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

此为本书预览版，只包含目录和第一部分“基础知识” (1-9 章)，帮助读者了解本书的内容和质量。有需要的读者，可以联系本人。本书目前尚未出版实体书，只提供电子版。

联系方式：

QQ: 59739150

E-mail: liuning.1982@qq.com

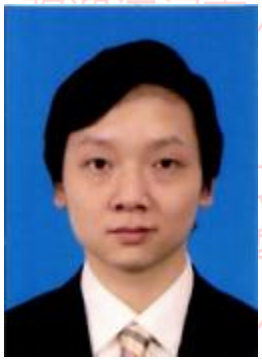
需要帮助请咨询 转载请注明出处
欢迎访问本人博客：http://blog.csdn.net/liu7788414
qq群：485732827

修改日期：2015-09-01

目 录

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，涉及的国家和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. 它有用于**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，并且可能发生意想不到的失败。

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 f()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 jekyll(http://jekyllrb.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 stringAsFactors = 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，通常是你在"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	`x[[1]]`	`x[1]`
List	`x[[1]]`	`x[1]`
Factor	`x[1:4, drop = T]`	`x[1:4]`
Array	`x[1, ]` **or** `x[, 1]`	`x[1, , drop = F]` **or** `x[, 1, drop = F]`
Data frame	`x[, 1]` **or** `x[[1]]`	`x[, 1, drop = F]` **or** `x[1]`

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 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(Rightt-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"
unname(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 unname()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 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, strftime, 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, mul
tinom, 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")，而是来自于计算机科学术语"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，j()，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的方法)。

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**：

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

这样是不能工作的，因为 R 语言搜索一切东西都是依靠**词法作用域**，甚至+运算符。无法让一个函数完全独立，因为你必须依赖于 R 语言的 base 包或者其它包中定义的函数。

你可以使用同样的思路做一些其它极其不明智的事情。例如，由于 R 语言中所有的**标准运算符**都是函数，所以你可以使用**自己实现的方法**覆盖它们。如果你是个特别淘气的家伙，那么请在你的朋友离开电脑时，偷偷在他们的电脑上运行以下代码：

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

这将引入一个特别有害的错误：每 10%的次数中，任何括号内的数值计算结果都会被加上 1。这也是另一个在重启时要用干净的 R 会话的理由！

6.2.5 练习

1. 下列代码返回什么？为什么？这些 c 分别代表什么含义？

```
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(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。它没有正式的类定义。

S4 与 S3 类似，但是更正式。与 S3 相比，它主要有两个不同。S4 有正式的类定义，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。ftype()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:ftype()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(disp), data = mtcars)
```

