R 包开发

发布 *0.0.1*

2020 年 08 月 26 日

Contents

本项目是 *R Packages*（第二版）中文翻译，旨在为学习 R 包开发的人们提供一些微小的帮助。

CHAPTER 1

翻译进度

：翻译完成 ： 还未翻译

- Cover
- Preface
- Introduction
- The whole game
- System setup
- Package structure and state
- Fundamental development workflows
- R code
- Package metadata
- Object documentation
- Vignettes: long-form documentation
- Testing
- Namespace
- External data
- Compiled code
- Installed files

- Other components
- Git and GitHub
- Automated checking
- Releasing a package

CHAPTER 2

相关链接

- **原书**: R Packages
- **原书网页文档版**: R Packages
- **个人译本**: R Packages zh-CN
- GitHub Repo: https://github.com/YuanchenZhu2020/R_Packages_zh_CN

CHAPTER 3

全书目录

3.1 翻译说明

3.1.1 保留英文的情况

1. 人名、地名、书名等专有名词；
2. 用简洁的中文难以清晰表达的词语，放入中文后的括号内。如 bundled ： 压缩的（bundled）。

3.1.2 意译和混用翻译

部分英文单词根据上下文语意翻译，具体如下表：

English	中文翻译	依据
convention	约定、规范	上下文语意
The whole game	整个流程	第二章内容
small toy package	小型的示例程序包	第二章内容
function	功能、函数	上下文语意
helper	帮助函数、函数	上下文语意
basename	文件名	网络词典
render	渲染、渲染生成	文意及网络词典
wrapper function	封装函数	文意
useRs, developRs	使用 useR/developR 的人	（不翻译）
qualified call	合法调用	“有资格的调用”
aggravating	令人烦恼的	上下文语意
bundled	压缩的	第四章描述
venue	环境，原意为聚会场所	上下文语意
complication artefacts	编译文件（已编译的文件）	上下文语意
distributed product	分发式的产品	上下文语意
downstream forms	后面的工作流	“下游”、“形式”
vignettes	使用指南	文意及网络词典
old convention(ch_14)	旧式的	上下文语意
landscape	运行环境	第一版翻译
unergonomic	舒适正确的使用方式	上下文语意
test drive	测试函数（原意为试车）	上下文语意
CRAN Notes	CRAN 注记	第一版翻译

3.1.3 部分名词的简单解释

Workflow：工作流，即工作流程在计算机系统上的自动化。某些工作有着固定的流程，可以分为多个耦合小的步骤，每个步骤由一个或多个软件系统完成。**Bundled**：捆绑的。根据 4.3 中的描述，理解成进行处理的压缩文件形式的程序包。**tar**：打包存档。重点在于将很多文件打包为一个文件，进行存档。原生的 tar 不具有压缩功能，打包后文件的大小甚至会更大。

3.2 排版约定

3.2.1 源码

为防止源码中文本过长，采取了行末 \ 的“硬断行”，但是在列表中由于本人不会“硬断行”，因此源码中仍为一行。具体请参考“序言”$_{\text{temp}}$ 的源码

3.2.2 困惑的单词

翻译过程中，一些困惑的单词会被暂时留置，用说明文字作为下标标记，当它的上下文确定无误后再进行翻译。例如：$\text{Unknowned}_{\text{temp}}$

3.2.3 代码和目录

1. 行内代码或文件目录使用行内代码风格：`Codes/`；

3.2.4 链接

1. 文本形式的链接具有前导空格和后导空格：链接 https://github.com 链接

3.2.5 英文单词和数字

1. 英文单词和数字具有前导空格和后导空格：具有 1 个空格
2. 英文单词和数字的一侧为标点符号时，该侧无空格：使用 Leading and Trailing Spaces。

3.2.6 图片名称

正文中出现的图片，其名称使用如下所示的 HTML 标签进行居中处理：

```
.. raw:: html

    <center> 我是图片的编号和名称</center>
```

3.3 封面

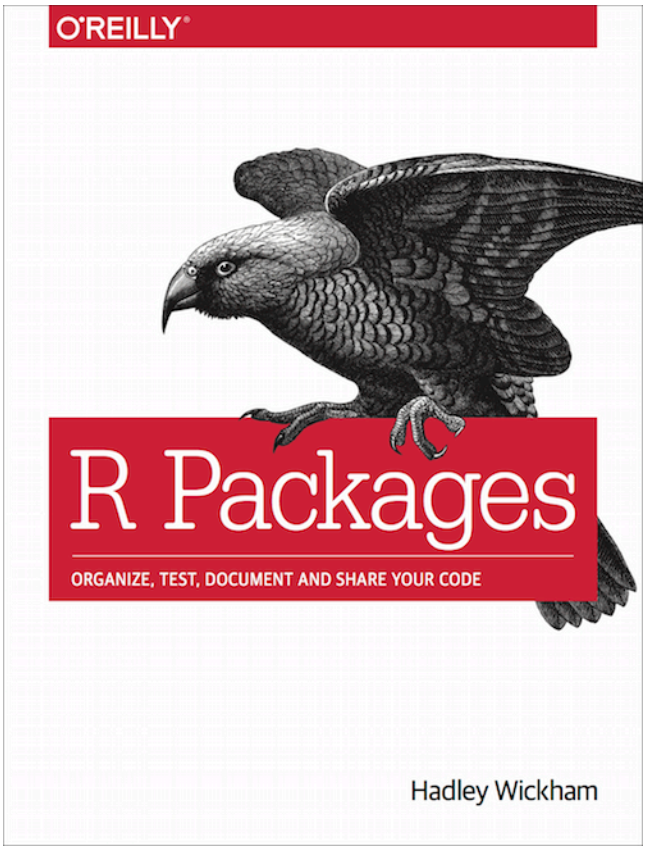
3.3.1 R Packages

Hadley Wickham, *Jennifer Bryan*

3.3.2 R Packages

在 R 中，包（*packages*）是可分享 R 代码的基本单位。它包含可复用的 R 函数、描述怎样使用这些函数的文档和测试数据。在本书中，您将学习如何将代码转换成他人能够轻松下载和使用的程序包。起初，您可能感觉开发这样一个程序包似乎十分麻烦。因此，让我们从基础开始，逐渐提升我们的开发水平。您不必担心您的第一个版本并不完美，因为下一版本将会变得更好。

我们在这里开发本书的第二版，您仍然可以从 http://r-pkgs.had.co.nz/ 获得本书的第一版。



3.4 序言

3.4.1 序言

欢迎来到第二版 R packages！如果熟悉第一版本的内容，您会发现这篇序言介绍了第二版主要的内容更改，以便您能够将学习的重点放在新的部分上。

此版本有十个主要目标：

- 更新以反映 `devtools` 程序包中的更改，特别是其“有意识地解耦”，变成了一组更小、功能更集中的程序包（packages）。
- 扩展了工作流（workflows）和流程（process）内容的覆盖范围，并且介绍了构成 R 包的所有重要的、可变动的部分。temp
- ……还有更多

各章的详细信息：

- 新的第二章：“整个流程”，概述了程序包开发过程中关键步骤。
- 新的第三章：“系统设置”已经从第一版的导言中删除，在这里对其做了更加详细的描述。

- 删除了第一版第六章“R 代码”中的“组织你的函数”和“代码风格”段落，以便使用**在线风格指南**，https://style.tidyverse.org/。代码风格指南和新的样式程序包 `styler` 是配合使用的（Müller and Walthert 2018），够自动为您的代码应用许多样式规则。
- 之前被称为“包的结构”的章节已经被扩展并分为两章，一章介绍了包的结构和状态（Chapter 4），另一章介绍了工作流（`workflows`）和工具（`tools`）（Chapter 5）。
- 关于 `git/github` 的一些内容

3.4.2 参考文献

Müller, Kirill, and Lorenz Walthert. 2018. *Styler: Non-Invasive Pretty Printing of R Code*. http://styler.r-lib.org/.

3.5 第一章简介

在 R 中，包（packages）是可分享 R 代码的基本单位。程序包将代码、数据、文档和测试代码整合到一起，很容易与人分享。截止到 2019 年 6 月，在综合 R 包存档网络 CRAN（**C**omprehensive **R** **A**rchive **N**etwork），或者说 R 包的公共发布平台上已经有 14,000 多个可用的程序包。拥有这么众多的程序包是 R 成功的原因之一：也许有人已经解决了您正在尝试解决的问题，因此您可以通过下载程序包来从他们的工作中获益。

如果您正在阅读这本书，那么就应该已经知道如何使用包了：

- 从 CRAN 下载安装包：`install.packages("x")`
- 在 R 中使用包：`library("x")`
- 获取包的帮助文档：`package?x` 以及 `help(package = "x")`

本书的目的是教您如何开发程序包，这样您就可以自己编写自己的程序包，而不仅仅是使用他人的程序包。为什么我们要写程序包呢？一个令人信服的理由是，您想要与他人分享自己的代码。将您的代码整合到一个程序包中能让他人更加轻松地使用它，因为和您一样，他们也知道如何使用包。如果您的代码放在包中，任何 R 的用户都可以便利地下载、安装并且学会如何使用。

但对您来说，即使从不共享代码，包也很有用。正如 Hilary Parker 在介绍软件包时所说：“严格来说，这不一定与共享代码有关（虽然这是一个附加的好处），它主要是能够节省你自己的时间。”将您的代码组织在程序包中能够使工作变得更轻松，这是因为包有一些通用的约定。比如，你需要将 R 代码放在 `R/` 目录下，将测试放在 `tests/` 目录下，还有将数据放在 `data/` 目录下。这些约定很有用，因为：

- 它们能节省时间——您不必考虑组织程序包的最佳方法，只需要按照模板来就可以了。
- 标准化的规范带来标准化的工具——如果您遵循 R 包的封装规范，那么您能够免费获得许多工具。

正如 Marwick, Boettiger 以及 Mullen 在 (Marwick, Boettiger, and Mullen 2018a) (Marwick, Boettiger, and Mullen 2018b) 中所说，您甚至可以用程序包来构建您的数据分析流程。

3.5.1 1.1 开发理念

这本书支持了我们的程序包开发理念：任何能够自动化的东西都应该自动化。尽量减少手动操作。用函数完成尽可能多的事情。这样是希望您将时间用于思考您想要包做什么工作，而不是包结构的各种细节。

这种开发理念主要是 devtools 开发包来实现的，这个程序包是让通用开发任务自动化的 R 函数套件中的代表。devtools 在 2018 年 10 月发布了 2.0.0 版本，这标志着其内部结构重组为一系列功能更具针对性的程序包，而它则更像是一个元程序包（meta-package ）。usethis 程序包是您最有可能与之交互的子程序包，我们将在 3.2 节解释 devtools - usethis 之间的关系。

像往常一样，devtools 程序包的目的是让程序包的开发尽可能的轻松便利。它囊括了第一作者 Hadley Wickham 这些年来作为一名多产的独立开发者的最佳实践经验。最近，他在 RStudio 集合了一个由 10 名开发人员组成的团队，共同负责约 150 种开源 R 包的维护，其中包括被称为 tidyverse 的著名程序包。这个团队的影响力使得我们能够在一个巨大的规模上探索所有可能出现的错误与问题。幸运的是，它还让我们有机会与专家和友善的同事一起反思成功与失败。我们尝试找到一些能够使包的维护者和使用者更快乐地工作生活的做法，而正是在 devtools 程序包里，您将看到这些方法是如何具体实现的。

> 在本书中，我们将在这种特殊格式的部分重点介绍：如何使用 `Rstudio` 的特殊方法加快包的开发工作流程。

`devtools` 与 `RStudio` 联系紧密、携手共进，而后者正是我们认为对大多数 R 用户而言最佳的开发环境。它的主要替代者是 `Emacs Speaks Statistics(ESS)`，如果您愿意花时间学习 `Emacs` 并根据自己的需求来自定义它，那么它将是一个回报丰厚的开发环境。`ESS` 的历史可以追溯到 20 年前（甚至比 R 还早!），但是它至今仍很活跃，本书中描述的许多工作流程也可以在上面使用。对于那些忠于 `Vim` 的人，我们推荐使用 Nvim-R plugin 插件。

`devtools` 和 `RStudio` 一起，让您无需关注包是怎样建立的这种低级细节。但是当您开始开发更多的程序包时，我们强烈建议你去了解这些细节。有关软件包开发细节的最好的官方资源，始终是官方 R 扩展开发手册。然而，如果您不熟悉包的基础知识，是很难理解这本手册的。它也很详尽，涵盖了所有可能的包的组件，而不是像本书一样专注于那些最常见和最有用的组件。一旦你掌握了 R 包的基础知识，并且想深入了解更多的知识，那么官方 R 扩展开发手册 将是一个很有用的资源。

3.5.2 1.2 本书内容

第二章介绍了一个小型的示例程序包的开发。这是为了在我们深入研究 R 包的关键组件之前，描绘开发的大图景并提出对一个工作流程的建议。

第三章介绍了如何为开发程序包准备好您的系统，这与仅仅运行 R 脚本相比具有更多要求。这一章包括对一些可选设置的建议，这些设置可以使您的工作流程更加轻松，从而带来更高质量的产品。

程序包的基本结构以及它在不同状态下如何变化将在第 ?? 章中进行说明。

第五章介绍了程序包开发人员反复提出的核心工作流程。本章还介绍了我们喜欢的工具，比如 devtools / usethis 和 RStudio 与这些工具设计时的指导理念 意译 之间的联系。

本书的后续章节将详细介绍 R 包的每个组件。它们按重要性顺序被大致组织起来：

- 第六章，R 代码：最重要的目录是 R 代码所在的目录 `R/`。即使是只有该目录的程序包仍然是可以使用的。（的确，即使您在本章结束后不再阅读，您仍然会学到一些有用的新技能。）
- 第七章，包的元数据：`DESCRIPTION` 文件描述您的包需要依赖什么来正常工作。如果您要共享程序包，也会使用 `DESCRIPTION` 文件来描述程序包能做什么，谁可以使用它（许可证）以及在出现问题时与谁联系。
- 第八章，对象文档：如果您希望其他人（包括未来的您!）了解如何使用程序包中的函数，就需要撰写它们的使用文档。我们将向您展示如何使用 roxygen2 为函数撰写文档。我们建议使用 roxygen2 ，因为它可以让您在编写代码的同时直接撰写文档，同时能够 疑问 生成 R 的标准格式的文档。
- 第九章，Vignettes：函数文档介绍了包中每个函数的细节，而 Vignettes 则描绘了如何从整体意义上使用包的功能。它们是长篇文档，显示了如何结合程序包的多个功能来解决实际问题。我们将向您展示如何使用 Rmarkdown 和 knitr 创建 Vignettes 。
- 第十章，测试：为确保您的程序包按照设计的那样工作（并在您修改它后能继续工作），编写定义了正确行为并在函数中断时提醒您的单元测试至关重要。在本章中，我们将教您如何使用 testthat 包将您正在做的的非正式的交互测试转换为正式的、自动化的测试。
- 第十一章，命名空间：为了和其他的包很好地协作，您的程序包需要定义它能够让其他程序包使用哪些函数，以及它需要使用其他程序包的哪些函数。这是 `NAMESPACE` 文件的工作，我们将向您展示如何使用 roxygen2 生成它。`NAMESPACE` 是开发 R 程序包中更具挑战性的部分之一，但如果您希望程序包可靠地工作，那么掌握它是至关重要的。
- 第十二章，外部数据：`data/` 目录允许您在包中包含数据。您可能会用这种方式在包种捆绑数据，以使 R 用户易于访问，或者只是在文档中提供令人信服的例子。
- 第十三章，编译后的代码：R 代码是为提高人类效率而设计的，而不是为了提高计算机效率，因此，如果包含一个允许您编写快速运行的代码的工具将非常有用。`src/` 目录允许您引入快速运行的、编译好的 C 和 C++ 代码，以解决程序包中的性能瓶颈。
- 第十五章，其他组件：本章介绍了其他几个很少用到的组件：`demo/`，`exec/`，`po/` 和 `tools/`。

最后几章介绍了几个通用的最佳做法，它们并没有专门针对包中某个特定的目录：

- 第十六章，Git 和 GitHub ：掌握版本控制系统对于轻松地与他人协作至关重要，即使单独工作，它也非常有用，因为它使您可以轻松地将文件回退到错误之前。在本章中，您将学习如何基于 RStudio 使用流行的 Git 和 GitHub。
- 第十七章，自动检查：R 以 `R CMD check` 的形式提供了非常有用的自动质量检查。定期运行它们是避免许多常见错误的好方法。有时它的结果可能有点含糊不清，因此我们提供了一个全面的备忘清单，可

以帮助您将警告变为可理解的形式。意译

- 第十八章，发布：包的生命周期随着向公众发布而结束。本章比较了发布平台的两个主要选择（CRAN 和 GitHub ），并提供了有关管理流程的一般建议。

还有很多东西需要学习，但不要感到不知所措。从一些有用功能的最小一部分开始（例如，只有一个 `R/` 目录!），然后逐步建立并完善它。套用禅师铃木俊隆（Shunryu Suzuki ）的话："每个包都以它完美的方式存在——只是可以稍作改进"。出处?

3.5.3 1.3 致谢

自第一版 R Packages 发行以来，支持本书描述的工作流程的程序包已经得到了广泛的开发。原来的 devtools，roxygen2 和 testthat 的三重组合已经扩展为一系列由 devtools 的 "有意识的解耦" 所创建的程序包。由于它们与 devtools 的渊源，大多数程序包都源自 Hadley Wickham（HW）。还有许多其他重要的贡献者，其中许多人现在成为了维护者：

- devtools: HW, Winston Chang, Jim Hester (maintainer, >= v1.13.5)
- usethis: HW, Jennifer Bryan (maintainer >= v1.5.0)
- roxygen2: HW (maintainer), Peter Danenburg, Manuel Eugster
- testthat: HW (maintainer)
- desc: Gábor Csárdi (maintainer), Kirill Müller, Jim Hester
- pkgbuild: HW, Jim Hester (maintainer)
- pkgload: HW, Jim Hester (maintainer), Winston Chang
- rcmdcheck: Gábor Csárdi (maintainer)
- remotes: HW, Jim Hester (maintainer), Gábor Csárdi, Winston Chang, Martin Morgan, Dan Tenenbaum
- revdepcheck: HW, Gábor Csárdi (maintainer)
- sessioninfo: HW, Gábor Csárdi (maintainer), Winston Chang, Robert Flight, Kirill Müller, Jim Hester

待办事项：第二版即将完成时，请重新阅读本节的其余部分。当前适用于并使用第 1 版的用词表示。

通常，我学习正确操作方法的唯一办法就是首先以错误的方法进行操作。由于遇到了许多程序包开发错误，我要感谢所有 CRAN 维护人员，尤其是 Brian Ripley, Uwe Ligges 和 Kurt Hornik。

本书是公开编写和修订的，它的确是社区的工作成果：许多人阅读原稿，修正错字，提出改进建议并提供内容。没有这些贡献者，这本书的质量将像现在看到的那样好，我们非常感谢他们的帮助。

特别感谢 Peter Li，他从头到尾阅读了本书，并提供了许多解决方案。我也非常感谢审稿人（Duncan Murdoch, Karthik Ram, Vitalie Spinu and Ramnath Vaidyanathan）花费时间阅读本书并给予我详尽的反馈意见。

感谢所有通过 GitHub（按字母顺序）提交改进的贡献者：`@aaronwolen`, `@adessy`, Adrien Todeschini, Andrea Cantieni, Andy Visser, `@apomatix`, Ben Bond-Lamberty, Ben Marwick, Brett K, Brett Klamer,

@contravariant, Craig Citro, David Robinson, David Smith, @davidkane9, Dean Attali, Eduardo Ariño de la Rubia, Federico Marini, Gerhard Nachtmann, Gerrit-Jan Schutten, Hadley Wickham, Henrik Bengtsson, @heogden, Ian Gow, @jacobbien, Jennifer (Jenny) Bryan, Jim Hester, @jmarshallnz, Jo-Anne Tan, Joanna Zhao, Joe Cainey, John Blischak, @jowalski, Justin Alford, Karl Broman, Karthik Ram, Kevin Ushey, Kun Ren, @kwenzig, @kylelundstedt, @lancelote, Lech Madeyski, @lindbrook, @maiermarco, Manuel Reif, Michael Buckley, @MikeLeonard, Nick Carchedi, Oliver Keyes, Patrick Kimes, Paul Blischak, Peter Meissner, @PeterDee, Po Su, R. Mark Sharp, Richard M. Smith, @rmar073, @rmsharp, Robert Krzyzanowski, @ryanatanner, Sascha Holzhauer, @scharne, Sean Wilkinson, @SimonPBiggs, Stefan Widgren, Stephen Frank, Stephen Rushe, Tony Breyal, Tony Fischetti, @urmils, Vlad Petyuk, Winston Chang, @winterschlaefer, @wrathematics, @zhaoy.

用于提示工作流程的灯泡图像来自 www.vecteezy.com。

3.5.4 1.4 约定

在整本书中，我们用 `foo()` 来表示函数，用 `bar` 来表示变量和函数参数，以及使用 `baz/` 来表示路径。

较大的代码块将输入和输出混合在一起。输出带有注释，因此，如果您有本书的电子版本，例如，访问 https://r-pkgs.org，则可以轻松地将示例复制并粘贴到 R 中。输出注释看起来像 '#>'，这将它们与常规注释区分开。

3.5.5 1.5 Colophon

版权页标记

这本书是在 RStudio 中使用 R Markdown 和 bookdown 编写的。该网站由 Netlify 托管，并在 Travis-CI 每次提交后自动更新。完整的资源可从 GitHub 获得。

该书的该版本使用以下内容构建：

```
library(devtools)
#> Loading required package: usethis
library(roxygen2)
library(testthat)
#>
#> Attaching package: 'testthat'
#> The following object is masked from 'package:devtools':
#>
#>     test_file
devtools::session_info()
#>  Session info
#>  setting  value
#>  version  R version 4.0.2 (2020-06-22)
```

(下页继续)

(续上页)

```
#>  os       macOS Catalina 10.15.6
#>  system   x86_64, darwin17.0
#>  ui       X11
#>  language (EN)
#>  collate  en_US.UTF-8
#>  ctype    en_US.UTF-8
#>  tz       UTC
#>  date     2020-08-18
#>
#>  Packages
#>  package     * version     date       lib source
#>  assertthat    0.2.1       2019-03-21 [1] CRAN (R 4.0.2)
#>  backports     1.1.8       2020-06-17 [1] CRAN (R 4.0.2)
#>  bookdown      0.20        2020-06-23 [1] CRAN (R 4.0.2)
#>  callr         3.4.3       2020-03-28 [1] CRAN (R 4.0.2)
#>  cli           2.0.2       2020-02-28 [1] CRAN (R 4.0.2)
#>  crayon        1.3.4       2017-09-16 [1] CRAN (R 4.0.2)
#>  desc          1.2.0       2018-05-01 [1] CRAN (R 4.0.2)
#>  devtools    * 2.3.1.9000  2020-08-16 [1] Github (r-lib/devtools@df619ce)
#>  digest        0.6.25      2020-02-23 [1] CRAN (R 4.0.2)
#>  ellipsis      0.3.1       2020-05-15 [1] CRAN (R 4.0.2)
#>  evaluate      0.14        2019-05-28 [1] CRAN (R 4.0.1)
#>  fansi         0.4.1       2020-01-08 [1] CRAN (R 4.0.2)
#>  fs            1.5.0       2020-07-31 [1] CRAN (R 4.0.2)
#>  glue          1.4.1       2020-05-13 [1] CRAN (R 4.0.2)
#>  htmltools     0.5.0       2020-06-16 [1] CRAN (R 4.0.2)
#>  knitr         1.29        2020-06-23 [1] CRAN (R 4.0.2)
#>  lifecycle     0.2.0       2020-03-06 [1] CRAN (R 4.0.2)
#>  magrittr      1.5         2014-11-22 [1] CRAN (R 4.0.2)
#>  memoise       1.1.0       2017-04-21 [1] CRAN (R 4.0.2)
#>  pkgbuild      1.1.0       2020-07-13 [1] CRAN (R 4.0.2)
#>  pkgload       1.1.0       2020-05-29 [1] CRAN (R 4.0.2)
#>  prettyunits   1.1.1       2020-01-24 [1] CRAN (R 4.0.2)
#>  processx      3.4.3       2020-07-05 [1] CRAN (R 4.0.2)
#>  ps            1.3.4       2020-08-11 [1] CRAN (R 4.0.2)
#>  purrr         0.3.4       2020-04-17 [1] CRAN (R 4.0.2)
#>  R6            2.4.1       2019-11-12 [1] CRAN (R 4.0.2)
#>  Rcpp          1.0.5       2020-07-06 [1] CRAN (R 4.0.2)
#>  remotes       2.2.0       2020-07-21 [1] CRAN (R 4.0.2)
#>  rlang         0.4.7       2020-07-09 [1] CRAN (R 4.0.2)
```

(下页继续)

(续上页)

```
#>  rmarkdown     2.3         2020-06-18 [1] CRAN (R 4.0.2)
#>  roxygen2    * 7.1.1       2020-06-27 [1] CRAN (R 4.0.2)
#>  rprojroot     1.3-2       2018-01-03 [1] CRAN (R 4.0.2)
#>  sessioninfo   1.1.1       2018-11-05 [1] CRAN (R 4.0.2)
#>  stringi       1.4.6       2020-02-17 [1] CRAN (R 4.0.2)
#>  stringr       1.4.0       2019-02-10 [1] CRAN (R 4.0.2)
#>  testthat    * 2.99.0.9000 2020-08-16 [1] Github (r-lib/testthat@9e643d8)
#>  usethis     * 1.6.1.9001  2020-08-16 [1] Github (r-lib/usethis@860c1ea)
#>  withr         2.2.0       2020-04-20 [1] CRAN (R 4.0.2)
#>  xfun          0.16        2020-07-24 [1] CRAN (R 4.0.2)
#>  xml2          1.3.2       2020-04-23 [1] CRAN (R 4.0.2)
#>  yaml          2.2.1       2020-02-01 [1] CRAN (R 4.0.2)
#>
#> [1] /Users/runner/work/_temp/Library
#> [2] /Library/Frameworks/R.framework/Versions/4.0/Resources/library
```

3.5.6 参考文献

Marwick, Ben, Carl Boettiger, and Lincoln Mullen. 2018a. "Packaging Data Analytical Work Reproducibly Using R (and Friends)." *The American Statistician* 72 (1). Taylor & Francis:80–88. https://doi.org/10.1080/00031305.2017.1375986.

