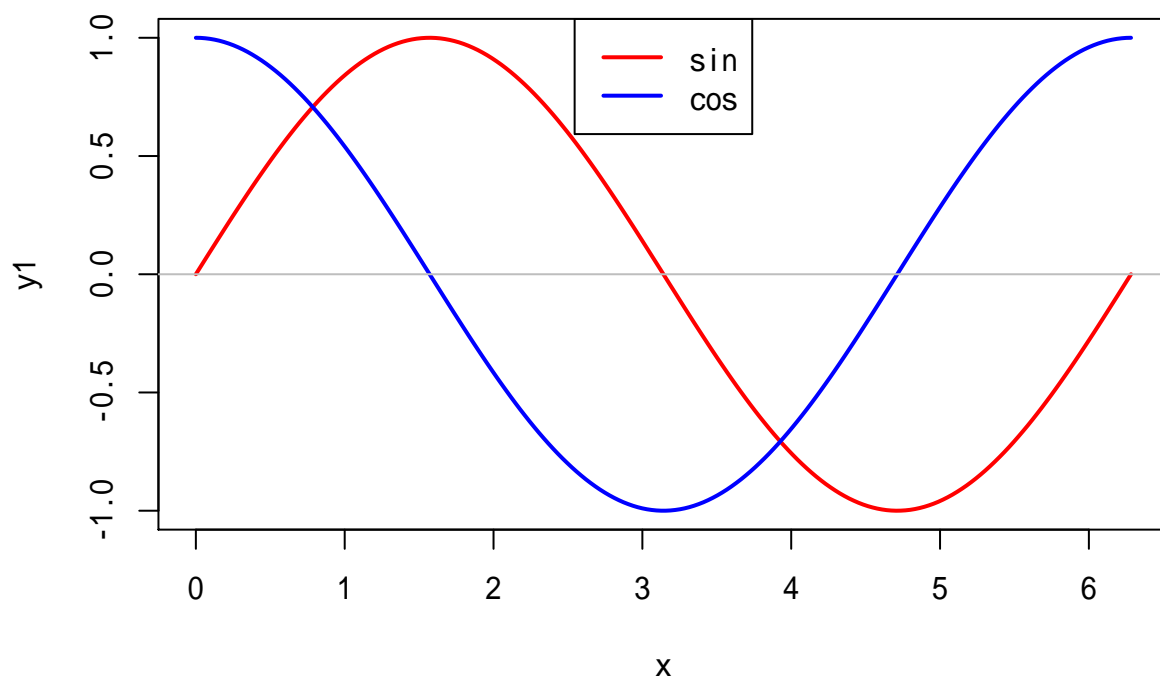
28.2.4 图例

可以用 `legend` 函数增加标注，如

```
x <- seq(0, 2*pi, length=200)
y1 <- sin(x)
y2 <- cos(x)
plot(x, y1, type='l', lwd=2, col="red")
lines(x, y2, lwd=2, col="blue")
abline(h=0, col='gray')
legend(0, -0.5, col=c("red", "blue"),
      lty=c(1,1), lwd=c(2,2),
      legend=c("sin", "cos"))
```



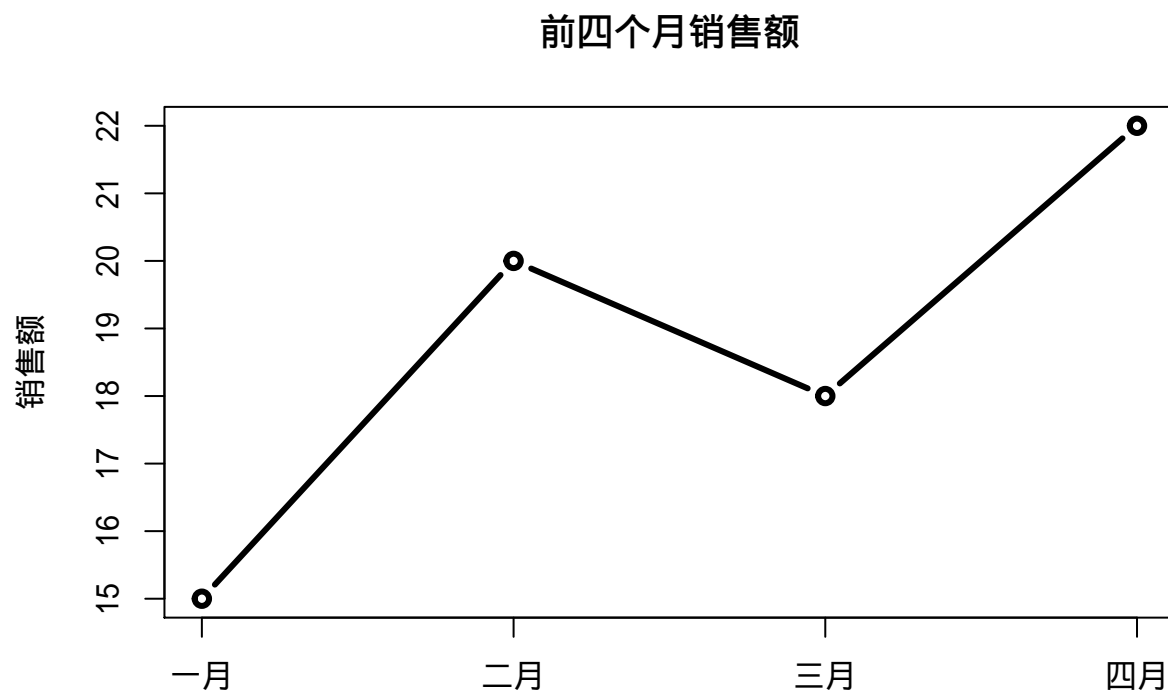
```
x <- seq(0, 2*pi, length=200)
y1 <- sin(x)
y2 <- cos(x)
plot(x, y1, type='l', lwd=2, col="red")
lines(x, y2, lwd=2, col="blue")
abline(h=0, col='gray')
legend('top', col=c("red", "blue"),
      lty=c(1,1), lwd=c(2,2),
      legend=c("sin", "cos"))
```



28.2.5 axis()

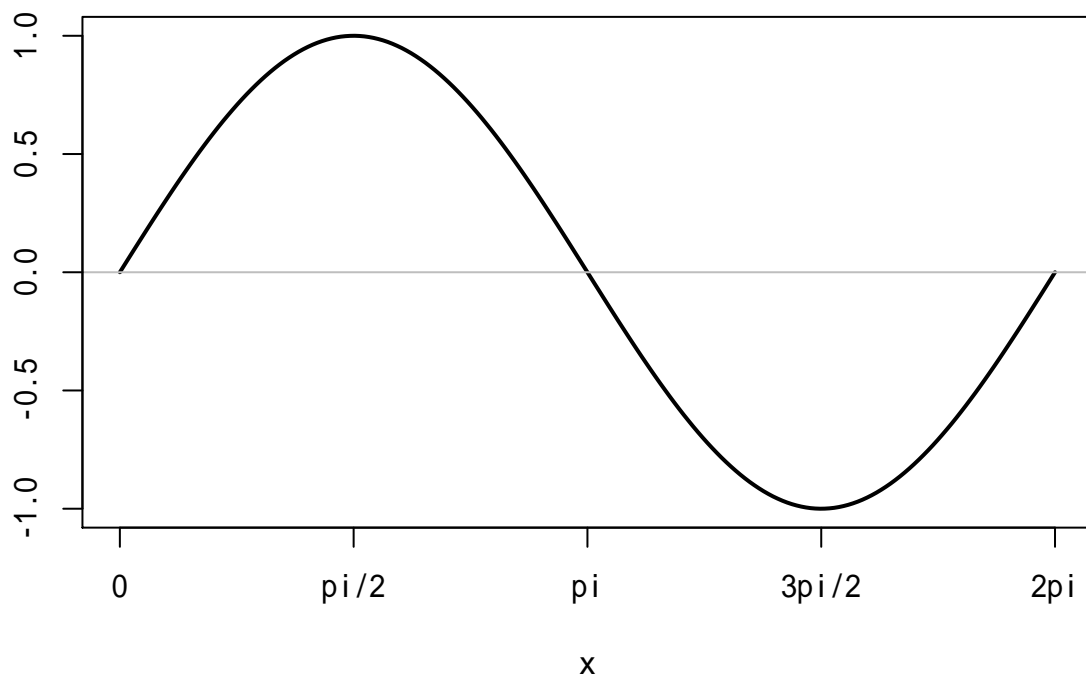
在 `plot()` 函数中用 `axes=FALSE` 可以取消自动的坐标轴。用 `box()` 函数画坐标边框。用 `axis` 函数单独绘制坐标轴。`axis` 的第一个参数取 1, 2, 3, 4, 分别表示横轴、纵轴、上方和右方。`axis` 的参数 `at` 为刻度线位置, `labels` 为标签。如

```
x <- c(' 一月'=15, ' 二月'=20,
       ' 三月'=18, ' 四月'=22)
plot(seq(along=x), x, axes=FALSE,
     type='b', lwd=3,
     main=' 前四个月销售额',
     xlab='', ylab=' 销售额')
box(); axis(2)
axis(1, at=seq(along=x), labels=names(x))
```



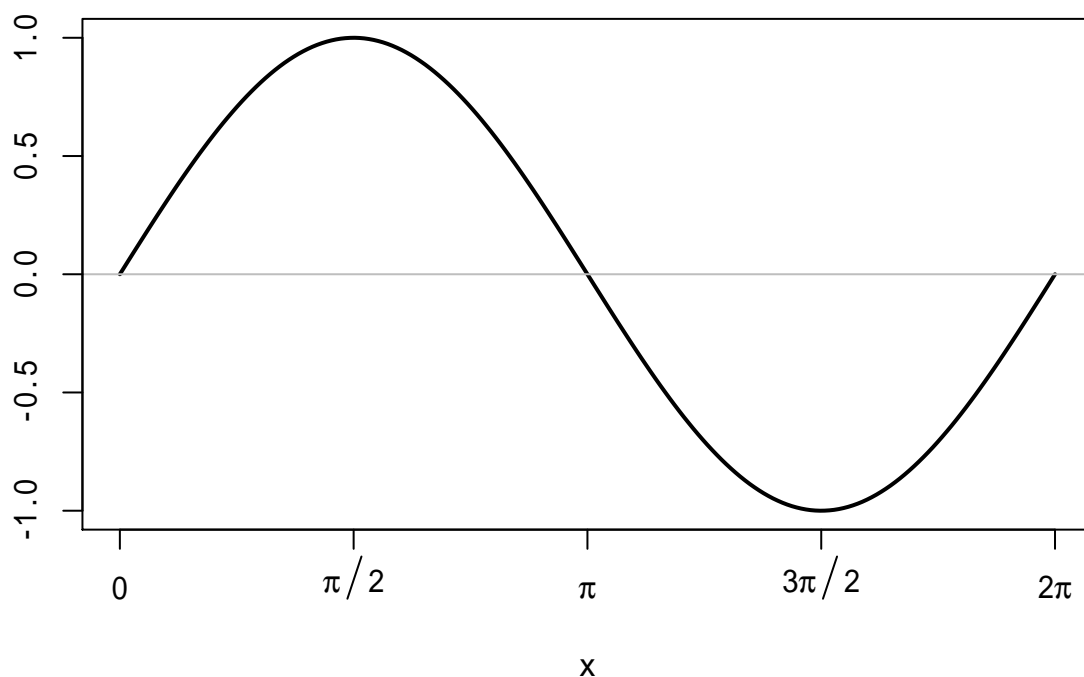
R 基本绘图支持少量的数学公式显示功能，如不用数学符号时：

```
x <- seq(0, 2*pi, length=200)
y1 <- sin(x)
plot(x, y1, type='l', lwd=2,
      axes=FALSE,
      xlab='x', ylab='')
abline(h=0, col='gray')
box()
axis(2)
axis(1, at=(0:4)/2*pi,
      labels=c('0', 'pi/2', 'pi', '3pi/2', '2pi'))
```



使用数学符号时:

```
x <- seq(0, 2*pi, length=200)
y1 <- sin(x)
plot(x, y1, type='l', lwd=2,
     axes=FALSE,
     xlab='x', ylab='')
abline(h=0, col='gray')
box()
axis(2)
axis(1, at=(0:4)/2*pi,
     labels=c(0, expression(pi/2),
              expression(pi), expression(3*pi/2),
              expression(2*pi)))
```



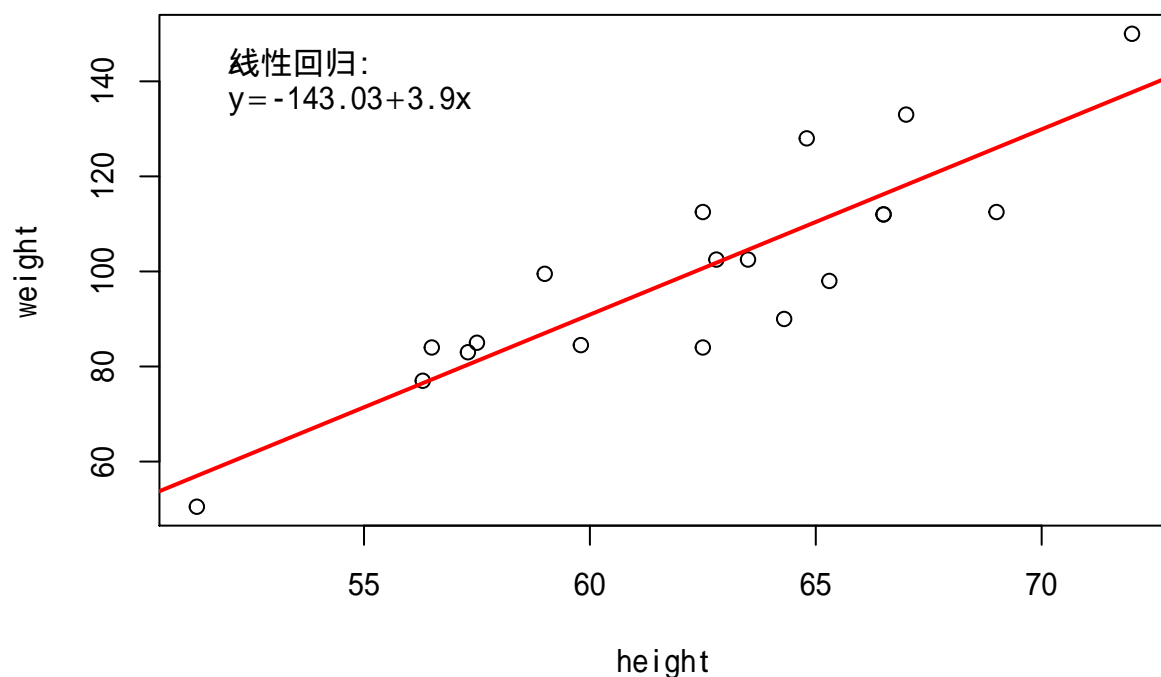
绘图中使用数学符号的演示：

```
demo(plotmath)
```

28.2.6 text()

`text()` 在坐标区域内添加文字。`mtext()` 在边空处添加文字。如

```
with(d.class, plot(height, weight))
lm1 <- lm(weight ~ height, data=d.class)
abline(lm1, col='red', lwd=2)
a <- coef(lm1)[1]
b <- coef(lm1)[2]
text(52, 145, adj=0, ' 线性回归:')
text(52, 140, adj=0,
      substitute(hat(y) == a + b*x,
                  list(a=round(coef(lm1)[1], 2),
                       b=round(coef(lm1)[2], 2))))
```

28.2.7 locator() 和 identify()

`locator()` 函数在执行时等待用户在图形的坐标区域内点击并返回点击处的坐标。可以用参数 `n` 指定要点击的点的个数。不指定个数则需要用右键菜单退出。这个函数也可以用来要求用户点击以进行到下一图形。如

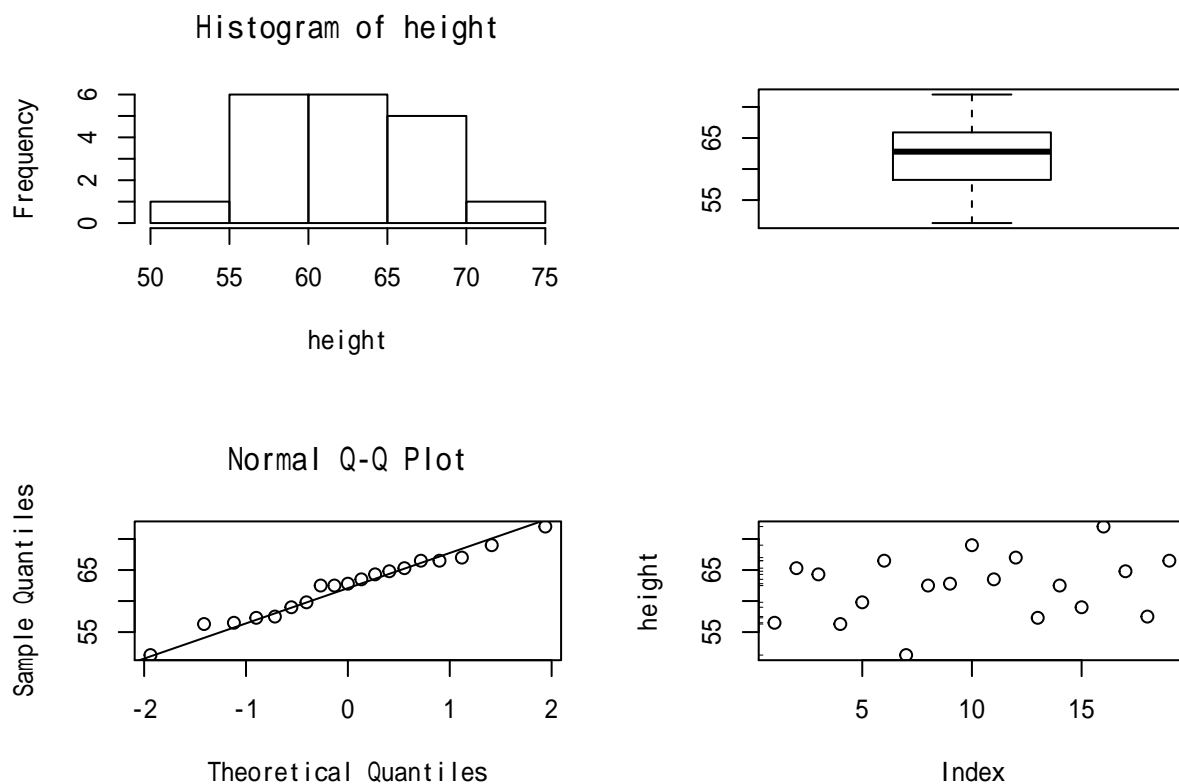
```
x <- seq(0, 2*pi, length=200)
y1 <- sin(x); y2 <- cos(x)
plot(x, y1, type='l',
     col="red")
lines(x, y2, col="blue")
legend(locator(1), col=c("red", "blue"),
      lty=c(1,1), legend=c("sin", "cos"))
```

`identify()` 可以识别点击处的点并标注标签。

28.3 图形参数

用图形参数可以选择点的形状、颜色、线型、粗细、坐标轴做法、边空、一页多图等。有些参数直接用在绘图函数内，如 `plot` 函数可以用 `pch`、`col`、`cex`、`lty`、`lwd` 等参数。有些图形参数必须使用 `par()` 函数指定。`par` 函数指定图形参数并返回原来的参数值，所以在修改参数值作图后通常应该恢复原始参数值，做法如

```
opar <- par(mfrow=c(2,2))
with(d.class, {hist(height);
  boxplot(height);
  qqnorm(height); qqline(height);
  plot(height); rug(height,side=2)})
```



```
par(opar)
```

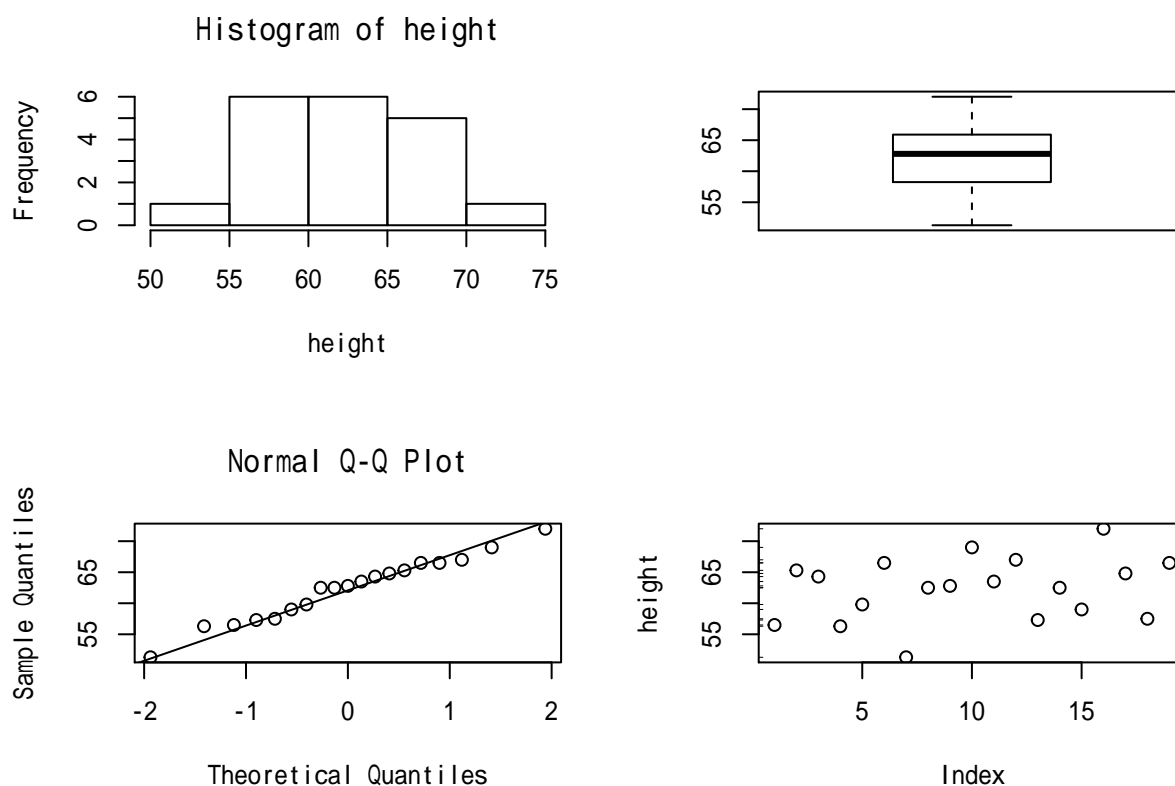
在函数内，可以在函数开头修改了图形参数后，用 `on.exit()` 函数将恢复原始图形参数作为函数退出前必须完成的任务，如

```
f <- function(){
  opar <- par(mfrow=c(2,2)); on.exit(par(opar))
  with(
```

```

d.class,
{hist(height);
 boxplot(height);
 qqnorm(height); qqline(height);
 plot(height); rug(height,side=2)
})
}
f()

```



28.3.1 例子：用图形参数解决 barplot 图形横坐标值过宽

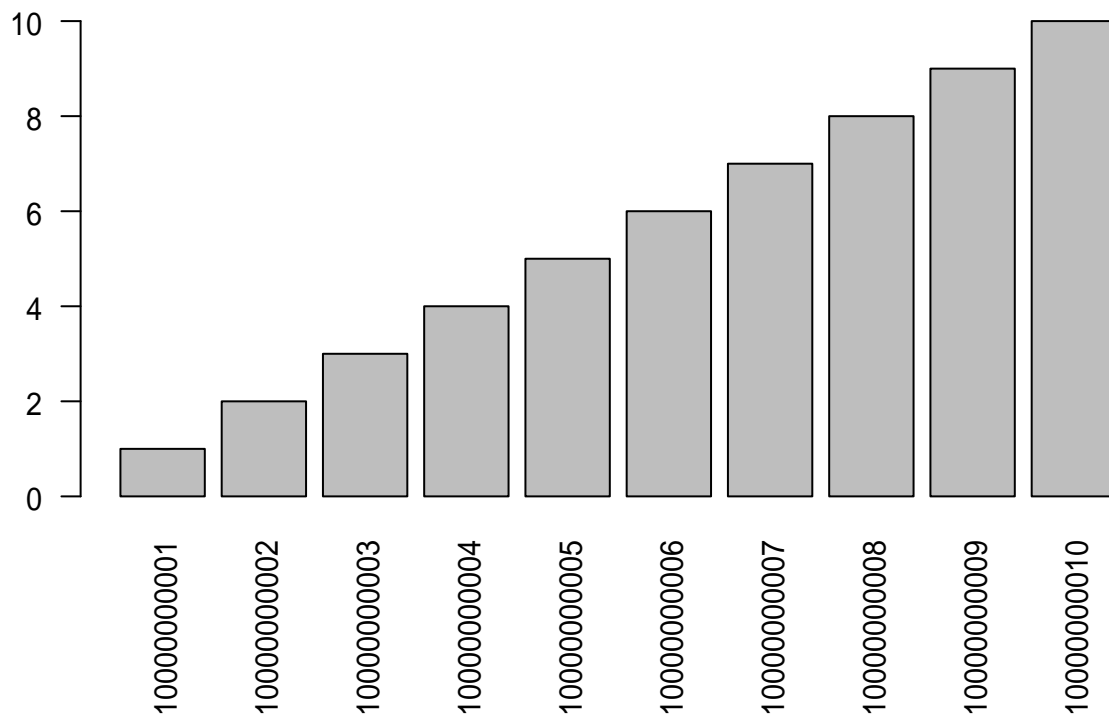
barplot 的横坐标标注太宽时，自动将某些标注省略。用 `las=2` 指定坐标轴刻度标签垂直于坐标轴，这样 x 轴的刻度值就变成了纵向的。注意使用 `mar` 参数增加横坐标边空大小。例如

```

f <- function(){
  opar <- par(mar=c(8, 4, 2, 0.5)); on.exit(par(opar))
  x <- 1:10
  names(x) <- paste(10000000000 + (1:10))
  barplot(x, las=2)
}

