Уикем | Гроссер | Буманн

ADVANCED

R

С РЕШЕНИЯМИ И КОММЕНТАРИЯМИ

К вершинам мастерства

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Hadley Wickham, Malte Grosser, Henning Bumann

Advanced R

Advanced R Solutions

Second Edition

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

УДК 004.438R
ББК 32.973.22
У35

Уикем Х., Гроссер М., Буманн Х.
У35 R. К вершинам мастерства / пер. с англ. А. Ю. Гинько. – М.: ДМК Пресс, 2024. – 752 с.: ил.

ISBN 978-5-93700-247-1

R – безусловно лучшая среда для интерактивного анализа данных. Тем не менее язык R имеет множество особенностей, которые иногда скудно документированы. В данной книге Хэдли Уикем, один из лучших в мире гуру по R, проясняет эти неясные уголки и знакомит с современными библиотеками языка. В книге приведены решения и подробные комментарии ко всем упражнениям.

Издание предназначено программистам R, желающим углубить свои знания, а также будет полезно разработчикам на других языках, стремящимся узнать, что же делает язык R таким особенным.

УДК 004.438R
ББК 32.973.22

ISBN 978-0-815-38457-1 (англ.)

ISBN 978-5-93700-247-1 (рус.)

Содержание

Предисловие

Добро пожаловать в книгу *R. К вершинам мастерства* (Advanced R, 2nd Edition). При написании этого издания (в оригинале второго. – *Прим. пер.*) я ставил перед собой три главные цели:

- улучшить освещение концепций, важность которых я осознал только после публикации первого издания книги;
- уменьшить количество тем, практичность которых, несмотря на их кажущуюся привлекательность, оказалась под вопросом или полезность которых не прошла проверку временем;
- улучшить читаемость книги за счет упрощения повествования, облегчения кода и добавления более сотни полезных диаграмм.

Одно из важнейших изменений в этом издании книги связано с переходом на новые пакеты, в частности на `rlang` (http://rlang.r-lib.org), обеспечивающий прозрачное взаимодействие с низкоуровневыми структурами данных и операциями. В первом издании мы практически полностью пользовались исключительно базовыми функциями языка R, что приводило к определенным сложностям, связанным с их постоянными изменениями в отношении имен и аргументов на протяжении многих лет. Я продолжу по необходимости указывать эквиваленты используемого функционала применительно к базовому пакету R, но если вам важна реализация описываемых концепций исключительно в базовом R, вам лучше будет обратиться к первому изданию этой книги, которое можно найти по адресу http://adv-r.had.co.nz.

С момента выхода первого издания базовые концепции языка R не претерпели серьезных изменений, чего не скажешь о моем их понимании. В связи с этим общая структура части под названием *Основы* осталась практически без изменений, в то время как отдельные главы были значительно расширены и улучшены:

- глава 2 – *Имена и значения* – представляет собой абсолютно новую главу, которая поможет вам лучше понять отличия между объектами и их именами. Это понимание позволит вам более точно прогнозировать ситуации, когда R выполняет копирование структур данных, и заложит основы представления о функциональном программировании;
- глава 3 – *Векторы* (ранее она называлась *Структуры данных*) – была полностью переписана с прицелом на векторные типы данных в R, в числе которых – целочисленные значения, факторы и датафреймы. Также в ней мы более детально обсудим важную тему, связанную с векторами S3 (такими как дата и дата со временем), и поговорим о разновидности датафреймов, представленной в пакете `tibble` [Кирилл Мюллер (Kirill Müller) и Хэдли Уикем (Hadley Wickham), 2018]. Кроме того, я поделюсь

своим претерпевшим изменения мнением об использовании векторных типов данных;

- в главе 4 – *Подмножества* – мы поговорим о различиях между функциями [и [[по их назначению. Функция [извлекает множество значений, тогда как [[– единственное значение (ранее эти операции упоминались как функция с *сохранением формата* (preserved) и *упрощенная* (simplified) соответственно). В разделе 4.3 вы увидите паровозик, который поможет понять, как функция [[работает со списками, а также познакомитесь с новыми функциями, обеспечивающими предсказуемое поведение объектов при выходе индексов за допустимые границы;
- глава 5 – *Управляющие структуры* – впервые появилась в этом издании. Как-то я забыл в первом издании упомянуть про `if` и `for`...;
- в главе 6 – *Функции* – был изменен порядок повествования: оператор создания конвейеров (`%>%`) теперь упоминается в качестве третьего варианта вызова последовательности функций (раздел 6.3). Кроме того, большое внимание уделяется формам функций (раздел 6.8);
- в главе 7 – *Окружения* – подвергся изменениям раздел 7.4, посвященный особым окружениям, а также была расширена дискуссия, касающаяся стека вызовов (раздел 7.5);
- в главе 8 – *Состояния* – содержится материал, ранее находившийся в главе *Исключения и отладка*, а также абсолютно новые темы, касающиеся того, как в R работает система состояний. Помимо этого, в разделе 8.5 будет показано, как можно создать собственные классы системы состояний.

Главы после первой части были серьезно реорганизованы вокруг трех важнейших парадигм языка R: функционального программирования, объектно ориентированного программирования и метапрограммирования:

- тему функционального программирования я разбил на три главы по основным техникам: функционалы (глава 9), фабрики функций (глава 10) и функциональные операторы (глава 11). При этом основной упор я сделал на темах, имеющих практическое применение в науке о данных, и сократил количество теоретических отступлений. Теперь в этих главах используются функции из пакета `purrr` [Лайонел Хенри (Lionel Henry) и Хэдли Уикем (Hadley Wickham), 2018], что позволило мне больше сосредоточиться на лежащих в основе описываемых подходов идеях, а не на малозначительных деталях. В результате мне удалось существенно упростить главу, посвященную функциональным операторам;
- во втором издании книги объектно ориентированному программированию (ООП) уделено довольно много внимания. В том числе были добавлены абсолютно новые главы, посвященные базовым типам (глава 12), классам S3 (глава 13), S4 (глава 15), R6 (глава 14) и компромиссам при выборе подходящей системы ООП. В этих главах мы поговорим о том, как работают разные системы в R, а не о том, как использовать их максимально эффективно. Это вынужденная мера, связанная с тем, что многие технические подробности в других местах не описаны, а эффек-

тивному применению объектно ориентированного программирования в R можно посвятить целую книгу;

- метапрограммирование относится к набору инструментов, который может быть использован для создания кода из самого кода. В сравнении с первым изданием материал в этих главах был значительно расширен, теперь он целиком построен на базе фреймворка *tidy evaluation*, представляющего совокупность концепций, облегчающих использование принципов метапрограммирования и делающих его более безопасным. В главе 17 – *Общая картина* – описывается концепция метапрограммирования в широком смысле. В главе 18 – *Выражения* – мы взглянем на структуры данных, лежащие в основе концепции. Глава 19 будет посвящена квазицитированию, а глава 20 – выполнению кода в особых окружениях. В главе 21 – *Транслирование кода R* – мы соберем все полученные знания воедино и узнаем, как можно выполнять транслирование кода из одного языка (программирования) в другой;
- в заключительной части книги я собрал главы, посвященные важным техникам программирования, включая профилирование, измерение и повышение производительности, а также использование пакета `Rcpp`, обеспечивающего взаимодействие между R и C++. Содержание этих глав практически не изменилось по сравнению с первым изданием книги, но их организация была немного скорректирована. В частности, я поправил текст под использование более новых пакетов (`microbenchmark -> bench`, `lineprof -> profvis`), но по большей части материал остался прежним.

Несмотря на более подробное освещение многих тем во втором издании книги, с пятью главами мне пришлось распрощаться:

- глава со словарем была исключена из книги, поскольку она всегда казалась мне немного странной. Существуют более подходящие способы для демонстрации словарных сведений, не обязательно отводить для них отдельную главу;
- глава, посвященная стилям, перекочевала в онлайн по адресу https://style.tidyverse.org. Руководство по стилям также содержит информацию о новом пакете `styler` [Кирилл Мюллер (Kirill Müller) и Лоренц Вальтхерт (Lorenz Walthert), 2018], с помощью которого можно применять различные правила;
- глава, посвященная C, переехала на сайт https://github.com/hadley/r-internals, на котором со временем будет собрано полное руководство по написанию кода на C, способного работать со структурами данных R;
- глава, в которой обсуждались вопросы, связанные с памятью, также была исключена из книги. Большинство имеющегося материала было включено в главу 2, а с оставшейся частью, более технического характера, было решено расстаться;
- из книги пропала глава, посвященная производительности языка R. В ней было не так много полезных сведений, и она утратила свою актуальность с развитием R.

1

Введение

Я программирую на языке R вот уже более 15 лет, а последние пять лет делаю это на постоянной основе. Это позволило мне довольно хорошо изучить, как работает данный язык. И книга, которую вы держите в руках, – это моя попытка поделиться опытом и сделать путь постижения R моих читателей максимально легким и безболезненным, насколько это возможно. Чтение ее поможет вам избежать ловушек и тупиков, в которых я провел немало времени, и позволит изучить полезные инструменты и техники для решения большинства возникающих задач. Вместе с тем я попытаюсь показать вам, что, несмотря на некоторые причуды этого языка, идеально подходящего для применения в области науки о данных, он обладает присущей ему одному красотой и элегантностью.

1.1 Почему R?

Если вы только краем уха слышали об R, то можете задаться вопросом, зачем вам стоит изучать столь причудливый язык. Вот лишь несколько ответов на него:

- R – бесплатный язык с открытым исходным кодом, и он доступен на всех известных платформах. Таким образом, при выполнении анализа в языке R вы можете быть уверены, что любой сможет воспроизвести полученные вами результаты вне зависимости от места обитания и уровня достатка;
- язык R славится своим многогранным и дружелюбным сообществом как в онлайне (например, `#rstats` в твиттере – https://twitter.com/search?q=%23rstats), так и в офлайне (планирование митапов по адресу https://www.meetup.com/topics/r-programming-language). Также стоит упомянуть об интереснейшей еженедельной новостной рассылке https://rweekly.org и отдельном сообществе R-Ladies по адресу https://rladies.org, в котором представительницы прекрасного пола могут с легкостью найти своих единомышленниц;
- для языка R написано множество пакетов в области статистического моделирования, машинного обучения, визуализации, импорта и манипулирования данными. Какую бы модель или график вы ни задумали

построить, скорее всего, найдется кто-то, кто уже делал это, и вы с легкостью сможете воспользоваться его опытом;

- в R присутствует масса инструментов для обмена рабочими материалами. Язык разметки RMarkdown (https:// rmarkdown.rstudio.com) поможет вам быстро преобразовать результаты своего труда в документы HTML, PDF или Word, а также в презентации PowerPoint, дашборды и прочие форматы. А с помощью Shiny (http://shiny.rstudio.com) можно без труда создавать полноценные интерактивные приложения без знания HTML и Javascript;
- интерактивная среда разработки RStudio (http://www.rstudio.com/ide) идеально подходит для написания проектов, связанных с наукой о данных, интерактивным и статистическим анализом;
- возможность пользоваться передовыми разработками. Многие исследователи в области статистики и машинного обучения прилагают к своим научным трудам соответствующие пакеты на языке R, что дает разработчикам возможность использовать в своих целях наиболее актуальные статистические техники и реализации;
- удобства при проведении анализа данных. В R присутствуют богатые возможности для работы с датафреймами, векторизацией и пропущенными значениями;
- применение функционального программирования. Идеи функционального программирования могут быть активно использованы при решении задач в области науки о данных, а язык R по своей природе функционален и предлагает большое количество возможностей для эффективного программирования;
- RStudio (https://www.rstudio.com) – компания, зарабатывающая деньги на продаже профессиональных продуктов командам пользователей R и большую часть прибыли инвестирующая в сообщество разработчиков открытого исходного кода (более половины программистов-разработчиков в RStudio работают над проектами с открытым кодом). Лично я работаю в RStudio, поскольку свято верю в миссию этой компании;
- мощные возможности в области метапрограммирования. В R вы можете писать очень лаконичные функции. Кроме того, R предлагает богатейшие возможности для разработки специфических инструментов для конкретной предметной области, таких как `ggplot2`, `dplyr`, `data.table` и др.;
- легкость, с которой R взаимодействует с высокопроизводительными языками программирования вроде C, Fortran и C++.

Конечно, язык R не идеален. Самой большой проблемой (и одновременно возможностью!) языка R является то, что большинство его пользователей не являются программистами. Это приводит к следующим сложностям:

- зачастую вы будете сталкиваться с кодом на языке R, который был написан впопыхах для решения конкретной задачи. Как результат от такого кода вряд ли будет уместно ожидать какой-то особой элегантности,

быстроты или удобочитаемости. Большинство разработчиков не тратят время на ревизию кода и исправление недочетов;

- в сравнении с другими языками программирования сообщество R больше сосредоточено на результате, а не на процессе. Большинство знаний по оптимизации процесса написания программного обеспечения не систематизированы и разрозненны. К примеру, далеко не все разработчики на R используют в своей практике средства управления версиями исходного кода и принципы автоматизированного тестирования;
- метапрограммирование – это палка о двух концах. Слишком в большом количестве функций в R применяются разные ухищрения для снижения объема ввода, что в результате делает код более трудным для чтения и неустойчивым к отказам;
- в различных сторонних пакетах R и даже в базовом наблюдаются примеры непоследовательности. В итоге каждый раз, когда вы программируете на R, вы, по сути, противостоите 25-летней эволюции языка, что затрудняет процесс его освоения новичками – слишком много всего нужно запоминать;
- язык R не славится высокой скоростью, а неоптимально написанный код на нем может оказаться чрезвычайно медленным. Кроме того, R довольно расточительно расходует память.

Лично мне кажется, что эти несовершенства создают богатые возможности для опытных программистов, позволяющие им влиять на развитие языка R и его сообщества. Пользователи R стремятся писать высокоэффективный код, особенно в части воспроизводимых исследований, но у них не хватает для этого знаний и навыков. Я надеюсь, что эта книга не только позволит большему числу пользователей R перейти в разряд разработчиков, но и поспособствует переходу в R специалистов из других языков программирования.

1.2 Для кого эта книга

Книга, которую вы держите в руках, главным образом нацелена на две дополняющие друг друга группы читателей:

- программисты на R, обладающие определенными навыками, но желающие более глубоко погрузиться в язык, понять, как он работает, и изучить новые приемы для решения разного рода задач;
- программисты из других языков, делающие свои первые шаги в R и желающие понять, почему он работает так, как работает.

Чтобы извлечь максимум возможного из этой книги, за плечами у вас должен быть определенный опыт написания кода на R или любом другом языке программирования. Таким образом, вы уже должны быть знакомы с основами анализа данных, включающими в себя импорт, манипулирование данных и их визуализацию, обладать определенным опытом написания

функций и быть знакомым с установкой и использованием пакетов из репозитория CRAN.

В процессе написания этой книги я попытался уместить ее на тонкой грани между справочником, который можно использовать для поиска нужной информации, и книгой, пригодной для чтения от корки до корки. В результате мне пришлось идти на определенные компромиссы из-за сложности сочетания повествовательного стиля с сохранением справочной структуры, а некоторые концепции гораздо легче объяснять при наличии у читателя соответствующей технической базы. Где это было необходимо, я попытался использовать заметки и перекрестные ссылки, чтобы у вас складывалось более цельное представление о прочитанном материале.

1.3 Что вы узнаете из этой книги

В этой книге я сделаю акцент на главном качестве, которым, по моему мнению, должен обладать любой опытный разработчик на R. Оно состоит в обладании глубокими знаниями основ языка в сочетании с умением изучить любую тему самостоятельно при необходимости.

В процессе чтения этой книги вы:

- узнаете все основы языка R. Изучите сложные типы данных и сможете выполнять необходимые действия над ними. Глубоко погрузитесь в понимание того, как работают функции, узнаете об окружениях и научитесь использовать систему состояний;
- получите представление о принципах функционального программирования и поймете, почему они широко применяются в области науки о данных. Вы сможете как использовать имеющиеся средства функционального программирования, так и создавать свои собственные при необходимости;
- познакомитесь со всем разнообразием систем объектно ориентированного программирования в R. По большей части вы научитесь работать с системой S3, но также получите представление о том, как функционируют системы S4 и R6 и где в случае необходимости можно почерпнуть о них полезную информацию;
- научитесь обращаться с палкой о двух концах под названием метапрограммирование. Узнаете, как создавать функции, использующие принципы *tidy evaluation*, с целью повышения элегантности кода при выполнении важных операций. Вместе с тем вы поймете, какие опасности могут ожидать вас на этом пути, и научитесь их избегать;
- разовьете интуицию относительно того, какие операции в R выполняются медленно или расходуют чрезмерные ресурсы памяти. Вы научитесь использовать механизмы профилирования кода с целью определения узких мест и изучите C++ в объеме, необходимом для преобразования медленных функций на R в эффективные эквиваленты на C++.

1.4 Чего вы не узнаете из этой книги

Эта книга посвящена языку программирования R как таковому, а не языку R как средству анализа данных. Если вы хотите улучшить свои навыки в области науки о данных, я бы порекомендовал вам погрузиться в изучение tidyverse (https://www.tidyverse.org) – коллекции полезных пакетов, разработанных мной и моими коллегами. Читая эту книгу, вы познакомитесь с приемами и техниками, которые, в частности, использовались при разработке пакетов tidyverse. Если же вам хочется узнать, как именно работают уже написанные пакеты, вам стоит почитать книгу *R for Data Science* (https://r4ds.had.co.nz).

Чтобы поделиться кодом, написанным на R, с коллегами, вам необходимо создать пакет. Это поможет объединить код с документацией и модульными тестами и легко выкладывать наработки в репозиторий CRAN. Мне кажется, проще всего при разработке пакетов пользоваться следующими инструментами: `devtools` (http://devtools.r-lib.org), `roxygen2` (http:// klutometis.github.io/roxygen), `testthat` (http://testthat.r-lib.org) и `usethis` (http://usethis.r-lib.org). Подробно о процессе создания пакетов с помощью перечисленных инструментов вы можете почитать в книге *R packages* (http://r-pkgs.had.co.nz).

1.5 Метатехники

Существуют две метатехники, помогающие существенно улучшить свои навыки как программиста на языке R: чтение исходного кода и развитие научного образа мышления.

Читать чужие исходные коды очень важно, поскольку это помогает улучшить качество собственного кода. Вы можете начать с чтения исходников функций и пакетов, которыми постоянно пользуетесь в своей работе. В процессе вы будете обнаруживать приемы и техники, которые можно использовать в собственных наработках, и постепенно начнете понимать, как должен выглядеть хорошо написанный код на языке R. Вместе с тем вы будете сталкиваться с приемами, которые не придутся вам по душе, либо по причине неочевидности их ценности, либо из-за внутреннего неприятия. Такие моменты тоже очень полезны – они помогают укрепить ваше мнение о том, что хорошо, а что плохо.

Что касается научного образа мышления, то его присутствие очень помогает при изучении языка R. Если вы не понимаете, как работает тот или иной прием, вы должны выдвинуть определенную гипотезу, разработать серию экспериментов, прогнать их и зафиксировать результаты. Это бывает очень полезно, когда вы не можете с чем-то разобраться самостоятельно и хотите показать коллегам свои изыскания. Кроме того, это подготовит вас к тому, чтобы легко и безболезненно изменить свое мнение, когда вы узнаете правильный ответ.

1.6 Список рекомендуемой литературы

Поскольку сообщество R по большей части состоит из специалистов по работе с данными, а не инженеров в области вычислительной техники, вы встретите не так много книг, посвященных глубоким техническим аспектам этого языка. На своем пути я в основном использовал ресурсы, связанные с другими языками программирования. В R одновременно используются принципы функционального и объектно ориентированного программирования. Изучение того, как эти аспекты реализованы в R, поможет вам по максимуму использовать знания в этих областях, полученные в других языках программирования, и определить области, в которых вам необходимо прибавить.

В постижении того, почему объектные системы в R работают именно так, как работают, мне лично очень помогла книга *The Structure and Interpretation of Computer Programs*[1] [Гарольд Абельсон (Harold Abelson) и др., 1996]. Это одновременно компактная и глубокая книга, после прочтения которой я впервые почувствовал, что могу разработать собственную объектно ориентированную систему. Эта книга помогла мне глубже проникнуть в реализацию объектно ориентированного программирования в R и понять ее сильные и слабые стороны. Кроме того, книга оказывает помощь в развитии функционального мышления, когда вы создаете функции, которые по отдельности являются довольно простыми, а вместе представляют слаженный и мощный механизм.

В понимании того, на какие компромиссы пошли создатели языка R в сравнении с другими языками программирования и почему они это сделали, мне помогла книга *Concepts, Techniques and Models of Computer Programming* [Питер Ван Рой (Peter Van Roy) и Сеиф Хариди (Seif Haridi), 2004]. Она поспособствовала моему осознанию того, что принцип *копирования при изменении* (copy-on-modify), принятый в R, значительно облегчает понимание кода и что текущая реализация этого принципа обладает некоторыми недостатками, которые можно исправить.

Если вы хотите улучшить свои навыки как программиста, лучшей книги, чем *Программист-прагматик* (*The Pragmatic Programmer*) [Эндрю Хант (Andrew Hunt) и Дэвид Томас (David Thomas), 1990], вам не найти. Эта книга не привязана к какому-то конкретному языку, а развивает навыки программирования в целом.

1.7 Помощь в разработке

На данный момент существует три основных источника помощи тем, кто столкнулся с проблемами при разработке на языке R: RStudio Community

[1] Вы можете прочитать ее бесплатно на английском по адресу https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book.html.

(https://community.rstudio.com), StackOverflow (http://stackoverflow.com) и список рассылки R-help (https://stat.ethz.ch/mailman/listinfo/r-help). Любой из этих вариантов может вам помочь в решении возникших проблем в разработке, но у каждого из них есть своя культура. Обычно бывает уместно потратить какое-то время на изучение особенностей, характерных для этих сообществ, прежде чем создавать свой первый запрос.

В русскоязычном сообществе помощь в решении любых вопросов можно получить в специализированных чатах в Telegram:

- R (язык программирования) – https://t.me/rlang_ru;
- Горячая линия R – https://t.me/hotlineR_EU;
- BioStat <- R – https://t.me/chat_biostat_R.

Пара советов, которые помогут сократить время получения ответов:

- убедитесь, что у вас установлена самая свежая версия R и пакетов, ставших источником проблем. Вполне возможно, что проблема, с которой вы столкнулись, была решена в одной из более поздних версий;
- потратьте немного времени на создание *воспроизводимого примера* (***re**producible **ex**ample*, или *reprex*), который поможет другим оказать вам помощь. Кроме того, зачастую вы сможете найти ответ на интересующий вас вопрос прямо в процессе создания этого примера. Я настоятельно рекомендую воспользоваться специальным пакетом `reprex` (https://reprex.tidyverse.org) при разработке воспроизводимых примеров.

В русскоязычном издании книги приведены решения всех упражнений с подробным описанием, которые изначально были изданы в виде отдельной книги Advanced R Solutions, написанной Мальте Гроссером (Malte Grosser) и Хеннингом Буманном (Henning Bumann). Электронная версия книги располагается по адресу https://advanced-r-solutions.rbind.io.

1.8 Благодарности

Я бы хотел поблагодарить за помощь в написании книги очень многих людей в сообществах R-devel, R-help, Stack Overflow и RStudio. Всех перечислять будет слишком долго, но отдельно я хотел бы выделить Люка Тьерни (Luke Tierney), Джона Чемберса (John Chambers), Джей Джей Аллаира (JJ Allaire) и Брайана Рипли (Brian Ripley) за время, потраченное на бесчисленное количество исправлений неточностей.

Эту книгу я писал в открытом пространстве (https://github.com/hadley/adv-r), а отдельные главы по их готовности выкладывал в своем твиттере (https://twitter.com/hadleywickham). Таким образом, ее можно считать результатом коллективного творчества: многие в процессе написания глав проверяли мои черновики, исправляли опечатки, предлагали улучшения и добавляли свои примеры. Без этих дополнений и исправлений книга не получилась бы такой, какая она есть сейчас, и я очень благодарен всем, кто поучаствовал

в процессе ее создания. Также отдельно хотелось бы сказать спасибо Джеффу Хаммербахеру (Jeff Hammerbacher), Питеру Ли (Peter Li), Дункану Мердоку (Duncan Murdoch) и Грегу Уилсону (Greg Wilson), прочитавшим книгу от корки до корки и внесшим целый ряд поправок и дополнений.

Кроме того, я хочу перечислить ники 386 людей (в алфавитном порядке), которые внесли свой вклад в написание книги: Aaron Wolen (@aaronwolen), @absolutelyNoWarranty, Adam Hunt (@adamphunt), @agrabovsky, Alexander Grueneberg (@agrueneberg), Anthony Damico (@ajdamico), James Manton (@ajdm), Aaron Schumacher (@ajschumacher), Alan Dipert (@alandipert), Alex Brown (@alexbbrown), @alexperrone, Alex Whitworth (@alexWhitworth), Alexandros Kokkalis (@alko989), @amarchin, Amelia McNamara (@AmeliaMN), Bryce Mecum (@amoeba), Andrew Laucius (@andrewla), Andrew Bray (@andrewpbray), Andrie de Vries (@andrie), Angela Li (@angela-li), @aranlunzer, Ari Lamstein (@arilamstein), @asnr, Andy Teucher (@ateucher), Albert Vilella (@avilella), baptiste (@baptiste), Brian G. Barkley (@BarkleyBG), Mara Averick (@batpigandme), Byron (@bcjaeger), Brandon Greenwell (@bgreenwell), Brandon Hurr (@bhive01), Jason Knight (@binarybana), Brett Klamer (@bklamer), Jesse Anderson (@blindjesse), Brian Mayer (@blmayer), Benjamin L. Moore (@blmoore), Brian Diggs (@BrianDiggs), Brian S. Yandell (@byandell), @carey1024, Chip Hogg (@chiphogg), Chris Muir (@ChrisMuir), Christopher Gandrud (@christophergandrud), Clay Ford (@clayford), Colin Fay (@ColinFay), @cortinah, Cameron Plouffe (@cplouffe), Carson Sievert (@cpsievert), Craig Citro (@craigcitro), Craig Grabowski (@craiggrabowski), Christopher Roach (@croach), Peter Meilstrup (@crowding), Crt Ahlin (@crtahlin), Carlos Scheidegger (@cscheid), Colin Gillespie (@csgillespie), Christopher Brown (@ctbrown), Davor Cubranic (@cubranic), Darren Cusanovich (@cusanovich), Christian G. Warden (@cwarden), Charlotte Wickham (@cwickham), Dean Attali (@daattali), Dan Sullivan (@dan87134), Daniel Barnett (@daniel-barnett), Daniel (@danielruc91), Kenny Darrell (@darrkj), Tracy Nance (@datapixie), Dave Childers (@davechilders), David Vukovic (@david-vukovic), David Rubinger (@davidrubinger), David Chudzicki (@dchudz), Deependra Dhakal (@DeependraD), Daisuke ICHIKAWA (@dichika), david kahle (@dkahle), David LeBauer (@dlebauer), David Schweizer (@dlschweizer), David Montaner (@dmontaner), @dmurdoch, Zhuoer Dong (@dongzhuoer), Doug Mitarotonda (@dougmitarotonda), Dragoș Moldovan-Grünfeld (@dragosmg), Jonathan Hill (@Dripdrop12), @drtjc, Julian During (@duju211), @duncanwadsworth, @eaurele, Dirk Eddelbuettel (@eddelbuettel), @EdFineOKL, Eduard Szöcs (@EDiLD), Edwin Thoen (@EdwinTh), Ethan Heinzen (@eheinzen), @eijoac, Joel Schwartz (@eipi10), Eric Ronald Legrand (@elegrand), Elio Campitelli (@eliocamp), Ellis Valentiner (@ellisvalentiner), Emil Hvitfeldt (@EmilHvitfeldt), Emil Rehnberg (@EmilRehnberg), Daniel Lee (@erget), Eric C. Anderson (@eriqande), Enrico Spinielli (@espinielli), @etb, David Hajage (@eusebe), Fabian Scheipl (@fabian-s), @flammy0530, François Michonneau (@fmichonneau), Francois Pepin (@fpepin), Frank Farach (@frankfarach), @freezby, Frans van Dunné (@FvD), @fyears, @gagnagaman, Garrett Grolemund (@garrettgman), Gavin Simpson (@gavinsimpson), Brooke Anderson (@geanders), @gezakiss7, @gggtest, Gökçen Eraslan (@gokceneraslan), Josh Goldberg

(@GoldbergData), Georg Russ (@gr650), @grasshoppermouse, Gregor Thomas (@gregorp), Garrett See (@gsee), Ari Friedman (@gsk3), Gunnlaugur Thor Briem (@gthb), Greg Wilson (@gvwilson), Hamed (@hamedbh), Jeff Hammerbacher (@hammer), Harley Day (@harleyday), @hassaad85, @helmingstay, Henning (@henningsway), Henrik Bengtsson (@HenrikBengtsson), Ching Boon (@hoscb), @hplieninger, Hörmet Yiltiz (@hyiltiz), Iain Dillingham (@iaindillingham), @IanKopacka, Ian Lyttle (@ijlyttle), Ilan Man (@ilanman), Imanuel Costigan (@imanuelcostigan), Thomas Bürli (@initdch), Os Keyes (@Ironholds), @irudnyts, i (@isomorphisms), Irene Steves (@isteves), Jan Gleixner (@jan-glx), Jannes Muenchow (@jannes-m), Jason Asher (@jasonasher), Jason Davies (@jasondavies), Chris (@jastingo), jcborras (@jcborras), Joe Cheng (@jcheng5), John Blischak (@jdblischak), @jeharmse, Lukas Burk (@jemus42), Jennifer (Jenny) Bryan (@jennybc), Justin Jent (@jentjr), Jeston (@JestonBlu), Josh Cook (@jhrcook), Jim Hester (@jimhester), @JimInNashville, @jimmyliu2017, Jim Vine (@jimvine), Jinlong Yang (@jinlong25), J.J. Allaire (@jjallaire), @JMHay, Jochen Van de Velde (@jochenvdv), Johann Hibschman (@johannh), John Baumgartner (@johnbaums), John Horton (@johnjosephhorton), @johnthomas12, Jon Calder (@jonmcalder), Jon Harmon (@jonthegeek), Julia Gustavsen (@joooolia), JorneBiccler (@JorneBiccler), Jeffrey Arnold (@jrnold), Joyce Robbins (@jtr13), Juan Manuel Truppia (@juancentro), @juangomezduaso, Kevin Markham (@justmarkham), john verzani (@jverzani), Michael Kane (@kaneplusplus), Bart Kastermans (@kasterma), Kevin D'Auria (@kdauria), Karandeep Singh (@kdpsingh), Ken Williams (@kenahoo), Kendon Bell (@kendonB), Kent Johnson (@kent37), Kevin Ushey (@kevinushey), 电线杆 (@kfeng123), Karl Forner (@kforner), Kirill Sevastyanenko (@kirillseva), Brian Knaus (@knausb), Kirill Müller (@krlmlr), Kriti Sen Sharma (@ksens), Kai Tang(唐恺) (@ktang), Kevin Wright (@kwstat), suo.lawrence.liu (@Lawrence-Liu), @ldfmrails, Kevin Kainan Li (@legendre6891), Rachel Severson (@leighseverson), Laurent Gatto (@lgatto), C. Jason Liang (@liangcj), Steve Lianoglou (@lianos), Yongfu Liao (@liao961120), Likan (@likanzhan), @lindbrook, Lingbing Feng (@Lingbing), Marcel Ramos (@LiNk-NY), Zhongpeng Lin (@linzhp), Lionel Henry (@lionel-), Lluís (@llrs), myq (@lrcg), Luke W Johnston (@lwjohnst86), Kevin Lynagh (@lynaghk), @MajoroMask, Malcolm Barrett (@malcolmbarrett), @mannyishere, @mascaretti, Matt (@mattbaggott), Matthew Grogan (@mattgrogan), @matthewhillary, Matthieu Gomez (@matthieugomez), Matt Malin (@mattmalin), Mauro Lepore (@maurolepore), Max Ghenis (@MaxGhenis), Maximilian Held (@maxheld83), Michal Bojanowski (@mbojan), Mark Rosenstein (@mbrmbr), Michael Sumner (@mdsumner), Jun Mei (@meijun), merkliopas (@merkliopas), mfrasco (@mfrasco), Michael Bach (@michaelbach), Michael Bishop (@MichaelMBishop), Michael Buckley (@michaelmikebuckley), Michael Quinn (@michaelquinn32), @miguelmorin, Michael (@mikekaminsky), Mine Cetinkaya-Rundel (@mine-cetinkaya-rundel), @mjsduncan, Mamoun Benghezal (@MoBeng), Matt Pettis (@mpettis), Martin Morgan (@mtmorgan), Guy Dawson (@Mullefa), Nacho Caballero (@nachocab), Natalya Rapstine (@natalya-patrikeeva), Nick Carchedi (@ncarchedi), Pascal Burkhard (@Nenuial), Noah Greifer (@ngreifer), Nicholas Vasile (@nickv9), Nikos Ignatiadis (@nignatiadis), Nina Munkholt Jakobsen (@nmjakobsen), Xavier Laviron

(@norival), Nick Pullen (@nstjhp), Oge Nnadi (@ogennadi), Oliver Paisley (@oliverpaisley), Pariksheet Nanda (@omsai), Øystein Sørensen (@osorensen), Paul (@otepoti), Otho Mantegazza (@othomantegazza), Dewey Dunnington (@paleolimbot), Paola Corrales (@paocorrales), Parker Abercrombie (@parkerabercrombie), Patrick Hausmann (@patperu), Patrick Miller (@patr1ckm), Patrick Werkmeister (@Patrick01), @paulponcet, @pdb61, Tom Crockett (@pelotom), @pengyu, Jeremiah (@perryjer1), Peter Hickey (@PeteHaitch), Phil Chalmers (@philchalmers), Jose Antonio Magaña Mesa (@picarus), Pierre Casadebaig (@picasa), Antonio Piccolboni (@piccolbo), Pierre Roudier (@pierreroudier), Poor Yorick (@pooryorick), Marie-Helene Burle (@prosoitos), Peter Schulam (@pschulam), John (@quantbo), Quyu Kong (@qykong), Ramiro Magno (@ramiromagno), Ramnath Vaidyanathan (@ramnathv), Kun Ren (@renkun-ken), Richard Reeve (@richardreeve), Richard Cotton (@richierocks), Robert M Flight (@rmflight), R. Mark Sharp (@rmsharp), Robert Krzyzanowski (@robertzk), @robiRagan, Romain François (@romainfrancois), Ross Holmberg (@rossholmberg), Ricardo Pietrobon (@rpietro), @rrunner, Ryan Walker (@rtwalker), @rubenfcasal, Rob Weyant (@rweyant), Rumen Zarev (@rzarev), Nan Wang (@sailingwave), Samuel Perreault (@samperochkin), @sbgraves237, Scott Kostyshak (@scottkosty), Scott Leishman (@scttl), Sean Hughes (@seaaan), Sean Anderson (@seananderson), Sean Carmody (@seancarmody), Sebastian (@sebastian-c), Matthew Sedaghatfar (@sedaghatfar), @see24, Sven E. Templer (@setempler), @sflippl, @shabbybanks, Steven Pav (@shabbychef), Shannon Rush (@shannonrush), S'busiso Mkhondwane (@sibusiso16), Sigfried Gold (@Sigfried), Simon O'Hanlon (@simonohanlon101), Simon Potter (@sjp), Leo Razoumov (@slonik-az), Richard M. Smith (@Smudgerville), Steve (@SplashDance), Scott Ritchie (@sritchie73), Tim Cole (@statist7), @ste-fan, @stephens999, Steve Walker (@stevencarlislewalker), Stefan Widgren (@stewid), Homer Strong (@strongh), Suman Khanal (@sumanstats), Dirk (@surmann), Sebastien Vigneau (@svigneau), Steven Nydick (@swnydick), Taekyun Kim (@taekyunk), Tal Galili (@talgalili), @Tazinho, Tyler Bradley (@tbradley1013), Tom B (@tbuckl), @tdenes, @thomasherbig, Thomas (@thomaskern), Thomas Lin Pedersen (@thomasp85), Thomas Zumbrunn (@thomaszumbrunn), Tim Waterhouse (@timwaterhouse), TJ Mahr (@tjmahr), Thomas Nagler (@tnagler), Anton Antonov (@tonytonov), Ben Torvaney (@Torvaney), Jeff Allen (@trestletech), Tyler Rinker (@trinker), Chitu Okoli (@Tripartio), Kirill Tsukanov (@tskir), Terence Teo (@tteo), Tim Triche, Jr. (@ttriche), @tyhenkaline, Tyler Ritchie (@tylerritchie), Tyler Littlefield (@tyluRp), Varun Agrawal (@varun729), Vijay Barve (@vijaybarve), Victor (@vkryukov), Vaidotas Zemlys-Balevičius (@vzemlys), Winston Chang (@wch), Linda Chin (@wchi144), Welliton Souza (@Welliton309), Gregg Whitworth (@whitwort), Will Beasley (@wibeasley), William R Bauer (@WilCrofter), William Doane (@WilDoane), Sean Wilkinson (@wilkinson), Christof Winter (@winterschlaefer), Jake Thompson (@wjakethompson), Bill Carver (@wmc3), Wolfgang Huber (@wolfganghuber), Krishna Sankar (@xsankar), Yihui Xie (@yihui), yang (@yiluheihei), Yoni Ben-Meshulam (@yoni), @yuchouchen, Yuqi Liao (@yuqiliao), Hiroaki Yutani (@yutannihilation), Zachary Foster (@zachary-foster), @zachcp, @zackham, Sergio Oller (@zeehio), Edward Cho (@zerokarmaleft), Albert Zhao (@zxzb).

1.9 Соглашения

Функции в этой книге будут обозначаться как `f()`, переменные и аргументы функций – как `g`, а пути – как `h/`.

В объемных блоках кода ввод и вывод могут идти вперемешку. Для удобства чтения я буду предварять строки вывода комментариями (`#>`), чтобы вы могли легко копировать исходный код в свой редактор, если читаете электронную версию книги.

Во многих примерах из книги используется генератор случайных чисел. Чтобы эти примеры были воспроизводимы в вашей среде, в начале каждой главы автоматически запускается инструкция `set.seed(1014)`, инициализирующая генератор.

1.10 Выходные данные

Книга была написана с помощью пакета `bookdown` (http://bookdown.org) в RStudio (http://www.rstudio.com/ide). Сайт https://adv-r.hadley.nz размещен на хостинге netlify (http://netlify.com) и автоматически обновляется после каждой фиксации изменений в travis-ci (https://travis-ci.org). Исходный код доступен на GitHub по адресу https://github.com/hadley/adv-r. В оригинальной печатной версии книги код написан с использованием шрифта inconsolata (http://levien.com/type/myfonts/ inconsolata.html). Эмодзи в оригинальной книге были позаимствованы из библиотеки с открытой лицензией Twitter Emoji (https://github.com/twitter/twemoji).

Данная версия книги была написана с использованием R версии 3.5.2 (2018-12-20) и следующих версий пакетов:

`bench`: 1.0.1, *Github (r-lib/bench@97844d5)*
`bookdown`: 0.9, *CRAN (R 3.5.0)*
`dbplyr`: 1.3.0.9000, *local*
`desc`: 1.2.0, *Github (r-lib/desc@42b9578)*
`emo`: 0.0.0.9000, *Github (hadley/emo@02a5206)*
`ggbeeswarm`: 0.6.0, *CRAN (R 3.5.0)*
`ggplot2`: 3.0.0, *CRAN (R 3.5.0)*
`knitr`: 1.20, *standard (@1.20)*
`lobstr`: 1.0.1, *CRAN (R 3.5.1)*
`memoise`: 1.1.0.9000, *Github (hadley/memoise@1650ad7)*
`png`: 0.1-7, *CRAN (R 3.5.0)*
`profvis`: 0.3.5, *CRAN (R 3.5.1)*
`Rcpp`: 1.0.0.1, *Github (RcppCore/Rcpp@0c9f683)*
`rlang`: 0.3.1.9000, *Github (r-lib/rlang@7243c6d)*
`rmarkdown`: 1.11, *CRAN (R 3.5.0)*
`RSQLite`: 2.1.1.9002, *Github (r-dbi/RSQLite@0db36af)*

`scales`: 1.0.0, *CRAN (R 3.5.0)*
`sessioninfo`: 1.1.1, *CRAN (R 3.5.1)*
`sloop`: 1.0.0.9000, *local*
`testthat`: 2.0.1.9000, *local*
`tidyr`: 0.8.3.9000, *local*
`vctrs`: 0.1.0.9002, *Github (r-lib/vctrs@098154c)*
`zeallot`: 0.1.0, *CRAN (R 3.5.0)*

Часть I

Основы

Введение

Главы, присутствующие в первой части книги, помогут вам заложить прочный фундамент в путешествии по основным компонентам R. Я ожидаю, что со многими из этих компонентов вы уже встречались ранее, но, возможно, не изучали их отдельно. Для проверки ваших текущих знаний каждая глава будет начинаться с небольшого опроса. Если вы ответили на все поставленные вопросы правильно, можете спокойно переходить к следующей главе.

1. В главе 2 вы узнаете о важном разделении в языке R, о котором ранее, возможно, не задумывались. Речь идет о разделении объектов и их имен. Узнав об этом, вы сможете с легкостью предугадывать, когда R будет выполнять дорогостоящее копирование данных, что поможет вам понять, какие базовые операции являются дешевыми, а какие – дорогими.
2. В главе 3 мы погрузимся в мир векторов и увидим, как различные типы векторов уживаются вместе. Также мы узнаем об атрибутах, позволяющих хранить произвольные метаданные, и заложим основы двух используемых в R объектно ориентированных систем.
3. В главе 4 мы увидим, как можно использовать подмножества для написания лаконичного и эффективного кода на R. Понимание базовых компонентов позволит вам решать задачи путем комбинирования строительных блоков по-новому.
4. В главе 5 будут представлены механизмы управляющих структур, которые позволят запускать код только при выполнении определенных условий или осуществлять циклическое выполнение с изменяющимися входными характеристиками. Эти управляющие структуры включают в себя конструкции `if` и `for`, а также связанные с ними инструкции `switch()` и `while`.
5. В главе 6 мы познакомимся с функциями – наиболее важным строительным блоком в R. Вы узнаете, как работают функции, и познакомитесь с правилами видимости, согласно которым осуществляется поиск значений по именам. Также мы уделим внимание технологии ленивых, или отложенных, вычислений и посмотрим, как можно управлять происходящим при выходе из функции.
6. Глава 7 будет посвящена окружениям, представляющим структуру данных, не имеющую никакого отношения к анализу данных, но играющую важнейшую роль в понимании того, как работает R. Окружения позволяют связывать имена со значениями и лежат в основе механизма пространства имен пакетов. В отличие от большинства языков программирования, в R окружения являются объектами первого класса, что подразумевает возможность манипулирования с ними как с любыми другими объектами.

7. В главе 8 мы познакомимся с состояниями. Этот термин включает в себя ошибки, предупреждения и сообщения, возникающие в процессе выполнения кода. Вы наверняка сталкивались с ними ранее, а в этой главе узнаете, как можно оперировать с ними в своих собственных функциях и обрабатывать их возникновение в других местах в коде.

2

Имена и значения

2.1 Введение

В языке программирования R очень важно понимать разницу между объектом и его именем. Это поможет вам:

- более точно предсказывать производительность и потребление памяти при выполнении кода;
- писать более быстрый код, избегая создания случайных копий объектов, что является одним из распространенных источников проблем с производительностью;
- быстрее освоить инструменты функционального программирования R.

Цель этой главы состоит в том, чтобы помочь вам понять, чем отличаются имена и значения и когда именно R создает копии объектов.

Контрольные вопросы

Ответьте на приведенные ниже вопросы, чтобы понять, можете ли вы пропустить эту главу. Ответы вы найдете в конце главы, в разделе 2.7.

1. Дан следующий датафрейм. Как создать в нем колонку с именем `3`, содержащую сумму значений колонок `1` и `2`? Вы можете пользоваться только оператором `$`, но не `[[`. Какие трудности могут возникнуть при именовании переменных `1`, `2` и `3`?

```
df <- data.frame(runif(3), runif(3))
names(df) <- c(1, 2)
```

2. Сколько места в памяти будет занимать переменная `y` в приведенном ниже коде?

```
x <- runif(1e6)
y <- list(x, x, x)
```

3. В какой строке кода создается копия объекта a?

```
a <- c(1, 5, 3, 2)
b <- a
b[[1]] <- 10
```

Структура главы

- В разделе 2.2 мы поговорим об отличиях между именами и значениями, а также увидим, как оператор <- осуществляет связывание, или ссылку, между именем и значением.
- В разделе 2.3 рассматриваются случаи, когда R создает копии объектов: всякий раз, когда вы модифицируете вектор, вы с большой долей вероятности инициируете создание его копии. В этом разделе вы научитесь использовать функцию tracemem() для определения того, создается ли в действительности копия объекта. Также вы узнаете о последствиях создания копий объектов применительно к вызовам функций, спискам, датафреймам и символьным векторам.
- В разделе 2.4 мы рассмотрим предпосылки первых двух разделов главы в отношении занимаемой объектами памяти. Ваша интуиция в плане расхода памяти может очень сильно вас подводить, а функция utils::object.size(), к сожалению, может показывать неправильные результаты, так что мы научимся пользоваться другой функцией – lobstr::obj_size().
- В разделе 2.5 описываются два важнейших исключения из правила копирования при изменении: в случае с окружениями и значениями с одним именем объекты действительно изменяются на месте.
- В разделе 2.6 мы завершим главу разговором о сборщике мусора, освобождающем память от объектов, на которые больше нет ссылок по имени.

Требования

Для разбора внутреннего устройства объектов в R мы воспользуемся пакетом lobstr (https://github.com/r-lib/lobstr).

```
library(lobstr)
```

Источники

Подробности управления памятью в R не содержатся в виде документации где-то в едином хранилище. Большинство советов из этой главы были почерпнуты из различных источников, включая документацию (в частности, ?Memory и ?gc), раздел, посвященный профилированию памяти, из документа *Writing R extensions [R Core Team, 2018a]* (https://cran.r-project.org/doc/manuals/R-exts.html#Profiling-R-code-for-memory-use) и секции документа *R internals [R Core Team, 2018b]*, посвященной SEXPs (https://cran.r-project.org/doc/

manuals/R-ints.html#SEXPs). Остальное я извлек, читая исходный код на C, ставя небольшие эксперименты и задавая вопросы на R-devel. За все возможные ошибки в книге отвечаю я лично.

2.2 Основы связывания

Рассмотрим простой код, приведенный ниже:

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

По-простому эту инструкцию можно прочитать так: «создать объект с именем x, содержащий значения 1, 2 и 3». К сожалению, такое упрощение приводит к неправильному пониманию того, что R делает за кулисами. Если выражаться более точно, этот фрагмент кода выполняет следующие две вещи:

- создает объект, вектор значений, `c(1, 2, 3)`;
- связывает созданный объект с именем `x`.

Иначе говоря, у самого объекта, или значения, нет имени. Наоборот, у имени есть значение.

Для большей наглядности взгляните на рис. 2.1.

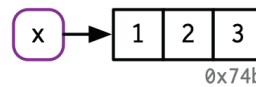


Рис. 2.1 Связывание имен и значений

Имя (name), `x`, показано здесь слева в квадратике со скругленными углами. Стрелка от него идет (связывает, ссылается) к *значению* (value), коим является вектор `c(1, 2, 3)`. Обратите внимание, что направление стрелки здесь противоположно направлению стрелки в коде: оператор `<-` выполняет *связывание* (binding) от имени, располагающегося слева, к объекту, находящемуся справа.

Таким образом, мы можем думать об имени как о *ссылке* (reference) на значение. К примеру, если вы запустите код, показанный ниже, вы не получите на выходе еще одну копию значения `c(1, 2, 3)`. Вместо этого будет выполнено еще одно связывание с существующим объектом, как показано на рис. 2.2:

```
y <- x
```

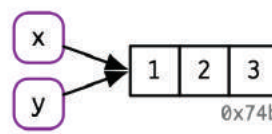


Рис. 2.2 Два имени указывают на одно и то же значение

Вы могли заметить на рис. 2.2, что значению `c(1, 2, 3)` присвоена метка `0x74b`. Хотя у векторов нет имен, мне время от времени нужно будет ссылаться на них независимо от их привязки. Для этого я буду снабжать значения уникальными идентификаторами. Внешний вид этих идентификаторов может напоминать адреса объектов в памяти. Но поскольку физические адреса меняются каждый раз при запуске кода, мы будем вместо них использовать идентификаторы.

К идентификатору объекта можно получить доступ с помощью функции `lobstr::obj_addr()`. Код, показанный ниже, дает понять, что имена `x` и `y` указывают на один и тот же идентификатор:

```
obj_addr(x)
#> [1] "0x7f8e16850e58"
obj_addr(y)
#> [1] "0x7f8e16850e58"
```

Эти идентификаторы слишком длинные и меняются каждый раз при перезапуске R.

Вам может потребоваться какое-то время, чтобы привыкнуть к различиям между именами и значениями, но когда вы это прочувствуете, вам будет значительно легче освоиться в функциональном программировании, где функции могут иметь разные имена в разных контекстах.

2.2.1 Синтаксически неправильные имена

В языке R существуют строгие правила, которым должны отвечать имена. Синтаксически правильные имена должны состоять из букв[1], цифр, точек и символов подчеркивания, но при этом не могут начинаться с символа подчеркивания или точки. Кроме того, вы не можете использовать ни одно из *зарезервированных слов* (reserved words), таких как `TRUE`, `NULL`, `if` или `function`. Полный список зарезервированных слов можно найти с помощью инструкции `?Reserved`. Имена, не отвечающие приведенным выше правилам, считают синтаксически неверными, и если вы попытаетесь их использовать, то получите ошибку, как показано ниже:

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

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

[1] Интересно, что состав буквенных символов в R определяется с учетом текущей локали. Это означает, что синтаксис в R может отличаться от компьютера к компьютеру, и файл, прекрасно работающий на одной машине, может даже не пройти синтаксический анализ на другой. Во избежание проблем с синтаксисом я советую придерживаться при написании имен, насколько это возможно, таблицы символов ASCII, т. е. использовать латинские буквы.

При этом вы можете нарушать эти правила и использовать в именах любые символы при условии обрамления их знаками обратного апострофа (`), как показано ниже:

```
`_abc` <- 1
`_abc`
#> [1] 1

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

Хотя вам вряд ли вздумается создавать такие странные имена, вам необходимо уметь работать с ними на случай, если столкнетесь с ними в источниках, созданных за пределами R.

Вы также можете создавать синтаксически неправильные связывания при помощи одиночных или двойных кавычек (например, `"_abc" <- 1`) вместо использования обратных апострофов, но вам не следует этого делать, поскольку в этом случае вам придется использовать другой синтаксис для извлечения значений. Возможность использования строк в левой части выражений с присваиванием – это исторический артефакт, к помощи которого прибегали до появления в R поддержки символа обратного апострофа.

2.2.2 Упражнения

1. Опишите взаимосвязь между `a`, `b`, `c` и `d` в приведенном ниже коде:

   ```
   a <- 1:10
   b <- a
   c <- b
   d <- 1:10
   ```

2. Ниже показано несколько способов доступа к функции `mean`. Все ли они обращаются к одному и тому же объекту функции? Проверьте это с помощью функции `lobstr::obj_addr()`:

   ```
   mean
   base::mean
   get("mean")
   evalq(mean)
   match.fun("mean")
   ```

3. По умолчанию функции импорта данных из базового пакета R, такие как `read.csv()`, автоматически конвертируют синтаксически неправильные имена в синтаксически правильные. Почему это может быть сопряжено с проблемами? Какие есть варианты для корректировки такого поведения?

4. Какие правила использует функция `make.names()` при конвертации синтаксически неправильных имен в синтаксически правильные?
5. Я немного упростил правила, регулирующие синтаксически правильные имена. Почему `.123e1` не является синтаксически корректным именем? Воспользуйтесь справкой `?make.names` для получения полной информации.

2.3 Копирование при изменении

Рассмотрим следующий код. Здесь мы выполняем привязку имен `x` и `y` к одному и тому же значению, после чего изменяем `y`[1].

```
x <- c(1, 2, 3)
y <- x

y[[3]] <- 4
x
#> [1] 1 2 3
```

Очевидно, что изменение `y` не затронуло `x`. Что же произошло с нашим совместным связыванием, которое мы видели на рис. 2.2? Хотя значение, ассоциированное с вектором `y`, изменилось, исходный объект `x` остался прежним. Дело в том, что R создал абсолютно новый объект с идентификатором `0xcd2` (что показано на рис. 2.3), являющийся копией объекта с идентификатором `0x74b` с одним измененным элементом внутри, после чего осуществил привязку имени `y` к этому новому объекту.

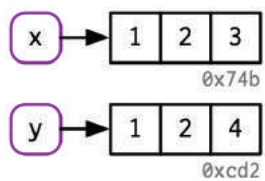


Рис. 2.3 Имя `y` теперь привязано к новому значению

Такое поведение известно как *копирование при изменении* (copy-on-modify). Понимание этой концепции поможет вам лучше и быстрее разобраться с тем, что стоит за производительностью кода на R. Иногда такое поведение описывается тем, что объекты в R являются *неизменяемыми* (immutable). Я не

[1] Вы можете удивиться, увидев оператор `[[`, использованный для извлечения подмножества из числового вектора. Мы вернемся к этому в разделе 4.3, но, если коротко, я думаю, вам всегда следует использовать оператор `[[` при извлечении или изменении одного элемента в векторе.

сторонник данного термина по причине того, что существует пара важных исключений из принципа копирования при изменении, о которых мы поговорим в разделе 2.5.

При интерактивной проверке концепции копирования при изменении будьте готовы к тому, что в RStudio вы получите другие результаты. Причина в том, что для отображения информации об объектах на панели окружения нужно, чтобы на каждый из них были ссылки. Эта особенность может помешать интерактивному отслеживанию данного принципа, но не влияет на работу функций и производительность при анализе данных. Для подобных экспериментов я рекомендую либо запускать R напрямую из терминала, либо пользоваться `RMarkdown`, как в этой книге.

2.3.1 tracemem()

Помощь в отслеживании создания копий объектов вам может оказать удобная функция *base*::`tracemem()`. При вызове с объектом в качестве аргумента она возвращает его текущий адрес в памяти:

```
x <- c(1, 2, 3)
cat(tracemem(x), "\n")
#> <0x7f80c0e0ffc8>
```

Далее запущенная функция будет осуществлять отслеживание заданного объекта и выводить сообщение об изменении адреса в памяти всякий раз при создании его копии. При этом вместе с новым адресом объекта будет показан и его прежний адрес, что видно в следующем примере:

```
y <- x
y[[3]] <- 4L
#> tracemem[0x7f80c0e0ffc8 -> 0x7f80c4427f40]:
```

Если еще раз изменить объект `y`, он не будет скопирован. Это происходит из-за того, что к новому объекту у нас на данный момент привязано одно имя. В результате срабатывает оптимизация, известная как *изменение на месте* (modify-in-place). Мы вернемся к этому вопросу в разделе 2.5.

```
y[[3]] <- 5L
untracemem(y)
```

Функция `untracemem()` является полной противоположностью `tracemem()` – она отменяет отслеживание переданного объекта.

2.3.2 Вызовы функций

Те же правила копирования применяются и в отношении вызовов функций. Рассмотрим следующий код:

```
f <- function(a) {
    a
}

x <- c(1, 2, 3)
cat(tracemem(x), "\n")
#> <0x7f8e10fd5d38>

z <- f(x)
# Здесь копия не создается

untracemem(x)
```

При запуске функции `f()` имя `a` внутри нее указывает на то же значение, что и `x` за пределами функции, что видно на рис. 2.4.

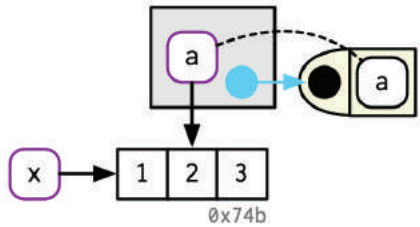


Рис. 2.4 Имена `a` и `x` указывают на один и тот же объект

В разделе 7.4.4 вы подробнее узнаете о соглашениях, используемых на этом рисунке. Если кратко, функция `f()` отображается в виде желтого объекта справа. У нее есть *формальный параметр* (formal argument) `a`, который обретает связывание, обозначенное черной пунктирной линией, в среде выполнения (серый прямоугольник) при запуске функции.

После завершения работы функции `f()` имена `x` и `z` будут ссылаться на один и тот же объект, как показано на рис. 2.5. При этом копии объекта с идентификатором `0x74b` создаваться не будут, поскольку сам объект не меняется. Если бы функция `f()` изменяла объект `x`, была бы создана его копия, после чего имя `z` было бы привязано к новому объекту.

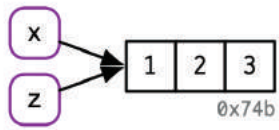


Рис. 2.5 Имена `x` и `z` ссылаются на один и тот же объект

2.3.3 Списки

На значения могут ссылаться не только имена, т. е. *переменные* (variable), но и элементы *списков* (list). Рассмотрим следующий список, внешне очень напоминающий числовой вектор, с которым мы работали ранее:

```
l1 <- list(1, 2, 3)
```

Но список – это на самом деле более сложная структура, хранящая не сами значения, а ссылки на них, что видно на рис. 2.6.

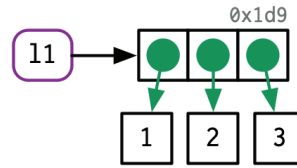


Рис. 2.6 Элементы списка ссылаются на значения

Это особенно важно понимать при изменении списков:

```
l2 <- l1
```

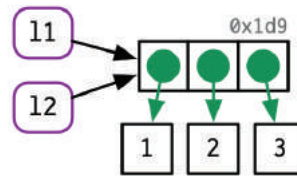


Рис. 2.7 Два имени ссылаются на один список элементов

```
l2[[3]] <- 4
```

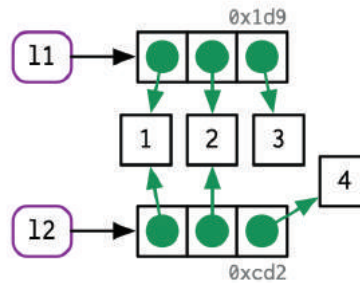


Рис. 2.8 Создание копии при модифицировании списка

Как и в случае с векторами, при работе со списками используется принцип копирования при изменении. Исходный список остается нетронутым, а рядом создается его копия. Но это так называемая *поверхностная копия* (shallow copy): объект списка и его привязки копируются, а сами значения, на которые ссылаются элементы списка, – нет. Противоположностью создания поверхностной копии в R является создание *глубокой копии* (deep copy), при котором производится копирование всего содержимого всех ссылок. Вплоть до версии R 3.1.0 все копии объектов были глубокими.

Чтобы увидеть значения, совместно используемые разными списками, можно воспользоваться функцией `lobstr::ref()`. Эта функция выводит адреса памяти для каждого объекта вместе с локальными идентификаторами, чтобы вы могли перекрестно ссылаться на общие компоненты.

```
ref(l1, l2)
#> ▉ [1:0x7f8e161c9198] <list>
#> ├─[2:0x7f8e1689c178] <dbl>
#> ├─[3:0x7f8e1689c140] <dbl>
#> └─[4:0x7f8e1689c108] <dbl>
#>
#> ▉ [5:0x7f8e162d0c98] <list>
#> ├─[2:0x7f8e1689c178]
#> ├─[3:0x7f8e1689c140]
#> └─[6:0x7f8e163423a0] <dbl>
```

2.3.4 Датафреймы

Датафреймы (data frames) представляют собой списки векторов, так что общий принцип копирования при изменении играет важную роль при изменении датафреймов. Рассмотрим в качестве примера приведенный ниже и на рис. 2.9 датафрейм:

```
d1 <- data.frame(x = c(1, 5, 6), y = c(2, 4, 3))
```

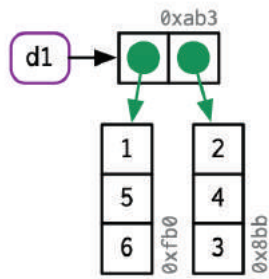


Рис. 2.9 Исходный датафрейм

При изменении колонки подвергнуться модификации должна только она, в то время как другие колонки должны сохранить свои изначальные ссылки, что видно на рис. 2.10:

```
d2 <- d1
d2[, 2] <- d2[, 2] * 2
```

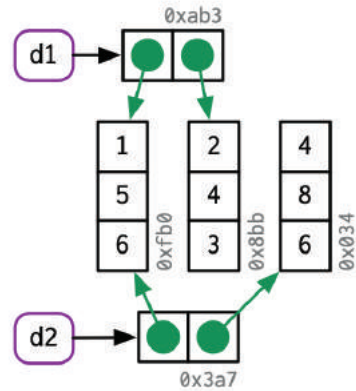


Рис. 2.10 Измененная колонка ссылается на новый вектор значений

Вместе с тем при модификации строки все колонки должны подвергнуться изменениям, что приводит к созданию их копий, как видно на рис. 2.11:

```
d3 <- d1
d3[1, ] <- d3[1, ] * 3
```

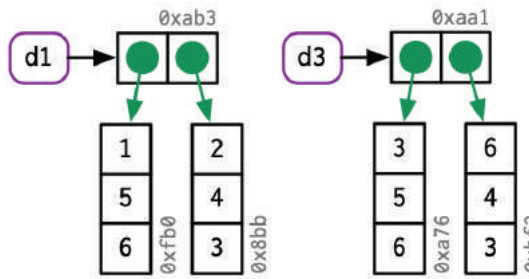


Рис. 2.11 Последствие изменения строки – создание копий всех колонок датафрейма

2.3.5 Символьные векторы

Ссылки в R используются также при работе с *символьными векторами* (character vector)[1]. Обычно я изображаю символьные векторы так, как показано на рис. 2.12:

```
x <- c("a", "a", "abc", "d")
```

[1] Вы удивитесь, но символьные векторы содержат целые строки, а не отдельные символы.

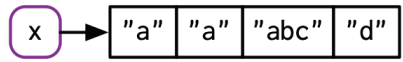


Рис. 2.12 Символьный вектор

Но это скорее пример добровольного заблуждения, поскольку R на самом деле использует *глобальный пул строк* (global string pool), в котором каждый элемент символьного вектора является указателем на уникальную строку в пуле, как показано на рис. 2.13.

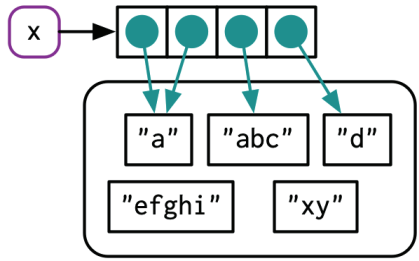


Рис. 2.13 Глобальный пул строк

Вы можете воспользоваться уже знакомой вам функцией `ref()` для отображения этих ссылок, установив при этом параметру `character` значение `TRUE`:

```
ref(x, character = TRUE)
#> ▉ [1:0x7f8e149ef008] <chr>
#> ├─[2:0x7f8e1094d598] <string: "a">
#> ├─[2:0x7f8e1094d598]
#> ├─[3:0x7f8e152e21c8] <string: "abc">
#> └─[4:0x7f8e10b0e6b0] <string: "d">
```

Такой принцип хранения оказывает существенное влияние на объем занимаемой символьными векторами памяти, но в остальном это не так важно, так что на протяжении этой книги я буду обращаться с символьными векторами так, как будто строки располагаются в самих векторах.

2.3.6 Упражнения

1. Почему в инструкции `tracemem(1:10)` нет никакого смысла?
2. Объясните, почему при запуске кода, показанного ниже, функция `tracemem()` показывает две копии? Подсказка: сравните приведенный здесь код с кодом, показанным выше в этом разделе:

```
x <- c(1L, 2L, 3L)
tracemem(x)

x[[3]] <- 4
```

3. Опишите взаимосвязи между следующими объектами:

```
a <- 1:10
b <- list(a, a)
c <- list(b, a, 1:10)
```

4. Что произойдет при запуске следующего фрагмента кода?

```
x <- list(1:10)
x[[2]] <- x
```

Изобразите результат графически.

2.4 Размеры объектов

Узнать объем памяти, занимаемой объектом, можно при помощи функции `lobstr::obj_size()`[1]:

```
obj_size(letters)
#> 1,712 B
obj_size(ggplot2::diamonds)
#> 3,456,344 B
```

Так как элементы списка ссылаются на значения, размер списка может оказаться гораздо меньше, чем вы ожидаете:

```
x <- runif(1e6)
obj_size(x)
#> 8,000,048 B

y <- list(x, x, x)
obj_size(y)
#> 8,000,128 B
```

Объект `y` занимает всего на 80 байт больше, чем объект `x`[2]. Это размер пустого списка с тремя элементами:

```
obj_size(list(NULL, NULL, NULL))
#> 80 B
```

По причине использования в R глобального пула строк символьные векторы также могут занимать гораздо меньше места в памяти в сравнении

[1] Остерегайтесь использования функции `utils::object.size()`, поскольку она не умеет корректно учитывать размеры совместно используемых ссылок и возвращает слишком большие значения.

[2] Если вы запустите 32-битную версию R, вы увидите несколько иные цифры.

с ожиданиями. В результате 1000-кратный повтор строки в векторе не приведет к 1000-кратному увеличению занимаемой памяти.

```
banana <- "bananas bananas bananas"
obj_size(banana)
#> 136 B
obj_size(rep(banana, 100))
#> 928 B
```

Ссылки также затрудняют размышления о том, какой объем памяти в действительности занимают конкретные объекты. К примеру, `obj_size(x) + obj_size(y)` будет равно `obj_size(x, y)` только в том случае, если эти объекты не используют значения совместно. В нашем случае объединенный объем переменных `x` и `y` будет равен объему одной переменной `y`:

```
obj_size(x, y)
#> 8,000,128 B
```

В языке R начиная с версии 3.5.0 появилась интересная возможность хранения, которая может вас удивить, получившая название `ALTREP` (сокращение от *alternative representation*, альтернативное представление). Этот механизм позволяет R представлять векторы определенных типов в очень компактном виде. В основном эта концепция используется вместе с оператором двоеточия (`:`) – в этом случае R может хранить в памяти не все значения из последовательности, а лишь первое и последнее значения. В результате объемы приведенных ниже объектов будут одинаковыми:

```
obj_size(1:3)
#> 680 B
obj_size(1:1e3)
#> 680 B
obj_size(1:1e6)
#> 680 B
obj_size(1:1e9)
#> 680 B
```

2.4.1 Упражнения

1. Почему в приведенном ниже примере величины `object.size(y)` и `obj_size(y)` так кардинально отличаются? Обратитесь за помощью к документации функции `object.size()`.

```
y <- rep(list(runif(1e4)), 100)

object.size(y)
#> 8005648 bytes
obj_size(y)
#> 80,896 B
```

2. Рассмотрим следующий список. Почему его размер может вводить в заблуждение?

```
funs <- list(mean, sd, var)
obj_size(funs)
#> 17,608 B
```

3. Предугадайте вывод следующего фрагмента кода:

```
a <- runif(1e6)
obj_size(a)

b <- list(a, a)
obj_size(b)
obj_size(a, b)

b[[1]][[1]] <- 10
obj_size(b)
obj_size(a, b)

b[[2]][[1]] <- 10
obj_size(b)
obj_size(a, b)
```

2.5 Изменение на месте

Как мы уже видели ранее, в R модифицирование объектов обычно приводит к созданию их копии. Но есть из этого правила два исключения:

- объекты с единственной привязкой особым образом оптимизируются с точки зрения производительности;
- окружения, представляющие особый тип в R, всегда *изменяются на месте* (modify-in-place).

2.5.1 Объекты с единственной привязкой

Если к объекту привязано только одно имя, как на рис. 2.14, R будет изменять его на месте, без создания копии:

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

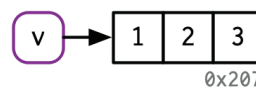


Рис. 2.14 Вектор с единственным ссылающимся на него именем

```
v[[3]] <- 4
```

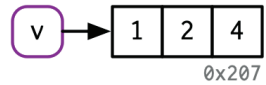


Рис. 2.15 Объект меняется на месте, без создания копии

Обратите внимание на рис. 2.15, что идентификатор объекта (`0x207`), на который ссылается имя `v`, не изменился.

Есть, однако, два момента, затрудняющих предсказание того, будет ли R применять к объекту показанную оптимизацию:

- что касается привязок, на текущий момент[1] R учитывает их как 0, 1 или много. Это означает, что если у объекта было две привязки, счетчик не вернется в состояние 1, поскольку много минус один – это по-прежнему много. В результате R будет создавать копии объекта, когда в этом нет никакой необходимости;
- при вызове подавляющего большинства функций R будет создавать ссылку на объект. Единственным исключением, пожалуй, являются примитивные функции на C. Они могут быть написаны только разработчиками R и в основном встречаются в базовом пакете.

Эти моменты серьезно усложняют жизнь программиста в плане предсказания того, будут ли в той или иной ситуации создаваться копии объектов. Так что лучше подстраховываться и действовать эмпирически, воспользовавшись функцией `tracemem()`.

Давайте рассмотрим один пример с применением цикла `for`. Считается, что традиционные циклы в R работают медленно, но зачастую проблема кроется в создании копий объектов на каждой итерации. Взгляните на приведенный ниже код. Здесь мы вычитаем медианное значение каждой колонки из всех значений в ней применительно к объемному датафрейму:

```
x <- data.frame(matrix(runif(5 * 1e4), ncol = 5))
medians <- vapply(x, median, numeric(1))

for (i in seq_along(medians)) {
    x[[i]] <- x[[i]] - medians[[i]]
}
```

Этот цикл будет выполняться на удивление долго, поскольку на каждой итерации будет создаваться отдельная копия датафрейма. В этом можно убедиться, воспользовавшись функцией `tracemem()`:

[1] К моменту, когда вы будете читать эту книгу, в данной области могут произойти изменения согласно планам по улучшению подсчета ссылок: https://developer.r-project.org/Refcnt.html.

```
cat(tracemem(x), "\n")
#> <0x7f80c429e020>

for (i in 1:5) {
    x[[i]] <- x[[i]] - medians[[i]]
}

#> tracemem[0x7f80c429e020 -> 0x7f80c0c144d8]:
#> tracemem[0x7f80c0c144d8 -> 0x7f80c0c14540]: [[<-.data.frame [[<-
#> tracemem[0x7f80c0c14540 -> 0x7f80c0c145a8]: [[<-.data.frame [[<-
#> tracemem[0x7f80c0c145a8 -> 0x7f80c0c14610]:
#> tracemem[0x7f80c0c14610 -> 0x7f80c0c14678]: [[<-.data.frame [[<-
#> tracemem[0x7f80c0c14678 -> 0x7f80c0c146e0]: [[<-.data.frame [[<-
#> tracemem[0x7f80c0c146e0 -> 0x7f80c0c14748]:
#> tracemem[0x7f80c0c14748 -> 0x7f80c0c147b0]: [[<-.data.frame [[<-
#> tracemem[0x7f80c0c147b0 -> 0x7f80c0c14818]: [[<-.data.frame [[<-
#> tracemem[0x7f80c0c14818 -> 0x7f80c0c14880]:
#> tracemem[0x7f80c0c14880 -> 0x7f80c0c148e8]: [[<-.data.frame [[<-
#> tracemem[0x7f80c0c148e8 -> 0x7f80c0c14950]: [[<-.data.frame [[<-
#> tracemem[0x7f80c0c14950 -> 0x7f80c0c149b8]:
#> tracemem[0x7f80c0c149b8 -> 0x7f80c0c14a20]: [[<-.data.frame [[<-
#> tracemem[0x7f80c0c14a20 -> 0x7f80c0c14a88]: [[<-.data.frame [[<-

untracemem(x)
```

Фактически на каждой итерации мы создаем не одну и даже не две копии датафрейма, а целых три! Две копии создаются при помощи функции `[[.data.frame`, а еще одна – по причине того, что `[[.data.frame` представляет собой обычную функцию, увеличивающую на единицу количество ссылок на `x`[1].

Уменьшить количество создаваемых копий можно, воспользовавшись списком вместо датафрейма. Модифицирование списка выполняется с участием внутреннего кода на C, в результате чего количество ссылок не увеличивается, а создается только одна копия:

```
y <- as.list(x)
cat(tracemem(y), "\n")
#> <0x7f80c5c3de20>

for (i in 1:5) {
    y[[i]] <- y[[i]] - medians[[i]]
}
#> tracemem[0x7f80c5c3de20 -> 0x7f80c48de210]:
```

[1] Эти копии являются поверхностными: при их создании копируются только ссылки на колонки, но не их содержимое. Это хорошая новость в отношении производительности, но не лучшая из всех возможных.

Определить, создаются ли копии объектов, бывает гораздо проще, чем предотвратить их создание. Если вам приходится прибегать к совсем уж экзотическим способам устранения создания копий объектов, возможно, пришло время переписать свои функции с использованием языка C++, о чем мы будем говорить в главе 25.

2.5.2 Окружения

Подробно об *окружениях* (environments) мы будем говорить в главе 7, но я считаю, что важно упомянуть о них и здесь, поскольку их поведение отличается от всех других объектов тем, что окружения всегда изменяются на месте. Иногда это свойство называют *ссылочной семантикой* (reference semantics), поскольку при модифицировании окружения все существующие привязки к нему сохраняют свои прежние ссылки.

Давайте создадим окружение и свяжем его с именами `e1` и `e2`, как показано на рис. 2.16:

```
e1 <- rlang::env(a = 1, b = 2, c = 3)
e2 <- e1
```

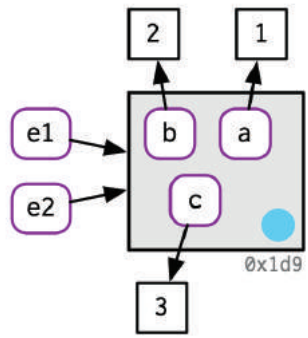


Рис. 2.16 Два имени ссылаются на одно окружение

Если мы изменим привязку, окружение будет модифицировано на месте, что видно на рис. 2.17:

```
e1$c <- 4
e2$c
#> [1] 4
```

Эта идея может быть использована при создании функций, «запоминающих» свое предыдущее состояние. В разделе 10.2.4 мы будем говорить об этом подробнее. Это свойство также используется при реализации объектно ориентированной системы R6, которой будет посвящена глава 14.

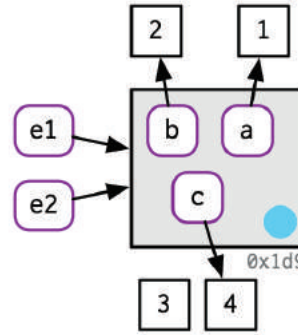


Рис. 2.17 Изменение окружения на месте

Одним из следствий данного свойства является то, что окружения могут содержать сами себя, как показано на рис. 2.18:

```
e <- rlang::env()
e$self <- e

ref(e)
#> ▊ [1:0x7f8e18918480] <env>
#> └─self = [1:0x7f8e18918480]
```

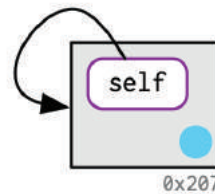


Рис. 2.18 Окружение, содержащее само себя

Это уникальное свойство окружений!

2.5.3 Упражнения

1. Объясните, почему в результате запуска следующего кода не создается *циклический список* (circular list):

```
x <- list()
x[[1]] <- x
```

2. Оберните два метода для вычитания медианы в две функции, после чего воспользуйтесь пакетом `bench` [Эстер (Hester), 2018] для тщательного сравнения скорости их работы. Как изменяется быстродействие с ростом количества колонок?

3. Что будет, если воспользоваться функцией `tracemem()` применительно к окружениям?

2.6 Отвязывание и сборщик мусора

Давайте рассмотрим следующий код, иллюстрация которого приведена на рис. 2.19:

```
x <- 1:3
```

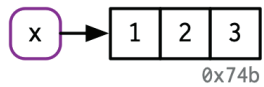


Рис. 2.19 Вектор x

Теперь так, как показано ниже и на рис. 2.20:

```
x <- 2:4
```

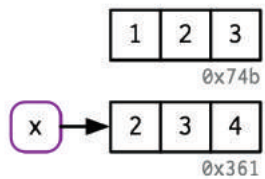


Рис. 2.20 Имя x указывает на новый объект

И в завершение так (рис. 2.21):

```
rm(x)
```

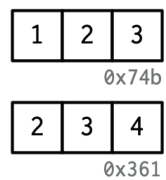


Рис. 2.21 Ни один из объектов не имеет имени

Мы создали два объекта, но к моменту завершения работы кода ни один из них не оказался привязан к соответствующим именам. Как эти объекты удаляются из памяти? Для этого существует *сборщик мусора* (garbage collector – GC). Он освобождает память, удаляя объекты в R, которые больше не

используются, и запрашивает больше памяти у операционной системы при необходимости.

В R используется *отслеживающий* (tracing) сборщик мусора. Это означает, что он сам следит за всеми объектами, доступными в глобальном окружении[1], и объектами, к которым можно получить доступ через эти объекты, т. е. ссылки в списках и окружениях просматриваются рекурсивно. Сборщик мусора не пользуется счетчиком ссылок механизма изменения на месте, о котором мы говорили выше. Хотя эти две идеи тесно связаны, используемые в них внутренние структуры данных оптимизированы для разных целей.

Сборщик мусора запускается автоматически всякий раз, когда R требуется получить больше памяти для создания объекта. Со стороны бывает трудно предсказать, когда именно запустится сборщик. Но вы даже не должны пытаться этого делать. Если вам интересно, когда стартует сборщик мусора, вы можете воспользоваться функцией `gcinfo(TRUE)`, которая будет выводить сообщение в консоль при каждом запуске сборщика.

Вы можете принудительно запустить сборщик мусора с помощью функции `gc()`. Однако, что бы вы ни прочитали в других источниках, нет ни малейшей необходимости делать это вручную. Единственный случай, когда вам может понадобиться прибегнуть к помощи этой функции, – это если вы хотите вернуть память операционной системе, чтобы другие программы могли ей воспользоваться, или желаете узнать, сколько памяти используется в данный момент:

```
gc()
#>      used (Mb)      gc trigger (Mb) limit (Mb) max used (Mb)
#> Ncells  720514 38.5    1329171   71         NA  1329171   71
#> Vcells 5455049 41.7   19041363  145      16384 15603142  119
```

Функция `lobstr::mem_used()` является оберткой функции `gc()` и выводит на экран общее количество использованных байтов:

```
mem_used()
#> 83,989,184 B
```

Это число не будет совпадать со счетчиком памяти, содержащимся в операционной системе. Причин на то три.

1. Сюда включаются объекты, созданные R, а не интерпретатором R.
2. R и операционная система работают в отложенном режиме: они не будут возвращать память, пока она действительно не потребуется. Таким образом, память в R может оставаться зарезервирована просто потому, что операционная система пока не потребовала ее вернуть.
3. R учитывает память, занятую объектами, но при этом в памяти могут присутствовать пропуски из-за удаленных объектов. Эта проблема известна как фрагментация памяти (memory fragmentation).

[1] И любыми другими окружениями в текущем стеке вызовов.

2.7 Ответы на контрольные вопросы

1. Вы должны обрамлять синтаксически неправильные имена с помощью обратных апострофов (`). К примеру, это касается переменных с именами 1, 2 и 3.

```
df <- data.frame(runif(3), runif(3))
names(df) <- c(1, 2)

df$`3` <- df$`1` + df$`2`
```

2. Переменная будет занимать порядка 8 Мб.

```
x <- runif(1e6)
y <- list(x, x, x)
obj_size(y)
#> 8,000,128 B
```

3. Объект a будет скопирован в момент изменения объекта b, т. е. в строке `b[[1]] <- 10`.

3

Векторы

3.1 Введение

В данной главе мы обсудим наиболее важное семейство типов данных в базовом R – *векторы* (vector)[1]. Хотя вы уже наверняка сталкивались со многими (если не со всеми) типами векторов, вероятно, вы не задумывались о том, как именно они взаимосвязаны. В этой главе мы не будем подробно останавливаться на отдельных типах векторов, но я покажу вам, как члены этого семейства уживаются друг с другом. Если вам понадобится больше информации, вы всегда можете найти ее в документации к R.

Векторы в общем виде подразделяются на *атомарные векторы* (atomic vector) и *списки* (list)[2]. Отличаются они тем, что атомарные векторы могут хранить только элементы одного типа, а списки – разных. По сути не являясь вектором, значение `NULL` тесно связано с векторами и часто выполняет роль обобщенного вектора нулевой длины. На диаграмме, показанной на рис. 3.1, которую мы будем расширять с течением главы, показаны базовые связи между векторами.

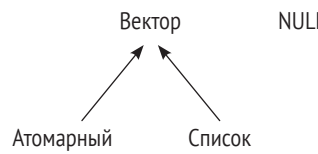


Рис. 3.1 Взаимосвязь между типами векторов

Каждый вектор может иметь свои *атрибуты* (attribute), которые можно воспринимать как именованный список произвольных метаданных. При этом два из этих атрибутов особенно важны. Речь идет об атрибуте `dimension`,

---

[1] Совокупно все остальные типы в R называются *типами узлов* (node type), и к ним причисляются такие типы, как функции, окружения и т. д. С этим глубоко техническим термином, например, можно встретиться при работе с функцией сборщика мусора `gc()`: буква `N` в слове `Ncells` в выводе этой функции означает узлы (nodes), а `V` в `Vcells` – векторы.

[2] В некоторых разделах документации к R списки именуются *обобщенными векторами* (generic vector), чтобы подчеркнуть их отличие от атомарных векторов.

с помощью которого можно представлять векторы в виде матриц и массивов, и атрибуте `class`, лежащем в основе объектной системы S3. Хотя подробно о системе S3 мы будем говорить в главе 13, здесь мы перечислим наиболее важные из векторов S3, коими являются факторы, дата и время, датафреймы и тибблы. Несмотря на то что двумерные структуры данных, такие как матрицы и датафреймы, – это далеко не первое, что приходит в голову при мысли о векторах, вы узнаете, что в R эти объекты также воспринимаются как векторы.

Контрольные вопросы

Попробуйте ответить на приведенные ниже вопросы, чтобы понять, стоит ли вам тратить время на чтение этой главы. Если вы с легкостью ответили на все вопросы, можете пропустить главу и двигаться дальше. Ответы приведены в разделе 3.8.

1. Какие четыре распространенных типа атомарных векторов существуют? Назовите еще два редких типа.
2. Что такое атрибуты? Как их можно устанавливать и считывать?
3. Чем список отличается от атомарного вектора? А чем матрица отличается от датафрейма?
4. Может ли список являться матрицей? Может ли в датафрейме присутствовать колонка, представленная матрицей?
5. Чем поведение тибблов отличается от поведения датафреймов?

Структура главы

- В разделе 3.2 мы познакомимся с различными типами атомарных векторов: логическими, целочисленными, двойной точности и символьными. Это простейшие структуры данных в R.
- В разделе 3.3 мы сделаем небольшой экскурс в мир атрибутов – гибкое представление метаданных в R. Наиболее важные атрибуты отвечают за имена, размер объекта и его класс.
- Раздел 3.4 будет посвящен важным типам векторов, создаваемым посредством объединения атомарных векторов со специальными атрибутами. К ним относятся факторы, дата и время и длительности.
- В разделе 3.5 мы погрузимся в мир списков. Списки очень похожи на атомарные векторы, за исключением одного важного отличия: они могут содержать элементы разных типов, включая списки. Это делает их пригодными для представления иерархий.
- В разделе 3.6 мы познакомимся с датафреймами и тибблами, которые используются для представления данных в табличном виде. Эти типы данных объединяют в себе поведение списков и матриц, что повышает их ценность при обработке статистических данных.

3.2 Атомарные векторы

Существует четыре основных типа атомарных векторов: *логический* (logical), *целочисленный* (integer), *двойной точности* (double) и *символьный* (character), содержащий строки. Вместе целочисленные векторы и векторы двойной точности составляют тип *числовых векторов* (numeric vectors)[1]. Есть еще два редко используемых типа векторов: вектор *комплексных чисел* (complex) и вектор *сырых данных*, или байтовых последовательностей (raw). Мы не будем подробно говорить о них, поскольку комплексные числа довольно редко встречаются в области статистики, а вектор байтовых последовательностей может пригодиться разве что для обработки двоичных данных. Общая схема типов векторов в R представлена на рис. 3.2.

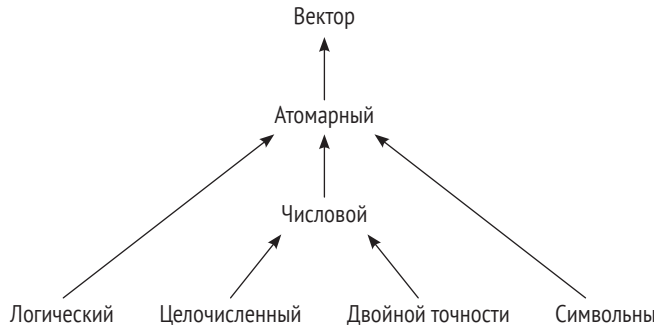


Рис. 3.2 Иерархия типов векторов в R

3.2.1 Скаляры

Каждый из четырех перечисленных выше типов векторов обладает особым синтаксисом для создания отдельных значений, или скаляров[2]:

- логические значения могут быть записаны в полной (`TRUE` или `FALSE`) или сокращенной (`T` или `F`) форме;
- числа с двойной точностью могут записываться в десятичном (`0.1234`), научном (`1.23e4`) или шестнадцатеричном (`0xcafe`) виде. Существует три особых значения, относящихся к типу двойной точности: `Inf`, `-Inf` и `NaN` (не число). Эти значения определены в стандарте чисел с плавающей запятой;

[1] Это определенное упрощение, поскольку в R не используется числовой тип данных единообразно.

[2] Чисто технически в R не поддерживаются скаляры. Вместо этого все значения, выглядящие как скаляры, в действительности представляют собой векторы единичной длины. Это больше теоретическое отличие, но оно объясняет, почему работают выражения вроде `1[1]`.

- целочисленные значения записываются аналогично числам с двойной точностью, но завершаются заглавной латинской буквой `L`[1] (`1234L`, `1e4L` или `0xcafeL`) и не могут содержать дробную часть;
- строки в R обрамляются двойными (") или одинарными (') кавычками (`"hi"` или `'bye'`). Специальные символы экранируются при помощи обратного слеша (\). За подробностями можно обратиться к документации, вызываемой командой `?Quotes`.

3.2.2 Создание длинных векторов с помощью функции c()

Для создания более длинных векторов можно воспользоваться функцией `c()` (от англ. *combine* – объединять):

```
lgl_var <- c(TRUE, FALSE)
int_var <- c(1L, 6L, 10L)
dbl_var <- c(1, 2.5, 4.5)
chr_var <- c("these are", "some strings")
```

При поступлении на вход атомарных векторов функция `c()` всегда создает на их основе другой атомарный вектор, как бы выравнивая переданные ей аргументы:

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

На диаграммах я буду обозначать векторы в виде соединенных прямоугольников, так что показанный выше код можно графически выразить так, как показано на рис. 3.3.

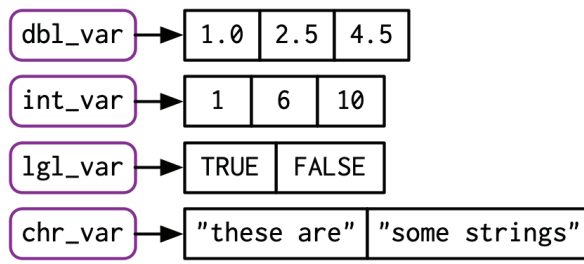


Рис. 3.3 Графическое представление векторов

[1] Выбор для обозначения целочисленных значений буквы `L` кажется не самым интуитивно понятным. Откуда это пошло? Когда этот суффикс стал использоваться в R, в этом языке целочисленный тип данных был эквивалентен длинным целым числам (long integer) в языке C, где суффикс `l` или `L` мог использоваться для обозначения таких чисел. Разработчики R посчитали, что строчная буква `l` визуально сильно напоминает литеру `i`, используемую для обозначения комплексных чисел, в результате чего было решено остановиться на прописной `L`.

Определить тип вектора можно при помощи функции `typeof()`[1], а его длину – посредством функции `length()`.

```
typeof(lgl_var)
#> [1] "logical"
typeof(int_var)
#> [1] "integer"
typeof(dbl_var)
#> [1] "double"
typeof(chr_var)
#> [1] "character"
```

3.2.3 Пропущенные значения

В языке R пропущенные, или отсутствующие, значения обозначаются с помощью особого значения `NA` (от англ. *not applicable* – неприменимо). Пропущенные значения в R обладают определенной вирусностью, поскольку большинство вычислений с их участием также возвращают пропущенное значение.

```
NA > 5
#> [1] NA
10 * NA
#> [1] NA
!NA
#> [1] NA
```

Из этого правила есть, однако, несколько исключений, возникающих вследствие некой идентичности, сохраняющейся для всех возможных входных значений:

```
NA ^ 0
#> [1] 1
NA | TRUE
#> [1] TRUE
NA & FALSE
#> [1] FALSE
```

Стремление к размножению отсутствующих значений зачастую приводит к ошибкам при определении наличия в векторе значений `NA`:

```
x <- c(NA, 5, NA, 10)
x == NA
#> [1] NA NA NA NA
```

---

[1] Вы также могли слышать о функциях `mode()` и `storage.mode()`. Их использовать не стоит, поскольку они присутствуют в языке только для обратной совместимости с языком S.

Полученный результат абсолютно корректен (хотя кого-то он может удивить), поскольку нет никаких оснований утверждать, что одно пропущенное значение равно другому. Вместо оператора равенства в данном случае более уместно использовать функцию `is.na()` для определения вхождения отсутствующих значений:

```
is.na(x)
#> [1] TRUE FALSE TRUE FALSE
```

Примечание: чисто технически существует четыре разных обозначения пропущенных значений для каждого типа атомарных векторов: `NA` (логический), `NA_integer_` (целочисленный), `NA_real_` (с двойной точностью) и `NA_character_` (символьный). Обычно это не имеет значения, поскольку значение `NA` будет автоматически приведено к нужному типу при необходимости.

3.2.4 Определение и приведение типов векторов

Обычно проверка на тип вектора осуществляется при помощи функций группы `is.*()`. Но их следует использовать с большой осторожностью. Функции `is.logical()`, `is.integer()`, `is.double()` и `is.character()` делают ровно то, что и должны, а именно проверяют, является ли вектор логическим, целочисленным, двойной точности или символьным соответственно. Избегайте использования функций `is.vector()`, `is.atomic()` и `is.numeric()`: они не выполняют проверку на то, является ли объект вектором, атомарным вектором или числовым вектором. Внимательно ознакомьтесь с документацией по этим функциям, чтобы понять, что в действительности они делают.

Для атомарных векторов тип является свойством вектора в целом: все его элементы должны быть одного типа. При попытке использовать разные типы данных будет выполняться *приведение типов* (coercion) в следующем фиксированном порядке: символьный → двойной точности → целочисленный → логический. К примеру, если попробовать объединить вместе значения целочисленного и символьного типов, мы получим символьный вектор, как показано ниже:

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

Приведение типов зачастую выполняется автоматически. Большинство математических функций (`+`, `log`, `abs` и др.) выполняют приведение к числовому типу. Это бывает очень удобно применительно к логическим векторам, поскольку значения `TRUE` преобразуются в единицу, а `FALSE` – в ноль.

```
x <- c(FALSE, FALSE, TRUE)
as.numeric(x)
```

```
#> [1] 0 0 1

# Общее количество значений TRUE
sum(x)
#> [1] 1

# Доля значений TRUE
mean(x)
#> [1] 0.333
```

Обычно явное приведение типов выполняется с помощью функций группы `as.*()`, таких как `as.logical()`, `as.integer()`, `as.double()` и `as.character()`. В результате неудачного приведения символьных значений будет выдано предупреждение, показанное ниже, а соответствующие значения станут пропущенными:

```
as.integer(c("1", "1.5", "a"))
#> Warning: NAs introduced by coercion
#> [1] 1 1 NA
```

3.2.5 Упражнения

1. Как создать скаляры комплексных чисел и байтовых последовательностей (см. команды `?raw` и `?complex`)?
2. Проверьте свое знание правил приведения типов данных, предсказав вывод следующих вызовов функции `c()`:

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

3. Почему `1 == "1"` дает TRUE? Почему `-1 < FALSE` дает TRUE? Почему `"one" < 2` дает FALSE?
4. Почему отсутствующее значение по умолчанию `NA` – это логический вектор? Что особенного есть в логических векторах? Подсказка: подумайте о векторе `c(FALSE, NA_character_)`.
5. Какую именно проверку осуществляют функции `is.atomic()`, `is.numeric()` и `is.vector()`?

3.3 Атрибуты

Вы могли заметить, что в число атомарных векторов не входят такие распространенные структуры данных, как матрицы, массивы, факторы или дата со временем. Эти типы строятся поверх атомарных векторов путем добавления соответствующих атрибутов. В этом разделе мы познакомимся с основами

атрибутов (attribute), а также узнаем, как при помощи атрибута `dim` можно создавать матрицы и массивы. В следующем разделе вы узнаете, как можно использовать атрибут `class` для создания векторов S3, включая факторы, даты и даты со временем.

3.3.1 Получение и установка

Вы можете думать об атрибутах как о парах имя–значение[1], с помощью которых к объектам присоединяются метаданные. Значения отдельных атрибутов могут быть извлечены и изменены с помощью функции `attr()`. Для массового извлечения атрибутов можно воспользоваться функцией `attributes()`, а для массовой установки – функцией `structure()`.

```
a <- 1:3
attr(a, "x") <- "abcdef"
attr(a, "x")
#> [1] "abcdef"

attr(a, "y") <- 4:6
str(attributes(a))
#> List of 2
#>  $ x: chr "abcdef"
#>  $ y: int [1:3] 4 5 6

# Эквивалент
a <- structure(
    1:3,
    x = "abcdef",
    y = 4:6
)
str(attributes(a))
#> List of 2
#>  $ x: chr "abcdef"
#>  $ y: int [1:3] 4 5 6
```

Графическое отображение этого примера показано на рис. 3.4.

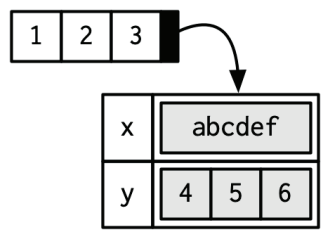


Рис. 3.4 Вектор с атрибутами

[1] Атрибуты ведут себя как именованные списки, фактически являясь при этом списками пар (pairlist). Списки пар функционально неотличимы от обычных списков, но внутренне устроены совсем иначе. Подробнее о них мы будем говорить в разделе 18.6.1.

К атрибутам по большей части нужно относиться как к эфемерным сущностям. К примеру, большинство атрибутов теряются при выполнении множества операций, как показано ниже:

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

Есть лишь два атрибута, которые стабильно сохраняются:

- `names` – символьный вектор с именами для всех элементов;
- `dim` – сокращенно от *dimensions* (размерности) – целочисленный вектор, используемый для превращения векторов в матрицы или массивы.

Для сохранения остальных атрибутов вам необходимо создавать собственные классы S3, о чем мы будем подробно говорить в главе 13.

3.3.2 Имена

Задать имена элементам вектора можно тремя разными способами:

```
# При создании:
x <- c(a = 1, b = 2, c = 3)

# Присвоив символьный вектор атрибуту names()
x <- 1:3
names(x) <- c("a", "b", "c")

# С помощью функции setNames():
x <- setNames(1:3, c("a", "b", "c"))
```

Избегайте использования выражения `attr(x, "names")` – оно более длинное и менее понятное по сравнению с `names(x)`. Избавить вектор от имен можно двумя способами: вызвав функцию `unname(x)` или присвоив атрибуту `names(x)` значение NULL.

Технически корректно изобразить именованный вектор можно так, как на рис. 3.5.

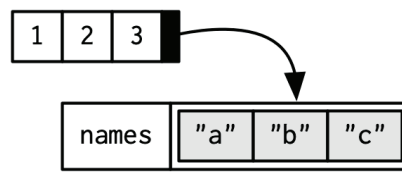


Рис. 3.5 Именованный вектор (версия 1)

В то же время имена элементов играют столь важную роль, что я предпочитаю прописывать их непосредственно под вектором, как на рис. 3.6, если моей целью не является привлечь внимание к структуре данных атрибутов.

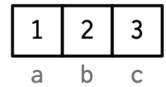


Рис. 3.6 Именованный вектор (версия 2)

Для корректного использования с символьными подмножествами (раздел 4.5.1) имена элементов должны быть уникальными, и в них не должно быть пропусков, но это не обязательно с точки зрения R. В зависимости от выбранного способа установки атрибута пропущенные имена могут быть представлены либо пустыми строками, либо значениями `NA_character_`. Если в векторе все имена пропущены, `names()` вернет значение `NULL`.

3.3.3 Размерности

Добавление вектору атрибута `dim` позволяет ему вести себя как двумерная *матрица* (matrix) или многомерный *массив* (array). Матрицы и массивы изначально представляют собой математические и статистические структуры, а не инструменты для программирования, так что сталкиваться с ними приходится не так часто, и в этой книге мы будем освещать их не слишком подробно. Наиболее полезной возможностью в работе с ними является извлечение многомерных подмножеств, и об этом мы поговорим в разделе 4.2.3.

Матрицы и массивы можно создавать при помощи специальных функций `matrix()` и `array()`, а также используя метод присваивания значения атрибуту `dim`, как показано ниже:

```
# С помощью двух скалярных аргументов задается количество строк и столбцов
a <- matrix(1:6, nrow = 2, ncol = 3)
a
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6

# Все размерности массива задаются с помощью одного аргумента
b <- array(1:12, c(2, 3, 2))
b
#> , , 1
#>
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
#>
#> , , 2
#>
#>      [,1] [,2] [,3]
#> [1,]    7    9   11
```

```
#> [2,]    8   10   12

# Можно также изменить объект на месте, установив значение атрибута dim()
c <- 1:6
dim(c) <- c(3, 2)
c
#>      [,1] [,2]
#> [1,]    1    4
#> [2,]    2    5
#> [3,]    3    6
```

Многие функции для работы с векторами имеют обобщенные формы для взаимодействия с матрицами и массивами, что показано в табл. 3.1.

Таблица 3.1 Обобщенные функции для работы с матрицами и массивами

Векторы	Матрицы	Массивы
`names()`	`rownames()`, `colnames()`	`dimnames()`
`length()`	`nrow()`, `ncol()`	`dim()`
`c()`	`rbind()`, `cbind()`	`abind::abind()`
–	`t()`	`aperm()`
`is.null(dim(x))`	`is.matrix()`	`is.array()`

Вектор без установленного атрибута `dim` часто воспринимается как одномерный, но на самом деле он имеет `NULL` измерений. Также у вас могут быть матрицы с одной строкой или колонкой или массивы с одним измерением. При выводе они могут выглядеть одинаково, но вести себя при этом будут по-разному. Отличия будут не такими весомыми, но знать о них необходимо на случай, если вы, например, столкнетесь со странным выводом функции `tapply()`. Как всегда, используйте функцию `str()` для обнаружения отличий между разными структурами данных.

```
str(1:3)                   # Одномерный вектор
#> int [1:3] 1 2 3
str(matrix(1:3, ncol = 1)) # Вектор с одной колонкой
#> int [1:3, 1] 1 2 3
str(matrix(1:3, nrow = 1)) # Вектор с одной строкой
#> int [1, 1:3] 1 2 3
str(array(1:3, 3))         # Вектор типа массив
#> int [1:3(1d)] 1 2 3
```

3.3.4 Упражнения

1. Как используются функции `setNames()` и `unname()`? Ознакомьтесь с исходным кодом.
2. Что возвращает `dim()` применительно к одномерному вектору? Когда можно было бы использовать функции `nrow()` или `ncol()`?

3. Как бы вы описали следующие три объекта? Чем они отличаются от `1:5`?

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

4. Вы решили использовать следующий код для демонстрации работы функции `structure()`:

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

Но в выводе вы не увидели установленного значения атрибута `comment`. Почему? Атрибут был просто пропущен или с ним что-то не так? Подсказка: воспользуйтесь справкой по функции.

3.4 Атомарные векторы S3

Одним из наиболее важных атрибутов векторов является атрибут `class`, лежащий в основе *объектной системы S3* (S3 object system). Наличие атрибута `class` переводит объект в разряд *объектов S3* (S3 object), а это означает, что он будет вести себя иначе по сравнению с обычным вектором при передаче в *обобщенную функцию* (generic function). Каждый объект S3 строится на основе базового типа и зачастую хранит дополнительную информацию в других атрибутах. В главе 13 вы узнаете о внутреннем устройстве объектной системы S3 и о том, как можно создавать собственные классы S3.

В этом разделе мы остановимся на четырех важных векторах S3, используемых в базовом R и показанных на рис. 3.7:

- категориальные данные, в которых значения исходят из фиксированного набора уровней, хранятся в виде *векторов факторов* (factor vector);

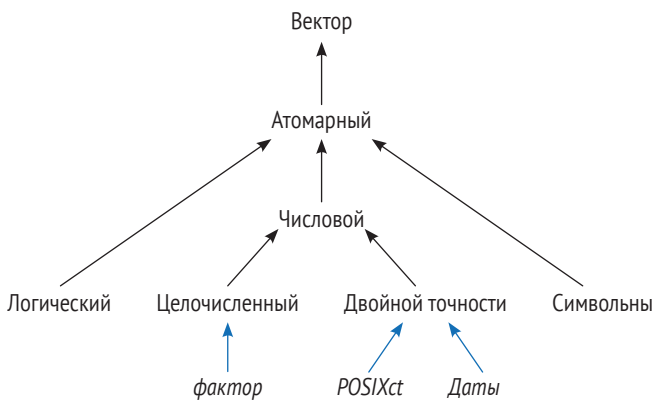


Рис. 3.7 Расширенная иерархия типов векторов в R

- даты (с детализацией до дня) хранятся в виде *векторов дат* (date vector);
- даты со временем (с детализацией до секунды или долей секунды) представлены в виде *векторов POSIXct* (POSIXct vector);
- длительности хранятся в виде *векторов difftime* (difftime vector).

3.4.1 Факторы

Фактор (factor) представляет собой вектор, содержащий только предопределенные значения и использующийся для хранения категориальных данных. Факторы строятся на базе целочисленных векторов с двумя атрибутами: `class`, равным `"factor"`, что заставляет их вести себя иначе по сравнению с обычными целочисленными векторами, и `levels`, в котором перечислены допустимые для вектора значения.

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

typeof(x)
#> [1] "integer"
attributes(x)
#> $levels
#> [1] "a" "b"
#>
#> $class
#> [1] "factor"
```

Графическое представление фактора показано на рис. 3.8.

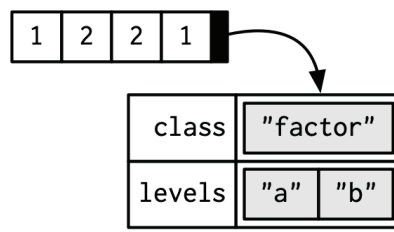


Рис. 3.8 Графическое представление фактора

Факторы можно использовать в ситуациях, когда вам заранее известен набор возможных значений в векторе, но не все они в нем присутствуют. В отличие от символьного вектора, при трансформации фактора в таблицу вы получите общее количество по всем уровням, даже по тем, которые в векторе не встречаются:

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

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

Разновидностью фактора является *упорядоченный фактор* (ordered factor). Он ведет себя так же, как обычный фактор, но в его случае выполняется учет порядка следования уровней (низкий, средний, высокий). Это свойство автоматически подхватывается некоторыми функциями моделирования и визуализации.

```
grade <- ordered(c("b", "b", "a", "c"), levels = c("c", "b", "a"))
grade
#> [1] b b a c
#> Levels: c < b < a
```

В базовом пакете R[1] вам довольно часто придется работать с факторами, поскольку многие основные функции из пакета `base`, такие как `read.csv()` и `data.frame()`, автоматически конвертируют символьные векторы в факторы. Это не всегда бывает удобно, поскольку мы никак не можем передать этим функциям полный набор доступных уровней или их порядок: уровни относятся к теоретической или семантической области, но никак не к данным. Есть вариант использовать аргумент `stringsAsFactors = FALSE` для предотвращения автоматического приведения строк к факторам, после чего вручную преобразовывать символьные векторы в факторы на основе каких-то теоретических знаний. Для погружения в исторический контекст такого поведения я рекомендую вам ознакомиться со статьей *stringsAsFactors: An unauthorized biography* Роджера Пенга (Roger Peng) по адресу https://simplystatistics.org/posts/2015-07-24-stringsasfactors-an-unauthorized-biography и *stringsAsFactors* = <*sigh*> Томаса Ламли (Thomas Lumley) по адресу https://notstatschat.tumblr.com/post/124987394001/stringsasfactors-sigh.

Хотя факторы и выглядят (и часто ведут себя) как символьные векторы, необходимо понимать, что они строятся поверх целочисленных векторов. Так что не стоит относиться к ним как к строкам. Некоторые строковые функции, такие как `gsub()` и `grepl()`, автоматически приводят факторы к строкам, другие, как `nchar()`, возвращают ошибку, а третьи, как `c()`, используют лежащие в их основе целочисленные значения. Поэтому обычно бывает лучше явно преобразовывать факторы в символьные векторы, если вам необходимо, чтобы они вели себя как строки.

---

[1] В наборе пакетов tidyverse никогда не выполняется автоматическое приведение символов к факторам, а для работы с факторами в нем присутствует отдельный пакет `forcats` [Уикем, 2018].

3.4.2 Даты

Векторы дат (date vector) построены поверх векторов двойной точности. У них прописан класс `"Date"`, а другие атрибуты не установлены:

```
today <- Sys.Date()

typeof(today)
#> [1] "double"
attributes(today)
#> $class
#> [1] "Date"
```

Значение с типом двойной точности (которое можно увидеть, если избавиться от атрибута `class`) представляет количество дней, прошедших с 1 января 1970 года[1]:

```
date <- as.Date("1970-02-01")
unclass(date)
#> [1] 31
```

3.4.3 Даты со временем

Базовый пакет R[2] предлагает два способа хранения информации о дате и времени: *POSIXct* и *POSIXlt*. Признаться, довольно странные названия. *POSIX* расшифровывается как *Portable Operating System Interface* (интерфейс переносимой операционной системы) и представляет семейство кросс-платформенных стандартов. Суффикс `ct` означает *calendar time* (календарное время, тип `time_t` в языке C), а `lt` – *local time* (местное время, тип `struct tm` в языке C). Здесь мы будем говорить о классе *POSIXct* по причине того, что он достаточно прост, построен на базе атомарного вектора и идеально подходит для работы с датафреймами. В основе *векторов POSIXct* (POSIXct vector) лежат векторы двойной точности, в которых значения представлены в виде количества секунд, прошедших с 1 января 1970 года.

```
now_ct <- as.POSIXct("2018-08-01 22:00", tz = "UTC")
now_ct
#> [1] "2018-08-01 22:00:00 UTC"

typeof(now_ct)
#> [1] "double"
attributes(now_ct)
```

---

[1] Эта особая дата известна также как *момент начала отсчета времени Unix* (Unix Epoch).

[2] В наборе пакетов tidyverse представлен пакет `lubridate` [Гролмунд (Grolemund) и Уикем, 2011] для работы с датами и временем. В нем содержится множество удобных вспомогательных функций, работающих с типом POSIXct.

```
#> $class
#> [1] "POSIXct" "POSIXt"
#>
#> $tzone
#> [1] "UTC"
```

Атрибут `tzone` отвечает только за формат отображения даты и не участвует в формировании даты и времени, представленной вектором. Обратите внимание, что для полночи составляющая времени не отображается.

```
structure(now_ct, tzone = "Asia/Tokyo")
#> [1] "2018-08-02 07:00:00 JST"
structure(now_ct, tzone = "America/New_York")
#> [1] "2018-08-01 18:00:00 EDT"
structure(now_ct, tzone = "Australia/Lord_Howe")
#> [1] "2018-08-02 08:30:00 +1030"
structure(now_ct, tzone = "Europe/Paris")
#> [1] "2018-08-02 CEST"
```

3.4.4 Длительности

Длительности (durations), выраженные в единицах времени, прошедших с одной даты (или даты со временем) до другой, хранятся в виде типа данных `difftime`. Эти интервалы построены на основе чисел двойной точности и обладают атрибутом `units`, определяющим, как необходимо интерпретировать целочисленную часть интервала:

```
one_week_1 <- as.difftime(1, units = "weeks")
one_week_1
#> Time difference of 1 weeks

typeof(one_week_1)
#> [1] "double"
attributes(one_week_1)
#> $class
#> [1] "difftime"
#>
#> $units
#> [1] "weeks"

one_week_2 <- as.difftime(7, units = "days")
one_week_2
#> Time difference of 7 days

typeof(one_week_2)
#> [1] "double"
attributes(one_week_2)
#> $class
#> [1] "difftime"
#>
```

```
#> $units
#> [1] "days"
```

3.4.5 Упражнения

1. Объект какого вида возвращает функция `table()`? Какой у него тип? Какими атрибутами обладает? Как изменится размерность при преобразовании в таблицу большего количества переменных?
2. Что произойдет с фактором при изменении его атрибута `levels`?

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

3. Что делает приведенный ниже код? Чем `f2` и `f3` отличаются от `f1`?

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

3.5 Списки

Списки (list) – это шаг в сторону усложнения по сравнению с атомарными векторами: каждый элемент в списке может быть представлен любым типом, а не только вектором. Чисто технически все элементы списка принадлежат к одному типу данных, поскольку, как мы узнали в разделе 2.3.3, каждый элемент представляет собой ссылку на объект, который может быть любого типа.

3.5.1 Создание

Списки создаются с помощью функции `list()`, как показано ниже:

```
l1 <- list(
    1:3,
    "a",
    c(TRUE, FALSE, TRUE),
    c(2.3, 5.9)
)

typeof(l1)
#> [1] "list"
str(l1)
#> List of 4
#>  $ : int [1:3] 1 2 3
#>  $ : chr "a"
#>  $ : logi [1:3] TRUE FALSE TRUE
#>  $ : num [1:2] 2.3 5.9
```

Поскольку все элементы являются ссылками, создание списка не подразумевает копирования компонентов в список. Именно поэтому физический размер списка может быть меньше, чем вы можете ожидать.

```
lobstr::obj_size(mtcars)
#> 7,208 B

l2 <- list(mtcars, mtcars, mtcars, mtcars)
lobstr::obj_size(l2)
#> 7,288 B
```

Списки могут содержать сложные объекты, так что невозможно выбрать универсальный стиль графического отображения для каждого списка. В основном я буду изображать списки в виде векторов, как на рис. 3.9, с применением цветов при необходимости для отображения иерархий.

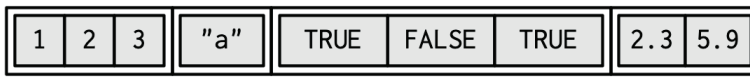


Рис. 3.9 Графическое представление простого списка

Иногда списки называют *рекурсивными векторами* (recursive vector) из-за возможности содержания в своем составе других списков, как показано на рис. 3.10. Этим списки кардинально отличаются от атомарных векторов.

```
l3 <- list(list(list(1)))
str(l3)
#> List of 1
#>  $ :List of 1
#>   ..$ :List of 1
#>   .. ..$ : num 1
```

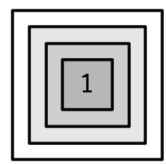


Рис. 3.10 Список, включающий другие списки

Функция `c()` может объединять несколько списков в один. Если ей на вход подать комбинацию из списков и атомарных векторов, она преобразует векторы в списки перед объединением. Сравните результаты выполнения функций `list()` и `c()`, которые схематично показаны на рис. 3.11:

```
l4 <- list(list(1, 2), c(3, 4))
l5 <- c(list(1, 2), c(3, 4))
str(l4)
#> List of 2
#>  $ :List of 2
#>   ..$ : num 1
#>   ..$ : num 2
#>  $ : num [1:2] 3 4
str(l5)
#> List of 4
#>  $ : num 1
#>  $ : num 2
#>  $ : num 3
#>  $ : num 4
```

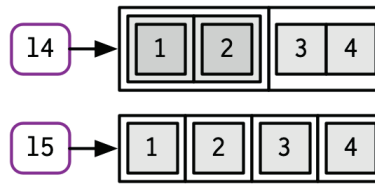


Рис. 3.11 Результат объединения элементов при помощи функций `list()` и `c()`

3.5.2 Определение и приведение типов

Применение функции `typeof()` к списку вернет `list`. Вы можете определить, является ли объект списком, с помощью функции `is.list()`, а привести объект к списочному типу можно посредством функции `as.list()`.

```
list(1:3)
#> [[1]]
#> [1] 1 2 3
as.list(1:3)
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] 2
#>
#> [[3]]
#> [1] 3
```

Преобразовать список в атомарный вектор можно при помощи функции `unlist()`. Правила такого преобразования достаточно сложны и плохо документированы, а результат не всегда будет эквивалентен тому, что можно получить с помощью функции `c()`.

3.5.3 Матрицы и массивы

Для создания матриц на основе атомарных векторов обычно используется атрибут `dim`. Применительно к спискам установка атрибута `dim` приведет к созданию *списков-матриц* (list-matrices) или *списков-массивов* (list-arrays):

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

l[[1, 1]]
#> [1] 1 2 3
```

Это довольно редко используемые структуры данных, но они могут пригодиться при необходимости разместить объекты в табличном виде. К примеру, при работе с пространственно-временными моделями вам может понадобиться хранить модели в виде трехмерных массивов, соответствующих структуре сетки.

3.5.4 Упражнения

1. Перечислите все известные вам отличия списков от атомарных векторов.
2. Почему вам необходимо использовать функцию `unlist()` для преобразования списка в атомарный вектор? Почему не достаточно вызвать функцию `as.vector()`?
3. Сравните использование функций `c()` и `unlist()` для объединения даты и даты со временем в одном векторе.

3.6 Датафреймы и тибблы

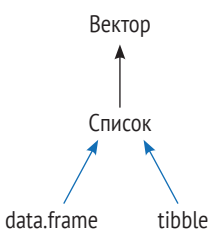


Рис. 3.12 Датафреймы и тибблы в иерархии объектов

Два наиболее важных класса векторов S3, построенных поверх списков, – это *датафреймы* (data frame) и *тибблы* (tibble), схематично показанные на рис. 3.12.

Если вы выполняете анализ данных в R, вам наверняка придется использовать датафреймы. Датафрейм представляет собой именованный список векторов с атрибутами для имен

колонок (names) и строк (row.names)[1], а также для класса (представлен строкой "data.frame"):

```
df1 <- data.frame(x = 1:3, y = letters[1:3])
typeof(df1)
#> [1] "list"

attributes(df1)
#> $names
#> [1] "x" "y"
#>
#> $class
#> [1] "data.frame"
#>
#> $row.names
#> [1] 1 2 3
```

В отличие от обычных списков, датафреймы обладают одним дополнительным ограничением: длины всех векторов, входящих в их состав, должны быть равны. Это обеспечивает датафреймам характерную для них табличную структуру и объясняет, почему они одновременно обладают свойствами матриц и списков:

- к датафрейму применимы функции rownames()[2] и colnames(). Функция names() применительно к датафрейму обозначает имена столбцов;
- для подсчета строк и столбцов в датафрейме присутствуют функции nrow() и ncol() соответственно. Функция length() применительно к датафрейму возвращает количество его столбцов.

Датафреймы представляют одну из наиболее важных концепций в R, отличающих этот язык программирования от остальных. Однако за 20 с лишним лет с момента его создания принципы использования языка претерпели значительные изменения, и некоторые идеи, имевшие смысл в то время, сегодня кажутся устаревшими.

Все это привело к созданию новой концепции датафреймов, получившей название *тиббл* (tibble) [Кирилл Мюллер и Хэдли Уикем, 2018]. Изначально тибблы задумывались как улучшенная версия датафреймов, лишенная при-

---

[1] Имена строк представляют собой на удивление одну из самых сложных структур данных в R. Кроме того, они зачастую являются источниками проблем с производительностью. Наиболее простой реализацией имен строк является представление в виде символьных или целочисленных векторов с одним элементом на каждую строку. Но есть также компактное представление «автоматических» имен строк (в виде последовательных целых чисел), возвращаемое функцией .set_row_names(). В R версии 3.5 присутствует особый метод отложенного преобразования целых чисел в строки, который был специально предложен для ускорения функции lm(). За подробностями можно обратиться по адресу https://svn.r-project.org/R/branches/ALTREP/ALTREP.html#deferred_string_conversions.

[2] Чисто технически вам следует использовать при работе с датафреймами функцию row.names() вместо rownames(), но это не столь важно.

сущих им недостатков. Если говорить об изменениях в тибблах кратко, то получится так, что они стали более ленивыми и ворчливыми – делают меньше, а жалуются больше. Впоследствии вы поймете, что здесь имеется в виду.

Тибблы представлены в пакете `tibble` и обладают такой же структурой, как и датафреймы. Единственным отличием является то, что атрибут `class` у них стал длиннее и включает в себя класс `"tbl_df"`. Это позволяет тибблам вести себя иначе в сравнении с датафреймами, о чем мы поговорим далее.

```
library(tibble)

df2 <- tibble(x = 1:3, y = letters[1:3])
typeof(df2)
#> [1] "list"

attributes(df2)
#> $names
#> [1] "x" "y"
#>
#> $row.names
#> [1] 1 2 3
#>
#> $class
#> [1] "tbl_df"    "tbl"       "data.frame"
```

3.6.1 Создание

Датафреймы создаются путем передачи пар имя–вектор функции `data.frame()`:

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

Помните о том, что по умолчанию строки могут преобразовываться в факторы. Для подавления такого поведения и сохранения символьных векторов в исходном виде используйте параметр `stringsAsFactors = FALSE`:

```
df1 <- data.frame(
    x = 1:3,
    y = c("a", "b", "c"),
    stringsAsFactors = FALSE
)
str(df1)
#> 'data.frame':     3 obs. of 2 variables:
```

```
#> $ x: int 1 2 3
#> $ y: chr "a" "b" "c"
```

Создание объекта tibble выполняется схожим образом. Разница состоит лишь в том, что при создании тибблов никогда не происходит принудительное преобразование данных (это одна из особенностей, которые делают тибблы ленивыми):

```
df2 <- tibble(
    x = 1:3,
    y = c("a", "b", "c")
)
str(df2)
#> Classes 'tbl_df', 'tbl' and 'data.frame':     3 obs. of 2 variables:
#> $ x: int  1 2 3
#> $ y: chr  "a" "b" "c"
```

Также при создании датафреймов выполняется автоматическое преобразование синтаксически неправильных имен (если не указан параметр check.names = FALSE), чего не происходит в случае с тибблами, хотя синтаксически неправильные имена в этом случае выводятся заключенными в обратные апострофы.

```
names(data.frame(`1` = 1))
#> [1] "X1"

names(tibble(`1` = 1))
#> [1] "1"
```

Тогда как все элементы датафреймов и тибблов должны иметь одинаковую длину, при их создании элементы меньшего размера будут расширяться циклически, как показано ниже. Причем если датафреймы умеют восстанавливать значения столбцов при задании в них кратного количества элементов относительно общей длины вектора, то в тибблах могут автоматически расширяться только элементы единичной длины:

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

tibble(x = 1:4, y = 1)
#> # A tibble: 4 x 2
#>       x     y
```

```
#>   <int> <dbl>
#> 1     1     1
#> 2     2     1
#> 3     3     1
#> 4     4     1
tibble(x = 1:4, y = 1:2)
#> Error: Tibble columns must have consistent lengths, only values of
#> length one are recycled:
#> * Length 2: Column `y`
#> * Length 4: Column `x`
```

Есть и еще одно отличие, состоящее в том, что в функции `tibble()` вы можете ссылаться на переменные, прописанные ранее в том же выражении:

```
tibble(
    x = 1:3,
    y = x * 2
)
#> # A tibble: 3 x 2
#>       x     y
#>   <int> <dbl>
#> 1     1     2
#> 2     2     4
#> 3     3     6
```

Входные переменные в этом случае обрабатываются слева направо.

При графическом отображении датафреймов и тибблов, вместо того чтобы концентрироваться на внутренней реализации, т. е. атрибутах (как показано на рис. 3.13), мы будем выводить их в виде именованных списков, но с подчеркиванием их колоночной структуры, как на рис. 3.14.

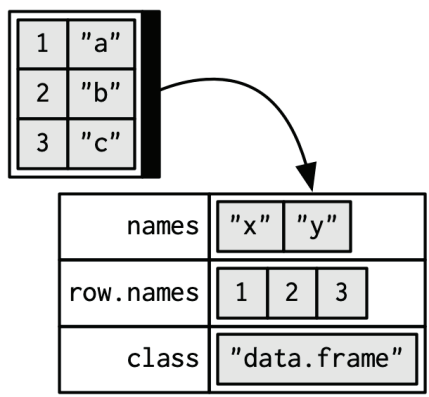


Рис. 3.13 Атрибуты датафрейма

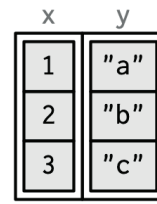


Рис. 3.14 Датафрейм в виде колоночной структуры

3.6.2 Имена строк

Датафреймы позволяют снабжать строки именами в виде символьных векторов, состоящих из уникальных значений:

```
df3 <- data.frame(
    age = c(35, 27, 18),
    hair = c("blond", "brown", "black"),
    row.names = c("Bob", "Susan", "Sam")
)
df3
#>       age  hair
#> Bob    35 blond
#> Susan  27 brown
#> Sam    18 black
```

Получать и устанавливать имена строк можно с помощью функции `rownames()`, и их можно использовать при создании подмножеств строк:

```
rownames(df3)
#> [1] "Bob"    "Susan"   "Sam"
df3["Bob", ]
#>     age  hair
#> Bob  35 blond
```

Имена строк воспринимаются вполне естественно, если думать о датафреймах как о двумерных структурах вроде матриц: у колонок (переменных) есть свои имена, значит, они должны быть и у строк (наблюдений). В большинстве случаев матрицы наполнены числовыми значениями, так что очень важно иметь место для хранения символьных меток. Однако эта аналогия с матрицами не вполне уместна, поскольку матрицы обладают одним важным свойством, которого нет у датафреймов, – транспонируемостью. В матрицах строки и столбцы взаимозаменяемы, а их *транспонирование* (transposing) дает новую матрицу. Ее повторное транспонирование вернет нас к исходной матрице. В случае с датафреймами строки и колонки не являются взаимозаменяемыми: таким образом, транспонирование датафрейма не приведет к созданию нового датафрейма.

Есть три причины, по которым использовать имена для строк нежелательно:

- метаданные – это тоже данные, так что хранить их иначе по сравнению с другими данными – не лучшая идея. Также это означает, что вам придется изучить дополнительный набор инструментов для работы с именами строк. Использовать уже знакомые вам принципы для работы с колонками не получится;
- имена строк – довольно ущербная абстракция для условного обозначения наблюдений, поскольку работает она только при возможности уникальной идентификации строк по символьной метке. Но зачастую это бывает не так – например, когда вам необходимо идентифицировать строки с использованием вектора, отличного от символьного, допустим с помощью точек во времени, или сразу нескольких векторов (координаты с участием широты и долготы);
- имена строк должны быть уникальными, так что любое дублирование строк (например, в результате применения *бутстрэппинга* (bootstrapping)) приведет к созданию новых имен строк. В итоге для выполнения анализа данных с сопоставлением строк до и после преобразования вам придется очень серьезно поработать со строками, что видно ниже:

```
df3[c(1, 1, 1), ]
#>       age  hair
#> Bob    35 blond
#> Bob.1  35 blond
#> Bob.2  35 blond
```

Именно поэтому в тибблах не поддерживаются имена строк в традиционном виде. Вместо этого пакет `tibble` предлагает инструменты для легкого преобразования имен строк в обычный столбец. Для этого можно воспользоваться специальной функцией `rownames_to_column()` или аргументом `rownames` функции `as_tibble()`:

```
as_tibble(df3, rownames = "name")
#> # A tibble: 3 x 3
#>   name    age hair
#>   <chr> <dbl> <fct>
#> 1 Bob      35 blond
#> 2 Susan    27 brown
#> 3 Sam      18 black
```

3.6.3 Вывод на экран

Одним из наиболее заметных отличий между датафреймами и тибблами является их способ вывода на экран. Я полагаю, вы уже знакомы с тем, как выводятся датафреймы, так что здесь я продемонстрирую некоторые важные отличия вывода тибблов на примере набора данных, включенного в пакет `dplyr`:

```
dplyr::starwars
#> # A tibble: 87 x 13
#>    name  height  mass hair_color skin_color eye_color birth_year
#>    <chr>  <int> <dbl> <chr>      <chr>      <chr>          <dbl>
#>  1 Luke…    172    77 blond      fair       blue            19
#>  2 C-3PO    167    75 <NA>       gold       yellow         112
#>  3 R2-D2     96    32 <NA>       white, bl… red             33
#>  4 Dart…    202   136 none       white      yellow          41.9
#>  5 Leia…    150    49 brown      light      brown           19
#>  6 Owen…    178   120 brown, gr… light      blue            52
#>  7 Beru…    165    75 brown      light      blue            47
#>  8 R5-D4     97    32 <NA>       white, red red             NA
#>  9 Bigg…    183    84 black      light      brown           24
#> 10 Obi-…    182    77 auburn, w… fair       blue-gray       57
#> # … with 77 more rows, and 6 more variables: gender <chr>,
#> # homeworld <chr>, species <chr>, films <list>, vehicles <list>,
#> # starships <list>
```

- в случае с тибблами выводятся только первые десять строк и все колонки, которые помещаются на экран. Оставшиеся колонки перечислены в нижней части вывода;
- каждая колонка обозначается своим типом данных, сокращенным до трех или четырех символов;
- содержимое колонок с объемными значениями обрезается, чтобы не занимать всю строку. Работа в области отображения ведется постоянно, поскольку очень непросто бывает найти подходящий компромисс между выводом как можно большего количества колонок и полнотой их отображения;
- при использовании консольных окружений, поддерживающих цветовые схемы, важная информация выделяется цветом, тем самым снимая акцент с дополнительных сведений.

3.6.4 Извлечение подмножеств

Как вы узнаете в главе 4, данные из датафрейма или тиббла в виде *подмножества* (subset) можно извлекать в форме одномерной (тогда они ведут себя как список) или двумерной (матрица) структуры.

По моему мнению, датафреймам присущи два нежелательных поведения в отношении извлечения подмножеств:

- при извлечении подмножества колонок с помощью выражения `df[, vars]` вы получите вектор, если `vars` указывает на один столбец, и датафрейм в обратном случае. Зачастую такое поведение становится источником ошибок при использовании `[` в функции, если вы не применяете дополнительный аргумент `drop`: `df[, vars, drop = FALSE]`;
- в отсутствие в датафрейме колонки с именем `x` попытка извлечь ее с помощью выражения `df$x` завершится выбором любой другой пере-

менной, имя которой начинается с x. Если таких переменных не окажется, выражение df$x вернет NULL. Это зачастую приводит к ошибочному выбору нежелательных столбцов.

В тибблах применение квадратных скобок всегда приводит к извлечению нового тиббла, а использование символа доллара не ведет к поиску частичных совпадений. В отсутствие запрашиваемого столбца выводится предупреждение, что и делает пакет tibble таким ворчливым.

```
df1 <- data.frame(xyz = "a")
df2 <- tibble(xyz = "a")

str(df1$x)
#> Factor w/ 1 level "a": 1
str(df2$x)
#> Warning: Unknown or uninitialised column: 'x'.
#> NULL
```

Склонность тибблов возвращать табличные данные при использовании квадратных скобок может привести к проблемам с обратной совместимостью с кодом, в котором выражения вроде df[, "col"] применяются с целью извлечения результата в виде вектора. Если вам нужно поработать с одной колонкой, я рекомендую воспользоваться синтаксисом df[["col"]]. Такой способ полностью отражает ваши намерения и будет успешно работать как с датафреймами, так и с тибблами.

3.6.5 Определение и приведение табличных типов

Для проверки того, является ли объект датафреймом или тибблом, можно использовать функцию is.data.frame():

```
is.data.frame(df1)
#> [1] TRUE
is.data.frame(df2)
#> [1] TRUE
```

Обычно не имеет большого значения, работаете вы с тибблом или с датафреймом, но если вам это важно, воспользуйтесь специальной функцией is_tibble():

```
is_tibble(df1)
#> [1] FALSE
is_tibble(df2)
#> [1] TRUE
```

Преобразовать объект в датафрейм можно с помощью функции as.data.frame(), а в тиббл – с помощью функции as_tibble().

3.6.6 Списки в колонках

Поскольку датафреймы по своей природе являются списками векторов, вполне допустимо, чтобы в виде колонки в датафрейме присутствовал список, как показано на рис. 3.15. Это бывает довольно удобно из-за возможности списков хранить разнородные данные, что позволяет любой объект поместить в датафрейм. Так вы можете хранить связанные объекты в строке вне зависимости от того, насколько сложными являются отдельные объекты. Применение этого приема на практике подробно рассматривается в главе *Обработка множества моделей* (Many Models) книги *Язык R в задачах науки о данных* (R for Data Science): https://r4ds.had.co.nz/many-models.html.

Колонки-списки допустимы в датафреймах, но вам необходимо выполнить некую дополнительную работу для их хранения: либо добавить их после создания датафрейма, либо обернуть список в функцию `I()`[1] на этапе его создания.

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

data.frame(
    x = 1:3,
    y = I(list(1:2, 1:3, 1:4))
)
#>   x          y
#> 1 1       1, 2
#> 2 2    1, 2, 3
#> 3 3 1, 2, 3, 4
```

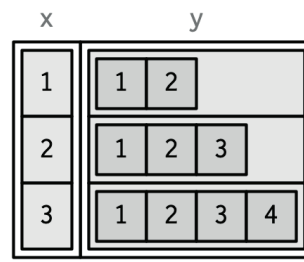


Рис. 3.15 Схематическое представление колонки со списком

Колонки-списки легче использовать в тибблах, поскольку их можно определять прямо при создании объекта, и они выводятся на экран в сжатой форме:

[1] Название функции `I()` происходит от слова *identity* (сущность), и она часто используется для обозначения того, что аргумент не должен трансформироваться автоматически, а должен предстать как есть.

```
tibble(
    x = 1:3,
    y = list(1:2, 1:3, 1:4)
)
#> # A tibble: 3 x 2
#>       x y
#>   <int> <list>
#> 1     1 <int [2]>
#> 2     2 <int [3]>
#> 3     3 <int [4]>
```

3.6.7 Матрицы и датафреймы в колонках

Если количество строк в матрице или массиве соответствует числу строк в датафрейме, этот объект можно использовать в виде колонки в этом датафрейме. Правда, это требует небольшой корректировки определения датафрейма: здесь мы говорим о равенстве для каждой колонки не `length()`, а `NROW()`. В случае с колонками-списками их необходимо либо добавлять после создания датафрейма, либо использовать вспомогательную функцию `I()`.

```
dfm <- data.frame(
    x = 1:3 * 10
)
dfm$y <- matrix(1:9, nrow = 3)
dfm$z <- data.frame(a = 3:1, b = letters[1:3], stringsAsFactors = FALSE)

str(dfm)
#> 'data.frame':     3 obs. of   3 variables:
#>  $ x: num  10 20 30
#>  $ y: int [1:3, 1:3] 1 2 3 4 5 6 7 8 9
#>  $ z:'data.frame':   3 obs. of 2 variables:
#>    ..$ a: int  3 2 1
#>    ..$ b: chr  "a" "b" "c"
```

Внешне такой датафрейм можно представить так, как показано на рис. 3.16.

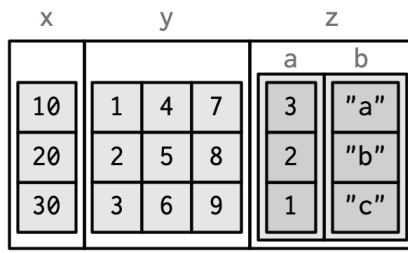


Рис. 3.16. Датафрейм с матрицей и другим датафреймом в качестве столбцов

При использовании матриц и датафреймов в виде столбцов фрейма данных необходимо проявлять особую осторожность. Многие функции для работы с датафреймами рассчитывают на то, что во всех колонках объекта представлены векторы. Кроме того, вывод таких датафреймов на экран может сбивать с толку.

```
dfm[1, ]
#>    x y.1 y.2 y.3 z.a z.b
#> 1 10   1   4   7   3   a
```

3.6.8 Упражнения

1. Может ли в датафрейме не быть строк? А как насчет отсутствия столбцов?
2. Что произойдет, если при определении имен строк использовать вектор с неуникальными значениями?
3. Если `df` – это датафрейм, то что вы можете сказать о выражениях `t(df)` и `t(t(df))`? Проведите несколько экспериментов, используя при этом разные типы колонок.
4. Что делает функция `as.matrix()` применительно к датафрейму с колонками разных типов? Чем она отличается от функции `data.matrix()`?

3.7 NULL

В заключение этой главы поговорим еще об одной важной структуре данных, имеющей непосредственное отношение к векторам. Речь пойдет о значении `NULL`. Значение `NULL` во всех отношениях особенное, поскольку обладает уникальным типом, нулевой длиной и не может иметь собственных атрибутов:

```
typeof(NULL)
#> [1] "NULL"

length(NULL)
#> [1] 0

x <- NULL
attr(x, "y") <- 1
#> Error in attr(x, "y") <- 1: attempt to set an attribute on NULL
```

Проверить значение на `NULL` можно с помощью функции `is.null()`:

```
is.null(NULL)
#> [1] TRUE
```

Существует два распространенных способа использования NULL:

- для представления пустого вектора (вектора нулевой длины) произвольного типа. К примеру, использование функции c() без аргументов вернет значение NULL, а конкатенирование значения NULL с вектором оставит его неизменным:

```
c()
#> NULL
```

- для представления *отсутствующего вектора* (absent vector). К примеру, NULL часто используется в качестве аргумента функции по умолчанию, если этот аргумент необязательный, но значение по умолчанию требует определенных вычислений (подробнее об этом мы поговорим в разделе 6.5.3). Почувствуйте разницу со значением NA, которое используется для указания на то, что *элемент* вектора отсутствует.

Если вы знакомы с языком запросов SQL, то знаете о том, как там используется значение NULL. Вы могли бы подумать, что в R все так же. Но нет, аналогом этого значения NULL в R является не NULL, а NA.

3.8 Ответы на контрольные вопросы

1. Четыре распространенных типа атомарных векторов – логический, целочисленный, двойной точности и символьный. Два редко используемых типа – это вектор комплексных чисел и вектор сырых данных, или байтовых последовательностей.
2. Атрибуты позволяют ассоциировать произвольные добавочные метаданные с любым объектом. Для чтения и установки значений атрибутов можно воспользоваться конструкциями attr(x, "y") и attr(x, "y") <- value. Извлечь или установить значения для всех атрибутов разом можно с помощью функции attributes().
3. Элементы списков могут иметь разные типы данных (это могут быть даже другие списки). Элементы атомарных векторов должны быть однотипными. Что касается табличного представления, здесь матрицы могут хранить только элементы одного типа, а в датафреймах колонки могут быть разных типов.
4. Вы можете создать список-массив, задав для списка размерности. Чтобы использовать матрицу в качестве колонки в датафрейме, необходимо воспользоваться конструкцией df$x <- matrix() или применить функцию I() при создании датафрейма следующим образом: data.frame(x = I(matrix())).
5. Тибблы отличаются от датафреймов своим выводом на экран, отсутствием приведения строк к факторам по умолчанию и более строгими правилами извлечения подмножеств.

4

Подмножества

4.1 Введение

Операторы *извлечения подмножеств* (subsetting) в R работают быстро и весело. Их освоение позволит вам с помощью очень лаконичных конструкций выполнять довольно сложные действия, что недостижимо в большинстве других языков программирования. Научиться извлекать подмножества в R на базовом уровне можно очень быстро, но чтобы достигнуть вершин мастерства в этой области, потребуется впитать сразу несколько пересекающихся концепций, связанных с тем, что:

- существует сразу шесть способов извлечения подмножеств из атомарных векторов;
- есть целых три оператора для получения подмножеств: `[[`, `[` и `$`;
- операторы извлечения подмножеств по-разному взаимодействуют с разными типами векторов (атомарными векторами, списками, факторами, матрицами и датафреймами);
- извлечение подмножеств можно совмещать с операциями присваивания.

Работа с подмножествами является естественным дополнением к функции `str()`. Если функция `str()` позволяет увидеть все составляющие объекта (его внутреннюю структуру), то операторы, которые мы будем изучать в этой главе, дают непосредственный доступ к этим составляющим. Для больших и сложных объектов я рекомендую воспользоваться интерактивным инструментом *RStudio Viewer*, который можно активировать командой `View(my_object)`.

Контрольные вопросы

Ответьте на вопросы и узнайте, есть ли вам необходимость читать эту главу. Если вы легко и непринужденно можете ответить на все из них, можете сразу переходить к следующей главе. Ответы на вопросы – в разделе 4.6.

1. Каким будет результат извлечения подмножества из вектора с использованием положительных целочисленных значений? Отрицательных? Логического вектора? Символьного вектора?

2. Чем отличается использование операторов `[`, `[[` и `$` применительно к спискам?
3. Когда необходимо использовать аргумент `drop = FALSE`?
4. Если `x` – это матрица, что произойдет после выполнения команды `x[] <- 0`? Чем она отличается от команды `x <- 0`?
5. Как можно использовать именованный вектор для изменения меток категориальных переменных?

Структура главы

- Раздел 4.2 мы начнем с изучения оператора `[`. Вы узнаете о шести разных способах извлечения подмножеств из атомарных векторов. После этого вы увидите, как все эти способы применяются для получения подмножеств из списков, матриц и датафреймов.
- В разделе 4.3 к вашим знаниям об извлечении подмножеств добавятся операторы `[[` и `$`.
- В разделе 4.4 вы узнаете о прелестях совмещения операций извлечения подмножеств и присваивания для модификации частей объектов.
- В разделе 4.5 мы пройдем по восьми важным, но не самым очевидным способам применения подмножеств при решении типичных проблем из области анализа данных.

4.2 Выбор нескольких элементов

Для выбора нескольких элементов из вектора можно воспользоваться оператором `[`. В целях демонстрации мы сперва используем этот оператор с одномерными атомарными векторами, после чего расширим его применение до более сложных объектов и большего количества измерений.

4.2.1 Атомарные векторы

Давайте рассмотрим разные варианты извлечения подмножеств на примере простого вектора `x`.

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

Обратите внимание, что мы так подобрали числа, чтобы цифра после запятой обозначала исходную позицию элемента в векторе.

Приведем шесть разных способов для выбора элементов из вектора:

- **положительные целочисленные индексы**: возвращаются элементы, находящиеся на указанных позициях:

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

# Дублирование индексов приведет к повтору значений
x[c(1, 1)]
#> [1] 2.1 2.1

# Вещественные числа молча обрезаются до целочисленных
x[c(2.1, 2.9)]
#> [1] 4.2 4.2
```

- **отрицательные целочисленные индексы**: возвращаются все элементы, кроме тех, что указаны:

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

Обратите внимание, что вы не можете объединять в одном списке и положительные, и отрицательные значения:

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

- **логический вектор**: выбираются те элементы из исходного вектора, для которых соответствующее логическое значение равно `TRUE`. Вероятно, это один из наиболее удобных способов извлечения подмножеств, поскольку вы сами можете писать выражения, возвращающие логические значения:

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

А что произойдет, если в выражении `x[y]` векторы `x` и `y` будут разной длины? В этом случае сработает *правило переписывания* (recycling rule), и вектор меньшей длины растянется до нужного размера. Легче всего это правило понять для случая, когда вектор `x` или `y` обладает единичной длиной, в то же время я не рекомендую использовать его для других длин векторов, поскольку в базовом R это правило применяется непоследовательно.

```
x[c(TRUE, FALSE)]
#> [1] 2.1 3.3
# Эквивалент:
```

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

Обратите внимание, что пропущенные значения в векторе индексов приведут к появлению пропущенных значений и в итоговом векторе:

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

- **пустое значение**: вернется исходный вектор. Это не очень интересно в случае с одномерными векторами, но, как вы увидите далее, исключительно полезно применительно к матрицам, датафреймам и массивам. Этот способ также может эффективно применяться совместно с операцией присваивания.

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

- **ноль**: вернется вектор нулевой длины. Вряд ли вы будете пользоваться этим приемом намеренно, но он может пригодиться, например, при моделировании тестовых данных:

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

- если исходный вектор именованный, можно использовать в качестве индекса **символьный вектор** – в этом случае будут возвращены значения, соответствующие указанным именам:

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

# Как и в случае с целочисленными индексами, допускаются повторы
y[c("a", "a", "a")]
#>   a   a   a
#> 2.1 2.1 2.1

# При использовании оператора [ имена сопоставляются только целиком
z <- c(abc = 1, def = 2)
z[c("a", "d")]
#> <NA> <NA>
#>   NA   NA
```

Примечание: факторы при извлечении подмножеств воспринимаются на основе лежащих в их основе целочисленных векторов, без учета

указанных в них уровней. Это может приводить в замешательство, так что лучше избегать использования факторов при индексации элементов:

```
y[factor("b")]
#>   a
#> 2.1
```

4.2.2 Списки

Извлечение подмножеств из списков происходит так же, как и с атомарными векторами. При этом использование оператора [всегда приводит к извлечению списка, а операторы [[и $, как вы узнаете в разделе 4.3, позволяют выбрать сами элементы.

4.2.3 Матрицы и массивы

Извлекать подмножества из структур повышенной размерности можно тремя разными способами:

- с помощью нескольких векторов;
- с помощью одного вектора;
- с помощью матрицы.

Наиболее распространенным способом получения подмножеств на основе матриц (два измерения) и массивов (больше двух измерений) является обобщение одномерного извлечения подмножеств путем передачи векторов с индексами для каждого измерения, разделенных запятыми. В этом случае пропуск индекса играет важную роль, позволяя выбрать все строки или столбцы.

```
a <- matrix(1:9, nrow = 3)
colnames(a) <- c("A", "B", "C")
a[1:2, ]
#>      A B C
#> [1,] 1 4 7
#> [2,] 2 5 8
a[c(TRUE, FALSE, TRUE), c("B", "A")]
#>      B A
#> [1,] 4 1
#> [2,] 6 3
a[0, -2]
#>      A C
```

По умолчанию оператор [упрощает результаты до самой низкой возможной размерности. К примеру, оба выражения, показанных ниже, возвращают одномерные векторы. В разделе 4.2.5 вы узнаете, как можно избежать потери размерностей:

```
a[1, ]
#> A B C
#> 1 4 7
a[1, 1]
#> A
#> 1
```

Поскольку матрицы и массивы по своей сути представляют векторы со специальными атрибутами, вы можете извлекать из них подмножества с помощью обычных векторов, как если бы они сами были одномерными векторами. Не забывайте, что в R массивы хранятся в виде колонок:

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

vals[c(4, 15)]
#> [1] "4,1" "5,3"
```

Также можно извлекать подмножества из структур данных с большим количеством размерностей при помощи целочисленных матриц (в случае с именованными структурами можно воспользоваться символьными матрицами). Каждая строка матрицы определяет местоположение значения, а каждая колонка соответствует измерению в массиве. Это означает, что вы можете воспользоваться матрицей с двумя колонками для извлечения матрицы, с тремя колонками – для получения трехмерного массива и т. д. Результатом будет вектор значений:

```
select <- matrix(ncol = 2, byrow = TRUE, c(
    1, 1,
    3, 1,
    2, 4
))
vals[select]
#> [1] "1,1" "3,1" "2,4"
```

4.2.4 Датафреймы и тибблы

Датафреймы одновременно обладают характеристиками и списков, и матриц:

- при извлечении подмножеств с помощью одного индекса они ведут себя как списки, а значит, индекс будет распространяться на столбцы. Таким образом, выражение `df[1:2]` выберет две первые колонки;

- при передаче двух индексов датафреймы начинают вести себя как матрицы, так что выражение `df[1:3, ]` приведет к выбору первых трех *строк* (и всех колонок)[1].

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

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

# Есть два способа выбрать колонки из датафрейма
# Как список
df[c("x", "z")]
#>   x z
#> 1 1 a
#> 2 2 b
#> 3 3 c
# Как матрица
df[, c("x", "z")]
#>   x z
#> 1 1 a
#> 2 2 b
#> 3 3 c

# При выборе одной колонки есть важный нюанс:
# выбор в виде матрицы по умолчанию выполняет упрощение, а в виде списка – нет
str(df["x"])
#> 'data.frame':     3 obs. of 1 variable:
#> $ x: int  1 2 3
str(df[, "x"])
#> int [1:3] 1 2 3
```

Извлечение подмножеств из тиббла с помощью оператора [всегда возвращает тиббл:

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

str(df["x"])
#> Classes 'tbl_df', 'tbl' and 'data.frame':     3 obs. of 1 variable:
#> $ x: int 1 2 3
str(df[, "x"])
```

[1] Если вы пришли из мира Python, это может вас немного смутить, поскольку в вашем понимании выражение `df[1:3, 1:2]` должно приводить к выбору трех колонок и двух строк. В основном R «думает» об измерениях как о строках и столбцах, а Python – как о столбцах и строках.

```
#> Classes 'tbl_df', 'tbl' and 'data.frame':     3 obs. of 1 variable:
#> $ x: int 1 2 3
```

4.2.5 Сохранение размерностей

По умолчанию извлечение подмножеств из матрицы или датафрейма с помощью одного числа, имени или логического вектора, состоящего из единственного значения TRUE, будет приводить к упрощению итогового результата, т. е. снижению количества его измерений. Для сохранения исходной размерности объекта необходимо воспользоваться атрибутом drop = FALSE:

- для матриц и массивов все измерения единичной длины будут отброшены:

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

str(a[1, , drop = FALSE])
#> int [1, 1:2] 1 3
```

- датафреймы, состоящие из одной колонки, будут возвращены в виде одной колонки:

```
df <- data.frame(a = 1:2, b = 1:2)
str(df[, "a"])
#> int [1:2] 1 2

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

Значение аргумента drop по умолчанию (TRUE) зачастую является источником ошибок в функциях: разработчик проверяет свой код на датафреймах или матрицах с несколькими столбцами, и все работает. Но спустя полгода он (или кто-то другой) вызывает функцию применительно к датафрейму с одной колонкой, и она отваливается со странной ошибкой. Возьмите за привычку при написании функций всегда использовать аргумент drop = FALSE при извлечении подмножеств из двумерных объектов. Именно поэтому при работе с тиббламиэтот аргумент по умолчанию равен FALSE, а оператор [всегда возвращает другой тиббл.

При извлечении подмножеств из факторов аргумент drop также присутствует, но его смысл совершенно иной. В данном случае он отвечает за то, будут ли сохраняться уровни (а не размерности), и его значение по умолчанию равно FALSE. Если вам часто приходится использовать аргумент drop = TRUE, это может быть сигналом к тому, чтобы перейти с факторов на обычные символьные векторы.

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

4.2.6 Упражнения

1. Исправьте ошибки в приведенных ниже примерах извлечения подмножеств из датафрейма:

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

2. Почему приведенный ниже код возвращает пять пропущенных значений? Подсказка: чем это выражение отличается от `x[NA_real_]`?

```
x <- 1:5
x[NA]
#> [1] NA NA NA NA NA
```

3. Что вернет функция `upper.tri()`? Как здесь выполняется извлечение подмножества из матрицы? Нужны ли какие-то дополнительные правила получения подмножеств для описания такого поведения?

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

4. Почему выражение `mtcars[1:20]` возвращает ошибку? Чем оно отличается от подобного ему выражения `mtcars[1:20, ]`?
5. Реализуйте свою функцию извлечения диагональных элементов из матрицы. Она должна работать подобно функции `diag(x)` при передаче ей матрицы `x`.
6. Что делает инструкция `df[is.na(df)] <- 0`? Как она работает?

4.3 Выбор одного элемента

Есть еще два оператора для извлечения подмножеств: `[[` и `$`. Оператор `[[` используется для получения одиночных элементов, а выражение `x$y` представляет собой сокращение выражения `x[["y"]]`.

4.3.1 [[

Оператор [[наиболее часто используется при работе со списками, поскольку извлечение подмножеств из списков с помощью оператора [всегда приводит к получению списка меньшего размера. Для лучшего понимания этой концепции можно использовать простую метафору:

> *Если представить x как поезд с объектами, то x[[5]] вернет объект, содержащийся в вагоне № 5, а x[4:6] – состав из вагонов с номерами 4, 5 и 6.*
>
> *— @RLangTip*

Давайте применим эту метафору к следующему списку:

```
x <- list(1:3, "a", 4:6)
```

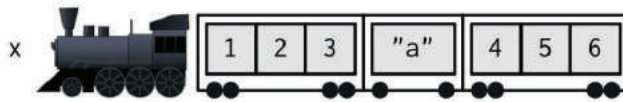


Рис. 4.1 Список в виде поезда

Графически можно изобразить этот список в виде поезда, показанного на рис. 4.1. При необходимости извлечения одного элемента вы можете воспользоваться двумя способами: либо оставить состав меньшего размера, отцепив от него лишние вагоны, либо извлечь содержимое нужного вам вагона, а поезд отпустить. В этом и состоит разница между операторами [и [[, показанная на рис. 4.2.

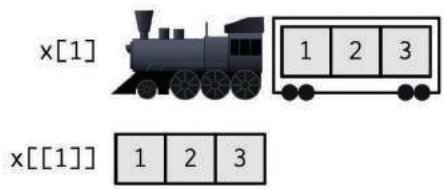


Рис. 4.2 Поезд с нужным нам вагоном и содержимое нужного нам вагона отдельно

Таким образом, при извлечении нескольких элементов (даже если это ноль элементов) вы собираете новый поезд, что видно на рис. 4.3.

Поскольку оператор [[способен возвращать лишь отдельные элементы, его допускается использовать только с одним числовым или символьным индексом. Использование этого оператора с вектором значений приведет к рекурсивному извлечению подмножеств, т. е. выражения `x[[c(1, 2)]]` и `x[[1]][[2]]` можно считать полностью эквивалентными. Об этом знают немногие, так что я рекомендую не использовать такие конструкции, а лучше воспользоваться функцией `purrr::pluck()`, о которой мы будем говорить в разделе 4.3.3.

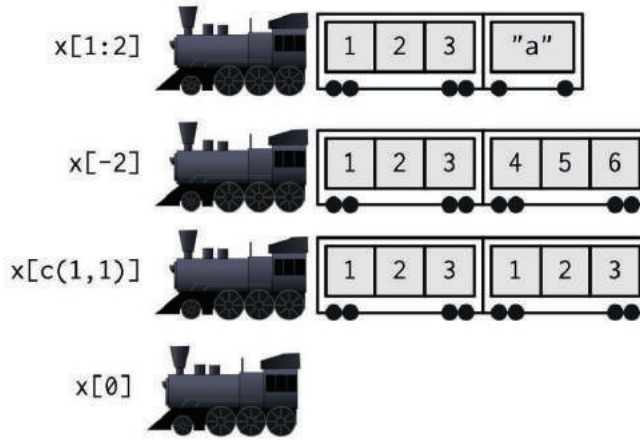


Рис. 4.3. Поезда, соответствующие разным примерам извлечения элементов

Хотя оператор [[в основном применяется в работе со списками, я советую использовать его и с атомарными векторами при необходимости извлечения одного значения. К примеру, вместо написания следующего цикла

```
for (i in 2:length(x)) {
    out[i] <- fun(x[i], out[i - 1])
}
```

лучше будет воспользоваться такой конструкцией:

```
for (i in 2:length(x)) {
    out[[i]] <- fun(x[[i]], out[[i - 1]])
}
```

Таким образом вы лишний раз подчеркиваете, что извлекаете и устанавливаете отдельные значения.

4.3.2 $

Оператор $ является сокращением для оператора [[, так что, если говорить грубо, выражение `x$y` эквивалентно `x[["y"]]`. Часто этот оператор используется для доступа к переменным датафрейма, например `mtcars$cyl` или `diamonds$carat`. Распространенной ошибкой использования оператора $ является его применение с переменной, в которой сохранено имя столбца:

```
var <- "cyl"
# Не сработает - mtcars$var преобразуется в mtcars[["var"]]
mtcars$var
#> NULL

# Вместо этого нужно использовать [[
```

```
mtcars[[var]]
#> [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4
```

Важное отличие между операторами $ и [[состоит в том, что оператор $ допускает частичное (слева направо) соответствие:

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

Во избежание такого поведения я настоятельно рекомендую устанавливать глобальный параметр warnPartialMatchDollar равным TRUE:

```
options(warnPartialMatchDollar = TRUE)
x$a
#> Warning in x$a: partial match of 'a' to 'abc'
#> [1] 1
```

Применительно к датафреймам подобной проблемы можно избежать, если использовать тибблы, в которых никогда не проверяется частичное соответствие.

4.3.3 Отсутствующие и выходящие за границы индексы

Всегда необходимо понимать, как поведет себя оператор [[при передаче ему некорректного индекса. В табл. 4.1 сведены шаблоны поведения при извлечении подмножеств в логическом векторе, списке и значении NULL с помощью передачи оператору [[объекта нулевой длины (как NULL или logical()), значения, выходящего за пределы (OOB – *out of bounds*), и отсутствующего значения (как NA_integer_). Каждая ячейка таблицы содержит результат извлечения подмножества применительно к объекту, показанному слева в строке, с помощью индекса, отображенного сверху в колонке. Я показал результаты только для логических векторов, но другие атомарные векторы будут вести себя точно так же, возвращая элементы того же типа (int = integer (целочисленный); chr = character (символьный)).

Таблица 4.1 Результаты извлечения подмножеств с помощью разных значений

row[[col]]	Нулевая длина	OOB (int)	OOB (chr)	Пропущенные значения
Атомарный	Ошибка	Ошибка	Ошибка	Ошибка
Список	Ошибка	Ошибка	NULL	NULL
NULL	NULL	NULL	NULL	NULL

Если мы имеем дело с индексированием именованного вектора, имена для значений за границами, отсутствующих значений и NULL будут <NA>.

Противоречия, которые вы видите в табл. 4.1, привели к появлению функций purrr::pluck() и purrr::chuck(). В случае если элемент отсутствует, функция pluck() всегда возвращает NULL (или значение аргумента .default), а функция chuck() выбрасывает ошибку. Поведение функции pluck() идеально подходит для индексирования структур данных с большой степенью вложенности, где искомый компонент может отсутствовать вовсе (как при работе с данными в формате JSON, пришедшими из внешнего API). Функция pluck() также позволяет смешивать целочисленные и символьные индексы и предлагает указать значение по умолчанию на случай отсутствия искомого значения:

```
x <- list(
    a = list(1, 2, 3),
    b = list(3, 4, 5)
)

purrr::pluck(x, "a", 1)
#> [1] 1

purrr::pluck(x, "c", 1)
#> NULL

purrr::pluck(x, "c", 1, .default = NA)
#> [1] NA
```

4.3.4 @ и slot()

Существует еще два дополнительных оператора для извлечения подмножеств, которые применяются к объектам S4: это @ (аналог $) и slot() (аналог [[). Оператор @ обладает большим количеством ограничений по сравнению с $, к примеру он возвращает ошибку в случае неудачного поиска. Подробнее об этом мы будем говорить в главе 15.

4.3.5 Упражнения

1. Назовите как можно больше способов для извлечения третьего по порядку значения из переменной cyl в наборе данных mtcars.
2. Воспользовавшись линейной моделью (например, такой: mod <- lm(mpg ~ wt, data = mtcars)), извлеките значение остаточных степеней свободы (df.residual). После этого извлеките коэффициент детерминации (R-squared) из общей сводки по модели (summary(mod)).

