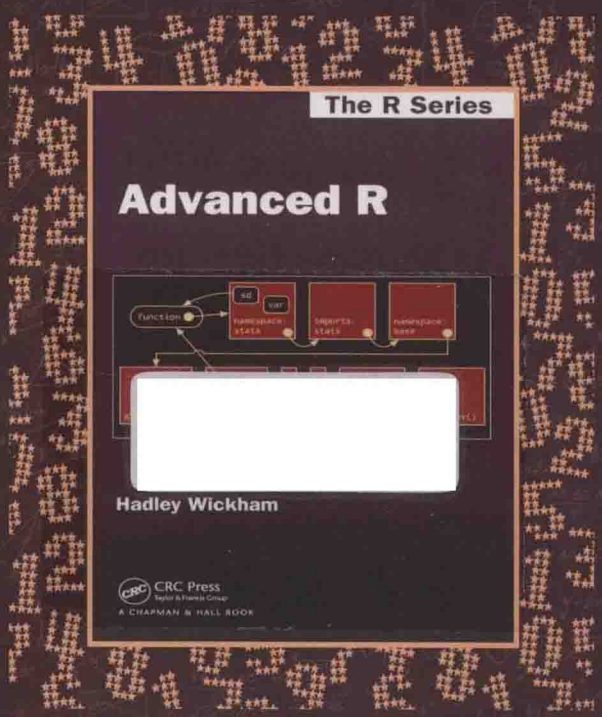
HZ BOOKS
华章IT
CRC Press
Taylor & Francis Group

数据科学与工程技术丛书

高级R语言编程指南

[美] 哈德利·威克汉姆（Hadley Wickham）著
李洪成 段力辉 何占军 译



ADVANCED R

机械工业出版社
China Machine Press

数据科学与工程技术丛书

ADVANCED R

高级R语言编程指南

[美] 哈德利·威克汉姆（Hadley Wickham） 著

李洪成 段力辉 何占军 译

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

高级 R 语言编程指南 /（美）哈德利 · 威克汉姆（Hadley Wickham）著；李洪成，段力辉，何占军译 . —北京：机械工业出版社，2016.6
（数据科学与工程技术丛书）
书名原文：Advanced R

ISBN 978-7-111-54067-0

I. 高… II. ① 哈… ② 李… ③ 段… ④ 何… III. 程序语言－程序设计 IV. TP312

中国版本图书馆 CIP 数据核字（2016）第 130804 号

本书版权登记号：图字：01-2015-3694

本书从 R 语言的基础知识入手，深入而详细地介绍了 R 语言及其编程技术。本书共分 4 部分：R 语言基础、函数式编程、R 语言计算和 R 性能分析。第一部分详细介绍 R 的基础知识，包括数据结构、子集选取、常用函数与数据结构、编程风格指南、函数、面向对象编程、环境、程序调试和防御性编程等。第二部分介绍函数式编程、泛函、函数运算符等。第三部分介绍非标准求值、表达式、领域特定语言（HTML 和 LaTex）等。第四部分介绍 R 的性能、代码调优、内存管理、高性能计算和 C 语言编程接口等。

本书适合 R 的初学者和具有一定编程经验的 R 用户，他们都能从书中找到对自己有用的内容。

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：盛思源　　责任校对：殷　虹
印　　刷：三河市宏图印务有限公司　　版　　次：2016 年 6 月第 1 版第 1 次印刷
开　　本：185mm × 260mm　1/16　　印　　张：19.5
书　　号：ISBN 978-7-111-54067-0　　定　　价：79.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

推 荐 序

根据调查，R 语言现在是高级分析、统计学、数据可视化和预测模型开发领域最流行的工具。在 Statistics.com 在线课程中，R 语言方面的课程都有大量的需求，这些课程包括本书作者 Hadley Wickham 开发的 ggplot2、数据挖掘、统计模型、地图、空间分析和贝叶斯统计等。R 语言流行的一个原因是，巨大的 R 社区贡献了数千个适用于各种应用的 R 添加包。

R 在统计学家和数据挖掘人员中的流行，奠定了它在编程语言中的独特地位。最初，用户应用 R 进行统计模型拟合、数据可视化或者其他的统计任务。对这些用户而言，R 是一个功能强大的、免费的、基于命令行的统计软件。至少，一开始他们可能不需要开发高效的、长度太大的复杂代码。这些用户随着应用 R 的范围变广、应用加深，把 R 作为一门编程语言来进行最佳实践的需求就变得迫切了。

本书是进一步学习 R 语言和进行标准编程实践一个很好的指南，它与应用 R 进行数据分析密切相关。该书系统地讲解了数据结构、子集选取、函数、泛函和面向对象编程。那些被 R 的速度慢而困扰的用户可能喜欢本书有关内存应用、代码优化和高性能计算部分的内容。

越多的 R 用户遵从本书开始几章列出的原则，在 R 社区中就会看到更多的高水平的 R 代码。

美国统计教育学院，Statistics.com 在线课程网站总裁和创始人 Peter Bruce

According to some surveys, R is now the most popular tool for advanced analytics, statistics, visualization and predictive model development. At Statistics.com, courses in R are in high demand, including the ggplot2 course that Hadley Wickham developed, as well as courses in data mining, statistical modeling, mapping, spatial analysis, Bayesian statistics, and more. One reason for this is the huge global R community, which has contributed thousands of R packages for all kinds of purposes.

R's popularity among statisticians and data miners has given it a unique position among programming languages. Most of its users originally came to R with a goal of fitting a statistical

model, producing a visualization, or performing some other statistical task. For these users, R is essentially a powerful, free command-line statistical software program. They probably did not need to create code that was lengthy, complex, or especially fast, at least at first. With time, though, as the general use of R broadened and deepened, the need to treat R as a programming language, and observe "best practices," grew.

This book provides a guide for these users who need to step back and learn the nitty-gritty of standard programming practices, as it they pertain to the use of R for data analysis. It provides a systematic treatment of data structures, notation, subsetting, functions, and functional and object-oriented programming. Those frustrated with R's slowness will like the sections on memory use, code optimization and high performance computing. The more R-users who follow the principles outlined in the coming chapters, the better the overall level of code we will see in the R community.

Peter Bruce

President and Founder

The Institute for Statistics Education at Statistics.com

译 者 序

随着大数据的概念变得越来越流行，对数据的探索、分析和预测将成为大数据分析领域的基本技能之一。作为探索和分析数据的基本工具，数据分析软件包是必不可少的。R软件作为功能强大且开源的数据分析工具，在数据分析领域获得了越来越多用户的青睐。目前，市场上出版了大量的与R语言有关的书籍，这些书籍基本可以分为两类：一类是通过R语言介绍某个主题或者课程；另一类是R软件或者R语言入门性质的介绍。目前尚没有一本深入而详细地介绍R语言与编程的书籍。本书恰好填补了这方面的空白。

本书从R语言的基础知识入手，深入介绍R函数式编程、R语言的面向对象特性、程序调试、代码优化和性能调优。同时，本书也介绍了R语言如何与HTML和LaTex语言相结合的技术，介绍了高性能计算以及C语言编程接口。

本书作者Hadley Wickham是R语言专家，他具有10年以上应用R语言的实践经验，他编写了许多高质量的R添加包，例如ggplot2、plyr、reshape2等，这些都是在R社区广泛使用的添加包。通过本书的学习，一方面可以更深入地了解R语言编程的核心知识，另一方面也可以在某种程度上了解这样一位众所周知的R语言专家所编写的R添加包。

本书共分4部分：R语言基础、函数式编程、R语言计算和R性能分析。第一部分详细介绍R的基础知识，包括数据结构、子集选取、常用函数与数据结构、编程风格指南、函数、面向对象编程、环境、程序调试和防御性编程；第二部分介绍函数式编程、泛函、函数运算符；第三部分介绍非标准求值、表达式、领域特定语言（HTML和LaTex）；第四部分介绍R的性能、代码调优、内存管理、高性能计算和C语言编程接口。

本书是学习R语言难得的手册。不管是初学者还是具有一定编程经验的R用户，都可以从本书获益。初学者可以先从第一部分入手，然后根据需要逐步学习后面的部分。熟练的R用户可以从自己感兴趣的内容入手。

本书的翻译得到了国家自然科学基金（项目编号71461005）和广西高校数据分析与计算重点实验室的资助。在本书的翻译过程中，得到了明永玲编辑的大力支持和帮助。本书责任编辑盛思源老师具有丰富的经验，为本书的出版付出了大量的劳动。这里对她们的支持和帮助表示衷心的感谢。本书的翻译工作主要由李洪成、段力辉共同完成，何占军和吴立明协助翻译了本书的部分内容。

由于时间和水平所限，难免会有不当之处，希望同行和读者多加指正。

目　录

第四部分　性能

第 1 章
简　介

本书可以帮助你避免犯我曾经犯过的错误，带你走出我曾经无法跳出的死胡同，并会教给你使用一些有用的工具、技术以及习惯，从而帮助你解决编程中遇到的多种问题。在这个过程中，我希望让你知道虽然 R 语言有很多令人沮丧的奇怪之处，但是 R 从本质上来说是一款十分适合数据分析和统计的优雅而美丽的语言。

如果你刚刚接触 R，你可能会问这门奇怪的语言有什么值得学习的呢。对于我来说，R 语言最好的特性有下面几条：

- R 是免费、开源、跨平台的。所以如果使用 R 进行数据分析，每个人都可以很容易地复制它（重现分析过程）。
- 大量可以用于统计建模、机器学习、可视化、数据导入与操作的添加包（package）可以供 R 使用。可能别人都已经做好了你正在尝试构建的任何模型或者图形。即使不能完全照搬，你也会从他们的工作中学到很多经验从而对你的工作起到帮助作用。
- 最前沿的工具。统计学和机器学习领域的研究者在发表他们的研究论文时，经常会同时发布一个相应的 R 添加包。这就意味着你可以马上获得最新的统计技术并可以迅速实施。
- 对数据分析根深蒂固的支持。它包括很多特性，比如缺失值、数据框和子集选取。
- R 语言的爱好者组成了一个非常棒的社区。你可以非常容易地在 R- 帮助邮件列表（https://stat.ethz.ch/mailman/listinfo/r-help），stackoverflow（http://stackoverflow.com/questions/tagged/r），或者在其他一些针对特殊项目的邮件列表如 R-SIG-mixed-models（https://st at.ethz.ch.cnl/mailman/listinfo/r-sig-mixed-models） 或 ggplot2（https://groups.google.com/forum/#!forum/ggplot2）中得到专家的帮助。你也可以通过 Twitter（https://twitter.com/search?q=%23rstats）、LinkedIn（http://www.linkedin.com/groups/R-Project-Statistical-Computing-77616），或者用户组（http://blog.revolutionanalytics.com/local-r-groups.html）与其他 R 学员进行联系。
- 交流结果的强大工具。R 添加包可以非常容易地将结果输出成 html 或 pdf 报告（http://yihui.name/knitr/），或者创建一个交互式网站（http://www.rstudio.com/shiny）。
- 强大的函数式编程基础。函数式编程的思想非常适合应对很多数据分析的挑战。R 提供了一个强大而又灵活的工具包，让你可以写出简洁的描述性代码。

- 一个为交互式数据分析和统计编程量身定做的IDE（交互式开发环境）（http://www.rstudio.com/ide/）。
- 强大的元编程（metaprogramming）工具。R不仅仅是一门编程语言，还是一个数据分析的交互环境。它的元编程功能可以让你写出超级简洁明了的函数并为设计领域特定语言（domain-specific language）提供了出色的环境。
- 可以与高性能编程语言（如C、Fortran和C++）连接。

当然R也不是完美的。R最大的挑战就是大多数用户都不是程序员。这就意味着：

- 你看到的很多R代码都是在急于解决某个紧迫问题的情况下编写的。因此这些代码并不是非常简洁、高效或者易于理解。大多数用户不会修改他们的代码来克服这些缺点。
- 与其他编程语言相比，R社区更注重结果而非过程。软件工程最佳实践的知识不够完整。例如，使用源代码控制或自动化测试的R程序员还不够多。
- 元编程是一把双刃剑。有太多的R函数通过使用一些技巧来减少代码的输入量，由此造成的结果就是使代码变得很难理解，有些还会以意想不到的方式失败。
- 不同作者贡献的各种R添加包之间经常会出现矛盾，甚至与R的基础包发生冲突。每次使用R时，你都要面对它20多年的进化史。由于需要记住很多特例，所以R语言的学习会比较困难。
- R并不是速度很快的编程语言，尤其是写得很差的R代码运行起来会非常慢。R还非常耗费内存。

从个人角度来看，我觉得这些挑战也为有经验的程序员创造了一个极大的机会，让他们可以对R和R社区产生深远而又有益的影响。R用户确实应该写出高质量的代码，尤其是在进行可重复研究时，但是他们现在还不具有这样的能力。我希望本书不仅可以帮助更多的R用户成为R程序员，还非常鼓励正在使用其他语言的程序员对R语言做出贡献。

1.1 本书的目标读者

本书是针对两个互补的群体：

- 想深入学习R并学习解决各种问题的新策略的中级R程序员。
- 正在学习R，并想知道R为什么这样工作的其他语言的程序员。

要从本书获得最大收益，你需要编写大量的R语言或者其他语言的代码。你可能不了解函数的所有细节，尽管你目前可能为如何高效地应用它们而努力，但是你应该熟悉R中的函数是如何工作的，你应该熟悉apply系列函数（如`apply()`和`lapply()`）。

1.2 通过本书你可以学到什么

我认为本书描述的这些技能是一个高级R程序员应该掌握的：能够写出可以在各种环境中使用的高质量代码。

读完本书之后，你将：

- 熟悉R的基础。你将理解复杂的数据类型和对它们进行运算的最佳方法。对函数如何工作有更深入的理解，还将认识并使用R的4个对象系统。
- 理解什么是函数式编程，以及为什么函数式编程是数据分析的有用工具。你将能快

速地学习如何使用现有工具，以及如何在需要的时候创建自己的函数。

- 欣赏元编程这把双刃剑。你将能在遵守原则的前提下使用非标准运算（non-standard evaluation）来创建函数，从而减少代码的输入量并创建优雅的代码来表达重要的运算。你将理解元编程的危险并知道为什么使用它时要小心。
- 对于R中什么样的操作会很慢且耗费内存能够产生很好的直觉。你将知道如何使用分析来找到阻碍性能提高的瓶颈，还会学到足够多的C++知识，让你可以将很慢的R函数转换成非常快的C++程序。
- 自如地阅读并理解大多数R代码。你将认识一些R的常用语（即使你自己不使用它们），并能够对其他人的代码做出评判。

1.3 元技术

有两种元技术对改善R程序员的技术非常有帮助：阅读源代码和采用科学的思维方式。

阅读源代码之所以重要是因为它可以帮助你写出更好的代码。开始培养这种技能的一种好方法就是查看你经常使用的函数和添加包的源代码。你会发现其中有很多值得效仿的地方，这样你就会知道什么可以使你的代码更好。你也会看到一些你不喜欢的东西，或许因为它的优点不够明显或者它让你感到不舒服。这样的代码也不是没有意义，它可以让你对好代码和坏代码有更具体的认识。

学习R时具有科学的思维方式是极有帮助的。如果你不知道有些事情是如何运行的，那就提出一个假设，设计一些实验，做实验并记录结果。这种训练是非常有用的，即使你不能解决这个问题并需要别人的帮助，但在寻求帮助时你可以告诉他们你已经尝试过哪些方法。另外，当你得到新的答案后，你也已经在思想上做好了准备来更新你的世界观。当我向别人描述问题时（构建一个可再现示例（http://stackoverflow.com/questions/5963269）的艺术），我都会指出自己的解决方案。

1.4 推荐阅读

R仍然是一门相对比较年轻的语言，可以帮助你理解它的资源还在不断地完善之中。在理解R的整个过程中，我发现使用其他编程语言的资源是非常有帮助的。R同时具有函数式编程和面向对象（OO）编程的特点。学会如何使用R来实现这些概念可以帮助你利用起你自己关于其他编程语言的知识，并让你发现（R）哪里还需要改进。

我发现Harold Abelson和Gerald Jay Sussman的《The Structure and Interpretation of Computer Programs》（http://mitpress.mit.edu/sicp/full-text/book/book.html）（SICP）一书对于理解R的对象系统为什么那样运行是非常有帮助的。这是一本简明但又深奥的书。读完本书之后，我第一次感觉到我可以设计出自己的面向对象系统。在本书中我第一次认识了R面向对象系统中常用的泛型函数（generic function）风格。它帮助我理解了R的优缺点。SICP同样讲述了很多函数式编程的知识，以及如何构建简单的函数，当把这些简单函数结合在一起时它又会变得非常强大。

要理解R相对于其他编程语言的优缺点，我发现Peter van Roy和Sef Haridi的《Concepts, Techniques and Models of Computer Programming》（http://amzn.com/0262220695?tag=devtools-20）非常有用。它让我知道了为什么R的copy-on-modify语义使得对代码的理解变

得如此简单，虽然现在实现起来还不是非常有效，但这个问题是可以解决的。

5 如果你想成为一个更好的程序员，没有哪本书比 Andrew Hunt 和 David Thomas 的《The Pragmatic Programmer》（http://amzn.com/020161622X?tag=devtools-20）更好了。这本书并不针对特定的语言，它对如何使你成为更好的程序员提供了很多建议。

1.5 获取帮助

目前当你遇到困难并且不知道如何解决它时，你有两个渠道可以获得帮助：stackoverflow（http://stackoverflow.com）和 R-help 邮件列表。在这两个地方你能得到有益的帮助，但是每个社区都有自己的文化和期望。通常情况下你最好先花一段时间潜水，在了解了该社区的文化和期望之后再提出你的第一个问题。

一些建议：

- 当你遇到问题时首先要确保你使用的是 R 的最新版本和添加包，很可能这些问题已经在最新版本中被修复了。
- 花一段时间构建一个可再现示例（http://stackoverflow.com/questions/5963269）。这通常是一个非常有用的过程，在再现问题的过程中你经常会找到问题的原因。
- 在求助前首先搜索一下相关问题。如果有人已经问过同样的问题并得到了回答，使用现有答案会更快速方便。

1.6 致谢

我要感谢 R-help 以及 stackoverflow（http://stackoverflow.com/questions/tagged/r）上不
6 知疲倦的贡献者。有太多人需要感谢了，尤其要感谢 Luke Tierney、John Chambers、Dirk Eddelbuettel、JJ Allaire 和 Brian Ripley 花费了大量的时间和精力为我纠正无数的错误。

本书是以开源方式编写的（https://github.com/hadley/adv-r），Twitter（https://twitter.com/hadleywickham）上的朋友对本书的章节给出了很多建议。这是社区的智慧结晶：很多人阅读了书稿，修改了错别字，对修改提出了建议，对本书的内容做出了贡献。没有这些人的帮助，本书不可能像现在这样给力，非常感谢他们的帮助。尤其要感谢 Peter Li，他从头到尾阅读了本书并修订了很多错误。其他非常给力的贡献者有：Aaron Schumacher、@crtahlin、Lingbing Feng、@juancentro 和 @johnbaums。

以字母顺序感谢所有的贡献者：Aaron Schumacher、Aaron Wolen、@aaronwolen、@absolutelyNoWarranty、Adam Hunt、@agrabovsky、@ajdm、@alexbbrown、@alko989、@allegretto、@AmeliaMN、@andrewla、Andy Teucher、Anthony Damico、Anton Antonov、@aranlunzer、@arilamstein、@avilella、@baptiste、@blindjesse、@blmoore、@bnjmn、Brandon Hurr、@BrianDiggs、@Bryce、C. Jason Liang、@Carson、@cdrv、Ching Boon、@chiphogg、Christopher Brown、@christophergandrud、Clay Ford、@cornelius1729、@cplouffe、Craig Citro、@crossfitAL、@crowding、CrtAhlin、@crtahlin、@cscheid、@csgillespie、@cusanovich、@cwarden、@cwickham、Daniel Lee、@darrkj、@Dasonk、David Hajage、David LeBauer、@dchudz、dennis feehan、@dfeehan、Dirk Eddelbuettel、@dkahle、@dlebauer、@dlschweizer、@dmontaner、@dougmitarotonda、@dpatschke、@duncandonutz、@EdFineOKL、@EDiLD、@eipi10、@elegrand、@EmilRehnberg、

Eric C. Anderson、@etb、@fabian-s、Facundo Muñoz、@flammy0530、@fpepin、Frank @Farach、@freezby、@fyears、Garrett Grolemund、@garrettgman、@gavinsimpson、@gggtest、GökçenEraslan、Gregg Whitworth、@gregorp、@gsee、@gsk3、@gthb、@hassaad85、@i、Iain Dillingham、@ijlyttle、Ilan Man、@imanuelcostigan、@initdch、Jason Asher、Jason Knight、@jasondavies、@jastingo、@jcborras、Jeff Allen、@jeharmse、@jentjr、@JestonBlu、@JimInNashville、@jinlong25、JJ Allaire、Jochen Van de Velde、Johann Hibschman、John Blischak、John Verzani、@johnbaums、@johnjosephhorton、Joris Muller、Joseph Casillas、@juancentro、@kdauria、@kenahoo、@kent37、Kevin Markham、Kevin Ushey、@kforner、Kirill Müller、Kun Ren、Laurent Gatto、@Lawrence-Liu、@ldfmrails、@lgatto、@liangcj、Lingbing Feng、@lynaghk、Maarten Kruijver、Mamoun Benghezal、@mannyishere、Matt Pettis、@mattbaggott、Matthew Grogan、@mattmalin、Michael Kane、@michaelbach、@mjsduncan、@Mullefa、@myqlarson、Nacho Caballero、Nick Carchedi、@nstjhp、@ogennadi、Oliver Keyes、@otepoti、Parker Abercrombie、@patperu、Patrick Miller、@pdb61、@pengyu、Peter F Schulam、Peter Lindbrook、Peter Meilstrup、@philchalmers、@picasa、@piccolbo、@pierreroudier、@pooryorick、R. Mark Sharp、Ramnath Vaidyanathan、@ramnathv、@Rappster、Ricardo Pietrobon、Richard Cotton、@richardreeve、@rmflight、@rmsharp、Robert M Flight、@RobertZK、@robiRagan、Romain François、@rrunner、@rubenfcasal、@sailingwave、@sarunasmerkliopas、@sbgraves237、Scott Ritchie、@scottko、@scottl、Sean Anderson、Sean Carmody、Sean Wilkinson、@sebastian-c、Sebastien Vigneau、@shabbychef、Shannon Rush、Simon O'Hanlon、Simon Potter、@SplashDance、@ste-fan、Stefan Widgren、@stephens999、Steven Pav、@strongh、@stuttungur、@surmann、@swnydick、@taekyunk、Tal Galili、@talgalili、@tdenes、@Thomas、@thomasherbig、@thomaszumbrunn、Tim Cole、@tjmahr、Tom Buckley、Tom Crockett、@ttriche、@twjacobs、@tyhenkaline、@tylerritchie、@ulrichatz、@varun729、@victorkryukov、@vijaybarve、@vzemlys、@wchi144、@wibeasley、@WilCrofter、William Doane、Winston Chang、@wmc3、@wordnerd、Yoni Ben-Meshulam、@zackham、@zerokarmaleft、Zhongpeng Lin。

1.7 约定

在本书中 `f()` 代表函数，`g` 代表变量和函数参数，`h/` 代表路径。

大的代码块包含输入和输出。输出已经被注释掉了，所以如果你有本书的电子版，例如，http://adv-r.had.co.nz，你就可以很简单将例子复制并粘贴到 R 中。为了与正常的注释相区别，将注释输出的注释符设定为 `#>`。

1.8 声明

本书是在 Rstudio（http://www.rstudio.com/ide/）中使用 Rmarkdown（http://rmarkdown.rstudio.com/）编写的。使用 knitr（http://yihui.name/knitr）和 pandoc（http://johnmacfarlane.net/pandoc/）将原始的 Rmarkdown 转换成 html 和 pdf 。本书的网站（http://adv-r.had.

co.nz）使用 jekyll（http://jekyllrb.com/）构建，使用 bootstrap 样式（http://getbootstrap.com/），并通过 travis-ci（https://travis-ci.org/）自动推送到 Amazon 的 S3（http://aws.amazon.com/s3）云服务器。本书的完整代码可以在 github（https://github.com/hadley/adv-r）获取。

8
~
10

第一部分

基础知识

第 2 章
数据结构

本章将对 R 基础包中最重要的数据结构进行总结。即使你以前没有使用过全部数据类型，你也可能使用过其中的大多数。然而，你可能没有深入地思考过它们之间的相互关系。本章不会对每个数据类型进行深入探讨，而会告诉你它们是如何协作从而构成一个整体的。如果想了解更多细节，可以到 R 的文档中查找。

可以根据数据的维度（1D、2D 或 *n*D）和它们是同质（必须由相同类型的数据组成）还是异质（可以由不同类型的数据组成）来对 R 的基础数据结构进行组织。这就产生了数据分析中最常用的 5 种数据类型：

	同 质	异 质
1D	原子向量	列表
2D	矩阵	数据框
*n*D	数组	

所有其他的对象都是在这些基础上构建的。在第 7 章中你将看到由这些简单的数据类型可以构造出更复杂的对象。需要注意的是：R 没有 0 维，或者标量类型。你可能会认为单独的数字或者字符串是标量，其实它们都是长度为 1 的向量。

11 ~ 13

给定一个对象，知道它是由什么数据结构组成的最好方法就是使用 `str()`。`str()` 是结构（structure）的简写，它给出任意类型的 R 数据结构的描述，这些描述既简洁又易于理解。

小测验

通过这个小测验可以帮助你决定是否需要阅读本章。如果你能够快速给出正确答案，那你就可以跳过本章。可以到 2.5 节中检查你的回答是否正确。

1）除了所包含的内容之外，向量的 3 个性质是什么？

2）原子向量的 4 种常见类型是什么？两种罕见类型是什么？

3）属性是什么？如何获取属性以及如何设置它们？

4）列表与原子向量有哪些不同？矩阵与数据框有哪些不同？

5）能由矩阵构成一个列表吗？数据框中的某一列能由矩阵组成吗？

主要内容

- 2.1 节介绍 R 的一维（1D）数据结构：原子向量和列表。

- 2.2 节讨论属性，它是 R 灵活的元数据规范。这里，你将学习通过设置原子向量的属性来构建 R 的重要数据结构：因子（factor）。
- 2.3 节介绍存储二维（2D）或高维数据的数据结构：矩阵和数组。
- 2.4 节介绍 R 中存储数据最重要的数据结构：数据框。数据框将列表和矩阵的行为结合到一起从而构成一个非常适合于数据统计的数据结构。

2.1 向量

R 中最基础的数据结构就是向量。向量包括两种风格：原子向量和列表。它们有 3 个共同性质属性：

- 类型，`typeof()`，它是什么。
- 长度，`length()`，它包含多少元素。
- 属性，`attributes()`，附加的任意元数据。

它们的区别就在于它们所包含元素的类型：原子向量中的所有元素必须是相同类型的数据，而列表中的元素可以是不同的类型。

14

注：不要使用 `is.vector()` 来判断一个对象是否是向量。只有在对象是向量且除了名字之外没有其他属性的情况下它才返回 `TRUE`。使用 `is.atomic(x) || is.list(x)` 来判断一个对象是否为向量。

2.1.1 原子向量

下面详细讨论 4 种常见类型的原子向量：逻辑型（logical）、整型（integer），双精度型（double）（通常又称为数值型（numeric））和字符型（character）。对于另外两种罕见类型：复合型（complex）和原始型（raw），这里不会进一步讨论。

通常使用 `c()` 来构建原子向量，它是联合（combine）的简写：

```
dbl_var <- c(1, 2.5, 4.5)
# With the L suffix, you get an integer rather than a double
int_var <- c(1L, 6L, 10L)
# Use TRUE and FALSE (or T and F) to create logical vectors
log_var <- c(TRUE, FALSE, T, F)
chr_var <- c("these are", "some strings")
```

原子向量总是被展开的，即便嵌套使用 `c()`：

```
c(1, c(2, c(3, 4)))
#> [1] 1 2 3 4
# the same as
c(1, 2, 3, 4)
#> [1] 1 2 3 4
```

缺失值用 `NA` 代表，它是一个长度为 1 的逻辑向量。如果在 `c()` 中使用 `NA`，`NA` 就会被强制转换成正确的数据类型，或者可以输入 `NA_real_`（双精度向量）、`NA_integer_` 和 `NA_character_` 来创建不同类型的 `NA`。

15

类型和测试

给定一个向量，可以使用 `typeof()` 来确定它的类型，或者使用“is”函数来判断它是否为指定的类型：`is.character()`，`is.double()`，`is.integer()`，`is.logical()`，或

者更一般地，is.atomic()。

```
int_var <- c(1L, 6L, 10L)
typeof(int_var)
#> [1] "integer"
is.integer(int_var)
#> [1] TRUE
is.atomic(int_var)
#> [1] TRUE

dbl_var <- c(1, 2.5, 4.5)
typeof(dbl_var)
#> [1] "double"
is.double(dbl_var)
#> [1] TRUE
is.atomic(dbl_var)
#> [1] TRUE
```

注：is.numeric()通常用来判断向量的“数值性”，无论是整型还是双精度型向量它都返回TRUE。虽然经常把双精度型（double）叫作数值型（numeric），但是is.numeric()并不是专门用来判断这种双精度型的。

```
is.numeric(int_var)
#> [1] TRUE
is.numeric(dbl_var)
#> [1] TRUE
```

强制转换

原子向量的所有元素必须具有相同的类型，所以当把不同类型的数据结合成一个向量时，它们会被**强制转换**（coerce）成最具灵活性的数据类型。数据类型的灵活性由低到高的排序为：逻辑型、整型、双精度型和字符型。

例如，将一个字符型数据和一个整型数据结合在一起时就产生一个字符型数据：

```
str(c("a", 1))
#>  chr [1:2] "a" "1"
```

16

当一个逻辑型向量被强制转换成整型或双精度型时，TRUE变成1，FALSE变成0。当sum()和mean()联合使用时，这会非常有用。

```
x <- c(FALSE, FALSE, TRUE)
as.numeric(x)
#> [1] 0 0 1

# Total number of TRUEs
sum(x)
#> [1] 1

# Proportion that are TRUE
mean(x)
#> [1] 0.3333
```

强制转换经常是自动进行的。大多数数学函数（+、log、abs等）会将数据转换成双精度型或整型。大多数逻辑运算符（&、|、any等）会将数据转换成逻辑型。如果强制转换会造成数据丢失，R会发出警告信息。如果有可能导致混淆，可以使用as.character()、as.double()、as.integer()或as.logical()来进行显式强制转换。

2.1.2 列表

列表与原子向量是不同的，因为列表中的元素可以是任意类型，甚至包括列表。使用`list()`而不是`c()`来构建列表：

```
x <- list(1:3, "a", c(TRUE, FALSE, TRUE), c(2.3, 5.9))
str(x)
#> List of 4
#>  $ : int [1:3] 1 2 3
#>  $ : chr "a"
#>  $ : logi [1:3] TRUE FALSE TRUE
#>  $ : num [1:2] 2.3 5.9
```

列表有时称为**递归**（recursive）向量，因为一个列表可以包含其他列表。这使它从根本上不同于原子向量。

```
x <- list(list(list(list())))
str(x)
#> List of 1
#>  $ :List of 1
#>   ..$ :List of 1
#>   .. ..$ : list()
is.recursive(x)
#> [1] TRUE
```

`c()`可以将多个列表结合成一个列表。如果将原子向量和列表结合在一起，`c()`会强制将向量转换成列表，然后再将它们结合到一起。下面比较`list()`和`c()`的结果：

```
x <- list(list(1, 2), c(3, 4))
y <- c(list(1, 2), c(3, 4))
str(x)
#> List of 2
#>  $ :List of 2
#>   ..$ : num 1
#>   ..$ : num 2
#>  $ : num [1:2] 3 4
str(y)
#> List of 4
#>  $ : num 1
#>  $ : num 2
#>  $ : num 3
#>  $ : num 4
```

对列表调用`typeof()`会得到`list`。可以使用`is.list()`来判断列表，使用`as.list()`强制转换为列表。使用`unlist()`将列表转换成原子向量。如果列表中存在不同类型的数据，`unlist()`会使用与`c()`一样的强制转换规则。

在R中经常使用列表来构建更复杂的数据结构。例如，数据框（data frame）（2.4节）和线性模型对象（使用`lm()`构建）都是列表：

```
is.list(mtcars)
#> [1] TRUE

mod <- lm(mpg ~ wt, data = mtcars)
is.list(mod)
#> [1] TRUE
```

2.1.3 练习

1. 6种类型的原子向量分别是什么？列表与原子向量的区别是什么？
2. 与 `is.list()` 和 `is.character()` 相比，`is.vector()` 和 `is.numeric()` 的根本区别是什么？
3. 根据你对向量强制转换规则的认识，对下列代码的结果进行预测：

```
c(1, FALSE)
c("a", 1)
c(list(1), "a")
c(TRUE, 1L)
```

4. 为什么将列表转换成原子向量要使用 `unlist()`？`as.vector()` 为什么不行？
5. 为什么 `1 == "1"` 返回的结果为 `TRUE`？为什么 `-1 < FALSE` 返回的结果也是 `TRUE`？为什么 `"one" < 2` 返回的结果为 `FALSE`？
6. 为什么默认的缺失值 `NA` 是逻辑型向量？逻辑型向量的特殊之处是什么？（提示：考虑 `c(FALSE, NA_character_)`。）

2.2 属性

所有的对象都可以通过任意附加的属性来存储对象的元数据。可以把属性作为一个命名列表（每一个名字都是独一无二的）。可以通过 `attr()` 单独访问对象的每一个属性，也可以使用 `attributes()` 同时访问所有的属性（以列表的形式）。

```
y <- 1:10
attr(y, "my_attribute") <- "This is a vector"
attr(y, "my_attribute")
#> [1] "This is a vector"
str(attributes(y))
#> List of 1
#>  $ my_attribute: chr "This is a vector"
```

19

`structure()` 函数可以返回一个属性被修改了的新对象：

```
structure(1:10, my_attribute = "This is a vector")
#>  [1]  1  2  3  4  5  6  7  8  9 10
#> attr(,"my_attribute")
#> [1] "This is a vector"
```

默认情况下，当向量被修改后它的大多数属性都会丢失：

```
attributes(y[1])
#> NULL
attributes(sum(y))
#> NULL
```

但3个最重要的属性不会丢失：

- 名字（name），给每个元素一个名字的字符向量，将在下面讨论。
- 维度（dimension），可以用来将向量转变成矩阵和数组，将在2.3节讨论。
- 类（class），用于实现S3对象系统，将在7.2节讨论。

这些属性中的每一个都使用特定的存取函数（accessor function）来获取和设置。当使用这些属性时，使用 `names(x)`、`class(x)` 和 `dim(x)`，而不是 `attr(x, "names")`、`attr(x, "class")` 和 `attr(x, "dim")`。

名字

可以用3种方式来命名向量中的元素：

- 创建时命名：`x <- c(a = 1, b = 2, c = 3)`。
- 修改现有向量的名字：`x <- 1:3; names(x) <- c("a", "b", "c")`。
- 复制一个向量并修改它的名字：`x <- setNames(1:3, c("a", "b", "c"))`。

虽然名字没有要求一定是唯一的，但3.4.1节讲到的字符子集选取告诉我们：名称是非常重要的，唯一性命名更是非常有用的。

并不是向量的每一元素都需要一个名字。如果有些元素没有名字，`names()`就为这些元素返回空字符串。如果所有元素都没有名字，`names()`就返回`NULL`。 [20]

```
y <- c(a = 1, 2, 3)
names(y)
#> [1] "a" ""  ""

z <- c(1, 2, 3)
names(z)
#> NULL
```

可以使用`unname(x)`来创建没有名字的新向量，或者使用`names(x) <- NULL`去掉向量中的名称。

2.2.1 因子

属性的一个重要应用就是定义因子。因子就是只能包含预先定义值的向量，它经常用来存储分类数据。因子建立在整型向量的基础之上，它具有两个属性：`class()`和`levels()`。`class()`使得因子与通常的整型向量具有不同的行为，而`levels()`定义因子中所有可能的取值。

```
x <- factor(c("a", "b", "b", "a"))
x
#> [1] a b b a
#> Levels: a b
class(x)
#> [1] "factor"
levels(x)
#> [1] "a" "b"

# You can't use values that are not in the levels
x[2] <- "c"
#> Warning: invalid factor level, NA generated
x
#> [1] a    <NA> b    a
#> Levels: a b

# NB: you can't combine factors
c(factor("a"), factor("b"))
#> [1] 1 1
```

即使你没有看到数据集中的所有值，当你知道一个变量所有可能的取值时，因子也是非常有用的。使用因子而不使用字符向量可以使一些没有观测值的组看起来也很清楚（虽然没有观测值，但可以根据因子的`levels()`属性来获知它可能的取值）。 [21]

```
sex_char <- c("m", "m", "m")
sex_factor <- factor(sex_char, levels = c("m", "f"))

table(sex_char)
#> sex_char
#> m
#> 3
table(sex_factor)
#> sex_factor
#> m f
#> 3 0
```

有时，当从文件中直接读取一个数据框时，你可能发现本来应该是数值向量类型的一列数据变成了因子类型。这可能是由这一列中的非数值型数据造成的，通常是用特殊方式编码的缺失值，例如，.或 -。解决这个问题的方法是：将向量由因子型强制转换成字符型，再由字符型转换成双精度型（在完成这些步骤之后一定要记得对缺失值进行检查）。当然，更好的方法是找到问题的原因并解决它。在调用 `read.csv()` 函数时，对 `na.strings` 参数进行设置通常是一个好的方法。

```
# Reading in "text" instead of from a file here:
z <- read.csv(text = "value\n12\n1\n.\n9")
typeof(z$value)
#> [1] "integer"
as.double(z$value)
#> [1] 3 2 1 4
# Oops, that's not right: 3 2 1 4 are the levels of a factor,
# not the values we read in!
class(z$value)
#> [1] "factor"
# We can fix it now:
as.double(as.character(z$value))
#> Warning: NAs introduced by coercion
#> [1] 12  1 NA  9
# Or change how we read it in:
z <- read.csv(text = "value\n12\n1\n.\n9", na.strings=".")
typeof(z$value)
#> [1] "integer"
class(z$value)
#> [1] "integer"
z$value
#> [1] 12  1 NA  9
# Perfect! :)
```

但是，R 中的大多数数据加载函数自动将字符型向量转换成因子型。这不是最佳的，因为这些函数不可能知道这个数据集所有可能的取值和它们的最优顺序。相反，可以使用参数 `stringsAsFactors = FALSE` 来制止这种情况的发生，然后根据你对这些数据的认识手动将字符型向量转换成因子型。也可以使用全局选项 `options(stringsAsFactors = FALSE)` 来控制这种行为，但我不推荐这种做法。当与其他代码（可能来自于其他包或者你自己正在编写的代码）联合使用时，改变全局选项可能会造成无法预知的后果。修改全局选项会使代码变得很难理解，因为为了弄明白这种修改到底会产生什么样的后果你必须要阅读更多的代码。

虽然因子看上去（以及行为上）很像字符型向量，但它其实是整型。当把因子看作字符串处理时一定要多加小心。有些处理字符串的方法（比如 `gsub()` 和 `grepl()`）会将因子强

制转换成字符串，而有些方法（比如 nchar()）则会抛出错误，还有些（比如 c()）会使用其潜在的整型数值。由于这些原因，如果需要类似字符串的行为，通常最好显式地将因子转换成字符向量。在 R 的早期版本中，使用因子比使用字符型向量有内存方面的优势，但现在已经不是这样了。

2.2.2 练习

1. 以前我曾使用下面的代码来演示 structure() 的用法：

```
structure(1:5, comment = "my attribute")
#> [1] 1 2 3 4 5
```

但是，当输出这个对象时并不会看到 comment 属性。这是为什么呢？这个属性丢了吗？这里还有什么特别之处吗？（提示：使用帮助文档。）

2. 当修改了因子的水平时，会对因子产生什么样的影响？

```
f1 <- factor(letters)
levels(f1) <- rev(levels(f1))
```

22 ~ 23

3. 这段代码的作用是什么？ f2 和 f3 与 f1 有什么不同？

```
f2 <- rev(factor(letters))

f3 <- factor(letters, levels = rev(letters))
```

2.3 矩阵和数组

给一个原子向量添加 dim() 属性之后，它的行为就像多维**数组**（array）了。数组的一个特例就是**矩阵**（matrix），它是二维的。矩阵是统计学中最常用的数学工具。数组用得不多，但是值得注意。

```
# Two scalar arguments to specify rows and columns
a <- matrix(1:6, ncol = 3, nrow = 2)
# One vector argument to describe all dimensions
b <- array(1:12, c(2, 3, 2))

# You can also modify an object in place by setting dim()
c <- 1:6
dim(c) <- c(3, 2)
c
#>      [,1] [,2]
#> [1,]    1    4
#> [2,]    2    5
#> [3,]    3    6
dim(c) <- c(2, 3)
c
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
```

length() 和 names() 具有高维普适性（二维矩阵或高维数组都可以使用）。

- 对矩阵来说，length() 返回矩阵的行数和列数（相当于 nrow() 和 ncol()）；对于数组，返回数组的维度（相当于 dim()）。
- 对矩阵来说，names() 相当于 rownames() 和 colnames()；对数组来说，相当于 dimnames()（会返回一个字符型向量）。

24

```
length(a)
#> [1] 6
nrow(a)
#> [1] 2
ncol(a)
#> [1] 3
rownames(a) <- c("A", "B")
colnames(a) <- c("a", "b", "c")
a
#>   a b c
#> A 1 3 5
#> B 2 4 6

length(b)
#> [1] 12
dim(b)
#> [1] 2 3 2
dimnames(b) <- list(c("one", "two"), c("a", "b", "c"), c("A", "B"))
b
#> , , A
#>
#>     a b c
#> one 1 3 5
#> two 2 4 6
#>
#> , , B
#>
#>     a  b  c
#> one 7  9 11
#> two 8 10 12
```

对于矩阵，`c()` 相当于 `cbind()` 和 `rbind()`；对于数组，`c()` 相当于 `abind()`（由添加包 `abind` 提供）。可以使用 `t()` 对矩阵进行转置；对于数组使用 `aperm()`。

可以使用 `is.matrix()` 和 `is.array()` 来判断一个对象是矩阵还是向量，使用 `dim()` 来查看长度。使用 `as.matrix()` 和 `as.array()` 可以轻松地将现有向量转换成矩阵或数组。

25

向量不是唯一的一维数据结构。可以创建只有一行或一列的矩阵，或仅有一维的数组。它们看起来可能很像，但是行为是不同的。虽然这些差别并不是非常重要，但是当你使用某个函数（比如 `tapply()`）得到异常输出时，知道这些差别对于你解决问题还是很有用的。

```
str(1:3)                   # 1d vector
#>  int [1:3] 1 2 3
str(matrix(1:3, ncol = 1)) # column vector
#>  int [1:3, 1] 1 2 3
str(matrix(1:3, nrow = 1)) # row vector
#>  int [1, 1:3] 1 2 3
str(array(1:3, 3))         # "array" vector
#>  int [1:3(1d)] 1 2 3
```

虽然原子向量经常转换成矩阵，但还可以通过设置列表的维度属性构建列表矩阵和列表数组：

```
l <- list(1:3, "a", TRUE, 1.0)
dim(l) <- c(2, 2)
```

```
1
#>      [,1]      [,2]
#> [1,] Integer,3 TRUE
#> [2,] "a"       1
```

这些都是相对复杂的数据结构，但是如果希望将对象按照网格状（grid-like）的结构存储，它们非常有用。例如，如果你正在一个时空网格中运行你的模型，那么将模型以三维数组的格式保存对于保护网格结构来说是很重要的。

练习

1. 对向量使用 `dim()` 会返回什么？
2. 如果 `is.matrix(x)` 的返回值为 `TRUE`，那么 `is.array(x)` 的返回值是什么？
3. 下面的 3 个对象应该如何描述？它们与 `1:5` 的差别是什么？

```
x1 <- array(1:5, c(1, 1, 5))
x2 <- array(1:5, c(1, 5, 1))
x3 <- array(1:5, c(5, 1, 1))
```

26

2.4　数据框

在 R 中存储数据使用最多的应该就是数据框了，如果能够系统地使用它（http://vita.had.co.nz/papers/tidy-data.pdf），数据分析就会变得更加容易。实质上，数据框是由相同长度的向量构成的列表。这就让数据框变成了二维结构，所以它既具有矩阵的性质又具有列表的性质。这就意味着可以对数据框使用函数 `names()`、`colnames()` 和 `rownames()`，尽管此时 `names()` 和 `colnames()` 的返回值是相同的。对数据框使用函数 `length()`，可以获得这个基础列表的长度，其实它与函数 `ncol()` 的返回值是一样的；而 `nrow()` 返回行数。

如第 3 章所述，可以将数据框分割成一维结构（这时就像列表），或者二维结构（此时就像矩阵）。

2.4.1　数据框构建

可以使用函数 `data.frame()` 来构建数据框，它的输入是已经命名的向量。

```
df <- data.frame(x = 1:3, y = c("a", "b", "c"))
str(df)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y: Factor w/ 3 levels "a","b","c": 1 2 3
```

需要注意的是，`data.frame()` 的默认行为是把字符串转换成因子。使用参数 `stringAsFactors = FALSE` 来禁止这种转换。

```
df <- data.frame(
  x = 1:3,
  y = c("a", "b", "c"),
  stringsAsFactors = FALSE)
str(df)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y: chr  "a" "b" "c"
```

27

2.4.2 类型判断与强制转换

由于 data.frame 是 S3 类，所以它的类型反映了构建它的基础向量：列表。要判断一个对象是不是数据框，可以使用 class()，或者直接使用 is.data.frame()：

```
typeof(df)
#> [1] "list"
class(df)
#> [1] "data.frame"
is.data.frame(df)
#> [1] TRUE
```

可以使用 as.data.frame() 将对象强制转换成数据框：

- 向量将创建一列的数据框。
- 列表将为每个元素创建一列。如果列表元素的长度不相等，就会报错。
- 矩阵创建一个具有相同行数和列数的数据框。

2.4.3 合并数据框

可以使用 cbind() 和 rbind() 对数据框进行合并：

```
cbind(df, data.frame(z = 3:1))
#>   x y z
#> 1 1 a 3
#> 2 2 b 2
#> 3 3 c 1
rbind(df, data.frame(x = 10, y = "z"))
#>    x y
#> 1  1 a
#> 2  2 b
#> 3  3 c
#> 4 10 z
```

当进行列向（column-wise）合并时，两个数据框的行数必须保持一致，行的名字可以忽略。当进行行向 (row-wise) 合并时，列的名字和列数都必须一致。如果两个数据框没有相同列，可以使用 plyr::rbind.fill() 来进行合并。

28

经常犯的一个错误就是使用 cbind() 将向量合并到一起来创建数据框。这是没用的，因为 cbind() 将创建一个矩阵，除非其中有一个参数本身就是数据框。另外，可以直接使用 data.frame()：

```
bad <- data.frame(cbind(a = 1:2, b = c("a", "b")))
str(bad)
#> 'data.frame':    2 obs. of  2 variables:
#>  $ a: Factor w/ 2 levels "1","2": 1 2
#>  $ b: Factor w/ 2 levels "a","b": 1 2
good <- data.frame(a = 1:2, b = c("a", "b"),
  stringsAsFactors = FALSE)
str(good)
#> 'data.frame':    2 obs. of  2 variables:
#>  $ a: int  1 2
#>  $ b: chr  "a" "b"
```

cbind() 的转换规则是非常复杂的，最好通过保证所有输入都是同一类型来避免这种情况发生。

2.4.4 特殊列

由于数据框是一个向量列表，所以数据框也可能有一列是由列表构成的：

```
df <- data.frame(x = 1:3)
df$y <- list(1:2, 1:3, 1:4)
df
#>   x          y
#> 1 1       1, 2
#> 2 2    1, 2, 3
#> 3 3 1, 2, 3, 4
```

但是，当对列表使用 `data.frame()` 时，它会将列表中的每个元素都放入自己的列中，所以就会出错：

```
data.frame(x = 1:3, y = list(1:2, 1:3, 1:4))
#> Error: arguments imply differing number of rows: 2, 3, 4
```

29

使用 `I()` 可以避免这种错误，它使 `data.frame()` 把列表看作一个单元：

```
dfl <- data.frame(x = 1:3, y = I(list(1:2, 1:3, 1:4)))
str(dfl)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y:List of 3
#>   ..$ : int  1 2
#>   ..$ : int  1 2 3
#>   ..$ : int  1 2 3 4
#>   ..- attr(*, "class")= chr "AsIs"
dfl[2, "y"]
#> [[1]]
#> [1] 1 2 3
```

`I()` 可以给它的输入加上 `AsIs` 类，这一般是没有问题的，通常可以放心地忽略。

同样，数据框中的某一列也可以是矩阵或数组，只要行数与数据框的行数相匹配：

```
dfm <- data.frame(x = 1:3, y = I(matrix(1:9, nrow = 3)))
str(dfm)
#> 'data.frame':    3 obs. of  2 variables:
#>  $ x: int  1 2 3
#>  $ y: 'AsIs' int [1:3, 1:3] 1 2 3 4 5 6 7 8 9
dfm[2, "y"]
#>      [,1] [,2] [,3]
#> [1,]    2    5    8
```

使用列表和数组的列时一定要小心：处理数据框的很多函数都假设所有的列都是原子向量。

30

2.4.5 练习

1. 数据框拥有的属性是什么？
2. 当数据框中的不同列具有不同的数据类型时，对其使用 `as.matrix()` 会发生什么？
3. 可以有 0 行的数据框吗？可以有 0 列的数据框吗？

2.5 答案

1）向量的 3 个性质是：类型、长度和属性。

2）原子向量的4种常见类型为：逻辑型、整型、双精度型（有时称为数值型）和字符型。两种罕见类型为：复杂型和原始型。

3）可以使用属性给任意对象添加任意的附加元数据。可以使用`attr(x, "y")`获取对象属性，使用`attr(x, "y") <- value`设置对象属性，还可以使用`attributes()`同时获取和设置所有的属性。

4）列表的元素可以是任意类型的数据（甚至可以是列表）；而原子向量的所有元素必须是相同的类型。同样，矩阵的每个元素也必须是相同的类型；数据框中的不同列可以有不同的数据类型（但同一列必须是相同的类型）。

5）为列表设置维度就可以构建“列表数组”。使用`df$x <- matrix()`，或者在创建新数据框时使用`data.frame(x = I(matrix()))`，可以将矩阵作为一列添加到数据框中。

31 ~ 32

第3章
子集选取

R有强大而又快速的子集选取运算符。掌握子集选取技术可以让你用非常简明的语句来对数据进行复杂操作，这是其他编程语言无法比拟的。由于需要掌握一系列相关概念，所以子集选取比较难学：

- 3个子集选取运算符。
- 子集选取的6种不同方法。
- 对于不同对象（例如，向量、列表、因子、矩阵和数据框），子集选取有很大的不同。
- 将赋值与子集选取相结合。

本章从最简单的子集选取开始来帮助你掌握它：使用`[`对一个原子向量选取子集。接着我会慢慢展开，首先介绍对更复杂的数据类型（例如，数组和列表）进行操作，然后介绍其他选取子集运算符：`[[`和`$`。其次，你还会学习通过联合使用赋值和子集选取来对一个对象的各个部分进行修改。最后，你将看到子集选取大量的应用。

子集选取是对`str()`的天然补充。`str()`说明一个对象的结构，而子集选取可以让你获取其中你最感兴趣的部分数据。

小测验

你可以通过这个小测验来决定是否需要阅读本章内容。如果你能够快速地给出正确答案，那么你就可以跳过本章。你可以到3.5节中去检查你的回答是否正确。

1）对一个向量进行子集选取时，分别使用正整数、负整数、逻辑向量或字符向量作为参数会得到什么结果？

2）对列表分别使用`[`、`[[`和`$`会有什么不同？

3）什么时候需要使用`drop = FALSE`？

4）如果`x`是矩阵，执行`x[] <- 0`后会有什么结果？这与`x <- 0`有什么不同？

5）如何使用一个命名向量来对分类变量设置标签。

主要内容

- 3.1节从学习`[`开始。你将学习使用6种类型的数据分别对原子向量进行子集选取。接着学习如何把这6种不同类型的数据应用到列表、矩阵、数据框和S3对象的子集选取中。
- 3.2节对知识进行扩展：学习`[[`和`$`。焦点集中在最重要的原则上：简化与保留。
- 3.3节你将学习联合使用子集选取和赋值来对对象的子集进行修改，从而实现对子集的赋值。

- 3.4 节学习子集选取的 8 种虽然不明显但非常重要的应用，这可以帮你解决数据分析中经常遇到的问题。

3.1 数据类型

首先学习如何对原子向量进行子集选取，然后学习如何把这些原则推广到高维度或者更复杂的对象，这是学习子集选取的最简单方法。我们将从最常用的运算符 [开始。3.2 节将介绍另外两种主要的子集选取运算符 [[和 $。

3.1.1 原子向量

现在通过一个简单向量 x 来探索不同类型的子集选取方法。

34

```
x <- c(2.1, 4.2, 3.3, 5.4)
```

注意，小数点后面的数字代表该元素在向量中的原始位置。

可以使用 5 种方法来为一个向量选取子集。

- **正整数**返回指定位置上的元素：

```
x[c(3, 1)]
#> [1] 3.3 2.1
x[order(x)]
#> [1] 2.1 3.3 4.2 5.4

# Duplicated indices yield duplicated values
x[c(1, 1)]
#> [1] 2.1 2.1

# Real numbers are silently truncated to integers
x[c(2.1, 2.9)]
#> [1] 4.2 4.2
```

- **负整数**不包含指定位置上的元素：

```
x[-c(3, 1)]
#> [1] 4.2 5.4
```

在同一个子集选取操作中，不能同时使用正整数和负整数：

```
x[c(-1, 2)]
#> Error: only 0's may be mixed with negative subscripts
```

- **逻辑向量**只选择对应于逻辑向量的相应位置为 TRUE 的元素。这可能是子集选取最有用的方法，因为你可以自己写表达式来创建逻辑向量：

```
x[c(TRUE, TRUE, FALSE, FALSE)]
#> [1] 2.1 4.2
x[x > 3]
#> [1] 4.2 3.3 5.4
```

如果逻辑向量比要从中选取子集的向量短，那么逻辑向量就会被循环（recycled）使用直到与对应的向量有相同的长度。

```
x[c(TRUE, FALSE)]
#> [1] 2.1 3.3
# Equivalent to
x[c(TRUE, FALSE, TRUE, FALSE)]
#> [1] 2.1 3.3
```

35

索引中有缺失值总是在输出结果的相应位置产生缺失值：

```
x[c(TRUE, TRUE, NA, FALSE)]
#> [1] 2.1 4.2  NA
```

- **空索引**（Nothing）返回原始向量。虽然这对于向量来说没什么用，但它对矩阵、数据框和数组很有用。它与赋值操作联合使用也是很有用的。

```
x[]
#> [1] 2.1 4.2 3.3 5.4
```

- 0（Zero）返回长度为0的向量。平时你可能不会这样做，但是这对于创建测试数据来说很有帮助。

```
x[0]
#> numeric(0)
```

如果向量中的元素都有名字，也可以使用它们的名字作为索引来选取子集。

- **字符向量**返回与索引中的名字相匹配的元素。

```
(y <- setNames(x, letters[1:4]))
#>   a   b   c   d
#> 2.1 4.2 3.3 5.4
y[c("d", "c", "a")]
#>   d   c   a
#> 5.4 3.3 2.1

# Like integer indices, you can repeat indices
y[c("a", "a", "a")]
#>   a   a   a
#> 2.1 2.1 2.1

# When subsetting with [ names are always matched exactly
z <- c(abc = 1, def = 2)
z[c("a", "d")]
#> <NA> <NA>
#>   NA   NA
```

36

3.1.2 列表

列表的子集选取与原子向量的子集选择相同。使用 `[` 总是返回列表。下面要讲到的 `[[` 和 `$`，让你将列表中的元素取出。

3.1.3 矩阵和数组

可以使用3种方法从高维数据结构中选取子集：

- 使用多个向量。
- 使用单个向量。
- 使用矩阵。

对矩阵（二维）和数组（大于二维）进行子集选取的最常用方法是对一维子集选取方法进行泛化：为每一维数据给出一个一维索引，用逗号隔开。如果某一维的索引为空，说明要保留这一维的所有数据。

```
a <- matrix(1:9, nrow = 3)
colnames(a) <- c("A", "B", "C")
a[1:2, ]
```

```
#>   A B C
#> [1,] 1 4 7
#> [2,] 2 5 8
a[c(T, F, T), c("B", "A")]
#>   B A
#> [1,] 4 1
#> [2,] 6 3
a[0, -2]
#>   A C
```

默认情况下，[将结果简化到尽可能低的维度。3.1 节将告诉你如何避免这种情况发生。

由于矩阵和数组是以具有特殊属性的向量来实现的，所以可以使用一个单一的向量来对它们进行子集选取。在这种情况下，它们看起来很像向量。R 中的数组是以列优先的顺序存储的。

```
(vals <- outer(1:5, 1:5, FUN = "paste", sep = ","))
#>      [,1]  [,2]  [,3]  [,4]  [,5]
#> [1,] "1,1" "1,2" "1,3" "1,4" "1,5"
#> [2,] "2,1" "2,2" "2,3" "2,4" "2,5"
#> [3,] "3,1" "3,2" "3,3" "3,4" "3,5"
#> [4,] "4,1" "4,2" "4,3" "4,4" "4,5"
#> [5,] "5,1" "5,2" "5,3" "5,4" "5,5"
vals[c(4, 15)]
#> [1] "4,1" "5,3"
```

还可以使用整数矩阵来对更高维的数据进行子集选取（如果元素是命名的，还可以使用字符矩阵）。矩阵的每一行给出要选取元素的位置，矩阵的每一列对应于要选取子集的数组的维度。这说明你可以使用一个 2 列矩阵从一个矩阵中选取子集，使用一个 3 列矩阵从 3D 数组中选取子集，等等。返回的结果是由选取的值构成的向量：

```
vals <- outer(1:5, 1:5, FUN = "paste", sep = ",")
select <- matrix(ncol = 2, byrow = TRUE, c(
  1, 1,
  3, 1,
  2, 4
))
vals[select]
#> [1] "1,1" "3,1" "2,4"
```

3.1.4 数据框

数据框兼具列表和矩阵的特点：如果仅仅使用一个单一的向量来选取数据框的子集，则数据框的行为就像列表；如果使用两个向量来对数据框选取子集，则数据框的行为就像矩阵。

```
df <- data.frame(x = 1:3, y = 3:1, z = letters[1:3])

df[df$x == 2, ]
#>   x y z
#> 2 2 2 b
df[c(1, 3), ]
#>   x y z
#> 1 1 3 a
#> 3 3 1 c

# There are two ways to select columns from a data frame
```

```
# Like a list:
df[c("x", "z")]
#>   x z
#> 1 1 a
#> 2 2 b
#> 3 3 c
# Like a matrix
df[, c("x", "z")]
#>   x z
#> 1 1 a
#> 2 2 b
#> 3 3 c

# There's an important difference if you select a single
# column: matrix subsetting simplifies by default, list
# subsetting does not.
str(df["x"])
#> 'data.frame':    3 obs. of  1 variable:
#>  $ x: int  1 2 3
str(df[, "x"])
#>  int [1:3] 1 2 3
```

3.1.5 S3 对象

S3 对象是由原子向量、数组和列表构成的，所以你可以使用上面的技术和通过 `str()` 函数获得信息来对 S3 对象进行子集选取。

3.1.6 S4 对象

如果要从 S4 对象选取子集，还需要另外两个运算符：`@`（等价于 `$`）和 `slot()`（等价于 `[[`）。`@` 比 `$` 更严格，因为如果相应的字段（slot）不存在，`@` 将返回一个错误。这些将在 7.3 节中详细地描述。

3.1.7 练习

37 ~ 39

1. 对下面这些在数据框子集选取过程中常犯的错误进行修改：

```
mtcars[mtcars$cyl = 4, ]
mtcars[-1:4, ]
mtcars[mtcars$cyl <= 5]
mtcars[mtcars$cyl == 4 | 6, ]
```

2. 为什么 `x <- 1:5; x[NA]` 返回 5 个缺失值？（提示：它与 `x[NA_real_]` 有什么不同？）
3. `upper.tri()` 的返回值是什么？使用它对矩阵进行子集选取会怎样？还需要其他子集选取规则来描述这种行为吗？

```
x <- outer(1:5, 1:5, FUN = "*")
x[upper.tri(x)]
```

4. 为什么 `mtcars[1:20]` 会返回错误？它与 `mtcars[1:20,]` 有什么不同？
5. 自己构建一个函数来提取矩阵中的对角元素（类似于 `diag(x)`，其中 `x` 为矩阵）。
6. `df[is.na(df)] <- 0` 会做什么？它是如何工作的？

3.2 子集选取运算符

还有另外两个子集选取运算符：[[和 $。[[和 [类似，但它只能返回一个值，并可以让你取出列表的内容。$ 运算符与选定的字符串一起就成为 [[的简写。

当处理列表时应该使用 [[。这是因为当 [应用于列表时，它总是返回一个列表：它永远不会返回列表的内容。要获取内容，就需要使用 [[：

> 如果把列表 x 看成装载对象（货物）的火车，那 x[[5]] 就是 5 号车厢中的对象（货物）；而 x[4:6] 就是火车的 4～6 号车厢。
>
> ——@RLangTip

40 由于 [[只返回一个值，所以它必须与一个正整数或一个字符串一起使用：

```
a <- list(a = 1, b = 2)
a[[1]]
#> [1] 1
a[["a"]]
#> [1] 1

# If you do supply a vector it indexes recursively
b <- list(a = list(b = list(c = list(d = 1))))
b[[c("a", "b", "c", "d")]]
#> [1] 1
# Same as
b[["a"]][["b"]][["c"]][["d"]]
#> [1] 1
```

由于数据框是由列构成的列表，所以可以使用 [[提取一列：mtcars[[1]]，mtcars[["cyl"]]。

S3 和 S4 对象可以重写 [和 [[的标准行为，因此对不同类型的对象它们有不同的行为。关键的不同通常体现在如何在简化或者保留行为之间进行选择，或者默认的行为是什么。

3.2.1 简化与保留

在子集选取时，我们一定要注意最后的子集是不是还保持着原来的数据结构，这一点非常重要。对子集进行简化使得到的子集保持尽量最简单的数据结构，这对于交互式数据分析很有用，因为通常返回值都是你想要的。而使选取的子集保持原有的数据结构对于编程更好，因为结果总是相同的类型。在对矩阵和数据框进行子集选取时，经常省略 drop = FALSE，这是最常见的编程错误的来源。（在测试时它是没问题的，但如果有人给它输入数据框的一列数据，它就会莫名其妙地报错。）

不幸的是，对于不同的数据类型，如何在简化和保留之间进行转换也是不同的，总结如下。

	简　化	保　留
向量	x[[1]]	x[1]
列表	x[[1]]	x[1]
因子	x[1:4, drop = T]	x[1:4]
数组	x[1,] 或 x[, 1]	x[1, , drop = F] 或 x[, 1, drop = F]
数据框	x[, 1] 或 x[[1]]	x[, 1, drop = F] 或 x[1]

对于所有的数据类型，保留是相同的：可以得到与输入相同类型的输出。不同类型数据之间的简化却有些不同，如下所述。

- **原子向量**：去除名字。

```
x <- c(a = 1, b = 2)
x[1]
#> a
#> 1
x[[1]]
#> [1] 1
```

- **列表**：返回列表中的对象，而不是一个元素列表。

```
y <- list(a = 1, b = 2)
str(y[1])
#> List of 1
#>  $ a: num 1
str(y[[1]])
#>  num 1
```

- **因子**：扔掉所有不用的水平。

```
z <- factor(c("a", "b"))
z[1]
#> [1] a
#> Levels: a b
z[1, drop = TRUE]
#> [1] a
#> Levels: a
```

41
~
42

- **矩阵或者数组**：抛弃所有长度为 1 的维度。

```
a <- matrix(1:4, nrow = 2)
a[1, , drop = FALSE]
#>      [,1] [,2]
#> [1,]    1    3
a[1, ]
#> [1] 1 3
```

- **数据框**：如果输出仅有一列，返回一个向量而不是数据框。

```
df <- data.frame(a = 1:2, b = 1:2)
str(df[1])
#> 'data.frame':    2 obs. of  1 variable:
#>  $ a: int  1 2
str(df[[1]])
#>  int [1:2] 1 2
str(df[, "a", drop = FALSE])
#> 'data.frame':    2 obs. of  1 variable:
#>  $ a: int  1 2
str(df[, "a"])
#>  int [1:2] 1 2
```

3.2.2 $

`$` 是一个简写运算符，这里 `x$y` 等价于 `x[["y", exact = FALSE]]`。它经常用来访问数据框中的变量。例如，`mtcars$cyl` 或 `diamonds$carat`。

使用 `$` 最常见的一个错误是：如果你知道一个数据框中某一列的名字，但这个名字存储在一个变量中，如果在 `$` 后边不直接引用列的名字，而是使用变量就会出错：

```
var <- "cyl"
# Doesn't work - mtcars$var translated to mtcars[["var"]]
mtcars$var
#> NULL

# Instead use [[
mtcars[[var]]
#>  [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8
#> [30] 6 8 4
```

43 $ 与 [[之间有一点非常重要的不同。$ 是部分匹配：

```
x <- list(abc = 1)
x$a
#> [1] 1
x[["a"]]
#> NULL
```

如果你想禁止这种行为，可以将全局变量 warnPartialMatchDollar 设置为 TRUE。使用时需要注意：这样做可能会影响其他代码的行为（例如，你已经加载的添加包）。

3.2.3 缺失 / 超出索引边界（越界引用）

当索引越界时，[和 [[的行为略有不同。例如，当提取一个长度为 4 的向量的第 5 个元素时，或者用 NA 或者 NULL 为索引对向量进行子集选取时：

```
x <- 1:4
str(x[5])
#>  int NA
str(x[NA_real_])
#>  int NA
str(x[NULL])
#>  int(0)
```

下面这张表总结了使用 [和 [[以及不同类型的越界值对原子向量和列表进行子集选取的结果。

运 算 符	索 引	原子向量	列 表
[	OOB	NA	list(NULL)
[	NA_real_	NA	list(NULL)
[	NULL	x[0]	list(NULL)
[[	OOB	错误	错误
[[	NA_real_	错误	NULL
[[	NULL	错误	错误

如果输入向量中的每个元素都有名字，那么越界值、缺失值或者 NULL 元素的名字显示
44 为 "<NA>"。

3.2.4 练习

假设有一个线性模型，例如，mod <- lm(mpg~wt, data = mtcars)，提取它的残差自由度。从这个模型汇总（summary(mod)）中提取 R^2。

3.3 子集选取与赋值

所有的子集选取运算符可以和赋值结合在一起使用，从而修改输入向量的选定的值。

```
x <- 1:5
x[c(1, 2)] <- 2:3
x
#> [1] 2 3 3 4 5

# The length of the LHS needs to match the RHS
x[-1] <- 4:1
x
#> [1] 2 4 3 2 1

# Note that there's no checking for duplicate indices
x[c(1, 1)] <- 2:3
x
#> [1] 3 4 3 2 1

# You can't combine integer indices with NA
x[c(1, NA)] <- c(1, 2)
#> Error: NAs are not allowed in subscripted assignments
# But you can combine logical indices with NA
# (where they're treated as false).
x[c(T, F, NA)] <- 1
x
#> [1] 1 4 3 1 1

# This is mostly useful when conditionally modifying vectors
df <- data.frame(a = c(1, 10, NA))
df$a[df$a < 5] <- 0
df$a
#> [1]  0 10 NA
```

子集选取时使用空引用再结合赋值操作会比较有用，因为它会保持原有的对象类和数据结构。对比下面两个表达式。第一个，`mtcars` 仍然是数据框。第二个，`mtcars` 就会变成列表。

```
mtcars[] <- lapply(mtcars, as.integer)
mtcars <- lapply(mtcars, as.integer)
```

45

对于列表，可以使用子集选取＋赋值＋`NULL` 来去除列表中元素。使用 `[` 和 `list(NULL)` 在列表中添加合法的 `NULL`。

```
x <- list(a = 1, b = 2)
x[["b"]] <- NULL
str(x)
#> List of 1
#>  $ a: num 1

y <- list(a = 1)
y["b"] <- list(NULL)
str(y)
#> List of 2
#>  $ a: num 1
#>  $ b: NULL
```

3.4 应用

上面描述的这些基本规则为子集选取提供了广泛的应用。下面讲述最重要的几个应用。在这些基本的技术中有很多已经被打包成更简洁的函数（比如，`subset()`、`merge()`、`plyr::arrange()`），但是知道如何通过最基础的子集选取来实现它们也是非常有帮助的。当现有函数不能满足需求时，这些知识可以帮助你解决问题。

3.4.1 查询表（字符子集选取）

46

字符匹配为创建查询表提供了一个强大的方法。例如，你想将简写转换成全拼：

```
x <- c("m", "f", "u", "f", "f", "m", "m")
lookup <- c(m = "Male", f = "Female", u = NA)
lookup[x]
#>        m        f        u        f        f        m
#>   "Male" "Female"       NA "Female" "Female"   "Male"
#>        m
#>   "Male"
unname(lookup[x])
#> [1] "Male"   "Female" NA       "Female" "Female" "Male"
#> [7] "Male"

# Or with fewer output values
c(m = "Known", f = "Known", u = "Unknown")[x]
#>         m         f         u         f         f         m
#>   "Known"   "Known" "Unknown"   "Known"   "Known"   "Known"
#>         m
#>   "Known"
```

如果不想在结果中显示名字，可以使用`unname()`去掉它。

3.4.2 人工比对与合并（整数子集选取）

可以应用具有多个信息列的复杂查询表。假设有一个成绩向量，其取值为整数；还有一个表来描述整数成绩的性质。

```
grades <- c(1, 2, 2, 3, 1)

info <- data.frame(
  grade = 3:1,
  desc = c("Excellent", "Good", "Poor"),
  fail = c(F, F, T)
)
```

从`info`表中查找与`grades`变量的取值相匹配的数据，从而把`info`表中相应于`grades`变量的每一个值的数据复制过来。有两种方法可以实现：使用`match()`和整数子集选取；或者使用`rownames()`和字符子集选取。

47

```
grades
#> [1] 1 2 2 3 1
# Using match
id <- match(grades, info$grade)
info[id, ]
#>   grade      desc  fail
#> 3     1      Poor  TRUE
#> 2     2      Good FALSE
```

```
#> 2.1     2      Good FALSE
#> 1       3 Excellent FALSE
#> 3.1     1      Poor  TRUE

# Using rownames
rownames(info) <- info$grade
info[as.character(grades), ]
#>     grade      desc  fail
#> 1       1      Poor  TRUE
#> 2       2      Good FALSE
#> 2.1     2      Good FALSE
#> 3       3 Excellent FALSE
#> 1.1     1      Poor  TRUE
```

如果有多列匹配，就需要首先将它们整合成一列（使用 `interaction()`、`paste()` 或 `plyr::id()`）。也可以使用 `merge()` 或 `plyr::join()`，它们也做同样的事情——请阅读源代码看看它们是如何实现的。

3.4.3 随机样本 / 自助法（整数子集选取）

可以应用整数值索引来对向量或者数据框进行随机采样或者自助法抽样。函数 `sample()` 产生一个索引向量，然后用它为索引来提取相应的值。

```
df <- data.frame(x = rep(1:3, each = 2), y = 6:1, z = letters[1:6])

# Randomly reorder
df[sample(nrow(df)), ]
#>   x y z
#> 5 3 2 e
#> 6 3 1 f
#> 1 1 6 a
#> 2 1 5 b
#> 3 2 4 c
#> 4 2 3 d
# Select 3 random rows
df[sample(nrow(df), 3), ]
#>   x y z
#> 2 1 5 b
#> 1 1 6 a
#> 5 3 2 e
# Select 6 bootstrap replicates
df[sample(nrow(df), 6, rep = T), ]
#>     x y z
#> 1   1 6 a
#> 2   1 5 b
#> 6   3 1 f
#> 1.1 1 6 a
#> 4   2 3 d
#> 1.2 1 6 a
```

48

`sample()` 的参数控制要提取的样本数以及是否执行替换采样。

3.4.4 排序（整数子集选取）

`order()` 以一个向量作为输入，返回一个用于描述其中子集向量排列顺序的整型向量。

```
x <- c("b", "c", "a")
order(x)
#> [1] 3 1 2
x[order(x)]
#> [1] "a" "b" "c"
```

为了处理并列大小，可以为函数 order() 添加其他排序变量；同时，可以为 order() 添加参数 decreasing = TRUE，这样就从升序变成降序。默认情况下，缺失值会放在向量的末尾。但是，可以使用 na.last = NA 来去除它们，或者使用 na.last = FALSE 将它们放在向量的开头。

对于二维或更高维数据，order() 和整数子集选取可以很容易地对一个对象的行或列进行排序：

```
# Randomly reorder df
df2 <- df[sample(nrow(df)), 3:1]
df2
#>   z y x
#> 4 d 3 2
#> 1 a 6 1
#> 3 c 4 2
#> 6 f 1 3
#> 5 e 2 3
#> 2 b 5 1

df2[order(df2$x), ]
#>   z y x
#> 1 a 6 1
#> 2 b 5 1
#> 4 d 3 2
#> 3 c 4 2
#> 6 f 1 3
#> 5 e 2 3
df2[, order(names(df2))]
#>   x y z
#> 4 2 3 d
#> 1 1 6 a
#> 3 2 4 c
#> 6 3 1 f
#> 5 3 2 e
#> 2 1 5 b
```

表达式越简洁，其灵活性也就越低。向量排序可以使用 sort()，数据框可以使用 plyr::arrange()。

3.4.5 展开重复记录（整数子集选取）

有时你得到的数据框是已经整理过的，其中相同的记录已整合为一条记录，数据框增加一列来统计此条记录出现的次数。此时可以使用 rep() 来创建重复行的索引，再使用整数子集选取将这些整合的重复记录展开：

```
df <- data.frame(x = c(2, 4, 1), y = c(9, 11, 6), n = c(3, 5, 1))
rep(1:nrow(df), df$n)
#> [1] 1 1 1 2 2 2 2 2 3
df[rep(1:nrow(df), df$n), ]
```

```
#>     x  y n
#> 1   2  9 3
#> 1.1 2  9 3
#> 1.2 2  9 3
#> 2   4 11 5
#> 2.1 4 11 5
#> 2.2 4 11 5
#> 2.3 4 11 5
#> 2.4 4 11 5
#> 3   1  6 1
```

50

3.4.6 剔除数据框中某些列（字符子集选取）

有两种方式从数据框中剔除列数据。可以把这些列分别设为 NULL：

```
df <- data.frame(x = 1:3, y = 3:1, z = letters[1:3])
df$z <- NULL
```

或者也可以进行子集选择只返回需要的列。

```
df <- data.frame(x = 1:3, y = 3:1, z = letters[1:3])
df[c("x", "y")]
#>   x y
#> 1 1 3
#> 2 2 2
#> 3 3 1
```

如果知道不需要的列，可以使用集合运算来找出需要保留的列：

```
df[setdiff(names(df), "z")]
#>   x y
#> 1 1 3
#> 2 2 2
#> 3 3 1
```

3.4.7 根据条件选取行（逻辑子集选取）

由于可以很容易地组合来自多列的条件，所以从数据框中提取行的最常用的技术可能是应用逻辑子集选取方法。

51

```
mtcars[mtcars$gear == 5, ]
#>     mpg cyl  disp  hp drat    wt qsec vs am gear carb
#> 27 26.0   4 120.3  91 4.43 2.140 16.7  0  1    5    2
#> 28 30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
#> 29 15.8   8 351.0 264 4.22 3.170 14.5  0  1    5    4
#> 30 19.7   6 145.0 175 3.62 2.770 15.5  0  1    5    6
#> 31 15.0   8 301.0 335 3.54 3.570 14.6  0  1    5    8
mtcars[mtcars$gear == 5 & mtcars$cyl == 4, ]
#>     mpg cyl  disp  hp drat    wt qsec vs am gear carb
#> 27 26.0   4 120.3  91 4.43 2.140 16.7  0  1    5    2
#> 28 30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
```