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

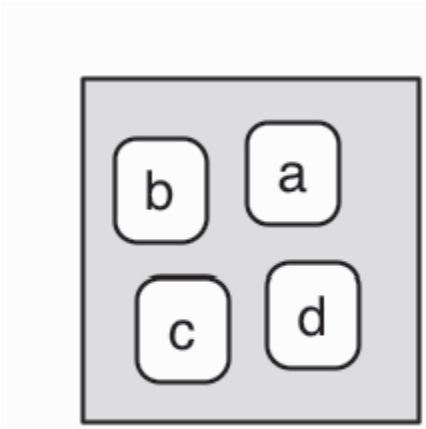
8.5 节讨论了三个问题，环境是有用的数据结构，在作用域中具有独立的作用。

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。

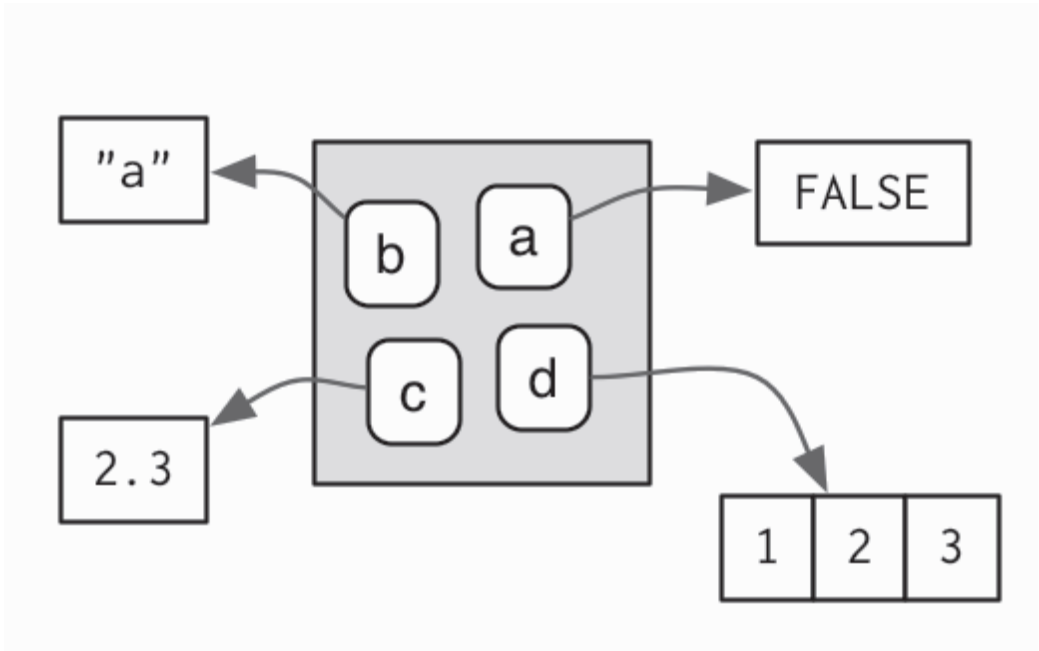
8.1 环境基础

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：



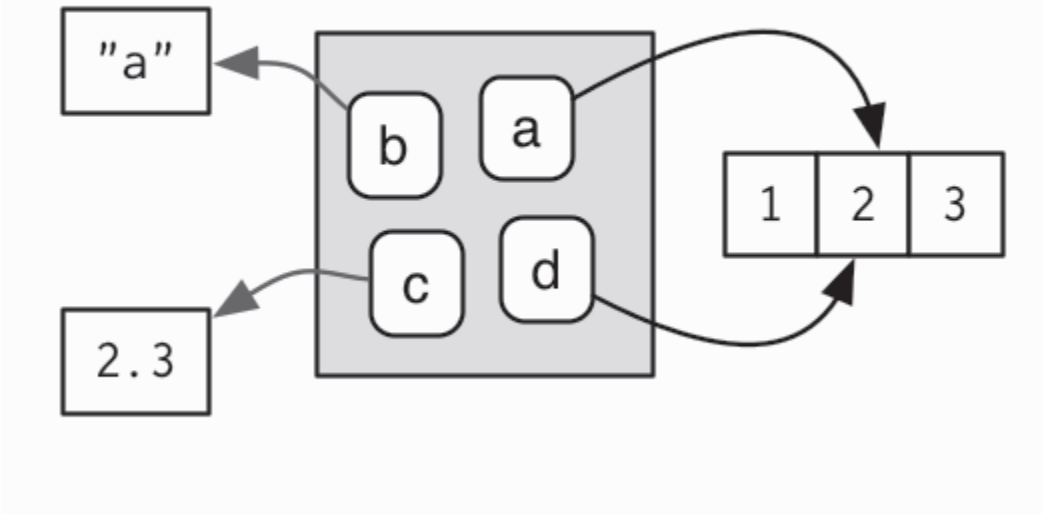
每个名字都指向一个对象，这个对象保存在内存中的某个地方：

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



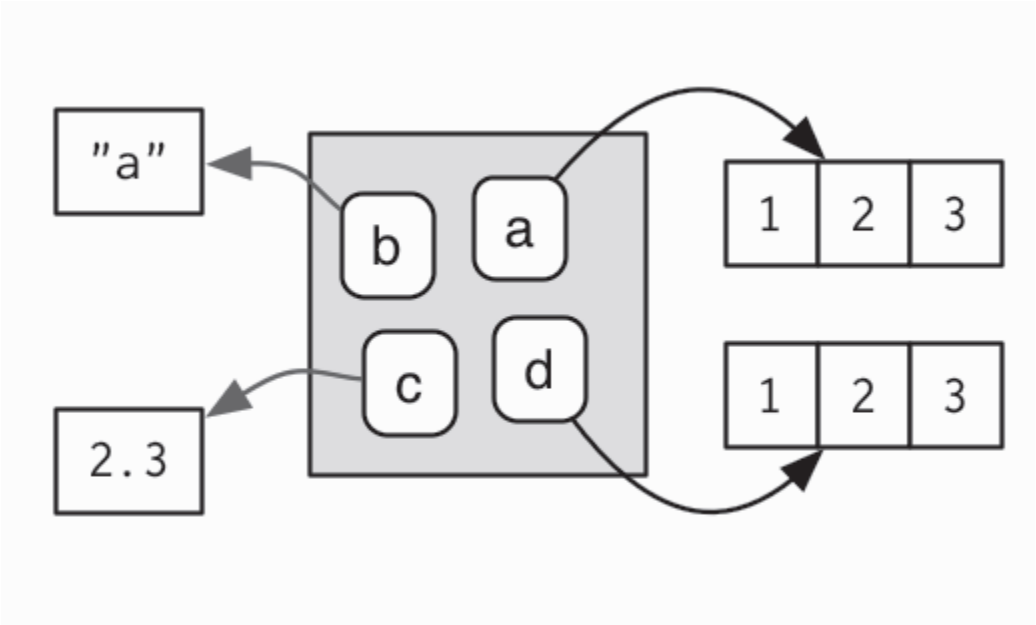