4.4 Извлечение множеств и присваивание

Все операторы извлечения подмножеств могут использоваться совместно с операторами *присваивания* (assignment) с целью изменения избранных значений входного вектора. Основная форма записи при этом такая: `x[i] <- value`:

```
x <- 1:5
x[c(1, 2)] <- c(101, 102)
x
#> [1] 101 102   3   4   5
```

Я рекомендую вам проверять, чтобы `length(value)` была равна `length(x[i])`, а `i` был уникальным. Хотя R и будет применять в случае необходимости правила переписывания, о которых мы упоминали ранее, в целом лучше стараться обходиться без этих сложностей, которые непременно возникнут, особенно если `i` будет содержать пропущенные или дублирующиеся значения.

Применительно к спискам вы можете воспользоваться инструкцией `x[[i]] <- NULL`, чтобы избавиться от одного из элементов. Для добавления значений `NULL` необходимо воспользоваться конструкцией `x[i] <- list(NULL)`:

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

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

Извлечение пустого подмножества может пригодиться при совместном использовании с оператором присваивания ввиду сохранения структуры исходного объекта. Сравните приведенные ниже выражения. В первом случае в переменной `mtcars` по-прежнему будет находиться датафрейм, поскольку вы обозначили изменение содержимого переменной, а не ее самой. Во втором случае в переменной `mtcars` окажется уже список из-за изменения объекта привязки:

```
mtcars[] <- lapply(mtcars, as.integer)
is.data.frame(mtcars)
#> [1] TRUE

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

```
is.data.frame(mtcars)
#> [1] FALSE
```

4.5 Применение

Принципы и концепции, описанные выше, широко применяются в приложениях на языке R. Ниже мы приведем несколько наиболее важных аспектов их использования. Хотя многие базовые принципы извлечения подмножеств изначально заложены в удобные функции вроде `subset()`, `merge()` и `dplyr::arrange()`, доскональное понимание того, как именно работают эти принципы, поможет вам в ситуациях, когда нужной функции под рукой не окажется.

4.5.1 Таблицы поиска (символьное извлечение подмножеств)

Символьное сопоставление (character matching) – это очень мощный инструмент создания *таблиц поиска* (lookup table). Скажем, вам необходимо преобразовать некие сокращения в векторе данных:

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

Если вам не хочется, чтобы в результирующем наборе данных присутствовали имена, вы можете воспользоваться функцией `unname()` для их удаления.

```
unname(lookup[x])
#> [1] "Male" "Female"       NA "Female" "Female"   "Male"   "Male"
```

4.5.2 Сопоставление и объединение в ручном режиме (целочисленное извлечение подмножеств)

Также у вас могут быть более сложные таблицы поиска со множеством колонок. Представьте, что вы располагаете вектором целочисленных оценок и таблицей, описывающей их характеристики:

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

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

```
    fail = c(F, F, T)
)
```

Теперь предположим, что вам понадобилось продублировать информацию из таблицы таким образом, чтобы для каждого значения из вектора `grades` у нас появилась целая строка с описанием оценки. Можно очень элегантно решить эту задачу путем комбинирования функции `match()` и целочисленного извлечения подмножеств (`match(needles, haystack)` возвращает позиции, в которых были встречены значения из вектора `needles` (иголки) в структуре данных `haystack` (стог сена)).

```
id <- match(grades, info$grade)
id
#> [1] 3 2 2 1 3
info[id, ]
#>     grade      desc  fail
#> 3       1      Poor  TRUE
#> 2       2      Good FALSE
#> 2.1     2      Good FALSE
#> 1       3 Excellent FALSE
#> 3.1     1      Poor  TRUE
```

Если вам необходимо осуществить сопоставление по нескольким колонкам, вам нужно будет сначала свернуть их в одну колонку (например, с помощью функции `interaction()`). Но лучше будет обратить взор на функции, специально созданные для объединения таблиц, такие как `merge()` или `dplyr::left_join()`.

4.5.3 Случайные выборки и бутстрэпы (целочисленное извлечение подмножеств)

Целочисленные индексы могут быть использованы для случайной выборки или бутстрэпирования данных на основе векторов или датафреймов. Просто воспользуйтесь функцией `sample(n)` для создания случайного распределения значений из диапазона `1:n`, после чего примените полученные в результате данные для извлечения подмножества значений:

```
df <- data.frame(x = c(1, 2, 3, 1, 2), y = 5:1, z = letters[1:5])

# Случайное упорядочивание
df[sample(nrow(df)), ]
#>   x y z
#> 1 1 5 a
#> 4 1 2 d
#> 2 2 4 b
#> 5 2 1 e
#> 3 3 3 c
```

```
# Выбор трех случайных строк
df[sample(nrow(df), 3), ]
#>   x y z
#> 3 3 3 c
#> 2 2 4 b
#> 1 1 5 a

# Выбор шести случайных строк с возвратами (бутстрэп)
df[sample(nrow(df), 6, replace = TRUE), ]
#>     x y z
#> 4   1 2 d
#> 4.1 1 2 d
#> 5   2 1 e
#> 1   1 5 a
#> 1.1 1 5 a
#> 2   2 4 b
```

С помощью аргументов функции `sample()` задается количество элементов в случайной выборке и возможность повторного выбора элементов (`replace`).

4.5.4 Упорядочивание (целочисленное извлечение подмножеств)

Функция `order()` принимает на вход вектор и возвращает целочисленный вектор, соответствующий индексам элементов с учетом упорядочивания[1]:

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

Для решения проблем с одинаковыми индексами можно передать функции `order()` дополнительные переменные. Также вы можете изменить порядок сортировки элементов на обратный с помощью аргумента `decreasing = TRUE`. По умолчанию все элементы с пропущенными значениями будут скапливаться в конце вектора, но вы можете удалить их, указав аргумент `na.last = NA`, или разместить в начале вектора, прописав значение параметра `na.last = FALSE`.

Для объектов с двумя и более измерениями функция `order()` и целочисленное извлечение подмножеств облегчают задачу упорядочивания по строкам или столбцам:

```
# Случайным образом упорядоченный датафрейм
df2 <- df[sample(nrow(df)), 3:1]
```

[1] Это так называемые индексы упорядочивания, т. е. значение `order(x)[i]` соответствует индексу элемента `x[i]` в исходном векторе.

```
df2
#>   z y x
#> 3 c 3 3
#> 1 a 5 1
#> 2 b 4 2
#> 4 d 2 1
#> 5 e 1 2

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

Вы можете упорядочивать векторы напрямую с помощью функции `sort()` или воспользоваться функцией `dplyr::arrange()` для упорядочивания датафреймов.

4.5.5 Разворачивание агрегированных данных (целочисленное извлечение подмножеств)

Иногда нам приходится работать с данными, в которых дублирующиеся строки собраны вместе с добавленным количеством повторов в отдельном столбце. С помощью функции `rep()` и целочисленного извлечения подмножеств можно легко развернуть такие данные, воспользовавшись векторизованной природой `rep()`: выражение `rep(x, y)` повторяет `x[i]` `y[i]` раз.

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

df[rep(1:nrow(df), df$n), ]
#>     x  y n
#> 1   2  9 3
#> 1.1 2  9 3
#> 1.2 2  9 3
#> 2   4 11 5
#> 2.1 4 11 5
#> 2.2 4 11 5
#> 2.3 4 11 5
```

```
#> 2.4 4 11 5
#> 3   1  6 1
```

4.5.6 Удаление столбцов из датафрейма (символьное извлечение подмножеств)

Существует два способа удаления колонки из датафрейма. Можно присвоить ненужной колонке значение NULL:

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

Также можно вернуть только нужные нам колонки:

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

Если же вам известны только имена столбцов, от которых вы хотите избавиться, можно воспользоваться функциями для работы со множествами для определения столбцов, которые нужно оставить:

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

4.5.7 Выбор строк по условию (логическое извлечение подмножеств)

Поскольку логическое извлечение подмножеств позволяет легко комбинировать условия по нескольким колонкам, его традиционно применяют для извлечения нужных строк из датафрейма.

```
mtcars[mtcars$gear == 5, ]
#>                 mpg  cyl  disp  hp drat   wt qsec vs am gear carb
#> Porsche 914-2  26.0    4 120.3  91 4.43 2.14 16.7  0  1    5    2
#> Lotus Europa   30.4    4  95.1 113 3.77 1.51 16.9  1  1    5    2
#> Ford Pantera L 15.8    8 351.0 264 4.22 3.17 14.5  0  1    5    4
#> Ferrari Dino   19.7    6 145.0 175 3.62 2.77 15.5  0  1    5    6
#> Maserati Bora  15.0    8 301.0 335 3.54 3.57 14.6  0  1    5    8

mtcars[mtcars$gear == 5 & mtcars$cyl == 4, ]
```

```
#>                mpg cyl  disp  hp drat   wt qsec vs am gear carb
#> Porsche 914-2 26.0   4 120.3  91 4.43 2.14 16.7  0  1    5    2
#> Lotus Europa  30.4   4  95.1 113 3.77 1.51 16.9  1  1    5    2
```

Помните о том, что нужно использовать векторные булевы операторы `&` и `|`, а не их скалярные аналоги `&&` и `||`, которые больше употребимы внутри выражений. И не забывайте про *законы де Моргана* (De Morgan's laws) (https://ru.wikipedia.org/wiki/Законы_де_Моргана), помогающие упростить выражение отрицания:

- `!(X & Y)` – это то же, что и `!X | !Y`;
- `!(X | Y)` – это то же, что и `!X & !Y`.

Таким образом, выражение `!(X & !(Y | Z))` может быть упрощено до `!X | !!(Y|Z)`, а затем до `!X | Y | Z`.

4.5.8 Булева алгебра против множеств (логическое и целочисленное извлечение подмножеств)

Очень важно помнить о естественной эквивалентности между операциями над множествами (целочисленное извлечение подмножеств) и булевой алгеброй (логическое извлечение подмножеств). Использование операций над множествами может быть более эффективным, если:

- вам необходимо найти первое (или последнее) значение `TRUE`;
- у вас есть очень много значений `TRUE` и очень мало `FALSE`; представление в виде множеств в этом случае может оказаться более быстрым и потребовать меньше памяти.

Функция `which()` позволяет преобразовать булево представление в целочисленное. В базовом пакете R не существует обратной операции, но нам ничто не мешает создать ее самостоятельно:

```
x <- sample(10) < 4
which(x)
#> [1] 2 5 8

unwhich <- function(x, n) {
    out <- rep_len(FALSE, n)
    out[x] <- TRUE
    out
}

unwhich(which(x), 10)
#> [1] FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE
```

Давайте создадим два логических вектора и их целочисленные эквиваленты, а затем посмотрим на связь между булевыми операциями и операциями над множествами.

```
(x1 <- 1:10 %% 2 == 0)
#> [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
(x2 <- which(x1))
#> [1]  2  4  6  8 10
(y1 <- 1:10 %% 5 == 0)
#> [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE
(y2 <- which(y1))
#> [1]  5 10

# X & Y <-> intersect(x, y)
x1 & y1
#> [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
intersect(x2, y2)
#> [1] 10

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

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

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

На начальной стадии изучения операций извлечения подмножеств разработчики часто допускают одну и ту же ошибку, используя выражение `x[which(y)]` вместо `x[y]`. В данном случае применение функции `which()` не несет никакой смысловой нагрузки: мы просто переключаемся с логического на целочисленное извлечение подмножеств, но результат остается прежним. В более общих случаях есть два важных отличия:

- когда логический вектор содержит значения `NA`, логическое извлечение подмножеств заменит их на `NA`, тогда как функция `which()` просто опустит их. Функцию `which()` нередко используют для вызова этого побочного эффекта, но я лично не рекомендую так делать, поскольку избавление от пропущенных значений – не дело функции `which()`;
- выражение `x[-which(y)]` **не** эквивалентно `x[!y]`: если в `y` содержатся только значения `FALSE`, выражение `which(y)` вернет `integer(0)`, и `-integer(0)` также останется `integer(0)`, так что вы не выберете ни одного значения, а не все значения.

В целом необходимо стараться избегать переключения с логического на целочисленное извлечение подмножеств, если вам не нужно, к примеру, получить первое или последнее значение TRUE.

4.5.9 Упражнения

1. Как бы вы случайным образом упорядочили столбцы в датафрейме (это важная техника, используемая в алгоритме случайного леса)? Можно ли одновременно переупорядочить строки и столбцы за один шаг?
2. Как бы выбрали m случайных строк из датафрейма? Что, если выборка должна быть непрерывной (т. е. состоять из начальной строки, конечной и всех строк между ними)?
3. Как бы вы отсортировали колонки в датафрейме в алфавитном порядке?

4.6 Ответы на контрольные вопросы

1. Положительные целочисленные индексы позволяют выбирать элементы на указанных позициях, отрицательные исключают выбор элементов. Использование логических векторов в качестве индексов приводит к сохранению элементов, соответствующих значениям TRUE, а с помощью символьных векторов можно выбрать элементы с сопоставлением имен.
2. Оператор [служит для выбора подсписков. На выходе всегда будет список. Использование этого оператора с одним целочисленным значением приведет к возвращению списка единичной длины. Оператор [[служит для выбора элемента внутри списка. Оператор $ является сокращенной версией оператора [[: выражение x$y эквивалентно x[["y"]].
3. Аргумент drop = FALSE используется при извлечении подмножеств из матриц, массивов и датафреймов при желании сохранить исходные размерности. В функциях всегда рекомендуется использовать это значение аргумента для единообразия возвращаемых значений.
4. Если x – это матрица, выражение x[] <- 0 заменит все элементы структуры нулями, сохранив при этом количество строк и столбцов. В то же время выражение x <- 0 заменит всю матрицу одним нулевым значением.
5. Именованный символьный вектор может выступать в роли таблицы поиска: c(x = 1, y = 2, z = 3)[c("y", "z", "x")].

5

Управляющие структуры

5.1 Введение

В R существует два инструмента для управления потоком выполнения программы: операторы выбора и операторы циклов. Операторы выбора, такие как `if` или `switch()`, позволяют запускать разные фрагменты кода в зависимости от определенного условия. В то же время операторы циклов, такие как `for` и `while`, служат для повторяющегося запуска кода, обычно в изменяющихся обстоятельствах. Смею предположить, что вы уже знакомы с этими базовыми конструкциями языка, так что я кратко остановлюсь на технических деталях, после чего переключусь на полезные, но редко используемые возможности этих инструментов.

Система состояний (сообщения, предупреждения и ошибки), которой будет посвящена глава 8, также предлагает глобальные управляющие конструкции.

Контрольные вопросы

Не хотите читать эту главу? Никаких проблем. Правильно ответьте на все поставленные вопросы – и спокойно переходите к следующей главе. Ответы находятся в разделе 5.4.

1. Какая разница между операторами `if` и `ifelse()`?
2. Каким окажется значение `y` в приведенном ниже коде, если `x` равен `TRUE`? А если `FALSE`? Ну а если `NA`?

```
y <- if (x) 3
```

3. Что вернет выражение `switch("x", x = , y = 2, z = 3)`?

Структура главы

- В разделе 5.2 мы погрузимся в нюансы использования условного оператора `if`, после чего обсудим применение родственных ему операторов `ifelse()` и `switch()`.

- Раздел 5.3 мы начнем с напоминаний о том, как устроен цикл `for` в R, затем пройдемся по возможным ловушкам при использовании циклов, после чего обсудим связанные операторы `while` и `repeat`.

5.2 Выбор

Базовая форма оператора `if` в языке R выглядит следующим образом:

```
if (condition) true_action
if (condition) true_action else false_action
```

Если условие `condition` вернет логическое значение `TRUE`, выполнится действие `true_action`; если условие `condition` вернет `FALSE`, выполнится необязательное действие `false_action`.

Обычно вместо действий присутствуют целые блоки кода, заключенные в фигурные скобки {}:

```
grade <- function(x) {
    if (x > 90) {
        "A"
    } else if (x > 80) {
        "B"
    } else if (x > 50) {
        "C"
    } else {
        "F"
    }
}
```

Сам оператор `if` возвращает значение, которое вы можете присвоить переменной, как показано ниже:

```
x1 <- if (TRUE) 1 else 2
x2 <- if (FALSE) 1 else 2

c(x1, x2)
#> [1] 1 2
```

Я рекомендую присваивать переменной значение, возвращаемое оператором `if`, только в случае если выражение целиком помещается в одну строку. В противном случае может быть затруднено чтение кода.

При использовании оператора `if` без продолжения в виде `else` он будет возвращать значение `NULL` в случае ложности условия. В связи с тем что функции вроде `c()` и `paste()` пропускают значения `NULL`, можно писать подобные приведенным ниже лаконичные идиомы:

```
greet <- function(name, birthday = FALSE) {
    paste0(
        "Hi ", name,
        if (birthday) " and HAPPY BIRTHDAY"
    )
}
greet("Maria", FALSE)
#> [1] "Hi Maria"
greet("Jaime", TRUE)
#> [1] "Hi Jaime and HAPPY BIRTHDAY"
```

5.2.1 Некорректные входные значения

Условие, передаваемое оператору `if`, должно возвращать одно из двух значений: `TRUE` или `FALSE`. В противном случае будет сгенерирована ошибка, как показано ниже:

```
if ("x") 1
#> Error in if ("x") 1: argument is not interpretable as logical
if (logical()) 1
#> Error in if (logical()) 1: argument is of length zero
if (NA) 1
#> Error in if (NA) 1: missing value where TRUE/FALSE needed
```

Исключение составляет логический вектор длиной больше единицы – в случае с ним будет выведено предупреждение:

```
if (c(TRUE, FALSE)) 1
#> Warning in if (c(TRUE, FALSE)) 1: the condition has length > 1 and
#> only the first element will be used
#> [1] 1
```

В R версии 3.5.0 и выше (спасибо Хенрику Бенгтссону (Henrik Bengtsson, https://github.com/HenrikBengtsson/Wishlist-for-R/issues/38)) можно превратить это предупреждение в ошибку, установив переменную окружения:

```
Sys.setenv("_R_CHECK_LENGTH_1_CONDITION_" = "true")
if (c(TRUE, FALSE)) 1
#> Error in if (c(TRUE, FALSE)) 1: the condition has length > 1
```

Думаю, это вполне логично, поскольку позволяет отловить ошибки, которые иначе можно упустить из виду.

5.2.2 Векторизованный if

Учитывая то, что оператор `if` работает только с одиночными значениями `TRUE` и `FALSE`, у вас может возникнуть резонный вопрос: а что делать с векторами, содержащими логические значения? А это уже вотчина оператора `ifelse()`,

представляющего векторизованный аналог оператора `if` со своими векторами `test`, `yes`, и `no`, которые преобразовываются к одной длине:

```
x <- 1:10
ifelse(x %% 5 == 0, "XXX", as.character(x))
#> [1] "1"   "2"   "3"   "4"   "XXX"   "6"   "7"   "8"   "9"   "XXX"

ifelse(x %% 2 == 0, "even", "odd")
#> [1] "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even" "odd"
#> [10] "even"
```

Отсутствующие значения в исходном векторе будут сохраняться и в выходном векторе.

Я рекомендую использовать оператор `ifelse()` только в случае, если векторы `yes` и `no` характеризуются одним и тем же типом данных, поскольку в обратном случае предсказать результат может быть проблематично. В разделе сайта https://vctrs.r-lib.org/articles/stability.html#ifelse можно почитать об этом подробнее.

Еще одним векторизованным аналогом оператора `if` является более обобщенная функция `dplyr::case_when()`. В ней допускается использование любого количества пар условие–вектор:

```
dplyr::case_when(
    x %% 35 == 0 ~ "fizz buzz",
    x %% 5 == 0 ~ "fizz",
    x %% 7 == 0 ~ "buzz",
    is.na(x) ~ "???",
    TRUE ~ as.character(x)
)
#> [1] "1"   "2"   "3"   "4"   "fizz" "6"   "buzz" "8"   "9"
#> [10] "fizz"
```

5.2.3 Оператор switch()

Оператор `switch()` весьма похож на `if`, но обладает облегченной структурой и пишется гораздо более компактно. Например, приведенную ниже громоздкую конструкцию с использованием `if`

```
x_option <- function(x) {
    if (x == "a") {
        "option 1"
    } else if (x == "b") {
        "option 2"
    } else if (x == "c") {
        "option 3"
    } else {
        stop("Invalid `x` value")
    }
}
```

можно компактно переписать с использованием оператора switch():

```
x_option <- function(x) {
    switch(x,
        a = "option 1",
        b = "option 2",
        c = "option 3",
        stop("Invalid `x` value")
    )
}
```

Последний компонент оператора switch() всегда должен генерировать ошибку, в противном случае будет невидимо возвращено значение NULL:

```
(switch("c", a = 1, b = 2))
#> NULL
```

Если сразу несколько вариантов условий должны возвращать одинаковые значения, вы можете оставить правую часть от знака равенства пустой, и значение будет извлечено из следующего непустого условия. Это напоминает поведение оператора switch из языка C:

```
legs <- function(x) {
    switch(x,
        cow = ,
        horse = ,
        dog = 4,
        human = ,
        chicken = 2,
        plant = 0,
        stop("Unknown input")
    )
}
legs("cow")
#> [1] 4
legs("dog")
#> [1] 4
```

Оператор switch() допустимо использовать с числовыми условиями, но читать код в этом случае будет непросто, к тому же могут возникать непредвиденные ошибки, если на вход подать не целочисленное значение. Лично я рекомендую применять этот оператор преимущественно к символьным входным значениям.

5.2.4 Упражнения

1. Вектор какого типа будет возвращен в каждом из приведенных ниже случаев вызова оператора ifelse()?

```
ifelse(TRUE, 1, "no")
ifelse(FALSE, 1, "no")
ifelse(NA, 1, "no")
```

Ознакомьтесь с документацией и выпишите соответствующие правила своими словами.

2. Почему приведенный ниже код работает?

```
x <- 1:10
if (length(x)) "not empty" else "empty"
#> [1] "not empty"

x <- numeric()
if (length(x)) "not empty" else "empty"
#> [1] "empty"
```

5.3 Циклы

Циклы `for` используются в R для осуществления итераций внутри вектора. Базовая форма цикла выглядит так:

```
for (item in vector) perform_action
```

Таким образом, для каждого элемента в векторе действие `perform_action` выполняется один раз, после чего изменяется значение `item` и стартует следующая итерация цикла.

```
for (i in 1:3) {
    print(i)
}
#> [1] 1
#> [1] 2
#> [1] 3
```

При выполнении итераций по вектору индексов принято использовать короткие имена переменных `i`, `j` или `k`.

Примечание. Оператор `for` назначает `item` в текущем окружении, перезаписывая переменную с тем же именем, если она существует:

```
i <- 100
for (i in 1:3) {}
i
#> [1] 3
```

Существует два способа прервать выполняющийся цикл досрочно:

- оператор `next` немедленно переходит к следующей итерации цикла;
- оператор `break` немедленно завершает цикл.

```
for (i in 1:10) {
    if (i < 3)
        next
    print(i)
    if (i >= 5)
        break
}
#> [1] 3
#> [1] 4
#> [1] 5
```

5.3.1 Распространенные ловушки

При использовании циклов `for` вас могут подстерегать три ловушки, в которые раз за разом попадают неопытные разработчики. Во-первых, если вы генерируете данные, заранее выделите память для выходного контейнера, иначе цикл будет работать очень медленно. В разделах 23.2.2 и 24.6 мы будем говорить об этом подробнее. Здесь вам поможет функция `vector()`

```
means <- c(1, 50, 20)
out <- vector("list", length(means))
for (i in 1:length(means)) {
    out[[i]] <- rnorm(10, means[[i]])
}
```

Также остерегайтесь проходить циклом по `1:length(x)`. Если `x` окажется вектором нулевой длины, вы получите непредсказуемый результат или ошибку:

```
means <- c()
out <- vector("list", length(means))
for (i in 1:length(means)) {
    out[[i]] <- rnorm(10, means[[i]])
}
#> Error in rnorm(10, means[[i]]): invalid arguments
```

Причина в том, что R работает как с восходящими, так и с нисходящими последовательностями:

```
1:length(means)
#> [1] 1 0
```

Вместо этого можно безопасно использовать функцию `seq_along(x)`, которая всегда возвращает значение той же длины, что и `x`:

```
seq_along(means)
#> integer(0)

out <- vector("list", length(means))
for (i in seq_along(means)) {
    out[[i]] <- rnorm(10, means[[i]])
}
```

Наконец, вы можете столкнуться с трудностями при итерировании по векторам S3, поскольку в циклах обычно обрезаются атрибуты:

```
xs <- as.Date(c("2020-01-01", "2010-01-01"))
for (x in xs) {
    print(x)
}
#> [1] 18262
#> [1] 14610
```

Обойти этот момент можно, используя оператор `[[` явно:

```
for (i in seq_along(xs)) {
    print(xs[[i]])
}
#> [1] "2020-01-01"
#> [1] "2010-01-01"
```

5.3.2 Сопутствующие инструменты

Циклы `for` применяются в случаях, когда вам заранее известен набор значений, по которым необходимо пройти. Если у вас такой информации нет, вы можете применить следующие более гибкие конструкции:

- `while(condition) action`: выполняет действие `action`, пока условие `condition` возвращает `TRUE`;
- `repeat(action)`: повторяет действие `action` бесконечно (т. е. пока не встретится оператор `break`).

В R отсутствует эквивалент синтаксиса `do {action} while (condition)`, который применяется во многих языках программирования.

Любой цикл `for` можно переписать с использованием оператора `while`, а любой цикл `while` – с использованием `repeat`. Обратная цепочка здесь не действует. Таким образом, можно сделать вывод, что оператор `while` является более гибким по сравнению с `for`, а `repeat` – более гибким по сравнению с `while`. При этом хорошим тоном считается применение наименее гибкого подхода из всех возможных применительно к конкретной задаче. Так что вы можете пользоваться оператором `for` всегда, когда это возможно.

В целом же в задачах, связанных с анализом данных, лучше не использовать циклы `for`, а везде, где это допустимо, заменять их на функции `map()`

и apply(), предоставляющие наименьшую и достаточную гибкость для большинства задач. Подробнее об этих функциях мы будем говорить в главе 9.

5.3.3 Упражнения

1. Почему приведенный ниже код выполняется без ошибок и предупреждений?

```
x <- numeric()
out <- vector("list", length(x))
for (i in 1:length(x)) {
    out[i] <- x[i] ^ 2
}
out
```

2. Что вы можете сказать о векторе, по которому выполняются итерации в следующем фрагменте кода?

```
xs <- c(1, 2, 3)
for (x in xs) {
    xs <- c(xs, x * 2)
}
xs
#> [1] 1 2 3 2 4 6
```

3. О чем говорит показанный ниже код, в котором значение индекса в цикле меняется?

```
for (i in 1:3) {
    i <- i * 2
    print(i)
}
#> [1] 2
#> [1] 4
#> [1] 6
```

5.4 Ответы на контрольные вопросы

1. Оператор if работает со скалярами, а ifelse() – с векторами.
2. При x, равном TRUE, значение y будет равно 3; при FALSE y = NULL; при NA оператор if выдаст ошибку.
3. В выражении switch() есть пропущенные значения, так что в результате мы получим 2. Подробности смотрите в разделе 5.2.3.

6

Функции

6.1 Введение

Если вы читаете эту книгу, значит, вероятно, написали уже не одну функцию на R и знаете, как использовать их во избежание дублирования фрагментов кода. В этой главе вы поймете, как можно превратить это утилитарное практическое знание в фундаментальное теоретическое осмысление. На этом пути вы встретитесь с разными интересными трюками и техниками, но не забывайте, что темы, освещенные в данной главе, помогут вам в освоении более сложных концепций, с которыми вы встретитесь в этой книге.

Контрольные вопросы

Ответьте на следующие вопросы, и если вы сделаете это правильно, можете смело переходить к следующей главе. Ответы вы найдете в разделе 6.9.

1. Какие три компонента функции вам известны?
2. Что вернет следующий фрагмент кода?

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

3. Как бы вы обычно записали следующий код?

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

4. Как можно облегчить чтение следующей строки кода?

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

5. Возникнет ли ошибка при запуске следующего кода? И почему?

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

6. Что такое инфиксная функция и как ее писать? Что такое замещающая функция и как ее использовать?
7. Как можно гарантировать запуск очистки вне зависимости от того, как завершилась функция?

Структура главы

- В разделе 6.2 описываются основы создания функций, три их основных компонента и примитивные функции, составляющие исключение из множества правил для функций.
- В разделе 6.3 обсуждаются сильные и слабые стороны трех видов комбинирования вызовов функций, используемых в R.
- Раздел 6.4 посвящен тому, как R находит значения, связанные с именами, т. е. правилам лексического поиска.
- В разделе 6.5 мы поговорим о важнейшем свойстве аргументов функций, а именно о том, что они вычисляются только в момент первого использования.
- В разделе 6.6 мы обсудим особый аргумент … (многоточие), позволяющий передавать дополнительные аргументы другой функции.
- В разделе 6.7 мы рассмотрим два основных способа завершения работы функции и научимся определять обработчик выхода, представляющий фрагмент кода, запускающийся по окончании работы функции вне зависимости от того, что стало причиной.
- В разделе 6.8 будут рассмотрены различные варианты маскировки обычного вызова функций в R и способ применения стандартной префиксной формы для лучшего понимания происходящего.

6.2 Основы функций

Для досконального понимания работы *функций* (function) в R вам необходимо усвоить две важнейшие идеи:

- функции могут быть разбиты на три компонента: аргументы, тело и окружение. Из каждого правила есть исключения, и в данном случае они выражены в виде небольшого набора примитивных базовых функций, реализованных на языке C;
- функции представляют собой объекты, такие же, как векторы.

6.2.1 Компоненты функций

Функции состоят из трех частей:

- `formals()` – список аргументов, с помощью которого происходит управление вызовом функции;
- `body()` – код, находящийся внутри функции;
- `environment()` – структура данных, служащая для определения того, как функция находит связанные с именами значения.

Список аргументов (formals) и *тело* (body) функции явно определяются при ее создании, а *окружение* (environment) указывается неявно на основе того, *где* вы определяете функцию. Окружение функции присутствует всегда, но выводится только тогда, когда функция определена не в глобальном окружении.

```
f02 <- function(x, y) {
    # Комментарий
    x + y
}

formals(f02)
#> $x
#>
#>
#> $y

body(f02)
#> {
#>     x + y
#> }

environment(f02)
#> <environment: R_GlobalEnv>
```

Я буду изображать функции так, как показано на рис. 6.1. Черная точка слева символизирует окружение. Справа располагаются два аргумента функции. Тело функции на рисунке не показано, поскольку оно обычно объемное и никак не помогает в понимании общей структуры функции.

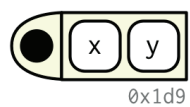


Рис. 6.1 Диаграмма отображения функции

Как и все объекты в R, функции могут обладать любым количеством дополнительных атрибутов (`attributes()`). Один из атрибутов, используемый R, – это `srcref` (сокр. от *source reference* – ссылка на источник). Он указывает

на исходный код, использующийся для создания функции. Атрибут `srcref` используется для вывода, поскольку, в отличие от `body()`, он содержит в себе комментарии и прочее форматирование.

```
attr(f02, "srcref")
#> function(x, y) {
#>     # Комментарий
#>     x + y
#> }
```

6.2.2 Примитивные функции

Существует одно исключение из правила о том, что функции состоят из трех компонентов, и оно выражается в *примитивных функциях* (primitive function) вроде `sum()` и `[`, которые вызывают код на языке C напрямую.

```
sum
#> function (..., na.rm = FALSE) .Primitive("sum")
`[`
#> .Primitive("[")
```

Типом таких функций является либо `builtin`, либо `special`.

```
typeof(sum)
#> [1] "builtin"
typeof(`[`)
#> [1] "special"
```

Эти функции присутствуют исключительно в C, а не в R, так что их компоненты `formals()`, `body()` и `environment()` возвращают `NULL`:

```
formals(sum)
#> NULL
body(sum)
#> NULL
environment(sum)
#> NULL
```

Примитивные функции располагаются в пакете `base`. И хотя они обладают некоторыми преимуществами в плане эффективности, это достигается ценой их плохой читаемости. Поэтому разработчики R в основном стараются избегать создания примитивных функций и делают это только в случае, когда без этого не обойтись.

6.2.3 Функции первого класса

Очень важно понимать, что в R функции являются полноценными объектами, и это свойство делает их *функциями первого класса* (first-class functions).

В отличие от многих других языков программирования, в R нет специального синтаксиса для определения и именования функций: вы просто создаете объект *функция* с помощью ключевого слова `function` и присваиваете его имени посредством оператора `<-`, что схематично показано на рис. 6.2:

```
f01 <- function(x) {
    sin(1 / x ^ 2)
}
```

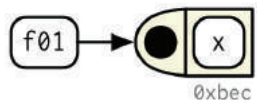


Рис. 6.2 Именованная функция

Хотя вы в основном будете присваивать созданную функцию определенному имени, по большому счету этот шаг является необязательным. Если вы решите не давать функции имя, то получите *анонимную функцию* (anonymous function). Это бывает полезно, когда вам нет необходимости обращаться к созданной функции в дальнейшем:

```
lapply(mtcars, function(x) length(unique(x)))
Filter(function(x) !is.numeric(x), mtcars)
integrate(function(x) sin(x) ^ 2, 0, pi)
```

Также вы можете объединять функции в списки:

```
funs <- list(
    half = function(x) x / 2,
    double = function(x) x * 2
)

funs$double(10)
#> [1] 20
```

В R вы часто будете сталкиваться с функциями, именуемыми *замыканиями* (closure). Под этим термином подразумеваются функции, которые захватывают (или замыкают) свои окружения, о чем мы подробнее поговорим в разделе 7.4.2.

6.2.4 Вызов функций

Обычно вызов функции происходит с указанием ее имени и списка аргументов, заключенных в скобки: `mean(1:10, na.rm = TRUE)`. Но что, если нужные вам аргументы уже заключены в отдельную структуру данных?

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

В этом случае вы можете воспользоваться вспомогательной функцией `do.call()`, которая принимает два аргумента: саму функцию и список, содержащий аргументы:

```
do.call(mean, args)
#> [1] 5.5
```

Мы вернемся к этой идее в разделе 19.6.

6.2.5 Упражнения

1. При наличии имени (например, `"mean"`) вспомогательная функция `match.fun()` поможет вам обнаружить искомую функцию. А если у вас есть функция, можно найти ее имя? Поясните, почему это не имеет смысла в R.
2. Вы можете (хотя это не совсем типично) вызывать анонимные функции. Какой из предложенных ниже синтаксисов является корректным и почему?

   ```
   function(x) 3()
   #> function(x) 3()
   (function(x) 3)()
   #> [1] 3
   ```

3. Правилом хорошего тона считается размещение тела анонимной функции в одной строке без использования фигурных скобок ({}). Проанализируйте свой код. Где вы могли бы воспользоваться анонимными функциями вместо именованных? И в каких случаях необходимо заменять анонимные функции именованными?
4. С помощью какой вспомогательной функции можно определить, что объект является функцией? А какой функцией можно воспользоваться, чтобы узнать, является ли функция примитивной?
5. Код, приведенный ниже, создает список из всех функций в пакете `base`:

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

   Используйте этот код для ответов на следующие вопросы:
   a) какая из базовых функций содержит наибольшее количество аргументов?
   b) у скольких базовых функций отсутствуют аргументы? Что в этих функциях особенного?
   c) как можно изменить этот код для нахождения всех примитивных функций?
6. Какие три компонента функции вам известны?
7. Когда в выводе функции не отображается окружение, в котором она была создана?

6.3 Комбинирование функций

В базовом R существует два основных способа *комбинирования* вызовов нескольких функций. Представьте, что вам необходимо рассчитать стандартное отклонение по популяции с использованием функций `sqrt()` и `mean()` в качестве строительных блоков:

```
square <- function(x) x^2
deviation <- function(x) x - mean(x)
```

Вы можете вложить вызовы функций один в другой, как показано ниже:

```
x <- runif(100)

sqrt(mean(square(deviation(x))))
#> [1] 0.274
```

Также вы можете сохранять промежуточные результаты в виде переменных:

```
out <- deviation(x)
out <- square(out)
out <- mean(out)
out <- sqrt(out)
out
#> [1] 0.274
```

В пакете `magrittr` [Баш (Bache) и Уикем, 2014] реализован третий способ комбинирования вызовов функций – с помощью бинарного оператора `%>%`, который называется *оператором конвейера* и произносится как «а затем...»:

```
library(magrittr)

x %>%
    deviation() %>%
    square() %>%
    mean() %>%
    sqrt()
#> [1] 0.274
```

Запись `x %>% f()` эквивалентна `f(x)`, а `x %>% f(y)` эквивалентна `f(x, y)`. Создание конвейера позволяет вам сосредоточиться на высокоуровневом комбинировании функций вместо отслеживания низкоуровневых потоков данных. Таким образом, акцент смещается на то, что *делается* (глагол), а не на то, какие *изменения* происходят (существительное). Такой подход общепринят в языках Haskell и F#, которые и вдохновили нас на создание пакета magrittr, и в языках программирования для стековой архитектуры вроде Forth и Factor.

У каждого из подходов есть свои преимущества и недостатки:

- вложение (`f(g(x))`) представляет собой довольно лаконичную форму и идеально подходит для коротких последовательностей. Но более длинные цепочки выглядят трудночитаемыми, в том числе из-за необходимости разворачивать их изнутри наружу и справа налево. В результате аргументы функций путаются и размазываются по всей длине выражения, подобно высокому многослойному сэндвичу;
- способ с использованием промежуточных результатов (`y <- f(x); g(y)`) требует именования всех временных переменных. Это является преимуществом, если вам нужны в использовании эти промежуточные вычисления. В противном случае, когда вам нужен только итог вычисления, это уже недостаток;
- использование конвейера (`x %>% f() %>% g()`) позволяет читать код в привычной манере – слева направо – и не требует задействования промежуточных переменных. Но этот способ применим только в случае линейной последовательности преобразований одного исходного объекта. Кроме того, он требует использования дополнительного пакета и предполагает, что разработчик, читающий исходный код, знаком с конвейерной нотацией.

В большинстве случаев в коде можно использовать комбинацию из всех трех подходов. Конвейеры наиболее распространены в анализе данных, поскольку большая часть анализа зачастую сводится к последовательности преобразований объектов (например, датафрейма или графика). Я не так часто использую конвейеры при написании пакетов – не потому, что это плохая идея, а потому, что код получается менее привычным.

6.4 Лексический поиск

В главе 2 мы говорили об операции присваивания, представляющей связывание имен со значениями. Здесь мы рассмотрим *поиск в области видимости* (scoping), заключающийся в нахождении значений, ассоциированных с именами.

Базовые правила выполнения поиска в области видимости довольно просты и интуитивно понятны – возможно, они покажутся вам знакомыми, даже если вы специально их не изучали. Рассмотрим пример, приведенный ниже. Что вернет код: 10 или 20[1]?

```
x <- 10
g01 <- function() {
    x <- 20
```

[1] Я буду «прятать» ответы на такие вопросы в сносках. Пытайтесь решить задачу сами и только после этого обращайтесь к сноске. Это поможет вам лучше понять тему и запомнить ответ. В данном случае вызов `g01()` вернет 20.

```
    x
}

g01()
```

В этом разделе мы поговорим о формальных правилах поиска в области видимости, а также о некоторых связанных с ними нюансах. Более глубокое понимание областей видимости позволит вам более свободно и гибко использовать инструменты функционального программирования и даже писать инструменты для преобразования кода на R в другие языки программирования.

В R используется *лексический поиск в области видимости* (lexical scoping)[1], при котором значения для имен ищутся на основе того, как функция определена, а не как она вызывается. Здесь слово *лексический* не относится к словам или символам, а означает чисто технический термин, согласно которому правила поиска опираются на структуру, основанную на времени разбора (парсинга), а не времени выполнения.

Лексический поиск в области видимости, применяемый в R, базируется на следующих четырех правилах:

- маскировка имен;
- функции против переменных;
- с чистого листа;
- динамический поиск.

6.4.1 Маскировка имен

Основной принцип лексического поиска в области видимости состоит в том, что имена, определенные в рамках функции, маскируют имена, определенные за ее пределами. Это проиллюстрировано в следующем примере.

```
x <- 10
y <- 20
g02 <- function() {
    x <- 1
    y <- 2
    c(x, y)
}
g02()
#> [1] 1 2
```

Если же имя не было определено внутри функции, R поднимется в поисках на уровень выше.

---

[1] Функции, использующие автоматическое цитирование одного или нескольких аргументов, могут переопределять тип поиска в области видимости. Подробнее об этом мы будем говорить в главе 20.

```
x <- 2
g03 <- function() {
    y <- 1
    c(x, y)
}
g03()
#> [1] 2 1

# Это никак не повлияет на предыдущее значение y
y
#> [1] 20
```

Эти же правила применяются при определении функции в рамках другой функции. Сперва R будет выполнять поиск в рамках текущей функции. После этого поиск продолжится там, где определена эта функция, и так по уровням вверх, вплоть до глобального окружения. Далее будет осуществляться поиск в загруженных пакетах.

Запустите приведенный ниже код сначала в голове, а затем убедитесь в правильности своих суждений, выполнив его в R[1].

```
x <- 1
g04 <- function() {
    y <- 2
    i <- function() {
        z <- 3
        c(x, y, z)
    }
    i()
}
g04()
```

Эти же правила применяются и к функциям, созданным другими функциями. О фабриках функций мы будем говорить в главе 10.

6.4.2 Функции против переменных

В R функции представляют собой обычные объекты. Это означает, что правила, описанные выше, будут применяться и к функциям:

```
g07 <- function(x) x + 1
g08 <- function() {
    g07 <- function(x) x + 100
    g07(10)
}
g08()
#> [1] 110
```

[1] Функция `g04()` вернет `c(1, 2, 3)`.

В то же время когда одно и то же имя относится к обычной переменной и к функции (разумеется, они должны располагаться в разных окружениях), правила лексического поиска становятся несколько сложнее. При использовании имени в вызове функции R игнорирует объекты, не являющиеся функциями. Посмотрите на пример ниже, в котором имя `g09` ссылается на два разных значения:

```
g09 <- function(x) x + 100
g10 <- function() {
    g09 <- 10
    g09(g09)
}
g10()
#> [1] 110
```

Естественно, желательно не допускать использования одного и того же имени для разных объектов.

6.4.3 С чистого листа

Что происходит со значениями между разными вызовами функции? Рассмотрим пример, приведенный ниже. Что произойдет при первом вызове функции? А при втором[1]? (Если вы ранее не встречались с функцией `exists()`, она служит для определения существования переменной. Если она существует, возвращается `TRUE`, в противном случае возвращается `FALSE`.)

```
g11 <- function() {
    if (!exists("a")) {
        a <- 1
    } else {
        a <- a + 1
    }
    a
}

g11()
g11()
```

Вероятно, вы были удивлены тому, что функция `g11()` при каждом запуске возвращает одно и то же значение. Это происходит по причине того, что для каждого запуска функции создается отдельное окружение. А значит, функция никак не может сообщить о том, что произошло в ней в предыдущий раз: каждый новый запуск выполняется независимо. В разделе 10.2.4 мы рассмотрим способы обхода этого ограничения.

---

[1] Функция `g11()` при каждом запуске будет возвращать единицу.

6.4.4 Динамический поиск

Лексический поиск определяет, где искать значения, но не когда. R осуществляет поиск значений в момент запуска функции, а не в момент ее создания. Таким образом, вывод функции может отличаться в зависимости от объектов за пределами окружения функции:

```
g12 <- function() x + 1
x <- 15
g12()
#> [1] 16

x <- 20
g12()
#> [1] 21
```

Такое поведение функций может сильно раздражать. Если вы допустите опечатку при написании функции, то не получите сообщение об ошибке в момент ее создания. А в зависимости от переменных, определенных в глобальном окружении, вы можете не получить сообщения об ошибке даже при запуске такой функции.

Для обнаружения подобных проблем можно воспользоваться функцией `codetools::findGlobals()`. Она выводит все внешние зависимости (непривязанные имена) для указанной функции:

```
codetools::findGlobals(g12)
#> [1] "+" "x"
```

Чтобы предотвратить такое поведение функций, вы можете вручную изменить окружение функции на `emptyenv()` – ничего не содержащее окружение:

```
environment(g12) <- emptyenv()
g12()
#> Error in x + 1: could not find function "+"
```

Проблема и ее решение демонстрируют, почему в R присутствует такое внешне нежелательное поведение: R полагается на лексический поиск абсолютно всегда – от простых ситуаций, когда нужно найти функцию вроде `mean()`, до менее очевидных, как в случае с поиском операторов `+` или `{`. Это придает правилам поиска в R такое обаяние простоты.

6.4.5 Упражнения

1. Что вернет следующий код? Почему? Опишите, как будут проинтерпретированы все три `c`.

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

2. Каковы четыре правила, которыми руководствуется R при поиске значений?
3. Что вернет следующая функция? Попробуйте предугадать поведение функции, прежде чем запускать код.

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

6.5 Ленивые вычисления

В языке R аргументы функций вычисляются в *ленивой* (lazy), или отложенной, манере, т. е. при первом доступе к ним. К примеру, приведенный ниже код не выдаст ошибку, потому что аргумент `x` так и не будет вычислен:

```
h01 <- function(x) {
    10
}
h01(stop("This is an error!"))
#> [1] 10
```

Это очень важная особенность, позволяющая передавать в функцию достаточно дорогие вычисления, которые будут произведены только при необходимости.

6.5.1 Промисы

Ленивые вычисления осуществляются при помощи структуры данных, называемой *промис* (promise). Реже используется англоязычный термин *thunk*. Это одна из особенностей R, делающих этот язык программирования столь мощным и привлекательным (мы еще вернемся к промисам в разделе 20.3).

Промис включает три компонента:

- выражение наподобие `x + y`, инициирующее отложенное вычисление;
- окружение, в котором выражение должно быть вычислено, т. е. окружение, в котором вызывается функция. Именно поэтому следующий вызов функции вернет не 101, а 11:

```
y <- 10
h02 <- function(x) {
    y <- 100
    x + 1
}

h02(y)
#> [1] 11
```

Это также означает, что при выполнении присваивания внутри вызова функции привязка осуществляется снаружи функции, а не внутри нее:

```
h02(y <- 1000)
#> [1] 1001
y
#> [1] 1000
```

- значение, которое рассчитывается и кешируется при первом доступе к промису, когда выражение вычисляется в указанном окружении. Это гарантирует вычисление промиса не более одного раза, и именно поэтому в приведенном ниже коде слово `Calculating...` выводится на экран лишь раз.

```
double <- function(x) {
    message("Calculating...")
    x * 2
}

h03 <- function(x) {
    c(x, x)
}

h03(double(x))
#> Calculating...
#> [1] 40 40
```

Вы не можете управлять промисами в коде R. Промис по своей сути похож на квантовое состояние: любая попытка исследовать его приводит к его немедленному вычислению и последующему исчезновению. Позже, в разделе 20.3, вы узнаете о структуре данных *quosure*, способной преобразовывать промисы в объекты R с возможностью исследовать выражения и окружения.

6.5.2 Аргументы по умолчанию

Благодаря отложенным вычислениям значения аргументов по умолчанию могут основываться на значениях предыдущих аргументов и даже на значениях переменных, определенных внутри функции до первого обращения к этому аргументу:

```
h04 <- function(x = 1, y = x * 2, z = a + b) {
    a <- 10
    b <- 100

    c(x, y, z)
}

h04()
#> [1]   1   2  110
```

Многие базовые функции в R используют эту технику, но лично я делать этого не рекомендую. Это сильно затрудняет чтение кода: чтобы понять, *что* вернет функция, вам необходимо знать, в каком именно порядке вычисляются значения аргументов по умолчанию.

Окружение вычисления немного отличается для аргументов по умолчанию и аргументов, переданных пользователем, поскольку аргументы по умолчанию вычисляются внутри функции. Таким образом, внешне похожие вызовы функции могут приводить к совершенно разным результатам. Это можно продемонстрировать на таком примере:

```
h05 <- function(x = ls()) {
    a <- 1
    x
}

# ls() вычисляется внутри h05:
h05()
#> [1] "a" "x"

# ls() вычисляется в глобальном окружении:
h05(ls())
#> [1] "h05"
```

6.5.3 Пропущенные аргументы

Для определения того, используется ли значение аргумента по умолчанию или переданное пользователем явно, можно воспользоваться функцией `missing()`:

```
h06 <- function(x = 10) {
    list(missing(x), x)
}
str(h06())
#> List of 2
#>  $ : logi TRUE
#>  $ : num 10
str(h06(10))
```

```
#> List of 2
#>  $ : logi FALSE
#>  $ : num 10
```

Но этой функцией следует пользоваться с осторожностью. Взглянем на базовую функцию `sample()`. Сколько аргументов она требует?

```
args(sample)
#> function (x, size, replace = FALSE, prob = NULL)
#> NULL
```

Кажется, что она требует явной передачи двух первых аргументов – `x` и `size`, – но на самом деле если аргумент `size` не передан, функция `sample()` воспользуется функцией `missing()` для предоставления значения по умолчанию. Если бы я переписывал функцию `sample`, я бы явно использовал значение по умолчанию `NULL`, чтобы дать понять, что аргумент `size` не требуется, но может быть передан:

```
sample <- function(x, size = NULL, replace = FALSE, prob = NULL) {
    if (is.null(size)) {
        size <- length(x)
    }

    x[sample.int(length(x), size, replace = replace, prob = prob)]
}
```

А при наличии двоичной структуры, созданной с помощью инфиксной функции `%||%`, в которой левая часть выражения будет использоваться только в случае, если она не равна `NULL`, а иначе правая, можно еще больше упростить функцию `sample()`:

```
`%||%` <- function(lhs, rhs) {
    if (!is.null(lhs)) {
        lhs
    } else {
        rhs
    }
}

sample <- function(x, size = NULL, replace = FALSE, prob = NULL) {
    size <- size %||% length(x)
    x[sample.int(length(x), size, replace = replace, prob = prob)]
}
```

Благодаря ленивым вычислениям вам нет повода беспокоиться о выполнении ненужных расчетов: правая часть оператора `%||%` будет вычисляться, только если в левой будет `NULL`.

6.5.4 Упражнения

1. Какое важное свойство оператора `&&` позволяет функции `x_ok()` нормально работать?

```
x_ok <- function(x) {
    !is.null(x) && length(x) == 1 && x > 0
}

x_ok(NULL)
#> [1] FALSE
x_ok(1)
#> [1] TRUE
x_ok(1:3)
#> [1] FALSE
```

   Чем отличается приведенный ниже код? Почему такое поведение здесь нежелательно?

```
x_ok <- function(x) {
    !is.null(x) & length(x) == 1 & x > 0
}

x_ok(NULL)
#> logical(0)
x_ok(1)
#> [1] TRUE
x_ok(1:3)
#> [1] FALSE FALSE FALSE
```

2. Что вернет эта функция? Почему? Какой принцип здесь демонстрируется?

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

3. Что вернет эта функция? Почему? Какой принцип здесь демонстрируется?

```
y <- 10
f1 <- function(x = {y <- 1; 2}, y = 0) {
    c(x, y)
}
f1()
y
```

4. В функции `hist()` значением по умолчанию для аргумента `xlim` является `range(breaks)`, значением по умолчанию для `breaks` является `"Sturges"`, и

```
range("Sturges")
#> [1] "Sturges" "Sturges"
```

   Поясните, что делает функция `hist()` для получения корректного значения аргумента `xlim`.

5. Объясните, почему эта функция работает. Почему она не так очевидна?

```
show_time <- function(x = stop("Error!")) {
    stop <- function(...) Sys.time()
    print(x)
}
show_time()
#> [1] "2019-04-04 11:48:56 CDT"
```

6. Сколько аргументов требуется передать функции `library()`?

6.6 ... (точка–точка–точка)

Функции могут принимать особый аргумент в виде трех точек (...). Он позволяет функции принимать любое количество дополнительных аргументов. В других языках программирования такой тип аргумента обычно именуется *varargs* (от *variable arguments* – переменное количество аргументов), а функция, использующая его, называется *функцией с переменным количеством аргументов* (*variadic*).

Вы можете использовать многоточие для передачи дополнительных аргументов в другую функцию, как показано ниже.

```
i01 <- function(y, z) {
    list(y = y, z = z)
}

i02 <- function(x, ...) {
    i01(...)
}

str(i02(x = 1, y = 2, z = 3))
#> List of 2
#>  $ y: num 2
#>  $ z: num 3
```

Используя особую форму этого аргумента, `..N`, можно (хотя это редко применяется) ссылаться на элементы аргумента ... по номеру позиции:

```
i03 <- function(...) {
    list(first = ..1, third = ..3)
}
str(i03(1, 2, 3))
#> List of 2
#>  $ first: num 1
#>  $ third: num 3
```

Гораздо чаще разработчики прибегают к помощи конструкции `list(...)`, которая вычисляет аргументы и сохраняет их в виде списка:

```
i04 <- function(...) {
    list(...)
}
str(i04(a = 1, b = 2))
#> List of 2
#>  $ a: num 1
#>  $ b: num 2
```

(См. также функцию `rlang::list2()`, поддерживающую сращивание списков и игнорирование завершающих запятых, и функцию `rlang::enquos()`, служащую для захвата невычисляемых аргументов, – эта тема связана с *квазицитированием* (quasiquotation).)

У аргумента … есть два распространенных типа использования, о которых мы подробно будем говорить далее в этой книге:

- если ваша функция принимает другую функцию в качестве аргумента, вам необходим какой-то способ передачи ей дополнительных аргументов. В примере, показанном ниже, функция `lapply()` использует … для передачи аргумента `na.rm` в функцию `mean()`:

  ```
  x <- list(c(1, 3, NA), c(4, NA, 6))
  str(lapply(x, mean, na.rm = TRUE))
  #> List of 2
  #>  $ : num 2
  #>  $ : num 5
  ```

  Мы вернемся к этой технике в разделе 9.2.3;

- если ваша функция является обобщенной функцией S3, вам нужен способ, который позволит методам принимать произвольное количество аргументов. Возьмем для примера функцию `print()`. Поскольку в зависимости от типа объекта вывод на экран может отличаться, невозможно заранее предусмотреть все возможные аргументы, а … позволяет каждому методу иметь свои аргументы:

  ```
  print(factor(letters), max.levels = 4)
  print(y ~ x, showEnv = TRUE)
  ```

  К такому использованию … мы вернемся в разделе 13.4.3.

Применение многоточия связано с двумя недостатками:

- когда вы используете ... для передачи аргументов другой функции, вы должны подробно объяснить пользователю, куда эти аргументы пойдут. Это осложняет понимание возможностей таких функций, как `lapply()` и `plot()`;
- аргумент, написанный с опечаткой, не приведет к возникновению ошибки. Это повышает риск появления трудно отлавливаемых багов:

```
sum(1, 2, NA, na_rm = TRUE)
#> [1] NA
```

6.6.1 Упражнения

1. Поясните следующие результаты:

```
sum(1, 2, 3)
#> [1] 6
mean(1, 2, 3)
#> [1] 1

sum(1, 2, 3, na.omit = TRUE)
#> [1] 7
mean(1, 2, 3, na.omit = TRUE)
#> [1] 1
```

2. Объясните, как найти документацию по именованным аргументам в следующем вызове функции:

```
plot(1:10, col = "red", pch = 20, xlab = "x", col.lab = "blue")
```

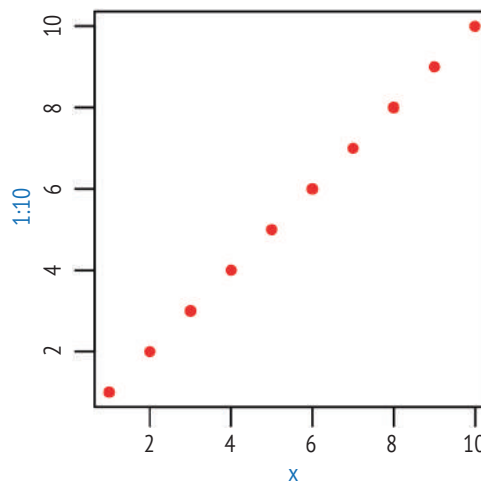


Рис. 6.3 График по десяти точкам данных

3. Почему вызов функции `plot(1:10, col = "red")` раскрашивает только точки данных, но не оси или метки? Прочитайте исходный код для метода `plot.default()`, чтобы разобраться.

6.7 Выход из функции

Большинство функций завершают свою работу одним из двух способов[1]: они либо возвращают значение, что символизирует успешное окончание работы, либо выбрасывают ошибку, что говорит о неудаче. В этом разделе мы поговорим о возвращаемых значениях (явных и неявных, видимых и невидимых), коротко коснемся ошибок и посмотрим на обработчики выхода, позволяющие запускать некий код при завершении работы функции.

6.7.1 Явный и неявный возврат

Существует два способа возврата значений из функции:

- *неявный* (implicit), при котором последнее вычисленное выражение и становится возвращаемым значением:

```
j01 <- function(x) {
    if (x < 10) {
        0
    } else {
        10
    }
}
j01(5)
#> [1] 0
j01(15)
#> [1] 10
```

- *явный* (explicit) – с использованием ключевого слова `return`:

```
j02 <- function(x) {
    if (x < 10) {
        return(0)
    } else {
        return(10)
    }
}
```

[1] Выход из функций может осуществляться и более экзотически: с помощью состояний, перехватываемых обработчиками, инициирования повторного запуска или нажатия на клавишу **Q** в интерактивном браузере.

6.7.2 Невидимые значения

Большинство функций возвращают видимые значения. Таким образом, их вызов в интерактивном контексте приводит к выводу результата.

```
j03 <- function() 1
j03()
#> [1] 1
```

Но этот вывод можно предотвратить, применив функцию `invisible()` к последнему значению:

```
j04 <- function() invisible(1)
j04()
```

Для проверки того, что это значение в действительности существует, его можно вывести на экран явно или обернуть в скобки:

```
print(j04())
#> [1] 1

(j04())
#> [1] 1
```

Также можно воспользоваться функцией `withVisible()` для возврата самого значения и флага видимости:

```
str(withVisible(j04()))
#> List of 2
#>  $ value  : num 1
#>  $ visible: logi FALSE
```

Наиболее распространенной функцией, использующей невидимый вывод, является `<-`:

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

Это позволяет выполнять цепные присваивания:

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

В общем случае любые функции, вызываемые главным образом ради побочного эффекта, такие как `<-`, `print()` или `plot()`, должны возвращать невидимое значение (обычно значение первого аргумента).

6.7.3 Ошибки

Если функция не может выполнить возложенные на нее задачи, она должна явным образом инициировать ошибку посредством функции `stop()`, что приводит к немедленному прекращению ее работы.

```
j05 <- function() {
    stop("Я ошибка!")
    return(10)
}
j05()
#> Error in j05(): Я ошибка!
```

Ошибка показывает, что что-то пошло не так, и вынуждает пользователя предпринимать действия, чтобы с ней справиться. В некоторых языках программирования, таких как C, Go и Rust, принято полагаться на особые значения, возвращаемые функциями, для обнаружения проблем в них, но в R всегда нужно выбрасывать ошибки. Далее в этой книге мы подробно поговорим об обработке возникающих ошибок.

6.7.4 Обработчики выхода

Иногда функции необходимо внести какие-то временные изменения в глобальное состояние. Но возврат к предыдущему состоянию может быть болезненным (а что, если возникнет ошибка?). Для гарантии того, что изменения были отменены, а глобальное состояние восстановилось, вне зависимости от того, как завершилась функция, можно воспользоваться функцией `on.exit()` для установки *обработчика выхода* (exit handler). В приведенном ниже простом примере видно, что обработчик выхода запускается независимо от того, завершилась функция нормально или с ошибкой.

```
j06 <- function(x) {
    cat("Hello\n")
    on.exit(cat("Goodbye!\n"), add = TRUE)

    if (x) {
        return(10)
    } else {
        stop("Error")
    }
}

j06(TRUE)
#> Hello
#> Goodbye!
#> [1] 10

j06(FALSE)
```

```
#> Hello
#> Error in j06(FALSE): Error
#> Goodbye!
```

Всегда при вызове функции `on.exit()` передавайте аргумент `add = TRUE`. В противном случае каждый вызов функции `on.exit()` будет перезаписывать предыдущий обработчик выхода. Даже при регистрации единственного обработчика выхода полезно устанавливать для аргумента `add` значение `TRUE`, чтобы в дальнейшем, если вам потребуется добавить новые обработчики, не возникло неприятных сюрпризов.

Функция `on.exit()` хороша тем, что позволяет добавить код очистки непосредственно за фрагментом, требующим очистки:

```
cleanup <- function(dir, code) {
    old_dir <- setwd(dir)
    on.exit(setwd(old_dir), add = TRUE)

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

Вкупе с отложенными вычислениями это создает удобную возможность для запуска блока кода в измененном окружении:

```
with_dir <- function(dir, code) {
    old <- setwd(dir)
    on.exit(setwd(old), add = TRUE)

    force(code)
}

getwd()
#> [1] "/Users/hadley/Documents/adv-r/adv-r"
with_dir("~", getwd())
#> [1] "/Users/hadley"
```

В использовании функции `force()` здесь нет особой необходимости, поскольку одной ссылки на код будет достаточно для его запуска. Функция `force()` говорит здесь скорее о нашем добровольном намерении запустить переданный код. В главе 10 мы поговорим о других способах использования функции `force()`.

Пакет `withr` [Эстер, 2018] предлагает ряд иных функций для установки временного состояния.

В R версии 3.4 и более ранних выражения `on.exit()` всегда запускаются в порядке их создания:

```
j08 <- function() {
    on.exit(message("a"), add = TRUE)
```

```
    on.exit(message("b"), add = TRUE)
}
j08()
#> a
#> b
```

Это может осложнить процесс очистки, если некоторые действия должны быть выполнены в строго определенном порядке. Зачастую требуется запускать последние добавленные блоки первыми. В R версии 3.5 и более поздних вы можете контролировать это поведение блоков очистки с помощью аргумента `after = FALSE`:

```
j09 <- function() {
    on.exit(message("a"), add = TRUE, after = FALSE)
    on.exit(message("b"), add = TRUE, after = FALSE)
}
j09()
#> b
#> a
```

6.7.5 Упражнения

1. Что возвращает функция `load()`? Почему мы обычно не видим этих значений?
2. Что возвращает функция `write.table()`? Можно ли придумать что-то более полезное?
3. Чем установка аргумента `chdir` в функции `source()` отличается от использования функции `in_dir()`? Какой вариант предпочли бы вы?
4. Напишите функцию, открывающую графическое устройство, запускающую переданный код и закрывающую графическое устройство (в любом случае вне зависимости от того, вывелся график или нет).
5. Функцию `on.exit()` можно использовать в качестве упрощенной версии функции `capture.output()`.

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

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

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

Сравните функции `capture.output()` и `capture.output2()`. Чем они отличаются? Что я удалил, чтобы сделать ключевые идеи более очевидными? Как я переписал сами ключевые идеи, чтобы их было легче понять?

6.8 Формы записи функций

Для лучшего понимания вычислений в R нужно усвоить две вещи:
- *все существующее – объекты;*
- *все происходящее – вызовы функций.*

– Джон Чемберс (John Chambers)

Хотя все происходящее в R является результатом вызовов функций, не все вызовы похожи друг на друга. Всего существует четыре *формы записи функций*:

- *префиксная* (prefix): имя функции располагается перед аргументами: `foofy(a, b, c)`. Большинство функций в R соответствуют этой форме;
- *инфиксная* (infix): имя функции располагается между аргументами, как в примере `x + y`. Такая форма записи используется для большинства математических операторов, а также для пользовательских функций, начинающихся и заканчивающихся символом `%`;
- *замещающая* (replacement): функции, изменяющие значения посредством операции присваивания, например `names(df) <- c("a", "b", "c")`. На самом деле внешне они похожи на префиксные функции;
- *особая* (special): функции вроде `[[`, `if` и `for`. Хотя эта форма записи не имеет характерной структуры, представленные функции играют существенную роль в языке R.

При существовании четырех форм записи функций вам в действительности необходимо знать только одну, поскольку любой вызов можно преобразовать к префиксной форме. В следующем разделе я продемонстрирую эту возможность, а затем мы перейдем к подробному разбору каждой формы записи.

6.8.1 Преобразование в префиксную форму записи

Любопытная особенность языка R состоит в том, что любая из перечисленных выше форм записи функций может быть переписана в виде префиксной нотации. Уметь делать это очень полезно, поскольку это помогает лучше понять структуру языка, узнать истинные имена функций и изменять поведение функций при необходимости (или забавы ради).

Ниже показаны три пары эквивалентных вызовов с преобразованием инфиксной, замещающей и особой форм записи функций в префиксную.

```
x + y
`+`(x, y)

names(df) <- c("x", "y", "z")
`names<-`(df, c("x", "y", "z"))

for(i in 1:10) print(i)
`for`(i, 1:10, print(i))
```

Как это ни удивительно, но в R цикл `for` действительно может быть запущен с помощью традиционной записи в виде функции! То же самое относится практически ко всем операциям в языке, а это означает, что, зная имя функции для непрефиксной записи, вы можете легко переопределить ее поведение. Если вы как-нибудь вдруг сильно разозлитесь на своего коллегу, запустите приведенный ниже код на его компьютере в его отсутствие. Результат будет очень забавным: в 10 % случаев к выражениям, указанным в скобках, будет добавляться единица.

```
`(` <- function(e1) {
    if (is.numeric(e1) && runif(1) < 0.1) {
        e1 + 1
    } else {
        e1
    }
}
replicate(50, (1 + 2))
#> [1] 3 3 3 3 3 3 3 3 3 3 3 3 4 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
#> [33] 3 3 3 3 3 3 4 3 4 3 3 3 3 4 3 3 3 3
rm("(")
```

Разумеется, в общем случае переопределять поведение встроенных функций – это плохая идея, но, как вы узнаете в разделе 21.2.5, можно применять изменения только к выборочным фрагментам кода. Это позволяет легко и элегантно писать языки для конкретной предметной области и трансляторы между языками.

Более полезное применение этого свойства можно найти при использовании инструментов функционального программирования. К примеру, вы могли бы воспользоваться функцией `lapply()` для прибавления трех к каждому элементу списка, сперва определив функцию `add()`:

```
add <- function(x, y) x + y
lapply(list(1:3, 4:5), add, 3)
#> [[1]]
#> [1] 4 5 6
#>
#> [[2]]
#> [1] 7 8
```

Но того же результата можно добиться, положившись на уже существующую функцию с емким именем +:

```
lapply(list(1:3, 4:5), `+`, 3)
#> [[1]]
#> [1] 4 5 6
#>
#> [[2]]
#> [1] 7 8
```

Подробно эту концепцию мы рассмотрим в главе 9.

6.8.2 Префиксная форма

Префиксная (prefix) форма записи функций является наиболее распространенной в R, как и в большинстве других языков программирования. При этом префиксная нотация в R немного отличается по причине возможности указания аргументов тремя разными способами:

- по позиции, как в `help(mean)`;
- по частичному соответствию, как, например, в `help(top = mean)`;
- по имени, как в `help(topic = mean)`.

Как показано во фрагменте кода ниже, аргументы сначала сопоставляются по полному имени, затем по уникальным префиксам и после этого – по позиции.

```
k01 <- function(abcdef, bcde1, bcde2) {
    list(a = abcdef, b1 = bcde1, b2 = bcde2)
}
str(k01(1, 2, 3))
#> List of 3
#>  $ a : num 1
#>  $ b1: num 2
#>  $ b2: num 3
str(k01(2, 3, abcdef = 1))
#> List of 3
#>  $ a : num 1
#>  $ b1: num 2
#>  $ b2: num 3

# Можно сокращать длинные имена аргументов:
str(k01(2, 3, a = 1))
#> List of 3
#>  $ a : num 1
#>  $ b1: num 2
#>  $ b2: num 3
```

```
# Но не в случае появления двусмысленности их именования
str(k01(1, 3, b = 1))
#> Error in k01(1, 3, b = 1): argument 3 matches multiple formal
#> arguments
```

В основном рекомендуется использовать позиционную передачу аргументов для первого или первых двух аргументов: они используются чаще других, и большинство читателей знают их предназначение. Избегайте применения позиционной передачи для редко используемых аргументов, а частичное соответствие не используйте никогда. К сожалению, невозможно отключить частичное соответствие полностью, но вы можете воспользоваться опцией `warnPartialMatchArgs`, позволяющей выводить предупреждения при обнаружении частичного соответствия:

```
options(warnPartialMatchArgs = TRUE)
x <- k01(a = 1, 2, 3)
#> Warning in k01(a = 1, 2, 3): partial argument match of 'a' to
#> 'abcdef'
```

6.8.3 Инфиксная форма

Инфиксная (infix) форма записи функций получила свое название из-за размещения функции между двумя аргументами. В R присутствует множество инфиксных функций, среди которых `:`, `::`, `:::`, `$`, `@`, `^`, `*`, `/`, `+`, `-`, `>`, `>=`, `<`, `<=`, `==`, `!=`, `!`, `&`, `&&`, `|`, `||`, `~`, `<-` и `<<-`. Также вы можете создавать собственные инфиксные функции, начинающиеся и заканчивающиеся символом `%`. В базовом R такой шаблон использован для определения следующих функций: `%%`, `%*%`, `%/%`, `%in%`, `%o%` и `%x%`.

Определять собственные инфиксные функции очень легко. Вы просто создаете функцию с двумя аргументами и связываете ее с именем, начинающимся и заканчивающимся символом `%`:

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

Имена инфиксных функций могут быть более гибкими по сравнению с обычными функциями в R: они могут содержать любую последовательность символов, за исключением `%`. При определении функций вам необходимо экранировать специальные символы в именах, а при их использовании такой надобности нет:

```
`% %` <- function(a, b) paste(a, b)
`%/\\%` <- function(a, b) paste(a, b)

"a" % % "b"
#> [1] "a b"
```

```
"a" %/\% "b"
#> [1] "a b"
```

Принятые в R правила старшинства предписывают выполнение инфиксных функций слева направо:

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

Есть две инфиксные функции, которые могут быть вызваны с единственным аргументом: это + и -.

```
-1
#> [1] -1
+10
#> [1] 10
```

6.8.4 Замещающая форма

Замещающая (replacement) форма записи функций предполагает выполнение функций так, как будто аргументы изменяются прямо на месте, с использованием особой формы имени `xxx<-`. У таких функций должно быть два аргумента `x` и `value`, и они должны возвращать измененный объект. К примеру, следующая функция изменяет второй элемент в векторе:

```
`second<-` <- function(x, value) {
    x[2] <- value
    x
}
```

Замещающие функции используются посредством их помещения слева от оператора `<-`:

```
x <- 1:10
second(x) <- 5L
x
#> [1] 1 5 3 4 5 6 7 8 9 10
```

Я написал, что замещающие функции выполняются так, *как будто* их аргументы изменяются прямо на месте, поскольку на самом деле, как было сказано в разделе 2.5, они просто создают их копии. Убедиться в этом можно с помощью функции `tracemem()`:

```
x <- 1:10
tracemem(x)
#> <0x7ffae71bd880>
```

```
second(x) <- 6L
#> tracemem[0x7ffae71bd880 -> 0x7ffae61b5480]:
#> tracemem[0x7ffae61b5480 -> 0x7ffae73f0408]: second<-
```

Если вашей замещающей функции нужны дополнительные аргументы, поместите их между аргументами `x` и `value` и передавайте их при вызове функции:

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

Когда вы пишете `modify(x, 1) <- 10`, R превращает ваше выражение в следующее:

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

Комбинирование замещающих функций с другими формами требует более сложного транслирования. Например, запись

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

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

транслируется в

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

Да, здесь действительно происходит создание временной переменной `*tmp*`, которая впоследствии удаляется.

6.8.5 Особая форма

Наконец, в R существует целый ряд языковых конструкций, которые записываются в *особой* (special) форме, но при этом также могут быть представлены в префиксной форме. Такие конструкции включают в себя:

- скобки:
  - `(x)` (`` `(`(x)``);
  - `{x}` (`` `{`(x)``);

- операторы для извлечения подмножеств:
  - `x[i] (`[`(x, i));`
  - `x[[i]] (`[[`(x, i));`
- инструменты управления потоком:
  - `if (cond) true (`if`(cond, true));`
  - `if (cond) true else false (`if`(cond, true, false));`
  - `for(var in seq) action (`for`(var, seq, action));`
  - `while(cond) action (`while`(cond, action));`
  - `repeat expr (`repeat`(expr));`
  - `next (`next`());`
  - `break (`break`());`
- наконец, самой сложной в этом ряду является функция по имени `function`:
  - `function(arg1, arg2) {body} (`function`(alist(arg1, arg2), body, env)).`

Знать истинное имя функции, использующееся в особой форме записи, бывает очень полезно для получения подробной документации: выражение `?(` вернет синтаксическую ошибку, тогда как `?`(`` выведет документацию по использованию круглых скобок.

Все функции с особой записью реализованы с помощью примитивных функций на языке C, а значит, их вывод будет неинформативным:

```
`for`
#> .Primitive("for")
```

6.8.6 Упражнения

1. Перепишите следующие выражения в виде префиксной формы записи:

```
1 + 2 + 3
1 + (2 + 3)
if (length(x) <= 5) x[[5]] else x[[n]]
```

2. Поясните подробности представленных ниже вызовов функций:

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

3. Почему следующий код завершается ошибкой?

```
modify(get("x"), 1) <- 10
#> Error: target of assignment expands to non-language object
```

4. Создайте замещающую функцию, которая изменяет случайный элемент в векторе.
5. Напишите свою версию функции +, которая будет конкатенировать переданные ей аргументы, если они являются строковыми векторами,

и выполнять обычную операцию в противном случае. Иными словами, сделайте так, чтобы приведенный ниже код работал корректно:

```
1 + 2
#> [1] 3
"a" + "b"
#> [1] "ab"
```

6. Создайте список из всех замещающих функций из пакета `base`. Какие из них являются примитивными функциями? Подсказка: воспользуйтесь функцией `apropos()`.
7. Какие имена допустимы для пользовательских инфиксных функций?
8. Создайте инфиксный оператор `xor()`.
9. Создайте инфиксные версии функций для работы со множествами `intersect()`, `union()` и `setdiff()`. Вы можете назвать их `%n%`, `%u%` и `%/%` с целью соответствия соглашениям, принятым в математике.

6.9 Ответы на контрольные вопросы

1. Три компонента функций – это тело, аргументы и окружение.
2. Выражение `f1(1)()` вернет 11.
3. Обычно вы бы написали его в инфиксном стиле: `1 + (2 * 3)`.
4. Если переписать выражение в виде `mean(c(1:10, NA), na.rm = TRUE)`, все будет гораздо понятнее.
5. Нет, ошибка не появится, поскольку второй аргумент так и не будет задействован.
6. См. разделы 6.8.3 и 6.8.4.
7. Использовать функцию `on.exit()`. Подробности – в разделе 6.7.4.

7

Окружения

7.1 Введение

Окружение (environment) представляет собой структуру данных, лежащую в основе поиска в области видимости. В этой главе мы погрузимся в обсуждение окружений, рассмотрим их структуру и узнаем, как с их помощью можно лучше понять четыре правила поиска в области видимости, описанные в разделе 6.4. Доскональное понимание окружений не является незаменимым навыком для ежедневного использования языка R. Но знать о существовании окружений крайне необходимо, поскольку на них базируются многие важные концепции языка, такие как лексический поиск, пространства имен и классы R6. Кроме того, они позволяют взаимодействовать с вычислениями, что открывает путь к написанию собственных диалектов для конкретной предметной области наподобие `dplyr` и `ggplot2`.

Контрольные вопросы

Если вы сможете правильно ответить на приведенные ниже вопросы, значит, вам незачем читать эту главу и можно двигаться дальше. Ответы на вопросы будут приведены в разделе 7.7.

1. Перечислите по крайней мере три отличия окружения от списка.
2. Что является родительским элементом для глобального окружения? У какого единственного окружения нет родителя?
3. Что является замыкающим окружением для функции? Почему это так важно?
4. Как определяется окружение, из которого была вызвана функция?
5. Чем отличаются операторы `<-` и `<<-`?

Структура главы

- В разделе 7.2 мы познакомимся с основными свойствами окружений и узнаем, как создавать свои окружения.
- В разделе 7.3 будет представлен шаблон удобной функции для работы с окружениями.

- В разделе 7.4 мы узнаем об особых окружениях, создаваемых в рамках пакетов, пространств имен, функций, а также для каждого вызова функции.
- Раздел 7.5 будет посвящен наиболее значимому окружению, а именно вызывающему. Здесь мы познакомимся со стеком вызовов, определяющим порядок вызовов функций. Вы наверняка сталкивались со стеком вызовов ранее, если пользовались функцией `traceback()` в целях отладки.
- В разделе 7.6 мы перечислим три случая использования окружений в виде удобных структур данных, помогающих решать определенные задачи.

Требования

В этой главе мы будем пользоваться функциями из пакета `rlang` (https://rlang.r-lib.org) для работы с окружениями, поскольку они позволяют сосредоточиться на самих окружениях, а не на сопутствующих деталях.

```
library(rlang)
```

Функции с префиксом `env_` из пакета `rlang` работают в виде конвейеров: все они принимают окружение в качестве первого аргумента, и многие из них также возвращают окружение. Я не буду использовать конвейеры в этой главе, чтобы код оставался максимально понятным, но вы в своих изысканиях вполне можете на них полагаться.

7.2 Основы окружений

В общем и целом окружения похожи на именованные списки с четырьмя важными отличиями:

- каждое имя должно быть уникальным;
- имена в окружениях не упорядочены;
- у окружения есть родитель;
- окружения не копируются при изменении.

Давайте погрузимся в эти идеи с фрагментами кода и сопутствующими рисунками.

7.2.1 Основы

Для создания окружения можно воспользоваться функцией `rlang::env()`. Она работает подобно функции `list()`, принимая на вход пары имя–значение:

```
e1 <- env(
    a = FALSE,
    b = "a",
```

```
    c = 2.3,
    d = 1:3,
)
```

Примечание. В базовом R для создания окружения можно использовать функцию `new.env()`. Игнорируйте аргументы `hash` и `size`, они вам не нужны. Вы не можете одновременно создавать и определять значения; воспользуйтесь конструкцией `$<-`, как показано ниже.

Назначение окружения состоит в том, чтобы *связывать* (bind) набор имен с набором значений. Таким образом, окружение можно представить как коробку с именами без установленного порядка (т. е. не имеет смысла интересоваться тем, какой из элементов в окружении первый). Внешне можно представить окружение так, как показано на рис. 7.1.

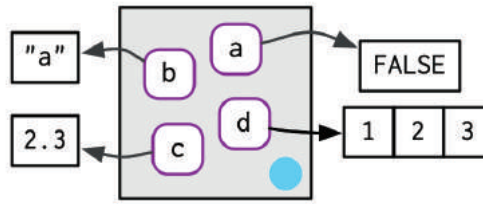


Рис. 7.1 Окружение

Как было сказано в разделе 2.5.2, окружения обладают ссылочной семантикой: в отличие от большинства объектов в R, при изменении окружения вы модифицируете их прямо на месте, без создания копии. Одним из важных следствий этого является то, что окружения могут содержать сами себя.

```
e1$d <- e1
```

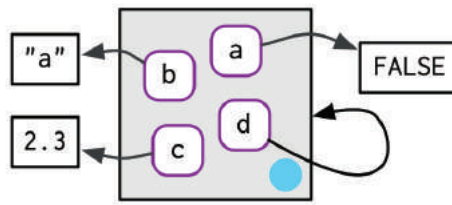


Рис. 7.2 Окружение, содержащее само себя

Вывод окружения на экран приведет к отображению только его адреса в памяти, что не слишком полезно:

```
e1
#> <environment: 0x7fba6eac4b08>
```

Вместо этого лучше воспользоваться функцией `env_print()`, которая предоставляет куда больше полезной информации:

```
env_print(e1)
#> <environment: 0x7fba6eac4b08>
#> parent: <environment: global>
#> bindings:
#>  * a: <lgl>
#>  * b: <chr>
#>  * c: <dbl>
#>  * d: <env>
```

Также можно прибегнуть к помощи функции `env_names()` для получения символьного вектора текущих привязок окружения:

```
env_names(e1)
#> [1] "a" "b" "c" "d"
```

Примечание. В базовом R версии 3.2.0 и выше вы можете использовать функцию `names()` для вывода списка привязок окружения. Если вашему коду предстоит работать с R версии 3.1.0 или более ранней, воспользуйтесь функцией `ls()`, но не забудьте указать значение аргумента `all.names = TRUE` для отображения всех привязок.

7.2.2 Важные окружения

В разделе 7.4 мы подробно поговорим об особых окружениях, а здесь лишь упомянем два из них. *Текущее окружение* (current environment), или `current_env()`, представляет собой окружение, в котором в данный момент выполняется код. Во время интерактивных экспериментов таковым обычно является *глобальное окружение* (global environment), или `global_env()`. Иногда глобальное окружение называют просто *рабочим пространством* (workspace), поскольку в нем выполняются все интерактивные вычисления за пределами функций.

Для сравнения окружений можно воспользоваться функцией `identical()`, а не оператором `==`. Причина в том, что оператор `==` является векторизованным, а окружения не являются векторами.

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

Примечание. В базовом R к глобальному окружению можно получить доступ с помощью функции `globalenv()`, а к текущему – с помощью `environment()`. Глобальное окружение отображается как `Rf_GlobalEnv` и `.GlobalEnv`.

7.2.3 Родители

У каждого окружения есть *родитель* (parent) в виде другого окружения. На наших диаграммах мы будем показывать наличие родителя у окружения в виде голубого кружка и стрелки, указывающей на родительское окружение, как на рис. 7.3. Родительское окружение используется при осуществлении лексического поиска: если имя не обнаруживается в текущем окружении, R продолжит поиск в родительском окружении и т. д. Родительское окружение можно задать, передав функции `env()` безымянный аргумент. Если его не указать, по умолчанию будет использоваться текущее окружение. В коде, показанном ниже, окружение `e2a` становится родительским для `e2b`.

```
e2a <- env(d = 4, e = 5)
e2b <- env(e2a, a = 1, b = 2, c = 3)
```

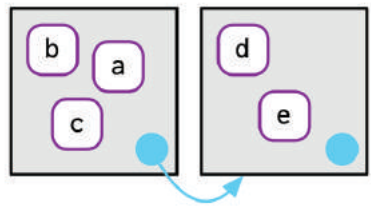


Рис. 7.3 Окружение `e2b` и его родитель `e2a`

Для экономии места я не буду в явном виде указывать всех родителей у всех окружений. Просто запомните: если видите на диаграмме голубой кружок, значит, у окружения где-то есть родитель.

Родительское окружение можно обнаружить с помощью функции `env_parent()`:

```
env_parent(e2b)
#> <environment: 0x7fba74178f48>
env_parent(e2a)
#> <environment: R_GlobalEnv>
```

Только у одного окружения не существует родителя – у *пустого* (empty). Пустое окружение я буду показывать на диаграммах с помощью полого голубого кружка, как на рис. 7.4, а где будет достаточно места – и с помощью метки `R_EmptyEnv`, которую R использует для отображения пустого окружения.

```
e2c <- env(empty_env(), d = 4, e = 5)
e2d <- env(e2c, a = 1, b = 2, c = 3)
```

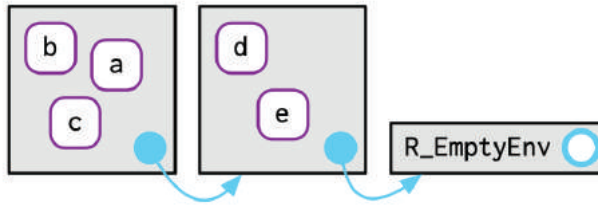


Рис. 7.4 Пустое окружение

Наследование окружений заканчивается при наличии в иерархии пустого окружения. Всех родителей окружения можно вывести с помощью функции `env_parents()`:

```
env_parents(e2b)
#> [[1]] <env: 0x7fba74178f48>
#> [[2]] $ <env: global>
env_parents(e2d)
#> [[1]] <env: 0x7fba74f70088>
#> [[2]] $ <env: empty>
```

По умолчанию эта функция останавливается при обнаружении в иерархии глобального окружения. Это бывает полезно, поскольку в родителях глобального окружения числятся все загруженные пакеты. Если вам необходимо увидеть полную иерархию, вы можете воспользоваться для этого аргументом `last`, как показано ниже. Мы вернемся к этим особым окружениям в разделе 7.4.1.

```
env_parents(e2b, last = empty_env())
#>  [[1]]   <env: 0x7fba74178f48>
#>  [[2]] $ <env: global>
#>  [[3]] $ <env: package:rlang>
#>  [[4]] $ <env: package:stats>
#>  [[5]] $ <env: package:graphics>
#>  [[6]] $ <env: package:grDevices>
#>  [[7]] $ <env: package:utils>
#>  [[8]] $ <env: package:datasets>
#>  [[9]] $ <env: package:methods>
#> [[10]] $ <env: Autoloads>
#> [[11]] $ <env: package:base>
#> [[12]] $ <env: empty>
```

Примечание. В базовом R вы можете воспользоваться функцией `parent.env()` для вывода родительского окружения. Всех родителей с использованием стандартных функций вам получить не удастся.

7.2.4 Присваивание в родительском окружении <<-

Наследственность окружений имеет прямое отношение к оператору присваивания в родительском окружении <<-. Оператор обычного присваивания, <-, всегда создает переменную в текущем окружении. В то же время оператор присваивания в родительском окружении, <<-, никогда не создает переменную в текущем окружении, а вместо этого изменяет значение переменной, найденной в родительском окружении.

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

Если оператору <<- не удается обнаружить искомую переменную, он создает ее в глобальном окружении. Обычно это нежелательно, поскольку *глобальные переменные* (global variables) приводят к возникновению неочевидных зависимостей между функциями. Чаще всего оператор <<- используется совместно с фабриками функций, что будет показано в разделе 10.2.4.

7.2.5 Получение и установка значений

Элементы окружений можно получать и устанавливать с помощью операторов $ и [[подобно тому, как это делается в списках:

```
e3 <- env(x = 1, y = 2)
e3$x
#> [1] 1
e3$z <- 3
e3[["z"]]
#> [1] 3
```

Но вы не можете использовать оператор [[с числовыми индексами, а также недопустимо применять одинарные квадратные скобки ([):

```
e3[[1]]
#> Error in e3[[1]]: wrong arguments for subsetting an environment
e3[c("x", "y")]
#> Error in e3[c("x", "y")]: object of type 'environment' is not
#> subsettable
```

Операторы $ и [[вернут NULL в случае отсутствия привязок. Если вам нужно получить ошибку, используйте функцию env_get():

```
e3$xyz
#> NULL
```

```
env_get(e3, "xyz")
#> Error in env_get(e3, "xyz"): object 'xyz' not found
```

Для получения значения по умолчанию в случае отсутствия привязки можно воспользоваться аргументом `default` функции `env_get()`:

```
env_get(e3, "xyz", default = NA)
#> [1] NA
```

Существует еще два способа добавления привязок к окружению:

- функция `env_poke()`[1] принимает на вход имя переменной (в виде строки) и значение:

```
env_poke(e3, "a", 100)
e3$a
#> [1] 100
```

- функция `env_bind()` позволяет выполнять множественную привязку значений:

```
env_bind(e3, a = 10, b = 20)
env_names(e3)
#> [1] "x" "y" "z" "a" "b"
```

Определить, есть ли у окружения какие-либо привязки, можно с помощью функции `env_has()`:

```
env_has(e3, "a")
#> a
#> TRUE
```

В отличие от списков, в окружениях присваивание элементу значения `NULL` не приводит к его удалению, поскольку иногда вам просто необходимо, чтобы определенное имя ссылалось на `NULL`. Для отвязки элемента от окружения можно использовать функцию `env_unbind()`:

```
e3$a <- NULL
env_has(e3, "a")
#>    a
#> TRUE

env_unbind(e3, "a")
```

[1] Вы можете удивиться, почему в пакете `rlang` эта функция называется `env_poke()`, а не `env_set()`. Причина – в преемственности. Дело в том, что функции с постфиксами `_set()` возвращают измененную копию, а постфикс `_poke()` говорит о модификации данных на месте.

```
env_has(e3, "a")
#>     a
#> FALSE
```

Отвязка значения не приводит к удалению объекта. Это задача сборщика мусора, который автоматически удаляет объекты без привязки к именам. Более детально данный процесс описывается в разделе 2.6.

Примечание. В базовом R аналогичными приведенным выше функциям являются функции `get()`, `assign()`, `exists()` и `rm()`. Они предназначены для работы в интерактивном режиме с текущим окружением, и использовать их с другими окружениями бывает сложновато. Также остерегайтесь аргумента `inherits`: по умолчанию он установлен в `TRUE`, а это означает, что при вызове функции будут исследоваться как переданное окружение, так и все его предки.

7.2.6 Продвинутые привязки

Существует два более экзотических аналога функции `env_bind()`:

- функция `env_bind_lazy()` создает *отложенную привязку* (delayed bindings), которая вычисляется при первом доступе к переменной. Фактически отложенная привязка реализована с помощью промисов, так что ведет она себя так же, как аргументы функций.

  ```
  env_bind_lazy(current_env(), b = {Sys.sleep(1); 1})

  system.time(print(b))
  #> [1] 1
  #>    user  system elapsed
  #>   0.001   0.000   1.003
  system.time(print(b))
  #> [1] 1
  #>    user  system elapsed
  #>       0       0       0
  ```

  Главным образом отложенная привязка используется при работе с функцией `autoload()`, позволяющей пакетам предоставлять наборы данных, которые ведут себя так, будто располагаются в памяти, хотя на самом деле загружаются с диска при необходимости;

- функция `env_bind_active()` создает *активные привязки* (active bindings), которые вычисляются всякий раз при доступе к ним:

  ```
  env_bind_active(current_env(), z1 = function(val) runif(1))

  z1
  #> [1] 0.0808
  ```

```
z1
#> [1] 0.834
```

Активные привязки используются при реализации активных полей в R6, о чем мы будем подробно говорить в разделе 14.3.2.

Примечание. В базовом R присутствуют специальные функции, справку по которым можно вызвать с помощью команд `?delayedAssign()` и `?makeActiveBinding()`.

7.2.7 Упражнения

1. Назовите три отличия окружений от списков.
2. Создайте окружение, показанное на рис. 7.5.

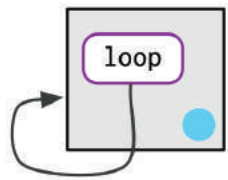


Рис. 7.5 Окружение, содержащее само себя

3. Создайте пару окружений, показанных на рис. 7.6.

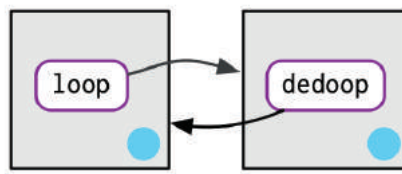


Рис. 7.6 Два связанных окружения

4. Расскажите, что не так с выражениями `e[[1]]` и `e[c("a", "b")]`, при условии что `e` – это окружение.
5. Напишите аналог функции `env_poke()`, которая будет осуществлять только новые привязки и не будет обновлять имеющиеся. В некоторых языках программирования, называемых *языками с однократным присваиванием* (single assignment languages), принята такая практика.
6. Что делает эта функция? Чем она отличается от оператора `<<-` и почему является более предпочтительной?

```
rebind <- function(name, value, env = caller_env()) {
    if (identical(env, empty_env())) {
```

```
        stop("Can't find `", name, "`", call. = FALSE)
    } else if (env_has(env, name)) {
        env_poke(env, name, value)
    } else {
        rebind(name, value, env_parent(env))
    }
}

rebind("a", 10)
#> Error: Can't find `a`
a <- 5
rebind("a", 10)
a
#> [1] 10
```

7.3 Рекурсия по окружениям

Если вам необходимо пройти по всей наследственной иерархии окружения, зачастую лучшим выходом будет написание рекурсивной функции. В этом разделе будет показано, как можно, основываясь на ваших знаниях об окружениях, написать функцию `where()`, которая, как ясно из ее имени, будет осуществлять поиск окружения, где определена заданная переменная, с использованием обычных правил и функций, описанных выше.

Определение функции `where()` будет весьма простым. На вход она будет принимать два аргумента: имя (в виде строки), которое необходимо найти, и окружение, с которого нужно начать поиск. В разделе 7.5 мы узнаем, почему функция `caller_env()` является лучшим кандидатом для значения по умолчанию аргумента с окружением.

```
where <- function(name, env = caller_env()) {
    if (identical(env, empty_env())) {
        # Базовый случай
        stop("Не могу найти имя ", name, call. = FALSE)
    } else if (env_has(env, name)) {
        # Успех
        env
    } else {
        # Рекурсивный случай
        where(name, env_parent(env))
    }
}
```

В этой функции перечислены три случая:

- базовый случай: мы дошли до пустого окружения, так и не найдя нужную привязку. Дальше идти некуда, так что возвращаем ошибку;
- успех: имя найдено в текущем окружении, и это окружение мы возвращаем из функции;

- рекурсивный случай: имя не найдено в текущем окружении, обратимся к непосредственному родителю.

Эти три случая проиллюстрированы в следующем примере использования написанной нами функции:

```
where("yyy")
#> Error: Не могу найти имя yyy
x <- 5
where("x")
#> <environment: R_GlobalEnv>
where("mean")
#> <environment: base>
```

Это может помочь взглянуть на картину в целом. Представьте, что у вас есть два окружения, как в показанном ниже коде и на рис. 7.7:

```
e4a <- env(empty_env(), a = 1, b = 2)
e4b <- env(e4a, x = 10, a = 11)
```

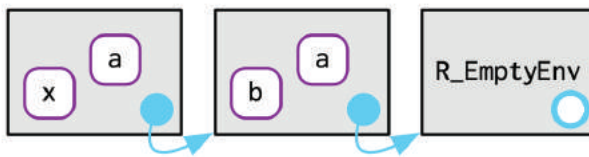


Рис. 7.7 Пара окружений

Применительно к этому раскладу:

- вызов функции `where("a", e4b)` найдет имя `a` в окружении `e4b`;
- вызов функции `where("b", e4b)` не найдет имя `b` в окружении `e4b`, обратится к непосредственному родителю – окружению `e4a` – и найдет его там;
- вызов функции `where("c", e4b)` попробует отыскать имя `c` в окружениях `e4b` и `e4a`, но не сможет этого сделать, дойдет до пустого окружения и ожидаемо вернет ошибку.

С окружениями часто работают с помощью рекурсий, поскольку это вполне естественно, так что наша простая функция `where()` может стать для вас отправной точкой при написании полноценных рекурсивных функций. Если избавить эту функцию от ее специфики, мы придем к шаблонному варианту, показанному ниже:

```
f <- function(..., env = caller_env()) {
    if (identical(env, empty_env())) {
        # Базовый случай
    } else if (success) {
        # Успех
    } else {
```

```
        # Рекурсивный случай
        f(..., env = env_parent(env))
    }
}
```

Итерации против рекурсии

Вы можете воспользоваться простым итерационным алгоритмом вместо рекурсии. Мне такая версия кажется более сложной для понимания, но я включил ее сюда, поскольку у вас может быть иное мнение, особенно если у вас не такой богатый опыт в написании рекурсий.

```
f2 <- function(..., env = caller_env()) {
    while (!identical(env, empty_env())) {
        if (success) {
            # Успех
            return()
        }
        # inspect parent
        env <- env_parent(env)
    }

    # Базовый случай
}
```

7.3.1 Упражнения

1. Измените функцию `where()` таким образом, чтобы она возвращала *все* окружения, в которых было найдено переданное имя. Подумайте над тем, объект какого типа в данном случае должна возвращать функция.
2. Напишите функцию с именем `fget()`, которая будет искать только объекты, являющиеся функциями. Она должна принимать на вход два аргумента – `name` и `env` – и следовать стандартным правилам поиска для функций: в случае нахождения объекта с искомым именем, не являющегося функцией, необходимо обратиться к родительскому окружению. В качестве задачки со звездочкой добавьте третий аргумент, `inherits`, отвечающий за то, будет ли поиск выполняться по всей иерархии наследования или только в ближайшем окружении.

7.4 Особые окружения

Большинство существующих окружений не создаются вами как разработчиком (с использованием функции `env()`), а присутствуют в R изначально. В этом разделе вы узнаете о наиболее важных окружениях, начиная с окружений пакетов. После этого вы познакомитесь с окружениями функций,

привязанными к ним в момент их создания, а также с недолговечными окружениями выполнения, создаваемыми каждый раз при вызове функции. Наконец, вы узнаете, как между собой взаимодействуют окружения пакетов и функций с целью поддержки пространств имен, что гарантирует одинаковое поведение пакетов вне зависимости от того, какие еще пакеты загрузил пользователь.

7.4.1 Окружения пакетов и путь для поиска

Каждый из пакетов, загруженных с помощью функции `library()` или `require()`, становится в ряд предков глобального окружения. Непосредственным родителем глобального окружения является последний *присоединенный* пакет[1], его родителем – предпоследний присоединенный пакет и т. д., что видно на рис. 7.8.

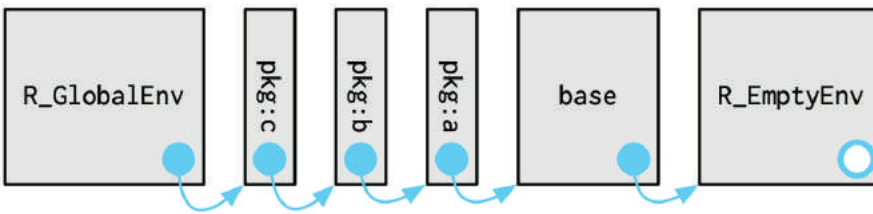


Рис. 7.8 Иерархия родителей окружений пакетов

Если пройти по всей ветке наследования окружений, можно восстановить полный порядок, в котором пакеты присоединялись. Этот порядок получил имя *путь поиска* (search path), поскольку все объекты в этих окружениях могут быть найдены из интерактивного рабочего пространства верхнего уровня. Имена окружений из пути поиска можно получить с помощью функции `base::search()`, а сами объекты окружений – с помощью `rlang::search_envs()`:

```
search()
#>  [1] ".GlobalEnv"        "package:rlang"     "package:stats"
#>  [4] "package:graphics"  "package:grDevices" "package:utils"
#>  [7] "package:datasets"  "package:methods"   "Autoloads"
#> [10] "package:base"

search_envs()
#>  [[1]] $ <env: global>
#>  [[2]] $ <env: package:rlang>
#>  [[3]] $ <env: package:stats>
#>  [[4]] $ <env: package:graphics>
```

[1] Обратите внимание на разницу между *присоединенным* (attached) пакетом и *загруженным* (loaded). Пакет загружается автоматически при обращении к любой его функции с помощью оператора `::`. При вызове функций `library()` или `require()` пакет просто *присоединяется* (attached) к пути поиска.

```
#>  [[5]] $ <env: package:grDevices>
#>  [[6]] $ <env: package:utils>
#>  [[7]] $ <env: package:datasets>
#>  [[8]] $ <env: package:methods>
#>  [[9]] $ <env: Autoloads>
#> [[10]] $ <env: package:base>
```

Последние два окружения в пути поиска всегда будут одинаковые:

- окружение `Autoloads` использует отложенное связывание для экономии памяти путем загрузки пакетов (таких как большие наборы данных) только при необходимости;
- окружение `base` представляет собой окружение базового пакета `package:base`, или просто `base`. Это особое окружение, поскольку с помощью него осуществляется инициализация загрузки остальных пакетов. К этому окружению можно обратиться напрямую посредством функции `base_env()`.

Обратите внимание, что при присоединении пакета с помощью функции `library()` родительское окружение глобального окружения меняется, что показано на рис. 7.9.

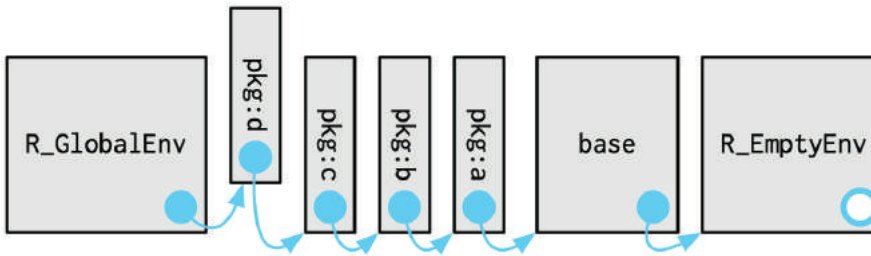


Рис. 7.9 Присоединение пакета pkg:d

7.4.2 Окружения функций

В момент создания функция осуществляет привязку к текущему окружению, которое в результате становится *окружением функции* (function environment) и используется для лексического поиска. В языках программирования функции, *захватывающие* (capture), или *замыкающие* (enclose), окружения, называются *замыканиями* (closure), поэтому этот термин часто используется в документации по R наравне с термином *функция*.

Окружение функции можно получить с помощью функции `fn_env()`:

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

Примечание. В базовом R вы можете воспользоваться функцией `environment(f)` для доступа к окружению функции `f`.

На рисунках я буду отображать функции как блоки с закругленной пристройкой, связанной с окружением, как показано на рис. 7.10.

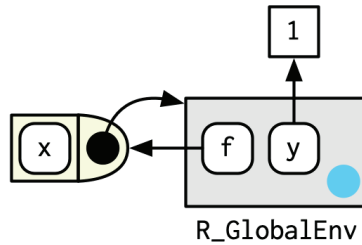


Рис. 7.10 Функция со связанным окружением

В данном случае `f()` привязана к окружению, в котором выполнена привязка имени `f` к нашей функции. Но так бывает не всегда. В следующем примере, показанном на рис. 7.11, привязка имени `g` происходит в новом созданном окружении `e`, тогда как `g()` привязывается к глобальному окружению. Кажущаяся незначительной разница между «привязкой в» и «привязкой к» на самом деле довольно существенна и заключается в том, каким образом мы находим функцию `g()` и каким образом `g()` находит используемые в ней переменные.

```
e <- env()
e$g <- function(x) 1
```

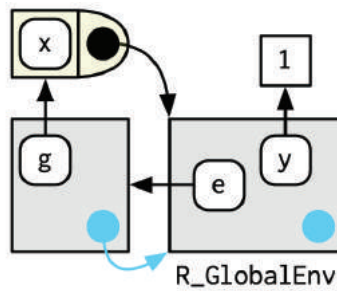


Рис. 7.11 Функция, объявленная в стороннем окружении

7.4.3 Пространства имен

Выше мы видели, что родительское окружение пакета может меняться в зависимости от того, какие еще пакеты загружены. Может показаться, что здесь есть повод для беспокойства: не означает ли это, что пакет может находить разные функции при загрузке пакетов в разном порядке? Основное назначение *пространств имен* (namespace) состоит в том, чтобы не допустить этого

и чтобы каждый пакет работал одинаково – вне зависимости от того, какие пакеты присоединены пользователем.

Возьмем, к примеру, функцию `sd()`, вычисляющую стандартное отклонение:

```
sd
#> function (x, na.rm = FALSE)
#> sqrt(var(if (is.vector(x) || is.factor(x)) x else as.double(x),
#> na.rm = na.rm))
#> <bytecode: 0x7fba712cc020>
#> <environment: namespace:stats>
```

В определении функции `sd()` используется обращение к функции `var()`, так что вас вполне может обеспокоить тот факт, что может быть вызвана не нужная вам функция `var()`, а одноименная функция, определенная либо в глобальном окружении, либо в каком-либо из присоединенных пакетов. Но в R эта проблема решена за счет установки приоритета окружения функции по сравнению с окружением привязки, как было показано выше. Каждая функция в пакете ассоциируется с парой окружений: окружением пакета, о котором мы уже узнали ранее, и *окружением пространства имен* (namespace environment):

- окружение пакета представляет собой внешний интерфейс для пакета. С помощью него вы как пользователь R находите функции в присоединенном пакете или посредством оператора `::`. Родитель этого окружения определяется с помощью пути поиска, т. е. с учетом порядка присоединения пакетов;
- окружение пространства имен является внутренним интерфейсом для пакета. Поиск функции определяется окружением пакета, а окружение пространства имен отвечает за то, как функция находит нужные ей переменные.

Каждая привязка в окружении пакета также присутствует в окружении пространства имен: это обеспечивает возможность поиска функций из других функций в пакете. Но некоторых привязок, присутствующих в окружении пространства имен, нет в окружении пакета. Такие привязки именуются *внутренними* (internal), или *неэкспортируемыми* (non-exported), объектами, и они позволяют спрятать внутренние детали реализации от пользователя.

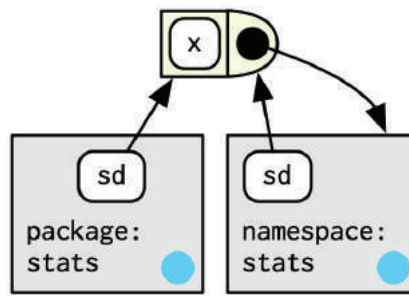


Рис. 7.12 Окружение пакета и окружение пространства имен

Все окружения пространств имен содержат одинаковый набор предков:

- у каждого пространства имен есть окружение `imports`, в котором находятся привязки ко всем функциям, используемым пакетом. Этим окружением управляет разработчик пакета посредством файла `NAMESPACE`;
- явно импортировать все базовые функции было бы слишком утомительно, в связи с чем родителем окружения `imports` является пространство имен `base`, что показано на рис. 7.13. Это пространство имен содержит все те же привязки, что и окружение `base`, но родитель у него другой;
- родителем пространства имен `base` является глобальное окружение. Это означает, что в случае если пакет не сможет найти привязку в окружении `imports`, он продолжит поиск обычным образом. Чаще всего это плохая идея, поскольку это делает код зависимым от загруженных пакетов, о чем вам сразу сообщит `R CMD check`. В основном это поддерживается из соображений совместимости, в частности из-за особенностей работы методов S3.

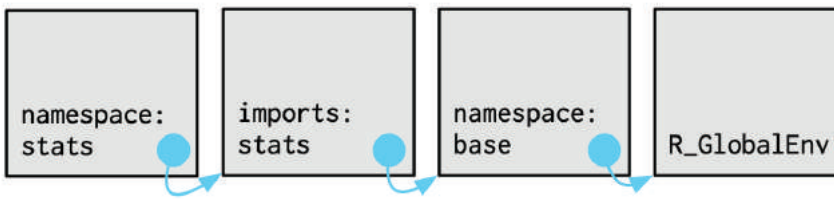


Рис. 7.13 Иерархия наследования окружения пространств

Если собрать все продемонстрированные рисунки воедино, получится картина, показанная на рис. 7.14.

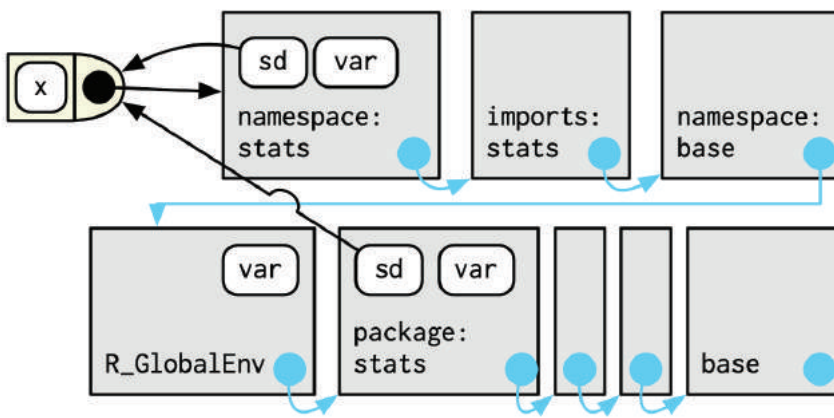


Рис. 7.14 Общая схема наследования

Таким образом, когда функция `sd()` ищет значение для имени `var`, она всегда находит его в последовательности окружений, определенной разработчиком пакета, а не пользователем. Этим обеспечивается стабильность работы пакетов вне зависимости от того, какие еще пакеты присоединит пользователь.

Между окружениями пакетов и окружениями пространств имен нет прямой связи. Эта связь определяется окружениями функций.

7.4.4 Окружения выполнения

Последняя важная тема, которую необходимо здесь обсудить, посвящена окружениям выполнения. Что приведенная ниже функция вернет после первого запуска? А после второго?

```
g <- function(x) {
    if (!env_has(current_env(), "a")) {
        message("Defining a")
        a <- 1
    } else {
        a <- a + 1
    }
    a
}
```

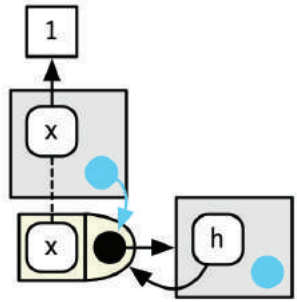
Подумайте немного, а затем продолжайте чтение.

```
g(10)
#> Defining a
#> [1] 1
g(10)
#> Defining a
#> [1] 1
```

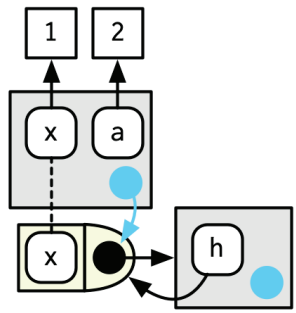
Функция при каждом запуске возвращает одно и то же значение, следуя принципу начала с чистого листа, о котором мы говорили в разделе 6.4.3. Каждый раз при вызове функции для ее выполнения создается новое окружение. Оно называется *окружением выполнения* (execution environment), а его родителем выступает окружение функции. Давайте проиллюстрируем этот процесс на примере простой функции. На рис. 7.15 графически показаны окружения выполнения, при этом родительские окружения в данном случае определяются посредством окружения функции.

```
h <- function(x) {
    # 1.
    a <- 2 # 2.
    x + a
}
y <- h(1) # 3.
```

1. Вызов функции с x = 1



2. Переменной a присваивается значение 2



3. Функция завершается с возвратом значения 3. Окружение выполнения удаляется

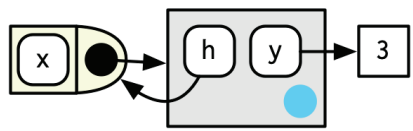


Рис. 7.15 Окружение выполнения для простой функции. Обратите внимание, что родителем окружения выполнения является окружение функции

Окружение выполнения обычно существует очень недолго: после окончания выполнения функции оно тут же уничтожается сборщиком мусора. Но есть несколько способов продлить эти мгновения. Первый заключается в явном возврате окружения из функции:

```
h2 <- function(x) {
    a <- x * 2
    current_env()
}

e <- h2(x = 10)
env_print(e)
#> <environment: 0x7fba76593d20>
```

```
#> parent: <environment: global>
#> bindings:
#>  * a: <dbl>
#>  * x: <dbl>
fn_env(h2)
#> <environment: R_GlobalEnv>
```

Второй способ состоит в возврате из функции объекта с привязкой к окружению вроде другой функции. Эта идея проиллюстрирована на втором примере с использованием фабрики функций, `plus()`. Мы воспользуемся этой фабрикой для создания функции с именем `plus_one()`.

На диаграмме, показанной на рис. 7.16, происходит много всего, поскольку замыкающее окружение функции `plus_one()` является окружением выполнения функции `plus()`.

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

plus_one <- plus(1)
plus_one
#> function(y) x + y
#> <environment: 0x7fba76650600>
```

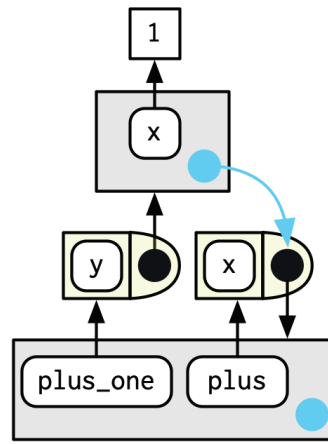


Рис. 7.16 Фабрика функций

Что происходит при вызове функции `plus_one()`? Ее окружение выполнения получит в качестве родителя захваченное окружение выполнения функции `plus()`, как видно на рис. 7.17:

```
plus_one(2)
#> [1] 3
```

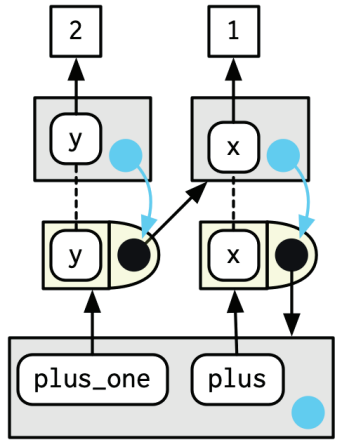


Рис. 7.17 Запуск фабрики функций

Подробнее о фабриках функций мы будем говорить в разделе 10.2.

7.4.5 Упражнения

1. Чем функция `search_envs()` отличается от `env_parents(global_env())`?
2. Нарисуйте диаграмму, показывающую замыкание окружений для следующих функций:

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

3. Напишите улучшенную версию функции `str()`, предоставляющую больше полезной информации о функциях. Покажите, где функция была найдена и в каком окружении была определена.

7.5 Стеки вызовов

Последнее особое окружение, о котором мы поговорим в этой главе, – это *вызывающее окружение* (caller environment), доступ к которому можно получить с помощью функции `rlang::caller_env()`. Речь идет об окружении, из

которого была вызвана функция, а значит, это окружение может меняться в зависимости от способа вызова функции, а не ее создания. Как мы видели выше, функцию `caller_env()` бывает удобно передавать в функцию в качестве значения по умолчанию, если она принимает на вход окружение.

Примечание. В базовом R аналогом функции `caller_env()` является `parent.frame()`. Обратите внимание, что она возвращает именно окружение, а не фрейм.

Для полного понимания того, что из себя представляет вызывающее окружение, нам необходимо обсудить две важные связанные концепции: *стек вызовов* (call stack) и *фреймы* (frame). Вызов функции приводит к созданию двух типов контекста. Об одном мы уже поговорили – это окружение выполнения, являющееся дочерним для окружения функции, которое определяется по тому, где была создана функция. Но есть еще один тип контекста, создаваемый во время вызова функции, и это стек вызовов.

7.5.1 Простые стеки вызовов

Давайте проиллюстрируем стек вызовов на простом примере последовательности вызовов функций: `f()` вызывает `g()`, которая вызывает `h()`:

```
f <- function(x) {
    g(x = 2)
}
g <- function(x) {
    h(x = 3)
}
h <- function(x) {
    stop()
}
```

Проще всего посмотреть на стек вызовов в R можно с помощью функции `traceback()`, которую можно вызвать после возникновения ошибки:

```
f(x = 1)
#> Error:
traceback()
#> 4: stop()
#> 3: h(x = 3)
#> 2: g(x = 2)
#> 1: f(x = 1)
```

Вместо последовательности `stop()` + `traceback()` для понимания работы стека вызовов мы воспользуемся функцией `lobstr::cst()` (от *call stack tree* – дерево стека вызовов):

```
h <- function(x) {
    lobstr::cst()
}
f(x = 1)
#> ▉
#> └─f(x = 1)
#>   └─g(x = 2)
#>     └─h(x = 3)
#>       └─lobstr::cst()
```

Как видно на выводе, функция `cst()` была вызвана из функции `h()`, которая была вызвана из `g()`, а та – из `f()`. Обратите внимание, что последовательность вывода функций здесь обратная по сравнению с функцией `traceback()`. В случае со сложными стеками вызовов легче бывает вникать в структуру вызовов, если функции перечисляются в прямом порядке, а не в обратном, т. е. `f()` вызывает `g()`, а не `g()` вызывается из `f()`.

7.5.2 Ленивые вычисления

Выше был показан очень простой стек вызовов: хотя намек на древовидную структуру здесь есть, все выполняется в единственной ветке. Это типичная ситуация для стека вызовов, когда все аргументы вычисляются *в жадной манере* (eagerly).

Давайте создадим более сложный пример, подразумевающий ленивые вычисления. Допустим, у нас есть последовательность функций `a()`, `b()` и `c()`, передающих друг другу аргумент `x`:

```
a <- function(x) b(x)
b <- function(x) c(x)
c <- function(x) x

a(f())
#> ▉
#> ├─a(f())
#> │ └─b(x)
#> │   └─c(x)
#> └─f()
#>   └─g(x = 2)
#>     └─h(x = 3)
#>       └─lobstr::cst()
```

Аргумент `x` в данном случае будет вычисляться в ленивой манере, так что в нашем дереве будет присутствовать две ветки. В первой ветке функция `a()` вызывает функцию `b()`, а функция `b()` – функцию `c()`. Вторая ветка инициируется в момент, когда функция `c()` вычисляет свой аргумент `x`. Этот аргумент будет вычисляться в отдельной ветке, поскольку он вычисляется в глобальном окружении, а не в окружении функции `c()`.

7.5.3 Фреймы

Каждый элемент стека вызовов представляет собой *фрейм* (frame)[1], также известный как контекст вычисления. Фрейм – это очень важная внутренняя структура данных, но из кода R можно получить доступ лишь к малой части этой структуры, поскольку обширное вмешательство может привести к отказу R. Фрейм состоит из трех компонентов:

- выражение (помеченное на рис. 7.18 как `expr`), инициирующее вызов функции. Это то, что выводит функция `traceback()`;
- окружение (помеченное на рис. 7.18 как `env`), обычно представленное окружением выполнения функции. Здесь могут быть два исключения: окружением глобального фрейма является глобальное окружение, а вызов функции `eval()` также создает фрейм, в котором окружение может быть любым;
- родительский элемент (предшествующий вызов в стеке вызовов, показанный на рис. 7.18 серыми стрелками).

На рис. 7.18 показан стек для вызова `f(x = 1)`, проиллюстрированного в разделе 7.5.1.

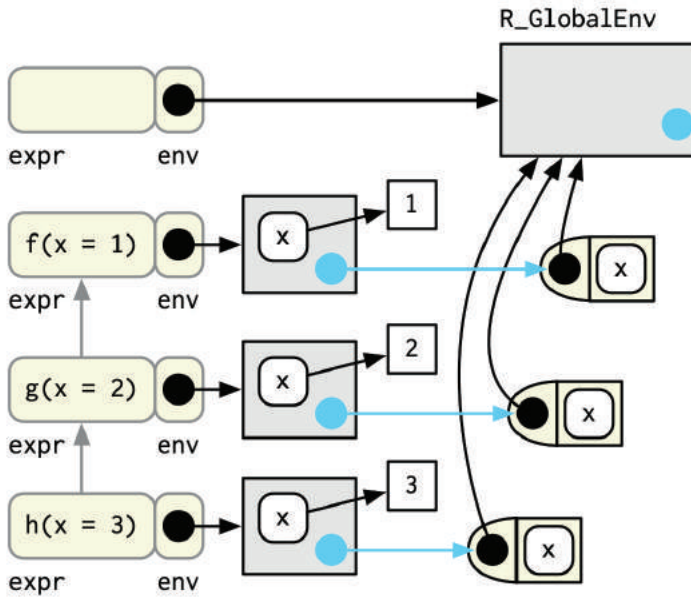


Рис. 7.18 Графическое отображение простого стека вызовов

[1] Важно: в документации `?environment` понятие фрейма используется несколько иначе: «Окружения состоят из фрейма, или коллекции именованных объектов, и указателя на замыкающее окружение». Мы не будем следовать этой терминологии относительно фреймов, принятой в S, из-за ее специфики и небольшой распространенности в R. К примеру, фрейм в `parent.frame()` является контекстом выполнения, а не коллекцией именованных объектов.

Чтобы сосредоточить внимание на вызывающих окружениях, я опустил привязки в глобальном окружении от `f`, `g` и `h` к соответствующим объектам функций.

Фрейм также содержит обработчики выхода, созданные с помощью функции `on.exit()`, рестарты (restart) и обработчики системы состояний, а еще контекст, в который функция должна вернуться после завершения работы. Это все важные детали внутреннего устройства, недоступные из кода R.

7.5.4 Динамический поиск

Поиск переменных в стеке вызовов вместо замыкающего окружения называется *динамическим поиском* (dynamic scoping). Лишь немногие языки программирования реализуют такую возможность (Emacs Lisp (http://www.gnu.org/software/emacs/emacs-paper.html#SEC15) – одно из исключений). Причина в том, что в условиях динамического поиска гораздо труднее бывает понять, как работает функция: вам недостаточно знать, как она была определена, необходимо также знать контекст, в котором она была вызвана. Динамический поиск может быть полезен при разработке функций, предназначенных для интерактивного анализа данных. Один из таких примеров будет рассмотрен в главе 20.

7.5.5 Упражнения

1. Напишите функцию, перечисляющую все переменные, определенные в окружении, в котором функция была вызвана. Она должна возвращать такой же результат, как функция `ls()`.

7.6 Окружения как структуры данных

Помимо помощи в поиске значений имен, окружения могут быть полезны и сами по себе – в виде структур данных, поскольку обладают ссылочной семантикой. Ниже перечислены три основных типа проблем, с которыми помогают справиться окружения:

- **копирование больших данных**. Поскольку окружения обладают ссылочной семантикой, вы никогда не столкнетесь с проблемой создания копий больших объемов данных. При этом с окружениями в чистом виде работать может быть некомфортно, поэтому я рекомендую использовать объекты R6, построенные на базе окружений. Подробнее об этом мы будем говорить в главе 14;
- **управление состояниями внутри пакетов**. Использование окружений в явном виде при работе с пакетами может оказаться очень удобным, поскольку они помогают поддерживать состояния между вызовами функций. Обычно объекты в пакете заблокированы, так что вы не

можете модифицировать их напрямую. Вместо этого вы можете сделать следующее:

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

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

Возврат прежнего значения элемента из функции, устанавливающей новое значение, – это хорошая практика, поскольку такой подход позволяет восстановить прежнее значение с помощью функции `on.exit()`, о которой мы говорили в разделе 6.7.4;

- **использование в виде хешмапа**. Хешмап представляет собой структуру данных с постоянной сложностью поиска элемента по ключу, равной O(1). Окружения по умолчанию реализуют такое поведение, так что легко могут быть использованы как замена хешмапа. Взгляните на пакет `hash` [Браун (Brown), 2013] для ознакомления с этой идеей.

7.7 Ответы на контрольные вопросы

1. Существует четыре отличия окружений от списков:
   a) каждый объект в окружении должен обладать именем;
   b) порядок расположения объектов в окружении не имеет значения;
   c) у окружений есть родители;
   d) окружения обладают ссылочной семантикой.
2. Родителем глобального окружения является последний загруженный пакет. Единственное окружение, у которого нет родителя, – это пустое окружение.
3. Замыкающим окружением функции является окружение, в котором она была создана. Оно определяет область поиска функцией переменных.
4. Воспользоваться функцией `caller_env()` или `parent.frame()`.
5. Оператор `<-` всегда создает привязку в текущем окружении, а оператор `<<-` модифицирует значение существующего элемента в родительском окружении.

8

Состояния

8.1 Введение

Система состояний (condition system) представляет собой парный набор инструментов, позволяющий автору функции сообщить о каком-то ее необычном поведении, а пользователю этой функции – правильно обработать это поведение. Автор функции сигнализирует о возникновении определенного состояния с помощью таких функций, как `stop()` (для ошибок), `warning()` (для предупреждений) и `message()` (для сообщений), после чего пользователь функции может перехватить это состояние посредством функций вроде `tryCatch()` и `withCallingHandlers()`. Досконально понимать работу системы состояний крайне важно, поскольку зачастую вам придется выполнять обе роли сразу: сигнализировать о возникновении состояний в ваших функциях и перехватывать сигналы в вызываемых функциях.

R располагает чрезвычайно мощной системой состояний, базирующейся на идеях из диалекта Common Lisp. Как и в случае с подходом языка R к объектно ориентированному программированию, система состояний имеет существенные отличия от аналогичных систем, принятых в других языках, так что в ней может быть непросто разобраться, тем более в отсутствие большого количества инструкций по ее эффективному использованию. Таким образом, исторически так сложилось, что лишь немногие разработчики (я из их числа) использовали эту систему на полную мощность. Цель этой главы – исправить эту плачевную ситуацию. Здесь я вам подробно объясню, как устроена система состояний в R, и покажу несколько практических инструментов, которые помогут сделать ваш код более надежным.

При написании данной главы я пользовался в основном двумя следующими источниками. Вы тоже можете с ними ознакомиться, если хотите узнать о предпосылках и истории зарождения системы состояний в R:

- *A Prototype of a Condition System for R* (http://homepage.stat.uiowa.edu/~luke/R/exceptions/simpcond.html) от Роберта Джентльмена (Robert Gentleman) и Люка Тьерни (Luke Tierney). В этой статье описывается ранняя версия системы состояний в R. Хотя с даты публикации статьи многое в этом отношении изменилось, она по-прежнему может быть вам полезна, если вы хотите погрузиться в мысли о том, как все устроено внутри;

- Beyond exception handling: conditions and restarts (https://gigamonkeys.com/book/beyond-exception-handling-conditions-and-restarts.html) от Питера Сибеля (Peter Seibel). В этом материале описывается функционал обработчика исключительных ситуаций в языке Lisp, который принципиально очень близок к подходу, применяемому в R. Здесь вы также найдете множество полезных примеров. Я адаптировал эту главу для языка R, и с моей адаптацией можно ознакомиться по адресу http://adv-r.had.co.nz/beyond-exception-handling.html.

Кроме того, оказалось полезно исследовать лежащий в основе системы состояний код на языке C. Если вам интересно, как это все работает, вы можете ознакомиться с моими заметками по ссылке https://gist.github.com/hadley/4278d0a6d3a10e42533d59905fbed0ac.

Контрольные вопросы

Хотите пропустить эту главу? Пожалуйста, но только если сможете правильно ответить на приведенные ниже вопросы. Ответы вы найдете в конце главы, в разделе 8.7.

1. Какие существуют три основных типа состояний?
2. Какой функцией можно воспользоваться, чтобы проигнорировать ошибки во фрагменте кода?
3. Какое основное отличие между функциями `tryCatch()` и `withCallingHandlers()`?
4. Почему вам может понадобиться создать пользовательский объект ошибки?

Структура главы

- В разделе 8.2 мы представим основные инструменты сигнализирования о возникших состояниях и обсудим каждый тип состояния отдельно.
- Раздел 8.3 будет посвящен простейшему механизму обработки состояний с помощью функций `try()` и `supressMessages()`, которые перехватывают состояния и не дают им добраться до верхнего уровня.
- В разделе 8.4 мы познакомимся с объектами состояний и двумя основными инструментами обработки состояний: функцией `tryCatch()` для ошибок и функцией `withCallingHandlers()` для всего остального.
- В разделе 8.5 вы узнаете, как можно расширить встроенные объекты состояний за счет новых полезных данных, которые могут быть использованы обработчиками состояний для принятия более взвешенных решений.
- В разделе 8.6 мы подведем итоги главы с помощью нескольких практических примеров, основывающихся на низкоуровневых инструментах, описанных в предыдущих разделах.

8.1.1 Требования

Наравне с базовыми функциями R мы будем использовать в этой главе функции сигнализирования и обработки состояний из пакета rlang (https://rlang.r-lib.org).

```
library(rlang)
```

8.2 Сигнализирование о состояниях

Существует три типа *состояний* (condition), о возникновении которых вы можете просигнализировать в коде: *ошибки* (errors), *предупреждения* (warnings) и *сообщения* (messages):

- ошибки – это наиболее серьезный тип состояния. Их возникновение говорит о том, что функция не может продолжить свое выполнение и вынуждена прерваться;
- предупреждения – это нечто среднее между ошибками и сообщениями. Обычно с помощью них разработчик уведомляет о том, что что-то пошло не так, но функция завершилась по крайней мере частично;
- к разряду сообщений относятся самые безобидные состояния. С помощью них можно проинформировать пользователя о выполнении каких-либо действий по их просьбе.

Есть еще одно состояние, которое может быть сгенерировано только интерактивно. Речь идет о *прерывании* (interrupt), показывающем, что пользователь сам прервал выполнение программы, нажав на **Escape**, **Ctrl**+**Break** или **Ctrl**+**C**, в зависимости от платформы.

Информация о возникающих состояниях обычно отображается очень заметно, с помощью жирного шрифта или выделения красным цветом – зависит от используемого интерфейса R. Различать состояния очень легко: оповещения об ошибках всегда предваряются словом `Error`, предупреждения – словом `Warning` или фразой `Warning message`, а сообщения выводятся как есть.

```
stop("Так выглядит ошибка")
#> Error in eval(expr, envir, enclos): Так выглядит ошибка

warning("Так выглядит предупреждение")
#> Warning: Так выглядит предупреждение

message("Так выглядит сообщение")
#> Так выглядит сообщение
```

В следующих трех разделах мы рассмотрим каждый тип состояния отдельно.

8.2.1 Ошибки

В базовом R *ошибки* (error) инициируются, или *выбрасываются* (thrown), с помощью функции `stop()`:

```
f <- function() g()
g <- function() h()
h <- function() stop("Это ошибка!")

f()
#> Error in h(): Это ошибка!
```

По умолчанию сообщение об ошибке содержит предшествующий ей вызов, хотя мне эта информация не кажется полезной (к тому же она полностью дублирует информацию, которую можно получить из функции `traceback()`), так что лично я советую передавать аргумент `call. = FALSE`[1], как показано ниже:

```
h <- function() stop("Это ошибка!", call. = FALSE)
f()
#> Error: Это ошибка
```

Эквивалент функции `stop()` в пакете `rlang` – функция `rlang::abort()` – делает это автоматически. На протяжении данной главы мы будем повсеместно использовать функцию `abort()`, но до ее удивительной особенности, заключающейся в возможности добавлять вспомогательные метаданные к объекту состояния, мы доберемся только к концу главы.

```
h <- function() abort("Это ошибка!")
f()
#> Error: Это ошибка
```

Примечание. Функция `stop()` склеивает вместе множественные входные данные, тогда как функция `abort()` этого не делает. При необходимости вывести сложное сообщение об ошибке я рекомендую пользоваться функцией `glue::glue()`. Это позволит нам присоединить другие полезные аргументы функции `abort()`, о которых вы узнаете из раздела 8.5.

В идеале сообщения об ошибках должны содержать не только информацию о том, что пошло не так, но и инструкцию, позволяющую решить возникшую проблему. Писать максимально понятные сообщения об ошибках непросто, поскольку обычно они возникают, когда пользователь не до конца понимает назначение функции. Как разработчику вам может быть невдомек, как вообще можно неправильно понять назначение вполне очевидной функции. Отсюда и сложности в написании кратких и понятных сообщений об

[1] Завершающая точка в названии аргумента `call.` – это просто такая странность функции `stop()`, никаких других объяснений этому даже не ищите.

ошибках. Несмотря на все это, в руководстве по tidyverse удалось собрать все наиболее важные моменты, связанные с этим аспектом, с которыми можно ознакомиться по адресу https://style.tidyverse.org/error-messages.html.

8.2.2 Предупреждения

Предупреждения (warning), сигналы о которых посылаются с помощью функции `warning()`, обладают менее серьезным уровнем критичности по сравнению с ошибками. Как правило, с помощью них пользователю сообщают, что что-то пошло не так, но код смог обработать эту ситуацию и продолжился. В отличие от ошибок, один вызов функции может генерировать сразу несколько предупреждений:

```
fw <- function() {
    cat("1\n")
    warning("W1")
    cat("2\n")
    warning("W2")
    cat("3\n")
    warning("W3")
}
```

По умолчанию сообщения о предупреждениях кешируются и выводятся на экран только по возвращении контроля над управлением на верхний уровень:

```
fw()
#> 1
#> 2
#> 3
#> Warning messages:
#> 1: In f() : W1
#> 2: In f() : W2
#> 3: In f() : W3
```

Такое поведение может быть скорректировано с помощью опции `warn`:

- чтобы сообщения о предупреждениях выводились на экран немедленно, воспользуйтесь командой `options(warn = 1)`;
- чтобы превратить предупреждения в ошибки, введите команду `options(warn = 2)`. Это простейший способ отладки предупреждений, поскольку в этом случае вы можете использовать такие инструменты, как `traceback()`;
- поведение по умолчанию восстанавливается при запуске команды `options(warn = 0)`.

Как и у функции `stop()`, у функции `warning()` есть аргумент `call`. В данном случае дополнительный вывод имеет чуть больше смысла, поскольку предупреждения зачастую располагаются довольно далеко от источника, но

я все равно рекомендую отключать эту опцию с помощью передачи `call. = FALSE`. Как и в случае с функцией `rlang::abort()`, в пакете `rlang` присутствует аналог функции `warning()`, именуемый `rlang::warn()`, также подавляющий аргумент `call.` по умолчанию.

Предупреждения располагаются где-то посередине между сообщениями («вам следует это знать») и ошибками («вам нужно это исправить!»), и мне сложно дать четкий совет, когда именно их нужно использовать. В целом не стоит злоупотреблять предупреждениями, поскольку их бывает легко пропустить, особенно если приложение подразумевает много другого вывода, а вам бы не хотелось, чтобы функция спокойно продолжала работать при неправильных входных данных. Лично мне кажется, что в базовом R предупреждения используются уж слишком часто, многие из них было бы неплохо заменить на ошибки. К примеру, показанные ниже предупреждения для меня являются первыми кандидатами на перевод в разряд ошибок:

```
formals(1)
#> Warning in formals(fun): argument is not a function
#> NULL

file.remove("this-file-doesn't-exist")
#> Warning in file.remove("this-file-doesn't-exist"): cannot remove file
#> 'this-file-doesn't-exist', reason 'No such file or directory'
#> [1] FALSE
lag(1:3, k = 1.5)
#> Warning in lag.default(1:3, k = 1.5): 'k' is not an integer
#> [1] 1 2 3
#> attr(,"tsp")
#> [1] -1 1 1
```

Я знаю два случая, когда предупреждения могут быть уместны:

- при обновлении функции до новой версии вы хотите, чтобы старый код продолжал нормально работать (таким образом, пользователь может проигнорировать это предупреждение), но при этом было бы неплохо, чтобы переход на новую версию состоялся как можно раньше;
- когда вы почти наверняка знаете, что возникшие неожиданности не помешают вашему коду успешно завершиться. При уверенности в этом на 100 % можно обойтись без всяких сообщений, а при отсутствии уверенности самое время задуматься о выбросе ошибки.

В остальных случаях старайтесь не использовать предупреждения – спокойно меняйте их на ошибки.

8.2.3 Сообщения

Сообщения (message), отправляемые функцией `message()`, являются чисто информационными. Используйте их для того, чтобы сообщить пользователю о действиях, выполненных по их просьбе. В идеале сообщения должны быть

очень хорошо сбалансированы по объему и наполнению: предоставьте пользователю необходимое и достаточное количество информации, но не более того, чтобы не докучать ему.

Функция `message()` срабатывает незамедлительно и не имеет аргумента `call.`:

```
fm <- function() {
    cat("1\n")
    message("M1")
    cat("2\n")
    message("M2")
    cat("3\n")
    message("M3")
}

fm()
#> 1
#> M1
#> 2
#> M2
#> 3
#> M3
```

Подходящие случаи для использования сообщений:

- когда значение аргумента по умолчанию требует каких-то вычислений и вы хотите сообщить пользователю о том, какое значение используется. Например, пакет `ggplot2` сообщает пользователю о количестве используемых столбиков на диаграмме, если он явно не передал значение для аргумента `binwidth`;
- в функциях, вызываемых ради побочных эффектов, которые в противном случае могли быть безмолвными. К примеру, при записи файла на диск, вызове web API или сохранении данных в базе бывает полезно периодически сообщать пользователю о происходящем;
- перед запуском длительного процесса, не подразумевающего промежуточных уведомлений. Конечно, лучше будет показывать индикатор процесса (https://github.com/r-lib/progress), но начать можно и с простых сообщений;
- при написании пакетов вам может понадобиться отображать сообщение при загрузке пакета (т. е. в методе `.onAttach()`). В этом случае вы можете воспользоваться функцией `packageStartupMessage()`.

В общем случае каждая функция, предполагающая вывод сообщений, должна предполагать вызов в тихом режиме, например при помощи аргумента `quiet = TRUE`. Как вы узнаете совсем скоро, можно подавить вывод всех сообщений с помощью функции `suppressMessages()`, но лучше предусмотреть такую возможность и для самой функции.

Очень важно сравнить близкие по смыслу функции `message()` и `cat()`. В плане использования и вывода результата эти функции действительно очень похожи[1]:

```
cat("Hi!\n")
#> Hi!
message("Hi!")
#> Hi!
```

На самом же деле назначение этих функций сильно отличается. Функцию `cat()` следует использовать при необходимости вывести что-то на консоль, подобно функции `print()` или `str()`. Функцию `message()` можно применять как вспомогательное средство для вывода информации на консоль, когда основное назначение функции состоит совсем в другом. Иными словами, функцию `cat()` можно использовать, когда пользователь просит что-то напечатать, а функцию `message()` – когда разработчик решает что-то вывести на экран.

8.2.4 Упражнения

1. Напишите обертку для функции `file.remove()`, которая будет выбрасывать ошибку в случае, если файл, предназначенный для удаления, отсутствует.
2. Для чего предназначен аргумент `appendLF` в функции `message()`? Как он соотносится с функцией `cat()`?

8.3 Игнорирование состояний

Простейшим способом обработки состояний в R является их игнорирование:

- игнорирование ошибок с помощью функции `try()`;
- игнорирование предупреждений с помощью функции `suppressWarnings()`;
- игнорирование сообщений с помощью функции `suppressMessages()`.

Эти функции не слишком удобны в использовании по причине того, что вы не можете использовать их для подавления одного отдельного типа состояния, о котором вам известно, пропуская все остальные. Позже в данной главе мы вернемся к этой проблеме.

Функция `try()` позволяет продолжить выполнение программы даже после возникновения ошибки. Функции, содержащие ошибки, обычно завершают свою работу немедленно после их возникновения, не возвращая при этом никаких значений:

---

[1] Обратите внимание, что функция `cat()` требует явно постановки завершающего символа «\n» для переноса строки.

```
f1 <- function(x) {
    log(x)
    10
}
f1("x")
#> Error in log(x): non-numeric argument to mathematical function
```

Однако если заключить блок кода с ошибкой в функцию `try()`, сообщение об ошибке появится[1], но выполнение программы продолжится:

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

Можно, хотя и не рекомендуется, сохранить результат выполнения функции `try()` и продолжать выполнение кода в зависимости от того, возникла ошибка или нет[2]. Вместо этого лучше воспользоваться функцией `tryCatch()` или высокоуровневыми вспомогательными средствами, о которых вы узнаете совсем скоро.

Простым, но полезным шаблоном является присваивание внутри вызова: оно позволяет определить значение по умолчанию, которое будет использовано, если выполнение кода потерпит неудачу. Это работает по причине того, что аргумент вычисляется в вызывающем окружении, а не внутри функции (см. раздел 6.5.1).

```
default <- NULL
try(default <- read.csv("possibly-bad-input.csv"), silent = TRUE)
```

Функции `suppressWarnings()` и `suppressMessages()` позволяют подавить вывод всех предупреждений и сообщений. В отличие от ошибок, предупреждения и сообщения не прерывают выполнение кода, так что в одном блоке кода их может присутствовать сразу несколько.

```
suppressWarnings({
    warning("Uhoh!")
    warning("Another warning")
    1
})
#> [1] 1
```

---

[1] Вы можете подавить вывод сообщения с помощью параметра `silent`: `try(..., silent = TRUE)`.

[2] В случае возникновения ошибки результат будет принадлежать классу `try-error`.

```
suppressMessages({
    message("Hello there")
    2
})
#> [1] 2

suppressWarnings({
    message("You can still see me")
    3
})
#> You can still see me
#> [1] 3
```

8.4 Обработка состояний

У каждого типа состояния есть свое поведение по умолчанию: ошибки прерывают выполнение программы и возвращаются на верхний уровень, предупреждения захватываются и выводятся все вместе, а сообщения отображаются немедленно. *Обработчики состояний* (condition handler) позволяют временно переопределить или дополнить их поведение.

Функции `tryCatch()` и `withCallingHandlers()` служат для регистрации обработчиков, т. е. функций, принимающих возникшие состояния в качестве единственного аргумента. Эти функции-обработчики обладают одинаковой базовой формой:

```
tryCatch(
    error = function(cnd) {
        # Код, выполняемый при возникновении ошибки
    },
    code_to_run_while_handlers_are_active
)

withCallingHandlers(
    warning = function(cnd) {
        # Код, выполняемый при появлении предупреждения
    },
    message = function(cnd) {
        # Код, выполняемый при появлении сообщения
    },
    code_to_run_while_handlers_are_active
)
```

Отличаются они типом создаваемых обработчиков:

- функция `tryCatch()` определяет *обработчики выхода* (exiting handler); после обработки состояния управление возвращается в контекст, в котором была вызвана функция `tryCatch()`. Это делает данную функцию

наиболее подходящей для использования с ошибками и прерываниями, поскольку они все равно завершают выполнение кода;

- функция `withCallingHandlers()` определяет *обработчики вызова* (calling handler); после захвата состояния управление возвращается в контекст, в котором возникло состояние. Это хорошо подходит для работы с состояниями, не связанными с ошибками.

Но перед тем как научиться взаимодействовать с этими обработчиками, нам необходимо немного поговорить об объектах состояний. Эти объекты создаются неявно всякий раз при появлении сигнала о возникновении состояния, но внутри обработчика они присутствуют в явном виде.

8.4.1 Объекты состояний

До сих пор мы только сигнализировали о возникновении состояний, но не обращали внимания на объекты, создаваемые за кулисами. Проще всего можно увидеть *объект состояния* (condition object), если перехватить его после возникновения соответствующего сигнала. Это делается с помощью функции `rlang::catch_cnd()`:

```
cnd <- catch_cnd(stop("An error"))
str(cnd)
#> List of 2
#> $ message: chr "An error"
#> $ call : language force(expr)
#> - attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

Встроенные состояния представляют собой списки, состоящие из двух элементов:

- `message` – символьный вектор, содержащий текст для отображения пользователю. Для извлечения этого элемента можно воспользоваться функцией `conditionMessage()`;
- `call` – вызов, осуществленный посредством состояния. Как было сказано выше, мы не используем этот вызов, так что часто значение данного элемента будет равно `NULL`. Для извлечения этого элемента служит функция `conditionCall(cnd)`.

Пользовательские состояния могут содержать другие элементы, о чем мы поговорим в разделе 8.5.

У состояний также есть атрибут `class`, который делает их объектами S3. Об объектной системе S3 мы будем говорить только в главе 13, но, к счастью, для работы с состояниями вам не нужно быть глубоко в теме, к тому же эти объекты являются достаточно простыми. Важно лишь понимать, что атрибут `class` представляет собой символьный вектор, определяющий, какие обработчики соответствуют данному состоянию.

8.4.2 Обработчики выхода

Функция `tryCatch()` регистрирует *обработчики выхода* (exiting handler), и обычно она используется для обработки состояний, представляющих ошибки. Это позволяет вам переопределить поведение ошибок по умолчанию. К примеру, следующий код вернет значение `NA` вместо выброса ошибки:

```
f3 <- function(x) {
    tryCatch(
        error = function(cnd) NA,
        log(x)
    )
}

f3("x")
#> [1] NA
```

В случае отсутствия сигнала о возникновении состояния или несоответствия его класса имени обработчика код будет выполняться как обычно:

```
tryCatch(
    error = function(cnd) 10,
    1 + 1
)
#> [1] 2

tryCatch(
    error = function(cnd) 10,
    {
        message("Hi!")
        1 + 1
    }
)
#> Hi!
#> [1] 2
```

Обработчики, созданные при помощи функции `tryCatch()`, называются обработчиками выхода, поскольку после сигнала о возникновении состояния управление передается обработчику и больше не возвращается в исходный код, что означает завершение выполнения программы:

```
tryCatch(
    message = function(cnd) "There",
    {
        message("Here")
        stop("This code is never run!")
    }
)
#> [1] "There"
```

Защищенный код выполняется в окружении функции `tryCatch()`, а код обработчика – нет, поскольку обработчик – это функция. Об этом важно помнить при попытке модификации объектов в родительском окружении.

Функции-обработчики вызываются с единственным аргументом, представляющим собой объект состояния. Следуя соглашениям, я именую этот аргумент `cnd`. Значение данного аргумента трудно назвать очень полезным применительно к базовым состояниям, поскольку оно содержит крайне мало информации. Больше пользы от этого аргумента может быть при создании собственных объектов состояний, как вы узнаете совсем скоро.

```
tryCatch(
    error = function(cnd) {
        paste0("--", conditionMessage(cnd), "--")
    },
    stop("This is an error")
)
#> [1] "--This is an error--"
```

У функции `tryCatch()` есть еще один аргумент с именем `finally`. С помощью него можно определить блок кода (не функцию), который будет запускаться вне зависимости от того, завершится исходное выражение успешно или с ошибкой. Этот блок кода может быть полезен для выполнения очистки, например для удаления файлов или закрытия подключений. Функционально блок `finally` эквивалентен использованию функции `on.exit()` (по сути, он реализован так же точно), но он может включать в себя меньший объем кода по сравнению с полноценной функцией.

```
path <- tempfile()
tryCatch(
    {
        writeLines("Hi!", path)
        # ...
    },
    finally = {
        # запускается всегда
        unlink(path)
    }
)
```

8.4.3 Обработчики вызова

Обработчики, созданные функцией `tryCatch()`, называются обработчиками выхода, поскольку они приводят к завершению кода после перехвата состояния. Напротив, функция `withCallingHandlers()` создает так называемые *обработчики вызова* (calling handler), не нарушающие ход выполнения программы после возврата из обработчика. Это делает функцию `withCallingHandlers()` пригодной для использования совместно с состояниями, не свя-

занными с ошибками. По существу, в обработчиках выхода и обработчиках вызова слово обработчик используется по-разному:

- обработчик выхода воспринимает сигнал о состоянии как возникновение серьезной проблемы, с которой нужно справиться;
- обработчик вызова воспринимает сигнал о состоянии как нечто рядовое, после чего жизнь продолжается в привычном ритме.

Сравните результаты работы функций `tryCatch()` и `withCallingHandlers()` в примере ниже. В первом случае сообщения не вывелись, поскольку код завершил свою работу по окончании запуска обработчика выхода. Во втором случае сообщения вывелись, потому что обработчик вызова не инициирует завершение работы.

```
tryCatch(
    message = function(cnd) cat("Перехватили сообщение!\n"),
    {
        message("Есть кто?")
        message("Да, а что?")
    }
)
#> Перехватили сообщение!

withCallingHandlers(
    message = function(c) cat("Перехватили сообщение!\n"),
    {
        message("Есть кто?")
        message("Да, а что?")
    }
)
#> Перехватили сообщение!
#> Есть кто?
#> Перехватили сообщение!
#> Да, а что?
```

Обработчики запускаются в порядке их объявления, так что вам не стоит беспокоиться о том, что вы застрянете в бесконечном цикле. В следующем примере состояние `message()`, возникающее в процессе работы самого обработчика, не перехватывается:

```
withCallingHandlers(
    message = function(cnd) message("Второе сообщение"),
    message("Первое сообщение")
)
#> Второе сообщение
#> Первое сообщение
```

Но будьте осторожны при объявлении нескольких обработчиков, некоторые из которых генерируют состояния, которые могут быть перехвачены

другими обработчиками. В этом случае вам нужно хорошо продумать последовательность обработчиков.

Возвращаемое значение из обработчика вызова игнорируется, поскольку код по завершении обработчика продолжает выполняться и это значение просто некуда девать. Это означает, что обработчики вызова могут быть полезны только благодаря своим побочным эффектам.

Одним из полезных побочных эффектов обработчиков вызова является возможность предотвращать дальнейшее распространение сигналов. По умолчанию состояние поднимается по иерархии к родительским обработчикам вплоть до обработчика по умолчанию (или обработчика выхода, если он есть):

```
# Сигнал поднимается вплоть до обработчика по умолчанию, который выводит сообщение
withCallingHandlers(
    message = function(cnd) cat("Level 2\n"),
    withCallingHandlers(
        message = function(cnd) cat("Level 1\n"),
        message("Hello")
    )
)
#> Level 1
#> Level 2
#> Hello

# Сигнал поднимается в tryCatch
tryCatch(
    message = function(cnd) cat("Level 2\n"),
    withCallingHandlers(
        message = function(cnd) cat("Level 1\n"),
        message("Hello")
    )
)
#> Level 1
#> Level 2
```

Если вам необходимо предотвратить распространение сигнала, но вы хотите, чтобы дальнейший код в блоке выполнялся, вы можете заблокировать состояние при помощи функции `rlang::cnd_muffle()`, как показано ниже:

```
# Блокируем обработчик по умолчанию, который выводит сообщение
withCallingHandlers(
    message = function(cnd) {
        cat("Level 2\n")
        cnd_muffle(cnd)
    },
    withCallingHandlers(
        message = function(cnd) cat("Level 1\n"),
        message("Hello")
    )
```

```
)
#> Level 1
#> Level 2

# Блокируем обработчик уровня 2 и обработчик по умолчанию
withCallingHandlers(
    message = function(cnd) cat("Level 2\n"),
    withCallingHandlers(
        message = function(cnd) {
            cat("Level 1\n")
            cnd_muffle(cnd)
        },
        message("Hello")
    )
)
#> Level 1
```

8.4.4 Стеки вызовов

В завершение раздела приведем некоторые отличия между *стеками вызовов* (call stack) обработчиков выхода и обработчиков вызова. На самом деле эти отличия не так важны, но я включил их в этот раздел, поскольку нахожу их весьма полезными и не хотел бы о них забыть!

Легче всего обнаружить эти отличия на простом примере с использованием функции `lobstr::cst()`:

```
f <- function() g()
g <- function() h()
h <- function() message("!")
```

Обработчики вызова запускаются в контексте вызова, сигнализирующего о возникновении состояния:

```
withCallingHandlers(f(), message = function(cnd) {
    lobstr::cst()
    cnd_muffle(cnd)
})
#>      ▉
#>    1. ├─base::withCallingHandlers(...)
#>    2. ├─global::f()
#>    3. │ └─global::g()
#>    4. │   └─global::h()
#>    5. │     └─base::message("!")
#>    6. │       ├─base::withRestarts(...)
#>    7. │       │ └─base:::withOneRestart(expr, restarts[[1L]])
#>    8. │       │   └─base:::doWithOneRestart(return(expr), restart)
#>    9. │       └─base::signalCondition(cond)
#>   10. └─(function (cnd) ...
#>   11.   └─lobstr::cst()
```

В то же время обработчики выхода вызываются в контексте обращения к функции `tryCatch()`:

```
tryCatch(f(), message = function(cnd) lobstr::cst())
#>      ▉
#>   1. └base::tryCatch(f(), message = function(cnd) lobstr::cst())
#>   2.  └base:::tryCatchList(expr, classes, parentenv, handlers)
#>   3.   └base:::tryCatchOne(expr, names, parentenv, handlers[[1L]])
#>   4.    └value[[3L]](cond)
#>   5.     └lobstr::cst()
```

8.4.5 Упражнения

1. Какую дополнительную информацию содержит состояние, сгенерированное функцией `abort()`, в сравнении с функцией `stop()`? Иными словами, какая разница между этими двумя объектами состояний? Прочитайте инструкцию с помощью команды `?abort`, чтобы узнать больше.

```
catch_cnd(stop("An error"))
catch_cnd(abort("An error"))
```

2. Предугадайте результат выполнения приведенного ниже кода:

```
show_condition <- function(code) {
    tryCatch(
        error = function(cnd) "error",
        warning = function(cnd) "warning",
        message = function(cnd) "message",
        {
            code
            NULL
        }
    )
}

show_condition(stop("!"))
show_condition(10)
show_condition(warning("?!"))
show_condition({
    10
    message("?")
    warning("?!")
})
```

3. Поясните результаты запуска представленного ниже фрагмента кода:

```
withCallingHandlers(
    message = function(cnd) message("b"),
    withCallingHandlers(
```

```
        message = function(cnd) message("a"),
        message("c")
    )
)
#> b
#> a
#> b
#> c
```

4. Ознакомьтесь с исходным кодом функции `catch_cnd()` и опишите ее работу.
5. Как можно переписать функцию show_condition(), чтобы она использовала один обработчик?

8.5 Пользовательские состояния

Одной из сложностей, связанных с обработкой состояний в языке R, является то, что большинство функций генерируют одно из встроенных состояний, которые содержат только атрибуты `message` и `call`. Таким образом, для обнаружения каких-то специфических типов ошибок вы можете использовать только передаваемое сообщение об ошибке. Подобный подход может приводить к нежелательному поведению не только из-за того, что сообщения могут меняться с течением времени, но и по причине перевода сообщений на другие языки.

К счастью, в R предусмотрена очень мощная, хотя и редко используемая, возможность создавать *пользовательские состояния* (custom condition), которые могут содержать в себе дополнительные метаданные. В базовом R создавать такие состояния достаточно сложно, но функция `rlang::abort()` значительно облегчает этот процесс за счет того, что вы можете передать вспомогательный аргумент `.subclass` и дополнительные метаданные.

В следующем примере приведен базовый шаблон. Я рекомендую использовать показанную структуру вызова для пользовательских состояний. Так вы сможете воспользоваться преимуществами R в отношении гибкости соответствия аргументов, передавая сначала тип ошибки, затем текст для отображения и после этого пользовательские метаданные.

```
abort(
    "error_not_found",
    message = "Path `blah.csv` not found",
    path = "blah.csv"
)
#> Error: Path `blah.csv` not found
```

При интерактивном использовании пользовательские состояния работают так же точно, как и стандартные, но при этом позволяют обработчикам выполнять больше работы.

8.5.1 Предпосылки

Для понимания всей картины давайте рассмотрим пример использования функции `base::log()`. Здесь при передаче некорректного аргумента выполняется минимум действий:

```
log(letters)
#> Error in log(letters): non-numeric argument to mathematical function
log(1:10, base = letters)
#> Error in log(1:10, base = letters): non-numeric argument to
#> mathematical function
```

Полагаю, в этом случае мы могли бы более явно указать на то, с каким именно аргументом возникла проблема (`x` или `base`), а также подробнее расписать суть проблемы.

```
my_log <- function(x, base = exp(1)) {
    if (!is.numeric(x)) {
        abort(paste0(
          "`x` must be a numeric vector; not ", typeof(x), "."
        ))
    }
    if (!is.numeric(base)) {
        abort(paste0(
          "`base` must be a numeric vector; not ", typeof(base), "."
        ))
    }

    base::log(x, base = base)
}
```

Теперь вывод будет более информативным:

```
my_log(letters)
#> Error: `x` must be a numeric vector; not character.
my_log(1:10, base = letters)
#> Error: `base` must be a numeric vector; not character.
```

В плане интерактивного использования функции мы сделали большой шаг вперед, значительно улучшив вывод об ошибке. Но это нам никак не поможет, если мы хотим программно обрабатывать возникающие ошибки, поскольку все полезные метаданные об ошибках будут передаваться в виде одной строки.

8.5.2 Сигнализирование

Давайте построим небольшую инфраструктуру для улучшения ситуации. Начнем с написания пользовательской функции `abort()` для некорректных аргументов. Применительно к данному примеру такое обобщение может

быть чрезмерным, но здесь мы увидим распространенный шаблон, который часто встречается в других функциях. Этот шаблон крайне прост. Мы создаем понятное для пользователя сообщение об ошибке с помощью функции `glue::glue()` и сохраняем метаданные в вызове состояния для разработчика.

```
abort_bad_argument <- function(arg, must, not = NULL) {
    msg <- glue::glue("`{arg}` must {must}")
    if (!is.null(not)) {
        not <- typeof(not)
        msg <- glue::glue("{msg}; not {not}.")
    }

    abort("error_bad_argument",
        message = msg,
        arg = arg,
        must = must,
        not = not
    )
}
```

Примечание. Если вы хотите вызвать пользовательскую ошибку в базовом R без использования пакета `rlang`, то можете создать объект состояния «вручную» и затем передать его функции stop():

```
stop_custom <- function(.subclass, message, call = NULL, ...) {
    err <- structure(
        list(
            message = message,
            call = call,
            ...
        ),
        class = c(.subclass, "error", "condition")
    )
    stop(err)
}

err <- catch_cnd(
    stop_custom("error_new", "This is a custom error", x = 10)
)
class(err)
err$x
```

Теперь можно переписать функцию `my_log()` с использованием этой новой вспомогательной функции:

```
my_log <- function(x, base = exp(1)) {
    if (!is.numeric(x)) {
        abort_bad_argument("x", must = "be numeric", not = x)
```

```
    }
    if (!is.numeric(base)) {
        abort_bad_argument("base", must = "be numeric", not = base)
    }

    base::log(x, base = base)
}
```

Сама функция `my_log()` меньше не стала, но она стала гораздо более осмысленной, и теперь она лучше обрабатывает передачу некорректных аргументов. Функция выводит те же интерактивные сообщения об ошибках, что и раньше:

```
my_log(letters)
#> Error: `x` must be numeric; not character.
my_log(1:10, base = letters)
#> Error: `base` must be numeric; not character.
```

8.5.3 Обработка

Программировать с использованием этих структурированных объектов состояний гораздо легче. Первая область, в которой можно применить эти возможности, – это тестирование ваших функций. Модульное тестирование – не предмет данной книги (см. книгу *R Packages* (https://r-pkgs.org)), но его основы понять несложно. В приведенном ниже коде мы перехватываем ошибку, а затем утверждаем, что ее структура соответствует ожидаемой.

