

2015 版

R 语言高级程序设计

——《Advanced R》中文版

原著者: Hadley Wickham
译者: 刘宁

修改日期: 2015-09-01

目 录

前言	9
译者简介	9
译者序	10
中文版版权声明	11
第一部分 基础知识	12
1 介绍	12
1.1 谁应该阅读本书?	14
1.2 你在本书中能学到什么?	15
1.3 元技术	16
1.4 推荐阅读	16
1.5 得到帮助	17
1.6 鸣谢	18
1.7 约定	18
1.8 版权声明	19
2 数据结构	20
2.1 向量	21
2.2 属性	27
2.3 矩阵和数组	33

2.4 数据框	37
2.5 答案	41
3 取子集操作	43
3.1 数据类型	44
3.2 取子集操作符	52
3.3 取子集与赋值	57
3.4 应用	59
3.5 答案	70
4 词汇表	71
4.1 基础	71
4.2 通用数据结构	74
4.3 统计学	75
4.4 使用 R 语言工作	76
4.5 输入/输出	77
5 编码风格指南	79
5.1 标识符和命名	79
5.2 语法	81
5.3 组织	85
6 函数	86

6.1 函数的组成部分	88
6.2 词法作用域	90
6.3 所有的操作都是函数调用	97
6.4 函数参数	100
6.5 特殊调用	110
6.6 返回值	114
6.7 小测验答案	120
7 面向对象指南	121
7.1 基本类型	123
7.2 S3	124
7.3 S4	134
7.4 引用类	141
7.5 选择一种系统	145
7.6 小测验答案	146
8 环境	147
8.1 环境基础	148
8.2 在环境中进行递归	156
8.3 函数环境	159
8.4 把名字绑定到值上	170

8.5 显式环境.....	174
8.6 小测验答案	176
9 调试、条件处理和防御性编程.....	178
9.1 调试技术.....	180
9.2 调试工具.....	182
9.3 条件处理.....	191
9.4 防御性编程	201
9.5 小测验答案	203
第二部分 函数式编程	205
10 函数式编程.....	205
10.1 使用函数式编程的动机.....	206
10.2 匿名函数.....	212
10.3 闭包.....	214
10.4 函数列表.....	221
10.5 案例研究：数值积分	227
11 泛函.....	231
11.1 我的第一个泛函: lapply()	233
11.2 for 循环泛函: lapply()的朋友们.....	238
11.3 操作矩阵和数据框.....	249

11.4 操作列表.....	256
11.5 数学泛函.....	259
11.6 循环应该被保留的情况.....	262
11.7 函数族.....	265
12 函数运算符.....	273
12.1 行为函数运算符.....	275
12.2 输出函数运算符.....	287
12.3 输入函数运算符.....	291
12.4 联合函数运算符.....	295
第三部分 编程语言层面的计算.....	301
13 非标准计算.....	301
13.1 捕获表达式.....	303
13.2 子集中的非标准计算.....	305
13.3 作用域问题.....	310
13.4 从另一个函数进行调用.....	313
13.5 substitute()	318
13.6 非标准计算的缺点.....	323
14 表达式.....	326
14.1 表达式的结构.....	326

14.2 名字.....	332
14.3 调用.....	334
14.4 捕获当前调用	339
14.5 成对列表.....	343
14.6 解析与逆解析.....	347
14.7 使用递归函数遍历抽象语法树.....	349
15 领域特定语言	362
15.1 HTML.....	363
15.2 LaTeX.....	373
第四部分 性能	384
16 性能.....	384
16.1 R 语言为什么慢?	385
16.2 微基准测试.....	386
16.3 语言性能.....	388
16.4 实现性能.....	395
16.5 其它的 R 语言实现.....	399
17 优化代码.....	404
17.1 测量性能.....	406
17.2 改善性能.....	411

17.3 代码组织.....	411
17.4 有人已经解决了这个问题吗?	413
17.5 尽量少做.....	415
17.6 向量化.....	424
17.7 避免复制.....	426
17.8 编译成字节码.....	428
17.9 案例研究: t 检验	429
17.10 并行化.....	432
17.11 其它技术.....	434
18 内存.....	436
18.1 对象的大小.....	437
18.2 内存使用和垃圾收集.....	443
18.3 内存分析与 lineprof.....	446
18.4 就地修改.....	450
19 使用 Rcpp 包编写高性能函数.....	457
19.1 开始使用 C++.....	459
19.2 属性和其它的类.....	470
19.3 缺失值.....	474
19.4 Rcpp 语法糖.....	478

19.5 标准模板库.....	481
19.6 案例研究.....	490
19.7 在包中使用 Rcpp.....	495
19.8 更多学习资源.....	496
19.9 鸣谢.....	498
20 R 的 C 语言接口.....	499
20.1 在 R 中调用 C 语言函数.....	501
20.2 C 语言的数据结构.....	502
20.3 创建和修改向量.....	504
20.4 成对列表.....	513
20.5 输入验证.....	516
20.6 找到一个函数的 C 源代码.....	518

前言

译者简介



刘宁，现任某互联网金融公司技术总监，资深软件分析员，R 语言专家，《R 语言基础编程技巧汇编》作者，2007 年毕业于浙江大学计算机科学与技术专业，获硕士学位，长期供职于大型跨国 IT 企业，主要从事软件研发、系统架构、数据分析、数据挖掘、数据可视化等相关技术的咨询和管理工作，为国内外各大企业提供专业的软件解决方案，涉及的行业包括互联网、金融、能源、石油化工、电力、造船、道路桥梁等，涉及的国家 and 地区包括中国大陆地区、台湾地区、韩国、新加坡和美国等。

可通过下列方式联系译者：

QQ: 59739150

E-mail: liuning.1982@qq.com

欢迎访问译者的博客: <http://blog.csdn.net/liu7788414>

欢迎加入本书的 qq 群: 485732827（只接受中高级 R 语言开发者，不讨论初级编程问题。初级开发者请加入此群：R 语言基础编程技巧 437199880）

译者序

本书原著者 **Hadley Wickham** 是 R 语言领域屈指可数的顶尖专家，他开发了 R 语言中大量非常重要的包，比如 **ggplot2**、**plyr**、**reshape2** 等等，可以说没有哪个 R 语言开发者没有使用过他开发的函数。本书是 Hadley Wickham 根据他多年的 R 语言编程经验编写的**经典著作**，主要介绍了**R 语言的本质**，也是国外 R 语言开发人员必读的**核心书籍**之一。是否掌握了本书的内容，也是判断 R 语言开发者水平的重要标准之一，可以认为：掌握了本书内容的使用者是**专业的 R 语言程序员**，否则就是**业余的 R 语言用户**。本书对于**加深对 R 语言的理解，提高 R 语言的开发水平**，具有很大的作用。

但是，由于本书具有一定的难度，并且原书又是英文撰写的，因此，本书在国内的 R 语言开发人员中普及程度很低，大多数 R 语言开发者甚至没有听说过本书，这导致了国内 R 语言开发者的水平与国外开发者相比，存在很大的差距。

如今，市面上有很多 R 语言的相关书籍，但是内容基本上都是将 R 语言直接应用到某个具体的领域，而很少介绍 R 语言本身，以及它的各种高级特性。用武侠小说中的语言来说，就是**只重招式而不重内力修为**。而本书就是像《九阳神功》这种可以提高读者内力的著作，一旦你理解了本书的内容，一定会有**"一览众山小"**的感觉，阅读其它书籍会觉得相当轻松。

虽然本书名为《Advanced R》，看似只针对高级开发人员，但是我**建议所有的 R 语言开发者，都应该读一读本书，一定会有收获**。

如果对本书内容有任何意见和建议，可以与本人联系，联系方式见"译者简介"一节。

中文版版权声明

本书原文全部来自于原著者 Hadley Wickham 的网站 <http://adv-r.had.co.nz/Introduction.html>，为完全公开的内容。原书版权完全归属原著者 Hadley Wickham。本人的工作主要是对原书进行翻译、补充、校验、批注和排版，以便更适合国内 R 语言开发者阅读，希望能帮助国内 R 语言开发者提高开发水平。

第一部分 基础知识

1 介绍

我(Hadley Wickham)有超过 10 年的 R 语言编程经验，所以我能花大量时间进行实验，理解 R 语言是如何工作的。在本书中，我试图向你传达我所理解的内容，帮助你能够很快地成为一名**高效的 R 程序员**。阅读本书将帮助你避免我所犯过的错误和碰到过的死胡同，本书会教你一些有用的**工具、技术和术语**，它们可以帮助你解决各种类型的问题。尽管 R 语言确实看起来有些怪异，但是在这个过程中，我希望向你展示 R 语言本质上是一种**优雅且迷人的语言**，它是专为**数据分析和统计学**设计的。

如果你是刚刚接触 R 语言，那么你可能想知道为什么要学习这样一种古怪的语言，这值得吗？对我来说，R 语言有一些很棒的优点：

1. 它是**免费的**，也是**开源的**，并且在每一种主要平台上都有相应的实现。因此，如果你使用 R 语言进行分析，那么任何人都可以很容易地进行复制。
2. 它具有大量关于**统计建模、机器学习、可视化以及导入数据和操作数据的包**。无论你是否正在尝试拟合什么模型或者绘制什么图形，很可能有人已经尝试做过了。就算不是这样，你至少可以从他们的工作中学习到一些经验。
3. 它有**最前沿**的工具。统计学和机器学习的研究人员，经常会为他们的论文发布一个新的 R 包。这意味着，你可以立即使用到最新的统计技术及其实现。
4. 对于数据分析，它有**深度的语言支持**。这包括缺失值、数据框以及**取子集操作**等特性。
5. 它有一个**神奇的社区**。我们可以很容易地从 R-help 邮件列表(<https://stat.ethz.ch/mailman/listinfo/r-help>)、

stackoverflow(<http://stackoverflow.com/questions/tagged/r>)或者特定主题的邮件列表(如 R-SIG-mixed-models, <https://stat.ethz.ch/mailman/listinfo/r-sig-mixed-models> 或者 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/RProject-Statistical-Computing-77616>)或者许多其它用户组(<http://blog.revolutionanalytics.com/local-rgroups.html>), 来联系其它 R 语言学习者。

6. 它有助于交流结果的强大工具。R 语言包, 使得产生 html 或 pdf 报告(<http://yihui.name/knitr/>), 或者创建交互式网站(<http://www.rstudio.com/shiny/>)变得很容易。
7. 它对函数式编程有强大的支持。函数式编程的思想适合解决许多有挑战的数据分析问题。R 提供了功能强大且灵活的工具包, 它让你可以编写简洁但表现力很强的代码。
8. 它具有适用于交互式数据分析和统计编程的集成开发环境(Integrated Development Environment, IDE)(<http://www.rstudio.com/ide/>)。
9. 它有强大的元编程(metaprogramming)工具。R 语言不仅仅是一种编程语言, 也是一种交互式数据分析环境。它的元编程功能允许你编写紧凑且简洁的函数, 并且为设计领域特定语言提供了优秀的环境。
10. 它被设计成可以连接到像 C、Fortran 和 C++这样的高性能编程语言。

当然, R 语言并不完美。R 语言的最大挑战是, 大多数用户不是程序员。因此:

1. 你会看到有许多 R 代码, 都是为了急于解决一个紧迫的问题而编写的。因此, 那些代码不是很优雅、快速或者容易理解。而大多数用户并不修改他们代码中的这些缺点。

2. 与其它编程语言相比，R 语言社区更倾向于关注结果，而不是过程。软件工程知识的最佳实践是不完整的：例如，很少有 R 程序员使用源代码控制器(译者注：比如 SVN、Github、Team Foundation Server 等等)或者进行自动化测试。
3. 元编程是一把双刃剑。太多的 R 语言函数使用了一些小技巧，以便减少键盘输入，这使得代码变得很难理解，并且可能发生产想不到的失败。
4. 在各种包中，甚至基础 R 语言中，都存在不一致的情况。你面对的是已经进化了超过 20 年的 R 语言。R 语言的学习很困难，因为我们需要记住很多特殊的情况。
5. R 语言不是特别快的编程语言，写得很差 R 语言代码可以变得非常缓慢。R 语言也非常消耗内存。

但是，就我个人而言，我认为这些挑战，为经验丰富的程序员，对 R 语言本身以及在 R 语言社区中，产生深远且积极的影响，创造了一个很好的机会。R 语言用户实际上很关心如何编写高质量的代码，特别是可复用的代码，但是他们还没有能力这样做。我希望本书不仅能帮助更多的 R 语言用户成为 R 语言程序员，而且能鼓励其它语言的程序员为 R 语言作出贡献。

1.1 谁应该阅读本书？

这本书主要针对两类读者：

1. 想更深入地探索 R 语言的**中级 R 语言程序员**，他们可以学习新的策略来解决不同的问题。
2. 正在学习 R 语言的**其它语言程序员**，他们希望理解 R 语言为什么是这样工作的。

为了好好利用这本书，你需要具有一定的 R 语言或者其它编程语言的编码经验。你可能不知道所有的细节，但是你应该熟悉在 R 语言中函数是如何工作的；尽管目前你可能难以有效地使用**泛函(functional)**，但是你应该熟悉 **apply** 族函数(如 **apply()**和 **lapply()**)。

1.2 你在本书中能学到什么？

本书描述了我认为一名高级 R 语言程序员应该拥有的技能：他们具有**生产高质量代码**的能力，并且这些代码可以用于各种各样的情况。

读完了本书，你将会：

1. 熟悉 R 语言的基本原理。你会理解复杂的数据类型，并且学习对它们进行操作的最好方法。你将会对函数是如何工作的有深刻的理解，并且可以认识和使用 R 语言中的四种面向对象系统。
2. 理解什么是**函数式编程**，以及为什么它是进行数据分析的有用的工具。你将可以很快学会使用现有工具，以及在必要的时候创建自己的函数工具的知识。
3. 领会元编程的双刃剑。你能够创建这样的函数：它使用了非标准计算，它减少了键盘输入，以及它使用了优雅的代码来表达重要的操作。你还将了解元编程的危险，以及在使用它时，为什么你应该小心。
4. 对 R 语言中哪些操作很慢，哪些操作很耗内存，能拥有很好的直觉。你将会知道如何使用分析工具来确定性能瓶颈，以及你将学习足够的 C++知识，以便把缓慢的 R 语言函数转化为快速的 C++等价函数。
5. 轻松地阅读并且理解大多数 R 语言代码。你会认识常见的术语(即使你自己并不会使用它们)，并且能够评判别人的代码。

1.3 元技术

有两种**元技术**(Meta-techniques)非常有利于提高 R 语言程序员的技能：**阅读源代码**和**采用科学的思路**。

阅读源代码是很重要的，因为它将帮助你编写出更好的代码。要发展这种能力，一个好的开头是看看你最经常使用的那些函数和包的源代码。你会发现有价值的内容，值得在你自己的代码中进行模仿，并且渐渐地，你能感受到什么样的 R 语言代码才是好的。当然，你也会看到你不喜欢的内容，要么是因为它的优点并不是那么明显，要么是它违背了你的感觉。尽管如此，这样的代码还是有价值的，因为它让你对好的代码和坏的代码有具体的体会。

在学习 R 语言的时候，科学的思维方式非常有用。如果你不理解事物是如何工作的，那么请提出一个假设，然后设计一些实验，接着运行它们，最后记录结果。这种练习是非常有用的，因为如果你不能得出结论，并且需要得到他人的帮助，那么你可以很容易地告诉别人你已经尝试过方法。同样，当你学到了正确的答案，你也会在内心中更新你的知识。当我清楚地把问题向别人描述的时候，(《the art of creating a reproducible example》

(<http://stackoverflow.com/questions/5963269>))，我经常自己就找出了解决方案。

1.4 推荐阅读

R 语言仍然是一种相对年轻的语言，并且它的学习资料还在逐步完善的过程中。在我个人理解 R 语言的过程之中，我发现使用其它编程语言的资源特别有用。R 语言拥有**函数式编程**和**面向对象编程**语言两个方面。学习在 R 语言中如何表达这些概念，将帮助你利用其它编程语言中的已有知识，并且将帮助你识别可以改善的领域。

为了理解为什么 R 语言的**对象系统**要以这种方式工作，我发现 Harold Abelson 和 Gerald Jay Sussman 写的《Structure and Interpretation of Computer Programs》(<http://mitpress.mit.edu/sicp/full-text/book/book.html>)(SICP)特别有用。这是一本简明但深刻的书。在阅读该书之后，我第一次觉得我可以设计出我自己的面向对象系统。该书是我的第一本导论，它介绍了 R 语言中常见的面向对象**泛型函数**风格。该书帮助我理解它的优点和缺点。该书还讲了许多**函数式编程**的内容，以及如何创建简单的函数，它们在联合使用时会变得非常强大。

为了理解 R 语言相比于其它编程语言的优缺点，我发现 Peter van Roy 和 Sef Haridi 写的《Concepts, Techniques and Models of Computer Programming》(<http://amzn.com/0262220695?tag=devtools-20>)极有帮助。它帮助我理解 R 语言的**修改时复制**(copy-on-modify)语义，让我理解代码变得更容易，但是，当前在 R 语言中的实现效率并不是特别高，不过这是一个可以解决的问题。

如果你想学习成为一名更好的程序员，没有比由 Andrew Hunt 和 David Thomas 编写的《The Pragmatic Programmer》(<http://amzn.com/020161622X?tag=devtools-20>)更好的资料了。这本书为如何成为更好的程序员提供了很好的建议。

1.5 得到帮助

目前，当你遇到了问题，但是又不知道问题的原因的时候，主要有两个寻求帮助的地方：**stackoverflow**(<http://stackoverflow.com>)和 **R-help** 邮件列表。你可以在这两个地方得到非常好的帮助，但是它们都有各自的社区文化和发帖要求。在你发布帖子之前，花一点时间在这些社区上多看看、多学习这些社区的要求，通常是个好主意。这里有一些好的建议：

1. 确保你的 R 语言和你有问题的包都是**最新版本**的。因为，你的问题可能是最近修改过的错误。

2. 花一些时间创建一个可重现的例子

(<http://stackoverflow.com/questions/5963269>)。通常，这本身就是一个有用的过程，因为在让问题重现的过程中，你常常自己就找到了导致问题原因。

3. 在发帖前搜索相关问题。如果有人已经问过你的问题，并且得到了解答，那么使用现有答案会快得多。

1.6 鸣谢

我要感谢那些为 R-help 和

stackoverflow(<http://stackoverflow.com/questions/tagged/r>)不知疲倦地进行贡献的人。这里有太多的名字，但我要特别感谢 Luke Tierney、John Chambers、Dirk Eddelbuettel、JJ Allaire 和 Brian Ripley 慷慨地付出了自己的时间，并且纠正了我无数的误解。

本书是开放式编写的(<https://github.com/hadley/adv-r/>)，章节完成时会发布在 twitter 上(<https://twitter.com/hadleywickham>)。这是真正的社区共同努力的结果：许多人阅读草稿，修改错字，给出改进的建议以及贡献文章内容。如果没有这些贡献者，本书不可能有这么好，我深切地感激他们的帮助。要特别感谢 Peter Li，他完整地阅读了本书，并且给出了很多修改建议。还要感谢很多其它的贡献者。

1.7 约定

在本书中我使用 **f0** 来表示函数，**g** 来表示变量和函数参数，以及用 **h/** 表示路径。在更大的代码块中，输入和输出是混合在一起的。输出被加了注释，所以如果你拥有本书的电子版，例如 <http://advr.had.co.nz>，那么你可以很容易地复制并粘贴到 R 中。输出的注释看起来像 **#>**，以区别于常规的注释。

1.8 版权声明

本书是使用 RStudio(<http://www.rstudio.com/ide/>)中的 Rmarkdown(<http://rmarkdown.rstudio.com/>)写成的。使用了 **knitr**(<http://yihui.name/knitr/>)和 **pandoc**(<http://johnmacfarlane.net/pandoc/>)把原始的 Rmarkdown 文本转化为 html 和 pdf 格式。本书网站(<http://adv-r.had.co.nz>)是使用 **jeekyll**(<http://jeekyllrb.com/>)建立的，并且使用了 **bootstrap**(<http://getbootstrap.com/>)来制作风格，然后通过 **travis-ci**(<https://travis-ci.org/>)自动发布到亚马逊的 **S3**(<http://aws.amazon.com/s3/>)。完整的代码在 **github**(<https://github.com/hadley/adv-r>)上可以得到。代码是在 **inconsolata**(<http://levien.com/type/myfonts/inconsolata.html>)中设置的。

2 数据结构

本章总结了 R 语言中最重要的数据结构。你以前可能使用了这些数据结构很久了(也许不是所有的数据结构),但是可能从来没有深入思考过它们之间的关系是怎么样的。在本章短暂的概述中,我不会深入地讨论每种类型。相反,我将展示如何将它们组合在一起作为一个整体。如果你需要更多的细节,那么你可以在 R 语言的文档中找到它们。

R 语言的**基础数据结构**可以按照**维度**来划分(1 维、2 维...n 维);也可以按照**它们所包含的数据类型是否相同**来划分。这样就产生了五种数据类型,它们是数据分析中最常用的:

	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data frame
nd	Array	

几乎所有的其它对象都是建立在这些基础数据结构之上。在第 7 章中,你将看到如何以这些简单的数据结构,构建更加复杂的对象。请注意, R 语言**没有 0 维数据结构(标量)**。你可能认为单个的数字或者字符串是标量,但是实际上它们是**长度为 1 的向量**。

给定一个对象，要理解它们的数据结构的最好方式，是使用 `str()` 函数。`str()` 是单词 `structure` 的缩写，它可以应用到任何 R 语言数据类型，并且可以给出紧凑的、容易理解的描述信息。

小测验

做这个简短的测试来确定你是否需要阅读本章。如果你很快就能想到答案，那么你可以轻松地跳过这一章。答案在 2.5 节。

1. 除了包含的数据以外，向量的三个性质是什么？
2. 四种常见的原子向量类型是什么？两种罕见的类型是什么？
3. 属性是什么？如何存取属性？
4. 原子向量和列表有什么不同？矩阵和数据框有什么不同？
5. 列表也可以是矩阵吗？数据框能把矩阵作为一系列数据吗？

本章概要

2.1 节介绍原子向量、列表、R 的一维数据结构。

2.2 节将讨论属性，R 语言的灵活的元数据规范。你将学习因子类型：一种重要的数据结构，它是通过设置原子向量的属性而得来的。

2.3 节介绍了矩阵和数组：用来存储 2 维或更高维数据的数据结构。

2.4 节介绍数据框：在 R 语言中存储数据最重要的数据结构。数据框把列表和矩阵的行为结合起来，使得这种结构适合统计数据的需要。

2.1 向量

R 语言中最基本的数据结构是向量。向量有两种形式：原子向量和列表。它们有三个共同的属性：

1. 类型, `typeof()`, 它是什么。
2. 长度, `length()`, 它包含有多少元素。
3. 属性, `attributes()`, 额外的任意元数据。

它们的不同在于元素类型: 原子向量中的所有元素都必须是相同的类型; 而列表中的元素可以是不同的类型。

注: `is.vector()` 并不能测试一个对象是不是向量。相反, 仅当对象是除了名字以外, 不包含其它属性时, 它才返回 `TRUE`。所以, 请使用 `is.atomic(x) || is.list(x)` 来测试一个对象是不是向量。

2.1.1 原子向量

我将详细讨论四种常见的原子向量类型: 逻辑类型、整数类型、双精度类型(通常称为数值类型)和字符类型。我们不讨论两种罕见类型: 复数类型和 `raw` 类型。

原子向量通常由 `c()` 函数创建, 它是单词 `combine` 的简称:

```
dbl_var <- c(1, 2.5, 4.5)
# 使用 L 后缀, 你可以得到整数而不是双精度浮点数
int_var <- c(1L, 6L, 10L)
# 使用 TRUE 和 FALSE(或 T 和 F)来创建逻辑向量
log_var <- c(TRUE, FALSE, T, F)
chr_var <- c("these are", "some strings")
```

原子向量总是"平的"(flat), 甚至把 `c()` 函数嵌套起来也是这样:

```
c(1, c(2, c(3, 4)))
#> [1] 1 2 3 4
# 与以下的相同
c(1, 2, 3, 4)
#> [1] 1 2 3 4
```

缺失值用 **NA** 来表示，这是一个长度为 **1** 的逻辑向量。在 **c()** 函数中使用时，**NA** 总是被强制转换为正确的类型。(译者注：**NA** 的转换依赖于其它元素的类型。) 或者你可以使用 **NA_real_**(双精度浮点数向量)、**NA_integer_** 和 **NA_character_** 来创建一系列确定类型的 **NA**。

2.1.1.1 类型和测试

给定一个向量，可以使用 **typeof()** 来确定其类型，或者使用 "is" 开头的函数来检查它是不是某种特定类型：**is.character()**、**is.double()**、**is.integer()**、**is.logical()** 或者更通用的 **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
```

注意：**is.numeric()** 是用于测试向量是不是 "数值" ("numberliness") 类型的，它对整数和双精度浮点数向量都会返回 **TRUE**。这个函数不是针对双精度浮点数向量的，双精度浮点数(double)通常被称为数值(numeric)。


```
is.numeric(int_var)
```

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

```
is.numeric(dbl_var)
```

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

2.1.1.2 强制转换

一个原子向量中的所有元素都必须是**相同的类型**。所以，当你试图合并不同类型的数据时，将向最灵活的类型进行强制转换。以灵活程度排序，从小到大依次为：逻辑、整数、双精度浮点数和字符。例如，合并字符和整数将得到字符：

```
str(c("a", 1))
```

```
#> chr [1:2] "a" "1"
```

当逻辑向量被强制转换为整数或者双精度浮点数类型时，**TRUE** 将变成 **1**，**FALSE** 将变成 **0**。这项特性使得逻辑向量与 **sum()** 和 **mean()** 函数结合使用时，变得非常方便。

```
x <- c(FALSE, FALSE, TRUE)
```

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

```
#> [1] 0 0 1
```

```
# TRUE 的总数量
```

```
sum(x)
```

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

```
# TRUE 的比例
```

```
mean(x)
```

```
#> [1] 0.3333
```

强制转换经常会自动发生。大多数数学函数(**+**、**log**、**abs** 等)，将向双精度浮点数或者整数类型进行强制转换；而大多数逻辑运算符，将向逻辑类型进行转换。如果转换过程中可能会丢失信息，那么你将会得到警告消息；如果转换遇到歧义，则

需要使用 `as.character()`、`as.double()`、`as.integer()` 或者 `as.logical()` 等函数，进行明确的强制转换。

2.1.2 列表

列表与原子向量是不同的，因为它们的元素可以是任何类型，甚至也包括列表类型本身。创建列表使用 `list()` 函数，而不是 `c()` 函数：

```
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
```

列表有时被称为递归向量，因为一个列表可以包含其它列表：这使得它们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()` 可以将几个向量合并成一个。如果原子向量和列表同时存在，那么在合并之前，`c()` 会将原子向量强制转换成列表。比较一下调用 `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()` 函数，得到的结果是列表。你可以用 `is.list()` 来测试列表，或者使用 `as.list()` 来强制转换成列表。你可以使用 `unlist()` 把一个列表转换为原子向量。如果列表中的元素具有不同的类型，那么 `unlist()` 将使用与 `c()` 相同的强制转换规则。

列表用来建立 R 语言中的许多更加复杂的数据结构。比如，**数据框**(在第 2.4 节中描述)和**线性模型对象**(由 `lm()` 产生)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. 原子向量的六种类型是什么？列表和原子向量的区别是什么？
2. `is.vector()`和 `is.numeric()`与 `is.list()`和 `is.character()`的根本区别是什么？
3. 推测下面 `c()`函数的强制转换结果：

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

4. 为什么要使用 `unlist()`把列表转换为原子向量？为什么不能使用 `as.vector()`？
5. 为什么 `1 == "1"`为 `TRUE`？为什么 `-1 < FALSE` 为 `TRUE`？为什么 `"one" < 2` 为 `FALSE`？
6. 为什么默认缺失值 `NA` 是一个逻辑向量？逻辑向量有什么特殊的地方吗？（提示：想想 `c(FALSE, NA_character_)`。）

2.2 属性

所有的对象都可以拥有任意多个**附加属性**，**附加属性**用来存取与该对象相关的**元数据**。属性可以看做是**已命名的列表**(带有不重复的名字)。属性可以使用 `attr()`函数一个一个的访问，也可以使用 `attributes()`函数一次性访问。

```
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"
```

`structure()` 函数返回一个带有被修改了属性的新对象:

```
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, 当修改向量时, 大多数属性会丢失:

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

但是, 有三种重要的属性不会丢失:

1. **名字(name)**, 一个字符向量, 用来为每一个元素赋予一个名字, 将在第 2.2.0.1 节介绍。
2. **维度(dimension)**, 用来将向量转换成矩阵和数组, 将在第 2.3 节介绍。
3. **类(class)**, 用于实现 S3 对象系统, 在第 7.2 节介绍。

这些属性中的每一个都有特定的访问函数来存取它们的属性值。当访问这些属性时, 请使用 `names(x)`、`class(x)` 和 `dim(x)`, 而不是 `attr(x, "names")`、`attr(x, "class")` 和 `attr(x, "dim")`。

2.2.0.1 名字

你可以用三种方法为向量命名:

1. **创建**向量时:

```
x <- c(a = 1, b = 2, c = 3)
```

2. 就地修改(modify in place)向量时:

```
x <- 1:3; names(x) <- c("a","b", "c")
```

3. 创建向量的一个被修改了的副本时:

```
x <- setNames(1:3, c("a","b", "c"))
```

名字不必是唯一的。但是,就像将在第 3.4.1 节描述的那样,对向量进行命名,最重要的原因是要使用字符对其进行**取子集操作**。而当名字都是唯一的时候,**取子集操作**会更加有用。

并不是向量中的所有元素都需要名字。如果一些元素的名字缺失了,那么 `names()` 将为这些元素返回空字符串(即`""`)。如果所有元素的名字都缺失了,那么 `names()` 将返回 `NULL`。

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

你可以使用 `unname(x)` 创建没有名字的新向量,或者使用 `names(x) <- NULL` 去掉名字。

2.2.1 因子

属性的一个重要应用是定义**因子**。因子是仅包含**预定义值**的向量,用来保存"水平"(level)(或者"种类"(category))数据。(译者注:类似于其它语言中的枚举类型)因子构建于**整数向量**之上,带有两个属性:

1. 类(**class()**), "factor", 使它们与普通的整数向量表现出不同的行为。
2. 水平(**levels()**), 定义了可以允许的取值的集合。

```
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"
# 你不能使用 levels 中没有的值
x[2] <- "c"
#> Warning: invalid factor level, NA generated
x
#> [1] a <NA> b a
#> Levels: a b
# 注意: 不能连接因子
c(factor("a"), factor("b"))
#> [1] 1 1
```

当你知道一个变量可能的取值时, 甚至在数据集中看不到所有的取值时, 因子挺有用的。使用因子代替**字符向量**, 也可以使得一些没有包含**观测数据**的分组, 变得更加醒目。

```
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，当从文件里直接读取数据框时，你认为某列应该产生数值向量，但是却变成了因子。这是由于这一列中有非数值数据造成的，通常，缺失值使用`.或者-`这样的特殊符号来表示。为了避免这个情况，可以先把因子向量转换成字符向量，然后再从字符向量转换到双精度浮点数向量（这个过程之后，务必检查缺失值！）。

当然，一个更好的方法是从一开始就积极寻找出现问题的原因，并予以修改；在`read.csv()`中使用 `na.strings` 参数来解析缺失值字符，通常会更好。

```
# 在这里，从"text"中读取，而不是从文件中读取：
z <- read.csv(text = "value\n12\n1\n.\n9")
typeof(z$value)
#> [1] "integer"
as.double(z$value)
#> [1] 3 2 1 4
# 哦，不对：3 2 1 4 是因子的水平值，而不是我们读进来的值
class(z$value)
#> [1] "factor"
# 我们现在可以进行修复：
as.double(as.character(z$value))
#> Warning: NAs introduced by coercion
#> [1] 12 1 NA 9
# 或者改变我们读取的方式：
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
# 完美! :)
```

不幸的是，在 R 语言中，大多数的**数据读取函数**会自动地把字符向量转换成因子。这种方式挺理想化的，因为这些函数并没有办法知道所有的**因子水平**，以及因子水平最合理的排序方式。我们可以使用 `stringsAsFactors = FALSE` 参数来避免这种行为，然后再根据你对数据的理解，通过手动方式把字符向量转换为因子。全局设置 `options(stringsAsFactors = FALSE)` 也可以控制这个行为，但是我不建议这么做。因为如果改变了全局设置，那么在运行其它代码时(不管是**程序包**的代码还是你自己的代码)，都可能会带来意想不到的后果；修改全局设置也使得代码变得更难理解，因为增加了代码量，而且你得清楚地知道这行代码起了什么作用。因子看起来像字符向量(有时候行为也像)，但是实际上它是**整数**。当把它们作为**字符串**来使用时，必须非常小心。一些处理字符串的方法（比如 `gsub()` 和 `grepl()`），会把因子强制转换成字符串；一些方法(比如 `nchar()`)会抛出错误；而另一些(比如 `c()`)则会使用它们的整数值。因此，如果你需要让因子表现出类似字符串的行为，最好是明确地把因子转换为字符向量。在 R 语言早期版本里，使用因子来代替字符向量，是有节省计算机内存方面的考虑的，但是现在内存已经不再是一个问题。

2.2.2 练习

1. 以下代码定义了一个结构，使用了 `structure()` 函数：

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

但是，当你打印该对象的时候，却看不到 `comment` 属性。这是为什么呢？是属性缺失，或者是其它原因呢？（提示：尝试使用帮助文档。）

2. 当你修改因子的水平时，会发生什么？

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

3. 这段代码起了什么作用？ `f2` 和 `f3` 与 `f1` 相比，有什么区别？

```
f2 <- rev(factor(letters))
f3 <- factor(letters, levels = rev(letters))
```

2.3 矩阵和数组

为原子向量添加一个 `dim()` 属性，可以让它变成多维数组。数组的一种特例是矩阵，即二维数组。矩阵在统计学中使用得非常广泛。高维数组则用得少多了，但是也需要一定的了解。矩阵和数组是由 `matrix()` 和 `array()` 函数创建的，或者通过使用 `dim()` 函数对维度(`dimension`)属性进行设置来得到。

```
# 两个标量参数指定了行和列
a <- matrix(1:6, ncol = 3, nrow = 2)
# 一个向量参数描述所有的维度
b <- array(1:12, c(2, 3, 2))
# 你也可以通过设置 dim() 就地修改一个对象
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()`在任何维度上都可以使用，而对于矩阵和数组，则有更细分的函数：

1. `length()`: 对于矩阵，`nrow()`和 `ncol()`分别获取行数和列数；对于数组，`dim()`获取每个维度。
2. `names()`: 对于矩阵，`rownames()`和 `colnames()`分别获取行名和列名；对于数组，`dimnames()`获取每个维度的名字。

```
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
```

`cbind()`和`rbind()`函数是`c()`函数对矩阵的推广；`abind()`函数是`c()`函数对数组的推广(由`abind`包提供)。你可以使用`t()`转置一个矩阵；它对数组的推广，则是`aperm()`函数。你可以使用`is.matrix()`和`is.array()`来测试一个对象是不是矩阵或者数组，或者查看`dim()`函数返回的维度值。`as.matrix()`和`as.array()`使向量转化为矩阵或数组变得简单。向量不是仅有的一维数据结构。你可以创建单行或者单列矩阵，也可以创建单维数组。它们看起来很像，但是行为是不同的。它们的区别并不是那么重要，但是当你运行某些函数(比如`tapply()`函数常出现这个情况)得到了奇怪的输出时，你要能想到它们的存在。同样地，使用`str()`可以查看它们的区别。

```
str(1:3) # 一维向量
```

```
#> int [1:3] 1 2 3
```

```
str(matrix(1:3, ncol = 1)) # 列向量
```

```
#> int [1:3, 1] 1 2 3
```

```
str(matrix(1:3, nrow = 1)) # 行向量
```

```
#> int [1, 1:3] 1 2 3
str(array(1:3, 3)) # "数组"("array")向量
#> int [1:3(1d)] 1 2 3
```

原子向量通常被转化成矩阵，**维度属性**也可以在列表中设置，从而得到**列表矩阵**或者**列表数组**。(译者注：列表矩阵或列表数组中的元素可以是不同的类型。)

```
l <- list(1:3, "a", TRUE, 1.0)
dim(l) <- c(2, 2)
l
#> [,1] [,2]
#> [1,] Integer,3 TRUE
#> [2,] "a" 1
```

这些都是比较深奥的数据结构，但是如果你想把对象排列在类似于“网格”的结构中，那么还是挺有用的。例如，如果你在时空网格上运行模型，通过把模型存储在三维数组中，那么它可以自然地保持网格结构。

2.3.1 练习

1. 对向量使用 `dim()` 函数时，会返回什么？
2. 如果 `is.matrix(x)` 返回 `TRUE`，那么 `is.array(x)` 会返回什么？
3. 如何描述以下三个对象？它们使得 `1:5` 有什么不同？

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

2.4 数据框

数据框是 R 语言中最常用的存储数据的方式，如果使用得当，可以使数据分析工作变得更轻松(<http://vita.had.co.nz/papers/tidy-data.pdf>)。数据框是由等长向量构成的列表。它也是二维结构，所以它具有矩阵和列表双重属性。也就是说，数据框拥有 `names()`、`colnames()` 和 `rownames()`，尽管 `names()` 和 `colnames()` 对数据框来说是一样的。数据框的 `length()` 是列表的长度，所以和 `ncol()` 相同；`nrow()` 则得到行数。

在第三章，你可以像对一维结构(列表行为)或二维结构(矩阵行为)那样，对数据框进行取子集操作。

2.4.1 创建

你可以使用 `data.frame()` 来创建数据框，它以带命名的向量作为输入：

```
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
```

注意 `data.frame()` 的默认行为是把字符转换为因子，可以使用 `stringsAsFactors = FALSE` 避免这个行为：

```
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"
```

2.4.2 测试和强制转换

由于数据框是 S3 类，它是由向量构建而成，所以它的类型反映出向量的特性：它是列表。要检查一个对象是不是数据框，可以使用 `class()` 函数或者 `is.data.frame()` 函数：

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

你可以使用 `as.data.frame()` 把一个对象转换成数据框：

1. 一个原子向量会创建单列数据框。
2. 列表中的每个元素会成为数据框的一列；如果元素的长度不同，则会发生错误。
3. `n` 行 `m` 列的矩阵会转换为 `n` 行 `m` 列数据框。

2.4.3 连接数据框

可以使用 `cbind()` 和 `rbind()` 来连接数据框：

```
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
```

当按列连接时，行数必须相匹配，但是行名会被忽略；当按行连接时，列数和列名必须都要匹配。使用 `plyr::rbind.fill()` 可以连接列数不同的数据框。

通过 `cbind()` 把原子向量连接在一起来创建数据框，是一种常见的错误。这是行不通的，因为除非 `cbind()` 的参数中含有数据框，否则 `cbind()` 将创建矩阵类型，而不是数据框类型。

因此，请直接使用 `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()` 的转换规则相当复杂，所以为了避免转换，最好确认所有的输入参数都是相同的类型。

2.4.4 特殊列

由于数据框是一个包含向量的列表，所以数据框的某个是列表类型是有可能的：

```
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
```

然而，当把列表传入 `data.frame()` 函数时，该函数将试图把列表的每一个元素都放到单独的一列中，所以，下面的代码会失败：

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

一种绕开的方法是使用 `I()` 函数，它使得 `data.frame()` 把列表看成一个整体单元：

```
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()`函数增加了 `AsIs` 类作为输入，但是通常忽略它也没关系。类似地，也可以把矩阵或者数组作为数据框的一列，只要与数据框的**行数**相匹配即可：

```
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
```

使用列表和数组作为列时，需要特别注意，因为许多操作数据框的函数，都会假定数据框的所有列都是原子向量。

2.4.5 练习

1. 数据框拥有哪些属性？
2. 某个数据框包含不同数据类型的列，对其使用 `as.matrix()` 时，会发生什么？
3. 可以创建 0 行的数据框吗？0 列的呢？

2.5 答案

1. 向量的三个性质是：类型、长度和属性。
2. 四种常用的原子向量类型是：逻辑型、整数型、双精度浮点数型(有时称作数值型)和字符型。两种罕见类型是：复数类型和 `raw` 类型。
3. 属性可以让你在任意对象上附加任意元数据。你可以通过 `attr(x, "y")` 和 `attr(x, "y") <- value` 来存取单个属性；或者通过 `attributes()` 存取所有属性。

4. 列表中的元素可以是任意类型，包括列表类型本身。原子向量的所有元素都是相同的类型；在一个数据框中，不同列可以包含不同类型。
5. 你可以通过在列表中设置维度属性的方式来创建列表数组(list-array)。你可以使用 `df$x<- matrix()`，让一个矩阵作为数据框的一列，或者在创建一个新数据框时使用 `I()`函数，例如 `data.frame(x = I(matrix()))`。

3 取子集操作

R 语言的**取子集操作**既强大又迅速。掌握了**取子集操作**可以让你实现其它语言无法完成的复杂操作。学习**取子集操作**比较难，因为你需要掌握许多相关的概念：

1. 三种**取子集操作符**。
2. 六种类型的取子集方式。
3. 不同对象之间**取子集操作**的重要区别(比如向量、列表、因子、矩阵以及数据框)。
4. **赋值**和**取子集操作**联合使用。

本章从最简单的**取子集操作**类型开始，帮助你掌握**取子集操作**：首先，使用`[`为原子向量取子集，然后逐渐扩展你的知识，转到更复杂的数据类型(如数组和列表)；然后，转到其它**取子集操作符**，`[[`和`$`。然后，你将学习如何把**取子集操作**和**赋值操作**结合起来，用来修改对象的某一部分；最后，你将看到许多有用的应用。

取子集操作是 `str()` 函数的补充操作，`str()` 向你展示了对象的结构，**取子集操作**则可以让你取出对象中感兴趣的部分。

小测验

做个简短的测试来确定你是否需要阅读这一章。如果你能很快地想到答案，那么你可以轻松地跳过这一章。答案在第 3.5 节。

1. 用正整数、负整数、逻辑向量或字符向量对向量进行**取子集操作**的结果是什么？
2. 当应用于列表时，`[`、`[[`和`$`的区别是什么？
3. 在什么时候你应该使用 `drop = FALSE`？

4. 如果 `x` 是一个矩阵，那么 `x[] <- 0` 会做什么？它和 `x <- 0` 有什么不同？
5. 怎样使用已命名的向量，为分类变量重新贴上标签(`relabel categorical variables`)？

本章概要

第 3.1 节从教你使用 `[]` 开始。你将学习六种可以为原子向量进行取子集操作的数据。然后，你将学习为列表、矩阵、数据框和 S3 对象进行取子集操作时，这六种数据类型是如何工作的。

第 3.2 节扩充你在取子集操作符方面知识，包括 `[]` 和 `$`，并关注于“简化”和“保持”的重要原则。

在第 3.3 节，你将学习为子集赋值的方法，它们把取子集操作和赋值操作结合起来，以便修改对象的一部分。

第 3.4 节通过八个重要但是不太明显的取子集操作的应用示例，来帮助你解决在数据分析中经常碰到的问题。

3.1 数据类型

通过原子向量来学习取子集操作是如何工作的，是最容易的；然后，学习它是如何推广到更高维度和更复杂的对象的。我们从 `[]` 开始，它是最常用的操作符。第 3.2 节将讲述 `[]` 和 `$`，它们是另外两个主要的取子集操作符。

3.1.1 原子向量

让我们对以下简单的向量 `x` 进行取子集操作，来研究不同的取子集操作类型：

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

注意：小数点后面的数字给出了元素的**初始索引值**。你可以用五种方法对向量进行**取子集操作**：

1. **正整数**返回该位置的元素：

```
x[c(3, 1)]
#> [1] 3.3 2.1
x[order(x)]
#> [1] 2.1 3.3 4.2 5.4
# 重复的索引得到重复的值
x[c(1, 1)]
#> [1] 2.1 2.1
# 实数会被隐式地截断成整数
x[c(2.1, 2.9)]
#> [1] 4.2 4.2
```

2. **负整数**忽略该位置的元素：

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

在一个**取子集操作**中，不能混合正负整数：

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

3. **逻辑向量**选出对应位置为 **TRUE** 的元素。当你使用**表达式**来创建逻辑向量时，这种方法可能是最有用的：

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

如果逻辑向量比被取子集的向量短，那么它会进行循环填充，直到和向量一样长。

```
x[c(TRUE, FALSE)]
```

```
#> [1] 2.1 3.3
```

```
# 相当于
```

```
x[c(TRUE, FALSE, TRUE, FALSE)]
```

```
#> [1] 2.1 3.3
```

缺失值总是在输出中产生缺失值：

```
x[c(TRUE, TRUE, NA, FALSE)]
```

```
#> [1] 2.1 4.2 NA
```

4. 什么都不写，则返回原始向量。它对向量不是那么有用，但是对于矩阵、数据框和数组是非常有用的。它与赋值操作结合起来时，也挺有用的。

```
x[]
```

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

5. 0 返回零长度的向量。我们通常不会这么做，但是在产生测试数据时，会有点用处。

```
x[0]
```

```
#> numeric(0)
```

如果向量已经被命名，则可以这么使用：

6. 字符向量，将返回名字与该字符匹配的元素。

```
(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
# 就像整数索引，你可以重复索引。
y[c("a", "a", "a")]
#> a a a
#> 2.1 2.1 2.1
# 当通过[进行**取子集操作**时，名字总是**精确匹配**的。
z <- c(abc = 1, def = 2)
z[c("a", "d")]
#> <NA> <NA>
#> NA NA
```

3.1.2 列表

对列表进行**取子集操作**与原子向量相同。使用[将总是返回一个列表；使用[[和\$，如下所述，则让你取出列表的一部分。

3.1.3 矩阵和数组

你可以通过三种方式对更高维的结构进行**取子集操作**：

1. 使用多个向量
2. 使用单个向量
3. 使用矩阵

对矩阵(2 维)和数组(大于 2 维)进行**取子集操作**的最常见的方式，是对一维**取子集操作**的简单推广：你对每一个维度都提供一个**一维索引**，并以逗号分隔。以**留空白(blank)**的方式来取子集，现在变得有用了，因为它代表取出**所有的**行或者列。

```
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，`[`将把结果简化到尽可能低的维数。参见 3.2.1 节学习如何避免这种情况。因为矩阵和数组是由带有特别属性的向量来实现的，所以你可以使用单个向量对它们进行取子集操作。在这种情况下，它们会表现得像一个向量。R 语言中的数组，是按照以列为主的顺序来存储的：

```
(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"
```

你也可以通过**整数矩阵**(或者，如果已经命名了，那么可以使用字符矩阵)，对更高维的数据结构进行取子集操作。矩阵中的每一行指定一个值的位置，每一列对应

一个维度。也就是说，使用 2 列的矩阵对矩阵进行**取子集操作**，用 3 列的矩阵对三维数组进行**取子集操作**，以此类推。得到的结果是一个向量值：

```
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

数据框既具有列表的特点，又具有矩阵的特点：如果你用一个向量对它们进行**取子集操作**，则它们表现得像列表；如果你用两个向量对它们进行**取子集操作**，则它们看起来像矩阵。

```
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
# 从数据框中选择列，有两种方法
# 像列表：
df[c("x", "z")]
#> x z
#> 1 1 a
```

```
#> 2 2 b
#> 3 3 c
# 像矩阵
df[, c("x", "z")]
#> x z
#> 1 1 a
#> 2 2 b
#> 3 3 c
# 如果你选择单个列，那么有重要的区别：
# 默认情况下，矩阵的**取子集操作**会对结果进行简化，
# 而列表却不是这样。
str(df["x"])
#> 'data.frame': 3 obs. of 1 variable:
#> $ x: int 1 2 3
str(df[, "x"])
#> int [1:3] 1 2 3
```

（译者注：所谓"简化"，是指如果对矩阵取子集得到的结果是一维的，那么会默认会转化为向量，使用 `drop = FALSE` 参数可以避免这一点。例如，`df[, "x", drop = FALSE]`）

3.1.5 S3 对象

S3 对象是由原子向量、数组和列表组成的，所以你可以使用上面描述的技术对 S3 对象进行取子集操作。你可以通过 `str()` 函数获得的它们的结构信息。

3.1.6 S4 对象

对 S4 对象来说, 有另外两种**取子集操作符**: `@`(相当于 `$`)和 `slot()`(相当于 `[]`)。 `@`比 `$`更加严格, 如果**槽(slot)**不存在, 那么它会返回错误。在第 7.3 节将描述更多细节。

3.1.7 练习

1. 下列代码试图对数据框进行**取子集操作**, 但是都有错误, 请修改:

```
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]` 产生了 5 个缺失值? (提示: 为什么与 `x[NA_real_]` 不同?)
3. `upper.tri()` 函数返回什么? 它是怎样对矩阵进行**取子集操作**的? 我们需要其它**取子集操作**的规则来描述它的行为吗?

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

4. 为什么 `mtcars[1:20]` 返回错误? 它跟看起来很相似的 `mtcars[1:20,]` 有什么不同?
5. 实现一个函数, 它将取出矩阵**主对角线**上的元素(它应该与 `diag(x)` 函数表现相似的行为, 其中 `x` 是一个矩阵)。
6. 语句 `df[is.na(df)] <- 0` 做了什么? 它是如何做的?

3.2 取子集操作符

还有另外两个取子集操作符：`[[`和`$`。除了只能返回单个值以外，`[[`与`[`是类似的，它还可以用于取出列表的一部分。当通过字符进行取子集操作时，`$`是`[[`一种有用的简化写法。

对列表进行操作时，需要`[[`。这是因为当`[`应用于列表时，总是返回列表：它从来不会返回列表包含的内容。为了获得列表的内容，需要使用`[[`：

"如果列表 `x` 是一列载有对象的火车的话，那么 `x[[5]]` 就是在第 5 节车厢里的对象；而 `x[4:6]` 就是第 4-6 节车厢。" ——@RLangTip

因为它只能返回单个值，所以，你必须把单个正整数或字符串与`[[`联合起来使用：

```
a <- list(a = 1, b = 2)
a[[1]]
#> [1] 1
a[["a"]]
#> [1] 1
# 如果你提供的是向量，那么会进行递归索引。
b <- list(a = list(b = list(c = list(d = 1))))
b[[c("a", "b", "c", "d")]]
#> [1] 1
# 与以下相同
b[["a"]][["b"]][["c"]][["d"]]
#> [1] 1
```

因为数据框是包含一些列的列表，所以可以对数据框使用`[[`来提取列：

`mtcars[[1]]`、`mtcars[["cyl"]]`。S3 和 S4 对象则可以覆盖`[`和`[[`的标准行为，对于不同类型的对象，它们的行为会有所不同。关键的区别，通常是你选择"简化"或"保持"行为之间如何进行选择，以及默认的行为是什么。

3.2.1 取子集方式：简化与保持

理解**简化**(simplifying)和**保持**(preserving)之间的区别很重要。（译者注：所谓简化与保持，即**取子集操作**之后是否把结果转化为更简单的数据类型，比如取出数据框的一列，如果选择简化，则返回的是向量，如果选择保持，则返回的仍然是数据框）"简化"取子集操作，将返回**可以表示输出并且尽可能简单的数据结构**，在交互环境下，这是有用的，因为它通常能给你想要的结果。"保持"取子集操作，使输出与输入的结构保持相同，在编程环境下，通常是更好的，因为结果将永远是相同的类型。在对矩阵和数据框进行**取子集操作**时忽略了 `drop = FALSE`，是编程中最常见的错误来源之一。（比如，你写了一个函数，用你的测试例子可以正常工作，但是如果有人传入了单列数据框，则可能会发生意想不到的、不明确的失败。）不幸的是，如何在简化和保持之间进行切换，是随着不同的数据类型变化的，请看下表中的总结：

	Simplifying	Preserving
Vector	<code>x[[1]]</code>	<code>x[1]</code>
List	<code>x[[1]]</code>	<code>x[1]</code>
Factor	<code>x[1:4, drop = T]</code>	<code>x[1:4]</code>
Array	<code>x[1,]</code> or <code>x[, 1]</code>	<code>x[1, , drop = F]</code> or <code>x[, 1, drop = F]</code>
Data frame	<code>x[, 1]</code> or <code>x[[1]]</code>	<code>x[, 1, drop = F]</code> or <code>x[1]</code>

保持对所有数据类型都是相同的：你得到的输出类型与输入类型是相同的。**简化**的行为随着不同数据类型略有变化，如下所述：

1. **原子向量**: 移除名字。

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

```
#> a
#> 1
x[[1]]
#> [1] 1
```

2. **列表**: 返回列表内的对象，而不是包含单个元素的列表。

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

3. **因子**: 丢弃所有没有用到的水平。

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

4. **矩阵和数组**: 如果任一维度的长度为 1，则丢弃那个维度。

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

5. **数据框**: 如果输出是单列的, 那么将用**向量**替代**数据框**返回。

```
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 \$

\$是一种简化的操作符, **x\$y** 等价于 **x[["y", exact = FALSE]]**。它通常用于访问数据框中的变量, 比如 **mtcars\$cyl** 或 **diamonds\$carat**。一种使用**\$**的常见错误是, 试图把它与**存有列名的变量**联合使用:

```
var <- "cyl"
# 不可行 - mtcars$var 被解释成 mtcars[["var"]]
mtcars$var
#> NULL
# 使用[[
mtcars[[var]]
#> [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8
#> [30] 6 8 4
```

\$与**[[**之间有一个重要区别——**\$**可以**部分匹配**列名:


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

如果你想避免这种行为，你可以把全局变量 `options("warnPartialMatchDollar")` 设置为 **TRUE**。使用时请注意：这样设置可能会影响你加载的其它代码的行为（比如程序包中的代码）。

3.2.3 索引缺失/索引越界

`[`和`[[`在索引越界（out of bounds, OOB）情况下的行为有所区别，例如，当你尝试取出一个长度为 4 的向量中的第 5 个元素时，或者使用 `NA` 或 `NULL` 对向量进行取子集操作时：

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

下面的表格总结了使用 `[`和`[[`对原子向量和列表进行取子集操作的结果，以及不同类型的索引越界值。

Operator	Index	Atomic	List
[OOB	NA	list(NULL)
[NA_real_	NA	list(NULL)
[NULL	x[0]	list(NULL)
[[OOB	Error	Error
[[NA_real_	Error	NULL
[[NULL	Error	Error

如果输入向量已经命名，并且通过名字的方法取子集，那么，找不到名字的匹配值、名字缺失或者名字为 **NULL**，都将返回 "<NA>"。

3.2.4 练习

1. 给定一个线性模型，比如 `mod <- lm(mpg ~ wt, data = mtcars)`，取出残差自由度（the residual degrees of freedom）。从模型的汇总信息(`summary(mod)`)中取出 **R** 方（R-squared）的值。

3.3 取子集与赋值

所有的取子集操作符都可以与赋值操作结合起来使用，用以修改输入向量中被选定的值。

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

```
x
#> [1] 2 3 3 4 5
# LHS(Left-Hand Side, 等式的左边)的长度需要与 RHS(Right-Hand Side, 等式的
# 右边)相匹配
x[-1] <- 4:1
x
#> [1] 2 4 3 2 1
# 注意, 对重复的索引不会进行检查
x[c(1, 1)] <- 2:3
x
#> [1] 3 4 3 2 1
# 不能把整数索引与 NA 连接起来
x[c(1, NA)] <- c(1, 2)
#> Error: NAs are not allowed in subscripted assignments
# 但是, 可以把逻辑索引与 NA 连接起来
# (NA 会被当做 FALSE 处理).
x[c(T, F, NA)] <- 1
x
#> [1] 1 4 3 1 1
# 在根据条件修改向量时, 这是最有用的。
df <- data.frame(a = c(1, 10, NA))
df$a[df$a < 5] <- 0
df$a
#> [1] 0 10 NA
```

以留空白（**nothing**）的方式进行取子集操作，并结合赋值操作，是非常有用的，因为它将保持原始对象的类和结构。

比较以下两个表达式。

在第一个里面，`mtcars` 将保持为数据框。

而在第二个里面，`mtcars` 将成为一个列表。

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

对于列表，你可以使用 `subsetting + assignment + NULL` 从列表中删除元素。为了把 `NULL` 添加到列表中，请使用 `[` 和 `list(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

前面阐述的基本方法可以产生各种各样的应用。下面将描述一些最重要的应用。

许多基本技巧已经被封装成了更简洁的函数(如 `subset()`、`merge()`、`plyr::arrange()`)，但是理解它们是怎样通过基本的取子集操作来实现的，仍然是非常有用的。这将让你能够处理现有函数无法完成的情况。

3.4.1 查询表(字符取子集操作)

字符匹配提供了一种创建查询表的强大方法。比如，你想进行单词缩写转换：

```
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"
unnname(lookup[x])
#> [1] "Male" "Female" NA "Female" "Female" "Male"
#> [7] "Male"
# 或者更少的输出值
c(m = "Known", f = "Known", u = "Unknown")[x]
#> m f u f f m
#> "Known" "Known" "Unknown" "Known" "Known" "Known"
#> m
#> "Known"
```

如果你不想让名字显示在结果里，那么可以使用 `unnname()` 函数来删除。

3.4.2 手动匹配与合并(整数取子集操作)

你可能有一张带有多个列的信息、更加复杂的查询表。假设我们有一个表示年级的整数向量，和一张描述它们的属性表：

```
grades <- c(1, 2, 2, 3, 1)
info <- data.frame(
  grade = 3:1,
  desc = c("Excellent", "Good", "Poor"),
  fail = c(F, F, T)
)
```

我们想要复制信息表，以便 `grade` 向量中的每一个值都有一行数据。我们可以通过两种方式来做，使用 `match()` 和整数取子集操作，或者使用 `rownames()` 和字符取子集操作：

```
grades
#> [1] 1 2 2 3 1

# 使用 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

# 使用 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
```

如果你有多列需要匹配，你可能需要首先把它们压缩到单个列之中(使用 `interaction()`、`paste()` 或 `plyr::id()`)。你也可以使用 `merge()` 或 `plyr::join()`，它们可以做同样的事情。请阅读它们的源代码来了解更多信息。

3.4.3 随机采样/自助法(整数取子集操作)

对于向量或者数据框，你可以使用整数索引进行**随机抽样**或**自助法抽样**。

`sample()`函数生成一个包含**随机索引**的向量，然后，通过这些索引进行**取子集操作**：

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

```
# 随机重排序
```

```
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
```

```
# 随机选择 3 行
```

```
df[sample(nrow(df), 3),]
```

```
#> x y z
```

```
#> 2 1 5 b
```

```
#> 1 1 6 a
```

```
#> 5 3 2 e
```

```
# 有放回抽样选择 6 行
```

```
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
```

`sample()`的参数控制着抽样的数量，以及是不是有放回的抽样。

3.4.4 排序(整数取子集操作)

`order()`函数将一个向量作为输入，然后返回一个整数向量，这个整数向量描述了输入向量应该如何排序，以索引的方式来表示：

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

你可以给 `order()` 函数提供额外的变量，你可以使用 `decreasing = TRUE` 把排序方式从升序变成降序。默认情况下，缺失值都将排在向量的末尾；但是，你可以设置 `na.last = NA` 来删除缺失值，或者设置 `na.last = FALSE` 把缺失值排在向量的开头。

对于二维或者更高维的情况，`order()` 函数联合整数取子集操作，使得为对象的行或者列进行排序变得很简单：

```
# 对 df 进行随机重排序
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
```

另外，也可以使用 `sort()` 函数进行排序，会更加简洁，但是不那么灵活；对于数据框，也可以使用 `plyr::arrange()` 函数。

3.4.5 展开聚合的数据(整数取子集操作)

展开聚合的数据(Expanding aggregated counts)：有时你得到了一个数据框，相同的行已经合并到了一起，一个用来计数的列也已经添加进去了。`rep()` 函数联合整数取子集操作，使得展开这种数据变得容易，它通过重复行的索引来进行：

```
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 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
```

3.4.6 从数据框中删除列(字符取子集操作)

有两种方法可以删除数据框的列。你可以把某列设为 **NULL**：

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

或者你可以只取出想要的列：

```
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
```

如果你知道哪些列不需要，可以使用集合操作计算出来：

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

```
#> 1 1 3
#> 2 2 2
#> 3 3 1
```

3.4.7 基于某些条件选择行(逻辑取子集操作)

因为可以让你轻松地把来自多列的条件联合起来，逻辑取子集操作可能是最常用的提取数据框的行的技术。

```
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
```

记住，要使用向量的布尔操作符`&`和`|`，而不是有短路功能的标量操作符`&&`和`||`，后者是适用于 `if` 语句的。不要忘了德摩根法则

(http://en.wikipedia.org/wiki/De_Morgan's_laws)，它可以用来简化语句：

```
!(X & Y)与!X | !Y 相同
!(X | Y)与!X & !Y 相同
```

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()`函数是用于对数据框进行**取子集操作**的一种简便函数，它可以让你少打些字，因为它可以让你不必重复输入数据框的名字。你将在第 13 章中学习怎么使用它。

```
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
```

3.4.8 布尔代数与集合(逻辑和整数取子集操作)

理解**集合操作**（整数取子集操作）与**布尔代数**（逻辑取子集操作）之间的**自然等价性**，是有意义的。在以下情况下，使用**集合操作**会更加有效率：

你想找到第一个（或最后一个）为 **TRUE** 的值。

你只有很少的 **TRUE** 值但是有很多的 **FALSE** 值；集合操作可能会更快并使用更少的内存。

`which()`函数让你把布尔操作转换成整数操作。在 R 语言基础包中，没有逆转换函数，但是我们可以创建一个，很简单：

```
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
```

让我们创建两个逻辑向量，以及它们的整数等价形式，然后研究布尔操作和集合操作之间的关系。

```
(x1 <- 1:10 %% 2 == 0)  
#> [1] FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE  
#> [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  
#> [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  
#> [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
```

第一次学习取子集操作时，一个常见的错误是使用了 `x[which(y)]` 代替 `x[y]`。这里的 `which()` 函数什么作用也没有：它把逻辑值转换成了整数值，但是结果将是完全相同的。还要注意，`x[-which(y)]` 并不是等价于 `x[!y]`：如果 `y` 中所有的值都是 `FALSE`，那么 `which(y)` 将为 `integer(0)`，而 `-integer(0)` 仍然是 `integer(0)`。所以你得不到任何值，而不是所有的值。一般来说，在取子集操作时，应该避免把逻辑值转换成整数值，除非是你真的打算这么干，例如，第一个或者最后一个 `TRUE` 值。

3.4.9 练习

1. 如何随机排列数据框的列？（这是在随机森林中的一个重要技术。）你能在一步之中同时排列行和列吗？

2. 如何在数据框中选择一个 `m` 行随机样本？如果要求样本是连续的(比如，给定一个开始行、一个结束行，得到它们之间的所有行)，应该怎么做呢？
3. 怎样把数据框中的列按照字母顺序进行排序？

3.5 答案

1. 正整数选择在特定位置的元素，负整数删除元素；逻辑向量选择对应位置为 `TRUE` 的元素，字符向量选择与元素名字匹配的元素。
2. `[`选择子列表。它总是返回列表；如果你使用单个正整数来执行它，则返回长度为 1 的列表。`[[`选择列表中的元素。`$`是一种简便的操作符：`x$y` 等价于 `x[["y"]]`。
3. 如果你对矩阵、数组或数据框执行**取子集操作**，那么当你希望保持原始维度的时候，可以使用 `drop = FALSE`。在函数内部执行**取子集操作**时，则应该总是设置该参数。
4. 如果 `x` 是一个矩阵，那么 `x[] <- 0` 将把所有的元素替换为 0，并保持相同的行数和列数。`x <- 0` 则完全把矩阵替换成了数值 0。
5. 已命名的字符向量可以当做一个简单的查询表：`c(x = 1, y = 2, z = 3)[c("y", "z", "x")]`

4 词汇表

为了熟练地使用 R 语言进行编程，一个重要的组成部分是掌握好一系列的**函数**、**包**、**各种设置**等等。下面，我列出了我认为重要的函数，并把它们组成了一张词汇表。你不需要熟悉每一个函数的细节，但是你至少应该知道它们的存在，如果在此列表中有你没有听说过的函数，那么我强烈推荐你去阅读它们的帮助文档。通过浏览了 **base** 包、**stats** 包以及 **utils** 包中的所有函数，我想出了这个列表，并选出了我认为其中最有用的。列表中还列出了一些具有特别重要功能的包，以及一些重要的 **options()** 设置。

4.1 基础

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

基本数学函数

*, +, -, /, ^, %%, %/%

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

用于函数的函数

function

missing

on.exit

return, invisible

逻辑和集合

&, |, !, xor

all, any

intersect, union, setdiff, setequal

which

向量和矩阵

c, matrix

自动强制转换规则 character > numeric > logical

```
length, dim, ncol, nrow
cbind, rbind
names, colnames, rownames
t
diag
sweep
as.matrix, data.matrix

# 创建向量
c
rep, rep_len
seq, seq_len, seq_along
Vocabulary 59
rev
sample
choose, factorial, combn
(is/as).(character/numeric/logical/...)