需要记住的是：要使用向量的布尔运算符 & 和 |，而不是短路标量（short-circuiting）运算符 && 和 ||，短路标量运算符在 if 语句中非常有用。别忘了德·摩根定理，它对简化逆运算很有用：

- !(X & Y) 等价于 !X | !Y
- !(X | Y) 等价于 !X & !Y

例如，!(X & !(Y | Z)) 可以简化为 !X | !!(Y|Z)，然后进一步简化为 !X | Y | Z。

subset()是对数据框进行子集选取的专用简写函数，它可以减少键盘输入，因为不需要再重复数据框的名字。将在第13章讨论它。

```
subset(mtcars, gear == 5)
#>     mpg cyl  disp  hp drat    wt qsec vs am gear carb
#> 27 26.0   4 120.3  91 4.43 2.140 16.7  0  1    5    2
#> 28 30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
#> 29 15.8   8 351.0 264 4.22 3.170 14.5  0  1    5    4
#> 30 19.7   6 145.0 175 3.62 2.770 15.5  0  1    5    6
#> 31 15.0   8 301.0 335 3.54 3.570 14.6  0  1    5    8
subset(mtcars, gear == 5 & cyl == 4)
#>     mpg cyl  disp  hp drat    wt qsec vs am gear carb
#> 27 26.0   4 120.3  91 4.43 2.140 16.7  0  1    5    2
#> 28 30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
```

52

3.4.8 布尔代数与集合（逻辑和整数子集选取）

集合运算（整数子集选取）和布尔代数（逻辑子集选取）之间具有天然的对等性，知道这一点是非常有用的。在下列情况中集合运算更有效率：

- 需要找到第一个（或最后一个）TRUE。
- 数据中只有非常少的TRUE，却有非常多的FALSE；集合表示会更快并需要更少的内存。

which()函数可以将布尔表示转换成整数表示。在R基础包中没有对此过程进行逆操作的函数，但是创建一个也很容易：

```
x <- sample(10) < 4
which(x)
#> [1] 3 8 9

unwhich <- function(x, n) {
  out <- rep_len(FALSE, n)
  out[x] <- TRUE
  out
}
unwhich(which(x), 10)
#>  [1] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE
#> [10] FALSE
```

首先，创建两个逻辑向量和它们的整数等价体，然后探索布尔运算和集合运算的关系。

```
(x1 <- 1:10 %% 2 == 0)
#>  [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
#> [10]  TRUE
(x2 <- which(x1))
#> [1]  2  4  6  8 10
(y1 <- 1:10 %% 5 == 0)
#>  [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE
#> [10]  TRUE
(y2 <- which(y1))
#> [1]  5 10
# X & Y <-> intersect(x, y)
x1 & y1
#>  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
#> [10]  TRUE
intersect(x2, y2)
#> [1] 10
```

```
# X | Y <-> union(x, y)
x1 | y1
#>  [1] FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE
#> [10]  TRUE
union(x2, y2)
#> [1]  2  4  6  8 10  5

# X & !Y <-> setdiff(x, y)
x1 & !y1
#>  [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
#> [10] FALSE
setdiff(x2, y2)
#> [1] 2 4 6 8

# xor(X, Y) <-> setdiff(union(x, y), intersect(x, y))
xor(x1, y1)
#>  [1] FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE
#> [10] FALSE
setdiff(union(x2, y2), intersect(x2, y2))
#> [1] 2 4 6 8 5
```

在第一次学习子集选取时，最常犯的一个错误就是用 `x[which(y)]` 而不是 `x[y]`。这里，`which()` 不做任何事情：它从逻辑子集选取转换为整数子集选取，但结果是完全一样的。`x[-which(y)]` 与 `x[!y]` 也是不等价的：如果 `y` 全是 `FALSE`，`which(y)` 就是 `integer(0)`，`-integer(0)` 仍然是 `integer(0)`，所以最后没有得到值。通常，除非必需这样做，例如，第一个或最后一个 `TRUE` 值，否则不要将逻辑子集选取转换成整数子集选取。

3.4.9 练习

53 ~ 54

1. 如何随机重新排列数据框的列？（在随机森林中这是一项非常重要的技术。）能同时对行和列重新排列吗？
2. 如何从数据框中随机选择 `m` 行？如果样本必须连续的呢（比如，选取起始行、终止行以及它们之间的所有行）？
3. 如何将数据框中的列按字母顺序排序？

3.5 答案

1）使用正整数可以选择指定位置上的元素，负整数舍弃该位置上的元素；逻辑向量只保留对应位置为 `TRUE` 的元素；字符向量选择匹配名字的元素。

2）`[` 选择子列表。它总是返回列表。如果你把它和单个正整数联合使用，它返回长度为 1 的列表。`[[` 选择列表中的一个元素。`$` 是一个方便的简写：`x$y` 与 `x[["y"]]` 是等价的。

3）如果你对矩阵、数组或数据框进行子集选取并且想保留初始维度，那么使用参数 `drop = FALSE`。当你在函数内部进行子集选取时，你应该使用它。

4）如果 `x` 是矩阵，`x[] <- 0` 将每个元素变成 0，但行数和列数不变。而 `x <- 0` 完全用 0 替换该矩阵。

5）可以使用一个命名的字符向量作为简单的查询表：`c(x = 1, y = 2, z = 3)[c("y", "z", "x")]`。

55 ~ 56

第 4 章

常用函数与数据结构

R 语言中的一个重要组成部分是常用函数与数据结构。我相信下面列出的这些函数就能够组成一个列表。虽然你不需要通晓每一个函数的细节，但是至少要知道它们的存在。如果这个列表中的有些函数你还从未听说过，我强烈建议你阅读它们的文档。

本章是在我浏览了 R 基础包、统计包和实用工具包中所有的函数之后总结出来的，它包含了我认为最有用的所有函数。本章还包含了一些其他包中特别重要的函数及重要的 `options()` 的选项。

4.1 基础函数

```
# 首先要学习的函数
?
str
# 重要的运算符和赋值函数
%in%, match
=, <-, <<-
$, [, [[, head, tail, subset
with
assign, get
# 比较
all.equal, identical
!=, ==, >, >=, <, <=
is.na, complete.cases
is.finite
# 基础数学
*, +, -, /, ^, %%, %/%
abs, sign
acos, asin, atan, atan2
sin, cos, tan
ceiling, floor, round, trunc, signif
exp, log, log10, log2, sqrt

max, min, prod, sum
cummax, cummin, cumprod, cumsum, diff
pmax, pmin
```

```
range
mean, median, cor, sd, var
rle
#处理函数的函数
function
missing
on.exit
return, invisible
#逻辑与集合
&, |, !, xor
all, any
intersect, union, setdiff, setequal
which
#向量与矩阵
c, matrix
#强制转换规则字符型（character）〉数值型（numeric）〉逻辑型（logical）
length, dim, ncol, nrow
cbind, rbind
names, colnames, rownames
t
diag
sweep
as.matrix, data.matrix
#构建向量
c
rep, rep_len
seq, seq_len, seq_along
rev
sample
choose, factorial, combn
(is/as).(character/numeric/logical/...)
#列表与数据框
list, unlist
data.frame, as.data.frame
split
expand.grid
#控制流
if, &&, || (short circuiting)
for, while
next, break
switch
ifelse
# apply 函数和相似函数
lapply, sapply, vapply
apply
tapply
replicate
```

4.2　常见数据结构

```
#日期时间
```

```
ISOdate, ISOdatetime, strftime, strptime, date
difftime
julian, months, quarters, weekdays
library(lubridate)
#字符处理
grep, agrep
gsub
strsplit
chartr
nchar
tolower, toupper
substr
paste
library(stringr)
#因子
factor, levels, nlevels
reorder, relevel
cut, findInterval
interaction
options(stringsAsFactors = FALSE)
#数组处理
array
dim
dimnames
aperm
library(abind)
```

59

4.3 统计函数

```
#排序和制表
duplicated, unique
merge
order, rank, quantile
sort
table, ftable
#线性模型
fitted, predict, resid, rstandard
lm, glm
hat, influence.measures
logLik, df, deviance
formula, ~, I
anova, coef, confint, vcov
contrasts
#测试类函数
apropos("\\.test$")
#随机变量
(q, p, d, r) * (beta, binom, cauchy, chisq, exp, f, gamma, geom,
  hyper, lnorm, logis, multinom, nbinom, norm, pois, signrank, t,
  unif, weibull, wilcox, birthday, tukey)
#矩阵运算
crossprod, tcrossprod
eigen, qr, svd
%*%, %o%, outer
rcond
solve
```

60

4.4 使用 R

```
# 工作空间
ls, exists, rm
getwd, setwd
q
source
install.packages, library, require
# 帮助
help, ?
help.search
apropos
RSiteSearch
citation
demo
example
vignette
# 调试
traceback
browser
recover
options(error = )
stop, warning, message
tryCatch, try
```

61

4.5 I/O 函数

```
# 输出
print, cat
message, warning
dput
format
sink, capture.output
# 读 / 写数据
data
count.fields
read.csv, write.csv
read.delim, write.delim
read.fwf
readLines, writeLines
readRDS, saveRDS
load, save
library(foreign)
# 文件和目录
dir
basename, dirname, tools::file_ext
file.path
path.expand, normalizePath
file.choose
file.copy, file.create, file.remove, file.rename, dir.create
file.exists, file.info
tempdir, tempfile
download.file, library(downloader)
```

62

第 5 章

R 编程风格指南

好的编程风格（程序编写格式）就像正确使用标点符号。没有它也可以，但是它会让你编写的程序更易于理解。就像标点符号的使用方法一样，它也可以有很多变体。下面描述我（在本书和任何地方）使用的编程风格。它是在 Google R 编程风格指南（http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html）的基础上稍做修改而成的。你不一定要使用我的编程风格，但是你真的应该使用一个统一的编程风格。

因为虽然你的代码只有一个作者，但通常有许多读者，所以好的风格是很重要的。当你和其他人合作编写代码时，尤其要注意编程风格。此时，一开始就编程风格达成一致是上策。由于没有一种风格严格优于其他风格，所以与其他人合作意味着你可能需要对你自己喜好的某些风格做出让步。

Yihui Xie 编写的 formatR 添加包使得整理格式混乱的代码变得容易。该包不能解决一切问题，但可以快速使代码面目一新。在应用该添加包之前确保阅读维基上的信息（`https://github.com/yihui/formatR/wiki`）。

5.1 符号和名字

5.1.1 文件名

文件名应该有一定的意义，并且以 `.R` 结尾。

```
# Good
fit-models.R
utility-functions.R

# Bad
foo.r
stuff.r
```

如果文件需要按顺序执行，那么最好在命名时加上数字前缀：

```
0-download.R
1-parse.R
2-explore.R
```

5.1.2 对象名

“在计算机科学中有两件非常难的事：缓存失效和对象命名。”

——Phil Karlton

变量和函数名应该是小写字母。使用下划线 (_) 将名字中的单词分开。通常，变量名应该是名词，函数名应该是动词。追求简洁而有意义的名字。(这通常也是不容易的！)

```
# Good
day_one
day_1

# Bad
first_day_of_the_month
DayOne
dayone
djm1
```

如果可能，避免使用现有函数和变量的名字。这将给阅读代码的人带来疑惑。

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

63 ~ 64

5.2 语法

5.2.1 空格

在所有中缀运算符（=、+、-、<- 等）的两边使用空格。当在函数调用中使用 = 时也可以应用相同的规则。要在逗号的后面而不是前面加上空格（就像普通英文书写一样）。

```
# Good
average <- mean(feet / 12 + inches, na.rm = TRUE)

# Bad
average<-mean(feet/12+inches,na.rm=TRUE)
```

这个规则也有一个小小的例外：:、:: 和 ::: 的两边不需要空格。

```
# Good
x <- 1:10
base::get

# Bad
x <- 1 : 10
base :: get
```

在小括号的左边放置一个空格，函数调用除外。

```
# Good
if (debug) do(x)
plot(x, y)

# Bad
if(debug)do(x)
plot (x, y)
```

如果为了对齐等号或赋值符号（<-），可以使用额外的空格（一行中可以有多个空格）。

65

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

小括号或中括号内代码的两侧不需要放置空格（除非有逗号，要在逗号的后边放置

空格)。

```
# Good
if (debug) do(x)
diamonds[5, ]

# Bad
if ( debug ) do(x)  # No spaces around debug
x[1,]   # Needs a space after the comma
x[1 ,]  # Space goes after comma not before
```

5.2.2 大括号

大括号的左半边不能独占一行，它后边应该新起一行。右半边应该独占一行或者它的后边是 `else`。

对大括号中的代码要进行缩进。

```
# Good

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

if (y == 0) {
  log(x)
} else {
  y ^ x
}

# Bad

if (y < 0 && debug)
message("Y is negative")
if (y == 0) {
  log(x)
}
else {
  y ^ x
}
```


可以在同一行上留下非常短的语句：

```
if (y < 0 && debug) message("Y is negative")
```

5.2.3 行的长度

努力使每行代码不超过 80 个字符。使用合理大小的字体将代码打印到纸上时这样的长度会比较合适。如果你发现你的代码超过了这个长度，这说明应该将一些功能打包成独立的函数。

5.2.4 缩进

缩进代码时使用两个空格。不要使用制表符（tab）或者制表符加空格。

唯一的例外是，如果一个函数定义跨越了很多行。这种情况下，第二行要缩进到定义开始的地方：

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

5.2.5　赋值

赋值时，使用 <- 而不是 =。

```
# Good
x <- 5
# Bad
x = 5
```

67

5.3　结构

注释指南

对代码进行注释。每行的注释都应该以注释符和一个空格开始：#。注释应该解释为什么，而不是什么。

使用由 - 和 = 和构成的注释行将文件分割成容易理解的段落（块）。

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

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

68

第 6 章
函　数

函数是 R 的基本组成单元：要熟练掌握本书中提到的更多高级技术，就必须理解函数是如何工作的。你可能已经创建过很多 R 函数，也对函数是如何工作的有一定的认识，你可能已经具有了关于函数的一些零碎知识。本章重点是加强对函数是什么以及它们是如何工作的严谨理解。本章学习一些有意思的技巧和技术，同时这些知识也是以后学习其他更高级技术的基础。

函数本身就是对象，这是 R 语言中我们需要理解的最重要一点。我们可以像处理其他对象那样处理函数。第 10 章会更加深入地探讨这个话题。

小测验

回答下面的问题，看看你是不是可以跳过本章。可以在 6.7 节中找到答案。

1）函数的 3 个组成部分是什么？

2）下面代码的返回值是多少？

```
y <- 10
f1 <- function(x) {
  function() {
    x + 10
  }
}
f1(1)()
```

3）如何写下面的代码，使其更具一般性？


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

4）如何使下面的调用变得更易于理解？

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

5）调用下面的函数时会出错吗？为什么会出错或者不会出错？

```
f2 <- function(a, b) {
  a * 10
}
f2(10, stop("This is an error!"))
```

6）什么是中缀（infix）函数？如何编写中缀函数？什么是替换（replacement）函数？如何编写替换函数？

7）无论函数如何终止，我们都希望出现一个清理动作，使用哪个函数可以做到这一点？

主要内容

- 6.1 节介绍函数的 3 个主要组成部分。
- 6.2 节学习 R 如何从变量名找到其相应取值，也就是词法作用域的过程。
- 6.3 节说明 R 中发生的所有事情都是函数调用的结果，即便看上去并不是那样。
- 6.4 节介绍为函数提供参数的 3 种方法，如何使用参数列表来调用函数，以及惰性求值的影响。
- 6.5 节描述两个特殊类型的函数：中缀函数和替换函数。
- 6.6 节讨论函数什么时候以及如何返回值，如何在函数退出前确保它做一些事情。

预备条件

本章需要的唯一添加包就是 `pryr`，当一个向量被修改后可以使用它来查看到底发生了什么。安装它的命令为 `install.packages("pryr")`。 70

6.1 函数组成部分

所有的 R 函数都包含 3 个部分：

- body()，函数的内部代码。
- formals()，控制如何调用函数的参数列表。
- environment()，函数变量位置的“地图”。

当在 R 中输出函数时，将说明这 3 个重要部分。如果没有显示环境，这表示该函数是在全局环境中构建的。

```
f <- function(x) x^2
f
#> function(x) x^2

formals(f)
#> $x
body(f)
#> x^2
environment(f)
#> <environment: R_GlobalEnv>
```

可以使用 `body()`、`formals()` 和 `environment()` 的赋值形式对函数进行修改。

与 R 中的所有对象一样，函数也可以有任意多的附加属性（`attributes()`）。基础 R 包中使用的一个属性是“srcref”，它是源参考（source reference）的简写，它指向用来创建函数的源代码。与 `body()` 不同，它包含代码的注释和其他格式。也可以为函数添加属性。例如，可以设置 `class()` 并添加自定义的 `print()` 方法。

6.1.1 原函数

原则上函数包含 3 个部分，但有一个例外：原函数。例如，原函数 `sum()`，它使用 `.Primitive()` 直接调用 C 代码且不包含 R 代码。因此，它们的 `formals()`、`body()` 和 `environment()` 都是 `NULL`： 71

```
sum
#> function (..., na.rm = FALSE)  .Primitive("sum")
formals(sum)
#> NULL
body(sum)
#> NULL
environment(sum)
#> NULL
```

原函数只存在于 base（基础）包中，由于它们在低层进行运算，所以它们可以更高效（原始替换函数不需要复制），参数匹配也有不一样的规则（例如，switch 和 call）。但是，这样做的代价是与 R 中的所有其他函数的行为都不同。因此 R 的核心团队通常避免创建这样的函数，除非别无选择。

6.1.2 练习

1. 使用什么函数来判断一个对象是不是函数？使用什么函数来判断一个函数是不是原函数？
2. 下面的代码可以列出基础包中的所有函数。

```
objs <- mget(ls("package:base"), inherits = TRUE)
funs <- Filter(is.function, objs)
```

使用它来回答下面的问题：

a. 哪个基础函数有最多的参数？

b. 多少个基础函数没有参数？这些函数有哪些特别之处？

c. 如何对这段代码进行修改以便找到所有的原函数？

3. 函数的 3 个重要部分是什么？

72 4. 什么时候输出一个函数不会显示创建它所在的环境？

6.2 词法作用域

词法作用域是一组规则，它指引 R 如何找到一个符号的值。在下面的例子中，作用域就是 R 用来从符号 x 找到它的值 10 的一组规则：

```
x <- 10
x
#> [1] 10
```

对作用域的理解可以帮助你：

- ❑ 通过组合函数来创建工具，如第 10 章所述。
- ❑ 重写通常的求值规则并进行非标准求值（non-standard evaluation），如第 13 章所述。

R 包含两种类型的作用域：**词法作用域**（lexical scoping），它在语言层上自动实施；**动态作用域**（dynamic scoping），它主要用于在交互式分析中选择函数来减少键盘输入量。这里我们主要讨论词法作用域，因为它与函数创建密切相关。动态作用域将在 13.3 节中详细描述。

词法作用域根据函数在创建时的嵌套结构来查找符号的值，而不是根据调用它们时的嵌套方式。有了词法作用域，我们就不需要知道函数在调用时在何处找到变量的值。我们只需要查找函数的定义。

词法作用域中的“词法”（lexical）并不是英语中的词法（即语言中的字或词汇，区别于它们的语法和结构），而是来源于计算机科学中的术语“词法分析”（lexing），它是把用文本

表示的代码（伪代码）转换为编程语言能够理解的有意义代码的过程的一部分。

R 词法作用域实现的 4 个基本原则是：

- 名字屏蔽
- 函数与变量
- 重新开始
- 动态查找

73

尽管你可能没有条理化地考虑这些原则，但你或许已经了解了这些原则的许多内容。在查看答案之前，首先在大脑中运行下面每一个代码块中的代码，以此来测试你的编程知识。

6.2.1 名字屏蔽

下面的例子说明了词法作用域的最基本原则，你应该能够正确预测它的输出。

```
f <- function() {
  x <- 1
  y <- 2
  c(x, y)
}
f()
rm(f)
```

如果名字不是在函数内部定义的，那么 R 就会到上一层去进行查找。

```
x <- 2
g <- function() {
  y <- 1
  c(x, y)
}
g()
rm(x, g)
```

如果一个函数在另一个函数的内部定义，也使用同样的规则：在当前函数内查找，然后到定义函数的地方查找，等等，直到在全局环境中查找，然后到其他已加载的添加包中查找。在大脑中运行下面的代码，然后通过运行 R 代码，看看你预测的结果是否正确。

```
x <- 1
h <- function() {
  y <- 2
  i <- function() {
    z <- 3
    c(x, y, z)
  }
  i()
}
h()
rm(x, h)
```

74

相同的规则也适用于闭包（closure），闭包是由其他函数创建的函数。闭包将在第 10 章中详细介绍。这里，我们只看看它们是如何与作用域相互作用的。下面的函数 `j()` 返回一个函数。当调用这个函数时，你认为它将返回什么？

```
j <- function(x) {
  y <- 2
  function() {
    c(x, y)
  }
```

```
}
k <- j(1)
k()
rm(j, k)
```

这看上去有点神奇（在这个函数被调用之后，R 是如何知道 y 的值呢？）这是因为 k 保留了这个函数定义时的环境且这个环境中包含 y 的值。第 8 章将给出一些指导：如何深入了解和剖析与每个函数相关联的环境中保存的那些值。

6.2.2 函数与变量

与值的类型无关，相同的原则同样适用——寻找函数与寻找变量的方法相同：

```
l <- function(x) x + 1
m <- function() {
  l <- function(x) x * 2
  l(10)
}
m()
#> [1] 20
rm(l, m)
```

75

对于函数，这个规则有一点小修改。如果你正在使用的名字在其上下文中明显地为一个函数，（例如，f(3)），R 在查找过程中会忽略那些不是函数的对象。在下面的例子中，根据 R 查找函数还是查找变量，n 的值也会不同。

```
n <- function(x) x / 2
o <- function() {
  n <- 10
  n(n)
}
o()
#> [1] 5
rm(n, o)
```

但是，对象与函数的名字相同将使代码变得很难理解，因此要尽量避免这种情况。

6.2.3 重新开始

函数调用之间值会发生什么变化？第一次运行这个函数会发生什么？第二次会发生什么？（如果以前你没有看到过函数 exists()：如果参数中给出的变量存在，它返回 TRUE，否则它返回 FALSE。）

```
j <- function() {
  if (!exists("a")) {
    a <- 1
  } else {
    a <- a + 1
  }
  print(a)
}
j()
rm(j)
```

76

你可能对它返回相同的值感到惊讶，每次都是 1。这是由于每次调用这个函数都创建一个新的环境。一个函数不可能知道上次被调用时发生了什么；每次调用都是完全独立的（我

们将在 10.3.2 节中学习绕过这个问题的方法)。

6.2.4 动态查找

词法作用域决定到哪里查找这些值，而不是什么时候查找。当函数运行时，R 查找这些值，而不是在函数被创建时。这就意味着，依据环境中的对象函数的输出可能不同。

```
f <- function() x
x <- 15
f()
#> [1] 15

x <- 20
f()
#> [1] 20
```

通常情况下，你应该避免这种情况发生，因为这表示函数已经不是独立的。这是一个常见错误——如果你的代码中包含一个拼写错误，那么在你创建函数时不会得到错误，甚至在运行该函数时也不会得到错误，这取决于在全局环境中定义了什么变量。

检测这个问题的一种方法是使用 `codetools` 包中 `findGlobals()` 函数。这个函数返回一个函数的所有外界依赖关系：

```
f <- function() x + 1
codetools::findGlobals(f)
#> [1] "+" "x"
```

另一种尝试解决问题的方法是手动将函数的环境设置为 `emptyenv()`，即一个什么都没有的环境：

```
environment(f) <- emptyenv()
f()
#> Error: could not find function "+"
```

77

由于 R 是通过词法作用域来找到所有对象，甚至 + 运算符，所以这样做是没有用的。由于你必须总是依赖于 R 基础包或其他添加包中的函数，所以不可能创建一个完全独立的函数。

你也可以利用这个思想来做其他事情，虽然这是一个非常坏的建议。例如，因为 R 中的所有运算符都是函数，所以你可以重写它们。如果你对某人感到非常厌恶，当他离开他的计算机时，你可以在他的计算机上运行下列代码：

```
`(` <- function(e1) {
  if (is.numeric(e1) && runif(1) < 0.1) {
    e1 + 1
  } else {
    e1
  }
}
replicate(50, (1 + 2))
#>  [1] 3 3 3 3 4 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 4 3 3 3 4 3
#> [30] 3 3 3 3 3 4 3 3 4 3 3 3 3 3 3 3 3 3 3 3
rm("(")
```

这将引入一个致命的错误：有 10% 的机会，括号中的数值计算会被加上 1。所以在使用

R 时尽量开始一个新的对话（清空前面对话的数据）！

6.2.5 练习

1. 下面的代码会返回什么？为什么？下面的 3 个 c 分别代表什么意思？

```
c <- 10
c(c = c)
```

2. 控制 R 如何找到符号所对应值的 4 个基本原则是什么？

3. 下面函数返回什么？在运行你自己的代码前先预测一下。

```
f <- function(x) {
  f <- function(x) {
    f <- function(x) {
      x ^ 2
    }
    f(x) + 1
  }
  f(x) * 2
}
f(10)
```

78

6.3 每个运算都是一次函数调用

“为了理解 R 中的计算，下面的两个口号是有用的：

- 一切皆是对象。
- 一切皆是函数调用。”

——John Chambers

在前面的例子中我们对 (进行了重新定义，由于 R 中的每个运算都是一次函数调用，不管它看上去是否像函数调用。这包括中缀运算符（infix operator），如 + ；流控制运算符，如 `for`、`if` 和 `while`；子集选取运算符，如 `[]` 和 `$`；甚至是大括号 {。这就意味着下面例子中的每对语句都是等价的。需要注意的是，重音符 “`”，可以让你引用预留的或非法命名的函数或者变量。

```
x <- 10; y <- 5
x + y
#> [1] 15
`+`(x, y)
#> [1] 15

for (i in 1:2) print(i)
#> [1] 1
#> [1] 2
`for`(i, 1:2, print(i))
#> [1] 1
#> [1] 2
```

79

```
if (i == 1) print("yes!") else print("no.")
#> [1] "no."
`if`(i == 1, print("yes!"), print("no."))
#> [1] "no."
```

```
x[3]
#> [1] NA
`[`(x, 3)
#> [1] NA

{ print(1); print(2); print(3) }
#> [1] 1
#> [1] 2
#> [1] 3
`{`(print(1), print(2), print(3))
#> [1] 1
#> [1] 2
#> [1] 3
```

可以对这些特殊函数的定义进行重写，但基本上可以肯定这是一个坏主意。不过，这样做有时可能会有用：它可以让我们做一些本来不可能做的事情。例如，这个特点使 dplyr 包可以将 R 表达式转变成 SQL 表达式。第 15 章使用这个想法来创建特定领域的语言，从而使我们可以用现有的 R 结构来简洁地表达新概念。

将特殊函数看作普通函数在大多数情况下会有用。例如，我们可以先定义一个函数 add()，然后使用 sapply() 为列表中的每一个元素加 3，如下所示：

```
add <- function(x, y) x + y
sapply(1:10, add, 3)
#>  [1]  4  5  6  7  8  9 10 11 12 13
```

但是使用内置的 + 函数同样可以实现相同的效果。

```
sapply(1:5, `+`, 3)
#> [1] 4 5 6 7 8
sapply(1:5, "+", 3)
#> [1] 4 5 6 7 8
```

注意 `+` 和 "+" 是不同的。第一个是称为 + 的对象的值，第二个是一个包含字符 + 的字符串。第二个版本之所以也能使用，是因为 lapply 可以用函数的名字而不是函数本身作为参数：如果你阅读 lapply 的源代码，你会发现第一行使用 match.fun() 来查找给定名字的函数。 80

一个更有用的应用是将 lapply() 或 sapply() 与子集选取联合使用：

```
x <- list(1:3, 4:9, 10:12)
sapply(x, "[", 2)
#> [1]  2  5 11

# equivalent to
sapply(x, function(x) x[2])
#> [1]  2  5 11
```

记住，R 中发生的一切都是函数调用，这一点将在第 14 章中帮助你。

6.4 函数参数

区分函数的形式参数（形参）和实际参数（实参）是很有用的。函数的形参是函数的一个性质，而实参或者调用参数在函数的每次调用中都可能不同。本节介绍实参是怎样映射为形参的；如何调有给定参数列表的函数；默认参数是如何工作的，以及惰性求值的影响。

6.4.1 函数调用

在调用函数时设置参数的方法包括：指定参数位置、输入参数全名或者部分名称。将实参映射到形参的方法的优先级是：首先将名字完全匹配的参数进行映射；再将前缀匹配的参数进行映射；最后是将位置匹配的参数进行映射。

```
f <- function(abcdef, bcde1, bcde2) {
  list(a = abcdef, b1 = bcde1, b2 = bcde2)
}
str(f(1, 2, 3))
#> List of 3
#>  $ a : num 1
#>  $ b1: num 2
#>  $ b2: num 3
str(f(2, 3, abcdef = 1))
#> List of 3
#>  $ a : num 1
#>  $ b1: num 2
#>  $ b2: num 3

# Can abbreviate long argument names:
str(f(2, 3, a = 1))
#> List of 3
#>  $ a : num 1
#>  $ b1: num 2
#>  $ b2: num 3

# But this doesn't work because abbreviation is ambigu
str(f(1, 3, b = 1))
#> Error: argument 3 matches multiple formal arguments
```

81

通常，我们只对第一个参数或者前两个参数使用位置匹配。它们是最常用的，并且大多数读者也都知道它们代表什么。对于很少使用的参数，尽量避免使用位置匹配，部分匹配时也只使用可读的简写。（如果你在为一个将要在 CRAN 上发布的添加包编写代码，那么你不能使用部分匹配，此时必须使用完全匹配。）命名参数应该总是放在匿名参数的后边。如果一个函数使用了“...”参数（后边将详细讨论），你只能在“...”后边设置参数列表，而且要使用这些参数的全名。

下面是好的调用：

```
mean(1:10)
mean(1:10, trim = 0.05)
```

下面这行代码可能有点繁琐：

```
mean(x = 1:10)
```

82

下面的这些代码可能造成误解：

```
mean(1:10, n = T)
mean(1:10, , FALSE)
mean(1:10, 0.05)
mean(, TRUE, x = c(1:10, NA))
```

6.4.2 使用参数列表来调用函数

假设有一个函数参数列表：

```
args <- list(1:10, na.rm = TRUE)
```

如何将这个参数列表发送给函数 `mean()` 呢？需要使用 `do.call()`：

```
do.call(mean, list(1:10, na.rm = TRUE))
#> [1] 5.5
# Equivalent to
mean(1:10, na.rm = TRUE)
#> [1] 5.5
```

6.4.3 默认参数和缺失参数

R 中的函数参数可以有默认值。

```
f <- function(a = 1, b = 2) {
  c(a, b)
}
f()
#> [1] 1 2
```

由于 R 中的参数使用惰性求值（后边将学习更多内容），所以参数的默认值可以通过其他参数来定义：

```
g <- function(a = 1, b = a * 2) {
  c(a, b)
}
g()
#> [1] 1 2
g(10)
#> [1] 10 20
```

83

默认参数甚至可以用函数内部创建的变量来定义。这在 R 的基础包中经常使用，但是我觉得这不是一个好的做法，因为如果不阅读完整的源代码，就不能理解这个默认值到底代表什么。

```
h <- function(a = 1, b = d) {
  d <- (a + 1) ^ 2
  c(a, b)
}
h()
#> [1] 1 4
h(10)
#> [1]  10 121
```

可以使用 `missing()` 函数来确定一个参数是否被设置了。

```
i <- function(a, b) {
  c(missing(a), missing(b))
}
i()
#> [1] TRUE TRUE
i(a = 1)
#> [1] FALSE  TRUE
```

```
i(b = 2)
#> [1]  TRUE FALSE
i(1, 2)
#> [1] FALSE FALSE
```

有时我们需要设置一个非常重要的默认值，而这个值需要多行代码才能计算出来。如果代码太长，最好不要将它写入函数定义的内部，可以使用 `missing()` 函数，在需要时根据条件去计算它。但是，这样做的缺点是：如果不阅读文档，就很难知道哪个参数是必需的，哪个参数是可选的。相反，我经常将默认值设置为 `NULL`，然后使用函数 `is.null()` 来检查这个参数是否被设置了。

6.4.4 惰性求值

84 默认情况下，R 函数的参数都使用惰性求值——它们只有在实际被用到时才会被求值：

```
f <- function(x) {
  10
}
f(stop("This is an error!"))
#> [1] 10
```

如果你希望确保对一个参数求值，可以使用 `force()` 函数：

```
f <- function(x) {
  force(x)
  10
}
f(stop("This is an error!"))
#> Error: This is an error!
```

在使用 `lapply()` 或循环创建闭包时，这一点很重要。

```
add <- function(x) {
  function(y) x + y
}
adders <- lapply(1:10, add)
adders[[1]](10)
#> [1] 20
adders[[10]](10)
#> [1] 20
```

x 是在你第一次调用其中的某一个加法函数时才进行惰性估值。此时，循环已经完成，x 的最后值为 10。因此，所有的加法函数都给它们的输入加上 10，这可能并不是你想要的！可以手动强制求值来解决这个问题：

```
add <- function(x) {
  force(x)
  function(y) x + y
}
adders2 <- lapply(1:10, add)
adders2[[1]](10)
#> [1] 11
adders2[[10]](10)
#> [1] 20
```

85 这段代码完全等价于

```
add <- function(x) {
  x
  function(y) x + y
}
```

因为强制函数定义为 `force <- function(x) x`。但是，使用这个函数清楚地表明你正在进行强制求值，而不是不小心输入了一个 `x`。

默认参数在函数内部求值。这就意味着，如果表达式依赖于当前环境，那么计算结果将根据你使用默认值还是显式地提供参数值而不同。

```
f <- function(x = ls()) {
  a <- 1
  x
}

# ls() evaluated inside f:
f()
#> [1] "a" "x"

# ls() evaluated in global environment:
f(ls())
#>  [1] "add"     "adders"   "adders2"  "args"     "f"
#>  [6] "funs"    "g"        "h"        "i"        "metadata"
#> [11] "objs"    "x"        "y"
```

从技术上讲，没有被求值的参数称为**约定**（promise）或者（不太普遍的名称）形式转换（thumk）。约定通常由两部分组成：

- 产生延迟计算的表达式。（可以通过 `substitute()` 函数访问。可以参考第 13 章获得更多细节。）
- 创建和计算表达式的环境。

当第一次访问一个约定（promise）时，表达式将在创建它的环境中求值。将这个值缓存，所以后续对这个已经求值约定的访问不必重新计算该值（但是原始表达式还是与这个值相关联的，所以 `substitute()` 函数仍然能够访问它）。可以使用 `pryr::promise_info()` 找到更多关于约定的信息。使用一些 C++ 代码可以在不求值的情况下提取关于约定的信息，但只使用 R 代码是做不到的。 86

惰性在 if 语句中是很有用的——只有当第一个条件为真时，它后续的第二个语句才会被求值。如果不是这样，这条语句就会返回一个错误，因为 `NULL > 0` 不是 `if` 的合法输入，而是一个长度为 0 的逻辑向量。

```
x <- NULL
if (!is.null(x) && x > 0) {

}
```

我们可以实现自己的“&&”：

```
`&&` <- function(x, y) {
  if (!x) return(FALSE)
  if (!y) return(FALSE)

  TRUE
```

```
}
a <- NULL
!is.null(a) && a > 0
#> [1] FALSE
```

没有惰性求值这个函数就不能工作，因为总是要计算 x 和 y，即使当 a 为 NULL 时还要测试 a > 0。

有时，还可以使用惰性求值来完全删除一个 if 语句。例如，替代

```
if (is.null(a)) stop("a is null")
#> Error: a is null
```

可以写成：

```
!is.null(a) || stop("a is null")
#> Error: a is null
```

87

6.4.5 ... 参数

还有一个特殊参数称为 ...。这个参数将与所有没有匹配的参数进行匹配，并可以很容易地传递给其他函数。当如果想收集参数来调用其他函数，但又不想提前设定这些参数的名字时，这个特殊参数就很有用。... 经常与 S3 泛型函数联合使用以便使单个方法更加灵活。

使用 ... 的一个相对复杂的函数就是基础包中的函数 plot()。plot() 函数是一个含有参数 x、y 和 ... 的泛型方法。要理解一个函数中的 ... 参数的功能，需要阅读帮助文档："传递给方法的参数，比如绘图参数"。plot() 函数的最简单调用实际上是调用 plot.default()，它也有很多参数，也包括 ...。而且，阅读帮助文档，可知 ... 也接受其他绘图参数，在 par() 函数的帮助文档中列出了这些参数。我们可以将代码写成：

```
plot(1:5, col = "red")
plot(1:5, cex = 5, pch = 20)
```

上面的代码同时展示了参数 ... 的优点和缺点：它使 plot() 函数变得非常灵活，但是为了理解如何应用该参数，需要仔细阅读帮助文档。此外，如果我们阅读函数 plot.default 的源代码，就会发现一些文档中没有给出的特性。例如，可以给函数 Axis() 和 box() 传递其他参数：

```
plot(1:5, bty = "u")
plot(1:5, labels = FALSE)
```

可以使用 list(...) 这种易于使用的方式来捕获 ...。(在 13.5.2 节中查看其他获取 ... 但不对参数进行求值的方法。)

```
f <- function(...) {
  names(list(...))
}
f(a = 1, b = 2)
#> [1] "a" "b"
```

使用 ... 是有代价的——任何参数的拼写错误都不会产生错误警告，... 后边的参数必须使用全名。这就使得我们很容易忽略拼写错误：

88

```
sum(1, 2, NA, na.mr = TRUE)
#> [1] NA
```

通常情况下，显式设置参数比隐式设置好，所以你可以要求用户提供一个附加的参数列表。如果将 ... 与多个附加函数结合使用，这些肯定会变得容易。

6.4.6 练习

1. 解释下面这些奇怪的函数调用：

```
x <- sample(replace = TRUE, 20, x = c(1:10, NA))
y <- runif(min = 0, max = 1, 20)
cor(m = "k", y = y, u = "p", x = x)
```

2. 这个函数的返回值是什么？为什么？它说明了前面我们提到的哪条规则？

```
f1 <- function(x = {y <- 1; 2}, y = 0) {
  x + y
}
f1()
```

3. 这个函数的返回值是什么？为什么？它说明了前面我们提到的哪条规则？

```
f2 <- function(x = z) {
  z <- 100
  x
}
f2()
```

6.5 特殊调用

R 有两条附加语法来支持特殊类型函数的调用：中缀函数和替换函数。 89

6.5.1 中缀函数

R 中的大多数函数都是“前缀”运算符：函数的名字在参数的前面。也可以创建中缀函数，使函数的名字排在参数的中间，例如 + 或 −。所有用户创建的中缀函数必须以 % 开头，以 % 结尾，R 中预定义的中缀函数有：%%、%*%、%/%、%in%、%o%、%x%。（R 中不需要 % 的内置中缀运算符的完整列表为：::、:::、$、@、^、*、/、+、−、>、>=、<、<=、==、!=、!、&、&&、|、||、~、<−、<<−）。

例如，我们可以创建一个新的运算符，它将字符串连接到一起：

```
`%+%` <- function(a, b) paste(a, b, sep = "")
"new" %+% " string"
#> [1] "new string"
```

注意，在创建函数时，因为它是一个特殊的名字，所以需要将函数名放在反引号内。这只是普通函数调用的另一种方便的语法格式。就 R 而言，下面两个表达式是等价的：

```
"new" %+% " string"
#> [1] "new string"
`%+%`("new", " string")
#> [1] "new string"
```

下面两种方式也是等价的：

```
1 + 5
#> [1] 6
`+`(1, 5)
#> [1] 6
```

中缀函数的命名比普通的 R 函数更灵活：它们可以包含任何字符（当然，“%”除外）。应该避免在定义函数的字符串中使用特殊字符，但函数调用时可以使用：

```
`% %` <- function(a, b) paste(a, b)
`%'%` <- function(a, b) paste(a, b)
`%/\\%` <- function(a, b) paste(a, b)
"a" % % "b"
#> [1] "a b"
"a" %'% "b"
#> [1] "a b"
"a" %/\% "b"
#> [1] "a b"
```

90

根据 R 的默认优先级规则，中缀运算是从左到右进行的：

```
`%-%` <- function(a, b) paste0("(", a, " %-% ", b, "
"a" %-% "b" %-% "c"
#> [1] "((a %-% b) %-% c)"
```

有一个中缀函数是我经常使用的。它受 Ruby 语言的逻辑或运算符（||）的启发，Ruby 中对 if 语句的内容估值为 TRUE 的定义更加灵活，但是在 R 中稍有不同。当另一个函数的输出为 NULL 时，使用该运算符设置默认值将是一种很有用的方式：

```
`%||%` <- function(a, b) if (!is.null(a)) a else b
function_that_might_return_null() %||% default value
```

6.5.2 替换函数

替换函数看上去就像对它们的参数进行原地修改，而且有一个特殊的名字“xxx<-”。虽然可以有多个参数，但通常情况下只有两个（x 和 value），而且必须返回被修改的对象。例如，下面的函数允许你对一个向量的第二个值进行修改：

```
`second<-` <- function(x, value) {
  x[2] <- value
  x
}
x <- 1:10
second(x) <- 5L
x
#>  [1]  1  5  3  4  5  6  7  8  9 10
```

91

当 R 对赋值表达式“second(x) <- 5”进行计算时，它注意到“<-”的左边不是简单的名字，所以它查找一个名为“second<-”的函数来进行替换。

我们说是“看上去”像对参数进行原地修改，因为它们实际上创建了一个被修改的副本。我们可以使用 pryr::address() 来查看基础对象的内存地址。

```
library(pryr)
x <- 1:10
address(x)
#> [1] "0x7fb3024fad48"
second(x) <- 6L
address(x)
#> [1] "0x7fb3059d9888"
```

使用 .Primitive() 实现的内置函数将对对象进行原地修改：

```
x <- 1:10
address(x)
#> [1] "0x103945110"

x[2] <- 7L
address(x)
#> [1] "0x103945110"
```

因为这种行为会对性能产生重要影响，所以我们需要对这种行为引起重视。

如果想提供附加参数，可以把它们放置在 x 和 value 之间：

```
`modify<-` <- function(x, position, value) {
  x[position] <- value
  x
}
modify(x, 1) <- 10
x
#>  [1] 10  6  3  4  5  6  7  8  9 10
```

当调用 modify(x, 1) <- 10 时，R 在后台将其转换成： 92

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

这意味着不可以这样做：

```
modify(get("x"), 1) <- 10
```

因为这就变成了无效的代码：

```
get("x") <- `modify<-`(get("x"), 1, 10)
```

经常把替换与子集选取结合在一起使用：

```
x <- c(a = 1, b = 2, c = 3)
names(x)
#> [1] "a" "b" "c"
names(x)[2] <- "two"
names(x)
#> [1] "a"   "two" "c"
```

这样做是可以的，因为 names(x)[2] <- "two" 执行：

```
`*tmp*` <- names(x)
`*tmp*`[2] <- "two"
names(x) <- `*tmp*`
```

（对的，它的确创建了一个命名为 *tmp* 的局部变量，但随后就将它移除了。）

6.5.3 练习

1. 用基础包中的所有替换函数创建一个列表。其中哪些是原始函数？
2. 用户创建中缀函数时如何进行合法命名？
3. 创建一个中缀运算符 xor()。
4. 创建中缀版的集合函数 intersect()、union() 和 setdiff()。
5. 创建一个替换函数，它可以随机修改向量中的一个元素。 93

6.6 返回值

在一个函数中最后一个被计算的表达式成为函数的返回值，也就是调用该函数的结果。

```
f <- function(x) {
  if (x < 10) {
    0
  } else {
    10
  }
}
f(5)
#> [1] 0
f(15)
#> [1] 10
```

通常情况下，我认为当出现提前返回（例如，报错或者函数的一个简单条件被满足）时，应当显式地使用 `return()` 函数。这样的编程风格可以减少缩进的层次。同时，因为你可以在局部对其进行解读，所以通常使函数变得容易理解。

```
f <- function(x, y) {
  if (!x) return(y)

  # complicated processing here
}
```

函数只可以返回一个对象。但是这并不会产生任何限制，因为你可以返回一个包含任意数量对象的列表。

函数中最容易理解的应该是纯函数（pure function）：这些函数总是将相同的输入映射到相同的输出，并且不会对工作空间产生任何影响。换句话说，纯函数没有**副作用**（side effect）：除了返回值之外，它们不会对整个环境的状态产生任何影响。

R 使我们避免了一种副作用：大多数 R 对象都具有复制后修改（copy-on-modify）语义。所以，对函数的参数进行修改不会改变参数的原始值：

```
f <- function(x) {
  x$a <- 2
  x
}
x <- list(a = 1)
f(x)
#> $a
#> [1] 2
x$a
#> [1] 1
```

94

（对于复制后修改规则，有两个重要的例外：环境和参考类。它们可以被原地修改，所以在处理它们时一定要格外小心。）

这与其他可以对函数的输入进行修改的语言，如 java，有非常大的不同。这种复制后修改行为对性能产生重要影响，这会在第 17 章中详细讲述。（注意 R 使用复制后修改语义的一个结果是影响 R 的性能。但这并不是普遍适用的。有一门新的语言 Clojure，它大量地使用了复制后修改语义，但是对性能的影响有限。）

除了几个明显的例外，大多数的 R 基础函数是纯函数：

- `library()` 函数用于加载软件包，因此它修改搜索路径。
- `setwd()`、`Sys.setenv()`、`Sys.setlocale()` 分别修改工作路径、环境变量和局

部变量。

- ❑ plot() 和与它功能相似的函数产生图形输出。
- ❑ write()、write.csv()、saveRDS() 等将输出写入硬盘。
- ❑ options() 和 par() 修改全局设置变量。
- ❑ S4 相关的函数修改类和方法的全局表。
- ❑ 每次运行随机数发生器都会产生不同的结果。

最小化副作用通常是一个好注意，在可能的情况下将纯函数和不纯函数进行分离从而使副作用的影响最小化。纯函数相对容易测试（因为你只需要关心输入和输出），并且在不同平台或 R 的不同版本中结果都是一样的。例如，这是 ggplot2 的原则之一：大多数运算是对绘图的对象进行运算，只有最后的 print 或 plot 调用才对实际绘图产生副作用。 95

函数可以返回 invisible 值。当调用这个函数时，默认情况下不会输出这个值。

```
f1 <- function() 1
f2 <- function() invisible(1)

f1()
#> [1] 1
f2()
f1() == 1
#> [1] TRUE
f2() == 1
#> [1] TRUE
```

你可以使用括号将其括起来从而使其强制显示出来：

```
(f2())
#> [1] 1
```

“<-”应该是最常用的不显示返回值的函数：

```
a <- 2
(a <- 2)
#> [1] 2
```

这样我们就可以将一个值赋给多个变量：

```
a <- b <- c <- d <- 2
```

因为它被解析为：

```
(a <- (b <- (c <- (d <- 2))))
#> [1] 2
```

96

6.6.1 退出时

除了可以返回一个值外，当函数退出时可以使用 on.exit() 来触发其他事件。当函数退出时，它经常用来确保全局状态的改变能够恢复原状。无论函数是如何退出的（是否有显式的提前返回、是否出错，或者只是简单地执行到函数体的结尾），on.exit() 中的代码都会执行。

```
in_dir <- function(dir, code) {
  old <- setwd(dir)
  on.exit(setwd(old))
```

```
  force(code)
}
getwd()
#> [1] "/Users/hadley/Documents/adv-r/adv-r"
in_dir("~", getwd())
#> [1] "/Users/hadley"
```

基本原理很简单：

- 首先，我们设置一个新的工作目录，并从 `setwd()` 的输出获取当前位置。
- 然后，使用 `on.exit()` 确保工作目录返回到前一个值，不管函数如何退出。
- 最后，我们显式地强制执行代码。（实际上这里不需要使用 `force()`，但这会非常清楚地告诉读者我们正在这里做什么。）

注意：如果你正在一个函数中使用多个 `on.exit()` 调用，一定要确保设置了参数 `add = TRUE`。不幸的是，默认情况下 `on.exit()` 中为 `add = FALSE`，所以每次运行它时，它将重写现有退出表达式。因为 `on.exit()` 的实现方法，所以我们不可能创建一个 `add = TRUE` 的变体，因此在使用它时一定要小心。

6.6.2 练习

97

1. 函数 `source()` 的参数 `chdir` 与 `in_dir()` 有何不同？为什么我们更喜欢其中的一个呢？
2. `library()` 的功能是什么？如何保存和恢复 `options()` 和 `par()` 的值？
3. 写一个可以打开图形设备的函数，运行提供的函数，并关闭图形设备（无论绘图代码是否被执行，关闭动作总能执行）。
4. 使用 `on.exit()` 实现一个简单版本的 `capture.output()`。

```
capture.output2 <- function(code) {
  temp <- tempfile()
  on.exit(file.remove(temp), add = TRUE)

  sink(temp)
  on.exit(sink(), add = TRUE)

  force(code)
  readLines(temp)
}
capture.output2(cat("a", "b", "c", sep = "\n"))
#> [1] "a" "b" "c"
```

对比 `capture.output()` 和 `capture.output2()` 的结果。这两个函数有哪些不同？我除去哪些功能可以使大家更容易地理解这个函数的关键思想？

如何对此函数的关键思想进行改写？

6.7 答案

1）函数的 3 个组成部分为：函数体、参数和环境。

2）`f1(1)()` 的返回值为 11。

3）通常用中缀运算符，写成如下格式：`1 + (2 * 3)`。

4）写成 `mean(c(1:10, NA), na.rm = TRUE)` 会更容易理解。

5）不会抛出错误，因为永远不会使用第二个参数，所以它不会被计算。

6）参见 6.5.1 节和 6.5.2 节。

7）可以使用 `on.exit()`，具体细节请参见 6.6.1 节。 98

第 7 章
面向对象编程指南

本章是编程指南，它可以帮助我们初步认识和使用 R 的面向对象系统（Object Oriented System, OO)。R 有 3 个面向对象系统（再加上基础类型），所以它看上去有点吓人。本章的目标不是让你成为这 4 个系统的专家，而是帮助你认识正在使用的系统并使你可以更高效地应用它。

所有面向对象系统的核心都是类和方法的概念。**类**（class）通过描述对象的属性以及对象与其他类的关系来定义对象的行为。在选择**方法**（method）时同样要使用类，函数行为上的不同取决于它们输入类的不同。类通常是分层结构：如果子类中不存在某个方法，那它就会使用父类中的方法；子类从父类**继承**（inherit）方法。

R 的 3 个面向对象系统在类和方法的定义上有所不同：

- **S3** 使用了一种称为泛型函数 OO 的面向对象编程方式。这与大多数实现消息传递的 OO 编程语言（比如，Java、C++ 和 C#）不同。在消息传递中，消息（方法）被传递给对象，然后对象决定调用哪个函数。一般而言，对象在函数调用中以某种特殊标识呈现，一般出现在方法或消息名之前：例如，canvas.drawRect("blue")。S3 就不同了。虽然计算还是通过方法进行的，但它是由一种称为泛型函数（generic function）的特殊函数来决定调用哪个方法，例如，`drawRect(canvas, "blue")`。S3 是一个很不正式的 OO 系统。它没有类的正式定义。
- **S4** 的工作方式与 S3 类似，但是比 S3 更正式。它与 S3 有两点主要的不同。S4 有正式的类定义，它描述了每个类的表示方法和继承方法，并且有特殊的帮助函数用于定义泛型函数和方法。S4 还有多重分派，这就意味着泛型函数可以根据有任意参数数的类来选择调用方法。
- **参考类**（Reference class, RC），与 S3 和 S4 有显著不同，RC 实现消息传递 OO，所以方法属于类而不属于函数。`$` 用来分隔类和方法，所以函数调用的格式为 `canvas$drawRect("blue")`。RC 对象同样是可变的：它们不使用通常的复制后修改（copy-on-modify）语义，而是在原地进行修改。这使它们比较难推理，但也使它们可以解决一些使用 S3 和 S4 很难解决的问题。

还有另一个不完全的 OO 系统，但还是有必要提一提它：

- **基础类型**（base type），它是构成其他 OO 系统的内部 C 语言级别的类型。基本类型

大多数使用 C 代码操作，但是知道它们还是很重要的，因为它们提供了其他 OO 系统的基本单元。

下面各节从基础类型开始，依次介绍每个系统。你将学习如何识别一个对象到底属于哪个类，方法分派是如何工作的，如何为系统创建对象、类、泛型函数和方法。本章还对什么情况下使用什么样的系统给出了一些建议。

工具准备

使用 pryr 添加包提供的有用函数来检测 OO 性质，首先执行命令 `install.packages("pryr")` 来安装这个添加包。

小测验

你已经知道这些知识了吗？如果你能正确地回答下面的问题，你就可以跳过本章。你可以在 7.6 节查看答案。

1）如何知道一个对象与什么样的 OO 系统（基础、S3、S4 或者 RC）相关？

2）如何确定一个对象（比如，整型或列表）的基础类型？

3）什么是泛型函数？

4）S3 和 S4 的主要不同是什么？ S4 与 RC 的主要不同是什么？

100

主要内容

- 7.1 节学习 R 的基础对象系统。只有 R-core 可以给这个系统添加新的类，但是由于它是其他 3 个系统的基础，所以认识它还是很重要的。
- 7.2 节学习 S3 系统的基础知识。它是最简单也是使用最多的 OO 系统。
- 7.3 节学习更正式和严谨的 S4 系统。
- 7.4 节学习 R 的最新的 OO 系统：参考类（Reference Class，RC）。
- 7.5 节给出一些当你开始一个新项目时如何选择 OO 系统的建议。

7.1 基础类型

所有 R 对象的底层都是一个用来描述这个对象如何在内存中存储的 C 结构体。此结构体包含这个对象的内容、内存分配信息以及**类型**（type）。这是 R 对象的**基础类型**（base type）。基础类型不是真正的面向对象系统，因为只有 R 的核心团队才可以创建新类型。因此很少增加新类型：最近的改变是 2011 年，增加了两个你从来没有在 R 中看到过的独特类型（NEWSXP 和 FREESXP），但它们对诊断内存问题非常有用。在这之前，最近的改变是在 2005 年增加了 S4 的一个特殊基础类型（S4SXP）。

第 2 章解释了最常见的基础类型（原子向量和列表），但是基础类型也包括函数、环境和其他更独特的对象（比如，名字、调用和本书后面介绍的约定）。可以使用 `typeof()` 检测对象的基础类型。不幸的是，在整个 R 中，基础类型的名字也不是始终一致的，而且类型的返回值与相应的“is”函数的返回值也可能不同（可能使用不同的名字）：

```
# The type of a function is "closure"
f <- function() {}
typeof(f)
#> [1] "closure"
is.function(f)
#> [1] TRUE
```

101

```
# The type of a primitive function is "builtin"
typeof(sum)
#> [1] "builtin"
is.primitive(sum)
#> [1] TRUE
```

你可能已经听说过 `mode()` 和 `storage.mode()`。我建议你忘掉它们，因为它们只是函数 `typeof()` 返回值的别名，它们的存在只是为了与 S 保持兼容性。如果你想知道它们到底做了什么，可以阅读它们的源代码。

对不同基础类型具有不同行为的函数大部分使用 C 语言编写，它们的分派是通过 `switch` 语句实现的（例如，`switch(TYPEOF(x))`）。即使你永远都不会写 C 代码，但理解基础类型也是非常重要的，因为其他所有系统都是建立在它们之上的：可以使用任意基础类型来构建 S3 对象，使用特殊的基础类型来构建 S4 对象，RC 对象是 S4 与环境对象（另一个基础类型）的结合体。为了检查一个对象是不是一个纯基础类型（例如，它不包含 S3、S4 或 RC 的行为），检查 `is.object(x)` 的返回值是不是 `FALSE`。

7.2 S3

S3 是 R 的第一个也是最简单的 OO 系统。它是 R 基础包和统计包中唯一使用的 OO 系统，也是 CRAN 软件包中最常用的系统。S3 是非正式和特别的 OO 系统，但是它有一种极简主义的优雅：去掉其中任何一部分都会影响整个 OO 系统的使用。

7.2.1 认识对象、泛型函数和方法

我们遇到的大多数对象都是 S3 对象。但不幸的是，在 R 基础包中没有一个简单方法可以检查一个对象是不是 S3 对象。可以使用下面最接近的方法：`is.object(x) & !isS4(x)`，
102 确认 `x` 是对象但不是 S4。另一个简单方法是使用 `pryr::otype()`：

```
library(pryr)

df <- data.frame(x = 1:10, y = letters[1:10])
otype(df)    # A data frame is an S3 class
#> [1] "S3"
otype(df$x)  # A numeric vector isn't
#> [1] "base"
otype(df$y)  # A factor is
#> [1] "S3"
```

在 S3 中，方法属于函数，这个函数称为**泛型函数**（generic function）或者简称泛型。S3 方法不属于对象或类。这与其他大多数编程语言都不同，但它的确也是一种合法的 OO 风格。

为了知道一个函数是不是一个 S3 泛型函数，可以查看它的源代码，找到函数调用 `UseMethod()`：这个函数指出调用的正确方法，也就是**方法分派**（method dispatch）的过程。与 `otype()` 类似，`pryr` 还提供了 `ftype()` 函数，如果有与函数相关联的对象系统，它可以描述该系统：

```
mean
#> function (x, ...)
#> UseMethod("mean")
#> <bytecode: 0x7f96ac563c68>
#> <environment: namespace:base>
ftype(mean)
#> [1] "s3"      "generic"
```

与 [、Sum() 和 cbind() 类似，有些 S3 泛型函数不调用 UseMethod()，因为它们用 C 语言实现。作为代替，它们调用 C 函数 DispatchGroup() 或 DispatchOrEval()。在 C 代码中执行方法分派的函数称为**内部泛型**（internal generics），可以使用 ?"internal generic" 来查看它们的文档。ftype() 也对这些特殊情况进行描述。

假定有一个类，S3 泛型函数的任务就是调用正确的 S3 方法。根据名字就可以认别 S3 方法，它们看上去像 generic.class()。例如，泛型函数 mean() 的日期（Date）方法调用 mean.Date()，print() 的因子方法调用 print.factor()。

这就是大多数现代风格指南不支持在函数名中使用“.”的原因：这使它们看上去像 S3 方法。例如，t.test() 是对象 t 的 test 方法吗？同样，在类名中使用“.”也会造成迷惑：print.data.frame() 是 data.frames 的 print() 方法吗，或者 print.data() 是 frames 的方法？ pryr::ftype() 知道这些例外，所以可以使用它来检查一个函数是 S3 方法，还是 S3 泛型函数：

103

```
ftype(t.data.frame) # data frame method for t()
#> [1] "s3"     "method"
ftype(t.test)       # generic function for t tests
#> [1] "s3"      "generic"
```

可以使用 methods() 来查看属于一个泛型函数的所有方法：

```
methods("mean")
#> [1] mean.Date     mean.default  mean.difftime mean.POSIXct
#> [5] mean.POSIXlt
methods("t.test")
#> [1] t.test.default* t.test.formula*
#>
#>    Non-visible functions are asterisked
```

（除了基础包中定义的方法外，大部分 S3 方法是不可见的：使用 getS3method() 阅读它们的源代码。）

对一个给定的类，可以列出包含该类的方法的所有泛型函数：

```
methods(class = "ts")
#>  [1] [.ts*             [<-.ts*           aggregate.ts
#>  [4] as.data.frame.ts cbind.ts*         cycle.ts*
#>  [7] diff.ts*         diffinv.ts*       kernapply.ts*
#> [10] lines.ts*        monthplot.ts*     na.omit.ts*
#> [13] Ops.ts*          plot.ts           print.ts*
#> [16] t.ts*            time.ts*          window.ts*
#> [19] window<-.ts*
#>
#>    Non-visible functions are asterisked
```

在下一节你将知道没有方法可以列出 S3 的所有类。

104

7.2.2　定义类和创建对象

S3 是一个简单而又特别的系统；它没有正式的类定义。为给一个类创建一个对象实例，只需要使用已有的基础对象并设置类属性。在创建时可以使用 structure()，或者最后使用 class<-()：

```
# Create and assign class in one step
foo <- structure(list(), class = "foo")

# Create, then set class
foo <- list()
class(foo) <- "foo"
```

S3 对象通常建立在列表或带有属性的原子向量之上。(可以复习 2.2 节的属性知识。)也可以将一个函数转变成 S3 对象。其他基础类型要么在 R 中不常见，要么具有与属性不兼容的不常见的语义。

可以使用 `class(x)` 检查任意对象的属性，并可以使用 `inherits(x, "classname")` 来查看一个对象是否继承于一个特殊类。

```
class(foo)
#> [1] "foo"
inherits(foo, "foo")
#> [1] TRUE
```

一个 S3 对象的类可以是向量，这个向量描述从最具体到最一般的行为。例如，`glm()` 对象的类是 `c("glm", "lm")`，它表明广义线性模型是继承了线性模型的行为。类名通常是小写，避免使用“.”。否则，使用多个词对类命名时，又会陷入使用下划线（`my_class`）还是使用骆驼拼写法（`MyClass`）的争论之中。

大多数 S3 类都提供了一个构造函数：

```
foo <- function(x) {
  if (!is.numeric(x)) stop("X must be numeric")
  structure(list(x), class = "foo")
}
```

105

如果有构造函数（例如，`factor()` 和 `data.frame()`），应该使用它。它可以保证正在创建的类包含正确的组成成分。构造函数的名字通常与类的名字相同。

除了开发者提供构造函数外，S3 没有正确性检查。这就意味着我们可以改变已有对象的类：

```
# Create a linear model
mod <- lm(log(mpg) ~ log(disp), data = mtcars)
class(mod)
#> [1] "lm"
print(mod)
#>
#> Call:
#> lm(formula = log(mpg) ~ log(disp), data = mtcars)
#>
#> Coefficients:
#> (Intercept)    log(disp)
#>       5.381       -0.459

# Turn it into a data frame (?!)
class(mod) <- "data.frame"
# But unsurprisingly this doesn't work very well
print(mod)
#>  [1] coefficients  residuals     effects       rank
#>  [5] fitted.values assign        qr            df.residual
```

```
#>  [9] xlevels      call          terms         model
#> <0 rows> (or 0-length row.names)
# However, the data is still there
mod$coefficients
#> (Intercept)   log(disp)
#>      5.3810     -0.4586
```

如果使用过其他 OO 编程语言，这可能会让你觉得不舒服。但令人惊奇的是，这种灵活性并没有带来什么麻烦：虽然能改变对象的类型，但是永远不要这样做。R 保护不了我们：这会搬起石头砸自己的脚。如果不把枪对准自己并扣动扳机，永远都不会遇到问题。

7.2.3　创建新方法和泛型函数

为了添加一个新的泛型函数，创建一个调用 `UseMethod()` 的函数。`UseMethod()` 有两个参数：泛型函数的名字和方法分派的参数。如果忽略了第二参数，就会将第一个参数分派给函数。没有必要将泛型函数的所有参数都传递给 `UseMethod()`，也不应该这样做。`UseMethod()` 使用魔法来找到它们本身。

106

```
f <- function(x) UseMethod("f")
```

如果没有方法，泛型函数就没有用。为了添加方法，只需要创建一个具有正确（`generic.class`）名字的普通函数：

```
f.a <- function(x) "Class a"

a <- structure(list(), class = "a")
class(a)
#> [1] "a"
f(a)
#> [1] "Class a"
```

用同样的方式为一个已有的泛型函数添加一个新方法：

```
mean.a <- function(x) "a"
mean(a)
#> [1] "a"
```

如你所见，对于方法返回的类是否与泛型函数匹配没有做任何检查。是否要确保创建的新方法不会与已有的代码发生冲突完全取决于你自己。

7.2.4　方法分派

S3 方法分派非常简单。`UseMethod()` 创建一个由函数名构成的向量，类似于 `paste0("generic", ".", c(class(x), "default"))`，并且按顺序依次查找。默认类使我们可以为其他未知类建立一个回滚方法。

107

```
f <- function(x) UseMethod("f")
f.a <- function(x) "Class a"
f.default <- function(x) "Unknown class"
f(structure(list(), class = "a"))
#> [1] "Class a"
# No method for b class, so uses method for a class
f(structure(list(), class = c("b", "a")))
#> [1] "Class a"
```

```
# No method for c class, so falls back to default
f(structure(list(), class = "c"))
#> [1] "Unknown class"
```

组泛型函数方法具有更多的复杂性。组泛型函数可以仅使用一个函数就能执行多个泛型函数的方法。4 个组泛型函数以及它们所包含的函数是：

- Math：abs、sign、sqrt、floor、cos、sin、log、exp、...
- Ops：+、-、*、/、^、%%、%/%、&、|、!、==、!=、<、<=、>=、>
- Summary：all、any、sum、prod、min、max、range
- Complex：Arg、Conj、Im、Mod、Re

组泛型函数是相对高级的技术，已经超出了本章的范围，但是可以使用命令 `?groupGeneric` 来找到关于它们更多的相关内容。这里需要记住的最重要一点是：`Math`、`Ops`、`Summary` 和 `Complex` 并不是真正的函数，而是一组函数的代表。注意，在组泛型函数内有一个特殊的变量 `.Generic`，它提供了对实际泛型函数的调用。

如果你的类具有复杂的层次结构，那么对“父类”方法的调用有时很有用。很难准确地定义这到底意味着什么，但是如果没有当前方法那么它就是将要被调用的基本方法。同样，这也属于高级技术：可以使用命令 `?NextMethod` 来获取更多信息。

由于方法都是正常的 R 函数，所以可以直接调用它们：

```
c <- structure(list(), class = "c")
# Call the correct method:
f.default(c)
#> [1] "Unknown class"
# Force R to call the wrong method:
f.a(c)
#> [1] "Class a"
```

108 但是，这与修改对象的类一样危险，所以不应该这样做。不要用枪指着自己！唯一需要直接调用方法的原因是，跳过方法分派可以大大改进程序的性能。参见 17.5 节查看更多的知识。

还可以为非 S3 对象调用 S3 泛型函数。非内部 S3 泛型函数将在基础类型的**隐式类**（implicit class）上进行分派（出于性能方面的考虑，内置泛型函数不会这样做）。决定基础类型的隐式类的规则有点儿复杂，下面的函数中说明了这一点：

```
iclass <- function(x) {
  if (is.object(x)) {
    stop("x is not a primitive type", call. = FALSE)
  }

  c(
    if (is.matrix(x)) "matrix",
    if (is.array(x) && !is.matrix(x)) "array",
    if (is.double(x)) "double",
    if (is.integer(x)) "integer",
    mode(x)
  )
}
iclass(matrix(1:5))
#> [1] "matrix"  "integer" "numeric"
iclass(array(1.5))
#> [1] "array"   "double"  "numeric"
```

7.2.5 练习

1. 通过阅读 t() 和 t.test() 的源代码来确定 t.test() 是 S3 泛型函数而不是 S3 方法。如果使用 test 类创建一个对象并调用 t() 会发生什么？
2. 在 R 的基础包中哪些类具有 Math 组泛型函数的方法？阅读源代码看看它们是如何工作的？
3. R 有两个可以表示日期和时间的类，POSIXct 和 POSIXlt，它们都继承了 POSIXt。对于这两个类，哪些泛型函数有不同的行为？哪些具有相同的行为？
4. 哪个基础泛型函数具有最多的已定义方法？ 109
5. UseMethod() 使用一种特殊的方式调用方法。预测下面代码的返回值，然后运行它，阅读 UseMethod() 的帮助文档弄清楚到底发生了什么。用尽可能简单的方式写下该函数的规则。

```
y <- 1
g <- function(x) {
  y <- 2
  UseMethod("g")
}
g.numeric <- function(x) y
g(10)

h <- function(x) {
  x <- 10
  UseMethod("h")
}
h.character <- function(x) paste("char", x)
h.numeric <- function(x) paste("num", x)

h("a")
```

6. 内置泛型函数不会在基础类型的隐式类上进行分派。认真阅读文档（?"internal generic"），并确定为什么下面代码中的 f 和 g 的长度不同。哪些函数可以帮助你区分 f 和 g 的行为？

```
f <- function() 1
g <- function() 2
class(g) <- "function"

class(f)
class(g)

length.function <- function(x) "function"
length(f)
length(g)
```

110

7.3 S4

S4 的工作方式与 S3 类似，但是它更正式和严谨。方法仍然属于函数而不是类，但是：

- 类有描述它们的字段（field）和继承关系结构（父类）的正式定义。
- 可以基于一个泛型函数的多个参数进行方法分派。
- 有一个特殊运算符 @，它可以用来从 S4 对象中提取字段（slot）。

所有与 S4 相关的方法都存储在 methods 包中。当你交互运用 R 时，这个包总是可用的，但是当批处理运行 R 时，它可能就不可用。为此，当使用 S4 时，最好显式地包括 library(methods)。

S4 是一个丰富且复杂的系统。不可能仅用几页篇幅就对它做出详细的解释。这里主要集中在 S4 的关键概念上，这样就可以更加高效地使用已有的 S4 对象。要学习更多的知识，可以参考：

- S4 系统在 Bioconductor 中的发展（http://www.bioconductor.org/help/course-materials/2010/AdvancedR/S4InBioconductor.pdf）。
- John Chambers 的《Software for Data Analysis》（http://amzn.com/0387759352?tag=devtools-20）。
- stackoverflow 上 Martin Morgan 对 S4 问题的回答（http://stackoverflow.com/searth?tab=votes&q=user%3a547331%20%5bs4%5d%20is%3aanswe）。

7.3.1 识别对象、泛型函数和方法

识别 S4 的对象、泛型函数和方法很容易。对一个 S4 对象使用 `str()` 函数，返回值为
111 “formal”类，调用 `isS4()` 函数，返回值为 `TRUE`，而 `pryr::otype()` 的返回值为“S4”。S4 泛型函数和方法也容易识别，因为它们都是具有良好定义的 S4 对象。

在经常使用的基础包（`stats`、`graphics`、`utils`、`datasets` 和 `base`）中，基本上没有 S4 类，所以首先要从内置的 `stats4` 软件包创建一个 S4 对象，这个软件包提供了一些与最大似然估计相关的类和方法：

```
library(stats4)

# From example(mle)
y <- c(26, 17, 13, 12, 20, 5, 9, 8, 5, 4, 8)
nLL <- function(lambda) - sum(dpois(y, lambda, log = TRUE))
fit <- mle(nLL, start = list(lambda = 5), nobs = length(y))

# An S4 object
isS4(fit)
#> [1] TRUE
otype(fit)
#> [1] "S4"

# An S4 generic
isS4(nobs)
#> [1] TRUE
ftype(nobs)
#> [1] "s4"      "generic"

# Retrieve an S4 method, described later
mle_nobs <- method_from_call(nobs(fit))
isS4(mle_nobs)
#> [1] TRUE
ftype(mle_nobs)
#> [1] "s4"     "method"
```

使用一个参数的 `is()` 函数列出继承的所有对象；使用两个参数的 is() 检测一个对象是否继承了某个具体类。

```
is(fit)
#> [1] "mle"
is(fit, "mle")
#> [1] TRUE
```

可以使用 getGenerics() 获得所有 S4 泛型函数的列表；使用 getClasses() 获得所有 S4 类的列表。这个列表包含了 S3 类和基础类型的 shim 类。可以使用 showMethods() 列出所有 S4 方法，还可以通过设置 generic 或 class（或者两者都选）来选择性地列出这些方法。还可以提供参数 where = search() 来限制在全局环境中使用的搜索方法。 112

7.3.2 定义类并创建对象

在 S3 中，可以通过设置类属性将任意对象转变成另一个特定类的对象。S4 更严格：必须使用 setClass() 来定义一个类的表示，使用 new() 来创建一个新对象。可以使用下面这个特殊的语法来找到一个类的文档：class?className，例如，class?mle。

S4 类有三个关键性质：

- **名字**（name）：一个由字母和数字组成的标识符。根据惯例，S4 类名使用开头字母大写的驼峰拼法。
- 带有名字的**字段**（slot）列表。它定义字段名和允许的类。例如，一个 Person 类可以使用字符型的名字和数值型的年龄表示：list(name = "character", age = "numeric")。
- 一个用来描述它的父类的字符串，或者用 S4 的术语，它**包含**（contain）的类。可以提供多个类实现多重继承，但是该高级技术会增加复杂性。

在 slots 和 contains 中，可以使用 setOldClass() 注册的 S4 类、S3 类，或者基础类型的隐式类。在 slots 中，还可以使用特殊类 ANY，它不会对输入进行限制。

S4 类还有其他一些可选性质，如 validity 方法可以用来测试一个对象是否有效，prototype 对象可以用来定义默认字段值。使用 ?setClass 可以查看更多的细节。

下面的例子创建了一个包含名字和年龄字段的 Person 类，以及一个继承 Person 类的 Employee 类。Employee 类继承了 Person 类的字段和方法，并增加了一个新字段 boss。为了创建新对象，用类的名字和字段值的名字 – 值对来调用 new()。

```
setClass("Person",
  slots = list(name = "character", age = "numeric"))
setClass("Employee",
  slots = list(boss = "Person"),
  contains = "Person")

alice <- new("Person", name = "Alice", age = 40)
john <- new("Employee", name = "John", age = 20, boss = alice)
```

113

大部分 S4 类都有一个与类名相同的构造函数：如果存在构造函数，那么可以使用它而不用直接调用 new()。

获取 S4 对象的字段可以使用 @ 或 slot()：

```
alice@age
#> [1] 40
slot(john, "boss")
#> An object of class "Person"
#> Slot "name":
#> [1] "Alice"
#>
#> Slot "age":
#> [1] 40
```

(@等价于$，而slot()等价于[[。)

如果一个S4对象继承自一个S3类或者一个基础类型，它就包含一个特殊的.Data字段，这个字段包含底层基础类型或S3对象：

```
setClass("RangedNumeric",
  contains = "numeric",
  slots = list(min = "numeric", max = "numeric"))
rn <- new("RangedNumeric", 1:10, min = 1, max = 10)
rn@min
#> [1] 1
rn@.Data
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

因为R是一个交互式编程语言，所以它可以随时创建一个新类或者重新定义现有类。当你正在与S4进行交互式时，这可能产生问题。如果你对一个类进行修改，一定要重新创建
114 这个类的所有对象，否则就会以无效对象终止。

7.3.3 创建新方法和泛型函数

S4为创建新的泛型函数和方法提供了特殊函数。setGeneric()可以创建新的泛型函数或者将已有的函数转换成泛型函数。setMethod()的参数包括：泛型函数的名字、与该方法关联在一起的类、执行方法的函数。例如，我们可以把通常只在向量上使用的union()应用在数据框上：

```
setGeneric("union")
#> [1] "union"
setMethod("union",
  c(x = "data.frame", y = "data.frame"),
  function(x, y) {
    unique(rbind(x, y))
  }
)
#> [1] "union"
```

如果从头创建一个新的泛型函数，需要为它提供一个调用standardGeneric()的函数：

```
setGeneric("myGeneric", function(x) {
  standardGeneric("myGeneric")
})
#> [1] "myGeneric"
```

S4中的standardGeneric()相当于UseMethod()。

7.3.4 方法分派

如果S4泛型函数对一个仅有一个父类的类进行分派，那么S4方法分派与S3分派方法类似。它们的主要不同是如何设置默认值：S4使用特殊类ANY来匹配任何类，使用“missing”来匹配缺失参数。与S3类似，S4也有组泛型函数，可以使用?S4groupGeneric获得相关文档，调用“父类”方法的方式是callNextMethod()。

如果要对多个参数进行分派或者你的类使用了多重继承，那么方法分派就变得更加复
115 杂。使用?Methods可以找到分派的规则，但是规则非常复杂，也很难预测它将调用哪个方

法。正是由于这个原因，我强烈建议除非必须，否则不要使用多重继承和多重分派。

最后，给定一个泛型函数调用的规范，有两种方法可以找到哪个方法被调用了：

