R 编程

目录

1 R 输入输出量

1.1 输入输出的简单方法

1.1.1 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")
cat("y =", y, "\n")
```

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。虽然允许保存多个变量到同一文件中，但尽可能仅保存一个变量，而且使用变量名作为文件名。用 save() 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。如

```
save(scores, file="scores.RData") load("scores.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

但是，在 Microsoft 的中文版 Windows 操作系统中，默认编码是 GB 编码，用 write_csv() 生成的 CSV 文件总是使用 UTF-8 编码，系统中的 MS Office 软件不能自动识别这样编码的 CSV 文件。write.csv() 函数不存在这个问题。

1.1.2 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 ncol 选项，可以忽略 nrow 选项。如

```
M <- matrix(scan('mat.txt', quiet=TRUE), ncol=3, 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。

1.2 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	$Master'sdegree$
3	3	$Bond, James$
4	4	$Aspecialgift$

read_csv() 还可以从字符串读入一个数据框，如

```
library(tidyverse)
```

```
d.small <- read_csv("name,x,y
John, 33, 95
Kim, 21, 64
Sandy, 49, 100
")
d.small
```

```
## # A tibble: 3 x 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 x 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"))
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 lelvels= 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

最简单地用 read_csv() 读入上述 CSV 文件，程序如:

```
d.class <- read_csv('class.csv')
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_excel_csv() 函数将 tibble 保存为 csv 文件，总是使用 UTF-8 编码，结果可以被 MS Office 读取。

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。

1.3 Excel 表访问

1.3.1 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, file="tmp1.csv", row.names=FALSE)
```

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

1.3.2 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)
}
```

1.3.3 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: 要读入哪一个工作簿 (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。

1.3.4 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.exsits() 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:

1	101	85	90
2	103	60	78
3	104	73	80

1.3.5 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。

1.4 使用专用接口访问数据库

1.4.1 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;“%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端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。

1.4.2 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)
```

1.5 文件访问

1.5.1 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。

1.5.2 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，不包含文件中用来分开各行的换行标志。可以指定要读的行数。如

```
ll <- readLines(file('class.csv'))
print(ll)
```

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

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。

1.5.3 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 Im-port/Export Manual”。

1.5.4 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)
```

```
##   name score
## 1 王芳    78
## 2 孙莉    85
## 3 张聪    80
```

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)
```

```
## [1] "Trial of text connection." "1 2 3 4 5 6 7 8 9 10 "
```

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。

1.6 中文编码问题

读写文本格式的数据，或者用 readLines()、readr::read_lines() 读写文本文件，可能会遇到中文编码不匹配的问题。这里总结一些常用解决方法，所用的操作系统为中文 Windows10, 在 RStudio 中运行，R 版本为 3.4.3。常见的中文编码有 GBK(或 GB18030, GB)，UTF-8，UTF-8 有 BOM 标志等。

可以用 iconvlist() 查看 R 支持的编码名称。

假设有如下的含有中文的文件：

序号, 收缩压

1,145

5,110

6, 未测

9,150

10, 拒绝

15,115

这个文件是在中文版 MS Office 的 Excel 软件中输入后，用 Office 的 “文件— —另存为——.csv 格式” 生成的，结果的编码是 GBK 编码，或 GB18030 编码。文件下载：bp.csv

我们用工具软件将其转换成 UTF-8 无 BOM 格式，下载链接：bp-utf8nobom. csv 转为 UTF-8 有 BOM 格式，下载链接：bp-utf8bom.csv

1.6.1 用基本 R 的读取函数读取

与所用操作系统默认编码相同的文本文件，R 基本软件的 read.csv()、read.table()、readLines() 函数都可以正常读取，所以 bp.csv 文件可以正常读取，如

```
read.csv("bp.csv")
```

```
##   序号 收缩压
## 1    1    145
## 2    5    110
## 3    6   未测
## 4    9    150
## 5   10   拒绝
## 6   15    115
```

```
readLines("bp.csv")
```

```
## [1] "序号,收缩压" "1,145"       "5,110"       "6, 未测"     "9,150"
## [6] "10, 拒绝"    "15,115"
```

但是另外两个以 UTF-8 编码的文件则不能正确读入：

```
read.csv("bp-utf8nobom.csv")
## Error in make.names(col.names, unique = TRUE) : invalid multibyte string 2
```

```
readLines("bp-utf8bom.csv")
```

```
## [1] "锘垮簭鍙\xb7,鏀剁缉鍘\x8b" "1,145"
## [3] "5,110"                      "6, 鏈 祧"
## [5] "9,150"                      "10, 鎷掔粷"
```

```
## [7] "15,115"
```

读取 UTF-8 编码无 BOM 的文件时，在 read.csv() 和 read.table() 等函数中加 fileEncoding="UTF-8" 选项可以纠正编码问题：

```
read.csv("bp-utf8nobom.csv", fileEncoding="UTF-8")
```

```
##   序号 收缩压
## 1    1    145
## 2    5    110
## 3    6   未测
## 4    9    150
## 5   10   拒绝
## 6   15    115
```

读取 UTF-8 编码无 BOM 或者有 BOM 的文件时，在 readLines() 函数中加 encoding="UTF-8" 选项可以纠正编码问题：

```
readLines("bp-utf8nobom.csv", encoding="UTF-8")
```

```
## [1] "序号,收缩压" "1,145"       "5,110"       "6, 未测"     "9,150"
## [6] "10, 拒绝"    "15,115"
```

```
readLines("bp-utf8bom.csv", encoding="UTF-8")
```

```
## [1] "<U+FEFF>序号,收缩压" "1,145"       "5,110"       "6, 未测"     "9,150"
## [6] "10, 拒绝"    "15,115"
```

但是，UTF-8 有 BOM 标志的文本文件不能被 read.csv() 识别：

```
read.csv("bp-utf8bom.csv", fileEncoding="UTF-8")
## invalid input found on input connection 'bp-utf8bom.csv'
## incomplete final line found by readTableHeader on 'bp-utf8bom.csv'
```

1.6.2 用 readr 包读取

readr 包的 read_csv()、read_table2()、read_lines() 函数默认从 UTF-8 编码的文件中读取，无 BOM 或者有 BOM 都可以。如：

```
read_csv("bp-utf8nobom.csv")
```

```
## # 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
```

```
read_csv("bp-utf8bom.csv")
```

```
## # 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
```

```
read_lines("bp-utf8nobom.csv")
```

```
## [1] "序号,收缩压" "1,145"       "5,110"       "6, 未测"     "9,150"
## [6] "10, 拒绝"    "15,115"
```

```
read_lines("bp-utf8bom.csv")
```

```
## [1] "序号,收缩压" "1,145"       "5,110"       "6, 未测"     "9,150"
## [6] "10, 拒绝"    "15,115"
```

但是，对 GBK 编码的文件，不能直接读取：

```
read_csv("bp.csv")
read_lines("bp.csv")
```

为了读取 GBK(或 GB18030) 编码的文件,在 read_csv() 和 read_lines() 函数中加入 locale=locale(encoding="GBK") 选项：

```
read_csv("bp.csv", locale=locale(encoding="GBK"))
```

```
## # 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
```

```
read_lines("bp.csv", locale=locale(encoding="GBK"))
```

```
## [1] "序号,收缩压" "1,145"       "5,110"       "6, 未测"     "9,150"
## [6] "10, 拒绝"    "15,115"
```

1.6.3 输出文件的编码

write.csv()、writeLines() 生成的含有中文的文件的编码默认为操作系统的默认中文编码，这里是 GB18030。

readr 的 write_csv()、write_lines() 函数生成的含有中文的文件的编码默认 UTF-8 无 BOM。如

```
write_csv(tibble(" 姓名"=c(" 张三", " 李四")), "tmp.csv")
```

结果生成的文件编码为 UTF-8 无 BOM，这样的文件可以被 R 的 readr::read_csv() 正确读取，但是不能被 MS Excel 软件正确读取。

write_lines() 输出的文件也是编码为 UTF-8 无 BOM。

write_excel_csv() 可以生成带有 UTF-8 有 BOM 的 CSV 文件，这样的文件可以被 MS Office 正确识别：

```
write_excel_csv(tibble(" 姓名"=c(" 张三", " 李四")), "tmp2.csv")
```

1.6.4 分批读写

readLines()、readr::read_lines()、writeLines()、readr::writeLines() 支持分批读写。这需要预先打开要读取和写入的文件，所有内容都处理一遍以后关闭读取和写入的文件。

使用 file() 函数打开文件用于读写，使用 close() 函数关闭打开的文件。打开文件时可以用 encoding= 指定编码，但是 readr::read_lines() 不支持分批读入。

下面的程序每次从 UTF-8 无 BOM 编码的 bp-utf8nobom.csv 读入两行，不加处理第写入 tmp.csv 中，使用 readLines() 和 writeLines()，读入时用 encoding="UTF-8" 指定编码，写出时不指定编码，结果是操作系统默认的 GBK:

```
fin <- file("bp-utf8nobom.csv", "rt", encoding="UTF-8") fout <- file("tmp.csv", "wt")
repeat{
  lines <- readLines(fin, n=2) print(lines)
if(length(lines)==0) break
  writeLines(lines, fout)
}
close(fout)
close(fin)
## [1] " 序号, 收缩压" "1,145"
## [1] "5,110" "6, 未测"
```

```
## [1] "9,150" "10, 拒绝"
## [1] "15,115"
## character(0)
```

file() 中的 encoding="UTF-8" 特指 UTF-8 无 BOM 的格式，有 BOM 的 UTF-8 编码文件无法用上述方法打开。

上面的例子生成的结果 tmp.csv 使用了中文 Windows 系统的默认编码 GBK 编码。为了生成 UTF-8 无 BOM 的结果，可以在上述程序中打开输出文件时加选项 encoding="UTF-8"。即

```
fin <- file("bp-utf8nobom.csv", "rt", encoding="UTF-8")
fout <- file("tmp.csv", "wt", encoding="UTF-8")
...
```

readr::read_lines() 不支持从一个文件分批读入。readr::write_lines() 可以用 append=TRUE 选项向一个文件分批写出。

1.7 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(filenames)—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。

2 程序控制结构

2.1 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
}
```

2.2 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, 表达式可以是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
}
```

多个分支，可以在中间增加 else if，如：

```
x <- c(0.05, 0.6, 0.3, 0.9)
for(i in seq(along=x)){
  if(x[i] <= 0.2){
  cat("Small\n")
  } else if(x[i] <= 0.8){
  cat("Medium\n")
  } else {
```

```
    cat("Large\n")
  }
}
```

2.2.1 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。

2.3 循环结构

2.3.1 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] 1 0 0 0 1
```

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 <- 0
n <- 5
for(i in 1:n){
  x <- 2*x + 1
  s <- s + x
}
print(s)
```

```
## [1] 57
```