对象并不存在于环境中，所以多个名字可以指向同一个对象：

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



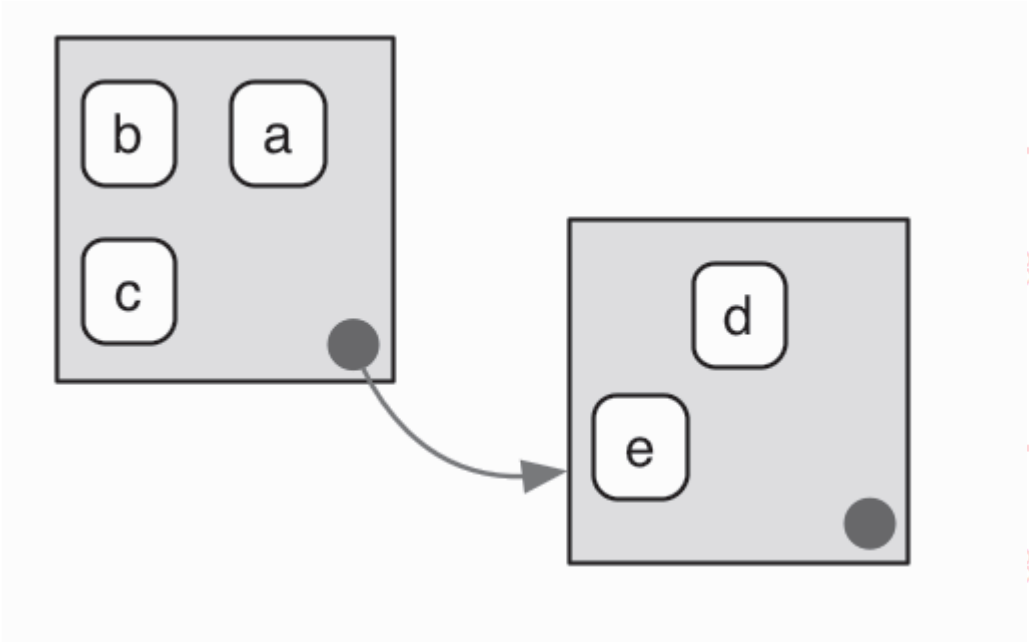
让人迷惑的是，它们也可以指向具有相同值的不同对象：

```
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**。

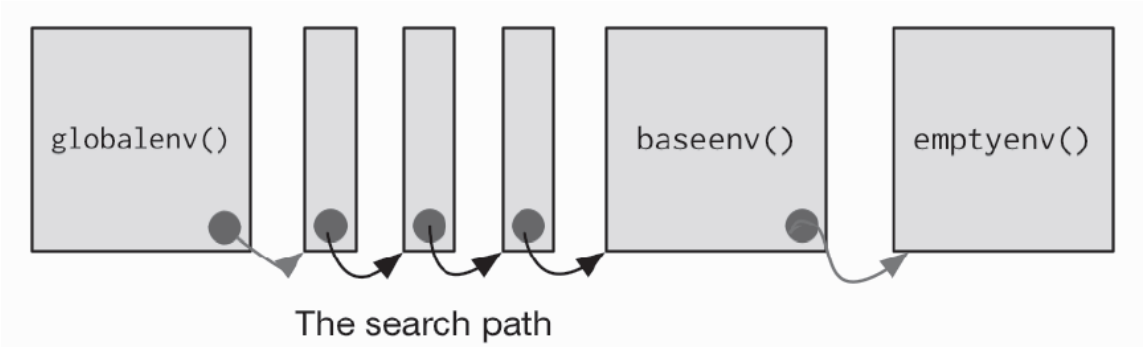
search()会列出全局环境的所有父环境。 这被称为**搜索路径**，因为在这些环境中的对象，都可以从顶层的交互工作区中找到。它为每一个加载的包以及其它 attach()的对象都包含一个环境。 它还包含一个名为 Autoloads 的特殊环境，它用于**按需加载**程序包，以便节省内存(如大型数据集)。

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))
}
}
```