# 列表和数据框
list, unlist
data.frame, as.data.frame
split
expand.grid

# 控制流
if, &&, || (short circuiting)
for, while
next, break
switch
```

ifelse

apply 族函数

lapply, sapply, vapply

apply

tapply

replicate

4.2 通用数据结构

日期与时间

ISOdate, ISOdatetime, strptime, strptime, date

difftime

julian, months, quarters, weekdays

library(lubridate)

字符操作

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)  
  
# 数组操作  
array  
dim  
dimnames  
aperm  
library(abind)
```

4.3 统计学

```
# 排序与制表  
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  
  
# 其它测试
```

```
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)
```

```
# 矩阵代数
```

```
crossprod, tcrossprod
```

```
eigen, qr, svd
```

```
%*%, %o%, outer
```

```
rcond
```

```
solve
```

4.4 使用 R 语言工作

```
# 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

4.5 输入/输出

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)

文件和路径

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)

5 编码风格指南

良好的编码风格就好像使用正确的标点符号。你可以不用标点符号，但是它肯定会让文章变得更容易阅读。带有标点符号的风格，可能有许多变化。以下指南描述了我使用的风格(在本书和其它地方)。它是基于《谷歌 R 代码风格指南》(<http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html>)，然后做了一些微调。你不一定要使用我的风格，但是你确实应该使用**一致的风格**。

良好的风格是很重要的，因为虽然你的代码只有一个作者，但是却通常有多个读者，尤其是当你与别人合作的时候。在这种情况下，事先约定好一种共同的风格，是个好主意。

由于没有哪种风格是最好的，所以与其它人一起工作时，可能意味着你需要牺牲一些你喜欢的风格。

由谢益辉创建的 **formatR** 包，可以使没有良好风格的代码变得清晰。尽管它没有办法做所有的事情，但是可以迅速地让你的代码从难看变得漂亮。在使用它之前，请务必阅读维基百科上的文章(<https://github.com/yihui/formatR/wiki>)。

5.1 标识符和命名

5.1.1 文件名

文件名应该有意义，并且以.R 结尾。

好

fit-models.R

utility-functions.R

不好

foo.r

stuff.r

如果文件需要按照顺序执行，则在文件名上加上数字前缀：

0-download.R

1-parse.R

2-explore.R

5.1.2 对象名

"计算机科学中只有两件事情很困难：缓存失效和命名。" ——Phil Karlton

变量名和**函数名**应该是小写的。使用下划线(_)分隔名字中的单词。一般来说，变量名应该是名词，函数名应该是动词。尽量把名字取得简洁而有意义(这并不容易！)。

好

day_one

day_1

不好

first_day_of_the_month

DayOne

dayone

djm1

只要有可能，就应该避免使用现有的函数和变量的名字。否则，将导致读者迷惑。

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

5.2 语法

5.2.1 空格

在中缀操作符(=, +, -, <-等)两边都留空格。在函数调用里使用=时, 也使用同样的规则。总是在逗号后面留一个空格, 而逗号前面则不要留(就像普通英语文章那样)。

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

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

这条规则有一个例外——在::和:::的两边都不要留空格。

```
# 好
x <- 1:10
base::get

# 不好
x <- 1 : 10
base :: get
```

除了函数调用以外, 在左括号前面留一个空格。

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

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

如果可以对齐等式或者赋值表达式(<-)，留有一些额外的空格(比如，在一行中多于一个空格)也是可以的。

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

不要在圆括号或者方括号内的代码两边留空格（除非有逗号，看前述关于逗号的规则）

```
# 好
if (debug) do(x)
diamonds[5,]

# 不好
if ( debug ) do(x) # debug 两边不要留空格
x[1,] # 逗号后面需要留一个空格
x[1,] # 空格要留在逗号后面，而不是前面
```

5.2.2 花括号

左花括号后面应该新起一行。右花括号后面则不应该新起一行，除非它后面跟着的是 `else` 语句。总是在花括号中进行代码缩进。

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

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

把几个很短的语句放在同一行，也是可以的：

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

5.2.3 代码行的长度

尽量使每行代码不超过 80 个字符。如果要使用合适的字体把代码打印出来，那么这是与其相符合的。如果你发现自己没有空间放置代码了，那么这说明你应该把一些工作封装在一个单独的函数里。

5.2.4 缩进

缩进代码时，使用两个空格。不要使用制表符或者把制表符和空格混合使用。唯一的例外是一个函数定义分成了多行的情况。在这种情况下，第二行缩进到函数定义开始的位置：

```
long_function_name <- function(a = "a long argument",  
b = "another argument",  
c = "another long argument") {  
  # 普通代码缩进两个空格  
}
```

5.2.5 赋值

使用<-，而不要使用=进行赋值。

```
# 好  
x <- 5  
  
# 不好  
x = 5
```

5.3 组织

5.3.1 注释指南

给你的代码添加注释。每一行注释都应该以一个注释符号和单个空格开头：`#`。注释应该解释代码的原理，而不是每行代码做了什么。

使用带有`-`和`=`的**注释线**把你的文件分隔成容易阅读的块。

```
# 加载数据 -----  
# 绘图 -----
```

6 函数

函数是 R 语言中的**基本构建块**：为了掌握本书中的许多更先进的技术，你需要一个坚实的基础，要理解函数是如何工作的。你可能已经创造过许多 R 语言函数，你也许基本上了解它们是如何工作的。本章的重点是把你现有的对函数**非正式的知识**转化为对函数**更严格的理解**，理解它们是什么以及它们是如何工作的。你将在这一章看到一些有趣的技巧和技术，但是你学到的大多数内容，都会成为构建更高级技术的基础。

理解 R 语言最重要的事情是要知道**函数本身也是对象**。你可以用与其它任何类型的对象完全一样的方式来使用它们。这个主题将在第十章深入研究。

小测验

回答下列问题，判断你是否可以跳过本章。答案在第 6.7 节。

1. 函数包含哪三个组成部分？
2. 下列代码返回什么？

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

3. 在什么情况下，你更需要写如下代码？

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

4. 怎样让下面的函数调用更具有可读性？

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

5. 调用下列函数时，会抛出错误吗？为什么会/不会？

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

6. 什么是**中缀函数**？怎样编写**中缀函数**？什么是**替换函数**？怎样编写**替换函数**？

7. 使用什么函数来确保**清理动作**的执行，无论函数怎样终止时，都会调用它？

本章概要

第 6.1 节描述了函数的三个主要组成部分。

第 6.2 节教你 R 语言如何通过名字找到值——**词法作用域**的过程。

第 6.3 节向你展示，所有在 R 语言中发生的事情都是**函数调用**的结果，即使看起来不像。

第 6.4 节讨论给函数提供参数的三种方式，以及如何通过提供**参数列表**来调用函数，以及**延迟计算**的影响。

第 6.5 节描述了两种特殊类型的函数：**中缀函数**和**替换函数**。

第 6.6 节讨论函数如何**返回值**，以及何时**返回值**，以及如何可以确保在函数退出之前做一些事情。

前提条件

你唯一需要的包是 **pryr**，它用于研究**就地修改**向量时发生了什么。使用 **install.packages("pryr")** 语句进行安装。

6.1 函数的组成部分

所有的 R 语言函数都有三个组成部分：

函数体，`body()`，函数的代码。

形式参数列表，`formals()`，控制函数调用的参数列表。

环境，`environment()`，函数的变量所在位置的"地图"。

当你打印 R 语言的函数时，它向你展示了这三个重要组成部分。如果环境没有显示，那么这意味着函数是在全局环境中创建的。

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

`body()`、`formals()`和 `environment()`的赋值形式，也可以用于修改函数。（译者注：赋值形式，即 `body()<-`、`formals()<-`和 `environment()<-`。）

像 R 语言中的所有对象一样，函数还可以拥有任意数量的附加属性。被基础 R 语言使用的一个属性，称为"srcref"，它是源引用(source reference)的简称，它指向用于创建函数的源代码。与函数体不同，它包含代码注释和其它格式。你还可以给一个函数添加属性。例如，你可以设置类 `class()`和添加一个自定义的 `print()`方法。

6.1.1 原语函数

函数都有三个组成部分的规则有一个例外。原语函数(Primitive functions)，比如 `sum()`，它直接使用了 `.Primitive()` 调用 C 语言代码，并且不包含 R 语言代码。因此，它们的 `formals()`、`body()` 和 `environment()` 都是 `NULL`：

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

原语函数只存在于 `base` 包中，由于它们在底层运作，所以它们可以更加高效(原语替换函数不需要进行复制)，可以对参数匹配使用不同的规则(如 `switch` 和 `call`)。然而，使用它们的成本是，它们与 R 语言所有的其它函数的行为都不同。因此，R 语言核心团队通常尽量避免创建它们，除非没有其它选择。

6.1.2 练习

1. 什么函数可以告诉你一个对象是不是函数？什么函数可以告诉你一个函数是不是原语函数？
2. 这段代码创建了 `base` 包中的所有函数的列表。

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

使用它来回答下面的问题：

- a. `base` 包中的哪一个函数拥有最多的参数？

- b. **base** 包中的有多少个函数没有参数？这些函数有什么特点？
- c. 怎样修改这段代码，使得它能找到所有的原语函数？
- 3. 函数中的三种重要组成部分是什么？
- 4. 在什么情况下打印一个函数，它**被创建时**所处的环境不会显示出来？

6.2 词法作用域

作用域是一组规则，这些规则控制着 R 语言如何查找**符号的值**。在下面的示例中，作用域是 R 语言用于从符号 **x** 找到它的值 **10** 的规则集：

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

理解作用域可以使你：

- 1. 通过组合函数来构建工具，如第 10 章所述。
- 2. 不按照通常的计算规则进行计算，而是进行**非标准计算**，如第 13 章所述。

R 语言有两种类型的作用域：**词法作用域**(Lexical scoping)，在语言层面自动实现，以及**动态作用域**(Dynamic scoping)，用于在交互式分析情景下，在选择函数时，可以减少键盘输入。

在这里，我们将讨论**词法作用域**，因为它是跟**函数创建**密切相关的。**动态作用域**将在第 13.3 节中详细介绍。

词法作用域进行**符号的值**的查找，是基于在函数创建时是如何嵌套的，而不是它们在调用时如何嵌套的。有了**词法作用域**，你不需要知道函数是怎么被调用的，以及在哪里查找变量的值。你只需要看看函数的定义即可。

在**词法作用域**中, "lexical"并不符合普通英语中的定义("或者相关的文字或语言的词汇, 区别于语法和结构"), 而是来自于计算机学术语"lexing"(词法分析), 它是这样一种过程的一部分: 它把表示为文本的**代码**转换为**编程语言能理解的**有意义的片段。

在 R 语言中, **词法作用域**的实现背后, 有四个基本原则:

1. 名字屏蔽
2. 函数和变量
3. 全新的开始状态
4. 动态查找

虽然你可能没有正式地考虑过这些原则, 但是你可能已经了解过一些。在后续的代码段里, 先不要查看答案, 心里先想想每一段代码是怎样运行的, 测试一下你的知识。

6.2.1 名字屏蔽

本节说明**名字屏蔽**(Name masking)。下面的例子说明了**词法作用域**的最基本的原则, 请你来预测一下输出, 应该问题不大。

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

如果一个名字在函数中没有定义, 那么 R 语言将向上一个层次查找。

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

如果一个函数内部定义了另一个函数，也适用同样的规则：首先，查看当前函数的内部，然后是这个函数被定义的环境，然后继续向上，以此类推，一直到**全局环境**，然后，再查找其它已经加载的包。先在你的脑子里运行下面的代码，然后通过运行 R 代码来确认输出是否正确。

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

同样的规则也适用于**闭包**——由其它函数创建的函数。**闭包**将在第 10 章进行更详细地描述；在这里，我们只是看看它们如何与作用域进行交互。以下函数，**i()**，返回一个函数。当我们调用它的时候，你认为这个函数将返回什么呢？

```
j <- function(x) {
  y <- 2
```

```
function() {  
  c(x, y)  
}  
}  
k <- j(1)  
k()  
rm(j, k)
```

这似乎有点不可思议(R 语言是如何知道函数被调用后, `y` 的值是什么)。这是因为 `k` 保存着定义它的环境, 而该环境中包含了 `y` 的值。第 8 章将提供一些启示, 告诉你如何深入环境, 并在与每个函数关联的环境中, 找出存储的值。

6.2.2 函数和变量

无论关联了什么类型的值, 都适用同样的原则——搜索函数与搜索变量的过程完全一样:

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

对于函数, 规则有一点点调整。如果很明显你要的是函数(例如, `f(3)`), 那么在这样的语境中, R 语言在搜索时, 将忽略那些不是函数的对象。在下面的示例中, `n` 的不同取值取决于 R 语言是在寻找一个函数还是一个变量。

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

然而，对函数和其它对象使用相同的名称，会使代码变得混乱，通常应该避免这种情况。

6.2.3 全新的开始状态

对同一个函数调用几次，那么这几次之间，变量值会发生什么变化？当你第一次运行这个函数时会发生什么？第二次运行时将会发生什么？(`exists()`函数的作用：如果以某个名字命名的变量存在，那么它返回 **TRUE**；否则，它返回 **FALSE**)

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

你可能会感到惊讶，每次它都返回相同的值——**1**。这是因为每一次函数被调用时，一个新的环境就会被创建出来，随后，函数会在该环境中执行。函数无法报

告它上一次被调用时发生了什么，因为每次调用都是完全独立的。[但是，我们将在第 10.3.2 节看到一些解决这个问题方法]。

6.2.4 动态查找

词法作用域决定了去哪里查找值，而不是决定在什么时候查找值。R 语言在**函数运行时**查找值，而不是在**函数创建时**查找值。这意味着，函数的输出是可以随着它所处的环境外面的对象，而发生变化的：

```
f <- function() x
x <- 15
f()
#> [1] 15
x <- 20
f()
#> [1] 20
```

你通常应该避免这种行为，因为这意味着函数不再是独立的。这是一种常见的错误——如果你的代码中存在拼写错误，那么当你创建一个函数时，你将得不到错误信息，甚至于你可能在运行该函数的时候，也不会得到错误信息，这取决于**全局环境**中的变量定义。发现这个问题的一种方法是使用 **codetools** 包中的 **findGlobals()** 函数。它会列出一个函数的所有**外部依赖项**：

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

尝试解决这个问题的另一种方法是把函数的环境**手动 (manually)**更改成 **emptyenv()**，它是完全不包含任何对象的空环境：


```
c <- 10
```

```
c(c = c)
```

2. R 语言查找对象值的四条原则是什么？

3. 下列函数返回什么？在运行代码前，自己先判断一下。

```
f <- function(x) {
```

```
f <- function(x) {
```

```
f <- function(x) {
```

```
x ^ 2
```

```
}
```

```
f(x) + 1
```

```
}
```

```
f(x) * 2
```

```
}
```

```
f(10)
```

6.3 所有的操作都是函数调用

"要理解 R 语言中的计算，有两个口号是有用的：万事万物都是对象；发生的所有事情都是函数调用。" ——John Chambers

前面重新定义 `f` 的例子之所以可以成功，是因为 R 语言中的每个操作都是函数调用，无论看起来像不像。这也包括中缀运算符，比如 `+`，控制流运算符，比如 `for`、`if` 和 `while`，取子集操作符，比如 `[]` 和 `$`，甚至花括号 `{`。这意味着，在以下例子中的每一对语句都是完全等价的。注意 `""`，重音符，可以让你引用以保留字或者非法字符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
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
```

我们可以覆盖这些特殊函数的定义，但是几乎可以肯定，这是一个坏主意。然而，有些场合可能是有用的：它可以让你做一些使用常规方法不可能做到的事情。例如，这一特性可以让 **dplyr** 包把 R 表达式翻译成 SQL 表达式。第 15 章将使用

这个想法来创建**领域特定语言**，它允许你使用现有的 R 结构，简明地表达新概念。

把特殊函数当做普通函数使用常常更有用。例如，首先可以定义一个函数 `add()`，然后我们可以使用 `sapply()` 为列表中的每个元素加上 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，我们也可以使用内置的 `+` 函数得到相同的效果。

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

注意以下两者的区别：

```
`+`
"+"
```

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。第二个版本可以起作用，是因为 `lapply` 可以输入一个函数的名称而不是函数本身：如果你读过 `lapply` 函数的代码，那么你可以看到第一行使用了 `match.fun()` 函数通过名字来找到函数。一个更有用的应用是把 `lapply()` 或 `sapply` 与取子集操作组合起来：

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

记住一切发生在 R 中的事情都是**函数调用**会帮助你学好第 14 章的。

6.4 函数参数

区分函数的**形式参数**和**实际参数**是有用的。**形式参数**是函数的一个属性，而**实际参数**或**调用参数**可以在每次调用函数时都不同。本节讨论**调用参数**是怎样映射到**形式参数**的，怎样通过保存了参数的列表来调用函数，**默认参数**是如何工作的，以及**延迟计算**的影响。

6.4.1 调用函数

当调用一个函数时，你可以通过**参数的位置**，或者通过**完整的名称**或者**部分的名称**，来匹配参数。参数匹配的顺序是：首先是**精确的名称匹配**(完美匹配)，然后通过**前缀匹配**，最后通过**位置匹配**。

```
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
```

```
# 可以缩写长参数名:
```

```
str(f(2, 3, a = 1))  
#> List of 3  
#> $ a : num 1  
#> $ b1: num 2  
#> $ b2: num 3  
# 但是，这里不可行，因为缩写有歧义  
str(f(1, 3, b = 1))  
#> Error: argument 3 matches multiple formal arguments
```

一般来说，你只希望使用**位置匹配**来匹配排在前面的一、两个参数；它们是最常用的，大多数用户都知道它们是什么。要避免使用**位置匹配**来匹配较少使用的参数，并且仅使用**具有可读性的缩写形式**来对参数进行**部分匹配**。(如果你正在为一个包编写代码，你想要把它发布在 CRAN 上，那么不能使用**部分匹配**，而必须使用完整的名字。)命名的参数应该总是排在未命名的参数后面。如果一个函数使用了... (下面将详细讨论)，则...之后列出的参数都必须使用它们的全名。

这些是好的调用方式：

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

这样可能就啰嗦了一点：(译者注：但是我本人推荐对所用参数都使用这种方式，让人一目了然，开销只是多输入了几个字符而已)

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

而这些只能让人糊涂了：

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

6.4.2 给定一个参数列表来调用函数

假设你有一个函数的参数列表：

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

怎样把这个列表传递给 `mean()` 函数呢？这时候，你需要 `do.call()` 函数：

```
do.call(mean, list(1:10, na.rm = TRUE))
```

```
#> [1] 5.5
```

```
# 相当于
```

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

```
#> [1] 5.5
```

6.4.3 默认参数与缺失参数

R 语言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
```

由于在 R 语言中参数都是延迟计算的，所以参数的默认值可以使用其它参数来定义：

```
g <- function(a = 1, b = a * 2) {
```

```
  c(a, b)
```

```
}
```

```
g()
```

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

```
g(10)
#> [1] 10 20
```

默认参数甚至可以使用在函数内部创建的变量来定义。这是经常在基本 R 函数中使用的技术，但是我认为这是不好的做法，因为如果没有阅读过完整的函数源代码，那么你将不知道默认值到底是什么。

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

你可以使用 `missing()` 函数来确定某个参数是否已经提供了。

```
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
```

有时你想添加一个简单的默认值，这可能需要几行代码来计算。为了不把这段代码插入函数定义，你可以在必要的时候使用 `missing()` 函数有条件地计算它。但

是，如果没有仔细阅读文档，那么将使我们很难知道哪些参数是必需的，哪些是可选的。所以，我通常设置默认值为 **NULL**，并且使用 **is.null()** 来检查参数是否被提供。

6.4.4 延迟计算

默认情况下，R 语言函数的参数是**延迟计算**的。仅当实际用到这些参数的时候，它们才会被计算出来：

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

如果你想确保参数被计算过了，那么你可以使用 **force()** 函数：

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

当使用 **lapply()** 创建**闭包**或者创建**循环**时，这非常重要：

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

```
adders[[10]](10)
#> [1] 20
```

第一次调用**加法器**(**adders**)函数时，**x** 会被**延迟计算**。每次调用**加法器函数**，**x** 都会增加 1，当循环完成时，**x** 的最终值是 10。因此，所有的**加法器函数**都会将输入增加 10，这个可能不是你想要的！我们可以通过手动方式，强制进行计算来修复这个问题：

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

这段代码完全等价于

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

因为 **force** 函数的定义为

```
force <- function(x) x
```

但是，使用这个函数可以清晰地表明你是要进行强制计算，而不是不小心输入了 **x**。

默认参数在函数内部进行计算。这意味着，如果表达式依赖于当前环境，那么结果将是变化的，取决于你是否使用了默认值或显式地提供了一个值。

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

ls() 在 f 内部被计算了:

```
f()
```

```
#> [1] "a" "x"
```

ls() 在全局环境中被计算了:

```
f(ls())
```

```
#> [1] "add" "adders" "adders2" "args" "f"
```

```
#> [6] "funs" "g" "h" "i" "metadata"
```

```
#> [11] "objs" "x" "y"
```

从技术上讲，一个未计算的参数称为**承诺(promise)**，或(很少使用)thunk。（译者注：即**形式参数**转换为**实际参数**的过程）**承诺**是由两部分组成的：

1. 产生**延迟计算**的表达式。（它可以用 `substitute()` 访问。详细信息，请参阅第 13 章。）
2. 创建表达式以及计算表达式的环境。

第一次访问一个**承诺**时，表达式会在它被创建时的环境中进行计算。这个值是缓存的，以便后续访问计算过的**承诺**时，不用再重新计算。（但原始表达式仍然是与值相关联的，因此 `substitute()` 可以继续访问它）你可以使用 `pryr::promise_info()` 找到更多关于**承诺**的信息。它使用了一些 C++ 代码来提取关于**承诺**的信息，但是不会进行计算，这是在纯 R 代码中不可能做到的。

延迟性在 **if** 语句中是有用的：以下的第二个语句将仅在第一个语句为 **TRUE** 时才会被计算。如果不是这样的话，那么该语句会返回一个错误，因为 **NULL > 0** 是一个长度为 0 的逻辑向量，而不是 **if** 的一个有效输入。

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

我们可以自己实现**&&**：

```
`&&` <- function(x, y) {
  if (!x) return(FALSE)
  if (!y) return(FALSE)
  TRUE
}
a <- NULL
!is.null(a) && a > 0
#> [1] FALSE
```

如果没有**延迟计算**，那么这个函数将不能工作，因为 **x** 和 **y** 总是会被计算，即使当 **a** 为 **NULL** 时，它也会测试 **a > 0**。有时，你还可以利用**延迟性**完全消除 **if** 语句。

例如，为了替代下面的代码：

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

你可以这样写：

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

6.4.5 ...

有一种特殊的参数称为`...`。这个参数将匹配任何**尚未匹配的参数**，并且可以很容易地传递给其它函数。如果你想把参数收集起来去调用另一个函数，但是又不想提前说明它们可能的名字，那么这是有用的。`...`常与 S3 **泛型函数** 结合使用，可以让每个方法变得更加灵活。

一个相对复杂的使用`...`的例子是 `base` 包中的 `plot()` 函数。`plot()` 是一个**泛型方法**，它带有参数 `x`、`y` 和 `...`。为了理解对于一个给定的函数，`...` 做了什么，我们需要阅读帮助文档，文档里面说到：“传递给方法的参数，比如图形参数”。最简单的 `plot()` 的调用最终会调用 `plot.default()`，该函数有更多参数，也有 `...`。再次，阅读文档揭示了 `...` 接受“其它图形参数”，这些参数列在 `par()` 函数的帮助文档中。因此，我们可以编写这样的代码：

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

这个例子说明了`...`的优缺点：它可以让 `plot()` 变得非常灵活，但是为了理解如何使用它，我们必须仔细阅读文档。另外，如果我们阅读了 `plot.default` 的源代码，那么我们可以发现在文档中没有提到的一些特性。传递(**pass along**)其它参数到 `Axis()` 和 `box()` 是可能的：

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

如果想要以一种更容易使用的表格方式来捕获`...`，那么你可以使用 `list(...)`。（见第 13.5.2 节使用其它方法来捕获`...`，而不计算参数）。

```
f <- function(...) {
  names(list(...))
}
```

```
f(a = 1, b = 2)
#> [1] "a" "b"
```

使用...是要付出代价的，任何拼写错误的参数都不会得到错误提示，任何...后的参数都必须使用全名。这使得拼写错误很容易被忽视：

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

通常，显式比隐式要好，所以你可能会要求用户提供一个额外参数的列表。如果你尝试使用...与多个附加函数，那当然更容易。

6.4.6 练习

1. 说明下面一些列奇怪的函数调用：

```
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. 这个函数返回什么？为什么？它说明了什么原则？

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

3. 这个函数返回什么？为什么？它说明了什么原则？

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

6.5 特殊调用

R 语言支持其它两种语法来调用特殊类型的函数：**中缀函数**和**替换函数**。

6.5.1 中缀函数

大多数 R 中的函数是**前缀操作符**：函数的名称排在参数的前面。你还可以创建**中缀函数**，函数名位于它的参数之间，比如`+`或`-`。所有用户创建的中缀函数都必须以`%`开始和结束，R 预定义了这些中缀函数：

`%>%`、`%*%`、`%/%`、`%in%`、`%o%`、`%x%`。

不需要`%`的内置中缀操作符的完整列表为：

```
::, :::, $, @, ?, *, /, +, -, >, >=, <, <=, ==, !=, !, &, &&, |, ||, ~, <-, <<-
```

例如，我们可以创建一个新的操作符把字符串连接在一起：

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

注意，在创建函数时，你必须把这个名字放在重音符````里面，因为它是一个特殊的名字。这只是调用普通函数的一种**语法糖(sugar)**而已(译者注：语法糖是编程语言提供的用于简化语句的功能。)；对 R 来说，这两个表达式没有区别：

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

或

```
1 + 5  
#> [1] 6
```

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

中缀函数的名字比普通 R 函数更加灵活：它们可以包含任何字符序列(当然，除了%以外)。你需要在定义函数的时候，对任何特殊字符进行转义，而不是在调用的时候进行转义：

```
`%%` <- 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"
```

R 的默认优先规则意味着中缀操作符是从左到右进行结合的：

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

我经常使用一个中缀函数。它的灵感来自于 Ruby 的`||`逻辑"或"运算符，它与 R 语言中的逻辑"或"运算符有所不同，因为在 `if` 语句中，Ruby 对于计算为 `TRUE` 有更灵活的定义。它很有用：在另一个函数的输出为 `NULL` 的情况下，可以提供一个默认值：

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


6.5.2 替换函数

替换函数的行为表现得好像它们可以**就地修改**(译者注: modify in place, 即修改立即生效, 直接作用在被修改的对象上)参数, 并且它们都拥有特别的名字 **xxx<-**。它们通常有两个参数(**x** 和值), 虽然它们可以有更多参数, 但是它们必须返回**修改过的对象**。例如, 下面的函数允许你修改向量的第二个元素:

```
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
```

当 R 计算赋值语句 **second(x) <- 5** 时, 它注意到左手边的**<-**不是一个简单的名称, 因此它寻找一个命名为 **second<-** 的函数来进行替换操作。

我之所以说它们"表现得好像"可以**就地修改**参数, 是因为实际上它们创建了一个修改后的副本。我们可以使用 **pryr::address()** 来查看, 找到底层对象的内存地址。

```
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**就地修改**参数:

```
x <- 1:10
address(x)
#> [1] "0x103945110"
x[2] <- 7L
address(x)
#> [1] "0x103945110"
```

认识到这种行为是很重要的，因为它有重要的性能影响。如果你想提供其它参数，那么它们会位于 `x` 和值之间：

```
`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
```

当你调用 `modify(x, 1) <- 10` 的时候，R 在后台把它转化为：

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

意味着你不能这么做：

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

因为这就变成了无效的代码：

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

把替换和取子集操作结合起来通常很有用处：

```
x <- c(a = 1, b = 2, c = 3)
names(x)
```

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

这样是可行的，因为表达式 `names(x)[2] <- "two"` 是这样被计算的，相当于你写了下面的代码：

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

(是的，它确实创建了一个名为 `*tmp*` 的局部变量，该变量在使用之后被删除了。)

6.5.3 练习

1. 创建一个列表，里面包含了所有 `base` 包中的替换函数。其中哪些是原语函数？
2. 什么样的名字对用户创建的中缀函数是有效的？
3. 创建中缀形式的 `xor()` 运算符。
4. 创建中缀版的集合操作函数 `intersect()`、`union()` 和 `setdiff()`。
5. 创建一个替换函数，它可以随机修改向量中的某个位置的元素。

6.6 返回值

函数中最后一个表达式的计算结果会成为函数的返回值，也就是调用函数的结果。

```
f <- function(x) {
  if (x < 10) {
```

```
0
} else {
10
}
}
f(5)
#> [1] 0
f(15)
#> [1] 10
```

一般来说，当你的函数需要提前进行返回时，比如发生了错误，或者是很简单的函数，那么，我认为明确地使用 `return()` 为函数返回值，是一种好的风格。这种编程风格也可以减少缩进层级，也通常使得函数更容易理解，因为你可以直观地看到返回值的原因。

```
f <- function(x, y) {
if (!x) return(y)
# 这里是复杂的处理过程
}
```

函数只能返回一个对象。但这不是一个限制，因为你可以返回一个列表，列表里面可以包含任意数量的对象。

最容易理解的函数是**纯函数**(pure function)：这种函数总是把相同的输入映射到相同的输出，并且不影响工作空间。换句话说，**纯函数**没有副作用：除了**返回值**以外，它们不以任何方式影响其它状态。

R 语言提供了保护机制，让你避免一类副作用：大多数 R 对象具有**修改时复制**(copy-on-modify)的语义。所以修改函数参数不会改变原始值：

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

(修改时复制规则有两个重要的例外：**environment** 类和引用类。这些类可以就地修改，所以使用它们时，要格外小心。)

注意，这与 Java 这样可以修改函数输入的语言是不同的。这种**修改时复制**的行为对性能有着重要影响，我们将在第 17 章中进行深入讨论。(注意，R 语言实现**修改时复制**的语义是一种性能影响结果；但是，它们一般都不是这样的。**闭包**是一种新语言，可以让大量使用的**修改时复制**语义只有有限的性能影响。)大多数基础 R 函数是**纯函数**，但是有一些是明显例外的：

1. **library()**，加载一个包，因此修改了**搜索路径**。
2. **setwd()**、**Sys.setenv()**、**Sys.setlocale()**分别改变了**工作目录**、**环境变量**和**语言环境**。
3. **plot()**族函数产生**图形输出**。
4. **write()**、**write.csv()**、**saveRDS()**等，会把输出**保存到磁盘上**。
5. **options()**和 **par()**会修改**全局设置**。
6. **S4** 相关函数修改**类和方法的全局表**。

7. 随机数发生器，在每次运行时，都会产生不同的数字。

一般来说，把使用的副作用降到最低是好主意，并且，如果有可能的话，那么应该通过分离纯的与不纯的函数使副作用最小化。**纯函数**更容易测试(因为你要关注的仅仅是输入值和输出值)，并且不太可能在不同版本的 R 语言上或者在不同的平台上，有不同的行为。

例如，这是 **ggplot2** 的原则之一：大多数操作是作用于**表示图形的对象**上，而只有最终的**打印(print)调用**或者**绘图(plot)调用**才会进行实际的带副作用的绘图操作。函数可以返回不可见的值，也就是在调用函数时，默认不要打印输出。

```
f1 <- function() 1
f2 <- function() invisible(1)
f1()
#> [1] 1
f2()
f1() == 1
#> [1] TRUE
f2() == 1
#> [1] TRUE
```

你可以用括号把不可见的值括起来，进行强制显示：

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

最常用的返回不可见值的函数是 **<-**：

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

这使得它可以把一个值赋给多个变量：

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

因为这会被解析为：

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

6.6.1 退出时

除了返回一个值以外，在函数结束时，函数也可以使用 `on.exit()` 函数，设置其它的触发动作。这是一种在函数退出时，恢复全局状态的常用方法。无论函数是如何退出的，比如明确的(早期的)返回、发生了错误或者干脆就是到了函数体的结尾，`on.exit()` 中的代码总是会执行。

```
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"
```

基本模式很简单：

首先，我们将工作目录设置到一个新的位置，使用 `setwd()` 的输出来获取当前的位置。

然后，我们使用 `on.exit()`，以确保无论函数在何时退出，都要把工作目录恢复到原来的值。

最后，我们明确地强制计算代码。(我们在这里实际上并不需要 `force()`，但是它让读者明白我们要做什么。)

警告：如果你在一个函数中调用多个 `on.exit()` 函数，那么请务必设置 `add = TRUE`。不幸的是，在 `on.exit()` 中 `add` 的默认值是 `add = FALSE`，这样每次你运行它的时候，它都会覆盖已有的退出(`exit`)表达式。由于 `on.exit()` 的特殊实现方式，因此无法创建一个默认设置为 `add = TRUE` 的变种函数，所以使用它的时候必须小心。

6.6.2 练习

1. `source()` 的 `chdir` 参数如何与 `in_dir()` 进行比较？与其它方法相比，说说你为什么更喜欢你选择的方法？
2. 什么函数可以解除 `library()` 的操作？如何保持和恢复 `options()` 和 `par()` 的值？
3. 编写一个函数，它会打开一个图形设备，然后运行提供的代码，最后关闭图形设备(要求无论绘图的代码是否工作，总是会执行)。
4. 我们可以使用 `on.exit()` 来实现一个简单的版本 `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"
```


比较 `capture.output()` 和 `capture.output2()`。这两个函数有什么不同？为了让关键思想更容易被看到，我删除了什么特性？为了更容易理解，我是如何重写关键思想的？

6.7 小测验答案

1. 函数的三个组成部分是函数体、参数列表和环境。
2. `f1(1)` 返回 11。
3. 通常写成中缀形式： `1 + (2 * 3)`。
4. 重写的调用形式 `mean(c(1:10, NA), na.rm = TRUE)` 更容易理解。
5. 不，它不会抛出错误，因为第二个参数是从未使用过的，所以它并没有进行计算。
6. 看第 6.5.1 节和第 6.5.2 节。
7. 使用 `on.exit()`；更详细的描述看第 6.6.1 节。

7 面向对象指南

本章是认识和使用 R 语言的**对象**的指南。R 语言有三种**面向对象系统**(还要加上**基本类型**)，所以听起来可能有点吓人。本指南的目的并不是让你成为所有四种系统的专家，但是可以帮助你认识到你正在使用哪些系统，并帮助你有效地使用它们。

任何面向对象系统的核心都是**类**和**方法**的概念。通过描述**类的属性**以及它与其它**类的关系**，**类**定义了**对象的行为**。**类**也用于选择**方法**，函数会根据输入的不同的**类**，表现出不同的行为。**类**通常被组织成**层次结构**：如果某个**子类的方法**不存在，那么将使用**父类的方法**。**子类**从**父类**那里继承行为。

R 语言的三种**面向对象系统**的差异表现在**类和方法**是怎样定义的：

S3 实现了一种被称为**泛型函数面向对象**的面向对象编程风格。这不同于大多数编程语言，如 Java、C++ 和 C#，它们实现了**消息传递**的面向对象系统。通过**消息传递**，**消息(方法)**被发送到对象，并且由对象来决定要调用哪一个函数。通常，这个对象在方法调用的时候，有特别的展现方式，它通常出现在方法或消息的名称之前：如 `canvas.drawRect("blue")`。但是，S3 是不同的。虽然仍旧是通过方法来进行计算，但是由一种称为**泛型函数**的特殊类型函数来决定调用哪个方法，如 `drawRect(canvas, "blue")`。S3 是一种非常随性的系统。它没有正式的定义。

S4 与 S3 类似，但是更正式。与 S3 相比，它主要有两个不同。S4 有正式的定义，为每个**类**描述了**表示方法**和**继承关系**，并使用特别的辅助函数来定义**泛型函数**和**方法**。S4 也有**多分派**，这意味着**泛型函数**能够选择方法，它可以对泛型函数进行基于多个参数的**方法分派**，而不只是一个。

引用类，简称为 RC(Reference Class)，与 S3 和 S4 有很大的不同。RC 实现**消息传递**的面向对象系统，所以**方法**属于**类**，而不是函数。**\$**用于**分开对象和方法**，所以方法调用看起来像这样：`canvas$drawRect("blue")`。RC 对象也是**可变的**

(mutable): 它们不使用 R 语言常用的**修改时复制**语义, 而是**就地修改**。这使得它们难以理解, 但是可以让它们解决 S3 和 S4 难以解决的问题。

还有另一种系统, 它不是完全面向对象的, 但是它很重要, 所以在这里提一下:

基本类型(base types), 构成其它**面向对象系统**的内部 C 语言级别的类型。基本类型主要是使用 C 代码来操作的, 但是了解它们很重要, 因为其它的面向对象系统都是以它们为基础而构建出来的。

以下章节依次描述每种系统, 从**基本类型**开始。你将学习如何识别一个对象属于哪种面向对象系统, **方法分派**是如何工作的, 以及如何为那个系统创建新的**对象、类、泛型函数和方法**。本章在最后进行了总结, 说明了应该在何时使用每种系统。

前提条件

你需要 **pryr** 包, 可以使用 `install.packages("pryr")` 安装, 它包含一些查看**面向对象属性**的函数。

小测验

认为你知道这些内容了吗? 如果你能正确地回答下面的问题, 那么你可以跳过本章。答案在 7.6 节。

1. 怎样说明一个对象关联到了什么**面向对象系统**(基本、S3、S4 或 RC)?
2. 如何确定一个对象的**基本类型**(如整型或列表)?
3. 什么是**泛型函数**?
4. S3 和 S4 之间的主要区别是什么? S4 和 RC 之间的主要区别是什么?

本章概要

7.1 节教你 R 语言的基本对象系统。只有 R-core 团队才能在这个系统中添加新类，但是了解它是很重要的，因为它支撑着其它三种系统。

7.2 节向你展示了 S3 对象系统的基础知识。它是最简单和最常用的面向对象系统。

7.3 节讨论了更加正式和严格的 S4 系统。

7.4 节教你 R 语言的最新面向对象系统：引用类，或简称为 RC。

7.5 节给你一些建议，当你开始一个新的项目时，应该选择哪种面向对象系统。

7.1 基本类型

每个 R 语言对象的底层都是一个 C 语言结构(即 C 语言的 `struct`)，它描述了该对象是如何存储在内存中的。该结构包括对象的内容、内存管理所需的信息，和本节最重要的东西——类型。这是 R 对象的基本类型。基本类型并不是真正的对象系统，因为只有 R 语言的核心团队才可以创建新的类型。因此，很少会添加新的基本类型：最近的变化，是发生在 2011 年，添加了两个你从来没有在 R 语言中见过的奇异类型，但是它们对于诊断内存问题非常有用(`NEWSXP` 和 `FREESXP`)。在此之前，最后补充的是在 2005 年为 S4 对象添加的一个特殊基本类型(`S4SXP`)。

第 2 章解释了最常见的基本类型(原子向量和列表)，但是基本类型还包括函数、环境和其它更多的特殊对象，比如名字(name)、调用(call)和承诺(promise)，你将在本书的后续章节中学习。你可以使用 `typeof()` 来确定一个对象的基本类型。不幸的是，在 R 中使用的基本类型的名称并不总是一致的，而且，类型和相应的“is”函数也可能使用不同的名称：

```
# 函数的类型是"closure"  
f <- function() {}  
typeof(f)
```

```
#> [1] "closure"
is.function(f)
#> [1] TRUE
# 原语函数的类型是"builtin"
typeof(sum)
#> [1] "builtin"
is.primitive(sum)
#> [1] TRUE
```

你可能听说过 `mode()` 和 `storage.mode()` 函数。我建议忽略这些函数，因为它们只是返回 `typeof()` 得到的名称的别名，它们的存在仅仅是为了与 S 语言保持兼容性。如果你想了解它们到底做了什么，可以阅读它们的源代码。

对不同的基本类型表现不同行为的函数，几乎都是用 C 语言编写的，方法分派时使用了 `switch` 语句(例如，`switch(TYPEOF(x))`)。即使你从未写过 C 代码，理解基本类型也是很重要的，因为其它的一切都是建立在其上的：**S3 对象**可以建立在任何基本类型之上，**S4 对象**使用了一种特殊的基本类型，而**引用类对象**是 S4 和环境(另一个基本类型)的组合体。要查看一个对象是不是一个纯粹的基本类型，即它没有 S3、S4 或 RC 的行为，可以检查 `is.object(x)` 是不是返回 `FALSE`。

7.2 S3

S3 是 R 语言中的第一种和最简单的面向对象系统。这是 `base` 包和 `stats` 包中唯一使用的面向对象系统，也是 CRAN 网站上发布的包中最常用的系统。S3 并不正式，但它非常的优雅和简洁：如果你拿走了 S3 系统中的任何一部分，那么这个系统就不再是面向对象的了。(译者注：这表示 S3 系统已经相当简化了，没有多余的部分了。)

7.2.1 认识对象、泛型函数和方法

你遇到的大多数对象都是 S3 对象。但不幸的是，在基本的 R 语言中，没有简单的方法来测试一个对象是不是 S3 对象。比较接近的测试方法是 `is.object(x)` & `!isS4(x)`，即，它是一个对象，但不是 S4 对象。一种更简单的方法是使用 `pryr::otype()`：

```
library(pryr)
df <- data.frame(x = 1:10, y = letters[1:10])
otype(df) # 数据框是 S3 类
#> [1] "S3"
otype(df$x) # 数值向量不是 S3 类
#> [1] "base"
otype(df$y) # 因子是 S3 类
#> [1] "S3"
```

在 S3 中，方法属于函数，称为泛型函数(generic functions)，或简称为泛型(generics)。S3 的方法不属于对象或类。这不同于大多数其它编程语言，但是这是一种有效的面向对象风格。

要确定一个函数是不是 S3 泛型函数，你可以调用 `UseMethod()` 函数查看它的源代码：这是用来找到需要调用的正确方法的函数，即方法分派的过程。与 `otype()` 类似，`pryr` 还提供了 `ftype()`，如果该函数存在，那么它描述了关联到这个函数的对象系统：

```
mean
#> function (x, ...)
#> UseMethod("mean")
#> <bytecode: 0x7f96ac563c68>
#> <environment: namespace:base>
```

```
ftype(mean)
```

```
#> [1] "s3" "generic"
```

一些 S3 泛型方法，如 `[`、`sum()` 和 `cbind()`，不调用 `UseMethod()` 函数，因为它们是用 C 语言实现的。相反，它们调用 C 函数 `DispatchGroup()` 或 `DispatchOrEval()`。用 C 代码写成的方法分派函数称为内部泛型函数(internal generic)，它们的相关文档可以使用 `? "internal generic"` 来查看。`f`type() 也知道这些特殊函数。给定一个类，S3 泛型函数的工作是调用正确的 S3 方法。你可以通过名字来认识 S3 方法，它们看起来像 `generic.class()`。例如，`Date` 类的 `mean()` 泛型函数是 `mean.Date()`，而 `factor` 类的 `print()` 泛型函数是 `print.factor()`。

大多数的现代编程风格指南，都不鼓励把 `.` 用在函数名中：因为这让它们看起来像 S3 的方法。例如，`t.test()` 是 `t` 对象的 `test()` 方法吗？同样，在类名中使用 `.` 也会造成混淆：`print.data.frame()` 是 `data.frame` 的 `print()` 方法，还是 `frames` 类的 `print.data()` 方法？`pryr::f`type() 知道这些异常情况，因此你可以使用它来查出一个函数是 S3 方法还是泛型函数：

```
ftype(t.data.frame) # 数据框的 t() 方法
```

```
#> [1] "s3" "method"
```

```
ftype(t.test) # t 检验的泛型函数
```

```
#> [1] "s3" "generic"
```

你可以使用 `methods()` 查看所有属于某个泛型函数的方法：

```
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 `base` 包中定义的方法以外，大多数 S3 方法都是不可见的：可以使用 `getS3method()` 阅读它们的源代码)。对于某个给定的类，你还可以列出至少有一个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
```

在接下来的小节中，你将知道无法列出所有的 S3 类。

7.2.2 定义类与创建对象

S3 是一个简单的系统；它没有类的正式定义。要创建一个类的实例，你只需要拿一个已有的基本对象，并设置它的 `class` 属性即可。你可以使用 `structure()` 函数进行创建，或者用 `class<-()` 来设置类：

```
# 在一步中创建并且设置类
foo <- structure(list(), class = "foo")
# 先创建，然后再设置类
foo <- list()
class(foo) <- "foo"
```


S3 对象通常是建立在列表之上或者是带有属性的原子向量。(你可以回忆一下 2.2 节中关于属性的内容)。你也可以把函数转化为 S3 对象。其它的基本类型要么很少在 R 语言中出现, 要么就是具有不寻常的语义, 不容易与属性一起使用。

你可以使用 `class(x)` 来确定任何对象的类, 也可以使用 `inherits(x, "classname")` 来判断一个对象是不是继承自某个特定的类。

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

S3 对象的类可以是一个向量, 它描述了从具体到抽象的行为。例如, `glm()` 对象的类是 `c("glm", "lm")`, 表明广义线性模型(generalised linear models)从线性模型(linear models)继承了行为。类名通常是小写的, 并且你应避免使用 `.`。另外, 对于多个单词的类名, 到底是使用下划线(`my_class`)形式还是使用 `CamelCase(MyClass)`形式, 这都是各说各有理了。大多数 S3 类会提供构造函数:

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

如果构造函数存在(比如 `factor()` 和 `data.frame()`), 那么你就应该使用它。这将确保你使用正确的组件来创建的类的实例。构造函数通常与类具有相同的名称。除了由开发人员提供的构造函数以外, S3 并不检查类的正确性。这意味着, 你可以改变现有对象的类:

```
# 创建一个线性模型
mod <- lm(log(mpg) ~ log(displ), data = mtcars)
```

```
class(mod)
#> [1] "lm"
print(mod)
#>
#> Call:
#> lm(formula = log(mpg) ~ log(displ), data = mtcars)
#>
#> Coefficients:
#> (Intercept) log(displ)
#> 5.381 -0.459
# 把它转换成数据框(!)
class(mod) <- "data.frame"
# 但是，不出所料，这个并不能正常工作
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)
# 但是，数据仍然存在
mod$coefficients
#> (Intercept) log(displ)
#> 5.3810 -0.4586
```

如果你已经使用过其它面向对象语言，那么这可能使你感到不舒服。但令人惊讶的是，这种灵活性并不会造成多大的问题：虽然你可以改变一个对象的类，但是你永远都不应该这么做。R 语言并不能控制你：你可以很容易地搬起石头砸自己的脚。所以，只要你不把枪对准你自己的脚并且扣动扳机，你就不会有什么问题。

7.2.3 创建新方法和泛型函数

要添加一个新的泛型函数，可以创建一个函数，然后调用 `UseMethod()`。

`UseMethod()` 有两个参数：泛型函数的名称，以及用于方法分派的参数。如果你省略了第二个参数，那么它将分派到函数的第一个参数。不需要对 `UseMethod()` 传递任何泛型函数的参数，并且你也不应该这样做。`UseMethod()` 会使用所谓的“黑魔法”来找到它们本身。

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

如果没有一些具体的方法，那么泛型函数是没用的。要添加一个方法，你只需使用正确的名称(`generic.class`)创建一个普通函数：

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

为已有的泛型函数添加一个方法，方式是相同的：

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

正如你看到的，这里并没有进行检查以确保该方法返回的类与泛型函数兼容。而是由你自己来确保你的方法不会违反现有代码的要求。

7.2.4 方法分派

S3 的方法分派(Method dispatch)相对简单。`UseMethod()`创建一个包含函数名的向量, 比如 `paste0("generic", ".", c(class(x), "default"))`, 然后依次寻找每一个函数。"default"类使得为未知的类, 设置一个默认方法提供了可能。

```
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"
# b 类没有方法, 所有使用 a 类的方法
f(structure(list(), class = c("b", "a")))
#> [1] "Class a"
# c 类没有方法, 所以使用默认方法
f(structure(list(), class = "c"))
#> [1] "Unknown class"
```

组泛型方法(Group generic methods)要更复杂一些。组泛型使得为一个函数实现多个泛型方法成为可能。四种组泛型方法以及函数如下所示:

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

组泛型函数是一种相对高级的技术, 超出了本章的范围, 但是你可以使用 `?groupGeneric` 找到更多关于它们的信息。在这里, 最重要的事情是要认识到 `Math`、`Ops`、`Summary` 和 `Complex` 并不是真正的函数, 而是表示了一些函数组。注意, 在组泛型函数内部, 有一个特殊变量 `Generic` 提供了实际的泛型函数调用。

如果你拥有复杂的**类层次结构**，那么有时候调用“父”(`parent`)方法是很有用的。要精确定义它的含义不是很容易，但是基本上就是说，当前的方法并不存在的时候被调用的方法。同样，这也是一种高级技术：你可以使用[?NextMethod](#) 阅读它的文档。

因为方法都是普通的 R 函数，所以你可以直接调用它们：

```
c <- structure(list(), class = "c")
# 调用正确的方法：
f.default(c)
#> [1] "Unknown class"
# 强制 R 调用错误的方法：
f.a(c)
#> [1] "Class a"
```

然而，这与改变对象的类同样危险，你不应该这么干。请不要把装满子弹的枪指向自己的脚！直接调用方法的唯一原因是：有时你可以跳过方法分派过程，从而得到很大的性能提升。相关的详细信息，请参阅第 17.5 节。

你也可以对非 S3 对象调用 S3 泛型函数。**非内部 S3 泛型函数**将分派到基本类型的隐式类。（由于性能原因，**内部泛型函数**不会这么做。）确定基本类型的隐式类的规则有些复杂，不过已经显示在了下面的函数中：

```
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. 阅读 `t()` 和 `t.test()` 的源代码，并确定 `t.test()` 是一个 S3 泛型函数而不是一个 S3 方法。如果你创建一个名为 `test` 的类的对象，并调用了 `t()`，会发生什么？
2. 在基础 R 语言中，有哪些类拥有 `Math` 组泛型函数的方法？阅读源代码。这些方法是如何工作的？
3. R 语言有两种代表日期时间数据的类，`POSIXct` 和 `POSIXt`，它们都继承自 `POSIXlt`。哪些泛型函数对这两种类有不同的行为？哪些泛型函数有同样的行为？
4. 哪一个基础的泛型函数拥有最多的方法？
5. `UseMethod()` 使用一种特殊的方式来调用方法。预测下面的代码将返回什么，然后运行它，并阅读 `UseMethod()` 的帮助，指出发生了什么。尽量以最简单的形式把规则写下来。

```
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. 内部泛型函数不会对基本类型的隐式类分派方法。仔细阅读?"internal generic", 并确定为什么下例中 `f` 和 `g` 的长度是不同的。哪个函数有助于区分 `f` 和 `g` 的行为呢?

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

7.3 S4

S4 以类似于 S3 的方式工作，但它更加正式和严谨。在 S4 中，方法仍然是属于函数的，而不是类，但是：

1. S4 类有正式的定义，描述了它们的字段(fields)和继承结构(inheritance structures) (即，父类(parent classes))。
2. 可以对泛型函数进行基于多个参数的方法分派，而不只是一个。

3. 有一个特殊的操作符, `@`, 它从 S4 对象中提取槽(又名字段)的数据。

所有 S4 相关的代码都存储在 `methods` 包中。当你以交互方式运行 R 的时候, 这个包总是可用的, 但是在批处理模式下, 它可能不是总是可用的。因此, 每当你使用 S4 的时候, 都应该使用显式的 `library(methods)` 加载 `methods` 包, 这是个好主意。S4 是丰富而复杂的系统, 在短短的几页纸中, 是无法完全解释它的。在这里, 我将关注于 S4 底层的关键思想, 这样你就可以有效地使用现有的 S4 对象。如果想了解更多, 那么这里有一些比较好的参考资料:

《S4 system development in Bioconductor》(《Bioconductor 包中的 S4 系统开发》<http://www.bioconductor.org/help/course-materials/2010/AdvancedR/S4InBioconductor.pdf>)

John Chambers 的《Software for Data Analysis》(《数据分析软件》<http://amzn.com/0387759352?tag=devtools-20>)

Martin Morgan 在 `stackoverflow` 上对 S4 问题的解答
(<http://stackoverflow.com/search?tab=votes&q=user%3a547331%20%5bs4%5d%20is%3aanswe>)

7.3.1 认识对象、泛型函数和方法

认识 S4 对象、泛型函数和方法很简单。你可以识别 S4 对象, 因为 `str()` 将它描述为一个 "formal" 类, `isS4()` 返回 `TRUE`, 而 `pryr::otype()` 返回 "S4"。S4 泛型函数和方法也很容易辨认, 因为它们是具有良好定义的类的 S4 对象。

在常用的基础包(`stats`、`graphics`、`utils`、`datasets` 和 `base`)中, 没有 S4 类, 所以我们从内置的 `stats4` 包中开始创建 S4 对象, 它提供了一些与最大似然估计相关的 S4 类和方法:

```
library(stats4)
# 本例来自 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))
# 一个 S4 对象
isS4(fit)
#> [1] TRUE
otype(fit)
#> [1] "S4"
# 一个 S4 泛型
isS4(nobs)
#> [1] TRUE
ftype(nobs)
#> [1] "s4" "generic"
# 取出 S4 方法，在后面描述
mle_nobs <- method_from_call(nobs(fit))
isS4(mle_nobs)
#> [1] TRUE
ftype(mle_nobs)
#> [1] "s4" "method"
```

使用带有一个参数的 `is()` 函数，可以列出一个对象继承自哪些类。使用带有两个参数的 `is()` 函数，可以测试一个对象是否继承自某个特定的类。

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

你可以使用 `getGenerics()` 得到所有 S4 泛型函数的列表，可以使用 `getClass()` 得到所有 S4 类的列表。这个列表还包括了为 S3 类和基本类型创建的 `shim` 类。你可

以使用 `showMethods()` 列出所有的 S4 方法，可以通过泛型函数或者类(或两者)来进行选择。设置 `where = search()` 来限制在全局环境中可用的搜索方法，也是个好主意。

7.3.2 定义类与创建对象

在 S3 中，你可以将任何对象变成某个特定类的对象，只需要设置 `class` 属性而已。S4 则严格得多：你必须使用 `setClass()` 定义一个类的表示方式，然后使用 `new()` 创建一个新对象。你可以使用特殊的语法找到类的文档：`class?className`，比如 `class?mle`。

S4 类有三种关键属性：

一个名字：一个由字母和数字构成的类标识符。按照惯例，S4 类使用 `UpperCamelCase` 风格的名称。（译者注：即每个单词的首字母要大写，像骆驼的驼峰似的。）

一个命名列表表示的一些槽(字段)，它定义了槽的名称和允许的类。例如，一个 `person` 类可能由一个字符型的姓名和一个数值型的年龄来表示：`list(name = "character", age = "numeric")`。

一个字符串给出了它继承自哪个类，或者用 S4 的术语来说，它包含(contain)的类。你可以提供多个类来实现**多重继承**，但这是一种高级技术，会增加复杂性。

在**槽**以及**包含的类**中，你可以使用 S4 类、使用 `setOldClass()` 进行注册的 S3 类，或者使用基本类型的隐式类。在槽中你也可以使用特殊类 `ANY`，它不限制输入。

S4 类有其它可选的属性，比如测试对象是否有效的 `validity` 方法，以及定义了默认槽值的原型对象。使用 `?setClass` 查看更多的细节。

下面的示例创建了一个 `Person` 类，它有名字和年龄两个字段，以及一个继承自 `Person` 类的 `Employee` 类。`Employee` 类从 `Person` 类继承了槽和方法，并添加了

一个额外的槽，`boss`。要创建对象，我们使用类名来调用 `new()` 函数，以及槽值的 "名称-值" 对。

```
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)
```

大多数 S4 类会有一个与类名相同的构造函数：如果存在构造函数，则应该使用它，而不是直接调用 `new()`。可以使用 `@` 或 `slot()` 来访问 S4 对象的槽：

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

(`@` 相当于 `$`，`slot()` 相当于 `[[`。)

如果一个 S4 对象包含(继承自)S3 类或者基本类型，那么它将有一个特别的 `.Data` 槽，它会包含底层的基本类型或者 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
```

由于 R 语言是一种交互式的编程语言，所以有可能在任何时候创建新类或者重新定义现有的类。当你使用 S4 进行交互式实验时，这可能是一个问题。如果你修改了一个类，那么请确保你也重新创建了该类的任何对象，否则，你将会得到无效的对象。

7.3.3 创建新的方法和泛型函数

S4 提供了特殊的函数来创建新的泛型函数和方法。`setGeneric()` 创建一个新的泛型函数，或者将现有的函数转换成一个泛型函数。`setMethod()` 则需要泛型函数的名称、方法应该关联的类以及一个实现该方法的函数。例如，以 `union()` 为例，它通常只对向量有效，现在我们要使它能够对数据框也有效：

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

如果你从头创建一个新的泛型函数，那么你需要提供一个调用了 `standardGeneric()` 的函数：

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

`standardGeneric()` 是 `UseMethod()` 在 S4 系统中的等价函数。

7.3.4 方法分派

如果仅对有单个父类的单个 S4 类进行泛型函数分派，则 S4 的方法分派与 S3 相同。主要的区别是如何设置默认值：S4 使用特殊类 `ANY` 来匹配任何类，以及使用 `"missing"` 来匹配一个缺失参数。类似于 S3，S4 也有组泛型函数，可以使用 `?S4groupGeneric` 查看相关文档，以及使用 `callNextMethod()` 来调用“父”方法。

如果对多个参数进行方法分派，或者如果你的类使用了多重继承，那么方法分派会变得复杂得多。使用 `?Methods` 可以查看规则的描述，但是它们是很复杂的，很难预测将调用哪个方法。出于这个原因，我强烈建议，除非绝对必要，否则应该尽量避免多重继承和多分派。

最后，如果给出了泛型调用的说明，那么有两种方法可以找到调用了哪个方法：

```
# 通过方法: 传入泛型名和类名  
selectMethod("nobs", list("mle"))  
# 通过 pryr: 传入未计算的函数调用  
method_from_call(nobs(fit))
```

7.3.5 练习

1. 哪一个 S4 泛型函数拥有最多的为它定义的方法？哪一个 S4 类拥有最多的与它关联的方法？

2. 如果你定义了一个新的 S4 类，但是并没有**包含**现有的类，会发生什么？(提示：使用`?Classes`了解一下**虚类**(virtual classes)。)
3. 如果你把 S4 对象传给 S3 泛型函数，会发生什么？如果你将 S3 对象传给 S4 泛型函数，会发生什么？(提示：第二种情况可以阅读`?setOldClass`)。

7.4 引用类

引用类(Reference Class, 简称 RC)是 R 语言中最新的面向对象系统。它们是在 R 语言的 2.12 版中引入的。它们完全不同于 S3 和 S4，这是因为：

引用类的方法属于对象，而不是函数。

引用类是可变的对象：R 语言中通常的**修改时复制**语义对它不适用。

这些属性使得引用类对象表现得更像在其它大多数编程语言中的对象，比如 Python、Ruby、Java 和 C#。引用类是使用 R 代码实现的：它们是封装在一个环境中的特殊 S4 类。

7.4.1 定义类与创建对象

由于在基础 R 包中没有提供任何引用类，所以我们首先要创建一个。引用类最好的应用是描述有状态的对象——随时间变化的对象，所以我们会创建一个简单的类来模拟银行账户。创建一个新的引用类与创建一个新的 S4 类很相似，但你要使用 `setRefClass()` 而不是 `setClass()`。首先，仅需的参数是一个由字母和数字组成的名字。虽然你也可以使用 `new()` 来创建新的引用类对象，但是利用 `setRefClass()` 返回的对象来生成新对象是一种更好的风格。(你也可以对 S4 类这样做，但不太常见)。

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

`setRefClass()` 还接受“名字-类”对的列表，它定义了类的字段(相当于 S4 的槽)。额外的命名参数传递给 `new()`，将设置字段的初始值。你可以使用 `$` 来存取字段值：

```
Account <- setRefClass("Account",  
fields = list(balance = "numeric"))  
a <- Account$new(balance = 100)  
a$balance  
#> [1] 100  
a$balance <- 200  
a$balance  
#> [1] 200
```

你可以提供单个参数的函数作为访问器方法，而不是提供一个字段的类名。当存取字段时，这允许你添加自定义行为。可以查看 `?setRefClass` 获取更多的细节。注意，引用类对象是可变的，即，它们具有引用语义，而不是修改时复制：

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

出于这个原因，引用类对象有一个 `copy()` 方法，它允许你复制这个对象：

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

如果没有用方法定义行为，那么对象是没那么有用的。引用类方法与类相关联，并且可以就地修改字段。在下面的示例中，请注意，你通过名字来访问字段的值，并且使用<<-来修改它们。在 8.4 节，你会了解到更多关于<<-的内容。

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

可以使用与访问引用类的域相同的方式来调用方法：

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

setRefClass() 最后一个重要的参数是 contains。这是一个父引用类的名字，当前类是从它继承过来的。下面的示例创建了一个新类型的银行账户，它通过返回错误来防止余额低于 0。