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，前面那个示性函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] 0 0 0 0 1
```

2.3.2 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 ，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。

2.4 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。

2.5 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)。

3 函数

3.1 函数基础

3.1.1 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。

3.1.2 函数定义

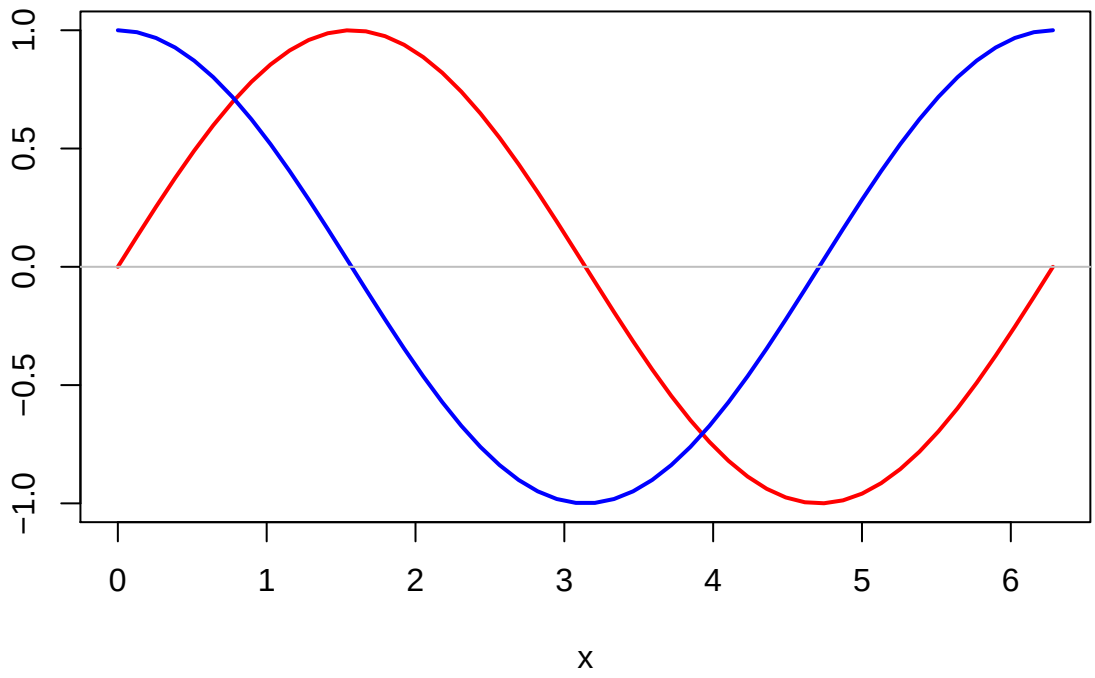
函数定义使用 function 关键字，一般格式为

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 x1, x2, . . . , xn 是一个总体的简单随机样本，其样本偏度统计量定义如下：

$$\hat{w} = \frac{n}{(n-1)(n-2)} \sum_{i=1}^{n} (\frac{x_i - \bar{x}}{S})^3$$

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 $((x - \bar{x})^3)$ 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，函数名只不过是被赋值为某个函数对象，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。

3.1.3 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
```

```
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
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
```

```
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
```

```
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。

3.2 变量作用域

3.2.1 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。

3.2.2 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 program-ming) 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。#### 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, xl, 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, xl 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。

3.2.2.1 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
```

3.2.2.2 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
```

这个问题可以用 R 的 closure 来解决。

3.2.3 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。

3.3 函数进阶

3.3.1 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 <- 100
  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。

3.3.1.1 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
```

3.3.1.2 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")
summary.df.numeric(d.class, min, na.rm=TRUE)
```

```
##    age height weight
##   11.0   51.3   50.5
```

3.3.1.3 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()

  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。

3.3.2 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。

3.3.3 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
```

3.3.4 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：

$$g(x) = \begin{cases} x^2, & |x| \le 1, \\ 1, & |x| > 1 \end{cases}$$

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
```

3.3.5 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。

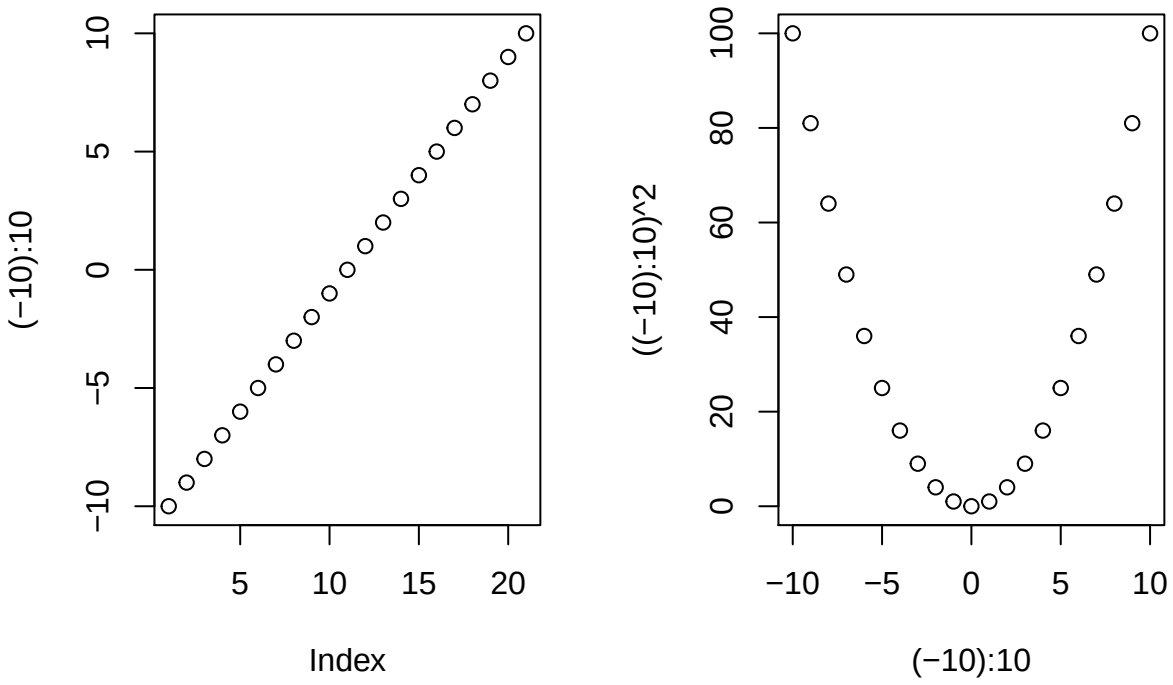
如果函数对相同的输入可以有不同的输出当然不是纯函数，例如 R 中的随机数函数 (sample(), runif(), rnorm 等)。

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要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。

3.4 程序调试

3.4.1 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 命令强项终止程序运行。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)
## debug在<text>#3: for (i in 1:n) {
##     s <- s + x[i]
## }
## debug在<text>#4: s <- s + x[i]
## debug在<text>#4: s <- s + x[i]
## debug在<text>#4: s <- s + x[i]
## debug在<text>#4: s <- s + x[i]
## debug在<text>#4: s <- s + x[i]
```

```
## 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))
## debug在<text>#3: n <- length(x)
## debug在<text>#4: s <- 0
## debug在<text>#5: for (i in 1:n) {
##     s <- s + x[i]
## }
## debug在<text>#6: s <- s + x[i]
## debug在<text>#6: s <- s + x[i]
## debug在<text>#8: s
```

```
## 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。

3.4.2 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。

3.4.3 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)
```

3.4.4 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。

3.4.5 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...
}
```

3.5 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 Haskel, Lisp 的运行与 R 的显式的、顺序的执行方式相差很大。

3.5.1 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。

3.5.2 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。

3.5.2.1 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
```

3.5.2.2 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] 5 1 5 1 4
##
## [[2]]
## [1] 5 1 2 3 1
##
## [[3]]
## [1] 3 2 3 1 1
##
## [[4]]
## [1] 4 3 1 5 3
```

```
intersect(intersect(intersect(x[[1]], x[[2]]), x[[3]]), x[[4]])
```

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

```
library(magrittr)
x[[1]] %>% intersect(x[[2]]) %>% intersect(x[[3]]) %>% intersect(x[[4]])
```

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

还可以写成循环:

```
y <- x[[1]]
for(i in 2:4) y <- intersect(y, x[[i]])
y
```

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

都比较繁琐。利用 Reduce 函数，只要写成

```
Reduce(intersect, x)
```

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

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。

3.5.2.3 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。

4 R 程序效率

4.1 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。

4.2 向量化编程

4.2.1 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 $x_1, x_2, \ldots, x_n$ 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 cld
##  f1(x) 1163.824 1285.141 1516.3378 1341.6875 1556.3055 5254.171   100   b
##  f2(x)  248.289  283.502  343.4925  325.6545  356.4985 2031.550   100  a
```

就平均运行时间（单位：毫秒）来看，f2() 比 f1() 快大约 30 倍。

4.2.2 示例 2

假设要编写函数计算

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

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)
```

4.2.3 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：

$$
\begin{aligned}
&P(\text{至少有两人同生日}) \\
=&1 - P(n\text{个人生日彼此不同}) \\
=&1 - \frac{365 \times 364 \times \ldots \times (365-(n-1))}{356^n} \\
=&1 - \frac{365-0}{365}\frac{365-1}{365}\cdots\frac{365-(n-1)}{365}
\end{aligned}
$$

对 n = 1, 2, . . . , 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 × 364 × ••• × (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 cld
##  f1() 2416.579 2482.8925 2694.61669 2556.402 2638.394 9246.846   100   c
##  f2()  319.229  328.4825  519.86528  343.390  369.606 6462.202   100  b
##  f3()    1.028    1.5420   24.18124    1.543    2.057 2196.049   100 a
```

f2() 比 f1() 快大约 7 倍，f3() 比 f2() 又快了大约 160 倍，f3() 比 f1() 快了一千倍以上！

4.3 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。

4.3.1 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。

4.3.1.1 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 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 ...
```

4.3.1.2 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 strspli() 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。

如果需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
```

使用 apply 类函数的程序写法简洁，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。

上例中的嵌套调用用 magrittr 包的管道运算符更容易理解：

```
library(magrittr)
s %>%
  strsplit(",", fixed=TRUE) %>%
  sapply(as.numeric) %>%
  colSums()
```

```
## [1] 25  9 18 25
```

4.3.1.3 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。

4.3.1.4 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。

但是，如果 sapply() 函数中的无名函数访问其它变量的话，容易产生作用域问题。

4.3.2 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。

4.3.3 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
```

apply() 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
```

4.4 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，查看其中与算术、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。

4.4.1 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。

4.4.2 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。

4.4.3 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。

4.4.4 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。

4.4.5 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。

4.4.6 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

```r
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。

4.4.7 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=fft(x)，则

```r
y[k] = sum(x * complex(
  argument = -2*pi * (0:(n-1)) * (k-1)/n))
```

即

$$y_{k+1} = \sum_{j=0}^{n-1} x_{j+1} \exp(-i2\pi \frac{kj}{n}), k = 0, 1, \ldots, n-1$$

注意没有除以 n，结果是复数向量。

另外，若 y=fft(x), z=fft(y, inverse=T), 则 x == z/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。

4.4.8 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。

4.4.8.1 示例 1：双侧滤波

对输入序列 xt, t = 1, 2, . . . , n，希望进行如下滤波计算:

$$y_t = \sum_{j=-k}^{k} a_j x_{t-j}, k+1 \le t \le n-k-1$$

其中 $(a_{-k}, ..., a_0, ..., a_k)$ 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，$(a_{-k}, ..., a_0, ..., a_k)$ 保存在向量 f 中，$y_{k+1}, ..., y_{n-k}$ 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

$y_t = 0.1 \text{»} x_{t+1} + 0.5 \text{»} x_t + 0.4 \text{»} x_{t-1}, t = 2, 3, ..., 5$

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
```

4.4.8.2 示例 2: 单侧滤波

对输入序列 $x_t, t = 1, 2, ..., n$，希望进行如下滤波计算:

$$y_t = \sum_{j=0}^{k} a_j x_{t-j}, k+1 \le t \le n$$

其中 $(a_0, ..., a_k)$ 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，$(a_0, ..., a_k)$ 保存在向量 f 中，$y_{k+1}, ..., y_n$ 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

$$y_t = 0.1 \times x_t + 0.5 \times x_{t-1} + 0.4 \times x_{t-2}, t = 3, 4, \ldots, 6$$