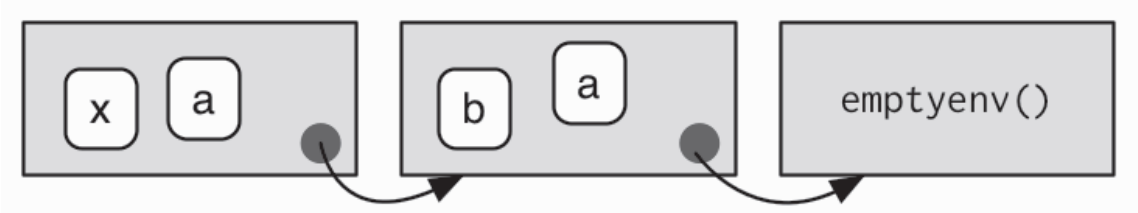
这里有三种情况：

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。

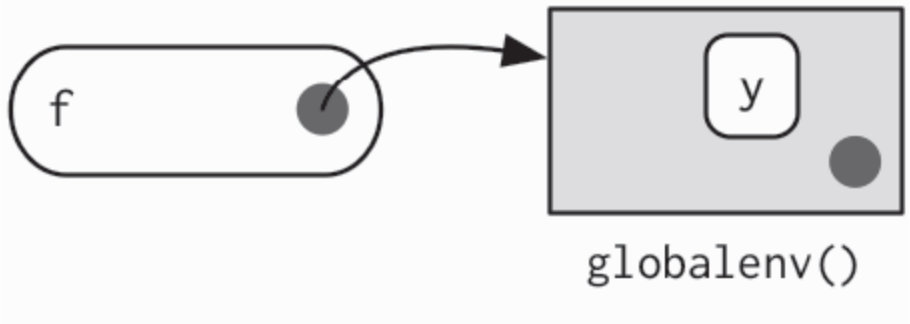
以下部分将解释为什么这些环境很重要，如何访问它们，以及如何使用它们。

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：

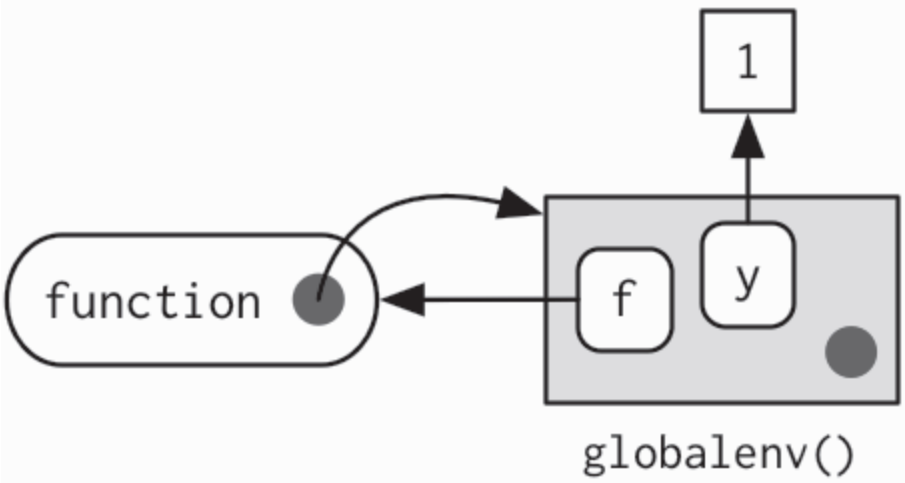
```
y <- 1
f <- function(x) x + y
environment(f)
#> <environment: R_GlobalEnv>
```