```
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
```

所有引用类最终都是从 `envRefClass` 继承的。它提供了有用的方法，比如 `copy()` (如上所示)、`callSuper()` (调用父类的字段)、`field()` (给定字段的名称，得到它的值)、`export()` (相当于 `as()`) 以及 `show()` (控制打印操作)。看 `setRefClass()` 的 `inheritance` 一节获取详情。

7.4.2 认识对象和方法

你可以识别出引用类对象，因为它们是继承自 "refClass" (`is(x, "refClass")`) 的 S4 对象 (`isS4(x)`)。 `pryr::otype()` 将返回 "RC"。引用类方法也是 S4 对象，是 `refMethodDef` 类。

7.4.3 方法分派

在引用类中，方法分派很简单，因为方法与类关联，而不是函数。当你调用 `x$f()` 的时候，R 将在类 `x` 中寻找方法 `f`，如果没有找到 `f`，则在 `x` 的父类中寻找，如果仍然没有找到，则在 `x` 的父类的父类中寻找，以此类推。在方法内部，你可以使用 `callSuper(...)` 直接调用父类的方法。

7.4.4 练习

1. 使用一个**字段函数**(field function)来阻止直接对帐户余额的操作。(提示：创建一个"隐藏的"余额(.balance)字段，并且阅读在 `setRefClass()` 中关于字段参数的帮助文档)。
2. 我曾说在基础的 R 语言中，不存在任何引用类，这个有点过于简单化了。使用 `getClass()` 找到哪些类是从 `envRefClass` 类继承扩展(`extend()`)而来的。这些类是作为什么来使用的？(提示：回忆一下如何查找一个类的帮助文档)。

7.5 选择一种系统

对于一种语言来说，拥有三种面向对象系统算是很多的，但是对大多数的 R 编程来说，S3 就足够了。在 R 中，你通常为已有的泛型函数，比如 `print()`、`summary()` 和 `plot()`，创建相当简单的对象和方法。S3 是适合这种任务的，我在 R 中写的大多数面向对象代码都是使用 S3。S3 是有点古怪，但是使用它来完成工作只需要最少量的代码。

如果要创建更复杂的相关对象的系统，那么 S4 可能更合适。由 Douglas Bates 和 Martin Maechler 创建的 `Matrix` 包是一个很好的例子。它的目的是高效地存储和计算许多不同类型的稀疏矩阵。在第 1.1.3 版中，它定义了 102 个类和 20 个泛型函数。这个包写得很好，获得了好评，并且 `vignette(vignette("Intro2Matrix", package = "Matrix"))` 为该包的结构给出了很好的概览。`Bioconductor` 包也深入地使用了 S4，它需要对生物学对象之间复杂的相互关系进行建模。`Bioconductor` 为学习 S4 提供了很多很好的资源

(<https://www.google.com/search?q=bioconductor+s4>)。如果你已经掌握了 S3，那么 S4 是相当易学的；所有的思想都一样，只不过更正式、更严格、更详细。

如果你使用过一种主流的面向对象编程语言进行编程，那么使用引用类会非常自然。但是由于它们可以通过可变状态引入不利影响，所以它们更难以理解。例

如，通常在 R 中，当你调用 `f(a, b)` 的时候，你可以假定 `a` 和 `b` 不会被修改。但是，如果 `a` 和 `b` 是引用类对象，那么它们可能被就地修改。通常，当使用引用类对象的时候，你都希望尽量减少不利影响，并且只在不得不使用可变状态的情况下，才使用它们。大部分的函数仍然应该是“函数式的”(functional)，以避免不利影响。这使得代码更容易让其它 R 语言程序员理解。

7.6 小测验答案

1. 要确定对象的面向对象系统，可以使用排除法。如果 `!is.object(x)`，那么它是基本对象。如果 `!isS4(x)`，那么它是 S3 对象。如果 `!is(x, "refClass")`，那么它是 S4 对象；否则，它就是引用类对象。
2. 使用 `typeof()` 来确定对象的基类。
3. 一个泛型函数根据输入的类调用特定方法。在 S3 和 S4 对象系统中，方法属于泛型函数，而不是像其它编程语言一样属于类。
4. S4 比 S3 更加正式，并且支持多重继承和多分派。引用类对象具有引用语义，它的方法属于类，而不是函数。

8 环境

环境(environments)是管理**作用域**的数据结构。这一章将深入探索环境，深入地描述它们的结构，并且使用它们来提高你对四个**作用域规则**的理解，这些规则是在 6.2 节中描述的。

由于环境具有的能力，它们也是一种有用的数据结构，因为它们具有**引用语义**。当你在一个环境中修改**绑定(binding)**的时候，环境不会进行复制，而是**就地修改**。**引用语义**并不经常使用，但是可以变得非常有用。

小测验

如果你能正确回答下列问题，那么你已经知道了本章中最重要的话题。你可以在末尾的 8.6 节找到答案。

1. 列出至少三个方面来说明环境是不同于列表的。
2. **全局环境**的**父环境**是什么？哪种独一无二的环境是没有父环境的？
3. 函数的**封闭环境**是什么？为什么它很重要？
4. 如何确定一个**函数调用**所处的环境？
5. **<-**和**<<-**有什么不同？

本章概要

8.1 节介绍环境的基本属性，并向你展示怎样创建环境。

8.2 节提供了一个**带着环境进行计算的函数模板**，并用一个有用的函数来说明这个思路。

8.3 节更深入地修正了 R 的**作用域规则**，显示它们如何对应于环境的四种类型，而这些环境是与每个函数相关联的。

8.4 节描述了名称必须遵循的规则，并显示了一些把名称绑定到一个值的方式。

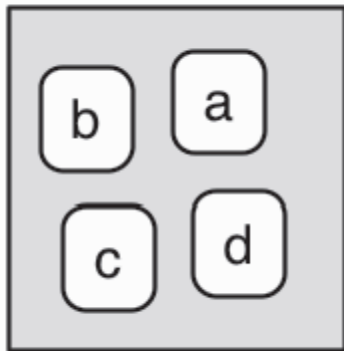
8.5 节讨论了三个问题，环境是有用的数据结构，在作用域中具有独立的作用。

前提条件

本章使用了许多来自于 `pryr` 包的函数，用以打开 R，看看它内部的细节。你可以通过 `install.packages("pryr")` 安装 `pryr` 包。

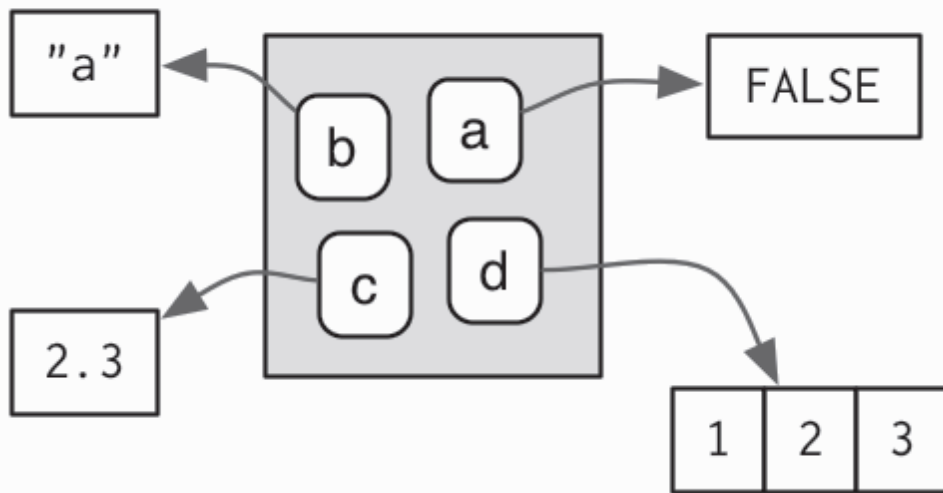
8.1 环境基础

环境的工作是关联或者绑定一组名称到一组值。你可以把环境想象成一个装了名字的袋子：



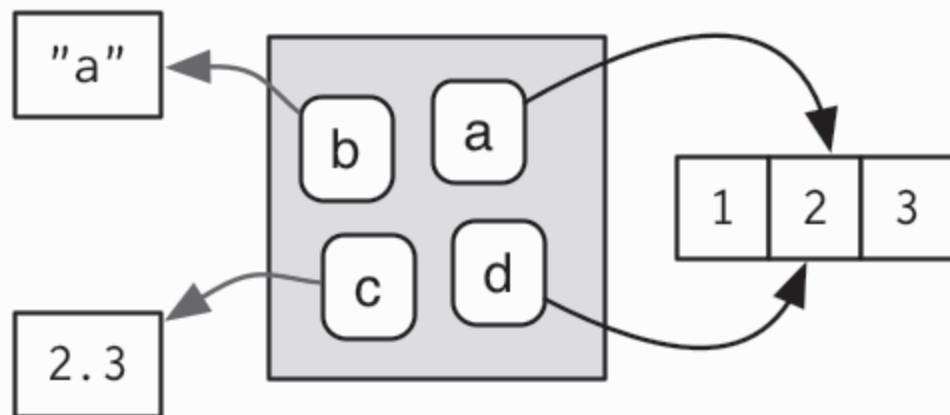
每个名字都指向一个对象，这个对象保存在内存中的某个地方：

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



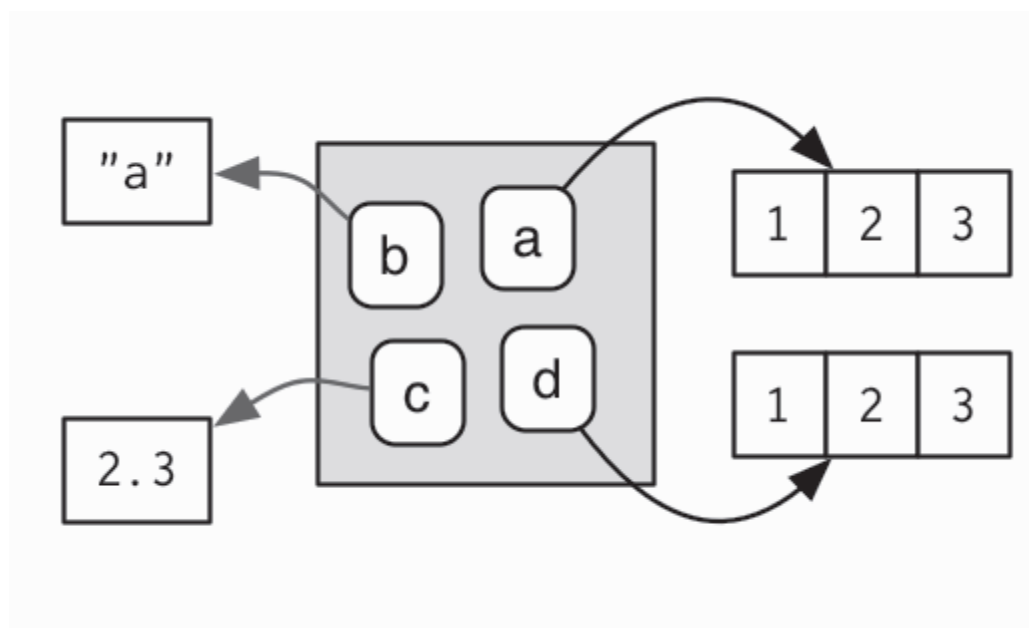
对象并不存在于环境中，所以多个名字可以指向同一个对象：

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



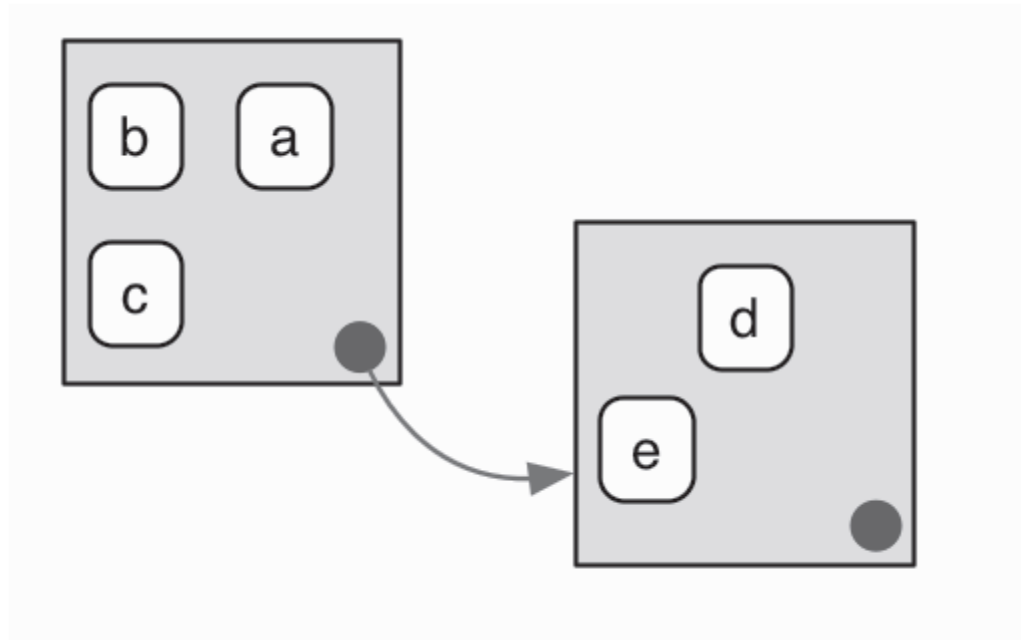
让人迷惑的是，它们也可以指向具有相同值的不同对象：

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



如果一个对象没有名字指向它，那么它会被**垃圾收集器**(garbage collector)自动删除。这个过程在 18.2 节会更详细地描述。

每个环境都有**父环境**，也就是另一个环境。在图表中，我将使用一个小黑圈代表指向父环境的**指针**。**父环境**是用来实现**词法作用域**的：如果在一个环境中没有找到某个名称，那么 R 将在它的父环境中查找，以此类推。只有一种环境没有父环境：**空环境**。



我们使用家庭关系来比喻环境。环境的**祖父环境**是父环境的父环境，而**祖先环境**则包括所有的父环境——从第一级父环境开始一直上溯到最高的**空环境**。很少会谈到环境的**子环境**，因为不存在从父环境指向子环境的链接：给定一个环境，我们无法找到它的**子环境**。一般来说，环境类似于列表，但是有四个重要区别：

1. 在一个环境中的每个对象都有**唯一的名称**。
2. 在一个环境中的对象是**无序的**。(比如，在一个环境中，查询排在"第一位"的对象是没有意义的)
3. 环境有一个**父环境**。
4. 环境具有**引用语义**。

从技术上讲，环境由两个组成部分：第一，**框架(frame)**，其中包含**名字和对象的绑定**(行为很像一个命名列表)；第二，**父环境**。不幸的是，在 R 中，**框架**的使用并不是一致的。例如，`parent.frame()`并不是给出父环境的框架。相反，它会给出**调用环境(calling environment)**。这将是第 8.3.4 节中要详细讨论的内容。有四种特殊的环境：

1. `globalenv()` 或者叫**全局环境**，是交互工作空间。这是普通的工作环境。全局环境的父环境是上一个你使用 `library()` 或 `require()` 加载的包。
2. `baseenv()`，或者叫**基础环境**，是 `base` 包的环境。它的父环境是空环境。
3. `emptyenv()`，或者叫**空环境**，是所有环境的终极祖先，也是唯一没有父环境的环境。
4. `environment()` 是**当前环境**。

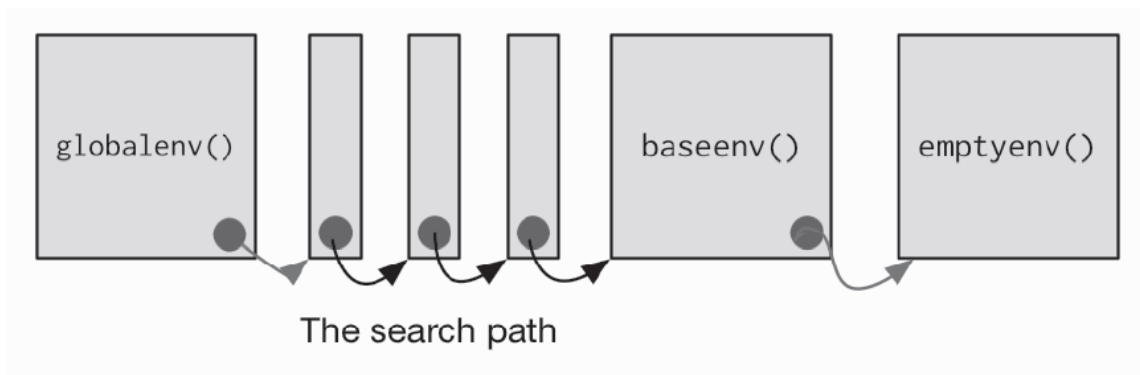
`search()` 会列出全局环境的所有父环境。这被称为**搜索路径**，因为在这些环境中的对象，都可以从顶层的交互工作区中找到。它为每一个加载的包以及其它 `attach()` 的对象都包含一个环境。它还包含一个名为 `Autoloads` 的特殊环境，它用于**按需加载**程序包，以便节省内存(如大型数据集)。

你可以使用 `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()` 的连接方式如下所示。每当你使用 `library()` 加载一个新的包，它就会被插入在**全局环境**以及**先前位于搜索路径顶部的包**之间。



要手动创建一个环境，可以使用 `new.env()`。你可以使用 `ls()` 列出环境框架之中的绑定关系，或者使用 `parent.env()` 查看它的父环境。

```
e <- new.env()
# 由 new.env() 提供的默认的父环境，是它被调用时所处的环境——在这个情况下，是全局环境。
parent.env(e)
#> <environment: R_GlobalEnv>
ls(e)
#> character(0)
```

修改环境中的绑定关系最简单的方式是把它看做列表：

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

默认情况下，`ls()` 仅显示不以 `.` 开头的名字。设置参数 `all.names = TRUE` 可以显示环境中所有的绑定关系：

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

另一个查看环境的方法是 `ls.str()`。它比 `str()` 更有用，因为它会显示环境中的每一个对象。和 `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，你可以通过 `$`、`[[` 或者 `get()` 来找到它绑定的值：

`$` 和 `[[` 仅在一个环境中查找，如果与名字关联的绑定关系不存在，则返回 `NULL`。
`get()` 使用普通的作用域规则，如果绑定关系找不到，则抛出错误。

```
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列表有所不同。对于列表，你可以把一个条目设置为 `NULL` 进行删除。而在环境中，这将创建一个绑定到 `NULL` 的新的绑定关系。所以，要使用 `rm()` 来删除绑定关系。

```
e <- new.env()
e$a <- 1
e$a <- NULL
ls(e)
#> [1] "a"
rm("a", envir = e)
ls(e)
#> character(0)
```

你可以使用 `exists()` 来确定某个绑定关系是否在环境中存在。而 `get()`，它的默认行为是遵循普通的作用域规则，并且会搜索父环境。如果你不想要这样的行为，那么设置参数 `inherits = FALSE`：

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

要比较环境，必须使用 `identical()` 而不是 `==`：

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

8.1.1 练习

1. 列出环境与列表三个不同的地方。

2. 如果你没有提供显式的环境，那么 `ls()` 和 `rm()` 会在哪里搜索？ `<-` 会在哪里创建绑定关系？
3. 使用 `parent.env()` 和循环(或者递归函数)，验证 `globalenv()` 的祖先包括 `baseenv()` 和 `emptyenv()`。使用相同的基本思路实现你自己的 `search()` 函数。

8.2 在环境中进行递归

环境形成了一棵树，所以编写递归函数通常比较方便。本节将向你展示，如何通过应用关于环境的新知识，来理解很有用的 `pryr::where()` 函数。给定一个名字，`where()` 函数使用 R 的普通作用域规则来寻找这个名字所在的环境：

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

`where()` 的定义很简单。它有两个参数：需要寻找的名字(字符串)，以及搜索开始的环境。(稍后我们将在 8.3.4 节了解为什么 `parent.frame()` 是一个很好的默认值。)

```
where <- function(name, env = parent.frame()) {
  if (identical(env, emptyenv())) {
    # 基本情况
    stop("Can't find ", name, call. = FALSE)
  } else if (exists(name, envir = env, inherits = FALSE)) {
    # 成功情况
    env
  } else {
```

```
# 递归情况
where(name, parent.env(env))
}
}
```

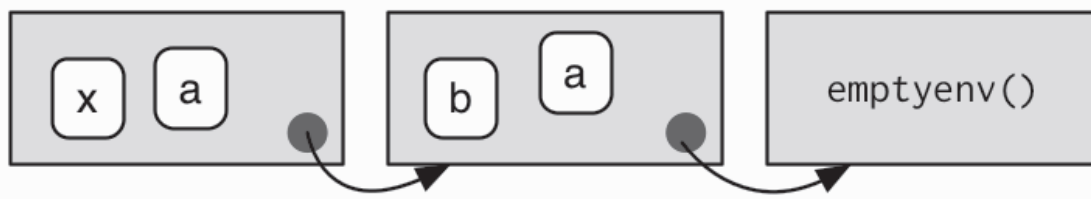
这里有三种情况：

基本情况：我们到达了空环境，并且没有找到绑定关系。我们无法继续搜索了，因此抛出一个错误。

成功情况：名字存在于这个环境中，所以我们返回这个环境。

递归情况：在这个环境中没有找到这个名字，所以尝试搜索它的父环境。

使用例子来看看发生了什么会更容易一些。设想你有如下图所示的两个环境：



如果你正在寻找 **a**，那么 **where()** 会发现它在第一个环境中。

如果你正在寻找 **b**，那么它不在第一个环境中，所以 **where()** 将搜索它的父环境，并在那里找到了 **b**。

如果你正在寻找 **c**，那么它既不在第一个环境中，也不在第二个环境中，那么 **where()** 会到达空环境，并抛出一个错误。

对环境使用**递归方式**是很自然的，所以 **where()** 函数提供了一个有用的模板。把 **where()** 源代码中的一些特征删除以后，可以更清楚地显示递归结构：(译者注：在 R 中使用直接输入 **where** 查看它的源代码。)

```
f <- function(..., env = parent.frame()) {  
  if (identical(env, emptyenv())) {  
    # 基本情况  
  } else if (success) {  
    # 成功情况  
  } else {  
    # 递归情况  
    f(..., env = parent.env(env))  
  }  
}
```

迭代和递归

可以使用循环来代替递归。这可能会略快一点(因为我们消除了一些函数调用),但是我认为这样会更难理解。我在这里提到这个,是因为如果你不太熟悉递归函数的话,你可能会更容易看出发生了什么事情。

```
is_empty <- function(x) identical(x, emptyenv())  
f2 <- function(..., env = parent.frame()) {  
  while(!is_empty(env)) {  
    if (success) {  
      # 成功情况  
      return()  
    }  
    # 检查父环境  
    env <- parent.env(env)  
  }  
  # 基本情况  
}
```

8.2.1 练习

1. 修改 `where()` 函数，使它找到包含以某个名字命名的绑定关系的所有环境。
2. 使用编写 `where()` 函数的风格，来编写你自己版本的 `get()` 函数。
3. 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`，应该遵守针对函数的普通作用域规则：如果存在一个名字匹配的对象，但是并不是函数，那么继续搜索父环境。另一个挑战，添加一个 `inherits` 参数，它控制着函数是否递归到父环境，或者仅仅搜索一个环境。
4. 写一个你自己的 `exists(inherits = FALSE)` 函数(提示：使用 `ls()`)。写一个类似于 `exists(inherits = TRUE)` 的递归版本的函数。

8.3 函数环境

大多数环境并不是由你通过 `new.env()` 来创建的，而是使用函数的结果。本节讨论四类与函数相关的环境：**封闭环境**、**绑定环境**、**执行环境**和**调用环境**。

封闭环境是函数创建时所处的环境。每个函数都有且只有一个封闭环境。

对于其它三种类型的环境，每个函数可以关联 0 个、1 个或者更多个：

使用 `<-` 把一个函数绑定到一个名称，就定义了一个**绑定环境**。

调用函数，创建了一个短暂的**执行环境**，它用于存储执行期间创建的变量。

每一个**执行环境**都关联了一个**调用环境**，它告诉你函数是从哪里调用的。

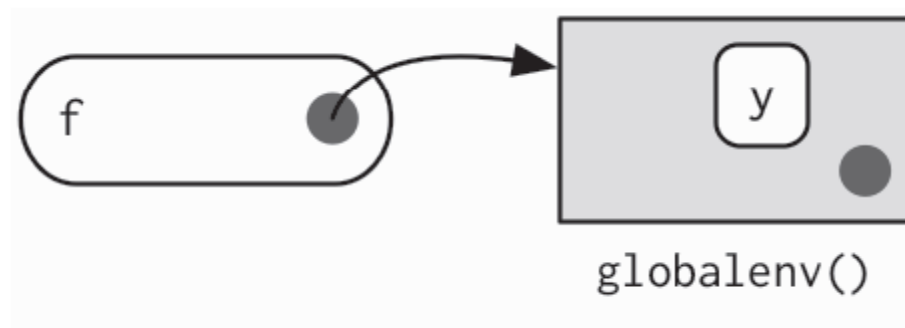
以下部分将解释为什么这些环境很重要，如何访问它们，以及如何使用它们。

8.3.1 封闭环境

创建一个函数时，它会得到它所处的环境的引用。这就是**封闭环境**，用于**词法作用域**。你可以调用 `environment()` 来确定一个函数的**封闭环境**，并把函数作为它的第一个参数：

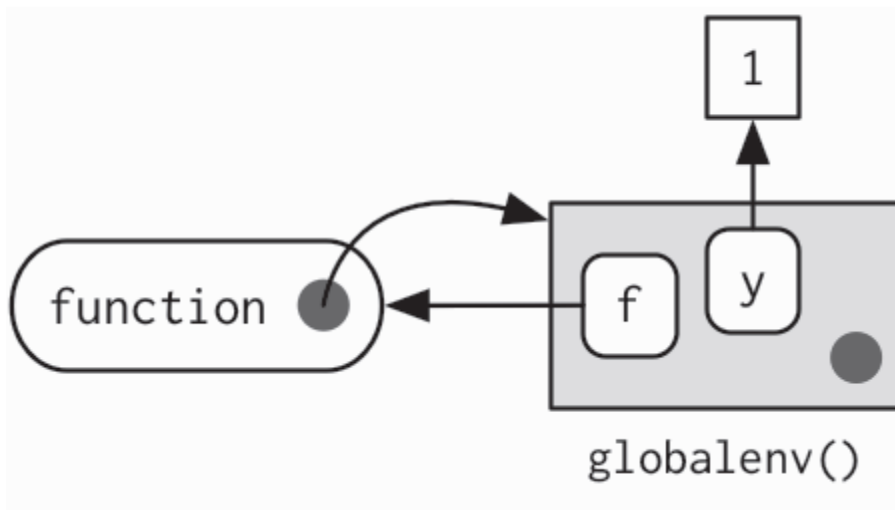
```
y <- 1
f <- function(x) x + y
environment(f)
#> <environment: R_GlobalEnv>
```

在下图中，我将把函数表示成圆角矩形。函数的**封闭环境**是由一个小黑圈表示：



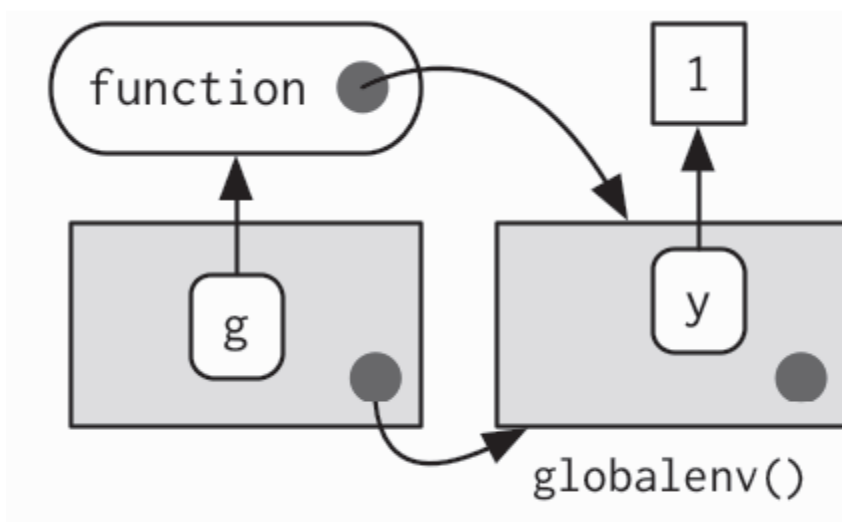
8.3.2 绑定环境

前面的图太简单了，因为函数没有名字。相反，一个函数的名称是由绑定来定义的。函数的**绑定环境**是所有与它有绑定关系的环境。下图更好地反映了这种关系，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。如果你把一个函数指定到不同的环境，那么它们会不同：

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



封闭环境属于函数，并且从来都不会改变，即使函数被搬到一个不同的环境中也是。**封闭环境**决定函数如何寻找值；**绑定环境**决定我们如何找到函数。

对于包的命名空间(namespace)来说, **绑定环境**和**封闭环境**的区别是很重要的。包的命名空间保持了包的独立性。例如, 如果包 **A** 使用了 **base** 包的 **mean()**函数, 那么, 如果包 **B** 也创建了自己的 **mean()**函数, 会发生什么呢? **命名空间** (namespace)确保包 **A** 继续使用 **base** 包的 **mean()**函数, 包 **A** 不会受到包 **B** 的影响(除非明确要求)。

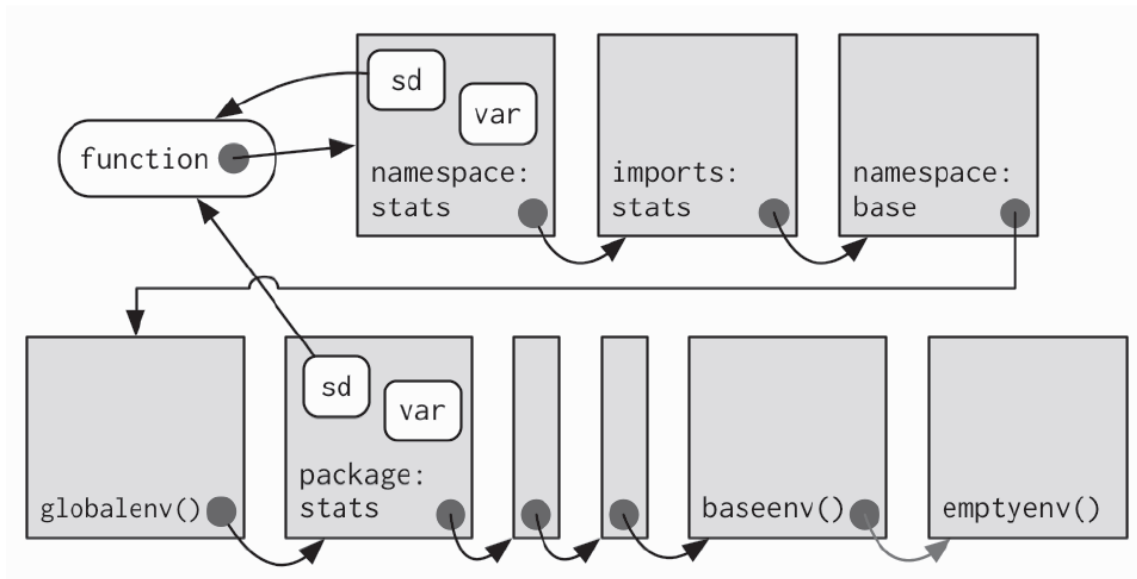
命名空间是使用环境来实现的, 基于这样的事实, 那么函数不需要存在于它们的**封闭环境**中。以基本函数 **sd()**为例。它的**绑定环境**和**封闭环境**是不同的:

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

sd()的定义使用了 **var()**, 但是如果我们创建了自己的 **var()**, 那么并不会影响 **sd()**:

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

这样是可行的, 因为每一个包都有与它关联的两个环境: **包环境**和**命名空间环境**。**包环境**包含所有可公开访问的函数, 并且被放置在了搜索路径之上。**命名空间环境**包含所有函数(包括内部函数), 并且其父环境是一个特殊的**导入(import)**环境, 它包含着这个包需要的所有函数的绑定关系。包中的每个**导出(exported)**函数都被绑定到**包环境**, 但是被**命名空间环境**进行封闭。这种复杂的关系如下图所示:



当我们在控制台输入 `var` 时，它首先在全局环境中被发现。而当 `sd()` 寻找 `var()` 时，它首先在其命名空间环境中发现 `var()`，因此永远都不会搜索 `globalenv()`。

8.3.3 执行环境

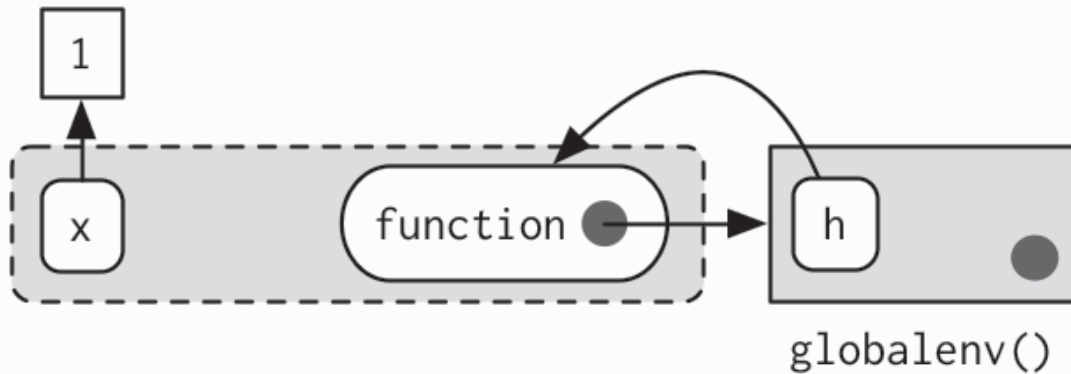
下面的函数第一次会返回什么？第二次呢？

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

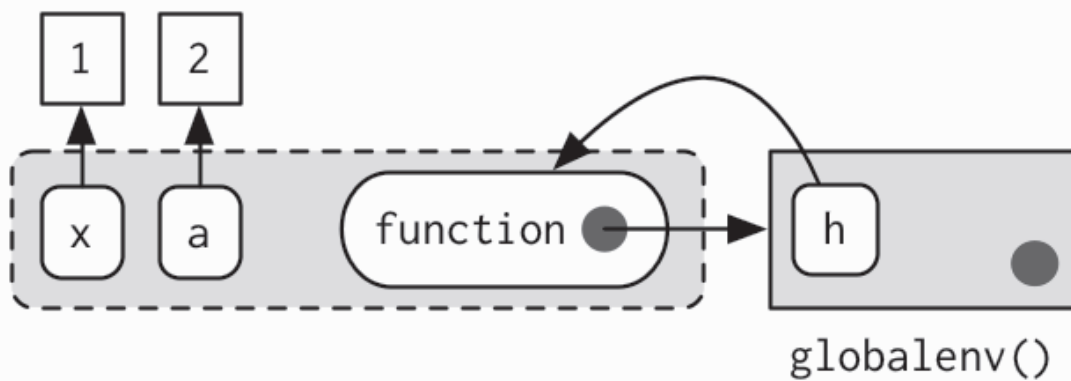
这个函数被调用时，每次都会返回相同的值，这是由于"全新的开始原则"(fresh start principle)，它已经在 6.2.3 节中描述过。每当函数被调用的时候，一个新的环境将被创建出来管理执行过程。**执行环境的父环境**是函数的**封闭环境**。一旦函数执行完毕，这个环境就会被抛弃。让我们使用图形来描述一个更简单的函数。我把函数所属的、围绕着它的**执行环境**，画了虚线边界。

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

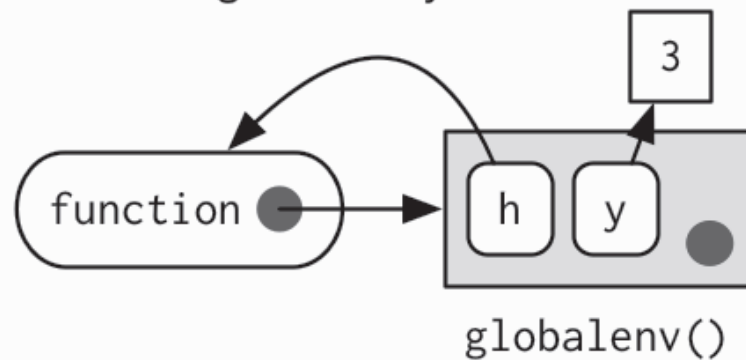
1. Function called with $x = 1$



2. `a` assigned value 2

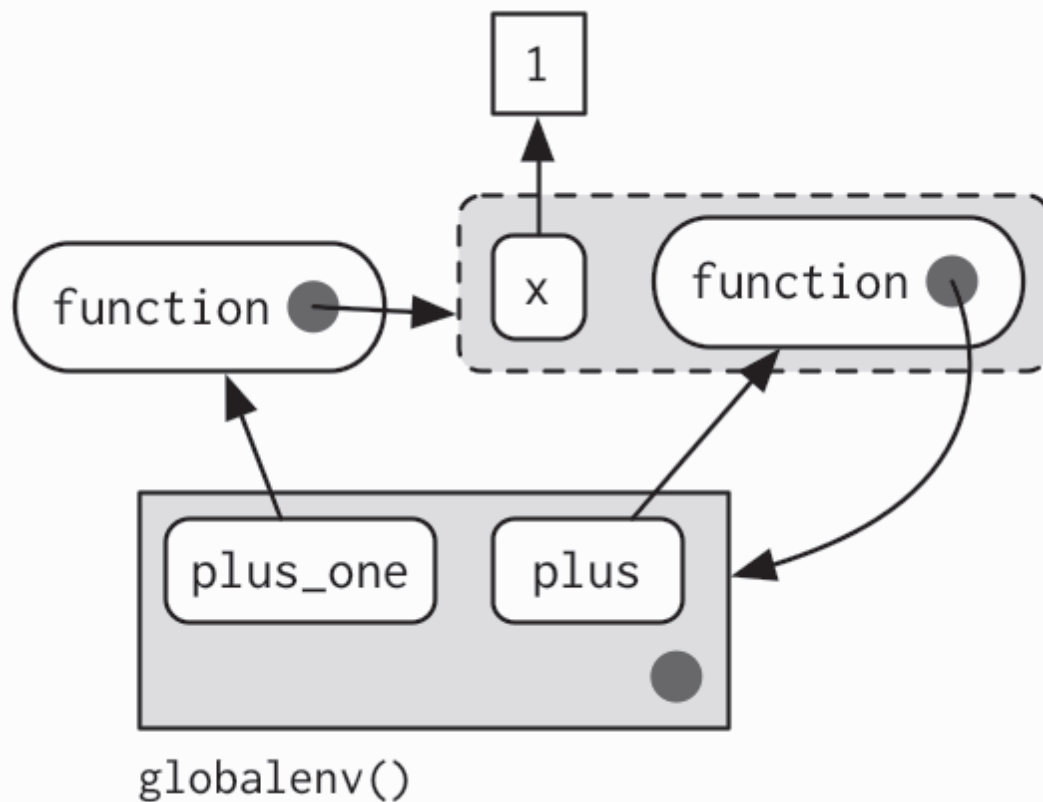


3. Function completes returning value 3. Execution environment goes away.



当你在一个函数中创建另一个函数时，子函数的封闭环境就是父函数的执行环境，并且执行环境不再是临时的。下面的示例使用函数工厂来说明了这种思想，`plus()`。我们使用函数工厂来创建一个名为 `plus_one()` 的函数。`plus_one()` 的封闭环境是 `plus()` 的执行环境，在该环境中，`x` 被绑定到 `1` 这个值。

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



你将在第 10 章学习更多关于函数工厂的知识。

8.3.4 调用环境

请看下面的代码。当运行这段代码的时候，判断一下 `i()` 会返回什么？

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

顶层的 `x` (被绑定到 20) 其实与结果没有什么关系：首先，`h()` 使用普通的作用域规则，在它被定义的环境中进行搜索，然后发现关联到 `x` 的值是 10。然而，在 `i()` 被调用的环境中，询问 `x` 关联到什么值，仍然是有意义的：在 `h()` 被定义的环境中，`x` 是 10，但是在 `h()` 被调用的环境中，它是 20。

我们还可以使用名字没有起好的 `parent.frame()` 来访问这个环境。这个函数返回某个函数被调用时所处的环境。我们也可以使用这个函数来查找那个环境中的名字的值：

```
f2 <- function() {  
  x <- 10  
  function() {  
    def <- get("x", environment())  
    cll <- get("x", parent.frame())  
    list(defined = def, called = cll)  
  }  
}
```

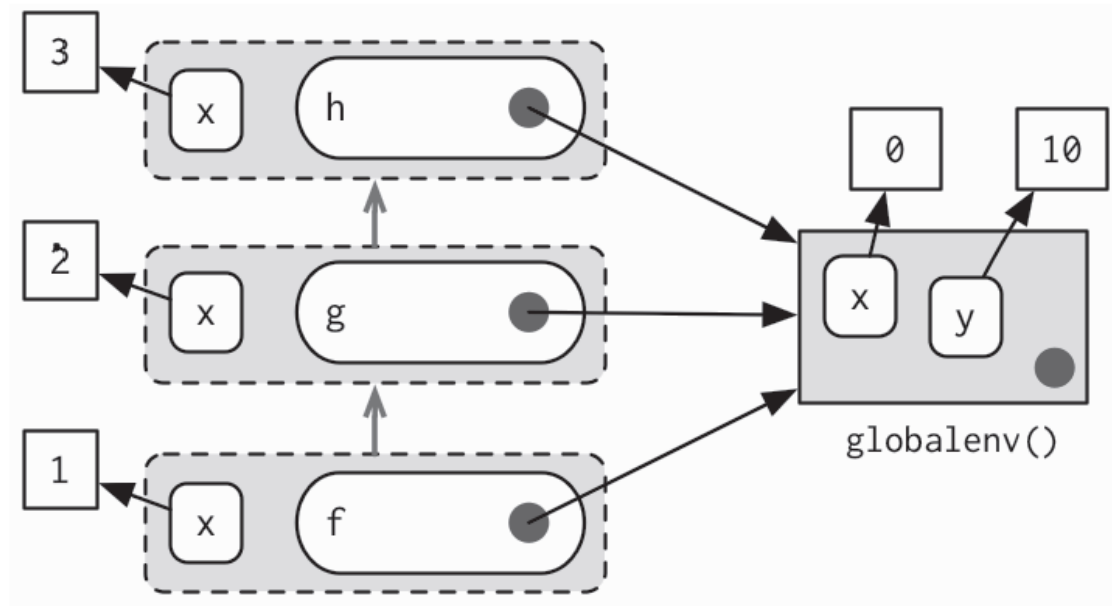


```
g2 <- f2()
x <- 20
str(g2())
#> List of 2
#> $ defined: num 10
#> $ called : num 20
```

在更复杂的场景中，不是只有一个父环境被调用，而是一系列的调用，它会导致从顶层的发起函数开始，一直调用到最底层函数。下面的代码生成了一个三层深的调用栈。开放式的箭头表示每一个执行环境的调用环境。

```
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 语言的常规作用域规则仅使用作为父环境的**封闭环境**；而 `parent.frame()` 允许你访问另外一个作为父环境的**调用环境**。

在**调用环境**中查找变量，而不是在**封闭环境**中查找，称为**动态作用域**。很少有语言实现**动态作用域**(Emacs Lisp 是一个值得注意的例外

(<http://www.gnu.org/software/emacs/emacs-paper.html#SEC15>)。这是因为，**动态范围**使得理解函数是如何运行的，变得困难得多：你不仅需要知道它是如何定义的，你还需要知道它被调用时的上下文环境。**动态作用域**主要是用于开发帮助交互式数据分析的函数。这是第 13 章中讨论的话题之一。

8.3.5 练习

1. 列出与一个函数相关的四种环境。每种环境分别是做什么用的？为什么封闭环境和绑定环境的区别特别重要？
2. 画一个图，显示这个函数的封闭环境：

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

3. 扩展上面的图，以显示函数绑定。
4. 再次扩展上面的图，以显示**执行环境**和**调用环境**。
5. 写一个增强版的 `str()` 函数，使它提供更多关于函数的信息。显示函数是在哪里被发现的，以及它是在什么环境中定义的。

8.4 把名字绑定到值上

在一个环境中，**赋值**就是把一个值绑定(或重新绑定)到一个名字的行为。它是与作用域相对应的，是决定**如何找到与一个名字相关联的值的规则集合**。与其它大多数语言相比，R 语言为绑定名称到值，提供了极其灵活的工具。事实上，你不仅可以**将值绑定到名称**，你也可以把**表达式(承诺)**，甚至**函数**绑定到名称，这样每当你访问与某个名字相关联的值的时候，你都会得到不同的东西！你可能在 R 语言中使用了成百上千次赋值操作。常规的**赋值操作**创建了当前的环境中，一个名称和一个对象之间的绑定。名字通常由字母、数字、`.`和`_`组成，而不能以`_`开始。如果你尝试使用一个不遵循这些规则的名字，那么你会得到一个错误：

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

保留字(比如 `TRUE`、`NULL`、`if` 和 `function`)也遵守该规则，但是它们已经被 R 语言保留，以作它用：

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

完整的保留字列表可以使用 `?Reserved` 找到。可以用重音符包围任何字符序列，这样可以覆盖通常的规则，生成特殊的名字：

```
`a + b` <- 3
`:)` <- "smile"
`` <- "spaces"
ls()
# [1] " " " :)" "a + b"
`:)`
# [1] "smile"
```

引号

你还可以使用单引号和双引号代替重音符，来创建非句法的(non-syntactic)绑定，但是我不推荐这样做。允许在赋值箭头符号的左边使用字符串，是一个历史遗留问题，它是在 R 语言支持重音符之前使用的。常规的赋值箭头符号，`<-`，总是在当前环境中创建一个变量。深赋值(deep assignment)箭头符号，`<<-`，永远不会在当前环境下创建一个变量，而是不断向上搜索父环境，直到找到变量后，直接修改现有变量。你也可以使用 `assign()` 进行深度绑定：`name <<- value` 相当于 `assign("name",value, inherits = TRUE)`。

```
x <- 0
f <- function() {
  x <<- 1
}
```

```
f()
x
#> [1] 1
```

如果`<-`没有找到现有变量，那么它将在全局环境中创建一个。这通常是不可取的，因为全局变量会引入不易察觉的函数之间的依赖关系。`<-`通常是与**闭包**一起使用的，将在 10.3 节描述。

还有两种特殊类型的绑定，**延迟绑定**和**活动绑定**：

延迟绑定创建和存储一个**表达式的承诺**，仅在需要的时候进行计算，而不是立即赋予表达式的结果。我们可以使用特殊赋值运算符`%<d-%`来创建**延迟绑定**，它由 `pryr` 包提供。

```
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-%`是基本函数 `delayedAssign()` 的包装，如果需要更多的控制，你可能需要使用直接它。**延迟绑定**用于实现 `autoload()`，它使 R 表现得似乎包的数据保存在了**内存中**，虽然它只是当你需要数据的时候，从磁盘加载进来而已。

活动绑定不会绑定到常量对象上。相反，每次访问它们的时候，都会重新进行计算：

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

```
x
#> [1] 0.8112
rm(x)
```

`%<a-%`是基本函数 `makeActiveBinding()` 的包装。如果你想要更多的控制，那么你可以直接使用这个函数。活动绑定被用于实现引用类的字段。

8.4.1 练习

1. 这个函数是做什么的？它与 `<-` 有什么不同，你为什么更应该喜欢它？

```
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. 创建另一个版本的 `assign()` 函数，它只会绑定新名称，而永远不会重新绑定旧名称。有一些编程语言只能这样做，它们被称为单赋值语言。

([http://en.wikipedia.org/wiki/Assignment_\(computer_science\)#Single_assignment](http://en.wikipedia.org/wiki/Assignment_(computer_science)#Single_assignment))。

3. 编写一个赋值函数，它可以进行活动绑定、延迟绑定以及锁定(**locked**)绑定。你该给它取什么名字呢？它应该接受什么参数呢？你能根据输入猜出是哪种类型的赋值吗？

8.5 显式环境

本节说明**显式环境**(Explicit environments)。与管理作用域一样，环境也是非常有用的数据结构，因为它们具有引用语义。不像 R 中的其它大多数对象，当你修改一个环境时，它不会进行复制。例如，看看这个 `modify()` 函数。

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

如果你把它应用到列表上，那么原始列表不会改变，因为修改列表实际上是复制了一个新列表，然后改变这个副本。

```
x_l <- list()  
x_l$a <- 1  
modify(x_l)  
x_l$a  
#> [1] 1
```

然而，如果你把它应用到环境上，则原始环境会被修改：

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

正如你可以使用列表在函数之间传递数据一样，你也可以使用环境。当创建你自己的环境时，请注意你应该把它的父环境设置为空环境。这样可以确保你不会不小心从别的地方继承对象：

```
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
```

环境是解决三种常见问题非常有用的数据结构：

避免复制大数据。

管理一个包的状态。

高效地通过名字查找值。

下面将一个一个进行描述。

8.5.1 避免复制

由于环境具有引用语义，因此你永远都不会意外地创建了一个副本。这使得它成为可以包含大对象的很有用的容器。这是 **bioconductor** 包用于管理大基因组对象，经常需要使用的技术。但是，R 3.1.0 中的改变使这种技术变得不那么重要了，因为修改列表不再会进行深拷贝(deep copy)了。此前，修改列表中的一个元素会导致复制所有元素，如果某些元素很大，那么这是一种开销很大的操作。而现在，修改列表时会有效地重用已有的向量，节省了大量时间。

8.5.2 包的状态

在包中，显式的环境是有用的，因为它们允许你在函数调用之间维护包的状态。通常，包中的对象是锁定的，所以你不能直接修改它们。但是，你可以这样做：

```
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)
}
```

从设置器(setter)函数返回旧的值，是一种良好的风格，因为可以联合 `on.exit()` 函数来使用，把值恢复成以前的值。(请看 6.6.1 节)。

8.5.3 作为哈希表

哈希表(hashmap)是一种数据结构，使用它根据名字来查找对象时，查找的时间复杂度是常数的， $O(1)$ 。在默认情况下，环境提供了这种行为，所以它可以用来模拟一个哈希表。可以看看 CRAN 上的 `hash` 包，它应用了这种思路。

8.6 小测验答案

1. 有四种方式：一个环境中的每个对象都必须有一个名称；顺序无关紧要；环境有父环境；环境具有引用语义。
2. 全局环境的父环境是上一个加载的包。唯一没有父环境的环境是空环境。

3. 函数的**封闭环境**是它被创建时所处的环境。它决定了函数在哪里查找变量。
4. 使用 `parent.frame()`。
5. `<-`总是在当前环境下创建一个绑定；`<<-`在当前环境的父环境中，重新绑定一个现有的名字。

9 调试、条件处理和防御性编程

当你的 R 代码出现了错误的时候，会发生什么情况呢？你会怎么做呢？你使用什么工具来解决这个问题？本章将教你如何解决问题(调试)，并且向你展示，函数是如何与用户交流这些问题的，以及基于这些交流的问题，应该如何采取行动(条件处理)，并教你如何避免这些常见问题(防御性编程)。**调试**是解决代码中意想不到的问题的艺术和科学。在这一节中，你将学习能帮你找到错误原因的工具和技术。你将学习通用的**调试策略**、像 `traceback()` 和 `browser()` 这类有用的函数，以及 RStudio 中的交互工具。

不是所有的问题都是意想不到的。当编写一个函数时，你通常可以预测潜在的问题(如文件不存在或错误的输入类型)。与用户交流这些问题，是**条件**(conditions)的工作：**错误**、**警告**和**消息**。

严重错误(Fatal error)由 `stop()` 发起，并强制终止所有执行的程序。当一个函数没有办法继续运行时，要使用**错误**(Error)。

警告(Warning)是由 `warning()` 产生的，用于显示潜在的问题，比如当某些向量化的输入元素无效的时候，如 `log(-1:2)`。

消息由 `message()` 产生，是一种提供信息输出的方法，用户可以很容易地忽略掉它们(`?suppressMessages()`)。我经常使用**信息**来让用户知道，对于一些缺失的重要参数，函数自动选择了什么值。

条件通常是突出显示的，根据你的 R 界面，它会使用**粗体字**或者**显示为红色**。你可以很容易地区分它们，因为**错误**总是以"Error"开头，**警告**总是以"Warning message"开头。

函数的作者也可以使用 `print()` 或者 `cat()` 与用户交流，但我认为这不是一种好主意，因为这种输出**很难进行捕获**或者**有选择地忽略掉**。`print()`的输出不是一个条

件，所以你无法使用任何有用的**条件处理工具**来处理它们，你将在下面学习**条件处理工具**。

条件处理工具(Condition handling tools)，比如 `withCallingHandlers()`、`tryCatch()` 和 `try()`，允许你在条件发生时采取特定的动作。例如，如果你要拟合很多模型，那么即使当某个模型无法收敛时，你可能也想要继续拟合其它的模型。根据 Common Lisp 的思想，R 语言提供了一个异常强大的**条件处理系统**，但是目前并没有很好的文档介绍这方面的内容，使用的也没有那么普遍。这一章会给你介绍最重要的基础内容，但是如果你想了解更多，那么我推荐下面两个资源：

Robert Gentleman 和 Luke Tierney 写的《A prototype of a condition system for R》(《R 的条件系统原型》)(<http://homepage.stat.uiowa.edu/~luke/R/exceptions/simpcond.html>)。它描述了一个早期版本的 R 语言条件系统。虽然自从该文档写成以来，R 语言的实现发生了一些变化，但是它为如何把各种设计片段组合在一起，提供了一种很好的概述，以及进行这些设计的动机。

Peter Seibel 写的《Beyond Exception Handling: Conditions and Restarts》(《超越异常处理:条件和重启》)(<http://www.gigamonkeys.com/book/beyond-exception-handling-conditionsand-restarts.html>)。它描述了 Lisp 的异常处理，与 R 语言使用的方法非常相似。它提供了有用的动机以及更复杂的例子。我为这些例子提供了 R 语言的版本，可以查看 <http://adv-r.had.co.nz/beyond-exception-handling.html>。

本章最后以**防御性编程**的讨论进行总结：如何在常见的错误发生之前避免它们。在短期来说，你可能需要花更多的时间写代码，但从长远来看，你会节省时间，因为**错误消息**将会提供更多的信息，让你更快地找出问题的根源。**防御性编程**的基本原则是**快速失败**(fail fast)，一旦出现问题就要发起错误。在 R 中，这通过三种

特定模式来实现：检查输入的正确性，避免非标准计算，以及避免函数可以返回不同类型的输出。

小测验

想跳过这一章吗？去吧，如果你能回答下面的问题的话。答案在本章末尾的第 9.5 节。

1. 怎样找到一个错误发生在哪里？
2. `browser()` 函数是做什么的？列出五个你可以在 `browser()` 环境里使用的键盘命令。
3. 在代码块中，你使用什么函数忽略错误？
4. 为什么要使用自定义的 S3 类来创建错误？

本章概要

9.1 节概述了发现和解决错误的一般方法。

9.2 节介绍了能帮助你正确定位发生错误的地方的 R 函数 `Rstudio` 的特性。

9.3 节向你展示了如何在你自己的代码中捕获条件(错误、警告和消息)。这使你创建的代码更加健壮，以及在错误存在的时候可以得到更多信息。

9.4 节介绍了**防御性编程**的一些重要技术，这些技术是用于防止错误发生的。

9.1 调试技术

"寻找错误是这样一种过程：确认很多你认为是正确的事情——直到你发现一个是不正确的。"—Norm Matloff

调试代码是具有挑战性的。许多错误是微妙的，而且很难找到。确实，如果错误是显而易见的，那么你可能第一时间就已经能够避免了。虽然，如果你的技术确

实不错，那么你可以仅仅使用 `print()` 来有效地调试问题，但是有时候其它的帮助也是有益的。在本节中，我们将讨论 R 和 RStudio 提供的一些有用的工具，并就一般的调试过程给出一个概要。

下面的过程绝不简单，希望在你调试的时候，能够帮助你组织一下思路。有四个步骤：

1. 意识到程序出现了错误

如果你正在读这一章，那么你可能已经完成了这一步。这是最重要的一步：如果你不知道错误的存在，那么你是不可能修复错误的。这就是为什么在编写高质量代码的时候，自动化测试工具很重要的原因之一。但是，自动化测试超出了本书的范围，不过你可以在 <http://r-pkgs.had.co.nz/tests.html> 上阅读更多关于它的内容。

2. 使错误可以重复出现

一旦你确定程序有错误，你就需要让它可以重现。没有这个过程，错误会变得很难分离出来，也很难找到错误的原因，并且没办法确认你是否已经成功地修复了错误。

通常，我们会从一个大的有错误的代码块开始，然后慢慢地缩小范围，最后得到一段导致错误的、尽可能小的代码片段。**二分查找(binary search)**在这个过程中特别有用。要进行**二分查找**，你将反复删除一半的代码，直到你定位到了错误代码段。这个过程是很快的，因为，每进行一步，你需要查看的代码数量都会减少一半。

如果产生错误需要很长的时间，那么研究一下如何能更快地生成错误也是值得的。你能越快地这样做，就能越快地找出原因。

当你创建一个最小的例子的时候，你可能会发现有些相似的输入并不会引发错误。请把它们记录下来：它们在诊断错误原因的时候会有用。

如果你使用**自动化测试**，那么这也是创建**自动化测试用例**的好时机。如果现有测试用例的覆盖率较低，那么可以借此机会增加一些测试用例，以保证程序现有的良好行为是一直保持着的。这会减少引入新的错误的机会。

3. 找出错误在哪里

如果你够幸运，那么下一节中的工具之一将会帮助你快速识别导致错误的代码。但是通常，你将会不得不更多地考虑这个问题。采取科学的方法是一个好主意。提出假设、设计实验来测试它们，并且记录你的结果。这看起来需要做很多工作，但是系统化的方法将最终会节省你的时间。我经常浪费很多时间并依靠直觉来解决错误("哦，这里一定是一个错误，所以我就在这里减 1 就可以了。"), 但是如果我使用了系统化的方法的话，将会更好。

4. 修改和测试

一旦你发现了错误，你就需要研究如何修复它，然后检查修复工作是不是正确的。这又是自动化测试非常有用的地方。这样不仅会帮助你确认确实修正了错误，也会帮你确保在这个过程中没有引入任何新的错误。在缺乏自动化测试的情况下，一定要仔细记录正确的输出，并且核对之前导致错误的输入。

9.2 调试工具

为了实现调试策略，你需要工具。在本节中，你将了解 R 语言和 RStudio IDE 提供的工具。RStudio 支持集成调试，它使用用户友好的方式来包装现有的 R 语言工具，能让用户使用起来更加轻松。我将向你展示 R 语言和 RStudio 两种方式，这样你就可以在任何环境中进行工作了。你也可以参考官方的 RStudio 调试文档(<http://www.rstudio.com/ide/docs/debugging/overview>)，它总是反映了 RStudio 最新版本的工具。目前，有三种主要的调试工具：

1. RStudio 的**错误检查器**以及能列出导致错误的函数调用序列的 `traceback()`。
2. RStudio 的"以调试方式重新运行"("Rerun with Debug")以及 `options(error = browser)`，可以在错误发生的地方，打开一个交互式会话。
3. RStudio 的**断点**(break point)以及能在代码的任意位置打开一个交互式会话的 `browser()`函数。

下面，我将更详细地解释每个工具。在编写新函数的时候，你应该不需要使用这些工具。如果你发现自己经常对新的代码使用它们，那么你可能需要重新考虑一下你的方法是不是合适。我们应该在小块代码上进行交互式工作，而不是试图一次性编写一个很大的函数。如果你从小事做起，那么你就可以快速识别为什么一些东西不起作用。但是如果你一开始就写大段大段的代码，那么你可能很难找出问题的原因。

9.2.1 确定调用顺序

第一种工具是**函数调用栈**——导致错误的函数调用序列。这里有一个简单的例子：你可以看到，`f()`调用了 `g()`，`g()`调用了 `h()`，`h()`调用了 `i()`，`i()`函数对一个数字和一个字符串做加法，然后抛出了一个错误：

```
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 中运行这段代码的时候，我们可以看到：

```
> f(10)
```

```
Error in "a" + d : non-numeric argument to binary operator
```

[Show Traceback](#)[Rerun with Debug](#)

在**错误消息**的右边会出现两个选项："Show Traceback"和"Rerun with Debug"。如果你点击"Show Traceback"，你将看到：

```
> 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)
```

如果你没有使用 Rstudio，那么你可以使用 `traceback()` 得到同样的信息：

```
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)
```

我们从下到上查看**调用栈**：初始调用是 `f()`，它调用了 `g()`，然后调用了 `h()`，最后是 `i()`，而 `i()` 会引发错误。如果你调用的是使用 `source()` 函数加载到 R 的代码，那么 `traceback` 还将以 `filename.r#linenumber` 的形式显示函数的位置。在 Rstudio 中，这些都是可以点击的，RStudio 将帮你在编辑器中定位到相应的代码行。

有时，这样已经有足够的信息让你跟踪错误并进行修复了。但是，多数情况下，这还是不够的。`traceback()` 显示了错误发生的地方，但是并没有说明是什么原因。下一个有用的工具是**交互式调试器**，它可以让你暂停执行函数以及交互地研究它的状态。

9.2.2 浏览错误

进入**交互式调试器**的最简单方法是通过 RStudio 的"Rerun with Debug"（重新运行与调试）工具。这样会重新运行发生错误的命令，并会在错误发生的地方暂停执


行。在函数内部，你现在进入了一种交互式状态，你可以与在这里定义的任何对象进行交互。你会在编辑器中看到相应的代码(下一条将运行的语句会高亮)，在当前环境中的对象，会显示在"Environment"面板中，调用栈则显示在"Traceback"面板中，并且你可以在控制台中运行任意 R 代码。


和普通 R 函数一样，这里有一些你可以在**调试模式**下使用的特殊命令。你可以通过 RStudio 的工具栏



或者键盘来访问它们：

Next, **n**: 在函数中执行下一步。如果你有一个名为 **n** 的变量，则要小心；你需要使用 **print(n)**来打印它的信息。

Step into,  或 **s**: 与 **next** 类似，但如果下一步是一个函数，那么它将进入该函数，这样你就可以执行函数的每一行代码。

Finish,  或 **f**: 执行完当前的循环或者函数。

Continue, **c**: 离开交互式调试并继续进行函数的常规执行过程。如果你修复了有问题的状态并想检查函数是否能正确执行，那么这个命令是有用的。

Stop, **Q**: 停止调试，终止函数，并返回到全局工作空间。一旦你找出了问题，然后准备修复错误并重新加载代码，你就可以使用这个命令。

另外，还有两个很少用到的命令，它们没有显示在工具栏上：

Enter: 重复上一条命令。我发现很容易就不小心激活了这个命令，所以我使用 **options(browserNLdisabled = TRUE)**来关掉它。

where: 打印当前调用的堆栈跟踪信息(相当于是在交互状态下的 **traceback()**)。

要想在 RStudio 之外进入这种调试风格, 可以使用 **error** 选项, 它指定了发生错误时运行哪个函数。与 Rstudio 调试最相似的函数是 **browser()**: 它将在发生错误的环境中, 启动一个交互式控制台。使用 **options(error = browser)** 将其打开, 重新运行前面的命令, 然后使用 **options(error = NULL)** 回到默认的错误行为。你还可以使用 **browseOnce()** 函数让这个过程自动化, **browseOnce()** 函数的定义如下:

```
browseOnce <- function() {  
  old <- getOption("error")  
  function() {  
    options(error = old)  
    browser()  
  }  
}  
options(error = browseOnce())  
f <- function() stop("!")  
# 进入浏览器  
f()  
# 正常运行  
f()
```

(你将在第 10 章学习更多关于返回函数的函数的知识)。另外, 还有两个你可以使用 **error** 选项的有用函数:

recover 是 **browser** 的小改进, 因为它允许你进入调用栈中任何调用所处的环境。这是非常有用的, 因为错误的根源是经常一些调用。

dump.frames 相当于非交互式代码的 **recover**。它在当前工作目录中创建了一个 **last.dump.rda** 文件。然后, 在后面的交互式 R 的会话中, 你加载该文件, 并使用 **debugger()**, 进入一个与 **recover()** 具有相同接口的交互式调试器。

这可以实现**批处理**代码的交互式调试。

```
# 在批处理 R 进程中 ----
dump_and_quit <- function() {
# 保存调试信息到文件 last.dump.rda 中
dump.frames(to.file = TRUE)
# 带着错误状态退出 R
q(status = 1)
}
options(error = dump_and_quit)
# 在稍后的交互式会话中 ----
load("last.dump.rda")
debugger()
```

使用 `options(error = NULL)` 把错误行为(error behavior)重置为默认状态。然后，错误(error)将打印信息，并且退出函数的执行。

9.2.3 浏览任意代码

碰到错误进入交互式控制台以后，你可以使用 Rstudio 断点或 `browser()` 进入任意的代码位置。你可以在 Rstudio 的行号左侧点击或者按下 **Shift + F9**，来设置断点。同样地，在你想要暂停的地方添加 `browser()`。**断点**的行为类似于 `browser()`，但它们更容易设置(一次点击，而不是按 9 个键)，并且也不用担心在你的源代码中意外地添加了 `browser()` 语句。

断点有两个小缺点：

1. 在少数情况下，断点是不能工作的：阅读断点故障排除 (<http://www.rstudio.com/ide/docs/debugging/breakpoint-troubleshooting>) 来获取更多的细节信息。

2. RStudio 目前不支持条件断点，但是你却总是可以把 `browser()` 放在一个 `if` 语句中。（译者注：即把 `browser()` 放在 `if` 语句内部，可以实现有条件地暂停。）

除了自己添加 `browser()` 以外，还有另外两个函数会将它添加到代码：

`debug()` 会在指定函数的第一行插入一个 `browser()` 语句。

`undebug()` 则会删除它。

或者，你可以使用 `debugonce()`，它只在函数下一次运行时执行一次 `browser()`。

`utils::setBreakpoint()` 也有类似的作用，但它不使用函数的名字作为参数，而是使用文件名和行号，然后为你找到合适的函数。

这两个函数都是 `trace()` 的特例，`trace()` 可以在现有函数的任意位置插入任意代码。当你调试没有使用 `source()` 加载过的代码时，`trace()` 偶尔会有用。从一个函数中删除 `trace()`，可以使用 `untrace()`。在每一个函数中，你只能执行一个 `trace()`，但是一个 `trace()` 可以调用多个函数。

9.2.4 调用栈： `traceback()`、`where()` 和 `recover()`

不幸的是，由 `traceback()`、`browser() + where` 打印出来的调用栈，与 `recover()` 打印出来的调用栈是不一致的。下面的表格说明了由这三种工具显示出来的一个简单的嵌套调用的调用栈。

<code>traceback()</code>	<code>where</code>	<code>recover()</code>
4: <code>stop("Error")</code>	<code>where 1: stop("Error")</code>	1: <code>f()</code>
3: <code>h(x)</code>	<code>where 2: h(x)</code>	2: <code>g(x)</code>
2: <code>g(x)</code>	<code>where 3: g(x) </code>	3: <code>h(x)</code>
1: <code>f()</code>	<code>where 4: f()</code>	

注意：编号在 `traceback()` 和 `where` 之间是不同的，而 `recover()` 是以相反的顺序来显示调用的，并且省略了对 `stop()` 的调用。RStudio 与 `traceback()` 以相同的顺序显示调用，但是省略了编号。

9.2.5 其它类型的错误

除了抛出错误或返回不正确的结果以外，函数的失败还有其它的方式。

函数可能产生未预料的警告。跟踪警告的最简单的方法是使用 `options(warn = 2)` 将它们转换成错误，并使用常规的调试工具。当你这么做的时候，你将会在调用栈中看到一些额外的调用，如 `doWithOneRestart()`、`withOneRestart()`、`withRestarts()` 和 `.signalSimpleWarning()`。可以忽略掉这些：它们是用来把警告变成错误的内部函数。

函数可能生成未预料的消息。没有内置的工具来帮助解决这个问题，但是我们可以创建一个：

```
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())
```

正如警告一样，你需要忽略一些 **traceback** 的调用(如前面的两条和后面的七条)。

函数可能永远不会返回。这种情况是特别难以自动调试的，但是有时，终止函数并查看调用栈，可能会得到一些信息。否则，就使用上面描述的基本的调试策略。

最糟糕的情况是，你的代码可能会导致 R 完全崩溃而直接退出，使你没有办法进入交互式的代码调试状态。这表明在底层的 C 语言代码中存在错误。这是很难调试的。有时，一些交互式调试工具，比如 **gdb**，可能会有用，但是描述如何使用它超出了本书的范围。

如果崩溃是由基本 R 代码导致的，那么请给 **R-help** 发送一个可重现的例子。如果崩溃发生在一个包中，那么可以联系包的维护人员。如果这是由你自己的 C 或

C++代码导致的，那么你需要使用大量的 `print()` 语句来定位错误所在的位置，然后你需要使用更多的 `print()` 语句来找出哪些数据结构没有得出你期望的属性。

9.3 条件处理

未预料的错误需要交互式调试来找出发生了什么错误。但是，有些错误是可以预料的，你要自动处理它们。在 R 语言中，当你为不同的数据集拟合许多模型的时候，可预料的错误出现得最频繁，比如自助法重复（bootstrap replicates）。有时，模型可能拟合失败，并且抛出一个错误，但是你并不想停止一切。相反，你想拟合尽可能多的模型，然后在拟合完成之后才执行诊断。在 R 语言中，有三种程序化地处理条件的工具(也包括错误)：

即使程序发生了错误，`try()`使你能够继续执行。

`tryCatch()`允许你指定处理(handler)函数，控制着当某种条件发生时，应该做什么。

`withCallingHandlers()`是 `tryCatch()`的一种变体，它在不同的上下文中运行处理函数。我们很少会需要它，但是留意一下它也是很有用的。下面更详细地描述了这些工具。

9.3.1 使用 `try()`忽略错误

即使程序发生了错误，`try()`可以允许继续执行。例如，通常，如果你运行一个函数，它抛出了一个错误，那么它会立即终止，并且不会返回值：

```
f1 <- function(x) {  
  log(x)  
  10  
}
```



```
f1("x")  
#> Error: non-numeric argument to mathematical function
```

然而，如果你把产生错误的语句包围在 `try()` 中，那么错误消息将打印出来，但是仍然会继续执行：

```
f2 <- function(x) {  
  try(log(x))  
  10  
}  
f2("a")  
#> Error in log(x) : non-numeric argument to mathematical function  
#> [1] 10
```

你可以使用 `try(..., silent = TRUE)` 屏蔽消息。要把更大的代码块传入 `try()` 中，需要把代码包围在 `{}` 中：

```
try({  
  a <- 1  
  b <- "x"  
  a + b  
})
```

你也可以捕获 `try()` 函数的输出。如果成功，它将被计算的代码块(就像一个函数)的最后结果。如果不成功，它将是一个 "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"
```

当你将一个函数应用于列表中的多个元素时，`try()`特别有用：

```
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
```

没有内置函数来测试 `try-error` 类，所以我们要定义一个。然后，你可以使用 `sapply()` (第 11 章中讨论) 很容易地找到错误的位置，并且提取成功信息或者看看导致失败的输入。

```
is.error <- function(x) inherits(x, "try-error")
succeeded <- !sapply(results, is.error)
# 看看成功的结果
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
# 看看导致失败的输入
str(elements[!succeeded])
#> List of 1
#> $ : chr [1:26] "a" "b" "c" "d" ...
```

另一个有用的 `try()` 用法是如果一个表达式失败，那么就使用默认值。在 `try` 块之外，简单地指定一个默认值，然后再运行有风险的代码：

```
default <- NULL
try(default <- read.csv("possibly-bad-input.csv"), silent = TRUE)
```

另外，还有 `plyr::failwith()`，它使得这种策略更容易实现。参见 12.2 节获取详情。

9.3.2 使用 `tryCatch()` 处理条件

`tryCatch()` 是一种处理条件的通用工具：除了错误以外，你还可以对警告、消息和中断采取不同的行动。在前面，你已经看到过错误(由 `stop()` 产生)、警告(`warning()`)和消息(`message()`)，但是中断(`interrupt`)是新的概念。中断不能由程序员直接产生，但是，当用户通过按 `Ctrl + Break`、`Escape` 或 `Ctrl + C`(依赖于平台)，试图终止执行的时候，中断就会产生。

使用 `tryCatch()`，你可以把条件映射到处理函数，它们是一些在条件发生时调用的命名函数，这些函数被作为输入传入 `tryCatch()`。如果发生了一个条件，那么 `tryCatch()` 将调用第一个名字与条件类之一匹配的处理程序。仅有的有用的内置名称是：错误(`error`)、警告(`warning`)、信息(`message`)、中断(`interrupt`)和万能(`catch-all`)条件。处理函数可以任何事情，但是通常它要么返回一个值或者创建一个内容更丰富的错误消息。例如，下面的 `show_condition()` 函数设置了处理函数，以返回发生的条件类型：

```
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"
# 如果没有捕获到条件，那么 tryCatch 会返回输入的值
show_condition(10)
#> [1] 10
```

你可以使用 `tryCatch()` 来实现 `try()`。一种简单的实现如下所示。为了使错误消息看起来更像没有使用 `tryCatch()` 的情况，`base::try()` 要更加复杂。注意

`conditionMessage()` 用于提取与原始错误相关的信息。

```
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)
```

与条件发生时返回默认值一样，处理函数可以用来生成内容更加丰富的错误消息。例如，以下函数包装了 `read.csv()` 函数，它通过修改存储在错误条件对象 (error condition object) 中的消息，会将文件名添加到任何错误中：

```
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)
```

当用户试图中止运行代码的时候,如果你想采取特殊行动,那么捕获中断可能是有用的。但是要小心,这样很容易就会创建无限循环(除非你强制停止(kill)R)!