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
```

4.4.8.3 示例 3: 自回归迭代

设输入 $e_t, t = 1, 2, ..., n$，要计算

$$y_t = \sum_{j=1}^{k} a_j y_{t-j} + e_t, t = 1, 2, \ldots, n$$

其中 $(a_1, ..., a_k)$ 是 k 个实数，$(y_{-k+1}, ..., y_0)$ 已知。

设 x 保存在向量 x 中，$(a_1, ..., a_k)$ 保存在向量 a 中，$(y_1, ..., y_n)$ 保存在向量 y 中。

如果 $(y_{-k+1}, \ldots, y_0)$ 都等于零，可以用如下程序计算 $y_1, y_2, ..., y_n$:

```
filter(x, a, method="recursive")
```

如果 $(y_0, \ldots, y_{-k+1})$ 保存在向量 b 中 (注意与时间顺序相反)，可以用如下程序计算 $y_1, y_2, \ldots, y_n$:

```
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

$$y_t = 0.9 \times y_{t-1} + 0.1 \times y_{t-2} + e_t, t = 1, 2, \ldots, 6$$

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
```

4.5 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。

4.5.1 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
```

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

用 makeCluster() 建立临时的有 8 个节点的单机集群：

```
nNodes <- 8
cpucl <- makeCluster(nNodes)
```

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(cpucl, 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(cpucl)
```

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：

```
cpucl <- makeCluster(nNodes)
clusterExport(cpucl, c("f10"))
f13 <- function(n=1000000){
  nk <- 20
  v <- parSapply(cpucl, 2:(nk+1), function(k) f10(k, n))
  v
}

system.time(f13())[3]
## elapsed
## 1.08 stopCluster(cpucl)
```

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(cpucl, library(dplyr))
```

4.5.2 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 为标准正态分布的双侧 α 分位数，参数 p 的近似 1 - α 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 - α, 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
}
ystem.time(f21())[3]
## elapsed
## 4.3
```

约运行 4.3 秒。

改为并行版本：

```
library(parallel)
nNodes <- 8
cpucl <- makeCluster(nNodes)
clusterExport(cpucl, 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(cpucl, lp, f20)
  res
}
system.time(f22())[3]
## elapsed
## 1.25 stopCluster(cpucl)
```

运行约 1.25 秒，速度提高 240% 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。

4.5.3 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 - α = 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(nsim=1E7){
  set.seed(101)
  n <- 30; p0 <- 0.01; conf <- 0.95
  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
## 42.61
```

单线程版本运行了大约 43 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。

```
library(parallel)
nNodes <- 8
cpucl <- makeCluster(nNodes)
each.seed <- function(s){
assign(".Random.seed", s, envir = .GlobalEnv) }
RNGkind("L'Ecuyer-CMRG")
set.seed(101)
seed0 <- .Random.seed
seeds <- as.list(1:nNodes)
for(i in 1:nNodes){ # 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(cpucl, seeds, each.seed)
f32 <- function(isim, nsimsub=10000){
  n <- 30; p0 <- 0.01; conf <- 0.95
  cover <- 0
  for(i in seq(nsimsub)){
    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(nsim=1E7){
  nbatch <- 40
  nsimsub <- nsim / nbatch
  cvs <- parSapply(cpucl, 1:nbatch, f32, nsimsub=nsimsub)
  print(cvs)
  sum(cvs)/(nsim*nbatch)
}
system.time(cvg2 <- f33())[3]
## [1] 963759 963660 963885 963739 963714 964171 963615 963822 963720 963939
## elapsed
## 13.63
stopCluster(cpucl)
```

并行版本运行了大约 14 秒，速度提高约 210%。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。

5 随机模拟

5.1 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。

5.2 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 = 正面) = 0.7 = 1 - P (X = 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] 3 6 8 9
```

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

```
## [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。

dplyr 包的 sample_n() 函数与 sample() 类似，但输入为数据框，输出为随机抽取的数据框行子集。

5.3 随机模拟示例

5.3.1 线性回归模拟

考虑如下线性回归模型

$$y = 10 + 2x + \epsilon, \epsilon \sim N(0, 0.5^2)$$

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
##      10.140        2.011
```

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
## -0.80770 -0.23111  0.00465  0.20912  0.76998
```

```
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 10.14042    0.46021   22.03 1.90e-08 ***
## x            2.01140    0.06574   30.59 1.42e-09 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.4849 on 8 degrees of freedom
## Multiple R-squared:  0.9915, Adjusted R-squared:  0.9905
## F-statistic:   936 on 1 and 8 DF,  p-value: 1.415e-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) 10.140420  0.4602129 22.03420 1.900574e-08
## x            2.011399  0.0657447 30.59409 1.415175e-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] 10.1404204  2.0113994  0.4602129  0.0657447
```

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()
```

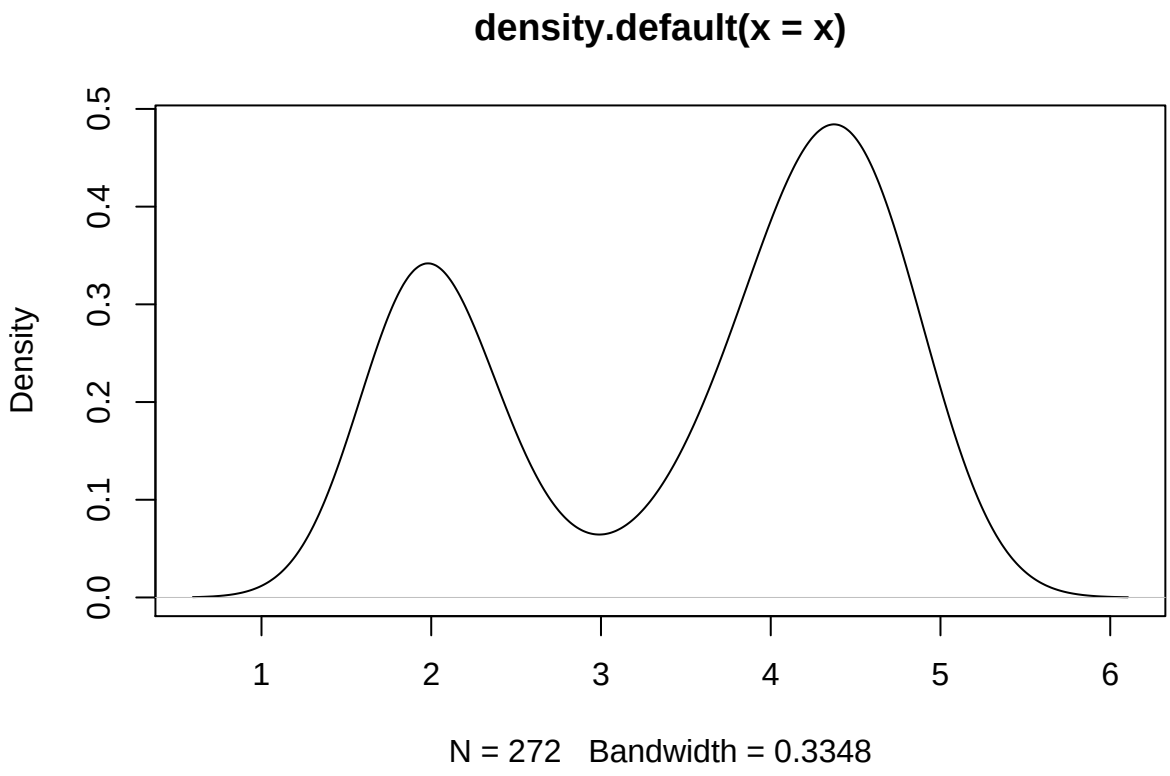
```
## 1000  次模拟的平均值:
##         a         b      SE.a      SE.b
## 9.9970476 1.9994490 0.3639505 0.0592510
## 1000  次模拟的标准差:
##          a          b       SE.a       SE.b
## 0.37974881 0.06297733 0.11992515 0.01795926
```

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。

5.3.2 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：

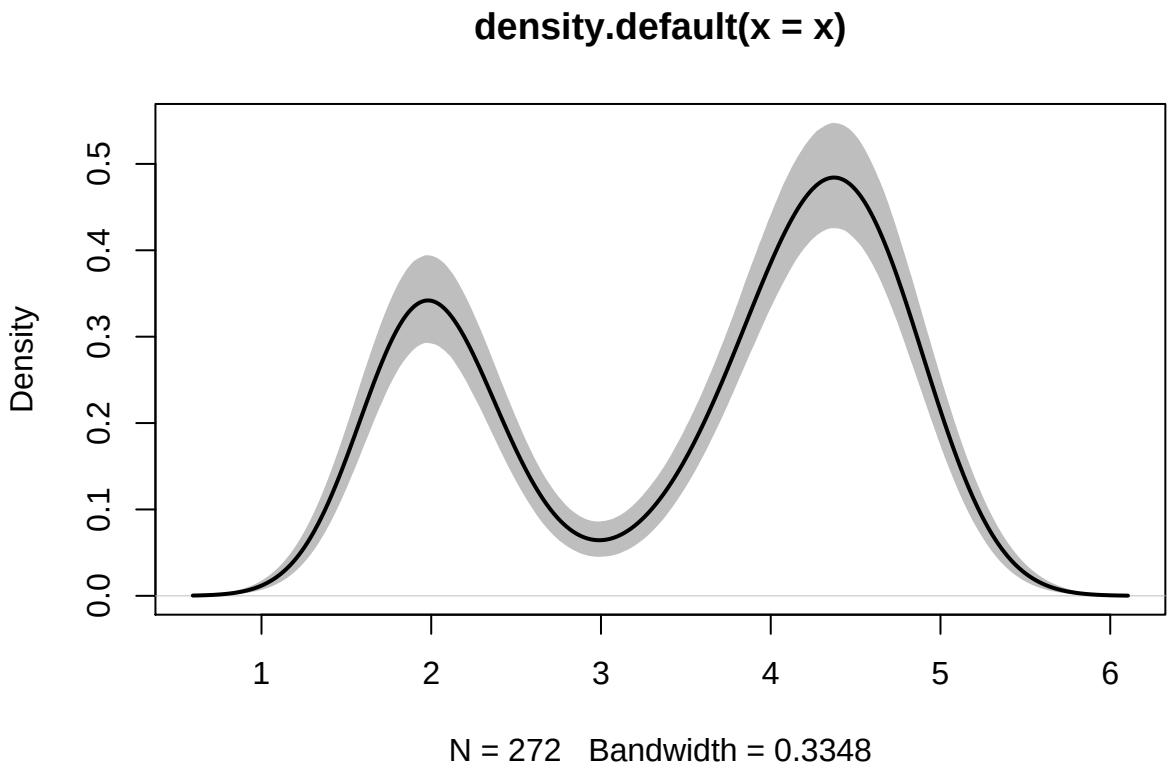
重复 B = 10000 次，每次从原始样本中有重复地抽取与原来大小相同的一组样本，对这组样本计算核密度估计，结果为 $(x_i, y_i^{(j)}), i = 1, 2, ..., N, j = 1, 2, ..., B$，每组样本估计 N 个格子点的密度曲线坐标, 横坐标不随样本改变。

对每个横坐标 x_i，取 bootstrap 得到的 B 个 $y_i^{(j)}, j = 1, 2, ...B$ 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。

6 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。主要参考书：

- (Venables and Ripley, 2002)

- (Kabacoff, 2012)

6.1 概率分布

R 中与 xxx 分布有关的函数包括：

- dxxx(x)，即 xxx 分布的分布密度函数 (PDF) 或概率函数 (PMF)p(x)。
- pxxx(q)，即 xxx 分布的分布函数 (CDF)F (q) = P (X q)。
- qxxx(p)，即 xxx 分布的分位数函数 q(p), p ∈(0, 1)，对连续型分布，q(p) = $F^{-1}(p)$，即 F (x) = p 的解 x。
- rxxx(n)，即 xxx 的随机数函数，可以生成 n 个 xxx 的随机数。

dxxx(x) 函数可以加选项 log=TRUE，用来计算\ln p(x)，这在计算对数似然函数时有用，比 log(dxxx(x)) 更精确。

pxxx(q) 可以加选项 lower.tail=FALSE，变成计算 P (X > q) = 1 - F (q)。

qxxx(p) 可以加选项 lower.tail=TRUE，表示求 P (X > x) = p 的解 x；可以加选项 log.p=TRUE，表示输入的 p 实际是 ln p。

这些函数都可以带有自己的分布参数，有些分布参数有缺省值，比如正态分布的缺省参数值为零均值单位标准差。

具体的分布类型可以在 R 命令行用?Distributions 查看列表。常用的分布密度有：

- 离散分布有 dbinom 二项分布, dpois 泊松分布，dgeom 几何分布，dnbinom 负二项分布，dmultinom 多项分布，dhyper 超几何分布。
- 连续分布有 dunif 均匀分布，dnorm 正态分布，dchisq 卡方分布，dt t 分布 (包括非中心 t)，df F 分布，dexp 指数分布，dweibull 威布尔分布，dgamma 伽马分布，dbeta 贝塔分布，dlnorm 对数正态分布，dcauchy 柯西分布, dlogis 逻辑斯谛分布。

R 的扩展包提供了更多的分布，参见 R 网站的如下链接：

- https://cran.r-project.org/web/views/Distributions.html

6.2 最大似然估计

R 函数 optim()、nlm()、optimize() 可以用来求函数极值，因此可以用来计算最大似然估计。optimize() 只能求一元函数极值。

6.2.1 一元正态分布参数最大似然估计

正态分布最大似然估计有解析表达式。作为示例，用 R 函数进行数值优化求解。对数似然函数为：

$$\ln L(\mu, \sigma^2) = -\frac{n}{2}\ln(2\pi) - \frac{n}{2}\ln(\sigma^2) - \frac{1}{2\sigma^2}\sum(X_i - \mu)^2$$

定义 R 的优化目标函数为上述对数似然函数去掉常数项以后乘以 -2，求其最小值点。目标函数为：