Marwick, Ben, Carl Boettiger, and Lincoln Mullen. 2018b. "Packaging Data Analytical Work Reproducibly Using R (and Friends)." *PeerJ Preprints* 6 (March):e3192v2. https://doi.org/10.7287/peerj.preprints.3192v2.

3.6 第二章整个流程

剧透警告!

本章介绍了一个小型示例程序包的开发。这是为了在我们深入研究 R 包的关键组件之前，描绘开发的大图景并提出对一个工作流程的建议。

为了保持快节奏，我们利用了 devtools 程序包和 RStudio IDE 中的现代化的便利功能在之后的章节中，我们将更详细地介绍它们能帮助我们做些什么。

3.6.1 2.1 加载 devtools 和相关程序包

无论您是否进入了 R 项目，您都可以从任何开放的 R 控制台（R Console）完成以下步骤。

加载 devtools 程序包，在支持程序包开发的各个方面的程序包中，它是个中翘楚。

```
library(devtools)
#> Loading required package: usethis
```

仅仅出于演示目的，我们使用 fs 帮助我们在文件系统上工作，使用 tidyverse 进行轻量数据的整理。

```
library(tidyverse)
library(fs)
```

3.6.2 2.2 示例程序包：foofactors

我们使用 devtools 中的各种功能从头开始构建一个小型的、作为示例的程序包，它具有在已发布的包中十分常见的功能：存疑

- 满足特定需求的函数，例如与因子（factors ）打交道的“助手”。存疑
- 可以访问已完成的工作流程以进行安装、获得帮助以及检查包的基本质量。优化表达
- 版本控制和开放的开发过程。- 在您的工作中，这是完全可选的，但我们建议这样做。您将看到 Git 和 GitHub 如何帮助我们展示程序包开发的所有中间阶段。
- 通过 roxygen2 建立各个函数的文档。
- 使用 testthat 进行单元测试。
- 通过可执行文件整体了解软件包的文件 `README.Rmd`。

我们称这个程序包包为 **foofactors**，它将有几个处理因子（factors ）的函数。请注意，这些功能非常简单，并且绝对不是本章的重点！有关正确处理因子（factors ）的软件包，请参阅 forcats。

构建 foofactors 包本身不是我们的目标。它只是一个示例，用于演示使用 devtools 进行程序包开发的典型工作流程。

3.6.3 2.3 看看成品

我们使用了 Git 版本控制系统追踪 foofactors 包的开发过程。这存粹是可选的，您当然可以不执行这个步骤。但是它有一个附加的好处：我们将其连接到 GitHub 上的远程储存库，这意味着您能够通过访问 GitHub 上的 foofactors 库来观看我们努力取得的光荣成果：https://github.com/jennybc/foofactors。通过检查 commit history（尤其是版本差异），您能够清楚地看到其下列出的过程中每个步骤更改了什么。

TODO: I think these diffs are extremely useful and would like to surface them better here.

3.6.4 2.4 `create_package()`

调用 `create_package()`，选择计算机一个目录，初始化新的程序包（如果有需要，还可以新建一个目录）。关于更多的信息，请参见 5.1.3 节。

仔细选择创建新程序包的目录。它可以与其他 R 项目共同位于主目录下。但是，它不应该被嵌套在另一个 RStudio 项目、R 包或者是 Git 存储库中。它也不应该在 R 的程序包库中，因为这个库目录下包含有已经被构建和安装的程序包。将这里创建的源程序包转换为已安装程序包的过程，是 devtools 包可以促进完成的一部分工作。不要尝试为此做 devtools 的工作！ 存疑 关于更多的信息，请参见 5.1.4 节。

将您选择的路径作为参数传入 `create_package()` 中，如下所示：

```
create_package("~/path/to/foofactors")
```

我们必须在临时目录中工作，因为这本书是在云上以非交互形式构建的。在幕后，我们执行自己的 `create_package()` 命令，但如果我们的输出与你的输出略有不同，请不要感到惊讶。

```
#> ✓ Creating '/tmp/RtmpsJ87ir/foofactors/'
#> ✓ Setting active project to '/tmp/RtmpsJ87ir/foofactors'
#> ✓ Creating 'R/'
#> ✓ Writing 'DESCRIPTION'
#> Package: foofactors
#> Title: What the Package Does (One Line, Title Case)
#> Version: 0.0.0.9000
#> Authors@R (parsed):
#>     * First Last <first.last@example.com> [aut, cre] (<https://orcid.org/YOUR-ORCID-
↪ID>)
#> Description: What the package does (one paragraph).
#> License: What license it uses
#> Encoding: UTF-8
#> LazyData: true
#> ✓ Writing 'NAMESPACE'
#> ✓ Writing 'foofactors.Rproj'
#> ✓ Adding '.Rproj.user' to '.gitignore'
#> ✓ Adding '^foofactors\\.Rproj$', '^\\.Rproj\\.user$' to '.Rbuildignore'
#> ✓ Setting active project to '<no active project>'
#> ✓ Setting active project to '/tmp/RtmpsJ87ir/foofactors'
```

如果您在 RStudio 中工作，您会发现自己进入了一个新的 RStudio 程序界面，它已经打开了新的 foofactors 包（或项目）目录。如果您处于某种原因需要手动执行这个操作，请进入该目录并双击 `foofactors.Rproj`。RStudio 对于软件包做了特殊处理，您现在应该可以在 *Evironment* 和 *History* 所在的窗格中看到 *Build* 选项卡。

TODO: good place for a screenshot.

在这个新目录里的内容是一个 R 包，也许还是个 RStudio Project？以下是文件清单（在本地，您可以查看 *Files* 选项卡）：

```
#> # A tibble: 6 x 2
#>   path             type
#>   <fs::path>       <fct>
#> 1 .Rbuildignore    file
#> 2 .gitignore       file
#> 3 DESCRIPTION      file
#> 4 NAMESPACE        file
#> 5 R                directory
#> 6 foofactors.Rproj file
```

在文件浏览器中，转到 *More* > *Show Hidden Files* 来切换隐藏文件（也称为 dotfiles）的可见性。一些文件是始终可见的，但有时您可能会希望看到全部的文件。

- `.Rbuildignore` 列出了我们编写 R 包时需要的，但是从源代码构建 R 包时并不应该包含进来的文件。详情请见 ??。
- `.Rproj.user`，如果有的话，它是 RStudio 内部使用的目录。
- `.gitignore` 为 Git 的使用做好准备。它将忽略一些由 R 或 RStudio 创建的标准的幕后文件。即使您不打算使用 Git，它也是没有妨害的。
- `DESCRIPTION` 提供了有关您的程序包的元数据。我们很快将开始编写它。
- `NAMESPACE` 声明了程序包导出以供外部使用的函数以及程序包从其他包导入的外部函数。现在，除了一个注释声明这是一个我们不会手工编辑的文件外，它是空的。
- `R/` 目录是程序包的“业务端”。它很快将包含带有函数声明的 `.R` 文件。
- `foofactors.Rproj` 是使得该目录成为 RStudio 项目的文件。即使你不使用 RStudio，这个文件也是没有妨害的。如果您不想创建它，可以使用 `create_package(..., rstudio = FALSE)`。详情请见 5.2。

3.6.5 2.5 `use_git()`

Git 或其他版本控制系统的使用是可选项，但是长期来看我们建议您使用。我们将在 16 中解释其重要性。

foofactors 目录是 R 源码包和 RStudio 项目。现在，我们使用 use_git() 让它也变成一个 Git 存储库。

```
use_git()
#> ✓ Initialising Git repo
#> ✓ Adding '.Rhistory', '.RData' to '.gitignore'
```

在交互式会话中，系统将询问您是否要在此处提交这些文件，您应该接受这个提议。在 R 中，您不会看到这些，但是在幕后，我们会进行同样的操作。

有什么新内容吗？仅仅是创建了 .git 目录，该目录在大多数环境中都是隐藏的，包括 RStudio 文件浏览器。但是它的存在证明我们确实在这里初始化了 Git 存储库。

```
dir_info(all = TRUE, regexp = "^[.]git$") %>%
    select(path, type)
#> # A tibble: 1 x 2
#>   path       type
#>   <fs::path> <fct>
#> 1 .git       directory
```

如果您使用的是 RStudio，它可能会请求在此项目中重新启动。您可以通过退出然后双击 foofactors.Rproj 重新启动来手动执行此操作。现在，除了程序包的开发支持外，您还可以在 *Git* 选项卡中访问一个基础的 Git 客户端。*Git* 选项卡与 *Environment/History/Build* 在同一个窗格中。

TODO: good place for a screenshot.

单击 *History* （时钟图标），如果您愿意，您将看到通过 use_git() 进行的初始提交：

```
#> # A tibble: 1 x 3
#>   commit                                  author                        message
#>   <chr>                                   <chr>                         <chr>
#> 1 4bdca5b6c90d21b1a25e5ef73fd0a5fa··· Whole Game <whole_game@exam··· "Initial commi···
```

只要您设置了 RStudio + Git 集成，RStudio 可以在任何项目中初始化 Git 存储库，即使它不是 R 包项目。依次点击 *Tools* > *Version Control* > *Project Setup*，然后选择 *Version control system: Git*，然后选择 *initialize a new git repository for this project*。

3.6.6 2.6 编写第一个函数

因子（factors ）的很多操作都令人费解。让我们看看当我们把两个因子（factors ）连接起来的时候会发生什么。

```
(a <- factor(c("character", "hits", "your", "eyeballs")))
#> [1] character hits      your      eyeballs
#> Levels: character eyeballs hits your
(b <- factor(c("but", "integer", "where it", "counts")))
#> [1] but       integer   where it  counts
#> Levels: but counts integer where it
c(a, b)
#> [1] 1 3 4 2 1 3 4 2
```

嗯？许多人并没有预料到，将两个因子（factors）连接起来的结果是一个包含数字 1，2，3，4 的整数向量。如果我们将每个因子（factors ）强制转换为字符，进行分类，然后重新转换为因子（factors ）呢？

```
factor(c(as.character(a), as.character(b)))
#> [1] character hits      your      eyeballs  but       integer   where it
#> [8] counts
#> Levels: but character counts eyeballs hits integer where it your
```

这似乎产生了更有意义的结果。让我们将这个操作放在名为 `fbind()` 的函数体中：

```
fbind <- function(a, b) {
    factor(c(as.character(a), as.character(b)))
}
```

这本书不会教您如何在 R 中编写函数。要了解更多有关该知识的信息，请查阅 *R for Data Science* 和 *Advanced R* 的 Functions Chapter。

3.6.7 2.7 `use_r()`

我们该在哪里定义 `fbind()` 函数？应该将它保存在程序包的 `/R` 子目录中的 `.R` 文件内。一个合理的开始是为程序包中的每个函数创建一个新的 `.R` 文件，并以功能的名字为它命名。当您添加更多的函数后，您可能会希望放宽这个要求，把相关的函数放在一起。我们将 `fbind()` 的函数定义放在 `R/fbind.R` 中。

函数 `use_r()` 将帮助我们在 `R/` 目录下创建和（或）打开脚本文件。当您在 `.R` 文件和关联测试的文件之间互相切换时，它确实在成熟的程序包中十分有用。但是，即使在这里，在 `Untitled4` 中工作时，它也可以避免您因为太投入而忘记了当前工作的目录。 存疑

```
use_r("fbind")
#>   Edit 'R/fbind.R'
#>   Call `use_test()` to create a matching test file
```

将函数 fbind() 的定义并且**仅仅是** fbind() **的定义**放在 R/fbind.R 中并保存。文件 R/fbind.R 不应包含我们最近执行的其他任何顶级代码，例如因子 a 和 b，library(devtools) 或 use_git() 的定义。这预示了在从编写 R 脚本过渡到 R 包时需要进行的调整。程序包和脚本使用不同的机制来声明它们对其他包的依赖，不同之处还有存储示例或测试代码的方式。我们在第 6 章中进一步探讨这一点。

3.6.8 2.8 load_all()

我们应该怎样来测试 fbind() 函数? 如果这是常规的 R 脚本，则可以使用 RStudio 将函数定义发送到 R 控制台，并且在全局工作空间中定义 fbind()。或者，我们可以使用 source("R/fbind.R") 来调用该脚本。但是，在程序包开发中，devtools 提供了更为可靠的方法。有关更多信息，请参见 5.4 节。

调用 load_all() 使得函数 fbind() 可用于测试。

```
load_all()
#> Loading foofactors
```

现在，调用 fbind(a, b) 看看它是怎样工作的。

```
fbind(a, b)
#> [1] character hits      your      eyeballs  but       integer   where it
#> [8] counts
#> Levels: but character counts eyeballs hits integer where it your
```

我们可以注意到，虽然 fbind() 函数并不在全局工作空间中，但是 load_all() 使得我们可以使用该函数。

```
exists("fbind", where = globalenv(), inherits = FALSE)
#> [1] FALSE
```

load_all() 模拟了构建、安装和添加 foofactors 程序包的过程。当您的程序包积累了更多的函数时，有的导出了而有的没有，有的互相调用而有的从依赖的其他程序包中调用，load_all() 相比于在全局工作空间中测试 drive 何解? 函数定义，能够使您对于程序包的开发方式有更为准确的了解。同样的，load_all() 允许比实际中构建、安装和添加程序包快得多的迭代。

到目前为止的内容：

- 我们已经编写了第一个函数 fbind()，它能够将两个因子（factors ）连接起来。
- 我们使用 load_all() 快捷地使得该函数可以用于交互使用，就好像我们已经构建并安装了 foofactors，并通过 library(foofactors) 添加到了工作环境中一样。

RStudio 提供了 `load_all()` 的快速调用。它位于 *Build* 菜单和 *Build* 窗格中，通过 *More* > *Load All* 或者键盘快捷键 Ctrl + Shift + L (WIndows & Linux) 或者 Cmd + Shift + L (MacOS) 调用。

2.8.1 提交 `fbind()` 的更改

如果您使用了 Git，可以使用您喜欢的方法来提交新的 `R/fbind.R` 文件。我们在幕后也是这样做的。以下是提交前后相关的差异。

```
#> diff --git a/R/fbind.R b/R/fbind.R
#> new file mode 100644
#> index 0000000..7b03d75
#> --- /dev/null
#> +++ b/R/fbind.R
#> @@ -0,0 +1,3 @@
#> +fbind <- function(a, b) {
#> +  factor(c(as.character(a), as.character(b)))
#> +}
```

从这一小节之后，我们每一步之后都会进行提交。这些提交在公共储存库中都是可用的。

3.6.9 2.9 `check()`

我们有经验可以证明 `fbind()` 是有效的。但是，我们如何确保 foofactors 包的其他所有可用的功能 moving parts 如何翻译 仍然有效呢？在添加很少的代码之后，仍然进行检查似乎很愚蠢，但养成经常检查的习惯是很好的。

`R CMD check` 会在 shell 中执行，它是检查 R 包是否处于完整工作状态的黄金标准。在不离开 R 会话的情况下，`check()` 是运行这个命令的便捷方法。

请注意，`check()` 将生成相当多的输出，并针对交互式使用进行了优化。我们在这里截取了一部分，并仅仅展示摘要。本地运行 `check()` 的输出将有所不同。

```
check()
```

```
#>   R CMD check results                  foofactors 0.0.0.9000
#> Duration: 9.1s
#>
#>   checking DESCRIPTION meta-information ... WARNING
#>   Non-standard license specification:
#>     `use_mit_license()`, `use_gpl3_license()` or friends to pick a
#>     license
#>   Standardizable: FALSE
#>
#> 0 errors ✓ | 1 warning   | 0 notes ✓
```

请阅读 check 的输出内容！尽可能早并经常性的解决出现的问题。就像在 .R 和 .Rmd 文件上的增量开发一样，您进行全面检查以确保一切正常的间隔时间越长，查明并解决问题的难度也越大。

在这一步中，我们收到了 1 个警告（0 个错误，0 个注释）：

- `Non-standard license specification`

我们将尽快解决它们。

RStudio 提供了 `check()` 的快速调用。它位于 *Build* 菜单和 *Build* 窗格中，通过 *Check* 或者键盘快捷键 Ctrl + Shift + E (WIndows & Linux) 或者 Cmd + Shift + E (MacOS) 调用。

3.6.10 2.10 编辑 DESCRIPTION

在解决有关许可证的警告之前，让我们先处理 DESCRIPTION 中的模板内容。DESCRIPTION 文件提供了有关您的程序包的元数据，我们将在第 7 章中全面介绍它。

在该文件中进行如下编辑; - 在 Author 字段中填上您的名字。- 在 Title 和 Description 字段中填上一些描述性的内容。

在 RStudio 中可以使用 Ctrl + . 并键入 DESCRIPTION 来激活一个帮助程序，这样您可以轻松地打开该文件并编辑。除了可以键入文件名外，还可以是函数名。一旦程序包在 R/ 目录下有许多函数文件时，这将十分方便。

当您完成后，DESCRIPTION 文件应该看起来像这样：

```
Package: foofactors
Title: Make Factors Less Aggravating
Version: 0.0.0.9000
Authors@R:
    person("Jane", "Doe", email = "jane@example.com", role = c("aut", "cre"))
Description: Factors have driven people to extreme measures, like ordering
    custom conference ribbons and laptop stickers to express how HELLNO we
    feel about stringsAsFactors. And yet, sometimes you need them. Can they
    be made less maddening? Let's find out.
License: What license it uses
Encoding: UTF-8
LazyData: true
```

3.6.11 2.11 use_mit_license()

> Pick a License, Any License. – Jeff Atwood

对于 foofactors，我们将使用 MIT 许可证。这需要在 DESCRIPTION 文件中进行规范的描述，并要求一个叫做 LICENSE 的附加文件来声明版权所有者和年份。我们将使用帮助程序 use_mit_license()，参数替换成您的名字。

```
use_mit_license("Jane Doe")
#> ✓ Setting License field in DESCRIPTION to 'MIT + file LICENSE'
#> ✓ Writing 'LICENSE.md'
#> ✓ Adding '^LICENSE\\.md$' to '.Rbuildignore'
#> ✓ Writing 'LICENSE'
```

打开新创建的 LICENSE 文件，确保它具有正确的年份和您的名字。

```
YEAR: 2019
COPYRIGHT HOLDER: Jane Doe
```

和其他创建许可证的函数一样，use_mit_license() 还将完整的许可证副本放入 LICENSE.md 文件，并将该文件添加进 .Rbuildignore 中。最好的做法是使程序包的源代码中包含完整的许可证，就像在 GitHub 中一样，但是 CRAN 禁止在程序包源代码中包含此文件。

3.6.12 2.12 document()

就像其他 R 函数那样，在使用 fbind() 时能够获得帮助文档，这不是很好吗？这要求程序包具有特殊的 R 文档文件，man/fbind.Rd，一个以类似于 LaTeX 的 R 的特殊标记语言编写的文档。幸运的是，我们不一定需要直接编辑它。

我们在源代码文件中的 fbind() 函数体上直接编写一个特别格式的注释，然后让一个名为 roxygen2 的包来完成 man/fbind.Rd 的创建。roxygen2 的使用和机制将在第 8 章中介绍。

如果您使用 RStudio，则在源代码编辑器中打开 R/fbind.R，并将光标放在 fbind() 函数定义中的某处。现在点击 *Code > Insert roxygen skeleton*。函数上方应该会出现一个非常特殊的注释模板，每行以 #' 开头。RStudio 只插入模板框架，因此您需要对其进行编辑，如下所示。

如果您不使用 RStudio，请自己创建注释。无论如何，您应该修改它，让它看起来像下面那样：

```
#' Bind two factors
#'
#' Create a new factor from two existing factors, where the new factor's levels
#' are the union of the levels of the input factors.
#'
#' @param a factor
#' @param b factor
#'
#' @return factor
#' @export
#' @examples
#' fbind(iris$Species[c(1, 51, 101)], PlantGrowth$group[c(1, 11, 21)])
```

TODO: mention how RStudio helps you execute examples here?