```
# 不允许用户中断代码  
i <- 1  
while(i < 3) {  
  tryCatch({  
    Sys.sleep(0.5)  
    message("Try to escape")  
  }, interrupt = function(x) {  
    message("Try again!")  
    i <- i + 1  
  })  
}
```

`tryCatch()`还有另一个参数: `finally`。它指定了一块要执行的代码(而不是一个函数),不管前面的表达式是成功还是失败,这段代码永远都会被执行。这可以用于清理工作如删除文件、关闭连接等等)。这在功能上相当于使用了 `on.exit()`,但是它可以包装更小的代码块,而不是整个函数。

9.3.3 withCallingHandlers()

`withCallingHandlers()`是 `tryCatch()`的一种替代。这两个函数之间有两种主要的区别:

`tryCatch()`的处理函数的返回值是由 `tryCatch()`返回的，而 `withCallingHandlers()`处理函数的返回值将被忽略：

```
f <- function() stop("!")
tryCatch(f(), error = function(e) 1)
#> [1] 1
withCallingHandlers(f(), error = function(e) 1)
#> Error: !
```

`withCallingHandlers()`中的处理函数，是在产生了条件的调用的上下文中，被调用的；而 `tryCatch()`的处理函数是在 `tryCatch()`的上下文中被调用的。这里，我们使用了 `sys.calls()`函数来显示这些情况，该函数相当于是运行时(run-time)版本的 `traceback()`函数——它列出了引出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()
# [[5]] stop("!")
# [[6]] .handleSimpleError(
```

```
# function (e) print(sys.calls()), "!", quote(h()))  
# [[7]] h(simpleError(msg, call))
```

这也会影响调用 `on.exit()` 的顺序。这些细微的差别很少会有用，除非你想精确地捕获错误，并将其传递给另一个函数。大多数情况下，你不应该使用 `withCallingHandlers()`。

9.3.4 自定义信号类

本节说明自定义信号类(Custom signal classes)。在 R 语言中进行错误处理的挑战之一是，大多数函数只是使用一个字符串来调用 `stop()`。这意味着如果你想找出某个特定的错误是否发生了，那么你必须看看错误消息的文本。这是很容易出错的，不仅因为错误的文本随着时间的推移可能会发生改变，还因为许多错误消息是经过翻译的，所以消息可能跟你所期望的是完全不同的。

R 语言有一种鲜为人知并且很少有人使用的特性，以解决这个问题。`condition` 是 S3 类，因此，如果你想区分不同类型的错误，那么你可以定义自己的类。每个发生条件信号的函数，`stop()`、`warning()`和 `message()`，都可以传入一个字符串列表，或者一个自定义的 S3 条件对象。自定义条件对象并不是经常使用的，但是非常有用，因为它让用户可以用不同的方式来应对不同的错误。例如，“意料之中的”(`"expected"`)错误(如对某些输入数据集，模型未能收敛)，可以被默默地忽略掉，而未预料的错误(如没有可用的磁盘空间)则可以传播给用户。

R 语言没有为条件提供内置的构造函数，但是我们可以很容易地添加一个。条件必须包含消息和调用组件，还可能包含其它有用的组件。当创建一个新条件时，它应该总是继承自 `condition` 类，以及错误(error)、警告(warning)或消息(message)之一。

```
condition <- function(subclass, message, call = sys.call(-1), ...) {  
  structure(  
    message,
```

```
class = c(subclass, "condition"),  
list(message = message, call = call),  
...  
)  
}  
  
is.condition <- function(x) inherits(x, "condition")
```

你可以使用 `signalCondition()` 产生任意条件信号，但是除非你实例化自定义信号处理器 (custom signal handler) (使用了 `tryCatch()` 或 `withCallingHandlers()`)，否则什么都不会发生。相反，更合适的是使用 `stop()`、`warning()` 或 `message()` 来触发平常的处理。如果你的条件的类与函数不匹配，那么 R 语言并不会报错，但是在实际的代码中应该避免这种情况。

```
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，你可以使用 `tryCatch()` 为不同的错误类型采取不同的行动。在这个例子中，我们创建了一个方便的 `custom_stop()` 函数，它让我们使用任意类来产生错误条件信号。在一个真正的应用程序中，最好有单独的 S3 构造函数，这样你就可以写入文档，详细地描述错误类 (error class)。

```
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，当对 `tryCatch()` 使用多个处理函数和自定义类的时候，第一个可匹配的处理程序会被调用，而不是匹配最好的。由于这个原因，你需要确保把最具体的处理函数放在第一位：（译者注：与 C++、Java 和 C# 中的 `try...catch` 块类似，捕获具体的 `Exception` 子类的 `catch` 块应该放在前面，而捕获通用的 `Exception` 类的 `catch` 块则应该放在后面，以捕获其它未预料的异常。）

```
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

比较以下的 `message2error()` 函数的两种实现。在这种情况下，`withCallingHandlers()` 的主要优点是什么？（提示：仔细看看 `traceback`。）

```
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(Defensive programming)是一种艺术，即使发生了未预料的错误，它也可以使代码以一种良好定义的方式，进入失败状态。**防御性编程**的一个重要原则是**快速失败**(fail fast)：一旦发现了错误，就立即发出错误信号。这对函数的作者(也就是你!)来说，意味着更多的工作，但是对用户来说，它可以使调试变得更容易，因为他们能很早就能得到错误，而不是等到未预料的输入已经传入了几个函数之后。

在 R 语言中，通过三种方式实现"快速失败"的原则：

严格限制你接受的参数。例如，如果你的函数的输入参数不是向量化的，那么确保检查一下输入是不是标量。你可以使用 `stopifnot()`、`assertthat` 包 (<https://github.com/hadley/assertthat>)，或者简单的 `if` 语句和 `stop()` 来实现。

避免使用非标准计算的函数，例如 `subset`、`transform` 和 `with`。以交互的方式使用时，这些函数可以节省时间，这是因为它们会做出一些假设，以减少键盘输入；而当它们失败了的时候，通常不会提供信息丰富的错误消息。你可以在第 13 章了解更多关于非标准计算的知识。

避免根据输入的不同，而返回不同类型输出的函数。最大的两个违背这一条的函数是 `[` 和 `sapply()`。每当在函数内部对数据框进行取子集操作时，你应该总是设置 `drop = FALSE`，否则你会不小心地把单列数据框转化成向量。同样，永远不要在函数内部使用 `sapply()`：永远使用更严格的 `vapply()`，如果输入不正确，那么它将抛出一个错误；甚至对零长度的输入，它也会返回正确的输出类型。（译者注：这一点相当重要，一定要记住！）

交互式分析和编程之间有一定的矛盾。当你进行交互式工作的时候，你希望 R 能按照你的想法进行工作。如果 R 猜错了，那么你希望能马上发现问题，这样你就可以进行修复。当你进行编程的时候，你希望函数对哪怕是轻微的错误或遗漏都能产生错误。在编写函数时候，请在心里记住这一点。如果你编写的函数是用来使交互式数据分析更加便利的，那么可以随意猜测数据分析师的想法，并能自动地从小错误中进行恢复。如果你编写的函数是用于编程的，那么一定要严格。永远不要猜测调用者需要什么。

9.4.1 练习

下面的 `col_means()` 函数的目的是计算数据框中每个数值列的均值(列内均值，不是整个数据框)。

```
col_means <- function(df) {  
  numeric <- sapply(df, is.numeric)  
  numeric_cols <- df[, numeric]  
  data.frame(lapply(numeric_cols, mean))  
}
```

然而，这个函数对于异常的输入，并不是那么具有健壮性(robust)。看看下面的结果，确定哪些是不正确的，然后修改 `col_means()` 使它变得更健壮。(提示：在 `col_means()` 函数内部，有两个函数调用特别容易出现问题的)。

```
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)
```

以下函数“滞后”(lag)一个向量，它返回这样的一个向量：相对于原始的 `x`，它把 `x` 推后 `n` 个位置，前面以 `NA` 来填充。改进该函数，使得它具备以下功能：

- (1)如果 `n` 是一个向量，那么返回一个有用的错误消息；
- (2)当 `n` 是 0 或者比 `x` 长的时候，它能有合理的行为。

```
lag <- function(x, n = 1L) {
  xlen <- length(x)
  c(rep(NA, n), x[seq_len(xlen - n)])
}
```

9.5 小测验答案

1. 定位错误发生在哪里的最有用的工具是 `traceback()`。或者使用 Rstudio，它可以自动显示发生错误的地方。

2. `browser()` 允许在指定的行暂停执行代码，并让你进入交互式环境。在这种环境下，有五个有用的命令：
 - `n`，执行下一条命令；
 - `s`，进入下一个函数；
 - `f`，执行完当前的循环或函数；
 - `c`，继续正常执行代码；
 - `Q`，停止函数并返回到控制台。
3. 你可以使用 `try()` 或 `tryCatch()`。
4. 因为你可以使用 `tryCatch()` 捕获特定类型的错误，而不是通过比较错误字符串，而比较错误字符串是有风险的，尤其是当消息经过了翻译的时候。

第二部分 函数式编程

10 函数式编程

R 语言是一种**函数式编程**语言(FP)。这意味着它提供了许多工具用来创建和操纵函数。特别地，R 语言拥有所谓的**一等函数**(first class functions, https://en.wikipedia.org/wiki/First-class_function)。你可以对函数做任何事情，就跟向量一样：你可以将它们赋予给变量，将它们存储在列表里，把它们作为其它函数的参数进行传递，在函数内部创建它们，甚至把它们作为一个函数的返回结果。

本章首先展示一个激动人心的例子——通过代码来去除冗余和重复的数据，用来**清洗数据**和**汇总数据**。然后，你将了解**函数式编程**的三个**构建块**(building block)：**匿名函数**、**闭包**(在其它函数内部写的函数，并作为结果返回)和**函数列表**。我们会把这些技术联合在一起使用，展示如何为**数值积分**构建一套工具，从非常简单的**原语函数**开始。这是反复出现在**函数式编程**中的过程：从小函数开始，首先是易于理解的构建块，然后把它们组合成更复杂的结构，最后带着信心地去使用它们。

函数式编程的讨论将在后面的两章继续进行：

第 11 章探讨了以函数作为参数的函数，它们以向量作为输出，第 12 章探讨了将函数作为输入，并仍以函数作为输出的函数。

本章概要

10.1 节使用**函数式编程**解决一种常见的问题：在正式进行数据分析之前，进行**清洗数据**和**汇总数据**。

10.2 节展示了你可能没有了解过的**函数的另一面**：你可以使用没有名字的函数——**匿名函数**。

10.3 节介绍了**闭包**——由另一个函数创建的函数。**闭包**可以访问自己的参数，以及它的**父环境**中定义的变量。

10.4 节展示了如何将函数放在一个列表里，并解释你为什么需要关心这个。

10.5 节以一个案例研究来总结本章，该案例使用了**匿名函数**、**闭包**和**函数列表**来构建一个灵活的**数值积分**工具。

前提条件

你应该熟悉**词法作用域**的基本规则，参考 6.2 节。确保你已经使用 `install.packages("pryr")` 安装了 `pryr` 包。

10.1 使用函数式编程的动机

想象你加载了一个数据文件，如下所示，使用 `99` 表示缺失值。你想把所有的 `-99` 都替换成 `NA`。

```
# 生成样本数据集
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
```

当你第一次开始写 R 代码的时候，你可能已经通过复制粘贴解决了这个问题：

```
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
```

复制粘贴的一个问题是很容易犯错误。你能发现上面代码中的两处错误吗？这些错误是不一致的，因为我们没有所需的动作的一个权威性的描述(用 `NA` 替换 `99`)。重复的动作会让错误变得更多，也让代码变得很难改变。例如，如果要把缺失值的代码从 `-99` 更改到 `-9999`，那么你需要在多个地方都进行修改。

为了防止错误，以及创建更灵活的代码，请采用“不要重复自己”(“do not repeat yourself”)的原则。该原则是由于 Dave Thomas 和 Andy Hunt 写的《pragmatic programmers》(<http://pragprog.com/about>)而变得流行的，这个原则说到：“在一个体系中，每个知识都必须有一个明确的、权威的表达”。函数式编程工具是有价值的，因为它们提供了工具来减少重复。我们可以开始应用函数式编程的思想，通过编写一个函数来修正单个向量中的缺失值：

```
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)
```

这减少了可能的错误的范围，但并没有消除错误：你不再会偶然把-99 打成-98 了，但是你仍然可能写错变量的名称。下一步是通过结合两个函数，来删除这个可能的错误源。一个函数，`fix_missing()`，知道如何修正单个向量；另一个函数，`lapply()`，知道如何为数据框的每一列做些事情。`lapply()`有三个输入：`x`，一个列表；`f`，一个函数；以及...，传递给 `f()` 的其它参数。它对列表的每个元素都调用函数，并返回一个新列表。`lapply(x, f, ...)` 相当于下面的 `for` 循环：

```
out <- vector("list", length(x))
for (i in seq_along(x)) {
  out[[i]] <- f(x[[i]], ...)
}
```

真正的 `lapply()` 要复杂得多，因为由于效率原因，它是由 C 语言实现的，但是算法的本质是相同的。`lapply()` 被称为**泛函**，因为它接受一个函数作为参数。**泛函**，是**函数式编程**的一个重要组成部分。你将在第 11 章学习更多关于**泛函**的知识。

我们之所以可以在这个问题上应用 `lapply()`，是因为数据框是列表。我们只需要一个小技巧，以确保我们将得到的是一个数据框，而不是一个列表。我们将 `lapply()` 的结果赋给 `df[]`，而不是 `df`。R 的规则将确保我们得到一个数据框，而不是一个列表。（如果你觉得很惊讶，那么你可能需要阅读 3.3 节）。把这些合在一起，我们得到：

```
fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}
df[] <- lapply(df, fix_missing)
```

这段代码与复制粘贴相比，有五个优点：

1. 它更紧凑。
2. 如果缺失值的代码发生变化，只需要更新一个地方。
3. 它适用于任何数量的列。不会错过任何一列。
4. 不会让一列的操作与另一列不同。
5. 很容易把这种技术推广到列的一个子集：

```
df[1:5] <- lapply(df[1:5], fix_missing)
```

关键思想是函数的组合。使用两个简单的函数，一个用于对每一列做些事情，而另一个用于修正缺失值，并把它们结合起来修正每一列中的缺失值。编写独立的、容易理解的简单函数，然后再进行组合，是一种强大的技术。

如果不同的列使用了不同的缺失值代码该怎么办？你可能会进行复制粘贴：

```
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  
}
```

和之前一样，这样很容易产生错误。相反，我们可以使用**闭包**——返回函数的函数。**闭包**允许我们基于一个模板来创建函数：

```
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
```

额外的参数

在这种情况下，你可能会争辩我们应该只需要添加一个参数即可：

```
fix_missing <- function(x, na.value) {  
  x[x == na.value] <- NA  
  x  
}
```

在这里，这是一个合理的解决方案，但是它并不是在每一种情况下都适用的。我们将会在第 11.5 节看到使用**闭包**的方法。

现在考虑一个相关的问题。一旦你完成了**清洗数据**，你可能想要对每一个变量进行相同的**数值汇总**。你可以像这样编写代码：

```
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)
```

但是，最好识别并删除重复项。在继续阅读之前，请花一两分钟来考虑如何解决这个问题。

一种方法是编写一个汇总函数，然后对每一列进行调用：

```
summary <- function(x) {
  c(mean(x), median(x), sd(x), mad(x), IQR(x))
}
lapply(df, summary)
```

这是一个好的开始，但是仍有一些重复。我们把汇总函数做一些实际的改变，会更容易看到：

```
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))
}
```

所有的五个函数调用，都使用了相同的参数(`x` 和 `na.rm`)，重复了 5 次。像往常一样，**重复**会使代码变得脆弱：它很容易引入错误，并且难以适应不断变化的需求。要删除这个重复的来源，你可以利用另一个**函数式编程技巧**：把函数存储在列表里。

```
summary <- function(x) {  
  funs <- c(mean, median, sd, mad, IQR)  
  lapply(funs, function(f) f(x, na.rm = TRUE))  
}
```

这一章将更详细地讨论这些技术。但在开始学习它们之前，你需要学习最简单的**函数式编程工具——匿名函数**。

10.2 匿名函数

在 R 中，函数也是对象。它们不是**自动绑定**到一个名称的。与许多语言(比如 C, C++, Python 和 Ruby)不同，R 语言没有创建**命名函数**的特殊语法：当你创建一个函数时，使用常规的赋值运算符给它一个名字即可。如果你选择不为函数指定一个名称，那么你会得到一个**匿名函数**。

当没有必要给函数一个名字的时候，你可以使用**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)
```

像 R 语言中的所有函数一样，**匿名函数**也有**形式参数** `formals()`，**函数体** `body()`以及**父环境** `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>
```

你可以调用没有名字的**匿名函数**，但是代码有点难读，因为你必须通过两种不同的方法来使用括号：

第一，调用一个函数；

第二，让它清楚地知道你想调用**匿名函数**本身，而不是调用在**匿名函数**内部的(可能是无效的)函数：

```
# 这并没有调用匿名函数。
# (注意"3"不是一个合法的函数。)
function(x) 3()
#> function(x) 3()
# 加上合适的括号，函数可以调用了：
(function(x) 3)()
#> [1] 3
# 所以这个匿名函数语法的行为
(function(x) x + 3)(10)
#> [1] 13
# 与以下函数的行为是相同的
f <- function(x) x + 3
f(10)
#> [1] 13
```

你可以调用拥有**命名参数**的**匿名函数**，但是其实这样表示你的函数需要一个名字。**匿名函数**的最常见的用途之一是创建**闭包**——由其它函数创建的函数。下一节将描述**闭包**。

10.2.1 练习

1. 给定一个函数，比如 `mean`，`match.fun()`可以让你找到一个函数。给定一个函数，你能找到它的名字吗？为什么在 R 语言中这是没有意义的？
2. 使用 `lapply()`和一个匿名函数对 `mtcars` 数据集的所有列，求出变异系数(标准差除以平均值)。
3. 使用 `integrate()`和一个匿名函数求出下列函数曲线下的面积。使用 Wolfram Alpha(<http://www.wolframalpha.com/>)检查你的答案。

`y = x ^ 2 - x, x in [0, 10]`

`y = sin(x) + cos(x), x in [-π, π]`

`y = exp(x) / x, x in [10, 20]`

4. 一个好的经验法则是匿名函数应该写在一行上，并且不需要使用 `{}`。检查你的代码。你能在哪里使用匿名函数而不是命名函数？你应该在哪里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

匿名函数的一种应用，是创建不值得命名的小函数。另一个重要用途是创建闭包——函数创建的函数。因为闭包被封闭在父函数的环境之中，并且可以访问父函数的所有变量，所以取了这个名字。这很有用，因为它允许我们有两个层次的参数：控制操作的父层和进行工作的子层。下面的例子使用这个思路生成了一系列幂函数，在父函数(`power()`)中，创建了两个子函数(`square()`和 `cube()`)。

```
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
```

当你打印一个闭包时，你看不到什么有用的信息：

```
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改变。不同的是封闭环境，`environment(square)`。要看到环境的内容的一种方法是将它转换为一个列表：


```
as.list(environment(square))
#> $exponent
#> [1] 2
as.list(environment(cube))
#> $exponent
#> [1] 3
```

要看到发生的事情的另一种方法是使用 `pryr::unenclose()`。这个函数把在封闭环境中定义的变量，替换成它们的值：

```
library(pryr)
unenclose(square)
#> function (x)
#> {
#> x^2
#> }
unenclose(cube)
#> function (x)
#> {
#> x^3
#> }
```

闭包的父环境的是创建它的函数的执行环境，如这段代码所示：

```
power <- function(exponent) {
  print(environment())
  function(x) x ^ exponent
}
zero <- power(0)
#> <environment: 0x7fa58ad7e7f0>
```

```
environment(zero)
```

```
#> <environment: 0x7fa58ad7e7f0>
```

执行环境通常在函数返回一个值后就会消失。然而，函数会捕获它们的**封闭环境**。这意味着，当函数 **a** 返回了函数 **b**，函数 **b** 捕获并存储了函数 **a** 的**执行环境**，则该**执行环境**不会消失。(这对内存使用有着重要的影响，细节参见 18.2 节。)

在 R 语言中，几乎所有的函数都是**闭包**。所有的函数都会记录**它被创建时所处的环境**，通常，如果是你写的函数，那么是**全局环境**；如果是别人写的函数，那么是**包的环境**。唯一例外的是，直接调用 C 语言代码的**原语函数**，它们没有关联的环境。

闭包对创建函数工厂是有用的，也是在 R 中管理**可变状态**的一种方法。

10.3.1 函数工厂

函数工厂(function factory)是创建新函数的是一种要素。我们已经看到过两个函数工厂的例子：**missing_fixer()**和 **power()**。你可以使用一些参数调用它，这些参数描述了想得到的功能，然后它会为你返回一个函数。对于 **missing_fixer()**和 **power()**来说，使用**函数工厂**而不是使用带有多个参数的单个函数，并没有带来多少好处。下列的情况下，使用函数工厂是最有用的：

不同的层次(译者注：**父函数**和**子函数**两层)更加复杂，并带有多个参数以及复杂的函数体。

当函数生成时，一些工作只需要做一次。

函数工厂尤其适合**最大似然问题**，你会在 11.5 节看到更加有效的使用它们。

10.3.2 可变状态

在两个层次(译者注:父函数和子函数两层)拥有变量,可以让你在函数调用之间维护状态。这是有可能的,因为每当执行环境刷新的时候,封闭环境都是保持不变的。在不同层次管理变量的关键,是双箭头赋值操作符(`<-`)。普通的单箭头赋值操作符(`<`),总是在当前环境下进行赋值,与它不同的是,双箭头操作符将继续沿着父环境链进行查找,直到找到一个匹配的名字。(8.4 节有更多的细节来解释它是如何工作的。)静态(static)父环境和`<-`联合在一起,可以维护函数调用之间的状态。下面的示例显示了一个计数器,它记录了一个函数被调用的次数。每当 `new_counter` 运行时,它都会创建一个环境,并在这个环境中对计数器 `i` 进行初始化,然后创建一个新的函数。

```
new_counter <- function() {  
  i <- 0  
  function() {  
    i <- i + 1  
    i  
  }  
}
```

新函数是一个闭包,它的封闭环境是由 `new_counter()` 运行时创建的环境。通常,函数的执行环境是暂时的,但是闭包保持着访问创建它的环境的能力。在下面的示例中,当运行的时候,闭包 `counter_one()` 和 `counter_two()` 都有各自的封闭环境,因此它们的计数是不同的。

```
counter_one <- new_counter()  
counter_two <- new_counter()  
counter_one()  
#> [1] 1  
counter_one()
```

```
#> [1] 2
counter_two()
#> [1] 1
```

通过不修改它们的本地环境中的变量，计数器绕过了"全新的开始"的限制。由于更改是在不变的父(或封闭)环境中进行的，因此，在函数调用之间，这些更改都会被保存。

如果你不使用闭包会发生什么呢？如果你使用<-而不是<<-会发生什么呢？预测一下，如果你使用下面的变体函数来替换 new_counter 将会发生什么？然后，运行这些代码，并检查你的预测。

```
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种重要技术，因为它是 R 中生成可变状态(mutable state)的一种方法。可变状态通常是很难的，因为，每当看起来好像是在修改一个对象时，实际上是创建了一个副本，然后修改了副本。然而，如果你确实需要可变对象，并且代码不是很简单，那么通常最好使用引用类，如 7.4 节所述。闭包的力

量是与第 11 章和第 12 章中更高级的技术紧密结合的。你会在那两章看到更多的闭包。下一节讨论 R 中函数式编程的第三种技术：把函数存储在列表中。

10.3.3 练习

1. 为什么由其它函数创建的函数称为闭包？
2. 下面的统计函数是做什么的？你能取一个更好的名字吗？(现有的名字已经给出了一点点提示。)

```
bc <- function(lambda) {  
  if (lambda == 0) {  
    function(x) log(x)  
  } else {  
    function(x) (x ^ lambda - 1) / lambda  
  }  
}
```

3. `approxfun()` 函数是做什么的？它返回什么？
4. `ecdf()` 函数是做什么？它返回什么？
5. 创建一个由函数创建的函数，它能计算一个数值向量的第 *i* 个中心矩 (http://en.wikipedia.org/wiki/Central_moment)。你可以通过运行以下代码进行测试：

```
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，传入一个索引 `i` 作为一个参数，并返回一个带有一个参数 `x` 的函数，它使用 `i` 对 `x` 进行取子集操作。

```
lapply(mtcars, pick(5))  
# 应该与下面这个做相同的事情  
lapply(mtcars, function(x) x[[5]])
```

10.4 函数列表

在 R 语言中，函数可以存储在列表里。这使得它可以把相关的函数更容易地组织在一起，就像数据框把相关的向量组织在一起一样。我们将从一个简单的基准测试的例子开始。假设你正在比较几种计算算术平均值的算法的性能。你可以把每个方法(函数)存储在列表中：

```
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  
  }  
)
```

从列表中调用函数非常简单。首先进行提取，然后进行调用：

```
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
```

要调用每个函数(例如, 检查它们是不是都返回相同的结果), 可以使用 `lapply()`。由于没有内置函数来处理这种情况, 所以我们需要一个匿名函数或者一个新的命名函数。

```
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
```

要对每个函数进行计时，我们可以结合 `lapply()` 和 `system.time()`：

```
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
```

另一个使用**函数列表**的例子是对一个对象使用多种汇总方式。要做到这一点，我们可以把每个汇总函数存储在一个列表里，然后用 `lapply()` 运行它们：

```
x <- 1:10  
funcs <- list(  
  sum = sum,  
  mean = mean,  
  median = median  
)  
lapply(funcs, function(f) f(x))  
#> $sum  
#> [1] 55  
#>
```



```
#> $mean
#> [1] 5.5
#>
#> $median
#> [1] 5.5
```

如果我们希望**汇总函数**能自动删除**缺失值**该怎么办？一种方法是创建一个**匿名函数**的列表，这些**匿名函数**会使用合适的参数调用我们的**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
```

然而，这会导致大量的重复。除了函数名以外，每个函数几乎都是相同的。一个更好的方法是修改我们的 `lapply()` 调用，让它包括额外的参数：`lapply(funs, function(f) f(x, na.rm = TRUE))`。

10.4.1 把函数列表移到全局环境

有时，你可能要创建一个函数列表，来规避特殊语法。例如，假设你想通过把每个标签(tag)映射到一个 R 的函数，来创建 HTML 代码。下面的示例使用了一个函数工厂，为标签<p> (paragraph, 段)、(bold, 粗体)和<i>(italics, 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)
```

我已经把函数放在了列表中，因为我不想让它们一直都可用。现有的 R 函数与 HTML 标签函数之间产生冲突的风险是很高的。但是把它们放在列表里，使得代码更冗长了：

```
html$p("This is ", html$b("bold"), " text.")  
#> [1] "<p>This is <b>bold</b> text.</p>"
```

根据我们想要效果持续多久，你有三种选择来避免使用 `html$`：

对于很短暂的效果，你可以使用 `with()`：

```
with(html, p("This is ", b("bold"), " text."))  
#> [1] "<p>This is <b>bold</b> text.</p>"
```

对于较长的效果，你可以把函数 `attach()` 在搜索路径上，当工作完成后，再使用 `detach()`：

```
attach(html)
p("This is ", b("bold"), " text.")
#> [1] "<p>This is <b>bold</b> text.</p>"
detach(html)
```

最后，你可以使用 `list2env()` 把函数复制到全局环境。在你的工作完成后，你可以通过删除该函数进行撤销。

```
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())
```

我推荐第一种选项，使用 `with()`，因为它很清楚地表明何时在特殊语境中执行代码，以及是什么特殊语境。

10.4.2 练习

1. 实现一个汇总函数，它像 `base::summary()` 那样工作，但是使用函数列表。修改这个函数，使它返回闭包，让它变成函数工厂。
2. 下面的命令中，哪个相当于 `with(x,f(z))`？
 - (a) `x$f(x$z)`.
 - (b) `f(x$z)`.
 - (c) `x$f(z)`.
 - (d) `f(z)`.
 - (e) 看具体情况.

10.5 案例研究：数值积分

为了给本章一个总结，我将使用"一等函数"来开发一个简单的**数值积分**工具。工具开发过程中的每一步都是希望**减少重复**以及让方法更加通用。

数值积分背后的思想很简单：通过使用简单的组成部分来逼近曲线，并求出曲线下方的面积。两个最简单的方法是**中点规则**和**梯形规则**。**中点规则**使用**矩形**来逼近曲线。**梯形规则**使用**梯形**。每一种都需要一个**被积函数 f** ，以及一个**积分范围**，从 a 到 b 。对于这个示例，我将尝试对 $\sin(x)$ 函数进行积分， x 的取值范围是从 0 到 π 。这是一个很好的测试选择，因为它有一个简单的答案： 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
```

这些函数给出了很好的近似。使用**微积分**的基本思想使它们更加准确：我们将把积分范围划分成小块，然后使用一种简单的规则，把每一块合并起来。这就是所谓的**复合积分**。我将会使用两个新的函数来实现它：

```
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
```

你会发现在 `midpoint_composite()` 和 `trapezoid_composite()` 之间有很多的重复部分。除了 `for` 循环内部的语句不同以外，它们基本上是相同的。你可以从这些具体的函数中，抽象出一个更通用的复合积分函数：

```
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
```

这个函数接受两个函数作为参数：**被积函数**和**积分规则**。我们现在可以添加在更小的范围进行积分的更好的规则：

```
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
```

事实证明，中点、梯形、Simpson 和 Boole 规则都是更一般的 Newton-Cotes 规则的特例(http://en.wikipedia.org/wiki/Newton%E2%80%93Cotes_formulas)。(它

们具有多项式复杂度。) 我们可以使用这个通用结构编写一个函数, 它可以生成任何一般的 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
```

在数学上, 改善数值积分的下一步是使用更高级的技术, 比如高斯求积法。这些内容超出了这个案例研究的范围, 但是你可以使用类似的技术来实现它。

10.5.1 练习

1. 我们可以把函数存储在一个列表里, 而不是创建单个函数(如 `midpoint()`、`trapezoid()`、`simpson()` 等等)。如果我们这样做, 如何更改代码? 你能通过 Newton-Cotes 公式的系数列表, 来创建函数列表吗?
2. 对于积分规则来说, 越复杂的规则计算越慢, 但是需要的分段也更少。对于函数 $\sin(x)$, 其中 x 在 $[0, \pi]$ 范围之内, 确定所需的分段数量, 以便让每个规则都同样精确。使用图形来说明你的结果。对于不同的函数它们是如何改变的? $\sin(1/x^2)$ 会特别具有挑战性。

11 泛函

"为了使代码变得更加可靠，代码必须变得更加透明。特别是，我们必须带着高度地怀疑来查看**嵌套条件**和**循环**。复杂的控制流使程序员迷惑。混乱的代码常常隐藏着错误。" ——Bjarne Stroustrup

高阶函数(higher-order function)是这样的函数：它接受一个函数作为输入或者返回一个函数作为输出。我们已经看到了一种**高阶函数**——**闭包**，它返回另一个函数。**闭包**的互补函数是**泛函**(functional)——一种接受函数作为输入，并返回一个向量作为输出的函数。下面是一个简单的**泛函**：它为作为输入的函数提供了 1000 个**随机均匀分布**数值。

```
randomise <- function(f) f(runif(1e3))
randomise(mean)
#> [1] 0.4897
randomise(mean)
#> [1] 0.4858
randomise(sum)
#> [1] 497.5
```

也许你已经使用过**泛函**：三大最常用的**泛函**是 **lapply()**、**apply()**和 **tapply()**。所有这三个**泛函**都是以函数作为输入(除了别的之外)，并返回一个向量作为输出。**泛函**的常见用途是作为 **for** 循环的替代。R 语言中的 **for** 循环口碑比较差。它们有一个坏名声就是很慢(尽管这个名声只是部分属实，详情参见 18.4 节)。但 **for** 循环的真正缺点是它们不是很富有表现力。**for** 循环表达了它在某些东西上进行迭代，但并没有清楚地传达一个高层次的目标。（译者注：即 **for** 循环过于琐碎，不容易看出来循环的目的到底是什么）最好使用**泛函**来代替 **for** 循环。每一种**泛函**都是针对某个特定任务的，所以当你认识到**泛函**的时候，可以立即知道**它被使用的原因**。除了可以替代 **for** 循环以外，**泛函**还扮演着其它角色。它们可以用于**封装常**

用的**数据操作任务**，比如**分解-应用-联合(split-apply-combine)**、以“**泛函的方式**”进行思考以及使用**数学函数**进行工作等等。

泛函通过更好地交流意图，来减少代码中的错误。在基础 R 语言中实现的泛函，是经过良好测试的(没有错误)，也是效率很高的，因为它们已经被太多的人使用过了。很多泛函是用 C 语言编写的，并使用了特殊的技巧来提高性能。也就是说，虽然使用**泛函**并不会总是产生最快的代码，但是，它可以帮助你进行**清晰地沟通**以及**构建解决各种各样问题的工具**。一开始就专注于速度是错误的，速度只在确实是个问题的时候才需要关注。一旦你有了清晰、正确的代码，你可以使用第 17 章中使用的技术使它变得更快。

本章概要

11.1 节介绍了你的第一个泛函:**lapply()**。

11.2 节向你展示了 **lapply()** 的变种，它们产生不同的输出，采用不同的输入以及用不同的方式来分配计算。

11.3 节讨论了使用更复杂的数据结构的泛函，比如矩阵和数组。

11.4 节教你强大的 **Reduce()** 和 **Filter()** 函数，它们在处理列表方面是很有用的。

11.5 节讨论你可能从数学中已经熟悉的泛函，比如**求根**、**积分**以及**优化**。

11.6 节提供了一些重要的警告，它告诉你在什么情况下，不应该试图把循环转换成泛函。

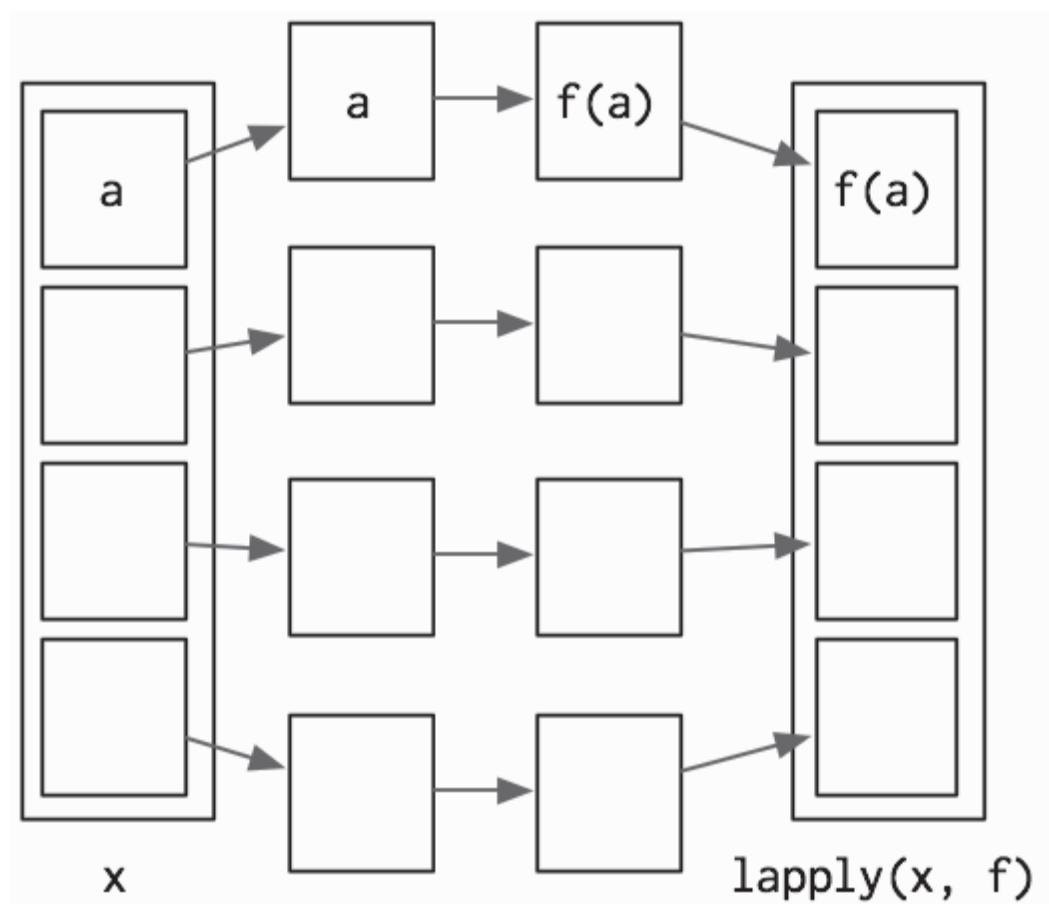
11.7 节是本章的总结，它展示了泛函如何利用简单的构建块，并且可以使用它来创建一组强大且一致的工具。

前提条件

你将经常会同时使用闭包与泛函。如果你需要复习一下，请复习 10.3 节。

11.1 我的第一个泛函:lapply()

最简单的泛函是 `lapply()`，你可能已经熟悉它了。`lapply()` 接受一个函数，把它应用于列表中的每个元素，最后以列表的形式返回结果。`lapply()` 是构建许多其它泛函的基础，所以了解它是如何工作的是很重要的。这里有一个图示：



由于性能关系，`lapply()` 是用 C 语言编写，但我们可以创建一个简单的、做同样事情的 R 语言实现版本：

```
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，你可以看到，`lapply()`是对一种通用的 `for` 循环模式的包装器：为输出创建一个容器，把 `f()` 应用到列表的每个组件，并把结果填充到容器中。所有其它的 `for` 循环泛函，都是这种模式的变种：它们只是使用不同的类型输入或输出。

通过消除大部分与循环相关的工作，`lapply()`更容易应用在列表之上。这使得你可以关注要应用的函数：

```
# 创建一些随机数据  
l <- replicate(20, runif(sample(1:10, 1)), simplify = FALSE)  
# 使用 for 循环  
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 6  
#> [20] 9  
# 使用 lapply  
unlist(lapply(l, length))  
#> [1] 8 8 1 4 6 6 9 8 9 10 2 6 4 3 2 7 3 10 6  
#> [20] 9
```

(我使用 `unlist()` 将输出从列表转换为向量，使得它更紧凑。我们稍后将看到其它使输出变成向量的方式)。

由于数据框也是列表，所以当你想要对数据框中的每一列做一些事情的时候，`lapply()`也很有用：

```
# 每一列是什么类？
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"
# 每一列都除以均值
mtcars[] <- lapply(mtcars, function(x) x / mean(x))
```

提供的 `x` 总是 `f` 的第一个参数。如果你想要变成不同的参数，那么你可以使用一个匿名函数。当计算固定的 `x` 的均值的时候，下面的例子中改变了所用的截取 (trimming) 量。

```
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
```

11.1.1 循环模式

请记住，对于向量，有三种基本的循环模式，这是很有用的：

1. 在元素上循环： `for (x in xs)`
2. 在数字索引上循环： `for (i in seq_along(xs))`
3. 在名字上循环： `for (nm in names(xs))`

对于 `for` 循环，第一种形式通常不是好选择，因为它导致了低效的保存输出的方式。使用这种形式，很自然地会通过扩展数据结构来保存输出，比如在这个例子中：

```
xs <- runif(1e3)
res <- c()
for (x in xs) {
  # 这很慢！
  res <- c(res, sqrt(x))
}
```

这是很慢的，因为每当你扩展向量的时候，R 不得不复制所有现有的元素。17.7 节更深入地讨论了这个问题。所以，最好是事先为输出创建好所需要的空间，然后进行填充。使用第二种形式来实现这种方式是最简单的：

```
res <- numeric(length(xs))
for (i in seq_along(xs)) {
  res[i] <- sqrt(xs[i])
}
```

就像使用 `for` 循环有三种基本方法一样，也有三种使用 `lapply()` 的基本方法：

```
lapply(xs, function(x) {})  
lapply(seq_along(xs), function(i) {})  
lapply(names(xs), function(nm) {})
```

通常你会使用第一种形式，因为 `lapply()` 可以负责为你保存输出。但是，如果你需要知道你正在使用的元素的位置或者名字，那么你应该使用第二种或者第三种形式。两者都给了你元素的位置 (`i`、`nm`) 和值 (`xs[[i]]`、`xs[[nm]]`)。如果你使用一种形式来解决问题的时候遇到困难，那么可能会发现换一种形式更容易。

11.1.2 练习

1. 为什么下面的两个 `lapply()` 调用是等价的？

```
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. 下面的函数对向量进行标准化，使得它落在 `[0,1]` 区间之内。怎样把它应用到数据框的每一列？怎样把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. 使用循环和 `lapply()` 对 `mtcars` 进行线性拟合，并且使用保存在下面列表中的公式(formula):

```
formulas <- list(
  mpg ~ disp,
  mpg ~ I(1 / disp),
  mpg ~ disp + wt,
  mpg ~ I(1 / disp) + wt
)
```

4. 使用 `for` 循环和 `lapply()`，对以下列表中的每一个 `mtcars` 抽取自助法重复样本，并且拟合模型 `mpg ~ disp`。不使用匿名函数可以做到吗？

```
bootstraps <- lapply(1:10, function(i) {
  rows <- sample(1:nrow(mtcars), rep = TRUE)
```

```
mtcars[rows,]  
})
```

5. 对前面两个练习中的每个模型，使用下面的函数抽取 R^2 (即 R-square) 值。

```
rsq <- function(mod) summary(mod)$r.squared
```

11.2 for 循环泛函：lapply() 的朋友们

使用泛函来代替 for 循环的关键是，识别出常见的循环模式，这些循环模式在现有的基本泛函中已经实现了。一旦你掌握了现有的这些泛函，下一步就是开始编写自己的泛函：如果你发现在很多地方都在重复相同的循环模式，那么你应该把它们提取出来变成一个函数。

后面的小节建立在 lapply() 之上，它们讨论了：

sapply() 和 vapply()：lapply() 的变种，它们产生向量、矩阵和数组作为输出，而不是列表。

Map() 和 mapply()，它们并行地 (parallel) 遍历多个输入的数据结构。

mclapply() 和 mcMap()，并行版本的 lapply() 和 Map()。

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，唯一的区别是它们简化了输出，产生的是原子向量。sapply() 会猜测输出类型，而 vapply() 使用一个额外的参数来指定输出类型。sapply() 是在交互式使用时很棒，因为它可以让用户少打字，但是如果你在函数中使用它，那么当你提供了错误类型的输入的时候，会得到奇怪的错误。

vapply() 更繁琐一些，但是它会给出含有更多信息的错误消息，它也从来不会默默地失败。它是适合于在其它函数中使用。

下面的例子说明了这些差异。

当给定一个数据框时，`sapply()`和`vapply()`返回相同的结果。

当给定一个空列表时，`sapply()`返回另一个空列表，而不是更正确的零长度逻辑向量。

```
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长度的结果，那么`sapply()`会默默地返回一个列表，而`vapply()`将抛出一个错误。`sapply()`对交互式使用来说是合适的，因为你通常会注意到是否出现了错误，但是在编写函数时，这是很危险的。

下面的例子说明了在数据框中提取各列的类名时，可能出现的问题：如果你错误地假设了类名只有一个值，并使用了`sapply()`，那么直到未来某些函数被传入了列表，而不是字符向量的时候，你才会发现这个问题。

```
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 `lapply()` 一种很简单的包装，它只是在最后一步把列表转换成向量。

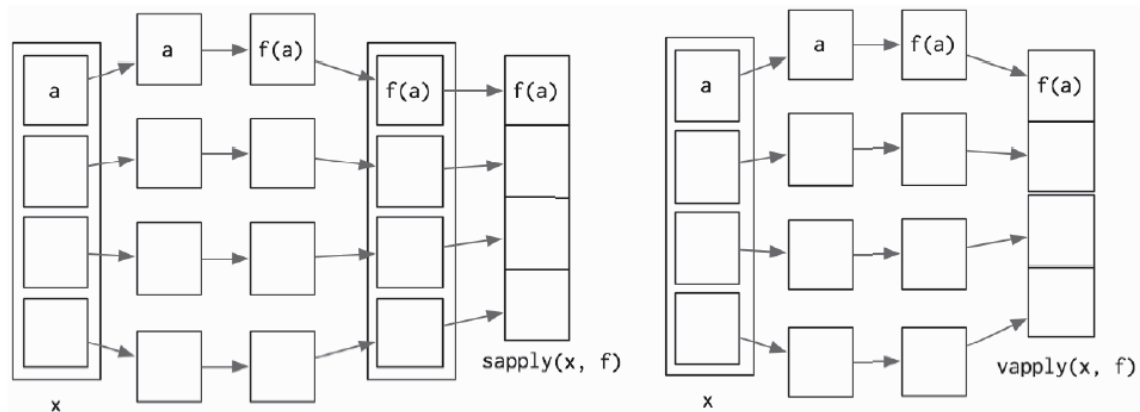
`vapply()`是 `lapply()` 的一种实现，它将结果赋给一个适当类型的向量(或矩阵)，而不是列表。下面的代码，显示了一个纯 R 语言实现的 `sapply()`和 `apply()`的本质(真正的函数有更好的错误处理以及保持名字等等)。

```
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。后面的小节会讨论 `Map()`，它则是输入不同。

11.2.2 多个输入：Map(和 mapply)

使用 `lapply()`，只有一个作为参数的函数是变化的；其它参数是固定的。这使得它对于一些问题不太适合。例如，当你有两个列表时，一个存放观察数据，另一个存放权重，你怎样计算加权平均数呢？

生成一些样本数据

```

xs <- replicate(5, runif(10), simplify = FALSE)
ws <- replicate(5, rpois(10, 5) + 1, simplify = FALSE)

```

使用 `lapply()`可以很容易计算非加权平均数：

```

unlist(lapply(xs, mean))
#> [1] 0.4696 0.4793 0.4474 0.5755 0.4490

```

但是，我们怎样把权重提供给 `weighted.mean()` 呢？使用 `lapply(x, means, w)` 是不可行的，因为 `lapply()` 的附加参数会传递给每一个调用。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
```

这样可以工作，但是显得有点笨拙。一种更好的替代方法是使用 `Map`，它是 `lapply()` 的一个变种，它的所有参数都可以变化。我们可以这样写：

```
unlist(Map(weighted.mean, xs, ws))  
#> [1] 0.4554 0.5370 0.4301 0.6391 0.4411
```

注意参数的顺序有点不同：`Map()` 的第一个参数是函数，而 `lapply()` 的第二个参数是函数。这相当于：

```
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()` 之间有一种自然的等价，因为你总是可以把 `Map()` 转换成 `lapply()`，使用 `lapply()` 对索引进行遍历。但是使用 `Map()` 可以更简洁、更清楚地表明你的想做的事情。

只要你有两个(或更多)列表(或数据框)需要并行处理，`Map` 就是有用的。例如，标准化列的另一种方式是，首先计算均值，然后除以这一列。我们可以使用 `lapply()` 这样做，但是如果我们分两步来做，我们可以更容易地检查每一步的结果，如果第一步是非常复杂的话，那么这是很重要的。

```
mtmeans <- lapply(mtcars, mean)
mtmeans[] <- Map(`/`, mtcars, mtmeans)
# 在这种情况下，相当于
mtcars[] <- lapply(mtcars, function(x) x / mean(x))
```

如果某些参数应该固定不变，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，我们将会看到一种更紧凑的方式来表达同样的想法。

mapply

相比 `Map()`，你可能更熟悉 `mapply()`。我更喜欢 `Map()`，因为：

它相当于 `mapply()` 设置了 `simplify = FALSE`，而这几乎是总是你想要的。

`mapply` 有 `MoreArgs` 参数，需要一个额外参数列表，这些参数将提供给每个调用，而不是使用匿名函数来提供固定的输入。这打破了 R 通常的延迟计算语义，并与其它函数不一致。

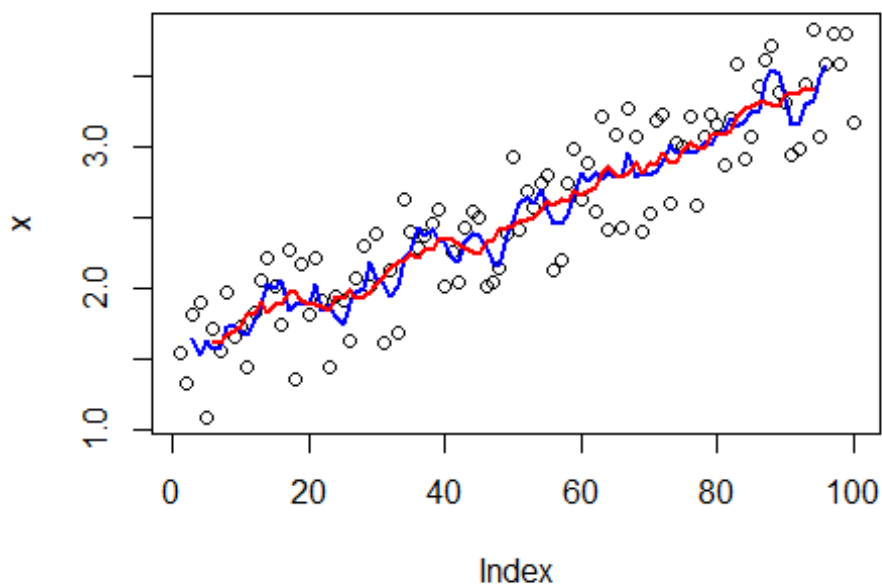
总之，使用 `mapply()` 增加了复杂性，而只获得了小小的收益。

11.2.3 滚动计算

本节提供了滚动计算(Rolling computations)的示例。如果你需要的 `for` 循环的替代者在基础 R 语言中不存在该怎么办？你通常可以通过识别通用的循环结构，来创建你自己的替代者，并实现自己的封装。例如，你可能对这个感兴趣：使用滚动(或移动)均值函数(rolling (or running) mean function)来平滑(smooth)你的数据。

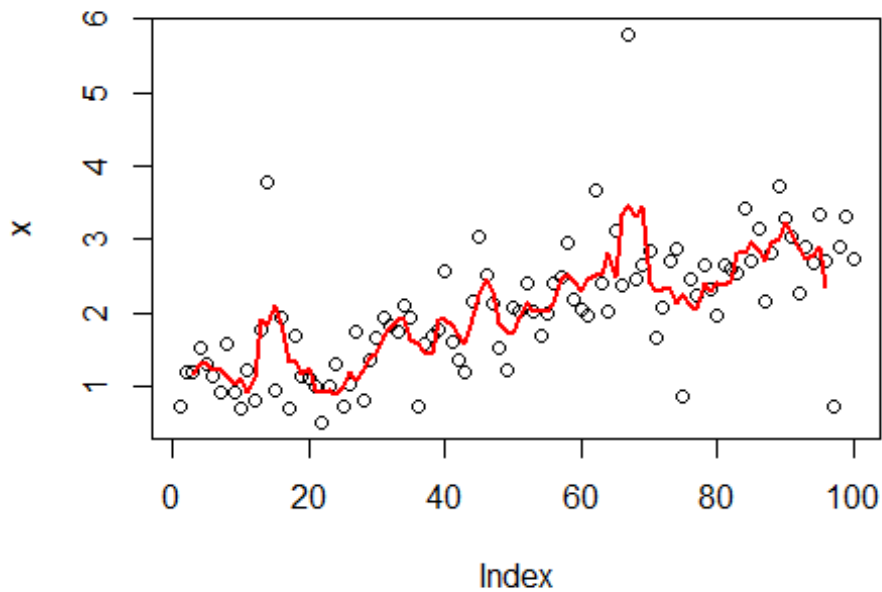
```
rollmean <- function(x, n) {
  out <- rep(NA, length(x))
  offset <- trunc(n / 2)
  for (i in (offset + 1):(length(x) - n + offset - 1)) {
```

```
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(noise)是更加多变(variable)的(即，它有更长的尾部(longer tail))，那么你可能担心你的滚动均值对离群值(outliers)太敏感。因此，你可能想要计算滚动中位数(rolling median)。

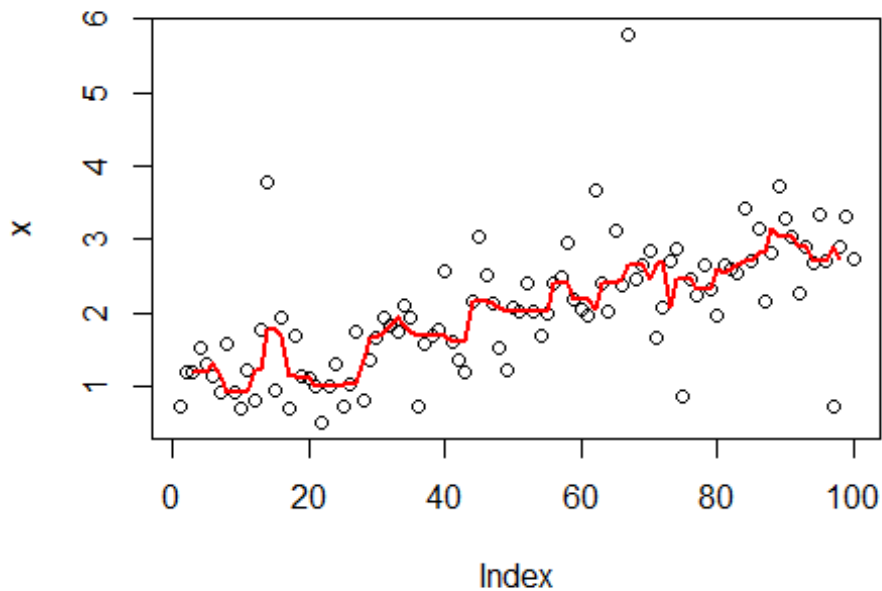
```
x <- seq(1, 3, length = 1e2) + rt(1e2, df = 2) / 3
plot(x)
lines(rollmean(x, 5), col = "red", lwd = 2)
```



把 `rollmean()` 变成 `rollmedian()`，你需要做的就是在循环中把 `mean` 替换成 `median`。但是，我们可以把计算滚动汇总的思想提取出来，并写成函数，而不是通过复制和粘贴来创建一个新的函数：

```
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)
```



你可能注意到，内部循环看起来非常类似于 `vapply()` 循环，所以我们可以把函数重写为：

```
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)
}
```

这与 `zoo::rollapply()` 中的实现同样有效，只不过 `zoo::rollapply()` 提供了更多的功能和错误检查。

11.2.4 并行化

实现 `lapply()` 的一个有趣的事是，因为每次迭代相对于其它所有的迭代都是独立的，所以它们之间的计算顺序并不重要。例如，下面的 `lapply3()` 打乱了计算顺序，但是结果总是相同的：

```
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
```

这导致了一个非常重要的结果：因为我们可以用任意顺序来计算元素，所以可以很容易地把任务分派到不同的核（译者注：多核 CPU 的核）上，进行并行计算。这就是 `parallel::mclapply()`（以及 `parallel::mcMap()`）要做的事情。（在 Windows 中，这些函数不可用，但是你可以使用相似的 `parLapply()` 函数，只需要多做一点点工作。查看 17.10 节获取更多细节。）

```
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
```

在这种情况下，`mclapply()`的性能实际上是低于 `lapply()` 的。这是因为单次计算的开销很低，反而把计算发送到不同的核，并收集计算结果，需要额外的工作。

如果我们采取一个更现实的例子，例如产生线性模型的自助法重复样本，那么优势是显而易见的：

```
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
```

虽然增加核的数量并不总是会导致性能发生线性地改进，但是把 `lapply()` 或 `Map()` 转换到它的并行化形式，是可以显著提高计算性能的。

11.2.5 练习

1. 使用 `vapply()`：
 - a) 在数值数据框中，计算每列的标准差。
 - b) 在混合类型的数据框中，计算每个数值列的标准差。(提示：需要使用两次 `vapply()`。)

2. 为什么使用 `sapply()` 对数据框中的每个元素应用 `class()` 得到类，是危险的？
3. 下面的代码模拟了对非标准数据进行 **t 检验**。使用 `sapply()` 和匿名函数来抽取每次试验的 **p 值**(p-value)。

```
trials <- replicate(  
  100,  
  t.test(rpois(10, 10), rpois(7, 10)),  
  simplify = FALSE  
)
```

附加挑战：通过直接使用 `[[` 来去掉匿名函数。

4. `replicate()` 是做什么的？它消除了哪种 `for` 循环？为什么它的参数与 `lapply()` 族函数不同？
5. 实现另一种版本的 `lapply()`，它使用每个组件的名字和价值来提供 `FUN`。
6. 实现 `Map()` 和 `vapply()` 的组合，来创建一个 `lapply()` 的变种，它对所有的输入进行并行迭代，并把输出保存在向量（或矩阵）中。这个函数需要什么参数？
7. 实现 `mcsapply()`，一个多核版的 `sapply()`。你能实现 `mcvapply()` 吗？它是并行版的 `vapply()`。说明为什么能或者不能？

11.3 操作矩阵和数据框

在一般的数据操作任务中，泛函也可以用来消除循环。在本节中，我们将对可用的选项作一个简短的概述，提示它们如何能帮助你，并且帮你指出正确的方向，以学习更多的知识。我们将讨论三种数据结构的泛函：

`apply()`、`sweep()` 和 `outer()` 可用于矩阵。

`tapply()` 对一个向量进行分组汇总，而分组是由另一个向量定义的。

`plyr` 包，它推广了 `tapply()`，使它更容易地处理作为输入的数据框、列表或数组，以及使用数据框、列表或数组作为输出。

11.3.1 矩阵和数组操作

到目前为止，我们看到的所有泛函，都是处理一维输入结构的。在本节中的三个泛函，提供了有用的工具来处理更高维的数据结构。`apply()` 是 `sapply()` 的一个变种，它可以处理矩阵和数组。你可以把它想象成这样的操作：它通过把矩阵或数组的每行或者每列分解成单个数值，对矩阵或数组进行汇总计算。它有四个参数：

X，需要进行汇总计算的矩阵或数组。

MARGIN，一个整数向量，它给出需要进行汇总计算的维度，`1 = rows`、`2 = columns` 等等。

FUN，汇总计算的函数。

...，传给 **FUN** 的其它参数。

一个典型的 `apply()` 示例看起来像这样：

```
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()` 时需要注意一些问题。它没有 `simplify` 参数，所以你不能完全确定你会得到什么类型的输出。这意味着，在函数内部使用 `apply()` 是不安全的，除非你

仔细地检查了输入。在这个意义上，`apply()`也不是幂等的(idempotent)，如果汇总计算函数是 `identity` 运算符，那么输出并不总是与输入相同：

```
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
```

(你可以使用 `aperm()` 或者 `plyr::aapply()` 把高维数组还原到正确的顺序，这是幂等的。)

`sweep()` 允许你"扫描"统计汇总的值。它通常是与 `apply()` 联合使用的，以对数组进行标准化。下面的例子对矩阵的行进行标准化(scale)，这样使得所有的值都介于 0 和 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), `/`)
```

最后的矩阵函数是 `outer()`。它有点不同，它需要多个向量输入，并创建一个矩阵或数组作为输出，对于每一种输入组合，都应用输入函数：（译者注：对作为输入的两个向量中的所有元素，进行两两配对，每一个配对作为参数输入到函数中。）

```
# 创建乘法表
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
```

一些学习 `apply()` 和它的朋友们的好地方是：

Peter Werner 写的《Using apply, sapply, lapply in R》

(<http://petewerner.blogspot.com/2012/12/using-apply-sapply-lapply-in-r.html>)。

Slawa Rokicki 写的《The infamous apply function》

(<http://rforpublichealth.blogspot.no/2012/09/the-infamous-apply-function.html>)。

axiomOfChoice 写的《The R apply function - a tutorial with examples》

(<http://forgetfulfunctor.blogspot.com/2011/07/r-apply-functiontutorial-with-examples.html>)。

stackoverflow 问题《R Grouping functions: sapply vs. lapply vs. apply vs. tapply vs. by vs. aggregate》(<http://stackoverflow.com/questions/3505701>)。

11.3.2 分组应用

你可以把 `tapply()` 想成是 `apply()` 一种推广，它允许输入参差不齐(ragged)的数组，这种数组每行的列数可以不同。当你尝试对数据集进行汇总计算的时候，这通常是需要的。例如，想象你在医学试验中搜集了脉搏数据，并且你想比较两个组的数据：

```
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
```

tapply() 从一组输入中创建参差不齐的数据结构，然后对那个结构中的单个元素应用函数。第一个任务是由 **split()** 函数完成的。它有两个输入，并返回一个列表，列表中是一些分组数据，分组是根据第二个向量中的元素或者种类，来对第一个向量中的元素进行分组：

```
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
```

所以 **tapply()** 仅仅是 **split()** 和 **sapply()** 的联合：

```
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
```

能够把 **tapply()** 重写成 **split()** 和 **sapply()** 的组合，表明我们发现了一些有用的构建块。

11.3.3 plyr 包

使用基本泛函的一个挑战是，它们是随着时间的推移逐渐成长的，是由多个作者编写的。这意味着，它们不是非常一致的：

使用 `tapply()` 和 `sapply()`，`simplify` 参数称为 `simplify`。

使用 `mapply()`，它叫做 `SIMPLIFY`。

使用 `apply()`，这个参数是没有的。

`vapply()` 是 `sapply()` 一个变种，它允许你描述输出应该是什么，但是对于 `tapply()`、`apply()` 或 `Map()` 却没有相应的变种。

大多数的基本泛函的第一个参数是一个向量，但是 `Map()` 的第一个参数是函数。

这使得学习这些操作符具有挑战性，因为你必须记住所有的变化。此外，如果你考虑所有可能的输入和输出的组合类型，那么在基础 R 语言中仅覆盖了部分情况：

	list	data frame	array
list	<code>lapply()</code>		<code>sapply()</code>
data frame	<code>by()</code>		
array			<code>apply()</code>

这是创建 `plyr` 包的主要动机之一。它提供了一致的命名函数，这些函数使用一致的命名参数，并且涵盖所有输入和输出数据结构的组合：

	list	data frame	array
list	lapply()	ldply()	laply()
data frame	dlply()	ddply()	daply()
array	alply()	adply()	aaply()

这些函数将分解输入，对每一块应用一个函数，然后将结果合并起来。总的来说，这种处理过程被称为“分解-应用-合并”(Split-Apply-Combine)。你可以在

《The Split-Apply-Combine Strategy for Data Analysis》

(<http://www.jstatsoft.org/v40/i01/>)中阅读到更多关于这种处理过程和 `plyr` 包的内容，它是发表在《Journal of Statistical Software》上可以公开访问的文章。

11.3.4 练习

1. `apply()`是如何组织输出的？阅读文档并进行一些实验。
2. 目前，没有等价于 `split()+ vapply()` 的泛函。应该需要这个泛函吗？什么时候它会有用？实现一个这样的泛函。
3. 实现一个纯 R 版本的 `split()`。(提示：使用 `unique()`和取子集操作)。你可以不用 `for` 循环来实现吗？
4. 缺失的其它的输入和输出类型是什么？在从 `plyr` 的文档中寻找答案之前，自己先想一想。(<http://www.jstatsoft.org/v40/i01/>)。

11.4 操作列表

思考泛函的另一种方式，是作为一组通用的工具，改变列表、对列表取子集以及消去列表。每一种函数式编程语言都有三种工具来做这些事情：`Map()`、`Reduce()` 和 `Filter()`。我们已经看过了 `Map()`，以下部分将描述 `Reduce()`——强大的扩展双参数函数(二元函数)的工具，以及 `Filter()`——作用于谓词函数的重要泛函，谓词函数是返回单个的 `TRUE` 或 `FALSE` 的函数。

11.4.1 Reduce()

`Reduce()` 可以削减向量 `x` 成为单个值，它通过递归地调用函数 `f` 做到这一点，在每两个参数之间调用一次 `f`。它使用 `f` 来合并前两个元素，得到一个结果，然后合并这个结果与第三个元素，...，以此类推。调用 `Reduce(f, 1:3)` 相当于调用 `f(f(1, 2), 3)`。`Reduce` 也被称为 `fold`，因为它将列表中的相邻元素合并在了一起。下面的两个例子说明了 `Reduce` 与中缀函数和前缀函数联合使用时，它做了什么：