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：



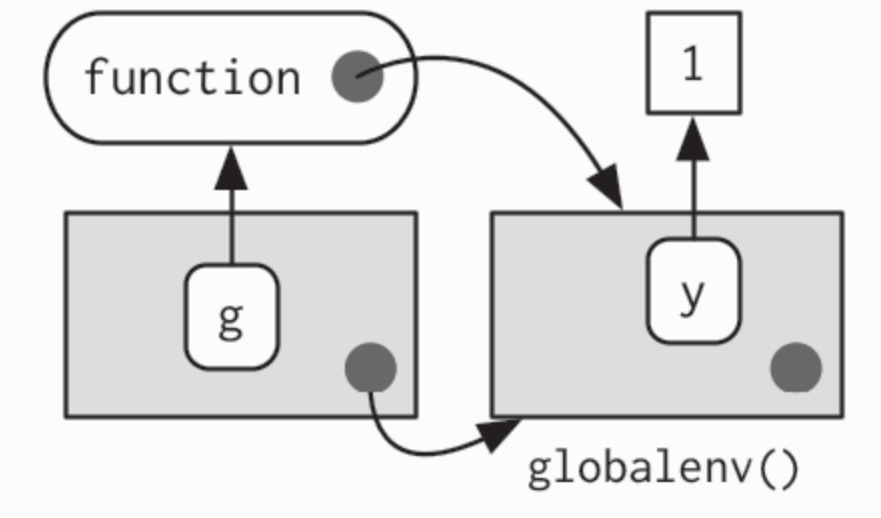
8.3.2 绑定环境

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)。

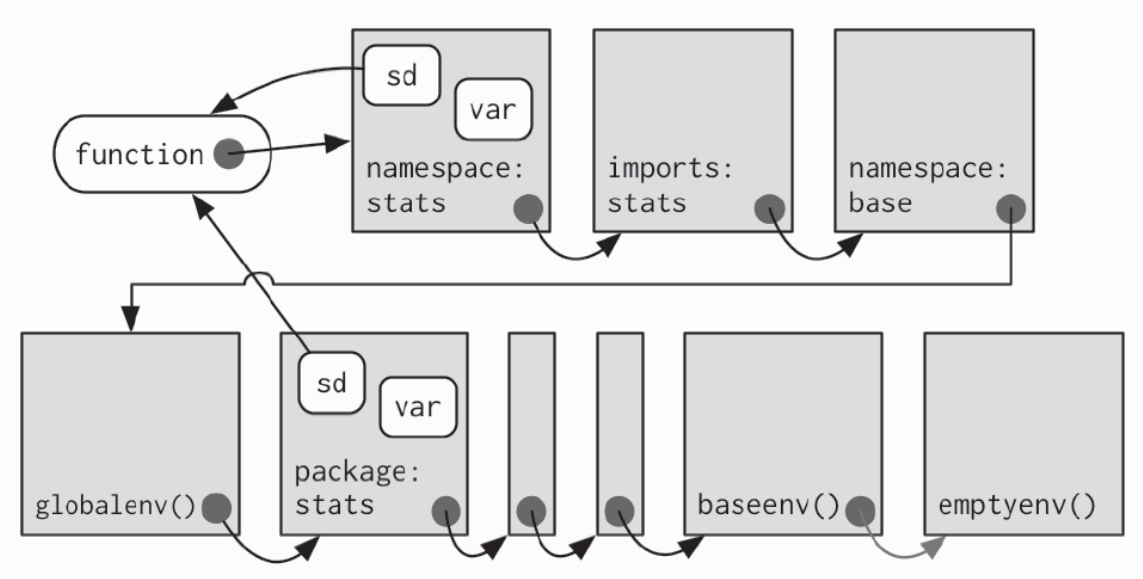