但是我们还没有完成！我们还需要用 document() 将这个新的 roxygen 注释转变为 man/fbind.Rd：

```
document()
#> Updating foofactors documentation
#> Updating roxygen version in /private/var/folders/24/8k48jl6d249_n_qfxwsl6xvm0000gn/T/
↪RtmpKNHtqz/foofactors/DESCRIPTION
#> Loading foofactors
#> Writing NAMESPACE
#> Writing fbind.Rd
```

RStudio 提供了 `document()` 的快速调用。它位于 *Build* 菜单和 *Build* 窗格中，通过 *More* > *Document* 或者键盘快捷键 Ctrl + Shift + D (WIndows & Linux) 或者 Cmd + Shift + D (MacOS) 调用。

您现在应该已经可以预览帮助文档，如下所示：

```
?fbind
```

您将看到一条消息，如 “Rendering development documentation for ‘fbind’”，它提醒您，您基本上是在预览文档草稿。也就是说，该文档存在于程序包的源代码中，但尚未存在于已安装的包中。事实上，我们还没有安装 foofactors，但我们很快就会安装它。

另请注意，在正式构建和安装包之前，您的程序包的文档不会被正确连接。这样可以改善一些细微之处，例如帮助文档之间的链接和程序包索引的创建。暂时不能理解

2.12.1 NAMESPACE 的更改

除了将 `fbind()` 函数的特殊注释转变为 `man/fbind.Rd` 文档，调用 `document()` 还能基于 roxygen 注释中的 `@export` 指令更新 `NAMESPACE`。您可以检查 `NAMESPACE` 文件，里面的内容应为：

```
# Generated by roxygen2: do not edit by hand

export(fbind)
```

`NAMESPACE` 中的 export 指令是通过 `library(foofactors)` 添加 foofactors 库后，`fbind()` 函数对于用户可用的原因。就像可以 “亲手” 编写 `.Rd` 文件一样，您可以自己显式地管理 `NAMESPACE` 文件。但是我们选择将其委托给 devtools（以及 roxygen2）。

3.6.13 2.13 再次 `check()`

foofactors 现在应该可以立刻并永远通过 `R CMD check` 了：0 个错误，0 个警告，0 个注释。

```
check()
```

```
#>   R CMD check results                  foofactors 0.0.0.9000
#> Duration: 12.8s
```

(下页继续)

(续上页)

```
#>
#> 0 errors ✓ | 0 warnings ✓ | 0 notes ✓
```

3.6.14 2.14 install()

由于我们现在已经有了一个最小的完整可行的产品，因此可以通过 install() 将 foofactors 包安装到您的库中：

```
install()
```

```
   checking for file ‘/private/var/folders/24/8k48jl6d249_n_qfxwsl6xvm0000gn/T/
↪RtmpKNHtqz/foofactors/DESCRIPTION’ ...
✓  checking for file ‘/private/var/folders/24/8k48jl6d249_n_qfxwsl6xvm0000gn/T/
↪RtmpKNHtqz/foofactors/DESCRIPTION’
   preparing ‘foofactors’:
checking DESCRIPTION meta-information ...
✓  checking DESCRIPTION meta-information
   checking for LF line-endings in source and make files and shell scripts
   checking for empty or unneeded directories
   building ‘foofactors_0.0.0.9000.tar.gz’
Running /Library/Frameworks/R.framework/Resources/bin/R CMD INSTALL \
/var/folders/24/8k48jl6d249_n_qfxwsl6xvm0000gn/T//RtmpKNHtqz/foofactors_0.0.0.9000.tar.
↪gz \
--install-tests
* installing to library ‘/Users/runner/work/_temp/Library’
* installing *source* package ‘foofactors’ ...
** using staged installation
** R
** byte-compile and prepare package for lazy loading
** help
*** installing help indices
** building package indices
** testing if installed package can be loaded from temporary location
** testing if installed package can be loaded from final location
** testing if installed package keeps a record of temporary installation path
* DONE (foofactors)
```

RStudio 在 *Build* 菜单和 *Build* 窗格提供了类似的功能，通过 *Install and Restart* 调用。

现在，我们可以像其他任何程序包一样添加并使用 foofactors 了。让我们从头回顾一下我们的小例子。这是重新启动 R 会话（R Session）并清理工作区的好时机。

```
library(foofactors)

a <- factor(c("character", "hits", "your", "eyeballs"))
b <- factor(c("but", "integer", "where it", "counts"))

fbind(a, b)
#> [1] character hits      your      eyeballs  but       integer   where it
#> [8] counts
#> Levels: but character counts eyeballs hits integer where it your
```

我们成功了！

3.6.15 2.15 use_testthat()

在一个示例中，我们已经对 `fbind()` 进行了简单测试，我们可以通过一些单元测试（unit test）来形式化和扩展它。这意味着我们对于 `fbind()` 在各种输入数据下的正确结果得有一些具体的期望。

首先，我们声明我们将使用 testthat 包中的 `use_testthat()` 来编写单元测试：

```
use_testthat()
#> ✓ Adding 'testthat' to Suggests field in DESCRIPTION
#> ✓ Creating 'tests/testthat/'
#> ✓ Writing 'tests/testthat.R'
#>   Call `use_test()` to initialize a basic test file and open it for editing.
```

这将为您的程序包初始化单元测试机器。它在 `DESCRIPTION` 中添加了 `Suggests: testthat` ，创建了目录 `test/testthat` 斌添加了脚本 `test/testthat.R` 。

然而，实际的测试仍然是由**您**来编写！

函数 `use_test()` 打开并/或创建测试文件。您可以提供文件名，或者，如果您在 RStudio 中编辑相关的源文件，文件名将自动生成。由于本书是非交互构建的，我们必须显式地提供文件名：

```
use_test("fbind")
#> ✓ Increasing 'testthat' version to '>= 2.1.0' in DESCRIPTION
#> ✓ Writing 'tests/testthat/test-fbind.R'
```

它将会生成文件 `tests/testthat/test-fbind.R` 。将以下内容编写入该文件：

```
test_that("fbind() binds factor (or character)", {
    x <- c("a", "b")
    x_fact <- factor(x)
    y <- c("c", "d")
    z <- factor(c("a", "b", "c", "d"))

    expect_identical(fbind(x, y), z)
    expect_identical(fbind(x_fact, y), z)
})
```

这将测试 `fbind()` 在组合两个因子（factors）、一个字符向量（character vector）和一个因子（factor）时是否会给出预期结果。

以交互方式运行此测试，就像编写自己的测试一样。注意，您必须首先在 R 会话（R Session）中通过 `lirary(testthat)` 添加 testthat，并且您可能需要 `load(all)`。

接下来，您的测试将主要（*en masse*）通过 `test()` 集中（arms' s length）运行：

TODO: work on the aesthetics of this output. Or maybe testthat 3e will save me the trouble.

```
test()
#> ✓ |  OK F W S | Context
#>
  |   0       | fbind
✓ |   2       | fbind
#>
#>   Results
#> [ PASS x2 FAIL x0 WARN x0 SKIP x0 ]
```

RStudio 提供了 `test()` 的快速调用。它位于在 *Build* 菜单和 *Build* 窗格中，通过 *More* > *Test package*, 或者键盘快捷键 Ctrl + Shift + T (WIndows & Linux) 或者 Cmd + Shift + T (MacOS)

调用。

每当您使用 `check()` 检查程序包时，您的测试也会运行。这样，您基本上就可以使用自己的一些特定于您的包的检查代码来增强标准检查。使用 cove package 跟踪该测试所执行的源代码的比例也是一个好主意。更多细节见第 10 章。

3.6.16 2.16 use_package()

您将不可避免地在自己的程序包中使用其他程序包中的函数。就像我们需要**导出**函数 `fbind()` 一样，我们需要从其他程序包的命名空间内**导入**函数。如果您打算将您的程序包提交到 CRAN，请注意，这甚至适用于那些您认为“始终可用”的程序包，例如 `stats::median()` 或者是 `utils::head()`。

我们将向 foofactors 包中添加另一个函数，该函数将会为因子（factors）生成一个排序的频率表。我们将从 forcats 包中借用一些技巧，特别是函数 `forcats::fct_count()`。

首先，使用 `use_package()` 声明我们将使用 forcats 命名空间中的某些函数：

```
use_package("forcats")
#> ✓ Adding 'forcats' to Imports field in DESCRIPTION
#>   Refer to functions with `forcats::fun()`
```

这会将 forcats 包添加进 `DESCRIPTION` 中的“Imports”部分，仅此而已。

现在，我们向 foofactors 中添加第二个函数：假设我们想要一个因子（factors ）的频率表，它的形式是一个具有漂亮变量名的常规数据框（data frame），而不是作为 `table` 类的一个对象或者是有奇怪名字的一些东西。我们还对它进行排序，使得最受欢迎的层级位于顶部。

使用 `use_r()` 在 `R/` 下初始化一个新的 ``.R 文件：

```
use_r("fcount")
#>   Edit 'R/fcount.R'
#>   Call `use_test()` to create a matching test file
```

将以下内容放入 `R/fcount.R`：

```
#' Make a sorted frequency table for a factor
#'
#' @param x factor
#'
#' @return A tibble
#' @export
#' @examples
#' fcount(iris$Species)
fcount <- function(x) {
```

（下页继续）

(续上页)

```
    forcats::fct_count(x, sort = TRUE)
}
```

请注意我们是如何使用 `forcats::` 作为对 forcats 中函数的调用。这样我们指定了要从 forcats 命名空间内调用 `fct_count()` 函数。从其他程序包中调用函数的方法不止一种，我们在这里使用的方法在第 11 章中有详细的说明。

通过使用 `load_all()` 模拟程序包的安装来尝试新的 `fcount()` 函数。

```
load_all()
#> Loading foofactors
fcount(iris$Species)
#> # A tibble: 3 x 2
#>   f              n
#>   <fct>      <int>
#> 1 setosa        50
#> 2 versicolor    50
#> 3 virginica     50
```

通过 `document()` 来生成相关联的帮助文档。这一过程也会将 `fcount()` 作为一个导出以供使用的函数添加进 `NAMESPACE`。

```
document()
#> Updating foofactors documentation
#> Writing NAMESPACE
#> Loading foofactors
#> Writing NAMESPACE
#> Writing fcount.Rd
```

3.6.17 2.17 use_github()

您已经看到我们在 foofactors 的开发过程中进行了许多提交。您可以在 https://github.com/jennybc/foofacts 上看到指示性的历史记录。我们使用版本控制系统并公开开发过程的决定意味着您可以在每个开发阶段检查 foofactors 源代码的状态。通过查看所谓的 diff，您可以确切地看到每个 devtools 帮助函数是如何修改构成 foofactors 程序包的源文件。

如何将本地 foofactors 程序包和 Git 存储库连接到 GitHub 上的配套存储库呢?

- `use_github()` 是我们一直以来推荐的帮助函数。我们不会在这里演示，因为它需要在您的主机端进行一些特殊的设置。我们也不想在每次建立这本书的时候都删除和重建公共 foofactors 程序包。
- 先设置 GitHub repo！这听起来有悖常理，但让您的工作进入 GitHub 的最简单方法是在那里启动，然后使用 RStudio 在同步的本地副本中开始工作。这种方法在 Happy Git 的工作流 New project, GitHub

first 和 Existing project，GitHub first 中描述。

- 命令行 Git（Command line Git）始终可以用于添加远程存储库 *post hoc*。这在 Happy Git 的工作流 ‘ Existing project, GitHub last <https://happygitwithr.com/existing-github-last.html>‘___ 中进行了描述。

这些方法中的任何一种都会将本地 foofactors 项目连接到 GitHub repo（公共或私有），您可以使用 RStudio 中内置的 Git 客户端来推送或拉取它。

3.6.18 2.18 use_readme_rmd()

既然您的程序包位于 GitHub 上，那么 README.md 文件就很重要。它是程序包的主页和欢迎界面，至少在你决定为它建立一个网站（见 pkgdown）、添加一个 vignette（见第 9 章）或提交给 CRAN 之前（见第 18 章）。

use_readme_rmd() 函数的作用是初始化一个基本的、可执行的 README.Rmd，以便您编辑：

```
use_readme_rmd()
#> ✓ Writing 'README.Rmd'
#> ✓ Adding '^README\\.Rmd$' to '.Rbuildignore'
#> ✓ Writing '.git/hooks/pre-commit'
```

除了创建 README.Rmd 之外，它还会在 .Rbuildignore 中添加一些行，并创建一个 Git 预提交 Hook，以帮助您保持 README.Rmd 和 README.md 的同步。

README.Rmd 已经有了如下部分:

- 提示您描述程序包的用途。
- 提供安装程序包的代码。
- 提示您展示一些用法。

如何填充这个模板骨架？从 DESCRIPTION 和任何正式或非正式的测试或例子中大量地复制内容。有内容总比什么都没有好。否则……您希望人们自己安装您的程序包，仔细检查各个帮助文件，并找出如何使用它吗？他们可能并不会这样做。

我们喜欢用 R Markdown 来编写 README，因此它可以展示实际的用法。包含实时代码还可以减少 README 变得过时和与实际的程序包不同步的可能性。

如果 RStudio 还没有这样做，请打开 README.Rmd 进行编辑。例如，确保它显示了 fbind() 和/或 fcount() 的一些用法。

我们使用的 README.Rmd 在这里：README.Rmd ，以下是内容：

TODO: update this link after merge into r-pkgs.

```
---
output: github_document
---
```

(下页继续)

(续上页)

```
<!-- README.md is generated from README.Rmd. Please edit that file -->

```{r, include = FALSE}
knitr::opts_chunk$set(
collapse = TRUE,
comment = "#>",
fig.path = "man/figures/README-",
out.width = "100%"
)
```

**NOTE: This is a toy package created for expository purposes, for the second edition of␣
↪[R Packages](https://r-pkgs.org). It is not meant to actually be useful. If you want a␣
↪package for factor handling, please see [forcats](https://forcats.tidyverse.org).**

foofactors

<!-- badges: start -->
<!-- badges: end -->

Factors are a very useful type of variable in R, but they can also be very aggravating.␣
↪This package provides some helper functions for the care and feeding of factors.

Installation

You can install foofactors like so:

``` r
devtools::install_github("jennybc/foofactors")
```

Quick demo

Binding two factors via `fbind()`:

```{r}
library(foofactors)
a <- factor(c("character", "hits", "your", "eyeballs"))
b <- factor(c("but", "integer", "where it", "counts"))
```
```

(下页继续)

(续上页)

```
```

Simply catenating two factors leads to a result that most don't expect.

```{r}
c(a, b)
```

The `fbind()` function glues two factors together and returns factor.

```{r}
fbind(a, b)
```

Often we want a table of frequencies for the levels of a factor. The base `table()` function returns an object of class `table`, which can be inconvenient for downstream work.

```{r}
set.seed(1234)
x <- factor(sample(letters[1:5], size = 100, replace = TRUE))
table(x)
```

The `fcount()` function returns a frequency table as a tibble with a column of factor levels and another of frequencies:

```{r}
fcount(x)
```
```

别忘了渲染它并产生 README.md！如果您尝试提交 README.Rmd 而不是 "README.md"，并且 README.md 似乎已过期，那么预提交 Hook 应该会提醒您。

渲染 README.Rmd 的最好方式是和 build_readme() 一起使用，因为它会注意使用程序包的最新版本进行渲染，即从当前源安装一个临时副本。

```
build_readme()
#> Installing foofactors in temporary library
#> Building /private/var/folders/24/8k48jl6d249_n_qfxwsl6xvm0000gn/T/RtmpKNHtqz/
↪foofactors/README.Rmd
```
```

您只需要访问 GitHub 上的 foofactors 就可以看到已经渲染的 README.md。

最后，别忘了做最后一次提交。如果您用的是 GitHub，还需要推送至远程仓库。

3.6.19 2.19 最后一步: check() 以及 install()

让我们再次运行 check() 以确保程序包一切正常。

```
check()
```

```
#>   R CMD check results                  foofactors 0.0.0.9000
#> Duration: 13.4s
#>
#> 0 errors ✓ | 0 warnings ✓ | 0 notes ✓
```

foofactors 应该没有错误、警告或注释信息。现在是重新构建和安装它的最好时机。庆祝一下！

```
install()
```

```
   checking for file ‘/private/var/folders/24/8k48jl6d249_n_qfxwsl6xvm0000gn/T/
↪RtmpKNHtqz/foofactors/DESCRIPTION’ ...
✓  checking for file ‘/private/var/folders/24/8k48jl6d249_n_qfxwsl6xvm0000gn/T/
↪RtmpKNHtqz/foofactors/DESCRIPTION’
  preparing ‘foofactors’:
checking DESCRIPTION meta-information ...
✓  checking DESCRIPTION meta-information
   checking for LF line-endings in source and make files and shell scripts
   checking for empty or unneeded directories
Removed empty directory ‘foofactors/tests/testthat/_snaps’
   building ‘foofactors_0.0.0.9000.tar.gz’
Running /Library/Frameworks/R.framework/Resources/bin/R CMD INSTALL \
/var/folders/24/8k48jl6d249_n_qfxwsl6xvm0000gn/T//RtmpKNHtqz/foofactors_0.0.0.9000.tar.
↪gz \
--install-tests
* installing to library ‘/Users/runner/work/_temp/Library’
* installing *source* package ‘foofactors’ ...
** using staged installation
** R
** tests
** byte-compile and prepare package for lazy loading
** help
*** installing help indices
```

(下页继续)

(续上页)

```
** building package indices
** testing if installed package can be loaded from temporary location
** testing if installed package can be loaded from final location
** testing if installed package keeps a record of temporary installation path
* DONE (foofactors)
```

请随意访问 GitHub 上的 foofactors package，它正是在这里开发的。提交历史记录反映了每个单独的步骤，因此可以使用 diff 来查看随着包的发展，哪些文件被添加和修改。本书的其余部分将更详细地介绍您在这里看到的每一个步骤以及更多内容。

3.7 第三章系统设置

3.7.1 3.1 准备你的系统

在开始之前，请确保您已经安装了最新版本的 R（至少需要 4.0.2 版本，本书也是使用该版本渲染生成的），然后再运行以下代码来获取您将使用到的程序包：

```
install.packages(c("devtools", "roxygen2", "testthat", "knitr"))
```

请确保您已经安装了最新版本的 RStudio 集成开发环境（IDE）。事实上，您可以考虑使用预览版本并定期升级。与正式发布的版本相比，预览版能够让您体验到最新、最强大的功能，并且仅会稍微增加遇见 Bug 的机会。它不同于更不稳定的每日构建的版本。

- 预览版本（Preview version）：https://www.rstudio.com/products/rstudio/download/preview/
- 正式版本（Released version）：https://www.rstudio.com/products/rstudio/download/
  - 大部分读者可以使用 RStudio Desktop 的免费、开源版本。

3.7.2 3.2 devtools, usethis, 还有您

> "I am large, I contain multitudes."
>
> —Walt Whitman, Song of Myself

经过 7 年的发展，devtools 已经成长为一个相当笨拙的程序包，这让它的维护变得很困难。2.0.0 版本在 2018 年年末发布，它标志着 devtools 的 有意识地解耦 。devtools 的大多数功能都转移到 7 个较小的程序中。尽管 devtools 主要在其他地方维护，但是通过各种方法，devtools 将继续提供它的所有常用功能。例如，devtools 可能会提供一个封装函数（wrapper function），以便设置用户友好的默认值、引入有用的交互行为或者组合多个子程序包中的功能。

我们推荐的关于使用 devtools 及组成其的程序包的方法会有所不同，这具体取决于您是在 useR 还是 developR 模式下工作：

- 对于交互式使用，useRs 应引入 devtools 程序包，并将其视为您最喜欢的程序包开发功能的提供者。
- 对于编程使用，例如在另一个包内，developRs **不应该**依赖于 devtools，而应通过作为函数主目录的程序包来访问函数。
  - devtools 应该尽量避免以 `foo::fcn()` 的形式出现在 `foo` 程序包的合法调用中。相反，应该是在 `foo` 中定义 `fcn()` 函数。
  - 上述的一个例外情况是，即使 `install_github()` 函数实际上位于远程的 即作为在线资源 程序包中，我们仍将以 `devtools::install_github()` 作为在 README 中描述的安装程序包开发版本的方式。这是因为这一条建议与交互式使用有关，我们更喜欢强调 devtools。
- 尝试报告作为函数所在主目录的程序包中的 Bug。

以下示例展示了在交互式开发过程中如何模拟安装和加载程序包：

```
library(devtools)
load_all()
```

如果在 R 程序包中使用相同的功能，则以下是首选的调用方式：

```
pkgload::load_all()
```

usethis 程序包是一个更多人可能知道并且能够直接使用的组成程序包。现在，它包含对 R 项目中的文件和文件夹执行操作的函数，尤其是对于同时也是 R 程序包的任何项目而言。devtools 提供了所有 usethis 中的功能。因此，一旦添加了 devtools，就可以无限制地使用 usethis 中的任何函数，即只需调用 `use_testthat()`。如果您指定了命名空间（例如，在以更具编程风格的方式工作时），则直接访问 usethis 中的函数：使用 `usethis::use_testthat()` 而不是 `devtools::use_testthat()`。

3.2.1 个人启动配置

您可以通过以下方式添加 devtools 程序包：

```
library(devtools)
```

但是随着在每个 R 会话（R Session）中反复添加 devtools，它就变得令人烦恼了。因此，我们强烈建议将 devtools 附加到您的 `.Rprofile` 启动文件中，如下所示：

```
if (interactive()) {
  suppressMessages(require(devtools))
}
```

为了方便起见，`use_devtools()` 函数会在您需要时创建 `.Rprofile` 文件，将其打开并进行编辑，然后再剪切板和屏幕上放置必要的代码行。您可能想要用这种方式处理的另一个程序包是 testthat。

通常来说，在 .Rprofile 中添加程序包是一个坏主意，因为它邀请您通过显式调用 library(foo) 创建不反映所有依赖关系的 R 脚本。但是 devtools 是一个工作流程包，它简化了程序包的开发流程，因此不太可能融入到任何分析脚本中。请注意，我们仍然注意只在交互式会话中添加。

例如，在从头（*de novo*）创建 R 包时，usethis 参考了某些选项。这允许您指定程序包维护者或首选许可证等个人默认设置。下面是 .Rprofile 中可能包含的代码片段的示例：

```
options(
  usethis.full_name = "Jane Doe",
  usethis.description = list(
    `Authors@R` = 'person("Jane", "Doe", email = "jane@example.com", role = c("aut", "cre
↪"),
    comment = c(ORCID = "YOUR-ORCID-ID"))',
    License = "MIT + file LICENSE",
    Version = "0.0.0.9000"
  ),
  usethis.protocol  = "ssh"
)
```

以下代码安装了 devtools 和 usethis 的开发版本，这在本书的修订过程中可能很重要。

```
devtools::install_github("r-lib/devtools")
devtools::install_github("r-lib/usethis")
```

3.7.3 3.3 R 构建工具链

要能够完全从源代码构建 R 程序包，还需要一个编译器和其他一些命令行工具。这可能不是必须的，除非您想构建包含 C 或 C++ 代码的程序包（第 13 章的主题）。特别是如果您正在使用 RStudio，您可以暂时把它放在一边。一旦您尝试执行需要您设置开发环境的操作，IDE 将向您发出警报并提供支持。请继续阅读以获取有关自己操作的建议。

3.3.1 Windows

在 Windows 上，从源代码构建程序包所需要的工具集叫做 RTools。

RTools **不是** R 程序包，它不能通过 instaaa.packages() 安装，而是通过从 https://cran.r-project.org/bin/windows/Rtools/ 下载并运行安装程序来安装。

在 RTools 的安装过程中，您可能会看到一个窗口，它询问您 “Select Additional Tasks”。

- **不要**选中“Edit the system PATH”。devtools 和 Rstudio 应该在需要时自动将 RTools 放入 `PATH`。
- 选中“Save version information to registry”。它应该是默认选中的。

3.3.2 macOS

您需要安装 Xcode 命令行工具，这需要您注册成为 Apple 开发人员（不要担心，它是免费的）。

然后，在 shell 中执行如下操作：

```
xcode-select --install
```

或者，您可以从 Mac App Store 安装最新版本的完整的 Xcode。这会包含许多您不需要的东西，但是它有 App Store 便利性的优势。

3.3.3 Linux

请确保您不仅已经安装了 R，还安装了 R 开发工具。例如，在 Ubuntu（以及 Debian）上，您需要安装 `r-base-dev` 程序包。

3.3.4 验证系统准备工作

通过运行以下代码，您可以检查是否已安装所有组件并正常工作：

```
# TODO: replace with whatever results from https://github.com/r-lib/devtools/issues/1970
library(devtools)
has_devel()
#> [1] TRUE
```