```
# From methods: takes generic name and class names
selectMethod("nobs", list("mle"))

# From pryr: takes an unevaluated function call
method_from_call(nobs(fit))
```

7.3.5 练习

1. 哪个 S4 泛型函数具有最多的方法？哪个 S4 类具有最多的方法？
2. 如果定义了一个不包含已有类的新的 S4 类，将会发生什么？（提示：阅读命令 `?Classes` 给出的文档中的虚拟类部分。）
3. 如果将一个 S4 对象传递给一个 S3 泛型函数将会发生什么？如果将一个 S3 对象传递给一个 S4 泛型函数呢？（提示：对于第二问题，可以使用 `?setOldClass` 阅读相关文档。）

7.4 RC

参考类（RC）是 R 中最新的 OO 系统。它们是在 2.12 版本中提出的。它们与 S3 和 S4 有根本上的区别，因为：

- RC 方法属于对象而不是函数。
- RC 对象是可变的：不适用于通常 R 的复制后修改（copy-on-modify）语义。

这些性质使 RC 对象与其他编程语言（例如，Python、Ruby、Java 和 C#）的对象更加类似。参考类是使用 R 代码实现的：它们是包含环境的一个特殊的 S4 类。 116

7.4.1 定义类和创建对象

由于 R 的基础包中没有任何 RC 类，所以首先创建一个。RC 类最适合用来描述状态对象，对象随着时间发生改变，所以创建一个对银行账户建模的简单类。

创建一个新 RC 类与创建一个新 S4 类类似，但使用 `setRefClass()` 而不是使用 `setClass()`。第一个，也是唯一一个必需的参数，就是一个由字母和数字构成的**名字**（name）。虽然可以使用 `new()` 创建一个新的 RC 类，但是最好使用 `setRefClass()` 返回的对象来产生一个新对象。（在 S4 中也可以这样做，但是不常见。）

```
Account <- setRefClass("Account")
Account$new()
#> Reference class object of class "Account"
```

`setRefClass()` 还接收一个定义类**字段**（相当于 S4 字段）的名字 - 类对的列表。传递给 `new()` 的附加名字参数将设置该字段的初始值。使用 `$` 可以获取和设置某个字段的值：

```
Account <- setRefClass("Account",
  fields = list(balance = "numeric"))

a <- Account$new(balance = 100)
a$balance
#> [1] 100
a$balance <- 200
a$balance
#> [1] 200
```

不给字段提供类名，可以提供一个作为存取函数的单个参数函数。这允许你在获取或设置一个字段时可以添加一些自定义的行为。使用 `?setRefClass` 可以查看更多细节。

注意 RC 对象是**可变的**（mutable），即它们有引用语义，不是复制后修改：

117
```
b <- a
b$balance
#> [1] 200
a$balance <- 0
b$balance
#> [1] 0
```

正是由于这个原因，所以 RC 对象有一个 `copy()` 方法，它允许复制对象：

```
c <- a$copy()
c$balance
#> [1] 0
a$balance <- 100
c$balance
#> [1] 0
```

没有用**方法**定义某些行为，那么这个对象就不太有用。RC 方法是与类相关联的，并且可以在原地对其字段进行修改。在下面的例子中，注意通过名字来访问字段的值，使用“`<<-`”对其进行修改。可以在 8.4 节找到更多的相关知识。

```
Account <- setRefClass("Account",
  fields = list(balance = "numeric"),
  methods = list(
    withdraw = function(x) {
      balance <<- balance - x
    },
    deposit = function(x) {
      balance <<- balance + x
    }
  )
)
```

可以采用与访问字段相同的方式调用 RC 方法。

```
a <- Account$new(balance = 100)
a$deposit(100)
a$balance
#> [1] 200
```

118 `setRefClass()` 的最后一个重要参数为 `contains`。它给出当前类所继承的父 RC 类。下面的例子创建了一个新类型的银行账户，当存款值低于 0 时它返回一个错误。

```
NoOverdraft <- setRefClass("NoOverdraft",
  contains = "Account",
  methods = list(
    withdraw = function(x) {
      if (balance < x) stop("Not enough money")
      balance <<- balance - x
    }
  )
)
accountJohn <- NoOverdraft$new(balance = 100)
accountJohn$deposit(50)
```

```
accountJohn$balance
#> [1] 150
accountJohn$withdraw(200)
#> Error: Not enough money
```

所有的 RC 类最终都是继承自 `envRefClass`。它提供了一些很有用的方法，例如，`copy()`（前面已经讲过）、`callSuper()`（调用父类字段）、`field()`（根据名字获取某个字段的值）、`export()`（等价于 `as()`）以及 `show()`（控制输出）。在 `setRefClass()` 中查看有关继承的相关细节。

7.4.2　识别类和方法

由于 RC 对象是继承自“refClass”（`is(x, "refClass")`）的 S4 对象（`isS4(x)`），所以可以很容易地识别它们。`pryr::otype()` 将返回“RC”。RC 方法也是 S4 对象，它的类是 `refMethodDef`。

7.4.3　方法分派

由于 RC 中的方法与类相关联而不与函数相关联，所以方法分派也很简单。当调用 `x$f()` 时，R 将在 `x` 类中查找方法 `f`，然后在它的父类中查找，在父类的父类中查找，等等。从一个方法中，可以使用 `callSuper(...)` 直接调用父类的方法。 119

7.4.4　练习

1. 使用一个字段函数来避免账户余额被直接修改。（提示：创建一个隐藏的 `.balance` 字段，并阅读 `setRefClass()` 文档中关于字段参数的部分。）
2. 我在前面声称 R 的基础包中没有 RC 类，但这过于简单化了。使用 `getClasses()` 并找到哪个类是从 `envRefClass` 扩展来的（`extand()`）。这个类是用来做什么的？（提示：回忆如何查找一个类的文档。）

7.5　选择一个系统

对于一个语言来说，3 个 OO 系统实在是太多了，对于大多数 R 编程任务来讲，S3 已经够了。在 R 中，经常要为已经存在的泛型函数（例如，`print()`、`summary()` 和 `plot()`）创建非常简单的对象和方法。S3 非常适合这样的工作，我使用 S3 编写大多数 OO 代码。虽然 S3 有点古怪，但是有了它之后我们就可以使用最短的代码来完成任务。

如果正在为相互关联的对象创建更加复杂的系统，S4 可能更合适。一个很好的例子是由 Douglas Bates 和 Martin Maechler 编写的 `Matrix` 包。设计这个包的目的是更有效地存储和计算多种不同类型的稀疏矩阵。在 1.1.3 版本中，它定义了 102 个类和 20 个泛型函数。这个包写得很好，注释也很到位，附带的文档（`vignette("Intro2Matrix", package = "Matrix")`）对整个包的结构做了很好的概述。在 `Bioconductor` 包（需要对生物对象之间的复杂关系建模）中也大量地使用了 S4。`Bioconductor` 也为学习 S4 提供了很多好资源（https://www.google.com/search?q=bioconductor+s4）。如果掌握了 S3，S4 就相对容易，它们的思想是一样的，只是 S4 更正式、更严格、更详细一些。

如果你曾经使用过主流的 OO 编程语言，RC 看上去可能会更容易接受一些。但是，由

于它们的可变状态会引入副作用，所以它变得很难理解。例如，通常情况下当你在 R 中调用 `f(a, b)` 时，你假设 `a` 和 `b` 是不会被改变的。但是如果 `a` 和 `b` 是 RC 对象，它们就可能会被
120 改变。所以只有在必须使用可变状态时，才使用 RC 对象，这样才能将副作用控制到最小。大部分函数还是应该保持“函数化”，并且要避免副作用。这就会使我们的代码变得很容易推理，且很容易被其他 R 程序员理解。

7.6 答案

1）为了确定一个对象的 OO 系统，可以使用排除法。如果 `!is.object(x)` 返回 `TRUE`，那么它是基础对象。如果 `!isS4(x)` 返回 `TRUE`，那么它是 S3。如果 `!is(x, "refClass")` 返回 `TRUE`，那么它是 S4；否则它是 RC。

2）使用 `typeof()` 确定一个对象的基础类。

3）泛型函数根据输入的类调用具体方法。在 S3 和 S4 对象系统中，方法属于泛型函数而不属于类，这与其他编程语言不同。

121 ~ 122 4）S4 比 S3 更正式，并且它支持多重继承和多重调用。RC 对象具有引用语义，并且方法属于类而不属于函数。

第 8 章
环　境

环境就是作用域发挥作用的数据结构。本章将深入学习环境，详细介绍它们的结构，借助它们来帮助你理解 6.2 节中描述的 4 种作用域规则。

由于环境具有引用语义，所以它们本身也是一种很有用的数据结构。当在一个环境中对其绑定的元素进行修改时，环境不会被复制，修改会在原地进行。虽然并不需要经常使用引用语义，但它还是非常有用的。

小测验

如果你能正确地回答下列问题，说明你已经掌握了本章的重点知识。在 8.6 节中可以找到答案。

1）至少从 3 个方面描述环境与列表的不同。

2）全局环境的父环境是什么？没有父环境的唯一环境是什么？

3）函数的封闭环境是什么？为什么它很重要？

4）如何确定函数被调用的环境？

5）`<-` 和 `<<-` 的不同是什么？

主要内容

- 8.1 节介绍环境的基本性质以及如何创建自己的环境。
- 8.2 节提供了一个可以对环境进行计算的函数模板，用一个很有用的函数来说明这个思想。
- 8.3 节更深入地介绍 R 的作用域规则，说明它们如何对应于与每个函数相关联的 4 种环境类型。
- 8.4 节介绍命名必须遵守的一些规则（以及如何进行绑定），以及其他一些绑定方法。
- 8.5 节讨论环境本身作为一种数据结构可以解决的 3 个问题，它们独立于环境空间在作用域中所扮演的角色。

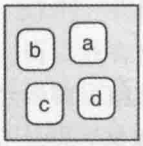
预备工具

本章会使用 `pryr` 添加包中的多个函数来帮助我们深入剖析 R，并让我们能够查看其中的细节。可以运行 `install.packages("pryr")` 来安装 `pryr` 添加包。

8.1　环境基础

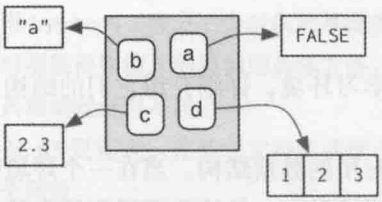
环境的作用就是将一些名字与一些值进行关联，或者**绑定**（bind）。可以把环境看作一个

装满名字的口袋：



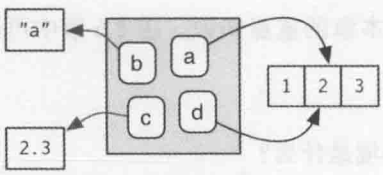
每个名字都指向存储在内存中的一个对象：

```
e <- new.env()
e$a <- FALSE
e$b <- "a"
e$c <- 2.3
e$d <- 1:3
```



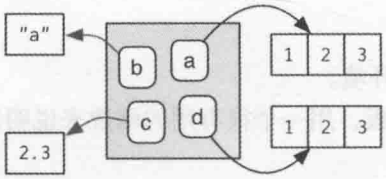
对象并不生存在环境中，所以多个名字可以指向同一个对象：

```
e$a <- e$d
```



令人困惑的是，它们还可以指向具有相同值的不同对象：

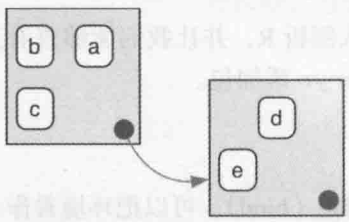
```
e$a <- 1:3
```



124 ~ 125

如果对象没有指向它的名字，那么这个对象就会被垃圾回收器（garbage collector）自动删除。将在 18.2 节详细讲解这个过程。

每个环境都有父环境，它是另一个环境。在下图中，黑色小圆球代表指向父环境的指针。父环境用于实现词法作用域：如果一个名字在一个当前环境中没有找到，R 就会到它的父环境中寻找（直到找到或者找遍所有环境）。只有**空**（empty）环境没有父环境。



我们可以将环境之间的关系比作家庭成员之间的关系。一个环境的爷爷就是它父亲的父亲，它的祖先就包括直到空环境的所有父环境。我们基本上不会说一个环境的子环境，因为它们之间没有反向链接：给定一个环境我们没有办法找到它的子环境。

通常，环境与列表相似，除了以下 4 点外：

- 环境中的每个对象都有一个唯一的名字。
- 环境中的对象是没有顺序的（即，在环境中查找第一个对象是没有意义的）。
- 环境有父环境。
- 环境具有引用语义。

更专业一点儿，环境是由两部分构成的：**对象框**（frame），它包含名称 - 对象的绑定关系（行为上更像一个命名列表）；它的父环境。不幸的是，在 R 中“对象框”的概念是不一致的。例如，`parent.frame()` 并不是给出一个环境的父环境，而是给出正在调用的环境。我们将在 8.3.4 节详细讨论它。

还有 4 个特殊环境：

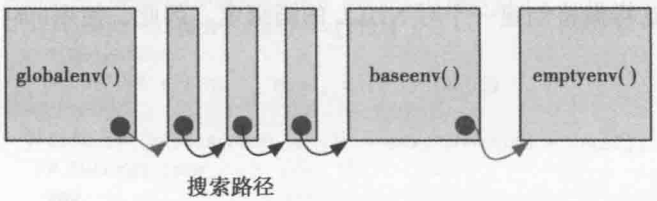
- `globalenv()`，或者全局环境，它是一个交互式的工作空间。通常情况下我们就是在这个环境中工作。全局环境的父环境是由 `library()` 或 `require()` 添加的最后一个添加包。
- `baseenv()`，基础环境，它是 R 基础软件包的环境。它的父环境是空环境。
- `emptyenv()`，空环境，它是所有环境的祖先，也是唯一一个没有父环境的环境。
- `environment()`，它是当前环境。

`search()` 可以列出全局环境的所有父环境。这称为搜索路径，因为这些环境中的对象都可以在顶层交互式工作空间中找到。它包含每一个已经添加的添加包的环境空间和应用 `attach()` 函数添加的对象。它还包含一个称为 `Autoloads` 的特殊环境，它可以在需要时通过只加载添加包对象来减少内存的使用。

你可以用 `as.environment()` 访问搜索列表中的任何环境。

```
search()
#> [1] ".GlobalEnv"        "package:stats"     "package:graphics"
#> [4] "package:grDevices" "package:utils"     "package:datasets"
#> [7] "package:methods"   "Autoloads"         "package:base"

as.environment("package:stats")
#> <environment: package:stats>
```

`globalenv()`、`baseenv()` 搜索路径上的环境和 `emptyenv()` 的关系如下图所示。每次当你使用 `library()` 加载新的添加包时，将该添加包就插入全局环境空间与先前位于搜索路径顶端的添加包之间。



使用 `new.env()` 可以手动创建一个环境。使用 `ls()` 可以将此环境的对象框中的所有绑定关系列出来，可以使用 `parent.env()` 查看它的父环境。

```
e <- new.env()
# the default parent provided by new.env() is environment from
# which it is called - in this case that's the global environment.
parent.env(e)
#> <environment: R_GlobalEnv>
ls(e)
#> character(0)
```

对一个环境中的绑定关系进行修改的最简单方法就是将其看作列表：

```
e$a <- 1
e$b <- 2
ls(e)
#> [1] "a" "b"
e$a
#> [1] 1
```

默认情况下，`ls()` 只能列出不是以“.”开始的名字。可以通过设置参数 `all.names = TRUE` 来显示一个环境中的所有绑定关系：

```
e$.a <- 2
ls(e)
#> [1] "a" "b"
ls(e, all.names = TRUE)
#> [1] ".a" "a"  "b"
```

查看环境的另一个有用方法就是 `ls.str()`。由于它可以将环境中的所有对象都显示出来，所以它比 `str()` 更有用。与 `ls()` 一样，它也有一个 `all.names` 参数。

```
str(e)
#> <environment: 0x7fdd1d4cff10>
ls.str(e)
#> a :  num 1
#> b :  num 2
```

给定一个名字，可以使用 `$`、`[[` 或 `get()` 来获取与其绑定的值：

128

- `$` 和 `[[` 只在一个环境中进行查找，如果不存在与该名字绑定的值它就返回 `NULL`。
- `get()` 使用普通的作用域法则，如果没有找到绑定它就会抛出一个错误。

```
e$c <- 3
e$c
#> [1] 3
e[["c"]]
#> [1] 3
get("c", envir = e)
#> [1] 3
```

从环境中删除对象与从列表中删除对象有些不同。在列表中可以通过将其设置为 `NULL` 来删除一个表项。而在环境中，这样做将创建一个对 NULL 的新绑定。因此，使用 `rm()` 来删除绑定。

```
e <- new.env()

e$a <- 1
e$a <- NULL
ls(e)
#> [1] "a"
```

```
rm("a", envir = e)
ls(e)
#> character(0)
```

可以使用 `exists()` 来确定一个绑定是否存在。与 `get()` 一样，它的默认行为是按照普通的作用域法则在其父环境中查找。如果你不希望在父环境中查找，可以设置参数 `inherits = FALSE`：

```
x <- 10
exists("x", envir = e)
#> [1] TRUE
exists("x", envir = e, inherits = FALSE)
#> [1] FALSE
```

使用 `identical()` 而不是 == 对两个环境进行比较：

```
identical(globalenv(), environment())
#> [1] TRUE
globalenv() == environment()
#> Error: comparison (1) is possible only for atomic and list
#> types
```

练习

1. 从 3 个方面列出环境与列表的不同。
2. 如果没有提供一个显式的环境，`ls()` 和 `rm()` 在哪里查找？“<-” 在哪里绑定？
3. 使用 `parent.env()` 和循环（或者一个递归函数）来证明 `globalenv()` 的祖先包括 `baseenv()` 和 `emptyenv()`。使用相同的思想来创建一个你自己的 `search()`。

8.2 环境递归

环境可以构成一棵树，因此我们可以非常方便地写出一个递归函数。本节说明应用环境知识来查看 `pryr::where()` 是如何工作的。给定一个名字，`where()` 就会使用 R 的作用域法则找到定义这个名字的环境：

```
library(pryr)
x <- 5
where("x")
#> <environment: R_GlobalEnv>
where("mean")
#> <environment: base>
```

`where()` 的定义是很直观的。它有两个参数：要查找的名字（一个字符串）；开始查找的环境。（在 8.3.4 节中我们将学习为什么默认从 `parent.frame()` 开始。）

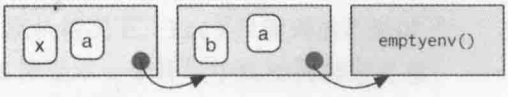
```
where <- function(name, env = parent.frame()) {
  if (identical(env, emptyenv())) {
    # Base case
    stop("Can't find ", name, call. = FALSE)

  } else if (exists(name, envir = env, inherits = FALSE)) {
    # Success case
    env

  } else {
```

```
    # Recursive case
    where(name, parent.env(env))

  }
}
```

有 3 种情况：

- 基本情况：已经到达空环境，但没找到绑定。因为不能继续搜索，所以抛出一个错误。
- 成功情况：在这个环境中，存在该名字，所以返回这个环境。
- 递归情况：在这个环境中没有找到该名字，所以尝试到它的父环境中继续查找。

通过一个例子就可以很容易地理解这个过程。假设你有下图所示的两个环境：



- 如果你正在查找 a，where() 将在第一个环境中找到它。
- 如果你正在查找 b，由于它不在第一个环境中，所以 where() 将在它的父环境中继续查找并找到它。
- 如果你正在查找 c，它既不在第一个环境中，也不在第二个环境中，所以 where() 就会到达空环境中并抛出错误信息。

129 ~ 131

递归地使用环境是很自然的，所以 where() 为我们提供了一个有用的模板。去掉 where() 函数中的细节，整个结构看起来会更清晰。

```
f <- function(..., env = parent.frame()) {
  if (identical(env, emptyenv())) {
    # base case
  } else if (success) {
    # success case
  } else {
    # recursive case
    f(..., env = parent.env(env))
  }
}
```

循环与递归

可以使用循环而不使用递归。这样运行起来可能更快一些（因为减少了一些函数调用），但是理解起来也会更困难一些。如果你对递归函数不熟悉，你可能觉得循环更容易理解，所以本节还是简单介绍循环。

```
is_empty <- function(x) identical(x, emptyenv())

f2 <- function(..., env = parent.frame()) {
  while(!is_empty(env)) {
    if (success) {
      # success case
      return()
    }
    # inspect parent
    env <- parent.env(env)
```

```
    }

    # base case
  }
```

练习

1. 对 `where()` 进行修改以便找到包含某个 `name` 的绑定关系的所有环境。
2. 采用 `where()` 函数的编写风格编写一个自己的 `get()` 函数。 132
3. 编写一个称为 `fget()` 的函数，它只寻找函数对象。它应该有两个参数：`name` 和 `env`，并且要遵守函数的普通作用域法则：如果匹配的对象不是函数，就到父环境中继续查找。为了增加一点儿难度，可以添加一个 `inherits` 参数，它控制是否到父环境中进行递归查找还是只在一个环境中查找。
4. 自己写一个 `exists(inherits = FALSE)` 函数（提示：使用 `ls()`。）写一个与 `exists(inherits = TRUE)` 形为类似的递归版本。

8.3 函数环境

大多数环境并不是通过 `new.env()` 创建的，而是使用函数的结果。本节讨论 4 种与函数相关的环境：封闭、绑定、执行和调用。

封闭（enclosing）环境就是创建函数的环境。每个函数有且仅有一个封闭环境。对于其他三种类型的环境，每个函数可以有 0 个、1 个或多个相关联的环境：

- 使用 `<-` 将一个函数与一个名字进行绑定，这样就可以定义一个**绑定**（binding）环境。
- 调用函数创建一个临时**执行**（execution）环境，它用来存储执行期间创建的各种变量。
- 每一个执行环境都与一个**调用**（calling）环境相关联，它说明函数在哪里调用。

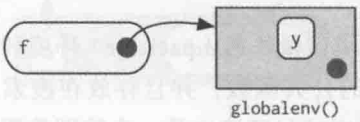
下面各节将解释为什么这些环境很重要，如何对它们进行访问以及如何使用它们。

8.3.1 封闭环境

当创建一个函数时，它就获得对创建它的环境的引用。这就是**封闭环境**（enclosing environment），它用作词法作用域。为了确定一个函数的封闭环境，只需要调用 `environment()`，并将函数名作为第一个参数。 133

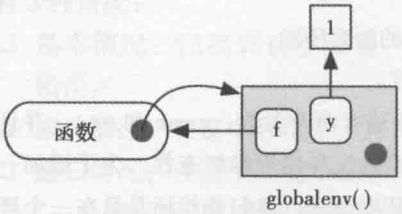
```
y <- 1
f <- function(x) x + y
environment(f)
#> <environment: R_GlobalEnv>
```

在下图中，使用圆角矩形来代表函数。黑色小圆代表一个函数的封闭环境。



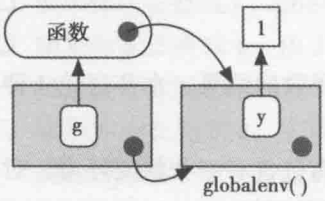
8.3.2 绑定环境

上图太简单了，因为函数没有名字。函数的名字可以通过绑定来定义。一个函数的绑定环境就是与其绑定的所有环境。下图可以更好地反映这种关系，因为封闭环境包含一个从 f 到函数的绑定：



在这种情况下，封闭环境和绑定环境是相同的。当将一个函数分配给另一个不同的环境，那么它们就不同了。

```
e <- new.env()
e$g <- function() 1
```



封闭环境属于该函数，永远都不会发生改变，甚至将该函数移动到不同的环境中。封闭环境决定了这个函数如何找到值，而绑定环境空间决定如何找到函数。

绑定环境与封闭环境的区别对于软件包命名空间是非常重要的。软件包命名空间使软件包之间保持独立。例如，如果软件包 A 使用基础包中的 mean() 函数，那么如果软件包 B 也创建了它自己的 mean() 函数会有什么后果呢？命名空间确保软件包 A 能够继续使用基础包中的 mean() 函数而不受软件包 B 的影响（除非显式地调用）。

命名空间使用环境来实现，利用函数不一定存在于它们自己的封闭环境中的事实。例如，基础包中的 sd() 函数，它的封闭环境与绑定环境是不同的：

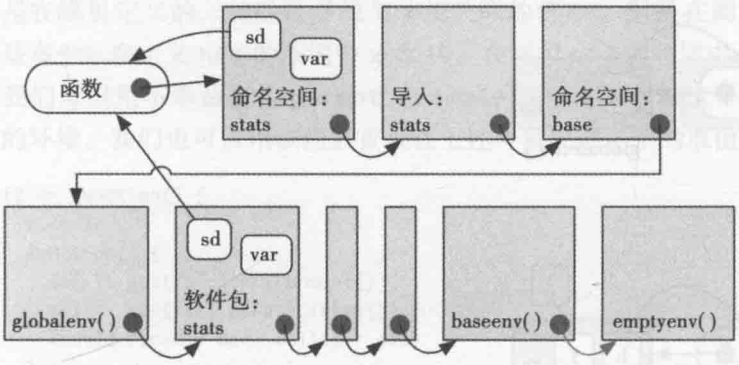
```
environment(sd)
#> <environment: namespace:stats>
where("sd")
# <environment: package:stats>
```

函数 sd() 的定义使用 var()，但如果创建自己的 var() 函数，那么它也不会影响 sd()：

```
x <- 1:10
sd(x)
#> [1] 3.028
var <- function(x, na.rm = TRUE) 100
sd(x)
#> [1] 3.028
```

这是可行的，因为每个软件包有两个与它相关联的环境：软件包（package）环境和命名空间（namespace）环境。软件包环境包含所有可以访问的公共函数，并且存放在搜索路径上。命名空间环境包含所有的函数（包括内部函数），并且它的父环境也是一个特别重要的环境，其中包含了这个软件包需要的所有函数的绑定。软件包中的每一个导出函数都绑定到软

件包环境，但都在命名空间环境中。可以使用下图来展现这种复杂的关系：



当给控制台输入 var 时，R 首先在全局环境中进行查找。当 sd() 查找 var() 时，它首先到命名空间环境中查找，并且永远不会在 globalenv() 中查找。

8.3.3 执行环境

第一次执行下面的函数时它返回什么？第二次呢？

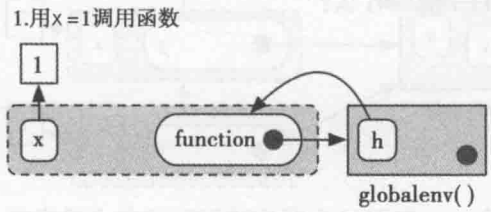
```
g <- function(x) {
  if (!exists("a", inherits = FALSE)) {
    message("Defining a")
    a <- 1
  } else {
    a <- a + 1
  }
  a
}
g(10)
g(10)
```

每次调用这个函数都返回相同的值，这就是 6.2.3 节描述的重新开始原则。每次调用函数时，都创建一个新的宿主执行环境。执行环境的父环境就是函数的封闭环境。一旦函数执行结束，这个环境就会被销毁。

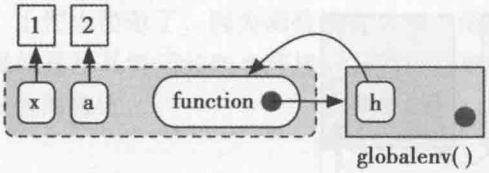
134
~
136

结合一个简单函数来对这个场景进行描述。在下图中，被虚线框包围的围绕函数的就是该函数的执行环境。

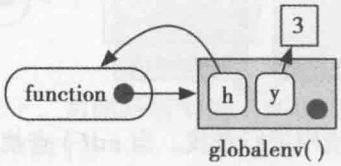
```
h <- function(x) {
  a <- 2
  x + a
}
y <- h(1)
```



2. a被赋值为2

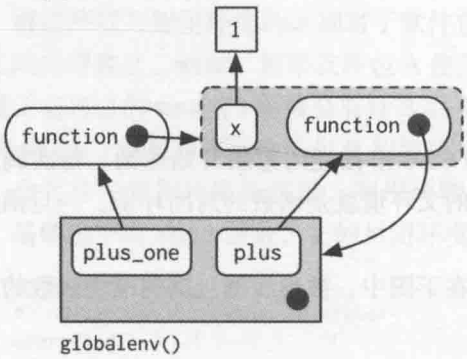


3.函数完成返回值为3，
扫行环境销毁



当在另一个函数中创建一个函数时，子函数的封闭环境就是父函数的执行环境，而且执行环境也不再是临时的了。下面的例子用一个函数工厂 `plus()` 来说明这个想法。我们使用这个工厂创建一个称为 `plus_one()` 的函数。`plus_one()` 的封闭环境是 `plus()` 的执行环境，其中 `x` 与数值 1 绑定。

```
plus <- function(x) {
  function(y) x + y
}
plus_one <- plus(1)
identical(parent.env(environment(plus_one)), environment(plus))
#> [1] TRUE
```



我们将在第 10 章中学习更多关于函数工厂的知识。

8.3.4 调用环境

查看下面的代码。当代码运行时，你期望 `i()` 返回什么？

```
h <- function() {
  x <- 10
  function() {
    x
  }
}
i <- h()
x <- 20
i()
```

137 ~ 138

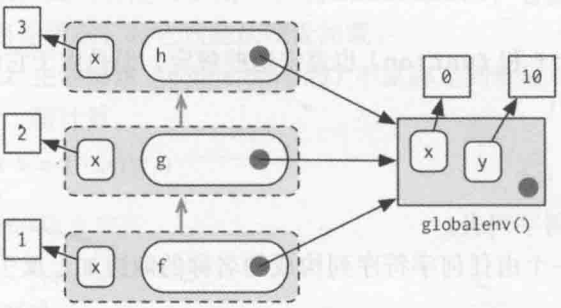
最外层的 x（绑定到 20）是为了分散你的注意力：使用普通作用域法则，h() 首先找到自己是在哪里定义的，然后再寻找与 x 相关联的值 10。但是在调用 i() 的环境中与 x 相关的值是多少？在定义 h() 的环境中 x 为 10，在调用 h() 的环境中 x 为 20。

我们可以用不幸命名为 parent.frame() 的函数来获取这个环境。该函数返回函数被调用的环境。我们也可以用该函数查找在上述环境中的名字的取值。

```
f2 <- function() {
  x <- 10
  function() {
    def <- get("x", environment())
    cll <- get("x", parent.frame())
    list(defined = def, called = cll)
  }
}
g2 <- f2()
x <- 20
str(g2())
#> List of 2
#>  $ defined: num 10
#>  $ called : num 20
```

在更加复杂的场景中，不仅只有一个父函数调用，而是一系列都指向起始函数的调用，也就是顶层调用（自上而下的调用）。下面的代码产生一个三层的调用栈。开放式箭头代表每一个执行环境的主调环境。

```
x <- 0
y <- 10
f <- function() {
  x <- 1
  g()
}
g <- function() {
  x <- 2
  h()
}
h <- function() {
  x <- 3
  x + y
}
f()
#> [1] 13
```



注意每个执行环境都有两个父环境：一个调用环境和一个封闭环境。R 普通作用域法则只使用封闭的父环境；parent.frame() 允许你访问调用父环境。

在调用环境而不是封闭环境中查找变量称为**动态作用域**（dynamic scoping）。很少的编程语言使用动态作用域（Emacs Lisp 就是一个非常明显的特例（http://www.gnu.org/software/emacs-paper.html#SEC15）.）这是因为动态调用使我们更难理解函数是如何运行的：不仅要知道它是如何定义的，还要知道它被调用时的上下文。动态调用主要用于开发交互式数据分析的函数。这是第 13 章的一个主题。

8.3.5 练习

1. 列出与函数相关的 4 种环境，它们都发挥什么作用？为什么了解封闭环境与绑定环境的不同非常重要？
2. 绘制一个示意图，用来显示下面函数的封闭环境：

```
f1 <- function(x1) {
  f2 <- function(x2) {
    f3 <- function(x3) {
      x1 + x2 + x3
    }
    f3(3)
  }
  f2(2)
}
f1(1)
```

139 ~ 140

3. 扩展上面的示意图来说明函数绑定。
4. 再次扩展上述示意图，说明执行环境和调用环境。
5. 自己编写一个增强版的 `str()` 函数，它可以显示函数的更多信息。显示定义函数的环境空间以及在哪里可以找到该函数。

8.4 绑定名字和数值

赋值操作其实就是将一个名字与一个值进行绑定（或重绑定）。它对应于作用域，这套规则决定如何找到与一个名字相关联的值。与大部分语言相比，R 语言有非常灵活的工具来对名称和数值进行绑定。事实上，不仅可以将值绑定到名字，还可以将表达式甚至函数绑定到名字，所以每次你访问与一个名字相关联的值时，你都会得到一些不同的东西！

可能你已经使用过上千次 R 赋值操作。赋值操作在当前环境中为名称和对象建立一种绑定关系。名字通常可以包含字母、数字、. 和 _，但不能以 _ 开头。如果不遵守这些规则就会出现错误：

```
_abc <- 1
# Error: unexpected input in "_"
```

虽然 R 的保留字（如 `TRUE`、`NULL`、`if` 和 `funtion`）也遵守这些规定，但是由于它们已经有其他用处了，所以我们不能使用它们：

```
if <- 10
#> Error: unexpected assignment in "if <-"
```

使用 `?Reserved` 可以获得完整的保留字列表。

这些通常的规则也可以被重写。在一个由任何字符序列构成的名称的两边加上反引号，就可以应用该名称了。

```
`a + b` <- 3
`:)` <- "smile"
`  ` <- "spaces"
```

```
ls()
#  [1] "   "   ":)"     "a + b"
`:)`
#  [1] "smile"
```

> **引 用**
>
> 除了反引号外，还可以使用单引号和双引号来创建非语法的绑定，但是不推荐这样做。在赋值箭头的左边使用字符串属于历史问题，在R开始支持反引号之前就已经开始使用了。

普通的赋值箭头 `<-` 总是在当前环境中创建一个变量。强制赋值箭头，`<<-` 不会在当前环境中创建变量，但是它修改父环境中已有的变量。也可以使用 `assign()` 来进行深度绑定：`name <<- value` 就等价于 `assign("name", value, inherits = TRUE)`。

```
x <- 0
f <- function() {
  x <<- 1
}
f()
x
#> [1] 1
```

如果 `<<-` 没有找到已有变量，它就在全局环境中创建一个。但这通常是不可取的，因为全局变量会在函数之间引入一些不明确的依赖关系。如10.3节所述，`<<-` 经常应用在闭包中。141 ~ 142

还有另外两个特殊类型的绑定，延迟绑定和主动绑定：

- **延时绑定**（delayed binding）不是立即把结果赋值给一个表达式，它创建和存储一个约定（promise），在需要时对约定中的表达式进行求值。

用特殊的赋值运算符 `%<d-%` 来创建延迟绑定。

```
library(pryr)
system.time(b %<d-% {Sys.sleep(1); 1})
#>    user  system elapsed
#>   0.000   0.000   0.001
system.time(b)
#>    user  system elapsed
#>   0.000   0.000   1.001
```

`%<d-%` 是对基础 `delayedAssign()` 函数的封装，如果需要更多的控制可以直接使用这个函数。延时绑定可以用来实现 `autoload()`，它使R的行为好像从内存加载软件包数据，即使当你请求它时它只是从硬盘加载。

- **主动绑定**（active binding）不是绑定到常量对象。相反，每次对其进行访问时都要重新计算。

```
x %<a-% runif(1)
x
#> [1] 0.4424
x
#> [1] 0.8112
rm(x)
```

`%<a-%` 是对基础函数 `makeActiveBinding()` 的封装。如果需要更多的控制可以直接使用这个函数。主动绑定用来实现引用类字段。

练习

1. 下面这个函数是做什么的？它与 <<- 的区别是什么？它的优点是什么？

```
rebind <- function(name, value, env = parent.frame()) {
  if (identical(env, emptyenv())) {
    stop("Can't find ", name, call. = FALSE)
  } else if (exists(name, envir = env, inherits = FALSE)) {
    assign(name, value, envir = env)
  } else {
    rebind(name, value, parent.env(env))
  }
}
rebind("a", 10)
#> Error: Can't find a
a <- 5
rebind("a", 10)
a
#> [1] 10
```


2. 创建一个新的 `assign()` 函数，使它只能绑定新名字而不会重新绑定原有名字。有些编程语言只能进行这样的赋值，这就是所谓的单赋值语言（http://en.wikipedia.org/wiki/Assignment_(computer_science)#Single_assignment）。
3. 创建一个可以进行主动绑定、延迟绑定和锁定绑定的赋值函数。你给它取个什么名字？它应该接收什么参数？根据它的输入能否猜出它进行的是什么绑定？

8.5 显式环境

除了服务于作用域之外，环境也是一种很有用的数据结构，因此它们有**引用语义**。与 R 中的大多数对象不同，当你对环境进行修改时，R 不会对其进行复制。例如，下面这个 `modify()` 函数。

```
modify <- function(x) {
  x$a <- 2
  invisible()
}
```

如果将这个函数应用于列表，原始列表不会被改变，因为修改列表实际上是创建和修改副本。

```
x_l <- list()
x_l$a <- 1
modify(x_l)

x_l$a
#> [1] 1
```

但是，如果将这个函数应用于环境，那么原始环境就会被修改：

```
x_e <- new.env()
x_e$a <- 1
modify(x_e)
x_e$a
#> [1] 2
```

就像可以使用列表在函数之间传递数据一样，也可以使用环境。当你创建自己的环境时，应该将父环境设置为空环境。这就可以确保不会从其他地方继承对象：

```
x <- 1
e1 <- new.env()
get("x", envir = e1)
#> [1] 1

e2 <- new.env(parent = emptyenv())
get("x", envir = e2)
#> Error: object 'x' not found
```

环境是解决下面 3 类常见问题的有用数据结构。

- 避免大数据的复制。
- 管理一个软件包内部的状态。
- 根据名字高效地查找与其绑定的值。

下面我们依次介绍它们。

8.5.1 避免复制

由于环境具有引用语义，所以决不会无意识地创建一个副本。这使它成为大对象的有用容器。bioconductor 包中经常使用这种技术，因为它经常需要对非常大的基因对象进行管理。在 3.1.0 版的 R 中，这种技术已经不像以前那样重要了，因为修改列表不再做深度复制了。以前，修改列表的一个元素也要复制整个列表，如果有些元素非常大，这样做就会造成昂贵的操作。现在，修改列表可以有效地重用已有的向量，能够节省许多时间。

144 ~ 145

8.5.2 软件包状态

显式环境在软件包中很有用，因为它们允许你在函数调用之间保持软件包的状态。正常情况下，软件包中的对象是被锁定的，所以你不能直接修改它们。但是，可以这样做：

```
my_env <- new.env(parent = emptyenv())
my_env$a <- 1

get_a <- function() {
  my_env$a
}
set_a <- function(value) {
  old <- my_env$a
  my_env$a <- value
  invisible(old)
}
```

从设置函数返回归值是非常好的模式，因为与这样可以很容易地重新设置与 `on.exit()` 相关联的前一个值（详情见 6.6.1 节）。

8.5.3 模拟 hashmap

hashmap 是一个非常有用的数据结构，它根据名字查找对象的时间复杂度为 O(1)。环境默认提供这种行为，所以可以用它来模拟 hashmap。CRAN 的 `hash` 包就是用这个思想开发的。

146

8.6 答案

1）4 个方面包括：环境空间中的每一个对象都必须有名字；排列顺序不重要；环境有父环境；环境有引用语义。

2）全局环境的父环境是你加载的最后一个添加包。唯一没有父环境的环境是空环境。

3）函数的封闭环境就是创建这个函数的环境。它决定该函数在哪里寻找变量。

147 ~ 148

4）使用 `parent.frame()`。

5）`<-` 总是在当前环境中创建一个绑定；`<<-` 重新绑定当前环境的父环境中的已有名字。

第 9 章
调试、条件处理和防御性编程

当你运行 R 代码时如果出错会怎样？你会怎么办？你会使用哪些工具来帮助你找到问题的所在？本章学习如何解决程序中的问题（调试），说明函数如何交流它们遇到的问题，我们应该如何根据这些信息来解决问题（条件处理），学习如何在问题发生前做出预防（防御性编程）。

调试可以说是一门用来解决代码中意想不到的错误的艺术和科学。本节学习一些可以帮助我们找到问题根源的工具和技术。我们将学习调试的通用策略，以及像 `traceback()` 和 `browser()` 等一些有用的 R 函数，还有 RStudio 中的交互式工具。

并不是所有的错误都是不可预期的。在编写函数时，我们可能预测一些潜在的问题（比如，文件不存在或者输入的错误类型）。将这些可能遇到的问题告诉用户就是**条件**（condition）的工作：错误、警告和消息。

- 致命错误是由 `stop()` 引起的，强制所有执行终止。当函数没有办法继续运行时就使用错误（Error）。
- 警告是由 `warning()` 产生的，用于显示潜在的问题，例如当输入的向量中存在一些无效元素时（如 `log(-1:2)`）。
- 消息是由 `message()` 产生的，用于给出具有充足信息的输出，用户可以非常简单地制止这些信息的输出（提示：可以通过命令 `?suppressMessages()` 来查看帮助文档）。在函数缺失重要参数时，我经常使用消息来告诉用户函数为缺失参数选用的默认值是什么。

条件通常是使用加粗字体或红色（根据使用的 R 界面会有所不同）突出显示的。错误总是使用“Error”开头，警告总是使用“Warning message”开头，所以它们很容易区分。函数的作者还可以使用 `print()` 或 `cat()` 来与用户交流，但我认为这并不是一个好主意，因为很难捕获或者有选择地忽略这类输出。打印的输出不属于条件，所以不能使用任何后边讲到的有用的条件处理工具。

当条件发生时，像 `withCallingHandlers()`、`tryCatch()` 和 `try()` 这样的条件处理工具允许你采取相应的行动。例如，你正在对很多模型进行拟合，如果其中有一个不能收敛，那么你可能仍然希望对其他模型继续进行拟合。基于 Common Lisp 的条件处理思想，R 提供一个超级强大的条件处理系统，但是它的文档现在还不够完善，也不经常被使用。本章

将学习最重要的基础知识，但如果你想了解更多，可以参考下面的两个资源：

- A prototype of a condition system for R（http://homepage.stat.uiowa.edu/~luke/R/exceptions/simpcond.html），由 Robert Gentleman 和 Luke Tierney 编写。它描述了 R 条件系统的早期版本。虽然现在的实现方法已经有所改变，但是本文还是很有价值的，你会对整个系统的构成有一个整体的认识，并对整个系统的设计动机有所了解。
- Beyond Exception Handling：Conditions and Restarts（http://www.gigamonkeys.com/book/beyond-exception-handling-conditions-and-restarts.html）由 Peter Seibel 著。本书描述了 Lisp 中的异常处理机制，R 的处理方法与其非常相似。它提供了更加复杂的例子（和有用的动力）。我已经将它翻译成了 R 语言版了 http://adv-r.had.co.nz/beyond-exception-handling.html。

在本章的最后我们讨论防御性编程：在问题发生前避免出现常识性错误的方法。从短期来看我们需要花费更多的时间来编写代码，但从长远来看它会帮助我们节约很多时间，因为这样编写的程序可以给我们提供很多有用的信息，从而可以使我们快速地找到问题的根源并解决它。防御性编程的基本原则是："快速失败"（fail fast），如果有问题马上发出错误信息。在 R 中可以采用 3 种形式：检查输入是否正确；避免非标准求值；避免使用返回不同类型输出的函数。

小测验

想跳过本章吗？如果你能回答下面的这些问题，你就可以大胆地跳过去。9.5 节有这些
150 问题的答案。

1）怎样知道哪里发生了错误？

2）`browser()` 函数的功能是什么？列出 `browser()` 环境中 5 个有用的单键命令。

3）使用哪个函数来忽略代码块中的错误？

4）为什么我们想为一个自定义的 S3 类创建一个错误？

主要内容

1. 9.1 节列出发现并解决程序中漏洞（bug）的通用方法。

2. 9.2 节介绍一些 R 函数和 Rstudio 特性，它们可以帮助我们准确地找到错误发生的位置。

3. 9.3 节介绍如何捕获程序代码中的各种条件（错误、警告和消息）。当错误出现的时候，这些技术可以使代码更加健壮，即使出错也能提供更多有用的信息帮助我们解决问题。

4. 9.4 节介绍一些防御性编程的重要技术，这些技术可以帮助我们防止错误的发生。

9.1 调试技巧

"漏洞查找就是确认那些你认为对的的确是对的，直到找到一个错的（本来你认为是对的）。"

——Norm Matloff

代码调试是非常具有挑战性的。很多漏洞都很细微并且很难发现。实际上，如果错误很明显，我们可能就第一时间发现并改正它。如果有非常好的技术，仅仅使用 `print()` 就可以找到很多漏洞，但有时可能需要一些额外的帮助。在本节中，我们将讨论 R 和 RStudio 提

供的一些有用的工具，并给出代码调试的通用步骤。

虽然下面的步骤不是绝对万无一失的，但在调试代码时它有助于我们组织思路。有以下 4 步：

（1）意识到漏洞的存在

如果你正在阅读本章，可能你已经完成这步了。这是非常重要的一步：只有知道漏洞的存在，才能修复它。这就是为什么当产生高质量代码时，自动化测试套件是非常重要的原因。不幸的是，自动化测试超出了本书的范围，但可以在 http://r-pkgs.had.co.nz/tests.html 了解更多相关知识。

（2）可以重现漏洞

一旦确定了漏洞存在，就可以使用命令来重现这个漏洞。如果不能重现，就很难找到问题的原因，也很难确定漏洞是否已经被修复。

通常情况下，我们从知道问题原因的大代码块开始，然后逐渐缩小范围，直到仍然能够产生错误的最短代码。这时折半搜索法很有用，不断地排除一半的代码直到找到漏洞。由于每次你都能减少一半的代码搜索量，所以这种方法是很快的。

如果产生一个错误需要很长时间，那么就需要我们找到可以在短时间内快速产生错误的方法。错误产生的速度越快，找到漏洞的速度也就越快。

在你创建一个最小的实例时，你可能发现同样的输入却不会产生问题。要注意这些细节：在诊断漏洞时它们会很有帮助。

如果你正在使用自动化测试，那么现在也正是创建自动化测试实例的好时间。如果现有的测试只能涵盖教少的部分，那么就需要添加一些相似的测试来确保现有的好行为能够得到保持。这会减少产生新漏洞的机会。

（3）找到漏洞的根源

如果你足够幸运，本节接下来要讲到的工具可能会帮助你快速地找到产生漏洞的代码行。但是，通常情况下，我们不得不对漏洞做进一步的考虑。采用科学的方法是一个好主意。产生一个假设，设计实验来测试这个假设，记录你的结果。看起来好像有很大的工作量，但是系统方法可以帮助你节约不少时间。我经常依靠直觉来调试代码的方法浪费了我很多时间（直觉告诉我："肯定是一个相差 1 的错误，这里减去 1 就好了！"），而如果采用一套系统方法会好很多。

（4）修复漏洞并进行测试

一旦发现漏洞，就需要找到修复它的方法，并检查这种修复是否有用。同样，使用自动化测试会很有帮助。它不仅仅能够帮助你确定漏洞已经修复，而且还能帮助你确定在整个漏洞修复过程中没有引入新的漏洞。如果没有自动化测试，就要记录正确的结果，并重新使用以前会出错的输入以便确保现在不会有错误产生。

9.2　调试工具

我们需要使用工具来实现调试策略。在本节中，我们要学习 R 和 RStudio 集成用户界面（IDE）提供的调试工具。RStudio 将 R 提供的调试工具友好地展现在用户面前，使我们能够更方便地对程序进行调试。我们会同时学习 R 和 RStudio 的调试方法，这样在你使用其他环境时也能很方便地进行程序调试。你可能还需要参考官方的 RStudio 调试文档（http://www.

rstudio.com/ide/docs/debugging/overview)，这里你可以学习 RStudio 的最新调试工具。

有 3 个主要的调试工具：

- RStudio 的错误查看器和 `traceback()`，它列出导致错误的调用顺序。
- RStudio 的“Rerun with Debug”工具和 `options(error = browser)`，它在错误发生的地方打开一个交互式对话。
- RStudio 的断点和 `browser()`，它可以在代码中的任意位置打开一个交互式对话。

下面对每一个工具进行详细介绍。

153 在编写新函数时，你应该不需要使用这些工具。如果在你的新代码中要经常使用这些工具，说明你应该重新考虑你的方法。不要尝试一开始就编写一个具有所有功能的大函数，你应该将所有功能分成很多小的部分来分别实现。如果从一小段代码开始，可以非常容易地检测哪里出现问题。如果从一大段代码开始，可能很难找到问题的根源。

9.2.1 确定调用顺序

第一个工具是**调用栈**（call stack），导致错误的调用顺序。这里有一个简单的例子：你会发现 `f()` 调用了 `g()`，`g()` 调用了 `h()`，`h()` 又调用了 `i()`，最后导致了一个错误：字符不能与数字做加法运算。

```
f <- function(a) g(a)
g <- function(b) h(b)
h <- function(c) i(c)
i <- function(d) "a" + d
f(10)
```

当我们在 Rstudio 中运行这段代码时，我们将看到：

```
> f(10)
Error in "a" + d : non-numeric argument to binary operator
```

Show Traceback
Rerun with Debug

在错误消息的右侧有两个选项：“Show Traceback”和“Rerun with Debug”。如果你单击“Show traceback”，会看到：

```
> f(10)
Error in "a" + d : non-numeric argument to binary operator

 4 i(c) at exceptions-example.R#3
 3 h(b) at exceptions-example.R#2
 2 g(a) at exceptions-example.R#1
 1 f(10)
```

Hide Traceback
Rerun with Debug

如果你没有使用 Rstudio，你可以使用 `traceback()` 函数来获得相同的信息：

```
traceback()
# 4: i(c) at exceptions-example.R#3
# 3: h(b) at exceptions-example.R#2
# 2: g(a) at exceptions-example.R#1
# 1: f(10)
```

从下往上阅读调用栈：初始调用为 `f()`，它又调用了 `g()`，`g()` 又调用了 `h()`，`h()` 又调用了 `i()`，最终导致错误。如果你已经将这段代码 `source()` 进 R，回溯信息还会以 filename.r#Linenumer 的形式显示函数的位置。在 RStudio 中这些都是可以单击的，而且它将你带到编辑器中的相应行。

有时这些信息已经足够帮助你找到错误并修复它。但是，通常情况下这些信息是不够的。`traceback()` 只显示问题在哪里发生，而不会告诉你为什么会出错。下一个有用的工具是交互式调试器，它可以帮助你暂时停止执行一个函数并交互式地查看它的状态。

9.2.2 查看错误

打开交互式调试器的最简单方法就是单击“ Rerun with Debug”。它返回发生错误的那个命令，并停止在错误发生的地方。现在我们就处于函数内部交互状态，并可以与那里定义的任何对象进行交互。还会在编辑器中看到相应的代码（下一个将要执行的语句会被高亮显示），当前环境中的对象会在“Environment”面板中显示，“Traceback”面板中显示调用栈，还可以在控制台中运行任意的 R 代码。

与所有常规的 R 函数一样，在调试模式下还可以使用几个特殊命令。可以通过 RStudio 工具栏（Next | | | Continue | Stop）或键盘来访问它们。

- Next（执行下一步），`n`：执行函数的下一步。如果有变量名为 `n` 的对象，那么就要小心一点。需要使用 `print(n)` 来将其输出。
- Stop into（单步执行），或 `s`：与执行下一步类似，但如果下一步是一个函数，那么就单步执行这个函数，就这样来通过函数的每一行。
- Finish（结束），或 `f`：结束当前循环或函数的执行。
- Continue（继续），`c`：离开交互式调试并继续函数的正常执行。如果你已经解决了最糟糕的问题并想使用它来检查函数能否正确执行，那么它是有用的。
- Stop（停止），`Q`：停止调试，终止函数，并返回全局工作空间。当你找到问题在哪儿，准备修复这个漏洞并重载代码时，可以使用它。

154 ~ 155

还有两个不太有用的并且在工具栏中找不到的命令：

- Enter：重复前一个命令。它很容易被我们不小心地激活，所以可以使用 `options(browserNLdisabled = TRUE)` 将它关闭。
- `where`：输出当前调用的栈追踪（等价于交互式的 `traceback`）。

为了在非 RStudio 下使用这种风格的调试，可以对 `error` 选项进行设置，它确定当错误发生时要执行哪个函数。与 Rstudio 调试最相似的函数就是 `browser()`：它会在错误发生的环境中打开一个交互式控制台。使用 `options(error = browser)` 开启它，重新运行前一条命令，然后使用 `options(error = NULL)` 返回到默认的错误行为。可以使用下面定义的 `browseOnce()` 函数来自动完成这一系列操作：

```
browseOnce <- function() {
  old <- getOption("error")
  function() {
    options(error = old)
    browser()
  }
}
options(error = browseOnce())

f <- function() stop("!")
# Enters browser
f()
# Runs normally
f()
```

（我们将在第 10 章中了解更多可以返回函数的函数。）

还有另外两个可以和 error 选项一起使用的函数：

- recover 是 browser 的加强版，它可以让我们进入调用栈中任何一个调用的环境。由于问题的根源通常是许多调用，所以它也是很有用的。
- dump.frames 等价于非交互式代码的 recover。首先它在当前工作目录中创建一个 last.dump.rda 文件。然后，在后边的交互式会话中，加载这个文件，使用 debugger() 进入交互式调试器，该调试器的界面与 recover() 的界面相同。它允许交互式地调试代码。

156

```
# In batch R process ----
dump_and_quit <- function() {
  # Save debugging info to file last.dump.rda
  dump.frames(to.file = TRUE)
  # Quit R with error status
  q(status = 1)
}
options(error = dump_and_quit)

# In a later interactive session ----
load("last.dump.rda")
debugger()
```

使用 options(error = NULL) 将错误行为重置为默认状态。然后错误将输出一条信息并中止函数的执行。

9.2.3 查看任意代码

与在出错的地方进入交互式终端一样，你也可以进入代码的任意位置，只需要使用 Rstudio 断点或者 browser() 函数。在 Rstudio 中，只需要单击左侧的行号就可以设置断点，或者使用 Shift + F9。等价于在你希望程序暂停的地方添加 browser()。断点在行为上与 browser() 类似，但更容易设置（只需要单击，而不是敲击 9 次键盘），并且也没有必要冒险在代码中添加 browser() 语句。当然断点也有两个小缺点：

- 在一些异常情况下断点会失效：可以参考断点故障排除（http://www.rstudio.com/ide/docs/debugging/breakpoint-troubleshooting）以了解更详细的信息。
- RStudio 现在还不支持条件断点，但是可以将 browser() 放到 if 语句中。

157

与我们自己添加 browser() 一样，还有另外两个函数也可以添加在代码中。

- debug() 在指定函数的第一行插入一个浏览器语句。undebug() 去除它。另外，可以使用 debugonce() 函数只浏览下一次运行。
- utils::setBreakpoint() 也一样，不过它不接受函数名，它接受文件名和行号，并帮助你找到合适的函数。

这两个函数都是 trace() 函数的特例，trace() 可以在一个现有函数的任意位置插入任意代码。当我们在对一个没有源文件的代码进行调试时，trace() 偶尔是有用的。每个函数只能执行一次追踪，但是一次追踪可以调用多个函数。

9.2.4 调用栈：traceback()、where 和 recover()

不幸的是，traceback()、browser() + where 和 recover() 输出的调用栈是不一

致的。下表显示了使用这 3 个工具输出的调用栈（还是对上例的分析）。

traceback()	where	recover()
4: stop("Error")	where 1: stop("Error")	1: f()
3: h(x)	where 2: h(x)	2: g(x)
2: g(x)	where 3: g(x)	3: h(x)
1: f()	where 4: f()	

注意，编号是 `traceback()` 和 `where` 之间的不同，`recover()` 以相反顺序显示调用，并省略了对 `stop()` 的调用。RStudio 显示的调用顺序与 `traceback()` 相同，但省略了编号。

9.2.5 其他类型的故障

除了抛出错误或者返回一个不正确的值外，函数还有其他出错方式：

- 函数可能产生一个意想不到的警告。查找警告的最简单方法就是使用 `options(warn = 2)` 将其转变成错误，并再使用常规调试工具。当你这样做时，你会在调用栈中看到一些额外的调用，如 `doWithOneRestart()`、`withOneRestart()`、`withRestarts()` 和 `.signalSimpleWarning()`。这些都可以忽略：它们是用来将警告转变成错误的内部函数。 158
- 函数还可能产生一些意想不到的消息。目前还没有内置工具可以帮助我们解决这个问题，但是我们可以自己创建一个：

```
message2error <- function(code) {
  withCallingHandlers(code, message = function(e) stop(e))
}

f <- function() g()
g <- function() message("Hi!")
g()
# Error in message("Hi!"): Hi!
message2error(g())
traceback()
# 10: stop(e) at #2
# 9: (function (e) stop(e))(list(message = "Hi!\n",
#       call = message("Hi!")))
# 8: signalCondition(cond)
# 7: doWithOneRestart(return(expr), restart)
# 6: withOneRestart(expr, restarts[[1L]])
# 5: withRestarts()
# 4: message("Hi!") at #1
# 3: g()
# 2: withCallingHandlers(code, message = function(e) stop(e))
#       at #2
# 1: message2error(g())
```

与警告一样，我们需要忽略追踪到的一些调用（例如，前两个和最后 7 个）。

- 函数可能没有返回。这尤其很难自动调试，但有时终止函数并查看它的调用栈，也许会得到一些有用的信息。否则，使用前面描述的基本调试策略。
- 最坏的场景是，你的代码让 R 彻底崩溃，使你无法进行交互式调试。这表明漏洞可能出现在 C 代码中。这很难调试。有时交互式调试器，如 `gdb`，可能有用，但是如何使用它已经超出了本书的范围。 159

如果这种崩溃是由基础 R 代码造成的，我们就可以到 R-help 发帖求助（记得要有一个可重现例子）。如果它在一个软件包内，就需要向软件包的维护者求助。如果它是你自己的 C 或者是 C++ 代码造成的，那么就需要使用许多 `print()` 语句来缩小漏洞的范围，然后你还需要使用更多的输出语句来查看哪个数据结构不符合你的预期。

9.3 条件处理

意想不到的错误需要交互式调试来找到问题的根源。然而，有些错误是可以预期到的，你希望自动处理这些错误。当使用多种模型对不同的数据集进行拟合时，经常会突然出现一些可以预期的错误。有时拟合会失败并抛出错误，但是你并不想退出整个程序。相反，你想拟合尽可能多的模型，然后再去对代码进行诊断。

在 R 中，有 3 个处理条件（包括错误）编程的工具：

- `try()` 允许我们在错误发生时继续执行代码。
- `tryCatch()` 可以让我们设置一个**处理器**（handler）函数，该函数控制在条件发生时应该做什么。
- `withCallingHandlers()` 是 `tryCatch()` 的一个变体，它可以在不同的环境中运行它的处理器函数。虽然很少用到它，但是对它有所了解还是很有用的。

下面几节将对这些工具进行详细介绍。

9.3.1 使用 try 来忽略错误

`try()` 允许我们在错误发生时继续执行代码。例如，通常情况下如果你运行一个抛出错
160 误的函数，它立即终止且不会返回任何值：

```
f1 <- function(x) {
  log(x)
  10
}
f1("x")
#> Error: non-numeric argument to mathematical function
```

但是，如果你把这个可能出错的函数放在 `try()` 函数中，错误消息将被输出但程序将继续执行：

```
f2 <- function(x) {
  try(log(x))
  10
}
f2("a")
#> Error in log(x) : non-numeric argument to mathematical function
#> [1] 10
```

可以使用 `try(..., silent = TRUE)` 来禁止这个消息。

如果要在 `try()` 中放置长代码，需要使用 `{}` 将它们括起来：

```
try({
  a <- 1
  b <- "x"
  a + b
})
```

我们还可以捕获 try() 的输出。如果执行成功，返回值就是括号中代码块的最后结果（与函数一样）。如果失败，返回值就是一个“try-error”类的（不可见的）对象。

```
success <- try(1 + 2)
failure <- try("a" + "b")
class(success)
#> [1] "numeric"
class(failure)
#> [1] "try-error"
```

当需要对一个列表中多个元素使用一个函数时，try() 就相当有用：

161

```
elements <- list(1:10, c(-1, 10), c(T, F), letters)
results <- lapply(elements, log)
#> Warning: NaNs produced
#> Error: non-numeric argument to mathematical function
results <- lapply(elements, function(x) try(log(x)))
#> Warning: NaNs produced
```

没有内置函数可以检测 try-error 类，因此需要我们自己定义一个。然后就可以使用 sapply() 方便地找到错误的位置（如第 11 章中的讨论），并提取成功信息或者查找导致失败的输入。

```
is.error <- function(x) inherits(x, "try-error")
succeeded <- !sapply(results, is.error)

# look at successful results
str(results[succeeded])
#> List of 3
#>  $ : num [1:10] 0 0.693 1.099 1.386 1.609 ...
#>  $ : num [1:2] NaN 2.3
#>  $ : num [1:2] 0 -Inf

# look at inputs that failed
str(elements[!succeeded])
#> List of 1
#>  $ : chr [1:26] "a" "b" "c" "d" ...
```

try() 的另一种常用方法是在表达式失败时使用默认值。只需要在 try 代码块的外边进行简单的赋值，然后运行这个风险代码：

```
default <- NULL
try(default <- read.csv("possibly-bad-input.csv"), silent = TRUE)
```

还可以使用 plyr::failwith()，它使这种策略更容易实施。可以在 12.2 节中查看更多细节。

9.3.2　使用 tryCatch() 处理条件

tryCatch() 是处理条件的一个通用工具：除了处理错误外，还可以对警告、消息和中断采取不同的行动。在前面我们已经见过错误（由 stop() 产生）、警告（由 warning() 产生）和消息（由 message() 产生），但是我们还没有讨论中断。程序员不能通过代码直接产生中断，但当用户试图强行终止（使用 Ctrl + Break、Escape 或者 Ctrl + C，不同的平台不一样）程序执行时中断才会发生。

162

通过 tryCatch() 可以将条件映射到**处理程序**（handler），它们是一些以条件作为输入的命名函数。当条件发生时，tryCatch() 调用名字与任何一个条件类相匹配的第一个处理程序。所有可以应用的内置名字有：error、warning、message、interrupt 和全部匹配 condition。处理程序可以做任何事情，但通常情况下使用它返回一个值或者创建一条包含更多信息的错误消息。例如，下面的 show_condition() 函数创建了一个可以返回触发条件类型的处理程序：

```
show_condition <- function(code) {
  tryCatch(code,
    error = function(c) "error",
    warning = function(c) "warning",
    message = function(c) "message"
  )
}
show_condition(stop("!"))
#> [1] "error"
show_condition(warning("?!"))
#> [1] "warning"
show_condition(message("?"))
#> [1] "message"

# If no condition is captured, tryCatch returns the
# value of the input
show_condition(10)
#> [1] 10
```

可以使用 tryCatch() 来实现 try()。下面是一个简单的实现。如果没有使用 tryCatch()，为了使错误消息更像你所看到的，base::try() 更加复杂。注意这里使用 conditionMessage() 来提取与原始错误相关联的消息。

```
try2 <- function(code, silent = FALSE) {
  tryCatch(code, error = function(c) {
    msg <- conditionMessage(c)
    if (!silent) message(c)
    invisible(structure(msg, class = "try-error"))
  })
}

try2(1)
#> [1] 1
try2(stop("Hi"))
try2(stop("Hi"), silent = TRUE)
```

当条件发生时，处理程序不仅返回默认值，还可以用来发出一些更有用的错误消息。例如，通过修改存储在错误条件对象中的消息，下面的函数封装 read.csv() 来给错误添加文件名：

```
read.csv2 <- function(file, ...) {
  tryCatch(read.csv(file, ...), error = function(c) {
    c$message <- paste0(c$message, " (in ", file, ")")
    stop(c)
  })
}
read.csv("code/dummy.csv")
#> Error: cannot open the connection
read.csv2("code/dummy.csv")
#> Error: cannot open the connection (in code/dummy.csv)
```

如果想要在用户终止代码执行时采取一些特殊的行动，那么捕获中断就非常有用。但是要注意的是，这样做很容易形成死循环（除非直接终止 R 进程）！

```
# Don't let the user interrupt the code
i <- 1
while(i < 3) {
  tryCatch({
    Sys.sleep(0.5)
    message("Try to escape")
  }, interrupt = function(x) {
    message("Try again!")
    i <<- i + 1
  })
}
```

163
〜
164

tryCatch() 有另一个参数：finally。由它设定的代码块（不是函数）无论初始表达式的执行是否成功都会执行。在进行清理工作时这很有用（例如，删除文件、关闭连接）。它在功能上等价于 on.exit()，但是它只能包含一小段代码而不能包含整个函数。

9.3.3 withCallingHandlers()

withCallingHandlers() 是 tryCatch() 的替代函数。这两个函数之间有两个主要不同：

- tryCatch() 处理程序的返回值是由 tryCatch() 返回的，而 withCallingHandlers() 处理程序的返回值是被忽略的：

```
f <- function() stop("!")
tryCatch(f(), error = function(e) 1)
#> [1] 1
withCallingHandlers(f(), error = function(e) 1)
#> Error: !
```

- withCallingHandlers() 中的处理程序是在产生条件的调用的上下文（环境）中被调用的，而 tryCatch() 中的处理程序是在 tryCatch() 的上下文（环境）中被调用的。这里使用 sys.calls() 来说明，该函数是 traceback() 的运行时等价函数——它列出导致当前函数的所有调用。

```
f <- function() g()
g <- function() h()
h <- function() stop("!")

tryCatch(f(), error = function(e) print(sys.calls()))
# [[1]] tryCatch(f(), error = function(e) print(sys.calls()))
# [[2]] tryCatchList(expr, classes, parentenv, handlers)
# [[3]] tryCatchOne(expr, names, parentenv, handlers[[1L]])
# [[4]] value[[3L]](cond)

withCallingHandlers(f(), error = function(e) print(sys.calls()))
# [[1]] withCallingHandlers(f(),
#     error = function(e) print(sys.calls()))
# [[2]] f()
# [[3]] g()
# [[4]] h()
```

165

```
# [[5]] stop("!")
# [[6]] .handleSimpleError(
#     function (e) print(sys.calls()), "!", quote(h()))
# [[7]] h(simpleError(msg, call))
```

这还影响 `on.exit()` 在其中被调用的顺序。

这么微小的差异很少有用，除非希望准确地捕获什么出错了并将该出错对象传递给其他函数。在大多数情况下，没有必要使用 `withCallingHandlers()`。

9.3.4 自定义信号类

在R语言中进行错误处理的一个挑战是：大部分函数都只使用一个字符串来调用 `stop()`。这说明，如果你想知道到底是发生了什么错误，就必须阅读错误消息文本。这是很容易出错的，不仅因为错误消息的文本可能经常改变，而且还因为很多错误消息都被转换，所以这些消息可能与你期望的完全不一样。

R有一个鲜为人知也很少使用的特点，可以帮助我们解决这个问题。条件是S3类，所以如果我们想区分不同类型的错误，我们可以定义自己的类。每一个条件信号函数 `stop()`、`warning()` 和 `message()`，都可以接收一个字符串列表或者一个自定义的S3条件对象。虽然自定义的条件对象并不是经常被使用，但是它可以使用户对不同的错误做出不同的反应，所以它很有用。例如，当程序遇到“可以预期”的错误时（例如，模型对于某些输入数据集拟合的不够好），可以直接跳过。而遇到意想不到的错误时（如没有可用的硬盘空间），才将消息传递给用户。

R没有内置构造函数的条件，但我们可以容易地给它添加一个。条件必须包含 `message` 和 `call` 元素，也可以包含其他有用的元素。当创建新的条件时，它应该总是从 `condition` 和 `error`、`warning` 或 `message` 三个中的一个继承。

```
condition <- function(subclass, message, call = sys.call(-1), ...) {
  structure(
    class = c(subclass, "condition"),
    list(message = message, call = call),
    ...
  )
}
is.condition <- function(x) inherits(x, "condition")
```

166

可以使用 `signalCondition()` 来发出各种条件，但是什么都不会发生除非已经实例化了一个自定义的信号处理程序（使用 `tryCatch()` 或 `withCallingHandlers()`）。实际上，使用 `stop()`、`warning()` 或 `message()` 来触发通常的处理。如果条件的类与函数不匹配，R不会报错，但在真实的代码中要避免这种情况。

```
c <- condition(c("my_error", "error"), "This is an error")
signalCondition(c)
# NULL
stop(c)
# Error: This is an error
warning(c)
# Warning message: This is an error
message(c)
# This is an error
```

然后就可以使用 `tryCatch()` 对不同的错误采取不同的行动。在本例中，我们创建一个 `custom_stop()` 函数，它允许我们发出任意类的错误条件。在实际应用中，最好有独立的 S3 构造函数，这样就可以在文档中更加详细地描述错误类型。

```
custom_stop <- function(subclass, message, call = sys.call(-1),
                        ...) {
  c <- condition(c(subclass, "error"), message, call = call, ...)
  stop(c)
}

my_log <- function(x) {
  if (!is.numeric(x))
    custom_stop("invalid_class", "my_log() needs numeric input")
  if (any(x < 0))
    custom_stop("invalid_value", "my_log() needs positive inputs"

  log(x)
}
tryCatch(
  my_log("a"),
  invalid_class = function(c) "class",
  invalid_value = function(c) "value"
)
#> [1] "class"
```

注意，当使用具有多个处理程序和自定义类的 `tryCatch()` 时，调用与信号类层次结构中的任意类相匹配的第一个处理程序，而不是最好的匹配。为此，需要将最具体的处理程序放在最前面：

```
tryCatch(customStop("my_error", "!"),
  error = function(c) "error",
  my_error = function(c) "my_error"
)
#> [1] "error"
tryCatch(custom_stop("my_error", "!"),
  my_error = function(c) "my_error",
  error = function(c) "error"
)
#> [1] "my_error"
```

9.3.5 练习

对下面的两个 `message2error()` 的实现进行比较。在这种情况下 `withCallingHandlers()` 的主要优点是什么（提示：仔细查看回溯）？

```
message2error <- function(code) {
  withCallingHandlers(code, message = function(e) stop(e))
}
message2error <- function(code) {
  tryCatch(code, message = function(e) stop(e))
}
```

9.4 防御性编程

防御性编程就是当有意想不到的错误发生时，程序仍然能够按照预定义的方式出错。防御性编程的关键原则就是“快速失败”：如果问题出现就尽量快地发出错误信息。这需要函

167 ~ 168

数的编写者做更多的工作，但这使用户的调试工作变得更加容易，因为他们可以更早地得到错误消息，而不是在非法输入已经通过了很多函数之后。

在 R 中，“快速失败”原则以下面 3 种方式实现：

- 更严格的输入控制。例如，如果函数的输入不是向量化的，但却以向量化的方式为函数提供输入，那么确保检查输入是标量。可以使用 `assertthat`（http://github.com/hadley/assertthat）添加包中的 `stopifnot()` 函数，或者使用 `if` 语句和 `stop()`。
- 避免使用非标准计算的函数，如 `subset`、`transform` 和 `with`。在交互使用时这些函数可以节约时间，但因为它们假设减少键盘输入量，所以当它们出错时，它们通常给出无用的信息。我们将在第 13 章中学习非标准计算。
- 避免使用根据输入返回不同类型输出的函数。两个最大的元凶就是 `[` 和 `sapply()`。当在函数中对数据框进行子集选取时，应该总是使用 `drop = FALSE`，否则就将仅有一列的数据框转变成向量。同样，永远不要在函数内部使用 `sapply()`：总是使用更加严谨的 `vapply()`，如果输入的类型不正确，它将抛出错误消息，甚至当输入的长度为 0 时，它也能返回正确的输出类型。

交互式分析和编程之间存在着一定的分歧。当我们进行交互式工作时，我们希望 R 做我们想做的。如果猜错了，我们希望能够马上找到并修复它。在编程时，我们希望函数能够及时汇报错误消息，即使是轻微的错误或遗漏。在编写函数时需要记住两者之间的这种分歧。如果正在编写交互式数据分析的函数，可以对用户的需求进行自由猜测并对很小的错误设定自动恢复。如果正在为一个程序设计编写函数，就应该更加严格一点儿。永远不要猜测对方想要什么。

练习

169 1. 下面定义的 `col_means()` 函数可以对数据框中的每一列计算平均值。

```
col_means <- function(df) {
  numeric <- sapply(df, is.numeric)
  numeric_cols <- df[, numeric]

  data.frame(lapply(numeric_cols, mean))
}
```

但是，这个函数对于异常输入不稳健。看看下面这些结果，确定哪个是错误的，并对 `col_means()` 函数进行修改使它变得更加稳健。（提示：在 `col_means()` 中有两个函数调用非常容易出错。）

```
col_means(mtcars)
col_means(mtcars[, 0])
col_means(mtcars[0, ])
col_means(mtcars[, "mpg", drop = F])
col_means(1:10)
col_means(as.matrix(mtcars))
col_means(as.list(mtcars))

mtcars2 <- mtcars
mtcars2[-1] <- lapply(mtcars2[-1], as.character)
col_means(mtcars2)
```

2. 下面的函数 `lag` 可以对一个向量 `x` 进行滞后操作，它返回一个滞后了 `n` 值的 `x`。对这个函数进行修改，使它可以：（1）如果 `n` 不是向量，可以返回一个有用的错误消息；（2）可以合理地处理 `n` 为 0 或者大于 `x` 时的情况。

```
lag <- function(x, n = 1L) {
  xlen <- length(x)
  c(rep(NA, n), x[seq_len(xlen - n)])
}
```

9.5　答案

1）检测错误在哪里发生的最佳工具为 `traceback()`。或者使用 RStudio，它自动显示错误发生的位置。 170

2）`browser()` 可以在指定的行中断代码的执行，并允许我们进入一个交互式环境。在这个环境有 5 个有用的命令：`n`，执行下一条命令；`s`，单步执行函数；`f`，完成当前循环或函数；`c`，继续执行；`Q`，关闭函数并返回到控制台。

3）可以使用 `try()` 或 `tryCatch()`。

4）因为我们可以使用 `tryCatch()` 捕获特定类型的错误，而不是依赖于错误字符串的比较，比较是有很大风险的，尤其是当消息被转换后。 171 ~ 172

第二部分

函数式编程

第 10 章
函数式编程

R 语言的核心其实是一门函数式编程（FP）语言。这就意味着它为我们提供了大量的创建和操作函数的工具。尤其是，R 还有所谓的一级函数。适用于向量的所有操作也都适用于函数：可以将函数赋值给变量，将函数存储在列表中，将函数作为参数传递给其他函数，在函数内再创建一个函数，甚至可以把函数作为一个函数的结果返回。

本章首先给出一个引导性的例子，它去除代码中的冗余和重复以便清洗和汇总数据。然后我们学习函数式编程的 3 个组件：匿名函数、闭包（由函数写的函数）和函数列表。最后，作为结尾部分，我们从最简单的原语入手，展示如何使用这些组件构建一套数值积分的工具。这在函数式编程中是一个反复出现的主题：从一个小的很容易理解的组件开始，将它们组合成更加复杂的结构，然后再充满信心地使用它们。

对函数式编程的讨论将分为两章：第 11 章探索以函数作为输入、以向量作为输出的函数；第 12 章探索以函数作为输入、以函数作为输出的函数。

主要内容

- 10.1 节通过一个常见的问题把大家引入函数式编程：在严肃的数据分析之前先对数据进行清洗和汇总。
- 19.2 节介绍你不知道的函数的另一面：没有名字的函数也可以使用。
- 19.3 节介绍闭包，它是由另一个函数编写的函数。闭包可以访问定义在它父环境中的参数和变量。
- 19.4 节介绍如何将函数放入列表，并解释为什么需要它。
- 19.5 节通过一个使用匿名函数、闭包和函数列表来构建数值积分工具包的例子来总结本章的知识。

173 ~ 175

预备条件

你应该熟悉 6.2 节中介绍的词法作用域的基本规则。确保已经使用 `install.packages("pryr")` 安装了 `pryr` 包。

10.1 动机

假设像下面一样加载了一个数据文件，其中 -99 代表缺失值。我们想用 `NA` 来替换所有的 -99。