命名空间是使用环境来实现的，基于这样的事实，那么函数不需要存在于它们的**封闭环境**中。 以基本函数 sd()为例。 它的**绑定环境**和**封闭环境**是不同的：

```
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()。

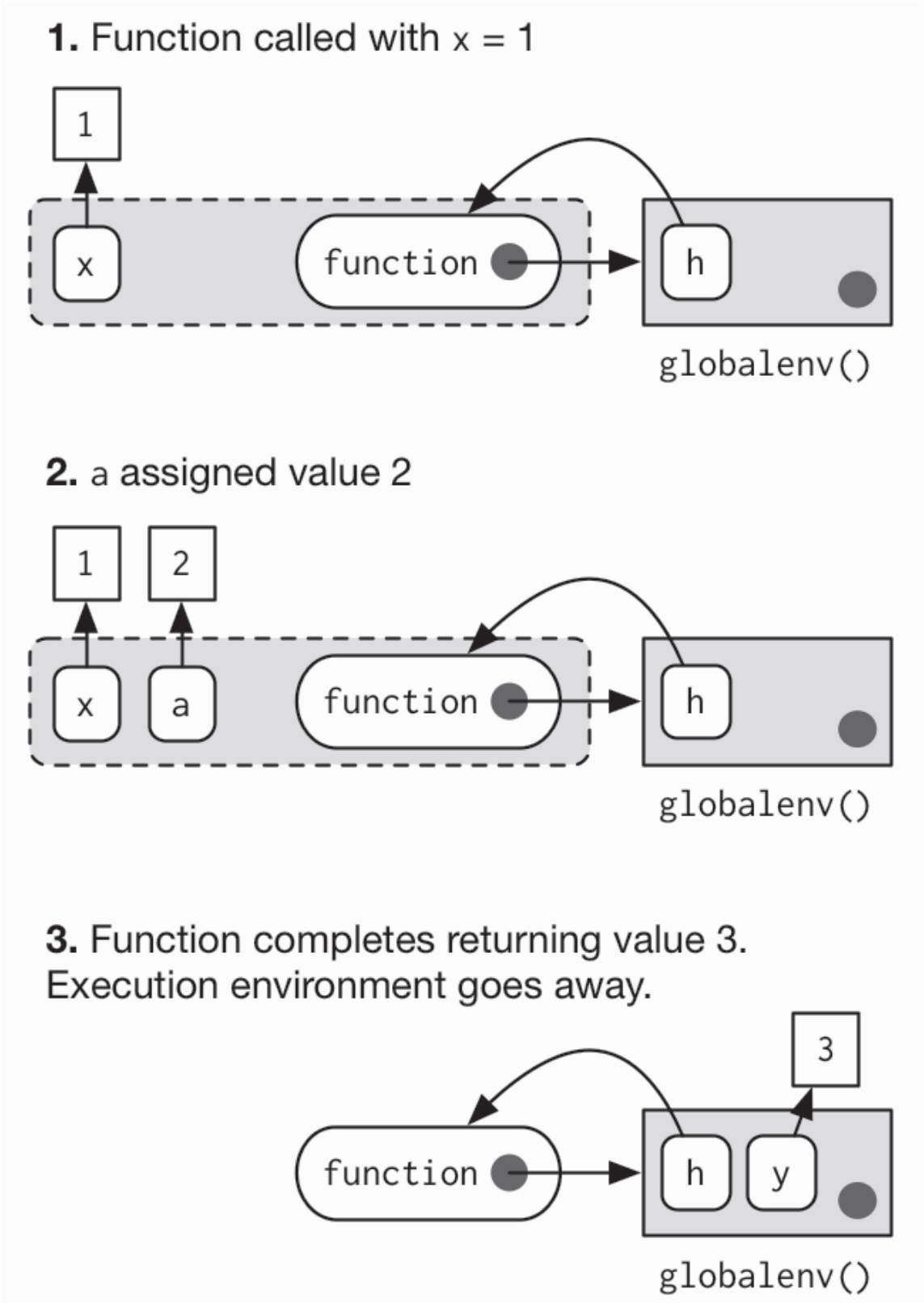
8.3.3 执行环境

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)
```