```
Reduce(+, 1:3) # -> [(1 + 2) + 3]
Reduce(sum, 1:3) # -> sum(sum(1, 2), 3)
```

`Reduce()` 的本质可以被描述成一个简单的 `for` 循环：

```
Reduce2 <- function(f, x) {
  out <- x[[1]]
  for(i in seq(2, length(x))) {
    out <- f(out, x[[i]])
  }

  out
}
```

真正的 `Reduce()` 更加复杂，因为它还包含一些其它参数：

right, 控制**值**是从左边还是从右边进行合并。

init, 一个可选的**初始值**。

accumulate, 是否输出**中间结果**的选项。

Reduce()是一种优雅的扩展函数的方式, 它可以把作用于两个参数的函数, 转换成可以处理任意数量的输入的函数。这是用于实现许多类型的**递归操作**的方法, 比如**合并**和**求交集**。(我们将会在最后的案例研究看到另一种使用方法。)想象一下, 你有一个列表, 列表的元素都是数值向量, 你想找到所有列表元素中都出现的值:

```
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 ...
```

你可以依次对每个元素求交集:

```
intersect(intersect(intersect(intersect(l[[1]], l[[2]]), l[[3]]), l[[4]]), l[[5]])
#> [1] 10 3
```

这段代码很难阅读。如果使用 **Reduce()**, 等价形式是:

```
Reduce(intersect, l)
#> [1] 10 3
```

11.4.2 谓词泛函

谓词函数是返回一个 **TRUE** 或 **FALSE** 的函数，比如 **is.character**、**all** 或 **is.NULL**。

谓词泛函对列表或数据框中的每个元素应用谓词函数。在基础 R 语言中，有三个有用的谓词泛函：**Filter()**、**Find()**和 **Position()**。

Filter()只选择那些匹配谓词函数的元素。**Find()**返回第一个匹配谓词函数的元素(或者如果 **right = TRUE**，则返回最后一个元素)。**Position()**返回第一个匹配谓词函数的元素的位置(或者如果 **right = TRUE**，则返回最后一个元素的位置)。(译者注：所谓"匹配谓词函数"，是指谓词函数返回 TRUE 的情况。)

另一个有用的谓词泛函是 **where()**，它是一个自定义的泛函，它根据一个列表(或数据框)以及一个谓词函数，生成一个逻辑向量：

```
where <- function(f, x) {  
  vapply(x, f, logical(1))  
}
```

下面的例子显示了，在什么情况下你会把这些泛函应用于数据框：

```
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()` 不是谓词函数？什么基本 R 函数最接近谓词函数版本的 `is.na()`？
2. 使用 `Filter()` 和 `vapply()` 来创建一个函数，它对数据框中的每个数值列应用 `summary` 进行统计。
3. `which()` 与 `Position()` 之间有什么关系？`where()` 与 `Filter()` 之间有什么关系？
4. 实现 `Any()` 函数，它需要一个列表和一个谓词函数作为输入，如果对列表中的任何元素谓词函数都返回 `TRUE`，那么该函数返回 `TRUE`。类似地，实现 `All()` 函数。（译者注：即列表版本的 `any()` 和 `all()` 函数）
5. 实现 Haskell 中的 `span()` 函数：给定一个列表 `x` 和一个谓词函数 `f`，`span` 返回使得谓词函数为 `TRUE` 的最长元素序列的位置。（提示：你可能会发现 `rle()` 函数有所帮助。）

11.5 数学泛函

在数学中，泛函是很常见的。极限函数(limit)、最大值函数(maximum)、求根函数(求出使得 $f(x) = 0$ 的点的集合)以及定积分函数都是泛函：给定一个函数，它们返回单个数值(或数值向量)。乍一看，这些函数似乎并不符合消除循环的主题，但是如果你更深入地研究它们，你会发现它们都是由使用了迭代的算法来实现的。在本节中，我们将使用一些 R 语言内置的数学泛函。有三种泛函，它们处理返回单个数值的函数：

`integrate()` 求出由 `f()` 定义的曲线下的面积。

`uniroot()` 求出 `f()` 在哪里为零。

`optimise()` 求出使 `f()` 达到最小（或最大）值的位置。

让我们研究这些泛函如何与一个简单的函数联合使用，这个函数是 `sin()`（正弦函数）：

```
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，**优化**(optimisation)通常用于**最大似然估计**(maximum likelihood estimation, MLE)。在**最大似然估计**问题中，我们两组参数：

数据，对于一个给定的问题它是固定的；

参数，它在我们尝试寻找最大值的时候会发生变化。

这两组参数使问题适合用**闭包**来解决。**闭包**与**优化**相结合产生了下面的方法来解**决最大似然估计**问题。

下面的例子展示了，如果我们的数据来自于泊松分布，那么我们如何为 λ 找到最大似然估计。首先，我们创建一个函数工厂，对函数工厂给定一个数据集，它返回一个对参数 λ 计算负对数似然（negative log likelihood, NLL）的函数。在 R 中，使用负值是很常见的，因为 `optimise()` 的默认行为是找最小值。（译者注：找出使 $f(x)$ 取最大值的 x ，也就是找出使 $-f(x)$ 取最小值的 x ）

```
poisson_nll <- function(x) {  
  n <- length(x)  
  sum_x <- sum(x)  
  function(lambda) {  
    n * lambda - sum_x * log(lambda) # + terms not involving lambda  
  }  
}
```

注意闭包是如何让我们预计算一些值的，这些值对于数据是不变的。（译者注：指 `n` 和 `sum_x` 的值，对一个输入向量 `x` 来说，每次进行迭代，`n` 和 `sum_x` 都是定值，所以不需要重复计算，只在生成闭包函数的时候计算一次即可。）

我们可以使用这个函数工厂来为输入数据生成具体的负对数似然函数。然后，给定一个较宽的开始范围，`optimise()` 可以让我们找到最佳值(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
```

我们可以分析一下，检查一下这些值是不是正确的：在这种情况下，它只是数据的均值，32.1 和 5.4667。另一个重要的数学泛函是 `optim()`。它是 `optimise()` 对维度大于一的情况的推广。如果你对它是如何工作的感兴趣的话，那么你可以研究 `Rvmmin` 包，它为 `optim()` 提供了一种纯 R 语言的实现。有趣的是，`Rvmmin` 并不比 `optim()` 慢，尽管它是用 R 语言而不是 C 语言编写的。对于这个问题来说，瓶颈并不在于控制优化，而在于多次计算函数。（译者注：即主要的性能瓶颈是被优化的函数，因为这个函数需要被调用很多次。）

11.5.1 练习

1. 实现 `arg_max()`。它需要一个函数和一个向量作为输入，并返回使得函数返回最大值的输入元素的位置。例如，`arg_max(-10:5, function(x) x ^ 2)` 应该返回 -10。`arg_max(-5:5, function(x) x ^ 2)` 应该返回 `c(-5, 5)`。也实现一下相应的 `arg_min()` 函数。
2. 挑战：阅读关于定点算法(fixed point algorithm)的文章 (http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-12.html#%_sec_1.3)。使用 R 语言完成练习。

11.6 循环应该被保留的情况

一些循环并没有对应的泛函。在本节中，你将了解三种常见的情况：

就地修改

递归函数

while 循环

当然，对这些问题花费很大的力气，使用泛函来实现，也是可能的，但这不是好主意。对于第一种情况，如果一定要使用泛函，那么你创建的代码将会变得难以理解。

11.6.1 就地修改

如果你需要修改现有数据框的一部分，通常使用 **for** 循环会更好。例如，下面的代码执行了一个**变量对变量**的(variable-by-variable)转换，它通过把列表中保存的**函数名字**匹配到数据框中的**变量名字**来实现。（译者注：数据框中的变量名字就是数据框的列名。）

```
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]])
}
```

我们通常不会直接使用 **lapply()** 来替换这个循环，但这是可能的。通过使用 **<<-**，可以使用 **lapply()** 来替换循环：

```
lapply(names(trans), function(var) {
  mtcars[[var]] <<- trans[[var]](mtcars[[var]])
})
```

for 循环消失了，但是代码更长了，也更难理解了。代码的读者需要理解 **<<-** 以及 **x[[y]] <<- z** 是如何工作的(这并不简单！)。简而言之，我们去掉了简单的、容易理解的 **for** 循环，并把它转化成很少有人能明白的代码：这不是个好主意！

11.6.2 递归关系

当元素之间的关系不是独立的，或者是递归定义的时候，是很难把 **for** 循环转换成泛函的。例如，**指数平滑法**(exponential smoothing)是这样工作的：它对**当前和前面的数据点**求加权平均数。下面的 **exps()** 函数使用 **for** 循环实现了**指数平滑法**。


```
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
```

我们不能消除这个 `for` 循环，因为我们没有见过这样的泛函：在 `i` 处的输出依赖 `i-1` 处的输入和输出。

在本例中，消除 `for` 循环的一种方法是解递归关系(solve the recurrence relation)(http://en.wikipedia.org/wiki/Recurrence_relation)，它可以消除递归，并代之以显式的引用。这需要一套新的数学工具，是有挑战性的，但回报是可以产生一个更简单的函数。

11.6.3 while 循环

R 语言中的另一种类型的循环结构是 `while` 循环。它会持续运行，直到满足某些条件时为止。`while` 循环比 `for` 循环更具有一般性：你可以把每一个 `for` 循环都写成 `while` 循环，但是反过来却不行。例如，我们可以把这个 `for` 循环，

```
for (i in 1:10) print(i)
```

转换为这个 **while** 循环：

```
i <- 1
while(i <= 10) {
  print(i)
  i <- i + 1
}
```

并不是每一个 **while** 循环都可以变成 **for** 循环，因为许多循环事先并不知道它们将运行多少次：

```
i <- 0
while(TRUE) {
  if (runif(1) > 0.9) break
  i <- i + 1
}
```

当你编写**模拟程序**的时候，这是个很常见的问题。在这种情况下，我们可以通过**识别问题的特征**，来删除循环。这里，在 **p = 0.1** 的**伯努利试验**失败之前，我们对**成功次数**进行计数。这是一个**几何随机变量**，所以你可以使用 **i <- rgeom(1, 0.1)** 来替换代码。以这种方式把这个问题进行变形，一般是很难做到的，但是如果你能对你的问题这么做的话，你将获益匪浅。

11.7 函数族

本节说明了**函数族**(A family of functions)。为了总结本章的内容，下面的**案例研究**将展示如何使用泛函，使得简单的**构建块**变得**强大和通用**。我将从简单的思路开始：把两个数加起来，然后使用泛函来扩展它，使它**支持多个数**、进行**并行计算**、计算**累积和**，以及**跨数组维度**求和。首先，我们定义一个非常简单的加法函数，它的输入是两个标量参数：

```
add <- function(x,y) {  
  stopifnot(length(x) == 1, length(y) == 1,  
    is.numeric(x), is.numeric(y))  
  x + y  
}
```

(我们在这里使用了 R 现有的加法运算符，它可以做更多事，但是这里关注的重点是，我们能如何扩展非常简单的构建块，使它们能做更多的事情。)

我还将添加一个 `na.rm` 参数。**辅助函数**将使这一点变得更容易：如果 `x` 缺失，那么它应该返回 `y`；如果 `y` 缺失，那么它应该返回 `x`；如果 `x` 和 `y` 都缺失，那么它应该返回函数的另一个参数：`identity`。这个函数可能比我们现在所需要的更加通用一些，但是如果我们要实现其它的**二元运算符**，那么它是有用的。

```
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
```

这让我们可以写出，能在必要时处理缺失值的 `add()` 函数：

```
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
```

为什么我们要把 `identity` 设为 0？为什么 `add(NA, NA, na.rm = TRUE)` 应该返回 0？嗯，对于其它每一个的输入，它返回一个数字，所以即使两个参数都是 `NA`，它仍然应该这样做。它应该返回什么数？我们可以找出答案，因为加法是具有结合律的，这意味着加法的顺序并不重要。也就是说，以下两个函数调用应该返回相同的值：

```
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
```

这意味着，`add(NA, NA, na.rm = TRUE)` 必须是 0，因此 `identity = 0` 是正确的默认值。

现在，我们有了最基本的、能工作的函数，然后我们可以扩展这个函数以处理更复杂的输入。一种显而易见的推广是对两个以上的数进行加法。我们可以通过迭代对两个数进行加法：如果输入是 `c(1, 2, 3)`，那么我们可以计算 `add(add(1, 2), 3)`。这是 `Reduce()` 函数的一种简单的应用场景：

```
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
```

这个看起来挺好，但是我们需要对一些特例进行测试：

```
r_add(NA, na.rm = TRUE)  
#> [1] NA  
r_add(numeric())  
#> NULL
```

这些都是错误的。在第一种情况下，我们会得到一个缺失值，尽管我们已经明确要求忽略它们。在第二种情况下，我们得到了 **NULL** 而不是长度为 1 的数值向量（就像我们使用的其它输入一样）。

这两个问题是相关的。如果我们给 **Reduce()** 传入一个长度为 1 的向量，那么它并不能“减少”(reduce)什么，所以它只是返回输入。如果我们给的输入的长度为零，那么它总是返回 **NULL**。解决这个问题最简单的方法是使用 **Reduce()** 的 **init** 参数。它会加在每一个输入向量的开头：

```
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()`相当于 `sum()`。

如果有向量化版本的 `add()`就好了，以便我们可以在两个数值向量的对应元素之间，执行加法运算。我们可以使用 `Map()`或 `vapply()`实现这一点，但是不是很完美。`Map()`返回一个列表，而不是数值向量，所以我们需要使用 `simplify2array()`。`vapply()`返回一个向量，但需要我们在的一组索引上进行循环。

```
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))  
}
```

一些测试用例有助于确保它按照我们所期望的行为运行。在这里，我们比基础 R 语言更加严格一些，因为我们不做回收。(如果你认为需要，你可以进行补充，但是我发现回收很容易导致难以发现的错误。)

```
# 两个版本都得到相同的结果  
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()`的另一个变种是**累计求和**。我们可以通过把 `Reduce()` 的 `accumulate` 参数设置为 `TRUE` 来实现：

```
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
```

这相当于 `cumsum()`。

最后，我们想为更复杂的数据结构，比如**矩阵**，定义加法。我们可以创建分别对**行**和**列**进行**求和**的函数，或者我们可以定义一个数组版本的函数，可以对任意维度进行求和。联合使用 `add()` 和 `apply()` 是很容易实现的。

```
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)
}
```

前两个相当于 `rowSums()` 和 `colSums()`。如果我们创建的每个函数在 R 中都已经有了等价函数，那么我们为什么要自寻烦恼呢？

主要有两个原因：

1. 因为所有的变种函数都是结合了简单的二元运算符(`add()`)和经过良好测试的函数(`Reduce()`, `Map()`, `apply()`)来实现的，所以我们知道这些变种函数的行为是一致的。
2. 我们可以把相同的基本框架应用到其它运算符，尤其是那些可能在基本 R 中没有合适变种的运算符。

这种方法的缺点是，这些实现的效率没有那么多高。(例如，`colSums(x)`比 `apply(x, 2, sum)`快得多。)

然而，即使它们不是那么快，简单的实现仍然是一个很好的起点，因为它们不太可能有错误。

当你创造更快版本的实现时，你可以比较执行结果，以确保你的快速版本仍然是正确的。如果你喜欢这一节，你可能还会喜欢《List out of lambda》

(<http://stevelosh.com/blog/2013/03/list-out-of-lambda/>)，一篇由 Steve Losh 撰写的博客文章，文章展示了如何由更原始的语言特性(如闭包，即 `lambdas`)来产生高层次的语言结构(比如列表)。

11.7.1 练习

1. 实现 `smaller` 和 `larger` 函数，给出两个输入，它们返回较小或较大的值。实现 `na.rm = TRUE`：相等应该是什么？(提示：`smaller(x, smaller(NA, NA, na.rm = TRUE), na.rm = TRUE)`必须为 `x`，所以 `smaller(NA, NA, na.rm = TRUE)`必须大于任何其它 `x` 的值)。使用 `smaller` 和 `larger` 来实现 `min()`、`max()`、`pmin()`、`pmax()`的等价函数以及新函数 `row_min()`和 `row_max()`。

2. 创建一个表，在列上有 `and`、`or`、`add`、`multiply`、`smaller` 和 `larger`，在行上有二元运算符、`reducing` 的变种、向量化的变种和数组变种。
 - a) 使用基本 R 中的函数名字来填写表格，每个函数都执行一种功能。
 - b) 比较现有 R 函数的名字和参数。它们是一致的吗？你能进行改善吗？
 - c) 实现表格项中缺失的函数。
3. `paste()` 如何适应这种结构？构成 `paste()` 的标量二元函数是什么？`paste()` 的 `sep` 和 `collapse` 参数相当于什么？有没有哪个 `paste` 的变种函数在 R 中没有实现？

12 函数运算符

在本章中，你将了解**函数运算符**(Function Operator)。**函数运算符**是一类特殊的函数：它将一个(或多个)函数作为输入，然后返回一个函数作为输出。在某些方面，**函数运算符**与**泛函**是相似的：没有它们你也可以工作，但是它们可以使你的代码**可读性更强、富有表现力**，并且它们帮你更快地编写代码。与泛函的主要区别在于，泛函是提取一般的**循环模式**，而**函数运算符**提取使用**匿名函数**的一般模式。

下面的代码显示了一个简单的**函数运算符**，**chatty()**。它封装了一个函数，生成了一个新的函数，该函数可以打印出它的第一个参数。这是有用的，因为它给了你一个查看泛函——比如 **vapply()**——如何工作的窗口。

```
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，我们看到了许多内置的泛函，如 `Reduce()`、`Filter()` 和 `Map()`，只有很少的参数，所以我们不得不使用匿名函数来告诉它们如何工作。在这一章中，我们将对常见的匿名函数构建专业的替代品，它允许我们更清晰地交流我们的意图。例如，在第 11.2.2 节中，我们把一个匿名函数与 `Map()` 联合起来使用，以提供固定的参数：

```
Map(function(x,y) f(x,y,zs), xs,ys)
```

在本章后面，我们将了解使用 `partial()` 函数的部分应用程序(partial application)。部分应用程序封装了匿名函数，为其提供默认参数，并让我们写出更简洁的代码：

```
Map(partial(f,zs=zs), xs,yz)
```

这是函数运算符的一种重要的应用：通过改变输入函数，你可以从泛函中消除参数。事实上，只要函数的输入和输出保持不变，这种方法会让你的泛函更有扩展性，通常是你没有想到过的方法。本章涵盖了四种重要的函数运算符类型：行为、输入、输出以及联合。对于每一种类型，我都将向你展示一些有用的函数运算符，以及如何使用另一种分解问题(decompose problems)的方法：组合多个函数而不是组合多个参数。我们的目标不是详尽地列出所有的函数运算符，而是有选择地展示它们如何与其它函数式编程技术一起工作。对于你自己的工作，你将需要考虑和试验函数运算符怎样帮助你解决重复出现的问题。

本章概要

12.1 节介绍了改变函数功能的函数运算符，比如，自动记录磁盘的使用情况，或者，保证函数只运行一次等等。

12.2 节向你展示了如何编写函数运算符以控制函数的输出。这些改变可以做简单的事情，比如捕获错误，或者从根本上改变函数的功能。

12.3 节描述了如何使用如 `Vectorize()` 或 `partial()` 这样的函数运算符，以修改输入的函数。

12.4 节显示了函数运算符的威力，它可以通过函数组合或者逻辑操作来结合多个函数。

前提条件

本章需要使用 `memoise`、`plyr` 和 `pryr` 包中的函数运算符。可以通过运行 `install.packages(c("memoise", "plyr", "pryr"))` 进行安装。

12.1 行为函数运算符

行为函数运算符(Behavioural FOs)保持函数的输入和输出不变，但是添加一些额外的行为。在本节中，我们将看看实现了三种有用行为的函数：

1. 添加延迟，以避免对服务器的请求过多。
2. 每进行了 `n` 次调用，都会把输出信息打印到控制台，以检查长时间运行的进程。
3. 缓存先前的计算结果以提高性能。

为了模拟这些行为，假设我们想通过一些 URL 下载文件，这些 URL 保存在一个长向量中。使用 `lapply()` 和 `download_file()` 实现很简单：

```
download_file <- function(url, ...) {  
  download.file(url, basename(url), ...)  
}  
lapply(urls, download_file)
```

(`download_file()` 是 `utils::download.file()` 的一个简单的包装，它提供了一个合理的默认文件名。)

我们可能想要为这个函数添加许多有用的行为。如果列表很长，那么我们可能想要在每十个 URL 之间打印一个`.`，这样我们就能知道函数仍然是在正常工作的。如果我们是在互联网上下载文件的，那么我们可能想在每两次请求之间添加一个小延迟(`delay`)，这样可以避免对服务器的过度请求。在 `for` 循环中要实现这些行为是相当复杂的。因为我们需要一个外部计数器，所以我们不能再使用 `lapply()`：

```
i <- 1
for(url in urls) {
  i <- i + 1

  if (i %% 10 == 0) cat(".")
  Sys.delay(1)
  download_file(url)
}
```

要理解这段代码是很困难的，因为它关注了不同的行为(迭代、打印和下载)，而且这些行为是互相交错的。在本节剩下的篇幅中，我们将为每种行为都创建进行封装的函数运算符，以便使我们可以写这样代码：

```
lapply(urls, dot_every(10, delay_by(1, download_file)))
```

实现 `delay_by()` 很简单，它遵循了一种基本模板，该基本模板与我们在本章看到的大多数函数运算符是相同的：

```
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()` 要更复杂一点，因为它需要管理一个计数器。幸运的是，我们已经在 10.3.2 节看过是怎样做的。

```
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))
#> .....
```

请注意，我在每个函数运算符中，都把函数作为最后一个参数。当我们组合多个函数运算符时，这样会更容易阅读。如果函数是第一个参数，那就不是这样了：

```
download <- dot_every(10, delay_by(1, download_file))
```

而是这样：

```
download <- dot_every(delay_by(download_file, 1), 10)
```

这样的代码难以阅读，因为(例如)`dot_every()` 的参数远离它的调用部分。(译者注：即 `dot_every` 和 `10` 离得太远了) 这种情况有时被称为多层三明治问题

(http://en.wikipedia.org/wiki/Dagwood_sandwich): 你在面包片(括号)之间, 填充了太多东西(太多很长的参数)。

我也尝试给**函数运算符**取一个具有描述性的名字: `delay by 1 (second),(print a) dot every 10 (invocations)`。(译者注: 作者取了英文句子中的一些关键词以组成函数运算符的名字。句子的意思是: 延迟一秒, 每十次调用打印一个点。) 你的代码中使用的函数名称越是能清晰地表达你的意图, 那么就更容易让其它人(包括你自己)阅读和理解你的代码。

12.1.1 备忘录

你可能担心的另一件事情是, 当下载多个文件时, 偶尔会多次下载同一个文件。你可以对输入的 URL 列表调用 `unique()` 以避免这个问题, 或者手动管理一种将 URL 映射到结果的数据结构。(译者注: 比如建立一个映射表, 映射表的"键"是 URL, "值"是下载的内容; 对于后续的 URL, 先在这个映射表中查找当前键值是不是存在。) 另一种可选择的方法是使用**备忘录(Memoisation)**: 修改函数让它自动缓存自己的结果。

```
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
```

备忘录是计算机科学中对**内存**和**速度**进行**平衡**的一个经典例子。有**备忘录**的函数能运行得更快，因为它使用更多的内存来存储所有以前的输入和输出。一个实际使用**备忘录**的例子是计算**斐波纳契数列**。**斐波纳契数列**是**递归**定义的：最前面的两个值是 **1** 和 **1**，然后 $f(n) = f(n-1) + f(n-2)$ 。R 语言实现的简单版本会非常缓慢，因为序列中的每一个值都计算了很多很多次，例如求 **fib(10)** 要计算 **fib(9)** 和 **fib(8)**，求 **fib(9)** 要计算 **fib(8)** 和 **fib(7)**，等等。加了**备忘录**的 **fib()** 实现了更快的版本，因为每个值仅计算一次。

```
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
```

让所有函数都使用备忘录，是不可行的。例如，使用了备忘录的随机数发生器将不再是随机的：

```
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
```

一旦我们理解了 `memoise()`，那么把它应用到我们的问题上是很简单的：

```
download <- dot_every(10, memoise(delay_by(1, download_file)))
```

这里给出了一个很容易与 `lapply()` 联合使用的函数。但是，如果在 `lapply()` 内部的循环中出现了错误，那么它很难告诉我们发生了什么情况。下一节将展示如何使用函数运算符看到 `lapply` 的内部情况。

12.1.2 捕获函数调用

使用泛函的一个挑战是我们很难看到它的内部情况。它们的内部喜欢使用 `for` 循环，但是不容易看到里面的情况。幸运的是，我们可以使用函数运算符 `tee()` 看到内部情况。

`tee()` 定义如下，它有三个参数，所有的参数都是函数：`f`，要修改的函数；`on_input`，对 `f` 函数的输入调用的函数；以及 `on_output`，对 `f` 函数的输出调用的函数。

```
ignore <- function(...) NULL
tee <- function(f, on_input = ignore, on_output = ignore) {
  function(...) {
    on_input(...)
    output <- f(...)
    on_output(output)
    output
  }
}
```

(该函数是受到 unix shell 命令 `tee` 的启发，它是对文件流进行分支的操作，这样你就既可以显示发生了什么，又可以将中间结果保存到一个文件中。)(译者注：`tee` 指令会从标准输入设备中读取数据，然后将其内容输出到标准输出设备，并且同时保存成文件。) 我们可以用 `tee()` 来查看泛函 `uniroot()` 的内部，并看看它是如何通过迭代来解决问题的。下面的例子寻找 `x` 和 `cos(x)` 的交点：

```
g <- function(x) cos(x) - x
zero <- uniroot(g, c(-5, 5))
show_x <- function(x, ...) cat(sprintf("%+.08f", x), "\n")
# 函数被计算的地方：
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
```

```
# 函数的值：
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
```

`cat()` 允许我们看到当函数运行时发生了什么，但在函数运行完成之后，我们没办法使用它打印出来的值。要做到这一点，我们可以创建一个 `remember()` 函数来捕获调用序列，它会记录每一个被调用的参数，并把它们存在一个列表中。这里需要少量在 7.2 节中讲过的 S3 代码。

```
remember <- function() {
  memory <- list()

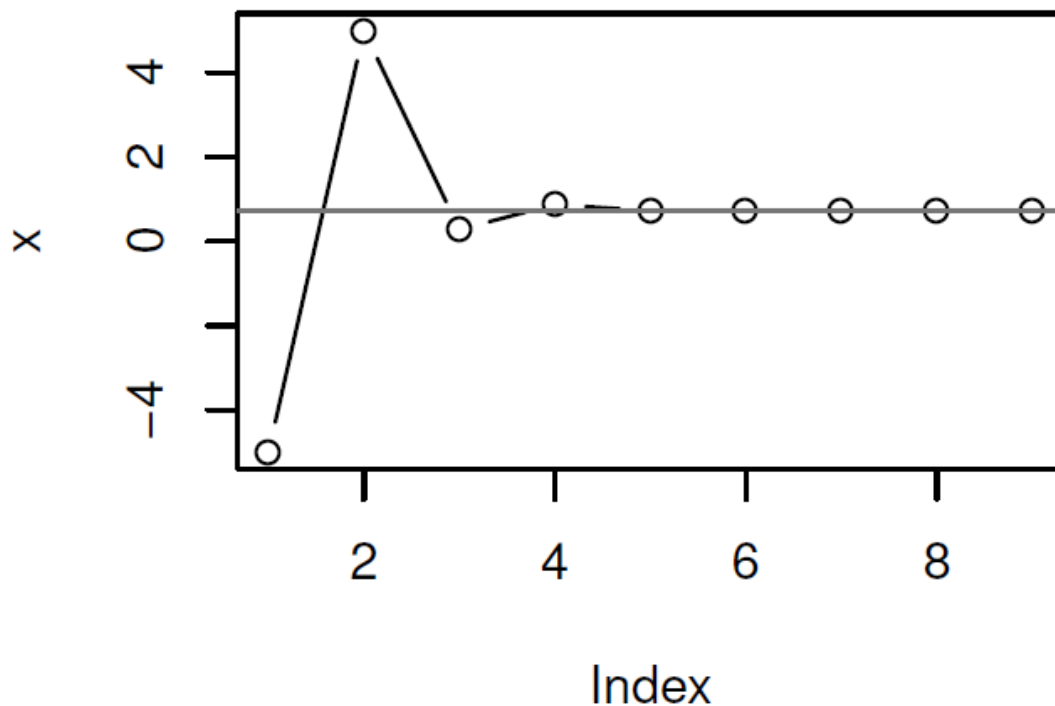
  f <- function(...) {
    # 这样是效率很差的！
    memory <- append(memory, list(...))
    invisible()
  }
  structure(f, class = "remember")
}

as.list.remember <- function(x, ...) {
  environment(x)$memory
}
```

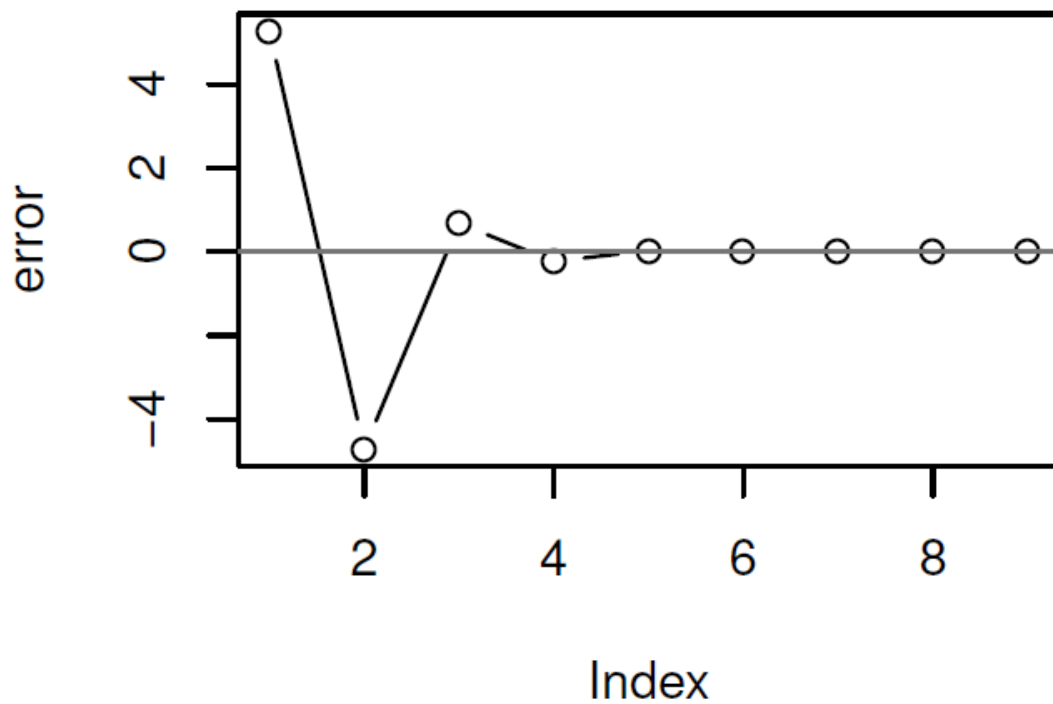
```
print.remember <- function(x, ...) {
  cat("Remembering...\n")
  str(as.list(x))
}
```

现在我们可以画一幅图显示，在最终答案中，**uniroot** 是怎样求根的：

```
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 延迟性

我们到现在为止看到的**函数运算符**都遵循一种通用的模式：

```
funop <- function(f, otherargs) {  
  function(...) {  
    # 可能做某些事  
    res <- f(...)  
    # 可能做其它事  
    res  
  }  
}
```

不幸的是，这种实现有一个问题，因为函数都是**延迟计算**的：在应用**函数运算符**和**计算函数**之间，**f()**可能已经发生了变化。你使用 **for** 循环或使用 **lapply()** 来应用多个**函数运算符**，这是一个特殊问题。在接下来的例子中，我们构造一个函数列

表，并对其中的每个函数都进行**延迟**。(译者注：调用 12.1 节中定义的 `delay_by` 函数) 当我们试图计算**均值**的时候，我们却得到了**和**。

```
funs <- list(mean = mean, sum = sum)
funs_m <- lapply(funs, delay_by, delay = 0.1)
funs_m$mean(1:10)
#> [1] 55
```

我们可以通过强制计算 `f()` 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
```

当你创建新的**函数运算符**时，都应该这么做，这是个好习惯。

12.1.4 练习

1. 编写一个**函数运算符**，每当一个函数运行的时候，它都会把**时间戳**和**消息**记录到一个文件中。
2. 以下函数是做什么的？能给它取一个好名字吗？

```
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. 修改 `delay_by()`，它可以保证，自从函数上一次被调用以来，经过(elapsed)了一定的时间，而不是延迟一段固定的时间。也就是说，如果你调用了 `g <- delay_by(1, f); g(); Sys.sleep(2); g()`，那么这里不应该有额外的延迟。
4. 写一个函数运算符 `wait_until()`，它会延迟执行函数，直到某个特定的时间，函数才会执行。
5. 有三个地方可以添加 `memoise` 调用：为什么我们要选择那一个？

```
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()` 函数的效率低下？你能以更高效的方式来实现它吗？
7. 下面的代码来自于
`stackoverflow(http://stackoverflow.com/questions/8440675)`，为什么它没有按照你预想的那样工作？

```
# 使用斜率 a 和截距 b 返回一个线性函数。
```

```
f <- function(a, b) function(x) a * x + b
```

```
# 使用不同的参数创建一个函数列表。
```

```
fs <- Map(f, a = c(0, 1), b = c(0, 1))
```

```
fs[[1]](3)
```

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

```
# 应该返回 0 * 3 + 0 = 0
```

怎样修改可以让 `f` 正确地工作？

12.2 输出函数运算符

下一个比较复杂的是修改函数的输出。这可能非常简单，也可能返回一些与其通常的输出完全不同的值，完全改变了函数的操作。在这一节中，你将了解两种简单的修改，`Negate()`和`failwith()`，以及两种根本性的修改，`capture_it()`和`time_it()`。

12.2.1 小的修改

`base::Negate()`和`plyr::failwith()`提供了两种比较小、但是很有用的函数修改，与泛函结合的时候尤为方便。`Negate()`接受一个返回逻辑向量的函数(即，谓词函数)，并返回该函数的否定值。(译者注：即，取反操作。)当你需要的返回值，是与函数返回值相反的时候，这可能是一种有用的快捷方式。`Negate()`的原理非常简单：

```
Negate <- function(f) {  
  force(f)  
  function(...) !f(...)  
}  
(Negate(is.null))(NULL)  
#> [1] FALSE
```


我经常使用这个思路创建一个函数，`compact()`，它会从列表中删除所有的空元素：

```
compact <- function(x) Filter(Negate(is.null), x)
```

`plyr::failwith()` 函数把抛出错误的函数转化为当发生错误时，返回一个默认值的函数。`failwith()` 的原理也很简单，它只是包装了 `try()`，这个函数捕获错误，并允许继续执行。

```
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
```

(如果你还没有见过 `try()`，那么在 9.3.1 节中有更详细地讨论。)

`failwith()` 结合泛函是非常有用的：你可以完成迭代，然后找出出错的原因，而不是传播失败，并且终止更高层次的循环。例如，假设你要拟合一组广义线性模型 (GLM)，数据存在一些数据框中，这些数据框放在一个列表中。由于优化问题，拟合有时会失败，但是你还是希望能够尽量拟合所有的模型，等到拟合完成之后，再回来看看那些失败的地方：

如果有任何模型拟合失败，那么所有的模型都会拟合失败：

```
models <- lapply(datasets, glm, formula = y ~ x1 + x2 * x3)
```

如果一个模型拟合失败，那么它将得到一个 NULL 值。

```
models <- lapply(datasets, failwith(NULL, glm),  
formula = y ~ x1 + x2 * x3)
```

使用 compact 删除拟合失败的模型(即 NULL)

```
ok_models <- compact(models)
```

提取与拟合失败的模型对应的数据集

```
failed_data <- datasets[vapply(models, is.null, logical(1))]
```

我认为这是一个很好的例子，它结合了**泛函**和**函数运算符**：它可以简洁地表达你需要什么来解决一个常见的数据分析问题。

12.2.2 改变函数的功能

其它的输出**函数运算符**，可以更深远地影响函数的操作。我们可以返回**函数计算**中的其它一些效果，而不是返回原来的返回值。这里有两个例子：

返回 **print()** 函数打印的文本：

```
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"
```

返回函数运行的时间：

```
time_it <- function(f) {  
  force(f)  
  function(...) {  
    system.time(f(...))  
  }  
}
```

`time_it()`可以让你重写"泛函"那一章的一些代码：

```
compute_mean <- list(  
  base = function(x) mean(x),  
  sum = function(x) sum(x) / length(x)  
)  
x <- runif(1e6)  
# 在前面，我们使用了一个匿名函数对执行时间进行计时：  
# lapply(compute_mean, function(f) system.time(f(x)))  
# 现在，我们可以组合函数运算符：  
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
```

在这个例子中，使用**函数运算符**并没有获得太大的好处，因为组合很简单，我们对每个函数进行了相同的操作。一般来说，在你使用多个操作符或者创建它们与使用它们之间的差距很大的情况下，使用**函数运算符**是最有效率的。

12.2.3 练习

1. 创建一个 **negative()** 函数运算符，它会**翻转**(flip)函数输出的符号。(译者注：正变负，负变正。)
2. **evaluate** 包使得从**表达式**中捕获**所有的输出**(结果、文本、消息、警告、错误和图形)变得很容易。创建一个像 **capture_it()** 的函数，它能捕获函数生成的**警告**和**错误**。
3. 创建一个**函数运算符**，它可以记录工作目录中文件的创建或删除。(提示：使用 **dir()**和 **setdiff()**) 你还想跟踪什么其它全局效果吗？

12.3 输入函数运算符

下一个比较复杂的是修改函数的输入。同样的，你可以对函数的工作方式进行小的**修改**(如，设置默认参数值)，或者大的**修改**(如，转换输入，从标量到向量，或者从向量到矩阵)。

12.3.1 填写函数的参数：部分函数应用

匿名函数的一种常见用法是创建函数的一个变体，它可以让某些参数已经被**填写**(filled in)过了。这就是所谓的**部分函数应用**(partial function application)，由 **pryr::partial()** 来实现。在你读过第 14 章后，我鼓励你阅读 **partial()** 的源代码，并指出它是如何工作的——它只有 5 行代码！**partial()** 允许我们把下面的代码：

```
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)
```

替换成这样：

```
f <- partial(g, b = 1)
compact <- partial(Filter, Negate(is.null))
Map(partial(f, zs = zs), xs, ys)
```

我们可以使用这个思路来简化使用函数列表时的代码。不要写这样的代码：

```
funs2 <- list(
  sum = function(...) sum(..., na.rm = TRUE),
  mean = function(...) mean(..., na.rm = TRUE),
  median = function(...) median(..., na.rm = TRUE)
)
```

我们可以这样写：

```
library(pryr)
funs2 <- list(
  sum = partial(sum, na.rm = TRUE),
  mean = partial(mean, na.rm = TRUE),
  median = partial(median, na.rm = TRUE)
)
```

在许多函数式编程语言中，使用部分函数应用是一种简单的工作，但是，它并不完全清楚如何与 R 的延迟计算规则进行交互。plyr::partial()使用的方法，是创建一个函数，该函数尽可能与你手工创建的匿名函数相似。Peter Meilstrup 在他的 ptools 包中使用了不同的方法(<https://github.com/crowding/ptools/>)。如果你对这个话题感兴趣，那么你可以了解一下他创造的二元操作符：%()%、%>>% 和%<<%。

12.3.2 改变输入类型

对函数的输入做出重大改变也是有可能的，可以使函数处理完全不同类型的数据。有一些现有的函数是按照这种思路进行工作的：

`base::Vectorize()` 将一个标量函数转换为向量函数。它需要传入一个非向量化的函数，以及在 `vectorize.args` 参数中指定需要向量化的函数参数。这并不会提升性能，但是如果你想以快速且简单的方法创建向量化的函数，那么它是有用的。例如，对 `sample()` 函数的一种有用的扩展是对 `size` 参数进行向量化。这么做可以让你在一次调用中生成多个样本。

```
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
```

在这个示例中，我们使用了 `SIMPLIFY = FALSE`，以确保新的向量化函数始终返回一个列表。这通常是你想要的。

`splat()` 把接受多个参数的函数，转换为需要单个参数列表的函数。

```
splat <- function (f) {
  force(f)
  function(args) {
```

```
do.call(f, args)
}
}
```

如果你想使用不同的参数来调用一个函数，那么这是有用的：

```
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
```

`plyr::colwise()` 把一个用于向量的函数，转化为可以用于数据框的函数：（译者注：对数据框的每一列调用某个函数。）

```
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. 我们之前的 `download()` 函数只能下载一个文件。怎样使用 `partial()` 和 `lapply()` 来创建一个函数，使它可以一次性下载多个文件？比较使用 `partial()` 和手动编写函数的优缺点分别是什么？
2. 阅读 `plyr::colwise()` 的源代码。代码是如何工作的？`colwise()` 的三个主要任务是什么？怎样可以通过把每个任务都实现成函数运算符，使得 `colwise()` 变得更简单？（提示：想一下 `partial()` 函数。）
3. 编写函数运算符，它将一个返回数据框的函数，转换为返回矩阵的函数，或者把返回矩阵的函数，转换为返回数据框的函数。如果你理解 S3，那么可以把它们称之为 `as.data.frame.function()` 和 `as.matrix.function()`。
4. 你已经见过了五个函数，它们可以对函数进行修改，使得函数的输出从一种形式转换成另一种形式。它们都是哪些函数？画一个表格列出各种输出类型的组合：表格的行都是输入类型，列表示输出类型。对于表中空白的格子，你能写出什么函数运算符以进行填充？请写出使用这些函数运算符的例子。
5. 在这一章以及前一章中，看看所有使用匿名函数来部分应用 (partially apply) 一个函数的例子。使用 `partial()` 来替换匿名函数。你认为结果怎么样？代码阅读起来是更容易了还是更难了？

12.4 联合函数运算符

除了操作单个函数以外，函数运算符还可以使用多个函数作为输入。一个简单的例子是 `plyr::each()`。它需要一个函数列表，函数列表中都是向量化的函数，然后把它们组合成单个函数。


```
summaries <- plyr::each(mean, sd, median)
summaries(1:10)
#> mean sd median
#> 5.500 3.028 5.500
```

两个更复杂的例子是通过**组合**或者通过**布尔代数**，来**联合**函数。这种能力像胶水一样，让我们把多个函数连接起来。

12.4.1 函数组合

把函数联合起来的一种重要方法是通过**组合**：**f(g(x))**。组合需要一个**函数列表**，并按照顺序把它们都应用到输入参数上。这是**匿名函数**的常见模式的一种替代方式，这些**匿名函数**可以把多个函数链接在一起，以得到想要的结果：

```
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
```

一种简单版本的**组合**看起来像这样：

```
compose <- function(f, g) {
  function(...) f(g(...))
}
```

(**plyr::compose()**提供了一个全功能(full-featured)的替代，它可以接受多个函数，将会用于其余的例子中。) 这让我们可以这样写：

```
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，函数组合往往是用中缀操作符表示的： o ， $(f \circ g)(x)$ 。Haskell——一种很流行的函数式编程语言——使用`.`来达到相同的目的。在 R 语言中，我们可以创建自己的中缀组合函数：

```
"%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
```

组合还可以用于实现非常简洁的 `Negate`，它是对 `compose()` 函数，调用了 `partial` 的版本。

```
Negate <- partial(compose, `!`)
```

我们可以使用函数组合实现总体标准偏差(population standard deviation)：

```
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
```

这种类型的编程被称为隐性(tacit)或无点(point-free)编程。(术语"无点"一词来自于拓扑学中使用"点"来引用值；这种风格也称为无点(pointless)风格)。在这种编程风格中，你可以隐式地引用变量。相反，你关注的是函数的高层组合，而不是底层的数据流；关注的是做什么动作，而不是动作作用的对象。因为我们只使用

了函数而不是参数，所以我们要使用动词，而不是名词。这种风格在 Haskell 中是很常见的，并且在基于堆栈的编程语言中，比如 Forth 和 Factor，这也是典型的风格。在 R 语言中，这不是一种非常自然或者优雅的风格，但是它确实挺有趣的。

`compose()` 结合 `partial()` 尤其有用，因为 `partial()` 允许你为组合函数提供附加参数。这种编程风格的一种不错的作用，是可以让函数参数保持在函数名附近。这是很重要的，因为随着代码的增长，代码会变得越来越难以理解。

下面，我将对本章第一节例子进行修改，使用上面描述的两种函数组合风格。用两种风格写的代码都比原始代码要长，但是它们可能更容易理解，因为函数和它的参数贴得更近。注意，我们仍然需要从右到左(从下到上)进行阅读：第一个被调用的函数是写在最后的那一个。我们可以定义相反方向的 `compose()` 函数，但是从长远来看，这可能会导致混乱，因为我们创建的这一小部分代码会与语言的其它部分的阅读方式不同。

```
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) %>%
  memoise %>%
  partial(delay_by, 1) %>%
  download_file
```

12.4.2 逻辑谓词和布尔代数

当我使用 `Filter()` 以及其它一些作用于逻辑谓词的泛函的时候，我发现自己经常使用匿名函数来组合多个条件：

```
Filter(function(x) is.character(x) || is.factor(x), iris)
```

作为替代，我们可以定义组合逻辑谓词的函数运算符：

```
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(...)  
  }  
}
```

这让我们可以这样写：

```
Filter(or(is.character, is.factor), iris)  
Filter(not(is.numeric), iris)
```

我们现在有了一个处理函数、而不是函数结果的布尔代数。

12.4.3 练习

1. 使用 `Reduce` 和 `%o%` 来实现你自己的 `compose()`。作为加分项，请不要调用函数进行实现。
2. 扩展 `and()` 和 `or()` 以处理任意数量的输入函数。你能使用 `Reduce()` 来做吗？你能让它们保持延迟计算吗？（比如对于 `and()`，一旦它发现第一个 `FALSE` 值就会立即返回？）
3. 实现 `xor()` 二元运算符。使用现有的 `xor()` 函数来实现。以 `and()` 和 `or()` 的组合来实现。每种方法的优点和缺点是什么？也考虑一下怎样避免新函数与现有的 `xor()` 函数发生冲突，以及如何改变 `and()`、`not()` 和 `or()` 的名字，以保持它们的一致性。
4. 在上面，我们为返回逻辑函数的函数实现了布尔代数。请实现返回数值向量的初等代数函数。（`plus()`、`minus()`、`multiply()`、`divide()`、`exponentiate()`、`log()`）

第三部分 编程语言层面的计算

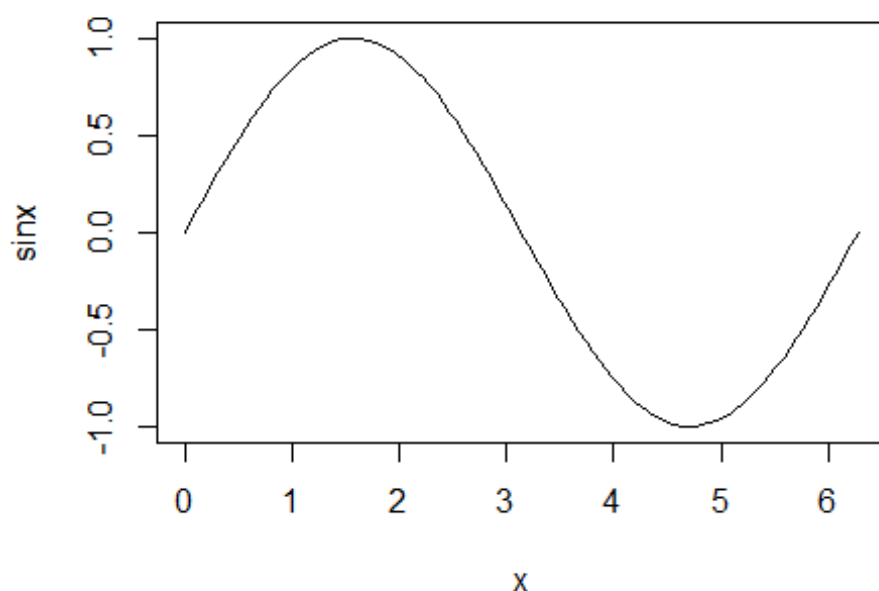
本部分讲述**编程语言层面的计算**(Computing on the language)的内容，它操纵的**编程语言本身**，这在各种编程语言中都是比较高级的内容和技巧。

13 非标准计算

"灵活的语法——如果它不导致歧义——对交互式编程语言来说似乎是一种合理的事情。" — Kent Pitman

R 语言不仅拥有强大的计算工具，也有对**算出这些值的动作**的强大工具。如果你使用过其它编程语言，那么它们是 R 语言最令人惊讶的特性之一。考虑以下绘制一条正弦曲线的简单代码片段：

```
x <- seq(0, 2 * pi, length = 100)
sinx <- sin(x)
plot(x, sinx, type = "l")
```



看看坐标轴上的标签。R 语言是怎么知道 `x` 轴上的变量称为 `x`，而 `y` 轴上的变量称为 `sinx` 的呢？在大多数编程语言中，你只能访问函数参数的值。而在 R 语言中，你还可以访问用于计算的代码。

这使得它可以通过非标准的方式来计算代码：使用所谓的“非标准计算”(non-standard evaluation, NSE)。在进行交互式数据分析时，非标准计算对于函数尤其有用，因为它可以大大减少字符的输入。

本章概要

13.1 节教你如何使用 `substitute()` 来捕获未计算的(unevaluated)表达式。

13.2 节向你展示了把 `subset()` 与 `substitute()`、`eval()` 结合使用，可以让你从数据框中简洁地选择行。

13.3 节讨论非标准计算的作用域问题，并将告诉你如何解决。

13.4 节给出了为什么每一个使用了非标准计算的函数都应该有个后门(escape hatch)函数，即，使用普通计算的函数。

13.5 节教导你如何使用 `substitute()` 处理没有后门的函数。

13.6 节通过讨论非标准计算的缺点来结束本章。

前提条件

在阅读这一章之前，确保你熟悉环境(第 8 章)和词法作用域(6.2 节)。你还需要使用 `install.packages("pryr")` 安装 `pryr` 包。一些练习需要 `plyr` 包，你可以使用 `install.packages("plyr")` 从 CRAN 上进行安装。

13.1 捕获表达式

`substitute()` 使非标准计算成为可能。它着眼于函数参数，而不是参数值，它看到的是用于计算的代码：

```
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，我们不用为 `substitute()` 返回的是什么而烦恼(第 14 章的话题)，它返回的是一个表达式(expression)。`substitute()` 之所以能起作用，是因为函数参数被表示为

一种称为**承诺**(promise)的特殊类型的对象。**承诺**捕获计算**值**需要的**表达式**，以及表达式计算所处的**环境**。通常你不会察觉到**承诺**，因为当你第一次访问一个**承诺**的时候，它的代码已经在它的环境中进行计算了，并产生了一个值。

`substitute()`通常与 `deparse()` 搭配使用。该函数需要 `substitute()` 的结果，即一个表达式，并把它转换成为一个字符向量。

```
g <- function(x) deparse(substitute(x))
g(1:10)
#> [1] "1:10"
g(x)
#> [1] "x"
g(x + y^2)
#> [1] "x + y^2"
```

基础 R 语言中有很多函数使用了这样的思路。有一些是为了避免使用引号：

```
library(ggplot2)
# 与以下相同
library("ggplot2")
```

其它函数，比如 `plot.default()`，使用它们来提供默认的标签(label)。`data.frame()` 使用计算变量的表达式来对变量打标签：(译者注：也就是为数据框的**列**取名字。)

```
x <- 1:4
y <- letters[1:4]
names(data.frame(x, y))
#> [1] "x" "y"
```

我们将通过一个特别有用的**非标准计算**应用，来学习所有这些示例背后的思想：

`subset()` 函数。

13.1.1 练习

1. 在编程时，`deparse()`的一个需要注意的重要功能是：如果输入太长了，那么它可以返回多个字符串。例如，下面的调用会产生一个长度为 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)
```

为什么会发生这种情况？请仔细阅读文档。你能为 `deparse()` 编写一个包装器，使得它总是返回单个字符串吗？

2. 为什么 `as.Date.default()` 要使用 `substitute()` 和 `deparse()`？为什么 `pairwise.t.test()` 也使用它们？阅读它们的源代码。
3. `pairwise.t.test()` 会假设 `deparse()` 总是返回一个长度为 1 的字符向量。你能构造一个输入，使它违反这种期望吗？为什么会这样呢？
4. 上面定义的 `f()`，只是调用了 `substitute()`。为什么我们不能使用它来定义 `g()`？换句话说，下面的代码将返回什么？首先做一个推测。然后运行代码并考虑一下结果。

```
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)))
```

13.2 子集中的非标准计算

虽然打印出提供了参数值的代码是有用的，但是实际上，我们可以对未计算的代码做更多的事情。以 `subset()` 为例。这是一种有用的交互的快捷方式，它对数据框进行**取子集操作**：使用它你可以减少输入，而不需要多次地重复输入数据框的名字。

```
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
# 相当于:
# sample_df[sample_df$a >= 4, ]
subset(sample_df, b == c)
#> a b c
#> 1 1 5 5
#> 5 5 1 1
# 相当于:
# sample_df[sample_df$b == sample_df$c, ]
```

`subset()`有些特别，因为它实现了不同的作用域规则：表达式 `a >= 4` 或 `b == c` 是在指定的数据框中进行计算的，而不是在当前环境中或者全局环境中进行计算的。这是非标准计算的本质。

`subset()`是如何工作的呢？我们已经看到过如何捕获参数的表达式而不是它的结果，所以我们只需要弄清楚，如何在合适的上下文中计算表达式即可。具体来说，我们希望 `x` 被解释成 `sample_df$x`，而不是 `globalenv()[x]`。要做到这一点，我们需要 `eval()`。此函数接受一个表达式，并在指定的环境中计算表达式。

在我们研究 `eval()` 之前，我们需要一个更有用的函数：`quote()`。它捕获未计算的表达式，比如 `substitute()`，但不进行任何让 `substitute()` 产生迷惑的高级转换。

`quote()`总是按照原样返回其输入：

```
quote(1:10)
#> 1:10
quote(x)
```

```
#> x
quote(x + y^2)
#> x + y^2
```

我们要对 `eval()` 进行实验，就需要 `quote()`，因为 `eval()` 的第一个参数是一个表达式。所以，如果你只提供一个参数，那么它将在当前环境下对表达式进行计算。因此，不管 `x` 是什么，`eval(quote(x))` 都完全等价于 `x`：

```
eval(quote(x <- 1))
eval(quote(x))
#> [1] 1
eval(quote(y))
#> Error: object 'y' not found
```

`quote()` 和 `eval()` 是相反的。在下面的示例中，每个 `eval()` 都会抵消一个 `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()` 的第二个参数指定了一个环境，代码会在这个环境中执行：

```
x <- 10
eval(quote(x))
#> [1] 10
```

```
e <- new.env()
e$x <- 20
eval(quote(x), e)
#> [1] 20
```

因为列表和数据框通过与环境相似的方式把名字绑定到值，所以 `eval()` 的第二个参数不需要仅限于环境：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
```

这给了我们 `subset()` 的一部分：

```
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()` 的一个常见错误是忘记引用 (`quote()`) 了第一个参数。结果比较如下：

```
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
```

我们可以使用 `eval()` 和 `substitute()` 来写 `subset()`。我们首先捕获表示条件 (`condition`) 的调用，然后我们在数据框的上下文(context of the data frame)中对它进行计算，最后，我们使用结果来进行取子集操作：

```
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
```

13.2.1 练习

1. 预测下面代码的结果：

```
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. 如果你对单列的数据框使用 `subset2()`，那么会有一个错误。以下代码应该返回什么？怎样修改 `subset2()`，使得它返回正确的对象类型？

```
sample_df2 <- data.frame(x = 1:10)  
subset2(sample_df2, x > 8)  
#> [1] 9 10
```

3. 真正的 `subset` 函数(`subset.data.frame()`)会在一些条件下删除缺失值。修改 `subset2()`，使它做相同的事情：去掉不满足条件的行。

4. 如果你在 `subset2()` 内部使用 `quote()` 而不是 `substitute()`，会发生什么？
5. `subset()` 的第二个参数允许你选择变量。(译者注：即，数据框的列。) 它似乎把变量名当成位置一样。这使得你可以这么做：`subset(mtcars, , -cyl)` 去掉 `cylinder` 列，或者 `subset(mtcars, , disp:drat)` 选择 `disp` 和 `drat` 之间的所有变量。这是怎么做到的呢？我抽取了一个更容易理解函数。

```
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)
```

6. `evalq()` 是做什么的？对于上面使用了 `eval()` 和 `quote()` 的例子，使用 `evalq()` 来减少输入的字符数量。

13.3 作用域问题

我们的 `subset2()` 看起来可以工作。但是，因为我们使用的是表达式而不是表达式的值，所以我们需要进行更广泛的测试。例如，下面对 `subset2()` 的调用，应该都返回相同的值，因为它们之间的唯一区别只是变量的名字：

```
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)
```

到底是哪里出了错？你可以从我选择的变量名中得到提示：它们都是 `subset2()` 内部定义的变量名。如果 `eval()` 在数据框内找不到变量(第二个参数)，那么它将在 `subset2()` 所处的环境中寻找。这显然不是我们想要的，所以如果 `eval()` 不能在数据框中找到变量，那么我们需要通过一些方法来告诉 `eval()` 应该在哪里寻找变量。

关键是 `eval()` 函数的第三个参数：`enclos`。这使我们能够为没有父环境的对象(比如列表和数据框)，指定一个父(或封闭)环境。如果在 `env` 中没有找到绑定(名字-值)，那么 `eval()` 将在 `enclos` 下看看，然后在 `enclos` 的父环境中查找。如果 `env`

是实际的环境，那么 `enclos` 会被忽略。我们想要在 `subset2()` 被调用时所处的环境中寻找 `x`。在 R 语言的术语中，这称为父框架(parent frame)，可以使用 `parent.frame()` 来访问。这是动态作用域的一种实例 (http://en.wikipedia.org/wiki/Scope_%28programming%29#Dynamic_scoping): 变量值来自于函数被调用时的位置，而不是变量被定义时的位置。做了这个修改，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` 只是把列表或者数据框转换成环境的一种捷径。我们可以使用 `list2env()` 得到同样的效果。它把列表转化为一个环境，并且明确地设置了一个父环境：

```
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
```

13.3.1 练习

1. `plyr::arrange()`的作用类似于 `subset()`，但是它并不选择行，而是对它们进行重新排序。它是如何工作的？`substitute(order(...))`是做什么的？创建一个函数只做这件事的函数，并进行试验。
2. `transform()`是做什么的？阅读它的文档。它是如何工作的？阅读 `transform.data.frame()`的源代码。`substitute(list(...))`是做什么的？
3. `plyr::mutate()`类似于 `transform()`，但是它会按照顺序进行变换 (transformation)，所以变换可以引用(refer to)刚刚创建的列：

```
df <- data.frame(x = 1:5)  
transform(df, x2 = x * x, x3 = x2 * x)  
plyr::mutate(df, x2 = x * x, x3 = x2 * x)
```

`plyr::mutate()`是如何工作的呢？`plyr::mutate()`和 `transform()`的主要区别是什么？

4. `with()`是做什么的？它是如何工作的？阅读 `with.default()`的源代码。
`within()`是做什么的？它是如何工作的？阅读 `within.data.frame()`的源代码。
为什么它的代码要比 `with()`复杂得多？

13.4 从另一个函数进行调用

通常，当函数是由用户直接调用的时候，编程语言层面的计算是最有用的，而当它们是由其它函数进行调用的时候，则没有那么大用处。虽然 `subset()`可以减少打字的输入，但是在非交互式场景下，实际上很难使用。例如，假设我们想创建一个函数，它对数据的行的一个子集进行随机排序。一种不错的方法是组合两种函数，一种用于选择子集，另一种用于排序。让我们试一试：

```
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))  
}
```

但是，这样并不可行：

```
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)
```

出了什么事呢？为了找出问题，让我们调试(debug())subset2()，并逐行运行代码：

```
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
```

你能发现问题是什么吗？`condition_call` 包含表达式 `condition`。所以，当我们计算 `condition_call` 的时候，也计算了 `condition`，它的值是 `a >= 4`。但是，这是不能计算的，因为在父环境中没有称为 `a` 的对象。但是，如果在全局环境中设置了 `a`，会发生更令人迷惑的事情：

```
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
```

这是一个例子，它表明了在设计函数时的矛盾：函数是在交互式环境中使用的，还是在编程时使用的？一个使用了 `substitute()` 的函数也许会减少打字的数量，但是这也使得它很难被另一个函数调用。

作为一名开发人员，你应该提供一个后门(escape hatch)函数：它是函数的另一个版本，并且使用了标准的计算方式。在这种情况下，我们可以编写另一个版本的 `subset2()`，它需要一个已经引用的(quoted)表达式：

```
subset2_q <- function(x, condition) {
  r <- eval(condition, x, parent.frame())
  x[r,]
}
```

在这里，我使用后缀 `_q` 表明需要一个引用表达式(quoted expression)。由于大多数用户并不需要这个函数，所以这个名字可能有点长。我们可以使用 `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
```

基本的 R 函数倾向于使用一种不同的后门。它们通常有一个参数来关闭非标准计算。例如，`require()` 的 `character.only = TRUE`。我不认为使用一个参数来改变另一个参数的行为是个好主意，因为它使函数调用变得更难理解。

13.4.1 练习

1. 下面的 R 函数都使用了非标准计算。对于每一个函数，描述它是怎样使用非标准计算的，阅读文档来确定它的后门是什么。

```
rm()
library()和 require()
substitute()
data()
data.frame()
```

2. 基本函数 `match.fun()`、`page()` 和 `ls()` 都试图自动确定你想要标准还是非标准的计算。它们每个都使用一种不同的方法。指出每种方法的本质，然后进行比较。
3. 通过把 `plyr::mutate()` 分裂成两个函数，为 `plyr::mutate()` 添加一个后门。一个函数应该捕获未计算的输入。另一个应该传入一个数据框以及表达式列表，并执行计算。

4. `ggplot2::aes()`的后门是什么？`plyr::()`呢？它们有什么共同点？它们的不同点各有什么优缺点？
5. 我提出的 `subset2_q()`函数，是一种简化版本。为什么下面的版本更好？

```
subset2_q <- function(x, cond, env = parent.frame()) {  
  r <- eval(cond, x, env)  
  x[r,]  
}
```

使用这个改进版本重写 `subset2()`和 `subscramble()`。

13.5 substitute()

大多数使用非标准计算的函数，都会提供一个后门。但是如果你想调用的函数没有后门该怎么办呢？例如，假设你希望通过给出两个变量名来创建一个 lattice 图形：

```
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
```

我们可能会用 `substitute()`来达到另一种目的：修改一个表达式。不幸的是，`substitute()`有一个特性，使得交互式地修改调用不是很容易。当在全局环境中运行时，它从来不会进行替换：事实上，在这种情况下它的行为与 `quote()`一样：

```
a <- 1  
b <- 2  
substitute(a + b + z)  
#> a + b + z
```

然而，如果你在函数内部运行它，那么 `substitute()` 会进行替换，并保留其它的内容：

```
f <- function() {  
  a <- 1  
  b <- 2  
  substitute(a + b + z)  
}  
f()  
#> 1 + 2 + z
```

为了使 `substitute()` 的试验更简单，`pryr` 包提供了 `subs()` 函数。除了名字更短和作用于全局环境中以外，它的工作方式与 `substitute()` 完全相同。这两种特性使得实验变得更简单：

```
a <- 1  
b <- 2  
subs(a + b + z)  
#> 1 + 2 + z
```

第二个参数(`subs()`和 `substitute()`)可以覆盖当前的环境，并通过一个"名字-值"对(key-value pair)的列表，来提供其它的选择。下面的示例使用了这种技术，显示了在替换字符串、变量名或函数调用时发生的变化：

```
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
```


请记住，在 R 语言中的每一个动作都是一个函数调用，所以我们可以把 `+` 替换成另一个函数：

```
subs(a + b, list("+ " = quote(f)))  
#> f(a, b)  
subs(a + b, list("+ " = quote(`*`)))  
#> a * b
```

你也可以写出没有意义的代码：

```
subs(y <- y + 1, list(y = 1))  
#> 1 <- 1 + 1
```

在形式上，替换是通过检查表达式中的所有名字来发生的。如果名字是指：

1. 一个普通的变量，则该变量会被值所取代。
2. 一个承诺(一个函数参数)，则取而代之的是与承诺相关的表达式。
3. `...`，会被替换成`...`的内容。

否则，名字会保持不变。我们可以使用它来创建对 `xyplot()` 正确调用：

```
x <- quote(mpg)  
y <- quote(displ)  
subs(xyplot(x ~ y, data = mtcars))  
#> xyplot(mpg ~ displ, data = mtcars)
```

在函数内部甚至会 simpler，因为我们不需要显式地引用变量 `x` 和 `y`(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)
```

如果我们把...包括在对 `substitute()` 的调用之中，那么我们可以添加额外的调用参数：

```
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")
```

要创建这个图形，我们可以使用 `eval()` 来计算这个调用。

13.5.1 为 `substitute` 添加一个后门函数

`substitute()` 本身是一个使用了非标准计算的函数，并且没有后门函数。这意味着，如果表达式保存在变量中，那么我们不能使用 `substitute()`：

```
x <- quote(a + b)
substitute(x, list(a = 1, b = 2))
#> x
```

虽然 `substitute()` 没有内置的后门函数，但是我们可以使用这个函数本身创建一个：

```
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()`的实现很短，但是很复杂。让我们逐步分析上面的例子：

`substitute_q(x, list(a = 1, b = 2))`。这里有些巧妙，因为 `substitute()` 使用了非标准计算，所以我们不能用像通常那样从内到外，通过剥离括号的方式来进行分析。

1. 首先，`substitute(substitute(y, env), list(y = x))`被计算。表达式 `substitute(y, env)` 被捕获，并且 `y` 被替换成 `x` 的值。因为我们把 `x` 放在了列表内，所以它会被计算，并且替换规则会将 `y` 替换成它的值。这样产生了一个表达式 `substitute(a + b, env)`。
2. 接下来，我们计算在当前函数中的表达式。`substitute()` 计算它的第一个参数，并且在 `env` 中寻找"名字-值"对(key-value pair)。在这里，它根据 `list(a = 1, b = 2)` 来计算 `x`。因为这些都是值(而不是承诺)，所以结果是 `1 + 2`。`pryr` 包提供了更加严格的 `substitute_q()` 版本。

13.5.2 捕获未计算的...