```
# Generate a sample dataset
set.seed(1014)
df <- data.frame(replicate(6, sample(c(1:10, -99), 6, rep = TRUE)))
names(df) <- letters[1:6]
df
#>    a  b c   d   e f
#> 1  1  6 1   5 -99 1
#> 2 10  4 4 -99   9 3
#> 3  7  9 5   4   1 4
#> 4  2  9 3   8   6 8
#> 5  1 10 5   9   8 6
#> 6  6  2 1   3   8 5
```

当我们第一次编写 R 代码时，我们可能使用复制和粘贴来解决这个问题：

```
df$a[df$a == -99] <- NA
df$b[df$b == -99] <- NA
df$c[df$c == -98] <- NA
df$d[df$d == -99] <- NA
df$e[df$e == -99] <- NA
df$f[df$g == -99] <- NA
```

使用复制和粘贴很容易出错。你发现上面的代码中有两处错误吗？由于我们没有对我们要做的行为（使用 NA 来替代 -99）给出正式的描述，所以发生的这些错误也可能不一样，上例中的两个错误就是不同的。重复代码很容易出错而且使代码变得很难修改。例如，如果需要将缺失值从 -99 变为 9999，就必须在多个地方进行修改。

为了避免产生这类漏洞并使代码更灵活，采用“不要自我重复”（do not repeat yourself），或者 DRY 原则。这是 Dave Thomas 和 Andy Hunt 在他们的《Pragmatic Programmers》(http://pragprog.com/about）一书中提出的，这个原则就是：“在系统中，每一条知识都必须有一条明确的正式表示”。函数式编程工具很有价值，因为它们提供了减少重复的工具。

现在我们就使用这种函数式编程思想来编写一个用来替换向量中缺失值的函数。

```
fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}
df$a <- fix_missing(df$a)
df$b <- fix_missing(df$b)
df$c <- fix_missing(df$c)
df$d <- fix_missing(df$d)
df$e <- fix_missing(df$e)
df$f <- fix_missing(df$e)
```

这减少了可能出错的范围，但不能完全消除它们：虽然再也不会偶然将 -99 输入成 -98，但是仍然可能将变量名弄错。下一步就是将两个函数结合到一起来避免这种错误的发生。第一个函数，`fix_missing()`，它知道如何对单个向量进行修复；第二个函数，`lapply()`，可以将这种操作应用到数据框中的每一列。

`lapply()` 可以接受 3 个输入：`x`，一个列表；`f`，一个函数；以及 `...`，传给 `f()` 的其他参数。它可以将这个函数应用到列表的每一个元素并返回一个新列表。`lapply(x, f, ...)` 等价于下面的循环语句：

176 ~ 177

```
out <- vector("list", length(x))
for (i in seq_along(x)) {
```

```
  out[[i]] <- f(x[[i]], ...)
}
```

真实的 `lapply()` 相当复杂，因为为了提高效率，`lapply()` 是用 C 语言实现的，但是算法的本质相同。由于 `lapply()` 以函数作为输入，所以它也是一个**泛函**（functional）。泛函是函数式编程中的一个非常重要部分。将在第 11 章中深入讨论它。

由于数据框是列表，所以我们可以使用 `lapply()` 来解决这个问题。只需要使用一个小技巧确信得到数据框而不是列表。不是将 `lapply()` 的结果赋值给 `df`，而是赋值给 `df[]`。R 常用规则确保我们得到数据框而不是列表。（如果你对这个感到惊讶，你可能需要阅读 3.3 节的相关内容）。将这些代码段放在一起可以得到：

```
fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}
df[] <- lapply(df, fix_missing)
```

与复制和粘贴相比，这段代码有 5 个优点：

- 更紧凑。
- 如果想改变代码的缺失值，只需要修改一处就可以。
- 无论有多少列都可以使用它。不会丢掉任何一列。
- 对所有列的操作都是相同的。
- 可以非常容易地将这个技术推广到列的子集：

```
df[1:5] <- lapply(df[1:5], fix_missing)
```

这个关键思想是函数组合。将两个简单函数组合起来，一个函数修复缺失值，另一个函数对每一列做同样的操作，将它们结合在一起修复每一列中的缺失值。编写一些很容易理解的简单函数，然后再将它们组合为一个强大的技术。

如果每一列使用不同的值来替换缺失值又该怎么办呢？可能又要复制和粘贴：

```
fix_missing_99 <- function(x) {
  x[x == -99] <- NA
  x
}
fix_missing_999 <- function(x) {
  x[x == -999] <- NA
  x
}
fix_missing_9999 <- function(x) {
  x[x == -999] <- NA
  x
}
```

与前面一样，它容易出错。然而我们可以使用闭包，它是创建并返回函数的函数。闭包允许我们基于模板来创建函数：

```
missing_fixer <- function(na_value) {
  function(x) {
    x[x == na_value] <- NA
    x
```

```
  }
}
fix_missing_99 <- missing_fixer(-99)
fix_missing_999 <- missing_fixer(-999)

fix_missing_99(c(-99, -999))
#> [1]   NA -999
fix_missing_999(c(-99, -999))
#> [1] -99  NA
```

178 ≀ 179

额外参数

在本例中，你可能认为应该只再添加一个参数：

```
fix_missing <- function(x, na.value) {
  x[x == na.value] <- NA
  x
}
```

这里是可以这样做的，但在其他地方它可能就不合适了。我们将在 11.5 节中看到更多复杂但又必须使用闭包的例子。

现在考虑一个相关问题。一旦将数据清洗干净，你可能想为每个变量计算同一组数值汇总。你可能把代码写成这样：

```
mean(df$a)
median(df$a)
sd(df$a)
mad(df$a)
IQR(df$a)

mean(df$b)
median(df$b)
sd(df$b)
mad(df$b)
IQR(df$b)
```

同样，最好标识并删除重复的项目。在继续阅读之前花一两分钟来思考怎样解决这个问题。

一种方法就是编写一个汇总函数，然后将它应用到每一列：

```
summary <- function(x) {
  c(mean(x), median(x), sd(x), mad(x), IQR(x))
}
lapply(df, summary)
```

180

这是一个好的开始，但是仍然有一些重复。很容易看出这以下个汇总函数实用：

```
summary <- function(x) {
 c(mean(x, na.rm = TRUE),
   median(x, na.rm = TRUE),
   sd(x, na.rm = TRUE),
   mad(x, na.rm = TRUE),
   IQR(x, na.rm = TRUE))
}
```

所有这 5 个函数都用相同的参数（x 和 na.rm）重复调用 5 次。同样，重复使代码变得很脆弱：它很容易产生漏洞并很难根据需求做出修改。

为了减少这种重复，可以利用函数式编程的另一种技术：在列表中存储函数。

```
summary <- function(x) {
  funs <- c(mean, median, sd, mad, IQR)
  lapply(funs, function(f) f(x, na.rm = TRUE))
}
```

本章将详细讨论这个函数。但是在开始学习之前，有必要学习函数式编程中最简单的工具：匿名函数。

10.2 匿名函数

在 R 中，函数本身就是对象。它们不会自动与名字绑定到一起。这与其他很多语言（例如，C、C++、Python 和 Ruby）不同，R 没有创建命名函数的特定语法：在创建函数时，使用常规赋值运算符给它命名。如果选择不给它命名，就得到了一个**匿名函数**（anonymous function）。

181

当觉得没有必要给函数命名时，就使用匿名函数：

```
lapply(mtcars, function(x) length(unique(x)))
Filter(function(x) !is.numeric(x), mtcars)
integrate(function(x) sin(x) ^ 2, 0, pi)
```

与 R 中所有的函数一样，匿名函数也有 formals()、body() 和父 environment()：

```
formals(function(x = 4) g(x) + h(x))
#> $x
#> [1] 4
body(function(x = 4) g(x) + h(x))
#> g(x) + h(x)
environment(function(x = 4) g(x) + h(x))
#> <environment: R_GlobalEnv>
```

调用匿名函数时可以不使用函数名，但是调用代码读起来可能有点晦涩，因为必须以两种不同的方式使用括号：首先，调用一个函数；其次，使 R 明白我们想调用的是匿名函数本身，而不是调用匿名函数内部的（可能无效的）函数：

```
# This does not call the anonymous function.
# (Note that "3" is not a valid function.)
function(x) 3()
#> function(x) 3()

# With appropriate parenthesis, the function is called:
(function(x) 3)()
#> [1] 3

# So this anonymous function syntax
(function(x) x + 3)(10)
#> [1] 13

# behaves exactly the same as
f <- function(x) x + 3
f(10)
#> [1] 13
```

在调用匿名函数时，可以使用命名的参数。但如果真需要这样，说明函数需要一个名字。

匿名函数最常见的应用就是创建闭包，由其他函数创建的函数。我们将在下一节学习闭包。 182

练习

1. 给定一个函数，如“`mean`”，`match.fun()`能为你找到一个函数？给定一个函数，你能找到它的名字吗？为什么在 R 中这样做是没意义的？
2. 使用`lapply()`和匿名函数来计算`mtcars`数据集中的所有列的变异的系数（标准差除以平均值）。
3. 使用`integrate()`和匿名函数计算下列曲线下方的面积。使用 Wolfram Alpha（http://www.wolframalpha.com）来检查你的答案。
   1）`y = x ^ 2 - x`　x 在`[0, 10]`中
   2）`y = sin(x) + cos(x)`　x 在`[-π, π]`中
   3）`y = exp(x) / x`　x 在`[10, 20]`中
4. 匿名函数最好在一行内，不应该使用`{}`。看看你写的代码。哪里已经使用了匿名函数而没有使用命名函数，哪里应该使用命名函数而不是匿名函数？

10.3　闭包

“对象是带有函数的数据。闭包是带有数据的函数。”

——John D. Cook

匿名函数的一个用途就是创建一些没有必要命名的小函数。另一个重要用途就是创建闭包，即由函数编写的函数。闭包的得名是因为：它将父函数的环境**封装**并可以访问它的所有变量。这一点是有用的，因为它允许我们使用两个层次的参数：父层次可以控制运算；子层次可以进行工作。

下面的例子使用这个思想创建一组幂函数，其中一个父函数（`power()`）创建了两个子函数（`square()`和`cube()`）。 183

```
power <- function(exponent) {
  function(x) {
    x ^ exponent
  }
}

square <- power(2)
square(2)
#> [1] 4
square(4)
#> [1] 16

cube <- power(3)
cube(2)
#> [1] 8
cube(4)
#> [1] 64
```

在输出闭包时，得不到任何有用的信息：

```
square
#> function(x) {
#>     x ^ exponent
#>   }
#> <environment: 0x7fa58d810940>
cube
#> function(x) {
#>     x ^ exponent
#>   }
#> <environment: 0x7fa58d98f098>
```

这是因为函数本身并没有发生改变。不同的是闭包环境，`environment(square)`。查看环境内容的一种方法是将其转换成列表：

```
as.list(environment(square))
#> $exponent
#> [1] 2
as.list(environment(cube))
#> $exponent
#> [1] 3
```

184

另一种方法是，使用 `pryr::unenclose()` 来查看到底发生了什么。这个函数使用相应的值来替换定义在封闭环境中的变量：

```
library(pryr)
unenclose(square)
#> function (x)
#> {
#>     x^2
#> }
unenclose(cube)
#> function (x)
#> {
#>     x^3
#> }
```

闭包的父环境是创建它的函数的执行环境，如下面的代码所示：

```
power <- function(exponent) {
  print(environment())
  function(x) x ^ exponent
}
zero <- power(0)
#> <environment: 0x7fa58ad7e7f0>
environment(zero)
#> <environment: 0x7fa58ad7e7f0>
```

在函数返回值后，执行环境通常就会消失。但是，函数捕获它们的封闭环境。这就意味着当函数 a 返回函数 b 时，函数 b 捕获并存储函数 a 的执行环境，并且它不会消失。(这对内存使用产生重要影响，详细内容见 18.2 节。)

在 R 中，几乎每一个函数都是闭包。所有函数都记住创建它们的环境，通常情况下，如果是自己写的函数，那么就是全局环境；如果是别人写的函数，那么就是添加包环境。唯一例外是元函数 (primitive function)，它们直接调用 C 代码，而且没有相关联的环境。

185

闭包对于创建函数工厂是很有用的，这是 R 中控制可变状态的一种方法。

10.3.1　函数工厂

函数工厂就是创建新函数的工厂。我们已经看到了两个函数工厂的例子，`missing_fixer()` 和 `power()`。可以使用描述期望行为的参数调用它，它返回一个符合你要求的函数。对于 `missing_fixer()` 和 `power()`，使用函数工厂的优势并不比使用带有多个参数的单个函数强很多。但在下列情况下，函数工厂会更有用：

- 不同层次更加复杂，带有多个参数和复杂的函数体。
- 当创建函数时，有些工作只需要做一次。

函数工厂尤其适合最大似然问题，我们将在 11.5 节中看到它更复杂的应用。

10.3.2　可变状态

具有两个层次的变量允许在函数调用之间保持状态。虽然执行环境每次都会更新，但是封闭环境保持不变，所以这是可能的。在不同层次管理变量的关键是双箭头赋值符（`<<-`）。通常使用的单箭头赋值符（`<-`）只是在当前环境进行赋值，双箭头赋值符会搜寻父环境链直到找到匹配的名字（在 8.4 节中有更多相关知识。）

将一个静态的父环境和 `<<-` 放在一起可以在函数调用之间保持状态。下面是一个计数器的例子，它记录一个函数被调用了多少次。每次运行 `new_counter` 都创建一个环境，初始化环境中的计数器 `i`，然后创建一个新函数。

```
new_counter <- function() {
  i <- 0
  function() {
    i <<- i + 1
    i
  }
}
```

186

这个新函数就是闭包，它的封闭环境就是 `new_counter()` 运行时创建的环境。通常情况下，函数执行环境是临时的，但是闭包可以一直访问它创建的环境。在下面的例子中，两个闭包 `counter_one()` 和 `counter_two()` 在运行时都可以获取它们自己的封闭环境，所以它们可以维持不同的计数。

```
counter_one <- new_counter()
counter_two <- new_counter()

counter_one()
#> [1] 1
counter_one()
#> [1] 2
counter_two()
#> [1] 1
```

计数器可以规避在它们的局部环境中不能修改变量的“重新开始”的限制。因为这种改变是在未改变的父（或者封闭）环境中进行的，所以在函数调用之间保留下来它们。

如果不使用闭包又会怎样呢？如果使用“`<-`”而不使用“`<<-`”呢？预测下面变体替换 `new_counter()` 会发生什么，运行代码并看看你的预测是否正确。

```
i <- 0
new_counter2 <- function() {
  i <<- i + 1
```

```
  i
}
new_counter3 <- function() {
  i <- 0
  function() {
    i <- i + 1
    i
  }
}
```

在父环境中修改值是一项很重要的技术，因为这是 R 中产生可变状态的方法。实现可变
187 状态通常是很难的，因为每次修改一个对象时，实际上都是创建对象然后对副本进行修改。但是，如果确实需要可变状态对象而且代码也不是非常简单，通常最好使用参考类，如 7.4 节所述。

闭包的强大功能通常与第 11 章和第 12 章中的更高级的思想紧密相连。在这两章中，我们将看到更多的闭包。在接下来的几节中，我们将讨论函数式编程的第 3 个技术：将函数存储在列表中。

10.3.3 练习

1. 为什么由其他函数创建的函数称为闭包？
2. 下面统计函数的功能是什么？能给它取一个更好的名字吗？（现有的名字有一点儿隐晦。）

```
bc <- function(lambda) {
  if (lambda == 0) {
    function(x) log(x)
  } else {
    function(x) (x ^ lambda - 1) / lambda
  }
}
```

3. 函数 `approxfun()` 的功能是什么？它返回什么？
4. 函数 `ecdf()` 的功能是什么？它返回什么？
5. 创建一个函数，该函数可以创建计算数值向量的第 i 阶中心矩的函数（http://en.wikipedia.org/wiki/Central_moment），运行下面的代码来检测你的函数。

```
m1 <- moment(1)
m2 <- moment(2)

x <- runif(100)
stopifnot(all.equal(m1(x), 0))
stopifnot(all.equal(m2(x), var(x) * 99 / 100))
```

6. 创建一个 `pick()` 函数，它接受一个索引参数 `i`，并根据这个参数返回一个带有参数 `x` 的函数，这个函数可以根据 `i` 对 `x` 进行子集选取。

```
lapply(mtcars, pick(5))
# should do the same as this
188 lapply(mtcars, function(x) x[[5]])
```

10.4 函数列表

在 R 中，函数可以存放在列表中。这使函数可以更容易地与一组相关函数一起运行，就

像数据框可以使函数更容易地与一组相关向量运行一样。

我们将从一个简单的基准测试例子开始学习。假设我们正在比较计算算术平均值的多种方法的效率。我们可以将这些方法（函数）都存储在一个列表中：

```
compute_mean <- list(
  base = function(x) mean(x),
  sum = function(x) sum(x) / length(x),
  manual = function(x) {
    total <- 0
    n <- length(x)
    for (i in seq_along(x)) {
      total <- total + x[i] / n
    }
    total
  }
)
```

可以直接从列表中调用函数。先提取函数然后再调用它：

```
x <- runif(1e5)
system.time(compute_mean$base(x))
#>    user  system elapsed
#>   0.001   0.000   0.001
system.time(compute_mean[[2]](x))
#>    user  system elapsed
#>       0       0       0
system.time(compute_mean[["manual"]](x))
#>    user  system elapsed
#>   0.053   0.003   0.055
```

使用 `lapply()` 调用每一个函数（即，检查它们的返回值是否相同）。由于没有处理这个问题的内置函数，所以需要使用一个匿名函数或者一个新的命名函数。 189

```
lapply(compute_mean, function(f) f(x))
#> $base
#> [1] 0.4995
#>
#> $sum
#> [1] 0.4995
#>
#> $manual
#> [1] 0.4995

call_fun <- function(f, ...) f(...)
lapply(compute_mean, call_fun, x)
#> $base
#> [1] 0.4995
#>
#> $sum
#> [1] 0.4995
#>
#> $manual
#> [1] 0.4995
```

为了给每个函数计时，可以将 `lapply()` 和 `system.time()` 结合使用：

```
lapply(compute_mean, function(f) system.time(f(x)))
#> $base
```

```
#>    user  system elapsed
#>   0.000   0.000   0.001
#>
#> $sum
#>    user  system elapsed
#>       0       0       0
#>
#> $manual
#>    user  system elapsed
#>   0.051   0.003   0.054
```

函数列表的另一个用途是对一个对象进行多方面汇总。为此，可以将每一个汇总函数存储在一个列表中，然后使用 `lapply()` 一起运行它们：

```
x <- 1:10
funs <- list(
  sum = sum,
  mean = mean,
  median = median
)
lapply(funs, function(f) f(x))
#> $sum
#> [1] 55
#>
#> $mean
#> [1] 5.5
#>
#> $median
#> [1] 5.5
```

如果希望汇总函数可以自动去除缺失值呢？一种方法是创建一个匿名函数列表，它使用适当的参数来调用汇总函数：

```
funs2 <- list(
  sum = function(x, ...) sum(x, ..., na.rm = TRUE),
  mean = function(x, ...) mean(x, ..., na.rm = TRUE),
  median = function(x, ...) median(x, ..., na.rm = TRUE)
)
lapply(funs2, function(f) f(x))
#> $sum
#> [1] 55
#>
#> $mean
#> [1] 5.5
#>
#> $median
#> [1] 5.5
```

但是，这样会造成大量的重复。除了不同的函数名之外，每个函数几乎是相同的。一种更好的方法是修改 `lapply()` 调用，让它包含一个额外参数：

```
lapply(funs, function(f) f(x, na.rm = TRUE))
```

10.4.1 将函数列表移到全局环境中

190 ~ 191

有时，可以创建一个不使用特殊语法的可用的函数列表。例如，想通过将每一个标签映射到一个 R 函数来创建 HTML 代码。下面的例子使用函数工厂来为标签 <p>（段落）、<b>（加

粗）和 <i>（斜体）创建函数。

```
simple_tag <- function(tag) {
  force(tag)
  function(...) {
    paste0("<", tag, ">", paste0(...), "</", tag, ">")
  }
}
tags <- c("p", "b", "i")
html <- lapply(setNames(tags, tags), simple_tag)
```

将这些函数放入一个列表，因为我们不希望它们一直可以使用。现有 R 函数与 HTML 标签之间的冲突风险是非常大的。但是将它们保存在列表中可以使代码更易读：

```
html$p("This is ", html$b("bold"), " text.")
#> [1] "<p>This is <b>bold</b> text.</p>"
```

根据我们希望标签发挥作用的范围，有 3 种选择可以终止 `html$` 的作用：

- 对于一个非常临时的作用，可以使用 `with()`：

```
with(html, p("This is ", b("bold"), " text."))
#> [1] "<p>This is <b>bold</b> text.</p>"
```

- 对于一个长作用，可以将这个函数 `attach()` 到搜索路径，在它使用完后再将其 `detach()`：

```
attach(html)
p("This is ", b("bold"), " text.")
#> [1] "<p>This is <b>bold</b> text.</p>"
detach(html)
```

- 最后，可以使用 `list2env()` 将这个函数复制到全局环境中。在它使用完后可以删除这些函数。

```
list2env(html, environment())
#> <environment: R_GlobalEnv>
p("This is ", b("bold"), " text.")
#> [1] "<p>This is <b>bold</b> text.</p>"
rm(list = names(html), envir = environment())
```

192

我个人更推荐第一种选择，使用 `with()`，因为它使代码运行的外部环境非常清楚，便于理解。

10.4.2 练习

1. 自己编写一个与 `base::summary()` 类似的汇总函数，要求使用函数列表。修改这个函数，让它返回一个闭包，使它可以当作一个函数工厂来使用。
2. 下面哪个命令与 `with(x, f(z))` 等价？
   (a) `x$f(x$z)`
   (b) `f(x$z)`
   (c) `x$f(z)`
   (d) `f(z)`
   (e) 视情况而定

10.5 案例研究：数值积分

作为本章的结束，我们使用一级函数来开发一个简单的数值积分工具。在开发本工具的每一步中，都要求尽量减少重复代码并使方法更加通用。

数值积分的思想很简单：通过用简单的成分近似曲线来寻找曲线下的面积。两个最简单方法是**中点**（midpoint）法则和**梯形**（trapezoid）法则。中点法则用矩阵近似曲线。梯形法则使用梯形近似。每个法则都以要积分的函数 `f` 和积分范围（`a`～`b`）为参数来进行积分。例如，尝试对 `sin x`（0～π）进行积分。这个选择很好，因为答案很简单：2。

```
midpoint <- function(f, a, b) {
  (b - a) * f((a + b) / 2)
}
trapezoid <- function(f, a, b) {
  (b - a) / 2 * (f(a) + f(b))
}

midpoint(sin, 0, pi)
#> [1] 3.142
trapezoid(sin, 0, pi)
#> [1] 1.924e-16
```

两个函数都没有给出正确答案。为了使答案更加准确，可以先将整个积分区间分成很多小区间，然后再对每一个小区间进行积分。这称为**组合积分**（composite integration）。使用下面两个新函数来实现它：

```
midpoint_composite <- function(f, a, b, n = 10) {
  points <- seq(a, b, length = n + 1)
  h <- (b - a) / n

  area <- 0
  for (i in seq_len(n)) {
    area <- area + h * f((points[i] + points[i + 1]) / 2)
  }
  area
}

trapezoid_composite <- function(f, a, b, n = 10) {
  points <- seq(a, b, length = n + 1)
  h <- (b - a) / n

  area <- 0
  for (i in seq_len(n)) {
    area <- area + h / 2 * (f(points[i]) + f(points[i + 1]))
  }
  area
}

midpoint_composite(sin, 0, pi, n = 10)
#> [1] 2.008
midpoint_composite(sin, 0, pi, n = 100)
#> [1] 2
trapezoid_composite(sin, 0, pi, n = 10)
#> [1] 1.984
trapezoid_composite(sin, 0, pi, n = 100)
#> [1] 2
```

注意 `midpoint_composite()` 和 `trapezoid_composite()` 之间有大量的重复。除了在大区间内积分使用的内部规则外，它们基本上是相同的。从这些特定的函数中，可以提取一个更通用的组合积分函数：

```
composite <- function(f, a, b, n = 10, rule) {
  points <- seq(a, b, length = n + 1)

  area <- 0
  for (i in seq_len(n)) {
    area <- area + rule(f, points[i], points[i + 1])
  }

  area
}

composite(sin, 0, pi, n = 10, rule = midpoint)
#> [1] 2.008
composite(sin, 0, pi, n = 10, rule = trapezoid)
#> [1] 1.984
```

这个函数以两个函数作为参数：要积分的函数和积分规则。现在可以为在更小区间内的积分添加更好的积分规则：

```
simpson <- function(f, a, b) {
  (b - a) / 6 * (f(a) + 4 * f((a + b) / 2) + f(b))
}

boole <- function(f, a, b) {
  pos <- function(i) a + i * (b - a) / 4
  fi <- function(i) f(pos(i))

  (b - a) / 90 *
    (7 * fi(0) + 32 * fi(1) + 12 * fi(2) + 32 * fi(3) + 7 * fi(4))
}

composite(sin, 0, pi, n = 10, rule = simpson)
#> [1] 2
composite(sin, 0, pi, n = 10, rule = boole)
#> [1] 2
```

上述中点法则、梯形法则、Simpson 法则和 Boole 法则都是一个更通用的称为牛顿 – 柯特斯规则（http://en.wikipedia.org/wiki/Newton%E2%80%93Cotes_formulas）的系列特例。（它们是更复杂的多项式。）可以使用这个共同的结构编写一个函数，这个函数可以产生任意通用的牛顿 – 柯特斯（Newton-Cotes）规则：

```
newton_cotes <- function(coef, open = FALSE) {
  n <- length(coef) + open

  function(f, a, b) {
    pos <- function(i) a + i * (b - a) / n
    points <- pos(seq.int(0, length(coef) - 1))

    (b - a) / sum(coef) * sum(f(points) * coef)
  }
}
```

```
boole <- newton_cotes(c(7, 32, 12, 32, 7))
milne <- newton_cotes(c(2, -1, 2), open = TRUE)
composite(sin, 0, pi, n = 10, rule = milne)
#> [1] 1.994
```

从数学的角度，对数值积分进行优化的下一步是将均匀分布的点变为在区间终点附近的更密集的点，例如高斯积分。这超出了本案例研究的范围，但是你可以使用相同的技术实现它。

练习

1. 除了创建单独的函数（例如，`midpoint()`、`trapezoid()`、`simpson()`等）外，还可以将它们存储在列表中。如果这样做，应该如何修改代码？你能根据系数列表为牛顿 – 柯特斯规则产生函数列表吗？
2. 不同积分规则之间进行取舍是越复杂的规则，计算速度越慢，但需要更少的区间。例如，对于`sin()`函数在 $[0, \pi]$ 之间积分，确定需要的区间数以便每个规则的精度相同。用图来展示你的结果。对于不同的函数，它们是如何改变的？函数`sin(1 / x^2)`非常具有挑战性。

193 ~ 198

第 11 章
泛　函

“为了使程序变得更加可靠，代码就必须变得更加透明。尤其是，嵌套条件和循环必须认真审视。复杂的控制流会让程序员迷惑。杂乱的代码经常隐藏着漏洞。”

——Bjarne Stroustrup

高阶函数就是以函数作为输入并以函数作为输出的函数。我们已经学习了一种类型的高阶函数：闭包，由另一个函数返回的函数。闭包的一个补充就是**泛函**（functional），以函数作为输入并返回一个向量的函数。这里有一个简单的泛函：它调用输入的函数，假设输入是 1 000 个随机均匀数。

```
randomise <- function(f) f(runif(1e3))
randomise(mean)
#> [1] 0.4897
randomise(mean)
#> [1] 0.4858
randomise(sum)
#> [1] 497.5
```

很可能你已经使用过泛函。3 个最常用的泛函为：`lapply()`、`apply()` 和 `tapply()`。它们都可以接收一个函数作为输入，并返回一个向量作为输出。

泛函的常用功能之一就是替代循环。在 R 语言中，循环的口碑很差。它们以慢著称（虽然这有点儿片面，详见第 18.4 节）。但循环的最大缺点是，它的表达不够清晰。for 循环是对某件事进行迭代，但不能清晰地表达更高层次的目的。除了使用循环外，泛函是更好的选择。每个泛函都是为一个特殊任务量身定做的，所以当你认识了泛函后你也就知道了为什么这里要使用它。泛函除了可以做 for 循环的替代外还可以扮演其他角色。它们还可以用来对公共数据的操作任务进行封装，例如，分割 – 应用 – 组合（split-apply-combine），以函数的方式思考并能处理数学函数。

由于泛函可以很清晰地表达程序的意图，所以它能减少漏洞的出现。由于使用的人很多，所以基础包中实现的泛函都经历了严格的测试（即，没有漏洞）并且效率很高。其中很多都是使用 C 语言编写的，并使用了一些特殊的技巧来提高性能。也就是说，使用泛函不一定产生最快的代码。但是，它可以帮助我们清晰地表达我们的想法，并创建可以解决很多问题的工具。当我们发现程序的执行速度可能是一个问题时，才考虑它是不对的。一旦我们有

了清晰准确的代码，我们就可以使用第 17 章中学习的技术来使它运行得更快。

主要内容

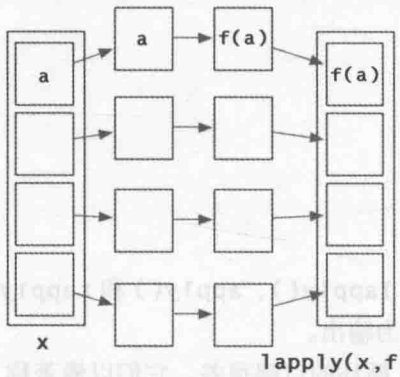
- 11.1 节介绍第一个泛函：`lapply()`。
- 11.2 节学习 `lapply()` 的变体，它可以接收不同的输入并产生不同的输出，还可以以不同的方法进行分布式计算。
- 11.3 节学习可以处理更复杂数据结构（例如，矩阵和数组）的泛函。
- 11.4 节学习强大的 `Reduce()` 和 `Filter()` 函数，它们在处理列表时很有用。
- 11.5 节学习一些数学上很常见的泛函，如求根、积分和最优化。
- 11.6 节告诉我们什么时候不应该将循环转换成泛函。
- 11.7 节作为本章的结束，将看到泛函如何使用简单的构建单元来创建一组强大而稳定的工具。

预备知识

200 闭包将频繁地与泛函联合使用。如果你需要补习相关知识，请阅读 10.3 节。

11.1 第一个泛函：lapply()

最简单的泛函就是 `lapply()`，你可能已经对它很熟悉了。`lapply()` 接收一个函数，并将这个函数应用到列表中的每一个元素，最后再将结果以列表的形式返回。`lapply()` 是很多其他函数的基本组件，所以了解它是如何工作的是很重要的。下面是一个图示：



出于对性能方面的考虑，`lapply()` 使用 C 语言实现，但是可以使用 R 来创建一个功能相同的简化版本：

```
lapply2 <- function(x, f, ...) {
  out <- vector("list", length(x))
  for (i in seq_along(x)) {
    out[[i]] <- f(x[[i]], ...)
  }
  out
}
```

从这段代码中，我们可以看出 `lapply()` 是对一个常见 for 循环模式的包装器：为输出创建一个容器，将 `f()` 应用到列表中的每一个元素，将结果填充到容器中。其他 for 循环泛函都是这一思路的变体：它们简单地使用不同类型的输入或输出。

`lapply()` 通过大量地减少与循环相关的引用代码，使得我们可以更容易地处理列表。

这使我们可以将注意力集中在将要应用的函数上：

201

```
# Create some random data
l <- replicate(20, runif(sample(1:10, 1)), simplify = FALSE)

# With a for loop
out <- vector("list", length(l))
for (i in seq_along(l)) {
  out[[i]] <- length(l[[i]])
}
unlist(out)
#>  [1]  8  8  1  4  6  6  9  8  9 10  2  6  4  3  2  7  3 10
#> [20]  9

# With lapply
unlist(lapply(l, length))
#>  [1]  8  8  1  4  6  6  9  8  9 10  2  6  4  3  2  7  3 10
#> [20]  9
```

（这里使用 `unlist()` 将输出从列表转换为向量，使其更加简洁。很快我们将学习其他可以将输出向量化的方法。）

因为数据框也是列表，所以当我们想对数据框中的每一列进行处理时，`lapply()` 也有用：

```
# What class is each column?
unlist(lapply(mtcars, class))
#>       mpg       cyl      disp        hp      drat        wt
#> "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
#>      qsec        vs        am      gear      carb
#> "numeric" "numeric" "numeric" "numeric" "numeric"

# Divide each column by the mean
mtcars[] <- lapply(mtcars, function(x) x / mean(x))
```

`x` 总是作为 `f` 的第一个参数。如果想改变参数，可以使用匿名函数。下面这个例子就是对一个固定的 `x` 进行不同程度的修剪，然后再计算其平均值。

```
trims <- c(0, 0.1, 0.2, 0.5)
x <- rcauchy(1000)
unlist(lapply(trims, function(trim) mean(x, trim = trim)))
#> [1]  1.48503 -0.07759 -0.04445 -0.03894
```

202

11.1.1 循环模式

有 3 种基本方法可以对一个向量进行循环操作：

1）对每一个元素进行循环：`for (x in xs)`。

2）根据元素的数值索引进行循环：`for (i in seq_along(xs))`。

3）根据元素的名字进行循环：`for (nm in names(xs))`。

对于循环来说，第一种方法通常不是一个好的选择，因为它会导致输出存储效率低。使用这种方法时，就会很自然地通过扩充已有的数据结构来保存新的结果，如下例所示（输出列表的大小会不断改变）：

```
xs <- runif(1e3)
res <- c()
for (x in xs) {
```

```
  # This is slow!
  res <- c(res, sqrt(x))
}
```

因为每次你对向量进行扩展时，R 不得不复制所有的现有元素，这样做很慢。17.7 节对此进行了深入讨论。所以，最好先为输出创建一个空向量（空间），然后再向其中添加元素。所以第二种方法是最简单的方法：

```
res <- numeric(length(xs))
for (i in seq_along(xs)) {
  res[i] <- sqrt(xs[i])
}
```

正如使用 for 循环有 3 种方法，使用 `lapply()` 也有 3 种方法：

```
lapply(xs, function(x) {})
lapply(seq_along(xs), function(i) {})
lapply(names(xs), function(nm) {})
```

通常情况下，选择第一种方法，因为 `lapply()` 会为我们保存结果。但是，如果需要知道元素的位置或名字，就应该选择第二种或第三种方法。它们都会给出元素的位置（`i`，`nm`）和对应的值（`xs[[i]]`,`xs[[nm]]`）。如果你正在努力地使用其中的一种方式来解决一个问题，也许你可以试试其他方法，可能会更简单。

203

11.1.2 练习

1. 为什么下面的两个 `lapply()` 是等价的？

```
trims <- c(0, 0.1, 0.2, 0.5)
x <- rcauchy(100)

lapply(trims, function(trim) mean(x, trim = trim))
lapply(trims, mean, x = x)
```

2. 下面的这个函数可以对一个向量进行量纲调整，使它的取值范围为 [0, 1]。如何将它应用到数据框的每一列？如何将它应用到数据框的每一个数值列？

```
scale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
```

3. 联合使用 for 循环、`lapply()` 以及存储在下面列表中的公式对 `mtcars` 进行线性模型拟合：

```
formulas <- list(
  mpg ~ disp,
  mpg ~ I(1 / disp),
  mpg ~ disp + wt,
  mpg ~ I(1 / disp) + wt
)
```

4. 联合使用 for 循环和 `lapply()` 将模型 mpg～disp 拟合下面列表中的每一个 mtcars 的自助抽样副本。可以不使用匿名函数吗？

```
bootstraps <- lapply(1:10, function(i) {
  rows <- sample(1:nrow(mtcars), rep = TRUE)
  mtcars[rows, ]
})
```

5. 对于前两个练习中的每个模型，使用下面的函数提取 R^2。

```
rsq <- function(mod) summary(mod)$r.squared
```

204

11.2 for 循环泛函：lapply() 的相似函数

使用泛函来取代 for 循环的关键是要认识到，常见的循环模式已经在现有的基本泛函中实现了。一旦你已经掌握了这些现有的泛函，下一步就是开始编写自己的泛函：如果你发现在你的代码中有很多地方出现了相同的循环模式，那你就应该将它提取出来，并使用泛函来代替。

下面几节建立在 `lapply()` 基础之上，并论讨：

- `lapply()` 的变体 `sapply()` 和 `vapply()`，输出可以是向量、矩阵和数组而不是列表。
- `Map()` 和 `mapply()` 以并行方式对多个**输入**数据结构进行迭代。
- `mclapply()` 和 `mcMap()`，`lapply()` 和 `Map()` 的并行版。
- 编写一个新函数 `rollapply()` 来解决一个新问题。

11.2.1 向量输出：sapply 和 vapply

`sapply()` 和 `vapply()` 与 `lapply()` 非常相似，除了它们输出的是原子向量外。`sapply()` 是通过猜测来设定输出的类型，而 `vapply()` 是通过一个附加参数来设定输出类型。`sapply()` 更适合在交互式数据分析中使用，因为它可以减少键盘输入。但是，如果在函数中使用它时，如果提供错误的输入类型就可能导致一些奇怪的错误。`vapply()` 虽然更冗长，但是它可以给出更有意义的出错信息，并且不会出现程序失败而不给出任何信息。所以它更适合应用在其他函数的内部。

下面的例子说明这些不同点。对于一个给定的数据框，`sapply()` 和 `vapply()` 返回相同的结果。对于一个给定的空列表，`sapply()` 返回另一个空列表而不是一个长度为 0 的逻辑向量。

```
sapply(mtcars, is.numeric)
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#> TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
vapply(mtcars, is.numeric, logical(1))
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#> TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
sapply(list(), is.numeric)
#> list()
vapply(list(), is.numeric, logical(1))
#> logical(0)
```

如果函数返回不同类型或者不同长度的结果，`sapply()` 就会静静地返回一个列表，而 `vapply()` 会抛出一个错误。`sapply()` 适合使用在交互式数据分析中，因为我们可以很容易地发现有些结果是不对的，但是在编写函数时，这样做是很危险的。

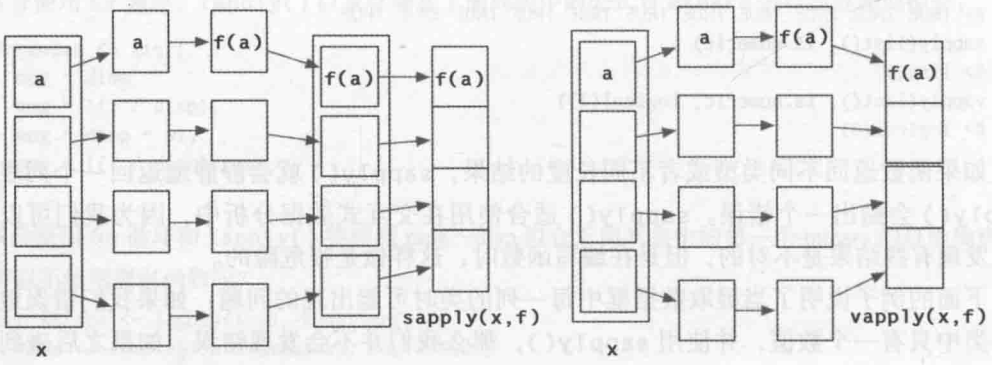
下面的例子说明了当提取数据框中每一列的类时可能出现的问题：如果我们错误地认为那个类中只有一个数值，并使用 `sapply()`，那么我们并不会发现错误，如果之后碰到某些函数给出的不是字符向量而是一个列表时就会发现这个问题。

```
df <- data.frame(x = 1:10, y = letters[1:10])
sapply(df, class)
#>          x          y
#> "integer"   "factor"
vapply(df, class, character(1))
#>          x          y
#> "integer"   "factor"

df2 <- data.frame(x = 1:10, y = Sys.time() + 1:10)
sapply(df2, class)
#> $x
#> [1] "integer"
#>
#> $y
#> [1] "POSIXct" "POSIXt"
vapply(df2, class, character(1))
#> Error: values must be length 1,
#>  but FUN(X[[2]]) result is length 2
```

sapply()是对lapply()的一个简单包装，它在最后一步将列表转换成向量。vapply()是将结果赋值给适当类型的向量（或矩阵）而不是列表的Lapply()的实现。下面的代码使用R语言实现sapply()和vapply()的主要功能（真实的函数具有更好的错误处理和保留名等）。

205 ~ 206

```
sapply2 <- function(x, f, ...) {
  res <- lapply2(x, f, ...)
  simplify2array(res)
}

vapply2 <- function(x, f, f.value, ...) {
  out <- matrix(rep(f.value, length(x)), nrow = length(x))
  for (i in seq_along(x)) {
    res <- f(x[i], ...)
    stopifnot(
      length(res) == length(f.value),
      typeof(res) == typeof(f.value)
    )
    out[i, ] <- res
  }
  out
}
```



vapply()和sapply()与lapply()的输出不同，下一节讨论Map()，它与lapply

() 具有不同的输入。

11.2.2 多重输入：Map（和 mapply）

lapply() 中只有一个参数是可以改变的，其他参数都是固定的。这非常不适合解决某些问题。例如，当你有两个列表，一个列表是观测值，另一个列表是权重值时，如何计算它们的加权平均值？

```
# Generate some sample data
xs <- replicate(5, runif(10), simplify = FALSE)
ws <- replicate(5, rpois(10, 5) + 1, simplify = FALSE)
```

207

使用 lapply() 可以很容易地计算非加权平均值：

```
unlist(lapply(xs, mean))
#> [1] 0.4696 0.4793 0.4474 0.5755 0.4490
```

但是如何使用权重并计算加权平均值 weighted.mean() 呢？不能使用 lapply(x, means, w)，因为给 lapply() 提供的附加参数是传送给每个调用的。我们可以改变循环的形式：

```
unlist(lapply(seq_along(xs), function(i) {
  weighted.mean(xs[[i]], ws[[i]])
}))
#> [1] 0.4554 0.5370 0.4301 0.6391 0.4411
```

这样做是可以的，但是有一点儿笨拙。一个看上去更清晰的替代品就是 Map，它是 lapply() 的一个变体，其中所有参数都是可以改变的。所以可以写成：

```
unlist(Map(weighted.mean, xs, ws))
#> [1] 0.4554 0.5370 0.4301 0.6391 0.4411
```

注意这里参数的顺序有一点儿不同：函数是 Map() 的第一个参数，但在 lapply() 中是第二个参数。

这等价于

```
stopifnot(length(xs) == length(ws))
out <- vector("list", length(xs))
for (i in seq_along(xs)) {
  out[[i]] <- weighted.mean(xs[[i]], ws[[i]])
}
```

Map() 和 lapply() 之间存在着一种天然的等价关系，因为总是可以将 Map() 转换成对索引进行迭代的 lapply()。但是使用 Map() 可以更简洁和清晰地表达我们想要执行的操作。

当我们需要对两个（或者更多）列表（或数据框）进行同时处理时，Map 很有用。例如，对每一列进行标准化的另一种方法是，首先计算每一列的平均值，然后再除以它们的平均值。可以使用 lapply()，但是如果分两步来实现，就可以每一步都检查结果，当第一步非常复杂时，这样做是非常重要的。

208

```
mtmeans <- lapply(mtcars, mean)
mtmeans[] <- Map(`/`, mtcars, mtmeans)

# In this case, equivalent to
mtcars[] <- lapply(mtcars, function(x) x / mean(x))
```

如果有些参数应该是固定的或者是常量，那么可以使用匿名函数：

```
Map(function(x, w) weighted.mean(x, w, na.rm = TRUE), xs, ws)
```

在下一章中，我们将看到表达同样思想的更加紧凑的方法。

mapply

相对于 `Map()` 来说，你可能对 `mapply()` 更熟悉。我更喜欢 `Map()` 的理由是：

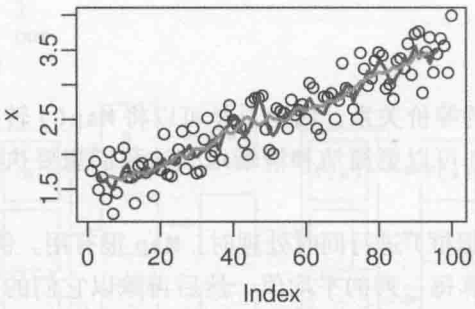
- 它等价于用 `simplify = FALSE` 进行 `mapply`，它总能给出我们想要的。
- 除了可以使用匿名函数来提供常量输入之外，`mapply` 还有一个 `MoreArgs` 参数，它接受一个由额外参数组成的列表，它可以接受一个提供给每次函数调用的额外参数列表。这打破了 R 的惰性求值原则，并且与其他函数也不一致。

简单地说，使用 `mapply()` 有点儿得不偿失。

11.2.3 滚动计算

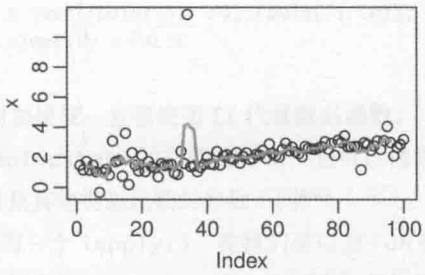
如果需要一个 for 循环的替代品，而基础包中又没有，那该怎么办呢？可以根据你对常用循环结构的认识以及对包装器的实现来自己构建一个。例如，你可能会对使用移动平均函数对数据进行平滑感兴趣：

```
rollmean <- function(x, n) {
  out <- rep(NA, length(x))

  offset <- trunc(n / 2)
  for (i in (offset + 1):(length(x) - n + offset - 1)) {
    out[i] <- mean(x[(i - offset):(i + offset - 1)])
  }
  out
}
x <- seq(1, 3, length = 1e2) + runif(1e2)
plot(x)
lines(rollmean(x, 5), col = "blue", lwd = 2)
lines(rollmean(x, 10), col = "red", lwd = 2)
```



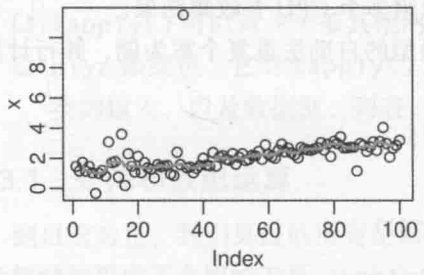
但是如果噪声变化大（即，它有一个长尾），你可能担心移动平均值会对异常值很敏感。另外，你可能想计算一个移动中位数。

```
x <- seq(1, 3, length = 1e2) + rt(1e2, df = 2) / 3
plot(x)
lines(rollmean(x, 5), col = "red", lwd = 2)
```



为了将 `rollmean()` 改为 `rollmedian()`，要做的只是在循环中将 `mean` 改为 `median`。但不是通过复制和粘贴来创建一个新函数，而是将计算移动汇总量的思想放入函数中：

```
rollapply <- function(x, n, f, ...) {
  out <- rep(NA, length(x))

  offset <- trunc(n / 2)
  for (i in (offset + 1):(length(x) - n + offset + 1)) {
    out[i] <- f(x[(i - offset):(i + offset)], ...)
  }
  out
}
plot(x)
lines(rollapply(x, 5, median), col = "red", lwd = 2)
```



你可能已经注意到这个内部循环看起来非常像 vapply() 循环，所以可以将这个函数重写为：

```
rollapply <- function(x, n, f, ...) {
  offset <- trunc(n / 2)
  locs <- (offset + 1):(length(x) - n + offset + 1)
  num <- vapply(
    locs,
    function(i) f(x[(i - offset):(i + offset)], ...),
    numeric(1)
  )

  c(rep(NA, offset), num)
}
```

这实际上与 `zoo::rollapply()` 类似，只是后者提供更多的特性和更多的错误检查。

11.2.4 并行化

实现 `lapply()` 时很有意思的一点是，由于每一次迭代都独立于其他迭代，所以迭代的顺序是不重要的。例如，`lapply3()` 就打破了原有的计算顺序，但是结果还是一样的：

```
lapply3 <- function(x, f, ...) {
  out <- vector("list", length(x))
```

```
  for (i in sample(seq_along(x))) {
    out[[i]] <- f(x[[i]], ...)
  }
  out
}
unlist(lapply(1:10, sqrt))
#>  [1] 1.000 1.414 1.732 2.000 2.236 2.449 2.646 2.828 3.000
#> [10] 3.162
unlist(lapply3(1:10, sqrt))
#>  [1] 1.000 1.414 1.732 2.000 2.236 2.449 2.646 2.828 3.000
#> [10] 3.162
```

这就产生一个非常重要的结果：因为可以按任意顺序进行计算，所以可以将这些计算任务分配给多个CPU从而实现并行计算。这就是parallel::mclapply()（和parallel::mcMap()）所做的。（这些函数不能在Windowns上运行，但是可以做更多的工作，使用类似的parLapply()函数来实现相同的效果。更详细的信息请参见17.10节）。

```
library(parallel)
unlist(mclapply(1:10, sqrt, mc.cores = 4))
#>  [1] 1.000 1.414 1.732 2.000 2.236 2.449 2.646 2.828 3.000
#> [10] 3.162
```

本例中，mclapply()实际上比lapply()慢。这主要是因为每个独立计算的开销都很小，而R还要再做一些额外的工作来实现将计算分派给多个CPU并收集结果。

209 ~ 212

如果我们使用一个更现实的例子，以生成线性模型的自助法重复个案为例，并行计算的优势就非常明显了：

```
boot_df <- function(x) x[sample(nrow(x), rep = T), ]
rsquared <- function(mod) summary(mod)$r.square
boot_lm <- function(i) {
  rsquared(lm(mpg ~ wt + disp, data = boot_df(mtcars)))
}

system.time(lapply(1:500, boot_lm))
#>    user  system elapsed
#>   0.776   0.003   0.779
system.time(mclapply(1:500, boot_lm, mc.cores = 2))
#>    user  system elapsed
#>   0.001   0.002   0.427
```

虽然增加CPU的核数并不总是能够带来性能的线性改善，但是使用lapply()或Map()的并行形式可以显著地提高计算性能。

11.2.5 练习

1. 使用vapply()：

   a）计算一个数值数据框中的每一列的标准差。

   b）计算一个混合数据框中每一个数值列的标准差。（提示：需要使用vapply()两次。）

2. 为什么使用sapply()获取数据框中每个元素的class()是危险的？

3. 下面的代码模拟对非正态数据t检验的性能。使用sapply()和一个匿名函数来提取每一次试验的p值。

```
trials <- replicate(
  100,
```

```
  t.test(rpois(10, 10), rpois(7, 10)),
  simplify = FALSE
)
```

附加挑战：直接使用 `[[` 代替匿名函数。

4. `replicate()` 的功能是什么？它可以消除哪种类型的 for 循环？为什么它的参数与 `lapply()` 函数以及其他类似函数的参数不同？ 213

5. 编写一个 `lapply()`，使我们可以为 FUN 提供每一个组成成分的名字和值。

6. 联合使用 `Map()` 和 `vapply()` 创建一个 `lapply()` 变体，使它可以并行处理所有的输入并将输出存储到一个向量中（或者一个矩阵）。这个函数应该有哪些参数呢？

7. 实现 `mcsapply()`，它是一个多核版的 `sapply()`。你能实现 `mcvapply()` 吗？它是一个并行版的 `vapply()`。为什么可以或者为什么不可以？

11.3 操作矩阵和数据框

在通常的数据操任务中也可以使用泛函来减少循环。在本节中，我们将对所有可选项做一个简要的总结，看看它们如何帮助我们，并为我们指出正确的学习方向。这里涵盖三类数据结构泛函：

- `apply()`、`sweep()` 和 `outer()` 处理矩阵。
- `tapply()` 可以对一个被其他向量分组的向量进行汇总。
- `plyr` 添加包，它对 `tapply()` 进行推广，使它更适合处理数据框、列表，或者数组类的输入，以及数据框、列表，或者数组类的输出。

11.3.1 矩阵和数组运算

到目前为止，我们见过的所有泛函都处理一维输入结构。本节介绍的 3 个泛函为处理高维数据结构提供了有用的工具。`apply()` 是 `sapply()` 的变体，它可以处理矩阵和数组。可以把它想象成用一个数来描述矩阵或者数组的每一列或每一行的汇总操作。它有 4 个参数：

- `X`，要进行汇总的矩阵或数组。
- `MARGIN`，一个整数向量，它用来设定需要进行汇总的维度，1＝行，2＝列，等等。 214
- `FUN`，一个汇总函数。
- `...`，传递给 FUN 的其他参数。

```
a <- matrix(1:20, nrow = 5)
apply(a, 1, mean)
#> [1]  8.5  9.5 10.5 11.5 12.5
apply(a, 2, mean)
#> [1]  3  8 13 18
```

使用 `apply()` 时有一些需要注意的地方。它没有简化参数，所以永远不可能完全确定将得到什么类型的输出。这就意味着在函数内部使用 `apply()` 是不安全的，除非对输入进行仔细检查。如果汇总函数是恒等运算符，`apply()` 在某种意义上也不是幂等（idempotent）的，输出也不总是与输入一致：

```
a1 <- apply(a, 1, identity)
identical(a, a1)
#> [1] FALSE
identical(a, t(a1))
```

```
#> [1] TRUE
a2 <- apply(a, 2, identity)
identical(a, a2)
#> [1] TRUE
```

（使用 `aperm()` 可以将高维数组放回到原来的位置，或者使用 `plyr::aaply()`，它是幂等的。）

`sweep()` 可以对统计汇总的值进行扫描。它经常和 `apply()` 一起使用对数组进行标准化。下面的例子对矩阵的行进行量纲调整，使所有值都位于 0～1 之间。

```
x <- matrix(rnorm(20, 0, 10), nrow = 4)
x1 <- sweep(x, 1, apply(x, 1, min), `-`)
x2 <- sweep(x1, 1, apply(x1, 1, max), `/`)
```

最后一个矩阵泛函是 `outer()`。它和前面几个有点儿不同，它可以接受多个向量输入并
215 创建一个矩阵或数组输出，其中输入函数对输入的所有可能组合进行运算：

```
# Create a times table
outer(1:3, 1:10, "*")
#>      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
#> [1,]    1    2    3    4    5    6    7    8    9    10
#> [2,]    2    4    6    8   10   12   14   16   18    20
#> [3,]    3    6    9   12   15   18   21   24   27    30
```

学习 `apply()` 及其相关函数的好地方有：

- Peter Werner 的“Using apply, sapply, lapply in R”（http://petewetner.blgspot.com/2012/12/using-apply-sapply-lapply-in-r.html）。
- Slawa Rokicki 的“The infamous apply function”（http://rforpublichealth.blogapot.no/2012/09/the-infamous-apply-function.html）。
- axiomOfChoice 的“The R apply function-a tutorial with examples”（http://forgetfulfunctor.blogspot.com/2011/07/r-apply-function-tutorial-with-examples.html）。
- The stackoverflow question “R Grouping functions: sapply vs. lapply vs. apply vs. tapply vs. by vs. aggregate”（http://stackoverflow.com/questions/3505701）。

11.3.2 组应用

可以把 `tapply()` 想象成 `apply()` 的一般化，它可以应用于不规则的数组，数组中的每一行可以有不同的列数。当对数据集进行汇总时经常需要使用这个函数。例如，假设你已经从一个医学实验中获得了一些脉搏速率数据，并希望在两组之间进行比较：

```
pulse <- round(rnorm(22, 70, 10 / 3)) + rep(c(0, 5), c(10, 12))
group <- rep(c("A", "B"), c(10, 12))

tapply(pulse, group, length)
#>  A  B
#> 10 12
tapply(pulse, group, mean)
#>    A    B
#> 71.6 74.5
```

216 `tapply()` 首先根据一组输入创建一个不规则的数据结构，然后将函数应用于这个数据结构的每一个元素。第一步实际上是由 `split()` 函数完成的。它接受两个输入并根据第二个向量对第一个向量进行分组，最后将已分组的结果以一个列表的形式返回。

```
split(pulse, group)
#> $A
#>  [1] 74 75 73 74 72 63 74 75 71 65
#>
#> $B
#>  [1] 72 72 73 77 73 78 68 78 75 77 71 80
```

那么 tapply() 就是 split() 和 sapply() 的组合：

```
tapply2 <- function(x, group, f, ..., simplify = TRUE) {
  pieces <- split(x, group)
  sapply(pieces, f, simplify = simplify)
}
tapply2(pulse, group, length)
#>  A  B
#> 10 12
tapply2(pulse, group, mean)
#>    A    B
#> 71.6 74.5
```

能够将 tapply() 重写为 split() 和 sapply() 的组合表明，我们已经确认了一些有用的组件。

11.3.3 plyr 添加包

使用基础泛函的一个挑战是，它们的开发都是比较随意的（没有严格的规范），并且是由很多作者编写的。这就意味着它们之间可能存在着很多不一致：

- 函数 tapply() 和 sapply() 的简化参数为 simplify。mapply() 的简化参数称为 SIMPLIFY。而 apply() 没有这个参数。
- vapply() 是 sapply() 的一个变体，它允许对输出的类型进行描述，但是 tapply()、apply() 或 Map() 却没有对应的变体。 217
- 大多数基础泛函的第一个参数为向量，但是 Map() 的第一个参数为函数。

这就为学习这些运算符带来了挑战，因为我们不得不记住所有这些变体。另外，如果我们考虑输入和输出类型的所有可能的组合，那么基础包只涵盖了其中的一部分情况：

	列 表	数 据 框	数 组
列表	lapply()		sapply()
数据框	by()		
数组			apply()

这是创建 plyr 包的驱动力之一。它为提供具有一致命名参数的一致命名函数，并涵盖输入和输出数据结构的所有组合：

	列 表	数 据 框	数 组
列表	llply()	ldply()	laply()
数据框	dlply()	ddply()	daply()
数组	alply()	adply()	aaply()

这些函数的每一个都将输入分成多个部分，将函数应用于每一个部分，然后再将结果组合起来。总之，这个过程称作“分割 - 应用 - 结合”。可以参考“The Split-Apply-Combine

Strategy for Data Analysis”（http://www.jstatsoft.org/v40/i01/）了解更多相关知识，可以免费获取《Journal of Statistical Software》上的文章。

11.3.4 练习

1. `apply()` 是如何安排输出的？阅读文档并做一些实验。
2. 没有与 `split()+vapply()` 等价的函数。应该有吗？它什么时候是有用的？自己编写一个。
3. 编写一个纯 R 语言的 `split()` 函数。（提示：使用 `unique()` 和子集选取。）不使用循环可以吗？
4. 哪些类型的输入和输出被遗漏了？在你到 `plyr` 包（http://www.jstatsoft.org/v40/i01/）的文档中寻找答案之前来一次大脑风暴吧。

11.4 列表操作

还可以把泛函看作对列表进行：改变、选取子集和汇总的常用工具集。每个函数式编程语言都有 3 个工具来完成这些任务：`Map()`、`Reduce()` 和 `Filter()`。我们已经学习了 `Map()`，下面几节将学习 `Reduce()`，它是对双参数函数进行扩展的强大工具；以及 `Filter()` 函数，它是另外一个重要的泛函类中的成员，它用来处理判断，只能返回 `TRUE` 或者 `FALSE`。

11.4.1 Reduce()

`Reduce()` 通过递归调用一个函数 `f`，每次有两个参数，将一个向量 `x` 简化为一个值。首先使用 `f` 对 `x` 中的前两个元素进行计算，得到一个结果，然后使用 `f` 对这个结果和 `x` 的第三个元素进行计算，以此类推。`Reduce(f, 1:3)` 等价于 `f(f(1, 2), 3)`。缩减也称为折叠，因为它将一个列表中的相邻元素“折叠”到一起。

下面的两个例子说明 `Reduce` 如何处理中缀函数和前缀函数：

```
Reduce('+', 1:3) # -> ((1 + 2) + 3)
Reduce(sum, 1:3) # -> sum(sum(1, 2), 3)
```

`Reduce()` 的本质可以用下面这个简单的 for 循环来描述：

```
Reduce2 <- function(f, x) {
  out <- x[[1]]
  for(i in seq(2, length(x))) {
    out <- f(out, x[[i]])
  }
  out
}
```

真正的 `Reduce()` 函数更加复杂，因为它还包含一些其他参数：参数 `right` 可以使 reduce 从向量的右端开始；一个可选的初始值（`init`）以及是否输出中间结果（`accumulate`）的选项。

`Reduce()` 可以将只能处理两个输入的函数扩展成一个可以处理任意多个输入的函数。它对实现多种类型的递归操作很有用，例如合并和交集。（在最后的案例学习中，我们将学习它的另一个用途。）假设有一个数值向量列表，我们希望找到这个列表中的每一个元素都包含的值：

```
l <- replicate(5, sample(1:10, 15, replace = T), simplify = FALSE)
str(l)
#> List of 5
#>  $ : int [1:15] 4 10 7 9 3 10 8 7 2 2 ...
#>  $ : int [1:15] 10 3 10 2 2 6 9 6 4 4 ...
#>  $ : int [1:15] 1 5 9 9 1 8 5 7 5 7 ...
#>  $ : int [1:15] 6 2 10 5 6 3 1 6 1 2 ...
#>  $ : int [1:15] 10 5 10 7 7 1 9 9 9 7 ...
```

可以取每个元素的交集：

```
intersect(intersect(intersect(intersect(l[[1]], l[[2]]),
  l[[3]]), l[[4]]), l[[5]])
#> [1] 10  3
```

这样读起来太困难了。可以使用 Reduce()，它等价于：

```
Reduce(intersect, l)
#> [1] 10  3
```

11.4.2 判断泛函

判断就是只能返回 TRUE 或者 FALSE 的函数，如 is.character、all 或 is.NULL。判断泛函就是对一个列表或者数据框中的每一个元素进行判断。基础包中有 3 个很有用的判断泛函：Filter()、Find() 和 Position()。

219 ~ 220

- Filter() 只选择满足判断条件的元素。
- Find() 返回满足判断条件的第一个（或者，如果 right = TRUE，则最后一个）元素。
- Position() 返回满足判断条件的第一个（或者，如果 right = TRUE，则最后一个）元素的位置。

另一个有用的判断泛函是 where()，它是一个自定义泛函，可以根据列表（或数据框）和判断（条件）返回一个逻辑向量：

```
where <- function(f, x) {
  vapply(x, f, logical(1))
}
```

下面的例子说明如何使用这些泛函来处理数据框数据：

```
df <- data.frame(x = 1:3, y = c("a", "b", "c"))
where(is.factor, df)
#>     x     y
#> FALSE  TRUE
str(Filter(is.factor, df))
#> 'data.frame':    3 obs. of  1 variable:
#>  $ y: Factor w/ 3 levels "a","b","c": 1 2 3
str(Find(is.factor, df))
#>  Factor w/ 3 levels "a","b","c": 1 2 3
Position(is.factor, df)
#> [1] 2
```

11.4.3 练习

1. 为什么 is.na() 不是一个判断泛函？基础包中的哪个函数与 is.na() 的判断功能最接近？
2. 使用 Filter() 和 vapply() 创建一个函数，它可以对数据框中的每个数字列进行统计汇总。
3. which() 和 Position() 的关系是什么？ where() 和 Filter() 的关系是什么？

4. 编写一个函数 `Any()`，参数为一个列表和一个判断函数，如果判断函数返回 `TRUE`，它就返回 `TRUE`。
221 编写一个类似的 `All()`。

5. 编写一个 `span()` 函数（出自 Haskell）：给定一个列表 `x` 和一个判断函数 `f`，`span` 函数返回判断为 `TRUE` 的最长连续游程元素的位置。（提示：也许可以发现 `rle()` 是有帮助的。）

11.5 数学泛函

泛函在数学中非常常见。极限、最大值、求根（满足 `f(x) = 0` 的点集 `x`）以及定积分都是泛函：给定一个函数，它们返回一个值（或者一个数字向量）。乍看上去，这些函数与本章的主题（减少循环）不符，但是如果深入思考，就会发现在实现它们的算法中都包含迭代。

本节将使用一些 R 的内置数学泛函。这里有 3 个泛函对函数进行处理并返回一个单一的数值：

- `integrate()` 计算曲线 `f()` 下的面积。
- `uniroot()` 计算 `f()=0` 的点集。
- `optimise()` 计算 `f()` 最高点和最低点的位置。

现在以一个简单的函数 `sin()` 为例，看看它们是如何工作的：

```
integrate(sin, 0, pi)
#> 2 with absolute error < 2.2e-14
str(uniroot(sin, pi * c(1 / 2, 3 / 2)))
#> List of 5
#>  $ root      : num 3.14
#>  $ f.root    : num 1.22e-16
#>  $ iter      : int 2
#>  $ init.it   : int NA
#>  $ estim.prec: num 6.1e-05
str(optimise(sin, c(0, 2 * pi)))
#> List of 2
#>  $ minimum  : num 4.71
#>  $ objective: num -1
str(optimise(sin, c(0, pi), maximum = TRUE))
#> List of 2
#>  $ maximum  : num 1.57
#>  $ objective: num 1
```

在统计学中，最大似然估计（Maximum Likelihood Estimation，MLE）经常要用到最优化。在 MLE 中，有两个参数集合：数据，对于给定问题数据是固定的；参数，在试图寻找最大值的过程中它不断地改变。这两个参数集合使得这类问题非常适合使用闭包。最优化与闭包相结合给出了下面解决 MLE 问题的方法。

如果数据来源于泊松分布，下面的例子说明如何找到 λ 的最大似然估计。首先，创建一个函数工厂，给定一个数据集，它返回一个为参数 `lambda` 计算负对数似然（NLL）的函数。在 R 中，经常处理负数，因为 `optimise()` 默认计算最小值（最大值取负数就是最小值）。

```
poisson_nll <- function(x) {
  n <- length(x)
  sum_x <- sum(x)
  function(lambda) {
    n * lambda - sum_x * log(lambda) # + terms not involving lambda
  }
}
```

注意，闭包允许我们预计算一些关于数据的常数。

我们可以使用这个函数工厂来为输入数据产生具体的 NLL 函数。给定一个好的起始范围，`optimise()` 函数就能为我们找到最佳值（最大似然估计）。

```
x1 <- c(41, 30, 31, 38, 29, 24, 30, 29, 31, 38)
x2 <- c(6, 4, 7, 3, 3, 7, 5, 2, 2, 7, 5, 4, 12, 6, 9)
nll1 <- poisson_nll(x1)
nll2 <- poisson_nll(x2)

optimise(nll1, c(0, 100))$minimum
#> [1] 32.1
optimise(nll2, c(0, 100))$minimum
#> [1] 5.467
```

222 ~ 223

我们可以将这些值与解析解进行对比，从而检查它们是否正确：在本例中，它就是数据的平均值，32.1 和 5.4667。

另一个重要的数学泛函为 `optim()`。它是 `optimise()` 的一般化，可以用来处理多于一维的数据。如果你感兴趣它是如何工作的，你应该研究 `Rvmmin` 软件包，它提供了一个纯 R 语言版的 `optim()`。有趣的是，虽然它是使用 R 语言编写的，但 `Rvmmin` 并不比 `optim()` 慢。对于这个问题，瓶颈不在于对最优化的控制而是不得不对函数进行多次求值。

练习

1. 编写函数 `arg_max()`。它可以接收一个函数和一个向量作为参数，并返回使函数取得最大值的输入元素。例如，`arg_max(-10:5, function(x) x ^ 2)` 的返回值应该是 `-10`。`arg_max(-5:5, function(x) x ^ 2)` 应该返回 `c(-5, 5)`。再编写一个相应的 `arg_min()` 函数。
2. 挑战：阅读关于不动点算法（http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-12.html#%_sec_1.3）。使用 R 语言来完成其中的练习。

11.6 应该保留的循环

有些循环没有天然等价的泛函。本节将学习 3 个常见案例：

- 原位修改
- 递归函数
- while 循环

虽然这些问题可以使用泛函来解决，但是这并不是一个好主意。我们要为此编写一段令人非常难理解的代码，这与我们使用泛函的初衷是相违背的。

224

11.6.1 原位修改

如果需要对现有数据框的一部分进行修改，最好使用循环。例如，下面的代码根据函数列表的函数名与数据框中的变量名是否匹配来进行“变量到变量”的转换。

```
trans <- list(
  disp = function(x) x * 0.0163871,
  am = function(x) factor(x, levels = c("auto", "manual"))
)
for(var in names(trans)) {
  mtcars[[var]] <- trans[[var]](mtcars[[var]])
}
```

我们通常不会使用 `lapply()` 来直接替代这个循环，虽然这是可以做到的。只需要使用 `<<-` 用 `lapply()` 代替循环就可以：

```
lapply(names(trans), function(var) {
  mtcars[[var]] <<- trans[[var]](mtcars[[var]])
})
```

虽然没有 for 循环，但是代码更长了且更难理解。读者需要知道 `<<-`，以及 `x[[y]] <<- z` 是如何工作的（这并不简单）。简单地说，把一个简单的容易被人理解的 for 循环转变成了一个很少有人理解的泛函：不是一个好主意！

11.6.2 递归关系

当元素之间的关系不是相互独立的，或者是递归定义的时，将一个循环转变成泛函就会很难。例如，指数平滑是通过计算当前点和前一个点的加权平均值来实现的。下面的 `exps()` 函数就是采用 `for` 循环实现的指数平滑。

```
exps <- function(x, alpha) {
  s <- numeric(length(x) + 1)
  for (i in seq_along(s)) {
    if (i == 1) {
      s[i] <- x[i]
    } else {
      s[i] <- alpha * x[i - 1] + (1 - alpha) * s[i - 1]
    }
  }
  s
}
x <- runif(6)
exps(x, 0.5)
#> [1] 0.3607 0.3607 0.2030 0.1066 0.1931 0.2594 0.4287
```

这里的循环不能删除，因为没有泛函可以根据输入和第 `i - 1` 个输出来计算第 `i` 个输出。

删除本例中 `for` 循环的一种方法，就是通过删除递归并使用显式引用来替代它以便解决递归关系（http://en.wikipedia.org/wiki/Pecurrence_relation#Solving）。这要求一个新的数学工具，并具有挑战性，但它可以通过产生一个简单的函数实现。

11.6.3 `while` 循环

R 中另一种类型的循环结构就是：`while` 循环。直到条件满足它才终止运行。`while` 循环比 `for` 更常用：任何 `for` 循环都可以改写成 `while` 循环，但反过来就不行。例如，可以将下面的 `for` 循环：

```
for (i in 1:10) print(i)
```

变成 `while` 循环：

```
i <- 1
while(i <= 10) {
  print(i)
  i <- i + 1
}
```

但不是每一个 `while` 循环都可以转变成 `for` 循环，因为很多 `while` 循环在终止前并知

道它们要执行多少次：

```
i <- 0
while(TRUE) {
  if (runif(1) > 0.9) break
  i <- i + 1
}
```

当进行模拟时，这种问题很常见。

在本例中通过认识问题的一个特殊性质来删除循环。这里，在 p=0.1 的伯努利实验失败之前，我们计算实验成功的数目。它是一个几何随机变量，所以可以使用 `i <- rgeom(1, 0.1)` 来代替这段代码。这种解决方案不具有普适性，但是如果你能用此方法解决问题，那么你将获得很多优势。

11.7 创建一个函数系列

在本章结尾的案例学习中，我们来看看如何使用泛函将简单的组件变得功能强大而又通用。我们从一个非常简单的想法开始，对两个数做加法，使用泛函将它扩展到对多个数做加法、并行化计算、计算累积和以及在数组的不同维度之间进行加法运算。

从一个简单的加法函数开始，它可以接受两个标量参数：

```
add <- function(x, y) {
  stopifnot(length(x) == 1, length(y) == 1,
    is.numeric(x), is.numeric(y))
  x + y
}
```

（这里使用 R 现有的加法运算符，它还可以做更多事情，但在这里的重点是说明如何使用简单的组件以及如何对它们进行扩展使它们完成更大的任务。）

还要增加一个 `na.rm` 参数。帮助函数可以使这变得更容易：如果缺失 `x` 就返回 `y`，如果缺失 `y` 就返回 `x`，如果都缺失就为函数返回另一个参数：`identity`。现在这个函数变得比我们需要的更通用，但是这有利于我们编写其他二元运算符。

225 ~ 227

```
rm_na <- function(x, y, identity) {
  if (is.na(x) && is.na(y)) {
    identity
  } else if (is.na(x)) {
    y
  } else {
    x
  }
}
rm_na(NA, 10, 0)
#> [1] 10
rm_na(10, NA, 0)
#> [1] 10
rm_na(NA, NA, 0)
#> [1] 0
```

这允许我们编写一个能够处理缺失值的 `add()` 函数（如果需要）：

```
add <- function(x, y, na.rm = FALSE) {
  if (na.rm && (is.na(x) || is.na(y))) rm_na(x, y, 0) else x + y
}
add(10, NA)
```

```
#> [1] NA
add(10, NA, na.rm = TRUE)
#> [1] 10
add(NA, NA)
#> [1] NA
add(NA, NA, na.rm = TRUE)
#> [1] 0
```

为什么选择了0作为标识？为什么 `add(NA, NA, na.rm = TRUE)` 的返回值应该为0？对于每一个其他输入，它都返回一个数字，因此即使两个参数都是 `NA`，它也要这样做。它应该返回什么数呢？我们能够弄明白它，因为加法是符合结合律的，这就意味着加法的顺序是不重要的。这也就意味着下面两个函数调用返回相同的值：

```
add(add(3, NA, na.rm = TRUE), NA, na.rm = TRUE)
#> [1] 3
add(3, add(NA, NA, na.rm = TRUE), na.rm = TRUE)
#> [1] 3
```

228

这说明 `add(NA, NA, na.rm = TRUE)` 必须等于0，因此默认 `identity = 0` 是正确的。

既然基础部分已经可以工作了，那么将它扩展到处理更复杂输入的函数。一种明显的推广就是对多于两个数的输入做加法。使用迭代方式将两个数的加法扩展到多个数的加法：如果输入是 `c(1, 2, 3)`，计算 `add(add(1, 2), 3)`，这就是 `Reduce()` 的一个简单应用：

```
r_add <- function(xs, na.rm = TRUE) {
  Reduce(function(x, y) add(x, y, na.rm = na.rm), xs)
}
r_add(c(1, 4, 10))
#> [1] 15
```

看上去不错，不过需要使用一些具体的例子进行测试：

```
r_add(NA, na.rm = TRUE)
#> [1] NA
r_add(numeric())
#> NULL
```

这些都是不对的。在第一个例子中，虽然已经显式地要求忽略缺失值，但仍然得到了一个缺失值。在第二个例子中，结果为 `NULL`，而不是一个长度为0的数值向量。

这两个问题是相关的。如果 `Reduce()` 接收到的是一个长度为1的向量，它就什么也不做，所以它只是返回输出。如果输入的长度为0，它总是返回 `NULL`。解决这个问题的最简单方法就是使用 `Reduce()` 函数的 `init` 参数。将它添加到每一个输入向量的起始：

```
r_add <- function(xs, na.rm = TRUE) {
  Reduce(function(x, y) add(x, y, na.rm = na.rm), xs, init = 0)
}
r_add(c(1, 4, 10))
#> [1] 15
r_add(NA, na.rm = TRUE)
#> [1] 0
r_add(numeric())
#> [1] 0
```

`r_add()` 等价于 `sum()`。

最好有一个向量版的 `add()`，这样就可以用逐个元素的方式对两个向量数值进行加法。可以使用 `Map()` 或 `vapply()` 来实现它，但是它们都不完美。`Map()` 返回列表，而不是数

值向量，所以需要使用 `simplify2array()`。`vapply()` 返回向量，但它需要对所有的索引进行循环遍历。

```
v_add1 <- function(x, y, na.rm = FALSE) {
  stopifnot(length(x) == length(y), is.numeric(x), is.numeric(y))
  if (length(x) == 0) return(numeric())
  simplify2array(
    Map(function(x, y) add(x, y, na.rm = na.rm), x, y)
  )
}

v_add2 <- function(x, y, na.rm = FALSE) {
  stopifnot(length(x) == length(y), is.numeric(x), is.numeric(y))
  vapply(seq_along(x), function(i) add(x[i], y[i], na.rm = na.rm),
    numeric(1))
}
```

有些测试案例有助于确保程序总能按照期望的方式运行。这里的要求比 R 基础包的要求更加严格，因为我们不做资源回收。(如果需要可以加上，但是我发现资源回收是造成程序漏洞的常见来源。)