```

```
}  
f()
```



图形参数可以分为如下四类

- 图形元素控制;
- 坐标轴与坐标刻度;
- 图形边空;
- 一页多图。

28.3.2 图形元素控制

- `pch=16` 参数。散点符号, 取 0 ~ 18 的数。
- `lty=2` 参数。线型, 1 为实线, 从 2 开始为各种虚线。
- `lwd=2` 参数, 线的粗细, 标准粗细为 1。
- `col=red` 参数, 颜色, 可以是数字 1 ~ 8, 或颜色名字符串如 `red`, `blue` 等。用 `colors()` 函数查询有名字的颜色。用 `rainbow(n)` 函数产生连续变化的颜色。
- `font=2` 参数, 字体, 一般 `font=1` 是正体, 2 是粗体, 3 是斜体, 4 是粗斜体。

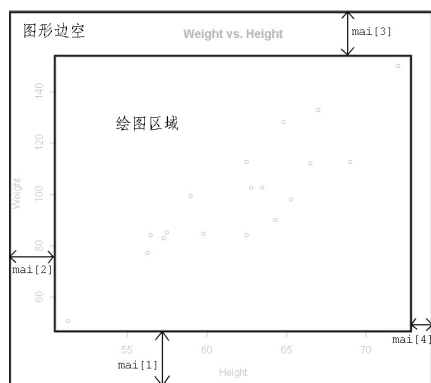


图 28.1:

- `adj=-0.1` 指定文本相对于给定坐标的对齐方式。取 0 表示左对齐, 取 1 表示右对齐, 取 0.5 表示居中。此参数的值实际代表的是出现在给定坐标左边的文本的比例。
- `cex=1.5` 绘点符号大小倍数, 基本值为 1。

28.3.3 坐标轴与坐标刻度

- `mgp=c(3,1,0)` 坐标轴的标签、刻度值、坐标轴线到实际的坐标轴位置的距离, 以行高为单位。经常用来缩小坐标轴所占的空间, 如 `mgp=c(1.5, 0.5, 0)`。
- `lab=c(5,7,12)` 提供刻度线多少的建议, 第一个数为 x 轴刻度线个数, 第二个数为 y 轴刻度线个数, 第三个数是坐标刻度标签的字符宽度。
- `las=1` 坐标刻度标签的方向。0 表示总是平行于坐标轴, 1 表示总是水平, 2 表示总是垂直于坐标轴。
- `tck=0.01` 坐标轴刻度线长度, 以绘图区域大小为单位 1。
- `xaxs=s, yaxs=e`: 控制 x 轴和 y 轴标刻度的方法。

取 `s`(即 standard) 或 `e`(即 extended) 的时候数据范围控制在最小刻度和最大刻度之间。取 `e` 时如果有数据点十分靠近边缘轴的范围会略微扩大。

取值为 `i` (即 internal) 或 `r` (此为缺省) 使得刻度线都落在数据范围内部, 而 `r` 方式所留的边空较小。

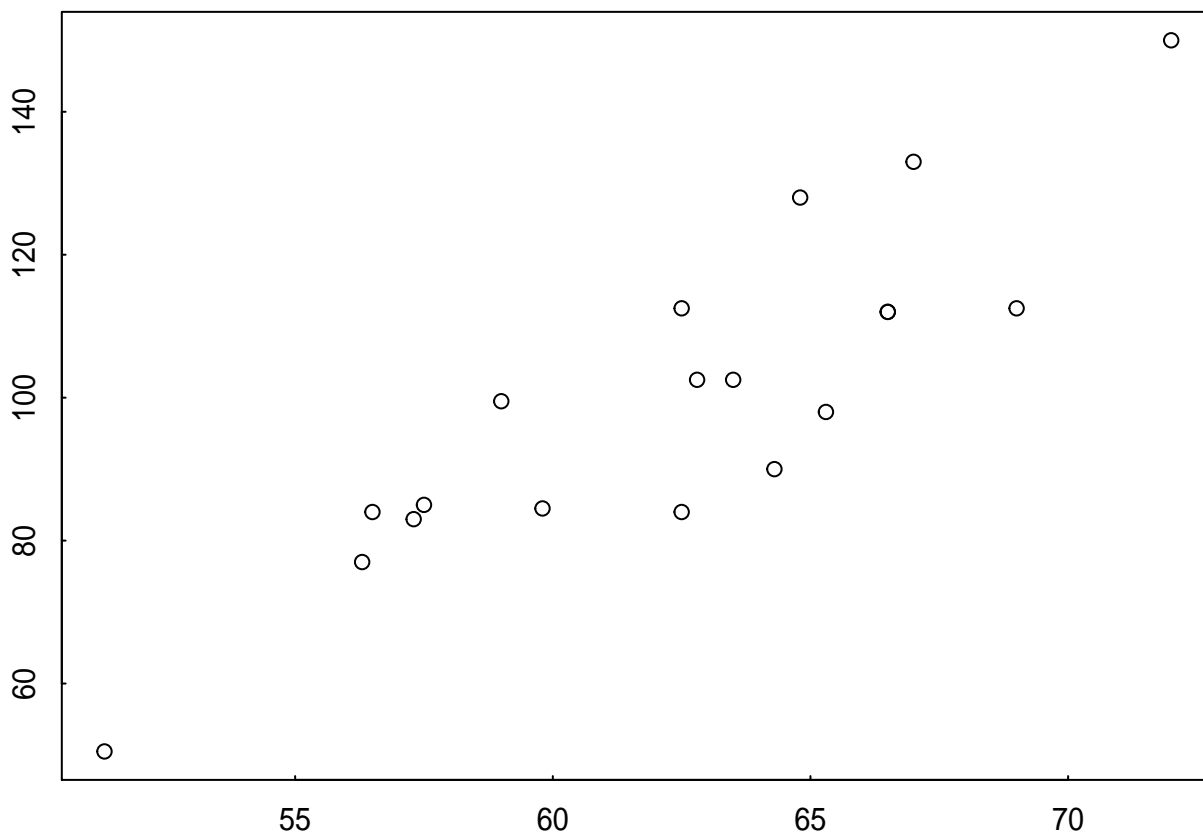
取值设为 `d` 时会锁定此坐标轴, 后续的图形都使用与它完全相同的坐标轴, 这在要生成一系列可比较的图形的时候是有用的。要解除锁定需要把这个图形参数设为其它值。

28.3.4 图形边空

一个单独的图由绘图区域 (绘图的点、线等画在这个区域中) 和包围绘图区域的边空组成, 边空中可以包含坐标轴标签、坐标轴刻度标签、标题、小标题等, 绘图区域一般被坐标轴包围。

边空的大小由 `mai` 参数或 `mar` 参数控制, 它们都是四个元素的向量, 分别规定下方、左方、上方、右方的边空大小, 其中 `mai` 取值的单位是英寸, 而 `mar` 的取值单位是文本行高度。例如:

```
opar <- par(mar=c(2,2,0.5,0.5),  
            mgp=c(0.5, 0.5, 0), tck=0.005)  
with(d.class, plot(height, weight,  
                   xlab='', ylab=''))
```



```
par(opar)
```

28.3.5 一页多图

R 可以在同一页面开若干个按行、列排列的窗格, 在每个窗格中可以作一幅图。每个图有自己的内边空, 而所有图的外面可以包一个“外边空”。

一页多图用 `mfrow` 参数或 `mfcol` 参数规定。用 `oma` 指定四个外边空的行数。用 `mtext` 加 `outer=T` 指定在外边空添加文本。如果没有 `outer=T` 则在内边空添加文本。如

```
opar <- par(mfrow=c(2,2),  
            oma=c(0,0,2,0))  
with(d.class, {hist(height);
```

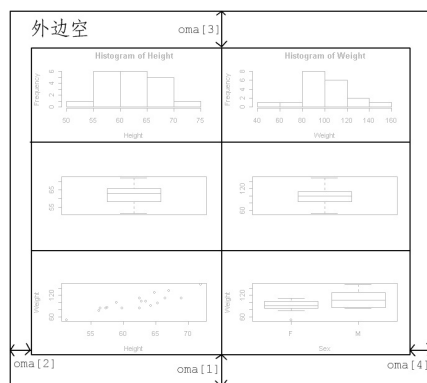


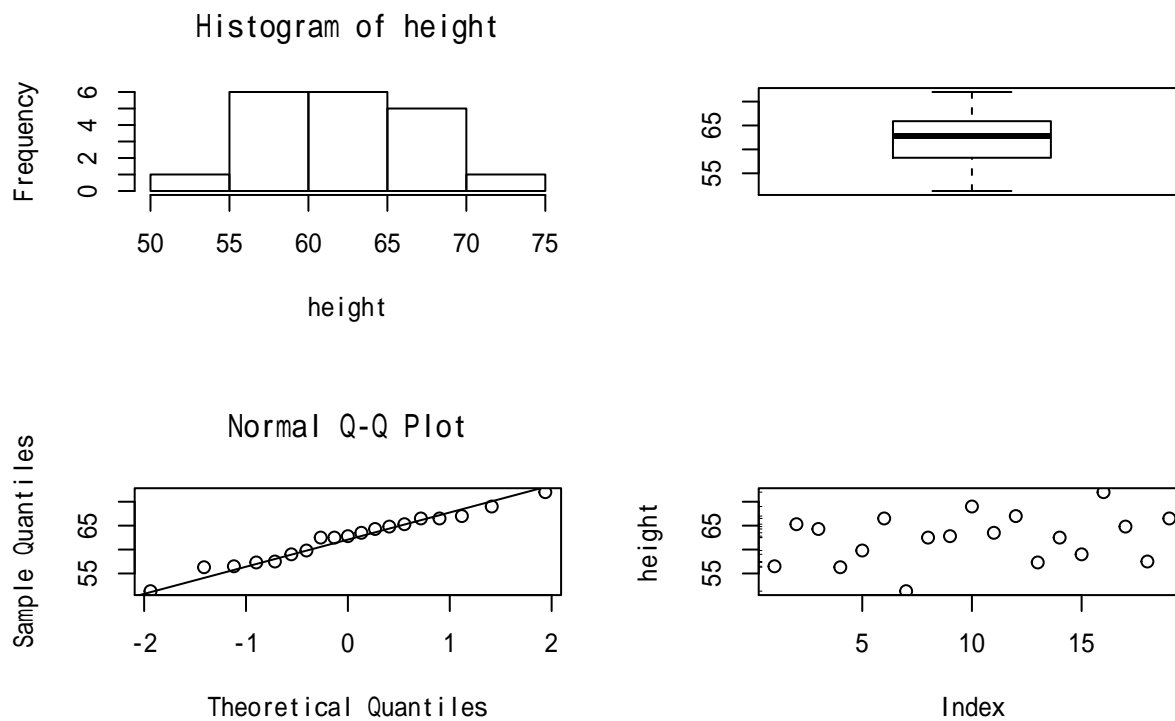
图 28.2:

```

boxplot(height);
qqnorm(height); qqline(height);
plot(height); rug(height,side=2)})
mtext(side=3, text=' 身高分布', cex=2, outer=T)

```

身高分布



```

par(opar)

```

28.4 图形输出

只要启用了高级绘图函数会自动选用当前绘图设备，缺省为屏幕窗口。

28.4.1 PDF 输出

用 `pdf` 函数可以指定输出到 PDF 文件。如

```
pdf(file='fig-hw.pdf', height=10/2.54,
    width=10/2.54, family='GB1')
with(d.class, plot(height, weight,
    main=' 体重与身高关系'))
dev.off()
```

用 `dev.off()` 关闭当前设备并生成输出文件（如果是屏幕窗口则没有保存结果）。

28.4.2 PNG 输出

```
png(file='fig-hw.png', height=1000, width=1000)
with(d.class, plot(height, weight,
    main=' 体重与身高关系'))
dev.off()
```

类似地，用 `jpeg()` 函数启用 JPEG 图形设备，用 `bmp()` 函数启用 BMP 图形设备，用 `postscript()` 函数启用 PostScript 图形设备。

28.5 包含多种中文字体的图形

为了使用 MS Windows 系统字体，一种办法是安装 `showtext` 包。该包替换画图时的添加文本函数命令，把文本内容替换成多边形（PDF 或 PS 图）或点阵（点阵图）。

需要的工作：

- 查看 Windows 的 `font` 目录内容，看文件名与字体名的对应关系。下面的程序中的列表是我的中文 Windows 10 的部分中文字体。
- 找到自己希望使用的中文字体的文件名。
- 用 `font.add()` 命令，增加一套自定义字体 `family`，一套中可以指定四种：常规 (`regular`)，粗体 (`bold`)，斜体 (`italic`)，粗斜体 (`bolditalic`)
- 程序中调入 `showtext` 包并运行 `showtext.auto()` 命令，这个命令使得文本命令采用 `showtext` 包
- 用 `par(family=)` 指定自定义的字体 `family`。
- 作图（主要是 PDF）。关闭图形设备。


```

test.chinese <- function(){
  require(showtext); showtext.auto()

  ## 建立文件名到字体名对照表
  fmap <- c(
    'msyh'=' 微软雅黑常规',
    'msyhbd'=' 微软雅黑粗体',
    'msyhl'=' 微软雅黑细体',
    'simsun'=' 宋体',
    'simfang'=' 仿宋',
    'simkai'=' 楷体',
    'simhei'=' 黑体',
    'SIMLI'=' 隶书',
    'SIMYOU'=' 幼圆',
    'STSONG'=' 华文宋体',
    'STZHONGS'=' 华文中宋',
    'STFANGSO'=' 华文仿宋',
    'STKAITI'=' 华文楷体',
    'STXIHEI'=' 华文细黑',
    'STLITI'=' 华文隶书',
    'STXINGKA'=' 华文行楷',
    'STXINWEI'=' 华文新魏',
    'STCAIYUN'=' 华文彩云',
    'STHUPO'=' 华文琥珀'
  )

  fmapr <- names(fmap); names(fmapr) <- unname(fmap)
  cat('==== 字体文件名与字体名称对应:\n')
  print(fmap)
  cat('==== 字体名与字体文件名对应:\n')
  print(fmapr)

  ## 找到某个字体的字体文件
  ## font.name 是字体名称
  find.font <- function(font.name){
    fname <- fmapr[font.name]
    flist0 <- font.files()
    flist1 <- sapply(strsplit(flist0, '[.]'), function(it) it[1])
    flist0[flist1==fname]
  }

```

```
ff1 <- find.font(' 宋体')
ff2 <- find.font(' 黑体')
ff3 <- find.font(' 仿宋')
ff4 <- find.font(' 隶书');

font.add('cjk4',
        regular=ff1,
        bold=ff2,
        italic=ff3,
        bolditalic=ff4)
##browser()

pdf('test-chinese.pdf'); on.exit(dev.off())
par(family='cjk4')

plot(c(0,1), c(0,1), type='n',
     axes=FALSE, xlab='', ylab='')
text(0.1, 0.9, ' 正体', font=1)
text(0.1, 0.8, ' 粗体', font=2)
text(0.1, 0.7, ' 斜体', font=3)
text(0.1, 0.6, ' 粗斜体', font=4)
}
test.chinese()
```

注意：图形参数 font=1 表示正体，font=2 表示粗体，font=3 表示斜体，font=4 表示粗斜体。

Part VII

统计分析

Chapter 29

R 初等统计分析

这一部分讲授如何用 R 进行统计分析，包括基本概括统计和探索性数据分析，置信区间和假设检验，回归分析与各种回归方法，广义线性模型，非线性回归与平滑，判别树和回归树，等等。

内容待完成。

主要参考书：

(Venables and Ripley, 2002)

Part VIII

用 Rcpp 连接 C++ 代码

Chapter 30

Rcpp 介绍

为了提高 R 程序的运行效率，可以尽量使用向量化编程，减少循环，尽量使用内建函数。对于效率的瓶颈，尤其是设计迭代算法时，可以采用编译代码，而 Rcpp 扩展包可以很容易地将 C++ 代码连接到 R 程序中，并且支持在 C++ 中使用类似于 R 的数据类型。

没有学过 C++ 语言的读者，如果需要编写比较独立的不太依赖于 R 的已有功能的算法，可以考虑学习使用 Julia 语言编写。Julia 语言是最近几年才发明的一种比 R 更现代、理念更先进的程序语言，其运行效率一般比 R 高得多，经常接近编译代码的效率。

Rcpp 可以很容易地把 C++ 代码与 R 程序连接在一起，可以从 R 中直接调用 C++ 代码而不需要用户关心那些繁琐的编译、链接、接口问题。可以在 R 数据类型和 C++ 数据类型之间容易地转换。

因为涉及到编译，所以 Rcpp 比一般的扩展包有更多的安装要求：除了要安装 Rcpp 包之外，MS Windows 用户还需要安装 RTools 包，这是用于 C, C++, Fortran 程序编译链接的开发工具包，是自由软件。用户的应用程序路径 (PATH) 中必须有 RTools 包可执行程序的路径 (安装 RTools 可以自动设置)。如果 Rcpp 不能找到编译器，可以把编译器安装到 Rcpp 默认的位置。Mac 操作系统和 Linux 操作系统中可以用操作系统自带的编译器。

Rcpp 支持把 C++ 代码写在 R 源程序文件内，执行时自动编译连接调用；也支持把 C++ 代码保存在单独的源文件中，执行 R 程序时自动编译连接调用；对较复杂的问题，应制作 R 扩展包，利用构建 R 扩展包的方法实现 C++ 代码的编译连接，这时接口部分也可以借助 Rcpp 属性功能或模块功能完成。

30.1 Rcpp 的用途

- 把已经用 R 代码完成的程序中运行速度瓶颈部分改写成 C++ 代码，提高运行效率。
- 对于 C++ 或 C 程序源代码或二进制代码提供的函数库，可以用 Rcpp 编写 C++ 界面程序进行 R 与 C++ 程序的输入、输出的传送，并在 C++ 界面程序中调用外来的函数库。

- 注意，用 Rcpp 编写 C++ 程序，不利于把程序脱离 R 运行或被其他的 C++ 程序调用。当然，可以只把 Rcpp 作为界面，主要的算法引擎完全不用 Rcpp 的数据类型。
- RInside 扩展包支持把 R 嵌入到 C++ 主程序中。

30.2 Rcpp 入门样例

30.2.1 用 cppFunction() 转换简单的 C++ 函数—Fibonacci 例子

考虑用 C++ 程序计算 Fibonacci 数的问题。Fibonacci 数满足 $f_0 = 0, f_1 = 1, f_t = f_{t-1} + f_{t-2}$ 。

可以使用如下 R 代码，其中有一部分 C++ 代码，用 `cppFunction` 转换成了 R 可以调用的同名 R 函数。

```
cppFunction(code='
  int fibonacci(const int x){
    if(x < 2) return x;
    else
      return ( fibonacci(x-1) + fibonacci(x-2) );
  }
')
print(fibonacci(5))
```

编译、链接、导入是在后台由 Rcpp 控制自动进行的，不需要用户去设置编译环境，也不需要用户执行编译、链接、导入 R 的工作。在没有修改 C++ 程序时，同一 R 会话期间重新运行不必重新编译。

上面的 Fibonacci 函数仅接受标量数值作为输入，不允许向量输入。

从算法角度评价，这个算法是极其低效的，其算法规模是 $O(2^n)$ ， n 是自变量值。

30.2.2 用 sourceCpp() 转换 C++ 程序—正负交替迭代例子

设 $x_t, t = 1, 2, \dots, n$ 保存在 R 向量 `x` 中，令

$$y_1 = x_1$$

$$y_t = (-1)^t y_{t-1} + x_t, \quad t = 2, \dots, n$$

希望用 C++ 函数对输入序列 `x` 计算输出 `y`，并用 R 调用这样的函数。

下面的程序用 R 函数 `sourceCpp()` 把保存在 R 字符串中的 C++ 代码编译并转换为同名的 R 函数。

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;
```

```

//[[Rcpp::export]]
NumericVector iters(NumericVector x){
  int n = x.size();
  NumericVector y(n);

  y[0] = x[0];
  double sign=-1;
  for(int t=1; t<n; t++){
    sign *= -1;
    y[t] = sign*y[t-1] + x[t];
  }

  return y;
}
')
print(iters(1:5))

```

这个例子说明 C++ 程序可以直接写在 R 程序文件内 (保存为 R 多行字符串), 用 `sourceCpp()` 函数编译。

Rcpp 包为 C++ 提供了一个 `NumericVector` 数据类型, 用来存储数值型向量。用成员函数 `size()` 访问其大小, 用方括号下标访问其元素。

C 程序中定义的函数可以返回 `NumericVector` 数据类型, 将自动转换为 R 的数值型向量。

特殊的注释 `//[[Rcpp::export]]` 用来指定哪些 C++ 函数是要转换为 R 函数的。这叫做 Rcpp 属性 (attributes) 功能。

30.2.3 用 `sourceCpp()` 转换 C++ 源文件中的程序——正负交替迭代例子

直接把 C++ 代码写在 R 源程序内部的好处是不用管理多个源文件, 缺点是当 C++ 代码较长时, 不能利用专用 C++ 编辑环境和调试环境, 出错时显示的错误行号不好定位, 而且把代码保存在 R 字符串内, C++ 代码中用到字符时需要特殊语法。所以, 稍复杂的 C++ 代码应该放在单独的 C++ 源文件内。

假设上面的 `iters` 函数的 C++ 代码单独存入了一个 `iters.cpp` 源文件中。用如下的 `sourceCpp()` 函数把 C++ 源文件中代码编译并转换为 R 可访问的同名函数, 测试调用:

```

sourceCpp(file='iters.cpp')
print(iters(1:5))
## [1] 1 3 0 4 1

```

30.2.4 用 sourceCpp() 转换 C++ 源程序文件——卷积例子

考虑向量 $\mathbf{x} = (x_0, x_1, \dots, x_{n-1})$, $\mathbf{y} = (y_0, y_1, \dots, y_{m-1})$, 定义 \mathbf{x} 与 \mathbf{y} 的离散卷积 $\mathbf{z} = (z_0, z_1, \dots, z_{n+m-2})$ 为

$$z_k = \sum_{(i,j): i+j=k} x_i y_j = \sum_{i=\max(0, k-m+1)}^{\min(k, n)} x_i y_{k-i}.$$

假设如下的 C++ 程序保存到了源文件 conv.cpp 中。

```
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
NumericVector convolveCpp(
    NumericVector a, NumericVector b){
    int na = a.size(), nb = b.size();
    int nab = na + nb - 1;
    NumericVector xab(nab);

    for(int i=0; i < na; i++)
        for(int j=0; j < nb; j++)
            xab[i+j] += a[i] * b[j];

    return xab;
}
```

如下的代码用 sourceCpp() 函数把上面的 C++ 源文件 conv.cpp 自动编译并转换为同名的 R 函数, 进行测试:

```
sourceCpp(file='conv1.cpp')
print(convolveCpp(1:5, 1:3))
## [1] 1 4 10 16 22 22 15
```

Chapter 31

R 与 C++ 的类型转换

R 程序与由 Rcpp 支持的 C++ 程序之间需要传递数据，就需要将 R 的数据类型经过转换后传递给 C++ 函数，将 C++ 函数的结果经过转换后传递给 R。

31.1 用 wrap() 把 C++ 变量返回到 R 中

在 R API 中用 .Call() 函数调用 C 程序库函数时，R 对象的数据类型一般是 SEXP。Rcpp 提供了模板化的 wrap() 函数把 C++ 的函数返回值转换成 R 的 SEXP 数据类型。此函数的声明为

```
template <typename T> SEXP wrap(const T& object);
```

wrap() 能转换的类型包括：

- 把 int, double, bool 等基本类型转换为 R 的原子向量类型（所有元素数据类型相同的向量）；
- 把 std::string 转换为 R 的字符型向量；
- 把 STL 容器如 std::vector<T> 或 std::map<T> 转换成基本类型为 T 的向量，条件是 T 能够转换；
- 把 STL 的映射 std::map<std::string, T> 转换为基本类型为 T 的有名向量，条件是 T 能够转换；
- 可以转换定义了 operator SEXP() 的 C++ 类的对象；
- 可以转换专门化过 wrap 模板的 C++ 对象。

是否可用 wrap() 转换是在编译时确定的。

31.2 用 `as()` 函数把 R 变量转换为 C++ 类型

Rcpp 提供了模板化的 `as()` 用来把 SEXP 类型转换成适当的 C++ 类型。`as()` 函数的声明为:

```
template <typename T> T as(SEXP x);
```

`as()` 可以把 R 对象转换为基本的类型如 `int`, `double`, `bool`, `std::string` 等, 可以转换到元素为基础类型的 STL 向量如 `std::vector` 等。如果 C++ 类定义了以 SEXP 为输入的构造函数也可以利用 `as()` 来转换。`as()` 可以针对用户自定义类作专门化。

31.3 `as()` 和 `wrap()` 的隐含调用

当 C++ 中赋值运算的右侧表达式是一个 R 对象或 R 对象的部分内容时, 可以隐含地调用 `as()` 将其转换成左侧的 C++ 类型。

当 C++ 中赋值运算的左侧表达式是一个 R 对象或其部分内容时, 可以隐含地调用 `wrap()` 将右侧的 C++ 类型转换成 R 类型。

在用 Rcpp 属性 (Rcpp 属性见下一节) 声明的 C++ 函数中, 可以直接以 `IntegerVector`, `NumericVector`, `CharacterVector`, `Function` 等作为自变量类型或返回值, 可以与 R 中相应的类型直接对应。

能自动转换到 R 中的缺省值类型还包括:

- 用双撇号界定的字符串常量;
- 十进数值如 10, 4.5;
- 预定义的常数如 `true`, `false`, `R_NilValue`, `NA_STRING`, `NA_INTEGER`, `NA_REAL`, `NA_LOGICAL`;

Chapter 32

Rcpp 属性

32.1 Rcpp 属性介绍

Rcpp 属性 (attributes) 用来简化把 C++ 函数变成 R 函数的过程。做法是在 C++ 源程序中加入一些特殊注释，利用其指示自动生成 C++ 与 R 的接口程序。属性是 C++11 标准的内容，现在的编译器支持还不多，所以在 Rcpp 支持的 C++ 程序中写成了特殊格式的注释。

Rcpp 属性有如下优点：

- 降低了同时使用 R 与 C++ 的学习难度；
- 取消了很多繁复的接口代码；
- 可以在 R 会话中很简单地调用 C++ 代码，不需要用户自己考虑编译、连接、接口问题；
- 可以先交互地调用 C++，成熟后改编为 R 扩展包而不需要修改界面代码。

Rcpp 属性的主要组成部分如下：

- 在 C++ 中，提供 `Rcpp::export` 标注要输出到 R 中的 C++ 函数。
- 在 R 中，提供 `sourceCpp()`，用来自动编译连接保存在文件或 R 字符串中的 C++ 代码，并自动生成界面程序把 C++ 函数转换为 R 函数。
- 在 R 中，提供 `cppFunction()` 函数，用来把保存在 R 字符串中的 C++ 函数自动编译连接并转换成 R 函数。提供 `evalCpp()` 函数，用来把保存在 R 字符串中的 C++ 代码片段自动编译连接并执行。
- 在 C++ 中，提供 `Rcpp::depends` 标注，说明编译连接时需要的外部头文件和库的位置。
- 在构建 R 扩展包时，提供 `compileAttributes()` R 函数，自动给 C++ 函数生成相应的 `extern C` 声明和 `.Call` 接口代码。

32.2 在 C++ 源程序中指定要导出的 C++ 函数

用特殊注释`//[[Rcpp::export]]` 说明某 C++ 函数需要在编译成动态链接库时，把这个函数导出到链接库的对外可见部分。

例如

```
//[[Rcpp::export]]
NumericVector convolveCpp(
  NumericVector a, NumericVector b){
  .....
}
```

具体程序参见前面“用 `sourceCpp()` 转换 C++ 源程序文件—卷积例子”。假设此 C++ 源程序保存到了当前工作目录的 `conv.cpp` 源文件中，为了在 R 中调用此 C++ 程序，只要用如：

```
sourceCpp(file='conv.cpp')
convolveCpp(1:3, 1:5)
```

注意 `sourceCpp()` 把 C++ 源程序自动进行了编译链接并转换成了同名的 R 函数。在同一 R 会话内，如果源程序和其依赖资源没有变化（根据文件更新时间判断），就不重新编译 C++ 源代码。

在用特殊注释说明要导出的 C++ 函数时，可以用特殊的 `name=` 参数指定函数导出到 R 中的 R 函数名。如果不指定，R 函数名和 C++ 函数名是相同的。

例如

```
//[[Rcpp::export(name="conv")]]
NumericVector convolveCpp(
  NumericVector a, NumericVector b){
  .....
}
```

则 C++ 函数 `convolveCpp` 导入到 R 中后，改名为“conv”。

对于要导出的 C++ 函数，必须在全局名字空间中定义，而不能在某个 C++ 名字空间声明内定义。自变量必须能够用 `Rcpp::as` 转换成 C++ 类型，返回值必须是空值或者能够用 `Rcpp::wrap` 转换成 R 类型。在自变量和返回值类型说明中，必须使用完整的类型，比如 `std::string` 不能简写成 `string`。Rcpp 提供的类型如 `NumericVector` 可以不必用 `Rcpp::` 修饰。

32.3 在 R 中编译链接 C++ 代码

`sourceCpp()` 函数可以用 `code=` 指定一个 R 字符串，字符串的内容是 C++ 源程序，其中还是用特殊注释`//[[Rcpp::export]]` 标识要导出的 C++ 函数。如


```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
NumericVector convolveCpp(
    NumericVector a, NumericVector b){
    .....
}
')
convolveCpp(1:3, 1:5)
```

对于比较简单的单个 C++ 函数，可以用 `cppFunction()` 函数的 `code=` 指定一个 R 字符串，字符串的内容是一个 C++ 函数定义，转换为一个 R 函数。例如

```
cppFunction(code='
int fibonacci(const int x){
    if(x < 2) return x;
    else
        return ( fibonacci(x-1) + fibonacci(x-2) );
}
')
print(fibonacci(5))
```

为了在 R 中计算一个简单的 C++ 表达式，可以用 `evalCpp(' C++ 表达式内容')`，如

```
evalCpp('std::numeric_limits<double>::max()')
```

函数将返回该 C++ 表达式的值。

在 `cppFunction()` 和 `evalCpp()` 中，可以用 `depends=` 参数指定要链接的其它库，如

```
sourceCpp(depends='RcppArmadillo', code='.....')
```

在编译代码时与 `RcppArmadillo` 的动态连接库连接。

也可以把这样的链接依赖关系写在特殊的 C++ 注释中，如

```
//[[Rcpp::depends(RcppArmadillo)]]
```

这样的注释仅对 `sourceCpp()` 和 `cppFunction()` 有效，在编译 R 扩展包时，仍需要把依赖的包列在 DESCRIPTION 文件的 Imports 中，把要链接的包列在 LinkingTo 中。

32.4 Rcpp 属性的其它功能

32.4.1 自变量有缺省值的函数

借助于 Rcpp, 自变量有缺省值的 C++ 函数可以自动转换成自变量有缺省值的 R 函数。定义时要符合 C++ 语法, 比如带缺省值的自变量都要在不带缺省值的自变量的后面, 缺省值不能有变量。

例如

```
DataFrame readData(  
    CharacterVector file,  
    CharacterVector colNames = CharacterVector::create(),  
    std::string comment = "#",  
    bool header = true){ ... }
```

转换到 R 中, 相当于

```
function(file,  
          colNames=character(),  
          comment="#",  
          header=TRUE)
```

32.4.2 允许用户中断

在 C++ 代码中进行长时间的计算时, 应该允许用户可以中断计算。Rcpp 的办法是在 C++ 计算过程中每隔若干步循环就插入一个 `Rcpp::checkUserInterrupt();` 语句。

32.4.3 把 R 代码写在 C++ 源文件中

正常情况下, 应该把 R 代码和 C++ 代码写在分别的源程序中, 当 C++ 代码比较短时, 也可以把 C++ 代码写在 R 源程序中作为一个字符串。

Rcpp 允许把 C++ 代码和 R 代码都写在一个 C++ 源文件中, R 代码作为特殊的注释, 以 `/** R` 行开头, 以正常的 `*/` 结束。在 R 中用 `sourceCpp()` 调用这个 C++ 源文件, 就可以编译 C++ 后执行其中特殊注释内的 R 代码。这样的特殊注释可以有多个。

例如, 下述内容保存在文件 `fibonacci.cpp` 中:

```
/**[Rcpp::export]  
int fibonacci(const int x){  
    if(x < 2) return x;  
    else  
        return ( fibonacci(x-1) + fibonacci(x-2) );  
}
```

```
}

/** R
  # 调用 C++ 中的 fibonacci() 函数
  print(fibonacci(10))
*/
```

只要在 R 中运行

```
sourceCpp(file='fibonacci.cpp')
```

就可以编译连接此 C++ 文件，把其中用 `[[Rcpp::export]]` 标识的函数转换为 R 函数，并在 R 中执行源文件内特殊注释中的 R 代码。

32.4.4 在 C++ 中调用 R 的随机数发生器

在 C 或 C++ 中调用 R 的随机数发生器，需要能够同步地更新随机数发生器状态。如果利用 Rcpp 属性编译 C++ 源程序，则 Rcpp 属性会自动添加一个 RNGScope 实例进行随机数发生器状态的同步。

Chapter 33

Rcpp 提供的 C++ 数据类型

33.1 RObject 类

Rcpp 包为 C++ 定义了 `NumericVector`, `IntegerVector`, `CharacterVector`, `Matrix` 等新数据类型，可以直接与 R 的 `numeric`, `charactor`, `matrix` 对应。

Rcpp 最基础的 R 数据类型是 `RObject`, 这是 `NumericVector`, `IntegerVector` 等的基类, 通常不直接使用。`RObject` 包裹了原来 R 的 C API 的 `SEXP` 数据结构, 并且提供了自动的内存管理, 不再需要用户自己处理建立内存和消除内存的工作。`RObject` 存储数据完全利用 R C API 的 `SEXP` 数据结构, 不进行额外的复制。

因为 `RObject` 类是基类, 所以其成员函数也适用于 `NumericVector` 等类。`isNull`, `isObject`, `isS4` 可以查询是否 NULL, 是否对象, 是否 S4 对象。`inherits` 可以查询是否继承自某个特定类。用 `attributeNames`, `hasAttribute`, `attr` 可以访问对象的属性。用 `hasSlot`, `slot` 可以访问 S4 对象的插口 (slot)。

`RObject` 有如下导出类:

- `IntegerVector`: 整数向量;
- `NumericVector`: 数值向量;
- `LogicalVector`: 逻辑向量;
- `CharacterVector`: 字符型向量;
- `GenericVector`: 列表;
- `ExpressionVector`: 表达式向量;
- `RawVector`: 元素为 `raw` 类型的向量。
- `IntegerMatrix`, `NumericMatrix`: 整数值或数值矩阵。

在 R 向量中，如果其元素都是同一类型（如整数、双精度数、逻辑、字符型），则称为原子向量。Rcpp 提供了 IntegerVector, NumericVector, LogicalVector, CharacterVector 等数据类型与 R 的原子向量类型对应。在 C++ 中可以用 [] 运算符存取向量元素，也可以用 STL 的迭代器。用 .begin(), .end() 等界定范围，用循环或者 accumulate 等 STL 算法处理整个向量。

33.2 IntegerVector 类

在 R 中通常不严格区分整数与浮点实数，但是在与 C++ 交互时，C++ 对整数与实数严格区分，所以 RCpp 中整数向量与数值向量是区分的。

在 R 中，如果定义了一个仅有整数的向量，其类型是整数 (integer) 的，否则是数值型 (numeric) 的，如：

```
x <- 1:5
class(x)
## [1] "integer"
y <- c(0, 0.5, 1)
class(y)
## [1] "numeric"
```

用 as.integer() 和 as.numeric() 函数可以显式地确保其自变量转为需要的整数型或数值型。

RCpp 可以把 R 的整数向量传递到 C++ 的 IntegerVector 中，也可以把 C++ 的 IntegerVector 函数结果传递回 R 中变成一个整数向量。也可以在 C++ 中生成 IntegerVector 向量，填入整数值。

33.2.1 IntegerVector 示例 1：返回完全数

如果一个正整数等于它所有的除本身以外的因子的和，称这个数为完全数。如

$$6 = 1 + 2 + 3$$

$$28 = 1 + 2 + 4 + 7 + 14$$

是完全数。

任务：用 C++ 程序输入前 4 个完全数偶数，返回到 R 中。这 4 个数为 6, 28, 496, 8182。

程序：

```
library(Rcpp)
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
```

```
IntegerVector wholeNumberA(){
    IntegerVector epn(4);
    epn[0] = 6;    epn[1] = 28;
    epn[2] = 496; epn[3] = 8182;

    return epn;
}
')
print(wholeNumberA())
## [1]    6   28 496 8182
```

从例子看出，可以在 C++ 中建立一个 IntegerVector，需指定大小。可以逐个填入数值。直接返回 IntegerVector 到 R 即可，不需使用 wrap() 显式地转换。

33.2.2 IntegerVector 示例 2：输入整数向量

任务：用 C++ 编写函数，从 R 中输入整数向量，计算其元素乘积（与 R 的 prod() 函数功能类似）。程序如下：

```
require(Rcpp)
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
IntegerVector prod1(IntegerVector x){
    int prod = 1;
    for(int i=0; i < x.size(); i++){
        prod *= x[i];
    }
    return wrap(prod);
}
')
print(prod1(1:5))
```

从程序看出，用 IntegerVector 从 R 中接受一个整数值向量时，不需要显式地转换。把一个 C++ 整数值返回给 R 时，必须用 IntegerVector 返回，因为返回值是一个 C++ 的 int 类型，所以需要用 Rcpp::wrap() 转换一下。在 sourceCpp 中可以省略 Rcpp:: 部分。

还可以用 C++ STL 的算法库进行这样的累计乘积计算，std::accumulate() 可以对指定范围进行遍历累计运算。前两个参数是一个范围，用迭代器 (iterators) 表示开始和结束，第三个参数是初值，第四个参数是

对每个元素累计的计算。程序如下：

```
require(Rcpp)
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
IntegerVector prod2(IntegerVector x){
    int prod = std::accumulate(
        x.begin(), x.end(),
        1, std::multiplies<int>());
    return wrap(prod);
}
')
print(prod2(1:5))
```

在以上的输入 IntegerVector 的 C++ 程序中，如果从 R 中输入了实数型的向量，则元素被转换成整数型。比如 `prod2(seq(1,1.9,by=0.1))` 结果将等于 1。如果输入了无法转换为整数型向量的内容，比如 `prod2(c('a', 'b', 'c'))`，程序会报错。

33.3 NumericVector 类

NumericVector 类在 C++ 中保存双精度型一维数组，可以与 R 的实数型向量 (class 为 numeric) 相互转换。这是自己用 C++ 程序与 R 交互时最常用到的数据类型。

33.3.1 示例 1：计算元素 p 次方的和

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
NumericVector ssp(
    NumericVector vec, double p){
    double sum = 0.0;
    for(int i=0; i < vec.size(); i++){
        sum += pow(vec[i], p);
    }
}
```



```

    return(wrap(sum));
}
')
ssp(1:4, 2)
## [1] 30
ssp((1:4)/10, 2.2)
## [1] 0.2392496
sum( ((1:4)/10)^2.2 )
## [1] 0.2392496

```

从程序看出，用 Rcpp 属性编译时，C++ 函数的输入与返回值类型转换有如下规则：R 中数值型向量在 C++ 中可以用 NumericVector 接收；R 中单个的实数在 C++ 中可以用 double 来接收；为了返回单个的实数，在 C++ 中需要以 NumericVector 为返回类型，可以从一个 double 型用 wrap() 转换。C++ 中如果返回 NumericVector，在 R 中转换为数值型向量。

33.3.2 示例 2: clone 函数

在自定义 R 函数时，输入的自变量的值不会被改变，相当于自变量都是局部变量。如果在自定义函数中修改了自变量的值，实际上只能修改自变量的一个副本的值。如

```

x <- 100
f <- function(x){
  print(x); x <- 99; print(x)
}
c(f(x), x)
## [1] 100
## [1] 99
## [1] 99 100

```

但是，在用 Rcpp 编写 R 函数时，因为 RObject 传递的是指针，并不自动复制自变量值，所以修改自变量值会真的修改原始的自变量变量值。如：

```

sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
NumericVector f2(NumericVector x){
  x[0] = 99.0;
  return(x);
}

```

```
'')
x <- 100
c(f2(x), x)
[1] 99 99
```

可见自变量的值被修改了。当然，对这个问题而言，因为输入的是一个标量，只要函数自变量不是 `NumericVector` 类型而是用 `double` 类型，则自变量值会被复制，达到值传递的效果，自变量值也就不会被真的修改。如

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
NumericVector f3(double x){
  x = 99.0;
  return(wrap(x));
}
')
x <- 100
c(f3(x), x)
## [1] 99 100
```

下面的程序把输入向量每个元素平方后返回，为了不修改输入自变量的值而是返回一个修改后的副本，使用了 `Rcpp` 的 `clone` 函数：

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
NumericVector square(NumericVector x){
  NumericVector y = clone(x);
  for(int i=0; i < x.size(); i++){
    y[i] = x[i]*x[i];
  }
  return(y);
}
')
x <- c(2, 7)
cbind(square(x), x)
```

```
##          x
## [1,]    4 2
## [2,]   49 7
```

33.3.3 示例 3：把输入矩阵制作副本计算元素平方根

NumericMatrix 是 Rcpp 提供的元素为双精度型的矩阵。下面的例子输入一个 R 矩阵，输出其元素的平方根，用了 clone 函数来避免对输入的直接修改。

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
NumericMatrix matSqrt(NumericMatrix x){
    NumericMatrix y = clone(x);
    std::transform(y.begin(), y.end(),
        y.begin(), ::sqrt);
    return(y);
}
')
x <- rbind(c(1,2), c(3,4))
cbind(matSqrt(x), x)
##          [,1]      [,2] [,3] [,4]
## [1,] 1.000000 1.414214    1    2
## [2,] 1.732051 2.000000    3    4
```

在上面的 C++ 程序中，NumericMatrix 看成了一维数组，用 STL 的 iterater 遍历，用 STL 的 transform 对每个元素计算变换。

33.4 Rcpp 的其它向量类

33.4.1 Rcpp 的 LogicalVector 类

LogicalVector 类可以存储 C++ 值 true, false, 还可以保存缺失值 NA_REAL, R_NaN, R_PosInf, 但是这些不同的缺失值转换到 R 中都变成 NA。

如:

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
LogicalVector f4(){
  LogicalVector x(5);
  x[0] = false; x[1] = true;
  x[2] = NA_REAL;
  x[3] = R_NaN; x[4] = R_PosInf;
  return(x);
}
')
f()
## [1] FALSE TRUE NA NA NA
identical(f(), c(FALSE, TRUE, rep(NA,3)))
## [1] TRUE
```

33.4.2 Rcpp 的 CharacterVector 类型

CharacterVector 类型可以与 R 的字符型向量相互交换信息，在 C++ 中其元素为字符串。字符型缺失值在 C++ 中为 R_NaString。R 的字符型向量也可以转换为 std::vector。

如：

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
CharacterVector f5(){
  CharacterVector x(3);
  x[0] = "This is a string";
  x[1] = "Test";
  x[2] = R_NaString;
  return(x);
}
')
f5()
## [1] "This is a string" "Test" NA
```

33.5 Rcpp 提供的其它数据类型

33.5.1 Named 类型

R 中的向量、矩阵、数据框可以有元素名、列名、行名。这些名字可以借助 Named 类添加。

例如：

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
NumericVector f6(){
  NumericVector x = NumericVector::create(
    Named("math") = 82,
    Named("chinese") = 95,
    Named("English") = 60);
  return(x);
}
')
f6()
##    math chinese English
##     82      95      60
```

“Named(元素名)”可以简写成“_元素名”。如：

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
NumericVector f6b(){
  NumericVector x = NumericVector::create(
    _["math"] = 82,
    _["chinese"] = 95,
    _["English"] = 60);
  return(x);
}
')
```

33.5.2 List 类型

Rcpp 提供的 List 类型对应于 R 的 list(列表) 类型，在 C++ 中也可以写成 GenericVector 类型。其元素可以不是同一类型，在 C++ 中可以用方括号和字符串下标的格式访问其元素。

例如，下面的函数输入一个列表，列表元素 vec 是数值型向量，列表元素 multiplier 是数值型标量，返回一个列表，列表元素 sum 为 vec 元素和，列表元素 dsum 为 vec 元素和乘以 multiplier 的结果：

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
List f7(List x){
    NumericVector vec = as<NumericVector>(x["vec"]);
    double multiplier = as<double>(x["multiplier"]);
    double y = 0.0, y2;
    for(int i=0; i<vec.length(); i++){
        y += vec[i];
    }
    y2 = y*multiplier;
    return(List::create(Named("sum")=y,
        Named("dsum")=y2));
}
')
f7(list(vec=1:5, multiplier=10))
## $sum
## [1] 15
##
## $dsum
## [1] 150
```

上面的程序用了 `Rcpp::List::create()` 当场生成 List 类型，因为用 Rcpp 属性功能编译所以可以略写 `Rcpp::`。也可以在程序中预先生成指定大小的列表，然后再给每个元素赋值，元素值可以是任意能够转化为 SEXP 的类型，如：

```
.....
List gv(2);
gv[0] = "abc";
gv[1] = 123;
```

可以用 List 的 `reserve` 函数为列表指定元素个数。

33.5.3 Rcpp 的 DataFrame 类

Rcpp 的 DataFrame 类用来与 R 的 data.frame 交换信息。

示例如：

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
DataFrame f8(){
  IntegerVector vec =
    IntegerVector::create(7,8,9);
  std::vector<std::string> s(3);
  s[0] = "abc"; s[1] = "ABC"; s[2] = "123";
  return(DataFrame::create(
    Named("x") = vec,
    Named("s") = s));
}
')
f8()
##   x   s
## 1 7 abc
## 2 8 ABC
## 3 9 123
```

33.5.4 Rcpp 的 Function 类

Rcpp 的 Function 类用来接收一个 R 函数，并且可以在 C++ 中调用这样的函数。

示例如：

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
SEXP dsort(Function sortFun, SEXP x){
  return sortFun(x, Named("decreasing", true));
}
')
```

```
dsort(sort, c('bb', 'ab', 'ca'))  
## [1] "ca" "bb" "ab"  
## dsort(sort, c(2,1,3))  
## [1] 3 2 1
```

程序用 Function 对象 sortFun 接收从 R 中传递过来的排序函数，实际调用时传递过来的是 R 的 sort 函数。

在 C++ 中调用 R 函数时，有名的自变量用 “Named(自变量字符串, 自变量值)” 的格式给出。

程序中的待排序的向量与排序后的向量都用了 SEXP 来说明，即直接传送原始 R API 指针，这样可以不管原来类型是什么，在 C++ 中完全不进行类型转换或复制。

从运行例子看出数值和字符串都正确地按照降序排序后输出了。

R 函数可以不作为 C++ 的函数自变量传递进来，而是直接调用 R 的函数。

下面的函数直接调用 rt 函数生成 2 个自由度为 3 的 t 分布随机数：

```
sourceCpp(code='  
#include <Rcpp.h>  
using namespace Rcpp;  
  
//[[Rcpp::export]]  
NumericVector rt23(){  
  Function rt("rt");  
  return rt(2, 3);  
}  
' )  
set.seed(1)  
rt23()  
## [1] -0.7027211 -0.5693196  
rt23()  
## [1] 0.6842766 -0.3620012
```

使用了 Rcpp 属性时，生成的界面程序会自动生成一个 RNGScope 的实例，用来保存当前的随机数发生器状态，在解构时将自动更新随机数发生器状态。

用 rt 这个 R 函数名初始化了一个 C++ 的 Function 对象 rt，然后就可以调用这个 C++ 函数了。从程序可以看出，连续两次调用的结果不同。

33.5.5 Rcpp 的 Environment 类

R 的环境是分层的，可以逐层查找变量名对应的内容。Rcpp 的 Environment 类用来与 R 环境对应。可以利用 Environment 来定位 R 的扩展包中的函数或数据，例如下面的程序片段在 C++ 中定位了 stats 扩展包中的 rnorm 函数并进行了调用：

```
Environment stats("package:stats");
Function rnorm = stats["rnorm"];
return rnorm(3, Named("sd", 100.0));
```

当然，也可以用 Function 对象直接从各环境中搜索 rnorm 函数名，但是这样指定环境更可靠。

下面的例子访问了 R 全局环境，取出了全局变量 x 的值存入 C++ 的双精度型 STL 向量中，并把一个 C++ 的 STL map 型数据转换成有名字符型向量存到了全局变量 y 中。

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
void f9(){
    Environment global = Environment::global_env();
    std::vector<double> vx = global["x"];
    std::map<std::string, std::string> ma;
    ma["foo"] = "abc";
    ma["bar"] = "123";
    global["y"] = ma;
    return;
}
')
x <- c(1,5)
f9()
y
##    bar    foo
## "123" "abc"
```


Chapter 34

Rcpp 糖

在 C++ 中，向量和矩阵的运算通常需要逐个元素进行，或者调用相应的函数。Rcpp 通过 C++ 的表达式模板 (expression template) 功能，可以在 C++ 中写出像 R 中对向量和矩阵运算那样的表达式。这称为 Rcpp 糖 (sugar)。

R 中的很多函数如 `sin` 等是向量化的，Rcpp 糖也提供了这样的功能。Rcpp 糖提供了一些向量化的函数如 `ifelse`, `sapply` 等。

比如，两个向量相加可以直接写成 `x + y` 而不是用循环或迭代器 (iterator) 逐元素计算；若 `x` 是一个 `NumericVector`，用 `sin(x)` 可以返回由 `x` 每个元素的正弦值组成的 `NumericVector`。

Rcpp 糖不仅简化了程序，还提高了运行效率。

34.1 简单示例

比如，函数

$$f(x, y) = \begin{cases} x^2 & x < y, \\ -y^2 & x \geq y \end{cases}$$

如下的程序可以在 C++ 中定义一个向量化的版本：

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
NumericVector f11(NumericVector x, NumericVector y){
    return ifelse(x < y, x*x, -(y*y));
}
```

```
'')
f11(c(1, 3), c(4,2))
## [1] 1 -4
```