```
objf.norm1 <- function(theta, x){
  mu <- theta[1]
  s2 <- exp(theta[2])
  n <- length(x)
  res <- n*log(s2) + 1/s2*sum((x - mu)^2)
  res
}
```

其中 θ_1 为均值参数 μ，θ_2 为方差参数 σ^2 的对数值。x 是样本数值组成的 R 向量。

可以用 optim 函数来求极小值点。下面是一个模拟演示：

```
norm1d.mledemo1 <- function(n=30){
  mu0 <- 20
  sigma0 <- 2
  set.seed(1)
  x <- rnorm(n, mu0, sigma0)
  theta0 <- c(0,0)
  ores <- optim(theta0, objf.norm1, x=x)
  print(ores)
  theta <- ores$par
  mu <- theta[1]
  sigma <- exp(0.5*theta[2])
  cat(' 真实 mu=', mu0, ' 公式估计 mu=', mean(x),
     ' 数值优化估计 mu=', mu, '\n')
  cat(' 真实 sigma=', sigma0,
      ' 公式估计 sigma=', sqrt(var(x)*(n-1)/n),
      ' 数值优化估计 sigma=', sigma, '\n')
}
norm1d.mledemo1()
```

```
## $par
## [1] 20.166892  1.193593
##
## $value
## [1] 65.83709
##
## $counts
```

```
## function gradient
##      115       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
##
##  真实 mu= 20  公式估计 mu= 20.16492  数值优化估计 mu= 20.16689
##  真实 sigma= 2  公式估计 sigma= 1.817177  数值优化估计 sigma= 1.816291
```

也可以用 nlm() 函数求最小值点，下面是正态分布最大似然估计的另一个演示：

```
norm1d.mledemo2 <- function(){
  set.seed(1)
  n <- 30
  mu <- 20
  sig <- 2
  z <- rnorm(n, mean=mu, sd=sig)
  neglogL <- function(parm) {
    -sum( dnorm(z, mean=parm[1],
                sd=exp(parm[2]), log=TRUE) )
  }
  res <- nlm(neglogL, c(10, log(10)))
  print(res)
  sig2 <- exp(res$estimate[2])
  cat(' 真实 mu=', mu, ' 公式估计 mu=', mean(z),
      ' 数值优化估计 mu=', res$estimate[1], '\n')
  cat(' 真实 sigma=', sig,
      ' 公式估计 sigma=', sqrt(var(z)*(n-1)/n),
      ' 数值优化估计 sigma=', sig2, '\n')
  invisible()
}
norm1d.mledemo2()
```

```
## $minimum
## [1] 60.48667
##
## $estimate
## [1] 20.1649179  0.5972841
```

```
##
## $gradient
## [1] 1.444576e-05 9.094805e-06
##
## $code
## [1] 2
##
## $iterations
## [1] 28
##
##  真实 mu= 20  公式估计 mu= 20.16492  数值优化估计 mu= 20.16492
##  真实 sigma= 2  公式估计 sigma= 1.817177  数值优化估计 sigma= 1.817177
```

函数 optim() 缺省使用 Nelder-Mead 单纯型搜索算法，此算法不要求计算梯度和海色阵，算法稳定性好，但是收敛速度比较慢。可以用选项 method="BFGS" 指定 BFGS 拟牛顿法。这时可以用 gr= 选项输入梯度函数，缺省使用数值微分计算梯度。如：

```
norm1d.mledemo1b <- function(n=30){
  mu0 <- 20
  sigma0 <- 2
  set.seed(1)
  x <- rnorm(n, mu0, sigma0)
  theta0 <- c(1,1)
  ores <- optim(theta0, objf.norm1, x=x, method="BFGS")
  print(ores)
  theta <- ores$par
  mu <- theta[1]
  sigma <- exp(0.5*theta[2])
  cat(' 真实 mu=', mu0, ' 公式估计 mu=', mean(x),
      ' 数值优化估计 mu=', mu, '\n')
  cat(' 真实 sigma=', sigma0,
      ' 公式估计 sigma=', sqrt(var(x)*(n-1)/n),
      ' 数值优化估计 sigma=', sigma, '\n')
}
norm1d.mledemo1b()
```

```
## $par
## [1] 20.164916  1.194568
##
## $value
## [1] 65.83704
```

```
##
## $counts
## function gradient
##       41       17
##
## $convergence
## [1] 0
##
## $message
## NULL
##
##  真实 mu= 20  公式估计 mu= 20.16492  数值优化估计 mu= 20.16492
##  真实 sigma= 2  公式估计 sigma= 1.817177  数值优化估计 sigma= 1.817177
```

注意使用数值微分时，参数的初值的数量级应该与最终结果相差不大，否则可能不收敛，比如上面程序取初值 (0, 0) 就会不收敛。

optim() 函数支持的方法还有 “CG” 是一种共轭梯度法，此方法不如 BFGS 稳健，但是可以减少存储量使用，对大规模问题可能更适用。“L-BFGS-B” 是一种区间约束的 BFGS 优化方法。“SANN” 是模拟退火算法，有利于求得全局最小值点，速度比其它方法慢得多。

函数 optimize() 进行一元函数数值优化计算。例如，在一元正态分布最大似然估计中，在对数似然函数中代入 $= \mathrm{x}$，目标函数（负 2 倍对数似然函数）为

$$h(\sigma^2) = \ln(\sigma^2) + \frac{1}{\sigma^2}\sum(X_i - \bar{X})^2$$

下面的例子模拟计算 σ^2 的最大似然估计：

```
norm1d.mledemo3 <- function(n=30){
  mu0 <- 20
  sigma0 <- 2
  set.seed(1)
  x <- rnorm(n, mu0, sigma0)
  mu <- mean(x)
  ss <- sum((x - mu)^2)/length(x)
  objf <- function(delta, ss) log(delta) + 1/delta*ss
  ores <- optimize(objf, lower=0.0001,
                   upper=1000, ss=ss)
  delta <- ores$minimum
  sigma <- sqrt(delta)
  print(ores)
  cat(' 真实 sigma=', sigma0,
```

```
      ' 公式估计 sigma=', sqrt(var(x)*(n-1)/n),
      ' 数值优化估计 sigma=', sigma, '\n')
}
norm1d.mledemo3()
```

```
## $minimum
## [1] 3.30214
##
## $objective
## [1] 2.194568
##
##  真实 sigma= 2  公式估计 sigma= 1.817177  数值优化估计 sigma= 1.817179
```

6.3 假设检验和置信区间

6.3.1 均值的假设检验和置信区间

6.3.1.1 单样本均值

6.3.1.1.1 大样本情形

设总体 X 期望为 μ，方差为 σ^2。$X_1, X_2, ..., X_n$ 是一组样本。

在大样本时，令

$$Z = \frac{\bar{x} - \mu}{S/\sqrt{n}} \overset{\bullet}{\sim} N(0,1)$$

其中 $\overset{\bullet}{\sim}$ 表示近似服从某分布。$\bar{x}$ 为样本均值，S 为样本标准差。σ 未知且大样本时，μ 近似 1 - σ 置信区间为

$$\bar{x} \pm \text{qnorm}(1 - \frac{\alpha}{2}) S/\sqrt{n}$$

用 BSDA::z.test(x, sigma.x=sd(x), conf.level=1-alpha) 可以计算如上的近似置信区间。

对于假设检验问题

$$H_0 : \mu = \mu_0 \leftrightarrow H_a : \mu \neq \mu_0$$
$$H_0 : \mu \leq \mu_0 \leftrightarrow H_a : \mu > \mu_0$$
$$H_0 : \mu \geq \mu_0 \leftrightarrow H_a : \mu < \mu_0$$

定义检验统计量

$$Z = \frac{\bar{x} - \mu_0}{S/\sqrt{n}}$$

当 $\mu = \mu_0$ 时 Z 近似服从 N(0,1) 分布。可以用 BSDA::z.test(x, mu=mu0, sigma.x=sd(x)) 进行双侧 Z 检验。加选项 alternative="greater" 作右侧检验，加选项 alternative="less" 作左侧检验。如果上述问题中标准差 σ 为已知，应该在 Z 的公式用 σ 替换 S，即

$$Z = \frac{\bar{x} - \mu_0}{\sigma/\sqrt{n}}$$

计算 μ 的置信区间程序为 BSDA::z.test(x, sigma.x=sigma0)，计算 Z 检验的程序为 BSDA::z.test(x, mu=mu0, sigma.x=sigma0)，仍可用选项 alternative= 指定双侧、右侧、左侧。

6.3.1.1.2 小样本正态情形

如果样本量较小（小于 30），则需要假定总体服从正态分布 $N(0,1)$

当 σ 已知时，有

$$Z = \frac{\bar{x} - \mu_0}{\sigma/\sqrt{n}} \sim N(0,1)$$

计算 μ 的置信区间程序为 BSDA::z.test(x, sigma.x=sigma0)，计算 Z 检验的程序为 BSDA::z.test(x, mu=mu0, sigma.x=sigma0)。可以自己写一个计算 Z 检验的较通用的 R 函数：