```
library(testthat)

err <- catch_cnd(my_log("a"))
expect_s3_class(err, "error_bad_argument")
expect_equal(err$arg, "x")
expect_equal(err$not, "character")
```

Мы также можем использовать класс (`error_bad_argument`) в функции `tryCatch()` для обработки только этой конкретной ошибки:

```
tryCatch(
    error_bad_argument = function(cnd) "bad_argument",
    error = function(cnd) "other error",
    my_log("a")
)
#> [1] "bad_argument"
```

При использовании функции `tryCatch()` с несколькими обработчиками и пользовательскими классами будет вызываться первый обработчик, соответствующий любому классу в векторе классов сигнала, а не обработчик с максимальным совпадением. По этой причине вам необходимо разме-

щать наиболее специфические обработчики первыми в списке. Приведенный ниже код может повести себя не так, как вы ожидаете:

```
tryCatch(
    error = function(cnd) "other error",
    error_bad_argument = function(cnd) "bad_argument",
    my_log("a")
)
#> [1] "other error"
```

8.5.4 Упражнения

1. Внутри пакетов порой бывает полезно перед использованием пакета проверять, что он установлен. Напишите функцию, которая будет проверять установку пакета (с помощью функции `requireNamespace("pkg", quietly = FALSE)`) и в противном случае сигнализировать о возникновении пользовательского состояния с названием пакета в метаданных.
2. Внутри пакета часто возникает необходимость инициировать ошибки, когда что-то идет не так. В пакетах, зависящих от нашего, эти ошибки можно проверить в рамках модульных тестов. Как можно поспособствовать тому, чтобы в этих пакетах игнорировались сообщения об ошибках, являющиеся частью интерфейса, а не API и, следовательно, могущие меняться без предупреждения?

8.6 Практическое применение

Теперь, когда мы изучили основы системы состояний, используемой в R, пришло время взглянуть на ее практическое применение. В данном разделе я не собираюсь показывать все возможные способы применения функций `tryCatch()` и `withCallingHandlers()`, мы обратим внимание лишь на наиболее распространенные шаблоны их использования. Надеюсь, это вдохновит вас на творческие изыскания, и когда вы столкнетесь с проблемой, то будете готовы предложить для нее элегантное решение.

8.6.1 Значение ошибки

Есть несколько простых, но полезных шаблонов использования функции `tryCatch()`, построенных на основе возвращаемых из обработчика ошибок значений. Простейший случай представляет собой обертку для возвращения значения по умолчанию при возникновении ошибки:

```
fail_with <- function(expr, value = NULL) {
    tryCatch(
        error = function(cnd) value,
```

```
        expr
    )
}

fail_with(log(10), NA_real_)
#> [1] 2.3
fail_with(log("x"), NA_real_)
#> [1] NA
```

Более сложные подходы могут основываться на функции `base::try()`. Ниже приведена реализация функции `try2()`, в которой заложена суть функции `base::try()`. При этом новая функция является более сложной – это сделано для того, чтобы сообщения об ошибках выглядели так, будто функция `tryCatch()` не используется.

```
try2 <- function(expr, silent = FALSE) {
    tryCatch(
        error = function(cnd) {
            msg <- conditionMessage(cnd)
            if (!silent) {
                message("Error: ", msg)
            }
            structure(msg, class = "try-error")
        },
        expr
    )
}

try2(1)
#> [1] 1
try2(stop("Hi"))
#> Error: Hi
#> [1] "Hi"
#> attr(,"class")
#> [1] "try-error"
try2(stop("Hi"), silent = TRUE)
#> [1] "Hi"
#> attr(,"class")
#> [1] "try-error"
```

8.6.2 Значения успеха и ошибки

Мы можем расширить приведенный выше шаблон, чтобы в случае успеха возвращалось одно значение (`success_val`), а в случае неудачи – другое (`error_val`). Это изменение требует использования небольшого трюка, заключающегося в вычислении пользовательского выражения с последующим возвратом значения `success_val`. Если в процессе возникнет ошибка, мы просто не доберемся до значения `success_val`, а вместо него будет возвращено значение `error_val`.

```
foo <- function(expr) {
    tryCatch(
        error = function(cnd) error_val,
        {
            expr
            success_val
        }
    )
}
```

Этот прием можно использовать для проверки ошибочности выражения:

```
does_error <- function(expr) {
    tryCatch(
        error = function(cnd) TRUE,
        {
            expr
            FALSE
        }
    )
}
```

Также его можно применять для перехвата любого состояния, подобно функции `rlang::catch_cnd()`:

```
catch_cnd <- function(expr) {
    tryCatch(
        condition = function(cnd) cnd,
        {
            expr
            NULL
        }
    )
}
```

Еще показанный выше способ допустимо использовать для создания разновидности функции `try()`. Одним из неудобств функции `try()` является то, что при ее использовании бывает трудно определить, завершился код успехом или неудачей. Вместо возврата объекта особого класса, думаю, было бы лучше оперировать списком с двумя компонентами: `result` и `error`.

```
safety <- function(expr) {
    tryCatch(
        error = function(cnd) {
            list(result = NULL, error = cnd)
        },
        list(result = expr, error = NULL)
    )
}
```

```
str(safety(1 + 10))
#> List of 2
#> $ result: num 11
#> $ error : NULL
str(safety(stop("Error!")))
#> List of 2
#> $ result: NULL
#> $ error :List of 2
#> ..$ message: chr "Error!"
#> ..$ call : language doTryCatch(return(expr), name, parentenv, h..
#> ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

Этот код перекликается с функцией `purrr::safely()`, о которой мы будем говорить в разделе 11.2.1.

8.6.3 Повторное сигнализирование

Помимо возвращения значений по умолчанию при возникновении состояний, обработчики могут быть использованы с целью создания более информативных сообщений об ошибках. Простым примером этого является функция, работающая наподобие `options(warn = 2)` применительно к одному фрагменту кода. Идея проста: мы обрабатываем предупреждения путем выбрасывания ошибок:

```
warning2error <- function(expr) {
    withCallingHandlers(
        warning = function(cnd) abort(conditionMessage(cnd)),
        expr
    )
}

warning2error({
    x <- 2 ^ 4
    warn("Hello")
})
#> Error: Hello
```

Подобную функцию можно написать, если вы пытаетесь найти источник раздражающего сообщения. Подробнее об этом мы поговорим в разделе 22.6.

8.6.4 Запись состояний

Еще одним распространенным шаблоном является запись состояний для их дальнейшего исследования. Здесь стоит заметить, что обработчики вызова запускаются только с целью осуществления побочных эффектов и не могут возвращать значения, а значит, должны модифицировать некий объект прямо на месте.

```
catch_cnds <- function(expr) {
    conds <- list()
    add_cond <- function(cnd) {
        conds <<- append(conds, list(cnd))
        cnd_muffle(cnd)
    }

    withCallingHandlers(
        message = add_cond,
        warning = add_cond,
        expr
    )

    conds
}

catch_cnds({
    inform("a")
    warn("b")
    inform("c")
})
#> [[1]]
#> <message: a
#> >
#>
#> [[2]]
#> <warning: b>
#>
#> [[3]]
#> <message: c
#> >
```

А что, если нам нужно также перехватывать ошибки? В этом случае вам понадобится обернуть `withCallingHandlers()` в функцию `tryCatch()`. Если возникнет ошибка, она будет последним состоянием в списке.

```
catch_cnds <- function(expr) {
    conds <- list()
    add_cond <- function(cnd) {
        conds <<- append(conds, list(cnd))
        cnd_muffle(cnd)
    }

    tryCatch(
        error = function(cnd) {
            conds <<- append(conds, list(cnd))
        },
        withCallingHandlers(
            message = add_cond,
            warning = add_cond,
```

```
            expr
        )
    )

    conds
}

catch_cnds({
    inform("a")
    warn("b")
    abort("C")
})
#> [[1]]
#> <message: a
#> >
#>
#> [[2]]
#> <warning: b>
#>
#> [[3]]
#> <error>
#> message: C
#> class: `rlang_error`
#> backtrace:
#> 1. global::catch_cnds(...)
#> 6. base::withCallingHandlers(...)
#> Call `rlang::last_trace()` to see the full backtrace
```

Данная техника лежит в основе пакета `evaluate` [Уикем и Се (Xie), 2018], который используется в пакете `knitr`: он захватывает весь вывод и сохраняет его в специальную структуру, которая позже может быть воспроизведена. На самом деле при написании пакета `evaluate` использовалась гораздо более сложная техника, чем описана здесь, поскольку нам нужно было также захватывать графики и текстовый вывод.

8.6.5 Отсутствие поведения по умолчанию

В качестве последнего полезного шаблона мы рассмотрим сигнализирование о возникновении состояний, не являющихся прямыми наследниками `message`, `warning` или `error`. В отсутствие поведения по умолчанию состояние не производит никаких действий, если пользователь специально их не вызовет. Например, можно себе представить систему логирования на основе состояний:

```
log <- function(message, level = c("info", "error", "fatal")) {
    level <- match.arg(level)
    signal(message, "log", level = level)
}
```

При вызове функции `log()` появляется сигнал о возникновении состояния, но никаких действий не выполняется из-за отсутствия поведения по умолчанию:

```
log("This code was run")
```

Для активации логирования нам необходим обработчик, который будет что-то делать с состоянием `log`. Ниже показана функция `record_log()`, которая осуществляет запись сообщений логов в файл:

```
record_log <- function(expr, path = stdout()) {
    withCallingHandlers(
        log = function(cnd) {
            cat(
                "[", cnd$level, "] ", cnd$message, "\n", sep = "",
                file = path, append = TRUE
            )
        },
        expr
    )
}

record_log(log("Hello"))
#> [info] Hello
```

Можно также реализовать функцию, позволяющую выборочно подавлять некоторые уровни логирования.

```
ignore_log_levels <- function(expr, levels) {
    withCallingHandlers(
        log = function(cnd) {
            if (cnd$level %in% levels) {
                cnd_muffle(cnd)
            }
        },
        expr
    )
}

record_log(ignore_log_levels(log("Hello"), "info"))
```

Примечание. Если вы в базовом R создаете объект состояния вручную и сигнализируете о его возникновении с помощью функции `signalCondition()`, функция `cnd_muffle()` не отработает. Вместо этого необходимо вызвать ее с помощью рестарта `muffle`, определенного подобным образом:

```
withRestarts(signalCondition(cond), muffle = function() NULL)
```

В настоящий момент рестарты выходят за рамки данной книги, но я полагаю, что они будут включены в третье издание.

8.6.6 Упражнения

1. Напишите функцию `suppressConditions()`, которая будет работать как `suppressMessages()` и `suppressWarnings()`, но при этом сможет подавлять вывод любых состояний. Хорошо подумайте над обработкой ошибок.
2. Сравните следующие две реализации функции `message2error()`. В чем состоит основное преимущество использования функции `withCallingHandlers()` в данном сценарии? Подсказка: внимательно рассмотрите трассировку функций.

```
message2error <- function(code) {
    withCallingHandlers(code, message = function(e) stop(e))
}
message2error <- function(code) {
    tryCatch(code, message = function(e) stop(e))
}
```

3. Как бы вы изменили определение функции `catch_cnds()`, если бы вам нужно было воссоздать исходное перемешивание предупреждений и сообщений?
4. В чем опасность перехватывания прерываний? Запустите приведенный ниже код, чтобы понять и объяснить это.

```
bottles_of_beer <- function(i = 99) {
    message(
        "There are ", i, " bottles of beer on the wall, ",
        i, " bottles of beer."
    )
    while(i > 0) {
        tryCatch(
            Sys.sleep(1),
            interrupt = function(err) {
                i <<- i - 1
                if (i > 0) {
                    message(
                        "Take one down, pass it around, ", i,
                        " bottle", if (i > 1) "s", " of beer on the wall."
                    )
                }
            }
        )
    }
    message(
        "No more bottles of beer on the wall, ",
```

```
        "no more bottles of beer."
    )
}
```

8.7 Ответы на контрольные вопросы

1. `error`, `warning` и `message`.
2. Необходимо воспользоваться функцией `try()` или `tryCatch()`.
3. Функция `tryCatch()` создает обработчики выхода, которые прерывают выполнение исходного кода. Функция `withCallingHandlers()`, в свою очередь, создает обработчики вызова, которые не оказывают влияния на ход выполнения программы.
4. Поскольку это даст возможность перехватывать специфические типы ошибок с помощью функции `tryCatch()`, а не полагаться на сравнение сообщений об ошибках, особенно с учетом перевода на разные языки.

Часть II

Функциональное программирование

Введение

R по своей сути является *функциональным языком программирования* (functional language). Это означает, что он обладает определенными техническими особенностями и, что более важно, исповедует особенный стиль решения задач, основанный на функциях. Ниже я приведу краткий обзор технических аспектов функциональных языков, но в основном мы в этой книге сосредоточимся на функциональном стиле программирования, поскольку я считаю, что он идеально подходит для большинства задач, с которыми мы сталкиваемся при анализе данных.

В последнее время функциональная парадигма программирования вызвала лавину интереса из-за ее способности решать возникающие в процессе работы задачи в очень эффективной и элегантной манере. Этот стиль подразумевает создание функций, поддающихся анализу в отрыве от остального кода (т. е. с использованием только локальных данных), что значительно облегчает процесс автоматической оптимизации или распараллеливания. Кроме того, в последние годы главные недостатки функциональных языков программирования, заключающиеся в низкой производительности и иногда в непредсказуемом расходовании памяти, практически удалось свести на нет. Функциональное программирование вполне можно рассматривать в качестве эффективного дополнения объектно ориентированного программирования, ставшего в последние десятилетия доминирующей парадигмой в этой области.

Функциональные языки программирования

В любом языке программирования присутствуют функции. Так что же в действительности делает язык функциональным? Здесь есть масса различных вариантов ответов, но два из них являются наиболее распространенными.

Во-первых, в функциональных языках программирования присутствуют *функции первого класса* (first-class functions), т. е. функции, ведущие себя как любые другие структуры данных. В R это означает, что с функциями вы можете делать многое из того, что можете делать с векторами: присваивать их переменным, хранить в списках, передавать в виде аргументов другим функциям, создавать их внутри других функций и даже возвращать их в качестве результата работы функции.

Во-вторых, многие функциональные языки программирования требуют, чтобы функции были *чистыми* (pure). Функция может называться чистой, если она удовлетворяет двум требованиям:

- ее вывод должен зависеть только от входа, т. е. вызов одной и той же функции с теми же входными аргументами всегда должен приводить

к одинаковому результату. Таким образом, мы исключаем функции вроде `runif()`, `read.csv()` или `Sys.time()`, поскольку они могут вести себя по-разному;

- функция не должна обладать побочными эффектами, т. е. изменять значения глобальных переменных, записывать данные на диск или отображать их на экране. Здесь мы отсекаем такие функции, как `print()`, `write.csv()` и `<-`.

Чистые функции весьма просты в обращении, но вполне очевидно, что они обладают серьезными недостатками: только представьте себе анализ данных без возможности сгенерировать случайные числа или считать данные с диска.

Если говорить строго, язык R не является функциональным в чистом виде, поскольку не накладывает ограничение на написание исключительно чистых функций. Тем не менее вы можете применять элементы функционального программирования. При этом вы не обязаны писать чистые функции, но часто будете это делать. По моему опыту, разделение кода на исключительно чистые функции и нечистые приводит к тому, что код становится гораздо более читабельным и масштабируемым.

Стиль функционального программирования

Дать точное определение функциональному стилю программирования непросто, но в целом мне кажется, что он заключается в декомпозиции больших и сложных задач на более мелкие и решении каждой из них отдельно при помощи одной или нескольких функций. При использовании функциональной парадигмы вы разбиваете компоненты общей задачи на отдельные функции, работающие независимо друг от друга. При этом каждая функция является абсолютно обособленной и достаточно простой для понимания, а сложность решения может быть обусловлена необходимостью возникновения нелинейных связей между различными функциями.

Следующие три главы мы посвятим трем разным техникам, призванным облегчить процесс декомпозиции сложных задач:

- в главе 9 мы узнаем, как можно заменить множество циклов `for` простыми функционалами, которые представляют из себя функции (вроде `lapply()`), принимающие другие функции в качестве аргументов. Функционалы позволяют взять обычную функцию, предназначенную для решения определенной задачи с единственным входом, и обобщить ее для использования со множеством входов. Использование функционалов является важнейшей техникой, и в процессе анализа данных вы будете прибегать к их помощи постоянно;
- в главе 10 мы познакомимся с фабриками функций, т. е. с функциями, производящими другие функции. Фабрики функций используются в ра-

боте реже, чем функционалы, но могут позволить вам очень элегантно разделить работу между различными участками кода;

- в главе 11 мы научимся создавать функциональные операторы – функции, принимающие на вход функции и возвращающие функции. Их можно сравнить с наречиями, поскольку обычно они изменяют действие функции.

Вместе эти типы функций называются *функциями высшего порядка* (higher-order functions), и их можно представить, как показано в табл. 9.1.

Таблица 9.1 Функции высшего порядка

Вход/выход	**Вектор**	**Функция**
Вектор	Обычная функция	Фабрика функций
Функция	Функционал	Функциональный оператор

9

Функционалы

9.1 Введение

> *Для существенного повышения надежности код должен быть более прозрачным. В частности, к вложенным состояниям и циклам следует относиться с большим подозрением. Усложненный поток выполнения программы приводит разработчиков в замешательство. В запутанном коде часто кроются баги.*
>
> *Бьерн Страуструп (Bjarne Stroustrup)*

Функционал (functional) – это функция, принимающая на вход другую функцию и возвращающая вектор. Ниже приведен простейший пример функционала: в нем переданная в качестве аргумента функция запускается для 1000 случайных значений из равномерного распределения:

```
randomise <- function(f) f(runif(1e3))
randomise(mean)
#> [1] 0.506
randomise(mean)
#> [1] 0.501
randomise(sum)
#> [1] 489
```

Велика вероятность, что вы уже встречались с функционалами в своей практике. Например, вы могли использовать в качестве замены циклам `for` функции базового R `lapply()`, `apply()` или `tapply()`, применять функцию `map()` из пакета `purrr` или обращаться к математическим функционалам вроде `integrate()` или `optim()`.

Как мы уже сказали, главное назначение функционалов состоит в замене привычных циклов `for`. Циклы в R обладают дурной славой, поскольку большинство разработчиков считают их медленными[1], но на самом деле главный их недостаток состоит в чрезмерной гибкости: циклы объявляют итерации,

[1] В действительности медленно работают не сами циклы, а то, что в них выполняется. Одним из виновников замедления работы циклов является изменение структур данных с повторным созданием копий. Смотрите разделы 2.5.1 и 24.6 для дополнительной информации.

но не указывают на то, что нужно делать с результатами. Подобно тому, что лучше использовать `while` вместо `repeat` и `for` вместо `while` (см. раздел 5.3.2), всегда лучше воспользоваться функционалом, чем циклом `for`. Каждый функционал разрабатывается под конкретную задачу, так что всякий раз, когда вы видите функционал, вы тут же понимаете, для чего он используется.

Если вы уверенно применяете в работе циклы, то можете воспринимать переход на функционалы в качестве упражнения по сопоставлению шаблонов. Смотрите на цикл `for` и ищите функционал, соответствующий его базовой форме. Если вы его не видите, не мучайте существующие функционалы подгонкой к нужному вам виду, а просто оставьте цикл как есть! Также при повторном использовании одного и того же цикла вы можете задуматься о написании собственного функционала.

Структура главы

- В разделе 9.2 мы познакомимся с первым функционалом – `purrr::map()`.
- Раздел 9.3 демонстрирует способы объединения простых функционалов для решения более сложных задач и отличия стиля, принятого в `purrr`, от других подходов.
- В разделе 9.4 мы рассмотрим 18 (!!) разных вариантов функционала `purrr::map()`. К счастью, примененная в них структура позволяет без труда запомнить их и научиться использовать на практике.
- В разделе 9.5 мы познакомимся с функционалом нового стиля: `purrr::reduce()`. `reduce()` постепенно сводит вектор к одиночному значению, последовательно применяя функцию к двум очередным входным параметрам.
- В разделе 9.6 мы поговорим о предикатах – функциях, возвращающих значение `TRUE` или `FALSE`, – а также о семействе функционалов, использующих эти функции для решения распространенных задач.
- В разделе 9.7 мы рассмотрим несколько функционалов из базового R, не входящих в семейства `map()`, `reduce()` и не являющихся предикатами.

Требования

В данной главе мы сосредоточимся на функционалах, представленных в пакете `purrr` [Хенри и Уикем, 2018a] (https://purrr.tidyverse.org). Эти функции обладают единообразным интерфейсом, что значительно облегчает процесс усвоения их ключевых идей в сравнении с их базовыми эквивалентами, которые на протяжении многих лет развивались обособленно друг от друга. В этой главе я буду приводить сравнение функционалов с их аналогами, а в заключение перечислю базовые функционалы, не имеющие эквивалентов в пакете `purrr`.

```
library(purrr)
```

9.2 Мой первый функционал: map()

Наиболее фундаментальным функционалом является `purrr::map()`[1]. На вход он принимает вектор и функцию, вызывает переданную функцию для каждого элемента в векторе и возвращает результат в виде списка. Иными словами, вызов `map(1:3, f)` эквивалентен выражению `list(f(1), f(2), f(3))`.

```
triple <- function(x) x * 3
map(1:3, triple)
#> [[1]]
#> [1] 3
#>
#> [[2]]
#> [1] 6
#>
#> [[3]]
#> [1] 9
```

Графически это можно представить так, как показано на рис. 9.1.

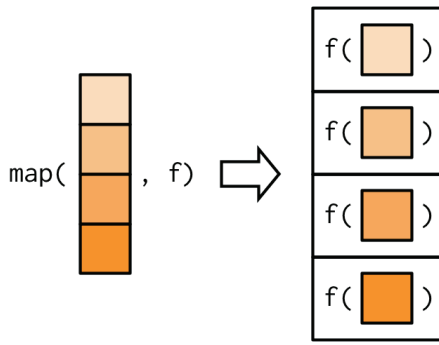


Рис. 9.1 Схема работы функционала `purrr::map()`

Примечание. Вас может заинтересовать, почему эта функция получила имя `map` (карта). Что у нее общего с изображениями географических карт? Фактически это название идет из области математики, где `map` относится к «операции, сопоставляющей каждый элемент заданного множества с одним или более элементами другого множества». В программировании функция `map` определяет соответствие между двумя векторами, что очень близко по смыслу к математическому определению. Кроме того, `map` – довольно короткое и емкое слово, что очень удобно для столь распространенной операции.

[1] Не путайте с гораздо более сложным функционалом `base::Map()`. Мы поговорим о нем в разделе 9.4.5.

Реализация функции `map()` достаточно проста. Мы создаем список такой же длины, как у входного вектора, после чего заполняем его с помощью цикла `for`. Таким образом, вся реализация может уместиться всего в несколько строк кода:

```
simple_map <- function(x, f, ...) {
    out <- vector("list", length(x))
    for (i in seq_along(x)) {
        out[[i]] <- f(x[[i]], ...)
    }
    out
}
```

В действительности же функция `purrr::map()` написана на языке C, чтобы получить максимум производительности, кроме того, она сохраняет имена и поддерживает несколько вариантов сокращения записи, о чем мы поговорим в разделе 9.2.2.

Примечание. В базовом R аналогом функции `map()` является `lapply()`. Единственное различие между ними состоит в том, что функция `lapply()` не поддерживает вспомогательный синтаксис, о котором мы будем говорить далее, так что если вы используете функцию `map()` из пакета `purrr` исключительно в чистом виде, вы можете не нагружать код дополнительными зависимостями, а просто воспользоваться функцией `lapply()`.

9.2.1 Создание атомарных векторов

Функция `map()` возвращает список, что делает ее наиболее обобщенной в своем семействе, поскольку в списке можно разместить что угодно. Однако бывает не совсем уместно возвращать целый список, когда нам необходима гораздо более простая структура данных. По этой причине существует четыре разновидности функции `map()`, возвращающие атомарные векторы соответствующего типа. Это функции `map_lgl()`, `map_int()`, `map_dbl()` и `map_chr()`:

```
# map_chr() всегда возвращает символьный вектор
map_chr(mtcars, typeof)
#>      mpg      cyl     disp       hp     drat       wt     qsec
#> "double" "double" "double" "double" "double" "double" "double"
#>       vs       am     gear     carb
#> "double" "double" "double" "double"

# map_lgl() всегда возвращает логический вектор
map_lgl(mtcars, is.double)
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#> TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

# map_int() всегда возвращает целочисленный вектор
```

```
n_unique <- function(x) length(unique(x))
map_int(mtcars, n_unique)
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#>   25    3   27   22   22   29   30    2    2    3    6

# map_dbl() всегда возвращает вектор двойной точности
map_dbl(mtcars, mean)
#>     mpg     cyl    disp      hp    drat      wt    qsec      vs
#>  20.091  6.188 230.722 146.688   3.597   3.217  17.849   0.438
#>      am    gear    carb
#>   0.406  3.688   2.812
```

В пакете `purrr` используется соглашение, по которому окончание имени функции (например, `_dbl()`) указывает на соответствующий тип вывода. Все функции группы `map_*()` могут принимать на вход вектор любого типа. В примерах, показанных выше, мы воспользовались тем, что `mtcars` является датафреймом, а датафреймы представляют собой списки, содержащие векторы одинаковой длины. Это становится более очевидным, если изобразить датафрейм в виде, принятом в векторах, что показано на рис. 9.2.

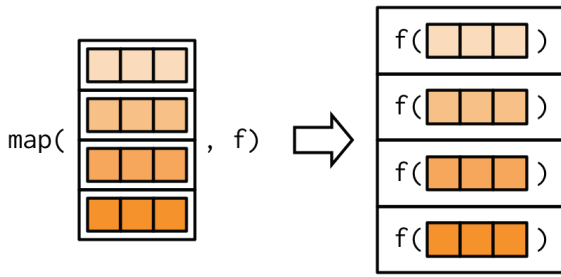


Рис. 9.2 Отображение датафрейма в виде векторов

Все функции группы `map` возвращают вектор той же длины, что и входной, а это предполагает, что каждый вызов внутренней функции должен возвращать единственное значение. В противном случае вы получите ошибку:

```
pair <- function(x) c(x, x)
map_dbl(1:2, pair)
#> Error: Result 1 must be a single double, not an integer vector of
#> length 2
```

Похожую ошибку вы увидите и в случае, если внутренняя функция возвращает результат неправильного типа:

```
map_dbl(1:2, as.character)
#> Error: Can't coerce element 1 from a character to a double
```

В обоих случаях было бы полезно вернуться к использованию функции `map()`, поскольку она допускает вывод любого типа. Это позволит вам выявить проблему в выводе и решить, что с ней делать.

```
map(1:2, pair)
#> [[1]]
#> [1] 1 1
#>
#> [[2]]
#> [1] 2 2
map(1:2, as.character)
#> [[1]]
#> [1] "1"
#>
#> [[2]]
#> [1] "2"
```

Примечание. В базовом R представлены две функции, способные возвращать атомарные векторы: `sapply()` и `vapply()`. Лично я рекомендую избегать использования функции `sapply()` по причине того, что она всегда пытается упростить результат, из-за чего вы можете получить на выходе список, вектор или матрицу. Это затрудняет автоматизацию процесса, и в неинтерактивном окружении я советую эту функцию не использовать. Функция `vapply()` является более безопасной, поскольку позволяет передать шаблон `FUN.VALUE`, описывающий форму вывода. Если вы не хотите использовать пакет `purrr`, я рекомендую всегда делать выбор в пользу функции `vapply()`, а не `sapply()`. Основным недостатком `vapply()` является ее многословность. Судите сами: эквивалентом лаконичного вызова `map_dbl(x, mean, na.rm = TRUE)` в базовом пакете будет `vapply(x, mean, na.rm = TRUE, FUN.VALUE = double(1))`.

9.2.2 Анонимные функции и сокращенная запись

Вместо использования функции `map()` с существующей функцией вы можете создать встроенную *анонимную функцию* (anonymous function), как было показано в разделе 6.2.3:

```
map_dbl(mtcars, function(x) length(unique(x)))
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#>   25    3   27   22   22   29   30    2    2    3    6
```

Анонимные функции могут быть очень полезны, но их синтаксис выглядит громоздким. В связи с этим в пакете `purrr` поддерживаются определенные сокращения:

```
map_dbl(mtcars, ~ length(unique(.x)))
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#>   25    3   27   22   22   29   30    2    2    3    6
```

Это работает по причине того, что все функции пакета `purrr` преобразуют формулы, обозначенные с помощью символа ~ (тильда), в функции. Вы можете увидеть, что происходит за сценой, вызвав функцию `as_mapper()`:

```
as_mapper(~ length(unique(.x)))
#> <lambda>
#> function (..., .x = ..1, .y = ..2, . = ..1)
#> length(unique(.x))
#> attr(,"class")
#> [1] "rlang_lambda_function"
```

Согласен, аргументы функции выглядят несколько причудливо, но здесь лишь показано, что вы можете обратиться при помощи точки (`.`) к единственному аргументу внутренней функции, использовать запись `.x` и `.y` при работе с функциями, принимающими два аргумента, и применять нотацию `..1`, `..2`, `..3` и т. д. для получения произвольного количества аргументов функции. Возможность обращения к единственному аргументу функции при помощи точки была сохранена с целью обратной совместимости, но я не рекомендую использовать такую нотацию, поскольку ее легко спутать с аналогичным обращением в конвейерах при работе с пакетом `magrittr`.

Сокращенная запись может быть, в частности, полезна при генерировании случайных данных:

```
x <- map(1:3, ~ runif(2))
str(x)
#> List of 3
#> $ : num [1:2] 0.281 0.53
#> $ : num [1:2] 0.433 0.917
#> $ : num [1:2] 0.0275 0.8249
```

Приберегите такой синтаксис для коротких и простых функций. Как правило, если функция занимает несколько строк или ее тело заключается в фигурные скобки ({}), будет лучше дать ей уникальное имя.

В функциях группы `map` также может использоваться сокращенная запись для извлечения значений из вектора, в основе которой лежит функция `purrr::pluck()`. При этом вы можете воспользоваться символьным вектором для выбора элементов по имени, целочисленным вектором для выбора по позиции или списком для смешанного выбора. Этот прием может быть полезен при работе со списками большой вложенности, что характерно при взаимодействии с документами JSON.

```
x <- list(
    list(-1, x = 1, y = c(2), z = "a"),
    list(-2, x = 4, y = c(5, 6), z = "b"),
```

```
    list(-3, x = 8, y = c(9, 10, 11))
)

# Выбор по имени
map_dbl(x, "x")
#> [1] 1 4 8

# Или по позиции
map_dbl(x, 1)
#> [1] -1 -2 -3

# Или и так, и так
map_dbl(x, list("y", 1))
#> [1] 2 5 9

# Если элемент отсутствует, вы получите ошибку
map_chr(x, "z")
#> Error: Result 3 must be a single string, not NULL of length 0

# Можно передать значение по умолчанию
map_chr(x, "z", .default = NA)
#> [1] "a" "b" NA
```

Примечание. В базовом R функции, подобные `lapply()`, могут принимать имя вызываемой функции в виде строки. Не сказать, чтобы в этом было много смысла, поскольку запись `lapply(x, "f")` почти всегда будет эквивалентна `lapply(x, f)`.

9.2.3 Передача дополнительных аргументов

Часто бывает необходимо передать дополнительные аргументы во внутреннюю функцию. К примеру, вам может понадобиться пробросить аргумент `na.rm = TRUE` в функцию `mean()`. Это можно сделать с помощью анонимной функции следующим образом:

```
x <- list(1:5, c(1:10, NA))
map_dbl(x, ~ mean(.x, na.rm = TRUE))
#> [1] 3.0 5.5
```

Но поскольку функции группы `map` передают дальше аргумент …, можно воспользоваться и более простой формой записи:

```
map_dbl(x, mean, na.rm = TRUE)
#> [1] 3.0 5.5
```

Легче всего это продемонстрировать на рисунке: все аргументы, поступающие после `f` при вызове функции `map()`, передаются после аргумента с данными во внутреннюю функцию `f()`, что показано на рис. 9.3.

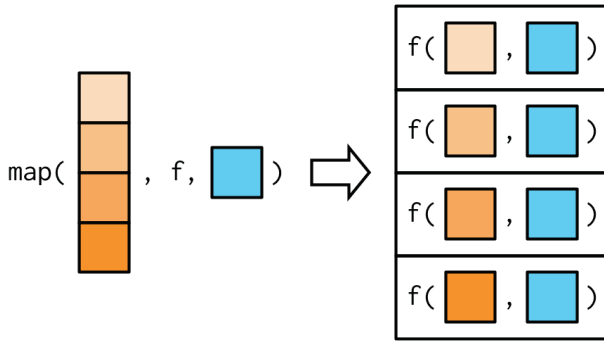


Рис. 9.3 Передача дополнительных аргументов во внутреннюю функцию

Важно отметить, что эти аргументы не раскладываются на составляющие. Иначе говоря, функция `map()` является векторизованной только по своему первому аргументу. Если после `f` будет передан вектор, он будет отправлен во внутреннюю функцию как есть. Это показано на рис. 9.4.

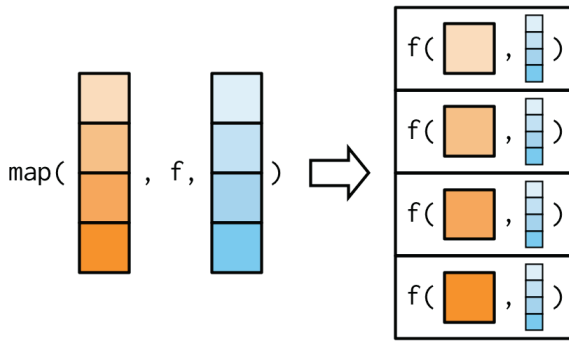


Рис. 9.4 Передача вектора во внутреннюю функцию

В разделах 9.4.2 и 9.4.5 вы узнаете о разновидностях функции `map()`, векторизованных по нескольким аргументам.

Важно отметить небольшую, но важную разницу между передачей дополнительных аргументов с помощью анонимной функции и непосредственно в функцию `map()`. В первом случае они будут вычисляться всякий раз, когда будет запускаться функция `f()`, а во втором – единожды, при вызове функции `map()`. Это легко увидеть, если в качестве дополнительного аргумента передать случайное число:

```
plus <- function(x, y) x + y

x <- c(0, 0, 0, 0)
map_dbl(x, plus, runif(1))
#> [1] 0.0625 0.0625 0.0625 0.0625
```

```
map_dbl(x, ~ plus(.x, runif(1)))
#> [1] 0.903 0.132 0.629 0.945
```

9.2.4 Имена аргументов

На приведенных выше рисунках я намеренно опускал имена аргументов, чтобы сосредоточиться на самой концепции. При этом я настоятельно рекомендую использовать полные имена аргументов в вашем коде, поскольку это значительно облегчает процесс чтения. Вызов `map(x, mean, 0.1)` вполне корректен с точки зрения синтаксиса, но он приведет к запуску выражения `mean(x[[1]], 0.1)`, а это означает, что читателю будет необходимо помнить, что вторым аргументом функции `mean()` является `trim`. Таким образом, если вы не хотите лишний раз перегружать мозги читателя вашего кода ненужной информацией[1], потрудитесь оформить вызов следующим образом: `map(x, mean, trim = 0.1)`.

В этом кроется причина такого странного именования аргументов в функции `map()`: вместо привычных `x` и `f` они приобретают имена `.x` и `.f`. Повод для такого именования можно легко обнаружить на примере функции `simple_map()`, определенной нами ранее. Эта функция принимает на вход аргументы `x` и `f`, так что вас ожидают проблемы, в случае если у вызываемой внутри функции также найдутся аргументы с такими именами:

```
boostrap_summary <- function(x, f) {
    f(sample(x, replace = TRUE))
}

simple_map(mtcars, boostrap_summary, f = mean)
#> Error in mean.default(x[[i]], ...): 'trim' must be numeric of length
#> one
```

Появившаяся ошибка может сбивать с толку, если не вспомнить, что приведенный вызов `simple_map()` эквивалентен `simple_map(x = mtcars, f = mean, boot-strap_summary)`, поскольку именованные аргументы обладают более высоким приоритетом по сравнению с позиционными.

В пакете `purrr` проблема возможных конфликтов имен решена с помощью именования аргументов при помощи точки. Конечно, эта техника не идеальна, поскольку в вызываемых функциях также могут присутствовать аргументы с такими странными именами, как `.f` и `.x`, но она помогает решить 99 % проблем. В оставшихся случаях можно воспользоваться анонимными функциями.

Примечание. Базовые функции, передающие во внутреннюю функцию аргумент …, используют следующие соглашения об именовании во избежание конфликтов имен:

[1] А этим читателем можете оказаться вы сами в недалеком будущем!

- в семействе функций `apply()` по большей части используются заглавные буквы для аргументов (т. е. `X`, `FUN` и т. д.);
- в функции `transform()` используется более экзотический префикс `_`: это делает имя синтаксически неправильным, что вынуждает нас использовать обрамление знаками обратного апострофа (`` ` ``), как мы говорили в разделе 2.2.1. Это дает почти полную гарантию отсутствия конфликтов;
- в некоторых функционалах, таких как `uniroot()` и `optim()`, не предпринимается никаких шагов для предотвращения конфликтов имен, но они чаще всего используются в совокупности со специально созданными функциями, так что риск невелик.

9.2.5 Изменение другого аргумента

До сих пор первый аргумент функции `map()` всегда передавался первым во внутреннюю функцию. Но что, если первый аргумент должен быть константой, а меняться будет другой аргумент? Как получить результат, показанный на рис. 9.5?

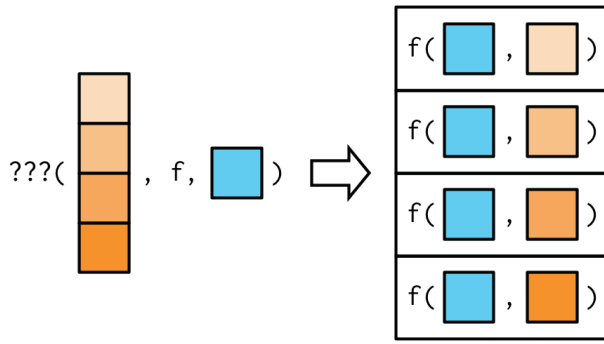


Рис. 9.5 Изменение другого аргумента при вызове функции

Оказывается, нет способа сделать это напрямую, зато есть пара трюков, которые могут позволить осуществить подобную реализацию. Представьте, что у нас есть вектор с несколькими значениями и нам необходимо исследовать эффект усечения среднего с применением каждого из этих значений. В данном случае первый аргумент функции `mean()` будет константой, а варьироваться будет второй аргумент, представляющий величину усечения среднего (`trim`).

```
trims <- c(0, 0.1, 0.2, 0.5)
x <- rcauchy(1000)
```

Простейшей техникой здесь является применение анонимной функции для изменения порядка следования аргументов:

```
map_dbl(trims, ~ mean(x, trim = .x))
#> [1] -0.3500 0.0434 0.0354 0.0502
```

Но это не лучший способ, поскольку мне пришлось явно использовать аргументы x и .x. В этом случае лучше отказаться от использования сокращенного варианта и написать так:

```
map_dbl(trims, function(trim) mean(x, trim = trim))
#> [1] -0.3500  0.0434  0.0354  0.0502
```

Иногда, чтобы продемонстрировать свой (чрезмерный) интеллект, вы можете воспользоваться преимуществами гибкой системы сопоставления аргументов в R, о которой мы писали в разделе 6.8.2. Например, в данном примере вы могли бы переписать выражение `mean(x, trim = 0.1)` так: `mean(0.1, x = x)` – и в результате прийти к следующему вызову функции `map_dbl()`:

```
map_dbl(trims, mean, x = x)
#> [1] -0.3500  0.0434  0.0354  0.0502
```

Лично я не рекомендую использовать эту технику, поскольку она предполагает, что читатель знаком и с порядком передачи аргументов во внутреннюю функцию, и с правилами сопоставления аргументов в R.

Еще один альтернативный вариант реализации этого требования мы рассмотрим в разделе 9.4.5.

9.2.6 Упражнения

1. С помощью функции `as_mapper()` исследуйте, как `purrr` генерирует анонимные функции для целых чисел, строк и списков. Какая вспомогательная функция позволяет вам извлечь атрибуты? Ознакомьтесь с документацией, чтобы ответить на вопросы.
2. `map(1:3, ~ runif(2))` – полезный шаблон для генерирования случайных чисел, тогда как `map(1:3, runif(2))` – нет. Почему? Можете объяснить, почему это выражение возвращает такой результат?
3. Воспользуйтесь подходящей функцией `map()` для:
   a) расчета стандартных отклонений по каждой колонке числового датафрейма;
   b) расчета стандартных отклонений по каждой числовой колонке в смешанном датафрейме. Подсказка: для этого вам придется выполнить два шага;
   c) расчета количества уровней для каждого фактора в датафрейме.
4. Следующий код моделирует выполнение t-теста для данных из распределения, не являющегося нормальным. Извлеките p-значение для каждого теста и визуализируйте результаты.

   ```
   trials <- map(1:100, ~ t.test(rpois(10, 10), rpois(7, 10)))
   ```

5. В приведенном ниже коде функция `map()` используется внутри другой функции `map()` с целью применения функции к каждому элементу вложенного списка. Почему этот код не работает? И что необходимо сделать, чтобы исправить ситуацию?

```
x <- list(
    list(1, c(3, 9)),
    list(c(3, 6), 7, c(4, 7, 6))
)

triple <- function(x) x * 3
map(x, map, .f = triple)
#> Error in .f(.x[[i]], ...): unused argument (map)
```

6. Воспользуйтесь функцией `map()` для подгонки линейных моделей к набору данных `mtcars` с помощью формул, сохраненных в виде списка:

```
formulas <- list(
    mpg ~ disp,
    mpg ~ I(1 / disp),
    mpg ~ disp + wt,
    mpg ~ I(1 / disp) + wt
)
```

7. Выполните подгонку модели `mpg ~ disp` к каждому бутстрэпу данных из набора `mtcars` в списке, представленном ниже, после чего извлеките показатель R^2. Подсказка: показатель R^2 может быть вычислен с помощью функции `summary()`.

9.3 Стиль purrr

Перед тем как перейти к рассмотрению разных вариантов функции `map`, давайте бросим взгляд на то, как можно использовать разные функции из пакета `purrr` для решения вполне реалистичных задач: подгонки модели к каждой подгруппе и извлечения коэффициентов модели. Для нашего примера я разобью набор данных `mtcars` на группы по количеству цилиндров с помощью базовой функции `split`:

```
by_cyl <- split(mtcars, mtcars$cyl)
```

В результате мы получим три датафрейма для машин с четырьмя, шестью и восемью цилиндрами соответственно.

Теперь представим, что нам необходимо выполнить подгонку линейной модели, после чего извлечь ее второй коэффициент, т. е. угол наклона. Ниже показано, как это можно сделать с помощью пакета `purrr`:

```
by_cyl %>%
    map(~ lm(mpg ~ wt, data = .x)) %>%
    map(coef) %>%
    map_dbl(2)
#> 4 6 8
#> -5.65 -2.78 -2.19
```

Если вы раньше не встречались с оператором конвейера `%>%`, обратитесь к разделу 6.3.

Полагаю, этот код очень просто читать по причине того, что, во-первых, каждая строка соответствует отдельному шагу, во-вторых, можно легко отделить функционалы от их действий, ну и вспомогательные функции пакета `purrr` позволяют кратко и четко описывать происходящее на каждом шаге.

Как можно было бы решить эту задачу с использованием только средств базового R? Вы могли бы заменить все функции `purrr` их аналогами из пакета `base`:

```
by_cyl %>%
    lapply(function(data) lm(mpg ~ wt, data = data)) %>%
    lapply(coef) %>%
    vapply(function(x) x[[2]], double(1))
#>     4     6     8
#> -5.65 -2.78 -2.19
```

Но и это не совсем базовый R, поскольку мы здесь используем операторы конвейера. Чтобы очистить решение от них, вы могли бы воспользоваться промежуточной переменной, но тогда на каждом шаге вам пришлось бы выполнять больше работы:

```
models <- lapply(by_cyl, function(data) lm(mpg ~ wt, data = data))
vapply(models, function(x) coef(x)[[2]], double(1))
#>     4     6     8
#> -5.65 -2.78 -2.19
```

Конечно, можно было бы воспользоваться и циклом `for`, как показано ниже:

```
intercepts <- double(length(by_cyl))
for (i in seq_along(by_cyl)) {
    model <- lm(mpg ~ wt, data = by_cyl[[i]])
    intercepts[[i]] <- coef(model)[[2]]
}
intercepts
#> [1] -5.65 -2.78 -2.19
```

Заметьте, что при движении от стиля, используемого в пакете `purrr`, через базовые функции с конвейерами и к чистому базовому R с циклами вам приходилось на каждой итерации выполнять все больше действий. При при-

менении пакета purrr мы использовали три итерации (map(), map(), map_dbl()), с функциями группы apply мы перешли к двум итерациям (lapply(), vapply()), а в решении с циклом for у нас осталась одна итерация. Лично я предпочитаю вариант с большим количеством более простых шагов. Такой код гораздо легче читать и поддерживать.

9.4 Разновидности функции map

Существует 23 основные разновидности функции map(). До сих пор мы говорили о пяти из них (map(), map_lgl(), map_int(), map_dbl() и map_chr()). А значит, нам предстоит изучить еще 18 (!!) вариантов. Звучит неподъемно, но, к счастью, структура пакета purrr позволяет свести это количество всего к пяти основным идеям:

- вывод того же типа, что и ввод (modify());
- итерации по двум входам (map2());
- итерации с индексом (imap());
- отсутствие возврата (walk());
- итерации по любому количеству входов (pmap()).

Семейство map() предполагает наличие независимости входов и выходов функций, что позволяет нам свести эти функции в таблицу с входами в строках и выходами в колонках. Таким образом, если вам известны входы, вы можете совместить их с любым нужным вам количеством выходов, и наоборот. Эти взаимосвязи показаны в табл. 9.2.

Таблица 9.2 Функции семейства map()

	Список	Атомарный вектор	Вектор того же типа	Нет выхода
Один аргумент	map()	map_lgl(), …	modify()	walk()
Два аргумента	map2()	map2_lgl(), …	modify2()	walk2()
Один аргумент + индекс	imap()	imap_lgl(), …	imodify()	iwalk()
N аргументов	pmap()	pmap_lgl(), …	_	pwalk()

9.4.1 Вход и выход одного типа: modify()

Представьте, что вам нужно удвоить все значения в определенной колонке в датафрейме. Можно воспользоваться функцией map(), но она всегда возвращает список:

```
df <- data.frame(
    x = 1:3,
    y = 6:4
```

```
)

map(df, ~ .x * 2)
#> $x
#> [1] 2 4 6
#>
#> $y
#> [1] 12 10  8
```

Если необходимо, чтобы результат также имел вид датафрейма, будет лучше использовать функцию `modify()`, которая всегда возвращает результат того же типа, что и данные на входе:

```
modify(df, ~ .x * 2)
#>   x  y
#> 1 2 12
#> 2 4 10
#> 3 6  8
```

Несмотря на свое имя, функция `modify()` не изменяет значения на месте, а возвращает модифицированную копию входных данных, так что если вы хотите оставить результирующий набор данных в исходном датафрейме, вам необходимо его переназначить:

```
df <- modify(df, ~ .x * 2)
```

Как обычно, базовая реализация функции `modify()` очень проста. Она даже проще, чем реализация функции `map()`, поскольку нам не нужно создавать новый выходной вектор, мы просто можем последовательно менять входные данные (в действительности код реализации чуть сложнее, что помогает правильно обработать частные случаи).

```
simple_modify <- function(x, f, ...) {
    for (i in seq_along(x)) {
        x[[i]] <- f(x[[i]], ...)
    }
    x
}
```

В разделе 9.6.2 мы узнаем об очень полезной разновидности `modify()` – функции `modify_if()`. Она, например, позволяет удвоить значения только в числовых колонках датафрейма при помощи следующей записи: `modify_if(df, is.numeric, ~ .x * 2)`.

9.4.2 Два входа: функция map2() и другие

Функция `map()`, как мы уже говорили, является векторизованной по одному первому аргументу `.x`. Это означает, что при вызове функции `.f` перебор бу-

дет осуществляться только по аргументу .x, а все остальные аргументы будут передаваться во внутреннюю функцию без изменений, что неприемлемо для целого ряда задач. К примеру, как нам найти средневзвешенное, если у нас есть список наблюдений и список весов? Допустим, у нас есть следующие исходные данные:

```
xs <- map(1:8, ~ runif(10))
xs[[1]][[1]] <- NA
ws <- map(1:8, ~ rpois(10, 5) + 1)
```

Можно воспользоваться функцией map_dbl() для расчета невзвешенных средних:

```
map_dbl(xs, mean)
#> [1] NA    0.463 0.551 0.453 0.564 0.501 0.371 0.443
```

Но если передать дополнительным аргументом переменную ws, в которой хранятся веса, все сломается, поскольку аргументы после .f не преобразованы (это проиллюстрировано на рис. 9.6):

```
map_dbl(xs, weighted.mean, w = ws)
#> Error in weighted.mean.default(.x[[i]], ...): 'x' and 'w' must have
#> the same length
```

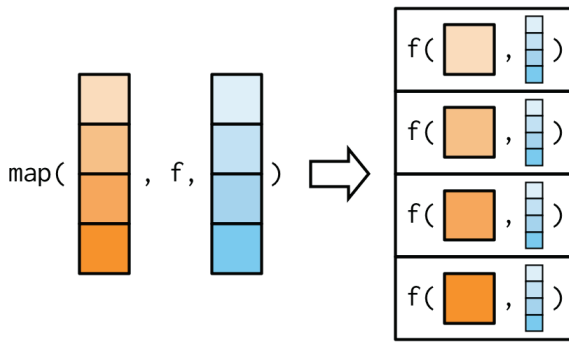


Рис. 9.6 Попытка передать веса элементов для вычисления средневзвешенного

Таким образом, нам нужен новый инструмент, и им может стать функция map2(), векторизованная по двум аргументам. Это означает, что оба аргумента .x и .y будут перебираться при каждом вызове функции .f, что показано на рис. 9.7:

```
map2_dbl(xs, ws, weighted.mean)
#> [1]    NA 0.451 0.603 0.452 0.563 0.510 0.342 0.464
```

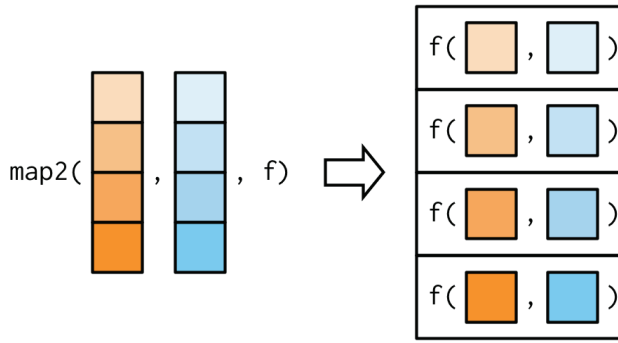


Рис. 9.7 Использование функции группы map2

Аргументы функции map2() немного отличаются от map(), поскольку оба вектора в данном случае идут перед функцией, а не один из них. Дополнительные аргументы по-прежнему перечисляются в конце, что показано на рис. 9.8:

```
map2_dbl(xs, ws, weighted.mean, na.rm = TRUE)
#> [1] 0.504 0.451 0.603 0.452 0.563 0.510 0.342 0.464
```

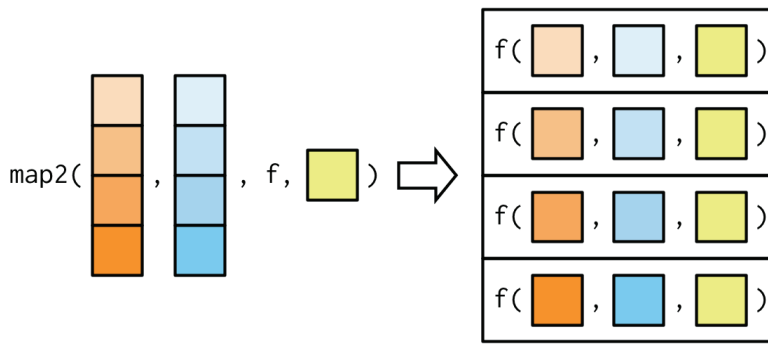


Рис. 9.8 Использование дополнительных аргументов в функции группы map2

Базовая реализация функции map2() также является достаточно простой, и она очень похожа на реализацию функции map(). Здесь вместо выполнения итераций по одному вектору мы проходим сразу по обоим параллельно:

```
simple_map2 <- function(x, y, f, ...) {
    out <- vector("list", length(xs))
    for (i in seq_along(x)) {
        out[[i]] <- f(x[[i]], y[[i]], ...)
    }
    out
}
```

Одно из основных отличий действительной реализации функции `map2()` от представленной выше упрощенной версии заключается в том, что в ней используются правила переписывания для входных векторов, чтобы они имели одну длину, что показано на рис. 9.9.

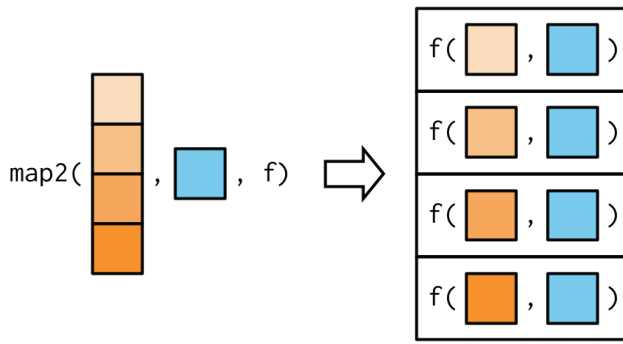


Рис. 9.9 Правило переписывания, примененное ко второму аргументу

Иными словами, вызов вида `map2(x, y, f)` будет при необходимости автоматически вести себя как `map(x, f, y)`. Это может быть полезно при написании функций. В скриптах вы в основном будете использовать наиболее простую версию напрямую.

Примечание. В базовом R ближайшим аналогом функции `map2()` является `Map()`, которую мы обсудим в разделе 9.4.5.

9.4.3 Никакого вывода: функция walk() и другие

В большинстве случаев мы обращаемся к функциям, чтобы как-то использовать возвращаемое значение, так что было бы вполне разумно сохранить значение на выходе функции `map()` для дальнейшего анализа. В то же время есть функции, которые вызываются исключительно ради побочных эффектов (например, `cat()`, `write.csv()` или `ggsave()`), и сохранять их выход не имеет никакого смысла. Давайте рассмотрим простой пример, в котором приветственное сообщение отображается при помощи функции `cat()`. Эта функция возвращает `NULL`, так что в процессе своей работы (создания приветствий) функция `map()` также возвращает список `list(NULL, NULL)`.

```
welcome <- function(x) {
    cat("Welcome ", x, "!\n", sep = "")
}
names <- c("Hadley", "Jenny")

# Помимо создания приветствий, функция также
# выводит значения, возвращаемые функцией cat()
```

```
map(names, welcome)
#> Welcome Hadley!
#> Welcome Jenny!
#> [[1]]
#> NULL
#>
#> [[2]]
#> NULL
```

Вы можете решить эту проблему, присвоив результат функции `map()` переменной, которую вы никогда не будете использовать и которая сделает ваш код грязным. В качестве альтернативы в пакете `purrr` присутствует семейство функций `walk()`, игнорирующих результат работы функции `.f` и невидимо возвращающих аргумент `.x`[1].

```
walk(names, welcome)
#> Welcome Hadley!
#> Welcome Jenny!
```

На рис. 9.10, на котором я попытался визуально запечатлеть разницу в поведении функций `walk()` и `map()`, показано, что вывод внутренней функции является эфемерным, а входные данные возвращаются в режиме невидимости.

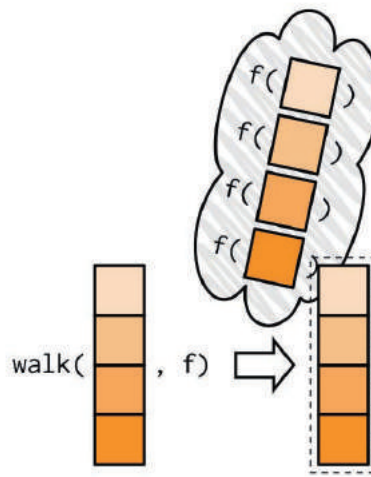


Рис. 9.10 Работа функции `walk()`

[1] Если кратко, невидимые значения выводятся только тогда, когда вы явно этого требуете. Это делает их идеально подходящими для функций, вызываемых исключительно ради побочных эффектов, поскольку позволяет по умолчанию игнорировать их выход с возможностью явного перехвата. В разделе 6.7.2 мы говорили об этом подробно.

Одной из наиболее употребимых разновидностей функции `walk()` является ее парная версия `walk2()` (рис. 9.11), поскольку чаще всего в виде побочного эффекта используется операция записи на диск, требующая пары значений (объект и путь для его сохранения).

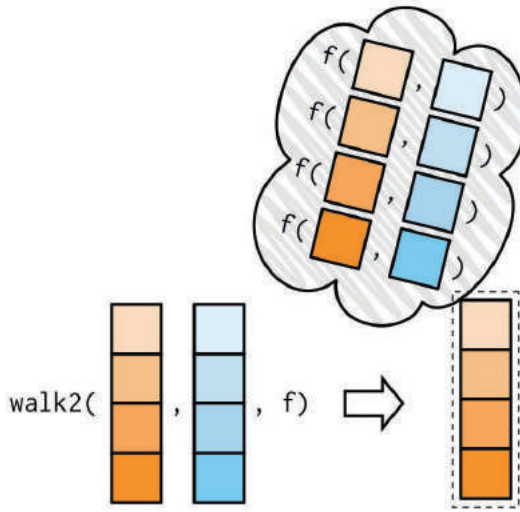


Рис. 9.11 Работа функции `walk2()`

Представьте, что у вас есть список датафреймов (который я здесь создам с помощью функции `split()`) и вам необходимо каждый из них сохранить в отдельный файл CSV. Это очень легко сделать, воспользовавшись функцией `walk2()`:

```
temp <- tempfile()
dir.create(temp)

cyls <- split(mtcars, mtcars$cyl)
paths <- file.path(temp, paste0("cyl-", names(cyls), ".csv"))
walk2(cyls, paths, write.csv)

dir(temp)
#> [1] "cyl-4.csv" "cyl-6.csv" "cyl-8.csv"
```

Здесь вызов функции `walk2()` эквивалентен следующей последовательности вызовов: `write.csv(cyls[[1]], paths[[1]])`, `write.csv(cyls[[2]], paths[[2]])`, `write.csv(cyls[[3]], paths[[3]])`.

Примечание. В базовом R не существует прямых аналогов функции `walk()`. Вам придется либо оборачивать результат функции `lapply()` в `invisible()`, либо сохранять его в отдельную переменную, которую вы никогда не будете использовать.

9.4.4 Итерации по значениям и индексам

Существует три основных способа пройти по вектору с помощью цикла `for`:

- пройти по элементам: `for (x in xs)`;
- пройти по числовым индексам: `for (i in seq_along(xs))`;
- пройти по именам: `for (nm in names(xs))`.

Первый способ аналогичен применению функций семейства `map()`. Второй и третий способы похожи на функции группы `imap()`, позволяющие одновременно проходить по значениям и индексам вектора одновременно.

Функция `imap()` похожа на функцию `map2()` в том отношении, что ваша функция `.f` вызывается с двумя аргументами, но здесь они оба происходят от вектора. Вызов `imap(x, f)` эквивалентен записи `map2(x, names(x), f)`, если вектор `x` именованный, и `map2(x, seq_along(x), f)` в противном случае. Функция `imap()` бывает полезна при создании меток:

```
imap_chr(iris, ~ paste0("The first value of ", .y, " is ", .x[[1]]))
#>                              Sepal.Length
#> "The first value of Sepal.Length is 5.1"
#>                               Sepal.Width
#>  "The first value of Sepal.Width is 3.5"
#>                              Petal.Length
#> "The first value of Petal.Length is 1.4"
#>                               Petal.Width
#>  "The first value of Petal.Width is 0.2"
#>                                   Species
#>   "The first value of Species is setosa"
```

Если вектор неименованный, вторым аргументом будет индекс:

```
x <- map(1:6, ~ sample(1000, 10))
imap_chr(x, ~ paste0("The highest value of ", .y, " is ", max(.x)))
#> [1] "The highest value of 1 is 885" "The highest value of 2 is 808"
#> [3] "The highest value of 3 is 942" "The highest value of 4 is 966"
#> [5] "The highest value of 5 is 857" "The highest value of 6 is 671"
```

Функция `imap()` может стать незаменимым помощником при работе со значениями вектора одновременно с их позициями.

9.4.5 Любое количество входов: функция pmap() и другие

Раз у нас есть функции `map()` и `map2()`, вы вполне могли ожидать, что список продолжится: `map3()`, `map4()`, `map5()` и т. д. Но где тогда остановиться? Вместо обобщения функции `map2()` для произвольного количества аргументов в пакете `purrr` присутствует отдельная функция `pmap()`, принимающая на вход список, содержащий любое количество аргументов. В большинстве случаев это будет список векторов одинаковой длины, т. е. нечто, похожее на да-

тафрейм. На рис. 9.12 я подчеркнул эту особенность, изобразив вход в виде, похожем на датафрейм.

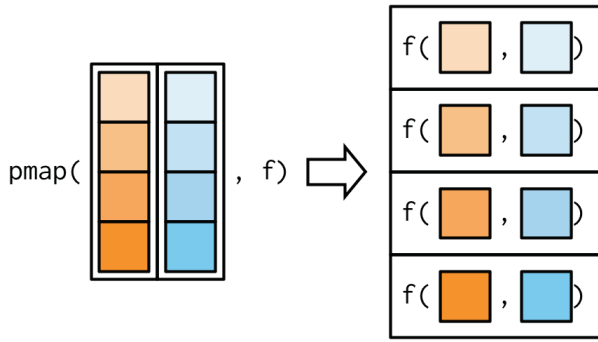


Рис. 9.12 Работа функции `pmap()`

Между функциями `map2()` и `pmap()` есть определенное сходство: выражение `map2(x, y, f)` эквивалентно `pmap(list(x, y), f)`. Таким образом, с помощью функции `pmap()` мы можем записать инструкцию `map2_dbl(xs, ws, weighted.mean)` следующим образом:

```
pmap_dbl(list(xs, ws), weighted.mean)
#> [1]    NA 0.451 0.603 0.452 0.563 0.510 0.342 0.464
```

Как и прежде, изменяющиеся аргументы располагаются перед функцией `.f` (хотя в данном случае они должны быть упакованы в список), а постоянные аргументы следуют за ней.

```
pmap_dbl(list(xs, ws), weighted.mean, na.rm = TRUE)
#> [1] 0.504 0.451 0.603 0.452 0.563 0.510 0.342 0.464
```

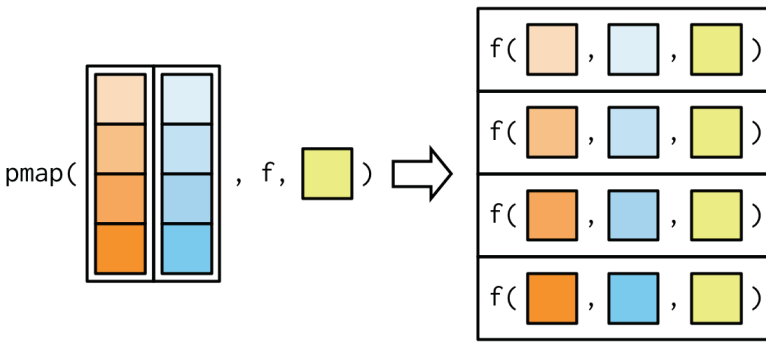


Рис. 9.13 Работа функции `pmap()` в присутствии дополнительных аргументов

Существенное отличие функции `pmap()` от других функций `map()` состоит в том, что она обеспечивает гораздо лучший контроль за аргументами из-за

возможности именовать компоненты списка. Возвращаясь к нашему примеру из раздела 9.2.5, где нам нужно было варьировать аргумент `trim`, можно было бы воспользоваться функцией `pmap()`, как показано ниже:

```
trims <- c(0, 0.1, 0.2, 0.5)
x <- rcauchy(1000)

pmap_dbl(list(trim = trims), mean, x = x)
#> [1] -6.6754  0.0192  0.0228  0.0151
```

Мне кажется хорошей практикой именование компонентов списка, позволяющее значительно облегчить чтение кода в дальнейшем.

Функцию `pmap()` бывает удобно использовать совместно с датафреймами. Создать датафрейм с его описанием по строкам, а не по колонкам, как обычно, можно с помощью функции `tibble::tribble()`. В этом случае вы можете думать о параметрах функции как о датафрейме (рис. 9.14), что является очень мощным шаблоном. В приведенном ниже примере показано, как можно вычислить случайные значения из равномерного распределения, согласующиеся с заданными с помощью датафрейма параметрами:

```
params <- tibble::tribble(
  ~ n, ~ min, ~ max,
   1L,     0,     1,
   2L,    10,   100,
   3L,   100,  1000
)

pmap(params, runif)
#> [[1]]
#> [1] 0.718
#>
#> [[2]]
#> [1] 19.5 39.9
#>
#> [[3]]
#> [1] 535 476 231
```

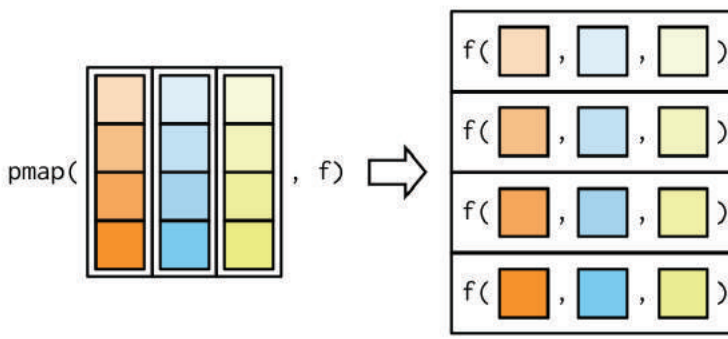


Рис. 9.14 Работа функции `pmap()` с параметрами в виде датафрейма

Здесь важную роль играют имена колонок. Я выбрал их таким образом, чтобы они совпадали с именами аргументов функции `runif()`. В результате выражение `pmap(params, runif)` стало полным эквивалентом `runif(n = 1L, min = 0, max = 1)`, `runif(n = 2, min = 10, max = 100)`, `runif(n = 3L, min = 100, max = 1000)`. Если у вас уже есть готовый датафрейм, имена в котором не совпадают с вашими требованиями, вы можете воспользоваться функцией `dplyr::rename()` или ей подобной, чтобы переименовать столбцы.

Примечание. В базовом R присутствует два аналога функций семейства `pmap()`: это функция `Map()` и `mapply()`. И у них обеих есть серьезные недостатки:

- функция `Map()` является векторизованной по всем своим аргументам, так что вы не сможете передать ей аргументы, которые не должны меняться;
- функция `mapply()` является многомерной версией функции `sapply()`. По сути, она берет вывод функции `Map()` и упрощает его, если есть такая возможность. Это влечет за собой проблемы, присущие функции `sapply()`. У функции `vapply()` не существует многомерного аналога.

9.4.6 Упражнения

1. Опишите результат работы функции `modify(mtcars, 1)`.
2. Перепишите показанный ниже код таким образом, чтобы в нем использовалась функция `iwalk()` вместо `walk2()`. Какие у этой функции есть преимущества и недостатки?

```
cyls <- split(mtcars, mtcars$cyl)
paths <- file.path(temp, paste0("cyl-", names(cyls), ".csv"))
walk2(cyls, paths, write.csv)
```

3. Поясните, как следующий код преобразует датафрейм с использованием функций, сохраненных в списке:

```
trans <- list(
    disp = function(x) x * 0.0163871,
    am = function(x) factor(x, labels = c("auto", "manual"))
)

nm <- names(trans)
mtcars[nm] <- map2(trans, mtcars[nm], function(f, var) f(var))
```

   Сравните и приведите отличия подхода с использованием функции `map2()` и показанного ниже – с использованием `map()`:

```
mtcars[vars] <- map(vars, ~ trans[[.x]](mtcars[[.x]]))
```

4. Что возвращает функция `write.csv()`? То есть что произойдет, если использовать ее совместно с функцией `map2()`, а не `walk2()`?

9.5 Семейство функций reduce()

Еще одним важным семейством функций, помимо `map()`, является семейство `reduce()`. В этой группе намного меньше функций, чем в группе `map()`, – по сути, они делятся на два основных подвида, – и применяются они реже. Но в целом семейство `reduce()` реализует очень важные идеи, лежащие в основе фреймворка `map-reduce`, часто используемого для обработки больших наборов данных.

9.5.1 Основы

Функция `reduce()` принимает на вход вектор длины *n*, а возвращает вектор длины *1*, получаемый в результате последовательного применения переданной функции к парам элементов. Фактически вызов функции `reduce(1:4, f)` равносилен записи `f(f(f(1, 2), 3), 4)`, что проиллюстрировано на рис. 9.15.

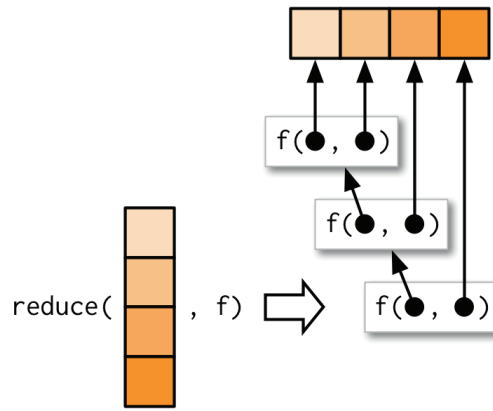


Рис. 9.15 Схематическое выполнение функции `reduce()`

С помощью функции `reduce()` можно удобно обобщить работу бинарных функций (принимающих два входных аргумента) для использования с произвольным количеством аргументов. Представьте, что у вас есть список, состоящий из числовых векторов, и вам нужно найти значения, присутствующие во всех этих векторах. Для начала сгенерируем некие исходные данные:

```
l <- map(1:4, ~ sample(1:10, 15, replace = T))
str(l)
#> List of 4
#>  $ : int [1:15] 7 5 9 7 9 9 5 10 5 5 ...
#>  $ : int [1:15] 6 3 6 10 3 4 4 2 9 9 ...
#>  $ : int [1:15] 5 3 4 6 1 1 9 9 6 8 ...
#>  $ : int [1:15] 4 2 6 6 8 5 10 6 7 1 ...
```

Для решения поставленной задачи мы можем последовательно применить функцию `intersect()`, как показано ниже:

```
out <- l[[1]]
out <- intersect(out, l[[2]])
out <- intersect(out, l[[3]])
out <- intersect(out, l[[4]])
out
#> [1] 5 1
```

Функция `reduce()` позволяет автоматизировать этот процесс, так что мы можем использовать следующий простой вариант:

```
reduce(l, intersect)
#> [1] 5 1
```

Мы можем использовать похожую схему для извлечения элементов, встречающихся хотя бы в одном векторе. Для этого достаточно поменять функцию `intersect()` на `union()`:

```
reduce(l, union)
#> [1] 7 5 9 10 1 6 3 4 2 8
```

Как и в случае с функцией `map()`, в функцию `reduce()` вы также можете передавать дополнительные аргументы. Функции `intersect()` и `union()` не предусматривают передачу дополнительных параметров, так что я не могу продемонстрировать эту возможность на предыдущем примере, но принцип здесь очень прост, он показан на рис. 9.16.

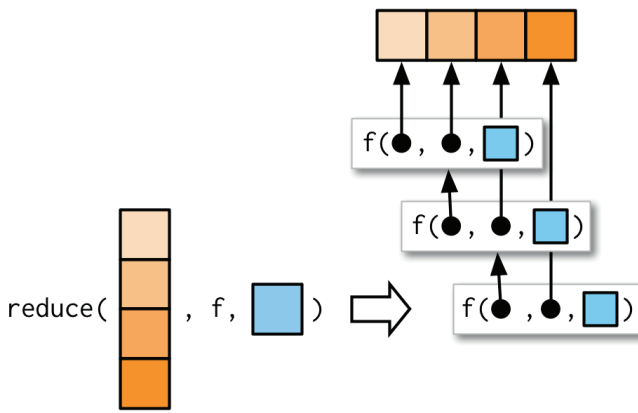


Рис. 9.16 Схематическое выполнение функции `reduce()` с дополнительными параметрами

Как обычно, действия, выполняемые функцией `reduce()`, можно свести к простому циклу `for`, показанному ниже:

```
simple_reduce <- function(x, f) {
    out <- x[[1]]
    for (i in seq(2, length(x))) {
        out <- f(out, x[[i]])
    }
    out
}
```

Примечание. В базовом R присутствует аналог этой функции, который записывается так: `Reduce()`. Обратите внимание, что порядок следования аргументов в этой функции другой: сначала передается функция, а затем вектор, без возможности передачи дополнительных аргументов.

9.5.2 Accumulate

Первой разновидностью функции `reduce()` является функция `accumulate()`. Эта функция полезна тем, что позволяет понять, как работает `reduce`, поскольку возвращает не только итоговый результат, но и все промежуточные:

```
accumulate(l, intersect)
#> [[1]]
#> [1] 7 5 9 7 9 9 5 10 5 5 5 10 9 9 1
#>
#> [[2]]
#> [1] 5 9 10 1
#>
#> [[3]]
#> [1] 5 9 1
#>
#> [[4]]
#> [1] 5 1
```

Также для лучшего понимания концепции `reduce` можно подумать о функции `sum()`: выражение `sum(x)` эквивалентно `x[[1]] + x[[2]] + x[[3]] + ...`, т. е. `reduce(x, `+`)`. А функция `accumulate(x, `+`)` в этом случае будет выводить накопительную сумму:

```
x <- c(4, 3, 10)
reduce(x, `+`)
#> [1] 17

accumulate(x, `+`)
#> [1] 4 7 17
```

9.5.3 Выходные типы

Возвращаясь к примеру, показанному выше (с оператором +), что должна вернуть функция `reduce()`, если переданный ей аргумент `x` имеет длину 1 или 0? В первом случае в отсутствие дополнительных аргументов функция `reduce()` просто вернет входной параметр `x`:

```
reduce(1, `+`)
#> [1] 1
```

Таким образом, функция `reduce()` не проверяет входной аргумент на валидность:

```
reduce("a", `+`)
#> [1] "a"
```

А что, если передать функции аргумент нулевой длины? Мы получим ошибку, говорящую о том, что необходимо использовать аргумент `.init`:

```
reduce(integer(), `+`)
#> Error: `.x` is empty, and no `.init` supplied
```

Каким тут должен быть аргумент `.init`? Чтобы это понять, давайте посмотрим на рис. 9.17, что происходит при передаче аргумента `.init`.

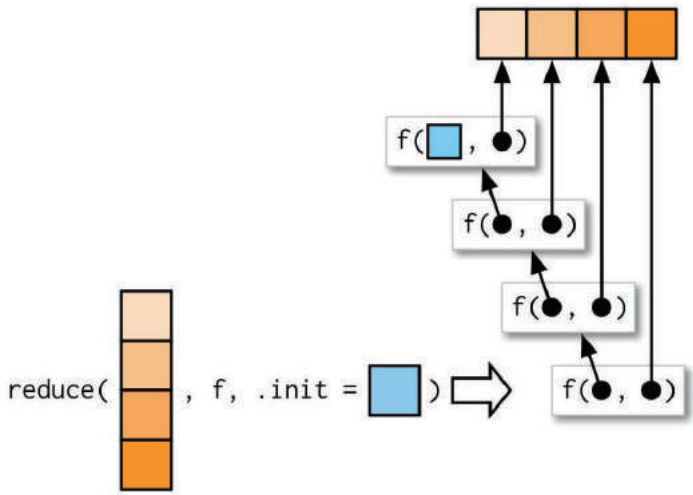


Рис. 9.17 Схематическое выполнение функции `reduce()` с аргументом `.init`

Получается, если вызвать функцию `reduce(1, `+`, init)`, результатом будет `1 + init`. Мы же знаем, что в итоге получаем единицу, а это означает, что значение параметра `init` равно нулю:

```
reduce(integer(), `+`, .init = 0)
#> [1] 0
```

Это также обеспечивает гарантию проверки валидности входных параметров единичной длины на совместимость с используемой функцией:

```
reduce("a", `+`, .init = 0)
#> Error in .x + .y: non-numeric argument to binary operator
```

Если говорить алгебраическим языком, 0 здесь выбирается из соображений тождественности выражений для операции сложения: если прибавить 0 к любому числу, это число не изменится. В R этот же принцип применяется для определения того, что должна возвращать агрегирующая функция применительно к аргументу нулевой длины:

```
sum(integer()) # x + 0 = x
#> [1] 0
prod(integer()) # x * 1 = x
#> [1] 1
min(integer()) # min(x, Inf) = x
#> [1] Inf
max(integer()) # max(x, -Inf) = x
#> [1] -Inf
```

Если вы применяете `reduce()` в функции, то всегда должны использовать параметр `.init`. Всегда думайте о том, что должна возвращать ваша функция при передаче ей вектора длины 1 или 0, и проверяйте ее корректность для этих частных случаев.

9.5.4 Множественные входы

В очень редких случаях вам может понадобиться передавать в редуцирующую функцию сразу два аргумента. К примеру, у вас может быть список, состоящий из датафреймов, которые вы хотите объединить вместе, а переменная, используемая для объединения, должна варьироваться от элемента к элементу. Это весьма специфический сценарий, так что вам не стоит уделять данному аспекту много времени. Но знать про существование функции `reduce2()` все же надо.

Длина второго аргумента может варьироваться в зависимости от того, используется параметр `.init` или нет. Если в векторе `x` находится четыре элемента, функция `f` будет вызвана только три раза, что видно на рис. 9.18. А при передаче параметра `.init` – четыре раза, что показано на рис. 9.19.

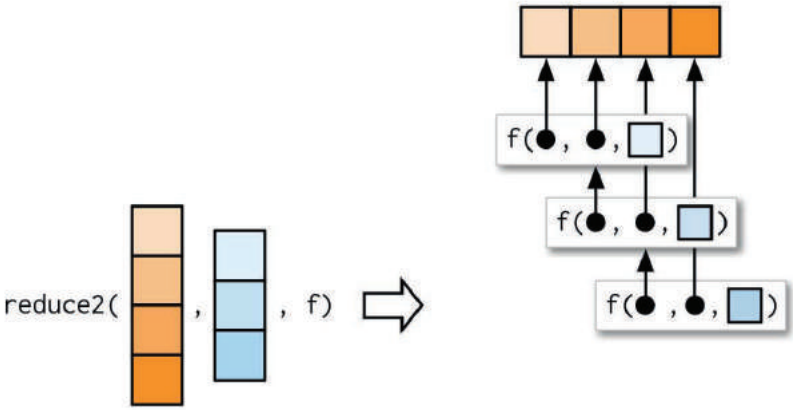


Рис. 9.18 Схематическое выполнение функции `reduce2()` без аргумента `.init`

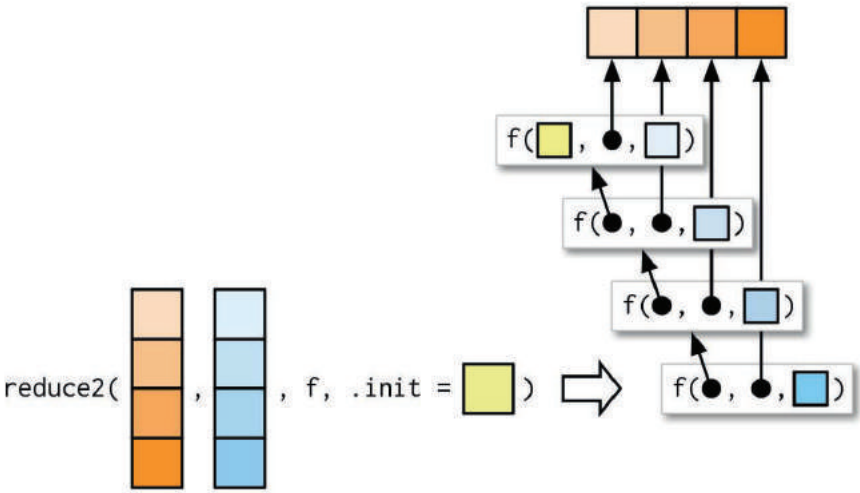


Рис. 9.19 Схематическое выполнение функции `reduce2()` с аргументом `.init`

9.5.5 Map-reduce

Вероятно, вы слышали про технологию `map-reduce`, идеи которой лежат в основе таких платформ, как Hadoop. Теперь вы можете оценить, насколько простой и мощной является эта концепция, сочетающая в себе операции `map` и `reduce`. Разница применительно к работе с большими данными состоит в том, что информация может быть физически распределена по разным компьютерам. На каждом компьютере выполняется операция `map` с имеющимися данными, после чего результат передается на другую машину-координатор, которая с помощью операции `reduce` вычисляет итог.

В качестве простого примера можно представить вычисление среднего значения для очень большого вектора – настолько, что его приходится

хранить по частям на разных компьютерах. Вы могли бы каждому из этих компьютеров поручить вычисление суммы и количества имеющихся на них элементов, после чего передать полученные данные координатору, который легко рассчитает среднее, сложив все суммы и поделив на общее количество элементов.

9.6 Предикаты-функционалы

Предикатом (predicate) называется функция, возвращающая значение `TRUE` или `FALSE`. Примерами таких функций могут быть `is.character()`, `is.null()` или `all()`. Мы говорим, что предикат согласуется с вектором, если он возвращает `TRUE`.

9.6.1 Основы

Предикаты-функционалы (predicate functional) применяют предикат к каждому элементу вектора. В пакете `purrr` представлено сразу шесть таких функций, которые можно разделить на три пары:

- функция `some(.x, .p)` возвращает `TRUE`, если *хотя бы один* элемент согласуется с предикатом, а функция `every(.x, .p)` – если *все* элементы согласуются. Эти функции похожи на `any(map_lgl(.x, .p))` и `all(map_lgl(.x, .p))`, но они завершаются раньше: функция `some()` возвращает `TRUE` при обнаружении первого же согласования, а функция `every()` возвращает `FALSE` при первом же рассогласовании;
- функция `detect(.x, .p)` возвращает *значение* первого согласующегося элемента, а функция `detect_index(.x, .p)` – его *позицию*;
- функция `keep(.x, .p)` оставляет в векторе только согласующиеся элементы, а функция `discard(.x, .p)`, наоборот, отбрасывает их.

В следующем примере показано, как можно использовать эти функционалы применительно к датафрейму:

```
df <- data.frame(x = 1:3, y = c("a", "b", "c"))
detect(df, is.factor)
#> [1] a b c
#> Levels: a b c
detect_index(df, is.factor)
#> [1] 2

str(keep(df, is.factor))
#> 'data.frame':    3 obs. of  1 variable:
#>  $ y: Factor w/ 3 levels "a","b","c": 1 2 3
str(discard(df, is.factor))
#> 'data.frame':    3 obs. of  1 variable:
#>  $ x: int  1 2 3
```

9.6.2 Вариации функции map()

Функции `map()` и `modify()` располагают разновидностями, позволяющими передавать предикат, чтобы преобразования выполнялись только для тех элементов `.x`, для которых `.p` возвращает значение `TRUE`.

```
df <- data.frame(
    num1 = c(0, 10, 20),
    num2 = c(5, 6, 7),
    chr1 = c("a", "b", "c"),
    stringsAsFactors = FALSE
)

str(map_if(df, is.numeric, mean))
#> List of 3
#>  $ num1: num 10
#>  $ num2: num 6
#>  $ chr1: chr [1:3] "a" "b" "c"
str(modify_if(df, is.numeric, mean))
#> 'data.frame':    3 obs. of  3 variables:
#>  $ num1: num  10 10 10
#>  $ num2: num  6 6 6
#>  $ chr1: chr  "a" "b" "c"
str(map(keep(df, is.numeric), mean))
#> List of 2
#>  $ num1: num 10
#>  $ num2: num 6
```

9.6.3 Упражнения

1. Почему функция `is.na()` не является предикатом? Какая базовая функция R лучше всего подходит на роль предикатной версии этой функции?
2. Функция `simple_reduce()` имеет определенные проблемы, когда длина аргумента `x` равна 1 или 0. Поясните суть этих проблем и предложите их решение.

```
simple_reduce <- function(x, f) {
    out <- x[[1]]
    for (i in seq(2, length(x))) {
        out <- f(out, x[[i]])
    }
    out
}
```

3. Реализуйте функцию `span()` из языка `Haskell`: для заданного аргумента `x` и предиката-функционала `f` функция `span(x, f)` возвращает расположение самой длинной последовательности элементов вектора, для которых `f` возвращает `TRUE`. Подсказка: вам может помочь функция `rle()`.

4. Напишите функцию `arg_max()`. На вход она должна принимать функцию и вектор и возвращать элементы вектора, для которых функция вернет наивысшее значение. К примеру, вызов `arg_max(-10:5, function(x) x ^ 2)` должен возвращать –10, а `arg_max(-5:5, function(x) x ^ 2)` должен возвращать `c(-5, 5)`. Реализуйте также функцию `arg_min()`.
5. Приведенная ниже функция масштабирует вектор таким образом, чтобы его элементы умещались в диапазон значений [0, 1]. Как бы вы применили эту функцию ко всем колонкам в датафрейме? А только к числовым колонкам?

```
scale01 <- function(x) {
    rng <- range(x, na.rm = TRUE)
    (x - rng[1]) / (rng[2] - rng[1])
}
```

9.7 Функционалы базового R

В завершение этой главы я приведу некоторые функционалы из базового R, не входящие в семейства `map()`, `reduce()` и не являющиеся предикатами, а значит, не имеющие аналогов в пакете `purrr`. Не сказать, чтобы они совсем были не важны, но они чаще применяются в математике и статистике, а при анализе данных бывают не столь полезны.

9.7.1 Матрицы и массивы

Функция `map()` и ее собратья специализируются на работе с одномерными векторами. В то же время функция `base::apply()` призвана работать с векторами, обладающими двумя и более измерениями, т. е. с матрицами и массивами. Можно представить работу функции `apply()` в виде операции, агрегирующей матрицу или массив путем свертывания каждой строки или колонки в одно значение. У этой функции есть четыре аргумента:

- `X` – матрица или массив для агрегирования;
- `MARGIN` – целочисленный вектор, определяющий измерения, по которым необходимо выполнить агрегацию, 1 = строки, 2 = колонки и т. д. Термин `MARGIN` появился в свете размышлений о пределах совместного распределения;
- `FUN` – агрегирующая функция;
- `...` – другие аргументы, переданные в функцию `FUN`.

Типичное применение функции `apply()` может выглядеть так:

```
a2d <- matrix(1:20, nrow = 5)
apply(a2d, 1, mean)
#> [1]  8.5  9.5 10.5 11.5 12.5
```

```
apply(a2d, 2, mean)
#> [1]  3  8 13 18
```

В качестве параметра MARGIN допустимо передавать несколько измерений, что бывает полезно при работе с многомерными массивами:

```
a3d <- array(1:24, c(2, 3, 4))
apply(a3d, 1, mean)
#> [1] 12 13
apply(a3d, c(1, 2), mean)
#>      [,1] [,2] [,3]
#> [1,]   10   12   14
#> [2,]   11   13   15
```

При использовании функции apply() стоит помнить о двух важных вещах:

- как и в случае с функцией base::sapply(), вы не управляете типом результирующих данных, которые будут автоматически преобразованы в список, матрицу или вектор. Но, как правило, вы будете использовать функцию apply() применительно к числовым массивам с числовыми агрегирующими функциями, так что подобные проблемы коснутся вас с меньшей вероятностью по сравнению с использованием функции sapply();
- функция apply() не является идемпотентной в том смысле, что если в качестве агрегирующей функции используется оператор identity, выход не всегда будет таким же, как вход:

```
a1 <- apply(a2d, 1, identity)
identical(a2d, a1)
#> [1] FALSE

a2 <- apply(a2d, 2, identity)
identical(a2d, a2)
#> [1] TRUE
```

- никогда не используйте функцию apply() совместно с датафреймами. В этом случае будет выполнено принудительное преобразование датафрейма в матрицу, что приведет к неожиданным результатам, если в датафрейме содержатся не только числовые данные.

```
df <- data.frame(x = 1:3, y = c("a", "b", "c"))
apply(df, 2, mean)
#> Warning in mean.default(newX[, i], ...): argument is not numeric or
#> logical: returning NA
#> Warning in mean.default(newX[, i], ...): argument is not numeric or
#> logical: returning NA
#>  x  y
#> NA NA
```

9.7.2 Математическое применение

Функционалы очень часто используются в области математики. Пределы, максимумы, корни уравнений (множество точек, при которых $f(x) = 0$) и определенные интегралы – это все примеры функционалов: они принимают на вход функцию и возвращают число (или вектор чисел). На первый взгляд кажется, что применительно к этим функциям не стоит проблема избавления от циклов, но если задуматься, все они так или иначе используют алгоритмы, связанные с итерациями.

В базовом R представлено сразу несколько полезных функционалов, в том числе:

- `integrate()` – определяет площадь под кривой, определенной функцией $f()$;
- `uniroot()` – находит точки, в которых $f() = 0$;
- `optimise()` – определяет минимальное и максимальное значения функции $f()$.

В приведенном ниже примере показано, как могут быть использованы функционалы совместно с простой функцией `sin()`:

```
integrate(sin, 0, pi)
#> 2 with absolute error < 2.2e-14
str(uniroot(sin, pi * c(1 / 2, 3 / 2)))
#> List of 5
#>  $ root      : num 3.14
#>  $ f.root    : num 1.22e-16
#>  $ iter      : int 2
#>  $ init.it   : int NA
#>  $ estim.prec: num 6.1e-05
str(optimise(sin, c(0, 2 * pi)))
#> List of 2
#>  $ minimum  : num 4.71
#>  $ objective: num -1
str(optimise(sin, c(0, pi), maximum = TRUE))
#> List of 2
#>  $ maximum  : num 1.57
#>  $ objective: num 1
```

9.7.3 Упражнения

1. Как функция `apply()` располагает выходные данные? Ознакомьтесь с документацией и проведите пару экспериментов.
2. Что делают функции `eapply()` и `rapply()`? Есть ли в пакете `purrr` аналоги?

10

Фабрики функций

10.1 Введение

Фабрикой функций (function factory) называется функция, производящая на свет другую функцию. Вот вам простой пример: мы можем использовать фабрику функций `power1()` для производства двух дочерних функций: `square()` и `cube()`:

```
power1 <- function(exp) {
    function(x) {
        x ^ exp
    }
}

square <- power1(2)
cube <- power1(3)
```

Не беспокойтесь, если вы сразу не ухватили суть, до конца главы все приобретенные знания улягутся! Мы будем называть функции `square()` и `cube()` *произведенными* (manufactured function), но этот термин можно использовать лишь для облегчения коммуникации с другими разработчиками R, поскольку с точки зрения самого языка между этими функциями и любыми другими нет никаких различий.

```
square(3)
#> [1] 9
cube(3)
#> [1] 27
```

Ранее вы уже изучили все предпосылки, благодаря которым стало доступно создание фабрик функций:

- в разделе 6.2.3 вы узнали про функции первого класса в R. В этом языке программирования вы связываете функцию с именем так же точно, как и любой другой объект, – при помощи оператора `<-`;
- в разделе 7.4.2 мы поговорили о том, что функция захватывает, или замыкает, окружение, в котором она создана;

- далее, в разделе 7.4.4, вы узнали, что при каждом вызове функции создается окружение выполнения. Это окружение обычно существует очень недолго, а в данном случае оно становится еще и замыкающим окружением для произведенной функции.

В данной главе вы увидите, как не вполне очевидная комбинация этих трех факторов может способствовать созданию фабрик функций. Мы также рассмотрим несколько примеров использования этой техники в области визуализации и статистики.

Среди всех трех инструментов функционального программирования (функционалы, фабрики функций и функциональные операторы) фабрики функций являются наименее популярными. Обычно их применение ведет не к упрощению всего кода в целом, а, скорее, к разделению кода на отдельные более понятные блоки. Также фабрики функций можно рассматривать в качестве строительных блоков для очень полезных функциональных операторов, о которых мы будем говорить в главе 11.

Структура главы

- В разделе 10.2 вы узнаете, как работают фабрики функций и как в них реализованы идеи создания окружений и поиска в области видимости. Вы также посмотрите, как можно использовать фабрики функций с целью выделения памяти для функций, что позволяет хранить данные между вызовами.
- В разделе 10.3 вы увидите, как можно использовать фабрики функций на примере пакета `ggplot2`. Два примера будут посвящены тому, как `ggplot2` работает с пользовательскими фабриками функций, и один – как в этом пакете используются внутренние фабрики.
- В разделе 10.4 мы применим фабрики функций для решения трех разных задач из области статистики, а именно мы рассмотрим преобразование Бокса-Кокса, реализуем оценку максимального правдоподобия и визуализируем повторные выборки, полученные с помощью бутстрэпа.
- В разделе 10.5 вы узнаете, как можно комбинировать фабрики функций и функционалы для быстрого создания семейств функций на основе данных.

Требования

Для комфортного чтения этой главы вам необходимо хорошо понимать темы, описанные в разделах 6.2.3 (функции первого класса), 7.4.2 (окружения функций) и 7.4.4 (окружения выполнения).

Для создания фабрик функций вам понадобится только базовый R. При этом мы будем использовать пакет `rlang` (https://rlang.r-lib.org) для исследования функций, а также воспользуемся пакетами `ggplot2` (https://ggplot2.tidyverse.org) и `scales` (https://scales.r-lib.org) с целью визуализации данных.

```
library(rlang)
library(ggplot2)
library(scales)
```

10.2 Основы фабрик функций

Ключевая идея, лежащая в основе фабрик функций, может быть выражена очень просто:

> *Замыкающее окружение произведенной функции является окружением выполнения для фабрики функций.*

Всего несколько слов для описания столь обширной идеи, но не радуйтесь заранее – полное понимание этой концепции может прийти только после долгих часов практики. Этот раздел позволит вам ухватить основные составляющие данной концепции благодаря интерактивным объяснениям и диаграммам.

10.2.1 Окружения

Давайте для начала взглянем на наши произведенные ранее функции `square()` и `cube()`:

```
square
#> function(x) {
#>     x ^ exp
#>   }
#> <environment: 0x7fa43cc11b00>

cube
#> function(x) {
#>     x ^ exp
#>   }
#> <bytecode: 0x7fa43d0e1740>
#> <environment: 0x7fa439297d50>
```

Вполне очевидно, откуда поступает `x`, но где R находит значение, ассоциированное с именем `exp`? Простой вывод текста произведенных функций не дает нам это понять, поскольку здесь мы не видим никаких различий. Гораздо важнее рассмотреть содержимое замыкающих окружений. Проникнуть глубже нам позволит функция `rlang::env_print()`. Благодаря ей мы видим, что у нас есть два разных окружения, каждое из которых изначально было окружением выполнения функции `power1()`. Родителем этих окружений является замыкающее окружение `power1()`, т. е. глобальное окружение.

```
env_print(square)
#> <environment: 0x7fa43cc11b00>
#> parent: <environment: global>
#> bindings:
#>  * exp: <dbl>

env_print(cube)
#> <environment: 0x7fa439297d50>
#> parent: <environment: global>
#> bindings:
#>  * exp: <dbl>
```

Также в выводе функции `env_print()` можно заметить, что оба окружения содержат привязки для имени `exp`, но нам бы хотелось получить конкретные значения[1]. Это можно сделать, сначала получив окружение функции, а затем достав нужное значение:

```
fn_env(square)$exp
#> [1] 2

fn_env(cube)$exp
#> [1] 3
```

Именно это отличает произведенные функции от всех остальных: имена в их замыкающих окружениях привязаны к разным значениям.

10.2.2 Обозначения на диаграммах

Все эти взаимосвязи можно показать при помощи диаграмм, как на рис. 10.1.

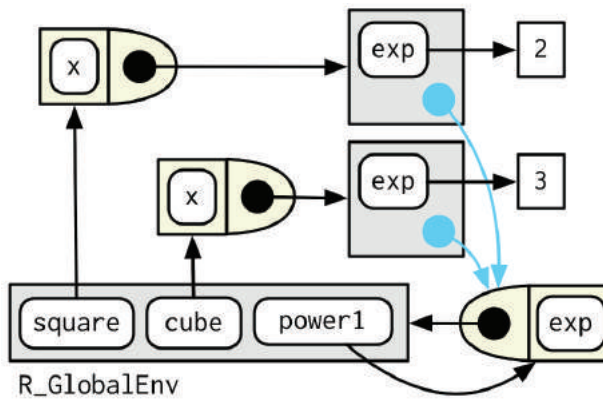


Рис. 10.1 Внутреннее устройство фабрик функций

[1] В следующих версиях функции `env_print()`, вероятно, будет расширен вывод, что позволит вам избежать этого дополнительного шага.

На приведенной диаграмме показано много всего, и далеко не все подробности так важны. Таким образом, можно упростить схему, применив два соглашения:

- любое свободное имя живет в глобальном окружении;
- любое окружение без явно указанного родителя наследуется от глобального окружения.

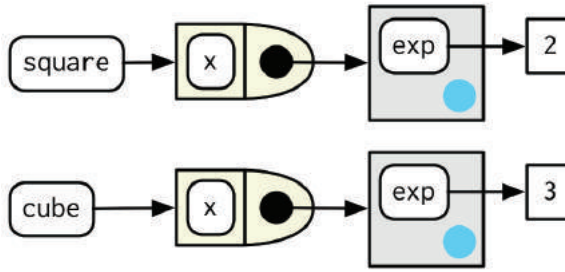


Рис. 10.2 Упрощенное внутреннее устройство фабрик функций

Схема, показанная на рис. 10.2, концентрируется на окружениях, и здесь не видно никаких прямых зависимостей между функциями `cube()` и `square()`. Дело в том, что связь таится в одинаковом для обеих функций содержании, которое не показано на диаграмме.

Теперь давайте взглянем на окружение выполнения функции `square(10)`. При вычислении формулы `x ^ exp` функция находит `x` в окружении выполнения, а `exp` – в замыкающем окружении, что проиллюстрировано на рис. 10.3.

```
square(10)
#> [1] 100
```

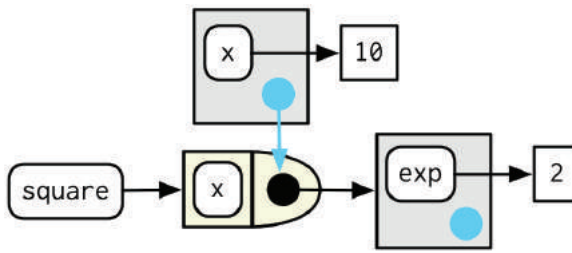


Рис. 10.3 Окружение выполнения функции `square(10)`

10.2.3 Форсирование вычислений

В нашей функции `power1()` есть небольшой баг, обусловленный отложенным вычислением. Для демонстрации проблемы давайте выполним следующий код:

```
x <- 2
square <- power1(x)
x <- 3
```

Что вернет `square(2)` в этом случае? Ожидается, что 4, а на самом деле:

```
square(2)
#> [1] 8
```

К сожалению, ожидаемого результата мы не получили по причине того, что значение переменной `x` извлекается в отложенной манере – при вызове произведенной функции `square()`, а не фабрики `power1()`. Такого рода казусы возникают всегда, когда привязка меняется между вызовами фабрики и произведенной ей функции. Да, на практике это встречается не так часто, но когда это происходит, мы получаем ошибку, которую очень трудно отловить.

Этой ошибки можно избежать, если использовать форсированное вычисление при помощи функции `force()`:

```
power2 <- function(exp) {
    force(exp)
    function(x) {
        x ^ exp
    }
}

x <- 2
square <- power2(x)
x <- 3
square(2)
#> [1] 4
```

При создании фабрик функций убедитесь, что каждый аргумент вычисляется при помощи функции `force()`, что необходимо, если аргументы используются только произведенными функциями.

10.2.4 Функции с отслеживанием состояния

Фабрики функций позволяют сохранять и поддерживать состояние между вызовами функций, что в обычных условиях трудно реализуемо из-за действия принципа чистого листа, описанного в разделе 6.4.3.

Поддержание состояния может быть реализовано благодаря тому, что:

- замыкающее окружение произведенной функции уникально и постоянно;
- в R есть оператор присваивания в родительском окружении (`<<-`), с помощью которого можно изменять привязки в замыкающем окружении.

Обычный оператор присваивания (<-) всегда создает привязку в текущем окружении, тогда как оператор присваивания <<- переопределяет существующее имя в родительском окружении.

В следующем примере, проиллюстрированном на рис. 10.4, показано, как можно объединить эти идеи при определении функции, сохраняющей информацию о том, сколько раз она была вызвана:

```
new_counter <- function() {
    i <- 0

    function() {
        i <<- i + 1
        i
    }
}

counter_one <- new_counter()
counter_two <- new_counter()
```

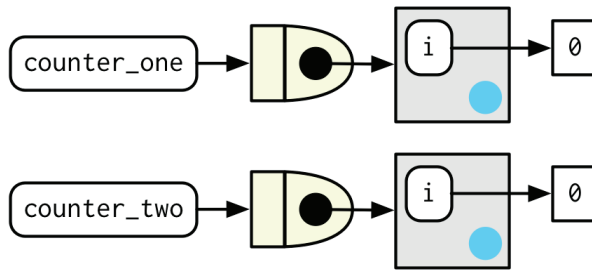


Рис. 10.4 Функции с отслеживанием состояния (до)

Инструкция `i <<- i + 1` внутри произведенной функции модифицирует значение имени `i` в ее замыкающем окружении. А поскольку все произведенные функции обладают своими независимыми замыкающими окружениями, счетчики у них тоже будут свои, что видно на рис. 10.5:

```
counter_one()
#> [1] 1
counter_one()
#> [1] 2
counter_two()
#> [1] 1
```

Функциями с отслеживанием состояния не стоит злоупотреблять. Если вы заметили, что вашим функциям приходится сохранять состояние более чем одной переменной, возможно, самое время переключиться на классы R6. Эта тема будет освещаться в главе 14.

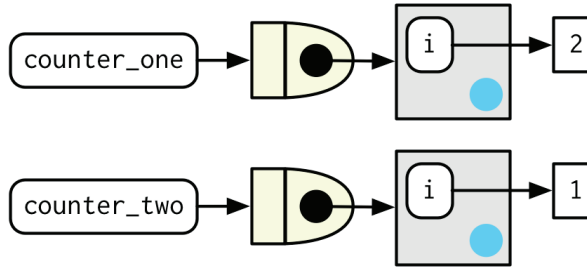


Рис. 10.5 Функции с отслеживанием состояния (после)

10.2.5 Сборка мусора

При работе с большинством функций вы можете полагаться на встроенный в R сборщик мусора, который будет тщательно подчищать все большие временные объекты, созданные внутри функций. Однако произведенные функции удерживают свое окружение выполнения, так что вам необходимо явным образом отвязывать большие временные объекты, когда они больше не нужны, с помощью функции `rm()`. Сравните размеры `g1()` и `g2()` в примере ниже:

```
f1 <- function(n) {
    x <- runif(n)
    m <- mean(x)
    function() m
}

g1 <- f1(1e6)
lobstr::obj_size(g1)
#> 8,013,120 B

f2 <- function(n) {
    x <- runif(n)
    m <- mean(x)
    rm(x)
    function() m
}

g2 <- f2(1e6)
lobstr::obj_size(g2)
#> 12,960 B
```

10.2.6 Упражнения

1. Определение функции `force()` – проще некуда:

```
force
#> function (x)
#> x
```

```
#> <bytecode: 0x7fa4381626a0>
#> <environment: namespace:base>
```

Почему лучше использовать `force(x)` вместо `x`?

2. В базовом R есть две фабрики функций `approxfun()` и `ecdf()`. Ознакомьтесь с документацией к ним и поэкспериментируйте с ними, чтобы понять, что эти функции делают и что возвращают.
3. Напишите функцию `pick()`, принимающую на вход индекс `i` в качестве аргумента и возвращающую функцию с аргументом `x`, которая извлекает подмножество из `x` с помощью `i`.

```
pick(1)(x)
# должно быть эквивалентно
x[[1]]

lapply(mtcars, pick(5))
# должно быть эквивалентно
lapply(mtcars, function(x) x[[5]])
```

4. Напишите функцию, создающую другую функцию, которая вычисляет *i*-й центральный момент (https://ru.wikipedia.org/wiki/Моменты_случайной_величины) числового вектора. Проверить ее вы можете с помощью следующего кода:

```
m1 <- moment(1)
m2 <- moment(2)

x <- runif(100)
stopifnot(all.equal(m1(x), 0))
stopifnot(all.equal(m2(x), var(x) * 99 / 100))
```

5. Что произойдет, если не использовать замыкание? Сначала подумайте, потом проверьте свою догадку с помощью кода ниже:

```
i <- 0
new_counter2 <- function() {
    i <<- i + 1
    i
}
```

6. Что будет, если вместо оператора `<<-` использовать оператор `<-`? Опять же, сначала подумайте, потом проверьте свою догадку с помощью кода ниже:

```
new_counter3 <- function() {
    i <- 0
    function() {
        i <- i + 1
        i
    }
}
```

10.3 Графические фабрики функций

Начнем исследование полезных фабрик функций с нескольких примеров из пакета `ggplot2`.

10.3.1 Метки

Одной из целей пакета `scales` (https://scales.r-lib.org) является облегчение кастомизации меток на графиках `ggplot2`. В этом пакете представлено множество функций для управления внешним видом осей и легенд. Функции форматирования[1] существенно облегчают вывод меток на осях, что показано на рис. 10.6. Структура этих функций может на первый взгляд показаться странной – все они возвращают функции, которые вы вызываете с целью форматирования чисел.

```
y <- c(12345, 123456, 1234567)
comma_format()(y)
#> [1] "12,345" "123,456" "1,234,567"

number_format(scale = 1e-3, suffix = " K")(y)
#> [1] "12 K" "123 K" "1 235 K"
```

Иными словами, здесь мы имеем дело с фабриками функций. Может показаться, что для такой простой задачи было выбрано не самое очевидное решение. На самом же деле такой подход позволил обеспечить бесшовную интеграцию с функциями масштабирования из пакета `ggplot2`, принимающими в качестве аргумента функцию:

```
df <- data.frame(x = 1, y = y)
core <- ggplot(df, aes(x, y)) +
    geom_point() +
    scale_x_continuous(breaks = 1, labels = NULL) +
    labs(x = NULL, y = NULL)

core
core + scale_y_continuous(
    labels = comma_format()
)
core + scale_y_continuous(
    labels = number_format(scale = 1e-3, suffix = " K")
)
core + scale_y_continuous(
```

[1] К сожалению, так получилось, что при именовании функций в пакете `scales` были использованы постфиксы, а не префиксы. Это произошло из-за того, что в тот момент я еще не понимал всех преимуществ автодополнения, позволяющего легко использовать похожие функции с одинаковыми префиксами.

```
    labels = scientific_format()
)
```

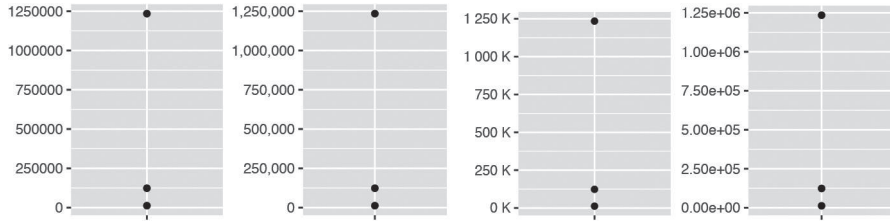


Рис. 10.6 Форматирование меток осей

10.3.2 Столбики на гистограммах

Мало кто знает, что в функции `geom_histogram()` аргумент `binwidth` может быть представлен функцией. Это может быть крайне удобно, поскольку функция вызывается единожды для каждой группы, а это означает, что на разных графиках у вас может быть своя ширина столбиков. Иным способом это реализовать нельзя.

Для иллюстрации данной идеи представим пример, в котором одинаковая ширина столбиков на разных графиках будет неприемлема.

```
# построим выборки с сильно отличающимися числами в каждой ячейке
sd <- c(1, 5, 15)
n <- 100

df <- data.frame(x = rnorm(3 * n, sd = sd), sd = rep(sd, n))

ggplot(df, aes(x)) +
    geom_histogram(binwidth = 2) +
    facet_wrap(~ sd, scales = "free_x") +
    labs(x = NULL)
```

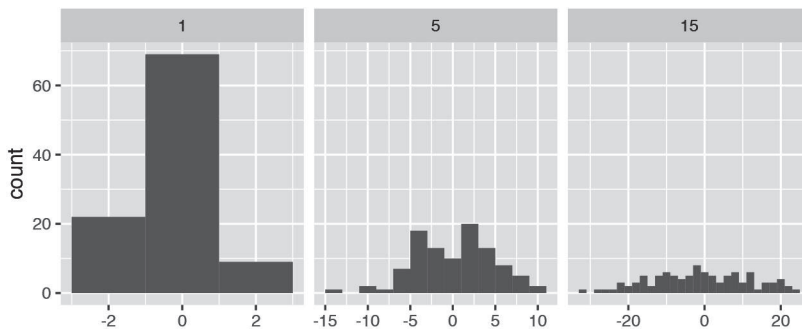


Рис. 10.7 Гистограмма с неизменной шириной столбиков

Как видно на рис. 10.7, здесь мы имеем дело с одинаковым количеством наблюдений, но с очень разной изменчивостью данных. Было бы удобно сделать варьирующуюся ширину столбиков, чтобы в каждый из них входило приблизительно одинаковое количество наблюдений. Один из способов реализовать это – использовать фабрику функций с желаемым количеством столбиков (`n`) на входе и функцией, принимающей числовой вектор и возвращающей ширину столбика, на выходе. Результат показан на рис. 10.8.

```
binwidth_bins <- function(n) {
    force(n)

    function(x) {
        (max(x) - min(x)) / n
    }
}

ggplot(df, aes(x)) +
    geom_histogram(binwidth = binwidth_bins(20)) +
    facet_wrap(~ sd, scales = "free_x") +
    labs(x = NULL)
```

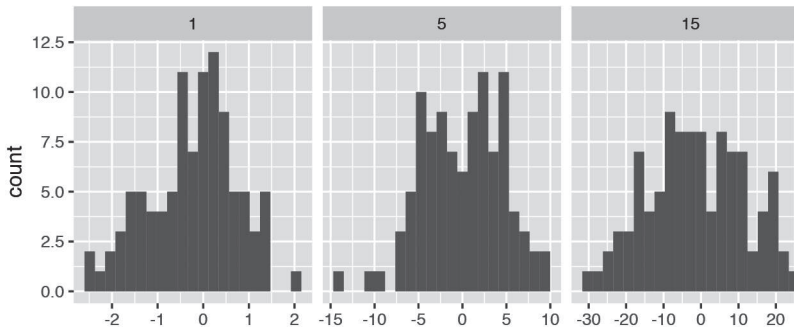


Рис. 10.8 Гистограмма с переменной шириной столбиков

Этот же шаблон можно использовать в качестве обертки для базовых функций R, осуществляющих автоматический поиск так называемой оптимальной[1] ширины столбиков, – `nclass.Sturges()`, `nclass.scott()` и `nclass.FD()`. Результат показан на рис. 10.9:

```
base_bins <- function(type) {
    fun <- switch(type,
        Sturges = nclass.Sturges,
```

[1] В пакете `ggplot2` эти функции не видны напрямую, поскольку, как мне кажется, определение оптимальности, необходимое для того, чтобы сделать задачу математически разрешимой, не совсем подходит для области исследования данных.

```
        scott = nclass.scott,
        FD = nclass.FD,
        stop("Unknown type", call. = FALSE)
    )

    function(x) {
        (max(x) - min(x)) / fun(x)
    }
}

ggplot(df, aes(x)) +
    geom_histogram(binwidth = base_bins("FD")) +
    facet_wrap(~ sd, scales = "free_x") +
    labs(x = NULL)
```

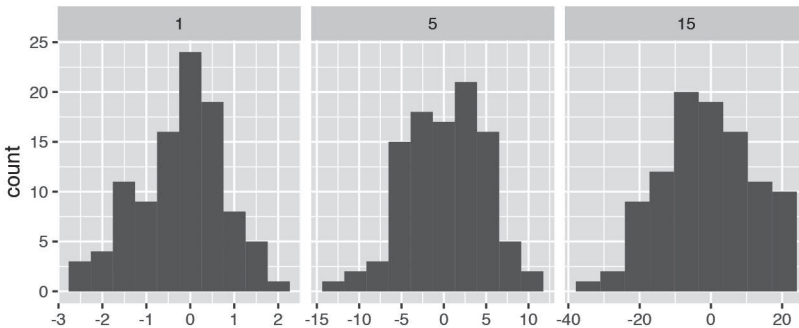


Рис. 10.9 Гистограмма с оптимальной шириной столбиков

10.3.3 ggsave()

Наконец, я хочу показать встроенную фабрику функций, применяемую в пакете `ggplot2`. Функция `ggplot2:::plot_dev()` используется в функции `ggsave()` для перехода от расширения файла (например, `png`, `jpeg` и т. д.) к функциям графического устройства (`png()`, `jpeg()`, ...). Сложность здесь состоит в том, что базовые графические устройства обладают небольшими несоответствиями, которые необходимо уладить:

- в большинстве из них первый аргумент называется `filename`, тогда как в некоторых – `file`;
- в параметрах `width` и `height` растровых графических устройств по умолчанию используются единицы измерения в виде пикселей, тогда как в векторной графике используются дюймы.

Давайте взглянем на несколько упрощенную версию функции `plot_dev()`:

```
plot_dev <- function(ext, dpi = 96) {
    force(dpi)

    switch(ext,
```

```
        eps = ,
        ps = function(path, ...) {
            grDevices::postscript(
                file = filename, ..., onefile = FALSE,
                horizontal = FALSE, paper = "special"
            )
        },
        pdf = function(filename, ...) grDevices::pdf(file = filename, ...),
        svg = function(filename, ...) svglite::svglite(file = filename, ...),
        emf = ,
        wmf = function(...) grDevices::win.metafile(...),
        png = function(...) grDevices::png(..., res = dpi, units = "in"),
        jpg = ,
        jpeg = function(...) grDevices::jpeg(..., res = dpi, units = "in"),
        bmp = function(...) grDevices::bmp(..., res = dpi, units = "in"),
        tiff = function(...) grDevices::tiff(..., res = dpi, units = "in"),
        stop("Unknown graphics extension: ", ext, call. = FALSE)
    )
}

plot_dev("pdf")
#> function(filename, ...) grDevices::pdf(file = filename, ...)
#> <bytecode: 0x7fa781166af8>
#> <environment: 0x7fa78049e698>
plot_dev("png")
#> function(...) grDevices::png(..., res = dpi, units = "in")
#> <bytecode: 0x7fa7812d2708>
#> <environment: 0x7fa780a65a40>
```

10.3.4 Упражнения

1. Сравните и противопоставьте функции `ggplot2::label_bquote()` и `scales::number_format()`.

10.4 Статистические фабрики функций

Еще более вдохновляющее применение фабрики функций нашли в области статистики, в частности в следующих методах:

- преобразование Бокса-Кокса;
- повторные выборки, полученные с помощью бутстрэпа;
- оценка максимального правдоподобия.

Все эти методы вполне можно реализовать и без фабрик функций, но мне кажется, что с их использованием эти задачи обретают весьма элегантные решения. Реализация перечисленных выше методов требует определенных познаний в области статистики, так что вы легко можете пропустить данный раздел, если это совершенно не ваша тема.

10.4.1 Преобразование Бокса-Кокса

Преобразование Бокса-Кокса (Box-Cox transformation), принадлежащее семейству степенных преобразований (https://ru.wikipedia.org/wiki/Преобразование_данных_(статистика)), часто используется для приведения ряда данных в соответствие нормальному распределению. Этот метод располагает единственным параметром λ, определяющим степень преобразования. В простейшем виде преобразование Бокса-Кокса можно реализовать в виде следующей функции с двумя аргументами:

```
boxcox1 <- function(x, lambda) {
    stopifnot(length(lambda) == 1)

    if (lambda == 0) {
        log(x)
    } else {
        (x ^ lambda - 1) / lambda
    }
}
```

Но если прописать это преобразование в виде фабрики функций, будет гораздо проще отслеживать его поведение с помощью функции `stat_function()`. Вывод показан на рис. 10.10.

```
boxcox2 <- function(lambda) {
    if (lambda == 0) {
        function(x) log(x)
    } else {
        function(x) (x ^ lambda - 1) / lambda
    }
}

stat_boxcox <- function(lambda) {
    stat_function(aes(colour = lambda), fun = boxcox2(lambda), size = 1)
}

ggplot(data.frame(x = c(0, 5)), aes(x)) +
    lapply(c(0.5, 1, 1.5), stat_boxcox) +
    scale_colour_viridis_c(limits = c(0, 1.5))

# визуально кажется, что функция log() подходит для преобразования при
# lambda = 0; с уменьшением значений  функция
# все больше приближается к логарифмическому преобразованию
ggplot(data.frame(x = c(0.01, 1)), aes(x)) +
    lapply(c(0.5, 0.25, 0.1, 0), stat_boxcox) +
    scale_colour_viridis_c(limits = c(0, 1.5))
```

В общем случае такой подход позволяет использовать преобразование Бокса-Кокса с любой унарной функцией, в которой вам не придется беспокоиться о передаче дополнительных аргументов с помощью …. Кроме того,

я считаю правильным разнесение аргументов `lambda` и `x` по разным функциям, поскольку они играют совсем разные роли.

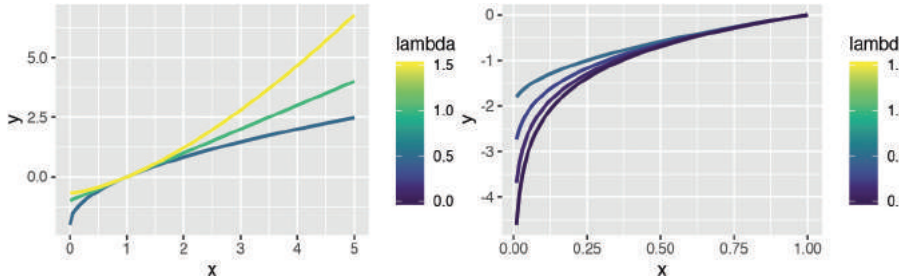


Рис. 10.10 Преобразование Бокса-Кокса

10.4.2 Генераторы повторных выборок по методу бутстрэпа

Фабрики функций отлично подходят для операций, связанных с бутстрэппингом. Вместо того чтобы думать об одном бутстрэпе (а одного всегда недостаточно!), вы можете переключиться на создание генератора бутстрэпов, т. е. функции, возвращающей свеженький набор данных при каждом обращении к ней:

```
boot_permute <- function(df, var) {
    n <- nrow(df)
    force(var)

    function() {
        col <- df[[var]]
        col[sample(n, replace = TRUE)]
    }
}

boot_mtcars1 <- boot_permute(mtcars, "mpg")
head(boot_mtcars1())
#> [1] 18.1 22.8 21.5 14.7 21.4 17.3
head(boot_mtcars1())
#> [1] 19.2 19.2 14.3 21.0 13.3 21.4
```

Преимущество использования фабрики функций становится еще более очевидным в случае с параметрическим бутстрэппингом, когда сначала вам нужно выполнить подгонку модели. Это можно сделать лишь раз при обращении к фабрике, а не делать каждый раз, когда вы хотите получить очередной бутстрэп:

```
boot_model <- function(df, formula) {
    mod <- lm(formula, data = df)
```

```
    fitted <- unname(fitted(mod))
    resid <- unname(resid(mod))
    rm(mod)

    function() {
        fitted + sample(resid)
    }
}

boot_mtcars2 <- boot_model(mtcars, mpg ~ wt)
head(boot_mtcars2())
#> [1] 23.1 24.3 23.0 19.1 19.1 16.2
head(boot_mtcars2())
#> [1] 30.2 17.4 31.3 26.1 17.8 16.7
```

Здесь я воспользовался функцией `rm(mod)`, потому что объекты линейных моделей довольно объемные (они включают полную копию матрицы модели и входные данные), а мне хочется, чтобы произведенная функция оставалась небольшой по размеру, насколько это возможно.

10.4.3 Оценка максимального правдоподобия

Целью *оценки максимального правдоподобия* (maximum likelihood estimation – MLE) является нахождение значений параметров для распределения, делающих наблюдаемые данные наиболее вероятными. Для проведения оценки максимального правдоподобия необходимо начать с функции вероятности. Возьмем, к примеру, распределение Пуассона. При известной λ мы можем вычислить вероятность получения вектора x, состоящего из $(x_1, x_2, \ldots, x_n)$, путем перемножения функций вероятности Пуассона по следующей формуле:

$$P(\lambda, x) = \prod_{i=1}^{n} \frac{\lambda^{x_i} e^{-\lambda}}{x_i!}.$$

В статистике мы почти всегда работаем с логарифмом этой функции. Логарифмирование функции представляет собой ее монотонное преобразование с сохранением важных свойств (т. е. экстремумы останутся на своих местах) и следующими преимуществами:

- логарифм превращает произведения в суммы, с которыми легче работать;
- перемножение маленьких чисел в результате дает еще более маленькое число, что делает аппроксимацию с плавающей запятой, используемую в компьютерах, менее точной.

Давайте применим логарифмическое преобразование к этой функции вероятности и упростим ее, насколько это возможно:

$$\log(P(\lambda, x)) = \prod_{i=1}^{n} \log\left(\frac{\lambda^{x_i} e^{-\lambda}}{x_i!}\right),$$

$$\log(P(\lambda, x)) = \sum_{i=1}^{n}(x_i \log(\lambda) - \lambda - \log(x_i!)),$$

$$\log(P(\lambda, x)) = \sum_{i=1}^{n} x_i \log(\lambda) - \sum_{i=1}^{n}\lambda - \sum_{i=1}^{n}\log(x_i!),$$

$$\log(P(\lambda, x)) = \log(\lambda)\sum_{i=1}^{n} x_i - n\lambda - \sum_{i=1}^{n}\log(x_i!).$$

Теперь можно вернуться в плоскость R и реализовать эту функцию. В R эту функцию можно написать очень элегантно, прибегнув к помощи векторизации. К тому же язык R идеально подходит для решения задач из области статистики, и в нем есть множество специальных встроенных функций вроде `lfactorial()`.

```
lprob_poisson <- function(lambda, x) {
    n <- length(x)
    (log(lambda) * sum(x)) - (n * lambda) - sum(lfactorial(x))
}
```

Рассмотрим приведенный ниже вектор наблюдений:

```
x1 <- c(41, 30, 31, 38, 29, 24, 30, 29, 31, 38)
```

Мы можем воспользоваться нашей функцией `lprob_poisson()` с целью вычисления (логарифмированной) вероятности `x1` для разных значений `lambda`.

```
lprob_poisson(10, x1)
#> [1] -184
lprob_poisson(20, x1)
#> [1] -61.1
lprob_poisson(30, x1)
#> [1] -31
```

До сих пор мы размышляли о `lambda` как о фиксированной и известной величине, а при помощи функции извлекали вероятность получения разных значений `x`. Но в реальной жизни у нас обычно есть наблюдение `x`, тогда как `lambda` неизвестна. *Правдоподобие* (likelihood) представляет собой функцию вероятности, рассмотренную сквозь эту призму: нам необходимо найти `lambda`, делающую наблюдение `x` наиболее вероятным. Иными словами, мы хотим узнать, при каком значении `lambda` для заданного `x` функция `lprob_poisson()` будет возвращать наивысшее значение.

В статистике такое изменение обозначается как $f_x(\lambda)$, а не $f_x(\lambda, x)$. В R мы для этого можем воспользоваться фабрикой функций. Мы будем передавать в нее `x` и возвращать функцию с единственным аргументом `λ`:

```
ll_poisson1 <- function(x) {
    n <- length(x)
```

```
    function(lambda) {
        log(lambda) * sum(x) - n * lambda - sum(lfactorial(x))
    }
}
```

Здесь нам нет необходимости использовать функцию `force()`, поскольку функция `length()` в неявном виде вычисляет значение `x`.

Одно из преимуществ такого подхода заключается в возможности выполнить некоторые предварительные вычисления: любые расчеты, в которых участвует только аргумент `x`, могут быть сделаны в фабрике лишь раз. Это может быть полезно, поскольку нам может понадобиться запускать эту функцию много раз для нахождения наилучшего значения `lambda`.

```
ll_poisson2 <- function(x) {
    n <- length(x)
    sum_x <- sum(x)
    c <- sum(lfactorial(x))

    function(lambda) {
        log(lambda) * sum_x - n * lambda - c
    }
}
```

Теперь мы можем воспользоваться этой функцией для нахождения значения `lambda`, при котором (логарифмированный) показатель правдоподобия будет максимальным:

```
ll1 <- ll_poisson2(x1)
ll1(10)
#> [1] -184
ll1(20)
#> [1] -61.1
ll1(30)
#> [1] -31
```

Вместо последовательного перебора мы можем автоматизировать этот процесс с помощью функции `optimise()`. Она будет запускать функцию `ll1()` множество раз и посредством математических приемов найдет оптимальное значение за наименьшее количество итераций. В результате мы получили наивысшее значение, равное –30,27 при `lambda`, равной 32,1:

```
optimise(ll1, c(0, 100), maximum = TRUE)
#> $maximum
#> [1] 32.1
#>
#> $objective
#> [1] -30.3
```

Мы могли бы решить эту задачу и без использования фабрики функций, поскольку функция `optimise()` может передавать аргумент … в оптимизируемую функцию. Это означает, что мы могли бы воспользоваться нашей исходной функцией напрямую, как показано ниже:

```
optimise(lprob_poisson, c(0, 100), x = x1, maximum = TRUE)
#> $maximum
#> [1] 32.1
#>
#> $objective
#> [1] -30.3
```

Как видите, в данном случае преимущество от применения фабрики функций оказалось несущественным, но необходимо учесть два нюанса:

- мы можем предварительно производить некоторые расчеты в фабрике, что сэкономит нам время, требуемое для выполнения итераций;
- двухуровневая структура функции лучше отражает математические предпосылки, лежащие в ее основе.

В задачах посложнее, при работе с несколькими параметрами и векторами данных, эти преимущества могут оказаться гораздо более существенными.

10.4.4 Упражнения

1. Почему в упомянутой ранее функции `boot_model()` нам нет необходимости форсировать вычисление аргументов `df` или `model` при помощи функции `force()`?
2. Почему мы могли бы написать функцию преобразования Бокса-Кокса следующим образом?

```
boxcox3 <- function(x) {
    function(lambda) {
        if (lambda == 0) {
            log(x)
        } else {
            (x ^ lambda - 1) / lambda
        }
    }
}
```

3. Почему не стоит беспокоиться о том, что функция `boot_permute()` будет хранить копию данных внутри сгенерированной ей функции?
4. Сколько времени позволит сэкономить функция `ll_poisson2()` в сравнении с `ll_poisson1()`? Воспользуйтесь функцией `bench::mark()`, чтобы узнать, насколько быстрее выполнится оптимизация. Как повлияет на результат длина вектора `x`?

10.5 Фабрики функций + функционалы

В завершение этой главы я покажу вам, как можно комбинировать фабрики функций с функционалами для преобразования данных во множество функций. В следующем примере мы создадим много особым образом названных функций для возведения в степень путем прохода по списку аргументов:

```
names <- list(
    square = 2,
    cube = 3,
    root = 1/2,
    cuberoot = 1/3,
    reciprocal = -1
)
funs <- purrr::map(names, power1)

funs$root(64)
#> [1] 8
funs$root
#> function(x) {
#>     x ^ exp
#>   }
#> <bytecode: 0x7fa43d0e1740>
#> <environment: 0x7fa43cb6bf90>
```

Эта идея может быть легко расширена, если ваша фабрика функций будет принимать два (замените функцию `map()` на `map2()`) или более (замените функцию `map()` на `pmap()`) аргументов.

Одним из недостатков нынешней конструкции является необходимость каждый вызов функции предварять префиксом `funs$`. Есть три способа избежать этого неудобства:

- в качестве временного решения можно воспользоваться функцией `with()`:

  ```
  with(funs, root(100))
  #> [1] 10
  ```

  Я рекомендую этот способ, поскольку он ясно дает понять, когда код выполняется в особом контексте и что из себя представляет этот контекст;

- для более длительного эффекта можно добавить функции к пути поиска с помощью функции `attach()` и удалить при необходимости с помощью функции `detach()`:

  ```
  attach(funs)
  #> The following objects are masked _by_ .GlobalEnv:
  #>
  #> cube, square
  ```

```
root(100)
#> [1] 10
detach(funs)
```

Возможно, вы слышали предостережения относительно использования функции `attach()`, и они не беспочвенны. Но эта ситуация немного отличается от обычной, ведь мы добавляем в путь поиска не датафрейм, а список функций. Вы с меньшей вероятностью решите модифицировать функцию по сравнению с колонкой в датафрейме, так что наиболее важные предостережения по поводу использования функции `attach()` здесь неприменимы;

- наконец, вы можете скопировать функции в глобальное окружение с помощью функции `env_bind()` (об операторе `!!!` вы узнаете в разделе 19.6). Действие этого изменения в большинстве случаев будет постоянным:

```
rlang::env_bind(globalenv(), !!!funs)
root(100)
#> [1] 10
```

Позже вы можете отвязать эти имена, но нет никакой гарантии, что за это время они не были привязаны кем-то другим. Так что это может привести к удалению чужих объектов.

```
rlang::env_unbind(globalenv(), names(funs))
```

Альтернативный способ решения этой задачи мы обсудим в разделе 19.7.4. Вместо использования фабрик функций вы могли бы создать функцию с квазицитированием. Это потребует от вас дополнительных знаний, но позволит создавать функции с понятным содержанием, не опасаясь за случайный захват больших объектов в замыкающей области видимости. Мы воспользуемся этой идеей в разделе 21.2.4, когда будем генерировать HTML из R.

10.5.1 Упражнения

1. Какие из приведенных ниже инструкций эквивалентны записи `with(x, f(z))`?
   a. `x$f(x$z)`
   b. `f(x$z)`
   c. `x$f(z)`
   d. `f(z)`
   e. Это зависит...
2. Сравните и противопоставьте эффект от использования функций `env_bind()` и `attach()` в приведенном ниже коде:

```
funs <- list(
    mean = function(x) mean(x, na.rm = TRUE),
```

```
    sum = function(x) sum(x, na.rm = TRUE)
)

attach(funs)
#> The following objects are masked from package:base:
#>
#> mean, sum
mean <- function(x) stop("Hi!")
detach(funs)

env_bind(globalenv(), !!!funs)
mean <- function(x) stop("Hi!")
env_unbind(globalenv(), names(funs))
```

11

Функциональные операторы

11.1 Введение

В этой главе мы поговорим о функциональных операторах. *Функциональным оператором* (function operator) называется функция, принимающая на вход одну или несколько функций и возвращающая функцию. В коде, приведенном ниже, показан простейший функциональный оператор `chatty()`. Он принимает на вход одну функцию и производит другую, которая, помимо выполнения действия, выводит на экран переданный ей аргумент.

```
chatty <- function(f) {
    force(f)

    function(x, ...) {
        res <- f(x, ...)
        cat("Processing ", x, "\n", sep = "")
        res
    }
}
f <- function(x) x ^ 2
s <- c(3, 2, 1)

purrr::map_dbl(s, chatty(f))
#> Processing 3
#> Processing 2
#> Processing 1
#> [1] 9 4 1
```

Функциональные операторы тесно связаны с фабриками функций. По сути, они и являются фабриками, принимающими на вход функции. Подобно фабрикам функций, функциональные операторы не являются неотъемлемой частью языка, без которой невозможно было бы что-то реализовать. Но они зачастую позволяют снизить сложность кода и облегчить его чтение и повторное использование.

Функциональные операторы обычно используются в паре с функционалами. Если у вас есть цикл `for`, вам вряд ли стоит использовать функциональ-

ный оператор, поскольку это усложнит код и даст лишь небольшую прибавку в быстродействии.

Если вы знакомы с языком Python, там функциональные операторы принято называть *декораторами* (decorator).

Структура главы

- В разделе 11.2 мы познакомимся с двумя чрезвычайно полезными функциональными операторами и научимся применять их на практике.
- В разделе 11.3 мы рассмотрим задачу, связанную с загрузкой множества веб-страниц, которую можно решить с помощью функциональных операторов.

Требования

Функциональные операторы представляют собой разновидность фабрик функций, так что перед чтением этой главы ознакомьтесь по крайней мере с разделом 6.2.

Мы будем использовать пакет `purrr` (https://purrr.tidyverse.org) для написания функционалов, с которыми вы познакомились в главе 9, и функциональных операторов, которые вам только предстоит изучить. Также мы загрузим пакет `memoise` (https://memoise.r-lib.org) [Уикем и др., 2018], чтобы воспользоваться оператором `memoise()`.

```
library(purrr)
library(memoise)
```

11.2 Существующие функциональные операторы

Есть два очень полезных функциональных оператора, которые помогут вам в решении часто возникающих задач и позволят понять, как в целом работают функциональные операторы. Это `purrr::safely()` и `memoise::memoise()`.

11.2.1 Перехват ошибок с помощью purrr::safely()

Одним из преимуществ циклов `for` является то, что при возникновении ошибки на одной из итераций вы по-прежнему можете воспользоваться результатами, полученными до ее появления:

```
x <- list(
    c(0.512, 0.165, 0.717),
```

```
    c(0.064, 0.781, 0.427),
    c(0.890, 0.785, 0.495),
    "oops"
)

out <- rep(NA_real_, length(x))
for (i in seq_along(x)) {
    out[[i]] <- sum(x[[i]])
}
#> Error in sum(x[[i]]): invalid 'type' (character) of argument
out
#> [1] 1.39 1.27 2.17   NA
```

Если сделать то же самое при помощи функционала, вы не увидите никакого вывода и не сможете понять, в какой момент возникла ошибка:

```
map_dbl(x, sum)
#> Error in .Primitive("sum")(..., na.rm = na.rm): invalid 'type'
#> (character) of argument
```

Функция `purrr::safely()` поможет в решении этой задачи. Этот функциональный оператор служит для преобразования функции таким образом, чтобы ошибки превращались в данные (основы этой идеи вы можете почерпнуть в разделе 8.6.2). Давайте для начала взглянем на этот оператор за пределами функции `map_dbl()`:

```
safe_sum <- safely(sum)
safe_sum
#> function (...)
#> capture_error(.f(...), otherwise, quiet)
#> <bytecode: 0x7fe46b20f348>
#> <environment: 0x7fe46b20eeb0>
```

Как и все функциональные операторы, `safely()` принимает на вход функцию и возвращает ее обертку в виде другой функции, которой мы впоследствии можем воспользоваться как обычно:

```
str(safe_sum(x[[1]]))
#> List of 2
#> $ result: num 1.39
#> $ error : NULL
str(safe_sum(x[[4]]))
#> List of 2
#>  $ result: NULL
#>  $ error :List of 2
#>   ..$ message: chr "invalid 'type' (character) of argument"
#>   ..$ call : language .Primitive("sum")(..., na.rm = na.rm)
#>   ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

Как видите, функция, полученная в результате преобразования с помощью оператора `safely()`, всегда возвращает список из двух элементов: `result` и `error`. При успешном завершении функции элемент `error` будет равен `NULL`, а `result` будет содержать результат вычисления. При неудаче, напротив, в элементе `result` будет `NULL`, а в `error` – информация об ошибке.

Теперь давайте используем оператор `safely()` совместно с функционалом:

```
out <- map(x, safely(sum))
str(out)
#> List of 4
#>  $ :List of 2
#>   ..$ result: num 1.39
#>   ..$ error : NULL
#>  $ :List of 2
#>   ..$ result: num 1.27
#>   ..$ error : NULL
#>  $ :List of 2
#>   ..$ result: num 2.17
#>   ..$ error : NULL
#>  $ :List of 2
#>   ..$ result: NULL
#>   ..$ error :List of 2
#>   .. ..$ message: chr "invalid 'type' (character) of argument"
#>   .. ..$ call : language .Primitive("sum")(..., na.rm = na.rm)
#>   .. ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condit"..
```

Вывод получился довольно необычным. Здесь мы видим четыре списка, в каждом из которых содержатся списки с элементами `result` и `error`. Упростить вывод можно с помощью функции `purrr::transpose()` – в результате мы получим список результатов и список ошибок:

```
out <- transpose(map(x, safely(sum)))
str(out)
#> List of 2
#>  $ result:List of 4
#>   ..$ : num 1.39
#>   ..$ : num 1.27
#>   ..$ : num 2.17
#>   ..$ : NULL
#>  $ error :List of 4
#>   ..$ : NULL
#>   ..$ : NULL
#>   ..$ : NULL
#>   ..$ :List of 2
#>   .. ..$ message: chr "invalid 'type' (character) of argument"
#>   .. ..$ call : language .Primitive("sum")(..., na.rm = na.rm)
#>   .. ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condit"..
```

Теперь можно легко извлечь результаты сработавших итераций и входные данные ошибочных:

```
ok <- map_lgl(out$error, is.null)
ok
#> [1] TRUE  TRUE  TRUE FALSE

x[!ok]
#> [[1]]
#> [1] "oops"

out$result[ok]
#> [[1]]
#> [1] 1.39
#>
#> [[2]]
#> [1] 1.27
#>
#> [[3]]
#> [1] 2.17
```

Эту технику можно использовать в самых разных сценариях. Представьте, что вы выполняете подгонку *обобщенной линейной модели* (generalised linear model – GLM) к списку датафреймов. Иногда при подгонке могут возникать ошибки, связанные с оптимизацией, но вы бы хотели попробовать пройтись по всем моделям, а позже проверить те, которые привели к ошибкам:

```
fit_model <- function(df) {
    glm(y ~ x1 + x2 * x3, data = df)
}

models <- transpose(map(datasets, safely(fit_model)))
ok <- map_lgl(models$error, is.null)

# наборы данных с ошибками
datasets[!ok]
# наборы данных без ошибок
models[ok]
```

Мне кажется это отличной демонстрацией мощи комбинирования функционалов и функциональных операторов: функция `safely()` здесь позволяет лаконично выразить, что вам нужно для решения поставленной задачи по анализу данных.

В библиотеке `purrr` также присутствует три других функциональных оператора со схожим назначением:

- `possibly()`: возвращает значение по умолчанию при возникновении ошибки. У этой функции нет механизма проверки, возникла ошибка или нет, так что она лучше подходит для случаев, когда у нас есть какое-то контрольное значение вроде `NA`;
- `quietly()`: преобразует побочные эффекты в виде вывода, сообщений и предупреждений в список с элементами `output`, `message` и `warning`;

- `auto_browser()`: автоматически запускает функцию `browser()` внутри функции при возникновении ошибки.

Ознакомьтесь с документацией для получения дополнительной информации.

11.2.2 Кеширование вычислений с помощью функции memoise::memoise()

Еще одним удобным функциональным оператором является `memoise::memoise()`. Он позволяет мемоизировать[1] функции, в результате чего они могут запоминать предыдущие входные значения и возвращать кешированный результат. *Мемоизация* – один из классических примеров компромисса между расходованием памяти и скоростью вычислений. Мемоизированная функция способна выполняться гораздо быстрее, но по причине хранения всех предыдущих входов и выходов она занимает больше места в памяти.

Давайте рассмотрим эту концепцию на простом примере, моделирующем выполнение дорогостоящей операции:

```
slow_function <- function(x) {
    Sys.sleep(1)
    x * 10 * runif(1)
}
system.time(print(slow_function(1)))
#> [1] 0.808
#>    user  system elapsed
#>    0.00    0.00    1.01

system.time(print(slow_function(1)))
#> [1] 8.34
#>    user  system elapsed
#>   0.002   0.000   1.006
```

После мемоизации функция замедляется при вызове с новыми аргументами. В то же время при вызове с аргументами, которые уже передавались ранее, она выполняется мгновенно, просто извлекая результат предыдущего вычисления.

```
fast_function <- memoise::memoise(slow_function)
system.time(print(fast_function(1)))
#> [1] 6.01
#>    user  system elapsed
#>       0       0       1

system.time(print(fast_function(1)))
#> [1] 6.01
```

[1] Термин *мемоизация* происходит от сочетания английских слов *memory* (память) и *optimization* (оптимизация). – *Прим. перев.*

```
#>    user  system elapsed
#>   0.019   0.000   0.019
```

Применительно к реальным задачам о пользе мемоизации можно говорить, например, при вычислении последовательности Фибоначчи. Как вы знаете, она определяется рекурсивно: первые два значения фиксированы – $f(0) = 0$, $f(1) = 1$, а далее вычисление производится по формуле $f(n) = f(n - 1) + f(n - 2)$ для любого положительного числа. Примитивная версия вычисления чисел Фибоначчи выполняется довольно медленно, поскольку, например, для получения числа `fib(10)` вам необходимо вычислить `fib(9)` и `fib(8)`, для получения `fib(9)` нужно рассчитать `fib(8)` и `fib(7)`.

```
fib <- function(n) {
    if (n < 2) return(1)
    fib(n - 2) + fib(n - 1)
}
system.time(fib(23))
#>    user  system elapsed
#>   0.040   0.002   0.043
system.time(fib(24))
#>    user  system elapsed
#>   0.064   0.003   0.068
```

Мемоизация функции `fib()` позволит значительно ускорить ее выполнение из-за однократного вычисления каждого значения:

```
fib2 <- memoise::memoise(function(n) {
    if (n < 2) return(1)
    fib2(n - 2) + fib2(n - 1)
})
system.time(fib2(23))
#>    user  system elapsed
#>    0.01    0.00    0.01
```

Будущие вызовы могут опираться на предыдущие вычисления:

```
system.time(fib2(24))
#>    user  system elapsed
#>   0.001   0.000   0.000
```

Здесь мы продемонстрировали пример *динамического программирования* (dynamic programming), когда сложная задача может разбиваться на множество пересекающихся подзадач, и сохранение результатов этих подзадач может существенно повысить производительность задачи в целом.

Дважды подумайте, перед тем как мемоизировать функцию. Если вы имеете дело не с *чистой* (pure) функцией, вывод которой зависит исключительно от входных аргументов, результат может вас очень удивить и привести в недоумение. При написании пакета `devtools` я допустил небольшую ошибку, мемоизировав медленную функцию `available.packages()`, которая выполняет

тяжелую загрузку из репозитория CRAN. Список доступных пакетов меняется не так часто, но если запустить процесс R и не останавливать его в течение нескольких дней, могут произойти изменения, которых вы не увидите. Из-за того что эта ошибка проявлялась только для длительных процессов, нам было не так легко отловить и исправить ее.

11.2.3 Упражнения

1. В базовом пакете R представлен функциональный оператор `Vectorize()`. Что он делает? Когда его можно использовать?
2. Прочитайте исходный код функции `possibly()`. Как она работает?
3. Прочитайте исходный код функции `safely()`. Как она работает?

11.3 Пример для разбора: создание собственных функциональных операторов

Операторы `memoise()` и `safely()` очень полезны в работе, но они достаточно сложны. В данном разделе вы научитесь писать свои собственные, более простые функциональные операторы. Представьте, что у вас есть именованный вектор с веб-адресами и вам необходимо скачать содержимое каждой ссылки на диск. Это можно очень просто сделать с помощью функций `walk2()` и `file.download()`:

```
urls <- c(
    "adv-r" = "https://adv-r.hadley.nz",
    "r4ds" = "http://r4ds.had.co.nz/"
    # и много-много других
)
path <- paste(tempdir(), names(urls), ".html")

walk2(urls, path, download.file, quiet = TRUE)
```

Неплохой подход для небольшого количества ссылок, но с ростом списка вам может понадобиться сделать следующие изменения:

- добавить небольшую задержку между обращениями, чтобы излишне не нагружать сервер;
- отображать точку (`.`) после каждых нескольких загрузок, чтобы мы понимали, что функция работает.

Это можно достаточно просто реализовать с помощью цикла `for`:

```
for(i in seq_along(urls)) {
    Sys.sleep(0.1)
    if (i %% 10 == 0) cat(".")
```

```
    download.file(urls[[i]], paths[[i]])
}
```

Но мне такой цикл кажется далеким от идеала, поскольку в нем чередуются разные задачи: остановка, отображение прогресса и загрузка. Это усложняет чтение кода и его повторное использование в других схожих ситуациях. Давайте посмотрим, смогут ли нам помочь функциональные операторы.

Для начала напишем функциональный оператор, добавляющий небольшую задержку. Я назову его `delay_by()` (скоро станет понятно, почему), и на вход он будет принимать два аргумента: функцию для оборачивания и время задержки. Наша итоговая реализация будет достаточно простой. Главное – не забыть принудительно вычислить все аргументы, о чем мы говорили в разделе 10.2.5, поскольку функциональные операторы представляют особый вид фабрик функций:

```
delay_by <- function(f, amount) {
    force(f)
    force(amount)

    function(...) {
        Sys.sleep(amount)
        f(...)
    }
}

system.time(runif(100))
#>    user  system elapsed
#>       0       0       0
system.time(delay_by(runif, 0.1)(100))
#>    user  system elapsed
#>   0.000   0.000   0.103
```

Мы можем использовать этот оператор совместно с функцией `walk2()`, как показано ниже:

```
walk2(urls, path, delay_by(download.file, 0.1), quiet = TRUE)
```

Написание функции для отображения прогресса в виде точек – задача посложнее, поскольку мы больше не можем полагаться на счетчик внутри цикла. Мы могли бы передавать индекс в качестве дополнительного аргумента, но это нарушит концепцию инкапсуляции, и функция отображения прогресса начнет зависеть от обработчика вышестоящего уровня. Вместо этого мы можем воспользоваться возможностью фабрик функций отслеживать свое состояние (мы говорили об этом в разделе 10.2.4), чтобы функция, отвечающая за отображение прогресса, располагала собственным внутренним счетчиком:

```
dot_every <- function(f, n) {
    force(f)
```

```
    force(n)

    i <- 0
    function(...) {
        i <<- i + 1
        if (i %% n == 0) cat(".")
        f(...)
    }
}
walk(1:100, runif)
walk(1:100, dot_every(runif, 10))
#> ..........
```

Теперь мы можем переписать наш цикл следующим образом:

```
walk2(
    urls, path,
    dot_every(delay_by(download.file, 0.1), 10),
    quiet = TRUE
)
```

Согласитесь, читать этот код не слишком легко из-за множества вызовов функций и разброса аргументов. Для облегчения чтения можно оформить код в виде конвейера:

```
walk2(
    urls, path,
    download.file %>% dot_every(10) %>% delay_by(0.1),
    quiet = TRUE
)
```

Конвейер смотрится здесь очень логично из-за подходящего выбора имен для наших функций. В результате мы получили почти полностью корректно выстроенное предложение на английском языке: *загрузи файл* (`download.file`), затем добавь *точку на каждые* (`dot_every`) 10 итераций, после чего *подожди* (`delay_by`) 0,1 с. Чем лучше вам удается выражать намерения кода с помощью имен функций, тем легче будет другим (и вам в будущем!) читать и понимать ваш код.

11.3.1 Упражнения

1. Взвесьте все за и против следующих двух конвейеров функций: `download.file %>% dot_every(10) %>% delay_by(0.1)` и `download.file %>% delay_by(0.1) %>% dot_every(10)`.
2. Стоит ли мемоизировать функцию `file.download()`? Аргументируйте свой ответ.
3. Напишите функциональный оператор, сообщающий о создании или удалении файлов в рабочей директории. Воспользуйтесь для этого

функциями `dir()` и `setdiff()`. Какие еще эффекты глобальных функций мы могли бы отслеживать?

4. Напишите функциональный оператор, записывающий в файл строку со временем и сообщением каждый раз при вызове функции.
5. Измените функцию `delay_by()` таким образом, чтобы вместо задержки на фиксированное время она отслеживала, что с момента последнего запуска функции прошло определенное время. Таким образом, при вызове последовательности команд `g <- delay_by(1, f); g(); Sys.sleep(2); g()` дополнительная задержка возникать не должна.

Часть III

Объектно ориентированное программирование

Введение

В следующих пяти главах книги мы будем говорить об *объектно ориентированном программировании* (object-oriented programming – OOP), или *ООП*. В R принципы объектно ориентированного программирования реализованы несколько сложнее в сравнении с другими языками, и на то немало причин:

- в R существует сразу несколько *систем ООП* (OOP system), из которых вы можете выбирать. Мы в этой книге будем подробно говорить о трех из них, которые я считаю наиболее важными и популярными: S3, R6 и S4. Системы S3 и S4 представлены в базовом пакете R. Система R6 поставляется в пакете R6, и она максимально похожа на *классы на основе ссылок* (reference classes – RC) из базового R;
- не существует общего мнения относительно важности каждой из представленных в R систем ООП. Мне, например, кажется наиболее полезной система S3, следом за которой по важности идут R6 и S4. Другие рассматривают в качестве приоритетной систему S4, а системы R6 и S3 не используют. В результате в разных сообществах R применяются разные подходы и системы ООП;
- в системах S3 и S4 используется подход к ООП на основе обобщенных функций, который отличается от подхода с инкапсуляцией, применяемого сегодня в большинстве других языков программирования[1]. Совсем скоро мы подробно поговорим обо всех этих терминах, но стоит отметить, что, несмотря на общность идей и концепций объектно ориентированного программирования, реализации этого подхода в разных языках могут кардинально отличаться. Это затрудняет адаптацию и применение навыков, приобретенных в других языках с применением ООП, в языке R.

В целом в R гораздо более важной и применимой в сравнении с ООП является концепция функционального программирования, позволяющая декомпозировать сложные задачи на отдельные функции, а не объекты. Тем не менее существуют довольно весомые причины для изучения всех трех систем ООП:

- система S3 позволяет вашим функциям возвращать подробные результаты в дружественном для пользователя виде и с полезным для программиста внутренним наполнением. Эта система повсеместно используется в базовом R, так что вам необходимо ее освоить, если вы хотите

[1] Исключением является язык Julia, в котором также используется система ООП на основе обобщенных функций. При этом реализация ООП в Julia, в отличие от R, является более совершенной и отличается высокой производительностью.

расширять базовые функции R для работы с новыми типами входных данных;

- система R6 представляет стандартизированный подход, связанный с отказом от принципа *копирования при изменении* (copy-on-modify), принятого в R. Это особенно важно, если вы хотите моделировать объекты, существующие отдельно от R. Сегодня в основном система R6 используется для моделирования данных, приходящих от внешних интерфейсов API, где изменения могут поступать как изнутри R, так и снаружи;
- система S4 является достаточно строгой, чтобы заставить разработчика со всей ответственностью относиться к структуре программного обеспечения. В связи с этим она отлично подходит для построения масштабных программных комплексов, развивающихся со временем, в разработке которых принимает участие множество программистов. По этой причине именно система S4 лежит в основе проекта Bioconductor. И одним из поводов для ее изучения является возможность в будущем внести свой вклад в развитие этого масштабного проекта.

Цель этого вводного раздела состоит в том, чтобы снабдить вас всей необходимой информацией для погружения в мир ООП и навыками для определения используемых систем в различных проектах. В следующих главах мы будем детально говорить о каждой системе ООП.

1. В главе 12 будет дано описание базовых типов, лежащих в основе всех систем объектно ориентированного программирования.
2. В главе 13 мы познакомимся с S3, наиболее простой и распространенной системой ООП в языке R.
3. Глава 14 будет посвящена системе R6, построенной на базе инкапсуляции, в основе которой лежат окружения.
4. В главе 15 мы поговорим о системе S4, которая от S3 отличается своей повышенной строгостью и формальностью.
5. В главе 16 мы проведем сравнительный анализ всех трех систем ООП, представленных в этой книге. Усвоив все преимущества и недостатки разных систем, вы сможете применять их там и тогда, когда это будет наиболее уместно.

В данной книге мы будем акцентировать внимание на механике работы систем ООП, а не на их эффективном использовании, и вам может быть непросто понять эту концепцию в целом, если вы ни разу не сталкивались с объектно ориентированным программированием ранее. Вас может удивить, почему я решил не углубляться в тему ООП досконально. Мое решение ограничиться освещением механики использования ООП продиктовано тем, что эта тема должна быть где-то полноценно расписана (при написании этих глав мне пришлось проштудировать много литературы и провести детальный анализ самостоятельно), а для погружения в область эффективного использования ООП на практике нашей книги все равно не хватит – тут без шансов!

Системы ООП

Разные разработчики используют терминологию, принятую в мире объектно ориентированного программирования, по-разному, так что начнем мы повествование с описания словаря важных терминов. При этом мы не будем углубляться в подробности терминологии, зато будем периодически возвращаться к этой теме в следующих главах.

Основная причина использования концепции ООП состоит в применении *полиморфизма* (polymorphism), т. е., дословно, многообразия форм. Полиморфизм позволяет разработчику рассматривать интерфейс функции отдельно от ее реализации, что дает возможность применять одну и ту же функциональную форму с разными типами входа. Эта идея тесно переплетается с принципом *инкапсуляции* (encapsulation), заключающимся в том, что пользователю нет необходимости заботиться о содержимом объекта, которое инкапсулировано (скрыто) за стандартным интерфейсом.

В качестве примера действия полиморфизма можно привести функцию `summary()`, вывод которой меняется в зависимости от типа аргумента:

```
diamonds <- ggplot2::diamonds

summary(diamonds$carat)
#>    Min.  1st Qu.  Median   Mean  3rd Qu.   Max.
#>    0.20     0.40    0.70   0.80     1.04   5.01

summary(diamonds$cut)
#>      Fair    Good  Very Good   Premium    Ideal
#>      1610    4906      12082     13791    21551
```

Вы могли бы представить себе функцию `summary()` как набор инструкций `if-else`, но это бы означало, что только автор функции может изменять реализацию этой функции. Использование системы объектно ориентированного программирования позволяет любому разработчику расширять интерфейс функции с применением отдельных реализаций для разных типов входа.

Если быть более точными, в системах ООП тип объекта называется его *классом* (class), а реализация функции, присущей конкретному классу, – *методом* (method). Грубо говоря, класс отвечает на вопрос о том, *чем* является объект, а метод – на вопрос о том, *что он делает*. В рамках класса определяются *поля* (field), отвечающие за данные, которыми владеет каждый *экземпляр* (instance) класса. Классы организованы в иерархии таким образом, что если метод не определен для конкретного класса, будет вызываться соответствующий метод его родителя, и в этом случае мы говорим, что класс *наследует* (inherit) поведение своего родителя. К примеру, в R *упорядоченный фактор* (ordered factor), который мы уже упоминали ранее, наследует поведение от обычного фактора, а класс обобщенной линейной модели происходит от обычной линейной модели. Процесс нахождения подходящего метода в иерархии называется *диспетчеризацией методов* (method dispatch).

Существуют две основные парадигмы объектно ориентированного программирования, различающиеся тем, как методы соотносятся с классами. Мы будем пользоваться терминологией, примененной в книге *Extending R* [Чемберс (Chambers), 2016], согласно которой эти парадигмы называются *инкапсулированным ООП* (encapsulated OOP) и *функциональным ООП* (functional OOP):

- при использовании инкапсулированного ООП методы принадлежат объектам или классам, а их вызов выглядит следующим образом: `object.method(arg1, arg2)`. Эта парадигма называется инкапсулированной по причине того, что объект в этом случае заключает в себе и данные (посредством полей), и поведение (посредством методов). Такой подход используется в большинстве популярных языков программирования;
- при использовании функционального ООП методы принадлежат *обобщенным функциям* (generic function), а их вызов со стороны выглядит как вызов обычных функций: `generic(object, arg2, arg3)`. Функциональным такой подход называется из-за того, что и внешне он напоминает вызов функций, и внутренне все компоненты реализованы в виде функций.

Итак, теперь вы владеете всей необходимой терминологией и готовы к погружению в разные системы ООП, использующиеся в R.

ООП в R

В базовом пакете R представлены три системы объектно ориентированного программирования: S3, S4 и классы на основе ссылок (reference classes – RC):

- *система S3* – первая система ООП в языке R, и она подробно описана в книге *Statistical Models in S* [Чемберс (Chambers) и Хасти (Hastie), 1992]. S3 представляет собой нестрогую реализацию функционального ООП и полагается больше на общепринятые соглашения, а не на жесткие правила. Это значительно снижает порог входа в данную систему для новичков, которые способны довольно быстро приступить к решению с ее помощью конкретных задач;
- *система S4* является формализованной и строгой версией системы S3, которая была впервые представлена в книге *Programming with Data* [Чемберс, 1998]. Для полноценного использования этой системы требуется выполнить гораздо больше подготовительной работы по сравнению с системой S3, но взамен вы получите больше гарантий в плане надежности системы и серьезный уровень инкапсуляции. Система S4 реализована в пакете `methods`, который всегда устанавливается вместе с базовым пакетом R. (Вы могли бы спросить: а куда подевались системы S1 и S2? А никуда. Системы S3 и S4 получили свои имена по версиям языка S, в которых были реализованы. В первых двух версиях S отсутствовала система ООП);
- *система RC* представляет собой реализацию функционального ООП. Объекты RC являются особыми типами объектов S4, допускающими изменение. Таким образом, вопреки семантике копирования при из-

менении, принятой в R, эти объекты могут быть изменены на месте. Это делает их более сложными для освоения, но при этом позволяет решать задачи, с которыми трудно справиться с помощью функционального стиля ООП, принятого в системах S3 и S4.

Также в репозитории CRAN вы можете обнаружить и другие системы ООП, включая следующие:

- *система R6* [Чанг (Chang), 2017] реализует инкапсулированный подход к ООП по примеру RC, но при этом исключает некоторые проблемы. В этой книге мы будем говорить именно об этой системе, а не о системе RC, по причинам, которые будут указаны в разделе 14.5;
- *система R.oo* [Бенгтссон (Bengtsson), 2003] добавляет системе S3 некоторой строгости и позволяет изменять объекты S3 на месте;
- *система proto* [Гротендик (Grothendieck) и др., 2016] реализует еще один стиль ООП, базирующийся на идее прототипов, что помогает размыть границы между классами и их экземплярами (объектами). Недолгое время я был очарован идеей программирования на основе прототипов и использовал эту технику в пакете `ggplot2`, но сейчас я больше склоняюсь к каноническому стилю.

Помимо R6, получившей широкое распространение, остальные системы представляют, скорее, теоретический интерес. У них есть свои преимущества, но не так много пользователей R знают и понимают их, что не добавляет им популярности на рынке.

sloop

Прежде чем двигаться дальше, я бы хотел познакомить вас с пакетом `sloop`:

```
library(sloop)
```

Пакет `sloop` предлагает набор вспомогательных функций, которых очень не хватает в базовом R. Одна из них – функция `sloop::otype()`. С помощью нее можно узнать, какую систему ООП использует тот или иной объект:

```
otype(1:10)
#> [1] "base"

otype(mtcars)
#> [1] "S3"

mle_obj <- stats4::mle(function(x = 1) (x - 2) ^ 2)
otype(mle_obj)
#> [1] "S4"
```

Вы можете использовать эту функцию, чтобы узнать, к какой главе обращаться за инструкциями по работе с нужным вам объектом.

12

Базовые типы

12.1 Введение

Чтобы начать обсуждение объектов и концепции объектно ориентированного программирования в языке R в целом, сначала нам необходимо прояснить само понятие *объект* (object) и рассмотреть два варианта его использования. До сих пор в этой книге мы применяли данный термин в общем смысле, подразумевая простое определение, данное Джоном Чемберсом: «Все, что присутствует в R, является объектом». Тем не менее, несмотря на объектную природу вещей в R, не все в этом языке имеет отношение к парадигме объектного ориентирования. Это недопонимание возникло в связи с тем, что базовые объекты пришли из языка S, и они были разработаны в те времена, когда никто не задумывался о том, что языку понадобится система ООП. В результате инструменты и термины на протяжении многих лет развивались без использования каких-то единых принципов.

В большинстве случаев различия между объектами и объектами ООП не так важны. Но нам необходимо докопаться до самой сути, так что мы будем использовать термины *базовый объект* (base object) и *объект ООП* (OO object), как показано на рис. 12.1.

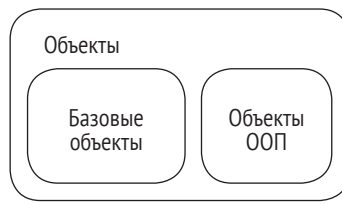


Рис. 12.1 Базовые объекты и объекты ООП

Структура главы

- В разделе 12.2 мы узнаем, как можно идентифицировать базовые объекты и объекты ООП.
- В разделе 12.3 мы представим полный набор базовых типов, используемых для построения всех объектов.

12.2 Базовые объекты против объектов ООП

Чтобы увидеть различия между базовыми объектами и объектами ООП, можно воспользоваться функциями `is.object()` или `sloop::otype()`:

```
# Базовый объект:
is.object(1:10)
#> [1] FALSE
sloop::otype(1:10)
#> [1] "base"

# Объект ООП
is.object(mtcars)
#> [1] TRUE
sloop::otype(mtcars)
#> [1] "S3"
```

Чисто технически отличие объекта ООП от базового объекта состоит в наличии у него атрибута `class`:

```
attr(1:10, "class")
#> NULL
attr(mtcars, "class")
#> [1] "data.frame"
```

Возможно, вы уже знакомы с функцией `class()`. Эту функцию можно безопасно использовать совместно с объектами S3 и S4, но применительно к базовым объектам она возвращает сбивающие с толку результаты. Гораздо безопаснее воспользоваться функцией `sloop::s3_class()`, которая возвращает неявный класс, который будет использован системами S3 и S4 для поиска методов. Подробнее о функции `s3_class()` вы сможете почитать в разделе 13.7.1.