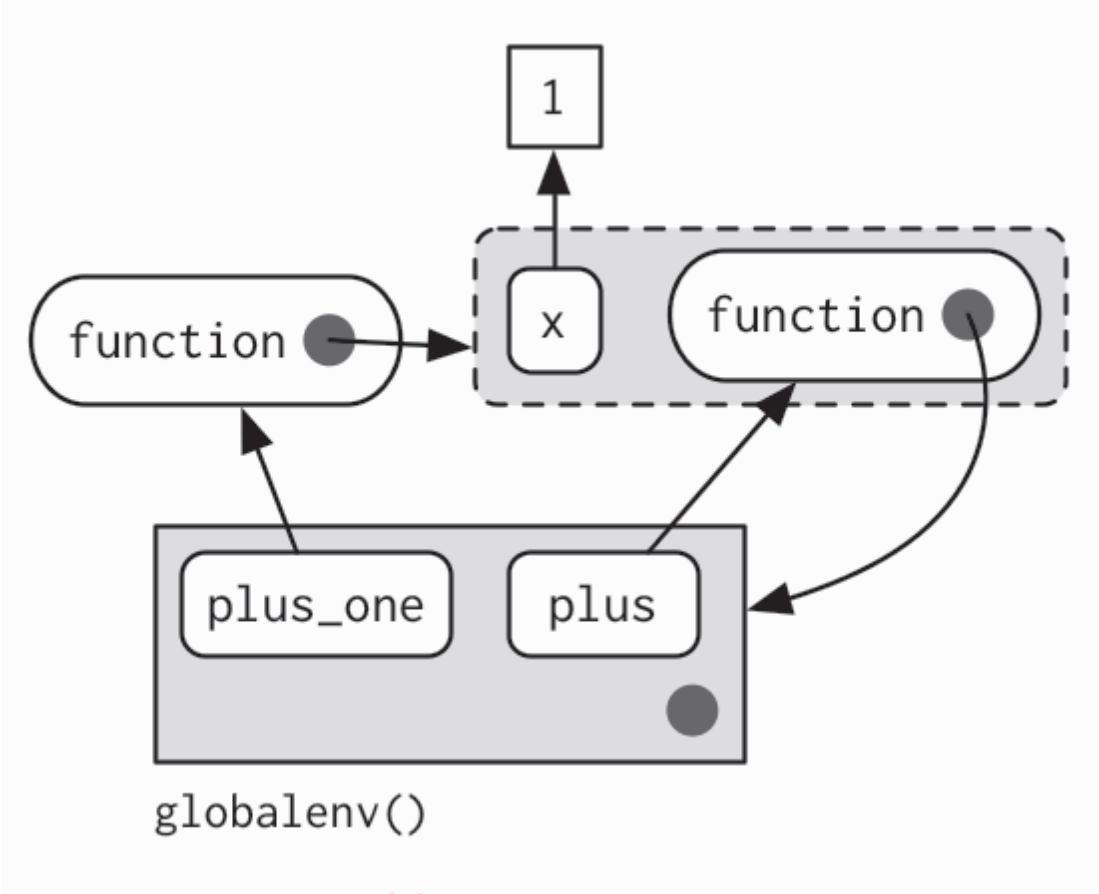
这个函数被调用时，每次都会返回相同的值，这是由于"全新的开始原则"(fresh start principle)，它已经在 6.2.3 节中描述过。 每当函数被调用的时候，一个新的环境将被创建出来管理执行过程。**执行环境**的**父环境**是函数的**封闭环境**。一旦函数执行完毕，这个环境就会被抛弃。让我们使用图形来描述一个更简单的函数。我把函数所属的、围绕着它的**执行环境**，画了虚线边界。

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


1. Function called with x = 1
1
x
function
h
globalenv()
2. a assigned value 2
1
2
x
a
function
h
globalenv()
3. Function completes returning value 3.
Execution environment goes away.
3
function
h
y
globalenv()