```
z.test.1s <- function(
  x, n=length(x), mu=0, sigma=sd(x), alternative="two.sided"){
  z <- (mean(x) - mu) / (sigma / sqrt(n))
  if(alternative=="two.sided"){ # 双侧检验
    pvalue <- 2*(1 - pnorm(abs(z)))
  } else if(alternative=="less"){ # 左侧检验
    pvalue <- pnorm(z)
  } else if(alternative=="greater"){ # 右侧检验
    pvalue <- 1 - pnorm(z)
  } else {
    stop("alternative unknown!")
  }
  c(stat=z, pvalue=pvalue)
}
```

这个函数输入样本 x 和 $\sigma = \sigma_0$，或者输入 $barx$ 为 x 和 $\sigma = \sigma_0$，输出检验统计量值和 p 值。

如果 σ 未知，应使用 t 统计量检验

$$Z = \frac{\bar{x} - \mu_0}{S/\sqrt{n}}$$

当 $\mu = \mu_0$ 时 T $\sim$ t(n - 1)。如果 x 保存了样本，用 t.test(x, conf.level=1-alpha) 计算　的置信区间；用 t.test(x, mu=mu0, alternative="two.side") 计算 $H_0 : \mu = \mu_0$ 的双侧检验，称为单样本 t 检验。

如果已知的 $\bar{x}$ 和 S 而非具体样本数据，可以自己写一个 R 函数计算 t 检验：

```
t.test.1s <- function(
  x, n=length(x), sigma=sd(x), mu=0,
  alternative="two.sided"){
  tstat <- (mean(x) - mu) / (sigma / sqrt(n))
  if(alternative=="two.sided"){ # 双侧检验
    pvalue <- 2*(1 - pt(abs(tstat), n-1))
  } else if(alternative=="less"){ # 左侧检验
    pvalue <- pt(tstat, n-1)
  } else if(alternative=="greater"){ # 右侧检验
    pvalue <- 1 - pt(tstat, n-1)
  } else {
    stop("alternative unknown!")
  }

  c(stat=tstat, pvalue=pvalue)
}
```

这个函数允许输入 $\bar{x}$ 到 x 中，输入 n，输入 S 到 sigma 中，然后计算 t 检验。

6.3.1.1.3　统计假设检验显著性的解释

统计假设检验如果拒绝了 H_0，也称 “结果显著”、“有显著差异” 等。这只代表样本与零假设的差距足够大，可以以比较小的代价（第一类错误概率）排除这样的差距是由于随机样本的随机性引起的，但是并不表明这样的差距是有科学意义或者有现实意义的。

尤其是超大样本时，完全没有实际意义的差距也会检验出来是统计显著的。这时，可以设定一个 “有实际意义的差距”，当差距超过　时才认为有实际差距，如：

$$H_0 : |\mu - \mu_0| \leq \delta \leftrightarrow H_a : |\mu - \mu_0| > \delta$$

统计假设检验如果不拒绝 H_0，也称为 “结果不显著”、“没有显著差异”，通常不认为 H_0 就是对的，看法是没有充分理由推翻 H_0。

进一步收集更多的数据，有可能就会发现显著差异。

例如，H_0 是嫌疑人无罪，最后不拒绝 H_0，只能是没有充分证据证明有罪，如果后来收集到了关键的罪证，就还可以再判决嫌疑人有罪。

6.3.1.1.4 置信区间与双侧检验的关系

设总体为正态分布 $N(\mu, sigma^2), sigma$ 已知。随机抽取了 n 个样品，计算得 $\bar{x}$。μ 的 1 ??? α 置信区间为

$$\bar{x} \pm z_{\alpha/2}/\sqrt{n}$$

置信区间理论依据：

$$P(\frac{\bar{x}-\mu}{\sigma/\sqrt{n}} \leq z_{\alpha/2}) = 1-\alpha \qquad (*)$$

对双侧检验

$$H_0 : \mu = \mu_0 \leftrightarrow H_a : \mu \neq \mu_0$$

当

$$\frac{\bar{x}-\mu}{\sigma/\sqrt{n}} \leq z_{\alpha/2}$$

时接受 H_0。

接受 H_0 的条件，就是 μ_0 满足 (*) 式的大括号中的条件，即当且仅当 μ_0 落入 μ 的 1 ??? α 置信区间时接受 H_0。

其它检验也与置信区间有类似关系，但是单侧检验对应于单侧置信区间。

6.3.1.1.5 单样本均值比较例子

载入假设检验部分示例程序所用数据：

```
load("hyptest-data.RData")
```

咖啡标重的单侧检验

美国联邦贸易委员会定期检查商品标签是否与实际相符。Hiltop Coffee 大罐咖啡标签注明含量为 3 磅。每罐具体多一些或少一些并不要紧，只要总体平均值不少于 3 磅，消费者利益就未受侵害。

只有在有充分证据时才应该做出分量不够的判决进而对厂家进行处罚。所以零假设是 $H_0: \geq 3$，对立假设是 $H_a: < 3$。

收集数据进行统计判决后，如果不拒绝 H_0，就不需要采取行动；如果拒绝了 H_0，就说明有足够证据认为厂商灌装分量不足，应进行处罚。

随机抽取了 36 件样品，计算平均净重 $\bar{x}$ 作为总体均值 μ 的估计。如果 $\bar{x} < 3$，则 H_0 有理由被怀疑。

$\bar{x}$ 比 $\mu_0 = 3$ 少多少，才让我们愿意拒绝 H_0? 做出拒绝 H_0 的决定可能会犯第一类错误，如果差距很小，这样的差距很可能是随机抽样的随机性引起的，拒绝 H_0 有很大的机会犯错。

必须先确定一个能够容忍的第一类错误概率。调查员认为，可以容忍 1% 的错误拒绝 H_0 的机会（不该处罚时给出了处罚）以换得能够在厂商分量不足时正确地做出处罚决定的机会。即选检验水平 $\alpha = 0.01$。

当 H_0 实际成立时，边界 $\mu = \mu_0 = 3$ 处犯第一类错误的概率最大（这里最不容易区分），所以计算第一类错误概率只要在边界处计算。

统计假设检验第一步是确定零假设与对立假设，第二步是确定检验水平 α。第三步是抽取随机样本，计算检验统计量。

以往经验证明净重的标准差是 $\sigma = 0.18$，且净重的分布为正态分布。则 $\bar{x}$ 的抽样分布为 $N(\mu, \sigma^2/n)$，$\bar{x}$ 的标准误差为 $\mathrm{SE}(\bar{x}) = 0.18/\sqrt{36} = 0.03$。

令

$$z = \frac{\bar{x} - \mu_0}{SE(\bar{x})}$$

则 H_0 成立且 μ 取边界处的 μ_0 时 $\mathrm{z} \sim \mathrm{N}(0, 1)$。

当 $\bar{x}$ 比 $\mu_0 = 3$ 小很多时应拒绝 H_0。即检验统计量 z 很小时拒绝 H_0。

如果 z = -1，标准正态分布取-1 或比-1 更小的值的概率是 0.1586 (用 R 程序 pnorm(-1) 计算)，这个概率不小，不小于 $\alpha = 0.01$，这时不应该拒绝 H_0。

如果 z = -2，标准正态分布取-2 或比-2 更小的值的概率是 0.0228，(用 R 程序 pnorm(-2) 计算)，这个概率已经比较小了，但还不小于预先确定的第一类错误概率 $\alpha = 0.01$，这时不应该拒绝 H_0。

如果 z = -3，标准正态分布取-3 或比-3 更小的值的概率是 0.0013，(用 R 程序 pnorm(-3) 计算)，这个概率已经很小了，也小于预先确定的第一类错误概率 $\alpha = 0.01$，这时应该拒绝 H_0。

对左侧检验问题 $H_0 : \mu \geq \mu_0 \leftrightarrow \mu < \mu_0$, 若 Z 表示标准正态分布随机变量，z 是检验统计量的值，在 $\mu = \mu_0$ 时 z 的抽样分布是标准正态分布，可以计算抽样分布中取到等于 z 以及比 z 更小的值的概率 P (Z $ $\leq$ z)，称为检验的 p 值。

p 值越小，说明 $\bar{x}$ 比 μ_0 小得越多，拒绝 H_0 也越有充分证据。当且仅当 p 值小于检验水平 α 时拒绝 H_0。

p 值小于 α，说明如果 H_0 真的成立，出现 $\bar{x}$ 那么小的检验统计量值的概率也很小，小于 α，所以第一类错误概率小于等于 α。

一般地，p 值是从样本计算的一个在假定零假设成立情况下，检验统计量取值更倾向于对立假设成立的概率。p 值越小，拒绝 H_0 越有依据。

设咖啡标称重量问题中抽取了 36 件样品，测得 $\bar{x} = 2.92$ 磅，计算得

$$z = \frac{\bar{x} - \mu_0}{\sigma/\sqrt{n}} = \frac{2.92 - 3}{0.18/\sqrt{36}} = -2.67$$

p 值为 $P(Z \leq -2.67)$(Z 是标准正态分布随机变量)，用 R 计算得 pnorm(-2.67)= $0.0038 \leq \alpha = 0.01$，拒绝 H0，应对该厂商进行处罚。

已经预先确定了检验水平 $\alpha = 0.01$ 就不能在看到检验结果后更改检验水平。但是，报告出 p 值后，其他的人可以利用这样的信息，不同的人或利益方可能有不同的检验水平的选择，这通过同一个 p 值都可以给出需要的结果。

p 值也称为“观测显著性水平”，因为它是能够拒绝 H_0 的最小的可选检验水平，选取比 p 值更小的检验水平就不能拒绝 H_0 了。

用自定义的 z.test.1s() 计算：

```
z.test.1s(x=2.92, mu=3, n=36, sigma=0.18, alternative="less")
```

```
##         stat       pvalue
## -2.666666667  0.003830381
```

p 值为 0.004，在 0.01 水平下拒绝原假设，认为咖啡平均重量显著地低于标称的 3 磅。

设 Coffee 数据框的 Weight 是这 36 个样本值，也可以将程序写成：

```
z.test.1s(Coffee[["Weight"]], mu=3, sigma=0.18, alternative="less")
BSDA::z.test(Coffee[["Weight"]], mu=3, sigma.x=0.18, alternative="less")
```

例：高尔夫球性能的双侧检验

美国高尔夫协会对高尔夫运动器械有一系列的规定。MaxFlight 生产高品质的高尔夫球，平均的飞行距离是 295 码。

生产线有时会有波动，当生产的球平均飞行距离不足 295 码时，用户会感觉不好用；当生产的球平均飞行距离超过 295 码时，美国高尔夫协会禁止使用这样的球。

该厂商定期抽检生产线上下来的产品，每次抽取 50 只。

当平均飞行距离与 295 码没有显著差距时，不需要采取措施；如果有显著差距，就需要调整生产线。

所以取 $H_0 : \mu = 295 \leftrightarrow \mu \neq 295$。选检验水平 $\alpha = 0.05$。

随机抽取 n 个样品，在可控条件下测试出每个的飞行距离，计算出平均值 $\bar{x}$。当 $\bar{x}$ 比 μ_0(这里是 295) 小很多或者 $\bar{x}$ 比 μ_0 大很多时拒绝 H_0，否则不拒绝 H_0。

设已知总体标准差 $\sigma = 12$, 且飞行距离服从正态分布。

取统计量

$$z = \frac{\bar{x} - \mu_0}{\sigma/\sqrt{n}}$$

在 H_0 成立时，z 的抽样分布为标准正态分布。

- 设测得 $\bar{x} = 297.6$，超过了 $\mu_0 = 295$。

$$z = \frac{\bar{x} - \mu_0}{\sigma/\sqrt{n}} = \frac{297.6 - 295}{12/\sqrt{50}} = 1.53$$

H_0 成立时 z 抽样分布为标准正态分布，设 Z 是标准正态分布随机变量，Z 取到 1.53 或者比 1.53 更倾向于反对 H_0 的值概率？这包括 P (Z $\geq$ 1.53)，还应包括 $P(Z \leq -1.53)$。

所以 p 值是 P (|Z| $\geq$ |z|)，R 中计算为 2*(1 - pnorm(abs(z)))。p 值等于 0.1260，超过检验水平，不拒绝 H_0，不需要调整生产线。

用自定义的 z.test.1s() 检验：