```
x <- matrix(1:4, nrow = 2)
class(x)
#> [1] "matrix"
sloop::s3_class(x)
#> [1] "matrix"  "integer" "numeric"
```

12.3 Базовые типы

Хотя атрибут `class` присутствует только у объектов ООП, *базовым типом* (base type) располагают все без исключения объекты:

```
typeof(1:10)
#> [1] "integer"

typeof(mtcars)
#> [1] "list"
```

Базовые типы не представляют собой пример работы системы ООП, поскольку функции, которые ведут себя по-разному для разных базовых типов, преимущественно написаны на языке C с использованием инструкций `switch`. Это означает, что только разработчики R могут добавлять новые типы, что связано с огромной работой по адаптации каждого обработчика `switch` к новому типу. Как следствие новые типы появляются очень редко. Последнее изменение в этой области произошло на моей памяти в 2011 году, когда были добавлены два экзотических типа, которые вы вряд ли встретите в самом R, но которые используются для диагностики проблем с расходом памяти. До этого последним добавленным типом был особый базовый тип для объектов S4, и было это в 2005 году.

В общей сложности существует 25 различных базовых типов. Давайте перечислим их все, сгруппировав по главам упоминания в этой книге. Наибольшую важность эти типы имеют для кода на C, так что вы зачастую будете видеть обращение к ним по имени типа в C. Эти имена я заключил в скобки:

- *векторы, глава 3*: `NULL` (`NULLSXP`), `logical` (`LGLSXP`), `integer` (`INTSXP`), `double` (`REALSXP`), `complex` (`CPLSXP`), `character` (`STRSXP`), `list` (`VECSXP`) и `raw` (`RAWSXP`);

  ```
  typeof(NULL)
  #> [1] "NULL"
  typeof(1L)
  #> [1] "integer"
  typeof(1i)
  #> [1] "complex"
  ```

- *функции, глава 6*: `closure` (обычные функции R, `CLOSXP`), `special` (внутренние функции, `SPECIALSXP`) и `builtin` (примитивные функции, `BUILTINSXP`).

  ```
  typeof(mean)
  #> [1] "closure"
  typeof(`[`)
  #> [1] "special"
  typeof(sum)
  #> [1] "builtin"
  ```

  Внутренние и примитивные функции были рассмотрены в разделе 6.2.2;

- *окружения, глава 7*: `environment` (`ENVSXP`);

  ```
  typeof(globalenv())
  #> [1] "environment"
  ```

- *тип S4 (*S4SXP*), глава 15*: используется для классов S4, не наследуемых от существующего базового типа;

```
mle_obj <- stats4::mle(function(x = 1) (x - 2) ^ 2)
typeof(mle_obj)
#> [1] "S4"
```

- *компоненты языка, глава 18*: symbol (также известный как имя, SYMSXP), language (обычно именуемый вызовом, LANGSXP) и pairlist (используемый для аргументов функций, LISTSXP).

```
typeof(quote(a))
#> [1] "symbol"
typeof(quote(a + 1))
#> [1] "language"
typeof(formals(mean))
#> [1] "pairlist"
```

  Специальный тип expression (EXPRSXP) служит для возвращения из функций parse() и expression(). Обычно в нем нет нужды в пользовательском коде;
- оставшиеся типы редко используются в R. В основном они нужны для кода на C: externalptr (EXTPTRSXP), weakref (WEAKREFSXP), bytecode (BCODESXP), promise (PROMSXP), ... (DOTSXP) и any (ANYSXP).

Вы также могли слышать о функциях mode() и storage.mode(). Не используйте эти функции: они присутствуют только для совместимости с языком S.

12.3.1 Числовой тип

Будьте осторожны, говоря о *числовом типе* (numeric type), поскольку в R термин numeric может быть использован в трех разных аспектах.

1. Иногда имя numeric используется в качестве псевдонима типа double. К примеру, записи as.numeric() и numeric() эквивалентны as.double() и double() соответственно. (Также в R иногда используется слово real вместо double; например, вы можете встретить это в примере с NA_real_.)
2. В системах S3 и S4 слово numeric используется в качестве псевдонима для типов integer и double и применяется для поиска методов:

```
sloop::s3_class(1)
#> [1] "double" "numeric"
sloop::s3_class(1L)
#> [1] "integer" "numeric"
```

3. Функция is.numeric() определяет, *ведет* ли себя объект как числовой. К примеру, факторы обладают типом integer, но не ведут себя как числа (т. е. нет никакого смысла вычислять среднее значение по фактору).

```
typeof(factor("x"))
#> [1] "integer"
is.numeric(factor("x"))
#> [1] FALSE
```

На протяжении этой книги я буду использовать слово `numeric` для обозначения объектов с типом `integer` или `double`.

13

Система S3

13.1 Введение

S3 является первой и простейшей системой объектно ориентированного программирования в R. Эту систему можно назвать нестрогой и узкоспециализированной, но в присущем ей минимализме есть своя элегантность: из S3 нельзя что-то убрать, не нарушив при этом целостность системы. Поэтому вы должны пользоваться именно этой системой ООП, если у вас нет достаточных оснований для иного выбора. S3 – это единственная система ООП, используемая в пакетах `base` и `stats`, да и в пакетах из репозитория CRAN в большинстве случаев используется именно она.

S3 – довольно гибкая система, позволяющая вам реализовывать даже не самые продуманные и оптимальные решения. Если вы пришли из среды наподобие Java, отличающейся своей строгостью, вас это может напугать, но в то же время это дает больше свободы разработчикам на языке R. Да, в этой свободной среде бывает непросто убедить людей не делать то, что не нужно, но, с другой стороны, пользователи не будут испытывать неудобства от того, что вы что-то не реализовали. Поскольку в системе S3 не так много встроенных ограничений, ключом к успешному ее использованию является наложение собственных ограничений. В этой главе, помимо прочего, вы познакомитесь с соглашениями, касающимися этой системы, которым (почти) всегда необходимо следовать.

Цель этой главы – продемонстрировать вам работу системы S3, а не научить вас оптимально использовать ее для создания новых классов и обобщенных функций. Я бы рекомендовал совмещать теоретические знания из этой главы с практическими наработками, заложенными в пакете `vctrs` (https://vctrs.r-lib.org).

Структура главы

- В разделе 13.2 мы бегло познакомимся со всеми основными компонентами системы S3: классами, обобщенными функциями и методами. Вы также узнаете о функции `sloop::s3_dispatch()`, которую мы будем использовать на протяжении всей главы для исследования принципов работы системы S3.

- В разделе 13.3 мы погрузимся в детали создания новых классов S3 и познакомимся с тремя функциями, которые должны присутствовать в большинстве классов.
- В разделе 13.4 описывается работа обобщенных функций и методов S3, а также основы диспетчеризации методов.
- В разделе 13.5 мы познакомимся с четырьмя основными стилями объектов S3: вектором, записью, датафреймом и скаляром.
- В разделе 13.6 демонстрируется работа наследования в системе S3 и описывается, что необходимо сделать, чтобы на основе класса можно было создавать подклассы.
- В разделе 13.7 мы подведем итоги главы и подробнее поговорим о диспетчеризации методов, базовых типах, внутренних и групповых обобщенных функциях, а также о двойной диспетчеризации.

Требования

Классы S3 реализуются посредством атрибутов, так что для освоения этой темы вам необходимо хорошо понимать все, о чем мы говорили в разделе 3.3. Для примеров мы будем использовать существующие базовые векторы S3, а значит, вы должны быть знакомы с факторами и классами `Date`, `difftime`, `POSIXct` и `POSIXlt`, которые мы описывали в разделе 3.4.

Также в этой главе мы будем пользоваться пакетом `sloop` (https://sloop.r-lib.org).

```
library(sloop)
```

13.2 Основы

Объект S3 представляет собой базовый тип, у которого как минимум присутствует атрибут `class` (другие атрибуты могут использоваться для хранения других данных). Возьмем, к примеру, фактор. Его базовым типом является целочисленный вектор, атрибут `class` хранит значение `"factor"`, а в атрибуте `levels` перечислены все допустимые уровни:

```
f <- factor(c("a", "b", "c"))
typeof(f)
#> [1] "integer"
attributes(f)
#> $levels
#> [1] "a" "b" "c"
#>
#> $class
#> [1] "factor"
```

Для получения базового типа можно деклассировать переменную с помощью функции `unclass()`, которая удаляет атрибут `class`, тем самым лишая переменную особого поведения:

```
unclass(f)
#> [1] 1 2 3
#> attr(,"levels")
#> [1] "a" "b" "c"
```

Объект S3 ведет себя иначе по сравнению с лежащим в его основе базовым типом при передаче в *обобщенную функцию* (generic function). Самый простой способ определения, является ли функция обобщенной, состоит в использовании функции `sloop::ftype()` с последующим поиском строки `"generic"` в выводе:

```
ftype(print)
#> [1] "S3"      "generic"
ftype(str)
#> [1] "S3"      "generic"
ftype(unclass)
#> [1] "primitive"
```

Обобщенная функция определяет интерфейс, использующий разные реализации в зависимости от класса аргумента (почти всегда первого). Многие базовые функции в R являются обобщенными, включая важную функцию `print()`:

```
print(f)
#> [1] a b c
#> Levels: a b c

# удаление атрибута class возвращает поведение целочисленной переменной
print(unclass(f))
#> [1] 1 2 3
#> attr(,"levels")
#> [1] "a" "b" "c"
```

Обратите внимание, что функция `str()` также является обобщенной, и некоторые классы S3 используют ее для скрытия внутренних деталей. К примеру, класс `POSIXlt`, применяемый для представления даты и времени, создан на основе списка, но эта информация скрывается на выходе из функции `str()`:

```
time <- strptime(c("2017-01-01", "2020-05-04 03:21"), "%Y-%m-%d")
str(time)
#>  POSIXlt[1:2], format: "2017-01-01" "2020-05-04"

str(unclass(time))
#> List of 11
#>  $ sec   : num [1:2] 0 0
```

```
#>  $ min   : int [1:2] 0 0
#>  $ hour  : int [1:2] 0 0
#>  $ mday  : int [1:2] 1 4
#>  $ mon   : int [1:2] 0 4
#>  $ year  : int [1:2] 117 120
#>  $ wday  : int [1:2] 0 1
#>  $ yday  : int [1:2] 0 124
#>  $ isdst : int [1:2] 0 1
#>  $ zone  : chr [1:2] "CST" "CDT"
#>  $ gmtoff: int [1:2] NA NA
#>  - attr(*, "tzone")= chr [1:3] "America/Chicago" "CST" "CDT"
```

Обобщенную функцию можно представить в виде своеобразного посредника: ее задача заключается в том, чтобы определить интерфейс (т. е. аргументы), а затем найти нужную реализацию для заданного случая. Реализация для конкретного класса называется *методом* (method), а процесс поиска нужного метода именуется диспетчеризацией.

Для отслеживания процесса диспетчеризации методов можно воспользоваться функцией `sloop::s3_dispatch()`:

```
s3_dispatch(print(f))
#> => print.factor
#>  * print.default
```

Подробнее о процессе диспетчеризации методов мы будем говорить в разделе 13.4.1, а сейчас вам достаточно знать, что методы S3 представляют собой особые функции со своим принципом именования, которые условно можно записать так: `generic.class()`. Например, метод `factor` для обобщенной функции `print()` будет именоваться `print.factor()`. Вам никогда не придется вызывать эти методы напрямую, вместо этого вы всегда будете полагаться на обобщенные функции, которые должны выполнять поиск нужного метода за вас.

В основном методы можно идентифицировать по наличию точки (`.`) в их именах, но в базовом R присутствует немало важных функций, написанных еще до появления системы S3, в именах которых точка используется для соединения слов. Если вы не уверены в происхождении имени, воспользуйтесь вспомогательной функцией `sloop::ftype()`:

```
ftype(t.test)
#> [1] "S3"      "generic"
ftype(t.data.frame)
#> [1] "S3"      "method"
```

В отличие от большинства функций, вы не можете видеть исходный код большей части методов S3[1], просто введя их имя. Причина этого в том, что методы S3 обычно не экспортируются: они живут внутри пакета и недо-

[1] Исключение составляют методы из базового класса вроде `t.data.frame`, а также методы, написанные вами самими.

ступны из глобального окружения. Для просмотра кода метода вы можете воспользоваться функцией `sloop::s3_get_method()`, которая сработает вне зависимости от того, где живет метод:

```
weighted.mean.Date
#> Error in eval(expr, envir, enclos): object 'weighted.mean.Date' not
#> found

s3_get_method(weighted.mean.Date)
#> function (x, w, ...)
#> structure(weighted.mean(unclass(x), w, ...), class = "Date")
#> <bytecode: 0x7fed4af8d778>
#> <environment: namespace:stats>
```

13.2.1 Упражнения

1. Опишите различия между функциями `t.test()` и `t.data.frame()`. Когда каждая из них вызывается?
2. Перечислите в списке известные вам популярные базовые функции R, не являющиеся методами S3, в именах которых присутствует точка.
3. Что делает метод `as.data.frame.data.frame()`? Почему он может сбивать с толку? Как можно избежать подобных недоразумений в собственном коде?
4. Опишите различия между показанными ниже вызовами:

```
set.seed(1014)
some_days <- as.Date("2017-01-31") + sample(10, 5)

mean(some_days)
#> [1] "2017-02-05"
mean(unclass(some_days))
#> [1] 17202
```

5. Какой класс объекта вернет следующий код? На каком базовом типе он базируется? Какие атрибуты использует?

```
x <- ecdf(rpois(100, 10))
x
#> Empirical CDF
#> Call: ecdf(rpois(100, 10))
#> x[1:18] = 2, 3, 4, ..., 2e+01, 2e+01
```

6. Какой класс объекта вернет следующий код? На каком базовом типе он базируется? Какие атрибуты использует?

```
x <- table(rpois(100, 5))
x
```

```
#>
#> 1  2  3  4  5  6  7  8  9 10
#> 8  5 18 14 12 19 12  3  5  4
```

13.3 Классы

Если вы сталкивались с объектно ориентированным программированием в других языках, то можете удивиться тому, что система S3 не предусматривает формального объявления классов. Чтобы сделать объект экземпляром класса, вы просто устанавливаете для него атрибут `class`. Это можно сделать как при создании объекта с помощью функции `structure()`, так и впоследствии, воспользовавшись конструкцией `class<-()`:

```
# Создание объекта и установка атрибута class в один шаг
x <- structure(list(), class = "my_class")

# Создание объекта, а затем установка атрибута class
x <- list()
class(x) <- "my_class"
```

Класс объекта S3 можно определить с помощью функции `class(x)`, а узнать, является ли объект экземпляром класса, – с помощью `inherits(x, "class-name")`.

```
class(x)
#> [1] "my_class"
inherits(x, "my_class")
#> [1] TRUE
inherits(x, "your_class")
#> [1] FALSE
```

В качестве имени класса можно использовать любые символы, но я рекомендовал бы ограничиться буквами и символами подчеркивания (_). Избегайте точек в именах, чтобы не возникала путаница с разделителями между именами обобщенных функций и именами классов. При использовании класса внутри пакета лучше включать название пакета в имя класса. Это позволит избежать случайных конфликтов с классами из других пакетов.

В системе S3 не реализована проверка на корректность имен, что позволяет изменять класс существующих объектов:

```
# Создаем линейную модель
mod <- lm(log(mpg) ~ log(disp), data = mtcars)
class(mod)
#> [1] "lm"
print(mod)
#>
```

```
#> Call:
#> lm(formula = log(mpg) ~ log(disp), data = mtcars)
#>
#> Coefficients:
#> (Intercept) log(disp)
#>       5.381    -0.459

# Превращаем объект в дату (?!)
class(mod) <- "Date"

# Неудивительно, что это не очень хорошо работает
print(mod)
#> Error in as.POSIXlt.Date(x): (list) object cannot be coerced to type
#> 'double'
```

Если вы работали с другими объектно ориентированными языками программирования, это может вызвать у вас приступ тошноты, но на практике такая гибкость не доставляет особых хлопот. R не будет препятствовать тому, чтобы вы выстрелили себе в ногу, но если не направлять заряженное оружие вниз и не нажимать на курок, проблем не будет.

Во избежание травмирования я рекомендую при создании собственных классов всегда снабжать их следующими функциями:

- низкоуровневый *конструктор* (constructor), `new_myclass()`, способствующий созданию новых объектов с правильной структурой;
- *валидатор* (validator), `validate_myclass()`, производящий более затратные с точки зрения ресурсов проверки значений объекта на корректность;
- *помощник* (helper), `myclass()`, облегчающий создание объектов вашего класса.

Для совсем простых классов нет никакой необходимости создавать валидатор, и вы можете не снабжать класс помощником, если собираетесь использовать его для внутренних целей. Но конструктор должен быть всегда.

13.3.1 Конструкторы

В системе S3 не предусмотрено формальное определение классов, так что нет никакой возможности встроенными средствами обеспечить единообразие всех объектов определенного класса в плане структуры (т. е. использования одного и того же базового типа и атрибутов одного типа). В связи с этим однородность структуры объектов необходимо поддерживать с помощью конструктора.

Конструктор класса должен отвечать трем требованиям:

- называться `new_myclass()`;
- располагать одним аргументом для базового объекта и дополнительными аргументами для каждого атрибута;
- осуществлять проверку типа базового объекта и типов всех атрибутов.

Я продемонстрирую эти идеи на примере конструкторов базовых классов[1], с которыми вы уже знакомы. Для начала рассмотрим конструктор для простейшего класса S3 `Date`, представляющего собой число двойной точности (`double`) с единственным атрибутом `class` со значением «Date». Таким образом, конструктор этого класса будет очень простым:

```
new_Date <- function(x = double()) {
    stopifnot(is.double(x))
    structure(x, class = "Date")
}

new_Date(c(-1, 0, 1))
#> [1] "1969-12-31" "1970-01-01" "1970-01-02"
```

Конструкторы призваны облегчить жизнь разработчиков. В связи с этим они могут быть предельно лаконичными, и в них вы не должны заботиться о реализации сложной системы сообщений об ошибках. Если предполагается, что пользователи также смогут создавать свои объекты, вы должны будете написать удобную функцию-помощник с именем `class_name()`, о чем мы поговорим совсем скоро.

Более сложным конструктором располагает класс `difftime`, с помощью которого может быть представлена разница во времени. В основе этого класса также лежит тип `double`, но у него еще есть атрибут `units`, который может принимать одно из допустимых значений:

```
new_difftime <- function(x = double(), units = "secs") {
    stopifnot(is.double(x))
    units <- match.arg(units, c("secs", "mins", "hours", "days", "weeks"))

    structure(x,
        class = "difftime",
        units = units
    )
}

new_difftime(c(1, 10, 3600), "secs")
#> Time differences in secs
#> [1] 1 10 3600
new_difftime(52, "weeks")
#> Time difference of 52 weeks
```

Конструктор – это функция разработчика, и она зачастую будет вызываться опытными пользователями. Это означает, что при его написании вы можете пожертвовать безопасностью ради быстродействия и вам нет необходимости выполнять ресурсоемкие проверки в конструкторе.

---

[1] В последних версиях R предусмотрены конструкторы для `.Date()`, `.difftime()`, `.POSIXct()` и `.POSIXlt()`, но они внутренние, плохо документированные и не отвечают рекомендуемым мной принципам.

13.3.2 Валидаторы

Более сложные классы требуют и более серьезного подхода к проверке данных на корректность. Рассмотрим в качестве примера факторы. Конструктор класса осуществляет проверку только на правильность типов, что при желании позволяет создавать нежизнеспособные факторы:

```
new_factor <- function(x = integer(), levels = character()) {
    stopifnot(is.integer(x))
    stopifnot(is.character(levels))

    structure(
        x,
        levels = levels,
        class = "factor"
    )
}

new_factor(1:5, "a")
#> Error in as.character.factor(x): malformed factor
new_factor(0:1, "a")
#> Error in as.character.factor(x): malformed factor
```

Вместо того чтобы расширять конструктор за счет дополнительных сложных проверок, лучше будет вынести их в отдельную функцию. Это позволит вам быстро создавать новые объекты, если вы уверены в правильности вводимых значений, и повторно использовать проверки, размещенные отдельно.

```
validate_factor <- function(x) {
    values <- unclass(x)
    levels <- attr(x, "levels")

    if (!all(!is.na(values) & values > 0)) {
        stop(
            "All `x` values must be non-missing and greater than zero",
            call. = FALSE
        )
    }

    if (length(levels) < max(values)) {
        stop(
            "There must at least as many `levels` as possible values in `x`",
            call. = FALSE
        )
    }

    x
}
```

```
validate_factor(new_factor(1:5, "a"))
#> Error: There must at least as many `levels` as possible values in
#> `x`
validate_factor(new_factor(0:1, "a"))
#> Error: All `x` values must be non-missing and greater than zero
```

Эта функция-валидатор в основном используется ради побочных эффектов (выброса ошибки в случае некорректности объекта), так что можно было бы ожидать, что она будет невидимо возвращать переданный ей аргумент. Однако, как вы увидите дальше, полезнее осуществлять возврат видимым образом.

13.3.3 Помощники

Если вам необходимо, чтобы пользователи сами создавали объекты на основе вашего класса, будет удобно снабдить класс функцией-помощником – это облегчит жизнь пользователям. Помощник должен отвечать следующим требованиям:

- иметь такое же имя, как и имя класса, например `myclass()`;
- заканчиваться вызовом конструктора и, если он есть, валидатора;
- содержать дружественную систему сообщений, ориентированную на пользователя;
- располагать удобным для пользователя интерфейсом с подходящими значениями по умолчанию и полезными преобразованиями.

Последний пункт бывает выполнить непросто, и какие-то общие рекомендации здесь дать затруднительно. Но мы можем рассмотреть три возможных шаблона:

- иногда все, что должен делать помощник, – это приводить входные параметры к нужным типам. К примеру, конструктор `new_difftime()` является довольно строгим, нарушая при этом принятое соглашение о том, что можно использовать целочисленный вектор везде, где используется вектор двойной точности:

  ```
  new_difftime(1:10)
  #> Error in new_difftime(1:10): is.double(x) is not TRUE
  ```

  Но конструктор и не должен обладать гибкостью, поэтому мы снабдим класс функцией-помощником, которая будет приводить входное значение к целочисленному типу двойной точности.

  ```
  difftime <- function(x = double(), units = "secs") {
      x <- as.double(x)
      new_difftime(x, units = units)
  }
  ```

```
difftime(1:10)
#> Time differences in secs
#> [1]  1  2  3  4  5  6  7  8  9 10
```

- зачастую наиболее естественным представлением сложного объекта является строка. К примеру, бывает очень удобно задавать факторы с помощью строкового вектора. Ниже представлена упрощенная версия `factor()`: на вход функция принимает символьный вектор и предполагает, что в качестве уровней должны быть уникальные значения. Это не всегда так (поскольку некоторые уровни могут не присутствовать в указанных данных), но для значения по умолчанию сгодится.

```
factor <- function(x = character(), levels = unique(x)) {
    ind <- match(x, levels)
    validate_factor(new_factor(ind, levels))
}

factor(c("a", "a", "b"))
#> [1] a a b
#> Levels: a b
```

- некоторые сложные объекты бывает удобно собирать из нескольких простых компонентов. К примеру, объект, представляющий дату и время, можно задать с помощью года, месяца, дня и т. д. Эта мысль привела к написанию такой функции-помощника `POSIXct()`, напоминающей существующую функцию `ISODatetime()`[1].

```
POSIXct <- function(year = integer(),
                    month = integer(),
                    day = integer(),
                    hour = 0L,
                    minute = 0L,
                    sec = 0,
                    tzone = "") {
    ISOdatetime(year, month, day, hour, minute, sec, tz = tzone)
}

POSIXct(2020, 1, 1, tzone = "America/New_York")
#> [1] "2020-01-01 EST"
```

Для более сложных классов можно пойти и гораздо дальше этих простых шаблонов, чтобы максимально облегчить жизнь пользователям.

---

[1] Этот помощник не отличается эффективностью: за кулисами функция `ISODatetime()` собирает поступившие компоненты в строку и затем вызывает функцию `strptime()`. Более эффективный аналог этой функции можно увидеть на примере `lubridate::make_datetime()`.

13.3.4 Упражнения

1. Напишите конструктор для объектов `data.frame`. На основе какого базового типа построены датафреймы? Какие атрибуты в них используются? Какие ограничения налагаются на отдельные элементы? А как насчет имен?
2. Расширьте написанную ранее функцию-помощник `factor()`, чтобы она более грамотно обрабатывала ситуации, когда одно или более значений не присутствуют в числе уровней. Что в таких случаях делает функция `base::factor()`?
3. Внимательно изучите код `factor()`. Что в нем происходит такого, чего не делает мой конструктор?
4. В факторах присутствует необязательный атрибут `contrasts`. Ознакомьтесь с инструкцией к классу `C()` и кратко опишите назначение этого атрибута. Какого типа он должен быть? Перепишите конструктор `new_factor()` таким образом, чтобы он включал этот атрибут.
5. Прочитайте документацию к `utils::as.roman()`. Как бы вы написали конструктор для этого класса? Нуждается ли он в валидаторе? Что может происходить в функции-помощнике?

13.4 Обобщенные функции и методы

Назначение обобщенной функции S3 состоит в осуществлении диспетчеризации методов, т. е. поиске подходящей реализации для класса. Диспетчеризация методов выполняется с помощью функции `UseMethod()`, к которой обращается каждая обобщенная функция[1]. Функция `UseMethod()` принимает два аргумента: имя обобщенной функции (обязательный параметр) и аргумент, используемый для диспетчеризации (необязательный). В отсутствие второго аргумента диспетчеризация будет проходить на основании первого аргумента, что нас почти всегда будет устраивать.

Большинство обобщенных функций очень простые и содержат только вызов функции `UseMethod()`. Рассмотрим для примера функцию `mean()`:

```
mean
#> function (x, ...)
#> UseMethod("mean")
#> <bytecode: 0x7fed45725430>
#> <environment: namespace:base>
```

[1] Исключение составляют внутренние обобщенные функции, реализованные на C, о которых мы будем говорить в разделе 13.7.2.

Создать собственную обобщенную функцию можно очень просто:

```
my_new_generic <- function(x) {
    UseMethod("my_new_generic")
}
```

Если вам интересно, почему мы дважды вынуждены были написать `my_new_generic`, обратитесь за разъяснениями к разделу 6.2.3.

Мы не передаем никакие аргументы обобщенной функции в функцию `UseMethod()` – это происходит автоматически при помощи черной магии. Сам этот процесс весьма непрост и зачастую удивителен, так что вам лучше избегать каких-то сложных вычислений в обобщенных функциях. Для получения более подробной информации обратитесь к документации `?UseMethod`.

13.4.1 Диспетчеризация методов

Как же работает функция `UseMethod()`? По сути, в ней создается вектор имен методов (`paste0("generic", ".", c(class(x), "default"))`), после чего осуществляется поиск по методам. Этот механизм в действии можно рассмотреть с помощью функции `sloop::s3_dispatch()`. Вы передаете этой функции вызов обобщенной функции S3, а она возвращает список возможных для использования методов. К примеру, так можно узнать, какой метод применяется для вывода объекта `Date`:

```
x <- Sys.Date()
s3_dispatch(print(x))
#> => print.Date
#>  * print.default
```

Обозначения здесь используются следующие:

- символ `=>` указывает на метод, выбранный для вызова. Здесь это `print.Date()`;
- символ `*` указывает на найденный, но не выбранный для вызова метод. Здесь это `print.default()`.

Класс `default` является *псевдоклассом* (pseudo-class). Это не настоящий класс, но он включается в список в качестве запасного варианта, на случай если специфический для класса метод недоступен.

Сама по себе диспетчеризация методов довольно проста, но в процессе чтения главы вы увидите, что этот процесс будет усложняться с целью включения наследования, базовых типов, внутренних и групповых обобщенных функций. Ниже показана пара примеров более сложной диспетчеризации, к которым мы вернемся в разделах 14.2.4 и 13.7.

```
x <- matrix(1:10, nrow = 2)
s3_dispatch(mean(x))
#>    mean.matrix
```

```
#>    mean.integer
#>    mean.numeric
#> => mean.default

s3_dispatch(sum(Sys.time()))
#>    sum.POSIXct
#>    sum.POSIXt
#>    sum.default
#> => Summary.POSIXct
#>    Summary.POSIXt
#>    Summary.default
#> -> sum (internal)
```

13.4.2 Поиск методов

Функция `sloop::s3_dispatch()` позволяет определить конкретный метод, выбранный для единственного вызова. А что, если вам необходимо найти все методы, определенные для обобщенной функции или связанные с классом? Это работа для функций `sloop::s3_methods_generic()` и `sloop::s3_methods_class()`:

```
s3_methods_generic("mean")
#> # A tibble: 6 x 4
#>   generic class    visible source
#>   <chr>   <chr>    <lgl>   <chr>
#> 1 mean    Date     TRUE    base
#> 2 mean    default  TRUE    base
#> 3 mean    difftime TRUE    base
#> 4 mean    POSIXct  TRUE    base
#> 5 mean    POSIXlt  TRUE    base
#> 6 mean    quosure  FALSE   registered S3method

s3_methods_class("ordered")
#> # A tibble: 4 x 4
#>   generic       class   visible source
#>   <chr>         <chr>   <lgl>   <chr>
#> 1 as.data.frame ordered TRUE    base
#> 2 Ops           ordered TRUE    base
#> 3 relevel       ordered FALSE   registered S3method
#> 4 Summary       ordered TRUE    base
```

13.4.3 Создание методов

Есть два тонких момента, о которых необходимо помнить при создании новых методов:

- вам следует писать метод только в том случае, если вы являетесь владельцем обобщенной функции или класса. R позволит вам определить метод, даже если вы не являетесь владельцем, но это считается дурным

тоном. Вместо того чтобы так делать, лучше связаться с автором обобщенной функции или класса для добавления метода;

- метод должен располагать теми же аргументами, что и обобщенная функция. При работе с пакетами этот момент автоматически отслеживается с помощью инструкции `R CMD check`, но вам необходимо следить за этим даже тогда, когда вы не создаете пакет. Из этого правила есть одно исключение: если в обобщенной функции присутствует аргумент `...`, то метод может содержать в себе расширенный набор аргументов. Это позволяет методам принимать дополнительные произвольные аргументы. Недостатком использования `...` является то, что в этом случае любые аргументы, написанные с ошибками, будут безмолвно приняты[1], о чем мы уже писали в разделе 6.6.

13.4.4 Упражнения

1. Ознакомьтесь с исходным кодом функций `t()` и `t.test()` и проверьте, что `t.test()` является обобщенной функцией S3, а не методом S3. Что произойдет, если создать объект класса `test` и вызвать для него `t()`? Почему?

```
x <- structure(1:10, class = "test")
t(x)
```

2. Для каких обобщенных функций есть методы в классе `table`?
3. Для каких обобщенных функций есть методы в классе `ecdf`?
4. Какая базовая обобщенная функция насчитывает самое большое количество определенных методов?
5. Внимательно ознакомьтесь с документацией к функции `UseMethod()` и объясните, почему код, приведенный ниже, возвращает такой результат. Какие два принятых правила вычисления функций нарушает функция `UseMethod()`?

```
g <- function(x) {
    x <- 10
    y <- 10
    UseMethod("g")
}
g.default <- function(x) c(x = x, y = y)

x <- 1
y <- 1
g(x)
```

[1] По ссылке https://github.com/r-lib/ellipsis вы можете ознакомиться с инструментом, позволяющим генерировать предупреждения в случаях, когда не все аргументы, переданные с помощью `...`, используются.

```
#> x y
#> 1 10
```

6. Каковы аргументы функции [? Почему на этот вопрос так трудно ответить?

13.5 Стили объектов

До сих пор мы вели речь о классах в стиле векторов вроде `Date` и `factor`. У всех у них присутствует ключевое свойство `length(x)`, показывающее количество наблюдений в векторе. Существует еще три стиля объектов, в которых это свойство отсутствует:

- объекты в стиле записей используют список векторов одинаковой длины для представления отдельных компонентов. Одним из примеров является класс `POSIXlt`, под капотом которого хранится список из 11 компонентов даты и времени вроде года, месяца, дня и т. д. Классы в стиле записей переопределяют `length()` и методы извлечения подмножеств для скрытия деталей реализации;

```
x <- as.POSIXlt(ISOdatetime(2020, 1, 1, 0, 0, 1:3))
x
#> [1] "2020-01-01 00:00:01 CST" "2020-01-01 00:00:02 CST"
#> [3] "2020-01-01 00:00:03 CST"

length(x)
#> [1] 3
length(unclass(x))
#> [1] 11

x[[1]] # первый элемент
#> [1] "2020-01-01 00:00:01 CST"
unclass(x)[[1]] # первый компонент, количество секунд
#> [1] 1 2 3
```

- объекты в стиле датафреймов похожи на объекты в стиле записей в том, что они также используют список векторов одинаковой длины. При этом концептуально датафреймы являются двумерными, а их отдельные компоненты легко доступны пользователям. Количество наблюдений в данном случае – это количество строк, а не `length`:

```
x <- data.frame(x = 1:100, y = 1:100)
length(x)
#> [1] 2
nrow(x)
#> [1] 100
```

- объекты в стиле скаляров используют список для представления одной сущности. К примеру, объект `lm` представляет собой список длины 12, но при этом описывает одну модель.

```
mod <- lm(mpg ~ wt, data = mtcars)
length(mod)
#> [1] 12
```

Скалярные объекты могут быть также построены на базе функций, вызовов и окружений[1]. Обычно это не так полезно, но некоторые примеры можно найти в `stats::ecdf()` и R6 (глава 14), а также в `rlang::quo()` (глава 19).

К сожалению, подробное описание примеров объектов каждого стиля выходит за рамки данной книги. Но вы можете узнать все детали в документации к пакету `vctrs` (https://vctrs.r-lib.org). В этом пакете также представлены конструкторы и помощники, облегчающие реализацию различных стилей.

13.5.1 Упражнения

1. Разделите объекты, возвращаемые функциями `lm()`, `factor()`, `table()`, `as.Date()`, `as.POSIXct()`, `ecdf()`, `ordered()` и `I()`, по стилям, перечисленным выше.
2. Как может выглядеть конструктор класса `lm` (`new_lm()`)? Воспользуйтесь командой `?lm` и поэкспериментируйте с определением требуемых полей и их типов.

13.6 Наследование

Классы S3 могут обмениваться поведением посредством механизма *наследования* (inheritance). В основе наследования лежат три идеи:

- атрибут класса может быть представлен символьным *вектором*. К примеру, классы `ordered` и `POSIXct` насчитывают по два компонента в атрибуте `class`:

```
class(ordered("x"))
#> [1] "ordered" "factor"
class(Sys.time())
#> [1] "POSIXct" "POSIXt"
```

- если метод не может быть найден для первого элемента этого вектора, R приступает к его поиску для второго элемента и т. д.:

[1] Объекты также можно создавать на основе *списков пар* (pairlist), но я пока не нашел ни одной причины это делать.

```
s3_dispatch(print(ordered("x")))
#>    print.ordered
#> => print.factor
#>  * print.default
s3_dispatch(print(Sys.time()))
#> => print.POSIXct
#>    print.POSIXt
#>  * print.default
```

- методы могут делегировать свою работу с помощью функции `NextMethod()`. Очень скоро мы поговорим об этом подробнее, а сейчас просто отметьте для себя, что функция `s3_dispatch()` сообщает о делегировании методов с помощью сочетания символов `->`.

```
s3_dispatch(ordered("x")[1])
#>    [.ordered
#> => [.factor
#>    [.default
#> -> [ (internal)
s3_dispatch(Sys.time()[1])
#> => [.POSIXct
#>    [.POSIXt
#>    [.default
#> -> [ (internal)
```

Перед тем как продолжить, нам необходимо пополнить словарь терминами для описания взаимосвязей между классами, совместно присутствующими в векторе-атрибуте `class`. Мы будем называть класс `ordered` *подклассом* (subclass), или дочерним классом, по отношению к классу `factor`, поскольку в векторе `class` он всегда располагается перед ним. И наоборот, класс `factor` мы будем именовать *суперклассом* (superclass), или родительским классом, по отношению к `ordered`.

Система S3 не накладывает никаких ограничений на взаимосвязи между подклассами и суперклассами, но вы существенно облегчите себе жизнь, если будете придерживаться определенных правил. Лично я рекомендовал бы вам следовать двум простым принципам при создании подклассов:

- базовый тип подкласса должен быть таким же, как у суперкласса;
- атрибуты подкласса должны представлять собой надмножество атрибутов суперкласса.

В случае с классом `POSIXt` мы не наблюдаем строгого следования этим принципам, поскольку типом класса `POSIXct` является `double`, а типом `POSIXlt` – список. Это означает, что `POSIXt` нельзя назвать суперклассом, и на этом примере показана возможность реализации с помощью системы наследования S3 других принципов программирования (в данном случае класс `POSIXt`, скорее, играет роль интерфейса). В целом же вы сами должны определить для себя допустимые рамки использования наследования с точки зрения безопасности и целостности.

13.6.1 NextMethod()

Использование функции NextMethod() представляет одну из самых сложных составляющих наследования, так что мы начнем с конкретного примера для наиболее распространенного случая: [. Создадим простой класс secret, который будет скрывать свое содержимое при выводе на экран:

```
new_secret <- function(x = double()) {
    stopifnot(is.double(x))
    structure(x, class = "secret")
}

print.secret <- function(x, ...) {
    print(strrep("x", nchar(x)))
    invisible(x)
}

x <- new_secret(c(15, 1, 456))
x
#> [1] "xx"  "x"   "xxx"
```

Это работает, но метод [по умолчанию не сохраняет класс:

```
s3_dispatch(x[1])
#>    [.secret
#>    [.default
#> => [ (internal)
x[1]
#> [1] 15
```

Чтобы поправить это, необходимо реализовать метод [.secret. Как мы можем это сделать? Самый простой способ не сработает, поскольку мы просто увязнем в бесконечном цикле:

```
`[.secret` <- function(x, i) {
    new_secret(x[i])
}
```

Вместо этого нам нужно как-то вызвать код, лежащий в основе [, т. е. ту реализацию, которая была бы вызвана, если бы у нас не было метода [.secret. Один из таких способов связан с использованием функции unclass():

```
`[.secret` <- function(x, i) {
    x <- unclass(x)
    new_secret(x[i])
}
x[1]
#> [1] "xx"
```

Это сработает, но данный подход не является эффективным, поскольку будет создана копия x. Гораздо лучше будет воспользоваться функцией NextMethod(), которая поможет элегантно решить возникшую проблему посредством делегирования методу, который был бы вызван, если бы метода [.secret не существовало:

```
`[.secret` <- function(x, i) {
    new_secret(NextMethod())
}
x[1]
#> [1] "xx"
```

Вызовите функцию sloop::s3_dispatch() и посмотрите, что произошло:

```
s3_dispatch(x[1])
#> => [.secret
#>    [.default
#> -> [ (internal)
```

Знак => указывает на то, что был вызван метод [.secret, но функция NextMethod() делегировала работу соответствующему *внутреннему* (internal) методу [, что видно по индикатору ->.

Как и в случае с функцией UseMethod(), точная семантика функции NextMethod() достаточно сложна. В частности, она отслеживает перечень потенциально следующих методов для вызова с помощью специальной переменной, а это значит, что изменение диспетчеризуемого объекта не повлияет на то, какой метод будет вызван следующим.

13.6.2 Разрешение создавать подклассы

При создании класса вы должны решить, будет ли возможность создавать его подклассы, поскольку это требует внесения определенных корректировок в конструктор и внимательной работы с методами.

Чтобы позволить создавать подклассы на основе существующего класса, необходимо, чтобы конструктор родительского класса принимал аргументы ... и class:

```
new_secret <- function(x, ..., class = character()) {
    stopifnot(is.double(x))

    structure(
        x,
        ...,
        class = c(class, "secret")
    )
}
```

После этого подкласс сможет при необходимости обращаться к родительскому конструктору с помощью дополнительных аргументов. К примеру,

представьте, что вы хотите создать сверхсекретный класс, который будет, помимо содержимого, скрывать и количество символов:

```
new_supersecret <- function(x) {
    new_secret(x, class = "supersecret")
}

print.supersecret <- function(x, ...) {
    print(rep("xxxxx", length(x)))
    invisible(x)
}

x2 <- new_supersecret(c(15, 1, 456))
x2
#> [1] "xxxxx" "xxxxx" "xxxxx"
```

Для осуществления наследования вам также следует тщательно продумать свои методы, поскольку вы больше не сможете использовать конструктор. Если вы это сделаете, ваш метод всегда будет возвращать один и тот же класс вне зависимости от входа. Это добавляет работы разработчику подкласса.

Если говорить конкретно, нам необходимо пересмотреть метод `[.secret`. На данный момент он всегда возвращает `secret()`, даже если ему передать объект `supersecret`:

```
`[.secret` <- function(x, ...) {
    new_secret(NextMethod())
}

x2[1:3]
#> [1] "xx" "x" "xxx"
```

Нам нужно убедиться, что метод `[.secret` будет возвращать тот же класс, что и `x`, даже если это подкласс. Насколько я знаю, пока эту проблему невозможно решить исключительно средствами базового R. Вместо этого вам придется воспользоваться пакетом `vctrs`, который предлагает решение в виде обобщенной функции `vctrs::vec_restore()`. Эта функция принимает два аргумента: объект, утративший информацию о подклассе, и шаблонный объект, который будет использован в процессе восстановления.

Обычно методы `vec_restore()` бывают очень простыми: в них просто вызывается конструктор с соответствующими аргументами:

```
vec_restore.secret <- function(x, to, ...) new_secret(x)
vec_restore.supersecret <- function(x, to, ...) new_supersecret(x)
```

(Если в вашем классе присутствуют атрибуты, вам необходимо передать их из аргумента `to` в конструктор.)

Теперь можно воспользоваться функцией `vec_restore()` в методе `[.secret`:

```
`[.secret` <- function(x, ...) {
    vctrs::vec_restore(NextMethod(), x)
}
x2[1:3]
#> [1] "xxxxx" "xxxxx" "xxxxx"
```

(До конца я осознал эту проблему лишь недавно, так что на момент написания книги она не была учтена в пакетах tidyverse. Надеюсь, когда вы будете читать эти строки, все будет реализовано, что облегчит вам, например, наследование тибблов.)

Если вы создаете класс с помощью инструментов из пакета `vctrs`, метод `[` подхватит это поведение автоматически. Вам нужно будет предоставить собственный метод `[`, только если вы используете атрибуты, зависящие от данных, или желаете реализовать нестандартное поведение при извлечении подмножеств. Подробности вы можете узнать в документации, доступной по команде `?vctrs::new_vctr`.

13.6.3 Упражнения

1. Как `[.Date` поддерживает подклассы? И как он их не поддерживает?
2. В R присутствуют два класса для представления даты и времени: `POSIXct` и `POSIXlt`, и оба они наследуются от класса `POSIXt`. Какие обобщенные функции отличаются поведением в двух этих классах? А какие ведут себя одинаково?
3. Как вы думаете, что вернет приведенный ниже код? И что он вернул на самом деле? Почему?

```
generic2 <- function(x) UseMethod("generic2")
generic2.a1 <- function(x) "a1"
generic2.a2 <- function(x) "a2"
generic2.b <- function(x) {
    class(x) <- "a1"
    NextMethod()
}

generic2(structure(list(), class = c("b", "a2")))
```

13.7 Детали диспетчеризации методов

В данном заключительном разделе главы мы рассмотрим некоторые подробности реализации диспетчеризации методов в R. Если вы только знакомитесь с системой S3, можете смело пропустить его.

13.7.1 S3 и базовые типы

Что произойдет, если вызвать обобщенную функцию S3 с базовым объектом, т. е. с объектом без класса? Вы могли бы подумать, что диспетчеризация будет происходить исходя из значения, возвращаемого функцией `class()`:

```
class(matrix(1:5))
#> [1] "matrix"
```

Но, к сожалению, диспетчеризация в этом случае будет основываться на *неявном классе* (implicit class), включающем три компонента:

- строка `"array"` или `"matrix"`, если у объекта есть измерения;
- результат функции `typeof()` с небольшими поправками;
- строка `"numeric"`, если объект является `"integer"` или `"double"`.

В базовом пакете нет функций, способных вычислять неявные классы, но вы можете воспользоваться функцией `sloop::s3_class()`:

```
s3_class(matrix(1:5))
#> [1] "matrix"  "integer" "numeric"
```

Ее, в свою очередь, использует функция `s3_dispatch()`:

```
s3_dispatch(print(matrix(1:5)))
#>    print.matrix
#>    print.integer
#>    print.numeric
#> => print.default
```

Это означает, что `class()` объекта не является уникальным определителем диспетчеризации:

```
x1 <- 1:5
class(x1)
#> [1] "integer"
s3_dispatch(mean(x1))
#>    mean.integer
#>    mean.numeric
#> => mean.default

x2 <- structure(x1, class = "integer")
class(x2)
#> [1] "integer"
s3_dispatch(mean(x2))
#>    mean.integer
#> => mean.default
```

13.7.2 Внутренние обобщенные функции

Некоторые базовые функции, такие как `[`, `sum()` и `cbind()`, называются *внутренними обобщенными функциями* (internal generic), поскольку они не используют функцию `UseMethod()`, а вместо этого вызывают функцию C `DispatchGroup()` или `DispatchOrEval()`. Функция `s3_dispatch()` показывает внутренние обобщенные функции путем включения имени функции, следом за которым идет слово `internal`:

```
s3_dispatch(Sys.time()[1])
#> => [.POSIXct
#>    [.POSIXt
#>    [.default
#> -> [ (internal)
```

В целях повышения производительности внутренние обобщенные функции не диспетчеризуются по методам, если у них не установлен атрибут `class`. Это означает, что такие функции не используют неявные классы. Так или иначе, если вы сомневаетесь в том, как будет выполняться диспетчеризация методов, вы всегда можете положиться на функцию `s3_dispatch()`.

13.7.3 Групповые обобщенные функции

Групповые обобщенные функции (group generic) представляют самую сложную составляющую процесса диспетчеризации методов, поскольку в них используется и функция `NextMethod()`, и внутренние обобщенные функции. Подобно внутренним функциям, они присутствуют только в базовом R, и вы не можете определить собственную групповую обобщенную функцию.

Существует четыре вида групповых обобщенных функций:

- *математические* (Math): `abs()`, `sign()`, `sqrt()`, `floor()`, `cos()`, `sin()`, `log()` и другие (полный список можно найти в документации `?Math`);
- *операторы* (Ops): `+`, `-`, `*`, `/`, `^`, `%%`, `%/%`, `&`, `|`, `!`, `==`, `!=`, `<`, `<=`, `>=` и `>`;
- *агрегирующие* (Summary): `all()`, `any()`, `sum()`, `prod()`, `min()`, `max()` и `range()`;
- *комплексные* (Complex): `Arg()`, `Conj()`, `Im()`, `Mod()`, `Re()`.

Объявление одной групповой обобщенной функции для вашего класса переопределит поведение по умолчанию для всех членов группы. Поиск методов для групповой обобщенной функции будет осуществляться только в случае, если не будет найден метод для конкретной обобщенной функции:

```
s3_dispatch(sum(Sys.time()))
#>    sum.POSIXct
#>    sum.POSIXt
#>    sum.default
#> => Summary.POSIXct
#>    Summary.POSIXt
```

```
#>    Summary.default
#> -> sum (internal)
```

Большинство групповых обобщенных функций включают в себя вызов функции `NextMethod()`. Возьмем для примера объекты `difftime()`. Если взглянуть на диспетчеризацию методов для `abs()`, можно заметить присутствие определения групповой обобщенной функции `Math`.

```
y <- as.difftime(10, units = "mins")
s3_dispatch(abs(y))
#>    abs.difftime
#>    abs.default
#> => Math.difftime
#>    Math.default
#> -> abs (internal)
```

Функция `Math.difftime` выглядит так:

```
Math.difftime <- function(x, ...) {
    new_difftime(NextMethod(), units = attr(x, "units"))
}
```

Для выполнения вычисления эта функция диспетчеризуется на следующий метод – здесь это внутренний метод по умолчанию, – после чего восстанавливает класс и атрибуты. (Для лучшей поддержки подклассов `difftime` здесь необходимо было бы вызвать функцию `vec_restore()`, как было показано в разделе 13.6.2.)

Внутри групповой обобщенной функции есть специальная переменная `.Generic`, с помощью которой можно узнать, какая именно обобщенная функция была вызвана. Это может быть полезно при создании сообщений об ошибках, а также при возникновении необходимости повторно вызвать обобщенную функцию с другими аргументами.

13.7.4 Двойная диспетчеризация методов

В обобщенных функциях из группы Ops, включающих в себя арифметические и логические операторы с двумя аргументами, таких как `-` и `&`, реализован особый способ диспетчеризации методов. Он базируется на типах *обоих* аргументов и называется *двойной диспетчеризацией* (double dispatch). Применение особого способа диспетчеризации необходимо для сохранения свойства коммутативности многих операторов. Например, результаты выражений `a + b` и `b + a` должны быть равны. Рассмотрим следующий пример:

```
date <- as.Date("2017-01-01")
integer <- 1L

date + integer
```

```
#> [1] "2017-01-02"
integer + date
#> [1] "2017-01-02"
```

Если бы оператор + диспетчеризовался исключительно исходя из первого аргумента, он вернул бы разные значения для двух случаев. Во избежание этого в обобщенных функциях из группы `Ops` используется несколько иная стратегия. Вместо одной диспетчеризации выполняется две – по одной для каждого входа. В результате поиска может возникнуть три разные ситуации:

- методы окажутся одинаковыми, и не будет никакой разницы, какой использовать;
- методы окажутся разными, и R воспользуется внутренним методом с предупреждением;
- один из методов окажется внутренним, и R выберет другой.

Этот подход не является надежным, так что если вам необходимо реализовать двойную диспетчеризацию для алгебраических операторов, я рекомендую использовать пакет `vctrs`. За подробностями вы можете обратиться к документации по команде `?vctrs::vec_arith`.

13.7.5 Упражнения

1. Опишите разницу в диспетчеризации в коде, показанном ниже:

```
length.integer <- function(x) 10

x1 <- 1:5
class(x1)
#> [1] "integer"
s3_dispatch(length(x1))
#>  * length.integer
#>    length.numeric
#>    length.default
#> => length (internal)

x2 <- structure(x1, class = "integer")
class(x2)
#> [1] "integer"
s3_dispatch(length(x2))
#> => length.integer
#>    length.default
#>  * length (internal)
```

2. У каких классов в базовом R есть метод для групповой обобщенной функции `Math`? Почитайте исходный код. Как работают методы?
3. Метод `Math.difftime()` является более сложным по сравнению с тем, как я его описал. Почему?

14

Система R6

14.1 Введение

Эта глава будет посвящена системе ООП *R6*. Эта система обладает двумя важными свойствами:

- в ней используется парадигма инкапсулирования, а это означает, что методы принадлежат напрямую объектам, а не обобщенным функциям, и их вызов осуществляется следующим образом: `object$method()`;
- объекты R6 изменяемые, т. е. они модифицируются на месте, что обусловливает их ссылочную семантику.

Если вы работали с другими объектно ориентированными языками программирования, система R6 покажется вам более знакомой, и вам наверняка захочется использовать именно ее, а не S3. Но не стоит идти по пути наименьшего сопротивления: в большинстве случаев применение R6 приводит к образованию кода, не свойственного для языка R. Мы вернемся к этой теме в разделе 16.3.

Система R6 очень схожа с базовой системой ООП, построенной с использованием *классов на основе ссылок* (reference classes – RC). В разделе 14.5 я поясню, почему в этой книге сделал выбор в пользу описания системы R6, а не RC.

Структура главы

- В разделе 14.2 мы познакомимся с функцией `R6::R6class()` – единственной функцией, которую вам необходимо знать для создания классов R6. Вы узнаете о конструкторе `$new()`, позволяющем создавать объекты R6, а также о других важных методах, включая `$initialize()` и `$print()`.
- В разделе 14.3 речь пойдет о механизмах доступа R6, а именно о приватных и активных полях. Вместе эти концепции позволяют скрыть чувствительные данные от пользователя или разрешить ему чтение приватных данных, но не запись.
- В разделе 14.4 мы рассмотрим некоторые последствия использования ссылочной семантики в системе R6. Вы узнаете, как использовать финализаторы (методы завершения), позволяющие автоматически подчищать операции, выполненные в инициализаторе, а также познако-

митесь с типичными ошибками, возникающими при использовании объектов R6 в качестве полей других объектов R6.

- В разделе 14.5 я расскажу, почему в этой книге я сделал выбор в пользу описания системы R6, а не RC.

Требования

Поскольку система R6 (https://r6.r-lib.org) не встроена в базовый R, для работы с ней вам необходимо будет установить и загрузить пакет R6:

```
# install.packages("R6")
library(R6)
```

Объекты R6 обладают ссылочной семантикой, что позволяет изменять их на месте, без использования концепции *копирования при изменении* (copy-on-modify). Если вы незнакомы с этой терминологией, освежите свои знания, прочитав раздел 2.5.

14.2 Классы и методы

В системе R6 вы можете с помощью всего одной функции `R6::R6Class()` создать и класс, и его методы. Это единственная функция из пакета, к помощи которой вы будете прибегать[1]!

В примере ниже показаны два важнейших аргумента функции `R6Class()`:

- первый аргумент – `classname`. В нем нет строгой необходимости, но он помогает сделать сообщения об ошибках более понятными и позволяет использовать объекты R6 совместно с обобщенными функциями S3. По общему соглашению имена классов R6 должны иметь слитное написание слов с заглавными буквами («верблюжий» стиль);
- второй аргумент – `public` – отвечает за список методов (функций) и полей (все остальное), составляющих публичный интерфейс объекта. По общему соглашению имена методов и полей записываются с использованием символа подчеркивания («змеиный» стиль). Методы могут обращаться к методам и полям текущего объекта с помощью предопределенного слова `self`[2].

```
Accumulator <- R6Class("Accumulator", list(
    sum = 0,
    add = function(x = 1) {
```

[1] Это значит, что при создании R6 в пакете вам необходимо будет убедиться только в том, что он присутствует в разделе `Imports` файла `DESCRIPTION`. При этом нет необходимости импортировать пакет в файле `NAMESPACE`.

[2] В отличие от Python, переменная `self` автоматически предоставляется R6 и не является частью сигнатуры метода.

```
        self$sum <- self$sum + x
        invisible(self)
    })
)
```

Вы всегда должны присваивать результат функции `R6Class()` переменной с таким же именем, как у класса, поскольку эта функция возвращает объект R6, определяющий класс:

```
Accumulator
#> <Accumulator> object generator
#>   Public:
#>     sum: 0
#>     add: function (x = 1)
#>     clone: function (deep = FALSE)
#>   Parent env: <environment: R_GlobalEnv>
#>   Locked objects: TRUE
#>   Locked class: FALSE
#>   Portable: TRUE
```

Новый объект на основе класса создается с помощью метода `$new()`. В R6 методы принадлежат объектам, так что для доступа к ним необходимо использовать символ `$`:

```
x <- Accumulator$new()
```

В дальнейшем вы также можете обращаться к методам и полям объекта при помощи символа `$`:

```
x$add(4)
x$sum
#> [1] 4
```

В данном классе все методы и поля являются публичными, что позволяет вам извне получать и устанавливать значения любых полей. Позже мы узнаем, как можно использовать приватные поля и методы классов для ограничения свободного доступа к ним.

Для большей ясности при упоминании полей и методов, в отличие от переменных и функций, я буду предварять их имена символом `$`. Таким образом, класс `Accumulate` содержит поле `$sum` и метод `$add()`.

14.2.1 Цепочки методов

Метод `$add()` вызывается главным образом ради побочного эффекта, заключающегося в обновлении поля `$sum`.

```
Accumulator <- R6Class("Accumulator", list(
    sum = 0,
```

```
    add = function(x = 1) {
        self$sum <- self$sum + x
        invisible(self)
    })
)
```

Методы R6 с побочными эффектами всегда должны возвращать `self` невидимым образом. Это приводит к возврату «текущего» объекта и позволяет объединять в цепочки несколько вызовов, как показано ниже:

```
x$add(10)$add(10)$sum
#> [1] 24
```

Для лучшей читаемости вы можете писать каждый вызов функции на отдельной строке:

```
x$
    add(10)$
    add(10)$
    sum
#> [1] 44
```

Такая техника называется *сцеплением методов* (method chaining) и часто используется в языках вроде Python и JavaScript. Процесс сцепления методов очень похож на создание конвейеров, и в разделе 16.3.3 мы подробно обсудим преимущества и недостатки обоих способов.

14.2.2 Важные методы

Существует два важнейших метода, которые должны быть определены для большинства классов: `$initialize()` и `$print()`. Эти методы не являются обязательными, но их создание значительно облегчит использование ваших классов.

Метод `$initialize()` переопределяет поведение по умолчанию метода `$new()`. К примеру, с помощью приведенного ниже кода вы можете объявить класс `Person` с полями `$name` и `$age`. Для гарантии того, что в поле `$name` у нас всегда будет одно строковое значение, а в поле `$age` – одно числовое, можно разместить необходимые проверки в методе `$initialize()`.

```
Person <- R6Class("Person", list(
    name = NULL,
    age = NA,
    initialize = function(name, age = NA) {
        stopifnot(is.character(name), length(name) == 1)
        stopifnot(is.numeric(age), length(age) == 1)

        self$name <- name
        self$age <- age
```

```
    }
))

hadley <- Person$new("Hadley", age = "thirty-eight")
#> Error in .subset2(public_bind_env, "initialize")(...):
#> is.numeric(age) is not TRUE

hadley <- Person$new("Hadley", age = 38)
```

Если вам требуется реализовать более ресурсоемкие проверки, разместите их в отдельном методе `$validate()` и вызывайте его при необходимости.

Определение метода `$print()` позволяет переопределить поведение по умолчанию для вывода на экран. Как и в случае с любыми другими методами, вызываемыми ради побочных эффектов, метод `$print()` должен завершаться невидимым возвратом объекта `invisible(self)`.

```
Person <- R6Class("Person", list(
    name = NULL,
    age = NA,
    initialize = function(name, age = NA) {
        self$name <- name
        self$age <- age
    },
    print = function(...) {
        cat("Person: \n")
        cat(" Name: ", self$name, "\n", sep = "")
        cat(" Age: ", self$age, "\n", sep = "")
        invisible(self)
    }
))

hadley2 <- Person$new("Hadley")
hadley2
#> Person:
#>   Name: Hadley
#>   Age:  NA
```

Этот код иллюстрирует важную особенность системы R6. Поскольку все методы у нас привязаны к объектам, ранее созданный объект `hadley` не будет располагать этим новым методом:

```
hadley
#> <Person>
#>   Public:
#>     age: 38
#>     clone: function (deep = FALSE)
#>     initialize: function (name, age = NA)
#>     name: Hadley
```

```
hadley$print
#> NULL
```

С точки зрения системы R6 между объектами `hadley` и `hadley2` нет никакой взаимосвязи: они лишь по случайному совпадению связаны одним именем класса. Это не доставляет проблем при использовании ранее разработанных объектов R6, но может мешать при проведении интерактивных экспериментов. Если вы меняете код и не понимаете, почему при этом не меняется поведение объекта, убедитесь, что вы пересоздали этот объект после внесения изменений в класс.

14.2.3 Добавление методов после создания класса

Помимо создания новых классов, у вас есть возможность изменения полей и методов в существующих классах. Это бывает очень удобно в процессе отладки класса или при наличии множества функций в классе, с которыми вы хотите работать по отдельности. Добавить новые элементы в класс можно с помощью метода `$set()`, передав ему в качестве аргументов признак видимости элемента (подробнее об этом мы будем говорить в разделе 14.3), имя и само содержимое.

```
Accumulator <- R6Class("Accumulator")
Accumulator$set("public", "sum", 0)
Accumulator$set("public", "add", function(x = 1) {
    self$sum <- self$sum + x
    invisible(self)
})
```

Как мы уже говорили ранее, вновь созданные методы и поля будут доступны только для новых объектов. Они не будут добавлены ретроспективно в уже созданные ранее объекты.

14.2.4 Наследование

Для реализации наследования поведения от любого существующего класса передайте объект этого класса в качестве аргумента `inherit` при создании подкласса:

```
AccumulatorChatty <- R6Class("AccumulatorChatty",
    inherit = Accumulator,
    public = list(
        add = function(x = 1) {
            cat("Adding ", x, "\n", sep = "")
            super$add(x = x)
        }
    )
)
```

```
x2 <- AccumulatorChatty$new()
x2$add(10)$add(1)$sum
#> Adding 10
#> Adding 1
#> [1] 11
```

Метод `$add()` переопределяет поведение реализации этого метода в суперклассе, при этом мы по-прежнему можем напрямую вызвать метод родителя с помощью конструкции `super$`. (Это аналогично использованию функции `NextMethod()` в системе S3, о которой мы говорили в разделе 13.6.) Любые методы, не переопределенные в подклассе, будут вызываться из родительского класса.

14.2.5 Интроспекция

У каждого объекта R6 присутствует атрибут S3 `class`, отражающий всю иерархию классов R6. Таким образом, простейший способ определения класса объекта и всей его цепочки наследования состоит в использовании атрибута `class()`:

```
class(hadley2)
#> [1] "Person" "R6"
```

Как видите, в иерархию классов S3 включен базовый класс R6. Он определяет базовое поведение объектов, включая метод `print.R6()`, который вызывает `$print()`, как было показано ранее.

Все методы и поля объекта можно увидеть, воспользовавшись функцией `names()`:

```
names(hadley2)
#> [1] ".__enclos_env__" "age"           "name"
#> [4] "clone"           "print"         "initialize"
```

Мы определили элементы `$name`, `$age`, `$print` и `$initialize`. Как ясно из названия, элемент `.__enclos_env__` представляет собой внутреннюю реализацию, которую вам нет необходимости трогать, а к методу `$clone()` мы вернемся в разделе 14.4.

14.2.6 Упражнения

1. Создайте класс R6, отвечающий за банковский счет, в котором будет храниться информация о балансе, а также будет возможность пополнять баланс и снимать денежные средства. Также создайте подкласс этого класса, который будет выбрасывать ошибку при превышении кредитного лимита. Создайте еще один подкласс, в котором будет допускаться превышение кредитного лимита, но при этом будет взиматься комиссия.

2. Создайте класс R6, имитирующий колоду игральных карт. У вас должна быть возможность взять определенное количество карт из колоды с помощью метода `$draw(n)`, а также вернуть все карты в колоду и перемешать ее посредством метода `$reshuffle()`. Воспользуйтесь приведенным ниже кодом для получения вектора колоды.

```
suit <- c("SPADE", "HEARTS", "DIAMOND", "CLUB")
value <- c("A", 2:10, "J", "Q", "K")
cards <- paste(rep(value, 4), suit)
```

3. Почему нельзя смоделировать создание банковского счета или колоды карт при помощи класса S3?
4. Создайте класс R6, позволяющий получать и устанавливать часовой пояс. Вы можете получить доступ к текущему часовому поясу в R с помощью функции `Sys.timezone()`, а изменить его можно так: `Sys.setenv(TZ = "newtimezone")`. При установке часового пояса убедитесь, что его название присутствует в списке, возвращаемом функцией `OlsonNames()`.
5. Создайте класс R6 для управления текущей рабочей директорией. В нем должны присутствовать методы `$get()` и `$set()`.
6. Почему нельзя смоделировать часовые пояса или управление текущей рабочей директорией при помощи класса S3?
7. На основе какого базового типа строятся объекты R6? Какие у них присутствуют атрибуты?

14.3 Управление доступом

У функции `R6Class()` есть два дополнительных аргумента, ведущих себя подобно аргументу `public`:

- аргумент `private` позволяет вам создавать поля и методы, которые будут доступны только внутри класса, но не за его пределами;
- аргумент `active` позволяет вам использовать *функции доступа* (accessor function) для определения динамических, или активных, полей.

Подробнее об этом мы поговорим в следующих разделах.

14.3.1 Приватность

Система R6 позволяет вам объявлять *приватные* (private) поля и методы, доступ к которым можно получить только внутри класса, но не за его пределами[1]. Вам необходимо знать две вещи, чтобы воспользоваться всеми преимуществами приватных элементов:

[1] Язык R отличается своей гибкостью, так что чисто технически вы можете получить доступ к частным элементам, но для этого вам придется потрудиться и, в частности, изучить детали реализации системы R6.

- аргумент `private` в функции `R6Class` работает так же, как и аргумент `public`: вы передаете ему именованный список методов (функций) и полей;
- к полям и методам, определенным в аргументе `private`, доступ из других методов класса осуществляется при помощи конструкции `private$`, а не `self$`. В то же время из-за пределов класса доступ к приватным элементам запрещен.

Для примера мы можем сделать поля `$age` and `$name` в нашем классе `Person` приватными. С таким определением класса `Person` мы можем установить значения полей `$age` и `$name` только при создании объекта, а получить к ним доступ извне класса нам не удастся.

```
Person <- R6Class("Person",
    public = list(
        initialize = function(name, age = NA) {
            private$name <- name
            private$age <- age
        },
        print = function(...) {
            cat("Person: \n")
            cat(" Name: ", private$name, "\n", sep = "")
            cat(" Age: ", private$age, "\n", sep = "")
        }
    ),
    private = list(
        age = NA,
        name = NULL
    )
)

hadley3 <- Person$new("Hadley")
hadley3
#> Person:
#>   Name: Hadley
#>   Age:  NA
hadley3$name
#> NULL
```

Различия между публичными и приватными полями становятся особенно важны при создании сложных сплетений классов, где необходимо тщательно продумывать систему доступа. Частные поля класса обычно можно безопасно изменять, поскольку извне класса к ним никто обращаться не может. Что касается частных методов, то в R в сравнении с другими языками программирования они играют не такую важную роль, поскольку иерархии объектов здесь редко бывают очень сложными.

14.3.2 Активные поля

С помощью *активных полей* (active field) вы можете объявлять компоненты класса, снаружи выглядящие как поля, но на самом деле реализованные,

подобно методам, в виде функций. Внутренне активные поля реализуются с помощью *активных привязок* (active bindings), о которых мы говорили в разделе 7.2.6. Каждая активная привязка представляет собой функцию, принимающую единственный аргумент: `value`. Если аргумент отсутствует (`missing()`), значение извлекается, в противном случае модифицируется.

К примеру, вы могли бы объявить активное поле `random`, при каждом обращении к нему возвращающее разные значения:

```
Rando <- R6::R6Class("Rando", active = list(
    random = function(value) {
        if (missing(value)) {
            runif(1)
        } else {
            stop("Can't set `$random`", call. = FALSE)
        }
    }
))

x <- Rando$new()
x$random
#> [1] 0.0808
x$random
#> [1] 0.834
x$random
#> [1] 0.601
```

Активные поля бывают особенно полезны в связке с приватными полями, поскольку позволяют реализовывать компоненты, выглядящие со стороны как обычные поля, но при этом поддерживающие дополнительную проверку. К примеру, с помощью них мы можем реализовать поле `age`, которое будет доступно только для чтения, и поле `name`, которое может быть исключительно символьным вектором единичной длины.

```
Person <- R6Class("Person",
    private = list(
        .age = NA,
        .name = NULL
    ),
    active = list(
        age = function(value) {
            if (missing(value)) {
                private$.age
            } else {
                stop("`$age` is read only", call. = FALSE)
            }
        },
        name = function(value) {
            if (missing(value)) {
                private$.name
            } else {
```

```
                stopifnot(is.character(value), length(value) == 1)
                private$.name <- value
                self
            }
        }
    ),
    public = list(
        initialize = function(name, age = NA) {
            private$.name <- name
            private$.age <- age
        }
    )
)

hadley4 <- Person$new("Hadley", age = 38)
hadley4$name
#> [1] "Hadley"
hadley4$name <- 10
#> Error in (function (value) : is.character(value) is not TRUE
hadley4$age <- 20
#> Error: `$age` is read only
```

14.3.3 Упражнения

1. Создайте класс, отвечающий за банковский счет, в котором нельзя будет напрямую устанавливать сумму баланса, а можно будет лишь класть деньги на счет и снимать их. При попытке превысить кредитный лимит должна выбрасываться ошибка.
2. Создайте класс с полем `$password`, доступным только для записи. В классе должен присутствовать метод для проверки пароля `$check_password(password)`, возвращающий `TRUE` или `FALSE`, но возможности для просмотра самого пароля быть не должно.
3. Расширьте класс `Rando` с помощью дополнительной активной привязки, позволяющей получить доступ к предыдущему случайному значению. Убедитесь, что активная привязка является единственным способом осуществления доступа к этому значению.
4. Могут ли подклассы обращаться к приватным полям/методам родительского класса? Найдите ответ на этот вопрос экспериментальным путем.

14.4 Ссылочная семантика

Одним из главных отличий объектов R6 от большинства других объектов является их *ссылочная семантика* (reference semantics). Основное последствие ее – возможность не копировать объекты при их изменении:

```
y1 <- Accumulator$new()
y2 <- y1

y1$add(10)
c(y1 = y1$sum, y2 = y2$sum)
#> y1 y2
#> 10 10
```

В то же время для явного создания копии объекта вам необходимо напрямую обратиться к методу `$clone()`:

```
y1 <- Accumulator$new()
y2 <- y1$clone()

y1$add(10)
c(y1 = y1$sum, y2 = y2$sum)
#> y1 y2
#> 10 0
```

(Метод `$clone()` не выполняет рекурсивное клонирование вложенных объектов R6. Если вам нужно именно это, воспользуйтесь дополнительным аргументом: `$clone(deep = TRUE)`.)

Есть и три других, менее значительных, последствия ссылочной семантики:

- код на языке R, в котором присутствуют объекты R6, труднее понимать из-за необходимости быть погруженным в контекст;
- нужно задумываться об удалении объектов R6, для чего может понадобиться написать вспомогательные методы `$finalize()` в дополнение к `$initialize()`;
- если одно из полей класса является объектом R6, его необходимо создать в методе `$initialize()`, а не в `R6Class()`.

Далее мы подробнее поговорим об этих последствиях.

14.4.1 Осмысление кода

В основном использование ссылочной семантики ведет к усложнению кода с точки зрения его понимания. Рассмотрим простой пример:

```
x <- list(a = 1)
y <- list(b = 2)

z <- f(x, y)
```

Применительно к большинству функций вы можете с уверенностью сказать, что в последней строке кода происходит всего лишь изменение переменной `z`.

Теперь рассмотрим похожий пример, в котором используется вымышленный ссылочный класс `List`:

```
x <- List$new(a = 1)
y <- List$new(b = 2)

z <- f(x, y)
```

Здесь последняя строка кода становится уже не такой очевидной. Если функция `f()` обращается к методам объектов `x` или `y`, они также могут измениться вместе с `z`. Это существенный потенциальный недостаток системы R6, и во избежание его проявления вы должны стараться делать так, чтобы функции либо возвращали значение, либо изменяли поступающие на вход объекты R6, но не выполняли оба действия сразу. В то же время иногда совмещение указанных действий способно приводить к значительному упрощению кода, и мы вернемся к этой теме в разделе 16.3.2.

14.4.2 Финализатор

Одним из полезных свойств ссылочной семантики является возможность задуматься о том, когда объекты R6 финализируются, т. е. удаляются. Для большинства объектов этот вопрос не имеет особого смысла, поскольку концепция копирования при изменении предполагает наличие множества переходных версий объекта, как было упомянуто в разделе 2.6. К примеру, в коде, показанном ниже, создается два объекта фактора: второй создается в момент модификации уровней, оставляя первый на съедение сборщику мусора.

```
x <- factor(c("a", "b", "c"))
levels(x) <- c("c", "b", "a")
```

Поскольку объекты R6 не подвержены копированию при изменении, они удаляются лишь раз, и есть смысл задуматься о том, чтобы вдобавок к инициализатору (`$initialize()`) написать *финализатор* (`$finalize()`). Обычно финализаторы работают аналогично функции `on.exit()`, о которой мы говорили в разделе 6.7.4, и очищают все ресурсы, задействованные во время инициализации объекта. К примеру, в коде, показанном ниже, класс захватывает временный файл при инициализации и автоматически удаляет его во время финализации.

```
TemporaryFile <- R6Class("TemporaryFile", list(
    path = NULL,
    initialize = function() {
        self$path <- tempfile()
    },
    finalize = function() {
        message("Cleaning up ", self$path)
        unlink(self$path)
    }
))
```

Метод финализации будет автоматически запущен в момент удаления объекта (если быть точнее, то это произойдет при первой после отвязки объекта от всех имен сборке мусора) или при завершении R. Это означает, что процедура финализации может быть запущена практически в любом месте вашего кода, что делает нецелесообразным затрагивание в ней общих структур данных. Старайтесь избегать подобных ошибок и используйте финализаторы исключительно для очистки ресурсов, выделенных инициализатором.

```
tf <- TemporaryFile$new()
rm(tf)
#> Cleaning up /tmp/Rtmpk73JdI/file155f31d8424bd
```

14.4.3 Поля R6

В этом разделе мы обсудим еще одно следствие ссылочной семантики, которое может проявиться в самый неожиданный для вас момент. Если вы используете класс R6 в качестве значения поля по умолчанию, он будет общим для всех экземпляров! Рассмотрите приведенный ниже код. Мы хотим, чтобы временная база данных создавалась каждый раз при вызове `TemporaryDatabase$new()`, но в нашем коде все время будет использоваться один и тот же путь.

```
TemporaryDatabase <- R6Class("TemporaryDatabase", list(
    con = NULL,
    file = TemporaryFile$new(),
    initialize = function() {
        self$con <- DBI::dbConnect(RSQLite::SQLite(), path = file$path)
    },
    finalize = function() {
        DBI::dbDisconnect(self$con)
    }
))

db_a <- TemporaryDatabase$new()
db_b <- TemporaryDatabase$new()

db_a$file$path == db_b$file$path
#> [1] TRUE
```

(Если вы знакомы с языком Python, эта проблема очень похожа на проблему с изменяемым аргументом по умолчанию.)

Суть проблемы заключается в том, что метод `TemporaryFile$new()` вызывается лишь раз, при определении класса `TemporaryDatabase`. Для исправления этой проблемы нам необходимо сделать так, чтобы он вызывался всякий раз, когда идет обращение к методу `TemporaryDatabase$new()`, а значит, нам нужно поместить этот метод внутрь `$initialize()`:

```
TemporaryDatabase <- R6Class("TemporaryDatabase", list(
    con = NULL,
    file = NULL,
    initialize = function() {
        self$file <- TemporaryFile$new()
        self$con <- DBI::dbConnect(RSQLite::SQLite(), path = file$path)
    },
    finalize = function() {
        DBI::dbDisconnect(self$con)
    }
))

db_a <- TemporaryDatabase$new()
db_b <- TemporaryDatabase$new()

db_a$file$path == db_b$file$path
#> [1] FALSE
```

14.4.4 Упражнения

1. Создайте класс, который позволит вам записывать строку в указанный файл. Вам необходимо открыть подключение к файлу в методе `$initialize()`, добавить строку с помощью функции `cat()` в методе `$append_line()` и закрыть подключение в методе `$finalize()`.

14.5 Почему R6?

R6 очень похожа на встроенную систему ООП, именуемую *классами на основе ссылок* (reference classes – RC). Лично я предпочитаю R6 по следующим причинам:

- R6 намного проще. Обе системы (R6 и RC) построены на основе окружений, но если R6 использует S3, то RC – S4. Таким образом, чтобы досконально понять RC, вам необходимо разобраться в том, как работает не самая простая система S4;
- R6 обладает расширенной онлайн-документацией: https://r6.r-lib.org;
- в R6 реализован более простой механизм межпакетного наследования – настолько простой, что вы о нем можете даже не задумываться. Чтобы разобраться с этим в RC, вам необходимо ознакомиться с разделом *External Methods; Inter-Package Superclasses* в справке `?setRefClass`;
- RC смешивает переменные и поля в одном стеке окружений, так что вы получаете (`field`) и устанавливаете (`field <<- value`) значения полей так же, как обычные значения. В R6 поля помещаются в отдельное окружение, в результате чего получение (`self$field`) и установка (`self$field <-value`) значений выполняются с префиксом. Подход,

принятый в R6, более многословный, но мне он больше по душе из-за явности намерений;

- R6 намного быстрее RC. Как правило, скорость диспетчеризации методов не так важна, если речь не идет о пакетах замера времени выполнения. В то же время нельзя не отметить, что RC работает довольно медленно, и в том же пакете `shiny` переход с RC на R6 дал существенный рост производительности. За подробностями можно обратиться к виньетке `vignette("Performance", "R6")`;
- RC привязана к R. Это означает, что в случае исправления каких-то ошибок вы сможете воспользоваться всеми преимуществами от этого, только установив новую версию R. Это доставляет трудности при работе с пакетами (такими как из состава tidyverse), которые должны нормально функционировать с разными версиями R;
- наконец, поскольку в основе систем R6 и RC лежат похожие идеи, вам не потребуется много усилий, чтобы изучить особенности работы RC при необходимости.

15

Система S4

15.1 Введение

Система *S4* предлагает более формализованный подход к объектно ориентированному программированию. В ее основе лежат практически те же идеи и принципы, что и в основе системы S3, которую мы обсуждали в главе 13, но сама ее реализация является более строгой – с использованием специализированных функций для создания классов (`setClass()`), обобщенных функций (`setGeneric()`) и методов (`setMethod()`). Помимо этого, система S4 поддерживает *множественное наследование* (multiple inheritance), т. е. класс может иметь сразу несколько родительских классов, и *множественную диспетчеризацию методов* (multiple dispatch).

В системе S4 важную роль играют так называемые *слоты* (slot), представляющие собой именованные компоненты объектов, доступ к которым осуществляется посредством оператора `@`. Наборы слотов и их классов формируют важную часть определения классов S4.

Структура главы

- В разделе 15.2 мы бегло пройдемся по всем компонентам системы S4, включая классы, обобщенные функции и методы.
- Раздел 15.3 будет посвящен более детальному изучению строения классов, и здесь мы коснемся прототипов, конструкторов, помощников и валидаторов.
- В разделе 15.4 мы научимся создавать новые обобщенные функции S4 и снабжать их нужными нам методами. Также мы познакомимся с *функциями доступа* (accessor function), предназначенными для безопасного просмотра и модификации слотов объектов.
- В разделе 15.5 мы погрузимся в тонкости диспетчеризации методов в системе S4. Сначала познакомимся с базовыми идеями, после чего разовьем их до применения множественного наследования и диспетчеризации.
- В разделе 15.6 мы обсудим вопросы взаимодействия систем S4 и S3 и узнаем, как их можно использовать совместно.

Материалы для изучения

Как и в других главах, посвященных объектно ориентированному программированию, здесь мы главным образом сосредоточимся на том, как работает система S4, а не на вопросах ее оптимального применения. Если вы захотите воспользоваться этой системой на практике, то наверняка столкнетесь с двумя сложностями:

- единой документации, отвечающей на все вопросы, касающиеся S4, просто не существует;
- встроенная в R документация зачастую идет вразрез с оптимальными подходами к использованию S4.

В процессе работы с системой S4 вам придется самим собирать по крупицам полезную информацию из имеющейся документации, объединять ее с советами, полученными на StackOverflow, и проверять все на практике. Могу дать пару рекомендаций:

- в сообществе Bioconductor система S4 используется очень давно и продуктивно, результатом чего стало создание большого количества полезных материалов по ее эффективному применению. Вы можете начать со статьи *S4 classes and methods* по адресу https://bioconductor.org/help/course-materials/2017/Zurich/S4-classes-and-methods.html (Мартин Морган (Martin Morgan) и Эрве Пажес (Hervé Pagès)) или с обновленной версии на странице курсов по адресу https://bioconductor.org/help/course-materials. Мартин Морган входит в состав разработчиков R и является руководителем проекта Bioconductor. Его можно с уверенностью назвать экспертом мирового уровня в области практического применения классов S4, и я рекомендую читать все, что он пишет, включая ответы на вопросы на сайте StackOverflow;
- Джон Чемберс (John Chambers) – автор системы S4 – изложил свои идеи и историческую подоплеку в труде *Object-oriented programming, functional programming and R* [Чемберс, 2014]. Для более глубокого погружения в S4 можно ознакомиться с его книгой *Software for Data Analysis* [Чемберс, 2008].

Требования

Все функции, относящиеся к системе S4, находятся в пакете `methods`. Этот пакет всегда можно использовать при запуске R в интерактивном режиме, но он может быть недоступен при работе R в пакетном режиме, т. е. с использованием утилиты *Rscript*[1]. Таким образом, вам лучше всегда использовать

[1] Так сложилось исторически по причине того, что на загрузку пакета `methods` может уходить довольно много времени, тогда как утилита Rscript предполагает быструю работу в режиме командной строки.

команду `library(methods)` при работе с системой S4. Это также сообщит пользователям, читающим ваш код, об использовании объектной системы S4.

```
library(methods)
```

15.2 Основы

Начнем, как и планировали, с быстрого обзора компонентов системы S4. Класс S4 объявляется при помощи функции `setClass()` с указанием имени класса и определением его *слотов* (slot) с именами и классами данных:

```
setClass("Person",
    slots = c(
        name = "character",
        age = "numeric"
    )
)
```

После объявления класса вы можете создавать новые объекты с помощью функции `new()`, на вход которой подаются имя класса и значения его слотов:

```
john <- new("Person", name = "John Smith", age = NA_real_)
```

Определить класс созданного ранее объекта можно с помощью функции `is()`, а доступ к слотам можно получить посредством оператора `@` (эквивалент `$`) и функции `slot()` (эквивалент `[[`):

```
is(john)
#> [1] "Person"
john@name
#> [1] "John Smith"
slot(john, "age")
#> [1] NA
```

В основном в своих методах вам достаточно будет использовать оператор `@`. При работе с чужими классами вы можете найти *функции доступа* (accessor function), которые помогут вам безопасно считывать и устанавливать значения слотов. Будучи разработчиком класса, вам также стоит озаботиться созданием функций доступа для него. Функции доступа обычно представляют собой обобщенные функции S4, позволяющие разным классам реализовывать один внешний интерфейс.

Давайте напишем функции доступа для *установки* (setter) и *чтения* (getter) значения слота `age`, для чего сначала создадим обобщенные функции с помощью функции `setGeneric()`:

```
setGeneric("age", function(x) standardGeneric("age"))
setGeneric("age<-", function(x, value) standardGeneric("age<-"))
```

После этого определим методы, воспользовавшись функцией `setMethod()`:

```
setMethod("age", "Person", function(x) x@age)
setMethod("age<-", "Person", function(x, value) {
    x@age <- value
    x
})

age(john) <- 50
age(john)
#> [1] 50
```

Если вы работаете с классом S4, определенным в пакете, то можете получить информацию о нем с помощью команды `class?Person`. Для получения инструкции по работе с методом поместите вопросительный знак (`?`) перед вызовом (например, `?age(john)`), и для поиска нужной вам страницы с документацией будут использованы классы аргументов.

Наконец, вы можете воспользоваться функциями из пакета `sloop` для определения сторонних объектов и обобщенных функций S4:

```
sloop::otype(john)
#> [1] "S4"
sloop::ftype(age)
#> [1] "S4"      "generic"
```

15.2.1 Упражнения

1. Функция `lubridate::period()` возвращает класс S4. Какие у него есть слоты? Какому классу принадлежит каждый слот? Какие функции доступа предоставляет класс?
2. Какими еще способами вы можете воспользоваться для получения помощи по методу? Примените команду `?"?"` и подведите итоги.

15.3 Классы

Для определения класса S4 вам необходимо воспользоваться функцией `setClass()` с тремя аргументами:

- имя класса (`name`). По общему соглашению имена классов S4 должны иметь слитное написание слов с заглавными буквами («верблюжий» стиль);
- именованный символьный вектор (`slots`), описывающий имена и классы слотов (полей). К примеру, класс, характеризующий человека, может

быть представлен символьным именем (name) и числовым возрастом (age) следующим образом: c(name = "character", age = "numeric"). Псевдокласс ANY позволяет слотам принимать объекты любых типов;

- прототип (prototype) – список значений по умолчанию для каждого слота. Чисто технически этот аргумент является необязательным[1], но вам всегда следует его указывать.

В коде, представленном ниже, мы указали все три аргумента при создании класса Person с символьным именем (name) и числовым возрастом (age):

```
setClass("Person",
    slots = c(
        name = "character",
        age = "numeric"
    ),
    prototype = list(
        name = NA_character_,
        age = NA_real_
    )
)

me <- new("Person", name = "Hadley")
str(me)
#> Formal class 'Person' [package ".GlobalEnv"] with 2 slots
#>   ..@ name: chr "Hadley"
#>   ..@ age : num NA
```

15.3.1 Наследование

Функция setClass() также принимает еще один важный аргумент contains. С помощью него можно указать класс (или классы), от которого(ых) будет наследовать слоты и поведение создаваемый класс. К примеру, мы можем создать класс Employee, который будет наследником класса Person, но при этом будет обладать одним дополнительным слотом, отвечающим за руководителя (boss).

```
setClass("Employee",
    contains = "Person",
    slots = c(
        boss = "Person"
    ),
    prototype = list(
        boss = new("Person")
    )
)
```

[1] В документации ?setClass рекомендуется не использовать аргумент prototype, но, по общему мнению, это является не лучшей практикой.

```
str(new("Employee"))
#> Formal class 'Employee' [package ".GlobalEnv"] with 3 slots
#>   ..@ boss:Formal class 'Person' [package ".GlobalEnv"] with 2 slots
#>   .. .. ..@ name: chr NA
#>   .. .. ..@ age : num NA
#>   ..@ name: chr NA
#>   ..@ age : num NA
```

Функция `setClass()` также может принимать девять других необязательных аргументов, но все они либо устаревшие, либо не рекомендованы к использованию.

15.3.2 Интроспекция

Для определения того, от какого класса наследуется тот или иной объект, воспользуйтесь функцией `is()`:

```
is(new("Person"))
#> [1] "Person"
is(new("Employee"))
#> [1] "Employee" "Person"
```

Для проверки, является ли объект потомком определенного класса, можно использовать второй аргумент этой функции:

```
is(john, "person")
#> [1] FALSE
```

15.3.3 Переопределение

В большинстве языков программирования объявление классов происходит *во время компиляции* (compile-time), а создание объектов – во *время выполнения* (run-time). В R, однако, и объявление классов, и создание объектов происходит во время выполнения. При вызове функции `setClass()` происходит регистрация определения класса в (скрытой) глобальной переменной. Как и в случае с любыми другими функциями, изменяющими состояние, вам необходимо быть крайне осторожными при использовании функции `setClass()`. Повторное объявление класса при наличии созданных ранее объектов может привести к появлению недопустимых объектов:

```
setClass("A", slots = c(x = "numeric"))
a <- new("A", x = 10)

setClass("A", slots = c(a_different_slot = "numeric"))
a
#> An object of class "A"
#> Slot "a_different_slot":
```

```
#> Error in slot(object, what): no slot of name "a_different_slot" for
#> this object of class "A"
```

Это может доставлять неудобства при интерактивном создании новых классов. (В системе R6 тоже присутствует подобная проблема, о чем мы писали в разделе 14.2.2.)

15.3.4 Помощники

Функция `new()` представляет собой низкоуровневый конструктор, удобный для вас как для разработчика. Пользовательские типы всегда должны сопровождаться вспомогательными функциями (функциями-помощниками), с которыми будет удобно работать пользователю. *Функция-помощник* (helper) всегда должна:

- иметь то же имя, что и класс, например `myclass()`;
- обладать тщательно продуманным пользовательским интерфейсом с хорошо подобранными значениями по умолчанию и удобными преобразованиями;
- генерировать сообщения об ошибках, понятные пользователю;
- заканчиваться вызовом функции `methods::new()`.

Наш класс `Person` крайне прост и не нуждается в дополнительных помощниках, но мы можем воспользоваться вспомогательной функцией для явного указания на то, что возраст при создании объекта может не передаваться, тогда как имя является обязательным атрибутом. Мы также реализуем приведение возраста к числу с двойной точностью, чтобы помощник мог работать с целочисленными значениями.

```
Person <- function(name, age = NA) {
    age <- as.double(age)

    new("Person", name = name, age = age)
}

Person("Hadley")
#> An object of class "Person"
#> Slot "name":
#> [1] "Hadley"
#>
#> Slot "age":
#> [1] NA
```

15.3.5 Валидаторы

Конструктор автоматически проверяет корректность классов для слотов при создании объекта:

```
Person(mtcars)
#> Error in validObject(.Object): invalid class "Person" object:
#> invalid object for slot "name" in class "Person": got class
#> "data.frame", should be or extend class "character"
```

При необходимости выполнять дополнительные проверки вам придется реализовать эту логику самостоятельно с помощью *валидатора* (validator). К примеру, нам может понадобиться явно указать, что класс `Person` поддерживает векторизацию и может использоваться для хранения данных сразу о нескольких людях. Сейчас на это ничто не указывает, поскольку слоты `@name` и `@age` могут иметь разную длину:

```
Person("Hadley", age = c(30, 37))
#> An object of class "Person"
#> Slot "name":
#> [1] "Hadley"
#>
#> Slot "age":
#> [1] 30 37
```

Для обеспечения этих дополнительных проверок мы можем написать свой валидатор, воспользовавшись функцией `setValidity()`. На вход она принимает класс и функцию, возвращающую `TRUE` для валидного ввода и символьный вектор с описанием проблемы в случае возникновения ошибок:

```
setValidity("Person", function(object) {
    if (length(object@name) != length(object@age)) {
        "@name and @age must be same length"
    } else {
        TRUE
    }
})
```

Теперь нам не удастся создать недопустимый объект:

```
Person("Hadley", age = c(30, 37))
#> Error in validObject(.Object): invalid class "Person" object: @name
#> and @age must be same length
```

Примечание. Метод валидации вызывается автоматически в функции `new()`, так что вы по-прежнему можете создать недопустимый объект путем его изменения:

```
alex <- Person("Alex", age = 30)
alex@age <- 1:10
```

Чтобы явным образом запустить проверку объекта, можно вызвать функцию `validObject()` вручную:

```
validObject(alex)
#> Error in validObject(alex): invalid class "Person" object: @name and
#> @age must be same length
```

В разделе 15.4.4 мы воспользуемся функцией `validObject()` при написании функций доступа, запрещающих создавать недопустимые объекты.

15.3.6 Упражнения

1. Расширьте класс `Person` при помощи полей таким образом, чтобы он соответствовал классу `utils::person()`. Подумайте, какие слоты вам для этого понадобятся, какие должны быть у них классы и какие проверки вам придется выполнить в методе валидации.
2. Что произойдет, если создать класс S4 вовсе без слотов? Подсказка: прочитайте раздел, посвященный виртуальным классам, в документации `?setClass`.
3. Представьте, что вам необходимо реализовать факторы, даты и датафреймы при помощи классов S4. Как могли бы выглядеть вызовы функции `setClass()` для определения этих классов? Тщательно продумайте содержимое аргументов `slots` и `prototype`.

15.4 Обобщенные функции и методы

Назначение обобщенных функций состоит в выполнении диспетчеризации методов, т. е. в поиске конкретной реализации метода для комбинации классов, переданной в функцию. В данном разделе мы научимся объявлять обобщенные функции и методы S4, а в следующем увидим, как именно происходит процесс диспетчеризации.

Для создания обобщенной функции S4 необходимо вызвать функцию `setGeneric()` с переданной ей функцией, вызывающей `standardGeneric()`, как показано ниже:

```
setGeneric("myGeneric", function(x) standardGeneric("myGeneric"))
```

По общему соглашению, имена классов R6 должны иметь слитное написание слов с заглавными буквами («верблюжий» стиль), при этом первая буква должна быть строчной.

В теле обобщенной функции не стоит использовать фигурные скобки ({}), поскольку это ведет к запуску особого случая и удорожанию процесса выполнения:

```
# Не делайте так!
setGeneric("myGeneric", function(x) {
```

```
    standardGeneric("myGeneric")
})
```

15.4.1 Аргумент signature

Как и `setClass()`, функция `setGeneric()` может принимать множество разных аргументов. Вам стоит знать только об одном из них – `signature`. С его помощью вы можете управлять аргументами, используемыми при диспетчеризации методов. Если этот аргумент не передан, будут использованы все аргументы, кроме ….. Иногда бывает полезно исключить аргументы из процесса диспетчеризации методов. Это позволяет передавать такие аргументы, как `verbose = TRUE` или `quiet = FALSE`, которые не принимают участия в процессе диспетчеризации методов.

```
setGeneric("myGeneric",
    function(x, ..., verbose = TRUE) standardGeneric("myGeneric"),
    signature = "x"
)
```

15.4.2 Методы

Обобщенная функция не имеет никакого смысла без методов, которые в системе S4 мы объявляем с помощью функции `setMethod()`. Эта функция принимает три важных аргумента: имя обобщенной функции, имя класса и, собственно, сам метод.

```
setMethod("myGeneric", "Person", function(x) {
    # реализация метода
})
```

Если быть точными, второй аргумент функции `setMethod()` называется `signature`. В системе S4, в отличие от S3, сигнатура может включать в себя несколько аргументов. Это делает процесс диспетчеризации методов в S4 гораздо более сложным, но позволяет избежать реализации двойной диспетчеризации как частного случая. В следующем разделе мы поговорим о множественной диспетчеризации более подробно. Функция `setMethod()` обладает и другими аргументами, но вам нет необходимости их указывать.

Для просмотра всех методов, принадлежащих обобщенной функции или ассоциированных с классом, воспользуйтесь вызовом функции `methods("generic")` или `methods(class = "class")`. Чтобы найти нужную реализацию конкретного метода, вызовите функцию `selectMethod("generic", "class")`.

15.4.3 Метод show

Наиболее распространенным методом S4, используемым для вывода объекта, является метод `show()`, с помощью которого можно управлять тем, как

именно отображается объект. Чтобы объявить метод для существующей обобщенной функции, сначала необходимо определить аргументы. Их можно посмотреть в документации или воспользоваться функцией `args()` для обобщенной функции:

```
args(getGeneric("show"))
#> function (object)
#> NULL
```

У нашего метода `show` должен быть один аргумент `object`:

```
setMethod("show", "Person", function(object) {
    cat(is(object)[[1]], "\n",
        " Name: ", object@name, "\n",
        " Age: ", object@age, "\n",
        sep = ""
    )
})
john
#> Person
#>   Name: John Smith
#>   Age:  50
```

15.4.4 Функции доступа

Слоты необходимо рассматривать как внутреннюю реализацию объекта: они могут изменяться без предупреждения, и пользовательский код не должен иметь к ним непосредственного доступа. Вместо этого все слоты должны поставляться с парой *функций доступа* (accessor). Если слот используется исключительно в одном классе, это может быть просто функция:

```
person_name <- function(x) x@name
```

Однако обычно вы будете определять обобщенные функции таким образом, чтобы несколько классов могли использовать один интерфейс:

```
setGeneric("name", function(x) standardGeneric("name"))
setMethod("name", "Person", function(x) x@name)

name(john)
#> [1] "John Smith"
```

Если слот доступен для записи, вы должны также предоставить *функцию установки*, или *сеттер* (setter). В этой функции всегда должен присутствовать вызов функции `validObject()` во избежание создания пользователем недопустимых объектов.

```
setGeneric("name<-", function(x, value) standardGeneric("name<-"))
setMethod("name<-", "Person", function(x, value) {
```

```
    x@name <- value
    validObject(x)
    x
})

name(john) <- "Jon Smythe"
name(john)
#> [1] "Jon Smythe"

name(john) <- letters
#> Error in validObject(x): invalid class "Person" object: @name and
#> @age must be same length
```

Если вы незнакомы с нотацией `name<-`, обратитесь к разделу 6.8.

15.4.5 Упражнения

1. Добавьте функции доступа `age()` для класса `Person`.
2. Почему при определении обобщенной функции необходимо дважды повторять ее имя?
3. Почему в методе `show()`, показанном в разделе 15.4.3, используется запись `is(object)[[1]]`? Подсказка: попробуйте вывести на экран подкласс сотрудника.
4. Что произойдет, если объявить метод с именами аргументов, отличающимися от обобщенной функции?

15.5 Диспетчеризация методов

Процесс диспетчеризации методов в системе S4 осложнен по причине двух факторов:

- множественное наследование, подразумевающее наличие у класса сразу нескольких родителей;
- множественная диспетчеризация, при которой обобщенная функция может использовать несколько аргументов в процессе поиска нужного метода.

Присутствие этих аспектов делает систему S4 очень мощной, но вместе с тем затрудняет понимание того, какой именно метод будет выбран для заданной комбинации входных параметров. На практике я рекомендую реализовывать максимально простую, насколько это возможно, диспетчеризацию методов, избегая использования множественного наследования, а к множественной диспетчеризации прибегать только в исключительных случаях, когда без нее просто не обойтись.

В то же время мы не можем обойти вниманием эти сложные концепции. Начнем, как и всегда, с простых сценариев с простым наследованием и обыч-

ной диспетчеризацией, а затем перейдем к более сложным случаям. Для иллюстрации своих идей будем использовать воображаемый *граф классов* (class graph) на основе эмодзи, показанный на рис. 15.1.

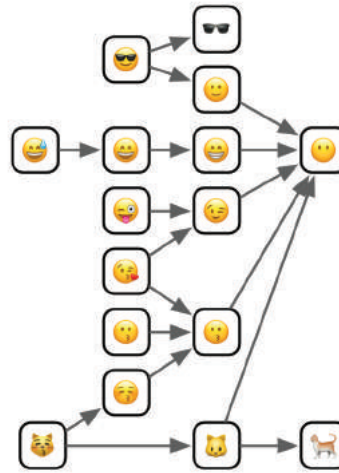


Рис. 15.1 Граф классов

С помощью эмодзи можно очень лаконично выражать имена классов и понимать связи между ними. Взгляните на рис. 15.1. Здесь хорошо видно, что класс 😜 наследуется от класса 😉, который в свою очередь происходит от класса 😶, тогда как класс 😎 наследуется сразу от двух классов: 🕶 и 🙂.

15.5.1 Простая диспетчеризация

Давайте начнем с простейшего случая, а именно с обобщенной функции и диспетчеризации с использованием одного класса с одним родителем. Это простая диспетчеризация, на примере которой можно продемонстрировать графические соглашения (рис. 15.2), которые мы будем использовать и для более сложных случаев.

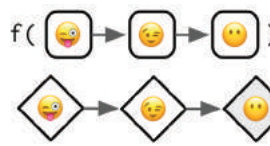


Рис. 15.2 Простая диспетчеризация

Показанная диаграмма состоит из двух частей:

- в верхней части (`f(...)`) определяется область видимости диаграммы. Здесь мы имеем дело с обобщенной функцией с одним аргументом, выраженным с помощью иерархии классов, состоящей из трех уровней;

- в нижней части показан *граф методов* (method graph), на котором отображены все возможные для определения методы. Существующие методы, т. е. те, которые были объявлены с помощью функции `setMethod()`, обладают серым фоном.

Для поиска вызванного метода мы начинаем с наиболее специфичного класса указанных аргументов, после чего двигаемся по стрелкам, пока не найдем существующий метод. К примеру, если функция была вызвана с объектом класса 😉, мы должны будем проследовать по стрелке направо, где и обнаружим метод, определенный для более общего класса 😶. Если ни один метод не будет найден, процесс диспетчеризации прервется с ошибкой. На практике это означает, что вам необходимо всегда объявлять методы для концевых узлов, показанных на графе справа.

Существует два *псевдокласса* (pseudo-class), для которых вы можете определять методы. Псевдоклассами они называются по причине того, что сами они не существуют, но при этом позволяют определить полезное поведение. Первым псевдоклассом является `ANY`, соответствующий любому классу[1]. По техническим соображениям, о которых я поведаю позже, связь с `ANY` является более длинной, чем с другими классами, что показано на рис. 15.3.

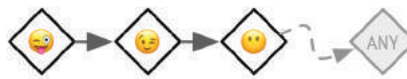


Рис. 15.3 Связь с псевдоклассом `ANY`

Второй псевдокласс, который мы упомянули, – это `MISSING`. При определении метода для этого псевдокласса он будет соответствовать случаю с пропущенным аргументом. Это не несет никакого смысла в случае с простой диспетчеризацией, но очень полезно для функций вроде `+` и `-`, в которых используется двойная диспетчеризация и которые ведут себя по-разному в зависимости от того, один аргумент передан или два.

15.5.2 Множественное наследование

Ситуация осложняется, когда у класса может быть больше одного родителя, как показано на рис. 15.4.

В основном процедура остается прежней: начинаем мы с класса, переданного обобщенной функции, после чего следуем по стрелкам до нахождения существующего метода. Загвоздка в том, что теперь от одного класса может отходить сразу несколько стрелок, так что вы можете найти больше одного метода. В этом случае вы должны остановиться на ближайшем методе, т. е. на том, до которого путь был кратчайшим.

---

[1] Псевдокласс `ANY` в S4 играет такую же роль, как псевдокласс по умолчанию в S3.

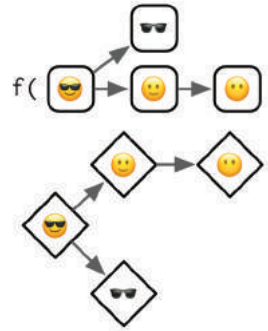


Рис. 15.4 Множественное наследование

Примечание. Несмотря на внешнюю привлекательность метафоры с графами методов, представленной в этом разделе, на практике такой вид реализации является не самым эффективным, и в реальности в системе S4 используется другой подход. С подробностями вы можете ознакомиться в справке `?Methods_Details`.

Что происходит, если два метода находятся на одинаковом удалении от исходного класса? Представьте, что мы определили методы для классов 🕶 и 🙂, после чего вызвали обобщенную функцию с классом 😎. Обратите внимание, что для класса 😶 не может быть найден ни один метод, что показано при помощи двойной обводки на рис. 15.5.

Такая ситуация называется *противоречивой*, или неопределенной (ambiguous), и на диаграммах я буду показывать ее с помощью пунктирной обводки класса, как на рис. 15.4. При возникновении неопределенности в R вы получите предупреждение, а класс с методом будет выбран в алфавитном порядке (этот подход сопряжен со случайностью, и в работе полагаться на него не стоит). При обнаружении неопределенности в поиске методов необходимо разрешить конфликт, как показано на рис. 15.6, предоставив классу нужный метод.

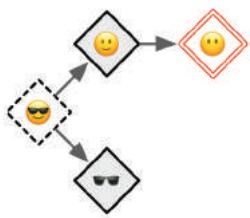


Рис. 15.5 Множественное наследование и поиск методов

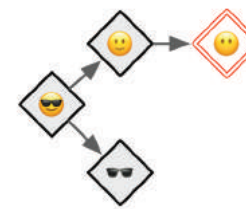


Рис. 15.6 Разрешение конфликтов при поиске методов

У нас также есть в запасе резервный метод ANY, но правила для его использования чуть сложнее. Как видно на рис. 15.7, пути до метода ANY, показанные пунктирными извилистыми линиями, всегда будут считаться более длинными по сравнению с маршрутом до реального класса. Это означает, что в данном случае не будет возникать неопределенности при поиске методов.

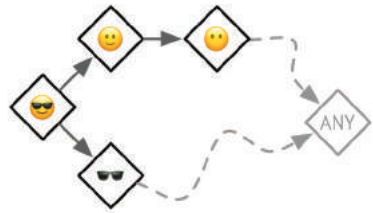


Рис. 15.7 Долгий путь до метода ANY

При использовании множественного наследования бывает очень трудно одновременно избегать неопределенности при поиске методов, следить за реализацией методов на концевых узлах и минимизировать количество объявленных методов (чтобы воспользоваться всеми преимуществами ООП). К примеру, из шести способов определения двух методов, показанных на рис. 15.8, только один лишен проблем. Именно поэтому я рекомендую относиться к множественному наследованию с большой осторожностью: при его использовании вам необходимо тщательно продумывать граф методов и иметь четкий план.

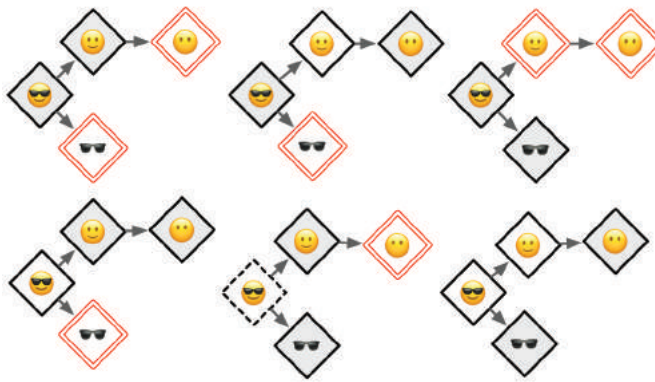


Рис. 15.8 Шесть способов определения двух методов с множественным наследованием

15.5.3 Множественная диспетчеризация

Поняв, как работает множественное наследование, вы без труда разберетесь и с множественной диспетчеризацией. Вы просто следуете по стрелкам слева направо, как и раньше, но теперь каждый метод определяется двумя классами (разделенными запятыми), как показано на рис. 15.9.

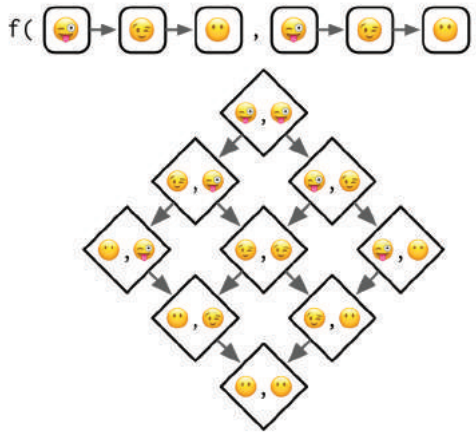


Рис. 15.9 Множественная диспетчеризация

Я не буду демонстрировать примеры диспетчеризации с использованием более двух аргументов – при желании вы сможете без труда построить собственные графы методов, используя базовые принципы.

Главным отличием между множественным наследованием и множественной диспетчеризацией является необходимость следования гораздо большему количеству стрелок. На рис. 15.10 показан пример с четырьмя объявленными методами и двумя неопределенными ситуациями.

При работе с множественной диспетчеризацией обычно бывает меньше сложностей, чем с множественным наследованием, что связано с меньшим количеством комбинаций концевых классов. В показанном выше примере такая комбинация всего одна. Это означает, что вам достаточно объявить один метод, и для всех входных параметров будет определено поведение по умолчанию.

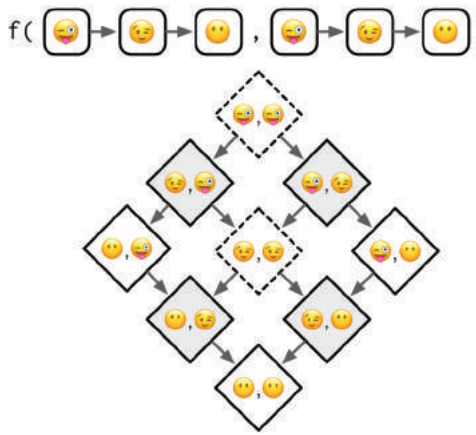


Рис. 15.10 Конфликты при множественной диспетчеризации

15.5.4 Множественная диспетчеризация и множественное наследование

Разумеется, никто не может запретить вам использовать множественную диспетчеризацию совместно с множественным наследованием, что показано на рис. 15.11.

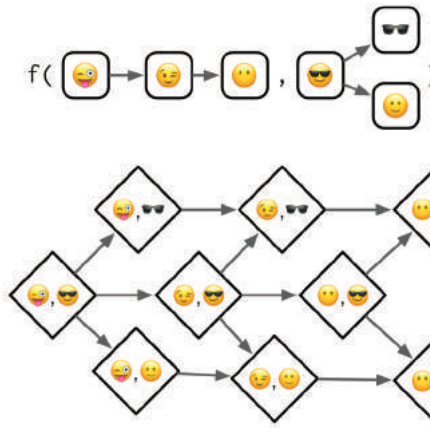


Рис. 15.11 Совместное использование множественной диспетчеризации и множественного наследования

В еще более сложном случае, показанном на рис. 15.12, диспетчеризация производится по двум классам, в каждом из которых присутствует множественное наследование.

С увеличением сложности графа методов становится все труднее определить, какой именно метод будет выбран при разных комбинациях входных аргументов, а также растет риск возникновения противоречий. Если вам приходится рисовать диаграммы, чтобы определить, какой метод будет вызван в том или ином случае, пора задуматься об упрощении структуры кода.

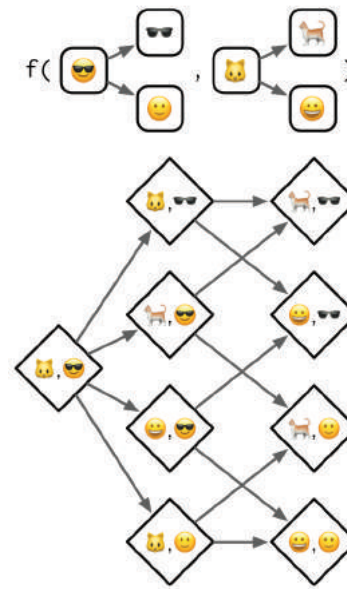


Рис. 15.12 Сложный случай совместного использования множественной диспетчеризации и множественного наследования

15.5.5 Упражнения

1. Нарисуйте граф методов для `f(😁, 🐮)`.
2. Нарисуйте граф методов для `f(😀, 😉, 😚)`.
3. Рассмотрите последний пример с диспетчеризацией по двум классам, в каждом из которых есть множественное

наследование. Что произойдет при объявлении методов для концевых классов? Почему диспетчеризация методов не помогает нам сократить объем работы?

15.6 S4 и S3

При написании кода с использованием системы S4 вам зачастую приходится взаимодействовать с классами и обобщенными функциями S3. В этом разделе мы посмотрим, как классы, методы и обобщенные функции S4 ведут себя в присутствии другого кода.

15.6.1 Классы

В аргументах `slots` и `contains` вы можете использовать классы S4, классы S3, а также неявные классы (раздел 13.7.1) базового типа. Для использования класса S3 вам сначала придется его зарегистрировать при помощи функции `setOldClass()`. Эту функцию необходимо вызвать один раз для каждого класса S3, передав ей на вход атрибут класса. Приведенные ниже объявления уже включены в базовый R:

```
setOldClass("data.frame")
setOldClass(c("ordered", "factor"))
setOldClass(c("glm", "lm"))
```

В то же время будет лучше, если вы снабдите классы полным определением S4 с атрибутами `slots` и `prototype`:

```
setClass("factor",
    contains = "integer",
    slots = c(
        levels = "character"
    ),
    prototype = structure(
        integer(),
        levels = character()
    )
)
setOldClass("factor", S4Class = "factor")
```

Обычно подобные определения поставляет сам разработчик классов S3. Если вы хотите построить класс S4 на базе класса S3, представленного в пакете, вам необходимо попросить разработчика пакета внести этот вызов в пакет, вместо того чтобы добавлять его в свой код.

При наследовании класса S4 от класса S3 или от базового типа он будет содержать особый виртуальный слот по имени `.Data`, в котором будет находиться этот базовый тип или объект S3:

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

Вы можете определять методы S3 для обобщенных функций S4 и наоборот (при условии вызова функции `setOldClass()`). Но такие сценарии являются более сложными, чем кажется на первый взгляд, и перед их реализацией я советую вам внимательно прочитать документацию по команде `?Methods_for_S3`.

15.6.2 Обобщенные функции

Помимо создания обобщенной функции с нуля, вы также можете преобразовать существующую обобщенную функцию S3 в формат S4:

```
setGeneric("mean")
```

В этом случае существующая функция станет методом по умолчанию (`ANY`):

```
selectMethod("mean", "ANY")
#> Method Definition (Class "derivedDefaultMethod"):
#>
#> function (x, ...)
#> UseMethod("mean")
#> <bytecode: 0x7fd256468a30>
#> <environment: namespace:base>
#>
#> Signatures:
#>          x
#> target  "ANY"
#> defined "ANY"
```

Примечание. Функция `setMethod()` автоматически вызовет функцию `setGeneric()`, если ее первый аргумент не является обобщенной функцией, что позволяет вам превратить любую существующую функцию в обобщенную функцию S4. В преобразовании существующих обобщенных функций S3 в формат S4 нет ничего предосудительного, однако вам стоит избегать трансформации обычных функций в обобщенные функции S4 в пакетах, поскольку это требует тщательной координации действий при работе с несколькими пакетами.

15.6.3 Упражнения

1. Как бы выглядело полное определение функции `setOldClass()` для упорядоченного фактора (добавьте атрибуты `slots` и `prototype` к определению выше)?
2. Определите метод `length` для класса `Person`.

16

Компромиссы

16.1 Введение

Итак, вы узнали о трех наиболее мощных инструментах объектно ориентированного программирования, присутствующих в R. И теперь, когда вы понимаете идеи и принципы, лежащие в их основе, можно попробовать сравнить их и определить сильные и слабые стороны каждого инструмента. Это поможет вам выбрать наиболее подходящую систему под конкретную задачу.

В целом при выборе системы ООП я рекомендую начинать с S3 в качестве первого кандидата. Эта система широко используется в базовом R и в репозитории CRAN. И хотя она далека от идеала, освоить ее будет несложно, а для преодоления всех ее недостатков давно выработаны полезные подходы. Если у вас есть опыт программирования на других языках, вам, вероятно, больше придется по душе система R6, поскольку она покажется вам наиболее знакомой. Мне кажется, есть как минимум две причины не идти по этому пути. Во-первых, при использовании R6 очень легко создать несвойственный для языка R API, который опытным пользователям покажется довольно странным. К тому же он может обладать рядом недостатков, связанных со ссылочной семантикой. Во-вторых, категорично выбрав систему R6, вы упустите возможность изучить новый подход к объектно ориентированному программированию, который могли бы использовать для решения насущных проблем.

Структура главы

- В разделе 16.2 мы проведем сравнение систем S3 и S4. Если говорить коротко, система S4 является гораздо более строгой и требует более серьезного предварительного планирования. Это делает ее наиболее пригодной для больших проектов, в разработке которых принимает участие целая команда, а не один человек.
- В разделе 16.3 мы сравним системы S3 и R6. Этот раздел будет достаточно длинным, поскольку эти две системы существенно отличаются друг от друга, и при выборе между ними необходимо учитывать немало аспектов.

Требования

Для чтения этой главы вы должны быть знакомы с системами S3, S4 и R6 на уровне, предложенном в трех предыдущих главах.

16.2 S4 против S3

Освоив систему ООП S3, вам будет достаточно просто при необходимости переключиться на S4. В основе S4 лежат те же идеи, при этом сама эта система отличается большей строгостью и многословностью. Эти качества делают систему S4 более пригодной для масштабных разработок в команде. Благодаря строгой структуре системы вам не потребуется вырабатывать специальные соглашения, а новым разработчикам понадобится не так много времени, чтобы освоиться в проекте. В то же время система S4 требует гораздо более тщательного предварительного планирования по сравнению с S3, и это также делает ее более пригодной для применения в крупных проектах с большим количеством доступных ресурсов.

Одним из примеров успешного использования S4 в масштабной разработке является проект Bioconductor. Он представляет собой некое подобие репозитория CRAN в том смысле, что помогает распространять множество полезных пакетов. В то же время Bioconductor обладает гораздо меньшим масштабом по сравнению с CRAN (на начало 2024 года он насчитывает чуть больше 2200 пакетов, тогда как в CRAN количество пакетов перевалило за 20 000. – *Прим. перев.*), притом что пакеты в нем более тесно интегрированы друг с другом по причине общности предметной области и более строгого процесса рецензирования. Пакеты, входящие в репозиторий Bioconductor, не обязаны использовать систему S4, но в большинстве из них применяется именно она – это обусловлено тем, что ключевые структуры данных, такие как `SummarizedExperiment`, `IRanges` и `DNAStringSet`, построены на основе S4.

Система S4 также отлично подходит для воплощения сложных систем со множеством взаимосвязанных объектов в связи с возможностью минимизировать дублирование кода путем реализации подходящих методов. Лучшим примером такой системы является пакет `Matrix` [Бейтс (Bates) и Мехлер (Maechler), 2018]. Он предназначен для эффективного хранения и вычисления различных типов разреженных и плотных матриц. В версии 1.2.15 пакет насчитывал 102 класса, 21 обобщенную функцию и 1993 метода. Для понимания всей сложности этого пакета на рис. 16.1 приведен небольшой фрагмент его графа классов.

Данная предметная область идеально подходит для использования системы S4 из-за наличия шаблонных вычислений для разных комбинаций разреженных матриц. С помощью S4 можно легко реализовать общие методы, которые будут использоваться для всех входных данных, после чего добавить более узконаправленные методы, оптимизированные под конкретные типы входов. Такой подход требует тщательного планирования во избежание

возникновения неопределенности в процессе диспетчеризации методов, но взамен вы можете получить высокую производительность системы.

Самой большой преградой в использовании системы S4 является ее сложность в сочетании с отсутствием единой документации. S4 очень непросто применять на максимуме заложенных в нее возможностей. Это не было бы такой большой проблемой, если бы документация по S4 не была столь хаотично разбросана по книгам, сайтам и инструкциям по R. Эта система достойна отдельной книги, которой, увы, пока не существует. (Кстати, с документацией по S3 такая же история, но в данном случае это не так критично, поскольку разобраться с этой системой не так сложно.)

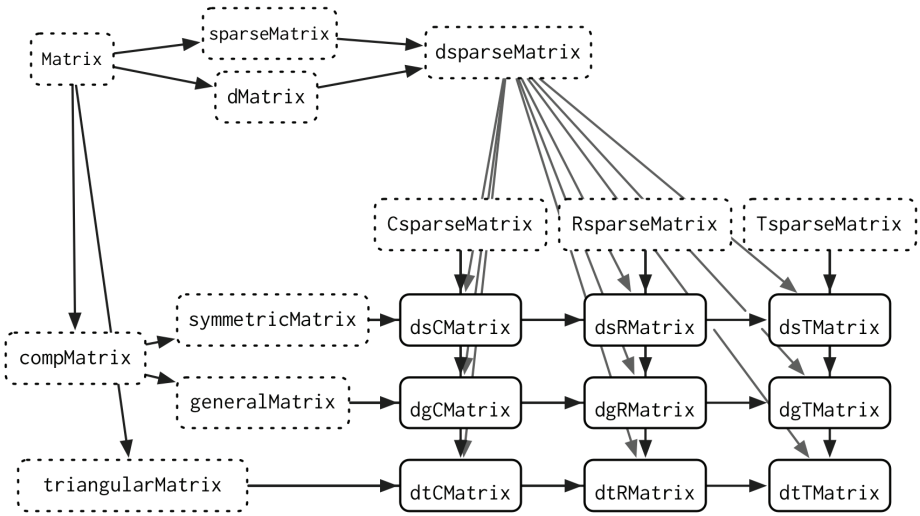


Рис. 16.1 Небольшой фрагмент графа классов пакета `Matrix`, демонстрирующий иерархию наследования разреженных матриц. Каждый класс здесь наследуется от двух виртуальных классов: один из них отвечает за хранение (C = колоночное хранение, R = строчное хранение, T = хранение на основе меток), а второй описывает ограничения матрицы (s = симметричная, t = треугольная, g = обобщенная)

16.3 R6 против S3

Система ООП R6 кардинально отличается от S3 и S4 тем, что она построена на основе инкапсуляции объектов, а не на обобщенных функциях. Кроме того, объекты R6 обладают ссылочной семантикой, что позволяет изменять их на месте. Эти два отличия тянут за собой большое количество не самых очевидных последствий, которых мы коснемся в этом разделе:

- обобщенная функция – это обычная функция, которая располагается в глобальном пространстве имен. Методы R6, напротив, принадлежат

конкретному объекту, а значит, находятся в локальном пространстве имен. Это обусловливает специфику подхода к именованию;

- ссылочная семантика, характерная для R6, позволяет методам одновременно возвращать значение и модифицировать объект, что решает проблему, известную как *режим протягивания* (threading state);
- методы R6 вызываются с помощью инфиксного оператора `$`, что при правильной настройке методов позволяет выстраивать их в цепочки в качестве альтернативы использованию конвейеров.

Таковы основные компромиссы, с которыми мы сталкиваемся при сравнении функционального и инкапсулированного подходов к ООП, и они также лежат в основе дискуссии относительно проектирования систем в R и Python.

16.3.1 Пространства имен

Одно из не самых очевидных отличий между системами S3 и R6 связано с пространствами имен, в которых располагаются методы:

- обобщенные функции являются глобальными: все пакеты разделяют одно и то же пространство имен;
- инкапсулированные методы локальны, и каждый из них привязан к конкретному объекту.

Преимуществом использования глобального пространства имен является возможность для разных пакетов использовать одни и те же методы при работе с разными типами объектов. Обобщенные функции предлагают единый API, облегчающий выполнение типичных операций применительно к новым объектам из-за наличия строгих соглашений об именовании. Это бывает очень удобно при анализе данных, поскольку вам зачастую требуется выполнять схожие действия с объектами разных типов. В том числе поэтому система создания моделей в R отличается своим удобством: вне зависимости от того, где была реализована модель, вы всегда можете работать с ней с использованием одних и тех же инструментов (`summary()`, `predict()`, ...).

Недостатком использования глобального пространства имен можно назвать необходимость более тщательного продумывания системы именования. Всегда необходимо стремиться к отсутствию конфликтов имен между обобщенными функциями и функциями в других пакетах, поскольку в этом случае пользователю придется часто использовать оператор `::` для указания принадлежности пакету. Но это не так просто, ведь обычно в качестве имен функций применяются английские глаголы, которые в большинстве случаев обладают несколькими значениями. Возьмем, к примеру, функцию `plot()`:

```
plot(data) # вывести на график (plot) какие-то данные
plot(bank_heist) # спланировать (plot) преступление
plot(land) # выделить (plot) новый земельный участок
plot(movie) # сочинить (plot) сценарий фильма
```

Обычно следует избегать использования методов, являющихся омонимами исходной обобщенной функции, – вместо этого лучше определить новую обобщенную функцию.

При работе с методами R6 такая проблема отсутствует из-за их привязки к конкретному объекту. С кодом, приведенным ниже, все в порядке, ведь в данном случае вполне очевидно использование метода `plot()` в контексте каждого отдельного объекта R6:

```
data$plot()
bank_heist$plot()
land$plot()
movie$plot()
```

Подобные размышления применимы и к аргументам обобщенных функций. Обобщенные функции S3 должны принимать одинаковые ключевые аргументы, что делает их именование менее осмысленным и более общим (`x`, `.data` и т. д.). Обычно обобщенным функциям S3 требуется аргумент `...` для передачи дополнительных аргументов методам, а это зачастую становится источником проблем, связанных с тем, что наличие опечаток при именовании аргументов не приводит к возникновению ошибок. В то же время методы R6 могут отличаться большим разнообразием и принимать аргументы с более осмысленными именами.

Еще одним преимуществом использования локального пространства имен является дешевизна создания методов R6. В большинстве инкапсулированных систем ООП используются небольшие по объему методы, обладающие осмысленными именами и выполняющие конкретные действия. Создание метода в системе S3 сопряжено с большими расходами, поскольку попутно вам может понадобиться написать соответствующую обобщенную функцию и тщательно продумать все имена. Это делает совет создавать множество мелких методов неприменимым к S3. Да, вы по-прежнему можете разбивать свой код на отдельные части для его лучшего структурирования и понимания, но в основном это будет касаться обычных функций, а не методов как таковых.

16.3.2 Режим протягивания

Одним из недостатков системы S3 является невозможность одновременного изменения объекта в функции и возвращения значения. Это противоречит директиве о том, что функция должна вызываться либо ради возврата значения, либо для осуществления побочного эффекта, ведь зачастую нам необходимо сочетание этих двух действий.

Представьте, что вам нужно создать стек объектов, в котором должны присутствовать два метода:

- метод `push()` добавляет новый объект в верхнюю часть стека;
- метод `pop()` возвращает верхний элемент и удаляет его из стека.

Реализация конструктора и метода `push()` не должна вызывать вопросов. Стек содержит список объектов, и добавление объекта в стек представляет собой его добавление в список.

```
new_stack <- function(items = list()) {
    structure(list(items = items), class = "stack")
}

push <- function(x, y) {
    x$items <- c(x$items, list(y))
    x
}
```

(Я не стал создавать реальный метод `push()`, поскольку его обобщение привело бы к излишнему усложнению нашего примера.)

Что касается метода `pop()`, то с его реализацией могут возникнуть сложности, поскольку он должен и возвращать значение (объект с вершины стека), и выполнять побочный эффект в виде удаления этого объекта из стека. Поскольку в S3 мы не можем модифицировать входной объект, нам необходимо вернуть две вещи: само значение и обновленный объект.

```
pop <- function(x) {
    n <- length(x$items)
    item <- x$items[[n]]
    x$items <- x$items[-n]

    list(item = item, x = x)
}
```

В результате это приведет к довольно неуклюжему коду:

```
s <- new_stack()
s <- push(s, 10)
s <- push(s, 20)

out <- pop(s)
out$item
#> [1] 20
s <- out$x
s
#> $items
#> $items[[1]]
#> [1] 10
#>
#>
#> attr(,"class")
#> [1] "stack"
```

Эта проблема получила название *режим протягивания* (threading state), или *аккумуляторное программирование* (accumulator programming), поскольк-

ку вне зависимости от того, на какой глубине был вызван метод `pop()`, вам придется протягивать модифицированный объект стека обратно.

Одним из способов решения данной проблемы в других функциональных языках программирования является операция *множественного присваивания* (multiple assign), позволяющая присваивать несколько значений за раз. В пакете `zeallot` [Титор (Teetor), 2018] реализовано множественное присваивание в R с помощью оператора `%<-%`. Его использование делает код чуть более элегантным, но не решает проблему как таковую:

```
library(zeallot)

c(value, s) %<-% pop(s)
value
#> [1] 10
```

В системе R6 реализация стека будет выглядеть проще из-за возможности изменить объект прямо в методе `$pop()`, а обратно вернуть только верхний элемент:

```
Stack <- R6::R6Class("Stack", list(
    items = list(),
    push = function(x) {
        self$items <- c(self$items, x)
        invisible(self)
    },
    pop = function() {
        item <- self$items[[self$length()]]
        self$items <- self$items[-self$length()]
        item
    },
    length = function() {
        length(self$items)
    }
))
```

В результате код примет более естественный вид:

```
s <- Stack$new()
s$push(10)
s$push(20)
s$pop()
#> [1] 20
```

В реальной жизни я столкнулся с этой проблемой при работе со шкалами в пакете `ggplot2`. Сложность работы со шкалами заключается в необходимости объединять в них данные со всех графиков и слоев. Изначально я использовал классы S3, в результате чего мне потребовалось передавать данные шкал во множество функций и получать их обратно. Переход на систему R6 позволил значительно упростить код. Но вместе с тем я нажил себе

новые проблемы, забыв про вызов метода `$clone()` при модифицировании графиков. В результате не зависящие друг от друга графики получили возможность использовать данные шкал совместно, что породило трудную для отслеживания ошибку.

16.3.3 Сцепление методов

Мы уже говорили про очень полезный оператор создания конвейеров `%>%`, позволяющий объединить вызовы функций в цепочку, которая будет выполняться слева направо. Но для R6 этот оператор не так важен, поскольку в этой системе используется собственный инфиксный оператор `$`. В результате мы можем связать воедино несколько методов в одном выражении. Этот способ называется *сцеплением методов* (method chaining):

```
s <- Stack$new()
s$
    push(10)$
    push(20)$
    pop()
#> [1] 20
```

Обычно подобные техники используются в других языках программирования, таких как Python и JavaScript, а в R это возможно с одной оговоркой: любой метод R6, вызываемый ради выполнения побочного эффекта (обычно для модификации объекта), должен возвращать `invisible(self)`.

Главным преимуществом сцепления методов является использование автодополнения, а главным недостатком – то, что только разработчик класса может добавлять новые методы (и отсутствие возможности использования множественной диспетчеризации).

Часть IV

Метапрограммирование

Введение

Одна из интригующих особенностей языка R состоит в возможности использовать *метапрограммирование* (metaprogramming). Идея этой концепции состоит в восприятии кода в качестве данных, которые могут быть исследованы и модифицированы программным путем. Это очень мощная техника, оказавшая влияние на код, использующийся в R. На самом базовом уровне именно идеи, лежащие в основе метапрограммирования, позволяют вам писать `library(purrr)` (без кавычек) вместо `library("purrr")` и при выполнении команды `plot(x, sin(x))` автоматически помечать оси как `x` и `sin(x)`. Если идти глубже, можно вспомнить, что принципы метапрограммирования позволяют записывать модель, предсказывающую значение переменной `y` на основе предикторов `x1` и `x2`, в виде формулы `y ~ x1 + x2`, транслировать выражение `subset(df, x == y)` в `df[df$x == df$y, , drop = FALSE]` и использовать запись `dplyr::filter(db, is.na(x))` для генерирования SQL-выражения `WHERE x IS NULL`, когда `db` представляет таблицу в удаленной базе данных.

Тесно связанным с метапрограммированием понятием является *нестандартное вычисление* (non-standard evaluation – NSE). С этим термином, который часто используется для описания поведения функций в R, есть две проблемы. Во-первых, NSE на самом деле является свойством аргумента (или аргументов) функции, так что говорить об NSE применительно к функциям как минимум некорректно. Во-вторых, довольно странно давать чему-то определение, исходя из того, чем (стандартом) это что-то не является, так что в этой книге я буду использовать более точные термины.

В основном в этой книге мы ориентируемся на концепцию *tidy evaluation* (коротко `tidy eval`). Данные принципы реализованы в пакете `rlang` [Хенри и Уикем, 2018b], к которому мы будем часто обращаться в следующих главах. Это позволит вам увидеть описываемую концепцию в целом, не отвлекаясь на детали реализации, многие из которых восходят к истории развития языка R. После описания общих идей с использованием пакета `rlang` я расскажу о том, как они реализованы в базовом R. Кому-то такой подход может показаться неэффективным, но я бы провел параллель с обучением вождению с использованием автоматической коробки передач вместо ручной: это позволяет сосредоточиться на общей картине, перед тем как вдаваться в детали. В этой книге мы остановимся на теоретических аспектах *tidy evaluation*, чтобы вы могли досконально понять, как это все работает. Если вам необходимо практическое введение, вы можете обратиться к сайту https://tidyeval.tidyverse.org.

В следующих пяти главах вы узнаете о метапрограммировании и *tidy evaluation*.

1. В главе 17 – *Общая картина* – описывается концепция метапрограммирования в широком смысле и дается краткое введение в ее основные компоненты и их совместную работу.

2. В главе 18 – *Выражения* – вы узнаете о том, что весь код на языке R может быть представлен в виде дерева. Кроме того, вы научитесь визуализировать эти деревья, увидите, как принятые в R правила позволяют преобразовывать линейные последовательности символов в деревья, и узнаете об использовании рекурсивных функций для работы с этими деревьями кода.
3. В главе 19 будут представлены инструменты из пакета `rlang`, предназначенные для захвата (цитирования) невычисляемых аргументов функции. Здесь вы познакомитесь с квазицитированием, представляющим собой набор техник для расцитирования входов с целью простого генерирования новых деревьев на основе фрагментов кода.
4. В главе 20 мы перейдем к вычислению захваченного кода. Здесь вы познакомитесь с важной структурой данных *quosure*, помогающей корректно вычислять выражение путем захвата кода для вычисления и окружения, в котором это вычисление необходимо произвести. В этой главе мы соберем все воедино для понимания того, как NSE работает в базовом R, и научимся писать функции, подобные `subset()`.
5. В главе 21 – *Транслирование* – мы объединим окружения первого класса, лексический поиск в области видимости и метапрограммирование с целью преобразования кода на R в другие языки, такие как HTML и LaTeX.

17

Общая картина

17.1 Введение

Метапрограммирование – это наиболее сложная тема в данной книге, поскольку она объединяет в себе множество тем, которые до этого момента никак не были связаны, и заставляет решать задачи, о существовании которых вы могли даже не знать. Кроме того, в следующих главах вам предстоит освоить новую для себя лексику, и поначалу вам будет казаться, что каждый новый термин строится на основе трех других, о которых вы и слыхом не слыхивали. И даже если вы можете похвастаться недюжинным опытом в других языках программирования, весь этот опыт может оказаться бесполезным при освоении метапрограммирования, поскольку в большинстве современных языков эта концепция не доведена до уровня, сопоставимого с R. Так что не пугайтесь, если сперва вам все покажется очень странным и непонятным. Через это проходили все!

В то же время мне кажется, что сейчас вам будет изучить метапрограммирование гораздо легче, чем это было раньше. За последние несколько лет в теории и практике был достигнут небывалый прогресс в отношении базовых знаний в совокупности с инструментами, позволяющими решать самые разные задачи. В этой главе мы рассмотрим все основные компоненты метапрограммирования и узнаем, как они сочетаются в работе.

Структура главы

В каждом разделе этой главы мы будем знакомиться с очередной важной идеей концепции метапрограммирования:

- в разделе 17.2 мы научимся представлять код в виде данных и создавать и модифицировать выражения путем захватывания кода;
- в разделе 17.3 будет представлена древовидная структура кода, называемая абстрактным синтаксическим деревом;
- в разделе 17.4 будет затронута тема создания выражений программным путем;
- раздел 17.5 будет посвящен выполнению выражений путем их вычисления в окружении;

- в разделе 17.6 будет продемонстрирована кастомизация вычисления посредством передачи пользовательских функций в новое окружение;
- в разделе 17.7 будет введено понятие маски данных, призванной размыть границы между окружениями и датафреймами;
- в разделе 17.8 мы познакомимся с новой структурой данных, называемой *quosure*, с помощью которой этот процесс можно облегчить и сделать более естественным.

Требования

В этой главе мы познакомимся с массой важных идей, реализованных в пакете `rlang`. С аналогом этого функционала в базовом R мы встретимся в следующих главах. Кроме того, мы воспользуемся пакетом `lobstr` для исследования древовидной структуры кода.

```
library(rlang)
library(lobstr)
```

Убедитесь также, что вы знакомы с окружениями (раздел 7.2) и датафреймами (раздел 3.6).

17.2 Код как данные

Первая важная идея состоит в том, что код представляет собой данные: вы можете осуществить *захват* (capture) кода и вычислить его подобно тому, как делаете это с данными любого другого типа. Первый способ захвата кода состоит в использовании функции `rlang::expr()`. Вы можете думать об expr() как о функции, возвращающей то, что ей было передано:

```
expr(mean(x, na.rm = TRUE))
#> mean(x, na.rm = TRUE)
expr(10 + 100 + 1000)
#> 10 + 100 + 1000
```

Если говорить точнее, захваченный код называется *выражением* (expression). Выражение не представляет собой какой-то один тип объекта, а является обобщенным термином для любого из четырех типов (*вызов* (call), *символ* (symbol), *константа* (constant) или *список пар* (pairlist)), о чем мы подробно будем говорить в главе 18.

Функция `expr()` позволяет захватывать код, который вы вводите. Для захвата кода, переданного в функцию, нам понадобится другой инструмент, поскольку функция `expr()` нам здесь не поможет:

```
capture_it <- function(x) {
    expr(x)
```

```
}

capture_it(a + b + c)
#> x
```

Таким образом, в этом случае мы должны воспользоваться функцией, специально разработанной для захвата пользовательского ввода в аргументе функции: `enexpr()`. Приставку *en* можно воспринимать как сокращение от *enrich* (расширенная): функция `enexpr()` принимает аргумент с отложенным (ленивым) вычислением и превращает его в выражение:

```
capture_it <- function(x) {
    enexpr(x)
}

capture_it(a + b + c)
#> a + b + c
```

Поскольку в нашей функции `capture_it()` вызывается функция `enexpr()`, мы говорим, что функция автоматически *цитирует* (quote), или квотирует, свой первый аргумент. Больше об этом вы узнаете в разделе 19.2.1.

После захвата выражения вы можете исследовать и изменить его. Поведение сложных выражений при этом напоминает списки. Это означает, что вы можете модифицировать их при помощи операторов `[[` и `$`:

```
f <- expr(f(x = 1, y = 2))

# Добавим новый аргумент
f$z <- 3
f
#> f(x = 1, y = 2, z = 3)

# Или удалим аргумент
f[[2]] <- NULL
f
#> f(y = 2, z = 3)
```

Первым элементом в данном случае является сама вызываемая функция, так что доступ к первому аргументу осуществляется с использованием индекса 2. Подробнее об этом мы будем говорить в разделе 18.3.

17.3 Код как дерево

Для осуществления более сложных манипуляций с выражениями нам необходимо досконально понимать их структуру. За кулисами почти в любом языке программирования код представляется в виде дерева, часто назы-

ваемого *абстрактным синтаксическим деревом* (abstract syntax tree – AST). Язык R отличается от остальных тем, что дает возможность просматривать и изменять это дерево.

Удобным инструментом для представления древовидной структуры кода является функция `str::ast()`. Она принимает на вход некий код и возвращает его древовидную структуру. При этом вызовы функций образуют отдельные ветви дерева и обозначаются при помощи закрашенных прямоугольников. Концевыми, или листовыми, элементами дерева являются символы (такие как `a`) или константы (такие как `"b"`).

```
lobstr::ast(f(a, "b"))
#> ▉─f
#> ├─a
#> └─"b"
```

Вложенные функции на диаграмме отображаются в виде разветвленных деревьев:

```
lobstr::ast(f1(f2(a, b), f3(1, f4(2))))
#> ▉─f1
#> ├─▉─f2
#> │ ├─a
#> │ └─b
#> └─▉─f3
#> ├─1
#> └─▉─f4
#> └─2
```

Поскольку в R любая функция может быть записана в префиксной форме (см. раздел 6.8.2), абсолютно любое выражение может быть представлено в виде дерева:

```
lobstr::ast(1 + 2 * 3)
#> ▉─`+`
#> ├─1
#> └─▉─`*`
#> ├─2
#> └─3
```

Вывод выражений в таком виде существенно облегчает понимание грамматики R, о чем мы поговорим далее в разделе 18.4.

17.4 Код может генерировать код

Подобно тому, как мы рассматриваем древовидную структуру кода, введенного вручную, мы можем использовать некий код для создания новых де-

ревьев. Для этого существует два основных инструмента: функция `call2()` и *расцитирование* (unquoting).

Функция `rlang::call2()` конструирует вызов функции на основе переданных ей компонентов, представляющих функцию для вызова и аргументы, с которыми она должна быть вызвана.

```
call2("f", 1, 2, 3)
#> f(1, 2, 3)
call2("+", 1, call2("*", 2, 3))
#> 1 + 2 * 3
```

Функция `call2()` бывает очень полезна при написании кода, но для интерактивного использования она слишком громоздкая. В качестве альтернативы при построении сложных деревьев можно использовать более простые деревья совместно с шаблоном. В функциях `expr()` и `enexpr()` поддержка этой идеи реализована с помощью *оператора расцитирования* (unquote operator) (`!!`).

Подробно об этой технике мы будем говорить в разделе 19.4, а сейчас вам достаточно знать, что инструкция `!!x` позволяет вставить в выражение дерево кода, сохраненное в переменной `x`. Это помогает при построении сложных деревьев на основе более мелких фрагментов:

```
xx <- expr(x + x)
yy <- expr(y + y)

expr(!!xx / !!yy)
#> (x + x)/(y + y)
```

Обратите внимание, что на выходе сохранился приоритет операций, в результате чего мы получили выражение `(x + x) / (y + y)`, а не `x + x / y + y`, что было бы эквивалентно `x + (x / y) + y`. Это очень важно, особенно если вы подумали, что проще было бы просто сцепить две строки.

Техника с использованием расцитирования может оказаться еще более полезной при заключении логики в функцию, в которой сначала выполняется захват пользовательского выражения при помощи функции `enexpr()`, а затем используется функция `expr()` совместно с оператором расцитирования (`!!`) для создания нового выражения по шаблону. В примере, показанном ниже, мы генерируем выражение для вычисления коэффициента вариации:

```
cv <- function(var) {
    var <- enexpr(var)
    expr(sd(!!var) / mean(!!var))
}

cv(x)
#> sd(x)/mean(x)
cv(x + y)
#> sd(x + y)/mean(x + y)
```

(В данном случае это не так полезно, но в более сложных примерах возможность оперировать такими строительными блоками бывает просто незаменима.)

Важно заметить, что написанная нами функция будет прекрасно работать и с переменными с довольно странными именами:

```
cv(`)`)
#> sd(`)`)/mean(`)`)
```

Обработка таких странных имен[1] – это еще одна причина избегать использования функции `paste()` при генерировании кода на R. Вам эта проблема может показаться надуманной, но если не принимать этот аспект во внимание, при генерировании SQL-запросов в веб-приложениях можно легко столкнуться с так называемым *внедрением SQL* (SQL injection), что может привести к огромным убыткам.

17.5 Исполнение запускает код

Исследование и модификация кода перед запуском – это очень важный инструмент в нашем арсенале. Вторым, не менее важным инструментом является вычисление, т. е. запуск или исполнение, выражения. Вычисление выражения требует наличия окружения, по которому можно будет определить, что в выражении означают те или иные символы. Подробно об окружениях в контексте вычислений мы будем говорить в главе 20.

Основным инструментом вычисления выражений является функция `base::eval()`, принимающая на вход выражение и окружение:

```
eval(expr(x + y), env(x = 1, y = 10))
#> [1] 11
eval(expr(x + y), env(x = 2, y = 100))
#> [1] 102
```

Если опустить окружение, при вычислении выражения будет использовано текущее окружение:

```
x <- 10
y <- 100
eval(expr(x + y))
#> [1] 110
```

Одним из главных преимуществ ручного вычисления выражений является возможность управлять окружением. И на то есть две основные причины:

[1] Чисто технически такие имена называются синтаксически неправильными, и мы затрагивали эту тему в разделе 2.2.1.

- временное переопределение функций для реализации языка, специфичного для предметной области;
- добавление маски данных для возможности ссылаться на переменные в датафрейме, как если бы они находились в окружении.

17.6 Вычисления с измененными функциями

В примере выше мы использовали окружение с именами x и y, привязанными к векторам. Менее очевидным является тот факт, что вы также можете привязывать имена к функциям для переопределения поведения существующих функций. Это очень важная идея, которую мы подробно обсудим в главе 21, в которой будем преобразовывать код R в HTML и LaTeX. В коде, показанном ниже, вы можете почувствовать всю мощь, заложенную в этот прием. Здесь мы вычисляем код в особом окружении, в котором операторы * и + переопределены для работы со строками, а не с числами:

```
string_math <- function(x) {
    e <- env(
        caller_env(),
        `+` = function(x, y) paste0(x, y),
        `*` = function(x, y) strrep(x, y)
    )

    eval(enexpr(x), e)
}

name <- "Hadley"
string_math("Hello " + name)
#> [1] "Hello Hadley"
string_math(("x" * 2 + "-y") * 3)
#> [1] "xx-yxx-yxx-y"
```

В пакете dplyr эта идея используется в полный рост при генерировании SQL-запросов к удаленной базе данных посредством запуска кода в окружении:

```
library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
```

```
#>     intersect, setdiff, setequal, union

con <- DBI::dbConnect(RSQLite::SQLite(), filename = ":memory:")
mtcars_db <- copy_to(con, mtcars)
mtcars_db %>%
    filter(cyl > 2) %>%
    select(mpg:hp) %>%
    head(10) %>%
    show_query()
#> <SQL>
#> SELECT `mpg`, `cyl`, `disp`, `hp`
#> FROM `mtcars`
#> WHERE (`cyl` > 2.0)
#> LIMIT 10

DBI::dbDisconnect(con)
```

17.7 Вычисления с измененными данными

Переназначение функций при вычислении выражений – это очень мощная техника, но она требует больших ресурсов в виде вмешательства в код. Гораздо проще с точки зрения практического применения бывает запустить вычисление с поиском переменных в датафрейме вместо окружения. Эта идея лежит в основе реализации базовых функций `subset()` и `transform()`, а также многих функций из коллекции пакетов tidyverse, таких как `ggplot2::aes()` и `dplyr::mutate()`. В этих случаях можно использовать функцию `eval()`, но стоит помнить о нескольких потенциально опасных ловушках, которые мы обсудим в разделе 20.6. Поэтому мы воспользуемся функцией `rlang::eval_tidy()`.

Помимо выражения и окружения, функция `eval_tidy()` также принимает *маску данных* (data mask), обычно в виде датафрейма:

```
df <- data.frame(x = 1:5, y = sample(5))
eval_tidy(expr(x + y), df)
#> [1] 2 6 5 9 8
```

Вычисление выражения с маской данных бывает полезным при выполнении интерактивной аналитики, поскольку позволяет использовать запись `x + y` вместо `df$x + df$y`. Но за такое удобство приходится платить возникающей неопределенностью записи. В разделе 20.4 мы узнаем, как можно избежать возникновения конфликтов имен при помощи префиксов `.data` и `.env`.

Этот шаблон можно заключить в функцию с использованием функции `enexpr()`. В результате мы получим функцию, очень похожую на базовую функцию `base::with()`:

```
with2 <- function(df, expr) {
    eval_tidy(enexpr(expr), df)
```

```
}

with2(df, x + y)
#> [1] 2 6 5 9 8
```

К сожалению, в этой функции присутствует небольшая ошибка, и для ее устранения нам понадобится новая структура данных.

17.8 Quosure

Чтобы сделать существующую проблему более очевидной, давайте изменим функцию `with2()`. Проблема, о которой мы говорим, присутствует и без этих изменений, но так заметить ее сложнее.

```
with2 <- function(df, expr) {
    a <- 1000
    eval_tidy(enexpr(expr), df)
}
```

Проблема проявится при использовании функции `with2()` с обращением к переменной `a`. Нам бы хотелось, чтобы значение переменной `a` бралось из существующей привязки (10), а не из внутренней привязки в функции (1000):

```
df <- data.frame(x = 1:3)
a <- 10
with2(df, x + a)
#> [1] 1001 1002 1003
```

Источник проблемы кроется в необходимости вычислять захваченное выражение в окружении, в котором оно было написано (где значение `a` равно 10), а не в окружении внутри функции `with2()` (где значение `a` равно 1000).

К счастью, мы можем решить эту проблему с помощью новой структуры данных, именуемой `quosure`, которая связывает выражение с окружением. Функция `eval_tidy()` умеет работать с этой структурой, а все, что нам нужно сделать, – это заменить вызов функции `enexpr()` на `enquo()`:

```
with2 <- function(df, expr) {
    a <- 1000
    eval_tidy(enquo(expr), df)
}

with2(df, x + a)
#> [1] 11 12 13
```

Всякий раз при применении маски данных вы должны использовать функцию `enquo()` вместо `enexpr()`. Подробнее об этом мы будем говорить в главе 20.

18

Выражения

18.1 Введение

Чтобы выполнять вычисления в языке, сначала необходимо понять его структуру. Это требует освоения новой лексики, некоторых новых инструментов и новых подходов к коду на языке R. Первое, что нужно осознать, – это различие между операцией и ее результатом. Рассмотрим для примера следующий код, в котором значение `x` умножается на 10 и результат присваивается новой переменной `y`. Он не сработает, поскольку переменная `x` здесь не определена:

```
y <- x * 10
#> Error in eval(expr, envir, enclos): object 'x' not found
```

Было бы неплохо уметь захватывать намерения кода без необходимости немедленно исполнять его. Иными словами, мы бы хотели отделить описание действия от самого действия.

Один из способов состоит в использовании функции `rlang::expr()`:

```
z <- rlang::expr(y <- x * 10)
z
#> y <- x * 10
```

Функция `expr()` возвращает *выражение* (expression), т. е. объект, захватывающий структуру кода, но не выполняющий его. При наличии выражения вы можете вычислить его с помощью функции `base::eval()`:

```
x <- 4
eval(z)
y
#> [1] 40
```

В этой главе мы сосредоточимся на структурах данных, лежащих в основе выражений. Усвоение этого материала позволит вам исследовать и модифи-

цировать захваченный код, а также генерировать код посредством другого кода. К функции `expr()` мы еще вернемся в главе 19, а к `eval()` – в главе 20.

Структура главы

- В разделе 18.2 мы представим идею абстрактного синтаксического дерева (AST) и увидим, какая древовидная структура лежит в основе всего кода на R.
- В разделе 18.3 мы погрузимся в реализацию структуры, служащей основой для AST: константы, символы и вызовы, которые вместе именуются выражениями.
- Раздел 18.4 будет посвящен парсингу, или разбору, кода, т. е. преобразованию линейной последовательности символов кода в AST, а также описанию некоторых связанных с этим особенностей грамматики языка R.
- В разделе 18.5 мы узнаем, как можно использовать рекурсивные функции для выполнения вычислений в языке, и напишем функции, производящие вычисления с выражениями.
- В разделе 18.6 мы поговорим о трех специализированных структурах данных: *списках пар* (pairlist), пропущенных аргументах и векторах выражений.

Требования

Перед чтением этой главы я настоятельно рекомендую вам ознакомиться с предыдущей главой, в которой мы заложили основы метапрограммирования в R, а также описали присущую ему базовую лексику. С целью выполнения примеров из этой главы вам потребуется пакет `rlang` (https://rlang.r-lib.org) для захвата и вычисления выражений, а также пакет `lobstr` (https://lobstr.r-lib.org) для их визуализации.