```
# Both versions give the same results
v_add1(1:10, 1:10)
#>  [1]  2  4  6  8 10 12 14 16 18 20
v_add1(numeric(), numeric())
#> numeric(0)
v_add1(c(1, NA), c(1, NA))
#> [1]  2 NA
v_add1(c(1, NA), c(1, NA), na.rm = TRUE)
#> [1] 2 0
```

`add()` 的另一变体是计算累积和。将 `Reduce()` 的 `accumulate` 参数设置为 `TRUE` 就可以了：

```
c_add <- function(xs, na.rm = FALSE) {
  Reduce(function(x, y) add(x, y, na.rm = na.rm), xs,
    accumulate = TRUE)
}
c_add(1:10)
#>  [1]  1  3  6 10 15 21 28 36 45 55
c_add(10:1)
#>  [1] 10 19 27 34 40 45 49 52 54 55
```

这等价于 `cumsum()`。

最后，我们可能需要为更复杂的数据结构（如矩阵）定义加法运算。我们可以创建 `row` 和 `col` 变体，它们分别按行或列进行求和，或者创建一个数组版的函数，它可以对任意维度的集合（同一维度内，不同维度间）求和。联合使用 `add()` 和 `apply()` 可以轻松实现这一点。

```
row_sum <- function(x, na.rm = FALSE) {
  apply(x, 1, add, na.rm = na.rm)
}
col_sum <- function(x, na.rm = FALSE) {
  apply(x, 2, add, na.rm = na.rm)
}
arr_sum <- function(x, dim, na.rm = FALSE) {
  apply(x, dim, add, na.rm = na.rm)
}
```

前两个函数等价于 `rowSums()` 和 `colSums()`。

如果我们创建的每个函数在 R 中都有等价的函数，为什么我们还要自己编写呢？这里有两个主要原因：

- 因为我们编写所有变体都是由简单的二元运算符（`add()`）和经过严格测试的泛函（`Reduce()`、`Map()`、`apply()`）构成的，所以我们知道它们的行为永远都是一致的。
- 我们可以将这些基础组件应用到其他运算符，特别是那些在 R 的基础包中没有完整变体的运算符。

这种方法的缺点是：效率不够高。（例如，`colSums(x)` 比 `apply(x, 2, sum)` 快很多。）但是，即使它们运行得比较慢，但这种简单实现方法对我们来说仍然是一个好的开始，因为它们几乎没有漏洞。在我们创建运行速度更快的函数时，我们可以对比这两个版本的结果，从而保证快速版的函数也能得到正确的结果。

229 ~ 231

如果你喜欢本节，你也应该喜欢下面这篇文章：“List out of lambda”（http://stevelosh.com/blog/2013/03/list-out-of-lambda/），这是 Steve Losh 的一篇博客文章，它说明如何使用更原始语言特性（例如，闭包、aka lambdas）来产生高级语言结构（比如，列表）。

练习

1. 编写 `smaller` 和 `larger` 函数，给定两个输入，它返回比较小或比较大的值。采用 `na.rm = TRUE` 参数：标识应该是什么呢？（提示：`smaller(x, smaller(NA, NA, na.rm = TRUE), na.rm = TRUE)` 肯定是 `x`，所以 `smaller(NA, NA, na.rm = TRUE)` 一定比所有的 `x` 大。使用 `smaller` 和 `larger` 来实现等价的 `min()`、`max()`、`pmin()`、`pmax()` 以及新函数 `row_min()` 和 `row_max()`。）
2. 创建一个表，它的列包含：`and`、`or`、`add`、`multiply`、`smaller` 和 `larger`；行包含：`binary operator`、`reducing variant`、`vectorised variant` 和 `array variants`。
   a）使用 R 基础包中能完成相应功能的函数来填充相应的表格。
   b）对比现有 R 函数的名字和参数。它们一致吗？如何对它们进行改进？
   c）如果这个表中有缺失的格子，编写函数完成相应的功能。
3. 如何使 `paste()` 适合这个结构中？构成 `paste()` 的标量二元函数是什么？`paste()` 的 `sep` 和 `collapse` 参数等价于什么？还有哪些 `paste` 变体在 R 中没有实现？

232

第 12 章
函数运算符

本章将要学习函数运算符（Function Operator，FO）。函数运算符以一个（或多个）函数作为输入，并返回一个函数作为输出。在某些方面，函数运算符与泛函相似：虽然它们并不是必需的，但是使用它们可以使代码变得更加易读，表达力更强，并且可以帮助我们提高代码的编写速度。函数运算符与泛函的主要不同是：泛函提取循环使用的通用模式，而函数运算符提取匿名函数使用的通用模式。

下面的代码展示了一个简单的函数运算符 `chatty()`。它包装一个函数，并创建一个可以将它的第一个参数打印输出的新函数。这个函数运算符是很有用的，因为它为我们提供了一个查看函数（例如，`vapply()`）运行状态的窗口。

```
chatty <- function(f) {
  function(x, ...) {
    res <- f(x, ...)
    cat("Processing ", x, "\n", sep = "")
    res
  }
}
f <- function(x) x ^ 2
s <- c(3, 2, 1)
chatty(f)(1)
#> Processing 1
#> [1] 1

vapply(s, chatty(f), numeric(1))
#> Processing 3
#> Processing 2
#> Processing 1
#> [1] 9 4 1
```

在上一章中，我们学习了很多 R 语言的内置泛函，如 `Reduce()`、`Filter()` 和 `Map()`，它们有很少的参数，所以我们不得不使用匿名函数来修改它们的工作方式。在本章中，我们将为常见的匿名函数构建专用替代品，使我们可以更清楚地表达自己的意图。例如，在 11.2.2 节中，将匿名函数和 `Map()` 一起使用来提供固定参数：


```
Map(function(x, y) f(x, y, zs), xs, ys)
```

在本章的后半部分，我们将使用 `partial()` 函数来学习局部应用（partial application）。

局部应用将匿名函数进行封装来提供默认参数，使我们可以写出简洁的代码：

```
Map(partial(f, zs = zs), xs, yz)
```

这是函数运算符的一个重要应用：通过变换输入函数，可以删除函数的参数。实际上，只要函数的输入和输出仍然相同，这种方法可以使泛函具有更大的扩展性。

本章包含了4种重要类型的函数运算符：行为、输入、输出以及组合。对于每种类型，我都会为你展示一些有用的函数运算符，并告诉你如何使用它们来分解问题：对多个函数进行组合而不是对参数进行组合。目的不是将所有的函数运算符列出来，而是说明它们如何与其他函数式编程技术联合使用。对于你自己的工作，应该思考并进行一些实验来看看函数运算符能否帮助你解决一些经常出现的问题。

主要内容

- 12.1节介绍改变一个函数行为的函数运算符，例如，自动将使用日志记录到硬盘或者保证一个函数只运行一次。
- 12.2节说明如何编写可以处理函数输出的函数运算符。它们可以完成一些简单的任务，如捕获错误；或者从根本上改变函数的功能。
- 12.3节学习如何使用函数运算符（如`Vectorize()`或`partial()`）来修改函数的输入。
- 12.4节使用函数组合和逻辑运算符并结合多个函数来说明函数运算符的强大力量。

234

预备条件

与从头编写函数运算符一样，本章使用`memoise`，`plyr`和`pryr`软件包中的函数运算符。运行`install.packages(c("memoise", "plyr", "pryr"))`来安装它们。

12.1 行为函数运算符

行为函数运算符不会改变函数的输入和输出，但是给函数添加一些附加的行为。在本节中，我们学习实现下面3种行为的函数：

- 增加延迟来避免服务器被请求淹没。
- 每*n*次调用将信息输出到控制台来帮助我们检查一个长时间运行的进程。
- 缓存上一步的计算结果来改善性能。

为了激发这些行为，假设我们需要下载很多URL的长向量。使用`lapply()`和`download_file()`可以很容易实现：

```
download_file <- function(url, ...) {
  download.file(url, basename(url), ...)
}
lapply(urls, download_file)
```

（`download_file()`是`utils::download.file()`的简单包装，它提供了一个合理的默认文件名。）

我们可能还想为这个函数增加许多有用的行为。如果列表很长，我们可能希望每10个URL就输出一个“.”，这样就可以知道程序还在运行。如果从互联网下载文件，我们可能希望在每两个请求之间增加一个小小的延迟，这样可以避免服务器负担过重。使用`for`循环来实现这些行为非常复杂。由于需要一个外部计数器，所以可以不再使用`lapply()`。

```
i <- 1
for(url in urls) {
  i <- i + 1
  if (i %% 10 == 0) cat(".")
  Sys.delay(1)
  download_file(url)
}
```

这段代码理解起来是有点儿困难的，因为不同的关注点（迭代、输出和下载）交叉在一起。在本节的剩下部分，我们将创建一些函数运算符，它们用来对这些行为进行封装，并将代码写成下面这样：

```
lapply(urls, dot_every(10, delay_by(1, download_file)))
```

可以很直接地实现 `delay_by()`，并且遵循本章中大多数函数运算的相同基本模板。

```
delay_by <- function(delay, f) {
  function(...) {
    Sys.sleep(delay)
    f(...)
  }
}
system.time(runif(100))
#>    user  system elapsed
#>   0.000   0.000   0.001
system.time(delay_by(0.1, runif)(100))
#>    user  system elapsed
#>     0.0     0.0     0.1
```

`dot_every()` 稍微有点儿复杂，因为它需要管理一个计数器。幸运的是，我们已经在 10.3.2 节中学习过了。

```
dot_every <- function(n, f) {
  i <- 1
  function(...) {
    if (i %% n == 0) cat(".")
    i <<- i + 1
    f(...)
  }
}
x <- lapply(1:100, runif)
x <- lapply(1:100, dot_every(10, runif))
#> ..........
```

235
~
236

注意在每个函数运算符中我都把函数设置为最后一个参数。在组合多个函数运算符时，这样做会使代码更加易读。如果函数是第一个参数，代码就不是：

```
download <- dot_every(10, delay_by(1, download_file))
```

而是

```
download <- dot_every(delay_by(download_file, 1), 10)
```

由于 `dot_every()` 的参数与它的调用相距较远，所以很难找到该函数对应的参数。有时它也称为 Dagwood sandwich（http://en.wikipedia.org/wiki/Dagwood_sandwich）问题：在两片面包（括号）之间添加了太多食物（超级长的参数）。

我已经尝试给这些函数运算符取了一些描述性的名字：delay-by（延迟 1 秒），dot-every（每 10 个输出一个点）。在你的代码中，你可以根据你的意图来为函数取一个更有意义（清

晰）的名字，这对于其他人（也包括你自己）阅读并理解你的代码很有帮助。

12.1.1 缓存

我们可能担心的另一个问题是：在下载多个文件时会不会多次下载同一个文件。可以对输入 URL 列表调用 `unique()` 来避免这一问题，或者手动管理将 URL 映射到结果的数据结构。另一种方法就是使用缓存：对函数进行修改使它自动将结果缓存。

```
library(memoise)

slow_function <- function(x) {
  Sys.sleep(1)
  10
}
system.time(slow_function())
#>    user  system elapsed
#>   0.000   0.000   1.001
system.time(slow_function())
#>    user  system elapsed
#>   0.000   0.000   1.001
fast_function <- memoise(slow_function)
system.time(fast_function())
#>    user  system elapsed
#>   0.000   0.001   1.002
system.time(fast_function())
#>    user  system elapsed
#>       0       0       0
```

缓存问题是计算机科学以内存换速度的典型例子。一个被缓存的函数可以运行得非常快，因为它存储以前的输入和输出，所以它使用更多的内存。

使用缓存的一个真实案例是计算斐波那契数列。斐波那契数列是递归定义的：最前面的两个值都是 1，后续值的计算公式为 $f(n)=f(n-1)+f(n-2)$。使用 R 语言的原始版本会非常慢，例如，`fib(10)` 要计算 `fib(9)` 和 `fib(8)`，`fib(9)` 又要计算 `fib(8)` 和 `fib(7)`，以此类推。因此，这个数列中的每一个值都要被计算很多次。对 `fib()` 进行缓存可以使它的计算变得非常快，因为每个值只需要计算一次。

```
fib <- function(n) {
  if (n < 2) return(1)
  fib(n - 2) + fib(n - 1)
}
system.time(fib(23))
#>    user  system elapsed
#>   0.071   0.003   0.074
system.time(fib(24))
#>    user  system elapsed
#>   0.108   0.000   0.108

fib2 <- memoise(function(n) {
  if (n < 2) return(1)
  fib2(n - 2) + fib2(n - 1)
})
system.time(fib2(23))
#>    user  system elapsed
#>   0.003   0.000   0.003
```

```
system.time(fib2(24))
#>    user  system elapsed
#>       0       0       0
```

237 ⌇ 238

当然也不是所有的函数都需要缓存。例如，如果一个随机数生成器被缓存了，那么生成的结果就不再是随机数了：

```
runifm <- memoise(runif)
runifm(5)
#> [1] 0.5916 0.2663 0.9651 0.4808 0.4759
runifm(5)
#> [1] 0.5916 0.2663 0.9651 0.4808 0.4759
```

在理解了 `memoise()` 后，就可以将它直接应用到我们的问题上：

```
download <- dot_every(10, memoise(delay_by(1, download_file)))
```

这样我们就得到了一个函数，它可以和 `lapply()` 一起使用。但是，如果 `lapply()` 内部的循环出现了问题，我们也很难知道程序的运行状况。下一节学习如何使用函数运算符来掀开它的面纱从而使我们看到它的内部。

12.1.2　捕获函数调用

使用泛函的一个挑战就是，我们看不到函数内部的执行情况。不可能像 `for` 循环那样很容易地查看它的内部。幸运的是，可以使用函数运算符与 `tee()` 来查看其内部执行情况。

函数 `tee()` 的定义如下，它有 3 个参数，这 3 个参数都是函数：`f`，要修改的函数；`on_input`，与函数 `f` 的输入一起被调用的函数；`on_output`，与函数 `f` 的输出一起被调用的函数。

```
ignore <- function(...) NULL
tee <- function(f, on_input = ignore, on_output = ignore) {
  function(...) {
    on_input(...)
    output <- f(...)
    on_output(output)
    output
  }
}
```

（这个函数的灵感来源于 UNIX 命令 `tee`，它可以用来分割文件操作流，所以可以将函数运行过程显示出来，并将中间结果保存在文件中。）

239

可以使用 `tee()` 来查看泛函 `uniroot()` 的内部，查看它如何通过迭代得到结果。下面的例子可以找到 `x` 和 `cos(x)` 的交集：

```
g <- function(x) cos(x) - x
zero <- uniroot(g, c(-5, 5))
show_x <- function(x, ...) cat(sprintf("%+.08f", x), "\n")

# The location where the function is evaluated:
zero <- uniroot(tee(g, on_input = show_x), c(-5, 5))
#> -5.00000000
#> +5.00000000
#> +0.28366219
```

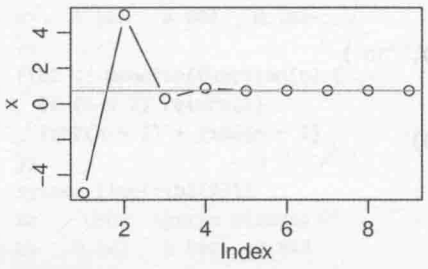
```
#> +0.87520341
#> +0.72298040
#> +0.73863091
#> +0.73908529
#> +0.73902425
#> +0.73908529
# The value of the function:
zero <- uniroot(tee(g, on_output = show_x), c(-5, 5))
#> +5.28366219
#> -4.71633781
#> +0.67637474
#> -0.23436269
#> +0.02685676
#> +0.00076012
#> -0.00000026
#> +0.00010189
#> -0.00000026
```

在函数运行时，`cat()`可以让我们查看函数的运行情况，但是在函数运行后我们就不能对这些数据进行处理。为了做到这一点，可以创建一个函数`remember()`，让它来捕获调用序列，它记录每一个被调用的参数并对它们进行检索。这里需要少量的S3代码，这些代码已经在7.2节中解释过了。

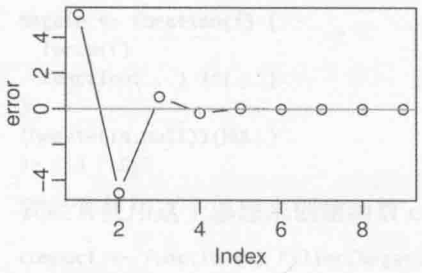
```
remember <- function() {
  memory <- list()
  f <- function(...) {
    # This is inefficient!
    memory <<- append(memory, list(...))
    invisible()
  }

  structure(f, class = "remember")
}
as.list.remember <- function(x, ...) {
  environment(x)$memory
}
print.remember <- function(x, ...) {
  cat("Remembering...\n")
  str(as.list(x))
}
```

现在我们可以绘制一幅图来展示`unitroot`函数是如何得到最终解的：

```
locs <- remember()
vals <- remember()
zero <- uniroot(tee(g, locs, vals), c(-5, 5))
x <- unlist(as.list(locs))
error <- unlist(as.list(vals))
plot(x, type = "b"); abline(h = 0.739, col = "grey50")
```



```
plot(error, type = "b"); abline(h = 0, col = "grey50")
```



12.1.3　惰性

我们目前看到的函数运算符都遵循下面这个常见模式：

```
funop <- function(f, otherargs) {
  function(...) {
    # maybe do something
    res <- f(...)
    # maybe do something else
    res
  }
}
```

不幸的是，这种实现方法有一个问题，因为函数的参数总是被惰性求值的：在使用函数运算符与对函数求值之间，`f()` 可能已经发生改变。如果通过 `for` 循环或 `lapply()` 来使用多个函数运算符，就会产生这种特殊问题。在下面的例子中，有一个函数列表，每一个都有延迟。但是当我们试图计算平均值时，还是得到总和。

```
funs <- list(mean = mean, sum = sum)
funs_m <- lapply(funs, delay_by, delay = 0.1)

funs_m$mean(1:10)
#> [1] 55
```

240 ~ 242

可以通过对 `f()` 进行显式的强制求值来避免这个问题：

```
delay_by <- function(delay, f) {
  force(f)
  function(...) {
    Sys.sleep(delay)
    f(...)
  }
}

funs_m <- lapply(funs, delay_by, delay = 0.1)
funs_m$mean(1:10)
#> [1] 5.5
```

当创建一个新的函数运算符时，这样做比较好。

12.1.4　练习

1. 编写一个可以将函数每次运行的时间戳和信息保存到文件中的函数运算符。
2. 下面这个函数的功能是什么？应该给它取一个什么名字？

```
f <- function(g) {
  force(g)
  result <- NULL
  function(...) {
    if (is.null(result)) {
      result <<- g(...)
    }
    result
  }
}
runif2 <- f(runif)
runif2(5)
#> [1] 0.01581 0.33986 0.55830 0.45425 0.05724
runif2(10)
#> [1] 0.01581 0.33986 0.55830 0.45425 0.05724
```

3. 对 `delay_by()` 进行修改，使它不是延迟一个固定的时间而是在函数最后一次调用后等待一个固定的时间。也就是说，如果调用：`g <- delay_by(1, f); g(); Sys.sleep(2); g()`，不应该有
243 附加的延迟。
4. 编写一个 `wait_until()`，它等待一个具体时间后才执行。
5. 有 3 个地方可以添加缓存调用：为什么我们只选择了其中一个？

```
download <- memoise(dot_every(10, delay_by(1, download_file)))
download <- dot_every(10, memoise(delay_by(1, download_file)))
download <- dot_every(10, delay_by(1, memoise(download_file)))
```

6. 为什么 `remember()` 函数效率低下？你能提高它的效率吗？
7. 为什么下面这段代码不能做我们想做的事？代码来源于 stackoverflow（http://stackoverflow.com/questions/8440675）。

```
# return a linear function with slope a and intercept b.
f <- function(a, b) function(x) a * x + b

# create a list of functions with different parameters.
fs <- Map(f, a = c(0, 1), b = c(0, 1))

fs[[1]](3)
#> [1] 4
# should return 0 * 3 + 0 = 0
```

如何对其修改，使其正确工作？

12.2 输出函数运算符

从复杂性上来说，下一步是对函数的输出进行修改。这可能是相当简单的，或者也可以从根本上改变函数的运算，从而返回完全不同于通常的输出。本节将学习两个简单的改变操
244 作，`Negate()` 和 `failwith()`，以及两个基础的改变操作，`capture_it()` 和 `time_it()`。

12.2.1 简单修饰

`base::Negate()` 和 `plyr::failwith()` 提供了两个简单但很有用的函数运算符，它与泛函一起使用非常方便。

`Negate()` 接收一个可以返回逻辑向量的函数（一个判断函数），并返回这个函数的结果的逆。当函数返回的结果与你想要的结果正好相反时，这样做会很方便。`Negate()` 的本质

很简单：

```
Negate <- function(f) {
  force(f)
  function(...) !f(...)
}
(Negate(is.null))(NULL)
#> [1] FALSE
```

我经常使用这个思想来创建函数 `compact()`，它可以去除列表中所有的 `null` 元素：

```
compact <- function(x) Filter(Negate(is.null), x)
```

当错误发生时，`plyr::failwith()` 可以将抛出错误的函数转换为另一个可以返回默认值的函数。同样，`failwith()` 的本质也很简单。它仅仅是 `try()` 的一个包装，它捕获错误并让程序继续执行。

```
failwith <- function(default = NULL, f, quiet = FALSE) {
  force(f)
  function(...) {
    out <- default
    try(out <- f(...), silent = quiet)
    out
  }
}
log("a")
#> Error: non-numeric argument to mathematical function
failwith(NA, log)("a")
#> [1] NA
failwith(NA, log, quiet = TRUE)("a")
#> [1] NA
```

245

（如果你还没有看到过 `try()` 函数，那么可以参见 9.3.1 节中的详细讨论。）

`failwith()` 与泛函一起使用非常有用：错误不会扩散，外层循环也不会终止，而是可以完成迭代并帮助我们找到错误的地方。假如我们正在将一个数据框列表拟合到一组广义线性模型（GLM）。虽然由于最优化问题，广义线性模型有时候会失败，但是我们仍然希望尝试拟合所有的模型，然后再查看拟合失败的数据：

```
# If any model fails, all models fail to fit:
models <- lapply(datasets, glm, formula = y ~ x1 + x2 * x3)
# If a model fails, it will get a NULL value
models <- lapply(datasets, failwith(NULL, glm),
  formula = y ~ x1 + x2 * x3)

# remove failed models (NULLs) with compact
ok_models <- compact(models)
# extract the datasets corresponding to failed models
failed_data <- datasets[vapply(models, is.null, logical(1))]
```

我觉得这是一个可以展现泛函与函数运算符强大功能的好例子：它可以非常简洁清晰地表达我们要解决的常见数据分析问题。

12.2.2　改变函数的输出

其他输出函数运算符能够对函数的运行效果产生更加重要的影响。除了返回初始返回值外，还可以让函数返回函数计算的一些其他结果。这里有两个例子：

❑ 返回函数 `print()` 输出的文本：

```
capture_it <- function(f) {
  force(f)
  function(...) {
    capture.output(f(...))
  }
}
str_out <- capture_it(str)
str(1:10)
#>  int [1:10] 1 2 3 4 5 6 7 8 9 10
str_out(1:10)
#> [1] " int [1:10] 1 2 3 4 5 6 7 8 9 10"
```

❑ 返回一个函数的运行时间：

```
time_it <- function(f) {
  force(f)
  function(...) {
    system.time(f(...))
  }
}
```

`time_it()` 允许我们可以对上一章中的一些代码进行修改：

```
compute_mean <- list(
  base = function(x) mean(x),
  sum = function(x) sum(x) / length(x)
)
x <- runif(1e6)

# Previously we used an anonymous function to time execution:
# lapply(compute_mean, function(f) system.time(f(x)))

# Now we can compose function operators:
call_fun <- function(f, ...) f(...)
lapply(compute_mean, time_it(call_fun), x)
#> $base
#>    user  system elapsed
#>   0.002   0.000   0.002
#>
#> $sum
#>    user  system elapsed
#>   0.001   0.000   0.001
```

在本例中，使用函数运算符的优势并不明显，因为它的组成成分非常简单，并且我们将相同的运算符应用到每一个函数。通常，当使用多个运算符时，或者创建它们与使用它们之间的间隔非常大时，使用函数运算符才非常有效。

246 ~ 247

12.2.3 练习

1. 创建一个 `negative()` 函数运算符，它可以将函数输出的符号反转。
2. `evaluate` 添加包可以很容易地捕获一个表达式的所有输出（结果、文本、消息、警告、错误和绘图）。创建一个与 `capture_it()` 类似的函数，它捕获一个函数产生的警告和错误信息。
3. 创建一个可以跟踪工作目录中文件创建和删除的函数（提示：使用 `dir()` 和 `setdiff()`）。你还要跟踪哪些函数的其他全局影响？

12.3　输入函数运算符

从复杂性上来说，下一步是对函数的输入进行修改。同样，我们可以仅对函数的工作方式稍做修改（例如，设置默认参数值），或者也可以是比较大的修改（例如，将标量输入转换成向量，或者向量转换成矩阵）。

12.3.1　预填充函数参数：局部函数应用

匿名函数经常用来创建一个具有特定（并且已经填充好）参数的函数的变体。这称为“局部函数应用”，它由 `pryr::partial()` 实现。一旦你已经阅读了第 14 章，我鼓励你阅读 `partial()` 的源代码并找出它的工作原理——它仅有 5 行代码！

`partial()` 允许我们将下面的代码：

```
f <- function(a) g(a, b = 1)
compact <- function(x) Filter(Negate(is.null), x)
Map(function(x, y) f(x, y, zs), xs, ys)
```

替换成

```
f <- partial(g, b = 1)
compact <- partial(Filter, Negate(is.null))
Map(partial(f, zs = zs), xs, ys)
```

可以使用这种思想来简化处理函数列表的代码。不必再写成：

248

```
funs2 <- list(
  sum = function(...) sum(..., na.rm = TRUE),
  mean = function(...) mean(..., na.rm = TRUE),
  median = function(...) median(..., na.rm = TRUE)
)
```

而写成：

```
library(pryr)
funs2 <- list(
  sum = partial(sum, na.rm = TRUE),
  mean = partial(mean, na.rm = TRUE),
  median = partial(median, na.rm = TRUE)
)
```

在很多函数式编程语言中，使用局部函数应用是一个很简单的工作，但不完全清楚它如何与 R 的惰性求值原则相互作用。`plyr::partial()` 采用的方法就是创建一个尽量与你手工创建的匿名函数相似的函数。Peter Meilstrup 在它的 ptools 软件包（https://github.com/crowding/ptools/）中采用了一个不同的方法。如果你对这个主题感兴趣，也许你想读一读他创建的二元运算符：`%()%`、`%>>%` 和 `%<<%`。

12.3.2　改变输入类型

还可以对函数的输入做很大的修改，使函数可以处理不同类型的数据。现在已经有一些函数可以做这样的工作：

- `base::Vectorize()` 可以将一个标量函数转换成一个向量函数。它接收非向量化函数并根据 `vectorize.args` 参数中设置的参数将其向量化。这并不会带来性能上的大幅提升，但是如果你需要一个快速构建向量化函数的方法，它将非常有用。

下面是对 sample() 的一个很有用的扩展，它可以根据数据的大小将其向量化。这样做可以让我们在一次调用中产生多个样本。

249

```
sample2 <- Vectorize(sample, "size", SIMPLIFY = FALSE)
str(sample2(1:5, c(1, 1, 3)))
#> List of 3
#>  $ : int 3
#>  $ : int 2
#>  $ : int [1:3] 1 4 3
str(sample2(1:5, 5:3))
#> List of 3
#>  $ : int [1:5] 4 3 5 2 1
#>  $ : int [1:4] 5 4 2 1
#>  $ : int [1:3] 1 4 2
```

在这个例子中，我们使用了 SIMPLIFY = FALSE 来确保新的向量化的函数总是返回一个列表。这通常就是我们想要的。

- splat() 将接收多个参数的函数转换成只接收一个参数列表的函数。

```
splat <- function (f) {
  force(f)
  function(args) {
    do.call(f, args)
  }
}
```

如果你希望使用变化的参数来调用一个函数时，它非常有用：

```
x <- c(NA, runif(100), 1000)
args <- list(
  list(x),
  list(x, na.rm = TRUE),
  list(x, na.rm = TRUE, trim = 0.1)
)
lapply(args, splat(mean))
#> [[1]]
#> [1] NA
#>
#> [[2]]
#> [1] 10.37
#>
#> [[3]]
#> [1] 0.4757
```

250

- plyr::colwise() 将向量函数转换成处理数据框的函数：

```
median(mtcars)
#> Error: need numeric data
median(mtcars$mpg)
#> [1] 19.2
plyr::colwise(median)(mtcars)
#>    mpg cyl  disp  hp  drat    wt  qsec vs am gear carb
#> 1 19.2   6 196.3 123 3.695 3.325 17.71  0  0    4    2
```

12.3.3 练习

1. 前面的 download() 函数一次只能下载一个文件。如何使用 partial() 和 lapply() 创建一个一次可以下载多个文件的函数？使用 partial() 与使用自己写的函数的优缺点各是什么？

2. 阅读 plyr::colwise() 的源代码。这段代码是如何工作的？ colwise() 的 3 个主要任务是什么？如何使用函数运算符来完成每一项任务从而简化 colwise()？（提示：考虑 partial()。）
3. 编写函数运算符，它将函数的返回值从数据框转换成矩阵；或者反过来。如果你已经学习了 S3，可以把它们称为 as.data.frame.function() 和 as.matrix.function()。
4. 我们已经看到了 5 个函数，它们可以将函数的输出从一种形式转变成另一种形式。它们分别是什么？将不同类型的输出组合成一张表：行和列分别是什么？缺失单元中应该填写的函数运算符是什么？举几个案例。
5. 查看本章和前一章中使用匿名函数代替局部函数应用的所有例子。使用 partial() 来替换匿名函数。你认为结果如何？更易于阅读吗？ 251

12.4　组合函数运算符

除了可以对单个函数进行操作外，函数运算符还可以接受多个函数作为输入。plyr::each() 就是其中的一个简单例子。它接收一个向量化函数的列表并将它们组合为一个函数。

```
summaries <- plyr::each(mean, sd, median)
summaries(1:10)
#>   mean     sd median
#>  5.500  3.028  5.500
```

两个更加复杂的例子分别是：通过复合来组合函数；通过布尔代数来组合函数。这些功能为我们提供了将多个函数连在一起的胶水。

12.4.1　函数复合

将函数组合的一个重要方式是通过：f(g(x))。复合接收函数列表，然后将这些函数依次应用于输入。它是常用匿名函数模式（将多个函数连接在一起来获得你想要的结果）的一种替代：

```
sapply(mtcars, function(x) length(unique(x)))
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#>   25    3   27   22   22   29   30    2    2    3    6
```

一个简单版的复合如下所示：

```
compose <- function(f, g) {
  function(...) f(g(...))
}
```

（pryr::compose() 提供了一个功能更加完全的替代品，它可以接收多个函数，剩下的例子就使用它。）

它可以写成： 252

```
sapply(mtcars, compose(length, unique))
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#>   25    3   27   22   22   29   30    2    2    3    6
```

在数学上，函数复合通常是使用中缀运算符 o（(f o g)(x)）表示。Haskell，一种流行的函数式编程语言，使用 . 来进行函数复合。在 R 中，可以创建一个自己的中缀函数：

```
"%o%" <- compose
sapply(mtcars, length %o% unique)
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#>   25    3   27   22   22   29   30    2    2    3    6

sqrt(1 + 8)
#> [1] 3
compose(sqrt, `+`)(1, 8)
#> [1] 3
(sqrt %o% `+`)(1, 8)
#> [1] 3
```

复合提供了一种非常简洁的方式来实现函数 Negate，它只是一个局部求值版的 compose()。

```
Negate <- partial(compose, `!`)
```

可以使用函数复合来计算总体标准差：

```
square <- function(x) x^2
deviation <- function(x) x - mean(x)

sd2 <- sqrt %o% mean %o% square %o% deviation
sd2(1:10)
#> [1] 2.872
```

这种类型的编程就称为隐式编程或者 point-free 编程。（这里的 point-free 来源于拓扑学中指代数值的“point”，这种编程也可以简称为“pointless”。）在这种编程风格中，不需要显式地引用变量。只需要将精力集中在高层的函数组合上，而不用处理的底层的流数据。主要关注做什么，而不是要对谁做。因为我们只使用函数并且没有参数，所以我们要使用动词而不使用名词。这种风格在 Haskell 中非常常见，它也是基于栈的编程语言（例如，Forth 和 Factor）的典型风格。虽然在 R 语言中这并不是一种自然或优雅的编程风格，但还是很有意思的。

compose() 与 partial() 一起使用非常有用，因为 partial() 可以让你给被复合的函数提供额外参数。这种编程风格的一个副作用就是使函数参数与函数名很近。这种副作用很重要，因为如果你必须记住非常长的代码块（例如，参数与函数名距离很远时），那么这样的代码也很难理解。

下面的代码是对本章第一节例子的修改，使用了上面描述的两种函数组合风格。它们都比原始代码长，但是现在的代码可能更容易理解，因为函数与它的参数距离很近。但是仍然需要从右往左（从下向上）阅读：第一个被调用的函数是最后一个编号的函数。可以对 compose() 进行定义，使它按照相反的方向处理，但是从长远角度考虑，这样做会使得一部分代码的阅读方式与其他代码不同，因此给阅读人员带来迷惑，所以还是不要这样做。

```
download <- dot_every(10, memoise(delay_by(1, download_file)))

download <- pryr::compose(
  partial(dot_every, 10),
  memoise,
  partial(delay_by, 1),
  download_file
)

download <- partial(dot_every, 10) %o%
  memoise %o%
```

```
partial(delay_by, 1) %o%
download_file
```

12.4.2　逻辑判断和布尔代数

当使用 `Filter()` 以及其他需要与逻辑判断一起工作的泛函时，我们经常发现需要使用匿名函数来组合多个条件：

253 ~ 254

```
Filter(function(x) is.character(x) || is.factor(x), iris)
```

另一个可选方案是，可以定义一些将逻辑判断组合在一起的函数运算符：

```
and <- function(f1, f2) {
  force(f1); force(f2)
  function(...) {
    f1(...) && f2(...)
  }
}

or <- function(f1, f2) {
  force(f1); force(f2)
  function(...) {
    f1(...) || f2(...)
  }
}

not <- function(f) {
  force(f)
  function(...) {
    !f(...)
  }
}
```

这样就可以写成：

```
Filter(or(is.character, is.factor), iris)
Filter(not(is.numeric), iris)
```

现在，我们可以对函数本身而不是函数的结果进行布尔代数了。

12.4.3　练习

1. 使用 `Reduce` 和 `%o%` 编写一个自己的 `compose()` 函数。为了获得奖励加分，在不调用 `function` 的情况下编写一个。
2. 对 `and()` 和 `or()` 进行扩展使它可以处理任意多个输入。使用 `Reduce()` 能实现吗？能使它们继续保持惰性求值的特性吗（例如，对于 `and()` 来说，一旦它发现第一个 `FALSE`，它就返回 `FALSE`）？ 255
3. 编写一个 `xor()` 二元运算符。使用现有的 `xor()` 函数来实现它。组合使用 `and()` 和 `or()` 来实现它。这两种方法的优缺点各是什么？同时思考你应该给这个函数取一个什么名字，使它不与现有的 `xor()` 函数发生冲突；如何修改 `and()`、`not()` 和 `or()` 的名字使它们保持一致。
4. 上面，我们已经实现了函数的布尔代数运算，它返回一个逻辑函数。实现初等代数运算（例如，`plus()`、`minus()`、`multiply()`、`divide()`、`exponentiate()`、`log()`），使它们返回数值向量。 256

第三部分

语言计算

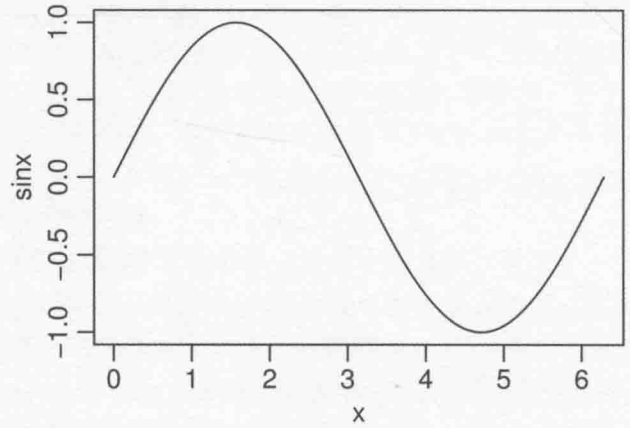
第 13 章
非标准计算

“对于交互式编程语言来说，在不导致语义模糊的情况下，我们应该追求更加灵活的语法。”

——Kent Pitman

R 不仅具有对数值进行计算的强大工具，而且还可以对得到这些值的整个计算过程的每一步进行计算。如果你以前使用其他编程语言，那么对于你来说这一点应该是最惊人的。考虑下面绘制正弦函数曲线的简单代码片段：

```
x <- seq(0, 2 * pi, length = 100)
sinx <- sin(x)
plot(x, sinx, type = "l")
```



257 ~ 259

注意坐标轴上的标签。R 如何知道 x 轴上的变量为 x，而 y 轴上的变量为 sinx 的呢？在大多数编程语言中，只能访问函数参数的值。而在 R 中，还可以访问用来对函数参数进行计算的代码。这就使得我们可以用非标准的方式对代码进行计算：称为**非标准计算**（NSE）。由于 NSE 可以显著地减少键盘输入量，所以它对于交互式数据分析非常有用。

主要内容

- 13.1 节学习使用 `substitute()` 来捕获还没有计算的表达式。
- 13.2 节说明如何组合使用 `subset()`、`substitute()` 和 `eval()` 来简洁地选取数据框中的某些行。

- 13.3 节讨论 NSE 中特有的作用域问题，并学习如何解决这些问题。
- 13.4 节介绍为什么所有使用 NSE 的函数都应该有一个应急方案，一个使用常规计算方法的函数。
- 13.5 节学习使用 `substitute()` 来处理没有应急方案的函数。
- 13.6 节讨论 NSE 的缺点。

预备条件

在阅读本章之前，确保你已经熟悉环境（见第 8 章）和词法作用域（见 6.2 节）。还应该首先使用 `install.packages("pryr")` 命令安装 `pryr` 添加包。有些练习还需要使用 `plyr` 添加包，可以使用 `install.packages("plyr")` 命令从 CRAN 来安装它。

13.1 表达式获取

`substitute()` 使非标准计算成为可能。它查找函数参数并且不看函数参数的值，它看用来计算参数值的代码：

260

```
f <- function(x) {
  substitute(x)
}
f(1:10)
#> 1:10

x <- 10
f(x)
#> x

y <- 13
f(x + y^2)
#> x + y^2
```

现在，我们还不需要准确地知道 `substitute()` 返回的是什么（这是第 14 章的主题），但我们先把这个返回值称为表达式。

`substitute()` 能够工作，因为该函数的参数使用了一个称为**约定**（promise）的特殊类型的对象。约定捕获用来计算的表达式和执行该表达式的环境。我们没有感受到约定的存在，因为在我们第一次访问它时，它的代码就在相应的环境中执行，并产生一个值。

`substitute()` 经常与 `deparse()` 一起使用。`deparse()` 函数以 `substitute()` 的结果（一个表达式）为参数，并把它转变成一个字符向量。

```
g <- function(x) deparse(substitute(x))
g(1:10)
#> [1] "1:10"
g(x)
#> [1] "x"
g(x + y^2)
#> [1] "x + y^2"
```

在基础 R 包中有很多函数使用了这种想法。有些使用它们来避免使用引号：

```
library(ggplot2)
# the same as
library("ggplot2")
```

其他函数，如 `plot.default()`，使用它们来提供默认的标签。`data.frame()` 应用对它们进行计算的相应表达式来标记该变量：

```
x <- 1:4
y <- letters[1:4]
names(data.frame(x, y))
#> [1] "x" "y"
```

通过对学习一个 NSE 非常有用的应用（`subset()`），可以理解所有这些例子背后的思想。

练习

1. 在编程时需要注意 `deparse()` 的一个非常重要的特点：如果输入太长，它可能返回多个字符串。例如，下面的调用产生一个长度为 2 的向量。

```
g(a + b + c + d + e + f + g + h + i + j + k + l + m +
  n + o + p + q + r + s + t + u + v + w + x + y + z)
```

   为什么会这样呢？仔细阅读文档。能写一个 `deparse()` 的包装函数，使它每次都只能返回一个字符串吗？

2. 为什么 `as.Date.default()` 使用 substitute() 和 deparse()？为什么 pairwise.t.test() 也使用它们？请阅读源代码。

3. `pairwise.t.test()` 假设 `deparse()` 总是返回一个长度为 1 的字符向量。你能构建一个违反这个假设的输入吗？会发生什么？

4. 上面定义的 `f()`，只调用 `substitute()`。为什么我们不能使用它来定义 `g()` 呢？换句话说，下面代码返回什么呢？先预测一下，然后运行代码来看看你的预测是否正确。

```
f <- function(x) substitute(x)
g <- function(x) deparse(f(x))
g(1:10)
g(x)
g(x + y ^ 2 / z + exp(a * sin(b)))
```

261 ~ 262

13.2 在子集中进行非标准计算

虽然将提供参数值的代码打印出来可能很有用，但其实我们可以对未计算的代码做更多事情。以 `subset()` 为例。它是数据框子集选取的一个非常有用的交互式快捷方式：不需要多次重复输入数据框的名字，所以可以减少一些键盘输入。

```
sample_df <- data.frame(a = 1:5, b = 5:1, c = c(5, 3, 1, 4, 1))

subset(sample_df, a >= 4)
#>   a b c
#> 4 4 2 4
#> 5 5 1 1
# equivalent to:
# sample_df[sample_df$a >= 4, ]

subset(sample_df, b == c)
#>   a b c
#> 1 1 5 5
#> 5 5 1 1
# equivalent to:
# sample_df[sample_df$b == sample_df$c, ]
```

`subset()` 很特别，因为它采用不同的作用域规则：表达式 `a >= 4` 或 `b == c` 是在指定数据框中执行，而不是当前或全局环境中。这就是非标准计算的本质。

`subset()` 如何工作呢？我们已经知道如何捕获参数表达式而不是它的结果，所以只需要知道如何在正确的上下文中对表达式进行计算。具体地，我们希望 `x` 能解释成 `sample_df$x` 而不是 `globalenv()$x`。为此，我们需要 `eval()`。这个函数可以在指定环境中接收表达式并对表达式进行计算。

在对 `eval()` 进行探索之前，我们还需要另一个有用的函数：`quote()`。它可以像 `substitute()` 那样捕获表达式，但不会做更进一步的转换。`quote()` 总是以下面的形式返回它的输入：

263

```
quote(1:10)
#> 1:10
quote(x)
#> x
quote(x + y^2)
#> x + y^2
```

我们需要用 `quote()` 对 `eval()` 做一些试验，因为 `eval()` 的第一个参数是表达式。所以，如果只提供了一个参数，它就在当前环境中对表达式进行求值。这样无论 `x` 是什么，`eval(quote(x))` 都会完全等价于 `x`。

```
eval(quote(x <- 1))
eval(quote(x))
#> [1] 1

eval(quote(y))
#> Error: object 'y' not found
```

`quote()` 和 `eval()` 是对立的。在下面的例子中，每个 `eval()` 剥去一层 `quote()`。

```
quote(2 + 2)
#> 2 + 2
eval(quote(2 + 2))
#> [1] 4

quote(quote(2 + 2))
#> quote(2 + 2)
eval(quote(quote(2 + 2)))
#> 2 + 2
eval(eval(quote(quote(2 + 2))))
#> [1] 4
```

`eval()` 的第二个参数设置执行代码的环境：

```
x <- 10
eval(quote(x))
#> [1] 10
e <- new.env()
e$x <- 20
eval(quote(x), e)
#> [1] 20
```

因为列表和数据框将名字与数值进行绑定的方式与环境绑定的方式类似，所以 `eval()` 的第二个参数也不需要限于环境：它还可以是列表或数据框。

```
eval(quote(x), list(x = 30))
#> [1] 30
eval(quote(x), data.frame(x = 40))
#> [1] 40
```

这将给出 subset() 的部分功能：

```
eval(quote(a >= 4), sample_df)
#> [1] FALSE FALSE FALSE  TRUE  TRUE
eval(quote(b == c), sample_df)
#> [1]  TRUE FALSE FALSE FALSE  TRUE
```

使用 `eval()` 时一个常见错误是，忘记对第一个参数进行引用。对比下面的结果：

```
a <- 10
eval(quote(a), sample_df)
#> [1] 1 2 3 4 5
eval(a, sample_df)
#> [1] 10

eval(quote(b), sample_df)
#> [1] 5 4 3 2 1
eval(b, sample_df)
#> Error: object 'b' not found
```

可以使用 `eval()` 和 `substitute()` 来编写 `subset()`。首先捕获代表条件的调用，然后在数据框的上下文中执行它，最后使用这个结果进行子集选取：

```
subset2 <- function(x, condition) {
  condition_call <- substitute(condition)
  r <- eval(condition_call, x)
  x[r, ]
}
subset2(sample_df, a >= 4)
#>   a b c
#> 4 4 2 4
#> 5 5 1 1
```

练习

1. 预测下面几行代码的结果：

```
eval(quote(eval(quote(eval(quote(2 + 2))))))
eval(eval(quote(eval(quote(eval(quote(2 + 2)))))))
quote(eval(quote(eval(quote(eval(quote(2 + 2)))))))
```

2. 如果用一个仅有一列的数据框使用 `subset()`，它有一个漏洞。下面的代码应该返回什么？如何对 `subset()` 进行修改使它返回正确类型的结果？

```
sample_df2 <- data.frame(x = 1:10)
subset2(sample_df2, x > 8)
#> [1]  9 10
```

3. 真实的子集选取函数（`subset.data.frame()`）在条件中删除缺失值。对 `subset2()` 进行修改，使它也能实现同样的功能：去除不符合要求的行。
4. 如果在 `subset2()` 中使用 `quote()` 而不使用 `substitute()`，会发生什么？
5. `subset()` 中的第二个参数允许我们选择变量。它把变量名字作为位置看待。它允许我们像 `subset(mtcars, , -cyl)` 一样去掉 `cylinder` 变量；或者像 `subset(mtcars, , disp:drat)`

一样选择 disp 和 drat 之间的所有变量。它是如何工作的呢？为了便于理解，我已经将执行这个功能的代码提取出来。

```
select <- function(df, vars) {
  vars <- substitute(vars)
  var_pos <- setNames(as.list(seq_along(df)), names(df))
  pos <- eval(vars, var_pos)
  df[, pos, drop = FALSE]
}
select(mtcars, -cyl)
```

264 ～ 266

6. evalq() 的功能是什么？使用它来减少上面应用 eval() 和 quote() 的例子中的键盘输入量。

13.3 作用域问题

很明显 subset2() 能够工作。但是，由于我们处理表达式而不是数值，所以需要进行大量的测试。例如，下面对 subset2() 的应用应该返回同样的值，因为它们之间的唯一不同是变量名：

```
y <- 4
x <- 4
condition <- 4
condition_call <- 4

subset2(sample_df, a == 4)
#>   a b c
#> 4 4 2 4
subset2(sample_df, a == y)
#>   a b c
#> 4 4 2 4
subset2(sample_df, a == x)
#>    a  b  c
#> 1  1  5  5
#> 2  2  4  3
#> 3  3  3  1
#> 4  4  2  4
#> 5  5  1  1
#> NA NA NA NA
#> NA.1 NA NA NA
subset2(sample_df, a == condition)
#> Error: object 'a' not found
subset2(sample_df, a == condition_call)
#> Warning: longer object length is not a multiple of shorter
#> object length
#> [1] a b c
#> <0 rows> (or 0-length row.names)
```

哪里出现了问题？从我选择的变量名中你应该可以得到一点提示：它们都是 subset2() 函数内部定义的变量名。如果 eval() 在数据框（它的第二个参数）中找不到变量，它就到 subset2() 的环境中查找。显然这不是我们想要的，所以我们需要告诉 eval() 如果它在数据框中找不到这个变量，它应该到哪里去找。

关键是 eval() 的第三个参数：enclos。通过它可以为没有父（或者封闭）环境的对象（例如，列表和数据框）设置一个父参数（或者封闭）环境。如果在 env 参数中找不到相应

的绑定，eval() 就会在 enclos 参数中查找，然后在 enclos 参数的父环境中查找。如果在 env 参数中能够找到，enclos 就会被忽略。我们希望在调用 subset2() 的环境中查找 x。在 R 术语中，这称为**父对象框**（parent frame），可以通过 parent.frame() 来获取。这是动态作用域（http://en.wikipedia.org/wiki/Scope_%28programming%29#Dynamic_scoping）的例子：该值来源于函数被调用的位置而不是它被定义的地方。

经过这些修改，我们的函数现在可以工作了：

```
subset2 <- function(x, condition) {
  condition_call <- substitute(condition)
  r <- eval(condition_call, x, parent.frame())
  x[r, ]
}

x <- 4
subset2(sample_df, a == x)
#>   a b c
#> 4 4 2 4
```

使用 enclos 只是将列表或数据框转换成环境的捷径。使用 list2env() 可以实现同样的目的。它可以将列表转换成具有显式父环境的环境：

```
subset2a <- function(x, condition) {
  condition_call <- substitute(condition)
  env <- list2env(x, parent = parent.frame())
  r <- eval(condition_call, env)
  x[r, ]
}

x <- 5
subset2a(sample_df, a == x)
#>   a b c
#> 5 5 1 1
```

练习

1. plyr::arrange() 与 subset() 的工作方式类似，但它不是对行进行选择，它记录它们。它是如何工作的呢？ substitute(order(...)) 的功能是什么？创建一个只做这件事的函数并对它进行测试。
2. 阅读 transform() 的文档，该函数的功能是什么？它是如何运行的？阅读 transform.data.frame() 的源代码。substitute(list(...)) 的功能是什么？
3. plyr::mutate() 与 transform() 类似，但它按顺序进行变换，所以它可以根据刚刚创建的列进行变换：

```
df <- data.frame(x = 1:5)
transform(df, x2 = x * x, x3 = x2 * x)
plyr::mutate(df, x2 = x * x, x3 = x2 * x)
```

mutate 是如何工作的呢？ mutate() 和 transform() 的关键不同点是什么？

4. with() 的功能是什么？它是如何工作的？阅读 with.default() 的源代码。within() 的功能是什么？它是如何工作的？阅读 within.data.frame() 的源代码。为什么它的代码比 with() 的代码复杂？

13.4 从其他函数调用

通常，对语言进行计算的另一个特点是：这些函数在被用户直接调用时非常有用，但它们被其他函数调用时就没有太大用处。虽然 `subset()` 可以减少键盘的输入量，但是在非交互式的编程中它非常难用。例如，假如我们要创建一个函数，该函数对数据的某些行构成的子集进行随机排序。一个好的实现方式就是创建一个由重新排序函数和子集选取函数复合而成的函数。让我们试试吧：

267 ~ 269

```
subset2 <- function(x, condition) {
  condition_call <- substitute(condition)
  r <- eval(condition_call, x, parent.frame())
  x[r, ]
}

scramble <- function(x) x[sample(nrow(x)), ]

subscramble <- function(x, condition) {
  scramble(subset2(x, condition))
}
```

但它不能工作：

```
subscramble(sample_df, a >= 4)
# Error in eval(expr, envir, enclos) : object 'a' not found
traceback()
#> 5: eval(expr, envir, enclos)
#> 4: eval(condition_call, x, parent.frame()) at #3
#> 3: subset2(x, condition) at #1
#> 2: scramble(subset2(x, condition)) at #2
#> 1: subscramble(sample_df, a >= 4)
```

哪里出错了？为了找到问题，我们来逐行对其进行调试：

```
debugonce(subset2)
subscramble(sample_df, a >= 4)
#> debugging in: subset2(x, condition)
#> debug at #1: {
#>     condition_call <- substitute(condition)
#>     r <- eval(condition_call, x, parent.frame())
#>     x[r, ]
#> }
n
#> debug at #2: condition_call <- substitute(condition)
n
#> debug at #3: r <- eval(condition_call, x, parent.frame())
r <- eval(condition_call, x, parent.frame())
#> Error in eval(expr, envir, enclos) : object 'a' not found
condition_call
#> condition
eval(condition_call, x)
#> Error in eval(expr, envir, enclos) : object 'a' not found
Q
```

找到问题了吗？`condition_call` 包含表达式 `condition`。因此，当我们对 `condition_call` 进行计算时，也会对 `condition`（`a >= 4`）进行计算。但是这个计算是不能正常完成的，因为在父环境中没有称为 `a` 的对象。但是，如果在全局环境中存在对象 `a`，那么得到的

结果就会更让人迷惑：

```
a <- 4
subscramble(sample_df, a == 4)
#>   a b c
#> 4 4 2 4
#> 3 3 3 1
#> 1 1 5 5
#> 5 5 1 1
#> 2 2 4 3

a <- c(1, 1, 4, 4, 4, 4)
subscramble(sample_df, a >= 4)
#>      a  b  c
#> 5    5  1  1
#> NA NA NA NA
#> 4    4  2  4
#> 3    3  3  1
```

这个例子说明了为交互式使用而设计的函数和为安全编程而设计的函数的区别。使用 `substitute()` 的函数可能减少键盘的输入量，但是从其他函数调用这个函数就非常困难。

作为一个开发者，总是应该为用户提供一个应急方案：提供一个使用标准计算的备用函数。就本例而言，可以编写 `subset2()` 的另一个版本，该版本的函数使用一个已经引用的表达式。

```
subset2_q <- function(x, condition) {
  r <- eval(condition, x, parent.frame())
  x[r, ]
}
```

这里使用了后缀 `_q` 来表示它可以接收一个引用的表达式。大多数用户不需要使用到这个函数，所以它的名字可以稍微长一点儿。

可以使用 `subset2_q()` 来重写 `subset2()` 和 `subscramble()`：

```
subset2 <- function(x, condition) {
  subset2_q(x, substitute(condition))
}

subscramble <- function(x, condition) {
  condition <- substitute(condition)
  scramble(subset2_q(x, condition))
}

subscramble(sample_df, a >= 3)
#>   a b c
#> 5 5 1 1
#> 4 4 2 4
#> 3 3 3 1
subscramble(sample_df, a >= 3)
#>   a b c
#> 3 3 3 1
#> 5 5 1 1
#> 4 4 2 4
```

R 基础包中的函数使用不同的应急方案。它们经常包含一个可以将 NSE 关闭的参数。例如，`require()` 有一参数 `character.only = TRUE`。我觉得使用一个参数来改变另一

个参数的行为并不是一个好主意，因为这导致函数调用很难理解。

练习

1. 下面的函数都使用到 NSE。说说它们都是如何使用 NSE 的，阅读它们的文档，确定它们的应急方案是什么？

270 ≀ 272

   - `rm()`
   - `library()` and `require()`
   - `substitute()`
   - `data()`
   - `data.frame()`

2. 基础函数 `match.fun()`、`page()` 和 `ls()` 都尝试自动选择是否使用非标准计算。每个都使用不同的方法。找出这些方法的本质，然后对它们进行比较。

3. 通过将 `plyr::mutate()` 拆分成两个函数来为其增加一个应急方案。一个函数应该捕获未计算的输入。另外一个函数应该接收数据框和表达式列表并进行计算。

4. `ggplot2::aes()` 的应急方案是什么？`plyr::()` 呢？它们有哪些共同点？它们的不同点有哪些优缺点？

5. 上面演示的 `subset2_q()` 是对真实代码的简化。为什么下面这个版本更好？

```
subset2_q <- function(x, cond, env = parent.frame()) {
  r <- eval(cond, x, env)
  x[r, ]
}
```

   使用这个改进版的函数来重写 `subset2()` 和 `subscramble()`。

13.5 替换

大多数使用非标准计算的函数会提供应急方案。但是，如果我们想调用一个没有应急方案的函数，又该怎么办呢？例如，我们想根据两个变量的名字来创建一个晶格图（lattice graphic）：

```
library(lattice)
xyplot(mpg ~ disp, data = mtcars)

x <- quote(mpg)
y <- quote(disp)
xyplot(x ~ y, data = mtcars)
#> Error: object of type 'symbol' is not subsettable
```

我们可能要求助于 `substitute()`，以其他目的来使用它：对表达式进行修改。不幸的是，`substitute()` 有一个修改调用使它不适合交互式的特点。在它全局环境中运行时，它从不进行替换：实际上，在这种情况下，它与 `quote()` 的行为相同。

```
a <- 1
b <- 2
substitute(a + b + z)
#> a + b + z
```

但是，如果是在函数内部运行它，`substitute()` 就进行替换，并将不能替换的保留下来：

```
f <- function() {
  a <- 1
  b <- 2
  substitute(a + b + z)
}
f()
#> 1 + 2 + z
```

为了能够更方便地使用 substitute()，pryr 添加包提供了 subs() 函数。它的工作方式与 substitute() 相同，但是它的名字更短且在全局环境中进行计算。这两个特点使我们可以很容易地使用它进行实验：

```
a <- 1
b <- 2
subs(a + b + z)
#> 1 + 2 + z
```

函数 subs() 和 substitute() 的第二个参数可以重写正在使用的当前环境，并通过一个名字 – 值对的列表来提供一个替代品。下面的例子使用这个技术来说明对字符串、变量名或者函数调用进行替换的不同方法。

273 ~ 274

```
subs(a + b, list(a = "y"))
#> "y" + b
subs(a + b, list(a = quote(y)))
#> y + b
subs(a + b, list(a = quote(y())))
#> y() + b
```

记住，在 R 语言中所有动作都是函数调用，所以也可以使用其他函数来替换 +：

```
subs(a + b, list("+" = quote(f)))
#> f(a, b)
subs(a + b, list("+" = quote(`*`)))
#> a * b
```

可以写出完全没有意义的代码：

```
subs(y <- y + 1, list(y = 1))
#> 1 <- 1 + 1
```

形式上，通过对表达式中的所有名字进行检查来进行替换。如果名字引用：

1）一个普通变量，它就被变量的值替换。

2）一个约定（一个函数参数），它就被与约定相关联的表达式替换。

3）...，它被 ... 的内容替换。

否则，名字就保留原样不变。

可以使用这些知识来创建一个正确的 xyplot() 调用：

```
x <- quote(mpg)
y <- quote(disp)
subs(xyplot(x ~ y, data = mtcars))
#> xyplot(mpg ~ disp, data = mtcars)
```

275 在函数中它更简单，因为不再需要显式地引用 x 和 y 的变量（上面的第 2 条规则）：

```
xyplot2 <- function(x, y, data = data) {
  substitute(xyplot(x ~ y, data = data))
}
```

```
xyplot2(mpg, disp, data = mtcars)
#> xyplot(mpg ~ disp, data = mtcars)
```

如果在调用替换中还包含 ...，就可以为调用添加更多的参数：

```
xyplot3 <- function(x, y, ...) {
  substitute(xyplot(x ~ y, ...))
}
xyplot3(mpg, disp, data = mtcars, col = "red", aspect = "xy")
#> xyplot(mpg ~ disp, data = mtcars, col = "red", aspect = "xy")
```

为了创建图形，使用 eval() 来执行调用。

13.5.1 为替换提供应急方案

substitute() 本身就是使用 NSE 的函数并且它没有应急方案。这就意味着如果有一个保存在变量中的表达式，那么就不能使用 substitute()。

```
x <- quote(a + b)
substitute(x, list(a = 1, b = 2))
#> x
```

虽然 substitute() 没有内置应急方案，但是可以利用函数本身来创建一个：

```
substitute_q <- function(x, env) {
  call <- substitute(substitute(y, env), list(y = x))
  eval(call)
}

x <- quote(a + b)
substitute_q(x, list(a = 1, b = 2))
#> 1 + 2
```

substitute_q() 的实现简洁而深入。我们再来看看上面的例子：substitute_q(x, list(a = 1, b = 2))。这里有一点小技巧，由于 substitute() 使用 NSE，所以就不能由内到外执行括号中的代码。

1）首先执行 substitute(substitute(y, env), list(y = x))。捕获表达式 substitute(y, env)，使用 x 来替换 y。因为已经将 x 放入列表，所以它将被求值并且根据替换规则将用它的值替换 y。这样就产生了表达式 substitute(a + b, env)。

2）接下来，计算当前函数中的表达式。substitute() 计算它的第一个参数并在 env 中查找名字 - 值对。这里，它计算 list(a = 1, b = 2)。因为这些都是值（而不是约定），所以结果是 1 + 2。

pryr 添加包中还有一个更严谨的 substitute_q() 函数。

13.5.2 捕获未计算的表达式 ...

另一个有用的技术就是捕获 ... 中所有未计算的表达式。基础 R 函数有多种方法可以完成这种任务，但是有一个技术适合于多种情况：

```
dots <- function(...) {
  eval(substitute(alist(...)))
}
```

这里使用 alist() 函数，它可以简单地捕获所有参数。这个函数与 pryr::dots() 相同。

pryr 也提供 pryr::named_dots()，它使用解析过的表达式作为默认名，保证所有的参数都是命名的（就像 data.frame() 一样）。

13.5.3 练习

1. 对于下面的每对表达式，使用 subs() 将 LHS 转换成 RHS。

   - a + b + c ->a * b * c
   - f(g(a, b), c) ->(a + b) * c
   - f(a < b, c, d) ->if (a < b) c else d

276 ~ 277

2. 对于下面的每对表达式，描述为什么不能使用 subs() 对其进行转换。

   - a + b + c ->a + b * c
   - f(a, b) ->f(a, b, c)
   - f(a, b, c) ->f(a, b)

3. pryr::named_dots() 是如何工作的？阅读源代码进行说明。

13.6 非标准计算的缺点

NSE 的最大缺点是，使用 NSE 的函数就不再是引用透明的（http://en.wikipedia.org/wiki/Referential_transparency_(computer_science)）。一个函数是**引用透明**的，如果可以用它们的值替换相应的参数且函数的行为不发生改变。例如，如果函数 f() 是引用透明的，而且 x 和 y 都等于 10，那么 f(x)、f(y) 和 f(10) 都返回相同的结果。引用透明的代码很容易探究，因为对象的名字是不重要的，并且总可以从最内部的括号向外进行计算。

还有很多重要的函数，它们本质上就不是引用透明的。它们接受参数运算符。你不能使用 a <- 1，不能用它的值代替 a，不能得到相同的结果。这就是人们通常在函数的顶层进行赋值的一个原因。如下的代码很难探究：

```
a <- 1
b <- 2
if ((b <- a + 1) > (a <- b - 1)) {
  b <- b + 2
}
```

使用 NSE 抑制了函数的引用透明性。这使得需要的心智模型（mental model）正确预测输出变得非常复杂。所以，如果对程序有极大的提升才值得使用 NSE。例如，调用 library() 和 require() 时，可以使用也可以不使用引号，因为在内部它们使用 deparse(substitute(x)) 以及一些其他技巧。这意味着下面这两行代码做相同的事情：

278

```
library(ggplot2)
library("ggplot2")
```

如果变量与值相关联，那么事情开始变得复杂。下面的代码将加载哪个软件包呢？

```
ggplot2 <- "plyr"
library(ggplot2)
```

还有一些其他的 R 函数也以这种方式工作，如 ls()、rm()、data()、demo()、example() 和 vignette()。对我来说，为了减少两次键盘输入而丢失引用透明性是不值得的，如果是出于这个目的（减少两次键盘输入），我也不推荐你使用 NSE。

非常适合使用非标准计算的一种情况是 `data.frame()`。如果没有显式地提供，它使用输入自动地对输出变量进行命名：

```
x <- 10
y <- "a"
df <- data.frame(x, y)
names(df)
#> [1] "x" "y"
```

我认为这是值得的，因为当我们从已有变量创建数据框时，它可以减少常见情况中的许多冗余。更重要的是，如果需要，还可以方便地通过为每个变量提供名字来对其进行重写。

非标准计算可以让我们写出功能非常强大的函数。但是，它们阅读起来非常困难而且很难应用于程序中。同时还要提供应急方案，所以在一个新领域中使用 NSE 前，最好仔细地思考 NSE 的成本与收益。

练习

1. 下面这个函数的功能是什么？它的应急方案是什么？你觉得这是 NSE 的一个合理应用吗？

```
nl <- function(...) {
  dots <- named_dots(...)
  lapply(dots, eval, parent.frame())
}
```

2. 不依赖于约定，你还可以使用 ~ 创建的公式来显式地捕获表达式和它的环境。显式引用的优缺点是什么？它如何影响引用透明性？

3. 在（http://developer.r-project.org/nonstandard-eval.pdf）上阅读非标准计算的标准规则。

279
〜
280

第 14 章
表 达 式

在第 13 章中，我们已经学习了 R 语言计算的底层获取和计算表达式的基础知识。本章将学习如何使用代码来对这些表达式进行操作。我们将学习如何进行元编程：如何使用其他程序来创建程序！

主要内容

- 14.1 节深入学习表达式的结构。我们将学习表达式的 4 个元素：常量、名字、调用和成对列表。
- 14.2 节更详细地学习名字。
- 14.3 节更详细地学习调用。
- 14.4 节稍微偏离主题讨论 R 基础包中调用的常见用途。
- 14.5 节完成对表达式的 4 个主要元素的讨论，并学习如何根据这些元素来创建函数。
- 14.6 节讨论表达式和文本之间的相互转换。
- 14.7 节对本章进行总结，结合我们学习的编写函数的知识，对任意的 R 代码进行计算和修改。

预备条件

在本章中，我们使用 pryr 添加包提供的工具。如果还没有安装它，可以使用 `install.packages("pryr")` 命令来安装它。

281

14.1 表达式的结构

为了对语言进行计算，首先需要理解语言的结构。这需要学习一些新的词汇、一些新的工具和对 R 代码进行思考的新方法。需要理解的第一件事就是运算和结果之间的区别：

```
x <- 4
y <- x * 10
y
#> [1] 40
```

我们要将 x 乘以 10 并将它赋值给 y 的动作与实际结果（40）区分。与前一章看到的一样，我们能够使用 `quote()` 来捕获这个动作：

```
z <- quote(y <- x * 10)
z
#> y <- x * 10
```

`quote()` 返回一个**表达式**（expression）：它是一个对象，它表示一个可以被 R 执行的动作。（不幸的是，`expression()` 不能返回这种表达式。相反，它返回更像表达式列表的某些东西。详细信息参见 14.6 节。）

因为表达式表示代码的分层树状结构，所以它也称为抽象语法树（AST）。使用 `pryr::ast()` 可以更清晰地了解它：

```
ast(y <- x * 10)
#> \- ()
#>   \- `<-
#>   \- `y
#>   \- ()
#>     \- `*
#>     \- x
#>     \- 10
```

282

表达式有 4 个可能的元素：常量、名字、调用和成对列表。

- **常量**（constant）是长度为 1 的原子向量，如 `"a"` 或 `10`。使用 `ast()` 来显示它们：

```
ast("a")
#> \- "a"
ast(1)
#> \- 1
ast(1L)
#> \- 1L
ast(TRUE)
#> \- TRUE
```

  引用常量不会改变它：

```
identical(1, quote(1))
#> [1] TRUE
identical("test", quote("test"))
#> [1] TRUE
```

- **名字**（name），或者符号，表示对象的名字而不是它的值。`ast()` 使用反引号前缀来表示名字。

```
ast(x)
#> \- `x
ast(mean)
#> \- `mean
ast(`an unusual name`)
#> \- `an unusual name
```

- **调用**（call）表示调用一个函数的动作。与列表类似，调用是递归的：它们可以包含常量、名字、成对列表和其他调用。`ast()` 输出 ()，然后是将子元素列出。第一个子元素就是被调用的函数，剩下的子元素就是函数的参数。

```
ast(f())
#> \- ()
#>   \- `f
ast(f(1, 2))
#> \- ()
#>   \- `f
#>   \- 1
#>   \- 2
ast(f(a, b))
#> \- ()
```

```
#>   \- `f
#>     \- `a
#>     \- `b
ast(f(g(), h(1, a)))
#> \- ()
#>   \- `f
#>   \- ()
#>     \- `g
#>   \- ()
#>     \- `h
#>     \- 1
#>     \- `a
```

与 6.3 节中提到的，即使看上去不像函数调用的运算也具有这样的分层结构。

```
ast(a + b)
#> \- ()
#>   \- `+
#>   \- `a
#>   \- `b
ast(if (x > 1) x else 1/x)
#> \- ()
#>   \- `if
#>   \- ()
#>     \- `>
#>     \- `x
#>     \- 1
#>   \- `x
#>   \- ()
#>     \- `/
#>     \- 1
#>     \- `x
```

- 成对列表（pairlist），是点对列表（Dotted Pair Lists）的简写，是 R 过去的“遗产”。它们只在一个地方使用：函数的正式参数。`ast()` 在成对列表的顶层输出 `[]`。与调用类似，成对列表也是递归并且可以包含常量、名字和调用。

```
ast(function(x = 1, y) x)
#> \- ()
#>   \- `function
#>   \- []
#>     \ x = 1
#>     \ y =`MISSING
#>   \- `x
#>   \- <srcref>
ast(function(x = 1, y = x * 2) {x / y})
#> \- ()
#>   \- `function
#>   \- []
#>     \ x = 1
#>     \ y =()
#>       \- `*
#>       \- `x
#>       \- 2
#>   \- ()
#>     \- `{
#>     \- ()
#>       \- `/
```

```
#>      \- `x
#>         \- `y
#>    \- <srcref>
```

注意，str() 不遵守对象的这些命名惯例。它将名称描述为符号，将调用描述为语言对象：

```
str(quote(a))
#>  symbol a
str(quote(a + b))
#>  language a + b
```

使用底层函数可以创建包含对象而不包含常量、名字、调用和成对列表的调用树。接下来的例子使用 substitute() 将数据框插入调用树。但是，这并不是一个好主意，因为对象不能正确输出：输出的调用看起来应该返回“列表”，但是当求值时，它返回“数据框”。

```
class_df <- substitute(class(df), list(df = data.frame(x = 10)))
class_df
#> class(list(x = 10))
eval(class_df)
#> [1] "data.frame"
```

283 ~ 285

所有 R 代码的结构都可以使用这 4 个元素来定义。在下面各节中我们将详细介绍它们。

练习

1. 没有现有的基础函数可以检查一个元素是否是一个表达式的有效元素（即，判断元素是否为常量、名字、调用或者成对列表）。通过实现相应名称的“is”函数来猜测相应元素为调用、名字和成对列表。
2. pryr::ast() 使用非标准计算。它对于标准计算的应急方案是什么？
3. 带有多个 else 条件的 if 语句的调用树看起来像什么？
4. 将 ast(x + y %+% z) 与 ast(x ^ y %+% z) 进行对比。常用的中缀函数的优先级是什么样的？
5. 为什么表达式不能包含一个长度大于 1 的原子向量？6 种类型的原子向量的哪种不能出现在表达式中？为什么？

14.2 名字

通常，使用 quote() 来捕获名字。还可以使用 as.name() 将字符串转换成名字。但是，这只有当函数接收字符串作为输入时它才最有用。否则就比 quote() 包含更多的键盘输入。（可以使用 is.name() 来检测对象是否是名字。）

```
as.name("name")
#> name
identical(quote(name), as.name("name"))
#> [1] TRUE

is.name("name")
#> [1] FALSE
is.name(quote(name))
#> [1] TRUE
is.name(quote(f(name)))
#> [1] FALSE
```

（名字也称为符号。as.symbol()和is.symbol()与as.name()和is.name()是完
286 全相同的。）

无效的名字会自动加上反引号：

```
as.name("a b")
#> `a b`
as.name("if")
#> `if`
```

还有一个特别的名字需要深入讨论：空名字。它经常用来表示缺失值。这个对象的行为有点儿奇怪。不能将它绑定到一个变量。如果这样做，它就触发关于缺失值的错误。如果我们想以编程方式创建一个带有缺失参数的函数，那么它才会有用。

```
f <- function(x) 10
formals(f)$x
is.name(formals(f)$x)
#> [1] TRUE
as.character(formals(f)$x)
#> [1] ""

missing_arg <- formals(f)$x
# Doesn't work!
is.name(missing_arg)
#> Error: argument "missing_arg" is missing, with no default
```

为了当需要时显式地创建它，使用命名的参数调用quote()：

```
quote(expr =)
```

练习

1. 使用formals()既可以获取也可以设置函数的参数。使用formals()来修改下面的函数，使x的默认值为缺失，y的默认值为10。

```
g <- function(x = 20, y) {
  x + y
287 }
```

2. 使用as.name()和eval()编写一个与get()等价的函数。使用as.name()、substitute()和eval()编写一个与assign()等价的函数。（这里不要考虑选择环境的多种方式，假定用户显式地设置环境。）

14.3 调用

调用与列表非常相似。它的方法有length、[[和[，由于调用可以包含其他调用，所以它也是递归的。调用的第一个元素是被调用的函数。通常它是函数的名字（name）：

```
x <- quote(read.csv("important.csv", row.names = FALSE))
x[[1]]
#> read.csv
is.name(x[[1]])
#> [1] TRUE
```

但它也可以是其他调用：

```
y <- quote(add(10)(20))
```

```
y[[1]]
#> add(10)
is.call(y[[1]])
#> [1] TRUE
```

剩下的元素为参数。可以通过名字和位置提取它们。

```
x <- quote(read.csv("important.csv", row.names = FALSE))
x[[2]]
#> [1] "important.csv"
x$row.names
#> [1] FALSE
names(x)
#> [1] ""          ""              "row.names"
```

调用的长度减去 1 就是参数的个数：

```
length(x) - 1
#> [1] 2
```

14.3.1 修改调用

使用标准的替换运算符 `$<-` 和 `[[<-` 可以增加、修改和删除调用的元素。

```
y <- quote(read.csv("important.csv", row.names = FALSE))
y$row.names <- TRUE
y$col.names <- FALSE
y
#> read.csv("important.csv", row.names = TRUE, col.names = FALSE)

y[[2]] <- quote(paste0(filename, ".csv"))
y[[4]] <- NULL
y
#> read.csv(paste0(filename, ".csv"), row.names = TRUE)

y$sep <- ","
y
#> read.csv(paste0(filename, ".csv"), row.names = TRUE, sep = ",")
```

调用也支持 `[` 方法。但使用它时要小心。删除第一个元素与创建一个有用的调用不同。

```
x[-3] # remove the second argument
#> read.csv("important.csv")
x[-1] # remove the function name - but it's still a call!
#> "important.csv"(row.names = FALSE)
x
#> read.csv("important.csv", row.names = FALSE)
```

如果需要一个未计算的参数（表达式）的列表，可以使用显式强制转换：

```
# A list of the unevaluated arguments
as.list(x[-1])
#> [[1]]
#> [1] "important.csv"
#>
#> $row.names
#> [1] FALSE
```

通常来讲，由于 R 的函数调用语义非常灵活，所以通过位置来获取或设置参数比较危险。例如，即使每个位置上的值不同，但下面这 3 个调用的效果却是相同的：

```
m1 <- quote(read.delim("data.txt", sep = "|"))
m2 <- quote(read.delim(s = "|", "data.txt"))
m3 <- quote(read.delim(file = "data.txt", , "|"))
```

为了解决这个问题，pryr 软件包提供了 standardise_call()。它使用基础的 match.call() 函数将所有位置参数转换成命名参数：

```
standardise_call(m1)
#> read.delim(file = "data.txt", sep = "|")
standardise_call(m2)
#> read.delim(file = "data.txt", sep = "|")
standardise_call(m3)
#> read.delim(file = "data.txt", sep = "|")
```

14.3.2 根据调用的元素来创建调用

可以应用 call() 或 as.call()，根据调用的元素来创建一个新调用。call() 的第一个参数是表示函数名的字符串。其他参数是表示调用参数的表达式。

```
call(":", 1, 10)
#> 1:10
call("mean", quote(1:10), na.rm = TRUE)
#> mean(1:10, na.rm = TRUE)
```

288 ~ 290

as.call() 是 call() 的一个变体，它以一个单独列表作为输入。第一个元素是名字或调用。接下来的元素是参数。