上面简单例子中, $x < y$ 是向量化的, $x * x, y * y, -(y * y)$ 都是向量化的。`ifelse` 也是向量化的。

34.2 向量化的运算符

34.2.1 向量化的四则运算

Rcpp 糖向量化了 $+$, $-$, $*$, $/$ 。设 x, y 都是 `NumericVector`, 则 $x + y, x - y, x * y, x / y$ 将返回对应元素进行四则运算后的 `NumericVector` 变量。

向量与标量运算, 如 $x + 2.0, 2.0 - x, x * 2.0, 2.0 / x$ 将返回标量与向量每个元素进行四则运算后的 `NumericVector` 变量。

还可以进行混合四则运算, 如:

```
NumericVector res = x * y + y / 2.0;
NumericVector res = x * (y - 2.0);
NumericVector res = x / (y * y);
```

参加四则运算的或者都是同基本类型的向量而且长度相等, 或者一边是向量, 一边是同类型的标量。

注意: 对向量整体赋一个相同的值, 不能简单地写成如 $x=0$; 这样的赋值, 需要用循环或 STL 的 `fill` 算法, 如 `std::fill(x.begin(), x.end(), 0);`。

34.2.2 向量化的二元逻辑运算

Rcpp 糖扩展了两个元素的比较到两个等长向量的比较, 以及一个向量与一个同类型标量的比较, 结果是一个同长度的 `LogicalVector`。比较运算符包括 $<$, $>$, $<=$, $>=$, $==$, $!=$ 。

也可以使用嵌套的表达式, 比如, 设 x, y 是两个 `NumericVector`, 可以写:

```
LogicalVector res = (x + y) < (x * x);
```

34.2.3 向量化的一元运算符

对数值型向量或返回数值型向量的表达式前面加负号, 可以返回每个元素取相反数的结果。比如, 设 x 是 `NumericVector`, 可以写:

```
NumericVector res = -x;
NumericVector res = -x * (x + 2.0);
```

对逻辑型向量或返回逻辑型向量的表达式前面加叹号，可以返回每个元素取反的结果，如：

```
NumericVector y, z;
NumericVector res = ! ( y < z );
```

34.3 用 Rcpp 访问数学函数

在 C 和 C++ 代码中可以调用 R 中的函数，但是格式比较复杂，而且原来不支持向量化。Rcpp 糖则使得从 C++ 中调用 R 函数变得和在 R 调用函数格式类似。

R 源程序中提供了许多数学和统计相关的函数，这些函数可以编译成一个独立的函数库，供其它程序链接使用，函数内容在 R 的 Rmath.h 头文件中有详细列表。

Rcpp 糖把 R 中的数学函数在 C++ 中向量化了，输入一个向量，结果是对应元素为函数值的向量。自变量类型可以是数值型或整数型。这些数学函数包括 abs, exp, floor, ceil, pow 等。在 Rcpp 中支持的 C++ 源程序中调用这些函数，最简单的一种方法是直接在 Rcpp 名字空间中使用和 R 中相同的函数名和调用方法，类似 sqrt 这样的函数允许输入一个向量，对向量的每个元素计算。

比如，下面的程序输入一个向量，计算相应的标准正态分布函数值：

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
NumericVector f10(NumericVector x){
  NumericVector y(x.size());
  y = pnorm(x);
  return y;
}
')
f10(c(0, 1.96, 2.58))
## [1] 0.5000000 0.9750021 0.9950600
```

如果需要对单个的数值计算，可以使用 Rmath.h 中定义的带有 Rf_ 的版本，如：

```
y = ::Rf_pnorm5(x, 0.0, 1.0, 1, 0);
```

这里用:: 使用了缺省的名字空间。注意所有自变量都不能省略。

Rcpp 还提供了一个 R 名字空间，可以用不带 Rf_ 前缀的函数，但是自变量也不能省略。如

```
y = R::pnorm(x, 0.0, 1.0, 1, 0);
```

在 Rcpp 的 R 名字空间中有许多的数学和统计相关的函数，各函数的自变量参见 Rcpp 的 Rmath.h 文件。

R 中提供了许多分布的分布密度（或概率质量函数）、分布函数、分位数函数，分布密度函数和概率质量函数命名类似 `dxxxx`，分布函数命名类似 `pxxxx`，分位数函数命名类似 `qxxxx`。在 Rcpp 糖支持下，这些函数可以直接用类似 R 中的格式调用。如

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
List f14(){
  NumericVector x =
    NumericVector::create(0, 1.96, 2.58);
  NumericVector p =
    NumericVector::create(0.95, 0.975, 0.995);
  NumericVector y1 = dnorm(x, 0.0, 1.0);
  NumericVector y2 = pnorm(x, 0.0, 1.0);
  NumericVector y3 = qnorm(p, 0.0, 1.0);
  return List::create(Named("y1")=y1,
    Named("y2")=y2, Named("y3")=y3);
}
')
f14()
## $y1
## [1] 0.39894228 0.05844094 0.01430511
##
## $y2
## [1] 0.5000000 0.9750021 0.9950600
##
## $y3
## [1] 1.644854 1.959964 2.575829
##
```

R 中的 `rxxxx` 类的函数可以产生各种分布的随机数向量，随机数向量与当前种子有关。Rcpp 属性会自动地维护随机数发生器的状态使其与 R 同步。如

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
NumericVector f15(){
```

```

    NumericVector x = rnorm(10, 0.0, 1.0);
    return x;
}
')
round(f15(), 2)
## [1]  0.62 -0.06 -0.16 -1.47 -0.48
## [6]  0.42  1.36 -0.10  0.39 -0.05

```

34.4 返回单一逻辑值的函数

在 R 中, `any()` 和 `all()` 对一个逻辑向量分别判断是否有任何真值, 以及所有元素为真值。Rcpp 糖在 C++ 中也提供了这样的 `any()` 和 `all()` 函数。如

```

sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
List f12(){
    IntegerVector x = seq_len(1000);
    LogicalVector res1 = any( x*x < 3 );
    LogicalVector res2 = all( x*x < 3 );
    return List::create(Named("any")=res1,
        Named("all")=res2);
}
')
f12()
## $any
## [1] TRUE
##
## $all
## [1] FALSE

```

`any()` 和 `all()` 不是直接返回逻辑值, 而是返回一个类对象, 该类定义了 `is_true`, `is_false`, `is_na` 方法与向 SEXP 转换的运算符。此种结果可以保存到 `LogicalVector` 中, 但不能赋值到 `bool` 类型, 因为可能有缺失值。

逻辑型糖表达式结果可以用 `is_true`, `is_false`, `is_na` 来判断结果。如

```
bool res1 = is_true( any( x < y ) );
bool res2 = is_na( all( x < y ) );
```

在求 `any()` 结果时，一旦遇到一个真值结果就为真值，即使后面有缺失值也没有关系。在求 `all()` 结果时，一旦遇到一个假值结果就为假值，即使后面有缺失值也没有关系。

34.5 返回糖表达式的函数

`is_na` 以任何糖表达式为输入，输出一个逻辑类型的元素个数相同的糖表达式。结果每个元素当输入中对应元素缺失时为 `TRUE`，否则为 `FALSE`。如

```
IntegerVector x = IntegerVector::create(0, 1, NA_INTEGER, 3);
LogicalVector res1 = is_na(x);
LogicalVector res2 = all( is_na(x) );
if( is_true( any( ! is_na(x) ) ) ) ...
```

`seq_along` 输入一个向量，输出一个元素为该向量的各个下标值的糖表达式。如

```
IntegerVector x = IntegerVector::create(0, 1, NA_INTEGER, 3);
seq_along(x);
seq_along(x*x*x*x*x);
```

注意上述程序不会计算 `x*x*x*x*x` 的值而只利用其结果的元素个数。所以两次调用的计算量是一样的。

`seq_len` 自变量为个数，返回一个元素值为正数的元素个数等于自变量值的糖表达式，第 i 个元素等于 i 。常可与 `sapply`, `lapply` 配合使用。如

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
List f13(){
  IntegerVector x =seq_len(3);
  List y = lapply( seq_len(3), seq_len);
  return List::create(Named("x")=x,
    Named("y")=y);
}
')
f13()
## $x
## [1] 1 2 3
```



```
##
## $y
## $y[[1]]
## [1] 1
##
## $y[[2]]
## [1] 1 2
##
## $y[[3]]
## [1] 1 2 3
##
```

`pmin` 和 `pmax` 自变量为两个等长的向量或糖表达式，返回长度相同的结果，结果元素只等于对应元素的最小值或最大值。

自变量也可以取一个向量一个标量，向量的每个元素与标量比较得到最小值或最大值。如

```
IntegerVector x =seq_len(10);
pmin(x, x*x);
pmin(x*x, 2);
```

`ifelse` 函数有三个自变量，第一自变量是一个逻辑型向量值的糖表达式，第二和第三自变量或者是两个同类型并与第一自变量等长的糖表达式，或者其中一个是同类型标量，结果仍为向量型糖表达式，第 i 元素当第一自变量第 i 元素为真时取第一自变量的第 i 元素，当第一自变量第 i 元素为假时取第二自变量的第 i 元素，当第一自变量第 i 元素为缺失值时取缺失值。如

```
IntegerVector x, y;
IntegerVector res1 = ifelse( x < y, x, (x+y)*y );
IntegerVector res2 = ifelse( x > y, x, 2 );
```

`sapply` 函数第一自变量是一个向量或列表，第二自变量是一个函数。返回值类型在编译时从函数的结果类型导出。第二自变量可以是任意的 C++ 函数，比如，可以是如下的重载的模板化函数：

```
template <typename T>
T square( const T& x ){
    return x * x;
}
sapply( seq_len(4), square<int> );
```

下面的例子中 `sapply` 第二自变量使用了 functor，这是一种能够产生函数的函数：

```
template <typename T>
struct square : std::unary_function<T,T> {
    T operator() (const T& x) {
```

```

    return x * x;
  }
}
sapply( seq_len(4), square<int>() );

```

`lapply` 函数与 `sapply` 函数基本相同，只不过 `lapply` 函数总是返回列表，列表在 Rcpp 中为 `List` 或 `GenericVector`，在 R API 中类型为 `VECSXP`。

`sign` 函数输入一个数值型或整型表达式，返回各元素值在 $\{1, 0, -1, NA\}$ 中取值的糖表达式，可以保存到 `IntegerVector` 中。结果各元素的取值表示输入中对应元素为正、零、负和缺失。如

```

IntegerVector x;
IntegerVector res1 = sign( x );
IntegerVector res2 = sign( x*x );

```

`diff` 函数自变量为数值型或整数型向量或有这样结果的糖表达式，输出后一元素减去前一元素的结果，结果长度比输入长度少一。如

```

IntegerVector res = diff( seq_len(5) );

```

34.6 R 与 Rcpp 不同语法示例

Rcpp 糖通过一些现代的 C++ 技术，支持部分的向量化运算，比如，`NumericVector` 与标量相加，两个等长 `NumericVector` 相加，对 `NumericVector` 计算如绝对值这样的数学函数值等。但是，C++ 毕竟和 R 有很大差别，要区分 Rcpp 能做的和不能做的。

例如，`NumericVector` 为了把向量所有元素都改成同一值，不能直接用等于赋值。可以用 `std::fill(y.begin(), y.end(), 99.99)` 这样的做法。

Chapter 35

用 Rcpp 帮助制作 R 扩展包

R 扩展包是把解决某种问题的可复用代码、文档整合在一起的最好的方法。写成 R 扩展包后，可以自己用，也可以利用 CRAN 分发。扩展包用户一般不用自己编译。

使用扩展包来组织程序，多个源程序、头文件之间的依赖关系可以自动得到处理。

扩展包提供了测试、文档和一致性检查的统一框架。

扩展包中代码可以仅有 R 程序，也可以包括 C 程序、C++ 程序、Fortran 程序。如果仅有 R 代码，就不需要借助于 Rcpp，可以使用 `package.skeleton()` 函数生成一个扩展包框架。如果有 C++ 代码，就可以用 Rcpp 作为接口，并用 Rcpp 提供的 `Rcpp.package.skeleton()` 函数制作扩展包框架。

Rcpp 属性的 Exports 注释仍可在制作扩展包时指定如何输出 C++ 中定义的函数使其在 R 中可调用。

35.1 不用扩展包共享 C++ 代码的方法

Rcpp 属性的 `sourceCpp()` 通常只适用于写在 R 程序内部的简短 C++ 代码，或者写在一个单独 C++ 文件中，不依赖于其它 C++ 程序的单独代码。如果有多个 C++ 源程序、头文件，彼此有依赖关系，最好使用扩展包。

在多个单独的 C++ 文件共享某些简单的代码，彼此不互相依赖时，可以用 C++ 的预处理 `include` 命令共享这些代码。

比如，有多个 C++ 源程序都用到如下的代码：

```
#ifndef __UTILITIES__
#define __UTILITIES__
inline double timesTwo(double x) {
    return x * 2;
}
#endif // __UTILITIES__
```

假设这段代码保存到当前子目录的“utilities.hpp”文件中。则在每个需要用到这段代码的 C++ 源程序中，插入如：

```
#include "utilities.hpp"
//[[Rcpp::export]]
double transformValue(double x){
    return timesTwo(x) * 10;
}
```

35.2 生成扩展包

35.2.1 利用已有基于 Rcpp 属性的源程序制作扩展包

假设在当前目录中有了若干个 C++ 文件，其中需要转换到 R 中的 C++ 函数已经用 `Rcpp::export` 声明过。其中一个是 `conv1.cpp`。

从当前目录启动 R，运行

```
Rcpp.package.skeleton("testpack",
  example_code=FALSE,
  attributes=TRUE,
  cpp_files=c("conv1.cpp"))
```

运行显示：

```
Creating directories ...
Creating DESCRIPTION ...
Creating NAMESPACE ...
Creating Read-and-delete-me ...
Saving functions and data ...
Making help files ...
Done.
Further steps are described in './testpack/Read-and-delete-me'.
```

Adding Rcpp settings

```
>> added Imports: Rcpp
>> added LinkingTo: Rcpp
>> added useDynLib directive to NAMESPACE
>> added importFrom(Rcpp, evalCpp) directive to NAMESPACE
>> copied conv1.cpp to src directory
```

运行完后，在当前目录生成了一个 `testpack` 子目录，这是要制作的扩展包的名字。在 `testpack` 子目录中，

有文件 DESCRIPTION, NAMESPACE, Read-and-delete-me, 有子目录 src, R, man。

子目录 src 中为 C++ 和 C 源程序、头文件。子目录 R 中为 Rcpp 从 C++ 程序转换过来的 R 接口程序，用户自己的 R 程序也可以放在这里。子目录 man 是特殊格式的文档，其格式类似 LaTeX。

35.2.2 DESCRIPTION 文件

在 DESCRIPTION 文件中，有扩展包名称、版本、日期、作者姓名、维护者姓名和联系方式、简单描述、授权，还有 Imports 和 LinkingTo 两项。除此之外，还有许多可选的域，如 Depends。

Imports 给出本软件包要调用的其它扩展包，但是这些扩展包并不随本扩展包一起调入，仅是会调入其名字空间。这里的值为

```
Imports: Rcpp (>= 0.12.3)
```

LinkingTo 指定在编译本软件包的 C、C++、Fortran 等源程序时，会用到哪些其它扩展包的头文件。这里的值为

```
LinkingTo: Rcpp
```

指定的这些扩展包一般是编译时才有用的，所以一般不会出现在 Depends 和 Imports 域中。

LinkingTo 只解决了头文件的问题，要链接除了 Rcpp 之外的二进制库文件，还需要手工编辑 src/Makevars 和 src/Makevars.win 文件。

和 Imports 有些相像的 DESCRIPTION 域是 Depends，指定调入本扩展包时必须预先调入的软件包。这里“调入”是指用 `library()` 或 `require()` 调入扩展包。多个扩展包名用逗号分开，可以在扩展包名字后面加圆括号，在圆括号内写上 `>=` 某个版本号，如“`MASS(>=3.1-20)`”。

Depends 也可以指定依赖于某个 R 版本之后，如“`R(>=2.14.0)`”。

DESCRIPTION 文件中的 Suggests 与和 Depends 域类似，但不是本扩展包必须的，比如仅用在某个例子中或测试中、仅用来编译 vignettes。

35.2.3 NAMESPACE 文件

示例 NAMESPACE 文件如下：

```
useDynLib(testpack)
exportPattern("^[[:alpha:]]+")
importFrom(Rcpp, evalCpp)
```

其中第一行指定调用本软件包时，需要调入的本扩展包的动态链接库。第二行指定扩展包需要对外部可见的 R 函数是所有函数名字以字母开头的 R 函数。用户可以自己指定其它的模式或者指定固定的若干个函数。第三行说明了需要从 Rcpp 包导入 evalCpp 函数。

35.3 重新编译

修改了扩展包中的 C++ 源程序后，需要重新编译。只要在 R 中把工作目录设为软件包的子目录内，运行

```
compileAttributes()
```

这会自动生成两个文件，一个是 `src/RcppExports.cpp`，是 C++ 程序的接口函数。另一个是 `R/RcppExports.R`，用 `.Call` 来调用 C++ 接口函数，转换成 R 函数。这两个文件不要自己修改。

35.4 建立 C++ 用的接口界面

利用了 `Rcpp` 属性可以指定要输出到 R 中的函数。

在 C++ 源程序中加入特殊注释

```
//[[Rcpp::interfaces(r, cpp)]]
```

则软件包在编译时也会生成该源程序文件中函数的外部可访问的接口，这些接口的界面会在安装后的包的 `include` 子目录中出现，在开发时出现在 `inst/include` 子目录中。

设要生成的扩展包名为 `testpack`，则界面文件包括 `include` 子目录中的 `testpack_RcppExports.h` 文件和 `testpack.h` 文件，`testpack.h` 文件仅用来包含入 `testpack_RcppExports.h` 文件。

如果需要添加自己的一些界面程序，可以修改 `testpack.h` 文件，这时需要去掉文件开始的自动生成标记，并且保留对 `testpack_RcppExports.h` 文件的包含。

导出的 C++ 界面都在与制作的扩展包同名的名字空间中，比如，如果制作的软件包名为 `testpack`，其中导出的一个 C++ 函数为 `convolveCpp`，则在别的包的 C++ 源程序中调用时，应该包含 `testpack.h` 文件，并用 `testpack::convolveCpp()` 格式调用。

如果自己本扩展包需要在编译时包含这些头文件，需要自己编辑 `src` 子目录中的 `Makevars` 文件和 `Makevars.win` 文件，添加行：

```
PKG_CPPFLAGS += -I../inst/include/
```

Part IX

专题

Chapter 36

R 语言的文本处理

36.1 简单的文本处理

在信息爆炸性增长的今天，大量的信息是文本型的，如互联网上的大多数资源。R 具有基本的文本数据处理能力，而且因为 R 的向量语言特点和强大的统计计算和图形功能，用 R 处理文本数据是可行的。

字符串常量写在两个双撇号或者两个单撇号中间，如果内容中有单撇号或者双撇号，可以在前面加反斜杠\。

R 中处理文本型数据的函数有文件访问函数，`readLines`，`nchar`，`paste`，`sprintf`，`format`，`formatC`，`substring` 等。

R 支持正则表达式，函数 `grep`，`grepl`，`sub`，`gsub`，`regexpr`，`gregexpr`，`strsplit` 与正则表达式有关。

注意 R 的字符型向量每个元素是一个字符串，所以字符型函数一般也是对输入的一个字符型向量的每个元素操作的。

R 扩展包 `stringr` 和 `stringi` 提供了更方便、功能更强的字符串功能。

36.1.1 `nchar()` 函数

函数 `nchar(text)` 计算字符串长度，默认按照字符个数计算而不是按字节数计算，如

```
nchar(c("a", "bc", "def", " 北京"))
```

```
## [1] 1 2 3 2
```

注意函数对输入的字符型向量每个元素计算长度。

加选项 `type="bytes"` 可用按字符串占用的字节数计算，这时一个汉字占用多个字节（具体占用多少与编码有关）。如

```
nchar(c("a", "bc", "def", " 北京"), type="bytes")
```

```
## [1] 1 2 3 4
```

36.1.2 toupper()、tolower() 函数

`toupper()` 将字符型向量的每个元素中的小写字母转换为大写, `tolower()` 转小写。

36.1.3 trimws() 函数

`trimws(x)` 删去字符型向量 `x` 的每个元素的开头和结尾的所有空格。加选项 `which='left'` 可以仅删去开头的空格, 选项 `which='right'` 可以仅删去结尾的空格。如

```
trimws(c(" 李明", " 李明 ", " 李明 ", " 李 明"))
```

```
## [1] "李明" "李明" "李明" "李 明"
```

```
trimws(c(" 李明", " 李明 ", " 李明 ", " 李 明"), which="left")
```

```
## [1] "李明" "李明" "李明" "李 明"
```

```
trimws(c(" 李明", " 李明 ", " 李明 ", " 李 明"), which="right")
```

```
## [1] " 李明" "李明" " 李明" "李 明"
```

36.1.4 chartr() 函数

`chartr(old, new, x)` 函数指定一个字符对应关系, 旧字符在 `old` 中, 新字符在 `new` 中, `x` 是一个要进行替换的字符型向量。比如, 下面的例子把所有! 替换成., 把所有; 替换成,:

```
chartr("!", ".", c('Hi; boy!', 'How do you do!'))
```

```
## [1] "Hi, boy." "How do you do."
```

```
chartr("。; 县", ".,; 区", " 昌平区, 大兴县; 固安县。")
```

```
## [1] "昌平区, 大兴区; 固安区."
```

第二个例子中被替换的标点是中文表达, 替换成了相应的英文标点。

36.1.5 paste() 函数

`paste()` 是最常用的字符串函数, 用来把多个部分组合为一个字符型向量。各部分之间缺省用空格连接, 可以用 `sep=` 指定连接用的字符串。各部分长度不同时短的自动循环使用。非字符串类型自动转换为字符型。如

```
paste(c("x", "y"), c("a", "b"))
```

```
## [1] "x a" "y b"
```

```
paste("data", 1:3, ".txt", sep="")
```

```
## [1] "data1.txt" "data2.txt" "data3.txt"
```

paste() 函数使用 collapse 参数可以把一个输入字符型向量各元素用指定的字符串连接成一个长字符串。如

```
paste(c('a', 'bc', 'def'), collapse='.')
```

```
## [1] "a.bc.def"
```

在使用了 collapse 时如果 paste 有多个要连接的部分，函数先将各部分连接成为一个字符型向量，然后再把结果的各个向量元素连接起来。如

```
paste("data", 1:3, ".txt", sep="", collapse=";")
```

```
## [1] "data1.txt;data2.txt;data3.txt"
```

36.1.6 sprintf() 函数

sprintf 是 C 语言中 sprintf 的向量化版本，可以把一个元素或一个向量的各个元素按照 C 语言输出格式转换为字符型向量。第一个自变量是 C 语言格式的输出格式字符串，比如，%d 表示输出整数，%f 表示输出实数，%02d 表示输出宽度为 2、不够左填 0 的整数，%6.2f 表示输出宽度为 6、宽度不足时左填空格、含两位小数的实数。

比如，标量转换

```
sprintf('%6.2f', pi)
```

```
## [1] " 3.14"
```

又如，向量转换：

```
sprintf('tour%02d.jpg', 1:10)
```

```
## [1] "tour01.jpg" "tour02.jpg" "tour03.jpg" "tour04.jpg" "tour05.jpg"
```

```
## [6] "tour06.jpg" "tour07.jpg" "tour08.jpg" "tour09.jpg" "tour10.jpg"
```

还可以支持多个向量同时转换，如：

```
sprintf('%1dx%1d=%2d', 1:5, 5:1, (1:5)*(5:1))
```

```
## [1] "1x5= 5" "2x4= 8" "3x3= 9" "4x2= 8" "5x1= 5"
```

类似作用的函数还有 format, formatC 函数，但是 sprintf 的控制功能更强。

36.1.7 substring 函数

`substring` 函数求字符串子串，用开始字符位置和结束字符位置设定子串位置。如

```
substring(c("123456789", "abcdefg"), 3, 5)
```

```
## [1] "345" "cde"
```

不指定结束位置则取出剩余部分，如

```
substring(c("123456789", "abcdefg"), 3)
```

```
## [1] "3456789" "cdefg"
```

这个函数还允许修改某个字符串的指定子串的内容，如

```
s <- "123456789"
substring(s, 3, 5) <- "abc"
s
```

```
## [1] "12abc6789"
```

取子串时，一般按照字符个数计算位置，如

```
substring(" 北京市海淀区颐和园路 5 号", 4, 6)
```

```
## [1] "海淀区"
```

36.1.8 字符串查找

`startsWith(x, prefix)` 可以判断字符型向量 `x` 的每个元素是否以 `prefix` 开头，结果为一个与 `x` 长度相同的逻辑型向量。如

```
startsWith(c('xyz123', 'tu004'), 'tu')
```

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

`endsWith(x, suffix)` 可以判断字符型向量 `x` 的每个元素是否以 `suffix` 结尾，如

```
endsWith(c('xyz123', 'tu004'), '123')
```