```
library(rlang)
library(lobstr)
```

18.2 Абстрактные синтаксические деревья

Выражения также можно называть *абстрактными синтаксическими деревьями* (abstract syntax trees – AST) по причине иерархической природы структуры кода, позволяющей представить код в виде дерева. Понимание структуры этого дерева крайне важно для исследования и модификации выражений (т. е. метапрограммирования).

18.2.1 Внешнее отображение

Начнем с некоторых соглашений по поводу представления AST на примере простого вызова функции `f(x, "y", 1)` с отображением всех ее компонентов. Я буду рисовать деревья двумя способами[1]:

- вручную (с помощью приложения OmniGraffle), как показано на рис. 18.1;

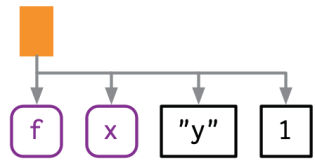


Рис. 18.1 AST в ручном режиме

- посредством функции `lobstr::ast()`:

```
lobstr::ast(f(x, "y", 1))
#> ▉─f
#> ├─x
#> ├─"y"
#> └─1
```

Оба подхода следуют принятым соглашениям, насколько это возможно:

- концевыми, или листовыми, элементами деревьев могут быть либо *символы* (symbol), такие как `f` и `x`, либо *константы* (constant), такие как `1` или `"y"`. Символы на диаграмме отображаются в виде фиолетовых блоков с закругленными углами. Блоки с константами отображаются черным цветом без закруглений. Строки и символы можно легко перепутать, в связи с чем первые принято заключать в кавычки;
- ветви дерева представляют собой объекты *вызовов* (call), соответствующие вызовам функций, и отображаются в виде закрашенных оранжевых блоков. В нашем случае первым дочерним объектом (`f`) является вызываемая функция, а последующими (`x`, `"y"` и `1`) – ее аргументы.

При интерактивном вызове функции `ast()` вы увидите соответствующее цветовое наполнение диаграмм, которое в книге отсутствует.

Пример, показанный выше, содержит всего один вызов функции, в связи с чем результирующее дерево получилось таким простым. Большинство выражений включают в себя гораздо большее количество вызовов, что приво-

[1] Для визуализации структуры более сложного кода вы можете также воспользоваться встроенным в RStudio инструментом для отображения деревьев, не подчиняющимся общим соглашениям, зато позволяющим визуализировать очень сложные древовидные структуры в интерактивном режиме. Попробуйте сами с помощью команды `View(expr(f(x, "y", 1)))`.

дит к образованию более разветвленных деревьев. Рассмотрим дерево AST для выражения `f(g(1, 2), h(3, 4, i()))`, показанное на рис. 18.2:

```
lobstr::ast(f(g(1, 2), h(3, 4, i())))
#> ▉─f
#> ├─▉─g
#> │ ├─1
#> │ └─2
#> └─▉─h
#> ├─3
#> ├─4
#> └─▉─i
```

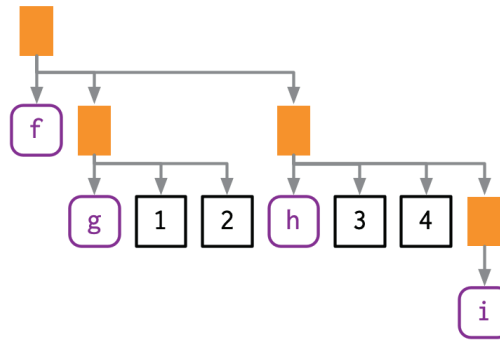


Рис. 18.2 AST для сложного выражения

Рукописную диаграмму можно читать слева направо, игнорируя при этом вертикальные смещения, а диаграмму, построенную при помощи пакета `lobstr`, – сверху вниз, не обращая внимания на положение по горизонтали. Глубина внутри дерева определяется степенью вложенности вызовов функций. Кроме того, она отвечает за порядок вычисления, который обычно направлен из глубины дерева к его корню, но так бывает не всегда, что мы видели на примере отложенных, или ленивых, вычислений в разделе 6.5. Обратите внимание на функцию `i()`, вызванную без аргументов. Она представляет собой ветвь с единственным (символьным) концевым элементом.

18.2.2 Компоненты без кода

Вас может заинтересовать, что именно делает эти синтаксические деревья абстрактными. Откуда взялось это слово? Дело в том, что в этих деревьях представлены только структурные подробности кода, без разделителей и комментариев:

```
ast(
f(x, y) # important!
)
#> ▉─f
```

```
#> ├─x
#> └─y

lobstr::ast(y <- x)
#> ├─`<-`
#> ├─y
#> └─x

lobstr::ast(y < -x)
#> ├─`<`
#> ├─y
#> └─├─`-`
#>   └─x
```

18.2.3 Инфиксные вызовы

Каждый вызов в R может быть представлен в виде дерева по причине того, что любой из них может быть выражен в префиксной форме (см. раздел 6.8.1). Давайте снова вернемся к примеру `y <- x * 10`: какие функции здесь вызываются? Ответить на этот вопрос несколько сложнее по сравнению с простым выражением `f(x, 1)` из-за наличия двух инфиксных операторов: `<-` и `*`. Это означает, что эти две строки кода эквивалентны:

```
y <- x * 10
`<-`(y, `*`(x, 10))
```

Дерево в обоих случаях будет таким, как на рис. 18.3[1]:

```
lobstr::ast(y <- x * 10)
#> ├─`<-`
#> ├─y
#> └─├─`*`
#>   ├─x
#>   └─10
```

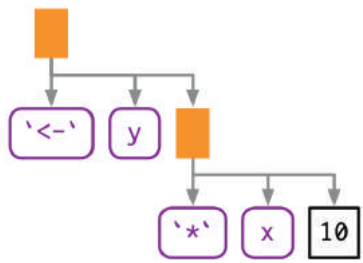


Рис. 18.3 AST для выражения `y <- x * 10`

[1] Имена непрефиксных функций являются синтаксически неправильными, так что я обрамляю их знаками обратного апострофа (`).

Между AST представленных выше выражений нет никакой разницы, и если сгенерировать выражение на основе префиксных вызовов, R все равно выведет его в инфиксной форме:

```
expr(`<-`(y, `*`(x, 10)))
#> y <- x * 10
```

Порядок, в котором вычисляются инфиксные операторы, определяется правилами приоритета операторов, и в разделе 18.4.1 мы воспользуемся функцией `lobstr::ast()` для знакомства с ними.

18.2.4 Упражнения

1. Воссоздайте код по представленным ниже деревьям:

```
#> ▉─f
#> └─▉─g
#> └─▉─h
#> ▉─`+`
#> ├─▉─`+`
#> │ ├─1
#> │ └─2
#> └─3
#> ▉─`*`
#> ├─▉─`(`
#> │ └─▉─`+`
#> │ ├─x
#> │ └─y
#> └─z
```

2. Нарисуйте от руки дерево для следующих выражений, а затем проверьте себя с помощью функции `lobstr::ast()`:

```
f(g(h(i(1, 2, 3))))
f(1, g(2, h(3, i())))
f(g(1, 2), h(3, i(4, 5)))
```

3. Что происходит в представленных ниже деревьях? Подсказка: внимательно изучите справку, доступную по команде `?"^"`:

```
lobstr::ast(`x` + `y`)
#> ▉─`+`
#> ├─x
#> └─y
lobstr::ast(x ** y)
#> ▉─`^`
#> ├─x
#> └─y
lobstr::ast(1 -> x)
```

```
#> ▉─`<-`
#> ├─x
#> └─1
```

4. Какие особенности содержатся в дереве, показанном ниже? Подсказка: перечитайте раздел 6.2.1.

```
lobstr::ast(function(x = 1, y = 2) {})
#> ▉─`function`
#> ├─▉─x = 1
#> │ └─y = 2
#> ├─▉─`{`
#> └─<inline srcref>
```

5. Как будет выглядеть дерево для условной конструкции `if` со множественными вхождениями `else if`? Почему так?

18.3 Выражения

Совокупно все структуры данных, присутствующие в AST, называются выражением. Выражение представляет собой любой член набора базовых типов, созданных в процессе парсинга, или разбора: скалярная константа, символ, вызов или список пар. Эти структуры данных используются для представления захваченного функцией `expr()` кода, и `is_expression(expr(...))` всегда будет возвращать истину[1]. Объекты констант, символов и вызовов играют очень важную роль, и именно им мы посвятим дальнейшее обсуждение. Списки пар и пустые символы являются более специализированными объектами, и подробно о них мы будем говорить в разделах 18.6.1 и 18.6.2.

В документации к базовому R термин *выражение* используется для обозначения двух вещей. Помимо определения, данного выше, он применяется для указания на тип объекта, возвращаемый функциями `expression()` и `parse()`, который обычно представляет собой список выражений, как было показано выше. В этой книге я буду именовать такие объекты *векторами выражений* (expression vector), о которых мы детально поговорим в разделе 18.6.3.

18.3.1 Константы

Скалярные *константы* (constant) являются простейшими компонентами AST. Если говорить точнее, константы могут представлять либо `NULL`, либо атомарный вектор единичной длины (или скаляр, как мы уже видели в разделе 3.2.1), например `TRUE`, `1L`, `2.5` или `"x"`. Проверку на константу вы можете осуществить с помощью функции `rlang::is_syntactic_literal()`.

[1] Вполне возможно вставить любой другой базовый объект в выражение, но необходимость в этом возникает крайне редко. Мы вернемся к этой идее в разделе 19.4.7.

Константы являются самоцитируемыми в том смысле, что выражения, используемые для представления констант, – это те же самые константы:

```
identical(expr(TRUE), TRUE)
#> [1] TRUE
identical(expr(1), 1)
#> [1] TRUE
identical(expr(2L), 2L)
#> [1] TRUE
identical(expr("x"), "x")
#> [1] TRUE
```

18.3.2 Символы

Символы (symbol) представляют собой имена объектов, такие как `x`, `mtcars` или `mean`. В базовом R термины *символ* и *имя* (name) являются взаимозаменяемыми (т. е. запись `is.name()` эквивалентна `is.symbol()`), но в данной книге я решил использовать термин *символ* по причине того, что *имя* имеет множество других значений.

Символ может быть создан двумя способами: путем захвата кода, ссылающегося на объект, посредством функции `expr()` или превратив строку в символ с помощью функции `rlang::sym()`:

```
expr(x)
#> x
sym("x")
#> x
```

Преобразовать символ обратно в строку вы можете с помощью функции `as.character()` или `rlang::as_string()`. При этом преимущество последней заключается в явном предупреждении о том, что в результате вы получите символьный вектор единичной длины.

```
as_string(expr(x))
#> [1] "x"
```

Символы легко распознать, поскольку они записываются без кавычек, функция `str()` применительно к ним говорит о том, что мы имеем дело с символом, а функция `is.symbol()` возвращает `TRUE`:

```
str(expr(x))
#> symbol x
is.symbol(expr(x))
#> [1] TRUE
```

Символьный тип не векторизован, так что символ всегда представляет объект единичной длины. Если вам необходимо создать несколько символов, вы можете поместить их в список, например с использованием функции `rlang::syms()`.

18.3.3 Вызовы

Объект *вызов* (call) представляет собой захваченный вызов функции. Эти объекты хранятся в виде специальных списков[1], в которых первый элемент соответствует вызываемой функции (обычно это символ), а остальные отвечают за аргументы. В AST объекты вызова характеризуются образованием ветвей дерева, поскольку одна функция может быть вызвана из другой, что ведет к образованию иерархии.

Идентифицировать объект вызова можно, выведя его на экран – он будет выглядеть как вызов функции. Вас может запутать то, что функции `typeof()` и `str()` возвращают для объекта вызова значение `"language"`[2], но при этом функция `is.call()` вернет `TRUE`:

```
lobstr::ast(read.table("important.csv", row.names = FALSE))
#> ▉─read.table
#> ├─"important.csv"
#> └─row.names = FALSE
x <- expr(read.table("important.csv", row.names = FALSE))

typeof(x)
#> [1] "language"
is.call(x)
#> [1] TRUE
```

18.3.3.1 Извлечение подмножеств

Объекты вызова обычно ведут себя как списки, что позволяет использовать применительно к ним стандартные механизмы извлечения подмножеств. Первым элементом списка при этом является вызываемая функция, часто представленная символом:

```
x[[1]]
#> read.table
is.symbol(x[[1]])
#> [1] TRUE
```

Остальные элементы списка представляют аргументы функции:

```
as.list(x[-1])
#> [[1]]
#> [1] "important.csv"
#>
#> $row.names
#> [1] FALSE
```

---

[1] Если быть точными, вызовы представлены в виде *списков пар* (см. раздел 18.6.1), хотя это различие обычно не имеет большой важности.

[2] Избегайте применения функции `is.language()`, которая для символов, вызовов и векторов выражений возвращает `TRUE`.

Отдельные аргументы можно извлечь с помощью оператора [[или, если речь идет об именованных элементах, посредством $:

```
x[[2]]
#> [1] "important.csv"
x$row.names
#> [1] FALSE
```

Количество аргументов, присутствующих в объекте вызова, можно определить, вычтя единицу из его длины:

```
length(x) - 1
#> [1] 2
```

Извлечение конкретных аргументов из объектов вызова – задача не из легких по причине применения гибких правил к сопоставлению аргументов в R. Фактически нужный вам аргумент может располагаться на любой позиции, с полным именем, сокращенным или без имени вовсе. Для решения этой проблемы можно воспользоваться функцией `rlang::call_standardise()`, которая стандартизирует все аргументы для использования с полными именами:

```
rlang::call_standardise(x)
#> read.table(file = "important.csv", row.names = FALSE)
```

Примечание. Если в функции используется аргумент ..., все ее аргументы стандартизировать не удастся.

Объекты вызова можно изменять так же, как списки:

```
x$header <- TRUE
x
#> read.table("important.csv", row.names = FALSE, header = TRUE)
```

18.3.3.2 Позиция функции

Первый элемент объекта вызова называется *позицией функции* (function position). В нем содержится функция, которая будет вызываться при вычислении выражения, обычно представленная в виде символа[1]:

```
lobstr::ast(foo())
#> └─foo
```

Поскольку R позволяет обрамлять имена функций кавычками, парсер преобразует такую запись в символ:

[1] Иногда функция может быть представлена числом, как в выражении `3()`. Но такой вызов всегда будет приводить к ошибке вычисления, т. к. число не является функцией.

```
lobstr::ast("foo"())
#> ▉─foo
```

Иногда, однако, функция не находится в текущем окружении, и вам необходимо выполнить определенные действия для ее извлечения. Это возможно, если функция располагается в другом пакете, является методом объекта R6 или была создана при помощи фабрики функций. В подобных случаях позиция функции будет занята другим вызовом, как показано на рис. 18.4:

```
lobstr::ast(pkg::foo(1))
#> ▉─▉─`::`
#> │ ├─pkg
#> │ └─foo
#> └─1
lobstr::ast(obj$foo(1))
#> ▉─▉─`$`
#> │ ├─obj
#> │ └─foo
#> └─1
lobstr::ast(foo(1)(2))
#> ▉─▉─foo
#> │ └─1
#> └─2
```

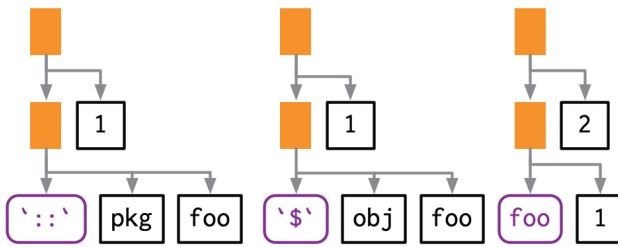


Рис. 18.4 AST при отсутствии функции в текущем окружении

18.3.3.3 Конструирование вызовов

Вы можете сконструировать объект вызова из его компонентов при помощи функции `rlang::call2()`. Первым аргументом эта функция принимает имя функции для вызова (в виде строки, символа или другого вызова). Остальные аргументы будут переданы вместе с вызовом:

```
call2("mean", x = expr(x), na.rm = TRUE)
#> mean(x = x, na.rm = TRUE)
call2(expr(base::mean), x = expr(x), na.rm = TRUE)
#> base::mean(x = x, na.rm = TRUE)
```

Инфиксные вызовы, созданные таким путем, будут выводиться как обычно.

```
call2("<-", expr(x), 10)
#> x <- 10
```

Создание сложных выражений при помощи функции `call2()` может показаться вам чересчур громоздким. В главе 19 мы рассмотрим еще одну технику.

18.3.4 Резюме

В табл. 18.1 сведено отображение различных типов выражений при использовании функций `str()` и `typeof()`.

Таблица 18.1 Разные типы выражений в функциях `str()` и `typeof()`

	`str()`	`typeof()`
Скалярные константы	logi/int/num/chr	logical/integer/double/character
Символы	symbol	symbol
Объекты вызова	language	language
Списки пар	Dotted pair list	pairlist
Векторы выражений	expression()	expression

В базовом R и в пакете `rlang` присутствуют функции для проверки каждого типа ввода, при этом сами покрытые функции немного отличаются, что видно в табл. 18.2. Вы можете легко отличать их, поскольку все базовые функции начинаются с `is.`, а функции из пакета `rlang` – с `is_`.

Таблица 18.2 Разные типы выражений в базовом R и в пакете `rlang`

	Базовый R	`rlang`
Скалярные константы	—	`is_syntactic_literal()`
Символы	`is.symbol()`	`is_symbol()`
Объекты вызова	`is.call()`	`is_call()`
Списки пар	`is.pairlist()`	`is_pairlist()`
Векторы выражений	`is.expression()`	—

18.3.5 Упражнения

1. Какие два из шести типов атомарных векторов не могут присутствовать в выражениях? Почему? И почему нельзя создать выражение, содержащее атомарный вектор длиной больше единицы?
2. Что произойдет, если применить к объекту вызова операцию по извлечению подмножеств для удаления первого элемента? Например, `expr(read.csv("foo.csv", header = TRUE))[-1]`. Почему?
3. Поясните разницу между приведенными ниже объектами вызова.

```
x <- 1:10
call2(median, x, na.rm = TRUE)
```

```
call2(expr(median), x, na.rm = TRUE)
call2(median, expr(x), na.rm = TRUE)
call2(expr(median), expr(x), na.rm = TRUE)
```

4. Для показанных ниже вызовов функция `rlang::call_standardise()` работает не очень хорошо. Почему? Что такого особенного в функции `mean()`?

```
call_standardise(quote(mean(1:10, na.rm = TRUE)))
#> mean(x = 1:10, na.rm = TRUE)
call_standardise(quote(mean(n = T, 1:10)))
#> mean(x = 1:10, n = T)
call_standardise(quote(mean(x = 1:10, , TRUE)))
#> mean(x = 1:10, , TRUE)
```

5. Почему приведенный ниже код не имеет смысла?

```
x <- expr(foo(x = 1))
names(x) <- c("x", "y")
```

6. Сконструируйте выражение `if(x > 1) "a" else "b"` с использованием нескольких вызовов функции `call2()`. Как структура кода отражает структуру AST?

18.4 Разбор и грамматика

Мы достаточно много внимания уделили выражениям и AST, но не говорили о том, как выражения создаются на основе написанного вами кода, например `"x + y"`. Процесс чтения языком программирования строк и конструирования на их основе выражений называется *разбором*, или *парсингом* (parsing), а правила, которые при этом используются, – *грамматикой* (grammar) языка. В этом разделе мы воспользуемся функцией `lobstr::ast()` для исследования внутренностей грамматики языка R, а затем покажем, как можно легко выполнять двусторонние преобразования между выражениями и строками.

18.4.1 Приоритет операций

Использование инфиксных функций порождает два вида неопределенности[1]. Первая неопределенность связана с тем, что делает выражение `1 + 2 * 3`. Мы

[1] Эти неопределенности отсутствуют в языках программирования, в которых представлены только префиксные и постфиксные вызовы. Для интереса можно сравнить простые арифметические операции в языках Lisp (префиксная запись) и Forth (постфиксная запись). В Lisp вы бы написали: `(* (+ 1 2) 3));`. Здесь двусмысленность избегается при помощи расставленных скобок. В то же время в языке Forth запись выглядела бы так: `1 2 + 3 *;`. Здесь не требуются скобки, но для чтения такая запись сложнее.

получим 9 (т. е. `(1 + 2) * 3`) или 7 (т. е. `1 + (2 * 3)`)? Иными словами, какое из приведенных на рис. 18.5 деревьев будет использовано?

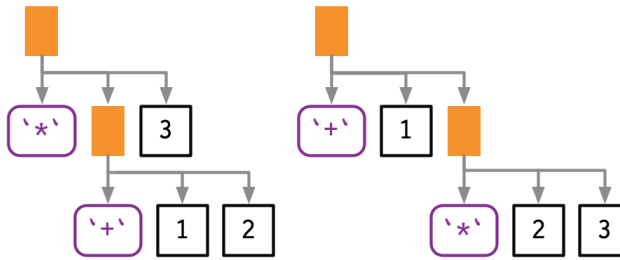


Рис. 18.5 Два возможных AST для выражения `1 + 2 * 3`

В языках программирования для разрешения подобных конфликтов используются соглашения, называемые правилами *приоритета операций* (operator precedence). Вы можете воспользоваться функцией `ast()` и посмотреть, какой вариант предпочтет R:

```
lobstr::ast(1 + 2 * 3)
#> ▉─`+`
#> ├─1
#> └─▉─`*`
#>   ├─2
#>   └─3
```

Предугадать приоритет арифметических операций обычно не составляет труда, поскольку в подавляющем большинстве языков программирования используются те же правила, которые вы назубок выучили в младших классах школы.

Что касается других операторов, здесь может быть все сложнее. В R есть один довольно любопытный пример, связанный с тем, что оператор `!` обладает намного более низким приоритетом (т. е. более слабой привязкой), чем можно было ожидать. Это позволяет писать полезные операции вроде следующей:

```
lobstr::ast(!x %in% y)
#> ▉─`!`
#> └─▉─`%in%`
#>   ├─x
#>   └─y
```

В R присутствует порядка 30 инфиксных операторов, разделенных на 18 групп приоритетов. Вряд ли кто-то помнит наизусть всю последовательность этих операторов, к тому же вы всегда можете воспользоваться справкой `?Syntax`, чтобы узнать о приоритетах нужных вам операторов. А в случаях неопределенности используйте скобки!

```
lobstr::ast((1 + 2) * 3)
#> ▆─`*`
#> ├─▆─`(`
#> │ └─▆─`+`
#> │   ├─1
#> │   └─2
#> └─3
```

Обратите внимание, что скобки в AST присутствуют в виде отдельных вызовов функции `(`.

18.4.2 Ассоциативность

Второй источник неопределенности связан с повторным использованием одной и той же инфиксной функции. Например, выражение `1 + 2 + 3` эквивалентно записи `(1 + 2) + 3` или `1 + (2 + 3)`? В данном случае это не имеет значения, поскольку `x + (y + z) == (x + y) + z`, т. е. операция сложения является ассоциативной. В то же время в некоторых классах S3 операция `+` реализована не ассоциативно. К примеру, в пакете `ggplot2` оператор `+` перегружается для построения сложных графиков на основе простых составляющих. При этом данная операция не является ассоциативной, поскольку слои, выведенные раньше, отображаются под слоями, добавленными позже (например, `geom_point() + geom_smooth()` и `geom_smooth() + geom_point()` приведут к выводу разных графиков).

В R большинство операторов *левоассоциативны* (left-associative), т. е. операции, стоящие слева, выполняются первыми:

```
lobstr::ast(1 + 2 + 3)
#> ▆─`+`
#> ├─▆─`+`
#> │ ├─1
#> │ └─2
#> └─3
```

Но есть и два исключения: оператор возведения в степень (^) и оператор присваивания (<-):

```
lobstr::ast(2^2^3)
#> ▆─`^`
#> ├─2
#> └─▆─`^`
#>   ├─2
#>   └─3
lobstr::ast(x <- y <- z)
#> ▆─`<-`
#> ├─x
#> └─▆─`<-`
#>   ├─y
#>   └─z
```

18.4.3 Парсинг и депарсинг

В большинстве случаев вы вводите текст кода в консоль, а R заботится о преобразовании введенных вами символов в AST. Но иногда бывает, что ваш код уже сохранен в виде строки, которую вам необходимо распарсить самостоятельно. Вы можете сделать это с помощью функции `rlang::parse_expr()`:

```
x1 <- "y <- x + 10"
x1
#> [1] "y <- x + 10"
is.call(x1)
#> [1] FALSE

x2 <- rlang::parse_expr(x1)
x2
#> y <- x + 10
is.call(x2)
#> [1] TRUE
```

Функция `parse_expr()` всегда возвращает одно выражение. Если у вас есть несколько выражений, разделенных символами `;` или `\n`, вам нужно будет воспользоваться функцией `rlang::parse_exprs()`. Она возвращает список выражений:

```
x3 <- "a <- 1; a + 1"
rlang::parse_exprs(x3)
#> [[1]]
#> a <- 1
#>
#> [[2]]
#> a + 1
```

Если вам достаточно часто приходится работать с кодом, представленным в виде строк, вам стоит пересмотреть свои процессы. Ознакомьтесь с главой 19 на предмет более безопасного генерирования выражений с использованием техники квазицитирования.

Примечание. В базовом R эквивалентом функции `parse_exprs()` является функция `parse()`. Она немного более сложна в применении, поскольку главным образом предназначена для парсинга кода R, сохраненного в файлах. Вам необходимо передать вашу строку с кодом в аргумент `text`, и функция вернет вектор выражений (см. раздел 18.6.3). Я рекомендую преобразовывать вывод в список:

```
as.list(parse(text = x1))
#> [[1]]
#> y <- x + 10
```

Обратной операцией парсингу является *депарсинг* (deparsing), который может пригодиться, если у вас есть выражение и вам необходимо получить строку, которая бы его сгенерировала. Это происходит автоматически при выводе выражений, также вы можете воспользоваться функцией `rlang::expr_text()`, как показано ниже:

```
z <- expr(y <- x + 10)
expr_text(z)
#> [1] "y <- x + 10"
```

Парсинг и депарсинг не являются строго симметричными операциями, поскольку в результате парсинга генерируется *абстрактное* синтаксическое дерево. Это означает потерю знаков обратного апострофа (`) в именах, комментариев и разделителей:

```
cat(expr_text(expr({
    # Это комментарий
    x <-            `x` + 1
})))
#> {
#>     x <- x + 1
#> }
```

Примечание. В базовом R необходимо соблюдать осторожность при использовании функции `deparse()`: она возвращает символьный вектор с одним элементом для каждой строки. При использовании этой функции нужно помнить, что длина вывода может быть больше единицы, и соответствующим образом реагировать на это.

18.4.4 Упражнения

1. В R круглые скобки могут использоваться немного по-разному, что видно на примере двух приведенных ниже вызовов:

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

   Сравните и противопоставьте два этих примера с помощью AST.
2. Оператор = также может использоваться двумя способами. Сконструируйте простой пример, демонстрирующий оба способа.
3. Какой ответ даст выражение `-2^2`? 4 или –4? И почему?
4. Что вернет выражение `!1 + !1`? Почему?
5. Почему работает выражение `x1 <- x2 <- x3 <- 0`? Назовите две причины.
6. Сравните AST двух выражений: `x + y %+% z` и `x ^ y %+% z`. Что вы можете сказать о правилах приоритета для пользовательских инфиксных функций?

7. Что произойдет, если вызвать функцию `parse_expr()` со строкой, генерирующей несколько выражений? Пример: `parse_expr("x + 1; y + 1")`.
8. Что случится, если вы попытаетесь выполнить парсинг некорректного выражения? Примеры: `"a +"` или `"f())"`.
9. При передаче функции `deparse()` длинных строк она генерирует векторы. К примеру, следующий вызов вернет вектор длины 2:

```
expr <- expr(g(a + b + c + d + e + f + g + h + i + j + k + l +
  m + n + o + p + q + r + s + t + u + v + w + x + y + z))

deparse(expr)
```

   Что вместо этого делает функция `expr_text()`?
10. Функция `pairwise.t.test()` полагается на то, что функция `deparse()` всегда будет возвращать символьный вектор единичной длины. Можете ли вы сконструировать вход, который нарушит это допущение? Что в этом случае произойдет?

18.5 Проход по AST с помощью рекурсивных функций

В заключение этой главы мы воспользуемся всеми полученными знаниями об AST для решения более сложных задач. Мы обратимся к базовому пакету `codetools`, в котором представлены две полезные функции:

- функция `findGlobals()` обнаруживает все глобальные переменные, используемые функцией. Она может вам пригодиться в случае необходимости выполнить проверку того, что ваша функция по ошибке не полагается на переменные, определенные в родительском окружении;
- функция `checkUsage()` проверяет код на предмет наличия распространенных проблем, включая неиспользуемые локальные переменные и параметры, а также частичное соответствие аргументов.

Понять полное предназначение этих функций достаточно сложно, и мы не будем сильно углубляться в детали. Вместо этого мы сосредоточимся на общей идее, состоящей в рекурсивном проходе по деревьям AST. *Рекурсивные функции* (recursive function) идеально подходят для анализа древовидных структур данных, поскольку состоят из двух частей, обрабатывающих разные компоненты дерева:

- *рекурсивный случай* (recursive case) выполняется при обработке узлов дерева. Обычно вы выполняете одно и то же действие для всех потомков узла, запуская для этого рекурсивную функцию, после чего собираете воедино полученные результаты. В случае с выражениями вам необходимо обрабатывать объекты вызова и списки пар (аргументы функций);

- *базовый случай* (base case) служит для обработки концевых, или листовых, узлов дерева. В базовом случае обеспечивается окончание цикла выполнения рекурсивной функции путем непосредственного вычисления простейшего выражения. В случае с выражениями на этом этапе вы будете обрабатывать символы и константы.

Для облегчения понимания этого шаблона нам понадобятся две вспомогательные функции. Для начала мы определим функцию `expr_type()`, которая будет возвращать строку `"constant"` для констант, `"symbol"` для символов, `"call"` для объектов вызова, `"pairlist"` для списков пар и `"type"` для всего остального:

```
expr_type <- function(x) {
    if (rlang::is_syntactic_literal(x)) {
        "constant"
    } else if (is.symbol(x)) {
        "symbol"
    } else if (is.call(x)) {
        "call"
    } else if (is.pairlist(x)) {
        "pairlist"
    } else {
        typeof(x)
    }
}

expr_type(expr("a"))
#> [1] "constant"
expr_type(expr(x))
#> [1] "symbol"
expr_type(expr(f(1, 2)))
#> [1] "call"
```

Теперь объединим эту функцию с оберткой вокруг вызова функции `switch`:

```
switch_expr <- function(x, ...) {
    switch(expr_type(x),
        ...,
        stop("Don't know how to handle type ", typeof(x), call. = FALSE)
    )
}
```

Вооружившись этими двумя функциями, мы можем написать базовый шаблон для любой функции, проходящей по AST с использованием функции `switch()` (см. раздел 5.2.3):

```
recurse_call <- function(x) {
    switch_expr(x,
        # Базовые случаи
        symbol = ,
```

```
        constant = ,

        # Рекурсивные случаи
        call = ,
        pairlist =
    )
}
```

Обычно проще всего разрешаются базовые случаи, так что мы сначала прописываем их, после чего проверяем результаты. С рекурсивными случаями иногда приходится повозиться подольше, нередко с применением техник функционального программирования.

18.5.1 Находим F и T

Начнем с функции, определяющей, использует ли другая функция логические аббревиатуры `T` или `F`, поскольку их использование зачастую считается дурным тоном в программировании. Наша цель – вернуть `TRUE`, если вход содержит логические аббревиатуры, и FALSE в противном случае.

Давайте для начала сравним типы `T` и `TRUE`:

```
expr_type(expr(TRUE))
#> [1] "constant"

expr_type(expr(T))
#> [1] "symbol"
```

Разбор `TRUE` выдал нам логический вектор единичной длины, тогда как `T` была определена как имя (символ). Это дает нам почву для написания базовых случаев для нашей рекурсивной функции: константа не может быть логической аббревиатурой, а символ является аббревиатурой, если он представлен литерой `"F"` или `"T"`:

```
logical_abbr_rec <- function(x) {
    switch_expr(x,
        constant = FALSE,
        symbol = as_string(x) %in% c("F", "T")
    )
}

logical_abbr_rec(expr(TRUE))
#> [1] FALSE
logical_abbr_rec(expr(T))
#> [1] TRUE
```

При написании функции `logical_abbr_rec()` я сделал допущение о том, что на вход будет подаваться выражение, поскольку это сделает рекурсивную операцию проще. В то же время при создании рекурсивных функций при-

нято писать обертки, предоставляющие значения по умолчанию и в целом облегчающие использование функции. В данном случае нам, как правило, достаточно будет обертки с принудительным цитированием ввода (больше об этом вы узнаете в следующей главе), чтобы не было необходимости каждый раз вызывать функцию `expr()`:

```
logical_abbr <- function(x) {
    logical_abbr_rec(enexpr(x))
}

logical_abbr(T)
#> [1] TRUE
logical_abbr(FALSE)
#> [1] FALSE
```

Теперь нам необходимо реализовать рекурсивные случаи. Мы будем делать одно и то же для объектов вызова и для списков пар, а именно рекурсивно вызывать функцию для каждого подкомпонента и возвращать `TRUE`, если хотя бы в одном из них будет содержаться логическая аббревиатура. Для этого мы воспользуемся удобной функцией `purrr::some()`, которая проходит по списку и возвращает `TRUE`, в случае если предикативная функция вернет истину хотя бы для одного элемента списка.

```
logical_abbr_rec <- function(x) {
    switch_expr(x,
        # Базовые случаи
        constant = FALSE,
        symbol = as_string(x) %in% c("F", "T"),

        # Рекурсивные случаи
        call = ,
        pairlist = purrr::some(x, logical_abbr_rec)
    )
}

logical_abbr(mean(x, na.rm = T))
#> [1] TRUE
logical_abbr(function(x, na.rm = T) FALSE)
#> [1] TRUE
```

18.5.2 Находим все переменные, созданные в результате присваивания

Функция `logical_abbr()` получилась довольно простой: она просто возвращает `TRUE` или `FALSE`. Следующая наша цель будет немного сложнее – мы будем возвращать все переменные, созданные в результате присваивания. Начнем мы с простого, после чего постепенно будем увеличивать сложность решения.

Итак, взглянем на AST для операции присваивания:

```
ast(x <- 10)
#> ▉─`<-`
#> ├─x
#> └─10
```

Присваивание представляет собой объект вызова, в котором первым элементом является символ `<-`, вторым – имя переменной, а третьим – ее значение.

Нам необходимо решить, какую структуру данных использовать для представления результатов. Мне кажется, проще всего возвращать символьный вектор. Если мы будем возвращать символы, нам придется использовать функцию `list()`, что сделает решение более сложным.

Итак, мы можем начать с реализации базовых случаев и удобной обертки для нашей рекурсивной функции. В данном случае базовые случаи будут очень простыми, поскольку мы знаем, что ни символы, ни константы не могут представлять операцию присваивания.

```
find_assign_rec <- function(x) {
    switch_expr(x,
        constant = ,
        symbol = character()
    )
}

find_assign <- function(x) find_assign_rec(enexpr(x))
find_assign("x")
#> character(0)
find_assign(x)
#> character(0)
```

Теперь займемся реализацией рекурсивных случаев. Это будет легче сделать с помощью функции, которая должна присутствовать в пакете `purrr`, но пока ее нет. Функция `flat_map_chr()` ожидает, что `.f` возвращает вектор произвольной длины и превращает все результаты в один символьный вектор.

```
flat_map_chr <- function(.x, .f, ...) {
    purrr::flatten_chr(purrr::map(.x, .f, ...))
}

flat_map_chr(letters[1:3], ~ rep(., sample(3, 1)))
#> [1] "a" "b" "b" "b" "c" "c"
```

Рекурсивный случай для списка пар будет довольно простым: мы будем проходить по всем элементам списка (т. е. по каждому аргументу функции) и объединять результаты. Рекурсивный случай для объекта вызова будет чуть более сложным и будет учитывать, что для вызова функции `<-` необходимо вернуть второй элемент:

```
find_assign_rec <- function(x) {
    switch_expr(x,
        # Базовые случаи
        constant = ,
        symbol = character(),

        # Рекурсивные случаи
        pairlist = flat_map_chr(as.list(x), find_assign_rec),
        call = {
            if (is_call(x, "<-")) {
                as_string(x[[2]])
            } else {
                flat_map_chr(as.list(x), find_assign_rec)
            }
        }
    )
}

find_assign(a <- 1)
#> [1] "a"
find_assign({
    a <- 1
    {
        b <- 2
    }
})
#> [1] "a" "b"
```

Теперь нам осталось сделать решение более надежным, чтобы оно не ломалось. Что произойдет, если мы несколько раз присвоим значение одной переменной?

```
find_assign({
    a <- 1
    a <- 2
})
#> [1] "a" "a"
```

Эту проблему легче всего решить на уровне функции-обертки:

```
find_assign <- function(x) unique(find_assign_rec(enexpr(x)))

find_assign({
    a <- 1
    a <- 2
})
#> [1] "a"
```

А что с вложенностью операторов `<-`? В данный момент мы возвращаем лишь первую переменную. Это происходит из-за того, что мы прекращаем рекурсию при первой же встрече с оператором присваивания.

```
find_assign({
    a <- b <- c <- 1
})
#> [1] "a"
```

Вместо этого нам понадобится более надежный подход. Мне кажется, рекурсивная функция должна работать только со структурой дерева, а новый механизм мы выделим в отдельную функцию `find_assign_call()`.

```
find_assign_call <- function(x) {
    if (is_call(x, "<-") && is_symbol(x[[2]])) {
        lhs <- as_string(x[[2]])
        children <- as.list(x)[-1]
    } else {
        lhs <- character()
        children <- as.list(x)
    }

    c(lhs, flat_map_chr(children, find_assign_rec))
}

find_assign_rec <- function(x) {
    switch_expr(x,
        # Базовые случаи
        constant = ,
        symbol = character(),

        # Рекурсивные случаи
        pairlist = flat_map_chr(x, find_assign_rec),
        call = find_assign_call(x)
    )
}

find_assign(a <- b <- c <- 1)
#> [1] "a" "b" "c"
find_assign(system.time(x <- print(y <- 5)))
#> [1] "x" "y"
```

Полная версия нашей функции оказалась достаточно сложной, но очень важно, что мы собрали ее из более простых составляющих.

18.5.3 Упражнения

1. Функция `logical_abbr()` возвращает `TRUE` для выражения `T(1, 2, 3)`. Как бы вы изменили функцию `logical_abbr_rec()`, чтобы она игнорировала вызовы функций, использующие `T` или `F`?
2. Функция `logical_abbr()` работает с выражениями. В настоящий момент она не может обрабатывать функции. Почему? Как можно изменить функцию `logical_abbr()` таким образом, чтобы она смогла работать

с функциями? По каким компонентам функции вам нужно проходить рекурсивно?

```
logical_abbr(function(x = TRUE) {
    g(x + T)
})
```

3. Измените функцию `find_assign()` таким образом, чтобы она определяла операции присваивания, использующие замещающие функции, например `names(x) <- y`.
4. Напишите функцию, извлекающую все вызовы конкретной указанной функции.

18.6 Специализированные структуры данных

Есть еще две особые структуры данных и один специальный символ, о которых стоит упомянуть в этой главе. На практике мы с ними сталкиваемся не так часто.

18.6.1 Списки пар

Списки пар (pairlist) являются в языке R пережитком прошлого, и почти везде они были заменены списками. Единственное место, где вы по-прежнему можете встретить списки пар в R[1], – это вызовы функции `function`, поскольку формальные параметры функции хранятся в списке пар:

```
f <- expr(function(x, y = 10) x + y)

args <- f[[2]]
args
#> $x
#>
#>
#> $y
#> [1] 10
typeof(args)
#> [1] "pairlist"
```

К счастью, при работе со списками пар с ними можно обращаться как с обычными списками:

[1] Если вы работаете с языком C, то будете встречаться со списками пар гораздо чаще. К примеру, объекты вызова реализованы с использованием этой структуры данных.

```
pl <- pairlist(x = 1, y = 2)
length(pl)
#> [1] 2
pl$x
#> [1] 1
```

Внутренне списки пар реализованы с использованием другой структуры данных, а именно *связанных списков* (linked list), а не массивов. Это значительно замедляет процесс извлечения подмножеств из списков пар по сравнению со списками, но в целом не имеет большого практического значения.

18.6.2 Пропущенные аргументы

Особый символ, требующий отдельного разговора, представлен в R пустым символом, который используется для *пропущенных аргументов* (missing argument), – и это не то же самое, что пропущенные значения! Заботиться об обработке пропущенных символов следует тогда, когда вы программно создаете функции с пропущенными аргументами. Мы вернемся к этой теме в разделе 19.4.3.

Пустой символ можно создать при помощи функции `missing_arg()` (или `expr()`):

```
missing_arg()
typeof(missing_arg())
#> [1] "symbol"
```

Пустой символ не выводится на экран, а проверить его наличие можно с помощью функции `rlang::is_missing()`:

```
is_missing(missing_arg())
#> [1] TRUE
```

Примеры можно встретить при работе с формальными параметрами функций:

```
f <- expr(function(x, y = 10) x + y)
args <- f[[2]]
is_missing(args[[1]])
#> [1] TRUE
```

Это особенно важно для аргумента …, который всегда ассоциируется с пустым символом:

```
f <- expr(function(...) list(...))
args <- f[[2]]
is_missing(args[[1]])
#> [1] TRUE
```

У пустого символа есть одно необычное свойство: если связать его с переменной, а после этого обратиться к ней, вы получите ошибку:

```
m <- missing_arg()
m
#> Error in eval(expr, envir, enclos): argument "m" is missing, with no
#> default
```

В то же время ошибка не возникнет, если хранить пустой символ в другой структуре данных!

```
ms <- list(missing_arg(), missing_arg())
ms[[1]]
```

Если вы хотите защититься от пропущенных переменных, вам может пригодиться функция `rlang::maybe_missing()`. Зачастую она помогает, позволяя ссылаться на потенциально пропущенную переменную без возбуждения ошибки. За примерами и подробностями вы можете обратиться к документации функции.

18.6.3 Векторы выражений

Наконец, мы должны вкратце поговорить о *векторах выражений* (expression vector). Они могут создаваться в результате выполнения только двух функций: `expression()` и `parse()`:

```
exp1 <- parse(text = c("
x <- 4
x
"))
exp2 <- expression(x <- 4, x)

typeof(exp1)
#> [1] "expression"
typeof(exp2)
#> [1] "expression"

exp1
#> expression(x <- 4, x)
exp2
#> expression(x <- 4, x)
```

Подобно спискам пар и объектам вызова, векторы выражений ведут себя точно так же, как списки:

```
length(exp1)
#> [1] 2
```

```
exp1[[1]]
#> x <- 4
```

Чисто концептуально вектор выражений представляет собой просто список выражений. Единственным отличием является то, что при вызове функции `eval()` вычисляется каждое отдельное значение. Но я не считаю, что это небольшое преимущество стоит введения еще одной структуры данных, так что вместо векторов выражений я обычно пользуюсь простыми списками выражений.

19

Квазицитирование

19.1 Введение

Теперь, когда вы знакомы с древовидной структурой кода на языке R, пришло время вернуться к ключевой идее, лежащей в основе функций `expr()` и `ast()`, а именно к *цитированию* (quotation). В концепции *tidy evaluation* все цитирующие функции на самом деле являются квазицитирующими, поскольку они также поддерживают технику расцитирования. Если цитирование заключается в захвате невычисленных выражений, то *расцитирование* (unquotation) представляет собой возможность выборочного вычисления частей цитируемого выражения. В совокупности эта техника получила название *квазицитирование* (quasiquotation). Квазицитирование облегчает процесс создания функций, сочетающих код, написанный автором функции, с кодом, написанным пользователем. Описанная техника позволяет решать широкий спектр достаточно сложных задач.

Квазицитирование – один из трех столпов концепции *tidy evaluation*. С двумя другими (структурой данных *quosure* и масками данных) вы познакомитесь в главе 20. При обособленном использовании квазицитирование в основном применяется для генерирования кода. Но в сочетании с другими техниками *tidy evaluation* эта концепция превращается в мощный инструмент анализа данных.

Структура главы

- В разделе 19.2 описывается применение квазицитирования на примере функции `cement()`, работающей подобно функции `paste()`, но автоматически заключающей свои аргументы в кавычки, чтобы вам не приходилось делать это вручную.
- В разделе 19.3 будет представлен набор инструментов для цитирования выражений вне зависимости от их авторства, работающий как с пакетом `rlang`, так и в базовом R.
- В разделе 19.4 будет описано главное отличие между функциями цитирования из пакета `rlang` и аналогом из базового R, заключающееся в способе расцитирования с помощью оператора `!!` или `!!!`.

- Раздел 19.5 будет посвящен трем основным техникам, которые используются в функциях базового R для отмены цитирования.
- В разделе 19.6 мы увидим еще одно место применения оператора `!!!`, а именно функции, принимающие аргумент `...`. Также здесь мы поговорим о специальном операторе `:=`, позволяющем динамически менять имена аргументов.
- В разделе 19.7 будет показано несколько примеров практического использования техники цитирования для решения задач, по своей природе требующих генерирования кода.
- Раздел 19.8 завершит эту главу экскурсом в историю квазицитирования – для тех, кому это интересно.

Требования

Перед чтением этой главы я настоятельно рекомендую ознакомиться с содержимым главы 17, в которой описаны основные принципы, лежащие в основе метапрограммирования, а еще представлена базовая лексика. Также вам будет полезно изучить древовидную структуру выражений, описанную в разделе 18.3.

Что касается написания кода, мы по-прежнему будем активно использовать инструменты из пакета `rlang` (https://rlang.r-lib.org), а ближе к концу главы воспользуемся богатыми возможностями пакета `purrr` (https://purrr.tidyverse.org).

```
library(rlang)
library(purrr)
```

Дополнительные материалы

Цитирующие функции имеют много общего с макросами в языке Lisp. При этом макросы обычно запускаются в момент компиляции, которая в R отсутствует, и они всегда принимают на вход и возвращают AST. К тому же цитирующие функции тесно связаны с загадочными объектами *fexprs* в Lisp, представляющими собой функции, в которых все аргументы цитируются по умолчанию. Об этом полезно знать при поиске похожих решений в других языках программирования.

19.2 Предпосылки

Начнем с конкретного примера, который поможет понять пользу от расцитирования, а значит, и от квазицитирования. Представьте, что вам необходимо собрать длинную строку путем объединения множества отдельных слов:

```
paste("Good", "morning", "Hadley")
#> [1] "Good morning Hadley"
paste("Good", "afternoon", "Alice")
#> [1] "Good afternoon Alice"
```

Вы очень быстро устанете выписывать все эти кавычки, и вам захочется просто перечислять слова. Для этого вы можете написать следующую функцию (не задумывайтесь о ее реализации, об этом мы поговорим позже):

```
cement <- function(...) {
    args <- ensyms(...)
    paste(purrr::map(args, as_string), collapse = " ")
}

cement(Good, morning, Hadley)
#> [1] "Good morning Hadley"
cement(Good, afternoon, Alice)
#> [1] "Good afternoon Alice"
```

Чисто формально эта функция цитирует весь поступающий в нее вход. Вы можете думать, что в процессе ее работы каждый поступающий аргумент автоматически заключается в кавычки. На самом деле это не так, поскольку промежуточные объекты, генерируемые функцией, представляют собой выражения, а не строки, но для первого приближения такое объяснение вполне сгодится, и его можно считать описанием термина *цитирование*.

В целом получилась очень полезная функция, теперь нам нет необходимости указывать все эти кавычки вокруг слов. Проблемы возникнут, когда нам понадобится совместно со словами использовать переменные. У функции `paste()` никаких трудностей не возникнет: вы можете передавать ей на вход переменные без кавычек, и все будет работать.

```
name <- "Hadley"
time <- "morning"

paste("Good", time, name)
#> [1] "Good morning Hadley"
```

Очевидно, с функцией `cement()` такого сделать не получится, поскольку она принудительно оборачивает в кавычки весь ввод:

```
cement(Good, time, name)
#> [1] "Good time name"
```

Получается, нам необходимо иметь какой-то способ явного *расцитирования* входных параметров, чтобы функция `cement()` не обрамляла их кавычками. В данном случае нам нужно, чтобы функция по-разному обрабатывала входной аргумент `Good` и аргументы `time` и `name`. Концепция *квазицитирования* предоставляет нам стандартное средство для достижения этой цели в виде

оператора !!, символизирующего расцитирование. Этот оператор говорит цитирующей функции отменить обрамление кавычками указанного аргумента:

```
cement(Good, !!time, !!name)
#> [1] "Good morning Hadley"
```

Полезно будет сравнить функции cement() и paste() напрямую. Функция paste() вычисляет передаваемые ей аргументы, так что нам необходимо ставить кавычки там, где это необходимо. Напротив, функция cement() цитирует все свои аргументы, и нам нужно явным образом использовать расцитирование для аргументов, представляющих переменные.

```
paste("Good", time, name)
cement(Good, !!time, !!name)
```

19.2.1 Лексика

Важно понять разницу между цитируемыми и вычисляемыми аргументами:

- *вычисляемые аргументы* (evaluated argument) подчиняются обычным правилам вычислений, принятым в R;
- *цитируемые аргументы* (quoted argument) захватываются функцией и обрабатываются особым образом.

В приведенном выше примере функция paste() вычисляет все свои аргументы, а функция cement() цитирует их.

Если вы не уверены в том, цитируется или вычисляется аргумент, попробуйте выполнить код за пределами функции. Если он не работает или ведет себя как-то иначе, значит, аргумент цитируется. К примеру, вы можете воспользоваться этой техникой для определения того, цитируется ли первый аргумент функции library():

```
# работает
library(MASS)

# не работает
MASS
#> Error in eval(expr, envir, enclos): object 'MASS' not found
```

Рассуждение о том, является ли аргумент цитируемым или вычисляемым, представляет собой более точный способ выражения того, использует ли функция *нестандартное вычисление* (non-standard evaluation – NSE). Иногда я буду применять термин *цитирующая функция* в качестве сокращения для функции, цитирующей один или более аргументов, но в основном буду говорить о цитируемых аргументах, поскольку именно на этом уровне проявляются различия.

19.2.2 Упражнения

1. Для каждой функции в показанном ниже коде базового R определите, какие аргументы являются цитируемыми, а какие – вычисляемыми.

```
library(MASS)

mtcars2 <- subset(mtcars, cyl == 4)

with(mtcars2, sum(vs))
sum(mtcars2$am)

rm(mtcars2)
```

2. Для каждой функции в показанном ниже коде tidyverse определите, какие аргументы являются цитируемыми, а какие – вычисляемыми.

```
library(dplyr)
library(ggplot2)

by_cyl <- mtcars %>%
    group_by(cyl) %>%
    summarise(mean = mean(mpg))

ggplot(by_cyl, aes(cyl, mean)) + geom_point()
```

19.3 Цитирование

Первой частью квазицитирования является цитирование, т. е. захват выражения без его вычисления. Здесь нам понадобится пара функций, поскольку выражение может быть передано как напрямую, так и нет, посредством аргумента функции с отложенным вычислением. Начнем мы с функций цитирования из пакета `rlang`, после чего посмотрим на существующие аналоги в базовом R.

19.3.1 Захват выражений

Существует четыре основные функции цитирования. Для интерактивного применения наиболее важной является функция `expr()`, захватывающая переданный аргумент в его исходном виде:

```
expr(x + y)
#> x + y
expr(1 / 2 / 3)
#> 1/2/3
```

(Помните, что разделители и комментарии не являются частью выражения, в связи с чем они не захватываются цитирующей функцией.)

Функция `expr()` идеально подходит для интерактивного использования, поскольку она захватывает весь ваш, как разработчика, ввод. Внутри функции она не так полезна:

```
f1 <- function(x) expr(x)
f1(a + b + c)
#> x
```

Таким образом, нам нужна еще одна функция – `enexpr()`. Она захватывает выражение, переданное в функцию вызывающим блоком, путем инспектирования внутренней структуры данных, называемой *промис* (promise) и лежащей в основе ленивых, или отложенных, вычислений (см. раздел 6.5.1).

```
f2 <- function(x) enexpr(x)
f2(a + b + c)
#> a + b + c
```

(Приставка *en* является сокращением от *enrich* (расширенная): функция `enexpr()` принимает аргумент с отложенным вычислением и превращает его в выражение.)

Для захвата всех аргументов, переданных в …, используется функция `enexprs()`.

```
f <- function(...) enexprs(...)
f(x = 1, y = 10 * z)
#> $x
#> [1] 1
#>
#> $y
#> 10 * z
```

Наконец, функция `exprs()` может применяться для интерактивного создания списка выражений:

```
exprs(x = x ^ 2, y = y ^ 3, z = z ^ 4)
# сокращение для
# list(x = expr(x ^ 2), y = expr(y ^ 3), z = expr(z ^ 4))
```

Если коротко, функции `enexpr()` и `enexprs()` используются для захвата выражений, поставляемых *пользователем* в качестве аргументов, а функции `expr()` и `exprs()` – для захвата выражений, предоставленных *вами* как разработчиком.

19.3.2 Захват символов

Иногда вам необходимо позволить пользователю указать только имя переменной, а не произвольное выражение. В этом случае вы можете воспользо-

ваться функциями ensym() или ensyms(). Это разновидности функций enexpr() и enexprs(), проверяющие, является ли захваченное выражение символом или строкой (которая преобразуется в символ[1]). Функции ensym() и ensyms() выдают ошибку при передаче чего-то другого.

```
f <- function(...) ensyms(...)
f(x)
#> [[1]]
#> x
f("x")
#> [[1]]
#> x
```

19.3.3 Базовый R

У каждой функции, описанной выше, есть аналог в базовом R. Главное отличие между ними состоит в том, что базовые аналоги не поддерживают расцитирование, о чем мы поговорим совсем скоро. Это делает их цитирующими функциями, но не квазицитирующими.

Базовым аналогом для функции expr() является функция quote():

```
quote(x + y)
#> x + y
```

Функцией из базового R, наиболее подходящей по смыслу к функции enexpr(), можно назвать substitute():

```
f3 <- function(x) substitute(x)
f3(x + y)
#> x + y
```

Вместо функции exprs() можно воспользоваться функцией alist():

```
alist(x = 1, y = x + 2)
#> $x
#> [1] 1
#>
#> $y
#> x + 2
```

В качестве эквивалента функции enexprs() можно использовать недокументированную особенность функции substitute()[2]:

---

[1] Это сделано для совместимости с базовым R, который во многих случаях позволяет передавать строку вместо символа: "x" <- 1, "foo"(x, y), c("x" = 1).

[2] Эта особенность была открыта Питером Милструпом (Peter Meilstrup) и описана в R-devel 13 августа 2018 года.

```
f <- function(...) as.list(substitute(...()))
f(x = 1, y = 10 * z)
#> $x
#> [1] 1
#>
#> $y
#> 10 * z
```

Есть еще две важные базовые цитирующие функции, о которых подробно мы будем говорить позже:

- функция `bquote()` позволяет в ограниченном виде реализовать квазицитирование (о ней мы поговорим в разделе 19.5);
- оператор ~ (формула) представляет собой цитирующую функцию, которая также захватывает окружение. Эта тема тесно связана с использованием такой структуры данных, как *quosure*, о которой мы будем говорить в следующей главе.

19.3.4 Подстановка

Чаще всего в коде, захватывающем невычисляемые аргументы, вы будете встречать функцию `substitute()`. Помимо самого цитирования, эта функция также выполняет операцию подстановки, что ясно из ее имени. Если передать на вход функции `substitute()` выражение, а не символ, она выполнит подстановку значений в текущем окружении.

```
f4 <- function(x) substitute(x * 2)
f4(a + b + c)
#> (a + b + c) * 2
```

Мне кажется, это делает код более трудным для чтения, поскольку в отрыве от контекста вы не можете сказать, что является истинной целью `substitute(x + y)`, – замена `x`, `y` или того и другого.

Если вы хотите использовать функцию `substitute()` для выполнения подстановок, я рекомендую воспользоваться вторым аргументом, чтобы сделать свои намерения более явными:

```
substitute(x * y * z, list(x = 10, y = quote(a + b)))
#> 10 * (a + b) * z
```

19.3.5 Подведение итогов

При цитировании (т. е. захвате кода) важно ответить себе на два ключевых вопроса:

- будет код поставляться разработчиком или пользователем? Иначе говоря, будет этот код фиксированным (т. е. содержаться в теле функции) или изменяемым (передаваться с помощью аргументов)?
- вы собираетесь захватывать одно или несколько выражений?

На основе ответов на эти вопросы можно составить две простые таблицы размером 2×2: одну для пакета `rlang` (табл. 19.1), а вторую – для базового R (табл. 19.2).

Таблица 19.1 Варианты цитирования в пакете `rlang`

	Разработчик	**Пользователь**
Одно выражение	`expr()`	`enexpr()`
Несколько выражений	`exprs()`	`enexprs()`

Таблица 19.2 Варианты цитирования в базовом R

	Разработчик	**Пользователь**
Одно выражение	`quote()`	`substitute()`
Несколько выражений	`alist()`	`as.list(substitute(...()))`

19.3.6 Упражнения

1. Какова реализация функции `expr()`? Почитайте исходный код функции.
2. Сравните и противопоставьте следующие две функции. Можете ли вы предсказать их вывод до запуска?

```
f1 <- function(x, y) {
exprs(x = x, y = y)
}
f2 <- function(x, y) {
enexprs(x = x, y = y)
}
f1(a + b, c + d)
f2(a + b, c + d)
```

3. Что произойдет, если попытаться использовать функцию `enexpr()` применительно к выражению (`enexpr(x + y)`)? Что случится, если функции `enexpr()` передать пропущенный аргумент?
4. Чем отличаются вызовы `exprs(a)` и `exprs(a = )`? Подумайте о входах функций и о выводе.
5. Какие еще есть отличия между функциями `exprs()` и `alist()`? Ознакомьтесь с разделом документации по функции `exprs()`, посвященным именованным аргументам.
6. В документации к функции `substitute()` говорится: подстановка выполняется путем рассмотрения каждого компонента дерева следующим образом:
   – если это несвязанный символ в `env`, он остается неизменным;

- если это объект промис (т. е. формальный аргумент функции), символ заменяется слотом `expression` этого объекта;
- если это обычная переменная, заменяется ее значение, если `env` не является `.GlobalEnv` (в этом случае символ остается неизменным).

Создайте пример, иллюстрирующий все из представленных выше случаев.

19.4 Расцитирование

До сих пор мы говорили только о незначительных преимуществах функций, связанных с цитированием, из пакета `rlang` над встроенными аналогами из базового пакета R. Фактически все сводилось к более последовательной схеме именования функций. Основным же отличием этих функций является то, что они являются не только цитирующими, но и квазицитирующими, поскольку поддерживают механизм расцитирования.

Расцитирование (unquoting) позволяет выборочно вычислять отдельные части выражения, которые в противном случае были бы цитированы, что фактически делает возможным объединение AST с помощью шаблона. Поскольку базовые функции не поддерживают механизм расцитирования, им для этого приходится использовать другие техники, о которых мы поговорим в разделе 19.5.

Расцитирование представляет собой процесс, обратный цитированию. С его помощью вы можете выборочно вычислять код внутри функции `expr()`, так что запись `expr(!!x)` эквивалентна `x`. В главе 20 вы познакомитесь с еще одним процессом, обратным цитированию, а именно с вычислением. Эта операция выполняется за пределами `expr()`, а запись `eval(expr(x))` также будет эквивалентна `x`.

19.4.1 Расцитирование одного аргумента

Для расцитирования одного аргумента в вызове функции воспользуйтесь оператором `!!`. Этот оператор принимает выражение, вычисляет его и внедряет результат в AST.

```
x <- expr(-1)
expr(f(!!x, y))
#> f(-1, y)
```

Мне кажется, проще всего это понять при помощи диаграммы. Оператор `!!` представлен на рис. 19.1 заменителем в AST в виде блока с пунктирными границами. В этот заменитель на втором шаге встраивается целое дерево `x`, показанное на диаграмме справа. Этот процесс проиллюстрирован пунктирной стрелкой.

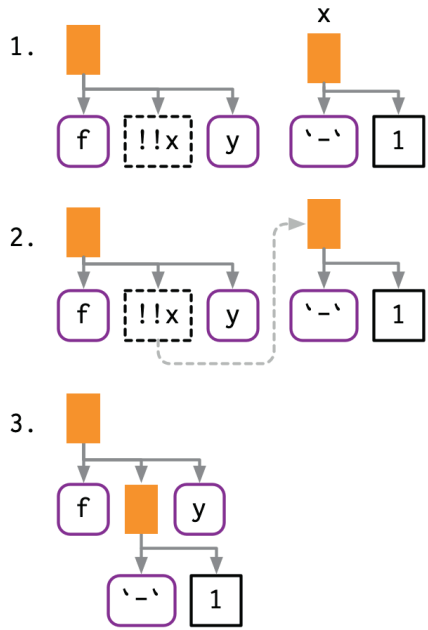


Рис. 19.1 Расцитирование одного аргумента (выражения)

Подобно объектам вызова, оператор `!!` умеет работать с символами и константами, что показано на рис. 19.2:

```
a <- sym("y")
b <- 1
expr(f(!!a, !!b))
#> f(y, 1)
```

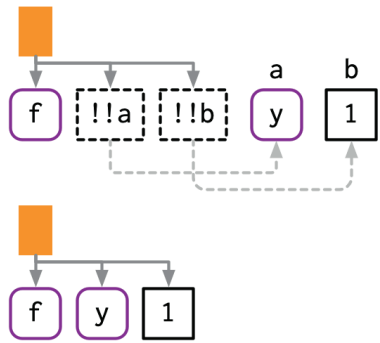


Рис. 19.2 Расцитирование символов и констант

Если справа от оператора `!!` находится вызов функции, эта функция будет выполнена, а результат встроится в AST:

```
mean_rm <- function(var) {
    var <- ensym(var)
    expr(mean(!!var, na.rm = TRUE))
}
expr(!!mean_rm(x) + !!mean_rm(y))
#> mean(x, na.rm = TRUE) + mean(y, na.rm = TRUE)
```

Оператор `!!` следует правилам приоритета, поскольку работает с выражениями, что видно на рис. 19.3.

```
x1 <- expr(x + 1)
x2 <- expr(x + 2)

expr(!!x1 / !!x2)
#> (x + 1)/(x + 2)
```

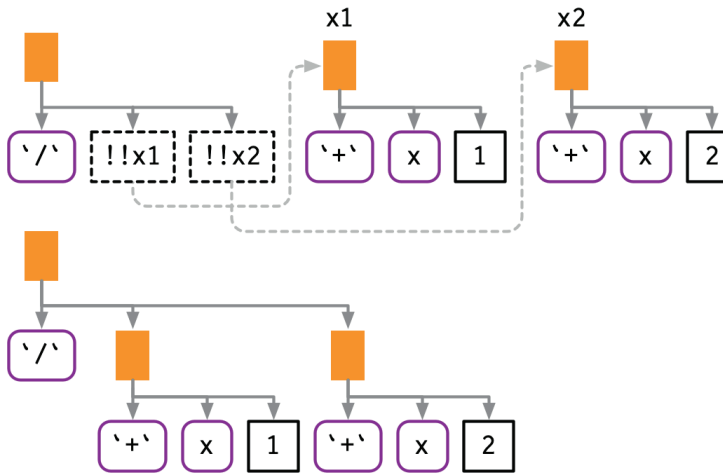


Рис. 19.3 Расцитирование с сохранением приоритетов

Если бы вместо этого мы просто объединили текст выражений вместе, то получили бы некорректное выражение `x + 1 / x + 2` с совсем другим AST, показанным на рис. 19.4.

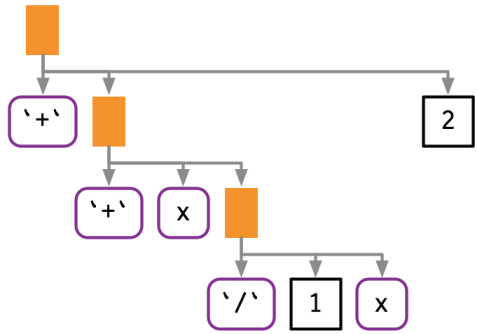


Рис. 19.4 Некорректное дерево выражения

19.4.2 Расцитирование функции

Обычно оператор `!!` используется для замены аргументов функции, но его можно применять и к самой функции. Единственной сложностью здесь является соблюдение правил приоритета: вызов `expr(!!f(x, y))` приведет к расцитированию результата `f(x, y)`, так что нам понадобится дополнительная пара скобок.

```
f <- expr(foo)
expr((!!f)(x, y))
#> foo(x, y)
```

Этот пример также будет работать, что видно на рис. 19.5:

```
f <- expr(pkg::foo)
expr((!!f)(x, y))
#> pkg::foo(x, y)
```

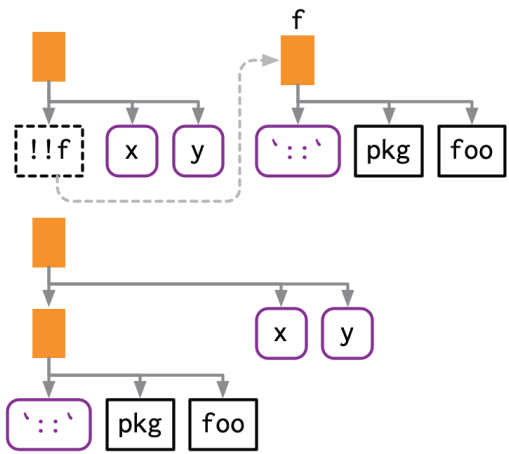


Рис. 19.5 Расцитирование функции из пакета

Во избежание появления большого количества скобок можно упростить запись при помощи функции `lang::call2()` следующим образом:

```
f <- expr(pkg::foo)
call2(f, expr(x), expr(y))
#> pkg::foo(x, y)
```

19.4.3 Расцитирование пропущенного аргумента

Изредка возникает необходимость в расцитировании пропущенного аргумента (см. раздел 18.6.2), но привычный способ здесь не подойдет:

```
arg <- missing_arg()
expr(foo(!!arg, !!arg))
#> Error in enexpr(expr): argument "arg" is missing, with no default
```

Эту проблему можно обойти с помощью специальной функции `rlang::maybe_missing()`:

```
expr(foo(!!maybe_missing(arg), !!maybe_missing(arg)))
#> foo(, )
```

19.4.4 Расцитирование в особых случаях

Существует несколько особых случаев, в которых операция расцитирования будет воспринята как синтаксическая ошибка. Возьмем для примера оператор `$`: после него обязательно должно следовать имя переменной, а не другое выражение. Это означает, что попытка расцитирования с `$` приведет к ошибке:

```
expr(df$!!x)
#> Error: unexpected '!' in "expr(df$!"
```

Чтобы расцитирование работало, необходимо воспользоваться префиксной формой функции `$` (см. раздел 6.8.1), как показано ниже:

```
x <- expr(x)
expr(`$`(df, !!x))
#> df$x
```

19.4.5 Расцитирование нескольких аргументов

Оператор `!!` работает по принципу подстановки один к одному. Для подстановки один ко многим существует оператор `!!!`. Он принимает список выражений и вставляет их на место оператора, как показано на рис. 19.6:

```
xs <- exprs(1, a, -b)
expr(f(!!!xs, y))
```

```
#> f(1, a, -b, y)

# Или с именами
ys <- set_names(xs, c("a", "b", "c"))
expr(f(!!!ys, d = 4))
#> f(a = 1, b = a, c = -b, d = 4)
```

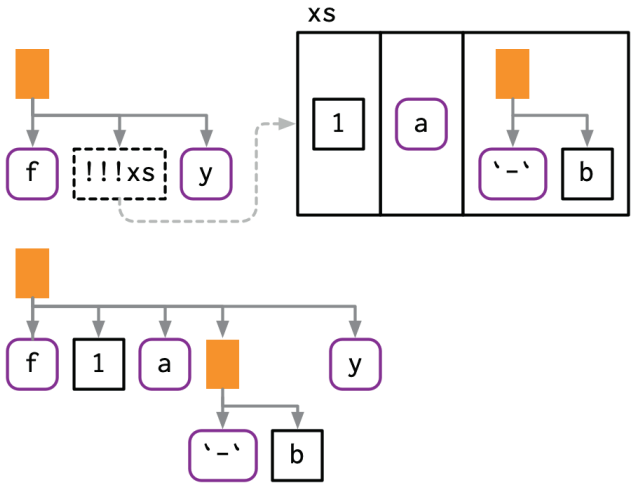


Рис. 19.6 Расщитирование нескольких аргументов

Оператор `!!!` может быть использован с любой функцией пакета `rlang`, принимающей аргумент ... вне зависимости от того, является этот аргумент цитируемым или вычисляемым. Мы вернемся к этому в разделе 19.6, а пока запомните, что этот оператор может вам пригодиться в функции `call2()`.

```
call2("f", !!!xs, expr(y))
#> f(1, a, -b, y)
```

19.4.6 Учтивое притворство оператора !!

До сих пор мы действовали так, будто `!!` и `!!!` являются обычными префиксными операторами вроде `+`, `-` и `!`. Но это не так. С точки зрения R `!!` и `!!!` – это не что иное, как повторение `!`:

```
!!TRUE
#> [1] TRUE
!!!TRUE
#> [1] FALSE
```

Внутри цитирующих функций из пакета rlang операторы `!!` и `!!!` ведут себя особым образом – как настоящие операторы с приоритетом, равным унарным операторам `+` и `-`. Это потребовало дополнительных усилий от раз-

работчиков пакета `rlang`, но зато теперь мы можем писать `!!x + !!y` вместо `(!!x) + (!!y)`.

Серьезный недостаток[1] использования такого притворного оператора состоит в том, что вы можете столкнуться с трудновыявляемыми ошибками, если неправильно примените его за пределами квазицитирующей функции. В большинстве случаев с этим не будет проблем, поскольку оператор `!!` обычно используется для расцитирования выражений или структур данных *quosure*. В таких ситуациях вы будете получать ошибку, говорящую о некорректном типе аргумента, из-за того, что выражения не поддерживаются оператором отрицания:

```
x <- quote(variable)
!!x
#> Error in !x: invalid argument type
```

В то же время вы можете получить ошибку в расчетах при работе с числовыми значениями:

```
df <- data.frame(x = 1:5)
y <- 100
with(df, x + !!y)
#> [1] 2 3 4 5 6
```

С учетом этих недостатков вы могли бы спросить, а зачем мы представили новый синтаксис вместо использования обычных функций. И действительно, в ранних версиях *tidy evaluation* использовались функции вроде `UQ()` и `UQS()`. Но это не настоящие функции, а делать вид, что это так, – значит неправильно воспринимать действительность. Мы остановились на синтаксисе `!!` и `!!!` как на меньшем из зол:

- это визуально заметный синтаксис, который трудно спутать с существующими языковыми конструкциями. Когда вы видите оператор `!!` или `!!!`, вы понимаете, что здесь происходит что-то не совсем обычное;
- эти операторы переопределяют редко используемый синтаксис, поскольку двойное отрицание вряд ли можно назвать популярной операцией в R[2]. Если же вам необходимо ей воспользоваться, вы всегда можете применить синтаксис `!(!x)`.

---

[1] Вплоть до версии R 3.5.1 это было сопряжено с еще одним недостатком: депарсер R воспринимал выражение `!!x` как `!(!x)`. Именно поэтому в старых версиях R вы можете встретить множество дополнительных скобок при выводе выражений. Хорошая новость состоит в том, что эти скобки не настоящие и в большинстве случаев могут быть безопасно проигнорированы. Плохо то, что они превратятся в настоящие при повторном парсинге вывода в код R. Такие циклические функции не будут работать как ожидается, поскольку выражение `!(!x)` не расцитируется.

[2] В отличие, например, от Javascript, где синтаксис `!!x` является распространенным шаблоном для преобразования числовых значений в логические.

19.4.7 Нестандартные AST

При использовании операций расцитирования легко могут создаваться нестандартные AST, т. е. содержащие компоненты, не являющиеся выражениями. (Нестандартные AST также можно создать путем прямого манипулирования объектами в выражении, но это трудно сделать случайно.) Созданные в результате деревья будут допустимыми, а иногда и полезными, но их эффективное использование выходит за рамки данной книги. При этом очень важно знать об их существовании, поскольку они могут сбивать с толку и выводиться неправильно.

К примеру, если встроить в выражение более сложные объекты, их атрибуты не будут выводиться. Это может вас удивить:

```
x1 <- expr(class(!!data.frame(x = 10)))
x1
#> class(list(x = 10))
eval(x1)
#> [1] "data.frame"
```

Существуют два инструмента для снижения градуса замешательства: `rlang::expr_print()` и `lobstr::ast()`:

```
expr_print(x1)
#> class(<data.frame>)
lobstr::ast(!!x1)
#> ■─class
#> └─<inline data.frame>
```

Также вас может ждать сюрприз при встраивании в выражение последовательности целых чисел:

```
x2 <- expr(f(!!c(1L, 2L, 3L, 4L, 5L)))
x2
#> f(1:5)
expr_print(x2)
#> f(<int: 1L, 2L, 3L, 4L, 5L>)
lobstr::ast(!!x2)
#> ■─f
#> └─<inline integer>
```

Кроме того, можно создавать AST, которые не могут быть сгенерированы из кода из-за правил приоритета операторов. В этом случае R покажет скобки, отсутствующие в AST:

```
x3 <- expr(1 + !!expr(2 + 3))
x3
#> 1 + (2 + 3)

lobstr::ast(!!x3)
```

```
#> ▉─`+`
#> ├─1
#> └─▉─`+`
#> ├─2
#> └─3
```

19.4.8 Упражнения

1. Вам даны следующие компоненты:

   ```
   xy <- expr(x + y)
   xz <- expr(x + z)
   yz <- expr(y + z)
   abc <- exprs(a, b, c)
   ```

   Воспользуйтесь квазицитированием для конструирования следующих вызовов:

   ```
   (x + y) / (y + z)
   -(x + z) ^ (y + z)
   (x + y) + (y + z) - (x + y)
   atan2(x + y, y + z)
   sum(x + y, x + y, y + z)
   sum(a, b, c)
   mean(c(a, b, c), na.rm = TRUE)
   foo(a = x + y, b = y + z)
   ```

2. Следующие два вызова выводят один и тот же результат, но сами результаты при этом не идентичны:

   ```
   (a <- expr(mean(1:10)))
   #> mean(1:10)
   (b <- expr(mean(!!(1:10))))
   #> mean(1:10)
   identical(a, b)
   #> [1] FALSE
   ```

   В чем их отличие? Какой вариант является более естественным?

19.5 Техники отмены цитирования

В базовом R есть одна функция, реализующая квазицитирование, – это `bquote()`. Для расцитирования в ней используется синтаксис `.()`:

```
xyz <- bquote((x + y + z))
bquote(-.(xyz) / 2)
#> -(x + y + z)/2
```

Функция `bquote()` не применяется в других функциях базового R и имеет относительно небольшое влияние на написание кода в R. С эффективным использованием функции `bquote()` есть несколько сложностей:

- эту функцию легко использовать только с вашим собственным кодом. Ее трудно применить к произвольному коду, предоставленному пользователем;
- функция `bquote()` не предлагает аналогов оператору расцитирования нескольких выражений, сохраненных в виде списка;
- функция `bquote()` не предоставляет возможностей для обработки кода в совокупности с окружением, что критически важно для функций, вычисляющих код в контексте датафреймов вроде `subset()` и других.

В базовых функциях R, позволяющих цитировать свои аргументы, используются другие техники, допускающие косвенную спецификацию. Эти подходы предпочитают выборочное отключение цитирования операции расцитирования, в связи с чем я и называю их *техниками отмены цитирования* (non-quoting techniques).

Существуют четыре основные формы отмены цитирования в базовом R:

- использование цитирующих и нецитирующих функций. Например, функция `$` принимает два аргумента, и второй из них цитируется. Это легко увидеть, если переписать вызов функции с применением префиксной формы: `` `$`(mtcars, cyl) `` вместо `mtcars$cyl`. Если вам нужно обратиться к переменной косвенно, воспользуйтесь оператором `[[`, принимающим имя переменной в виде строки.

  ```
  x <- list(var = 1, y = 2)
  var <- "y"

  x$var
  #> [1] 1
  x[[var]]
  #> [1] 2
  ```

  Есть три другие цитирующие функции, тесно связанные с `$`: `subset()`, `transform()` и `with()`. Их можно рассматривать как обертки вокруг `$`, подходящие только для интерактивного использования, и у всех у них есть одна и та же альтернатива без цитирования: `[`.
  `<-/assign()` и `::/getExportedValue()` работают аналогично `$/[`;

- использование цитирующих и нецитирующих аргументов. К примеру, функция `rm()` может принимать или чистые имена переменных в аргументе `...`, или символьный вектор из имен переменных в аргументе `list`:

  ```
  x <- 1
  rm(x)

  y <- 2
  ```

```
vars <- c("y", "vars")
rm(list = vars)
```

Функции `data()` и `save()` работают аналогично;

- использование аргумента, контролирующего цитируемость другого аргумента. К примеру, в функции `library()` аргумент `character.only` управляет поведением первого аргумента, `package`, в отношении цитирования:

```
library(MASS)

pkg <- "MASS"
library(pkg, character.only = TRUE)
```

Функции `demo()`, `detach()`, `example()` и `require()` работают аналогично;

- цитирование при невозможности вычисления. Например, в функции `help()` первый аргумент не цитируется, если он может быть вычислен в строку. В случае неудачи будет применено цитирование.

```
# Справка по var
help(var)

var <- "mean"
# Справка по mean
help(var)

var <- 10
# Справка по var
help(var)
```

Функции `ls()`, `page()` и `match.fun()` работают аналогично.

Также очень важными классами цитирующих функций являются функции моделирования данных и вывода графиков, придерживающиеся так называемого «свода правил стандартного нестандартного вычисления» (https://developer.r-project.org/nonstandard-eval.pdf). К примеру, в функции `lm()` цитируются аргументы `weight` и `subset`, а при использовании с аргументом `formula` функция `plot()` цитирует аргументы эстетик (`col`, `cex` и т. д.). Взгляните на следующий код: нам достаточно указать `col = Species` вместо `col = iris$Species`. Результат показан на рис. 19.7.

```
palette(RColorBrewer::brewer.pal(3, "Set1"))
plot(
    Sepal.Length ~ Petal.Length,
    data = iris,
    col = Species,
    pch = 20,
    cex = 2
)
```

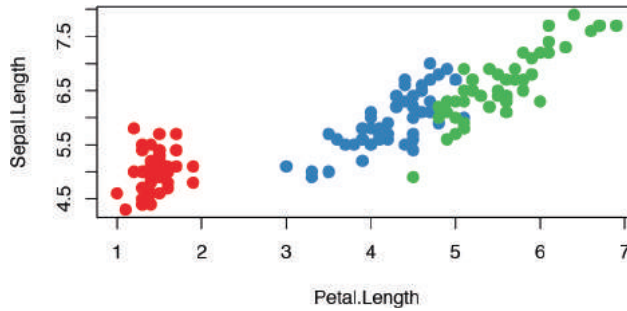


Рис. 19.7 Вывод графика функции `plot()`

У этих функций нет встроенных опций для косвенной спецификации, но в разделе 20.6 мы рассмотрим способ имитации расцитирования.

19.6 ... (точка–точка–точка)

Оператор `!!!` бывает достаточно полезен, поскольку нам часто бывает нужно вставить в вызов список выражений. Такой шаблон требуется повсеместно. Здесь есть две сопутствующие проблемы:

- что делать, если элементы, которые вам нужно поместить в аргумент ..., уже сохранены в списке? Представьте, к примеру, что у вас есть список датафреймов, которые вам необходимо объединить при помощи функции `rbind()`:

```
dfs <- list(
    a = data.frame(x = 1, y = 2),
    b = data.frame(x = 3, y = 4)
)
```

  В этом конкретном случае можно было бы применить запись `rbind(dfs$a, dfs$b)`, но как обобщить это решение для списка произвольной длины?

- что делать, если вам необходимо передать имя аргумента косвенно? Представьте, что вам нужно создать датафрейм с одной колонкой, имя которой должно задаваться с помощью переменной:

```
var <- "x"
val <- c(4, 3, 9)
```

  В этом случае вы могли бы создать датафрейм, а затем изменить имена – `setNames(data.frame(val), var)`), но это не самое элегантное решение. Как сделать это более красиво?

Размышляя о решении подобных проблем, можно провести явные параллели с квазицитированием:

- построчное объединение нескольких датафреймов похоже на применение оператора расцитирования нескольких аргументов (`!!!`): нам нужно встроить отдельные элементы списка в вызов:

```
dplyr::bind_rows(!!!dfs)
#>   x y
#> 1 1 2
#> 2 3 4
```

  В данном контексте поведение оператора `!!!` аналогично тому, что в языках Ruby, Go, PHP и Julia называется словом *spatting*. В языке Python ближайшим аналогом являются аргументы `*args` и `**kwarg`, служащие для распаковки переданных параметров;

- вторая проблема завязана на расцитировании левой части оператора `=`: вместо буквальной интерпретации `var` нам необходимо использовать значение, сохраненное в переменной `var`:

```
tibble::tibble(!!var := val)
#> # A tibble: 3 x 1
#>       x
#>   <dbl>
#> 1     4
#> 2     3
#> 3     9
```

  Обратите внимание на использование оператора `:=` (читается как *двоеточие–равно*) вместо `=`. К сожалению, нам необходимо применить эту новую операцию, поскольку грамматика R не позволяет использовать выражения в качестве имен аргументов:

```
tibble::tibble(!!var = value)
#> Error: unexpected '=' in "tibble::tibble(!!var ="
```

  Оператор `:=` подобен рудиментарному органу: он распознается парсером R, но не имеет связанного с ним кода. Ведет себя этот оператор так же, как `=`, но допускает присутствие выражений с обеих сторон от оператора, что делает его более гибким по сравнению с `=`. По этим же причинам этот оператор используется совместно с `data.table`.

В базовом R применяется другой подход, о котором мы поговорим в разделе 19.6.4.

Мы говорим, что функции, поддерживающие эти инструменты без цитирования аргументов, обладают *точками tidy* (tidy dots)[1]. Чтобы позволить своим функциям реализовывать такое поведение, вам необходимо воспользоваться функцией `list2()`.

---

[1] Это не самое креативное название, но по крайней мере оно дает понять, что данный функционал был добавлен в R постфактум.

19.6.1 Пример

Одним из примеров использования функции `list2()` является создание обертки функции `attributes()`, позволяющей гибко устанавливать атрибуты:

```
set_attr <- function(.x, ...) {
    attr <- rlang::list2(...)
    attributes(.x) <- attr
    .x
}

attrs <- list(x = 1, y = 2)
attr_name <- "z"

1:10 %>%
    set_attr(w = 0, !!!attrs, !!attr_name := 3) %>%
    str()
#>  int [1:10] 1 2 3 4 5 6 7 8 9 10
#>  - attr(*, "w")= num 0
#>  - attr(*, "x")= num 1
#>  - attr(*, "y")= num 2
#>  - attr(*, "z")= num 3
```

19.6.2 exec()

А что, если вам необходимо воспользоваться этой техникой с функцией без *tidy dots*? Один из вариантов – использовать `rlang::exec()` для вызова функции с передачей одних аргументов напрямую (через …), а остальных – косвенно (через список):

```
# Напрямую
exec("mean", x = 1:10, na.rm = TRUE, trim = 0.1)
#> [1] 5.5

# Косвенно
args <- list(x = 1:10, na.rm = TRUE, trim = 0.1)
exec("mean", !!!args)
#> [1] 5.5

# Смешанный подход
params <- list(na.rm = TRUE, trim = 0.1)
exec("mean", x = 1:10, !!!params)
#> [1] 5.5
```

Функция `rlang::exec()` также позволяет косвенно передавать имена аргументов:

```
arg_name <- "na.rm"
arg_val <- TRUE
```

```
exec("mean", 1:10, !!arg_name := arg_val)
#> [1] 5.5
```

И наконец, иногда полезно иметь вектор имен функций или список функций, которые необходимо вызвать с одинаковыми аргументами:

```
x <- c(runif(10), NA)
funs <- c("mean", "median", "sd")

purrr::map_dbl(funs, exec, x, na.rm = TRUE)
#> [1] 0.444 0.482 0.298
```

Функция `exec()` тесно связана с `call2()`; если функция `call2()` возвращает выражение, то функция `exec()` вычисляет его.

19.6.3 dots_list()

У функции `list2()` есть одна интересная особенность: по умолчанию она будет игнорировать любые пустые аргументы в конце. Это может быть удобно в функциях вроде `tibble::tibble()`, поскольку позволяет менять порядок переменных без необходимости заботиться о заключительной запятой:

```
# Можно легко перекинуть x в начало
tibble::tibble(
    y = 1:5,
    z = 3:-1,
    x = 5:1,
)

# Нужно удалить запятую у z и добавить к x
data.frame(
    y = 1:5,
    z = 3:-1,
    x = 5:1
)
```

Функция `list2()` является оберткой функции `rlang::dots_list()` с наиболее распространенными настройками. Больший контроль над происходящим можно получить путем вызова функции `dots_list()` напрямую:

- аргумент `.ignore_empty` позволяет указать, какие именно аргументы необходимо игнорировать. По умолчанию игнорируется один завершающий аргумент, что позволяет получить описанное выше поведение, но вы можете выбрать игнорирование всех пропущенных аргументов или ни одного;
- аргумент `.homonoyms` управляет тем, что происходит при использовании нескольких аргументов одного имени:

```
str(dots_list(x = 1, x = 2))
#> List of 2
#>  $ x: num 1
#>  $ x: num 2
str(dots_list(x = 1, x = 2, .homonyms = "first"))
#> List of 1
#>  $ x: num 1
str(dots_list(x = 1, x = 2, .homonyms = "last"))
#> List of 1
#>  $ x: num 2
str(dots_list(x = 1, x = 2, .homonyms = "error"))
#> Error: Arguments can't have the same name.
#> We found multiple arguments named `x` at positions 1 and 2
```

- если есть пустые аргументы, которые не игнорируются, с помощью аргумента `.preserve_empty` можно задать для них поведение. По умолчанию настроен выброс ошибки. Установка значения `.preserve_empty = TRUE` позволит возвращать пропущенные символы. Это бывает полезно при использовании функции `dots_list()` для генерирования вызовов функций.

19.6.4 В базовом R

В базовом R есть универсальное средство для решения подобных проблем – функция `do.call()`. Эта функция принимает два главных аргумента. Первым, `what`, является функция для вызова. В качестве второго аргумента, `args`, передается список аргументов для передачи в эту функцию, так что запись `do.call("f", list(x, y, z))` эквивалентна `f(x, y, z)`.

- Функция `do.call()` предоставляет простое решение для объединения нескольких датафреймов с помощью функции `rbind()`:

```
do.call("rbind", dfs)
#>   x y
#> a 1 2
#> b 3 4
```

- с небольшими дополнительными усилиями мы можем использовать функцию `do.call()` для решения второй проблемы. Сначала создадим список аргументов, затем заполним их имена, после чего воспользуемся функцией `do.call()`:

```
args <- list(val)
names(args) <- var

do.call("data.frame", args)
#>   x
#> 1 4
```

```
#> 2 3
#> 3 9
```

Некоторые базовые функции, такие как `interaction()`, `expand.grid()`, `options()` и `par()`, используют специальный трюк, чтобы не вызывать функцию `do.call()`: если первый элемент аргумента `...` – это список, они извлекают его компоненты, а не обращаются к другим элементам аргумента `...`. Реализация выглядит как-то так:

```
f <- function(...) {
    dots <- list(...)
    if (length(dots) == 1 && is.list(dots[[1]])) {
        dots <- dots[[1]]
    }

    # Что-то делаем
    ...
}
```

Еще один способ не вызывать функцию `do.call()` реализован в функции `RCurl::getURL()`, написанной Дунканом Темплом Лангом (Duncan Temple Lang). Функция `getURL()` берет оба аргумента, `...` и `.dots`, и конкатенирует их вместе, что выглядит примерно так:

```
f <- function(..., .dots) {
    dots <- c(list(...), .dots)
    # Что-то делаем
}
```

Когда я узнал об этой технике, она мне показалась восхитительной, поэтому я использовал ее повсеместно в tidyverse. Сейчас, однако, я предпочитаю подход, упомянутый ранее.

19.6.5 Упражнения

1. Один из способов реализации функции `exec()` показан ниже. Опишите работу этой функции. Какие ключевые идеи в ней используются?

   ```
   exec <- function(f, ..., .env = caller_env()) {
       args <- list2(...)
       do.call(f, args, envir = .env)
   }
   ```

2. Внимательно прочитайте исходный код функций `interaction()`, `expand.grid()` и `par()`. Сравните и противопоставьте техники, которые в них используются для переключения между точками и списками.
3. В чем состоит проблема такого определения функции `set_attr()`?

```
set_attr <- function(x, ...) {
    attr <- rlang::list2(...)
    attributes(x) <- attr
    x
}
set_attr(1:10, x = 10)
#> Error in attributes(x) <- attr: attributes must be named
```

19.7 Практические примеры

Для закрепления понимания идей квазицитирования в этом разделе мы рассмотрим несколько практических примеров решения конкретных задач. В некоторых из них будет использоваться пакет `purrr`: мне лично комбинация с применением квазицитирования совместно с функциональным программированием кажется крайне изящной.

19.7.1 lobstr::ast()

Квазицитирование позволяет решить досадную проблему с `lobstr::ast()`: что делать, если мы уже захватили выражение?

```
z <- expr(foo(x, y))
lobstr::ast(z)
#> z
```

Поскольку функция `ast()` цитирует свой первый аргумент, мы можем воспользоваться оператором `!!`:

```
lobstr::ast(!!z)
#> ▉─foo
#> ├─x
#> └─y
```

19.7.2 Map-reduce для генерации кода

Квазицитирование является мощным инструментом генерирования кода, особенно в сочетании с функциями `purrr::map()` и `purr::reduce()`. Представьте, например, что у вас есть линейная модель со следующими коэффициентами:

```
intercept <- 10
coefs <- c(x1 = 5, x2 = -4)
```

Вам необходимо преобразовать ее в формулу `10 + (x1 * 5) + (x2 * -4)`. Первое, что нам нужно сделать, – это преобразовать имена коэффициентов в список символов. В этом нам поможет функция `rlang::syms()`:

```
coef_sym <- syms(names(coefs))
coef_sym
#> [[1]]
#> x1
#>
#> [[2]]
#> x2
```

Теперь нам необходимо объединить имена переменных с коэффициентами. Это можно сделать с использованием функций `rlang::expr()` и `purrr::map2()`:

```
summands <- map2(coef_sym, coefs, ~ expr((!!.x * !!.y)))
summands
#> [[1]]
#> (x1 * 5)
#>
#> [[2]]
#> (x2 * -4)
```

В данном случае свободный член также является частью суммы, хотя и не вовлечен в операции умножения. Мы можем просто добавить его в начало вектора `summands`:

```
summands <- c(intercept, summands)
summands
#> [[1]]
#> [1] 10
#>
#> [[2]]
#> (x1 * 5)
#>
#> [[3]]
#> (x2 * -4)
```

Осталось применить операцию `reduce` (см. раздел 9.5) для объединения выражения:

```
eq <- reduce(summands, ~ expr(!!.x + !!.y))
eq
#> 10 + (x1 * 5) + (x2 * -4)
```

Можно еще больше обобщить процесс, позволив пользователю передавать имя коэффициента, как показано ниже:

```
var <- expr(y)
coef_sym <- map(seq_along(coefs), ~ expr((!!var)[[!!.x]]))
coef_sym
#> [[1]]
#> y[[1L]]
```

```
#>
#> [[2]]
#> y[[2L]]
```

Теперь объединим это все в функцию:

```
linear <- function(var, val) {
    var <- ensym(var)
    coef_name <- map(seq_along(val[-1]), ~ expr((!!var)[[!!.x]]))

    summands <- map2(val[-1], coef_name, ~ expr((!!.x * !!.y)))
    summands <- c(val[[1]], summands)

    reduce(summands, ~ expr(!!.x + !!.y))
}

linear(x, c(10, 5, -4))
#> 10 + (5 * x[[1L]]) + (-4 * x[[2L]])
```

Обратите внимание на использование функции `ensym()`: мы хотим, чтобы пользователь передавал имя одной переменной, а не сложное выражение.

19.7.3 Срезы массива

В базовом R не хватает полезной функции, позволяющей извлекать срез массива на основании номера измерения и индекса. К примеру, нам бы хотелось использовать вызов `slice(x, 2, 1)` для получения первого среза по второму измерению, т. е. `x[, 1, ]`. Это довольно непростая задача, поскольку она требует обработки пропущенных аргументов.

Нам понадобится сгенерировать вызов с несколькими пропущенными аргументами. Для начала создадим список пропущенных аргументов с помощью функций `rep()` и `missing_arg()`, после чего воспользуемся оператором расцитирования нескольких аргументов (`!!!`), чтобы собрать все в вызов:

```
indices <- rep(list(missing_arg()), 3)
expr(x[!!!indices])
#> x[, , ]
```

Далее применим оператор присваивания для вставки индекса в нужную позицию:

```
indices[[2]] <- 1
expr(x[!!!indices])
#> x[, 1, ]
```

В заключение обернем все в функцию, в которой пару раз воспользуемся функцией `stopifnot()` для очистки интерфейса:

```
slice <- function(x, along, index) {
    stopifnot(length(along) == 1)
    stopifnot(length(index) == 1)

    nd <- length(dim(x))
    indices <- rep(list(missing_arg()), nd)
    indices[[along]] <- index

    expr(x[!!!indices])
}

x <- array(sample(30), c(5, 2, 3))
slice(x, 1, 3)
#> x[3, , ]
slice(x, 2, 2)
#> x[, 2, ]
slice(x, 3, 1)
#> x[, , 1]
```

Настоящая функция `slice()` вычислила бы созданное выражение (об этом мы поговорим в главе 20), но здесь нам важнее было показать процесс генерирования кода, поскольку именно он вызывает больше всего вопросов.

19.7.4 Создание функций

Еще одним применением механизма цитирования является создание функций «вручную», с использованием функции `rlang::new_function()`. Она генерирует функцию с применением трех компонентов (раздел 6.2.1): аргументов, тела и (необязательно) окружения:

```
new_function(
    exprs(x = , y = ),
    expr({x + y})
)
#> function (x, y)
#> {
#> x + y
#> }
```

Примечание. Отсутствующие аргументы в `exprs()` позволяют сгенерировать аргументы без значений по умолчанию.

Функцию `new_function()` можно использовать в качестве альтернативы фабрикам функций со скалярными или символьными аргументами. К примеру, мы могли бы создать функцию, генерирующую функцию возведения в определенную степень.

```
power <- function(exponent) {
    new_function(
        exprs(x = ),
        expr({
            x ^ !!exponent
        }),
        caller_env()
    )
}

power(0.5)
#> function (x)
#> {
#>     x^0.5
#> }
```

Также функцию `new_function()` можно применить для создания функций, работающих подобно `graphics::curve()`, позволяющей выводить график математического выражения без создания функции, как показано на рис. 19.8:

```
curve(sin(exp(4 * x)), n = 1000)
```

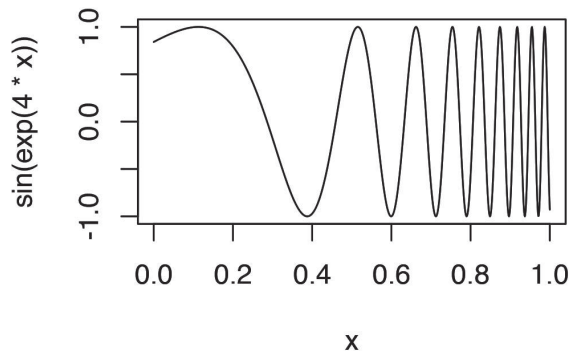


Рис. 19.8 Вывод графика математического выражения

В этом коде `x` является местоимением: он представляет собой не конкретное значение, а заменитель, варьирующийся на протяжении всего графика. Один из способов реализовать функцию `curve()` – превратить выражение в функцию с одним аргументом `x` и затем ее вызывать:

```
curve2 <- function(expr, xlim = c(0, 1), n = 100) {
    expr <- enexpr(expr)
    f <- new_function(exprs(x = ), expr)

    x <- seq(xlim[1], xlim[2], length = n)
    y <- f(x)

    plot(x, y, type = "l", ylab = expr_text(expr))
```

```
}
curve2(sin(exp(4 * x)), n = 1000)
```

Функции, подобные `curve()`, использующие выражения, которые содержат местоимения, известны как *анафорические функции*[1] (anaphoric function).

19.7.5 Упражнения

1. В примере с линейной моделью можно было бы заменить `expr()` в выражении `reduce(summands, ~ expr(!!.x + !!.y))` на `call2()`: `reduce(summands, call2, "+")`. Сравните и противопоставьте оба подхода. Какой вариант более легкий для чтения?
2. Перепишите механизм *преобразования Бокса-Кокса* (Box-Cox transform), показанный ниже, с использованием расцитирования и функции `new_function()`:

```
bc <- function(lambda) {
    if (lambda == 0) {
        function(x) log(x)
    } else {
        function(x) (x ^ lambda - 1) / lambda
    }
}
```

3. Перепишите простую функцию `compose()`, показанную ниже, с использованием квазицитирования и функции `new_function()`:

```
compose <- function(f, g) {
    function(...) f(g(...))
}
```

19.8 История

Идея квазицитирования весьма не нова. Впервые она была представлена философом Виллардом ван Орман Куином (Willard van Orman Quine)[2] в начале 1940-х. Этот подход был необходим философам для обозначения четкого разграничения между использованием слов и их значением, т. е. между самим объектом и словами, которыми мы его называем.

---

[1] Термин *анафорический* произошел от слова *анафора*, означающего выражение, зависящее от контекста. Анафорические функции можно встретить в Lisp-подобном языке Arc (http://arclanguage.github.io/ref/anaphoric.html), в Perl (https://www.perlmonks.org/index.pl?node_id=666047) и в Clojure.

[2] Вы можете быть знакомы с именем *Quine* благодаря самовоспроизводящимся программам, получившим название `quines`, которые при запуске выводят собственный исходный код.

В программировании квазицитирование впервые появилось в языке Lisp в середине 1970-х. В этом языке присутствует одна цитирующая функция, `, а символ , используется для расцитирования. Многие языки программирования, выросшие из Lisp, сохранили это поведение. К примеру, в Racket (` и @), Clojure (` и ~) и Julia (: и @) механизмы квазицитирования отличаются от подхода Lisp лишь незначительно. Во всех этих языках присутствует одна цитирующая функция, которую вы должны вызывать явно.

В то же время в языке R есть сразу несколько функций, цитирующих один или более входных аргументов. Это вносит некоторую неопределенность (поскольку вам необходимо постоянно читать документацию на предмет того, цитируются или вычисляются аргументы той или иной функции), но при этом позволяет писать очень элегантный и лаконичный код для исследования данных. В базовом R только одна функция (`bquote()`, написанная Томасом Ламли (Thomas Lumley) в 2003 году) поддерживает квазицитирование. Однако эта функция имеет ряд серьезных ограничений, помешавших ей широко распространиться в языке R (раздел 19.5).

Мои попытки преодолеть эти ограничения вылились в создание пакета `lazyeval` (2014–2015 годы). К сожалению, при его написании я не до конца осознавал всей глубины этих ограничений, так что, решив ряд проблем, я наплодил новые. И только в 2017 году в результате долгой и плодотворной совместной работы с Лайонелом Хенри (Lionel Henry) был создан фреймворк *tidy evaluation*. И сейчас, несмотря на относительную новизну этой концепции, я всех призываю применять ее, поскольку при умелом использовании она способна существенно облегчить решение сложнейших задач.

20

Вычисление

20.1 Введение

С точки зрения пользователя, обратной стороной цитирования является расцитирование, дающее ему возможность выборочно вычислять аргументы, которые иначе были бы цитированы. Если говорить со стороны разработчика, то дополнением к цитированию можно назвать *вычисление* (evaluation), предоставляющее ему возможность вычислять цитируемые выражения в определенном окружении для достижения поставленных целей.

Эту главу мы начнем с общей дискуссии о вычислениях в чистом виде. Вы узнаете о том, как можно при помощи функции `eval()` вычислять выражения в окружении, а также о ее использовании с целью реализации многих важных функций базового R. Получив основные знания, вы научитесь расширять вычисления, делая их более надежными. На этом этапе вы освоите две следующие важные идеи:

- *quosure* – структуры данных для захвата выражений вместе с окружением;
- маски данных, облегчающие задачу вычисления выражений в контексте датафреймов. Здесь может возникнуть некая неопределенность, которую мы без труда разрешим.

В совокупности квазицитирование, структуры *quosure* и маски данных образуют концепцию, называемую *tidy evaluation*, или сокращенно *tidy eval*. Эта концепция представляет собой принципиальный подход к нестандартным вычислениям, позволяющий использовать такие функции как интерактивно, так и совместно с другими функциями. *Tidy evaluation* фактически является вершиной этой теории, так что мы потратим определенное время на ее изучение. В заключение данной главы мы посмотрим, как дела с этим обстоят в базовом R, и узнаем о способах нивелирования присутствующих в нем недостатков.

Структура главы

- В разделе 20.2 мы поговорим об основах вычислений при помощи функции `eval()` и узнаем, как можно использовать ее для реализации таких базовых функций, как `local()` и `source()`.

- В разделе 20.3 мы познакомимся с новой структурой данных, *quosure*, способной объединять в себе выражение и окружение. Мы узнаем, как можно захватывать *quosure* из промисов и вычислять их при помощи функции `rlang::eval_tidy()`.
- Раздел 20.4 будет посвящен расширению вычислений при помощи маски данных, позволяющему связывать символы в окружении с переменными в датафрейме.
- В разделе 20.5 вы узнаете, как применять концепцию *tidy evaluation* на практике. Здесь мы сосредоточимся на распространенных шаблонах, связанных с цитированием и расцитированием, а также научимся преодолевать возникающие при вычислениях неопределенности.
- В разделе 20.6 мы вернемся к вычислениям в базовом R, обсудим присутствующие в этой области недостатки и узнаем, как можно использовать квазицитирование и вычисления для обертки функций, использующих NSE.

Требования

Для комфортного чтения этой главы вы должны быть хорошо знакомы с материалами, освещенными в главах 18 и 19, а также понимать, что из себя представляют окружения (раздел 7.2) и, в частности, что такое вызывающее окружение (раздел 7.5).

В этой главе мы продолжим использовать пакеты rlang (https://rlang.r-lib.org) и purrr (https://purrr.tidyverse.org).

```
library(rlang)
library(purrr)
```

20.2 Основы вычислений

В данном разделе мы подробно поговорим о функции `eval()`, которую бегло рассмотрели в предыдущей главе. Эта функция принимает два аргумента: `expr` и `envir`. Первый аргумент представляет объект для вычисления (обычно символ или выражение)[1]. Ни одна из функций вычисления не цитирует входные аргументы, так что обычно вы будете использовать их совместно с функцией `expr()` или другими похожими:

```
x <- 10
eval(expr(x))
#> [1] 10
```

[1] Все остальные объекты при передаче в функцию `eval()` возвращаются в исходном виде, т. е. `eval(x)` вернет `x`, если это не символ или выражение.

```
y <- 2
eval(expr(x + y))
#> [1] 12
```

Вторым аргументом в функцию передается окружение, в котором должно быть вычислено выражение, т. е. место поиска значений для имен `x`, `y` и `+`. По умолчанию передается текущее окружение, т. е. вызывающее окружение для функции `eval()`, но при необходимости вы можете переопределить его:

```
eval(expr(x + y), env(x = 1000))
#> [1] 1002
```

Первый аргумент вычисляется, а не цитируется, что может приводить к неожиданным результатам, если при использовании переопределенного окружения вы забудете выполнить цитирование вручную:

```
eval(print(x + 1), env(x = 1000))
#> [1] 11
#> [1] 11

eval(expr(print(x + 1)), env(x = 1000))
#> [1] 1001
```

Теперь, когда вы знаете основы вычислений, давайте рассмотрим несколько примеров их применения. Мы в основном будем говорить о базовых функциях R, которые вам наверняка доводилось использовать ранее, и будем реализовывать заложенные в них принципы с помощью пакета `rlang`.

20.2.1 Применение: local()

Иногда вам бывает необходимо выполнить блок вычислений, в процессе которых создаются некоторые промежуточные переменные. Эти переменные не имеют большой ценности, но могут занимать много места, так что от них лучше избавляться. Можно убирать за собой при помощи функции `rm()`. А можно оборачивать код в функцию и вызывать ее лишь раз. Но более элегантным решением является использование функции `local()`:

```
# Очистка переменных, созданных ранее
rm(x, y)

foo <- local({
    x <- 10
    y <- 200
    x + y
})

foo
#> [1] 210
```

```
x
#> Error in eval(expr, envir, enclos): object 'x' not found
y
#> Error in eval(expr, envir, enclos): object 'y' not found
```

Концепция функции `local()` очень проста, и мы реализуем ее ниже. Мы захватываем входящее выражение и создаем окружение, в котором будем его вычислять. Это новое окружение (так что присваивание значений переменным не повлияет на текущее окружение), родителем которого является вызывающее окружение, а значит, `expr` по-прежнему будет иметь доступ к переменным в этом окружении. Фактически этот подход имитирует запуск `expr`, как если бы она находилась внутри функции (в лексической области видимости, см. раздел 6.4).

```
local2 <- function(expr) {
    env <- env(caller_env())
    eval(enexpr(expr), env)
}

foo <- local2({
    x <- 10
    y <- 200
    x + y
})

foo
#> [1] 210
x
#> Error in eval(expr, envir, enclos): object 'x' not found
y
#> Error in eval(expr, envir, enclos): object 'y' not found
```

Понять, как работает функция `base::local()`, вам было бы намного сложнее, поскольку в ее реализации довольно сложно переплетены функции `eval()` и `substitute()`. Но разбираться в том, как именно все устроено в `eval()` и `substitute()`, бывает очень полезно, так что мы еще вернемся к этому в упражнениях.

20.2.2 Применение: source()

Мы можем создать упрощенную версию функции `source()` путем комбинирования функций `eval()` и `parse_expr()` из раздела 18.4.3. Смысл тут прост: мы читаем с диска файл, используем функцию `parse_expr()` для преобразования его содержимого в список выражений, после чего применяем функцию `eval()`, чтобы вычислить каждый элемент по очереди. Эта версия функции вычисляет код в вызывающем окружении и невидимо возвращает результат последнего выражения в файле, как и базовая функция `base::source()`.

```
source2 <- function(path, env = caller_env()) {
    file <- paste(readLines(path, warn = FALSE), collapse = "\n")
    exprs <- parse_exprs(file)

    res <- NULL
    for (i in seq_along(exprs)) {
        res <- eval(exprs[[i]], env)
    }

    invisible(res)
}
```

Код настоящей функции `source()` несколько сложнее, поскольку в ней есть возможность выводить выражения на экран и много дополнительных параметров, определяющих ее поведение.

Векторы выражений

В функции `base::eval()` заложено особое поведение применительно к векторам выражений, позволяющее вычислять каждый компонент по очереди. Это позволяет сделать реализацию функции `source2()` очень компактной благодаря тому, что функция `base::parse()` также возвращает выражение:

```
source3 <- function(file, env = parent.frame()) {
    lines <- parse(file)
    res <- eval(lines, envir = env)
    invisible(res)
}
```

Хотя реализация функции `source3()` получилась более лаконичной по сравнению с `source2()`, это единственное преимущество, связанное с использованием вектора выражений. В целом я не считаю, что этот плюс перевешивает расходы на введение еще одной структуры данных, так что в данной книге мы будем стараться избегать использования векторов выражений.

20.2.3 Ловушка: function()

Есть один небольшой нюанс при работе с функциями `eval()` и `expr()` для генерирования функций, о котором вам нужно знать:

```
x <- 10
y <- 20
f <- eval(expr(function(x, y) !!x + !!y))
f
#> function(x, y) !!x + !!y
```

Кажется, что эта функция не должна работать, но она работает:

```
f()
#> [1] 30
```

Это связано с тем, что, когда это возможно, функции выводят свой атрибут `srcref` (см. раздел 6.2.1), а поскольку этот атрибут принадлежит базовому R, он ничего не знает о квазицитировании.

Для обхода этой проблемы можно либо воспользоваться функцией `new_function()` (см. раздел 19.7.4) либо избавиться от атрибута `srcref`:

```
attr(f, "srcref") <- NULL
f
#> function (x, y)
#> 10 + 20
```

20.2.4 Упражнения

1. Внимательно прочитайте документацию к функции `source()`. Какое окружение в ней используется по умолчанию? Что будет, если передать ей параметр `local = TRUE`? И как передать функции другое окружение?
2. Предугадайте вывод следующих вызовов:

   ```
   eval(expr(eval(expr(eval(expr(2 + 2))))))
   eval(eval(expr(eval(expr(eval(expr(2 + 2)))))))
   expr(eval(expr(eval(expr(eval(expr(2 + 2)))))))
   ```

3. Заполните тело функций, приведенных ниже, реализовав тем самым функцию `get()` с использованием функций `sym()` и `eval()` и функцию `assign()` с использованием функций `sym()`, `expr()` и `eval()`. Не обращайте внимания на выбор окружения, реализованный в функциях `get()` и `assign()`. Мы предположим, что пользователь передает окружение явным образом.

   ```
   # name - это строка
   get2 <- function(name, env) {}
   assign2 <- function(name, value, env) {}
   ```

4. Измените функцию `source2()` таким образом, чтобы она возвращала результат *каждого* выражения, а не только последнего. Сможете обойтись без цикла `for`?
5. Мы можем улучшить читаемость функции `base::local()`, разбив ее на несколько строк:

   ```
   local3 <- function(expr, envir = new.env()) {
       call <- substitute(eval(quote(expr), envir))
   ```

```
    eval(call, envir = parent.frame())
}
```

Опишите словами, как работает функция `local()`. Подсказка: возможно, вам захочется воспользоваться выводом `print(call)`, чтобы понять, что делает функция `substitute()`, а также перечитать документацию, чтобы вспомнить, от какого окружения будет наследоваться `new.env()`.

20.3 Структура данных quosure

Почти каждое использование функции `eval()` включает в себя выражение и соответствующее окружение. Эта связка настолько важна, что для нее очень удобно иметь отдельную структуру данных, которая вместит обе составляющие. В базовом R такая структура отсутствует[1], и пакет `rlang` взялся заполнить эту пустоту с помощью структуры *quosure*, представляющей собой объект, вбирающий в себя одновременно и выражение, и окружение. Имя *quosure* восходит к непереводимой игре слов, замешанной на терминах *quoting* (цитирование) и *closure* (замыкание)[2], поскольку эта структура одновременно цитирует выражение и замыкает окружение. Структура *quosure* превращает внутренний объект промис (см. раздел 6.5.1) в нечто, с чем можно работать.

В данном разделе вы научитесь создавать и манипулировать объектами *quosure*, а также немного узнаете об их внутренней реализации.

20.3.1 Создание

Существует три способа создания объектов *quosure*:

- с помощью функций `enquo()` и `enquos()` для захвата пользовательских выражений. Подавляющее большинство этих объектов должны создаваться следующим образом:

```
foo <- function(x) enquo(x)
foo(a + b)
#> <quosure>
#> expr: ^a + b
#> env:  global
```

- для соответствия функциям `expr()` и `exprs()` в языке присутствуют функции `quo()` и `quos()`, но они включены только из соображений полноты

[1] Чисто технически формулы объединяют в себе выражение и окружение, но они тесно связаны с созданием моделей, так что для общих целей лучше иметь отдельную структуру данных.

[2] Мы могли бы поиграть словами и вывести новый термин вроде *цитикание* или *замытирование*, но не будем этим заниматься :). – *Прим. перев.*).

инструментальных средств и на практике применяются достаточно редко. При использовании этих функций подумайте о том, можно ли с помощью функции `expr()` и техники расцитирования избежать необходимости захвата окружения;

```
quo(x + y + z)
#> <quosure>
#> expr: ^x + y + z
#> env:  global
```

- с помощью функции `new_quosure()` можно создать объект *quosure* на основе его компонентов: выражения и окружения. Это редко требуется на практике, но очень полезно на этапе обучения, так что мы зачастую будем прибегать к такому подходу в данной главе.

```
new_quosure(expr(x + y), env(x = 1, y = 10))
#> <quosure>
#> expr: ^x + y
#> env:  0x7f87b9cc5fb8
```

20.3.2 Вычисление

Вычисление *quosure* выполняется посредством новой функции `eval_tidy()`, которая принимает на вход один объект *quosure* вместо пары из выражения и окружения. Использовать эту функцию очень просто:

```
q1 <- new_quosure(expr(x + y), env(x = 1, y = 10))
eval_tidy(q1)
#> [1] 11
```

В этом простом примере вызов `eval_tidy(q1)` равносилен составной инструкции `eval(get_expr(q1), get_env(q2))`. В то же время эта запись поддерживает две важные особенности *quosure*, о которых мы поговорим позже: вложенные объекты *quosure* (см. раздел 20.3.5) и местоимения (см. раздел 20.4.2).

20.3.3 Точки

Фактически объекты *quosure* можно использовать ради удобства – они существенно облегчают чтение кода из-за необходимости оперировать одним объектом вместо двух. Однако в них появляется крайняя необходимость при работе с ..., поскольку каждый аргумент, переданный в ..., может быть ассоциирован со своим окружением. Обратите внимание, что в следующем примере оба объекта *quosure* располагают одним и тем же выражением `x`, но окружения у них разные:

```
f <- function(...) {
    x <- 1
```

```
    g(..., f = x)
}
g <- function(...) {
    enquos(...)
}

x <- 0
qs <- f(global = x)
qs
#> <list_of<quosure>>
#>
#> $global
#> <quosure>
#> expr: ^x
#> env:  global
#>
#> $f
#> <quosure>
#> expr: ^x
#> env:  0x7f87bab6aa10
```

Это означает, что при их вычислении вы получите правильные результаты:

```
map_dbl(qs, eval_tidy)
#> global      f
#>      0      1
```

Корректное вычисление элементов ... и явилось главной причиной появления структуры данных *quosure*.

20.3.4 Под капотом

В основу использования структуры *quosure* в R легли формулы, позволяющие захватывать выражение и окружение:

```
f <- ~runif(3)
str(f)
#> Class 'formula' language ~runif(3)
#>   ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
```

В ранних версиях *tidy evaluation* вместо объектов *quosure* как раз использовались формулы, позволяющие выполнять цитирование с помощью нажатия всего одной клавиши ~. Но, к сожалению, не существует простого способа превратить ~ в квазицитирующую функцию.

Структура *quosure* представляет собой подкласс формул:

```
q4 <- new_quosure(expr(x + y + z))
class(q4)
#> [1] "quosure" "formula"
```

Это означает, что внутренне эти структуры, как и формулы, являются объектами вызова:

```
is_call(q4)
#> [1] TRUE

q4[[1]]
#> `~`
q4[[2]]
#> x + y + z
```

В соответствующем атрибуте этого объекта хранится окружение:

```
attr(q4, ".Environment")
#> <environment: R_GlobalEnv>
```

Если вам необходимо извлечь выражение или окружение, вам нет необходимости погружаться в детали реализации. Достаточно воспользоваться специальными функциями `get_expr()` и `get_env()`:

```
get_expr(q4)
#> x + y + z
get_env(q4)
#> <environment: R_GlobalEnv>
```

20.3.5 Вложенные quosure

Механизм квазицитирования может быть использован для встраивания *quosure* в выражение. Это достаточно мощный инструмент, о реализации которого вам зачастую не требуется думать, поскольку он прекрасно работает. В то же время вам стоит знать о вложенных *quosure*, чтобы уметь их распознавать и не пугаться их. Рассмотрите следующий пример, в котором два объекта *quosure* встраиваются в одно выражение:

```
q2 <- new_quosure(expr(x), env(x = 1))
q3 <- new_quosure(expr(x), env(x = 10))

x <- expr(!!q2 + !!q3)
```

Это выражение корректно вычисляется с помощью функции `eval_tidy()`:

```
eval_tidy(x)
#> [1] 11
```

В то же время если вывести это выражение на экран, вы увидите только иксы с явно прослеживающимся наследием от формул:

```
x
#> (~x) + ~x
```

Более приятный глазу вывод вы можете получить при помощи функции `rlang::expr_print()` (см. раздел 19.4.7):

```
expr_print(x)
#> (^x) + (^x)
```

При использовании функции `expr_print()` в консоли объекты *quosure* подсвечиваются цветом в соответствии со своим окружением, что облегчает процесс изучения символов и переменных.

20.3.6 Упражнения

1. Попробуйте угадать, что вернут следующие три объекта *quosure* при вычислении.

```
q1 <- new_quosure(expr(x), env(x = 1))
q1
#> <quosure>
#> expr: ^x
#> env:  0x7f87b70f83c8

q2 <- new_quosure(expr(x + !!q1), env(x = 10))
q2
#> <quosure>
#> expr: ^x + (^x)
#> env:  0x7f87baa5a908

q3 <- new_quosure(expr(x + !!q2), env(x = 100))
q3
#> <quosure>
#> expr: ^x + (^x + (^x))
#> env:  0x7f87bb0a5fb8
```

2. Напишите функцию `enenv()`, захватывающую окружение, ассоциированное с аргументом. Подсказка: это потребует всего двух вызовов функций.

20.4 Маски данных

В этом разделе вы узнаете о масках данных, представляющих собой датафреймы, в которых вычисляемый код будет в первую очередь искать определения переменных. Идея масок данных лежит в основе таких базовых

функций, как `with()`, `subset()` и `transform()`, и активно используется в пакетах tidyverse вроде `dplyr` и `ggplot2`.

20.4.1 Основы

Маска данных (data mask) позволяет совмещать переменные из окружения и из датафрейма в одном выражении. Маска данных передается вторым аргументом в функцию `eval_tidy()`:

```
q1 <- new_quosure(expr(x * y), env(x = 100))
df <- data.frame(y = 1:10)
eval_tidy(q1, df)
#> [1]  100  200  300  400  500  600  700  800  900 1000
```

Этот код непросто понять, поскольку здесь все объекты создаются с нуля. Все будет гораздо проще, если написать соответствующую обертку. Назовем ее `with2()`, ведь по функционалу она будет похожа на базовую функцию `base::with()`.

```
with2 <- function(data, expr) {
    expr <- enquo(expr)
    eval_tidy(expr, data)
}
```

Теперь можно переписать код, показанный выше, следующим образом:

```
x <- 100
with2(df, x * y)
#>  [1]  100  200  300  400  500  600  700  800  900 1000
```

Функция `base::eval()` имеет схожее назначение, хотя и не называет это маской данных. Таким образом, вы могли бы передать ей вторым аргументом датафрейм, а третьим – окружение. Получили бы такую реализацию функции `with()`:

```
with3 <- function(data, expr) {
    expr <- substitute(expr)
    eval(expr, data, caller_env())
}
```

20.4.2 Местоимения

Использование масок данных может приводить к неопределенностям. К примеру, в коде, показанном ниже, сразу непонятно, откуда берется `x` – из маски данных или из окружения, – если вы не знаете содержимое датафрейма `df`.

```
with2(df, x)
```

Это может затруднить чтение кода из-за необходимости хорошего знания контекста, что способно приводить к разного рода ошибкам. Для разрешения подобных проблем в масках данных используется два специальных префикса, или *местоимения* (pronoun): `.data` и `.env`:

- `.data$x` всегда ссылается на `x` в маске данных;
- `.env$x` всегда ссылается на `x` в окружении.

```
x <- 1
df <- data.frame(x = 2)

with2(df, .data$x)
#> [1] 2
with2(df, .env$x)
#> [1] 1
```

Вы также можете обращаться к `.data` и `.env` с помощью оператора извлечения подмножества `[[`, например `.data[["x"]]`. В остальном местоимения являются специальными объектами, и вам не стоит ожидать от них поведения датафрейма или окружения. В частности, они выбрасывают ошибку, в случае если объект не найден:

```
with2(df, .data$y)
#> Error: Column `y` not found in `.data`
```

20.4.3 Применение: subset()

Мы исследуем принципы работы *tidy evaluation* в контексте функции `base::subset()` – простой, но очень мощной функции, облегчающей процесс манипулирования данными. Если вы не пользовались этой функцией раньше, она, подобно `dplyr::filter()`, предлагает удобный способ извлечения строк из датафрейма. Вы передаете ей на вход данные, а также выражение, реализованное в контексте этих данных. При таком подходе вам не придется без конца повторять в коде имя датафрейма:

```
sample_df <- data.frame(a = 1:5, b = 5:1, c = c(5, 3, 1, 4, 1))

# Сокращение для sample_df[sample_df$a >= 4, ]
subset(sample_df, a >= 4)
#>   a b c
#> 4 4 2 4
#> 5 5 1 1

# Сокращение для sample_df[sample_df$b == sample_df$c, ]
subset(sample_df, b == c)
#>   a b c
#> 1 1 5 5
#> 5 5 1 1
```

Суть нашей версии функции subset() будет крайне проста. Она будет принимать на вход два аргумента: датафрейм, data, и выражение, rows. Далее будет вычислено выражение rows с использованием df в качестве маски данных, после чего нужные данные будут извлечены с помощью оператора [и возвращены пользователю. Я включил в нашу функцию простую проверку на то, что результат является логическим вектором. В реалистичном коде нужно было бы позаботиться о более информативных сообщениях об ошибках.

```
subset2 <- function(data, rows) {
    rows <- enquo(rows)
    rows_val <- eval_tidy(rows, data)
    stopifnot(is.logical(rows_val))

    data[rows_val, , drop = FALSE]
}

subset2(sample_df, b == c)
#>   a b c
#> 1 1 5 5
#> 5 5 1 1
```

20.4.4 Применение: transform()

Более сложное поведение реализовано в функции base::transform(), позволяющей добавлять новые переменные в датафрейм, вычисляя их выражения в контексте существующих переменных:

```
df <- data.frame(x = c(2, 3, 1), y = runif(3))
transform(df, x = -x, y2 = 2 * y)
#>    x      y    y2
#> 1 -2 0.0808 0.162
#> 2 -3 0.8343 1.669
#> 3 -1 0.6008 1.202
```

Наша реализация функции transform2() потребует написания дополнительного кода. Мы будем захватывать невычисленный аргумент ... с помощью функции enquos(...), после чего будем вычислять каждое выражение в цикле for. В реалистичном коде нам нужно было бы прописать разные проверки, чтобы убедиться, что все входные данные у нас именованные и при вычислении дают векторы той же длины, что и data.

```
transform2 <- function(.data, ...) {
    dots <- enquos(...)

    for (i in seq_along(dots)) {
        name <- names(dots)[[i]]
        dot <- dots[[i]]
        .data[[name]] <- eval_tidy(dot, .data)
```

```
    }

    .data
}

transform2(df, x2 = x * 2, y = -y)
#>   x       y x2
#> 1 2 -0.0808  4
#> 2 3 -0.8343  6
#> 3 1 -0.6008  2
```

Примечание. Я назвал первый аргумент `.data`, чтобы избежать проблем в случае, если пользователь даст одной из переменных имя `data`. Конечно, если он додумается назвать ее `.data`, проблемы возникнут, но все-таки это гораздо менее вероятно. По этой же причине аргументы функции `map()` названы `.x` и `.f` (см. раздел 9.2.4).

20.4.5 Применение: select()

Обычно маска данных представлена датафреймом, но иногда бывает полезно произвести какие-то вычисления над списком с более экзотичным содержимым. Именно для этого существует аргумент `select` в функции `base::subset()`. Он позволяет ссылаться на переменные так, как если бы они были представлены числами:

```
df <- data.frame(a = 1, b = 2, c = 3, d = 4, e = 5)
subset(df, select = b:d)
#>   b c d
#> 1 2 3 4
```

Ключевая идея состоит в создании именованного списка, в котором были бы представлены порядковые номера его компонентов:

```
vars <- as.list(set_names(seq_along(df), names(df)))
str(vars)
#> List of 5
#>  $ a: int 1
#>  $ b: int 2
#>  $ c: int 3
#>  $ d: int 4
#>  $ e: int 5
```

Реализация потребует написания всего нескольких строк кода:

```
select2 <- function(data, ...) {
    dots <- enquos(...)
```

```
    vars <- as.list(set_names(seq_along(data), names(data)))
    cols <- unlist(map(dots, eval_tidy, vars))

    df[, cols, drop = FALSE]
}
select2(df, b:d)
#>   b c d
#> 1 2 3 4
```

В функции `dplyr::select()` используется эта же идея, и на ее основе реализованы несколько полезных вспомогательных функций, позволяющих выбирать переменные, основываясь на их именах (например, `starts_with("x")` или `ends_with("_a")`).

20.4.6 Упражнения

1. Почему я воспользовался в реализации функции `transform2()` циклом `for` вместо применения функции `map()`? Рассмотрите вариант вызова `transform2(df, x = x * 2, x = x * 2)`.
2. Ниже показана альтернативная реализация функции `subset2()`:

   ```
   subset3 <- function(data, rows) {
       rows <- enquo(rows)
       eval_tidy(expr(data[!!rows, , drop = FALSE]), data = data)
   }

   df <- data.frame(x = 1:3)
   subset3(df, x == 1)
   ```

   Сравните и противопоставьте реализации `subset3()` и `subset2()`. Какие у каждой из них есть преимущества и недостатки?
3. В коде, приведенном ниже, представлена реализация базового функционала `dplyr::arrange()`. Напишите комментарии к каждой строке в функции. Можете ли вы объяснить, почему выражение `!!.na.last` является вполне корректным, но и пропуск `!!`, скорее всего, не приведет к появлению проблем?

   ```
   arrange2 <- function(.df, ..., .na.last = TRUE) {
       args <- enquos(...)
       order_call <- expr(order(!!!args, na.last = !!.na.last))

       ord <- eval_tidy(order_call, .df)
       stopifnot(length(ord) == nrow(.df))

       .df[ord, , drop = FALSE]
   }
   ```

20.5 Использование tidy evaluation

Хотя очень важно понимать работу функции `eval_tidy()`, вы не так часто будете обращаться к ней напрямую. Обычно вы будете использовать в работе функции, которые будут обращаться к функции `eval_tidy()`. В этом разделе мы рассмотрим несколько практических примеров обертки функций, использующих *tidy evaluation*.

20.5.1 Цитирование и расцитирование

Представьте, что мы написали функцию для извлечения выборок из датафрейма:

```
resample <- function(df, n) {
    idx <- sample(nrow(df), n, replace = TRUE)
    df[idx, , drop = FALSE]
}
```

Теперь мы хотим создать новую функцию, которая будет извлекать нужное нам подмножество, а затем возвращать выборку из него. Наивный подход здесь не сработает:

```
subsample <- function(df, cond, n = nrow(df)) {
    df <- subset2(df, cond)
    resample(df, n)
}

df <- data.frame(x = c(1, 1, 1, 2, 2), y = 1:5)
subsample(df, x == 1)
#> Error in eval_tidy(rows, data): object 'x' not found
```

Функция `subsample()` не цитирует свои аргументы, так что аргумент `cond` вычисляется обычным способом (без маски данных), и мы получаем ошибку при попытке поиска привязки для имени `x`. Для решения этой проблемы нам необходимо цитировать аргумент `cond`, а затем расцитировать его при передаче функции `subset2()`:

```
subsample <- function(df, cond, n = nrow(df)) {
    cond <- enquo(cond)

    df <- subset2(df, !!cond)
    resample(df, n)
}

subsample(df, x == 1)
#>     x y
```

```
#> 1   1 1
#> 1.1 1 1
#> 2   1 2
```

Это довольно распространенный шаблон. Всякий раз при вызове цитирующей функции с аргументами от пользователя вам необходимо цитировать их, а затем расцитировать.

20.5.2 Разрешение неопределенности

В предыдущем примере нам нужно было думать о *tidy evaluation* в связи с квазицитированием. Но помнить об этом нужно даже тогда, когда обертке нет необходимости цитировать аргументы. Взгляните на следующую функцию-обертку вокруг `subset2()`:

```
threshold_x <- function(df, val) {
    subset2(df, x >= val)
}
```

Эта функция может возвращать неправильные результаты в двух случаях:

- если `x` присутствует в вызывающем окружении, но не в `df`:

```
x <- 10
no_x <- data.frame(y = 1:3)
threshold_x(no_x, 2)
#>   y
#> 1 1
#> 2 2
#> 3 3
```

- если `val` присутствует в `df`:

```
has_val <- data.frame(x = 1:3, val = 9:11)
threshold_x(has_val, 2)
#> [1] x   val
#> <0 rows> (or 0-length row.names)
```

Такого рода ошибки возникают из-за неопределенных ситуаций в *tidy evaluation*, заключающихся в том, что переменная может быть найдена как в маске данных, так и в окружении. Избавиться от этой неопределенности можно при помощи специальных местоимений `.data` и `.env`:

```
threshold_x <- function(df, val) {
    subset2(df, .data$x >= .env$val)
}

x <- 10
threshold_x(no_x, 2)
```

```
#> Error: Column `x` not found in `.data`
threshold_x(has_val, 2)
#>   x val
#> 2 2  10
#> 3 3  11
```

В основном вместо местоимения `.env` вы можете использовать оператор расцитирования, как показано ниже:

```
threshold_x <- function(df, val) {
    subset2(df, .data$x >= !!val)
}
```

Здесь есть незначительные отличия в плане того, когда будет вычисляться переменная `val`. При использовании оператора расцитирования переменная будет вычислена сразу с помощью функции `enquo()`, а при задействовании местоимения `.env` она будет вычисляться отложенно, в функции `eval_tidy()`. Обычно это отличие не играет большой роли, так что вы можете выбирать тот вариант, который вам больше по душе.

20.5.3 Цитирование и неопределенность

В завершение нашей дискуссии давайте рассмотрим случай, в котором присутствует и цитирование, и потенциальная неопределенность. Для этого мы обобщим функцию `threshold_x()`, чтобы пользователь мог сам выбрать пороговое значение. В данном случае я воспользовался записью `.data[[var]]`, поскольку это облегчает чтение кода. В упражнениях вы сможете опробовать прием с использованием оператора `$`.

```
threshold_var <- function(df, var, val) {
    var <- as_string(ensym(var))
    subset2(df, .data[[var]] >= !!val)
}

df <- data.frame(x = 1:10)
threshold_var(df, x, 8)
#>    x
#> 8  8
#> 9  9
#> 10 10
```

Однако не всегда ответственность за решение неопределенности лежит на разработчике функции. Представьте, что мы еще больше обобщили функцию, и теперь пороговое значение может вычисляться на основании любого выражения:

```
threshold_expr <- function(df, expr, val) {
    expr <- enquo(expr)
```

```
    subset2(df, !!expr >= !!val)
}
```

Невозможно вычислить `expr` только в маске данных, поскольку маска не включает в себя функции вроде + и ==. В этом случае ответственность за разрешение неопределенности лежит исключительно на пользователе. Вы как автор функции обязаны исключить любые неопределенности с выражениями, которые создаете вы сами, а ответственность за выражения, создаваемые пользователем, лежит на нем.

20.5.4 Упражнения

1. Ниже приведена альтернативная версия функции `threshold_var()`. Чем этот подход отличается от использованного нами ранее? Что делает эту функцию более сложной для понимания?

```
threshold_var <- function(df, var, val) {
    var <- ensym(var)
    subset2(df, `$`(.data, !!var) >= !!val)
}
```

20.6 Вычисления в базовом R

Теперь, когда вы понимаете принципы вычислений *tidy evaluation*, пришло время обратиться к альтернативным подходам, используемым в базовом R. Здесь мы рассмотрим два наиболее часто применяемых подхода:

- функция `substitute()` и вычисления в вызывающем окружении, как в функции `subset()`. Я покажу эту технику и продемонстрирую все ее неудобства в использовании, описанные в документации к функции `subset()`;
- функция `match.call()`, позволяющая манипулировать вызовами и производить вычисления в вызывающем окружении, как в функциях `write.csv()` и `lm()`. На примере этой техники я покажу, как квазицитирование и (обычные) вычисления могут помочь в написании оберток вокруг таких функций.

Эти два подхода демонстрируют наиболее часто используемые формы *нестандартного вычисления* (non-standard evaluation – NSE).

20.6.1 substitute()

Одной из самых часто применяемых в базовом R форм NSE является сочетание функций `substitute()` и `eval()`. Ниже показан пример того, как мож-

но написать ядро функции subset() с использованием функций substitute() и eval() вместо enquo() и eval_tidy(). Здесь я также привожу код, представленный в разделе 20.4.3, чтобы вы могли сами провести сравнение. Основное отличие состоит в окружении вычисления: в функции subset_base() аргумент вычисляется в вызывающем окружении, а в subset_tidy() – в окружении, в котором он был определен.

```
subset_base <- function(data, rows) {
    rows <- substitute(rows)
    rows_val <- eval(rows, data, caller_env())
    stopifnot(is.logical(rows_val))

    data[rows_val, , drop = FALSE]
}

subset_tidy <- function(data, rows) {
    rows <- enquo(rows)
    rows_val <- eval_tidy(rows, data)
    stopifnot(is.logical(rows_val))

    data[rows_val, , drop = FALSE]
}
```

20.6.1.1 Программирование с использованием subset()

В документации к функции subset() содержится следующее предупреждение:

> *Это вспомогательная функция, предназначенная для интерактивного использования. При написании программ рекомендуется воспользоваться стандартными функциями извлечения подмножеств вроде* [*, к тому же нестандартное вычисление аргумента* subset *может иметь непредвиденные последствия.*

Здесь есть три основные проблемы:

- функция base::subset() всегда вычисляет rows в вызывающем окружении, но при использовании ... может потребоваться вычислить выражение где-то еще:

```
f1 <- function(df, ...) {
    xval <- 3
    subset_base(df, ...)
}

my_df <- data.frame(x = 1:3, y = 3:1)
xval <- 1
f1(my_df, x == xval)
#>   x y
#> 3 3 1
```

Данная проблема может показаться надуманной, но это означает, что функция `subset_base()` не может надежно работать с функционалами вроде `map()` или `lapply()`:

```
local({
    zzz <- 2
    dfs <- list(data.frame(x = 1:3), data.frame(x = 4:6))
    lapply(dfs, subset_base, x == zzz)
})
#> Error in eval(rows, data, caller_env()): object 'zzz' not found
```

- вызов функции `subset()` из другой функции требует определенной осторожности: вам необходимо воспользоваться функцией `substitute()` для захвата вызова полного выражения `subset()`, после чего отправить его на вычисление. Код, приведенный ниже, не так прост для чтения, поскольку функция `substitute()` не использует синтаксические маркеры для расцитирования. Я вывел на экран объект вызова для облегчения понимания того, что на самом деле происходит;

```
f2 <- function(df1, expr) {
    call <- substitute(subset_base(df1, expr))
    expr_print(call)
    eval(call, caller_env())
}

my_df <- data.frame(x = 1:3, y = 3:1)
f2(my_df, x == 1)
#> subset_base(my_df, x == 1)
#>   x y
#> 1 1 3
```

- в функции `eval()` не предусмотрено использование местоимений, в связи с чем отсутствует возможность указать, что часть выражения должна браться из данных. Насколько я могу судить, не существует способа обезопасить приведенную ниже функцию, за исключением ручной проверки существования переменной `z` в датафрейме `df`.

```
f3 <- function(df) {
    call <- substitute(subset_base(df, z > 0))
    expr_print(call)
    eval(call, caller_env())
}

my_df <- data.frame(x = 1:3, y = 3:1)
z <- -1
f3(my_df)
#> subset_base(my_df, z > 0)
#> [1] x y
#> <0 rows> (or 0-length row.names)
```

20.6.1.2 А как насчет [?

С учетом того, что концепция *tidy evaluation* достаточно сложна, почему бы просто не воспользоваться оператором `[`, как рекомендует справка `?subset`? Честно говоря, меня не очень привлекают функции, которые могут быть использованы только в интерактивном режиме, а не в других функциях.

Кроме того, даже в простейшей функции `subset()` представлены две полезные возможности в сравнении с `[`:

- в ней по умолчанию используется значение аргумента `drop = FALSE`, что гарантирует возврат датафрейма;
- в ней отбрасываются строки, для которых вычисление условия дает в результате `NA`.

Это означает, что вызов `subset(df, x == y)` не будет эквивалентен записи `df[x == y,]`, как вы могли бы предположить. Вместо этого его эквивалентом будет `df[x == y & !is.na(x == y), , drop = FALSE]`: как здесь много букв! Реальные альтернативы функции `subset()` вроде функции `dplyr::filter()` предлагают еще больше возможностей. К примеру, функция `dplyr::filter()` умеет преобразовывать выражения R в инструкции SQL, которые могут выполняться в базе данных. Это делает использование функции `filter()` очень полезным и практичным.

20.6.2 match.call()

Еще одной распространенной формой NSE является захват полного объекта вызова с помощью функции `match.call()`, его модифицирование и дальнейшее вычисление полученного результата. Использование функции `match.call()` похоже на применение `substitute()`, но вместо захвата одного аргумента осуществляется захват всего вызова. В пакете `rlang` нет аналога этой функции.

```
g <- function(x, y, z) {
    match.call()
}
g(1, 2, z = 3)
#> g(x = 1, y = 2, z = 3)
```

Наиболее ярким примером использования функции `match.call()` является функция `write.csv()`, в которой исходный вызов преобразуется в вызов функции `write.table()` с подходящим набором аргументов. Ниже показана суть функции `write.csv()`:

```
write.csv <- function(...) {
    call <- match.call(write.table, expand.dots = TRUE)

    call[[1]] <- quote(write.table)
    call$sep <- ","
    call$dec <- "."
```

```
    eval(call, parent.frame())
}
```

Мне не кажется подобная техника интересной, поскольку такого же результата можно добиться и без использования NSE:

```
write.csv <- function(...) {
    write.table(..., sep = ",", dec = ".")
}
```

Так или иначе, эту технику очень важно знать и понимать, поскольку она частенько используется в функциях для создания моделей. Кроме того, эти функции явным образом выводят захваченный вызов, что, как вы увидите далее, создает некоторые проблемы.

20.6.2.1 Обертки для функций создания моделей

Для начала рассмотрим простейшую из всех возможных обертку для функции `lm()`:

```
lm2 <- function(formula, data) {
    lm(formula, data)
}
```

Такая обертка работает, но далеко не оптимально, поскольку функция `lm()` захватывает свой вызов и выводит его на экран:

```
lm2(mpg ~ disp, mtcars)
#>
#> Call:
#> lm(formula = formula, data = data)
#>
#> Coefficients:
#> (Intercept)        disp
#>     29.5999     -0.0412
```

Полученный вывод необходимо поправить, поскольку с помощью него мы получаем всю необходимую информацию о спецификации построенной модели. Для этого достаточно захватить аргументы, создать вызов функции `lm()` с использованием расцитирования и затем выполнить его. Для лучшего понимания происходящего давайте выведем на экран итоговое выражение. В случае с более сложными моделями такой подход окажется еще более полезным и информативным.

```
lm3 <- function(formula, data, env = caller_env()) {
    formula <- enexpr(formula)
    data <- enexpr(data)
```

```
    lm_call <- expr(lm(!!formula, data = !!data))
    expr_print(lm_call)
    eval(lm_call, env)
}

lm3(mpg ~ disp, mtcars)
#> lm(mpg ~ disp, data = mtcars)
#>
#> Call:
#> lm(formula = mpg ~ disp, data = mtcars)
#>
#> Coefficients:
#> (Intercept)       disp
#>     29.5999    -0.0412
```

Есть три составляющие, которые необходимо учитывать при создании оберток базовых функций NSE представленным выше способом:

- невычисляемые аргументы мы захватываем с помощью функции `enexpr()`, а вызывающее окружение – с помощью `caller_env()`;
- новое выражение мы создаем посредством функции `expr()` и техники расцитирования;
- вычисляем полученное выражение в вызывающем окружении. Вы должны принять тот факт, что функция не будет корректно работать, если аргументы не будут определены в вызывающем окружении. Передача аргумента `env` позволит разработчику разрешить конфликт в случае возникновения проблем с окружением по умолчанию.

Использование функции `enexpr()` имеет приятный побочный эффект, состоящий в возможности применять расцитирование для динамического генерирования формул:

```
resp <- expr(mpg)
disp1 <- expr(vs)
disp2 <- expr(wt)
lm3(!!resp ~ !!disp1 + !!disp2, mtcars)
#> lm(mpg ~ vs + wt, data = mtcars)
#>
#> Call:
#> lm(formula = mpg ~ vs + wt, data = mtcars)
#>
#> Coefficients:
#> (Intercept)        vs         wt
#>       33.00      3.15      -4.44
```

20.6.2.2 Окружение вычисления

Что, если вам необходимо смешать объекты, переданные пользователем, с объектами, созданными в функции? Допустим, вы решили создать версию

функции `lm()` с автоматической выборкой данных. Вы могли бы попробовать реализовать это как-то так:

```
resample_lm0 <- function(formula, data, env = caller_env()) {
    formula <- enexpr(formula)
    resample_data <- resample(data, n = nrow(data))

    lm_call <- expr(lm(!!formula, data = resample_data))
    expr_print(lm_call)
    eval(lm_call, env)
}

df <- data.frame(x = 1:10, y = 5 + 3 * (1:10) + round(rnorm(10), 2))
resample_lm0(y ~ x, data = df)
#> lm(y ~ x, data = resample_data)
#> Error in is.data.frame(data): object 'resample_data' not found
```

Почему этот код не работает? Мы вычисляем `lm_call` в вызывающем окружении, но переменная `resample_data` располагается в окружении выполнения. Мы могли бы попробовать запустить вычисление в окружении выполнения функции `resample_lm0()`, но нет никакой гарантии, что имя `formula` в нем сможет быть вычислено.

Существует два основных способа для решения этой проблемы.

1. Расцитирование датафрейма в вызов. Это избавит код от необходимости выполнять поиск, но потянет за собой все проблемы, характерные для встраивания выражений, о которых мы говорили в разделе 19.4.7. В случае с функциями создания моделей это приведет к тому, что захваченный вызов будет неоптимальным:

   ```
   resample_lm1 <- function(formula, data, env = caller_env()) {
       formula <- enexpr(formula)
       resample_data <- resample(data, n = nrow(data))

       lm_call <- expr(lm(!!formula, data = !!resample_data))
       expr_print(lm_call)
       eval(lm_call, env)
   }

   resample_lm1(y ~ x, data = df)$call
   #> lm(y ~ x, data = <data.frame>)
   #> lm(formula = y ~ x, data = list(x = c(3L, 7L, 4L, 4L,
   #> 2L, 7L, 2L, 1L, 8L, 9L), y = c(13.21, 27.04, 18.63,
   #> 18.63, 10.99, 27.04, 10.99, 7.83, 28.14, 32.72)))
   ```

2. В качестве альтернативы можно создать новое окружение, унаследованное от вызывающего, и привязать к нему переменные, определенные внутри функции.

   ```
   resample_lm2 <- function(formula, data, env = caller_env()) {
       formula <- enexpr(formula)
   ```

```
    resample_data <- resample(data, n = nrow(data))

    lm_env <- env(env, resample_data = resample_data)
    lm_call <- expr(lm(!!formula, data = resample_data))
    expr_print(lm_call)
    eval(lm_call, lm_env)
}

resample_lm2(y ~ x, data = df)
#> lm(y ~ x, data = resample_data)
#>
#> Call:
#> lm(formula = y ~ x, data = resample_data)
#>
#> Coefficients:
#> (Intercept)        x
#>        5.17     3.00
```

Второй вариант требует больше усилий, но приводит к наиболее чистой спецификации.

20.6.3 Упражнения

1. Почему приведенная ниже функция выдает ошибку?

```
lm3a <- function(formula, data) {
    formula <- enexpr(formula)

    lm_call <- expr(lm(!!formula, data = data))
    eval(lm_call, caller_env())
}
lm3a(mpg ~ disp, mtcars)$call
#> Error in as.data.frame.default(data, optional = TRUE):
#> cannot coerce class '"function"' to a data.frame
```

2. При подборе модели отклик и данные обычно не меняются, в то время как с предикторами ведется активная работа. Напишите простую обертку, которая позволит снизить количество дублирующегося кода во фрагменте, показанном ниже.

```
lm(mpg ~ disp, data = mtcars)
lm(mpg ~ I(1 / disp), data = mtcars)
lm(mpg ~ disp * cyl, data = mtcars)
```

3. Еще один способ реализации функции `resample_lm()` предполагает включение выражения для создания выборки (`data[sample(nrow(data), replace = TRUE), , drop = FALSE]`) в аргумент `data`. Напишите такую функцию. Каковы ее преимущества и недостатки?

21

Транслирование кода R

21.1 Введение

Сочетание возможностей окружений первого класса, лексического поиска в области видимости и метапрограммирования в совокупности дает нам очень мощный набор инструментов для транслирования кода R в другие языки. Одним из примеров воплощения этой идеи является пакет `dbplyr`, обеспечивающий работу с базами данных для пакета `dplyr` и позволяющий напрямую транслировать процесс манипулирования данными из R в SQL. Ключевую идею этого транслирования можно видеть на примере функции `translate_sql()`, которая принимает код на R и возвращает эквивалент на SQL:

```
library(dbplyr)
translate_sql(x ^ 2)
#> <SQL> POWER(`x`, 2.0)
translate_sql(x < 5 & !is.na(x))
#> <SQL> `x` < 5.0 AND NOT(((`x`) IS NULL))
translate_sql(!first %in% c("John", "Roger", "Robert"))
#> <SQL> NOT(`first` IN ('John', 'Roger', 'Robert'))
translate_sql(select == 7)
#> <SQL> `select` = 7.0
```

Транслирование выражений R в инструкции SQL – задача не из легких по причине большого разнообразия диалектов SQL, так что мы разработаем два простых, но очень полезных *языка предметной области* (domain specific languages – *DSL*): один для генерирования разметки HTML, а второй – для создания математических уравнений в LaTeX.

Если вам интересна тема проектирования языков предметной области, я очень рекомендую прочитать книгу *Domain Specific Languages* Мартина Фаулера (Martin Fowler). В ней подробно обсуждается процесс разработки таких языков и приводится множество полезных примеров.

Структура главы

- В разделе 21.2 мы создадим DSL для генерирования разметки HTML с использованием квазицитирования, функций из пакета `purrr` и концепции *tidy evaluation*.

- В разделе 21.3 мы будем трансформировать математические выражения на языке R в соответствующие эквиваленты на LaTeX с использованием сочетания *tidy evaluation* и прохода по выражениям.

Требования

В данной главе мы воспользуемся множеством техник, о которых рассказывали в других главах этой книги. В частности, для комфортного чтения вам необходимо хорошо понимать все, что связано с окружениями, выражениями, *tidy evaluation*, функциональным программированием и объектной системой S3. Мы будем использовать пакет `rlang` (https://rlang.r-lib.org) для реализации техник метапрограммирования и пакет `purrr` (https://purrr.tidyverse.org) для функционального программирования.

```
library(rlang)
library(purrr)
```

21.2 HTML

Язык разметки *HTML* (HyperText Markup Language) лежит в основе подавляющего большинства сайтов в интернете. Этот язык представляет собой частный случай стандартного обобщенного языка разметки SGML (Standard Generalised Markup Language) и близок по смыслу, но не идентичен, с расширяемым языком разметки XML (eXtensible Markup Language). Код на языке HTML выглядит примерно так:

```
<body>
    <h1 id='first'>A heading</h1>
    <p>Some text &amp;amp; <b>some bold text.</b></p>
    <img src='myimg.png' width='100' height='100' />
</body>
```

Если вы никогда раньше не сталкивались с HTML, то можете заметить, что структура этого языка базируется на *тегах* (tag), которые выглядят следующим образом: `<tag></tag>` или `<tag />`. Теги могут быть вложены друг в друга и перемежаться с текстом. Существует более ста тегов HTML, но в этой главе мы будем работать лишь с несколькими из них:

- тег `<body>` является корневым и включает в себя все содержимое страницы;
- тег `<h1>` определяет заголовок первого уровня;
- тег `<p>` отвечает за абзац текста;
- тег `<b>` делает шрифт вложенного текста жирным;
- тег `<img>` позволяет вставить на страницу изображение.

Теги HTML могут обладать именованными *атрибутами* (attribute), которые выглядят примерно так: `<tag name1='value1' name2='value2'></tag>`. Двумя наиболее распространенными атрибутами тегов являются `id` и `class`, которые зачастую используются совместно с каскадными таблицами стилей *CSS* (Cascading Style Sheets) для управления визуальным отображением страницы.

Пустые теги (void tag), такие как `<img>`, не могут иметь дочерних тегов и автоматически закрываются: `<img />` вместо `<img></img>`. Из-за отсутствия вложенного содержимого в таких тегах на первый план выходят атрибуты, и у тега `<img>` насчитывается три важных атрибута: `src` (источник изображения), `width` и `height`.

Поскольку символы `<` и `>` в HTML наделены особым смыслом, вы не можете использовать их в тексте напрямую. Вместо этого вы должны воспользоваться специальными последовательностями – в данном случае это `&gt;` (от *greater than* – больше чем) и `&lt;` (от *less than* – меньше чем). Поскольку в этих последовательностях используется символ `&`, для применения самого амперсанда вам придется также воспользоваться специальной последовательностью `&amp;`.

21.2.1 Цель

Наша цель состоит в том, чтобы облегчить процесс генерирования кода HTML из R. К примеру, нам может понадобиться получить такую разметку:

```
<body>
    <h1 id='first'>A heading</h1>
    <p>Some text &amp;amp; <b>some bold text.</b></p>
    <img src='myimg.png' width='100' height='100' />
</body>
```

Мы можем воспользоваться кодом, максимально приближенным по структуре к требуемой разметке:

```
with_html(
    body(
        h1("A heading", id = "first"),
        p("Some text &", b("some bold text.")),
        img(src = "myimg.png", width = 100, height = 100)
    )
)
```

У этого DSL есть три важных свойства:

- вложенные вызовы функций в точности соответствуют вложенным тегам HTML;
- неименованные аргументы становятся содержимым тегов, а именованные – атрибутами;
- символ `&` и другие специальные символы автоматически экранируются (превращаются в соответствующие последовательности).

21.2.2 Экранирование

Экранирование (escaping) представляет собой фундаментальную часть любой операции транслирования, так что мы обсудим его в самом начале. Экранирование включает в себя две основные задачи:

- в пользовательском вводе нужно автоматически экранировать символы &, < и >;
- в то же время необходимо убедиться в том, что генерируемые нами символы &, < и > не подвержены двойному экранированию (т. е. что мы не получим задвоение вроде `&amp;`, `&lt;` и `&gt;`).

Простейший способ реализовать это – создать класс S3 (см. раздел 13.3), который будет различать обычный текст, нуждающийся в экранировании, и HTML, не нуждающийся в нем.

```
html <- function(x) structure(x, class = "advr_html")

print.advr_html <- function(x, ...) {
    out <- paste0("<HTML> ", x)
    cat(paste(strwrap(out), collapse = "\n"), "\n", sep = "")
}
```

После этого нужно написать обобщенную функцию экранирования. В ней будет два важных метода:

- метод `escape.character()` принимает на вход обычный символьный вектор и возвращает вектор HTML с экранированными специальными символами &, < и >;
- метод `escape.advr_html()` оставляет уже экранированный HTML нетронутым.

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

escape.character <- function(x) {
    x <- gsub("&", "&amp;", x)
    x <- gsub("<", "&lt;", x)
    x <- gsub(">", "&gt;", x)

    html(x)
}

escape.advr_html <- function(x) x
```

Теперь проверим, как это работает.

```
escape("This is some text.")
#> <HTML> This is some text.
escape("x > 1 & y < 2")
#> <HTML> x &gt; 1 &amp; y &lt; 2
```

```
# Двойное экранирование - не проблема
escape(escape("This is some text. 1 > 2"))
#> <HTML> This is some text. 1 &gt; 2

# И текст, который представляет HTML, не экранируется
escape(html("<hr />"))
#> <HTML> <hr />
```

Удобно то, что такое решение позволяет пользователю отказаться от экранирования, если он знает, что содержимое уже экранировано.

21.2.3 Базовые функции тегов

Сначала мы напишем функцию для обработки одного тега, после чего обобщим эту задачу, чтобы иметь возможность обрабатывать любые теги.

Начнем с простого тега `<p>`. У тегов HTML могут присутствовать как атрибуты вроде `id` или `class`, так и дочерние теги (`<b>`, `<i>` и т. д.). Нам необходимо как-то разделить эти понятия в вызове функции. С учетом того, что атрибуты именованы, а дочерние теги – нет, кажется логичным использовать для них именованные и неименованные аргументы соответственно. К примеру, вызов функции `p()` может выглядеть так:

```
p("Some text. ", b(i("some bold italic text")), class = "mypara")
```

Мы могли бы перечислить все возможные атрибуты тега `<p>` в определение функции, но это не так просто, поскольку атрибутов слишком много, к тому же есть возможность применять пользовательские атрибуты (http://html5doctor.com/html5-custom-data-attributes). Вместо этого мы будем использовать аргумент … и разделять компоненты на основании того, именованы они или нет. С учетом всего сказанного мы можем написать обертку к функции `rlang::list2()` (см. раздел 19.6), которая будет отдельно возвращать именованные и неименованные компоненты:

```
dots_partition <- function(...) {
    dots <- list2(...)

    if (is.null(names(dots))) {
        is_named <- rep(FALSE, length(dots))
    } else {
        is_named <- names(dots) != ""
    }

    list(
        named = dots[is_named],
        unnamed = dots[!is_named]
    )
}

str(dots_partition(a = 1, 2, b = 3, 4))
```

```
#> List of 2
#>  $ named :List of 2
#>   ..$ a: num 1
#>   ..$ b: num 3
#>  $ unnamed:List of 2
#>   ..$ : num 2
#>   ..$ : num 4
```

Теперь мы можем написать функцию `p()`. Обратите внимание на использование одной новой функции `html_attributes()`. Она принимает на вход именованный список и возвращает спецификацию атрибута HTML в виде строки. Это довольно непростая операция (частично из-за некоторых специфических особенностей HTML, о которых мы не будем говорить подробно), но в целом она не так важна, поскольку не несет в себе никаких новых идей программирования, так что не будем углубляться. Если вам интересно, вы можете ознакомиться с исходным кодом по адресу https://github.com/hadley/adv-r/blob/master/dsl-html-attributes.r.

```
source("dsl-html-attributes.r")
p <- function(...) {
    dots <- dots_partition(...)
    attribs <- html_attributes(dots$named)
    children <- map_chr(dots$unnamed, escape)

    html(paste0(
        "<p", attribs, ">",
        paste(children, collapse = ""),
        "</p>"
    ))
}

p("Some text")
#> <HTML> <p>Some text</p>
p("Some text", id = "myid")
#> <HTML> <p id='myid'>Some text</p>
p("Some text", class = "important", `data-value` = 10)
#> <HTML> <p class='important' data-value='10'>Some text</p>
```

21.2.4 Функции тегов

Функцию `p()` очень легко можно адаптировать под другие теги: для этого нужно просто заменить `"p"` на имя соответствующего тега. Можно очень элегантно сделать это с помощью функции `rlang::new_function()` (см. раздел 19.7.4), использующей технику расцитирования совместно с функцией `paste0()` для генерирования открывающих и закрывающих тегов.

```
tag <- function(tag) {
    new_function(
```

```
        exprs(... = ),
        expr({
            dots <- dots_partition(...)
            attribs <- html_attributes(dots$named)
            children <- map_chr(dots$unnamed, escape)

            html(paste0(
                !!paste0("<", tag), attribs, ">",
                paste(children, collapse = ""),
                !!paste0("</", tag, ">")
            ))
        }),
        caller_env()
    )
}

tag("b")
#> function (...)
#> {
#>     dots <- dots_partition(...)
#>     attribs <- html_attributes(dots$named)
#>     children <- map_chr(dots$unnamed, escape)
#>     html(paste0("<b", attribs, ">", paste(children, collapse = ""),
#>         "</b>"))
#> }
```

Нам пришлось использовать странную синтаксическую конструкцию `exprs(... = )` для создания пустого аргумента … в функции тега. За подробностями вы можете обратиться к разделу 18.6.2.