```
as.call(list(quote(mean), quote(1:10)))
#> mean(1:10)
as.call(list(quote(adder(10)), 20))
#> adder(10)(20)
```

14.3.3 练习

1. 下面的两个调用看起来相同，但实际上它们是不同的：

```
(a <- call("mean", 1:10))
#> mean(1:10)
(b <- call("mean", quote(1:10)))
#> mean(1:10)
identical(a, b)
#> [1] FALSE
```

它们的不同点是什么？应该使用哪个？

2. 编写一个纯 R 语言的 do.call()。

3. 通过使用 c() 连接一个调用和一个表达式的方式来创建一个列表。编写一个 concat() 函数，使下面的代码能够工作，使它可以将调用和附加参数结合起来。

```
concat(quote(f), a = 1, b = quote(mean(a)))
#> f(a = 1, b = mean(a))
```

4. 由于 list() 不属于表达式，所以我们应该创建一个更方便的调用构造函数，它可以自动地将列表组合到参数中。编写一个 make_call() 函数，使下面的代码能够工作。

```
make_call(quote(mean), list(quote(x), na.rm = TRUE))
#> mean(x, na.rm = TRUE)
make_call(quote(mean), quote(x), na.rm = TRUE)
#> mean(x, na.rm = TRUE)
```

5. `mode<-` 是如何工作的？它如何使用 `call()`？

6. 阅读 `pryr::standardise_call()` 的源代码。它是如何工作的？为什么需要 `is.primitive()`？

7. 在下面调用中，为什么 `standardise_call()` 不能很好地运行？

```
standardise_call(quote(mean(1:10, na.rm = TRUE)))
#> mean(x = 1:10, na.rm = TRUE)
standardise_call(quote(mean(n = T, 1:10)))
#> mean(x = 1:10, n = T)
standardise_call(quote(mean(x = 1:10, , TRUE)))
#> mean(x = 1:10, , TRUE)
```

291

8. 阅读 `pryr::modify_call()` 的文档。你觉得它是如何工作的？阅读源代码。

9. 使用 `ast()` 并做一些实验，找出 `if()` 调用中的 3 个参数。需要哪些元素？ `for()` 和 `while()` 调用的参数是什么？

14.4 捕获当前调用

很多基础 R 函数使用当前调用：导致当前函数运行的表达式。有两种方法可以捕获当前调用：

- `sys.call()` 准确地捕获用户的输入。
- `match.call()` 创建一个只使用命名参数的调用。它类似于对 `sys.call()` 的结果自动调用 `pryr::standardise_call()`。

```
f <- function(abc = 1, def = 2, ghi = 3) {
  list(sys = sys.call(), match = match.call())
}
f(d = 2, 2)
#> $sys
#> f(d = 2, 2)
#>
#> $match
#> f(abc = 2, def = 2)
```

对函数建模经常使用 `match.call()` 来捕获用于创建模型的调用。这样就可以对一个模型进行 `update()`，对一些原始参数进行修改之后对模型进行重新拟合。这里有一个可以运行 `update()` 的例子：

292

```
mod <- lm(mpg ~ wt, data = mtcars)
update(mod, formula = . ~ . + cyl)
#>
#> Call:
#> lm(formula = mpg ~ wt + cyl, data = mtcars)
#>
#> Coefficients:
#> (Intercept)          wt         cyl
#>       39.69       -3.19       -1.51
```

`update()` 是如何工作的？我们可以使用 `pryr` 中的一些工具来重新编写它，以便我们可以只关注算法的本质。

```
update_call <- function (object, formula., ...) {
  call <- object$call

  # Use update.formula to deal with formulas like . ~ .
```

```
  if (!missing(formula.)) {
    call$formula <- update.formula(formula(object), formula.)
  }

  modify_call(call, dots(...))
}
update_model <- function(object, formula., ...) {
  call <- update_call(object, formula., ...)
  eval(call, parent.frame())
}
update_model(mod, formula = . ~ . + cyl)
#>
#> Call:
#> lm(formula = mpg ~ wt + cyl, data = mtcars)
#>
#> Coefficients:
#> (Intercept)           wt          cyl
#>       39.69        -3.19        -1.51
```

原始的 `update()` 函数有一个 `evaluate` 参数，它控制函数返回调用还是返回结果。但是，从原则上，我觉得函数最好只能返回一种类型的对象，而不是根据函数的参数返回不同的类型。

293

这次重写还可以修补 `update()` 的一个小漏洞：它在全局环境中重新执行调用，而我们真正需要的是在模型第一次被拟合的环境中重新执行（即第一次被拟合时的公式环境）。

```
f <- function() {
  n <- 3
  lm(mpg ~ poly(wt, n), data = mtcars)
}
mod <- f()
update(mod, data = mtcars)
#> Error: object 'n' not found

update_model <- function(object, formula., ...) {
  call <- update_call(object, formula., ...)
  eval(call, environment(formula(object)))
}
update_model(mod, data = mtcars)
#>
#> Call:
#> lm(formula = mpg ~ poly(wt, n), data = mtcars)
#>
#> Coefficients:
#>  (Intercept)  poly(wt, n)1  poly(wt, n)2  poly(wt, n)3
#>       20.091       -29.116         8.636         0.275
```

这是一个需要记住的重要原则：如果希望重新运行 `match.call()` 捕获的代码，还需要捕获该代码执行的环境，通常使用 `parent.frame()`。这样做的缺点是：捕获环境同样意味着捕获该环境中的所有对象，所以不能释放内存。我们将在 18.2 节中详细讨论它。

有些基础 R 函数在没有必要的情况下使用 `match.call()`。例如，`write.csv()` 捕获对 `write.csv()` 的调用，并进行相应设置后再调用 `write.table()`：

```
write.csv <- function(...) {
  Call <- match.call(expand.dots = TRUE)
  for (arg in c("append", "col.names", "sep", "dec", "qmethod")) {
```

```
    if (!is.null(Call[[arg]])) {
      warning(gettextf("attempt to set '%s' ignored", arg))
    }
  }
  rn <- eval.parent(Call$row.names)
  Call$append <- NULL
  Call$col.names <- if (is.logical(rn) && !rn) TRUE else NA
  Call$sep <- ","
  Call$dec <- "."
  Call$qmethod <- "double"
  Call[[1L]] <- as.name("write.table")
  eval.parent(Call)
}
```

为了修复这个问题，可以使用常规函数调用语义来实现 `write.csv()`：

```
write.csv <- function(x, file = "", sep = ",", qmethod = "double",
                      ...) {
  write.table(x = x, file = file, sep = sep, qmethod = qmethod,
    ...)
}
```

这样就更容易理解：它只使用不同的默认值来调用 `write.table()`。这也修复了原来 `write.csv()` 中的一个小漏洞：`write.csv(mtcars, row = FALSE)` 会引起错误，但 `write.csv(mtcars, row.names = FALSE)` 不会。这里我们学习了：永远使用尽可能简单的工具来解决问题。

练习

1. 对比 `update_model()` 和 `update.default()`。
2. 为什么 `write.csv(mtcars, "mtcars.csv", row = FALSE)` 不能工作？原来的作者忘记了参数匹配的什么性质？
3. 使用 R 语言重新编写 `update.formula()`。
4. 有时需要找到调用当前函数的函数的调用函数（即，祖父函数，而不是父函数）。使用 `sys.call()` 或 `match.call()` 如何找到这个函数？

294 ~ 295

14.5 成对列表

成对列表是 R 以前留下来的遗产。它们的行为与列表相同，但是内部表示方式不同（像一个链表而不是一个向量）。除了应用在函数参数中外，成对列表已经被列表取代。

如果需要手动构建函数，才需要关注列表与成对列表的不同。例如，下面的函数允许我们从函数的组成成分（形式参数列表、函数体和环境）来创建一个函数。这个函数使用 `as.pairlist()` 函数确保 `function()` 有自己需要的成对列表 `args`。

```
make_function <- function(args, body, env = parent.frame()) {
  args <- as.pairlist(args)

  eval(call("function", args, body), env)
}
```

pryr 添加包中也有这个函数，不过它会做更多的参数检查。`make_function()` 最好

与参数列表函数 alist() 一起使用。alist() 不会参数求值，所以 alist(x = a) 是 list(x = quote(a)) 的简写。

```
add <- make_function(alist(a = 1, b = 2), quote(a + b))
add(1)
#> [1] 3
add(1, 2)
#> [1] 3

# To have an argument with no default, you need an explicit =
make_function(alist(a = , b = a), quote(a + b))
#> function (a, b = a)
#> a + b
# To take `...` as an argument put it on the LHS of =
make_function(alist(a = , b = , ... =), quote(a + b))
#> function (a, b, ...)
#> a + b
```

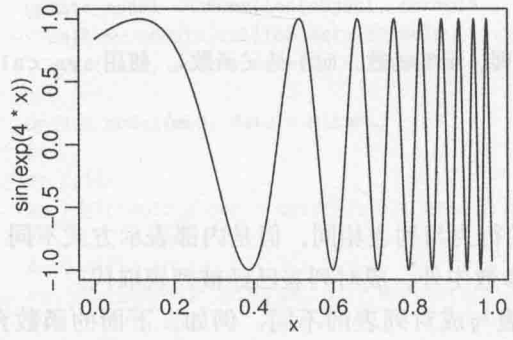
构建函数时，使用 make_function() 比使用闭包有一个优点：使用它，可以很容易地阅读源代码。例如，

296

```
adder <- function(x) {
  make_function(alist(y =), substitute({x + y}), parent.frame())
}
adder(10)
#> function (y)
#> {
#>     10 + y
#> }
```

make_function() 的一个很有用的应用就是在 curve() 这样的函数中。curve() 可以让我们在不需要创建显式的 R 函数的情况下绘制一个数学函数的曲线：

```
curve(sin(exp(4 * x)), n = 1000)
```



这里 x 是个代词。x 不代表某个具体值，而是可以随着绘制范围而改变的占位符。实现 curve() 的一个方法就是 make_function()：

```
curve2 <- function(expr, xlim = c(0, 1), n = 100,
                   env = parent.frame()) {
  f <- make_function(alist(x = ), substitute(expr), env)

  x <- seq(xlim[1], xlim[2], length = n)
  y <- f(x)
```

```
  plot(x, y, type = "l", ylab = deparse(substitute(expr)))
}
```

使用代词的函数称为回指（anaphoric）函数（http://en.wikipedia.org/wiki/Anaphora_(linguistics)）。它们应用在 Arc（http://www.arcfn.com/doc/anaphoric.html）（一种类似于 lisp 的语言）、Perl（http://www.perlmonks.org/index.pl?node_id=666047）和 Clojure（http://amalloy.hubpages.com/bub/Unhygenic-anaphoric-Clojure-macros-for-fun-and-profit）中。

练习

1. `alist(a)` 和 `alist(a = )` 的不同之处是什么？考虑输入和输出。
2. 阅读 `pryr::partial()` 的文档和源代码。它的功能是什么？它是如何工作的。阅读 `pryr::unenclose()` 的文档和源代码。它的功能是什么？它是如何工作的？
3. 真实的 `curve()` 函数看起来更像：

```
curve3 <- function(expr, xlim = c(0, 1), n = 100,
                   env = parent.frame()) {
  env2 <- new.env(parent = env)
  env2$x <- seq(xlim[1], xlim[2], length = n)

  y <- eval(substitute(expr), env2)
  plot(env2$x, y, type = "l",
    ylab = deparse(substitute(expr)))
}
```

这个方法与上面定义的 `curve2()` 有哪些不同？

14.6 解析与逆解析

有时候，代码表示为字符串而不是表达式。可以使用 `parse()` 将字符串转换成表达式。`parse()` 与 `deparse()` 的功能相反：它接收一个字符串向量，并返回一个表达式对象。`parse()` 的主要用途是将代码文件解析到硬盘，所以第一个参数是文件路径。注意，如果代码是字符向量，那么需要使用 `text` 参数：

297 ~ 298

```
z <- quote(y <- x * 10)
deparse(z)
#> [1] "y <- x * 10"

parse(text = deparse(z))
#> expression(y <- x * 10)
```

由于在文件中可能存在很多高级调用，所以 `parse()` 返回的不是一个单独的表达式，而是一个表达式对象，其实就是一个表达式列表：

```
exp <- parse(text = c("
  x <- 4
  x
  5
"))
length(exp)
#> [1] 3
typeof(exp)
#> [1] "expression"
```

```
exp[[1]]
#> x <- 4
exp[[2]]
#> x
```

使用 expression() 函数可以手动创建表达式对象，但是我不推荐这么做。如果你已经知道如何使用表达式，那么就没有必要学习这种神秘的数据结构。

使用 parse() 和 eval()，可以自己写一个简化版的 source() 函数。首先从硬盘中读取文件，parse() 它，然后在规定环境中 eval() 每一个组成成分。这个版本默认是在一个新的环境中执行，所以它不影响已有的对象。source() 以不可见的方式返回文件中最后一个表达式的结果，simple_source() 也一样。

```
simple_source <- function(file, envir = new.env()) {
  stopifnot(file.exists(file))
  stopifnot(is.environment(envir))

  lines <- readLines(file, warn = FALSE)
  exprs <- parse(text = lines)

  n <- length(exprs)
  if (n == 0L) return(invisible())

  for (i in seq_len(n - 1)) {
    eval(exprs[i], envir)
  }
  invisible(eval(exprs[n], envir))
}
```

真实的 source() 考虑了更复杂的情况，因为它可以 echo 输入和输出，而且还有很多附加的设置来控制程序的行为。

练习

1. quote() 和 expression() 有哪些不同？
2. 阅读 deparse() 的帮助文档，构建一个调用，使 deparse() 和 parse() 不能在该调用上对称使用。
3. 对比 source() 和 sys.source()。
4. 对 simple_source() 进行修改使它不仅能返回最后一个表达式的结果，而且还能返回每个表达式的结果。
5. simple_source() 产生的代码缺少源参考。阅读 sys.source() 的源代码和 srcfilecopy() 的帮助文档，然后对 simple_source() 进行修改，使其能够保留源参考。使用一个带有注释的函数来测试你的代码。如果测试成功，你不仅能够看到源代码还应该能够看到代码注释。

14.7 使用递归函数遍历抽象语法树

299 ~ 300

使用 substitute() 或 pryr::modify_call() 对单个调用进行修改是很容易的。对于更加复杂的任务，需要直接对 AST 进行处理。基础 codetools 包提供了一些可以帮助我们的例子：

- `findGlobals()` 可以定位函数使用到的所有全局变量。如果我们对函数进行检查，查看它是不是（由于我们的疏忽）依赖于父环境中的变量时会很有用。
- `checkUsage()` 可以检查一系列常见问题，包括未使用的局部变量、未使用的参数和部分参数匹配的使用情况。

为了编写 `findGlobals()` 和 `checkUsage()` 这样的函数，还需要一些新工具。由于表达式具有树结构，所以递归就是最自然的选择。实现它的关键是正确地进行递归。也就是确定基本情况是什么，并找到如何对递归得到的结果进行组合。对于调用来说，有两种基本情况（原子向量和名字）和两种递归情况（调用和成对列表）。处理表达式的函数看起来如下所示：

```
recurse_call <- function(x) {
  if (is.atomic(x)) {
    # Return a value
  } else if (is.name(x)) {
    # Return a value
  } else if (is.call(x)) {
    # Call recurse_call recursively
  } else if (is.pairlist(x)) {
    # Call recurse_call recursively
  } else {
    # User supplied incorrect input
    stop("Don't know how to handle type ", typeof(x),
      call. = FALSE)
  }
}
```

14.7.1 寻找 F 和 T

我们从一个简单的函数开始，这个函数可以判断一个函数是否使用了简写的逻辑符 `T` 和 `F`。代码中出现 `T` 和 `F`，通常认为是不好的，有时候 `R CMD check` 还会发出警告。首先对比 `T` 与 `TRUE` 的 AST：

301

```
ast(TRUE)
#> \-  TRUE
ast(T)
#> \- `T
```

`TRUE` 被解析为长度为 1 的逻辑向量，而 `T` 却被解析为一个名字。这样我们就知道如何编写递归函数的基本情况：虽然原子向量永远不可能是简写的逻辑符，但是名字是可以的，所以需要对 `T` 和 `F` 进行检测。递归情况可以结合到一起，因为这两种情况的处理是一样的：它们都是对对象的每一个元素递归地调用 `logical_abbr()`。

```
logical_abbr <- function(x) {
  if (is.atomic(x)) {
    FALSE
  } else if (is.name(x)) {
    identical(x, quote(T)) || identical(x, quote(F))
  } else if (is.call(x) || is.pairlist(x)) {
    for (i in seq_along(x)) {
```

```
      if (logical_abbr(x[[i]])) return(TRUE)
    }
    FALSE
  } else {
    stop("Don't know how to handle type ", typeof(x),
      call. = FALSE)
  }
}

logical_abbr(quote(TRUE))
#> [1] FALSE
logical_abbr(quote(T))
#> [1] TRUE
logical_abbr(quote(mean(x, na.rm = T)))
#> [1] TRUE
logical_abbr(quote(function(x, na.rm = T) FALSE))
#> [1] TRUE
```

14.7.2 寻找通过赋值创建的所有变量

`logical_abbr()` 非常简单：它只能返回一个单独的 TRUE 或 FALSE。下一个任务，列出通过赋值创建的所有变量，这有一点儿复杂。先从简单情况开始，然后逐步完善我们的函数。

302

同样，我们从查看赋值运算的 AST 开始：

```
ast(x <- 10)
#> \- ()
#>   \- `<-
#>   \- `x
#>   \- 10
```

赋值运算是一个调用，调用的第一个元素是名字 <-，第二个元素是名字被赋值的对象，第三个元素是被赋给的值。这样就使得基本情况比较简单了：常量和名字不会创建赋值，所以它们返回 NULL。递归情况也不太复杂。使用 `lapply()` 将函数应用于成对列表和调用而不是 <-。

```
find_assign <- function(x) {
  if (is.atomic(x) || is.name(x)) {
    NULL
  } else if (is.call(x)) {
    if (identical(x[[1]], quote(`<-`))) {
      x[[2]]
    } else {
      lapply(x, find_assign)
    }
  } else if (is.pairlist(x)) {
    lapply(x, find_assign)
  } else {
    stop("Don't know how to handle type ", typeof(x),
      call. = FALSE)
  }
```

```
}
find_assign(quote(a <- 1))
#> a
find_assign(quote({
  a <- 1
  b <- 2
}))
#> [[1]]
#> NULL
#>
#> [[2]]
#> a
#>
#> [[3]]
#> b
```

对于这个简单的例子，这个函数是没问题的，但是输出结果有点啰嗦，还包含一些多余的 NULL。为了保持简单的结果，返回一个字符向量，而不是返回一个列表。我们还使用了两个更复杂的例子来进行测试：

```
find_assign2 <- function(x) {
  if (is.atomic(x) || is.name(x)) {
    character()
  } else if (is.call(x)) {
    if (identical(x[[1]], quote(`<-`))) {
      as.character(x[[2]])
    } else {
      unlist(lapply(x, find_assign2))
    }
  } else if (is.pairlist(x)) {
    unlist(lapply(x, find_assign2))
  } else {
    stop("Don't know how to handle type ", typeof(x)
      call. = FALSE)
  }
}

find_assign2(quote({
  a <- 1
  b <- 2
  a <- 3
}))
#> [1] "a" "b" "a"

find_assign2(quote({
  system.time(x <- print(y <- 5))
}))
#> [1] "x"
```

303 ~ 304

这次好点了，但还存在两个问题：处理重复名字和忽略在其他赋值内部的赋值。第一个问题很容易修复。需要使用 unique() 对递归情况进行包装来去除重复的赋值。第二个问题的修复需要点技巧。当调用是 <- 时，仍然需要继续递归。find_assign3() 实现了这两点：

```
find_assign3 <- function(x) {
  if (is.atomic(x) || is.name(x)) {
```

```
    character()
  } else if (is.call(x)) {
    if (identical(x[[1]], quote(`<-`))) {
      lhs <- as.character(x[[2]])
    } else {
      lhs <- character()
    }

    unique(c(lhs, unlist(lapply(x, find_assign3))))
  } else if (is.pairlist(x)) {
    unique(unlist(lapply(x, find_assign3)))
  } else {
    stop("Don't know how to handle type ", typeof(x),
      call. = FALSE)
  }
}

find_assign3(quote({
  a <- 1
  b <- 2
  a <- 3
}))
#> [1] "a" "b"

find_assign3(quote({
  system.time(x <- print(y <- 5))
}))
#> [1] "x" "y"
```

还需要对子赋值（subassignment）进行测试：

```
find_assign3(quote({
  l <- list()
  l$a <- 5
  names(l) <- "b"
}))
#> [1] "l"     "$"     "a"     "names"
```

只需要对象本身的赋值，而不是修改对象属性进行赋值。绘制引用对象的树可以帮助我们了解需要测试哪些情况。调用 <- 的第二个元素应该是一个名字而不是其他调用。

```
ast(l$a <- 5)
#> \- ()
#>   \- `<-
#>   \- ()
#>     \- `$
#>     \- `l
#>     \- `a
#>   \- 5
ast(names(l) <- "b")
#> \- ()
#>   \- `<-
#>   \- ()
#>     \- `names
#>     \- `l
#>   \- "b"
```

现在我们有一个完整的函数：

```
find_assign4 <- function(x) {
  if (is.atomic(x) || is.name(x)) {
    character()
  } else if (is.call(x)) {
    if (identical(x[[1]], quote(`<-`)) && is.name(x[[2]])) {
      lhs <- as.character(x[[2]])
    } else {
      lhs <- character()
    }

    unique(c(lhs, unlist(lapply(x, find_assign4))))
  } else if (is.pairlist(x)) {
    unique(unlist(lapply(x, find_assign4)))
  } else {
    stop("Don't know how to handle type ", typeof(x),
      call. = FALSE)
  }
}

find_assign4(quote({
  l <- list()
  l$a <- 5
  names(l) <- "b"
}))
#> [1] "l"
```

虽然函数的完整版有点儿复杂，但要记住我们可以通过一步步编写简单的组成部分来实现它。

14.7.3 修改调用树

下一步更复杂的就是返回一个被修改的调用树，就像使用 `bquote()` 获得的结果一样。`bquote()` 是更加灵活的 `quote`：它允许我们有选择地对表达式的一些部分进行引用和非引用（有点像 Lisp 中的反引号运算符）。一切都是引用，除非使用 `.()` 括起来，这种情况下括号内的内容会被求值，然后再将结果插入：

```
a <- 1
b <- 3
bquote(a + b)
#> a + b
bquote(a + .(b))
#> a + 3
bquote(.(a) + .(b))
#> 1 + 3
bquote(.(a + b))
#> [1] 4
```

这样我们就可以非常容易地控制什么在什么时候被求值了。`bquote()` 是如何工作的？下面是我重写的 `bquote()`：它希望输入是已经被引用的，使基本情况和递归的情况更加明显：

```
bquote2 <- function (x, where = parent.frame()) {
  if (is.atomic(x) || is.name(x)) {
    # Leave unchanged
    x
```

```
  } else if (is.call(x)) {
    if (identical(x[[1]], quote(.))) {
      # Call to .(), so evaluate
      eval(x[[2]], where)
    } else {
      # Otherwise apply recursively, turning result back into call
      as.call(lapply(x, bquote2, where = where))
    }
  } else if (is.pairlist(x)) {
    as.pairlist(lapply(x, bquote2, where = where))
  } else {
    # User supplied incorrect input
    stop("Don't know how to handle type ", typeof(x),
      call. = FALSE)
  }
}

x <- 1
y <- 2
bquote2(quote(x == .(x)))
#> x == 1
bquote2(quote(function(x = .(x)) {
  x + .(y)
}))
#> function(x = 1) {
#>     x + 2
#> }
```

它与前面递归函数的主要不同是：在处理调用和成对列表的每个元素后，需要将它们强制转换成它们的原始类型。

注意，修改源树的函数对创建用于运行时的表达式非常有用，而不是将它们保存在原始的源代码文件中。这是因为所有的非代码信息都会丢失：

```
bquote2(quote(function(x = .(x)) {
  # This is a comment
  x +  # funky spacing
    .(y)
}))
#> function(x = 1) {
#>     x + 2
#> }
```

这些工具在某种程度上有点像 Lisp 宏，它在 Thomas Lumley 的《Programmer's Niche: Macros in R》(http://www.r-project.org/doc/Rnews/Rnews_2001-3.pdf#page=10）中有讨论。但是，宏是在编译时运行的，这对 R 来说没有什么意义，并且总是返回表达式。它们还有点象 Lisp fexpr（http://en.wikipedia.org/wiki/Fexpr）。fexpr 是一个函数，在默认情况下它的参数是不会被计算的。当需要从其他语言中查找一些有用的技术时，宏和 fexpr 这样的名词是很有用的。

14.7.4 练习

1. 为什么 `logical_abbr()` 使用了 `for` 循环而不是 `lapply()` 这样的函数？

2. 当给定一个引用对象时，`logical_abbr()` 能够执行；但当给定现有函数（如下所示），它就不能正

常执行，为什么？如何对其进行修改使它可以处理函数？思考函数是由哪些元素组成的。

```
f <- function(x = TRUE) {
  g(x + T)
}
logical_abbr(f)
```

3. 编写一个称为 `ast_type()` 的函数，它可以返回“constant”、“name”、“call”或“pairlist”。使用这个函数和 `switch()` 语句重写 `logical_abbr()`、`find_assign()` 和 `bquote2()`。
4. 编写一个可以提取一个函数的所有调用的函数。将它与 `pryr::fun_calls()` 进行比较。
5. 包装非标准计算的 `bquote2()`，这样就不需要显式地对输入进行 `quote()` 运算。
6. 比较 `bquote2()` 和 `bquote()`。`bquote()` 中有一个小漏洞：它不会替换参数为空的函数调用。为什么？ 305 ~ 309

```
bquote(.(x)(), list(x = quote(f)))
#> .(x)()
bquote(.(x)(1), list(x = quote(f)))
#> f(1)
```

7. 对基础 `recurse_call()` 模板进行改进，使它也可以处理函数列表和表达式列表（如 `parse(path_to_file)`）。 310

第 15 章
领域特定语言

第一类环境、词法作用域、非标准计算和元编程的组合为我们在 R 语言中创建嵌入式领域特定语言（Domain Specific Language，DSL）提供了一个强大的工具箱。嵌入式 DSL 采用了宿主语言的解析和执行框架，但是对语义进行了调整使它更适合特定工作。DSL 是一个非常大的主题，本章只是对其进行比较浅显的讨论，重点集中在重要的实现技术上而不是当初如何提出这个语言的。如果对这些感兴趣，非常推荐你阅读 Martin Fowler 的《Domain Specific Languages》（http://amzn.com/0321712943?tag=devtools-20）。它讨论了创建 DSL 的很多选项，并提供了不同语言的很多例子。

R 语言中最流行的 DSL 是公式的设定，它为我们提供了一种描述模型、预测变量与响应变量之间关系的简洁方法。其他的例子包括 `ggplot2`（可视化）和 `plyr`（数据处理）。另一个大量使用这一思想的添加包是 `dplyr`，它提供 `translate_sql()` 函数，将 R 表达式转换成 SQL：

```
library(dplyr)
translate_sql(sin(x) + tan(y))
#> <SQL> SIN("x") + TAN("y")
translate_sql(x < 5 & !(y >= 5))
#> <SQL> "x" < 5.0 AND NOT(("y" >= 5.0))
translate_sql(first %like% "Had*")
#> <SQL> "first" LIKE 'Had*'
translate_sql(first %in% c("John", "Roger", "Robert"))
#> <SQL> "first" IN ('John', 'Roger', 'Robert')
translate_sql(like == 7)
#> <SQL> "like" = 7.0
```

311 本章开发两个简单但很有用的 DSL：一个产生 HTML，另一个将 R 语言编写的数学表达式转换成 LaTex。

预备条件

本章使用本书中讨论的多种技术。尤其是，要理解环境、泛函、非标准计算和元编程。

15.1 HTML

HTML（超文本标记语言）是大多数网页的编写语言。它是 SGML（标准广义标记语言）的一个特例，它与 XML 相似但不完全相同。HTML 一般是这样的：

```
<body>
  <h1 id='first'>A heading</h1>
  <p>Some text &amp;amp; <b>some bold text.</b></p>
  <img src='myimg.png' width='100' height='100' />
</body>
```

即使你以前从来没有见过 HTML 代码，你也能够看到编码结构的关键成分是标签：`<tag></tag>`。标签可以在其他标签之中，中间夹杂着文本。通常情况下，HTML 忽略空白（一连串的空白等价于一个空格），所以可以将上面的例子放入一行中，但在浏览器中显示的结果却是一样的：

```
<body><h1 id='first'>A heading</h1><p>Some text &amp;amp; <b>some bold
text.</b></p><img src='myimg.png' width='100' height='100' />
</body>
```

但是，与 R 代码一样，我们通常希望对 HTML 代码进行缩进，这样整个代码的结构会更清晰。

现在已经有超过 100 个 HTML 标签。但是为了说明 HTML 语言，我们只需要关注其中的一小部分：

- `<body>`：最顶层的标签，所有的内容都包含在其中。
- `<h1>`：创建一级标题，最顶层的标题。
- `<p>`：创建段落。
- `<b>`：加粗文字。
- `<img>`：嵌入图像。

（也许你已经猜到这些标签的功能了！）

标签也可以具有命名属性。它们看上去就像 `<tag a="a" b="b"></tag>`。标签值应该总是被单引号或双引号括起来。每个标签都使用的两个重要属性是 `id` 和 `class`。这些通常与层叠样式表（Cascading Style Sheet，CSS）一起使用来控制文档的格式。

有些标签，如 `<img>`，不能有任何内容。这些称为**空白标签**（void tag），它们的语法也有些不同。不是写成 `<img></img>`，而写成 `<img/>`。因为它们没有内容，所以属性就更重要。实际上，`img` 有 3 个几乎每个图像都要使用的重属性：`src`（图像的位置）、`width` 和 `height`。

由于 < 和 > 在 HTML 中具有特殊的意义，所以不能直接使用它们。而是必须使用 HTML 转义符：`&gt;` 和 `&lt;`。由于这些转义符使用了 &，所以当需要使用 & 时就必须使用 `&amp;`。

15.1.1　目标

我们的目标是从 R 代码中更容易地产生 HTML 代码。给一个具体的例子，我们希望使用 R 代码产生下面的 HTML 代码，越像越好。

```
<body>
  <h1 id='first'>A heading</h1>
  <p>Some text &amp;amp; <b>some bold text.</b></p>
  <img src='myimg.png' width='100' height='100' />
</body>
```

为此，我们将一步步地实现下面的 DSL：

```
with_html(body(
  h1("A heading", id = "first"),
  p("Some text &", b("some bold text.")),
  img(src = "myimg.png", width = 100, height = 100)
313 ))
```

注意，这里函数调用的嵌套与标签的嵌套相同：未命名参数变成了标签的内容，命名参数变成了标签的属性。因为在这个 API 中标签和文本明显不同，所以我们可以自动地对 & 和其他特殊字符进行转义。

15.1.2 转义

转义是 DSL 的基础，所以我们把它作为第一个主题。为了创建一个对字符进行转义的方法，需要给“ &”一个不以两个空格为结尾的特殊含义。最简单的方法就是创建一个可以区分常规文本（需要转义）和 HTML（不需要转义）的 S3 类。

```
html <- function(x) structure(x, class = "html")
print.html <- function(x, ...) {
  out <- paste0("<HTML> ", x)
  cat(paste(strwrap(out), collapse = "\n"), "\n", sep = "")
}
```

然后编写转义的方法，它保持 HTML 不改变并对普通文本的特殊字符（&、<、>）进行转义。为了方便起见，再增加一个列表方法。

```
escape <- function(x) UseMethod("escape")
escape.html <- function(x) x
escape.character <- function(x) {
  x <- gsub("&", "&amp;", x)
  x <- gsub("<", "&lt;", x)
  x <- gsub(">", "&gt;", x)

  html(x)
}
escape.list <- function(x) {
  lapply(x, escape)
}

# Now we check that it works
escape("This is some text.")
#> <HTML> This is some text.
escape("x > 1 & y < 2")
#> <HTML> x &gt; 1 &amp; y &lt; 2

# Double escaping is not a problem
escape(escape("This is some text. 1 > 2"))
#> <HTML> This is some text. 1 &gt; 2

# And text we know is HTML doesn't get escaped.
escape(html("<hr />"))
#> <HTML> <hr />
```

对于很多 DSL 来说，转义都是一个非常重要的组成成分。

15.1.3 基本标签函数

下面，我们编写一个简单的标签函数，然后将这个函数推广让它涵盖所有可能的 HTML 标签。从 <p> 开始。HTML 标签可以包含属性（例如，id 或 class）和子标签（例如，<b> 或 <i>）。我们需要一些在函数调用中对它们进行分割的方法。给定属性是命名的值，子标签没有名字，很自然可以根据有没有名字的参数将它们区分。例如，对 p() 的调用应该是：

```
p("Some text.", b("some bold text"), class = "mypara")
```

在函数定义中，可以将标签 <p> 的所有属性都列出来。但是，这几乎是不可能的，因为有太多的属性，甚至还可能使用自定义属性（http://html5doctor.com/html5-custom-data-attributes/）。所以，我们只使用 ...，并根据它们是否被命名来进行区分。为此，我们需要注意 names() 的不一致：

```
names(c(a = 1, b = 2))
#> [1] "a" "b"
names(c(a = 1, 2))
#> [1] "a" ""
names(c(1, 2))
#> NULL
```

314
〜
315

记住，我们可以创建两个帮助函数来提取向量的命名成分和非命名成分：

```
named <- function(x) {
  if (is.null(names(x))) return(NULL)
  x[names(x) != ""]
}
unnamed <- function(x) {
  if (is.null(names(x))) return(x)
  x[names(x) == ""]
}
```

现在我们可以创建自己的 p() 函数。注意，这里有一个新函数：html_attributes()。它使用一个名字 – 值对列表来创建正确的 HTML 属性设置。它有点儿复杂（从某种程度上来看，因为它处理本章没有讲到的一些 HTML 特质）。但是，由于它并不是非常重要而且没有引入任何新的想法，所以这里就不讨论该函数了（可以在网上找到源代码）。

```
source("dsl-html-attributes.r", local = TRUE)
p <- function(...) {
  args <- list(...)
  attribs <- html_attributes(named(args))
  children <- unlist(escape(unnamed(args)))

  html(paste0(
    "<p", attribs, ">",
    paste(children, collapse = ""),
    "</p>"
  ))
}

p("Some text")
#> <HTML> <p>Some text</p>
p("Some text", id = "myid")
#> <HTML> <p id = 'myid'>Some text</p>
p("Some text", image = NULL)
```

```
#> <HTML> <p image>Some text</p>
p("Some text", class = "important", "data-value" = 10)
#> <HTML> <p class = 'important' data-value = '10'>Some
#> text</p>
```

316

15.1.4 标签函数

使用 p() 的这个定义，可以很容易地将这个方法应用到不同的标签：只需要用一个变量来替换 "p"。给定标签名，我们将使用闭包产生一个标签函数：

```
tag <- function(tag) {
  force(tag)
  function(...) {
    args <- list(...)
    attribs <- html_attributes(named(args))
    children <- unlist(escape(unnamed(args)))

    html(paste0(
      "<", tag, attribs, ">",
      paste(children, collapse = ""),
      "</", tag, ">"
    ))
  }
}
```

（我们对 tag 进行强制求值，这是为了我们可以在循环中调用这个函数。这有助于避免由惰性求值带来的潜在漏洞。）

现在可以运行前面的例子：

```
p <- tag("p")
b <- tag("b")
i <- tag("i")
p("Some text.", b("Some bold text"), i("Some italic text"),
  class = "mypara")
#> <HTML> <p class = 'mypara'>Some text.<b>Some bold
#> text</b><i>Some italic text</i></p>
```

在我们继续为每一个可能的 HTML 标签编写函数前，我们还需要为空白标签创建一个 tag() 函数的变体。它与 tag() 非常相似，但是如果有未命名的标签，它需要抛出错误。还要注意，标签本身看起来也有些不同：

```
void_tag <- function(tag) {
  force(tag)
  function(...) {
    args <- list(...)
    if (length(unnamed(args)) > 0) {
      stop("Tag ", tag, " can not have children", call. = FALSE)
    }
    attribs <- html_attributes(named(args))

    html(paste0("<", tag, attribs, " />"))
  }
}

img <- void_tag("img")
img(src = "myimage.png", width = 100, height = 100)
```

```
#> <HTML> <img src = 'myimage.png' width = '100' height =
#> '100' />
```

15.1.5　处理所有标签

下面需要一个由所有 HTML 标签构成的列表：

```
tags <- c("a", "abbr", "address", "article", "aside", "audio",
  "b","bdi", "bdo", "blockquote", "body", "button", "canvas",
  "caption","cite", "code", "colgroup", "data", "datalist",
  "dd", "del","details", "dfn", "div", "dl", "dt", "em",
  "eventsource","fieldset", "figcaption", "figure", "footer",
  "form", "h1", "h2", "h3", "h4", "h5", "h6", "head", "header",
  "hgroup", "html", "i","iframe", "ins", "kbd", "label",
  "legend", "li", "mark", "map","menu", "meter", "nav",
  "noscript", "object", "ol", "optgroup", "option", "output",
  "p", "pre", "progress", "q", "ruby", "rp","rt", "s", "samp",
  "script", "section", "select", "small", "span", "strong",
  "style", "sub", "summary", "sup", "table", "tbody", "td",
  "textarea", "tfoot", "th", "thead", "time", "title", "tr",
  "u", "ul", "var", "video")

void_tags <- c("area", "base", "br", "col", "command", "embed",
  "hr", "img", "input", "keygen", "link", "meta", "param",
  "source", "track", "wbr")
```

如果仔细地查看这个列表，就会发现有很少一部分标签的名字与某些 R 基础函数的名字相同（`body`、`col`、`q`、`source`、`sub`、`summary`、`table`），还有一些与其他的添加包同名（例如，`map`）。这就意味着在全局环境或者添加包的环境中，不希望默认地访问所有的函数。我们希望将它们放入一个列表，再添加一些额外代码，以便在我们需要时可以方便地应用它们。首先，创建一个命名列表：

317 ~ 318

```
tag_fs <- c(
  setNames(lapply(tags, tag), tags),
  setNames(lapply(void_tags, void_tag), void_tags)
)
```

这样我们就得到了一个显式（但有点儿冗长）调用标签函数的方法：

```
tag_fs$p("Some text.", tag_fs$b("Some bold text"),
  tag_fs$i("Some italic text"))
#> <HTML> <p>Some text.<b>Some bold text</b><i>Some
#> italic text</i></p>
```

我们用一个在列表的上下文中执行代码的函数来完成 HTML DSL。

```
with_html <- function(code) {
  eval(substitute(code), tag_fs)
}
```

这样我们就有了一个简洁的 API，它使得在需要时我们可以写出 HTML 代码，而在不需要时，它不会混乱名称空间。

```
with_html(body(
  h1("A heading", id = "first"),
  p("Some text &", b("some bold text.")),
  img(src = "myimg.png", width = 100, height = 100)
))
```

```
#> <HTML> <body><h1 id = 'first'>A heading</h1><p>Some
#> text &amp;<b>some bold text.</b></p><img src =
#> 'myimg.png' width = '100' height = '100' /></body>
```

如果需要应用由于和 with_html() 内的 HTML 标签同名而被覆盖的 R 函数，可以使用 package::function 的方式进行显式设定。

319

15.1.6 练习

1. 标签 `<script>` 和 `<style>` 的转义规则是不同的：我们不希望对尖括号或 & 符号进行转义，但希望对 `</script>` 或 `</style>` 进行转义。对上面的代码做出调整使它遵守这个规则。
2. 对所有的函数都使用 ... 有一个很大的缺点。没有输入验证，它们如何在函数中应用的文档以及如何自动补全的信息很少。创建一个新函数，当给定标签的命名列表和它们的属性名（如下所示）时，创建解决上述问题的函数。

```
list(
  a = c("href"),
  img = c("src", "width", "height")
)
```

所有的标签都应该有 `class` 和 `id` 属性。

3. 现在，这个 HTML 还是不够漂亮，很难看出结构。如何对 `tag()` 进行修改使它可以实现缩进和格式化？

15.2 LaTeX

接下来要学习的 DSL 可以将 R 表达式转换成与其对应的 LaTex 数学表达式（这有点像 `?plotmath`，但是现在是文本而不是图形）。LaTex 是数学家和统计学家使用的通用语：当需要用文本的形式描绘一个等式时（例如，写邮件），就把它写成 LaTex 等式。由于很多报告都是使用 R 和 LaTex 创建的，所以能够自动地将数学表达式从一种语言转换成另一种语言是很有用的。

由于我们需要同时转换函数和名字，所以这个数学 DSL 比前面的 HTML DSL 更复杂一点儿。我们同样要创建一个“默认的”转换，这样我们不知道的函数也能获得一个标准转换。与 HTML DSL 类似，我们还要编写一个泛函来更方便地产生翻译器。

320 在开始前，先快速地学习如何使用 LaTeX 表示数学公式。

15.2.1 LaTeX 数学

LaTeX 数学是复杂的。幸运的是，它们的文档很完善（http://en.wikibooks.org/wiki/LaTex/Mathematics）。也就是说，它们的结构非常简单。

- 大多数简单的数学等式与使用 R 语言书写的格式相同：`x * y`、`z ^ 5`。使用 _ 来写下标（例如，`x_1`）。
- 特殊字符以 \ 为开头：`\pi` = π、`\pm` = ± 等。在 LaTeX 中还有大量的符号。谷歌搜索 `latex math symbols` 会返回很多列表（http://www.sunilpatel.co.uk/latex-type/latex-math-symbols/）。甚至还有一个服务（http://detexify.kirelabs.org/classify.html），你在浏览器中绘制一个符号，它就帮你查找相应的 LaTeX 符号。

- 更复杂的函数看起来像 `\name{arg1}{arg2}`。例如，为了写一个分数，需要使用 `\frac{a}{b}`。为了写一个平方根，需要使用 `\sqrt{a}`。
- 使用 `{}` 可以将元素分组：即，`x ^ a + b` 与 `x ^ {a + b}`。
- 在好的数学排版系统中，变量和函数之间是有区别的。但是，如果没有更多的信息，LaTeX 不知道 `f(a * b)` 代表以 `a * b` 作为输入调用函数 `f`，还是 `f * (a * b)` 的简写。如果 `f` 是一个函数，我们可以告诉 LaTeX 使用 `\textrm{f}(a * b)` 的垂直字体进行排版。

15.2.2　目标

我们的目标是自动地将 R 表达式转换成它的适当的 LaTeX 表示方式。我们将分 4 个阶段来实现这个目标：

- 转换已知符号：`pi` → `\pi`。
- 保留不需要转换的符号：`x` → `x`、`y` → `y`。
- 将已知函数转换成它们特定的形式：`sqrt(frac(a,b))` → `\sqrt{\frac{a,b}}`。
- 使用 `\textrm` 对未知函数进行包装：`f(a)` → `\textrm{f}(a)`。

该转换的编码方式与上面处理 HTML DSL 的方式相反。我们从基础结构开始，因为这样就可以很容易地对 DSL 进行测试，然后再一步步完善，直到得到想要的输出。 321

15.2.3　to_math

首先，需要一个封装函数，它将 R 表达式转换成 LaTeX 数学表达式。它的工作方式与 `to_html()` 相同：捕获未求值的表达式并在一个特殊环境中计算它。但是，这个特殊环境不再是固定的了。它根据表达式而发生改变。这样做的目的是处理我们还没有见过的符号和函数。

```
to_math <- function(x) {
  expr <- substitute(x)
  eval(expr, latex_env(expr))
}
```

15.2.4　已知符号

第一步是创建一个转换希腊字母的特殊 LaTeX 符号（例如，`pi` 转换为 `\pi`）的环境。这是在 `subset` 中使用的基本技能，它可以根据名字来选择列范围（`subset(mtcars, , cyl:wt)`）：在特殊环境中将一个名字与一个字符串进行绑定。

可以通过下述方式来创建环境：命名一个向量，将向量转换成列表，把列表转换成环境。

```
greek <- c(
  "alpha", "theta", "tau", "beta", "vartheta", "pi", "upsilon",
  "gamma", "gamma", "varpi", "phi", "delta", "kappa", "rho",
  "varphi", "epsilon", "lambda", "varrho", "chi", "varepsilon",
  "mu", "sigma", "psi", "zeta", "nu", "varsigma", "omega", "eta",
  "xi", "Gamma", "Lambda", "Sigma", "Psi", "Delta", "Xi",
  "Upsilon", "Omega", "Theta", "Pi", "Phi")
greek_list <- setNames(paste0("\\", greek), greek)
greek_env <- list2env(as.list(greek_list), parent = emptyenv())
```

然后，我们可以检查它：

```
latex_env <- function(expr) {
  greek_env
}

to_math(pi)
#> [1] "\\pi"
to_math(beta)
#> [1] "\\beta"
```

322 ≀ 324

15.2.5 未知符号

如果符号不是希腊字母，那么就保留它的原状。这有点儿技巧，因为我们事先不知道要使用到什么符号，也不可能全部生成它们。所以，我们使用一些元编程技术来找出表达式中出现的符号。函数 `all_names` 接受一个表达式，然后进行如下动作：如果它是一个名字，将它转换成字符串；如果它是一个调用，通过它的参数继续递归。

```
all_names <- function(x) {
  if (is.atomic(x)) {
    character()
  } else if (is.name(x)) {
    as.character(x)
  } else if (is.call(x) || is.pairlist(x)) {
    children <- lapply(x[-1], all_names)
    unique(unlist(children))
  } else {
    stop("Don't know how to handle type ", typeof(x),
      call. = FALSE)
  }
}

all_names(quote(x + y + f(a, b, c, 10)))
#> [1] "x" "y" "a" "b" "c"
```

现在我们想接受那个符号列表，并将其转换到一个环境中，这样每个符号都映射到对应的字符串表示（例如，`eval(quote(x), env)` 将输出 `"x"`）。再次使用将命名字符向量转换成列表的转换模式，然后将列表转换到环境中。

```
latex_env <- function(expr) {
  names <- all_names(expr)
  symbol_list <- setNames(as.list(names), names)
  symbol_env <- list2env(symbol_list)
  symbol_env
}

to_math(x)
#> [1] "x"
to_math(longvariablename)
#> [1] "longvariablename"
to_math(pi)
#> [1] "pi"
```

这样做是可以的，但是需要把它与希腊符号环境结合起来。由于相对于默认情况，我们首选希腊符号（例如，`to_math(pi)` 将转换成 `"\\pi"`，而不是 `"pi"`），所以 `symbol_env`

应该是 greek_env 的父环境。为此，我们需要在一个新的父环境中对 greek_env 进行复制。虽然 R 中没有专门对环境进行复制的函数，但是可以通过组合两个现有函数很容易地创建一个：

```
clone_env <- function(env, parent = parent.env(env)) {
  list2env(as.list(env), parent = parent)
}
```

这样，我们得到一个可以对已知（希腊）和未知符号进行转换的函数。

```
latex_env <- function(expr) {
  # Unknown symbols
  names <- all_names(expr)
  symbol_list <- setNames(as.list(names), names)
  symbol_env <- list2env(symbol_list)

  # Known symbols
  clone_env(greek_env, symbol_env)
}

to_math(x)
#> [1] "x"
to_math(longvariablename)
#> [1] "longvariablename"
to_math(pi)
#> [1] "\\pi"
```

15.2.6 已知函数

接下来，为 DSL 添加函数。我们从多个帮助闭包开始，它可以使我们很容易地添加一元和二元运算符。这些函数都非常简单：它们只是对字符串进行组合。（又一次使用 force()，保证在正确的时间对参数进行计算）。

325 ~ 326

```
unary_op <- function(left, right) {
  force(left)
  force(right)
  function(e1) {
    paste0(left, e1, right)
  }
}

binary_op <- function(sep) {
  force(sep)
  function(e1, e2) {
    paste0(e1, sep, e2)
  }
}
```

有了这些帮助闭包，我们可以映射多个从 R 转换成 LaTeX 的例子。注意，借助 R 的词法作用域，我们可以很容易地为标准函数（如，+、- 和 *，甚至 (和 {）提供新的含义。

```
# Binary operators
f_env <- new.env(parent = emptyenv())
f_env$"+" <- binary_op(" + ")
f_env$"-" <- binary_op(" - ")
f_env$"*" <- binary_op(" * ")
```

```
f_env$"/" <- binary_op(" / ")
f_env$"^" <- binary_op("^")
f_env$"[" <- binary_op("_")

# Grouping
f_env$"{" <- unary_op("\\left{ ", " \\right}")
f_env$"(" <- unary_op("\\left( ", " \\right)")
f_env$paste <- paste

# Other math functions
f_env$sqrt <- unary_op("\\sqrt{", "}")
f_env$sin <- unary_op("\\sin(", ")")
f_env$log <- unary_op("\\log(", ")")
f_env$abs <- unary_op("\\left| ", "\\right| ")
f_env$frac <- function(a, b) {
  paste0("\\frac{", a, "}{", b, "}")
}

# Labelling
f_env$hat <- unary_op("\\hat{", "}")
f_env$tilde <- unary_op("\\tilde{", "}")
```

我们再次修改 `latex_env()`，使其包含这个环境。它应该是 R 查找名字的最后一个环境：换句话说，`sin(sin)` 应该能够工作了。

```
latex_env <- function(expr) {
  # Known functions
  f_env

  # Default symbols
  names <- all_names(expr)
  symbol_list <- setNames(as.list(names), names)
  symbol_env <- list2env(symbol_list, parent = f_env)

  # Known symbols
  greek_env <- clone_env(greek_env, parent = symbol_env)
}

to_math(sin(x + pi))
#> [1] "\\sin(x + \\pi)"
to_math(log(x_i ^ 2))
#> [1] "\\log(x_i^2)"
to_math(sin(sin))
#> [1] "\\sin(sin)"
```

15.2.7 未知函数

最后，我们为我们还不知道的函数添加默认情况。与未知名字一样，我们不可能提前知道它们是什么，所以我们又要利用元编程来处理它们：

```
all_calls <- function(x) {
  if (is.atomic(x) || is.name(x)) {
    character()
  } else if (is.call(x)) {
    fname <- as.character(x[[1]])
    children <- lapply(x[-1], all_calls)
    unique(c(fname, unlist(children)))
```

```
  } else if (is.pairlist(x)) {
    unique(unlist(lapply(x[-1], all_calls), use.names = FALSE))
  } else {
    stop("Don't know how to handle type ", typeof(x), call. = FALSE)
  }
}

all_calls(quote(f(g + b, c, d(a))))
#> [1] "f" "+" "d"
```

我们还需要一个闭包，它可以为每一个未知调用产生函数。

```
unknown_op <- function(op) {
  force(op)
  function(...) {
    contents <- paste(..., collapse = ", ")
    paste0("\\mathrm{", op, "}(", contents, ")")
  }
}
```

再次更新 `latex_env()`：

```
latex_env <- function(expr) {
  calls <- all_calls(expr)
  call_list <- setNames(lapply(calls, unknown_op), calls)
  call_env <- list2env(call_list)

  # Known functions
  f_env <- clone_env(f_env, call_env)

  # Default symbols
  symbols <- all_names(expr)
  symbol_list <- setNames(as.list(symbols), symbols)
  symbol_env <- list2env(symbol_list, parent = f_env)
  # Known symbols
  greek_env <- clone_env(greek_env, parent = symbol_env)
}

to_math(f(a * b))
#> [1] "\\mathrm{f}(a * b)"
```

15.2.8　练习

1. 添加转义。通过在前面添加反斜杠来转义的特殊字符号有：\、$ 和 %。与 HTML 相同，我们需要确保不能以双反斜杠结尾。所以我们需要创建一个小的 S3 类，然后在函数运算符中使用。这样就可以在需要时嵌入任意的 LaTeX 代码。
2. 完成这个 DSL，使它支持 `plotmath` 支持的所有函数。
3. 在 `latex_env()` 中有一个重复模式：接受一个字符向量，对每个元素进行操作，将其转换成列表，将列表转换成环境。编写一个函数可以自动完成这项任务，然后重写 `latex_env()`。
4. 研究 `dplyr` 的源代码。它结构的一个非常重要的部分是 `partial_eval()`，当有些组成成分需要参考数据库中的变量而其他组成成分需要参考本地 R 对象时，它有助于管理表达式。注意如果想将小的 R 表达式转换成其他语言（例如，JavaScript 或 Python），可以使用非常相似的想法。

327
⌇
328

第四部分

性　　能

第 16 章
性　　能

R 不是一门速度快的语言。这并不意外。R 的设计目的是使数据分析和统计变得容易。它并不是为了使计算机的生活变得轻松。尽管与其他编程语言相比，R 有点儿慢，但是对于大多数的应用，R 的速度足够快了。

本书该部分的目的是让你对 R 的性能特点有一个深入的了解。在本章中，你将学习 R 在灵活性和性能之间权衡。当需要提高 R 的速度时，接下来的 4 章给出了提高 R 代码速度的技巧：

- 第 17 章将学习如何系统化地使得你的代码变快。首先，你要找出什么是慢的，然后你应用一些通用的技巧使速度慢的这部分变快。
- 第 18 章将学习 R 如何利用内存，垃圾回收、复制后修改如何影响 R 的性能和内存利用。
- 对于真正的高性能代码，可以把它们移到 R 的外面，并使用其他编程语言实现。第 19 章学习需要知道的关于 C++ 的基本知识，这样你可以用 Rcpp 编写快速代码。
- 为了真正理解内置基础函数的性能，需要学习一些关于 R 语言的 C 语言应用程序接口（API）。第 20 章将学习一些有关 R 的内置 C 语言的知识。

329 ~ 331

我们从学习 R 为什么速度慢开始。

16.1　R 为什么速度慢

为了理解 R 的性能，考虑 R 既作为语言又作为语言的实现是有帮助的。R 语言是抽象的：它定义了 R 代码的含义和它们如何工作的方式。实现是具体的：它读 R 代码并计算结果。最流行的实现就是 R-project.org（http://r-project.org）。我称它为 GNU-R，以便区别于 R 语言和本章后面要讨论的其他实现。

R 语言和 GNU-R 的区别有点儿模糊，因为 R 语言没有正式的定义。尽管有 R 语言的定义（http://cran.r-project.org/doc/manuals/R-lang.html），但它是非正式的且是不完整的。R 语言至多定义了 GNU-R 如何工作。这与其他语言相反（如 C++（http://isocpp.org/std/the-standard）　和 javascript（http://www.ecma-international.org/publications/standards/Ecma-262.htm）），这些编程语言在语言和其实现之间有明确的区分，在它们语言的正式说明中描述了该语言的各个方面应该如何工作的每一个具体细节。然而，R 语言和 GNU-R 之间的区别还

是有用的：性能较弱，因为在不彻底改变现有代码的情况下，很难修正该语言。因为实现比较容易，所以可以修正较弱的性能。

在 16.3 节，讨论 R 语言设计对 R 速度有本质影响的某些方面。在 16.4 节，讨论为什么现在的 GNU-R 远没有达到理论上的最快，为什么性能提高的进展如此缓慢。尽管很难准确知道一个好的实现能够提高到多快，然而超过 10 倍的速度提高看起来是能够达到的。在 16.5 节，讨论一些有希望的 R 的新实现，描述它们用来使得 R 变快的重要技术。

除了由于设计和实现导致的性能限制外，必须指出许多 R 代码速度慢的原因是由于编写的不好。极少数的 R 用户具有正规的编程和软件开发训练。少数人通过编写 R 程序谋生。大多数人用 R 来理解数据：快速得到问题的答案远比开发一个在各种情况下都适用的系统重要很多。这说明可以相对较容易地让大多数 R 代码运行得更快。从下面的几章可以看出这点。 332

在考察 R 语言和 GNU-R 的速度较慢的部分前，我们先学习一些衡量测试，这样就使我们对性能的直觉有了具体的依据。

16.2 微测试

微测试是对一小段代码性能的度量，有时也就运行几微秒（μs）或者几纳秒（ns）。我打算用微测试来展示最底层 R 代码的性能，这可以帮助你对 R 如何工作的有一个直观的了解。总的来说，直觉对于提高真正代码的速度不太有用。用微测试观测到的差异通常是由真实代码的高阶影响决定的，对亚原子物理的深刻理解对烤面包没有帮助。不要因为这些微测试而改变你的编码方式。应该等待，直到你阅读完接下来几章给出的实用建议。

在 R 中，最好的微测试工具是 `microbenchmark`（http://cran.r-project.org/web/packages/microbenchmark/）添加包。该包中提供了非常精确的计时，使得比较极短时间的运算成为可能。下面的代码比较了 2 种计算平方根方法的速度：

```
library(microbenchmark)

x <- runif(100)
microbenchmark(
  sqrt(x),
  x ^ 0.5
)
#> Unit: nanoseconds
#>      expr   min    lq median    uq    max neval
#>  sqrt(x)   595   625    640   680  4,380   100
#>    x^0.5 3,660 3,700  3,730 3,770 33,600   100
```

默认情况下，`microbenchmark()` 对每一个表达式运行 100 次（由参数 `times` 控制）。在这个过程中，它也对表达式的顺序进行随机化。它对运行的结果进行汇总：最小值（`min`）、低四分位数（`lq`）、中位数、高四分位数（`uq`）和最大值（`max`）。关注中位数并使用高四分位数和低四分位数（`lq` 和 `uq`）对时间的波动产生一种直觉。从本例可以看出特殊用途的 `sqrt()` 函数比普通的指数运算符快很多。 333

对于所有的微测试，要特别注意单位：每一个计算用时 800 纳秒。为了有助于校准微测试对运行时间的影响，考虑一个函数需要运行多少次才能用时 1 秒，是有益的。如果一个微测试用时：

- 1 毫秒，那么 1 000 次调用用时 1 秒。
- 1 微秒，那么 100 万次调用用时 1 秒。
- 1 纳秒，那么 10 亿次调用用时 1 秒。

计算 100 个数的平方根 `sqrt()` 用时 800 纳秒或者 0.8 微秒。这说明如果重复运行该运算 100 万次，它将耗时 0.8 秒。因此，改变计算平方根的方法不会显著地影响真正的代码。

练习

1. 除了应用函数 `microbenchmark()` 外，你也可以应用 R 的内部函数 `system.time()`。但是，函数 `system.time()` 的精确度要差很多，因此需要应用循环来对每一个运算重复进行多次，然后找出每一个运算的平均运行时间。如下面的代码所示。

```
n <- 1:1e6
system.time(for (i in n) sqrt(x)) / length(n)
system.time(for (i in n) x ^ 0.5) / length(n)
```

   与函数 `microbenchmark()` 相比，函数 `system.time()` 给出的时间估计怎么样？它们为什么不同？

2. 下面是计算向量平方根的其他两种方式。你认为哪一个最快？哪一个最慢？应用微测试验证你的答案。

```
x ^ (1 / 2)
exp(log(x) / 2)
```

334

3. 根据基本算术运算符（+、−、×、/ 和 ^）的速度，应用微测试对这些基本算术运算符排序。对结果进行可视化。比较对整型和浮点数的算术运算的速度。
4. 微测试的结果是以参数 `unit` 给出的单位来表示的，可以改变该参数给出的单位。应用参数 `unit = "eps"` 来显示 1 秒可以进行运算的次数。应用 `eps` 单位重新运行上面的微测试。

16.3 语言性能

本节探索制约 R 语言性能的 3 个方面的折衷：极端动态性、可变环境的名字查找和函数参数的惰性求值。我将用微测试来说明每一方面折衷，说明它如何使 GNU-R 变慢。因为你不能测试 R 语言（它不能运行代码），所以我测试 GNU-R。这意味着这些测试结果仅仅是这些设计决策的成本提示，但它们是有用的。我选取这 3 个例子来说明这些折衷，它们是语言设计的关键：设计者必须平衡速度、灵活性和易于实现。

如果你想学习更多的 R 语言性能特征，了解它们如何影响真实代码，建议你阅读 FlorealMorandat、Brandon Hill、Leo Osvald 和 JanVitek 撰写的“Evaluating the Design of the R Language”（评估 R 语言的设计）（http://www.cs.purdue.edu/homes/jv/pubs/ecoop12.pdf）该文采用了一个强力的方法，它结合了改进的 R 解释器和大范围内找到的大量代码。

16.3.1 极端动态性

R 是一个极端动态的编程语言，几乎任何东西创建后都可以进行修改。仅举几个例子，你可以：

- 改变函数体、参数和函数的环境。
- 改变一个泛型函数的 S4 方法。

- 为S3对象添加新的字段，或者甚至改变它的类。
- 用<<-修改本地环境外的对象。

335

你不能够修改的东西差不多就是封装在名字空间中的对象，它在加载包时创建。

动态性的好处是，你只需要最少的提前计划。任何时候你都可以改变主意，不用重新开始就可以在答案上重复你的方式。动态性的缺点是，对于函数调用，它很难准确预测将发生什么。这是一个问题，因为对解释器和编译器而言，越容易预测将要发生什么，就越容易进行优化。(如果需要了解更多细节，可以参见Charles Nutter的“On Languages, VMs, Optimization, and the Way of the World”(http://blog.headius.com/2013/05/on-languages-vms-optimzation-and-way.html))。如果解释器不能预测什么将要发生，在它找到正确的结果前，它必须考虑许多选项。例如，下面的循环在R中是很慢的，因为R不知道x总是一个整数。这意味着，R必须在循环的每一次遍历中查找合适的+方法(即，它是浮点数加法还是整型加法)。

```
x <- 0L
for (i in 1:1e6) {
  x <- x + 1
}
```

为非原始函数找到合适方法的代价很高。下面的微测试说明了S3、S4和RC方法调度的代价。我为每一个OO系统创建了一个泛型函数和一个方法，然后调用泛型函数并观察找到和调用该方法所花费的时间。为了比较，我也对调用空函数进行计时。

```
f <- function(x) NULL

s3 <- function(x) UseMethod("s3")
s3.integer <- f

A <- setClass("A", representation(a = "list"))
setGeneric("s4", function(x) standardGeneric("s4"))
setMethod(s4, "A", f)

B <- setRefClass("B", methods = list(rc = f))

a <- A()
b <- B$new()
microbenchmark(
  fun = f(),
  S3 = s3(1L),
  S4 = s4(a),
  RC = b$rc()
)
#> Unit: nanoseconds
#>  expr    min     lq median     uq     max neval
#>   fun    155    201    242    300   1,670   100
#>    S3  1,960  2,460  2,790  3,150  32,900   100
#>    S4 10,000 11,800 12,500 13,500  19,800   100
#>    RC  9,650 10,600 11,200 11,700 568,000   100
```

在我的计算机上，空函数耗时200纳秒。S3方法调度需要额外的2 000纳秒，S4方法调度需要额外的11 000纳秒；RC方法调度需要额外的10 000纳秒。S3和S4方法耗时多的原因是，每次调用泛型函数时，R必须寻找合适的方法。在这次和上次调用之间它已经变化。通过缓存调用之间的方法，R可能更好一些，但是很难正确地进行缓存且缓存是漏洞的恶名

昭著的源头。

16.3.2 可变环境下的名字搜索

在 R 语言中，找到与一个名字关联的值令人吃惊地困难。这是由于词法作用域和极端动态性造成的。考虑下面的例子。每一次输出一个来自不同环境的字符 a。

```
a <- 1
f <- function() {
  g <- function() {
    print(a)
    assign("a", 2, envir = parent.frame())
    print(a)
    a <- 3
    print(a)
  }
  g()
}
f()
#> [1] 1
#> [1] 2
#> [1] 3
```

这意味着你不能一次完成名字查找：每一次都必须从头开始。由于几乎每一个运算都是词法作用域的函数调用，所以这个问题就更加严重了。你可能认为下面的简单函数调用 2 个函数：+ 和 ^。事实上，它们调用 4 个函数，因为 { 和 (是 R 中的正常函数。

```
f <- function(x, y) {
  (x + y) ^ 2
}
```

由于这些函数在全局环境中，所以 R 必须查找搜索路径中的每个环境，这样的环境可以轻易地达到 10 个或者 20 个。下面的微测试提示了性能成本。我们创建了 4 个版本的 f()，每一个函数 f() 在它的环境与定义 +、^、(和 { 的基础环境之间都还有一个环境（包含 26 个绑定）。

```
random_env <- function(parent = globalenv()) {
  letter_list <- setNames(as.list(runif(26)), LETTERS)
  list2env(letter_list, envir = new.env(parent = parent))
}
set_env <- function(f, e) {
  environment(f) <- e
  f
}
f2 <- set_env(f, random_env())
f3 <- set_env(f, random_env(environment(f2)))
f4 <- set_env(f, random_env(environment(f3)))

microbenchmark(
  f(1, 2),
  f2(1, 2),
  f3(1, 2),
  f4(1, 2),
  times = 10000
)
```

```
#> Unit: nanoseconds
#>       expr min  lq median  uq     max neval
#>   f(1, 2) 591 643    730 876 711,000 10000
#>  f2(1, 2) 616 690    778 920  56,700 10000
#>  f3(1, 2) 666 722    808 958  32,600 10000
#>  f4(1, 2) 690 762    850 995 846,000 10000
```

在 f() 和基础环境之间的每一个额外环境都使函数慢了 30 纳秒。

也许可以实现一个缓存系统，这样 R 只需要查看每一个名字的值一次。这是很难的，因为改变与名字相关联的值的方法有许多：<<-、assign()、eval() 等。任何缓存系统都必须了解这些函数，以便确保缓存正确地失效并且不会得到过期的值。

另一个简单的修正是增加更多的你不能重写的内置常量。例如，这就意味着 R 总是准确地知道 +、^、(和 { 的含义，你不必重复地查找它们的定义。这就使解释器更加复杂（因为有更多的特殊情况）且难以维护，同时语言也缺少灵活性。这会改变 R 语言，但是由于重写像 { 和 (这样的函数是很不好的想法，所以这不太可能过多地影响现有的代码。

16.3.3 惰性求值开销

在 R 中，函数参数惰性地求值（参见 6.4.4 节和 13.1 节）。为了实现惰性求值，R 使用一个约定对象，它包含计算结果需要的表达式和执行计算的环境。创建这些对象有一些开销，因此函数每附加一个参数都会降低一些运行速度。

下面的微测试比较了一个简单函数的运行时间。每一个版本的函数有一个额外参数。这说明增加一个额外参数使得函数大约慢 20 纳秒。

```
f0 <- function() NULL
f1 <- function(a = 1) NULL
f2 <- function(a = 1, b = 1) NULL
f3 <- function(a = 1, b = 2, c = 3) NULL
f4 <- function(a = 1, b = 2, c = 4, d = 4) NULL
f5 <- function(a = 1, b = 2, c = 4, d = 4, e = 5) NULL
microbenchmark(f0(), f1(), f2(), f3(), f4(), f5(), times = 10000)
#> Unit: nanoseconds
#>  expr min  lq median  uq     max neval
#>  f0() 129 149    153 220   8,830 10000
#>  f1() 153 174    181 290  19,800 10000
#>  f2() 170 196    214 367  30,400 10000
#>  f3() 195 216    258 454   7,520 10000
#>  f4() 206 237    324 534  59,400 10000
#>  f5() 219 256    372 589 865,000 10000
```

在大多数的其他编程语言中，增加额外参数有一些开销。许多编译语言甚至警告你，如果参数从未使用（如上例），则自动从函数中移除它。

16.3.4 练习

1. scan() 是基础函数中参数个数最多的（21 个）。每次调用 scan 时，21 个约定大约要花费多长时间？给定一个简单输入（例如，scan(text = "1 2 3",queit = T)），总的运行时间中有多大比例是用来创建这些约定的？
2. 阅读“Evaluating the Design of the R Language”（评估 R 语言的设计）（https://www.cs.purdue.edu/homes/jv/pubs/ecoop12.pdf）。R 语言的哪些方面使得它变慢？构建微测试来说明。

3. 随着类向量长度的变化，S3 方法调度的性能如何变化？随着超类个数的变化，S4 方法调度的性能如何变化？RC 类呢？

336 ~ 340

4. S4 方法调度的多重继承和多重调度的成本是多少？
5. 为什么在基础包中查找函数名字的成本较低？

16.4 实现的性能

R 语言的设计限制了它的最大理论性能，但是 GNU-R 目前还没有接近这个最大值。为了提高性能有许多事情可以（且将要）做。本节讨论 GNU-R 慢的某些方面，不是因为它们的定义，而是因为它们的实现。

R 有 20 多年的历史。它有近 80 万行代码（大约 45% 是 C 代码，19% 为 R 代码，17% 为 FORTRAN 代码）。只有 R 核心组（简称 R 核心）的成员才能修改基础 R。目前 R 核心有 20 个成员（http://www.r-project.org/contributors.html），但是只有 6 位活跃在日常的开发中。没有一位 R 核心成员全职工作在 R 上。大多数人是统计学教授，他们仅仅花费相对较少的时间在 R 上。由于必须特别小心避免破坏已有的代码，所以 R 核心在接受新代码上倾向于十分保守。看到 R 核心拒绝可以提高 R 性能的提议是令人沮丧的。然而，R 核心的关注点不是使得 R 变快，而是构建一个数据分析和统计的稳健平台。

下面我给出 2 个小的但具有说明性的 R 的例子，它们目前较慢，经过一些努力可以变快。它们不是基础 R 的关键部分，然而，过去它们一直是麻烦我的原因。与所有的微测试一样，它们不会影响大部分 R 代码的性能，但是它对某些特殊情形很重要。

16.4.1 从数据框提取单一值

下面的微测试说明访问内置 `mtcars` 数据集中的单一值（右下角的数值）的 7 种方法。性能的变化是惊人的：最慢方法花费的时间是最快方法的 30 倍。不应该有如此大的性能差异。这里的原因很简单，没有人有时间修复它。

```
microbenchmark(
  "[32, 11]"      = mtcars[32, 11],
  "$carb[32]"     = mtcars$carb[32],
  "[[c(11, 32)]]" = mtcars[[c(11, 32)]],
  "[[11]][32]"    = mtcars[[11]][32],
  ".subset2"      = .subset2(mtcars, 11)[32]
)
#> Unit: nanoseconds
#>           expr    min     lq median     uq     max neval
#>       [32, 11] 17,300 18,300 18,900 20,000  50,700   100
#>     $carb[32]  9,300 10,400 10,800 11,500 389,000   100
#>  [[c(11, 32)]] 7,350  8,460  8,970  9,640  19,300   100
#>     [[11]][32] 7,110  8,010  8,600  9,160  25,100   100
#>       .subset2   253    398    434    472   2,010   100
```

16.4.2 ifelse()、pmin() 和 pmax()

众所周知，有些基础函数较慢。例如，考虑下面 `squish()` 函数的 3 种实现方法，该函数确保一个向量的最小值至少为 `a`，最大值至多为 `b`。第一个实现 `squish_ife()` 使用 `ifelse()`。由于 `ifelse()` 是相对通用的且必须对全部参数进行求值，所以它是较慢的。

第二个实现 `squish_p()` 使用 `pmin()` 和 `pmax()`。因为这两个函数是专用的，所以有人可能期盼它们比较快。然而，它们事实上很慢。这是因为它们可以接受任意数量的参数，它们必须进行相对复杂的检查来确定采用哪一个方法。最后一个实现使用基础赋值。

```
squish_ife <- function(x, a, b) {
  ifelse(x <= a, a, ifelse(x >= b, b, x))
}
squish_p <- function(x, a, b) {
  pmax(pmin(x, b), a)
}
squish_in_place <- function(x, a, b) {
  x[x <= a] <- a
  x[x >= b] <- b
  x
}

x <- runif(100, -1.5, 1.5)
microbenchmark(
  squish_ife      = squish_ife(x, -1, 1),
  squish_p        = squish_p(x, -1, 1),
  squish_in_place = squish_in_place(x, -1, 1),
  unit = "us"
)
#> Unit: microseconds
#>            expr  min    lq median   uq   max neval
#>       squish_ife 78.8 83.90   85.1 87.0 151.0   100
#>         squish_p 18.8 21.50   22.5 24.6 426.0   100
#>  squish_in_place  7.2  8.81   10.3 10.9  64.6   100
```

应用 `pmin()` 和 `pmax()` 比应用 `ifelse()` 大约快 3 倍，而直接应用子集选取又大约快 2 倍。我们应用 C++ 通常可以做得更好。下面的例子对最好的 R 实现和一个尽管啰嗦但相对简单的 C++ 实现进行比较。即使你从未应用过 C++，你也应该能够了解基本策略：对向量的每一个元素进行循环，根据值是否小于 `a` 和大于 `b` 采用不同的行动。采用 C++ 的实现比最好的纯 R 实现快大约 3 倍。

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector squish_cpp(NumericVector x, double a, double b) {
  int n = x.length();
  NumericVector out(n);

  for (int i = 0; i < n; ++i) {
    double xi = x[i];
    if (xi < a) {
      out[i] = a;
    } else if (xi > b) {
      out[i] = b;
    } else {
      out[i] = xi;
    }
  }

  return out;
}
```

（你将在第 19 章中学习如何在 R 中访问 C++ 的代码。）

```
microbenchmark(
  squish_in_place = squish_in_place(x, -1, 1),
  squish_cpp      = squish_cpp(x, -1, 1),
  unit = "us"
)
#> Unit: microseconds
#>             expr  min   lq median    uq  max neval
#>  squish_in_place 7.45 8.33   9.82 10.30 43.8   100
#>       squish_cpp 2.49 2.98   3.27  3.47 33.7   100
```

16.4.3 练习

1. `squish_ife()`、`squish_p()` 和 `squish_in_place()` 的性能特征随着 `x` 的大小不同会有相当大的变化。解释这些不同。什么大小的 `x` 会导致最大的和最小的不同。
2. 比较从列表中提取一个元素、从矩阵中提取一列和从数据框中提取一列的性能成本。对提取一行的操作进行同样的比较。

16.5 其他的 R 实现

现在已有一些令人兴奋的 R 的新实现。它们都在尽可能地保持与现有语言定义接近的同时，通过应用一些现代解释器的设计理念来提高速度。4 个最成熟的开源代码项目是：

- pqR（http://www.pqr-project.org/），由 Radford Neal 开发。建立在 R 2.15.0 的基础之上，它修复了许多明显的性能问题，提供了更好的内存管理和某些自动多线程支持。
- Renjin（http://www.renjin.org/），由 BeDataDriven 开发。Renjin 应用 Java 虚拟机技术，它有丰富的测试套件（http://packages.renjin.org/）。
- FastR（https://github.com/allr/fastr），由普渡大学的一个团队开发。FastR 与 Renjin 相似，但是它做了更大的优化，它还不是很成熟。
- Riposte（http://github.com/jtalbot/riposte），由 JustinTalbot 和 Zachary DeVito 开发。Riposte 是实验性的且颇具野心。对于它已经实现的 R 部分，它特别快。在文章“Riposte: A Trace-Driven Compiler and Parallel VM for Vector Code in R”（http://www.justintalbot.com/wp-content/uploads/2012/10/pact080talbot.pdf）中对 Riposte 有详细的介绍。

340 ~ 344

上述系统大致按照最实用到最具野心的顺序排列。另一个由 Andrew Runnalls 开发的项目 CXXR（http://www.cs.kent.ac.uk/projects/cxxr/），没有提供任何性能提高。然而，它的目的是重新规划 R 的内部 C 代码，以便为未来的开发建立一个更强大的基础；保持系统的行为与 GNU-R 一致；建立更好、更丰富的内核文档。

R 是一个巨大的语言，还不明确是否上面提到的方法最终会成为主流。实现一个与 GNU-R 一样运行所有 R 代码的其他系统是一项艰难的任务。你能想象重新实现基础 R 中的每一个函数，它不仅更快，还要具有完全一样的已记载的漏洞？即使这些实现对应用 GNU-R 没有丝毫的影响，它们还是给出了好处：