如果一切正常，它将返回 `TRUE`。否则，它将显示有关该问题的一切诊断信息。

3.8 第四章程序包结构与状态

本章将通过把您从 *使用 R 包*中获得的隐性知识转换为 *创建和修改*它们所需的显式知识，从而开始开发程序包。您将了解程序包的各种状态，以及它和库（library）之间的区别（以及为什么要关心二者的区别）。

3.8.1 4.1 程序包状态

创建或修改程序包时，需要在它的“源代码”或“源文件”上进行。您能够以**源代码**的形式与正在开发的程序包进行交互。当然，这**并不是**你日常使用中最熟悉的程序包的形式。如果您了解 R 程序包可能处于的五种状态，那么程序包开发的工作流将变得更有意义：

- 源代码（source）

- 捆绑的（bundled）
- 二进制文件（binary）
- 已安装的（installed）
- 载入内存中的（in-memory）

您已经知道一些将程序包转入这些状态的函数。例如，`install.packages()` 和 `devtools::install_github()` 将程序包从源代码（source）、已绑定的（bundled）或二进制文件（binary）状态转移到已安装（installed）状态。`library()` 函数的作用是：将已安装的程序包加载到内存中，以便可以马上直接使用。

3.8.2 4.2 源码包（Source Package）

一个**源代码**程序包就是一个有着特定结构的文件目录。它包含特定的组件，例如一个 `DESCRIPTION` 文件、包含 `.R` 文件的 `R/` 目录等。本书余下的大部分章节都致力于详细说明这些组成部分。

如果您刚刚接触 R 程序包的开发，那么您可能从未见过源代码形式的程序包！您的计算机上可能甚至没有任何源程序包。以源代码形式查看程序包的最简单的方法是在 web 上浏览其代码。

许多 R 程序包是在 GitHub（或者 GitLab 以及类似的平台）上的公开库（open）中开发的。最好的方案是访问程序包的 CRAN 主页（landing page），例如：

- forcats: https://cran.r-project.org/package=forcats
- readxl: https://cran.r-project.org/package=readxl

并且其中一个网页链接（URLs）链接到公共托管服务上的存储库，例如：

- forcats: https://github.com/tidyverse/forcats
- readxl: https://github.com/tidyverse/readxl

即使程序包是在公共存储库中开发的，一些维护人员还是忘记了列出这个网页链接（URL），但是您仍然可以通过搜索来发现它。

即使程序包不是在公共平台上开发的，也可以在 METACRAN 维护的非官方只读镜像中访问其源代码。示例：

- MASS: https://github.com/cran/MASS
- car: https://github.com/cran/car

请注意，这与探索程序包真正的开发环境不同，因为这里的源代码及其演变过程只是对程序包的 CRAN 发行版本进行逆向工程的结果。它提供了对程序包及其开发历史的审查视图，但根据定义，源代码及其历史包含了所有程序包开发的必需的内容。意译，有待审查

3.8.3 4.3 捆绑包（Bundled package）

捆绑的程序包是被压缩成单个文件的程序包。按照惯例（该惯例来自 Linux），R 中的捆绑程序包使用 `.tar.gz` 扩展名，并且有时被称为“源码压缩包”。这意味着多个文件已经被打包为一个文件（`.tar`）并使用 gzip（`.gz`）进行压缩。虽然捆绑程序包本身并不那么有用，但它是源码包和已安装包之间平台无关的、便于传输的中间媒介。

在从本地开发的程序包中生成捆绑程序包这种罕见的情况下，请使用 `devtools::build()`。在幕后，它会调用 `pkgbuild::build()` 并最终调用 `R CMD build`，这些会在 Writing R Extensions 的 Building package tarballs 部分中进一步阐述。

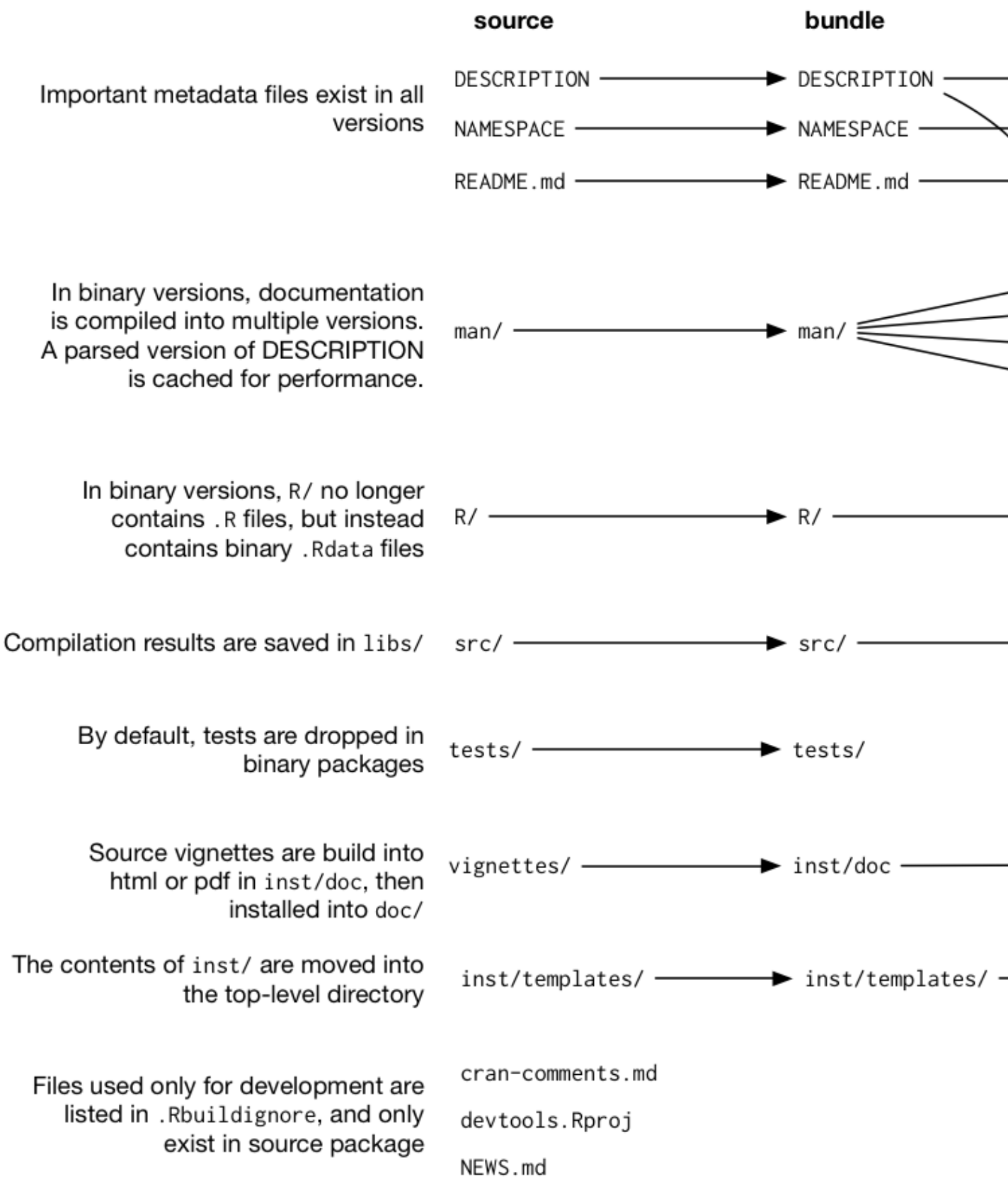
这应该会提醒您，捆绑程序包或“源码压缩包”（source tarball）不仅仅是对源文件进行 tar 打包存档，然后使用 gzip 压缩的结果。按照惯例，在 R 世界中，在制作 `.tar.gz` 文件时还要执行一些操作。**这就是我们选择将其称为捆绑程序包的原因**。理解“捆绑”一词的重要语段

每一个 CRAN 程序包都以捆绑程序包的形式提供，可以通过个人登录界面（individual landing pages）的”Packages Source” 字段内容获取。继续我们上面的示例，您可以下载 `forcats_0.4.0.tar.gz` 和 `readxl_1.3.1.tar.gz` 的捆绑包。（或者任何当前的版本）。您可以在 shell（而不是 R 控制台）中进行解压缩：

```
tar xvf forcats_0.4.0.tar.gz
```

如果您解压缩一个捆绑包，您将看到它看起来几乎与源码包相同。下图总结了 devtools 的源码版本、捆绑版本和二进制版本的顶级目录中出现的文件。

TODO: Remake this figure https://github.com/hadley/r-pkgs/issues/587.


source
bundle
Important metadata files exist in all versions
DESCRIPTION
NAMESPACE
README.md
In binary versions, documentation is compiled into multiple versions. A parsed version of DESCRIPTION is cached for performance.
man/
In binary versions, R/ no longer contains .R files, but instead contains binary .Rdata files
R/
Compilation results are saved in libs/
src/
By default, tests are dropped in binary packages
tests/
Source vignettes are build into html or pdf in inst/doc, then installed into doc/
vignettes/
inst/doc
The contents of inst/ are moved into the top-level directory
inst/templates/
Files used only for development are listed in .Rbuildignore, and only exist in source package
cran-comments.md
devtools.Rproj
NEWS.md

源码包和未压缩的捆绑包之间的主要区别为：

- 已经生成了 Vignettes，因此以已渲染的输出（如 HTML）出现在 `inst/doc/` 目录下，并且 Vignette 索引出现在 `build/` 目录中，通常还有一个 PDF 的程序包手册。
- 本地源码包可能包含用于在开发期间节省时间的临时文件，如 `src/` 中的编译文件。这些文件从来没有在捆绑包中找到过。
- `.Rbuildignore` 中列出的任何文件都不包含在捆绑包中。这些文件通常有助于您的开发过程，但应该从分发式 原意“分布式” 的产品中排除。

4.3.1 .Rbuildignore

您不需要非常频繁地考虑 `.tar.gz` 文件形式的程序包的确切结构，但您确实需要了解 `.Rbuildignore` 文件。它决定了源码包中的哪些文件可以进入后面的工作流。

`.Rbuildignore` 的每一行都是与 Perl 兼容的正则表达式，它与源码包中每个文件的路径匹配，而不考虑大小写。[1] 如果与正则表达式匹配，则排除该文件或目录。注意，有一些默认排除项由 R 本身执行，主要与经典的版本控制系统和编辑器（如 SVN、Git 和 Emacs）有关。

要排除特定的文件或目录（最常见的使用示例），您**必须**锚定（anchor）正则表达式。例如，要排除名为”notes”的目录，请使用 `^notes$`。正则表达式 `notes` 将匹配任何包含 `notes` 的文件名，例如 `R/notes.R`、`man/important-notes.R`、`data/endnotes.Rdata` 等。排除特定文件或目录的最安全方法是使用 `usethis::use_build_ignore("notes")`，它将为您执行转义。

`.Rbuildignore` 是解决让您更便利地开发的操作 意译 与 CRAN 提交和分发的要求之间一些紧张关系的一种方法。即使您不打算在 CRAN 上发布，遵循这些约定能让您最好地使用 R 的内置工具来检查和安装程序包。受影响的文件分为两个半重叠的类别：

- 帮助您以编程方式生成程序包内容的文件。例如：
  - 使用 `README.Rmd` 生成信息和当前的 `README.md`。
  - 存储 `.R` 脚本以创建和更新内部的或导出的数据。
- 驱动程序包开发、检查和产生文档的文件，不在 CRAN 的范围内。例如：
  - 与 RStudio IDE 相关的文件
  - 使用 pkddown package 生成的网站。
  - 与持续集成/部署和监视测试覆盖范围相关的配置文件。

以下是 tidyverse 中程序包的 `.Rbuildignore` 文件中典型条目的非完整列表：

```
^.*\.Rproj$          # Designates the directory as an RStudio Project
^\.Rproj\.user$      # Used by RStudio for temporary files
^README\.Rmd$        # An Rmd file used to generate README.md
```

(下页继续)

[1] 要查看应该出现在您的雷达上的文件路径，请在程序包的顶级目录下执行 `dir(full.names = TRUE, recursive = TRUE, include.dirs = TRUE, all.files = TRUE)`。

(续上页)

```
^LICENSE\.md$        # Full text of the license
^cran-comments\.md$ # Comments for CRAN submission
^\.travis\.yml$      # Used by Travis-CI for continuous integration testing
^data-raw$           # Code used to create data included in the package
^pkgdown$            # Resources used for the package website
^_pkgdown\.yml$      # Configuration info for the package website
^\.github$           # Contributing guidelines, CoC, issue templates, etc.
```

请注意，上面的注释不能出现在实际的 `.Rbuildignore` 文件中。此处包含这些注释只是为了演示。

我们会在需要的时候提到何时需要向 `.Rbuildignore` 中添加排除项 意译。请记住，`usethis::use_build_ignore()` 是管理此文件的一种有吸引力的方法。

3.8.4 4.4 二进制包（Binary package）

如果要将程序包分发给没有程序包开发工具的 R 用户，则需要提供二进制包。与捆绑包一样 意译，二进制包是单个文件。但是与捆绑包不同，二进制包是平台相关的，有两种基本类型：Windows 和 macOS。(Linux 用户通常需要具备从 `.tar.gz` 文件安装程序包所需要的工具。)

macOS 平台上的二进制包储存为以 `.tgz` 为后缀的文件，而 Windows 平台的二进制包则以 `.zip` 为文件后缀。如果你需要制作一个二进制包，则需要在相关的平台上使用 `devtools::build(binary = TRUE)`。在幕后，该函数调用 `pkgbuild::build(binary= TRUE)` 并且最终调用 `R CMD INSTALL --build`。这些会在 Writing R Extensions 的 Building binary packages 部分作进一步阐述。

需要明确的是，二进制包的主要制作者和分发者是 CRAN，而不是个人维护者。如果您的程序包是供公众使用的，那么使其广泛可用的最高效的方法是在 CRAN 上发布它。您提交捆绑包，然后 CRAN 将制作并分发二进制包。

不论是 macOS 或 Windows，还是 R 的当前、先前和（可能的）开发版本，CRAN 通常都能以二进制包形式提供。继续我们上面的例子，您能够下载二进制包，例如：

- forcats for macOS: `forcats_0.4.0.tgz`
- readxl for Windows: `readxl_1.3.1.zip`

事实上，这是您在调用 `isntall.packages()` 时通常进行的部分幕后操作。

如果解压缩二进制包，您将看到它的内部结构与源码包或捆绑包有很大不同。图 4.1 包含了二者的比较。以下是一些最显著的区别：

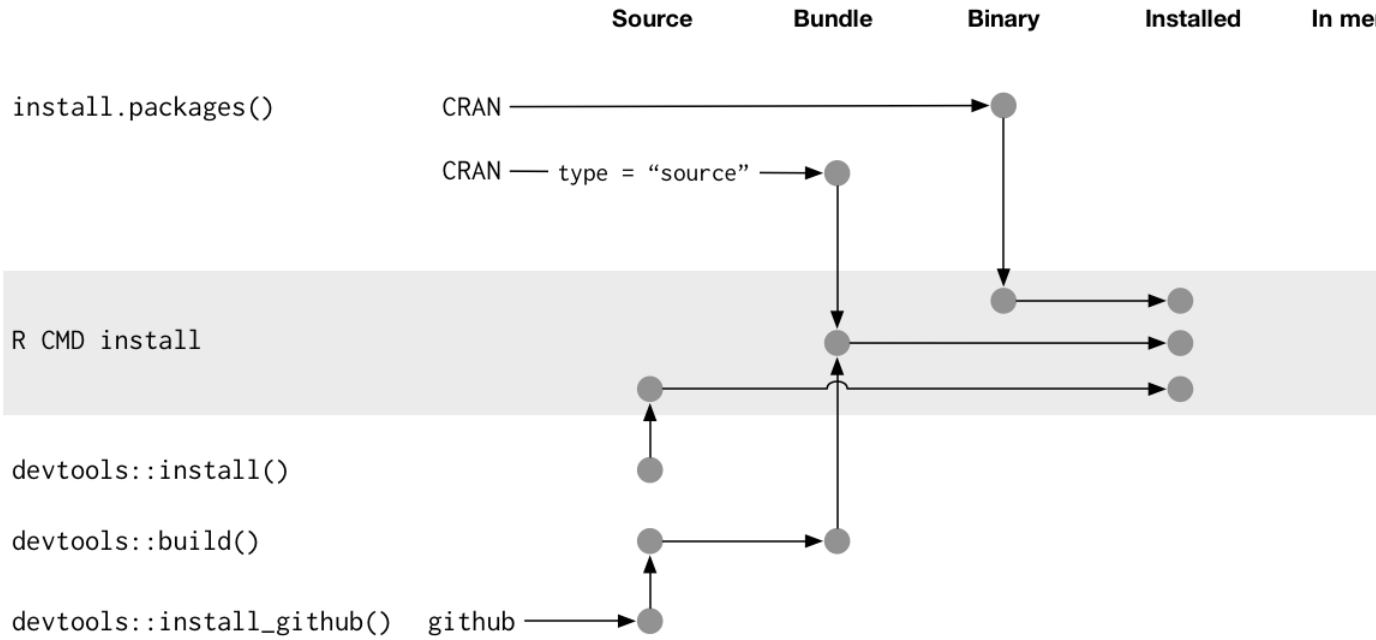
- 在 `R/` 目录中没有 `.R` 文件，而是有三个文件以有效的文件格式存储着解析的函数。这基本上是加载所有 R 代码，然后用 `save()` 保存函数的结果。(在这个过程中，这会添加一些额外的元数据，使得过程尽可能地快)。
- `Meta/` 目录中包含许多 `.rds` 文件。这些文件包含有关包的缓存元数据，如帮助文件所涵盖的主题和 `DESCRIPTION` 文件的解析版本。(您可以使用 `readRDS()` 查看这些文件中的内容)。这些文件通过缓存

代价高昂的计算使程序包更快地加载。

- 实际的帮助内容出现在 `help/` 和 `html/``（不再出现在 ``man/`）中。
- 如果 `src/` 目录中有任何代码，那么现在将有一个 `libs/` 目录，其中包含经过编译的代码。在 Windows 上，有 32 位（i386/）和 64 位（x64/）环境的子目录。
- 如果 `data/` 中有任何对象，则它们现在已转换为更具效率的形式。
- `inst/` 的内容被移动到顶层目录。例如，vignette 文件现在位于 `doc/` 中。
- 一些文件和文件夹已被删除，如 `README`、`build/`、`tests/` 和 `vignettes/`。

3.8.5 4.5 已安装的包（Installed package）

已安装的包是已解压缩到程序包库中的二进制包（如 4.7 所述）。下图说明了安装程序包的多种方法。这个图表很复杂！在理想情况下，安装包需要将一组简单的步骤串在一起：source -> bundle，bundle -> binary，binary -> installed。在现实世界中，这个过程并不是这么简单，因为通常有（更快的）快捷方式可用。



内置命令行工具 R CMD INSTALL 支持所有程序包的安装。它可以从源文件、捆绑包（也称为源码压缩包（source tarball））或二进制包安装程序包。有关详细信息，请参阅 R Installation and Administration 的 Installing packages 部分。与 `devtools::build()` 一样，devtools 提供了一个包装函数 `devtools::install()`，使该工具在 R 会话（R Session）中可用。

可以理解，大多数用户喜欢 R 会话（R Session）的舒适性，因此直接从 CRAN 安装软件包。内置函数 `install.packages()` 满足了这一需要。它可以以各种形式下载程序包并安装它，还可以选择程序包依赖项的安装。

devtools 公开了一系列 `install_*()` 函数，以满足某些超出 `install.packages()` 范围的需求，或者使现有功能更容易使用。这些功能实际上在 remotes packages 中维护，并由 devtools 重新导出。

```
library(remotes)

funs <- as.character(lsf.str("package:remotes"))
grep("^install_.+", funs, value = TRUE)
#>  [1] "install_bioc"      "install_bitbucket" "install_cran"
#>  [4] "install_deps"      "install_dev"       "install_git"
#>  [7] "install_github"    "install_gitlab"    "install_local"
#> [10] "install_svn"       "install_url"       "install_version"
```

`install_github()` 是这个子系列函数的最佳示例，这些函数可以从非 CRAN 的远程位置下载程序包，并执行安装包所需的任何操作。其余的 devtools/remotes `install()` 函数旨在使基本工具在技术上更简单或更明确一些，例如 `install_version()`，它能够安装特定版本的 CRAN 包。

与 `.Rbuildignore` 类似，如第 4.3.1 节所述，`.Rinstignore` 允许您将捆绑包中的文件保留在已安装包之外。然而，与 `.Rbuildignore` 相反，这个功能相当模糊，而且很少需要这样做。

TODO: Revisit this section later with respect to pak https://pak.r-lib.org.