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

函数 `grep()`, `grep1()` 等可以用于查找子字符串，位置不限于开头和结尾，详见“正则表达式”章节。

36.1.9 字符串替换

用 `gsub(pattern, replacement, x, fixed=TRUE)` 把字符型向量 `x` 中每个元素中出现的子串 `pattern` 都替换为 `replacement`。如

```
gsub('the', '**',
     c('New theme', 'Old times', 'In the present theme'),
     fixed=TRUE)
```

```
## [1] "New **me"          "Old times"          "In ** present **me"
```

例如，有些输入要求使用逗号 “,” 分隔，但是用户可能输入了中文逗号 “，”，可以用 `gsub()` 来替换：

```
x <- c('15.34,14.11', '13.25, 16.92')
x <- gsub(',', '，', x, fixed=TRUE); x
```

```
## [1] "15.34,14.11" "13.25,16.92"
```

例子中 `x` 的第二个元素中的逗号是中文逗号。

函数 `sub()` 与 `gsub()` 类似，但是仅替换第一次出现的 `pattern`。

36.1.10 字符串拆分

`strsplit(x,split,fixed=TRUE)` 可以把字符型向量 `x` 的每一个元素按分隔符 `split` 拆分为一个字符型向量，`strsplit` 的结果为一个列表，每个列表元素对应于 `x` 的每个元素。

如

```
x <- c('11,12', '21,22,23', '31,32,33,34')
res1 <- strsplit(x, split=',', fixed=TRUE)
res1
```

```
## [[1]]
## [1] "11" "12"
##
## [[2]]
## [1] "21" "22" "23"
##
## [[3]]
## [1] "31" "32" "33" "34"
```

得到了 3 个元素的结果列表 `res1`，每个结果元素是把输入按逗号拆分成的字符型向量。

如下的程序可以把 `res1` 的每个元素转换成数值型：

```
res2 <- lapply(res1, as.numeric); res2
```

```
## [[1]]
## [1] 11 12
##
## [[2]]
```

```
## [1] 21 22 23
##
## [[3]]
## [1] 31 32 33 34
```

res2 列表可以用如下程序把所有数值合并到一个向量中：

```
res3 <- unlist(res2); res3
```

```
## [1] 11 12 21 22 23 31 32 33 34
```

注意，即使输入只有一个字符串，结果也是列表，所以输入只有一个字符串时我们应该取出结果列表的第一个元素，如

```
strsplit('31,32,33,34', split=',', fixed=TRUE)[[1]]
```

```
## [1] "31" "32" "33" "34"
```

36.2 文本文件读写

文本文件是内容为普通文字、用换行分隔成多行的文件，与二进制文件有区别，二进制文件中换行符没有特殊含义，而且二进制文件的内容往往也不是文字内容。二进制文件的代表有图片、声音，以及各种专用软件的私有格式文件，如 Word 文件、Excel 文件。

对于文本文件，可以用 `readLines()` 函数将其各行的内容读入为一个字符型数组，字符型数组的每一个元素对应于文件中的一行，读入的字符型数组元素不包含分隔行用的换行符。

最简单的用法是读入一个本地的文本文件，一次性读入所有内容，用如

```
lines <- readLines("filename.ext")
```

其中 `filename.ext` 是文件名，也可以用全路径名或相对路径名。

当文本文件很大的时候，整体读入有时存不下，即使能存下处理速度也很慢，可以一次读入部分行，逐批读入并且逐批处理，这样程序效率更高。这样的程序要复杂一些，例如

```
infcon <- file("filename.ext", open="rt")
batch <- 1000
repeat{
  lines <- readLines(infcon, n=batch)
  if(length(lines)==0) break
  ## 处理读入的这些行
}
close(infcon)
```

以上程序先打开一个文件，`inffcon` 是打开的文件的读写入口（称为一个“连接对象”）。每次读入指定的行并处理读入的行，直到读入了 0 行为止，最后关闭 `infcon` 连接。

对文本文件的典型处理是读入后作一些修改，另外保存。函数 `writeLines(lines, con="outfilename.txt")` 可以将字符型向量 `lines` 的各个元素变成输出文件的各行保存起来，自动添加分隔行的换行符。如果是分批读入分批处理的，则写入也需要分批写入，以上的分批处理程序变成：

```
infcon <- file("filename.ext", open="rt")
outfcon <- file("outfilename.txt", open="wt")
batch <- 1000
while(TRUE){
  lines <- readLines(infcon, n=batch)
  if(length(lines)==0) break
  ## 处理读入的这些行，变换成 outlines
  writeLines(outlines, con=outfcon)
}
close(outfcon)
close(infcon)
```

`readLines()` 也可以直接读取网站的网页文件，如

```
lines <- readLines(url("https://www.r-project.org/"))
length(lines)
## [1] 116
head(lines)
## [1] "<!DOCTYPE html>"
## [2] "<html lang=\"en\">"
## [3] "  <head>"
## [4] "    <meta charset=\"utf-8\">"
## [5] "    <meta http-equiv=\"X-UA-Compatible\" content=\"IE=edge\">"
## [6] "    <meta name=\"viewport\" content=\"width=device-width, initial-scale=1\">"
```

`readr` 包的 `read_lines()` 和 `write_lines()` 函数起到与基本 R 中 `readLines()` 和 `writeLines()` 类似的作用，`read_file()` 和 `read_file_raw()` 可以将整个文件读入为一个字符串。

36.3 正则表达式

在对字符串进行查找或替换时，有时要查找替换的不是固定的子串而是某种模式。比如，要查找或替换连续三个数字，正文中的电子邮件地址，网址，电话号码，等等。正则表达式 (regular expressions) 用于表示各种复杂模式。

R 函数 `grep`, `sub`, `gsub`, `regexpr`, `gregexpr`, `regexec` 中的 `pattern` 参数可以是正则表达式，这时应设

参数 `fixed=FALSE`。`strsplit` 函数中的参数 `split` 也可以是正则表达式。

`regmatches` 函数从 `regexpr`, `gregexpr`, `regexec` 的结果中提取匹配的字符串。

正则表达式有多种定义，我们使用 perl 语言的正则表达式，在匹配中规定参数 `perl=TRUE`。

在正则表达式的模式 (pattern) 中，

`. * + ? { } \ [] ^ $ ()`

等字符是特殊字符，有特殊的解释。除了 \ 之外的其它 12 个都称为“元字符” (meta characters)。

在 R 语言中使用正则表达式时，需要注意 R 字符型常量中一个 \ 要写成两个。

36.3.1 原样匹配与匹配函数

如果模式中不含特殊字符，匹配为原样的子串。也叫做字面 (literal) 匹配。如

```
x <- c('New theme', 'Old times', 'In the present theme')
regexpr('the', x, perl=TRUE)
```

```
## [1] 5 -1 4
## attr("match.length")
## [1] 3 -1 3
## attr("useBytes")
## [1] TRUE
```

这里使用了 `regexpr` 函数。`regexpr` 函数的一般用法为：

```
regexpr(pattern, text, ignore.case = FALSE, perl = FALSE,
        fixed = FALSE, useBytes = FALSE)
```

自变量为：

- `pattern` 是一个正则表达式，如果用了 `fixed=TRUE` 选项，则当作普通原样文本来匹配；
- `text` 是源字符串向量，要从其每个元素中查找 `pattern` 模式出现的位置；
- `ignore.case`：是否要忽略大小写匹配；
- `perl` 选择是否采用 perl 格式，如果不把 `pattern` 当作普通原样文本，应该选 `perl=TRUE`，perl 语言的正则表达式是事实上的标准，所以这样兼容性更好；
- `fixed` 当 `fixed=TRUE` 时 `pattern` 作为普通原样文本解释；
- `useBytes` 为 `TRUE` 时逐字节进行匹配，否则逐字符进行匹配。之所以有这样的区别，是因为有些编码中一个字符由多个字节构成，BGK 编码的汉字由两个字节组成，UTF-8 编码的汉字也是由两个字节构成。

`regexpr()` 函数返回一个整数值的向量，长度与 `text` 向量长度相同，结果的每个元素是在 `text` 的对应元素中 `pattern` 的首次匹配位置；没有匹配时结果元素取 -1。结果会有一个 `match.length` 属性，表示每个匹配的长度，无匹配时取 -1。

如果仅关心源字符串向量 `text` 中哪些元素能匹配 `pattern`，可以用 `grep` 函数，如

```
grep('the', x, perl=TRUE)
```

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

结果说明源字符串向量的三个元素中仅有第 1、第 3 号元素能匹配。如果都不匹配，返回 `integer(0)`。

`grep` 可以使用与 `regexpr` 相同的自变量，另外还可以加选项 `invert=TRUE`，这时返回的是不匹配的元素的下标。

`grep()` 如果添加选项 `value=TRUE`，则结果不是返回有匹配的元素的下标而是返回有匹配的元素本身（不是匹配的子串），如

```
grep('the', x, perl=TRUE, value=TRUE)
```

```
## [1] "New theme"          "In the present theme"
```

`grepl` 的作用与 `grep` 类似，但是其返回值是一个长度与源字符串向量 `text` 等长的逻辑型向量，每个元素的真假对应于源字符串向量中对应元素的匹配与否。如

```
grepl('the', x, perl=TRUE)
```

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

就像 `grep()` 与 `grepl()` 本质上给出相同的结果，只是结果的表示方式不同，`regexec()` 与 `regexpr()` 也给出仅在表示方式上有区别的结果。`regexpr()` 主要的结果是每个元素的匹配位置，用一个统一的属性返回各个匹配长度；`regexec()` 则返回一个与源字符串向量等长的列表，列表的每个元素为匹配的位置，并且列表的每个元素有匹配长度作为属性。所以，这两个函数只需要用其中一个就可以，下面仅使用 `regexpr()`。`regexec()` 的使用效果如

```
regexec('the', x, perl=TRUE)
```

```
## [[1]]
## [1] 5
## attr("match.length")
## [1] 3
## attr("useBytes")
## [1] TRUE
##
## [[2]]
## [1] -1
## attr("match.length")
## [1] -1
## attr("useBytes")
## [1] TRUE
##
```

```
## [[3]]
## [1] 4
## attr(,"match.length")
## [1] 3
## attr(,"useBytes")
## [1] TRUE
```

`grep()`, `grepl()`, `regexpr()`, `regexec()` 都只能找到源字符串向量的每个元素中模式的首次匹配，不能找到所有匹配。`gregexpr()` 函数可以找到所有匹配。如

```
gregexpr('the', x, perl=TRUE)
```

```
## [[1]]
## [1] 5
## attr(,"match.length")
## [1] 3
## attr(,"useBytes")
## [1] TRUE
##
## [[2]]
## [1] -1
## attr(,"match.length")
## [1] -1
## attr(,"useBytes")
## [1] TRUE
##
## [[3]]
## [1] 4 16
## attr(,"match.length")
## [1] 3 3
## attr(,"useBytes")
## [1] TRUE
```

其结果是一个与源字符串向量等长的列表，格式与 `regexec()` 的结果格式类似，列表的每个元素对应于源字符串向量的相应元素，列表元素值为匹配的位置，并有属性 `match.length` 保存了匹配长度。匹配位置和匹配长度包含了所有的匹配，见上面例子中第三个元素的匹配结果。

函数 `grep`, `grepl` 结果仅给出每个元素能否匹配。`regexpr()`, `regexec()`, `gregexpr()` 则包含了匹配位置与匹配长度，这时，可以用 `regmatches()` 函数取出具体的匹配字符串。`regmatches()` 一般格式为

```
regmatches(x, m, invert = FALSE)
```

其中 `x` 是源字符串向量，`m` 是 `regexpr()`、`regexec()` 或 `gregexpr()` 的匹配结果。如

```
x <- c('New theme', 'Old times', 'In the present theme')
m <- regexpr('the', x, perl=TRUE)
regmatches(x, m)
```

```
## [1] "the" "the"
```

可以看出，`regmatches()` 仅取出有匹配时的匹配内容，无匹配的内容被忽略。

取出多处匹配的例子如：

```
x <- c('New theme', 'Old times', 'In the present theme')
m <- gregexpr('the', x, perl=TRUE)
regmatches(x, m)
```

```
## [[1]]
## [1] "the"
##
## [[2]]
## character(0)
##
## [[3]]
## [1] "the" "the"
```

当 `regmatches()` 第二个自变量是 `gregexpr()` 的结果时，其输出结果变成一个列表，并且不再忽略无匹配的元素，无匹配元素对应的列表元素为 `character(0)`，即长度为零的字符型向量。对有匹配的元素，对应的列表元素为所有的匹配字符串组成的字符型向量。

实际上，如果 `pattern` 中没有正则表达式，`grep()`，`grepl()`，`regexpr()`，`gregexpr()` 中都可以用 `fixed=TRUE` 参数取代 `perl=TRUE` 参数，这时匹配总是解释为原样匹配，即使 `pattern` 中包含特殊字符也是进行原样匹配。

36.3.2 不区分大小写匹配

正则表达式的模式一般是区分大小写的。为了不区分大小写匹配，可以在 `grep` 等函数调用时加选项 `ignore.case=TRUE`；或者在模式前面附加 `(?i)` 前缀式选项。如

```
grep('Dr', c("Dr. Wang", 'DR. WANG', 'dR. W.R.'))
```

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

```
grep('dr', c("Dr. Wang", 'DR. WANG', 'dR. W.R.'), ignore.case=TRUE)
```

```
## [1] 1 2 3
```

```
grep('(?!i)dr', c("Dr. Wang", 'DR. WANG', 'dR. W.R.'))
```

```
## [1] 1 2 3
```

36.3.3 用句点匹配单个字符

在模式中用 “.” 匹配任意一个字符（除了换行符 '\n'，能否匹配此字符与选项有关）。如

```
s <- c('abc', 'cabs', 'lab')
mres <- regexpr('ab.', s, perl=TRUE); mres
```

```
## [1] 1 2 -1
## attr("match.length")
## [1] 3 3 -1
## attr("useBytes")
## [1] TRUE
```

`regexpr` 仅给出每个元素中模式的首次匹配位置而不是给出匹配的内容。`regmatches` 函数以原始字符型向量和匹配结果为输入，结果返回每个元素中匹配的各个子字符串（不是整个元素），如：

```
regmatches(s, mres)
```

```
## [1] "abc" "abs"
```

注意返回结果和输入字符型向量元素不是一一对应的，仅返回有匹配的结果。

像句点这样的字符称为元字符（meta characters），在正则表达式中有特殊函数。如果需要匹配句点本身，用 “[.]” 或者 “\.” 表示。比如，要匹配 `a.txt` 这个文件名，如下做法有错误：

```
grep('a.txt', c('a.txt', 'a0txt'), perl=TRUE)
```

```
## [1] 1 2
```

结果连 `a0txt` 也匹配了。用 “[.]” 表示句点则将句点不做特殊解释：

```
grep('a[.]txt', c('a.txt', 'a0txt'), perl=TRUE)
```

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

```
grep('a\\.txt', c('a.txt', 'a0txt'), perl=TRUE)
```

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

注意在 R 语言字符型常量中一个 \ 需要写成两个。

如果仅需按照原样进行查找，也可以在 `grep()`，`grepl()`，`regexpr()`，`gregexpr()` 等函数中加选项 `fixed=TRUE`，这时不要再用 `perl=TRUE` 选项。如

```
grep('a.txt', c('a.txt', 'a0txt'), fixed=TRUE)
```

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

36.3.4 匹配一组字符中的某一个

模式中使用方括号给定一个字符类，单个字符与字符类中任何一个字符相同都算是匹配成功。比如，模式 “[ns]a.[.]xls” 表示匹配的第一个字符是 n 或 s，第二个字符是 a，第三个字符任意，第四个字符是句点，然后是 xls。例：

```
regexpr("[ns]a.[.]xls", c("sa1.xls", "dna2.xlss", "na3.xls"), perl=T)
```

```
## [1] 1 2 1
## attr("match.length")
## [1] 7 7 7
## attr("useBytes")
## [1] TRUE
```

注意匹配并不需要从开头匹配到结尾，中间匹配是允许的，类似于搜索符合某种规律的子串。在上例中第二个元素是从第二个字符开始匹配的。

例：模式 [Rr]eg[Ee]x 可以匹配 RegEx 或 Regex 或 regex 或 regEx。

如果希望完全忽略大小写进行匹配，可以使用在 regexpr 等函数中指定 ignore.case=TRUE 选项。

例：模式 “sa[0-9][.]xls” 要求匹配的第三个字符为数字。

在 “[]” 中允许用 - 表示一个范围。如 [a-z] 匹配小写英文字母，[A-Z] 匹配大写英文字母，[a-zA-Z] 匹配大小写的英文字母，[a-zA-Z0-9] 匹配大小写的英文字母和数字。

为了匹配一个 16 进制数字，可以用 [0-9A-Fa-f]。

在方括号内第一个位置的 ^ 表示对指定的范围取余集。例如，模式 sa[^0-9][.]xls 要求匹配的第三个字符不能为数字。

36.3.5 原样匹配元字符

元字符 (meta characters) 是在正则表达式中有特殊含义的字符。比如句点可以匹配任意一个字符，左方括号代表字符集合的开始。所以元字符不能直接匹配自身，可以用 “[.]” 匹配一个句点。为匹配左方括号，在前面加上转义字符 \ 变成 \[，但是在 R 字符串中一个 \ 必须用 \\ 表示，所以模式 “[\” 在 R 中写成字符串常量，必须写成 '\\[’。其它的元字符如果要原样匹配也可以在前面加上转义字符 \，比如匹配 \ 本身可以用 \\，但是在 R 字符型常量中需要写成 '\\\\’。如

```
grep("int x\\[5\\]", c("int x;", "int x[5]"), perl=T)
```

```
## [1] 2
```

也可以用 “[]” 表示 “[”，用 “[]” 表示 “]”，如

```
grep("int x[[]5[]]", c("int x;", "int x[5]"), perl=T)
```

```
## [1] 2
```

36.3.6 匹配空白

表示空白的元字符有：

```
\f 换页符
\n 换行符
\r 回车符
\t 制表符
\v 垂直制表符
```

不同操作系统的文本文件的行分隔符不同，为了匹配 Windows 格式的文本文件中的空行，用 “\r\n\r\n”；为了匹配 Unix 格式的文本文件中的空行则用 “\r\n”。写成 R 的字符型常量时，这些表示本身也是 R 的相应字符的表示，所以在 R 字符型常量中这些字符不需要用两个\表示一个\。

匹配任意一个空白字符用 “\s”，这等价于 “[\f\n\r\t\v]”。大写的 “\S” 则匹配任意一个非空白的字符。

36.3.7 匹配数字

用\d 匹配一个数字，相当于 [0-9]。用\D 匹配一个非数字。如

```
grep("n\\d[.]xls", c("n1.xls", "na.xls"), perl=TRUE)
```

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

36.3.8 匹配开头和末尾

模式匹配相当于在字符串内部搜索某种模式，如果要从字符串开头匹配，在模式中取第一个字符为 ^ 或\A。如果模式中最后一个字符是 \$ 或\Z，则需要匹配到字符串末尾。用\Z 匹配字符串末尾时如果末尾有一个换行符则匹配到换行符之前。

如

```
grep("^n\\d[.]xls$", c("n1.xls", "na.xls", "cn1.xls", "n1.xlsx"), perl=TRUE)
```

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

```
grep("\\An\\d[.]xls\\Z", c("n1.xls", "na.xls", "cn1.xls", "n1.xlsx"), perl=TRUE)
```

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

只匹配了第一个输入字符串。

有时候源文本的每个字符串保存了一个文本文件内容，各行用\n 分隔。后面将给出匹配每行的行首与行尾的方法。

36.3.9 匹配字母、数字、下划线

匹配字母、数字、下划线字符用 \w，等价于 [a-zA-Z0-9_]。 \W 匹配这些字符以外的字符。如

```
m <- regexpr("s\\w[.]", c("file-s1.xls", "s#.xls"), perl=TRUE)
regmatches(c("file-s1.xls", "s#.xls"), m)
```

```
## [1] "s1."
```

可以看出，模式匹配了 s1. 而没有匹配 s#.。

36.3.10 十六进制和八进制数

在模式中可以用十六进制数和八进制数表示特殊的字符。十六进制数用 \x 引入，比如 \x0A 对应 \n 字符。八进制数用 \0 引入，比如 \011 表示 \t 字符。

例如

```
gregexpr("\\x0A", "abc\\nefg\\n")[[1]]
```

```
## [1] 4 8
## attr("match.length")
## [1] 1 1
## attr("useBytes")
## [1] TRUE
```

匹配了两个换行符。

36.3.11 POSIX 字符类

\d, \w 这样的字符类不方便用在方括号中组成字符集合，而且也不容易记忆和认读。在模式中方括号内可以用 [:alpha:] 表示任意一个字母。比如，[:alpha:] 匹配任意一个字母（外层的方括号表示字符集合，内层的方括号是 POSIX 字符类的固有界定符）。

这样的 POSIX 字符类有：

- [:alpha:] 表示任意一个字母；
- [:lower:] 为小写字母；
- [:upper:] 为大写字母；
- [:digit:] 为数字；

- `[:xdigit:]` 为十六进制数字。
- `[:alnum:]` 为字母数字 (不包括下划线);
- `[:blank:]` 为空格或制表符;
- `[:space:]` 为任何一种空白字符, 包括空格、制表符、换页符、换行符、回车符;
- `[:print:]` 为可打印字符;
- `[:graph:]` 和 `[:print:]` 一样但不包括空格;
- `[:punct:]` 为 `[:print:]` 中除 `[:alnum:]` 和空白以外的所有字符;

例如:

```
grep("[:alpha:]_."[:alnum:]_.", c('x1', '_x', '.x', '.1'))
```

```
## [1] 1 2 3 4
```

模式试图匹配长度为 2 的 R 合法变量名, 但是最后一个非变量名 `.1` 也被匹配了。解决这样的问题可以采用后面讲到的 | 备择模式。

36.3.12 加号重复匹配

模式中在一个字符或字符集合后加后缀 `+` 表示一个或多个前一字符。比如

```
s <- c("sa1", "dsa123")
mres <- regexpr("sa[:digit:]+", s, perl=TRUE)
regmatches(s, mres)
```

```
## [1] "sa1" "sa123"
```

例如:

```
p <- "^[:alnum:]_+@[:alnum:]_+.[[:alnum:]_]+$"
x <- "abc123@efg.com"
m <- regexpr(p, x, perl=TRUE)
regmatches(x, m)
```

```
## [1] "abc123@efg.com"
```

匹配的电子邮件地址在 `@` 前面可以使用任意多个字母、数字、下划线, 在 `@` 后面由小数点分成两段, 每段可以使用任意多个字母、数字、下划线。这里用了 `^` 和 `$` 表示全字符串匹配。

36.3.13 星号和问号重复匹配

在一个字符或字符集合后加后缀 `*` 表示零个或多个前一字符, 后缀 `?` 表示零个或一个前一字符。

比如, `^https?:/[[:alnum:]]/+$` 可以匹配 `http` 或 `https` 开始的网址。如


```
s <- c('http://www.163.net', 'https://123.456.')
grep('^https?:/[[:alnum:]]_./+$', s, perl=TRUE)
```

```
## [1] 1 2
```

(注意第二个字符串不是合法网址但是按这个正则表达式也能匹配)

例: `x\\d*` 能匹配 “x”, “x1”, “x123” 这样的变量名。

36.3.14 计数重复

问号可以表示零个或一个，而加号、星号重复不能控制重复次数。在后缀大括号中写一个整数表示精确的重复次数。如

```
grep('[[[:digit:]]{3}', c('1', '12', '123', '1234'))
```

```
## [1] 3 4
```

模式匹配的是三位的数字。

可以在后缀大括号中指定重复的最小和最大次数，中间用逗号分隔。比如，月日年的日期格式可以用

```
[[[:digit:]]{1,2}[-/] [[[:digit:]]{1,2}[-/] [[[:digit:]]{2,4}
```

来匹配。如 (注意这个模式还会匹配非日期)

```
pat <- paste(
  c("[[:digit:]]{1,2}[-/]",
    "[[:digit:]]{1,2}[-/]",
    "[[:digit:]]{2,4}"), collapse="")
grep(pat, c("2/4/1998", "13/15/198"))
```

```
## [1] 1 2
```

重复数允许指定为 0。重复数的逗号后面空置表示重复数没有上限。例如，后缀 `{3,}` 表示前一模式必须至少重复 3 次。

36.3.15 贪婪匹配和懒惰匹配

无上限的重复匹配如 `*`, `+`, `{3,}` 等缺省是贪婪型的，重复直到文本中能匹配的最长范围。比如我们希望找出 HTML 源文件中 “.....” 这样的结构，很容易想到用 `.+` 这样的模式，但是这不会恰好匹配一次，模式会一直搜索到最后一个 `` 为止。

例如：

```
s <- '<B>1st</B> other <B>2nd</B>'
p1 <- '<[Bb]>.*</[Bb]>'
```

```
m1 <- regexpr(p1, s, perl=TRUE)
regmatches(s, m1)[[1]]
```

```
## [1] "<B>1st</B> other <B>2nd</B>"
```

我们本来期望的是提取第一个 “.....” 组合，不料提取了两个 “.....” 组合以及中间的部分。

如果要求尽可能短的匹配，使用 `*?`, `+?`, `{3,}?` 等“懒惰型”重复模式。在无上限重复标志后面加问号表示懒惰性重复。

比如，上例中模式修改后得到了期望的结果：

```
s <- '<B>1st</B> other <B>2nd</B>'
p2 <- '<[Bb]>.*?</[Bb]>'
m2 <- regexpr(p2, s, perl=TRUE)
regmatches(s, m2)[[1]]
```

```
## [1] "<B>1st</B>"
```