```
z.test.1s(GolfTest[['Yards']], mu=295, sigma=12, alternative="two.sided")
```

或用 BSDA::z.test():

```
BSDA::z.test(GolfTest[['Yards']], mu=295, sigma.x=12, alternative="two.sided")"
```

p 值为 0.13，在 0.05 水平下不拒绝 H_0，生产的球的平均飞行距离与标准的 295 码之间没有显著差异。

Heathrow 机场打分的检验

某旅行杂志希望根据乘机进行商务旅行的意见给跨大西洋门户机场评分。乘客评分取 0 到 10 分，超过 7 分算作是服务一流的机场。在每个机场随机选取 60 位商务旅行乘客打分评价。英国 Heathrow 机场的打分样本 $\bar{x} = 7.25$, S = 1.052。英国 Heathrow 机场可以算是服务一流吗？

由于随机样本的随机性，要判断的是 $\mu > 7$ 是否成立，这里 $\bar{x} = 7.25$ 并不能很确定第说 $\mu > 7$ 成立。

为将机场评为一流需要有充分证据，所以将对立假设 H_a 设为 $\mu > 7$，然后取 $H_0 : \mu \leq 7$。

取检验水平 $\alpha = 0.05$。

计算检验统计量

$$t = \frac{\bar{x} - \mu_0}{S/\sqrt{n}} = \frac{7.25 - 7}{1.052/\sqrt{60}} = 1.84$$

设 T 是服从 t(n - 1) 分布的随机变量，p 值为 P (T $\geq$ 1.84)。用 R 软件计算

为 1 - pt(1.84, 60-1)= 0.0354。p 值小于 α，拒绝 H_0，认为平均评分显著高于 7，可以认为 Heathrow 机场是服务一流。

从原始样本数据用 t.test() 检验：

```
t.test(AirRating[["Rating"]], mu=7, alternative="greater")
```

用自定义的 t.test.1s() 和样本统计量计算：

```
t.test.1s(x=7.25, sigma=1.052, n=60, mu=7, alternative="greater")
```

```
##       stat     pvalue
## 1.84077155 0.03534255
```

玩具厂商订货量的假设检验

玩具厂商 Holiday Toys 通过超过 1000 家玩具商店售货。为了准备即将到来的冬季销售季节，销售主管需要确定今年某新款玩具的生产量。凭经验判断平均每家商店需要 40 件。

为了确定，随机抽取了 25 家商店，提供了新款玩具的特点、进货价和建议销售价，让他们给出订货量估计。目的是判断按每家 40 件来生产是否合适。

因为多生产或少生产都不合适，所以取零假设为 $H_0 : \mu = 40$。对立假设为 $H_a : \mu \neq 40$。只有拒绝 H_0 时才需要更改生产计划。

取检验水平 $\alpha = 0.05$。样本计算得 n = 25，$\bar{x} = 37.4$, S = 11.79。

检验统计量

$$t = \frac{\bar{x} - \mu_0}{S/\sqrt{n}} = \frac{37.4 - 40}{11.79/\sqrt{25}} = -1.10$$

设 T 为服从 t(n - 1) 分布的随机变量，p 值为 2P (|T | > | - 1.10|)，用 R 计算为 2*(1 - pt(abs(-1.10), 25-1))= 0.2822。

不拒绝 H_0，所以不需要更改生产计划。

从原始样本数据用 t.test() 检验：

```
t.test(Orders[["Units"]], mu=40, alternative="two.sided")
```

用自定义的 t.test.1s() 和样本统计量计算：

```
t.test.1s(x=37.4, sigma=11.79, n=25, mu=40, alternative="two.sided")
```

```
##       stat     pvalue
## -1.1026293  0.2811226
```

6.3.1.2 均值比较

6.3.1.2.1 独立两样本 Z 检验

假设有两个独立的总体 X, Y ，例如，男性与女性的寿命。要比较两个总体的期望 $\mu 1 = EX$ 和 $\mu_2 = EY$ 。设两个总体的方差分别为 $\frac{2}{1}$ 和 $\frac{2}{2}$。可以对均值作双侧、左侧、右侧检验。比如，双侧检验为

$$H_0 : \mu_1 = \mu_2 \leftrightarrow H_a : \mu_1 \neq \mu_2$$

如果两个总体都服从正态分布，而且分别的方差 $\frac{2}{1}$ 和 $\frac{2}{2}$ 已知，设 X 的样本量为 n_1 的样本得到样本均值 $\bar{x}$，设 Y 的样本量为 n_2 的样本得到样本均值 $\bar{y}$，

取统计量

$$Z = \frac{\bar{x} - \bar{y}}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}}$$

其中分母是分子的标准误差。当 $\mu_1 = \mu_2$ 时 $Z \sim N(0, 1)$。

如果分布不一定是正态分布但是样本量足够大，可以令

$$Z = \frac{\bar{x} - \bar{y}}{\sqrt{\frac{S_x^2}{n_1} + \frac{S_y^2}{n_2}}}$$

当 $\mu_1 = \mu_2$ 时 Z 近似服从 N(0, 1)。检验问题还可以变成 $\mu_1 - \mu_2$ 与某个 δ 的比较，如

$$H_0 : \mu_1 \leq \mu_2 - \delta \leftrightarrow H_a : \mu_1 > \mu_2 - \delta$$

这个对立假设是 “不次于” 的结论。

可以写一个大样本方差未知或已知，小样本独立两正态总体方差已知情况做 Z 检验的 R 函数：

```
z.test.2s <- function(
  x, y, n1=length(x), n2=length(y), delta=0,
  sigma1=sd(x), sigma2=sd(y), alternative="two.sided"){
  z <- (mean(x) - mean(y) - delta) / sqrt(sigma1^2 / n1 + sigma2^2 / n2)
  if(alternative=="two.sided"){ # 双侧检验
    pvalue <- 2*(1 - pnorm(abs(z)))
  } else if(alternative=="less"){ # 左侧检验
    pvalue <- pnorm(z)
  } else if(alternative=="greater"){ # 右侧检验
    pvalue <- 1 - pnorm(z)
  } else {
    stop("alternative unknown!")
  }

  c(stat=z, pvalue=pvalue)
}
```

函数允许输入两组样本到 x 和 y 中，输入 $sigma_1$ 和 $sigma_2$ 到 sigma1 和 sigma2 中，这时不输入 sigma1 和 sigma2 则从样本中计算样本统计量代替。也允许输入 $\bar{x}$ 和 $\bar{y}$ 到 x 和 y 中，输入 $sigma_1$ 和 $sigma_2$ 到 sigma1 和 sigma2 中，或者输入 S_x 和 S_y 到 sigma1 和 sigma2 中，输入 n_1 到 n1 中，输入 n_2 到 n2 中，计算独立两样本均值比较的 Z 检验。

6.3.1.2.2 独立两样本 t 检验

如果样本是小样本，但两个总体分别服从正态分布，两个总体的方差未知，但已知 $\sigma_1^2 = \sigma_2^2$。这时 $\sigma_1^2 = \sigma_2^2$ 可以统一估计为

$$S_p^2 = \frac{1}{n_1 + n_2 - 2}((n_1 - 1)S_x^2 + (n_2 - 1)S_y^2)$$

取检验统计量

$$T = \frac{\bar{x} - \bar{y}}{S_p\sqrt{\frac{1}{n_1} + \frac{1}{n_2}}}$$

当 $\mu_1 = \mu_2$ 时 $T \sim t(n_1 + n_2 - 2)$。

当 x 和 y 存放了两组独立样本时，可以用 t.test(x, y, var.equal=TRUE) 计算两样本 t 检验，用 alternative= 选项指定双侧、右侧、左侧检验。

可以自己写一个这样的 R 函数作两样本 t 检验，允许只输入统计量而非具体样本值：

```
t.test.2s <- function(
  x, y, n1=length(x), n2=length(y),
  sigma1=sd(x), sigma2=sd(y), delta=0,
  alternative="two.sided"){
  sp <- sqrt(1/(n1+n2-2) * ((n1-1)*sigma1^2 + (n2-1)*sigma2^2))
  t <- (mean(x) - mean(y) - delta) / (sp * sqrt(1 / n1 + 1 / n2))
  if(alternative=="two.sided"){ # 双侧检验
    pvalue <- 2*(1 - pt(abs(t), n1+n2-2))
  } else if(alternative=="less"){ # 左侧检验
  pvalue <- pt(t, n1+n2-2)
  } else if(alternative=="greater"){ # 右侧检验
  pvalue <- 1 - pt(t, n1+n2-2)
  } else {
    stop("alternative unknown!")
  }

  c(stat=t, pvalue=pvalue)
}
```

如果两个独立正态总体的方差不相等，样本量不够大，可以用如下的在 H_0 下近似服从 t 分布的检验统计量：

$$T = \frac{\bar{x} - \bar{y}}{\sqrt{\frac{S_x^2}{n_1} + \frac{S_y^2}{n_2}}}$$

自由度为

$$m = \frac{(S_x^2/n_1 + S_y^2/n_2)^2}{\frac{(S_x^2/n_1)^2}{n_1-1} + \frac{(S_y^2/n_2)^2}{n_2-1}}$$

这个检验称为 Welch 两样本 t 检验，或 Satterthwaite 两样本 t 检验当 x 和 y 存放了两组独立样本时，可以用 t.test(x, y, var.equal=FALSE) 计算 Welch 两样本 t 检验。

如果方差是否相等不易判断，可以直接选用 Welch 检验。

因为大样本时 t 分布与标准正态分布基本相同，所以非正态总体大样本 Z 检验也可以用 t.test() 计算。

6.3.1.2.3 成对均值比较问题

设总体包含两个 X, Y 分量，比如，一组病人的服药前血压与服药后血压。这两个变量是相关的，不能用独立两总体的均值比较方法比较 $\mu_1 = EX$ 与 $\mu_2 = EY$ 。

若 ξ = X - Y 服从正态分布，则可以对 ξ 进行均值为零的单样本 t 检验。这样的检验称为成对均值的 t 检验。

若 x 和 y 分别为 X 和 Y 的样本值，可以用 t.test(x, y, paired=TRUE) 计算计算成对均值 t 检验。

6.3.1.2.4 均值比较的例子

顾客平均年龄差别比较

HomeStyle 是一个连锁家具店，两家分店一个在城区，一个在郊区。经理发现同一种家具有可能在一个分店卖的很好，另一个分店则不好。怀疑是顾客人群的人口学差异（年龄、性别、种族等）。

独立地抽取两个分店的顾客样本，比较平均年龄。设城内的分店顾客年龄 $\sigma_1 = 9$ 已知，郊区的分店顾客年龄 $\sigma_2 = 10$ 已知。

城内调查了 $n_1 = 36$ 位顾客，年龄的样本平均值 $\bar{x} = 40$; 郊区调查了 $n_2 = 49$ 位顾客，年龄的样本平均值 $\bar{y} = 35$。$\bar{x}$ - $\bar{y} = 5$。

作双侧检验，水平 $\alpha = 0.05$。

计算 Z 统计量：

$$Z = \frac{40 - 35}{\sqrt{\frac{9^2}{36} + \frac{10^2}{49}}} = 2.4138$$

p 值为 P (|V | ≥ |2.4138|),其中 V 为标准正态分布随机变量。用 R 计算为 2*(1 - pnorm(abs(2.4138)))=0.016，两个分店的顾客年龄有显著差异，城里的顾客平均年龄显著地高。

基于样本统计量用自定义的 z.test.2s() 计算：