Теперь мы можем запустить наш предыдущий пример:

```
p <- tag("p")
b <- tag("b")
i <- tag("i")
p("Some text. ", b(i("some bold italic text")), class = "mypara")
#> <HTML> <p class='mypara'>Some text. <b><i>some bold italic
#> text</i></b></p>
```

Перед тем как сгенерировать функции для всех возможных тегов HTML, нам необходимо создать разновидность функции, обрабатывающей пустые теги. Функция `void_tag()` будет довольно сильно похожа на функцию `tag()`, но она будет выбрасывать ошибку при наличии неименованных тегов, и сам тег будет выглядеть несколько иначе.

```
void_tag <- function(tag) {
    new_function(
        exprs(... = ),
        expr({
            dots <- dots_partition(...)
            if (length(dots$unnamed) > 0) {
```

```
                abort(!!paste0("<", tag, "> must not have unnamed arguments"))
            }
            attribs <- html_attributes(dots$named)

            html(paste0(!!paste0("<", tag), attribs, " />"))
        }),
        caller_env()
    )
}

img <- void_tag("img")
img
#> function (...)
#> {
#>     dots <- dots_partition(...)
#>     if (length(dots$unnamed) > 0) {
#>         abort("<img> must not have unnamed arguments")
#>     }
#>     attribs <- html_attributes(dots$named)
#>     html(paste0("<img", attribs, " />"))
#> }
img(src = "myimage.png", width = 100, height = 100)
#> <HTML> <img src='myimage.png' width='100' height='100' />
```

21.2.5 Обработка всех тегов

Теперь нам необходимо создать наши служебные функции для всех возможных тегов. Начнем с создания списка тегов HTML:

```
tags <- c("a", "abbr", "address", "article", "aside", "audio",
    "b","bdi", "bdo", "blockquote", "body", "button", "canvas",
    "caption","cite", "code", "colgroup", "data", "datalist",
    "dd", "del","details", "dfn", "div", "dl", "dt", "em",
    "eventsource","fieldset", "figcaption", "figure", "footer",
    "form", "h1", "h2", "h3", "h4", "h5", "h6", "head", "header",
    "hgroup", "html", "i","iframe", "ins", "kbd", "label",
    "legend", "li", "mark", "map","menu", "meter", "nav",
    "noscript", "object", "ol", "optgroup", "option", "output",
    "p", "pre", "progress", "q", "ruby", "rp","rt", "s", "samp",
    "script", "section", "select", "small", "span", "strong",
    "style", "sub", "summary", "sup", "table", "tbody", "td",
    "textarea", "tfoot", "th", "thead", "time", "title", "tr",
    "u", "ul", "var", "video"
)

void_tags <- c("area", "base", "br", "col", "command", "embed",
    "hr", "img", "input", "keygen", "link", "meta", "param",
    "source", "track", "wbr"
)
```

Если внимательно присмотреться к этому списку, можно заметить, что многие теги называются так же, как базовые функции R (`body`, `col`, `q`, `source`, `sub`, `summary`, `table`). Это означает, что мы не должны делать наши функции доступными по умолчанию ни в глобальном окружении, ни в пакете. Вместо этого мы поместим их в список (как в разделе 10.5) и напишем вспомогательную функцию для облегчения их использования при необходимости. Для начала создадим именованный список, содержащий все функции тегов:

```
html_tags <- c(
    tags %>% set_names() %>% map(tag),
    void_tags %>% set_names() %>% map(void_tag)
)
```

Это дает нам явный (но слишком многословный) способ создания разметки HTML:

```
html_tags$p(
    "Some text. ",
    html_tags$b(html_tags$i("some bold italic text")),
    class = "mypara"
)
#> <HTML> <p class='mypara'>Some text. <b><i>some bold italic
#> text</i></b></p>
```

Осталось завершить создание нашего HTML DSL при помощи функции, позволяющей вычислять код в контексте нашего списка. Здесь мы немного неправильно воспользуемся маской данных, передав не датафрейм, а список функций. Это самый легкий способ смешать окружение выполнения аргумента `code` с функциями из списка `html_tags`.

```
with_html <- function(code) {
    code <- enquo(code)
    eval_tidy(code, html_tags)
}
```

В результате мы получим довольно лаконичный API, позволяющий при необходимости создавать разметку HTML без нарушения границ пространств имен.

```
with_html(
    body(
        h1("A heading", id = "first"),
        p("Some text &", b("some bold text.")),
        img(src = "myimg.png", width = 100, height = 100)
    )
)
#> <HTML> <body><h1 id='first'>A heading</h1><p>Some text
#> &amp;<b>some bold text.</b></p><img src='myimg.png'
#> width='100' height='100' /></body>
```

Если вам понадобится внутри функции `with_html()` обратиться к функции R с таким же именем, как у тега HTML, вы можете написать ее полное имя с указанием пакета: `package::function`.

21.2.6 Упражнения

1. Правила экранирования для тегов `<script>` отличаются по причине того, что внутри них содержится код JavaScript, а не разметка HTML. Вместо использования треугольных скобок или амперсандов нам необходимо так экранировать `</script>`, чтобы тег не завершился раньше положенного. Иными словами, код `script("'</script>'")` должен генерировать не такой вывод:

   ```
   <script>'</script>'</script>
   ```

   а такой:

   ```
   <script>'<\/script>'</script>
   ```

   Измените функцию `escape()` таким образом, чтобы она реализовывала такое поведение при установленном дополнительном аргументе `script` в значение `TRUE`.

2. Использование аргумента ... в функциях связано с определенными недостатками. Вы не сможете реализовать проверку аргументов, а в документации к функции информации о них будет совсем мало. Напишите новую функцию, которая при передаче ей именованного списка тегов и имен их атрибутов (как показано ниже) создавала бы функции тегов с именованными аргументами.

   ```
   list(
       a = c("href"),
       img = c("src", "width", "height")
   )
   ```

   Все теги должны принимать атрибуты `class` и `id`.

3. Рассмотрите следующие примеры вызова функции `with_html()` со ссылкой на объекты в окружении. Сработает этот код или завершится ошибкой? Почему? Запустите код, чтобы подтвердить или опровергнуть свои доводы.

   ```
   greeting <- "Hello!"
   with_html(p(greeting))

   p <- function() "p"
   address <- "123 anywhere street"
   with_html(p(address))
   ```

4. В настоящий момент наша функция генерирует разметку HTML без особого форматирования, и читать ее очень неудобно. Как можно было бы изменить функцию `tag()`, чтобы в разметке появились нормальные отступы и форматирование? Возможно, вам понадобится изучить тему блочных и строчных тегов.

21.3 LaTeX

Следующий наш DSL будет преобразовывать выражения R в математические эквиваленты LaTeX. (Это немного похоже на `?plotmath`, но не для графиков, а для текста.) Макропакет *LaTeX* – это общепринятый язык для математиков и статистиков: нотацию LaTeX очень часто используют для представления математических уравнений в виде текста, например для отправки по электронной почте. Поскольку многие отчеты формируются при помощи R и LaTeX, было бы полезно располагать инструментом для автоматического преобразования математических выражений из одного языка в другой.

В данном случае нам будет необходимо преобразовывать как функции, так и имена, так что этот DSL будет превосходить по сложности написанный ранее движок для разметки HTML. Кроме того, нам потребуется создать механизм преобразования по умолчанию, чтобы неизвестные нам символы преобразовывались стандартно. Это означает, что нам будет недостаточно использовать только вычисления – придется попутешествовать и по абстрактным синтаксическим деревьям выражений.

21.3.1 Математика LaTeX

Для начала давайте посмотрим, как именно математические формулы выражаются в LaTeX. Полный стандарт весьма сложен, но, к счастью, очень хорошо документирован (https://en.wikibooks.org/wiki/LaTeX/Mathematics). Наиболее часто используемые команды обладают довольно простой структурой:

- большинство примитивных математических выражений записываются так же, как и в R: `x * y`, `z ^ 5`. Нижние индексы обозначаются с помощью символа подчеркивания (_) (например, `x_1`);
- специальные символы начинаются с `\`: `\pi` = π, `\pm` = $\pm$ и т. д. В LaTeX доступно большое количество символов. Простой поиск по словам *latex math symbols* выдает массу результатов, включая https://www.sunilpatel.co.uk/other-stuff/latex-math-symbols. Существует даже специальный сервис (http://detexify.kirelabs.org/classify.html), позволяющий выполнять поиск символов, которые вы вводите в браузере;
- более сложные функции выглядят примерно так: `\name{arg1}{arg2}`. К примеру, для отображения дроби можно написать `\frac{a}{b}`, а для квадратного корня – `\sqrt{a}`;

- для объединения элементов используются фигурные скобки, {}: т. е. `x ^ a + b` и `x ^ {a + b}`;
- в хорошем математическом наборе символов всегда проводится различие между переменными и функциями. Но без дополнительной информации LaTeX не понимает, является ли запись `f(a * b)` вызовом функции `f` от `a * b` или сокращением от `f * (a * b)`. Если `f` – это функция, вы можете дополнительно указать на это с помощью прямого шрифта: `\textrm{f}(a * b)`. (Здесь `rm` означает *Roman* как противоположность курсиву.)

21.3.2 Цель

Наша цель состоит в использовании описанных выше правил для преобразования выражений R в соответствующее представление в LaTeX. Мы будем решать эту задачу в четыре шага:

- преобразуем известные символы: `pi` → `\pi`;
- остальные символы оставим неизменными: `x` → `x`, `y` → `y`;
- преобразуем известные функции в специальную форму: `sqrt(frac(a, b))` → `\sqrt{\frac{a, b}}`;
- обернем неизвестные функции с помощью `\textrm`: `f(a)` → `\textrm{f}(a)`.

Преобразование мы будем выполнять в порядке, обратном тому, который использовали при создании HTML DSL. Начнем мы с инфраструктуры, которая позволит нам свободно экспериментировать с нашим DSL, а затем вернемся и сгенерируем нужный вывод.

21.3.3 to_math()

Для начала нам понадобится функция-обертка для преобразования выражений R в математические выражения LaTeX. Она должна работать по примеру функции `to_html()` – захватывать невычисленное выражение и вычислять его в особом окружении. Между этими функциями должно быть два важных отличия:

- окружение вычисления не должно быть постоянным, а должно меняться в зависимости от входа. Это необходимо для обработки неизвестных символов и функций;
- мы никогда не производим вычисление в окружении аргумента, поскольку все функции преобразовываем в выражения LaTeX. Для выполнения обычного вычисления пользователю необходимо явным образом использовать оператор `!!`.

В результате мы получим следующую функцию:

```
to_math <- function(x) {
    expr <- enexpr(x)
    out <- eval_bare(expr, latex_env(expr))
```

```
    latex(out)
}

latex <- function(x) structure(x, class = "advr_latex")
print.advr_latex <- function(x) {
    cat("<LATEX> ", x, "\n", sep = "")
}
```

Теперь мы перейдем к функции `latex_env()` – начнем с простого и постепенно усложним ее реализацию.

21.3.4 Известные символы

На первом шаге нам необходимо создать окружение, которое будет конвертировать специальные символы LaTeX, используемые для обозначения букв греческого алфавита, например `pi` в `\pi`. Мы воспользуемся трюком из раздела 20.4.3 для связывания символа `pi` со значением `"\pi"`.

```
greek <- c(
    "alpha", "theta", "tau", "beta", "vartheta", "pi", "upsilon",
    "gamma", "varpi", "phi", "delta", "kappa", "rho",
    "varphi", "epsilon", "lambda", "varrho", "chi", "varepsilon",
    "mu", "sigma", "psi", "zeta", "nu", "varsigma", "omega", "eta",
    "xi", "Gamma", "Lambda", "Sigma", "Psi", "Delta", "Xi",
    "Upsilon", "Omega", "Theta", "Pi", "Phi"
)
greek_list <- set_names(paste0("\\", greek), greek)
greek_env <- as_environment(greek_list)
```

Можем проверить, что получилось:

```
latex_env <- function(expr) {
    greek_env
}

to_math(pi)
#> <LATEX> \pi
to_math(beta)
#> <LATEX> \beta
```

21.3.5 Неизвестные символы

Если символ не входит в греческий алфавит, мы должны оставить его как есть. Это не так просто, поскольку мы не знаем заранее, какие символы будут использованы, и, очевидно, не можем сгенерировать их все. Вместо этого мы воспользуемся подходом, описанным в разделе 18.5: пройдемся по AST для поиска всех символов. В результате получим функцию `all_names_rec()` и вспомогательную функцию `all_names()`:

```
all_names_rec <- function(x) {
    switch_expr(x,
        constant = character(),
        symbol = as.character(x),
        call = flat_map_chr(as.list(x[-1]), all_names)
    )
}

all_names <- function(x) {
    unique(all_names_rec(x))
}

all_names(expr(x + y + f(a, b, c, 10)))
#> [1] "x" "y" "a" "b" "c"
```

Теперь нам нужно взять этот список символов и преобразовать его в окружение, чтобы каждый символ в нем был ассоциирован с конкретным строковым представлением (например, чтобы выражение `eval(quote(x), env)` приводило к получению `"x"`). Снова воспользуемся шаблоном преобразования именованного символьного вектора в список, после чего превратим список в окружение.

```
latex_env <- function(expr) {
    names <- all_names(expr)
    symbol_env <- as_environment(set_names(names))

    symbol_env
}

to_math(x)
#> <LATEX> x
to_math(longvariablename)
#> <LATEX> longvariablename
to_math(pi)
#> <LATEX> pi
```

Все работает, но нам необходимо объединить это с окружением с греческими символами. Поскольку мы хотим, чтобы у греческих символов был приоритет по сравнению с символами по умолчанию (т. е. вызов `to_math(pi)` должен возвращать `"\\pi"`, а не `"pi"`), окружение `symbol_env` должно быть родительским по отношению к `greek_env`. Для реализации этого нам нужно создать копию окружения `greek_env` с новым родителем. В результате получим функцию, которая сможет конвертировать как известные (греческие), так и неизвестные символы.

```
latex_env <- function(expr) {
    # Неизвестные символы
    names <- all_names(expr)
    symbol_env <- as_environment(set_names(names))
```

```
    # Известные символы
    env_clone(greek_env, parent = symbol_env)
}

to_math(x)
#> <LATEX> x
to_math(longvariablename)
#> <LATEX> longvariablename
to_math(pi)
#> <LATEX> \pi
```

21.3.6 Известные функции

После этого нам нужно снабдить наш DSL функциями. Начнем с пары вспомогательных функций, которые облегчат добавление новых унарных и бинарных операторов. Эти функции будут очень простыми: они просто собирают строки.

```
unary_op <- function(left, right) {
    new_function(
        exprs(e1 = ),
        expr(
            paste0(!!left, e1, !!right)
        ),
        caller_env()
    )
}

binary_op <- function(sep) {
    new_function(
        exprs(e1 = , e2 = ),
        expr(
            paste0(e1, !!sep, e2)
        ),
        caller_env()
    )
}

unary_op("\\sqrt{", "}")
#> function (e1)
#> paste0("\\sqrt{", e1, "}")

binary_op("+")
#> function (e1, e2)
#> paste0(e1, "+", e2)
```

С использованием этих вспомогательных функций мы можем выполнить несколько простых преобразований выражений R в LaTeX. Обратите внима-

ние, что с помощью правил лексического поиска в области видимости мы можем легко наделить новыми значениями стандартные функции вроде +, -, * и даже (и {.

```
# Бинарные операторы
f_env <- child_env(
    .parent = empty_env(),
    `+` = binary_op(" + "),
    `-` = binary_op(" - "),
    `*` = binary_op(" * "),
    `/` = binary_op(" / "),
    `^` = binary_op("^"),
    `[` = binary_op("_"),

    # Группировка
    `{` = unary_op("\\left{ ", " \\right}"),
    `(` = unary_op("\\left( ", " \\right)"),
    paste = paste,

    # Другие математические функции
    sqrt = unary_op("\\sqrt{", "}"),
    sin = unary_op("\\sin(", ")"),
    log = unary_op("\\log(", ")"),
    abs = unary_op("\\left| ", "\\right| "),
    frac = function(a, b) {
        paste0("\\frac{", a, "}{", b, "}")
    },

    # Специальные маркеры
    hat = unary_op("\\hat{", "}"),
    tilde = unary_op("\\tilde{", "}")
)
```

Снова изменим функцию `latex_env()`, включив это окружение. Оно должно быть последним окружением, в котором R будет выполнять поиск имен, чтобы выражения вроде `sin(sin)` работали.

```
latex_env <- function(expr) {
    # Известные функции
    f_env

    # Символы по умолчанию
    names <- all_names(expr)
    symbol_env <- as_environment(set_names(names), parent = f_env)

    # Известные символы
    greek_env <- env_clone(greek_env, parent = symbol_env)

    greek_env
}
```

```
to_math(sin(x + pi))
#> <LATEX> \sin(x + \pi)
to_math(log(x[i]^2))
#> <LATEX> \log(x_i^2)
to_math(sin(sin))
#> <LATEX> \sin(sin)
```

21.3.7 Неизвестные функции

Наконец, добавим умолчания для функций, о которых нам пока неизвестно. Мы не можем знать заранее, какие неизвестные функции будут использоваться, так что снова пойдем по AST, выполняя поиск:

```
all_calls_rec <- function(x) {
    switch_expr(x,
        constant = ,
        symbol = character(),
        call = {
            fname <- as.character(x[[1]])
            children <- flat_map_chr(as.list(x[-1]), all_calls)
            c(fname, children)
        }
    )
}
all_calls <- function(x) {
    unique(all_calls_rec(x))
}

all_calls(expr(f(g + b, c, d(a))))
#> [1] "f" "+" "d"
```

Нам понадобится *замыкание* (closure), которое будет генерировать функции для каждого неизвестного вызова:

```
unknown_op <- function(op) {
    new_function(
        exprs(... = ),
        expr({
            contents <- paste(..., collapse = ", ")
            paste0(!!paste0("\\mathrm{", op, "}("), contents, ")")
        })
    )
}
unknown_op("foo")
#> function (...)
#> {
#>     contents <- paste(..., collapse = ", ")
#>     paste0("\\mathrm{foo}(", contents, ")")
```

```
#> }
#> <environment: 0x7f98e739d128>
```

И снова обновляем функцию `latex_env()`:

```
latex_env <- function(expr) {
    calls <- all_calls(expr)
    call_list <- map(set_names(calls), unknown_op)
    call_env <- as_environment(call_list)

    # Известные функции
    f_env <- env_clone(f_env, call_env)

    # Символы по умолчанию
    names <- all_names(expr)
    symbol_env <- as_environment(set_names(names), parent = f_env)

    # Известные символы
    greek_env <- env_clone(greek_env, parent = symbol_env)
    greek_env
}
```

Теперь все наши первоначальные требования удовлетворены:

```
to_math(sin(pi) + f(a))
#> <LATEX> \sin(\pi) + \mathrm{f}(a)
```

Конечно, вы могли бы развить эту идею и реализовать преобразование типов математических выражений, но никакие дополнительные инструменты метапрограммирования вам для этого не понадобятся.

21.3.8 Упражнения

1. Добавьте экранирование. В число специальных символов, требующих постановки перед ними обратного слеша, должны входить символы `\`, `$` и `%`. Как и в случае с HTML, убедитесь, что в результате вы избежите ловушки с двойным экранированием. Вам понадобится создать небольшой класс S3 и использовать его в функциональных операторах. Это также позволит вам при необходимости встраивать произвольные выражения LaTeX.
2. Дополните DSL поддержкой всех функций, реализованных в функции `plotmath`.

Часть V

Техники

Введение

В заключительных четырех главах книги мы обсудим две важнейшие техники в программировании: поиск и исправление ошибок, а также фиксацию и устранение проблем с производительностью. Инструменты, которые мы будем использовать для повышения эффективности работы кода, имеют огромную важность, поскольку язык R не славится своей скоростью. И это не случайно, ведь он изначально разрабатывался для облегчения процесса интерактивного анализа данных, а не с целью ускорить выполнение вычислений до предела. Несмотря на некую свою задумчивость в сравнении с другими языками программирования, для решения большей части задач скорости R вполне достаточно. В этих главах мы сосредоточимся на тех ситуациях, когда производительности R действительно не хватает, и узнаем, как можно повысить эффективность выполнения кода R или перенести часть вычислений в язык программирования C++, изначально предназначенный для высокоэффективных вычислений.

1. В главе 22 мы будем говорить о процессе отладки кода, поскольку поиск источника ошибок зачастую представляет собой весьма утомительный процесс. К счастью, R располагает набором достаточно эффективных средств отладки, благодаря которому и вкупе с четко выстроенной стратегией вы сможете быстро и безболезненно находить источники большинства возникающих проблем.
2. Глава 23 будет посвящена измерению производительности кода.
3. В главе 24 мы покажем, какие существуют способы для повышения эффективности выполнения кода на языке R.

22

Отладка

22.1 Введение

Что вы делаете, когда при выполнении кода на R появляется неожиданная ошибка? Какие инструменты есть в вашем распоряжении для поиска и устранения ошибок? В этой главе мы обратимся к искусству отладки кода – начнем с базовой стратегии, после чего перейдем к рассмотрению конкретных инструментов.

Здесь я продемонстрирую инструменты, присутствующие в самом R и в среде разработки RStudio. Лично я рекомендую по возможности пользоваться средствами RStudio, но в своем обзоре также не обойду вниманием и эквиваленты, работающие в любой среде. Официальная документация по отладке в RStudio располагается по адресу https://support.posit.co/hc/en-us/articles/205612627-Debugging-with-RStudio и всегда отражает информацию об актуальных средствах, имеющихся в последних версиях RStudio.

Примечание. Не стоит прибегать к этим инструментам отладки при написании *новых* функций. Если вы понимаете, что вам часто приходится обращаться к отладке в процессе разработки кода, пересмотрите свой подход. Вместо того чтобы пытаться написать с нуля большую функцию, лучше сначала в интерактивном режиме отработать все ее составляющие части. В этом случае вы сможете сразу фиксировать и исправлять возникающие ошибки, для этого вам не потребуются полноценные инструменты отладки кода.

Структура главы

- В разделе 22.2 мы обсудим базовую стратегию поиска и исправления ошибок.
- Раздел 22.3 будет посвящен функции `traceback()`, помогающей точно определить, где именно возникла ошибка.
- В разделе 22.4 мы увидим, как можно ставить выполнение функции на паузу и запускать окружение, в котором в интерактивном режиме исследовать, что произошло.

- В разделе 22.5 мы обсудим сложности с отладкой при запуске кода в неинтерактивном режиме.
- В разделе 22.6 мы рассмотрим ряд проблем, не связанных с ошибками, но требующих отладки.

22.2 Общий подход

Нахождение ошибок – это процесс, связанный с подтверждением истинности множества явлений, которые вы считали истинными, вплоть до нахождения одного, не являющегося истиной.

— Норм Матлофф

Поиск источника проблем всегда связан с определенными трудностями. Большинство ошибок сложно обнаружить именно потому, что если бы это было сделать легко, мы бы не допустили их при написании кода. Здесь помогает четко выверенная стратегия. Ниже я обозначу четыре шага, которые считаю полезными в деле обнаружения и устранения ошибок.

1. **Погуглите!**
   Всякий раз, когда вы видите сообщение об ошибке, начните с поиска информации в интернете. Если вам повезет, вы обнаружите, что имеете дело с распространенной ошибкой, которая давно решена. При поиске старайтесь обходиться общими формулировками и избегать упоминания имен переменных или значений, специфичных для вашего конкретного кода.
   Вы можете автоматизировать этот процесс при помощи специальных пакетов `errorist` [Баламута (Balamuta), 2018a] и `searcher` [Баламута, 2018b]. За подробностями по ним вы можете обратиться на сайты этих пакетов.
2. **Сделайте ошибку воспроизводимой.**
   Для поиска источника проблемы вам придется запускать код множество раз, проверяя те или иные догадки и гипотезы. Для облегчения этого процесса вам стоит потратить некоторое время, чтобы сделать ошибку легковоспроизводимой.
   Начните с создания *воспроизводимого примера* (***repr****oducible* ***ex****ample*, или *reprex*), который мы упоминали в разделе 1.7. После этого минимизируйте содержимое примера, избавившись от ненужного кода и упростив данные. В процессе вы можете обнаружить, что определенные входные данные не приводят к возникновению ошибки. Отметьте этот момент, он поможет в определении источника проблемы.
   Если вы используете механизмы автоматизации тестирования, самое время создать такой автоматизированный пример. Если существующего покрытия тестами недостаточно, воспользуйтесь возможностью добавить некие сторонние тесты, чтобы убедиться, что безошибочное поведение сохраняется. Это снизит шанс возникновения новых ошибок.

3. **Определите, где именно возникает ошибка.**
   Если вам повезет, использование одного из инструментов, о которых мы будем говорить далее, приведет к обнаружению конкретной строки кода, в которой возникает ошибка. Но чаще всего вам придется поломать голову над проблемой. Здесь вполне применим научный подход. Вы можете выдвигать гипотезы, проводить эксперименты для их проверки и фиксировать результаты. Может показаться, что это слишком сложно, но в конечном счете применение системного подхода сэкономит вам время. Я зачастую полагаюсь на свою интуицию при поиске ошибок («Ой, да это просто ошибка завышения/занижения на единицу, надо взять и вычесть тут единицу!»), когда, по уму, нужно применить системный подход.
   Если ничто не помогает, нужно попросить помощи у товарищей. Пройдя путь, описанный выше, вы получите небольшой фрагмент кода, которым легко будет поделиться с другими. Это облегчит им понимание сути вашей проблемы и ускорит процесс нахождения решения.
4. **Исправьте ошибку и протестируйте код.**
   После обнаружения источника ошибки вам необходимо понять, как ее исправить, и убедиться, что это исправление действительно поможет. Опять же, очень полезно располагать какими-то автоматизированными тестами. Это позволяет убедиться не только в том, что вы действительно решили проблему, но и в том, что в процессе не наплодили новых ошибок. В отсутствие автоматизированных тестов тщательно записывайте все результаты проверок и обязательно прогоните код с входными параметрами, которые ранее порождали ошибку.

22.3 Обнаружение ошибок

После того как вы сделали проблему воспроизводимой, следующим шагом должно быть обнаружение конкретного места возникновения ошибки. Наиболее важным инструментом в этом процессе является функция `traceback()`, показывающая последовательность вызовов (также называемую *стеком вызовов* (call stack), о котором мы упоминали в разделе 7.5), приведших к появлению ошибки.

В следующем простом примере функция `f()` вызывает функцию `g()`, которая вызывает функцию `h()`, та, свою очередь, обращается к функции `i()`, в которой выполняется проверка на то, что аргумент является числовым:

```
f <- function(a) g(a)
g <- function(b) h(b)
h <- function(c) i(c)
i <- function(d) {
    if (!is.numeric(d)) {
```

```
        stop("`d` must be numeric", call. = FALSE)
    }
    d + 10
}
```

Если мы выполним вызов `f("a")` в RStudio, то увидим текст, показанный на рис. 22.1.

Рис. 22.1 Вывод ошибки в RStudio

Справа от вывода вы можете видеть две ссылки: **Show Traceback** и **Rerun with Debug**. Если нажать на первую, вы увидите вывод, представленный на рис. 22.2.

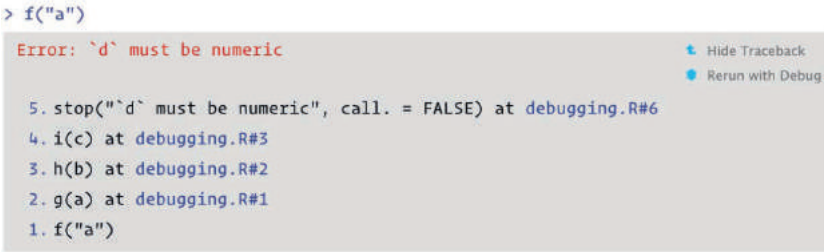


Рис. 22.2 Запуск трассировки в RStudio

Если вы не используете интерактивную среду разработки RStudio, то можете для получения той же информации, но с менее привлекательным форматированием, вызвать функцию `traceback()`:

```
traceback()
#> 5: stop("`d` must be numeric", call. = FALSE) at debugging.R#6
#> 4: i(c) at debugging.R#3
#> 3: h(b) at debugging.R#2
#> 2: g(a) at debugging.R#1
#> 1: f("a")
```

Примечание. Вывод функции `traceback()` необходимо читать снизу вверх: от функции `f()` до функции `i()`, в которой и возникла ошибка. Если вы выполняете код, который загрузили в R с помощью функции `source()`, то при трассировке также увидите расположение функции в формате `filename.r#linenumber`. В RStudio вы сможете щелкнуть по ссылке и перейти к соответствующей строке кода в редакторе.

22.3.1 Отложенные вычисления

К недостаткам функции `traceback()` можно отнести то, что она всегда приводит дерево вызовов к линейному виду, что может сбивать с толку при наличии отложенных, или ленивых, вычислений (см. раздел 7.5.2). Давайте обратимся к следующему примеру, в котором ошибка возникает при вычислении первого аргумента функции `f()`:

```
j <- function() k()
k <- function() stop("Oops!", call. = FALSE)
f(j())
#> Error: Oops!

traceback()
#> 7: stop("Oops!") at #1
#> 6: k() at #1
#> 5: j() at debugging.R#1
#> 4: i(c) at debugging.R#3
#> 3: h(b) at debugging.R#2
#> 2: g(a) at debugging.R#1
#> 1: f(j())
```

Вы можете воспользоваться функциями `rlang::with_abort()` и `rlang::last_trace()`, чтобы увидеть дерево вызовов. Мне кажется, что просмотр дерева вызовов значительно облегчает процесс обнаружения источника ошибок. Взгляните на последнюю ветвь дерева, и вы увидите, что ошибка возникла при вызове функции `k()` из функции `j()`.

```
rlang::with_abort(f(j()))
#> Error: Oops!
rlang::last_trace()
#>     ▉
#>  1. ├─rlang::with_abort(f(j()))
#>  2. │ └─base::withCallingHandlers(...)
#>  3. ├─global::f(j())
#>  4. │ └─global::g(a) debugging.R:1:5
#>  5. │   └─global::h(b) debugging.R:2:5
#>  6. │     └─global::i(c) debugging.R:3:5
#>  7. └─global::j() debugging.R:1:5
#>  8.   └─global::k()
```

Примечание. Вывод функции `rlang::last_trace()` упорядочен обратным образом по сравнению с функцией `traceback()`. Мы вернемся к этому случаю в разделе 22.4.2.4.

22.4 Интерактивный отладчик

Иногда достаточно знать точное место возникновения ошибки, чтобы отследить ее появление и устранить ее. Но зачастую для избавления от ошибки вам потребуется дополнительная информация, которую легче всего получить с помощью интерактивного отладчика, позволяющего ставить выполнение функции на паузу и интерактивно исследовать ее состояние.

При использовании RStudio проще всего активировать интерактивный отладчик с помощью кнопки **Rerun with Debug**. Это приведет к повторному запуску команды, приведшей к возникновению ошибки, с остановкой на этом моменте. Также вы можете воспользоваться функцией `browser()` в том месте, где хотите остановиться, и повторно запустить функцию. К примеру, так мы бы могли вставить вызов функции `browser()` в функцию `g()`:

```
g <- function(b) {
    browser()
    h(b)
}
f(10)
```

`browser()` – это обычный вызов функции, а это значит, что вы можете использовать его в условных конструкциях, как показано ниже:

```
g <- function(b) {
    if (b < 0) {
        browser()
    }
    h(b)
}
```

В обоих случаях вы окажетесь в интерактивном окружении *внутри* функции, где сможете запускать любой произвольный код для исследования состояния функции. О том, что вы находитесь в режиме интерактивной отладки, вы поймете благодаря особой подсказке:

```
Browse[1]>
```

В RStudio соответствующий код вы увидите в редакторе (с подсветкой строки кода, которая будет выполнена следующей), объекты в текущем окружении – на панели **Environment**, а стек вызовов – на панели **Traceback**.

22.4.1 Команды browser()

Помимо возможности запускать обычный код R, функция `browser()` предоставляет несколько полезных команд. Воспользоваться ими можно либо по-

средством текстового ввода, либо с помощью кнопки на панели инструментов RStudio, показанной на рис. 22.3.

Рис. 22.3 Панель отладки в RStudio

- **Next**, **n**: выполняет следующий шаг в функции. Если у вас есть переменная с именем `n`, вам понадобится написать `print(n)` для вывода ее значения;
- **Step into**, **s** или : работает подобно команде **Next**, но если следующим шагом будет располагаться вызов функции, выполнение перейдет в нее, что позволит вам исследовать ее поведение;
- **Finish**, **f** или : завершает выполнение текущего цикла или функции;
- **Continue**, **c**: завершает процесс интерактивной отладки и запускает обычное выполнение функции с текущей позиции. Это бывает полезно, если вы исправили ошибочное состояние и хотите проверить, успешно ли выполнится функция;
- **Stop**, **Q**: завершает процесс отладки, прерывает выполнение функции и возвращает глобальное рабочее пространство. Используйте этот выбор, если вы поняли, как исправить ошибку, и готовы запустить функцию заново.

Есть еще две менее популярные команды, не вынесенные на панель инструментов:

- **Enter**: повторяет предыдущую команду. Я зачастую случайно активирую эту опцию, так что предпочитаю отключать ее при помощи команды `options(browserNLdisabled = TRUE)`;
- **where**: выводит стек трассировки активных вызовов (интерактивный эквивалент `traceback`).

22.4.2 Альтернативы

Существует три альтернативы использованию функции `browser()`: установка точек останова в RStudio, команда `option(error = recover)` и использование функции `debug()` и других подобных функций.

22.4.2.1 Точки останова

В RStudio вы можете установить *точку останова* (breakpoint), щелкнув мышью слева от номера нужной строки кода или нажав сочетание клавиш **Shift**+**F9**. Точки останова ведут себя подобно функции `browser()`, но их легче устанавливать (один щелчок вместо девяти нажатий клавиш), и с ними исключается возможность случайно оставить функции `browser()` в коде по его готовности. В то же время точки останова обладают двумя небольшими недостатками:

- есть определенные редко встречающиеся ситуации, когда точки останова работать не будут. За подробностями можно обратиться к статье по адресу https://support.posit.co/hc/en-us/articles/200534337-Breakpoint-Troubleshooting;
- на этапе написания книги RStudio не поддерживал условные точки останова.

22.4.2.2 recover()

Еще один способ воспользоваться функцией `browser()` состоит в применении команды `options(error = recover)`. В этом случае при появлении ошибки будет выводиться интерактивная подсказка с отображением трассировки и возможностью выполнения интерактивной отладки внутри конкретного *фрейма* (frame):

```
options(error = recover)
f("x")
#> Error: `d` must be numeric
#>
#> Enter a frame number, or 0 to exit
#>
#> 1: f("x")
#> 2: debugging.R#1: g(a)
#> 3: debugging.R#2: h(b)
#> 4: debugging.R#3: i(c)
#>
#> Selection:
```

К обычной процедуре перехвата ошибок можно вернуться с помощью команды `options(error = NULL)`.

22.4.2.3 debug()

Другой подход состоит в вызове одной из функций, которые сами вставляют вызовы функции `browser()` за вас:

- `debug()` вставляет функцию `browser()` в первую строку указанной функции. Удаляет ее функция `undebug()`. Также вы можете воспользоваться функцией `debugonce()`, чтобы использовать функцию `browser()` только при следующем запуске;
- функция `utils::setBreakpoint()` работает похожим образом, но вместо приема имени функции она принимает имя файла и строку кода и ищет нужную функцию за вас.

Обе эти функции представляют собой частные случаи функции `trace()`, которая может вставлять произвольный код в любое место существующей функции. Сама функция `trace()` может пригодиться при необходимости выполнить отладку кода, к которому у вас нет доступа. Для удаления трасси-

ровки воспользуйтесь функцией `untrace()`. Вы можете выполнять только одну трассировку на функцию, но в ней могут вызываться разные функции.

22.4.2.4 Стек вызовов

К сожалению, стеки вызовов, которые выводят функции `traceback()`, `browser()`, `where` и `recover()`, не согласуются друг с другом. В табл. 22.1 показано, как три разных инструмента отображают стек вызовов для простого вложенного набора вызовов. Разница порядковых номеров иллюстрирует разный порядок отображения стеков.

Таблица 22.1 Стеки вызовов с помощью разных инструментов

`traceback()`	`where`	`recover()`	Функции `rlang`
`5: stop("...")`			
`4: i(c)`	`where 1: i(c)`	`1: f()`	`1. └─global::f(10)`
`3: h(b)`	`where 2: h(b)`	`2: g(a)`	`2.  └─global::g(a)`
`2: g(a)`	`where 3: g(a)`	`3: h(b)`	`3.  └─global::h(b)`
`1: f("a")`	`where 4: f("a")`	`4: i("a")`	`4.  └─global::i("a")`

В RStudio вызовы отображаются в том же порядке, что и в функции `traceback()`. Функции из пакета `rlang` используют ту же нумерацию и порядок, что и функция `recover()`, но с соответствующими отступами для иллюстрации иерархии.

22.4.3 Скомпилированный код

Также вы можете использовать интерактивный отладчик (*gdb* или *lldb*) для скомпилированного кода (типа C или C++). К сожалению, эта тема выходит за рамки данной книги, но есть немало ресурсов, на которых вы можете почерпнуть полезную информацию:

- https://r-pkgs.org;
- https://github.com/wch/r-debug/blob/master/debugging-r.md;
- http://kevinushey.github.io/blog/2015/04/05/debugging-with-valgrind;
- https://www.jimhester.com/talk/2019-crug-debugging.

22.5 Неинтерактивная отладка

Процесс отладки серьезно осложняется, когда у вас нет возможности запустить код интерактивно, обычно по причине того, что он является составной частью некого определенного автоматизированного конвейера (вероятно, работающего на другом компьютере), или из-за того, что ошибка не возникает при интерактивном запуске. Иногда это сильно раздражает!

В этом разделе мы обсудим некоторые полезные инструменты, но при их использовании не забывайте о базовой стратегии отладки, описанной

в разделе 22.2. Если вы не можете запустить код интерактивно, придется потратить некоторое время на то, чтобы уменьшить масштаб проблемного кода для максимально быстрого решения проблемы. Иногда бывает полезно воспользоваться функцией `callr::r(f, list(1, 2))`. В результате будет вызвана функция `f(1, 2)` в новой сессии, что может помочь в воспроизведении проблемы.

Также вам необходимо дважды проверить следующие распространенные источники проблем:

- отличается ли глобальное окружение? Может, вы загрузили разные пакеты? Может, объекты, оставшиеся от предыдущих сессий, являются причинами различий?
- отличается ли рабочая директория?
- отличается ли переменная окружения `PATH`, определяющая местоположение внешних команд, таких как `git`?
- отличается ли переменная окружения `R_LIBS`, отвечающая за поиск пакетов функцией `library()`?

22.5.1 dump.frames()

Функция `dump.frames()` аналогична функции `recover()` применительно к неинтерактивной отладке кода. Она сохраняет файл `last.dump.rda` в рабочей директории. Позже, в интерактивной сессии, вы можете воспользоваться функциями `load("last.dump.rda")`; `debugger()` для входа в интерактивный режим отладки с тем же интерфейсом, что и у `recover()`. Такой подход позволяет немного сжульничать и интерактивно отлаживать код, который был запущен не интерактивно.

```
# В пакетном режиме R ----
dump_and_quit <- function() {
    # Сохраняем отладку в файл last.dump.rda
    dump.frames(to.file = TRUE)
    # Завершаем R со статусом ошибки
    q(status = 1)
}
options(error = dump_and_quit)

# Позже в интерактивной сессии ----
load("last.dump.rda")
debugger()
```

22.5.2 Вывод отладочной информации

Если функция `dump.frames()` не помогает, обходным путем может быть вывод отладочной информации на экран. В этом случае вы вставляете в код множество инструкций для вывода переменных, необходимых для отладки, и отслеживаете их значения. Этот подход довольно медленный и примитив-

ный, но он всегда работает, так что нужно иметь его в своем арсенале на случай, когда трассировки вам недоступны. Начните с вывода общих маркеров, после чего спускайтесь до более гранулярных при определении источника проблемы.

```
f <- function(a) {
    cat("f()\n")
    g(a)
}
g <- function(b) {
    cat("g()\n")
    cat("b =", b, "\n")
    h(b)
}
h <- function(c) {
    cat("i()\n")
    i(c)
}

f(10)
#> f()
#> g()
#> b = 10
#> i()
#> [1] 20
```

Этот подход зачастую спасает при отладке скомпилированного кода, поскольку компилятор нередко настолько сильно меняет ваш код, что определить источник проблемы становится трудно даже при интерактивной отладке.

22.5.3 RMarkdown

Отладка кода внутри файлов *RMarkdown* требует применения особых инструментов. Во-первых, если вы работаете с файлом в RStudio, переключитесь на вызов функции `rmarkdown::render("path/to/file.Rmd")`. Это позволит запустить код в текущей сессии, что облегчит процесс отладки. Если одно это поможет избавиться от ошибки, нужно смотреть в сторону различий в окружениях.

Если проблема осталась, воспользуйтесь своими навыками интерактивной отладки кода. Какой бы способ вы при этом ни применили, вам потребуется сделать дополнительный шаг: в обработчике ошибок нужно вызвать функцию `sink()`. Это позволит удалить настройки по умолчанию, которые использует пакет `knitr` для захвата вывода, и гарантирует вывод результатов на консоль. К примеру, чтобы воспользоваться функцией `recover()` с RMarkdown, вам необходимо поместить следующий код в блок настройки:

```
options(error = function() {
    sink()
```

```
    recover()
})
```

Это приведет к появлению предупреждения о невозможности удалить настройки, когда knitr завершит работу. Вы можете безопасно проигнорировать его.

Если вы просто хотите выполнить трассировку, самым простым способом будет применить функцию rlang::trace_back(), воспользовавшись преимуществами опции rlang_trace_top_env, как показано ниже. Это гарантирует вам, что вы будете видеть трассировку только своего кода, а не всех функций, вызванных в RMarkdown и knitr.

```
options(rlang_trace_top_env = rlang::current_env())
options(error = function() {
    sink()
    print(rlang::trace_back(bottom = sys.frame(-1)), simplify = "none")
})
```

22.6 Проблемы, не связанные с ошибками

Функции могут завершаться неудачей и без выброса ошибки. Ситуации могут быть следующие:

- функция может сгенерировать неожиданное *предупреждение* (warning). Самым простым способом отследить предупреждения является их преобразование в ошибки с помощью опции options(warn = 2) и использование стека вызовов посредством стандартных инструментов отладки вроде doWithOneRestart() и withOneRestart(). После этого вы увидите несколько дополнительных вызовов withRestarts() и .signalSimpleWarning(). Игнорируйте их: это внутренние функции, используемые для превращения предупреждений в ошибки;
- функция может сгенерировать неожиданное *сообщение* (message). Для превращения сообщений в ошибки можно воспользоваться функцией rlang::with_abort():

```
f <- function() g()
g <- function() message("Hi!")
f()
#> Hi!

rlang::with_abort(f(), "message")
#> Error: Hi!
rlang::last_trace()
#>     ▉
#>  1. ├─rlang::with_abort(f(), "message")
#>  2. │ └─base::withCallingHandlers(...)
```

```
#>  3. └─global::f()
#>  4.   └─global::g()
```

- функция может никогда не вернуться. Это трудная ситуация для автоматической отладки, но иногда может помочь прерывание работы функции и поиск по трассировке функции `traceback()`. Если нет, вы можете воспользоваться выводом на экран отладочных сообщений, как описывалось в разделе 22.5.2;
- в худшем из возможных сценариев ваш код может приводить к аварийному завершению работы R, что не позволит использовать интерактивную отладку. Это свидетельствует об ошибке в скомпилированном коде (C или C++). В этом случае вам следует перейти по ссылкам из раздела 22.4.3 и поучиться использовать интерактивный отладчик в C (или вставить множество отладочных выводов). Если ошибка возникает в пакете или в базовом R, вам необходимо связаться с поддержкой. Здесь также будет нелишним создать небольшой воспроизводимый пример (см. раздел 1.7) для помощи разработчикам.

23

Измерение производительности

23.1 Введение

> *Программисты тратят огромное время, размышляя и беспокоясь о скорости работы некритических частей своих программ, и их попытки повысить эффективность кода зачастую оказывают негативное влияние, когда речь идет об отладке и поддержке.*
>
> *– Дональд Кнут*

Перед тем как ускорять выполнение программы, необходимо сначала определиться с тем, что замедляет ее работу. На словах это выглядит просто, но на деле все не так тривиально. Даже опытнейшие программисты порой тратят много времени на поиск узких мест в своем коде. Таким образом, вместо того чтобы полагаться на интуицию, нужно использовать технику *профилирования* (profiling) кода, т. е. измерять время выполнения каждой его строки с использованием реалистичных входных данных.

После нахождения узких мест вам необходимо заняться поиском альтернативных решений, которые позволят с меньшими затратами добиться тех же результатов. В главе 24 мы рассмотрим разные способы ускорения кода, но сначала вам нужно научиться проводить эталонное микротестирование, чтобы точно определять разницу в производительности.

Структура главы

- В разделе 23.2 мы рассмотрим варианты использования техники профилирования для оценки того, что именно замедляет работу кода.
- В разделе 23.3 мы узнаем, как использовать эталонное микротестирование для выбора наиболее оптимальной реализации кода.

Требования

В данной главе мы будем пользоваться пакетами `profvis` (https://rstudio.github.io/profvis) и `bench` (https://bench.r-lib.org) для выполнения микротестирования.

```
library(profvis)
library(bench)
```

23.2 Профилирование

Во всех без исключения языках программирования первостепенным инструментом для анализа быстродействия кода является *профайлер* (profiler). Существует несколько типов профайлеров, но в R используется простейший из них, называемый *семплирующим*, или *статистическим*, профайлером (statistical profiler). Этот тип профайлера останавливает выполнение кода каждые несколько миллисекунд и фиксирует стек вызовов (т. е. информацию о том, какая функция в данный момент выполняется, какая функция ее вызвала и т. д.). Рассмотрим пример с функцией `f()`, показанной ниже:

```
f <- function() {
    pause(0.1)
    g()
    h()
}
g <- function() {
    pause(0.1)
    h()
}
h <- function() {
    pause(0.1)
}
```

(Я использую функцию `profvis::pause()` вместо `Sys.sleep()`, поскольку последняя не отображается в выводе профайлера из-за того, что, как видит R, она не использует вычислительное время.)

Если бы мы профилировали вызов `f()` с остановками кода каждые 0,1 с, мы бы увидели следующий результат:

```
"pause" "f"
"pause" "f" "g"
"pause" "f" "g" "h"
"pause" "f" "h"
```

Каждая строка здесь представляет один такт работы профайлера (в нашем случае это 0,1 с), а запись вызовов функций производится справа налево. В первой строке мы видим, что функция `f()` вызвала функцию `pause()`. Далее мы видим, что код на протяжении 0,1 с выполнял функцию `f()`, затем 0,2 с ушло на выполнение функции `g()`, потом 0,1 с на выполнение функции `h()`.

При профилировании вызова `f()` с помощью функции `utils::Rprof()`, как показано ниже, мы вряд ли получили бы такие точные результаты.

```
tmp <- tempfile()
Rprof(tmp, interval = 0.1)
```

```
f()
Rprof(NULL)
writeLines(readLines(tmp))
#> sample.interval=100000
#> "pause" "g" "f"
#> "pause" "h" "g" "f"
#> "pause" "h" "f"
```

Причина в том, что все профайлеры вынуждены балансировать между точностью оценки и производительностью. Компромисс, достигаемый при использовании семплирующего профайлера, оказывает лишь незначительное влияние на производительность и в целом носит случайный характер из-за существующей изменчивости как в отношении точности таймера, так и в плане времени, необходимого для выполнения каждой операции. В результате этого при каждом очередном запуске профайлера вы будете получать различающиеся результаты. К счастью, эта изменчивость главным образом затрагивает функции, не требующие большого времени на выполнение, а именно такие функции интересуют нас меньше всего.

23.2.1 Визуализация профилирования

По умолчанию разрешающая способность профайлера достаточно мала, так что даже если ваша функция выполняется всего несколько секунд, по ней будут сгенерированы сотни выборок. Как вы понимаете, такой объем данных просматривать очень сложно, так что вместо пакета `utils::Rprof()` мы будем пользоваться для визуализации агрегированных данных пакетом `profvis`. Этот пакет также позволяет связывать информацию о профилировании с исходным кодом, что облегчает понимание того, какие фрагменты нуждаются в изменениях. Если вам не поможет пакет `profvis`, вы можете попробовать воспользоваться функцией `utils::summaryRprof()` или пакетом `proftools` [Тьерни (Tierney) и Жаржур (Jarjour), 2016].

Существует два способа использования пакета `profvis`:

- из меню **Profile** в RStudio;
- с помощью функции `profvis::profvis()`. Я рекомендую хранить код в отдельном файле и загружать его с помощью функции `source()`. Это обеспечит наилучшее взаимодействие между профилирующими данными и исходным кодом.

```
source("profiling-example.R")
profvis(f())
```

После завершения работы `profvis` будет открыт интерактивный документ HTML, в котором вы сможете изучить результаты. При этом вам будут доступны две панели, что видно на рис. 23.1.

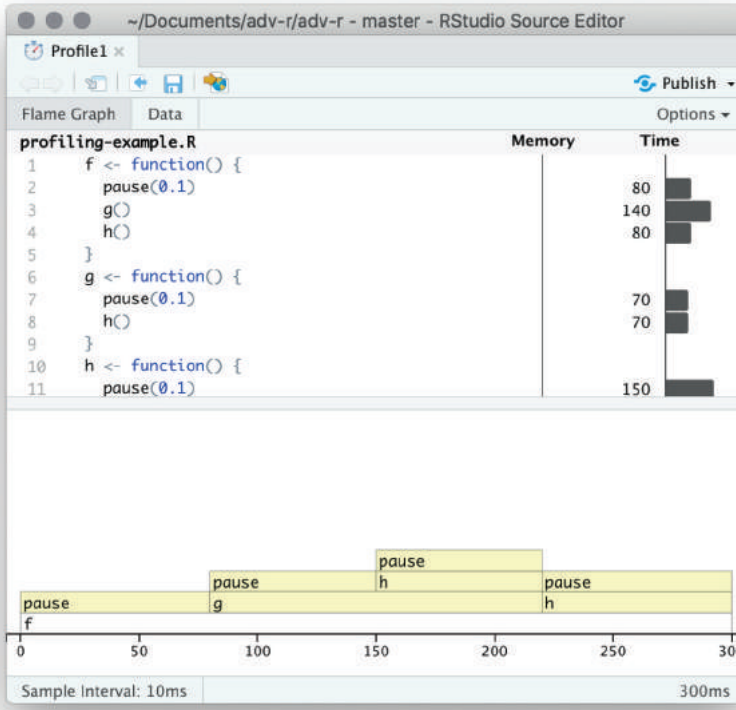


Рис. 23.1 Результаты профилирования с помощью пакета `profvis`: вверху – исходный код, внизу – *график пламени* (flame graph)

Верхняя область отведена под исходный код, справа от которого отображается информация о расходуемой памяти и времени выполнения в построчном режиме. Пока мы сосредоточимся на времени выполнения, а к памяти вернемся позже. Такой тип отображения наглядно дает понять, где могут находиться узкие места в коде, но не всегда позволяет точно узнать истинную причину их образования. В данном примере мы видим, что функция `h()` выполняется 150 мс, что вдвое больше, чем функция `g()`. Причина этого не в том, что сама функция более медленная, а в том, что она вызывалась в два раза чаще.

На нижней панели отображается так называемый *график пламени* (flame graph), показывающий полный стек вызовов. С помощью него можно увидеть всю последовательность вызовов. В частности, мы видим, что функция `h()` вызывалась в двух местах. При наведении на определенный вызов мышью вы увидите дополнительную информацию, и одновременно с этим будет подсвечена соответствующая строка кода, как показано на рис. 23.2.

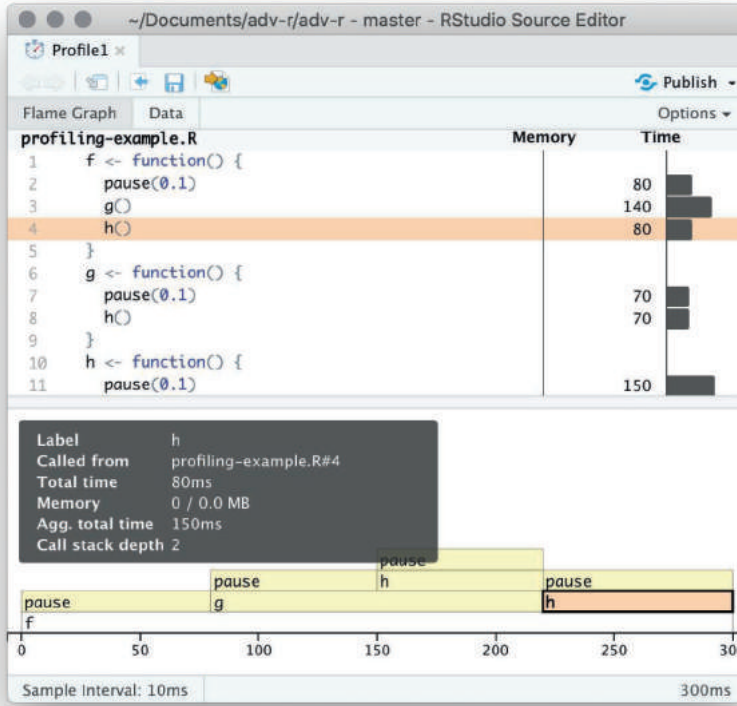


Рис. 23.2 Наведение мышью на вызов функции приводит к выделению соответствующей строки кода и отображению дополнительной информации о производительности

Также вы можете переключиться на панель **Data**, где, как показано на рис. 23.3, увидите интерактивное дерево информации о быстродействии. По сути, это тот же график пламени, но развернутый на 90°. Пользоваться им удобно при наличии в вашем стеке большого количества вложенных вызовов, поскольку здесь вы можете углубляться в детализацию нужных вам компонентов.

23.2.2 Профилирование памяти

На графике пламени может присутствовать особый маркер **<GC>**, отсутствующий в вашем исходном коде. Этот маркер иллюстрирует работу *сборщика мусора* (garbage collector). Если **<GC>** занимает много времени, обычно это означает, что в вашем коде создается излишнее количество объектов с коротким временем жизни. Давайте для примера рассмотрим следующий фрагмент кода:

```
x <- integer()
for (i in 1:1e4) {
    x <- c(x, i)
}
```

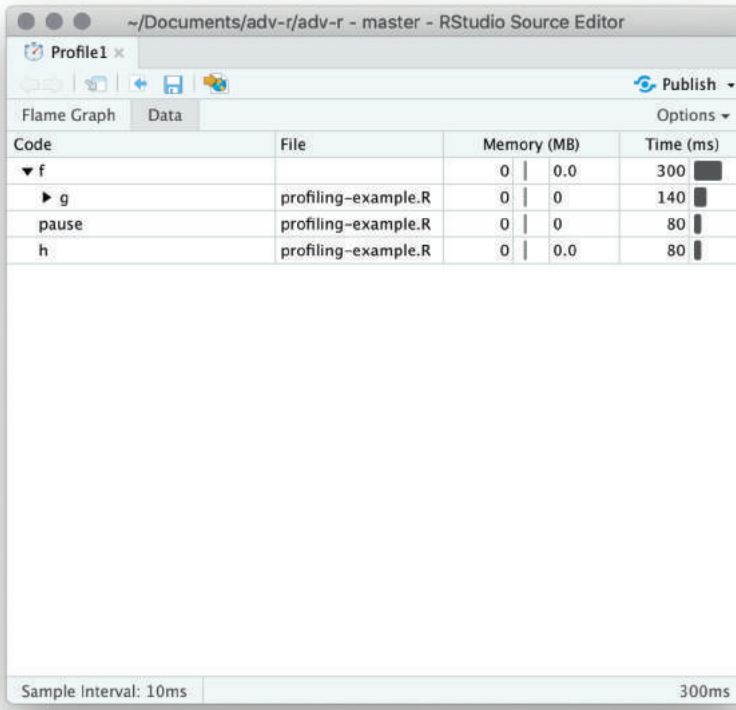


Рис. 23.3 Интерактивное дерево, позволяющее выборочно углубляться в интересующий вас компонент

Если выполнить профилирование этого кода, можно заметить, что большую часть времени занимает работа сборщика мусора (см. рис. 23.4).

В таких случаях зачастую можно обнаружить источник проблемы, взглянув на колонку **Memory**: вы увидите большие значения выделения памяти (правый столбик) и ее освобождения (левый столбик). В нашем случае проблема возникла из-за механизма копирования при изменении (см. раздел 2.3): на каждой итерации цикла создается новая копия объекта `x`. В разделе 24.6 мы поговорим детально о решении подобных проблем.

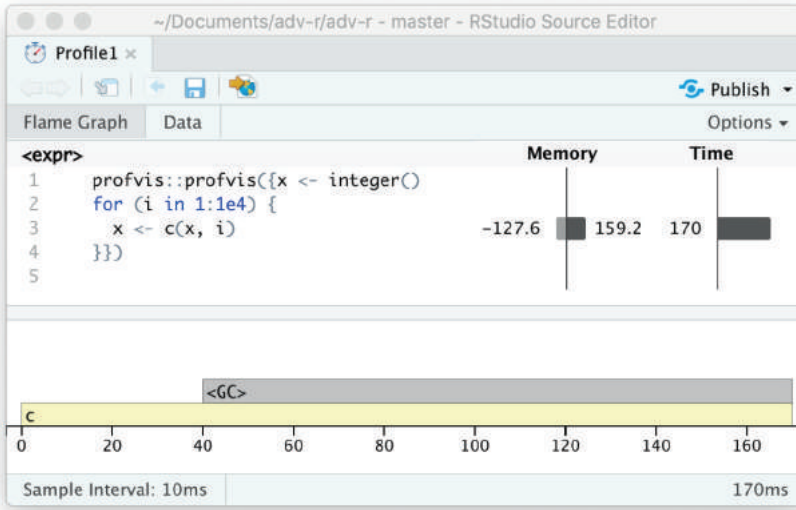


Рис. 23.4 Профилирование кода, модифицирующего переменную в цикле, показывает высокую загрузку сборщика мусора

23.2.3 Ограничения

Процесс профилирования имеет ряд ограничений:

- профилирование не распространяется на код, написанный на C. Вы сможете увидеть, что ваш код на R вызывает код на C/C++, но что происходит в самом этом блоке, для вас будет скрыто. К сожалению, рассмотрение инструментов для профилирования скомпилированного кода выходит за рамки этой книги, но вы можете начать самостоятельное исследование отсюда: https://github.com/r-prof/jointprof;
- если в вашем коде активно используется парадигма функционального программирования с анонимными функциями, вам может быть затруднительно определить, когда и какие именно функции вызываются. Простейший способ избежать этого – именовать функции;
- применение отложенного вычисления приводит к тому, что аргументы зачастую вычисляются внутри других функций, что затрудняет построение стека вызовов (см. раздел 7.5.2). К сожалению, профайлер в R хранит недостаточное количество информации для распутывания клубка отложенных вычислений, так что при профилировании показанного ниже кода может сложиться впечатление, что функция `i()` была вызвана из функции `j()`, по причине того, что аргумент не вычисляется до тех пор, пока не понадобится в функции `j()`.

```
i <- function() {
    pause(0.1)
```

```
    10
}
j <- function(x) {
    x + 10
}
j(i())
```

Если вас это сбивает с толку, вы можете воспользоваться функцией `force()`, позволяющей осуществить вычисления принудительно (см. раздел 10.2.3).

23.2.4 Упражнения

1. Выполните профилирование следующего кода с помощью функции `profvis::profvis()` с параметром `torture = TRUE`. Вас что-то удивило? Ознакомьтесь с исходным кодом функции `rm()`, чтобы понять, что происходит.

```
f <- function(n = 1e5) {
    x <- rep(1, n)
    rm(x)
}
```

23.3 Эталонное микротестирование

Эталонное микротестирование (microbenchmark) представляет собой процесс измерения производительности очень небольших участков кода, которые могут требовать несколько милли-, микро- или наносекунд для выполнения. Эталонное микротестирование бывает полезно при сравнении эффективности небольших фрагментов кода, предназначенных для выполнения конкретных задач. Избегайте обобщения результатов микротестирования на реальный код: наблюдаемые различия в результатах в большинстве случаев будут нивелированы эффектами более высокого порядка, присущими для реального кода. А глубокое понимание физики элементарных частиц не слишком-то помогает при хлебопечении.

Очень полезным инструментом эталонного микротестирования в R является пакет `bench` [Эстер (Hester), 2018]. В этом пакете используется высокоточный таймер, позволяющий сравнивать ничтожно малые промежутки времени. К примеру, в следующем коде сравнивается скорость вычисления квадратного корня с использованием разных подходов.

```
x <- runif(100)
(lb <- bench::mark(
    sqrt(x),
    x ^ 0.5
```

```
))
#> # A tibble: 2 x 10
#>   expression      min   mean  median     max `itr/sec` mem_alloc
#>   <chr>      <bch:tm> <bch:> <bch:t> <bch:t>     <dbl> <bch:byt>
#> 1 sqrt(x)       442ns 1.29µs   665ns  45.6µs   776952.      848B
#> 2 x^0.5        3.05µs 3.65µs   3.2µs 100.4µs   274024.      848B
#> # … with 3 more variables: n_gc <dbl>, n_itr <int>,
#> #   total_time <bch:tm>
```

По умолчанию функция `bench::mark()` запускает каждое выражение минимум один раз (`min_iterations = 1`) и максимальное количество раз, которые уместятся в 0,5 с (`min_time = 0.5`). При этом функция проверяет, что все запуски возвращают одно и то же значение, ведь обычно нам требуется именно это. Если вы хотите сравнить производительность выражений, возвращающих разные результаты, запустите функцию с аргументом `check = FALSE`.

23.3.1 Результаты bench::mark()

Функция `bench::mark()` возвращает результат в виде табличного объекта `tibble`, с одной строкой для каждого входного выражения и следующими колонками:

- в колонках `min`, `mean`, `median`, `max` и `itr/sec` агрегируется время, потребовавшееся для выполнения выражения. Вы можете обращать внимание только на минимальное значение (`min`) и медианное, т. е. типичное (`median`). В нашем случае видно, что использование специальной функции `sqrt()` оказалось более выгодным по сравнению с оператором возведения в степень. Визуализировать распределение отдельных таймингов можно при помощи функции `plot()` следующим образом:

  ```
  plot(lb)
  ```

  Результат показан на рис. 23.5.

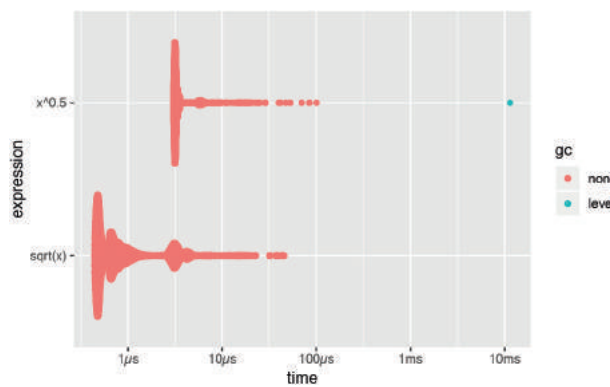


Рис. 23.5 Вывод результатов функции `bench::mark()`

Распределение имеет небольшой скос в правую сторону (обратите внимание, что ось *x* уже логарифмирована!), из-за чего вы должны избегать сравнения средних значений. Также вы зачастую будете сталкиваться с мультимодальностью распределения по причине наличия других фоновых процессов в вашем компьютере;

- в колонке `mem_alloc` выводится объем памяти, выделенный при первом запуске, а в колонке `n_gc` – общее количество запусков сборщика мусора за все время выполнения. Эту информацию можно использовать при оценке израсходованной памяти;
- информация в колонках `n_itr` и `total_time` говорит об общем количестве запусков выражения и общем времени выполнения соответственно. Значение в колонке `n_itr` всегда будет превосходить установку параметра `min_iteration`, а значение `total_time` всегда будет больше, чем значение параметра `min_time`;
- колонки `result`, `memory`, `time` и `gc` служат для хранения исходных данных в виде списков.

Из-за того что данные представлены в виде таблицы `tibble`, вы можете воспользоваться оператором [для извлечения только нужных вам колонок. В следующей главе мы будем часто использовать этот прием.

```
lb[c("expression", "min", "median", "itr/sec", "n_gc")]
#> # A tibble: 2 x 5
#>   expression      min   median `itr/sec`  n_gc
#>   <chr>      <bch:tm> <bch:tm>     <dbl> <dbl>
#> 1 sqrt(x)       442ns    665ns   776952.     0
#> 2 x^0.5        3.05µs    3.2µs   274024.     1
```

23.3.2 Интерпретирование результатов

В случае со всеми выполняемыми эталонными микротестированиями очень важно обращать внимание на единицы измерения времени. В нашем случае каждое вычисление занимало порядка 440 нс, т. е. 440 миллиардных долей секунды. Для калибровки результатов тестирования бывает полезно думать о том, сколько раз функция должна быть выполнена, чтобы уместиться в секунду. Если микротестирование занимает:

- 1 мс, значит, 1000 вызовов займут 1 секунду;
- 1 мкс, значит, 1 000 000 вызовов займут 1 секунду;
- 1 нс, значит, 1 000 000 000 вызовов займут 1 секунду.

Выполнение функции `sqrt()` потребовало порядка 440 нс, или 0,44 мкс, на вычисление квадратного корня 100 раз. Таким образом, при повторении этой операции миллион раз общее время составит 0,44 с, а значит, изменение способа вычисления корня вряд ли окажет заметное влияние на реальный код. Именно поэтому необходимо обращать большое внимание на единицы измерения и обобщение результатов эталонного микротестирования.

23.3.3 Упражнения

1. Вместо функции `bench::mark()` мы могли бы использовать встроенную функцию `system.time()`. Но в точности она значительно уступает функции `bench::mark()`, в связи с чем вам придется повторять каждую операцию много раз в цикле, после чего выполнять деление, чтобы найти среднее время выполнения, что проиллюстрировано в коде ниже.

```
n <- 1e6
system.time(for (i in 1:n) sqrt(x)) / n
system.time(for (i in 1:n) x ^ 0.5) / n
```

   Насколько оценки, полученные при помощи функции `system.time()`, будут отличаться от результатов функции `bench::mark()`? Почему наблюдается такая разница?

2. Ниже представлены два способа вычисления квадратного корня применительно к векторам. Как вы думаете, какой из них окажется быстрее? Проверьте свои догадки, выполнив эталонное микротестирование.

```
x ^ (1 / 2)
exp(log(x) / 2)
```

24

Повышение производительности

24.1 Введение

> *Не стоит тратить время на производительность, не имеющую большого значения, а это порядка 97 %. Ненужная оптимизация – корень всех зол. Но в оставшихся 3 % критических случаев нельзя упускать возможности для оптимизации. Хороший программист должен очень внимательно относиться к быстродействию критически важного кода, но только после его точной идентификации.*
>
> *— Дональд Кнут*

Воспользовавшись техникой профилирования кода для поиска узких мест, вы должны подумать, как можно ускорить найденные фрагменты. Очень трудно дать какие-то общие советы по ускорению кода, так что я остановлюсь на четырех основных техниках, применимых в области оптимизации в большинстве случаев. Также я предложу базовую стратегию оптимизации, позволяющую убедиться в том, что ускорение работы кода не отразится на его корректности.

Легко попасть в ловушку стремления избавиться от всех без исключения узких мест в коде. Не стоит этого делать! Ваше время – это ваш актив, и лучше будет потратить его на анализ данных, а не на истребление любых мелких изъянов в коде, связанных с производительностью. Включайте прагматика и не тратьте часы своего драгоценного времени на то, чтобы сэкономить какие-то секунды времени выполнения программы. Для этого вы должны всегда ставить для своего кода целевые временные показатели и оптимизировать код только до их достижения. Нет, вы не избавитесь от всех узких мест. К каким-то из них вы даже не приступите, поскольку к тому времени уже достигнете установленной цели. Все оставшиеся узкие места можно проигнорировать либо ввиду отсутствия легких и быстрых способов их устранить, либо по причине того, что ваш код уже достаточно оптимизирован и не нуждается в дополнительном ускорении. Научитесь удовлетворяться достигнутым результатом и двигаться дальше.

Если вам хочется глубже изучить тему быстродействия кода в R, я рекомендую прочитать материал *Evaluating the Design of the R Language* [Морандат (Morandat) и др., 2012]. В нем собраны самые разные идеи от изменений в интерпретаторе R до использования стороннего кода.

Структура главы

- В разделе 24.2 мы узнаем о принципах организации кода, способствующих его легкой и безболезненной оптимизации.
- Раздел 24.3 будет посвящен важности использования существующих решений.
- В разделе 24.4 мы узнаем, как полезно иногда лениться: зачастую простейший путь ускорения работы функции состоит в том, чтобы не нагружать ее лишней работой.
- В разделе 24.5 мы познакомимся с векторизацией и научимся извлекать максимум из встроенных функций.
- В разделе 24.6 мы обсудим риски, связанные с копированием данных.
- В разделе 24.7 мы сведем все полученные знания воедино и реализуем действия по ускорению процедуры, связанной с повторным вычислением t-критерия, в тысячу раз.
- В разделе 24.8 мы подведем итог главы и перечислим дополнительные ресурсы, которые позволят вам писать быстрый и эффективный код.

Требования

В данной главе мы будем пользоваться пакетом `bench` (https://bench.r-lib.org) для сравнения быстродействия небольших независимых блоков кода.

```
library(bench)
```

24.2 Организация кода

Существуют две ловушки, в которые легко угодить в попытках сделать свой код быстрее.

1. Код ускорится, но работать станет неправильно.
2. Вам будет казаться, что код ускорился, хотя на самом деле все будет не так.

Стратегия, описанная ниже, поможет вам избежать попадания в такие ситуации.

В попытке устранить узкое место в коде вы зачастую прибегаете к разным подходам. Напишите отдельную функцию для каждого из них, реализовав в ней все необходимое поведение. Это облегчит проверку того, что подходы возвращают правильные результаты, и позволит точно определить затрачиваемое время. Для демонстрации данной стратегии сравним два подхода к вычислению среднего:

```
mean1 <- function(x) mean(x)
mean2 <- function(x) sum(x) / length(x)
```

Я рекомендую вам записывать все полученные результаты, даже ошибочные. Если с подобной проблемой вы столкнетесь в будущем, вам будет полезно вспомнить, какие шаги для ее решения вы уже предпринимали. Для этого удобно воспользоваться разметкой RMarkdown, позволяющей смешивать код с подробными комментариями и заметками.

После этого создайте репрезентативный сценарий тестирования. Он должен быть достаточно сложным, чтобы передавать суть проблемы, и достаточно простым, чтобы выполняться не более нескольких секунд. Не стоит делать этот сценарий слишком длинным, поскольку в процессе оптимизации вам придется не раз его выполнять для сравнения различных подходов. С другой стороны, не делайте сценарий слишком коротким, ведь в этом случае полученные результаты трудно будет масштабировать до реальной проблемы. В нашем случае достаточно будет использовать 100 000 чисел:

```
x <- runif(1e5)
```

Теперь воспользуемся функцией `bench::mark()` для точного сравнения результатов. Эта функция автоматически проверяет, что все вызовы возвращают одинаковые значения. Это не гарантирует того, что функция выполняется одинаково для всех входных значений. В идеале вы также должны использовать *модульное тестирование* (unit tests), чтобы убедиться, что вы случайно не меняете поведение функции.

```
bench::mark(
    mean1(x),
    mean2(x)
)[c("expression", "min", "median", "itr/sec", "n_gc")]
#> # A tibble: 2 x 5
#>   expression      min   median `itr/sec`  n_gc
#>   <chr>      <bch:tm> <bch:tm>     <dbl> <dbl>
#> 1 mean1(x)      191µs    219µs     4558.     0
#> 2 mean2(x)      101µs    113µs     8832.     0
```

(Вас могут удивить полученные результаты: функция `mean(x)` оказалась значительно медленнее, чем выражение `sum(x) / length(x)`. Помимо прочих причин, дело еще и в том, что функция `mean(x)` осуществляет двойной проход по вектору для большей точности.)

Если вы хотите увидеть эту стратегию в действии, я не раз использовал ее на StackOverflow:

- http://stackoverflow.com/questions/22515525#22518603;
- http://stackoverflow.com/questions/22515175#22515856;
- http://stackoverflow.com/questions/3476015#22511936.

24.3 Поиск существующих решений

Осуществив правильную организацию кода и выявив все возможные способы решения проблемы, вы непременно захотите узнать, что на этом поприще сделали другие разработчики. Вы – часть большого сообщества, и велики шансы, что кто-то до вас уже сталкивался с подобной проблемой. Начать поиск можно с двух следующих ресурсов:

- страницы https://cran.rstudio.com/web/views. Если в списке присутствуют задачи со схожей с вашей предметной областью, стоит присмотреться к используемым в них пакетам;
- зависимости из пакета Rcpp, перечисленные на его странице в CRAN (http://cran.r-project.org/web/packages/Rcpp). Поскольку в этих пакетах используется язык C++, они, скорее всего, будут отличаться быстродействием.

Если это не поможет, вам придется наиболее точно подбирать слова для описания проблемы, с которой столкнулись. Если бы вы знали, как обозначается ваша проблема и какие у нее есть синонимы, эффективность поиска была бы выше. Но обычно вы этого не знаете, что значительно осложняет поиск. Лучший способ здесь – читать много литературы, чтобы расширять свой словарный запас. Также вы можете обсудить ключевые слова с товарищами. Посоветуйтесь с коллегами, выпишите какие-то общие ключевые слова и отправляйтесь с ними в Google и StackOverflow. Очень часто бывает полезно ограничить свой поиск языком R. В Google вы можете воспользоваться сервисом http://www.rseek.org. Что касается StackOverflow, вы можете ограничить поиск, включив в него тег R, `[R]`.

Запишите все найденные решения, а не только те, которые кажутся более быстрыми. Некоторые решения могут поначалу работать медленно, но по итогам оптимизации стать довольно быстрыми. Также вы можете комбинировать наиболее быстрые части различных решений. Если в процессе вы нашли оптимальное решение своей проблемы, что ж, поздравляю! Если нет, читайте дальше.

24.3.1 Упражнения

1. Какие есть более быстрые альтернативы функции `lm()`? Какие из них специально предназначены для работы с большими наборами данных?
2. Какой пакет реализует версию функции `match()`, оптимизированную для выполнения повторного поиска? Насколько эта реализация быстрее?
3. Назовите четыре функции (не только из состава базового R), предназначенные для преобразования строк в объекты даты и времени. Какие у них есть преимущества и недостатки?

4. Какие пакеты предоставляют возможности для расчета скользящего среднего?
5. Какие существуют альтернативы функции `optim()`?

24.4 Не делайте больше, чем нужно

Самый простой способ повысить быстродействие функции состоит в том, чтобы наделить ее меньшим объемом работы. Один из вариантов – использовать функции, реализованные под конкретный тип ввода и вывода или призванные решать определенные задачи. Примеры:

- функции `rowSums()`, `colSums()`, `rowMeans()` и `colMeans()` будут выполняться быстрее соответствующих им аналогов, использующих функцию `apply()`, по причине своей векторизованности (см. раздел 24.5);
- функция `vapply()` работает быстрее в сравнении с `sapply()` из-за конкретно определенного типа данных на выходе;
- если вам нужно проверить, содержит ли вектор нужное вам значение, функция `any(x == 10)` отработает гораздо быстрее по сравнению с выражением `10 %in% x` в связи с тем, что проверка на равенство является менее затратной операцией на фоне проверки на вхождение во множество.

Но чтобы знать о существовании такого разнообразия решений одних и тех же задач, необходимо постоянно расширять свой кругозор в R. Читайте много кода. В этом вам могут помочь списки рассылок на сайте https://stat.ethz.ch/mailman/listinfo/r-help и тот же StackOverflow с соответствующим тегом (http://stackoverflow.com/questions/tagged/r).

Некоторые функции выполняют приведение типов входных параметров. Таким образом, если вы передаете на вход функции данные иного типа, она будет вынуждена потратить определенное время на соответствующие преобразования. В связи с этим лучше искать функции, которые будут работать с вашими данными как есть. Если таковых нет, можно изменить вариант хранения исходных данных. Наиболее показательным примером здесь является использование функции `apply()` применительно к датафреймам. Функция `apply()` всегда преобразует входные данные к матричному виду. Мало того, что это может быть сопряжено с логическими ошибками (поскольку датафрейм представляет собой более общий тип данных в сравнении с матрицей), так еще и время тратится.

Есть функции, которые при передаче им дополнительной информации будут выполнять меньше работы. Для выявления подобных возможностей нужно всегда внимательно читать документацию и экспериментировать с различными параметрами. Вот лишь некоторые примеры, с которыми на практике сталкивался лично я:

- `read.csv()`: укажите известные типы колонок с помощью аргумента `colClasses`. Вместе с тем вы можете рассмотреть идею перехода на

более быстрые реализации в виде функций `readr::read_csv()` и `data.table::fread()`);

- `factor()`: укажите известные уровни с помощью аргумента `levels`;
- `cut()`: не генерируйте метки с помощью аргумента `labels = FALSE`, если они вам не нужны, а еще лучше воспользуйтесь функцией `findInterval()`, как описано в дополнительной секции документации;
- вызов `unlist(x, use.names = FALSE)` сработает гораздо быстрее, чем `unlist(x)`;
- `interaction()`: если вам нужны только те комбинации, которые присутствуют в данных, воспользуйтесь аргументом `drop = TRUE`.

Давайте посмотрим на примерах, как можно оптимизировать применение распространенных функций `mean()` и `as.data.frame()`.

24.4.1 mean()

Иногда можно ускорить работу функции путем избежания диспетчеризации методов. При вызове метода в коротком цикле можно снизить накладные расходы за счет однократного поиска метода:

- в случае с S3 это можно реализовать, вызвав `generic.class()` вместо `generic()`;
- в S4 это можно сделать с помощью функции `getMethod()` для поиска метода, сохранения его в переменную и дальнейшего вызова.

К примеру, для небольших векторов вызов метода `mean.default()` будет гораздо более эффективным в сравнении с `mean()`:

```
x <- runif(1e2)

bench::mark(
    mean(x),
    mean.default(x)
)[c("expression", "min", "median", "itr/sec", "n_gc")]
#> # A tibble: 2 x 5
#>   expression           min   median `itr/sec`  n_gc
#>   <chr>           <bch:tm> <bch:tm>     <dbl> <dbl>
#> 1 mean(x)          2.48µs   2.81µs   325109.     0
#> 2 mean.default(x)  1.18µs   1.29µs   703275.     1
```

Такая оптимизация сопряжена с определенными рисками. Тогда как на ста значениях метод `mean.default()` показывает вдвое лучший результат, все будет совсем иначе, если `x` будет представлен нечисловым вектором.

Еще более опасно осуществлять вызов лежащей в основе функции `.Internal` напрямую. Она показывает очень высокую эффективность, что связано с отсутствием необходимости выполнять проверки входных параметров и обработку значений NA, так что мы платим скоростью за безопасность.

```
x <- runif(1e2)
bench::mark(
    mean(x),
    mean.default(x),
    .Internal(mean(x))
)[c("expression", "min", "median", "itr/sec", "n_gc")]
#> # A tibble: 3 x 5
#>   expression              min   median `itr/sec`  n_gc
#>   <chr>              <bch:tm> <bch:tm>     <dbl> <dbl>
#> 1 mean(x)              2.47µs   2.79µs   344523.     1
#> 2 mean.default(x)      1.16µs   1.37µs   697365.     0
#> 3 .Internal(mean(x))    325ns    343ns  2813476.     0
```

Примечание. Большинство этих различий в скорости возникают вследствие небольших значений x. При увеличении x разница может нивелироваться по причине того, что основные ресурсы будут тратиться на расчет среднего, а не на поиск подходящей реализации метода. Это хороший пример того, что объем входных параметров важен, и свою оптимизацию вы должны строить на основе реалистичных данных.

```
x <- runif(1e4)
bench::mark(
    mean(x),
    mean.default(x),
    .Internal(mean(x))
)[c("expression", "min", "median", "itr/sec", "n_gc")]
#> # A tibble: 3 x 5
#>   expression              min   median `itr/sec`  n_gc
#>   <chr>              <bch:tm> <bch:tm>     <dbl> <dbl>
#> 1 mean(x)                21µs   24.5µs    40569.     1
#> 2 mean.default(x)      19.7µs   19.8µs    48892.     0
#> 3 .Internal(mean(x))   18.9µs   20.2µs    48834.     0
```

24.4.2 as.data.frame()

Зная конкретный тип входа, можно, исходя из этого, оптимизировать свой код. К примеру, функция `as.data.frame()` является довольно медленной, поскольку она преобразует каждый элемент в датафрейм, после чего применяет к ним функцию `rbind()`. Если у вас есть именованный список из векторов одинаковой длины, вы можете выполнить его непосредственное преобразование в датафрейм. Таким образом, если вы воспользуетесь своими знаниями о типе входящих значений, то сможете написать метод, который будет по скорости значительно превосходить метод по умолчанию.

```
quickdf <- function(l) {
    class(l) <- "data.frame"
    attr(l, "row.names") <- .set_row_names(length(l[[1]]))
```

```
    l
}

l <- lapply(1:26, function(i) runif(1e3))
names(l) <- letters

bench::mark(
    as.data.frame = as.data.frame(l),
    quick_df = quickdf(l)
)[c("expression", "min", "median", "itr/sec", "n_gc")]
#> # A tibble: 2 x 5
#>   expression         min   median `itr/sec`  n_gc
#>   <chr>         <bch:tm> <bch:tm>     <dbl> <dbl>
#> 1 as.data.frame   1.01ms   1.12ms      875.     9
#> 2 quick_df        6.34µs   7.16µs   125452.     2
```

Отметим определенный компромисс. Этот метод быстрее, но опаснее. Если передать ему на вход не то, что ожидается, на выходе мы получим поврежденный датафрейм.

```
quickdf(list(x = 1, y = 1:2))
#> Warning in format.data.frame(if (omit) x[seq_len(n0), , drop = FALSE]
#> else x, : corrupt data frame: columns will be truncated or padded
#> with NAs
#>   x y
#> 1 1 1
```

Для реализации такого минимального метода я внимательно изучил и затем переписал исходный код функций `as.data.frame.list()` и `data.frame()`. Я внес множество мелких изменений, каждый раз проверяя, не затронул ли я поведение функций по умолчанию. После нескольких часов работы мне удалось выделить минимальную реализацию кода, показанную выше. Это очень полезная техника. Большинство базовых функций R разработаны с учетом высокой гибкости и функциональности, а отнюдь не скорости. Таким образом, переписывание их согласно собственным нуждам зачастую может давать существенный рост производительности. Для этого вам нужно уметь читать исходный код. Это бывает очень непросто, но не стоит сдаваться!

24.4.3 Упражнения

1. Какая разница между функциями `rowSums()` и `.rowSums()`?
2. Напишите более быструю версию функции `chisq.test()`, которая будет вычислять критерий хи-квадрат только при поступлении на вход двух числовых векторов с отсутствием пропущенных значений. Вы можете попытаться модифицировать функцию `chisq.test()` или написать свою, взяв математическую базу из Википедии (https://ru.wikipedia.org/wiki/Критерий_согласия_Пирсона).

3. Можете ли вы написать более быструю версию функции `table()` для случая с двумя входными целочисленными векторами с отсутствием пропущенных значений? Можно ли использовать ее для ускорения функции вычисления критерия хи-квадрат?

24.5 Векторизация

Если у вас есть хоть какой-нибудь опыт работы с R, вы наверняка не раз слышали совет векторизовать свой код. Что же это на самом деле означает? *Векторизация* (vectorising) кода не подразумевает только лишь избавление от циклов, это лишь один из шагов. Суть векторизации кроется в полнообъектном подходе к решению задачи, подразумевающем оперирование векторами, а не скалярами. Векторизованные функции обладают двумя ключевыми характеристиками:

- они облегчают решение задач. Вместо того чтобы думать о компонентах векторов, мы оперируем целыми векторами;
- циклы в векторизованных функциях реализованы на языке C, а не R, что приводит к увеличению быстродействия за счет снижения накладных расходов.

В главе 9 мы уже говорили о важности векторизации кода, способствующей повышению уровня абстракции. Векторизация также очень важна при написании эффективного кода на R. Она подразумевает не только использование функций `map()` или `lapply()`, но и поиск существующих функций в R, которые реализованы на C, с их использованием при решении поставленных задач.

Векторизованные функции, применимые к большей части задач, представляющих узкие места в коде:

- `rowSums()`, `colSums()`, `rowMeans()` и `colMeans()`. Эти векторизованные матричные функции всегда будут давать фору использованию функции `apply()`. Иногда эти функции можно прменять для создания других векторизованных функций;

```
rowAny <- function(x) rowSums(x) > 0
rowAll <- function(x) rowSums(x) == ncol(x)
```

- векторизованное извлечение подмножеств может приводить к существенному повышению быстродействия кода. Вспомните техники, лежащие в основе использования таблиц поиска (см. раздел 4.5.1) и сопоставления и объединения в ручном режиме (см. раздел 4.5.2). Также не забывайте, что можно совмещать извлечение подмножеств с операциями присваивания для массовой замены значений. Если `x` – это вектор, матрица или датафрейм, то `x[is.na(x)] <- 0` заменит все пропущенные в этой структуре данных значения нулями;

- если вы извлекаете или меняете значения в разрозненных местах матрицы или датафрейма, воспользуйтесь получением подмножеств на основе целочисленной матрицы. За подробностями можно обратиться к разделу 4.2.3;
- при преобразовании непрерывных значений в категориальные убедитесь, что вы знаете, как использовать функции `cut()` и `findInterval()`;
- не забывайте об использовании векторизованных функций `cumsum()` и `diff()`.

Матричная алгебра представляет собой общий пример векторизации. Циклы здесь выполняются при помощи хорошо настроенных внешних библиотек, таких как *BLAS*. Если вы знаете, как можно использовать матричную алгебру для удовлетворения своих потребностей, скорее всего, у вас получится очень эффективное решение. Но здесь нужен немалый опыт. И набирать его можно начать с дискуссий с опытными разработчиками в вашей предметной области.

Но у векторизации есть и недостаток: сложно предсказать заранее, как поведут себя операции при масштабировании. В примере, приведенном ниже, мы замерим время, требующееся на использование символьного подмножества для поиска 1, 10 и 100 элементов в списке. Вы могли бы подумать, что на поиск десяти элементов уйдет в десять раз больше времени, чем на поиск одного, а на поиск ста элементов – еще в десять раз больше времени. На деле же получается, и это видно в выводе ниже, что на поиск ста элементов мы затратили примерно в десять раз больше времени по сравнению с поиском одного элемента. Причина в том, что, дойдя до определенного объема, внутренняя реализация переключается на стратегию, требующую больше времени на подготовку, но при этом лучше масштабируемую.

```
lookup <- setNames(as.list(sample(100, 26)), letters)

x1 <- "j"
x10 <- sample(letters, 10)
x100 <- sample(letters, 100, replace = TRUE)

bench::mark(
    lookup[x1],
    lookup[x10],
    lookup[x100],
    check = FALSE
)[c("expression", "min", "median", "itr/sec", "n_gc")]
#> # A tibble: 3 x 5
#>   expression        min   median `itr/sec`  n_gc
#>   <chr>        <bch:tm> <bch:tm>     <dbl> <dbl>
#> 1 lookup[x1]      508ns    545ns  1571545.     1
#> 2 lookup[x10]    1.55µs   1.64µs   527835.     0
#> 3 lookup[x100]   4.93µs   7.53µs   127306.     0
```

Векторизация не является решением всех проблем. И вместо того чтобы менять существующий подход таким образом, чтобы он использовал векторизацию, зачастую будет лучше написать собственную векторизованную функцию на C++. В главе 25 вы узнаете, как можно это сделать.

24.5.1 Упражнения

1. Функции плотности, такие как `dnorm()`, обладают общим интерфейсом. Какие их аргументы векторизованы? Что делает вызов функции `rnorm(10, mean = 10:1)`?
2. Сравните скорость выражений `apply(x, 1, sum)` и `rowSums(x)` при разных `x`.
3. Как можно воспользоваться функцией `crossprod()` для вычисления средневзвешенного? Насколько быстрее она по сравнению с традиционным подходом с использованием `sum(x * w)`?

24.6 Избежание создания копий

Еще одной распространенной проблемой кода на R является увеличение размера объектов при выполнении циклов. Всякий раз, когда вы используете функции `c()`, `append()`, `cbind()`, `rbind()` или `paste()` для накапливания объектов, R должен сначала выделить память для нового объекта, а затем скопировать старый объект на новое место. При многократном повторении данной операции, например в цикле `for`, ее стоимость может значительно увеличиться. В результате вы обнаруживаете себя на втором круге *ада R* (R inferno) (http://www.burns-stat.com/pages/Tutor/R_inferno.pdf).

Один из примеров такого поведения мы наблюдали в разделе 23.2.2, а здесь мы рассмотрим более сложный пример на ту же тему. Для начала сгенерируем несколько случайных строк, а затем объединим их либо итеративно, с помощью функции `collapse()`, либо с использованием одного прохода – посредством функции `paste()`. Обратите внимание на снижение эффективности с ростом количества строк: объединение 100 строк занимает почти в 30 раз больше времени по сравнению с объединением 10 строк.

```
random_string <- function() {
    paste(sample(letters, 50, replace = TRUE), collapse = "")
}
strings10 <- replicate(10, random_string())
strings100 <- replicate(100, random_string())

collapse <- function(xs) {
    out <- ""
    for (x in xs) {
        out <- paste0(out, x)
```

```
    }
    out
}

bench::mark(
    loop10 = collapse(strings10),
    loop100 = collapse(strings100),
    vec10 = paste(strings10, collapse = ""),
    vec100 = paste(strings100, collapse = ""),
    check = FALSE
)[c("expression", "min", "median", "itr/sec", "n_gc")]
#> # A tibble: 4 x 5
#>   expression      min   median `itr/sec`  n_gc
#>   <chr>      <bch:tm> <bch:tm>     <dbl> <dbl>
#> 1 loop10      16.87µs   19.8µs    49742.     3
#> 2 loop100     578.8µs  618.2µs     1575.     3
#> 3 vec10        4.53µs      5µs   196192.     0
#> 4 vec100      31.65µs   32.8µs    29970.     0
```

Модификация объекта внутри цикла (например, `x[i] <- y`) также может привести к созданию копии в зависимости от класса объекта `x`. В разделе 2.5.1 мы более подробно обсуждали эту проблему и определили для себя набор инструментов для определения того, создаются ли копии объектов.

24.7 Практический пример: расчет t-критерия Стьюдента

В данном разделе мы узнаем, как можно ускорить процедуру расчета *t-критерия Стьюдента* (t-test) с использованием техник, описанных выше. Мы возьмем за основу пример, описанный в труде Хольгера Швендера (Holger Schwender) и Тины Мюллер (Tina Müller) *Computing thousands of test statistics simultaneously in R*. Я очень рекомендую вам найти эту работу и ознакомиться с идеями, применяемыми в ней к другим тестам.

Представим, что мы провели 1000 экспериментов (строки), в каждом из которых собрали сведения по 50 испытуемым (колонки). При этом первых 25 испытуемых в каждом эксперименте мы отнесли к группе 1, а оставшихся 25 – к группе 2. Сначала сгенерируем случайные данные для нашей задачи:

```
m <- 1000
n <- 50
X <- matrix(rnorm(m * n, mean = 10, sd = 3), nrow = m)
grp <- rep(1:2, each = n / 2)
```

Применительно к данным, представленным в таком виде, есть два способа использования функции `t.test()`. Можно либо воспользоваться формулой,

либо передать два вектора – по одному для каждой группы. Исследование показало, что вариант с формулой значительно медленнее:

```
system.time(
    for (i in 1:m) {
        t.test(X[i, ] ~ grp)$statistic
    }
)
#>     user  system elapsed
#>    0.707   0.003   0.712
system.time(
    for (i in 1:m) {
        t.test(X[i, grp == 1], X[i, grp == 2])$statistic
    }
)
#>     user  system elapsed
#>    0.186   0.002   0.189
```

Конечно, в цикле `for` у нас производятся вычисления, но результаты не сохраняются. Для этого мы можем воспользоваться функцией `map_dbl()`, о которой упоминали в разделе 9.2.1. Это несколько повысит накладные расходы:

```
compT <- function(i){
    t.test(X[i, grp == 1], X[i, grp == 2])$statistic
}
system.time(t1 <- purrr::map_dbl(1:m, compT))
#>     user  system elapsed
#>    0.186   0.001   0.187
```

Как можно ускорить этот код? Первое, что нужно сделать, – это избавить функцию от лишней нагрузки. Если взглянуть на исходный код метода `stats:::t.test.default()`, можно заметить, что в нем выполняется гораздо больше операций, чем просто расчет t-критерия. Помимо прочего, этом метод вычисляет p-значение и форматирует вывод. Мы можем ускорить наш код, исключив эти ненужные действия.

```
my_t <- function(x, grp) {
    t_stat <- function(x) {
        m <- mean(x)
        n <- length(x)
        var <- sum((x - m) ^ 2) / (n - 1)

        list(m = m, n = n, var = var)
    }

    g1 <- t_stat(x[grp == 1])
    g2 <- t_stat(x[grp == 2])

    se_total <- sqrt(g1$var / g1$n + g2$var / g2$n)
```

```
    (g1$m - g2$m) / se_total
}

system.time(t2 <- purrr::map_dbl(1:m, ~ my_t(X[.,], grp)))
#>     user  system elapsed
#>    0.028   0.000   0.028
stopifnot(all.equal(t1, t2))
```

Это позволило нам примерно в шесть раз увеличить быстродействие кода.

Теперь, когда мы получили довольно простую функцию, мы можем ускорить ее за счет векторизации. Чтобы не осуществлять перебор по массиву вне функции, мы изменим функцию `t_stat()` таким образом, чтобы она могла работать с матрицами значений. В результате `mean()` превратится в `rowMeans()`, `length()` – в `ncol()`, а `sum()` – в `rowSums()`. Остальной код сохранится без изменений.

```
rowtstat <- function(X, grp){
    t_stat <- function(X) {
        m <- rowMeans(X)
        n <- ncol(X)
        var <- rowSums((X - m) ^ 2) / (n - 1)

        list(m = m, n = n, var = var)
    }

    g1 <- t_stat(X[, grp == 1])
    g2 <- t_stat(X[, grp == 2])

    se_total <- sqrt(g1$var / g1$n + g2$var / g2$n)
    (g1$m - g2$m) / se_total
}
system.time(t3 <- rowtstat(X, grp))
#>     user  system elapsed
#>    0.011   0.000   0.011
stopifnot(all.equal(t1, t3))
```

Вот так-то лучше! Мы получили примерно 40-кратное ускорение по сравнению с предыдущей реализацией и 1000-кратное, если сравнивать с нашей отправной точкой.

24.8 Другие техники

Умение писать быстрый код на R – неотъемлемый навык любого хорошего программиста на этом языке. Помимо использования приемов и техник, которые мы уже обсудили в этой главе, вам необходимо повышать и уровень своей компетенции в искусстве программирования в целом. Вот какие есть варианты для этого:

- читать блоги по R (https://www.r-bloggers.com), чтобы понимать, с какими проблемами в отношении производительности сталкиваются другие разработчики на R и как они их решают;
- читать другие книги по программированию на R, такие как *The Art of R Programming* (Матлофф (Matloff), 2011) или *R Inferno* Патрика Бернса (Patrick Burns) (https://www.burns-stat.com/documents/books/the-r-inferno), чтобы узнать об иных распространенных ловушках и узких местах;
- пройти курс по алгоритмам и структурам данных для изучения способов решения классических задач. Я слышал много хороших слов о курсе *Princeton's Algorithms* на Coursera;
- изучать принципы параллелизма в коде. Начать можно с книг *Parallel R* [МакКаллум (McCallum) и Уэстон (Weston), 2011] и *Parallel Computing for Data Science* [Матлофф (Matloff), 2015];
- читать книги по оптимизации, такие как *Mature optimisation* [Буэно (Bueno), 2013] или *Pragmatic Programmer* [Хант (Hunt) и Томас (Thomas), 1990].

Вы также всегда можете обратиться к сообществу за помощью. Сайт StackOverflow – это отличный ресурс. Конечно, для его эффективного использования вам придется создавать обобщенные воспроизводимые примеры, демонстрирующие возникшие у вас проблемы. Если ваши примеры будут слишком сложными, мало кто согласится тратить свое время, чтобы пробиться к корню проблемы. С другой стороны, если сильно упростить пример, вы получите очень общие ответы, подходящие для игрушечного сценария, а не для полноценного воплощения на практике. Начните сами отвечать на вопросы на том же StackOverflow, и вы очень быстро поймете, что из себя представляет хорошо заданный вопрос.

25

Переписывание кода R на C++

25.1 Введение

Иногда код, написанный на R, оказывается недостаточно быстрым. Тогда вы используете профилирование, находите узкие места в коде, оптимизируете их и делаете все, что возможно сделать средствами языка R. Но в итоге все равно остаетесь недовольны скоростью. В этой главе мы поговорим о том, как можно повысить эффективность кода на R за счет переписывания его ключевых функций на C++. Это волшебство становится возможным благодаря пакету `Rcpp` (https://www.rcpp.org) [Эддельбуэттель (Eddelbuettel) и Франсуа (François), 2011, при участии Дага Бейтса (Doug Bates), Джона Чемберса (John Chambers) и Джей Джей Аллаира (JJ Allaire)].

С помощью пакета `Rcpp` можно легко и просто объединить языки C++ и R. И хотя теоретически можно писать для интеграции с R фрагменты кода на C и Fortran, это будет не так просто. Пакет `Rcpp` предоставляет простой и доступный API, позволяющий писать высокопроизводительный код в отрыве от сложного API языка C в R.

Типичные узкие места в коде, с которыми может эффективно справиться интеграция с C++:

- циклы, которые не могут быть легко векторизованы, например вследствие жесткой зависимости следующих итераций от предыдущих;
- рекурсивные функции или задачи, требующие вызова функции миллионы раз. Накладные расходы на вызов функций в C++ гораздо меньше, чем в R;
- задачи, требующие применения продвинутых структур данных и алгоритмов, не реализованных в R. В стандартной библиотеке шаблонов (standard template library – STL) в C++ содержатся реализации многих важных структур данных, начиная с *упорядоченного отображения* (ordered map) и заканчивая *двусторонней очередью* (double-ended queue).

Цель этой главы состоит в обсуждении только тех аспектов C++ и пакета Rcpp, которые могут помочь в устранении узких мест в вашем коде. Мы не будем тратить время на продвинутые техники, такие как объектно ориен-

тированное программирование или шаблоны, поскольку наша цель ограничивается написанием небольших полезных функций, а не полноценных программ. Опыт работы с C++ вам здесь, безусловно, поможет, хотя и не является обязательным. По этому языку есть множество бесплатных руководств и обучающих материалов, включая https://www.learncpp.com и https://en.cppreference.com/w/cpp. Для более глубокого изучения языка подойдет серия книг Скотта Майерса (Scott Meyers) *Effective C++*.

Структура главы

- В разделе 25.2 мы узнаем о том, как писать код на C++, конвертируя простые функции R в соответствующие эквиваленты. Мы поговорим об отличиях C++ от R и о том, какие ключевые скалярные, векторные и матричные классы вызываются.
- В разделе 25.2.5 мы покажем, как можно использовать функцию `sourceCpp()` для загрузки файла C++ с диска таким же образом, как в случае с функцией `source()` для загрузки кода на R.
- В разделе 25.3 будет обсуждаться тема модифицирования атрибутов из `Rcpp`, а также мы упомянем другие важные классы.
- Раздел 25.4 будет посвящен работе с пропущенными значениями в C++.
- В разделе 25.5 мы увидим, как можно использовать некоторые полезные структуры данных и алгоритмы из встроенной в C++ стандартной библиотеки шаблонов.
- В разделе 25.6 мы рассмотрим два реальных примера использования пакета `Rcpp` с целью повышения производительности кода.
- В главе 25.7 мы научимся добавлять код C++ к пакетам.
- В главе 25.8 мы подведем итоги главы и перечислим некоторые полезные ресурсы для самостоятельного изучения `Rcpp` и C++.

Требования

Мы будем использовать пакет `Rcpp` (http://www.rcpp.org) для вызова функций C++ из R:

```
library(Rcpp)
```

Также мы будем работать с компилятором C++. Для этого:

- на Windows установите набор инструментов *Rtools* (https://cran.r-project.org/bin/windows/Rtools);
- на Mac установите Xcode из официального магазина;
- на Linux воспользуйтесь командой `sudo apt-get install r-base-dev` или аналогичной.

25.2 Начинаем работать с C++

Функция `cppFunction()` позволяет писать функции на языке C++ в рамках кода на R:

```
cppFunction('int add(int x, int y, int z) {
    int sum = x + y + z;
    return sum;
}')
# Функция add работает как обычная функция на R
add
#> function (x, y, z)
#> .Call(<pointer: 0x109eeac90>, x, y, z)
add(1, 2, 3)
#> [1] 6
```

При запуске этого кода движок `Rcpp` скомпилирует код на C++ и сконструирует функцию R, которая будет обращаться к скомпилированному коду C++. При этом под капотом происходит очень много сложных действий, о которых заботится пакет `Rcpp`, скрывая от вас подробности.

В следующих разделах мы научимся преобразовывать простые функции на R в эквиваленты на C++. Начнем мы с функции, не принимающей аргументов и возвращающей скалярную величину, после чего постепенно будем усложнять наш сценарий согласно следующему плану:

- скалярный вход, скалярный выход;
- векторный вход, скалярный выход;
- векторный вход, векторный выход;
- матричный вход, векторный выход.

25.2.1 Нет входа, скалярный выход

Начнем с простейшей функции. Она не будет иметь входных аргументов, а на выход всегда будет отдавать целочисленное значение 1:

```
one <- function() 1L
```

Эквивалент на языке C++ будет выглядеть так:

```
int one() {
    return 1;
}
```

Мы можем скомпилировать и использовать этот код в R с помощью функции `cppFunction()`:

```
cppFunction('int one() {
    return 1;
}')
```

На примере такой простой функции видны некоторые ключевые отличия между C++ и R:

- синтаксис создания функции напоминает синтаксис ее вызова: в C++ не используется оператор присваивания при определении функций;
- вы обязаны объявить тип возвращаемого функцией значения. В нашем случае функция возвращает целочисленный результат (`int`). В R наиболее распространенные типы векторов – это `NumericVector`, `IntegerVector`, `CharacterVector` и `LogicalVector`;
- скаляры отличаются от векторов. Эквивалентами скалярных значений для числового, целочисленного, символьного и логического векторов выступают `double`, `int`, `String` и `bool`;
- вы должны явно указывать ключевое слово `return` для возвращения результата функции;
- каждая инструкция в C++ завершается точкой с запятой (`;`).

25.2.2 Скалярный вход, скалярный выход

Ниже представлена реализация скалярной версии функции `sign()`, которая возвращает 1, если на вход поступило положительное значение, –1 – если отрицательное, и 0 – если нулевое:

```
signR <- function(x) {
    if (x > 0) {
        1
    } else if (x == 0) {
        0
    } else {
        -1
    }
}

cppFunction('int signC(int x) {
    if (x > 0) {
        return 1;
    } else if (x == 0) {
        return 0;
    } else {
        return -1;
    }
}')
```

В версии C++ мы:

- декларируем тип каждого входного аргумента так же точно, как декларируем тип возвращаемого значения. Хотя это делает код более много-

словным, в качестве плюса мы имеем строгую типизацию входов и выходов функций;

- в целом синтаксис функции остался без изменений. Да, между C++ и R есть очень много отличий, но присутствуют и сходства. В C++ тоже есть ключевое слово `while`, работающее так же точно, как в R. Как и в R, здесь вы можете воспользоваться инструкцией `break` для прекращения цикла, но для перехода к следующей итерации придется воспользоваться командой `continue` вместо `next`.

25.2.3 Векторный вход, скалярный выход

Одним из существенных отличий R от C++ является бóльшая стоимость выполнения циклов. К примеру, мы могли бы реализовать функцию `sum` в R с использованием цикла. Если у вас есть хоть какой-то опыт программирования в R, эта функция вызовет у вас отвращение!

```
sumR <- function(x) {
    total <- 0
    for (i in seq_along(x)) {
        total <- total + x[i]
    }
    total
}
```

В C++ циклы обладают очень низкими накладными расходами, так что в их использовании нет ничего зазорного. В разделе 25.5 мы рассмотрим альтернативы циклам `for`, с помощью которых вы сможете более ясно выражать свои намерения. Они не будут выполняться быстрее, но сделают ваш код более интуитивно понятным.

```
cppFunction('double sumC(NumericVector x) {
    int n = x.size();
    double total = 0;
    for(int i = 0; i < n; ++i) {
        total += x[i];
    }
    return total;
}')
```

Версия функции в C++ очень похожа на реализацию в R, за исключением следующих моментов:

- для нахождения длины вектора используется метод `.size()`, возвращающий целочисленное значение. Методы в C++ вызываются с помощью точечной нотации;
- инструкция цикла `for` имеет отличный от R синтаксис, а именно `for(init; check; increment)`. Наш цикл инициализируется путем создания новой переменной `i` со значением 0. Перед выполнением каждой итерации

мы проверяем условие `i < n` и прерываем цикл, если оно не выполняется. По окончании итерации мы увеличиваем значение переменной `i` на единицу с использованием специального префиксного оператора `++`;

- в C++ индексы векторов начинаются с нуля, а значит, индекс последнего элемента равен `n - 1`. Я повторю это еще разок, поскольку это очень важно: **В C++ ИНДЕКСЫ ВЕКТОРОВ НАЧИНАЮТСЯ С НУЛЯ**! Это одна из самых распространенных ошибок при конвертации кода из R в C++;
- для присвоения используется оператор `=`, а не `<-`;
- в C++ представлены операторы, модифицирующие переменные на месте: запись `total += x[i]` эквивалентна `total = total + x[i]`. Также модифицирующими на месте операторами являются `-=`, `*=` и `/=`.

Это хороший пример, демонстрирующий преимущества C++ в быстродействии. Как видно в приведенном ниже фрагменте кода, функция `sumC()` сравнима по скорости со встроенной и хорошо оптимизированной функцией `sum()`, тогда как функция `sumR()` значительно им уступает.

```
x <- runif(1e3)
bench::mark(
    sum(x),
    sumC(x),
    sumR(x)
)[1:6]
#> # A tibble: 3 x 6
#>   expression      min     mean   median      max `itr/sec`
#>   <chr>      <bch:tm> <bch:tm> <bch:tm> <bch:tm>     <dbl>
#> 1 sum(x)       1.13µs   1.21µs   1.17µs   17.2µs   823472.
#> 2 sumC(x)      2.52µs   4.53µs   4.99µs  775.4µs   220921.
#> 3 sumR(x)     42.41µs  46.03µs   43.2µs  137.9µs    21723.
```

25.2.4 Векторный вход, векторный выход

Теперь напишем функцию, вычисляющую евклидово расстояние между значением и вектором значений:

```
pdistR <- function(x, ys) {
    sqrt((x - ys) ^ 2)
}
```

В функции, написанной на языке R, не вполне очевидно, что в аргументе `x` мы ждем скалярную величину, и это нужно прописать в документации. В C++ таких неудобств нет, поскольку в нем все без исключения строго типизировано:

```
cppFunction('NumericVector pdistC(double x, NumericVector ys) {
    int n = ys.size();
    NumericVector out(n);
```

```
    for(int i = 0; i < n; ++i) {
        out[i] = sqrt(pow(ys[i] - x, 2.0));
    }
    return out;
}')
```

В этой функции представлена пара новых концепций:

- мы создаем новый числовой вектор длины `n` с помощью конструктора `NumericVector out(n)`. Также мы могли бы получить новый вектор путем копирования существующего: `NumericVector zs = clone(ys);`
- в C++ для возведения в степень используется функция `pow()`, а не оператор `^`.

Обратите внимание на то, что, поскольку наша функция R полностью векторизована, она будет несильно уступать в скорости функции C++.

```
y <- runif(1e6)
bench::mark(
    pdistR(0.5, y),
    pdistC(0.5, y)
)[1:6]
#> # A tibble: 2 x 6
#>   expression          min     mean   median      max `itr/sec`
#>   <chr>          <bch:tm> <bch:tm> <bch:tm> <bch:tm>     <dbl>
#> 1 pdistR(0.5, y)   5.21ms   5.57ms   5.24ms  10.89ms      180.
#> 2 pdistC(0.5, y)   2.31ms   2.48ms   2.39ms   3.39ms      404.
```

На моем компьютере время обработки вектора из 1 млн элементов составило около 5 мс. Функция, написанная на C++, справилась в 2,5 раза быстрее, примерно за 2 мс, но с учетом тех десяти минут, которые у меня ушли на ее написание, можно сказать, что вам нужно воспользоваться этой функцией не менее 200 тысяч раз, чтобы окупить время на ее создание. Причины скорости работы кода на языке C++ кроются в более эффективной работе с памятью. Версия кода на R вынуждена создавать промежуточный вектор той же длины, что и `y` (`x - ys`), а выделение памяти – операция довольно дорогостоящая. Функция C++ таких недостатков лишена, поскольку в ней используются промежуточные скаляры.

25.2.5 Использование sourceCpp

До сих пор мы использовали код на языке C++, встраиваемый с помощью функции `cppFunction()`. Это удобно с точки зрения демонстрации, но для решения реальных задач лучше использовать отдельные файлы C++, импортируемые в код на R с помощью функции `sourceCpp()`. Это позволяет писать код на C++ в подходящем для этого редакторе с подсветкой синтаксиса, а также облегчает задачу поиска ошибок компиляции по номерам строк кода.

Сохраненные файлы с кодом C++ должны иметь расширение `.cpp` и начинаться следующими инструкциями:

```
#include <Rcpp.h>
using namespace Rcpp;
```

Кроме того, каждую функцию, которая должна быть доступна в коде R, необходимо предварять следующей строкой:

```
// [[Rcpp::export]]
```

Если вы знакомы с пакетом `roxygen2`, то можете задаться вопросом, связано ли это как-то с `@export`. `Rcpp::export` отслеживает экспорт функций из C++ в R, тогда как `@export` отвечает за контроль экспорта функции из пакета и его доступ пользователю.

Вы можете встраивать код на языке R в C++ с помощью специальных блоков комментариев. Это бывает очень удобно при необходимости выполнения проверок:

```
/*** R
# This is R code
*/
```

Код на R запускается с помощью функции `source(echo = TRUE)`, так что вам нет необходимости осуществлять вывод явным образом.

Для компиляции кода на C++ вы можете воспользоваться вызовом `sourceCpp("path/to/file.cpp")`. Это приведет к созданию соответствующих функций в R и их добавлению в вашу текущую сессию. Обратите внимание, что эти функции не могут быть сохранены в файл `.Rdata` и загружены в одной из следующих сессий, – они должны создаваться заново при каждом запуске R.

К примеру, запуск приведенного ниже кода с помощью функции `sourceCpp()` приведет к реализации функции расчета среднего на C++ и его сравнению со встроенной функцией `mean()`:

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double meanC(NumericVector x) {
    int n = x.size();
    double total = 0;

    for(int i = 0; i < n; ++i) {
        total += x[i];
    }
    return total / n;
}
```

```
/*** R
x <- runif(1e5)
bench::mark(
    mean(x),
    meanC(x)
)
*/
```

Примечание. Если вы запустите этот код, то увидите, что функция `meanC()` значительно превосходит в быстродействии функцию `mean()`. Причина в том, что она жертвует точностью вычислений ради скорости.

В оставшейся части главы код на языке C++ будет представлен в виде отдельных файлов, а не в виде вызовов функции `cppFunction()`. Если вы хотите попробовать скомпилировать и/или модифицировать примеры, то можете скопировать код в файл C++, включающий в себя вспомогательные строки, показанные выше. Это можно легко сделать с помощью RMarkdown: все, что вам нужно сделать, – это написать `engine = "Rcpp"`.

25.2.6 Упражнения

1. Теперь, когда вы знаете самые основы C++, пришло время попрактиковаться в чтении и написании простых функций на этом языке. Просмотрите функции, приведенные ниже, и подумайте, какие базовые функции R могут им соответствовать. Вы можете не понимать код досконально, но вполне можете догадаться о его предназначении.

```
double f1(NumericVector x) {
    int n = x.size();
    double y = 0;

    for(int i = 0; i < n; ++i) {
        y += x[i] / n;
    }
    return y;
}

NumericVector f2(NumericVector x) {
    int n = x.size();
    NumericVector out(n);

    out[0] = x[0];
    for(int i = 1; i < n; ++i) {
        out[i] = out[i - 1] + x[i];
    }
    return out;
}
```

```
bool f3(LogicalVector x) {
    int n = x.size();

    for(int i = 0; i < n; ++i) {
        if (x[i]) return true;
    }
    return false;
}

int f4(Function pred, List x) {
    int n = x.size();

    for(int i = 0; i < n; ++i) {
        LogicalVector res = pred(x[i]);
        if (res[0]) return i + 1;
    }
    return 0;
}

NumericVector f5(NumericVector x, NumericVector y) {
    int n = std::max(x.size(), y.size());
    NumericVector x1 = rep_len(x, n);
    NumericVector y1 = rep_len(y, n);

    NumericVector out(n);

    for (int i = 0; i < n; ++i) {
        out[i] = std::min(x1[i], y1[i]);
    }
    return out;
}
```

2. Для приобретения навыка написания кода на C++ попробуйте переписать перечисленные ниже функции на этот язык. Пока будем допускать, что входные параметры не могут иметь пропущенных значений.
   2.1. `all()`.
   2.2. `cumprod()`, `cummin()`, `cummax()`.
   2.3. `diff()`. Начните с предположения о единичном интервале (аргумент `lag`), после чего обобщите функцию для интервала `n`.
   2.4. `range()`.
   2.5. `var()`. Почитайте в интернете, какие подходы можно применить при расчете дисперсии: `https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance`. Всякий раз при реализации численных алгоритмов полезно узнавать, как принято решать поставленную задачу.

25.3 Другие классы

Мы уже рассмотрели базовые классы векторов (`IntegerVector`, `NumericVector`, `LogicalVector`, `CharacterVector`) и их скалярные эквиваленты (`int`, `double`, `bool`, `String`). Пакет `Rcpp` также предоставляет обертки для всех остальных базовых типов данных. Наиболее важными из них являются обертки для списков и датафреймов, а также для функций и атрибутов, о которых мы поговорим далее. Кроме того, в `Rcpp` представлены классы для других типов, таких как `Environment`, `DottedPair`, `Language`, `Symbol` и др., но их описание выходит за рамки данной главы.

25.3.1 Списки и датафреймы

Пакет `Rcpp`, помимо прочего, предлагает классы `List` и `DataFrame`, но они более полезны для выхода, а не для входа. Причина в том, что списки и датафреймы могут содержать произвольные классы, но C++ необходимо знать их классы заранее.

Если список обладает известной структурой (например, это объект S3), вы можете извлечь его компоненты и вручную преобразовать в эквиваленты C++ с помощью функции `as()`. К примеру, объект, созданный функцией `lm()`, предназначенной для подгонки линейной модели, представляет собой список, компоненты которого всегда представлены одним типом данных. В коде, приведенном ниже, показано, как можно извлечь величину *средней процентной ошибки* (mean percentage error – MPE) линейной модели. Это не самый подходящий пример использования кода C++, поскольку эти операции легко реализуются в R, но он демонстрирует работу с важным классом S3. Обратите внимание на использование метода `.inherits()` и функции `stop()` для проверки того, что входящий объект представляет собой линейную модель.

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double mpe(List mod) {
    if (!mod.inherits("lm")) stop("Input must be a linear model");

    NumericVector resid = as<NumericVector>(mod["residuals"]);
    NumericVector fitted = as<NumericVector>(mod["fitted.values"]);

    int n = resid.size();
    double err = 0;
    for(int i = 0; i < n; ++i) {
        err += resid[i] / (fitted[i] + resid[i]);
    }
```

```
    return err / n;
}

mod <- lm(mpg ~ wt, data = mtcars)
mpe(mod)
#> [1] -0.0154
```

25.3.2 Функции

Вы можете передавать функции R в виде объекта типа `Function`. Это упрощает процесс вызова функций R из C++. Единственная сложность заключается в том, что мы не знаем, объект какого типа вернет функция, так что воспользуемся обобщенным типом `RObject`.

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
RObject callWithOne(Function f) {
    return f(1);
}

callWithOne(function(x) x + 1)
#> [1] 2
callWithOne(paste)
#> [1] "1"
```

Вызов функций R с позиционными аргументами выполняется как обычно:

```
f("y", 1);
```

Однако для передачи именованных аргументов вам потребуется воспользоваться особым синтаксисом:

```
f(_["x"] = "y", _["value"] = 1);
```

25.3.3 Атрибуты

Все объекты в R располагают атрибутами, которые можно получить и модифицировать при помощи метода `.attr()`. В пакете `Rcpp` также представлен метод `.names()` в качестве псевдонима для атрибута с именем. В следующем фрагменте кода показано использование этих методов. Обратите внимание на использование метода `::create()`, являющегося методом класса. Это позволяет создать вектор R из скалярных значений C++:

```
#include <Rcpp.h>
using namespace Rcpp;
```

```
// [[Rcpp::export]]
NumericVector attribs() {
    NumericVector out = NumericVector::create(1, 2, 3);

    out.names() = CharacterVector::create("a", "b", "c");
    out.attr("my-attr") = "my-value";
    out.attr("class") = "my-class";

    return out;
}
```

Применительно к объектам S4 роль, аналогичную `.attr()`, выполняет метод `.slot()`.

25.4 Пропущенные значения

При работе с пропущенными значениями вам необходимо знать следующие две вещи:

- как пропущенные значения R ведут себя в скалярах C++ (например, `double`);
- как извлекать и устанавливать пропущенные значения в векторах (например, `NumericVector`).

25.4.1 Скаляры

В следующем фрагменте кода показано, что произойдет, если взять одно пропущенное значение в R, привести его к скалярному значению, а затем – обратно в вектор R. Заметьте, что такая процедура бывает очень полезна при определении того, что делает та или иная операция.

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
List scalar_missings() {
    int int_s = NA_INTEGER;
    String chr_s = NA_STRING;
    bool lgl_s = NA_LOGICAL;
    double num_s = NA_REAL;

    return List::create(int_s, chr_s, lgl_s, num_s);
}

str(scalar_missings())
#> List of 4
#>  $ : int NA
```

```
#>  $ : chr NA
#>  $ : logi TRUE
#>  $ : num NA
```

За исключением типа bool, здесь все выглядит неплохо: все пропущенные значения были сохранены. Однако, как мы увидим в следующих разделах, и здесь все не так просто, как кажется.

25.4.1.1 Целые числа

Применительно к целым числам пропущенные значения сохраняются в виде минимального целочисленного представления. Если с ними ничего не делать, они просто сохранятся как есть. Однако в связи с тем, что C++ не знает о таком особом поведении минимально допустимого целочисленного типа, при выполнении любого действия с этим значением мы с большой вероятностью получим некорректный результат: например, выражение evalCpp('NA_INTEGER + 1') дает –2147483647.

Таким образом, если вы хотите работать с пропущенными значениями в виде целых чисел, либо используйте тип IntegerVector длины 1, либо проявляйте большую осторожность.

25.4.1.2 Числа двойной точности

При работе с типом двойной точности вы можете спокойно игнорировать пропущенные значения и оперировать значениями NaN. Причина в том, что тип NA в R представляет собой частный случай числа с плавающей запятой NaN, определенного в стандарте IEEE 754. Таким образом, любая логическая операция, в которой задействован NaN (или, в случае с C++, NAN), всегда будет возвращать значение FALSE:

```
evalCpp("NAN == 1")
#> [1] FALSE
evalCpp("NAN < 1")
#> [1] FALSE
evalCpp("NAN > 1")
#> [1] FALSE
evalCpp("NAN == NAN")
#> [1] FALSE
```

(Здесь я воспользовался функцией evalCpp(), позволяющей выводить результат запуска одного выражения C++, что делает ее идеально подходящей для таких интерактивных экспериментов.)

Но будьте осторожны при комбинировании с булевыми значениями:

```
evalCpp("NAN && TRUE")
#> [1] TRUE
evalCpp("NAN || FALSE")
#> [1] TRUE
```

Что касается числового контекста вычислений, пропущенные значения будут сохраняться:

```
evalCpp("NAN + 1")
#> [1] NaN
evalCpp("NAN - 1")
#> [1] NaN
evalCpp("NAN / 1")
#> [1] NaN
evalCpp("NAN * 1")
#> [1] NaN
```

25.4.2 Строки

`String` представляет собой скалярный строковый тип из пакета `Rcpp`, так что с ним никаких проблем с пропущенными значениями возникать не должно.

25.4.3 Булевы значения

Если тип данных `bool` в C++ подразумевает два возможных значения (`true` или `false`), то логический вектор в R представляет три значения (`TRUE`, `FALSE` и `NA`). Если вы выполняете приведение логического вектора единичной длины, убедитесь, что в нем не содержатся пропущенные значения. В противном случае они будут преобразованы в значения `TRUE`. Лучше вместо этого использовать тип `int`, поскольку он может представлять значения `TRUE`, `FALSE` и `NA`.

25.4.4 Векторы

При работе с векторами вам необходимо использовать пропущенные значения, специфичные для каждого типа вектора (`NA_REAL`, `NA_INTEGER`, `NA_LOGICAL`, `NA_STRING`):

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
List missing_sampler() {
    return List::create(
        NumericVector::create(NA_REAL),
        IntegerVector::create(NA_INTEGER),
        LogicalVector::create(NA_LOGICAL),
        CharacterVector::create(NA_STRING)
    );
}

str(missing_sampler())
#> List of 4
#> $ : num NA
```

```
#> $ : int NA
#> $ : logi NA
#> $ : chr NA
```

25.4.5 Упражнения

1. Перепишите любую функцию из раздела 25.2.6 таким образом, чтобы она могла работать с пропущенными значениями. Если в аргументе `na.rm` передано значение `True`, игнорируйте пропущенные значения. В противном случае возвращайте пропущенные значения, если они содержатся на входе. Вы можете попрактиковаться с функциями `min()`, `max()`, `range()`, `mean()` и `var()`.
2. Перепишите функции `cumsum()` и `diff()` таким образом, чтобы они могли обрабатывать пропущенные значения. Обратите внимание, что эти функции обладают более сложным поведением.

25.5 Стандартная библиотека шаблонов

Истинная мощь языка C++ проявляется при необходимости реализации довольно сложных алгоритмов. *Стандартная библиотека шаблонов* (standard template library – STL) содержит в себе большой набор чрезвычайно полезных структур данных и алгоритмов. В этом разделе мы поговорим о наиболее полезных из них, а также я подскажу вам, где искать информацию по остальным. Я не могу научить вас всему, что вам необходимо знать об STL, но представленные в книге примеры продемонстрируют всю мощь этой библиотеки, а остальное вы сможете добрать сами.

Если вам понадобятся алгоритмы или структуры данных, не реализованные в STL, вы можете начать поиск с сайта https://www.boost.org. Описание процесса установки `boost` на компьютере выходит за рамки данной главы, но после инсталляции вы сможете воспользоваться структурами данных и алгоритмами представленных библиотек, вставив в файл соответствующий заголовок, например `#include <boost/array.hpp>`.

25.5.1 Использование итераторов

Итераторы (iterator) интенсивно используются в STL: многие функции в библиотеке либо принимают на вход, либо возвращают итераторы. Итераторы являются следующим шагом после базовых циклов, и они полностью абстрагируются от лежащих в их основе структур данных. Для итераторов характерны три следующих оператора:

- продвижение вперед с помощью оператора `++`;
- получение значения, на которое ссылается итератор, или *разыменование* (dereference), с помощью оператора `*`;
- сравнение с помощью оператора `==`.

К примеру, мы могли бы переписать нашу функцию суммирования с использованием итераторов следующим образом:

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double sum3(NumericVector x) {
    double total = 0;

    NumericVector::iterator it;
    for(it = x.begin(); it != x.end(); ++it) {
        total += *it;
    }
    return total;
}
```

Главные отличия от предыдущей версии функции содержатся в цикле `for`:

- мы начинаем движение с `x.begin()`, а заканчиваем, когда достигнем `x.end()`. Небольшая оптимизация может заключаться в сохранении элемента итератора, чтобы не нужно было его искать каждый раз. Это позволяет сэкономить всего около 2 нс на итерацию, так что это может быть важно только в случае очень простых вычислений в итерациях;
- вместо индексирования `x` мы используем разыменовывающий оператор `*it` для получения текущего значения итератора;
- обратите внимание на тип итератора: `NumericVector::iterator`. Каждому типу вектора соответствует свой тип итератора: `LogicalVector::iterator`, `CharacterVector::iterator` и т. д.

Этот код можно еще больше упростить, воспользовавшись циклами `for` на основе диапазонов, появившимися в стандарте C++11. Этот стандарт общедоступен и может быть легко активирован для работы с пакетом `Rcpp` путем добавления инструкции `[[Rcpp::plugins(cpp11)]]`.

```
// [[Rcpp::plugins(cpp11)]]
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double sum4(NumericVector xs) {
    double total = 0;

    for(const auto &x : xs) {
        total += x;
    }
    return total;
}
```

Итераторы также позволяют воспользоваться эквивалентами функций семейства `apply` в C++. К примеру, мы можем еще раз переписать функцию

суммирования с использованием функции accumulate(), которая принимает начало и конец итератора и суммирует все значения в векторе. Третьим аргументом функция принимает исходное значение, которое также определяет тип данных, используемый функцией для агрегации (мы использовали значение 0.0, а не 0, чтобы операции производились с типом double, а не int). Для применения функции accumulate() нам необходимо включить в заголовки <numeric>, как показано ниже.

```
#include <numeric>
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double sum5(NumericVector x) {
    return std::accumulate(x.begin(), x.end(), 0.0);
}
```

25.5.2 Алгоритмы

Заголовок <algorithm> позволяет воспользоваться большим разнообразием алгоритмов, работающих с итераторами. Подробнее об этой библиотеке можно почитать по ссылке https://en.cppreference.com/w/cpp/algorithm. К примеру, мы можем написать базовую Rcpp-версию функции findInterval(), которая будет принимать два аргумента – вектор значений и вектор границ интервалов, – и возвращать принадлежность всех элементов x корзинам. В ней показаны некоторые важные особенности работы с итераторами. Прочитайте код, приведенный ниже, и убедитесь, что поняли, как он работает.

```
#include <algorithm>
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
IntegerVector findInterval2(NumericVector x, NumericVector breaks) {
    IntegerVector out(x.size());

    NumericVector::iterator it, pos;
    IntegerVector::iterator out_it;

    for(it = x.begin(), out_it = out.begin(); it != x.end(); ++it, ++out_it) {
        pos = std::upper_bound(breaks.begin(), breaks.end(), *it);
        *out_it = std::distance(breaks.begin(), pos);
    }
    return out;
}
```

Ключевые моменты здесь следующие:

- мы осуществляем проход по двум итераторам (входному и выходному) одновременно;
- мы можем присваивать значения выходному итератору (`out_it`) с помощью оператора разыменования для изменения значений в переменной `out`;
- функция `upper_bound()` возвращает итератор. Если бы нам нужны были значения из `upper_bound()`, мы могли бы воспользоваться оператором разыменования; для определения его местоположения мы используем функцию `distance()`;
- небольшое замечание: если бы мы хотели, чтобы наша функция не уступала в скорости функции `findInterval()` в R (которая написана на C), мы бы рассчитали вызовы `.begin()` и `.end()` лишь раз и сохранили результаты. Сделать это несложно, но мы решили не акцентировать на этом внимание в данном примере. А так мы могли бы получить функцию, превосходящую по скорости функцию `findInterval()`, но занимающую примерно десятую часть ее кода.

Обычно лучше использовать специальные алгоритмы из библиотеки STL, чем писать циклы вручную. В книге Скотта Майерса *Effective STL* приведены три причины этого: эффективность, точность и сопровождаемость. Алгоритмы, присутствующие в STL, написаны экспертами в области C++ и обладают очень высоким быстродействием. Кроме того, им уже много лет, и за это время они успешно прошли массовое тестирование. Также использование стандартных алгоритмов позволяет сделать код более понятным, что облегчает его поддержку и развитие.

25.5.3 Структуры данных

Библиотека STL предоставляет большой набор структур данных, включая следующие: `array`, `bitset`, `list`, `forward_list`, `map`, `multimap`, `multiset`, `priority_queue`, `queue`, `deque`, `set`, `stack`, `unordered_map`, `unordered_set`, `unordered_multimap`, `unordered_multiset` и `vector`. Наиболее важными из них являются `vector`, `unordered_set` и `unordered_map`. В этом разделе мы поработаем с ними, но с остальными структурами вы можете работать похожим образом с учетом компромиссов, связанных с производительностью. К примеру, структура данных `deque` обладает очень схожим интерфейсом с векторами, но обладает несколько иной реализацией, что сказывается на производительности. Вы всегда можете воспользоваться ей для решения своих задач. Хорошее руководство по структурам данных STL находится по адресу https://en.cppreference.com/w/cpp/container, и я рекомендую всегда держать его под рукой при работе с этой библиотекой.

Пакет `Rcpp` умеет выполнять преобразования из многих структур данных STL в соответствующие аналоги в R, так что вы можете возвращать их из ваших функций, не утруждая себя явным преобразованием к формату R.

25.5.4 Векторы

Векторы STL очень похожи на векторы в R, за исключением того, что они эффективно расширяются. Это делает их полезными для использования в ситуациях, когда вы не знаете заранее, какого размера будет вывод. Векторы STL используют шаблоны – это означает, что при создании вектора вам необходимо указать тип объектов, которые он будет хранить: `vector<int>`, `vector<bool>`, `vector<double>`, `vector<String>`. К отдельным элементам вектора можно обращаться с использованием стандартной нотации `[]`, а для добавления элемента в конец вектора можно воспользоваться методом `.push_back()`. Если вы знаете заранее, какого размера будет ваш вектор, вы можете использовать метод `.reserve()` для выделения достаточного места в памяти.

В коде, приведенном ниже, реализован метод кодирования на основе длин серий (`rle()`). На выходе мы получаем два вектора: вектор значений (`values`) и вектор длин серий (`lengths`) с информацией о том, сколько раз повторялся каждый элемент. Функция работает путем прохода по входному вектору `x` и сравнения каждого следующего значения с предыдущим. Если значения совпадают, увеличивается последнее значение в векторе `lengths`. В противном случае значение добавляется в конец вектора `values`, и соответствующая ему длина серии инициализируется единицей.

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
List rleC(NumericVector x) {
    std::vector<int> lengths;
    std::vector<double> values;

    // Инициализируем первое значение
    int i = 0;
    double prev = x[0];
    values.push_back(prev);
    lengths.push_back(1);

    NumericVector::iterator it;
    for(it = x.begin() + 1; it != x.end(); ++it) {
        if (prev == *it) {
            lengths[i]++;
        } else {
            values.push_back(*it);
            lengths.push_back(1);
            i++;
            prev = *it;
        }
    }

    return List::create(
        _["lengths"] = lengths,
```

```
        _["values"] = values
    );
}
```

(В альтернативной реализации функции мы могли бы заменить `i` на итератор `lengths.rbegin()`, всегда указывающий на последний элемент вектора. Вы можете попробовать сами реализовать этот вариант.)

С другими методами, присутствующими у векторов, вы можете ознакомиться по адресу https://en.cppreference.com/w/cpp/container/vector.

25.5.5 Множества

Множества (set) содержат список уникальных значений и могут быстро сообщать вам о присутствии в списке того или иного значения. Их бывает удобно использовать в алгоритмах, связанных с поиском дубликатов или уникальных значений (как в `unique`, `duplicated` или `in`). В C++ представлены как упорядоченные множества (`std::set`), так и неупорядоченные (`std::unordered_set`), и вы можете использовать их по своему усмотрению в зависимости от требований. Неупорядоченные множества работают намного быстрее (поскольку внутри они используют кеш-таблицы, а не деревья), так что если вам нужно упорядоченное множество, рассмотрите вариант использования неупорядоченного с последующей сортировкой. Подобно векторам, множества используют шаблоны, в связи с чем вам необходимо указывать конкретный тип хранящихся в них элементов: `unordered_set<int>`, `unordered_set<bool>` и т. д. Больше подробностей по работе со множествами можно узнать по адресам https://en.cppreference.com/w/cpp/container/set и https://en.cppreference.com/w/cpp/container/unordered_set.

В функции, показанной ниже, используется неупорядоченное множество с целью реализации аналога функции `duplicated()` для целочисленных векторов. Обратите внимание на использование конструкции `seen.insert(x[i]).second`. Функция `insert()` возвращает пару значений: `.first` – это итератор, указывающий на элемент, а `.second` – булево значение, установленное в `true`, если мы имеем дело с новым значением во множестве.

```
// [[Rcpp::plugins(cpp11)]]
#include <Rcpp.h>
#include <unordered_set>
using namespace Rcpp;

// [[Rcpp::export]]
LogicalVector duplicatedC(IntegerVector x) {
    std::unordered_set<int> seen;
    int n = x.size();
    LogicalVector out(n);

    for (int i = 0; i < n; ++i) {
        out[i] = !seen.insert(x[i]).second;
    }
```

```
    return out;
}
```

25.5.6 Ассоциативные массивы

Ассоциативные массивы (map) близки по смыслу к множествам, но вместо информации о присутствии или отсутствии элементов в структуре они хранят дополнительные данные, связанные с элементами. Такие массивы удобно использовать в функциях вроде `table()` или `match()`, требующих поиска значений. Как и в случае со множествами, бывают упорядоченные ассоциативные массивы (`std::map`) и неупорядоченные (`std::unordered_map`). Поскольку ассоциативные массивы состоят из ключей и значений, при создании вам необходимо указать типы данных для обеих составляющих: `map<double, int>`, `unordered_map<int, double>` и т. д. В приведенном ниже фрагменте кода показан пример реализации функции `table()` для числовых векторов с использованием ассоциативных массивов:

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
std::map<double, int> tableC(NumericVector x) {
    std::map<double, int> counts;

    int n = x.size();
    for (int i = 0; i < n; i++) {
        counts[x[i]]++;
    }

    return counts;
}
```

25.5.7 Упражнения

Для приобретения навыков работы с алгоритмами и структурами данных STL реализуйте следующие функции R в C++, воспользовавшись советами.

1. `median.default()` с использованием `partial_sort`.
2. `%in%` с применением `unordered_set` и методов `find()` или `count()`.
3. `unique()` с использованием `unordered_set` (дополнение: сделайте все в одной строке!).
4. `min()` с применением `std::min()` или `max()` с использованием `std::max()`.
5. `which.min()` с применением `min_element()` или `which.max()` с использованием `max_element()`.
6. `setdiff()`, `union()` и `intersect()` для целых чисел с использованием упорядоченных диапазонов, а также `set_union()`, `set_intersection()` и `set_difference()`.

25.6 Практические примеры

В следующих разделах мы продемонстрируем несколько примеров практического применения фрагментов кода на C++ для замены медленного кода на R.

25.6.1 Семплирование по Гиббсу

Пример, который мы будем обсуждать в этой главе, является продолжением темы, описанной Дирком Эддельбуэттелем (Dirk Eddelbuettel) в своем блоге на странице http://dirk.eddelbuettel.com/blog/2011/07/14 и посвященной переносу процедуры семплирования по Гиббсу (Gibbs sampler) из R в C++. Фрагменты кода на R и C++, показанные ниже, будут очень похожи (мне понадобилось всего несколько минут, чтобы выполнить это преобразование), но разница в скорости составила на моем компьютере примерно 20 раз. Пост Дирка содержит также пример оптимизации кода с использованием более быстрых функций генерации случайных чисел из библиотеки GSL (доступной в R посредством пакета `RcppGSL`), которые позволили сделать код еще быстрее.

Код на R получился таким:

```
gibbs_r <- function(N, thin) {
    mat <- matrix(nrow = N, ncol = 2)
    x <- y <- 0

    for (i in 1:N) {
        for (j in 1:thin) {
            x <- rgamma(1, 3, y * y + 4)
            y <- rnorm(1, 1 / (x + 1), 1 / sqrt(2 * (x + 1)))
        }
        mat[i, ] <- c(x, y)
    }
    mat
}
```

Этот код легко перевести на язык C++. Для этого нам понадобится:

- добавить типы данных для всех переменных;
- воспользоваться скобками `(` вместо `[` для индексирования матриц;
- поместить результаты выполнения функций `rgamma` и `rnorm` в индексы для преобразования вектора в скаляр.

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericMatrix gibbs_cpp(int N, int thin) {
    NumericMatrix mat(N, 2);
    double x = 0, y = 0;
```

```
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < thin; j++) {
            x = rgamma(1, 3, 1 / (y * y + 4))[0];
            y = rnorm(1, 1 / (x + 1), 1 / sqrt(2 * (x + 1)))[0];
        }
        mat(i, 0) = x;
        mat(i, 1) = y;
    }

    return(mat);
}
```

Проверим на быстродействие обе реализации:

```
bench::mark(
    gibbs_r(100, 10),
    gibbs_cpp(100, 10),
    check = FALSE
)
#> # A tibble: 2 x 10
#>   expression      min     mean   median    max `itr/sec` mem_alloc
#>   <chr>      <bch:tm> <bch:tm> <bch:tm> <bch:>     <dbl> <bch:byt>
#> 1 gibbs_r(1…   4.25ms   5.53ms   4.96ms 9.38ms      181.    4.97MB
#> 2 gibbs_cpp… 223.76µs 259.25µs 255.99µs 1.17ms     3857.     4.1KB
#> # … with 3 more variables: n_gc <dbl>, n_itr <int>,
#> # total_time <bch:tm>
```

25.6.2 Векторизация в R против векторизации в C++

Этот пример адаптирован из поста *Rcpp is smoking fast for agent-based models in data frames* (https://gweissman.github.io/post/rcpp-is-smoking-fast-for-agent-based-models-in-data-frames). Задача состоит в предсказании отклика модели на основании трех входных параметров. Версия функции в базовом R выглядит так:

```
vacc1a <- function(age, female, ily) {
    p <- 0.25 + 0.3 * 1 / (1 - exp(0.04 * age)) + 0.1 * ily
    p <- p * if (female) 1.25 else 0.75
    p <- max(0, p)
    p <- min(1, p)
    p
}
```

Нам необходимо иметь возможность применять эту функцию к разным входным данным, так что мы могли бы написать версию функции на основе векторов с использованием цикла `for`.

```
vacc1 <- function(age, female, ily) {
    n <- length(age)
    out <- numeric(n)
```

```
    for (i in seq_len(n)) {
        out[i] <- vacc1a(age[i], female[i], ily[i])
    }
    out
}
```

Если вы знакомы с R, то должны с легкостью предсказать, что такая функция будет работать очень медленно. И это так и есть. Есть два способа справиться с этим недугом. При наличии богатого лексикона R вы могли бы с легкостью векторизовать эту функцию с помощью функций `ifelse()`, `pmin()` и `pmax()`. Также можно было бы переписать функции `vacc1a()` и `vacc1()` на C++, помня о том, что циклы и вызовы функций в этом языке обладают гораздо меньшими накладными расходами по сравнению с R.

Оба способа довольно просты для реализации. Вариант с R:

```
vacc2 <- function(age, female, ily) {
    p <- 0.25 + 0.3 * 1 / (1 - exp(0.04 * age)) + 0.1 * ily
    p <- p * ifelse(female, 1.25, 0.75)
    p <- pmax(0, p)
    p <- pmin(1, p)
    p
}
```

(Если вы можете похвастаться приличным опытом работы в R, то наверняка заметили узкое место в этом коде. Дело в том, что функции `ifelse`, `pmin` и `pmax` отличаются низким быстродействием, и их можно было бы заменить на `p * 0.75 + p * 0.5 * female`, `p[p < 0] <- 0` и `p[p > 1] <- 1`. Вы можете внедрить эти изменения самостоятельно и произвести замер скорости.)

Вариант с C++:

```
#include <Rcpp.h>
using namespace Rcpp;

double vacc3a(double age, bool female, bool ily){
    double p = 0.25 + 0.3 * 1 / (1 - exp(0.04 * age)) + 0.1 * ily;
    p = p * (female ? 1.25 : 0.75);
    p = std::max(p, 0.0);
    p = std::min(p, 1.0);
    return p;
}

// [[Rcpp::export]]
NumericVector vacc3(NumericVector age, LogicalVector female,
                    LogicalVector ily) {
    int n = age.size();
    NumericVector out(n);

    for(int i = 0; i < n; ++i) {
        out[i] = vacc3a(age[i], female[i], ily[i]);
    }
```

```
    return out;
}
```

Теперь сгенерируем тестовые данные и проверим, что все три версии возвращают одинаковый результат:

```
n <- 1000
age <- rnorm(n, mean = 50, sd = 10)
female <- sample(c(T, F), n, rep = TRUE)
ily <- sample(c(T, F), n, prob = c(0.8, 0.2), rep = TRUE)

stopifnot(
    all.equal(vacc1(age, female, ily), vacc2(age, female, ily)),
    all.equal(vacc1(age, female, ily), vacc3(age, female, ily))
)
```

Автор оригинального поста в блоге забыл это сделать, и его ошибка в версии кода на C++ (`0.004` вместо `0.04`) осталась незамеченной. Итак, сравним все три подхода:

```
bench::mark(
    vacc1 = vacc1(age, female, ily),
    vacc2 = vacc2(age, female, ily),
    vacc3 = vacc3(age, female, ily)
)
#> # A tibble: 3 x 10
#>   expression     min     mean   median     max `itr/sec` mem_alloc
#>   <chr>      <bch:t> <bch:tm> <bch:tm> <bch:t>     <dbl> <bch:byt>
#> 1 vacc1       1.62ms   1.81ms   1.79ms   2.6ms      552.    7.86KB
#> 2 vacc2      84.73µs 115.71µs 100.89µs 569.1µs     8642.     224KB
#> 3 vacc3      13.38µs  15.63µs   15.6µs  75.4µs    63988.   14.48KB
#> # … with 3 more variables: n_gc <dbl>, n_itr <int>,
#> # total_time <bch:tm>
```

Неудивительно, что наш исходный код с применением циклов оказался таким медленным. Векторизация в R дала ощутимый прирост производительности, но еще большей эффективности нам удалось добиться с использованием циклов в C++ (примерно в десять раз). Лично я был удивлен тем, насколько большим оказался скачок при применении C++, но причина оказалась в том, что в версии R создается 11 векторов для хранения промежуточных результатов, тогда как в C++ – только один.

25.7 Использование Rcpp в пакете

Тот же код на C++, который использовался с помощью функции `sourceCpp()`, может быть также упакован в пакет. Есть сразу несколько преимуществ перехода от использования отдельных файлов с исходным кодом C++ к пакетам.

1. Ваш код может быть доступен для пользователей, у которых не установлены инструменты разработки C++.
2. Разные исходные файлы и их зависимости будут находиться под управлением одной автоматизированной системы сборки пакетов R.
3. Пакеты предоставляют дополнительную инфраструктуру для тестирования, документации и целостности кода.

Для добавления `Rcpp` в существующий пакет вам необходимо поместить файлы с исходным кодом на C++ в директорию `src/` и создать или изменить следующие конфигурационные файлы:

- в `DESCRIPTION` добавить:

```
LinkingTo: Rcpp
Imports: Rcpp
```

- убедиться, что файл `NAMESPACE` содержит:

```
useDynLib(mypackage)
importFrom(Rcpp, sourceCpp)
```

Нужно импортировать что-нибудь (что угодно) из пакета `Rcpp`, чтобы внутренний код `Rcpp` был корректно загружен. Это баг, присутствующий в R, который, будем надеяться, будет устранен в будущем.

Простейшим способом автоматической настройки всего этого хозяйства является вызов функции `usethis::use_rcpp()`.

Перед сборкой пакета необходимо запустить функцию `Rcpp::compileAttributes()`. Эта функция сканирует файлы C++ на предмет атрибутов `Rcpp::export` и генерирует код, требующийся для того, чтобы функции были доступны в R. При добавлении, удалении или изменении сигнатуры функций вам необходимо повторно вызывать функцию `compileAttributes()`. При использовании пакета `devtools` или Rstudio это делается автоматически.

За дополнительной информацией можно обратиться к виньетке пакета `Rcpp`: `vignette("Rcpp-package")`.

25.8 Дополнительная литература для изучения

В данной главе мы обсудили лишь малую часть того, что связано с пакетом `Rcpp`, и познакомились с самыми основными инструментами для преобразования медленного кода на R в быстрый код на C++. Как мы уже упоминали, пакет `Rcpp` располагает множеством других механизмов для связывания кода на R с существующим кодом на C++, включая следующие:

- дополнительные возможности атрибутов, в том числе указание аргументов по умолчанию, привязка внешних зависимостей C++ и экс-

портирование интерфейсов C++ из пакетов. Подробнее об этих возможностях можно почитать в виньетке, посвященной атрибутам Rcpp: `vignette("Rcpp-attributes");`

- автоматическое создание оберток между структурами данных в R и C++, включая сопоставление классов C++ с классами на основе ссылок. С введением в эту тему можно ознакомиться в виньетке: `vignette("Rcpp-modules");`
- краткое руководство по пакету Rcpp (`vignette("Rcpp-quickref")`) содержит полезную информацию о классах Rcpp и их применении.

Я настоятельно рекомендую посетить домашнюю страницу пакета Rcpp (https://www.rcpp.org) и подписаться на список рассылки (https://lists.r-forge.r-project.org/cgi-bin/mailman/listinfo/rcpp-devel).

Другие полезные ресурсы для изучения языка C++:

- *Effective C++* [Майерс, 2005] и *Effective STL* [Майерс, 2001];
- *C++ Annotations* (http://www.icce.rug.nl/documents/cplusplus/cplusplus.html), предназначенный для опытных разработчиков на C (или на других языках со схожей с C грамматикой вроде Perl или Java), которые хотят переключиться на C++;
- *Algorithm Libraries* (http://www.cs.helsinki.fi/u/tpkarkka/alglib/k06), с более техническим, но при этом кратким описанием основных концепций библиотеки STL.

Необходимость написания высокопроизводительного кода может также привести к переосмыслению основных подходов к программированию, и знакомство с базовыми структурами данных и алгоритмами может вам в этом помочь. В этом смысле я бы порекомендовал также следующие источники: *Algorithm Design Manual* [Скиена (Skiena), 1998], *Introduction to Algorithms* от MIT (https://ocw.mit.edu/courses/6-046j-introduction-to-algorithms-sma-5503-fall-2005), *Algorithms* от Роберта Седжвика (Robert Sedgewick) и Кевина Уэйна (Kevin Wayne), для которой есть бесплатный онлайн-учебник (https://algs4.cs.princeton.edu/home) и соответствующий курс на Coursera (https://www.coursera.org/learn/algorithms-part1).

25.9 Благодарности

Я бы хотел поблагодарить список рассылки Rcpp за помощь в написании этой главы, а особенно Ромена Франсуа и Дирка Эддельбуэттеля, которые не только терпеливо отвечали на все мои вопросы, но и быстро реагировали на замечания по поводу работы пакета. Написание этой главы было бы невозможно без помощи Джей Джей Аллаира. Он вдохновил меня на изучение C++ и отвечал на все мои дурацкие вопросы в процессе.

Решения и комментарии к упражнениям

Ответы на упражнения из главы 2

2.2.2. Ответы на упражнения

1 Переменные `a`, `b` и `c` указывают на один и тот же объект (с одним адресом в памяти). Значение этого объекта – `1:10`. Переменная `d` указывает на другой объект с таким же значением.

```
list_of_names <- list(a, b, c, d)
obj_addrs(list_of_names)
#> [1] "0x7fb3c6ca4c10" "0x7fb3c6ca4c10" "0x7fb3c6ca4c10" "0x7fb3c7887b30"
```

2 Да, все они указывают на один и тот же объект. В этом можно убедиться, посмотрев на адреса в памяти объектов функций. Они совпадают:

```
mean_functions <- list(
    mean,
    base::mean,
    get("mean"),
    evalq(mean),
    match.fun("mean")
)

unique(obj_addrs(mean_functions))
#> [1] "0x7fb3c2a82160"
```

3 Имена колонок зачастую содержат данные, а преобразования, выполняемые функцией `make.names()`, необратимы, так что поведение по умолчанию может привести к повреждению данных. Во избежание этого можно воспользоваться аргументом `check.names = FALSE`.

4 Синтаксически правильные имена должны начинаться с буквы или точки (за которой не следует цифра) и могут содержать цифры и символы подчеркивания (символ "_" разрешено использовать начиная с R версии 1.9.0).

Три главных механизма, обеспечивающих создание синтаксически правильных имен (см. `?make.names`):

1. Имена, не начинающиеся с буквы или точки, будут предваряться символом "X".

```
make.names("")    # предваряющий "X"
#> [1] "X"
```

   То же самое правило распространяется на имена, начинающиеся с точки, за которой следует цифра.

```
make.names(".1")  # предваряющий "X"
#> [1] "X.1"
```

2. Дополнительно синтаксически неправильные символы заменяются на точку.

```
make.names("non-valid")  # замена на "."
#> [1] "non.valid"
make.names("@")          # предваряющий "X" + замена на "."
#> [1] "X."
make.names("  R")        # предваряющий "X" + замена на ".."
#> [1] "X..R"
```

3. Зарезервированные в R слова (см. `?reserved`) завершаются точкой.

```
make.names("if")  # завершающая "."
#> [1] "if."
```

   Некоторые изменения могут быть обусловлены текущей локалью. Из справки `?make.names`:

   > *Определение буквы зависит от текущей локали, и только цифры ASCII рассматриваются как цифры.*

5 Имя `.123e1` не является синтаксически правильным, поскольку оно начинается с точки, следом за которой идет цифра. Это превращает его в число 1,23.

2.3.6. Ответы на упражнения

1 При вызове `1:10` в памяти создается объект со своим адресом, но без привязки к имени. Таким образом, манипулировать этим объектом из R нельзя. Поскольку копия объекта создаваться не будет, отслеживать его для копирования бессмысленно.

```
obj_addr(1:10)  # Объект существует, но без имени
#> [1] "0x7fb3c4c4f388"
```

2 Изначально вектор `x` обладает целочисленным типом. Операция присваивания задает для третьего элемента вектора тип двойной точности, что запускает процесс *копирования при изменении* (copy-on-modify).

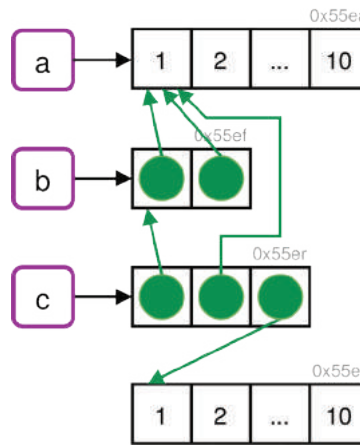
```
x <- c(1L, 2L, 3L)
tracemem(x)
#> <0x66a4a70>

x[[3]] <- 4
#> tracemem[0x55eec7b3af38 -> 0x55eec774cc18]:
```

Избежать этого можно, если явным образом указать целочисленный тип для присваиваемого значения:

```
x <- c(1L, 2L, 3L)
tracemem(x)
#> <0x55eec6940ae0>

x[[3]] <- 4L
```

Учтите, что при запуске этого кода в RStudio будут созданы дополнительные копии по причине наличия ссылок из панели окружения.

3 Переменная `a` содержит ссылку на адрес, по которому располагается значение `1:10`. Переменная `b` содержит список из двух ссылок на тот же адрес, на который ссылается переменная `a`. Переменная `c` содержит список из переменной `b` (в которой находятся две ссылки на `a`), переменной `a` (содержащей ту же ссылку) и ссылки на другой адрес, по которому располагается уже другое значение `1:10`.



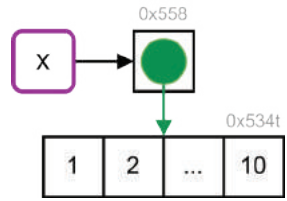
Мы можем проверить свои догадки, рассмотрев дерево ссылок в R.

```
ref(c)
#> ■ [1:0x55erc93cbdd8] <list>    # c
#> ├─■ [2:0x55efcb8246e8] <list>  # - b
#> │ ├─[3:0x55eac7df4e98] <int>   # -- a
#> │ └─[3:0x55eac7df4e98]         # -- a
```

```
#> ├─[3:0x55eac7df4e98]           # - a
#> └─[4:0x55etc7aa6968] <int>     # - 1:10
```

4 В исходном дереве ссылок переменной x видно, что имя x связано с объектом списка. Этот объект содержит ссылку на целочисленный вектор 1:10.

```
x <- list(1:10)

ref(x)
#> ■ [1:0x55853b74ff40] <list>
#> └─[2:0x534t3abffad8] <int>
```

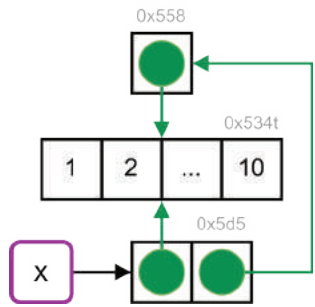


Когда x присваивается элементу самого себя, запускается процесс копирования при изменении, и список копируется по новому адресу в памяти.

```
tracemem(x)
x[[2]] <- x
#> tracemem[0x55853b74ff40 -> 0x5d553bacdcd8]:
```

Объект списка, ранее связанный с x, теперь ссылается на новый объект списка. Он больше не привязан к имени. В результате мы получим две ссылки на целочисленный вектор.

```
ref(x)
#> ■ [1:0x5d553bacdcd8] <list>
#> ├─[2:0x534t3abffad8] <int>
#> └─■ [3:0x55853b74ff40] <list>
#>     └─[2:0x534t3abffad8]
```



2.4.1. Ответы на упражнения

1 Функция `object.size()` не учитывает объем общих элементов в списке. Таким образом, будет получено увеличение в объеме в 100-кратном размере.

2 Все три функции встроены в R в составе пакетов {base} и {stats}, а значит, они доступны всегда. Теперь зададимся вопросом, что означает определить размер того, что уже включено в R.

(Есть и другой типовой вопрос, касающийся того, зачем именно вы хотите знать объем чего бы то ни было: вас интересует, сколько данных нужно будет послать по сети, если вы захотите с кем-то поделиться этим объектом (например, путем сериализации), или вы хотите знать, сколько памяти освободится при удалении объекта?)

Давайте попробуем узнать, к какому количеству объектов подобное применимо.

Следующие пакеты обычно загружаются по умолчанию:

```
base_pkgs <- c(
    "package:stats", "package:graphics", "package:grDevices",
    "package:utils", "package:datasets", "package:methods",
    "package:base"
)
```

Чтобы найти все функции, присутствующие в этих пакетах, нам необходимо пройти по `base_pkgs` и применить функции `ls()` и `mget()` на каждой итерации.

```
base_objs <- base_pkgs %>%
    lapply(as.environment) %>%
    lapply(function(x) mget(ls(x, all.names = TRUE), x)) %>%
    setNames(base_pkgs)
```

В результате мы получим более 2700 объектов, обычно доступных по умолчанию.

```
sum(lengths(base_objs))
#> [1] 2709

# Можно также вывести размер в Мб для каждого пакета
vapply(base_objs, obj_size, double(1)) / 1024^2
#>     package:stats  package:graphics package:grDevices     package:utils
#>             11.00              3.08              1.97              7.09
#>  package:datasets   package:methods      package:base
#>              0.54             13.23             18.86

# Проверим на преувеличение подсчета
as.numeric(obj_size(!!!base_objs)) / 1024^2
#> [1] 54
```

3 В языке R (на большинстве платформ) вектор нулевой длины обладает накладными расходами в размере 48 байт.

```
obj_size(list())
#> 48 B
obj_size(double())
#> 48 B
obj_size(character())
#> 48 B
```

Число с типом `double` занимает дополнительные 8 байт памяти.