懒惰匹配会造成搜索效率降低，应仅在需要的时候使用。

36.3.16 单词边界

用 `\b` 匹配单词边界，这样可以查找作为单词而不是单词的一部分存在的内容。`\B` 匹配非单词边界。如

```
grep('\\bcat\\b', c('a cat meaos', 'the category'))
```

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

36.3.17 多行和单行模式

句点通配符一般不能匹配换行，如

```
s <- '<B>1st\n</B>\n'
grep('<[Bb]>.*?</[Bb]>', s, perl=TRUE)
```

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

跨行匹配失败。而在 HTML 的规范中换行是正常的。一种办法是预先用 `gsub` 把所有换行符替换为空格。但是这只能解决部分问题。

另一方法是在 Perl 正则表达式开头添加 `(?s)` 选项，这个选项使得句点通配符可以匹配换行符。如

```
s <- '<B>1st\n</B>\n'
mres <- regexpr('(s)<[Bb]>.*?</[Bb]>', s, perl=TRUE)
regmatches(s, mres)
```

```
## [1] "<B>1st\n</B>"
```

在 Perl 规则的正则表达式开头用 `(?m)` 表示把整个输入字符串看成用换行符分开的多行。这时 `^` 和 `$` 匹配每行的开头和结尾，“每行”是指字符串中用换行符分开的各个字符串。`(?s)` 与 `(?m)` 可以同时使用，它们没有矛盾：`(?s)` 使得句点通配符可以匹配换行符，`(?m)` 使得 `^` 和 `$` 匹配每行的首尾而不是整个字符串的首尾。如

```
s <- '<B>1st\n</B>\n'
mres1 <- gregexpr('^<.+?>', s, perl=TRUE)
mres2 <- gregexpr('(?m)^<.+?>', s, perl=TRUE)
regmatches(s, mres1)[[1]]
```

```
## [1] "<B>"
```

```
regmatches(s, mres2)[[1]]
```

```
## [1] "<B>" "</B>"
```

字符串 `s` 包含两行内容，中间用 `\n` 分隔。`mres1` 的匹配模式没有打开多行选项，所以模式中的 `^` 只能匹配 `s` 中整个字符串开头。`mres2` 的匹配模式打开了多行选项，所以模式中的 `^` 可以匹配 `s` 中每行的开头。

36.3.18 逐行处理

R 的 `readLines()` 函数可以把一整个文本文件读成一个字符型向量，每个元素为一行，元素中不包含换行符。R 的字符型函数可以对这样的字符型向量每个元素同时处理，也就实现了逐行处理。

如果字符串 `x` 中包含了一整个文本文件内容，其中以 `\n` 分隔各行，为了实现逐行处理，可以先用 `strsplit()` 函数拆分成不同行：

```
cl <- strsplit(x, split='\\r?\\n', fixed=FALSE, perl=TRUE)[[1]]
```

结果将是一个字符型向量，每个元素是原来的一行。如

```
x <- c('This is first line.\\nThis is second line.\\n')
cl <- strsplit(x, split='\\r?\\n', fixed=FALSE, perl=TRUE)[[1]]
cl
```

```
## [1] "This is first line." "This is second line."
```

36.3.19 备择模式

如果有两种模式都算正确匹配，则用 `|` 连接这两个模式表示两者都可以。例如，某人的名字用 James 和 Jim 都可以，表示为 `James|Jim`，如

```
s <- c('James, Bond', 'Jim boy')
pat <- 'James|Jim'
```

```
mres <- gregexpr(pat, s, perl=TRUE)
regmatches(s, mres)
```

```
## [[1]]
## [1] "James"
##
## [[2]]
## [1] "Jim"
```

36.3.20 分组与捕获

在正则表达式中用圆括号来分出组，作用是确定优先规则、组成一个整体、拆分出模式中的部分内容（称为捕获）。类似于 R 语言中的复合语句。在替换时，可以用\1, \2 等表示匹配中第一个开括号对应的分组，第二个开括号对应的分组，.....。

例：希望把“.....”两边的“”和“”删除，可以用如下的替换方法：

```
x <- '<B>1st</B> other <B>2nd</B>'
pat <- '(?s)<[Bb]>(.*?)</[Bb]>'
repl <- '\\1'
gsub(pat, repl, x, perl=TRUE)
```

```
## [1] "1st other 2nd"
```

替换模式中的\1(写成 R 字符型常量时\要写成\\)表示第一个圆括号匹配的内容，但是表示选项的圆括号((?s))不算在内。

例：希望把带有前导零的数字的前导零删除，可以用如

```
x <- c('123', '0123', '00123')
pat <- '\\b0+([1-9][0-9]*)\\b'
repl <- '\\1'
gsub(pat, repl, x, perl=TRUE)
```

```
## [1] "123" "123" "123"
```

其中的\b 模式表示单词边界，这可以排除在一个没有用空格或标点分隔的字符串内部拆分出数字的情况。

例：为了交换横纵坐标，可以用如下替换

```
s <- '1st: (5,3.6), 2nd: (2.5, 1.1)'
pat <- paste(
  '([[:digit:]]+)',
  '[:space:]*([[:digit:]]+)', sep='')
gsub(pat, paste('\\2', '\\1'), s)
```

```
repl <- '(\2, \1)'
gsub(pat, repl, s, perl=TRUE)
```

```
## [1] "1st: (3.6, 5), 2nd: (1.1, 2.5)"
```

例如，要匹配 yyyy-mm-dd 这样的日期，并将其改写为 mm/dd/yyyy，就可以用这样的替换模式：

```
pat <- '([0-9]{4})-([0-9]{1,2})-([0-9]{1,2})'
repl <- '\2/\3/\1'
gsub(pat, repl, c('1998-05-31', '2017-01-14'))
```

```
## [1] "05/31/1998" "01/14/2017"
```

分组除了可以做替换外，还可以用来表示模式中的重复出现内容。例如，([a-z]{3})\1 这样的模式可以匹配如 abcabc, uxzuxz 这样的三字母重复。如

```
grep('[a-z]{3})\1', c('abcabc', 'aabbcc'))
```

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

又例如，下面的程序找出了年（后两位）、月、日数字相同的日期：

```
x <- c('2008-08-08', '2017-01-18')
m <- regexpr('\d\d(\d\d)-\1-\1', x)
regmatches(x, m)
```

```
## [1] "2008-08-08"
```

如果某个分组仅想到分组作用但是不会提取具体的匹配内容也不会用该组内容做替换，可以将该组变成“非捕获分组”，办法是把表示分组开始左圆括号变成 (?: 三个字符。这在用分组表示优先级时比较有用。非捕获分组在替换时不计入\1、\2 这样的排列中。比如，把 1921-2020 之间的世纪号删去，可以用

```
pat <- '\\A(?:19|20)([0-9]{2})\\Z'
repl <- '\\1'
x <- c('1978', '2017', '2035')
gsub(pat, repl, x, perl=TRUE)
```

```
## [1] "78" "17" "35"
```

其中用了非捕获分组使得备择模式 19|20 优先匹配。注意模式并没有能保证日期在 1921-2020 之间。更周密的程序可以写成：

```
x <- c('1978', '2017', '2035')
p1 <- '\\A19(2[1-9] | [3-9][0-9])\\Z'
r1 <- '\\1'
p2 <- '\\A20([01][0-9] | 20)\\Z'
x <- gsub(p1, r1, x, perl=TRUE)
```

```
x <- gsub(p2, r1, x, perl=TRUE)
x

## [1] "78"    "17"    "2035"
```

36.4 stringr 包

stringr 扩展包与 stringi 扩展包提供了更方便使用的字符串函数。stringr 包的函数以 `str_` 开头, stringi 包的函数以 `stri_` 开头。其对字符串的处理可以在不同操作系统下得到一致的效果, 而基本 R 软件的字符串处理受到一些操作系统设置的影响。stringr 和 stringi 支持正则表达式, 使用与 perl 规则类似的 ICU 正则表达式。

```
library(stringr)
```

36.4.1 字符串长度

`str_length()` 求字符型向量每个元素的长度。一个汉字长度为 1。

36.4.2 连接字符串

`str_c()` 起到与 `paste()` 相同的作用, 用 `sep` 指定分隔符, 用 `collapse` 指定将多个元素合并时的分隔符。默认的分隔符与 `paste()` 不同, 默认是没有分隔。

字符型缺失值参与连接时, 结果变成缺失值; 可以用 `str_replace_na()` 函数将待连接的字符型向量中的缺失值转换成字符串 "NA" 再连接。

36.4.3 取子串

`str_sub()` 起到与 `substring()` 相同的作用。增强的地方是, 允许开始位置与结束位置用负数, 这时最后一个字符对应-1, 倒数第二个字符对应-2, 以此类推。如果要求取的子串没有那么长就有多少取多少, 如果起始位置就已经超过总长度就返回空字符串。

如:

```
str_sub("term2017", 5, 8)

## [1] "2017"
```

36.4.4 按指定的 locale 排序

`str_sort()` 对字符型向量排序, 可以用 `locale` 选项指定所依据的 locale, 不同的 locale 下次序不同。通常的 locale 是 “en”(英语), 中国大陆的 GB 编码对应的 locale 是 “zh”。

36.4.5 长行分段

`str_wrap()` 可以将长字符串拆分为基本等长的行。

36.4.6 删除首尾空格

`str_trim()` 起到与 `trimws()` 相同的作用, 删除首尾空格, 也可以要求仅删除开头空格 (指定 `side="left"`) 或者仅删除结尾空格 (指定 `side="right"`)。

36.4.7 查看匹配

`str_view(string, pattern)` 在 RStudio 中打开 View 窗格, 显示 `pattern` 给出的正则表达式模式在 `string` 中的首个匹配。`string` 是输入的字符型向量。用 `str_view_all()` 显示所有匹配。

如果要匹配的是固定字符串, 写成 `str_view(string, fixed(pattern))`。

如果要匹配的是单词等的边界, 模式用 `boundary()` 函数表示, 如 `str_view("a brown fox", boundary("word"))` 将匹配首个单词。

36.4.8 匹配与否

`str_detect(string, pattern)` 返回字符型向量 `string` 的每个元素是否匹配 `pattern` 中的模式的逻辑型结果。与 `grepl()` 作用类似。

`str_count()` 则返回每个元素匹配的次数。如

```
str_count(c("123,456", "011"), "[[:digit:]]")
```

```
## [1] 6 3
```

36.4.9 返回匹配的元素

`str_subset(string, pattern)` 返回字符型向量中能匹配 `pattern` 的那些元素组成的子集, 与 `grep(pattern, string, value=TRUE)` 效果相同。比如查找人名中间有空格的:

```
str_subset(c(" 马思聪", " 李 明"), "[[:alpha:]]+[[:space:]]+[[:alpha:]]+")
```

```
## [1] "李 明"
```

当要查找的内容是 tibble 的一列时，用 `filter()` 与 `str_detect()` 配合。比如，在数据框的人名中查找中间有空格的名字：

```
tibble(name=c(" 马思聪", " 李 明")) %>%
  filter(str_detect(name, "[[:alpha:]]+[[:space:]]+[[:alpha:]]+"))
```

```
## # A tibble: 1 x 1
##   name
##   <chr>
## 1 李 明
```

36.4.10 提取匹配内容

`str_subset()` 返回的是有匹配的源字符串，而不是匹配的部分子字符串。用 `str_extract(string, pattern)` 从源字符串中取出首次匹配的子串。

如

```
str_extract("A falling ball", "all")
```

```
## [1] "all"
```

`str_extract_all()` 取出所有匹配子串，这时可以加选项 `simplyfy=TRUE`，使得返回结果变成一个字符型矩阵，每行是原来一个元素中取出的各个子串。

36.4.11 提取分组捕获内容

`str_match()` 提取匹配内容以及各个捕获分组内容。如：

```
str_match(c(" 马思聪", " 李 明"), "([[:alpha:]]+)[[:space:]]+([[:alpha:]]+)")
```

```
##      [,1]      [,2] [,3]
## [1,] NA      NA    NA
## [2,] "李 明" "李"  "明"
```

36.4.12 替换

用 `str_replace_all()` 实现与 `gsub()` 类似功能。如：

```
str_replace_all(c("123,456", "011"), ",", "")
```

```
## [1] "123456" "011"
```

注意参数次序与 `gsub()` 不同。

36.4.13 字符串拆分

`str_split()` 起到与 `strsplit()` 类似作用，并且可以加 `simplify=TRUE` 选项使得原来每个元素拆分出的部分存入结果矩阵的一行中。可以用 `boundary()` 函数指定模式为单词等边界。

36.4.14 定位匹配位置

`str_locate()` 和 `str_locate_all()` 返回匹配的开始和结束位置。注意如果需要取出匹配的元素可以用 `str_subset()`，要取出匹配的子串可以用 `str_extract()`，取出匹配的分组捕获可以用 `str_match()`。

36.4.15 `regex()` 函数

前面用到正则表达式模式的地方，实际上应该写成 `regex(pattern)`，只写模式本身是一种简写。`regex()` 函数可以指定 `ignore_case=TRUE` 要求不区分大小写，指定 `multi_line=TRUE` 使得 `^` 和 `$` 匹配用换行符分开的每行的开头和结尾，`dotall=TRUE` 使得 `.` 能够匹配合换行符。`comment=TRUE` 使得模式可以写成多行，行尾的井号后面表示注释，这时空格不再原样匹配，为了匹配空格需要写在方括号内或者用反斜杠开头。与 `regex()` 类似的表示模式的函数有 `fixed()`，`boundary()`，`coll()`。

36.5 正则表达式应用例子

36.5.1 数据预处理

在原始数据中，经常需要审核数据是否合法，已经把一些常见错误输入自动更正。这都可以用正则表达式实现。

36.5.1.1 除去字符串开头和结尾的空格

函数 `trimws()` 可以除去字符串开头与结尾的空格，也可以仅除去开头或仅除去结尾的空格。

这个任务如果用字符串替换函数来编写，可以写成：

```
### 把字符串向量 x 的元素去除首尾的空白。
strip <- function(x){
  x <- gsub("^[:space:]+", "", x, perl=TRUE)
  x <- gsub("[:space:]+$", "", x, perl=TRUE)
  x
}
```

这个版本可以除去包括空格在内的所有空白字符。

36.5.1.2 除去字符串向量每个元素中所有空格

```
compress <- function(x){
  x <- gsub(" ", "", x, fixed=TRUE)
  x
}
```

这可以解决"李明"与"李 明"不相等这样的问题。这样的办法也可以用来把中文的标点替换成英文的标点。

36.5.1.3 判断日期是否合法

设日期必须为 yyyy-mm-dd 格式, 年的数字可以是两位、三位、四位, 程序为:

```
is.yyyymmdd <- function(x){
  pyear <- '([0-9]{2}|[1-9][0-9]{2}|[1-2][0-9]{3})'
  pmon <- '([1-9]|0[1-9]|1[0-2])'
  pday <- '([1-9]|0[1-9]|1[0-9]|2[0-9]|3[01])'
  pat <- paste('\\A', pyear, '-', pmon, '-', pday, '\\Z', sep='')
  grepl(pat, x, perl=TRUE)
}
```

这样的规则还没有排除诸如 9 月 31 号、2 月 30 号这样的错误。

例如

```
x <- c('49-10-1', '1949-10-01', '532-3-15', '2015-6-1',
      '2017-02-30', '2017-13-11', '2017-1-32')
is.yyyymmdd(x)
```

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

注意错误的 2 月 30 号没有识别出来。

36.5.1.4 把字符型日期变成 yyyy-mm-dd 格式。

```
make.date <- function(x){
  x <- trimws(x)
  x <- gsub("[[:space:]]+", "-", x)
  x <- gsub("/", "-", x)
  x <- gsub("[.]", "-", x)
  x <- gsub("^([0-9]{2})(-[0-9]{1,2}-[0-9]{1,2})$", "20\\1\\2", x)
  x <- gsub("^([0-9]{4})-([0-9])-([0-9]{1,2})$", "\\1-0\\2-\\3", x)
```

```
x <- gsub("^[0-9]{4}-[0-9]{2}-([0-9])$", "\\1-0\\2", x)

x
}
```

另一办法是用 `strsplit()` 拆分成三个部分，转换为整数，再转换回字符型。

36.5.1.5 合并段落为一行

在某些纯文本格式中，各段之间用空行分隔，没有用空行分隔的各行看成同一段。如下的函数把其中的不表示分段的换行删去从而合并这些段落。函数以一个文件名作为输入，合并段落后存回原文件。注意，这样修改文件的函数在调试时，应该注意先备份文件，等程序没有任何错误以后才可以忽略备份。

```
combine.paragraph <- function(fname){
  lines <- readLines(fname)
  s <- paste(lines, collapse="\n")
  s <- gsub("^[[:space:]]+\n", "\n", s, perl=TRUE)
  s <- gsub('([^\n]+\n)', '\\1 ', s, perl=TRUE)
  s <- gsub('([^\n]+\n)', '\\1\n\n', s, perl=TRUE)
  writeLines(strsplit(s, '\n', fixed=TRUE)[[1]],
             con=fname)
}
```

函数首先把仅有空格的行中的空格删除，将有内容的行的行尾换行符替换成一个空格，再把剩余的有内容的行的行尾换行符多加一个换行符。

36.5.2 不规则 Excel 文件处理

- 作为字符型数据处理示例，考察如下的一个 Excel 表格数据。

假设一个中学把所有课外小组的信息汇总到了 Excel 表的一个工作簿中。每个课外小组占一块区域，各小组上下排列，但不能作为一个数据框读取。下图为这样的文件的一个简化样例：

实际数据可能有很多个小组，而且数据是随时更新的，所以复制粘贴另存的方法不太可行，需要一个通用的程序处理。Excel 文件 (.xls 后缀或.xlsx 后缀) 不是文本型数据。在 Excel 中，用“另存为”把文件保存为 CSV 格式，内容如下：

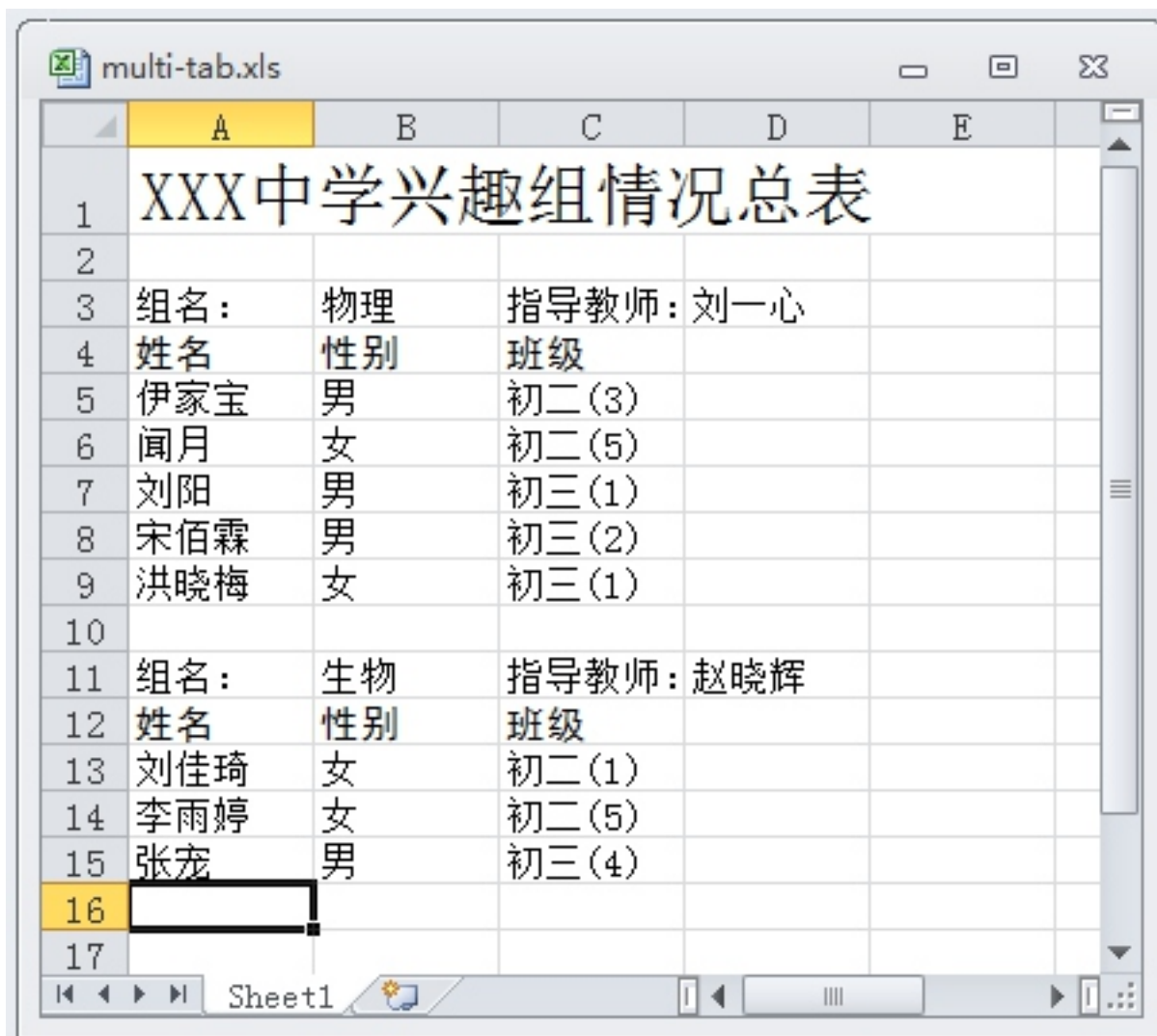
XXX中学兴趣组情况总表,,,

,,,

组名：,物理,指导教师：,刘一心

姓名,性别,班级,

伊家宝,男,初二(3),



| | A | B | C | D | E |
|----|--------------|----|-------|-----|---|
| 1 | XXX中学兴趣组情况总表 | | | | |
| 2 | | | | | |
| 3 | 组名: | 物理 | 指导教师: | 刘一心 | |
| 4 | 姓名 | 性别 | 班级 | | |
| 5 | 伊家宝 | 男 | 初二(3) | | |
| 6 | 闻月 | 女 | 初二(5) | | |
| 7 | 刘阳 | 男 | 初三(1) | | |
| 8 | 宋佰霖 | 男 | 初三(2) | | |
| 9 | 洪晓梅 | 女 | 初三(1) | | |
| 10 | | | | | |
| 11 | 组名: | 生物 | 指导教师: | 赵晓辉 | |
| 12 | 姓名 | 性别 | 班级 | | |
| 13 | 刘佳琦 | 女 | 初二(1) | | |
| 14 | 李雨婷 | 女 | 初二(5) | | |
| 15 | 张宠 | 男 | 初三(4) | | |
| 16 | | | | | |
| 17 | | | | | |

图 36.1: 不规则 Excel 文件样例图形

闻月,女,初二(5),
 刘阳,男,初三(1),
 宋佰霖,男,初三(2),
 洪晓梅,女,初三(1),
 ,,,
 组名: ,生物,指导教师: ,赵晓辉
 姓名,性别,班级,
 刘佳琦,女,初二(1),
 李雨婷,女,初二(5),
 张宠,男,初三(4),

生成测试用的数据文件:

```

demo.multitab.data <- function(){
  s <- "
  XXX 中学兴趣组情况总表,,,
  ,,,
  组名: , 物理, 指导教师: , 刘一心
  姓名, 性别, 班级,
  伊家宝, 男, 初二 (3),
  闻月, 女, 初二 (5),
  刘阳, 男, 初三 (1),
  宋佰霖, 男, 初三 (2),
  洪晓梅, 女, 初三 (1),
  ,,,
  组名: , 生物, 指导教师: , 赵晓辉
  姓名, 性别, 班级,
  刘佳琦, 女, 初二 (1),
  李雨婷, 女, 初二 (5),
  张宠, 男, 初三 (4),
  "
  writeLines(s, 'multitab.csv')
}
demo.multitab.data()
  
```

读入测试用的数据, 转换为多个数据框:

```

demo.multitab1 <- function(){
  ## 读入所有行
  lines <- readLines('multitab.csv')

  ## 删去所有空行和只有逗号的行
  }
  
```

```

empty <- grep('^[:space:],]*$', lines)
if(length(empty)>0){
  lines <- lines[-empty]
}

## 找到所有包含 ``组名: `` 的行对应的行号
heads <- grep(' 组名: ', lines, fixed=TRUE)

## 定位每个表的开始行和结束行 (不包括组名和表头所在的行)
start <- heads + 2
end <- c(heads[-1]-1, length(lines))
ngroups <- length(heads)

## 先把数据读入一个列表。
resl <- vector(ngroups, mode="list")
for(ii in seq(along=resl)){
  item <- list()
  line <- lines[heads[ii]]
  v <- strsplit(line, ',')[[1]]
  item[[' 组名']] <- v[2]
  item[[' 指导教师']] <- v[4]
  s <- paste(lines[start[ii]:end[ii]], collapse='\n')
  con <- textConnection(s, 'rt')
  item[[' 学生名单']] <-
    read.csv(con, header=FALSE,
             colClasses=c(
               "character", "character", "character", "NULL"
             ))
  close(con)
  names(item[[" 学生名单"]]) <- c(" 姓名", " 性别", " 班级")

  resl[[ii]] <- item
}

resl
}

resl <- demo.multitab1()

```

在程序中，用 `readLines` 函数读取文本文件各行到一个字符型向量。用 `grep` 可以找到每个小组开头的行（有“组名：”的行）。然后可以找出每个小组学生名单的开始行号和结束行号。各小组循环处理，读入后每