另一个有用的技巧是捕获...中所有未计算的表达式。基本 R 函数使用多种方式来做到这一点，但是有一种技术在各种各样的情况下都可以应用：

```
dots <- function(...) {  
  eval(substitute(alist(...)))  
}
```

这里使用了 `alist()` 函数，它只是简单地捕获所有的参数而已。这个函数与 `pryr::dots()` 是一样的。`pryr` 包还提供了 `pryr::named_dots()`，它通过使用逆解析的表达式作为默认的名称，来确保所有的参数都被命名了(就像 `data.frame()`)。

13.5.3 练习

1. 对下面所有的表达式对(expression pair)，使用 `subs()` 把 LHS(Left-Hand Side，等式的左边)转化为：

```
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
```

2. 对下面所有的表达式对，描述一下为什么不能使用 `subs()` 从一个转化为另一个。

```
a + b + c -> a + b * c
f(a, b) -> f(a, b, c)
f(a, b, c) -> f(a, b)
```

3. `pryr::named_dots()` 是怎样工作的？请阅读它的源代码。

13.6 非标准计算的缺点

非标准计算的最大缺点是，使用了非标准计算的函数，不再是引用透明 (referentially transparent) 的

([http://en.wikipedia.org/wiki/Referential_transparency_\(computer_science\)](http://en.wikipedia.org/wiki/Referential_transparency_(computer_science)))。如果你可以把一个函数的参数替换成它们的值，并且它的行为并没有改变，那么这个函数就是引用透明的。例如，如果一个函数 `f()` 是引用透明的，并且 `x` 和 `y` 都是 `10`，那么 `f(x)`、`f(y)` 和 `f(10)` 都将返回相同的结果。引用透明的代码更容易理解，因为对象的名字并不重要，也因为你总是可以从最内部的括号开始，由内向外进行处理。有许多重要的函数，由于其本身的性质，而导致它们不是引用透明的。比如赋值运算符。你不能把 `a <- 1` 中的 `a` 替换成它的值而得到同样的行为。这是人们通常在函数的顶层进行赋值操作的原因之一。（译者注：赋值操作通常不会内嵌到其它语句当中，否则语句会难以理解。）很难理解这样的代码：

```
a <- 1
b <- 2
if ((b <- a + 1) > (a <- b - 1)) {
```

```
b <- b + 2  
}
```

使用**非标准计算**使得函数不是**引用透明**的。这使得精确地预测函数的输出变得复杂得多。所以，只有**收益很大**的时候，使用**非标准计算**才是值得的。例如，

`library()`和 `require()`可以使用引号或者不使用引号进行调用，因为在内部，它们使用了 `deparse(substitute(x))`以及其它一些技巧。这意味着，这两行做同样的事：

```
library(ggplot2)  
library("ggplot2")
```

如果变量 `ggplot2` 已经关联了一个值，那么情况就开始变得复杂了。到底会加载什么包呢？

```
ggplot2 <- "plyr"  
library(ggplot2)
```

有许多其它的 R 函数以这种方式工作，如 `ls()`、`rm()`、`data()`、`demo()`、`example()`以及 `vignette()`。对我来说，减少了两个按键的输入(译者注：即两个引号)，并不值得损失**引用透明性**，所以我不推荐你为了这样的目的来使用**非标准计算**。

一种值得使用**非标准计算**的情形是 `data.frame()`。如果没有显式地提供列名，那么它将自动使用输入的**变量名**作为数据框的**列名**：

```
x <- 10  
y <- "a"  
df <- data.frame(x, y)  
names(df)  
#> [1] "x" "y"
```

我认为这是值得的，因为当你通过现有的变量，来创建一个数据框时，它可以消除大量的冗余，而这是一种很常见的场景。更重要的是，如果需要的话，通过为每个列提供名称，那么可以很容易地覆盖此行为。

非标准计算可以让你编写出非常强大的函数。然而，它们更难理解以及更难用于编程。除了总是应该提供一个**后门**以外，在一个新的领域中使用**非标准计算**之前，请仔细考虑一下它的开销与收益。

13.6.1 练习

1. 以下函数是做什么的？它的**后门**函数是什么？你认为在这里使用**非标准计算**合适吗？

```
nl <- function(...) {  
  dots <- named_dots(...)  
  lapply(dots, eval, parent.frame())  
}
```

2. 你可以使用由 `~` 创建的**公式**(formula)，而不是**承诺**(promise)，来显式地捕获一个**表达式**以及它的**环境**。**显式引用**的优点和缺点是什么？它是如何影响**引用透明性**的？
3. 阅读 <http://developer.r-project.org/nonstandard-eval.pdf> 中描述的标准的**非标准计算**规则。

14 表达式

在第 13 章，你学习了在 R 语言中**访问**和**计算**表达式。在这一章中，你将学习如何使用代码来操纵这些表达式。你将学习**元编程**——怎样使用其它程序来创建程序！

本章概要

14.1 节开始深入了解**表达式**的结构。你将了解**表达式**的四个组件：**常量**、**名字**、**调用**和**成对列表**(pairlist)。

14.2 节更进一步的讨论**名字**的细节。

14.3 节提供了**调用**的更多细节。

14.4 节将讨论一些在基本 R 语言中**调用**的常见使用方式。

14.5 节完成了讨论**表达式**的四个主要组成部分，并展示如何从它们的组成部分来创建函数。

14.6 节讨论如何在**表达式**和**文本**之间进行来回转换。

14.7 节总结了这一章，并结合你所学的一切，来编写可以**修改**和**计算**任意 R 语言代码的函数。

前提条件

本章我们需要 **pryr** 包。可以使用 `install.packages("pryr")` 进行安装。

14.1 表达式的结构

编程语言层面的计算(compute on the language)，我们首先需要了解语言的结构。这将需要一些新的**函数**、一些新的**工具**和一些新的**思考 R 语言代码的方式**。你需要理解的第一件事是**操作**(operation)和**结果**(result)之间的区别：

```
x <- 4
y <- x * 10
y
#> [1] 40
```

我们要区别“把 `x` 乘以 `10` 并把结果赋予给 `y`”的动作和实际结果(`40`)。正如我们在前面一章所看到的那样, 我们可以使用 `quote()` 来捕获动作:

```
z <- quote(y <- x * 10)
z
#> y <- x * 10
```

`quote()` 返回一个表达式, 表达式代表可以由 R 语言来执行的一个动作的对象。(不幸的是, `expression()` 并不返回这个意义上的表达式。相反, 它返回的东西更像是一个表达式列表。参见 14.6 节获取详情)。

表达式也被称为抽象语法树(abstract syntax tree, AST), 因为它代表了代码的分层树状结构。我们将使用 `pryr::ast()` 来更加清楚地认识这一点:

```
ast(y <- x * 10)
#> \- 0
#> \- `<-
#> \- `y
#> \- 0
#> \- `*
#> \- `x
#> \- 10
```

一个表达式有四种可能的组件: 常量(constant)、名字(name)、调用(call)和成对列表(pairlist)。

常量是长度为 1 的原子向量, 比如 `"a"` 或 `10`。 `ast()` 把显示它们成:


```
ast("a")
#> \- "a"
ast(1)
#> \- 1
ast(1L)
#> \- 1L
ast(TRUE)
#> \- TRUE
```

用 `quote()` 引用一个常量将保持不变:

```
identical(1, quote(1))
#> [1] TRUE
identical("test", quote("test"))
#> [1] TRUE
```

名字或者符号(symbol), 代表一个对象的名字, 而不是它的值。 `ast()` 在名字前加上一个重音符前缀。

```
ast(x)
#> \- `x`
ast(mean)
#> \- `mean`
ast(an unusual name)
#> \- `an unusual name`
```

调用表示调用函数的动作。就像列表一样, 调用是递归的: 它们可以包含常量、名字、成对列表和其它调用。

`ast()` 先打印(), 然后列出子节点。第一个子节点是被调用的函数, 剩下的子节点是该函数的参数。

```
ast(f())  
#> \- 0  
#> \- `f  
ast(f(1, 2))  
#> \- 0  
#> \- `f  
#> \- 1  
#> \- 2  
ast(f(a, b))  
#> \- 0  
#> \- `f  
#> \- `a  
#> \- `b  
ast(f(g(), h(1, a)))  
#> \- 0  
#> \- `f  
#> \- 0  
#> \- `g  
#> \- 0  
#> \- `h  
#> \- 1  
#> \- `a
```

正如 6.3 节中提到的那样，就算看起来不像函数调用，仍然会有这种层次结构：

```
ast(a + b)  
#> \- 0  
#> \- `+  
#> \- `a  
#> \- `b
```

```
ast(if (x > 1) x else 1/x)
```

```
#> \- 0
```

```
#> \- `if
```

```
#> \- 0
```

```
#> \- `>
```

```
#> \- `x
```

```
#> \- 1
```

```
#> \- `x
```

```
#> \- 0
```

```
#> \- `/
```

```
#> \- 1
```

```
#> \- `x
```

成对列表，是“有点的(dotted)成对列表”的简称，是从 R 语言的历史版本中遗留下来的。它们只能用在—个地方：函数的形式参数。`ast()`在成对列表的顶层打印[]。和调用—样，成对列表也是递归的，也可以包含常量、名字和调用。

```
ast(function(x = 1, y) x)
```

```
#> \- 0
```

```
#> \- `function
```

```
#> \- []
```

```
#> \ x = 1
```

```
#> \ y = `MISSING
```

```
#> \- `x
```

```
#> \- <srcref>
```

```
ast(function(x = 1, y = x * 2) {x / y})
```

```
#> \- 0
```

```
#> \- `function
```

```
#> \- []
```

```
#> \ x = 1
```

```
#> \ y=0
#> \- `*
#> \- `x
#> \- 2
#> \- 0
#> \- `{
#> \- 0
#> \- `/
#> \- `x
#> \- `y
#> \- <srcref>
```

注意，`str()`在描述对象的时候不遵循下面的命名惯例。相反，它把名字描述成符号(symbol)以及把调用描述成语言对象(language object):

```
str(quote(a))
#> symbol a
str(quote(a + b))
#> language a + b
```

使用低层的函数，可以创建包含对象的调用树(call tree)，这些对象可以不是常量、名字、调用和成对列表这些。下面的示例使用了 `substitute()` 将一个数据框插入调用树。然而，这是一个坏主意，因为打印对象会不正确：被打印的调用看起来应该返回"list"，但是当计算之后，它却返回了"data.frame"。

```
class_df <- substitute(class(df), list(df = data.frame(x = 10)))
class_df
#> class(list(x = 10))
eval(class_df)
#> [1] "data.frame"
```

这四个组件一起定义了所有 R 代码的结构。以下章节将更加详细地解释它们。

14.1.1 练习

1. 目前，没有基本函数可以检查一个元素是不是一个表达式的有效组成部分(比如它是常数、名字、调用或成对列表)。实现一个这样的函数。
2. `pryr::ast()` 使用了非标准计算。它的后门(`escape hatch`)函数是什么？
3. 带有多条 `else` 条件的 `if` 语句的调用树是什么样子的？
4. 比较 `ast(x + y %>% z)` 和 `ast(x ? y %>% z)`。关于自定义中缀函数的优先级，它们告诉你了什么？
5. 为什么表达式不能包含长度大于 1 的原子向量？六种原子向量的哪一个不能出现在表达式中？为什么？

14.2 名字

通常，我们使用 `quote()` 来捕获名字。你还可以使用 `as.name()` 将一个字符串转化为名字。然而，这仅当你的函数接受字符串作为输入的时候，是最有用的。否则，它比使用 `quote()` 还要打更多的字。(你可以使用 `is.name()` 来测试一个对象是不是一个名字。)

```
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()` 是相同的。)

无效的名字会自动被重音符包围：

```
as.name("a b")
```

```
#> `a b`
```

```
as.name("if")
```

```
#> `if`
```

有一种特别的名称，需要一点额外讨论：空名称(empty name)。它是用来表示缺失参数的。这个对象的行为很奇怪。你不能将它绑定到一个变量中。如果你这样做，那么它会触发一个关于缺失参数的错误。如果你想通过编程方式，创建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
```

```
# 不可行！
```

```
is.name(missing_arg)
```

```
#> Error: argument "missing_arg" is missing, with no default
```

想要在需要的时候显式地创建它，可以使用命名参数来调用 `quote()`：

```
quote(expr =)
```

14.2.1 练习

1. 你可以使用 `formals()` 来存取一个函数的参数。使用 `formals()` 来修改以下函数，使得 `x` 的在默认状态下是缺失的，`y` 的默认值是 `10`。

```
g <- function(x = 20, y) {  
  x + y  
}
```

2. 使用 `as.name()` 和 `eval()` 编写一个相当于 `get()` 的函数。使用 `s.name()`、`substitute()` 和 `eval()` 编写一个相当于 `assign()` 的函数。[不要关心选择环境的多种方式；假定用户已经显式地提供了环境]。

14.3 调用

调用(`call`)非常类似于列表。它有 `length`、`[[`和`[`方法，它也是递归的，因为调用可以包含其它调用。调用的第一个元素是被调用的函数。它通常是一个函数的名字：

```
x <- quote(read.csv("important.csv", row.names = FALSE))  
x[[1]]  
#> read.csv  
is.name(x[[1]])  
#> [1] TRUE
```

但是也可以是另一个调用：

```
y <- quote(add(10)(20))  
y[[1]]  
#> add(10)  
is.call(y[[1]])  
#> [1] TRUE
```

剩下的元素是**参数**。它们可以通过**名字**或**位置**来提取。

```
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 1，可以得到参数的个数：

```
length(x) - 1
#> [1] 2
```

14.3.1 修改调用

你可以使用标准替换操作符**\$<-**和**[[<-**来添加、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 = ",")
```


调用还支持[]方法。但使用它时要小心。删除了第一个元素是不可能创建一个有用的调用的。

```
x[-3] # 删除第二个参数
#> read.csv("important.csv")
x[-1] # 删除函数名——但是它仍然是一个调用！
#> "important.csv"(row.names = FALSE)
x
#> read.csv("important.csv", row.names = FALSE)
```

如果你想要一个未计算的(unevaluated)参数列表(表达式)，那么可以使用显式地强制转换：

```
# 未计算的参数列表
as.list(x[-1])

#> [[1]]
#> [1] "important.csv"
#>
#> $row.names
#> [1] FALSE
```

一般来说，因为 R 语言的函数调用语义非常灵活，所以通过位置来获取参数或者设置参数都是很危险的。例如，尽管在每个位置的值都是不同的，下面的三个调用都有同样的效果：

```
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` 提供了 `standardise_call()`。它使用基本的 `match.call()` 函数把所有基于位置的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 通过组件来创建调用

要通过组件来创建新的调用，你可以使用 `call()` 或者 `as.call()`。`call()` 的第一个参数是一个字符串，它给出了函数名。其它的参数是表达式，表示调用的参数。

```
call(":", 1, 10)
#> 1:10
call("mean", quote(1:10), na.rm = TRUE)
#> mean(1:10, na.rm = TRUE)
```

`as.call()` 是 `call()` 的一个小的变体，它把单个列表作为输入。第一个元素是一个名字或调用。随后的元素是参数。

```
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. 下面两个调用看起来相同，但是实际上是不同的：

```
(a <- call("mean", 1:10))
#> mean(1:10)
(b <- call("mean", quote(1:10)))
#> mean(1:10)
```

```
identical(a, b)
```

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

不同在哪里？ 你更应该使用哪一个？

2. 实现一个纯 R 版本的 `do.call()`。
3. 使用 `c()` 连接调用和表达式可以创建列表。实现 `concat()`，使得下面的代码可以用于连接调用和额外参数。

```
concat(quote(f), a = 1, b = quote(mean(a)))
```

```
#> f(a = 1, b = mean(a))
```

4. 由于列表(`list()`)不属于表达式，所以我们可以创建一个更方便的调用构造器 (call constructor)，它自动把列表合并到参数中。实现 `make_call()`，使得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<-`是怎样工作的？ 如果使用 `call()` 调用它，应该怎么做？
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. 对于下面的调用，`standardise_call()` 工作得不好。为什么？

```
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)
```

8. 阅读 `pryr::modify_call()` 的文档。你认为它是怎样工作的？阅读它的源代码。
9. 使用 `ast()` 并进行实验，指出 `if()` 调用的三个参数。哪些组件是需要的？`for()` 和 `while()` 的参数是什么？

14.4 捕获当前调用

许多基本 R 函数使用当前调用(current call)：引发当前函数运行的表达式。有两种方法来捕获当前调用：

1. `sys.call()` 捕获用户输入了什么。
2. `match.call()` 创建一个仅使用命名参数的调用。

这就好像，对 `sys.call()` 的结果，自动调用了 `pryr::standardise_call()`。下面的例子说明了两者的区别：

```
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)
```

建模函数(modelling function)经常使用 `match.call()` 来捕获用于创建模型的调用。这使得它可以更新(`update()`)模型，重新拟合修改了一些原始参数的模型。这里有一个 `update()` 的例子：

```
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()`是如何工作的呢？我们可以使用一些 `pryr` 包中的工具对它进行重写，以便专注于它的算法本质。

```
update_call <- function (object, formula, ...) {
  call <- object$call
  # 使用 update.formula 来处理像 . ~ . 这样的公式
  if (!missing(formula.)) {
    call$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()` 有一个 `evaluate` 参数，它控制着函数是返回调用还是返回结果。但是，原则上，我认为这样才是更好的：一个函数只返回一种类型的对象，而不是根据参数来返回不同的类型。

对 `update()` 的重写还使我们可以修复它的一个小错误：当我们真正想要的是，在模型最初在公式(`formula`)中进行拟合时的环境中，进行重新计算(re-evaluate)调用的时候，它却在全局环境中重新计算(re-evaluate)了调用。

```
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
```

这是需要记住的重要原则：如果你想要重新运行 `match.call()` 捕获的代码，那么你还

需要捕获代码计算时所处的环境，这个环境通常是 `parent.frame()`。但是，缺点是捕获环境也意味着要捕获那个环境内任何大型对象，会阻止它们的内存被释放。这个话题在 18.2 节中将进行更详细地探讨。

一些基本的 R 函数使用了 `match.call()`，虽然并不是必需的。例如，`write.csv()` 捕获对 `write.csv()` 的调用，和并改为调用 `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)  
}
```

为了进行修复，我们可以使用常规的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,
...)
}
```

这是更容易理解的：它只是使用不同的默认值来调用 `write.table()`。这也修复了在原始 `write.csv()` 中的一个微妙的错误：`write.csv(mtcars, row = FALSE)` 抛出了一个错误，但 `write.csv(mtcars, row.names = FALSE)` 却没有。这里的教训是，用最简单的工具来解决问题总是更好的。

14.4.1 练习

1. 比较 `update_model()` 与 `update.default()`。
2. 为什么 `write.csv(mtcars, "mtcars.csv", row = FALSE)` 不能工作？原作者忘记了参数匹配的什么属性？
3. 使用 R 代码代替 C 代码重写 `update.formula()`。
4. 有时候，有必要暴露这样的函数：它调用了一个函数，然后这个函数又调用了当前函数。(即上上级别、祖父级别的调用函数，而不是父级别的调用函数)。怎样使用 `sys.call()` 或 `match.call()` 来找到这个函数？

14.5 成对列表

成对列表(Pairlists)是以前的 R 版本中遗留下来的。它们的行为与列表相似，但是内部表示是不同的(表示为链表 linked list，而不是向量)。除了函数参数以外，在其它所有方面，成对列表都已经被列表取代了。

唯一需要关心成对列表和列表之间的区别的地方，是当你要手工构造函数的時候。例如，以下函数允许你通过函数组件来构建一个函数：一个形式参数列表、

一个函数体和一个环境。该函数使用了 `as.pairlist()`，以确保 `function()` 拥有所需的 `args` 成对列表参数。

```
make_function <- function(args, body, env = parent.frame()) {  
  args <- as.pairlist(args)  
  eval(call("function", args, body), env)  
}
```

这个函数也可以在 `pryr` 中得到，并进行了一些额外的参数检查。`make_function()` 与 `alist()` 联合使用是最好的，`alist()` 是参数列表函数。`alist()` 并不计算它的参数，所以 `alist(x = a)` 是 `list(x = quote(a))` 的简化。

```
add <- make_function(alist(a = 1, b = 2), quote(a + b))  
add(1)  
#> [1] 3  
add(1, 2)  
#> [1] 3  
# 为了让参数不带有默认值，你需要使用显式的=  
make_function(alist(a = , b = a), quote(a + b))  
#> function (a, b = a)  
#> a + b  
# 为了接受`...`作为参数，要把它放在=的左边  
make_function(alist(a = , b = , ... =), quote(a + b))  
#> function (a, b, ...)  
#> a + b
```

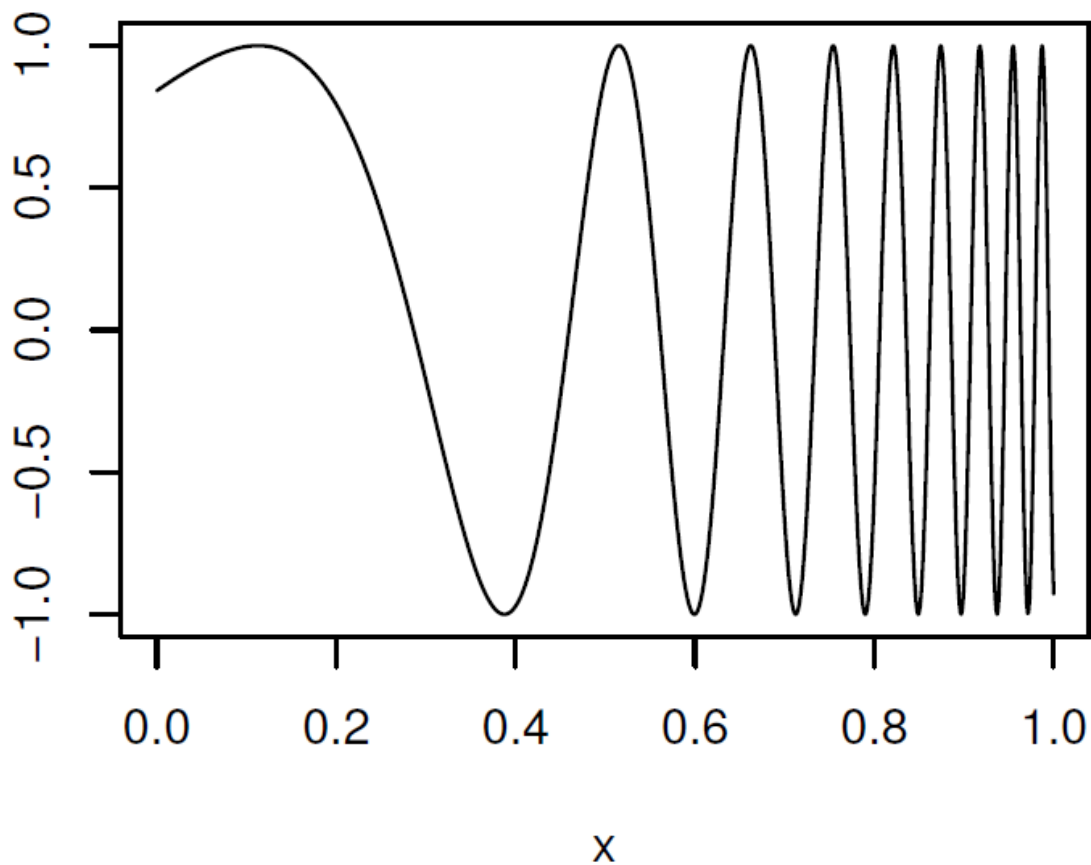
`make_function()` 有一个优势，它使用闭包来构造函数：使用了它，你可以轻松地阅读源代码。例如：

```
adder <- function(x) {  
  make_function(alist(y =), substitute({x + y}), parent.frame())  
}
```

```
}  
adder(10)  
#> function (y)  
#> {  
#> 10 + y  
#> }
```

`make_function()`的一种有用的应用是在 `curve()` 这类函数里面。`curve()` 允许你绘制数学函数曲线，而不用创建显式的 R 函数：

```
curve(sin(exp(4 * x)), n = 1000)
```



这里的 `x` 是一个代词(pronoun)。`x` 并不代表一个具体的值，而是一个占位符，随着绘图区间的变化而不同。一种实现 `curve()` 的方法是使用 `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**首语重复法**(anaphoric)函数

([http://en.wikipedia.org/wiki/Anaphora_\(linguistics\)](http://en.wikipedia.org/wiki/Anaphora_(linguistics)))。 它们被用于

Arc(<http://www.arcfn.com/doc/anaphoric.html>)(一个 lisp 语言)、

Perl(http://www.perlmonks.org/index.pl?node_id=666047)和

Clojure(<http://amalloy.hubpages.com/hub/Unhygenic-anaphoric-Clojure-macros-for-fun-and-profit>)。

14.5.1 练习

1. `alist(a)`与 `alist(a =)`有什么不同? 考虑一下输入和输出。
2. 阅读 `pryr::partial()`的文档和源代码。它是做什么的? 它是如何工作的? 阅读 `pryr::unenclose()`的文档和源代码。它是做什么的? 它是如何工作的?
3. `curve()`的实际实现看起来更像

```
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)))  
}
```

这种方法与上面定义的 `curve2()` 有什么区别？

14.6 解析与逆解析

有时，代码表示成字符串，而不是表达式。你可以使用 `parse()` 将字符串转换为表达式。与解析(`parse()`)相反的是逆解析(`deparse()`)：给它传入一个字符向量，它返回一个表达式对象。`parse()` 的主要用途是解析磁盘上的代码文件，所以它的第一个参数是一个文件路径。注意，如果你的代码是一个字符向量表示的，那么你需要使用 `text` 参数：

```
z <- quote(y <- x * 10)  
deparse(z)  
#> [1] "y <- x * 10"  
parse(text = deparse(z))  
#> expression(y <- x * 10)
```

因为文件中可能有许多顶层的调用，所以 `parse()` 并不是仅仅返回一个表达式。相反，它返回一个 `expression` 对象，它实际上是一个表达式列表：

```
exp <- parse(text = c("  
x <- 4  
x  
5  
"))  
length(exp)  
#> [1] 3  
typeof(exp)  
#> [1] "expression"
```

```
exp[[1]]  
#> x <- 4  
exp[[2]]  
#> x
```

你可以使用 `expression()` 手工创建表达式对象，但是我不推荐这么做。如果你已经知道了如何使用表达式，那么不需要了解这个深奥的数据结构。

使用 `parse()` 和 `eval()`，可以编写简单版本的 `source()` 函数。我们从磁盘读取文件，解析(`parse()`)它，然后在指定的环境中计算(`eval()`)每个组件。这个版本默认在一个新环境中进行计算，所以它不会影响现有的对象。`source()` 隐式地返回文件中最后一个表达式的结果，所以 `simple_source()` 也会做同样的事情。

```
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))  
}
```

真正的 `source()` 更加复杂，因为它可以回显(echo)输入和输出，还有许多额外的设置来控制行为。

14.6.1 练习

1. `quote()`和 `expression()`的区别是什么？
2. 阅读 `deparse()`的帮助文档，并构造一个调用，使得 `deparse()`和 `parse()`的操作是不对称的。
3. 比较 `source()`和 `sys.source()`。
4. 修改 `simple_source()`，使它返回每个表达式的结果，而不仅仅是最后一个结果。
5. 由 `simple_source()`生成的代码缺少引用来源(source reference)。阅读 `sys.source()`的源代码和 `srcfilecopy()`的帮助文档，然后修改 `simple_source()`以保存引用来源。你可以使用 `source()`加载包含注释的函数，来测试你的代码。如果成功，那么当你查看函数的时候，你将能看到注释，而不仅仅是源代码。

14.7 使用递归函数遍历抽象语法树

使用 `substitute()`或 `pryr::modify_call()`可以很容易地修改单个调用。而对于更复杂的任务，我们需要直接处理抽象语法树。

基本的 `codetools` 包提供了一些有用的例子，来告诉我们如何能做到这一点：

`findGlobals()`可以定位(`locate`)函数使用的所有全局变量。如果你想检查你的函数是否意外地依赖了父环境中定义的变量，那么它是有用的。

`checkUsage()`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等等。

要编写像 `findGlobals()`和 `checkUsage()`这样的函数，我们需要一种新的工具。因为表达式拥有树状结构，所以使用递归函数是很自然的选择。这样做的关键是获

取正确的**递归结构**。也就是说，你要知道递归的出口(译者注：即递归终止的条件)，并且找出如何组合递归结果。(译者注：即上一级和下一级递归之间的关系，递归子问题) 对于**调用**来说，有两种**基本情况**(原子向量和名字)以及两种**递归情况**(调用和成对列表)。也就是说，操作表达式的函数看起来像这样：

```
recurse_call <- function(x) {  
  if (is.atomic(x)) {  
    # 返回一个值  
  } else if (is.name(x)) {  
    # 返回一个值  
  } else if (is.call(x)) {  
    # 递归地调用 recurse_call  
  } else if (is.pairlist(x)) {  
    # 递归地调用 recurse_call  
  } else {  
    # 用户提供了错误的输入  
    stop("Don't know how to handle type ", typeof(x),  
         call. = FALSE)  
  }  
}
```

14.7.1 寻找 F 和 T

我们将从简单的函数开始，该函数确定一个函数是否使用了逻辑缩写 **T** 和 **F**。使用 **T** 和 **F** 通常被认为是较差的编码实践，**R CMD** 的检查也会对这种情况发出警告。让我们首先比较 **T** 与 **TRUE** 的抽象语法树：

```
ast(TRUE)  
#> \- TRUE
```

```
ast(T)
#> \- `T
```

TRUE 被解析成长度为 **1** 的逻辑向量，而 **T** 被解析成了一个名字。这告诉了我们如何为递归函数写出递归出口：由于原子向量永远不会成为逻辑缩写，而名字则有可能，因此我们需要测试 **T** 和 **F** 两种情况。递归情况可以组合在一起，因为在这两种情况下，它们都做同样的事情：对于对象中的每一个元素，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 找到所有由赋值创建的变量

`logical_abbr()` 非常简单:它只返回单一的 **TRUE** 或 **FALSE**。下一个任务——列出所有由赋值创建的变量——更复杂一些。我们从简单的开始,然后逐步使函数变得更加严密。同样,我们从观察赋值操作的抽象语法树开始:

```
ast(x <- 10)  
#> \- 0  
#> \- `<-  
#> \- `x  
#> \- 10
```

赋值是一个调用,它的第一个元素是名字 `<-`,第二个元素是名字被赋予的对象,第三个元素是要赋的值。这使得基本情况很简单:常量和名字不会创建赋值,所以它们返回 **NULL**。递归的情形也不是太难。我们对成对列表以及除了 `<-` 以外的函数调用,使用 `lapply()`。

```
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
```

在这些简单的情况下，这个函数可以工作，但是输出过于详细，包括了一些无关紧要的 **NULL** 值。我们让它保持简单，并且使用字符向量，而不是返回列表。我们还将使用两个稍微复杂一点例子对其进行测试：

```
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"

```

这样更好，但是我们还有两个问题：处理重复的名字，和，忽略在其它赋值操作内部的赋值操作。修复第一个问题很容易。我们需要用 `unique()` 来包装递归情况，以删除重复的赋值操作。解决第二个问题有点棘手。当调用是 `<-` 时，我们也需要递归。 `find_assign3()` 实现了这两种策略：

```

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"
```

我们还需要测试子赋值(subassignment):

```
find_assign3(quote({  
  l <- list()  
  l$a <- 5
```

```
names(l) <- "b"

)))

#> [1] "l" "$" "a" "names"
```

我们只需要**对象本身**的赋值操作，而不是**修改对象属性**的赋值操作。绘制**被引号引用的对象**的树，可以帮我们看看测试条件是什么。`<-`调用的第二个元素应该是一个名字，而不是另一个调用。

```
ast(l$a <- 5)

#> \- 0
#> \- `<-
#> \- 0
#> \- `$
#> \- `l
#> \- `a
#> \- 5

ast(names(l) <- "b")

#> \- 0
#> \- `<-
#> \- 0
#> \- `names
#> \- `l
#> \- "b"
```

现在我们有完整的版本：

```
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"
```

虽然这个函数的完整版本是相当复杂的，但是要记住我们是通过一步步添加简单的组成部分来编写它的。

14.7.3 修改调用树

下一步比较复杂的是返回一个修改过的调用树，就像你使用 `bquote()` 得到的结果一样。`bquote()` 是 `quote()` 的一种更灵活的形式：它允许你有选择地引用或者不引用表达式的某些部分（类似于 Lisp 的撇号操作符）。除了包围在 `.` 中的部分以外，

其它一切部分都是被引用的，包围在`.()`中的部分会被计算，然后计算结果会被插入到函数执行结果中：

```
a <- 1
b <- 3
bquote(a + b)
#> a + b
bquote(a + .(b))
#> a + 3
bquote(.(a) + .(b))
#> 1 + 3
bquote(.(a + b))
#> [1] 4
```

这提供了一种相当简单的方法来控制需要计算什么以及什么时候计算。`bquote()`是如何工作的呢？下面，同我们的其它函数一样，我使用了相同的风格重写了**`bquote()`**：它需要已经被引用的输入，并使基本情况和递归情况更加清楚：

```
bquote2 <- function (x, where = parent.frame()) {
  if (is.atomic(x) || is.name(x)) {
    # 保持不变
    x
  } else if (is.call(x)) {
    if (identical(x[[1]], quote(.))) {
      # 调用了.(), 所以要计算
      eval(x[[2]], where)
    } else {
      # 否则，递归地 apply，并把返回的结果转换成调用
      as.call(lapply(x, bquote2, where = where))
    }
  }
}
```

```
} else if (is.pairlist(x)) {  
  as.pairlist(lapply(x, bquote2, where = where))  
} else {  
  # 用户提供了错误的输入  
  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  
  #> }
```

这和之前的递归函数之间的主要区别是，在我们处理了调用和成对列表的每个元素之后，我们需要把它们强制转回它们的原始类型。

注意，对于创建运行时(run-time)使用的表达式来说，修改源代码树(source tree)的函数是最有用的，而不是那些存回最初的源代码文件(original source file)的函数。这是因为所有非代码的信息(non-code information)都丢失了：

```
bquote2(quote(function(x = .(x)) {  
  # 这是一行注释  
  x + # funky spacing  
  .(y)
```



```
)))  
#> function(x = 1) {  
#> x + 2  
#> }
```

这些工具类似于 Lisp 宏(macro)，就像在 Thomas Lumley 写的《Programmer's Niche: Macros in R》(http://www.r-project.org/doc/Rnews/Rnews_2001-3.pdf#page=10)中讨论的那样。然而，宏是在编译时(compile-time)运行的，在 R 语言中没有任何意义，并且总是返回表达式。它们也有点像 Lisp 的 **fexprs**(<http://en.wikipedia.org/wiki/Fexpr>)。fexpr 是默认情况下参数不会被计算的函数。当从其它语言中寻找有用的技术的时候，理解术语宏和 **fexpr** 是有用的。

14.7.4 练习

1. 为什么 **logical_abbrev()** 要使用 **for** 循环，而不是泛函，比如 **lapply()**？
2. 当给定了引用的对象的时候，**logical_abbrev()** 可以正常工作，但是给定了现有函数时，却不能正常工作，如以下示例。为什么不能呢？怎样修改 **logical_abbrev()** 可以让它工作在函数上呢？想想组成函数的组成部分。

```
f <- function(x = TRUE) {  
  g(x + T)  
}  
logical_abbrev(f)
```

3. 写一个名为 **ast_type()** 的函数，它返回 **"constant"**、**"name"**、**"call"** 或 **"pairlist"**。 ("常量"、"名字"、"调用"或"成对列表"。) 重写 **logical_abbrev()**、**find_assign()** 和 **bquote2()** 使用 **switch()** 来调用这个函数，而不是嵌套的 **if** 语句。

4. 写一个函数，它提取对某个函数的所有调用。比较你的函数和 `pryr::fun_calls()`。
5. 编写一个包装器 `bquote2()`，它进行非标准计算，这样你不就需要显式地对输入使用 `quote()`。
6. 比较 `bquote2()` 和 `bquote()`。`bquote()` 有一个小错误：它不会替换不带参数的函数调用。为什么呢？
7. 改进基本的 `recurse_call()` 模板，使它能处理函数和表达式列表。(比如，从 `parse(path_to_file)` 开始)

15 领域特定语言

R 语言中的**第一类环境**、**词法作用域**、**非标准计算**以及**元编程**(metaprogramming)的结合,给了我们一个功能强大的工具箱,可以用来创建嵌入式的**领域特定语言**(Domain specific languages)。嵌入式领域特定语言利用了**宿主语言**的**解析**和**执行**框架,但是调整了**语义**,以使它们更适合于某种**特定的任务**。**领域特定语言**是一个非常**大**的话题,本章只能讨论一点皮毛,只关注于重要的**实现技术**,而不是首先让你怎样想出语言。如果你有兴趣学习更多这方面的知识,那么我强烈推荐 Martin Fowler 写的《领域特定语言》

(<http://amzn.com/0321712943?tag=devtools-20>)。它为创建领域特定语言讨论了许多选项,并且提供了许多不同的语言的例子。

R 语言中最受欢迎的**领域特定语言**是**公式规范**(formula specification),它提供了一种简洁的方式来描述模型中**预测变量**和**响应变量**之间的关系。其它例子还包括 **ggplot2**(可视化)和 **plyr**(数据操作)。另一个广泛地使用了这些想法的包是 **dplyr**,它提供了 **translate_sql()**函数将 R 表达式转化成 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
```

本章将开发两个简单但有用的**领域特定语言**：一个用于生成 HTML，另一个用于把 R 代码中的**数学表达式**翻译成 LaTeX 的形式。

前提条件

本章把本书其它章节中讨论的许多技术结合在了一起。特别是，你需要了解**环境**、**泛函**、**非标准计算**和**元编程**。

15.1 HTML

HTML(超文本标记语言)是大多数网络的基础语言。它是一种 SGML(标准通用标记语言)的特例，它与 XML(可扩展标记语言)相似但是不相同。HTML 看起来像这样：

```
<body>
<h1 id='first'>A heading</h1>
<p>Some text & <b>some bold text.</b></p>
<img src='myimg.png' width='100' height='100' />
</body>
```

即使你以前从未见过 HTML，你仍然可以发现其编码结构的关键组成部分是**标签** (tag): `<tag></tag>`。标签可以被包含在**其它标签**之内，并可以**嵌入**文本信息。一般来说，HTML 会忽略空格(一串空格相当于一个空格)，这样，你可以把前面的示例放在一行之中，它仍然会在浏览器中显示相同的内容：

```
<body><h1 id='first'>A heading</h1><p>Some text & <b>some bold
text.</b></p><img src='myimg.png' width='100' height='100' />
</body>
```

然而，就像 R 代码一样，你通常会对代码进行缩进，以使得 HTML 代码的结构更加清晰。有超过 100 种的 HTML **标签**。但是为了说明 HTML，我们只需要关注这么几个：

`<body>`: 顶层的标签, 所有其它的内容都包含在它之内。

`<h1>`: 创建标题 1(heading-1), 顶层的标题。

`<p>`: 创建一个段落(paragraph)。

``: 加粗的文本。

``: 嵌入一张图片。

(你可能已经猜到了这些标签是做什么用的!)

标签也可以拥有命名的属性。它们看起来像`<tag a="a" b="b"></tag>`。标签的值应该总是包围在单引号或双引号之内。有两个重要属性几乎用于所有标签: `id` 和 `class`。这些都可以与 CSS(级联样式表)联合使用, 来控制文档的样式。

有一些标签, 比如``, 可以没有任何内容。这些被称为空白标签, 并且有一种稍微不同的语法。你可以写成``, 而不用写成``。因为它们没有内容, 所以属性更加重要。事实上, `img` 有三个几乎用于所有图片的属性: `src`(图像的地址)、`width`(宽度)和 `height`(高度)。

因为在 HTML 中`<`和`>`有特殊的含义, 所以你不能在 HTML 中直接地写出它们。因此, 你必须使用 HTML 的转义字符: `>`和`<`。因为那些转义字符都使用了`&`, 所以如果你想要文字中出现`&`, 那么你需要使用`&`进行转义。

15.1.1 目标

我们的目标是通过 R 语言很容易地产生 HTML 代码。在一个具体的例子中, 我们想生成下面的 HTML 代码, 它看起来尽可能与真实的 HTML 代码类似。

```
<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>
```

为了做到这些，我们将针对下面的领域特定语言展开工作：

```
with_html(body(
  h1("A heading", id = "first"),
  p("Some text &", b("some bold text")),
  img(src = "myimg.png", width = 100, height = 100)
))
```

注意嵌套的函数调用与嵌套的标签是相同的：未命名的参数变成了标签的内容，而命名参数成为它们的属性。因为在这个 API 中，标签和文本是明显不同的，所以我们可以自动对&和其它特殊字符进行转义。

15.1.2 转义

转义(Escaping)在领域特定语言中是非常基础的，所以这将是我们的第一个话题。为了创建转义字符，我们需要给&赋予特别的意义，并且不以双转义字符(double-escaping)结尾。最简单的方法是创建一个 **S3** 类来区分普通文本(需要转义)和 **HTML**(不需要转义)。

```
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，我们写一个转义方法，它使 **HTML** 保持不变，而对普通文本则对特殊字符(&, <, >)进行转义。为了方便，我们还添加一个用于列表的方法。

```
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)  
}  
# 现在我们检查一下它能不能工作  
escape("This is some text.")  
#> <HTML> This is some text.  
escape("x > 1 & y < 2")  
  
#> <HTML> x &gt; 1 &amp; y &lt; 2  
# 两层转义也没问题  
escape(escape("This is some text. 1 > 2"))  
#> <HTML> This is some text. 1 &gt; 2  
# 我们知道的 HTML 文本不会被转义。  
escape(html("<hr />"))  
#> <HTML> <hr />
```

转义是许多领域特定语言的重要组成部分。

15.1.3 基本标签函数

接下来，我们将编写一些简单的标签函数，然后找出如何推广这个函数，以便覆盖所有可能的 HTML 标签。让我们从<p>开始。HTML 标签可以同时具有属性(比如 id 和 class)和子标签(比如或<i>)两种性质。我们需要在函数调用中找到一些方法来区分它们。鉴于属性是已命名的值，而子标签没有名字，似乎使用已命名或

未命名的参数来区分是很自然的事情。例如，一个对 `p()` 的调用，可能看起来像这样：

```
p("Some text", b("some bold text"), class = "mypara")
```

我们可以在函数定义中列出 `<p>` 标签所有可能的属性。然而，这不仅很困难，因为有很多属性，还因为它还可以使用自定义属性(<http://html5doctor.com/html5-custom-data-attributes/>)。所以，我们只使用 `...`，并根据是否命名来区分组件。要正确地做到这一点，我们需要知道 `names()` 中的不一致：

```
names(c(a = 1, b = 2))
```

```
#> [1] "a" "b"
```

```
names(c(a = 1, 2))
```

```
#> [1] "a" ""
```

```
names(c(1, 2))
```

```
#> NULL
```

出于这种考虑，我们创建两个辅助函数来提取向量中的已命名和未命名的组件：

```
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) == ""]  
}
```

我们现在可以创建 `p()` 函数。请注意，这里有一个新函数：`html_attributes()`。它使用一个“名称-值”对列表，来创建正确的 HTML 属性规范。它有点复杂(部分原因是它处理了一些我没有提到的 HTML 特性)。但是，因为它不是那么重要，也没有引入任何新的思想，所以我不会在这里进行讨论(你可以在网络上找到相关资料)。


```
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>
```

15.1.4 标签函数

使用这个 `p0` 函数的定义，可以很容易地看出我们应该如何把这种方法应用到不同的标签上：我们只需要用一个变量来替换“`p`”即可。我们将使用闭包来方便地为给定的标签名称生成标签函数：

```
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, ">"
))
}
```

(我们强制标签进行计算，因为我们可能会从循环中调用这个函数。这将有助于避免由延迟计算造成的潜在错误)。

现在我们可以运行之前的例子：

```
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 标签编写函数之前，我们需要为空标签创建一个 `tag()` 的变种。它可以与 `tag()` 非常相似，但是如果存在任何未命名的标签，那么它需要抛出一个错误。还要注意，标签本身会略有不同：

```
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 处理所有的标签

下一步，我们需要一个包含所有 HTML 标签的列表：

```
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")
```

如果你仔细看看这个列表，你将会发现有相当多的标签与基本 R 函数具有相同的名称(`body`、`col`、`q`、`source`、`sub`、`summary`、`table`)，以及其它一些标签与一些流行的包具有相同的名称(比如 `map`)。这意味着，在全局环境或者包环境中，在默认情况下，我们不想让所有函数都可用。因此，我们要把它们放在一个列表中，并且添加一些额外的代码，使得在必要时可以很容易地使用它们。首先，我们创建一个命名列表：

```
tag_fs <- c(  
  setNames(lapply(tags, tag), tags),  
  setNames(lapply(void_tags, void_tag), void_tags)  
)
```

这样给出了显式(但是不简洁)地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>
```

然后，通过一个函数，让我们在那个列表形成的上下文中计算代码，我们可以完成我们的 HTML 领域特定语言：

```
with_html <- function(code) {  
  eval(substitute(code), tag_fs)  
}
```

这给了我们一个简洁的 API，它允许我们写出 HTML，而不需要总是指定命名空间(译者注：即 `tag_fs`)。

```
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>
```

如果在 `with_html()` 内部，你想访问那些由于与 HTML 标签同名，而被覆盖的 R 语言函数，那么你可以使用完整的 `package::function` 的方式来访问。

15.1.6 练习

1. `<script>`与`<style>`标签的转义规则是不同的：你并不想对尖括号(`<`)或"与"(`&`)符号进行转义，但是你想对`</script>`和`</style>`进行转义。调整上面的代码以遵循这些规则。
2. 对所有的函数都使用`...`，有一些大缺点。没有了输入验证，并且函数文档也会缺少参数的描述，或者代码自动补全功能也不能用。创建一个新函数，当给定一个已命名的列表，该列表中保存了标签，以及标签的属性名称(如下)，那么该函数会创建函数来定位这个问题。

```
list(  
  a = c("href"),  
  img = c("src", "width", "height")  
)
```

所有的标签都应该得到 `class` 和 `id` 属性。

3. 目前，HTML 看起来并不是那么漂亮，而且很难看到它的结构。怎样修改 `tag()`，使得它能进行缩进和格式化？

15.2 LaTeX

下一个领域特定语言将把 R 表达式转换成它们的 LaTeX 数学等价形式。(这有点像 `?plotmath`, 不过是对文本的, 而不是对绘图的罢了。) LaTeX 是数学家和统计学家的通用语言: 每当你想用文本形式来描述一个方程(equation)时(比如在一封电子邮件中), 你可以把它写成一个 LaTeX 方程。因为许多报告既使用了 R, 又使用了 LaTeX, 所以对于把数学表达式从一种语言自动转换到另一种语言来说, 它可能是有用的。

因为我们需要转换函数和名字, 所以这个数学领域特定语言将比 HTML 领域特定语言更加复杂。我们还需要创建一个"默认的"转换, 这样可以对我们不知道的函数给出一种标准的转换。像 HTML 领域特定语言一样, 我们也需要泛函, 以便更容易地生成翻译器。

在我们开始之前, 让我们快速浏览一下, 在 LaTeX 中, 公式(formula)是如何表达的。

15.2.1 LaTeX 数学

LaTeX 数学是很复杂的。幸运的是, 它们的文档都写得很好 (<http://en.wikibooks.org/wiki/LaTeX/Mathematics>)。文档中提到, 它们都有一个相当简单的结构:

大多数简单的数学公式是跟 R 语言使用同样的方式写成的: `x*y`, `z^5`。

下标使用 `_` 来表示(如 `x_1`)。

特殊字符以 `\` 开头: `\pi = \pi`、`\pm = \pm` 等等。

在 LaTeX 中有大量的符号。在 Google 中搜索 LaTeX 数学符号将返回许多列表 (<http://www.sunilpatel.co.uk/latextype/latex-math-symbols/>)。甚至, 这里还有

一个服务(<http://detexify.kirelabs.org/classify.html>), 它可以查找你在浏览器中描绘出的符号。

更复杂的函数看起来像`\name{arg1}{arg2}`。例如, 写一个分数(fraction), 你可以使用`\frac{a}{b}`。写一个平方根, 你可以使用`\sqrt{a}`。

把元素组织在一起, 可以使用`{}`: 比如, `x ^ a + b` 与 `x ^ {a + b}`。

在好的数学排版中, 变量和函数之间是有区别。但是, 如果没有额外的信息, LaTeX 并不知道 `f(a * b)` 是代表使用输入 `a * b` 对函数 `f` 进行调用, 还是 `f*(a * b)` 的简写。如果 `f` 是一个函数, 那么你可以用`\textrm{f}(a * b)`来告诉 LaTeX 使用直立的字体对其进行排版。

15.2.2 目标

我们的目标, 是使用这些规则, 自动地将一个 R 表达式, 转化为适当的 LaTeX 表示形式。我们会分四个阶段来解决这个问题:

1. 转化已知的符号: `pi -> \pi`
2. 其它符号保持不变: `x -> x`、`y -> y`
3. 把已知函数转化为它们的特殊形式: `sqrt(frac(a, b)) -> \sqrt{\frac{a}{b}}`
4. 使用`\textrm` 包装未知函数: `f(a) -> \textrm{f}(a)`

我们将按照与 HTML 的领域特定语言相反的方向, 来编写这段翻译代码。我们将从基础开始, 因为这使得使用我们的领域特定语言进行实验变得容易, 然后我们将回到产生所需输出的工作。

15.2.3 to_math 函数

首先，我们需要一个**包装器**函数，它将把 R 表达式转换成 LaTeX 数学表达式。它与 `to_html()` 的工作方式是一样的：捕获未计算的表达式，并且在一个特殊的环境中对它进行计算。然而，特殊的环境不再是固定的。它会根据表达式而有所不同。我们这样做是为了能够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

我们的第一步是创建一个环境，它将用于转换**特殊的 LaTeX 希腊字符**，比如 `pi` 到 `\pi`。这与 `subset` 函数使用了相同的基本技巧，可以通过**列名**来选择列的范围 (`subset(mtcars, , cyl:wt)`)：在一个特殊环境中，把名字绑定到一个字符串上。

我们这样来创建这个环境：对一个向量进行命名，然后把向量转换成一个列表，并且将列表转换成一个环境。

```
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，我们可以检查一下：


```
latex_env <- function(expr) {  
  greek_env  
}  
to_math(pi)  
  
#> [1] "\\pi"  
to_math(beta)  
#> [1] "\\beta"
```

15.2.5 未知符号

如果一个符号不是希腊字符，那么我们想让它保持不变。这是很棘手的，因为我们事先并不知道什么符号会被使用，所以我们不可能把所有的符号都生成出来。所以，我们将使用一点点的元编程技术，以找出在一个表达式中，存在什么符号。all_names 函数需要一个表达式，并做下面的事情：如果表达式是一个名称，那么 all_names 将把表达式转换成一个字符串；如果表达式是一个调用，那么 all_names 将对其参数进行递归向下搜寻。

```
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"
```

我们现在想要传入那个符号列表，并将其转换成一个环境，这样，每个符号都会映射到其对应的字符串表示形式(比如，`eval(quote(x), env)`产生"x")。我们再次应用这种模式：把命名的字符向量转化为一个列表，然后把这个列表转换成一个环境。

```
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"
```

这样可以工作，但是我们需要把它与希腊符号环境联合起来。由于在默认情况下，我们希望给予希腊符号更高的优先级(例如，`to_math(pi)`应该得到"`\\pi`"，而不是"pi")，所以 `symbol_env` 需成为 `greek_env` 的父环境。要做到这一点，我们需要用一个新的父环境来复制 `greek_env`。虽然 R 语言没有附带复制环境的函数，但是我们可以很容易地通过结合现有的两个函数来创建一个：

```
clone_env <- function(env, parent = parent.env(env)) {  
  list2env(as.list(env), parent = parent)  
}
```

这里给出了一个函数，它可以转换已知(希腊字符)的和未知的符号。

```
latex_env <- function(expr) {  
  # 未知符号  
  names <- all_names(expr)  
  symbol_list <- setNames(as.list(names), names)  
  symbol_env <- list2env(symbol_list)  
  # 已知符号  
  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，我们将为我们的领域特定语言添加函数。我们将从两个辅助闭包函数开始，它们可以让添加新的一元和二元运算符变得很容易。这些函数很简单：它们仅仅组装字符串。(同样，我们使用 `force()` 函数以确保参数在正确的时候被计算了。)

```
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)  
  }  
}
```

使用这些辅助函数，我们可以通过一些映射示例，来说明如何从 R 转换到 LaTeX。注意，由于 R 的词法作用域规则的帮助，我们能很容易地为标准函数提供新的含义，比如+、-和*，甚至(和{。

```
# 二元运算符  
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("_")  
  
# 分组  
f_env$"{" <- unary_op("\\left{ ", "\\right}")  
f_env$"(" <- unary_op("\\left( ", "\\right)")  
f_env$paste <- paste  
  
# 其它数学函数  
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$frac <- function(a, b) {  
  paste0("\\frac{", a, "{", b, "}")  
}  
# 打标签  
f_env$hat <- unary_op("\\hat{", "}")  
f_env$tilde <- unary_op("\\tilde{", "}")
```

我们再次修改 `latex_env()` 来包含这个环境。它应该是最后那个 R 用于查找名称的环境：换句话说，`sin(sin)` 应该是可以工作的。

```
latex_env <- function(expr) {  
  # 已知函数  
  f_env  
  # 默认符号  
  names <- all_names(expr)  
  symbol_list <- setNames(as.list(names), names)  
  symbol_env <- list2env(symbol_list, parent = f_env)  
  # 已知符号  
  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，我们将添加默认行为，处理那些我们还不知道的函数。就像未知的名字一样，我们无法事先知道这些是什么，所以我们再次使用一点元编程技巧把它们找出来：

```
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"
```

我们需要一个闭包，它将为每个未知的调用生成函数。

```
unknown_op <- function(op) {  
  force(op)  
  function(...) {  
    contents <- paste(..., collapse = ",")  
    paste0("\\mathrm{", op, "}", contents, "")  
  }  
}
```

我们再次更新了 `latex_env()`:

```
latex_env <- function(expr) {  
  calls <- all_calls(expr)  
  call_list <- setNames(lapply(calls, unknown_op), calls)  
  call_env <- list2env(call_list)  
  # 已知函数  
  f_env <- clone_env(f_env, call_env)  
  # 默认函数  
  symbols <- all_names(expr)  
  symbol_list <- setNames(as.list(symbols), symbols)  
  symbol_env <- list2env(symbol_list, parent = f_env)  
  
  # 已知符号  
  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。一些特殊符号，需要在前面添加一个反斜杠来进行转义，它们是：\、\$和%。与 HTML 一样，你需要确保不会以双转义字符结尾。所以，你需要创建一个小的 S3 类，然后在函数运算符中使用该类。这也将让你在必要时可以嵌入任意 LaTeX 文本。
2. 完成领域特定语言，以支持 `plotmath` 所支持的所有功能。
3. 在 `latex_env()` 中有重复的模式：我们传入一个字符向量，为每一块做点工作，然后将其转化为一个列表，再然后将列表转换为一个环境。写一个函数使这个任务自动化，然后重写 `latex_env()`。

4. 研究 `dplyr` 的源代码。它的结构中的一个重要组成部分是 `partial_eval()`，当一些组件引用(refer to)了数据库中的变量，而另一些引用了本地 R 对象的时候，它可以帮助管理表达式。请注意，如果你需要把小的 R 表达式翻译成其它语言，比如 JavaScript 或 Python，那么你可以使用非常类似的思路。

第四部分 性能

16 性能

R 语言不是一种快的语言。这不是偶然的。R 语言的设计目的是为了使你的数据分析和统计工作变得更加容易。它不是为了使你的计算机生活得更容易而创造的。虽然，与其它编程语言相比，R 语言是缓慢的，但是在大多数情况下，它已经足够快了。

本书这一部分的目标，是让你对 R 语言的性能特征有更深层次的理解。在本章中，你将了解到 R 语言做出了一些权衡：相对于**性能问题**，它更重视**灵活性**。后续四章将给你提供一些技能，在必要的时候，可以用来提高你的代码的速度：

在第 17 章中，你将系统地学习如何让你的代码变得更快。首先，你要找出代码为什么会慢，然后你应用一些通用技术使缓慢的部分变得更快。

在第 18 章，你将了解 R 语言如何使用**内存**，以及**垃圾收集**和**修改时复制**(copy-on-modify)如何影响**性能**和**内存使用**。对于真正的高性能代码，你可以使用另一种编程语言。

第 19 章会教你最基本的 C++ 知识，这样你就可以使用 **Rcpp** 包编写快速的代码。要真正了解内置的基本函数的性能，你需要学习一些关于 R 的 C 语言 API 的知识。

在第 20 章，你将学习一点关于 R 的 C 语言内核的知识。

让我们从理解 R 语言为什么缓慢开始学习。

16.1 R 语言为什么慢？

为了理解 R 语言的性能，把 R 语言想象成既是一种语言，又是那种语言的一个实现，是有帮助的。R 语言是**抽象的**：它定义了 R 语言代码的意义以及应该如何工作。**实现是具体的**：它读取 R 代码并计算出结果。最流行的实现，是来自于 [r-project.org](http://rproject.org) 的实现(<http://rproject.org>)。我把这个实现称为 **GNU-R**，以便与其它 R 语言实现区别开来，在本章的后面，我将讨论其它的 R 语言实现。

R 语言与 GNU-R 的区别有点不清楚，因为 R 语言并没有正式定义过。虽然，确实有 **R 语言定义**(<http://cran.r-project.org/doc/manuals/R-lang.html>)，但是它不是正式的并且不完整。R 语言主要是定义了 **GNU-R 是如何工作**的一些条款。其它语言却不是这样，比如 C++(<http://isocpp.org/std/the-standard>)和 javascript(<http://www.ecma-international.org/publications/standards/Ecma-262.htm>)，通过制定正式的规范，明确地区分了**语言**和**实现**，描述了语言应该如何工作的方方面面的细枝末节。但是，**R 语言**和 **GNU-R** 之间的区别仍然是有用的：在不破坏现有代码的情况下，由**语言**造成的低性能是很难修复的；而由**实现**造成的低性能则更容易修复。

在 16.3 节，我将讨论一些设计 R 语言所使用的方法，它们对 R 语言的速度造成了**基本的限制**。在 16.4 节，我讨论了为什么 GNU-R 目前距离理论上的最大速度还很远，以及为什么性能的改进会这么慢。虽然，很难确切地知道更好的实现会快多少，但是使速度提高大于 **10 倍**应该是可以实现的。在 16.5 节，我讨论了一些较有前景的、新的 R 语言实现，并描述了一种它们用到的、使 R 代码运行得更快的重要技术。

除了由于**设计**和**实现**造成的性能限制以外，不得不说，很多 R 代码是因为**写得太差**才很缓慢。很少有 R 用户经过了任何正式的编程或者软件开发训练。更少有人以写 R 代码为生。大多数人使用 R 语言来理解数据：**快速得到一个答案**比**开发在**

各种各样的情况下都能工作的系统更加重要。这意味着，让大多数 R 语言代码变得更快是相对容易的，在接下来的章节中，我们将会看到这一点。

在我们检查 R 语言和 GNU-R 中一些比较慢的部分之前，我们需要学习一点关于**基准测试**的内容，这样可以使我们对性能的直觉，有一个实实在在的基础。

16.2 微基准测试

微基准测试(Microbenchmarking)是对一段非常短小的代码段进行**性能测量**，这可能需要在**微秒**(μs)或者**纳秒**(ns)的级别上。我将使用**微基准测试**来说明非常底层的 R 代码的性能，这样可以帮助你发展直觉，直观地理解 R 语言是如何工作的。虽然，这种直觉对于提高真实代码的速度来说，并没有多大作用的，但是，在**微基准测试**中观察到的差异，通常是由实际代码中的**高阶效应**(higher-order effect)控制的；深入理解**亚原子物理学**(subatomic physics)对于**烘烤**来说，并不是那么有用。不要因为这些**微基准测试**而改变你编写代码的方式。等到你读过了后续章节中的**实用建议**之后再说。

在 R 语言中，最好的**微基准测试**工具是 **microbenchmark** 包(<http://cran.r-project.org/web/packages/microbenchmark/>)。它提供了非常精确的计时，使得对只花了少量时间的操作进行比较变为可能。例如，下面的代码比较了两种计算平方根的方法的速度。

```
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()`会运行每个表达式 100 次(由 `times` 参数控制)。在这个过程中，它还会**随机选择**表达式的执行顺序。它在结果中汇总了这些指标：最小值(min)，低四分位数(lq)，中位数，高四分位数(uq)和最大值(max)。关注于**中位数**，并使用高、低四分位数(lq 和 uq)来感受一下数值的变化情况。在这个示例中，你可以看到使用**特殊目的**的 `sqrt()`函数比一般的**求幂运算符**的速度要快。

如同所有的**微基准测试**一样，要特别注意**单位**：每次计算需要大约 800 纳秒(ns)，也就是 800 个十亿分之一秒。为了帮助校准(calibrate)**微基准测试**在运行时的影响，考虑一下一个函数在一秒钟内需要运行多少次是有用的。如果一个**微基准测试**需要：

- 1 毫秒(ms)，那么一千次调用需要一秒。
- 1 微秒(us)，那么一百万次调用需要一秒。
- 1 纳秒(ns)，那么十亿次调用需要一秒。

`sqrt()`函数需要大约 800 纳秒，也就是 0.8 微秒，来计算 100 个数的平方根。这意味着，如果你重复这个操作一百万次，则需要 0.8 秒。所以，改变你计算平方根的方法不太可能显著地影响实际代码。

16.2.1 练习

1. 你可以使用内置函数 `system.time()`，而不是使用 `microbenchmark()`。但 `system.time()`远没有那么精确，所以你需要一个**循环**来重复每次操作很多遍，然后通过除法来求出每次操作的平均运行时间，就像下面的代码这样。

```
n <- 1:1e6  
system.time(for (i in n) sqrt(x)) / length(n)  
system.time(for (i in n) x ^ 0.5) / length(n)
```

由 `system.time()` 得到的估计值与由 `microbenchmark()` 得到的估计值相比怎么样？为什么它们是不同的？

2. 这里有其它的两种方法来计算一个向量的平方根。你认为哪一个是最快的？哪一个是最慢的？使用**微基准测试**来测试你的答案。

```
x ^ (1 / 2)  
exp(log(x) / 2)
```

3. 使用**微基准测试**对基本算术运算符(`+`、`-`、`*`、`/` 和 `^`)的速度进行排序。将结果进行可视化。比较**整数**与**双精度浮点数**的算术运算速度。
4. 你可以改变**微基准测试**结果的单位，它用 `unit` 参数来表示。使用 `unit = "eps"` 显示每 1 秒中需要的计算数量。请使用 `eps` 单位重复上面的基准测试。它改变了你对性能的直觉吗？

16.3 语言性能

在这一节中，我将探讨三个限制了 R 语言性能的机制：**极端的动态机制**、**在可变的环境中进行名字查找**以及**函数参数的延迟计算**。我将通过**微基准测试**来说明每一种机制，显示它如何使 GNU-R 变慢。我对 GNU-R 进行基准测试，因为你无法对 R 语言进行基准测试(它不能运行代码)。(译者注：我们只能测试 R 语言的具体实现，而不能测试抽象的 R 语言定义。)这意味着，测试的结果对这些设计决策的开销来说，只具有一定的参考意义，不过仍然是有用的。我已经挑选了这三个例子，来说明对语言设计来说，非常关键的权衡考虑：设计师必须在**速度**、**灵活性**和**实现的难易度**方面取得平衡。

如果你想了解更多关于 R 语言的性能特征方面的内容，以及它们是如何影响真正的代码的，那么我强烈推荐你阅读由 Floreal Morandat、Brandon Hill、Leo Osvald 和 Jan Vitek 撰写的《Evaluating the Design of the R Language》

(<https://www.cs.purdue.edu/homes/jv/pubs/ecoop12.pdf>)。它使用了一种强大的方法，该方法结合了修改过的 R 解释器以及一组来源广泛的代码。