- 更简单的实现使得在将新方法移植到 GNU-R 前，对这些方法的验证变得简单。
- 知道语言哪些方面的变化对现有代码影响最小而对性能影响最大，这可以引导我们

应该向哪一方面倾注注意力。

- 其他的实现给 R 核心施加压力，以便包含性能提高。

pqR、Renjin、FastR 和 Riposte 系统都在探索的一个重要方法是延迟求值。与 Riposte 的作者 JustinTalbot 指出的一样："对于长向量，R 的执行完全在内存中。它花费几乎所有的时间将中间向量读 / 写到内存中。"如果能删除这些中间向量，就可以提高性能并减少内存使用。

下面的例子说明延迟求值是如何有用的简单例子。有 3 个向量 x、y、z，每个向量包含 1 100 万个元素，我们想在 z 为 TRUE 时求向量 x + y 的和。（这代表了一个常见的数据分析问题的简化情况。）

345

```
x <- runif(1e6)
y <- runif(1e6)
z <- sample(c(T, F), 1e6, rep = TRUE)

sum((x + y)[z])
```

在 R 中，这创建了 2 个大的临时向量：x + y，包含 100 万个元素；(x + y)[z]，大约有 50 万个元素。这意味着需要有多余的内存用于中间计算，并且必须在 CPU 和内存之间来回传送数据。

如果 CPU 总是在等待更多的数据进来，那么由于 CPU 不能以最大效率工作所以导致计算变慢。

然而，如果使用像 C++ 这样的语言通过一个循环来重新编写上述函数，那么仅仅需要一个中间值：我们看到的所有值的和。

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double cond_sum_cpp(NumericVector x, NumericVector y,
                    LogicalVector z) {
  double sum = 0;
  int n = x.length();

  for(int i = 0; i < n; i++) {
    if (!z[i]) continue;
    sum += x[i] + y[i];
  }

  return sum;
}
```

在我的计算机上，这种方法比向量化的 R 等价函数快 8 倍，这已经足够快了。

```
cond_sum_r <- function(x, y, z) {
  sum((x + y)[z])
}

microbenchmark(
  cond_sum_cpp(x, y, z),
  cond_sum_r(x, y, z),
```

```
  unit = "ms"
)
#> Unit: milliseconds
#>                  expr   min    lq median    uq   max neval
#>  cond_sum_cpp(x, y, z)  4.09  4.11   4.13  4.15  4.33   100
#>    cond_sum_r(x, y, z) 30.60 31.60  31.70 31.80 64.60   100
```

延迟求值的目的是自动执行这种变换，因此可以写出精简的R代码并且使它自动变换为高效的机器码。复杂的翻译器可以知道如何尽量利用多核。在上面的例子中，如果有4个核，那么你可以把x、y和z分成4段，在每个核上执行条件求和，然后把4个结果加到一起。延迟求值和还可以与for循环联合运行，自动寻找可以向量化的运算。

346 ~ 348

本章讨论了R慢的基本原因。当R慢时，下面的几章给出了可以应用于代码中的一些工具。

第 17 章
代码优化

“程序员经常在一些非关键性的代码上花费巨大的时间和精力来研究它们的执行速度（运行效率），但是这种为提高效率而做出的努力常常会对代码调试和代码维护带来负面影响。”

——Donald Knuth

通过优化代码使程序变快是一个迭代过程：

1）找出最大的瓶颈（代码的最慢部分）。

2）尝试删除它（就算不成功也没有问题）。

3）重复上面的过程，直到代码变得足够快。

说起来容易，但做起来就难了。

即使非常有经验的程序员要找出程序的瓶颈都是很困难的。我们不能依靠直觉，而应该对代码进行性能分析：使用真实输入，测量程序中每一部分的运行时间。只有当我们能够找到最重要的瓶颈时，我们才能想办法消除它。对于提高性能来说，很难提出一个通用的建议，但通常我会尽最大努力使用 6 种技术来提高性能，这 6 种技术通常适用于多种情况。对于性能优化来说，我还提出一个通用策略，它可以保证优化后的代码仍然是正确的代码。

非常容易陷入尝试删除所有瓶颈这一困境。没有这个必要！我们的时间是很宝贵的，最好把时间用来分析程序而不是消除所有可能的低效率代码。要务实一点：没有必要花费几个小时的时间来为计算机节约几秒。这一点非常重要，为了不浪费宝贵的时间，我们应该为代码的运行设定一个目标时间。只要能够实现这个目标就没有必要再进行优化了。这也就意味着没有必要消除所有的瓶颈，因为有些小问题不会影响目标的实现。还有一些我们不得不放弃和接受，因为目前没有更好的方法了。对于所有这些可能性我们要能够接受并及时进入下一步，找寻下一段需要优化的代码。

主要内容

- 17.1 节介绍如何使用逐行性能分析寻找代码中的瓶颈。
- 17.2 节介绍提高代码性能的 7 个通用策略。
- 17.3 节学习如何更好地组织代码，使代码优化尽可能简单、漏洞更少。
- 17.4 节提醒你寻找一些已经有的解决方案。

- 17.5 节强调“懒惰”的重要性：使函数变快的最简单方法就是让它少做一点儿工作。
- 17.6 节简单地定义向量化，并说明如何充分利用内置函数。
- 17.7 节讨论复制数据的性能风险。
- 17.8 节说明如何利用 R 的字节码编译器。
- 17.9 节将上面的知识结合在一起，并使用这些知识将重复运行的 t-tests 程序的性能提高大约 1 000 倍。
- 17.10 节学习如何使用并行化来充分利用计算机的所有核来加速运算。
- 17.11 节提供更多的资源来帮助你写出更加高效的代码。

预备条件

本章将使用 `lineprof` 添加包来查看 R 代码的性能。从 devtools::install_github("hadley/lineprof") 上得到该添加包。

17.1 性能测试

为了了解性能，我们需要使用一个性能分析器（profiler）。有很多不同类型的性能分析器。
350 R 使用一种非常简单的类型，称为采样或统计性能分析器。采样性能分析器每隔几毫秒中断程序的运行，并记录当前正在执行的函数（以及调用此函数的函数等）。例如，考虑下面的 `f()`：

```
library(lineprof)
f <- function() {
  pause(0.1)
  g()
  h()
}
g <- function() {
  pause(0.1)
  h()
}
h <- function() {
  pause(0.1)
}
```

（这里我使用了 `pause()` 而不是 `Sys.sleep()`，这是因为，就 R 能告诉我们的而言，`Sys.sleep()` 运行时不占用任何运算时间，因此它也不会出现在性能分析的输出结果中。）

如果对 `f()` 进行性能分析，每 0.1 秒中断一次程序的执行，我们将看到如下的结果。每行代表性能分析器的一次中断的结果（此例中为 0.1 秒），函数的调用关系用 > 表示。结果表明，这段代码花费 0.1 秒运行 `f()`，花费 0.2 秒运行 `g()`，然后 0.1 秒运行 `h()`。

```
f()
f() > g()
f() > g() > h()
f() > h()
```

如果我们使用下面的代码对 `f()` 进行性能分析，不可能得到如此清晰的结果。

```
tmp <- tempfile()
Rprof(tmp, interval = 0.1)
f()
351 Rprof(NULL)
```

这是因为性能分析很难做到在不使代码运行变慢（可以变慢几个数量级）的情况下给出

准确的结果。Rprof() 做出的妥协是使用采样，此时性能分析只对整体性能产生很小的影响，但基对性能的影响基本上是随机的。计时器给出的时间与每一次运行的实际时间的精度都有所不同，所以你每一次进行性能分析的结果并不是完全一致的。幸运的是，不需要非常精确地知道最慢的那段代码。

除了关注单个调用外，我们还可以使用 lineprof 添加包来对性能分析的汇总结果进行可视化。有许多其他选择，如 summaryRprof()、proftools 添加包和 profr 添加包，但这些工具都超出了本书的范围。我编写 lineprof 添加包的目的就是为可视化性能分析数据提供一种简单方法。与它的名字一样，在 lineprof() 中分析的基本单位是一行代码。这也使 lineprof 的结果没有其他替代软件的分析结果的准确（因为一行代码可以包含多个函数调用），但这也使我们理解整个代码的上下文变得更加容易。

为了使用 lineprof，首先将代码保存在一个文件中并使用 source() 加载它。profiling-example.R 包含 f()、g() 和 h() 的定义。注意必须使用 source() 来加载代码。这是因为 lineprof 使用 srcrefs 将代码与性能分析进行匹配，当从硬盘加载代码时才能创建所需的 srcrefs。然后使用 lineprof() 运行函数并捕获定时输出。输出这个结果对象会显示一些基本信息。从现在开始我们只关注时间列，它估计每一行代码的运行时间；以及 ref 列，它表明运行了哪行代码。估计并不精确，但是比例大概是正确的。

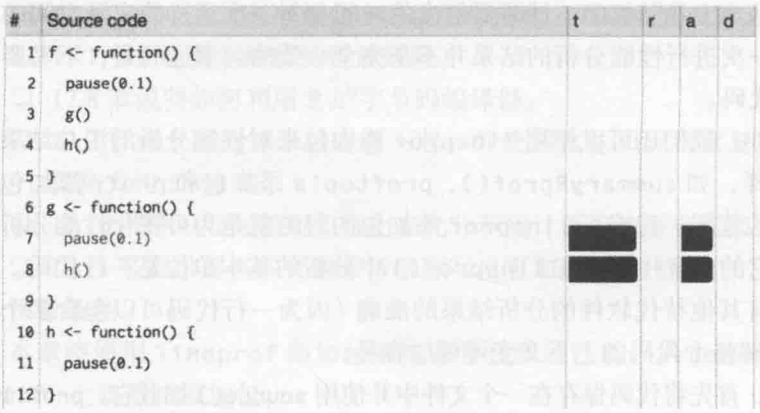
```
library(lineprof)
source("profiling-example.R")
l <- lineprof(f())
l
#>   time alloc release dups      ref     src
#> 1 0.074 0.001       0    0 profiling.R#2 f/pause
#> 2 0.143 0.002       0    0 profiling.R#3 f/g
#> 3 0.071 0.000       0    0 profiling.R#4 f/h
```

lineprof 提供了一些函数来帮助我们查看结果，但是这些函数有点儿笨拙。可以使用 shiny 包打开一个交互式浏览器。shine(l) 将打开一个新网页（如果使用 RStudio，打开一个新面板），它显示源代码和每一行代码的运行时间。shine() 启动一个 shiny 应用，它“阻止”现有的 R 会话。为了退出，需要使用 ctrl + c 或者 escape 来终止这个进程。 352

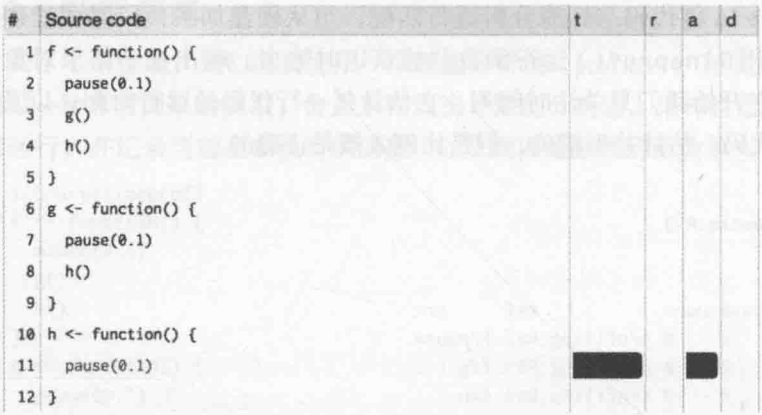
#	Source code	t	r	a	d
1	f <- function() {				
2	pause(0.1)				
3	g()				
4	h()				
5	}				
6	g <- function() {				
7	pause(0.1)				
8	h()				
9	}				
10	h <- function() {				
11	pause(0.1)				
12	}				

t 列显示每行代码运行所需的时间。（我们将在 18.3 节中学习其他列的内容。）虽然不够精确，但是它可以帮助我们找到程序的瓶颈，通过将鼠标悬停在每个条上可以得到精确的数字。它说明 g() 的运行时间是 h() 的两倍，所以应该进入 g() 来查看更详细的内容。为此，

只需单击 g()：



然后单击 h()：



这项技术可以帮助我们快速地定位代码中的主要瓶颈。

局限性

性能分析还有一些局限性：

- 性能分析不能扩展到 C 代码。如果 R 代码调用了 C/C++ 代码，你可以看到调用，但是看不到 C/C++ 代码中的调用函数。不幸的是，对编译代码的性能分析超出了本书的范围（换言之，我不知道应该怎么做）。
- 同样，我们看不到原函数以及字节码编译后代码的内部情况。
- 如果使用匿名函数进行大量的函数式编程，那么很难准确地找到哪个函数被调用了。解决这个问题的最简单方法就是为函数命名。
- 惰性计算意味着其参数经常在其他函数的内部进行求值。例如，在下面的代码中，性能分析看起来 i() 是被 j() 调用的，因为只有 j() 需要时才对参数进行求值。

```
i <- function() {
  pause(0.1)
  10
}
j <- function(x) {
  x + 10
}
j(i())
```

如果这里有点儿迷惑，可以创建临时变量以便提前将其计算出来。

17.2 改进性能

“我们应该忽略那些对提高代码效率贡献很小的（瓶颈）代码，大概 97% 的代码都属于这一种：不完善的代码优化是一切错误的源泉。而对于提高效率非常重要的 3% 的代码，则不应该放过。一个好的程序员不应该盲目提高代码的效率，一旦找到关键瓶颈，他应该愿意仔细地查看这段代码。”

——Donald Knuth

一旦找到了程序的瓶颈，就需要使它的运行速度变快。下面几节学习一些这方面的技术，这些技术通常适用于多种情况：

1）查找已有的解决方案。

2）减少工作量。

3）向量化。

4）并行化。

5）避免复制。

6）字节码编译。

最后一个技术是：使用其他更快的语言（如 C++）重写。这个话题比较大，将在第 19 章中详细介绍。

在学习具体技术之前，首先介绍一个通用策略和代码组织风格，这些知识对于提高程序性能很有帮助。

353 ~ 355

17.3 组织代码

当我们试图使代码运行得更快时，我们经常落入两个陷阱：

1）代码速度变快了，但代码是错的。

2）你认为快的代码但实际上并不快。

下面列出的策略可以帮助我们避免落入这两个陷阱。

在处理瓶颈时，通常有多种解决方案。为每一种方案编写一个函数，将所有相关的行为封装到这个函数中。这样就便于我们检查每一个方法是否返回了正确的结果，以及这个方法所需要的时间。下面演示这个策略，使用两种不同的方法来求平均值：

```
mean1 <- function(x) mean(x)
mean2 <- function(x) sum(x) / length(x)
```

对每一次尝试，无论结果成败都做一个记录。如果以后出现同样的问题，这将非常有帮助。为此，我经常使用 R Markdown，它允许我们将代码和详细的文档注释和说明放在一起。

接下来，创建一个有代表性的测试案例。这个案例要足够大以便可以帮助我们捕获问题的本质，也要足够小以便可以在几秒内运行完。没有人希望它运行很长时间，因为我们需要运行很多次来比较不同方法的优劣。另一方面，这个案例又不能太小，因为结果可能不能反映真实的问题。

使用这个测试案例快速地检查所有方法是否返回相同的结果。一个简单方法是使用

stopifnot() 和 all.equal()。对于现实中具有多种不同输出的问题，可能需要做更多的测试，确保不是偶然返回正确的结果。这与求平均值的问题不一样。

```
x <- runif(100)
stopifnot(all.equal(mean1(x), mean2(x)))
```

356

最后，使用 microbenchmark 包来比较每一种解决方案所花费的时间。对于较大的问题，可以减少 times 参数，这样就可以在几秒内运行完。主要看中位数时间，并使用上下四分位数来评估这次测试的波动性。

```
microbenchmark(
  mean1(x),
  mean2(x)
)
#> Unit: nanoseconds
#>       expr   min    lq median    uq    max neval
#>  mean1(x) 5,030 5,260  5,400 5,660 47,100   100
#>  mean2(x)   709   808  1,020 1,140 35,900   100
```

（你可能对结果有些惊讶：mean(x) 竟然比 sum(x)/length(x) 慢。在众多的原因中，其中一个原因是 mean(x) 对向量进行两轮校验以便使数值更准确。

在开始实验前，我们应该为自己定下一个目标，当程序达到某一速度时，就认为这个瓶颈解决了。设置目标非常重要，因为我们并不想将宝贵的时间浪费在代码的过度优化上。

如果你想看看这一策略在实际工作中的应用，我在 stackoverflow 中已经使用它多次了：

- http://stackoverflow.com/questions/22515525#22518603
- http://stackoverflow.com/questions/22515175#22515856
- http://stackoverflow.com/questions/3476015#22511936

17.4 有人已经解决了这个问题吗

当我们组织好自己的代码并把所有能够想到的方法全部实现后，很自然应该看看其他人是如何解决这个问题的。我们只是一个非常大的社区的一部分，非常有可能其他人也遇到过同样的问题。如果你遇到的瓶颈是一个添加包中的一个函数，你可以在其他添加包寻找有没有完成同样任务的函数，这将非常有帮助。可以从下面两个地方开始：

357

- CRAN task views（http://cran.rstudio.com/web/views/）如果有一个 CRAN 任务视图与你的问题有关，就非常值得你看看这里的包列表。
- CRAN 页面中列出的 Rcpp 的反向依赖关系（http://cran.r-project.org/web/packages/Rcpp）。由于这些包使用 C++ 编写，所以可能找到一种使用高性能语言来解决你遇到的瓶颈的方法。

另一个挑战是如何描述遇到的问题以便帮助我们找到相关的问题和答案。知道问题的名字或者同义词可以使搜索变得更加容易。但如果不知道它叫什么，就很难进行搜索了！通过学习大量的统计和算法方面的知识，长时间的积累可以帮助我们建立自己的知识库。或者可以询问其他人。与同事进行交流，进行头脑风暴找出一些可能的名字，然后在谷歌和 stackoverflow 上搜索。将搜索限制在与 R 有关的页面经常会很有帮助。对于谷歌，可以尝试 rseek（http://www.rseek.org/）。对于 stackoverflow，可以使用 R 标签 [R] 来限制搜索的结果。

如上所述，记录我们能够找到的所有解决方法，而不仅仅是那些运行速度快的。有些解

决方法可能刚开始时比较慢，但由于它们更容易优化，所以最终会变得很快。还可以将不同方法中最快的部分结合起来。如果你已经找到一种足够快的解决方案，那么就恭喜你！如果合适，你可以把你的解决方案分享给 R 社区。如果没有，那就继续找。

练习

1. 比 `lm` 更快的可替代函数是什么？哪些用于专门处理比较大的数据集？
2. 哪个包实现了可以更快速地进行重复查找的 `match()` 函数？它到底有多快？
3. 列出 4 个可以将字符串转换成时间对象的函数。它们的优缺点是什么？
4. R 中有多少种方法可以帮我们计算一维密度估计？
5. 哪个包提供了计算移动平均的功能？
6. `optim()` 的替代函数是什么？

358

17.5 尽可能少做

让函数变快的最简单方法就是让它做尽可能少的任务。做到这一点的一个方法就是为更具体类型的输入或输出，或更具体的问题定制一个函数。例如：

- `rowSums()`、`colSums()`、`rowMeans()` 和 `colMeans()` 比使用 apply() 的等价调用快，因为它们是向量化的（这是下一节的主题）。
- `vapply()` 比 `sapply()` 快，因为它预前设定了输出类型。
- 如果你想查看一个向量是否包含一个值，`any(x == 10)` 比 `10 %in% x` 快很多。这是因为测试等式比测试包含关系更简单。

要想对这些知识了如指掌，就需要知道有其他相同功能的函数存在：必须有良好的词汇（知道很多函数）。从第 4 章开始，通过经常阅读 R 代码来扩充词汇量。阅读代码的好地方是 R 帮助邮件列表（http://stat.ethz.ch/mailman/listinfo/r-help）和 stackoverflow（http://stackoverflow.com/questions/tagged/r）。

有些函数强制要求它的输入为特定的类型。如果输入的类型不对，函数就必须要做更多额外的工作。不应该让函数适应（不同类型的）数据，而应该在存储数据时多做一些考虑，从而便于处理。这个问题最典型的例子就是在数据框数据上使用 `apply()` 函数。`apply()` 总是将它的输入转变成矩阵。这样做不仅容易出错，而且还会使程序变慢。

如果能够提供更多与问题相关的信息，其他函数也可以减少工作量。认真阅读文档并使用不同的参数进行试验总是非常有用的。下面是我以前发现的一些例子：

- `read.csv()`：使用 `colClasses` 为已知列设置数据类型。
- `factor()`：使用 `levels` 设定因子的已知水平。

359

- `cut()`：如果不需要标签，可以使用 `labels = FALSE` 不产生标签，或者更好地，根据文档中“see also”中的建议使用 `findInterval()`。
- `unlist(x, use.names = FALSE)` 比 `unlist(x)` 快很多。
- `interaction()`：如果只需要对数据中存在的对象进行组合，使用 `drop = TRUE`。

有时避免方法调度也可以使函数变快。如我们在 6.3.1 节中看到的，R 中方法调度是非常消耗资源的。如果在循环中调用一个方法，可以通过只进行一次方法查找来避免资源消耗：

- 对于 S3 系统，可以使用 `generic.class()` 而不使用 `generic()`。
- 对于 S4 系统，可以使用 `findMethod()` 来查找方法，将它保存在变量中，然后调用这个函数。

例如，对于小的向量，调用 `mean.default()` 比调用 `mean()` 要快一些：

```
x <- runif(1e2)

microbenchmark(
  mean(x),
  mean.default(x)
)
#> Unit: microseconds
#>             expr  min   lq median   uq   max neval
#>          mean(x) 4.38 4.56   4.70 4.89 43.70   100
#>  mean.default(x) 1.32 1.44   1.52 1.66  6.92   100
```

虽然 `mean.default()` 几乎比 `mean()` 快两倍，但是这种优化是有危险的，如果 `x` 不是数值向量，它将以一种非常惊人的方式出错。只有当我们能够确切地知道 `x` 的类型时，我们才可以使用它。

如果我们知道要处理输入的具体类型，可以使用其他的方法写出更快的代码。例如，`as.data.frame()` 非常慢，它将每个元素强制转换成数据框，然后将它们 `rbind()` 到一起。如果有一个有命名列表，其中向量都是等长度，我们就可以直接将它转换成数据框。在这种情况下，如果我们能够对输入做出很强的假设，我们就可以写出比默认函数快 20 倍的函数。

360

```
quickdf <- function(l) {
  class(l) <- "data.frame"
  attr(l, "row.names") <- .set_row_names(length(l[[1]]))
  l
}

l <- lapply(1:26, function(i) runif(1e3))
names(l) <- letters

microbenchmark(
  quick_df      = quickdf(l),
  as.data.frame = as.data.frame(l)
)
#> Unit: microseconds
#>          expr     min      lq  median      uq      max neval
#>      quick_df    13.9    15.9    20.6    23.1    35.3   100
#>  as.data.frame 1,410.0 1,470.0 1,510.0 1,540.0 31,500.0   100
```

同样，注意代价。这种方法快，因为它有危险。如果给它错误的输入，就会得到错误的数据框：

```
quickdf(list(x = 1, y = 1:2))
#> Warning: corrupt data frame: columns will be truncated or
#> padded with NAs
#>   x y
#> 1 1 1
```

为了提出这个最短的方法，我仔细阅读并重写了 `as.data.frame.list()` 和 `data.`

`frame()` 的源代码。我做了一些小修改，每次我都做了检查，以保证我的修改不会破坏原有的行为。经过几个小时的工作，我分离出了上面这段最短的代码。这个技术非常有用。大多数基础 R 函数的编写都更加注重灵活性和功能而不是性能。因此，为了具体的需求重新编写源代码可能对程序的性能有显著的提高。为此，需要认真地阅读源代码。这可能有点儿复杂且令人困惑，但是不要放弃！

如果我们只想比较相邻值之间的不同，下面的例子说明如何对 `diff()` 进行逐步简化。在每一步，使用一个具体案例来替换一个参数，然后看看这个函数是否还能执行。原始函数很长、很复杂，但是通过限制参数，不仅使它比原来的函数快两倍，而且还使它变得更容易理解。 361

首先，我需要 `diff()` 的源代码并把它转换为我自己常用的风格：

```
diff1 <- function (x, lag = 1L, differences = 1L) {
  ismat <- is.matrix(x)
  xlen <- if (ismat) dim(x)[1L] else length(x)
  if (length(lag) > 1L || length(differences) > 1L ||
      lag < 1L || differences < 1L)
    stop("'lag' and 'differences' must be integers >= 1")

  if (lag * differences >= xlen) {
    return(x[0L])
  }

  r <- unclass(x)
  i1 <- -seq_len(lag)
  if (ismat) {
    for (i in seq_len(differences)) {
      r <- r[i1, , drop = FALSE] -
        r[-nrow(r):-(nrow(r) - lag + 1L), , drop = FALSE]
    }
  } else {
    for (i in seq_len(differences)) {
      r <- r[i1] - r[-length(r):-(length(r) - lag + 1L)]
    }
  }
  class(r) <- oldClass(x)
  r
}
```

接下来，假设输入是向量。这样就可以去除 `is.matrix()` 测试和使用矩阵进行子集选取的方法。

```
diff2 <- function (x, lag = 1L, differences = 1L) {
  xlen <- length(x)
  if (length(lag) > 1L || length(differences) > 1L ||
      lag < 1L || differences < 1L)
    stop("'lag' and 'differences' must be integers >= 1")

  if (lag * differences >= xlen) {
    return(x[0L])
  }
  i1 <- -seq_len(lag)
  for (i in seq_len(differences)) {
    x <- x[i1] - x[-length(x):-(length(x) - lag + 1L)]
  }
  x
```

```
}
diff2(cumsum(0:10))
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

接下来，假设 `difference = 1L`。这将简化输入检查并删除 `for` 循环：

```
diff3 <- function (x, lag = 1L) {
  xlen <- length(x)
  if (length(lag) > 1L || lag < 1L)
    stop("'lag' must be integer >= 1")

  if (lag >= xlen) {
    return(x[0L])
  }

  i1 <- -seq_len(lag)
  x[i1] - x[-length(x):-(length(x) - lag + 1L)]
}
diff3(cumsum(0:10))
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

最后，假设 `lag = 1L`。这将删除输入检查并简化子集选取。

```
diff4 <- function (x) {
  xlen <- length(x)
  if (xlen <= 1) return(x[0L])

  x[-1] - x[-xlen]
}
diff4(cumsum(0:10))
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

现在，`diff4()` 比 `diff1()` 简单很多且快很多：

```
x <- runif(100)
microbenchmark(
  diff1(x),
  diff2(x),
  diff3(x),
  diff4(x)
)
#> Unit: microseconds
#>       expr   min    lq median    uq   max neval
#>  diff1(x) 10.90 12.60  13.90 14.20 28.4   100
#>  diff2(x)  9.42 10.30  12.00 12.40 65.7   100
#>  diff3(x)  8.00  9.11  10.80 11.10 44.4   100
#>  diff4(x)  6.56  7.21   8.95  9.24 15.0   100
```

当学习了第 19 章后，针对这种特殊情况我们可以写出更快的代码。

最后一个例子是使用简单的数据结构。例如，当处理数据框的行数据时，处理行索引比处理数据框快很多。例如，计算数据框中两列数据的相关系数的自助法估计值，可以有两种方法：直接处理整个数据框；或者对两个单独向量进行计算。下面的例子说明处理向量比处理数据框的速度快两倍。

```
sample_rows <- function(df, i) sample.int(nrow(df), i,
  replace = TRUE)

# Generate a new data frame containing randomly selected rows
```

```
boot_cor1 <- function(df, i) {
  sub <- df[sample_rows(df, i), , drop = FALSE]
  cor(sub$x, sub$y)
}

# Generate new vectors from random rows
boot_cor2 <- function(df, i ) {
  idx <- sample_rows(df, i)
  cor(df$x[idx], df$y[idx])
}

df <- data.frame(x = runif(100), y = runif(100))
microbenchmark(
  boot_cor1(df, 10),
  boot_cor2(df, 10)
)
#> Unit: microseconds
#>               expr   min    lq median    uq max neval
#>  boot_cor1(df, 10) 123.0 132.0  137.0 149.0 665   100
#>  boot_cor2(df, 10)  74.7  78.5   80.2  86.1 109   100
```

练习

1. 如果计算 10 000 个而不是 100 个观测值的平均值，`mean()` 和 `mean.default()` 的结果变化是怎样的？
2. 下面是 `rowSums()` 的另一种实现。对于这样的输入，为什么它会更快？

```
rowSums2 <- function(df) {
  out <- df[[1L]]
  if (ncol(df) == 1) return(out)

  for (i in 2:ncol(df)) {
    out <- out + df[[i]]
  }
  out
}

df <- as.data.frame(
  replicate(1e3, sample(100, 1e4, replace = TRUE))
)
system.time(rowSums(df))
#>    user  system elapsed
#>   0.063   0.001   0.063
system.time(rowSums2(df))
#>    user  system elapsed
#>   0.039   0.009   0.049
```

3. `rowSums()` 和 `.rowSums()` 有什么不同？
4. 当输入是两个没有缺失值的数值向量时，编写一个只计算卡方检验统计量的更快的 `chisq.test()`。你可以对 `chisq.test()` 进行简化或者根据数学定义（http://en.wikipedia.org/wiki/Pearson%27s_chi-squared_test）来编程。
5. 输入为两个没有缺失值的整数向量，可以编写一个较快的 `table()` 吗？能用这个函数来加速卡方检验吗？
6. 假设你想使用 `cor_df()` 和下面例子中的数据来计算样本相关性的自助法分布。如果多次运行这个程序，能让这段代码运行得更快吗？（提示：这个函数有 3 个部分可以加速。）

362 ~ 365

```
n <- 1e6
df <- data.frame(a = rnorm(n), b = rnorm(n))

cor_df <- function(i) {
  i <- sample(seq(n), n * 0.01)
  cor(q[i, , drop = FALSE])[2,1]
}
```

有方法使这个过程向量化吗?

17.6 向量化

如果你使用 R 有一段时间了，有人可能会告诫你要“向量化你的代码”。这到底是什么意思呢？向量化代码不只是避免使用循环，虽然这是经常要做的。向量化是对解决的问题有一种“整体观”，以向量的方式思考，而不是标量。向量化的函数有两个关键属性：

- 它可以使很多问题变简单。我们只需要对整个向量进行思考，而不是针对向量中的每个元素。
- 向量化函数中的循环使用 C 语言编写。由于它们的开销很少，所以 C 语言编写的循环非常快。

第 11 章以更高层次的抽象强调向量化代码的重要性。向量化对于编写快速的 R 代码也是非常重要的。但这并不是简单地使用 `apply()`、`lapply()` 或 `Vectorise()` 就行了。这些函数只是改变了函数的接口而没有从根本上改善它的性能。使用向量化改善性能意味着找
366 到用 C 语言实现的现有 R 函数，并尽量使用它们来解决问题。

向量化函数可以解决多种性能瓶颈：

- `rowSums()`、`colSums()`、`rowMeans()` 和 `colMeans()`。这些向量化矩阵函数总是比使用 `apply()` 快。有时可以使用这些函数来构建其他向量化函数。

```
rowAny <- function(x) rowSums(x) > 0
rowAll <- function(x) rowSums(x) == ncol(x)
```

- 向量化子集选取可以大大地改善性能。记住查询表（3.4.1 节）以及人工比对与合并（3.4.2 节）所使用的技术。同样要记住，可以在一步中使用子集选取赋值来替换多个值。如果 `x` 是一个向量、矩阵或数据框，则 `x[is.na(x)] <- 0` 用 0 来替换所有的缺失值。
- 如果要从矩阵或数据框中的分散位置提取或替换值，可以使用整数矩阵来选取子集。详情请参阅 3.1.3 节。
- 如果要将连续值分类，确保知道如何使用 `cut()` 和 `findInterval()`。
- 知道一些向量化的函数，如 `cumsum()` 和 `diff()`。

矩阵代数是向量化的一般例子。其中的循环使用了极度优化的外部库，如 BLAS。如果能够找到一种方法使用矩阵代数来解决问题，那么非常可能找到一个快速的解决方法。使用矩阵代数解决问题的能力来源于经验。时间的磨练可以帮助我们练就这项本领，我们可以经常向有经验的人请教。

向量化的缺点是，很难预测某个运算的速度被提升了多少倍。下面的例子是测量从一个列表中取出 1、10、100 个元素所花费的时间。我们可能认为取出 10 个元素的的时间是取出 1 个元素的时间的 10 倍，取出 100 个元素的时间是取出 10 个元素的时间的 10 倍。实际上，

取出 100 个元素所用的时间仅仅是取出一个元素的时间的 9 倍。

```
lookup <- setNames(as.list(sample(100, 26)), letters)
x1 <- "j"
x10 <- sample(letters, 10)
x100 <- sample(letters, 100, replace = TRUE)

microbenchmark(
  lookup[x1],
  lookup[x10],
  lookup[x100]
)
#> Unit: nanoseconds
#>          expr   min    lq median    uq    max neval
#>    lookup[x1]   549   616    709   818  2,040   100
#>   lookup[x10] 1,580 1,680  1,840 2,090 34,900   100
#>  lookup[x100] 5,450 5,570  7,670 8,160 25,900   100
```

向量化并不能解决所有问题，我们不必要将一个已有算法用矢量化的方法来实现，而更应该使用 C++ 自己编写一个向量化的函数。我们将在第 19 章中学习这一点。

练习

1. 密度函数，如 `dnorm()`，有一个通用接口。哪些参数被向量化了？`rnorm(10, mean = 10:1)` 是做什么的？
2. 对于不同长度的 `x`，将 `apply(x, 1, sum)` 和 `rowSums(x)` 的速度进行比较。
3. 如何使用 `crossprod()` 计算加权和？它比 `sum(x * w)` 快多少？

17.7 避免复制

使 R 代码变慢的一个致命根源是使用循环不断使对象变大。当使用 `c()`、`append()`、`cbind()`、`rbind()` 或 `paste()` 创建一个更大的对象时，R 必须首先为这个新对象分配空间，然后将旧对象复制到这个新空间。如果重复操作多次，比如在一个 `for` 循环中，这将消耗巨大的资源。你已经进入“R inferno”（http://www.burns-stat.com/pages/Tutor/R_inferno.pdf）的 Circle 2。

367 ~ 368

这里是一个简单的例子，用来说明这个问题。首先，产生一些随机字符串，然后在一个循环中使用 `collapse()` 将它们组合在一起，或者就简单地使用一次 `paste()`。注意，随着字符串数目的增长，`collapse()` 的性能变得越来越糟糕：组合 100 个字符串所花费的时间是组合 10 个字符串所花费时间的 10 倍。

```
random_string <- function() {
  paste(sample(letters, 50, replace = TRUE), collapse = "")
}
strings10 <- replicate(10, random_string())
strings100 <- replicate(100, random_string())

collapse <- function(xs) {
  out <- ""
  for (x in xs) {
    out <- paste0(out, x)
  }
  out
}
```

```
}

microbenchmark(
  loop10  = collapse(strings10),
  loop100 = collapse(strings100),
  vec10   = paste(strings10, collapse = ""),
  vec100  = paste(strings100, collapse = "")
)
#> Unit: microseconds
#>     expr    min     lq median     uq     max neval
#>   loop10  22.50  24.70  25.80  27.70    69.6   100
#>  loop100 866.00 894.00 900.00 919.00 1,350.0   100
#>    vec10   5.67   6.13   6.62   7.33    40.7   100
#>   vec100  45.90  47.70  48.80  52.60    74.7   100
```

在循环中对一个对象进行修改，如 `x[i] <- y`，也可能造成复制问题，这取决于 x 的类。18.4 节对这个问题有比较深入的讨论，当需要复制时，这里还给出了一些工具来对 x 进行判定。

369

17.8 字节码编译

R 2.13.0 引入了字节码编译器，它可以使某些代码加速。使用这个编译器是改进性能的一种简单方法。由于使用起来很简单，不需要耗费大量的时间，因此就算它对你的函数的效果不明显也没有关系。下面例子说明了一个纯 R 语言版的 `lapply()`，它来自 11.1 节。它编译后可以有一个非常明显的速度提升，虽然它现在还是没有基础 R 中 C 语言版本的函数快。

```
lapply2 <- function(x, f, ...) {
  out <- vector("list", length(x))
  for (i in seq_along(x)) {
    out[[i]] <- f(x[[i]], ...)
  }
  out
}

lapply2_c <- compiler::cmpfun(lapply2)

x <- list(1:10, letters, c(F, T), NULL)
microbenchmark(
  lapply2(x, is.null),
  lapply2_c(x, is.null),
  lapply(x, is.null)
)
#> Unit: microseconds
#>                 expr  min   lq median   uq  max neval
#>   lapply2(x, is.null) 5.49 5.88   6.15 6.83 17.2   100
#> lapply2_c(x, is.null) 3.06 3.30   3.47 3.79 34.8   100
#>    lapply(x, is.null) 2.25 2.49   2.67 2.92 38.5   100
```

这里字节码编译的确发挥了作用，但是大多数情况下它只能使性能提升 5%～10%。所有基础 R 函数默认都是编译过的。

370

17.9 案例研究：*t* 检验

下面的案例学习说明如何使用上面描述的技术使 *t* 检验变得更快。它基于 Holger

Schwender 和 Tina Müller 的“Computing thousands of test statistics simultaneously in R”（http://statcomputing.org/newsletter/issues/scgn-18-1.pdf）中的例子。我强烈建议阅读全文，看看怎样把同样的想法应用到其他检验中。

假设我们已经做了 1000 次实验（行），每次实验有 50 个数据（列）。每次实验的前 25 个结果归入第一组，剩下的归入第二组。首先，产生一些随机数来描述这个问题：

```
m <- 1000
n <- 50
X <- matrix(rnorm(m * n, mean = 10, sd = 3), nrow = m)
grp <- rep(1:2, each = n / 2)
```

对于这种形式的数据，有两种使用 `t.test()` 的方法。既可以使用公式接口，也可以提供两个向量，每个组一个。计时显示公式接口相当慢。

```
system.time(for(i in 1:m) t.test(X[i, ] ~ grp)$stat)
#>    user  system elapsed
#>   0.975   0.005   0.980
system.time(
  for(i in 1:m) t.test(X[i, grp == 1], X[i, grp == 2])$stat
)
#>    user  system elapsed
#>   0.210   0.001   0.211
```

当然，`for` 循环计算，但不保存值。可以使用 `apply()` 实现，但这会增加一点儿开销：

```
compT <- function(x, grp){
  t.test(x[grp == 1], x[grp == 2])$stat
}
system.time(t1 <- apply(X, 1, compT, grp = grp))
#>    user  system elapsed
#>   0.223   0.001   0.224
```

如何能使它变得快一点？首先，可以让它做的少一点。如果查看 `stats:::t.test.default()` 的源代码，就会发现它不只是进行 t 检验。它还计算 p 值并格式化输出的格式。可以通过剥离这些功能来为它加速。

```
my_t <- function(x, grp) {
  t_stat <- function(x) {
    m <- mean(x)
    n <- length(x)
    var <- sum((x - m) ^ 2) / (n - 1)

    list(m = m, n = n, var = var)
  }

  g1 <- t_stat(x[grp == 1])
  g2 <- t_stat(x[grp == 2])

  se_total <- sqrt(g1$var / g1$n + g2$var / g2$n)
  (g1$m - g2$m) / se_total
}
system.time(t2 <- apply(X, 1, my_t, grp = grp))
#>    user  system elapsed
#>   0.035   0.000   0.036
stopifnot(all.equal(t1, t2))
```

现在速度已经提升了6倍。

既然有了一个相对简单的函数，可以通过向量化它来让它变得更快。可以对 t_stat() 进行修改，使它可以处理矩阵数据。因此，mean() 变成 rowMeans()、length() 变成 ncol()、sum() 变成 rowSums()。其余代码保持不变。

```
rowtstat <- function(X, grp){
  t_stat <- function(X) {
    m <- rowMeans(X)
    n <- ncol(X)
    var <- rowSums((X - m) ^ 2) / (n - 1)

    list(m = m, n = n, var = var)
  }
  g1 <- t_stat(X[, grp == 1])
  g2 <- t_stat(X[, grp == 2])

  se_total <- sqrt(g1$var / g1$n + g2$var / g2$n)
  (g1$m - g2$m) / se_total
}
system.time(t3 <- rowtstat(X, grp))
#>    user  system elapsed
#>   0.001   0.000   0.001
stopifnot(all.equal(t1, t3))
```

现在更快了！它是上一个版本的40倍，比刚开始时大约快了1 000倍。

最后，尝试字节码编译。这里需要使用 microbenchmark() 而不是 system.time()，这样才能看到足够精确的差别：

```
rowtstat_bc <- compiler::cmpfun(rowtstat)

microbenchmark(
  rowtstat(X, grp),
  rowtstat_bc(X, grp),
  unit = "ms"
)
#> Unit: milliseconds
#>                 expr  min   lq median   uq  max neval
#>     rowtstat(X, grp) 0.819 1.11   1.16 1.19 14.0   100
#>  rowtstat_bc(X, grp) 0.788 1.12   1.16 1.19 14.6   100
```

371 ~ 373

在本例中，字节码编译几乎没有起到任何作用。

17.10 并行化

并行化使用多个核对一个问题的不同部分同时进行操作。它并不会减少计算时间，但是它可以通过使用更多的计算机资源来节约时间。并行计算是一个复杂的主题，这里我们不可能对其进行深入的探讨。推荐一些资源：

- Ethan McCallum 和 Stephen Weston 编写的《Parallel R》(http://amazon.com/B005Z29QT4)。
- Norm Matloff 编写的《Parallel Computing for Data Science》(http://heather.cs.ucdavis.edu/paralleldatasci.pdf)。

这里要说明的是并行计算的一个简单应用，它也称为“尴尬的并行问题”。尴尬的并

行问题就是一个由很多简单的问题组成的问题，而且它们可以分别独立地解决。尴尬的并行问题的一个最简单的例子就是 `lapply()`，因为它就是对每一个元素进行单独处理。在 Linux 和 Mac 上可以很容易地实现 `lapply()` 的并行化，因为只需要使用 `mclapply()` 替换 `lapply()`。下面的代码使用计算机的所有核进行一个小测试。

```
library(parallel)

cores <- detectCores()
cores
#> [1] 8

pause <- function(i) {
  function(x) Sys.sleep(i)
}

system.time(lapply(1:10, pause(0.25)))
#>    user  system elapsed
#>     0.0     0.0     2.5
system.time(mclapply(1:10, pause(0.25), mc.cores = cores))
#>    user  system elapsed
#>   0.018   0.041   0.517
```

Windows 上有点儿复杂。需要首先设置一个局域集群，然后使用 `parLapply()`：

```
cluster <- makePSOCKcluster(cores)
system.time(parLapply(cluster, 1:10, function(i) Sys.sleep(1)))
#>    user  system elapsed
#>   0.003   0.000   2.004
```

`mclapply()` 和 `makePSOCKcluster()` 的主要不同是：`mclapply()` 产生的单个进程都继承了当前进程，`makePSOCKcluster()` 产生的单个进程都是开始一个新的会话。这就意味着真实的代码需要一些设置。使用 `clusterEvalQ()` 在每一个集群上运行代码，加载需要的包，使用 `clusterExport()` 将当前会话中的对象复制到远程会话。 374

```
x <- 10
psock <- parallel::makePSOCKcluster(1L)
clusterEvalQ(psock, x)
#> Error: one node produced an error: object 'x' not found

clusterExport(psock, "x")
clusterEvalQ(psock, x)
#> [[1]]
#> [1] 10
```

并行计算中还有一些通信开销。如果子问题非常小，那么并行化带来的可能不是帮助而是破坏。也可以在计算机网络中进行分布式计算（而不仅仅是同一台计算机的不同核上），但是这又超出了本书的范围，因为通信开销和计算开销的平衡问题使这个问题变得更加复杂。学习这方面的知识，可以参考：high performance computing CRAN task view（http://cran.r-project.org/web/views/HighPerformanceComputing.html）。

17.11 其他技术

能够写出快速的 R 代码是成为一个好 R 程序员的一部分。如果想写出快速的代码，除了本章提出的这些技巧之外，还需要提高编程技巧。实现这一点的方法有：

- 阅读R博客（http://www.r-bloggers.com/）看看其他人都遇到了哪些性能方面的问题，以及他们是如何解决的。
- 阅读R的编程书籍，比如Norm Matloff的《The Art of R Programming》（http://amazon.com/1593273843）或者Patrick Burns的《R Inferno》（http://www.burns-stat.com/ducuments/books/the-r-inferno/）来学习一些常见的陷阱。
- 学习一些算法和数据结构的课程，学习一些处理具体问题的常用方法。我从Coursera提供的普林斯顿大学的算法课程（http://www.coursera.org/course/algs4partI）学到了不少东西。
- 阅读一些与优化相关的书籍，如Carlos Bueno的《Mature optimisation》（http://carlos.bueno.org/optimization/mature-optimization.pdf）或者Andrew Hunt和David Thomas的《Pragmatic Programmer》(http://amazon.com/020161622X)。

你也可以到社区中寻找帮助。Stackoverflow有很多有用的资源。你需要花费一些时间来创建一个容易让人理解的例子，其中包含了你遇到问题的突出特性。如果你的例子太过复杂，没有人有时间和动力来帮你解决问题。如果太简单，你得到的答案可能只是帮你解决这个例子的问题，而不能解决实际中的问题。如果你也尝试在stackoverflow上回答一些问题，你可以很快地学会如何提出一个好问题。

375
~
376

第 18 章
内　存

对 R 语言内存管理的深入理解有助于我们对一个给定任务需要多少内存做出正确的预测，有助于我们充分利用内存资源。它甚至可以使代码变得更快，因为偶然的复制是使代码变慢的主要原因。本章的主要目的就是帮助你理解 R 中内存管理的基本知识，从单个对象到函数再到大段的代码。在这个过程中，你将学习澄清一些常见的误区，例如你需要调用 `gc()` 来释放内存，或者 `for` 循环总是很慢。

主要内容

- 18.1 节说明如何使用 `object_size()` 来查看对象占用内存的大小，以此作为一个切入点来帮助我们理解 R 如何在内存中存储对象。
- 18.2 节学习两个函数：`mem_used()` 和 `mem_change()`。这两个函数可以帮助我们理解 R 如何分配和释放内存。
- 18.3 节说明如何使用 `lineprof` 添加包在大的代码块中对内存进行分配和释放。
- 18.4 节学习 `address()` 和 `refs()`。这样我们就能理解 R 如何在原位进行修改以及如何在复制中进行修改。理解复制对象的时机对于提高代码效率非常重要。

预备条件

本章使用到 `pryr` 和 `lineprof` 添加包中的一些工具来帮助我们理解内存使用，还需要 ggplot2 的一个样本数据库。如果你还没有安装这些工具，运行下面的代码来安装它们：

```
install.packages("ggplot2")
install.packages("pryr")
devtools::install_github("hadley/lineprof")
```


资料来源

R 内存管理的细节内容来源于多个地方。本章中的大多数信息来自官方文档（特别是 `?Memory` 和 `?gc`）、R-exts 的内存性能分析（http://cran.r-project.org/doc/manuals/R-exts.html#Profiling-R-code-for-memory-use），以及 R-ints 的 SEXPs（http://cran.r-project.org/doc/manuals/R-ints.html#SEXPs）。其余的是根据我阅读源代码、做一些小实验，以及我在 R-devel 提问总结而来的。任何谬误都由我个人来负责任。

18.1　对象大小

首先从 `pryr::object_size()` 开始来学习 R 的内存使用。这个函数可以告诉我们一

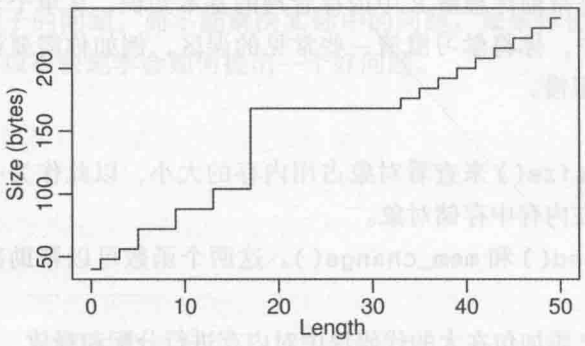
个对象占用了多少字节：

```
library(pryr)
object_size(1:10)
#> 88 B
object_size(mean)
#> 832 B
object_size(mtcars)
#> 6.74 kB
```

（这个函数比内置 `object.size()` 函数好，因为它计算对象中的共享元素并包含环境的大小。）

如果我们使用 `object_size()` 来系统地查看一个整数向量的大小，那么就会发现一些有趣的现象。下面的代码计算并绘制长度从 0～50 的不同整数向量的内存使用情况。你可能期望空向量的大小应该为 0，内存使用与向量的长度成正比。这些都是错的！

```
sizes <- sapply(0:50, function(n) object_size(seq_len(n)))
plot(0:50, sizes, xlab = "Length", ylab = "Size (bytes)",
  type = "s")
```



这不只是整数向量的假象。每个长度为 0 的向量都占用 40 字节的内存：

```
object_size(numeric())
#> 40 B
object_size(logical())
#> 40 B
object_size(raw())
#> 40 B
object_size(list())
#> 40 B
```

这 40 字节用来存储 R 的每个对象的 4 个组成成分：

- 对象元数据（4 字节）。这些元数据存储基础类型（例如，整型）以及用于调试和内存管理的信息。
- 两个指针：一个指向内存中的下一个对象；一个指向前一个对象（2×8 字节）。这种双向链表使得内部 R 代码可以很容易地对内存中的对象进行循环操作。
- 一个指向属性的指针（8 字节）。

378
～
379

所有向量都有 3 个附加成分：

- 向量的长度（4 字节）。仅仅使用 4 字节，所以你可能期望 R 最多可以支持 $2^{4\times8-1}$（2^{31}，大约 20 亿）个元素的向量。但是，在 R 3.0.0 及其以后的版本中，R 可以支持

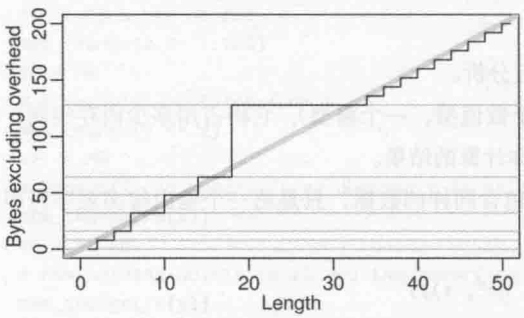
的向量长度最多为 2^{52} 个元素。阅读 R-internals（http://cran.r-project.org/doc/manuals/R-ints.html#Long-vectors）看看在不改变这个字段大小的情况下怎样支持长向量。

- 向量“真正”的长度（4 字节）。基本上永远不使用它，除了当对象是用于环境的散列表外。在这种情况下，真正的长度代表分配的空间，长度代表当前使用的空间。
- 数据（??bytes）。空向量包含 0 字节的数据，这显然也是最重要的！数值向量的每个元素占 8 字节，整型向量占 4 字节，复数向量占 16 字节。

如果把这些加起来也只有 36 字节。剩下的 4 字节是用于填充使得每个组成成分都从一个 8 字节（=64 bit）的边界开始。大多数 CPU 架构要求指针以这样的方式对齐，即使它们不需要这样，访问未对齐的指针会更慢。（如果对此感兴趣，可以阅读 C structure packing(http://www.catb.org/esr/structure-packing/)。）

这解释了图上的截距。但是，为什么内存的大小以不规则的方式增加呢？为了理解这一点，需要知道 R 如何从操作系统请求内存。（使用 `malloc()`）请求内存是一个相对昂贵的操作。如果每创建一个小的向量都请求内存，就会显著地降低 R 的速度。相反，R 请求一大块内存，然后自己管理这块内存，这个内存块称为小向量池，用来创建长度小于 128 字节的向量。为了提高效率和简化操作，它只分配长度为 8、16、32、48、64 或者 128 字节的向量。如果调整前面的图把上面的 40 字节去掉，就会发现这些值正好与内存使用的变动值相对应。

```
plot(0:50, sizes - 40, xlab = "Length",
  ylab = "Bytes excluding overhead", type = "n")
abline(h = 0, col = "grey80")
abline(h = c(8, 16, 32, 48, 64, 128), col = "grey80")
abline(a = 0, b = 4, col = "grey90", lwd = 4)
lines(sizes - 40, type = "s")
```



超过 128 字节，对于 R 管理向量就没有意义了。毕竟，操作系统更擅长分配大段内存。超过 128 字节，R 就请求 8 字节整数倍的内存。这样保证良好的对齐。

一个对象大小的微妙之处在于多个对象之间可以共享组成成分。例如，下面这段代码：

```
x <- 1:1e6
object_size(x)
#> 4 MB

y <- list(x, x, x)
object_size(y)
#> 4 MB
```

y 占用内存不是 x 的 3 倍，因为 R 很聪明，它并没有直接复制 x 3 次，相反它只是将指针指向已经存在的 x。

单独查看 x 和 y 的大小会产生误导。如果想知道它们总共占用了多少内存，必须把它们作为 object_size() 的参数：

```
object_size(x, y)
#> 4 MB
```

380 ~ 381

在这种情况下，x 和 y 一起占用内存的大小与 y 独自占用内存的大小一样。并不总是这样。如果没有共享的组成成分，就可以通过将每个组成成分占用的内存大小加起来求出它们占用的总内存大小，如下面的例子：

```
x1 <- 1:1e6
y1 <- list(1:1e6, 1:1e6, 1:1e6)

object_size(x1)
#> 4 MB
object_size(y1)
#> 12 MB
object_size(x1, y1)
#> 16 MB
object_size(x1) + object_size(y1) == object_size(x1, y1)
#> [1] TRUE
```

同样的问题也出现在字符串中，因为 R 有一个全局字符串池。这说明每个唯一的字符串只存储在一个地方，因此字符向量总是占用比我们想象的少的内存空间：

```
object_size("banana")
#> 96 B
object_size(rep("banana", 10))
#> 216 B
```

练习

1. 用上面的方法，对数值、逻辑以及复数向量进行分析。
2. 如果一个数据框有 100 万行，有 3 个变量（两个数值型，一个整型），它将占用多少内存空间？根据理论计算，然后创建一个这样的数据框来验证你计算的结果。
3. 比较下面两个列表中元素的大小。它们基本上包含同样的数据，只是第一个是由很多短字符串构建的向量，而第二个只包含一个长字符串。

```
vec <- lapply(0:50, function(i) c("ba", rep("na", i)))
str <- lapply(vec, paste0, collapse = "")
```

382

4. 因子 (x) 或等价的字符向量 (as.character(x))，哪一个占用的内存更多？为什么？
5. 解释为什么 1:5 和 list(1:5) 占用的内存大小不一样。

18.2 内存使用与垃圾回收

object_size() 可以告诉我们一个对象占用的内存大小，而 pryr::mem_used() 可以告诉我们内存中所有对象的总大小：

```
library(pryr)
mem_used()
#> 45.4 MB
```

由于下面的这些原因，我们得到数字与操作系统报告的数字不一致：

1）它只包含 R 创建的对象，并不包含 R 接口本身。

2）R 和操作系统都是懒惰的：如果没有实际需要，它们都不会对内存进行回收再利用。由于操作系统没有请求回收内存，所以 R 可能一直占用某些内存。

3）R 计算对象占用的内存，但由于存在一些被删除的对象，所以可能存在一些差异。这个问题也称为内存碎片。

在 `mem_used()` 之上构建的 `mem_change()` 告诉我们在代码执行过程中内存如何变化。正数代表 R 使用的内存正在增加，负数代表正在减少。

```
# Need about 4 mb to store 1 million integers
mem_change(x <- 1:1e6)
#> 4.01 MB
# We get that memory back when we delete it
mem_change(rm(x))
#> -4 MB
```

甚至什么都不做的操作也会占用一点儿内存。这是因为 R 跟踪记录我们所做的每一件事。我们可以忽略任何 2 kB 左右的内存变化。 383

```
mem_change(NULL)
#> 1.47 kB
mem_change(NULL)
#> 1.47 kB
```

在有些语言中，我们必须显式地删除不使用对象所占用的内存。R 使用另一种方法：垃圾回收（GC）。当一个对象不再使用时，GC 会自动释放它所占用的内存。R 通过跟踪每一个对象被多少个名字指向来实现这一点，当没有名字指向一个对象时，这个对象就被删除。

```
# Create a big object
mem_change(x <- 1:1e6)
#> 4 MB
# Also point to 1:1e6 from y
mem_change(y <- x)
#> -4 MB
# Remove x, no memory freed because y is still pointing to it
mem_change(rm(x))
#> 1.42 kB
# Now nothing points to it and the memory can be freed
mem_change(rm(y))
#> -4 MB
```

尽管你可能在哪里看到过，但是永远没有必要自己调用 `gc()` 函数。当需要更多空间时，R 自动进行垃圾回收。如果我们想看看 R 如何调用 GC，我们可以调用 `gcinfo(TRUE)`。我们唯一需要调用 `gc()` 函数的理由就是让 R 将内存归还给操作系统。但是，即便是这样也可能不会有任何作用：老版本的 Windows 没有提供任何方法让程序将内存返回给操作系统。

GC 注重释放不再使用对象所占用的内存。但是，我们还应该意识到存在内存泄漏的可能性。当一直保持指向一个对象而不释放它时就会发生内存泄漏。在 R 中造成内存泄漏有两个主要原因：公式和闭包。因为它们都捕获封闭环境。下面的代码演示了这个问题。在 `f1()` 中，`1:1e6` 只是作为函数内部的参考，因此，当函数结束时内存就返回，内存使用的净改变为 0。`f2()` 和 `f3()` 都返回捕获环境的对象，所以在函数结束时 `x` 不会被释放。 384

```
f1 <- function() {
  x <- 1:1e6
  10
}
mem_change(x <- f1())
#> 1.38 kB
object_size(x)
#> 48 B

f2 <- function() {
  x <- 1:1e6
  a ~ b
}
mem_change(y <- f2())
#> 4 MB
object_size(y)
#> 4 MB

f3 <- function() {
  x <- 1:1e6
  function() 10
}
mem_change(z <- f3())
#> 4 MB
object_size(z)
#> 4.01 MB
```

18.3 使用 lineprof 对内存进行性能分析

当运行代码时，mem_change() 可以捕获内存中的净改变。但有时我们想知道内存的增量变化。一种方法就是使用内存性能分析捕获每隔几毫秒的内存使用。utils::Rprof() 提供这一功能，但是它没有提供有用的结果显示方法。所以我们还是使用 lineprof（https://github.com/hadley/lineprof）包。它的底层也依赖于 Rprof()，只是在显示结果时给出了更多的信息。

385 为了显示 lineprof，我们打算只使用 3 个参数来探索 read.delim()。

```
read_delim <- function(file, header = TRUE, sep = ",") {
  # Determine number of fields by reading first line
  first <- scan(file, what = character(1), nlines = 1,
    sep = sep, quiet = TRUE)
  p <- length(first)

  # Load all fields as character vectors
  all <- scan(file, what = as.list(rep("character", p)),
    sep = sep, skip = if (header) 1 else 0, quiet = TRUE)

  # Convert from strings to appropriate types (never to factors)
  all[] <- lapply(all, type.convert, as.is = TRUE)

  # Set column names
  if (header) {
    names(all) <- first
  } else {
    names(all) <- paste0("V", seq_along(all))
  }
```

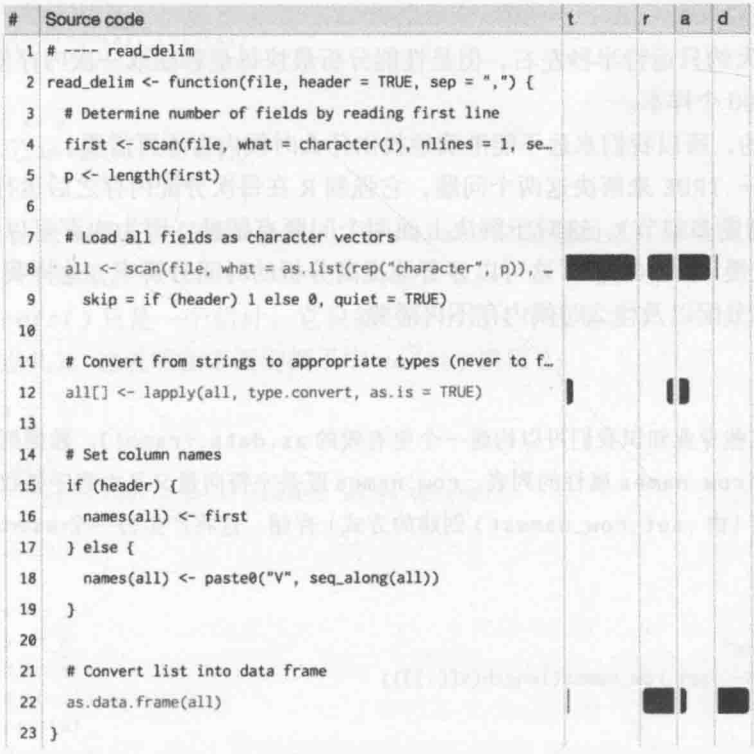
```
  # Convert list into data frame
  as.data.frame(all)
}
```

我们还要创建一个样本 csv 文件：

```
library(ggplot2)
write.csv(diamonds, "diamonds.csv", row.names = FALSE)
```

使用 `lineprof` 很简单。`source()` 代码，`lineprof()` 表达式，然后使用 `shine()` 查看结果。注意，必须使用 `source()` 加载代码。这是因为 `lineprof` 使用 `srcrefs` 来对代码和运行时间进行匹配。而所需的 `srcrefs` 只有从硬盘中加载代码时才会创建。

```
library(lineprof)

source("code/read-delim.R")
prof <- lineprof(read_delim("diamonds.csv"))
shine(prof)
```



`shine()` 将打开一个新的网页（或者，如果使用 RStudio，就打开一个新的面板），它会显示源代码以及相应的内存使用情况。`shine()` 打开一个 shiny 应用，这个应用“阻断”正在进行的 R 会话。按下 escape 或者 ctrl + break，退出这个 shiny 应用。

接下来是 sourcecode，4 列为我们提供了代码性能的详情：

- `t`，该行代码运行所花费的时间（单位为秒）(在 17.1 节中介绍过)。
- `a`，这行代码分配的内存（单位为 MB）。
- `r`，这行代码释放的内存（单位为 MB）。虽然内存分配是确定的，但内存释放是随机的：这取决于 GS 什么时候运行。这说明内存释放只告诉你在运行之前已经不需要这些内存了。

386 ~ 387

- d，向量复制发生的次数。当 R 通过复制向量来对其进行修改时就会发生向量复制。

可以将鼠标悬停在每个条状上来得到准确的数值。在本例中，查看内存分配情况可以得到很多信息：

- scan() 分配了大约 2.5 MB 的内存，这与文件占用硬盘的大小（2.8 MB）非常接近。我们不能期待这两个数完全一样，因为 R 不需要存储标点符号，而且全局字符串池的使用也会节省一些内存。
- 列转换分配另外 0.6 MB 内存。我们可能期望这一步应该释放一些内存，因为我们已经将字符串转换成整数和数值列（它们占用更少的内存），但由于 GC 还没有运行所以我们看不到这种释放。
- 最后，对列表调用 as.data.frame()，分配大约 1.6 MB 内存并进行 600 次复制。这是因为 as.data.frame() 的效率不高，并且它复制输入多次。下一节将讨论更多关于复制方面的内容。

这种分析有两个缺点：

1）read_delim() 大约只运行半秒左右，但是性能分析最快每毫秒获取一次内存使用。这就是说我们只能得到 500 个样本。

2）由于 GC 是惰性的，所以我们永远不能准确地找出什么时候内存不再需要。

可以使用 torture = TRUE 来解决这两个问题，它强制 R 在每次分配内存之后运行 GC（查看 gctorture() 了解更多细节）。这对于解决上面两个问题有帮助，因为内存可以被尽快地释放，R 的运行也变慢 10～100 倍。这可以显著地提高分析的时间分辨率，这样我们就可以看到一些更小的内存分配以及什么时候内存不再需要。

练习

388 1. 当输入是列表时，利用某些专业知识我们可以构建一个更有效的 as.data.frame()。数据框是一个具有 data.frame 类和 row.names 属性的列表。row.names 既是字符向量又是由有序整数组成的向量，并以特殊的形式（由 .set_row_names() 创建的方式）存储。这将产生另一个 as.data.frame()：

```
to_df <- function(x) {
  class(x) <- "data.frame"
  attr(x, "row.names") <- .set_row_names(length(x[[1]]))
  x
}
```

这个函数对 read_delim() 有什么影响？这个函数的缺点是什么？

2. 使用 torture = TRUE 对下面的代码进行性能分析。有让你吃惊的结果吗？阅读 rm() 的源代码看看到底发生了什么。

```
f <- function(n = 1e5) {
  x <- rep(1, n)
  rm(x)
}
```

18.4 原地修改

在下面的代码中，x 发生了什么？

```
x <- 1:10
x[5] <- 10
x
#> [1]  1  2  3  4 10  6  7  8  9 10
```

有两种可能性：

1）R 在原地对 x 进行修改。

2）R 将 x 复制到一个新地方，对新的拷贝进行修改，然后使用名字 x 指向这个新的位置。 389

它表明 R 可以根据环境来选择上面的哪一种。在上面的例子中，R 将在原地进行修改。但是如果还有其他变量也指向 x（内存中位置），R 就会选择第二种方式。要想更仔细地查看到底发生了什么，需要使用 pryr 包的两个工具。给定一个变量名，address() 将告诉我们这个变量在内存中的位置，refs() 将告诉我们有多少名字指向这个位置。

```
library(pryr)
x <- 1:10
c(address(x), refs(x))
# [1] "0x103100060" "1"

y <- x
c(address(y), refs(y))
# [1] "0x103100060" "2"
```

（注意如果正在使用 RStudio，则 refs() 总是返回 2：环境浏览器引用在命令行中创建的每一个对象。）

refs() 只是一个估计。它只能区分出是 1 个还是多于 1 个引用（以后版本的 R 可能会好一点儿）。这说明在下面的例子中 refs() 返回 2：

```
x <- 1:5
y <- x
rm(y)
# Should really be 1, because we've deleted y
refs(x)
#> [1] 2

x <- 1:5
y <- x
z <- x
# Should really be 3
refs(x)
#> [1] 2
```

当 refs(x) 为 1 时，就进行原位修改。当 refs(x) 为 2 时，R 就在拷贝上进行修改（这样才能保证其他指向这里的指针不受影响）。注意，在下面的例子中，虽然 x 改变了但 y 保持指向相同的位置。 390

```
x <- 1:10
y <- x
c(address(x), address(y))
#> [1] "0x7fa9239c65b0" "0x7fa9239c65b0"

x[5] <- 6L
c(address(x), address(y))
#> [1] "0x7fa926be6a08" "0x7fa9239c65b0"
```

另一个有用的函数是 tracemem()。每次复制跟踪的对象就输出一条消息：

```
x <- 1:10
# Prints the current memory location of the object
tracemem(x)
# [1] "<0x7feeaaa1c6b8>"

x[5] <- 6L

y <- x
# Prints where it has moved from and to
x[5] <- 6L
# tracemem[0x7feeaaa1c6b8 -> 0x7feeaaa1c768]:
```

为了交互式使用，tracemem() 比 refs() 更有用一些，但是由于它只是输出一条消息，所以编程时很难使用它。由于它与（文本和代码混合编排的工具）knitr（http://yihui.name/knitr/）的交互非常差，所以在本书中我没有使用它。

对对象进行处理的非原函数也总是增加引用的数目，原函数通常不会。（这里的原因有点儿复杂，可以参考 R-devel 论坛中的话题“有关 NAMED 的疑惑”（http://r.789695.n4.nabble.com/Confused-about-NAMED-td4103326.html）。）

```
# Touching the object forces an increment
f <- function(x) x
{x <- 1:10; f(x); refs(x)}
#> [1] 2

# Sum is primitive, so no increment
{x <- 1:10; sum(x); refs(x)}
#> [1] 1
# f() and g() never evaluate x, so refs don't increment
f <- function(x) 10
g <- function(x) substitute(x)

{x <- 1:10; f(x); refs(x)}
#> [1] 1
{x <- 1:10; g(x); refs(x)}
#> [1] 1
```

通常情况下，假如对象没有其他引用到其他地方，任何原替换函数都在原位进行修改。这包含：[[<-、[<-、@<-、$<-、attr<-、attributes<-、class<-、dim<-、dimnames<-、names<- 和 levels<-。确切地说，所有的非原函数都递增引用，但原函数可能不会。规则非常复杂，没有必要记住它们。然而，我们解决问题的实用方法是，使用 refs() 和 address() 来查看对象何时被复制。

决定复制并不困难，难的是拒绝复制。如果你不得不费尽心机来避免复制发生，最好还是使用 C++ 重写你的函数，我们将在第 19 章中学习它。

18.4.1 循环

391 ~ 392

R 循环的低效率是出了名的。通常这种慢是由于对拷贝进行修改而不是在原位修改。考虑下面的代码，从一个大的数据框的每一列减去中位数：

```
x <- data.frame(matrix(runif(100 * 1e4), ncol = 100))
medians <- vapply(x, median, numeric(1))
```

```
for(i in seq_along(medians)) {
  x[, i] <- x[, i] - medians[i]
}
```

当意识到循环的每次迭代都要对数据框进行复制时，你一定非常吃惊。使用 `address()` 和 `refs()` 来对一个小的循环样本进行分析，可以看得更清楚：

```
for(i in 1:5) {
  x[, i] <- x[, i] - medians[i]
  print(c(address(x), refs(x)))
}
#> [1] "0x7fa92502c3e0" "2"
#> [1] "0x7fa92502cdd0" "2"
#> [1] "0x7fa92502d7c0" "2"
#> [1] "0x7fa92502e1b0" "2"
#> [1] "0x7fa92500bfe0" "2"
```

对于每次迭代，都将 x 移动到一个新位置，所以 `refs(x)` 总是返回 2。之所以这样是因为 `[<-.data.frame` 不是原函数，因此它会使引用增加。如果使用列表代替数据框，这个函数的效率会得到显著的提高。使用原函数对列表进行修改，引用不会增加，所有的修改都在原位进行：

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

for(i in 1:5) {
  y[[i]] <- y[[i]] - medians[i]
  print(c(address(y), refs(y)))
}
#> [1] "0x7fa9250238d0" "2"
#> [1] "0x7fa925016850" "2"
#> [1] "0x7fa9250027e0" "2"
#> [1] "0x7fa92501fd60" "2"
#> [1] "0x7fa925006be0" "2"
```

对于 R 3.1.0 之前的版本，这个问题更加严重，因为数据框的每次复制都是深度复制。所以原来需要运行 5 秒的程序现在只需要 0.01 秒。

18.4.2 练习

1. 下面的代码进行了一次复制。复制发生在哪里？为什么？（提示：查看 `refs(y)`。）

```
y <- as.list(x)
for(i in seq_along(medians)) {
  y[[i]] <- y[[i]] - medians[i]
}
```

2. 在上节中的 `as.data.frame()` 有一个很大的缺点。这个缺点是什么，如何避免？

393
〜
394

第 19 章
使用 Rcpp 编写高性能函数

有时 R 代码就是不够快。虽然我们已经使用性能分析找到了程序的瓶颈，也已经在 R 中做了所有可以做的一切，但是程序仍然不够快。本章将学习如何使用 C++ 重写关键函数来改善代码性能。这个魔法是通过 Rcpp（hyttp://www.rcop.org/）包实现的，它是由 Dirk Eddelbuettel 和 Romain Francois（以及 Doug Bates、John Chambers 和 JJ Allaire 的关键贡献）编写的一个神奇工具。Rcpp 使 C++ 和 R 的结合变得非常简单。虽然也可以编写用于 R 中的 C 或 FORTRAN 代码，但相比来说这些都非常痛苦。Rcpp 为我们提供了一个简洁易用的 API，它使我们可以编写高性能的代码，并把我们与 R 晦涩难懂的 C API 分离开来。

C++ 可以解决的典型瓶颈有：

- 不容易向量化的循环，因为后边的迭代依赖于前面迭代的结果。
- 递归函数，或者包含数百万次的调用函数的问题。使用 C++ 调用函数的开销比使用 R 少很多。
- 需要 R 没有提供的高级数据结构和算法的问题。通过标准模板库（STL），C++ 已经有效实现了从有序图到双端队列等多种重要数据结构。

本章的目的就是学习 C++ 和 Rcpp 的这些知识，它们绝对可以帮我们减少代码中的瓶颈。我们不会在一些高级特征（比如，面向程序的对象、模板）上花费太多时间，因为我们的目的就是编写一些小的、自我包含的函数，而不是大程序。如果有 C++ 的使用经验，将会很有帮助，但这不是必需的。有很多好的免费教程，如 `http://www.learncpp.com/` 和 `http://www.cplusplus.com/`。对于高级主题，Scott Meyers 的《Effective C++》很受欢迎。你也有可能喜欢 Dirk Eddelbuettel 的《Seamless R and C++ integration with Rcpp》（http://www.springer.com/statistics/computational+statistics/book/978-1-4614-6867-7），它对 Rcpp 的各方面都做了详细介绍。

主要内容

- 19.1 节通过将简单的 R 函数转换成等价的 C++ 函数来学习 C++。我们将学习 R 和 C++ 的不同点，以及什么是关键标量、向量和矩阵类。
- 19.1.6 节学习使用 `sourceCpp()` 从硬盘中加载 C++ 文件，与 `source()` 从硬盘中加载 R 代码文件的方式相同。
- 19.2 节讨论如何修改 Rccp 的属性，并学习一些其他重要的类。

- 19.3 节学习如何在 C++ 中处理 R 中的缺失值。
- 19.4 节学习 Rcpp 的“语法糖”，它可以帮我们避免在 C++ 中使用循环，编写与向量化的 R 代码看起来很像的 C++ 代码。
- 19.5 节学习如何使用 C++ 内置的标准模板库（STL）中的最重要的数据结构和算法。
- 19.6 节给出两个真正的案例学习，它们使用 Rcpp 使程序性能得到大幅提升。
- 19.7 节学习如何将 C++ 代码打包进 R 软件包。
- 19.8 节总结本章的所有知识，并提供更多有助于学习 C++ 的资源。

预备条件

本章中的所有例子都需要使用 0.10.1 或更高版本的 Rcpp 包。这个版本的 Rccp 包含 `cppFunction()` 和 `sourceCpp()`，它们使 C++ 和 R 的链接更简单。使用 `install.packages("Rcpp")` 从 CRAN 中安装最新版的 Rcpp。

我们还需要一个可以使用的 C++ 编译器。使用下面的命令安装：

- Windows：安装 Rtools（http://cran.r-project.org/bin/widows/Rtools/）。
- Mac：从应用程序商店安装 Xcode。
- Linux：使用 `sudo apt-get install r-base-dev` 或者其他类似命令。 396

19.1　开始使用 C++

`cppFunction()` 允许我们在 R 中编写 C++ 函数：

```
library(Rcpp)
#>
#> Attaching package: 'Rcpp'
#>
#> The following object is masked from 'package:inline':
#>
#>     registerPlugin
cppFunction('int add(int x, int y, int z) {
  int sum = x + y + z;
  return sum;
}')
# add works like a regular R function
add
#> function (x, y, z)
#> .Primitive(".Call")(<pointer: 0x10fff9e10>, x, y, z)
add(1, 2, 3)
#> [1] 6
```

当我们运行这段代码时，Rcpp 将对 C++ 代码进行编译并构建一个与编译后的 C++ 函数连接在一起的 R 函数。我们将使用这个简单接口来学习如何编写 C++ 代码。C++ 是一门大型语言，不可能在一章中包含全部内容。但是，我们可以学习一些基础知识，这样我们就可以编写一些有用的函数来解决 R 代码中的瓶颈问题。

下面几节我们将通过将简单的 R 函数转换成相应的 C++ 函数来学习一些基础知识。首先从一个简单的没有输入、只有一个标量输出的 R 函数开始，然后再学习一些更复杂的函数：

- 标量输入和标量输出
- 向量输入和标量输出

- 向量输入和向量输出
397 - 矩阵输入和向量输出

19.1.1 没有输入，标量输出

从一个最简单的函数开始。这个函数没有参数且总是返回整数 1：