个小组先保存为列表的一个元素。用 `strsplit` 函数拆分用逗号分开的数据项。用 `textConnection` 函数可以把一个字符串当作文件读取，这样 `read.csv` 函数可以从一个字符串读入数据。

需要的话可以把所有组整合为一个大数据框。用两种方法，第一种方法是将各个数据框纵向合并：

```
demo.multitab2 <- function(resl){
  ngroups <- length(resl)
  for(ii in seq(along=resl)){
    resl[[ii]][[" 学生名单"]][," 组名"] <- resl[[ii]][[" 组名"]]
    resl[[ii]][[" 学生名单"]][," 指导教师"] <- resl[[ii]][[" 指导教师"]]
  }
  resd <- resl[[1]][[" 学生名单"]]
  if(ngroups > 1){
    for(ii in 2:ngroups){
      resd <- rbind(resd, resl[[ii]][[" 学生名单"]])
    }
  }
  resd <- resd[, c(" 组名", " 指导教师", " 姓名", " 性别", " 班级")]

  resd
}
d1 <- demo.multitab2(resl)
```

这个做法的优点是程序简单，缺点是多次合并，每次结果数据框是按行增大的，没有预先确定结果数据框的大小。

合并数据框的第二种做法是预先确定结果数据框的大小，然后填入各个行子集。程序如下：

```
demo.multitab3 <- function(resl){
  ngroups <- length(resl)
  nstu <- vapply(resl, function(item) nrow(item[[" 学生名单"]]), 1L)
  ntot <- sum(nstu)
  resd <- data.frame(" 组名"="", " 指导教师"="",
                    " 姓名"=character(ntot), " 性别"="", " 班级"="",
                    stringsAsFactors = FALSE)

  ndone <- 0
  for(ii in seq(along=resl)){
    inds <- ndone + seq(nstu[ii])
    resd[inds, " 组名"] <- resl[[ii]][[" 组名"]]
    resd[inds, " 指导教师"] <- resl[[ii]][[" 指导教师"]]
    resd[inds, c(" 姓名", " 性别", " 班级")] <- resl[[ii]][[" 学生名单"]]
    ndone <- ndone + nstu[ii]
  }
}
```

```

}

resd
}
d2 <- demo.multitab3(res1)

```

36.5.3 网站数据获取

很多网站定期频繁发布数据，所以传统的手工复制粘贴整理是不现实的。

但是，这些数据网页往往有固定模式，如果网页不是依赖 JavaScript 来展示的话，可以读取网页然后通过字符型数据处理方法获得数据。

url 可以打开一个网址链接然后用 `readLines` 函数读取。用 `gsub` 去掉不需要的成分。用 `grep` 查找关键行。具体例子略。

36.5.4 数字验证

36.5.4.1 整数

字符串完全为十进制正整数的模式，写成 R 字符型常量：

```
'\\A[0-9]+\\Z'
```

这个模式也允许正整数以 0 开始，如果不允许以零开始，可以写成

```
'\\A[1-9][0-9]*\\Z'
```

对于一般的整数，字符串完全为十进制整数，但是允许前后有空格，正负号与数字之间允许有空格，模式可以写成：

```
'\\A[ ]*[+-]?[ ]*[1-9][0-9]*\\Z'
```

36.5.4.2 十六进制数字

字符串仅有十六进制数字，模式写成 R 字符型常量为

```
'\\A[0-9A-Fa-f]+\\Z'
```

在文中匹配带有 0x 前缀的十六进制数字，模式为

```
'\\b0x[0-9A-Fa-f]+\\b'
```


36.5.4.3 二进制数字

为了在文中匹配一个以 B 或 b 结尾的二进制非负整数，可以用

```
'\\b[01]+[Bb]\\b'
```

36.5.4.4 有范围的整数

1-12:

```
'\\b1[012] | [1-9]\\b'
```

1-24:

```
'\\b2[0-4] | 1[0-9] | [1-9]\\b'
```

1-31:

```
'\\b3[01] | [12][0-9] | [1-9]\\b'
```

1900-2099:

```
'\\b(?:19|20)[0-9]{2}\\b'
```

这里的分组仅用于在 19 和 20 之间选择，不需要捕获，所以用了 (?: 的非捕获分组格式。

36.5.4.5 判断字符型向量每个元素是否数值

如下的 R 函数用了多种数字的正则表达式来判断字符型向量每个元素是否合法数值。

```
all.numbers <- function(x){
  x <- gsub("\\A[ ]+", "", x, perl=TRUE)
  x <- gsub("[ ]+\\Z", "", x, perl=TRUE)

  pint <- '\\A[+-]?[0-9]+\\Z' # 整数，允许有前导零
  ## 浮点数 1，整数部分必须，小数部分可选，指数部分可选
  pf1 <- '\\A[+-]?[0-9]+(\\.[0-9]*)?([Ee][+-]?[0-9]+)?\\Z'
  ## 浮点数 2，整数部分省略，小数部分必须，指数部分可选
  pf2 <- '\\A[+-]?\\. [0-9]+([Ee][+-]?[0-9]+)?\\Z'
  pat <- paste(pint, pf1, pf2, sep='|')
  grepl(pat, x, perl=TRUE)
}
```

测试：

```
all.numbers(c('1', '12', '-12', '12.', '-12.',  
              '123.45', '-123.45', '.45', '-.45',  
              '1E3', '-12E-10', '1.1E3', '-1.1E-3'))
```

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

Chapter 37

随机模拟

37.1 随机数

随机模拟是统计研究的重要方法，另外许多现代统计计算方法（如 MCMC）也是基于随机模拟的。R 中提供了多种不同概率分布的随机数函数，可以批量地产生随机数。一些 R 扩展包利用了随机模拟方法，如 boot 包进行 bootstrap 估计。

所谓随机数，实际是“伪随机数”，是从一组起始值（称为种子），按照某种递推算法向前递推得到的。所以，从同一种子出发，得到的随机数序列是相同的。

为了得到可重现的结果，随机模拟应该从固定不变的种子开始模拟。用 `set.seed(k)` 指定一个编号为 k 的种子，这样每次从编号 k 种子运行相同的模拟程序可以得到相同的结果。

还可以用 `set.seed()` 加选项 `kind=` 指定后续程序要使用的随机数发生器名称，用 `normal.kind=` 指定要使用的正态分布随机数发生器名称。

R 提供了多种分布的随机数函数，如 `runif(n)` 产生 n 个标准均匀分布随机数，`rnorm(n)` 产生 n 个标准正态分布随机数。例如：

```
round(runif(5), 2)
## [1] 0.44 0.56 0.93 0.23 0.22
round(rnorm(5), 2)
## [1] -0.20 1.10 -0.02 0.16 2.02
```

注意因为没有指定种子，每次运行会得到不同的结果。

在 R 命令行运行

```
?Distributions
```

可以查看 R 中提供的不同概率分布。

37.2 sample() 函数

`sample()` 函数从一个有限集合中无放回或有放回地随机抽取，产生随机结果。

例如，为了设随机变量 X 取值于 {正面, 反面}，且 $P(X = \text{正面}) = 0.7 = 1 - P(X = \text{反面})$ ，如下程序产生 X 的 10 个随机抽样值：

```
sample(c(' 正面', ' 反面'), size=10,
       prob=c(0.7, 0.3), replace=TRUE)
## [1] " 反面" " 反面" " 反面" " 反面" " 正面"
## [6] " 正面" " 正面" " 正面" " 反面" " 反面"
```

`sample()` 的选项 `size` 指定抽样个数，`prob` 指定每个值的概率，`replace=TRUE` 说明是有放回抽样。

如果要做无放回等概率的随机抽样，可以不指定 `prob` 和 `replace` (缺省是 `FALSE`)。比如，下面的程序从 1:10 随机抽取 4 个：

```
sample(1:10, size=4)
```

```
## [1] 7 4 8 2
```

```
## [1] 1 5 8 10
```

如果要从 $1:n$ 中等概率无放回随机抽样直到每一个都被抽过，只要用如：

```
sample(10)
```

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

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

这实际上返回了 $1:10$ 的一个重排。

37.3 随机模拟示例

37.3.1 线性回归模拟

考虑如下线性回归模型

$$y = 10 + 2x + \varepsilon, \varepsilon \sim N(0, 0.5^2).$$

假设有样本量 $n = 10$ 的一组样本，R 函数 `lm()` 可以得到截距 a ，斜率 b 的估计 \hat{a}, \hat{b} ，以及相应的标准误差 $SE(\hat{a}), SE(\hat{b})$ 。样本可以模拟产生。

模型中的自变量 x 可以用随机数产生，比如，用 `sample()` 函数从 $1:10$ 中随机有放回地抽取 n 个。模型中的随机误差项 ε 可以用 `rnorm()` 产生。产生一组样本的程序如：

```
n <- 10; a <- 10; b <- 2
x <- sample(1:10, size=n, replace=TRUE)
eps <- rnorm(n, 0, 0.5)
y <- a + b * x + eps
```

如下程序计算线性回归:

```
lm(y ~ x)
```

```
##
## Call:
## lm(formula = y ~ x)
##
## Coefficients:
## (Intercept)          x
##      10.251      1.986
```

如下程序计算线性回归的多种统计量:

```
summary(lm(y ~ x))
```

```
##
## Call:
## lm(formula = y ~ x)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.13478 -0.45527  0.09335  0.54731  0.84451
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 10.25058    0.44422   23.08 1.32e-08 ***
## x           1.98595    0.06921   28.70 2.35e-09 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.6865 on 8 degrees of freedom
## Multiple R-squared:  0.9904, Adjusted R-squared:  0.9892
## F-statistic: 823.5 on 1 and 8 DF,  p-value: 2.352e-09
```

如下程序返回一个矩阵, 包括 a, b 的估计值、标准误差、 t 检验统计量、检验 p 值:

```
summary(lm(y ~ x))$coefficients
```

```
##              Estimate Std. Error  t value      Pr(>|t|)
## (Intercept) 10.250576 0.44421534 23.07569 1.320270e-08
## x           1.985948 0.06920619 28.69611 2.352431e-09
```

如下程序把上述矩阵的前两列拉直成一个向量返回：

```
c(summary(lm(y ~ x))$coefficients[,1:2])
```

```
## [1] 10.25057626 1.98594837 0.44421534 0.06920619
```

这样得到 $\hat{a}, \hat{b}, SE(\hat{a}), SE(\hat{b})$ 这四个值。

用 `replicate()` (复合语句) 执行多次模拟, 返回向量或矩阵结果, 返回矩阵时, 每列是一次模拟的结果。下面是线性回归整个模拟程序, 写成了一个函数。

```
reg.sim <- function(
  a=10, b=2, sigma=0.5,
  n=10, B=1000){
  set.seed(1)
  resm <- replicate(B, {
    x <- sample(1:10, size=n, replace=TRUE)
    eps <- rnorm(n, 0, 0.5)
    y <- a + b * x + eps
    c(summary(lm(y ~ x))$coefficients[,1:2])
  })
  resm <- t(resm)
  colnames(resm) <- c('a', 'b', 'SE.a', 'SE.b')
  cat(B, ' 次模拟的平均值:\n')
  print( apply(resm, 2, mean) )
  cat(B, ' 次模拟的标准差:\n')
  print( apply(resm, 2, sd) )
}
```

运行测试：

```
set.seed(1)
reg.sim()
```

```
## 1000 次模拟的平均值：
```

```
##           a           b          SE.a          SE.b
## 9.99624063 1.99889264 0.36333861 0.05948963
```

```
## 1000 次模拟的标准差：
```

```
##           a           b          SE.a          SE.b
```

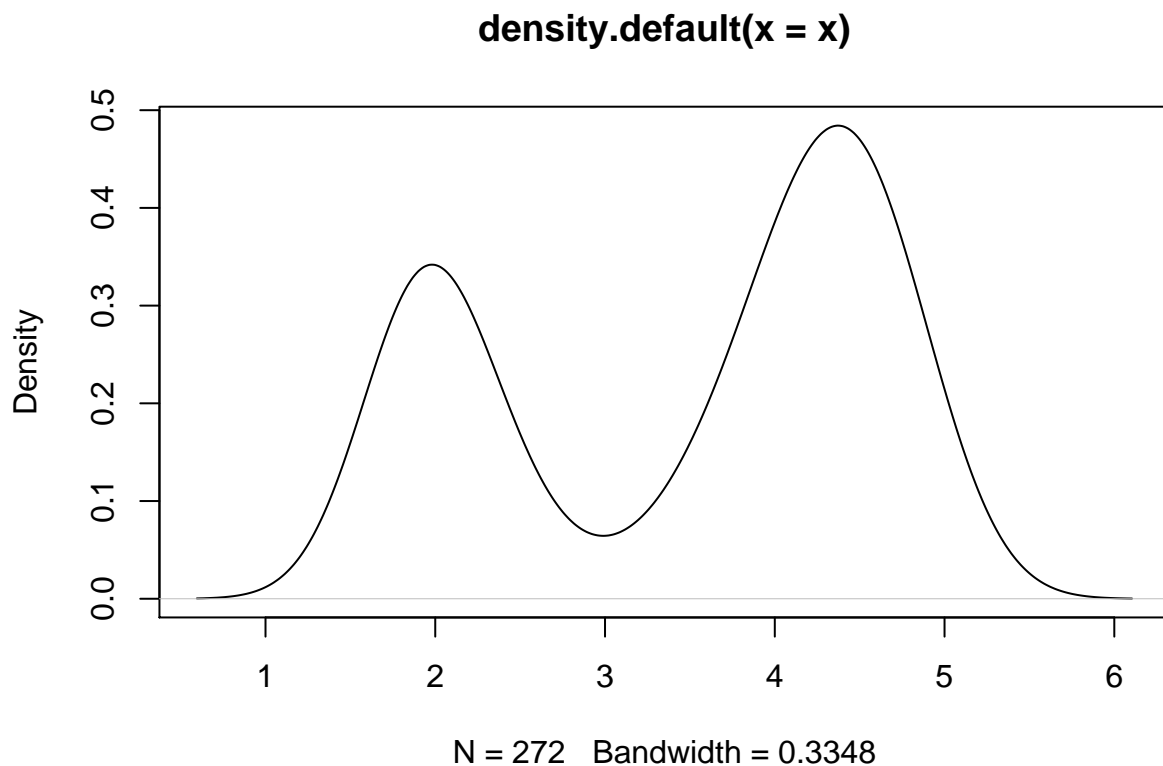
```
## 0.39281551 0.06369336 0.12802849 0.01945043
```

可以看出，标准误差作为 \hat{a}, \hat{b} 的标准差估计，与多次模拟得到多个 \hat{a}, \hat{b} 样本计算得到的标准差估计是比较接近的。结果中 $SE(\hat{a})$ 的平均值为 0.363, 1000 次模拟的 \hat{a} 的样本标准差为 0.393, 比较接近； $SE(\hat{b})$ 的平均值为 0.0594, 1000 次模拟的 \hat{b} 的样本标准差为 0.0637, 比较接近。

37.3.2 核密度的 bootstrap 置信区间

R 自带的数据框 `faithful` 内保存了美国黄石国家公园 Faithful 火山的 272 次爆发持续时间和间歇时间。为估计爆发持续时间的密度，可以用核密度估计方法，R 函数 `density` 可以执行此估计，返回 N 个格子点上的密度曲线坐标：

```
x <- faithful$eruptions
est0 <- density(x)
plot(est0)
```



这个密度估计明显呈现出双峰形态。

核密度估计是统计估计，为了得到其置信区间（给定每个 x 坐标，真实密度 $f(x)$ 的单点的置信区间），采用如下非参数 bootstrap 方法：

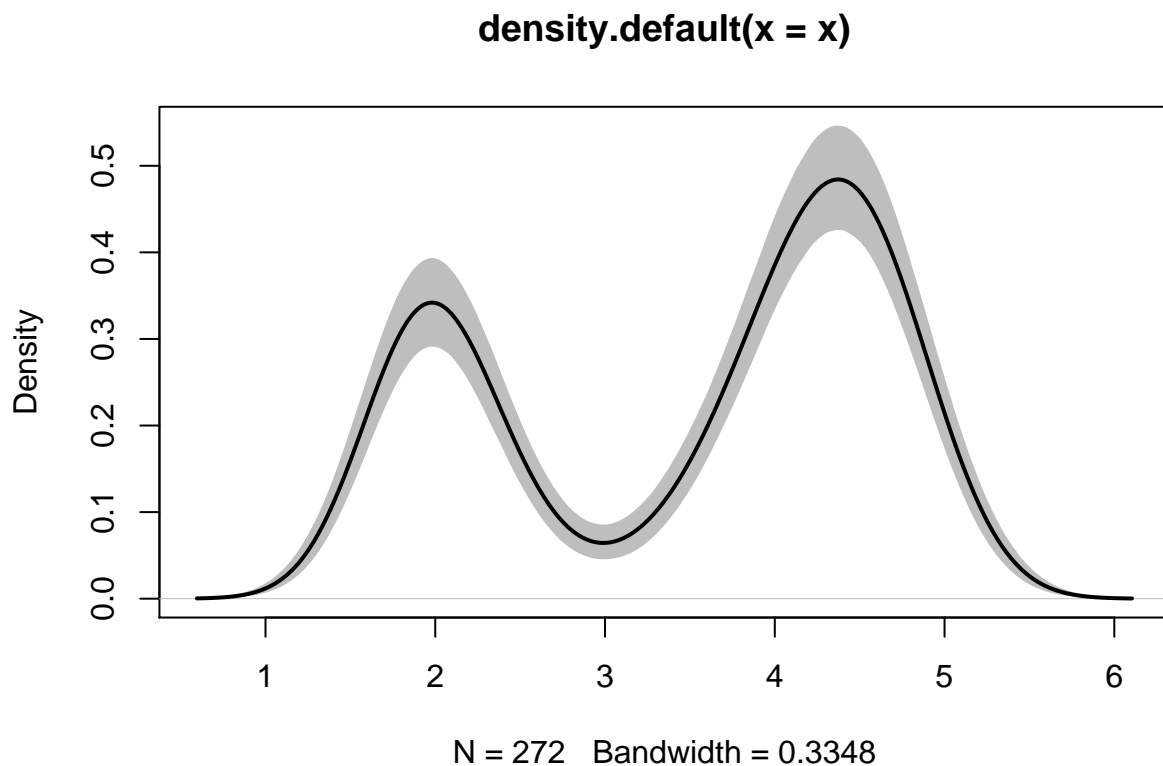
重复 $B = 10000$ 次，每次从原始样本中有重复地抽取与原来大小相同的一组样本，对这组样本计算核密度

估计, 结果为 $(x_i, y_i^{(j)}), i = 1, 2, \dots, N, j = 1, 2, \dots, B$, 每组样本估计 N 个格子点的密度曲线坐标, 横坐标不随样本改变。

对每个横坐标 x_i , 取 bootstrap 得到的 B 个 $y_i^{(j)}, j = 1, 2, \dots, B$ 的 0.025 和 0.975 样本分位数, 作为真实密度 $f(x_i)$ 的 bootstrap 置信区间。

在 R 中利用 `replicate()` 函数实现:

```
set.seed(1)
resm <- replicate(10000, {
  x1 <- sample(x, replace=TRUE)
  density(x1, from=min(est0$x),
          to=max(est0$x))$y
})
CI <- apply(resm, 1, quantile, c(0.025, 0.975))
plot(est0, ylim=range(CI), type='n')
polygon(c(est0$x, rev(est0$x)),
        c(CI[1,], rev(CI[2,])),
        col='grey', border=FALSE)
lines(est0, lwd=2)
```



程序中用 `set.seed(1)` 保证每次运行得到的结果是不变的, `replicate()` 函数第一参数是重复模拟次数,

第二参数是复合语句，这些语句是每次模拟要执行的计算。在每次模拟中，用带有 `replace=TRUE` 选项的 `sample()` 函数从样本中有放回地抽样得到一组 bootstrap 样本，每次模拟的结果是在指定格子点上计算的核密度估计的纵坐标。`replicate()` 的结果为一个矩阵，每一列是一次模拟得到的纵坐标集合。对每个横坐标格子点，用 `quantile()` 函数计算 B 个 bootstrap 样本的 2.5% 和 97.5% 分位数，循环用 `apply()` 函数表示。`polygon()` 函数指定一个多边形的顺序的顶点坐标用 `col=` 指定的颜色填充，本程序中实现了置信下限与置信上限两条曲线之间的颜色填充。`lines()` 函数绘制了与原始样本对应的核密度估计曲线。

Part X

R 编程例子

Chapter 38

R 编程例子

因为这些例子要用作学生习题，所以这里只有问题，没有实际内容。

38.1 R 语言

38.1.1 用向量作逆变换

设向量 \mathbf{x} 长度为 n ，其中保存了 1 到 n 的正整数的一个排列。把 \mathbf{x} 看成是在集合 $\{1, 2, \dots, n\}$ 上的一个一一变换，求向量 \mathbf{y} 使得 \mathbf{y} 能够表示上述变换的逆变换。即任给长度为 n 的向量 \mathbf{z} ， $\mathbf{z}[\mathbf{x}]$ 表示按照 \mathbf{x} 的次序重新排列 \mathbf{z} 的元素，而 $\mathbf{z}[\mathbf{x}][\mathbf{y}]$ 则应该恢复为 \mathbf{z} 。

38.1.2 斐波那契数列计算

设数列 $x_0 = 0, x_1 = 1$ ，后续值按如下公式递推计算：

$$x_n = x_{n-2} + x_{n-1}, \quad n = 2, 3, \dots$$

这样的数列叫做斐波那契数列。希望编写 R 函数，输入 n ，返回计算的 x_n 的值。

38.1.3 穷举所有排列

设向量 \mathbf{x} 的各个元素为某个集合的元素。要列出 \mathbf{x} 的元素的所有不同排列。比如，如果 $\mathbf{x} = 1:3$ ，所有排列为

```
1 2 3
1 3 2
2 1 3
2 3 1
```

3 1 2

3 2 1

共 $3! = 6$ 种不同的排列。

38.1.4 可重复分组方式穷举

设有 n 个编号卡片，分别有号码 $1, 2, \dots, n$ 。从中有放回地抽取 m 个并记录每次的号码，穷举 m 个号码中多少个 1，多少个 2， \dots ，多少个 n 这样的结果。

例如，有 3 个编号卡片，随机有放回地抽取 2 次。用 (x_1, x_2, x_3) 表示每一种个数组合， x_1 表示 2 次抽取中号码 1 的个数， x_2 表示 2 次抽取中号码 2 的个数， x_3 表示 2 次抽取中号码 3 的个数。问题就是列出 (x_1, x_2, x_3) 的所有不同值。

38.2 概率

38.3 智者千虑必有一失

成语说：“智者千虑，必有一失；愚者千虑，必有一得”。设智者作判断的准确率为 $p_1 = 0.99$ ，愚者作判断的准确率为 $p_2 = 0.01$ ，计算智者做 1000 次独立的判断至少犯一次错误的概率，与愚者做 1000 次独立判断至少对一次的概率。

38.3.1 圆桌夫妇座位问题

在一张圆桌上用餐时， n 对夫妇随机入座。计算没有任何一位妻子和她的丈夫相邻的概率。通过推导可得此概率为

$$p_n = 1 + \sum_{k=1}^n (-1)^k C_n^k \frac{(2n-k-1)! 2^k}{(2n-1)!}. \quad (1)$$

例如 $p_2 = 1/3$, $p_3 = 4/15 = 0.2667$, $p_4 = 0.2952$, $p_5 = 0.3101$ 。

分别用上面的理论公式以及直接穷举验证的方法，对 $n = 2, 3, 4, 5$ 的情形进行验证。

38.4 科学计算

38.4.1 城市间最短路径

假设有 n 个城市，编号为 $1, 2, \dots, n$ 。已知其中的部分城市之间有高速公路连通，每对连通城市记为 (F_i, T_i) ，其中 $F_i, T_i \in \{1, 2, \dots, n\}$ 且 $F_i < T_i$, $i = 1, 2, \dots, m$ 。除了这些直接连通的城市以外，其它的任意两个城

市只能途经别的城市连通, 或者根本不能靠高速公路连通。用一个 $m \times 2$ 的 R 矩阵 M 可以输入这些连通情况, 矩阵的每行是一对 (F_i, T_i) 值。

要求编写一个 R 函数, 输入直接连通情况 M 后, 输出一个 $n \times n$ 矩阵 R , $R[i,i]=0$, $R[i,j]=1$ 表示直接相连, $R[i,j]=k$ ($k \geq 2$) 表示城市 i 与城市 j 至少需要经过 k 段高速公路连通, $R[i,j]=\text{Inf}$ 表示城市 i 与城市 j 不能靠高速公路连通。 R 的元素值仅考虑途经的高速公路段数而不考虑具体里程。如果从一个城市通过高速公路移动到直接相连城市叫做移动一步, R 的 (i, j) 元素是从第 i 城市通过高速公路到第 j 城市需要移动的步骤数。

这个问题也可以作为“相识”问题的模型。设 n 个人中有些人是直接相识的, 如果两个不相识的人想认识, 假设必须经过相识的人引荐, 问最少需要多少个引荐人。

本问题必须使用循环, 很难向量化, 属于 R 比较不擅长的问题。可以考虑使用 Rcpp 把程序用 C++ 语言实现, 可以大大加速。当然, 如果这个程序仅需要执行不多的次数, 用 R 就足够了。