```
z.test.2s(n1=36, x=40, sigma1=9,
          n2=49, y=35, sigma2=10,
          alternative="two.sided")
```

```
##       stat     pvalue
## 2.41379310 0.01578742
```

两个银行营业所顾客平均存款比较

某银行要比较两个营业所的支票账户的平均存款额。作双侧检验，水平 0.05。没有方差信息时可以直接选用 Welch 检验。

设第一个营业所随机抽查了 $n_1 = 28$ 个支票账户，均值为 $\bar{x} = 1025$(美元)，样本标准差为 $S_1 = 150$，第二个营业所随机抽查了 $n_2 = 22$ 个支票账户，均值为 $\bar{y} = 910$, 样本标准差为 $S_2 = 125$。

从原始数据出发计算检验：

```
t.test(CheckAcct[[1]], CheckAcct[[2]], var.equal = FALSE, alternative = "two.sided")"
```

p 值为 0.005，在 0.05 水平下显著，两个营业所的账户平均余额有显著差异。

两种工具软件的比较

考虑开发信息系统的两种工具软件的比较。新的软件声称比旧的软件能加快进度。设使用旧软件时平均项目完成时间为 μ1，使用新软件时为 μ2。

检验：

$$H_0 : \mu_1 \leq \mu_2 \leftrightarrow H_a : \mu_1 > \mu_1$$

(对立假设是新软件的工期更短) 取 0.05 水平。

```
t.test(SoftwareTest[['Current']], SoftwareTest[['New']],
var.equal = FALSE, alternative = 'greater')
```

p 值为 0.017, 使用新的工具软件的平均项目完成时间显著地少于使用旧工具软件。

两种工艺所需时间的比较

工厂希望比较同一生产问题两种工艺各自需要的时间，将选用时间较短的工艺。可以一组工人用工艺 1，一组工人用工艺 2，随机分组。两组之间工人的个体差异会使得差异的比较误差较大。

所以，让同一个工人分别采用两种工艺，次序先后随机。这样不会引入个体差异的影响，精度较高。使用成对检验。用双侧检验，水平 0.05。

```
t.test(Matched[["Method 1"]], Matched[["Method 2"]],
paired=TRUE, alternative = "two.sided")
```

6.3.1.3 多元均值置信域

待补充。

6.3.2 比例的假设检验和置信区间

6.3.2.1 单个比例的问题

设要比较的是总体的某个比例，如选民中对候选人甲的支持率。这时总体是两点分布总体，支持为 1，不支持为 0，参数为 p = P (X = 1)。其它的比例问题类似，p 是 “成功概率”。

关于单总体的比例，也有双侧、右侧、左侧检验问题：

$$
\begin{aligned}
H_0 : p = p_0 &\leftrightarrow H_a : p \neq p_0 \\
H_0 : p \leq p_0 &\leftrightarrow H_a : p > p_0 \\
H_0 : p \geq p_0 &\leftrightarrow H_a : p < p_0
\end{aligned}
$$

在 R 中，大样本情况下可以用 prop.test(x, n, p=p0) 作单个比例的检验，n 是样本量，x 是样本中符合条件（如支持）的个数，检验使用近似卡方统计量，用 alternative 选项选择双侧、右侧、左侧检验。这个函数也给出比例 p 的近似 1 - α 置信区间，用 conf.level 选项指定置信度。

大样本时还有

$$
\frac{\hat{p} - p}{\sqrt{p(1-p)/n}}
$$

近似服从标准正态分布，可以据此计算 Z 检验。自定义的 R 函数如下：

```
prop.test.1s <- function(x, n, p=0.5, alternative="two.sided"){
  phat <- x/n
  zstat <- (phat - p)/sqrt(p*(1-p)/n)
  if(alternative=="two.sided"){ # 双侧检验
  pvalue <- 2*(1 - pnorm(abs(zstat)))
  } else if(alternative=="less"){ # 左侧检验
    pvalue <- pnorm(zstat)
  } else if(alternative=="greater"){ # 右侧检验
  pvalue <- 1 - pnorm(zstat)
  } else {
    stop("alternative unknown!")
  }
```

```
  c(stat=zstat, pvalue=pvalue)
}
```

因为比例问题本质上是一个二项分布推断问题，所以 R 还提供了 binom.test(x, n, p=p0) 函数进行精确检验并计算精确置信区间，但其结果略保守。用法与 prop.test() 类似。

6.3.2.2 两个比例的比较

设有 X 和 Y 两个独立的总体，如男性和女性，比较 $p_1 = \mathrm{P}\ (\mathrm{X} = 1)$ 和 $p_2 = \mathrm{P}\ (\mathrm{Y} = 1)$。

检验问题包括：

$$H_0 : p_1 = p_2 \leftrightarrow H_a : p_1 \neq p_2$$
$$H_0 : p_1 \leq p_2 \leftrightarrow H_a : p_1 > p_2$$
$$H_0 : p_1 \geq p_2 \leftrightarrow H_a : p_1 < p_2$$

在大样本情况下，可以用 R 函数 prop.test(c(nsucc1, nsucc2), c(n1,n2), alternative=...) 进行独立两样本比例的比较检验，其中 n1 和 n2 是两个样本量，nsucc1 和 nsucc2 是两个样本中符合特征的个数（个数除以样本量即样本中的比例），alternative 的选择也是 “two.sided”、“less”、“greater”。还可以比较 p1 - p2 与某个 δ。设两个总体中分别抽取了 n1 和 n2 个样本点，样本比例分别为 $\hat{p}_1$ 和 $\hat{p}_2$。估计共同的样本比例

$$\hat{p} = \frac{n_1\hat{p_1} + n_2\hat{p_2}}{n_1 + n_2}$$

当 $\delta = 0$ 时取统计量

$$Z = \frac{\hat{p}_1 - \hat{p}_2}{\sqrt{\hat{p}(1-\hat{p})(1/n_1 + 1/n_2)}}$$

当 p1 = p2 时 Z 在大样本情况下近似服从标准正态分布。当 $\delta \neq = 0$ 时取统计量

$$Z = \frac{\hat{p}_1 - \hat{p}_2}{\sqrt{\frac{\hat{p}_1(1-\hat{p}_1)}{n_1} + \frac{\hat{p}_2(1-\hat{p}_2)}{n_2}}}$$

当 p1 - p2 = δ 时 Z 在大样本情况下近似服从 N(0,1) 分布。

可以定义 Z 检验 R 函数为：

```
prop.test.2s <- function(x, n, delta=0.0, alternative="two.sided"){
  phat <- sum(x)/sum(n)
  p <- x / n
```

```
  if(delta==0.0){
  zstat <- (p[1] - p[2])/sqrt(phat*(1-phat)*(1/n[1] + 1/n[2]))
  } else {
  zstat <- (p[1] - p[2] - delta)/sqrt(p[1]*(1-p[1])/n[1] +
                                      p[2]*(1-p[2])/n[2])
  }
  if(alternative=="two.sided"){ # 双侧检验
    pvalue <- 2*(1 - pnorm(abs(zstat)))
  } else if(alternative=="less"){ # 左侧检验
    pvalue <- pnorm(zstat)
  } else if(alternative=="greater"){ # 右侧检验
    pvalue <- 1 - pnorm(zstat)
  } else {
    stop("alternative unknown!")
  }

  c(stat=zstat, pvalue=pvalue)
}
```

独立两总体比例比较的小样本情形意义不大，可考虑使用 Fisher 精确检验，R 函数为 fisher.test()。

6.3.2.3 比例检验的例子

6.3.2.3.1 高尔夫培训女生比例检验例子

橡树溪高尔夫培训机构的女性学员比例较少，只有 20%。为了增加女性学员，设计了促销措施。一个月后调查，希望证明女性学员比例增加了。

设女性学员比例为 p，

$$H_0 : p \leq 0.20 \leftrightarrow H_a : p > 0.20$$

取检验水平 0.05。抽查了 400 名学员，其中 100 名是女性学员，$\hat{p} = 0.25$。用 prop.test() 检验：

```
prop.test(100, 400, p=0.20, alternative = "greater")
```

```
##
##  1-sample proportions test with continuity correction
##
## data:  100 out of 400, null probability 0.2
## X-squared = 5.9414, df = 1, p-value = 0.007395
```

```
## alternative hypothesis: true p is greater than 0.2
## 95 percent confidence interval:
##  0.2149649 1.0000000
## sample estimates:
##    p
## 0.25
```

检验 p 值为 0.0074，小于检验水平 0.05，所以女性学员比例已经显著地高于过去的 20% 了。

用自定义的大样本 Z 检验函数 prop.test.1s():

```
prop.test.1s(100, 400, p=0.20, alternative = "greater")
```

```
##        stat      pvalue
## 2.500000000 0.006209665
```

检验 p 值为 0.0062，与基于卡方检验的 prop.test() 结果略有差别。

用基于二项分布的 binom.test() 检验:

```
binom.test(100, 400, p=0.20, alternative = "greater")
```

```
##
##  Exact binomial test
##
## data:  100 and 400
## number of successes = 100, number of trials = 400, p-value =
## 0.008595
## alternative hypothesis: true probability of success is greater than 0.2
## 95 percent confidence interval:
##  0.2146019 1.0000000
## sample estimates:
## probability of success
##                   0.25
```

p 值为 0.0086。

6.3.2.3.2 报税代理分理处的错误率比较

某个代理报税的机构希望比较下属的两个分理处申报纳税返还的出错率。各自随机选取取了 n1 = 250 和 n2 = 300 件申报，第一个分理处错了 35 件，错误率 $\hat{p}_1 = 0.14$; 第二个分理处错了 27 件，错误率 $\hat{p}_2 = 0.09$。

作双侧检验，水平 0.10。

用 prop.test():

```r
prop.test(c(35,27), c(250,300), alternative = "two.sided")
```

```
##
##  2-sample test for equality of proportions with continuity
##  correction
##
## data:  c(35, 27) out of c(250, 300)
## X-squared = 2.9268, df = 1, p-value = 0.08712
## alternative hypothesis: two.sided
## 95 percent confidence interval:
##  -0.007506845  0.107506845
## sample estimates:
## prop 1 prop 2
##   0.14   0.09
```

p 值 0.087，有显著差异，第一处的错误率高。因为比例需要较大样本量，所以样本量不太大时，有人用 0.10 这样的检验水平。

用自定义的 prop.test.2s():