3.8.6 4.6 内存中的包（In-memory package）

我们终于讲述到了一个每个使用 R 的人都熟悉的命令。

```
library(usethis)
```

假设已经安装了 usethis，这个语句将使得里面的所有函数可用，即现在我们可以执行以下操作：

```
create_package("/path/to/my/coolpackage")
```

这样，usethis 包已加载到内存中，并且实际上也已附加到搜索路径。在编写脚本时，加载和附加程序包之间的区别并不重要，但在编写程序包时非常重要。在 search path 中您将了解更多关于两者差异的信息，以及为什么它在搜索路径中很重要。

`library()` 并不是迭代调整和测试正在开发的程序包的好方法，因为它只适用于已安装的包。在第 5.4 节中，您将了解 `devtools::load_all()` 如何通过允许您将源码包直接加载到内存中来加速开发过程。

3.8.7 4.7 程序包的库（Package libraries）

我们刚刚讨论了 `library()` 函数，它的名字源于它的作用。当你调用 `library(foo)` 时，R 会在当前 **库**中查找一个叫做“foo”的已安装包，如果成功了，R 将让 foo 变得可以使用。

在 R 中，一个 **库**就是一个包含了已安装程序包的目录，有点像图书库。不幸的是，在 R 的世界，您将会经常遇到“库”和“包”的混淆用法。例如，delyr 是一个程序包，但是通常有人将其称为一个库。造成这种混乱的原因有几个。首先，R 的术语可以说是与更广泛的编程约定背道而驰的，“库”的通常含义更接近于我

们所说的“包”。library() 函数本身的名称可能会强化这一错误的关联。最后，这种词汇错误通常是无害的，因此 R 用户很容易养成错误的习惯，而指出这个错误的人看起来像是令人无法忍受的学究。但底线是：

> 我们使用 library() 函数加载[2] 一个程序包。

当您参与包开发时，两者之间的区别是重要且有用的。

您的计算机上可以有多个库。事实上，你们中的很多人已经这样做了，尤其是在 Windows 上。可以使用 .libPaths() 查看当前处于活动状态的库。以下是在 Windows 上的外观：

```
# on Windows
.libPaths()
#> [1] "C:/Users/jenny/Documents/R/win-library/3.6"
#> [2] "C:/Program Files/R/R-3.6.0/library"

lapply(.libPaths(), list.dirs, recursive = FALSE, full.names = FALSE)
#> [[1]]
#>   [1] "abc"           "anytime"       "askpass"       "assertthat"
#>  ...
#> [145] "zeallot"
#>
#> [[2]]
#>  [1] "base"         "boot"         "class"        "cluster"
#>  [5] "codetools"    "compiler"     "datasets"     "foreign"
#>  [9] "graphics"     "grDevices"    "grid"         "KernSmooth"
#> [13] "lattice"      "MASS"         "Matrix"       "methods"
#> [17] "mgcv"         "nlme"         "nnet"         "parallel"
#> [21] "rpart"        "spatial"      "splines"      "stats"
#> [25] "stats4"       "survival"     "tcltk"        "tools"
#> [29] "translations" "utils"
```

以下是在 macOS 上类似的表现（但您的结果可能会有所不同）：

```
# on macOS
.libPaths()
#> [1] "/Users/jenny/Library/R/3.6/library"
#> [2] "/Library/Frameworks/R.framework/Versions/3.6/Resources/library"

lapply(.libPaths(), list.dirs, recursive = FALSE, full.names = FALSE)
#> [[1]]
#>    [1] "abc"                  "abc.data"             "abind"
#>  ...
```

(下页继续)

[2] 实际上，library() 加载并附加一个程序包到环境中，但这是另一节（11.2）的主题。

(续上页)

```
#> [1033] "Zelig"                "zip"                  "zoo"
#>
#> [[2]]
#>  [1] "base"         "boot"         "class"        "cluster"
#>  [5] "codetools"    "compiler"     "datasets"     "foreign"
#>  [9] "graphics"     "grDevices"    "grid"         "KernSmooth"
#> [13] "lattice"      "MASS"         "Matrix"       "methods"
#> [17] "mgcv"         "nlme"         "nnet"         "parallel"
#> [21] "rpart"        "spatial"      "splines"      "stats"
#> [25] "stats4"       "survival"     "tcltk"        "tools"
#> [29] "translations" "utils"
```

在这两种情况下，我们可以看到两个活动库，它们的查询顺序如下：

1. 用户库
2. 系统级或全局库

这样的设置是 Windows 上的经典设置，但通常是 macOS 上需要选择的设置。[3] 在这样的设置之下，从 CRAN（或其他地方）安装的或本地开发的附加程序包保存在用户库中。和上面一样，macOS 系统被用作主要的开发机器，这里有很多软件包（大约 1000 个），而 Windows 系统只是偶尔使用，而且要简朴得多。R 附带的基本和推荐程序包的核心集位于系统级库中，这一点在 macOS 和 Windows 上是相同的。这种分离对许多开发人员很有吸引力，例如，在不干扰 base R 的安装的情况下使得清理附加包变得很容易。

如果您在 macOS 上只看到一个库，并不需要紧急更改任何内容。但下次升级 R 时，请考虑创建一个用户级库。默认情况下，R 查找存储在环境变量 `R_LIBS_USER` 中的路径下的用户库，默认为 `~/Library/R/x.y/library`。当您安装 R `x.y.z` 时，并且在安装任何附加程序包之前，请使用 `dir.create("~/Library/R/x.y/library")` 设置用户库。现在您将看到像上面一样的库设置。或者，您也可以在其他地方设置一个用户库，并通过在 `.Renviron` 中设置 `R_LIBS_user` 环境变量来告诉 R。

这些库的文件路径也清楚地表明它们与特定版本的 R（在编写本文时是 3.6.x）相关联，这也是经典的。这反映并强化了这样一个事实：当您将 R 从 3.5 更新到 3.6，即一个在 ** 次要（minor）** 版本上的更改时，您需要重新安装附加程序包。对于在 ** 补丁（patch）** 版本上的更改，例如从 R 3.6.0 到 3.6.1，通常不需要重新安装附加程序包。

随着 R 的使用变得越来越复杂，开始更加有意地管理程序包库是十分平常的。例如，像 renv（及其前身 packrat）这样的工具可以使管理项目特定库的过程自动化。这对于使数据产品具有可复制性、可移植性和相互隔离性非常重要。程序包开发人员可能会在库的搜索路径前添加一个临时库，其中包含一组特定版本的程序包，以便在不影响其他日常工作的情况下探索前后兼容性问题。反向依赖性检查（Reverse dependency checks）是另一个显式管理库的搜索路径的例子。

以下是按范围和持久性顺序控制哪些库处于活动状态的主要杠杆：

- 环境变量，如 `R_LIBS` 和 `R_LIBS_USER`，它们在启动时被查询。

[3] 有关更多详细信息，请参阅 *What They Forgot To Teach You About R* 中的 Maintaining R Section

- 使用一个或多个文件路径调用 `.libPaths()`。
- 通过 `withr::with_libpaths()` 使用临时更改的库搜索路径执行小型的代码段。
- 单个函数的参数，比如 `install.packages(lib =)` 和 `library(lib.loc =)`。

最后，需要注意的是，`library()` **永远**不应该在 *程序包* 中使用。程序包和脚本依赖于不同的机制来声明它们的依赖性，这是您需要在您的心理模型（mental model）和习惯中做出的最大调整之一。我们将在第 11 章全面探讨这个话题。

3.9 第五章基本开发工作流程

在第 4 章中略微了解了 R 程序包和库后，在这里，我们提供了创建程序包以及使其转变为开发过程中出现的不同状态的基本工作流。

3.9.1 5.1 创建程序包

5.1.1 调查现有运行环境

许多程序包都是由于一个人对一些本应更容易完成的普通任务感到沮丧而产生的。您应该如何判断某些东西是否值得制作为程序包呢？虽然这个问题没有明确的答案，但了解至少两种类型的回报对您是有帮助的：

- 结果（Product）：当这个功能正式实现时，您的工作生活会变得更好。
- 过程（Progress）：更好地掌握 R 会使您的工作更有效率。

如果您只关心结果带来的好处，那么您的主要目标就是在现有的程序包中探索。Silge，Nash 和 Graves 在 useR! 2017 组织了一次调查和会议。他们为 R Journal（Silge、Nash 和 Graves 2018）撰写的文章提供了全面的资源综述。

如果您正在寻找方法来提高对 R 的掌握，那么您仍然应该对 R 的运行环境多加了解。但是，即使有相关的前期工作，也有很多充分的理由来制作自己的程序包。专家们的方式是通过实践来为许多功能构建程序包，通常是非常基本的功能，并且您应该有同样的机会通过修补来学习。如果您只被允许做一些从未接触过的事情，那么您可能会遇到一些非常模糊或非常困难的问题。茫然不知所云

最后，根据用户界面、默认值和在极端情况下的表现来评估现有工具的适用性也是有效的。如果一个程序包在技术上可以满足您的需要，但是对于您的用例来说非常不符合舒适正确的使用方式，那么仍然可以说它不能满足您的需求。在这种情况下，开发自己的实现方法或编写隐藏了 sharp edges 疑问? 的封装函数仍然是有意义的。

5.1.2 为您的程序包取名

> “在计算机科学中只有两件困难的事：缓存失效和命名。” —Phil Karlton

在创建程序包之前，您需要为它取一个明自。这可能是创建程序包的过程中最困难的部分！（尤其是因为没有人可以为您实现取名的自动化。）

5.1.2.1 正式的要求

有三个正式的要求：

1. 名称只能由字母、数字和句点组成，即 .。
2. 它必须以字母开头。
3. 它不能以句点结尾。

不幸的是，这意味着您不能在您的包名中使用连字符或下划线，即 - 或 \。我们建议不要在包名中使用句点，因为这会混淆句点与文件扩展名和 S3 方法的关联。

5.1.2.2 实用的建议

如果您打算和别人分享您的程序包，那么花几分钟想一个好名字是值得的。以下是一些需要考虑的事项：

- 选择一个便于 Google 搜索的独特名称。这使得潜在的用户能够很容易地找到您的程序包（以及相关的资源），并能让您看到是谁在使用它。
- 不要选择一个已经在 CRAN 或 Bioconductor 上使用的包名。您可能还需要考虑一些其他类型的命名冲突：
  - 是否有在 GitHub 上成熟的且正在开发中的程序包，该程序包已经有了一定的历史，并且似乎即将发布？
  - 这个名称是否已经用于另一个软件，例如是 Python 或 JavaScript 生态系统中的库或框架？
- 避免同时使用大写和小写字母：这样做会使包名称难以键入，甚至难以记住。例如，很难记住一个程序包叫做 Rgtk2 还是 RGTK2 或 RGtk2。
- 优先选择可发音的名字，这样人们在谈论你的程序包时会很舒服，并且能够在他们的脑海里听到它。
- 找到一个能唤起对问题的联想的单词，并对其进行修改，使其具有唯一性：
  - lubridate 使日期和时间更容易。
  - rvest 从网页中“收获”内容。
  - r2d3 提供了使用 D3 可视化的实用程序。
  - forcats 是因子（factors）的变位词，我们用它来表示分类数据（ **for cat**egorical data ）。
- 使用缩写：
  - RCpp = R + C++ (Plus Plus)
  - brms = 使用 Stan 的贝叶斯回归模型（Bayesian Regression Models using Stan）
- 名字后添加额外的字符 R：
  - stringr 提供字符串工具。
  - beepr 播放通知声音。

  - callr 从 R 调用 R。

- 别被起诉。
  - 如果您要创建一个与商业服务交互的包，请查看商标使用指南。例如，rDrop 不被称为 rDropbox，因为 Dropbox 禁止任何应用程序使用完整的商标名。

Nick Tierney 在他的 Naming Things 博客文章中展示了一个有趣的程序包名称类型学；请参阅该文章以获取更多鼓舞人心的示例。他也有一些重命名包的经验，因此，如果您第一次取名没有做对，他的博客文章 So, you' ve decided to change your r package name 将是一个很好的资源。

5.1.2.3 使用 available 程序包

同时遵守上述所有建议是十分困难的，因此您显然需要做出一些权衡。available 程序包中有一个名为 `available()` 的函数，可以帮助您从多个角度评估可能的程序包名称：

```
library(available)

available("doofus")
#> Urban Dictionary can contain potentially offensive results,
#>   should they be included? [Y]es / [N]o:
#> 1: 1
#>   doofus
#> Name valid: ✓
#> Available on CRAN: ✓
#> Available on Bioconductor: ✓
#> Available on GitHub:  ✓
#> Abbreviations: http://www.abbreviations.com/doofus
#> Wikipedia: https://en.wikipedia.org/wiki/doofus
#> Wiktionary: https://en.wiktionary.org/wiki/doofus
#> Sentiment:???
```

‘‘available::available()‘‘ 执行以下操作：

- 检查有效性。
- 检查在 CRAN、Bioconductor 和其他产品上的可用性。
- 搜索各种网站，帮助您发现任何意料之外的含义。在交互式会话中，您在上面看到的 URLs 将在浏览器选项卡中打开。
- 试图报告该名称是否有积极情绪或消极情绪。

5.1.3 程序包的创建

为程序包命名后，有两种创建程序包的方法：

- 调用 `usethis::create_package()`。
- 在 RStudio 中，依次点击 *File* > *New Project* > *New Dictionary* > *R Package*，它最终会调用 `usethis::create_package()`，所以实际上只有一种创建程序包的方法。

TODO: revisit when I tackle usethis + RStudio project templates https://github.com/r-lib/usethis/issues/770. In particular, contemplate whether to reinstate any screenshot-y coverage of RStudio workflows here.

这将产生最小的 可工作的程序包，它包含三个组件：

1. 一个 `R/` 目录，您将在 R Code 中了解到具体内容。
2. 一个基础的 `DESCRIPTION` 文件，您将在 package metadata 中了解到具体内容。
3. 一个基础的 `NAMESPACE` 文件，您将在 the namespace 中了解到具体内容。

它也可能包含一个 RStudio 项目文件，`pkgname.Rproj`，这使您的程序包易于与 RStudio 一起使用，如下所述。基础的 `.Rbuildignore` 和 `.gitignore` 文件也被包含在目录中。

不要使用 `package.skeleton()` 创建程序包。因为这个函数与 R 一起提供，您可能会想使用它，但是它会创建一个在调用 R CMD build 时立刻抛出错误的程序包。它期望的开发过程与我们在这里使用的不同，所以修复这个损坏的初始状态只会让使用 devtools（尤其是 roxygen2）的人做不必要的工作。请使用 `create_package()`。

5.1.4 您应该在哪里执行 `create_package()` ?

`create_package()` 的主要且唯一必需的参数是新程序包所在的 `path`：

```
create_package("path/to/package/pkgname")
```

请记住，这是您的程序包在**源代码**形式（第 4.2 节）时所处的位置，而不是**已安装**形式（第 4.5 节）。已安装的包位于**库**中，我们在第 4.7 节中讨论了库的常规设置。

源码包应该放在哪里？主要原则是该位置应该与已安装包所在的位置不同。在没有其他外部考虑的情况下，典型的用户应该在其主目录中为 R（源代码）包指定一个目录。我们与同事讨论过这一点，您最喜欢的一些 R 包的源代码位于 `~/rrr/`、`~/documents/tidyverse/`、`~/R/packages/` 或 `~/pkg/` 等目录中。我们中的一些人使用一个目录来实现这一点，其他人则根据他们的开发角色（contributor vs. not）、GitHub 组织（tidyverse vs r-lib）、开发阶段（active vs. not）等将源码包划分为几个目录。

以上内容可能反映出我们主要是工具构建者。学术研究人员可能会围绕单个出版物组织他们的文件，而数据科学家可能会围绕数据产品和报告来组织。对于每一种特定的方法，没有特定的技术或传统原因来说明为何要选择它。只要在源码包和已安装的包之间保持清晰的区别，仅仅需要选择一种在整个系统中有效的文件组织策略，并始终如一地使用它即可。

3.9.2 5.2 RStudio 项目

devtools 与 RStudio，一个我们相信是大多数 R 用户的最佳开发环境联系紧密、携手合作。明确地说，您可以使用 devtools 而不使用 RStudio，也可以在 RStudio 中开发程序包而不使用 devtools。但是这种特殊的、

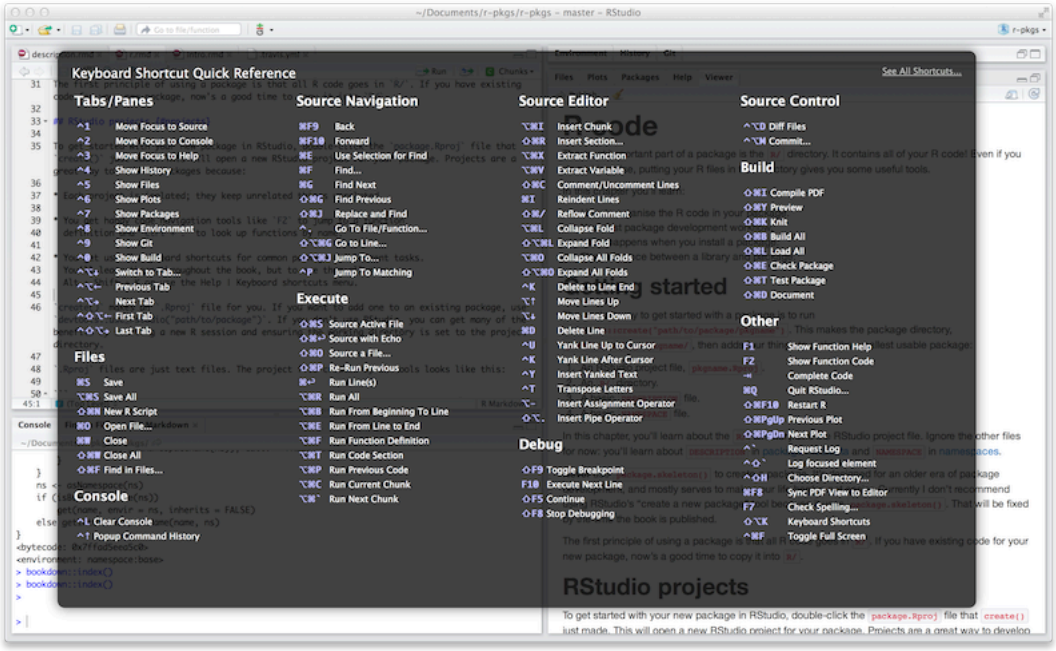
双向的关系使得一起使用 devtools 和 RStudio 变得非常有意义。

一个 RStudio 项目（Project，包含一个大写字母 “P”），是您计算机上的一个常规目录，其中包含一些（大部分是隐藏的）RStudio 基础文件，以便您在一个或多个项目（project，带有小写的 “P”）上工作。一个项目（project）可以是一个 R 包、一个数据分析报告、一个 Shiny app、一本书、一个博客等等。

5.2.1 RStudio 项目的好处

从第 4.2 节中，您已经知道源码包位于您计算机上的目录中。我们强烈建议将每个源码包作为一个 RStudio 项目。以下是这样做的好处：

- 项目是非常 “可启动的”（launch-able）。文件浏览器和工作目录将完全按照您需要的方式进行设置，马上可以开始工作，从而很容易在一个项目中启动一个新的 RStudio 实例，。
- 每个项目都是独立的；在一个项目中运行的代码不会影响任何其他项目。
  - 您可以同时打开多个 RStudio 项目，并且在项目 A 中执行的代码不会对项目 B 的 R session 和工作区（workspace）产生任何影响。
- 您可以使用方便的代码导航工具，如 `F2` 跳转到函数定义，`Ctrl + .` 来按名称查找函数或文件。
- 您可以使用很有帮助的键盘快捷键和可点击的界面，以执行常见的程序包开发任务，如生成文档、运行测试或检查整个程序包。



查看最有用的键盘快捷键，请按 `Alt + Shift + K`，或者使用 *Help* > *Keyboard Shortcuts Help*。

在 Twitter 上关注 @rstudiotips 以获取 RStudio 的常规提示和使用技巧。

5.2.2 怎样开始 RStudio 项目

如果您按照我们的建议使用 `create_package()` 创建新的程序包，那么这会自行解决。如果你在 RStudio 工作，每个新程序包也将是一个 RStudio 项目。

有多种方法可以将预先存在源码包的目录指定为 RStudio 项目：

- 在 RStudio 中，执行 *File* > *newproject* > *Existing Directory*。
- 使用预先存在的 R 源包的路径调用 `create_package()`。

- 调用 `usethis::use_rstudio()`，将活动的 usethis 项目设置为现有的 R 包。实际上，这可能意味着您只需要确保工作目录在已经存在的程序包中。

5.2.3 RStudio 项目文件是什么？

RStudio 项目的目录将包含一个 `.Rproj` 文件。通常，如果目录名为“foo”，则项目文件为 `foo.Rproj`。如果这个目录也是一个 R 包，那么包名通常也是“foo”。故障最少的方法是使所有这些名称一致，并且**不要**将程序包嵌套在项目内的子目录中。如果您决定采用其他工作流程，那么可能会让您觉得您在与工具进行不必要的争斗。

`.Rproj` 文件只是一个文本文件。以下是 usethis 使用的默认项目文件：

```
Version: 1.0

RestoreWorkspace: No
SaveWorkspace: No
AlwaysSaveHistory: Default

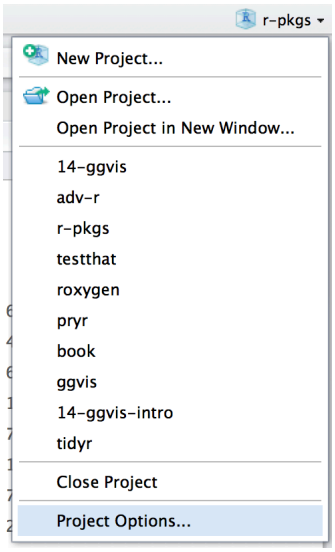
EnableCodeIndexing: Yes
Encoding: UTF-8

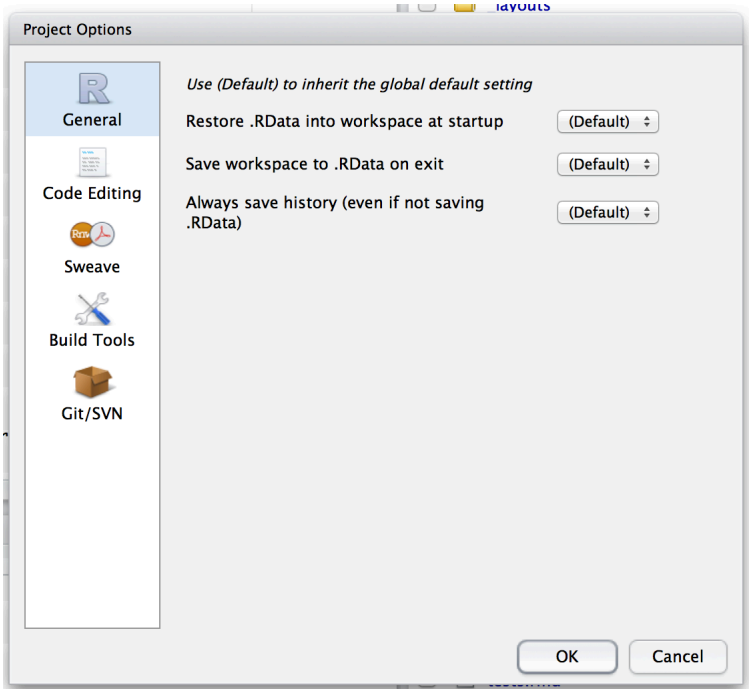
AutoAppendNewline: Yes
StripTrailingWhitespace: Yes

BuildType: Package
PackageUseDevtools: Yes
PackageInstallArgs: --no-multiarch --with-keep.source
PackageRoxygenize: rd,collate,namespace
```

您不需要手动修改这个文件。可以通过 *Tools* > *Project Options* 或者右上角 Project 菜单栏中的 *Project Options* 提供的界面进行编辑。

TODO: update these and deal with layout.