```
obj_size(double(1))
#> 56 B
obj_size(double(2))
#> 64 B
```

Таким образом, 1 млн чисел с типом `double` должны занимать 8 000 048 байт.

```
a <- runif(1e6)
obj_size(a)
#> 8,000,048 B
```

(Если внимательно посмотреть на объем памяти, занимаемый короткими векторами, можно заметить, что шаблон на самом деле будет более сложным. Это связано с тем, как R выделяет память, но это не столь важно. Если вы хотите знать все подробности, мы обсуждали их в первом издании книги: http://adv-r.had.co.nz/memory.html#object-size).

В случае с `b <- list(a, a)` оба элемента списка содержат ссылку на один и тот же адрес в памяти.

```
b <- list(a, a)
ref(a, b)
#> [1:0x7fb3b4d00000] <dbl>
#>
#> ■ [2:0x7fb3c97bd708] <list>
#> ├─[1:0x7fb3b4d00000]
#> └─[1:0x7fb3b4d00000]
```

Таким образом, для второго элемента списка не потребуется выделять дополнительную память. Сам по себе список требует 64 байта памяти, 48 байт для пустого списка и по 8 байт для каждого элемента (`obj_size(vector("list", 2))`). На этом основании можно сделать соответствующий прогноз: 8 000 048 байт + 64 байта = 8 000 112 байт.

```
obj_size(b)
#> 8,000,112 B
```

При модификации первого элемента списка `b[[1]]` запускается процесс копирования при изменении. Оба элемента будут по-прежнему иметь тот

же объем (8 000 040 Б), но первый из них получит новый адрес в памяти. Поскольку элементы b больше не делят ссылки, их размеры будут суммированы и добавлены к объему списка длины 2: 8 000 048 байт + 8 000 048 байт + 64 байта = 16 000 160 байт (16 Мб).

```
b[[1]][[1]] <- 10
obj_size(b)
#> 16,000,160 B
```

Второй элемент b по-прежнему ссылается на тот же адрес в памяти, что и a, так что объединенный размер a и b будет таким же, как размер b.

```
obj_size(a, b)
#> 16,000,160 B
ref(a, b)
#> [1:0x7fb3b4d00000] <dbl>
#>
#> ▉ [2:0x7fb3d082c508] <list>
#> ├─[3:0x7fb3b3de7000] <dbl>
#> └─[1:0x7fb3b4d00000]
```

При изменении второго элемента b этот элемент будет указывать на новый адрес в памяти. Это не повлияет на размер списка.

```
b[[2]][[1]] <- 10
obj_size(b)
#> 16,000,160 B
```

Но поскольку b больше не делит ссылку с a, объем памяти комбинированных объектов возрастет.

```
ref(a, b)
#> [1:0x7fb3b4d00000] <dbl>
#>
#> ▉ [2:0x7fb3cddcb2c8] <list>
#> ├─[3:0x7fb3b3de7000] <dbl>
#> └─[4:0x7fb3b2400000] <dbl>
obj_size(a, b)
#> 24,000,208 B
```

2.5.3. Ответы на упражнения

1 В данной ситуации механизм копирования при изменении предотвращает образование циклического списка. Давайте посмотрим детально:

```
x <- list()  # создание исходного объекта
obj_addr(x)
#> [1] "0x55862f23ab80"
```

```
tracemem(x)
#> [1] "<0x55862f23ab80>"
x[[1]] <- x  # Механизм копирования при изменении инициирует создание копии
#> tracemem[0x55862f23ab80 -> 0x55862e8ce028]:

obj_addr(x)       # скопированный объект располагается в памяти по новому адресу
#> [1] "0x55862e8ce028"
obj_addr(x[[1]])  # элемент списка содержит старый адрес
#> [1] "0x55862f23ab80"
```

2 Для начала напишем некую функцию для создания случайных данных.

```
create_random_df <- function(nrow, ncol) {
    random_matrix <- matrix(runif(nrow * ncol), nrow = nrow)
    as.data.frame(random_matrix)
}

create_random_df(2, 2)
#>      V1     V2
#> 1 0.972 0.0116
#> 2 0.849 0.4339
```

Теперь обернем в разные функции два метода вычитания числового значения (в нашем случае медианы) в обоих столбцах. Имена функциям выберем в соответствии с тем, работает функция с датафреймом или со списком. Для честного сравнения вторая функция также содержит код с накладными расходами на преобразование между списком и датафреймом.

```
subtract_df <- function(x, medians) {
    for (i in seq_along(medians)) {
      x[[i]] <- x[[i]] - medians[[i]]
    }
    x
}

subtract_list <- function(x, medians) {
    x <- as.list(x)
    x <- subtract_df(x, medians)
    list2DF(x)
}
```

Это позволит нам выполнить профилирование на датафреймах с разным количеством столбцов. Для этого создадим вспомогательную функцию, которая будет создавать датафрейм и медианы перед проверкой производительности при помощи пакета `bench`.

```
benchmark_medians <- function(ncol) {
    df <- create_random_df(nrow = 1e4, ncol = ncol)
    medians <- vapply(df, median, numeric(1), USE.NAMES = FALSE)
```

```
    bench::mark(
      "data frame" = subtract_df(df, medians),
      "list" = subtract_list(df, medians),
      time_unit = "ms"
    )
}

benchmark_medians(1)
#> # A tibble: 2 x 6
#>   expression    min median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr>  <dbl>  <dbl>     <dbl> <bch:byt>    <dbl>
#> 1 data frame 0.0419 0.0740    12450.     344KB     11.2
#> 2 list       0.0543 0.125      7866.     156KB     16.4
```

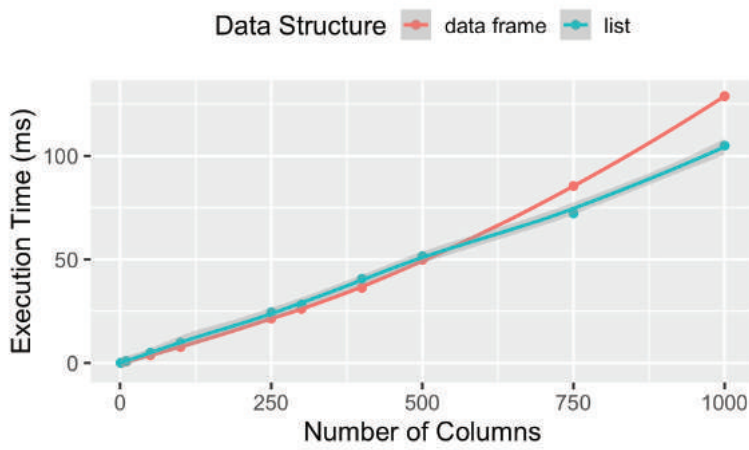
Функция `bench::press()` позволяет запускать нашу вспомогательную функцию по сетке параметров. Мы воспользуемся этим для плавного увеличения количества колонок в датафрейме в нашем прогоне.

```
results <- bench::press(
    ncol = c(1, 10, 50, 100, 250, 300, 400, 500, 750, 1000),
    benchmark_medians(ncol)
)
#> Running with:
#>     ncol
#>  1     1
#>  2    10
#>  3    50
#>  4   100
#>  5   250
#>  6   300
#>  7   400
#>  8   500
#>  9   750
#> 10  1000
```

Наконец, мы можем вывести на графике и проанализировать полученные результаты.

```
library(ggplot2)

ggplot(
    results,
    aes(ncol, median, col = attr(expression, "description"))
) +
    geom_point(size = 2) +
    geom_smooth() +
    labs(
        x = "Number of Columns",
        y = "Execution Time (ms)",
```

```
        colour = "Data Structure"
    ) +
    theme(legend.position = "top")
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



При работе с датафреймом напрямую время выполнения увеличивается квадратично с ростом количества колонок. Причина в том, что первая колонка должна быть скопирована `n` раз, вторая – `n-1` и т. д. При работе со списком время выполнения растет практически линейно.

Очевидно, что на дистанции линейный рост обеспечивает нам меньшее время выполнения, но здесь есть один нюанс, состоящий в необходимости выполнять преобразования между структурами данных с помощью функций `as.list()` и `list2DF()`. Несмотря на их высокую эффективность, реальной отдачи мы дождемся только при достижении 300 колонок в датафрейме (точное количество зависит от технических характеристик системы).

3 Функция `tracemem()` не может быть использована для отслеживания и трассировки окружений.

```
x <- new.env()
tracemem(x)
#> Error in tracemem(x): 'tracemem' is not useful for promise and environment objects
```

Причина появления ошибки состоит в том, что нет никакой пользы в трассировке `NULL`, окружений, промисов, слабых ссылок или внешних указателей на объекты, поскольку они не дублируются (см. `?tracemem`). Окружения всегда модифицируются на месте.

Ответы на упражнения из главы 3

3.2.5. Ответы на упражнения

1 В R скаляры представлены векторами единичной длины. При этом не существует отдельного синтаксиса, как для логических, целочисленных, символьных и векторов двойной точности, для создания отдельных комплексных чисел и байтовых последовательностей. Вместо этого для их создания необходимо использовать функции.

Чтобы создать вектор байтов на основе числового или строкового значения, нужно воспользоваться функциями `as.raw()` и `charToRaw()` соответственно.

```
as.raw(42)
#> [1] 2a
charToRaw("A")
#> [1] 41
```

В случае с комплексными числами вещественная и мнимая части могут быть напрямую переданы на вход конструктору `complex()`.

```
complex(length.out = 1, real = 1, imaginary = 1)
#> [1] 1+1i
```

Вы можете создать и *чисто мнимое число* (purely imaginary number), например `1i`, но невозможно создать комплексное число без использования оператора `+` (например, `1i + 1`).

2

```
c(1, FALSE)      # Приведение к типу double    -> 1 0
c("a", 1)        # Приведение к типу character -> "a" "1"
c(TRUE, 1L)      # Приведение к типу integer   -> 1 1
```

3 Такие сравнения выполняются при помощи операторов-функций (`==`, `<`), которые преобразуют свои аргументы к общему типу данных. В примерах из этого упражнения типы, соответственно, будут следующие: символьный, двойной точности и снова символьный: `1` будет приведен к `"1"`, `FALSE` будет представлен как `0`, а `2` превратится в `"2"` (и числа предшествуют буквам в лексикографическом порядке (может зависеть от локали)).

4 Присутствие пропущенного значения не должно влиять на тип объекта. Вспомните последовательность приведения типов: `character` → `double` → `integer` → `logical`. При комбинировании значений `NA` с другими атомарны-

ми типами NA будет приводиться к целочисленному типу (NA_integer_), типу с двойной точностью (NA_real_) или символьному (NA_character_), но не в обратном порядке. Если пропущенное значение оказалось символьным, то при объединении с другими значениями все они будут приведены также к символьному типу.

5 В документации сказано:

- функция is.atomic() проверяет, является объект атомарным вектором (согласно определению из этой книги) или NULL (!);
- функция is.numeric() проверяет, является объект целочисленным или двойной точности и не принадлежит ли классам factor, Date, POSIXt или difftime;
- функция is.vector() проверяет, является объект вектором (согласно определению из этой книги) или выражением и не имеет ли атрибутов, не считая имен.

Атомарные векторы определены в книге как объекты логического, целочисленного, двойной точности, комплексного, символьного или байтового типа. Векторы определены как атомарные векторы или списки.

3.3.4. Ответы на упражнения

1 Определение функции setNames() выглядит так:

```
setNames <- function(object = nm, nm) {
    names(object) <- nm
    object
}
```

Поскольку аргумент с данными идет первым, функция setNames() прекрасно работает в составе конвейеров из пакета magrittr. При отсутствующем первом аргументе результатом работы функции будет именованный вектор (это весьма нетипично, поскольку обязательные аргументы обычно идут первыми):

```
setNames( , c("a", "b", "c"))
#>   a   b   c
#> "a" "b" "c"
```

Функция unname() реализована следующим образом:

```
unname <- function(obj, force = FALSE) {
    if (!is.null(names(obj)))
      names(obj) <- NULL
    if (!is.null(dimnames(obj)) && (force || !is.data.frame(obj)))
      dimnames(obj) <- NULL
    obj
}
```

Функция `unname()` удаляет существующие имена (или имена измерений) путем установки их в `NULL`.

2 Из справки `?nrow`:

> *Функция* `dim()` *возвращает* `NULL` *при применении к одномерным векторам.*

Кому-то может понадобиться воспользоваться функцией `NROW()` или `NCOL()` для обработки атомарных векторов, списков и значений `NULL`, подобно матрицам или датафреймам с одной колонкой. Для таких объектов функции `nrow()` и `ncol()` будут возвращать `NULL`:

```
x <- 1:10

# Возвращают NULL
nrow(x)
#> NULL
ncol(x)
#> NULL

# Притворяются, будто бы это вектор с одной колонкой
NROW(x)
#> [1] 10
NCOL(x)
#> [1] 1
```

3

```
x1 <- array(1:5, c(1, 1, 5))  # 1 строка,  1 колонка,  5 в третьем измерении.
x2 <- array(1:5, c(1, 5, 1))  # 1 строка,  5 колонок, 1 в третьем измерении.
x3 <- array(1:5, c(5, 1, 1))  # 5 строк, 1 колонка,  1 в третьем измерении.
```

Это все объекты с одним измерением. Если представить себе 3-мерный куб, то `x1` будет представлять измерение `x`, `x2` – измерение `y`, а `x3` – измерение `z`. В отличие от `1:5`, объекты `x1`, `x2` и `x3` обладают атрибутом `dim`.

4 Из справки `?comment`:

> *В отличие от остальных атрибутов, атрибут* `comment` *не выводится (функциями* `print` *или* `print.default`*).*

Также из справки `?attributes`:

> *Обратите внимание, что некоторые атрибуты, такие как* `class`, `comment`, `dim`, `dimnames`, `names`, `row.names` *и* `tsp`, *воспринимаются несколько иначе и имеют строгие ограничения относительно устанавливаемых значений.*

Мы можем извлечь значение атрибута `comment`, обратившись к нему явно:

```
foo <- structure(1:5, comment = "my attribute")

attributes(foo)
#> $comment
#> [1] "my attribute"
attr(foo, which = "comment")
#> [1] "my attribute"
```

3.4.5. Ответы на упражнения

1 Функция `table()` возвращает таблицу сопряженности на основании входных переменных. Она реализована в виде целочисленного вектора класса `table` и с измерениями (которые позволяют ей вести себя как таблица). Атрибутами являются `dim` (измерения) и `dimnames` (по одному имени для каждой колонки). Измерения соответствуют количеству уникальных значений (уровни факторов) в каждой из переменных.

```
x <- table(mtcars[c("vs", "cyl", "am")])

typeof(x)
#> [1] "integer"
attributes(x)
#> $dim
#> [1] 2 3 2
#>
#> $dimnames
#> $dimnames$vs
#> [1] "0" "1"
#>
#> $dimnames$cyl
#> [1] "4" "6" "8"
#>
#> $dimnames$am
#> [1] "0" "1"
#>
#>
#> $class
#> [1] "table"

# Подмножества из x, как если бы это был массив
x[ , , 1]
#>    cyl
#> vs   4  6  8
#>   0  0  0 12
#>   1  3  4  0
x[ , , 2]
#>    cyl
#> vs  4 6 8
```

```
#>   0 1 3 2
#>   1 7 0 0
```

2 Целочисленные значения, лежащие в основе объекта, останутся прежними, а уровни изменятся, что может быть похоже на изменившиеся данные.

```
f1 <- factor(letters)
f1
#>  [1] a b c d e f g h i j k l m n o p q r s t u v w x y z
#> Levels: a b c d e f g h i j k l m n o p q r s t u v w x y z
as.integer(f1)
#>  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
#> [26] 26

levels(f1) <- rev(levels(f1))
f1
#>  [1] z y x w v u t s r q p o n m l k j i h g f e d c b a
#> Levels: z y x w v u t s r q p o n m l k j i h g f e d c b a
as.integer(f1)
#>  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
#> [26] 26
```

3 В переменных `f2` и `f3` поменяется на обратный либо порядок элементов фактора, либо уровни. В переменной `f1` выполнились оба преобразования.

```
# Обратный порядок элементов
(f2 <- rev(factor(letters)))
#>  [1] z y x w v u t s r q p o n m l k j i h g f e d c b a
#> Levels: a b c d e f g h i j k l m n o p q r s t u v w x y z
as.integer(f2)
#>  [1] 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2
#> [26]  1

# Обратный порядок факторов (при создании фактора)
(f3 <- factor(letters, levels = rev(letters)))
#>  [1] a b c d e f g h i j k l m n o p q r s t u v w x y z
#> Levels: z y x w v u t s r q p o n m l k j i h g f e d c b a
as.integer(f3)
#>  [1] 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2
#> [26]  1
```

3.5.4. Ответы на упражнения

1 Список отличий:

- атомарные векторы всегда однородны (все элементы имеют одинаковый тип). Списки могут быть разнородными (элементы могут обладать разными типами), как было сказано во введении к главе, посвященной векторам;

- атомарные векторы указывают на один адрес в памяти, тогда как списки хранят отдельные ссылки для разных элементов. Мы говорили об этом в разделе, посвященном спискам;

```
lobstr::ref(1:2)
#> [1:0x7fcd936f6e80] <int>
lobstr::ref(list(1:2, 2))
#> ▉ [1:0x7fcd93d53048] <list>
#> ├─[2:0x7fcd91377e40] <int>
#> └─[3:0x7fcd93b41eb0] <dbl>
```

- извлечение подмножеств с выходящими за границы значениями и NA приводит к разным результатам. К примеру, использование оператора [вернет NA для атомарных векторов и NULL для списков. Более детально это описывалось в разделе, посвященном извлечению подмножеств;

```
# Извлечение подмножеств из атомарных векторов
(1:2)[3]
#> [1] NA
(1:2)[NA]
#> [1] NA NA

# Извлечение подмножеств из списков
as.list(1:2)[3]
#> [[1]]
#> NULL
as.list(1:2)[NA]
#> [[1]]
#> NULL
#>
#> [[2]]
#> NULL
```

2 Список сам по себе является вектором, хоть и не атомарным.

Обратите внимание, что в функциях `as.vector()` и `is.vector()` используются разные определения вектора.

```
is.vector(as.vector(mtcars))
#> [1] FALSE
```

3 Объекты даты и даты со временем построены на базе чисел с двойной точностью. Если даты хранят количество дней, прошедших с 1 января 1970 года, то даты со временем (`POSIXct`) – количество секунд.

```
date    <- as.Date("1970-01-02")
dttm_ct <- as.POSIXct("1970-01-01 01:00", tz = "UTC")

# Внутреннее представление
unclass(date)
```

```
#> [1] 1
unclass(dttm_ct)
#> [1] 3600
#> attr(,"tzone")
#> [1] "UTC"
```

Поскольку обобщенная функция `c()` диспетчеризуется только по первому аргументу, объединение объектов с датой и датой со временем с помощью этой функции могло приводить к неочевидным результатам в версиях R до 4.0.0:

```
# Вывод в R версии 3.6.2
c(date, dttm_ct)  # эквивалентно c.Date(date, dttm_ct)
#> [1] "1970-01-02" "1979-11-10"
c(dttm_ct, date)  # эквивалентно c.POSIXct(date, dttm_ct)
#> [1] "1970-01-01 02:00:00 CET" "1970-01-01 01:00:01 CET"
```

В первом выражении в приведенном выше коде вычисление производится с помощью метода `c.Date()`, что приводит к некорректному восприятию значения, лежащего в основе `dttm_ct` (3600), в виде количества дней, а не секунд. И наоборот, когда метод `c.POSIXct()` применяется для дат, один день преобразуется в одну секунду.

Этот механизм хорошо прослеживается в следующем примере:

```
# Вывод в R версии 3.6.2
unclass(c(date, dttm_ct))  # внутреннее представление
#> [1] 1 3600
date + 3599
#> "1979-11-10"
```

Начиная с версии R 4.0.0 эти проблемы были решены, и теперь оба метода преобразуют входные данные в `POSIXct` и `Date` соответственно.

```
c(dttm_ct, date)
#> [1] "1970-01-01 01:00:00 UTC" "1970-01-02 00:00:00 UTC"
unclass(c(dttm_ct, date))
#> [1]  3600 86400

c(date, dttm_ct)
#> [1] "1970-01-02" "1970-01-01"
unclass(c(date, dttm_ct))
#> [1] 1 0
```

Однако поскольку функция `c()` обрезает часовой пояс (и другие атрибуты) объектов `POSIXct`, с ней по-прежнему нужно вести себя осторожно.

```
(dttm_ct <- as.POSIXct("1970-01-01 01:00", tz = "HST"))
#> [1] "1970-01-01 01:00:00 HST"
attributes(c(dttm_ct))
```

```
#> $class
#> [1] "POSIXct" "POSIXt"
```

Есть пакет vctrs, который подходит к этим проблемам более глубоко и предлагает собственное структурное решение.

Давайте взглянем, как функция unlist() обрабатывает список.

```
# Атрибуты были отрезаны
unlist(list(date, dttm_ct))
#> [1]     1 39600
```

Мы снова видим, что внутренне даты и даты со временем хранятся в виде чисел двойной точности. К сожалению, это все, что нам осталось, поскольку функция unlist() отсекла атрибуты списка.

Подводя итог, можно сказать, что функция c() выполняет приведение типов данных и обрезает часовые пояса. В старых версиях R использование этой функции может приводить к проблемам из-за несовершенности используемых механизмов диспетчеризации методов. Функция unlist() отрезает атрибуты.

3.6.8. Ответы на упражнения

1 Да, вы спокойно можете построить такие датафреймы – либо прямо на этапе создания, либо при извлечении подмножеств. Даже оба измерения могут быть нулевыми.

Создадим датафреймы с нулем колонок и/или нулем строк (пустой датафрейм) напрямую:

```
data.frame(a = integer(), b = logical())
#> [1] a b
#> <0 строк> (или row.names нулевой длины)

data.frame(row.names = 1:3)  # или data.frame()[1:3, ]
#> датафрейм с 0 колонок и 3 строками

data.frame()
#> датафрейм с 0 колонок и 0 строк
```

Создать подобные датафреймы можно и посредством извлечения подмножеств. При этом вы можете использовать 0, NULL, FALSE или любой атомарный вектор нулевой длины (logical(0), character(0), integer(0), double(0)). Последовательности с отрицательными целочисленными значениями также подойдут. В следующем примере мы воспользуемся нулем:

```
mtcars[0, ]
#>  [1] mpg  cyl  disp hp   drat wt   qsec vs   am   gear carb
#> <0 строк> (или row.names нулевой длины)
```

```
mtcars[ , 0]  # or mtcars[0]
#> датафрейм с 0 колонок и 32 строками

mtcars[0, 0]
#> датафрейм с 0 колонок и 0 строк
```

2 В матрицах могут присутствовать дублирующиеся имена строк, так что никаких проблем с ними возникнуть не должно.

Что касается датафреймов, они требуют наличия уникальных имен строк, и вы получите разные результаты в зависимости от того, как будете устанавливать их. Если попытаться установить имена напрямую или посредством функции `row.names()`, мы получим ошибку:

```
data.frame(row.names = c("x", "y", "y"))
#> Error in data.frame(row.names = c("x", "y", "y")): duplicate row.names: y

df <- data.frame(x = 1:3)
row.names(df) <- c("x", "y", "y")
#> Warning: non-unique value when setting 'row.names': 'y'
#> Error in `.rowNamesDF<-`(x, value = value): duplicate 'row.names' are not allowed
```

При использовании техники извлечения подмножеств оператор [выполнит исключение дубликатов:

```
row.names(df) <- c("x", "y", "z")
df[c(1, 1, 1), , drop = FALSE]
#>     x
#> x   1
#> x.1 1
#> x.2 1
```

3 Оба вызова (`t(df)` и `t(t(df))`) вернут матрицу:

```
df <- data.frame(x = 1:3, y = letters[1:3])
is.matrix(df)
#> [1] FALSE
is.matrix(t(df))
#> [1] TRUE
is.matrix(t(t(df)))
#> [1] TRUE
```

Измерения будут следовать обычным правилам транспонирования:

```
dim(df)
#> [1] 3 2
dim(t(df))
#> [1] 2 3
dim(t(t(df)))
#> [1] 3 2
```

Поскольку на выходе мы получили матрицу, каждая колонка будет приведена к одному типу данных. Это реализовано в `t.data.frame()` посредством функции `as.matrix()`, как описано ниже.

```
df
#>   x y
#> 1 1 a
#> 2 2 b
#> 3 3 c
t(df)
#>   [,1] [,2] [,3]
#> x "1"  "2"  "3"
#> y "a"  "b"  "c"
```

4 Тип результата функции `as.matrix()` зависит от типов данных исходных колонок (см. `?as.matrix`):

> *Для датафреймов метод будет возвращать символьную матрицу, если в нем содержатся только атомарные колонки и любой не(числовой/логический/комплексный) столбец, при этом к факторам будет применена функция* `as.vector()`*, а к другим несимвольным колонкам – функция* `format()`*. В противном случае будет использована стандартная иерархия приведения типов (*`logical` < `integer` < `double` < `complex`*) – например, полностью логические датафреймы будут преобразованы в логические матрицы, смешанные логически-целочисленные датафреймы будут давать целочисленные матрицы и т. д.*

С другой стороны, функция `data.matrix()` всегда будет возвращать числовую матрицу (см. `?data.matrix()`):

> *Возвращает матрицу, полученную путем преобразования всех переменных датафрейма в числовой вид и объединения в виде столбцов матрицы. Факторы и упорядоченные факторы будут заменены в соответствии с внутренними кодами. […] Символьные колонки сначала преобразуются в факторы, а затем – в целочисленные значения.*

Мы можем проиллюстрировать и сравнить механизмы действия этих функций на конкретном примере. Функция `as.matrix()` позволяет извлечь большую часть исходной информации из датафрейма, но оставляет нас с символьными представлениями. Для получения всей нужной информации из вывода функции `data.matrix()` нам потребуется таблица поиска для каждого столбца.

```
df_coltypes <- data.frame(
    a = c("a", "b"),
    b = c(TRUE, FALSE),
    c = c(1L, 0L),
    d = c(1.5, 2),
    e = factor(c("f1", "f2"))
```

```
)

as.matrix(df_coltypes)
#>      a   b       c   d     e
#> [1,] "a" "TRUE"  "1" "1.5" "f1"
#> [2,] "b" "FALSE" "0" "2.0" "f2"
data.matrix(df_coltypes)
#>      a b c   d e
#> [1,] 1 1 1 1.5 1
#> [2,] 2 0 0 2.0 2
```

Ответы на упражнения из главы 4

4.2.6. Ответы на упражнения

1

```
mtcars[mtcars$cyl = 4, ]
# используйте `==`              (вместо `=`)

mtcars[-1:4, ]
# используйте `-(1:4)`          (вместо `-1:4`)

mtcars[mtcars$cyl <= 5]
# `,` пропущена

mtcars[mtcars$cyl == 4 | 6, ]
# используйте `mtcars$cyl == 6` (вместо `6`)
#  или `%in% c(4, 6)`           (вместо `== 4 | 6`)
```

2 В отличие от `NA_real`, `NA` имеет логический тип, и логические векторы переписываются, приобретая ту же длину, что и у вектора, из которого извлекается подмножество. Таким образом, `x[NA]` переписывается в `x[NA, NA, NA, NA, NA]`.

3 Вызов `upper.tri(x)` возвращает логическую матрицу, содержащую значения `TRUE` в элементах над главной диагональю и значения `FALSE` в остальных элементах. В случае с `upper.tri()` позиции для значений `TRUE` и `FALSE` определяются путем сравнения индексов строки и столбца у `x` с использованием следующего неравенства: `.row(dim(x)) < .col(dim(x))`.

```
x
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,]    1    2    3    4    5
#> [2,]    2    4    6    8   10
#> [3,]    3    6    9   12   15
#> [4,]    4    8   12   16   20
```

```
#> [5,]    5   10   15   20   25
upper.tri(x)
#>       [,1]  [,2]  [,3]  [,4]  [,5]
#> [1,] FALSE  TRUE  TRUE  TRUE  TRUE
#> [2,] FALSE FALSE  TRUE  TRUE  TRUE
#> [3,] FALSE FALSE FALSE  TRUE  TRUE
#> [4,] FALSE FALSE FALSE FALSE  TRUE
#> [5,] FALSE FALSE FALSE FALSE FALSE
```

При извлечении подмножеств применительно к логическим матрицам будут выбраны все элементы, содержащие значение `TRUE`. Матрицы расширяют векторы с помощью атрибута измерения, так что можно использовать векторную форму при извлечении подмножеств (включая логические подмножества). Нужно внимательно проверять, чтобы матрица, используемая для создания подмножества, соответствовала целевому объекту, – в противном случае может быть сделан неожиданный выбор элементов по причине применения правил переписывания векторов. Обратите внимание, что такая форма извлечения подмножества возвращает вектор, а не матрицу, поскольку эта операция оказывает влияние на измерения объекта.

```
x[upper.tri(x)]
#>  [1]  2  3  6  4  8 12  5 10 15 20
```

4 При извлечении подмножества из датафрейма при помощи простого вектора поведение будет таким же, как при извлечении подмножества из списка колонок. Таким образом, выражение `mtcars[1:20]` должно было бы вернуть датафрейм, содержащий первые 20 колонок из набора данных. Однако набор `mtcars` насчитывает всего 11 колонок, что и явилось источником ошибки выхода за допустимые границы. В свою очередь, выражение `mtcars[1:20, ]` осуществляет извлечение подмножеств с помощью двух векторов, и в этом случае первый индекс будет относиться не к колонкам, а к строкам.

5 Элементы, располагающиеся на главной диагонали матрицы, обладают одинаковыми индексами для строки и колонки. Это свойство можно использовать с целью создания подходящей числовой матрицы, которую можно будет применить для извлечения подмножеств.

```
diag2 <- function(x) {
    n <- min(nrow(x), ncol(x))
    idx <- cbind(seq_len(n), seq_len(n))

    x[idx]
}

# Проверим
(x <- matrix(1:30, 5))
#>      [,1] [,2] [,3] [,4] [,5] [,6]
#> [1,]    1    6   11   16   21   26
```

```
#> [2,]    2    7   12   17   22   27
#> [3,]    3    8   13   18   23   28
#> [4,]    4    9   14   19   24   29
#> [5,]    5   10   15   20   25   30

diag(x)
#> [1]  1  7 13 19 25
diag2(x)
#> [1]  1  7 13 19 25
```

6 Это выражение заменит значения `NA` в `df` на 0. В данном случае `is.na(df)` возвращает логическую матрицу, кодирующую позиции пропущенных значений в `df`. В результате объединения операций извлечения подмножества и присваивания в датафрейме будут обнулены только пропущенные значения.

4.3.5. Ответы на упражнения

1 В базовом R для этой задачи присутствует масса возможностей:

```
# Сначала выбирается колонка
mtcars$cyl[[3]]
#> [1] 4
mtcars[ , "cyl"][[3]]
#> [1] 4
mtcars[["cyl"]][[3]]
#> [1] 4
with(mtcars, cyl[[3]])
#> [1] 4

# Сначала выбирается строка
mtcars[3, ]$cyl
#> [1] 4
mtcars[3, "cyl"]
#> [1] 4
mtcars[3, ][ , "cyl"]
#> [1] 4
mtcars[3, ][["cyl"]]
#> [1] 4

# Одновременный выбор
mtcars[3, 2]
#> [1] 4
mtcars[[c(2, 3)]]
#> [1] 4
```

2 Переменная `mod` представлена типом список, что открывает несколько вариантов для решения поставленной задачи. Воспользуемся операторами `$` и `[[` для извлечения одного элемента:

```
mod <- lm(mpg ~ wt, data = mtcars)

mod$df.residual
#> [1] 30
mod[["df.residual"]]
#> [1] 30
```

То же самое применимо и к `summary(mod)`, так что мы можем использовать следующее выражение:

```
summary(mod)$r.squared
#> [1] 0.753
```

Совет: в пакете `broom` представлены очень полезные инструменты для работы с моделями.

4.5.9. Ответы на упражнения

1 Это можно реализовать с помощью комбинации оператора `[` и функции `sample()`:

```
# Перемешиваем колонки
mtcars[sample(ncol(mtcars))]

# Перемешиваем колонки и строки за один шаг
mtcars[sample(nrow(mtcars)), sample(ncol(mtcars))]
```

2 Выбрать `m` случайных строк из датафрейма можно с помощью операции извлечения подмножеств.

```
m <- 10
mtcars[sample(nrow(mtcars), m), ]
```

Чтобы сохранить непрерывность выборки, нужно внимательно отнестись к расчету начального и конечного индексов.

```
start <- sample(nrow(mtcars) - m + 1, 1)
end <- start + m - 1
mtcars[start:end, , drop = FALSE]
```

3 Для решения этой задачи можно совместно использовать оператор `[` с функцией `order()` или `sort()`:

```
mtcars[order(names(mtcars))]
mtcars[sort(names(mtcars))]
```

Ответы на упражнения из главы 5

5.2.4. Ответы на упражнения

1 Аргументы функции `ifelse()` именуются так: `test`, `yes` и `no`. В общем случае функция `ifelse()` возвращает значение аргумента `yes`, если значение аргумента `test` равно `TRUE`, и значение аргумента `no`, если значение аргумента `test` равно `FALSE`. Значение `NA` возвращается в случае, если значение аргумента `test` равно `NA`. Таким образом, эти выражения будут возвращать векторы типа `double` (`1`), `character` (`"no"`) и `logical` (`NA`).

Для большей точности процитируем документацию к функции `ifelse()` в части возвращаемого значения:

> *Вектор той же длины и с теми же атрибутами (включая измерения и* `class`*), что и объект* `test`*, и со значениями, соответствующими аргументам* `yes` *или* `no`*. Тип возвращаемого значения будет приведен из логического таким образом, чтобы сначала удовлетворять любым элементам из аргумента* `yes`*, а затем – любым элементам из аргумента* `no`*.*

Это удивляет, поскольку здесь используется тип аргумента `test`. На практике это означает, что сначала аргумент `test` приводится к логическому типу, а если результат не будет соответствовать `TRUE` или `FALSE`, вернется результат выражения `as.logical(test)`.

```
ifelse(logical(), 1, "no")
#> logical(0)
ifelse(NaN, 1, "no")
#> [1] NA
ifelse(NA_character_, 1, "no")
#> [1] NA
ifelse("a", 1, "no")
#> [1] NA
ifelse("true", 1, "no")
#> [1] 1
```

2 Функция `if()` ожидает на вход логическое выражение, но может принимать и числовой вектор, в котором нулевые значения будут интерпретироваться как `FALSE`, а все остальные – как `TRUE`. Наличие в векторе пропущенных значений (включая `NaN`) приведет к ошибке, как и передача в функцию пропущенного логического значения (`NA`).

5.3.3. Ответы на упражнения

1 Этот цикл представляет собой довольно деликатный случай, и для ответа на поставленный вопрос нам необходимо прояснить несколько моментов.

Для начала рассмотрим выражение `1:length(x)`, определяющее индексы цикла `for`. Поскольку `x` обладает нулевой длиной, выражение `1:length(x)` приведет к отсчету значений от 1 до 0. Обычно во избежание подобных проблем прибегают к помощи функции `seq_along(x)` или подобным, которые в данном случае выдали бы `integer(0)`.

Поскольку дальше мы используем операторы `[<-` и `[` для индексирования векторов нулевой длины с индексами 1 и 0, необходимо вспомнить правила извлечения подмножеств для нулевого индекса и индекса, выходящего за границы.

На первой итерации `x[1]` даст в результате `NA` (правило индексирования за пределами границ для атомарных векторов). После возведения в квадрат остается значение `NA`, которое будет присвоено элементу с индексом 1 пустого списка единичной длины (`out[1]`) (правило индексирования за пределами границ для списков).

На следующей итерации цикла выражение `x[0]` вернет `numeric(0)` (правило индексирования нулем для атомарных векторов). Возведение в квадрат снова не изменит значение, и `numeric(0)` будет присвоен `out[0]` (правило индексирования нулем для списков). Присваивание вектора нулевой длины подмножеству нулевой длины работает, но не приводит к изменению исходного объекта.

В результате мы видим, что этот код работать будет, поскольку он включает в себя допустимые для R операции. Другое дело, что результат, скорее всего, будет для пользователя весьма неожиданным.

2 В представленном цикле переменная `x` принимает значения исходной переменной `xs` (1, 2 и 3), поскольку она вычисляется один раз в начале цикла, а не на каждой итерации. В противном случае мы бы вошли в бесконечный цикл.

3 В цикле `for` индекс обновляется в начале каждой итерации. Вследствие этого присваивание этому символу другого значения при выполнении итерации никак не влияет на положение вещей в начале следующей итерации. И снова в этом случае мы бы попали в бесконечный цикл.

Ответы на упражнения из главы 6

6.2.5. Ответы на упражнения

1 В R нет привязки «один к одному» между функциями и именами. Имя всегда указывает на один объект, тогда как сам объект может иметь одно имя, несколько, а может и не иметь вовсе.

Давайте рассмотрим это на примере:

```
function(x) sd(x) / mean(x)
#> function(x) sd(x) / mean(x)
```

```
f1 <- function(x) (x - min(x)) / (max(x) - min(x))
f2 <- f1
f3 <- f1
```

Если первая функция не привязана ни к одному имени, то на вторую указывают сразу три имени (`f1`, `f2` и `f3`). Таким образом, главная мысль здесь состоит в том, что связь между именами и объектами в R определяется только в одном направлении.

Несмотря на это, в R, разумеется, есть способы получения имен функции. Однако для полной уверенности в том, что вы найдете правильные имена, необходимо сравнивать не только код (тело), но также аргументы (формальные) и окружение. Поскольку функции `formals()`, `body()` и `environment()` возвращают `NULL` для примитивных функций, самым простым способом проверки двух функций на равенство будет использование функции `identical()`.

2 Корректным является второй подход.

Анонимная функция `function(x) 3` обрамляется круглыми скобками перед ее вызовом с помощью оператора `()`. Эти дополнительные скобки служат для отделения вызова функции от тела анонимной функции. Без них вернется функция с неправильным телом `3()`, которая при вызове выдаст ошибку. Это легко увидеть, если дать функции имя:

```
f <- function(x) 3()
f
#> function(x) 3()
f()
#> Error in f(): attempt to apply non-function
```

3 Использование анонимных функций позволяет в определенных ситуациях добиться лаконичности и элегантности кода. В то же время в отсутствие осмысленного имени по прошествии времени бывает непросто вспомнить, что именно делает функция. Именно по этой причине хорошим тоном считается давать длинным и сложным функциям описательные имена. Вы можете просмотреть свой код или код коллег на наличие анонимных функций, которым неплохо было бы дать осмысленные и понятные имена.

4 Проверить, является ли объект функцией, можно с помощью функции `is.function()`. Для определения того, является ли функция примитивной, можно воспользоваться функцией `is.primitive()`.

5 Давайте рассмотрим каждый вопрос в отдельности.

a Для нахождения функции с наибольшим количеством аргументов для начала посчитаем длину `formals()`.

```
library(purrr)

n_args <- funs %>%
```

```
    map(formals) %>%
    map_int(length)
```

Затем отсортируем `n_args` в порядке убывания и выберем первые элементы.

```
n_args %>%
    sort(decreasing = TRUE) %>%
    head()
#> scan format.default source
#> 22 16 16
#> formatC library merge.data.frame
#> 15 13 13
```

b Далее мы можем использовать тот же объект `n_args` для подсчета количества функций, в которых аргументы отсутствуют:

```
sum(n_args == 0)
#> [1] 248
```

Но эта цифра сильно преувеличена, поскольку функция `formals()` возвращает `NULL` для примитивных функций, а `length(NULL)` равно 0. Чтобы откорректировать сумму, можно сначала избавиться от примитивных функций:

```
n_args2 <- funs %>%
    discard(is.primitive) %>%
    map(formals) %>%
    map_int(length)

sum(n_args2 == 0)
#> [1] 47
```

И действительно, большинство функций без аргументов – это примитивные функции.

c Для поиска всех примитивных функций можно поменять предикат в функции `Filter()` с `is.function()` на `is.primitive()`:

```
funs <- Filter(is.primitive, objs)
length(funs)
#> [1] 201
```

6 Эти три компонента – это `body()`, `formals()` и `environment()`. Однако как мы уже упоминали в книге:

> *Есть одно исключение из правила о том, что все функции насчитывают три компонента. Примитивные функции, такие как* `sum()`*, напрямую вызывают код на C с помощью функции* `.Primitive()` *и не содержат кода на языке R. Таким образом, все их компоненты –* `formals()`*,* `body()` *и* `environment()` *– содержат значение* `NULL`*.*

7 Окружение не выводится для примитивных функций и функций, созданных в глобальном окружении.

6.4.5. Ответы на упражнения

1 Этот код вернет именованный числовой вектор единичной длины – с одним элементом со значением 10 и именем `"c"`. Первая `c` обозначает функцию `c()`, вторая `c` интерпретируется как (цитированное) имя, а третья `c` – как значение.

2 Правила лексического поиска базируются на четырех основных принципах:

- маскировка имен;
- функции против переменных;
- с чистого листа;
- динамический поиск.

3 Внутри этой иерархической функции `f` объявлены и вызываются еще две функции с таким же именем. Поскольку каждая функция вызывается в своем окружении, R в процессе поиска будет использовать последнюю функцию, определенную в этих окружениях. Сама вложенная функция `f()` будет вызвана последней, хотя она первая вернет свое значение. Таким образом, порядок вычислений будет идти изнутри наружу, и функция вернет ((10 ^ 2) + 1) * 2 = 202.

6.5.4. Ответы на упражнения

1 Если говорить коротко, то оператор `&&` выполняется по сокращенной схеме, а это означает, что если выражение, стоящее слева от оператора, вернет `FALSE`, то правое выражение не вычисляется за ненадобностью. Точно так же если выражение слева от оператора `||` возвращает `TRUE`, правая часть не вычисляется.

Мы ожидаем, что наша функция `x_ok()` будет проверять входящий аргумент по определенным критериям: он не должен являться `NULL`, должен иметь единичную длину и быть больше нуля. Итогом проверки могут быть значения `TRUE`, `FALSE` или `NA`. Желаемое поведение достигается путем комбинации утверждений, пропущенных через оператор `&&`, а не `&`.

Оператор `&&` не выполняет поэлементные сравнения. Вместо этого он использует только первый элемент каждого значения. Также в нем применяется отложенное вычисление в том смысле, что оно *«производится ровно до того момента, как будет определен результат»* (из справки `?Logic`). Это означает, что выражение, стоящее справа от оператора `&&`, не будет вычисляться вовсе, если в левой части мы уже получили ожидаемый результат (т. е. `FALSE`). Такое поведение называется выполнением по сокращенной схеме. Для некоторых значений (`x = 1`) оба оператора дадут одинаковый результат. Но так будет не всегда. Для `x = NULL` оператор `&&` остановится после вычисления выражения `!is.null` и вернет результат. Остальные условия даже не будут проверяться!

(Если бы другие выражения также вычислялись (с использованием оператора &), результат бы изменился. NULL > 0 возвращает logical(0), что в нашем случае не помогает.)

Мы также видим различия в поведении функций при x = 1:3. Оператор && возвращает значение на основании условия length(x) == 1, дающего FALSE. Использование & в качестве логического оператора приводит к векторизации условия x > 0 и возвращению соответствующего результата.

2 Функция вернет значение 100. Аргумент по умолчанию (x = z) вычисляется отложенно в окружении функции при доступе к x. В этот момент имя z уже будет связано со значением 100. На этом примере демонстрируется принцип отложенного, или ленивого, вычисления.

3 Функция вернет вектор c(2, 1) по причине маскировки имен. При доступе к x внутри c() промис x = {y <- 1; 2} вычисляется в окружении функции f1(). Имя y получает привязку к единице, а возвращаемое значение {() (2) присваивается x. При следующем обращении к y внутри c() у этого имени уже есть привязанное значение, и R нет необходимости выполнять новый поиск. Таким образом, промис y = 0 не будет вычислен. Также, поскольку переменной y было присвоено значение внутри окружения функции f1(), значение глобальной переменной с тем же именем останется нетронутым.

4 С помощью аргумента xlim функции hist() задается диапазон оси *x* на гистограмме. Для правильного отображения оси аргумент xlim должен содержать числовой вектор, состоящий исключительно из двух уникальных значений. Таким образом, для значения по умолчанию (xlim = range(breaks)) выражение breaks должно давать при вычислении минимум два уникальных значения.

Во время выполнения функция hist() перезаписывает breaks. При этом выражение breaks обладает достаточной гибкостью и позволяет пользователю задать значения явно или вычислить их любым доступным способом. Таким образом, конкретное поведение функции в большой степени зависит от входа. В то же время функция hist проверяет, что выражение аргумента breaks при вычислении дает числовой вектор, содержащий как минимум два уникальных элемента, перед тем как вычислить аргумент xlim.

5 Перед получением доступа к аргументу x (по умолчанию stop("Error")) функция show_time() маскирует функцию stop() при помощи выражения function(...) Sys.time(). Поскольку аргументы по умолчанию вычисляются в окружении функции, выражение print(x) будет вычислено как print(Sys.time()).

Данная функция может сбивать с толку из-за того, что ее поведение меняется, когда аргумент x передается явно. В этом случае будет использовано значение из вызывающего окружения, и переопределение функции stop() не повлияет на x.

```
show_time(x = stop("Error!"))
#> Error in print(x): Error!
```

Функция `library()` не имеет обязательных аргументов. При вызове без аргументов она невидимо возвращает список класса `libraryIQR`, содержащий матрицу результатов с одной строкой и тремя колонками на каждый установленный пакет. В этих колонках содержатся название пакета (`"Package"`), путь к нему (`"LibPath"`) и заголовок пакета (`"Title"`). Функция `library()` также располагает своим собственным методом `print` (`print.libraryIQR()`), отображающим информацию в удобном для просмотра виде в отдельном окне.

Такое поведение документировано в разделе с подробностями работы функции `library()` на странице справки (`?library`):

> *При вызове функции без указания пакета или аргумента* `help` *она формирует полный список всех доступных пакетов в библиотеках, указанных в аргументе* `lib.loc`*, и возвращает информацию в виде объекта класса* `libraryIQR`*. (Структура этого класса может измениться в будущем.) Воспользуйтесь функцией* `.packages(all = TRUE)` *для получения только имен доступных пакетов или функцией* `installed.packages()` *для извлечения дополнительной информации.*

Из-за того что в аргументах `package` и `help` функции `library()` не показываются значения по умолчанию, легко подумать, что без них эту функцию использовать нельзя. (Вместо передачи `NULL` в качестве значений по умолчанию функция `library()` использует функцию `missing()` для проверки фактических значений аргументов.)

```
str(formals(library))
#> Dotted pair list of 13
#>  $ package        : symbol
#>  $ help           : symbol
#>  $ pos            : num 2
#>  $ lib.loc        : NULL
#>  $ character.only : logi FALSE
#>  $ logical.return : logi FALSE
#>  $ warn.conflicts : symbol
#>  $ quietly        : logi FALSE
#>  $ verbose        : language getOption("verbose")
#>  $ mask.ok        : symbol
#>  $ exclude        : symbol
#>  $ include.only   : symbol
#>  $ attach.required: language missing(include.only)
```

6.6.1. Ответы на упражнения

1 Давайте рассмотрим аргументы и их порядок в обеих функциях. Для функции `sum()` это `...` и `na.rm`:

```
str(sum)
#> function (..., na.rm = FALSE)
```

В качестве аргумента `...` функция `sum()` ожидает получить числовой, комплексный или логический вектор (см. справку `?sum`). К сожалению, при ис-

пользовании … неправильно записанные аргументы, такие как `na.omit`, воспринимаются как логические и входят в состав аргумента …. В результате это значение интерпретируется как единица и входит в состав суммы. Все остальные аргументы остаются неизменными. Именно поэтому выражение `sum(1, 2, 3)` возвращает 6, а `sum(1, 2, 3, na.omit = TRUE)` – 7.

Напротив, обобщенная функция `mean()` ожидает для своего метода по умолчанию аргументы `x`, `trim`, `na.rm` и ….

```
str(mean.default)
#> function (x, trim = 0, na.rm = FALSE, ...)
```

Поскольку `na.omit` не входит в число именованных аргументов `mean()` и не является кандидатом на частичное соответствие, он также входит в состав аргумента …. Однако, в отличие от функции `sum()`, элементы … не являются частью расчета среднего. Остальные переданные аргументы сопоставляются в соответствии с порядком, т. е. `x = 1`, `trim = 2`, а `na.rm = 3`. Поскольку аргумент `x` представляет собой вектор длины 1, а не `NA`, параметры `trim` и `na.rm` не влияют на вычисление среднего. В результате оба вызова (`mean(1, 2, 3)` и `mean(1, 2, 3, na.omit = TRUE)`) возвращают единицу.

2 Для начала вводим в консоли команду `?plot` и проверяем раздел **Usage**, в котором есть следующая запись:

```
plot(x, y, ...)
```

Аргументы, о которых нам хочется узнать побольше (`col`, `pch`, `xlab`, `col.lab`), входят в …. Здесь мы можем обнаружить информацию об аргументе `xlab`, а для остальных аргументов рекомендуется ввести команду `?par`. На панели поиска вводим `"col"` и попадаем в раздел цветовой спецификации. Так же точно мы можем найти и аргумент `pch`, в разделе по которому содержится рекомендация ознакомиться еще со справкой `?points`. Аргумент `col.lab` содержится и в документации `?par`.

3 Чтобы изучить внутренности метода `plot.default()`, необходимо добавить вызов функции `browser()` в первую строку кода и интерактивно запустить инструкцию `plot(1:10, col = "red")`. В результате мы сможем увидеть, как строится график, и узнаем, где добавляются оси.

Это приводит нас к следующему вызову функции:

```
localTitle(main = main, sub = sub, xlab = xlab, ylab = ylab, ...)
```

Функция `localTitle()` объявляется в первых строках метода `plot.default()`:

```
localTitle <- function(..., col, bg, pch, cex, lty, lwd) title(...)
```

В вызове функции `localTitle()` параметр `col` передается в составе аргумента … в функцию `title()`. Справка `?title` говорит о том, что функция `title()`

определяет четыре составляющие графика: `main` (заголовок графика), `sub` (подзаголовок графика) и метки обеих осей. Таким образом, при использовании `col` в функции `title()` в явном виде возникла бы неопределенность. Вместо этого у нас есть возможность передать параметр `col` посредством аргумента …, через аргумент `col.lab` или как часть аргумента `xlab` в виде `xlab = list(c("index"), col = "red")` (то же касается и `ylab`).

6.7.5. Ответы на упражнения

1 Функция `load()` загружает объекты, сохраненные в файлах `.Rdata` функцией `save()`. При успешном запуске функция `load()` невидимо возвращает символьный вектор, содержащий имена загруженных объектов. Для вывода этих имен на консоль можно установить аргументу `verbose` значение `TRUE` или заключить вызов функции в скобки для запуска механизма автоматического вывода.

2 Функция `write.table()` записывает объект (обычно датафрейм или матрицу) на диск. Функция невидимо возвращает `NULL`. Было бы удобнее, если бы эта функция невидимо возвращала входные данные, `x`. Это позволило бы сохранять промежуточные результаты и напрямую запускать следующие шаги по их обработке, не прерывая поток кода (т. е. не разбивая его на отдельные строки). Одним из пакетов, в которых используется этот шаблон, является пакет `readr`, входящий в экосистему tidyverse.

3 Код функции `with_dir()` был представлен в книге следующим образом:

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

    force(code)
}
```

Функция `with_dir()` принимает путь к рабочей директории (`dir`) в качестве первого аргумента. Это та директория, в которой должен быть выполнен переданный код (`code`). В начале функции меняется текущая рабочая директория при помощи функции `setwd()`. Затем, на выходе из функции (`on.exit()`), значение текущей директории восстанавливается в исходное значение. Передавая путь явно, пользователь получает полный доступ к директории, в которой выполняется код.

В функцию `source()` код передается посредством аргумента `file` (путь к файлу). С помощью аргумента `chdir` указывается, будет ли рабочая директория меняться на директорию, содержащую файл. По умолчанию значение аргумента `chdir` установлено в `FALSE`, так что вам нет нужды передавать значение явно. В то же время ограничение на передачу только значений `TRUE` или `FALSE` лишает вас достаточной гибкости в отношении выбора рабочей директории для выполнения кода.

4 Для контроля за графическим устройством можно воспользоваться функциями `pdf()` и `dev.off()`. Безопасное завершение работы гарантируется использованием функции `on.exit()`.

```
plot_pdf <- function(code) {
    pdf("test.pdf")
    on.exit(dev.off(), add = TRUE)
    code
}
```

5 С помощью инструкции `body(capture.output)` мы можем получить доступ к исходному коду функции `capture.output()`: ее реализация оказалась гораздо более объемной – 39 строк против семи.

В функции `capture_output2()` переданный аргумент отправляется на вычисление принудительно, а вывод захватывается во временном файле с помощью функции `sink()`. Дополнительной особенностью функции `capture_output()` является возможность захватывать сообщения путем установки значения аргумента `type = "message"`. Поскольку внутренне этот функционал реализован при помощи функции `sink()`, такое поведение (а также аргумент `split` функции `sink()`) может быть легко воплощено и в функции `capture_output2()`.

Главным отличием между этими функциями является осуществление функцией `capture.output()` вывода, что обусловливает такую разницу при их вызове:

```
capture.output({1})
#> [1] "[1] 1"
capture.output2({1})
#> character(0)
```

6.8.6. Ответы на упражнения

1 Давайте напишем выражения, которые семантически будут полностью соответствовать приведенному выше коду. Поскольку префиксные функции сами определяют порядок вычисления, мы можем опустить скобки во втором выражении.

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

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

`if`(`<=`(length(x), 5), `[[`(x, 5), `[[`(x, n))
```

2 Ни в одной из представленных функций нет аргумента …. Таким образом, сначала аргументы будут сопоставляться буквально, затем частично и в заключение – по позиции. Это приводит нас к показанным ниже явным вызовам функций:

```
x <- sample(c(1:10, NA), size = 20, replace = TRUE)
y <- runif(20, min = 0, max = 1)
cor(x, y, use = "pairwise.complete.obs", method = "kendall")
```

3 Давайте для начала определим переменную `x` и вспомним объявление функции `modify()`, приведенное в книге:

```
x <- 1:3

`modify<-` <- function(x, position, value) {
    x[position] <- value
    x
}
```

Внутри R происходит преобразование кода, и преобразованный код воспроизводит показанную выше ошибку:

```
get("x") <- `modify<-`(get("x"), 1, 10)
#> Error in get("x") <- `modify<-`(get("x"), 1, 10) :
#>   target of assignment expands to non-language object
```

Ошибка возникает в момент присваивания из-за отсутствия соответствующей замещающей функции, т. е. `get<-`. Для подтверждения этого мы воспроизведем ошибку в следующем упрощенном примере.

```
get("x") <- 2
#> Error in get("x") <- 2 :
#>   target of assignment expands to non-language object
```

4 Определим замещающую функцию `random<-` следующим образом:

```
`random<-` <- function(x, value) {
    idx <- sample(length(x), 1)
    x[idx] <- value
    x
}
```

5 Чтобы добиться такого поведения, нам необходимо переопределить оператор `+`. При этом нам нужно внимательно следить за тем, чтобы не использовать сам оператор `+` внутри определения функции, поскольку это приведет к нежелательной бесконечной рекурсии. Мы также добавим значение по умолчанию для аргумента `b = 0L`, чтобы сохранить поведение унарного оператора `+`, т. е. чтобы выражение `+1` работало и не выбрасывало ошибку.

```
`+` <- function(a, b = 0L) {
    if (is.character(a) && is.character(b)) {
      paste0(a, b)
    } else {
```

```
      base::`+`(a, b)
    }
}

# Проверка
+ 1
#> [1] 1
1 + 2
#> [1] 3
"a" + "b"
#> [1] "ab"

# Возвращаем исходный оператор `+`
rm(`+`)
```

6 Подсказка дает нам понять, что нам нужно осуществить поиск функций по определенному шаблону: замещающие функции традиционно заканчиваются последовательностью символов `<-`. Для этого можно передать регулярное выражение `"<-$"` в функцию `apropos()`. Функция `apropos()` также позволяет вернуть позицию в пути поиска (`search()`) для каждого вхождения путем установки значения аргумента `where = TRUE`. Наконец, мы можем передать аргумент `mode = function`, чтобы ограничить поиск только нужными нам объектами. Получим следующий вызов, с которого можно начать:

```
repls <- apropos("<-", where = TRUE, mode = "function")
head(repls, 30)
#>                      10                      10                      10
#>        ".rowNamesDF<-"                 "[[<-"      "[[<-.data.frame"
#>                      10                      10                      10
#>         "[[<-.factor" "[[<-.numeric_version"         "[[<-.POSIXlt"
#>                      10                      10                      10
#>                 "[<-"       "[<-.data.frame"             "[<-.Date"
#>                      10                      10                      10
#>          "[<-.factor"  "[<-.numeric_version"          "[<-.POSIXct"
#>                      10                      10                      10
#>         "[<-.POSIXlt"                  "@<-"                    "<-"
#>                      10                      10                      10
#>                 "<<-"                  "$<-"       "$<-.data.frame"
#>                       8                      10                      10
#>                "as<-"               "attr<-"         "attributes<-"
#>                       8                      10                      10
#>              "body<-"               "body<-"              "class<-"
#>                       8                      10                      10
#>           "coerce<-"           "colnames<-"            "comment<-"
#>                       3                      10                      10
#>         "contrasts<-"               "diag<-"                "dim<-"
```

Для получения только замещающих функций из базового пакета мы ограничим выбор нужной нам позицией в пути поиска.

```
repls_base <- repls[names(repls) == length(search())]
repls_base
#>                         10                         10                         10
#>           ".rowNamesDF<-"                    "[[<-"         "[[<-.data.frame"
#>                         10                         10                         10
#>             "[[<-.factor"    "[[<-.numeric_version"            "[[<-.POSIXlt"
#>                         10                         10                         10
#>                      "[<-"          "[<-.data.frame"                "[<-.Date"
#>                         10                         10                         10
#>              "[<-.factor"     "[<-.numeric_version"             "[<-.POSIXct"
#>                         10                         10                         10
#>             "[<-.POSIXlt"                     "@<-"                      "<-"
#>                         10                         10                         10
#>                      "<<-"                     "$<-"          "$<-.data.frame"
#>                         10                         10                         10
#>                  "attr<-"            "attributes<-"                "body<-"
#>                         10                         10                         10
#>                "class<-"              "colnames<-"               "comment<-"
#>                         10                         10                         10
#>                 "diag<-"                   "dim<-"              "dimnames<-"
#>                         10                         10                         10
#>   "dimnames<-.data.frame"              "Encoding<-"           "environment<-"
#>                         10                         10                         10
#>               "formals<-"                 "is.na<-"         "is.na<-.default"
#>                         10                         10                         10
#>          "is.na<-.factor" "is.na<-.numeric_version"               "length<-"
#>                         10                         10                         10
#>           "length<-.Date"       "length<-.difftime"         "length<-.factor"
#>                         10                         10                         10
#>        "length<-.POSIXct"        "length<-.POSIXlt"               "levels<-"
#>                         10                         10                         10
#>         "levels<-.factor"                  "mode<-"        "mostattributes<-"
#>                         10                         10                         10
#>                "names<-"          "names<-.POSIXlt"             "oldClass<-"
#>                         10                         10                         10
#>            "parent.env<-"            "regmatches<-"            "row.names<-"
#>                         10                         10                         10
#>  "row.names<-.data.frame"     "row.names<-.default"             "rownames<-"
#>                         10                         10                         10
#>                "split<-"      "split<-.data.frame"         "split<-.default"
#>                         10                         10                         10
#>          "storage.mode<-"                "substr<-"            "substring<-"
#>                         10                         10
#>                "units<-"         "units<-.difftime"
```

Для определения того, какие из найденных функций являются примитивными, мы сначала попытаемся найти их с помощью функции `mget()`, после чего отфильтруем результаты, применив функции `Filter()` и `is.primitive()`.

```
repls_base_prim <- mget(repls_base, envir = baseenv()) %>%
    Filter(is.primitive, .) %>%
```

```
    names()

repls_base_prim
#>  [1] "[[<-"           "[<-"            "@<-"            "<-"
#>  [5] "<<-"            "$<-"            "attr<-"         "attributes<-"
#>  [9] "class<-"        "dim<-"          "dimnames<-"     "environment<-"
#> [13] "length<-"       "levels<-"       "names<-"        "oldClass<-"
#> [17] "storage.mode<-"
```

Всего в базовом пакете содержится 62 замещающие функции, 17 из которых – примитивные.

7 Процитируем раздел из этой книги, посвященный функциональным формам:

> *Имена инфиксных функций могут быть более гибкими по сравнению с обычными функциями в R: они могут содержать любую последовательность символов, за исключением %.*

8 Инфиксный оператор `%xor%` можно создать так:

```
`%xor%` <- function(a, b) {
      xor(a, b)
}
TRUE %xor% TRUE
#> [1] FALSE
FALSE %xor% TRUE
#> [1] TRUE
```

9 Эти инфиксные операторы могут быть определены следующим образом (последовательность `%/%` была выбрана вместо `%\%`, поскольку символ \ используется в качестве экранирующего):

```
`%n%` <- function(a, b) {
    intersect(a, b)
}

`%u%` <- function(a, b) {
    union(a, b)
}

`%/%` <- function(a, b) {
    setdiff(a, b)
}

x <- c("a", "b", "d")
y <- c("a", "c", "d")

x %u% y
#> [1] "a" "b" "d" "c"
```

```
x %n% y
#> [1] "a" "d"
x %/% y
#> [1] "b"
```

Ответы на упражнения из главы 7

7.2.7. Ответы на упражнения

1 Наиболее важные отличия между окружениями и списками следующие:

- окружения обладают ссылочной семантикой (т. е. не поддерживают парадигму копирования при изменении);
- у окружений присутствует родительский элемент;
- в содержимом окружения должны присутствовать уникальные имена;
- содержимое окружений не упорядочено;
- (окружения можно сравнивать только с помощью функции `identical()`, а не с помощью оператора ==);
- окружения могут содержать сами себя.

2 Давайте создадим окружение, содержащее само себя.

```
e1 <- env()
e1$loop <- e1

# Выводим окружение
env_print(e1)
#> <environment: 0x7f87749ff828>
#> parent: <environment: global>
#> bindings:
#>  * loop: <env>

# Проверяем, что оно содержит само себя
lobstr::ref(e1)
#> ▉ [1:0x7f87749ff828] <env>
#> └─loop = [1:0x7f87749ff828]
```

3 Показанные окружения содержат друг друга:

```
e1 <- env()
e2 <- env()

e1$loop   <- e2
e2$dedoop <- e1

lobstr::ref(e1)
```

```
#> ▊ [1:0x7f87745ed8c0] <env>
#> └loop = ▊ [2:0x7f87746aa698] <env>
#>          └dedoop = [1:0x7f87745ed8c0]
lobstr::ref(e2)
#> ▊ [1:0x7f87746aa698] <env>
#> └dedoop = ▊ [2:0x7f87745ed8c0] <env>
#>            └loop = [1:0x7f87746aa698]
```

4 Первое выражение не имеет смысла, поскольку элементы окружения не упорядочены. Второе выражение вернуло бы два объекта одновременно. Внутри какой структуры данных они будут содержаться?

5 Как мы уже говорили, функция `rlang::env_poke()` принимает имя (в виде строки) и значение для осуществления присваивания (или переприсваивания) в окружении.

```
e3 <- new.env()

env_poke(e3, "a", 100)
e3$a
#> [1] 100
env_poke(e3, "a", 200)
e3$a
#> [1] 200
```

Таким образом, в функции `env_poke2()` нам необходимо добавить проверку на существование имени в заданном окружении. Это можно сделать с помощью функции `env_has()`. Если имя уже присутствует, будем выводить содержательную ошибку.

```
env_poke2 <- function(env, name, value) {
    if (env_has(env, name)) {
        abort(paste0("\"", name, "\" уже присутствует в окружении."))
    }

    env_poke(env, name, value)
    invisible(env)
}

# Проверка
env_poke2(e3, "b", 100)
e3$b
#> [1] 100
env_poke2(e3, "b", 200)
#> Error: "b" is already assigned to a value.
```

6 Главным отличием функции `rebind()` от оператора `<<-` является то, что она будет выполнять присваивание только в том случае, если требуемая привязка существует. В отличие от оператора `<<-`, она не будет создавать новые

привязки в глобальном окружении. Такое поведение оператора <<- обычно нежелательно, поскольку глобальные переменные вводят неочевидные зависимости между функциями.

7.3.1. Ответы на упражнения

1 Функция `where()` рекурсивно ищет указанное имя в заданном окружении и его предках. При нахождении имени в одном из этих окружений возвращается имя окружения. В противном случае возбуждается ошибка.

Определение функции `where()` было представлено в книге следующим образом:

```
where <- function(name, env = caller_env()) {
    if (identical(env, empty_env())) {
        # Базовый случай
        stop("Can't find `", name, "`.", call. = FALSE)
    } else if (env_has(env, name)) {
        # Успех
        env
    } else {
        # Рекурсивный случай
        where(name, env_parent(env))
    }
}
```

Наша модифицированная функция будет вызываться рекурсивно до тех пор, пока не будет достигнуто пустое окружение. При этом не имеет значения, было на этот момент найдено запрашиваемое имя или нет. По пути функция будет проверять наличие имени в окружениях. Наконец, функция будет возвращать список окружений, в которых были обнаружены привязки к искомому имени. Если ни в одном из них имя не будет найдено, вернется пустой список.

Обратите внимание также на то, как именно инициализируется список с помощью значения аргумента по умолчанию при первом вызове функции. Это может сбивать с толку, и по этой причине принято оборачивать рекурсивные функции в другие, более дружественные и понятные функции.

```
where2 <- function(name, env = caller_env(), results = list()) {
    if (identical(env, empty_env())) {
        # Базовый случай
        results
    } else {
        # Рекурсивный случай
        if (env_has(env, name)) {
            results <- c(results, env)
        }
        where2(name, env_parent(env), results)
    }
}
```

```
# Проверка
e1a <- env(empty_env(), a = 1, b = 2)
e1b <- env(e1a, b = 10, c = 11)
e1c <- env(e1b, a = 12, d = 13)

where2("a", e1c)
#> [[1]]
#> <environment: 0x7f8771e9b000>
#>
#> [[2]]
#> <environment: 0x7f87728d8ee8>
```

2 Используем тот же подход, что и в предыдущем упражнении. На этот раз мы будем дополнительно проверять, является ли найденный объект функцией, а также введем еще один аргумент, с помощью которого можно при желании отключить рекурсивный поиск.

```
fget <- function(name, env = caller_env(), inherits = TRUE) {
    # Базовый случай
    if (env_has(env, name)) {
        obj <- env_get(env, name)

        if (is.function(obj)) {
            return(obj)
        }
    }

    if (identical(env, emptyenv()) || !inherits) {
        stop("Could not find a function called \"", name, "\".",
          call. = FALSE
        )
    }

    # Рекурсивный случай
    fget(name, env_parent(env))
}

# Проверка
mean <- 10
fget("mean", inherits = TRUE)
#> function (x, ...)
#> UseMethod("mean")
#> <bytecode: 0x7f876f282208>
#> <environment: namespace:base>
```

7.4.5. Ответы на упражнения

1 Функция `search_envs()` возвращает все окружения в пути поиска, который представляет собой цепочку окружений, содержащую экспортированные функции присоединенных пакетов (из справки `?search_envs`). Каждый раз, когда вы присоединяете новый пакет, путь поиска расширяется. Завершается этот путь базовым окружением. Глобальное окружение входит в путь поиска по причине того, что функции, представленные в нем, всегда будут частью этого пути.

```
search_envs()
#>  [[1]] $ <env: global>
#>  [[2]] $ <env: package:rlang>
#>  [[3]] $ <env: package:stats>
#>  [[4]] $ <env: package:graphics>
#>  [[5]] $ <env: package:grDevices>
#>  [[6]] $ <env: package:utils>
#>  [[7]] $ <env: package:datasets>
#>  [[8]] $ <env: package:methods>
#>  [[9]] $ <env: Autoloads>
#> [[10]] $ <env: package:base>
```

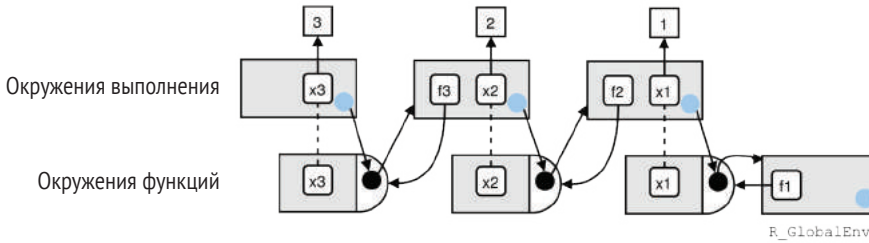
Вызов функции `env_parents(global_env())` вернет список всех предков глобального окружения, но само глобальное окружение в список не войдет. В этот список будет включен и финальный предок в виде пустого окружения. Это окружение не входит в путь поиска, поскольку не содержит объектов.

```
env_parents(global_env())
#>  [[1]] $ <env: package:rlang>
#>  [[2]] $ <env: package:stats>
#>  [[3]] $ <env: package:graphics>
#>  [[4]] $ <env: package:grDevices>
#>  [[5]] $ <env: package:utils>
#>  [[6]] $ <env: package:datasets>
#>  [[7]] $ <env: package:methods>
#>  [[8]] $ <env: Autoloads>
#>  [[9]] $ <env: package:base>
#> [[10]] $ <env: empty>
```

2 Это упражнение заставляет нас задуматься об окружении функции в момент ее создания.

При создании функция `f1` привязывается к своему родительскому окружению, которым является глобальное окружение. Но функция `f2` будет создана только в момент выполнения функции `f1`, а значит, будет привязываться к окружению выполнения функции `f1`. Значение 1 также будет привязываться к имени `x1` во время выполнения. То же самое касается `x2`, `f3` и `x3`.

На рисунке схематично показаны отношения между окружениями функций.



Мы также можем отследить привязку окружений, добавив несколько выводов в функцию. Обратите внимание, что функции `print` будут вызываться во время выполнения. Таким образом, выполнение инструкции `f1(1)` будет давать разные результаты при каждом запуске.

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

f1(1)
#> [1] "f3"
#> <environment: 0x7f877079f758>
#> parent: <environment: 0x7f87707a34a8>
#> bindings:
#>  * x3: <dbl>
#> <environment: 0x7f877079f758>
#> [1] "f2"
#> <environment: 0x7f87707a34a8>
#> parent: <environment: 0x7f87707a32e8>
#> bindings:
#>  * f3: <fn>
#>  * x2: <dbl>
#> <environment: 0x7f87707a34a8>
#> [1] "f1"
#> <environment: 0x7f87707a32e8>
#> parent: <environment: global>
```

```
#> bindings:
#>  * f2: <fn>
#>  * x1: <dbl>
#> <environment: 0x7f87707a32e8>
```

3 Для решения этой задачи нам необходимо написать функцию, которая будет принимать имя функции и осуществлять ее поиск. Возвращать она будет как саму функцию, так и окружение, в котором она была найдена.

```
fget2 <- function(name, env = caller_env()) {
    # Базовый случай
    if (env_has(env, name)) {
        obj <- env_get(env, name)

        if (is.function(obj)) {
            return(list(fun = obj, env = env))
        }
    }

    if (identical(env, emptyenv())) {
        stop("Could not find a function called \"", name, "\"",
            call. = FALSE
        )
    }

    # Рекурсивный случай
    fget2(name, env_parent(env))
}

fstr <- function(fun_name, env = caller_env()) {
    if (!is.character(fun_name) && length(fun_name) == 1) {
        stop("`fun_name` must be a string.", call. = FALSE)
    }
    fun_env <- fget2(fun_name, env)

    list(
        where = fun_env$env,
        enclosing = fn_env(fun_env$fun)
    )
}

# Проверка
fstr("mean")
#> $where
#> <environment: base>
#>
#> $enclosing
#> <environment: namespace:base>
```

После изучения концепции *tidy evaluation* вы могли бы переписать функцию `fstr()` таким образом, чтобы она использовала функцию `enquo()`. Это позволило бы вызывать ее подобно функции `str()`, например `fstr(sum)`.

7.5.5. Ответы на упражнения

1 Мы можем реализовать динамический поиск путем явной ссылки на вызывающее окружение. Обратите внимание, что при использовании этого подхода также возвращаются переменные, начинающиеся с точки, – эта опция обычно требуется для функции `ls()`.

```
ls2 <- function(env = caller_env()) {
    sort(env_names(env))
}

# Проверка в глобальном окружении
ls(all.names = TRUE)
#>  [1] ".Random.seed" "%>%"          "a"            "e1"           "e1a"
#>  [6] "e1b"          "e1c"          "e2"           "e3"           "env_poke2"
#> [11] "error_wrap"   "f1"           "fget"         "fget2"        "fstr"
#> [16] "ls2"          "mean"         "rebind"       "where"        "where2"
ls2()
#>  [1] ".Random.seed" "%>%"          "a"            "e1"           "e1a"
#>  [6] "e1b"          "e1c"          "e2"           "e3"           "env_poke2"
#> [11] "error_wrap"   "f1"           "fget"         "fget2"        "fstr"
#> [16] "ls2"          "mean"         "rebind"       "where"        "where2"

# Проверка в окружении-песочнице
e1 <- env(a = 1, b = 2)
ls(e1)
#> [1] "a" "b"
ls2(e1)
#> [1] "a" "b"
```

Ответы на упражнения из главы 8

8.2.4. Ответы на упражнения

1 Мы предпочтем следующее решение задачи, простое и понятное.

```
file_remove_strict <- function(path) {
    if (!file.exists(path)) {
        stop("Can't delete the file \"", path,
               "\" because it doesn't exist.",
               call. = FALSE
        )
    }
    file.remove(path)
}

# Проверка
saveRDS(mtcars, "mtcars.rds")
```

```
file_remove_strict("mtcars.rds")
#> [1] TRUE
file_remove_strict("mtcars.rds")
#> Error: Can't delete the file "mtcars.rds" because it doesn't exist.
```

2 Аргумент `appendLF` автоматически добавляет перенос строки к сообщению. Давайте проиллюстрируем это на простом примере:

```
multiline_msg <- function(appendLF = TRUE) {
    message("first", appendLF = appendLF)
    message("second", appendLF = appendLF)
    cat("third")
    cat("fourth")
}

multiline_msg(appendLF = TRUE)
#> first
#> second
#> thirdfourth
multiline_msg(appendLF = FALSE)
#> firstsecondthirdfourth
```

Похожее поведение в отношении переносов строк можно реализовать в функции `cat()`, если установить аргументу `sep` значение `"\n"`.

8.4.5. Ответы на упражнения

1 В отличие от функции `stop()`, которая содержит вызов, функция `abort()` сохраняет всю трассировку, сгенерированную функцией `rlang::trace_back()`. А это очень много дополнительных данных!

```
str(catch_cnd(stop("An error")))
#> List of 2
#>  $ message: chr "An error"
#>  $ call   : language force(expr)
#>  - attr(*, "class")= chr [1:3] "simpleError" "error" "condition"

str(catch_cnd(abort("An error")))
#> List of 3
#>  $ message: chr "An error"
#>  $ trace  :List of 3
#>   ..$ calls  :List of 8
#>   .. ..$ : language utils::str(catch_cnd(abort("An error")))
#>   .. ..$ : language rlang::catch_cnd(abort("An error"))
#>   .. ..$ : language rlang::eval_bare(rlang::expr(tryCatch(!!!handlers, {     f..
#>   .. ..$ : language base::tryCatch(condition = function (x)  x, { ...
#>   .. ..$ : language base:::tryCatchList(expr, classes, parentenv, handlers)
#>   .. ..$ : language base:::tryCatchOne(expr, names, parentenv, handlers[[1L]])
#>   .. ..$ : language base:::doTryCatch(return(expr), name, parentenv, handler)
#>   .. ..$ : language base::force(expr)
```

```
#>   ..$ parents: int [1:8] 0 0 2 2 4 5 6 2
#>   ..$ indices: int [1:8] 27 28 29 30 31 32 33 34
#>   ..- attr(*, "class")= chr "rlang_trace"
#>   ..- attr(*, "version")= int 1
#>  $ parent : NULL
#>  - attr(*, "class")= chr [1:3] "rlang_error" "error" "condition"
```

2 С первыми тремя примерами все просто:

```
show_condition(stop("!"))      # stop возбуждает ошибку
#> [1] "error"
show_condition(10)             # никакого состояния нет
#> NULL
show_condition(warning("?!"))  # warning возбуждает предупреждение
#> [1] "warning"
```

Последний пример интереснее, здесь мы знакомимся с любопытной особенностью функции `tryCatch()`, которая завершает выполнение программы при ее вызове.

```
show_condition({
    10
    message("?")
    warning("?!")
})
#> [1] "message"
```

3

```
withCallingHandlers(  # (1)
    message = function(cnd) message("b"),
    withCallingHandlers(  # (2)
        message = function(cnd) message("a"),
        message("c")
    )
)
#> b
#> a
#> b
#> c
```

Распутывать такие узлы бывает непросто. Сначала осуществляется вызов функции `message("c")`, и он перехватывается меткой `(1)`. Затем происходит вызов `message("a")`, который перехватывается меткой `(2)`, после чего вызывается `message("b")`. Вызов `message("b")` нигде не перехватывается, так что на консоль выводится `b`, после чего выводится `a`. Но почему перед выводом `c` мы видим еще одну `b`? Причина в том, что мы не обработали сообщение, из-за чего оно поднялось во внешний обработчик.

4 Функция `catch_cnd()` является просто оберткой для функции `tryCatch()`. Если возникает состояние, оно перехватывается и возвращается. В отсутствие состояний выполнение продолжается последовательно, и функция возвращает `NULL`.

Текущая версия функции `catch_cnd()` немного более сложная – она позволяет указать, какие классы состояний вы хотите перехватывать. Это требует написания кода вручную, поскольку интерфейс `tryCatch()` представляет классы состояний в виде имен аргументов.

```
rlang::catch_cnd
#> function (expr, classes = "condition")
#> {
#>     stopifnot(is_character(classes))
#>     handlers <- rep_named(classes, list(identity))
#>     eval_bare(rlang::expr(tryCatch(!!!handlers, {
#>         force(expr)
#>         return(NULL)
#>     })))
#> }
#> <bytecode: 0x7fd75d598c88>
#> <environment: namespace:rlang>
```

5 Определение функции `show_condition()` было представлено в одном из предыдущих упражнений. Давайте воспользуемся аргументом `condition` функции `tryCatch()`, как показано выше в `rlang::catch_cond()`, для нашей реализации:

```
show_condition2 <- function(code) {
    tryCatch(
        condition = function(cnd) {
            if (inherits(cnd, "error"))   return("error")
            if (inherits(cnd, "warning")) return("warning")
            if (inherits(cnd, "message")) return("message")
        },
        {
            code
            NULL
        }
    )
}

# Проверка
show_condition2(stop("!"))
#> [1] "error"
show_condition2(10)
#> NULL
show_condition2(warning("?!"))
#> [1] "warning"
show_condition2({
```

```
  10
  message("?")
  warning("?!")
})
#> [1] "message"
```

Функция `tryCatch()` выполняет код и перехватывает все возникающие состояния. Функция, переданная в качестве аргумента `condition`, перехватывает состояние. В этом случае происходит диспетчеризация на класс события.

8.5.4. Ответы на упражнения

1 Воспользуемся функцией `rlang::abort()` для передачи метаданных об ошибке:

```
check_installed <- function(package) {
    if (!requireNamespace(package, quietly = FALSE)) {
        abort(
            "error_pkg_not_found",
            message = paste0("package '", package, "' not installed."),
            package = package
        )
    }

    TRUE
}

check_installed("ggplot2")
#> [1] TRUE
check_installed("ggplot3")
#> Loading required namespace: ggplot3
#> Error: package 'ggplot3' not installed.
```

2 Вместо возвращения ошибки бывает предпочтительно выбрасывать пользовательское состояние и в метаданные помещать стандартное сообщение об ошибке. Затем нижестоящий пакет может проверить класс состояния вместо рассмотрения сообщения.

8.6.6. Ответы на упражнения

1 Обычно мы стремимся перехватывать возникающие ошибки, поскольку в них содержится важная информация для отладки. Для подавления сообщений об ошибках и скрытия возвращаемого объекта ошибки в консоли мы обрабатываем ошибки внутри функции `tryCatch()` и возвращаем объект ошибки невидимо:

```
suppressErrors <- function(expr) {
    tryCatch(
        error = function(cnd) invisible(cnd),
```

```
        interrupt = function(cnd) {
            stop("Terminated by the user.",
              call. = FALSE
            )
        },
        expr
    )
}
```

После определения обработчика ошибок можно объединить его с другими обработчиками при создании функции `suppressConditions()`:

```
suppressConditions <- function(expr) {
    suppressErrors(suppressWarnings(suppressMessages(expr)))
}
```

Для проверки новой функции применим ее к набору состояний и проанализируем возвращенный объект ошибки.

```
# Сообщения/предупреждения/состояния успешно подавляются
error_obj <- suppressConditions({
    message("message")
    warning("warning")
    abort("error")
})

error_obj
#> <error/rlang_error>
#> error
#> Backtrace:
#>   1. global::suppressConditions(...)
#>  12. base::suppressMessages(expr)
#>  13. base::withCallingHandlers(...)
```

2 Эти функции отличаются способом обработки состояний. Функция `withCallingHandlers()` создает обработчик вызова, который запускается внутри функции. Это позволяет детально проанализировать стек вызовов, что очень помогает при идентификации возникающих состояний.

Что касается функции `tryCatch()`, то она создает обработчик выхода, а это означает, что функция будет завершена сразу после возникновения состояния. Одновременно с этим контроль управления возвращается в контекст, в котором была вызвана функция `tryCatch()`.

В данном примере использование функции `withCallingHandlers()` позволяет получить больше информации по сравнению с функцией `tryCatch()`. В результате мы можем точно определить вызов, в котором возникло состояние.

```
message2error1 <- function(code) {
    withCallingHandlers(code, message = function(e) stop("error"))
}
```

```
message2error1({1;  message("hidden error"); NULL})
#> Error in (function (e) : error
traceback()
#> 9: stop("error") at #2
#> 8: (function (e)
#>    stop("error"))(list(message = "hidden error\n",
#>      call = message("hidden error")))
#> 7: signalCondition(cond)
#> 6: doWithOneRestart(return(expr), restart)
#> 5: withOneRestart(expr, restarts[[1L]])
#> 4: withRestarts({
#>        signalCondition(cond)
#>        defaultHandler(cond)
#>    }, muffleMessage = function() NULL)
#> 3: message("hidden error") at #1
#> 2: withCallingHandlers(code,
#>      message = function(e) stop("error")) at #2
#> 1: message2error1({
#>        1
#>        message("hidden error")
#>        NULL
#>    })

message2error2 <- function(code) {
    tryCatch(code, message = function(e) (stop("error")))
}

message2error2({1; stop("hidden error"); NULL})
#> Error in value[[3L]](cond) : error
traceback()
#> 6: stop("error") at #2
#> 5: value[[3L]](cond)
#> 4: tryCatchOne(expr, names, parentenv, handlers[[1L]])
#> 3: tryCatchList(expr, classes, parentenv, handlers)
#> 2: tryCatch(code, message = function(e) (stop("error"))) at #2
#> 1: message2error2({
#>        1
#>        message("hidden error")
#>        NULL
#>    })
```

3 В реализации функции `catch_cnds()`, представленной в книге, уже решена эта задача путем сохранения всех сообщений и предупреждений в их исходном порядке в списке.

```
catch_cnds <- function(expr) {
    conds <- list()
    add_cond <- function(cnd) {
```

```
        conds <<- append(conds, list(cnd))
        cnd_muffle(cnd)
    }

    tryCatch(
        error = function(cnd) {
            conds <<- append(conds, list(cnd))
        },
        withCallingHandlers(
            message = add_cond,
            warning = add_cond,
            expr
        )
    )

    conds
}

# Проверка
catch_cnds({
    inform("message a")
    warn("warning b")
    inform("message c")
})
#> [[1]]
#> <message: message a
#> >
#>
#> [[2]]
#> <warning: warning b>
#>
#> [[3]]
#> <message: message c
#> >
```

4 При запуске функции `bottles_of_beer()` в консоли вывод должен выглядеть как-то так:

```
bottles_of_beer()
#> There are 99 bottles of beer on the wall, 99 bottles of beer.
#> Take one down, pass it around, 98 bottles of beer on the wall.
#> Take one down, pass it around, 97 bottles of beer on the wall.
#> Take one down, pass it around, 96 bottles of beer on the wall.
#> Take one down, pass it around, 95 bottles of beer on the wall.
#>
```

Возможно, на этом этапе вы поймете, как сложно уменьшить количество бутылок с 99 до 0. Выйти из функции невозможно, поскольку мы захватываем прерывание, которое обычно используется!

Ответы на упражнения из главы 9

9.2.6. Ответы на упражнения

1 Функция `map()` позволяет по-разному передавать аргумент, связанный с применяемой функцией (`.f`): в виде формулы, функции или *функции-экстрактора* (extractor function). Таким образом, сначала в функции `map()` выполняется преобразование входного значения в допустимую функцию, чем занимается вспомогательная функция `as_mapper()`, вызываемая каждый раз при использовании функции `map()`.

С учетом типа входного параметра (символьный, числовой или списочный) функция `as_mapper()` создает функцию-экстрактор. Символьные данные выбираются по имени, числовые – по позиции, а списочные позволяют воспользоваться смешанным подходом. Этот интерфейс экстрактора может быть очень полезным при работе с вложенными данными.

Функция-экстрактор реализуется в виде вызова функции `purrr::pluck()`, принимающей список функций доступа (accessor function), которые позволяют получить доступ к объекту данных.

```
as_mapper(c(1, 2))  # эквивалентно function(x) x[[1]][[2]]
#> function (x, ...)
#> pluck(x, 1, 2, .default = NULL)
#> <environment: 0x7fd304648470>
as_mapper(c("a", "b"))  # эквивалентно function(x) x[["a"]][["b]]
#> function (x, ...)
#> pluck(x, "a", "b", .default = NULL)
#> <environment: 0x7fd30471a470>
as_mapper(list(1, "b"))  # эквивалентно function(x) x[[1]][["b]]
#> function (x, ...)
#> pluck(x, 1, "b", .default = NULL)
#> <environment: 0x7fd3047952b0>
```

Помимо смеси из позиций и имен, вы можете также передать саму функцию доступа. Обычно она представляет собой анонимную функцию для получения информации об определенном аспекте входных данных. Вы можете объявлять собственные функции доступа.

Если вам необходимо получить доступ к конкретным атрибутам, вы можете воспользоваться предопределенной вспомогательной функцией `attr_getter(y)`, которая создаст функцию доступа за вас.

```
# Определяем пользовательскую функцию доступа
get_class <- function(x) attr(x, "class")
pluck(mtcars, get_class)
#> [1] "data.frame"

# Используем attr_getter() в качестве вспомогательной функции
```

```
pluck(mtcars, attr_getter("class"))
#> [1] "data.frame"
```

2 Первая запись приводит к генерации нескольких случайных чисел, поскольку в аргументе `~ runif(2)` успешно используется интерфейс формул. При запуске функция `map()` применяет вспомогательную функцию `as_mapper()` к переданной формуле, что приводит к преобразованию выражения `~ runif(2)` в анонимную функцию. Впоследствии функция `runif(2)` вызывается трижды (по разу на каждой итерации), что приводит к генерированию трех пар случайных значений.

Во второй записи функция `runif(2)` выполняется один раз, после чего результаты передаются в функцию `map()`. Вследствие этого функция `as_mapper()` создает функцию-экстрактор на основе возвращаемых значений из `runif(2)` (с помощью `pluck()`). Это приводит к появлению трех значений `NULL` (аргумент `.default` в функции `pluck()`) из-за невозможности найти значения, соответствующие индексу.

```
as_mapper(~ runif(2))
#> <lambda>
#> function (..., .x = ..1, .y = ..2, . = ..1)
#> runif(2)
#> attr(,"class")
#> [1] "rlang_lambda_function" "function"
as_mapper(runif(2))
#> function (x, ...)
#> pluck(x, 0.0807501375675201, 0.834333037259057, .default = NULL)
#> <environment: 0x7fd304ed4550>
```

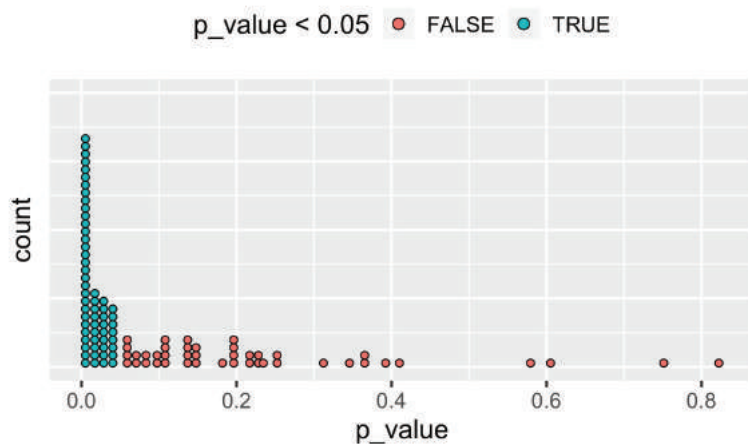
3 Для решения этих задач мы воспользуемся типизированными разновидностями функции `map()`, обеспечивающими более лаконичный вывод, а также используем функцию `map_lgl()` для выбора колонок из датафрейма (позже мы познакомимся с функцией `keep()`, немного упрощающей этот шаблон).

```
map_dbl(mtcars, sd)
#>     mpg     cyl    disp      hp    drat      wt    qsec      vs      am    gear
#>   6.027   1.786 123.939  68.563   0.535   0.978   1.787   0.504   0.499   0.738
#>    carb
#>   1.615

penguins <- palmerpenguins::penguins

penguins_numeric <- map_lgl(penguins, is.numeric)
map_dbl(penguins[penguins_numeric], sd, na.rm = TRUE)
#>    bill_length_mm     bill_depth_mm flipper_length_mm       body_mass_g
#>             5.460             1.975            14.062           801.955
#>              year
#>             0.818
```

```
penguins_factor <- map_lgl(penguins, is.factor)
map_int(penguins[penguins_factor], ~ length(levels(.x)))
#> species  island     sex
#>       3       3       2
```

4 Существует несколько способов для визуализации данных. Но поскольку у нас есть всего 100 наблюдений, мы остановимся на точечной диаграмме для отображения этого распределения. К сожалению, функция `geom_dotplot()` из пакета `ggplot2` не позволяет рассчитывать точные количества, поскольку она была разработана для визуализации плотностей распределений, а не частот. В связи с этим гистограмма могла бы подойти лучше.

```
library(ggplot2)

df_trials <- tibble::tibble(p_value = map_dbl(trials, "p.value"))

df_trials %>%
    ggplot(aes(x = p_value, fill = p_value < 0.05)) +
    geom_dotplot(binwidth = .01) +  # альтернатива - geom_histogram()
    theme(
        axis.text.y = element_blank(),
        axis.ticks.y = element_blank(),
        legend.position = "top"
    )
```



5 Ошибка здесь возникает по причине того, что функция `triple()` передана в качестве аргумента `.f`, а значит, принадлежит внешней функции `map()`. Неименованный аргумент `map` воспринимается как аргумент функции `triple()`, что и приводит к возникновению ошибки.

Есть разные способы решения этой проблемы. Для этого простого примера выбор способа не так важен, но применительно к более сложным задачам вам будет полезно опробовать разные альтернативы.

```
# Не именуйте аргумент
map(x, map, triple)

# Воспользуйтесь анонимной функцией в стиле magrittr
map(x, . %>% map(triple))

# Воспользуйтесь анонимной функцией в стиле purrr
map(x, ~ map(.x, triple))
```

6 Данные (`mtcars`) постоянны для всех этих моделей, так что мы можем пройти итерациями по `formulas`. Поскольку формула передается в функцию `lm()` первым аргументом, нет необходимости указывать имя явно.

```
models <- map(formulas, lm, data = mtcars)
```

7 Для решения этой задачи мы воспользуемся шаблоном «список на входе, список на выходе» функции `map()`. Это позволит нам связать различные преобразования вместе. Начнем с подгонки моделей, после чего рассчитаем итоговые результаты и извлечем значения R^2. В последнем вызове воспользуемся типизированной функцией `map_dbl()`, обеспечивающей лаконичный вывод.

```
bootstraps %>%
    map(~ lm(mpg ~ disp, data = .x)) %>%
    map(summary) %>%
    map_dbl("r.squared")
#>  [1] 0.588 0.822 0.745 0.746 0.784 0.749 0.613 0.792 0.653 0.726
```

9.4.6. Ответы на упражнения

1 Функция `modify()` базируется на функции `map()`, и в данном случае будет использован интерфейс функции-экстрактора, которая будет извлекать первый элемент из каждой колонки в наборе данных `mtcars`. Функция `modify()` всегда возвращает ту же структуру, что и во входном аргументе: здесь это приводит к 32-кратному повторению первой строки (внутренне в функции `modify()` используется следующий механизм присваивания: `.x[] <- map(.x, .f, ...)`).

```
head(modify(mtcars, 1))
#>                   mpg cyl disp  hp drat   wt qsec vs am gear carb
#> Mazda RX4          21   6  160 110  3.9 2.62 16.5  0  1    4    4
#> Mazda RX4 Wag      21   6  160 110  3.9 2.62 16.5  0  1    4    4
#> Datsun 710         21   6  160 110  3.9 2.62 16.5  0  1    4    4
#> Hornet 4 Drive     21   6  160 110  3.9 2.62 16.5  0  1    4    4
#> Hornet Sportabout  21   6  160 110  3.9 2.62 16.5  0  1    4    4
#> Valiant            21   6  160 110  3.9 2.62 16.5  0  1    4    4
```

2 Функция `iwalk()` позволяет вам использовать одну переменную, сохраняя путь вывода в именах.

```
temp <- tempfile()
dir.create(temp)

cyls <- split(mtcars, mtcars$cyl)
names(cyls) <- file.path(temp, paste0("cyl-", names(cyls), ".csv"))
iwalk(cyls, ~ write.csv(.x, .y))
```

Это можно было бы реализовать в виде единого конвейера, воспользовавшись преимуществами функции `set_names()`:

```
mtcars %>%
    split(mtcars$cyl) %>%
    set_names(~ file.path(temp, paste0("cyl-", .x, ".csv"))) %>%
    iwalk(~ write.csv(.x, .y))
```

3 В первом подходе

```
mtcars[nm] <- map2(trans, mtcars[nm], function(f, var) f(var))
```

список из двух функций (`trans`) и две колонки, выбранные из датафрейма (`mtcars[nm]`), передаются в функцию `map2()`. Функция `map2()` создает анонимную функцию (`f(var)`), которая применяет функции к переменным при выполнении итераций в `map2()` по (тем же) индексам. В левой части выражения два соответствующих элемента `mtcars` заменяются на выполненные преобразования.

В варианте с использованием функции `map()`

```
mtcars[nm] <- map(nm, ~ trans[[.x]](mtcars[[.x]]))
```

в основном происходит то же самое. Однако здесь производится итерирование по именам преобразований (`nm`) напрямую. Таким образом, колонки из датафрейма извлекаются во время итераций.

Помимо итерационного шаблона, два этих подхода отличаются в отношении возможностей именования аргументов в рамках параметра `.f`. В подходе с функцией `map2()` мы выполняем итерации по элементам `x` и `y`. Это позволило нам выбрать подходящие имена для заменителей, такие как `f` и `var`. В результате мы добились большей выразительности анонимной функции в обмен на более длинную запись. Нам кажется, что использование интерфейса формул таким образом является более предпочтительным в сравнении с малопонятной записью `mtcars[nm] <- map2(trans, mtcars[nm], ~ .x(.y))`.

В подходе с использованием функции `map()` мы осуществляем сопоставление по именам переменных, что не дает нам использовать заменители для функции и имен переменных. Синтаксис, характерный для формул, вкупе с местоимением `.x` дает довольно компактный стиль записи. Имена объектов и квадратные скобки явно демонстрируют применение преобразований к указанным колонкам в наборе данных `mtcars`. В этом случае итерирование по именам переменных может оказаться очень уместным, поскольку оно

позволяет подчеркнуть важность соответствия между именами элементов `trans` и `mtcars`. В совокупности с операцией замены в левой части выражения эта форма записи выглядит довольно понятной.

Подводя итог, можно сказать, что в ситуациях, когда можно использовать обе функции (`map()` и `map2()`), необходимо рассматривать разные факторы при осуществлении выбора.

4 Функция `write.csv()` возвращает `NULL`. Поскольку мы вызываем функцию ради ее побочного эффекта (создания файла CSV), функция `walk2()` здесь подошла бы лучше. В противном случае мы получим неинформативный список из значений `NULL`.

```
cyls <- split(mtcars, mtcars$cyl)
paths <- file.path(tempdir(), paste0("cyl-", names(cyls), ".csv"))

map2(cyls, paths, write.csv)
#> $`4`
#> NULL
#>
#> $`6`
#> NULL
#>
#> $`8`
#> NULL
```

9.6.3. Ответы на упражнения

1 Функция `is.na()` не является предикатом, поскольку она возвращает логический *вектор* той же длины, что и входной параметр, а не просто `TRUE` или `FALSE`.

Ближайшим аналогом этой функции можно назвать функцию `anyNA()`, которая всегда возвращает `TRUE` или `FALSE` в зависимости от наличия хотя бы одного пропущенного значения. Можно было бы себе также нафантазировать функцию `allNA()`, которая возвращала бы `TRUE`, в случае если бы все значения были пропущенными, но в ней было бы не так много пользы, так что в базовом R она отсутствует.

2 Цикл внутри функции `simple_reduce()` всегда начинается с двойки, а функция `seq()` умеет вести отсчет как вверх, так и *вниз*:

```
seq(2, 0)
#> [1] 2 1 0
seq(2, 1)
#> [1] 2 1
```

Таким образом, в случае с векторами единичной и нулевой длины при извлечении подмножества с помощью оператора `[[` будет возбуждаться ошибка выхода за границы диапазона. Во избежание такого поведения мы можем выполнить некоторые проверки еще до старта цикла.

```
simple_reduce <- function(x, f, default) {
    if (length(x) == 0L) return(default)
    if (length(x) == 1L) return(x[[1L]])

    out <- x[[1]]
    for (i in seq(2, length(x))) {
        out <- f(out, x[[i]])
    }
    out
}
```

Теперь наша функция `simple_reduce()` будет работать так, как и ожидалось:

```
simple_reduce(integer(0), `+`)
#> Error in simple_reduce(integer(0), `+`): argument "default" is missing, with no default
simple_reduce(integer(0), `+`, default = 0L)
#> [1] 0
simple_reduce(1, `+`)
#> [1] 1
simple_reduce(1:3, `+`)
#> [1] 6
```

3 Наша функция `span_r()` будет возвращать индекс (первой) самой длинной последовательности элементов вектора, для которых `f` возвращает `TRUE`. Если таких элементов не встретится, вернем `integer(0)`.

```
span_r <- function(x, f) {
    idx <- unname(map_lgl(x, ~ f(.x)))
    rle <- rle(idx)

    # Проверяем, есть ли значения TRUE
    if (!any(rle$values)) {
        return(integer(0))
    }

    # Находим длину самой продолжительной последовательности
    longest <- max(rle$lengths[rle$values])
    # Находим позицию первой самой длинной последовательности
    longest_idx <- which(rle$values & rle$lengths == longest)[1]

    ind_before_longest <- sum(rle$lengths[seq_len(longest_idx - 1)])

    out_start <- ind_before_longest + 1L
    out_end <- ind_before_longest + longest
    out_start:out_end
}

# Проверяем работу
span_r(c(0,  0,  0,  0,  0), is.na)
#> integer(0)
```

```
span_r(c(NA, 0,  0,  0,  0), is.na)
#> [1] 1
span_r(c(NA, 0, NA, NA, NA), is.na)
#> [1] 3 4 5
```

4 Обе функции будут принимать на вход вектор и функцию. Возврат функции осуществляется при помощи операции извлечения подмножеств.

```
arg_max <- function(x, f) {
    y <- map_dbl(x, f)
    x[y == max(y)]
}

arg_min <- function(x, f) {
    y <- map_dbl(x, f)
    x[y == min(y)]
}

arg_max(-10:5, function(x) x ^ 2)
#> [1] -10
arg_min(-10:5, function(x) x ^ 2)
#> [1] 0
```

5 Для применения функции к каждой колонке датафрейма можно воспользоваться функцией `purrr::modify()` (или `purrr::map_dfr()`), которая также возвращает результат в виде датафрейма. Чтобы ограничить применение функции только числовыми колонками, можно воспользоваться функцией `modify_if()`.

```
modify_if(mtcars, is.numeric, scale01)
```

9.7.3. Ответы на упражнения

1 В базовом виде функция `apply()` применяет функцию к измерениям массива. В случае с двумя измерениями применение будет осуществляться к строкам и столбцам массива. Давайте посмотрим на примере.

```
arr2 <- array(1:12, dim = c(3, 4))
rownames(arr2) <- paste0("row", 1:3)
colnames(arr2) <- paste0("col", 1:4)
arr2
#>      col1 col2 col3 col4
#> row1    1    4    7   10
#> row2    2    5    8   11
#> row3    3    6    9   12
```

При применении функции `head()` к первому измерению `arr2` (т. е. к строкам) результаты будут выведены в транспонированном виде по отношению к исходному массиву.

```
apply(arr2, 1, function(x) x[1:2])
#>      row1 row2 row3
#> col1    1    2    3
#> col2    4    5    6
```

Обратная история будет для применения функции ко второму измерению (колонкам):

```
apply(arr2, 2, function(x) x[1:2])
#>      col1 col2 col3 col4
#> row1    1    4    7   10
#> row2    2    5    8   11
```

Вывод функции `apply()` осуществляется сначала по измерениям, с которыми ведется работа, затем по результатам функции. Это может сбивать с толку при работе с массивами большой размерности.

2 Функция `eapply()` является вариацией функции `lapply()`, которая осуществляет итерации по (именованным) элементам окружения. В пакете `purrr` нет аналога функции `eapply()`, поскольку в нем главным образом представлены функции, оперирующие с векторами и функциями, а не с окружениями.

Функция `rapply()` применяет переданную функцию к элементам списка рекурсивно. При этом вы можете указать, к каким именно классам должна применяться функция (по умолчанию `classes = ANY`). Также можно задать поведение для других классов с помощью аргументов `how` и `default`. Ближайшим эквивалентом этой функции в пакете purrr является функция `modify_depth()`, позволяющая модифицировать элементы на заданной глубине вложенного списка.

Ответы на упражнения из главы 10

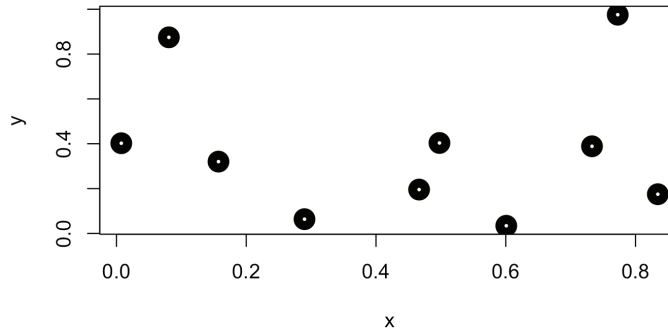
10.2.6. Ответы на упражнения

1 Как видите, `force(x)` – это то же самое, что и просто `x`. Как мы уже писали в этой книге, предпочтительно использовать явную форму, потому что *использование вызова функции* `force()` *явно выражает ваши намерения, а не напоминает случайно вставленную букву* `x`.

2 Начнем с функции `approxfun()`, поскольку она используется и внутри функции `ecdf()`.

Функция `approxfun()` принимает на вход комбинацию из наблюдений (значения `x` и `y`) и возвращает функцию линейной (или постоянной) интерполяции. Чтобы понять, что это значит, давайте сначала создадим случайный набор наблюдений.

```
x <- runif(10)
y <- runif(10)
plot(x, y, lwd = 10)
```

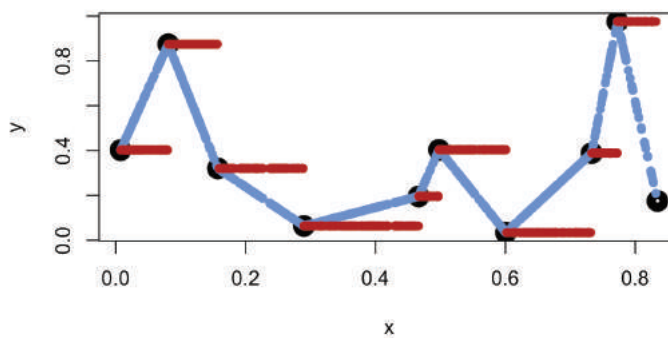


Теперь воспользуемся функцией `approxfun()` для создания функций линейной и постоянной интерполяции для наших значений `x` и `y`.

```
f_lin <- approxfun(x, y)
f_con <- approxfun(x, y, method = "constant")

# Обе функции в точности воспроизводят входные значения y
identical(f_lin(x), y)
#> [1] TRUE
identical(f_con(x), y)
#> [1] TRUE
```

При применении полученных функций к новым значениям `x` производится их сопоставление с линиями, соединяющими исходные значения `y` (линейная интерполяция), или с теми же значениями `y` (постоянная интерполяция).

```
x_new <- runif(1000)

plot(x, y, lwd = 10)
points(x_new, f_lin(x_new), col = "cornflowerblue", pch = 16)
points(x_new, f_con(x_new), col = "firebrick", pch = 16)
```



В то же время обе функции определены только в диапазоне `range(x)`, что показано ниже.

```
f_lin(range(x))
#> [1] 0.402 0.175
f_con(range(x))
#> [1] 0.402 0.175

(eps <- .Machine$double.neg.eps)
#> [1] 1.11e-16

f_lin(c(min(x) - eps, max(x) + eps))
#> [1] NA NA
f_con(c(min(x) - eps, max(x) + eps))
#> [1] NA NA
```

Для изменения такого поведения можно передать аргумент `rule = 2`. Это приведет к возвращению граничных значений для точек, выходящих за границы `range(x)`.

```
f_lin <- approxfun(x, y, rule = 2)
f_con <- approxfun(x, y, method = "constant", rule = 2)

f_lin(c(-Inf, Inf))
#> [1] 0.402 0.175
f_con(c(-Inf, Inf))
#> [1] 0.402 0.175
```

Также можно задать собственные граничные значения с помощью аргументов `yleft` и/или `yright`.

```
f_lin <- approxfun(x, y, yleft = 5)
f_con <- approxfun(x, y, method = "constant", yleft = 5, yright = -5)

f_lin(c(-Inf, Inf))
#> [1]  5 NA
f_con(c(-Inf, Inf))
#> [1]  5 -5
```

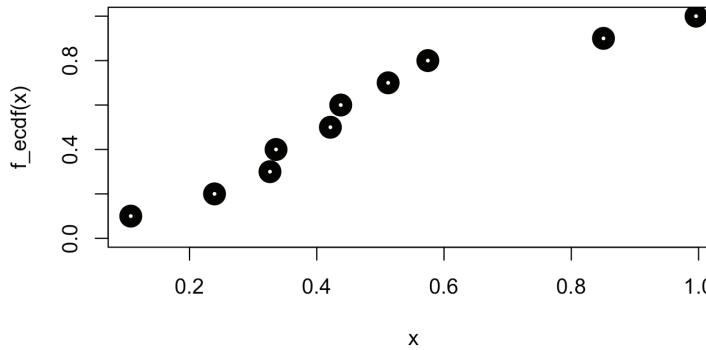
Кроме того, функция `approxfun()` позволяет сдвинуть значения `y` при выбранном аргументе `method = "constant"` между левым и правым значениями. Согласно документации, это обеспечивает компромисс между левым и правым непрерывными шагами.

```
f_con <- approxfun(x, y, method = "constant", f = .5)

plot(x, y, lwd = 10)
points(x_new, f_con(x_new), pch = 16)
```

Наконец, аргумент `ties` позволяет агрегировать значения `y` в случае, если для одного `x` их насчитывается несколько. Допустим, мы можем применить функцию `mean()` для агрегирования средних значений `y` перед их использованием в интерполяции: `approxfun(x = c(1,1,2), y = 1:3, ties = mean)`.

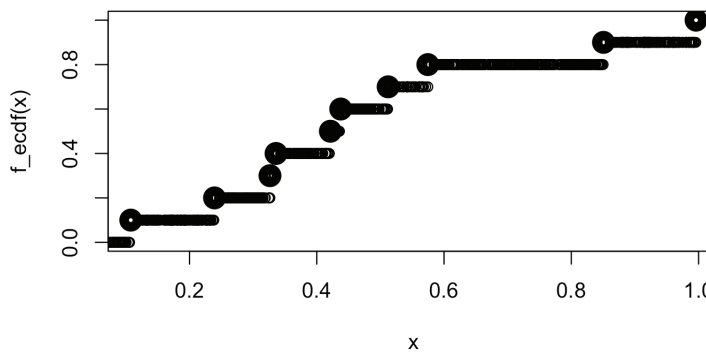
Теперь перейдем к функции `ecdf()`. Ее название является сокращением от *empirical cumulative distribution function* (эмпирическая накопительная функция распределения). Для числового вектора со значениями плотности функция `ecdf()` создает пары (`x`, `y`) для узлов функции плотности, после чего передает их в функцию `approxfun()`, которая вызывается со специфическим набором параметров (`approxfun(vals, cumsum(tabulate(match(x, vals)))/n, method = "constant", yleft = 0, yright = 1, f = 0, ties = "ordered")`).

```
x <- runif(10)
f_ecdf <- ecdf(x)
class(f_ecdf)
#> [1] "ecdf"     "stepfun"  "function"

plot(x, f_ecdf(x), lwd = 10, ylim = 0:1)
```



Новые значения затем сопоставляются со значениями `y` следующих наименьших значений `x` из входных данных.

```
x_new <- runif(1000)

plot(x, f_ecdf(x), lwd = 10, ylim = 0:1)
points(x_new, f_ecdf(x_new), ylim = 0:1)
```



3 В данном упражнении `pick(i)` выступает в качестве фабрики функций, возвращающей требуемую функцию извлечения подмножества.

```
pick <- function(i) {
    force(i)

    function(x) x[[i]]
}

x <- 1:3
identical(x[[1]], pick(1)(x))
#> [1] TRUE
identical(
    lapply(mtcars, function(x) x[[5]]),
    lapply(mtcars, pick(5))
)
#> [1] TRUE
```

4 Первый момент тесно связан со средним значением и описывает среднее отклонение от среднего значения, равного нулю (в пределах числовой погрешности). Второй момент описывает дисперсию входных данных. Если мы хотим сравнить ее с `var()`, нужно отменить поправку Бесселя путем умножения на $\frac{N-1}{N}$.

```
moment <- function(i) {
    force(i)

    function(x) sum((x - mean(x)) ^ i) / length(x)
}

m1 <- moment(1)
m2 <- moment(2)

x <- runif(100)
all.equal(m1(x), 0)  # убрали stopifnot() для ясности
#> [1] TRUE
all.equal(m2(x), var(x) * 99 / 100)
#> [1] TRUE
```

5 Без захвата и инкапсуляции окружения замыкания счетчики будут храниться в глобальном окружении, где они могут быть перезаписаны или удалены, а также могут мешать работе других счетчиков.

```
new_counter2()
#> [1] 1
i
#> [1] 1
new_counter2()
#> [1] 2
```

```
i
#> [1] 2

i <- 0
new_counter2()
#> [1] 1
i
#> [1] 1
```

6 Без использования оператора присваивания в родительском окружении (`<<-`) счетчик всегда будет возвращать единицу. Счетчик всегда будет инициализироваться в новом окружении выполнения в рамках одного замыкающего окружения, содержащего неизменное значение для переменной `i` (в нашем случае ноль).

```
new_counter_3 <- new_counter3()

new_counter_3()
#> [1] 1
new_counter_3()
#> [1] 1
```

10.3.4. Ответы на упражнения

1 Обе функции помогут вам при оформлении вывода, например на графиках, и делают они это путем возвращения функции форматирования.

Функция `ggplot2::label_bquote()` принимает относительно простые выражения `plotmath` и использует их для построения меток графиков в пакете `ggplot2`. Поскольку эта функция используется в `ggplot2`, она должна возвращать функцию с `class = "labeller"`.

Функция `scales::number_format()` принудительно вычисляет все аргументы с помощью функции `force()`. По сути, это параметризованная обертка для функции `scales::number()`, помогающая вам соответствующим образом форматировать числа. На выходе этой функции будет обычная функция.

10.4.4. Ответы на упражнения

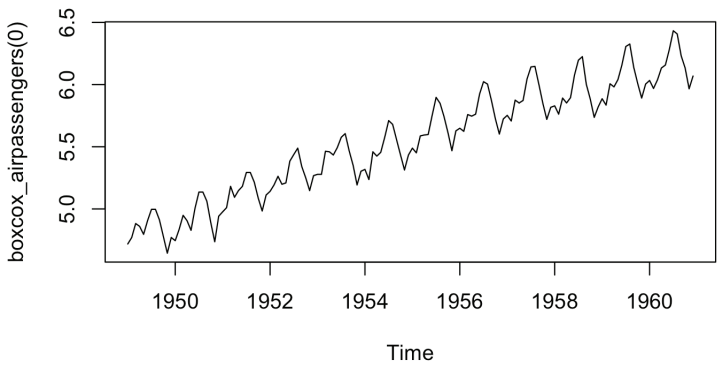
1 Функция `boot_model()` на выходе возвращает функцию, а всякий раз, когда вы возвращаете функцию, вам нужно убедиться, что все входы явно вычисляются. Здесь это происходит автоматически, поскольку мы задействуем аргументы `df` и `formula` в функции `lm()` еще перед осуществлением возврата.

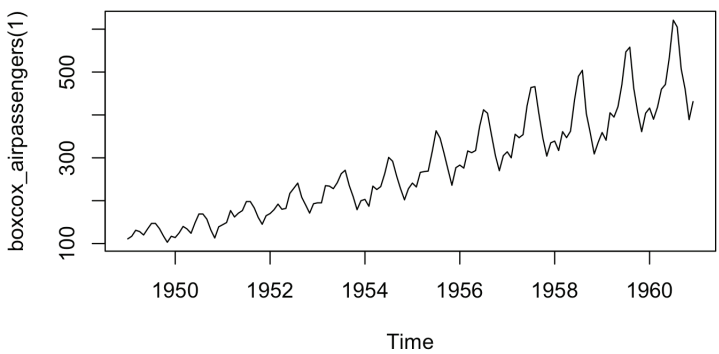
```
boot_model <- function(df, formula) {
    mod <- lm(formula, data = df)
    fitted <- unname(fitted(mod))
    resid <- unname(resid(mod))
    rm(mod)
```

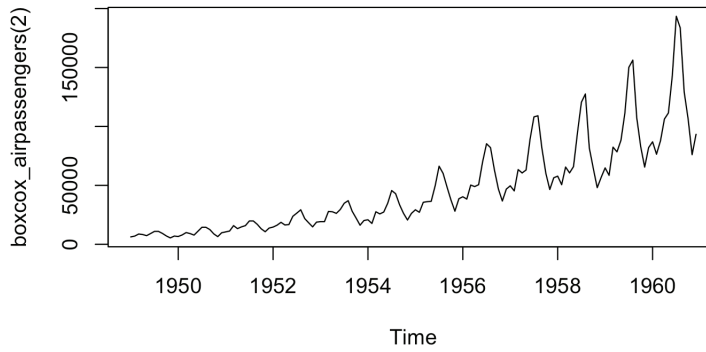
```
    function() {
        fitted + sample(resid)
    }
}
```

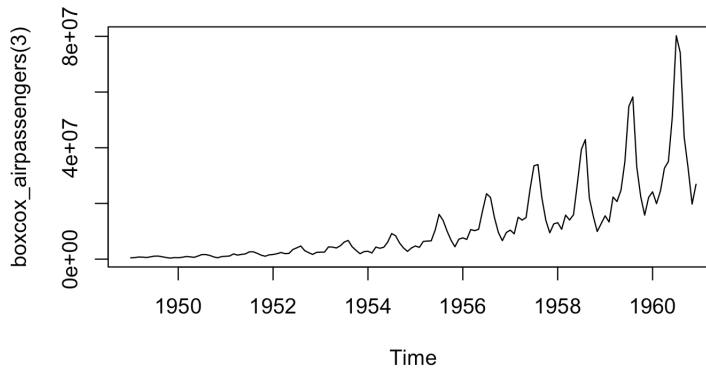
2 Функция `boxcox3()` возвращает функцию, где аргумент `x` фиксирован (хотя он не вычисляется явно, так что его можно изменить позже). Это позволяет применить и проверить различные преобразования для разных входных значений и дать им описательные имена.

```
boxcox_airpassengers <- boxcox3(AirPassengers)

plot(boxcox_airpassengers(0))
plot(boxcox_airpassengers(1))
plot(boxcox_airpassengers(2))
plot(boxcox_airpassengers(3))
```









3 Ранее мы определили функцию `boot_permute()` следующим образом:

```
boot_permute <- function(df, var) {
    n <- nrow(df)
    force(var)

    function() {
        col <- df[[var]]
        col[sample(n, replace = TRUE)]
    }
}
```

Нам не стоит беспокоиться, что она будет хранить копию данных, поскольку на самом деле она этого не делает. Это просто имя, которое указывает на тот же объект в памяти.

```
boot_mtcars1 <- boot_permute(mtcars, "mpg")

lobstr::obj_size(mtcars)
#> 7,208 B
lobstr::obj_size(boot_mtcars1)
#> 20,248 B
lobstr::obj_sizes(mtcars, boot_mtcars1)
```

```
#> *  7,208 B
#> * 13,040 B
```

4 Давайте вспомним определения функций `ll_poisson1()` и `ll_poisson2()`, а также тестовые данные `x1`:

```
ll_poisson1 <- function(x) {
    n <- length(x)

    function(lambda) {
        log(lambda) * sum(x) - n * lambda - sum(lfactorial(x))
    }
}

ll_poisson2 <- function(x) {
    n <- length(x)
    sum_x <- sum(x)
    c <- sum(lfactorial(x))

    function(lambda) {
        log(lambda) * sum_x - n * lambda - c
    }
}

x1 <- c(41, 30, 31, 38, 29, 24, 30, 29, 31, 38)
```

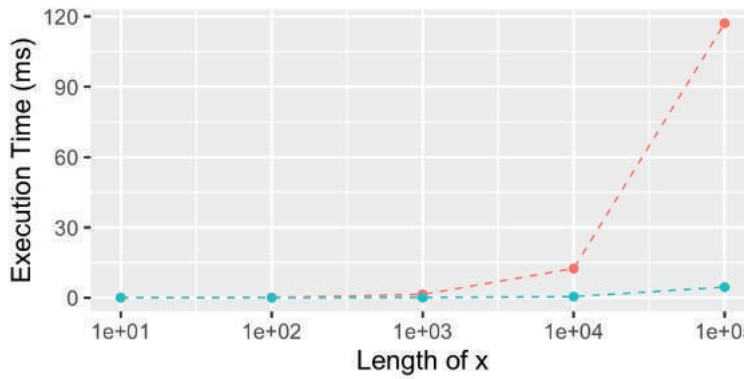
Проверка производительности показала двукратный рост быстродействия функции `ll_poisson2()` в сравнении с `ll_poisson1()`:

```
bench::mark(
    llp1 = optimise(ll_poisson1(x1), c(0, 100), maximum = TRUE),
    llp2 = optimise(ll_poisson2(x1), c(0, 100), maximum = TRUE)
)
#> # A tibble: 2 x 6
#>   expression      min   median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
#> 1 llp1         49.3µs   54.7µs    17761.    12.8KB     22.1
#> 2 llp2         26.8µs   30.1µs    31795.        0B     22.3
```

Поскольку с ростом массива исходных данных будет увеличиваться и объем работы, выполняемой функцией `ll_poisson1()`, мы можем ожидать дальнейшего спада ее быстродействия. Ниже показано, что с увеличением объема данных в `x1` до 100 000 функция `ll_poisson2()` начинает опережать `ll_poisson1()` примерно в 20 раз.

```
bench_poisson <- function(x_length) {
    x <- rpois(x_length, 100L)

    bench::mark(
```

```
        llp1 = optimise(ll_poisson1(x), c(0, 100), maximum = TRUE),
        llp2 = optimise(ll_poisson2(x), c(0, 100), maximum = TRUE),
        time_unit = "ms"
    )
}

performances <- map_dfr(10^(1:5), bench_poisson)

df_perf <- tibble(
    x_length = rep(10^(1:5), each = 2),
    method   = rep(attr(performances$expression, "description"), 5),
    median   = performances$median
)

ggplot(df_perf, aes(x_length, median, col = method)) +
    geom_point(size = 2) +
    geom_line(linetype = 2) +
    scale_x_log10() +
    labs(
        x = "Length of x",
        y = "Execution Time (ms)",
        color = "Method"
    ) +
    theme(legend.position = "top")
```



10.5.1. Ответы на упражнения

1 Правильный ответ *e (это зависит...)*. Обычно функция `with()` используется с датафреймом, так что можно было бы предположить, что правильным ответом будет `b`, но если `x` – это список, правильными могут быть все варианты.

```
f <- mean
z <- 1
x <- list(f = mean, z = 1)
```

```
identical(with(x, f(z)), x$f(x$z))
#> [1] TRUE
identical(with(x, f(z)), f(x$z))
#> [1] TRUE
identical(with(x, f(z)), x$f(z))
#> [1] TRUE
identical(with(x, f(z)), f(z))
#> [1] TRUE
```

2 Функция `attach()` добавляет аргумент `funs` к пути поиска. Таким образом, созданные функции будут найдены раньше, чем их аналоги из базового пакета. Кроме того, они не могут быть случайно перезаписаны функциями с аналогичными именами в глобальном окружении. Досадным недостатком использования функции `attach()` является возможность подключить один и тот же объект несколько раз, что вынуждает пользоваться функцией `detach()` чаще, чем можно было бы.

```
attach(funs)
#> The following objects are masked from package:base:
#>
#>     mean, sum
attach(funs)
#> The following objects are masked from funs (pos = 3):
#>
#>     mean, sum
#>
#> The following objects are masked from package:base:
#>
#>     mean, sum

head(search())
#> [1] ".GlobalEnv"      "funs"            "funs"            "package:ggplot2"
#> [5] "package:purrr"   "package:dplyr"
detach(funs)
detach(funs)
```

Напротив, функция `rlang::env_bind()` просто добавляет функции из `funs` в глобальное окружение. Никаких других побочных эффектов эта функция не имеет, и функции перезаписываются при определении новых функций с такими же именами.

```
env_bind(globalenv(), !!!funs)
head(search())
#> [1] ".GlobalEnv"      "package:ggplot2" "package:purrr"   "package:dplyr"
#> [5] "package:rlang"   "package:stats"
```

Ответы на упражнения из главы 11

11.2.3. Ответы на упражнения

1 В R многие функции векторизованы. Векторизация означает, во-первых, то, что функция обычно на вход принимает вектор или векторы и выполняет какие-то операции с ними поэлементно. Во-вторых, векторизация чаще всего предполагает, что эти операции над элементами реализованы на C или Fortran, что подразумевает их высокое быстродействие.

Однако, несмотря на свое имя, функция `Vectorize()` не способна ускорить выполнение переданной ей функции. Она просто меняет формат входных аргументов (`vectorize.args`), чтобы по ним можно было проходить итерациями.

Давайте взглянем на пример из документации:

```
vrep <- Vectorize(rep.int)
vrep
#> function (x, times)
#> {
#>   args <- lapply(as.list(match.call())[-1L], eval, parent.frame())
#>   names <- if (is.null(names(args)))
#>     character(length(args))
#>   else names(args)
#>   dovec <- names %in% vectorize.args
#>   do.call("mapply", c(FUN = FUN, args[dovec],
#>                       MoreArgs = list(args[!dovec]),
#>                       SIMPLIFY = SIMPLIFY, USE.NAMES = USE.NAMES))
#> }
#> <environment: 0x558902db65d0>

# Применение
vrep(1:2, 3:4)
#> [[1]]
#> [1] 1 1 1
#>
#> [[2]]
#> [1] 2 2 2 2
```

Функция `Vectorize()` обеспечивает удобную и краткую нотацию для выполнения итераций по нескольким аргументам, но при этом обладает серьезными недостатками, мешающими использовать ее на постоянной основе. За подробностями можно обратиться по адресу https://www.jimhester.com/post/2018-04-12-vectorize.

2 Функция `possibly()` изменяет функции таким образом, чтобы они возвращали значение по умолчанию (`otherwise`) в случае возникновения ошибки и подавляли любые сообщения об ошибках (`quiet = TRUE`).

Читая исходный код функции, можно обнаружить, что функция `possibly()` в своей реализации использует функцию `purrr::as_mapper()`. Это позволяет передавать ей не только функции, но также формулы или атомарные векторы с помощью того же синтаксиса, принятого в пакете `purrr`. Помимо этого, новое значение по умолчанию (`otherwise`) вычисляется лишь один раз, что делает его (почти) неизменным.

```
possibly
#> function (.f, otherwise, quiet = TRUE)
#> {
#>     .f <- as_mapper(.f)
#>     force(otherwise)
#>     function(...) {
#>         tryCatch(.f(...), error = function(e) {
#>             if (!quiet)
#>                 message("Error: ", e$message)
#>             otherwise
#>         }, interrupt = function(e) {
#>             stop("Terminated by user", call. = FALSE)
#>         })
#>     }
#> }
#> <bytecode: 0x7fdddd6159b0>
#> <environment: namespace:purrr>
```

Основной функционал `possibly()` заключен в блоке функции `base::tryCatch()`. Здесь переданная функция (`.f`) получает свою обертку с особой обработкой ошибок и прерываний.

3 Функция `safely()` изменяет функции таким образом, чтобы они возвращали список, содержащий элементы `result` и `error`. Работает она примерно так же, как и функция `possibly()`, но, помимо использования функции `as_mapper()`, функция `safely()` также принимает аргументы `otherwise` и `quiet`. Чтобы результаты и ошибки возвращались в согласованной манере, блок с функцией `tryCatch()` в обоих случаях возвращает список с похожей структурой. В случае успеха элемент списка `error` будет содержать значение `NULL`, а в случае ошибки элемент `result` будет содержать значение аргумента `otherwise`, которое по умолчанию равно `NULL`.

Поскольку блок функции `tryCatch()` здесь скрыт за внутренним вызовом функции `purrr:::capture_error()`, мы приводим и ее исходный код:

```
safely
#> function (.f, otherwise = NULL, quiet = TRUE)
#> {
#>     .f <- as_mapper(.f)
#>     function(...) capture_error(.f(...), otherwise, quiet)
#> }
#> <bytecode: 0x7fdddd89be78>
#> <environment: namespace:purrr>
```

```
purrr:::capture_error
#> function (code, otherwise = NULL, quiet = TRUE)
#> {
#>     tryCatch(list(result = code, error = NULL), error = function(e) {
#>         if (!quiet)
#>             message("Error: ", e$message)
#>         list(result = otherwise, error = e)
#>     }, interrupt = function(e) {
#>         stop("Terminated by user", call. = FALSE)
#>     })
#> }
#> <bytecode: 0x7fdddd901a50>
#> <environment: namespace:purrr>
```

В этой книге, а также в документации к функции `safely()` показано, как можно эффективно использовать такое поведение, например при выполнении подгонки нескольких моделей.

11.3.1. Ответы на упражнения

1 Обе команды будут выводить точку на экран после каждых десяти загрузок и выполняться одинаковое количество времени, так что отличия между ними кроются в нюансах.

В первом случае к функции `download.file()` сначала добавляется функционал вывода точки. После этого к уже измененной функции добавляется задержка. Это означает, что на вывод точек тоже будет распространяться задержка, а первая точка будет выведена только при начале загрузки десятого файла.

Во втором случае сначала добавляется задержка, а затем – вывод точки. Это позволит вывести первую точку сразу по окончании загрузки девятого файла, после чего возникнет небольшая пауза и только затем начнется загрузка десятого.

2 Мемоизация функции `file.download()` сработает только в случае, если файлы будут неизменными, т. е. по одинаковым адресам будут располагаться всегда одни и те же файлы. Если это не так, никакого смысла в мемоизации нет. И даже если это так, при использовании мемоизации мы вынуждены будем хранить результаты в памяти, а файлы могут быть и большими.

Из этого можно заключить, что мемоизация функции `file.download()` редко может принести пользу. Единственным исключением может быть ситуация, когда вы загружаете небольшие статические файлы множество раз.

3 Начнем с функции, сообщающей о различиях в векторах, содержащих имена файлов:

```
dir_compare <- function(old, new) {
    if (setequal(old, new)) {
        return()
```

```
    }

    added <- setdiff(new, old)
    removed <- setdiff(old, new)

    changes <- c(
        if (length(added) > 0) paste0(" * '", added, "' was added"),
        if (length(removed) > 0) paste0(" * '", removed ,
                                        "' was removed")
    )
    message(paste(changes, collapse = "\n"))
}

dir_compare(c("x", "y"), c("x", "y"))
#> NULL
dir_compare(c("x", "y"), c("x", "a"))
#>  * 'a' was added
#>  * 'y' was removed
```

Теперь обернем ее в функциональный оператор:

```
track_dir <- function(f) {
    force(f)
    function(...) {
        dir_old <- dir()
        on.exit(dir_compare(dir_old, dir()), add = TRUE)

        f(...)
    }
}
```

Попробуем, что получилось, создав обертки для функций `file.create()` и `file.remove()`:

```
file_create <- track_dir(file.create)
file_remove <- track_dir(file.remove)

file_create("delete_me")
#>  * 'delete_me' was added
#> [1] TRUE
file_remove("delete_me")
#>  * 'delete_me' was removed
#> [1] TRUE
```

Для создания полновесной версии функции `track_dir()` мы могли бы предусмотреть возможность установки аргументов `full.names` и `recursive` функции `dir()` в значение `TRUE`. Это позволило бы нам отслеживать создание/удаление скрытых файлов и файлов, содержащихся внутри папок в рабочей директории.

Также мы могли бы отслеживать другие глобальные эффекты, включая изменения, касающиеся:

- пути поиска и возможных конфликтов (`conflicts()`);
- функций `options()` и `par()`, модифицирующих глобальные параметры;
- пути рабочей директории;
- переменных окружения.

4 Наш функциональный оператор `logger()` будет принимать на вход функцию и путь к файлу. Первая строка со временем будет записываться в файл по пути `log_path` при вызове функции `logger()`, а последующие будут дописываться в тот же файл при каждом вызове новой функции.

```
append_line <- function(path, ...) {
    cat(..., "\n", sep = "", file = path, append = TRUE)
}

logger <- function(f, log_path) {
    force(f)
    force(log_path)

    append_line(log_path, "created at: ", as.character(Sys.time()))
    function(...) {
        append_line(log_path, "called at: ", as.character(Sys.time()))
        f(...)
    }
}
```

Теперь проверим, как все работает, на примере функции `mean()`.

```
log_path <- tempfile()
mean2 <- logger(mean, log_path)
Sys.sleep(5)
mean2(1:4)
#> [1] 2.5
Sys.sleep(1)
mean2(1:4)
#> [1] 2.5

readLines(log_path)
#> [1] "created at: 2021-05-02 09:44:23" "called at: 2021-05-02 09:44:28"
#> [3] "called at: 2021-05-02 09:44:29"
```

5 В книге функция `delay_by()` была определена следующим образом:

```
delay_by <- function(f, amount) {
    force(f)
    force(amount)

    function(...) {
        Sys.sleep(amount)
```

```
        f(...)
    }
}
```

Чтобы функция, созданная функцией `delay_by()`, могла отслеживать, что с момента последнего ее запуска прошло определенное время, внесем в нашу реализацию функции `delay_atleast()` три небольших изменения, помеченных комментариями в коде, представленном ниже.

```
delay_atleast <- function(amount, f) {
    force(f)
    force(amount)

    # Сохраняем последнее время вызова функции
    last_time <- NULL

    # Возвращаем измененную функцию
    function(...) {
        if (!is.null(last_time)) {
            wait <- (last_time - Sys.time()) + amount
            if (wait > 0) {
                Sys.sleep(wait)
            }
        }

        # Обновляем время после окончания работы функции
        on.exit(last_time <<- Sys.time())

        f(...)
    }
}
```

Ответы на упражнения из главы 13

13.2.1. Ответы на упражнения

1 Из-за схемы именования `generic.class()`, принятой в системе S3, эти функции могут на первый взгляд показаться похожими, хотя по факту они никак не связаны:

- `t.test()` представляет собой *обобщенную функцию*, выполняющую расчет t-критерия Стьюдента;
- `t.data.frame()` – это *метод*, вызываемый обобщенной функцией `t()` с целью транспонирования датафрейма.

Согласно правилам диспетчеризации методов, принятым в S3, функция `t.test()` также вызывалась бы при применении обобщенной функции `t()` к объекту класса `test`.

2 В последние годы наиболее распространенным при именовании функций и переменных в R стал «змеиный» стиль (с использованием символа подчеркивания). При этом в именах некоторых функций по-прежнему присутствуют точки, что вносит элемент неразберихи в код на R.

```
# Некоторые базовые функции R, в именах которых присутствуют точки
install.packages()
read.csv()

list.files()
download.file()

data.frame()
as.character()
Sys.Date()

all.equal()

do.call()
on.exit()
```

3 В функции `as.data.frame.data.frame()` реализуется метод `data.frame()` для обобщенной функции `as.data.frame()`, приводящий объект к типу датафрейма.

Имя функции, конечно, сбивает с толку, поскольку оно даже ясно не передает тип функции (обычная функция, обобщенная или метод). Даже если мы предположим, что это метод, количество точек в имени не позволяет явным образом разделить части, связанные с обобщенной функцией и с классом. Может, это метод `data.frame.data.frame()` обобщенной функции `as()`? Или метод `frame.data.frame()` обобщенной функции `as.data()`?

Этой путаницы можно избежать, если использовать «змеиный» стиль (с использованием символа подчеркивания) применительно к именам классов и функций.

4 `mean()` представляет собой обобщенную функцию, которая будет выбирать подходящий метод на основании класса входного аргумента. Переменная `some_days` обладает классом `Date`, так что для расчета средней даты по `some_days` будет вызван метод `mean.Date(some_days)`.

После удаления атрибута класса у переменной `some_date` с помощью функции `unclass()` будет выбран метод по умолчанию. В результате метод `mean.default(unclass(some_days))` рассчитает среднее значение на основании чисел двойной точности.

5 Это выражение вернет объект класса `ecdf` (*empirical cumulative distribution function* – эмпирическая накопительная функция распределения) с родительскими классами `stepfun` и `function`. В основе объекта `ecdf` лежит базовый тип `closure` (функция). Выражение, которое было использовано для его создания (`rpois(100, 10)`), сохраняется в атрибуте `call`.

```
typeof(x)
#> [1] "closure"

attributes(x)
#> $class
#> [1] "ecdf"     "stepfun"  "function"
#>
#> $call
#> ecdf(rpois(100, 10))
```

5 Этот код возвращает объект `table`, основанный на базе типа `integer`. Атрибут `dimnames` используется для именования элементов целочисленного вектора.

```
typeof(x)
#> [1] "integer"

attributes(x)
#> $dim
#> [1] 10
#>
#> $dimnames
#> $dimnames[[1]]
#>  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
#>
#>
#> $class
#> [1] "table"
```

13.3.4. Ответы на упражнения

1 Датафреймы построены на базе именованных списков векторов одинаковой длины. Помимо атрибутов `class` и `names` (имена колонок), в них присутствует атрибут `row.names`. Это должен быть символьный вектор той же длины, что и другие векторы.

Нам необходимо предоставить конструктору количество строк, чтобы была возможность создавать датафреймы с нулем колонок и определенным количеством строк.

В результате получим следующий конструктор:

```
new_data.frame <- function(x, n, row.names = NULL) {
    # Проверка на список
    stopifnot(is.list(x))

    # Проверка на длину векторов
    # (Здесь допускается случай, когда x равен 0)
    stopifnot(all(lengths(x) == n))

    if (is.null(row.names)) {
```

```
        # Используем специальную вспомогательную функцию из базового R
        row.names <- .set_row_names(n)
    } else {
        # Проверка на символьный вектор правильной длины
        stopifnot(is.character(row.names), length(row.names) == n)
    }

    structure(
        x,
        class = "data.frame",
        row.names = row.names
    )
}

# Проверка
x <- list(a = 1, b = 2)
new_data.frame(x, n = 1)
#>   a b
#> 1 1 2
new_data.frame(x, n = 1, row.names = "l1")
#>    a b
#> l1 1 2

# Создаем датафрейм с 0 колонок и 2 строками
new_data.frame(list(), n = 2)
```

Существуют два дополнительных ограничения, которые мы могли бы реализовать, если бы действовали предельно строго, – это требование на уникальность имен строк и имен столбцов.

2 Функция `base::factor()` молча преобразует такие значения в `NA`:

```
factor(c("a", "b", "c"), levels = c("a", "b"))
#> [1] a    b    <NA>
#> Levels: a b
```

Вспомогательная функция `factor()`, включая конструктор (`new_factor()`) и валидатор (`validate_factor()`), показана в книге. Однако поскольку наша задача состоит в возбуждении ошибок во вспомогательной функции, мы просто повторим прежний код:

```
# Упрощенная версия функции factor(), как в книге
factor <- function(x = character(), levels = unique(x)) {
  ind <- match(x, levels)
  validate_factor(new_factor(ind, levels))
}
```

В улучшенной версии вспомогательной функции `factor()` мы будем выводить понятное сообщение об ошибке.

```
factor2 <- function(x, levels = unique(x)) {
    new_levels <- match(x, levels)

    # Ошибка, если в levels не включены все значения
    missing <- unique(setdiff(x, levels))
    if (length(missing) > 0) {
        stop(
          "The following values do not occur in the levels of x: ",
          paste0("'", missing, "'", collapse = ", "), ".",
          call. = FALSE
        )
    }

    validate_factor(new_factor(new_levels, levels))
}

# Проверка
factor2(c("a", "b", "c"), levels = c("a", "b"))
#> Error: The following values do not occur in the levels of x: 'c'.
```

3 Оригинальная реализация функции (`base::factor()`) допускает более гибкий подход в отношении `x`. В ней выполняется приведение `x` к символьному виду или замена на `character(0)` (в случае `NULL`). Также в этой функции обеспечивается уникальность значений в `levels`. Это достигается с помощью инструкции `base::levels<-`, которая выдает ошибку в случае передачи неуникальных значений.

4 При использовании переменных факторов, представляющих категориальные данные, в статистических моделях они обычно кодируются в виде *фиктивных* (dummy) переменных, и по умолчанию каждый уровень сравнивается с первым уровнем фактора. В то же время кодировки (`"contrasts"`) могут быть разными (см. https://en.wikipedia.org/wiki/Contrast_(statistics)).

В интерфейсе формул в R вы можете обернуть фактор в `stats::C()` и задать нужный вам вариант. Также можно задать атрибут `contrasts` факторной переменной, принимающий на вход матрицу (за подробностями можно обратиться к справке `?contr.helmert`).

Конструктор `new_factor()` мы привели в книге:

```
# Конструктор new_factor() из книги
new_factor <- function(x = integer(), levels = character()) {
    stopifnot(is.integer(x))
    stopifnot(is.character(levels))

    structure(
        x,
        levels = levels,
        class = "factor"
    )
}
```

Наш обновленный конструктор `new_factor()` будет принимать аргумент `contrasts` в виде числовой матрицы или `NULL` (по умолчанию).

```
# Обновленный конструктор new_factor()
new_factor <- function(
    x = integer(),
    levels = character(),
    contrasts = NULL
) {
    stopifnot(is.integer(x))
    stopifnot(is.character(levels))

    if (!is.null(constrasts)) {
        stopifnot(is.matrix(contrasts) && is.numeric(contrasts))
    }

    structure(
        x,
        levels = levels,
        class = "factor",
        contrasts = contrasts
    )
}
```

5 Эта функция преобразовывает числовой ввод в римские цифры. Построена она на базе целочисленного типа, что дает нам следующий конструктор.

```
new_roman <- function(x = integer()) {
    stopifnot(is.integer(x))
    structure(x, class = "roman")
}
```

В документации сказано, что функция корректно обрабатывает числовые значения в диапазоне от 1 до 3899. Мы включим эту информацию в наш валидатор.

```
validate_roman <- function(x) {
    values <- unclass(x)

    if (any(values < 1 | values > 3899)) {
        stop(
          "Roman numbers must fall between 1 and 3899.",
          call. = FALSE
        )
    }

    x
}
```

Для удобства мы также позволим пользователям передавать значения вспомогательной функции.

```
roman <- function(x = integer()) {
    x <- as.integer(x)

    validate_roman(new_roman(x))
}

# Проверка
roman(c(1, 753, 2019))
#> [1] I       DCCLIII MMXIX
roman(0)
#> Error: Roman numbers must fall between 1 and 3899.
```

13.4.4. Ответы на упражнения

1 Определить, что функция `t.test()` является обобщенной, можно по тому, что она вызывает функцию `UseMethod()`:

```
t.test
#> function (x, ...)
#> UseMethod("t.test")
#> <bytecode: 0x7fdf0d5ac2c8>
#> <environment: namespace:stats>

# или просто вызов
ftype(t.test)
#> [1] "S3"      "generic"

# то же и для t()
t
#> function (x)
#> UseMethod("t")
#> <bytecode: 0x7fdf0fda1f38>
#> <environment: namespace:base>
```

Любопытно, что в R присутствуют вспомогательные функции, возвращающие имена функций, которые внешне похожи на методы, но таковыми не являются:

```
tools::nonS3methods("stats")
#> [1] "anova.lmlist"        "expand.model.frame"  "fitted.values"
#> [4] "influence.measures"  "lag.plot"            "t.test"
#> [7] "plot.spec.phase"     "plot.spec.coherency"
```

При создании объекта класса `test` функция `t()` диспетчеризуется на метод `t.default()`. Это происходит из-за того, что функция `UseMethod()` просто ищет функции с именем `paste0("generic", ".", c(class(x), "default"))`.

```
x <- structure(1:10, class = "test")

t(x)
#>      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
#> [1,]    1    2    3    4    5    6    7    8    9    10
#> attr(,"class")
#> [1] "test"
```

Однако в старых версиях R (до версии 4.0.0) это поведение было несколько иным. Вместо диспетчеризации на метод `t.default()` обобщенная функция `t.test()` по ошибке воспринималась как метод `t()`, что приводило к выбору метода `t.test.default()` или (если таковой имелся) метода `t.test.test()`.

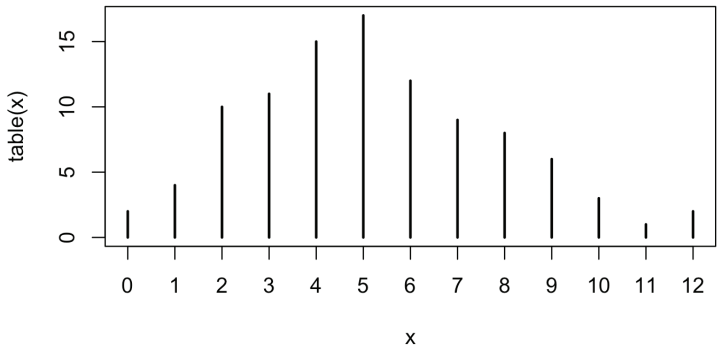
```
# Вывод в R версии 3.6.2
x <- structure(1:10, class = "test")
t(x)
#>
#>  One Sample t-test
#>
#> data:  x
#> t = 5.7446, df = 9, p-value = 0.0002782
#> alternative hypothesis: true mean is not equal to 0
#> 95 percent confidence interval:
#>  3.334149 7.665851
#> sample estimates:
#> mean of x
#>       5.5

t.test.test <- function(x) "Hi!"
t(x)
#>[1] "Hi!"
```

2 Это легко узнать с помощью функции `sloop::s3_methods_class()`:

```
s3_methods_class("table")
#> # A tibble: 10 x 4
#>    generic       class visible source
#>    <chr>         <chr> <lgl>   <chr>
#>  1 [             table TRUE    base
#>  2 aperm         table TRUE    base
#>  3 as.data.frame table TRUE    base
#>  4 Axis          table FALSE   registered S3method
#>  5 lines         table FALSE   registered S3method
#>  6 plot          table FALSE   registered S3method
#>  7 points        table FALSE   registered S3method
#>  8 print         table TRUE    base
#>  9 summary       table TRUE    base
#> 10 tail          table FALSE   registered S3method
```

Интересно, что в классе `table` присутствует множество методов для графической визуализации данных.

```
x <- rpois(100, 5)
plot(table(x))
```



3 Воспользуемся той же функцией, что и выше:

```
s3_methods_class("ecdf")
#> # A tibble: 4 x 4
#>   generic  class visible source
#>   <chr>    <chr> <lgl>   <chr>
#> 1 plot     ecdf  TRUE    stats
#> 2 print    ecdf  FALSE   registered S3method
#> 3 quantile ecdf  FALSE   registered S3method
#> 4 summary  ecdf  FALSE   registered S3method
```

В основном здесь присутствуют методы для отображения (`plot()`, `print()`, `summary()`), но вы также можете извлечь нужные квантили с помощью метода `quantile()`.

4 Несложные изыскания (и размышления о том, какие функции в R являются наиболее популярными) привели нас к догадке о том, что наибольшее количество методов должно быть у обобщенной функции `print()`.

```
nrow(s3_methods_generic("print"))
#> [1] 286
nrow(s3_methods_generic("summary"))
#> [1] 38
nrow(s3_methods_generic("plot"))
#> [1] 34
```

Проверим нашу догадку, воспользовавшись средствами, изученными в этой и предыдущих главах.

```
library(purrr)

ls(all.names = TRUE, env = baseenv()) %>%
    mget(envir = baseenv()) %>%
    keep(is_function) %>%
    names() %>%
    keep(is_s3_generic) %>%
    map(~ set_names(nrow(s3_methods_generic(.x)), .x)) %>%
    flatten_int() %>%
    sort(decreasing = TRUE) %>%
    head()
#>        print       format            [ as.character      summary         plot
#>          286          118           58           40           38           34
```

5 Давайте по порядку. При вызове `g.default(x)` напрямую вы, как и ожидали, получите `c(1, 1)`.

Значение, привязанное к `x`, берется из аргумента, а значение для `y` – из глобального окружения.

```
g.default(x)
#> x y
#> 1 1
```

Но при вызове `g(x)` вы получите `c(1, 10)`:

```
g(x)
#>  x  y
#>  1 10
```

Кажется, здесь наблюдается какая-то непоследовательность: почему `x` берется из значения, определенного внутри `g()`, а `y` по-прежнему приходит из глобального окружения? Причина в том, что функция `UseMethod()` вызывает `g.default()` особым образом – чтобы переменные, определенные в обобщенной функции, были доступны методам. Исключение составляют аргументы, переданные в функцию: они передаются как есть и не затрагиваются кодом внутри обобщенной функции.

6 Оператор извлечения подмножеств `[` представляет собой примитивную и обобщенную функцию, что можно подтвердить с помощью функции `ftype()`.

```
ftype(`[`)
#> [1] "primitive" "generic"
```

Для примитивных функций вызов `formals([)` возвращает `NULL`, так что нам нужен другой способ для определения аргументов функции. Один из вариантов состоит в инспекции исходного кода на C, который можно увидеть с помощью функции `pryr::show_c_source(.Primitive("["))`.

При выводе аргументов некоторых методов [можно увидеть, что набор аргументов меняется в зависимости от класса x.

```
names(formals(`[.data.frame`))
#> [1] "x"    "i"    "j"    "drop"
names(formals(`[.table`))
#> [1] "x"    "i"    "j"    "..."  "drop"
names(formals(`[.Date`))
#> [1] "x"    "..."  "drop"
names(formals(`[.AsIs`))
#> [1] "x"   "i"   "..."
```

Для получения полного отчета нам нужно приложить немного усилий и снова воспользоваться функцией `s3_methods_generic()`.

```
library(dplyr)

s3_methods_generic("[") %>%
    filter(visible) %>%
    mutate(
        method = paste0("[.", class),
        argnames = purrr::map(method, ~ names(formals(.x))),
        args = purrr::map(method, ~ formals(.x)),
        args = purrr::map2(
            argnames, args,
            ~ paste(.x, .y, sep = " = ")
        ),
        args = purrr::set_names(args, method)
    ) %>%
    pull(args) %>%
    head()
#> $`[.AsIs`
#> [1] "x = "   "i = "   "... = "
#>
#> $`[.data.frame`
#> [1] "x = "
#> [2] "i = "
#> [3] "j = "
#> [4] "drop = if (missing(i)) TRUE else length(cols) == 1"
#>
#> $`[.Date`
#> [1] "x = "        "... = "      "drop = TRUE"
#>
#> $`[.difftime`
#> [1] "x = "        "... = "      "drop = TRUE"
#>
#> $`[.Dlist`
#> [1] "x = "   "i = "   "... = "
#>
#> $`[.DLLInfoList`
#> [1] "x = "   "... = "
```

13.5.1. Ответы на упражнения

1 Мы можем категоризировать возвращаемые объекты в зависимости от того, как рассчитывается количество наблюдений. Для классов векторного стиля количество наблюдений представлено в виде `length(x)`. В объектах в стиле записей используется список элементов одинаковой длины для представления отдельных компонентов. В датафреймах и матрицах наблюдения представлены в виде строк. В объектах в стиле скаляров используется список для представления одной сущности.

Это приводит к следующей классификации:

- векторный стиль: `factor()`, `table()`, `as.Date()`, `as.POSIXct()`, `ordered()`;
- стиль записей: нет;
- стиль датафреймов: нет;
- стиль скаляров: `lm()`, `ecdf()`.

Стиль `I()` зависит от входа, поскольку эта функция возвращает *«копию объекта с добавленным классом* `AsIs`*»*.

2 Конструктор должен заполнять значения атрибутов объекта `lm` и проверять их типы на корректность. Давайте начнем с создания простого объекта `lm` и исследования его базового типа и атрибутов:

```
mod <- lm(cyl ~ ., data = mtcars)

typeof(mod)
#> [1] "list"

attributes(mod)
#> $names
#>  [1] "coefficients"  "residuals"     "effects"       "rank"
#>  [5] "fitted.values" "assign"        "qr"            "df.residual"
#>  [9] "xlevels"       "call"          "terms"         "model"
#>
#> $class
#> [1] "lm"
```

Поскольку в основе объекта `mod` лежит список, мы можем воспользоваться вызовом функции `map(mod, typeof)` для определения базовых типов этих элементов. (Также можно изучить справку `?lm` для получения дополнительной информации об отдельных атрибутах.)

```
map_chr(mod, typeof)
#>  coefficients     residuals       effects          rank fitted.values
#>      "double"      "double"      "double"     "integer"      "double"
#>        assign            qr   df.residual       xlevels          call
#>     "integer"        "list"     "integer"        "list"    "language"
#>         terms         model
#>    "language"        "list"
```

Теперь у нас достаточно информации для создания конструктора для новых объектов `lm`.

```
new_lm <- function(
    coefficients, residuals, effects, rank, fitted.values, assign,
    qr, df.residual, xlevels, call, terms, model
)   {

    stopifnot(
        is.double(coefficients), is.double(residuals),
        is.double(effects), is.integer(rank), is.double(fitted.values),
        is.integer(assign), is.list(qr), is.integer(df.residual),
        is.list(xlevels), is.language(call), is.language(terms),
        is.list(model)
    )

    structure(
        list(
            coefficients = coefficients,
            residuals = residuals,
            effects = effects,
            rank = rank,
            fitted.values = fitted.values,
            assign = assign,
            qr = qr,
            df.residual = df.residual,
            xlevels = xlevels,
            call = call,
            terms = terms,
            model = model
        ),
        class = "lm"
    )
}
```

13.6.3. Ответы на упражнения

1 `[.Date` вызывает функцию `.Date`, передавая ей на вход результат вызова функции `[` применительно к родительскому классу, а также `oldClass()`:

```
`[.Date`
#> function (x, ..., drop = TRUE)
#> {
#>     .Date(NextMethod("["), oldClass(x))
#> }
#> <bytecode: 0x7fdf0dc3e600>
#> <environment: namespace:base>
```

`.Date` – это что-то вроде конструктора для классов дат, но без проверки корректности типов входных данных:

```
.Date
#> function (xx, cl = "Date")
#> `class<-`(xx, cl)
#> <bytecode: 0x7fdf0e215800>
#> <environment: namespace:base>
```

`oldClass()` – это по сути то же самое, что и `class()`, за исключением того, что он не возвращает неявные классы, т. е. фактически это `attr(x, "class")`. Если взглянуть на код на языке C, лежащий в его основе, можно увидеть, что именно это он и делает, а также поддерживает классы S4.

Поскольку `oldClass()` по своей сути является `class()`, мы можем переписать `[.Date`, чтобы сделать реализацию более понятной:

```
`[.Date` <- function(x, ..., drop = TRUE) {
    out <- NextMethod("[")
    class(out) <- class(x)
    out
}
```

Таким образом, `[.Date` гарантирует, что на выходе будет тот же тип, что и на входе. А как насчет других атрибутов, которыми мог бы обладать подкласс? Они теряются:

```
x <- structure(1:4, test = "test", class = c("myDate", "Date"))
attributes(x[1])
#> $class
#> [1] "myDate" "Date"
```

2 Для ответа на этот вопрос нам надо сначала получить соответствующие обобщенные функции.

```
generics_t  <- s3_methods_class("POSIXt")$generic
generics_ct <- s3_methods_class("POSIXct")$generic
generics_lt <- s3_methods_class("POSIXlt")$generic
```

Обобщенные функции в `generics_t` с методом для родительского класса `POSIXt` потенциально разделяют одно и то же поведение в обоих подклассах. Однако если обобщенная функция располагает конкретным методом, специфичным для одного из подклассов, нужно произвести логическое вычитание:

```
# Эти обобщенные функции содержат методы, специфичные для подклассов
union(generics_ct, generics_lt)
#>  [1] "["              "[["             "[<-"            "as.data.frame"
#>  [5] "as.Date"        "as.list"        "as.POSIXlt"     "c"
#>  [9] "format"         "length<-"       "mean"           "print"
#> [13] "rep"            "split"          "summary"        "Summary"
#> [17] "weighted.mean" "xtfrm"          "[[<-"           "anyNA"
#> [21] "as.double"      "as.matrix"      "as.POSIXct"     "duplicated"
#> [25] "is.na"          "length"         "names"          "names<-"
```

```
#> [29] "sort"          "unique"

# Эти обобщенные функции разделяют (унаследованные) методы для обоих подклассов
setdiff(generics_t, union(generics_ct, generics_lt))
#>  [1] "-"             "+"             "all.equal"    "as.character" "Axis"
#>  [6] "cut"           "diff"          "hist"         "is.numeric"   "julian"
#> [11] "Math"          "months"        "Ops"          "pretty"       "quantile"
#> [16] "quarters"      "round"         "seq"          "str"          "trunc"
#> [21] "weekdays"
```

3 При запуске этого кода происходит следующее:

- мы передаем объект из классов `b` и `a2` функции `generic2()`, что побуждает R искать метод `generic2.b()`;
- в методе `generic2.b()` класс меняется на `a1`, и происходит вызов функции `NextMethod()`;
- в этот момент можно было бы подумать, что произойдет вызов метода `generic2.a1()`, но на самом деле, как упоминалось в книге, функция `NextMethod()` *в действительности не взаимодействует с атрибутом* `class` *объекта, а использует особую глобальную переменную (*`.Class`*) для определения того, какой метод вызвать следующим*.

```
generic2(structure(list(), class = c("b", "a2")))
#> [1] "a2"
```

Давайте дважды проверим показанное выше выражение и вычислим переменную `.Class` внутри метода `generic2.b()` явно.