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：

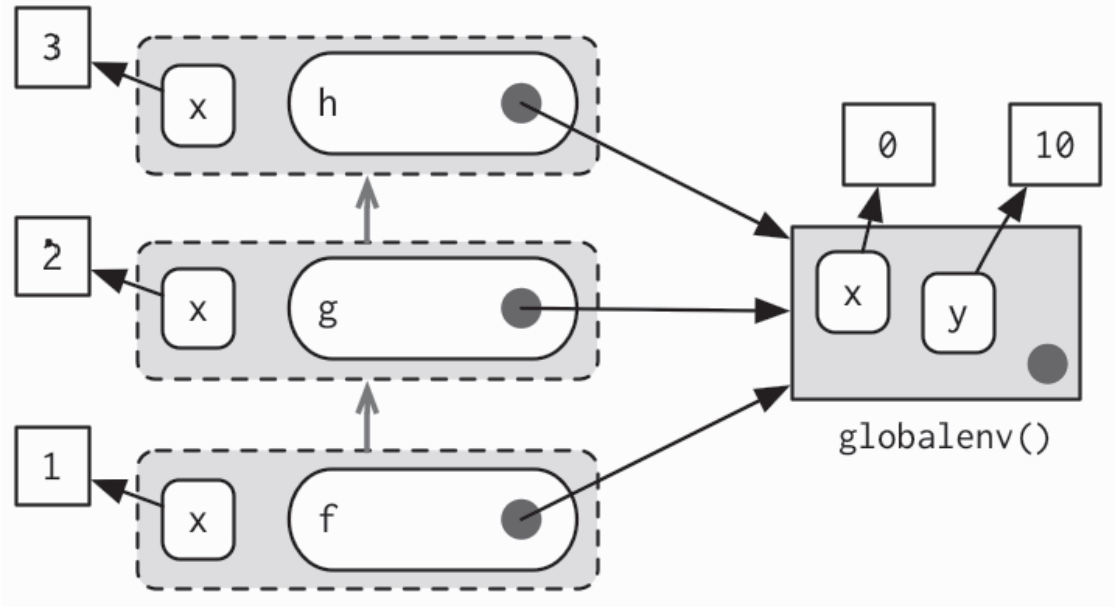
```
f2 <- function() {
x <- 10
function() {
def <- get("x", environment())
cll <- get("x", parent.frame())
list(defined = def, called = cll)
}
}
```

```
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**创建了在当前的环境中，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)。

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意外的问题(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：

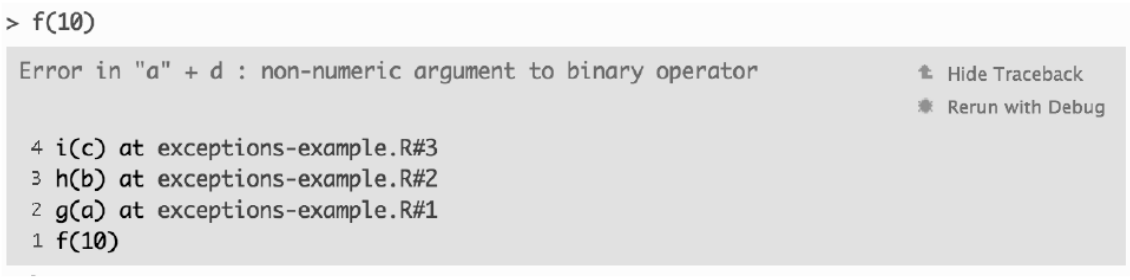
```
f <- function(a) g(a)
g <- function(b) h(b)
h <- function(c) i(c)
i <- function(d) "a" + d
f(10)
```

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：



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。

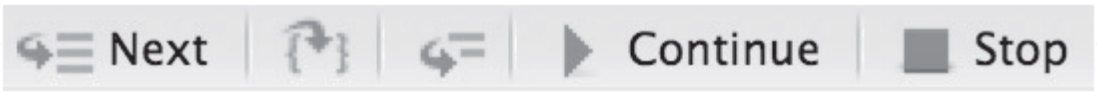
有时，这样已经有足够的信息让你跟踪错误并进行修复了。 但是，多数情况下，这还是不够的。 traceback()显示了错误发生的地方，但是并没有说明是什么原因。 下一个有用的工具是**交互式调试器**，它可以让你暂停执行函数以及交互地研究它的状态。

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。

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

注意：编号在 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，你就会在**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，它将是被计算的代码块(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**可以做任何事情，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(
```

```
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)。

```
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。