5.2.4 怎样启动一个 RStudio 项目

在 macOS 的 Finder 或 Windows 资源管理器中双击 `foo.Rproj` 文件，以便在 RStudio 中启动 foo Project。

您也可以通过 *File > Open Project (in New Session)* 或右上角的 Project 菜单从 RStudio 中启动 Projects。

如果您使用的是一个生产力应用程序或启动器应用程序，您可能可以将其配置为对 `.Rproj` 文件执行一些令人愉快的操作。我们都使用 Alfred 来实现这一点，[4] 只有 macOS 有该工具，但是 Windows 也有类似的工具。事实上，这是一个非常好的理由首选使用生产力应用程序。

[4] 具体来说，当建议打开应用程序或文件时，我们配置 Alfred 在其搜索结果中优先打开 `.Rproj` 文件。要向 Alfred 注册 `.Rproj` 文件类型，请转到 *Preferences > Features > Default Results > Advanced*。将任意 `.Rproj` 文件拖到此位置，然后关闭即可。

一次性打开多个项目是非常正常的——而且富有成效！

5.2.5 RStudio Project vs. active usethis project

您会注意到，大多数 usethis 函数不使用路径：它们对“active usethis project”中的文件进行操作。usethis 程序包假设以下这些在 95% 的时间内都是一致的：

- 当前的 RStudio Project，如果使用 RStudio。
- 活跃的 usethat 项目。
- R 进程的当前工作目录。

如果事情看起来很奇怪，可以调用 `proj_sitrep()` 来获取“情况报告”。这将识别出一些特殊情况，并提出如何回到更良好的状态。

```
# these should usually be the same (or unset)
proj_sitrep()
#> *   working_directory: '/Users/jenny/rrr/readxl'
#> * active_usethis_proj: '/Users/jenny/rrr/readxl'
#> * active_rstudio_proj: '/Users/jenny/rrr/readxl'
```

3.9.3 5.3 工作目录和文件路径规范

在开发包时，您将会执行 R 代码。这将是工作流调用（例如 `document()` 或 `test()`）和帮助您编写函数、示例和测试的特殊（*ad hoc*）调用的混合。我们强烈建议您将 R 进程的工作目录设置为源码包的顶层目录。

如果您在程序包开发方面毫无经验，那么您没有太多的基础来支持或抵制此建议。但那些有经验的人可能会觉得有些不安。在子目录中工作时，我们应该如何表示路径，比如 `tests/` ？当它变得与我们的工作相关时，我们将向您展示如何利用路径构建帮助器，例如 `testthat::test_path()`，它会在执行时确定路径。

它的基本思想是，通过不使用工作目录，鼓励您编写能够明确表达意图的路径（“从测试目录中读取 `foo.csv`），而不是隐式的表达（“从当前的工作目录中读取 `foo.csv`，我认为该目录将是测试目录）。依赖隐式路径的一个可靠迹象就是不断地修改工作目录，因为您正在使用 `setwd()` 手动实现路径中隐含的假设。

这种思想可以消除所有路径问题，让日常的开发变得更加愉快。隐式路径难以正确设置的原因有两个：

- 回想一下在开发周期中程序包可以采用的不同形式（第 4 章）。在这些状态中，存在哪些文件和文件夹以及它们在层次结构中的相对位置都是互不相同的。编写满足所有程序包状态的相对路径是很困难的。
- 最终，您的程序包将由您和 CRAN 使用内置工具（built-in tools）处理，如 `R CMD build`、`R CMD check` 和 `R CMD INSTALL`。很难跟踪这些过程中每个阶段的工作目录是什么。

像 `testthat::test_path()`、`fs::path_package()` 和 rprojroot package 这样的路径帮助器对于构建弹性的路径非常有用，这些路径可以在开发和使用过程中出现的所有情况下都有效。消除脆弱路径的另一种方法是严格使用在程序包中存储数据的适当方法（第 12 章），并在适当的时候采用会话（session）的临时目录，例如针对短暂的测试工件（ephemeral testing artefacts）（第 10 章）。

3.9.4 5.4 使用 load_all() 测试函数

load_all() 函数无疑是 devtools 工作流中最重要的部分。

```
# with devtools attached and
# working directory set to top-level of your source package ...

load_all()

# ... now experiment with the functions in your package
```

load_all() 是在“lather, rinse, repeat”这一程序包开发周期中的关键步骤：

1. 调整函数定义。
2. load_all()
3. 通过运行一个较小的示例或一些测试来尝试更改。

当您刚接触程序包开发或 devtools 时，很容易忽视 load_all() 的重要性，并在数据分析工作流中养成一些不合适的习惯。

5.4.1 load_all() 的好处

当您第一次开始使用开发环境，如 RStudio 或 Emacs + ESS 时，最大的便利之处是能够从 .R 脚本中发送代码到 R 控制台（R console）中执行。这种流动性使得遵循将源代码视为真实存在[5]（而不是工作区中的对象）和保存 .R 文件（而不是保存和重新加载 .Rdata）的最佳做法是可以接受的。疑惑

load_all() 对于程序包开发来说有着同样的意义，相反的是，它要求您**不要**像脚本代码那样测试程序包代码。load_all() 模拟查看源代码更改效果的完整过程，这是一个非常笨重的[6] 过程，您不会希望经常这样做。load_all() 的主要优点有：

- 您可以快速迭代，这将鼓励探索和渐进式开发过程。
  - 这种迭代加速对于具有编译代码的程序包来说尤其显著。
- 您可以在命名空间机制下进行交互开发，该机制准确模拟了其他人使用您的已安装程序包时的情况：
  - 您可以直接调用自己的内部函数，而不必使用 :::，也不必在全局工作区中临时定义函数。
  - 您还可以从导入到 NAMESPACE 的其他程序包中调用函数，而不必试图通过 library() 附加这些依赖项。

load_all() 消除了开发工作流中的麻烦，也消除了使用替代方法的诱惑，该替代方法通常会导致命名空间和依赖项管理方面的错误。

[5] 引用 Emacs Speaks Statistics (ESS) 所支持的使用哲学。

[6] 使用命令行的方法是退出 R，转到 shell，在程序包的父目录中执行 R CMD build foo，然后执行 R CMD INSTALL foo_x.y.x.tar.gz，重新启动 R，并调用 library(foo)。在 R 中，一个近似的做法是 detach("package:foo", unload = TRUE); install.packages(".", repos = NULL, type = "source"); library(foo)。

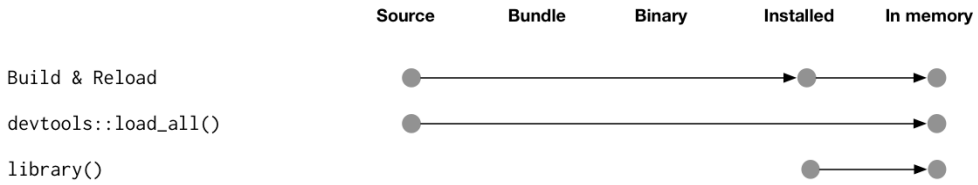
5.4.2 其它调用 load_all() 的方法

在程序包 Project 中工作时，RStudio 提供了几种调用 load_all() 的方法：

- 键盘快捷键：Cmd + Shift + L (macOS)、Ctrl + Shift + L (Windows, Linux)
- Build 窗格的 *More* ⋯菜单
- *Build* > *Load All*

devtools::load_all() 是 pkgload::load_all() 的一个简单封装，它增加了一点用户友好性。您不太可能以编程的方式或在另一个程序包中使用 load_all()，但如果您这样做了，您可能应该直接使用 pkgload::load_all()。

TODO: Decide how to update this diagram and then reposition and re-integrate it with the prose. For example, figure out how to frame w.r.t. RStudio Install and Restart vs. Clean and Rebuild.



3.9.5 参考文献

Silge, Julia, John C. Nash, and Spencer Graves. 2018. "Navigating the R Package Universe." The R Journal 10 (2):558–63. https://doi.org/10.32614/RJ-2018-058.

3.10 第六章 R 代码

使用程序包的第一个原则是所有 R 代码都放在 R/ 中。在本章中，您将了解 R/ 目录、我对将函数组织到文件中的建议，以及一些有关良好风格的提示。您还将了解脚本中的函数和程序包中的函数之间的一些重要区别。

3.10.1 6.1 R 代码工作流

使用程序包的第一个实际的优点是，很容易重新加载您的代码。可以运行 devtools::load_all()，或者在 RStudio 中按下 **Ctrl/Cmd + Shift + L**，同时保存所有打开的文件，以节省按键次数。

利用这个快捷键可建立一个流畅的开发流程。

1. 编辑一个 R 文件。
2. 按 Ctrl/Cmd + Shift + L。
3. 在控制台中浏览代码。

4. 修改代码，重复上面的过程。

祝贺您！您已经学到了第一个程序包的开发流程！即使您从本书中没有学到任何其他的东西，也已经了解了编辑和重新加载 R 代码的一个有用的工作流程。

3.10.2 6.2 组织您的函数

removed in deference to material in https://style.tidyverse.org; see tidyverse/style/#121

3.10.3 6.3 代码风格

removed in deference to material in https://style.tidyverse.org; see tidyverse/style/#122

TL;DR = "Use the styler package" .

3.10.4 6.4 顶层代码

到目前为止，您可能一直在编写**脚本**，使用 `source()` 加载保存在文件中的 R 代码。脚本和程序包中的代码有两个主要区别：

- 在脚本中，代码在加载时运行。在程序包中，代码在编译时运行。这意味着您的程序包代码应该只创建对象，其中绝大多数是函数。
- 程序包中的函数将被用于您没有想象过的情况。这意味着您的函数需要小心处理它们与外界之间的交互。

接下来的两节将讨论这些重要的差异。

6.4.1 加载代码

当您用 `source()` 加载脚本时，每一行代码都会执行，且执行的结果可以立刻使用。对程序包来说，情况有所不同，它的加载过程分为两步。当包在编译时（例如通过 CRAN），`R/` 目录下所有的代码都会被执行，结果会被保存下来。当使用 `library()` 或 `require()` 加载一个程序包时，这些保存的结果就可以供您使用了。如果用这个方式来加载脚本的话，代码看起来是这样的：

```
# Load a script into a new environment and save it
env <- new.env(parent = emptyenv())
source("my-script.R", local = env)
save(envir = env, "my-script.Rdata")

# Later, in another R session
load("my-script.Rdata")
```

以 `x <- Sys.time()` 为例，如果您把它放入一个脚本中，`x` 会告诉您脚本是什么时候被执行 `source()` 的。但是如果您把相同的代码放入程序包中，`x` 会告诉您程序包是什么时候被编译的。

这意味着您不应该在程序包的顶层运行代码：程序包的代码只能创建对象，大部分是函数。例如，假设你的 foo 程序包包含这样的代码：

```
library(ggplot2)

show_mtcars <- function() {
    qplot(mpg, wt, data = mtcars)
}
```

如果某人试图这样使用它：

```
library(foo)
show_mtcars()
```

该代码不会工作，因为 ggplot2 的 `qplot()` 函数不可用：`library(foo)` 不会执行 `library(ggplot2)`。程序包的顶层代码只会在程序包被编译的时候执行，而不是加载的时候。

为了解决这个问题，您可能会做如下修改：

```
show_mtcars <- function() {
    library(ggplot2)
    qplot(mpg, wt, data = mtcars)
}
```

一会儿您将会看到，这同样是有问题的。需要在 `DESCRIPTION` 中描述您的代码所需要的程序包，您将在 package dependencies 学到这一内容。

6.4.2 R 运行环境

脚本和程序包的另一个巨大区别是：别人会使用您的程序包，并且会在一个您从未想到的环境中使用它。这意味着你需要注意 R 的运行环境，这不仅包括那些可用的函数和对象，也包括所有的全局设置。如果用 `library()` 加载了一个包，或者用 `options()` 修改了一个全局设置，或者利用 `setwd()` 修改了工作目录，那么您已经修改了 R 的运行环境。如果有其他函数的行为在运行您的函数前后发生了改变，那么您就已经修改了 R 的运行环境。修改 R 的运行环境是不好的，因为这会使得代码很难理解。

有些修改全局设置的函数不应该被使用，因为有更好的替代方法：

- **不要使用**`library()`** 或者**`require()`。这些函数修改了搜索路径，影响了全局环境下可用的函数。更好的方式是用 `DESCRIPTION` 来指定您的程序包的需求，这将在下一章说明。这种方式也保证了您的程序包被安装时，它需要的程序包也会被安装。
- **不要使用**`source()` 从文件加载代码。`source()` 会将代码执行的结果添加到当前环境，因此会修改当前环境。您可以使用工具 `devtools::load_all()`，它会自动加载 `R/` 目录下所有的文件。如果您

要用 source() 来建立数据集，请使用 data/ 目录，这将在 datasets 中讲到。

还有其他一些函数需要谨慎使用。如果你要使用它们，请确保使用 on.exit() 在退出的时候清理干净。

- 如果你修改全局的 options() 或图形的 par()，先保存好旧的设置，然后在你用完之后恢复到原来的值：

```
old <- options(stringsAsFactors = FALSE)
on.exit(options(old), add = TRUE)
```

- 不要修改工作目录。如果必须修改它，确保在您完成工作后改回去：

```
old <- setwd(tempdir())
on.exit(setwd(old), add = TRUE)
```

- 创建图像和输出到控制台是另外两种影响 R 全局环境的方式。通常你无法避免这些（因为它们很重要！），但好的做法是把它们封装成**只能**产生输出的独立的函数。这也使得其他人更容易将你的工作用于新的用途。例如，如果你将数据准备和绘图分成两个函数，其他人可以使用你的数据准备工作（通常是最难的部分！）来创建新的可视化结果。

另一方面，您应该避免依赖用户的运行环境，因为这些环境可能和你的不同。例如，函数 read.csv() 是危险的，因为 stringsAsFactors 参数的值是来自全局的 stringsAsFactors 参数。如果您希望它是 TRUE``(默认值)，但用户如果把它设为 ``FALSE，那您的代码就可能会出错。

6.4.3 何时需要副作用

偶尔，程序包确实需要一些副作用。最常见的情况是，您的程序包需要与外部系统进行交互——当程序包加载时，您可能需要做一些初始化设置。为此，您可以使用两个特殊函数：.onLoad() 和 .onAttach()。当程序包加载和附加时，这两个函数会被调用。在 Namespaces 中您会了解到这两者的区别。目前您应该总是使用 .onLoad()，除非明确指出应该使用 .onAttach()。

.onLoad() 和 .onAttach() 的常见用法包括以下这些。

- 在程序包加载时显示一些有用的信息。这可以使得程序包的使用条件明确，或者显示一些有用的提示。启动信息是一个您应该使用 .onAttach() 而不是 .onLoad() 的地方。要显示启动消息，请总是使用 packageStartupMessage() 而不是 message()``(这可以让 ``suppressPackageStartupMessages() 函数来选择是否显示包的启动消息)。

```
.onAttach <- function(libname, pkgname) {
    packageStartupMessage("Welcome to my package")
}
```

- 用 options() 来为您的程序包设置自定义选项。为避免和其他程序包的冲突，要确保选项名使用您的程序包名作为前缀。还要注意不要覆盖用户已设置的选项。

我在 devtools 中使用下面的代码来建立选项：

```
.onLoad <- function(libname, pkgname) {
    op <- options()
    op.devtools <- list(
        devtools.path = "~/R-dev",
        devtools.install.args = "",
        devtools.name = "Your name goes here",
        devtools.desc.author = "First Last <first.last@example.com> [aut, cre]",
        devtools.desc.license = "What license is it under?",
        devtools.desc.suggests = NULL,
        devtools.desc = list()
    )
    toset <- !(names(op.devtools) %in% names(op))
    if(any(toset)) options(op.devtools[toset])

    invisible()
}
```

然后 devtools 函数可以使用比如 `getOption("devtools.name")` 来获得程序包作者的名字，或者判断一个默认值是否已经被设置。

- 把 R 连接到另一种编程语言。例如，如果你使用 rJava 来跟一个 `.jar` 文件交互，你需要调用 `rJava::jpackage()`。要想在 R 中使用 Rcpp 模块来引用 C++ 类，可以调用 `Rcpp::loadRcppModules()`。
- 使用 `tools::vignetteEngine()`，注册一个 vignette 生成引擎。

正如您在上面的例子中看到的，`.onLoad()` 和 `.onAttach()` 函数带有两个参数：`libname` 和 `pkgname`。但它们很少使用（当需要使用 `library.dynam()` 来加载已编译的代码时，它们才会被用到）。它们给出了程序包安装的路径（也就是库），以及程序包的名称。

如果您使用了 `.onLoad()`，请考虑使用 `.onUnload()` 来清理任何副作用。按照惯例，`.onLoad()` 以及相关函数通常保存在一个叫 `zzz.R` 的文件中。(注意，`.First.lib()` 和 `.Last.lib()` 是 `.onLoad()` 和 `.onUnload()` 的老版本，不应该继续使用了。)

6.4.4 S4 类、泛型和方法

另一种类型的副作用是定义 S4 类、方法和泛型。R 包会捕捉这些副作用，以便当包被加载的时候可以重现它们，但它们需要按照正确的顺序调用。例如，在定义一个方法之前，你必须定义泛型和类。这要求 R 文件按照指定的顺序加载。这一顺序是由 `DESCRIPTION` 文件中的 `Collate` 字段来控制的。在 docimenting S4 中有详尽的描述。

3.10.5 6.5 CRAN 注记

(每章的最后都会给出提交程序包到 CRAN 的一些提示。如果不打算提交你的程序包到 CRAN，可以忽略这些内容!)

如果打算提交您的程序包到 CRAN，您在 .R 文件中就只能使用 ASCII 字符。但您仍然可以在字符串中包含 Unicode 字符，这需要使用特殊的 Unicode 转义格式（例如 "\u1234"）。最简单的做法是使用 stringi::stri_escape_unicode()：

```
x <- "This is a bullet •"
y <- "This is a bullet \u2022"
identical(x, y)
#> [1] TRUE

cat(stringi::stri_escape_unicode(x))
#> This is a bullet \u2022
```

您可以将 .R 文件传入 tools::ShowNonASCIIfile() 以检测包含非 ASCII 字符的所有行：

```
library(purrr)

walk(list.files("R", full.names = TRUE),
    tools::showNonASCIIfile)
```

3.11 第七章程序包元数据

DESCRIPTION 文件的功能是储存关于您的程序包的重要元数据。当您第一次开始编写程序包时，您将主要使用这些元数据来记录运行您的程序包所需的其他程序包。然而，随着时间的推移，您开始与其他人共享您的包，元数据文件变得越来越重要，因为它指定了谁可以使用它（许可证）以及如果有什么问题时与谁联系(您!)。

每个程序包都必须有 DESCRIPTION 文件。事实上，它是定义一个程序包的关键特征（RStudio 和 devtools 将包含 DESCRIPTION 的任何目录都视为程序包）。首先，usethis::create_package("mypackage") 会自动添加一个基本的描述文件。这将允许您开始编写程序包而不必担心元数据，直到您需要为止。最基础的描述会根据您的设置有所不同，但应该看起来像下面这样：

```
Package: mypackage
Title: What The Package Does (one line, title case required)
Version: 0.1
Authors@R: person("First", "Last", email = "first.last@example.com",
                  role = c("aut", "cre"))
Description: What the package does (one paragraph)
```

(下页继续)

(续上页)

```
Depends: R (>= 3.1.0)
License: What license is it under?
LazyData: true
```

(如果您编写了很多程序包，可以通过 devtools.desc.author, devtools.desc.license, devtools.desc.suggests 和 devtools.desc 设置全局选项。使用 package?devtools 了解更多详细信息。)

DESCRIPTION 使用了一个名为 DCF 的简单文件格式，即 Debian 控件格式（Debian Control Format）。您可以在下面的简单示例中看到大部分结构。每行由一个**字段**名和一个值组成，用冒号分隔。当值跨越多行时，需要行首缩进：

```
Description: The description of a package is usually long,
    spanning multiple lines. The second and subsequent lines
    should be indented, usually with four spaces.
```

本章将向您展示怎样使用最重要的一些 DESCRIPTION 字段。

3.11.1 7.1 依赖项：您的程序包需要什么？

DESCRIPTION 的作用是列出您的程序包工作时需要的其他程序包。R 有一套丰富的方法来描述潜在的依赖关系。例如，以下几行表示您的包需要 ggvis 和 dplyr 才能工作：

```
Imports:
    dplyr,
    ggvis
```

然而，下面的几行表示，虽然您的程序包可以利用 ggvis 和 dplyr，但并不需要它们：

```
Suggests:
    dplyr,
    ggvis
```

Imports 和 Suggests 都采用逗号分隔的程序包名称列表。我建议每行放一个程序包，并按字母顺序排列。这样就很容易浏览。

Imports 和 Suggests 在依赖程度上有所不同：

- Imports：这里列出的程序包 *必须*存在，您的程序包才能正常工作。实际上，无论何时安装程序包，它们（如果尚未存在）都将安装在您的计算机上（devtools::load_all() 还会检查程序包是否已安装）。

  > 在此处添加程序包依赖项可确保程序包能够正确安装。但是，这并不意味着它将与您的程序包一起添加到环境中（即 library(x)）。最佳方法是在代码中使用语法 package::function() 显式引用外部函数。这使得我们很容易确定哪些函数位于程序包之外。并且在以后阅读代码时特别有用。

如果您需要使用其他程序包中的许多函数，这将相当冗长。还有一个与 :: 相关的轻微性能损失（大约为 5µs，因此只有当您调用函数数百万次时才会有影响）。您将在 namespace imports 中了解调用其他程序包中函数的其他方法。

- Suggests：您的程序包可以使用这些程序包，但并不需要它们。您可以使用建议的程序包（例如数据集）来运行测试、构建 vignette，或者可能只有一个函数需要该包。

  Suggests 中列出的程序包不会与程序包一起自动安装。这意味着您需要在使用包之前检查它是否可用（使用 requireNamespace(x, quietly = TRUE)）。有两种基本情况：

```
# You need the suggested package for this function
my_fun <- function(a, b) {
    if (!requireNamespace("pkg", quietly = TRUE)) {
        stop("Package \"pkg\" needed for this function to work. Please␣
↪install it.",
        call. = FALSE)
    }
}

# There's a fallback method if the package isn't available
my_fun <- function(a, b) {
    if (requireNamespace("pkg", quietly = TRUE)) {
        pkg::f()
    } else {
        g()
    }
}
```

在本地开发程序包时，永远不需要使用 Suggests。发布程序包时，使用 Suggests 是对用户的礼貌。它使用户避免下载很少需要的程序包，并让他们尽快开始使用您的程序包。

将 Imports 和 Suggests 添加到程序包中的最简单方法是使用 usethis::use_package()。这会自动将它们放在 DESCRIPTION 中的正确位置，并提醒您如何使用它们。

```
usethis::use_package("dplyr") # Defaults to imports
#> Adding dplyr to Imports
#> Refer to functions with dplyr::fun()
usethis::use_package("dplyr", "Suggests")
#> Adding dplyr to Suggests
#> Use requireNamespace("dplyr", quietly = TRUE) to test if package is
#>  installed, then use dplyr::fun() to refer to functions.
```

7.1.1 版本控制

如果需要程序包的特定版本，请在其名称后的括号中指定：

```
Imports:
    ggvis (>= 0.2),
    dplyr (>= 0.3.0.1)
Suggests:
    MASS (>= 7.3.0)
```

您应该总是希望指定最小的版本，而不是精确的版本（`MASS (== 7.3.0)`）。因为 R 不能同时加载同一个包的多个版本，所以指定一个确切的依赖关系会大大增加版本冲突的可能性。

当您发布包时，版本控制是最重要的。通常情况下，人们安装的程序包版本与您不完全相同。如果有人有一个旧的包，它没有您的包需要的功能，那么他们会得到一个无用的错误消息。但是，如果您提供版本号，他们会收到一条错误消息，告诉他们问题的确切原因：一个低版本的程序包。

如果您声明了一个最低版本，请仔细考虑。在某种意义上，最安全的做法是要求版本大于或等于程序包的当前版本。对于合作工作，最自然的就是定义为程序包的当前 CRAN 版本；私人或个人项目可以采用其他一些惯例。但是重要的是要意识到对哪些尝试安装您的程序包的人的影响：如果他们的本地安装不能满足您对版本的所有要求，那么安装要么失败要么强制升级这些依赖项。如果您的最低版本要求是真实需要的，那么这是可取的，即如果不这样做，您的程序包将被破坏。但是，如果您所说的版本需求并没有扎实的理由，那么这可能是不必要的保守并造成不方便。

在没有明确的硬性要求的情况下，您应该根据您预期的用户基础、他们可能拥有的程序包版本以及过于宽松与过于保守的成本效益分析来设置最低版本（或不设置）。

TODO: return here and insert concrete details closer to publication. Thing 1: mention relevant usethis function(s). Today that is ‘‘use_tidy_version()‘‘n but it needs some work (https://github.com/r-lib/usethis/issues/771). Thing 2: See if Kirill’ s prototype of studying minimum version has matured into anything we can recommend for general use (https://github.com/r-lib/pillar/tree/c017f20476fce431ceee18cc3a637f7ed2884d3a/minver#readme)

7.1.2 其他依赖项

另外还有三个字段允许您表达更专业的依赖关系：

- `Depends`：在 R 2.14.0 推出命名空间（namespaces）之前，`Depends` 是“依赖”另一个包的唯一方法。现在，不管名字是什么，您应该总是使用 `Imports`，而不是依赖 `Depends`。您将了解在 namespaces 中为什么以及何时仍应使用 `Depends`。

  > 您还可以使用 `Depends` 来要求使用 R 的特定版本，例如 `Depends: R (>=3.4.0)`。同样，如果您这样做，请仔细考虑。这与为依赖的程序包设置最低版本是一样的问题，只是涉及到 R 本身的风险要高得多。用户不能简单地同意必要的升级，因此，如果其他程序包依赖于您的程序包，您对 R 的最低版本要求可能会导致一系列程序包安装的失败。

  - 如果您想使用 `trimws()` 之类的函数，backports 程序包 非常有用，它是在 3.3.0 版本中引入的，同时仍然支持较旧的 R 版本。
  - tidyverse 正式支持当前的 R 版本、devel 版本和四个以前的版本。我们在用于持续集成的标准构建矩阵中主动测试了这种支持。
  - 使用级别较低的程序包可能不需要这种严格级别。主要的收获是：如果您规定了一个最低限度的版本，您应该有令人信服的理由，并且应该采取合理的措施，以定期测试您的要求。