```
one <- function() 1L
```

等价的 C++ 函数是：

```
int one() {
  return 1;
}
```

我们可以在 R 中使用 cppFunction 对其进行编译和使用

```
cppFunction('int one() {
  return 1;
}')
```

这个小函数展示了 R 和 C++ 的几点重要差别：

- 创建函数的语法和调用函数的语法看起来很像，不需要像在 R 中使用赋值创建函数。
- 必须声明函数返回的输出类型。此函数返回一个 int（整数标量）。R 向量最常见的类型有：NumericVector、IntegerVector、CharacterVector 和 LogicalVector。
- 标量和向量是不同的。数值、整数、字符以及逻辑向量的等价类型为：double、int、String 和 bool。
- 在函数中必须使用显式的 return 语句来返回值。
398 - 每个语句都以 ; 结尾。

19.1.2 标量输入，标量输出

下面例子函数实现了一个标量版的 sign() 函数，如果输入是正数，则返回 1，如果输入是负数，则返回是 -1：

```
signR <- function(x) {
  if (x > 0) {
    1
  } else if (x == 0) {
    0
  } else {
    -1
  }
}

cppFunction('int signC(int x) {
  if (x > 0) {
    return 1;
  } else if (x == 0) {
    return 0;
  } else {
    return -1;
  }
}')
```

在 C++ 版中：

- 声明每个输入的类型与声明输出的类型采用相同的方式。虽然这使代码变得更冗长，但是也很清楚地指出了这个函数需要的输入类型。
- `if` 语法相同，虽然 R 和 C++ 之间有很大的不同，但也有很多相同之处！C++ 也有 `while` 语句，它与 R 的 `while` 语句类似。在 R 中可以使用 `break` 跳出循环，但为了跳过一次迭代需要使用 `continue` 而不是 `next`。

19.1.3　向量输入，标量输出

R 和 C++ 之间的很大不同点就是在 C++ 中循环的开销比 R 较小。例如，我们可以在 R 中使用循环来实现 `sum` 函数。如果你已经使用 R 进行编程有一段时间，使用循环可能是你的本能反应。

399

```
sumR <- function(x) {
  total <- 0
  for (i in seq_along(x)) {
    total <- total + x[i]
  }
  total
}
```

在 C++ 中，循环的开销很小，所以最好使用它们。在 19.5 节中有 `for` 循环的替代方案，它们可以清楚地表达我们的意图；它们虽然不会更快，但是读起来更容易让人理解。

```
cppFunction('double sumC(NumericVector x) {
  int n = x.size();
  double total = 0;
  for(int i = 0; i < n; ++i) {
    total += x[i];
  }
  return total;
}')
```

C++ 版与 R 版很相似，但是：

- 使用 `.size()` 方法来计算向量的长度，它返回一个整数。使用 `.` 来调用 C++ 方法。
- `for` 语句的语法有一点儿不同：`for(init; check; increment)`。通过创建一个称为 `i` 的新变量（值为 0）来开始循环。每次迭代前检查 `i < n` 是否成立，如果不成立就终止循环。在每次迭代后，`i` 的值就加 1，使用了特殊的前缀运算符 `++` 使值加 1。
- 在 C++ 中，向量的索引从 0 开始。重要的事情要说两遍：**在 C++ 中，向量的索引从 0 开始！**当我们将 R 函数转换成 C++ 函数时，这是非常常见的漏洞。
- 使用 `=` 进行赋值，而不是 `<-`。
- C++ 提供了在原位进行修改的运算符：`total += x[i]` 等价于 `total = total + x[i]`。类似的原位操作符有：`-=`、`*=` 和 `/=`。

400

这是 C++ 比 R 更高效的一个好例子。如下面的测试所示，`sumC()` 可以与内置的（经过高度优化）sum() 相媲美了，而 `sumR()` 却要慢几个数量级。

```
x <- runif(1e3)
microbenchmark(
  sum(x),
  sumC(x),
  sumR(x)
)
#> Unit: microseconds
#>      expr    min     lq median     uq      max neval
#>   sum(x)   1.07   1.33   1.66   1.95     6.24   100
#>  sumC(x)   2.49   2.99   3.54   4.36    21.50   100
#>  sumR(x) 333.00 367.00 388.00 428.00 1,390.00   100
```

19.1.4 向量输入，向量输出

下面我们要创建一个函数，它可以计算一个值与一个值向量之间的欧式距离：

```
pdistR <- function(x, ys) {
  sqrt((x - ys) ^ 2)
}
```

函数的定义没有明确地表明我们希望 x 为标量。这就需要我们在文档中进行详细的说明。在 C++ 中这不是问题，因为必须对 x 的类型进行显式声明：

```
cppFunction('NumericVector pdistC(double x, NumericVector ys) {
  int n = ys.size();
  NumericVector out(n);

  for(int i = 0; i < n; ++i) {
    out[i] = sqrt(pow(ys[i] - x, 2.0));
  }
  return out;
}')
```

401

这个函数只介绍了几个概念：

- 我们使用构造器创建了一个长度为 n 的新的数值向量：`NumericVector out(n)`。创建向量的另一种方法是复制一个已有的向量：`NumericVector zs = clone(ys)`。
- C++ 使用 `pow()` 而不是 ^ 计算平方值。

这里需要注意的是，因为 R 版的函数是完全向量化的，所以它已经很快了。在我的计算机上，对一个包含 100 万个元素的向量 y 进行计算，它需要 8 毫秒。C++ 函数比它快 2 倍，4 毫秒，但是假设编写一个 C++ 函数花费 10 分钟，那么这个函数至少使用大约 150 000 次才值得重写它。C++ 函数之所以快的原因很微妙，并且与内存管理有点儿关系。R 版本的函数需要一个长度与 y 相同的中间向量，而分配内存又是一个昂贵的操作。由于 C++ 函数使用中间标量，所以它避免了这种开销。

在 19.4 节中，我们将看到如何利用 Rcpp 的向量化运算来重写这个函数，这样 C++ 代码与 R 代码几乎一样简单明了。

19.1.5 矩阵输入，向量输出

每个向量类型都有一个矩阵的等价体：`NumericMatrix`、`IntegerMatrix`、`CharacterMatrix` 和 `LogicalMatrix`。可以直接使用它们。例如，我们可以创建一个再现 rowSums() 的函数：

```
cppFunction('NumericVector rowSumsC(NumericMatrix x) {
  int nrow = x.nrow(), ncol = x.ncol();
  NumericVector out(nrow);

  for (int i = 0; i < nrow; i++) {
    double total = 0;
    for (int j = 0; j < ncol; j++) {
      total += x(i, j);
    }
    out[i] = total;
  }
  return out;
}')
set.seed(1014)
x <- matrix(sample(100), 10)
rowSums(x)
#>  [1] 458 558 488 458 536 537 488 491 508 528
rowSumsC(x)
#>  [1] 458 558 488 458 536 537 488 491 508 528
```

主要的不同之处是：

- 在 C++ 中，使用 `()` 而不是 `[]` 对矩阵进行子集选取。
- 使用 `.nrow()` 和 `.ncol()` 方法计算矩阵的维度。

19.1.6　使用 sourceCpp

当目前为止，我们已经可以应用 `cppFunction()` 使用内联 C++ 了。在表述问题时这种做法很简单，但是对于实际问题，通常将 C++ 代码单独保存，在使用时把它们 `sourceCpp()` 到 R 中会更方便。这使我们能够利用文本编辑器支持 C++ 文件（例如，语法高亮显示），以及标识编译错误中的行数。

单独的 C++ 文件的扩展名为 `.cpp`，文件的开头应该是：

```
#include <Rcpp.h>
using namespace Rcpp;
```

对于我们希望在 R 中使用的函数，应该给它加上前缀：

```
// [[Rcpp::export]]
```

注意，这里的空格是强制的。

如果你熟悉 roxygen2，你可能会问这与 `@export` 有什么关系？`Rcpp::export` 控制函数是否从 C++ 导入 R；`@export` 控制函数是否从一个包中导出并可以被其他用户使用。

可以将 R 代码嵌入特殊的 C++ 注释块中。如果想运行一些测试代码，这很方便：

```
/*** R
# This is R code
*/
```

使用 `source(echo = TRUE)` 运行 R 代码，这样就不需要显式地打印输出。

为了编译 C++ 代码，使用 `sourceCpp("path/to/file.cpp")`。它将创建匹配的 R 函数并将它们添加到当前的会话。注意这些函数不能保存到 `.Rdata` 文件中，不能重新加载在后一个会话中；每次重启 R 时必须重新创建它们。例如，下面的文件运行 `sourceCpp()`，执行一个 C++ 版的求平均数函数，并把它与内置的 `mean()` 进行比较：

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double meanC(NumericVector x) {
  int n = x.size();
  double total = 0;

  for(int i = 0; i < n; ++i) {
    total += x[i];
  }
  return total / n;
}

/*** R
library(microbenchmark)
x <- runif(1e5)
microbenchmark(
  mean(x),
  meanC(x)
)
*/
```

注：如果你自己运行这段代码，你会发现 `meanC()` 比内置的 `mean()` 快很多。这是因为它在数值准确性上做出了一些牺牲。

在本章接下来的部分，所有的 C++ 代码都是单独存在的而不是包装在一个对 `cppFunction` 的调用中。如果你想对这些例子进行编译或修改，你应该将它们粘贴到一个包含上面这些元素的 C++ 源文件中。

402 ~ 404

19.1.7 练习

有了 C++ 的基础，现在是时候来练习阅读并编写一些简单的 C++ 函数了。阅读下面的每个函数并找出对应的基础 R 函数是什么？你可能现在还不理解程序的每个部分，但是你应该能够看出这些函数都是做什么的。

```
double f1(NumericVector x) {
  int n = x.size();
  double y = 0;

  for(int i = 0; i < n; ++i) {
    y += x[i] / n;
  }
  return y;
}

NumericVector f2(NumericVector x) {
  int n = x.size();
  NumericVector out(n);

  out[0] = x[0];
  for(int i = 1; i < n; ++i) {
    out[i] = out[i - 1] + x[i];
  }
  return out;
```

```
}

bool f3(LogicalVector x) {
  int n = x.size();

  for(int i = 0; i < n; ++i) {
    if (x[i]) return true;
  }
  return false;
}

int f4(Function pred, List x) {
  int n = x.size();

  for(int i = 0; i < n; ++i) {
    LogicalVector res = pred(x[i]);
    if (res[0]) return i + 1;
  }
  return 0;
}

NumericVector f5(NumericVector x, NumericVector y) {
  int n = std::max(x.size(), y.size());
  NumericVector x1 = rep_len(x, n);
  NumericVector y1 = rep_len(y, n);

  NumericVector out(n);

  for (int i = 0; i < n; ++i) {
    out[i] = std::min(x1[i], y1[i]);
  }

  return out;
}
```

为了实践函数编写技术，使用 C++ 重写下面的函数。从现在开始，假设输入没有缺失值。

1）`all()`。

2）`cumprod()`、`cummin()`、`cummax()`。

3）`diff()`。假设滞后为 1，然后再将函数一般化到滞后为 `n`。

4）`range`。

5）`var`。在维基百科（wikipedia）（http://en.wikipedia.org/wiki/Algorithms_for_calculating_variance）中阅读相关资料，看看可以使用哪种方法。当实现一种数值算法时，最好先查找相关的知识，看看哪些是已知的方法。

19.2　属性和其他类

405 ~ 406

我们已经学习了基本的向量类了（`IntegerVector`、`NumericVector`、`LogicalVector`、`CharacterVector`）以及它们对应的标量类（`int`、`double`、`bool`、`String`）和矩阵等价类（`IntegerMatrix`、`NumericMatrix`、`LogicalMatrix`、`CharacterMatrix`）。

所有 R 对象都有属性，可以使用 `.attr()` 来查询和修改这些属性。Rcpp 也提供 `.names()`

作为名字属性的别名。下面的代码片段说明了这些方法。注意 `::create()` 的使用，它是一种类方法。它允许我们从 C++ 标量值来创建 R 向量：

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector attribs() {
  NumericVector out = NumericVector::create(1, 2, 3);

  out.names() = CharacterVector::create("a", "b", "c");
  out.attr("my-attr") = "my-value";
  out.attr("class") = "my-class";

  return out;
}
```

对于 S4 对象，`.slot()` 与 `.attr()` 的作用类似。

19.2.1 列表和数据框

Rcpp 也提供 `List` 类和 `DataFrame` 类，但是它们对输出比输入更有用。这是因为列表和数据框可以包含各种类，而 C++ 需要事先知道它们的类。如果列表的结构已知（例如，它是一个 S3 对象），那么可以提取其元素并使用 `as()` 将它们转换成对应的 C++ 等价类。例如，利用 `lm()` 创建的对象、拟合线性模型的函数，是元素总是相同类型的列表。下面的代码演示了如何提取一个线性模型的平均百分比误差（`mpe()`）。这不是一个使用 C++ 的好例子，因为使用 R 更容易实现，但是它演示了如何处理重要的 S3 类。注意这里使用 `.inherits()` 和 `stop()` 来检查对象的确是一个线性模型。

```
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
double mpe(List mod) {
  if (!mod.inherits("lm")) stop("Input must be a linear model");

  NumericVector resid = as<NumericVector>(mod["residuals"]);
  NumericVector fitted = as<NumericVector>(mod["fitted.values"]);

  int n = resid.size();
  double err = 0;
  for(int i = 0; i < n; ++i) {
    err += resid[i] / (fitted[i] + resid[i]);
  }
  return err / n;
}

mod <- lm(mpg ~ wt, data = mtcars)
mpe(mod)
#> [1] -0.0154
```

19.2.2 函数

可以将 R 函数放入 `Function` 类型的对象中。这样就可以从 C++ 中直接调用 R 函数。首先定义 C++ 函数：

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
RObject callWithOne(Function f) {
  return f(1);
}
```

然后从 R 中调用它：

```
callWithOne(function(x) x + 1)
#> [1] 2
callWithOne(paste)
#> [1] "1"
```

407
~
408

R 函数返回的对象是什么类型呢？我们不知道，所以我们使用万能的对象类 `RObject`。另一个替代方案就是返回一个 `List`。例如，下面的代码就是使用 C++ 的一个 lapply 的基本实现：

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
List lapply1(List input, Function f) {
  int n = input.size();
  List out(n);

  for(int i = 0; i < n; i++) {
    out[i] = f(input[i]);
  }

  return out;
}
```

使用位置参数调用 R 函数：

```
f("y", 1);
```

但是为了使用命名的参数，需要使用特殊的语法：

```
f(_["x"] = "y", _["value"] = 1);
```

19.2.3　其他类型

还有一些用于更专业语言对象的类：`Environment`、`ComplexVector`、`RawVector`、`DottedPair`、`Language`、`Promise`、`Symbol`、`WeakReference` 等。这些都超出了本章的内容，后边也不会进行讨论。

19.3　缺失值

如果要处理缺失值，需要知道两件事：

409

- 如何在 C++ 的标量中体现 R 的缺失值（例如，`double`）。
- 如何在向量中获取并设置缺失值（例如，`NumericVector`）。

19.3.1　标量

下面的代码探索当我们选择一个 R 的缺失值时会发生什么，强制将它转换成标量，然后

强制转换成一个 R 向量。注意这种类型的试验对于我们学习其他操作也是非常有用的。

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
List scalar_missings() {
  int int_s = NA_INTEGER;
  String chr_s = NA_STRING;
  bool lgl_s = NA_LOGICAL;
  double num_s = NA_REAL;

  return List::create(int_s, chr_s, lgl_s, num_s);
}
```

```
str(scalar_missings())
#> List of 4
#>  $ : int NA
#>  $ : chr NA
#>  $ : logi TRUE
#>  $ : num NA
```

除了 `bool` 外，一切看起来都不错：保留了所有的缺失值。但是，正如我们后边看到的，事情并不是那么简单：

1. 整数

对于整数中，将缺失值存储为最小的整数。如果对它们什么都不做，它们将保留。但是，由于 C++ 不知道最小整数有这种特殊行为，所以如果我们对它进行操作就会得到错误的
410 结果：例如，`evalCpp('NA_INTEGER + 1')` 的结果是 -2147483647。

所以，如果我们想要以整数处理缺失值，可以使用一个长度为 1 的 `IntegerVector` 或者编写代码时小心一点儿。

2. 双精度

对于双精度，我们可以忽略缺失值而处理 NaN（并不是一个数）。这是因为 R 的 NA 是一种特殊类型的 IEEE 754 浮点数 NaN。所以任何包含 NaN（或者 C++ 中的 NAN）的逻辑表达式总是被当作 FALSE 来处理：

```
evalCpp("NAN == 1")
#> [1] FALSE
evalCpp("NAN < 1")
#> [1] FALSE
evalCpp("NAN > 1")
#> [1] FALSE
evalCpp("NAN == NAN")
#> [1] FALSE
```

如果和布尔值一起使用，就需要小心了：

```
evalCpp("NAN && TRUE")
#> [1] TRUE
evalCpp("NAN || FALSE")
#> [1] TRUE
```

但是，在数值环境中 NaN 的结果就是 NA：

```
evalCpp("NAN + 1")
#> [1] NaN
evalCpp("NAN - 1")
#> [1] NaN
evalCpp("NAN / 1")
#> [1] NaN
evalCpp("NAN * 1")
#> [1] NaN
```

19.3.2　字符串

String 是 Rcpp 引入的标量字符串类，所以它知道如何处理缺失值。 411

19.3.3　布尔型

C++ 的 bool 只有两种可能的取值：true 或 false，但 R 的逻辑向量有 3 种可能的取值：TRUE、FALSE 和 NA。如果对一个长度为 1 的逻辑向量进行转换，确保它没有包含缺失值，否则它们就会被转换成 TRUE。

19.3.4　向量

对于向量，针对不同类型的向量需要使用不同类型的缺失值：NA_REAL、NA_INTEGER、NA_LOGICAL、NA_STRING：

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
List missing_sampler() {
  return List::create(
    NumericVector::create(NA_REAL),
    IntegerVector::create(NA_INTEGER),
    LogicalVector::create(NA_LOGICAL),
    CharacterVector::create(NA_STRING));
}
```

```
str(missing_sampler())
#> List of 4
#>  $ : num NA
#>  $ : int NA
#>  $ : logi NA
#>  $ : chr NA
```

使用类方法 ::is_na() 可以检查向量中的值是否缺失：

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
LogicalVector is_naC(NumericVector x) {
  int n = x.size();
  LogicalVector out(n);
  for (int i = 0; i < n; ++i) {
    out[i] = NumericVector::is_na(x[i]);
  }
  return out;
```

```
}

is_naC(c(NA, 5.4, 3.2, NA))
#> [1]  TRUE FALSE FALSE  TRUE
```

或者，也可以使用语法糖函数 `is_na()`，它接收一个向量并返回一个逻辑向量。

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
LogicalVector is_naC2(NumericVector x) {
  return is_na(x);
}

is_naC2(c(NA, 5.4, 3.2, NA))
#> [1]  TRUE FALSE FALSE  TRUE
```

19.3.5 练习

1. 重写第一个练习中的所有函数，使它们可以处理缺失值。如果 `na.rm` 为 true，则忽略缺失值。如果 `na.rm` 为 false 且输入包含缺失值，则返回缺失值。可以尝试重写这些函数：`min()`、`max()`、`range()`、`mean()` 和 `var()`。
2. 重写 `cumsum()` 和 `diff()` 使它们可以处理缺失值。注意这些函数的行为有些复杂。

19.4 Rcpp 语法糖

412 ~ 413

Rcpp 还提供了大量的语法糖，从而使 C++ 函数可以像它们的 R 等价函数那样工作。实际上，Rcpp 语法糖可以使我们将 C++ 代码写的与其等价的 R 代码看起来完全一样。如果你感兴趣的语法糖函数，你应该使用它：它的表达能力很强并且经过了良好的测试。语法糖函数并不总是比我们自己编写的函数快，但如果以后花时间优化 Rcpp，它就会变快。

语法糖函数可以粗略地分为：

- 算术和逻辑运算符
- 逻辑总结函数
- 向量视图
- 其他有用的函数

19.4.1 算术和逻辑运算符

所有的基本算术和逻辑运算符都是向量化的：`+`、`*`、`-`、`/`、`pow`、`<`、`<=`、`>`、`>=`、`==`、`!=`、`!`。例如，我们可以使用语法糖大幅度地简化 `pdistC()` 函数。

```
pdistR <- function(x, ys) {
  sqrt((x - ys) ^ 2)
}

#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector pdistC2(double x, NumericVector ys) {
  return sqrt(pow((x - ys), 2));
}
```

19.4.2　逻辑总结函数

语法糖函数 `any()` 和 `all()` 都是完全惰性的，所以例如 `any(x == 0)`，可能只需要对向量中的一个元素进行评估，并返回一个可以被 `.is_true()`、`.is_false()` 或 `.is_na()` 转换成 `bool` 的特殊类型。我们可以使用这个语法糖来编写一个检测数值向量是否包含缺失值的函数。为了在 R 中实现它，可以使用 `any(is.na(x))`：

```
any_naR <- function(x) any(is.na(x))
```

414

但是，无论缺失值在什么位置，这个函数的工作量都是一样的。下面是 C++ 版实现：

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
bool any_naC(NumericVector x) {
  return is_true(any(is_na(x)));
}
```

```
x0 <- runif(1e5)
x1 <- c(x0, NA)
x2 <- c(NA, x0)

microbenchmark(
  any_naR(x0), any_naC(x0),
  any_naR(x1), any_naC(x1),
  any_naR(x2), any_naC(x2)
)
#> Unit: microseconds
#>         expr    min     lq median     uq   max neval
#>  any_naR(x0) 253.00 255.00 262.00 295.00 1,370   100
#>  any_naC(x0) 308.00 309.00 310.00 324.00 1,050   100
#>  any_naR(x1) 253.00 256.00 260.00 351.00 1,160   100
#>  any_naC(x1) 308.00 309.00 312.00 322.00 1,350   100
#>  any_naR(x2)  86.50  91.50  94.00 111.00 1,070   100
#>  any_naC(x2)   1.79   2.32   2.93   3.69    10   100
```

19.4.3　向量视图

许多有用的函数可以为我们提供向量的“视图”：`head()`、`tail()`、`rep_each()`、`rep_len()`、`rev()`、`seq_along()` 和 `seq_len()`。在 R 中这些函数在执行时都对向量进行复制，但是在 Rcpp 中它们都是简单的指向原向量，覆盖子集选取操作（`[`）而实现特殊的行为。所以它们的效率很高：例如，`rep_len(x, 1e6)` 不必对 x 复制 100 万次。

415

19.4.4　其他有用的函数

最后，还有一大批语法糖函数可以模拟经常使用的 R 函数：

- 数学函数：`abs()`、`acos()`、`asin()`、`atan()`、`beta()`、`ceil()`、`ceiling()`、`choose()`、`cos()`、`cosh()`、`digamma()`、`exp()`、`expm1()`、`factorial()`、`floor()`、`gamma()`、`lbeta()`、`lchoose()`、`lfactorial()`、`lgamma()`、`log()`、`log10()`、`log1p()`、`pentagamma()`、`psigamma()`、`round()`、`signif()`、`sin()`、`sinh()`、`sqrt()`、`tan()`、`tanh()`、`tetragamma()`、

trigamma()、trunc()。

- 标量总结函数：mean()、min()、max()、sum()、sd()和（针对向量的）var()。
- 向量总结函数：cumsum()、diff()、pmin()和pmax()。
- 查找特定值函数：match()、self_match()、which_max()、which_min()。
- 与复制相关的运算：duplicated()、unique()。
- 所有标准分布的d/q/p/r开始的函数。

最后，noNA(x)假设向量x中不包含缺失向量，并允许我们对一些数学运算做出优化。

19.5 STL

当我们需要实现更加复杂的算法时，C++才能显示出它真正的实力。标准模板库（STL）为我们提供了一套非常非常有用的数据结构和算法。本节将学习一些最重要的算法和数据结构，并为以后的学习指明正确的方向。我不可能教给你STL的所有知识，但希望这些例子能够向你展示STL的真正实力，并以此激发出你学习的兴趣。

如果你需要使用的一个算法或者数据结构，但它没有包含在STL中，你可以到boost（http://www.boost.org/doc/）中去查找。在计算机上安装boost已经超出了本章的范围，安装boost后，在文件的开头加上特定的头文件（例如，#include <boost/array.hpp>），就
416 可以使用包含在其中的数据结构和算法。

19.5.1 使用迭代器

迭代器在STL中广泛使用：很多函数或者接收迭代器或者输出迭代器。它们是基础循环抽象的下一个步骤，而并不关心具体数据结构的细节。迭代器包含3个主要的运算符：

1）使用++进入下一步。

2）使用*获取它们引用或**解引用**的值。

3）使用==进行比较。

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double sum3(NumericVector x) {
  double total = 0;

  NumericVector::iterator it;
  for(it = x.begin(); it != x.end(); ++it) {
    total += *it;
  }
  return total;
}
```

for循环中的主要变化有：

- 循环从x.begin()开始直到x.end()结束。可以做一点儿小优化就是将结束迭代器的值存储起来，这样就不必每次都查看它。每次迭代只能节约2纳秒，所以只有当循环内的操作非常简单时这样做才重要。
- 没有使用索引来获取x的值，而是使用解引用操作符来获取它的当前值：*it。

- 注意迭代器的类型：NumericVector::iterator。每个向量类型都有自己的迭代器类型：LogicalVector::iterator、CharacterVector::iterator 等。

迭代器还允许我们使用函数的 apply 族的 C++ 等价体。例如，我们再次重写 sum() 来使用 accumulate() 函数，它接收开始迭代器和结束迭代器，并将向量中的所有值加起来。第三个参数累计起始值：它非常重要，因为它还决定累计所使用的数据类型（我们使用 0.0 而不是 0，所以累计使用 double 而不是 int。）。为了使用 accumulate()，需要在头文件中包含 <numeric>。 417

```
#include <numeric>
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double sum4(NumericVector x) {
  return std::accumulate(x.begin(), x.end(), 0.0);
}
```

accumulate()（以及 <numeric> 中的其他函数，如 adjacent_difference()、inner_product() 和 partial_sum()）在 Rcpp 中并不是很重要，因为 Rcpp 语法糖提供与其等价的函数。

19.5.2　算法

<algorithm> 头文件提供了大量的用来处理迭代器的算法。一个好的参考资源是：http://www.cplusplus.com/。例如，我们可以编写一个基本 Rcpp 版的 findInterval()，它可以接受两个参数：一个值的向量和一个断点的向量，然后找出每一个 x 所在的区间。这个例子可以说明迭代器的一些高级特征。阅读下面的代码，看看它是如何工作的。

```
#include <algorithm>
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
IntegerVector findInterval2(NumericVector x, NumericVector breaks) {
  IntegerVector out(x.size());

  NumericVector::iterator it, pos;
  IntegerVector::iterator out_it;
  for(it = x.begin(), out_it = out.begin(); it != x.end();
      ++it, ++out_it) {
    pos = std::upper_bound(breaks.begin(), breaks.end(), *it);
    *out_it = std::distance(breaks.begin(), pos);
  }

  return out;
}
```

这段代码的关键点是：

- 同时单步调试两个迭代器（输入和输出）。
- 可以分配一个解引用迭代器（out_it）来改变 out 中的值。

- ❑ upper_bound() 返回一个迭代器。如果需要 upper_bound() 的值，可以对其解引用；可以使用 distance() 函数来找到它的位置。
- ❑ 如果希望这个函数像 R 中的 findInterval()（使用 C 语言编写）函数一样快，我们需要调用 .begin() 和 .end() 计算结果并将结果保存起来。这很简单，但是由于它偏离了学习的重点所以可以忽略它。做出这些改变后，这个函数就会比 R 的 findInterval() 稍微快一点儿，但它的长度只有 R 函数代码的 1/10。

使用 STL 中的算法通常比自己重新编写好。在《 Effective STL 》一书中，Scott Meyers 给出了 3 个原因：高效性、正确性和可维护性。STL 中的算法都是 C++ 专家编写的，效率极高，而且它们已经存在一段时间了，所以都经过了良好的测试。使用标准算法通常使代码的意图更加明确，有助于使它变得更加易读且更具可维护性。

19.5.3 数据结构

STL 提供了大量的数据结构：array、bitset、list、forward_list、map、multimap、multiset、priority_queue、queue、dequeue、set、stack、unordered_map、unordered_set、unordered_multimap、unordered_multiset 和 vector。这些数据结构中最重要的是：vector、unordered_set 和 unordered_map。本节将重点学习这 3 种数据结构，其他数据结构的使用也是一样的：它们只是性能有所不同。例如，deque 与向量的接口非常类似，但是底层实现不同，所以性能上也有所不同。你可能想对你遇到的问题尝试它们。STL 数据结构的一个好的参考是 http://www.cplusplus.com/reference/stl/——我建议你在使用 STL 时可以一直打开这个页面。

418 ~ 419

Rcpp 知道如何将很多 STL 数据结构转换成它们的 R 等价体，所以可以在函数中直接返回它们，而不必将它们转换成 R 的数据结构。

19.5.4 向量

STL 向量与 R 向量非常类似，但是它更有效率。当我们事先不知道输出有多大时，使用向量非常合适。向量是模板化的，这说明当我们创建向量时我们必须指定向量包含的对象的类型：vector<int>、vector<bool>、vector<double>、vector<String>。使用标准的 [] 可以获取向量中的单个元素，使用 .push_back() 可以在向量的末尾添加新的元素。如果我们事先知道向量的大小，可以使用 .reserve() 分配足够的存储空间。

下面的代码实现了一个 rle() 函数。它产生两个输出向量：一个值的向量；一个向量 lengths，它表示每个元素重复的次数。它的工作原理是：对整个输入向量 x 进行循环，将每个值与前一个值进行比较，如果相同，则将 lengths 中的最后一个值加 1；如果不同，就将该值与 values 的最后一个值相加，并将与其对应的长度设置为 1。

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
List rleC(NumericVector x) {
  std::vector<int> lengths;
  std::vector<double> values;

  // Initialise first value
  int i = 0;
```

```
  double prev = x[0];
  values.push_back(prev);
  lengths.push_back(1);

  NumericVector::iterator it;
  for(it = x.begin() + 1; it != x.end(); ++it) {
    if (prev == *it) {
      lengths[i]++;
    } else {
      values.push_back(*it);
      lengths.push_back(1);

      i++;
      prev = *it;
    }
  }

  return List::create(
    _["lengths"] = lengths,
    _["values"] = values
  );
}
```

（另一种解决方案是：用总是指向向量的最后一个元素的迭代器 `lengths.rbegin()` 替换 `i`。你可以自己试试。）

`http://www.cplusplus.com/` 中有其他关于向量方法的描述。

19.5.5　集合

集合可以保存唯一的一组值，并能够非常高效地查找一个值是否存在于这个集合中。对于涉及重复或唯一值（例如，`unique`、`duplicated` 或 `in`）的问题，它们非常有用。C++ 既提供了有序集合（`std::set`）又提供了无序集合（`std::unordered_set`），可以根据需要来选择使用。无序集合应该更快一点儿（因为它们使用散列表而不是树），所以即使我们需要有序集合，我们也应该考虑使用无序集合，然后再对输出的结果进行排序。与向量一样，集合也是模板化的，所以需要根据用途选择正确的类型：`unordered_set<int>`，`unordered_set<bool>` 等。更多细节知识可以参考：`http://www.cplusplus.com/reference/set/set/>` 和 `<http://www.cplusplus.com/reference/unordered_set/unordered_set/`。

420 ~ 421

下面的函数使用无序集合来实现整数向量的 `duplicated()` 函数。注意 `seen.insert(x[i]).second` 的使用。`insert()` 有两个返回值：`.first` 值是指向元素的一个迭代器；`.second` 是一个布尔值，如果值新加入集合，`.second` 就为 `TRUE`。

```
// [[Rcpp::plugins(cpp11)]]
#include <Rcpp.h>
#include <unordered_set>
using namespace Rcpp;

// [[Rcpp::export]]
LogicalVector duplicatedC(IntegerVector x) {
  std::unordered_set<int> seen;
  int n = x.size();
  LogicalVector out(n);
```

```
  for (int i = 0; i < n; ++i) {
    out[i] = !seen.insert(x[i]).second;
  }

  return out;
}
```

注意无序集合只在C++ 11中可以使用，也就是说需要使用cpp11插件，[[Rcpp::plugins(cpp11)]]。

19.5.6 图

图与集合类似，但它不是存储一个值是否存在，它能存储额外的数据。对需要查找值的函数（如table()或match()）它很有用。与集合一样，也存在有序图（std::map）和无序图（std::unordered_map）。由于图既有值又有主键，所以在初始化图时需要指定它们的类型：map<double,int>、unordered_map<int、double>等。下面的例子说明如何使用map来实现数值向量的table()：

```
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
std::map<double, int> tableC(NumericVector x) {
  std::map<double, int> counts;

  int n = x.size();
  for (int i = 0; i < n; i++) {
    counts[x[i]]++;
  }

  return counts;
}
```

注意无序图只有在C++ 11中可以使用，所以还是需要[[Rcpp::plugins(cpp11)]]。

19.5.7 练习

为了练习使用STL算法和数据结构，可使用C++中的R函数实现下面的函数，可使用给出的提示：

1）使用partial_sort实现median.default()。

2）使用unordered_set和find()或count()方法实现%in%。

3）使用unordered_set实现unique()。（挑战：一行代码搞定！）

4）min()使用std::min()，或者max()使用std::max()。

5）使用min_element实现which.min()，或者使用max_element实现which.max()。

6）对于整数，使用set_union、set_intersection和set_difference实现setdiff()、union()和intersect()。

19.6 案例研究

422 ~ 423

下面的案例学习展示了真实世界如何使用C++来代替低效率的R代码。

19.6.1　Gibbs 采样器

下面的案例学习是对 Dirk Eddelbuettel 博客（http://dirk.eddelbuettel.com/blog/2011/ 07/14）中一个例子的更新，它说明如何将 R 中的 Gibbs 采样器转换成 C++ 中采样器。下面的 R 代码和 C++ 代码非常类似（将 R 版本转换为 C 版本只需要几分钟），但是速度却提高了 20 倍（在我的计算机上）。Dirk 的博客还提供了另一种使它变得更快的方法：使用 GSL（通过 RcppGSL 包，在 R 中可以很容易使用）中更快的随机数生成器函数，可以提高 2～3 倍。

R 代码如下所示：

```
gibbs_r <- function(N, thin) {
  mat <- matrix(nrow = N, ncol = 2)
  x <- y <- 0

  for (i in 1:N) {
    for (j in 1:thin) {
      x <- rgamma(1, 3, y * y + 4)
      y <- rnorm(1, 1 / (x + 1), 1 / sqrt(2 * (x + 1)))
    }
    mat[i, ] <- c(x, y)
  }
  mat
}
```

将它直接转换成 C++ 代码。我们可以：

- 对所有的变量增加类型声明。
- 使用 (而不使用 [来对矩阵进行索引。
- 对 `rgamma` 和 `rnorm` 的结果增加下标，将它们从向量转换成标量。

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericMatrix gibbs_cpp(int N, int thin) {
  NumericMatrix mat(N, 2);
  double x = 0, y = 0;

  for(int i = 0; i < N; i++) {
    for(int j = 0; j < thin; j++) {
      x = rgamma(1, 3, 1 / (y * y + 4))[0];
      y = rnorm(1, 1 / (x + 1), 1 / sqrt(2 * (x + 1)))[0];
    }
    mat(i, 0) = x;
    mat(i, 1) = y;
  }

  return(mat);
}
```

对两段代码进行性能测试：

```
microbenchmark(
  gibbs_r(100, 10),
  gibbs_cpp(100, 10)
)
#> Unit: microseconds
```

```
#>              expr   min    lq median    uq    max neval
#>  gibbs_r(100, 10) 6,980 8,150  8,430 8,740 44,100   100
#>  gibbs_cpp(100, 10)   258   273    279   293  1,330   100
```

19.6.2 R 向量化与 C++ 向量化

这个例子改编自“Rcpp is smoking fast for agent-based models in data frames”（http://www.babelgraph.org/wp/?p=358）。这个函数根据 3 个输入来预测模型的输出。基础 R 版的预测函数为：

```
vacc1a <- function(age, female, ily) {
  p <- 0.25 + 0.3 * 1 / (1 - exp(0.04 * age)) + 0.1 * ily
  p <- p * if (female) 1.25 else 0.75
  p <- max(0, p)
  p <- min(1, p)
  p
}
```

424 ~ 425

我们希望将这个函数应用于许多输入，所以我们可以使用 for 循环来编写向量输入版的函数。

```
vacc1 <- function(age, female, ily) {
  n <- length(age)
  out <- numeric(n)
  for (i in seq_len(n)) {
    out[i] <- vacc1a(age[i], female[i], ily[i])
  }
  out
}
```

如果你熟悉 R，你已经意识到这样做会使程序变慢，的确如此！有两个方法可以帮助我们解决这个问题。如果你对 R 词汇掌握很好，你马上就会想到如何向量化这个函数（使用 `ifelse()`、`pmin()` 和 `pmax()`）。或者，可以使用 C++ 重写 `vacc1a()` 和 `vacc1()`，因为我们知道在 C++ 中循环和函数调用的开销都很小。

两种方法都很简单。在 R 中（第一种方法）：

```
vacc2 <- function(age, female, ily) {
  p <- 0.25 + 0.3 * 1 / (1 - exp(0.04 * age)) + 0.1 * ily
  p <- p * ifelse(female, 1.25, 0.75)
  p <- pmax(0, p)
  p <- pmin(1, p)
  p
}
```

（如果你使用 R 已很长时间，你可能会认识到这段代码存在瓶颈：`ifelse`、`pmin` 和 `pmax` 都很慢，所以你可以使用 `p + 0.75 + 0.5 * female`、`p[p < 0] <- 0` 和 `p[p > 1] <- 1` 来替代它们。你可能想对它们进行测试。）

或者在 C++ 中（第二种方法）：

```
#include <Rcpp.h>
using namespace Rcpp;

double vacc3a(double age, bool female, bool ily){
  double p = 0.25 + 0.3 * 1 / (1 - exp(0.04 * age)) + 0.1 * ily;
  p = p * (female ? 1.25 : 0.75);
```

```
  p = std::max(p, 0.0);
  p = std::min(p, 1.0);
  return p;
}
// [[Rcpp::export]]
NumericVector vacc3(NumericVector age, LogicalVector female,
                    LogicalVector ily) {
  int n = age.size();
  NumericVector out(n);

  for(int i = 0; i < n; ++i) {
    out[i] = vacc3a(age[i], female[i], ily[i]);
  }

  return out;
}
```

下面我们生成一些样本数据，并检查所有 3 个版本函数是否返回相同的值：

```
n <- 1000
age <- rnorm(n, mean = 50, sd = 10)
female <- sample(c(T, F), n, rep = TRUE)
ily <- sample(c(T, F), n, prob = c(0.8, 0.2), rep = TRUE)

stopifnot(
  all.equal(vacc1(age, female, ily), vacc2(age, female, ily)),
  all.equal(vacc1(age, female, ily), vacc3(age, female, ily))
)
```

原始博客中忘记做这一步检查，并且在 C++ 版本中引入了一个错误：它使用 `0.004` 而不是 `0.04`。最后，我们可以对这 3 种方法进行性能测试：

```
microbenchmark(
  vacc1 = vacc1(age, female, ily),
  vacc2 = vacc2(age, female, ily),
  vacc3 = vacc3(age, female, ily)
)
#> Unit: microseconds
#>   expr      min       lq   median       uq      max neval
#>  vacc1 4,210.0 4,680.0 5,150.0 6,050.0 11,800   100
#>  vacc2   327.0   373.0   403.0   449.0    750   100
#>  vacc3    16.6    20.1    27.5    34.1     82   100
```

426 ~ 428

毫不奇怪，使用循环的原始方法非常慢。向量化使 R 的速度有很大提升，使用 C++ 循环可以使速度提高大约 10 倍。C++ 如此之快使我有点儿惊讶，这可能是因为 R 不得不创建 11 个向量来存储中间结果，而 C++ 只需要创建一个。

19.7　在添加包中应用 Rcpp

与 `sourceCpp()` 一起使用的 C++ 代码也可以打包到软件包中。将标准的 C++ 源文件打包有以下几点好处：

1）没有 C++ 开发工具的用户也可以使用你的代码。

2）R 软件包构建系统可以将多个源文件以及它依赖的文件打包在一起。

3）软件包为测试、文档和一致性提供了附加的基础设施。

为了将 Rcpp 加入到现有软件包中，将 C++ 文件放在 src/ 目录中，并修改或创建下面的配置文件：

- 在 DESCRIPTION 添加

```
LinkingTo: Rcpp
Imports: Rcpp
```

- 确保 NAMESPACE 包含

```
useDynLib(mypackage)
importFrom(Rcpp, sourceCpp)
```

我们需要从 Rcpp 中导入某些（任意）东西，这样内部 Rcpp 代码才能正常加载。这是 R
428 的一个漏洞，希望未来能够修复它。

可以使用 Rcpp.package.skeleton() 产生一个包含简单的“hello world”函数的 Rcpp 包：

```
Rcpp.package.skeleton("NewPackage", attributes = TRUE)
```

为了基于和 sourceCpp() 一起应用的 C++ 文件来生成 R 添加包，可以使用参数 cpp_files。

```
Rcpp.package.skeleton("NewPackage", example_code = FALSE,
                      cpp_files = c("convolve.cpp"))
```

在构建软件包前，需要运行 Rcpp::compileAttributes()。这个函数扫描 C++ 文件来查找 Rcpp::export 属性，并产生使这些函数在 R 中可以使用的代码。当增加、删除或者修改函数时，需要重新运行 compileAttributes()。这是 devtools 包和 RStudio 自动完成的。

更多细节可以查看 Rcpp 软件包文档：vignette("Rcpp-package")。

19.8 更多学习资源

本章只接触了小部分 Rcpp，学习了一些基本工具以便使用 C++ 重写低效率 R 代码。Rcpp book（http://www.rcpp.org/book）是学习 Rcpp 最好的参考资料。注意，Rcpp 还有很多其他工具，使它可以在 R 和现有 C++ 之间进行交互，它们包括：

- 属性的附加特性包括：设置默认参数、链接外部 C++ 依赖的文档、从软件包导出 C++ 接口。这些特性和更多特性详见 Rcpp 属性文档：vignette("Rcpp-attributes")。
- 在 C++ 数据结构和 R 数据结构之间自动创建包装器，将 C++ 类映射到参考类。关于这个主题的好的介绍是 Rcpp 模块文档：vignette("Rcpp-modules")。
- Rcpp 快速参考指南：vignette("Rcpp-quickref") 提供了 Rcpp 类和一些编程常
429 用语的有用总结。

我强烈建议经常到 Rcpp 主页（http://www.rcpp.org）和 Dirk 的 Rcpp 网页（http://dirk.eddelbuettel.com/code/rcpp.html）看一看，并注册一个 Rcpp 邮件列表（http://lists.r-forge.r-project.org/cgi-bin/mailman/listinfo/rcpp-devel）。Rcpp 仍然处于活跃的开发阶段，每个更新版本都变得更好。

我找到的学习 C++ 很有帮助的其他资源有：

- Scott Meyers 的《 Effective C++ 》（ http://amzn.com/0321334876?tag=devtools-20 ）和《 Effective STL 》（http://amzn.com/0201749329?tag=devtools-20 ）。
- 《 C++ Annotations 》（http://www.icce.rug.nl/documents/cplusplus/cplusplus.html），目标读者是知识渊博的 C 语言用户（或者其他类似 C 语言的语言，例如 Perl 和 Java），如果你想学习更多知识，或者过渡到 C++，可以参考这本书。
- 《 Algorithm Libraries 》（ http://www.cs.helsinki.fi/u/tpkarkka/alglib/k06 ），它对重要的 STL 概念进行了更加专业但仍然简明的描述。

编写高效率的代码也要求你重新考虑你的基本方法：深刻的理解基本数据结构和算法是很有帮助的。这不在本书的讨论范围内。我建议阅读下述资料：《 Algorithm Design Manual 》（ http://amzn.com/0387948600?tag=devtools-20 ）。MIT 出版的《 Introduction to Algorithms 》（ http://ocw.mit.edu/courses/electrical-engineering-and-omputerscience/6-046j-introduction-to-algorithms-sma-5503-fall-2005/）。由 Robert Sedgewick 和 Kevin Wayne 编写的《Algorithms》，该书有免费在线版（ http://algs4.cs.princeton.edu/home/），并且有相应的 coursera 在线课程（https://www.coursera.org/course/algs4partI）。

19.9　致谢

我非常感谢 Rcpp 邮件列表中很多非常有帮助的交流，尤其是 Romain Francois 和 Dirk Eddelbuettel 不仅对我提出的很多问题做了详尽的回答，还迅速对 Rcpp 进行了完善。如果没有 JJ Allaire 就不可能有本章的内容，是他鼓励我学习 C++，并在这个过程中回答了我很多愚蠢的问题。

430

第 20 章
R 的 C 接口

阅读 R 源代码是提高编程技巧强有力的技术。然而，许多基础 R 函数以及旧的添加包中的许多函数都是用 C 来编写的。了解这些函数如何工作是有用的，因此本章介绍 R 的 C 应用程序接口（API）。你需要一些基本的 C 语言知识，你可以从标准 C 教材中学习这些知识（例如，Kernigan 和 Ritchie 的《C Programming Language》(http://amzn.com/0131101633?tag=devtools-20)），或者从第 19 章中学习。你需要一点儿耐心，但是阅读 R 的 C 源代码是可能的，通过阅读 C 代码你可以学习许多知识。

本章的内容主要来自 Writing R extensions（http://cran.r-project.org/doc/manuals/R-exts.html）的 Section5（“Sytem and foreign language interfaces”），但侧重于最好的实践和现代的工具。这就是说它不包含旧的 `.C` 接口、`Rdefines.h` 中定义的旧的 API 或很少使用的语言特性。为了了解完整的 C API，可以查看头文件 `Rinternals.h`。在 R 中可以容易地找到和显示这个文件。

```
rinternals <- file.path(R.home("include"), "Rinternals.h")
file.show(rinternals)
```

所有的函数名都有前缀 `Rf_` 或者 `R_`，但是导出后则不再含有这些前缀（除非应用了 `#define R_NO_REMAP`）。

我不建议用 C 来编写新的高性能代码。相反应该用 Rcpp 编写 C++。Rcpp API 保护你不受历史遗留的 R API 缺点的影响，它为你进行内存管理，并且提供许多有用的帮助方法。

主要内容

- 20.1 节介绍应用 inline 包来创建和调用 C 函数的基础知识。
- 20.2 节介绍如何把 R 的数据结构名字翻译为 C 语言。
- 20.3 节讲授如何在 C 语言中建立、修改和强制转换向量。
- 20.4 节介绍如何应用成对列表。因为成对列表与列表之间的区别在 C 中比在 R 中更重要，所以你需要了解它们。
- 20.5 节介绍输入验证的重要性，这样你的 C 函数就不会崩溃了。
- 20.6 节通过介绍如何为内部和原始 R 函数寻找 C 源代码来结束本章。

基本要求

为了理解现有的 C 代码，你自己创建简单的可以试验的例子是很有用的。为此，本章

中的所有例子都应用 inline 添加包，该包使得在现有的 R 会话中编译和链接 C 代码尤其方便。通过运行命令 install.packages("inline") 来安装这个包。为了方便地找到与内部和原始函数相关联的 C 代码，需要用到添加包 pryr 中的一个函数。通过命令 install.packages("pryr") 安装该添加包。

还需要一个 C 编译器。Windows 用户可以使用 Rtools（http://cran.r-project.org/bin/widows/Rtools/）。Mac 用户需要 Xcode 命令行工具（http://developer.apple.com/）。大多数 Linux 发行版都带有必需的编译器。

在 Windows 下，必须确保 Rtools 可执行文件的目录（一般为 C:\Rtools\bin）和 C 编译器可执行文件的目录（一般为 C:\Rtools\gcc-4.6.3\bin）都包含在 Windows 的 PATH 环境变量中。在 R 可以识别这些值前，需要重新启动 Windows。

20.1 从 R 中调用 C 函数

一般而言，从 R 中调用 C 函数需要两个函数：一个 C 函数和一个应用 .Call() 进行封装的 R 函数。下面的简单函数把两个数相加，并说明了 C 语言中编码的复杂性。 432

```
// In C ------------------------------------
#include <R.h>
#include <Rinternals.h>

SEXP add(SEXP a, SEXP b) {
  SEXP result = PROTECT(allocVector(REALSXP, 1));
  REAL(result)[0] = asReal(a) + asReal(b);
  UNPROTECT(1);

  return result;
}

# In R ------------------------------------
add <- function(a, b) {
  .Call("add", a, b)
}
```

（.Call 的一个替代是使用 .External。除了 C 函数接受一个包含 LISTSXP 的参数，参数可以从一个成对列表提取外，.External 和 .Call 的用法几乎相同。这使得编写一个接受可变参数数目的函数成为可能。然而，.External 函数在基础 R 中不常用，并且 inline 目前还不支持该函数，所以本章不再讨论该函数。）

在本章中，应用 inline 添加包，在一步中产生两个代码段。这允许我们编写：

```
add <- cfunction(c(a = "integer", b = "integer"),
  SEXP result = PROTECT(allocVector(REALSXP, 1));
  REAL(result)[0] = asReal(a) + asReal(b);
  UNPROTECT(1);

  return result;
")
add(1, 5)
#> [1] 6
```

在我们开始阅读和编写 C 代码前，我们需要了解一点儿基本数据结构。 433

20.2 C数据结构

在C语言层，所有的R对象都存储在一个共同数据类型SEXP或S表达式中。所有的R对象都是S表达式，所以你建立的每一个C函数都必须返回SEXP类型作为输出，接受SEXP作为输入。（技术上，这是一个指向类型为SEXPREC结构的指针。）SEXP是一个变异类型，它具有所有R的数据结构的子类型。最重要的类型是：

- REALSXP：数值向量
- INTSXP：整数向量
- LGLSXP：逻辑向量
- STRSXP：字符向量
- VECSXP：列表
- CLOSXP：函数（闭包）
- ENVSXP：环境

注：在C语言中，列表称为VECSXP，而不是LISTSXP。这是因为早期的列表实现是像Lisp那样的链表，这种链表现在变为了成对列表。

字符向量比其他原子向量复杂。STRSXP包含一个CHARSXP向量，其中每个CHARSXP指向存储在全局池中的C类型的字符串。这种设计允许单个CHARSXP可以在多个字符向量之间共享，从而减少内存使用。参见18.1节来获取更多信息。

还有适用于不太常用的对象类型的SEXP。

- CPLXSXP：复数向量。
- LISTSXP：成对列表。在R层，对函数参数而言，你仅仅需要关注列表和成对列表的区别，但是在内部它们被应用于许多地方。
- DOTSXP：'...'。
- SYMSXP：名字/符号。
- NILSXP：NULL。

434

并且SEXP用于内部对象，即通常仅仅被C语言函数而非R函数创建和使用的对象：

- LANGSXP：语言构造对象。
- CHARSXP："标量"字符串。
- PROMSXP：约定、惰性计算函数参数。
- EXPRSXP：表达式。

没有容易访问这些名字的内置R函数，但pryr添加包提供了函数sexp_type()：

```
library(pryr)

sexp_type(10L)
#> [1] "INTSXP"
sexp_type("a")
#> [1] "STRSXP"
sexp_type(T)
#> [1] "LGLSXP"
sexp_type(list(a = 1))
#> [1] "VECSXP"
sexp_type(pairlist(a = 1))
#> [1] "LISTSXP"
```

20.3 创建和修改向量

每一个 C 函数的核心是 R 数据结构和 C 数据结构之间的转换。输入和输出总是 R 的数据结构（SEXP），做任何工作都需要把它们转换为 C 数据结构。本节重点学习向量，因为它们是你最可能用到的对象类型。

需要额外编译的是垃圾回收器：如果你不保护你创建的每一个 R 对象，垃圾回收器会认为它们没有用并删除它们。

20.3.1 创建向量和垃圾回收

创建一个新的 R 层对象的最简单方法是应用 allocVector()。该函数有两个参数，需要创建的 SEXP（或者 SEXPTYPE）的类型，以及向量的长度。下面的代码创建了含有 3 个元素的列表，它包含一个逻辑向量、一个数值向量和一个整数向量，这 3 个向量的长度都是 4：

435

```
dummy <- cfunction(body = '
  SEXP dbls = PROTECT(allocVector(REALSXP, 4));
  SEXP lgls = PROTECT(allocVector(LGLSXP, 4));
  SEXP ints = PROTECT(allocVector(INTSXP, 4));

  SEXP vec = PROTECT(allocVector(VECSXP, 3));
  SET_VECTOR_ELT(vec, 0, dbls);
  SET_VECTOR_ELT(vec, 1, lgls);
  SET_VECTOR_ELT(vec, 2, ints);

  UNPROTECT(4);
  return vec;
')
dummy()
#> [[1]]
#> [1] 6.941e-310 6.941e-310 6.941e-310  0.000e+00
#>
#> [[2]]
#> [1] TRUE TRUE TRUE TRUE
#>
#> [[3]]
#> [1]     2     4     8 32710
```

你也许疑惑所有的 PROTECT() 调用是做什么的。如果垃圾回收器被激活，它们告诉 R 这个对象正在使用且不要删除。（我们不需要保护 R 已经知道的我们正在使用的对象，如函数参数。）

你也需要确保每一个被保护的对象是无保护的。UNPROTECT() 接受一个整数参数 n，解除对被保护的最后 *n* 个对象的保护。保护和解除保护必须匹配。否则，R 会给出一个警告“.Call 的栈不平衡”。在某些情况下，需要其他特殊形式的保护。

- UNPROTECT_PTR() 解除对于由 SEXP 指向的对象的保护。
- PROTECT_WITH_INDEX() 保存保护对象地址的索引，使用 REPROTECT()，该索引可以用来代替被保护的值。

参考 R 外部扩展部分获取更多关于垃圾回收的信息（http://cran.r-project.org/doc/manuals/R-exts.html#Garbage-Collection）。

436

正确保护你分配的 R 对象是特别重要的！不正确的保护会导致很难诊断错误，一般是段

错误（segfault），也可能是其他冲突。一般而言，如果你分配一个新的R对象，你必须保护（PROTECT）它。

如果你运行多次dummy()，你会注意到输出会有所变化。这是因为allocVector()会为每一个输出分配内存，但是它不会首先清理内存。对于真实的函数，你可能想通过循环遍历向量的每一个元素并设置它为常数。这样做最有效的方法是应用memset()：

```
zeroes <- cfunction(c(n_ = "integer"), '
  int n = asInteger(n_);

  SEXP out = PROTECT(allocVector(INTSXP, n));
  memset(INTEGER(out), 0, n * sizeof(int));
  UNPROTECT(1);

  return out;
')
zeroes(10);
#>  [1] 0 0 0 0 0 0 0 0 0 0
```

20.3.2 缺失值和非有限值

每一个原子向量都有一个用于获取或者设置缺失值的特殊常量。

- INTSXP：NA_INTEGER
- LGLSXP：NA_LOGICAL
- STRSXP：NA_STRING

对于REALSXP，缺失值尤其复杂，因为存在一个由浮点数标准（IEEE 754（http://en.wikipedia.org/wiki/IEEE_floating_point））定义的缺失值的现有协议。对于双精度浮点数，NA就是一个具有特殊位模式的NaN（最低字是1 954，即Ross Ihaka诞生之年）。对于正无穷大和负无穷大，还有其他特殊值。应用宏ISNA()、ISNAN()和!R_FINITE()来检查缺失值、NaN或者非有限值。应用常量NA_REAL、R_NaN、R_PosInf和R_NegInf来设置这些值。

437

可以应用这些知识来编写一个简单版本的is.na()：

```
is_na <- cfunction(c(x = "ANY"), '
  int n = length(x);

  SEXP out = PROTECT(allocVector(LGLSXP, n));

  for (int i = 0; i < n; i++) {
    switch(TYPEOF(x)) {
      case LGLSXP:
        LOGICAL(out)[i] = (LOGICAL(x)[i] == NA_LOGICAL);
        break;
      case INTSXP:
        LOGICAL(out)[i] = (INTEGER(x)[i] == NA_INTEGER);
        break;
      case REALSXP:
        LOGICAL(out)[i] = ISNA(REAL(x)[i]);
        break;
      case STRSXP:
        LOGICAL(out)[i] = (STRING_ELT(x, i) == NA_STRING);
        break;
      default:
```

```
        LOGICAL(out)[i] = NA_LOGICAL;
      }
    }
    UNPROTECT(1);

    return out;
')
is_na(c(NA, 1L))
#> [1]  TRUE FALSE
is_na(c(NA, 1))
#> [1]  TRUE FALSE
is_na(c(NA, "a"))
#> [1]  TRUE FALSE
is_na(c(NA, TRUE))
#> [1]  TRUE FALSE
```

注意，对于数值向量中的 NA 和 NaN，base::is.na() 都返回 TRUE；这与 C 语言中的宏 ISNA() 不同，它只对 NA_REAL 返回 TRUE。 438

20.3.3 访问向量数据

每一个原子向量都有一个帮助函数，它使你可以访问 C 数组，该 C 数组把数据存储在一个向量中。应用 REAL()、INTEGER()、LOGICAL()、COMPLEX() 和 RAW() 访问数值、逻辑值、复数和原始向量中的 C 数组。下面的例子说明如何应用 REAL() 检查和修改数值向量：

```
add_one <- cfunction(c(x = "numeric"), "
  int n = length(x);
  SEXP out = PROTECT(allocVector(REALSXP, n));

  for (int i = 0; i < n; i++) {
    REAL(out)[i] = REAL(x)[i] + 1;
  }
  UNPROTECT(1);

  return out;
")
add_one(as.numeric(1:10))
#>  [1]  2  3  4  5  6  7  8  9 10 11
```

当对长向量进行运算时，使用一次帮助函数并把结果保存在一个指针中具有性能上的优势。

```
add_two <- cfunction(c(x = "numeric"), "
  int n = length(x);
  double *px, *pout;

  SEXP out = PROTECT(allocVector(REALSXP, n));

  px = REAL(x);
  pout = REAL(out);
  for (int i = 0; i < n; i++) {
    pout[i] = px[i] + 2;
  }
  UNPROTECT(1);

  return out;
")
add_two(as.numeric(1:10))
#>  [1]  3  4  5  6  7  8  9 10 11 12
```

```
library(microbenchmark)
x <- as.numeric(1:1e6)
microbenchmark(
  add_one(x),
  add_two(x)
)
#> Unit: milliseconds
#>        expr   min    lq median    uq   max neval
#>  add_one(x) 4.950 5.093  6.199 7.571 41.81   100
#>  add_two(x) 1.106 1.220  2.309 3.673 34.88   100
```

在我的计算机上，对于一个 100 万个元素的向量，add_two() 大约比 add_one() 快 2 倍。在基础 R 中这是一个常用习惯。

20.3.4 字符向量和列表

字符串和列表更复杂，因为向量的单个元素是 SEXP 而不是基础的 C 数据结构。STRSXP 的每一个元素都是 CHARSXP，CHARSXP 是不可变对象，它包含一个指向存储在全局池中的 C 字符串的指针。应用 STRING_ELT(x, i) 来提取 CHARSXP，应用 CHAR(STRING_ELT(x, i)) 来获取实际的 const char* 字符串。用 SET_STRING_ELT(x, i, value) 来设置值。应用 mkChar() 将 C 字符串转变为 CHARSXP；用 mkString() 把 C 字符串转变为 STRSXP。应用 mkChar() 创建字符串插入已有的向量中，应用 mkString() 创建一个长度为 1 的新向量。

下面函数说明如何创建一个包含已知字符串的字符向量。

```
abc <- cfunction(NULL, '
  SEXP out = PROTECT(allocVector(STRSXP, 3));

  SET_STRING_ELT(out, 0, mkChar("a"));
  SET_STRING_ELT(out, 1, mkChar("b"));
  SET_STRING_ELT(out, 2, mkChar("c"));

  UNPROTECT(1);

  return out;
')
abc()
#> [1] "a" "b" "c"
```

439
〜
440

如果你想要修改向量中的字符串，可能有些困难，因为你需要知道 C 中字符串操作的大量知识（比较难，做对更难）。对于任何有关字符串修改的问题，最好应用 Rcpp。

一个列表的元素可以是任何其他的 SEXP，这通常使得它们更难以与 C 一起工作（你需要大量 switch 语句来处理各种可能性）。列表的访问函数是 VECTOR_ELT(x, i) 和 SET_VECTOR_ELT(x, i, value)。

20.3.5 修改输入

当你修改函数输入时，你必须非常小心。下面的函数具有一些难以预料的行为：

```
add_three <- cfunction(c(x = "numeric"), '
  REAL(x)[0] = REAL(x)[0] + 3;
  return x;
```

```
')
x <- 1
y <- x
add_three(x)
#> [1] 4
x
#> [1] 4
y
#> [1] 4
```

它不仅修改了 x 的值，而且它还修改了 y 的值。这种情况发生是由于 R 的惰性复制后修改语义。为了避免出现这样的问题，在修改它们前总是 duplicate() 输入。

```
add_four <- cfunction(c(x = "numeric"), '
  SEXP x_copy = PROTECT(duplicate(x));
  REAL(x_copy)[0] = REAL(x_copy)[0] + 4;
  UNPROTECT(1);
  return x_copy;
')
x <- 1
y <- x
add_four(x)
#> [1] 5
x
#> [1] 1
y
#> [1] 1
```

如果你正在应用列表，使用 shallow_duplicate() 进行浅复制。duplicate() 也会复制列表中的每一个元素。

20.3.6 强制转换标量

有几个帮助函数可以把长度为 1 的 R 向量转换为 C 标量。

- asLogical(x): INTSXP ->int
- asInteger(x): INTSXP ->int
- asReal(x): REALSXP -> double
- CHAR(asChar(x)): STRSXP ->const char*

还有做相反方向运算的帮助函数。

- ScalarLogical(x): int -> LGLSXP
- ScalarInteger(x): int -> INTSXP
- ScalarReal(x): double -> REALSXP
- mkString(x): const char* -> STRSXP

这些都创建了 R 层的对象，所以它们需要被 PROTECT()。

20.3.7 长向量

截至 R 3.0.0，R 向量的长度可以超过 2^{32}。这意味着这样长的向量不能再可靠地存储在一个 int 中，如果你希望你的代码能够处理长向量，你不能写如下的代码：int n = length(x)。相反，应该应用 R_xlen_t 类型和 xlength() 函数，并写为 R_xlen_t n = xlength(x)。

441 ~ 442

20.4 成对列表

在R代码中，只有极少数的情况需要关注成对列表与列表的区别（参见14.5节）。而在C中，成对列表起了更加重要的作用，因为它们用于调用、未计算的参数、属性以及...中。在C中，列表和成对列表的主要区别在于如何访问和命名元素。

与列表（VECSXP）不同，成对列表（LISTSXP）无法对任意地址进行索引。相反，R提供了一系列的帮助函数，它们可以遍历链表。基本的帮助函数有：CAR()，用于提取列表的第一个元素；CDR()，用于提取列表的其他元素。把二者组合，可以得到CAAR()、CDAR()、CADDR()、CADDDR()等。与这些获取元素的函数对应的是设置元素的函数，R提供的设置函数有SETCAR()、SETCDR()等。

下面的例子展示了如何应用CAR()和CDR()从引用函数调用中提取信息：

```
car <- cfunction(c(x = "ANY"), 'return CAR(x);')
cdr <- cfunction(c(x = "ANY"), 'return CDR(x);')
cadr <- cfunction(c(x = "ANY"), 'return CADR(x);')

x <- quote(f(a = 1, b = 2))
# The first element
car(x)
#> f
# Second and third elements
cdr(x)
#> $a
#> [1] 1
#>
#> $b
#> [1] 2
# Second element
car(cdr(x))
#> [1] 1
cadr(x)
#> [1] 1
```

成对列表总是以R_NilValue结尾。应用下面的模板来循环遍历一个成对列表的所有元素：

```
count <- cfunction(c(x = "ANY"), '
  SEXP el, nxt;
  int i = 0;

  for(nxt = x; nxt != R_NilValue; el = CAR(nxt), nxt = CDR(nxt)) {
    i++;
  }
  return ScalarInteger(i);
')
count(quote(f(a, b, c)))
#> [1] 4
count(quote(f()))
#> [1] 1
```

你可以用CONS()创建新的成对列表，用LCONS()创建新的调用。记住设置最后一个值为R_NilValue。由于这些也是R对象，所以它们也适用于垃圾回收并且必须被保护（PROTECT）。事实上，下面的代码是不安全的：

```
new_call <- cfunction(NULL, '
  return LCONS(install("+"), LCONS(
    ScalarReal(10), LCONS(
      ScalarReal(5), R_NilValue
    )
  ));
')
gctorture(TRUE)
new_call()
#> 5 + 5
gctorture(FALSE)
```

在我的机器上，我得到结果 `5 + 5`——这绝对出乎意料。事实上，因为每一个 R 对象分配都能触发垃圾回收器，所以为了安全起见，我们必须保护（`PROTECT`）生成的每一个 `ScalarReal`。

```
new_call <- cfunction(NULL, '
  SEXP REALSXP_10 = PROTECT(ScalarReal(10));
  SEXP REALSXP_5 = PROTECT(ScalarReal(5));
  SEXP out = PROTECT(LCONS(install("+"), LCONS(
    REALSXP_10, LCONS(
      REALSXP_5, R_NilValue
    )
  )));
  UNPROTECT(3);
  return out;
')
gctorture(TRUE)
new_call()
#> 10 + 5
gctorture(FALSE)
```

`TAG()` 和 `SET_TAG()` 允许你获取和设置与成对列表的元素相关联的标签（或者名字）。标签应该是一个符号。应用 `install()` 创建一个符号（R 中的等价命令为 `as.symbol()`）。

属性也是成对列表，但是它们有帮助函数 `setAttrib()` 和 `getAttrib()`：

```
set_attr <- cfunction(c(obj = "SEXP", attr = "SEXP", value = "SEXP"), '
  const char* attr_s = CHAR(asChar(attr));

  duplicate(obj);
  setAttrib(obj, install(attr_s), value);
  return obj;
')
x <- 1:10
set_attr(x, "a", 1)
#>  [1]  1  2  3  4  5  6  7  8  9 10
#> attr(,"a")
#> [1] 1
```

（注意 `setAttrib()` 和 `getAttrib()` 必须对属性成对列表进行线性搜索。）

有一些快捷方式（名字有些混乱）进行常见的设置操作：`classgets()`、`namesgets()`、`dimgets()` 和 `dimnamesgets()` 是 `class<-`、`names<-`、`dim<-` 和 `dimnames<-` 的默认方法的内部版本。

20.5 输入验证

443 ~ 445

如果用户为函数提供了意想不到的输入（例如，不是数值向量而是一个列表），这容易使

R 崩溃。为此，编写一个封装函数来检查参数是否为正确类型，是一个好主意。在 R 层上这通常容易做到。例如，回到我们 C 代码的第一个例子，我们可以重新命名 C 函数为 add_，并且为它编写一个封装函数，用来检查输入的正确性：

```
add_ <- cfunction(signature(a = "integer", b = "integer"), "
  SEXP result = PROTECT(allocVector(REALSXP, 1));
  REAL(result)[0] = asReal(a) + asReal(b);
  UNPROTECT(1);

  return result;
")
add <- function(a, b) {
  stopifnot(is.numeric(a), is.numeric(b))
  stopifnot(length(a) == 1, length(b) == 1)
  add_(a, b)
}
```

或者，如果我们需要接受多样化的输入，我们可以如下这样做：

```
add <- function(a, b) {
  a <- as.numeric(a)
  b <- as.numeric(b)

  if (length(a) > 1) warning("Only first element of a used")
  if (length(b) > 1) warning("Only first element of b used")

  add_(a, b)
}
```

应用命令 PROTECT(new = coerceVector(old, SEXPTYPE)) 来强制转换 C 层的对象。如果 SEXP 不能转换为期望的类型，这将返回一个错误。

为了检查对象是否为一个具体类型，可以应用 TYPEOF，它返回 SEXPTYPE：

```
is_numeric <- cfunction(c("x" = "ANY"), "
  return ScalarLogical(TYPEOF(x) == REALSXP);
")
is_numeric(7)
#> [1] TRUE
is_numeric("a")
#> [1] FALSE
```

还有许多帮助函数，当 FALSE 时返回 0，TRUE 时返回 1：

- 原子向量：isInteger()、isReal()、isComplex()、isLogical()、isString()。
- 原子向量的组合：isNumeric()（整数、逻辑和实数）、isNumber()（整数、逻辑、实数和复数）、isVectorAtomic()（逻辑、整数、数值、复数、字符串和原始）。
- 矩阵（isMatrix()）和数组（isArray()）。
- 更复杂的对象：isEnvironment()、isExpression()、isList()（成对列表）；isNewList()（列表）；isSymbol()、isNull()、isObject()（S4 对象）；isVector()（原子向量、列表和表达式）。

注意这些函数中的一部分与命名方式相似的具有相似名字的 R 函数的行为不同。例如，对于原子向量、列表和表达式 isVector() 返回 TURE，而 is.vector() 只有它的输入除了具有名字外没有属性时才返回 TRUE。

20.6 寻找一个函数的 C 源代码

在基础包中，R 不使用 .Call()。相反，它使用两个特殊函数：.Internal() 和 .Primitive()。寻找这些函数的源代码是一项艰巨的任务：你首先需要在 src/main/names.c 中寻找它们的 C 函数名，然后再搜索 R 源代码。函数 pryr::show_c_source() 应用 GitHub 代码搜寻方法来自动化这个任务：

```
tabulate
#> function (bin, nbins = max(1L, bin, na.rm = TRUE))
#> {
#>     if (!is.numeric(bin) && !is.factor(bin))
#>         stop("'bin' must be numeric or a factor")
#>     if (typeof(bin) != "integer")
#>         bin <- as.integer(bin)
#>     if (nbins > .Machine$integer.max)
#>         stop("attempt to make a table with >= 2^31 elements")
#>     nbins <- as.integer(nbins)
#>     if (is.na(nbins))
#>         stop("invalid value of 'nbins'")
#>     .Internal(tabulate(bin, nbins))
#> }
#> <bytecode: 0x7fc69d9930b0>
#> <environment: namespace:base>

pryr::show_c_source(.Internal(tabulate(bin, nbins)))
#> tabulate is implemented by do_tabulate with op = 0
```

这给出了下面的 C 源代码（为了清楚，进行了编辑）：

```
SEXP attribute_hidden do_tabulate(SEXP call, SEXP op, SEXP args,
                                  SEXP rho) {
  checkArity(op, args);
  SEXP in = CAR(args), nbin = CADR(args);
  if (TYPEOF(in) != INTSXP)  error("invalid input");

  R_xlen_t n = XLENGTH(in);
  /* FIXME: could in principle be a long vector */
  int nb = asInteger(nbin);
  if (nb == NA_INTEGER || nb < 0)
    error(_("invalid '%s' argument"), "nbin");

  SEXP ans = allocVector(INTSXP, nb);
  int *x = INTEGER(in), *y = INTEGER(ans);
  memset(y, 0, nb * sizeof(int));
  for(R_xlen_t i = 0 ; i < n ; i++) {
    if (x[i] != NA_INTEGER && x[i] > 0 && x[i] <= nb) {
      y[x[i] - 1]++;
    }
  }

  return ans;
}
```

446 ~ 448

内部函数和原函数的接口与 .Call() 函数的接口有些不同。它们总是有 4 个参数：

- SEXP call：对函数完全的调用。CAR(call) 给出函数的名字（作为一个符号），

CDR(call)给出参数。

- SEXP op：偏移指针。它用于当多个R函数使用同一个C函数时。例如，do_logic()实现了&、|和!。show_c_source()为你将它打印出来。
- SEXP args：成对列表，它包含函数的未计算的参数。
- SEXP rho：调用执行的环境。

这给内部函数提供了如何以及何时对参数进行计算的极大灵活性。例如，内部S3泛型函数调用DispatchOrEval()，它或者调用适当的S3方法或者在原地对所有参数进行计算。这种灵活性是有代价的，因为它使代码阅读起来更加困难。然而，参数计算通常是第一步，函数的剩余部分是很简单的。

下面的代码说明了如何将do_tabulate()转换为一个标准的.Call()接口：

```
tabulate2 <- cfunction(c(bin = "SEXP", nbins = "SEXP"), '
  if (TYPEOF(bin) != INTSXP)  error("invalid input");

  R_xlen_t n = XLENGTH(bin);
  /* FIXME: could in principle be a long vector */
  int nb = asInteger(nbins);
  if (nb == NA_INTEGER || nb < 0)
    error("invalid \'%s\' argument", "nbin");

  SEXP ans = allocVector(INTSXP, nb);
  int *x = INTEGER(bin), *y = INTEGER(ans);
  memset(y, 0, nb * sizeof(int));
  for(R_xlen_t i = 0 ; i < n ; i++) {
    if (x[i] != NA_INTEGER && x[i] > 0 && x[i] <= nb) {
      y[x[i] - 1]++;
    }
  }

  return ans;
')
tabulate2(c(1L, 1L, 1L, 2L, 2L), 3)
#> [1] 3 2 0
```

为了编译上面代码，我把调用移到_()，这是一个用于在不同语言之间翻译错误信息的R内部函数。

下面的最后版本把更多的强制转换逻辑移入一个伴随R函数中，并进行了一些较小的重新结构化以便使代码更容易理解。我还添加了一个PROTECT()；在原始函数中这也许漏掉，因为作者知道那样做是安全的。

```
tabulate_ <- cfunction(c(bin = "SEXP", nbins = "SEXP"), '
  int nb = asInteger(nbins);

  // Allocate vector for output - assumes that there are
  // less than 2^32 bins, and that each bin has less than
  // 2^32 elements in it.
  SEXP out = PROTECT(allocVector(INTSXP, nb));
  int *pbin = INTEGER(bin), *pout = INTEGER(out);
  memset(pout, 0, nb * sizeof(int));

  R_xlen_t n = xlength(bin);
  for(R_xlen_t i = 0; i < n; i++) {
```

```
    int val = pbin[i];
    if (val != NA_INTEGER && val > 0 && val <= nb) {
      pout[val - 1]++; // C is zero-indexed
    }
  }
  UNPROTECT(1);

  return out;
')

tabulate3 <- function(bin, nbins) {
  bin <- as.integer(bin)
  if (length(nbins) != 1 || nbins <= 0 || is.na(nbins)) {
    stop("nbins must be a positive integer", call. = FALSE)
  }
  tabulate_(bin, nbins)
}
tabulate3(c(1, 1, 1, 2, 2), 3)
#> [1] 3 2 0
```

449
〜
450

索　引

索引中的页码为英文原书页码，与书中页边标注的页码一致。

高级R语言编程指南

ADVANCED R

本书作者是R语言及相关领域的领军人物，为众多R语言社区的成员所熟知和欣赏。这就是本书最显著的优势。虽然越来越多的书籍在讨论本书讲述的主题，但是很少有相关书籍拥有同样的深度、技术上的准确性和权威性。

—— **Bill Venables**, CSIRO

本书为解决各种R语言编程问题提供了非常有用的工具和技术，帮助你少走弯路。书中阐释了R语言简洁、优美以及灵活的核心特点，展示了许多必备的技巧，通过它们可以创建在各个场景中使用的优质代码。

本书主要内容

- R语言基础知识，包括标准数据类型和函数。
- 将函数式编程作为有用的框架来解决多类问题。
- 元编程的优点与缺点。
- 编写快速及节省内存的程序。

本书展示了R的特别之处，为R软件使用者成为R程序员奠定了基础。中级R程序员则可以通过本书更深入地研究R语言，学习新策略来解决各种问题。而其他语言的程序员可以通过本书细致地了解R语言并理解R语言的运行方式。

作者简介

哈德利·威克汉姆（Hadley Wickham） R语言及相关领域的领军人物，RStudio首席科学家，美国莱斯大学统计学助理教授。他拥有超过10年的R语言编程经验，致力于开发用于数据处理、分析、成像的工具。他已经开发了几十个高质量的R软件包，如ggplot、lubridate、plyr、reshape2、 stringr和 httr等。

CRC Press
Taylor & Francis Group

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指导：计算机/R语言
ISBN 978-7-111-54067-0
9 787111 540670 >
定价：79.00元