```
generic2.b <- function(x) {
    class(x) <- "a1"
    print(.Class)
    NextMethod()
}

generic2(structure(list(), class = c("b", "a2")))
#> [1] "b"  "a2"
#> [1] "a2"
```

13.7.5. Ответы на упражнения

1 Функция `class()` в обоих случаях возвращает `integer`. В то же время класс для `x1` был создан неявно, и он наследуется от класса `numeric`, тогда как класс для `x2` был установлен явно. Это очень важно, поскольку `length()` – это внутренняя обобщенная функция, а они диспетчеризуются на методы, только если установлен атрибут `class`, т. е. внутренние обобщенные функции не работают с неявными классами.

Определить, что для объекта явно не задан класс, можно по тому, что в этом случае выражение `attr(x, "class")` для него вернет `NULL`:

```
attr(x1, "class")
#> NULL
attr(x2, "class")
#> [1] "integer"
```

Для просмотра классов, которые будут использоваться при диспетчеризации методов, можно воспользоваться функцией `sloop::s3_class()`:

```
s3_class(x1)  # неявно
#> [1] "integer" "numeric"

s3_class(x2)  # явно
#> [1] "integer"
```

Для лучшего понимания вывода функции `s3_dipatch()` можно обратиться к справке `?s3_dispatch`:

- `=>` – метод существует, и он обнаружен функцией `UseMethod()`;
- `->` – метод существует, и он *используется* функцией `UseMethod()`;
- `*` – метод существует, но не используется;
- ничего (выделен серым в консоли) – метод не существует.

2 Этой группе принадлежат следующие функции (см. `?Math`):

- `abs`, `sign`, `sqrt`, `floor`, `ceiling`, `trunc`, `round`, `signif`;
- `exp`, `log`, `expm1`, `log1p`, `cos`, `sin`, `tan`, `cospi`, `sinpi`, `tanpi`, `acos`, `asin`, `atan`, `cosh`, `sinh`, `tanh`, `acosh`, `asinh`, `atanh`;
- `lgamma`, `gamma`, `digamma`, `trigamma`;
- `cumsum`, `cumprod`, `cummax`, `cummin`.

У следующих классов присутствует метод для этой групповой обобщенной функции:

```
s3_methods_generic("Math")
#> # A tibble: 8 x 4
#>   generic class      visible source
#>   <chr>   <chr>      <lgl>   <chr>
#> 1 Math    data.frame TRUE    base
#> 2 Math    Date       TRUE    base
#> 3 Math    difftime   TRUE    base
#> 4 Math    factor     TRUE    base
#> 5 Math    POSIXt     TRUE    base
#> 6 Math    quosure    FALSE   registered S3method
#> 7 Math    vctrs_sclr FALSE   registered S3method
#> 8 Math    vctrs_vctr FALSE   registered S3method
```

Для описания общей идеи давайте просто перепишем метод датафрейма:

```
Math.data.frame <- function(x) "hello"
```

Теперь все функции из группы Math будут возвращать "hello":

```
abs(mtcars)
#> [1] "hello"
exp(mtcars)
#> [1] "hello"
lgamma(mtcars)
#> [1] "hello"
```

Конечно, разные функции должны производить разные вычисления. Здесь в игру вступает .Generic, позволяющий получить вызываемую обобщенную функцию в виде строки:

```
Math.data.frame <- function(x, ...) {
    .Generic
}

abs(mtcars)
#> [1] "abs"
exp(mtcars)
#> [1] "exp"
lgamma(mtcars)
#> [1] "lgamma"

rm(Math.data.frame)
```

Исходный код Math.data.frame() – отличный пример того, как можно преобразовать строку, возвращаемую .Generic, в конкретный метод. Примером метода может служить Math.factor(), который объявлен только с целью улучшения вывода сообщений об ошибках.

3 Помимо прочего, Math.difftime() исключает все случаи, кроме функций abs, sign, floor, ceiling, trunc, round и signif, и должен возвращать подходящие сообщения об ошибках.

Для сравнения, реализация Math.difftime() в книге:

```
Math.difftime <- function(x, ...) {
    new_difftime(NextMethod(), units = attr(x, "units"))
}
rm(Math.difftime)
```

Реализация Math.difftime() в базовом R:

```
Math.difftime
#> function (x, ...)
#> {
#>     switch(.Generic, abs = , sign = , floor = , ceiling = , trunc = ,
#>         round = , signif = {
#>             units <- attr(x, "units")
#>             .difftime(NextMethod(), units)
```

```
#>         }, stop(gettextf("'%s' not defined for \"difftime\" objects",
#>             .Generic), domain = NA))
#> }
#> <bytecode: 0x7fdf0aa3ea78>
#> <environment: namespace:base>
```

Ответы на упражнения из главы 14

14.2.6. Ответы на упражнения

1 Начнем с реализации класса банковского счета, похожей на `Accumulator` из книги.

```
BankAccount <- R6Class(
    classname = "BankAccount",
    public = list(
        balance = 0,
        deposit = function(dep = 0) {
            self$balance <- self$balance + dep
            invisible(self)
        },
        withdraw = function(draw) {
            self$balance <- self$balance - draw
            invisible(self)
        }
    )
)
```

Для проверки этого класса создадим один объект банковского счета и уйдем по нему в минус:

```
my_account <- BankAccount$new()
my_account$balance
#> [1] 0

my_account$
    deposit(5)$
    withdraw(15)$
    balance
#> [1] -10
```

Теперь создадим первый подкласс, в котором реализуем контроль отрицательного баланса с соответствующим сообщением.

```
BankAccountStrict <- R6Class(
    classname = "BankAccountStrict",
    inherit = BankAccount,
```

```
    public = list(
        withdraw = function(draw = 0) {
            if (self$balance - draw < 0) {
                stop("Your `withdraw` must be smaller ",
                    "than your `balance`.",
                    call. = FALSE
                )
            }
            super$withdraw(draw = draw)
        }
    )
)
```

В результате будем получать ошибку, как и ожидали.

```
my_strict_account <- BankAccountStrict$new()
my_strict_account$balance
#> [1] 0

my_strict_account$
    deposit(5)$
    withdraw(15)
#> Error: Your `withdraw` must be smaller than your `balance`.

my_strict_account$balance
#> [1] 5
```

Наконец, создадим еще один подкласс, в реализации которого будет допускаться превышение кредитного лимита (отрицательный баланс), но при этом будет взиматься комиссия.

```
BankAccountCharging <- R6Class(
    classname = "BankAccountCharging",
    inherit = BankAccount,
    public = list(
        withdraw = function(draw = 0) {
            if (self$balance - draw < 0) {
                draw <- draw + 1
            }
            super$withdraw(draw = draw)
        }
    )
)
```

Протестируем функционал. Итоговый баланс будет равен –12, поскольку мы осуществили две операции с комиссией.

```
my_charging_account <- BankAccountCharging$new()
my_charging_account$balance
#> [1] 0
```

```
my_charging_account$
    deposit(5)$
    withdraw(15)$
    withdraw(0)

my_charging_account$balance
#> [1] -12
```

2 Наш новый класс `ShuffledDeck` будет использовать функцию `sample()` и метод извлечения подмножеств с помощью целых чисел при реализации перемешивания колоды и раздачи карт. Также мы добавим проверку на то, чтобы нельзя было взять из колоды больше карт, чем в ней присутствует.

```
ShuffledDeck <- R6Class(
    classname = "ShuffledDeck",
    public = list(
        deck = NULL,
        initialize = function(deck = cards) {
            self$deck <- sample(deck)
        },
        reshuffle = function() {
            self$deck <- sample(cards)
            invisible(self)
        },
        n = function() {
            length(self$deck)
        },
        draw = function(n = 1) {
            if (n > self$n()) {
                stop("Only ", self$n(), " cards remaining.", call. = FALSE)
            }

            output <- self$deck[seq_len(n)]
            self$deck <- self$deck[-seq_len(n)]
            output
        }
    )
)
```

Для проверки этого класса создадим колоду карт (инициализируем экземпляр класса), возьмем из нее все карты, после чего дважды перемешаем колоду и проверим, что каждый раз мы получаем разные карты.

```
my_deck <- ShuffledDeck$new()

my_deck$draw(52)
#>  [1] "6 SPADE"    "10 DIAMOND" "Q CLUB"     "J SPADE"    "Q HEARTS"
#>  [6] "8 DIAMOND"  "5 DIAMOND"  "4 CLUB"     "9 CLUB"     "9 SPADE"
#> [11] "5 SPADE"    "3 HEARTS"   "J CLUB"     "2 DIAMOND"  "K SPADE"
```

```
#> [16] "2 HEARTS"   "2 SPADE"    "8 SPADE"    "8 HEARTS"   "6 HEARTS"
#> [21] "7 HEARTS"   "6 CLUB"     "K CLUB"     "3 CLUB"     "10 SPADE"
#> [26] "3 DIAMOND"  "Q SPADE"    "9 HEARTS"   "J DIAMOND"  "7 DIAMOND"
#> [31] "9 DIAMOND"  "7 SPADE"    "4 DIAMOND"  "10 HEARTS"  "2 CLUB"
#> [36] "4 SPADE"    "4 HEARTS"   "8 CLUB"     "K HEARTS"   "A SPADE"
#> [41] "A HEARTS"   "5 HEARTS"   "A DIAMOND"  "5 CLUB"     "7 CLUB"
#> [46] "Q DIAMOND"  "A CLUB"     "10 CLUB"    "3 SPADE"    "K DIAMOND"
#> [51] "J HEARTS"   "6 DIAMOND"
my_deck$draw(10)
#> Error: Only 0 cards remaining.

my_deck$reshuffle()$draw(5)
#> [1] "6 DIAMOND" "2 CLUB"    "Q DIAMOND" "9 CLUB"    "J DIAMOND"
my_deck$reshuffle()$draw(5)
#> [1] "8 CLUB"   "9 SPADE"  "2 SPADE"  "Q HEARTS" "6 SPADE"
```

3 Потому что классы S3 подчиняются базовой семантике R в отношении копирования при изменении, в результате чего при каждом депозите на счет или извлечении карт из колоды будут создаваться новые копии объектов.

Можно комбинировать классы S3 с окружениями (как работает система R6), но не рекомендуется создавать объекты, которые выглядят как обычные объекты R, но при этом обладают ссылочной семантикой.

4 Для создания класса R6, позволяющего получать и устанавливать часовой пояс, добавим в него соответствующие публичные методы.

```
Timezone <- R6Class(
    classname = "Timezone",
    public = list(
        get = function() {
            Sys.timezone()
        },
        set = function(value) {
            stopifnot(value %in% OlsonNames())

            old <- self$get()
            Sys.setenv(TZ = value)
            invisible(old)
        }
    )
)
```

(При установке часового пояса мы невидимо возвращаем старое значение, поскольку это облегчает процесс его восстановления.)

Теперь создадим один экземпляр этого класса и проверим установку и получение часового пояса.

```
tz <- Timezone$new()

old <- tz$set("Antarctica/South_Pole")
```

```
tz$get()
#> [1] "Antarctica/South_Pole"

tz$set(old)
tz$get()
#> [1] "UTC"
```

5 Взгляните на следующую довольно минималистическую реализацию класса:

```
WorkingDirectory <- R6Class(
    classname = "WorkingDirectory",
    public = list(
        get = function() {
            getwd()
        },
        set = function(value) {
            setwd(value)
        }
    )
)
```

6 Поскольку классы S3 не подходят для моделирования состояний, изменяющихся с течением времени. Методы S3 должны (почти) всегда возвращать один и тот же результат для одинаковых входов.

7 Объекты R6 построены на базе окружений. Они располагают атрибутом `class`, представляющим собой символьный вектор, содержащий имя класса, имена родительских классов (если таковые существуют) и строку `"R6"` в качестве завершающего элемента.

14.3.3. Ответы на упражнения

1 Для удовлетворения этим требованиям мы сделаем баланс приватным полем. В результате пользователю придется воспользоваться методами `$deposit()` и `$withdraw()`, которые смогут взаимодействовать с балансом.

```
BankAccountStrict2 <- R6Class(
    classname = "BankAccountStrict2",
    public = list(
        deposit = function(dep = 0) {
            private$balance <- private$balance + dep
            invisible(self)
        },
        withdraw = function(draw = 0) {
            if (private$balance - draw < 0) {
                stop(
                      "Your `withdraw` must be smaller ",
                      "than your `balance`.",
                      call. = FALSE
```

```
                )
            }
            private$balance <- private$balance - draw
            invisible(self)
        }
    ),
    private = list(
        balance = 0
    )
)
```

Для проверки класса создадим его экземпляр и попробуем уйти в минус.

```
my_account_strict_2 <- BankAccountStrict2$new()

my_account_strict_2$deposit(5)
my_account_strict_2$withdraw(10)
#> Error: Your `withdraw` must be smaller than your `balance`.
```

2 Для защиты пароля от изменения и прямого доступа сделаем его приватным полем. Кроме того, мы реализуем собственный метод `print`, который не позволит получить информацию о действующем пароле.

```
Password <- R6Class(
    classname = "Password",
    public = list(
        print = function(...) {
            cat("<Password>: ********\n")
            invisible(self)
        },
        set = function(value) {
            private$password <- value
        },
        check = function(password) {
            identical(password, private$password)
        }
    ),
    private = list(
        password = NULL
    )
)
```

Давайте создадим экземпляр нашего нового класса и убедимся, что наш пароль нельзя напрямую ни изменить, ни прочитать, но проверить его правильность можно.

```
my_pw <- Password$new()
my_pw$set("snuffles")
my_pw$password
#> NULL
```

```
my_pw
#> <Password>: ********
my_pw$check("snuggles")
#> [1] FALSE
my_pw$check("snuffles")
#> [1] TRUE
```

3 Для доступа к предыдущему сгенерированному случайному значению из экземпляра нам необходимо добавить в класс приватное поле `$last_random`, и мы изменим метод `$random()` таким образом, чтобы он осуществлял запись в это поле при каждом вызове. Для доступа к полю `$last_random` снабдим класс методом `$previous()`.

```
Rando <- R6::R6Class(
    classname = "Rando",
    private = list(
        last_random = NULL
    ),
    active = list(
        random = function(value) {
            if (missing(value)) {
                private$last_random <- runif(1)
                private$last_random
            } else {
                stop("Can't set `$random`.", call. = FALSE)
            }
        },
        previous = function(value) {
            if (missing(value)) {
                private$last_random
            }
        }
    )
)
```

Создадим экземпляр класса и проверим его работоспособность.

```
x <- Rando$new()
x$random
#> [1] 0.349
x$random
#> [1] 0.947
x$previous
#> [1] 0.947
```

4 Для проверки того, могут ли подклассы обращаться к приватным полям/методам родительского класса, мы сперва создадим класс `A` с приватным полем `foo` и приватным методом `bar()`. После этого в созданном экземпляре дочернего класса `B` попытаемся обратиться к родительскому полю и методу.

```
A <- R6Class(
    classname = "A",
    private = list(
        field = "foo",
        method = function() {
            "bar"
        }
    )
)

B <- R6Class(
    classname = "B",
    inherit = A,
    public = list(
        test = function() {
            cat("Field:  ", super$field, "\n", sep = "")
            cat("Method: ", super$method(), "\n", sep = "")
        }
    )
)

B$new()$test()
#> Field:
#> Method: bar
```

В результате мы выяснили, что подклассы могут обращаться к приватным методам родительского класса, но не имеют доступа к его приватным полям.

14.4.4. Ответы на упражнения

1 Наш класс `FileWriter` будет открывать подключение к файлу при инициализации. Обратите внимание, что мы передаем в функцию `file()` аргумент `open = "a"`, что позволяет открыть подключение в режиме добавления текста. В противном случае функция `cat()` могла бы работать только применительно к файлам, а не к подключениям, присутствующим в условии задачи. Также мы создадим метод `append_line()` для добавления данных в файл и реализуем закрытие подключения к файлу в методе `$finalize()`.

```
FileWriter <- R6::R6Class(
    classname = "FileWriter",
    public = list(
        con = NULL,
        initialize = function(filename) {
            self$con <- file(filename, open = "a")
        },

        finalize = function() {
            close(self$con)
        },

        append_line = function(x) {
```

```
            cat(x, "\n", sep = "", file = self$con)
        }
    )
)
```

Создадим экземпляр класса и проверим его работоспособность.

```
tmp_file <- tempfile()
my_fw <- FileWriter$new(tmp_file)

readLines(tmp_file)
#> character(0)
my_fw$append_line("First")
my_fw$append_line("Second")
readLines(tmp_file)
#> [1] "First"  "Second"
```

Ответы на упражнения из главы 15

15.2.1. Ответы на упражнения

1 Объекты класса S4 `Period` располагают шестью слотами с именами `year`, `month`, `day`, `hour`, `minute` и `.Data` (в котором содержится количество секунд). Все слоты обладают типом с двойной точностью. К большинству полей можно обращаться посредством аналогично названных функций доступа (например, `lubridate::year()` вернет значение соответствующего поля), а для получения значения из слота `.Data` необходимо воспользоваться методом `second()`.

В качестве простого примера создадим интервал из 1 секунды, 2 минут, 3 часов, 4 дней и 5 недель.

```
example_12345 <- lubridate::period(
    c(1, 2, 3, 4, 5),
    c("second", "minute", "hour", "day", "week")
)
```

В итоге должен получиться интервал, соответствующий 39 дням, 3 часам, 2 минутам и 1 секунде.

```
example_12345
#> [1] "39d 3H 2M 1S"
```

При анализе структуры объекта `example_12345` мы видим, что количество секунд хранится в поле `.Data`.

```
str(example_12345)
#> Formal class 'Period' [package "lubridate"] with 6 slots
#>   ..@ .Data : num 1
#>   ..@ year  : num 0
```

```
#>   ..@ month : num 0
#>   ..@ day   : num 39
#>   ..@ hour  : num 3
#>   ..@ minute: num 2
```

2 Помимо добавления вопросительного знака (?) непосредственно перед вызовом функции (например, `?method()`), можно также обратиться к:

- общей документации по обобщенной функции: `?genericName`;
- общей документации по методам обобщенной функции: `methods?genericName`;
- документации по конкретному методу: `ClassName?methodName`.

15.3.6. Ответы на упражнения

1 Класс `Person`, описанный в книге, содержит слоты `name` и `age`. В классе из пакета `utils` присутствуют слоты `given` (вектор имен), `family`, `role`, `email` и `comment` (см. `?utils::person`). Все слоты класса `utils::person()`, кроме `role`, должны быть символьного типа с единичной длиной. Содержимое слота `role` должно соответствовать одной из следующих аббревиатур: `"aut"`, `"com"`, `"cph"`, `"cre"`, `"ctb"`, `"ctr"`, `"dtc"`, `"fnd"`, `"rev"`, `"ths"`, `"trl"`. Таким образом, слот `role` может отличаться по длине от остальных слотов, и мы учтем это при написании валидатора.

```
# Определение класса Person
setClass("Person",
    slots = c(
        age = "numeric",
        given = "character",
        family = "character",
        role = "character",
        email = "character",
        comment = "character"
    ),
    prototype = list(
        age = NA_real_,
        given = NA_character_,
        family = NA_character_,
        role = NA_character_,
        email = NA_character_,
        comment = NA_character_
    )
)

# Функция-помощник для создания экземпляра класса Person
Person <- function(given, family,
                   age = NA_real_,
                   role = NA_character_,
                   email = NA_character_,
```

```
                    comment = NA_character_) {
    age <- as.double(age)

    new("Person",
        age = age,
        given = given,
        family = family,
        role = role,
        email = email,
        comment = comment
    )
}

# Валидатор, проверяющий длины слотов
setValidity("Person", function(object) {
    invalids <- c()
    if (length(object@age)     != 1 ||
        length(object@given)   != 1 ||
        length(object@family)  != 1 ||
        length(object@email)   != 1 ||
        length(object@comment) != 1) {
        invalids <- paste0("@name, @age, @given, @family, @email, ",
                           "@comment must be of length 1")
    }

    known_roles <- c(
        NA_character_, "aut", "com", "cph", "cre", "ctb",
        "ctr", "dtc", "fnd", "rev", "ths", "trl"
    )

    if (!all(object@role %in% known_roles)) {
        paste(
            "@role(s) must be one of",
            paste(known_roles, collapse = ", ")
        )
    }

    if (length(invalids)) return(invalids)
    TRUE
})
#> Class "Person" [in ".GlobalEnv"]
#>
#> Slots:
#>
#> Name:        age     given    family      role     email   comment
#> Class:   numeric character character character character character
```

2 Если мы наследуемся от другого класса, то получим такие же слоты. Интереснее ситуация возникает тогда, когда мы не наследуемся ни от одного из существующих классов. В результате мы получим виртуальный класс, экземпляр которого создать нельзя.

```
setClass("Human")
new("Human")
#> Error in new("Human"): trying to generate an object from a virtual class ("Human")
```

В то же время наследоваться от виртуального класса можно.

```
setClass("Programmer", contains = "Human")
```

3 Для этих классов нам понадобится один слот для данных и по одному слоту для каждого атрибута. Обратите внимание, что наследование имеет значение для упорядоченных факторов и дат. Для датафреймов внутри валидатора необходимо выполнить проверку на равенство длин элементов списков.

Для простоты мы не вводим явный подкласс для упорядоченных факторов. Вместо этого мы добавили `ordered` в виде слота.

```
setClass("Factor",
    slots = c(
        data = "integer",
        levels = "character",
        ordered = "logical"
    ),
    prototype = list(
        data = integer(),
        levels = character(),
        ordered = FALSE
    )
)

new("Factor", data = c(1L, 2L), levels = letters[1:3])
#> An object of class "Factor"
#> Slot "data":
#> [1] 1 2
#>
#> Slot "levels":
#> [1] "a" "b" "c"
#>
#> Slot "ordered":
#> [1] FALSE
```

В классе `Date2` даты хранятся в виде целых чисел, подобно тому, как это происходит в базовом R с числами двойной точности. Других атрибутов у дат нет.

```
setClass("Date2",
    slots = list(
        data = "integer"
    ),
    prototype = list(
        data = integer()
    )
)
```

```
new("Date2", data = 1L)
#> An object of class "Date2"
#> Slot "data":
#> [1] 1
```

В нашем классе `DataFrame` присутствует список и слот `row.names`. Большую часть логики (например, проверку на то, что все элементы списка являются векторами и обладают одинаковой длиной) можно было бы реализовать в валидаторе.

```
setClass("DataFrame",
    slots = c(
        data = "list",
        row.names = "character"
    ),
    prototype = list(
        data = list(),
        row.names = character(0)
    )
)

new("DataFrame", data = list(a = 1, b = 2))
#> An object of class "DataFrame"
#> Slot "data":
#> $a
#> [1] 1
#>
#> $b
#> [1] 2
#>
#>
#> Slot "row.names":
#> character(0)
```

15.4.5. Ответы на упражнения

1 Мы реализуем функции доступа с помощью обобщенной функции `age()`, метода класса `Person` и замещающей функции `age<-`:

```
setGeneric("age", function(x) standardGeneric("age"))
#> [1] "age"
setMethod("age", "Person", function(x) x@age)

setGeneric("age<-", function(x, value) standardGeneric("age<-"))
#> [1] "age<-"
setMethod("age<-", "Person", function(x, value) {
    x@age <- value
    validObject(x)
    x
})
```

2 В рамках функции `setGeneric()` первый аргумент, `name`, необходим для установки имени обобщенной функции. Затем это имя также в явном виде присутствует в диспетчере методов `standardGeneric()` (в параметре `def` функции `setGeneric()`). Это поведение похоже на использование функции `UseMethod()` в системе S3.

3 Функция `is(object)` возвращает класс указанного объекта. Также для классов, подобных `Employee`, вывод функции `is(object)` включает в себя родительский класс. Чтобы всегда показывать нужный нам класс (характерный для подкласса), метод `show()` возвращает первый элемент `is(object)`.

4 Зависит от ситуации. Давайте создадим объект `hadley` класса `Person`:

```
.Person <- setClass(
    "Person",
    slots = c(name = "character", age = "numeric")
)

hadley <- .Person(name = "Hadley")
hadley
#> An object of class "Person"
#> Slot "name":
#> [1] "Hadley"
#>
#> Slot "age":
#> numeric(0)
```

Теперь посмотрим, какие аргументы могут быть переданы в обобщенную функцию `show()`.

```
formals("show")
#> $object
```

Обычно мы используем этот аргумент при определении нового метода.

```
setMethod("show", "Person", function(object) {
    cat(object@name, "creates hard exercises")
})

hadley
#> Hadley creates hard exercises
```

При передаче другого имени в качестве первого элемента нашего метода (например, `x` вместо `object`) этот элемент будет сопоставлен с корректным аргументом `object`, и мы получим предупреждение. При этом метод продолжит работать:

```
setMethod("show", "Person", function(x) {
    cat(x@name, "creates hard exercises")
})
```

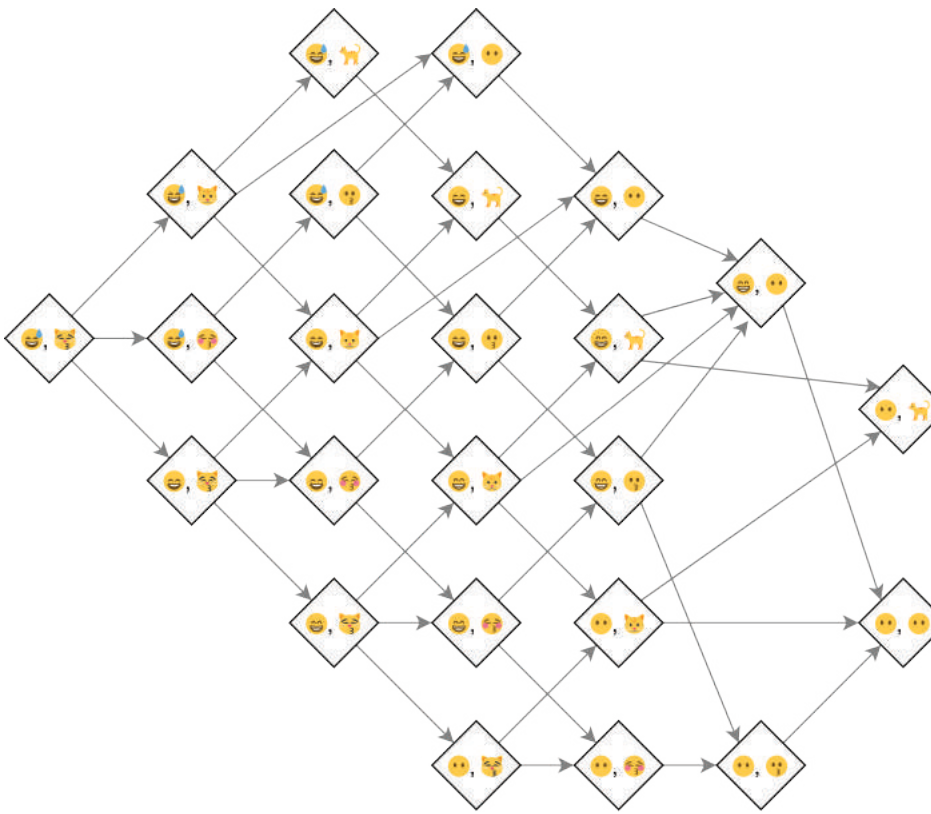
```
#> Warning: For function 'show', signature 'Person': argument in method definition
#> changed from (x) to (object)

hadley
#> Hadley creates hard exercises
```

Если добавить в наш метод больше аргументов, чем может обработать обобщенная функция, возникнет ошибка.

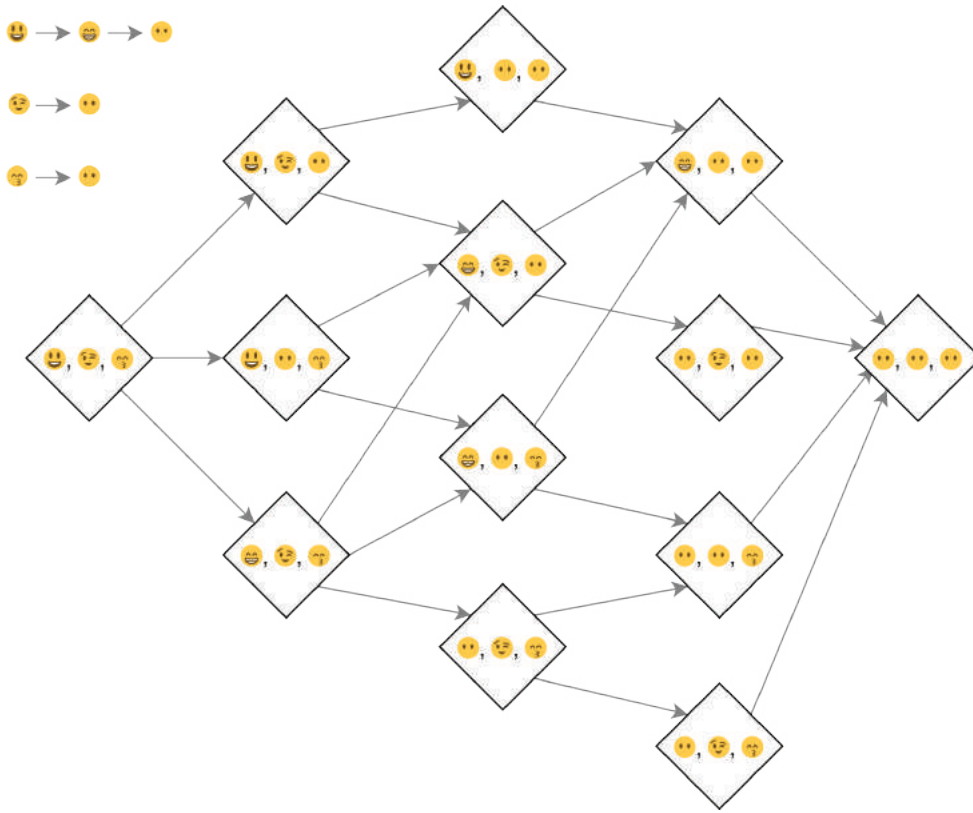
```
setMethod("show", "Person", function(x, y) {
  cat(x@name, "is", x@age, "years old")
})
#> Error in conformMethod(signature, mnames, fnames, f, fdef, definition): in method for
'show' with signature 'object="Person"': formal arguments (object = "Person") omitted in
the method definition cannot be in the signature
```

15.5.5. Ответы на упражнения

1 Взгляните на граф и повторяйте за мной: «Я никогда не буду без нужды усложнять структуру своих классов и использовать множественное наследование».



2 Как видите, граф для этого примера получился более простым по сравнению с первым. Говоря в целом, множественная диспетчеризация методов приводит к меньшим усложнениям по сравнению с множественным наследованием. Но и ее стоит применять умеренно!



3 Возникнет ситуация неопределенности по причине того, что расстояние от одного класса до всех концевых узлов будет составлять 2, а от остальных четырех классов до двух концевых узлов – по 1. Чтобы разрешить эту ситуацию, нужно определить еще пять методов, по одному на каждую комбинацию классов.

15.6.3. Ответы на упражнения

1 Целью вызова функции `setOldClass()` является регистрация класса S3 в виде «формально определенного класса», чтобы он мог использоваться в системе S4. При ее вызове мы можем передать аргумент `S4Class`, что приведет к наследованию всех слотов и их значений по умолчанию (`prototype`).

Давайте построим класс S4 `OrderedFactor` на основе класса S3 `factor` следующим образом.

```
setOldClass("factor")    # для краткости используем встроенное определение

OrderedFactor <- setClass(
    "OrderedFactor",
    contains = "factor",   # наследуем от зарегистрированного класса S3
    slots = c(
        levels = "character",
        ordered = "logical"  # добавляем логической слот
    ),
    prototype = structure(
        integer(),
        levels = character(),
        ordered = logical()  # добавляем значение по умолчанию
    )
)
```

Теперь можно зарегистрировать класс S3 `ordered`, предоставив шаблон класса S4. Мы также можем использовать класс S4 для создания нового объекта напрямую.

```
setOldClass("ordered", S4Class = "OrderedFactor")

x <- OrderedFactor(
    c(1L, 2L, 2L),
    levels = c("a", "b", "c"),
    ordered = TRUE
)
str(x)
#> Formal class 'OrderedFactor' [package ".GlobalEnv"] with 4 slots
#>   ..@ .Data   : int [1:3] 1 2 2
#>   ..@ levels  : chr [1:3] "a" "b" "c"
#>   ..@ ordered : logi TRUE
#>   ..@ .S3Class: chr "factor"
```

2 Для простоты мы будем просто возвращать строку `"180cm"` при вызове метода `length()` класса `Person`. Метод может быть определен как в S3 или в S4:

```
length.Person <- function(x) "180cm"  # S3
setMethod("length", "Person", function(x) "180cm")  # S4
```

Ответы на упражнения из главы 18

18.2.4. Ответы на упражнения

1 Да пребудут с вами фрагменты приведенного выше кода, и да покажут они вам, как были созданы сии абстрактные синтаксические деревья.

```
ast(f(g(h())))
#> ▉─f
#> └─▉─g
#>   └─▉─h

ast(1 + 2 + 3)
#> ▉─`+`
#> ├─▉─`+`
#> │ ├─1
#> │ └─2
#> └─3

ast((x + y) * z)
#> ▉─`*`
#> ├─▉─`(`
#> │ └─▉─`+`
#> │   ├─x
#> │   └─y
#> └─z
```

2 Давайте поручим отрисовку пакету `lobstr`.

```
ast(f(g(h(i(1, 2, 3)))))
#> ▉─f
#> └─▉─g
#>   └─▉─h
#>     └─▉─i
#>       ├─1
#>       ├─2
#>       └─3

ast(f(1, g(2, h(3, i()))))
#> ▉─f
#> ├─1
#> └─▉─g
#>   ├─2
#>   └─▉─h
#>     ├─3
#>     └─▉─i

ast(f(g(1, 2), h(3, i(4, 5))))
#> ▉─f
#> ├─▉─g
#> │ ├─1
#> │ └─2
#> └─▉─h
#>   ├─3
#>   └─▉─i
#>     ├─4
#>     └─5
```

3 Абстрактные синтаксические деревья начинаются с имен функций. Именно поэтому первое выражение было преобразовано в префиксную форму. Во втором случае парсер R преобразовал оператор `**` в функцию `^`, а в последнем выражение перевернулось при парсинге:

```
str(expr(x ** y))
#>  language x^y
str(expr(a -> b))
#>  language b <- a
```

4 Последний концевой элемент AST не указан в выражении явно. Вместо этого базовый R автоматически создал атрибут `srcref`, указывающий на исходный код функции.

5 AST для вложенных `else if` может выглядеть не слишком понятно из-за множества фигурных скобок. При этом мы видим, что в месте появления `else` просто вычисляется очередная ветвь `if`.

```
ast(
  if (FALSE) {
    1
  } else if (FALSE) {
    2
  } else if (TRUE) {
    3
  }
)
#> ▉─`if`
#> ├─FALSE
#> ├─▉─`{`
#> │ └─1
#> └─▉─`if`
#>   ├─FALSE
#>   ├─▉─`{`
#>   │ └─2
#>   └─▉─`if`
#>     ├─TRUE
#>     └─▉─`{`
#>       └─3
```

Структуру дерева можно сделать более понятной, если убрать фигурные скобки:

```
ast(
  if (FALSE) 1
  else if (FALSE) 2
  else if (TRUE) 3
)
#> ▉─`if`
```

```
#> ├─FALSE
#> ├─1
#> └─`if`
#>   ├─FALSE
#>   ├─2
#>   └─`if`
#>     ├─TRUE
#>     └─3
```

18.3.5. Ответы на упражнения

1 Невозможно создать атомарные векторы комплексных чисел и байтовых последовательностей без использования вызовов функций (это можно сделать только для мнимых скаляров, таких как `i` или `5i`). В свою очередь, выражения, содержащие функции, называются объектами вызова. Именно поэтому такие типы векторов не могут присутствовать в выражениях.

Также невозможно создать выражение, которое при вычислении дало бы атомарный вектор длиной больше единицы без использования функций (например, `c()`).

Давайте покажем это на примере:

```
# Атомарный вектор
is_atomic(expr(1))
#> [1] TRUE

# Неатомарный вектор (хотя при вычислении дал бы атомарный)
is_atomic(expr(c(1, 1)))
#> [1] FALSE
is_call(expr(c(1, 1)))
#> [1] TRUE
```

2 После удаления первого элемента из объекта вызова второй элемент, представляющий собой вызываемую функцию, переместится на первую позицию. В результате мы получим `"foo.csv"(header = TRUE)`.

3 Представленные объекты вызова отличаются первыми двумя элементами, которые в отдельных случаях вычисляются еще до создания вызова. В первом примере обе составляющие, `median()` и `x`, вычисляются и встраиваются в вызов. В результате в созданном вызове мы видим, что `median` – это обобщенная функция, а аргумент `x` – это `1:10`.

```
call2(median, x, na.rm = TRUE)
#> (function (x, na.rm = FALSE, ...)
#> UseMethod("median"))(1:10, na.rm = TRUE)
```

В следующих двух вызовах мы используем другие комбинации: в одном случае вычисляется только `x`, в другом – только `median()`.

```
call2(expr(median), x, na.rm = TRUE)
#> median(1:10, na.rm = TRUE)
call2(median, expr(x), na.rm = TRUE)
#> (function (x, na.rm = FALSE, ...)
#> UseMethod("median"))(x, na.rm = TRUE)
```

В последнем примере не вычисляются ни `x`, ни `median()`.

```
call2(expr(median), expr(x), na.rm = TRUE)
#> median(x, na.rm = TRUE)
```

Обратите внимание, что все эти вызовы при вычислении генерируют одинаковый результат. Ключевым отличием является момент нахождения значения, связанного с символами `x` и `median`.

4 Причина такого неожиданного поведения заключается в том, что функция `mean()` использует аргумент …, что не позволяет стандартизировать соответствующие аргументы. Поскольку в функции `mean()` применяется диспетчеризация методов S3 (т. е. вызывается функция `UseMethod()`), а в методе `mean.default()` присутствуют и другие аргументы, функция `call_standardise()` лучше справится с обработкой конкретного метода S3.

```
call_standardise(quote(mean.default(1:10, na.rm = TRUE)))
#> mean.default(x = 1:10, na.rm = TRUE)
call_standardise(quote(mean.default(n = T, 1:10)))
#> mean.default(x = 1:10, na.rm = T)
call_standardise(quote(mean.default(x = 1:10, , TRUE)))
#> mean.default(x = 1:10, na.rm = TRUE)
```

5 Как было написано в этой книге, *первым элементом в объекте вызова всегда является вызываемая функция*.

Давайте посмотрим, что произойдет при запуске кода.

```
x <- expr(foo(x = 1))
x
#> foo(x = 1)

names(x) <- c("x", "")
x
#> foo(1)

names(x) <- c("", "x")
x
#> foo(x = 1)
```

Таким образом, присвоение первому элементу имени просто добавляет метаданные, которые в R игнорируются.

6 Так же, как и в случае с префиксной версией, мы получим:

```
call2("if", call2(">", sym("x"), 1), "a", "b")
#> if (x > 1) "a" else "b"
```

При чтении AST слева направо мы получаем ту же структуру следующего вида: функция для вычисления, выражение, также представляющее собой функцию, которая выполнится первой, и две константы, которые будут вычислены следом.

```
ast(`if`(x > 1, "a", "b"))
#> ■─`if`
#> ├─■─`>`
#> │ ├─x
#> │ └─1
#> ├─"a"
#> └─"b"
```

18.4.4. Ответы на упражнения

1 Фокус здесь состоит в том, что в R скобка (() может быть как составной частью префиксной функции, так и вызовом функции (.

Таким образом, в дереве для первого примера мы не увидим внешней открывающей скобки, поскольку она входит в префиксную запись и принадлежит `f()`. Напротив, внутренняя скобка представляет собой функцию (что отражено в виде символа в AST):

```
ast(f((1)))
#> ■─f
#> └─■─`(`
#>   └─1
```

Во втором примере мы видим, что уже внешняя скобка – это функция, а внутренняя входит в ее синтаксис:

```
ast(`(`(1 + 1))
#> ■─`(`
#> └─■─`+`
#>   ├─1
#>   └─1
```

Для ясности давайте также создадим третий пример, в котором ни одна из скобок не будет входить в синтаксис другой функции:

```
ast(((1 + 1)))
#> ■─`(`
#> └─■─`(`
#>   └─■─`+`
```

```
#>     ├─1
#>     └─1
```

2 Оператор = может использоваться и как оператор присваивания, и для обозначения именованных аргументов в определениях функций:

```
b = c(c = 1)
```

Таким образом, простое включение в функцию `ast()` здесь не сработает:

```
ast(b = c(c = 1))
#> Error in ast(b = c(c = 1)): unused argument (b = c(c = 1))
```

Мы получили ошибку, потому что, встретив запись `b =`, R приступил к поиску аргумента с именем `b`. Но поскольку `x` – это единственный аргумент функции `ast()`, выполнение дальше не продвинулось.

Простейший способ решения этой проблемы состоит в заключении передаваемого в функцию `ast()` аргумента в фигурные скобки ({}).

```
ast({b = c(c = 1)})
#> ▉─`{`
#> └─▉─`=`
#>   ├─b
#>   └─▉─c
#>     └─c = 1
```

Если проигнорировать скобки и сравнить деревья, мы увидим, что первое вхождение = используется для присваивания, а второе представляет часть синтаксиса вызова функции.

3 Результат будет –4, поскольку оператор ^ обладает более высоким приоритетом по сравнению с -, что можно увидеть на AST (или взглянув в справку `?"Syntax"`):

```
-2^2
#> [1] -4

ast(-2^2)
#> ▉─`-`
#> └─▉─`^`
#>   ├─2
#>   └─2
```

4 Ответ может вас несколько удивить:

```
!1 + !1
#> [1] FALSE
```

Чтобы разобраться, что здесь происходит, взглянем на AST:

```
ast(!1 + !1)
#> ▉─`!`
#> └▉─`+`
#>   ├─1
#>   └▉─`!`
#>     └─1
```

Правое вхождение `!1` вычисляется первым и дает `FALSE`, поскольку при использовании логического оператора R преобразует любое ненулевое значение в `TRUE`, а отрицание `TRUE` дает `FALSE`.

Затем вычисляется выражение `1 + FALSE`, которое дает 1 вследствие приведения `FALSE` к нулю.

Наконец, `!1` дает при вычислении `FALSE`.

Обратите внимание, что если бы оператор `!` обладал более высоким приоритетом, промежуточное выражение выглядело бы так: `FALSE + FALSE`, что при вычислении дало бы 0.

5 Первая причина состоит в том, что оператор `<-` является правоассоциативным, т. е. вычисление с его участием выполняется справа налево:

```
x1 <- (x2 <- (x3 <- 0))
```

Вторая причина в том, что оператор `<-` невидимо возвращает значение операнда, стоящего справа.

```
(x3 <- 0)
#> [1] 0
```

6 Давайте взглянем на синтаксические деревья:

```
ast(x + y %+% z)
#> ▉─`+`
#> ├─x
#> └▉─`%+%`
#>   ├─y
#>   └─z
```

Здесь сначала вычисляется выражение `y %+% z`, после чего полученный результат прибавляется к `x`.

```
ast(x ^ y %+% z)
#> ▉─`%+%`
#> ├▉─`^`
#> │ ├─x
#> │ └─y
#> └─z
```

В данном случае сначала вычисляется выражение `x ^ y`, а результат используется в качестве первого аргумента функции `%+%()`.

Из этого можно заключить, что по приоритету пользовательские инфиксные функции располагаются между операторами сложения и возведения в степень.

Полную информацию о приоритете инфиксных функций можно получить в справке `?"Syntax"`, из которой видно, что такие функции следуют сразу за оператором последовательности (`:`) и перед операторами умножения и деления (`*` и `/`).

7 В этом случае функция `parse_expr()` определит необходимость обработки нескольких выражений и выдаст ошибку.

```
parse_expr("x + 1; y + 1")
#> Error: More than one expression parsed
```

8 Попытка парсинга некорректных выражений приведет к возникновению ошибки в функции `parse()`.

```
parse_expr("a +")
#> Error in parse(text = elt): <text>:2:0: unexpected end of input
#> 1: a +
#>    ^
parse_expr("f())")
#> Error in parse(text = elt): <text>:1:4: unexpected ')'
#> 1: f())
#>        ^

parse(text = "a +")
#> Error in parse(text = "a +"): <text>:2:0: unexpected end of input
#> 1: a +
#>    ^
parse(text = "f())")
#> Error in parse(text = "f())"): <text>:1:4: unexpected ')'
#> 1: f())
#>        ^
```

9 Функция `expr_text()` объединит вместе результаты функции `deparse(expr)` с использованием в качестве разделителя символа переноса строки (`\n`).

```
expr <- expr(g(a + b + c + d + e + f + g + h + i + j + k + l + m +
                 n + o + p + q + r + s + t + u + v + w + x + y + z))
deparse(expr)
#> [1] "g(a + b + c + d + e + f + g + h + i + j + k + l + m + n + "
#> [2] "o + p + q + r + s + t + u + v + w + x + y + z)"
expr_text(expr)
#> [1] "g(a + b + c + d + e + f + g + h + i + j + k + l + m + n
#> + \n    o + p + q + r + s + t + u + v + w + x + y + z)"
```

10 Функция `pairwise.t.test()` захватывает аргументы с данными (`x` и `g`), чтобы была возможность вывести входные выражения вместе с рассчитанными *p*-значениями. Вплоть до версии R 4.0.0 это делалось с помощью функций `deparse(substitute(x))` и `paste()`. Это могло приводить к неожиданным результатам в случае превышения одним из аргументов порогового значения на количество символов `width.cutoff` в функции `deparse()`, по умолчанию равного 60. В этом случае выражение могло быть преобразовано в символьный вектор длины больше 1.

```
# Вывод R версии 3.6.2
d <- 1
pairwise.t.test(2, d + d + d + d + d + d + d + d +
                  d + d + d + d + d + d + d + d + d)
#>  Pairwise comparisons using t tests with pooled SD
#>
#> data:  2 and d + d + d + d + d + d + d + d + d + d + d + d + d + d + d + d
#> + d + d +  2 and     d
#>
#> <0 x 0 matrix>
#>
#> P value adjustment method: holm
```

В R версии 4.0.0 функция `pairwise.t.test()` получила новую реализацию, в которой используется функция `deparse1()`, выступающая в роли обертки функции `deparse()`.

> *Функция* `deparse1()`*, добавленная в R версии 4.0.0 для обеспечения получения строковых результатов (в виде символьного вектора единичной длины), обычно используется при построении имен, как в* `deparse1(substitute(.))`.

```
# Вывод R версии 4.0.0
d <- 1
pairwise.t.test(2, d + d + d + d + d + d + d + d +
                  d + d + d + d + d + d + d + d + d)
#>  Pairwise comparisons using t tests with pooled SD
#>
#> data:  2 and d + d + d + d + d + d + d + d + d + d + d + d + d + d + d + d
#> + d + d + d
#>
#> <0 x 0 matrix>
#>
#> P value adjustment method: holm
```

18.5.3. Ответы на упражнения

1 Здесь можно применить ту же логику, что и в примере с присваиванием. Мы будем воспринимать это как частный случай и обрабатывать его с по-

мощью `find_T_call()`, которая при нахождении вызовов `T()` будет их отбрасывать. Мы также повторим функцию `expr_type()`, служащую для определения того, с каким случаем – базовым или рекурсивным – мы имеем дело.

```
expr_type <- function(x) {
    if (rlang::is_syntactic_literal(x)) {
        "constant"
    } else if (is.symbol(x)) {
        "symbol"
    } else if (is.call(x)) {
        "call"
    } else if (is.pairlist(x)) {
        "pairlist"
    } else {
        typeof(x)
    }
}

switch_expr <- function(x, ...) {
    switch(expr_type(x),
           ...,
           stop("Don't know how to handle type ",
                typeof(x), call. = FALSE))
}

find_T_call <- function(x) {
    if (is_call(x, "T")) {
        x <- as.list(x)[-1]
        purrr::some(x, logical_abbr_rec)
    } else {
        purrr::some(x, logical_abbr_rec)
    }
}

logical_abbr_rec <- function(x) {
    switch_expr(
        x,
        # Базовый случай
        constant = FALSE,
        symbol = as_string(x) %in% c("F", "T"),

        # Рекурсивные случаи
        pairlist = purrr::some(x, logical_abbr_rec),
        call = find_T_call(x)
    )
}

logical_abbr <- function(x) {
    logical_abbr_rec(enexpr(x))
}
```

Теперь давайте проверим нашу новую функцию `logical_abbr()`:

```
logical_abbr(T(1, 2, 3))
#> [1] FALSE
logical_abbr(T(T, T(3, 4)))
#> [1] TRUE
logical_abbr(T(T))
#> [1] TRUE
logical_abbr(T())
#> [1] FALSE
logical_abbr()
#> [1] FALSE
logical_abbr(c(T, T, T))
#> [1] TRUE
```

2 Функции в данный момент не поддерживаются по причине того, что в функции `switch_expr()` не обрабатывается значение `"closure"`.

```
logical_abbr(!!f)
#> Error: Don't know how to handle type closure
```

Если мы хотим, чтобы это работало, нужно написать функцию, которая будет выполнять итерации также по аргументам и телу входной функции.

3 Давайте сначала взглянем на AST предложенного присваивания:

```
ast(names(x) <- x)
#> ▐─`<-`
#> ├─▐─names
#> │ └─x
#> └─x
```

Таким образом, нам нужно перехватывать ситуации, когда первые два элемента являются вызовами. Первый вызов будет идентичен оператору `<-`, а значит, нам необходимо вернуть только второй вызов, чтобы узнать, каким объектам были присвоены новые значения.

С этой целью мы добавим следующий блок кода в инструкцию `else` функции `find_assign_call()`:

```
if (is_call(x, "<-") && is_call(x[[2]])) {
    lhs <- expr_text(x[[2]])
    children <- as.list(x)[-1]
}
```

Соберем все вместе и проверим работоспособность нашей новой функции:

```
flat_map_chr <- function(.x, .f, ...) {
    purrr::flatten_chr(purrr::map(.x, .f, ...))
}
```

```
find_assign <- function(x) unique(find_assign_rec(enexpr(x)))

find_assign_call <- function(x) {
    if (is_call(x, "<-") && is_symbol(x[[2]])) {
        lhs <- as_string(x[[2]])
        children <- as.list(x)[-1]
    } else {
        if (is_call(x, "<-") && is_call(x[[2]])) {
            lhs <- expr_text(x[[2]])
            children <- as.list(x)[-1]
        } else {
            lhs <- character()
            children <- as.list(x)
        }}

    c(lhs, flat_map_chr(children, find_assign_rec))
}

find_assign_rec <- function(x) {
    switch_expr(
        x,
        # Базовый случай
        constant = ,symbol = character(),
        # Рекурсивные случаи
        pairlist = flat_map_chr(x, find_assign_rec),
        call = find_assign_call(x)
    )
}

# Проверка функционала
find_assign(x <- y)
#> [1] "x"
find_assign(names(x))
#> character(0)
find_assign(names(x) <- y)
#> [1] "names(x)"
find_assign(names(x(y)) <- y)
#> [1] "names(x(y))"
find_assign(names(x(y)) <- y <- z)
#> [1] "names(x(y))" "y"
```

4 Здесь нам понадобится удалить ранее добавленную проверку из `else` и добавить проверку на вызов (не обязательно `<-`) в первом условии `if` в функции `find_assign_call()`. При нахождении вызова мы сохраняем его и возвращаем позже в составе нашего символьного вывода. Весь остальной код сохранится без изменений:

```
find_assign_call <- function(x) {
    if (is_call(x)) {
        lhs <- expr_text(x)
```

```
        children <- as.list(x)[-1]
    } else {
        lhs <- character()
        children <- as.list(x)
    }

    c(lhs, flat_map_chr(children, find_assign_rec))
}

find_assign_rec <- function(x) {
    switch_expr(
        x,
        # Базовый случай
        constant = ,
        symbol = character(),

        # Рекурсивные случаи
        pairlist = flat_map_chr(x, find_assign_rec),
        call = find_assign_call(x)
    )
}

find_assign(x <- y)
#> [1] "x <- y"
find_assign(names(x(y)) <- y <- z)
#> [1] "names(x(y)) <- y <- z" "names(x(y))"           "x(y)"
#> [4] "y <- z"
find_assign(mean(sum(1:3)))
#> [1] "mean(sum(1:3))" "sum(1:3)"       "1:3"
```

Ответы на упражнения из главы 19

19.2.2. Ответы на упражнения

1 Сперва для каждого аргумента мы последуем совету из этой главы и исполним их за пределами соответствующих функций. Поскольку `MASS`, `cyl`, `vs` и `am` не представляют собой объекты в глобальном окружении, попытка их выполнения приведет к возникновению ошибки `"Object not found"`. Таким образом мы можем подтвердить, что эти аргументы являются цитируемыми. Для других аргументов мы можем ознакомиться с исходным кодом (и документацией), чтобы определить, применяются ли к ним какие-то цитирующие механизмы или они просто вычисляются.

```
library(MASS)  # MASS -> цитируемый
```

Функция `library()` также принимает на вход символьные векторы и не выполняет цитирование, если аргумент `character.only` установлен в `TRUE`, так что вызов `library(MASS, character.only = TRUE)` вернет ошибку.

```
mtcars2 <- subset(mtcars, cyl == 4)  # mtcars -> вычисляемый
# cyl -> цитируемый

with(mtcars2, sum(vs))  # mtcars2 -> вычисляемый
# sum(vs) -> цитируемый

sum(mtcars2$am)  # matcars$am -> вычисляемый
# am -> цитируемый в $()
```

При инспекции исходного кода функции `rm()` мы обнаруживаем, что она захватывает свой аргумент … как невычисляемый вызов (в данном случае список пар) посредством функции `match.call()`. Впоследствии этот вызов преобразуется в строку для дальнейшего вычисления.

```
rm(mtcars2)  # mtcars2 -> цитируемый
```

2 Из предыдущего упражнения мы уже знаем, что функция `library()` выполняет цитирование своего первого аргумента.

```
library(dplyr)    # dplyr   -> цитируемый
library(ggplot2)  # ggplot2 -> цитируемый
```

Также становится ясно, что аргумент `cyl` цитируется в функции `group_by()`.

```
by_cyl <- mtcars %>%           # mtcars -> вычисляемый
    group_by(cyl) %>%            # cyl -> цитируемый
    summarise(mean = mean(mpg))  # mean = mean(mpg) -> цитируемый
```

Чтобы разобраться, что происходит внутри функции `summarise()`, необходимо изучить ее исходный код. Дойдя до фрагмента с диспетчеризацией методов S3 в функции `summarise()`, мы обнаруживаем, что аргумент … цитируется в функции `dplyr:::summarise_cols()`, которая вызывается в методе `summarise.data.frame()`.

```
dplyr::summarise
#> function (.data, ..., .groups = NULL)
#> {
#>     UseMethod("summarise")
#> }
#> <bytecode: 0x7fdd17d26010>
#> <environment: namespace:dplyr>

dplyr:::summarise.data.frame
#> function (.data, ..., .groups = NULL)
#> {
#>     cols <- summarise_cols(.data, ...)
#>     out <- summarise_build(.data, cols)
#>     if (identical(.groups, "rowwise")) {
#>         out <- rowwise_df(out, character())
```

```
#>     }
#>     out
#> }
#> <bytecode: 0x7fdd1844e908>
#> <environment: namespace:dplyr>

dplyr:::summarise_cols
#> function (.data, ...)
#> {
#>     mask <- DataMask$new(.data, caller_env())
#>     dots <- enquos(...)
#>     dots_names <- names(dots)
#>     auto_named_dots <- names(enquos(..., .named = TRUE))
#>     cols <- list()
#>     sizes <- 1L
#>     chunks <- vector("list", length(dots))
#>     types <- vector("list", length(dots))
#>
#>     ## function definition abbreviated for clarity ##
#> }
#> <bytecode: 0x55b540c07ca0>
#> <environment: namespace:dplyr>
```

В следующем выражении ggplot2 объекты cyl и mean цитируются.

```
ggplot(by_cyl,            # by_cyl -> вычисляется
       aes(cyl, mean)) +  # aes() -> вычисляется
  # cyl, mean -> цитируются (в aes)
  geom_point()
```

Подтверждение этому можно найти в исходном коде функции aes().

```
ggplot2::aes
#> function (x, y, ...)
#> {
#>     exprs <- enquos(x = x, y = y, ..., .ignore_empty = "all")
#>     aes <- new_aes(exprs, env = parent.frame())
#>     rename_aes(aes)
#> }
#> <bytecode: 0x7fdd18833fc8>
#> <environment: namespace:ggplot2>
```

19.3.6. Ответы на упражнения

1 Функция expr() выступает в качестве простой обертки, передающей свой аргумент в функцию enexpr().

```
expr
#> function (expr)
#> {
```

```
#>     enexpr(expr)
#> }
#> <bytecode: 0x7fdd189e9c80>
#> <environment: namespace:rlang>
```

2 Обе функции способны захватывать несколько аргументов и возвращают именованный список выражений. При этом функция `f1()` возвращает аргументы, определенные в теле функции `f1()`. Это происходит из-за того, что функция `exprs()` захватывает выражения, как они определены разработчиком при объявлении функции `f1()`.

```
f1(a + b, c + d)
#> $x
#> x
#>
#> $y
#> y
```

В то же время функция `f2()` возвращает аргументы, переданные в функцию `f2()`, как они были заданы пользователем при вызове функции.

```
f2(a + b, c + d)
#> $x
#> a + b
#>
#> $y
#> c + d
```

3 В первом случае будет выброшена ошибка:

```
on_expr <- function(x) {enexpr(expr(x))}
on_expr(x + y)
#> Error: `arg` must be a symbol
```

Во втором случае вернется пропущенный аргумент:

```
on_missing <- function(x) {enexpr(x)}
on_missing()
is_missing(on_missing())
#> [1] TRUE
```

4 В вызове `exprs(a)` входной параметр `a` интерпретируется как символ для неименованного аргумента. В результате мы получаем неименованный список с первым элементом, содержащим символ `a`.

```
out1 <- exprs(a)
str(out1)
#> List of 1
#>  $ : symbol a
```

В вызове exprs(a =) первый аргумент a является именованным, но без значения. В результате мы видим на выходе именованный список с первым элементом с именем a, содержащим пропущенный аргумент.

```
out2 <- exprs(a = )
str(out2)
#> List of 1
#>  $ a: symbol
is_missing(out2$a)
#> [1] TRUE
```

5 Функция exprs() предлагает дополнительные аргументы .named (= FALSE), .ignore_empty (c("trailing", "none", "all")) и .unquote_names (TRUE). Аргумент .named позволяет гарантировать, что все аргументы будут именованы. Аргумент .ignore_empty служит для обработки пустых аргументов для многоточия (значение "trailing") или для всех аргументов ("none" и "all"). Кроме того, посредством аргумента .unquote_names можно указать, будет ли оператор := восприниматься как =. Оператор := может быть очень полезен, поскольку он поддерживает операцию расцитирования (!!) в левой части.

6 Давайте создадим новое окружение my_env, в котором не будет содержаться объектов. В этом случае функция substitute() просто вернет свой первый аргумент (expr):

```
my_env <- env()
substitute(x, my_env)
#> x
```

При создании функции, содержащей аргумент, который возвращается сразу после применения функции substitute(), на выходе мы получим переданное выражение:

```
foo <- function(x) substitute(x)

foo(x + y * sin(0))
#> x + y * sin(0)
```

Если функция substitute() обнаружит выражения (его части) в окружении env, будет выполнена подстановка (если env не является .GlobalEnv).

```
my_env$x <- 7
substitute(x, my_env)
#> [1] 7

x <- 7
substitute(x, .GlobalEnv)
#> x
```

19.4.8. Ответы на упражнения

1 Объединим и расцитируем данные цитированные выражения, чтобы получить желаемые вызовы:

```
expr(!!xy / !!yz)                    # (1)
#> (x + y)/(y + z)

expr(-(!!xz)^(!!yz))                 # (2)
#> -(x + z)^(y + z)

expr(((!!xy)) + !!yz-!!xy)           # (3)
#> (x + y) + (y + z) - (x + y)

expr(atan2(!!xy, !!yz))              # (4)
#> atan2(x + y, y + z)

expr(sum(!!xy, !!xy, !!yz))          # (5)
#> sum(x + y, x + y, y + z)

expr(sum(!!!abc))                    # (6)
#> sum(a, b, c)

expr(mean(c(!!!abc), na.rm = TRUE))  # (7)
#> mean(c(a, b, c), na.rm = TRUE)

expr(foo(a = !!xy, b = !!yz))        # (8)
#> foo(a = x + y, b = y + z)
```

2 Отличия легко обнаружить при помощи функции `lobstr::ast()`:

```
lobstr::ast(mean(1:10))
#> ▉─mean
#> └▉─`:`
#>   ├─1
#>   └─10
lobstr::ast(mean(!!(1:10)))
#> ▉─mean
#> └─<inline integer>
```

В выражении `mean(!!(1:10))` вызов `1:10` вычисляется в целочисленный вектор, тогда как в `mean(1:10)` мы по-прежнему имеем дело с объектом вызова.

Первый вариант (`mean(1:10)`) выглядит более естественным. В нем происходит захват отложенного вычисления, которое будет запущено при вызове функции. Во втором варианте (`mean(!!(1:10))`) вектор напрямую встраивается в вызов.

19.6.5. Ответы на упражнения

1 Функция `exec()` принимает на вход функцию (`f`), ее аргументы (`...`) и окружение (`.env`). Это позволяет собрать вызов из `f` и `...` и запустить его в переданном окружении. Поскольку аргумент `...` обрабатывается функцией `list2()`, функция `exec()` поддерживает точки tidy (квазицитирование), а это означает, что аргументы и имена (справа от оператора `:=`) могут быть расцитированы с помощью операторов `!!` и `!!!`.

2 Все три функции перехватывают точки с помощью вызова `args <- list(...)`. Функция `interaction()` рассчитывает взаимодействие переданных на вход факторов путем прохода итерациями по `args`. При передаче списка это определяется с помощью инструкции `length(args) == 1 && is.list(args[[1]])`, и выбирается один уровень из списка (`args <- args[[1]]`). В остальном код функции не отличается в отношении обработки точек и списков.

```
# Оба вызова приводят к одному и тому же результату
interaction(a = c("a", "b", "c", "d"), b = c("e", "f"))   # точки
#> [1] a.e b.f c.e d.f
#> Levels: a.e b.e c.e d.e a.f b.f c.f d.f
interaction(list(a = c("a", "b", "c", "d"), b = c("e", "f")))  # список
#> [1] a.e b.f c.e d.f
#> Levels: a.e b.e c.e d.e a.f b.f c.f d.f
```

В функции `expand.grid()` используется та же стратегия, и в случае со списком извлечение происходит так же: (`args <- args[[1]]`), если `length(args) == 1 && is.list(args[[1]])`.

В функции `par()` выполняется много предварительной работы, чтобы гарантировать допустимую структуру аргумента `args`. В отсутствие точек (`!length(args)`) создается список аргументов из внутреннего символьного вектора (частично в зависимости от аргумента `no.readonly`). Далее, если все элементы `args` являются символьными векторами (`all(unlist(lapply(args, is.character)))`), `args` преобразовывается в список с помощью вызова `as.list(unlist(args))` (это позволяет выровнять вложенные списки). Так же, как и в остальных функциях, выделение одного уровня `args` выполняется с помощью инструкции `args <- args[[1L]]`, когда `args` представляет собой вектор единичной длины, и его первый элемент – это список.

3 Функция `set_attr()` ожидает объект с именем `x` и его атрибуты в виде `...`. К сожалению, это не позволяет нам передать атрибут с именем `x`, поскольку в этом случае возникнет конфликт с именем аргумента нашего объекта. Не поможет даже пропуск имени аргумента объекта – это можно увидеть в примере, где объект начинает восприниматься как неименованный атрибут.

Но мы можем назвать первый аргумент `.x`, что значительно снизит вероятность возникновения подобных конфликтов. В этом случае вектор `1:10` получит (именованный) атрибут `x = 10`:

```
set_attr <- function(.x, ...) {
    attr <- rlang::list2(...)

    attributes(.x) <- attr
    .x
}

set_attr(1:10, x = 10)
#>  [1]  1  2  3  4  5  6  7  8  9 10
#> attr(,"x")
#> [1] 10
```

19.7.5. Ответы на упражнения

1 Мы бы предпочли первый вариант как наиболее удобный для чтения. На первый взгляд кажется, что здесь много шаблонного кода, но синтаксис расцитирования довольно легко читать. И в целом все выражение кажется более понятным и менее сложным.

2 Здесь функция `new_function()` позволяет нам создать фабрику функций с использованием *tidy evaluation*.

```
bc2 <- function(lambda) {
    lambda <- enexpr(lambda)

    if (!!lambda == 0) {
        new_function(exprs(x = ), expr(log(x)))
    } else {
        new_function(exprs(x = ), expr((x ^ (!!lambda) - 1) / !!lambda))
    }
}

bc2(0)
#> function (x)
#> log(x)
#> <environment: 0x7fdd18273740>
bc2(2)
#> function (x)
#> (x^2 - 1)/2
#> <environment: 0x7fdd18302e48>
bc2(2)(2)
#> [1] 1.5
```

3 Реализация может быть довольно простой и прямолинейной, хотя внешне кажется, что со скобками здесь перебор.

```
compose2 <- function(f, g) {
    f <- enexpr(f)
    g <- enexpr(g)
```

```
    new_function(exprs(... = ), expr((!!f)((!!g)(...))))
}

compose(sin, cos)
#> function(...) f(g(...))
#> <environment: 0x7fdd142a53b0>
compose(sin, cos)(pi)
#> [1] -0.841

compose2(sin, cos)
#> function (...)
#> sin(cos(...))
#> <environment: 0x7fdd1785c668>
compose2(sin, cos)(pi)
#> [1] -0.841
```

Ответы на упражнения из главы 20

20.2.4. Ответы на упражнения

1 По умолчанию функция `source()` использует глобальное окружение (`local = FALSE`). Вы можете выбрать окружение выполнения по своему усмотрению, передав его в аргументе `local`. Для использования текущего окружения (т. е. вызывающего окружения для функции `source()`) передайте аргумент `local = TRUE`.

```
# Создаем временный подгружаемый скрипт на R, который печатает x
tmp_file <- tempfile()
writeLines("print(x)", tmp_file)

# Устанавливаем `x` глобально
x <- "global environment"
env2 <- env(x = "specified environment")

locate_evaluation <- function(file, local) {
    x <- "local environment"
    source(file, local = local)
}

# Где функция source() будет вычислять этот код?
locate_evaluation(tmp_file, local = FALSE)  # по умолчанию
#> [1] "global environment"
locate_evaluation(tmp_file, local = env2)
#> [1] "specified environment"
locate_evaluation(tmp_file, local = TRUE)
#> [1] "local environment"
```

2 Давайте вспомним цитату из первого оригинального издания этой книги:

> *Функции* `expr()` *и* `eval()` *противоположны. [...] каждое применение функции* `eval()` *снимает один слой применения функции* `expr()`.

В общем виде выражение `eval(expr(x))` вычисляется в `x`. Таким образом, пример (1) даст на выходе 2 + 2 = 4. Добавление еще одного вызова функции `eval()` здесь не играет никакой роли, так что пример (2) также даст 4. Но если обернуть (1) в функцию `expr()`, все выражение будет процитировано.

```
eval(expr(eval(expr(eval(expr(2 + 2))))))        # (1)
#> [1] 4
eval(eval(expr(eval(expr(eval(expr(2 + 2)))))))  # (2)
#> [1] 4
expr(eval(expr(eval(expr(eval(expr(2 + 2)))))))  # (3)
#> eval(expr(eval(expr(eval(expr(2 + 2))))))
```

3 Реализуем эти две функции с помощью *tidy evaluation*. Преобразуем строку `name` в символ, после чего вычислим его:

```
get2 <- function(name, env = caller_env()) {
    name_sym <- sym(name)
    eval(name_sym, env)
}

x <- 1
get2("x")
#> [1] 1
```

С целью создания корректного выражения для присваивания значения воспользуемся оператором расцитирования `!!`.

```
assign2 <- function(name, value, env = caller_env()) {
    name_sym <- sym(name)
    assign_expr <- expr(!!name_sym <- !!value)
    eval(assign_expr, env)
}

assign2("x", 4)
x
#> [1] 4
```

4 Код функции `source2()` был представлен ранее следующим образом:

```
source2 <- function(path, env = caller_env()) {
    file <- paste(readLines(path, warn = FALSE), collapse = "\n")
    exprs <- parse_exprs(file)
```

```
    res <- NULL
    for (i in seq_along(exprs)) {
        res <- eval(exprs[[i]], env)
    }

    invisible(res)
}
```

Чтобы подчеркнуть отличия в нашей новой функции `source2()`, сохраним изменившийся блок кода в комментарии.

```
source2 <- function(path, env = caller_env()) {
    file <- paste(readLines(path, warn = FALSE), collapse = "\n")
    exprs <- parse_exprs(file)

    # res <- NULL
    # for (i in seq_along(exprs)) {
    #   res[[i]] <- eval(exprs[[i]], env)
    # }

    res <- purrr::map(exprs, eval, env)

    invisible(res)
}
```

Теперь проверим нашу новую функцию `source2()`. Помните, что оператор `<-` невидимо возвращает результат.

```
tmp_file <- tempfile()
writeLines(
  "x <- 1
       x
       y <- 2
       y  # some comment",
  tmp_file
)

(source2(tmp_file))
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] 1
#>
#> [[3]]
#> [1] 2
#>
#> [[4]]
#> [1] 2
```

5 Давайте последуем совету и добавим в функцию `local3()` вызов `print(call)`:

```
local3 <- function(expr, envir = new.env()) {
    call <- substitute(eval(quote(expr), envir))
    print(call)
    eval(call, envir = parent.frame())
}
```

В первой строке кода генерируется вызов функции `eval()`, поскольку `substitute()` вычисляется в текущем окружении выполнения. Но в данном случае это не имеет значения – и `expr`, и `envir` являются промисами, а в справке `?substitute` написано, что *слот с выражением из промиса заменяет символ*.

```
local3({
    x <- 10
    x * 2
})
#> eval(quote({
#>     x <- 10
#>     x * 2
#> }), new.env())
#> [1] 20
```

Затем переменная `call` вычисляется в вызывающем окружении (также известном как родительский фрейм (`parent.frame()`)). Почему это важно в контексте того, что `call` содержит еще один вызов `eval()`? Ответ не так очевиден: внешнее окружение определяет, где должен производиться поиск привязок для `eval`, `quote` и `new.env`.

```
eval(quote({
    x <- 10
    x * 2
}), new.env())
#> [1] 20
exists("x")
#> [1] TRUE
```

20.3.6. Ответы на упражнения

1 Структура данных *quosure* вычисляется в своем собственном окружении, так что имя `x` будет каждый раз иметь разную привязку. В результате мы получим следующий вывод:

```
eval_tidy(q1)
#> [1] 1
eval_tidy(q2)
#> [1] 11
eval_tidy(q3)
#> [1] 111
```

2 Структура *quosure* захватывает и выражение, и окружение. Из *quosure* мы можем получить доступ к окружению с помощью функции `get_env()`.

```
enenv <- function(x) {
    get_env(enquo(x))
}

# Проверка
enenv(x)
#> <environment: R_GlobalEnv>

# Проверка на то, работает ли это внутри функции
capture_env <- function(x) {
  enenv(x)
}
capture_env(x)
#> <environment: 0x7fce0322b580>
```

20.4.6. Ответы на упражнения

1 Ранее мы определили функцию `transform2()` следующим образом:

```
transform2 <- function(.data, ...) {
    dots <- enquos(...)

    for (i in seq_along(dots)) {
        name <- names(dots)[[i]]
        dot <- dots[[i]]

        .data[[name]] <- eval_tidy(dot, .data)
    }

    .data
}
```

Цикл `for` позволяет выполнять обработку `.data` итеративно. Действия включают в себя обновление `.data` и повторное использование тех же имен переменных. Это позволяет применять преобразования последовательно, чтобы в каждом следующем действии мы могли обращаться к только что созданным колонкам.

2 Давайте сначала внимательно посмотрим на функцию `subset2()`:

```
subset2 <- function(data, rows) {
    rows <- enquo(rows)
    rows_val <- eval_tidy(rows, data)
    stopifnot(is.logical(rows_val))

    data[rows_val, , drop = FALSE]
}
```

В этой функции представлена дополнительная логическая проверка, которой нет в функции `subset3()`. Здесь `rows` вычисляется в контексте `data`, что дает логический вектор. После этого достаточно воспользоваться оператором `[` для извлечения подмножества.

```
# выполнение subset2()
(rows_val <- eval_tidy(quo(x == 1), df))
#> [1]  TRUE FALSE FALSE
df[rows_val, , drop = FALSE]
#>   x
#> 1 1
```

В случае с функцией `subset3()` оба эти действия выполняются в одной строке (что, вероятно, более похоже на то, как мы сделали бы это вручную). В результате извлечение подмножества также выполняется в контексте маски данных.

```
# выполнение subset3()
eval_tidy(expr(df[x == 1, , drop = FALSE]), df)
#>   x
#> 1 1
```

Эта версия функции более короткая (но при этом более трудная для чтения), поскольку вычисление и извлечение подмножества выполняются в одном выражении. В то же время она может быть подвержена неожиданным ошибкам, если в маске данных окажется элемент с именем `data`, поскольку объекты из маски данных обладают более высоким приоритетом по сравнению с аргументами функции.

```
df <- data.frame(x = 1:3, data = 1)
subset2(df, x == 1)
#>   x data
#> 1 1    1
subset3(df, x == 1)
#> Error in data[~x == 1, , drop = FALSE]: incorrect number of dimensions
```

3 Функция `arrange2()` упорядочивает столбцы датафрейма по одной или нескольким переменным. Поскольку эта функция позволяет передавать переменные в виде выражений (посредством аргумента `...`), их необходимо сначала цитировать. После этого они используются в вызове функции `order()`, который происходит в контексте датафрейма. Наконец, датафрейм упорядочивается посредством целочисленного извлечения подмножества. Давайте внимательно пройдемся по исходному коду:

```
arrange2 <- function(.df, ..., .na.last = TRUE) {
    # Захватываем и цитируем аргументы, определяющие порядок столбцов
    args <- enquos(...)
```

```
    # `!!!`: расцитируем несколько аргументов (!!!) в order()
    # `!!.na.last`: передаем аргумент контроля NA в order()
    # возвращаем выражение
    order_call <- expr(order(!!!args, na.last = !!.na.last))

    # Вычисляем order_call в рамках .df
    ord <- eval_tidy(order_call, .df)
    # Убеждаемся, что ничего не потеряли
    stopifnot(length(ord) == nrow(.df))

    # Реорганизуем колонки с помощью целочисленного извлечения подмножества
    .df[ord, , drop = FALSE]
}
```

Мы применили оператор расцитирования (`!!`) к аргументу `.na.last` при построении вызова функции `order()`. Так мы получаем корректное указание аргумента `na.last` (обычно `TRUE`, `FALSE` или `NA`).

Без применения расцитирования выражение получилось бы таким: `na.last = .na.last`, и значение `.na.last` все равно нужно было бы искать. Поскольку эти вычисления выполняются в рамках окружения выполнения функции (которое содержит `.na.last`), это вряд ли приведет к каким-то проблемам.

```
# Последствия расцитирования .na.last
.na.last <- FALSE
expr(order(..., na.last = !!.na.last))
#> order(..., na.last = FALSE)
expr(order(..., na.last = .na.last))
#> order(..., na.last = .na.last)
```

20.5.4. Ответы на упражнения

1 Давайте сравним этот подход с базовой реализацией функции:

```
threshold_var <- function(df, var, val) {
    var <- as_string(ensym(var))
    subset2(df, .data[[var]] >= !!val)
}
```

Как видите, в нашей новой реализации не выполняется приведение символа к строке. Это позволило использовать оператор `$` вместо `[[` для извлечения подмножества. Изначально мы полагали, что в операторе `$` будет использовано частичное соответствие, но с `.data` такая проблема отсутствует.

Префиксный вызов функции `$()` встречается реже по сравнению с инфиксным извлечением подмножества с помощью оператора `[[`, но в конечном счете оба подхода работают одинаково.

```
df <- data.frame(x = 1:10)
threshold_var(df, x, 8)
#>     x
```

```
#> 8   8
#> 9   9
#> 10 10
threshold_var2(df, x, 8)
#>     x
#> 8   8
#> 9   9
#> 10 10
```

20.6.3. Ответы на упражнения

1 В этой функции `lm_call` вычисляется в вызывающем окружении, которое представлено глобальным окружением. В данном окружении имя `data` привязано к `utils::data`. Чтобы исправить эту ошибку, можно либо указать окружение выполнения функции, либо расцитировать аргумент `data` при построении вызова функции `lm()`.

```
# Меняем окружение выполнения
lm3b <- function(formula, data) {
    formula <- enexpr(formula)

    lm_call <- expr(lm(!!formula, data = data))
    eval(lm_call, current_env())
}

lm3b(mpg ~ disp, mtcars)$call
#> lm(formula = mpg ~ disp, data = data)
lm3b(mpg ~ disp, data)$call  #воспроизводим исходную ошибку
#> Error in model.frame.default(formula = mpg ~ disp, data = data, drop.unused.levels =
TRUE): 'data' must be a data.frame, environment, or list
```

Если мы хотим расцитировать аргумент внутри функции, сначала необходимо захватить пользовательские данные (с помощью функции `enexpr()`).

```
# Расцитирование аргумента
lm3c <- function(formula, data) {
    formula <- enexpr(formula)
    data_quo <- enexpr(data)

    lm_call <- expr(lm(!!formula, data = !!data_quo))
    eval(lm_call, caller_env())
}
lm3c(mpg ~ disp, mtcars)$call
#> lm(formula = mpg ~ disp, data = mtcars)
```

2 В нашей обертке `lm_wrap()` мы для отклика и данных зададим значения по умолчанию в виде `mpg` и `mtcars`. Это даст неплохой баланс удобства функции и ее гибкости.

```
lm_wrap <- function(pred, resp = mpg, data = mtcars,
                    env = caller_env()) {
    pred <- enexpr(pred)
    resp <- enexpr(resp)
    data <- enexpr(data)

    formula <- expr(!!resp ~ !!pred)
    lm_call <- expr(lm(!!formula, data = !!data))
    eval(lm_call, envir = env)
}

# Проверяем вывод
lm_wrap(I(1 / disp) + disp * cyl)
#>
#> Call:
#> lm(formula = mpg ~ I(1/disp) + disp * cyl, data = mtcars)
#>
#> Coefficients:
#> (Intercept)    I(1/disp)         disp          cyl     disp:cyl
#>   -1.22e+00     1.85e+03     7.68e-02     1.18e+00    -9.14e-03

# Сравниваем результаты с вызовом функции lm() напрямую
identical(
  lm_wrap(I(1 / disp) + disp * cyl),
  lm(mpg ~ I(1 / disp) + disp * cyl, data = mtcars)
)
#> [1] TRUE
```

3 В книге было представлено множество версий функции `resample_lm()`, но ни в одной из них не была реализована повторная выборка в аргументе функции.

В данном подходе используется преимущество отложенных вычислений аргументов функции путем переноса шага с повторной выборкой в определение аргумента. Пользователь передает данные в функцию, но использоваться будет их измененная версия (`resample_data`).

```
resample_lm <- function(
    formula, data,
    resample_data = data[sample(nrow(data), replace = TRUE), ,
                         drop = FALSE],
    env = current_env()) {

    formula <- enexpr(formula)

    lm_call <- expr(lm(!!formula, data = resample_data))
    expr_print(lm_call)
    eval(lm_call, env)
}

df <- data.frame(x = 1:10, y = 5 + 3 * (1:10) + round(rnorm(10), 2))
```

```
(lm_1 <- resample_lm(y ~ x, data = df))
#> lm(y ~ x, data = resample_data)
#>
#> Call:
#> lm(formula = y ~ x, data = resample_data)
#>
#> Coefficients:
#> (Intercept)            x
#>        4.85         3.02
lm_1$call
#> lm(formula = y ~ x, data = resample_data)
```

При использовании такого подхода необходимо, чтобы вычисления производились в рамках окружения функции, поскольку набор данных с повторными выборками (объявленный как значение по умолчанию) будет доступен только в нем.

В целом в R не очень принято выполнять значимую обработку данных вне тела функции. В сравнении с версией функции с использованием расцитирования (`resample_lm1()`) в этом подходе производится более осмысленный захват вызова модели. К тому же при каждом обновлении (`update()`) модели будет производиться новая повторная выборка.

Ответы на упражнения из главы 21

21.2.6. Ответы на упражнения

1 Итак, нам нужно реализовать особый случай процедуры экранирования для тега `<script>`. Сначала мы повторим функции, приведенные в этой главе, и убедимся, что для тегов вроде `<p>` и `<b>` экранирование работает корректно, а для тега `<script>` – нет. После этого мы внесем изменения в функции `escape()` и `tag()` и проверим работу новых функций.

Обратите внимание, что тег `<style>`, содержащий информацию о стилях CSS, следует тем же правилам экранирования, что и тег `<script>`. Так что мы попутно реализуем нормальное экранирование и для этого тега.

Начнем с просмотра кода, который у нас есть на данный момент.

```
# Экранирование
html <- function(x) structure(x, class = "advr_html")

print.advr_html <- function(x, ...) {
    out <- paste0("<HTML> ", x)
    cat(paste(strwrap(out), collapse = "\n"), "\n", sep = "")
}

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

```
escape.character <- function(x) {
    x <- gsub("&", "&amp;", x)
    x <- gsub("<", "&lt;", x)
    x <- gsub(">", "&gt;", x)

    html(x)
}

escape.advr_html <- function(x) x

# Базовые функции тегов
dots_partition <- function(...) {
    dots <- list2(...)

    if (is.null(names(dots))) {
        is_named <- rep(FALSE, length(dots))
    } else {
        is_named <- names(dots) != ""
    }

    list(
        named = dots[is_named],
        unnamed = dots[!is_named]
    )
}

# Функция html_attributes() из репозитория GitHub Advanced R
# https://github.com/hadley/adv-r/blob/master/dsl-html-attributes.r

html_attributes <- function(list) {
    if (length(list) == 0) return("")

    attr <- map2_chr(names(list), list, html_attribute)
    paste0(" ", unlist(attr), collapse = "")
}

html_attribute <- function(name, value = NULL) {
    if (length(value) == 0) return(name) # для атрибутов без значений
    if (length(value) != 1) stop("`value` must be NULL or length 1")
    if (is.logical(value)) {
        # Преобразовываем T и F в true и false
        value <- tolower(value)
    } else {
        value <- escape_attr(value)
    }
    paste0(name, "='", value, "'")
}

escape_attr <- function(x) {
    x <- escape.character(x)
    x <- gsub("\'", '&#39;', x)
```

```
    x <- gsub("\"", '&quot;', x)
    x <- gsub("\r", '&#13;', x)
    x <- gsub("\n", '&#10;', x)
    x
}

# Функции тегов
tag <- function(tag) {
    new_function(
        exprs(... = ),
        expr({
            dots <- dots_partition(...)
            attribs <- html_attributes(dots$named)
            children <- map_chr(dots$unnamed, escape)

            html(paste0(
                !!paste0("<", tag), attribs, ">",
                paste(children, collapse = ""),
                !!paste0("</", tag, ">")
            ))
        }),
        caller_env()
    )
}
```

Этот код корректно экранирует теги <p> и <b>, но пока не справляется с тегами <script>:

```
p <- tag("p")
b <- tag("b")

identical(
    p("&","and <", b("& > will be escaped")) %>%
      as.character(),
    "<p>&amp;and &lt;<b>&amp; &gt; will be escaped</b></p>"
)
#> [1] TRUE

script <- tag("script")

identical(
    script("Don't escape &, <, > - escape </script> and </style>") %>%
      as.character(),
    paste("<script>Don't escape &, <, >",
          "- escape <\\/script> and <\\/style></script>")
)
#> [1] FALSE
```

Реализуем желаемое поведение и добавим необязательный аргумент `script` в функции `escape()` и `tag()` (значение по умолчанию: `script = FALSE`).

Этот аргумент должен быть добавлен во все методы обобщенной функции `escape()`.

```
escape <- function(x, script = FALSE) UseMethod("escape")

escape.character <- function(x, script = FALSE) {

    if (script) {
        x <- gsub("</script>", "<\\/script>", x, fixed = TRUE)
        x <- gsub("</style>",  "<\\/style>",  x, fixed = TRUE)
    } else {
        x <- gsub("&", "&amp;", x)
        x <- gsub("<", "&lt;", x)
        x <- gsub(">", "&gt;", x)
    }

    html(x)
}

escape.advr_html <- function(x, script = FALSE) x

tag <- function(tag, script = FALSE) {

    new_function(
        exprs(... = ),
        expr({
            dots <- dots_partition(...)
            attribs <- html_attributes(dots$named)
            children <- map_chr(dots$unnamed, escape, script = !!script)
            html(paste0(
                !!paste0("<", tag), attribs, ">",
                paste(children, collapse = ""),
                !!paste0("</", tag, ">")
            ))
        }),
        caller_env()
    )
}
```

Наконец, создадим новые функции тегов `<p>`, `<b>`, `<script>` и `<style>` и проверим их работоспособность.

```
p <- tag("p")
b <- tag("b")

identical(
    p("&","and <", b("& > will be escaped")) %>%
      as.character(),
    "<p>&amp;and &lt;<b>&amp; &gt; will be escaped</b></p>"
)
```

```
#> [1] TRUE

script <- tag("script", script = TRUE)
style  <- tag("style" , script = TRUE)

identical(
    script("Don't escape &, <, > - escape </script> and </style>") %>%
      as.character(),
    paste("<script>Don't escape &, <, >",
          "- escape <\\/script> and <\\/style></script>")
)
#> [1] TRUE

script("Don't escape &, <, > - escape </script> and </style>")
#> <HTML> <script>Don't escape &, <, > - escape <\/script> and
#> <\/style></script>
```

2 Это упражнение потребует от нас создания фабрики функций: именованный список имен атрибутов будет расширен (за счет атрибутов `class` и `id`) и сопоставлен с аргументами функции. Значением по умолчанию для этих аргументов будет `NULL`, так что пользователь может их и не передавать.

При создании функций тегов мы воспользуемся функцией `check_dots_unnamed()` из пакета `ellipsis`, чтобы убедиться, что именованные аргументы соответствуют ожидаемым значениям (а не созданы вследствие допуска опечаток). После этого мы воспользуемся той же логикой, что и при создании фабрики функций `tag()`, описанной выше.

Чтобы сосредоточиться на ключевых идеях, мы в своем решении будем игнорировать особые случаи вроде тегов `<script>`, `<style>` и пустых тегов (несмотря на то что это приведет к некорректной работе функции для тега `<img>`).

```
tag_factory <- function(tag, tag_attrs) {
    attrs <- c("class", "id", tag_attrs)

    attr_args <- set_names(rep(list(NULL), length(attrs)), attrs)
    attr_list <- call2("list", !!!syms(set_names(attrs)))

    new_function(
        exprs(... = , !!!attr_args),
        expr({
            ellipsis::check_dots_unnamed()

            attribs <- html_attributes(compact(!!attr_list))
            dots <- compact(list(...))
            children <- map_chr(dots, escape)

            html(paste0(
                !!paste0("<", tag), attribs, ">",
                paste(children, collapse = ""),
```

```
                !!paste0("</", tag, ">")
            ))
        })
    )
}
```

Для проверки корректности нашей новой фабрики функций модифицируем пример с функцией `with_html()` из этой главы для работы с новыми функциями тегов `a()` и `img()`.

```
tag_list <- list(
    a = c("href"),
    img = c("src", "width", "height")
)

tags <- map2(names(tag_list), unname(tag_list), tag_factory) %>%
    set_names(names(tag_list))

with_tags <- function(code) {
    code <- enquo(code)
    eval_tidy(code, tags)
}

with_tags(
    a(
        img("Correct me if I am wrong", id = "second"),
        href = "https://github.com/Tazinho/Advanced-R-Solutions/issues",
        id = "first"
    )
)
#> <HTML> <a id='first'
#> href='https://github.com/Tazinho/Advanced-R-Solutions/issues'><img
#> id='second'>Correct me if I am wrong</img></a>
```

3 Для начала запустим код из этой главы для определения функции `with_html()`. Обратите внимание, что мы пропустили фрагмент кода для пустых тегов, поскольку ни один из них не встречается в этом упражнении.

```
tags <- c(
    "a", "abbr", "address", "article", "aside", "audio",
    "b", "bdi", "bdo", "blockquote", "body", "button", "canvas",
    "caption", "cite", "code", "colgroup", "data", "datalist",
    "dd", "del", "details", "dfn", "div", "dl", "dt", "em",
    "eventsource", "fieldset", "figcaption", "figure", "footer",
    "form", "h1", "h2", "h3", "h4", "h5", "h6", "head", "header",
    "hgroup", "html", "i", "iframe", "ins", "kbd", "label",
    "legend", "li", "mark", "map", "menu", "meter", "nav",
    "noscript", "object", "ol", "optgroup", "option", "output",
    "p", "pre", "progress", "q", "ruby", "rp", "rt", "s", "samp",
    "script", "section", "select", "small", "span", "strong",
```

```
    "style", "sub", "summary", "sup", "table", "tbody", "td",
    "textarea", "tfoot", "th", "thead", "time", "title", "tr",
    "u", "ul", "var", "video"
)

html_tags <- tags %>% set_names() %>% map(tag)

with_html <- function(code) {
    code <- enquo(code)
    eval_tidy(code, html_tags)
}
```

Напомним, что функция `with_html()` была представлена для вычисления функций тегов внутри списка. В противном случае определение функций тегов вроде `body()`, `source()`, `summary()` и др. в глобальном окружении привело бы к конфликту с базовыми функциями R с такими же именами. Для предотвращения этого код *языка предметной области* (*domain specific languages* – DSL), обернутый в функцию `with_html()`, вычисляется в «контексте» `html_tags`, переданного в функцию `eval_tidy()` в виде маски данных. В справке `?rlang::as_data_mask` можно прочитать следующее: *объекты в маске данных обладают более высоким приоритетом по сравнению с объектами в окружении*.

Таким образом, функция `p()` ссылается на функцию тега из `html_tags` в обоих примерах из этого упражнения. Однако поскольку `address` является не только строкой в глобальном окружении, но и функцией тега в `html_tags` (HTML-тег `<address>` может использоваться для отображения контактной информации на странице), функция `p()` во втором примере оперирует с функцией `address()`. Это ожидаемо приводит к ошибке, поскольку мы не реализовали метод `escape.function()`.

```
greeting <- "Hello!"
with_html(p(greeting))
#> <HTML> <p>Hello!</p>

p <- function() "p"
address <- "123 anywhere street"
with_html(p(address))
#> Error in UseMethod("escape"): no applicable method for 'escape' applied to an object
of class "function"
```

4 Для начала загрузим нужные нам функции из этой главы:

```
tag <- function(tag) {
    new_function(
        exprs(... = ),
        expr({
            dots <- dots_partition(...)
            attribs <- html_attributes(dots$named)
            children <- map_chr(dots$unnamed, escape)
            html(paste0(
```

```
                !!paste0("<", tag), attribs, ">",
                paste(children, collapse = ""),
                !!paste0("</", tag, ">")
            ))
        }),

        caller_env()
    )
}

void_tag <- function(tag) {
    new_function(
        exprs(... = ),
        expr({
            dots <- dots_partition(...)
            if (length(dots$unnamed) > 0) {
                stop(
                    !!paste0("<", tag, "> must not have unnamed arguments"),
                    call. = FALSE
                )
            }

            attribs <- html_attributes(dots$named)

            html(paste0(!!paste0("<", tag), attribs, " />"))
        }),
        caller_env()
    )
}

tags <- c(
    "a", "abbr", "address", "article", "aside", "audio", "b",
    "bdi", "bdo", "blockquote", "body", "button", "canvas",
    "caption", "cite", "code", "colgroup", "data", "datalist",
    "dd", "del", "details", "dfn", "div", "dl", "dt", "em",
    "eventsource", "fieldset", "figcaption", "figure", "footer",
    "form", "h1", "h2", "h3", "h4", "h5", "h6", "head", "header",
    "hgroup", "html", "i", "iframe", "ins", "kbd", "label", "legend",
    "li", "mark", "map", "menu", "meter", "nav", "noscript", "object",
    "ol", "optgroup", "option", "output", "p", "pre", "progress", "q",
    "ruby", "rp", "rt", "s", "samp", "script", "section", "select",
    "small", "span", "strong", "style", "sub", "summary", "sup",
    "table", "tbody", "td", "textarea", "tfoot", "th", "thead",
    "time", "title", "tr", "u", "ul", "var", "video"
)

void_tags <- c(
    "area", "base", "br", "col", "command", "embed", "hr", "img",
    "input", "keygen", "link", "meta", "param", "source",
    "track", "wbr"
)
```

```
html_tags <- c(
    tags %>% set_names() %>% map(tag),
    void_tags %>% set_names() %>% map(void_tag)
)

with_html <- function(code) {
    code <- enquo(code)
    eval_tidy(code, html_tags)
}

Теперь взглянем на пример, показанный выше:

with_html(
    body(
        h1("A heading", id = "first"),
        p("Some text &", b("some bold text.")),
        img(src = "myimg.png", width = 100, height = 100)
    )
)
#> <HTML> <body><h1 id='first'>A heading</h1><p>Some text &amp;<b>some
#> bold text.</b></p><img src='myimg.png' width='100' height='100'
#> /></body>
```

Как видите, разметка HTML здесь представлена в одну строку, что сильно затрудняет ее чтение и проверку на правильность.

Как бы мы хотели, чтобы выполнялось форматирование? В руководстве Google по форматированию разметки (https://google.github.io/styleguide/htmlcssguide.html#HTML_Formatting_Rules) рекомендуется делать отступ в два пробела с переходом на новую строку для всех блочных элементов, списков и таблиц. Есть в этом документе и другие рекомендации, но мы для простоты примем эти несложные правила и будем стремиться к следующему виду разметки:

```
<body>
  <h1 id='first'>A heading</h1>
  <p>Some text &amp;<b>some bold text.</b></p>
  <img src='myimg.png'width='100' height='100' />
</body>
```

Для начала внесем изменения в метод `print.advr_html()` и избавимся от вызова функции `strwrap()`, которая повторно оборачивает HTML и затрудняет понимание происходящего.

```
html <- function(x) structure(x, class = "advr_html")

print.advr_html <- function(x, ...) {
    cat(paste("<HTML>", x, sep = "\n"))
}
```

В выводе, к которому мы стремимся, можно видеть, что содержание функции body требует иного форматирования по сравнению с другими функциями тегов. Таким образом, мы создадим новую функцию format_code(), которая позволит выборочно расставлять отступы и переносы строк.

```
indent <- function(x) {
    paste0("  ", gsub("\n", "\n  ", x))
}

format_code <- function(children, indent = FALSE) {
    if (indent) {
        paste0("\n", paste0(indent(children), collapse = "\n"), "\n")
    } else {
        paste(children, collapse = "")
    }
}
```

Изменим функцию body таким образом, чтобы она использовала вспомогательную функцию format_code(). (Это можно сделать также программно в фабрике функций.)

```
html_tags$body <- function(...) {
    dots <- dots_partition(...)
    attribs <- html_attributes(dots$named)
    children <- map_chr(dots$unnamed, escape)

    html(paste0(
        "<body", attribs, ">",
        format_code(children, indent = TRUE),
        "</body>"
    ))
}
```

Итоговый вывод получился намного более приятным для глаза.

```
with_html(
    body(
        h1("A heading", id = "first"),
        p("Some text &", b("some bold text.")),
        img(src = "myimg.png", width = 100, height = 100)
    )
)
#> <HTML>
#> <body>
#>   <h1 id='first'>A heading</h1>
#>   <p>Some text &amp;<b>some bold text.</b></p>
#>   <img src='myimg.png' width='100' height='100' />
#> </body>
```

21.3.8. Ответы на упражнения

1 В настоящий момент наша функция `to_math()` генерирует следующий вывод:

```
to_math(`$`)
#> <LATEX> \mathrm{f}($)    # вместо <LATEX> \$
to_math(a$b)
#> <LATEX> \mathrm{$}(a b)  # вместо <LATEX> \mathrm{\$}(a b)
to_math(`\\`)
#> <LATEX> \mathrm{f}(\)    # вместо <LATEX> \\
to_math(`%`)
#> <LATEX> \mathrm{f}(%)    # вместо <LATEX> \%
```

Чтобы скорректировать такое поведение, нам необходима функция экранирования с методами для классов `character` и `advr_latex`. Код, уже приведенный в этой главе, мы на этот раз повторять не будем.

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

escape_latex.character <- function(x) {
    x <- gsub("^\\\\$", "\\\\\\\\", x)
    x <- gsub("^\\$$", "\\\\$", x)
    x <- gsub("^\\%$", "\\\\%", x)

    latex(x)
}

escape_latex.advr_latex <- function(x) x
```

Мы применим функцию `escape_latex()` в функции `latex_env()` при создании окружений для неизвестных символов и неизвестных функций. При этом для неизвестных функций нам сначала нужно модифицировать функцию `unknown_op()`.

```
unknown_op <- function(op) {
    new_function(
        exprs(... = ),
        expr({
            contents <- paste(..., collapse = ", ")
            paste0(
              !!paste0("\\mathrm{", escape_latex(op), "}("), contents, ")"
            )
        })
    )
}

latex_env <- function(expr) {
    calls <- all_calls(expr)
    call_list <- map(set_names(calls), unknown_op)
```

```
    call_env <- as_environment(call_list)

    # Известные функции
    f_env <- env_clone(f_env, call_env)

    # Символы по умолчанию
    names <- all_names(expr)
    symbol_env <- as_environment(set_names(escape_latex(names), names),
                                 parent = f_env)

    # Известные символы
    greek_env <- env_clone(greek_env, parent = symbol_env)
    greek_env
}
```

Теперь можно проверить работу функции `to_math()` на тех же тестовых данных.

```
to_math(`$`)
#> <LATEX> \$
to_math(a$b)
#> <LATEX> \mathrm{\$}(a b)
to_math(`\\`)
#> <LATEX> \\
to_math(`%`)
#> <LATEX> \%
```

2 Список всех поддерживаемых функций можно получить в справке `?plotmath`. Их довольно много, так что мы остановимся на следующем джентльменском наборе:

```
to_math(x %+-% y)
to_math(x %*% y)
to_math(x %->% y)
to_math(bold(x))
to_math(x != y)
```

Реализация остальных функций включает в себя механическое повторение тех же действий применительно к другим выражениям LaTex, которые можно найти в Википедии.

При решении этой задачи мы будем следовать тому, что написано в разделе, посвященном LaTeX. Это поможет сделать повествование более понятным, поскольку нам придется лишь внести несколько изменений в существующий код.

Начнем с повторения функции-преобразователя `to_math()`, созданной в этой главе.

```
to_math <- function(x) {
    expr <- enexpr(x)
```

```
    out <- eval_bare(expr, latex_env(expr))

    latex(out)
}

latex <- function(x) structure(x, class = "advr_latex")
print.advr_latex <- function(x) {
    cat("<LATEX> ", x, "\n", sep = "")
}
```

Важным нюансом здесь является то, что окружение, в котором функция `to_math()` вычисляет выражение, не является постоянным, а зависит от того, что известно об этом выражении.

Теперь реализуем функцию `latex_env()`, в которой содержится перечень всех необходимых окружений, по которым проходит функция `to_math()` при вычислении выражения.

Первое окружение содержит буквы греческого алфавита.

```
greek <- c(
    "alpha", "theta", "tau", "beta", "vartheta", "pi", "upsilon",
    "gamma", "varpi", "phi", "delta", "kappa", "rho",
    "varphi", "epsilon", "lambda", "varrho", "chi", "varepsilon",
    "mu", "sigma", "psi", "zeta", "nu", "varsigma", "omega", "eta",
    "xi", "Gamma", "Lambda", "Sigma", "Psi", "Delta", "Xi",
    "Upsilon", "Omega", "Theta", "Pi", "Phi"
)
greek_list <- set_names(paste0("\\", greek), greek)
greek_env <- as_environment(greek_list)

latex_env <- function(expr) {
    greek_env
}
```

Мы уже видели, что выражение `to_math(pi)` в нашем случае будет корректно преобразовываться в `\\pi`. Двигаемся дальше.

Теперь переходим к более технической части. Не каждый символ представляет греческую букву (и не каждая часть выражения – это символ). Для определения того, какие символы присутствуют в выражении, мы воспользуемся подходом из главы, посвященной выражениям (а именно проходом по AST в поисках всех символов), в котором мы рекурсивно пробегали по дереву и проверяли типы.

Повторим вспомогательные функции из этого раздела:

```
expr_type <- function(x) {
    if (rlang::is_syntactic_literal(x)) {
        "constant"
    } else if (is.symbol(x)) {
        "symbol"
    } else if (is.call(x)) {
```

```
        "call"
    } else if (is.pairlist(x)) {
        "pairlist"
    } else {
        typeof(x)
    }
}

switch_expr <- function(x, ...) {
    switch(expr_type(x),
           ...,
           stop("Don't know how to handle type ",
                typeof(x), call. = FALSE)
    )
}

flat_map_chr <- function(.x, .f, ...) {
    purrr::flatten_chr(purrr::map(.x, .f, ...))
}
```

Это позволило нам написать функцию `all_names()`, возвращающую все желаемые символы, преобразованные в строки.

```
all_names_rec <- function(x) {
    switch_expr(x,
                constant = character(),
                symbol =   as.character(x),
                call =     flat_map_chr(as.list(x[-1]), all_names)
    )
}

all_names <- function(x) {
    unique(all_names_rec(x))
}

all_names(expr(x + y + f(a, b, c, 10)))
#> [1] "x" "y" "a" "b" "c"
```

Теперь воспользуемся функцией `all_names()` внутри функции `latex_env()` для создания окружения символов, найденных в выражении. Это окружение станет родительским для окружения `greek_env`.

```
latex_env <- function(expr) {
    # Неизвестные символы
    names <- all_names(expr)
    symbol_env <- as_environment(set_names(names))

    # Известные символы
    env_clone(greek_env, parent = symbol_env)
}
```

В результате функция `to_math()` будет сначала конвертировать все известные греческие буквы (найденные в `greek_env`), а затем все остальные, которые останутся без изменения (в этой реализации).

Нам также необходимо добавить поддержку функций.

Для реализации поддержки всех унарных и бинарных функций в рамках окружения функции (`f_env`), которое будет добавлено к `latex_env`, мы использовали в книге две следующие вспомогательные функции.

```
unary_op <- function(left, right) {
    new_function(
        exprs(e1 = ),
        expr(
            paste0(!!left, e1, !!right)
        ),
        caller_env()
    )
}

binary_op <- function(sep) {
    new_function(
        exprs(e1 = , e2 = ),
        expr(
            paste0(e1, !!sep, e2)
        ),
        caller_env()
    )
}
```

При определении окружения `f_env` мы по большей части продолжим копировать код, уже представленный в этой главе. Но в конце мы добавим небольшой фрагмент с определением дополнительных преобразований, входящих в состав `plotmath` (и выбранных нами для решения этой задачи).

```
f_env <- child_env(
    # Бинарные операторы
    .parent = empty_env(),
    `+` = binary_op(" + "),
    `-` = binary_op(" - "),
    `*` = binary_op(" * "),
    `/` = binary_op(" / "),
    `^` = binary_op("^"),
    `[` = binary_op("_"),

    # Группировка
    `{` = unary_op("\\left{ ", " \\right}"),
    `(` = unary_op("\\left( ", " \\right)"),
    paste = paste,

    # Другие математические функции
    sqrt = unary_op("\\sqrt{", "}"),
```

```
    sin =  unary_op("\\sin(", ")"),
    log =  unary_op("\\log(", ")"),
    abs =  unary_op("\\left| ", "\\right| "),
    frac = function(a, b) {
        paste0("\\frac{", a, "}{", b, "}")
    },

    # Специальные маркеры
    hat =   unary_op("\\hat{", "}"),
    tilde = unary_op("\\tilde{", "}"),

    # Plotmath
    `%+-%` = binary_op(" \\pm "),
    `%*%`  = binary_op(" \\times "),
    `%->%` = binary_op(" \\rightarrow "),
    bold = unary_op("\\textbf{", "}"),
    `!=` = binary_op(" \\neq ")
)
```

Снова расширим функцию `latex_env()` для включения дополнительного окружения, `f_env`, которое должно быть родительским для окружения `symbol_env` (которое является родителем окружения `greek_env`).

```
latex_env <- function(expr) {
    # Известные функции
    f_env

    # Символы по умолчанию
    names <- all_names(expr)
    symbol_env <- as_environment(set_names(names), parent = f_env)

    # Известные символы
    greek_env <- env_clone(greek_env, parent = symbol_env)

    greek_env
}
```

Теперь проверим, работает ли наш новый функционал:

```
# Новый функционал plotmath
to_math(x %+-% y)
#> <LATEX> x \pm y
to_math(x %*% y)
#> <LATEX> x \times y
to_math(x %->% y)
#> <LATEX> x \rightarrow y
to_math(bold(x))
#> <LATEX> \textbf{x}
to_math(x != y)
#> <LATEX> x \neq y
```

```
# Другие примеры из главы
to_math(sin(x + pi))
#> <LATEX> \sin(x + \pi)
to_math(log(x[i]^2))
#> <LATEX> \log(x_i^2)
to_math(sin(sin))
#> <LATEX> \sin(sin)
```

Если бы мы хотели, мы могли бы включить на этом этапе и другие функции `plotmath`. Если на какой-то стадии начнут возникать конфликты имен, можно создать окружение `f_plotmath_env` для поддержки функций `plotmath`. В нашем случае мы бы также добавили его в `latex_env()` (в качестве дочернего окружения для окружения функции `f_env`).

В завершение добавим поддержку неизвестных функций. Так же, как в примере с неизвестными символами, воспользуемся приемом с рекурсивным проходом по AST.

```
all_calls_rec <- function(x) {
    switch_expr(x,
                constant = ,
                symbol =   character(),
                call = {
                    fname <- as.character(x[[1]])
                    children <- flat_map_chr(as.list(x[-1]), all_calls)
                    c(fname, children)
                }
    )
}

all_calls <- function(x) {
    unique(all_calls_rec(x))
}

all_calls(expr(f(g + b, c, d(a))))
#> [1] "f" "+" "d"

unknown_op <- function(op) {
    new_function(
        exprs(... = ),
        expr({
            contents <- paste(..., collapse = ", ")
            paste0(!!paste0("\\mathrm{", op, "}("), contents, ")")
        })
    )
}
```

Конечно, нам придется добавить новое окружение `call_env` в `latex_env()`.

```
latex_env <- function(expr) {
    calls <- all_calls(expr)
```

```
    call_list <- map(set_names(calls), unknown_op)
    call_env <- as_environment(call_list)

    # Известные функции
    f_env <- env_clone(f_env, call_env)

    # Символы по умолчанию
    names <- all_names(expr)
    symbol_env <- as_environment(set_names(names), parent = f_env)

    # Известные символы
    greek_env <- env_clone(greek_env, parent = symbol_env)
    greek_env
}
```

Наконец, заново запустим наши тесты и внимательно проверим корректность работы операторов `plotmath`.

```
# Новый функционал plotmath
to_math(x %+-% y)
#> <LATEX> x \pm y
to_math(x %*% y)
#> <LATEX> x \times y
to_math(x %->% y)
#> <LATEX> x \rightarrow y
to_math(bold(x))
#> <LATEX> \textbf{x}
to_math(x != y)
#> <LATEX> x \neq y

# Другие примеры из главы
to_math(sin(x + pi))
#> <LATEX> \sin(x + \pi)
to_math(log(x[i]^2))
#> <LATEX> \log(x_i^2)
to_math(sin(sin))
#> <LATEX> \sin(sin)

# Неизвестные функции
to_math(f(g(x)))
#> <LATEX> \mathrm{f}(\mathrm{g}(x))
```

Ответы на упражнения из главы 23

23.2.4. Ответы на упражнения

1 Мы ожидаем, что функция `f()` создаст вектор (`x`) длины `n`, который затем будет удален, в результате чего функция вернет `NULL`. При профилировании

этой функции она выполняется слишком быстро, чтобы вернуть осмысленный результат.

```
profvis::profvis(f())
#> Error in parse_rprof(prof_output, expr_source): No parsing data available. Maybe your
function was too fast?
```

Установка параметра `torture = TRUE` позволяет запускать сборщик мусора после каждой операции выделения памяти, что может быть полезно при более точном профилировании памяти.

```
profvis::profvis(f(), torture = TRUE)
```

К нашему удивлению, в таком виде функция `f()` будет выполняться очень долго. В чем может быть дело?

Последуем совету из этого упражнения и рассмотрим исходный код функции `rm()`:

```
function (..., list = character(), pos = -1,
          envir = as.environment(pos),
          inherits = FALSE)
{
    dots <- match.call(expand.dots = FALSE)$...
    if (
        length(dots) && !all(
            vapply(dots, function(x) is.symbol(x) ||
                   is.character(x), NA, USE.NAMES = FALSE)
        )
    )
        stop("... must contain names or character strings")
    names <- vapply(dots, as.character, "")
    if (length(names) == 0L)
          names <- character()
    list <- .Primitive("c")(list, names)
    .Internal(remove(list, envir, inherits))
}
```

Оказывается, функция `rm()` выполняет довольно много работы для получения имени объекта, который нужно удалить, поскольку опирается на принципы *нестандартного вычисления* (non-standard evaluation – NSE).

Мы можем значительно упростить работу функции `rm()`, если воспользуемся ее аргументом `list`:

```
f2 <- function(n = 1e5) {
    x <- rep(1, n)
    rm(list = "x")
}
profvis::profvis(f2(), torture = TRUE)
```

К сожалению, выполнение функции по-прежнему происходит довольно медленно, и мы буквально застреваем в процессе профилирования.

Кстати, один из нас дождался-таки завершения процесса профилирования этой функции в одной из старых версий R. Но значимого результата он все равно не получил.

Так что этот вопрос так и остался для нас без ответа. И для Хэдли тоже.

23.3.3. Ответы на упражнения

1 Сначала протестируем оба выражения при помощи функции `bench::mark()` и убедимся, что средние значения не выводятся (поскольку на них обычно большое влияние оказывают выбросы).

```
n <- 1e6
x <- runif(100)

bench_df <- bench::mark(
    sqrt(x),
    x ^ 0.5,
    iterations = n
)

bench_df
#> # A tibble: 2 x 6
#>   expression      min   median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
#> 1 sqrt(x)       888ns   1.04µs   679098.      848B    10.9
#> 2 x^0.5        3.82µs   4.12µs   219102.      848B     7.01
```

Нам нужно получить доступ к исходным данным, чтобы мы могли сравнить результаты обоих подходов к оценке.

```
t1_bench <- mean(unlist(bench_df[1, "time"]))
t2_bench <- mean(unlist(bench_df[2, "time"]))

t1_systime <- system.time(for (i in 1:n) sqrt(x)) / n
t2_systime <- system.time(for (i in 1:n) x ^ 0.5) / n
```

Мы видим, что полученные результаты находятся в пределах одного порядка. Можно предположить, что результаты функции `bench::mark()` являются более правильными из-за использования таймера высокой точности. Также в подходе с `system.time()` могут присутствовать накладные расходы, связанные с циклом `for`.

```
# Сравнение результатов
t1_systime["elapsed"]
#>  elapsed
#> 1.24e-06
t1_bench
```

```
#> [1] 1.88e-06

t2_systime["elapsed"]
#>  elapsed
#> 4.18e-06
t2_bench
#> [1] 5e-06
```

Если вы хотите узнать отличия между временами `"user"`, `"system"` и `"elapsed"`, обратитесь к справке `?proc.time`.

2 Для сравнения этих подходов можно воспользоваться функцией `bench::mark()` с сортировкой по медианному времени выполнения.

```
x <- runif(100)

bm <- bench::mark(
    sqrt(x),
    x^0.5,
    x^(1 / 2),
    exp(log(x) / 2)
)

bm[order(bm$median), ]
#> # A tibble: 4 x 6
#>   expression         min   median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr>    <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
#> 1 sqrt(x)          908ns    1.1µs   656785.      848B      0
#> 2 exp(log(x)/2)   2.97µs   3.22µs   295051.      848B     29.5
#> 3 x^0.5           3.94µs   4.23µs   213837.      848B      0
#> 4 x^(1/2)         4.08µs   4.45µs   204753.      848B      0
```

Как и можно было предположить, реализованная в языке примитивная функция `sqrt()` оказалась самой быстрой. Выражение `exp(log(x) / 2)`, построенное на основе двух разных примитивных функций, заняло второе место, хоть и с приличным отставанием. Два других выражения оказались более медленными: `x ^ 0.5` опережает по скорости `x ^ (1 / 2)`, поскольку `0.5` требует меньших вычислений по сравнению с `(1 / 2)`.

Ответы на упражнения из главы 24

24.3.1. Ответы на упражнения

1 В специальном разделе CRAN для высокоэффективных вычислений (https://cran.rstudio.com/web/views/HighPerformanceComputing.html) есть сразу несколько рекомендаций по этому поводу. Конкретно нас интересует секция

Large memory and out-of-memory data, в которой, среди прочих, описываются функции `biglm::biglm()`, `speedglm::speedlm()` и `RcppEigen::fastLm()`.

На небольших наборах данных эти функции не дают серьезной прибавки в скорости (а какие-то выполняются даже медленнее):

```
penguins <- palmerpenguins::penguins

bench::mark(
    "lm" = lm(
        body_mass_g ~ bill_length_mm + species, data = penguins
    ) %>% coef(),
    "biglm" = biglm::biglm(
        body_mass_g ~ bill_length_mm + species, data = penguins
    ) %>% coef(),
    "speedglm" = speedglm::speedlm(
        body_mass_g ~ bill_length_mm + species, data = penguins
    ) %>% coef(),
    "fastLm" = RcppEigen::fastLm(
        body_mass_g ~ bill_length_mm + species, data = penguins
    ) %>% coef()
)
#> # A tibble: 4 x 6
#>   expression      min   median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
#> 1 lm           1.24ms   1.34ms      749.  908.48KB     4.13
#> 2 biglm      940.12µs  992.4µs      989.    5.43MB     6.40
#> 3 speedglm     1.6ms   1.68ms      589.   62.43MB     2.02
#> 4 fastLm     972.62µs   1.01ms      967.    3.42MB     2.03
```

Для более объемных данных выбор подходящего метода имеет большее значение:

```
eps <- rnorm(100000)
x1 <- rnorm(100000, 5, 3)
x2 <- rep(c("a", "b"), 50000)
y <- 7 * x1 + (x2 == "a") + eps
td <- data.frame(y = y, x1 = x1, x2 = x2, eps = eps)

bench::mark(
    "lm" = lm(y ~ x1 + x2, data = td) %>% coef(),
    "biglm" = biglm::biglm(y ~ x1 + x2, data = td) %>% coef(),
    "speedglm" = speedglm::speedlm(y ~ x1 + x2, data = td) %>% coef(),
    "fastLm" = RcppEigen::fastLm(y ~ x1 + x2, data = td) %>% coef()
)
#> # A tibble: 4 x 6
#>   expression      min   median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
#> 1 lm           46.3ms   48.6ms      13.4      27MB     17.3
#> 2 biglm        40.4ms   43.5ms      17.2    22.2MB     21.0
#> 3 speedglm     38.1ms   41.1ms      23.9    22.9MB     19.9
#> 4 fastLm       66.3ms     75ms      13.7    30.2MB     15.6
```

Для получения еще большего прироста скорости вы могли бы установить библиотеку линейной алгебры, оптимизированную для вашей системы (см. `?speedglm::speedlm`).

> *Функции класса* `speedlm` *способны ускорить процесс подгонки линейных моделей к большим наборам данных. Особенно высокая эффективность может быть достигнута, если ваш R привязан к оптимизированной библиотеке линейной алгебры BLAS, такой как ATLAS.*

Совет: если ваш набор данных хранится в базе данных, вы могли бы ознакомиться с пакетом `modeldb` (https://github.com/tidymodels/modeldb), позволяющим выполнять вычисления, связанные с линейными моделями, на стороне соответствующей базы данных.

2 Поиск в интернете приводит нас к пакету `fastmatch`. Воспользуемся им и увидим немалую прибавку в скорости в сравнении с функцией `base::match()`.

```
table <- 1:100000
x <- sample(table, 10000, replace = TRUE)

bench::mark(
    match = match(x, table),
    fastmatch = fastmatch::fmatch(x, table)
)
#> # A tibble: 2 x 6
#>   expression      min   median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
#> 1 match        18.9ms   19.5ms      51.3    1.46MB     2.05
#> 2 fastmatch   552.9µs  594.6µs    1671.   442.91KB     2.04
```

3 Обычно для этих целей в R используется обобщенная функция `as.POSIXct()`, дающая на выходе объект даты и времени класса `POSIXct` с целочисленным типом.

```
date_ct <- as.POSIXct("2020-01-01 12:30:25")
date_ct
#> [1] "2020-01-01 12:30:25 UTC"
```

Под капотом функция `as.POSIXct()` пользуется услугами функции `as.POSIXlt()` для преобразования символьных данных. Эта функция возвращает объект даты и времени класса `POSIXlt` с типом `list`.

```
date_lt <- as.POSIXlt("2020-01-01 12:30:25")
date_lt
#> [1] "2020-01-01 12:30:25 UTC"
```

Класс `POSIXlt` обладает преимуществом хранения отдельных компонентов в виде атрибутов. Это позволяет извлекать требующиеся нам компоненты с помощью обычных операторов для работы со списками.

```
attributes(date_lt)
#> $names
#> [1] "sec"   "min"   "hour"  "mday"  "mon"   "year"  "wday"  "yday"  "isdst"
#>
#> $class
#> [1] "POSIXlt" "POSIXt"
#>
#> $tzone
#> [1] "UTC"
date_lt$sec
#> [1] 25
```

Хотя списки могут приносить большую практическую пользу, однако вычисления на основе целых чисел зачастую выполняются быстрее и расходуют меньше памяти.

```
date_lt2 <- rep(date_lt, 10000)
date_ct2 <- rep(date_ct, 10000)

bench::mark(
    date_lt2 - date_lt2,
    date_ct2 - date_ct2,
    date_ct2 - date_lt2
)
#> # A tibble: 3 x 6
#>   expression               min   median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr>          <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
#> 1 date_lt2 - date_lt2   4.27ms   4.41ms      226.    1.14MB    11.5
#> 2 date_ct2 - date_ct2  89.48µs 177.61µs     5613.  195.45KB    24.9
#> 3 date_ct2 - date_lt2    2.1ms   2.29ms      437.  664.67KB     7.37
```

Функция `as.POSIXlt()`, в свою очередь, использует функцию `strptime()`, которая на выходе дает идентичный объект.

```
date_str <- strptime("2020-01-01 12:30:25",
                     format = "%Y-%m-%d %H:%M:%S")
identical(date_lt, date_str)
#> [1] TRUE
```

Функции `as.POSIXct()` и `as.POSIXlt()` по умолчанию принимают разный формат символьного ввода (например, `"2001-01-01  12:30"` или `"2001/1/1 12:30"`). Функция `strptime()` требует задания формата явным образом, а взамен дает высокую скорость вычислений.

```
bench::mark(
    as.POSIXct = as.POSIXct("2020-01-01 12:30:25"),
    as.POSIXct_format = as.POSIXct("2020-01-01 12:30:25",
        format = "%Y-%m-%d %H:%M:%S"
    ),
    strptime_fomat = strptime("2020-01-01 12:30:25",
```

```
        format = "%Y-%m-%d %H:%M:%S"
    )
)[1:3]
#> # A tibble: 3 x 3
#>   expression             min   median
#>   <bch:expr>        <bch:tm> <bch:tm>
#> 1 as.POSIXct         95.69µs  100.4µs
#> 2 as.POSIXct_format  30.37µs   33.4µs
#> 3 strptime_fomat      9.29µs   10.3µs
```

Четвертый способ состоит в использовании функций преобразования из пакета `lubridate`, среди которых есть функции-обертки с интуитивно понятным синтаксисом. Однако их использование ведет к снижению быстродействия.

```
library(lubridate)
ymd_hms("2013-07-24 23:55:26")
#> [1] "2013-07-24 23:55:26 UTC"

bench::mark(
    as.POSIXct = as.POSIXct("2013-07-24 23:55:26", tz = "UTC"),
    ymd_hms = ymd_hms("2013-07-24 23:55:26")
)[1:3]
#> # A tibble: 2 x 3
#>   expression      min   median
#>   <bch:expr> <bch:tm> <bch:tm>
#> 1 as.POSIXct  92.31µs  96.99µs
#> 2 ymd_hms      4.46ms   4.63ms
```

При необходимости дополнительные способы конвертирования строковых данных в даты можно обнаружить в пакетах `chron`, `anytime` и `fasttime`. В пакете `chron` представлены новые классы, а время хранится в виде долей от дня с использованием чисел двойной точности. При этом в данном пакете не поддерживается работа с часовыми поясами и практика перехода на летнее время. В пакете `anytime` заявляется о возможности преобразования «чего угодно к типу `POSIXct` или `Date`». Пакет `fasttime` содержит только одну функцию `fastPOSIXct()`.

4 Скользящее среднее представляет собой очень полезную метрику для сглаживания временных рядов, пространственных и иных типов данных. Ширина *скользящего окна* (rolling window) обычно задает степень сглаживания и количество отсутствующих значений по обе стороны анализируемого диапазона.

Функционал расчета скользящего среднего присутствует в разных пакетах, а отличия состоят в скорости и гибкости производимых вычислений. Ниже приведены примеры использования разных функций, служащих одной цели.

```
x <- 1:10
slider::slide_dbl(x, mean, .before = 1, .complete = TRUE)
```

```
#>  [1]  NA 1.5 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5

bench::mark(
    caTools = caTools::runmean(x, k = 2, endrule = "NA"),
    data.table = data.table::frollmean(x, 2),
    RcppRoll = RcppRoll::roll_mean(x, n = 2, fill = NA,
                                   align = "right"),
    slider = slider::slide_dbl(x, mean, .before = 1, .complete = TRUE),
    TTR = TTR::SMA(x, 2),
    zoo_apply = zoo::rollapply(x, 2, mean, fill = NA, align = "right"),
    zoo_rollmean = zoo::rollmean(x, 2, fill = NA, align = "right")
)
#> # A tibble: 7 x 6
#>   expression        min   median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr>   <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
#> 1 caTools        89.6µs   95.2µs    10414.  167.64KB     4.10
#> 2 data.table     49.9µs   55.4µs    17435.    1.42MB     6.23
#> 3 RcppRoll       47.5µs   51.1µs    19372.   39.23KB     4.08
#> 4 slider         76.2µs   84.7µs    11701.        0B     4.07
#> 5 TTR             470µs  489.6µs     2034.    1.18MB     2.02
#> 6 zoo_apply     389.4µs  421.5µs     2344.  430.22KB     4.07
#> 7 zoo_rollmean  334.6µs  370.8µs     2672.    6.42KB     4.08
```

Также вы можете обратиться к примеру из первого оригинального издания книги, в котором демонстрируется создание функции расчета скользящего среднего.

5 Согласно описанию (см. справку `?optim`), функция `optim()` реализует следующее:

> *Универсальная оптимизация на основе метода Нелдера–Мида, квазиньютоновских методов и метода сопряженных градиентов. Включает в себя возможность выполнения оптимизации с ограничениями и применение алгоритма имитации отжига.*

Функция `optim()` позволяет оптимизировать функцию (`fn`) на заданном интервале с помощью указанного метода (`method = c("Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN", "Brent")`). В документации к этой функции приведено множество полезных примеров ее использования. В простейшем случае мы можем задать для функции `optim()` начальное значение `par = 0` для расчета минимума квадратного многочлена:

```
optim(0, function(x) x^2 - 100 * x + 50,
    method = "Brent",
    lower = -1e20, upper = 1e20
)
#> $par
#> [1] 50
#>
#> $value
```

```
#> [1] -2450
#>
#> $counts
#> function gradient
#>       NA       NA
#>
#> $convergence
#> [1] 0
#>
#> $message
#> NULL
```

Поскольку мы решаем задачу одномерной оптимизации, с той же целью можно было бы воспользоваться функцией `stats::optimize()`.

```
optimize(function(x) x^2 - 100 * x + 50, c(-1e20, 1e20))
#> $minimum
#> [1] 50
#>
#> $objective
#> [1] -2450
```

В общем случае выбираемый подход тесно связан с типом оптимизации, которую вы хотите выполнить. В разделе *CRAN task view on optimisation and mathematical modelling* (https://cran.r-project.org/web/views/Optimization.html) есть множество полезных примеров:

- в пакете `optimx` представлено расширение функции `optim()` с тем же синтаксисом, но более богатым выбором методов;
- пакет `RcppNumerical` объединяет в себе сразу несколько библиотек с открытым исходным кодом для осуществления числовых вычислений (написанных на C++) и интегрирует их с R посредством пакета `Rcpp`;
- в пакете `DEoptim` представлен глобальный оптимизатор на основе метода дифференциальной эволюции.

24.4.3. Ответы на упражнения

1 При исследовании исходного кода функции `rowSums()` можно заметить, что она реализована в виде обертки функции `.rowSums()` с дополнительной проверкой входных данных, а также преобразованием и обработкой комплексных чисел.

```
rowSums
#> function (x, na.rm = FALSE, dims = 1L)
#> {
#>     if (is.data.frame(x))
#>         x <- as.matrix(x)
#>     if (!is.array(x) || length(dn <- dim(x)) < 2L)
#>         stop("'x' must be an array of at least two dimensions")
```

```
#>     if (dims < 1L || dims > length(dn) - 1L)
#>         stop("invalid 'dims'")
#>     p <- prod(dn[-(id <- seq_len(dims))])
#>     dn <- dn[id]
#>     z <- if (is.complex(x))
#>         .Internal(rowSums(Re(x), prod(dn), p, na.rm)) + (0+1i) *
#>             .Internal(rowSums(Im(x), prod(dn), p, na.rm))
#>     else .Internal(rowSums(x, prod(dn), p, na.rm))
#>     if (length(dn) > 1L) {
#>         dim(z) <- dn
#>         dimnames(z) <- dimnames(x)[id]
#>     }
#>     else names(z) <- dimnames(x)[[1L]]
#>     z
#> }
#> <bytecode: 0x7fdc052ed7b0>
#> <environment: namespace:base>
```

Функция `.rowSums()` вызывает внутреннюю функцию, встроенную в интерпретатор R. Эти скомпилированные функции могут работать очень быстро.

```
.rowSums
#> function (x, m, n, na.rm = FALSE)
#> .Internal(rowSums(x, m, n, na.rm))
#> <bytecode: 0x7fdc05a1fe40>
#> <environment: namespace:base>
```

При этом поскольку результаты тестирования показывают практически одинаковое время выполнения, мы бы предпочли более безопасный вариант.

```
m <- matrix(rnorm(1e6), nrow = 1000)

bench::mark(
    rowSums(m),
    .rowSums(m, 1000, 1000)
)
#> # A tibble: 2 x 6
#>   expression                   min   median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr>              <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
#> 1 rowSums(m)                1.67ms   1.79ms      559.    7.86KB        0
#> 2 .rowSums(m, 1000, 1000)   1.62ms   1.77ms      565.    7.86KB        0
```

2 Попробуем ускорить реализацию функции `chisq.test()` путем уменьшения объема выполняемых действий.

```
chisq.test2 <- function(x, y) {
    m <- rbind(x, y)
    margin1 <- rowSums(m)
    margin2 <- colSums(m)
```

```
    n <- sum(m)
    me <- tcrossprod(margin1, margin2) / n

    x_stat <- sum((m - me)^2 / me)
    df <- (length(margin1) - 1) * (length(margin2) - 1)
    p.value <- pchisq(x_stat, df = df, lower.tail = FALSE)

    list(x_stat = x_stat, df = df, p.value = p.value)
}
```

Проверим, что наша реализация возвращает те же значения, после чего запустим процедуру тестирования.

```
a <- 21:25
b <- seq(21, 29, 2)
m <- cbind(a, b)

chisq.test(m) %>% print(digits=5)
#>
#>  Pearson's Chi-squared test
#>
#> data:  m
#> X-squared = 0.162, df = 4, p-value = 1
chisq.test2(a, b)
#> $x_stat
#> [1] 0.162
#>
#> $df
#> [1] 4
#>
#> $p.value
#> [1] 0.997

bench::mark(
    chisq.test(m),
    chisq.test2(a, b),
    check = FALSE
)
#> # A tibble: 2 x 6
#>   expression             min   median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr>        <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
#> 1 chisq.test(m)      106.2µs    113µs     8686.        0B     2.03
#> 2 chisq.test2(a, b)   31.8µs   34.1µs    28932.        0B     5.79
```

3 При анализе исходного кода функции `table()` нам необходимо убрать все ненужное и извлечь только важные строительные блоки. Мы видим, что в основе функции `table()` лежит предельно эффективная функция `tabulate()`. Таким образом, наша задача сводится к выполнению предварительной обработки с максимально возможной скоростью.

Для начала вычислим измерения и имена будущей таблицы. После этого воспользуемся функцией `fastmatch::fmatch()` для сопоставления элементов каждого вектора с позициями внутри векторов (т. е. минимальное значение сопоставляется с `1L`, следующее – с `2L` и т. д.). Следуя логике, заложенной в функции `table()`, мы комбинируем и сдвигаем эти значения для сопоставления пар значений в наших данных с индексами в итоговой таблице. После этого функция `tabulate()` подсчитывает значения и возвращает целочисленный вектор с количествами по каждой позиции в таблице. В заключение мы воспользуемся кодом из функции `table()`, чтобы присвоить правильные измерение и класс.

```
table2 <- function(a, b){

    a_s <- sort(unique(a))
    b_s <- sort(unique(b))

    a_l <- length(a_s)
    b_l <- length(b_s)

    dims <- c(a_l, b_l)
    pr <- a_l * b_l
    dn <- list(a = a_s, b = b_s)

    bin <- fastmatch::fmatch(a, a_s) +
        a_l * fastmatch::fmatch(b, b_s) - a_l
    y <- tabulate(bin, pr)

    y <- array(y, dim = dims, dimnames = dn)
    class(y) <- "table"

    y
}

a <- sample(100, 10000, TRUE)
b <- sample(100, 10000, TRUE)

bench::mark(
    table(a, b),
    table2(a, b)
)
#> # A tibble: 2 x 6
#>   expression        min   median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr>   <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
#> 1 table(a, b)    1.74ms    1.9ms      528.    1.19MB     17.0
#> 2 table2(a, b) 637.01µs    821µs     1198.  694.48KB     16.2
```

Поскольку мы не использовали функцию `table()` в нашей реализации функции `chisq.test2()`, то не сможем ускорить ее выполнение за счет перехода на более быструю функцию `table2()`.

24.5.1. Ответы на упражнения

1 С интерфейсом этих функций можно ознакомиться в справке `?dnorm`:

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)
```

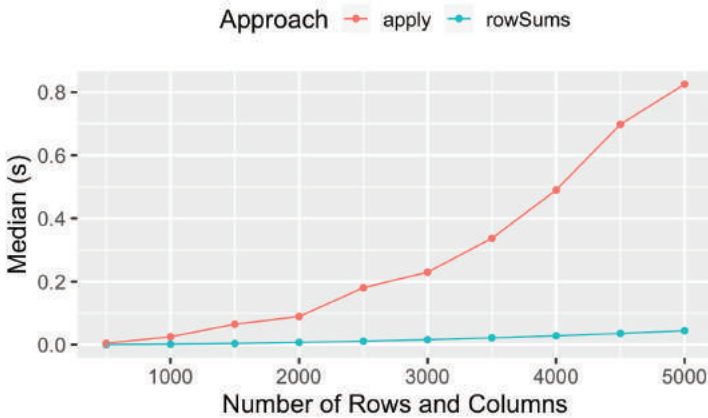
Данные функции векторизованы по своим числовым аргументам, включающим первый аргумент (`x`, `q`, `p`, `n`), а также аргументы `mean` и `sd`.

Вызов функции `rnorm(10, mean = 10:1)` генерирует случайные числа из разных нормальных распределений. Эти распределения отличаются своими средними значениями: в первом среднее значение будет равно 10, во втором – 9, в третьем – 8 и т. д.

2 Сравним скорость двух функций на примере квадратных матриц увеличивающегося размера:

```
rowsums <- bench::press(
    p = seq(500, 5000, length.out = 10),
    {
        mat <- tcrossprod(rnorm(p), rnorm(p))
        bench::mark(
          rowSums = rowSums(mat),
          apply = apply(mat, 1, sum)
        )
    }
)
#> Running with:
#>        p
#>  1   500
#>  2  1000
#>  3  1500
#>  4  2000
#>  5  2500
#>  6  3000
#>  7  3500
#>  8  4000
#>  9  4500
#> 10  5000

library(ggplot2)

rowsums %>%
    summary() %>%
    dplyr::mutate(Approach = as.character(expression)) %>%
    ggplot(
      aes(p, median, color = Approach, group = Approach)) +
    geom_point() +
```

```
    geom_line() +
    labs(x = "Number of Rows and Columns",
         y = "Median (s)") +
    theme(legend.position = "top")
```



Как видите, для матриц небольшого размера разница в скорости не так велика, но она становится более заметной при увеличении объема матрицы. Функция `apply()` представляет собой довольно универсальный инструмент, но в ней не применяется векторизация для повышения быстродействия, и она не настолько оптимизирована, как `rowSums()`.

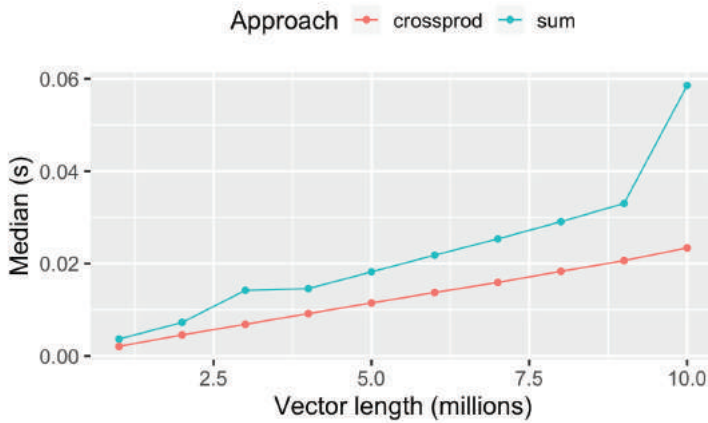
3 Мы можем передать векторы функции `crossprod()`, которая преобразует их в строку и колонку и перемножит. Результатом будет скалярное произведение, что и соответствует средневзвешенному.

```
x <- rnorm(10)
w <- rnorm(10)
all.equal(sum(x * w), crossprod(x, w)[[1]])
#> [1] TRUE
```

Проверка быстродействия обоих методов на векторах разной длины показывает, что вариант с использованием функции `crossprod()` почти вдвое опережает выражение `sum(x * w)`.

```
weightedsum <- bench::press(
    n = 1:10,
    {
        x <- rnorm(n * 1e6)
        bench::mark(
            sum = sum(x * x),
            crossprod = crossprod(x, x)[[1]]
        )
    }
```

```
)
#> Running with:
#>        n
#>  1     1
#>  2     2
#>  3     3
#>  4     4
#>  5     5
#>  6     6
#>  7     7
#>  8     8
#>  9     9
#> 10    10

weightedsum %>%
    summary() %>%
    dplyr::mutate(Approach = as.character(expression)) %>%
    ggplot(aes(n, median, color = Approach, group = Approach)) +
    geom_point() +
    geom_line() +
    labs(x = "Vector length (millions)",
         y = "Median (s)") +
    theme(legend.position = "top")
```



Ответы на упражнения из главы 25

25.2.6. Ответы на упражнения

1

- `f1: mean();`
- `f2: cumsum();`

- `f3: any();`
- `f4: Position();`
- `f5: pmin()`.

2 Что ж, давайте портируем эти функции в C++:

- `all()`:

```
bool allC(LogicalVector x) {
    int n = x.size();

    for (int i = 0; i < n; ++i) {
        if (!x[i]) return false;
    }
    return true;
}
```

- `cumprod()`, `cummin()`, `cummax()`:

```
NumericVector cumprodC(NumericVector x) {
    int n = x.size();
    NumericVector out(n);

    out[0] = x[0];
    for (int i = 1; i < n; ++i) {
        out[i]  = out[i - 1] * x[i];
    }
    return out;
}

NumericVector cumminC(NumericVector x) {
    int n = x.size();
    NumericVector out(n);

    out[0] = x[0];
    for (int i = 1; i < n; ++i) {
        out[i]  = std::min(out[i - 1], x[i]);
    }
    return out;
}

NumericVector cummaxC(NumericVector x) {
    int n = x.size();
    NumericVector out(n);

    out[0] = x[0];
    for (int i = 1; i < n; ++i) {
        out[i]  = std::max(out[i - 1], x[i]);
    }
    return out;
}
```

- `diff()` (начнем с предположения о единичном интервале (аргумент `lag`), после чего обобщим функцию для интервала `n`):

```
NumericVector diffC(NumericVector x) {
    int n = x.size();
    NumericVector out(n - 1);

    for (int i = 1; i < n; i++) {
        out[i - 1] = x[i] - x[i - 1];
    }
    return out ;
}

NumericVector difflagC(NumericVector x, int lag = 1) {
    int n = x.size();

    if (lag >= n) stop("`lag` must be less than `length(x)`.");

    NumericVector out(n - lag);

    for (int i = lag; i < n; i++) {
        out[i - lag] = x[i] - x[i - lag];
    }
    return out;
}
```

- `range()`:

```
NumericVector rangeC(NumericVector x) {
    double omin = x[0], omax = x[0];
    int n = x.size();

    if (n == 0) stop("`length(x)` must be greater than 0.");

    for (int i = 1; i < n; i++) {
        omin = std::min(x[i], omin);
        omax = std::max(x[i], omax);
    }

    NumericVector out(2);
    out[0] = omin;
    out[1] = omax;
    return out;
}
```

- `var()`:

```
double varC(NumericVector x) {
    int n = x.size();
```

```
    if (n < 2) {
        return NA_REAL;
    }

    double mx = 0;
    for (int i = 0; i < n; ++i) {
        mx += x[i] / n;
    }

    double out = 0;
    for (int i = 0; i < n; ++i) {
        out += pow(x[i] - mx, 2);
    }

    return out / (n - 1);
}
```

25.4.5. Ответы на упражнения

1 В этом упражнении мы начнем с написания функции `minC()` и расширения ее для корректной работы с пропущенными значениями. Аргумент `na_rm` предназначен как раз для обработки значений `NA`. Если вектор `x` содержит исключительно значения `NA`, функция `minC()` должна вернуть `Inf` для `na_rm = TRUE`. Для возвращаемых значений мы использовали векторный тип данных, чтобы избежать лишних преобразований типов.

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector minC(NumericVector x, bool na_rm = false) {
    int n = x.size();
    NumericVector out = NumericVector::create(R_PosInf);

    if (na_rm) {
        for (int i = 0; i < n; ++i) {
            if (x[i] == NA_REAL) {
                continue;
            }
            if (x[i] < out[0]) {
                out[0] = x[i];
            }
        }
    } else {
        for (int i = 0; i < n; ++i) {
            if (NumericVector::is_na(x[i])) {
                out[0] = NA_REAL;
                return out;
            }
```

```
            if (x[i] < out[0]) {
                out[0] = x[i];
            }
        }
    }

    return out;
}

minC(c(2:4, NA))
#> [1] NA
minC(c(2:4, NA), na_rm = TRUE)
#> [1] 2
minC(c(NA, NA), na_rm = TRUE)
#> [1] Inf
```

Мы также расширим функцию `anyC()`, чтобы она могла работать с пропущенными значениями. Заметьте, что мы (снова) не побрезговали использовать дублирующийся код. Во избежание этого мы могли бы разместить проверку на пропущенные значения во внутреннем цикле, жертвуя небольшой долей производительности. В качестве типа возвращаемого значения мы использовали `LogicalVector`. Если бы вместо этого мы воспользовались типом `bool`, значения C++ `NA_LOGICAL` были бы сконвертированы в `TRUE`.

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
LogicalVector anyC(LogicalVector x, bool na_rm = false) {
    int n = x.size();
    LogicalVector out = LogicalVector::create(false);

    if (na_rm == false) {
        for (int i = 0; i < n; ++i) {
            if (LogicalVector::is_na(x[i])) {
                out[0] = NA_LOGICAL;
                return out;
            } else {
                if (x[i]) {
                    out[0] = true;
                }
            }
        }
    }

    if (na_rm) {
        for (int i = 0; i < n; ++i) {
            if (LogicalVector::is_na(x[i])) {
                continue;
            }
```

```
            if (x[i]) {
                out[0] = true;
                return out;
            }
        }
    }

    return out;
}

anyC(c(NA, TRUE))  # any(c(NA, TRUE)) в этом случае вернул бы TRUE
#> [1] NA
anyC(c(NA, TRUE), na_rm = TRUE)
#> [1] TRUE
```

2 Наша версия функции `cumsumC()`, предусматривающая появление значений `NA`, будет возвращать вектор той же длины, что и `x`. По умолчанию (`na_rm = FALSE`) все значения после первого вхождения `NA` будут установлены в `NA`, поскольку они зависят от неизвестного пропущенного значения. Если `na_rm = TRUE`, пропущенные значения будут восприниматься как нули.

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector cumsumC(NumericVector x, bool na_rm = false) {
    int n = x.size();
    NumericVector out(n);
    LogicalVector is_missing = is_na(x);

    if (!na_rm) {
        out[0] = x[0];
        for (int i = 1; i < n; ++i) {
            if (is_missing[i - 1]) {
                out[i] = NA_REAL;
            } else{
                out[i] = out[i - 1] + x[i];
            }
        }
    }

    if (na_rm) {
        if (is_missing[0]) {
            out[0] = 0;
        } else {
            out[0] = x[0];
        }
        for (int i = 1; i < n; ++i) {
            if (is_missing[i]) {
                out[i] = out[i-1] + 0;
```

```
            } else {
                out[i] = out[i-1] + x[i];
            }
        }
    }

    return out;
}

cumsumC(c(1, NA, 2, 4))
#> [1]  1 NA NA NA
cumsumC(c(1, NA, 2, 4), na_rm = TRUE)
#> [1] 1 1 3 7
```

Реализация функции `diffC()` будет возвращать вектор `NA` длины `length(x) - lag`, если во входном векторе будут присутствовать пропущенные значения. Если установить `na_rm = TRUE`, будет возвращать `NA` для каждого различия в присутствии как минимум одного значения `NA`, переданного на вход.

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector diffC(NumericVector x, int lag = 1,
                    bool na_rm = false) {
    int n = x.size();

    if (lag >= n) stop("`lag` must be less than `length(x)`.");

    NumericVector out(n - lag);

    for (int i = lag; i < n; i++) {
        if (NumericVector::is_na(x[i]) ||
            NumericVector::is_na(x[i - lag])) {
            if (!na_rm) {
                return rep(NumericVector::create(NA_REAL), n - lag);
            }
            out[i - lag] = NA_REAL;
            continue;
        }
        out[i - lag] = x[i] - x[i - lag];
    }

    return out;
}

diffC(c(1, 3, NA, 10))
#> [1] NA NA NA
diffC(c(1, 3, NA, 10), na_rm = TRUE)
#> [1] 2 NA NA
```

25.5.7. Ответы на упражнения

1 Реализация расчета медианы производится по-разному для векторов четной и нечетной длины, что мы учли в нашей функции.

```
#include <algorithm>
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double medianC(NumericVector x) {
    int n = x.size();

    if (n % 2 == 0) {
        std::partial_sort (x.begin(), x.begin() + n / 2 + 1, x.end());
        return (x[n / 2 - 1] + x[n / 2]) / 2;
    } else {
        std::partial_sort (x.begin(), x.begin() + (n + 1) / 2, x.end());
        return x[(n + 1) / 2 - 1];
    }
}
```

2 Воспользуемся методом `find()` и циклом по `unordered_set`, пока не найдем совпадение или не завершим проход.

```
#include <Rcpp.h>
#include <unordered_set>
using namespace Rcpp;

// [[Rcpp::export]]
LogicalVector inC(CharacterVector x, CharacterVector table) {
    std::unordered_set<String> seen;
    seen.insert(table.begin(), table.end());

    int n = x.size();
    LogicalVector out(n);
    for (int i = 0; i < n; ++i) {
        out[i] = seen.find(x[i]) != seen.end();
    }

    return out;
}
```

3 В данной реализации мы будем добавлять элементы в вектор только в случае, если они в нем уже не присутствуют.

```
#include <Rcpp.h>
#include <unordered_set>
using namespace Rcpp;
```

```
// [[Rcpp::export]]
NumericVector uniqueC(NumericVector x) {
    std::unordered_set<int> seen;
    int n = x.size();

    std::vector<double> out;
    for (int i = 0; i < n; ++i) {
        if (seen.insert(x[i]).second) out.push_back(x[i]);
    }

    return wrap(out);
}


// В одну строку
// [[Rcpp::export]]
std::unordered_set<double> uniqueCC(NumericVector x) {
    return std::unordered_set<double>(x.begin(), x.end());
}
```

4 Реализуем функцию `min()` с помощью итераций по вектору и рекурсивного сравнения каждого значения с текущим минимальным значением.

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double minC(NumericVector x) {
    int n = x.size();
    double out = x[0];

    for (int i = 0; i < n; i++) {
        out = std::min(out, x[i]);
    }

    return out;
}
```

5 Для реализации функции `which.min()` мы сначала воспользуемся функцией `min_element`, после чего рассчитаем дистанцию до найденного элемента, ведя отсчет с начала вектора.

```
#include <Rcpp.h>
#include <algorithm>
#include <iterator>
using namespace Rcpp;

// [[Rcpp::export]]
double which_minC(NumericVector x) {
    int out = std::distance(
```

```
        x.begin(), std::min_element(x.begin(), x.end())
    );

    return out + 1;
}
```

6 Структура всех трех функций будет очень похожа.

Сначала мы сортируем входящий вектор. Затем применяем соответствующую функцию (`set_union`, `set_intersection` или `set_difference`). После этого результат будет находиться между итераторами `tmp.begin()` и `out_end`. Для извлечения результата мы проходим итерациями по диапазону между `tmp.begin()` и `out_end` в конце каждой функции.

В базовом R операции для работы над множествами исключают появление дубликатов в аргументах. Здесь мы будем реализовывать это с помощью дополнительного шага, на котором будут опускаться значения, соответствующие своему предшественнику. В случае с симметричными функциями множеств `unionC` и `intersectC` этот шаг будет применяться к итоговому вектору. В функции `setdiffC` мы будем обрабатывать первый входящий вектор.

```
#include <Rcpp.h>
#include <unordered_set>
#include <algorithm>
using namespace Rcpp;

// [[Rcpp::plugins(cpp11)]]
// [[Rcpp::export]]
IntegerVector unionC(IntegerVector x, IntegerVector y) {
    int nx = x.size();
    int ny = y.size();

    IntegerVector tmp(nx + ny);

    std::sort(x.begin(), x.end()); // unique
    std::sort(y.begin(), y.end());

    IntegerVector::iterator out_end = std::set_union(
        x.begin(), x.end(), y.begin(), y.end(), tmp.begin()
    );

    int prev_value = 0;
    IntegerVector out;
    for (IntegerVector::iterator it = tmp.begin();
         it != out_end; ++it) {
        if ((it != tmp.begin())  && (prev_value == *it)) continue;

        out.push_back(*it);

        prev_value = *it;
    }
```

```
    return out;
}

// [[Rcpp::export]]
IntegerVector intersectC(IntegerVector x, IntegerVector y) {
    int nx = x.size();
    int ny = y.size();

    IntegerVector tmp(std::min(nx, ny));

    std::sort(x.begin(), x.end());
    std::sort(y.begin(), y.end());

    IntegerVector::iterator out_end = std::set_intersection(
        x.begin(), x.end(), y.begin(), y.end(), tmp.begin()
    );

    int prev_value = 0;
    IntegerVector out;
    for (IntegerVector::iterator it = tmp.begin();
         it != out_end; ++it) {
        if ((it != tmp.begin()) && (prev_value == *it)) continue;

        out.push_back(*it);

        prev_value = *it;
    }

    return out;
}

// [[Rcpp::export]]
IntegerVector setdiffC(IntegerVector x, IntegerVector y) {
    int nx = x.size();
    int ny = y.size();

    IntegerVector tmp(nx);

    std::sort(x.begin(), x.end());

    int prev_value = 0;
    IntegerVector x_dedup;
    for (IntegerVector::iterator it = x.begin();
         it != x.end(); ++it) {
        if ((it != x.begin()) && (prev_value == *it)) continue;

        x_dedup.push_back(*it);

        prev_value = *it;
    }
```

```
    std::sort(y.begin(), y.end());

    IntegerVector::iterator out_end = std::set_difference(
        x_dedup.begin(), x_dedup.end(), y.begin(), y.end(), tmp.begin()
    );

    IntegerVector out;
    for (IntegerVector::iterator it = tmp.begin();
         it != out_end; ++it) {
        out.push_back(*it);
    }

    return out;
}
```

Убедимся, что наши функции работают так, как и ожидалось.

```
# Входные векторы, включающие дубликаты
x <- c(1, 2, 3, 3, 3)
y <- c(3, 3, 2, 5)

union(x, y)
#> [1] 1 2 3 5
unionC(x, y)
#> [1] 1 2 3 5

intersect(x, y)
#> [1] 2 3
intersectC(x, y)
#> [1] 2 3

setdiff(x, y)
#> [1] 1
setdiffC(x, y)
#> [1] 1
```

Библиография

Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1996[1].

Stefan Milton Bache and Hadley Wickham. *magrittr: A forward-pipe operator for R*, 2014. URL http://magrittr.tidyverse.org/.

James Balamuta. *errorist: Automatically Search Errors or Warnings*, 2018a. URL https://github.com/coatless/errorist.

James Balamuta. *searcher: Query Search Interfaces*, 2018b. URL https://github.com/coatless/searcher.

Douglas Bates and Martin Maechler. *Matrix: Sparse and dense matrix classes and methods*, 2018. URL https://CRAN.R-project.org/package=Matrix.

Alan Bawden. *Quasiquotation in Lisp*. InPEPM '99, pages 4–12, 1999. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.309.227.

Henrik Bengtsson. *The R.oo package - object-oriented programming with references using standard R code*. In Kurt Hornik, Friedrich Leisch, and Achim Zeileis, editors, *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)*, Vienna, Austria, March 2003. URL: https://www.r-project.org/conferences/DSC-2003/Proceedings/Bengtsson.pdf.

Christopher Brown. *hash: Full feature implementation of hash/associated arrays/dictionaries*, 2013. URL https://CRAN.R-project.org/package=hash.

Carlos Bueno. *Mature Optimization Handbook*. 2013. URL http://carlos.bueno.org/optimization.

John M Chambers. *Programming with Data: A Guide to the S Language*. Springer, 1998.

John M Chambers. *Software for Data Analysis: Programming with R*. Springer, 2008.

John M Chambers. *Object-oriented programming, functional programming and R*. Statistical Science, 29 (2): 167–180, 2014. URL https://projecteuclid.org/download/pdfview_1/euclid.ss/1408368569.

John M Chambers. *Extending R*. CRC Press, 2016.

John M Chambers and Trevor J Hastie. *Statistical Models in S*. Wadsworth & Brooks/Cole Advanced Books & Software, 1992.

Winston Chang. *R6: Classes with reference semantics*, 2017. URL https://r6.r-lib.org.

Dirk Eddelbuettel and Romain François. *Rcpp: Seamless R and C++ integration. Journal of Statistical Software*, 40 (8): 1–18, 2011. doi: 10.18637/jss.v040.i08. URL http://www.jstatsoft.org/v40/i08.

[1] Книга издана в России: https://kdu.ru/node/198.

Martin Fowler. *Domain-specific Languages*. Pearson Education, 2010. URL https://www.amazon.com/Domain-Specific-Languages-Addison-Wesley-Signature-Fowler/dp/0321712943.

Garrett Grolemund and Hadley Wickham. *Dates and times made easy with lubridate. Journal of Statistical Software*, 40 (3): 1–25, 2011. URL http://www.jstatsoft.org/v40/i03.

Gabor Grothendieck, Louis Kates, and Thomas Petzoldt. *proto: prototype object-based programming*, 2016. URL https://CRAN.R-project.org/package=proto.

Lionel Henry and Hadley Wickham. *rlang: tools for low-level R programming*, 2018b. URL https://rlang.r-lib.org.

Jim Hester. *bench: high precision timing of R expressions*, 2018. URL http://bench.r-lib.org.

Jim Hester, Kirill Müller, Kevin Ushey, Hadley Wickham, and Winston Chang. *withr: Run code with temporarily modified global state*, 2018. URL http://withr.r-lib.org.

Andrew Hunt and David Thomas. *The Pragmatic Programmer*. Addison Wesley, 1990[1].

Thomas Lumley. *Programmer's niche: Macros in R*. R News, 1 (3): 11–13, 2001. URL https://www.r-project.org/doc/Rnews/Rnews_2001-3.pdf.

Norman Matloff. *The Art of R Programming*. No Starch Press, 2011[2].

Norman Matloff. *Parallel Computing for Data Science*. Chapman & Hall/CRC, 2015. URL https://www.amazon.com/Parallel-Computing-Data-Science-Examples/dp/1466587016.

Q. Ethan McCallum and Steve Weston. *Parallel R*. O'Reilly, 2011. URL https://www.amazon.com/Parallel-Data-Analysis-Distributed-World/dp/1449309925.

Scott Meyers. *Effective STL: 50 specific ways to improve your use of the standard template library*. Pearson Education, 2001. URL https://www.amazon.com/Effective-STL-Specific-Standard-Template/dp/0201749629.

Scott Meyers. *Effective C++: 55 specific ways to improve your programs and designs*. Pearson Education, 2005. URL https://www.amazon.com/Effective-Specific-Improve-Programs-Designs/dp/0321334876.

Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. *Evaluating the design of the R language. InEuropean Conference on Object-Oriented Programming, pages 104–131*. Springer, 2012.

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

Kirill Müller and Hadley Wickham. *tibble: simple data frames*, 2018. URL http://tibble.tidyverse.org.

---

[1] Книга издана в России: http://www.williamspublishing.com/Books/978-5-907203-32-7.html.

[2] Книга издана в России: https://www.piter.com/collection/all/product/iskusstvo-programmirovaniya-na-r-pogruzhenie-v-bolshie-dannye

R Core Team. Writing R extensions. R Foundation for Statistical Computing, 2018a. URL https://cran.r-project.org/doc/manuals/r-devel/R-exts.html.

R Core Team. *R internals*. *R Foundation for Statistical Computing*, 2018b. URL https://cran.r-project.org/doc/manuals/r-devel/R-ints.html.

Steven S Skiena. *The Algorithm Design Manual*. Springer, 1998. URL https://www.amazon.com/Algorithm-Design-Manual-Steven-Skiena/dp/1849967202.

Nathan Teetor. *zeallot: multiple, unpacking, and destructuring assignment*, 2018. URL https://CRAN.R-project.org/package=zeallot.

Luke Tierney and Riad Jarjour. *proftools: Profile Output Processing Tools for R*, 2016. URL https://CRAN.R-project.org/package=proftools.

Peter Van-Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT press, 2004.

Hadley Wickham. *mutatr: mutable objects for R. Computational Statistics*, 26 (3): 405–418, 2011. doi: 10.1007/s00180-011-0235-7.

Hadley Wickham. *forcats: tools for working with categorical variables*, 2018. URL http://forcats.tidyverse.org.

Hadley Wickham and Yihui Xie. *evaluate: Parsing and Evaluation Tools that Provide More Details than the Default*, 2018. URL https://github.com/r-lib/evaluate.

Hadley Wickham, Jim Hester, Kirill Müller, and Daniel Cook. *memoise: Memoisation of Functions*, 2018. URL https://github.com/r-lib/memoise.

Предметный указатель

D

E

F

G

S

T

U

V

W

А

Б

В

Г

Д

З

И

К

Л

М

Н

О

П

Р