- `LinkingTo`：这里列出的程序包依赖于另一个包中的 C 或 C++ 代码。您将在 compiled code 中了解有关 `LinkingTo` 的更多信息。
- `Enhances`：这里列出的程序包由您的程序包 “增强”。通常，这意味着您为在另一个包中定义的类提供方法（与 `Suggests` 相反）。但是很难定义这意味着什么，所以我不建议使用 `Enhances`。

您需要的在 R 之外的东西也可以在 `SystemRequirements` 字段中列出。但这只是一个纯文本字段，不会自动检查。可以把它当作一个快速参考；您还需要在 README 中写入详细的系统需求（以及如何安装它们）。

3.11.2 7.2 标题和描述：您的程序包能够做什么？

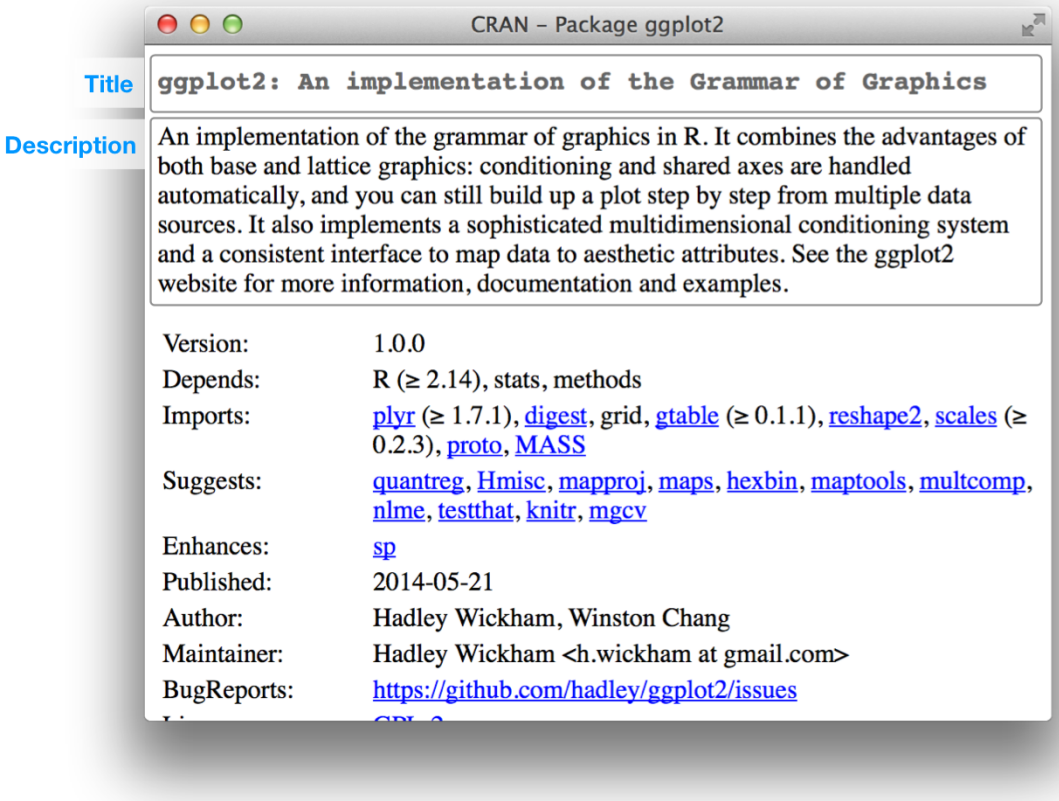
title 和 description 字段描述程序包的功能。它们只是在长度上有所不同：

- `Title` 是程序包的一行描述，通常显示在程序包列表中。它应该是纯文本（无标记），像标题一样大写，**不能**以句点结尾。让它保持简短：程序包列表通常会将标题截断为 65 个字符。
- `DESCRIPTION` 比 title 更详细。可以用多个句子，但是只能用一个段落。如果你的描述跨越多行（而且它应该有多行！），每行宽度不得超过 80 个字符。用 4 个空格缩进后继续一行。

ggplot2 的 `Title`` 和 `DESCRIPTION` 为：

```
Title: An implementation of the Grammar of Graphics
Description: An implementation of the grammar of graphics in R. It combines
    the advantages of both base and lattice graphics: conditioning and shared
    axes are handled automatically, and you can still build up a plot step
    by step from multiple data sources. It also implements a sophisticated
    multidimensional conditioning system and a consistent interface to map
    data to aesthetic attributes. See the ggplot2 website for more information,
    documentation and examples.
```

一个好的标题和描述是很重要的，特别是如果计划将程序包发布到 CRAN，因为它们出现在 CRAN 下载页面上，如下所示：



因为 `Description` 只给了您一小部分空间来描述程序包的功能，所以我还建议包含一个 `README.md` 文件，它深入到更深入程序包的地方，并展示一些示例。您将在 README.md中了解到这些。

3.11.3 7.3 作者：您是谁？

要确定程序包的作者，以及出错时与谁联系，请使用 `Authors@R` 字段。这个字段并不普通，因为它包含可执行的 R 代码而不是纯文本。下面是一个例子：

```
Authors@R: person("Hadley", "Wickham", email = "hadley@rstudio.com",
    role = c("aut", "cre"))
```

```
person("Hadley", "Wickham", email = "hadley@rstudio.com",
role = c("aut", "cre"))
#> [1] "Hadley Wickham <hadley@rstudio.com> [aut, cre]"
```

这个命令表示作者（aut）和维护者（cre）都是 Hadley Wickham，他的电子邮件地址是 `hadley@rstudio.com`。`person()` 函数有四个主要参数：

- 由前两个参数，`given` 和“family“（它们通常由位置而不是名称提供）指定的名称。在英国文化中，名在姓之前。在许多文化中，这种习俗并不成立。

- 电子邮件地址 email。
- 指定角色 role 的三个字母组成的代码。有四个重要角色：
  - cre：创建者或维护者，如果您有麻烦，您应该向其请求帮助。
  - aut：作者，那些对这个程序包做出重大贡献的人。
  - ctb：贡献者，那些贡献较小的人，比如贡献了补丁。
  - cph：版权所有人。如果版权由作者以外的人（通常是公司，即作者的雇主）持有，则使用此选项。

(完整的角色列表非常全面。如果您的软件包中有一个 woodcutter（“wdc”）、liricist（“lyr”）或 custome designer（“cst”），请放心，您可以正确描述他们在创建程序包中的作用（角色）。)

如果需要添加进一步的说明，也可以使用 comment 参数并以纯文本形式提供所需的信息。

可以使用 c() 列出多个作者：

```
Authors@R: c(
    person("Hadley", "Wickham", email = "hadley@rstudio.com", role = "cre"),
    person("Winston", "Chang", email = "winston@rstudio.com", role = "aut"))
```

每个程序包必须至少有一个作者（aut）和一个维护者（cre）（他们可能是同一个人）。维护者必须有一个电子邮件地址。这些字段用于生成程序包的基本引文（例如 citation("pkgname")）。只有被列为作者的人才会包含在自动生成的引文中。如果包含其他人编写的代码，还有一些额外的细节。因为这通常发生在封装 C 库时，所以在 compiled code 中讨论过。

除了您的电子邮件地址外，列出其他有帮助的资源也是一个好主意。您可以在 URL 中列出 URLs。多个 URLs 用逗号分隔。BugReports 是提交 BUG 报告的 URL。例如，knitr 有：

```
URL: https://yihui.name/knitr/
BugReports: https://github.com/yihui/knitr/issues
```

您还可以使用单独的 Maintainer 和 Author 字段。我不喜欢使用这些字段，因为 Authors@R 能够提供更丰富的元数据。

7.3.1 在 CRAN 上

最重要的是，您的电子邮件地址（即 cre 的地址）是 CRAN 将用来提供关于您的程序包相关信息的电子邮箱地址。所以确保使用的电子邮件地址将会存在一段时间。此外，由于该地址将用于自动邮寄，因此 CRAN 政策要求该地址只能用于一个人（而不是邮件列表），并且不需要任何确认或使用任何筛选。

3.11.4 7.4 许可证：谁可以使用您的程序包？

License 字段可以是开源许可证（如 GPL-2 或 BSD）的标准缩写，也可以是指向包含更多信息的文件（file LICENSE）的指针。只有当您计划发布您的程序包时，许可证才是真正重要的。如果没有这个打算，则可以

忽略此部分。如果您想明确程序包是不开源的，使用 `License: file LICENSE` 创建一个叫做 LICENSE 的文件，可以包含以下内容：

```
Proprietary

Do not distribute outside of Widgets Incorporated.
```

开源软件许可是一个丰富而复杂的领域。幸运的是，在我看来，对于您的 R 包，只有三个许可证需要考虑：

- MIT （类似 BSD 2-Clause 和 BSD 3-Clause 许可证）。这是一个简单的许可证。它允许人们使用和自由分发您的代码，但只有一个限制：许可证必须始终与代码一起分发。

  MIT 许可证是一个“模板”，因此如果您使用它，您需要 `License: MIT + file LICENSE`，以及如下所示的许可证文件：

  ```
  YEAR: <Year or years when changes have been made>
  COPYRIGHT HOLDER: <Name of the copyright holder>
  ```

- GPL-2 或 GPL-3。这些是“复制遗留”许可证。这意味着，以捆绑包形式分发您的代码的任何人都必须以 GPL 兼容方式作为捆绑包的许可。此外，任何分发代码修改版本（衍生作品）的人也必须开源他们的源代码。GPL-3 比 GPL-2 严格一点，弥补了一些老漏洞。

- CC0。它放弃您对代码和数据的所有权利，以便任何人都可以出于任何目的自由使用它。这有时被称为公共领域，这一术语在所有国家既没有明确定义也没有意义。

  此许可证最适合于数据包。至少在美国，数据是不受版权保护的，所以您不会放弃太多。这张许可证正好说明了这一点。

如果您想了解更多有关其他常用许可证的信息，查阅 Github 的 choosealicense.com 是个好的开始。另一个好的资源是 https://tldrlegal.com/，它解释了每个许可证的最重要部分。如果您使用的许可证不是我建议的三种许可证，请务必参考“Writing R Extensions”关于 licensing 的部分。

如果您的软件包中包含不是您编写的代码，则需要确保您的使用符合它的许可证。由于这通常发生在包含 C 源代码时，在 compiled code 中会更详细地讨论它。

7.4.1 在 CRAN 上

如果您想把您的程序包提交到 CRAN，您必须选择一个标准许可证。否则，CRAN 很难确定分发您的程序包是否合法！您可以找到 CRAN 认为有效的许可证的完整列表 https://svn.r-project.org/r/trunk/share/licenses/license.db。

3.11.5 7.5 版本

从形式上讲，R 程序包版本是由至少两个被 `.` 或 `-` 分隔的整数组成的序列。例如，`1.0` 和 `0.9.1-10` 是有效的版本号，但 `1` 或 `1.0-devel` 不是。您可以使用 `numeric_version` 解析版本号。

```
numeric_version("1.9") == numeric_version("1.9.0")
#> [1] TRUE
numeric_version("1.9.0") < numeric_version("1.10.0")
#> [1] TRUE
```

例如，程序包的版本可能是 1.9。R 认为此版本号与 1.9.0 相同，低于 1.9.2，且所有这些版本都低于 1.10（即“一点十”，而不是“一点一零”）。R 使用版本号来确定是否满足程序包依赖关系。例如，程序包可能会导入 `devtools (>=1.9.2)`，在这种情况下，版本 1.9 或 1.9.0 将无法工作。

程序包的版本号随着程序包的后续版本而增加，但它不仅仅是一个递增的计数器——每个版本的版本号更改的方式可以传达有关程序包中的更改类型的信息。

我不建议充分利用 R 的灵活性。而建议始终使用 . 分开版本号。

- 发布的版本号由三个数字组成，`<major>.<minor>.<patch>`。对于版本号 1.9.2，1 是主版本号，9 是次版本号，2 是补丁号。永远不要使用像 `1.0` 这样的版本，而应该总是详细说明 `1.0.0` 这三个组件。
- 开发中的程序包有第四个组件：开发版本。它从 9000 开始。例如，程序包的第一个版本应该是 `0.0.0.9000`。这项建议有两个原因：第一，可以很容易地看到程序包是发布的还是正在开发，第四个位置的使用意味着你不局限于下一个版本是什么。`0.0.1`、`0.1.0` 和 `1.0.0` 均优于 `0.0.0.9000`。

  增加开发版本号，例如，如果您添加了另一个开发程序包需要依赖的重要特性，则从 `9000` 增加到 `9001`。

  如果您使用 svn，那么可以嵌入顺序修订标识符，而不一定要使用 `9000`。

这里的建议部分来自 Semantic Versioning 和 X.Org 版本控制方案。如果您想了解更多关于许多开源项目所使用的版本控制标准，请阅读它们。

在发布程序包的介绍，即 picking a version number 中，我们将回到版本号的说明，选择一个版本号。现在，只需记住第一个版本号应该是 `0.0.0.9000`。

3.11.6 7.6 其他组件

本书其他地方描述了许多其他的字段：

- `Collate` 控制 R 文件的溯源顺序。这只在代码有副作用的情况下才重要；最常见的是因为您使用的是 S4。这在 documenting S4 中有更深入的描述。
- `LazyData` 使您更容易访问程序包中的数据。因为它非常重要，所以它包含在 devtools 创建的最小的描述文件中。在 external data 中有更详细的描述。

实际上还有许多其他很少使用的字段。完整的列表可以在 R extensions manual “The DESCRIPTION file” 部分找到。您也可以创建自己的字段来添加其他元数据。唯一的限制是您不应该使用现有的名字，而且如果打算提交给 CRAN，那么使用的名字应该是有效的英语单词（因此不会生成拼写检查记录）。

3.12 第九章 Vignettes：长篇文档

Vignette(使用指南) 是您的程序包的一篇长篇指南。如果您知道所需函数的名称，那么函数文档是便于使用的，但是在其他情况下就很难使用了。Vignette 就像一本书的一章或一篇学术论文：它可以描述你的程序包要解决的问题，然后向读者展示如何解决它。Vignette 应该将全部函数划分为多个清晰有用的类别，并演示如何协调使用多个函数来解决问题。如果想解释程序包的细节，vignette 也很有用。例如，如果实现了一个复杂的统计算法，那么您可能需要在一个 vignette 中描述算法的所有细节，这样程序包用户就可以了解到底发生了什么，并确信您的算法实现是正确的。

许多现有的程序包都有 vignettes。您可以使用 `browseVignettes()` 查看所有已安装包的 vignettes。要查看特定包的 vignette，请使用参数 `browseVignettes("packagename")`。\ 每个 `vignette` 提供了三个文档：原始源文件、可读的 `HTML` 页面或 `PDF`，以及一个 `R` 代码文件。可以使用 ``vignette(x)` 读取特定的 vignette，并使用 `edit(vignette(x))` 查看其代码。要查看尚未安装的包的 vignette，请查看其 CRAN 页面，例如: https://cran.r-project.org/web/packages/dplyr。

在 R 3.0.0 之前，只有使用 Sweave 才能够创建 vignette。这很具有挑战性，因为 Sweave 只使用 LaTeX，而且 LaTeX 既难学又难编译。现在，任何包都可以提供一个 vignette **引擎**，一个将输入文件转换成 HTML 或 PDF vignette 的标准接口。在本章中，我们将使用 knitr 提供的 R Markdown vignette 引擎。我推荐这款引擎是因为：

- 您将使用 Markdown 写作，它是一个纯文本格式系统。与 LaTeX 相比，Markdown 的功能是有限的，但是这个限制是很好的，因为它迫使你专注于内容。
- 它可以混合文本、代码和结果（不论是文本还是图像）。
- rmarkdown package 进一步简化了您的工作，它通过使用 pandoc 将 Markdown 转换为 HTML 并提供许多有用的模板来协调 Markdown 和 knitr。

从 Sweave 切换到 R Markdown 对我使用 vignette 产生了深远的影响。以前，由于制作一个 vignette 是缓慢而痛苦的，因此我很少动手。现在，vignette 是我的程序包中必不可少的一部分。每当我需要解释一个复杂的话题，或演示如何用多个步骤解决问题时，我都会用到它们。

目前，获得 R Markdown 的最简单方法是使用 RStudio。RStudio 将自动安装所有需要的依赖项。如果不使用 RStudio，则需要：

- 通过 `isntall.packages("rmarkdown")` 安装 rmarkdown 程序包。
- 安装 pandoc。

3.12.1 9.1 Vignette 工作流程

运行以下代码来创建您的第一个 vignette：

```
usethis::use_vignette("my-vignette")
```

这将会：

1. 创建一个 `vignettes/` 目录。
2. 向 `DESCRIPTION` 添加必要的依赖项（即，它向 `Suggests` 和 `VignetteBuilder` 字段添加字段值 knitr）。
3. 起草一个 vignette，`vignette/my-vignette.Rmd`。

草稿的设计是为了提醒你 R 降价文件的重要部分。当你创建一个新的小插曲时，它可以作为一个有用的参考。

一旦有了这个文件，工作流程就变得很简单：

- 修改 vignette。
- 按 Ctrl/Cmd + Shift + K（或单击 

）来生成文档并预览输出。

R Markdown vignette 有三个重要的组成部分：

- 初始元数据文本块（metadata）。
- 设置文本格式的 Markdown。
- 混合文本、代码和结果的 knitr。

这些将在接下来的各节中进行描述。

3.12.2 9.2 元数据

Vignette 的前几行包含重要的元数据。默认模板包含以下信息：

```
---
title: "Vignette Title"
output: rmarkdown::html_vignette
vignette: >
%\VignetteIndexEntry{Vignette Title}
%\VignetteEngine{knitr::rmarkdown}
\usepackage[utf8]{inputenc}
---
```

这个元数据是用 yaml 编写的，yaml 是一种设计成既可供人阅读也可由计算机读取的格式。语法的基本原理与 `DESCRIPTION` 文件非常相似，其中每行由字段名、冒号和字段值组成。我们在这里使用的一个特殊的 YAML 特性是 `>`。它表示以下几行文本是纯文本，不应该使用任何特殊的 YAML 特性。

字段包括：

- 标题、作者和日期：这是您放置 vignette 的标题、作者和日期的地方。您需要自己填写这些内容（如果不想在页面顶部显示标题栏，可以删除它们）。默认情况下会自动填写日期：它使用特殊的 knitr 语法（如下所述）插入今天的日期。

- Output：它会告诉 rmarkdown 使用哪个输出格式化程序。有许多选项对常规报表有用（包括 html、pdf、slideshows 等），但 `rmarkdown::html_vignette` 经过了专门设计，可以在程序包中正常工作。使用 `?rmarkdown::html_vignette` 获取更多详细信息。
- Vignette：这包含 R 所需的一个特殊的元数据文本块。在这里，您可以看到 LaTeX Vignette 的遗产：元数据看起来像 LaTeX 命令。您需要修改 `\VignetteIndexEntry` 以提供希望在 vignette 索引中显示的 vignette 文档标题。其余两行保持原样。它们告诉 R 使用 knitr 来处理文件，并且文件是用 UTF-8 编码的（这是编写 vignette 时应该使用的唯一编码）。

3.12.3 9.3 Markdown

R Markdown vignette 是用 Markdown 编写的，它是一种轻量级的标记语言。Markdown 的作者 John Gruber 总结了 Markdown 的目标和哲学：

> Markdown 的目标是尽可能地容易读写。
>
> 然而，可读性是最重要的。Markdown 格式的文档应该可以原样发布，就像纯文本一样，而不是看起来像是用标签或格式化说明标记的。Markdown 的语法受到了一些现有的 text-to-HTML 过滤器的影响，包括 Setext、atx、Textile、reStructuredText、Grutatext 和 EtText ——Markdown 语法的最大灵感来源是纯文本电子邮件的格式。
>
> 为此，Markdown 的语法完全由标点符号组成，这些标点符号是经过精心选择的，以便看起来像它们本身的意思。例如，一个单词周围的星号实际上看起来像是强调。Markdown 列表看起来像，emmm，列表。即使是方块引号也像是被引用的文本段落，如果你曾经使用过电子邮件的话。

Markdown 并不像 LaTeX、reStructuredText 或 docbook 那样强大，但它很简单，易于编写，甚至在没有渲染时也易于阅读。我发现 Markdown 的约束对写作很有帮助，因为它让我专注于内容，并防止我在样式设计上搞得一团糟

如果您从未使用过 Markdown，那么可以从 John Gruber 的 Markdown 语法文档 开始。Pandoc 的 Markdown 实现消除了一些粗糙的功能实现，并添加了一些新特性，因此我还建议您熟悉 pandoc readme。编辑 Markdown 文档时，RStudio 通过问号图标显示一个下拉菜单，该图标提供了一个 Markdown 参考卡片。

下面的各节向您展示了我认为 pandoc 的 Markdown 的最重要的特点。您应该能在 15 分钟内学会基本知识

9.3.1 小节（Sections）

标题由 `#` 标识：

```
# Heading 1
## Heading 2
### Heading 3
```

创建带有三个或更多连字符（或星号）的水平线：

```
--------
********
```

9.3.2 列表（Lists）

基础无序列表使用 *：

```
* Bulleted list
* Item 2
    * Nested bullets need a 4-space indent.
    * Item 2b
```

9.3.3 行内格式（Inline formating）

行内格式也很简单：

```
_italic_ or *italic*
__bold__ or **bold**
[link text](destination)
<http://this-is-a-raw-url.com>
```

9.3.4 表格（Tables）

有四种类型的表。我建议使用如下所示的管道符构成的表：

```
| Right | Left | Default | Center |
|------:|:-----|---------|:------:|
|   12  |  12  |    12   |    12  |
|  123  |  123 |   123   |   123  |
|    1  |    1 |     1   |     1  |
```

注意：在标题下的分隔符中使用了 :。这决定了列的对齐方式。

如果表的底层数据存在于 R 中，请不要手动布局。相反，请使用 `knitr::kable()`，或查看 printr 或 pander。

9.3.5 代码（code）

行内代码使用 `code`

对于较大的代码块，使用 ```。这些被称为“围墙”代码块：

```
```
# A comment
add <- function(a, b) a + b
```
```

要向代码添加语法高亮显示，请在反引号后加上语言的名称：

```
```c
int add(int a, int b) {
return a + b;
}
```
```

在打印时，pandoc 支持的语言有：actionscript、ada、apache、asn1、asp、awk、bash、bibtex、boo、c、changelog、clojure、cmake、coffee、coldfusion、commonlisp、cpp、cs、css、curry、d、diff、djangtemplate、doxygenlua、dtd、eiffel、email、erlang、fortran、fsharp、gnuassembler、go、haxe、html、ini、java，javadoc、javascript、json、jsp、julia、latex、lex、literatecurry、literatehaskell、lua、makefile、mandoc、matlab、maxima、metafont、mips、modula2、modula3、monobasic、nasm、noweb、objectivec、objectivecpp、ocaml、ocave、pascal、perl、php、pike、postscript、prolog、python、r、relaxcompact、rhtml、ruby、rust、scala、scheme、sci、sed、sgml、sql，sqlmysql、sqlpostgresql、tcl、texinfo、verilog、vhdl、xml、xorg、xslt、xul、yacc、yaml。语法高亮显示是由 haskell 包的 highlighting-kate 完成的；有关当前列表，请访问网站。）