38.4.2 Daubechies 小波函数计算

这个例子主要展示了不用每次计算函数值而是尽可能从已经计算并储存的函数值中查找的技巧。程序中用了比较多的循环, 如果有需要, 可考虑用 Rcpp 转换成 C++ 代码以提高效率。

小波是重要数学工具, 在图像处理、信号处理等方面有广泛应用。小波中一个重要的函数叫做尺度函数 (scale function), 它满足所谓双尺度方程:

$$\phi(x) = \sqrt{2} \sum_k h_k \phi(2x - k)$$

一种特殊的尺度函数是只在有限区间上非零的, 叫做紧支集的。紧支集尺度函数可以在给定 $\{h_k\}$ 后用以下迭代公式生成:

$$\begin{aligned} \eta_0(x) &= I_{[-0.5, 0.5]}(x) \\ \eta_{n+1}(x) &= \sqrt{2} \sum_{k=0}^{2N-1} h_k \eta_n(2x - k) \end{aligned}$$

其中 N 是正整数, $N=2$ 时 $h_0=0.482962913145$, $h_1=0.836516303738$, $h_2=0.224143868042$, $h_3 = -0.129409522551$ 。已知 $\phi(x)$ 的支集 (不为零的区间) 为 $[0, 2N - 1]$, $\eta_n(x)$ 的支集包含于 $[-0.5, 2N - 1]$ 中。作为例子, 我们来编写计算 $\phi(x)$ 的 S 程序。在迭代过程中, 应不重新结算函数格子点的值, 仅计算新加入的格子点的值。

38.4.3 房间加热温度变化

某个房屋带有天花板, 天花板与屋顶之间有一定的空间。用壁炉保持房间温度。为了研究房间内与天花板上方的温度变化, 建立了如下的微分方程组:

$$\begin{aligned} \frac{dx_1}{dt} &= 0.35 \left(-9.7 \sin \frac{(t+3)\pi}{12} + 8.3 - x_1(t) \right) + 0.46(x_2(t) - x_1(t)) + 11.1 \\ \frac{dx_2}{dt} &= 0.28 \left(-9.7 \sin \frac{(t+3)\pi}{12} + 8.3 - x_2(t) \right) + 0.46(x_1(t) - x_2(t)) \end{aligned}$$

其中 $x_1(t)$ 是房间在 t 时刻的温度 (单位: 摄氏度), $x_2(t)$ 是天花板上在 t 时刻的温度, t 是单位为小时的时间。

设 $t = 0$ 时 $x_1(t) = x_2(t) = 4$, 用每一秒钟重新计算的方法计算 24 小时内的房间温度与天花板上温度, 绘图。计算 7 天的温度, 查看周期性。当温度循环变化差距小于 0.05 度时认为开始周期变化了。

38.5 统计计算

38.5.1 核回归与核密度估计

考虑核回归问题。核回归是非参数回归的一种, 假设变量 Y 与变量 X 之间的关系为:

$$Y = f(X) + \varepsilon$$

其中函数 f 未知。观测到 X 和 Y 的一组样本 $X_i, Y_i, i=1, \dots, n$ 后, 对 f 的一种估计为:

$$\hat{f}(x) = \frac{\sum_{i=1}^n K\left(\frac{x-X_i}{h}\right) Y_i}{\sum_{i=1}^n K\left(\frac{x-X_i}{h}\right)}$$

其中 $h > 0$ 称为窗宽, 窗宽越大, 得到的密度估计越平滑。 K 叫做核函数, 一般是一个非负的偶函数, 原点处的函数值最大, 在两侧迅速趋于零。例如正态密度函数, 或所谓双三次函数核:

$$K(x) = \begin{cases} (1 - |x|^3)^3 & |x| \leq 1 \\ 0 & \text{其它} \end{cases}$$

与核回归类似, 可以用核平滑方法估计总体分布密度。设样本为 Y_1, Y_2, \dots, Y_n , 密度估计公式为

$$\hat{f}(x) = \frac{1}{n} \sum_{i=1}^n \frac{1}{h} K\left(\frac{x - Y_i}{h}\right) Y_i$$

其中 $K(x)$ 满足 $\int_{-\infty}^{\infty} K(x) dx = 1$ 。 h 是窗宽, 窗宽越大, 估计的曲线越光滑。 h 的一种建议公式为 $h = 1.06Sn^{-1/5}$, S 为样本标准差。

对以上两个问题进行编程, 其中窗宽 h 由用户输入。

38.5.2 二维随机模拟积分

设二元函数 $f(x, y)$ 定义如下

$$\begin{aligned} f(x, y) &= \exp\{-45(x + 0.4)^2 - 60(y - 0.5)^2\} \\ &\quad + 0.5 \exp\{-90(x - 0.5)^2 - 45(y + 0.1)^4\} \end{aligned}$$

(注意其中有一个 4 次方。) 求如下二重定积分

$$I = \int_{-1}^1 \int_{-1}^1 f(x, y) dx dy$$

$f(x, y)$ 有两个分别以 $(-0.4, 0.5)$ 和 $(0.5, -0.1)$ 为中心的峰, 对积分有贡献的区域主要集中在 $(-0.4, 0.5)$ 和 $(0.5, -0.1)$ 附近, 在其他地方函数值很小, 对积分贡献很小。可以用随机模拟方法估计 I 的值。

第一种估计方法是平均值法。设 $X_i, i = 1, 2, \dots, N$ 是均匀分布 $U(-1, 1)$ 的随机数, $Y_i, i = 1, 2, \dots, N$ 也是均匀分布 $U(-1, 1)$ 的随机数, 两者独立, 可估计 I 为

$$\hat{I}_1 = \frac{4}{N} \sum_{i=1}^N f(X_i, Y_i).$$

第二种估计方法是重要抽样法。设正态分布 $N(\mu, \sigma^2)$ 的密度记为 $p(x; \mu, \sigma^2)$, 令

$$\begin{aligned} g(x, y) &= 0.5358984p(x; -0.4, 90^{-1})p(y; 0.5, 120^{-1}) \\ &\quad + 0.4641016p(x; 0.5, 180^{-1})p(y; -0.1, 20^{-1}), \\ &\quad -\infty < x < \infty, -\infty < y < \infty, \end{aligned}$$

这是一个二元随机向量的密度, 是两个二元正态密度的混合分布。设 $K_i, i = 1, 2, \dots, N$ 是取值于 $\{1, 2\}$ 的随机数, $P(K_i = 1) = 0.5358984, P(K_i = 2) = 1 - P(K_i = 1)$ 。当 $K_i = 1$ 时, 取 X_i 为 $N(-0.4, 90^{-1})$ 随机数, Y_i 为 $N(0.5, 120^{-1})$ 随机数; 当 $K_i = 2$ 时, 取 X_i 为 $N(0.5, 180^{-1})$ 随机数, Y_i 为 $N(-0.1, 20^{-1})$ 随机数, 这样得到的 $(X_i, Y_i), i = 1, 2, \dots, N$ 是 $g(x, y)$ 的随机数。令

$$\hat{I}_2 = \frac{1}{N} \sum_{i=1}^n \frac{f(X_i, Y_i)}{g(X_i, Y_i)},$$

称为 I 的重要抽样法估计。

38.5.2.1 编程任务

1. 编写 R 函数估计计算 \hat{I}_1 。重复模拟 $B = 100$ 次, 得到 \hat{I}_1 的 B 个值, 计算这些值的平均值和标准差。
2. 编写 R 函数估计计算 \hat{I}_2 。重复模拟 $B = 100$ 次, 得到 \hat{I}_2 的 B 个值, 计算这些值的平均值和标准差。
3. 比较模拟得到的平均值和标准差, 验证两者是否基本一致, 通过标准差大小比较两种方法的精度。

注意尽量用向量化编程。

38.5.3 潜周期估计

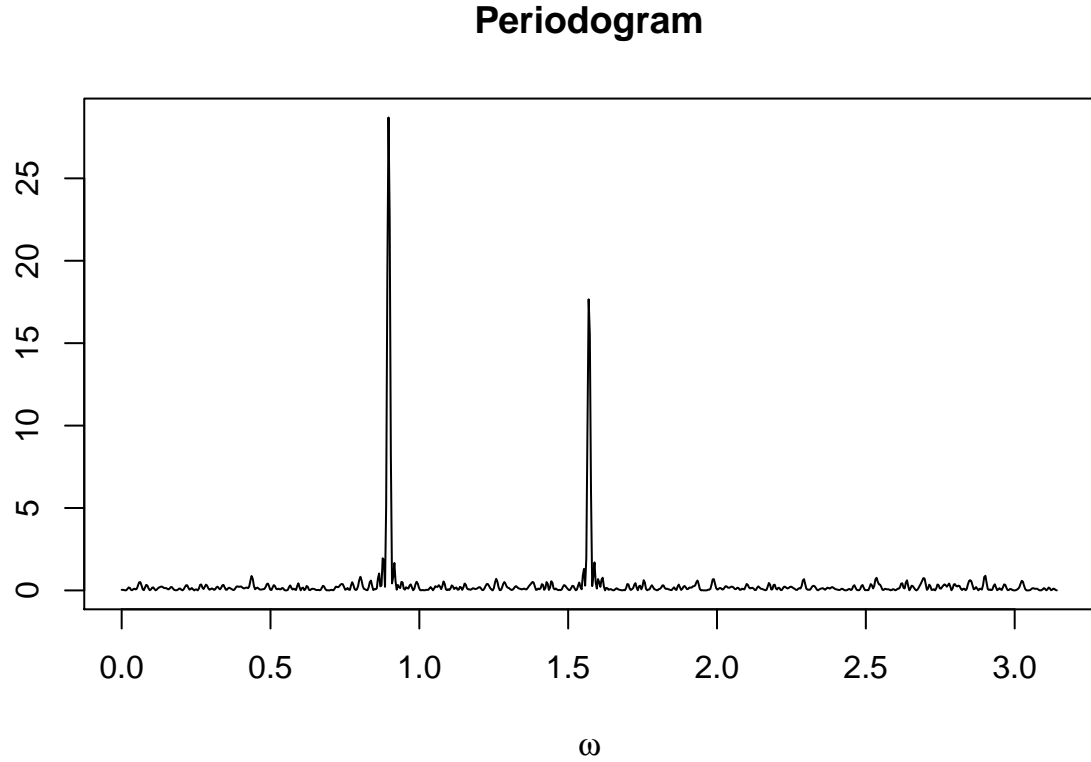
设时间序列 $\{y_t\}$ 有如下模型:

$$y_t = \sum_{k=1}^m A_k \cos(\lambda_k t + \phi_k) + x_t, \quad t = 1, 2, \dots$$

其中 x_t 为线性平稳时间序列, $\lambda_k \in (0, \pi)$, $k = 1, 2, \dots, m$ 。这样的模型称为潜周期模型。如果有 $\{y_t\}$ 的一组样本 y_1, y_2, \dots, y_n , 可以定义周期图函数

$$P(\omega) = \frac{1}{2\pi n} \left| \sum_{t=1}^n y_t e^{-it\omega} \right|^2, \quad \omega \in [0, \pi].$$

这里 ω 是角频率。对于潜周期数据, 在 λ_j 的对应位置 $P(\omega)$ 会有尖峰, 而且当 $n \rightarrow \infty$ 时尖峰高度趋于无穷。



下面是一个样例图形。

如下算法可以在 n 较大时估计 m 和 $\{\lambda_k\}$: 首先, 对 $\omega_j = \pi j/n$, $j = 1, 2, \dots, n$ 计算 $h_j = P(\omega_j)$, 求 $\{h_j, j = 1, 2, \dots, n\}$ 的 $3/4$ 分位数记为 q 。令 $C = qn^{0.25}$, 以 C 作为分界线, 设 $\{h_j\}$ 中大于 C 的下标 j 的集合为 J , 当 J 非空时, 把 J 中相邻点分入一组, 但是当两个下标的差大于等于 $n^{0.6}$ 时就把后一个点归入新的一组。在每组中, 以该组的 h_j 的最大值点对应的角频率 $j\pi/n$ 作为潜频率 $\{\lambda_k\}$ 中的一个的估计。

用如下 R 程序可以模拟生成一组 $\{y_t\}$ 的观测数据:

```
set.seed(1); n <- 500; tt <- seq(n)
m <- 2; lam <- 2*pi/c(4, 7); A <- c(1, 1.2)
y <- A[1]*cos(lam[1]*tt) + A[2]*cos(lam[2]*tt) + rnorm(n)
```

编写 R 程序:

- (1) 编写计算 $P(\omega)$ 的函数, 输入 $\mathbf{y} = (y_1, y_2, \dots, y_n)^T$ 和 $\boldsymbol{\omega} = (\omega_1, \omega_2, \dots, \omega_s)^T$, 输出 $(P(\omega_1), P(\omega_2), \dots, P(\omega_s))$ 。
- (2) 对输入的时间序列样本 y_1, y_2, \dots, y_n , 编写函数用以上描述的算法估计 m 和 $\{\lambda_j, j = 1, 2, \dots, m\}$ 。

- (3) 用上述模拟数据测试编写的算法程序。
- (4) 进一步地, 用 R 函数 `fft()` 计算 $h_j = P(\pi j/n), j = 1, 2, \dots, n$ 。
- (5) 把整个算法用 Rcpp 和 C++ 程序实现。

38.5.4 贮存可靠性评估

设某种设备的贮存寿命为随机变量 X , 服从指数分布 $\text{Exp}(\theta)$, $EX = \theta > 0$ 。假设有 n 台此种设备分别在时间 t_1, t_2, \dots, t_n 进行了试验, 第 i 台试验成功用 $\delta_i = 1$ 表示, 试验失效用 $\delta_i = 0$ 表示。把 n 次试验的结果写成

$$Z_n = \begin{pmatrix} t_1 & \delta_1 \\ t_2 & \delta_2 \\ \vdots & \vdots \\ t_n & \delta_n \end{pmatrix} \quad (1)$$

这里 $\delta_1, \delta_2, \dots, \delta_n$ 认为是随机变量, t_1, t_2, \dots, t_n 是固定的时间。 δ_i 服从两点分布 $B(1, \exp(-t_i/\theta))$ 。 Z_n 取某个特定组合的概率为

$$P(Z_n) = \prod_{i=1}^n \left\{ [\exp(-t_i/\theta)]^{\delta_i} [1 - \exp(-t_i/\theta)]^{1-\delta_i} \right\}. \quad (2)$$

要利用观测数据 Z_n 估计 θ 的置信度为 $1 - \alpha$ 的置信下限, 可以用陈家鼎《生存分析与可靠性》中的样本空间排序法。注意到 Z_n 中每个 δ_i 可以取 1 或 0, 所以 Z_n 的所有不同取值共有 2^n 个, 记这些所有不同取值为 \mathcal{D} , 在 \mathcal{D} 中定义如下的序: 设 Z'_n 与 Z_n 都是 \mathcal{D} 中的试验结果, 称 Z'_n 不次于 Z_n , 并记作 $Z'_n \succeq Z_n$, 如果如下两个条件之一成立:

- (1) $\sum_{i=1}^n \delta'_i > \sum_{i=1}^n \delta_i$;
- (2) $\sum_{i=1}^n \delta'_i = \sum_{i=1}^n \delta_i$, 但 $\sum_{i=1}^n \delta'_i t_i \geq \sum_{i=1}^n \delta_i t_i$ 。

记

$$G(\theta) = \sum_{Z'_n \succeq Z_n} P_\theta(Z'_n) = \sum_{Z'_n \succeq Z_n} \prod_{i=1}^n \left\{ [\exp(-t_i/\theta)]^{\delta'_i} [1 - \exp(-t_i/\theta)]^{1-\delta'_i} \right\}. \quad (3)$$

这是 θ 的严格单调增函数, 求解如下的方程

$$G(\theta) - \alpha = 0 \quad (4)$$

得到解 $\underline{\theta}$ 是 θ 的置信度为 $1 - \alpha$ 的置信下限。

当所有 n 次试验都失效时, 恒有 $G(\theta) = 1$, 方程 (4) 无解, 取 $\underline{\theta} = 0$ 。

当 n 次试验都没有失效, 即失效数 $n - \sum_{i=1}^n \delta_i = 0$ 时, 方程为

$$\exp\left(-\frac{1}{\theta} \sum_{i=1}^n t_i\right) - \alpha = 0,$$

解得

$$\underline{\theta} = \frac{\sum_{i=1}^n t_i}{\ln \frac{1}{\alpha}}.$$

当失效数为 1 时, 最多仅有 n 个结果不次于 Z_n , 很容易可以计算 $G(\theta)$ 的值。在 n 较大而且 Z_n 中失效数较多时, $G(\theta)$ 的求和 (3) 中项数很多, 计算量很大。当 $n = 10$ 时, \mathcal{D} 约有一千项, 当 $n = 20$ 时, \mathcal{D} 约有一百万项, 还在可以穷举计算的范围内。但是, 当 $n \geq 30$ 时, \mathcal{D} 就有 10 亿项, 每计算一次 $G(\theta)$ 都需要很长时间。

针对 n 不太大的情形, 可以用精确的公式 (3) 计算 $G(\theta)$ 并用 (4) 求解 $\underline{\theta}$, 编写这个问题的纯 R 程序版本, 输入一组 Z_n 值后 (包括试验时间和试验结果), 输出用 (3) 和 (4) 求解 $\underline{\theta}$ 得到的置信下限 $\underline{\theta}$ 的值。在文件 “store-reliab-data.csv” (内容见本文附录) 中已经生成了 10 组模拟数据, 对这 10 组模拟数据计算相应的 $\underline{\theta}$ 的值。在此数据文件中, testid 相同的行属于同一次试验的不同设备的时间和结果。

这个程序中用到比较多的循环, 考虑把主要计算部分用 Rcpp 包和 C++ 代码实现, 看能够提高效率多少倍。

另一种计算 $G(\theta)$ 的方法是用随机模拟方法估计 $G(\theta)$ 的值然后求解 (4)。随机模拟方法计算简单而且不受失效个数多少的影响, 但是有随机误差, 在求解 (4) 要考虑到随机误差的影响。随机模拟方法如下。取模拟次数 N , 对给定 θ , 为了计算 $G(\theta)$, 模拟生成 N 组独立的寿命 $(X_1^{(i)}, X_2^{(i)}, \dots, X_n^{(i)})$, $i = 1, 2, \dots, N$, 其中每个 $X_j^{(i)}$ 服从 $\text{Exp}(\theta)$ 分布, 各分量相互独立。计算 $\delta_j^{(i)} = I_{\{X_j^{(i)} > t_j\}}$, 记

$$Z_n^{(i)} = \begin{pmatrix} t_1 & \delta_1^{(i)} \\ t_2 & \delta_2^{(i)} \\ \vdots & \vdots \\ t_n & \delta_n^{(i)} \end{pmatrix}$$

用

$$\frac{1}{N} \sum_{i=1}^N I_{\{Z_n^{(i)} \gtrsim Z_n\}}$$

来估计 $G(\theta)$ 。

38.6 数据处理

38.6.1 小题分题型分数汇总

考虑中学某科的一次考试, 考卷各小题情况汇总在如下的用逗号分隔的文本文件 subscore-subtype.csv 中:

序号, 题型

1, 选择题

2, 选择题

3, 选择题

4, 选择题

5, 选择题

6, 选择题

- 7, 选择题
- 8, 选择题
- 9, 选择题
- 10, 选择题
- 11, 简答题
- 12, 简答题
- 13, 填空题
- 14, 简答题
- 15, 简答题
- 16, 简答题
- 17, 简答题
- 18, 写作

读入此数据为 R 数据框，只要用如下程序：

```
dm <- read.csv('subscore-subtype.csv', header=TRUE,  
               stringsAsFactors=FALSE)
```

结果显示如下：

```
knitr::kable(dm)
```

| 序号 | 题型 |
|----|-----|
| 1 | 选择题 |
| 2 | 选择题 |
| 3 | 选择题 |
| 4 | 选择题 |
| 5 | 选择题 |
| 6 | 选择题 |
| 7 | 选择题 |
| 8 | 选择题 |
| 9 | 选择题 |
| 10 | 选择题 |
| 11 | 简答题 |
| 12 | 简答题 |
| 13 | 填空题 |
| 14 | 简答题 |
| 15 | 简答题 |
| 16 | 简答题 |
| 17 | 简答题 |
| 18 | 写作 |

设部分学生的小题分录入到如下的用逗号分隔的文本文件 `subscore-subscore.csv` 中：

学号,Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8,Y9,Y10,Y11,Y12,Y13,Y14,Y15,Y16,Y17,Y18

1138010104,3,3,3,3,0,3,7.5,2.5,3.5,6,5,4,2.5,5.5,0,4,2,45.5

1138010108,3,0,3,0,3,0,6,3,4,4,2,5,2,5,6,4,2,48.5

1138010114,3,3,3,3,0,3,5.5,3.5,4,6,2,5,2,5.5,6,1,2,44.5

1138010128,3,3,3,0,0,0,8.5,3.5,2.5,4,5,5,0,4.5,6,4,2,45

1138010126,3,3,3,3,3,3,7,0,4,6,5,5,2.5,4.5,6,4.5,2,44

用如下 R 程序读入小题分数数据为 R 数据框:

```
ds <- read.csv('subscore-subscore.csv', header=TRUE,
               stringsAsFactors=FALSE)
```

结果显示如下:

```
knitr::kable(ds)
```

| 学号 | Y1 | Y2 | Y3 | Y4 | Y5 | Y6 | Y7 | Y8 | Y9 | Y10 | Y11 | Y12 | Y13 | Y14 | Y15 | Y16 | Y17 | Y18 |
|------------|----|----|----|----|----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1138010104 | 3 | 3 | 3 | 3 | 0 | 3 | 7.5 | 2.5 | 3.5 | 6 | 5 | 4 | 2.5 | 5.5 | 0 | 4.0 | 2 | |
| 1138010108 | 3 | 0 | 3 | 0 | 3 | 0 | 6.0 | 3.0 | 4.0 | 4 | 2 | 5 | 2.0 | 5.0 | 6 | 4.0 | 2 | |
| 1138010114 | 3 | 3 | 3 | 3 | 0 | 3 | 5.5 | 3.5 | 4.0 | 6 | 2 | 5 | 2.0 | 5.5 | 6 | 1.0 | 2 | |
| 1138010128 | 3 | 3 | 3 | 0 | 0 | 0 | 8.5 | 3.5 | 2.5 | 4 | 5 | 5 | 0.0 | 4.5 | 6 | 4.0 | 2 | |
| 1138010126 | 3 | 3 | 3 | 3 | 3 | 3 | 7.0 | 0.0 | 4.0 | 6 | 5 | 5 | 2.5 | 4.5 | 6 | 4.5 | 2 | |

在数据框 dm 中有每个小题的题型信息, 在数据框 ds 中有每个学生的每个小题的分数。从这两个数据框, 汇总计算每个学生的题型分, 即每个学生选择题共考多少分, 简答题共考多少分, 等等。

要注意的是, 最终的程序应该写成一个函数, 其中的计算不依赖于具体的题型名称、小题个数、小题与题型如何对应, 只要输入 dm 和 ds 两个数据框就可以进行统计。

如果没有这样的通用性要求, 这个问题就可以这样简单解决:

```
resm <- data.frame(
  '学号'=ds[, '学号'],
  '选择题'=rowSums(ds[, paste('Y', 1:10, sep='')]),
  '简答题'=rowSums(ds[, paste('Y', c(11,12,14:17), sep='')]),
  '填空题'=ds[, 'Y13'],
  '作文'=ds[, 'Y18']
)
knitr::kable(resm[order(resm[, '学号']),], row.names=FALSE)
```

| 学号 | 选择题 | 简答题 | 填空题 | 作文 |
|------------|------|------|-----|------|
| 1138010104 | 34.5 | 20.5 | 2.5 | 45.5 |
| 1138010108 | 26.0 | 24.0 | 2.0 | 48.5 |
| 1138010114 | 34.0 | 21.5 | 2.0 | 44.5 |
| 1138010126 | 35.0 | 27.0 | 2.5 | 44.0 |
| 1138010128 | 27.5 | 26.5 | 0.0 | 45.0 |

38.7 文本处理

38.7.1 用 R 语言下载处理《红楼梦》htm 文件

网上许多资源是 html 格式的文本文件。比如，《红楼梦》在许多网站可以浏览，是在浏览器中按章节浏览。我们希望将其下载到本地，并转换为 txt 格式，在手机或者电纸书阅读器中阅读。

这个任务涉及到 R 的文件访问，字符型连接，正则表达式，中文编码问题。

设下载网站主页是 <http://www.xiexingcun.net/honglouloumeng/index.html>，各个章节的文件名是 01.htm 到 99.htm，100.htm 到 120.htm。

已下载的文件在如下链接中：[.zip](#)

将这些文件转换成 txt 格式，然后合并成一个 txt 文件。

要求：每一段仅用一个换行；不同段落之间用换行分开；每一回目前后空行，格式为“第 xx 回 ‘标题’”。

参考文献

- Becker, R. A. and Chambers, J. M. (1984). *S: An Interactive Environment for Data Analysis and Graphics*. Wadsworth Advanced Books Program, Belmont CA.
- Becker, R. A., Chambers, J. M., and Wilks, A. R. (1988). *The New S Language*. Chapman and Hall, New York.
- Chambers, J. M. (2008). *Software for Data Analysis: Programming with R*. Springer.
- Chambers, J. M. and Hastie, T. (1992). *Statistical Models in S*. Chapman and Hall, New York.
- Venables, W. N. and Ripley, B. D. (2002). *Modern Applied Statistics with S*. Springer, 4ed edition.