16.3.1 极端的动态机制

R 语言是一种**极端动态**(extremely dynamic)的编程语言。几乎任何东西在创建后都可以修改。举几个例子来说明，你可以：

改变函数的**函数体**、**参数**和**环境**。

为**泛型函数**改变 S4 的方法。

为 S3 对象添加**新字段**，甚至改变它的**类**。

使用 `<-` 来修改本地环境以外的对象。

唯一不可以改变的东西是**封装在命名空间中的对象**，它是当你加载一个包时创建的。

动态机制的优点是在工作之前你只需要最小程度的前期计划。你可以在任何时候改变你的想法，通过**迭代**的方法来寻求解决方案，而不必重新开始。动态机制的缺点是，对于一个给定的**函数调用**，很难准确地预测将会发生什么。这是一个问题，因为越容易预测会发生什么，解释器或者编译器就越容易进行优化。(如果你了解更多的细节，那么 Charles Nutter 在《On Languages, VMs, Optimization, and the Way of the World》中扩展了这个思路 (<http://blog.headius.com/2013/05/on-languagesvms-optimization-and-way.html>)。) 如果解释器无法预测会发生什么，那么它必须事先考虑许多选项，才能找到一个正确的选择。例如，下面的循环在 R 语言中是缓慢的，因为 R 语言不

知道 `x` 总是一个整数。这意味着，在循环的每一次迭代中，R 语言都必须寻找正确的 `+` 方法。(比如，这里是在进行双精度浮点数加法还是整数加法?)

```
x <- 0L
for (i in 1:1e6) {
  x <- x + 1
}
```

对非原语函数来说，找到正确的方法的开销是比较高的。下面的微基准测试说明了对于 **S3** 类、**S4** 类和引用类的方法分派开销。我为每个面向对象系统都创建了一个泛型函数和一个方法，然后调用泛型函数，看看找到并且调用该方法需要多长时间。我还为直接调用函数进行计时，以进行比较。

```
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
```

在我的电脑上，直接调用函数需要大约 200 纳秒。S3 类的方法分派则多需要 2000 纳秒；S4 类的方法分派需要 11000 纳秒；引用类的方法分派需要 10000 纳秒。S3 类和 S4 类的方法分派开销是比较大的，因为每当调用泛型函数的时候，R 语言都必须寻找正确的方法；在两次调用之间，方法可能已经发生改变了。R 语言可以通过在调用之间进行缓存(caching)的方法来做得更好的，但是缓存要做得正确是很难的，它也是一种臭名昭著的错误来源。

16.3.2 在可变的环境中进行名字查找

令人惊讶的是，在 R 语言中很难找到与一个名字相关联的值。这是由词法作用域和极端的动态机制共同造成的。以下面的例子为例。每当我们打印 `a`，它都来自于一个不同的环境：

```
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，你不能仅仅做一次名字查找：你必须每次都从头开始查找。由于这个事实，这个问题变得更严重了：几乎每一个操作都是词法作用域的函数调用。你可能会认为以下的简单函数调用了两个函数：`+`和`^`。事实上，它调用了四个函数，因为`{`和`(`也是 R 语言中的普通函数。

```
f <- function(x, y) {
  (x + y) ^ 2
}
```

由于这些函数在全局环境中，所以 R 语言必须检查搜索路径上的每一个环境，而这很容易就有 10 个或者 20 个环境。下面的微基准测试提示了性能开销。我们创建了四个版本的 `f()` 函数，在 `f()` 的环境和 `base` 环境(即`+`、`^`、`(`和`{`的定义所处的 `base` 包的环境)之间，每个 `f()` 都会多比上个 `f()` 多一个环境(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 `base` 环境之间，每多一个额外的环境，就会使函数慢约 30 纳秒。

也许有可能实现一个缓存系统，使得 R 语言只需要查找每个名字的值一次。但是，这是很困难的，因为有太多方法可以改变与一个名字相关联的值：`<<-`、`assign()`、`eval()` 等等。任何缓存系统都必须知道这些函数，以确保缓存失效 (invalidated) 过程是正确的，以及你得到的不是一个过时的 (out-of-date) 值。

另一种简单的修改，可能是添加更多你不能覆盖 (override) 的内置常量 (built-in constant)。例如，这意味着，R 语言总是很清楚 `+`、`-`、`{` 和 `(` 的含义，那么你就不必反复查看它们的定义。但是，这将使解释器变得更加复杂 (因为有特例)，因此会难以维护，并且使语言变得不那么灵活。这也许能改变 R 语言，但是它不太可能影响很多现有的代码，因为覆盖像 `{` 和 `(` 这样的函数是很坏的主意。(译者注：意思是很少有人会重新定义这种常规的函数，所以即使 R 语言加入了处理特例的功能，对现有的 R 语言代码也起不了多大作用。)

16.3.3 延迟计算的开销

在 R 语言中，函数参数是延迟计算的。(如 6.4.4 节和 13.1 节中讨论的那样) 为了实现延迟计算，R 语言使用了承诺(promise)对象，它包含了用于计算结果的表达式以及执行计算的环境。创建这些对象会有一些开销，所以，函数的每一个额外参数，都会让它的速度降低一点。

下面的微基准测试比较了一个非常简单的函数的运行时间。每个版本的函数都比上一个版本多一个额外的参数。这表明，每添加一个额外的参数会让函数变慢 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
```

在其它的编程语言中，增加额外的参数几乎是没有开销的。如果参数永远都没有使用到，那么许多**编译型语言**甚至会警告你(如在上面的例子中)，并自动把它们从函数中删除。

16.3.4 练习

1. 在基本函数中，`scan()`拥有最多的参数(21 个)。每当调用 `scan` 的时候，计算 21 个**承诺**大约需要多少时间？给定一个简单的输入(如 `scan(text = "1 2 3", quiet = T)`)，总运行时间中的多大比例是用于创建这些**承诺**的？
2. 阅读《Evaluating the Design of the R Language》(<https://www.cs.purdue.edu/homes/jv/pubs/ecoop12.pdf>)。R 语言的哪些其它方面使它变慢呢？建立**微基准测试**来说明。
3. S3 的方法分派的性能是如何随着**类向量**的长度而改变的？S4 的方法分派的性能是如何随着**超类**(super class)的数量而改变的？对于**引用类**呢？
4. 对 S4 的方法分派来说，**多重继承**和**多分派**的开销是什么？
5. 为什么对于 `base` 包中的函数来说，**名字查找**的开销更小？

16.4 实现性能

R 语言的设计，限制了其最大的理论性能，但是 GNU-R 目前还远未达到这个最大值。我们可以(或者将来可以)做很多事情来提高它的性能。本节讨论了 GNU-R 的某些方面，它们的慢不是由于它们的定义，而是由于它们的实现。

R 语言有 20 多岁了。它包含近 80 万行代码(约 45%的 C 语言代码、19%的 R 代码和 17%的 Fortran 代码)。只有 R 核心团队的成员(简称 R-core)，才能改变基础 R 语言。目前 R-core 有 20 个成员(<http://www.r-project.org/contributors.html>)，但是只有 6 个成员在日常开发中是活跃的。R-core 中没有一个人全职工作在 R 语言上。他们中的大多数人是统计学教授，只能

把较少的时间花在 R 语言上。因为对**更新**必须保持谨慎，以避免破坏现有代码，所以 R-core 往往对接受新代码表现得非常保守。R-core 拒绝了可能提高性能的建议，是非常令人沮丧。但是，R-core 关注的首要问题并不是让 R 语言变得更快，而是建立一个稳定的数据分析和统计平台。

下面，我将介绍两个小但是有说明意义的示例，它们目前是 R 语言中较慢的部分，但是如果通过一些努力，是可以变得更快的。它们不是基础 R 语言的关键部分，但是在过去，对于我来说，它们一直是沮丧的来源。与所有的**微基准测试**一样，这些并不会影响大多数代码的性能，但是对特殊情况可能是很重要的。

16.4.1 从一个数据框中提取单个值

下面的**微基准测试**显示了七种从内置的 **mtcars** 数据集中访问**单个值**(在右下角的数)的方法。性能的变化是惊人的：最慢的方法比最快的方法慢了超过 30 倍。没有理由必须有这么大的性能差异，只是没人有时间来进行修复。

```
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()

一些基本函数是比较缓慢的。例如，采取以下三种方式来实现 `squish()`，该函数确保向量中的最小值至少是 `a`，而最大值至多是 `b`。第一种实现，`squish_ife()`，使用了 `ifelse()`。`ifelse()` 是比较慢的，因为它是相对通用的，所以它必须完全计算所有的参数。第二种实现，`squish_p()`，使用了 `pmin()` 和 `pmax()`。因为这两个函数都太特殊了，所以人们可能期望它们会很快。然而，它们实际上相当缓慢。这是因为它们可以接受任意数量的参数，并且要做一些相当复杂的检查，以确定使用哪种方法。最终的实现使用了基本的子集赋值操作。

```
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
```

使用 `pmin()` 和 `pmax()` 大约比 `ifelse()` 快 3 倍，而直接使用取子集操作则又快了两倍。我们经常可以通过使用 C++ 来做更好的。下面的例子比较了最好的 R 语言实现和一个相对简单但是冗长的 C++ 语言实现。即使你从来没有使用过 C++，你也应该能够看懂函数的基本策略：遍历向量中的每个元素，并且根据元素的值是否小于 `a` 和(或)是否大于 `b` 来进行不同的动作。C++ 实现的速度，大约是最好的纯 R 语言实现的 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 章中学习如何从 R 语言中访问这段 C++ 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` 的大小发生相当大的变化。研究这种差异。`x` 为多大的时候，会造成最大和最小的差异呢？
2. 比较从一个列表中、从矩阵的列中以及从数据框的列中提取一个元素的性能开销。对行也进行相同的比较。

16.5 其它的 R 语言实现

目前，有一些激动人心的新的 R 语言实现。尽管它们都试图尽可能坚持现有的语言定义，但是它们通过利用现代解释器的设计理念来提高运行速度。四个最成熟的开源项目有：

Radford Neal 编写的 pqR(非常快的 R, pretty quick R)(<http://www.pqr-project.org/>)。它建立在 R 2.15.0 之上,修复了许多明显的性能问题,并且提供了更好的内存管理以及对自动多线程提供了一些支持。

BeDataDriven 编写的 Renjin(<http://www.renjin.org/>)。Renjin 使用了 Java 虚拟机,并且有大量的包(<http://packages.renjin.org/>)。

由普渡大学的一个小组编写的 FastR(<https://github.com/allr/fastr>)。FastR 与 Renjin 类似,但是它进行了更加大胆的优化,只是还不太成熟。

Justin Talbot 和 Zachary DeVito 编写的 Riposte(<https://github.com/jtalbot/riposte>)。Riposte 是实验性质的,也挺雄心勃勃的。由它实现的 R 语言的部分,是相当快的。关于 Riposte 的更详细描述在这本书中:《Riposte: A Trace-Driven Compiler and Parallel VM for Vector Code in R》(<http://www.justintalbot.com/wp-content/uploads/2012/10/pact080talbot.pdf>)。

这些大致是按照从**最实用**到**最雄心勃勃**的顺序排列的。另一个项目,由 Andrew Runnalls 编写的 CXXR(<http://www.cs.kent.ac.uk/projects/cxxr/>)并不提供任何性能改进。相反,它旨在重构 R 内部的 C 语言代码,为未来的开发建立更坚实的基础,并且与 GNU-R 保持相同的行为,以及为它的内核创建更好、更可扩展的文档。

R 语言是一种庞大的语言,这些方法是否会成为主流,目前还并不清楚。创建出另一种可选择的实现,并且使用与 **GNU-R 同样的方式**运行所有的 R 代码,是一个艰巨的任务。重构基础 R 语言中的每个函数,并且不仅让这些函数变得更快,而且也有完全相同的、记录在案的错误,你能想象一下,这个工作量有多大吗?然而,即使这些实现从未在 GNU-R 的使用中引起过注意,它们仍然提供了一些好处:

1. 在移植到 GNU-R 之前，更简单的实现可以使得验证新方法变得更容易。
2. 了解语言的哪些方面的改变可以对现有代码只造成最小程度的影响，以及哪些方面改变可以对性能产生最大的影响，可以用于指导我们应该直接关注哪些方面。
3. R 语言的替代实现可以对 R-core 施加压力，促使他们积极进行性能改进。

pqR、Renjin、FastR 和 Riposte 研究的最重要的一种方法，是**延迟计算**的思想。正如 Justin Talbot——Riposte 的作者——指出：“对于长向量，R 语言的执行是完全受内存限制的。它几乎把所有时间都花费在了**把向量从内存中读出以及写到内存中**”。如果我们能够消除这些中间向量，那么我们可以提高性能并且减少内存的使用。

下面显示了一个非常简单的例子，它说明了**延迟计算**是如何起到作用的。我们有三个向量，**x**、**y**、**z**，每一个都包含 1 百万个元素，我们希望求出当 **z** 为 **TRUE** 时，**x + y** 的和。（这是一种很常见的数据分析问题的简化。）

```
x <- runif(1e6)
y <- runif(1e6)
z <- sample(c(T, F), 1e6, rep = TRUE)
sum((x + y)[z])
```

在 R 语言中，这就产生了两个大的临时向量：**x + y**，100 万个元素长，以及**(x + y)[z]**，长约 500000 个元素。这意味着，你需要使用额外的内存，存储**中间计算的数据**，并且你必须在 CPU 和内存之间来回传递数据。这使得计算变慢，因为如果 CPU 总是处于**等待数据**的状态，它就不能发挥最大的工作效率。但是，如果我们使用 C++ 语言，并且使用**循环**来重写这个函数，我们只需要一个中间值：我们看到的**所有值的总和**：

```
#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;
}
```

在我的电脑上，这种方法比等价的向量化的 R 方法快了大概八倍，虽然 R 方法已经算是非常快了。

```
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
```

延迟计算的目标是自动执行这个转换，因此你可以写出简洁的 R 代码，并让它自动翻译成高效的机器码。精妙的翻译器还可以指出如何充分利用多个 CPU 核。在上面的例子中，如果你有四个核，那么你可以把 **x**、**y**、**z** 分成四块，在每个核上执行**条件求和**，然后把四个结果加在一起。**延迟计算**还可以用于**循环**，它会自动发现可以向量化的操作。本章讨论了 R 语言缓慢的一些基本原因。接下来的章节将会告诉你一些工具，当 R 语言的缓慢影响了你的代码的时候，你可以使用。

17 优化代码

"程序员浪费了大量的时间进行思考，或者担心，他们的程序中那些**非关键部分**的速度，当进行调试和维护的时候，这些效率方面的尝试实际上产生了很强的负面影响。" — Donald Knuth

优化代码，让代码运行得更快是一个**迭代**的过程：

1. 找到最大的性能瓶颈(代码中最慢的那一部分)。
2. 尝试消除它(可能会不成功，但是没关系)。
3. 重复上述过程，直到你的代码变得"足够快"。

这听起来很简单，但事实却不是这样的。即使是经验丰富的程序员，也很难识别他们的代码中的性能瓶颈。你应该对你的代码进行评测，而不是依靠你的直觉：使用实际的输入，然后测量每次操作的运行时间。仅当你确定了最重要的瓶颈的时候，你才能尝试消除它们。很难为提高性能提供一般性的建议，但是我会尽最大的努力介绍六种技术，这些技术在许多情况下都可以应用。我还将提出一种通用的性能优化策略，它可以帮助你确保**更快的代码**仍然是**正确的代码**。

很容易就会陷入试图移除所有的性能瓶颈之中。千万不要这样！你的时间是宝贵的，最好是把时间花费在分析你的数据上，而不是用来消除你的代码中可能效率低下的部分。我们要务实一些：你的时间是以小时来计算的，而计算机的时间是以秒来计算的，不要用你的时间去交换计算机的时间。为了执行这个建议，你应该为你的代码设定一个**性能目标**，并且优化工作以这个目标为准。这意味着，你不会消除所有的性能瓶颈。一些性能瓶颈你甚至都不会遇到，因为你已经达到了你的目标，没必要再继续寻找了。其它的一些性能瓶颈，你可能需要忽略并且接受，要么是因为不存在快速便捷的解决方法，要么是因为代码已经经过优化了，再

进行重大的改进是不太可能了。接受这些可能性，然后进入到下一个候选的性能瓶颈。

本章概要

17.1 节描述了如何使用**代码行分析**(line profiling)找到你的代码中的性能瓶颈。

17.2 节概述了七种用来提高你的代码的性能的一般策略。

17.3 节教你如何组织代码，可以使得优化工作尽可能的简单，并且没有错误。

17.4 节提醒你去寻找现有的解决方案。

17.5 节强调了**懒惰**的重要性：通常，让一个函数变得更快的最简单的方法，是让它做更少的工作。

17.6 节简明地定义了**向量化**，并向你展示了如何充分利用内置函数。

17.7 节讨论了**复制数据**的性能风险。

17.8 节向你展示了如何利用 R 语言的**字节码编译器**。

17.9 节将各部分组合在一起形成一个案例研究，展示了如何把重复的(repeated)t 检验速度提高 1000 倍。

17.10 节教导你如何在计算机所有的核中使用**并行化**进行分布式计算。

17.11 节结束这一章，并指出了更多的资源，它们将会帮你写出快速的代码。

前提条件

在这一章里，我们将使用 **lineprof** 包来理解 R 语言代码的性能。使用下面的代码安装它：

```
devtools::install_github("hadley/lineprof")
```

17.1 测量性能

要理解性能，你需要使用**分析器**。有许多不同类型的分析器。R 语言使用一种非常简单的类型，称为**抽样或者统计分析器**。**抽样分析器**每隔几毫秒就会停止代码的执行，并且记录目前正在执行的函数(连同调用该函数的函数，等等)。例如，考虑下面的 **f()** 函数：

```
library(lineprof)
f <- function() {
  pause(0.1)
  g()
  h()
}
g <- function() {
  pause(0.1)
  h()
}
h <- function() {
  pause(0.1)
}
```

(我使用了 **pause()** 而不是 **Sys.sleep()**，是因为 **Sys.sleep()** 不会出现在分析的输出中，因为对 R 语言来说，它不消耗任何计算时间。)

如果我们对 **f()** 的执行过程进行分析，每 0.1 秒暂停执行代码一次，那么我们会看到像下面这样的分析。每一行代表了分析器的一个“记录点”(tick)(在这种情况下是 0.1 秒)，并且**函数调用**使用 **>** 进行嵌套。它显示出，运行 **f()** 的代码花了 0.1 秒，然后运行 **g()** 花了 0.2 秒，然后运行 **h()** 花了 0.1 秒。

```
f()
f() > g()
f() > g() > h()
f() > g() > h() >
```

```
f() > g() > h()
f() > h()
```

如果我们使用下面的代码对 `f()` 进行分析，那么我们不可能得到这么清晰的结果。

```
tmp <- tempfile()
Rprof(tmp, interval = 0.1)
f()
Rprof(NULL)
```

这是因为，如果没有把你的代码运行速度减缓几个数量级，那么分析是很难精确地进行下去的。`RProf()`使用的折衷是，**抽样**，对整体性能只有最小程度的影响，但是，从根本上是随机的。在**计时器的精度**和**每个操作的时间**之中都具有可变性，所以每当你进行分析的时候，你都会得到一个稍微不同的答案。幸运的是，对于确定代码的最慢部分来说，精确性并不是那么必要。(译者注：意思是只要能大致比较出时间的长短就可以了。)

我们将使用 `lineprof` 包对**总体**进行**可视化**，而不是关注**单个调用**。还有很多其它的选择，比如 `summaryRprof()`、`proftools` 包和 `profr` 包，但是这些工具超出了本书的范围。我编写 `lineprof` 包，是为了以一种简单的方式来对分析数据进行可视化。顾名思义，`lineprof()`中的基本分析单元是一个代码行。这使得 `lineprof` 的精度比其它替代方法要低(因为一行代码可以包含多个函数调用)，但是理解上下文会变得更加容易。

为了使用 `lineprof`，我们首先将代码保存在一个文件中，并用 `source()` 进行加载。在这里，`profiling-example.R` 包含了 `f()`、`g()` 和 `h()` 的定义。请注意，你必须使用 `source()` 加载代码。这是因为 `lineprof` 使用了 `srcrefs` 使得**代码**与**分析**进行匹配，并且所需要的 `srcrefs` 只有当你从磁盘加载代码时才会创建出来。然后，我们使用 `lineprof()` 来运行我们的函数，并且捕获时间输出。打印这个对象展示了一些基本信息。现在，我们只需要关注 **time** 列，它估计每一行运行了多长时间，以及 **ref**

列，它告诉我们哪个代码行正在运行。时间的估计是不完美的，但是比例看起来是正确的。

```
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 提供了一些函数来浏览这些数据结构，但是它们不是那么灵活。因此，我们将使用 **shiny** 包来启动一个交互式浏览器。**shine(l)** 将打开一个新网页(或者如果你使用的是 RStudio，那么会打开一个新的面板)，它显示了你的每一行源代码运行了多长时间的信息。**shine()** 启动一个 **shiny** 应用程序，它会“阻塞”(block)你的 R 会话(session)。要退出时，你需要使用 Esc 键或者 ctrl + c 来停止这个进程。

#	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	}				

在 **t** 列中可以看到在每一行上花了多少时间。(你会在第 18.3 节了解到其它列)。虽然不太精确，但是它可以让你发现性能瓶颈，并且你可以把鼠标悬停在每一栏上得到精确的数字。这里表明了 **g0** 花费了相当于 **h0** 的两倍的时间，所以需要深入到 **g0** 之中查看更多细节。为此，单击 **g0**：

#	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	}				

然后单击 **h0**：

#	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	}				

这项技术应该可以让你快速辨认出你的代码中的主要瓶颈。

17.1.1 局限性

当然，分析有一些其它的局限性：

分析不能扩展到 C 语言代码。你可以看到你的 R 代码是否调用了 C/C++ 代码，但是看不到你的 C/C++ 代码内部调用了什么函数。不幸的是，编译型代码的分析工具超出了本书的范围(其实我不知道怎么做)。

同样的，你也不能看到原语函数内部或者编译成字节码的代码。

如果你使用匿名函数来进行大量的函数式编程，那么很难确切地找出正在调用哪些函数。解决这个问题最简单的方法，是对你的函数进行命名。

延迟计算意味着参数通常是在另一个函数内部进行计算的。例如，在下面的代码中，分析使它看起来像 `i0` 被 `j0` 调用，因为直到被 `j0` 需要的时候，参数才会被计算出来。

```
i <- function() {  
  pause(0.1)  
  10  
}  
  
j <- function(x) {  
  x + 10  
}  
j(i0)
```

如果这个让你感到迷惑，那么你可以创建临时变量来进行强制计算。

17.2 改善性能

"我们应该忘掉那些小小的效益，它们大约占 97%的时间：过早的优化是万恶之源。然而，我们不应该在那关键的 3%上放弃我们的机会。一个优秀的程序员不会因为这种原因而自满，他将会明智地仔细查看关键的代码；但是，这仅仅是在该代码已经确定的情况下。" — Donald Knuth. (译者注：意思是只有在代码基本上已经定型的情况下，才应该考虑性能问题。代码在频繁变动的时候，是不适合进行性能优化的。)

一旦你使用分析技术识别出了性能瓶颈，你就需要使它变得更快。下面的小节会向你介绍一些技巧，而我发现这些技巧的应用很广泛：

1. 寻找现有的解决方案。
2. 少做一些工作。
3. 向量化。
4. 并行化。
5. 避免复制。
6. 编译成字节码。

终极方法是使用更快的语言进行重写，比如 C++。这是一个很大的话题，将在第 19 章进行讲述。在我们进入特定的技术之前，我将首先描述一些**一般策略和代码组织风格**，它们在提高性能方面通常是有用的。

17.3 代码组织

在试图让你的代码变得更快的时候，很容易落入两个陷阱之中：

1. 编写出更快的代码，但是错误的代码。

2. 编写出你认为更快的代码，但是实际上并没有变得更好。

下面列出的策略将帮助你避免这些陷阱。在处理性能瓶颈的时候，你可能会想出多个方法。对于每一种方法，都要编写一个函数来封装所有相关的行为。这使得检查每种方法是否返回了正确的结果变得更容易，也使得记录每种方法需要运行多长时间变得更简单。为了演示这种策略，我将比较两种计算均值的方法：

```
mean1 <- function(x) mean(x)
mean2 <- function(x) sum(x) / length(x)
```

我建议你记录你尝试过的一切方法，甚至连失败也应该记录下来。如果在未来出现类似的问题，那么看看你尝试过的一切方法将是有益的。为此我经常使用 R Markdown，在其中，可以很容易地把代码与详细的注释和标注组合起来。

接下来，生成一个有代表性的测试用例。这个测试用例应该足够大，足够捕获你的问题的本质，但是又要足够小，运行只需要几秒钟。你不会想让测试用例花太多时间，因为你需要多次运行测试用例来比较不同的方法。在另一方面，你不也希望测试用例太小，因为测试的结果可能达不到真实问题的规模。

使用这个测试用例快速检查所有的不同方法是不是都返回了相同的结果。一个简单的方法是使用 `stopifnot()` 和 `all.equal()`。对于可能的输出情况比较少的实际问题，你可能需要更多的测试用例，来确保某种方法并不是偶然返回了正确的答案。对于均值来说，这是不太可能的。

```
x <- runif(100)
stopifnot(all.equal(mean1(x), mean2(x)))
```

最后，使用 `microbenchmark` 包比较每种方法需要运行多长时间。对于更大的问题，可以减少 `times` 参数，使得运行只需要几秒钟。关注中位数时间，并且利用高、低四分位数来衡量时间的变化。

```
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
```

(你也许会对结果感到惊讶：`mean(x)`比`sum(x) / length(x)`要慢得多。这是因为，由于某些原因，`mean(x)`对向量进行了两次遍历，使得数值更加准确。)

在你开始实验之前，你应该有一个目标速度，它定义了达到什么速度的时候，性能瓶颈就不再是一个问题。设置这样一个目标是很重要的，因为你不会希望把宝贵的时间花在**过度优化**你的代码上。

如果你想看看这种策略的实际应用，那么我在 `stackoverflow` 上应用过几次：

<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？

一旦你组织好了代码，并且尝试了所有你能想到的方法，那么看看别人所做的工作是很自然的。你是一个大型社区的一部分，所以很可能有人已经解决了同样的问题。如果你的性能瓶颈是某个包中的一个函数，那么看看其它做同样的事情的包是值得。两个开始的好地方是：

1. CRAN 任务视图(<http://cran.rstudio.com/web/views/>)。如果有一个 CRAN 任务视图与你的问题领域是相关的，那么那里列出的包值得看看。

2. **Rcpp** 的反向依赖(译者注: 即 **Rcpp** 包所依赖的包, 那些包都是 C++编写的, 性能很高), 列在了它的 CRAN 页面上(<http://cran.r-project.org/web/packages/Rcpp>)。由于这些包使用了 C++, 所以使用它们有可能给你的性能瓶颈找到一个解决方案。

如果上述方法解决不了问题, 那么接下来的挑战就是通过某种方式来描述你的性能瓶颈, 以帮助你找到相关的问题和解决方案。如果你知道问题的名字或者同义词, 那么搜索将会更加容易。但是, 因为你不知道问题的名字叫什么, 所以往往很难进行搜索! 通过广泛阅读有关统计学和算法的相关内容, 随着时间的推移, 你可以建立自己的知识库。另外, 你还可以问问别人。跟你的同事进行一次头脑风暴, 想出一些可能的名字, 然后在 Google 和 stackoverflow 上进行搜索。通常, 把你的搜索限制在 R 语言相关的页面上是有用的。对于 Google, 可以试试 rseek(<http://www.rseek.org/>)。对于 stackoverflow, 在你的搜索中, 把 R 标签 **[R]** 包括进去。

正如上面所讨论的那样, 把你找到的所有解决方案都记录下来, 而不仅仅是那些立即能让性能似乎变得更快的解决方案。一些解决方案在最初的时候可能是比较慢的, 但是由于它们也许更容易优化, 所以最终会变得更快。你也可以把通过不同的方法获得的最快部分结合起来。如果你找到了一个足够快的解决办法, 那么要恭喜你! 如果合适的话, 那么你可能想要与 R 社区分享你的解决方案。如果没有找到解决方法, 那么请继续阅读后面的章节。

17.4.1 练习

1. **lm** 函数的更快的替代函数是什么? 哪些是专门为了处理更大的数据集而设计的?
2. 什么包实现了 **match()** 函数的另一个版本, 它对于重复查找时会更快吗? 快多少?

3. 列出四个函数(不仅仅是基础 R 语言中的函数)，它们把字符串(string)转换为日期时间(date time)对象。它们的优缺点分别是什么？
4. 在 R 语言中有多少种不同的方法可以计算出一维密度估计(1d density estimate)？
5. 哪些包提供了计算滚动均值(rolling mean)的能力？
6. `optim()`函数的替代函数是什么？

17.5 尽量少做

让一个函数变得更快的最简单的方法是让它做更少的工作。一种这样做的方法，就是让函数的**输入或输出**限制在某些特定类型，或者让函数处理**更具体**的问题。例如：

`rowSums()`、`colSums()`、`rowMeans()`和 `colMeans()`的速度比使用 `apply()`进行等价的操作要快得多，因为它们是**向量化**的(下一节的话题)。

`vapply()`的速度比 `sapply()`要快，因为它预先规定了输出类型。

如果你想看看某个向量是否包含一个值，那么 `any(x == 10)`速度远远超过 `10 %in% x`。这是因为**测试相等**要比**测试集合的包含关系**更简单。

要信手拈来这些知识，要求你知道可以选择的函数是存在的：你需要有一张很好的函数词汇表。从第四章开始，然后通过定期阅读 R 语言代码来扩大你的词汇量。阅读代码的好地方是 R-help 邮件列表(<https://stat.ethz.ch/mailman/listinfo/r-help>)和 `stackoverflow`(<http://stackoverflow.com/questions/tagged/r>)。

一些函数强制其输入为某种特定的类型。如果你的输入不是正确的类型，那么函数需要做额外的工作。因此，找到针对你的数据的那个函数，或者考虑改变存储你的数据的方式。这个问题的最常见的例子是对数据框使用 `apply()`。`apply()`总是

把它的输入转化成矩阵。这样不仅很容易出错(因为数据框比矩阵更加一般化),而且也很慢。

还有一些函数,如果你给了它们更多关于问题的信息,那么它们将会做更少的工作。仔细阅读函数的文档,并且尝试一下不同的参数往往是很值得的。我在过去发现了一些例子,包括:

1. `read.csv()`: 使用 `colClasses` 来指定已知的列类型。
2. `factor()`: 使用 `levels` 指定因子的水平。
3. `cut()`: 如果你不需要标签,那么使用 `labels = FALSE` 可以不产生标签,或者,更好的办法,是使用在文档的"see also"一节提到的 `findInterval()` 函数。
4. `unlist(x, use.names = FALSE)` 比 `unlist(x)` 要快得多。
5. `interaction()`: 如果你只需要数据中存在的组合,那么可以使用 `drop = TRUE`。

有时,你可以通过避免方法分派来使函数变得更快。就像我们在 16.3.1 节中看到的那样,R 语言中方法分派的开销很大。如果你在一个封闭的循环中调用一个方法,那么你可以通过只做一次方法查找来避免一些开销:

7. 对于 S3,你可以通过调用 `generic.class()` 而不是 `generic()` 的方式来调用方法。
8. 对于 S4,你可以通过使用 `findMethod()` 来找到方法,把它保存到一个变量中,然后调用该函数。

例如,对于小向量来说,调用 `mean.default()` 比调用 `mean()` 要快得多:

```
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
```

这种优化有些风险。虽然 `mean.default()` 几乎要快两倍，但是如果 `x` 不是一个数值向量，那么它会以意外的方式失败。只有你确切地知道 `x` 是什么，你才应该使用。

知道你正在处理一种特定类型的输入，是编写更快代码的另一种方法。例如，`as.data.frame()` 非常缓慢，因为它把每个元素都强制转换成数据框，然后使用 `rbind()` 把它们连接在一起。如果你有一个命名列表，列表内有等长的向量，那么你可以直接将其转换成数据框。在这种情况下，如果你能够对你的输入做出确定的假设，那么你可以编写出一个比默认方法快 20 倍的方法。

```
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
```

同样地，注意进行权衡。这种方法确实快，但是它是危险的。如果你给它传入了错误的输入，你将会得到一个糟糕的数据框：

```
quickdf(list(x = 1, y = 1:2))  
#> Warning: corrupt data frame: columns will be truncated or  
#> padded with NAs  
#> x y  
#> 1 1 1
```

为了想出这个最小的方法，我仔细阅读并且重写了 `as.data.frame.list()` 和 `data.frame()` 的源代码。我进行了许多小的修改，每一次修改之后，我都会检查一下，确保我没有破坏现有的行为。（译者注：即每次修改的时候都要进行回归测试。）经过几个小时的工作，我能够隔离出上面所示的最小代码了。这是一种非常有用的技术。大多数基础 R 函数都是为了灵活性以及功能而编写的，而不是性能。因此，对于你的特定需求进行重写，通常可以产生实质性的改进。要做到这一点，你需要阅读源代码。也许会很复杂和让人迷惑，但是请不要放弃！

如果你只需要计算相邻值之间的差异，那么下面的示例显示了 `diff()` 函数的逐步简化过程。在每一步中，我都用一种特例来替换一个参数，然后进行检查，确保函数仍然能工作。原始函数很长也很复杂，但是通过限制参数，我不仅让它快了两倍，也让它更加容易理解。

首先，我把 `diff()` 的代码拿过来，并且用我的风格重新进行格式化：

```
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，我假设输入是向量。这让我可以消除 `is.matrix()` 的测试，以及使用了矩阵取子集操作的方法。

```
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
```

我现在假设 `difference = 1L`。这简化了输入检查并且消除了 `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，我假设 `lag = 1L`。这就消除了输入检查并且简化了取子集操作。

```
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()` 简单得多也快得多：

```
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
```

当你读了第 19 章之后，对这种情况，你甚至还可以让 `diff()` 变得更快。

做更少的工作的最后一个例子，是使用更简单的数据结构。比如，在处理数据框的行的时候，通常使用行索引比直接使用数据框更快。又比如，如果你想计算数据框中两列之间的相关性的自助法估计(bootstrap estimate of the correlation)，那么有两种基本方法：你可以对整个数据框进行操作，或者对每个向量进行操作。下面的例子表明，对向量进行操作，要快两倍。

```
sample_rows <- function(df,i) sample.int(nrow(df), i,
replace = TRUE)
# 生成一个包含随机选取的行的新数据框
boot_cor1 <- function(df,i) {
sub <- df[sample_rows(df,i), , drop = FALSE]
cor(sub$x, sub$y)
}
# 从随机行中产生新向量
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
```

17.5.1 练习

1. 如果你比较对于 10000 个观测、而不是 100 个观测，调用 `mean()` 和 `mean.default()`，那么结果会如何变化？
2. 下面的代码提供了 `rowSums()` 的一种替代实现。为什么这种输入会更快？

```
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()` 有什么区别?
4. 实现一个更快版本的 `chisq.test()`，它只计算当输入是两个数值向量并且没有缺失值时的卡方检验统计量(chi-square test statistic)。你可以试着简化 `chisq.test()` 或者根据数学定义来编写代码 (http://en.wikipedia.org/wiki/Pearson%27s_chi-squared_test)。
5. 你能为这种情况实现一个更快版本的 `table()` 函数吗？它的输入是两个整数向量并且没有缺失值。你能用它来提高你的卡方检验的速度吗？
6. 想象一下，你想使用 `cor_df()` 来计算一个样本相关性的自助法分布(bootstrap distribution of a sample correlation)，并且数据在下面的示例中。如果你想运

行它很多次，那么怎样才能使这段代码更快？(提示：函数有三个你可以加快的组件)。

```
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]
}
```

有办法让这个过程向量化吗？

17.6 向量化

如果你使用了 R 语言一段时间，那么也许你已经听说过"让你的代码向量化"的说法。但是，这究竟是什么意思呢？**向量化**你的代码不仅仅是避免循环，尽管这通常是其中的一步。**向量化**是对问题采取操作"整体对象"(whole object)的方法，考虑的是向量，而不是标量。**向量化**的函数有两个关键属性：

它使得许多问题变得更简单。你只用考虑整个向量，而不用考虑向量的组成部分。

向量化的函数中的**循环**是用 C 语言编写的，而不是 R 语言。**循环**在 C 语言中要快得多，因为开销要小得多。

第 11 章强调了向量化代码作为一种更高层次抽象的重要性。向量化对于写出快速的 R 代码也很重要。这并不意味着仅仅简单地使用 `apply()` 和 `lapply()`，或者 `Vectorise()`。这些函数只是改善了函数的接口，但是并不会从根本上改变性能。对于性能问题而使用向量化，意味着要找到那些现有的 R 函数，它们是由 C 语言实现的，并且最适用于你的问题。

适用于许多常见性能瓶颈的向量化函数，包括以下这些：

`rowSums()`、`colSums()`、`rowMeans()`和 `colMeans()`。这些向量化的矩阵函数总是会比使用 `apply()`更快。你有时可以使用这些函数来构建其它向量化函数。

```
rowAny <- function(x) rowSums(x) > 0
rowAll <- function(x) rowSums(x) == ncol(x)
```

向量化的**取子集操作**可以有巨大的性能提升。回忆一下**查找表**(3.4.1 节)背后的技术以及**手工匹配与合并**(3.4.2 节)。再回忆一下，你可以使用**取子集操作**的方式进行赋值，以便在单一步骤中替换多个值。如果 `x` 是一个向量、矩阵或数据框，那么 `x[is.na(x)] <- 0` 将把所有的缺失值替换为 0。

如果你在一个矩阵或者数据框中，**提取**或者**替换**位置分散的值，那么可以使用整数矩阵来进行**取子集操作**。在第 3.1.3 节中可以看到更多的细节。

如果你要将**连续值**转换为**分类数据**，那么确保你知道如何使用 `cut()`和 `findInterval()`。

注意像 `cumsum()`和 `diff()`这样的向量化函数。

矩阵代数是向量化的一个一般示例。其中，**循环**是由像 BLAS 这样高度调整的外部库进行执行的。如果你能找到办法使用**矩阵代数**来解决你的问题，那么你常常会得到一个快速的解决方案。使用**矩阵代数**来解决问题的能力是来自于经验的。虽然这个技能是随着时间的推移慢慢发展起来的，但是一个很好的起点是问问你所在的领域中有经验的人。

向量化的缺点是，难以预测操作达到的规模。下面的示例，测量了使用字符**取子集操作**从一个列表中查找 1 个、10 个和 100 个元素分别需要多长时间。你可能会认为，查找 10 个元素需要 10 倍于查找 1 个元素的时间，查找 100 个元素需要 10 倍于查找 10 个元素的时间。事实上，下面的例子表明，查找 100 个元素只需要大约 9 倍于查找 1 个元素的时间。

```
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
```

向量化不能解决所有的问题，所以你最好经常使用 C++ 来编写自己的向量化函数，而不是把现有的算法折腾成使用向量化的方法。你将在第 19 章学习怎么做。

17.6.1 练习

1. 密度函数，如 `dnorm()`，有一个共同的接口。哪些参数是向量化的？
`rnorm(10, mean = 10:1)` 做了什么？
2. 基于不同大小的 `x`，比较 `apply(x, 1, sum)` 和 `rowSums(x)` 的速度。
3. 怎样使用 `crossprod()` 来计算加权和 (weighted sum)？它比简单的 `sum(x * w)` 快多少？

17.7 避免复制

缓慢的 R 代码的一种重要来源，是在循环中不断扩展一个对象。每当你使用 `c()`、`append()`、`cbind()`、`rbind()` 或 `paste()` 来创建一个更大的对象的时候，R 语言必须

首先为新对象分配空间，然后再把旧的对象复制进去。如果你重复这种步骤很多次，比如在一个 **for** 循环中，那么这样的开销是相当大的。你已经进入了《R 地狱》(《R inferno》)中的第 2 圈(Circle 2)(http://www.burns-stat.com/pages/Tutor/R_inferno.pdf)。

这里有一个小例子来说明这个问题。我们首先生成一些随机字符串，然后使用 **collapse()** 将它们在循环中迭代地连接在一起，或者使用 **paste()** 把它们连接在一起。注意，**collapse()** 的性能随着字符串数量的增加而相应地变得更差：连接 100 个字符串所需要的时间，是连接 10 个字符串所需要的时间的近 30 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
```

在循环中修改对象，比如 `x[i] <- y`，根据 `x` 的类，也可能会创建一个副本。第 18.4 节将更深入地讨论这个问题，并且给你一些工具来确定你在什么时候创建了副本。

17.8 编译成字节码

R 2.13.0 引入了一个**字节码编译器**，它可以提高一些代码的速度。使用这个编译器是一种提高速度的简单方法。即使它并不适合你的函数，你也不需要它身上投入大量的时间。下面的例子展示了第 11.1 节中的纯 R 版本的 `lapply()` 函数。虽然比不上由基础 R 语言提供的 C 语言版本的函数，但是把它编译成字节码，还是对它的速度有了相当大的提高。

```
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
```

编译成字节码在这里确实起到了作用，但是在大多数情况下，你更可能只得到 5%-10% 的改善。在默认情况下，所有的基础 R 语言函数都是编译成字节码的。

17.9 案例研究：t 检验

下面的案例研究展示了如何使用上面描述的一些技巧使得 t 检验更快。它是基于由 Holger Schwender 和 Tina Müller 写的《Computing thousands of test statistics simultaneously in R》(<http://statcomputing.org/newsletter/issues/scgn-18-1.pdf>) 中的一个例子。我强烈推荐完整地阅读这篇论文，看看如何把同样的思想应用到其它测试案例上。

想象我们已经运行了 1000 次试验(行)，每次试验收集了 50 个人(列)的数据。在每次实验中，前 25 个人被分配到第 1 组，其余的分配到第 2 组。首先，我们生成一些随机数据来表示这个问题：

```
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，使用 `ttest()` 有两种方法。我们可以使用公式(formula)接口或者提供两个向量，每组一个向量。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，一个 `for` 循环会计算值，但并不保存值。我们将使用 `apply()` 来做这个。这会增加一点点开销：

```
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
```

我们怎样才能使这个更快呢？首先，我们可以尝试做更少的工作。如果你看看 `stats::t.test.default()` 的源代码，你将看到它做的工作，远远多于计算 `t` 统计值。它还会计算 `p` 值，并且对打印输入进行格式化。我们可以尝试剔除那些代码来使我们的代码更快。

```
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))
```

这样让我们的速度提高了大约 6 倍。现在，我们有了一个相当简单的函数，我们还可以通过向量化让它变得更快。我们将修改 `t_stat()` 函数，使它工作于矩阵的值，而不是在函数外对数组进行循环。因此，`mean()` 变成了 `rowMeans()`，`length()` 变成了 `ncol()` 以及 `sum()` 变成了 `rowSums()`。代码的其余部分保持不变。

```
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))
```


这样快得多！这比我们之前的方法至少快了 40 倍，而比我们开始的时候快了大概 1000 倍。

最后，我们可以试试字节码编译。在这里我们需要使用 `microbenchmark()`，而不是 `system.time()`，因为为了看出区别，需要有足够的精度：

```
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
```

在这个例子中，编译成字节码根本没有起到什么作用。

17.10 并行化

并行化是对一个问题的不同部分使用多个 CPU 内核同时进行计算。它不会减少计算时间，但是它能节省你的时间，因为你使用了计算机上更多的资源。并行计算是一个复杂的话题，在这里没有办法进行深入探讨。我推荐一些资源：

Q. Ethan McCallum 和 Stephen Weston 写的《Parallel R》

(<http://amazon.com/B005Z29QT4>)。

Norm Matloff 写的《Parallel Computing for Data Science》

(<http://heather.cs.ucdavis.edu/paralleldatasci.pdf>)。

我想要展示的是并行计算的一种简单的应用，它是所谓的"理想并行问题"。理想并行问题是由许多可以独立解决的简单问题组成的。一个很好的例子是 `lapply()`，因为它独立地操作每个元素。在 Linux 和 Mac 系统上，很容易对 `lapply()` 进行并行化，因为你只需要简单地把 `lapply()` 替换成 `mclapply()` 即可。下面的代码片段在你的计算机所有 CPU 内核上运行了一个微小(但是缓慢)的函数。

```
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
```

在 Windows 系统上会复杂一点。首先，需要建立一个本地集群(cluster)，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()` 之间的主要区别是，由 `mclapply()` 产生的进程继承自当前进程，而由 `makePSOCKcluster()` 产生的进程则开始了一个新的会话。这意味着，大多数真正的代码将需要一些设置。在每个集群上使用 `clusterEvalQ()` 运

行任意代码，并且加载需要的包，以及使用 `clusterExport()` 把当前会话中的对象复制到远程会话中。

```
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
```

并行计算有一些通信开销。如果子问题非常小，那么并行化可能会更差，而不是更好。也可以通过计算机网络上进行分布式计算，(不仅仅是使用你的本地计算机上 CPU 内核)，但是这超出了本书的范围，因为在计算和通信的开销之间进行平衡，会变得越来越复杂。获得更多信息的一个好地方，是 CRAN 任务视图上关于高性能计算的内容(<http://cran.r-project.org/web/views/HighPerformanceComputing.html>)。

17.11 其它技术

能够写出快速的 R 代码是一个好的 R 程序员应该具有的能力。除了本章中给出的一些提示之外，如果你想写出快速的 R 代码，那么你需要提高通用编程技能。一些方法是：

1. 阅读 R 语言博客(<http://www.r-bloggers.com/>)，看看别人遇到过什么性能问题，以及他们如何使他们的代码变得更快。
2. 阅读其它的 R 语言编程书籍，比如像 Norm Matloff 的《The Art of R Programming》(中文版《R 语言编程艺术》，<http://amazon.com/1593273843>)，或者 Patrick Burns 的《R Inferno》

(<http://www.burns-stat.com/documents/books/the-r-inferno/>), 来了解常见的陷阱。

3. 参加算法和数据结构课程, 来学习一些处理某些类型问题的著名方法。我在 **Coursera** 上提供的普林斯顿大学的算法课程中学到了不少好东西 (<https://www.coursera.org/course/algs4partI>)。
4. 阅读通用书籍来了解优化技术, 比如 Carlos Bueno 写的《Mature optimisation》(<http://carlos.bueno.org/optimization/mature-optimization.pdf>), 或者 Andrew Hunt 和 David Thomas 写的《Pragmatic Programmer》(<http://amazon.com/020161622X>)。

你也可以到社区寻求帮助。Stackoverflow 是一个有用的资源。你需要努力创建一个规模较小、但是抓住了你的问题本质的例子。如果你的例子太复杂了, 那么很少有人会有时间和意愿来帮你找出一个解决方案。如果例子太简单了, 那么你能得到解决玩具(toy)问题的答案, 而不是真正的问题的答案。如果你也尝试着在 stackoverflow 上回答问题, 那么你很快就会了解到什么是好问题。

18 内存

对 R 语言的内存管理有着深刻的理解，会帮助你预测，对于一个给定的任务，你需要多少内存，以及帮助你最大限度地使用内存。它甚至可以帮助你编写更快的代码，因为**意外的复制**是代码缓慢的主要原因。本章的目的是帮助你了解 R 语言中内存管理的基本知识，从**单个对象**，到**函数**，再到**更大的代码块**。在这个过程中，你将了解一些常见的误解，比如**你需要调用 `gc()` 来释放内存**，或者 **`for` 循环总是缓慢的**。

本章概要

18.1 节向你展示了如何使用 `object_size()` 查看对象占用了多少内存，并以此为基点，提高你对 R 语言的对象是如何存储在内存中的理解。

18.2 节介绍了 `mem_used()` 和 `mem_changed()` 函数，这将帮助你理解 R 语言如何分配和释放内存。

18.3 节向你展示了如何使用 `lineprof` 包来理解，在更大的代码块中，内存是如何分配和释放的。

18.4 节介绍了 `address()` 和 `refs()` 函数，使你能理解 R 语言在什么时候进行**就地修改**，以及 R 语言在什么时候进行**复制**。理解对象在什么时候会被复制，对编写高效的 R 代码是非常重要的。

前提条件

在本章中，我们将使用 `pryr` 和 `lineprof` 包中的工具来理解内存使用情况，并使用 `ggplot2` 包中的样本数据集。如果你还没有安装，运行此代码来获得你需要的包：

```
install.packages("ggplot2")
install.packages("pryr")
devtools::install_github("hadley/lineprof")
```

资源

R 语言内存管理的细节文档不是只在一个地方。本章中的大部分信息，收集于 R 语言帮助文档(特别是?**Memory** 和?**gc**)，《R-exts》中的**内存分析**部分(<http://cran.r-project.org/doc/manuals/Rexts.html#Profiling-R-code-for-memory-use>)以及《R-ints》中的 SEXPs 部分(<http://cran.r-project.org/doc/manuals/R-ints.html#SEXPs>)。其它信息是我通过阅读 C 源代码、做小实验以及在 R-devel 上提问得到的。如果有任何错误，那么完全是我造成的。

18.1 对象的大小

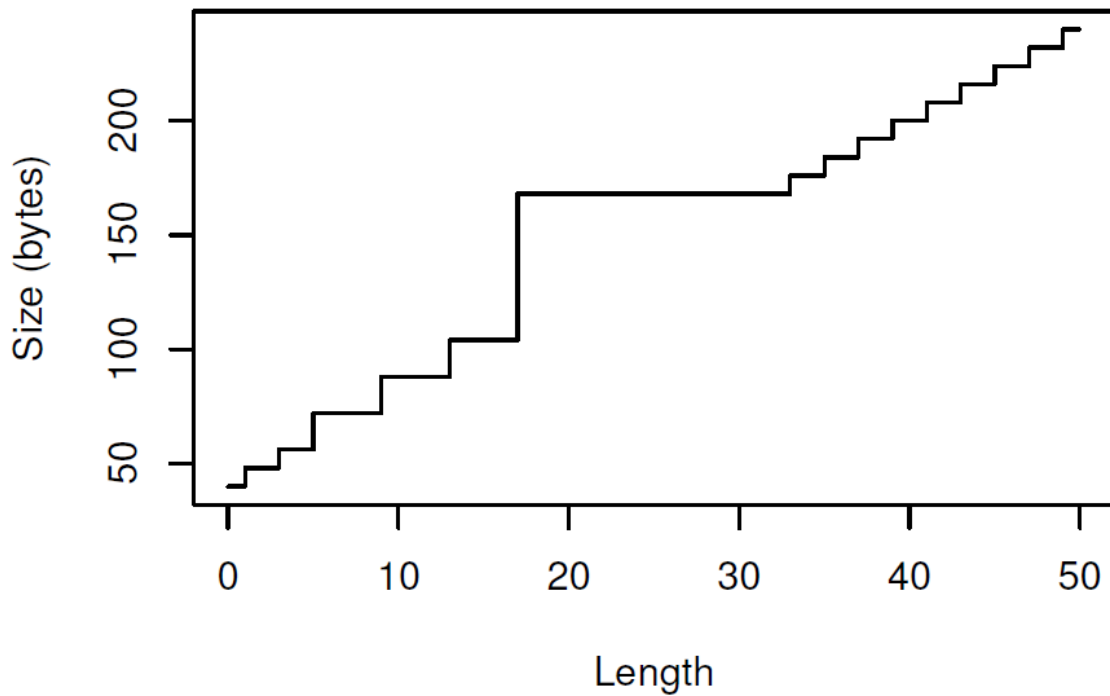
为了理解 R 语言的内存使用情况，我们将从 `pryr::object_size()` 开始。这个函数会告诉你一个对象占用了多少个字节的内存：

```
library(pryr)
object_size(1:10)
#> 88 B
object_size(mean)
#> 832 B
object_size(mtcars)
#> 6.74 kB
```

(这个函数比内置的 `object.size()` 函数更好，因为它会解释(account for)一个对象内的共享元素，并且包括**环境**的大小。)

如果我们使用 `object_size()` 系统地探索一个整数向量的大小，那么会发生一些有趣的事情。下面的代码计算了长度为 0 到 50 个元素的整数向量的内存使用情况，并绘制了图形。你可能认为一个**空向量**的大小应该是零，以及**内存的使用量**应该与**长度**成正比例增长。但是，这些都不是真的！

```
sizes <- sapply(0:50, function(n) object_size(seq_len(n)))  
plot(0:50, sizes, xlab = "Length", ylab = "Size (bytes)",  
type = "s")
```



这不仅仅是整数向量的现象。每一个长度为 0 的向量都会占用 40 个字节的内存：

```
object_size(numeric())
```

```
#> 40 B
```

```
object_size(logical())
```

```
#> 40 B
```

```
object_size(raw())
```

```
#> 40 B
```

```
object_size(list())
```

```
#> 40 B
```

这 40 个字节，被用于存储 R 语言中的每一个对象都拥有的四个组件：

1. 对象的元数据(4 个字节)。这些元数据存储了**基本类型**(比如, 整数 integer), 以及用于**调试**和**内存管理**的信息。
2. 两个指针(pointer): 一个指向**内存中的下一个对象**, 另一个指向**前面的对象**(2 * 8 个字节)。这种**双链表**使得内部的 R 语言代码对内存中的每个对象进行遍历变得很容易。
3. 指向**属性**的指针(8 个字节)。

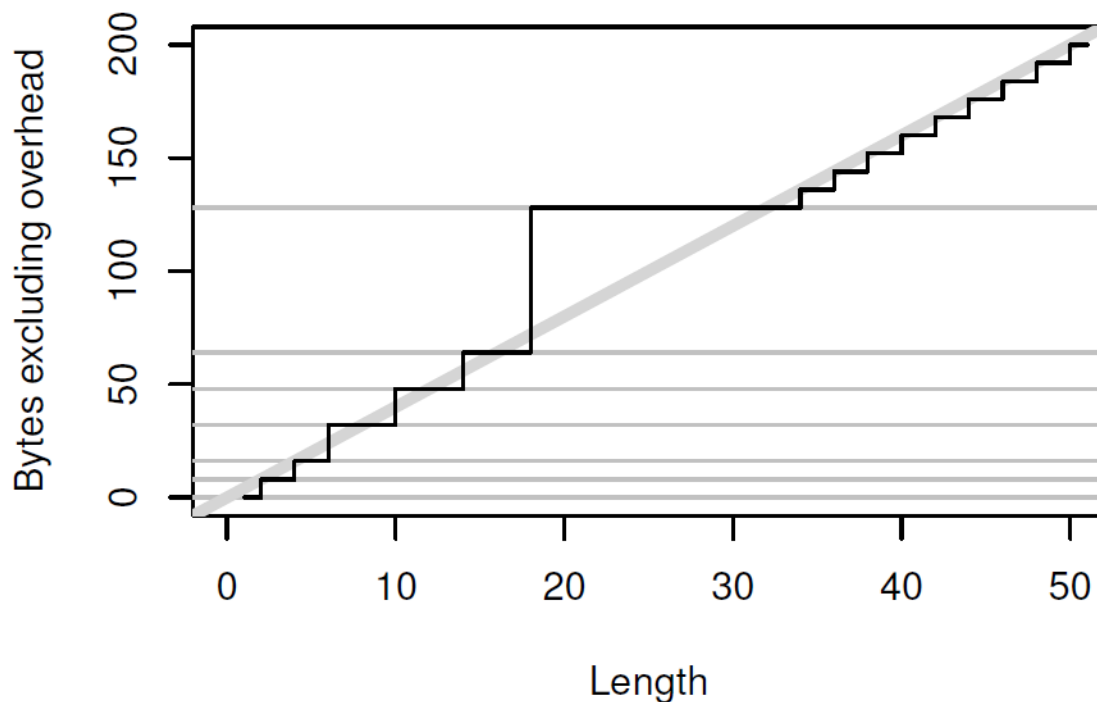
所有的向量都有三个附加组件:

4. 向量的长度(4 个字节)。这里只使用了 4 个字节, 你可能认为 R 语言只能支持向量中最多包含 $2^{(4 \times 8 - 1)}$ (2^{31} , 大约二十亿)个元素。但在 R 3.0.0 以及之后的版本中, 实际上你可以创建包含 2^{52} 个元素的向量。请阅读《R-internals》(<http://cran.rproject.org/doc/manuals/R-ints.html#Long-vectors>), 看一看不改变这个字段的大小, 而增加对**长向量**的支持, 是如何做到的。
5. 向量的"真"长度(4 个字节)。这个基本上是从未用到的, 除了对象是由**环境使用的哈希表**之外。在这种情况下, "真"长度代表**分配的空间**, 而长度代表**当前使用的空间**。
6. 数据(?? 字节)。一个**空向量**有 0 字节的数据, 但是它显然是很重要的! **数值向量**的每个元素占用 8 个字节, **整数向量** 4 个字节, **复数向量** 16 个字节。

如果你仔细地数一数, 那么你会注意到这里最多只有 36 个字节。剩下的 4 个字节是用于**对齐(padding)**的, 这样每个组件都是从一个 8 字节(= 64 位)边界开始。大多数 CPU 的架构都要求**指针**以这种方式对齐, 就算它们没有这种要求, 访问没有对齐的指针也会相当缓慢。(如果你感兴趣, 你可以在《C structure packing》(<http://www.catb.org/esr/structure-packing/>)中阅读更多相关内容。)

这就解释了图形上截断的原因。但是，为什么内存的大小是不规则增长的呢？要理解为什么，你需要了解 R 语言是如何从操作系统请求内存的。**请求内存**(使用 `malloc()`) 是一个开销相对较大的操作。如果每次创建一个小向量的时候，都不得不请求内存，那么会使 R 变得相当缓慢。相反，R 会请求一大块内存，然后自己管理那个内存块。这个块叫做“小向量池”，并用于长度不超过 128 个字节的向量。为了效率和简化，它只分配 8、16、32、48、64 或 128 个字节长的向量。如果我们调整之前的图形，删除 40 个字节的开销，那么我们可以看到这些值随着内存的使用而发生跳跃。

```
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")
```



超过了 128 个字节，就不再适合让 R 来管理向量了。毕竟，分配大量内存是操作系统擅长的工作。超过 128 个字节，R 将请求 8 个字节整数倍的内存。这样可以确保良好的对齐。

对象大小的精妙之处，是组件可以被多个对象**共享**。例如，看看下面的代码：

```
x <- 1:1e6
object_size(x)
#> 4 MB
y <- list(x, x, x)
object_size(y)
#> 4 MB
```

`y` 并不是 `x` 的三倍大，因为 R 语言很聪明，不会复制 `x` 三次；相反，它只是**指向**现有的 `x`。单独看 `x` 和 `y` 的大小是一种误导。如果你想要知道它们一起占用了多少内存空间，那么你必须对它们一起调用 `object_size()`：

```
object_size(x, y)
#> 4 MB
```

在这种情况下，`x` 和 `y` 一起占用的内存空间数量，与 `y` 独自的占用量相同。但是，并非总是如此。如果没有共享组件，正如接下来的例子中显示的一样，那么你可以把每个组件的大小加在一起求出总的空间大小：

```
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
```

对于字符串，也提出了同样的问题，因为 R 语言有一个全局字符串池。这意味着，每个唯一的字符串都只存储在一个地方，因此字符向量占用的内存会比你预期的要少：

```
object_size("banana")
#> 96 B
object_size(rep("banana", 10))
#> 216 B
```

18.1.1 练习

1. 对数值、逻辑和复数向量，重复以上的分析。
2. 如果一个数据框有一百万行，并且有三个变量(两个数值和一个整数)，那么它将占用多少空间？首先根据理论计算出来，然后通过创建数据框并测量其大小来验证你的结果。
3. 比较下面两个列表中的元素大小。每个都包含基本上相同的数据，但是一个包含小字符串向量，而另一个包含单个的长字符串。

```
vec <- lapply(0:50, function(i) c("ba", rep("na", i)))
str <- lapply(vec, paste0, collapse = "")
```

4. 哪一个会占用更多的内存：一个因子(x)还是等价的字符向量(as.character(x))？为什么？
5. 解释 1:5 和 list(1:5)之间大小差异的原因。

18.2 内存使用和垃圾收集

`object_size()` 可以告诉你一个对象的大小，而 `pryr::mem_used()` 可以告诉你内存中所有对象占用的内存总和：

```
library(pryr)
mem_used()
#> 45.4 MB
```

由于一些原因，这个数字并不会与操作系统报告的内存使用量一致：

1. 它只包含由 R 语言创建的对象，而不包含 R 语言解释器本身。
2. R 和操作系统都是**延迟**的：直到实际需要内存的时候，它们才会请求内存。R 可能会**持有**内存，因为操作系统还没有要求收回。
3. R 计算被对象所占用的内存，但是由于已删除的对象，可能会有差距。这个问题被称为**内存碎片**。

`mem_change()` 是在 `mem_used()` 的基础之上构建的，它告诉你在代码执行期间的内存变化情况。正数表示 R 使用的内存存在增加，而负数表示在减少。

```
# 需要大概 4 MB 来存储 1 百万个整数
mem_change(x <- 1:1e6)
#> 4.01 MB
# 当我们删除它的时候，我们可以把内存收回。
mem_change(rm(x))
#> -4 MB
```

甚至于**什么都不做**的操作都会使用一点内存。这是因为 R 语言会跟踪你所做的一切。你可以忽略近似 2kB 的东西。

```
mem_change(NULL)
```

```
#> 1.47 kB
```

```
mem_change(NULL)
```

```
#> 1.47 kB
```

在一些语言中，你必须显式地删除未使用的对象，以便让它们的内存返回。(译者注：比如 C 语言中的 `free` 函数以及 C++ 中的 `delete` 函数都是用来释放内存的。) R 语言使用另一种方法：垃圾收集(garbage collection，简称 GC)。当一个对象不再使用的时候，垃圾收集机制会自动释放内存。它通过追踪有多少个名字指向每个对象来做到这一点，当没有名字指向一个对象的时候，它就会删除该对象。

```
# 创建一个大对象
```

```
mem_change(x <- 1:1e6)
```

```
#> 4 MB
```

```
# 让 y 也指向 1:1e6
```

```
mem_change(y <- x)
```

```
#> -4 MB
```

```
# 删除 x，没有内存被释放，因为 y 仍然指向 1:1e6
```

```
mem_change(rm(x))
```

```
#> 1.42 kB
```

```
# 现在，没有谁指向 1:1e6 了，内存可以释放了。
```

```
mem_change(rm(y))
```

```
#> -4 MB
```

尽管你可能已经在其它地方读到过，自己调用 `gc()` 是没有任何必要的。每当 R 需要更多空间的时候，R 都将自动运行垃圾收集；如果你想看到具体是什么时候，那么可以调用 `gcinfo(TRUE)`。你可能会调用 `gc()` 的唯一原因，是要求 R 把内存返回给操作系统。但是，这样可能是没有任何效果的：旧版本的 Windows 没有办法让程序把内存返回给操作系统。

垃圾收集机制负责释放不再使用的对象。然而，你需要意识到可能的**内存泄漏**问题。当你持续指向一个对象，却没有意识到这一点的时候，就会发生**内存泄漏**。在 R 语言中，两种主要的**内存泄漏**原因是**公式(formula)**和**闭包(closure)**，因为它们都捕获**封闭环境**。下面的代码显示了这种问题。在 **f1()** 中，**1:1e6** 只是在函数内部被引用的，所以当函数完成的时候，内存会被返回，并且净变化为 0。内存的净变化将是 0。**f2()** 和 **f3()** 都返回**捕获了环境的对象**，因此当函数完成的时候，**x** 不会被释放。

```
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 内存分析与 lineprof

`mem_change()` 捕获代码运行时，内存的净变化。但是，有时候我们可能希望测量增量变化。一种方法是使用内存分析(memory profiling)，每隔几毫秒捕获内存的使用情况。这个功能由 `utils::Rprof()` 提供，但它没有提供非常有用的显示结果。因此，我们将使用 `lineprof` 包(<https://github.com/hadley/lineprof>)。它由 `Rprof()` 支持，但是会通过更丰富的方式来显示结果。为了演示 `lineprof`，我们要研究一个极简单的、只有三个参数的 `read.delim()` 实现：

```
read_delim <- function(file, header = TRUE, sep = ";") {  
  # 通过读取第一行来确定字段的数量  
  first <- scan(file, what = character(1), nlines = 1,  
    sep = sep, quiet = TRUE)  
  p <- length(first)  
  # 把所有的字段都加载成字符向量  
  all <- scan(file, what = as.list(rep("character", p)),  
    sep = sep, skip = if (header) 1 else 0, quiet = TRUE)  
  # 把字符串转换成合适的类型(不要转成因子)  
  all[] <- lapply(all, type.convert, as.is = TRUE)  
  # 设置列名  
  if (header) {  
    names(all) <- first  
  } else {  
    names(all) <- paste0("V", seq_along(all))  
  }  
}
```

```
# 把列表转换成数据框  
as.data.frame(all)  
}
```

我们也将创建一个样本 csv 文件：

```
library(ggplot2)  
write.csv(diamonds, "diamonds.csv", row.names = FALSE)
```

使用 `lineprof` 很简单。用 `source()` 加载代码，对表达式应用 `lineprof()`，然后使用 `shine()` 来查看结果。请注意，你必须使用 `source()` 加载代码。这是因为 `lineprof` 使用 `srcrefs` 来匹配代码和运行次数。所需的 `srcrefs` 只是当你从磁盘加载代码的时候才会被创建出来。

```
library(lineprof)  
source("code/read-delim.R")  
prof <- lineprof(read_delim("diamonds.csv"))  
shine(prof)
```


#	Source code	t	r	a	d
1	# ---- read_delim				
2	read_delim <- function(file, header = TRUE, sep = ",") {				
3	# Determine number of fields by reading first line				
4	first <- scan(file, what = character(1), nlines = 1, se...				
5	p <- length(first)				
6					
7	# Load all fields as character vectors				
8	all <- scan(file, what = as.list(rep("character", p)), ...	████████	██	██	
9	skip = if (header) 1 else 0, quiet = TRUE)				
10					
11	# Convert from strings to appropriate types (never to f...				
12	all[] <- lapply(all, type.convert, as.is = TRUE)				
13					
14	# Set column names				
15	if (header) {				
16	names(all) <- first				
17	} else {				
18	names(all) <- paste0("V", seq_along(all))				
19	}				
20					
21	# Convert list into data frame				
22	as.data.frame(all)			███	███
23	}				

shine() 也将打开一个新网页(或者如果你在使用 RStudio, 那么是一个新的窗口), 它显示了你的源代码, 并带有内存使用情况的注释信息。 **shine()** 会启动一个 **shiny** 应用, 它会"阻塞"你的 R 会话。如果要退出, 可以按 Esc 键或者 ctrl + break。源代码的旁边, 有四列提供了代码性能的细节:

t, 这行代码所花费的时间(以秒为单位), (在 17.1 节解释)。

a, 这行代码分配的内存(MB, 兆字节)。

r，这行代码释放的内存(MB，兆字节)。虽然**内存分配**都是确定的，但是**内存释放**是随机的：它取决于**垃圾收集**什么时候运行。这意味着，**内存释放**只告诉你，在这一行之前，释放的内存不再需要了。

d，向量复制发生的数量。作为 R 的**修改时复制**语义的结果，当 R 拷贝一个向量时，向量复制就会发生。

你可以把光标悬停在任何的长条上，以获取精确的数字。在这个例子中，看看分配的内存，它告诉了我们多数的信息：

scan()分配了大约 2.5MB 的内存，这非常接近于文件在磁盘上的占用的 2.8MB 空间。你不能期望这两个数字是相同的，因为 R 不需要存储**逗号**，也因为**全局字符串池**将会节省一些内存。