当您在 vignette 中添加 R 代码时，通常不会使用 ```r，而是使用由 knitr 专门处理的 ```{r}，如下所述。

3.12.4 9.4 Knitr

Knitr 允许您混合代码、结果和文本。Knitr 获取 R 代码，运行它，捕获输出，并将其转换为格式化的 Markdown。Knitr 捕获所有打印的输出、消息、警告、错误（可选）和绘图（basic graphics, lattice & ggplot 等）。

考虑下面这个简单的例子。请注意，knitr 块看起来类似于 fenced 代码块，但不是使用 `r`，而是使用 `{r}`。

```
```{r}
# Add two numbers together
add <- function(a, b) a + b
add(10, 20)
```
```

这将生成以下 Markdown 代码：

```
```r
# Add two numbers together
```

（下页继续）
```

(续上页)

```
add <- function(a, b) a + b
add(10, 20)
## [1] 30
```

这些会被渲染成为：

```
# Add two numbers together
add <- function(a, b) a + b
add(10, 20)
## 30
```

一旦您开始使用 knitr，就再也不用返回检查了。因为您的代码总是在构建 vignette 时运行，所以您可以放心地知道所有代码都可以工作。您的输入和输出不可能不同步。

9.4.1 选项

您可以指定附加选项来控制渲染过程：

- 要影响单个块（block），请添加块（block）设置：

```
```{r, opt1 = val1, opt2 = val2}
# code
```
```

- 要影响全部的块（block），在 knitr block 中调用 `knitr::opts_chunk$set()`：

```
```{r, echo = FALSE}
knitr::opts_chunk$set(
    opt1 = val1,
    opt2 = val2
)
```
```

最重要的选项如下所示：

- `eval = FALSE` 阻止代码的运行。如果您想显示一些需要很长时间才能运行的代码，这很有用。使用这个选项时要小心：因为代码没有运行，所以很容易引入 BUG。（另外，当用户复制粘贴代码时，如果代码不起作用，他们会感到困惑不解。）
- `echo = FALSE` 关闭代码 输入的打印（输出仍将被打印）。一般来说，您不应该在 vignette 中使用它，因为理解代码在做什么很重要。它在编写报告时更有用，因为代码通常比输出不那么重要。
- `results = "hide"` 关闭代码 输出的打印。

- `warning = FALSE` 和 `message = FALSE` 禁止显示警告和消息。
- `error = TRUE` 捕获块中的所有错误并以内联的方式显示它们。如果您想演示如果代码抛出错误会发生什么，这是很有用的。无论何时使用 `error = TRUE`，都需要使用 `purl = FALSE`。这是因为每个 vignette 都有一个包含 vignette 中所有代码的脚本文件。R 必须在不出错的情况下生成源文件，而 `purl = FALSE` 会阻止出错的代码插入到该文档中。
- `collapse = TRUE` 和 `comment = "#>"` 是我显示代码输出的首选方式。我通常通过在文档开头放置以下 knitr 块来全局设置它们。

```
```{r, echo = FALSE}
knitr::opts_chunk$set(collapse = TRUE, comment = "#>")
```
```

- `results = "asis"` 将 R 代码的输出视为 Markdown 文本。如果要从 R 代码生成文本，这很有用。例如，如果要使用 pander 包生成一个表格，可以执行以下操作：

```
```{r, results = "asis"}
pander::pandoc.table(iris[1:3, 1:4])
```
```

生成的 Markdown 表格如下所示：

```
------------------------------------------------------------
Sepal.Length   Sepal.Width   Petal.Length   Petal.Width
-------------- ------------- -------------- -------------
    5.1            3.5           1.4            0.2

    4.9             3            1.4            0.2

    4.7            3.2           1.3            0.2
------------------------------------------------------------
```

这会产生像这样的表格：

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
5.1	3.5	1.4	0.2
4.9	3	1.4	0.2
4.7	3.2	1.3	0.2

- `fig.show = "hold"` 保存所有数字，直到代码块结束。
- `fig.width = 5` 和 `fig.height = 5` 设置图形的高度和宽度（以英寸为单位）。

更多的其他选项在 https://yihui.name/knitr/options。

3.12.5 9.5 开发周期

使用 Cmd + Alt + C 一次运行一个代码块。在一个新的 R Session 中使用 Knit (Ctrl/Cmd + Shift + K) 重新运行整个文档。

您可以使用 `devtools::build_vignettes()` 从控制台构建所有 vignette，但这很少使用。相反，使用 `devtools::build()` 创建包含 vignette 的二进制包。RStudio 的 "Build & reload" 通过放弃构建 vignette 来节省时间。类似的，`devtools::install_github()`` (和类似的函数) 默认情况下不会构建 `vignette`, \ 因为它们很耗时，可能需要额外的程序包。您可以使用 ``devtools::install_github(build_vignettes = TRUE)` 强制构建。这也将安装所有建议的程序包。

3.12.6 9.6 有关撰写 vigntte 的建议

> If you' re thinking without writing, you only think you' re thinking. —Leslie Lamport

写一个 Vignette 时，您是在教别人如何使用您的程序包。因此需要设身处地为读者着想，采取一种 "初学者的思维"。这可能很困难，因为很难忘记已经内化的所有知识。出于这个原因，我发现亲自授课是一种非常有用的方法，可以获得对我的 Vignette 的反馈。您不仅可以直接得到反馈，而且还可以更容易地了解人们已经知道的东西。

这种方法的一个有用的作用是它可以帮助您改进代码。它迫使您重新审视最初的思考流程，并意识到什么部分是困难的。每次我写文章来描述最初的经历，我都意识到我错过了一些重要的功能。添加这些功能不仅对我的用户有帮助，而且还经常帮助我！(这是我喜欢写书的原因之一)。

- 我强烈推荐 Kathy Sierra 写的任何文章。她以前的博客 Creating passionate users 中充满了关于编程、教学和如何创建有价值的工具的建议。我建议您通读所有旧的内容。她的新博客 Serious Pony 没有那么多内容，但仍然有一些优秀的文章。
- 如果您想学习如何写得更好，我强烈推荐 Joseph M. Williams 和 Joseph Bizup 的 Style: Lessons in Clarity and Grace 。它可以帮助理解写作的结构，这样就能更好地识别和纠正糟糕的写作。

写一个 vignette 也可以让你从编程中得到片刻的休息。根据我的经验，与编程不同，写作使用了大脑中不同的部分，所以如果你厌倦了编程，那就试着写一点文字。(这与结构化拖延有关)。

9.6.1 组织

对于更加简单的程序包，一个 vignette 通常就足够了。但是对于更复杂的程序包，您可能需要不止一个。事实上，您可以有任何喜欢的 vignette。我倾向于把它们看作一本书的章节——它们应该是自成体系的，但仍然链接在一起成为一个连贯的整体。

虽然这只是一个小的改动，但是您可以通过利用文件在磁盘上的存储方式来链接各种 vignette：为了链接到 vignette `abc.Rmd`，只需产生一个链接到 `abc.html`.

3.12.7 9.7 CRAN 注记

注意，由于您在本地构建了 vignette，所以 CRAN 只接收 html/pdf 和源代码。然而，CRAN 并不会重建这个 vignette。它只检查代码是否可以运行（通过运行它）。这意味着 vignette 使用的任何程序包都必须在 `DESCRIPTION` 中声明。但这也意味着即使 CRAN 没有安装 pandoc，您也可以使用 Rmarkdown（它使用 pandoc）。

常见的问题：

- Vignette 是以交互方式构建的，但是在检查时会失败，并显示有关您知道已安装的缺失程序包的错误。这意味着您忘记在 `DESCRIPTION` 中声明该依赖关系（通常应该在 `Suggests` 字段中声明）。
- 所有事情都是交互式工作的，但是在安装了程序包之后，vignette 却不出现了。这可能发生了以下几种情况之一。首先，由于 RStudio 的 “build & reload” 并没有构建 vignette，所以您可能需要运行 `devtools::install()` 。接下来检查：
  1. 目录名称为 `vignettes/` 而不是 `vignette/`。
  2. 检查您是否无意中使用了 `.Rbuildignore` 排除了 vignette。
  3. 确保您写入了必要的 vignette 元数据。
- 如果您使用了 `error = TRUE`，那么必须使用 `purl = FALSE`。

您需要注意文件的大小。如果包含了大量的图片，则很容易创建一个非常大的文件。虽然没有硬性的规定，但是如果有一个非常大的 vignette，就得准备好要么证明文件大小是正确的，要么使其变小。

3.12.8 9.8 接下来是什么

如果您想对您的 vignette 的外观有更多的控制，需要学习更多关于 Rmarkdown 的知识。网站 https://rmarkdown.rstudio.com 是一个很好的起点。在那里，您可以了解其他输出格式（如 LaTeX 和 pdf）以及如何在需要额外控制的情况下合并原始 HTML 和 LaTeX。

如果您写了一篇不错的 vignette，考虑把它提交给 *Journal of Statistical Software* 或 *The R Journal*。这两种期刊都是电子版的，并经过同行评审。评论者的评论对提高您的 vignette 和相关软件的质量非常有帮助。

3.13 第十二章外部数据

在程序包中包含数据通常很有用。如果您要向广大的用户发布程序包，那么这就是一种为程序包的功能提供引人注目的使用示例的方法。如果您要将软件包发布给更加特定的受众，他们对数据（例如新西兰人口普查数据）或主题（例如人口统计学）感兴趣，那么这将是一种将数据连同文档一起分发的方法（只要您的受众是 R 用户）。

在程序包中包含数据的主要方法有三种，这取决于您希望如何处理数据以及谁能够使用它：

- 如果要存储二进制数据并使其对用户可用，请将其放在 `data/` 中。这是放置示例数据集的最佳位置。

- 如果您想存储已解析的数据，但不想让用户使用它，请将其放在 R/sysdata.rda 中。这是放置函数所需数据的最佳位置。
- 如果要存储原始数据，请将其放在 inst/extdata 中。

这三个方法的一个简单替代方法是将其包含在程序包的源代码中，要么手动创建，要么使用 dput() 将现有数据集序列化为 R 代码。

下面将更详细地描述每个可能存放数据的位置。

3.13.1 12.1 导出的数据

程序包数据最常见的存放位置是（震惊！）"data/。此目录中的每个文件都应该是由 ``save() 创建的 .RData``文件，\ 其中包含一个对象（与文件同名）。遵守这些规则的最简单方法是使用 ``usethis::use_data()：

```
x <- sample(1000)
usethis::use_data(x, mtcars)
```

当然也可以使用其他类型的文件，但我不建议这样做，因为 .RData 文件已经很快、很小而且很明确。其他可选的类型在 data() 中描述。对于较大的数据集，您可能需要尝试使用压缩设置。默认值是 bzip2，但有时 gzip 或 xz 可以创建更小的文件。

如果 DESCRIPTION 包含 LazyData:true，则数据集将被延迟加载。这意味着在你使用它们之前它们不会占用任何内存。下面的示例显示加载 nycflights13 程序包前后的内存使用情况。您可以看到，在检查存储在包中的 flights 数据集之前，内存使用情况不会发生变化。

```
pryr::mem_used()
#> Registered S3 method overwritten by 'pryr':
#>   method      from
#>   print.bytes Rcpp
#> 41.1 MB
library(nycflights13)
pryr::mem_used()
#> 44.3 MB

invisible(flights)
pryr::mem_used()
#> 85 MB
```

我建议您在 DESCRIPTION 中始终包含 LazyData:true。usethis::create_package() 可以为您执行此操作。

通常，您在 data/ 中包含的数据是从其他地方收集的原始数据的已处理版本。我强烈建议您花时间在程序包的源代码中包含用于处理数据的代码。这将使您更容易更新或再生产您的数据版本。我建议您将此代码放在

data-raw/ 中。在程序包的捆绑版本中不需要它，所以也要将其添加到 .Rbuildignore。使用以下代码一步完成所有这些操作：

```
usethis::use_data_raw()
```

您可以在我最近的一些数据包中看到这种方法的实际应用。我将这些作为程序包创建，因为数据很少会更改，而且多个车呢光绪包可以使用它们作为示例：

- babynames
- fueleconomy
- nasaweather
- nycflights13
- usdanutrients

12.1.1 添加数据集说明

data/ 中的对象总是有效地被导出（它们使用与 NAMEDPACE 稍有不同的机制，但细节并不重要）。这意味着它们必须拥有文档。为数据撰写文档就像给函数撰写文档一样，只是有一些细微的差别。与直接在数据中撰写相关文档不同，您为数据集的名称撰写文档并将其保存在 R/ 中。例如，用于记录 ggplot2 中钻石数据集的 roxygen2 代码块保存为 R/data.R，如下所示：

```
#' Prices of 50,000 round cut diamonds.
#'
#' A dataset containing the prices and other attributes of almost 54,000
#' diamonds.
#'
#' @format A data frame with 53940 rows and 10 variables:
#' \describe{
#'   \item{price}{price, in US dollars}
#'   \item{carat}{weight of the diamond, in carats}
#'   ...
#' }
#' @source \url{http://www.diamondse.info/}
"diamonds"
```

对于数据集文档，还有两个重要的附加标记：

- @format 提供了数据集的概述。对于数据框，您应该包括一个定义列表来描述每个变量。在这里描述变量的单位通常是个好主意。
- @source 提供了数据的来源详细信息，通常放在 \url{}。

从不使用 @export 导出数据集。

3.13.2 12.2 内部数据

有时函数需要预先计算好的数据表。如果你把它们放在 data/ 中，它们也可以被程序包用户使用，这是不合适的。相反，您可以将它们保存在 R/sysdata.rda. 例如，两个与颜色相关的程序包，munsell 和 dichromat，使用 R/sysdata.rda 存储大量的彩色数据表。

您可以使用带有参数 internal = TRUE 的 usethis::use_data() 创建文件：

```
x <- sample(1000)
usethis::use_data(x, mtcars, internal = TRUE)
```

同样，为了使这些数据具有可复制性，最好在程序包中包含用于生成它的代码。并把它放到 data-raw/ 中。

位于 R/sysdata.rda 中的对象不会被导出（它们不应该被导出），因此不需要文档。它们只在你的程序里提供。

3.13.3 12.3 原始数据

如果要显示加载/解析原始数据的示例，请将原始文件放在 inst/extdata 中。安装包后，inst/ 中的所有文件（和文件夹）都会上移一级到顶级目录（因此它们不能有 R/ 或 DESCRIPTION 之类的名称）。要引用 inst/extdata 中的文件（无论是否已安装），请使用 system.file()。例如，readr 包使用 inst/extdata 来存储分隔文件，以便在示例中使用：

```
system.file("extdata", "mtcars.csv", package = "readr")
#> [1] "/home/travis/R/Library/readr/extdata/mtcars.csv"
```

注意：默认情况下，如果文件不存在，system.file() 不返回错——它只返回空字符串：

```
system.file("extdata", "iris.csv", package = "readr")
#> [1] ""
```

如果要在文件不存在时显示错误消息，请添加参数 mustWork = TRUE：

```
system.file("extdata", "iris.csv", package = "readr", mustWork = TRUE)
#> Error in system.file("extdata", "iris.csv", package = "readr", mustWork =
#> TRUE): no file found
```

3.13.4 12.4 其他数据

- 测试数据：可以直接将小文件放在测试目录中。但是请记住，单元测试是为了测试正确性，而不是性能，所以要保持小的规模。
- 简介（vignettes）数据。如果您想展示如何使用已经加载的数据，那么就将数据放入 data/ 中。如果要演示如何加载原始数据，请将该数据放入 inst/extdata 中。

3.13.5 12.5 CRAN 注记

一般来说，程序包中的数据集应该小于 1 MB ——如果更大则需要申请豁免。如果数据在它自己的程序包中翻译存疑，并且不会经常更新，那么这样做通常会更容易。您还应该确保已经对数据进行了最佳压缩：

1. 运行 `tools::checkRdaFiles()` 确定每个文件的最佳压缩率。
2. 重新运行 `usethis::use_data()`，并将参数 `compress` 设置为该最佳值。如果丢失了重新创建文件的代码，您可以使用 `tools::resaveRdaFiles` 将其保存到位。

3.14 第十四章安装后的文件

当程序包被安装后，`inst/` 中的所有内容都将被复制到程序包的顶级目录中。从某种意义上说，`inst/` 是反面的 `.Rbuildignore` ——`.Rbuildignore` 允许您从顶层删除任意文件和目录，而 `inst/` 则允许您添加它们。您可以随意在 `inst/` 中放入任何内容，但要注意：因为 `inst/` 会被复制到顶级目录中，所以您决不能使用与现有目录同名的子目录。这意味着您应该避免 `inst/build`、`inst/data`、`inst/demo`、`inst/exec`、`inst/help`、`inst/html`、`inst/inst`、`inst/libs`、`inst/Meta`、`inst/man`、`inst/po`、`inst/R`、`inst/src`、`inst/tests`、`inst/tools` 和 `inst/vignettes`。

本章讨论 `inst/` 中最常见的文件：

- `inst/AUTHOR` 和 `inst/COPYRIGHT`。如果程序包的版权和作者身份特别复杂，可以使用纯文本文件 `inst/COPYRIGHTS` 和 `inst/AUTHORS` 来提供更多信息。
- `inst/CITATION`：如何引用程序包，详情请参阅 package citation。
- `inst/docs`：这是旧式的简介（vignette），在现代软件包中应该避免使用。
- `inst/extdata`：示例和简介（vignette）的附加外部数据。有关详细信息，请参见 external data。
- `inst/java`，`inst/python` 等，参见 other languages。

要通过代码从 `inst/` 中查找文件，请使用 `system.file()` 。例如，要查找 `inst/extdata/mydata.csv`，您应该调用 `system.file("extdata", "mydata.csv", package = "mypackage")`。请注意，您在路径中省略了 ``inst/ 目录。如果程序包已被安装，或者已使用 `devtools::load_all()` 加载，那么此方法是有效的。

3.14.1 14.1 程序包的引用

`CITATION` 文件位于 `inst` 目录中，并与 `citation()` 函数紧密相连，该函数告诉您如何引用 R 和 R 程序包。不带任何参数调用 `citation()` 将告诉您如何引用 base R：

```
citation()
#>
#> To cite R in publications use:
#>
#>   R Core Team (2020). R: A language and environment for statistical
```

(下页继续)

(续上页)

```
#>   computing. R Foundation for Statistical Computing, Vienna, Austria.
#>   URL https://www.R-project.org/.
#>
#> A BibTeX entry for LaTeX users is
#>
#>   @Manual{,
#>     title = {R: A Language and Environment for Statistical Computing},
#>     author = {{R Core Team}},
#>     organization = {R Foundation for Statistical Computing},
#>     address = {Vienna, Austria},
#>     year = {2020},
#>     url = {https://www.R-project.org/},
#>   }
#>
#> We have invested a lot of time and effort in creating R, please cite it
#> when using it for data analysis. See also 'citation("pkgname")' for
#> citing R packages.
```

使用程序包名称调用该函数将告诉您如何引用该程序包：

```
citation("lubridate")
#>
#> To cite lubridate in publications use:
#>
#>   Garrett Grolemund, Hadley Wickham (2011). Dates and Times Made Easy
#>   with lubridate. Journal of Statistical Software, 40(3), 1-25. URL
#>   http://www.jstatsoft.org/v40/i03/.
#>
#> A BibTeX entry for LaTeX users is
#>
#>   @Article{,
#>     title = {Dates and Times Made Easy with {lubridate}},
#>     author = {Garrett Grolemund and Hadley Wickham},
#>     journal = {Journal of Statistical Software},
#>     year = {2011},
#>     volume = {40},
#>     number = {3},
#>     pages = {1--25},
#>     url = {http://www.jstatsoft.org/v40/i03/},
#>   }
```

要自定义您的程序包的引文，请添加如下所示的 inst/CITATION：

```
citHeader("To cite lubridate in publications use:")

citEntry(entry = "Article",
title        = "Dates and Times Made Easy with {lubridate}",
author       = personList(as.person("Garrett Grolemund"),
                  as.person("Hadley Wickham")),
journal      = "Journal of Statistical Software",
year         = "2011",
volume       = "40",
number       = "3",
pages        = "1--25",
url          = "http://www.jstatsoft.org/v40/i03/",

textVersion  =
paste("Garrett Grolemund, Hadley Wickham (2011).",
        "Dates and Times Made Easy with lubridate.",
        "Journal of Statistical Software, 40(3), 1-25.",
        "URL http://www.jstatsoft.org/v40/i03/.")
)
```

您需要创建 isnt/CITATION。如您所见，它非常简单：您只需要学习一个新函数 citEntry()。最重要的参数是：

- entry：引文类型、" Artical"、" Book"、" PhDThesis" 等。
- 标准书目信息，如 title、author`` （应该是 ``personList()）、year、journal、volume、issue、pages、……

完整的参数列表可以在 ?bibentry 中找到。

使用 citHeader() 和 citFooter() 添加其他建议。

3.14.2 14.2 其他语言

有时一个程序包包含其他编程语言编写的有用的补充脚本。一般来说，您应该避免这样做，因为它增加了额外的依赖关系，但是当打包来自另一种语言的大量代码时，它可能会很有用。例如，gdata 打包了 Perl 模块 Spreadsheet::ParseExcel 以将 excel 文件读入 R。

通常的惯例是将这种性质的脚本放入 inst/、inst/python、inst/perl、inst/ruby 等的子目录中。如果这些脚本对您的程序包很重要，请确保在 DESCRIPTION 的 SystemRequirements 字段中也添加了适当的编程语言。（此字段供人类阅读，因此不必担心如何指定它。）

Java 是一种特殊情况，因为您需要同时包含源代码（应该在 Java/ 中，并在 .Rinstignore 中列出）和已编

译的 jar 文件（应该放在 `inst/Java` 中）。确保将 `rJava` 添加到 `Imports` 中。

3.15 第十五章其他组件

还有四个其他目录是有效的顶级目录。它们很少使用：

- `demo/`：用于程序包演示。在引入 vignettes 之前它还是有用的，但现如今已不再推荐使用。请见下文。
- `exec/`：用于存放可执行脚本。与其他目录中的文件不同，`exec/` 中的文件会自动标记为可执行文件。
- `po/`：消息的翻译。该目录是有用的，但超出了本书的范围。有关更多详细信息，请参阅 "R extension" 的 Internationalization 章节。
- `tools/`：配置期间需要的辅助文件，或者用于需要生成脚本的源文件。意义不明

3.15.1 15.1 Demos

demo 是位于 `demo/` 中的 `.R` 文件。Demos 就像示例，但往往会更长。它们展示了如何将多个函数组织在一起来解决问题，而不是专注于单个函数。

使用 `demo()` 列出并访问 demos：

- 显示所有可用的 demos：`demo()`。
- 显示程序包中的所有 demos：`demo(package = "httr")`。
- 运行一个特定的 demo：`demo("oauth1-twitter", package = "httr")`。
- 查找 demo：`system.file("demo", "oauth1-twitter.R", package = "httr")`。

每个 demo 必须按以下格式在 `demo/00Index` 中列出：`demo-name Demo description`。`demo-name` 是不带扩展名的文件名，例如，`demo/my-demo.R` 显示为 `my-demo`。

默认情况下，demo 要求对于每个展示都需要有人工输入："Hit to see next plot:"。可以通过向 demo 文件中添加 `devAskNewPage(ask = FALSE)` 来覆盖此行为。您可以通过添加 `readline("press any key to continue")` 来添加暂停点。

一般来说，我不建议使用 demos。相反，考虑写一个 vignette：

- Demo 不是由 `R CMD check` 自动测试的。这意味着它们很容易在你不知情的情况下出问题。
- vignette 有输入和输出，因此读者可以看到结果，而不必自己运行代码。
- 较大的 demos 需要将代码与解释混合在一起，而 RMarkdown 比 R comments 更适合这项任务。
- vignettes 展示在 CRAN 程序包页面上。这使得新用户更容易发现它们。
- genindex
- modindex
- search