```r
prop.test.2s(c(35,27), c(250,300), alternative = "two.sided")
```

```
##       stat     pvalue
## 1.84618928 0.06486473
```

6.3.3 方差的假设检验和置信区间

6.3.3.1 单总体方差的假设检验和置信区间

方差和标准差是总体的重要数字特征，在金融应用中标准差代表波动率，在工业生产中标准差也是重要的质量指标，如机床加工精度可以用标准差表示。

对总体方差 σ^2 可以做双侧、左侧、右侧检验，与某个 σ_0^2 比较。

设总体为正态总体，检验问题：

$$
\begin{aligned}
H_0 : \sigma^2 = \sigma_0^2 &\leftrightarrow H_a : \sigma^2 \neq \sigma_0^2 \\
H_0 : \sigma^2 \leq \sigma_0^2 &\leftrightarrow H_a : \sigma^2 > \sigma_0^2 \\
H_0 : \sigma^2 \geq \sigma_0^2 &\leftrightarrow H_a : \sigma^2 < \sigma_0^2
\end{aligned}
$$

可以用如下的自定义函数计算单正态总体方差的检验：

```r
var.test.1s <- function(x, n=length(x), var0, alternative="two.sided"){
  if(length(x)==1){ # 输入的是方差
  varx <- x
  } else {
    varx <- var(x)
  }
  xi <- (n-1)*varx/var0

  if(alternative=="less"){
    pvalue <- pchisq(xi, n-1)
  } else if (alternative=="right"){
    pvalue <- pchisq(xi, n-1, lower.tail=FALSE) }
  else if(alternative=="two.sided"){
    pvalue <- 2*min(pchisq(xi, n-1), pchisq(xi, n-1, lower.tail=FALSE))
  }

  c(statistic=xi, pvalue=pvalue)
}
```

可以输入样本数据到 x 中，输入 $sigma_0^2$ 到 var0 中，进行检验；也可以输入 S^2 到 x 中，输入样本量 n 到 n 中，输入 $sigma_0^2$ 到 var0 中，进行检验。

6.3.3.2 独立两总体方差的比较

设两个独立的正态总体的方差分别为 σ_1^2, σ_2^2，有双侧、左侧、右侧检验：

$$
\begin{aligned}
H_0 : \sigma_1^2 = \sigma_2^2 \leftrightarrow H_a : \sigma_1^2 \neq \sigma_2^2 \\
H_0 : \sigma_1^2 \leq \sigma_2^2 \leftrightarrow H_a : \sigma_1^2 > \sigma_2^2 \\
H_0 : \sigma_1^2 \geq \sigma_2^2 \leftrightarrow H_a : \sigma_1^2 < \sigma_2^2
\end{aligned}
$$

设两个总体分别抽取 n1 和 n2 个样本点，样本方差分别为 S_x^2 和 S_y^2。

检验统计量

$$
F = \frac{S_x^2}{S_y^2}
$$

在 $\sigma_1^2 = \sigma_2^2$ 时 F 服从 F (n_1 - 1, n_2 - 1) 分布。

如果输入 x, y 为两个样本的具体值，可以用 var.test(x, y, alternative=…) 检验，其中 alternative 的选择是 “two.sided”、“less”、“greater”。

var.test() 假定总体都服从正态分布。R 中 mood.test() 是一种不要求正态分布假定的检验方法。bartlett.test() 检验多个独立正态总体方差是否全相等。

6.3.3.3 方差检验例子

(待补充)

6.3.4 拟合优度检验

6.3.4.1 各类比例相等的检验

设类别变量 X 有 m 个不同类别，抽取 n 个样本点后，各类的频数分别为 $f_1, f_2, ..., f_m$。

零假设为所有类的比例相同；对立假设为各类比例不完全相同。

在零假设下，每个类的期望频数为 n/m。

检验统计量为

$$\chi^2 = \sum_{i=1}^{n} \frac{(f_i - n/m)^2}{n/m}$$

当 χ^2 超过某一临界值时拒绝零假设。在 H0 成立且大样本情况下 χ^2 统计量近似服从 χ^2(m - 1) 分布（自由度为组数减 1）。

设 x 中包含各类的频数，R 函数 chisq.test(x) 作各类的总体比例相等的拟合优度卡方检验。

6.3.4.2 各类比例为指定值的检验

设分类变量 X 的各类每类有一个总体比例的假设，希望作为零假设检验。设共有 m 个类，在零假设中，各类的比例分别为 $p_1, p_2, ..., p_m$。

取了 n 个样本点组成的样本，各类的频数分别为 $f_1, f_2, ..., f_m$，期望频数分别为 $np_1, np_2, ..., np_m$。

检验统计量为

$$\chi^2 = \sum_{i=1}^{m} \frac{(f_i - np_i)^2}{np_i}$$

在 H0 下成立且大样本情况下 χ^2 近似服从 χ^2(m - 1)，自由度为组数减 1。因为是大样本检验法，每个组的期望频数不能低于 5，否则可以合并较小的类。

设 x 中包含各类的频数，p 中包含 H0 假定的各类的概率，R 函数 chisq.test(x, p) 作各类的总体比例为指定概率的拟合优度卡方检验。

6.3.4.3 带有未知参数的单分类变量的拟合优度假设检验

如果 H0 下的概率有未知参数，先用最大似然估计法估计未知参数，然后将未知参数代入计算各类的概率，从而计算期望频数，计算卡方统计量。求 p 值时自由度要改为“组数减 1 减去未知参数个数”。

例：设某次选举有 5 位候选人，其中呼声最高的是甲和乙。问：甲、乙的支持率相等吗？

这不能用独立的两个比例的比较来解决，因为这两个比例是互斥的，不属于独立情况。

将类别简化为：甲，乙，其他。零假设为“甲、乙的支持率都等于 p，其他三位候选人的支持率为 1 - 2p，p 未知”；对立假设是甲、乙的支持率不相等。

假设调查了 1000 位选民，其中 300 名支持甲，200 名支持乙，500 名支持其他三位候选人。

在假定零假设成立的情况下先用最大似然估计法估计未知参数 p：$\hat{p} = (300 + 200)/1000/2 = 0.25$。这样，三个类的概率估计为 (0.25, 0.25, 0.50)。

然后，计算检验统计量：

```
chisq.test(c(300, 200, 500), c(0.25, 0.25, 0.50))
```

其中的 p 值所用的自由度错误，我们需要自己计算 p 值：

```
res <- chisq.test(c(300, 200, 500), c(0.25, 0.25, 0.50))
c(statistic=res$statistic, pvalue=pchisq(res$statistic, res$parameter - 1, lower.tail = FALSE))
```

检验 p 值为 0.08 而非原来的 0.22。

6.3.4.4 拟合优度检验例子

6.3.4.4.1 市场占有率的检验例子

某类产品中公司 A 占 30%, 公司 B 占 50%，公司 C 占 20%。近期 C 公司推出了新产品代替本公司的老产品。Scott Marketing Research 是一个调查公司，公司 C 委托 Scott Marketing Research 调查分析新产品是否导致了市场占有率的变动。

设 p1, p2, p3 分别为三个公司现在的市场占有率，零假设为:

$$H_0 : p_1 = 0.30, p_2 = 0.50, p_3 = 0.20$$

取检验水平 0.05。抽查了 200 位消费者，购买三个公司的产品的人数分配为：

$$(48, 98, 54)$$

卡方统计量

$$\chi^2 = \frac{(48 - 200 * 0.3)^2}{200 * 0.3} + \frac{(98 - 200 * 0.5)^2}{200 * 0.5} + \frac{(54 - 200 * 0.2)^2}{200 * 0.2} = 7.34$$

p 值 $=$ P $(\chi^2(3 - 1) > 7.34) = 0.02548$，拒绝零假设，认为市场占有率有变化。

用 chisq.test() 计算：

```
chisq.test(c(48, 98, 54), p=c(0.3, 0.5, 0.2))
```

```
##
##  Chi-squared test for given probabilities
##
## data:  c(48, 98, 54)
## X-squared = 7.34, df = 2, p-value = 0.02548
```

6.3.5 列联表独立性卡方检验

设分类变量 X 有 m 个类，分类变量 Y 有 k 个类，抽取了样本量为 n 的样本，设 X 的第 i 类与 Y 的第 j 类交叉的频数为 f_{ij}。

零假设为 X 与 Y 相互独立，即行变量与列变量相互独立。

交叉频数列成列联表形式，第 i 行第 j 列为 f_{ij}。

第 i 行的行和为 $f_{i.}$, X 的第 i 类的百分比为 $r_i = f_{i.}/\mathrm{n}$;

第 j 列的列和为 $f_{.j}$, Y 的第 j 类的百分比为 $c_j = f_{.j}/\mathrm{n}$。

当 X, Y 独立时，(i, j) 格子的期望频数为 $E_{ij} = n \text{»} r_i \text{»} c_j$。

列联表独立性检验统计量

$$\chi^2 = \sum_{i=1}^{m} \sum_{j=1}^{k} \frac{(f_i - E_{ij})^2}{E_{ij}}$$

其中 $E_{ij} = n r_i c_j$，r_i 为 X 第 i 类的百分比，c_j 为 Y 第 j 类的百分比。

在 H0 下 χ^2 近似服从 (m - 1)(k - 1) 自由度的卡方分布。设统计量值为 c，p 值为 P $(\chi^2((\mathrm{m} - 1)(\mathrm{k} - 1)) > \mathrm{c})$。

如果 x, y 分别是两个变量的原始观测值，chisq.test(x, y) 可以做列联表独立性检验。如果 x 保存了矩阵格式的列联表，矩阵行名是 X 各个类的名称，矩阵列名是 Y 各个类的名称，则 chisq.test(x) 可以做列联表独立性检验。

列联表卡方检验法的检验统计量在零假设下的卡方分布是大样本情况的近似分布。如果每个变量仅有两个类，每个类的期望频数不能少于 5; 如果有多个单元格，期望频数少于 5 的单元格的个数不能超过 20%，否则应该合并较小的类。

6.3.5.1 列联表独立性卡方检验例子

6.3.5.1.1 性别与啤酒种类的独立性检验

Alber's Brewery of Tucson, Arizona 是啤酒制造与销售商。有三类啤酒产品：淡啤酒，普通啤酒，黑啤酒。

了解不同顾客喜好有利于制定更精准的销售策略。希望了解男女顾客对不同类型的偏好有没有显著差异，实际就是检验性别与啤酒类型偏好的独立性。抽查了 150 位顾客，得到如下的列联表：

```
ctab.beer <- rbind(c(
20, 40, 20),
c(30,30,10))
colnames(ctab.beer) <- c("Light", "Regular", "Dark")
rownames(ctab.beer) <- c("Male", "Female")
addmargins(ctab.beer)
```

```
##        Light Regular Dark Sum
## Male      20      40   20  80
## Female    30      30   10  70
## Sum       50      70   30 150
```

列联表独立性检验：

```
chisq.test(ctab.beer)
```

```
##
##  Pearson's Chi-squared test
##
## data:  ctab.beer
## X-squared = 6.1224, df = 2, p-value = 0.04683
```

在 0.05 水平下认为啤酒类型偏好与性别有关。男性组的偏好分布、女性组的偏好分布、所有人的偏好分布：

```
tab2 <- round(prop.table(addmargins(ctab.beer, 1), 1), 3)
rownames(tab2)[3] <- "All"
tab2
```

```
##        Light Regular  Dark
## Male   0.250   0.500 0.250
## Female 0.429   0.429 0.143
## All    0.333   0.467 0.200
```

可以看出男性中对淡啤酒偏好偏少，女性中对淡啤酒偏好偏多。

6.3.6 非参数检验

待完成。