转换**列**的操作又分配了 0.6MB 的内存。你会期望这一步会释放一些内存，因为我们把字符串列转换成了整数和数值列(占用更少空间)，但是我们没有看到**内存释放**，因为**垃圾收集**还没有被触发。

最后，对列表调用 **as.data.frame()** 分配了约 1.6MB 的内存，并执行了超过 600 个复制操作。这是因为 **as.data.frame()** 并不是很有效率，它把输入复制了很多次。在下一节，我们将更详细地讨论复制的问题。

分析(profiling)有两个缺点：

1. **read_delim()** 只需要半秒钟左右，但是**分析**在最好的情况下，每一毫秒捕获一次内存使用情况。这意味着，我们将只有大约 500 个样本。(译者注：样本数量太少。)
2. 因为**垃圾收集**是**延迟**的，所以我们无法精确告知什么时候不再需要内存了。

你可以使用 **torture = TRUE** 参数解决这两个问题，它强制 R 在每次**内存分配**之后都运行**垃圾收集**(查看 **gctorture()** 获取更多细节)。这对这两个问题都有帮助，因

为内存会被尽可能快地释放，并且 R 的运行速度会慢 10 到 100 倍。这可以有效地使定时器的分辨率变得更大，这样你就可以看到更小的内存分配情况，以及精确地看到什么时候内存不再需要了。（译者注：相当于放慢动作，你可以看得更清楚一些。）

18.3.1 练习

1. 当输入是一个列表时，我们通过使用特殊的知识，可以创建更有效率的 `as.data.frame()`。数据框是列表，它的类是 `data.frame`，并且带有 `row.names` 的属性。`row.names` 是字符向量或者连续的整数向量，存储在由 `.set_row_names()` 创建的一种特殊格式里。这样，可以实现另一种 `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？这个函数有什么缺点？

2. 使用 `torture = TRUE` 参数，对下面的函数的代码进行分析。发现了什么意外的东西吗？阅读 `rm()` 的源代码来指出发生了什么。

```
f <- function(n = 1e5) {  
  x <- rep(1, n)  
  rm(x)  
}
```

18.4 就地修改

下面的代码中的 `x` 发生了什么情况？

```
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 语言对 `x` 进行就地修改。
2. R 语言把 `x` 复制到一个新的地址，并且修改副本，然后使用名称 `x` 指向新的地址。

事实证明，R 语言可以根据情况来做。在上面的例子中，它将对 `x` 进行就地修改。但是如果另一个变量也指向 `x`，那么 R 将把它复制到一个新的地址。为了探索更多的细节，我们使用 `pryr` 包中的两个工具。给定一个变量名，`address()` 会告诉我们变量在内存中的地址，`refs()` 会告诉我们有多少名字指向该地址。

```
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，如果你使用的是 RStudio，那么 `refs()` 将始终返回 2：环境浏览器为你在命令行中创建的所有对象，都创建了一个引用。)(译者注：本人在 RStudio 0.99.412 版上测试不是这样，也许老版本的 RStudio 是这样的。)

`refs()` 只是一个估计。它只能区分一个引用和多于一个引用两种情况(R 语言的未来版本可能会做得更好)。这意味着，`refs()` 在下列情况下都会返回 2：

```
x <- 1:5
y <- x
rm(y)
# 应该为 1，因为我们已经删除了 y
refs(x)
#> [1] 2
x <- 1:5
y <- x
z <- x
# 应该为 3
refs(x)
#> [1] 2
```

当 `refs(x)` 是 1 的时候，将会发生就地修改。当 `refs(x)` 是 2 的时候，R 将创建副本（这可以确保其它指向该对象的指针不受影响）。注意，在下面的示例中，`y` 一直指向同一个地址，而 `x` 则发生了变化。

```
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()`。每当被跟踪的对象发生复制的时候，它就会打印出一条消息：

```
x <- 1:10
# 打印出对象当前的内存地址
tracemem(x)
```

```
# [1] "<0x7feeaaa1c6b8>"
x[5] <- 6L
y <- x
# 打印出它从哪里移动到哪里了
x[5] <- 6L
# tracemem[0x7feeaaa1c6b8 -> 0x7feeaaa1c768]:
```

对于交互式使用，`tracemem()`比 `refs()`更加有用，但是因为它只是打印一条消息，所以很难用于编程。我不在本书中使用它，因为它与 `knitr` 的交互很差 (<http://yihui.name/knitr/>)，`knitr` 是我用来在文本中插入代码的工具。

由非原语函数来处理对象，总是会增加引用计数。但是原语函数通常不会。(原因有点复杂，但是可以看看 R-devel 上的一个帖子《Confused about NAMED》(<http://r.789695.n4.nabble.com/Confused-about-NAMED-td4103326.html>)。)

```
# 非原语函数处理对象，使得引用计数增加
f <- function(x) x
{x <- 1:10; f(x); refs(x)}
#> [1] 2
# Sum 是原语函数，所以引用计数不会增加
{x <- 1:10; sum(x); refs(x)}
#> [1] 1
# f()和 g()永远不会计算 x，所以引用计数不会增加
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
```

一般来说，假如对象没有在其地方被引用，那么任何原语替换函数将进行就地修改。这包括`[<-`、`[<-`、`@<-`、`$<-`、`attr<-`、`attributes<-`、`class<-`、`dim<-`、`dimnames<-`、`names<-`和`levels<-`。确切地说，所有的非原语函数都会增加引用量，但是原语函数可能由于编写方式的原因，它却不增加引用值。规则很复杂，不用努力去记住它们。相反，你应该通过使用 `refs()` 和 `address()` 来判断对象在什么时候被复制。

虽然，确定复制发生在什么时候并不困难，但是要阻止这样的行为。如果你发现自己诉诸于奇怪的技巧来避免复制，那么也许是时候用 C++ 来重写你的函数了，正如第 19 章所述的那样。

18.4.1 循环

R 语言中的 `for` 循环有个缓慢的坏名声。通常，这种缓慢是因为你在修改副本，而不是就地修改。考虑下面的代码。它从一个大数据框的每一列中减去中位数：

```
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]
}
```

你可能会惊讶地发现，循环中的每一次迭代都会复制数据框。我们可以通过使用 `address()` 和 `refs()` 更清楚地查看循环：

```
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"
```

对于每一次迭代，`x` 被移到了一个新的地址，所以 `refs(x)` 总是 2。这是因为 `[<-.data.frame` 不是原语函数，所以它总是会增加引用计数。我们可以通过使用列表而不是数据框，使这个函数更加有效率。修改列表使用了原语函数，所以引用计数并不会增加，并且所有的修改都是就地修改：

```
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"
```

在 R 3.1.0 之前的版本中，这种行为会导致更多的问题，因为数据框的每一个副本都是深拷贝(deep copy)。这会使这个例子需要运行大约 5 秒，而今天只需要 0.01 秒。

18.4.2 练习

1. 下面的代码里进行了一次复制。它发生在哪里？为什么？(提示：看看 `refs(y)`。)

```
y <- as.list(x)  
for(i in seq_along(medians)) {
```



```
y[[i]] <- y[[i]] - medians[i]  
}
```

2. 在上一节中的 `as.data.frame()` 实现有一个大缺点。是什么缺点？怎样避免？

19 使用 Rcpp 包编写高性能函数

有时候 R 代码就是不够快。你使用了分析工具找出了性能瓶颈在哪里，并且你做了在 R 语言中所能做的一切，但是你的代码仍然不够快。在这一章里，你将学习如何通过使用 C++ 重写**关键函数**来提高性能。这种魔法是由 Dirk Eddelbuettel 和 Romain Francois 创建的 **Rcpp** 包(<http://www.rcpp.org/>)提供的。**Rcpp** 包使得 C++ 连接到 R 变得很简单。虽然，可以用 C 语言或 Fortran 语言编写在 R 中使用的代码，但是相比 C++，这样将是痛苦的。**Rcpp** 提供了一种干净、友好的 API 让你编写高性能代码，从而与 R 的晦涩难懂的 C 语言 API 隔离开来。

C++ 可以解决的典型的性能瓶颈问题，包括：

1. 不能轻易向量化的**循环**，因为后续的迭代依赖于前面的结果。
2. **递归函数**，或者涉及数百万次**函数调用**的问题。在 C++ 中调用函数的开销比在 R 语言中要小得多。
3. 需要 R 语言没有提供的**高级数据结构和算法**的问题。通过标准模板库(STL)，C++ 有许多重要数据结构的高性能实现，从**有序映射表(ordered maps)**到**双头队列(double-ended queues)**。

本章的目的，是只讨论 C++ 和 **Rcpp** 中，在帮助你消除代码中的性能瓶颈时，绝对必要的那些方面。我们不会花很多时间在高级特性上，比如**面向对象编程**或者**模板**，因为我们的重点是写**小型并且独立**的函数，而不是大型程序。一定的 C++ 应用知识是有帮助的，但不是必要的。许多优秀的教程和参考资料都是免费的，包括 <http://www.learncpp.com/> 和 <http://www.cplusplus.com/>。对于更高级的主题，由 Scott Meyers 写的《Effective C++》系列书籍是受欢迎的选择。你还可以欣赏 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** 的所有方面进行了更加详细地讨论。

本章概要

19.1 节通过把简单的 R 函数转换为 C++ 的等价函数来教你如何编写 C++ 代码。你将了解 C++ 与 R 的区别，以及关键标、向量和矩阵类被称为什么。

19.1.6 节向你展示了如何使用 `sourceCpp()` 从磁盘上加载一个 C++ 文件，这种方式与使用 `source()` 加载一个 R 代码文件相同。

19.2 节讨论了如何通过 `Rcpp` 修改属性，并提到了其它一些重要的类。

19.3 节教你如何在 C++ 中处理 R 的缺失值。

19.4 节讨论了 `Rcpp` 语法糖(sugar)，它可以让你在 C++ 中避免循环，以及编写出与向量化的 R 代码看起来非常类似的代码。

19.5 节向你展示了如何使用 C++ 内置的**标准模板库(STL)**中一些最重要的**数据结构和算法**。

19.6 节展示了两个真实的案例研究，在它们之中，使用 `Rcpp` 使得性能得到了极大的提升。

19.7 节教你如何将 C++ 代码添加到包中。

19.8 节通过指出了更多帮助你学习 `Rcpp` 和 C++ 的资源来总结本章。

前提条件

本章中的所有例子都需要 0.10.1 或以上版本的 `Rcpp` 包。这个版本包括 `cppFunction()` 和 `sourceCpp()`，它们使得把 C++ 连接到 R 变得很容易。使用 `install.packages("Rcpp")` 从 CRAN 上安装最新版本的 `Rcpp`。

你还需要一个 C++ 编译器。要得到编译器：

在 Windows 上，安装 Rtools(<http://cran.r-project.org/bin/windows/Rtools/>)。

在 Mac 上，从 app store 上安装 Xcode。

在 Linux 上，`sudo apt-get install r-base-dev` 或者类似的命令。

19.1 开始使用 C++

`cppFunction()` 让你在 R 中可以编写 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 使用起来像普通的 R 函数
add
#> function (x, y, z)
#> .Primitive(".Call")(<pointer: 0x10fff9e10>, x, y, z)
add(1, 2, 3)
#> [1] 6
```

当你运行这段代码时，`Rcpp` 将编译 C++ 代码，并且构造一个 R 函数，该 R 函数会连接到编译的 C++ 函数。我们将使用这个简单的接口来学习如何编写 C++ 代码。C++ 是一个很大的语言，没有办法只在一章中就涵盖一切。因此，你会学习到基础知识，这样你就可以开始编写有用的函数来解决你的 R 代码中的性能瓶颈。

以下部分将教你把简单的 R 函数转换为等价的 C++函数的基础知识。我们将从简单的函数开始，它没有输入，只有一个标量输出，然后逐步创建越来越复杂的函数：

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

矩阵输入和向量输出

19.1.1 没有输入, 标量输出

让我们先从一个非常简单的函数开始。它没有参数并且永远返回整数 1：

```
one <- function() 1L
```

等价的 C++函数是：

```
int one() {  
    return 1;  
}
```

我们可以使用 `cppFunction()`，在 R 中编译和使用这个函数：

```
cppFunction('int one() {  
    return 1;  
}')
```

这个小函数说明了 R 和 C++的一些重要差异：

创建函数的语法看起来像**调用**函数语法；你不用像在 R 中那样使用赋值语句来创建函数。

你必须**声明**函数返回值的输出类型。这个函数返回一个 `int`(一个标量整数)。最常见的 R 向量类型的类为: `NumericVector`、`IntegerVector`、`CharacterVector` 和 `LogicalVector`。

标量和向量是不同的。与 `numeric`、`integer`、`character` 和 `logical` 向量的标量等价的是: `double`、`int`、`String` 和 `bool`。

你必须使用显式的返回语句从函数中返回值。

每一个语句都使用分号(`;`)结束。

19.1.2 标量输入, 标量输出

下一个示例函数实现了一个标量版本的 `sign()`, 如果输入是正数, 则它返回 1, 如果输入是负数, 则返回-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++ 版本中：

我们声明了每个输入的类型，与我们声明输出类型的方式相同。虽然，这使得代码更啰嗦了一点，但是它也使得**函数需要什么类型的输入**显得很明显的。

if 的语法是相同的——虽然在 R 和 C++ 之间有一些大的差异，但是也有很多相似之处！C++ 也有 **while** 语句，它与 R 的工作方式相同。正如在 R 中一样，你可以使用 **break** 来跳出循环，但是如果你要跳过循环中的一次迭代，那么你需要使用 **continue**，而不是 **next**。

19.1.3 向量输入，标量输出

R 和 C++ 之间有一个很大的区别是，在 C++ 中循环的开销要低得多。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 语言进行了一段时间的编程，那么你可能会对这个函数有本能反应！

```
sumR <- function(x) {  
  total <- 0  
  for (i in seq_along(x)) {  
    total <- total + x[i]  
  }  
  total  
}
```

在 C++ 中，循环的开销很少，所以可以使用循环。在 19.5 节中，你将看到 **for** 循环的替代选择，它们能更清楚地表达你的意图；它们并不是更快，但是它们可以使代码更容易理解。

```
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++版本的是相似的，但是：

1. 要找到向量的长度，我们使用了`.size()`方法，它返回一个整数。C++方法使用`.`进行调用(即一个句号)。
2. `for` 语句的语法有所不同：`for(init; check; increment)`。这个循环通过创建一个名为 `i` 且值为 `0` 的新变量进行初始化。每次迭代开始之前，我们检查 `i < n`，如果为否，则终止循环。每次迭代之后，我们把 `i` 的值增加 `1`，这里使用了特殊的前缀操作符`++`，它把 `i` 的值增加了 `1`。
3. 在 C++中，向量的索引从 `0` 开始。我要再说一遍，因为非常重要：在 C++ 中，向量的索引从 `0` 开始！当把 R 函数转换为 C++时，这是一种很常见的错误来源。
4. 使用`=`进行赋值，而不是`<-`。
5. C++提供了就地修改的运算符：`total += x[i]`与 `total = total + x[i]`是等价的。类似的就地修改运算符是`--`、`*=`和`/=`。

这是一个很好的展示 C++的哪些地方比 R 更高效例子。如以下的微基准测试所示，`sumC()`与内置的(且高度优化的)`sum()`相比是具有竞争力的，而 `sumR()`则要慢好几个数量级。


```
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

接下来，我们将创建一个函数，它计算一个值和一个向量值之间的欧氏距离：

```
pdistR <- function(x,ys) {
  sqrt((x - ys) ^ 2)
}
```

从函数的定义中看出，我们希望 `x` 是一个标量并不明显。我们不得不在文档中作出明确的说明。但是，在 C++ 版本当中，这不是一个问题，因为我们必须显式地声明类型：

```
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;
})
```

这个函数只引入了一些新概念：

我们使用**构造函数**来创建一个长度为 **n** 的新数值向量：**NumericVector(n)**。创建向量的另一个有用的方法是**复制**现有的向量：**NumericVector zs = clone(ys)**。

C++使用 **pow()**，而不是[^]来进行**幂运算**。

注意，因为 R 版本的是完全向量化的，所以它已经挺快了。在我的电脑里，它需要大约 8 毫秒来计算 100 万个元素的 **ys** 向量。C++函数要快两倍，4 毫秒，但是假设你花了十分钟来写 C++函数，那么你需要运行它 15 万次才值得重写。C++的函数更快的原因是微妙的，并且涉及到**内存管理**。R 版本需要创建一个与**(x - ys)** 长度一样的**中间向量**，而**分配内存**是一项昂贵的操作。C++函数避免了这个开销，因为它只使用了一个**中间标量**。在“语法糖”小节，你将看到如何利用 **Rcpp** 的向量化操作来重写这个函数，使得 C++代码几乎与 R 代码一样简洁。

19.1.5 矩阵输入，向量输出

每一种向量类型都有对应的矩阵类型：**NumericMatrix**、**IntegerMatrix**、**CharacterMatrix** 和 **LogicalMatrix**。使用它们是非常简单的。例如，我们可以创建一个函数，它实现了 **rowSums()**：

```
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
```

主要差异是：

在 C++ 中，你使用 `()` 对矩阵进行取子集操作，而不是使用 `[]`。

使用 `.nrow()` 和 `.ncol()` 方法获得一个矩阵的维度。

19.1.6 使用 `sourceCpp`

到目前为止，我们已经使用 `cppFunction` 内联了 C++ 代码。这使得表达更简单，但是对于真实的问题，使用独立的 C++ 文件通常更简单，然后使用 `sourceCpp()` 把它们加载到 R。这让你可以利用文本编辑器对 C++ 文件的支持(比如，语法高亮)，以及在发生编译错误时，更容易识别行号。你的独立 C++ 文件应该以 `.cpp` 作为扩展名，并且需要以这样开头：

```
#include <Rcpp.h>
using namespace Rcpp;
```

对每一个你想在 R 中使用的函数，你需要为它加上这样的前缀：

```
// [[Rcpp::export]]
```

注意，**空格**是强制要求的。

如果你熟悉 `roxygen2`，那么你可能想知道这跟 `@export` 是如何相关的。

`Rcpp::export` 控制着一个函数是否应该从 C++ 中导出到 R；`@export` 控制着一个函数是否从包中导出，并且提供给用户使用。

你可以把 R 代码嵌入在特殊的 C++**注释块**中。如果你想运行一些测试代码，那么这真的很方便：

```
/** R  
# This is R code  
*/
```

R 代码使用 `source(echo = TRUE)` 来运行，所以你不需要显式地打印输出。

要编译 C++ 代码，使用 `sourceCpp("path/to/file.cpp")`。这将创建相匹配的 R 函数，并将它们添加到你的当前会话中。注意，这些函数不能保存在一个 `.Rdata` 文件中，并在以后的会话中进行重新加载；每当你重启 R 的时候，它们都必须重新创建。例如，使用 `sourceCpp()` 运行以下使用 C++ 实现了 `mean` 函数的文件，然后把它与内置的 `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，那么你将注意到 `meanC()` 比内置的 `mean()` 要快得多。这是因为它为了提高速度而损失了数值精度。

在本章剩下的部分中，C++ 代码将独立表示，而不是包装在 `cppFunction` 的调用之中。如果你想试一试编译和(或)修改例子，那么你应该将它们粘贴到一个包含上述元素的 C++ 源文件中。

19.1.7 练习

现在是使用已有的 C++ 基本知识，来练习读写一些简单的 C++ 函数的好时候了。

对于下面的每一个函数，阅读它们的代码并找出对应的基础 R 函数。你可能没有理解代码的每一部分，但是你应该能够指出函数实现了什么功能。

```
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;  
}
```

为了练习函数编写技巧，请把以下函数转换成 C++ 版本。目前，我们假设输入中没有缺失值。

1. `all()`。
2. `cumprod()`、`cummin()`、`cummax()`。
3. `diff()`。首先假设 `lag` 为 1，然后推广到 `lag` 为 `n` 的情形。
4. `range`。
5. `var`。

你可以在维基百科上阅读这些方法

(http://en.wikipedia.org/wiki/Algorithms_for_calculating_variance)。无论什么时候要实现数值算法，检查一下关于这个问题我们已经知道了什么，总是有益的。

19.2 属性和其它的类

你已经看到了基本向量类(`IntegerVector`、`NumericVector`、`LogicalVector`、`CharacterVector`)及其标量(`int`、`double`、`bool`、`String`)和对应的矩阵(`IntegerMatrix`、`NumericMatrix`、`LogicalMatrix`、`CharacterMatrix`)。

所有的 R 对象都拥有属性，它们可以使用 `.attr()` 进行查询和修改。`Rcpp` 还提供了 `.names()` 作为 `name` 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，它是一个类的方法。这使你可以由 C++ 标量值创建 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()`起着类似的作用。

19.2.1 列表和数据框

`Rcpp` 还提供了 `List` 和 `DataFrame` 类，但是相比输入，它们对输出更有用。这是因为列表和数据框可以包含任意类，但是 C++ 需要事先知道它们是什么类。如果列表拥有已知的结构(如，它是一个 S3 对象)，那么你可以提取出元素，并且使用 `as()` 函数手动将它们转换为等价的 C++ 形式。例如，由 `lm()` 创建的对象——该函数用于拟合线性模型——是一个列表，该列表的元素总是相同的类型。(译者注：不是指列表内的元素类型相同，而是每次使用 `lm` 进行线性拟合得到的结果都含有相同类型的元素。)下面的代码演示了如何提取线性模型的平均百分误差(mean percentage error, `mpe()`)。这不是一个展示何时使用 C++ 的好例子，因为它用 R 语言很容易实现，但是它显示了如何处理一个重要的 S3 类。注意，这里使用了 `.inherits()` 和 `stop()` 来检查对象是不是一个线性模型。

```
#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

你可以把 R 函数放在一个类型为 **Function** 的对象中。这使得从 C++调用 R 函数变得很简单。首先，我们定义 C++函数：

```
#include <Rcpp.h>  
using namespace Rcpp;  
// [[Rcpp::export]]  
RObject callWithOne(Function f) {  
  return f(1);  
}
```

然后，在 R 中调用它：

```
callWithOne(function(x) x + 1)  
#> [1] 2
```

```
callWithOne(paste)
```

```
#> [1] "1"
```

R 函数返回什么类型的对象呢？我们不知道，所以我们使用通用的类型 `RObject`。另一种选择是返回一个 `List`。例如，下面的代码是在 C++ 中 `lapply` 的一种基本实现：

```
#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;
}
```

使用参数位置方式来调用 R 函数是显而易见的：

```
f("y", 1);
```

但是，如果使用命名参数的方式，那么你需要一种特殊的语法：

```
f(_["x"] = "y", _["value"] = 1);
```

19.2.3 其它类型

还有很多更专业的语言对象的类：`Environment`、`ComplexVector`、`RawVector`、`DottedPair`、`Language`、`Promise`、`Symbol`、`WeakReference` 等等。这些都超出了本章的范围，不会进一步讨论。

19.3 缺失值

如果你正在处理**缺失值**，那么你需要知道两件事情：

1. 在 C++ 的标量中，R 的缺失值有怎样的行为？(例如，**double**)。
2. 如何存取向量中的缺失值(例如，**NumericVector**)。

19.3.1 标量

下面的代码探讨了当你传入一个 R 的缺失值，并且把它强制转化为标量，然后转回成 R 向量时，会发生什么。注意，这种实验是一种有用的方法，它指出了任何操作做了什么。

```
#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` 值以外，情况看起来还不错：所有的缺失值都被保存了。但是，我们将在后面看到，并没有那么简单。

19.3.1.1 整数

对于**整数**(Integers)，缺失值被存储为**最小的整数**。如果你对它们不做任何事情，那么它们会被保留下来。但是，因为 C++ 并不知道最小的整数有这种特殊的行为，所以如果你对它们做了任何事，那么你可能会得到错误值：例如，`evalCpp('NA_INTEGER + 1')`会得到 `-2147483647`。(译者注：C++ 只是把 `NA_INTEGER` 当成一个普通的整数进行计算，而在 R 中，`NA + 1` 的计算结果应该还是 `NA`。)

所以，如果你想在整数中处理缺失值，那么，要么使用长度为 `1` 的 `IntegerVector`，要么在编写代码的时候保持谨慎。

19.3.1.2 双精度浮点数

对于**双精度浮点数**(Doubles)，你也许能够侥幸忽略缺失值，并使用 `NaN`(Not A Number，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`(或在 C++ 中，`NAN`) 的逻辑表达式，总是会计算成 `FALSE`：

```
evalCpp("NAN == 1")
#> [1] FALSE
evalCpp("NAN < 1")
#> [1] FALSE
evalCpp("NAN > 1")
#> [1] FALSE
evalCpp("NAN == NAN")
#> [1] FALSE
```

但是，当与布尔值联合的时候，要小心：

```
evalCpp("NAN && TRUE")
#> [1] TRUE
evalCpp("NAN || FALSE")
#> [1] TRUE
```

然而，在数值计算中，NaN 将传播 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 是由 **Rcpp** 引入的一个标量字符串 (Strings) 类，所以它知道如何处理缺失值。

19.3.3 布尔类型

在 C++ 中，布尔类型 (Boolean) 有两个可能的值 (**true** 或 **false**)，而在 R 中逻辑向量有三个值 (**TRUE**、**FALSE** 和 **NA**)。如果你强制转换一个长度为 1 的逻辑向量，那么请确保它不包含任何缺失值，否则它们将被转换为 **TRUE**。

19.3.4 向量

对于向量 (Vectors)，你需要使用特定向量类型的缺失值，**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
```

要检查向量中的一个值是否缺失，可以使用类方法 `::is_na()`：

```
#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
```

另一个替代方法是语法糖函数 `is_na()`，它接受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任意函数，以处理缺失值。如果 `na.rm` 是 `true`，那么忽略缺失值。如果 `na.rm` 是 `false`，那么如果输入包含任何缺失值，则返回缺失值。一些用来练习的好函数是 `min()`、`max()`、`range()`、`mean()` 和 `var()`。
2. 重写 `cumsum()` 和 `diff()`，使得它们可以处理缺失值。注意，这些函数具有稍微复杂一点的行为。

19.4 Rcpp 语法糖

`Rcpp` 提供了大量的语法糖(sugar)，来确保 C++ 函数与等价的 R 函数具有非常类似的工作方式。事实上，`Rcpp` 语法糖使得编写与等价的 R 代码看起来几乎相同的高效 C++ 代码成为可能。如果有一个你感兴趣的语法糖版本的函数，那么你应该使

用它：它既有表现力又是经过良好测试的。语法糖函数并不总是比手写的等价函数要快，但是它们在未来会变得更快，因为有更多的时间会花在优化 **Rcpp** 上。

语法糖函数大致可以分为：

算术和逻辑运算符(arithmetic and logical operators)

逻辑汇总函数(logical summary functions)

向量视图(vector views)

其它有用的函数

19.4.1 算术和逻辑运算符

所有基本的算术和逻辑运算符都是向量化的：**+**、*****、**-**、**/**、**pow**、**<**、**<=**、**>**、**>=**、**==**、**!=**、**!**。例如，我们可以用语法糖大大简化 **pdistC()** 的实现。

```
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 逻辑汇总函数

语法糖函数 **any()** 和 **all()** 是完全延迟计算的，例如 **any(x == 0)**，可能只需要计算向量的一个元素，并且返回一个可以使用 **.is_true()**、**.is_false()** 或 **.is_na()** 转换成 **bool** 值

的特殊类型。我们也可以使用这个语法糖写一个高效的函数来确定一个数值向量是否包含缺失值。要在 R 中做到这一点，我们可以使用 `any(is.na(x))`:

```
any_naR <- function(x) any(is.na(x))
```

然而，不管缺失值的位置在哪里，这都将做同样多的工作。下面是 C++的实现：

```
#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

许多有用的函数提供了向量视图：`head()`、`tail()`、`rep_each()`、`rep_len()`、`rev()`、`seq_along()`和`seq_len()`。在 R 中，这些都会产生向量的副本，但在 Rcpp 中，它们只是简单地指向现有的向量，并且覆盖取子集操作符`()`来实现特殊的行为。这使得它们非常高效：例如，`rep_len(x, 1e6)`并没有创建一百万个 `x` 的副本。

19.4.4 其它有用的函数

最后，这里列出了其它语法糖函数，它们模拟了常用的 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()`。

标量汇总：`mean()`、`min()`、`max()`、`sum()`、`sd()`和`var()`(对向量)。

向量汇总：`cumsum()`、`diff()`、`pmin()`和`pmax()`。

查值：`match()`、`self_match()`、`which_max()`、`which_min()`。

处理重复：`duplicated()`、`unique()`。

所有标准分布的 `d/q/p/r` 函数。

最后，`noNA(x)`判断向量 `x` 不包含任何缺失值，并允许一些数学运算上的优化。

19.5 标准模板库

当你需要实现更复杂的算法的时候，C++会显示出真正的力量。标准模板库(STL)提供了一些非常有用的数据结构和算法。本节将解释一些最重要的算法和数据结

构，并且为你指明学习更多知识的正确方向。我不能教你所有关于 STL 你需要知道的东西，但是希望例子能向你展示 STL 的力量，并且说服你学习更多 STL 的知识是有用的。

如果你需要在 STL 中没有实现的算法或数据结构，那么一个可以看看的好地方是 boost(<http://www.boost.org/doc/>)。如何在你的电脑上安装 boost 已经超出了本章的范围，但是一旦你安装好了，那么通过包含相应的头文件(例如 `#include <boost/array.hpp>`)，你就可以使用 boost 的数据结构和算法了。

19.5.1 使用迭代器

在 STL 中，**迭代器**是广泛使用的：许多函数**接受**或者**返回**迭代器。它们是基本循环的升级，是对底层数据结构的细节的抽象。

迭代器有三个主要运算符：

1. 使用 `++` 进行遍历。
2. 使用 `*` 来得到它们引用的值(或称为"解引用"(dereference))。
3. 使用 `==` 进行比较。

例如，我们可以使用迭代器重写 `sum` 函数：

```
#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;  
}
```

主要的变化是在 `for` 循环中：

我们从 `x.begin()` 开始循环，直到到达 `x.end()`。一个小优化是把终止迭代器的值保存起来，所以我们不需要每次都进行查询。在每次迭代中，这仅仅节省了大约 2 纳秒，所以只有当循环中的计算是非常简单的情况下，它才重要。

我们使用解引用操作符来得到它的当前值，而不是 `x` 的索引： `*it`。

注意迭代器的类型： `NumericVector::iterator`。每一种向量类型都有自己的迭代器类型： `LogicalVector::iterator`、 `CharacterVector::iterator` 等等。

迭代器还允许我们使用 `apply` 族函数在 C++ 中的等价函数。例如，我们可以使用 `accumulate()` 函数来再次重写 `sum()` 函数，它接受一个开始迭代器和一个终止迭代器，并把向量中的所有值都加起来。 `accumulate` 的第三个参数给出了初始值：这个参数特别重要，因为它也决定了 `accumulate` 使用的数据类型(因此我们使用 `0.0` 而不是 `0`，这样 `accumulate` 会使用 `double` 类型，而不是 `int` 类型)。要使用 `accumulate()`，我们需要包含 `<numeric>` 头文件。

```
#include <numeric>  
#include <Rcpp.h>  
using namespace Rcpp;  
// [[Rcpp::export]]  
double sum4(NumericVector x) {  
  return std::accumulate(x.begin(), x.end(), 0.0);  
}
```

`accumulate()`(连同 `<numeric>` 中的其它函数，如 `adjacent_difference()`、`inner_product()` 和 `partial_sum()`) 在 `Rcpp` 中并不是那么重要，因为 `Rcpp` 语法糖提供了等价函数。

19.5.2 算法

`<algorithm>` 头文件提供了大量使用迭代器的**算法**(Algorithms)。在 <http://www.cplusplus.com/reference/algorithm/> 上有很好的参考资料。例如，我们可以写一个基本的 `Rcpp` 版本的 `findInterval()`，它有两个向量参数，一个向量是**值**，另一个向量是**区间端点**，然后定位每个 `x` 所在的区间。这展示了一些更高级的迭代器特性。阅读下面的代码，看你能不能指出它是如何工作的。

```
#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;
}
```

其中的要点是：

我们同时对两个迭代器进行遍历(输入和输出)。

我们可以对解引用迭代器(`out_it`)进行赋值，来改变 `out` 里面的值。

`upper_bound()` 返回一个迭代器。如果我们想要 `upper_bound()` 的值，那么我们可以对其解引用；要找出它的位置，我们可以使用 `distance()` 函数。

小提示：如果我们想要这个函数与 R 中的 `findInterval()` (使用了手写的 C 代码) 一样快，那么我们需要对 `.begin()` 和 `.end()` 的调用进行一次计算，并保存结果。这是很容易的，但是偏离了本例的目的，所以省略了。进行这样的改变可以生成一个比 R 的 `findInterval()` 函数略快的函数，但是只需要约 1/10 的代码量。

使用 STL 算法通常比手动进行循环要好。在《Effective STL》中，Scott Meyers 给出了三个原因：效率、正确性和可维护性。STL 算法由 C++ 专家编写的，效率非常高，而且它们已经存在了很长一段时间，所以它们是经过良好测试的。使用标准算法也使得代码的意图更加清晰，有助于使其可读性更强，更易于维护。

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`。其中最重要的数据结构是 `vector`、`unordered_set` 和 `unordered_map`。

我们将在本节中关注这三种数据结构，但是其它数据结构的使用方法其实是类似的：它们只是在性能方面有不同的权衡而已。例如，`deque` (读作 deck) 与 `vector` 有着非常相似的接口，但是底层实现不同，它有不同的性能权衡。你可能想在你的问题中尝试它们。STL 数据结构的一个好的参考资料是 <http://www.cplusplus.com/reference/stl/>——我建议你在使用 STL 时，一直把它开着。

`Rcpp` 知道如何把许多 STL 数据结构转换为 R 中的等价数据结构，所以你可以从函数中直接返回它们，而不需要把它们明确地转换成 R 的数据结构。

19.5.4 向量

除了增长效率更高以外，STL 向量(Vector)与 R 向量非常相似。这使得向量适用于事先不知道输出有多大的情形。向量是**模板化**的，这意味着当你创建向量时，你需要指定向量将包含的对象的类型：`vector<int>`、`vector<bool>`、`vector<double>`、`vector<String>`。你可以使用标准的[]符号来访问向量中的单个元素的，你也可以使用`push_back()`在向量的末尾添加一个新元素。如果你在事先对向量的大小有一定的了解，那么你可以使用`reserve()`来分配足够的存储空间。下面的代码实现了**游程编码**(run length encoding, `rle()`)。它产生了两个向量的输出：一个向量是**值**，另一个向量是**长度**，它给出了每个元素重复了多少次。它是通过对输入向量 `x` 进行遍历，并把每个值与它前面的值进行比较：如果是相同的，那么**长度向量**中的最后一个值增加 1；如果是不同的，那么它在**值向量**的末尾添加一个值，并设置相应的长度为 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  
);  
}
```

(另一种实现是用迭代器 `lengths.rbegin()` 来替换 `i`，它总是指向向量的最后一个元素。你可以尝试自己来实现。) 向量的其它方法在

<http://www.cplusplus.com/reference/vector/vector/> 中有描述。

19.5.5 集合

集合维护着一组互不相同的值，它可以有效地告诉你以前是否见过某个值。它们对涉及到重复或惟一值的问题(如 `unique`、`duplicated` 或 `in`)是很有用的。C++提供了有序集(`std::set`)和无序集(`std::unordered_set`)，使用哪一个取决于排序对你是不是重要的。无序集往往会快得多(因为它们在内部使用了哈希表，而不是树型结构)，所以即使你需要有序集，你也应该考虑一下是否应该使用先无序集，然后再对输出进行排序。正如向量一样，集合也是模板化的，所以你需要根据你的目的，申请适当类型的集合：`unordered_set<int>`、`unordered_set<bool>`等。更多细节可以查看 `cpp` 和 `cpp`。

下面的函数使用了**无序集**来实现了用于**整数向量**的 `duplicated()` 的等价函数。注意 `seen.insert(x[i]).second` 的使用。`insert()` 返回一个 `pair`，`.first` 值是指向元素的迭代器，而 `.second` 值是一个布尔值，如果值是新添加进入集合的，那么它为 `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**只在 C++ 11 中可以使用，这意味着我们需要使用 `cpp11` 插件，`[[Rcpp::plugins(cpp11)]]`。

19.5.6 映射表

映射表(Map)与集合类似，它可以存储额外的数据，而不是存储存在与否。它对 `table()` 或者 `match()` 这样需要查找值的函数很有用。正如集合一样，它也存在**有序** (`std::map`)和**无序** (`std::unordered_map`)的版本。由于**映射表**有**值**(value)和**键**(key)，所以当初始化映射表时，你需要指定这两种类型：`map<double, int>`、`unordered_map<int, double>` 等等。以下示例展示了如何使用**映射表**为数值向量实现 `table()` 函数：

```
#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，无序映射表只能在 C++ 11 中使用，所以要使用它们，你也需要 `[[Rcpp::plugins(cpp11)]]`。

19.5.7 练习

为了练习使用 STL 算法和数据结构，请根据提示，在 C++ 中使用实现下面的 R 函数：

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：仅用一行代码！)。
4. 使用 `std::min()` 实现 `min()`，或者使用 `std::max()` 实现 `max()`。
5. 使用 `min_element` 实现 `which.min()`，或者使用 `max_element` 实现 `which.max()`。

- 使用 `sorted ranges` 和 `set_union`、`set_intersection` 和 `set_difference` 实现用于整数的 `setdiff()`、`union()` 和 `intersect()`。

19.6 案例研究

下面的案例研究展示了一些实际应用中使用 C++ 来替换缓慢的 R 代码的实例。

19.6.1 吉布斯采样器

下面的案例研究更新了 Dirk Eddelbuettel 的博客

(<http://dirk.eddelbuettel.com/blog/2011/07/14/>) 上的一个例子，说明了把 R 代码的吉布斯采样器(Gibbs sampler) 转换为 C++ 的过程。下面显示的 R 语言代码和 C++ 代码非常相似(只花了几分钟把 R 版本转换到 C++ 版本)，但是在我的电脑上，运行速度快了 20 倍。Dirk 的博客文章还显示了另一种让它变得更快的方法：使用 GSL 中更快的随机数生成器函数(从 R 中通过 `RcppGSL` 包可以很容易地访问)，可以让它再快 2 到 3 倍。

R 代码如下：

```
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  
}
```

转换为 C++ 代码是很简单的。我们：

为所有变量添加**类型声明**。

使用 `()` 而不是 `[]` 为矩阵进行取值。

给 `rgamma` 和 `rnorm` 的结果加上**下标**（译者注：即 `[0]`），把向量转换成标量。

```
#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);
}
```

对两种**实现**进行测评，得到：

```
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 的向量化与 C++的向量化

这个例子是改编自《Rcpp is smoking fast for agent-based models in data frames》

(《在数据框中，Rcpp 对基于主体的模型很快》，

<http://www.babelgraph.org/wp/?p=358>)。挑战是从三个输入中，预测模型的响应。基础的 R 版本预测器看起来像这样：

```
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  
}
```

我们希望能够将这个函数应用到许多输入，所以我们可能使用 **for** 循环编写一个向量输入的版本。

```
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 语言，那么你会有一种直觉，这将是很慢的，实际上也确实如此。有两种方法可以处理这个问题。如果你对 R 的函数很熟悉，那么你可以马上看出如何向量化这个函数(使用 `ifelse()`、`pmin()`和 `pmax()`)。另外，根据我们的知识，循环和函数调用的开销在 C++中是低得多的，所以我们可以用 C++重写 `vacc1a()`和 `vacc1()`。两种方法都是相当简单的。在 R 语言中：

```
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 语言很久了，那么你可能会认出这段代码中的一些潜在的**性能瓶颈**：`ifelse`、`pmin` 和 `pmax` 都是很慢的，可以替换为 `p + 0.75 + 0.5 * female`，`p[p < 0] <- 0, p[p > 1] <- 1`。你可以自己尝试测量一下时间变化。)

或者在 C++中：

```
#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;
}
```

下一步，我们生成一些样本数据，并且检查所有的三个版本都返回了相同的值：

```
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))
)
```

最初的博文忘了这样做，并且在 C++ 版本中引入了一个错误：它使用了 0.004 而不是 0.04。最后，我们可以对我们的三种方法进行基准测试：

```
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
```

毫不奇怪，我们最初使用了循环的方法非常缓慢。在 R 语言中进行向量化，对速度有巨大的提升，然后使用 C++ 的循环，我们又让性能有了更多的提升(约 10 倍)。我有点惊讶，C++ 会快这么多，这是因为 R 版本需要创建 11 个向量来存储中间结果，而 C++ 代码只需要创建 1 个。

19.7 在包中使用 Rcpp

使用 `sourceCpp()` 加载的相同 C++ 代码，也可以包装在一个包里。把独立的 C++ 源文件移到一个包中，有几种优点：

1. 你的代码可以让没有 C++ 开发工具的用户使用。
2. 多个源文件以及它们的依赖包，会由 R 包构建系统进行自动处理。
3. 包会为测试、编写文档以及一致性提供其它的基本支持。

要把 Rcpp 添加到一个现有的包中，你可以把你的 C++ 文件放在 `src/directory` 之中，并且修改(或创建)以下配置文件：

在 DESCRIPTION 中添加：

```
LinkingTo: Rcpp  
Imports: Rcpp
```

确保你的 NAMESPACE 包含：

```
useDynLib(mypackage)  
importFrom(Rcpp, sourceCpp)
```


我们需要从 `Rcpp` 中导入一些东西(任何东西)，以便让内部的 `Rcpp` 代码正确加载。这是 R 语言中的一个错误，希望在未来的版本中能够修复。

要生成一个包含简单的"hello world"函数的新 `Rcpp` 包，你可以使用

`Rcpp.package.skeleton()` 函数：

```
Rcpp.package.skeleton("NewPackage", attributes = TRUE)
```

要生成基于你使用 `sourceCpp()` 加载的 C++ 文件的包，可以使用 `cpp_files` 参数：

```
Rcpp.package.skeleton("NewPackage", example_code = FALSE,  
cpp_files = c("convolve.cpp"))
```

在构建包之前，你需要运行 `Rcpp::compileAttributes()`。这个函数在 C++ 文件中扫描 `Rcpp::export` 属性，并且生成必要的代码，使得函数在 R 语言中可用。无论何时，比如添加函数、删除函数或改变了函数的签名，都需要重新运行 `compileAttributes()`。这是由 `devtools` 包和 Rstudio 自动完成的。

更多细节请参见 `Rcpp` 包的 `vignette`，`vignette("Rcpp-package")`。

19.8 更多学习资源

本章只触及了 `Rcpp` 的一小部分，给了你使用 C++ 重写性能不佳的 R 代码的基本工具。《Rcpp book: Seamless R and C++ Integration with Rcpp》(《用 Rcpp 使 R 与 C++ 无缝集成》，<http://www.rcpp.org/book>) 是学习更多 `Rcpp` 知识的最好参考书籍。如上所述，`Rcpp` 有许多其它功能，使得 R 语言与现有的 C++ 代码进行对接，变得很容易，包括：

属性的附加特性，包括指定默认参数、链接外部 C++ 依赖、从包中导出 C++ 接口。这些特性以及更多内容，在 `Rcpp` 属性的 `vignette` 中进行介绍，`vignette("Rcpp-attributes")`。

在 C++ 数据结构和 R 语言数据结构之间，自动创建**包装器**，包括把 **C++ 类** 映射到引用类。介绍这个话题的一个好资料是 **Rcpp** 模块的 **vignette**，**vignette("Rcpp-modules")**。

Rcpp 快速参考指南，**vignette("Rcpp-quickref")**，包含了 **Rcpp** 类和常见的编程术语的有用的总结。

我强烈建议密切关注 **Rcpp** 的主页(<http://www.rcpp.org>)和 Dirk 的 **Rcpp** 页面(<http://dirk.eddelbuettel.com/code/rcpp.html>)，以及订阅 **Rcpp** 邮件列表(<http://lists.r-forge.r-project.org/cgi-bin/mailman/listinfo/rcppdevel>)。 **Rcpp** 仍在积极开发之中，每一次发布都会变得更好。

我找到的有助于学习 C++ 的其它资源是：

由 Scott Meyers 写的《Effective C++》(<http://amzn.com/0321334876?tag=devtools-20>)和《Effective STL》(<http://amzn.com/0201749629?tag=devtools-20>)。

《C++ Annotations》(<http://www.icce.rug.nl/documents/cplusplus/cplusplus.html>)，旨在"使具有丰富的 C 语言(或者任何其它与 C 语言语法相似的语言，比如 Perl 或 Java)编程经验的用户，了解更多 C++ 的知识，或者过渡到 C++ 语言"。

算法库(<http://www.cs.helsinki.fi/u/tpkarkka/alglib/k06/>)，它提供了更技术性的、但是仍然简洁的、重要的 STL 概念的描述。(网页上"Notes"下面的链接。)编写高性能的代码可能还需要你重新思考你的基本方法：充分理解基本数据结构和算法是非常有用的。这超出了本书的范围，但是我建议阅读《Algorithm Design Manual》(《算法设计手册》，<http://amzn.com/0387948600?tag=devtools-20>)，麻省理工学院的《Introduction to Algorithms》(《算法导论》，[需要帮助请咨询 QQ:59739150 E-mail: liuning.1982@qq.com](http://ocw.mit.edu/courses/electrical-engineering-and-computerscience/6-046j-</p></div><div data-bbox=)

introduction-to-algorithms-sma-5503-fall-2005/), 由 Robert Sedgewick 和 Kevin Wayne which 写的《Algorithms》(《算法》), 它有免费的在线教程 (<http://algs4.cs.princeton.edu/home/>)以及相应的 coursera 课程 (<https://www.coursera.org/course/algs4partI>)。

19.9 鸣谢

我要感谢 Rcpp 邮件列表中许多有益的对话, 特别是 Romain Francois 和 Dirk Eddelbuettel, 他们不仅对我的许多问题提供详细的回答, 也对改进 Rcpp 具有令人难以置信的响应速度。本章没有 JJ Allaire 的帮助是不可能写成的; 他鼓励我学习 C++并且在这个过程中, 他回答了我许多愚蠢的问题。

20 R 的 C 语言接口

阅读 R 的源代码是一种提高你的编程技能的非常强大的技术。然而，许多**基础 R 函数**以及**较早期的包**中的许多函数，是用 C 语言编写的。能够理解这些函数是如何工作的，是有用的，因此，本章将向你介绍 R 的 C 语言 API(译者注：Application Programming Interface，应用程序编程接口)。你需要一些基本的 C 语言知识，你可以从标准的 C 语言教材中(例如，由 Kernigan 和 Ritchie 写的《The C Programming Language》[《C 程序设计语言》，<http://amzn.com/0131101633?tag=devtools-20>])，或从第 19 章中获取。你需要一点耐心，阅读 R 的 C 语言源代码是可能的，并且从中你会学到很多东西。本章的内容大量地吸收了《Writing R extensions》(<http://cran.rproject.org/doc/manuals/R-exts.html>)第五节的内容("System and foreign language interfaces", "系统和其它语言接口")，但是重点关注在**最佳实践**和**现代工具**之上。这意味着，它不包括旧的**C 接口**、定义在 **Rdefines.h** 中的旧 **API** 以及很少使用的语言特性。要查看 R 的完整的 C 语言 API，可以看看 **Rinternals.h** 头文件。在 R 中找到和显示这个文件是很容易的：

```
rinternals <- file.path(R.home("include"), "Rinternals.h")
file.show(rinternals)
```

所有的函数都以前缀 **Rf_**或 **R_**进行定义，但**导出时**却没有前缀(除非使用了 **#define R_NO_REMAP**)。

我不建议使用 C 编写新的高性能的代码，而是应该使用 **Rcpp** 包编写 C++代码。**Rcpp** 的 API 可以让你免受 R 语言 API 中的一些历史特性之苦，并且帮助你进行内存管理，以及提供了许多有用的辅助方法。

本章概要

20.1 节展示了使用 **inline** 包**创建**和**调用** C 函数的基本知识。

20.2 节展示了如何将数据结构名称从 R 语言转换到 C 语言。

20.3 节教你如何在 C 语言中**创建向量、修改向量以及对向量进行强制转换**。

20.4 节向你展示了如何使用**成对列表**。你需要知道这一点，因为**成对列表和列表**之间的区别，在 C 语言中比在 R 语言中更重要。

20.5 节讨论了**输入验证**的重要性，它使得你的 C 函数不会让 R 崩溃。

20.6 节通过向你展示，如何找到 R 的**内部函数**和**原语函数**的 C 源代码，对本章进行了总结。

前提条件

为了理解现有的 C 代码，你自己生成能进行实验的简单示例，是很有用的。为此，本章中的所有例子都使用 **inline** 包，这使得编译 C 代码并且把它链接到你的当前 R 会话变得非常容易。通过运行 **install.packages("inline")** 安装这个包。要轻松地找到与**内部函数**和**原语函数**相关的 C 代码，你需要 **pryr** 包中的函数。使用 **install.packages("pryr")** 安装该包。

你还需要一个 **C 编译器**。Windows 用户可以使用 Rtools(<http://cran.r-project.org/bin/windows/Rtools/>)。Mac 用户需要 Xcode 命令行工具(<http://developer.apple.com/>)。大多数 Linux 发行版将会自带必要的编译器。

在 Windows 中，Rtools 可执行文件的路径(一般是 **C:\Rtools\bin**)以及 C 编译器可执行文件的路径(一般是 **C:\Rtools\gcc-4.6.3\bin**)，都需要包含在 Windows 的 **PATH** 环境变量之中。为了使 R 能识别这些值，你可能需要重新启动 Windows。

20.1 在 R 中调用 C 语言函数

通常，从 R 中调用 C 函数需要两个部分：一个 C 函数以及使用了 `.Call` 的 R 包装函数。下面的简单函数把两个数加起来，它说明了用 C 语言进行编码的一些复杂的地方：

```
// 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` 的替代是 `.External`。它的用法几乎相同，不同之处是这个 C 函数会接受一个包含 `LISTSXP` 的参数，它是一个成对列表，从它之中可以提取参数。这使它可以编写带有不定参数数量的函数。然而，在基础 R 中，这不是经常使用的，`inline` 目前并不支持 `.External`，所以在本章中我不会对其进行进一步讨论。)

在这一章里，我们将使用 `inline` 包在一个步骤中产生两个部分。我们可以这样写：

```
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
```

在我们开始读写 C 代码之前，我们需要知道一点基本的数据结构知识。

20.2 C 语言的数据结构

在 C 语言层面，所有的 R 对象都存储在一种共同的数据类型之中，**SEXP**，或称为 **S-expression**。所有 R 对象都是 **S-expressions**，所以你创建的每一个 C 函数，都必须返回一个 **SEXP** 作为输出，以及接受 **SEXP** 作为输入。(从技术上讲，这是一个指向 **typedef SEXPREC** 结构的指针)。**SEXP** 是一种变化的类型，它带有与所有的 R 数据结构匹配的子类型。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

注意：在 C 语言中，列表被称为 **VECSXP** 而不是 **LISTSXP**。这是因为列表的早期实现是使用与 Lisp 相似的链表，而现在，这个被称为“成对列表”(“pairlist”)。

字符向量比其它原子向量要更复杂一些。一个 **STRSXP** 包含一个 **CHARSXP** 向量，其中每个 **CHARSXP** 都指向存储在全局池中的 C 风格字符串。这种设计使得单个 **CHARSXP** 可以在多个字符向量之间共享，减少了内存使用量。参见 18.1 节获取详情。

还有不太常见的对象类型的 **SEXP**：

CPLXSXP：复数向量

LISTSXP："pair"列表。

在 R 级别，你只需要关心**列表**和**函数参数的成对列表**之间的区别，但是在 R 内部，会在更多的地方使用到它们。

DOTSXP：...

SYMSXP：名称/符号

NILSXP：空值 **NULL**

内部对象(即，创建出来被 C 函数，而不是 R 函数使用的对象)的 **SEXP**：

LANGSXP：语言结构

CHARSXP：标量字符串

PROMSXP：承诺，需要**延迟计算**的函数参数

EXPRSXP：表达式

没有内置的 R 函数可以轻松地访问这些名字，但是 **pryr** 包提供了 **sexp_type()** 函数来做到这一点：

```
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 函数的核心都是 R 的数据结构和 C 的数据结构之间的**转换**。输入和输出将永远是 R 数据结构(**SEXP**)，为了进行进一步的工作，你将需要将其转换成 C 数据结构。本节关注于向量，因为它们是你在工作中最有可能用到的对象类型。

垃圾收集器是另一个难点：如果你不**保护**你创建的每个 R 对象，那么**垃圾收集器**会认为它们是没有用到的，并且会删除它们。

20.3.1 创建向量和垃圾收集

创建一个新的 **R 层次的对象**(R-level object)的最简单方法是使用 **allocVector()**。它需要两个参数，要创建的 **SEXP** 类型(或 **SEXPTYPE**)和向量的长度。下面的代码创建了一个有三个元素的列表，它包含一个逻辑向量、一个数值向量和一个整数向量，它们的长度都是 4：

```
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
```

你可能想知道所有的 `PROTECT()` 调用是干什么的。它们告诉 R，这个对象正在使用之中，如果垃圾收集器激活了，那么该对象不应该被删除。(我们不需要保护 R 已经知道我们在使用的对象，比如函数参数)。

你还需要确保每个受过保护的對象不再受保护。`UNPROTECT()` 接受一个整数参数，`n`，它对最后 `n` 个受保护的對象解除保护。受保护的數量和解除保护的數量必須匹配。如果不匹配，那么 R 将警告："在 `.Call` 中堆栈失衡"。

在某些情況下，還需要其它特殊形式的保护：

`UNPROTECT_PTR()` 解除由 `SEXP` 指向的對象的保护。

`PROTECT_WITH_INDEX()` 保存保护位置的索引，使用 `REPROTECT()`，这些索引可以用来取代被保护的值得。

查阅《R externals》中垃圾收集的部分(<http://cran.r-project.org/doc/manuals/R-exts.html#Garbage-Collection>)以获取更多细节。

妥善保护你分配的 R 对象是非常重要的！保护不当会导致难以诊断的错误，典型的错误是段错误(segfaults)，但是其它错误也是可能的。一般来说，如果你分配一个新的 R 对象，那么你必须保护它。

如果你运行 `dummy()` 几次，那么你会注意到输出是不同的。这是因为 `allocVector()` 为每个输出分配内存，但是它并没有首先清理内存。对于实际的函数，你可能想要遍历向量中的元素，并将其设置为一个常数。这样做的最有效的方法是使用 `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

每种原子向量都有一种特殊常数来存取缺失值：

```
INTSXP: NA_INTEGER  
LGLSXP: NA_LOGICAL  
STRSXP: NA_STRING
```

对于 **REALSXP** 来说, 缺失值有些复杂, 因为对于缺失值, 现有一种由浮点数标准定义的协议(IEEE 754 (http://en.wikipedia.org/wiki/IEEE_floating_point)). 在双精度浮点数中, **NA** 是带有特殊位模式的 **NaN**(最低的字(word)是 1954, Ross Ihaka 出生的年份), 还有其它用于正负无穷的特殊值。使用 **ISNAQ**、**ISNANQ** 和 **!R_FINITEQ** 宏来检查缺失值、**NaN** 或非有限值。使用常量 **NA_REAL**、**R_NaN**、**R_PosInf** 和 **R_NegInf** 来设置这些值。

我们可以利用这些知识创建一个简单版本的 **is.NAQ**:

```
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`，而 C 的 `ISNANO` 宏，只对 `NA_REAL` 返回 `TRUE`。

20.3.3 访问向量数据

每个原子向量都有一个**辅助函数**，它允许你访问 C 数组，这个数组存储了向量中的数据。使用 `REAL()`、`INTEGER()`、`LOGICAL()`、`COMPLEX()` 和 `RAW()` 来访问 `numeric`、`integer`、`logical`、`complex` 和 `raw` 向量中的 C 数组。下面的例子展示了如何使用 `REAL()` 来**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
```

在处理长向量时，使用一次辅助函数有性能优势，并将结果保存在一个指针中：

```
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
```

在我的电脑上，对于一个有一百万个元素的向量，`add_two()`大约比 `add_one()`快两倍。这是基础 R 中的常用方法。

20.3.4 字符向量和列表

字符串和列表更加复杂，因为向量的每个元素都是 `SEXP`，而不是基本的 C 数据结构。`STRSXP` 的每个元素都是 `CHARSXP`，它是不可变对象，它包含一个指针，该指针指向存储在全局池中的 C 字符串。

使用 `STRING_ELT(x, i)`提取 `CHARSXP`，使用 `CHAR(STRING_ELT(x, i))`获得实际的 `const char*`字符串。

使用 `SET_STRING_ELT(x, i, value)`设置值。

使用 `mkChar()`将 C 字符串转换成 `CHARSXP`，使用 `mkString()`将 C 字符串转换成 `STRSXP`。

使用 `mkChar()`创建字符串，并插入到现有的向量中，使用 `mkString()`创建新(长度为 1)向量。

下面的函数显示了如何创建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"
```

如果你想修改向量中的字符串，那么会有点困难，因为你需要知道很多关于 C 字符串操作的知识(这个很难，要做好更难)。对于包括各种修改字符串操作的问题，你最好使用 **Rcpp**。

列表的元素可以是任何其它的 **SEXP**，这通常会使它们很难用 C 语言处理(你需要大量的 **switch** 语句来处理各种可能性)。列表的访问器(**accessor**)函数是 **VECTOR_ELT(x,i)**和 **SET_VECTOR_ELT(x, i, value)**。

20.3.5 修改输入

当修改函数输入的时候，你必须非常小心。下面的函数有意想不到的行为：

```
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，它也修改了 **y**！这是由于 R 语言延迟的修改时复制语义导致的。为了避免这样的问题，在修改输入之前，总是对其进行复制(**duplicate()**)：

```
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
```

如果你正在使用列表，那么使用 `shallow_duplicate()` 进行浅复制(shallow copy); `duplicate()` 也将复制列表中的每个元素。

20.3.6 强制转换成标量

有一些辅助函数，可以把长度为 1 的 R 向量转换成 C 标量：

```
asLogical(x): INTSXP -> int
asInteger(x): INTSXP -> int
asReal(x): REALSXP -> double
CHAR(asChar(x)): STRSXP -> const char*
```

相反转换方向的辅助函数是：

```
ScalarLogical(x): int -> LGLSXP
ScalarInteger(x): int -> INTSXP
ScalarReal(x): double -> REALSXP
mkString(x): const char* -> STRSXP
```

这些都会创建 R 层级的对象，所以它们需要使用 `PROTECT()` 进行保护。

20.3.7 长向量

从 R 3.0.0 版本开始, R 向量的长度可以大于 **232** 了。这意味着, 向量的长度无法可靠地存储在一个 **int** 中, 因此, 如果你希望在你的代码中使用**长向量**, 那么你不能编写类似 **int n = length(x)** 这样的代码。相反, 要使用 **R_xlen_t** 类型和 **xlength()** 函数, 然后编写 **R_xlen_t n = xlength(x)**。

20.4 成对列表

在 R 代码中, 只有在少数情况下, 你需要关心**成对列表**和**列表**之间的差异(如 14.5 节所述)。在 C 语言中, **成对列表**扮演着更重要的角色, 因为它们用于**调用**、**未计算的参数**、**属性**和 C 语言的**....**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**)无法通过索引来访问任意位置。相反, R 提供了一组辅助函数在链接列表中进行搜索。基本的辅助函数是 **CAR()**, 它提取列表的第一个元素; 而 **CDR()**则提取列表的其余部分。这些可以组合起来, 从而得到 **CAAR()**、**CDAR()**、**CADDR()**、**CADDDR()**等等。与**获取函数**(getter)对应, R 还提供了**设置函数**(setter)**SETCAR()**、**SETCDR()**等等。

下面的例子说明了 **CAR()**和 **CDR()**如何从**被引用**(quoted)的**函数调用**中取出内容:

```
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))
# 第一个元素
car(x)
#> f
# 第二和第三个元素
cdr(x)
#> $a
```

```
#> [1] 1
#>
#> $b
#> [1] 2
# 第二个元素
car(cdr(x))
#> [1] 1
cadr(x)
#> [1] 1
```

成对列表总是以 `R_NilValue` 结尾。要遍历成对列表中的所有元素，请使用这个模板：

```
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
```

你可以使用 `CONS()` 创建新的成对列表，也可以使用 `LCONS()` 创建新的调用。记得把最后一个值设为 `R_NilValue`。因为这些都是 R 对象，所以它们也会被加入垃圾收集，因此必须使用 `PROTECT()` 对其进行保护。事实上，像下面这样编写代码，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，我得到的结果是 `5 + 5` —— 完全在意料之外！事实上，为了安全起见，我们必须保护每个生成的 `ScalarReal`，因为每个 R 对象的内存分配都可以触发垃圾收集器。

```
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()`允许你存取与成对列表的元素相关联的标签(也称为名字)。标签应该是一个符号(symbol)。要创建一个符号(相当于 R 中的 `as.symbol()`), 可以使用 `install()`。属性也是成对列表, 但是有专用的辅助函数 `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()`必须对属性的成对列表进行线性搜索。)

对于常见的设置操作, 有简写的方式(但是名字会令人迷惑): `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

如果用户给你的函数提供了意想不到的输入(例如输入了列表, 而不是数值向量), 那么很容易让 R 崩溃。出于这个原因, 编写一个**包装函数**来检查参数是不是正确的类型, 是个很好的主意。在 R 层面, 做到这一点通常更容易。例如, 回到我们

的第一个 C 代码的例子，我们可以把 C 函数重命名为 `add_`，然后给它编写包装函数来检查输入是不是正确的：

```
add_ <- cfunction(signature(a = "integer", b = "integer"), "  
SEXPR 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)  
}
```

另外，如果我们想接受不同的输入，那么我们可以像下面这么做：

```
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)  
}
```

要在 C 层面强制转换对象，请使用 `PROTECT(new = coerceVector(old, SEXPTYPE))`。如果 `SEXPR` 不能转换为所需的类型，这将返回一个错误。

要检查一个对象是不是指定的类型，你可以使用 `TYPEOF`，它返回一个 `SEXPTYPE`：

```
is_numeric <- cfunction(c("x" = "ANY"), "  
return ScalarLogical(TYPEOF(x) == REALSXP);  
")  
is_numeric(7)  
  
#> [1] TRUE  
is_numeric("a")  
#> [1] FALSE
```

还有许多辅助函数，当结果为 **FALSE** 时，它们返回 0；当结果为 **TRUE** 时，它们返回 1：

原子向量：**isInteger()**、**isReal()**、**isComplex()**、**isLogical()**、**isString()**

原子向量的组合：**isNumeric()**(整数、逻辑、实数)、**isNumber()**(整数、逻辑、实数、复数)、**isVectorAtomic()**(逻辑、整数、数值、复数、字符串、raw 类型)。

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、表达式)。

请注意，其中一些函数与 R 中名字相似的函数的行为是不同的。例如，对于原子向量、列表和表达式，**isVector()** 返回 **true**，而只有当输入除了名字以外，没有其它属性时，**is.vector()** 才返回 **TRUE**。

20.6 找到一个函数的 C 源代码

在 **base** 包中，R 不使用 **.Call()**。相反，它使用了两个特殊的函数：**.Internal()** 和 **.Primitive()**。找出这些函数的源代码是个艰巨的任务：首先，你需要在

`src/main/names.c` 中寻找它们的 C 函数名，然后搜索 R 源代码。

`pryr::show_c_source()` 使用了 GitHub 代码搜索功能，让这个工作自动化了：

```
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
```

这显示了下面的 C 源代码(为了更清晰，略有编辑)：

```
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;
}
```

`.Internal()`和`.Primitive()`函数与`.Call()`函数相比，有稍微不同的接口。它们总是有四个参数：

SEXP call：对函数的完整调用。`CAR(call)`给出了函数的名字(作为符号)；

`CDR(call)`给出了参数。

SEXP op：“偏移指针”(“offset pointer”)。当多个 R 函数使用了相同的 C 函数时，这会使用到。例如，`do_logic()`实现了`&`、`|`和`!`。`show_c_source()`帮你把这些打印出来。

SEXP args：成对列表包含的函数未计算的参数。

SEXP rho：调用被执行时所处的环境。

这给了内部函数难以置信的灵活性，它可以控制如何以及何时对参数进行计算。例如，内部 S3 泛型方法调用 `DispatchOrEval()`，它要么调用适当的 S3 方法，要么

立即计算所有的参数。这种灵活性是有代价的，因为它使得代码变得难以理解。然而，计算这些参数，通常是第一步，而函数剩下的部分是简单的。

下面的代码显示了 `do_tabulate()` 转换为标准的 `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", "nbins");  
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
```

为了编译这段代码，我也删除了对 `_R_` 的调用，这是一个内部 R 函数，用于在不同的语言之间翻译错误消息。下面的最终版本，把更多的强制转换逻辑移到了一个辅助 R 函数之中，并且做了一些小的重组，使得代码更容易理解一点。我还添加了 `PROTECT()`；在最初的时候，这个可能缺失了，因为作者知道这里是安全的。

```
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
```